Broadview
www.broadview.com.cn

《基于J2EE的Ajax宝典》第二版

疯狂Ajax讲义

——Prototype/jQuery+DWR+Spring+Hibernate整合开发

李刚 编著

疯狂源自梦想

技术成就辉煌

疯狂源自梦想

技术成就辉煌

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

疯狂Ajax讲义

—Prototype/jQuery+DWR+Spring+Hibernate整合开发

李刚 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是《基于 J2EE 的 Ajax 宝典》的第二版。《基于 J2EE 的 Ajax 宝典》面市近 2 年，作为 Ajax 领域最全面、实用的图书，一直深受读者的好评。

全书主要分为三个部分。第一部分介绍了 XHTML、CSS、JavaScript 和 DOM 编程等内容。第二部分详细介绍了 Prototype、jQuery、DWR、AjaxTags 等四个最常用的 Ajax 框架的用法，并针对每个框架提供了一个实用案例。这两个部分是笔者在“疯狂 Java 实训营”的培训讲义，是本书的重点部分。第三部分则提供了 2 个综合性案例：Blog 系统和电子拍卖系统，让读者将前面所学真正应用到实际项目中。

本书绝大部分章节后都提供了相应的编程习题，供开发者巩固所学，将理论融入实际开发之用。关于这些编程习题的解题思路和参考答案可登录 http://www.crazyjava.org 获取。

本书是疯狂 Java 体系丛书之一，前 8 章基本以 XHTML、JavaScript 和 DOM 编程为主，无须任何基础即可阅读；第 9 章以后的内容则需要掌握 Spring、Hibernate 等 Java EE 知识，建议先认真阅读疯狂 Java 体系的《轻量级 Java EE 企业应用实战》一书。

图书在版编目（CIP）数据

疯狂 Ajax 讲义——Prototype/jQuery+DWR+Spring+Hibernate 整合开发/李刚编著.—北京：电子工业出版社，2009.4
ISBN 978-7-121-08440-9

Ⅰ. 疯… Ⅱ.李… Ⅲ.①计算机网络－程序设计②JAVA 语言－程序设计 Ⅳ.TP393.09 TP312

中国版本图书馆 CIP 数据核字（2009）第 030104 号

责任编辑：朱沭红
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：850×1168　1/16　印张：39.5　字数：1127 千字
印　　次：2009 年 4 月第 1 次印刷
印　　数：4000 册　定价：69.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

Ajax 技术已经不再是新技术，它已经成为企业开发中应用最广泛的技术之一，不管采用什么样的开发平台：Java EE 也好，.NET 也好，PHP 也好，Ruby on Rails 也好，只要开发 B/S 架构的应用，那么表现层就一定会使用 Ajax 技术。

Ajax 技术采用异步方式发送请求，避免了每个请求对应一个页面的模式，允许在一个页面发送多个请求，从而可以更大程度地利用已下载的页面，服务器每次响应生成的只是必需的数据，无须响应生成整个页面。对用户而言，发送异步请求不会阻塞当前的浏览器线程，浏览器可以继续下一步操作：比如继续浏览或再次发送异步请求。因此用户将不会处于等待状态，而是感觉自己一直与应用处于交互状态，从而带给了用户连续的体验。

Ajax 技术是 Web 2.0 的重要技术之一，互连网上各种 Blog 系统、RSS，以及 Wiki 系统和 SNS 交友网络等，都大量使用了 Ajax 技术。

Ajax 技术还催生了大量的网页游戏。国内的很多游戏运营商纷纷推出了自己的网页游戏。网页游戏具有无须下载、安装，即开即玩、简单便捷的特征，尤其对办公室上班族具有较大的吸引力，因此也具有很好的市场前景。在这种网页游戏中，每个网页上都包含了大量制作精美的图片，当游戏玩家单击、双击这些图片时，系统将采用 Ajax 技术与远程服务器通信，这是绝大部分网页游戏的底层运行机制。

2007 年出版的《基于 J2EE 的 Ajax 宝典》具有全面、专业的特征，书中不仅深入介绍了 Ajax 编程的底层原理和技术，还全面介绍了 Prototype、Dojo、DWR、JSON-RPC-Java 和 AjaxTags 等 5 个 Ajax 框架。不过，Dojo 版本更新太快，这一点限制了它在实际企业开发中的应用。本书作为《基于 J2EE 的 Ajax 宝典》的第二版，详细介绍了 XHTML、CSS、JavaScript、DOM 和 JavaScript 事件机制等基础知识，重点分析了 XMLHttpRequest 对象的运行机制和运行原理。Ajax 框架的介绍部分，Prototype、DWR、AjaxTags 等框架升级到了最新版本，另外新增介绍了 jQuery 框架。

本书的创作感言

写一本书真的很累！每次一本书写到最后几章时，都会有一种近似虚脱的感觉。此外，如果还遭遇一些来自外界的困扰，就让人更加难以静下来做事。不过，笔者现在的主要职业是培训，需要不断地面对新的学生，有责任引导他们进入软件开发行业，这大概是支持自己继续写下去的一个动力吧。

最初，笔者写书仅仅是为了作为笔者的培训教材，帮助自己的学生能更好地理解自己所讲授的内容，所以总是尽量使用清晰条理的方式来组织内容，用实用、易操作的实例来演示开发，用通俗易懂的语言进行表达。希望把实际企业开发中解决问题的方法，用通俗、简单的语言告诉学生。

在本书创作过程中，笔者一度感到非常困惑：其实笔者这些书的学术价值真的很少，因为基本上没有什么创新，绝大部分都是前人的智慧。充其量，笔者只是进行了再归纳、总结，于是难免感到意兴阑珊。

无聊中和一位美籍华人（一位资深 CTO）在 Skype 上聊天，他告诉笔者：印度的程序员数量大概是中国程序员的 100 倍，因为印度有大量的程序员基数，所以就孕育出了大量优秀的程序员。中国人，虽然是世界上非常聪明的种族之一，但中国程序员太少了，所以中国产生的优秀程序员也很少。你写的书虽然没有什么创新，但只要能把实际软件开发的方法和经验传播开来，让更多的年轻人走进软件开发行业，你就为中国的软件开发业做出了贡献。如果有一天中国拥有 1 亿以上的软件开发工程师，那中国软件行业就真正发展起来了。

那天之后，萧索的心情开始慢慢好转，毕竟自己还在做一件“切实爱国”的事情。笔者心想：爱

国，不是光喊口号的事情，而是需要埋头做事的。

本书有什么特点

不知道是否有人仔细研究过笔者写的书，书中的长句是很少的——因为很多语句自己都会反复地调整，有兴趣的读者可以仔细体会一下这个特点。

《基于 J2EE 的 Ajax 宝典》上市一年半了，其间收到不少读者来信，对书中内容也提了一些自己的见解。此外，该书也一直作为“疯狂 Java 实训营”的讲义，这些对本书的升级起到了很大的帮助。

此外，本书还有如下特点：

1. 通俗易懂，适合自学

该书第一版作为培训教材近 2 年了，在吸收大量学习者的学习体会和心得的基础上，本书重点讲解了学习过程中难以理解和掌握的知识点，降低了学习者的学习难度。

2. 知识丰富，内容全面

正如该书的第一版，书中知识非常全面：XHTML、CSS、JavaScript、DOM、Event 机制、XMLHttpRequest、Prototype 库、jQuery、DWR、AjaxTags 等 Ajax 知识的相关内容，都可在本书中找到详细的讲解。

3. 深入实用，实践性强

本书并不是一本 Ajax 的入门图书，本书将 Ajax 技术融入轻量级 Java EE 开发，深入介绍了 Ajax+Java EE 整合开发的方法和步骤，对实际企业开发具有极好的指导意义。

衷心感谢

本书创作过程中有一些小事情一度影响了笔者的心情，让笔者产生过心灰意懒、意兴阑珊的感觉，使得写作也一度中断。在此，要衷心感谢电子工业出版社的朋友，以及疯狂 Java 联盟的杨恩雄、heyitang、petrelsky5 等广大网友和所有给我鼓励的朋友，是你们的支持让我再度燃起创作热情。

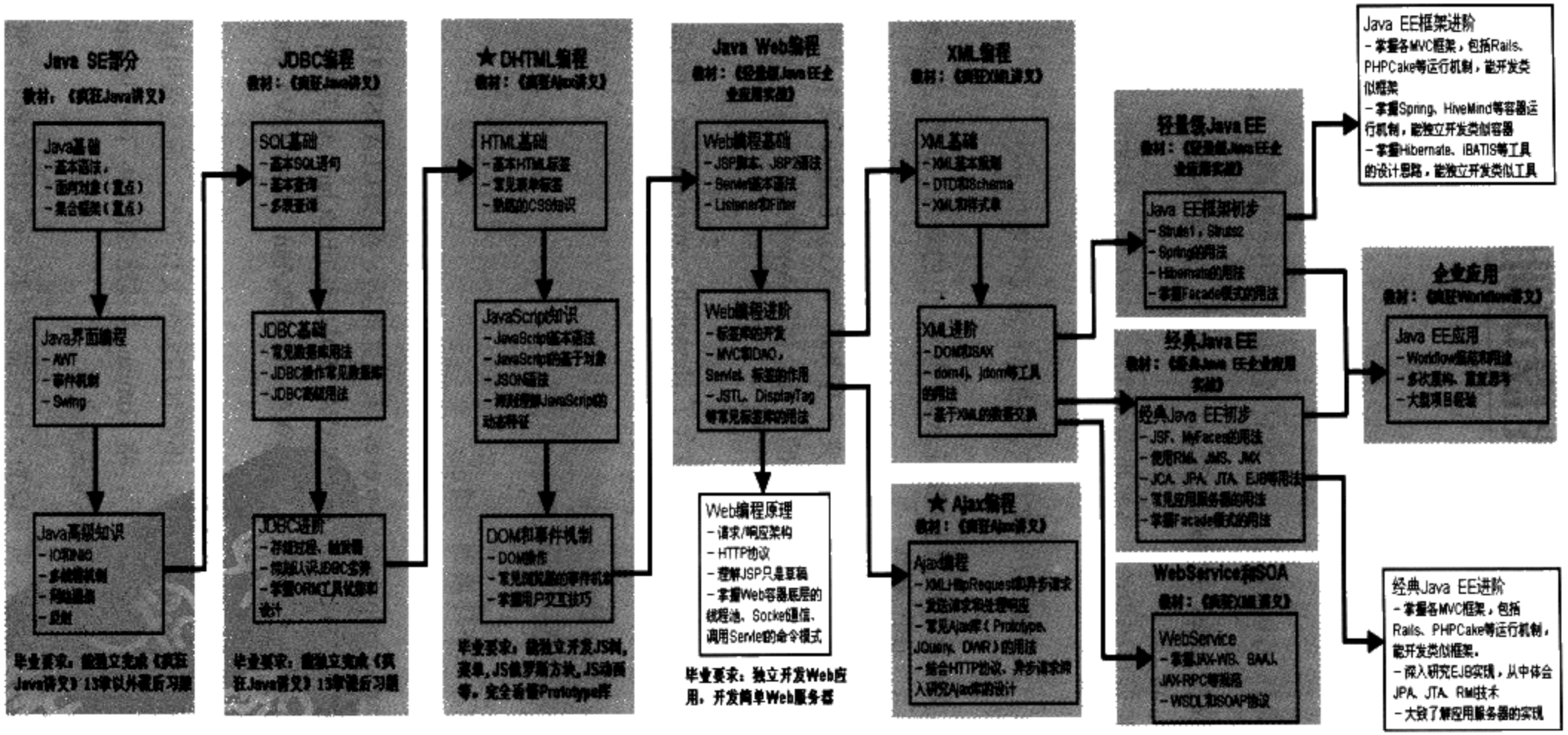
本书写给谁看

本书是疯狂 Java 体系丛书之一，前半部分（前 8 章）没有基础即可阅读，后半部分则需要一定的 Spring、Hibernate 等 Java EE 基础。如果读者只希望掌握 JavaScript 编程、DHTML 和 Ajax 基础，则无须任何基础；如果读者希望将 Ajax 融入实际的 Java EE 开发，则建议先阅读《轻量级 Java EE 企业应用实战》一书。

2008 年 3 月 17 日

疯狂 Java 学习路线图

笔者就自己对 Java EE 体系的理解，对 Java EE 学习者给出一个粗略线路图：



备注：

1.没有灰色覆盖的区域稍有难度，请谨慎尝试。

2.本学习线路图不涉及设计模式、软件方法学等概念，但希望大家能从开发中悟道。

3.本人并不认为 Spring、Hibernate 很复杂，只要基础扎实，掌握框架是水到渠成的。

光盘说明

一、光盘内容

本光盘是《疯狂 Ajax 讲义》一书的配书光盘，书中的代码按章、按节存放，即第 2 章、第 2 节所使用的代码放在 codes 文件夹的 02\2.2 文件夹下，依次类推。

另：书中每份源代码也给出与光盘源文件的对应关系，方便读者查找。

本光盘 codes 目录下有 17 个文件夹,其内容和含义说明如下：

（1）01～17 个文件夹名对应于《疯狂 Ajax 讲义》中的章名，即第二章所使用的代码放在 codes 文件夹的 02 文件夹下，依次类推。

（2）其中 10、12、14、16、17 文件夹下有 Xxx 和 Xxx_Eclipse 两个文件夹，它们是同一个项目的源文件，其中 Xxx 是 IDE 平台无关的项目，使用 Ant 来编译即可；而 Xxx_Eclipse 是该项目在 Eclipse IDE 工具中的项目文件。

（3）codes 文件夹下大量文件夹下包含了.classpath、.mymetadata、.project、.springBeans 等文件，它们是 Eclipse 项目文件，请不要删除。

二、运行环境

本书中的程序在以下环境调试通过：

（1）安装 jdk-6u6-windows-i586-p.exe，安装完成后，添加 CLASSPATH 环境变量，该环境变量的值为;%JAVA_HOME%/lib/tools.jar;;%JAVA_HOME%/lib/dt.jar。如果为了可以编译和运行 Java 程序，还应该在 PATH 环境变量中增加%JAVA_HOME%/bin。其中 JAVA_HOME 代表 JDK（不是 JRE）的安装路径。

（2）安装 Apache 的 Tomcat6.0.16，不要使用安装文件安装，而是采用解压缩的安装方式。安装 Tomcat 请参看第一章。安装完成后，将 Tomcat 安装路径的 lib 下的 jsp-api.jar 和 servlet-api.jar 两个 JAR 文件添加到 CLASSPATH 环境变量之后。

（3）安装 apache-ant-1.7.0。将下载的 Ant 压缩文件解压缩到任意路径，然后增加 ANT_HOME 的环境变量，让变量的值为 Ant 的解压缩路径。并在 PATH 环境变量中增加%ANT_HOME%/bin 环境变量。

（4）安装 MySQL5.0 或更高版本，安装 MysSQL 时候选择 GBK 的编码方式。

（5）安装 Eclipse3.3，并安装 MyEclipse6.0 的插件。

三、注意事项

（1）独立应用程序的代码中都包括 build.xml 文件，在 Dos 或 Shell 下进入 build.xml 文件所在路径，执行如下命令：

ant build -- 编译程序

ant run --运行程序

（2）对于 Web 应用，将该应用复制到%TOMCAT_HOME%/webapps 路径下，然后进入 build.xml 所在路径，执行如下命令：

ant build -- 编译应用

启动 Tomcat 服务器，使用浏览器即可访问该应用。

（3）对于 Eclipse 项目文件，导入 Eclipse 开发工具即可。

（4）代码中有大量代码需要连接数据库，读者应修改数据库 URL 以及用户名、密码让这些代码与读者运行环境一致。如果项目下有 SQL 脚本，导入 SQL 脚本即可，如果没有 SQL 脚本，系统将在运行时自动建表，读者只需创建对应数据库即可。

（5）在使用本光盘的程序时，请将程序拷贝到硬盘上，并去除文件的只读属性。

（6）本书绝大部分章节后都提供了相应的编程习题，供开发者巩固所学，将理论融入实际开发之用。关于这些编程习题的解题思路和参考答案可登录 http://www.crazyjava.org 获取。

四、技术支持

如果您使用本光盘中遇到什么问题，您可以登录如下网站与我们联系：

网站：http://www.crazyjava.org

或发 Email 到 kongyeeku@163.com

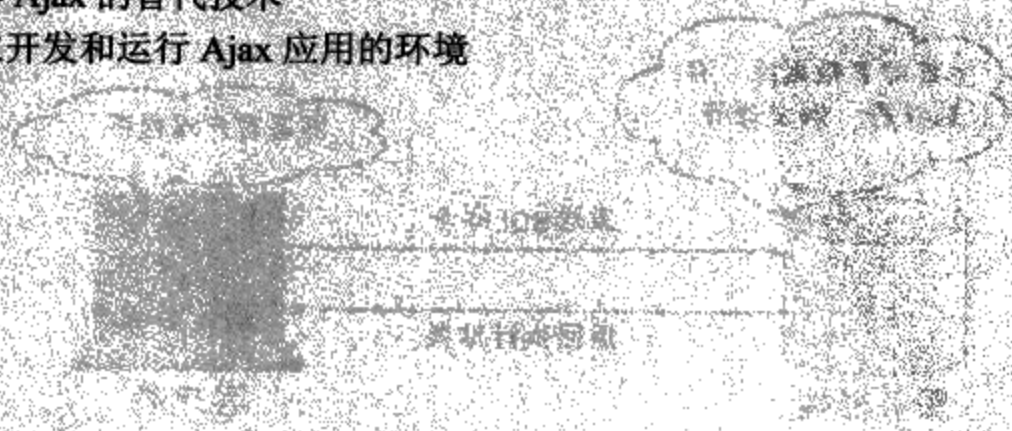
目录 CONTENTS

CHAPTER 1

第 1 章
Ajax 概述

本章要点

- C/S 模式应用的结构和缺点
- B/S 模式应用的结构和优势
- 传统 Web 应用的不足
- 如何改进传统的 Web 应用
- RIA 的改进和优势
- Ajax 的基础
- Ajax 的基本特征
- Ajax 依赖的核心技术
- 了解 Ajax 的替代技术
- 建立开发和运行 Ajax 应用的环境



Ajax（Asynchronous JavaScript And XML，异步 JavaScript 和 XML）是个相当新的名词，它在 2005 年由 Jesse James Garrett 首先提出。在接下来的极短时间内，Ajax 被广泛应用到大量 B/S 结构的应用中，改进了传统的 Web 应用，给浏览者一种更连续的体验。Ajax 的最大优势在于异步交互，即浏览者在浏览页面时，可同时向服务器发送请求，甚至可以不用等待前一次请求得到完全响应，便再次发送请求。这种异步请求的方式，非常类似于传统的桌面应用。通过使用 Ajax 技术，可以使互联网网页具有更友好的人机交互和更美观的浏览界面。

使用 Ajax 的异步请求方式，浏览器无须频繁地重新加载新页面，服务器的响应不再是整个页面内容，而只是必须更新的部分数据。Ajax 可以减轻服务器和带宽的负担，提供更好的服务响应。使用 Ajax 的异步模式，浏览器无须重新加载整个页面，就可以显示新的数据。浏览器通过 JavaScript 代码向服务器发送请求，JavaScript 代码负责解析服务器的响应数据，并把样式表加到数据上，然后在现有网页中显示出来。

Ajax 技术给互联网带来了一场革命——Web 2.0，而且它也正是这场革命中的核心技术。到目前为止，已很难找到一个没有使用 Ajax 技术的 Web 应用。Ajax 技术甚至催生了一种新的网络游戏平台：网页游戏——游戏玩家无须下载任何客户端，直接打开网页就可开始游戏。

1.1 重新思考 Web 应用

传统的 Web 应用经过多年的发展，在很多方面都是相当完善的。特别是 Java EE、.NET、Ruby on Rails 等平台的出现，更是规范了 Web 应用的开发。Ajax 的出现，让人不得不重新思考传统的 Web 应用。Ajax 给浏览者一种全新的体验：浏览者可以无须等待服务器响应，而多次以异步方式向服务器发送请求。这种体验方式，非常类似于传统的桌面应用。Ajax 并不是要颠覆传统的 B/S 结构的应用，而只是让 B/S 结构的应用更加完善。

➤➤1.1.1 应用系统的发展史

早期应用软件系统大都采用 C/S（客户机/服务器模式）结构，C/S 结构的软件分为客户机和服务器两层。客户机不是毫无运算能力的输入/输出设备，在客户端需要部署大量的应用程序，而且可能还具有一定的数据存储能力。

C/S 结构应用的服务器端通常主要安装数据库管理系统，当然也可能包含一些业务逻辑实现（这些业务逻辑实现通常以函数、存储过程和触发器的形式存在）。通过把软件系统的计算和数据合理地分配在客户机和服务器两端，可以有效地降低网络通信量和服务器运算量。

C/S 结构应用的结构图如图 1.1 所示。

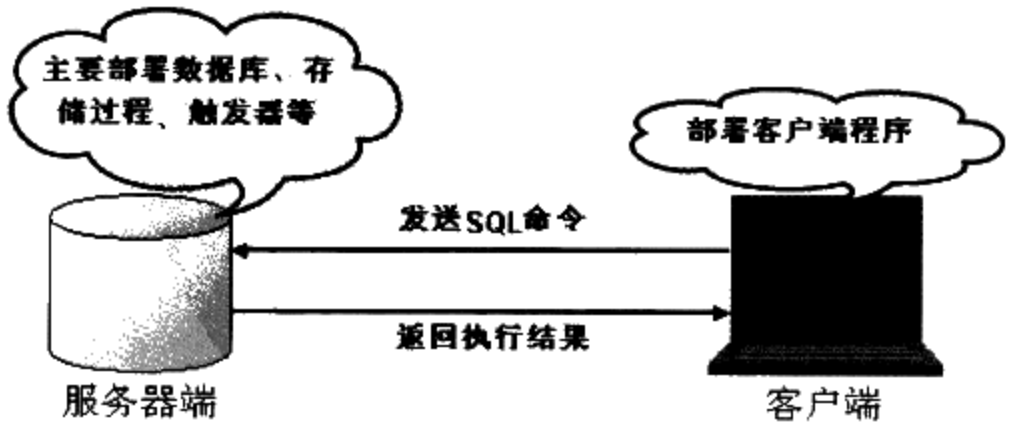


图 1.1 C/S 结构应用

对于 C/S 结构的应用而言，因为可以直接在客户端部署应用程序，所以可以让应用的人机交互界面更加友好，并可充分美化应用程序的人机界面。但由于服务器连接个数和数据通信量的限制，这种结构的软件适于在用户数目不多的局域网内使用。早期的大部分 ERP 软件产品即属于此类结构。

随着 Internet 技术的兴起，B/S（浏览器/服务器模式）结构得到了大规模应用。B/S 结构是对 C/S 结构的一种改进。在这种结构下，应用的业务逻辑完全在应用服务器端实现，用户表现完全在 Web 服务器上实现，客户端只需要浏览器即可进行业务处理，是一种全新的软件系统结构技术。这种结构是当今应用软件的首选体系结构。

在这种应用结构下，客户端的所有处理请求都以 HTTP 请求形式发送，而服务器端则将响应以 HTML 页面的形式送回客户端，由客户端浏览器负责显示 HTML 页面。B/S 结构应用的结构图如图 1.2 所示。

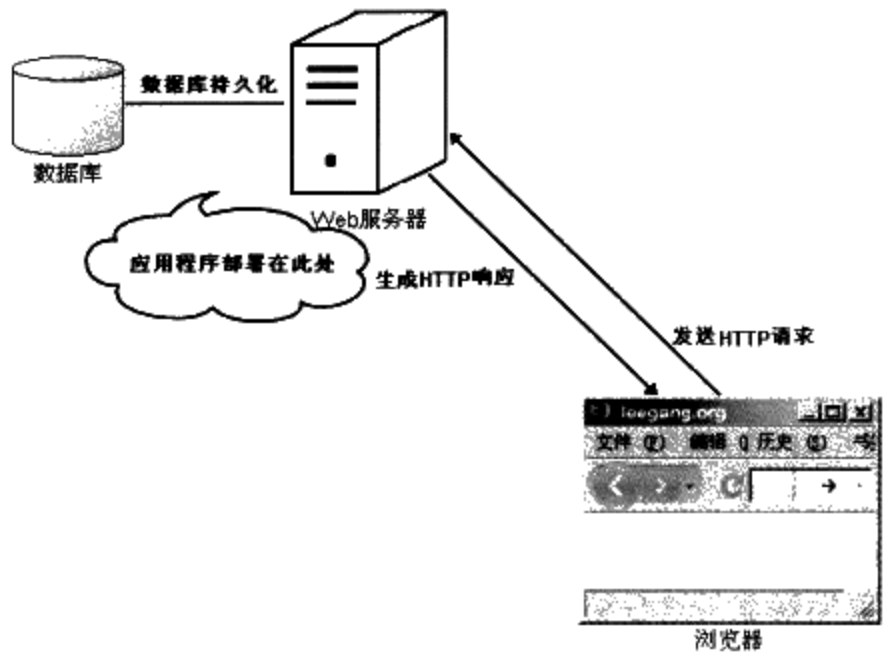


图 1.2　B/S 结构的应用

B/S 结构得到迅速推广是有原因的。在大部分情况下，B/S 结构的应用比 C/S 结构的应用更加优秀，适应性更广。相对而言，B/S 结构的系统有如下方面的优势：

- 数据安全性高。由于 C/S 结构软件的数据分布特性，客户端所发生的火灾、地震、盗抢、病毒、黑客攻击等都成了可怕的数据杀手。另外，对于集团级的异地软件应用，C/S 结构的软件必须在各地安装多台服务器，并在多台服务器之间进行数据同步。因此，每个数据点上的数据安全都会影响到整个应用的数据安全。对于跨区域的大型应用而言，C/S 结构软件的安全性是令人无法接受的。而对 B/S 结构的系统而言，数据集中存放于总部的数据库服务器（服务器数据可以通过多种方式备份存放），客户端不保存任何业务数据和数据库持久化信息，无须进行数据同步，所以这些安全问题自然也就不存在了。
- 数据一致性好。在 C/S 结构的解决方案里，对于跨区域的大型企业应用都采用各地安装区域级服务器，然后再进行数据同步的模式。这些服务器每天必须同步完毕之后，总部才可得到最终的数据。由于局部网络故障等原因，可能造成个别数据库无法正常同步。即使都能正常同步，但各服务器往往不能同时同步，因而数据也无法绝对一致，不能用于决策。而对于 B/S 结构的系统而言，数据是集中存放的，客户端发生的每次数据修改都直接进入到中央数据库，不存在数据一致性的问题。
- 数据实时性好。对于大型的跨区域应用而言，C/S 结构不可能随时跟踪各客户端的业务发生情况，因为数据都不是实时更新，因而看到的数据都是滞后的；而 B/S 结构则不同，因为其数据都是实时存入服务器端的数据库，因而服务器端可以实时看到当前发生的所有业务，能提供更好的企业决策支持。
- 系统更新方便。软件供应商提供的软件不可能是完美无缺的。即使是一个绝对完美的软件系统，当具体的业务环境发生改变后，系统也应随之改变。因而，必须经常对已部署的软件产品进行维护、升级。对 C/S 结构的软件而言，由于其应用是分布的，需要对每一个节点手动进行程序安装，所以，即使非常小的程序缺陷都需要很长的时间来重新部署。重新部署时，

为了保证各程序版本的一致性，必须暂停业务进行更新（即“休克更新”）。在很多情景下，这是不可忍受的。而 B/S 结构的软件则不同，其应用程序集中于服务器上，各应用节点没有任何程序，应用的更新只需要更新服务器端程序即可，因而可以做到快速的服务响应。

- 网络应用限制小。C/S 结构的软件通常仅适用于局域网内部用户或宽带用户（1M 以上）；而 B/S 结构的软件可以适用于任何网络结构（包括 28.8K 拨号入网方式），特别适于宽带不能到达的地方。

传统的 C/S 结构软件开发工具有早期的 Visual Basic、Visual FoxPro 等，这些开发工具目前已经趋于淘汰。目前依然使用的开发工具有 PowerBuilder、Delphi 等。除早期的一些系统外，现在新开发的系统大部分使用 B/S 结构。

B/S 结构的应用开发早期有 ASP、JSP 和 PHP 等。早期这些 B/S 结构开发技术相当混乱，系统中业务逻辑、数据持久化、控制逻辑混在一起，这些处理逻辑都通过页面的脚本实现。早期的 B/S 结构应用面临着后期维护困难、难以扩充的问题。

MVC 设计模式重新定义了 B/S 结构应用的开发模式，规定 B/S 结构应用应该分成 Model（M：模型）、View（V：视图）和 Controller（C：控制器）三个部分。MVC 模式分离的数据访问和数据表现，给系统提供了更好的解耦。MVC 架构的核心思想是，将程序分成相对独立而又能协同工作的三个部分。通过使用 MVC 架构，可以降低模块之间的耦合，提供应用的可扩展性。MVC 中的每个组件只关心组件内的逻辑，不应与其他组件的逻辑混合。

Java EE 的出现，则更加规范了 B/S 结构应用的开发。Java EE 推荐将应用分为数据持久层、业务逻辑层和 Web 层，各层之间以松耦合的方式组织在一起。

目前，Ajax 的出现，再次完善了传统的 Web 应用。Ajax 应用强调异步发送用户请求：用户在浏览页面的同时可以发送请求，在第一个请求的服务器响应还没有完全结束时，可以再次发送请求。这种请求的发送方式非常类似于传统的 C/S 结构的应用。

在传统的 Web 应用里，因为用户总是需要加载新页面时才提交请求，而提交请求后又需要等待服务器响应，所以如果服务器响应还没有完全结束，则用户只能等待，不能继续发送请求。

与传统 Web 应用不同的是，Ajax 将请求与页面分离：在传统的 Web 应用里，每个请求即对应一个页面，不管客户端以 POST 还是 GET 方式提交请求，每次请求都会丢弃当前页面，等待服务器生成新页面。在等待期间，旧的页面已经丢弃，新的页面还没有完全生成，整个浏览器将一片空白，而用户什么都做不了，只能等待——对于用户而言，这是一种不连续的体验，感觉非常不好。

➤➤1.1.2 传统 Web 应用的优势和缺点

经过前面对应用发展历史的介绍我们知道：B/S 结构依然是目前应用的主流结构。Ajax 技术并没有提出一种全新的应用开发结构。Ajax 并不是要取代传统 B/S 结构应用，而是对传统 B/S 结构应用的完善，从而提供给用户一种更连续的体验。

传统 Web 应用取代 C/S 结构的应用自然有其必然的理由，那就是传统 Web 应用的优势：

- 客户端的零安装，零部署。客户端软件就是浏览器，无须用户安装、部署新的客户端，客户端只需启动浏览器即可以运行系统。
- 系统更加安全。所有的数据和业务逻辑全部部署在服务器端，多层的应用结构将数据库隐藏在底层，使得系统更加安全。
- 数据抗风险能力加强。所有的业务数据由服务器统一管理，服务器数据是完整的业务数据，服务器可以提供完整的业务数据备份。
- 更广的网络适应。处于万维网上的应用可以被全世界的浏览者访问。

但传统的 Web 应用也存在种种问题，比如界面简单，独占式请求。传统的 Web 应用的不足主要表现在如下方面：

- 独占式的请求：比如一个任务需要多步骤或多选项该任务才能完成。在 HTML 里，一个多步骤的任务可以在单页内表达出来。但是由于 HTML 的互动性有限，便可能产生一份很长的页面，使用户感到混乱、笨拙而难以使用。或者将多个步骤分成几个页面分别提交，但传统的独占式的请求，如果前一个请求没有得到完全响应，则后一个请求不能发送。用户在等待服务器的响应期间，浏览器一片空白。这种独占式请求如图 1.3 所示。

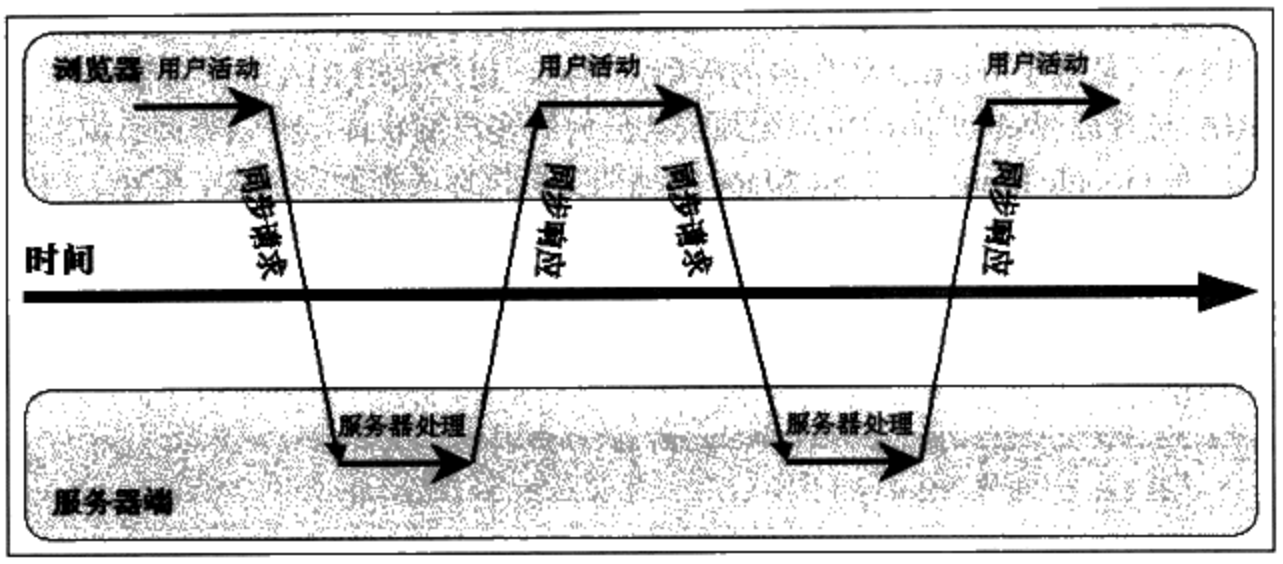


图 1.3　独占式请求的示意图

- 频繁的页面刷新：传统的 Web 应用基本上采用请求一页面的对应模式，每个请求都需要丢弃当前页面来重新加载新页面。频繁的页面刷新不仅让客户处于不连续的体验中，也使服务器的负担加重。
- 简陋的页面：传统 Web 应用因为需要频繁刷新页面，因而不可能制作出具有丰富表现功能的页面。丰富表现的页面导致页面文件过大，下载速度更慢，而且页面频繁刷新。一个表现丰富的页面下载需要相当多的时间，但随着请求的提交，又需要重新下载新页面，这样系统开销相当大。因而传统 Web 应用的页面不可能非常出色。

而 Ajax 正是为弥补以上不足而诞生，Ajax 使用 XMLHttpRequest 对象异步发送请求。Ajax 应用不采用请求对应页面的模式，请求就是请求，发送请求不要求重新加载页面。浏览器发送请求后，无须等待服务器响应，而是可以继续原来的操作。在服务器的响应完成后，客户端使用 JavaScript 函数将响应数据加载到浏览器中。

通过使用 Ajax 技术，用户发送请求，请求得到响应这个过程在后台进行，用户的界面以连续的方式进行。

1.2　重新设计 Web 应用

传统 Web 应用的不足，一直凸显在用户面前，用户常常抱怨系统的响应速度太慢。除了网络带宽的限制、业务逻辑复杂、硬件设备制约等因素外，频繁的页面刷新，每次响应都必须下载整个响应页面，也导致了响应速度慢。针对传统 Web 应用界面简单的弊病，有相当多的改进 Web 应用的设想，例如 DHTML 以及后来的所谓 RIA 应用。

➤➤1.2.1　富 Internet 应用

RIA，是 Rich Internet Application 的缩写，即富 Internet 应用。B/S 结构已成为应用程序开发的默认结构，用户对应用程序复杂性要求日增，Web 应用程序对完成复杂逻辑始终差强人意。

用户与复杂的 Web 应用程序交互时，其体验并不能令人满意。Web 模型是基于 HTML 页面的模型，缺少客户端智能机制。传统的 Web 应用几乎无法完成复杂的用户交互（如传统的 C/S 应用程序和

桌面应用程序中的用户交互）。因而，Web 在许多应用程序中难以发挥。

为了提高用户体验，出现了一种新类型的 Web 应用，那就是 Rich Internet Applications（RIA）。这些应用程序吸收了桌面应用程序的反应快、交互性强的优点，改进了 Web 应用程序的用户交互，可以提供一种更丰富、更具有交互性和响应性的用户体验。

RIA 架构可以理解为运行于 B/S 结构上的 C/S 应用。应用客户端采用标准的浏览器，但在浏览器内支持类似 C/S 应用的操作，所以 RIA 应用可以提供强大的功能，让用户有高交互性、高效率响应的体验。同时，RIA 又是基于 Internet 浏览器的应用，所以，用户使用 RIA 非常方便。

理想的 RIA 与普通 Web 页面，用户无须安装任何的客户端软件，而只需拥有浏览器。一个典型的富客户端应用是 Google Maps。Google Maps 的地图支持鼠标的拖动和放大、缩小。地图随着鼠标的拖动而拖动——明显加载了新的数据，但页面本身却无须重新加载。如果鼠标拖动得太远，可能出现部分空白区域，但这种空白只是地图区域在加载，而不是整个页面在加载。

当使用鼠标单击地图上的提示点时，地图上可以出现该点的更详细的介绍。图 1.4 显示了 Google Maps 中的美国地图。

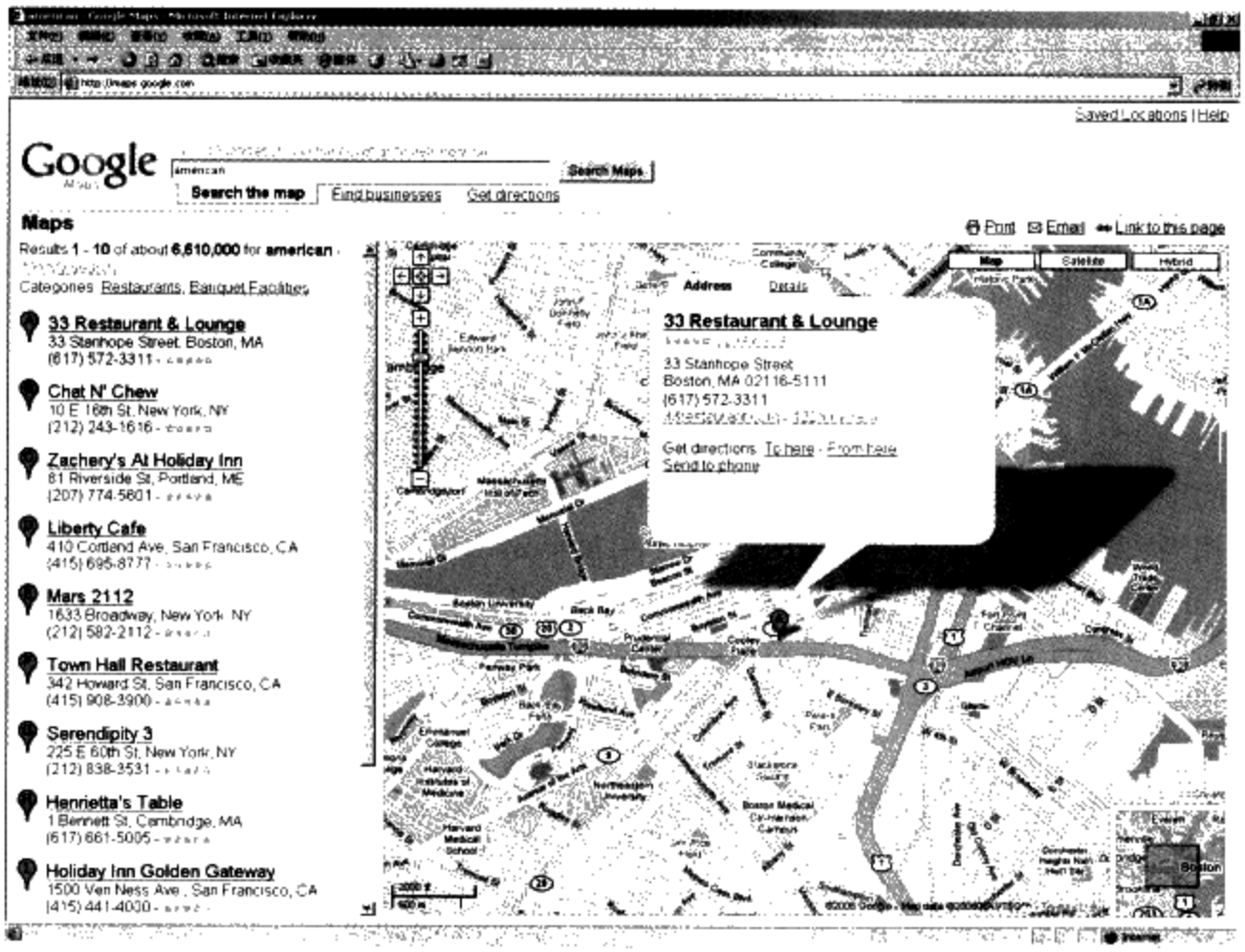


图 1.4　Google Maps 应用

目前，典型的 RIA 技术有：

- Microsoft 的 ClickOnce 技术。
- Sun 的 Java Web Start 技术。
- Adobe 的 Flash 技术。
- Ajax 技术。

可以说，RIA 代表着目前 Web 应用的发展趋势，因而各大软件厂商均希望在 RIA 方面获得自己的市场份额。而 Ajax 并不属于任何软件厂商。

Ajax 代表的是一种开源风格，而且采用的大部分都是早已存在的技术，如 JavaScript、CSS 等。Ajax 所采用的技术是基于标准的，并不属于特定厂商，是一种真正开源风格的 RIA。

同时，与其他 RIA 不同的是，基于 Ajax 技术的应用完全基于现有的浏览器，所以兼容性最好。

基于 Ajax 技术的应用，通常无须下载任何客户端，这也正是 Ajax 的魅力所在。

➤➤1.2.2 异步发送请求，避免等待

与之前简单的 DHTML 页面相比，所有 RIA 的共同特点是：允许在同一页面多次发送请求，都将传统 Web 应用的每请求对应页面拆分，页面只是单纯的视图，负责显示数据，而请求与页面之间并无一一对应的关系。

就某种程度而言，Ajax 就是传统的 DHTML 页面+异步发送请求，当然也包括了动态装载服务器数据。如图 1.5 所示为异步发送请求的示意图。

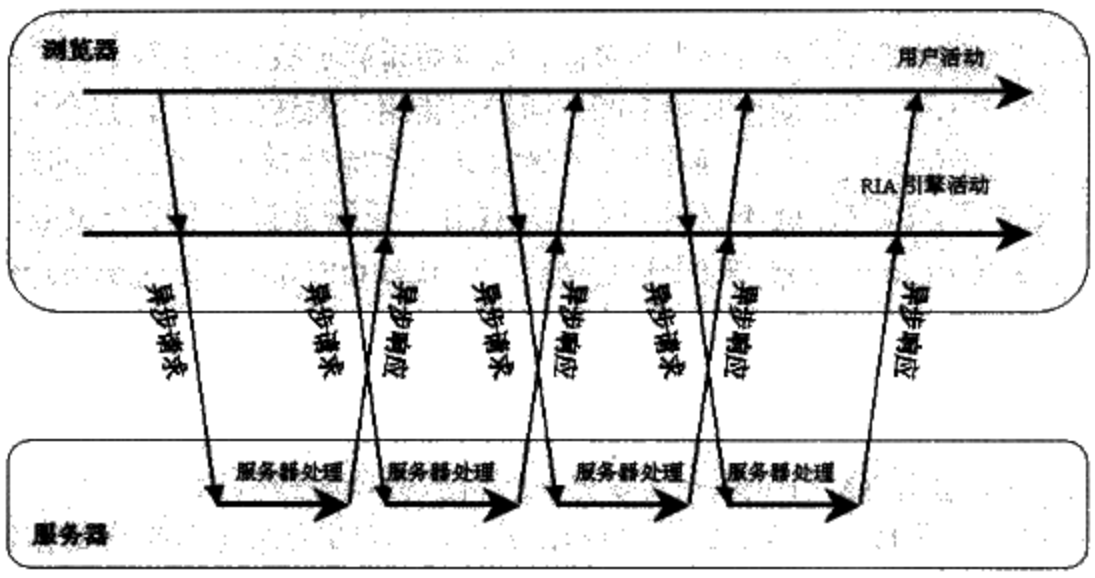


图 1.5 RIA 的异步发送请求

RIA 的特征除异步发送请求外，还有动态加载服务器响应数据。使用 RIA 能避免频繁刷新页面，服务器响应的是数据，而不是整个页面内容。RIA 负责获取服务器数据，然后将其动态加载到浏览器中。

➤➤1.2.3 使用 Ajax

在前面的介绍中已经讲过：Ajax 应用本质上是一种 RIA，而构建 RIA，Ajax 也并不是唯一的选择。甚至在某些特殊的情形下，Ajax 并不是最合适的选择。但我们完全有理由相信：在大部分情形下，使用 Ajax 改善传统 Web 应用，能提供更好的适应性。

Ajax 的优势非常明显：基于 Ajax 的应用无须浏览器下载任何插件，并可以在任何平台上良好运行。

Ajax 还有个显著的优势是，所用的技术大都是些“古老”的技术，例如 JavaScript、XML、DOM、CSS 等。对于开发人员而言，相比于重新选择新的技术，如 ClickOnce 等，他们将更乐意选择这些已经熟悉的技术。同时，这些技术都是标准的，并不属于任何特定的厂商，目前所有浏览器都对 Ajax 技术支持良好。所以 Ajax 技术自从 2005 年问世以来，已在业界得到迅速推广，到现在已很难找到没有使用 Ajax 的 B/S 应用了。

Ajax 使用简单的 XMLHttpRequest 对象发送请求，使用简单的 JavaScript 函数监视服务器响应。在服务器响应完成后，JavaScript 通过 DOM 动态更新 HTML 页面。自始至终，用户的动作无须中断，所感受的是一种连续的体验。

1.3 Ajax 介绍

Ajax 由 Jesse James Garrett 在 2005 年 2 月的一篇文章中提出。不过 Ajax 并不是一种新的语言或技术。实际上，它由几种已有的技术组合而成。

Ajax 通过在浏览器和服务器之间添加 Ajax 中间层，允许浏览器异步发送请求，同时允许动态加

载服务器响应。用户的请求不再直接向服务器提交，而是使用 XMLHttpRequest 异步地向服务器发送，从而避免了丢弃当前页面。

➤➤1.3.1 Ajax 的工作方式

Ajax 的核心是 JavaScript 对象 XMLHttpRequest。该对象在 Internet Explorer 5 中首次引入，它提供了异步发送请求的能力。简而言之，使用 XMLHttpRequest，可以通过 JavaScript 向服务器发送请求，并能够处理服务器响应，避免阻塞用户动作。通过使用 XMLHttpRequest 对象，浏览器通过客户端脚本与服务器交换数据，而 Web 页面无须频繁重新加载，Web 页面的内容也由客户端脚本动态更新。

异步，指基于 Ajax 的应用与服务器通信的方式。对于传统的 Web 应用，每次用户发送请求，向服务器请求获得新数据时，浏览器都会完全丢弃当前页面，而等待重新加载新的页面。而在服务器完全响应之前，用户浏览器将一片空白，用户的动作必须中断。而异步指用户发送请求后，完全无须等待，请求在后台发送，不会阻塞用户当前活动。用户无须等待第一次请求得到完全响应，可以立即发送第二次请求。

使用 Ajax 的异步模式，浏览器就不必等用户请求操作，无须重新下载整个页面，一样可以显示服务器的响应数据。Ajax 使用 JavaScript 来回传送数据，XMLHttpRequest 是 Ajax 的核心，JavaScript 则是 Ajax 技术的粘合剂。整个 Ajax 应用的工作过程如下：

（1）JavaScript 脚本使用 XMLHttpRequest 对象向服务器发送请求。发送请求时，既可以发送 GET 请求，也可以发送 POST 请求。

（2）JavaScript 脚本使用 XMLHttpRequest 对象解析服务器响应数据。

（3）JavaScript 脚本通过 DOM 动态更新 HTML 页面。也可以为服务器响应数据增加 CSS 样式表，在当前网页的某个部分加以显示。

在前面已经多次提到，Ajax 并不是新技术，而是一些传统技术的组合。在这些传统的技术中，除了 XMLHttpRequest 比较新外，其他技术都有了很多年的历史。下面就来简单介绍一下其中的关键技术。

➤➤1.3.2 Ajax 的核心：XMLHttpRequest

XMLHttpRequest 是整个 Ajax 技术的灵魂。可以说，没有 XMLHttpRequest，就没有 Ajax。Ajax 技术的核心是异步发送请求，而 XMLHttpRequest 则是异步发送请求的对象。如果抛开异步发送请求，Ajax 的其他技术将完全失去原有的意义。

最早应用 XMLHTTP 的是微软。IE（IE5 以上）允许在 Web 页面内部使用 XMLHTTP ActiveX 组件，从而扩展自身的功能，可以无须从当前 Web 页面发送请求，允许直接传输数据给服务器，并允许直接从服务器取数据。这个功能是很重要的。因为它减少了无状态连接的痛苦，还可以避免下载冗余 HTML 代码，从而可以提高进程的速度。

后来，Mozilla（Mozilla 1.0 以上及 Netscape Navigator 7 以上）也有了自己的实现——XMLHttpRequest 对象。Konqueror （还有 Safari 1.2，同样也是基于 KHTML 的浏览器）也支持 XMLHttpRequest 对象。而 Opera 在其 7.6 以后的版本中也增加了对 XMLHttpRequest 的支持。

大部分情况下，XMLHttpRequest 对象和 XMLHTTP 组件非常相似，方法和属性也类似。只有极个别的属性存在细微差异。XMLHttpRequest 对象在 Mozilla 浏览器中通过如下代码完成初始化：

```
var xmlhttp = new XMLHttpRequest();
```

微软的 XMLHTTP 组件在 JavaScript 中采用如下代码完成初始化：

```
var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
var xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
var xmlhttp = new ActiveXObject("Msxml3.XMLHTTP");
```

注意：

上面的代码中的 XMLHTTP 组件的前缀各不相同。因为在不同的 IE 版本中，XMLHTTP 的实现版本并不相同。

下面的代码则以一种更通用（能实现跨浏览器访问）的方式创建了 XMLHttpRequest 对象：

```
function createXMLHttpRequest()
{
    //对于基于 Mozilla 的浏览器
    if (window.XMLHttpRequest)
    {
        //对于基于 Mozilla 的浏览器，直接创建 XMLHttpRequest 对象
        return new XMLHttpRequest();
    }
    //对于 IE 浏览器
    else if (window.ActiveXObject)
    {
        //IE 浏览器中 XMLHTTP 的实现版本并不相同
        var msxmls = ["MSXML3", "MSXML2", "Microsoft"];
        for (var i=0; i < msxmls.length; i++)
        {
            try
            {
                //创建 XMLHTTP 组件
                return new ActiveXObject(msxmls[i]+".XMLHTTP");
            }
            catch (e)
            {
                alert("浏览器不支持 XMLHTTP 控件");
            }
        }
    }
}
```

只有借助 XMLHttpRequest 对象，Ajax 才能实现异步发送请求。XMLHttpRequest 是浏览器与服务器交换信息的载体。在整个 Ajax 技术中，XMLHttpRequest 是灵魂。

➤➤1.3.3 Ajax 的编程脚本：JavaScript 语言

JavaScript 是一种跨平台的脚本语言，虽然很多地方也称 JavaScript 为面向对象的语言，但实际上 JavaScript 并不是一种纯粹的面向对象语言。JavaScript 简单易用，而且在绝大部分浏览器中都运行良好。

JavaScript 脚本是 Ajax 技术中另一个重要部分。JavaScript 主要完成如下事情：

- 创建 XMLHttpRequest 对象。
- 通过 XMLHttpRequest 向服务器发送请求。
- 创建回调函数，监视服务器响应状态，在服务器响应完成后，回调函数启动。
- 回调函数通过 DOM 动态更新 HTML 页面。

JavaScript 是 Ajax 技术的粘合剂，通过将其他几个技术有机地结合在一起，从而形成了 Ajax 技术。

➤➤1.3.4 HTML 页面的 DOM 模型

DOM（Document Object Model）是操作 HTML 和 XML 文件的一组 API，它提供了文件的结构表述。通过使用 DOM，可以采用编程方式操作文档结构，可以改变文档的内容。通过使用 DOM，HTML 页面以一种结构化方式组织在一起，HTML 页面的内容以节点方式组织。Web 程序开发者可以增加文

件的节点、属性及事件，从而提供对 HTML 页面的动态更新，例如 document 就代表“HTML 文件本身”，而 table 对象则代表 HTML 的表格对象等。

HTML 页面中 DOM 模型的主要功能是允许 JavaScript 动态操作 HTML 文档。如图 1.6 所示为一个简单的 HTML 页面的 DOM 树形图。

由图 1.6 可以看出，通过 DOM 可将 HTML 页面视为一组包含父子关系的节点。JavaScript 可以访问每个节点，修改节点内容及其属性，也可以新增节点、删除节点。这些 DOM 操作将直接对应 HTML 页面内容的改变。简而言之，DOM 提供了动态改变 HTML 页面内容的方法。

DOM 也是 Ajax 的核心技术。没有 DOM，JavaScript 在获取服务器数据后无法动态更新 HTML 页面，获得的数据无法显示在用户的当前浏览页面中。事实上，DOM 也是 JavaScript 获取页面数据的方式。在 JavaScript 发送请求之前，已经需要使用 DOM 来获取请求数据了。

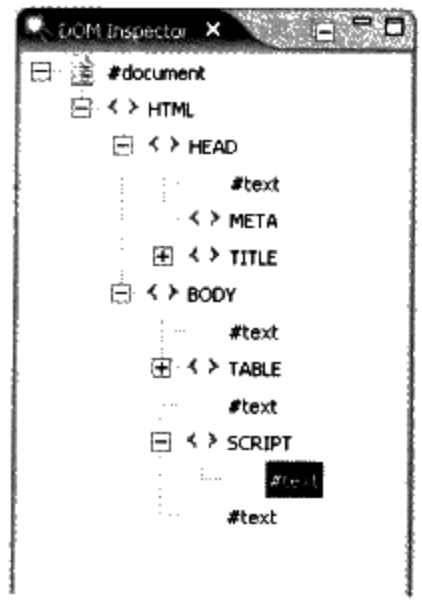


图 1.6　HTML 页面的 DOM 树形图

➤➤1.3.5　数据交换和显示

CSS（Cascading Style Sheets，级联样式单）和 XML（可扩展标记语言）并不是 Ajax 所必需的技术。但通常 Ajax 依然离不开这两项技术。实际上，HTML 页面本身就离不开 CSS，如果想让 HTML 页面更美观，就需要 CSS 的配合了。

在 Web 页面中采用 CSS 技术，可以有效地对页面的布局、字体、颜色、背景和其他效果实现更加精确的控制。通过 CSS 技术，只要对相应的代码做一些简单的修改，就可以改变同一页面的不同部分，CSS 技术的优点主要有：

- 目前几乎所有浏览器都支持 CSS 技术。
- 支持丰富的表现效果。以前一些采用图片实现的效果，现在完全可以通过 CSS 实现，从而提供更快的下载速度。
- 页面的字体更漂亮，更容易编排，页面效果更加美观。
- 支持更好的页面布局。
- 同一个 CSS 文件可以同时控制多个页面，从而避免重复更新每个页面。

CSS 主要的工作是让页面的表现更友好。虽然 CSS 并不是 Ajax 所必需的，但对于实际应用而言，用户界面的友好是非常重要的，因而 CSS 也是必不可少的技术。

XML 文档是一种结构化文档，其主要作用有：

- 用于简单数据的表示和交换。
- 用于面向消息的计算。
- 与用户界面相关，表示相关的上下文。

关于 Ajax 的这些相关技术，本书将在后面的章节详细介绍，这里就不再深入介绍了。

1.4　Ajax 的基本特征

传统的 Web 应用已经存在了很多年，而针对传统 Web 应用的改进一刻也没有停止过。可能有一些技术在很多方面与 Ajax 非常相似，但并不是 Ajax。下面介绍 Ajax 应用的基本特征。

➤➤1.4.1　异步发送请求

异步发送请求是 Ajax 应用最核心的内容。如果抛开了异步发送请求这个特征，那么不管页面做得

多么丰富多彩，外表上多么像桌面应用，也都不可能是 Ajax 应用。

Ajax 应用的巨大改进之处，在于给用户的连续体验。用户发送请求后，还可以在当前页面浏览，或者继续发送请求，即使服务器响应还没有完成。而服务器响应完成后，浏览器并不是重新加载整个页面，而是仅加载需要更新的部分。

➤➤1.4.2 服务器响应是数据，而不是页面内容

与传统的 Web 应用不同的是，服务器不再生成整个 Web 页面。这是一种非常“浪费”的行为，这种浪费不仅对于用户不利，对于服务器也是一样。用户从服务器完整下载了一个 Web 页面，随着服务器响应的到来，用户再次重新下载新的页面，也许这两个页面的基本内容完全相同，只有极个别的数据需要修改，但用户不得不下载全部页面，而服务器则不得不提供对应的带宽给用户下载。

例如对于一个实时的股票行情显示系统，每隔一段时间需要实时刷新当前的股票行情。当前页面的大部分内容如图片、Flash 动画等都无须改变，甚至股票名称文字也无须改变，需要改变的仅仅是当前的股票价格。但传统的 Web 应用里，每隔一段时间都需要重复下载整个页面，将导致服务器负载加重，而用户则处于一种不连续的体验中。

在 Ajax 应用中，网络负载主要集中在应用加载期，也就是页面第一次下载时。一旦页面下载成功，则相当于在客户端部署了复杂的应用。而后面的操作将是相当迅速的，客户端的 JavaScript 负责与服务器通信，从服务器获取必须更新的部分数据，而不再是整个页面内容。因此，Ajax 的累积网络流量比传统 Web 应用要小得多。

➤➤1.4.3 浏览器中的是应用，不是简单视图

传统的 Web 应用中，浏览器只是简单视图，负责显示系统状态，并收集用户信息提交给服务器，浏览器没有任何逻辑功能。当然在传统的 Web 应用中，也不允许浏览器中包含逻辑。因为如果在页面中包含逻辑，则随着用户请求的提交，页面被丢弃，所有的逻辑都将丢失。

传统的 Web 应用中，浏览器没有包含逻辑，更不能包含用户的会话状态。因为如果将状态保存在客户端，则随着页面的刷新，用户会话状态将丢失。

对于 Ajax 应用则完全不同，浏览器不仅可以包含简单逻辑，甚至可以保存用户会话状态。因为 Ajax 应用有个特点：无须刷新页面即可完成内容的动态更新。

例如一个简单在线购物系统，用户的购物车就是典型的会话状态。在传统的 Web 应用里，都会采用 session 保存会话状态，即将用户的状态信息保存在服务器端。每次用户购买物品，都必须提交一次请求，从而将购买物品提交到服务器 session 中。而在 Ajax 应用中则无须使用 session，Ajax 应用可采用 JavaScript 的变量保存用户购买的所有物品，而用户每次购买的物品也无须提交给服务器 session，而是直接修改浏览器中的 JavaScript 变量。在这种情况下，Web 页面既保存了用户的状态信息，又处理了部分业务逻辑。直到用户提交购买，数据需要持久化时，JavaScript 才将请求发送到服务器。

Ajax 应用初始化时，需要加载大量的 JavaScript 代码。这些 JavaScript 代码中已经包含了部分业务逻辑，将在后台默默工作，负责处理部分逻辑，异步提交请求，以及读取服务器响应数据，动态更新页面。

1.5 Ajax 的替代技术

正如前面所介绍的，Ajax 应用的本质就是一种 RIA。而构建 RIA，并不是非 Ajax 莫属。下面介绍一些 Ajax 技术的主要替代方案，这些替代方案可能在某些方面更优秀。

➤➤1.5.1 Sun 的 Java Web Start 技术

Java Web Start 是 Java 应用程序的一种部署解决方案。Java Web Start 提供一次单击激活应用程序

的简易方法，并保证始终运行应用程序的最新版本，从而可避免复杂的安装或升级过程。

传统情况下，通过 Web 发布软件需要用户在 Web 上查找、下载，然后在系统中存放，并执行安装程序。执行安装程序后，将提示指定安装路径和安装选项，例如完全、典型或最小安装。这是一项耗时而又复杂的任务，并且在安装软件的每个新版本时都必须重复进行。相反，通过 Web 部署的应用程序，都非常容易安装和使用。

通过使用 Java Web Start，浏览器自动完成整个安装过程。没有复杂的下载、安装和配置过程，并且确保软件是最新版本。

Java Web Start 提供了通过 Web 部署应用程序的解决方案，提供了基于 B/S 模式的 C/S 应用的解决方案，比简单的 B/S 应用有更好的特性：

- 高交互性的用户界面。
- 低带宽要求。应用程序不需要和 Web Server 始终保持连接，可以在本地的硬盘中保留应用程序的缓存，并通过缓存来运行应用程序。
- 支持离线工作。

使用 Java Web Start，只需要在第一次使用应用时下载该程序。而一旦下载完成，Java Web Start 会将应用程序在本地进行缓存。虽然第一次运行时所花的代价要比 HTML 应用高，但此后就可以在任何时候立即运行。

每次运行应用程序的时候，Java Web Start 都会自动连接服务器，检查是否有新的版本出现，如果有就下载需要更新的文件，从而保证运行的程序是最新版本，省却了手动升级的麻烦。

➤➤1.5.2 Microsoft 的 ClickOnce 技术

ClickOnce 是一种部署技术。使用该技术可创建自行更新的 Windows 应用，这些应用程序几乎无须用户参与即可完成应用的安装和运行。ClickOnce 克服了传统部署的三个主要问题：

- 更新应用程序困难。使用 Windows Installer 部署，每次应用程序更新，用户都必须重新安装整个应用程序。而使用 ClickOnce 部署，则可以提供自动更新。只有更改过的应用程序部分才会被下载，然后从新的并行文件夹重新安装完整的、更新后的应用程序。
- 对用户的计算机的影响。使用 Windows Installer 部署时，应用程序通常依赖于共享组件，这便有可能发生版本冲突。而使用 ClickOnce 部署时，每个应用程序都是独立的，不会干扰其他应用程序。
- 安全权限。Windows Installer 部署要求具有管理员权限并且只允许有限的用户安装。而 ClickOnce 部署允许非管理员用户安装应用程序并仅授予应用程序所需要的那些代码访问安全权限。

通过使用 ClickOnce 技术，可以创建基于 Web 的应用程序，但这种应用程序可以提供 Windows 窗体丰富的用户界面和响应性。通过使用 ClickOnce 技术，还可以开发出基于 B/S 架构的 Windows 程序。

➤➤1.5.3 基于 Flash 的 Flex

Flash 在早期仅仅是一种简单的网页动画，它的出现改善了静态 HTML 页面的外观。Flash 是一种流媒体的格式，无须等到动画完全下载，即可立即播放。

后来，Flash 增加了一种叫 ActionScript 的脚本，从而允许播放交互式电影。ActionScript 还提供了对输入表单 UI 组件的支持。

虽然 Flash 动画的运行必须依赖一种浏览器插件，但这种插件已经存在了很长时间。通常，各种主流浏览器都已经包含了 Flash 插件，而且该插件也可以跨 Windows、Linux 等操作平台。

Flex 应用与传统的 HTML 应用程序的主要区别在于：Flex 应用可提供给用户更快的响应，在不同状态与显示之间流畅过渡，并提供连续的用户体验。

Flex 应用程序框架由 MXML、ActionScript 及 Flex 类库构成。开发人员利用 MXML 及 ActionScript

编写 Flex 应用程序。利用 MXML 定义应用程序用户界面元素，利用 ActionScript 定义客户逻辑与程序控制。Flex 类库中包括 Flex 组件、管理器及行为等。利用基于 Flex 组件的开发模型，开发人员可在程序中加入内建组件、创建新组件等。

对比三种技术可以发现：Java Web Start 和 ClickOnce 本质上还是一种 C/S 结构应用，只是这种应用转为以 HTML 页面为容器，因而可以自动安装和更新客户端程序。Java Web Start 和 ClickOnce 都是对于传统 C/S 结构应用的改进，而不是对 B/S 结构应用的改进。因此所有的 Windows 程序、ClickOnce 的应用只能在 Windows 平台下运行。而基于 Flash 的 Flex 应用，则提供比 Ajax 更好的用户界面，因为 Flash 起初的目标就是制作网页动画，它支持非常丰富的图形界面。

可见，如果需要客户端完成相当多的业务逻辑，而且应用大部分在局域网内进行，可以采用 Java Web Start 和 ClickOnce 技术。如果应用需要在不同平台上运行，则应该使用 Java Web Start。如果应用需要非常丰富的客户端表现效果，例如游戏，则应该考虑使用 Flash。

在某种程度上，Ajax 综合了三项技术的优点，但达不到在三者的强项上的高度。JavaScript 要完全达到 Java Web Start 或 ClickOnce 那种桌面应用的界面，难度相当大（目前有个商用 JavaScript 库 Bindows 做得非常出色）。而要达到 Flash 那么丰富的动画效果，则更不可思议。但大部分情形下，采用 Ajax 是个不错的选择。毕竟，对于真正的应用而言，构建复杂的动画界面是不是太“舍本逐末”了一点？而 Java Web Start 和 ClickOnce 毕竟不是对 B/S 结构应用的改进，而是对传统 C/S 结构应用的改进。

1.6 搭建 Ajax 开发环境

Ajax 是一种技术，这种技术并不局限于任何特定语言、特定平台。从某个方面讲，Ajax 更像一种思想，而不是一种技术。在不同的 Web 开发平台上都可以施展 Ajax 技术，例如 ASP、PHP、.NET 和 Java EE 等。本节将介绍本书的 Ajax 所使用的语言平台，以及如何搭建 Ajax 的运行开发环境。

➤➤1.6.1 本书的 Ajax 开发环境

本书介绍的 Ajax 应用全部基于 Java EE 平台，虽然 Ajax 的思想在其他开发平台上完全一样。本书中所有的 Ajax 应用，都基于 Java EE 开发平台完成。在本书的 Ajax 应用中，Ajax 作为上层表现层技术，与下面的 Web 层交互，Web 层的控制器则负责拦截 Ajax 引擎发出的 XMLHttpRequest 请求，并调用业务逻辑层对象处理请求。底层的持久化将通过 DAO 组件完成。

本书作为疯狂 Java 体系的一员，并不打算为介绍 Ajax 而 Ajax，而是致力于让 Ajax 有机地融入 Java EE 体系中。本书并不准备讲述 Ajax 在其他 Web 开发技术上的应用，而将专注于 Ajax 与 Java EE 平台的整合。本书将介绍以 Ajax 技术为核心，以 Java EE 技术为支撑，二者有机结合的 Web 2.0 应用。

后面的实例都将以 Java EE 应用为核心——主要基于 Spring、Hibernate 两个轻量级框架，表层融入 Ajax 技术。这些实例演示了 Ajax 技术的作用：更好地改善传统 Java EE 应用给用户的体验。

提 示：

本书可分为 2 个部分阅读，前面 8 章主要介绍 XHTML、JavaScript、CSS、DOM 和事件机制等基础知识，阅读前面 8 章几乎无须任何基础知识。但本书后面章节的内容则大量使用了 Hibernate、Spring 框架知识，建议读者先阅读疯狂 Java 体系的《轻量级 Java EE 企业应用实战》一书。

本书采用的 Java 版本为 JDK 6 Update 6。本书介绍的 Ajax 应用都将以 Tomcat 6.0.16 作为服务器。本书介绍的 Ajax 应用都是轻量级 Java EE 应用整合了 Ajax 技术的产物，因而采用 Tomcat 6.0.16 作为 Web 容器即可——当然，也可以在其他 Web 服务器上运行良好。

➤➤1.6.2 安装 Tomcat 服务器

Tomcat 是 Java 领域最著名的开源 Web 容器，简单易用，稳定性极好。既可以作为个人学习之用，

也可以作为商业软件系统的 Web 服务器。Tomcat 不仅提供了 Web 容器的基本功能，还支持 JAAS 和 JNDI 绑定等。

Tomcat 最新的发布版本为 6.0.16，笔者所介绍的应用也是基于该版本的 Tomcat，建议读者安装这个版本的 Tomcat。

Tomcat 完全是纯 Java 实现，因此是平台无关的，在任何平台上运行都完全相同。在 Windows 和 Linux 平台上的安装和配置基本相同。本节以 Windows 平台为示范，介绍 Tomcat 的下载和安装。在 Windows 平台上安装 Tomcat 请按如下步骤进行：

（1）登录 http://tomcat.apache.org 站点，下载合适的 Tomcat 版本，本书使用 JDK 1.6，因此使用 Tomcat 6.0.x 系列。

提 示：

目前 Tomcat 有几个稳定的产品版本，通常 JDK 1.4 建议使用 Tomcat 5.0.x 系列，JDK 1.5 建议使用 Tomcat 5.5.x 系列，JDK 1.6 则建议使用 Tomcat 6.0.x 系列。

Tomcat 6.0.x 系列的最新稳定版本是 6.0.16，建议下载该版本，对于 Windows 平台下载.zip 包，而 Linux 平台则下载.tar.gz 包。建议不要下载安装文件。因为安装文件的 Tomcat 看不到启动、运行时控制台的输出，不利于开发者使用。

（2）解压缩下载到的压缩包，完成后应有如下目录结构：

- bin：存放启动和关闭 Tomcat 的命令。
- conf：存放 Tomcat 的配置，所有的 Tomcat 配置都在该目录下设置。
- lib：该目录存放着 Tomcat 服务器的核心类库（JAR 文件）。如果需要扩展 Tomcat 功能，也可将第三方类库复制到该目录下。
- logs：这是一个空目录，用于保存 Tomcat 每次运行时产生的日志。
- temp：保存 Web 应用运行过程中生成的临时文件。
- webapps：该目录用于自动部署 Web 应用。将 Web 应用复制在该目录下，Tomcat 会将该应用自动部署在容器中。
- work：保存 Web 应用运行过程中编译生成的.class 文件。该文件夹可以删除，但每次启动 Tomcat 服务器时，系统将再次建立该目录。
- LICENSE 等相关文档。

（3）将解压缩得到的文件夹放在任意路径下。

（4）运行 Tomcat 只需要一个环境变量：JAVA_HOME。不管是 Windows 还是 Linux，只需要增加该环境变量即可，该环境变量的值指向 JDK 安装路径。

提 示：

如果读者还不懂如何配置环境变量，说明对 Java 基础知识掌握得不够好，那么请先阅读疯狂 Java 体系的《疯狂 Java 讲义》一书的第一章。由于篇幅关系，本书将不会详细介绍如何配置环境变量的相关步骤。

注意：

此处 JAVA_HOME 环境变量应该指向 JDK 安装路径，而不是 JRE 的安装路径。JDK 安装路径下应该包含 bin 目录，该目录下应该有 javac.exe、native2ascii.exe 等程序；还包含一个 lib 目录，该目录下应该有 dt.jar 和 tools.jar 两个文件；还应该包含 jre 目录，这个 jre 目录就是 JDK 自带的 JRE。

（5）启动 Tomcat。对于 Windows 平台，只需要双击 Tomcat 安装路径下 bin 目录中的 startup.bat 文件即可。

启动 Tomcat 之后，打开浏览器，在地址栏中输入 http://localhost:8080，然后回车。浏览器出现如

图 1.7 所示界面，即表示 Tomcat 安装成功。

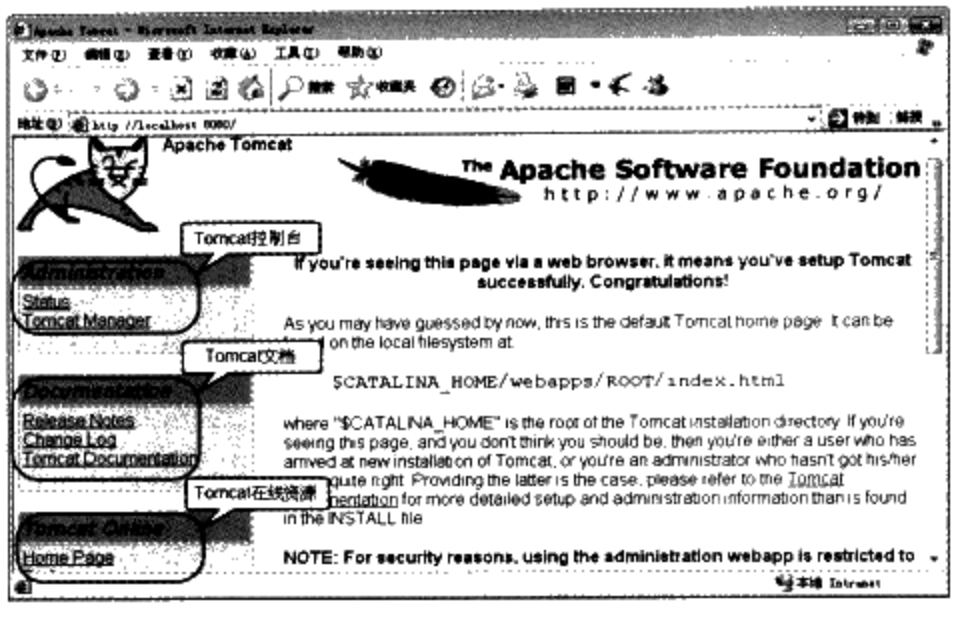


图 1.7　Tomcat 安装成功界面

Tomcat 安装成功后，必须对其进行简单的配置，这些配置包括 Tomcat 的端口、控制台等，下面详细介绍这些配置过程。

Tomcat 虽然是个免费的 Web 服务器，但也提供了图形界面控制台。通过控制台，用户可以方便地部署 Web 应用、监控 Web 应用的状态等。但对于一个开发者而言，笔者还是建议通过修改配置文件来管理 Tomcat 配置，而不是通过图形界面。

➤➤1.6.3　配置 Tomcat 的服务端口

Tomcat 的默认服务端口是 8080，可以通过管理 Tomcat 配置文件来改变该服务端口，甚至可以通过修改配置文件让 Tomcat 同时在多个端口提供服务。

Tomcat 的配置文件都放在 conf 目录下。打开 conf 下的 server.xml 文件。务必使用记事本或 vi 等无格式的编辑器，不要使用如写字板等有格式的编辑器。定位到 server.xml 文件的第 67 行处可看到如下代码：

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

其中的 8080 就是 Tomcat 提供 Web 服务的端口，可将其修改成任意的端口，建议使用 1024 以上的端口，避免与公用端口冲突。笔者将其修改为 8888，即 Tomcat 的 Web 服务的提供端口为 8888。

修改成功后，重新启动 Tomcat，在浏览器地址栏中输入 http://localhost:8888，然后回车，将再次看到如图 1.7 所示界面，即表明 Tomcat 端口修改成功。

注意：

如果需要让 Tomcat 运行多个服务，只需要复制 server.xml 文件中的<Service>元素，并修改相应的参数，便可以实现一个 Tomcat 运行多个服务，当然必须在不同的端口提供服务。

➤➤1.6.4　进入 Tomcat 控制台

在图 1.7 左上角显示有两个控制台，一个是 Status 控制台，还有一个是 Manager 控制台。Status 控制台用于监控 Web 应用的状态，而 Manager 控制台则可以部署、监控 Web 应用，因此我们通常只使用 Manager 控制台。

图 1.7 左上角的第二个链接即进入 Manager 控制台的链接，单击该链接将出现如图 1.8 所示的登录界面。

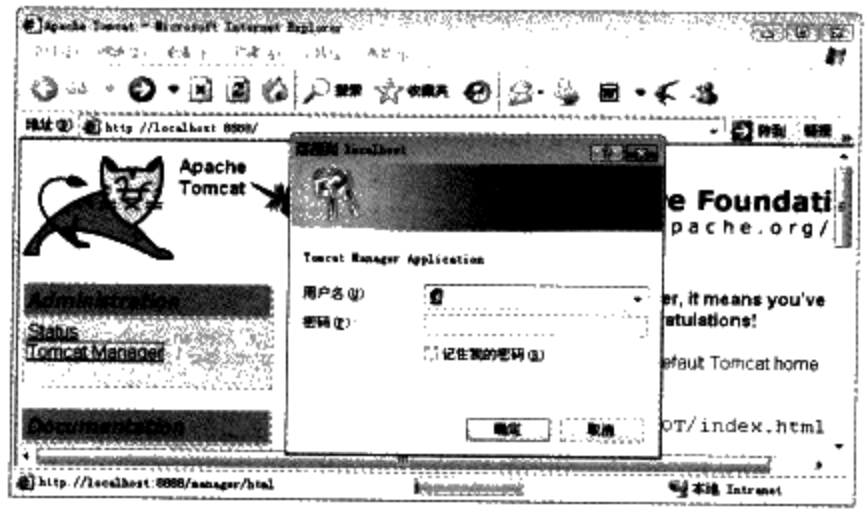


图 1.8　登录 Manager 控制台

这个控制台必须输入用户名和密码才可以登录，控制台的用户名和密码是通过 Tomcat 的 JAAS 控制的，下面介绍如何为这个控制台配置用户名和密码。

前面关于 Tomcat 目录结构的介绍已经指出：webapps 目录是 Web 应用的存放位置，而 Manager 控制台对应的 Web 应用也是放在该目录下的。进入 webapps\manager\WEB-INF，该目录下存放了 Manager 应用的配置文件，用无格式编辑器打开 web.xml 文件。

在该文件的最后部分，可看到如下配置片段：

```
<!--确定 JAAS 的登录方式-->
<login-config>
    <!-- BASIC 表明使用弹出式窗口登录 -->
    <auth-method>BASIC</auth-method>
    <realm-name>Tomcat Manager Application</realm-name>
</login-config>
<!-- 确定登录该应用所需的安全角色 -->
<security-role>
    <description>
        The role that is required to log in to the Manager Application
    </description>
    <!-- 只有 manager 角色才可以登录该应用 -->
    <role-name>manager</role-name>
</security-role>
```

通过上面的配置文件中的粗体字代码可以看出：登录 Manager 控制台需要使用 manager 角色。

为了可以登录 Manager 控制台，我们必须增加属于 manager 角色的用户。Tomcat 默认采用文件安全域，即以文件存放用户名和密码。Tomcat 的用户由 conf 目录下的 tomcat-users.xml 文件控制。打开该文件，可发现其中有：

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
</tomcat-users>
```

上面的配置文件显示 Tomcat 默认没有配置任何用户，所以无论我们在如图 1.8 所示登录对话框中输入什么内容，系统都不会让我们成功登录。为了正常登录 Manager 控制台，我们可以通过修改 tomcat-users.xml 文件来增加用户，并让该用户属于 manager 角色。Tomcat 允许通过在<tomcat-users>元素中增加<user>元素来增加用户。将 tomcat-users.xml 文件内容修改为：

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
    <!-- 增加一个用户，指定用户名、密码和角色即可 -->
    <user username="manager" password="manager" roles="manager"/>
</tomcat-users>
```

上面的配置文件中粗体字代码行增加了一个用户：用户名为 manager，密码为 manager，角色也为

manager。这样，我们就可以在如图 1.8 所示登录对话框中输入 manager、manager 来登录 Manger 控制台。成功登录后可看到如图 1.9 所示的界面。

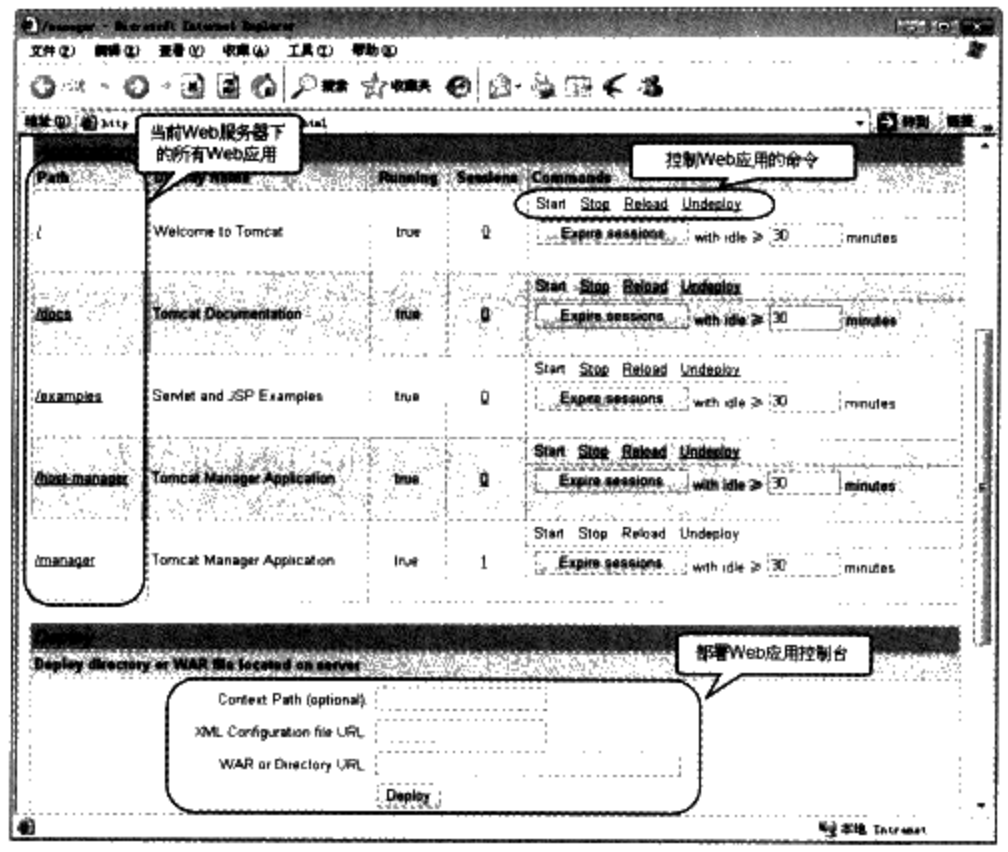


图 1.9　Tomcat 的 manager 控制台

在如图 1.9 所示的控制台中可监控到所有部署在该服务器下的 Web 应用，左边列出了所有部署在该 Web 容器内的 Web 应用，右边的四个链接则用于控制，包括启动、停止、重启等。

控制台下方的 Deploy 区则用于部署 Web 应用。Tomcat 控制台提供了两种方式来部署 Web 应用，一种是将整个目录部署成 Web 应用，另一种是将 WAR 文件部署成 Web 应用（图 1.9 中看不到这种方式，在 Deploy 区下面，还有一个 WAR file to deploy 区，用于部署 WAR 文件）。

➤➤1.6.5　部署 Web 应用

在 Tomcat 中部署 Web 应用的方式主要有：

- 利用 Tomcat 自动部署。
- 利用控制台部署。
- 增加自定义的 Web 部署文件。
- 修改 server.xml 文件部署 Web 应用。

利用 Tomcat 自动部署是最简单、最常用的方式。我们只要将一个 Web 应用复制到 Tomcat 的 webapps 下，系统就会把该应用部署到 Tomcat 中。

利用控制台部署 Web 应用也很简单。只要我们在部署 Web 应用的控制台中按如图 1.10 所示方式输入。

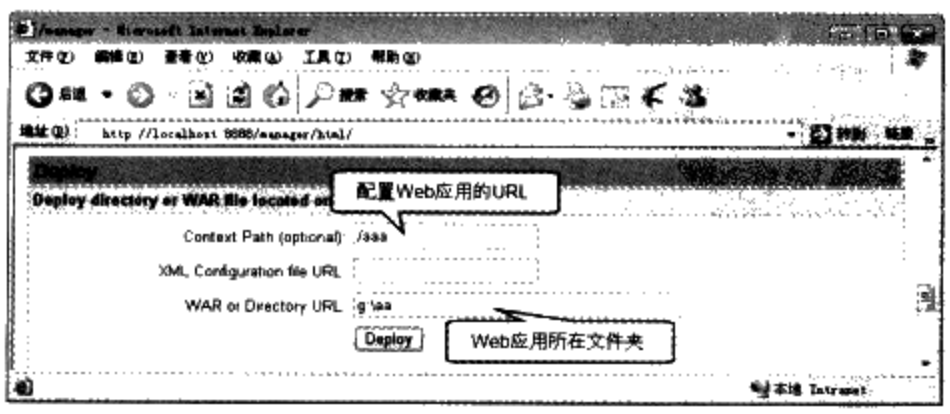


图 1.10　利用控制台部署 Web 应用

在我们按如图 1.10 所示方式输入后，单击“Deploy”按钮，将会看到 Tomcat 的 webapps 目录下多了名为 aaa 的文件夹，该文件夹的内容和 G:\下 aa 文件夹的内容完全相同——这表明：当我们利用控制台部署 Web 应用时，其实依然是利用了 Tomcat 的自动部署。

第三种方式则无须将 Web 应用复制到 Tomcat 安装路径下，只是部署方式稍稍复杂一点。我们需要在 conf 目录下新建 Catalina 目录，再在 Catalina 目录下新建 localhost 目录，最后在该目录下新建一个任意名字的 XML 文件——该文件就是部署 Web 应用的配置文件，其主文件名将作为 Web 应用的虚拟路径。例如我们在 conf\Catalina\localhost 下增加一个 dd.xml 文件，该文件的内容如下：

```
<Context docBase="G:/publish/codes/01/aa" debug="0" privileged="true">
</Context>
```

上面的配置文件中粗体字代码指定了 Web 应用的绝对路径。再次启动 Tomcat，Tomcat 将会把 G:\publish\codes\01\下的 aa 文件夹部署成 Web 应用。该应用的 URL 地址为：

```
http://<server_address>:<port>/dd
```

上面的 URL 中的 dd 就是 Web 部署文件的主文件名。

最后还有一种方式是修改 server.xml 文件，这种方式需要修改 conf 目录下的 server.xml 文件，而修改该文件可能会破坏 Tomcat 的系统文件，因此不建议采用。

➤➤1.6.6 配置 Tomcat 的数据源

从 5.5 开始，Tomcat 内置了 DBCP 的数据源实现，所以可以非常方便地配置 DBCP 数据源。

Tomcat 提供了两种配置数据源的方式，这两种方式所配置的数据源的访问范围不同：一种数据源可以让所有的 Web 应用访问，被称为全局数据源；另一种只能在单个的 Web 应用中访问，被称为局部数据源。

不管配置哪种数据源，都需要提供特定数据库的 JDBC 驱动。本书以 MySQL 为例来配置数据源，所以读者需要将 MySQL 的 JDBC 驱动程序复制到 Tomcat 的 lib 目录下。

提 示：

如果读者不了解数据库驱动程序的概念，请参阅疯狂 Java 体系的《疯狂 Java 讲义》一书。MySQL 数据库驱动可以到 MySQL 官方站点下载。

局部数据源无须修改系统的配置文件，只需修改用户自己的 Web 部署文件，因而不会造成系统的混乱。而且数据源被封装在一个 Web 应用之内，可防止被其他的 Web 应用访问，所以提供了更好的封装性。

局部数据源只与特定的 Web 应用相关，因此在该 Web 应用对应的部署文件中配置，例如为前面的 Web 应用增加局部数据源，只需修改 conf\Catalina\localhost 下的 dd.xml 文件即可，为 Context 元素增加一个 Resource 子元素，增加局部数据源后的 dd.xml 文件内容如下：

程序清单：codes\01\dd.xml

```
<?xml version="1.0" encoding="GBK"?>
<Context docBase="G:/publish/codes/01/aa" debug="0" privileged="true">
    <!-- 其中 name 指定数据源在容器中的 JNDI 名
        maxActive 指定数据源最大活动连接数
        maxIdle 指定数据池中最大的空闲连接数
        maxWait 指定数据池中最大等待获取连接的客户端
        username 指定连接数据库的用户名
        password 指定连接数据库的密码
        driverClassName 指定连接数据库的驱动
        url 指定数据库服务的 URL
    -->
    <Resource name="jdbc/dstest" auth="Container" type="javax.sql.DataSource"
            maxActive="5" maxIdle="2" maxWait="10000"
            username="root" password="32147"
            driverClassName="com.mysql.jdbc.Driver"
```

```
            url="jdbc:mysql://localhost:3306/javaee"/>
</Context>
```

上面的配置文件中粗体字标出的 Resource 元素就为该 Web 应用配置了一个局部数据源，该元素的各属性指定了数据源的各种配置信息。

提 示:

JNDI 的全称是 Java Naming Directory Inteface，即 Java 命名和目录接口，听起来非常专业，其实很简单：就是为某个 Java 对象起一个名字，例如上面 JNDI 的用途就是为 Tomcat 容器中的数据源起一个名字 jdbc/dstest，从而让其他程序可以通过该名字来访问该数据源对象。

再次启动 Tomcat，该 Web 应用即可通过 JNDI 名字来访问该数据源，下面是测试访问数据源的 JSP 页面代码片段：

程序清单：codes\01\aa\tomcatTest.jsp

```
//初始化 Context，使用 InitialContext 初始化 Context
Context ctx=new InitialContext();
/*
通过 JNDI 查找数据源，该 JNDI 为 java:comp/env/jdbc/dstest，分成两个部分
java:comp/env 是 Tomcat 固定的，Tomcat 提供的 JNDI 绑定都必须加该前缀
jdbc/dstest 是定义数据源时的数据源名
*/
DataSource ds=(DataSource)ctx.lookup("java:comp/env/jdbc/dstest");
//获取数据库连接
Connection conn=ds.getConnection();
//获取 Statement
Statement stmt=conn.createStatement();
//执行查询，返回 ResulteSet 对象
ResultSet rs=stmt.executeQuery("select * from newsinf");
while(rs.next())
{
    out.println(rs.getString(2) + "<br/>");
}
```

上面的粗体字代码实现了 JNDI 查找数据源对象，一旦获取了该数据源对象，程序就可以通过该数据源取得数据库连接，从而访问数据库。

提 示:

上面的测试代码需要读者在本机安装 MySQL 数据库，并提供一个名为 javaee 的数据库，该数据库下必须有一个名为 newsinf 的数据表，读者可以使用 codes\01 下的 test.sql 脚本来建立这些数据库对象——这些都是 JDBC 编程知识，读者可以阅读疯狂 Java 体系的《疯狂 Java 讲义》第 13 章来掌握相关知识。

上面的方式是配置局部数据源，如需要配置全局数据源，则应通过修改 server.xml 文件来实现。全局数据源的配置与局部数据源的配置基本类似，只是修改的文件不同。局部数据源只需修改 Web 应用的配置文件，而全局数据源需要修改 Tomcat 的 server.xml 文件。

注意：

使用全局数据源需要修改 Tomcat 的 conf 目录下的 server.xml 文件，所以可能导致破坏 Tomcat 系统，因而应尽量避免使用全局数据源。

1.6.7 安装 Ant

Ant 是一种基于 Java 的生成工具。从作用上来看，它有些类似于 C 编程（UNIX 平台上使用较多）

中的 Make 工具，C/C++项目经常使用 Make 工具来管理整个项目的编译、生成。

Make 使用 Shell 命令来定义生成任务，并定义任务之间的依赖关系，以便它们总是以必需的顺序来执行。

Make 工具主要有如下两个缺陷：

- Make 工具的本质还是依赖 UNIX 平台的 Shell 语言，所以 Make 工具无法跨平台。
- Make 工具的生成文件的格式比较严格，容易导致错误。

Ant 工具是基于 Java 语言的生成工具，所以具有跨平台的能力；而且 Ant 工具使用 XML 文件来编写生成文件，因而具有更好的适应性。

由此可见，Ant 是 Java 世界的 Make 工具，而且这个工具是跨平台的，并具有简单易用的特性。

提 示：

由于 Ant 具有跨平台的特性，所以编写 Ant 生成文件时可能会失去一些灵活性。为了弥补这个不足，Ant 提供了一个“exec”核心 task，这个 task 允许执行特定操作系统上的命令。

下载和安装 Ant 请按如下步骤进行：

（1）登录 http://ant.apache.org/bindownload.cgi 站点下载 Ant 最新版。笔者成书之时，Ant 的最新稳定版是 1.7.0，建议下载该版本。

虽然 Ant 是基于 Java 的生成工具，具有平台无关的特性，但考虑到解压缩的方便，通常建议 Windows 平台下载.ZIP 压缩包，而 Linux 平台则下载 GZ 压缩包。

（2）将下载到的压缩文件解压缩到任意路径，例如笔者解压缩到 D:\下，并将 Ant 文件夹重命名为 ant170。解压缩后可看到如下目录结构：

- bin：启动和运行 Ant 的可执行命令。
- docs：Ant 工具的相关文档。这些文档对学习使用 Ant 有很大的作用。
- etc：包含一些样式单文件。通常无须理会该目录下的文件。
- lib：包含 Ant 的核心类库，以及编译和运行 Ant 所依赖的第三方类库。
- LICENSE 等说明性文档。

提 示：

重命名 Ant 文件夹仅仅是为了方便、简捷，并不是必需的。读者可以像笔者一样重命名该文件夹，也可以不重命名该文件夹。

（3）Ant 的运行需要如下两个环境变量：

- JAVA_HOME：该环境变量应指向 JDK 的安装路径。如果已经成功安装了 Tomcat，则该环境变量应该已经是正确的。
- ANT_HOME：该环境变量应指向 Ant 的安装路径。

提 示：

Ant 的安装路径就是前面释放 Ant 压缩文件的路径。Ant 安装路径下应该包含 bin、docs、etc 和 lib 四个文件夹。

（4）Ant 工具的关键命令就是%ANT_HOME%\bin 下的 ant.bat，如果读者希望操作系统可以识别该命令，还应该将%ANT_HOME%\bin 添加到操作系统的 PATH 环境变量之中。

提 示：

当我们在命令行窗口、Shell 窗口中输入一条命令后，操作系统会到 PATH 环境变量所指定的一系列路径中去搜索，如果找到了该命令所对应的可执行程序，即运行该命令，否则将提示找不到命令。如果读者不嫌麻烦，愿意每次都输入%ANT_HOME%\bin\ant.bat 这样的全路径来运行 Ant 工具，则无须将%ANT_HOME%\bin 添加到 PATH 环境变量之中。

经过上面四个步骤，Ant 安装完毕，读者可以启动命令行窗口，输入 ant.bat 命令（如果读者未将%ANT_HOME%\bin 添加到 PATH 环境变量之中，则应该输入%ANT_HOME%\bin\ant.bat），则应该看到如下提示：

```
Buildfile: build.xml does not exist!
Build failed
```

如果看到上述提示信息，则表明 Ant 安装成功。

提 示：

关于 Ant 用法更深入的介绍，读者可以参考疯狂 Java 体系的《轻量级 Java EE 企业应用实战》1.6 节的内容。

➤➤1.6.8 Eclipse 的下载和安装

Eclipse 是一个免费的 IDE（集成开发环境）工具，而且并不仅限于 Java 开发，可支持多种开发语言。在免费的 Java 开发工具中，Eclipse 是最受欢迎的。

Eclipse 本身所提供的功能比较有限，但其插件大大增强了它的功能。Eclipse 的插件非常多，比如 Hibernate Synchronizer、Lomboz、MyEclipse 等。借助于这些插件，Eclipse 工具的表现相当出色。下面简单介绍 Eclipse 及其插件的安装和使用。

Eclipse 当前的最新版本是 3.4，但很多 Eclipse 插件还未支持该版本的 Eclipse，故笔者依然使用 Eclipse 3.3.2。登录 http://www.eclipse.org 站点，下载 Eclipse 3.3。Windows 平台下载 eclipse-SDK-3.3.2-win32.zip，Linux 平台下载 eclipse-SDK-3.3.2-linux-gtk.tar.gz 文件。解压缩下载得到的压缩文件，解压后的文件夹可放在任意目录。

直接双击 eclipse.exe 文件，即可看到 Eclipse 的启动界面，表明 Eclipse 已经安装成功。

Eclipse 本身的开发能力比较有限，通过插件可以大大增强它的功能。Eclipse 插件的安装方式主要可分为如下两种：

- 在线安装。
- 手动安装。

下面详细介绍 Eclipse 插件的两种安装方式。

➤➤1.6.9 在线安装 Eclipse 插件

在线安装简单方便，适合网络畅通的场景。在线安装可以保证插件的完整性，并可自由选择最新的版本。如果网络环境允许，在线安装是个较好的安装方式。

在线安装插件请按如下步骤进行：

（1）单击 Eclipse 的“Help”菜单，然后将光标移动到“Software Updates”菜单项上，单击“Software Updates”菜单项的“Find and Install”子菜单项，如图 1.11 所示。

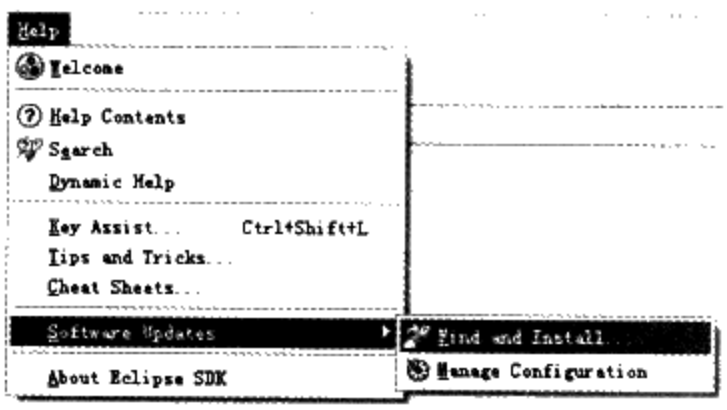


图 1.11 在线安装插件的菜单

（2）弹出如图 1.12 所示的对话框，该对话框用于选择安装新插件或升级已有插件。该对话框中有两个单选框，上面一个用于升级已有插件，下面一个用于安装新插件。

（3）如果需要升级已有插件，则选择第一个单选框，然后单击“Finish”按钮，等待 Eclipse 完成升级。

（4）如果需要安装新插件，则选择第二个单选框，此时“Next”按钮将变成可用，单击“Next”按钮，弹出如图 1.13 所示的对话框：

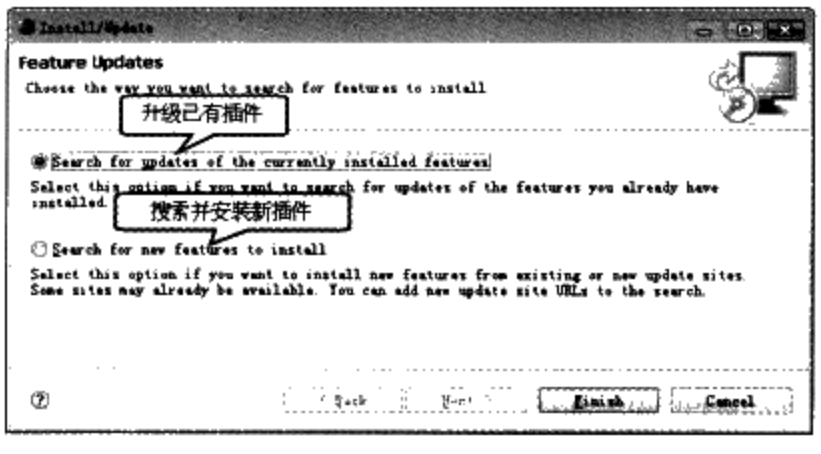


图 1.12　选择升级或安装新插件

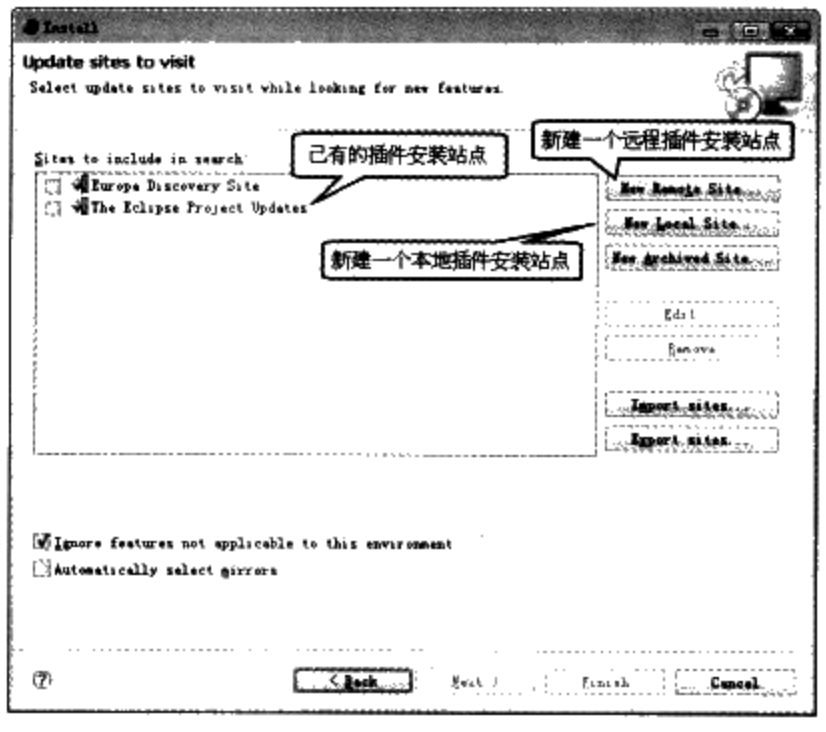


图 1.13　安装新插件

（5）图 1.13 所示窗口中空白处显示的是已经定义的插件安装项目。如需增加新的插件安装项，则单击右边的“New Remote Site”按钮，将弹出如图 1.14 所示的对话框。

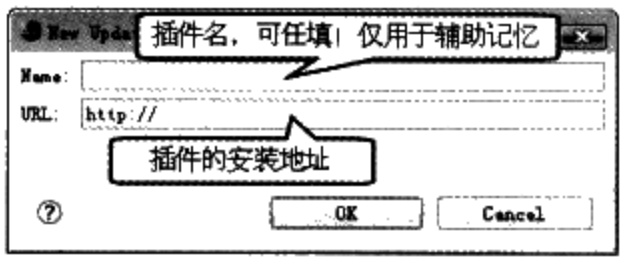


图 1.14　定义新安装地址

（6）在图 1.14 显示的对话框的 Name 文本框中输入插件名（该名称是任意的，只是用于标识该安装项），在 URL 文本框中输入插件的安装地址。输入完成，单击“OK”按钮，返回如图 1.13 所示的对话框。此时，新增的插件安装项已被添加在如图 1.13 所示的空白处。

注意：

Eclipse 插件的安装地址需要从各插件的官方站点上查询。

（7）在如图 1.13 所示的对话框中选择需要安装的插件（勾选上插件安装项之前的复选框），单击“Finish”按钮，进入安装界面。后面的过程随插件不同可能存在些许差异，但通常只需要等待即可。

➤➤1.6.10　手动安装 Eclipse 插件

手动安装只需已经下载的插件文件，无须网络支持。手动安装适合于没有网络支持的环境。手动安装的适应性广，但需要开发者自己保证插件版本与 Eclipse 版本的兼容性。

手动安装又可分为两种：

- 直接安装。
- 扩展安装。

直接安装

将插件中包含的 plugins 和 features 文件夹直接复制到 Eclipse 的 plugins 和 features 文件夹内，重新启动 Eclipse 即可。

直接安装简单易用，但效果非常不好，因为容易导致混乱。如果安装的插件非常多，可能导致用户无法精确判断哪些是 Eclipse 默认的插件，哪些是后来扩展的插件。

如果需要停用某些插件，则需要从 Eclipse 的 plugins 和 features 文件夹内删除这些插件的内容，因而卸载复杂。

扩展安装

通常推荐使用扩展安装，扩展安装请按如下步骤进行：

（1）在 Eclipse 安装路径下新建 links 文件夹。

（2）在 links 文件夹内建立 xxx.link 文件。该文件的文件名是任意的，但为了有较好的可读性，通常推荐该文件的主文件名与插件名相同，文件名后缀为.link。

（3）编辑 xxx.link 的内容，该文件内通常只需如下一行：

```
path=<pluginPath>
```

上面的内容中“path=”是固定的，而<pluginPath>是插件的扩展安装路径。

（4）在 xxx.link 文件中<pluginPath>所指的路径下新建 eclipse 文件夹，再在该 eclipse 文件夹内建立 plugins 和 features 文件夹。

（5）将插件中包含的 plugins 和 features 文件夹的内容，复制到上面建立的 plugins 和 features 文件夹中，重启 Eclipse 即完成安装。

扩展安装方式使得每个插件都放在单独的文件夹内，因而结构非常清晰。如果需要卸载某个插件，只需将该插件对应的.link 文件删除即可。

1.7 调试 JavaScript 脚本

虽然大部分用户都习惯于使用 Internet Explorer 作为上网的浏览器，但作为 Ajax 开发者，笔者宁愿选择 Firefox 作为浏览器。在使用 Internet Explorer 调试 JavaScript 时，常常感到力不从心。假设有如下 JavaScript 代码：

程序清单：codes\01\a.html

```
<script>
    alert(test);
</script>
```

上面的 JavaScript 脚本中的粗体字代码有错：因为该脚本中不曾定义变量 test。当我们用 Internet Explorer 来执行上面的 JavaScript 脚本时，浏览器不会出现任何错误——但执行该脚本也看不到弹出消息框。

当然，对于这样简单的 JavaScript 脚本，我们几乎无须使用任何调试工具，直接通过肉眼观察即可。但当我们面对几百行甚至几千行的 JavaScript 脚本时，我们就需要调试工具的帮助了——如果调试工具能报告程序错误的位置、错误的原因，将可以大大降低调试的难度，而 Firefox 就具有这样的功能。

提 示：

正如笔者常说的，程序开发中最可怕的错误并不是看到系统出现几百个错误提示，最可怕的错误是“没有错误”！如果程序编译、运行时都没有任何错误，只是运行结果与我们期望的不一致或者运行效率很低，这样的错误才比较棘手。

下载和安装 Firefox 与安装普通 Windows 程序完全一样，故此处不再赘述。

使用 Firefox 运行上面的 a.html 页面，Firefox 也不会弹出消息对话框——这一点与 Internet Explorer 完全相同。读者可以单击 Firefox 的“工具”菜单，并单击该菜单中的“错误控制台”菜单项，如图 1.15 所示。

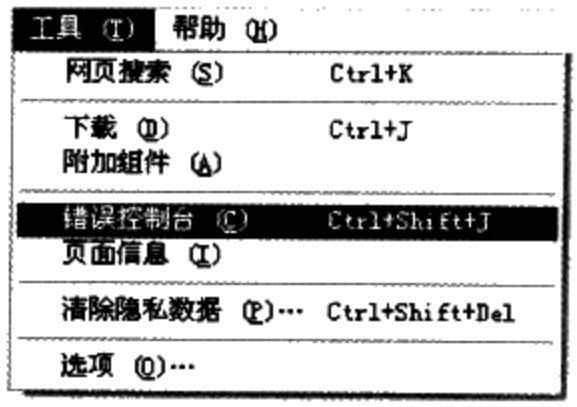


图 1.15　Firefox 的“错误控制台”菜单项

单击如图 1.15 所示的“错误控制台”菜单项之后，Firefox 将弹出如图 1.16 所示的“错误控制台”窗口，该窗口显示了 JavaScript 脚本的错误描述和错误发生位置。

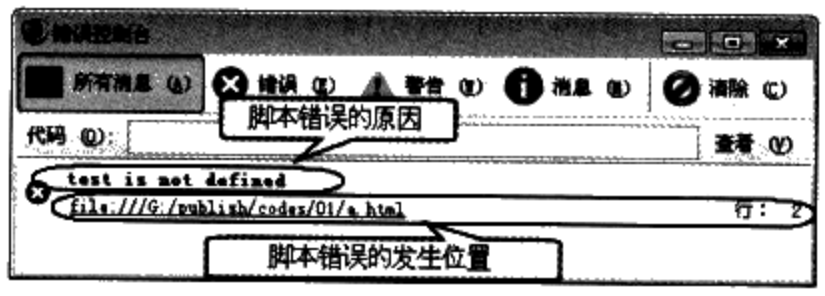


图 1.16　Firefox 的“错误控制台”窗口

开发者可以单击如图 1.16 所示的错误发生位置，Firefox 将打开脚本源代码，并将光标定位到错误发生位置，高亮显示脚本发生错误的代码行，如图 1.17 所示。

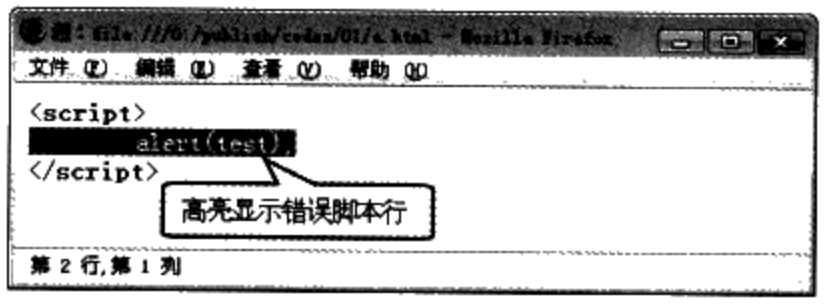


图 1.17　Firefox 的源代码浏览窗口

通过如图 1.16 和图 1.17 所示窗口可以看出，当 JavaScript 脚本发生错误时，使用 Firefox 调试会更加方便。

1.8　本章小结

本章简单介绍了应用程序的发展历史，介绍了 C/S、B/S 结构应用的优缺点。通过对比，说明了为什么 B/S 结构的应用会取代 C/S 结构的应用。同时也介绍了 B/S 结构应用面临的问题，从而引入 Ajax 技术：Ajax 技术正是为了完善传统 B/S 结构应用而出现。本章重点介绍了 Ajax 技术的发展、特征，以及组成 Ajax 技术的基本技术。本章指出了本书的适用范围：在 Java EE 应用中整合 Ajax 技术。本章还介绍了如何搭建 Ajax+Java EE 的整合开发环境，并简单介绍了如何使用 Firefox 的错误控制台来调试 JavaScript 脚本。

下一章将通过一个实际的 Ajax 应用，让读者体会到 Ajax 应用的魅力，从而让读者感受到 Ajax 技术对传统 Web 应用的改进，让读者体会 Ajax 对传统 B/S 架构应用的改进。

CHAPTER 2

第2章 Ajax 初体验

本章要点

- Ajax 技术带来的改变
- Ajax 技术的优势
- 体验 Ajax 聊天室的便捷
- 开发 JSP 聊天室
- 开发 Ajax 聊天室
- 发送异步请求
- 使用 Servlet 生成文本响应
- 使用 JSP 生成文本响应
- 获取服务器的响应内容
- 通过 DOM 加载服务器响应
- Ajax 编程的技术难点
- Ajax 和传统 Web 应用的对比

虽然 Ajax 是个很新的名词，但它并不是一种全新的技术。正如前面介绍的，Ajax 所使用的 JavaScript、CSS、DOM 对象等早已存在。Ajax 通过这些传统的对象来改善用户的交互体验，让用户能异步发送请求：用户可以在浏览页面的同时，向服务器发送请求。

JavaScript、CSS、DOM 都是相当老的技术，它们以前被称为 DHTML，即动态 HTML。DHTML 可以用来创建交互性很强的页面，但它的致命弱点在于：无法与服务器通信，无法异步发送请求。因此，虽然 DHTML 提供了非常漂亮的用户界面，但频繁的页面刷新限制了它的使用。Ajax 加入了 XMLHttpRequest 对象，这个对象提供了与服务器交互的能力，可以异步发送请求，提供了与服务器异步通信的能力，无须独占用户在页面上执行的操作。因而，Ajax 可给用户一种全新的体验。

2.1 Ajax 带来的优势

Ajax 技术的出现，改变了传统 Web 应用的模式。Ajax 技术既是对传统 Web 应用的完善，也是对传统 Web 应用的革命。Ajax 技术采用异步发送请求的方式，代替采用表单提交来更新 Web 页面的方式，从而揭开了无刷新动态更新页面时代的序幕。Ajax 可以被看做是 Web 应用开发史上的一块里程碑，因而很多地方将 Ajax 技术称为 Web 2.0 技术（当然，Web 2.0 还包含了其他的一些技术）。下面是一个关于级联菜单的应用场景，所谓级联菜单就是如图 2.1 所示的菜单。

图 2.1 级联菜单示范

级联菜单右边菜单的菜单项会根据左边菜单的改变而改变。随着左边菜单的改变，右边的菜单需要级联改变。如图 2.2 所示，当左边选中英国时，右边的下拉菜单也随之改变。

图 2.2 级联菜单的改变

在 Ajax 技术出现之前，客户端只能通过提交表单或者采用地址栏输入 URL 的方式来发送 HTTP 请求。不管采用哪种形式发送请求，都将导致页面被重新加载。在这种情况下，当用户选择第一个下拉列表框的某个选项时，程序无法即时向服务器发送请求——否则，将导致该页面被重新加载。

为了实现当用户选择第一个下拉列表框的某个选项时，第二个下拉列表框的选项随之改变，页面

必须在一开始就加载第二个列表框所有可能出现的选项。通常会一次性将第二个下拉列表框的所有数据全部读取出来并写入数组，然后根据用户的操作，通过 JavaScript 控制显示对应的列表项。

提 示：

读者可以参考 codes\02\2.1\cascadeMenu.htm 文件代码，查看该文件的 JavaScript 代码可发现，该页面里已经包含了第二个列表框所有可能出现的选项。

这可能不是实际想看到的结果：假设第一个列表框包含 200 个国家，而每个国家又包含 100 个城市。那么，该页面就需要在一开始就加载 20200（200×100+200）个选项值。也许某个浏览者属于中国，他根本不可能选择其他国家的城市，那么该页面加载的 199 个国家的城市就白白浪费了。如果更大呢？结果将更加糟糕。这样一次加载的方式将读取大量冗余数据，既增加了服务器负载，也浪费了网络带宽，更浪费了客户机的内存（浏览器 JavaScript 必须定义大数组来存放数据）。如果遇到菜单有很多级，每一级菜单又有上百个子菜单，那么这种资源的浪费将以几何级数增长。

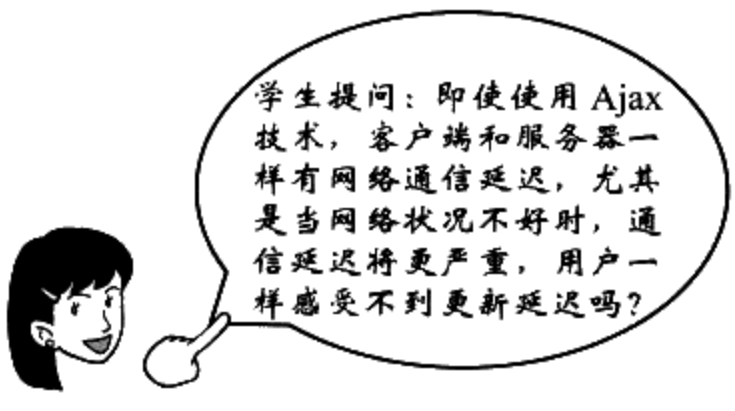
如果换成 Ajax 方案，将完全可以避免这些问题：页面无须一次加载所有的子菜单，可以在加载时只加载最左边的菜单。当用户单击了左边选择国家的下拉菜单后，异步向服务器发送请求，从服务器获取该菜单的子菜单。当用户单击第二级菜单时，再次异步向服务器发送请求，从服务器获取该菜单的全部子菜单，依此类推。通过这种方法，可避免一次加载全部菜单项，从而可提供更好的性能。

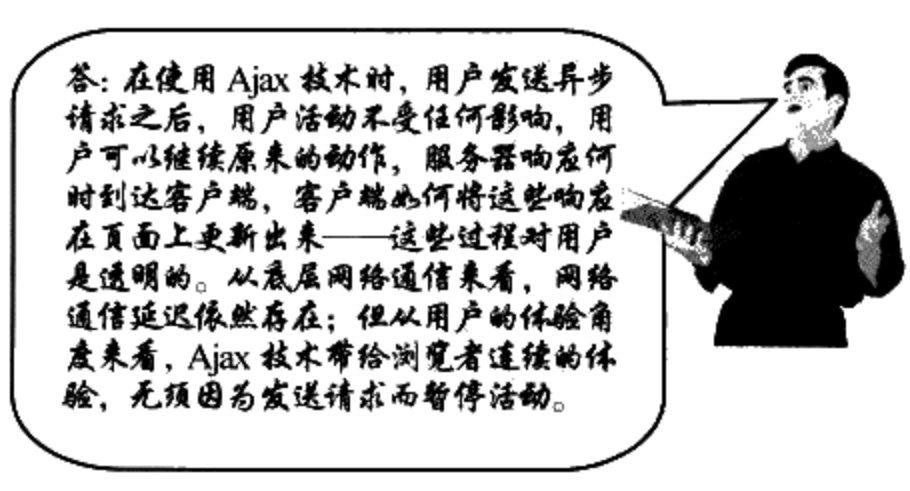
Ajax 应用特别适用于交互较多、频繁读数据、数据分类良好的 Web 应用。大体上，使用 Ajax 技术有如下优势：

- 减轻了客户端的内存消耗。Ajax 的根本理念是“按需取数据”，所以最大可能地减少了冗余请求，避免了客户端内存加载大量冗余数据。
- 无刷新更新页面。通过异步发送请求，避免了频繁刷新页面，从而减少了用户的等待时间，提供给用户一种连续的体验。
- Ajax 技术可以将传统的服务器的工作转嫁到客户端（例如购物车的状态），从而减轻服务器和带宽的负担，节约空间和带宽租用成本。
- Ajax 基于标准化技术，几乎所有浏览器都支持这种技术，无须下载插件或虚拟机程序。

“按需取数据”的模式降低了数据的实际读取量。传统的 Web 应用里，服务器的每次响应都是一个完整的页面；而基于 Ajax 技术的 Web 应用里，服务器的响应只是必须更新的数据。

如果服务器响应数据过大，传统 Web 应用将会在页面重新加载时出现白屏。由于 Ajax 采用异步的方式发送请求，页面的更新由 JavaScript 操作 DOM 完成，因而在读取数据的过程中浏览器中也不会出现白屏，而是保持原来的页面状态（也可使用 LOADING 提示框让用户了解读取状态）。在数据接收完成后，页面才开始更新部分内容，这种更新是瞬间完成的，用户甚至无法感受到更新延迟。





企业通过使用 Ajax，可以增强网站的功能，改善用户体验。用户可以通过滚动屏幕浏览大量的信息，可以更方便地将物品拖入在线购物车或者在线配置产品，而这些都无须刷新页面。事实上，相当多的企业都已开始考虑使用 Ajax 技术来改善用户的体验。

关于 Ajax 技术对传统 Web 应用的改善，下面将通过一个最简单的应用来示范 Ajax 的用法和优势，下面的聊天室应用将分别采用传统 Web 方式和 Ajax 技术加以实现，读者既可以感受到 Ajax 应用开发的大概过程，也可以作为浏览者感受到 Ajax 所带来的改善。

2.2 开始传统的 JSP 聊天室

JSP 聊天室需要实现的功能有两个：第一个功能是对用户的管理，包括用户登录、用户注册等；第二个功能是管理用户的聊天信息，系统需要保存用户最近的聊天信息。

通常情况下，系统会将用户信息、聊天信息都保存在数据库里。本应用为了简单起见，用户信息以 Properties 文件进行保存，而聊天信息只保存在内存中（使用一个 LinkedList 保存用户的聊天信息）。

本 JSP 聊天室一样遵循 MVC 的开发模式：客户端向控制器发送请求，控制器负责拦截用户请求，调用 Model 处理用户请求，控制器根据 Model 的处理结果，决定向用户呈现怎样的界面。JSP 聊天室的业务逻辑非常简单，包含三个简单用例：

- 用户注册：向保存用户名、密码的文件中增加一条记录。
- 用户登录：判断用户输入的用户名、密码是否正确，正确则允许登录聊天，否则拒绝聊天。
- 用户聊天：发送消息让所有的用户看到。

聊天室的组件结构图如图 2.3 所示。

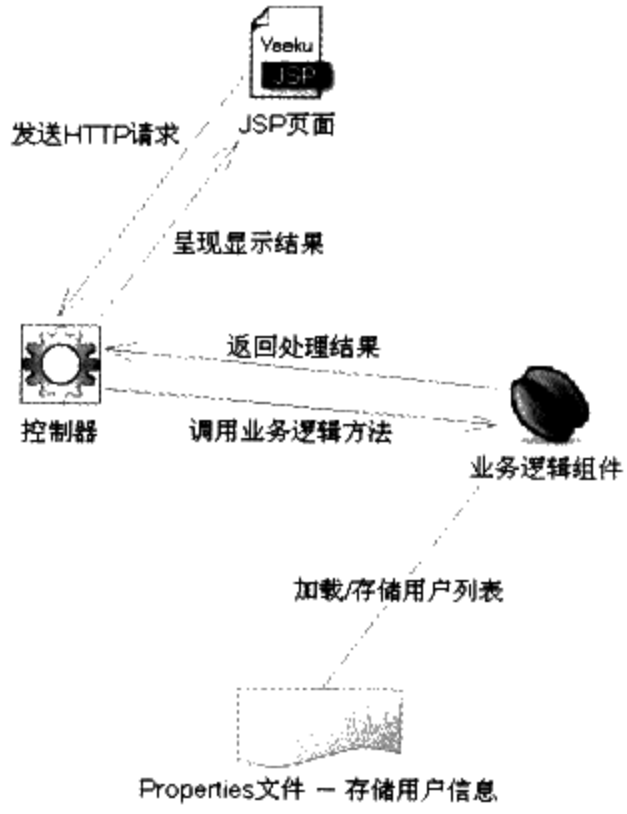


图 2.3 JSP 聊天室的组件关系图

➤➤2.2.1 实现业务逻辑组件

系统没有采用数据库存放用户信息，而是使用 Properties 文件存放用户名、密码。所有的用户登录验证、新用户注册都需要通过 Properties 文件校验。业务逻辑组件提供了如下方法用于加载属性文件：

程序清单：codes\02\2.2\jspchat\WEB-INF\src\lee\ChatService.java

```
//读取系统用户信息
private Properties loadUser() throws IOException
{
    if (userList == null)
    {
        //加载 userFile.properties 文件
```

```
            File f = new File("userFile.properties");
            //如果文件不存在，新建该文件
            if (!f.exists())
            {
                f.createNewFile();
            }
            //新建 Properties 文件
            userList = new Properties();
            //读取 userFile.properties 文件里的用户信息
            userList.load(new FileInputStream(f));
        }
        return userList;
    }
```

上面的程序中粗体字代码用于实现读取 userFile.properties 文件中的用户名、密码信息。这个方法是个工具方法，用于加载所有用户名、密码。userList 保存了当前系统中所有用户名、密码的列表，它是 ChatService 对象的实例属性，是一个 Properties 对象——其中属性名是用户名，属性值是密码。

当然，如果系统的注册用户非常多，则属性文件非常大，userList 将非常大，可能导致系统的性能下降，而采用数据库来保存用户名和密码将更合适。本范例仅为了演示传统 JSP 聊天室和 Ajax 聊天室的对比，因此此处没有使用数据库。

提 示：

使用 Properties 文件保存用户名、密码也不能处理多用户并发注册的情形。如果需要让该系统更加实用，建议改为使用数据库来保存用户名、密码。

此外，还有对应的方法用于将 userList 保存到 Properties 文件中，每次用户注册成功后都应该将新注册的用户保存到 Properties 文件中。保存 userList 的方法如下：

程序清单：codes\02\2.2\jspchat\WEB-INF\src\lee\ChatService.java

```
//保存系统所有用户
private boolean saveUserList() throws IOException
{
    if (userList == null)
    {
        return false;
    }
    //将 userList 信息保存到 Properties 文件中
    userList.store(new FileOutputStream("userFile.properties"),
        "Users Info List");
    return true;
}
```

上面的粗体字代码用于将 userList 对象中的用户名、密码信息保存到 userFile.properties 文件中。

上面的两个方法都是系统进行持久化的方法，只不过此处的持久化无须访问数据库，而只是使用 Properties 文件来保存持久化信息。业务逻辑对象必须向控制器提供的方法有：

- boolean validLogin(String user , String pass)：用于判断用户名、密码是否可以成功登录。
- boolean addUser(String name , String pass)：用于注册用户时向 Properties 文件中增加记录。
- String getMsg()：用于获取系统所保存的所有用户的聊天信息。
- void addMsg(String user , String msg)：用于增加聊天信息。聊天信息是瞬态信息，系统没有对聊天信息完成持久化，但每个用户的发言应该被增加到聊天信息里。

本系统的业务逻辑组件直接依赖上面的工具方法进行持久化，所以无须依赖持久化组件。本系统中业务逻辑组件 ChatService 的代码如下：

程序清单：codes\02\2.2\jspchat\WEB-INF\src\lee\ChatService.java

```
public class ChatService
{
    //使用单例模式来设计 ChatService
    private static ChatService cs;
    //使用 Properties 对象保存系统的所有用户
    private Properties userList;
    //使用 LinkedList 对象保存聊天信息
    private LinkedList<String> chatMsg;
    //构造器私有
    private ChatService()
    {
    }
    //通过静态方法返回唯一的 ChatService 对象
    public static ChatService instance()
    {
        if (cs == null)
        {
            cs = new ChatService();
        }
        return cs;
    }
    //验证用户的登录
    public boolean validLogin(String user , String pass)
        throws IOException
    {
        //根据用户名获取密码
        String loadPass = loadUser().getProperty(user);
        //登录成功
        if (loadPass != null
            && loadPass.equals(pass))
        {
            return true;
        }
        return false;
    }
    //新注册用户
    public boolean addUser(String name , String pass)
        throws Exception
    {
        //当 userList 为 null，初始化 userList 对象
        if (userList == null)
        {
            userList = loadUser();
        }
        //如果 userList 已经包含所需注册的用户
        if (userList.containsKey(name))
        {
            throw new Exception("用户名已经存在，请重新选择用户名");
        }
        userList.setProperty(name , pass);
        saveUserList();
        return true;
    }
    //获取系统中所有聊天信息
    public String getMsg()
    {
        //如果 chatMsg 对象为 null，表明不曾开始聊天
        if (chatMsg == null)
```

```
        {
            chatMsg = new LinkedList<String>();
            return "";
        }
        StringBuilder result = new StringBuilder("");
        //将 chatMsg 中所有聊天信息拼接起来
        for (String tmp : chatMsg)
        {
            result.append(tmp + "\n");
        }
        return result.toString();
    }
    //用户发言，添加聊天信息
    public void addMsg(String user , String msg)
    {
        //如果 chatMsg 对象为 null，初始化 chatMsg 对象
        if (chatMsg == null)
        {
            chatMsg = new LinkedList<String>();
        }
        //最多保存 40 条聊天信息，当超过 40 条之后，将前面的聊天信息删除
        if (chatMsg.size() > 40)
        {
            chatMsg.removeFirst();
        }
        //添加新的聊天信息
        chatMsg.add(user + "说: " + msg);
    }
    //下面省略了 loadUser 和 saveUserList 两个工具方法
    ...
}
```

➤➤2.2.2 实现控制器

系统的控制器使用 Servlet 充当，Servlet 负责拦截用户请求，然后调用 ChatService 对象处理用户请求，根据处理结果，将请求 forward 到合适的页面显示。本系统包含三个用例：用户注册、用户登录和用户聊天。三个用例分别对应三种请求。系统为每个请求配置一个控制器。控制器的运行结构大致相似，下面以注册所用的控制器为例进行讲解：

程序清单：codes\02\2.2\jspchat\WEB-INF\src\lee\RegServlet.java

```
public class RegServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response)throws IOException,ServletException
    {
        //设置使用 GBK 字符集来解析请求参数
        request.setCharacterEncoding("GBK");
        //取得用户的两个请求参数
        String name = request.getParameter("name");
        String pass = request.getParameter("pass");
        //进行服务器端的输入校验
        if (name == null || pass == null)
        {
            request.setAttribute("tip" , "用户名和密码都不能为空");
            forward("/reg.jsp" , request , response);
        }
        try
```

```
            {
                //调用 ChatService 对象的 addUser 方法来增加用户
                //如果注册成功
                if (ChatService.instance().addUser(name , pass))
                {
                    request.setAttribute("tip" , "注册成功，请登录系统");
                    forward("/reg.jsp" ,request , response);
                }
                //如果注册失败
                else
                {
                    request.setAttribute("tip" , "无法正常注册，请重试");
                    forward("/reg.jsp" ,request , response);
                }
            }
            catch (Exception e)
            {
                request.setAttribute("tip" , e.getMessage());
                forward("/reg.jsp" ,request , response);
            }
        }
        //执行转发请求的方法
        private void forward(String url , HttpServletRequest request,
            HttpServletResponse response)throws ServletException,IOException
        {
            //获取 RequestDispatcher 对象
            RequestDispatcher rd = request.getRequestDispatcher(url);
            //执行转发
            rd.forward(request,response);
        }
    }
```

正如上面的程序中粗体字代码所示，该 RegServlet 调用 ChatService 对象的 addUser()方法来注册新用户，也就是控制器调用业务逻辑组件方法来处理用户请求。

其余两个控制器 ChatServlet 和 LoginServlet 与此类似，ChatServlet 调用 addMsg()和 getMsg()方法来添加聊天信息和显示聊天信息。LoginServlet 则调用 validLogin()和 getMsg()方法用于验证登录和显示聊天信息。

两个控制器调用 getMsg()方法获取聊天记录后，将聊天记录放置到 HttpServletRequest 的 msg 属性中。JSP 页面则直接通过如下的表达式语言来输出聊天信息：

```
${requestScope.msg}
```

当然，为了使用该 Servlet，还需在 web.xml 文件中进行配置。相应的配置片段如下：

程序清单：codes\02\2.2\jspchat\WEB-INF\web.xml

```
<!-- 配置用户注册的 Servlet -->
<servlet>
    <servlet-name>reg</servlet-name>
    <servlet-class>lee.RegServlet</servlet-class>
</servlet>
<!-- 为用户注册的 Servlet 配置 URL -->
<servlet-mapping>
    <servlet-name>reg</servlet-name>
    <url-pattern>/reg.do</url-pattern>
</servlet-mapping>
```

经过上面的配置，RegServlet 对应的 URI 地址为 reg.do，注册表单的 action 属性可设置为 reg.do。

在提交表单时，lee. RegServlet 将调用业务逻辑组件，然后再处理用户请求。

➤➤2.2.3 实现视图

JSP 页面就是聊天室的视图，视图负责收集用户请求信息，向服务器发送请求。在发生 HTTP 请求之前，JSP 页面还可完成基本的客户端输入校验。视图还负责显示处理结果。

本聊天室有三个视图，分别对应用户登录、用户注册和用户聊天等三个用户界面。其中用户登录、用户注册两个视图非常相似，都负责收集用户名、密码，并将其发送到服务器，只是请求处理的逻辑有区别。因此两个视图只有请求的控制器不同。如图 2.4 所示是输入用户名、密码不匹配后的登录界面。

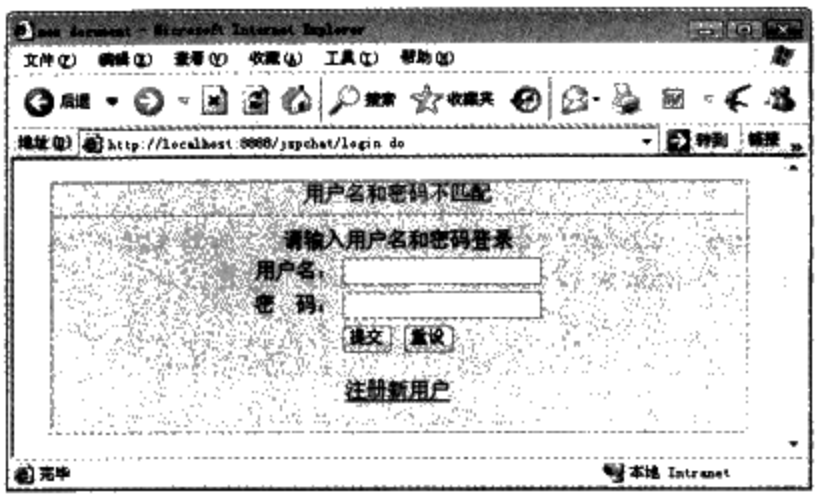


图 2.4 用户名、密码不匹配的登录界面

上面的视图除了包含基本的登录表单之外，还包含如下 JavaScript 代码用来对表单域完成客户端数据校验：

程序清单：codes\02\2.2\jspchat\index.jsp

```
<script>
    //自定义用户函数，该函数用于完成基本的客户端数据校验
    function check()
    {
        //通过 getElementById 方法获取文档中的用户名文本框对象
        var name = document.getElementById("name");
        //通过 getElementById 方法获取文档中的密码文本框对象
        var pass = document.getElementById("pass");
        var errStr = "";
        //当用户名为空时
        if (name.value == "" || name.value == null)
        {
            //添加错误提示字符串
            errStr += "用户名不能为空\n";
        }
        //当密码为空时
        if (pass.value == "" || pass.value == null)
        {
            //添加错误提示字符串
            errStr += "密码不能为空\n";
        }
        //如果错误提示字符串为空，表明用户名、密码都已经输入
        if (errStr == "" || errStr == null)
        {
            return true;
        }
        //否则弹出错误提示
        alert(errStr);
```

```
            //拒绝提交表单
            return false;
        }
        //关联表单提交与数据校验函数
        document.getElementById("loginForm").onsubmit = check;
    </script>
```

聊天界面则由一个文本域和一个文本框组成，文本框负责收集用户输入的聊天信息，文本域负责显示当前所有用户的聊天信息。聊天页面的代码片段如下：

程序清单：codes\02\2.2\jspchat\chat.jsp

```
<p align="center">
  <textarea name="textarea" cols="100" rows="30" readonly="readonly">
    ${requestScope.msg}</textarea>
</p>
<form name="form1" method="post" action="chat.do" >
  <div align="center">
    <input name="chatMsg" type="text" size="90" onclick="document.form1.submit;"/>
    <input type="submit" name="Submit" value="发送"/>
  </div>
</form>
```

上面的代码中第一行粗体字代码用于显示系统的聊天信息，第二行和第三行粗体字代码让用户输入聊天信息和发送聊天信息。除此之外，该页面也使用了 JavaScript 来提供客户端输入校验，这段 JavaScript 代码与前面的输入校验代码基本相似，此处不再赘述。

➤➤2.2.4 JSP 聊天室的问题

前面的程序已经实现了一个简单的 JSP 聊天室，但这个聊天室存在一些小小的问题。经典 B/S 结构的应用都是基于请求/响应的应用。客户端向服务器发送请求，而服务器则生成对客户端的响应。在这种结构模式里，服务器不会主动向客户端发送响应。如果客户端不发送任何请求，则即使系统的聊天信息发生改变，用户也依然看不到其他用户的聊天信息。

通过前文的介绍我们知道：用户发送请求，请求被控制器截获，控制器处理完用户请求后，将请求转发到 JSP 页面，由该 JSP 页面呈现处理结果。关键问题就在这里：每次用户发送请求后只能等待，等待服务器响应，如果服务器响应很慢，客户端浏览器将一直等待，什么事情也做不了。如果客户端想再次发送请求，则完全不可能，因为服务器没有生成响应，即客户端的浏览器是一片空白。

当用户聊天时，客户端浏览器需要不断地下载聊天页面，每次发送聊天信息后，都需要重新下载该页面。

服务器每次生成响应都是一个完整页面。实际应用中，完整页面包含的内容相当多，少则几百行，多则几千行、上万行。也许这个页面与客户端已经加载的页面大同小异，除了少量的数字需要改变，页面的其他修饰、效果、图片等都无须更新，但客户端必须重新下载这些已经下载过的资源。大量重复下载相同资源，严重占用了客户的宝贵网络带宽，也使得客户端的响应变慢。总体而言，前面的 JSP 聊天室有如下问题：

- JSP 页面无法异步发送请求，用户请求与服务器响应严格交替：用户请求→服务器响应。如果用户没有发送请求，服务器不会生成响应；如果服务器响应没有完成，用户无法再次发送请求。
- 服务器的响应总是完整 JSP 页面。大量下载重复资源。

2.3 Ajax 聊天室

针对 JSP 聊天室存在的问题，Ajax 聊天室做出了相应的提高。正如前面提到的：Ajax 并不是要取

代 B/S 结构的应用，而是更好地完善了传统的 Web 应用。

对于 JSP 存在的两个问题，Ajax 都有非常好的解决方案：Ajax 使用 XMLHttpRequest 异步发送请求，Ajax 的服务器响应仅是必须更新的数据，而不再是整个页面。JavaScript 负责将必须更新的数据加载到视图页面中。

使用 Ajax 可提高页面的复用：浏览器从服务器下载一个页面后，无须一旦发送请求就丢弃该页面而立即准备下载下一个页面——这种代价太大了，用户需要频繁下载完整页面；通过使用 Ajax 技术，请求和页面分离开来，一个视图页面可以发送多个请求，因而可以长时间使用同一个页面，故可以更好地复用一个已下载的页面。

➤➤2.3.1 异步发送请求

异步发送请求是 Ajax 最核心的内容，Ajax 的四个字符中的第一个字符就代表 Asynchronous（异步的），这也正说明了 Ajax 的核心，Ajax 使用 XMLHttpRequest 对象异步发送请求。在某种程度上，XMLHttpRequest 对象就是 Ajax 的核心，也是 Ajax 技术中唯一的新概念。Ajax 是以 XMLHttpRequest 对象为核心，结合 JavaScript、DOM、CSS 后组成的新技术。

与 JavaScript 相似的语言还有 JScript 和 ECMAScript。它们的核心语法相似，作用也相似。区别在于它们适应的浏览器以及各自的特性存在小小的区别。XMLHttpRequest 在不同浏览器中的实现也不相同，因而创建 XMLHttpRequest 对象的方法也存在区别。

关于 XMLHttpRequest 更详细的信息，请参看第 8 章“XMLHttpRequest 对象详解”。为了使用 XMLHttpRequest 对象，必须先创建 XMLHttpRequest 对象，创建该对象的代码如下：

程序清单：codes\02\2.3\ajaxchat1\chat.html

```
//创建 XMLHttpRequest 对象
function createXMLHttpRequest()
{
    if(window.XMLHttpRequest)
    {
        //DOM 2 浏览器
        XMLHttpReq = new XMLHttpRequest();
    }
    else if (window.ActiveXObject)
    {
        //Internet Explorer
        try
        {
            XMLHttpReq = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e)
        {
            try
            {
                XMLHttpReq = new
                    ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e)
            {
            }
        }
    }
}
```

上面的程序中粗体字代码可以在 Internet Explorer、Firefox、Opera（除 IE 之外的其他浏览器都会遵守 DOM 2 规范）等浏览器中创建 XMLHttpRequest 对象。因为 XMLHttpRequest 在不同的浏览器中

实现方式不同，因而在不同的浏览器中创建 XMLHttpRequest 对象的方式也略有差异。虽然上面的代码尽量兼顾了不同浏览器的实现，但不排除仍有些浏览器不支持上面的创建方法。

一旦 XMLHttpRequest 对象创建成功，系统就可以使用 XMLHttpRequest 发送请求。XMLHttpRequest 请求与传统的请求不同，传统的发送请求需要提交表单，或者请求新的网络页面——这都将导致浏览器重新发送请求，重新加载新页面。

而 XMLHttpRequest 发送请求则通过 JavaScript 代码完成，这就避免了页面的刷新——这也是异步发送请求的核心。

XMLHttpRequest 对象包含 send 方法用于发送请求。在发送请求之前，应先与请求的 URL 取得连接，XMLHttpRequest 通过 open 方法打开与请求 URL 的连接。下面是使用 XMLHttpRequest 发送请求的 JavaScript 代码：

程序清单：codes\02\2.3\ajaxchat1\chat.html

```
//发送请求函数
function sendRequest()
{
    //input 是个全局变量，就是用户输入聊天信息的单行文本框
    var chatMsg = input.value;
    //完成 XMLHttpRequest 对象的初始化
    createXMLHttpRequest();
    //定义发送请求的目标 URL
    var url = "chat.do";
    //通过 open 方法取得与服务器的连接
    //发送 POST 请求
    XMLHttpReq.open("POST", url, true);
    //设置请求头，发送 POST 请求时需要该请求头
    XMLHttpReq.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    //指定 XMLHttpRequest 状态改变时的处理函数
    XMLHttpReq.onreadystatechange = processResponse;
    //清空输入框的内容
    input.value = "";
    //发送请求，send 的参数包含许多的 key-value 对
    //即以“请求参数名=请求参数值”的形式发送请求参数
    XMLHttpReq.send("chatMsg=" + chatMsg);
}
```

上面的程序第一行粗体字代码使用 open 方法打开与请求资源的连接，因为本系统采用 POST 方式发送请求参数，因此在请求里增加了 Content-Type 请求头，并将该请求头的值设为 application/x-www-form- urlencoded，这是为了保证对请求参数采用合适的格式发送。程序中的粗体字代码是发送 POST 请求的完整过程。

一般而言，使用 XMLHttpRequest 发送请求应按如下步骤进行：

（1）使用 open 方法连接服务器 URL。

（2）调用 setRequestHeader 方法为请求设置合适的请求头。根据不同的请求，可能需要设置不同的请求头。

（3）指定回调函数。所谓回调函数就是用于检测 XMLHttpRequest 状态的函数（类似于事件监听器），当 XmlHttpRequest 的状态发生改变时，该回调函数将被触发而自动执行。

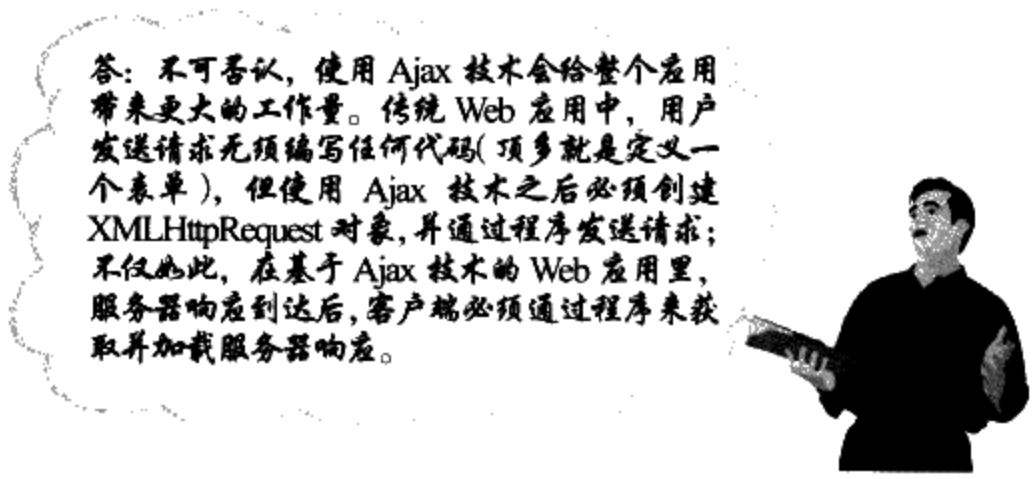
（4）调用 send 方法发送请求。

通过上面的程序我们发现，在采用 Ajax 发送请求时，发送请求比传统 Web 应用略为复杂。传统 Web 应用发送请求有两种形式：

- 在浏览器的地址栏输入请求资源后按回车发送 GET 请求。

- 提交表单发送 POST 或 GET 请求，具体发送何种请求取决于表单元素的 method 属性。

上面发送请求的方式都比较简单，基本无须编写任何程序代码。在改为使用 Ajax 请求后，我们需要先创建 XMLHttpRequest 对象，再使用该对象来发送异步请求。



➤➤2.3.2 解决多余刷新的问题

多余刷新在本聊天室中的副作用还不是十分明显，因为本系统的界面修饰相当简陋，没有多余的图片等页面资源。即使对于如此简陋的界面，一样可以对比两种模式下数据的流量。

前面的 JSP 聊天室，控制器处理用户请求后，转发到另一个 JSP 页面来显示处理结果。对于基于 Ajax 的聊天室，控制器可以不再转发请求，对于仅需要生成较少数据的响应，控制器自己生成响应数据。此时服务器响应的不再是完整的页面，而仅仅是必须更新的数据。

Ajax 主要用于改善用户体验，是一种表现层技术，并不会影响到底层的技术。对于 Java EE 应用而言，使用 Ajax 并不需要对中间层的任何组件做任何修改，更不需要对底层的 DAO 对象、Domain Object 进行修改。使用 Ajax 和使用 Hibernate、iBAITIS 或 Spring 等框架没有任何冲突，结合 Ajax 技术后的 Java EE 应用将更加完美，可以带给用户更好的体验。Ajax 也可以与 Struts、Struts 2、JSF 等框架结合使用。事实上，Struts 2、JSF 都已整合了良好的 Ajax 支持。

对于本系统而言，系统的业务逻辑组件 ChatService 没有任何改变，此处不再赘述。控制器 ChatServlet 则提供了简单的改变：对于 Ajax 系统而言，服务器响应无须是整个页面内容，可以仅是必需的数据，ChatServlet 不能将请求转发到 chat.jsp 页面。此处 ChatServlet 有两个选择：

- 直接生成简单的响应数据。
- 转向一个简单的 JSP 页面，使用 JSP 页面生成简单的响应。

本节将给出两种实现方式，通过对比两种方式，可以让读者来决定应该选择哪种实现方式。

2.3.2.1 直接使用控制器生成响应数据

在这种模式下，Servlet 直接通过 response 获取页面输出流，通过输出流生成字符响应。在这种方式下，无须转发请求，系统处理更加简单。下面是直接生成响应的 Servlet 代码：

程序清单：codes\02\2.3\ajaxchat1\WEB-INF\src\lee\ChatServlet.java

```
//聊天使用的 Serlvet，继承自 HttpServlet
public class ChatServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response)throws IOException,ServletException
    {
        //设置使用 UTF-8 字符集来解析请求参数
        //XMLHttpRequest 所发送的 POST 请求默认采用 UTF-8 字符集
        request.setCharacterEncoding("UTF-8");
        String msg = request.getParameter("chatMsg");
        //如果聊天信息不为空
```

```
        if ( msg != null && !msg.equals(""))
        {
            //取得当前用户
            String user = (String)request.getSession(true)
                .getAttribute("user");
            //调用 ChatService 的 addMsg 来添加聊天消息
            ChatService.instance().addMsg(user , msg);
        }
        //设置响应内容的类型
        response.setContentType("text/html;charset=GBK");
        //获取页面输出流
        PrintWriter out = response.getWriter();
        //直接生成响应
        out.println(ChatService.instance().getMsg());
    }
}
```

该 Servlet 是一个非常简单的 Servlet，它负责获取请求参数，调用 ChatService 对象的业务方法，输出所有的聊天记录。上面的程序中粗体字代码直接生成对客户端的响应。值得指出的是，该 Servlet 与生成 HTML 页面的 Servlet 存在的区别为：该 Servlet 没有生成任何 HTML 标签，没有生成任何页面效果，仅仅向客户端输出一个字符串。

此外，上面的代码有两个值得注意的地方：

- Ajax 使用 XMLHttpRequest 发送请求，XMLHttpRequest 发送请求时所有参数使用 UTF-8 字符集进行编码，因此 request 调用 setCharacterEncoding("UTF-8")来设置编码所用的字符集，通过如此设置才可以获取正确的请求参数。
- 生成响应时，一定要使用 response 的 setContentType 方法设置响应内容和编码方式。尤其值得指出的是：不能仅使用 response.setHeader("Charset","GBK");语句，仅使用该语句只能设置采用 GBK 字符集进行编码，但并没有指定响应内容是 HTML 页面。

对于上面的控制器而言，虽然生成了表现层内容，但并未生成完整的 JSP 页面，而是返回了模型数据，因而可以无须使用额外的 JSP 页面。

注意：

因为该响应数据是普通文本数据，而且响应数据相当简单，因而可以直接使用控制器生成客户端响应。但如果需要生成的响应非常复杂，即响应数据的数据量很大，而且具有丰富的表现格式，则应该考虑将请求转发到 JSP 页面，让 JSP 页面负责生成响应。对于服务器响应是否需要由 JSP 生成，笔者认为不可一概而论，而应取决于响应的数据量以及表现格式。

2.3.2.2 使用 JSP 页面生成响应

对于当前范例，这种做法很难说不是多此一举，控制器将请求转发到另外的 JSP 页面，而 JSP 页面仅仅负责输出聊天信息，下面是这种用法下的控制器代码：

程序清单：codes\02\2.3\ajaxchat2\WEB-INF\src\lee\ChatServlet.java

```
public class ChatServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response)throws IOException,ServletException
    {
        //设置使用 UTF-8 字符集来解析请求参数
        //XMLHttpRequest 所发送的 POST 请求默认采用 UTF-8 字符集
        request.setCharacterEncoding("UTF-8");
        String msg = request.getParameter("chatMsg");
```

```
            if ( msg != null && !msg.equals(""))
            {
                //取得当前用户
                String user = (String)request.getSession(true)
                    .getAttribute("user");
                //调用 ChatService 的 addMsg 来添加聊天消息
                ChatService.instance().addMsg(user , msg);
            }
            //将全部聊天信息设置成 request 属性
            request.setAttribute("chatList" ,
                ChatService.instance().getMsg());
            //转发到 chatreply.jsp 页面
            forward("/chatreply.jsp" , request , response);
        }
        //执行转发请求的方法
        private void forward(String url , HttpServletRequest request,
            HttpServletResponse response)throws ServletException,IOException
        {
            //获取 RequestDispatcher 对象
            RequestDispatcher rd = request.getRequestDispatcher(url);
            //执行转发
            rd.forward(request,response);
        }
    }
```

上面的 Servlet 与前一个 Servlet 基本相似，只是在 Servlet 处理用户请求结束后该 Servlet 并未直接生成响应，而是转发到 chatreply.jsp 页面，然后在 JSP 页面中输出聊天信息。该 JSP 页面代码如下：

程序清单：codes\02\2.3\ajaxchat2\chatreply.jsp

```
<%@ page contentType="text/html;charset=GBK" errorPage="error.jsp"%>
<%-- 输出当前的聊天信息 --%>
${requestScope.chatList}
```

这个 JSP 页面的作用也相当有限，仅仅完成简单的输出。由此可见，使用该 JSP 页面并不是十分有必要。

➤➤2.3.3 解析服务器响应

服务器响应生成简单的文本，而 XMLHttpRequest 包含有一个属性 responseText，该属性可获取服务器响应生成的文本。在解析服务器响应之前，必须先判断服务器响应是否完成，以及响应是否正确，例如生成状态码为 404 等的错误响应也是没有意义的。为了判断服务器响应是否完成，响应是否正确，XMLHttpRequest 同样提供了两个属性：

- readyState：判断服务器响应的状态，其中 4 表明响应完成。
- status：判断服务器响应对应的状态码，其中 200 表明响应正常，而 404 表明资源丢失，500 表明内部错误等。关于 XMLHttpRequest 的详细介绍请参考第 8 章。

判断完响应状态后，可以使用 responseText 属性获取服务器响应文本，并将该文本输出到页面显示。下面是解析、处理服务器响应的 JavaScript 代码。

程序清单：codes\02\2.3\ajaxchat1\chat.html

```
//处理返回信息函数
function processResponse()
{
    //当 XMLHttpRequest 读取服务器响应完成
    if (XMLHttpReq.readyState == 4)
    {
```

```
        //服务器响应正确（当服务器响应正确时，返回值为状态码 200）
        if (XMLHttpReq.status == 200)
        {
            //使用 chatArea 多行文本域显示服务器响应的文本
            document.getElementById("chatArea").value
                = XMLHttpReq.responseText;
        }
        else
        {
            //提示页面不正常
            window.alert("您所请求的页面有异常。");
        }
    }
}
```

上面的程序中斜体字代码先判断 XMLHttpRequest 的响应状态，当 readyState 属性为 4 时表明响应完成；再判断 XMLHttpRequest 的 status 是否为 200，200 表明服务器生成了正确的响应。

此时，浏览器的页面通过 JavaScript 与服务器进行的通信基本完成。客户端通过 sendRequest 函数向服务器发送请求，服务器通过 ChatServlet 处理用户请求，处理完用户请求后，有两种做法：Servlet 直接生成响应，或者将请求转发到 JSP 页面生成响应。

在服务器响应完成，且服务器生成了正确响应后，客户端通过 DOM 操作将服务器响应加载在视图页面上。

➤➤2.3.4 何时发送请求

虽然定义了请求发送的方法，但没有定义该何时发送请求。根据聊天室的特点，请求应该需要定时发送才行——因为即使本人没有参与聊天，他也希望看到其他人的聊天记录。但该请求与前面介绍的请求存在少许差别，这种定时发送的请求无须读取聊天记录，无须发送聊天信息。

下面的代码是这种定义发送请求的 JavaScript 代码，这种请求不发送任何请求参数。

程序清单：codes\02\2.3\ajaxchat1\chat.html

```
function sendEmptyRequest()
{
    //完成 XMLHttpRequest 对象的初始化
    createXMLHttpRequest();
    //定义发送请求的目标 URL
    var url = "chat.do";
    //发送 POST 请求
    XMLHttpReq.open("POST", url, true);
    //设置请求头，发送 POST 请求时需要该请求头
    XMLHttpReq.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    //指定 XMLHttpRequest 状态改变时的处理函数
    XMLHttpReq.onreadystatechange = processResponse;
    //发送请求，不发送任何参数
    XMLHttpReq.send(null);
    //指定 0.8 秒之后再次发送请求
    setTimeout("sendEmptyRequest()" , 800);
}
```

上面的程序中粗体字代码也用于发送请求，只是该代码发送请求时不再发送任何请求参数。值得指出的是：sendEmptyRequest 函数在最后调用了 setTimeout("sendEmptyRequest ()" , 800)，setTimeout 是 JavaScript 的计时器，该代码表示系统将在 0.8 秒后再次执行 sendEmptyRequest 函数。因此，该函数一旦开始执行，将会不断地重复执行。因为每次函数执行结束后，都将在 0.8 秒后再次调用该函数。

自动发送的请求应在进入聊天室后立即发送，可以设置页面加载后立即执行该函数，因此我们会在该页面的<body.../>元素中增加 onload ="sendEmptyRequest();"属性。

答：一定会的。这也正是笔者经常提出的一个命题：从每次服务器响应的数据量来看，Ajax技术可以降低服务器的负载；但一旦使用Ajax技术，服务器要处理的请求数量将大大增加，所以我们很难简单地说Ajax技术到底是降低了服务器负荷，还是增加了服务器负荷。仅就当前的聊天室来看，程序可以在客户端活动频率降低时，程序也降低发送请求的频率，这样可以适当减轻服务器的负荷。

此外，还有需要获取用户聊天信息和发送参数的请求。这种请求应该定义在按下“提交”按钮或在聊天文本框中按下回车时发送。要在按下回车后发送请求很简单：只需要为该按钮定义 onclick 事件即可。如需在文本框中按下回车时发送请求，则应为聊天文本框指定键盘处理函数，该函数监控文本框中所有的键盘事件，其代码如下：

程序清单：codes\02\2.3\ajaxchat1\chat.html

```
//键盘处理函数
function enterHandler(event)
{
    //获取用户单击键盘的“键值”
    var keyCode = event.keyCode ? event.keyCode
        : event.which ? event.which : event.charCode;
    //如果是回车键
    if (keyCode == 13)
    {
        sendRequest();
    }
}
```

整个聊天室 HTML 页面的代码如下：

程序清单：codes\02\2.3\ajaxchat1\chat.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta name="author" content="Yeeku.H.Lee" />
    <meta name="website" content="http://www.leegang.org" />
    <meta http-equiv="Content-Type" content="text/html; charset=GBK"/>
    <title>聊天页面</title>
</head>
<body onload ="sendEmptyRequest();">
<table width="780" border="1" align="center">
<tr>
<td><p align="center">聊天页面</p>
<p align="center">
<textarea id="chatArea" name="chatArea" cols="100"
    rows="30" readonly="readonly"></textarea>
</p>
<div align="center">
    <input id="chatMsg" name="chatMsg" type="text"
    size="90" onkeypress="enterHandler(event);"/>
    <input type="button" name="button" value="提交"
```

```
    onclick="sendRequest();"/>
</div>
</td>
</tr>
</table>
<script>
var input = document.getElementById("chatMsg");
input.focus();
var XMLHttpReq;
//创建 XMLHttpRequest 对象
function createXMLHttpRequest()
{
    if(window.XMLHttpRequest)
    {
        //DOM 2 浏览器
        XMLHttpReq = new XMLHttpRequest();
    }
    else if (window.ActiveXObject)
    {
        //Internet Explorer
        try
        {
            XMLHttpReq = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e)
        {
            try
            {
                XMLHttpReq = new
                    ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e)
            {
            }
        }
    }
}
//发送请求函数
function sendRequest()
{
    //input 是个全局变量，就是用户输入聊天信息的单行文本框
    var chatMsg = input.value;
    //完成 XMLHttpRequest 对象的初始化
    createXMLHttpRequest();
    //定义发送请求的目标 URL
    var url = "chat.do";
    //通过 open 方法取得与服务器的连接
    //发送 POST 请求
    XMLHttpReq.open("POST", url, true);
    //设置请求头，发送 POST 请求时需要该请求头
    XMLHttpReq.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    //指定 XMLHttpRequest 状态改变时的处理函数
    XMLHttpReq.onreadystatechange = processResponse;
    //清空输入框的内容
    input.value = "";
    //发送请求，send 的参数包含许多的 key-value 对
    //即以“请求参数名=请求参数值”的形式发送请求参数
```

```
        XMLHttpReq.send("chatMsg=" + chatMsg);
    }

    function sendEmptyRequest()
    {
        //完成 XMLHttpRequest 对象的初始化
        createXMLHttpRequest();
        //定义发送请求的目标 URL
        var url = "chat.do";
        //发送 POST 请求
        XMLHttpReq.open("POST", url, true);
        //设置请求头，发送 POST 请求时需要该请求头
        XMLHttpReq.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
        //指定 XMLHttpRequest 状态改变时的处理函数
        XMLHttpReq.onreadystatechange = processResponse;
        //发送请求，不发送任何参数
        XMLHttpReq.send(null);
        //指定 0.8 秒之后再次发送请求
        setTimeout("sendEmptyRequest()", 800);
    }

    //处理返回信息函数
    function processResponse()
    {
        //当 XMLHttpRequest 读取服务器响应完成
        if (XMLHttpReq.readyState == 4)
        {
            //服务器响应正确（当服务器响应正确时，返回值为状态码 200）
            if (XMLHttpReq.status == 200)
            {
                //使用 chatArea 多行文本域显示服务器响应的文本
                document.getElementById("chatArea").value
                    = XMLHttpReq.responseText;
            }
            else
            {
                //提示页面不正常
                window.alert("您所请求的页面有异常。");
            }
        }
    }
    //键盘处理函数
    function enterHandler(event)
    {
        //获取用户单击键盘的“键值”
        var keyCode = event.keyCode ? event.keyCode
            : event.which ? event.which : event.charCode;
        //如果是回车键
        if (keyCode == 13)
        {
            sendRequest();
        }
    }
    </script>
    </body>
    </html>
```

通过上面的页面，基于 Ajax 的聊天室已基本完成。Ajax 聊天室的客户端请求在后台异步发送，

客户端读取服务器响应也通过 JavaScript 完成。整个过程不会阻塞用户的聊天，即使服务器的响应变慢，客户端依然可发送请求，或者查看原有的聊天记录，无须等待下载页面。如图 2.5 所示为该聊天页面的运行效果。

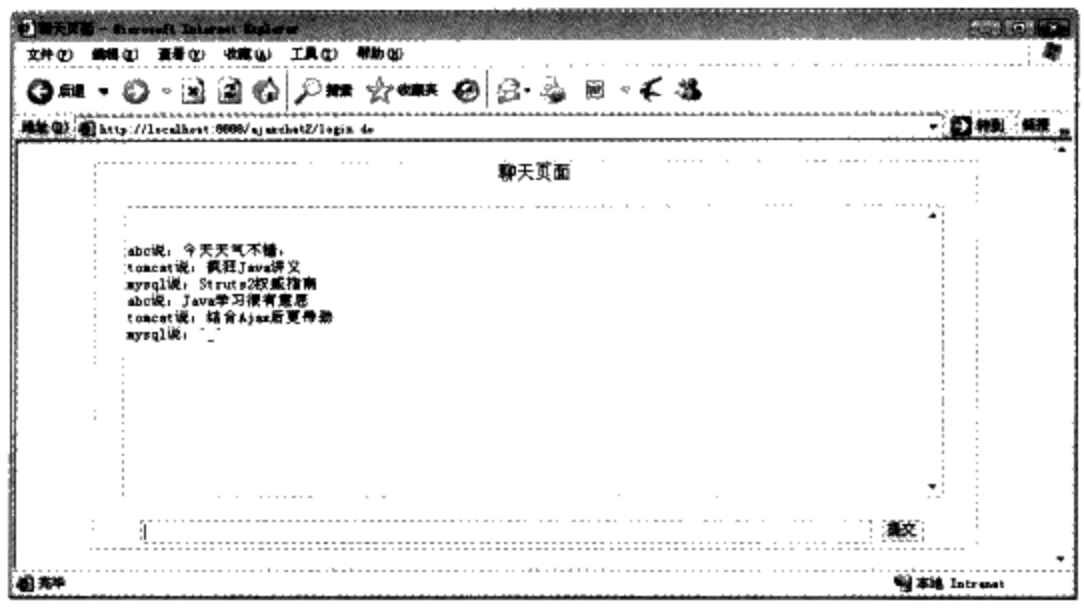


图 2.5　聊天页面的运行效果

➤➤2.3.5　Ajax 聊天室的特点

虽然 JSP 聊天室和 Ajax 聊天室的外观差不多，但用户聊天时可以体会到两者之间的区别，Ajax 聊天室不会重复下载页面，因而不会看到页面不停下载新页面。如图 2.5 所示，不管什么时候，页面的左下角都将显示“完毕”提示。

相对于传统的 JSP 聊天室而言，Ajax 聊天室的速度更快，响应更加流畅。对于复杂页面效果的情形，Ajax 的优势将更加明显：Ajax 聊天室只需从服务器获取必须更新的聊天记录，而无须下载整个页面。

Ajax 聊天室的最大特点是页面无须刷新，用户感觉不到页面的下载。使用 Ajax 聊天室时，用户感觉到仿佛在使用基于 Socket 的聊天室，虽然聊天室的页面无须刷新，但用户的聊天信息实时更新。这一切都依赖于 Ajax 的异步发送请求，动态更新页面。

2.4　Ajax 编程的技术难点

开发了上面这个简单的 Ajax 聊天室之后，下面可以分析一下 Ajax 编程和传统 Web 编程的联系和区别。

从本质上来说，Ajax 应用依然是基于请求/响应的架构，这是由 HTTP 协议所决定的，不会因为采用了 Ajax 技术而发生任何改变。不管是传统 Web 应用，还是增加了 Ajax 技术的 Web 应用，都是先由客户端发送 HTTP 请求，然后由服务器生成对客户端的响应——这个大致的流程是不会改变的。

传统 Web 编程和 Ajax 编程的区别主要在于以下三点：

客户端发送请求的方式不同

传统 Web 应用发送请求通常有两种方式：采用提交表单的方式发送 GET 请求或 POST 请求；让浏览器直接请求网络资源发送 GET 请求。在采用 Ajax 技术之后，应用需要使用 XMLHttpRequest 对象来发送请求。

服务器生成的响应不同

传统 Web 应用中服务器的响应总是完整的 HTML 页面：从<html>标签开始，然后是<head>...</head>，然后是<body>...</body>，最后由</html>结束。在采用 Ajax 技术之后，服务器响应不再是完整的 HTML 页面，而只是必须更新的数据，因此服务器生成的响应可能只是简单的文本（当然也可以是 XML 文本）。

客户端加载响应的方式不同

传统 Web 应用具有每个请求对应一个页面的关系，而且服务器响应就是一个完整的 HTML 页面，所以浏览器可以自动加载并显示服务器响应。在采用 Ajax 技术后，服务器响应的只是必须更新的数据，故客户端必须通过程序来动态加载服务器响应。

图 2.6 显示了 Ajax 应用与传统 Web 应用的主要区别。

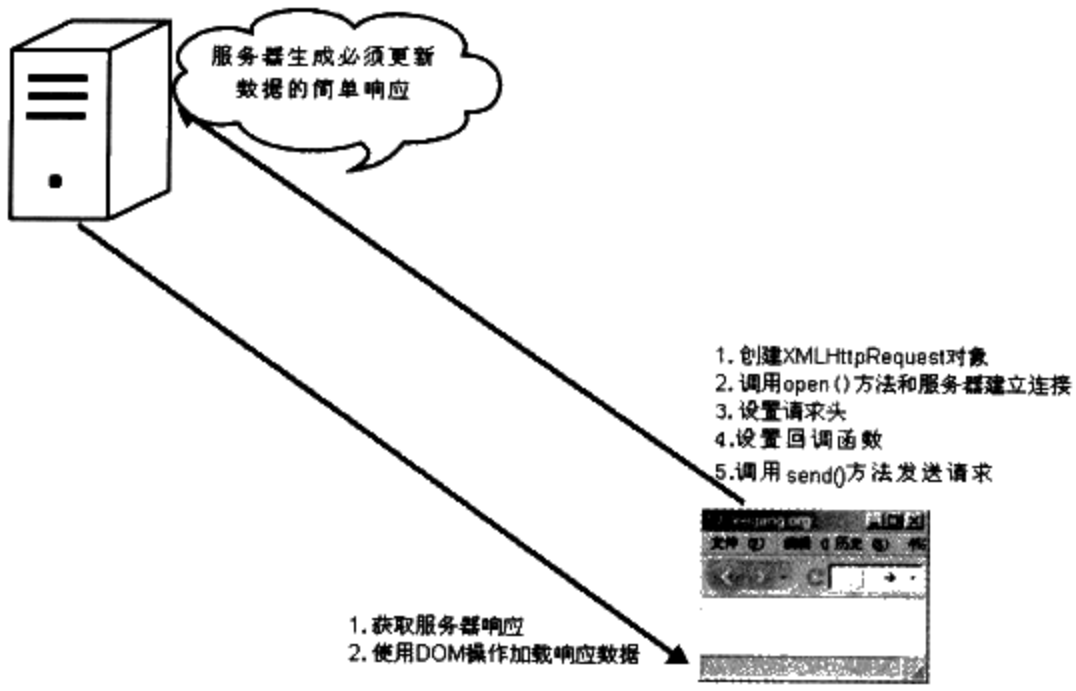


图 2.6　Ajax 应用与传统 Web 应用的主要区别

从图 2.6 可以看出，服务器端的改变最小：传统 Web 应用服务器响应为完整的 HTML 页面，而 Ajax 应用中服务器响应只是必须更新的数据——这没有任何的编程难度！甚至更简单了：因为服务器响应无须生成烦琐的 HTML 页面，只需输出简单响应即可。

Ajax 应用的编程重点在客户端也就是 JavaScript 编程，客户端的 JavaScript 编程分为两个部分：发送请求和处理响应。在客户端使用 JavaScript 发送请求时，程序按固定步骤执行类似代码即可——几乎每次都有完全类似的编程代码，因此这也不是编程难点。关键在于使用 DOM 操作加载响应数据，这才是 Ajax 编程的难点。

提　示：

后面我们会介绍一些 Ajax 框架，它们都对使用 XMLHttpRequest 发送请求提供了良好的封装，程序只需要一行代码即可成功发送 Ajax 请求，这就更突出了 Ajax 编程唯一的难点：使用 DOM 加载响应数据。有些 Ajax 框架还提供了一些简化 DOM 操作的 JavaScript 函数，但使用 DOM 加载响应数据依然是 Ajax 编程最难以处理的部分。

由此可见，一旦理解了 Ajax 技术，就会对 Ajax 技术有更清晰的认识：使用 Ajax 技术比传统 Web 应用编程略为复杂，这种复杂主要集中在如何使用 DOM 来动态加载服务器响应——因此我们需要重点掌握 JavaScript 编程和 DOM 操作。

2.5　传统 Web 应用与 Ajax 应用的对比

Ajax 技术是所谓 Web 2.0 技术的重要组成部分。Ajax 技术既是对传统 Web 技术的革命，也是对传统 Web 技术的一种改良和发展。引入 Ajax 技术后的 Web 应用，不仅改善了性能，也改善了用户体验。下面就几个方面谈谈传统 Web 应用与 Ajax 应用之间的对比：

- 用户体验方面：这是 Ajax 技术的最大改善之处。对于传统的 Web 应用，用户只能发送独占式请求，一旦请求发送出去，页面就处于等待状态，等待服务器响应完成。在服务器响应完

成之前，客户端的浏览器只能是一片空白；而 Ajax 技术则完全不同，它允许采用异步的方法发送请求，用户发送异步请求不会阻塞当前的浏览器线程，浏览器可以继续下一步操作：比如继续浏览，甚至再次发送异步请求。对于用户的体验而言，Ajax 提供了一种重大的改进：它让用户不会处于等待状态，用户感觉自己一直与应用处于交互状态，从而带给用户连续的体验。

- 响应速度：就响应速度而言，一般人会认为 Ajax 应用的速度比传统 Web 应用要快。但实际上，这种说法并不完全正确，正如前面所见到的，基于 Ajax 的应用需要大量增加 JavaScript 代码，大量增加 JavaScript 代码后的 Web 页面，在第一次加载时速度将比传统 Web 页还慢（因为必须下载大量的 JavaScript 代码）。但一旦进入该页后，响应速度可以明显提高，因为无须频繁地在各页面之间跳转，从服务器获得的仅仅是必须更新的数据。因此减少了冗余数据的下载，从而可以大幅度提高响应速度。有的人说 Ajax 包含的大量 JavaScript 代码会占用用户的大量带宽，这是相当错误的说法：通过使用 Ajax 技术，用户可以长时间复用一个视图页面。表面上看，第一次下载该页面时可能需要花费更多时间（因为包含大量的 JavaScript），但从长时间来看，传统 Web 应用则需要不停地重新下载新的 Web 页面。因此，传统的 Web 应用需要占用的网络带宽更大。
- 应用架构：传统 Web 应用主要由三层组成，而增加 Ajax 技术后的 Web 应用将在传统的 Web 应用上额外增加一个 Ajax 引擎，其实质就是一层 JavaScript 代码。这些 JavaScript 代码可以在客户端保存用户状态而无须使用 Session，能将控制器的部分功能转移到客户端页面，这必然导致安全性等各方面的问题，需要开发者认真对待。
- 开发的代码量：Ajax 技术中大部分功能都依赖于 JavaScript 语言来实现，大量的 JavaScript 代码严重降低了程序员的开发速度。JavaScript 本身不是面向对象的编程语言，这严重限制了 JavaScript 代码的可重用性。JavaScript 代码并没有一个完善的调试工具，这也加重了程序员的负担。有人调侃说，Ajax 技术是通过折磨程序员来取悦用户的技术。
- 服务器的负担：传统的看法是 Ajax 技术降低了服务器的负担，因为服务器只需要生成客户端必须更新的数据。这种说法在某些场合下也许正确。但实际的情形是，大量使用 Ajax 技术的 Web 应用，将导致服务器负担大大加重，而绝不是减轻。因为 Ajax 技术往往比传统 Web 应用需要发送更多的请求，例如对于一个自动完成的输入框：传统 Web 应用无须发送任何请求，等待用户输入即可；而 Ajax 技术的情形是，用户每输入一个字符，应用都将向服务器发送一次请求。

Ajax 技术是一种非常优秀的技术，但应该理性对待，绝不能为了 Ajax 而 Ajax，在整个应用中盲目增加大量的 Ajax 交互实际会增加服务器负荷，从而导致整个应用的性能下降。

2.6 本章小结

本章开头通过级联菜单应用的对比，介绍了采用 Ajax 技术带来的改进，详细介绍了 Ajax 带来的优势。本章重点对比了一个聊天室项目，该项目既有采用传统 Web 技术开发的案例，也有采用 Ajax 技术开发的案例。通过两个案例对比，希望读者能体会到 Ajax 带来的技术革命：Ajax 带来的不仅有应用性能上的提高，服务器负载的降低，还有对用户的体贴。

本章还讲解了 Ajax 编程的难点：关键就是 JavaScript 编程和 DOM 操作，因此本书后面章节将带着读者深入学习 Ajax 的各种相关知识，包括 XHTML、JavaScript、DOM 和 CSS 等。当然，这些知识很多都是“古老”的知识，如果读者对某些章节内容已经非常熟悉，则可跳过相应内容，直接进入下一章。

CHAPTER 3

第3章 XHTML 语言详解

本章要点

- HTML 的发展历史
- HTML 和 XHTML 的关系
- XHTML 文档结构和基本规则
- XHTML 文档和 DTD
- 基本 XHTML 标签
- XHTML 列表相关标签
- XHTML 图像相关标签
- XHTML 表格相关标签
- XHTML 框架相关标签
- XHTML 表单相关标签
- XHTML 表单控件相关标签
- XHTML 头部和元信息

HTML 的全称是 HyperText Markup Language（超文本标记语言），是互联网上应用最广泛的标记语言。不要把 HTML 和 Java、C 等编程语言混淆起来（把 HTML 想得很复杂）。HTML 是一种标记语言。简单地说，HTML 文件就是普通文本+HTML 标记（很多地方也称为 HTML 标签），而不同的 HTML 标记能表示不同效果：表格、图像、表单、声音和文字等。因此，学习 HTML 比学习 Java、C 等编程语言要简单得多。读者只需记住各种 HTML 标记，并将它们嵌入普通文本中即可。

HTML 具有免费、简单的特点，而且具有悠久的历史，并且在互联网上被广泛使用，使得 HTML 标记比较混乱：早期 HTML 标记比较少，后来很多浏览器为了达到特殊的功能，不断地扩充 HTML 标记，导致 HTML 语言逐渐庞大起来——增加这些标记很容易，但要删除它们则非常困难，因为它们已经被广泛地应用在互联网页中了。W3C 组织意识到这个问题后，制订了 HTML 3.2 规范，希望大家使用统一的 HTML 标签。如今 W3C 制订了 XHTML 1.1 规范和 HTML 4.0.1 规范（这两种规范互相兼容），这便是本章要介绍的重点。

3.1 XHTML 简介

HTML 是一种简单易用的标记语言，因为历史原因，HTML 已经走到了一个较为混乱的局面，而 XHTML 则可当成是 HTML 的升级版，它制订了更加规范、更加统一的标准。基本上，我们可以这样理解：XHTML 是更标准的 HTML。

➤➤3.1.1 HTML 的作用和历史

HTML 文档是非常简单的，简单到“写一份对的 HTML 文档很容易，但写一份错的 HTML 文档很难”，只要我们在文本文件中嵌入 HTML 标记（即使不嵌入也可以），并将其保存为一份扩展名为.htm 或.html 的文件，就可以得到一份 HTML 文档。

程序清单：codes\03\3.1\qs.htm

```
<ol>
    <li>疯狂 Java 讲义</li>
    <li>轻量级 Java EE 企业应用实战</li>
    <li>Struts 2 权威指南</li>
</ol>
```

上面的代码中粗体字代码就是被嵌入的 HTML 标签，此外就是普通文本内容。实际上，HTML 文档就是文本文件，如果用文本编辑器（包括 EditPlus、UltraEdit）来打开 HTML 文档，将看到它就是一份文本文件。在用浏览器来打开 HTML 文件时才会看到真正希望看到的效果，例如用 Firefox 打开上面的 HTML 文档，将可以看到如图 3.1 所示效果。

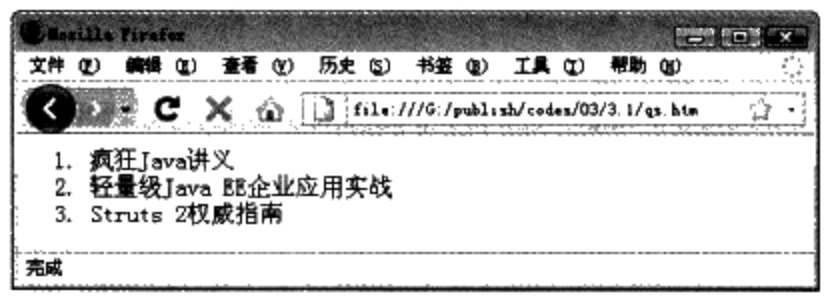


图 3.1 使用浏览器查看 HTML 文档

从图 3.1 中可以看出，
等 HTML 标记在浏览器中就可以呈现出特定效果——这就是 HTML 文档的作用：通过在文本文件中嵌入 HTML 标记，告诉浏览器如何显示页面，从而使 HTML 文件呈现出更丰富的表现效果。

在修改了 HTML 文档内容后，浏览器并不会自动更新该文档的显示，必须重新打开该文档，或者单击“刷新”按钮来重新加载该文档，从而让浏览器显示 HTML 文档最新的改变。

在 HTML 语言的发展历史中，大致经历了如下阶段：

HTML（第一版）：1993 年 6 月由互联网工程工作小组作为工作草案发布。

HTML 2.0：1995 年 11 月作为 RFC 1866 发布。

HTML 3.2：1996 年 1 月 14 日由 W3C 组织发布，是 HTML 文档第一个被广泛使用的标准。

HTML 4.0：1997 年 12 月 18 日由 W3C 组织发布，也是 W3C 推荐标准。

HTML 4.01：1999 年 12 月 24 日由 W3C 组织发布，是 HTML 文档另一个重要的、被广泛使用的标准。

XHTML 1.0：发布于 2000 年 1 月 26 日，是 W3C 推荐标准，后来经过修订于 2002 年 8 月 1 日重新发布。

在 HTML 3.2 之前，HTML 的发展极为混乱，各软件厂商经常自行增加 HTML 标记，而各浏览器厂商为了保持最好的兼容性，总是尽力支持各种 HTML 标记。在 HTML 的发展历史中，最广为人知的是 HTML 3.2 和 HTML 4.01，HTML 4.01 和 XHTML 1.0 相互兼容。我们现在编写 HTML 文档时，都应该遵守 HTML 4.01 规范。

➤➤3.1.2 HTML 4.01 和 XHTML

XHTML 的全称是（eXtensible HyperText Markup Language，即可扩展超文本标记语言），XHTML 和 HTML 4.01 具有很好的兼容性，而且是更严格、更纯净的 HTML。前面已经讲过了，由于 HTML 已经发展到了一种极度混乱的程度，所以 W3C 组织制订了 XHTML，它的目标就是逐步取代原有的 HTML。简单地说，XHTML 就是最新版本的 HTML 规范。

提 示：

由于互联网上存在大量不规范的 XHTML 网页——也就是依据早期的 HTML 3.2 规范编写的网页，因此 XHTML 取代 HTML 不可能一蹴而就，必须随着时间推移，让 HTML 逐渐退出互联网。现在我们重新编写互联网页时，应该遵循最新的规范 XHTML，这样可以保持较好的向后兼容性。

习惯上我们认为 HTML 也是一种结构化文档，但实际上 HTML 的语法非常自由、宽松（主要是各浏览器纵容的结果），所以即使我们有如下 HTML 代码：

程序清单：codes\03\3.1\bad.html

```
<html>
<head>
<title>混乱的 HTML 文档</title>
<body>
<h1>混乱的 HTML 文档
```

上面的代码中四个粗体字标签都没有正确结束，这显然违背了结构化文档的规则，但使用浏览器来浏览这份文档时，依然可以看到浏览效果——这就是 HTML 不规范的地方。而 XHTML 则致力于消除这种不规范，它要求 HTML 文档首先必须是一份 XML 文档。

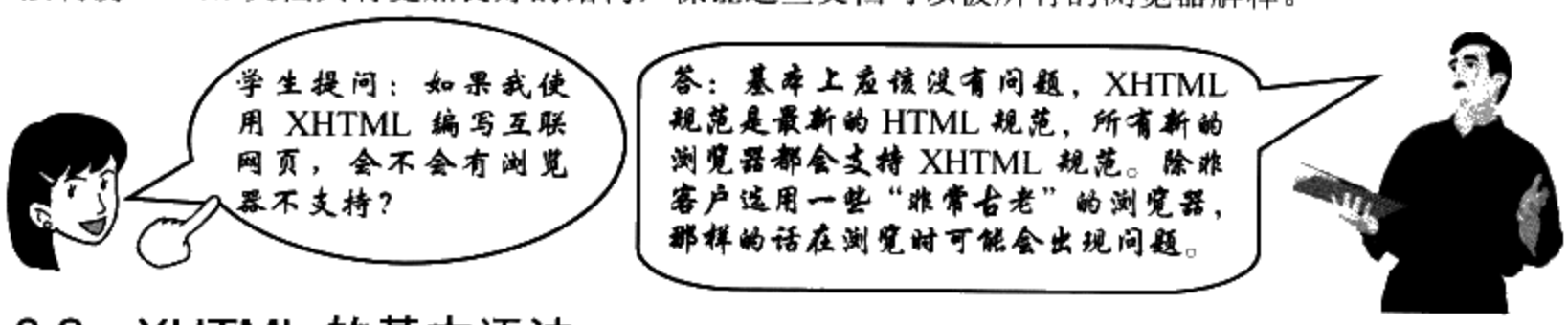
XML 文档是一种结构化文档，它有如下四条基本规则：

- 整个文档有且仅有一个根元素。
- 每个元素都由开始标签和结束标签组成（例如<a>和就是开始标签和结束标签），除非使用空元素语法（例如
就是空元素语法）。
- 元素与元素之间应该合理嵌套。例如<a>疯狂 Java 讲义，可以很明确地看出<b.../>元素是<a.../>元素的子元素，这就是合理嵌套；但<a>疯狂 Java 讲义这种写法就比较混乱，也就是所谓的不合理嵌套。
- 元素的属性必须有属性值，而且属性值应该用引号（单引号和双引号都可）引起来。

通常，计算机里的浏览器可以对付各种不规范的 HTML 文档，但现在很多浏览器运行在移动电话和手持设备上，它们就没有能力来处理那些糟糕的 HTML 文档。

为此，W3C 建议使用 XML 规范来约束 HTML 文档，将 HTML 和 XML 的长处加以结合，从而得到了现在和未来都能使用的标记语言 XHTML。

XHTML 可以被所有的支持 XML 的设备读取，在其余的浏览器升级至支持 XML 之前，XHTML 强制使 HTML 文档具有更加良好的结构，保证这些文档可以被所有的浏览器解释。



3.2 XHTML 的基本语法

XHTML 比 HTML 更加规范、更加严格，因而掌握了 XHTML 的知识自然也就学会了编写 HTML 文档，而且是个更加严格、更加规范的 HTML 文档。

➤➤3.2.1 XHTML 的基本结构和规则

XHTML 文档首先必须是一份 XML 文档，所以也必须遵守 XML 的四条基本规则。除了这四条基本规则，XHTML 文档还要求所有标记名全部使用小写字母！在传统 HTML 文档里经常使用的<HTML>标记，在 XHTML 文档里应该统一改为<html>。

对于一份基本的 HTML 文档而言，总有如下结构：

```
<html>
<head>
<title>页面标题</tile>
<!-- 此处还可插入 meta、样式单等信息 -->
</head>
<body>
页面内容部分
</body>
</html>
```

从上面的代码中可以看出，XHTML 文档的根元素是<html.../>，这是固定不变的内容。在<html.../>元素里包含<head.../>和<body.../>两个子元素。<head.../>元素主要定义 XHTML 文档的页面头，其中的<title.../>元素用于定义页面标题，此外还可在<head.../>元素中定义 meta、样式单等信息；<body.../>元

素用于定义页面主体，包括页面的文本内容和绝大部分标签。

注意：

不要在<html>和<head>之间插入任何内容！不要在</head>和<body>之间插入任何内容！不要在</body>和</html>之间插入任何内容！

XHTML 要求所有元素必须由开始标签和结束标签组成，这与以往的 HTML 规范很容易冲突，在以往的 HTML 规范中，如下标签都是正确的：

**
**：换行标签。

<hr>：水平线标签。

而在 XHTML 文档中，如上的两个标签必须写成：

**
**：换行标签。

<hr />：水平线标签。

还有，在传统的 HTML 文档中，我们要指定段落时只需要用一个<p>标记就可以了，如下：

```
<p>段落文字
```

但在 XHTML 规范中，必须写成：

```
<p>段落文字</p>
```

与 XHTML 元素类似的是，XHTML 要求所有元素的所有属性名都应该小写，所有属性都必须指定属性值，不能简写；而且所有属性值必须使用引号引起来。这也是许多传统 HTML 编写者不能适应的地方，例如下面的代码在 HTML 规范下是正确的，但在 XHTML 规范下就是错误的：

```
<!-- 属性名大写错误 -->
<table WIDTH="100%">
<!-- 属性名应全部小写 -->
<div onClick="alert('提示');">
```

传统的 HTML 规范中，很多习惯指定属性值不添加引号：

```
<!-- 属性值没有使用引号的错误 -->
<table width=100%>
```

在 XHTML 文档中应改为：

```
<table width="100%">
```

此外，许多 HTML 元素的属性允许简写，例如：

```
<input checked>
<input readonly>
<input disabled>
<option selected>
<frame noresize>
```

上面这些元素中的粗体字属性都只简写了属性名，没有指定属性值，在 HTML 规范下是正确的，但在 XHTML 规范下则应该写成：

```
<input checked="checked" />
<input readonly="readonly" />
<input disabled="disabled" />
<option selected="selected" />
<frame noresize="noresize" />
```

表 3.1 是 HTML 元素的简写属性以及在 XHTML 中的改写形式列表：

表 3.1 HTML 简写属性及 XHTML 改写形式

HTML	XHTML
compact	compact="compact"
checked	checked="checked"
declare	declare="declare"
readonly	readonly="readonly"
disabled	disabled="disabled"
selected	selected="selected"
defer	defer="defer"
ismap	ismap="ismap"
nohref	nohref="nohref"
noshade	noshade="noshade"
nowrap	nowrap="nowrap"
multiple	multiple="multiple"
noresize	noresize="noresize"

此外，在 XHTML 文档中使用空元素时，W3C 建议在斜线（/）前增加一个空格，也就是避免写
，而应该写成
，这样可以保持最好的兼容性。

➤➤3.2.2 XHTML 和 DTD

由于 XHTML 首先必须是一份 XML 文档，而 XML 文档除了需要满足基本的文档规则之外，还应该使用 DTD（Document Type Definition，即文档类型定义）或 Schema 来定义 XML 文档的语义约束，所以 XHTML 文档也应该指定合适的语义约束，XHTML 使用 DTD 来指定语义约束。

提 示:
关于 XML、DTD 和 Schema 的相关知识，请参阅疯狂 Java 体系的《疯狂 XML 讲义》。

DTD 信息应该添加在 XHTML 文档的开头部分，它是 XHTML 文档的必需部分，因此一份标准的 XHTML 文档应该有如下形式：

```
<!DOCTYPE ...>
<html>
<head> ... </head>
<body> ... </body>
</html>
```

XHTML 1.0 的三种文档类型，分别对应如下三种 DTD 声明：

XHTML 1.0 Strict

严格的 XHTML 语义约束，其 DTD 语法如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

当使用这种 DTD 语义约束时，整个 XHTML 文档需要使用干净的 XHTML 标记，避免表现上的混乱，通常需要与 CSS 结合使用。

XHTML 1.0 Transitional

传统的 XHTML 语义约束，其 DTD 语法如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

当使用这种 DTD 语义约束时，可以利用 HTML 文档在表现上的特性，并可为那些不支持 CSS 的浏览器编写 XHTML 文档。

这种 DTD 语义约束和传统的 HTML 文档保持了较好的兼容性，因而是最常用的 DTD 语义约束。

XHTML 1.0 Frameset

框架集 XHTML 语义约束，其 DTD 语法格式如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

当需要在 XHTML 页面中使用框架，将浏览器窗口分割为两个框架或更多框架时，就应该使用这种类型的 DTD 语义约束了。

当我们在 HTML 页面中使用框架集，也就是将一个浏览器窗口分成一个或多个部分时，需要为框架页面编写更多 HTML 页面。如下列框架集页面所示：

程序清单：codes\03\3.2\frame.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>框架集页面</title>
    <meta name="author" content="Yeeku.H.Lee" />
    <meta name="website" content="http://www.leegang.org" />
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
</head>
<!-- 定义框架集，无须 body 元素 -->
<frameset cols="160,*">
    <!-- 定义两个 frame，将浏览器分成 2 部分 -->
    <frame src="left.html" frameborder="1" name="leftFrame"
        scrolling="no" id="leftFrame" title="leftFrame" />
    <frame src="right.html" frameborder="1" name="mainFrame"
        id="mainFrame" title="mainFrame" />
</frameset>
</html>
```

上面的程序中前两行粗体字代码指定该页面使用框架集的 DTD 约束，后面的粗体字代码定义了一个框架集，该框架集包含左右两个部分，左边框架装载 left.html，右边框架装载 right.html。一旦指定了某个页面使用框架集，则该页面不能再包含<body.../>元素。

上面的框架页面的左右两部分分别装载 left.html 和 right.html，因此还需要为该页面定义 left.html 和 right.html 两个页面。提供了这两个页面后在浏览器中浏览 frame.html 页面将可看到如图 3.2 所示效果。

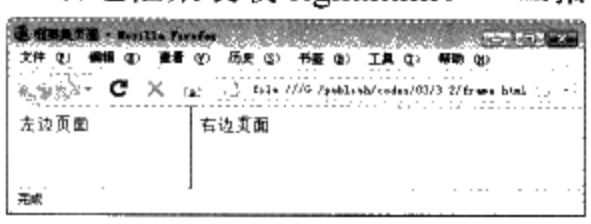


图 3.2 框架集页面

提 示：

XHTML 三种语义约束的 DTD 文件都可以在随书光盘的 codes\03\3.2 下找到。

在编写了 XHTML 页面后，也许我们不确定该页面是否足够规范，此时可以借助 W3C 组织提供的验证器来验证我们的页面，登录 http://validator.w3.org/站点即可看到 W3C 提供的标签验证页面。该页面提供了 3 个 Tab 页面，分别用于验证指定 URL 对应的网页，从本机上传文件进行验证和直接验证输入内容，如图 3.3 所示。

在输入需要验证的内容后，单击如图 3.3 所示页面中的“Check”按钮，如果验证通过，将可看到如图 3.4 所示的验证结果页面。

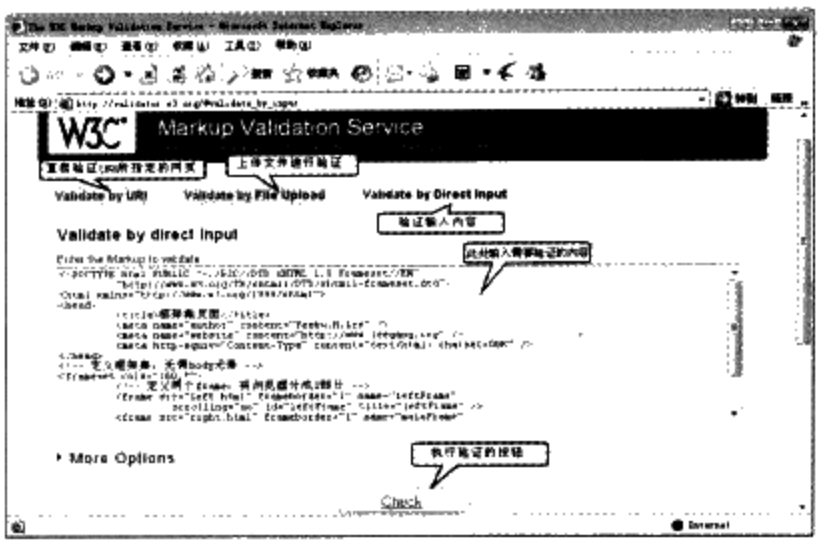


图 3.3　W3C 提供的 XHTML 验证工具

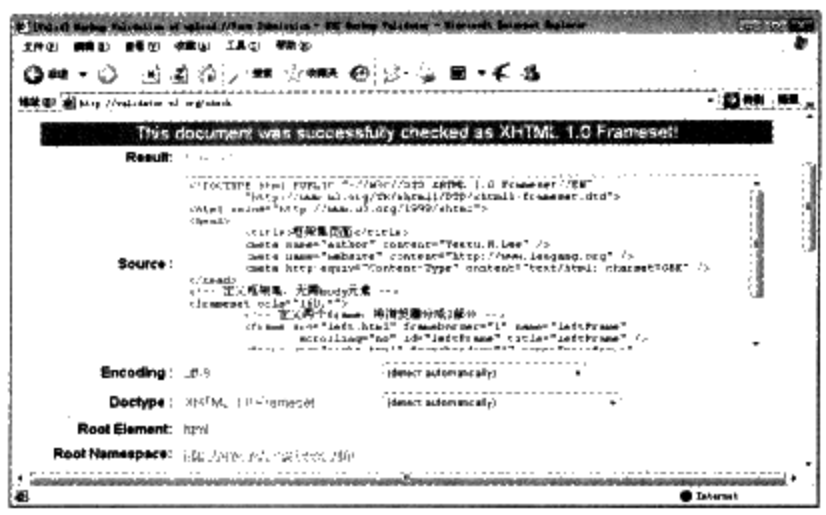


图 3.4　验证结果页面

3.3　XHTML 的常用标签

前面已经提过，学习 XHTML 语言非常简单，它只是一种标记语言，掌握 XHTML 语言的关键就是掌握常用的 XHTML 标签、各标签常用属性，以及浏览器对这些标记的解释效果。

➤➤3.3.1　基本标签

XHTML 最基本的标签有：

- <html>：它是 XHTML 文档的根元素，在 XHTML 文档中使用时可指定一个 xmlns 属性，其值只能是 http://www.w3.org/1999/xhtml。
- <body>：它用于定义 XHTML 文档的页面主体部分，该标签可以指定 id、class、style 等核心属性，还可指定 onload、onunload、onclick、ondblclick、onmousedown、onmouseup、onmouseover、onmousemove、onmouseout、onkeypress、onkeydown、onkeyup 等事件属性，这些属性用于指定 JavaScript 脚本。

注意：

关于 XHTML 元素的事件属性，请参阅本书第 7 章的内容，此处不会详细介绍这些事件属性的用法。后面介绍各标签时也不再详细列出各事件属性。

- <style>：该属性用于引入样式定义。参看本书第 5 章。
- <h1>到<h6>：定义标题一到标题六。
- <p>：定义段落，该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。

提 示：

几乎所有 HTML 元素都可指定 id、style 和 class 属性。其中 id 属性用于为 XHTML 元素指定一个唯一标识，该标识是通过 DOM 访问 XHTML 元素的重要途径，关于 id 属性的作用请参阅本书第 6 章。class 和 style 属性是 CSS 样式相关属性，关于 CSS 样式的作用和用法请参阅本书第 5 章。

-
：插入一个换行，该标签可以指定 id、class、style 等核心属性。
- <hr />：定义水平线，该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- <!--...-->：定义注释。
- ：定义文档中的节。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick

等各种事件属性。

- ：与 基本相似，区别是所定义的节默认不会换行。该标签可以指定和 相同的属性。

下面这份基本的 XHTML 文档中包含了这些标签。

程序清单：codes\03\3.3\basic.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 基本标签 </title>
</head>
<body>
    <!-- 采用标题一到标题六来输出文本 -->
    <h1>疯狂 Java 讲义</h1>
    <h2>疯狂 Java 讲义</h2>
    <h3>疯狂 Java 讲义</h3>
    <h4>疯狂 Java 讲义</h4>
    <h5>疯狂 Java 讲义</h5>
    <h6>疯狂 Java 讲义</h6>
    <!-- 输出一条水平线 -->
    <hr />
    <!-- 使用三个 span 定义三节 -->
    <span>Tomcat</span><span>Jetty</span><span>Resin</span>
    <!-- 输出换行 -->
    <br />
    <!-- 使用三个 div 定义三节 -->
    <div>Tomcat</div><div>Jetty</div><div>Resin</div>
    <!-- 使用三个 p 定义三个段落 -->
    <p>Tomcat</p><p>Jetty</p><p>Resin</p>
</body>
</html>
```

在浏览器中浏览上面的页面，可看到如图 3.5 所示效果。

从图 3.5 中可以看出，<span.../>、<div.../>和<p.../>三个元素的效果有点类似，它们都可作为其他内容的“容器”——容纳文本和其他内容。默认情况下，<span.../>元素不会导致换行，而<div.../>元素会导致换行，而<p.../>元素会产生一个段落，且段落和段落之间默认有更大的间距。

此外还有一点需要指出：<span.../>元素和<p.../>元素只能包含文本、图像、超级链接、文本格式化元素和表单控件元素等内容，<p.../>元素可以包含<span.../>元素，但<span.../> 元素不能包含<p.../> 元素；<div.../>元素除了可包含上面这些内容（包括<p.../>和<span.../>），还可以包含<h1.../>到<h6.../>、<form.../>、<table.../>、列表项元素和<div.../>元素——由此可见，<div.../>元素可以包含更多内容。

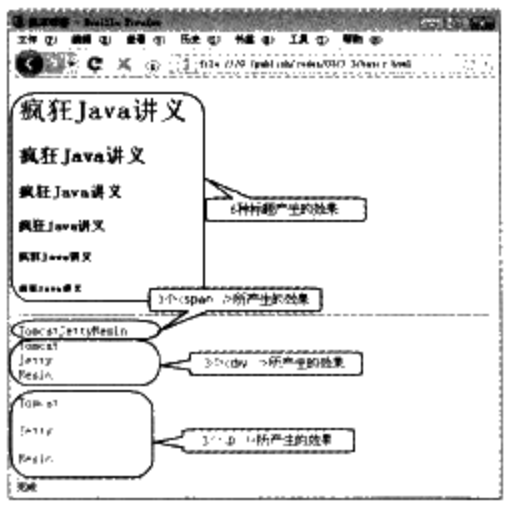


图 3.5 基本 XHTML 标签的效果

➤➤3.3.2 文本格式化标签

下面这些标签可以让文本内容在浏览器中表现出特定效果：

- <b>：定义粗体文本。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- <i>：定义斜体文本。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。

- ：定义强调文本，实际效果与斜体文本差不多。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- <big>：定义大号字体文本。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- ：定义粗体文本。与<b>标签的作用和用法基本相同。
- <small>：定义小号字体文本。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- <sup>：定义上标文本。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- <sub>：定义下标文本。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。
- <bdo>：定义文本显示的方向。该标签可以指定 id、class、style 等核心属性，还可以指定 onclick 等各种事件属性。此外，该标签还可以指定 dir 属性，该属性值只能是 ltr 或者 rtl。

上面这些文本格式化元素能包含文本、图像、超级链接、文本格式化元素和表单控件元素等内容。此外，这些元素还都可以和<span...>元素相互包含。如下 XHTML 页面示范了这些文本格式化相关标签的用法。

程序清单：codes\03\3.3\text.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 文本格式化标签 </title>
</head>
<body>
    <span><b>加粗文本</b></span><br />
    <span><i>斜体文本</i></span><br />
    <span><b><i>粗斜体文本</i></b></span><br />
    <span><em>被强调的文本</em></span><br />
    <big><span>大号字体文本</span></big><br />
    <p><strong>加粗文本</strong></p>
    <small><span>小号字体文本</span></small><br />
    <div>普通文本<sup>上标文本</sup></div>
    <span>普通文本<strong><sub>下标加粗文本</sub></strong></span><br />
    <!-- 指定文本从左向右（正常情况）排列 -->
    <bdo dir="ltr">从左向右排列的文本</bdo><br />
    <!-- 指定文本从右向左排列 -->
    <bdo dir="rtl">从右向左排列的文本</bdo><br />
</body>
</html>
```

在浏览器中浏览该页面，可看到如图 3.6 所示效果。

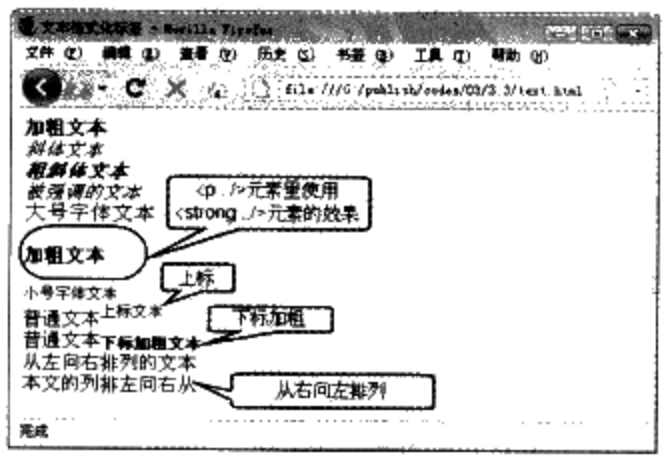
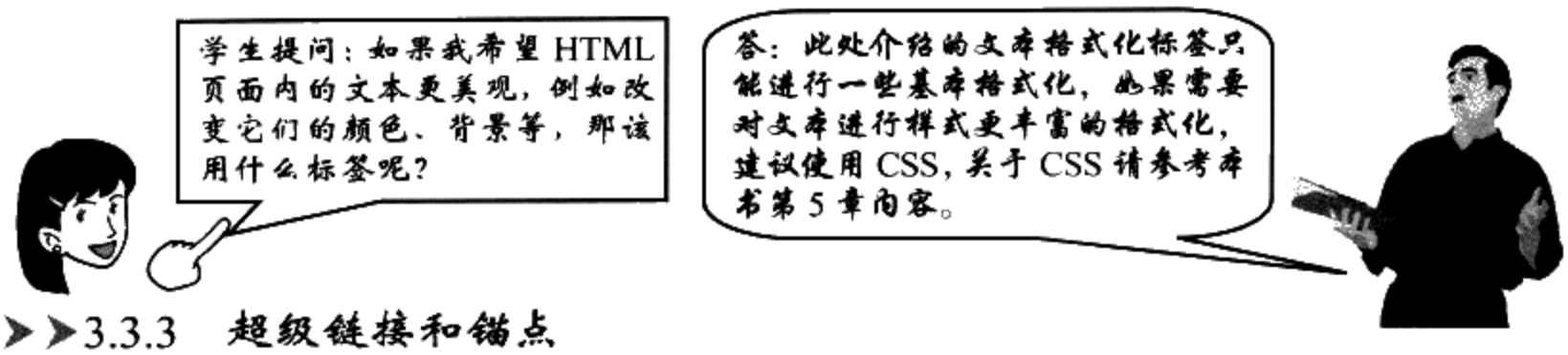


图 3.6 文本格式化标签



➤➤3.3.3 超级链接和锚点

XHTML 页面使用超级链接与网络上的另一个资源保持关联，当用户单击页面上的超级链接时，浏览器会导航到该超级链接所指的资源。

定义超级链接使用<a.../>元素，该元素可以指定 id、class、style 等核心属性，也可以指定 onclick 等各种事件属性，还可以指定如下两个重要属性：

- href：指定超级链接所关联的另一个资源。
- target：指定使用框架集中的哪个框架来装载另一个资源。该属性的属性值可以是_self、_blank、_top、_parent 等值，分别代表使用自身、新窗口、顶层框架、父框架来装载新资源。

<a.../>元素主要可以包含文本、图像、各种文本格式化元素和表单元素等内容。

下面的代码定义了四个超级链接：

程序清单：codes\03\3.3\anchor.html

```
<!-- 在本窗口打开另一个资源 -->
<a href="http://www.leegang.org"><b>疯狂 Java 联盟</b></a><br />
<!-- 在新窗口中打开另一个资源 -->
<a href="http://www.leegang.org"
    target="_blank"><em>疯狂 Java 联盟</em></a><br />
<!-- 为图像增加超级链接 -->
<a href="http://www.leegang.org"><img src="img/logo.jpg"
    alt="疯狂 Java 联盟"/></a><br />
<!-- 基于相对路径指定另一个资源 -->
<a href="text.html">文本格式化标签</a><br />
```

上面的代码定义了四个超级链接，分别是粗体字超级链接、斜体字超级链接、图像超级链接和普通超级链接，单击前三个超级链接中的任意一个，浏览器都会导航到“疯狂 Java 联盟”站点；单击最后一个链接会链接到 text.html。

上面的页面中前三个超级链接的 href 属性值为一个绝对网址，但最后一个超级链接的 href 属性值则只是一个文件名，那浏览器如何处理呢？这个文件名会被当成相对路径，浏览器会在该页面的基准路径上加上该文件名，作为此超级链接所关联的资源——于是将看到该链接实际会链接到 file:///G:/publish/codes/03/3.3/text.html（假设 anchor.html 文件放在 G:\publish\codes\03\3.3 目录下）。

在使用<a.../>元素时，href 属性值既可以是绝对路径，也可以是相对路径。指定绝对路径时，href 属性值为 URL（Uniform Resource Locator，即统一资源定位器），URL 用于对互联网上的文档（或其他资源）进行寻址。一个完整的网址，例如 http://www.leegang.org/index.php，遵守如下语法规则：

```
scheme://host.domain:port/path/filename
```

关于这个 URL 地址的解释如下：

- scheme：指定因特网服务的类型。最流行的类型是 HTTP。
- host：指定此域中的主机。如果被省略，则 HTTP 的默认主机是 www。
- domain：指定因特网域名，比如 leegang.org、crazyjava.org 等。
- port：指定主机的端口号。端口号通常可以被省略，HTTP 服务的默认端口是 80。

- path：指定远程服务器上的路径，该路径也可被省略，省略该路径则默认被定位到网站的根目录。
- filename：指定远程文档的名称。如果省略该文件名，通常会定位到 index.html、index.htm 等文件，或定位到 Web 服务器设置的其他文件。

表 3.2 显示了 URL 最流行的 scheme。

表 3.2　流行 scheme 以及对应资源

Scheme	对应资源
file	本地磁盘上的文件
ftp	远程 FTP 服务器上的文件
http	World Wide Web 服务器上的文件
news	新闻组上的文件
telnet	Telnet 连接
gopher	远程 Gopher 服务器上的文件

例如如下几个超级链接：

HTML Newsgroup：该链接将会产生一个访问新闻组资源的超级链接。

下载 Tomcat：这个链接将会产生一个指向 FTP 资源的链接。

写信给我：这个链接会产生一个邮件链接。单击该链接将会开始发送电子邮件。

此外，<a.../>元素还可以生成一个命名锚点，命名锚点用于在 XHTML 页面中生成一个定位点，这样允许超级链接直接连接到指定页面的该定位点。

插入定位锚点需要指定 name 属性，name 属性值就是该命名锚点的名称。例如如下代码：

```
<!-- 下面的代码会生成一个命名锚点 -->
<a name="test"></a>
```

用浏览器浏览命名锚点时，该命名锚点不会生成任何显示内容，我们可以使用如下超级链接来定位到该锚点：

```
<a href="anchor.html#test">定位到 test 锚点</a>
```

从上面的粗体字代码可以看出，定位到指定锚点需要在 URL 资源后指定锚点名，锚点名和 URL 资源之间以#隔开。

➤➤3.3.4　列表相关标签

XHTML 提供了如下几个列表相关标签：

- ：定义无序列表。该元素可以指定 id、style、class 等属性，还可以指定 onclick 等事件属性。该元素只能包含<li.../>子元素。
- ：定义有序列表。该元素可以指定 id、style、class 等属性，还可以指定 onclick 等事件属性。该元素只能包含<li.../>子元素。此外，在 XHTML 1.0 Transitional 的语义下，该元素还可以指定如下两个属性：
  - start：指定列表项的起始数字。默认是第一个，如 1、A 等。
  - type：指定使用哪种类型的编号，例如 1 代表使用数字，A 和 a 代表使用字母，I 和 i 代表使用罗马数字。
- ：定义列表项目。该元素可以指定 id、style、class 等属性，还可以指定 onclick 等事件属性。该元素里可包含与<div.../>完全类似的内容，因此可以包含较多类型的子元素。

- <dl>：也用于定义列表，该元素只能包含<dt.../>和<dd.../>两种子元素。该元素可以指定 id、style、class 等属性，还可以指定 onclick 等事件属性。
- <dt>：定义标题列表项。该元素可以指定 id、style、class 等属性，还可以指定 onclick 等事件属性。该元素只能包含文本、图像、超级链接、文本格式化元素和表单控件元素等。
- <dd>：定义普通列表项。该元素可以指定 id、style、class 等属性，还可以指定 onclick 等事件属性。该元素里可包含与<div.../>完全类似的内容，因此可以包含较多类型的子元素。

看如下页面代码。

程序清单：codes\03\3.3\list.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 列表项相关标签 </title>
</head>
<body>
    <!-- 定义无序列表 -->
    <ul>
        <li>疯狂 Java 讲义</li>
        <li>轻量级 Java EE 企业应用实战</li>
        <li>疯狂 Ajax 讲义</li>
    </ul>
    <!-- 定义有序列表 -->
    <ol start="2" type="I">
        <li>疯狂 Java 讲义</li>
        <li>轻量级 Java EE 企业应用实战</li>
        <li>疯狂 Ajax 讲义</li>
    </ol>
    <!-- 定义列表 -->
    <dl>
        <!-- 定义标题列表项-->
        <dt>疯狂 Java 体系</dt>
        <dd>疯狂 Java 讲义</dd>
        <dd>轻量级 Java EE 企业应用实战</dd>
        <dd>疯狂 Ajax 讲义</dd>
        <!-- 定义标题列表项-->
        <dt>作者其他图书</dt>
        <dd>Struts2 权威指南</dd>
        <dd>基于 J2EE 的 Ajax 宝典</dd>
    </dl>
</body>
</html>
```

在浏览器中查看该页面，可看到如图 3.7 所示效果。

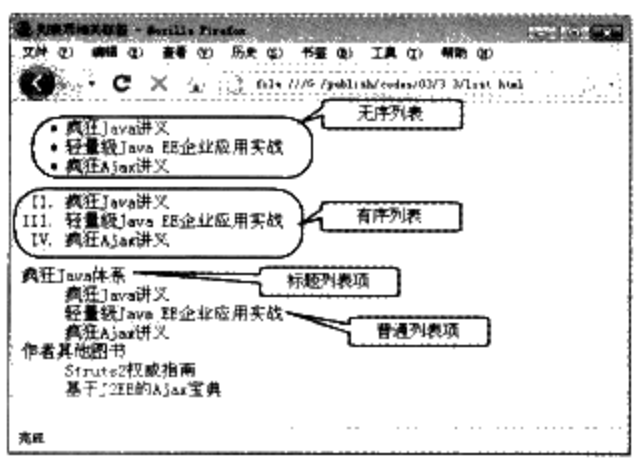


图 3.7　列表项相关标签

3.3.5　图像相关标签

XHTML 提供了<img.../>元素以便在页面中定义图像，这个元素只能是一个空元素，它不可以包含任何内容。该元素除了可以指定 id、style、class 等核心属性外，也可以指定 onclick 等事件属性。不仅如此，使用该元素必须指定如下两个属性：

- src：用于指定图片文件的所在位置，该属性值既可以是相对路径，也可以是绝对路径。
- alt：用于指定一段文本，该文本将作为该图片的提示信息。

此外，该元素还可以指定如下两个可选属性：

- height：用于指定该图像的高度，该属性值可以是百分比，也可以是像素值。
- width：用于指定该图像的宽度，该属性值可以是百分比，也可以是像素值。

此外，还有如下两个与图像相关的标签：

- <map>：用于定义图像映射。该元素主要可包含一个或多个<area.../>子元素，每个<area.../>子元素定义一个区域，不同区域可链接到不同 URL。
- <area>：用于定义图像映射的内部区域。该元素只能是一个空元素，除了可以指定 id、style、class 等核心属性外，也可以指定 onclick 等事件属性，还可以指定 onfocus、onblur 等焦点相关属性。此外，还可以指定如下属性：
  - shape：指定该内部区域是哪种区域，默认值是"rect"，即矩形区域；此外，还可以是 circle 和 poly，分别代表圆形区域和多边形区域。
  - coords：可指定多个坐标值，用于确定区域位置。
  - href：用于确定该区域所链接的资源。
  - alt：用于指定一段文本，该文本将作为该图片区域的提示信息。
  - target：用于指定使用框架集中的哪个框架来装载另一个资源。其值可以是_self、_blank、_top、_parent 等，分别代表使用自身、新窗口、顶层框架、父框架来装载新资源。

一旦我们使用<map.../>元素定义了图像映射，就可以让指定图片使用该图像映射，通过为<img.../>元素指定 usemap 属性让该图片使用图像映射，设置 usemap 属性值为#mapname 即可。

示例页面代码如下。

程序清单：codes\03\3.3\img.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 图片相关标签 </title>
</head>
<body>
<img src="img/logo.jpg" alt="疯狂 Java 的 Logo" /><br />
<!-- 定义图片，指定高、宽 -->
<img src="img/logo.jpg" width="300" height="120"
    alt="疯狂 Java 的 Logo" /><br />
<!-- 定义图片，使用指定的图片映射 -->
<img src="img/logo.jpg" width="300"
    height="120" border="0" usemap="#test"
    alt="疯狂 Java 的 Logo" /><br />
<!-- 定义图片映射 -->
<map name="test" id="test">
    <!-- 为该图片映射定义 2 个区域 -->
    <area shape="circle" coords="57,55,25"
        href="http://www.leegang.org" alt="leegang.org" />
    <area shape="poly" coords="188,28,185,50,200,74,224,72,246,51"
        href="http://www.crazyjava.org" alt="crazyjava.org" />
</map>
</body>
</html>
```

在浏览器中浏览该页面，可看到如图 3.8 所示效果。

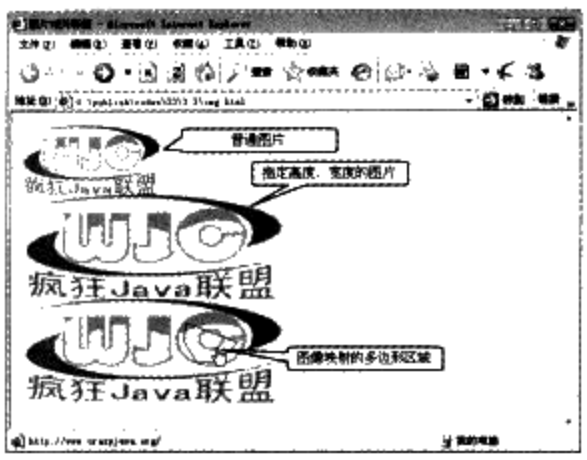


图 3.8　图片和图片映射

3.3.6　表格相关标签

XHTML 为定义表格提供了如下标签：

- <table>：用于定义表格，<table.../>元素只能包含 0 个或 1 个<caption.../>子元素（定义表格标题），0 个或 1 个<thead.../>子元素（定义表格头），0 个或 1 个<tfoot.../>子元素（定义表格脚），多个<tr.../>子元素（定义表格行），多个<tbody.../>子元素（定义表格体）。该元素可以指定 id、style 和 class 等普通属性，也可以指定 onclick 等事件属性，还可以指定如下属性：
  - align：用于指定表格自身的对齐方式，可指定 left、center 或 right 三者之一。
  - bgcolor：用于指定表格的背景色，可指定 rgb(x,x,x)、#xxxxxx 或 colorname 三种类型的颜色值。
  - border：用于指定表格边框的宽度，该值是一个整数，设置 border="0"表示表格无边框。
  - cellpadding：用于指定单元格内容和单元格边框之间的间距。其值既可以是像素值，也可以是百分比。
  - cellspacing：用于指定单元格之间的间距。其值既可以是像素值，也可以是百分比。
  - width：用于指定表格的宽度，其值既可以是像素值，也可以是百分比。
- <caption>：用于定义表格标题，该元素只能包含文本、图片、超级链接、文本格式化元素和表单控件元素等。
- <tr>：用于定义表格的行，该元素只能包含<td.../>或者<th.../>两种元素，可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件属性，还可以指定如下常用属性：
  - align：用于指定该行内所有单元格中文本的水平对齐方式，其值可以是 right、left、center、justify 等。
  - bgcolor：用于指定该行内所有单元格的背景色，其值可以是 rgb(x,x,x)、#xxxxxx 或 colorname 三者之一。
  - valign：用于指定该行内所有单元格中文本的垂直对齐方式，其值可以是 top、middle、bottom 或 baseline 其中之一。
- <td>：用于定义单元格，该元素和<div.../>元素一样，可以包含各种类型的子元素，包括在<td.../>元素里包含<table.../>子元素以再次插入一个表格。该元素可以指定 id、style 和 class 等普通属性，也可以指定 onclick 等事件属性，还可以指定如下属性：
  - align：用于指定该单元格中文本的水平对齐方式，其值可以是 right、left、center、justify 等。
  - valign：用于指定该单元格中文本的垂直对齐方式，其值可以是 top、middle、bottom 或 baseline 其中之一。
  - bgcolor：用于指定单元格的背景色，可指定 rgb(x,x,x)、#xxxxxx 或 colorname 三种类型的颜色值。
  - colspan：用于指定该单元格跨多少列，其值就是一个简单数字。
  - rowspan：用于指定该单元格可横跨的行数。
  - height：用于指定该单元格的高度，其值既可以是像素值，也可以是百分比。
  - width：用于指定该单元格的宽度，其值既可以是像素值，也可以是百分比。
- <th>：用于定义表格页眉的单元格，和<td>标签的用法几乎完全一样，只是浏览器在呈现时有一定差别。

- <tbody>：用于定义表格的主体，该元素只能包含<tr.../>子元素。该元素可以指定 id、style 和 class 等普通属性，也可以指定 onclick 等事件属性，还可以指定如下属性：
  - align：用于指定该表格主体内所有单元格中文本的水平对齐方式，其值可以是 right、left、center、justify 等。
  - valign：用于指定该表格主体内所有单元格中文本的垂直对齐方式，其值可以是 top、middle、bottom 或 baseline 其中之一。

使用 <tbody>标签，可以将一个表格分为几个独立的部分。<tbody.../>元素可将表格中的一行或几行合并成一组。在使用 Ajax 编程时常常需要动态修改表格中的某几行，这时就需要使用<tbody.../>元素了。

在<tbody.../>元素中，必须使用<tr.../>子元素来定义表格行，<tbody.../>元素本身并不会生成任何输出。一旦我们使用<tbody.../>将多行定义为一组，则一个<tbody.../>元素就是表格中的一个独立的部分，即不能从一个<tbody.../>跨越到另一个<tbody.../>中。

- <thead>：用于定义表格的页头，用法与<tbody.../>基本相似，指定功能稍有差别。
- <tfoot>：用于定义表格的页脚，用法与<tbody.../>基本相似，指定功能稍有差别。

<thead.../>、<tfoot.../>、<tbody.../>元素可以让我们对表格中的行进行分组，每个<tbody.../>都是一组，可以多行（在 Ajax 编程中经常用到该元素）。此外，在创建某个表格时，您也许希望拥有一个标题行，由多个数据行组成的组，以及位于底部的一个统计行，以便浏览器能对表格标题和页脚之间的表格内容进行滚动，且在打印长表格内容时表格的表头和页脚被打印在包含表格数据的每个页面上。这些都可借助这三个元素加以实现。

下面的代码使用这些表格标签定义了一个简单的表格。

程序清单：codes\03\3.3\simpleTable.html

```
<table border="1" width="400">
    <caption><b>疯狂 Java 体系图书</b></caption>
    <tr>
        <th>书名</th>
        <th>作者</th>
    </tr>
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>李刚</td>
    </tr>
    <tr>
        <td>轻量级 Java EE 企业应用实战</td>
        <td>李刚</td>
    </tr>
</table>
```

在浏览器中浏览该页面，将可看到如图 3.9 所示效果。

图 3.9　简单表格的效果

下面的代码示范了一个跨行、跨列的表格：

程序清单：codes\03\3.3\tableSpan.html

```
<table width="240" border="1">
    <tr>
```

```
            <td rowspan="2">跨 2 行的单元格</td>
            <td>普通单元格</td>
        </tr>
        <tr>
            <td>普通单元格</td>
        </tr>
        <tr>
            <td colspan="2">跨 2 列的单元格</td>
        </tr>
        <tr>
            <td>普通单元格</td>
            <td>普通单元格</td>
        </tr>
    </table>
```

上面的粗体字代码指定了 rowspan="2"和 colspan="2"两个属性，因此这两个单元格分别可以跨 2 行、跨 2 列。在浏览器中浏览该表格，可看到如图 3.10 所示效果。

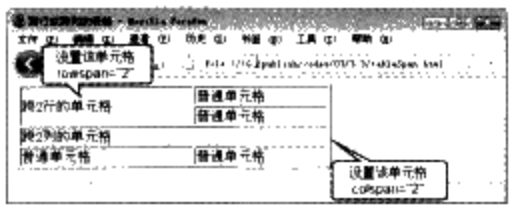


图 3.10　跨行、跨列的表格

下面的表格将使用<thead.../>、<tbody.../>和<tfoot.../>三个元素：

程序清单：codes\03\3.3\tableWithBody.html

```
<table border="1" width="400">
    <caption><b>疯狂 Java 体系图书</b></caption>
    <thead>
    <tr>
        <th>书名</th>
        <th>作者</th>
    </tr>
    </thead>
    <tfoot>
    <tr>
        <td colspan="2" align="right">现总计：2 本图书</td>
    </tr>
    </tfoot>
    <tbody>
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>李刚</td>
    </tr>
    <tr>
        <td>轻量级 Java EE 企业应用实战</td>
        <td>李刚</td>
    </tr>
    </tbody>
</table>
```

上面的代码中我们将<tfoot.../>元素放在<tbody.../>元素之前，但浏览器解释该表格时依然会将<tfoot.../>所包含的表格行放在最后。在浏览器中浏览该页面，将可看到如图 3.11 所示效果。

图 3.11　带 thead、tbody、tfoot 的表格

注意：

如果决定使用<thead.../>和<tfoot.../>元素，建议按如下次序来使用它们：<thead../>、<tfoot.../>、<tbody.../>，浏览器会自动将<tfoot.../>元素的内容呈现在表格最下面。此外，只能在<table.../>元素内使用这些标签。

➤➤3.3.7 框架相关标签

通过使用框架，可以将浏览器分成几个不同的部分，从而允许在同一个浏览器窗口中显示多个XHTML 页面。

如果需要在 XHTML 页面中使用框架，则应该在该 XHTML 页面中使用“XHTML 1.0 Frameset”DTD，而且框架集主页面的<html.../>元素中不允许包含<body.../>子元素（普通的<html.../>元素只能包含<head.../>和<body.../>两个子元素），如果需要在框架集页面中包含<body.../>元素，应将<body.../>子元素放在<noframes.../>元素之中。

框架集页面使用<frame.../>元素来装载其他页面。

XHTML 为框架页面提供了如下标签：

- <frameset>：用于定义一个框架集，以包含其他框架。该元素只能包含<frameset.../>（嵌套框架）、<frame.../>和<noframes.../>三种子元素。该元素可以指定 id、style、class 等核心属性，还可以指定如下两个重要属性：
  - rows：用于指定框架集中各框架的高度，当需要将框架集垂直划分为上下几块时指定该属性。其值应该是如下形式的值：80,60,*，它们分别指定框架集中每个框架的高度。最后一个星号（*）指定该部分高度不受限制。该属性值除了可以是像素值之外，还可以是百分比。
  - cols：用于指定框架集中各框架的宽度，当需要将框架集进行水平划分时指定该属性。该属性值与 rows 属性值的要求完全相同。
- <frame>：用于定义框架集中的一个框架，该元素只能是一个空元素。该元素可以指定 id、style、class 等核心属性，还可以指定如下属性：
  - frameborder：指定框架周围是否显示边框，该属性值只能是 0 或 1。
  - marginheight：指定框架中的顶部和底部的页边距，该属性值只能是像素值。
  - marginwidth：指定框架中的左侧和右侧的页边距，该属性值只能是像素值。
  - name：用于为框架指定一个唯一标识（通常应与 id 属性值相同）。
  - noresize：用于指定是否允许用户调整框架大小。设置该属性值为 noresize 表示不允许调整，默认可以调整大小。
  - scrolling：用于设置该框架的滚动条的行为，可以指定为 yes（有滚动条）、no（无滚动条）或 auto（当内容超出显示范围时显示滚动条）。
  - src：用于指定一个 URL，指定该框架将装载哪个页面。
- <noframes>：用于定义框架集的非框架部分，该元素里的内容不会被显示出来。

如下的页面代码定义了一个复杂的框架集：

程序清单：codes\03\3.3\frameset.html

```
<!-- 指定 cols 属性，该框架将被水平划分 -->
<frameset cols="80,60,*" >
    <!-- 插入第一个框架页面，指定装载 a.html，显示滚动条 -->
    <frame src="a.html" name="firstFrame" scrolling="yes"
        frameborder="1" id="firstFrame" title="firstFrame" />
    <!-- 插入第二个框架页面，指定装载 b.html -->
    <frame src="b.html" name="secondFrame" scrolling="no"
        noresize="noresize" id="secondFrame" title="secondFrame" />
    <!-- 再次插入嵌套的框架集 -->
    <frameset rows="80,*" >
        <!-- 下面再次插入 2 个框架页面，这两个页面可拖动大小 -->
        <frame src="c.html" name="topFrame"
            scrolling="no" id="topFrame" title="topFrame" />
        <frame src="d.html" name="bottomFrame"
            id="bottomFrame" title="bottomFrame" />
```

```
    </frameset>
    <!-- 页面的普通部分应该放在 noframes 里 -->
    <noframes>
        <body>
        主体部分
        </body>
    </noframes>
</frameset>
```

上面的代码中那些粗体字属性指定了这些框架页面是否有滚动条、是否可以拖动大小等，在浏览器中浏览该页面，将可看到如图 3.12 所示效果。

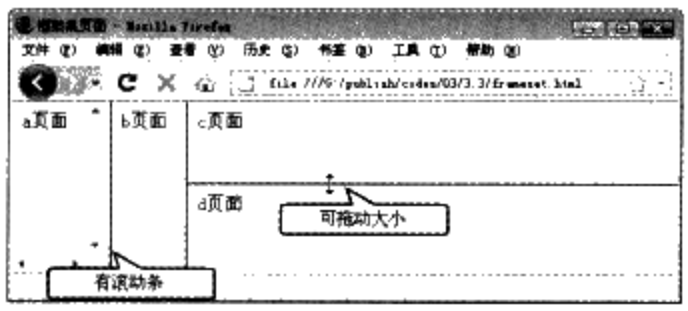


图 3.12　框架集页面

上面的框架集页面中一共定义了四个<frame.../>元素，每个<frame.../>元素都会装载一个页面，因此一共装载了 a.html、b.html、c.html 和 d.html 四个页面，所以我们还应为上面的页面提供如上所述四个页面。

注意：

不要在框架集页面中过多地使用框架，学习过 Java 中 JSplitPane 的读者应该记得，这种分割面板比较耗性能，尤其是打开“连续布局”特性时，性能下降得更加厉害。几乎所有浏览器对框架集页面都是使用“连续布局”行为来处理的。

与框架相关的还有一个<iframe.../>元素，该元素可以在普通 XHTML 页面中使用，用于生成一个内联框架。该元素在用法上与<frame.../>元素非常相似（但不支持指定 noresize 属性），只是该元素无须放在<frameset.../>元素内部，可以直接放在 HTML 页面的任意位置。

下面的 XHTML 页面中包含了一个<iframe.../>元素，该元素定义了一个内联框架。

程序清单：codes\03\3.3\iframe.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 内联框架 </title>
    <meta name="author" content="Yeeku.H.Lee" />
    <meta name="website" content="http://www.leegang.org" />
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
</head>
<body>
<iframe src="include.html" width="200" height="120"></iframe>
主页面内容
</body>
</html>
```

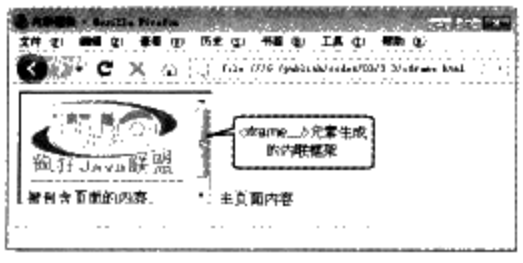


图 3.13　内联框架

上述页面中的粗体字代码定义了一个内联框架，该内联框架负责装载 include.html 页面，在浏览器中浏览该 HTML 页面将可以看到如图 3.13 所示效果。

3.4　XHTML 的表单标签

XHTML 页面中除了可包含前面介绍的基本标签之外，还可以包含一些表单、表单控件标签，表单、表单控件的主要作用就是收集用户输入，当用户提交表单时，用户输入内容将被作为请求参数提交到远程服务器。

➤➤3.4.1 表单标签

<form.../>元素用于创建输入表单，该元素不会生成可视化部分，但其他表单控件，如单行文本框、多行文本域、单选按钮、复选框等都必须放在<form.../>元素之内。

<form.../>元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件属性，还可以指定如下属性：

- action：用于指定在单击表单内的确认按钮时该表单被提交到哪个地址。该属性既可以指定一个绝对地址，也可以指定一个相对地址。该属性必填。
- method：用于指定提交表单时发送何种类型的请求，该属性值可为 get 或 post，分别用于发送 GET 或 POST 请求。通常建议发送 POST 请求。该属性必填。
- enctype：用于指定对表单内容进行编码所使用的字符集。
- name：用于指定表单的唯一名称，建议该属性值与 id 属性值保持一致。
- target：用于指定使用哪种方式打开目标 URL（提交请求会打开另一个 URL 资源），与超级链接的 target 可接受的属性值完全一样，该属性值可以是_blank、_parent、_self 和_top 其中之一。

<form.../>元素的 method 属性非常重要，它指定了该表单提交请求的方式，表单默认以 GET 方式提交请求。GET 请求和 POST 请求的区别如下：

- GET 方式的请求：直接在浏览器地址栏输入访问地址所发送的请求，或提交表单发送请求时该表单对应的<form.../>元素没有设置 method 属性，或设置 method 属性为 get，这几种请求都是 GET 方式的请求。GET 方式的请求会将请求参数的名和值转换成字符串，并附加在原 URL 之后，因此可以在地址栏中看到请求参数名和值。GET 请求传送的数据量较小，一般不能大于 2KB。
- POST 方式的请求：通常使用提交表单的方式来发送，且需要设置<form.../>元素的 method 属性为 post。POST 方式传送的数据量较大，通常认为 POST 请求参数的大小不受限制，但往往取决于服务器的限制，POST 请求传输的数据量总比 GET 传输的数据量大。而且 POST 方式发送的请求参数以及对应的值放在 HTML Header 中传输，用户不能在地址栏里看到请求参数值，安全性相对较高。

表单的 enctype 属性用于指定表单数据的编码方式，该属性有如下 3 个值：

- application/x-www-form-urlencoded：这是默认的编码方式，它只处理表单控件里的 value 属性值，采用这种编码方式的表单会将表单控件的值处理成 URL 编码方式。
- multipart/form-data：这种编码方式会以二进制流的方式来处理表单数据，会把文件域指定文件的内容也封装到请求参数里。需要通过表单上传文件时使用该属性值。
- text/plain：这种编码方式在表单的 action 属性为 mailto:URL 的形式时比较方便，这种方式主要适用于直接通过表单发送邮件。

单纯的<form.../>元素既不能生成可视化内容，也不包含任何表单控件，甚至不能提交表单，因此<form.../>元素必须与其他表单控件元素结合使用。

提 示：

在 XHTML 页面中，提交请求通常有两种方式，提交表单和使用超级链接。提交表单可以让用户输入请求参数，并以 POST 方式提交请求；如果以超级链接方式来提交请求，则只能提交 GET 请求。超级链接提交请求也可包含请求参数，只是不能收集用户输入而已。例如我们定义如下超级链接：发送请求，当用户单击该超级链接时，系统将会向 aa.jsp 页面发送请求，请求参数名为 name，参数值为 leegang.org。

在<form.../>元素里定义一个或多个表单控件，一旦提交该表单，该表单里的表单控件将会转换成请求参数，相应的规则如下：

- 每个有 name 属性的表单控件对应一个请求参数，没有 name 属性的表单控件不会生成请求参数。
- 如果有多个表单控件有相同的 name 属性，则多个表单控件只生成一个请求参数，只是该参数有多个值。
- 表单控件的 name 属性指定请求参数名，value 指定请求参数值。
- 如果某个表单控件设置了 disabled="true"属性，则该表单控件不再生成请求参数。

大部分表单控件，包括<input.../>元素所生成的绝大部分表单控件（除指定了 type="hidden"的隐藏域外）、<button.../>生成的按钮、<select.../>生成的列表框和下拉菜单，以及<textarea.../>生成的多行文本域，因为它们都可以获得鼠标焦点，响应鼠标事件，因此它们都可以指定 onfocus 和 onblur 属性，分别用于设置得到焦点和失去焦点的事件响应，而且这些表单控件都可以指定一个 tabindex 属性。假设 A 控件的 tabIndex 为 1，B 控件的 tabIndex 为 2，C 控件的 tabIndex 为 3，在 A 控件拥有输入焦点的情况下，用户单击 Tab 键将导致输入焦点转移到 B 控件上，再次单击 Tab 键将导致输入焦点转移到 C 控件上……相信读者已经明白了 tabIndex 的作用：通过设置 tabIndex 属性，可让用户无须使用鼠标就可让输入焦点在各表单控件上转移。

➤➤3.4.2 使用 input 元素

<input.../>元素是表单控件元素中功能最丰富的，如下几种输入元素都是通过<input.../>元素来生成的：

- 单行文本框：指定<input.../>元素的 type 属性为 text 即可。
- 密码输入框：指定<input.../>元素的 type 属性为 password 即可。
- 隐藏域：指定<input.../>元素的 type 属性为 hidden 即可。
- 单选框：指定<input.../>元素的 type 属性为 radio 即可。
- 复选框：指定<input.../>元素的 type 属性为 checkbox 即可。
- 图像域：指定<input.../>元素的 type 属性为 image 即可。
- 文件上传域：指定<input.../>元素的 type 属性为 file 即可。
- 提交、重设、无动作按钮：分别指定<input.../>元素的 type 属性为 submit、reset 或 button。

在上面的这些表单控件中，单行文本框、单行密码框都用于接收用户输入，而隐藏域不能接收用户输入，也不能生成可视化部分，它用于提交额外的请求参数，请求参数的值就是该隐藏域的 value 属性值，因此定义隐藏域时应指定 value 属性值。

单选框、复选框也不能接收用户输入，因此定义它们时也会指定 value 属性值，用于设置它们所对应的请求参数值。对于单选框、复选框而言，只有当它们被勾选后，才会生成对应的请求参数。

文件上传域会生成一个单行文本框和一个“浏览”按钮，允许用户浏览本地磁盘文件，并将该文件上传到服务器。

图像域和提交按钮的作用基本一样，单击它们都会导致表单被提交，区别是图像域是一个图像按钮。

重设按钮的作用是清空表单内的用户输入，将表单内所有表单控件的值恢复到初始状态。

无动作按钮，看它的名称就知道，它只是一个按钮。默认情况下，单击该按钮对表单不会有任何作用。通常我们可以为该按钮编写 JavaScript 脚本来响应它的单击、双击等事件。

<input.../>元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件属性，还可以指定 onfocus、onblur 等焦点事件属性。此外，还可以指定如下属性：

- checked：用于设置单选框、复选框的初始状态是否处于选中状态，其值只能是 checked，表示初始即被选中。只有当 type 属性值为 checkbox 或 radio 时才可指定该属性。
- disabled：用于设置首次加载时禁用此元素。其值只能是 disabled，表示该元素被禁用，此时该元素无法获得输入焦点，无法选中，无法在其中输入文本，无法响应鼠标单击、双击等事件。当 type="hidden"时不能指定该属性。
- maxlength：其值是一个数字，用于指定文本框中所允许输入的最大字符数。

- readonly：用于指定该文本框内的值不允许用户修改（可以使用 JavaScript 脚本修改）。
- size：该属性值是一个数字，用于指定该元素的宽度。当 type="hidden"时不能指定该属性。
- src：用于指定图像域所显示图像的 URL，只有当 type="image"时才可以指定该属性。
- align：用于指定图像域之后的文本的对齐方式。其值可以是 left、right、top、texttop、middle、absmiddle、baseline、bottom、absbottom 等，各值的作用读者一试即可明白。只有当 type="image"时才可指定该属性。

下面的表单页使用<input.../>元素定义了一些表单控件。

程序清单：codes\03\3.4\getForm.html

```
<form action="http://www.leegang.org" method="get">
    单行文本框：<input id="username" name="username" type="text" /><br />
    不能编辑的文本框：<input id="username2" name="username" type="text"
        readonly="readonly" /><br />
    密码框：<input id="password" name="password" type="password" /><br />
    隐藏域：<input id="hidden" name="hidden" type="hidden" /><br />
    第一组单选框：<br />
    <input id="color" name="color" type="radio" value="red"/>
    <input id="color2" name="color" type="radio" value="green" />
    <input id="color3" name="color" type="radio" value="blue"/><br />
    第二组单选框：<br />
    <input id="gender" name="gender" type="radio" value="male"/>
    <input id="gender2" name="gender" type="radio" value="female" /><br />
    两个复选框：<br />
    <input id="website" name="website" type="checkbox"
        value="leegang.org" />
    <input id="website2" name="website" type="checkbox"
        value="crazyjava.org" /><br />
    文件上传域：<input id="file" name="file" type="file"/><br />
    图像域：<input type="image" src="img/wjc.gif" /><br />
    下面是四个按钮：<br />
    <input id="ok" name="ok" type="submit" value="提交" />
    <input id="dis" name="dis" type="reset" value="提交"
        disabled="disabled" />
    <input id="cancel" name="cancel" type="submit" value="重填"/>
    <input id="no" name="no" type="button" value="无动作" />
</form>
```

上面的页面定义了大量表单控件，在浏览器中浏览该页面可看到如图 3.14 所示效果。

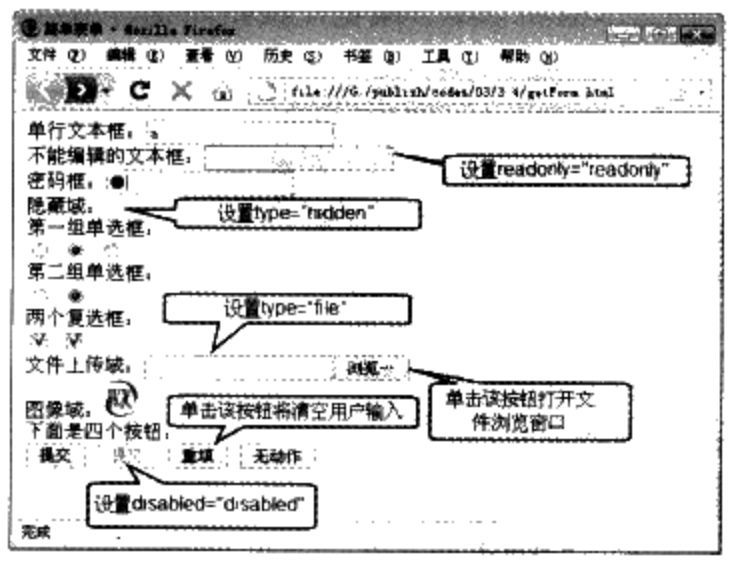


图 3.14　简单表单和表单控件

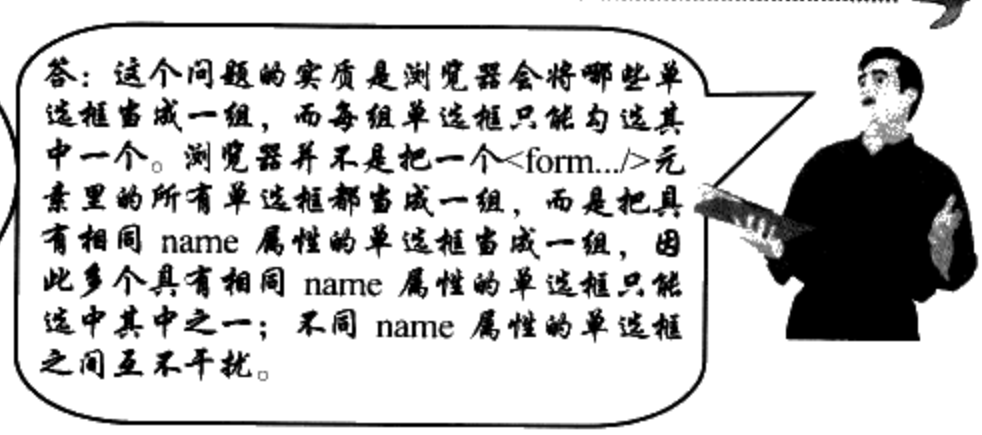
用户单击图像域或 “提交”按钮，该页面将会导航到 http://www.leegang.org 站点，并将用户输入内容作为请求参数发送到该站点，也就是说可以在浏览器地址栏中看到如下 URL：

```
http://www.leegang.org/?username=a&username=&password=b&hidden=&color=green&gender=female&website=leegang.org&website=crazyjava.org&file=&ok=%CC%E1%BD%BB
```

上面的 URL 中的粗体字部分就是发送的请求参数，因为该表单采用 GET 方式发送请求，所以请求参数名、请求参数值将被追加到 URL 之后。从上面的 URL 可以看出，使用 GET 方式发送请求参数时，参数字符串的请求参数名和请求参数值之间以等号（=）隔开，多组请求参数之间以&隔开。

注意：

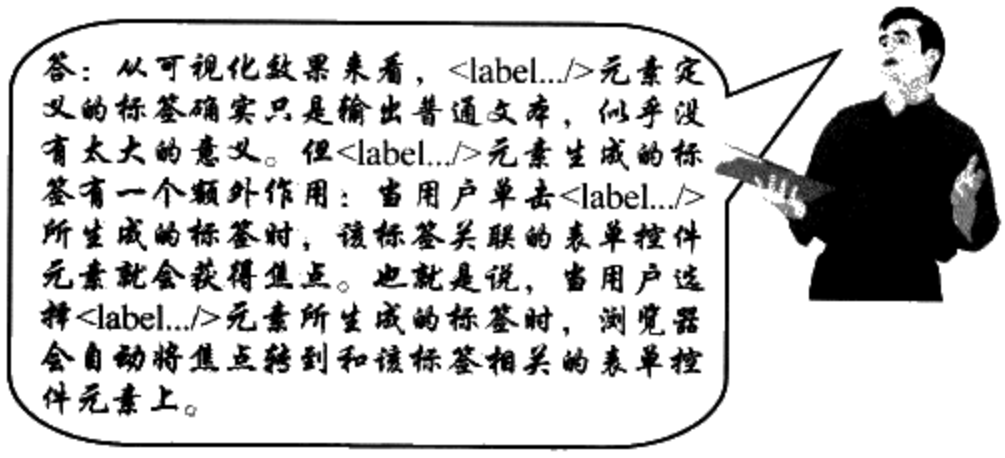
上面的请求 URL 的最后一部分是 ok=%CC%E1%BD%BB，该请求参数由用户单击的“提交”按钮生成——用户通过哪个按钮提交表单，则该按钮也会生成请求参数，前提是该按钮也指定了 name 属性值。该“提交”按钮的 value 属性值为“提交”，使用 application/x-www-form-urlencoded 对“提交”字符串编码后将得到“%CC%E1%BD%BB”字符串，读者可参考《疯狂 Java 讲义》17.2.2 节了解关于 application/x-www-form-urlencoded 编码更详细的内容。



➤➤3.4.3 使用 label 定义标签

<label.../>元素用于在表单元素中定义标签，这些标签可以对其他可生成请求参数的表单控件元素（如单行文本框、密码框等）进行说明，<label.../>元素不需要生成请求参数，因此不要为<label.../>元素指定 value 属性值。

<label.../>元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件属性，此外还可以指定一个 for 属性，用于指定该标签与哪个表单控件关联。



让标签和表单控件关联有两种方式：

- 隐式使用 for 属性：指定<label.../>元素的 for 属性为所关联表单控件的 id 属性值。
- 显式关联：将普通文本、表单控件一起放在<label.../>元素内部即可。

下面的表单代码分别使用了两种方式将标签和表单控件关联在一起。

程序清单：codes\03\3.4\label.html

```
<form action="http://www.leegang.org" method="get">
    <label for="username">单行文本框：</label>
    <input id="username" name="username" type="text" /><br />
```

```
    <label>密码框：<input id="password" name="password" type="password" />
    </label><br />
    <input id='ok' type="submit" value="登录疯狂 Java 联盟" />
</form>
```

上面的页面中粗体字代码用于添加<label.../>元素，而且该<label.../>元素可以和指定的表单控件关联。当用户单击表单控件前面的标签时，该表单控件就可以获得输入焦点，如图 3.15 所示。

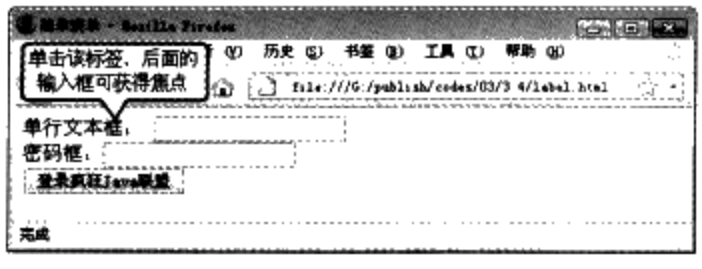


图 3.15　label 生成标签

注意：

尽量少用显式关联的方式，这种方式在笔者的 Internet Explorer 中没有很好的支持，当用户单击<label.../>元素对应的标签时，所关联的表单控件并不会获得输入焦点。

3.4.4　使用 button 定义按钮

<button.../>元素用于定义一个按钮，在其内部可以包含普通文本、文本格式化标签、图像等内容，这也正是<button.../>按钮和<input.../>按钮的不同之处。

<button.../>按钮与<input type="button" />相比，提供了更为强大的功能和更丰富的内容。<button>与</button>标记之间的所有内容都是该按钮的内容，其中包括任何可接受的正文内容，比如文本或图像。

值得指出的是，不要在<button>与</button>标记之间放置图像映射，因为它对鼠标和键盘敏感的动作会干扰表单按钮的行为。

<button.../>元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件响应属性，还可以指定如下属性：

- disabled：用于指定是否禁用此按钮。其值只能是 disabled。
- name：用于指定该按钮的唯一名称。其值应该与 id 属性值保持一致。
- type：用于指定该按钮属于哪种按钮，其值只能是 button、reset 或 submit 其中之一。
- value：用于指定该按钮的初始值。该值可通过脚本进行修改。

如下页面代码使用了<button.../>元素来定义按钮。

程序清单：codes\03\3.4\button.html

```
<form action="http://www.leegang.org" method="get">
    <button type="button"><b>提交</b></button><br />
    <button type="submit"><img src="img/wjc.gif" alt="crazyjava.org"/></button><br />
    <button type="reset">
        <table border="1" width="120">
        <tr><td>crazyjava.org</td></tr>
        <tr><td>leegang.org</td></tr>
        </table>
    </button><br />
</form>
```

上面的粗体字代码定义了三个按钮，其内容分别为文字、图像、表格，在浏览器中浏览该页面可看到如图 3.16 所示效果。

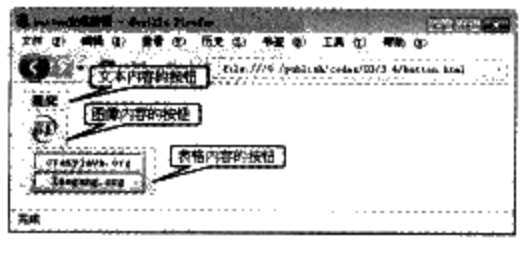


图 3.16　button 生成的按钮

➤➤3.4.5 列表框和下拉菜单

<select.../>元素用于创建列表框或下拉菜单，该元素必须和<option.../>元素结合使用，每个<option.../>元素代表一个列表项或菜单项。

与其他表单控件不同的是，<select.../>元素本身并不能指定 value 属性，列表框或下拉菜单控件对应的参数值由<option.../>元素来生成，当用户选中了多个列表项或菜单项后，这些列表项或菜单项的 value 值将作为该<select.../>元素所对应的请求参数值。

<select.../>元素可以指定 id、style、class 等核心属性，但仅可指定 onchange 事件属性——当该列表框或下拉列表内的选中项发生改变时，将触发 onchange 事件。此外还可指定如下属性：

- disabled：用于设置禁用该列表框和下拉菜单，其值只能是 disabled。
- multiple：用于设置该列表框和下拉菜单是否允许多选，其值只能是 multiple，即表示允许多选。一旦设置允许多选，<select.../>元素将自动生成列表框。
- size：用于指定该列表框内可同时显示多少个列表项。一旦指定该属性，则<select.../>元素自动生成列表框。

提 示：

一个<select.../>元素到底是生成列表框还是生成下拉菜单，完全由是否指定了 size 或 multiple 属性来决定，只要为<select .../>元素指定了这两个属性其中之一，浏览器就会生成列表框，否则就是下拉菜单。

在<select.../>元素里，只能包含<option.../>和<optgroup.../>两种子元素，其中<option.../>定义列表框选项或菜单项，而<optgroup.../>用于定义列表项或菜单项组。<option.../>元素里只能包含文本内容，作为该选项的文本，<optgroup.../>里只能包含<option.../>子元素，处于<optgroup.../>里的<option.../>就属于该组。

<option.../>元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件响应属性，还可以指定如下属性：

- disabled：用于指定禁用该选项，其值只能是 disabled。
- selected：用于指定该列表项一开始是否处于被选中状态。其值只能是 selected。
- value：用于指定该选项对应的请求参数值。

<optgroup.../>元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件响应属性，还可以指定如下属性：

- label：用于指定该选项组的标签。这个属性必填。
- disabled：用于设置禁用该选项组里的所有选项。其值只能是 disabled。

下面的代码使用<select.../>元素定义了 1 个下拉菜单和 2 个列表框。

程序清单：codes\03\3.4\list.html

```
<form action="http://www.leegang.org" method="post">
    下面是简单下拉菜单：<br />
    <select id="skills" name="skills">
        <option value="java">Java 语言</option>
        <option value="c">C 语言</option>
        <option value="ruby">Ruby 语言</option>
    </select><br /><br /><br />
    下面是允许多选的列表框：<br />
    <select id="books" name="books"
        multiple="multiple" size="4">
        <option value="java">Struts2 权威指南</option>
        <option value="c">疯狂 Java 讲义</option>
        <option value="ruby">Core Java</option>
```

```
    </select><br />
    下面是允许多选的列表框：<br />
    <select id="leegang" name="leegang"
        multiple="multiple" size="6">
        <optgroup label="疯狂 Java 体系图书">
            <option value="java" label="aaaaaaaa">疯狂 Java 讲义</option>
            <option value="ajax">疯狂 Ajax 讲义</option>
            <option value="xml">疯狂 XML 讲义</option>
    </optgroup>
    <optgroup label="李刚其他图书">
            <option value="java">Struts2 权威指南</option>
            <option value="ajax">RoR 敏捷开发最佳实践</option>
        </optgroup>
    </select><br />
    <button type="submit"><b>提交</b></button><br />
</form>
```

上面的粗体字代码中，multiple="multiple" size="4"指定该列表框高度为 4，并允许多选；<optgroup.../>元素则定义了列表框中的选项组。在浏览器中浏览该页面，可看到如图 3.17 所示效果。

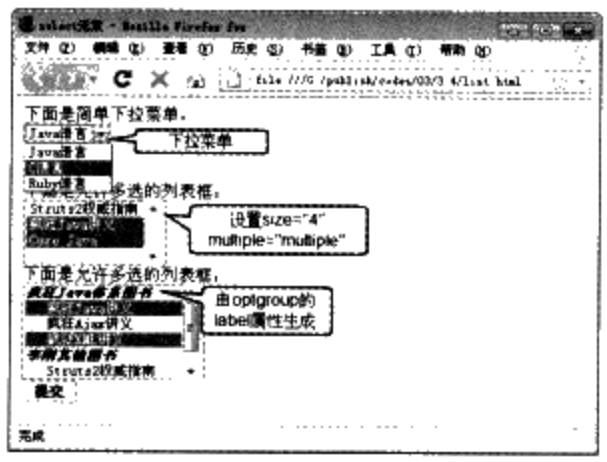


图 3.17　下拉菜单和列表框

从上面的页面代码可以看出，每个列表项也就是一个<option.../>，通常应该包含 2 个文本，一个是该<option.../>元素的 value 属性，另一个是<option>和</option>标记之间的内容，也就是每个选项的文本内容。

➤➤3.4.6　使用 textarea 定义文本域

<textarea.../>元素可以在 XHTML 页面上定义多行文本域，该元素可以指定 id、style、class 等核心属性，也可以指定 onclick 等事件属性。由于 texterea 的特殊性，它可以接收用户输入，用户可以选中文本域内的文本，所以还可以指定 onselect 和 onchange 两个属性，分别用于响应文本域内文本被选中和文本被修改的事件。此外还可以指定如下属性：

- cols：用于指定文本域的宽度，该属性必填。
- rows：用于指定文本域的高度，该属性必填。
- disabled：用于指定禁用该文本域。其值只能是 disabled，表示首次加载时禁用该文本域。
- readonly：用于指定该文本域只读。其值只能是 readonly。

与单行文本框相同的是，<textarea.../>元素也应指定 name 属性，该属性将作为 textarea 所对应的请求参数的参数名；与单行文本框不同的是，<textarea.../>元素不能指定 value 属性，<textarea>和</textarea>标记之间的内容将作为<textarea.../>所对应的请求参数的参数值。

下面的页面代码定义了两个多行文本域：

程序清单：codes\03\3.4\textarea.html

```
<form action="http://www.leegang.org" method="post">
    简单多行文本域：<br />
```

```
    <textarea cols="20" rows="2"></textarea><br />
    只读的多行文本域：<br />
    <textarea cols="28" rows="4" readonly="readonly">
疯狂 Java 讲义
轻量级 Java EE 企业应用实战
    </textarea><br />
    <button type="submit"><b>提交</b></button><br />
</form>
```

在浏览器中浏览该页面，将可以看到如图 3.18 所示效果。

图 3.18 多行文本域

3.5 XHTML 头部和元信息

到目前为止，我们已经掌握了 XHTML 文档中的绝大部分主要标签，现在还剩下一些非常简单的头部和元信息标签。

定义 XHTML 头部使用<head.../>元素，该元素可以包含<script.../>（可用于包含 JavaScript 脚本）、<style.../>（用于定义内部 CSS 样式）、<link.../>（用于链接外部 CSS 等资源）、<meta.../>、<title.../>和<base.../>等子元素。前面三个标签在后面的章节有更详细的介绍，故此处仅介绍后面三个标签。

<title.../>元素用于定义文档标题，通常该元素较为常用的属性是 id，作为其唯一标识。该元素只能包含文本内容，该文本内容就是该文档的标题。

<base.../>元素用于指定该页面中的所有链接的基准链接。该元素必须是空元素，除了可以指定 id 作为其唯一标识外，还可以指定如下属性：

- href：用于指定所有链接的基准链接。
- target：用于指定超级链接默认在哪个窗口打开。其值只能是_blank、_parent、_self 和_top 其中之一。

例如如下页面：

程序清单：codes\03\3.5\base.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> base 标签 </title>
    <base target="_blank" href="http://www.leegang.org" />
</head>
<body>
    <a href="index.php">疯狂 Java 联盟</a>
</body>
</html>
```

上面的页面代码中使用<base.../>指定了所有链接的基准路径为 http://www.leegang.org，默认使用新窗口打开连接，页面中的超级链接的地址为 index.php，则实际 URL 为 http://www.leegang.org/index.php，并使用新窗口打开。

<meta.../>用于定义页面元信息，定义元信息也就是指定一些 name-value 对。该元素除可以指定 id 属性外，还可以指定如下属性：

- http-equiv：用于指定元信息的名称，该属性指定的名称具有特殊意义，可向浏览器传回一些有用的信息，帮助浏览器正确地处理网页内容。
- name：用于指定元信息的名称，可随意指定。
- content：用于指定元信息的值。

根据上面的讲解我们知道：<meta.../>元素里的 http-equiv 属性和 name 属性的作用基本相同，只是 http-equiv 属性值通常有规定，应该是浏览器可以识别的、具有特殊意义的名称。

例如我们可以为网页指定如下关键字和描述信息：

```
<head>
    <title> 疯狂 Java 联盟 </title>
    <meta name="author" content="Yeeku.H.Lee" />
    <meta name="website" content="http://www.leegang.org" />
    <meta name="copyright" content="2001-2008 leegang.org" />
    <meta name="Keywords" content="Java 论坛,Java 技术论坛" />
</head>
```

为网页指定有效的关键字有利于搜索引擎收录本站点。

http-equiv 属性所支持的值主要有：

- Expires：用于指定网页的过期时间。一旦网页过期，则必须重新从服务器上下载。例如如下代码：

```
<meta http-equiv="Expires" content="Sat Sep 27 16:12:36 CST 2008" />
```

- Pragma：用于指定禁止浏览器从本地磁盘缓存中调阅页面内容，浏览器一旦离开该网页就无法脱机访问该页面。例如：

```
<meta http-equiv="Pragma" content="no-cache" />
```

- Refresh：用于指定浏览器多长时间后自动刷新指定页面。例如：

```
<!-- 设置 2 秒后自动刷新本页面 -->
<meta http-equiv="Refresh" content="2" />
<!-- 设置 2 秒后自动刷新 http://www.leegang.org -->
<meta http-equiv="Refresh" content="2;URL=http://www.leegang.org" />
```

- Set-Cookie：设置 Cookie。如果网页过期，那么客户端上的 Cookie 也将被删除。例如：

```
<meta http-equiv="Set-Cookie"
    content="name=value;expires=Sat Sep 27 16:12:36 CST 2008;path=/" />
```

- Content-Type：设置该页面的内容类型和所用的字符集。例如：

```
<meta http-equiv="Content-Type" content="text/html; charset=GBK" />
```

此外还可以设置一些不太常用的属性值，此处不再详述。

3.6 本章小结

本章详细介绍了 XHTML 语言的相关规范，介绍了 HTML 语言的发展历史，从而说明了 XHTML 是 HTML 的升级版本。因为 XHTML 语言并不是真正的编程语言，只是一种标记语言，所以本章内容比较简单，全部都是对各 XHTML 标签的语法规则的介绍。本章首先介绍了 XHTML 文档的结构和语法规则，接着详细介绍了各种基本的 XHTML 标签的用法、语法规则，以及各属性的作用和意义。本章重点介绍了各种常用的 XHTML 标签，包括列表相关标签、图像相关标签、表格相关标签、框架相关标签、表单相关标签、表单控件相关标签等。此外本章还简要介绍了 XHTML 头部和元信息相关标签。

➤➤本章练习

登录 http://www.crazyjava.org/ethos.php 页面，将你看到的页面效果做出来（可能需要结合第 5 章的知识）。

登录 http://www.crazyjava.org/index.php 页面，将你看到的页面效果做出来（可能需要结合第 5 章的知识）。

CHAPTER

4

第4章
JavaScript 语法详解

本章要点

- JavaScript 简介
- 嵌入、运行 JavaScript 代码
- JavaScript 的数据类型和变量声明
- JavaScript 的正则表达式支持
- JavaScript 的运算符
- JavaScript 的语句形式
- JavaScript 的流程控制
- JavaScript 的函数
- 函数的参数处理机制
- 函数、类、构造器的关系
- 使用函数创建对象
- 对象和关联数组
- JavaScript 的类和对象
- 通过 prototype 动态地扩展一个类
- 创建对象的三种方式

JavaScript 语言并不是 Java 语言，它是一种脚本语言。JavaScript 由 LiveScript 改名而来，这个名字也是 Netscape 公司在最后一刻才决定的，名为 JavaScript，其实与 Java 并无太大关联，可能仅出于一种市场考虑。JavaScript 是一种基于客户端浏览器的，面向对象、事件驱动式的脚本语言。JavaScript 也具有跨平台的特点。如同所有的脚本语言一样，它的脚本是动态解释执行的。

在 JavaScript 出现之前，互联网页都是静态内容，就像一张一张写满内容的纸，Netscape 公司为了丰富互联网功能，所以在 Navigator 浏览器中扩展了 JavaScript 支持，这样就大大扩展了互联网页的功能，使得互联网页可以拥有丰富多彩的动画和用户交互。直到现在，运行 JavaScript 的主要环境依然是各种浏览器，因此通常会将 JavaScript 嵌入互联网页中，由浏览器负责解释执行。JavaScript 的主要功能为：动态修改 HTML 页面内容，包括创建、删除 HTML 页面元素，修改 HTML 页面元素的内容、外观、位置、大小等。

因为 JavaScript 由网站服务器开发，供用户下载到客户端执行，因此 JavaScript 通常有如下两个限制：

- JavaScript 不能访问客户机的本地磁盘系统。
- JavaScript 不能打开客户机上的网络连接。

4.1 JavaScript 简介

正如前面提到的，JavaScript 并不是 Java。JavaScript 与 Java 的区别体现在：

- Java 和 JavaScript 是两个完全不同的产品。Java 是 SUN 公司推出的面向对象的程序设计语言；而 JavaScript 是 Netscape 公司的产品，其目的是为了扩展 Netscape 浏览器的功能。JavaScript 是一种可以嵌入 Web 页面中的解释性语言。
- Java 是面向对象的程序设计语言，即使是开发简单的程序，也必须从类定义开始。JavaScript 是基于对象的，本身提供了非常丰富的内部对象供设计人员使用。Java 语言的最小程序单位是类定义，而 JavaScript 中则充斥着大量函数。
- 两种语言的执行方式完全不同。Java 语言必须先经过编译生成字节码，然后由 Java 虚拟机运行这些字节码。JavaScript 是一种脚本语言，其源代码无须经过编译，由浏览器解释执行。
- 两种语言的变量声明也不一样。Java 采用强类型变量语言，所有变量必须先经过声明才可以使用，所有的变量都有其固定的数据类型。JavaScript 是弱类型变量语言，其变量在使用前无须声明，而由解释器在运行时检查其数据类型。
- 代码格式不一样。Java 的代码是一种与 HTML 无关的格式，必须通过像 HTML 中引用外媒体那样进行装载，其代码以字节码的形式保存在独立的文档中。JavaScript 的代码是一种文本字符格式，可以直接嵌入 HTML 文档中，并且可动态装载。编写 HTML 文档就像编辑文本文件一样方便。

在实际的使用中，还有另一种脚本语言：JScript 语言。JScript 与 JavaScript 的渊源比较深。事实上，两种语言的核心功能、作用基本一致，都是为了扩充浏览器的功能而开发的脚本语言，不同之处只是 JavaScript 由 Netscape 公司开发，而 JScript 语言由 Microsoft 公司开发。

早期的 JScript 和 JavaScript 差异相当大，Web 程序员不得不痛苦地为两种浏览器分别编写脚本。于是诞生了 ECMAScript，这是一种国际标准化的 JavaScript 版本，现在的主流浏览器都支持这种版本。而 Microsoft 制订 JScript 语言时，也不得不参考 ECMAScript 标准。

程序中需要使用 JavaScript 脚本时，最好写 <script language="javascript"> 而不是 <script language="jscript">。因为 JavaScript 是一个通用的名称，所有浏览器都可以识别，而 JScript 只有 Microsoft 的 Internet Explorer 能识别。

虽然 JavaScript 和 JScript 是不同公司的产品，存在不小的差异。但因为二者完成的功能如此接近，因此二者之间存在很大的相似性，两种语言的不同版本大致差不多。表 4.1 是 JavaScript 和 JScript 两种脚本语言版本的大致对照。

表 4.1 JavaScript 和 JScript 版本对照

JavaScript 版本	完全支持的浏览器	JScript 版本	完全支持的浏览器
JavaScript 1.0	Netscape 2.0	JScript 1.0	Internet Explorer 3.0
JavaScript 1.3	Netscape 4.06	JScript 3.0	Internet Explorer 4.0
JavaScript 1.5	Netscape 6.0 以上， 其他 Mozilla 内核的浏览器	JScript 5.5	Internet Explorer 5.5 以上版本

虽然 JavaScript、JScript 和 ECMAScript 三种脚本语言基本差不多，但具体的细节存在不少差异，编程时应尽量避免使用浏览器特定的功能，最好遵循 ECMAScript 标准。这样可以保证兼容性。

➤➤4.1.1 运行 JavaScript

前面已经介绍了 JavaScript 必须嵌在互联网页中执行，在 XHTML 页面中嵌入执行 JavaScript 代码有两种方式：

- 使用 javascript:前缀构建执行 JavaScript 代码的 URL。
- 使用<script.../>元素来包含 JavaScript 代码。

对于第一种情况，所有可以设置 URL 的地方都可以使用这种以 javascript:作为前缀的 URL，在用户激发该 URL 时，javascript:之后的 JavaScript 代码就会获得执行。

如果页面里需要包含大量 JavaScript 代码，则建议将这些 JavaScript 脚本放在标签<script>和</script>之间。<script.../>元素既可作为<head.../>子元素，也可作为<body.../>子元素。

例如如下页面代码片段。

程序清单：codes\04\4.1\run.html

```
<body>
    <a href="javascript:alert('运行 JavaScript! ');">运行 JavaScript</a>
        <script type="text/javascript">
            alert("直接运行的 JavaScript! ");
    </script>
</body>
```

上面的页面中粗体字代码示范了两种运行 JavaScript 代码的方式，第一种方式会生成一个超级链接，当用户单击该超级链接时，alert('运行 JavaScript！');就会获得执行。

➤➤4.1.2 导入 JavaScript 文件

为了让 XHTML 页面和 JavaScript 脚本更好地分离，我们可以将 JavaScript 脚本单独保存在一个*.js 文件中，XHTML 页面导入该*.js 文件即可。在 XHTML 页面中导入 JavaScript 脚本文件的语法格式如下：

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

上面的语法中 src 属性指定 JavaScript 脚本文件所在的 URL 即可。

例如我们有如下*.js 文件：

程序清单：codes\04\4.1\test.js

```
//弹出一个对话框
alert("测试");
```

只要在当前路径（codes\04\4.1\）的 XHTML 页面中增加如上所示一行，则与直接在这些 XHTML 页面中增加该 JavaScript 脚本的效果完全一样。

4.2 数据类型和变量

任何语言都离不开数据类型和变量。虽然 JavaScript 语言是弱类型的语言，但是一样支持变量的声明，变量一样存在作用范围，即有局部变量和全局变量之分。下面依次介绍 JavaScript 中数据类型

和变量的基本语法。

➤➤4.2.1 定义变量的方式

JavaScript 是弱类型的脚本语言。使用变量之前，可以无须定义，想使用某个变量时直接使用即可。归纳起来，JavaScript 支持以下两种方式来引入变量：

- 隐式定义：直接给变量赋值的方式。
- 显式定义：使用 var 关键字定义变量。

隐式定义的方式简单快捷。需要使用变量时，直接给需要使用的变量赋值即可。看下面的代码：

程序清单：codes\04\4.2\implicit_var.html

```
<script>
    //隐式定义变量 a
    a = "Hello JavaScript";
    //使用警告对话框输出 a 的值
    alert(a);
</script>
```

程序执行结果如图 4.1 所示。

图 4.1 隐式变量声明的效果

显式的声明方式是采用 var 关键字声明变量，声明时变量可以没有初始值，声明的变量数据类型是不确定的。在第一次给变量赋值时，变量的数据类型才确定下来，而且使用过程中变量的数据类型也可随意改变。看下面显式声明变量的示例代码：

程序清单：codes\04\4.2\explicit_var.html

```
<script>
    //显式声明变量 a
    var a ;
    //给变量 a 赋值，赋值后 a 的数据类型为布尔型
    a = true;
    //使用警告对话框输出 a 的值
    alert(a);
</script>
```

注意：

JavaScript 中的变量是区分大小写的。因此变量 abc 和 Abc 是两个不同的变量，读者编程时一定要注意。

与其他编程语言类似的是，JavaScript 也允许一次定义多个变量，例如如下代码：

```
//一次定义了 a、b、c 三个变量
var a , b , c;
```

还可以在定义变量时为变量指定初始值，例如如下代码：

```
//定义变量 i、j、k，其中 j、k 指定初始值
var i, j = 0, k = 0;
```

➤➤4.2.2 类型转换

JavaScript 支持自动类型转换，这种类型转换的功能非常强大，看如下代码：

程序清单：codes\04\4.2\autoConversion.html

```
<script>
    //定义字符串变量
    var a = "3.145";
```

```
    //让字符串变量和数值执行算术运算
    var b = a - 2;
    //让字符串变量和数值执行运算，到底是算术运算，还是字符串运算呢？
    var c = a + 2;
    //输出b和c的值
    alert (b + "\n" + c);
</script>
```

代码执行结果如图 4.2 所示。

图 4.2　自动类型转换结果

在上面的代码中，a 是值为 3.145 的字符串。让 a 和数值执行减法，自动执行算术运算，并将 a 的类型转换成数值。让 a 和数值执行加法，则 a 的类型转换成字符串。这就是自动类型转换。这种转换的规律是：

- 对于减号运算符，因为字符串不支持减法运算，所以系统自动将字符串转换成数值。
- 对于加号运算符，因为字符串可用加号作为连接运算符，所以系统自动将数值转化成字符串，并对两个字符串进行连接运算。

各种类型自动类型转换的结果如表 4.2 所示。

表 4.2　自动类型转换值

值	目标类型			
	字符串类型	数值型	布尔型	对象
undefined	"undefined"	NaN	false	Error
null	"null"	0	false	Error
字符串	不变	数值或 NaN	true	String 对象
空字符串	不变	0	false	String 对象
0	"0"	0	false	Number 对象
NaN	"NaN"	NaN	false	Number 对象
Infinity	"Infinity"	Infinity	true	Number 对象
-Infinity	"-Infinity"	-Infinity	true	Number 对象
数值	数值字符串	不变	true	Number 对象
true	"true"	1	不变	Boolean 对象
false	"false"	0	不变	Boolean 对象
对象	toString()返回值	valueOf()、toString()或 NaN	true	不变

这种自动类型转换虽然方便，但程序可读性非常差，而且有时候我们就是希望让字符串和数值执行加法运算，这时就需要使用强制类型转换了。JavaScript 还提供了如下几个函数来执行强制类型转换：

- toString()：将布尔值、数值等转换成字符串。
- parseInt()：将字符串、布尔值等转换成整数。
- parseFloat()：将字符串、布尔值等转换成浮点数。

如果需要让"3.145" + 2 这种表达式的结果为 5.145，可以使用强制类型转换：

```
<script>
    //定义值为 3.145 的字符串变量
    var a = "3.145";
    //直接相加，使用自动类型转换
    var b = a + 2;
    //使用强制类型转换
    var c = parseFloat(a) + 2;
```

```
    alert (b + "\n" + c);
</script>
```

代码执行结果如图 4.3 所示。

图 4.3 自动类型转换与强制类型转换的对比

对于 3.145 这种可以正常转换成数值的字符串，可以成功转换为数值；但对于包含其他字符的字符串，将转换成 NaN。

使用 parseInt()或 parseFloat()将各种类型变量转换成数值类型的结果如下：

- 字符串值：如果字符串是一个数值字符串，则可以转换成一个数值，否则将转换成 NaN。
- undefined、null、布尔值及其他对象：一律转换成 NaN。

使用 toString()函数将各种类型值向字符串转换的结果全部是 object。

➤➤4.2.3 变量

变量是程序设计语言里最重要也最基本的概念。与其他强类型语言不同的是，JavaScript 是弱类型的语言，同一个变量可以一会儿是数值，一会儿是字符串。正如前面讲到的，变量的声明可以有两种方式：显式声明和隐式声明。变量还有个重要的概念：作用范围。

根据变量定义的范围不同，变量有全局变量和局部变量两种。直接定义的变量是全局变量，全局变量可以被所有的脚本访问；在函数里定义的变量称为局部变量，局部变量只在函数内有效。如果全局变量和局部变量使用相同的变量名，局部变量将覆盖全局变量。看如下代码：

程序清单：codes\04\4.2\scope_test.html

```
<script>
    //定义全局变量 test
    var test = "全局变量";
    //定义函数 checkscope
    function checkscope()
    {
        //定义局部变量
        var test = "局部变量";
        //输出局部变量
        alert(test);
    }
    checkscope();
</script>
```

代码的执行结果是“局部变量”，代码中定义了全局变量 test，但在函数中又定义了局部变量 test，函数中的局部变量覆盖了全局变量。与 Java、C 等语言不同的是，JavaScript 语言没有块范围，看如下代码：

程序清单：codes\04\4.2\noBlockScope.html

```
<script>
    function test(o)
    {
        //定义变量 i，变量 i 的作用范围是整个函数
        var i = 0;
        if (typeof o == "object")
        {
            //定义变量 j，变量 j 的作用范围是整个函数内，而不是 if 块内
            var j = 5;
            for(var k = 0; k < 10; k++)
            {
                //因为 JavaScript 没有代码块范围
                //所以 k 的作用范围是整个函数内，而不是循环体内
                document.write(k);
```

```
            }
        }
        //即使出了循环体，k的值依然存在
        alert(k + "\n" + j);
    }
    test(document);
</script>
```

程序执行结果如图 4.4 所示。

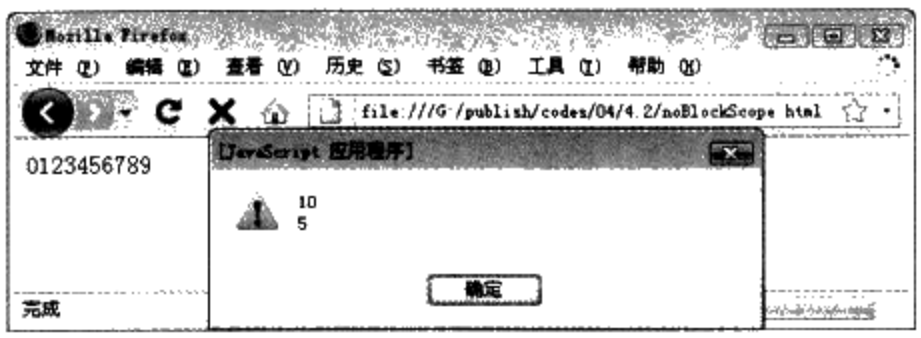


图 4.4　JavaScript 的变量没有块范围

JavaScript 的变量没有块范围，因此有时可能出现一些非常奇怪的结果。看如下代码：

程序清单：codes\04\4.2\noBlockScope2.html

```
<script>
    //定义全局变量
    var scope = "全局变量";
    function test()
    {
        //因为全局变量被局部变量覆盖
        //而此时 scope 局部变量尚未赋值，故此处输出 undefined
        document.writeln(scope + "<br />");
        //定义局部变量 scope，其作用范围为整个函数内
        var scope = "局部变量";
        //再次输出 scope 的值
        document.writeln(scope + "<br />");
    }
    test();
</script>
```

运行上面的代码可看到如图 4.5 所示效果。

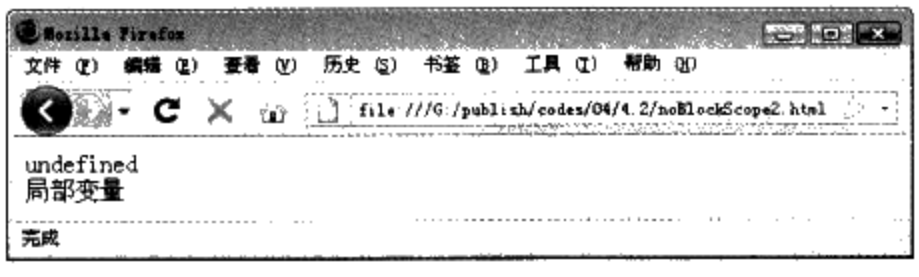


图 4.5　局部变量覆盖全局变量

代码第一次输出的 scope 值并不是“全局变量”，而是 undefined。这是因为全局变量 scope 在 test 函数中已经被覆盖了，局部变量 scope 在 test 函数中将全局有效，但此处 scope 还未被赋值，故此处输出 undefined。

变量作用范围对于执行 HTML 事件处理时一样有效，看如下代码：

程序清单：codes\04\4.2\globalInHandler.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
    <title> 事件处理中的局部变量和全局变量 </title>
    <script type="text/javascript">
        //定义全局变量
        var x = "全局变量";
    </script>
</head>
<body>
<!-- 在 onclick 事件中重新定义了局部变量 x-->
<input type="button" value="局部变量"
    onclick="var x = '局部变量'; alert('输出局部变量 x 的值：' + x);" />
<!-- 直接输出全局变量 x 的值 -->
<input type="button" value="全局变量"
    onclick="alert('输出全局变量 x 的值：' + x);" />
</body>
</html>
```

对于第一个按钮的事件处理脚本而言，因为该脚本中重新定义了局部变量 x，所以访问 x 将输出该局部变量的值；对于第二个按钮的事件处理脚本而言，由于该脚本中没有定义变量 x，所以访问变量 x 时将输出全局变量的值。单击第一个按钮将弹出局部变量的值，单击第二个按钮将弹出全局变量的值。

4.3 基本数据类型

JavaScript 是弱类型的脚本语言，声明变量时无须指定变量的数据类型。JavaScript 变量的数据类型是解释时动态决定的。但 JavaScript 的变量保存在内存中时也有数据类型。JavaScript 的基本数据类型有如下 5 种：

- 数值类型：包含整数和浮点数。
- 布尔类型：只有 true 和 false 两种值。
- 字符串类型：字符串变量必须以引号括起来，引号可以是单引号，也可以是双引号。
- undefined 类型：专门用来确定一个已经创建但是没有初值的变量。
- null：用于表明某个变量的值为空。

➤➤4.3.1 数值类型

与强类型语言如 C、Java 不同，JavaScript 的数值类型不仅包括所有的整型变量，也包括所有的浮点型变量。JavaScript 语言中的数值都以 IEEE 754-1985 双精度浮点数格式保存。JavaScript 中的数值形式可以非常丰富，完全支持科学计数法表示。科学记数法的语法格式是：

数字 1E 数字 2

这种形式的值为：数字 1×10 数字2。E 为间隔符号，不区分大小写。看如下代码：

程序清单：codes\04\4.3\simpleNumber.html

```
<script>
    //显式声明变量 a ,b
    var a , b;
    //给 a, b 使用科学记数法赋值，其值应该为 500
    a = 5E2;
    b = 1.23e-3;
    //使用警告提示框输出变量 a 的值
    alert(a + "\n" + b);
</script>
```

代码执行结果如图 4.6 所示。

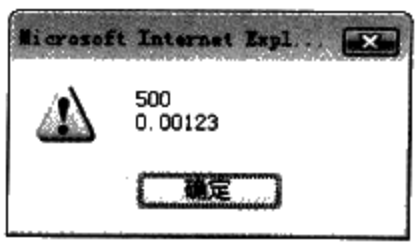


图 4.6　科学记数法表示的值

如果数值只有小数部分，则可以省略整数部分的 0，但小数点不能省略。看下面的代码：

程序清单：codes\04\4.3\simpleNumber2.html

```
<script>
    //使用隐式声明定义变量 b
    b = 3.12e1;
    //使用隐式声明定义变量 c
    c = 45.0;
    //使用隐式声明定义变量 d
    d = .34e4;
    //使用隐式声明定义变量 e
    e = .24e-2;
    //使用警告框输出四个变量值
    alert(b + '---' + c + '---' + d + '---' + e);
</script>
```

可以看到，JavaScript 支持的数值格式相当丰富，上述代码的输出结果如图 4.7 所示。

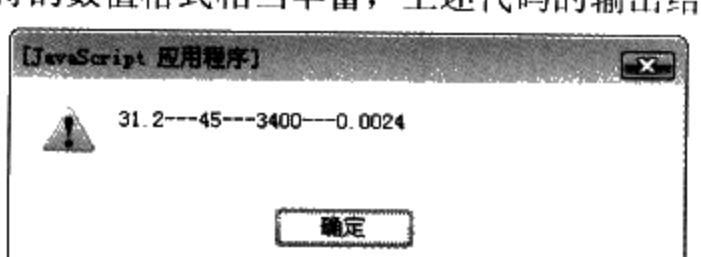


图 4.7　科学记数法的数值

注意：

数值直接量不要以 0 开始。因为 JavaScript 不仅支持十进制数，还支持其他进制的数。八进制和十六进制数都以 0 开始。

JavaScript 除了支持十进制数外，也支持十六进制数和八进制数。十六进制数以 0X 或 0x 开始，9 以上的数以 a～f 表示；八进制数以 0 开始，八进制数中只能出现 0～7 的数值。看下面的代码：

程序清单：codes\04\4.3\octal.html

```
<script>
    //显式定义变量 a
    var a;
    //使用十六进制方式给 a 赋值
    a = 0x13;
    //显式定义变量 b
    var b;
    //使用八进制方式给 b 赋值
    b = 014;
    //使用警告对话框输出两个变量的值
    alert(a + "---" + b);
</script>
```

代码的运行结果如图 4.8 所示。

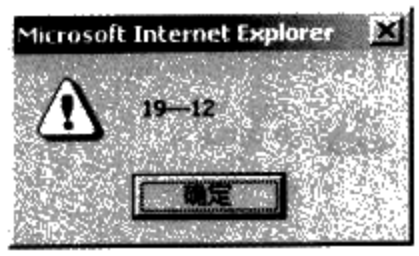


图 4.8　十六进制和八进制数的输出结果

正如期望的：0x13 转换成十进制数为 19，而 014 转换成十进制数为 12 。

注意：

由于 HTML 代码很多地方都需要使用十六进制数，因此，十六进制数是非常有用的。八进制的数并不是所有的浏览器都能支持，如需使用八进制数，请先确定代码运行的浏览器支持八进制数。

当数值变量的值超出了其表值范围时，将出现两个特殊值：Infinity（无穷大）和-Infinity（负无穷大）。前者表示数值大于数值类型的最大值，而后者表示数值小于数值类型的最小值。看如下代码：

程序清单：codes\04\4.3\infinity.html

```
<script>
    //定义 x 为最大的数值
    var x = 1.7976931348623157e308;
    //再次增加 x 的值
    x = x + 1e292;
    //使用警告对话框输出 x 的值
    alert(x);
</script>
```

代码的输出结果如图 4.9 所示。

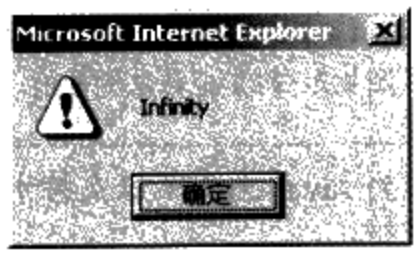


图 4.9　数值变量超出数值类型的表值范围

类似地，如果变量值小于数值变量的最小值将出现-Infinity 值。看如下代码：

程序清单：codes\04\4.3_infinity.html

```
<script>
    //定义 x 为最小的数值
    var x = -1.7976931348623157e308;
    //再次减少 x 的值
    x = x -1e292;
    //使用警告对话框输出 x 的值
    alert(x);
</script>
```

代码的执行结果将显示 x 的值为-Infinity。

注意：

Infinity、-Infinity 与其他数值进行算术运算时，整个算术表达式将变成另一个特殊值：NaN。但 Infinity 和-Infinity 都可以执行比较运算，即 Infinity 等于 Infinity，而-Infinity 等于-Infinity。

看下面的代码：

程序清单：codes\04\4.3\infinityArith.html

```
<script>
    //定义 y 为最小的数值
    var y = -1.79769313486231570e308;
    //再次减少 y 的值
    y = y - 1e292;
    //使用警告对话框输出 y 的值
    alert(y);
    //使用警告对话框输出 y 执行算术运算表达式的值
    alert(y + 3E3000);
    //定义 a 为 Infinity
    a = Number.POSITIVE_INFINITY;
    //定义 b 为-Infinity
    b = Number.NEGATIVE_INFINITY;
    //使用警告对话框输出 a+b 的值
    alert(a + b);
</script>
```

执行的结果是：第一次弹出警告框的值为-Infinity，后面两次警告框的值为 NaN。

但两个 Infinity 的值是相等的，看如下代码：

程序清单：codes\04\4.3\infinityEqual.html

```
<script>
    //使用显式定义变量的方式定义变量 a
    var a = 3e30000;
    //使用显式定义变量的方式定义变量 b
    var b = 5e20000;
    //比较 a 是否等于 b
    alert(a == b);
</script>
```

上面的变量定义代码中，a 的值与 b 的值明显不相等，但因为 a 和 b 都超出了数值的表示范围，执行结果是 a 与 b 的值相等。

注意：

JavaScript 中的算术运算允许除数为 0（除数和被除数也可同时为零，得到结果为 NaN），正数除零的结果就是 Infinity，负数除零的结果就是-Infinity。

NaN 是另一个特殊的数值，它是 Not a Number 三个单词的首字母缩写，表示非数。0 除 0，或者以 Infinity 执行算术运算都将产生 NaN 的结果。当然，如果算术表示中有个 NaN 的数值变量，整个算术表达式的值为 NaN。

注意：

NaN 与 Infinity 和-Infinity 不同的是，NaN 不会与任何数值变量相等，也就是 NaN==NaN 也返回 false。那如何判断某个变量是否为 NaN 呢？JavaScript 专门提供了 isNaN()函数来判断某个变量是否为 NaN。

例如下面的代码：

程序清单：codes\04\4.3\judgeNaN.html

```
<script>
    //定义 x 的值为 NaN
    var x = 0 / 0;
```

```
    //判断两个 NaN 是否相等
    if (x != x)
    {
        alert("NaN 不等于 NaN");
    }
    //调用 isNaN 判断变量
    if (isNaN(x))
    {
        alert("x 是一个 NaN");
    }
</script>
```

代码执行结束，将弹出两个警告对话框：表明两个 NaN 互不相等。isNaN()是 JavaScript 的内嵌函数，用于判断某个数值型变量是否为“非数”。

JavaScript 也提供了一些简单的方法来访问这些特殊值，特殊值通过 JavaScript 的内嵌类 Number 访问，访问方式如表 4.3 所示：

表 4.3　Number 类的常量与特殊值的对应

Number 类的常量	特殊值
Number.MAX_VALUE	数值型变量允许的最大值
Number.MIN_VALUE	数值型变量允许的最小值
Number.POSITIVE_INFINITY	Infinity（正无穷大）
Number.NEGATIVE_INFINITY	-Infinity（负无穷大）
Number.NaN	NaN（非数）

关于浮点型数，必须注意其精度问题。看如下代码：

程序清单：codes\04\4.3\losePrecision.html

```
<script>
    //显式定义变量 a
    var a = .3333;
    //定义变量 b，并为其赋值为 a * 5
    var b = a * 5;
    //使用对话框输出 b 的值
    alert(b);
</script>
```

在上面的代码中，a * 5 的值理论上为 1.6665，实际的结果如图 4.10 所示。

这种由于浮点数计算产生的问题，在很多语言中都会出现。对于浮点数值的比较，尽量不要直接比较，例如直接比较 b 是否等于 1.6665，将返回不相等。为了得到 1.6665 与 b 相等的结果，推荐使用差值比较法——判断两个浮点型变量是否相等，可以计算两个浮点型变量的差值，只要差值小于一个足够小的数即可认为相等。

图 4.10　0.3333*5 的输出结果

➤➤4.3.2　字符串类型

JavaScript 的字符串类型必须以引号括起来，此处的引号既可以是单引号，也可以是双引号。例如下面两种定义字符串变量的方式都是允许的：

```
a ="Hello JavaScript";
b ='Hello JavaScript';
```

这两种方式都是允许的，且 a 与 b 两个变量完全相等。

JavaScript 中没有字符类型，或者说字符类型和字符串类型是完全相同的，即使：

```
var a='a';
```

这行代码定义的 a 依然是字符串类型的变量，没有字符类型变量。

注意：

JavaScript 中字符串与 Java 中字符串主要有两点区别：1.JavaScript 中字符串可以用单引号引起来。2. JavaScript 中比较两个字符串的字符序列是否相等使用==即可，无须使用 equals()方法。

JavaScript 以 String 内建类来表示字符串，String 类有如下基本方法和属性可用于操作字符串：

- String()：类似于面向对象语言中的构造器，使用该方法可以构建一个字符串。
- charAt()：获取字符串特定索引处的字符。
- charCodeAt()：返回字符串中特定索引处的字符所对应的 Unicode 值。
- length：属性，直接返回字符串长度。JavaScript 中的中文字符算一个字符。
- toUpperCase()：将字符串的所有字母转换成大写字母。
- toLowerCase()：将字符串的所有字母转换成小写字母。
- fromCharCode()：静态方法，直接通过 String 类调用该方法，将系列 Unicode 值转换成字符串。
- indexOf()：返回字符串中特定字符串第一次出现的位置。
- lastIndexOf()：返回字符串中特定字符串最后一次出现的位置。
- substring()：返回字符串的某个子串。
- slice()：返回字符串的某个子串，功能比 substring 更强大，支持负数参数。
- match()：使用正则表达式搜索目标子字符串。
- search()：使用正则表达式搜索目标子字符串。
- concat()：用于将多个字符串拼接成一个字符串。
- split()：将某个字符串分割成多个字符串，可以指定分隔符。
- replace()：将字符串中某个子串以特定字符串替代。

下面的代码测试了 String 类的几个简单属性和方法。

程序清单：codes\04\4.3\StringMethod.html

```
<script>
    //定义字符串变量 a
    var a = "abc 中国";
    //获取 a 的长度
    var b = a.length;
    //将系列的 Unicode 值转换成字符串
    var c = String.fromCharCode(97,98,99);
    //输出 a 的长度，字符串 a 在索引 4 处的字符和
    //对应的 Unicode 值，以及字符串变量 c 的值
    alert(b + "---" + a.charAt(4) + "---"
        + a.charCodeAt(4) + "---" + c);
</script>
```

代码执行的结果如图 4.11 所示。

indexOf 和 lastIndexOf 用于判断某个子串的位置。其语法格式为：

- indexOf(searchString [, startIndex])：搜索目标字符串 searchString 出现的位置。其中 startIndex 指定不搜索左边 startIndex 个字符。

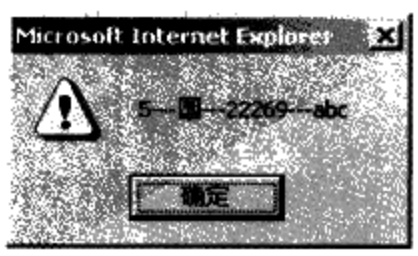


图 4.11　字符串函数的测试结果

- lastIndexOf(searchString [, startIndex])：搜索目标字符串 searchString 最后一次出现的位置。如果字符串中没有包含目标字符串，则返回-1。功能更强大的搜索函数是 search()函数，它使用正则表达式搜索。

看下面的代码：

程序清单：codes\04\4.3\StringSearch.html

```
<script>
    var a = "hellojavascript";
    //搜索 llo 子串出现的位置
    var b = a.indexOf("llo");
    //跳过左边 3 个字符，开始搜索 llo 子串
    var c = a.indexOf("llo" , 3);
    //搜索 a 子串最后一次出现的位置
    var d = a.lastIndexOf("a");
    alert(b + "\n" + c + "\n" + d);
</script>
```

输出的 b 值为 2，而 c 值为-1，d 值为 8。-1 表示 a 字符串从索引 3 处开始搜索，无法找到 llo 子串。a 字符串中最后一次出现 a 的位置为 8。

注意：

与 Java 字符串里字符索引类似，JavaScript 字符串里第一个字符的索引是 0，而不是 1。

JavaScript 中的 substring()和 slice()语法如下：

- substring(start [, end])：从 start（包括）索引处，截取到 end（不包括）索引处，不截取 end 索引处的字符。如果没有 end 参数，将从 start 处一直截取到字符串尾。
- slice(start[, end])：与 substring 的功能基本一致，区别是 slice 可以接受负数作为索引，但使用负索引值时，表示从字符串的右边开始计算索引，即最右边的索引为-1。

看如下代码：

程序清单：codes\04\4.3\StringSlice.html

```
<script>
    var s = "abcdefg";
    //取得第 1 个(包括)到第 5 个(不包括)的子串
    a = s.slice(0 , 4);
    //取得第 3 个(包括)到第 5 个(不包括)的子串
    b = s.slice(2 , 4);
    //取得第 5 个(包括)到最后的子串
    c = s.slice(4);
    //取得第 4 个(包括)到倒数第 1 个(不包括)的子串
    d = s.slice(3 , -1);
    //取得第 4 个(包括)到倒数第 2 个(不包括)的子串
    e = s.slice(3 , -2);
    //取得倒数第 3 个(包括)到倒数第 1 个(不包括)的子串
    f = s.slice(-3 , -1);
    alert("a : " + a + "\nb : "
        + b + "\nc : "
        + c + "\nd : "
        + d + "\ne : "
        + e + "\nf : "
        + f );
</script>
```

执行的结果如图 4.12 所示。

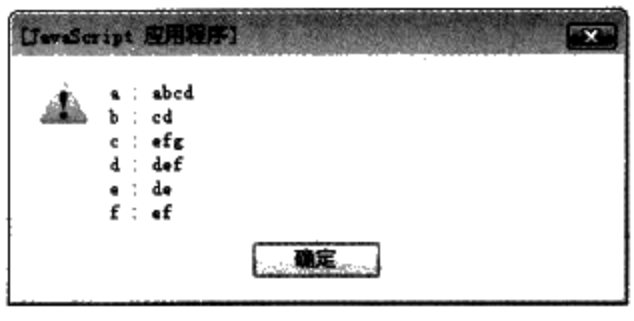


图 4.12　字符串 slice 方法的执行结果

match 和 search 方法都支持使用正则表示式作为子串。区别是前者返回匹配的子字符串，而后者返回匹配的索引值。match 支持使用全局匹配，通过使用 g 标志来表示全局匹配，match 方法返回所有匹配正则表达式的子串所组成的数组。

match 方法的返回值为字符串数组或 null，如果包含匹配值，将返回字符串数组；否则将返回 null。search 方法的返回值为整型变量，如果搜索到匹配子串，则返回子串的索引值；否则返回-1。

下面的代码示范了 search 和 match 方法的用法。

程序清单：codes\04\4.3\StringRegex.html

```
<script>
    //定义字符串 s 的值
    var s = "abfd--abc@d.comcdefg";
    //从 s 中匹配正则表达式
    a = s.search(/[a-z]+@d.[a-zA-Z]{2}m/);
    //定义字符串变量 str
    var str = "1dfd2dfs3df5";
    //查找字符串中所有单个的数值
    var b = str.match(/\d/g);
    //输出 a 和 b 的值
    alert(a + "\n" + b);
</script>
```

图 4.13　search 和 match 方法的效果

代码的执行结果如图 4.13 所示。

从上述执行结果可以看出，a 的值为 6，这表明目标字符串中和正则表达式匹配的第一个子串的位置是 6。正则表达式匹配的子串是“abc@d.com”。

match 方法在正则表达式后增加了 g 选项，表明执行全局匹配。匹配的结果返回一个数组，数组元素是目标字符串中的所有数值。

如果需要在字符串中使用单引号、双引号等特殊字符，则必须使用转义字符。JavaScript 的转义字符依然是\，下面是常用的转义字符：

- \b：代表退格。
- \t：表示一个制表符，即一个 Tab 空格。
- \n：换行回车。
- \v：垂直制表符。
- \r：回车。
- \"：双引号。
- \'：单引号。
- \\：反斜线，即\。
- \OOO：使用八进制数表示的拉丁字母。OOO 表示一个三位的八进制整数，范围是 000～377。
- \xHH：使用十六进制数表示的拉丁字母。HH 表示一个两位的 16 进制整数，范围是 00～FF。
- \uHHHH：使用十六进制数（该数值指定该字符的 Unicode 值）表示的字符。HHHH 表示一个四位的十六进制整数。

➤➤4.3.3　布尔类型

布尔类型的值只有两个，即 true 和 false。布尔类型的值通常是逻辑运算的结果，或者用于标识对

象的某种状态。例如使用如下代码判断浏览器是否允许使用 cookie：

程序清单：codes\04\4.3\boolean.html

```
<script>
    //如果浏览器支持 Cookie
    if (navigator.cookieEnabled)
    {
        alert("浏览器允许使用 Cookie");
    }
    //如果浏览器不支持 Cookie
    else
    {
        alert("浏览器禁用 Cookie");
    }
</script>
```

笔者的浏览器支持 Cookie，执行结果如图 4.14 所示。

图 4.14 判断浏览器是否支持 Cookie

➤➤4.3.4 undefined 和 null

undefined 类型的值只有 undefined 一个。该值用于表示某个变量不存在，或者没有为其分配值，也用于表示对象的属性不存在。null 用于表示变量的值为空。undefined 与 null 之间的差别比较微妙，总体而言：undefined 表示没有为变量设置值，而 null 表示将变量值设为空。

实际上，很多时候 undefined 和 null 本身就相等，即 null==undefined 将返回 true。如果我们要精确区分 null 和 undefined，应该考虑使用精确等于符（===）。

看如下代码：

```
var x = String.abc;
```

x 的值为 String 类一个并不存在的属性 abc，因此 x 的值为 undefined。

关于 undefined 和 null 的区别看如下代码：

程序清单：codes\04\4.3\undefined.html

```
<script>
    //声明变量 x , y
    var x , y = null;
    //判断 x 的值是否为空
    if (x === undefined)
    {
        alert('声明变量后默认值为 undefined');
    }
    if (x === null)
    {
        alert('声明变量后默认值为 null');
    }
    //判断 x（其值为 undefined）是否与 y（其值为 null）相等
    if (x == y)
    {
        alert("x (undefined) ==y (null) ");
    }
    //测试一个并不存在的属性
    if(String.xyz === undefined)
    {
        alert("不存在的属性值默认为 undefined");
    }
</script>
```

代码执行的结果是 x 为 undefined，且 x==y 返回真，String 的 xyz 属性值为 undefined。

注意：

定义一个变量后，如果没有为该变量赋值，则该变量的值默认为 undefined，这个值是系统默认分配的。访问对象并不存在属性时，该属性值也将返回 undefined。

与 null 不同的是，undefined 并不是 JavaScript 中的保留字，在 ECMAScript 3 标准规范中，undefined 是一个全局变量，其值就是 undefined——在这种情况下，我们把 undefined 当成关键字处理即可。某些浏览器可能不支持 undefined 值，此时可以在 JavaScript 脚本的第一行定义如下变量：

```
var undefined;
```

➤➤4.3.5 正则表达式

本节不打算详细叙述正则表达式林林总总的概念和各种细节，如果读者需要了解关于正则表达式更多的细节，可以参考 9.5 节的内容。

正则表达式的实质是一种特殊字符串，这种特殊字符串允许使用“通配符”，因此一个正则表达式字符串可以匹配一批普通字符串。从这个意义上来看，任意一个普通字符串也可算做正则表达式，只是该正则表达式里不包含“通配符”，因而它只能匹配一个字符串。

JavaScript 的正则表达式必须放在两条斜线之间，如/abc/就是一个正则表达式，只是这个正则表达式只能匹配"abc"字符串。

正则表达式中可以使用的普通字符如表 4.4 所示。

表 4.4 正则表达式所支持的合法字符

字　符	解　释
x（x 可代表任何合法的字符）	字符 x
\0mnn	八进制数 0mnn 所表示的字符
\xhh	十六进制值 0xhh 所表示的字符
\uhhhh	十六进制值 0xhhhh 所表示的 Unicode 字符
\t	制表符（'\u0009'）
\n	新行（换行）符（'\u000A'）
\r	回车符（'\u000D'）
\f	换页符（'\u000C'）
\a	报警（bell）符（'\u0007'）
\e	Escape 符（'\u001B'）
\cx	x 对应的控制符。例如，\cM 匹配 Ctrl-M。x 值必须为 A~Z 或 a~z 之一。

正则表达式所支持的“通配符”如表 4.5 所示。

表 4.5 正则表达式的“通配符”

预定义字符	说　明
.	可以匹配任何字符
\d	匹配 0~9 的所有数字
\D	匹配非数字
\s	匹配所有空白字符，包括空格、制表符、回车符、换页符、换行符等
\S	匹配所有非空白字符

续 表

预定义字符	说　明
\w	匹配所有单词字符，包括数字 0~9，26 个英文字母和下画线（_）
\W	匹配所有非单词字符 。
[]表示法	这种表示法最为灵活：例如[a-z]表示 a 到 z 之间任意一个字符，[a-z0-9]表示 a 到 z 或 0 到 9 之间的任意一个字符，[\u4e00-\u9fff]匹配任意一个汉字（u4e00 到 u9fff 是汉字的 Unicode 码值范围）
$	匹配一行的结尾。要匹配 $ 字符本身，请使用 \$
^	匹配一行的开头。要匹配 ^ 字符本身，请使用 \^

提　示：

上面的表格中前 7 个“通配符”其实很容易记忆：d 是 digit 的意思，代表数字；s 是 space 的意思，代表空白；w 是 word 的意思，代表单词。d、s、w 的大写形式恰好匹配与之相反的字符。

记住了这些“通配符”之后，还需要记住如表 4.6 所示的频率修饰词。

表 4.6　频率修饰词

特殊字符	说明
?	指定前面的子表达式可以出现零次或一次。要匹配 ? 字符，请使用 \?
*	指定前面的子表达式可以出现零次或多次。要匹配 * 字符，请使用 *
+	指定前面的子表达式可以出现一次或多次。要匹配 + 字符，请使用 \+
{m,n}表示法	这种表示法最灵活，前面子表达式最少出现 m 次，最多出现 n 次。m、n 两个数值都可以省略，如果省略 m，表示最少可出现 0 次；如果省略 n，表示最多可出现无限多次

除此之外，正则表达式还支持()表示法，用()可以将一个表达式形成一个固定组。除此之外，还可以在()内使用竖线表示互斥，例如/((abc)|(efg))/可匹配 abc 或 efg。

掌握上面三个表格的内容之后，就学会了正则表达式的基本用法了。JavaScript 的正则表达式提供了一个 test()方法，用于判断该正则表达式是否匹配某个字符串。

除此之外，JavaScript 字符串的 replace()方法也可使用正则表达式，考虑到 JavaScript 没有提供截去字符串前后空白（包括空格、制表符等）的方法，下面利用正则表达式和 replace()实现一个 trim()方法。代码如下：

程序清单：codes\04\4.3\regex.html

```
<script>
//用正则表达式来匹配超级链接
alert(/^<a href=(\'|\")[a-zA-Z0-9\/:\.]*(\'|\")>.*<\/a>$/
    .test("<a href='http://www.leegang.org'>疯狂 Java 联盟</a>"));

function trim(s)
{
    // \s 匹配任何空白字符，包括空格、制表符、换页符等
    //其中^\s*匹配字符串前面的多个空格，\s*$匹配字符串后面的多个空格
    // /g 表示尽可能多地匹配
    //最后将所有匹配的内容替换成""（即截取前、后两端的空格）
    return s.replace(/(^\s*)|(\s*$)/g,"");
}
//示范截去前后两端的空白
alert(trim('   Hello,JavaScript   '));
</script>
```

从上面的代码可以看到正则表达式的强大功能。实际上，正则表达式正是 JavaScript 的强大工具之一。

4.4 复合类型

复合类型是由多个基本数据类型（也可以包含复合类型）组成的数据体。JavaScript 中的复合类型大致上有如下三种：

- Object：对象。
- Array：数组。
- Function：函数。

下面依次介绍这三种复合类型：

➤➤4.4.1 对象

对象是一系列命名变量和函数的集合。其中的命名变量的类型可以是基本数据类型，也可以是复合类型。对象中的命名变量称为属性，而对象中的函数称为方法。对象访问属性和函数的方法都是通过“.”，例如如下代码用于判断浏览器的版本：

```
//获得浏览器版本
alert("浏览器的版本为: " + navigator.appVersion);
```

正如前文提到的，JavaScript 是基于对象的脚本语言，它提供了大量的内置对象供用户使用。除此之外，JavaScript 还提供了如下常用的内置类：

- Array：数组类。
- Date：日期类。
- Error：错误类。
- Function：函数类。
- Math：数学类，该对象包含相当多执行数学运算的方法。
- Number：数值类。
- Object：对象类。
- String：字符串类。

关于 JavaScript 对象在 4.10 节、4.11 节还有更详细的介绍，读者可参阅后面的介绍了解 JavaScript 对象的更多详细信息，此处不再赘述。

➤➤4.4.2 数组

数组是一系列的变量。与其他强类型语言不同的是，JavaScript 的数组中元素的类型可以不相同。定义一个数组有如下三种语法格式：

```
var a = [3 , 5 , 23];
var b = [];
var c = new Array();
```

第一种在定义数组时已为数组完成数组元素的初始化，第二种和第三种都只创建一个空数组。

看如下代码：

程序清单：codes\04\4.4\arr.html

```
<script>
    //定义一个数组，定义时直接给数组元素赋值
    var a = [3 , 5 , 23];
    //定义一个空数组
```

```
        var b = [];
        //定义一个空数组
        var c = new Array();
        //直接访问元素赋值
        b[0] = 'hello';
        //直接为数组元素赋值
        b[1] = 6;
        //直接为数组元素赋值
        c[5] = true;
        //直接为数组元素赋值
        c[7] = null;
        //输出三个数组值和数组长度
        alert(a + "\n" + b + "\n" + c
            + "\na 数组的长度:" + a.length
            + "\nb 数组的长度:" + b.length
            + "\nc 数组的长度:"+ c.length);
    </script>
```

代码执行结果如图 4.15 所示。

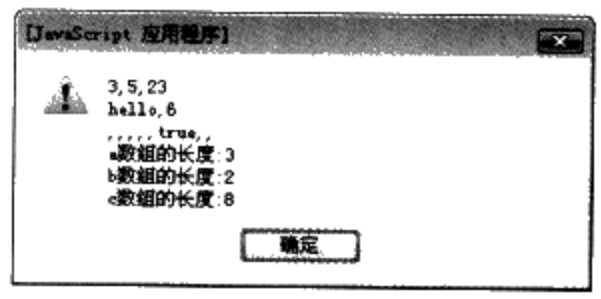


图 4.15　数组的输出结果

正如代码中所看到的：JavaScript 中数组的元素并不要求相同，同一个数组中的元素类型可以互不相同。

JavaScript 为数组提供了一个 length 属性，通过该属性可得到数组的长度。JavaScript 数组的长度可以随意变化，它总等于所有元素索引最大值 + 1。

注意：

JavaScript 的数组索引从 0 开始。

JavaScript 作为动态弱类型语言，其数组归纳起来有如下三个特征：

- JavaScript 数组的长度可变。
- 同一个数组里数组元素的类型可以互不相同。
- 访问数组元素时不会产生数组越界，访问并未赋值的数组元素时，该元素的值为 undefined。

4.4.3　函数

函数是 JavaScript 中另一种复合类型。函数可以包含一段可执行性代码，也可以接收调用者传入的参数。正如所有弱类型语言一样：JavaScript 的函数声明中，参数列表不需要数据类型说明，函数的返回值也不需要数据类型说明。函数定义的语法格式如下：

```
function functionName(param1,param2,...)
{
}
```

下面的代码定义了一个简单的函数。

程序清单：codes\04\4.4\simpleFunction.html

```
<script>
    //定义一个函数，定义函数无须定义返回值类型，也无须声明变量类型
    function judgeAge(age)
    {
        //如果参数值大于 60
        if(age > 60)
        {
            alert("老人");
```

```
        }
        //如果参数值大于 40
        else if(age > 40)
        {
            alert("中年人");
        }
        //如果参数值大于 15
        else if(age > 15)
        {
            alert("青年人");
        }
        //否则
        else
        {
            alert("儿童");
        }
    }
    //调用函数
    judgeAge(46);
</script>
```

上面的代码定义了一个简单的函数，然后通过 judgeAge(46)调用函数。代码执行的结果是“中年人”。

调用函数的语法如下：

```
functionName(value1,value2...);
```

上面的函数存在一个小小的问题：如果传入的参数不是数值会怎样呢？为了让程序更加严谨，应该先判断参数的数据类型，判断变量的数据类型使用 typeof 运算符，该运算符用于返回变量的数据类型。关于 typeof 运算符的介绍参见 4.5 节。

为了让上面的函数更加严谨，可以将其修改为：

```
function judgeAge(age)
{
    //要求 age 参数必须是数值
    if( typeof age === "number" )
    {
        //如果参数值大于 60
        if(age > 60)
        {
            alert("老人");
        }
        //如果参数值大于 40
        else if(age > 40)
        {
            alert("中年人");
        }
        //如果参数值大于 15
        else if(age > 15)
        {
            alert("青年人");
        }
        //否则
        else
        {
            alert("儿童");
        }
    }
```

```
        else
        {
            alert("参数必须为数值");
        }
}
```

JavaScript 中的函数与 Java 中的方法有些类似之处，归纳起来主要有如下四点区别：

- JavaScript 函数无须声明返回值类型。
- JavaScript 函数无须声明形参类型。
- JavaScript 中函数可以独立存在，无须属于任何类。
- JavaScript 函数必须使用 function 关键字定义。

4.5 运算符

JavaScript 也提供了相当丰富的运算符，运算符也是 JavaScript 语言的基础。通过运算符，可以将变量连接成语句，语句是 JavaScript 代码中的执行单位。

JavaScript 的运算符并不比其他高级语言的少，它同样提供了算术运算符、逻辑运算符、位运算符等。JavaScript 支持的运算符与 Java、C 所支持的非常相似。下面依次介绍 JavaScript 中的运算符。

➤➤4.5.1 赋值运算符

赋值运算符用于为变量指定变量值，与 Java、C 类似，JavaScript 也使用=作为赋值运算符号。通常，使用赋值运算符将一个常量值赋给变量，见如下示例代码。

程序清单：下面 4 段代码都来自 codes\04\4.5\assign.html

```
//为变量 str 赋值为"JavaScript"
var str = "JavaScript";
//为变量 pi 赋值为 3.14
var pi = 3.14;
//为变量 visited 赋值为 true
var visited = true;
```

除此之外，也可使用赋值运算符将一个变量的值赋给另一个变量。即如下代码也是正确的。

```
//为变量 str 赋值为"JavaScript"
var str = "JavaScript";
//将变量 str 的值赋给 str2
var str2 = str;
```

与 Java 类似的是，赋值语句本身是有值的，赋值语句的值就是等号（=）右边被赋的值。因此，赋值运算符支持连续赋值，通过使用多个赋值运算，可以一次为多个变量赋值，如下代码也是正确的。

```
//通过为 a, b, c,d 赋值，四个变量的值都是 7
var a = b = c = d = 7;
//输出四个变量的值
alert(a + '\n' + b + '\n' + c + '\n' + d);
```

赋值运算符还可用于将表达式的值赋给变量，如下代码也是正确的。

```
//为变量 x 赋值为 12.34
var x = 12.34;
//将表达式的值赋给 y
var y = x + 5;
//输出 y 的值
alert(y);
```

赋值运算符还可与其他运算符结合，成为功能更加强大的赋值运算符，参见 4.5.4 节。

➤➤4.5.2 算术运算符

JavaScript 支持所有的基本算术运算符，这些算术运算符号用于执行基本的数学运算：加、减、乘、除和求余等。下面是 7 个基本的算术运算符：

程序清单：下面 5 段代码都来自 codes\04\4.5\arith.html

+：加法运算符。例如如下代码：

```
var a = 5.2;
var b = 3.1;
var sum = a + b;
//sum 的值为 8.3
alert(sum);
```

-：减法运算符。例如如下代码：

```
var c = 5.2;
var d = 3.1;
var sub = c - d;
//sub 的值为 2.1
alert(sub);
```

*：乘法运算符。例如如下代码：

```
var e = 5.2;
var f = 3.1;
var product= e * f;
//product 的值为 16.12
alert(product);
```

/：除法运算符。例如如下代码：

```
var m = 36;
var n = 9;
var div = m / n;
//div 的值为 4
alert(div);
```

%：求余运算符。例如如下代码：

```
var x = 5.2;
var y = 3.1;
var mod = x % y;
//mod 的值为 2.1
alert(mod);
```

++：自加。这是个单目运算符，运算符既可以出现在操作数的左边，也可以出现在操作数的右边。但出现在左边和右边的效果是不一样的。看如下代码：

程序清单：codes\04\4.5\selfAdd1.html

```
<script>
    var a = 5;
    //让 a 先执行算术运算，然后自加
    var b = a++ + 6;
    alert(a + "\n" + b);
</script>
```

执行完后，a 的值为 6，而 b 的值为 11。当++在操作数的右边时，先执行算术运算，然后对操作数执行自加运算；当++在操作数的左边时，先执行自加，然后再执行算术运算。看如下代码：

程序清单：codes\04\4.5\selfAdd2.html

```
<script>
    var a = 5;
    //让 a 先执行自加，然后执行算术运算
    var b = ++a + 6;
    alert(a + "\n" + b);
</script>
```

执行的结果是 a 的值为 6，b 的值为 12，因为 b 的值等于 a 自加后（为 6）再加 6，也就是 12。

--：自减。也是个单目运算符，效果与++基本相似，只是将操作数的值减 1。

JavaScript 并没有提供其他更复杂的运算符，如需要完成乘方、开方等运算，可借助于 Math 类的方法来完成，见如下代码：

程序清单：codes\04\4.5\Math.html

```
<script>
    //定义变量 a 为 3.2
    var a = 3.2;
    //求 a 的 5 次方，并将计算结果赋给 b
    var b = Math.pow(a , 5);
    //输出 b 的值
    alert(b);
    //求 a 的平方根，并将结果赋给 c
    var c = Math.sqrt(a);
    //输出 c 的值
    alert(c);
    //计算随机数
    var d = Math.random();
    //输出随机数 d 的值
    alert(d);
</script>
```

Math 类下包含了丰富的静态方法，用于完成各种复杂的数学运算。

注意：

+ 除了可作为数学的加法运算符外，还可作为字符串的连接运算符。

- 除了可以作为减法运算符之外，还可以作为求负的运算符号。例如如下代码：

```
//定义变量 x，其值为-5
var x = -5;
//将 x 求负，其值变成 5
x = -x;
```

4.5.3 位运算符

JavaScript 支持的位运算符与 Java 所支持的基本相似，大致有如下 7 个：

- &：按位与。
- |：按位或。
- ~：按位非。
- ^：按位异或。
- <<：左位移运算符。
- >>：右位移运算符。
- >>>：无符号右移运算符。

按位与的运算法则如表 4.7 所示。

表 4.7 位运算符的运算结果表

第一个运算数	第二个运算数	按位与	按位或	按位异或
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

看如下代码：

程序清单：codes\04\4.5\bit.html

```
<script>
    //输出 5 & 9 和 5 | 9 的值
    alert(5 & 9);
    alert(5 | 9);
</script>
```

执行结果如图 4.16 所示。

程序执行的结果是 5 & 9 的结果是 1，5 | 9 的结果是 13。下面介绍运算的原理：

5 的二进制码是 00000101，而 9 的二进制码是 00001001。执行过程如图 4.17 所示。

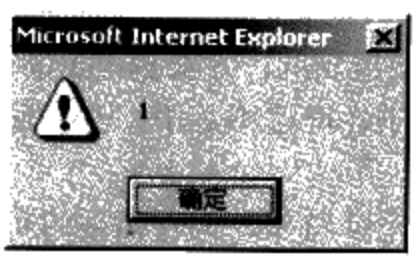


图 4.16 5&9 的结果

```
  00000101      00000101
& 00001001    | 00001001
----------    ----------
  00000001      00001101
```

图 4.17 位运算的过程

左位移运算符是将二进制码向左移动，右边以 0 补齐。看如下代码：

程序清单：codes\04\4.5\bitShift.html

```
<script>
    //输出 5 << 2 和 5 >> 2 的值
    alert(5 << 2);
    alert(5 >> 2);
</script>
```

代码运算的结果如图 4.18 所示。

5<<2 的结果是 20，而 5>>2 的结果是 1。下面介绍运算的原理：

5 的二进制码为 00000101，左移 2 位成为 00010100，即 20；右移两位成为 00000001，即 1。无符号右移与右移相似，>>运算后的左边以操作数二进制码的最高位补齐，而>>>运算后的左边以 0 补齐。

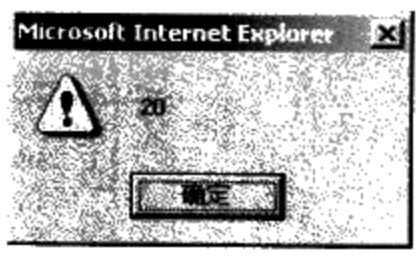


图 4.18 左位移运算的结果

➤➤4.5.4 加强的赋值运算符

赋值运算符可与算术运算符、位移运算符等结合为功能更加强大的运算符。结合后的加强运算符有：

- +=：对于 x += y，即对应于 x = x + y。
- -=：对于 x -= y，即对应于 x = x - y。
- *=：对于 x *= y，即对应于 x = x * y。
- /=：对于 x /= y，即对应于 x = x / y。
- %=：对于 x %= y，即对应于 x = x % y。
- &=：对于 x &= y，即对应于 x = x & y。

- |=：对于 x |= y，即对应于 x = x | y。
- ^=：对于 x ^= y，即对应于 x = x ^ y。
- <<=：对于 x <<= y，即对应于 x = x << y。
- >>=：对于 x >>= y，即对应于 x = x >> y。
- >>>=：对于 x >>>= y，即对应于 x = x >>> y。

归纳起来，赋值运算符可以和所有双目运算符结合，从而结合成功能更加强大的运算符。

➤➤4.5.5 比较运算符

比较运算符用于判断两个变量或常量的大小，比较运算的结果是一个布尔值。JavaScript 支持的比较运算符有：

- >：大于，如果前面变量的值大于后面变量的值，返回 true。
- >=：大于等于，如果前面变量的值大于等于后面变量的值，返回 true。
- <：小于，如果前面变量的值小于后面变量的值，返回 true。
- <=：小于等于，如果前面变量的值小于等于后面变量的值，返回 true。
- !=：不等于，如果前后两个变量的值不相等，返回 true。
- ==：等于，如果前后两个变量的值相等，返回 true。
- !==：严格不等于，如果前后两个变量的值不相等，或者数据类型不同，都将返回 true。
- ===：严格等于，必须前后两个变量的值相等，数据类型也相同，才会返回 true。

上面的比较运算符中，前面 5 个比较常见。但后面的严格等于、严格不等于，与普通等于、普通不等于的区别在于是否支持自动类型转换。

正如前面介绍的，JavaScript 支持自动类型转换，"5"本来是个字符串，但在需要时可以自动转换成数值型。因此，由于自动类型转换的缘故，5 =="5"将返回 true。看如下代码：

程序清单：codes\04\4.5\compare.html

```
<script>
    //判断 5 是否等于"5"
    alert(5 == "5");
    //判断 5 是否严格等于"5"
    alert(5 === "5");
</script>
```

第一次弹出如图 4.19 所示的对话框。

第二次弹出如图 4.20 所示的对话框。

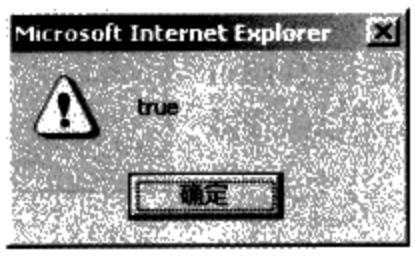


图 4.19 5=="5"判断的结果

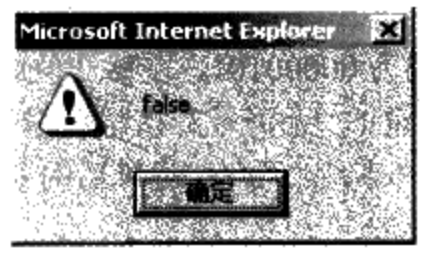


图 4.20 5==="5"判断的结果

其中==和===的区别在于：==支持自动类型转换，只要前后两个比较变量的值相等，即使数据类型不同也返回 true，而===则要求两个参与比较的变量的值相等，且数据类型相同才返回 true。

值得注意的是，比较运算符不仅可以在数值之间进行，也可以在字符串之间进行。字符串的比较规则是按字母的 Unicode 值进行比较。对于两个字符串，先比较它们的第一个字母，其 Unicode 值大的字符串大，如果它们的第一个字母相同，则比较第二个字母，依此类推。看如下代码：

程序清单：codes\04\4.5\strCompare.html

```
<script>
    //先比较第一个字母，z 的 Unicode 值比 a 的 Unicode 值大，返回真
    alert("z" > "abc");
    //先比较第一个字母，a 的 Unicode 值比 X 的 Unicode 值大，返回真
    alert("abc" > "XYZ");
    //前两个字母相同，比较第三个字母，C 的 Unicode 值比 B 的 Unicode 值大，返回真
    alert("ABC" > "ABB");
</script>
```

4.5.6 逻辑运算符

逻辑运算符用于操作两个布尔型的变量或常量。逻辑运算符主要有如下 3 个：

- &&：与，必须前后两个操作数都是 true 才返回 true，否则返回 false。
- ||：或，只要两个操作数中有一个为 true，就返回 true，否则返回 false。
- !：非，只操作一个操作数。如果操作数为 true，返回 false；如果操作数为 false，返回 true。

如下代码示范了逻辑运算符的功能：

程序清单：codes\04\4.5\logic.html

```
<script>
    //直接对 false 求非运算，将返回 true
    alert(!false);
    //5>3 返回 true，'6'自动类型转换为整数 6，6>10 返回 false，求与后返回 false
    alert(5 > 3 && '6' > 10);
    //4>=5 返回 false，"abc">"abb"返回 true。求或返回 true
    alert(4 >= 5 || "abc" > "abb");
    //4>=5 返回 false，"abc">"abb"返回 true。求异或返回 true
</script>
```

值得指出的是，JavaScript 虽然没有提供|（在 Java 中被称为不短路或）、&（在 Java 中被称为不短路与）、^（异或）等运算符，但实际上我们依然可将它们当成逻辑运算符使用。看如下代码：

程序清单：codes\04\4.5\fakeLogic.html

```
<script>
    //使用位运算符代替逻辑运算符
    alert( 6 > 5 | 3 > 4);
    alert( true ^ false);
</script>
```

执行上面的代码将输出 2 个 1，但根据 JavaScript 的自动类型转换规则，当数值 1 转换为布尔类型变量时，将会得到 true。从这个意义上来说，我们完全可以将|、&和^当成逻辑运算符使用。

当把|当成逻辑运算符使用时，该运算符将会变成不短路或，看如下代码：

程序清单：codes\04\4.5\bitOr.html

```
<script>
    //定义变量 a，b，并为其赋值
    var a = 5;
    var b = 10;
    //如果 a>4 或 b>10
    if (a > 4 | b++ > 10)
        alert(a + '\n' + b);
</script>
```

代码的执行结果如图 4.21 所示。

图 4.21 “不短路或”的执行效果

再看如下代码，下面的代码只是将上面示例中的“不短路或”改成了短路或（||），代码如下：

程序清单：codes\04\4.5\or.html

```
<script>
    //定义变量 a,b，并为其赋值
    var a = 5;
    var b = 10;
    //如果 a>4 或 b>10
        if (a > 4 || b++ > 10)
        alert(a + '\n' + b);
</script>
```

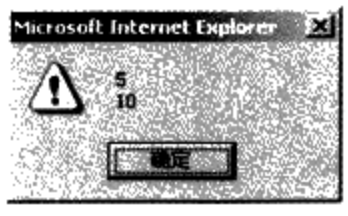


图 4.22　短路或（||）的执行效果

代码的执行结果如图 4.22 所示。

仅仅将按位或（不短路或）改成短路的逻辑或，程序最后输出的 b 值不再相同。因为对于短路的逻辑或（||）而言， 如果第一个操作数返回 true，|| 将不再对第二个操作数求值，直接返回 true。不会计算 b++ > 10 这个逻辑表达式，因而 b++没有获得执行的机会。因此，最后输出的 b 为 10。而按位或（|）总是执行前后两个操作数。

&与&&的区别类似：&总会计算前后两个操作数，而&&先计算左边的操作数，如果左边的操作数为 false，则直接返回 false，而根本不会计算右边的操作数。

➤➤4.5.7　三目运算符

三目运算符只有一个“?:”，三目运算符的语法格式如下：

```
(expression) ? if-true-statement : if-false-statement;
```

三目运算符的规则是：先对逻辑表达式 expression 求值，如果逻辑表达式返回 true，则执行第二部分的语句；如果逻辑表达式返回 false，则返回第三部分的语句。看如下代码：

程序清单：codes\04\4.5\three.html

```
<script>
    //使用三目运算符
    5 > 3 ? alert("5 大于 3") : alert("5 小于 3") ;
</script>
```

代码的运行结果如图 4.23 所示。

图 4.23　三目运算符的运算结果

大部分时候，三目运算符都用做 if else 的精简写法。只要 if else 的条件执行体都只有一条语句，我们就可以将这种写法换成三目运算符写法。看下面的代码：

```
//如果 5 大于 3，将执行下面的代码块
if(5 > 3)
{
    alert("5 大于 3");
}
//否则，执行下面的代码块
else
{
    alert("5 小于 3");
}
```

这两种代码写法的效果是完全相同的。三目运算符和 if else 的写法的区别在于：if 后的代码块可以有多条语句，而三目运算符是不支持多条语句的。看如下代码：

```
//如果 5 大于 3，将执行下面的代码块
if(5 > 3)
{
    alert("多行语句");
    alert("5 大于 3");
```

```
}
//否则，执行下面的代码块
else
{
    alert("多行语句");
    alert("5 小于 3");
}
```

对于上面的代码块，则无法转换成三目运算符。换成如下语句是无法正常运行的：

```
5 > 3 ? alert("多行语句");alert("5 大于 3") : alert("多行语句");alert("5 小于 3")
```

➤➤4.5.8 逗号运算符

逗号运算符允许将多个表达式排在一起，最后返回最右边表达式的值。看下面的代码：

程序清单：codes\04\4.5\comma.html

```
<script>
    //声明变量 a , b , c , d
    var a , b , c , d;
    //使用逗号运算符为 a 赋值，最右边的表达式为 56，因此 a 的值为 56
    a = (b = 5, c = 7, d = 56);
    //输出四个变量的值
    document.write('a = ' + a + ' b = '
        + b + ' c = ' + c + ' d = ' + d);
</script>
```

以上代码的执行结果如图 4.24 所示。

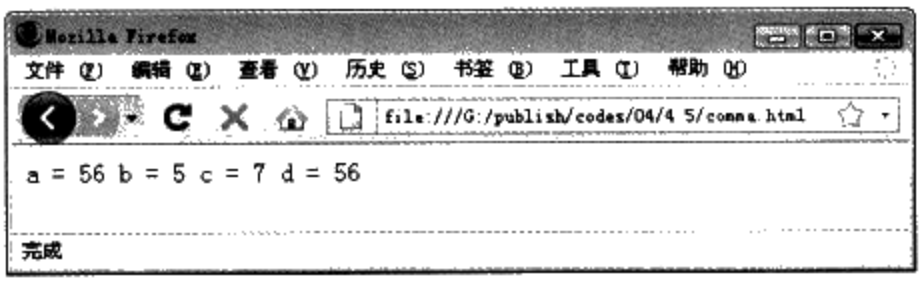


图 4.24 使用逗号运算符的结果

注意：

函数的参数列表也使用逗号作为分隔符，但参数列表中的逗号不是运算符。

➤➤4.5.9 void 运算符

void 运算符用于强行指定表达式不返回值。看如下代码：

程序清单：codes\04\4.5\void.html

```
<script>
    //声明变量 a,b,c,d
    var a , b , c , d;
    //使用逗号运算符为 a 赋值，最右边的表达式为 56，因此 a 的值为 56
    a = void(b = 5, c = 7, d = 56);
    //输出四个变量的值
    document.write('a = ' + a + ' b = '
        + b + ' c = ' + c + ' d = ' + d);
</script>
```

对(b=5, c=7, d=56)表达式使用 void 运算符，强制指定该表达式没有返回值。因此 a 没有值，代码

执行的结果如图 4.25 所示。

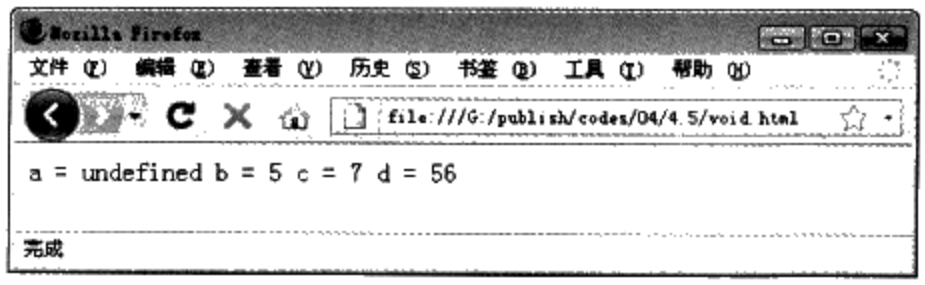


图 4.25 使用 void 运算符的结果

➤➤4.5.10 typeof 和 instanceof 运算符

typeof 运算符用于判断某个变量的数据类型，可作为函数来用，例如 typeof(a)，可返回变量 a 的数据类型；还可以作为一个运算符来使用，例如 typeof a，也可返回变量 a 的数据类型。

不同类型参数使用 typeof 运算符的返回值如下：

- undefined 值：undefined。
- null 值：object。
- 布尔型值：boolean。
- 数字型值：number。
- 字符串值：string。
- 对象：object。
- 函数：function。

下面的代码演示了 typeof 运算符的作用：

程序清单：codes\04\4.5\typeof.html

```
<script>
    var a = 5;
    var b = true;
    var str = "hello javascript";
    alert(typeof(a) + "\n" + typeof(b) + "\n" + typeof(str));
</script>
```

上面的代码使用 typeof 运算符分别判断三个变量的数据类型，代码执行的结果如图 4.26 所示。

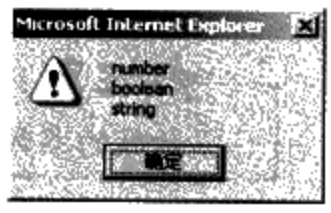


图 4.26 typeof 运算符

与 typeof 类似的运算符还有 instanceof，该运算符用于判断某个变量是否为指定类的实例，如果是，则返回 true，否则返回 false。例如如下代码：

程序清单：codes\04\4.5\instanceof.html

```
<script>
    //定义一个数组
    var a = [4, 5];
    //判断变量 a 是否为 Array 的实例
    alert(a instanceof Array);
    //判断变量 a 是否为 Object 的实例
    alert(a instanceof Object);
</script>
```

JavaScript 中所有类都是 Object 的子类，变量 a 是一个数组，因此运行上面的程序将弹出两个警告提示框，提示框都提示 true。

4.6 语句

语句是 JavaScript 的基本执行单位。JavaScript 要求所有的语句都以分号（;）结束。基本的语句可

以是简单的赋值语句，可以是算术运算语句，也可以是逻辑运算语句等。除此之外，还有一些特殊的语句，下面具体介绍这些特殊的语句：

➤➤4.6.1 语句块

所谓语句块就是使用花括号包含的多条语句，语句块是个整体的执行体，类似于一条单独的语句。下面是语句块的示例：

```
{
    x = Math.PI;
    cx = Math.cos(x);
    alert("Hello JavaScript");
}
```

虽然语句块类似于一条单独的语句，但语句块后不需要以分号结束。但语句块中的每条语句都需要以分号结束。

注意：

虽然 JavaScript 支持使用语句块，但 JavaScript 的语句块不能作为变量的作用域。

➤➤4.6.2 空语句

最简单的空语句仅有一个分号（;），如下代码是一条空语句：

```
//空语句
;
```

上面的空语句没有丝毫用处，因此实际中几乎不会使用这种空语句。空语句主要用于没有循环体的循环。看如下代码：

程序清单：codes\04\4.6\emptyStatement.html

```
<script>
    //声明一个数组
    var a = [];
    //使用空语句，完成数组的初始化
    for (var i = 0 ; i < 10 ; a[i++] = 20);
    //遍历数组元素
    for ( index in a)
    {
        document.writeln(a[index] + "<br>");
    }
</script>
```

上面的粗体字代码使用空语句完成数组的初始化，这种初始化更加简洁。

➤➤4.6.3 异常抛出语句

JavaScript 支持异常处理，支持手动抛出异常。与 Java 不同的是，JavaScript 的异常没有 Java 那么丰富，JavaScript 的所有异常都是 Error 对象。当 JavaScript 需要抛出异常时，总是通过 throw 语句抛出 Error 对象。抛出 Error 的语法如下：

```
throw new Error(errorString);
```

可以在代码执行过程中抛出异常，也可以在函数定义中抛出异常。代码执行过程中，一旦遇到异常，立即寻找对应的异常捕捉块（catch 块），如果没有对应的异常捕捉块，异常将传播给浏览器，程序非正常中止。看如下代码：

程序清单：codes\04\4.6\throw.html

```
<script>
    //对计数器 i 循环
    for (var i = 0 ; i < 10 ; i++)
    {
        //在页面输出 i
        document.writeln(i + '<br>');
        //当 i > 4 时，抛出用户自定义异常
        if (i > 4)
            throw new Error('用户自定义错误');
    }
</script>
```

代码的执行效果如图 4.27 所示。

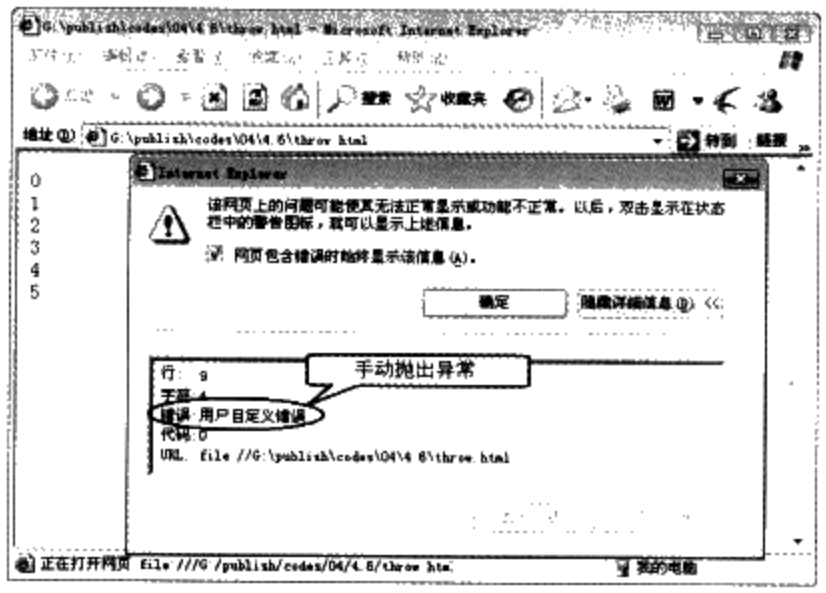


图 4.27　手动抛出异常的效果

如图 4.27 所示，当 i = 5 时，手动抛出异常，该异常没有得到处理，因而传播到浏览器，引起程序的非正常中止，浏览器也有关于错误的提示。

➤➤4.6.4　异常捕捉语句

当程序出现异常时，不管是用户手动抛出的异常，还是系统本身的异常，都可使用 catch 捕捉。JavaScript 代码运行中一旦出现异常，程序就跳转到对应的 catch 块。异常捕捉的语法格式如下：

```
try
{
    statements
}
catch(e)
{
    statements
}
finally
{
    statements
}
```

这种异常捕捉语句大致上类似于 Java 的异常捕捉语句，但有一些差别：因为 JavaScript 的异常体系远不如 Java 丰富，因此无须使用多个 catch 块。与 Java 异常机制类似的是 finally 块是可以省略的，但一旦指定了 finally 块，则 finally 代码块总会获得执行的机会。看如下代码：

程序清单：codes\04\4.6\throwCatch.html

```
<script>
try
{
    for (var i = 0 ; i < 10 ; i++)
    {
        //在页面输出 i 值
        document.writeln(i + '<br>');
        //当 i 大于 4 时，抛出异常
        if (i > 4)
            throw new Error('用户自定义错误');
    }
}
//如果 try 块中的代码出现异常，自动跳转到 catch 块执行
catch (e)
{
    document.writeln('系统出现异常' + e.message + '<br>');
}
//finally 块的代码总可以获得执行的机会
finally
{
    document.writeln('系统的 finally 块');
}
</script>
```

从上面的粗体字代码可以看出，JavaScript 同样可以获取异常的描述信息，通过异常对象的 message 属性即可访问异常对象的描述信息。

代码运行的结果如图 4.28 所示。

图 4.28 使用 try catch 块捕捉异常

归纳起来，JavaScript 异常机制与 Java 异常机制存在如下区别：

- JavaScript 中只有一个异常类 Error，无须在定义函数时声明抛出该异常，所以没有 throws 关键字。
- JavaScript 是弱类型语言，所以 catch 语句后括号里的异常实例无须声明类型。
- JavaScript 只有一个异常类，所以 try 块后最多只能有一个 catch 块。
- 获取异常的描述信息是通过异常对象的 message 属性，而不是通过 getMessage()方法。

4.6.5 with 语句

with 语句是一种更简洁的写法，使用 with 语句可以避免重复书写对象。with 语句的语法格式如下：

```
with(object)
{
    statements
}
```

如果 with 后的代码块只有一行语句，则可以省略花括号。但只有一行代码的情形下，使用 with 语句意义不大。关于 with 语句的作用，看如下代码：

```
document.writeln("Hello<br>");
document.writeln("World<br>");
document.writeln("JavaScript<br>");
```

上面的代码中，多次使用 document 的 writeln 方法重复输出静态字符串。使用 with 语句可以避免重复书写 document 对象。将上面代码该为如下形式，效果完全相同。

```
with(document)
```

```
{
    writeln("Hello<br>");
    writeln("World<br>");
    writeln("JavaScript<br>");
}
```

4.7 流程控制

JavaScript 支持的流程控制也很丰富。JavaScript 支持基本的分支语句，如 if、if...else 等，也支持基本的循环语句，如 while、for 等，还支持 foreach 循环等。此外，循环相关的 break、continue 也是支持的，而且也支持使用带标签的 break、continue 语句。

➤➤4.7.1 分支

分支语句主要有 if 语句和 switch 语句。其中 if 语句有如下三种形式：

第一种形式：

```
if ( logic expression )
{
    statements...
}
```

第二种形式：

```
if (logic expression)
{
    statements...
}
else
{
    statements...
}
```

第三种形式：

```
if (logic expression)
{
    statements...
}
else if(logic expression)
{
    statements...
}
...//可以有多条 else if 语句
else//最后的 else 语句也可以省略
{
    statement...
}
```

通常，不要省略 if、else、else if 后执行块的花括号，但如果语句执行块只有一行语句，则可以省略花括号，例如如下代码：

程序清单：codes\04\4.7\if.html

```
<script>
    //定义变量 a ，并为其赋值
    var a = 5;
    //如果 a>4，执行下面的执行体
    if (a > 4)
        alert('a 大于 4');
```

```
    //否则，执行下面的执行体
    else
        alert('a 不大于 4');
</script>
```

上面的代码完全可以正常执行。但如果代码变成如下形式，则不可正常执行：

程序清单：codes\04\4.7\errIf.html

```
<script>
    //定义变量 a ，并为其赋值
    var a = 5;
    //如果 a>4，执行下面的执行体
    if (a > 4)
        alert('a 大于 4');
    //否则，执行下面的执行体
    else
        a--;
        alert('a 不大于 4');
</script>
```

上面的代码中的 alert('a 不大于 4')将总是会执行，即总会弹出“a 不大于 4”的对话框，因为这行代码并不在 else 的语句块内。如果要达到期望的效果，代码应该修改为：

```
<script>
    //定义变量 a ，并为其赋值
    var a = 5;
    //如果 a>4，执行下面的执行体
    if (a > 4)
        alert('a 大于 4');
    //否则，执行下面的执行体
    else
    {
        a--;
        alert('a 不大于 4');
    }
</script>
```

switch 语句的语法格式如下：

```
switch (expression)
{
    case condition 1: statement(s)
        break;
    case condition 2: statement(s)
        break;
    ...
    case condition n: statement(s)
        break;
        default: statement(s)
}
```

这种分支语句的执行是先对 expression 求值，然后依次匹配 condition1、condition2、condition3 等条件，遇到匹配的条件即执行对应的执行体；如果前面的条件都没有正常匹配，则执行 default 后的执行体。看下面的代码：

程序清单：codes\04\4.7\switch.html

```
<script>
    //声明变量 score，并为其赋值'C'
    var score = 'C';
```

```
        //执行 swicth 分支语句
        switch (score)
        {
            case 'A': document.writeln("优秀.");
                break;
            case 'B': document.writeln("良好.");
                break;
            case 'C': document.writeln("中");
                break;
            case 'D': document.writeln("及格");
                break;
            case 'F': document.writeln("不及格");
                break;
            default: document.writeln("成绩输入错误");
        }
    </script>
```

上面的代码是最基本的 switch 语句的用法，输出结果与期望结果相同。代码将在页面上输出“中”。

与 Java 的 switch 语句完全类似，JavaScript 的 switch 语句中也可省略 case 块后的 break 语句。如果省略了 case 块后的 break 语句，JavaScript 将直接执行后面 case 块里的代码，不会理会 case 块里的条件，直到遇到 break 语句为止。

与 Java 中 switch 语句不同的是，switch 语句里的条件变量（就是 switch 后括号里的变量）的数据类型不仅可以是数值类型，也可以是字符串类型，如上面程序所示。

流程控制除基本的分支语句外，还有循环语句，JavaScript 同样提供了丰富的循环语句支持。JavaScript 中的循环语句主要有 while 循环、do while 循环、for 循环、for in 循环。大部分时候，for 循环可以完全代替 while 循环和 do while 循环。

➤➤4.7.2 while 循环

while 循环的语法格式如下：

```
while(expression)
{
    statements;
}
```

当 statements 循环体只有一行语句，即该循环体只有一行代码时，循环体的花括号可以省略。while 循环的作用是：先判断 expression 逻辑表达式的值，当 expression 为真时，执行循环体；如果 expression 为假，则结束循环。看如下代码：

程序清单：codes\04\4.7\while.html

```
<script>
    var count = 0;
    //只要 count < 10，程序将一直执行循环体
    while (count < 10)
    {
        document.write(count + "<br />");
        count++;
    }
    document.write("循环结束!");
</script>
```

这是个标准的 while 循环。对于 while 循环，值得注意的是，一定要让 expression 有为假的时候，否则循环将成为死循环，而永远无法结束。下面的代码演示了一个死循环：

程序清单：codes\04\4.7\deadLoop.html

```
<script>
    var count = 0;
    //因为 count < 10 一直为 true，所以该循环是死循环
    while (count < 10)
    {
        document.write("不停执行的死循环 " + count + "<br />");
        count--;
    }
    document.write("永远无法执行到的循环体");
</script>
```

注意：

while 循环必须包含循环条件，也就是 while 后的括号里必须有一个逻辑表达式。

4.7.3 do while 循环

do while 循环与 while 循环的区别在于：while 循环是先判断循环条件，如果条件为真才执行循环体；而 do while 循环则先执行循环体，然后判断循环条件，如果循环条件为真，则执行下一次循环，否则中止循环。do while 循环的语法格式如下：

```
do
{
    statements;
}
while (expression);
```

与 while 循环类似的是：如果循环体只有一行语句，则循环体的花括号可以省略。即如下的代码是完全正确的：

程序清单：codes\04\4.7\doWhile.html

```
<script>
    //定义变量 count
    var count = 0;
    //执行 do while 循环
    do
        document.write(count++ +"<br />");
    //当 count < 10 时执行下一次循环
    while (count < 10);
    document.write("循环结束!");
</script>
```

与 while 循环的区别在于：while 循环的循环体可能一直得不到执行，而 do while 的循环体至少可以执行一次。

4.7.4 for 循环

for 循环是更加简洁的循环语句。大部分情况下，for 循环可以代替 while 循环和 do while 循环。for 循环的基本语法格式如下：

```
for (initialization; test condition; iteration statement)
{
    statements
}
```

与前面的循环类似的是，如果循环体只有一行语句，则循环体的花括号可以省略。下面使用 for 循环代替前面的 while 循环，代码如下：

程序清单：codes\04\4.7\for.html

```
<script>
    for (var count = 0 ; count < 10 ; count++)
        document.write(count + "<br />");
    document.write("循环结束!");
</script>
```

对于 for 后面的括号，只有两个分号是必需的，其他都是可以省略的。如下所示代码没有语法错误：

程序清单：codes\04\4.7\deadFor.html

```
<script>
    //下面的 for 循环是死循环
    for ( ; ; )
        document.write('count' + "<br />");
    //下面的语句永远都无法执行到
    document.write("Loop done!");
</script>
```

上面的代码有如图 4.29 所示的执行效果。

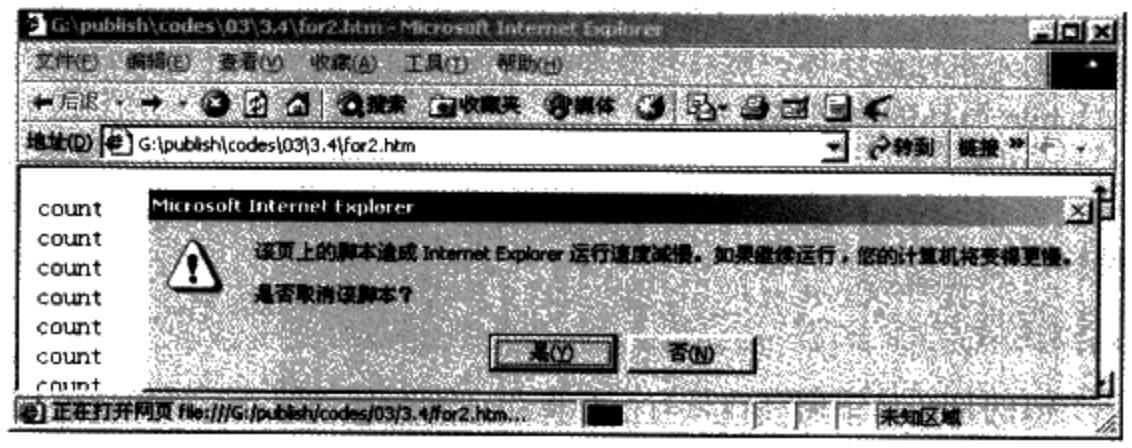


图 4.29　for 循环的执行效果

for 后的括号以两个分号隔开了三条语句，其中第一条语句是循环的初始化语句，每条循环语句只会执行一次，而且完全可以省略，因为初始化语句可以放在循环语句之前完成。第二条语句是个逻辑表达式，用于判断是否执行下一次循环。因此，通常情况下，第二条语句都是不可省略的，如果省略该循环条件，则循环条件一直为 true，也就变成了死循环；第三条语句是循环体执行完后最后执行的语句，这条语句也完全可以放在循环体最后执行。

➤➤4.7.5　for in 循环

for in 循环实质上是一种 foreach 循环，它主要有两个作用：

- 遍历数组里的所有数组元素。
- 遍历 JavaScript 对象的所有属性。

for in 循环的语法格式如下：

```
for (index in object)
{
    statements
}
```

与前面类似的是，如果循环体只有一行代码，则可以省略循环体的花括号。

在遍历数组时，for in 循环的循环计数器是数组元素的索引值。看如下代码：

程序清单：codes\04\4.7\forin1.html

```
<script>
    //定义数组
```

```
    var a = ['hello' , 'javascript' , 'world'];
    //遍历数组的每个元素
    for (str in a)
        document.writeln('索引' + str + '的值是:' + a[str] + "<br>" );
</script>
```

除此之外，for in 循环还可遍历对象的所有属性。此时，循环的计数器是该对象的属性名。看如下代码：

程序清单：codes\04\4.7\forin2.html

```
<script>
    //在页面输出静态文本
    document.write("<h1>Navigator 对象的全部属性如下：</h1>");
    //遍历 navigator 对象的所有属性
    for (propName in navigator)
    {
        //输出 navigator 对象的所有属性名，以及对应的属性值
        document.write('属性' + propName + '的值是：' + navigator[propName]);
        document.write("<br />");
    }
</script>
```

代码执行的结果如图 4.30 所示。

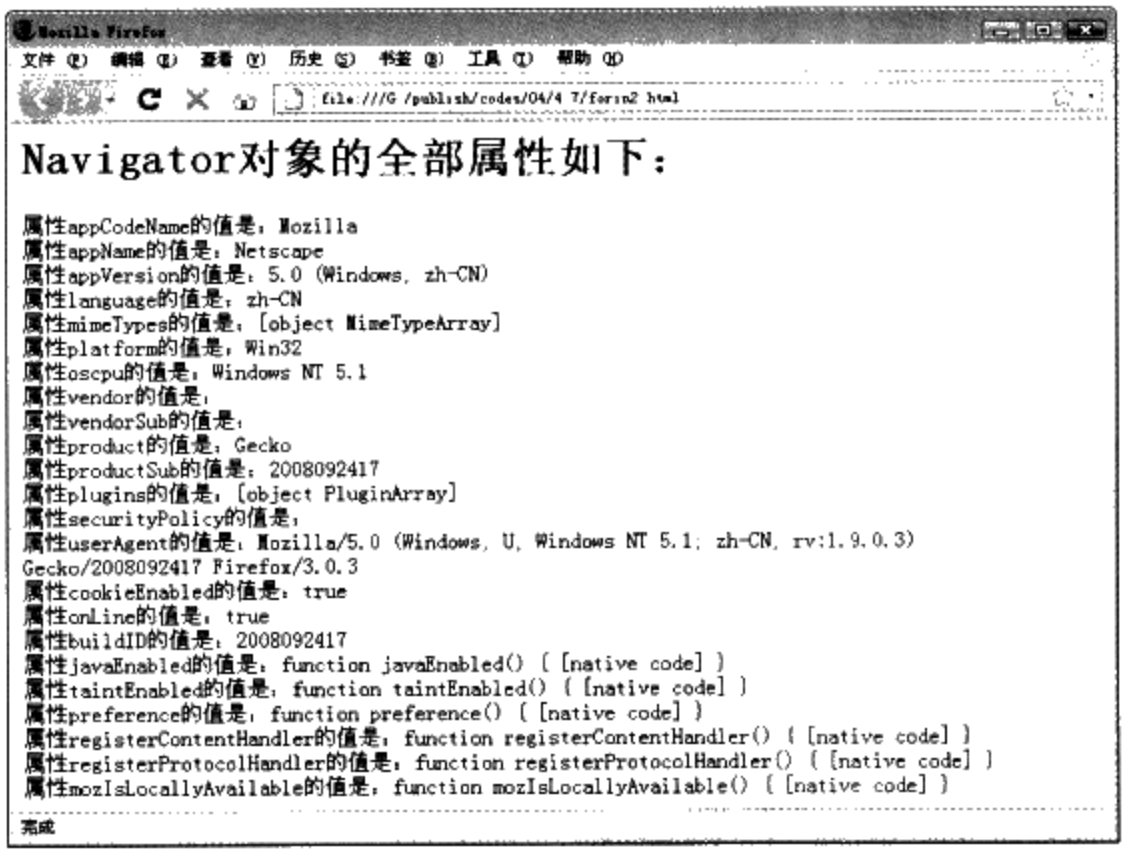


图 4.30　遍历 navigator 对象的全部属性

注意：

navigator 是 JavaScript 的内建对象。关于 navigator 对象的介绍，参见本书 6.8.2 节内容。

➤➤4.7.6　break 和 continue

break 和 continue 都可用于中止循环。区别是 continue 只是中止本次循环，接着开始下一次循环（也可以视为忽略本次循环后面的执行语句）；而 break 则是完全中止循环，跳出循环体。看下面的代码：

程序清单：codes\04\4.7\break.html

```
<script>
    //以 i 为计数器循环
```

```
    for (var i = 0 ; i < 5 ; i++)
    {
        //以 j 为计数器循环
        for (var j = 0 ; j < 5 ; j++)
        {
            document.writeln('j 的值为: ' + j);
            //当 i >= 2 时，使用 break 中止循环
            if (i >= 2) break;
            document.writeln('i 值为: ' + i);
            document.writeln('<br />');
        }
    }
</script>
```

因为 break 是中止循环，完全跳出循环体本身。当 i = 2 时，嵌套循环的第一行代码可以执行，然后执行 break，跳出循环体。嵌套循环结束，外部循环计数器再次增加，即 i = 3，依此类推。当 i 等于 2,3,4 时，嵌套循环都只执行一行代码：打印出“j 的值为：0”。代码执行结果如图 4.31 所示。

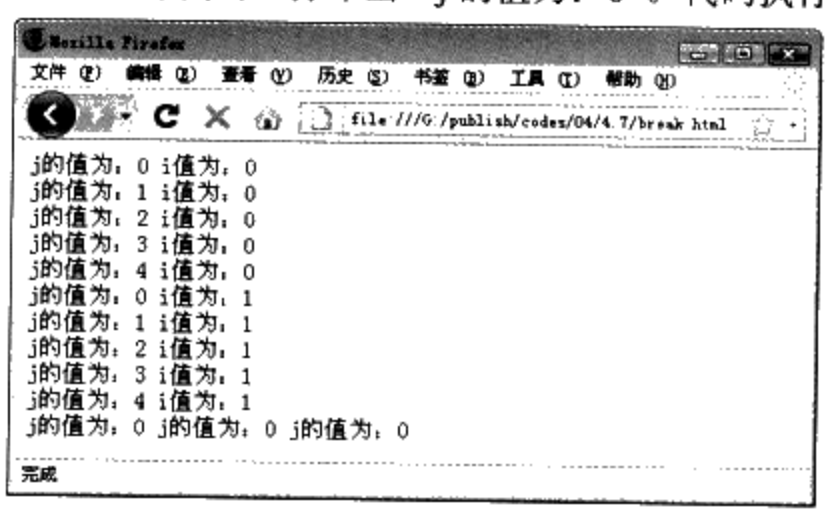


图 4.31 使用 break 中止循环

如果使用 continue 中止循环，则结果完全不同，看如下代码：

程序清单：codes\04\4.7\continue.html

```
<script>
    //以 i 为计数器循环
    for (var i = 0 ; i < 5 ; i++)
    {
        //以 j 为计数器循环
        for (var j = 0 ; j < 5 ; j++)
        {
            document.writeln('j 的值为: ' + j);
            //当 i >= 2 时，使用 continue 中止本次循环
            if (i >= 2) continue;
            document.writeln('i 的值为: ' + i);
            document.writeln('<br>');
        }
    }
</script>
```

因为 continue 仅仅中止本次循环（即略过本次循环后面的语句），并不完全跳出循环体。当 i = 2 时，执行了嵌套循环的第一行代码，即使用 continue 跳出循环：中止本次循环，并不跳出循环体，而是开始第二次循环，还在嵌套循环内运行，此时 i 依然等于 2，j = 1，同样只能执行到第一行代码，再次中止本次循环，开始 j = 2 的循环。这就是使用 break 和 continue 的区别，如图 4.32 所示是使用 continue 中止循环的结果。

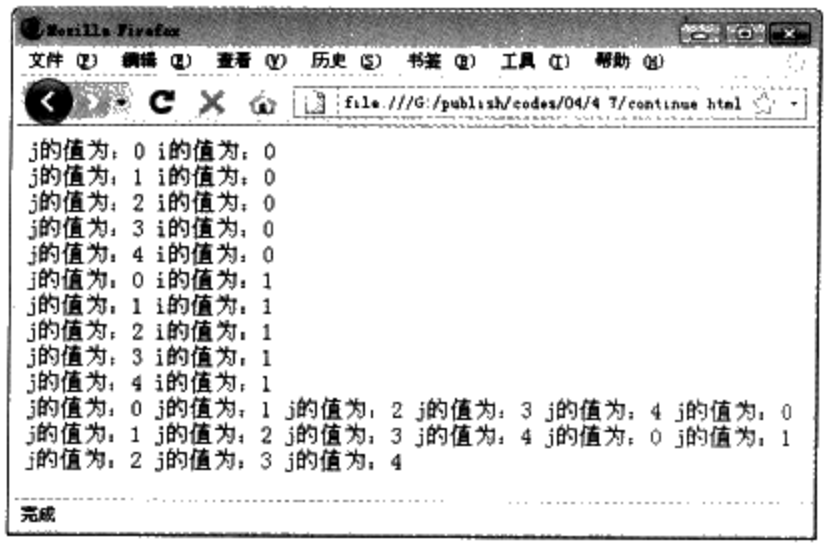


图 4.32　使用 continue 中止本次循环的结果

如果在 break 或 continue 后使用标签，则可以直接跳到标签所在的循环。至于使用 break 和 continue 的区别与前面类似，break 是完全中止标签所在的循环，而 continue 则是中止标签所在的本次循环。看如下代码：

注意：

所谓标签，就是在一个合法的标识符后紧跟一个英文冒号（:）。标签只有放在循环之前才有效，放在其他地方将没有任何意义，而且会引起语法错误。

程序清单：codes\04\4.7\breakLabel.html

```
<script>
    //使用 outer 标签表明外部循环
    outer :
    for (var i = 0 ; i < 5 ; i++)
    {
        for (var j = 0 ; j < 5 ; j++)
        {
            document.writeln('j 的值为：' + j);
            //当 j >= 2 时，使用 break 跳出 outer 循环
            if (j >= 2) break outer;
            document.writeln('i 的值为：' + i);
            document.writeln('<br>');
        }
    }
</script>
```

当 j = 2 时，使用 break 完全中止循环，break 后还有 outer 标签，这将完全中止 outer 标签对应的循环。即 j = 2 时，仅执行了 document.writeln('j 的值为：'+j)代码，然后两层循环即完全结束。

再看如下代码：

程序清单：codes\04\4.7\continueLabel.html

```
<script>
    //使用 outer 标签表明外部循环
    outer :
    for (var i = 0 ; i < 5 ; i++)
    {
        for (var j = 0 ; j < 5 ; j++)
        {
            document.writeln('j 的值为：' + j);
            //当 j >= 2 时，使用 continue 结束 outer 循环
            if (j >= 2) continue outer;
```

```
                document.writeln('i 的值为: ' + i);
                document.writeln('<br>');
            }
        }
    </script>
```

当 j = 2 时，使用 continue 结束本次循环，而且 continue 后紧跟着 outer 标签，即当执行到 j = 2 时，outer 标签所在的本次循环结束，此时不再执行内部的嵌套循环，而是直接开始 i = 3 的循环。当 i=3 时，运行到 continue outer 后再次结束外部循环的本次循环，而直接开始 i = 4 的循环，依此类推。

4.8 函数

JavaScript 是一种基于对象的脚本语言，JavaScript 代码复用的单位是函数，但它的函数比结构化程序设计语言的函数功能更丰富。JavaScript 语言中函数就是一等公民，它可以独立存在，而且 JavaScript 的函数完全可作为一个类来使用（而且它还是该类唯一的构造器），因此函数的功能非常丰富。不管怎样，JavaScript 中函数是相当重要的知识点。

➤➤4.8.1 函数定义

正如前面介绍的，JavaScript 是弱类型的语言。因此，定义函数时既不需要声明函数的返回值类型，也不需要声明函数的参数列表类型。函数定义的语法格式如下：

```
function functionName(parameter-list)
{
    statements
}
```

下面的代码定义了一个简单的函数，并调用了该函数：

程序清单：codes\04\4.8\simpleFunction.html

```
<script>
    hello('yeeku');
    //定义函数 hello，该函数需要一个参数
    function hello(name)
    {
        alert(name + ", 你好");
    }
</script>
```

函数最大的作用是提供代码复用，将需要重复使用的代码块定义成函数，可提供更好的代码复用。

注意：

从上面的程序中可以看出，在同一个<script.../>元素中时，JavaScript 允许先调用函数，然后再定义该函数；但在不同的<script.../>元素中时，必须先定义函数，再调用该函数。也就是说，后面的<script.../>元素中可以调用前面<script.../>里定义的函数。

函数可以有返回值，也可以没有返回值。函数的返回值使用 return 语句返回，函数运行过程中，一旦遇到一条 return 语句，函数即返回返回值，函数结束。看下面的代码：

程序清单：codes\04\4.8\functionReturn.html

```
<script>
    //定义函数 hello
    function hello(name)
    {
        //如果参数类型为字符串，则返回静态字符串
```

```
        if (typeof name == 'string')
        {
            return name + ", 你好";
        }
        //当参数类型不是字符串时，执行此处的返回语句
        return '名字只能为字符串'
    }
    alert(hello('yeeku'));
</script>
```

程序执行结果如图 4.33 所示。

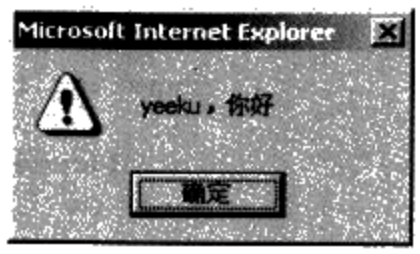


图 4.33　函数返回值

➤➤4.8.2　局部变量和局部函数

前面已经介绍了局部变量的概念，在函数里定义的变量称为局部变量，在函数外定义的变量则称为全局变量，如果局部变量和全局变量的变量名相同，则局部变量会覆盖全局变量。局部变量只能在函数里访问，而全局变量可以在所有的函数里访问。

与此类似的概念是局部函数，局部变量在函数里定义，而局部函数也在函数里定义。下面的代码在函数 outer 中定义了两个局部函数：

程序清单：codes\04\4.8\localFunction.html

```
<script>
    //定义全局函数
    function outer()
    {
        //定义第一个局部函数
        function inner1()
        {
            document.write("局部函数 11111<br />");
        }
        //定义第二个局部函数
        function inner2()
        {
            document.write("局部函数 22222<br />");
        }
        document.write("开始测试局部函数...<br />");
        //在浏览器中调用第一个局部函数
        inner1();
        //在浏览器中调用第二个局部函数
        inner2();
        document.write("结束测试局部函数...<br />");
    }
    document.write("调用 outer 之前...<br />");
    //调用全局函数
    outer();
    document.write("调用 outer 之后...<br />");
</script>
```

在上面的代码中定义了两个局部函数 inner1 和 inner2，并在 outer 函数内调用了这两个局部函数。这两个函数是在 outer 内定义的，因此可以在 outer 内访问它们。在 outer 以外，则无法访问它们——也就是说，inner1 和 inner2 两个函数仅在 outer 函数内有效。

还有一点需要指出：在外部函数里调用局部函数并不能让局部函数获得执行的机会，只有当外部函数被调用时，外部函数里调用的局部函数才会被执行。

如果将上面的程序稍加修改，在 outer 外增加对 inner 局部函数的调用：

程序清单：codes\04\4.8\localError.html

```
<script>
    //定义全局函数
    function outer()
    {
        //定义第一个局部函数
        function inner1()
        {
            document.write("局部函数 11111<br />");
        }
        //定义第二个局部函数
        function inner2()
        {
            document.write("局部函数 22222<br />");
        }
        document.write("开始测试局部函数...<br />");
        //在浏览器中调用第一个局部函数
        inner1();
        //在浏览器中调用第二个局部函数
        inner2();
        document.write("结束测试局部函数...<br />");
    }
    document.write("调用 outer 之前...<br />");
    //调用全局函数
    outer();
    //在外部函数之外的地方调用局部函数
    inner1();
    document.write("调用 outer 之后...<br />");
</script>
```

则在浏览器中运行该程序时，将出现如图 4.34 所示的结果。

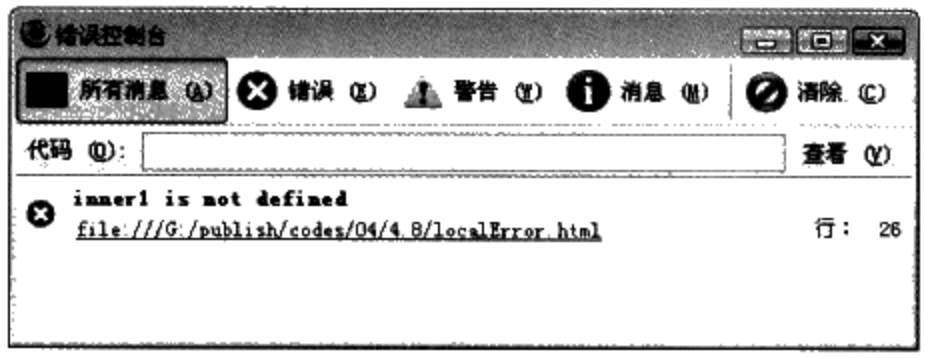


图 4.34　试图访问局部函数的结果

➤➤4.8.3　匿名函数

JavaScript 还提供了 Function 类，该类也可以用于定义函数。Function 的构造器的参数个数可以不受限制。Function 可以接受一系列的字符串参数，其中最后一个参数是函数的执行体，执行体的各语句以分号（;）隔开，而前面的各字符串参数则是函数的参数。看下面定义函数的方式：

程序清单：codes\04\4.8\newFunction.html

```
<script>
    //定义匿名函数，并将函数赋给变量 f
    var f = new Function('name' ,
        "document.writeln('Function 定义的函数<br>');"
        + "document.writeln('你好' + name);");
    //通过变量调用匿名函数
    f('yeeku');
</script>
```

上面的代码使用 new Function()语法定义了一个匿名函数，并将该匿名函数赋给变量 f。从而允许通过 f 来访问匿名函数。还有一种定义匿名函数的方式，可以无须使用 Function 类，而是直接使用 function 关键字。看如下代码：

程序清单：codes\04\4.8\annoymousFunction.html

```
<script>
    var f = function(name)
    {
        document.writeln('匿名函数<br>');
        document.writeln('你好' + name);
    };
    f('yeeku');
</script>
```

上面的代码使用更简单的方式创建了一个匿名函数。注意这种匿名函数的创建语法：

```
function(parameter list)
{
    statements
};
```

对于上面的函数创建语法，可以无须使用函数名，而是将参数列表紧跟 function 关键字。在匿名函数定义的语法最后不要忘记紧跟分号（;）。

实际上，在我们采用普通方式定义一个函数时，也可将该定义赋值给另一个变量，例如如下代码：

程序清单：codes\04\4.8\functionTest.html

```
<script>
    var f = function abc(name)
    {
        document.writeln('匿名函数<br>');
        document.writeln('你好' + name);
    };
    f('yeeku');
    abc('yeeku');
</script>
```

上面的代码中粗体字代码就是一个普通的函数定义，在我们直接将这个函数定义赋值给另一个变量时，粗体字代码定义的 abc()函数将会失去作用，在 Firefox 中调用执行上面的程序，可以在错误控制台中看到如图 4.35 所示的结果。

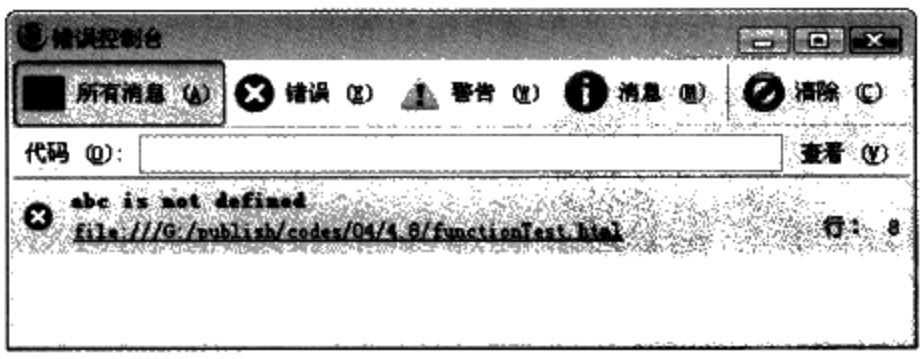


图 4.35　abc 函数失去作用

对于上面的代码，不仅在 Firefox 中 abc 函数会失去意义，在 Opera 等浏览器中也一样，但在 Internet Explorer 浏览器中则会得到与众不同的结果。使用 Internet Explorer 浏览器执行上面的程序，将可以看到如图 4.36 所示结果。

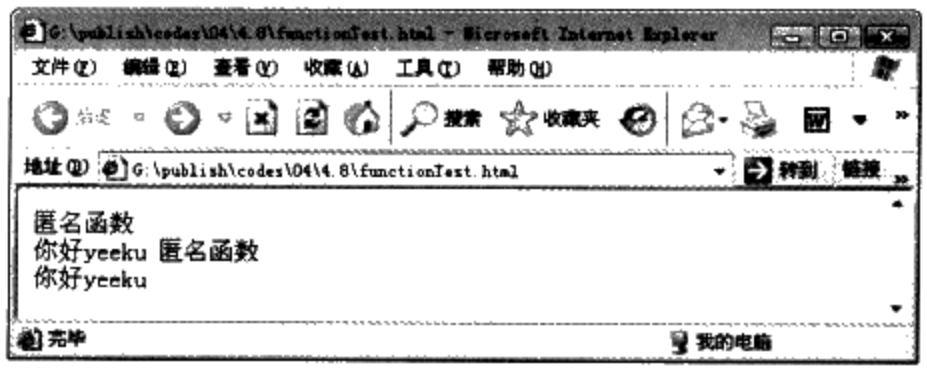


图 4.36　在 Internet Explorer 中 abc 依然存在

从图 4.36 可以看出，对于 functionTest.html 代码中这种直接将普通函数赋给另一个普通变量的情形，在 Internet Explorer 中将会得到两个函数。

➤➤4.8.4　函数和类

前面已经提过，JavaScript 的函数不仅是一个函数，更是一个类。在我们定义一个 JavaScript 函数的同时，也得到了一个与该函数同名的类，该函数也是该类唯一的构造器。

因此，在定义一个函数后，有如下两种调用方式：

- 直接调用函数：直接调用函数总是返回该函数体内最后一句 return 语句的返回值；如果该函数体内不包含 return 语句，则直接调用函数没有任何返回值。
- 使用关键字 new 调用函数：这种方式调用总有一个返回值，返回值就是一个 JavaScript 对象。

看如下代码：

程序清单：codes\04\4.8\functionClass.html

```
<script>
    //定义一个函数
    function test(name)
    {
        return "你好，" + name ;
    }
    //直接调用函数
    var rval = test('leegang');
    //将函数作为类的构造器
    var obj = new test('leegang');
    alert(rval + "\n" + obj);
</script>
```

上面的程序中两行粗体字代码示范了两种调用函数的方式，第一种是直接调用该函数，因此得到的返回值是该函数的返回值；第二种是使用关键字 new 来调用该函数，也就是将该函数当成类使用，所以得到一个对象。执行上面的程序可看到如图 4.37 所示结果。

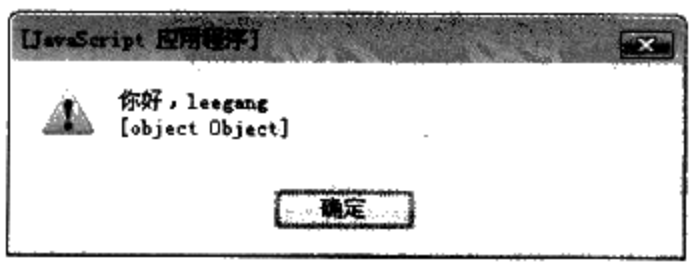


图 4.37　将函数当成类使用

使用匿名函数，允许动态生成和编译函数。使用匿名函数的另一个好处是更加方便，在需要为类、对象定义方法时，使用匿名函数的语法更加简洁。

下面的程序定义了一个 Person 函数，也就是定义了一个 Person 类，该 Person 函数也会作为 Person 类唯一的构造器。定义 Person 函数时希望为该函数定义一个方法，程序如下：

程序清单：codes\04\4.8\functionClass.html

```
<script>
    //定义一个函数，该函数也是一个类
    function Person(name , age)
    {
        //将参数 name 的值赋给 name 属性
        this.name = name;
        //将参数 age 的值赋给 age 属性
        this.age = age;
        //为函数分配 info 方法，使用匿名函数来定义方法
        this.info = function()
        {
            document.writeln("我的名字是：" + this.name + "<br />");
            document.writeln("我的年纪是：" + this.age + "<br />");
        };
    }
    //创建对象 p
    var p = new Person('yeeku' , 29);
    //执行 info 方法
    p.info();
</script>
```

上面的代码为 Person 类分配了一个方法，通过使用匿名函数，代码更加简洁，程序的执行结果如图 4.38 所示。

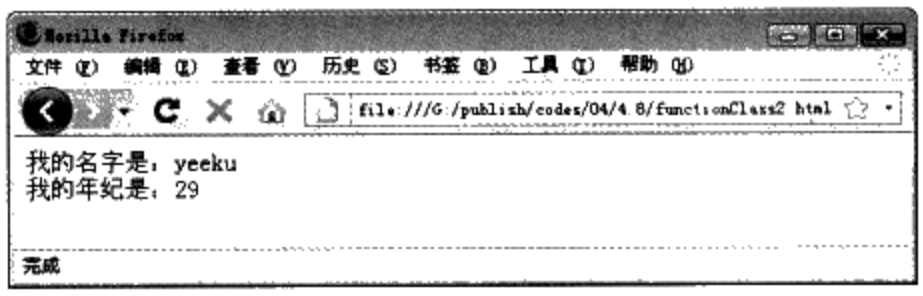


图 4.38 使用匿名函数为对象分配方法

上面的程序中使用了关键字 this，被关键字 this 修饰的变量不再是局部变量，而是该函数的实例属性。关于函数的实例属性和静态属性参看下一节中的介绍。

➤➤4.8.5 函数的实例属性和静态属性

JavaScript 函数不仅仅是一个函数，而且还是一个类，而且该函数还是此类唯一的构造器。只要在调用函数时使用 new 关键字，就可返回一个 Object，这个 Object 不是函数的返回值，而是函数本身产生的对象。

因此在 JavaScript 函数中定义的变量不仅有局部变量，还有实例属性和静态属性两种。根据函数中声明变量的方式，函数中的变量分为：

- 局部变量：在函数中以普通方式声明的变量，包括以 var 或不加任何前缀声明的变量。
- 实例属性：在函数中以 this 前缀修饰的变量。
- 静态属性：在函数中以函数名前缀修饰的变量。

前面已经对局部变量作了介绍，局部变量是只有在函数里才可以访问的变量。实例属性和静态属性则是面向对象的概念：实例属性是属于单个对象的，因此必须通过对象来访问；静态属性是属于整个类（也就是函数）的，因此必须通过类（也就是函数）来访问。

同一个类（也就是函数）只有一块静态属性内存，其每创建一个对象，系统都将会为该对象的实例属性分配一块内存。看如下代码：

程序清单：codes\04\4.8\instanceProperty.html

```
<script>
    //定义函数 Person
    function Person(national, age)
    {
        //this 修饰的属性为实例属性
        this.age = age;
        //Person 修饰的属性为静态属性
        Person.national =national;
        //以 var 定义的变量为局部变量
        var bb = 0;
    }
    //创建 Person 的第一个对象 p1。国籍为中国，年纪为 29
    var p1 = new Person('中国' , 29);
    document.writeln("创建第一个 Person 对象<br />");
    //输出第一个对象 p1 的年纪和国籍
    document.writeln("p1 的 age 属性为" + p1.age + "<br />");
    document.writeln("p1 的 national 属性为" + p1.national + "<br />");
    document.writeln("通过 Person 访问静态 national 属性为"
        + Person.national + "<br />");
    //输出 bb 属性
    document.writeln("p1 的 bb 属性为" + p1.bb + "<br /><hr />");
    //创建 Person 的第二个对象 p2
    var p2 = new Person('美国' , 32);
    document.writeln("创建两个 Person 对象之后<br />");
    //再次输出 p1 的年纪和国籍
    document.writeln("p1 的 age 属性为" + p1.age + "<br />");
    document.writeln("p1 的 national 属性为" + p1.national + "<br />");
    //输出 p2 的年纪和国籍
    document.writeln("p2 的 age 属性为" + p2.age + "<br>");
    document.writeln("p2 的 national 属性为" + p2.national + "<br />");
    //通过类名访问静态属性名
    document.writeln("通过 Person 访问静态 national 属性为"
        + Person.national + "<br />");
</script>
```

Person 函数的 age 属性为实例属性，因而每个实例的 age 属性都可以完全不同，程序应通过 Person 对象来访问 age 属性；national 属性为静态属性，该属性完全属于 Person 类，因此必须通过 Person 类来访问 national 属性，Person 对象并没有 national 属性，所以通过 Person 对象访问该属性将返回 undefined；而 bb 则是 Person 的局部变量，在 Person 函数以外无法访问该变量。程序执行的结果如图 4.39 所示。

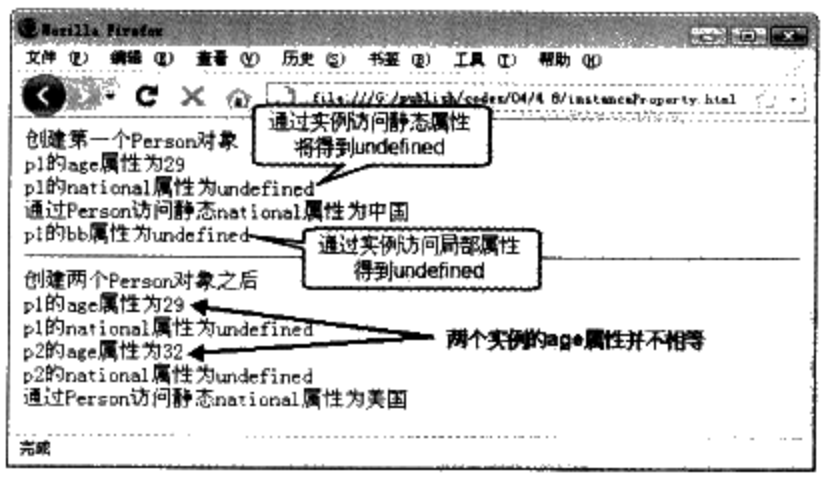


图 4.39　静态属性和实例属性

值得指出的是，JavaScript 与 Java 不一样，它是一种动态语言，允许随时为对象增加属性和方法。在我们直接为对象的某个属性赋值时，即可视为给对象增加属性，例如如下代码：

程序清单：codes\04\4.8\dynaProperty.html

```
<script>
    function Student(grade , subject)
    {
        //定义一个 grade 实例属性，
        //将形参 grade 的值赋值给该实例属性
        this.grade = grade;
        //定义一个 subject 静态属性，
        //将形参 subject 的值赋值给该静态属性
        Student.subject = subject;
    }
    s1 = new Student(5, 'Java');
    with(document)
    {
        writeln('s1 的 grade 属性：' + s1.grade + "<br />");
        writeln('s1 的 subject 属性：' + s1.subject + "<br />");
        writeln('Student 的 subject 属性：' + Student.subject + "<br />");
    }
    //为对象 s1 的 subject 属性赋值，即为它增加一个 subject 属性
    s1.subject = 'Ruby';
    with(document)
    {
        writeln('<hr />为 s1 的 subject 属性赋值后<br />');
        writeln('s1 的 subject 属性：' + s1.subject + "<br />");
        writeln('Student 的 subject 属性：' + Student.subject + "<br />");
    }
</script>
```

上面的程序中粗体字代码为 s1 的 subject 属性赋值，赋值后该 subject 属性值为'Ruby'，但这并不是修改 Student 的 subject 属性，这行代码仅仅是为对象 s1 动态增加了一个 subject 属性。运行上面的程序，可看到如图 4.40 所示结果。

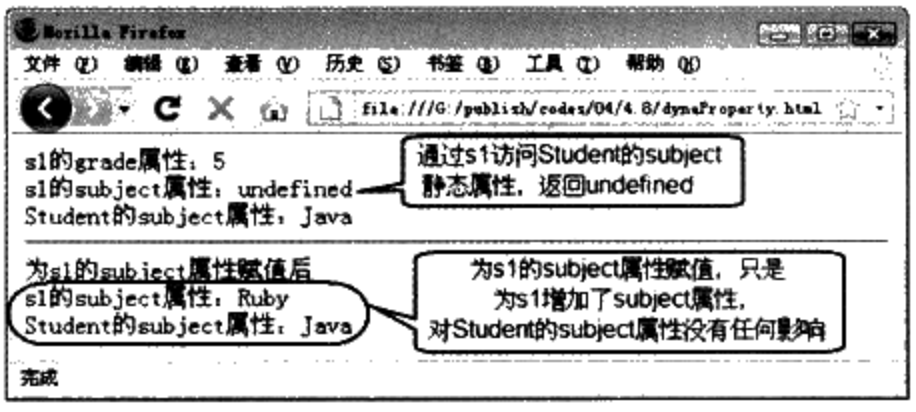


图 4.40　为 JavaScript 对象动态增加实例属性

从图 4.40 中可以看出，在为 s1 的 subject 属性赋值时，Student 的 subject 并不会受任何影响，这表明 JavaScript 对象不能访问它所属类的静态属性。

➤➤4.8.6　递归函数

递归函数是一种特殊的函数，允许在函数定义中调用函数本身。考虑对于如下计算：

```
n! = n * ( n - 1 ) * (n - 2) * ...* 1
```

希望能写一个简单的函数完成对 n!的求值。观察上面的等式发现：

```
(n - 1) ! = (n-1) * (n - 2) *... *1
```

因而有如下等式：

```
n ! = n * (n -1 )!
```

注意到等式左边需要求 n 的阶乘，等式右边则为求(n -1)的阶乘。而实质都是一个函数。因此可将

求阶乘的函数定义如下：

程序清单：codes\04\4.8\factorial.html

```
<script>
    //定义求阶乘的函数
    function factorial(n)
    {
        //如果 n 的类型是数值，才执行函数
        if (typeof(n) == "number")
        {
            //当 n 等于 1 时，直接返回 1
            if (n == 1)
            {
                return 1;
            }
            //当 n 不等于 1 时，通过递归返回值
            else
            {
                return n * factorial(n - 1);
            }
        }
        //当参数不是数值时，直接返回
        else
        {
            alert("参数类型不对！");
        }
    }
    //调用阶乘函数
    alert(factorial(5));
</script>
```

上面的程序中粗体字代码再次调用了 factorial()函数，这就是在函数里调用函数本身，也就是所谓的递归。上面的程序执行的结果是 120，可以正常求出 5 的阶乘。注意到程序中判断参数时，先判断了参数 n 是否为数值，而且要求 n 大于 0 才会继续运算。事实上，这个函数不仅要求 n 为数值，而且必须是大于 0 的整数，否则函数不仅不能得到正确结果，还将产生内存溢出。

对于上面的递归函数，当 n 为一个大于 0 的整数，例如 5 时，5 的阶乘为 4 的阶乘和 5 的乘积，同理，4 的阶乘为 3 的阶乘和 4 的乘积……依此类推，直到最后 1 的阶乘，代码中已经写明：当 n = 1 时，返回值为 1。然后反算回去，所有的值都变成已知的。反过来，当 n 为负数，例如-1 时，-1 的阶乘为-1 与-2 的阶乘的乘积，-2 的阶乘为-2 和-3 的阶乘的乘积……这样将一直追溯到负无穷大，没有尽头，会导致程序溢出。

可见，递归的方向很重要，一定要向已知的方向递归。对于上例而言，因为 1 的阶乘是已知的，因此递归一定要追溯到 1 的阶乘，递归一定要给定中止的条件，这一点与循环类似。没有中止条件的循环是死循环，不向中止点追溯的递归是无穷递归。

注意：

递归一定要向已知点追溯。

4.9 函数的参数处理

大部分的时候，函数都需要接受参数传递。与 Java 完全类似，JavaScript 的参数传递也全部是值传递方式。

注意：

虽然绝大部分书籍、资料，包括本书的第一版也曾经称（本书是《基于 J2EE 的 Ajax 宝典》的第二版）：JavaScript 中复合类型的参数，采用了引用传递的方式。但实际上，笔者不得不指出：JavaScript 的参数传递机制与 Java 的参数传递机制完全相同。

➤➤4.9.1 基本类型和复合类型的参数传递

对于基本类型参数，JavaScript 采用值传递方式。通过实参调用函数时，传入函数里的并不是实参本身，而只是实参的副本，因此在函数中修改参数值并不会对实参构成任何影响。看下面的程序：

程序清单：codes\04\4.9\transfer.html

```
<script>
    //定义一个函数，只有一个参数
    function change(arg1)
    {
        //对参数赋值，对实参不会有任何影响
        arg1 = 10;
        document.write("函数执行中 arg1 的值为：" + arg1 + "<br/>");
    }
    //定义变量 x 的值为 5
    var x = 5;
    //输出函数调用之前 x 的值
    document.write("函数调用之前 x 的值为：" + x + "<br />");
    change(x);
    document.write("函数调用之后 x 的值为：" + x + "<br />");
</script>
```

当使用变量 x 作为参数调用 change()函数时，x 并未真正传入 change()函数中，传入的仅仅是 x 的副本，因此在 change()中对参数赋值不会影响 x 的值。代码执行结果如图 4.41 所示。

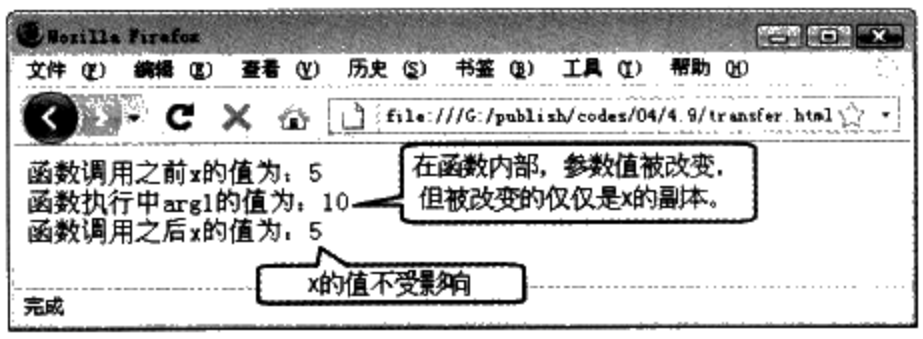


图 4.41　值传递方式的参数传递

从图 4.41 中可看到，函数调用之前，x 的值为 5，函数调用之后，x 的值依然为 5。虽然函数体内修改了 x 的值，但实际上 x 的值根本没有改变。这就是因为 JavaScript 的基本类型的参数传递采用值传递方式，实际传入函数的只是 x 的副本，所以 x 本身是不会有任何改变的。

而对于复合类型的参数，实际上依然是值传递，只是很容易混淆，看如下程序：

程序清单：codes\04\4.9\transfer2.html

```
<script>
    //定义函数，该函数接受一个参数
    function changeAge(person)
    {
        //改变 person 的 age 属性
        person.age = 10;
        //输出 person 的 age 属性
        document.write("函数执行中 person 的 age 值为："
            + person.age + "<br />");
```

```
        //将 person 变量直接赋为 null
        person = null;
    }
    //使用 JSON 语法定义 person 对象
    var person = {age : 5};
    //输出 person 的 age 属性
    document.write("函数调用之前 person 的 age 的值为："
        + person.age + "<br />");
    //调用函数
    changeAge(person);
    //输出函数调用后 person 实例的 age 属性值
    document.write("函数调用之后 person 的 age 的值为："
        + person.age + "<br />");
    document.write("person 对象为：" + person);
</script>
```

上面的代码中使用了 JSON 语法来创建 person 对象，关于 JSON 语法参见 4.11.3 节，在上面的程序中，传入 changeAge()函数中的不再是基本类型变量，而是一个复合类型变量。

执行上面的代码，可以看到如图 4.42 所示的结果。

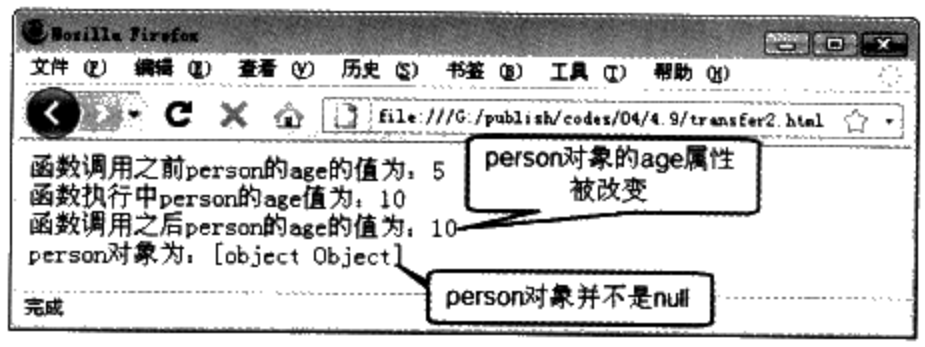


图 4.42 复合类型的参数传递

如果仅从 person 对象的 age 属性被改变来看，很多资料、书籍非常容易得到一个结论：复合类型的参数采用了引用传递方式，而不再是值传递。

但我们看到 changeAge()函数最后一行粗体字代码中，我们将 person 对象直接赋值为 null，但 changeAge()函数执行结束后，后面的 person 对象依然是一个对象，而并不是 null，这表明 person 本身并未传入 changeAge()函数中，传入 changeAge()函数的依然是副本。

上面的程序的关键是，复合类型变量本身并未持有对象本身，复合类型变量只是一个引用（类似于 Java 的引用变量），该引用指向实际的 JavaScript 对象。当一个 person 复合类型变量传入 changeAge()函数时，传入的依然是 person 变量的副本——只是该副本和原 person 对象指向同一个 JavaScript 对象。因此，不管修改副本所指的 JavaScript 对象，还是修改 person 对象所指的 JavaScript 对象，实际上修改的都是同一个对象。JavaScript 的复合类型包括对象、数组等。

➤➤4.9.2 空参数

看如下程序代码：

程序清单：codes\04\4.9\emptyArg.html

```
<script>
    function changeAge(person)
    {
        if (typeof person == 'object')
        {
            //改变参数的 age 属性
            person.age = 10;
            //输出参数的 age 属性
            document.write("函数执行中 person 的 Age 值为："
```

```
                + person.age + "<br />");
        }
        else
        {
            alert("参数类型不符合" + typeof person);
        }
    }
    changeAge();
</script>
```

上面的代码的函数声明中包含了一个参数，但调用函数时并没有传入任何参数。这种形式对于强类型的语言，如 Java 或 C，都是不允许的；但对于 JavaScript 却没有任何语法问题，因为 JavaScript 会将没有传入实参的参数值自动设置为 undefined 值。如图 4.43 所示是上面的程序执行的结果。

使用空参数完全没有任何程序问题，程序可以正常执行，只是没有传入实参的参数值将作为 undefined 处理。

图 4.43　使用空参数

➤➤4.9.3　参数类型

JavaScript 函数声明的参数列表无须声明类型，这是其作为弱类型语言的一个特征。但 JavaScript 语言又是基于对象的编程语言，这一点往往非常矛盾，例如对于如下的 Java 方法定义：

```
public void changeAge(Person p)
{
    p.setAge(34);
}
```

这个程序没有任何问题，因为 Java 要求参数列表具有类型声明，因而参数 p 属于 Person 实例，而 Person 实例具有 setAge()方法。如果 Person 类没有 setAge()方法，程序将在编译时出现错误。调用该方法时，如果没有传入参数，或者传入参数的类型不是 Person 对象，都将在编译时出现错误。

将上面的程序简单转换成 JavaScript 的写法，即变成如下形式：

```
function changeAge(p)
{
    p.setAge(34);
}
```

值得注意的是，JavaScript 无须类型声明。因此，调用函数时传入的 p 完全可以是整型变量或者布尔型变量，这些类型的数据都没有 setAge()方法，但程序强制调用该方法，因而肯定会导致程序出现错误，非正常中止。

JavaScript 函数定义的参数列表无须类型声明，这一点为函数的调用埋下了隐患，这也是 JavaScript 语言不如 Java、C 语言程序健壮的一个重要原因。

提 示：

实际上这个问题并不是 JavaScript 所独有的，而是所有弱类型语言所共有的问题，由于声明函数时形参无须定义数据类型，所以导致调用这些函数时可能出现问题。

为了解决弱类型语言所存在的问题，弱类型语言方面的专家提出了“鸭子类型（Duck Type）”的概念，他们认为：当你需要一个“鸭子类型”的参数时，由于编程语言本身是弱类型的，所以无法保证传入的参数一定是“鸭子类型”，这时你可以先判断这个对象是否能发出“嘎嘎”声，并具有走路左右摇摆的特征，也就是具有“鸭子类型”的特征——一旦该参数具有“鸭子类型”的特征，即使它不是“鸭子”，程序也可以将它当成“鸭子”使用。

简单地说，“鸭子类型”的理论认为：如果弱类型语言的函数需要接受参数，则应先判断参数类型，

并判断参数是否包含了需要访问的属性、方法。只有当这些条件都满足时，程序才开始真正处理调用参数的属性、方法。看如下代码：

程序清单：codes\04\4.9\duckType.html

```
<script>
    //定义函数 changeAge，函数需要一个参数
    function changeAge(person)
    {
        //首先要求 person 必须是对象，而且 person 的 age 属性为 number
        if (typeof person == 'object'
            && typeof person.age == 'number')
        {
                //执行函数所需的逻辑操作
            document.write("函数执行前 person 的 Age 值为："
                + person.age + "<br/>");
            person.age = 10;
            document.write("函数执行中 person 的 Age 值为："
                + person.age + "<br/>");
        }
        //否则将输出提示，参数类型不符合
        else
        {
            document.writeln("参数类型不符合" +
                typeof person + "<br>");
        }
    }
    //分别采用不同方式调用函数
    changeAge();
    changeAge('xxx');
    changeAge(true);
    //采用 JSON 语法创建第一个对象
    p = {abc : 34};
    changeAge(p);
    //采用 JSON 语法创建第二个对象
    person = {age : 25};
    changeAge(person);
</script>
```

这种语法要求，函数对参数执行逻辑操作之前，首先判断参数的数据类型，并检查参数的属性是否符合要求，在所有的要求满足后才执行逻辑操作，否则弹出警告。如图 4.44 所示为代码的执行结果。

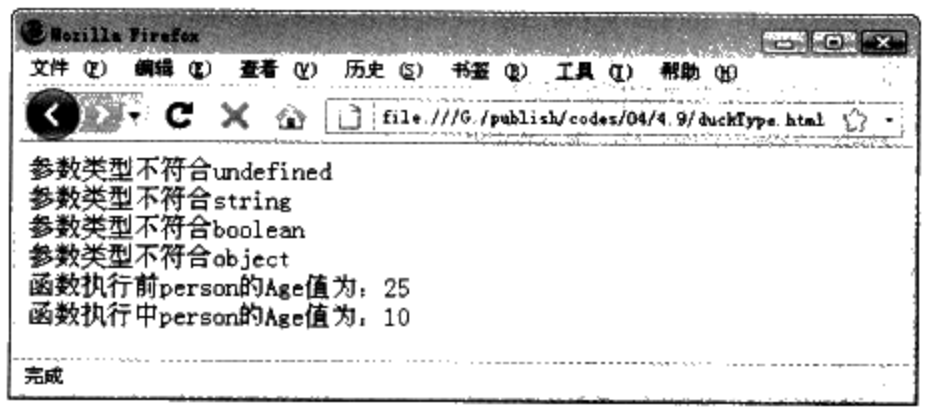


图 4.44　严格的参数检查

4.10　对象

JavaScript 并不严格地要求使用对象，甚至可以不使用函数，而将代码堆积成简单的顺序代码流。但随着代码的增加，为了提供更好的软件复用，建议使用对象和函数。

➤➤4.10.1 面向对象的概念

JavaScript 并不是面向对象的程序设计语言，面向对象设计的基本特征如继承、多态等没有得到很好的实现。纯粹的面向对象语言里，最基本的程序单位是类，类与类之间提供严格的继承关系。如在 Java 中，所有的类都可以通过 extends 显式继承父类，或者默认继承系统的 Object 类。而 JavaScript 并没有提供规范的语法让开发者定义类。

纯粹的面向对象程序设计语言里，严格使用关键字 new 创建对象，而关键字 new 调用该类的构造器，通过这种方式可以返回该类的实例。例如在 Java 中可以通过如下代码创建 Person 实例：

```
Person p = new Person();
```

假设 Person 类已有了 Person 的构造器，通过构造器即可返回 Person 实例。但 JavaScript 则没有这样严格的语法，JavaScript 中的每个函数都可用于创建对象，返回的对象既是函数类的实例，也是 Object 类的实例。看如下代码：

程序清单：codes\04\4.10\objectTest.html

```
<script>
    //定义简单函数
    function Person(name)
    {
        this.name = name;
    }
    //使用关键字 new，简单创建 Person 类的实例
    var p = new Person('yeeku');
    //如果 p 是 Person 实例，则输出静态文本
    if (p instanceof Person)
        document.writeln("p 是 Person 的实例<br />");
    //如果 p 是 Object 实例，则输出静态文本
    if(p instanceof Object)
        document.writeln("p 是 Object 的实例");
</script>
```

上面的 JavaScript 并没有定义 Person 类，而是定义了一个 Person 函数。但 JavaScript 定义 Person 函数的同时，也就得到了一个 Person 类。JavaScript 的函数不支持继承语法，因而 JavaScript 没有完善的继承机制。因此我们习惯上称 JavaScript 是基于对象的脚本语言。

JavaScript 不允许开发者指定类与类之间的继承关系，且并没有提供完善的继承语法，因此开发者定义的类没有父子关系，但这些类都是 Object 类的子类。

运行程序，将可看到变量 p 是 Person 的实例，也是 Object 的实例。

➤➤4.10.2 对象和关联数组

JavaScript 中的对象与纯粹的面向对象语言中的对象存在一定的区别：JavaScript 中的对象本质上是一个关联数组，或者说更像 Java 里的 Map 数据结构，由一组 key-value 对组成。与 Java 中 Map 对象存在的区别是，JavaScript 对象的 value 不仅可以是值（包括基本类型的值和复合类型的值），也可以是函数，此时的函数就是该对象的方法。当 value 为值（包括基本类型的值和复合类型的值）时，此时的 value 将是该对象的属性值。

因此，当需要访问某个 JavaScript 对象的属性时，不仅可以使用 obj.propName 的形式，也可采用 obj[propName]的形式，有些时候甚至必须使用这种形式。例如下面的代码：

程序清单：codes\04\4.10\objectTest2.html

```
<script>
    function Person(name , age)
    {
```

```
        //将形参 name、age 的值分别赋给实例属性 name、age
        this.name = name;
        this.age = age;
        this.info = function()
        {
            alert('info method!');
        }
    }
    var p = new Person('yeeku' , 30);
    for (propName in p)
    {
        //遍历 Person 对象的属性
        document.writeln('p 对象的' + propName
            + "属性值为：" + p[propName] + "<br />");
    }
</script>
```

上面的程序中粗体字代码遍历了 Person 对象的每个属性，因为遍历每个属性时循环计数器是 Person 对象的属性名，因此程序必须根据属性名来访问 Person 对象的属性，此时不能采用 p.propName 的方式，如果采用 p.propName 方式，则 JavaScript 不会把 propName 当成变量处理，而是试图直接访问该对象的 propName 属性——但该属性实际上并不存在。在浏览器中浏览该页面可看到如图 4.45 所示结果。

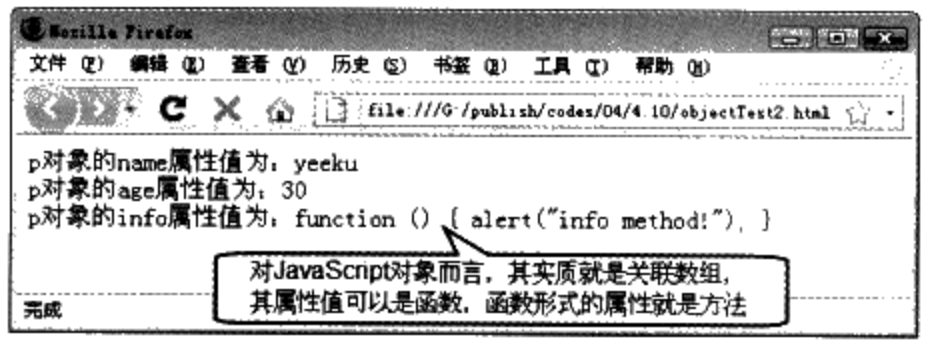


图 4.45　遍历 JavaScript 对象的属性

JavaScript 的函数没有提供显式的继承语法，因而 JavaScript 中的对象全部是 Object 的子类。这在前面已经介绍过了，因而各对象之间完全平等，并不存在直接的父子关系。JavaScript 提供了一些内建类，通过这些内建类可以方便地创建对象。

➤➤4.10.3　继承和 prototype

在面向对象的程序设计语言里，类与类之间有显式的继承关系，一个类可以显式指定继承哪个类，子类将具有父类的所有属性和方法。JavaScript 虽然也支持类、对象的概念，但没有继承的概念，只能通过一种特殊的手段来扩展原有的 JavaScript 类。

事实上，每个 JavaScript 对象都是相同基类（Object 类）的实例，因此所有 JavaScript 对象之间并没有明显的继承关系。而且 JavaScript 是一种动态语言，它允许自由地为对象增加属性和方法，当程序为对象的某个不存在的属性赋值时，即可认为是为该对象增加属性。例如如下代码：

```
//定义一个对象，该对象没有任何属性和方法
var p = {};
//为 p 对象增加 age 属性
p.age = 30;
//为 p 对象增加 info 属性，该属性值是函数，也就是增加了 info 方法
p.info = function()
{
    alert("info method!");
}
```

前面介绍过：定义 JavaScript 函数的同时，会得到一个同名的类，而且该函数就是该类的构造器。

因此，我们认为定义一个函数的同时，实质上也是定义了一个构造器。

在定义函数时，函数中以 this 修饰的属性是实例属性，如果某个属性值是函数，则可认为该属性变成了方法。例如如下代码：

程序清单：codes\04\4.10\classTest.html

```
<script>
    //创建 Person 函数
    function Person(name , age)
    {
        this.name = name;
        this.age = age;
        //为 Person 对象指定 info 方法
        this.info = function()
        {
            //输出 Person 实例的 name 和 age 属性
            document.writeln("姓名: " + this.name);
            document.writeln("年龄: " + this.age);
        }
    }
    //创建 Person 实例 p1
    var p1 = new Person('yeeku' , 29);
    //执行 p1 的 info 方法
    p1.info();
    document.writeln("<hr />");
    //创建 Person 实例 p2
    var p2 = new Person('wawa' , 20);
    //执行 p2 的 info 方法
    p2.info();
</script>
```

代码中定义 Person 函数的同时，也定义了一个 Person 类，而且该 Person 函数就是该 Person 类的构造器，该构造器不仅为 Person 实例完成了属性的初始化，还为 Person 实例提供了一个 info 方法。

但使用上面的方法为 Person 对象增加 info 方法相当不好，主要有如下两个原因：

性能低下：因为每次创建 Person 实例时，程序依次向下执行，每次执行程序中斜体字代码都将创建一个新 info()函数——创建多个 Person 对象时，系统就会有很多个 info()函数——这就会造成系统内存泄露，从而引起性能下降。实际上，info()函数只需要一个就够了。

使得 info()函数中的局部变量产生闭包：闭包会扩大局部变量的作用域，使得局部变量一直存活到函数之外的地方。看如下代码：

程序清单：codes\04\4.10\closureTest.html

```
<script>
    //创建 Person 函数
    function Person()
    {
        //locVal 是个局部变量，原本应该在该函数结束后立即失效
        var locVal = '疯狂 Java 联盟';
        this.info = function()
        {
            //此处会形成闭包
            document.writeln("locVal 的值为: " + locVal);
            return locVal;
        }
    }
    var p = new Person();
    //调用 p 对象的 info()方法
```

```
    var val = p.info();
    //输出返回值 val，该返回值就是局部变量 locVal
    alert(val);
</script>
```

从上面的代码中可以看出，由于 info 函数里访问了局部变量 locVal，所以形成了闭包，从而导致 locVal 的作用域被扩大。在最后一行粗体字代码处可以看到，即使离开了 info 函数，程序依然可以访问到局部变量的值。

为了避免这两种情况，通常不建议直接在函数定义（也就是类定义）中直接为该函数定义方法，而是建议使用 prototype 属性。

JavaScript 的所有类（也就是函数）都有一个 prototype 属性，在为 JavaScript 类的 prototype 属性增加函数、属性时，即可视为对原有类的扩展。我们可理解为：增加了 prototype 属性的类继承了原有的类——这就是 JavaScript 所提供的伪继承机制。看如下程序：

程序清单：codes\04\4.10\prototypeTest.html

```
<script>
    //定义一个 Person 函数，同时也定义了 Person 类
    function Person(name , age)
    {
        //将局部变量 name、age 赋值给实例属性 name、age
        this.name = name;
        this.age = age;
        //使用内嵌的函数定义 Person 类的方法
        this.info = function()
        {
            document.writeln("姓名：" + this.name + "<br />");
            document.writeln("年龄：" + this.age + "<br />");
        }
    }
    //创建 Person 的实例 p1
    var p1 = new Person('yeeku' , 29);
    //执行 Person 的 info 方法
    p1.info();
    //此处不可调用 walk 方法，变量 p 还没有 walk 方法
    //将 walk 方法增加到 Person 的 prototype 属性上
    Person.prototype.walk = function()
    {
        document.writeln(this.name + '正在慢慢溜达...<br />');
    }
    document.writeln('<hr />');
    //创建 Person 的实例 p2
    var p2 = new Person('leegang' , 30);
    //执行 p2 的 info 方法
    p2.info();
    document.writeln('<hr />');
    //执行 p2 的 walk 方法
    p2.walk();
    //此时 p1 也具有了 walk 方法——JavaScript 允许为类动态增加方法和属性
    //执行 p1 的 walk 方法
    p1.walk();
</script>
```

上面的程序中粗体字代码为 Person 类的 prototype 属性增加了 walk 函数，即可认为程序为 Person 类动态增加了 walk 实例方法——实际上，JavaScript 是一门动态语言，它不仅可以为对象动态增加属性和方法，也可为类动态增加属性和方法。

在为 Person 类增加 walk()实例方法之前，对象 p1 不能调用 walk()方法；在为 Person 类增加了 walk()实例方法之后，新创建的对象 p2，以及前面创建的对象 p1，就都拥有了 walk()方法，所以可以调用 walk()方法。

上面的程序利用 prototype 为 Person 类增加了一个 walk()方法，这样会让所有 Person 实例共享一个 walk()方法，而且该 walk()方法不在 Person 函数之内，因此不会产生闭包。

与 Java 等真正面向对象的继承不同，虽然使用 prototype 属性可以为一个类动态增加属性和方法，这可被当成一种“伪继承”；但这种“伪继承”的实质是修改了原有的类，而不是产生一个新的子类，这一点尤其需要注意。因此，原有的那个没有 walk()方法的 Person 类将不再存在！

注意：

JavaScript 并没有提供真正的继承，当通过某个类的 prototype 属性动态增加属性或方法时，其实质是对原有类的修改，而不是真正产生一个新的子类，所以这种机制依然只是一种伪继承。

通过使用 prototype 属性，可以对 JavaScript 的内建类进行扩展。下面的代码为 JavaScript 内建类 Array 增加了 indexof 方法，该方法用于判断数组中是否包含了某元素。看下面的代码：

程序清单：codes\04\4.10\extendsArray.html

```
<script>
    //为 Array 增加 indexof 方法，将该函数增加到 prototype 属性上
    Array.prototype.indexof = function(obj)
    {
        //定义 result 的值为-1
        var result = -1;
        //遍历数组的每个元素
        for (var i = 0 ; i < this.length ; i ++)
        {
            //当数组的第 i 个元素值等于 obj 时
            if (this[i] == obj)
            {
                //将 result 的值赋为 i，并结束循环
                result = i;
                break;
            }
            }
            //返回元素所在的位置
            return result;
    }
    var arr = [4, 5, 7, -2];
    //测试为 arr 新增的 indexof 方法
    alert(arr.indexof(-2));
</script>
```

上面的程序中第一段粗体字代码为 Array 类动态增加了 indexof()方法，使得代码后所有数组对象都可直接使用 indexof()方法，程序中最后一行粗体字代码就直接测试使用了数组对象的 indexof()方法。

如果将上面的代码放在 JavaScript 代码最上面，则代码中所有数组都会增加 indexof 方法。一定要将这段代码放在 JavaScript 脚本的开头，因为只有将 indexof 方法增加到 prototype 属性之后，创建的 Array 实例才有 indexof 方法。

虽然可以在任何时候为一个类增加属性和方法，但通常建议在类定义结束后立即增加该类所需的方法，这样可避免造成不必要的混乱。同时，对于需要在类定义中定义方法的情形，尽量避免直接在类定义中定义方法，这样可能造成内存泄漏和产生闭包。比较安全的方式是通过 prototype 属性为该类

增加属性和方法。

与传统面向对象程序设计语言不同的是，JavaScript 中的函数始终都是独立的，函数始终是一等公民，函数永远不会从属于其他类、对象。

即使匿名内嵌函数，在使用匿名内嵌函数定义某个类的方法时，该内嵌函数一样是独立存在的，该函数也不是完全作为该类实例的附庸存在，这些内嵌的函数也可以被分离出来独立使用，包括成为另一个对象的函数。看下面的代码：

程序清单：codes\04\4.10\separateFunction.html

```
<script>
    //定义 Dog 函数，等同于定义了 Dog 类
    function Dog(name , age , bark)
    {
        //将形参 name、age、bark 赋值给实例属性 name、age、bark
        this.name = name;
        this.age = age;
        this.bark = bark;
        //使用内嵌函数为 Dog 实例定义方法
        this.info = function()
        {
            return this.name + "的年纪为：" + this.age
                + ",它的叫声:" + this.bark;
        }
    }
    //创建 Dog 的实例
    var dog = new Dog("旺财" , 3 , '汪汪,汪汪...');
    //创建 Cat 函数，对应 Cat 类
    function Cat(name,age)
    {
        this.name = name;
        this.age = age;
    }
    //创建 Cat 实例
    var cat = new Cat("kitty" , 2);
    //将 dog 实例的 info 方法分离出来
    var tmp = dog.info;
    //通过 function 的 call 方法完成 cat 的 info 方法调用
    alert(tmp.call(cat));
</script>
```

上面的程序中第一段粗体字代码使用内嵌函数为 Dog 定义了名为 info 的实例方法，但这个 info 方法依然不从属于 Dog 实例，它依然是一个独立函数，所以程序在倒数第二行粗体字代码处将该函数分离出来，最后一行粗体字代码让 Cat 实例调用了该 info 方法。

执行这段代码，可看到如图 4.46 所示执行结果。

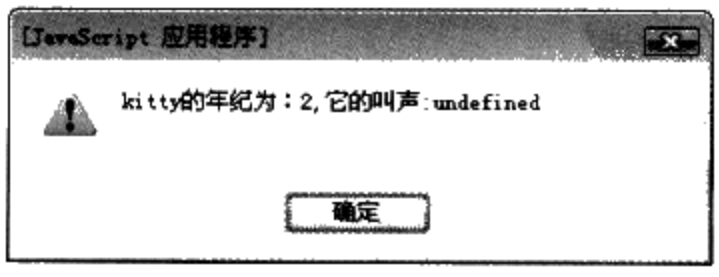


图 4.46 内嵌函数独立使用

提 示：

JavaScript 中的方法调用有两种方式，常用方式是前面一直所用的形式：obj.method(args...)；除此之外，还有另外一种调用方式：method.call(obj , args...)。上面的程序中最后一行代码通过 cat 调用 info 方法就是使用的这种形式。

前面已介绍过，尽量不要使用内嵌函数为类定义方法，而应该使用增加 prototype 属性的方式来增加方法。与此类似的是，利用增加 prototype 属性增加的函数一样可以被分离出来独立使用。分离方法与前面分离内嵌函数的方法完全一样，此处不再赘述。

通过 prototype 属性来为类动态增加属性和方法会让程序更加安全，性能更稳定。

4.11 建对象

正如前文所介绍的，JavaScript 对象是一个特殊的数据结构，JavaScript 的对象只是一种特殊的关联数组。创建对象并不是总需要先创建类，与纯粹面向对象语言不同的是，JavaScript 中创建对象可以不使用任何类。JavaScript 中创建对象大致有三种方式：

- 使用关键字 new 创建对象。
- 使用 Object 创建即时对象。
- 使用 JSON 语法创建对象。

➤➤4.11.1 使用关键字 new 创建对象

这是最接近面向对象语言创建对象的方式，关键字 new 后紧跟函数的方式非常类似于 Java 中 new 后紧跟构造器的方式，这种方式创建对象简单、直观。但需要接受的是：JavaScript 中所有的函数都可以作为构造器使用，使用 new 调用函数后总可以返回一个对象。看如下代码：

程序清单：codes\04\4.11\newObject.html

```
<script>
    //定义一个函数，同时也定义了一个 Person 类
    function Person(name , age)
    {
        //将形参 name、age 赋值给实例属性 name、age
        this.name = name;
        this.age = age;
    }
    //分别以两种方式创建 Person 实例
    var p1 = new Person();
    var p2 = new Person('yeeku' , 29);
    //输出 p1 的属性
    document.writeln("p1 的属性如下:"
        + p1.name + p1.age + "<br />");
    //输出 p2 的属性
    document.writeln("p2 的属性如下:"
        + p2.name + p2.age);
</script>
```

在上面的代码中，以两种不同方式创建了 Person 对象。因为 JavaScript 支持空参数特性，所以调用函数时，依然可以不传入参数，如果没有传入参数，则对应参数值是 undefined。

前面已经介绍过，在函数中使用 this 修饰的变量是该函数的实例属性，以函数名修饰的变量则是该函数的静态属性。实例属性以实例访问，静态属性则以函数名访问。以这种方式创建的对象是 Person 的实例，也是 Object 的实例。上面的代码的执行结果是 p1 的两个属性都是 undefined，而 p2 的 name 属性为 yeeku，age 属性为 29。

➤➤4.11.2 使用 Object 直接创建对象

JavaScript 的对象都是 Object 类的子类，因此可以采用如下方法创建对象：

```
//创建一个默认对象
var myObj = new Object();
```

这是空对象，该对象不包含任何的属性和方法。与 Java 不同的是，JavaScript 是动态语言，因此可以为该对象动态增加属性和方法。在静态语言（如 Java、C#）中，一个对象一旦创建成功，它所包含的属性值可以变化，但属性的类型、个数都不可改变，也不可增加方法。

但 JavaScript 既可以为对象动态增加方法，也可动态增加属性。看如下代码：

程序清单：codes\04\4.11\dynaObject.html

```
<script>
    //创建空对象
    var myObj = new Object();
    //增加属性
    myObj.name = 'yeeku';
    //增加属性
    myObj.age = 29;
    //输出对象的两个属性
    document.writeln(myObj.name + myObj.age);
</script>
```

上面的代码直接为对象增加了两个属性，这种语法从某个侧面反映了 JavaScript 对象的本质：它是一个特殊的关联数组。事实上，JavaScript 完全允许使用数组语法来访问属性，4.10.2 节已经看到过这种访问方式。

在 4.10.3 节的代码中使用了匿名函数为对象增加方法。此处没有必要使用有名字的函数，当然也可以使用有名字的函数，例如：

```
//为对象增加方法
myObj.info1 = function abc()
{
    document.writeln("对象的 name 属性:" + this.name);
    document.writeln("<br>");
    document.writeln("对象的 age 属性:" + this.age);
};
```

正如前面提到的，JavaScript 还可以通过 new Function()的方法来定义匿名函数，因此完全可以通过这种方式来为 JavaScript 对象增加方法，如以下代码所示：

程序清单：codes\04\4.11\dynaObject2.html

```
<script>
    var myObj = new Object();
    myObj.name = 'yeeku';
    myObj.age = 29;
    //为对象增加方法
    myObj.info = new Function("document.writeln('对象的 name 属性:' + this.name);"
        + "document.writeln('<br />');"
        + "document.writeln('对象的 age 属性:' + this.age)");
    document.writeln("<hr / >");
    myObj.info();
</script>
```

除此之外，JavaScript 也允许将一个已有的函数添加为对象的方法，看如下代码：

程序清单：codes\04\4.11\dynaObject3.html

```
<script>
    //创建空对象
    var myObj = new Object();
    //为空对象增加属性
    myObj.name = 'yeeku';
    myObj.age = 29;
    //创建一个函数
```

```
    function abc()
    {
        document.writeln("对象的 name 属性:" + this.name);
        document.writeln("<br />");
        document.writeln("对象的 age 属性:" + this.age);
    };
    //将已有的函数添加为对象的方法
    myObj.info = abc;
    document.writeln("<hr>");
    //调用方法
    myObj.info();
</script>
```

上面的程序中第一段粗体字代码定义了一个普通函数，程序的最后一行粗体字代码将 abc 函数直接赋值给 myObj 对象的 info 方法，这样就为 myObj 对象添加了一个 info 方法。

值得指出的是：将已有的函数添加为对象方法时，不能在函数名后添加括号。一旦添加了括号，将表示调用函数，则不再是将函数本身赋给对象的方法，而是将函数的返回值赋值给对象的属性。

注意：

为对象添加方法时，不要于定义函数后添加括号。一旦添加了括号，将表示要把函数的返回值赋值给对象的属性。

➤➤4.11.3 使用 JSON 语法创建对象

JSON（JavaScript Object Notation）语法提供了一种更简单的方式来创建对象，使用 JSON 语法可避免书写函数，也可避免使用 new 关键字，可以直接创建一个 JavaScript 对象。为了创建 JavaScript 对象，可以使用花括号，然后将每个属性写成“key : value”对的形式。

对于早期的 JavaScript 版本，如果要创建一个对象，通常情况下可能会这样写：

```
//定义一个函数，作为构造器
function Person(name, sex)
{
    this.name = name;
    this.sex = sex;
}
//创建一个 Person 实例
var p = new Person('yeeku', 'male');
```

从 Javascript 1.2 开始，创建对象有了一种更快捷的语法，语法如下：

```
var p = {
    "name": 'yeeku',
    "gender" : 'male'
};
alert(p);
```

这种语法就是一种 JSON 语法。显然，使用 JSON 语法创建对象更加简洁，更加方便。图 4.47 显示了这种语法示意。

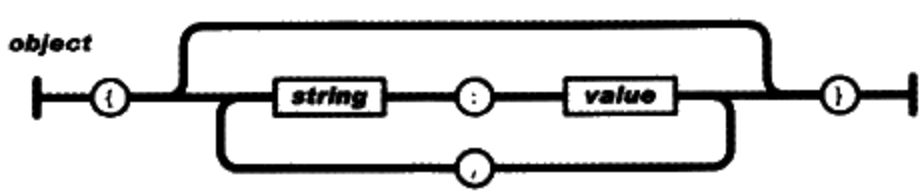


图 4.47　JSON 创建对象的语法示意

图 4.46 中，创建对象时，总以{开始，以}结束，对象的每个属性名和属性值之间以英文冒号（:）

隔开，多个属性定义之间以英文逗号（,）隔开。语法格式如下：

```
object =
{
    propertyName1 : propertyValue1 ,
    propertyName2 : propertyValue2 ,
    ...
}
```

必须注意的是，并不是每个属性定义后面都有英文逗号（,），只有在后面还有属性定义时才需要逗号（,），也就是最后一个属性定义后不再有英文逗号（,）。因此，下面的对象定义是错误的。

```
person =
{
    name : 'yeeku',
    gender: 'male',
}
```

因为 gender 属性定义后多出一个英文逗号，最后一个属性定义的后面应该直接以}结束，不能再有英文的逗号（,）。

使用 JSON 语法创建 JavaScript 对象时，属性值不仅可以是普通字符串，也可以是任何基本数据类型，还可以是函数、数组，甚至是另外一个 JSON 语法创建的对象。例如：

```
person =
{
    name : 'yeeku',
    gender : 'male',
    //使用 JSON 对象为其指定一个属性
    son : {
        name:'nono',
        grade:1
    },
    //使用 JSON 语法为 person 直接分配一个方法
    info : function()
    {
        document.writeln("姓名: " + this.name + "性别: " + this. gender);
    }
}
```

JSON 语法不仅可用于创建对象，使用 JSON 语法创建数组也是非常常见的情形。在早期的 JavaScript 语法里，我们通过如下方式来创建数组：

```
//创建数组对象
var a = new Array();
//为数组元素赋值
a[0] = 'yeeku';
//为数组元素赋值
a[1] = 'nono';
```

或者也可以通过如下方式创建数组：

```
//创建数组对象时直接赋值
var a = new Array('yeeku', 'nono');
```

但如果我们使用 JSON 语法，则可以通过如下方式创建数组：

```
//使用 JSON 语法创建数组
var a = ['yeeku', 'nono'];
```

图 4.48 是 JSON 创建数组的语法示意。

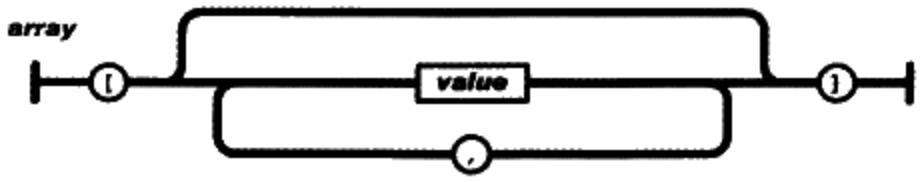


图 4.48　使用 JSON 创建数组的语法示意

正如图 4.48 中所见到的，JSON 创建数组总是以英文方括号（[）开始，然后依次放入数组元素，元素与元素之间以英文逗号（,）隔开，最后一个数组元素后面不需要英文逗号，以英文反方括号（]）结束。使用 JSON 创建数组的语法格式如下：

```
arr = [value1 , value 2 ...]
```

与 JSON 语法创建对象相似的是，数组的最后一个元素后面不能有逗号（,）。如下代码定义了一个更复杂的 JSON 对象。

程序清单：codes\04\4.11\json.html

```
<script>
    //代码
    var person =
    {
        //定义第一个简单属性
        name : 'wawa',
        //定义第二个简单属性
        age : 29 ,
        //定义第三个属性，数组
        schools : ['小学' , '中学' , "大学"],
        //定义第四个属性，对象数组
        parents :[
            {
                name : 'father',
                age : 60,
                address : '广州'
            }
            ,
            {
                name : 'mother',
                age : 58,
                address : '深圳'
            }
        ]
    };
    alert(person.parents);
</script>
```

实际上，JSON 已经发展成一种轻量级的、跨语言的数据交换格式。目前已经明确支持 JSON 语法的编程语言非常多，比如 Java、C/C++、C#、Ruby、Python、PHP、Perl 等主流编程语言都支持 JSON 格式的数据。JSON 的官方站点是 http://www.json.org，读者可以登录该站点了解关于 JSON 的更多信息。

由于 JSON 格式的数据交换具有轻量级、易理解、跨语言的优势，因此 JSON 格式已成为 XML 数据交换格式的有力竞争者。假设需要交换一个 Person 对象，其 name 属性为 yeeku，gender 属性为 male，age 属性为 29，使用 JSON 语法可写成如下形式：

```
person =
{
    name:'yeeku',
    gender:'male',
    age:29
}
```

如果使用 XML 数据交换格式，则需要写成如下形式：

```
<person>
    <name>yeeku</name>
    <gender>male</gender>
    <age>29</age>
</person>
```

对比两种表示方式，前一种方式明显比第二种方式更加简洁，数据传输量也更小。因此，在需要跨平台、跨语言地进行数据交换时，有时候宁愿选择 JSON 作为数据交换格式，而不是 XML。

4.12 本章小结

本章主要介绍了 Ajax 技术的基础：JavaScript 语言相关知识，包括 JavaScript 的变量、数据类型等，并全面介绍了 JavaScript 的各种运算符，还介绍了 JavaScript 的流程控制语句。本章重点介绍了 JavaScript 的函数，介绍函数的同时，力图全面阐释 JavaScript 基于对象的特征，以及它与面向对象语言存在差异的地方。JavaScript 的类、对象和伪继承等也是本章介绍的重点。本章另一个重要的知识点是掌握创建对象的三种语法。

本章的难点是理解 JavaScript 函数的复杂性：JavaScript 的函数既是一个函数，也是一个类，而且还是该类唯一的构造器。因此 JavaScript 有两种调用函数的方式，普通调用和使用关键字 new 调用。本章的另一个难点是理解 JavaScript 的动态特征，它既允许为对象动态地增加属性和方法，也允许通过 prototype 为类动态地增加属性和方法。

➤➤本章练习

使用 JavaScript 完成数值转换人民币读法的程序，例如输入 1001.235，程序转换后得到壹仟零壹元贰角三分（小数点后只处理 2 位，多余部分自动截断）。

使用 JavaScript 完成一个对字符串数组进行排序的小程序。排序依据是字符串中重复最多的字符的出现次数，例如 aaab（a 重复 3，该字符串的排序权值为 3）、ababxyxy（该字符串的排序权值为 2）、abcxyz（该字符串的排序权值为 1），排序后应该是 abcxyz、ababxyzy、aaab。

CHAPTER

5

第5章
级联样式单详解

本章要点

- 样式单概述
- 样式单的优势
- 级联样式单的三种使用方式
- 级联样式单的段落、文字相关属性
- 级联样式单的大小、定位属性
- 级联样式单的边框、轮廓属性
- 级联样式单中的表格属性
- 级联样式单中的背景色属性
- 属性选择器
- ID 选择器
- class 选择器
- 包含和子元素选择器
- 超级链接相关选择器
- 在脚本中控制级联样式单属性值

Cascading Style Sheet（级联样式单），缩写为 CSS，也被称为层叠样式单。主要用于网页风格设计，包括字体大小、颜色，以及元素的精确定位等。在传统的 Web 网页设计里，使用 CSS 能让原来单调的 XHTML 网页更富表现力。对于 Ajax 应用，在客户端 JavaScript 获取了服务器数据后，需要将数据动态显示在当前页面，这里的显示，不是简单的输出，当然也包括了丰富的显示格式，这些丰富的表现格式自然离不开 CSS。

CSS 2.0 是目前广泛使用的级联样式单规范，该规范推荐的是一套内容和表现效果分离的方式，HTML 元素可以通过 CSS 2.0 的样式控制显示效果，可完全不使用以往 XHTML 中的<table.../>和<td.../>等元素进行页面布局，只需使用 div 和 li 此类简单的 XHTML 标签，然后即可通过 CSS 2.0 样式来定义页面布局。CSS 2.0 提供了一种机制：让程序员开发时可以不考虑显示和界面，显示问题由美工或程序员在后期编写相应的 CSS 2.0 样式单来解决。

归纳起来，CSS 主要用于控制 XHTML 页面中元素的大小、位置、背景、颜色等外观，因而主要作用就是美化 XHTML 页面。

5.1 样式单概述

样式单（Style Sheet）是一种专门描述结构文档表现方式的文档，它既可以描述这些文档如何在屏幕上显示，也可以描述它们的打印效果，甚至声音效果。样式单一般不包含在结构化文档的内部，而以独立的文档方式存在。与 HTML 描述数据显示方式的传统方法相比，样式单有许多突出的优点：

- 表达效果丰富：样式单可以支持文字和图像的精确定位、三维层技术以及交互操作等，对于文档的表现力远远超过 XHTML 中的标记。更重要的是，样式单的标准规范独立于其他结构文档的规范，在需要实现更丰富的表达效果时，仅需修改样式单规范即可，无须修改原始的数据文档内容。
- 文档体积小：在实际应用中，如果相同标记下的内容有相同的表现方式，则使用传统的方法需要为每个标记分别定义显示格式，造成大量的重复定义。而在样式单中，对于同一类标记只需进行一次格式定义即可，大大缩小了需要传输的文件的体积，可提高传输速度，并节约带宽。
- 便于信息检索：虽然样式单可以实现非常复杂的显示效果，但样式单的显示逻辑与数据逻辑分离，显示细节的描述并不影响文档中数据的内在结构。因此，网络搜索引擎对文档进行检索时，更容易检索到有用信息。
- 可读性好：样式单对各种标记的显示进行集中定义，且定义方式直观易读。这使得它易学易用，可读性、可维护性都比较好。而结构化的数据文档也相对简洁、清晰，突出了对内容本身的描述功能。

正是由于样式单的这种种优点，W3C 组织大力提倡使用样式单描述结构文档的显示效果。迄今为止，W3C 已经给出了两种样式单语言的推荐标准，一种是级联样式单 CSS（Cascading Style Sheets），另一种是可扩展样式单语言 XSL（eXtensible Stylesheet Language）。

级联样式单是一系列格式规则，这些规则用于控制网页内容的外观：从精确的布局定位到特定的字体和样式，CSS 样式都可以一样表现出色，甚至对于一些网页特效，也可借助于 CSS 实现。CSS 样式单除了可用于控制 XHTML 文档的显示外，还可用于控制 XML 文档的显示格式。

CSS 也是一种样式单，因此也将数据逻辑和显示逻辑分离，从而可提高文件的可读性。除此之外，CSS 还可以提供其他的表现方式，例如声音（虽然这种情况很少见，但如果浏览者有这种需求也是可实现的）。CSS 主要提供如下两个功能：

- 对页面的字体、颜色控制更加细腻，使页面内容更富表现力，CSS 的表现效果远远超出传统 HTML 页面的 color、bgcolor 等属性的表现力。
- 通过 CSS 控制整站风格。CSS 可以同时控制整个站点所有页面的风格，如果需要整个站点所

有的页面效果都改变，可直接通过 CSS 控制，避免逐个修改每个页面文件。

5.2 CSS 的基本使用

CSS 可以控制 XML 文档的显示，也可以控制 XHTML 文档的显示。但在控制文档的显示之前，首先应在需要显示的结构化文档中导入 CSS。在 XHTML 文档中使用 CSS，有如下三种方式：

- 引入外部样式文件：这种方式将样式文件彻底与 XHTML 文档分离，样式单文件需要额外引入。在这种方式下，一批样式可控制多份文档。
- 使用内部样式定义：这种方式是在 XHTML 文档头定义样式单部分。在这种方式下，每批 CSS 样式只控制一份文档。
- 使用内联样式：这种方式将样式内联定义到具体的 XHTML 元素，这种方式通常用于精确控制一个 HTML 元素的表现。在这种方式下，每份 CSS 样式只控制单个 HTML 元素。

下面依次介绍每种使用样式单的方式。

5.2.1 引入外部样式文件

XHTML 文档中使用<link.../>元素来引入外部样式文件，引入外部样式文件应在<head.../>元素中增加如下<link.../>子元素：

```
<link type="text/css" rel="stylesheet" href="CSS 样式文件的 URL">
```

上面的语法格式中，type 和 rel 表明该页面使用了 CSS，对于引入 CSS 的情形，这两个属性的值无须改变。href 属性的值则指向 CSS 文件的地址，此处的地址既可以是相对地址，也可以是互联网上的绝对地址。

下面是一个简单的 HTML 文档的代码，该文档没有提供任何的显示格式，只是简单的 HTML 表格，包含了三个字符串：

程序清单：codes\05\5.2\outer.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>外部 CSS 样式单测试</title>
</head>
<body>
<table width="210" border="0">
    <tr>
        <td width="188">疯狂 Java 讲义</td>
    </tr>
    <tr>
        <td>轻量级 Java EE 企业开发实战</td>
    </tr>
    <tr>
        <td>疯狂 Ajax 讲义</td>
    </tr>
</table>
</body>
</html>
```

上面的页面中仅仅包含了一个简单的表格，并指定了表格宽度，没有其他任何显示格式，在浏览器中浏览该页面可看到如图 5.1 所示的简单页面。

图 5.1　没有 CSS 样式的 HTML 页面

为了让该页面更富表现力，可以为该页面指定外部的 CSS 文件，引入外部 CSS 文档的语法格式见上面的介绍。在该页面的<head.../>元素内插入如下<link.../>子元素：

```
<!-- 引入 outer.css 样式单文件 -->
<!-- link href="outer.css" rel="stylesheet" type="text/css" /-->
```

outer.css 样式单文件的代码如下：

程序清单：codes\05\5.2\outer.css

```
/* 设置整个表格的背景色 */
table {
    background-color: #003366;
}
/* 设置单元格的背景色、过滤器、字体等 */
td {
    background-color: #FFFFFF;
    filter: Blur(Add=20, Direction=20, Strength=20);
    font-family: "楷体_GB2312";
}
```

正如在上面的 CSS 样式文件中所见到的，CSS 总是由一个或多个样式定义组成，每个样式通常有如下语法格式：

```
Selector { property: value }
```

其中 Selector 是应用样式的选择器，Selector 决定对哪些 XHTML 元素起作用；而花括号里的属性名、属性值则指定字体、大小、背景、颜色等，也就是决定 XHTML 元素起怎样的作用。

图 5.2 显示了应用样式单后的页面效果。

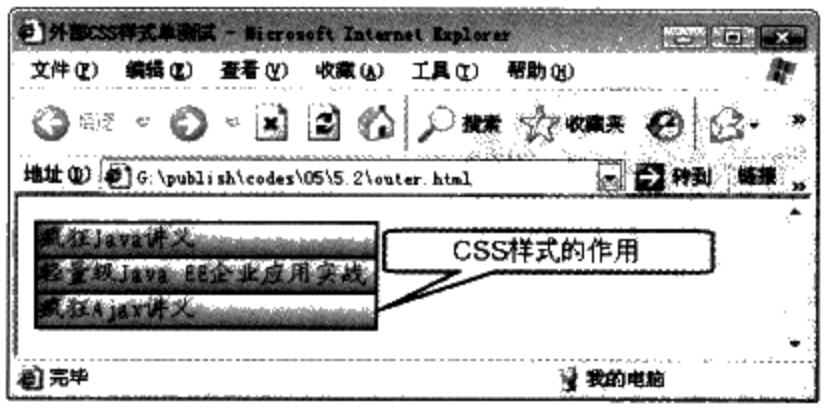


图 5.2　引入外部 CSS 后的效果

➤➤5.2.2　使用内部 CSS 样式

一般来说，我们不建议使用内部 CSS 样式，因为这种做法需要在 XHTML 文档内嵌入 CSS 样式定义，这种内部 CSS 样式主要有二大劣势：

➢　如果该 CSS 样式需要被其他 HTML 文档使用，那么这些 CSS 样式必须在其他 HTML 文档中

重复定义。

- 大量 CSS 嵌套在 HTML 文档中，必将导致 HTML 文档过大，大量的重复下载，导致网络负载加重。
- 如果需要修改整站风格，则必须依次打开每个页面重复修改，不利于软件工程化管理。

但内部样式定义也并非一无是处，如想让某些 CSS 样式仅对某个页面有效，而不会影响整个站点，则应该选择使用内部 CSS 样式定义，对于上面的 HTML 页面，可以使用内部 CSS。

内部 CSS 样式需要放在<style.../>元素中定义，每个 CSS 样式定义与外部 CSS 样式文件的内容完全相同。<style.../>元素应该放在<head.../>元素内，作为它的子元素。

使用内部 CSS 样式定义的语法格式如下：

```
<style type="text/css">
    样式单文件定义
</style>
```

下面是使用内部 CSS 的 XHTML 文档的源代码：

程序清单：codes\05\5.2\inner.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>内部 CSS 样式单测试</title>
    <style type="text/css">
        table {
            background-color: #003366;
        }
        td {
            background-color: #FFFFFF;
            filter: Blur(Add=20, Direction=20, Strength=20);
            font-family: "楷体_GB2312";
        }
        .title {
            font-size: 18px;
            font-weight: normal;
            color: #6600CC;
            height: 30px;
            width: 200px;
            border-top: 3px solid #CCCCCC;
            border-left: 3px solid #CCCCCC;
            border-bottom: 3px solid #000000;
            border-right: 3px solid #000000;
        }
    </style>
</head>
<body>
<div class="title">
疯狂 Java 体系图书：
</div><hr />
<table width="400" border="0">
    <tr>
        <td>疯狂 Java 讲义</td><td>经典 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>轻量级 Java EE 企业应用实战</td><td>疯狂 XML 讲义</td>
    </tr>
```

```
        <tr>
            <td>疯狂 Ajax 讲义</td><td>疯狂 Workflow 讲义</td>
        </tr>
    </table>
    </body>
    </html>
```

上面的页面中粗体字代码定义了三个样式定义，这三个样式定义都被嵌套在该 XHTML 页面内，因此该 CSS 样式定义仅对当前页面起作用，在浏览器中浏览该页面，可看到如图 5.3 所示效果。

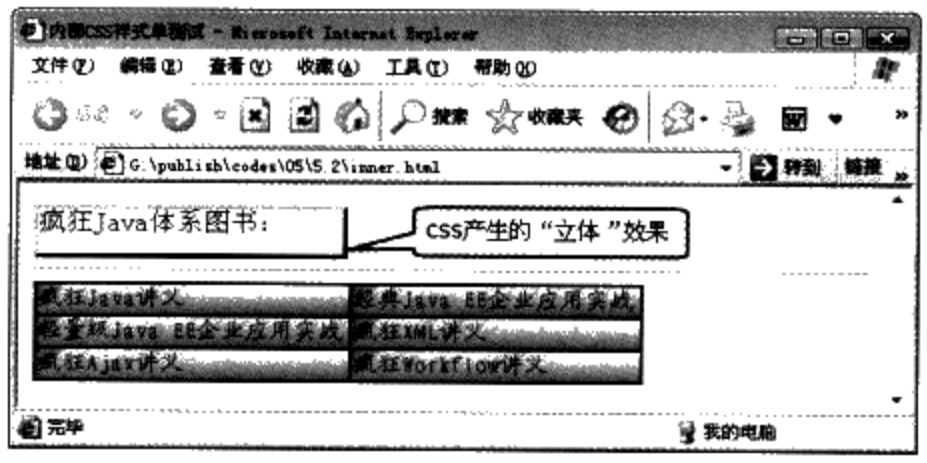


图 5.3　使用内部样式单文件的效果

➤➤5.2.3　使用内联样式

内联 CSS 样式只对单个标签有效，它甚至不会影响整个文件。内联样式定义可以精确控制某个 XHTML 元素的外观表现，并且允许通过 JavaScript 动态修改 XHTML 元素的 CSS 样式，从而改变该元素的外观。

为了使用内联样式，CSS 扩展了 XHTML 元素，几乎所有 XHTML 元素都增加了一个 style 核心属性，该属性值是一个或多个 CSS 样式定义，多个 CSS 样式定义之间以英文分号隔开。简单地说，使用内联样式定义时，style 属性值就是一个或多个 property: value 的组合，此处的 property: value 与前面 CSS 文件中的完全相同。

定义内联 CSS 样式的语法格式如下：

```
style="property1:value1;property2:value2..."
```

如果需要将 5.2.2 节中的样式文件改为使用内联样式定义，其 HTML 源代码如下：

程序清单：codes\05\5.2\inline.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>内联 CSS 样式单测试</title>
</head>
<body>
<div style="font-size: 18px;
            font-weight: normal;
            color: #6600CC;
            height: 30px;
            width: 200px;
            border-top: 3px solid #CCCCCC;
            border-left: 3px solid #CCCCCC;
            border-bottom: 3px solid #000000;
            border-right: 3px solid #000000;">
疯狂 Java 体系图书：
```

```
</div><hr />
<table width="400" border="0" style="background-color: #0099bb;">
    <tr>
        <td style="background-color: #FFFFFF;
        filter: Blur(Add=20, Direction=20, Strength=20);
        font-family: '楷体_GB2312';">疯狂 Java 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td style="background-color: #FFFFFF;
        filter: Blur(Add=20, Direction=20, Strength=20);
        font-family: '楷体_GB2312';">轻量级 Java EE 企业应用实战</td>
        <td>疯狂 XML 讲义</td>
    </tr>
    <tr>
        <td style="background-color: #FFFFFF;
        filter: Blur(Add=20, Direction=20, Strength=20);
        font-family: '楷体_GB2312';">疯狂 Ajax 讲义</td>
        <td>疯狂 Workflow 讲义</td>
    </tr>
</table>
</body>
</html>
```

上面的粗体字代码分别为不同的 XHTML 元素指定了 style 属性，注意这些 style 属性值，它们就是前面样式单文件里的样式定义部分。只是在使用内联样式时，已经直接将这些样式定义关联到了具体的 XHTML 元素，因此不再需要指定 Selector。

注意：

读者一定要牢记：在定义一个 CSS 样式时，需要指定两个部分：Selector，该部分决定对哪些 XHTML 元素起作用；属性定义部分，如{property: value...}，这些属性定义决定对 XHTML 元素起怎样的作用。在使用内联方式定义 CSS 样式时，CSS 样式定义直接指定到具体的 XHTML 元素，因此无须指定 Selector 部分。

上面的页面代码中粗体字 CSS 样式定义指定到具体的 XHTML 元素，因而这些 CSS 样式只对具有 style 属性的 XHTML 元素起作用。在浏览器中浏览该页面，可看到如图 5.4 所示效果。

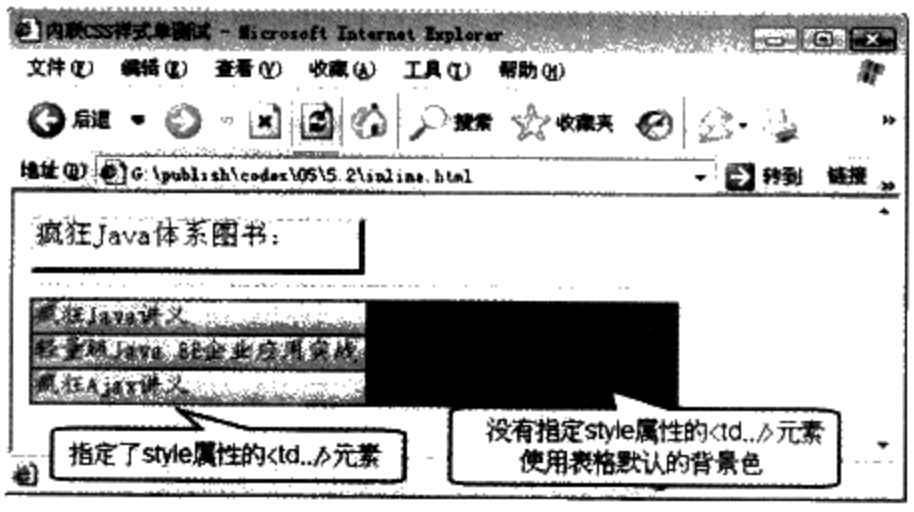


图 5.4　内联 CSS 样式的效果

5.3　使用 CSS 属性

正如在上面的介绍中所见到的，CSS 样式定义的格式要么是以样式单的形式存在，要么是以内联样式存在。不管以何种形式存在，都不可避免需要使用如下格式：

```
property:value
```

其中前面的 property 就是 CSS 样式的属性，通过这些合法的 CSS 属性可以控制 XHTML 文档的显示外观。下面将依次介绍 CSS 的常用属性和合法属性值。在介绍这些常用属性之前，读者必须明白，本节所介绍的常用 CSS 属性都是 Ajax 应用需要控制的 CSS 属性，至于一些特殊的 CSS 属性（例如滤镜），则不在本节的介绍范围内。

➤➤5.3.1 文字相关属性

文字相关属性主要用于控制文字的字体、颜色、修饰、阴影属性。常用的文字相关的属性如下：

- font：适用于 CSS1 和 CSS2。这是一个复合属性，使用 font 属性可同时控制文字的颜色、字体等属性，为了更具体地进行控制，通常不建议使用该属性。
- color：适用于 CSS1，该属性专用于控制文字颜色，其值通常采用十六进制的形式设置，也可以直接设置其 RGB 值，使用 RGB 值时应使用 rgb()函数。
- font-family：适用于 CSS1，设置文字的字体，因为字体需要浏览器的内嵌字体的支持，该属性可以设置多个显示字体，浏览器按该属性指定的多个字体依次搜索，以优先找到的字体来显示文字。多个属性值之间以英文逗号（,）隔开。
- font-size：适用于 CSS1，该属性用于设置文字的字体大小，此处的字体大小既可以是相对的字体大小，也可以是绝对的字体大小。但通常使用数字控制字体大小，例如 9pt。
- font-stretch：适用于 CSS2，用于改变文字横向的拉伸，该属性的默认值为 normal，即不拉伸。还有两个属性值 narrower 和 wider，前者是横向压缩，后者是横向拉伸。
- font-style：适用于 CSS1，该属性用于设置文字风格，即是否采用斜体等。该属性的常用属性值有 normal、italic、oblique 值，这些属性依次设置文字的正常、斜体、使用倾斜字体状态。
- font-weight：适用于 CSS1，该属性用于设置字体是否加粗。该属性的值是加粗的程度，加粗的程度有 lighter、normal、bold、bolder 等，即更细、正常、加粗、更粗。还可以使用具体的数值，从 100、200 一直到 900 来控制字体的加粗程度。
- text-decoration：适用于 CSS1，用于控制文字是否有修饰线，如下画线等。该属性有如下值：none、blink、underline、line-through 和 overline，分别对应的修饰效果为：无修饰、闪烁、下画线、中画线和上画线等。
- font-variant：适用于 CSS1，用于设置文字的大写字母的格式。
- text-shadow：适用于 CSS2，用于设置文字是否有阴影效果。可以设定多组效果，方式是用逗号隔开。
- text-transform：适用于 CSS1，用于设置文字的大小写。该属性值可以是 none、capitalize、uppercase 和 lowercase，分别表示不转换、首字母大写、全部大写和全部小写。
- line-height：适用于 CSS1，用于设置字体的行高，即字体最底端与字体内部顶端之间的距离。为负值的行高可用来实现阴影效果。
- letter-spacing：适用于 CSS1，用于设置文字之间的间隔。该属性将指定的间隔添加到每个文字之后，但最后一个文字不会受该属性的影响。
- word-spacing：适用于 CSS1，用于设置单词之间的间隔。

下面的代码对上面的各种属性设置做了简单的示范，有些 CSS 属性的显示效果在 Firefox 中可能不能得到很好的显示，但大部分属性的设置都可在浏览器中看到显示效果。该页面的代码非常清晰，左边是设置的属性值，右边是实际应用该设置后的效果，代码如下。

程序清单：codes\05\5.3\font.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>字体相关属性设置</title>
</head>
<body>
color:#888888;
<span style="color:#888888">测试文字</span><br />
font-family: 楷体_GB2312;
<span style="font-family: '楷体_GB2312'">测试文字</span><br />
font-size: 20pt;
<span style="font-size:20pt">测试文字</span><br />
font-stretch: narrower;
<span style="font-stretch:narrower">测试文字</span><br />
font-style: italic;
<span style="font-style:italic">测试文字</span><br />
font-weight: bold;
<span style="font-weight:bold">测试文字</span><br />
font-weight: 900;
<span style="font-weight:900">测试文字</span><br />
text-decoration: blink;
<span style="text-decoration: blink;">测试文字</span><br />
text-decoration: underline;
<span style="text-decoration:underline">测试文字</span><br />
text-decoration: line-through;
<span style="text-decoration:line-through">测试文字</span><br />
text-transform: uppercase;
<span style="text-transform:uppercase">hello</span><br />
line-height: 30pt;
<span style="line-height:30pt">测试文字</span><br />
letter-spacing: 20pt;
<span style="letter-spacing:20pt">测试文字</span><br />
word-spacing: 40pt;
<span style="word-spacing:40pt">测试文字</span><br />
</body>
</html>
```

图 5.5 显示了该页面在浏览器中测试的效果。

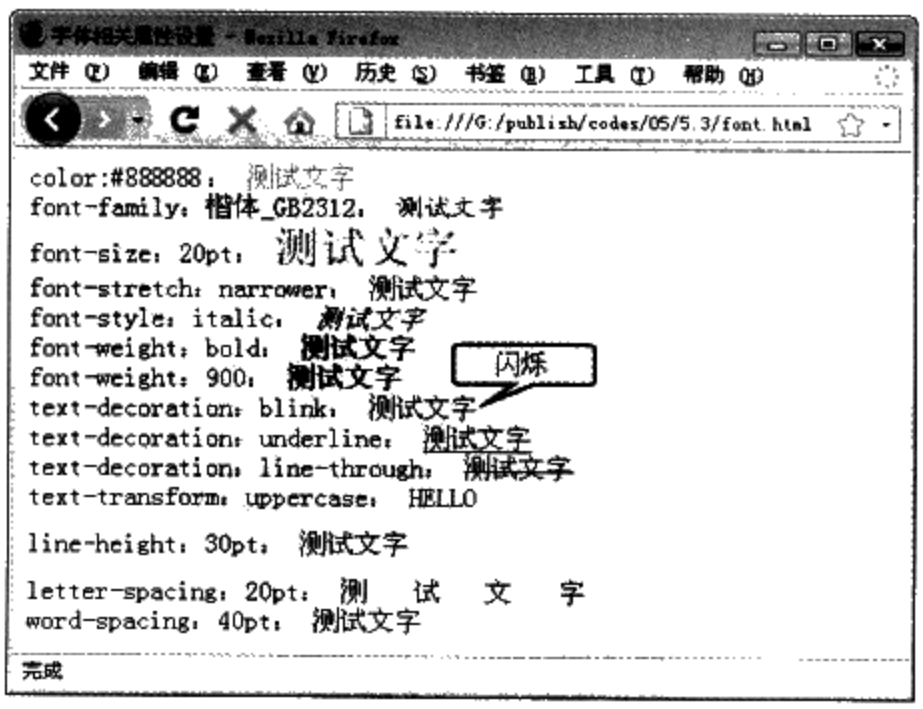


图 5.5 字体相关属性的设置效果

➤➤5.3.2 整体段落相关属性

整体段落相关属性用于控制整个段或整个<div.../>元素的显示效果，包括文字的缩进、段落内文字的对齐等显示方式：

- text-indent：适用于 CSS1。用于设置对应元素中的段落文本的缩进。默认值为 0 。被另一个元素（如<br.../>）断开的元素不能应用本属性。
- vertical-align：适用于 CSS1 和 CSS2。用于设置目标元素里内容的垂直对齐方式。通常有顶端对齐、底对齐等方式。
- text-align：适用于 CSS1，用于设置目标对象中文本的水平对齐方式。
- direction：适用于 CSS2，用于设置文本流入的方向，该属性的合法值有 ltr（从左向右）和 rtl（从右向左）。此属性不会影响拉丁字母、数字字符，它们总是以 ltr 值被呈现。但是此属性会作用于拉丁文的标点符号。
- white-space：适用于 CSS1，用于设置目标对象的空格的处理方式。

下面的代码演示了上面的常用 CSS 属性，为使读者能更清楚地看到段落属性的效果，将目标文本以方框包围起来，该页面的代码如下：

程序清单：codes\05\5.3\div.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>段落相关属性设置</title>
    <style type="text/css">
    /* 为 div 元素增加边框 */
    div{
        border:1px solid #000000;
        height: 30px;
        width: 200px;
    }
    </style>
</head>
<body>
<!-- 缩进 20pt -->
text-indent:20pt <div style="text-indent:20pt">测试文字</div>
<!-- 居中对齐 -->
text-align:center <div style="text-align:center">测试文字</div>
<!-- 居右对齐 -->
text-align:right <div style="text-align:right">测试文字</div>
<!-- 文本从右边流入 -->
direction:rtl <div style="direction:rtl">测试文字</div>
<!-- 文本从左边流入 -->
direction:ltr <div style="direction:ltr">测试文字</div>
<!-- 强制不换行，直到遇到 br 标签 -->
white-space:nowrap <div style="white-space:nowrap">
    测试文字,疯狂 Java 讲义，疯狂 XML 讲义</div>
</body>
</html>
```

该页面在 Firefox 中浏览的效果如图 5.6 所示。

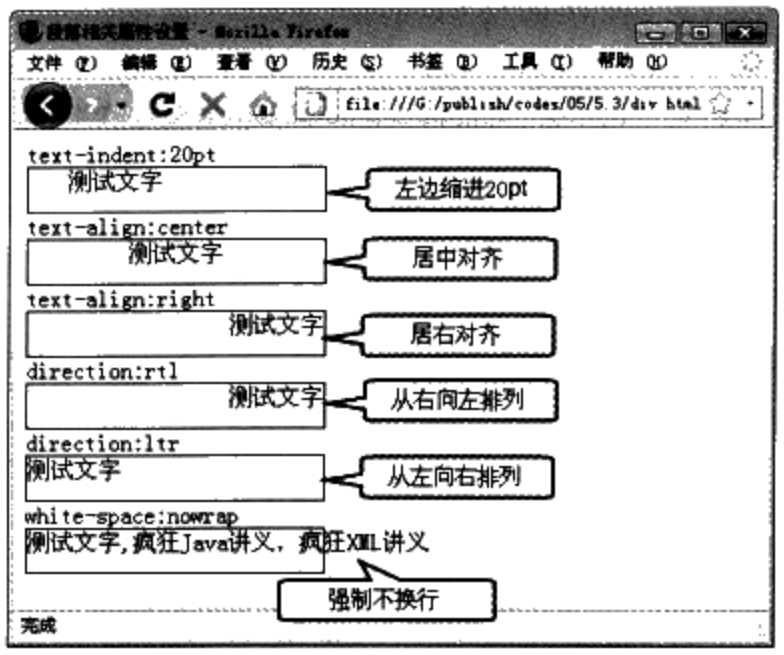


图 5.6 整体段落相关的属性测试

➤➤5.3.3 背景相关属性

背景相关属性用于控制背景色、背景图片等属性。在控制背景图片的同时，还可控制背景图片的排列方式。有如下几个常用的背景相关属性：

- background：适用于 CSS1，设置对象的背景样式。该属性是个复合属性，可用于同时设置背景色、背景图像等属性。
- background-attachment：适用于 CSS1，设置背景图像是随对象内容滚动，还是固定。指定该属性之前，必须先指定 background-image 属性。该属性有如下两个值：scroll、fixed，分别对应背景图像随对象内容滚动和背景图像固定。前者是默认值。
- background-color：适用于 CSS1，用于设置背景色。如果同时设置了背景色和背景图片，则背景图片将覆盖背景色。
- background-image：适用于 CSS1，用于设置背景图片。如果同时设置了背景色和背景图片，则背景图片将覆盖背景色。该属性需要使用 url()函数指定图片地址，图片地址既可以是相对地址，也可以是绝对地址。
- background-position：适用于 CSS1，用于设置对象的背景图像位置。如果只指定了一个值，该值将对应横坐标。纵坐标将默认为 50%。如果指定了两个值，第二个值将对应纵坐标。指定该属性之前，必须先指定 background-image 属性。
- background-repeat：适用于 CSS1，用于设置对象的背景图像是否使用平铺。指定该属性之前，必须先指定 background-image 属性。该属性有 repeat、no-repeat、repeat-x、repeat-y 四个值，分别对应在纵向和横向同时平铺、不平铺、仅在横向平铺、仅在纵向平铺。

下面的代码演示了这些背景相关属性的效果，为使读者更好地看到效果，页面将目标段落以方框包围起来。下面是该页面的测试代码：

程序清单：codes\05\5.3\bg.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>背景相关属性设置</title>
    <style type="text/css">
    /* 为 div 元素增加边框 */
    div{
```

```
            border:1px solid #000000;
            height: 50px;
            width: 200px;
        }
        </style>
    </head>
    <body>
    <!-- 灰色背景 -->
    background-color:#aaaaaa <div style="background-color:#aaaaaa">测试文字</div>
    <!-- 以默认样式指定背景图片，将平铺 -->
    background-image:url(logo.gif)<div style="background-image:url(logo.gif)">测试文字</div>
    <!-- 不平铺的背景图片 -->
    background-image:url(logo.gif);background-repeat: no-repeat
    <div style="background-image:url(logo.gif);background-repeat: no-repeat">测试文字</div>
    <!-- 横向平铺的背景图片 -->
    background-image:url(logo.gif);background-repeat: repeat-x
    <div style="background-image:url(logo.gif);
        background-repeat: repeat-x">测试文字</div>
    <!-- 不平铺的背景图片，并指定背景图片的位置 -->
    background-image:url(logo.gif);background-repeat:
        no-repeat;background-position: 35% 80%;
    <div style="background-image:url(logo.gif);background-repeat:
        no-repeat;background-position: 35% 80%; ">测试文字</div>
    </body>
    </html>
```

在浏览器中浏览该页面，可看到如图 5.7 所示效果。

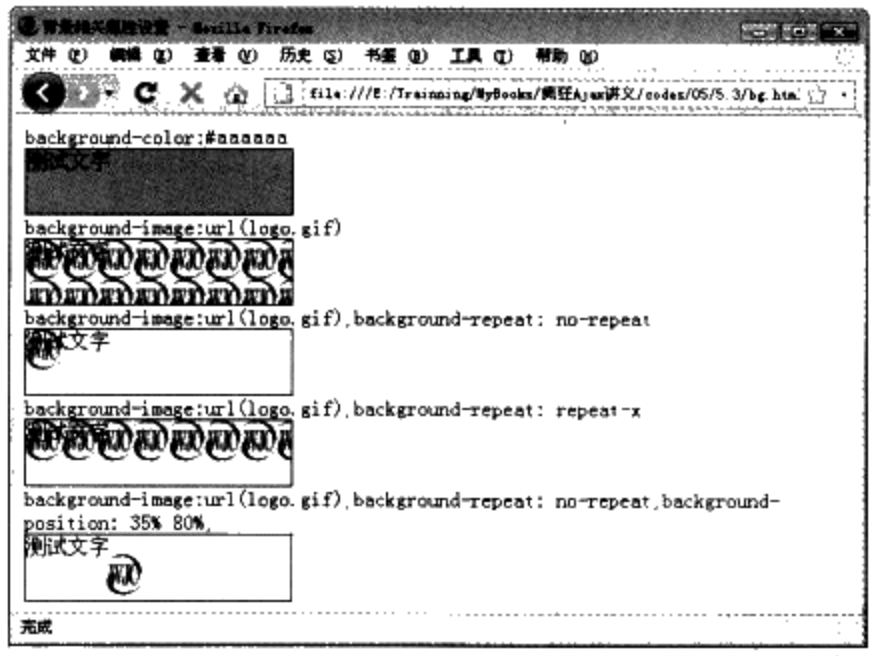


图 5.7　背景相关属性的 CSS 效果

➤➤5.3.4　表格相关属性

表格相关属性主要用于控制表格的外观表现，表格相关的常用属性有如下几个：

- border-collapse：适用于 CSS2，用于设置表格里行和单元格边框的显示方式，两个边框可以合并在一起，也可按照标准的 XHTML 样式分开。该属性有两个值：seperate、collapse，分别使得单元格的分隔线为双线、单线。但有时候效果也取决于浏览器。
- border-spacing：适用于 CSS2，用于设置当表格边框独立（即 border-collapse 属性等于 separate）时，行和单元格的边框在横向和纵向上的间距。该属性的值为一个距离值，在 Internet Explorer 中没有明显效果。
- caption-side：适用于 CSS2，用于设置表格的<caption.../>元素位于表格哪边。该属性必须和

<caption.../>元素（表格的标题）一起使用。该属性有四个值：top、bottom、left、right，分别对应于将表格标题放在表格的上、下、左、右四处。该属性在 Internet Explorer 6 中依然不被支持。

- empty-cells：适用于 CSS2，用于设置当表格的单元格无内容时，是否显示该单元格的边框。该属性有两个值：show 和 hide，分别对应显示表格边框和隐藏表格边框。
- padding：适用于 CSS1，用于设置单元格四边的空白。这个空白是个留空，不可能被文字填充，也不会被背景色填充。对于<td.../>和<th.../>对象，该属性的默认值为 1。
- padding-top：适用于 CSS1，用于设置单元格上方的空白。
- padding-right：适用于 CSS1，用于设置单元格右方的空白。
- padding-bottom：适用于 CSS1，用于设置目标单元格下方的空白。
- padding-left：适用于 CSS1，用于设置目标单元格左方的空白。

下面的页面中，包含了三个表格，分别用于测试上面的几个表格属性，代码如下：

程序清单：codes\05\5.3\table.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>表格相关属性测试</title>
    <style type="text/css">
    td {
        background-color:#cccccc;
    }
    </style>
</head>
<body>
<!-- 表格的单元格边框合并在一起，看起来分割线为单线，并将表格标题放在下边 -->
border-collapse:collapse;caption-side:bottom;
<table width="400" border="1"
    style="border-collapse:collapse;caption-side:bottom;">
    <caption>表格标题</caption>
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>轻量级 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 XML 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
</table>
<!-- 表格的单元格边框分开，看起来表格分割线为双线，并隐藏空格的边框线 -->
border-collapse:seperate;empty-cells: hide;
<table width="400" border="1"
    style="border-collapse:seperate;empty-cells: hide;">
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>轻量级 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 XML 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
</table>
<!-- 表格的单元格边框分开，IE 中看起来表格分割线为双线，并设置两个单元格的间距 -->
border-collapse:seperate;border-spacing:20px;
```

```
    <table width="400" border="1"
        style="border-collapse:seperate;border-spacing:20px">
        <tr>
            <td>疯狂 Java 讲义</td>
            <td>轻量级 Java EE 企业应用实战</td>
        </tr>
        <tr>
            <td>疯狂 XML 讲义</td>
            <td>经典 Java EE 企业应用实战</td>
        </tr>
    </table>
    </body>
    </html>
```

上面的代码为了更好地显示出效果，为所有单元格设置了灰色背景。上面的页面在不同浏览器中的效果并不完全相同，下面分别给出在 Internet Explorer 和 Firefox 中浏览该页面时看到的效果，如图 5.8 所示是该页面在 Internet Explorer 中浏览时的效果。

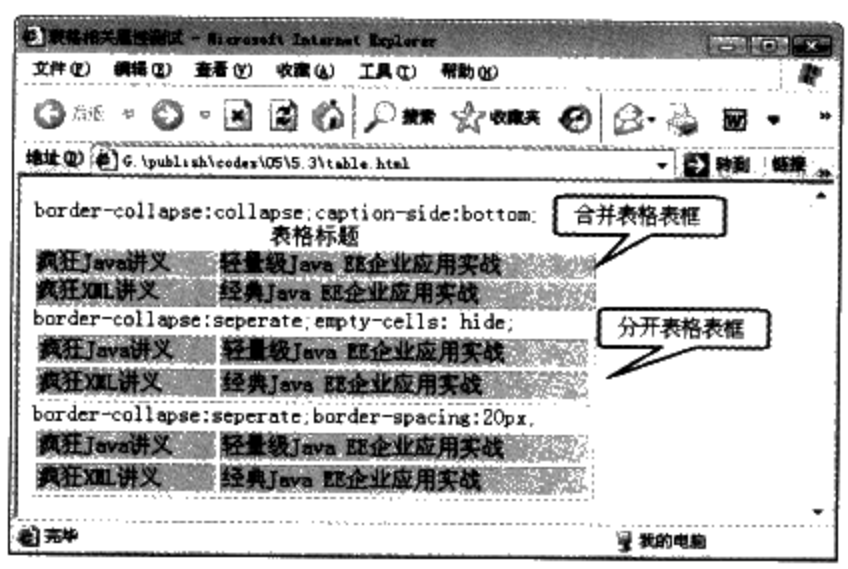


图 5.8　表格属性在 Internet Explorer 中的显示效果

在 Internet Explorer 中可看到表格边框分开后的效果，但无法看到设置 border-spacing 属性的效果，即使代码中设置了该属性值为 20px，但图中看不出任何效果。而且虽然我们设置了标题应位于表格之下，但 Internet Explorer 不能显示出该效果，标题依然在表格的上方。图 5.9 显示了在 Firefox 中浏览该页面时的效果。

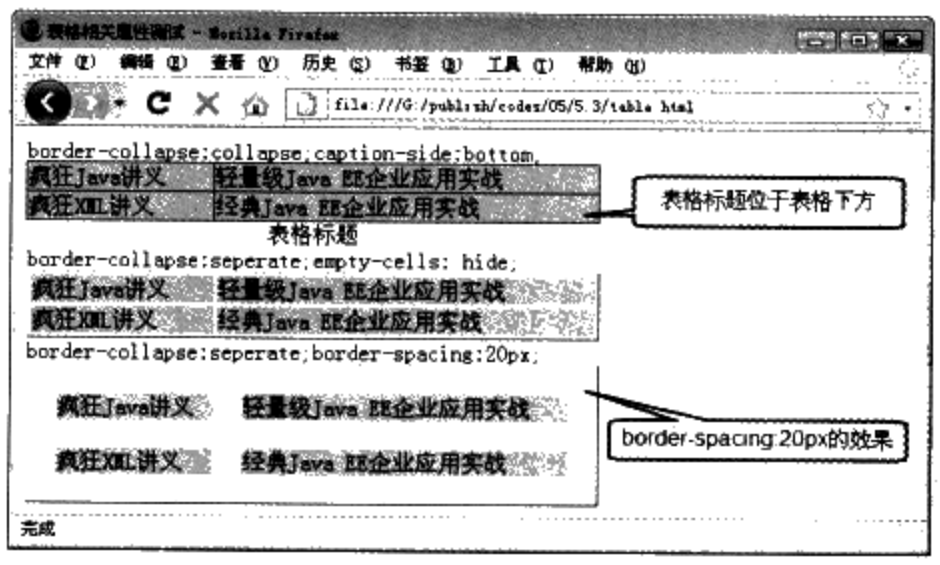


图 5.9　表格属性在 Firefox 中的显示效果

在 Firefox 中看到的表格测试效果比较明显，表格标题被放在表格的下方。border-spacing 属性值也确实产生了效果：两个表格边框之间增加了 20px 的间距。

5.3.5 大小相关属性

大小相关属性主要用于设置目标对象的长、宽。包括最大长度、宽度，以及最小长度、宽度。大小相关的常用属性有如下几个：

- height：适用于 CSS1，用于设置目标对象的高度。该属性值可以是任何有效的距离值。
- max-height：适用于 CSS2，用于设置目标对象的最大高度。如果此属性的值小于 min-height 属性值，将会被自动转换为 min-height 属性的值。该属性值可以是任何有效的距离值。
- min-height：适用于 CSS2，用于设置目标对象的最小高度。如果此属性的值大于 max-height 属性值，将会被自动转换为 max-height 属性的值。该属性值可以是任何有效的距离值。
- width：适用于 CSS1，用于设置目标对象的宽度。该属性值可以是任何有效的距离值。
- max-width：适用于 CSS2，用于设置目标对象的最大宽度。如果此属性的值小于 min-width 属性值，将会被自动转换为 min-width 属性的值。该属性值可以是任何有效的距离值。
- min-width：适用于 CSS2，用于设置目标对象的最小宽度。如果此属性的值大于 max-width 属性的值，将会被自动转换为 max-width 属性的值。该属性值可以是任何有效的距离值。

下面的页面使用了长、宽属性来控制页面中一个<div.../>元素的大小。

程序清单：codes\05\5.3\dimension.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>大小相关属性测试</title>
</head>
<body>
    <!-- 下面使用内联的 CSS 样式控制大小，
        为了得到更好的显示效果，并设置了其背景色 -->
    <div style="width:200px;height:40px;background-color:#dddddd">
    Java 学习
    </div>
</body>
</html>
```

该页面浏览的效果如图 5.10 所示。

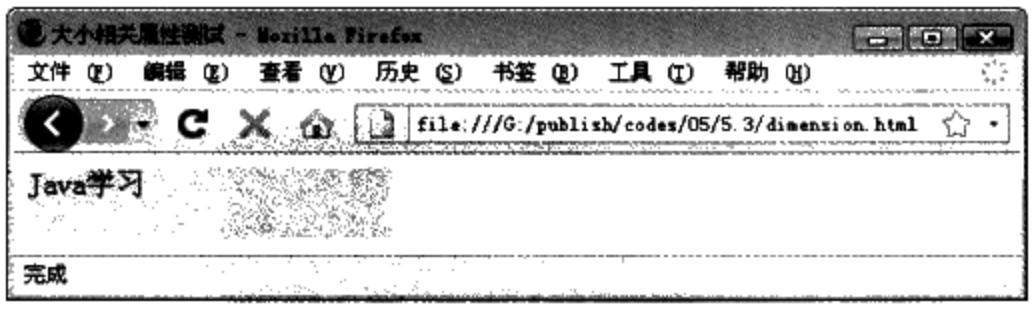


图 5.10 设置<div.../>元素的大小

5.3.6 位置相关属性

定位相关属性用于设置目标对象的位置，包括是否漂浮于页面之上，通过使用漂浮的<div.../>元素，可自由移动页面元素的位置，从而可在页面上产生动画效果。定位相关的常用属性如下：

- position：适用于 CSS2，用于设置目标对象的定位方式。设置此属性值为 absolute 可允许将该对象漂浮于页面之上，根本无须考虑它周围内容的布局。设置此属性值为 relative 会保持对象在正常的 HTML 流中，目标对象的位置将以前一个对象的位置为基准相对位移。如果该

属性被设置为 static 值，则目标对象仅以页面作为参照系。

- z-index：适用于 CSS2，用于设置目标对象的漂浮层，该值越大，漂浮层越处于上面。此属性仅当 position 属性值为 relative 或 absolute 时有效。此属性对窗口控件（如 <select.../> 元素）没有影响。
- top：适用于 CSS2，用于设置目标对象相对于最近一个具有定位设置的父对象的顶边偏移，此属性仅当对象的 position 属性为 absolute 或 relative 时有效。
- right：适用于 CSS2，用于设置目标对象相对于最近一个具有定位设置的父对象的右边偏移，此属性仅当对设置了对象的 position 属性为 absolute 或 relative 时有效。
- bottom：适用于 CSS2，用于设置目标对象相对于最近一个具有定位设置的父对象的底边偏移，此属性仅当对设置了对象的 position 属性为 absolute 或 relative 时有效。
- left：适用于 CSS2，用于设置目标对象相对于最近一个具有定位设置的父对象的左边偏移，此属性仅当对设置了对象的 position 属性为 absolute 或 relative 时有效。

下面的页面代码提供了 5 个<div.../>元素，分别用于测试上面的各种属性：

程序清单：codes\05\5.3\position.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>位置相关属性测试</title>
</head>
<body>
Struts<br />
Hibernate<br />
Spring<br />
jBPM<br />
<!-- 下面的<div.../>元素定位时使用了 absolute 值，因此不会受上面文本的影响
    它将直接基于页面定位-->
<div id="Layer1" style="position:absolute;
    left:40px; top:20px; width:180px; height:88px;
    z-index:2; background-color: #c1c1c1;">
Layer1,使用 positon 属性值为 absolute,该 Layer 将完全漂浮在页面之上，
不受其他对象位置影响。z-index:2
</div>
<!-- 下面的<div.../>元素定位时使用了 relative 值，因此会受上面文本的影响
    它将基于页面文本定位 -->
<div id="Layer2" style="position:relative;
    left:50px; top:10px; width:200px; height:88px;
    z-index:3; background-color: #999999;">
Layer2,使用 positon 属性值为 relative,该 Layer 将漂浮在页面之上，
但会受其他对象位置影响。z-index:3
</div>
<!-- 下面的 Layer3 和 Layer4 两个<div.../>虽然设置了 top 一个为 40px，另一个为 80px。
    但不会有任何作用,因为其 position 为 static -->
<div style="position:absolute; left:260px; top:80px; width:250px;
    height:200px; border:black solid 1px">
    <div id="Layer3" style="position:static; left:100px; top:40px;
        width:80px; height:88px; z-index:1; background-color: #666666;">
        position:static
    </div>
    <div id="Layer4" style="position:static; left:100px; top:80px;
        width:80px; height:88px; z-index:1; background-color: #999999;">
        position:static
```

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

在浏览器中浏览该页面，可看到如图 5.11 所示效果。

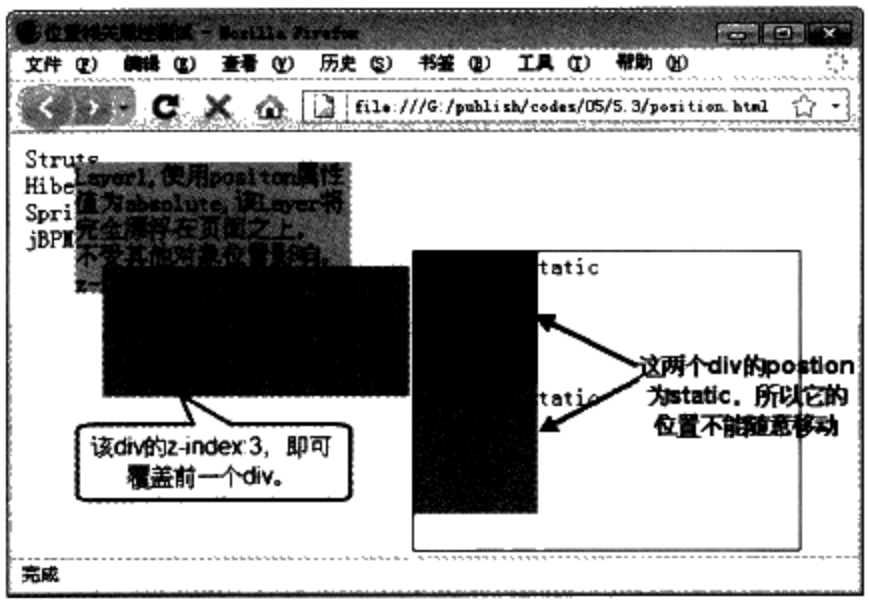


图 5.11　定位相关属性的效果

上面的页面中 postion 属性为 absolute 的<div.../>将完全不受页面其他元素的影响，直接基于页面定位。如果 postion 属性为 relative，则<div.../>将受到其他元素的影响，会基于页面中的文本元素定位。如果设置 postion 属性为 static，则 left、top 等定位属性失效。

➤➤5.3.7　边框相关属性

边框属性用于设置目标对象的边框特征，包括边框的颜色、粗细，以及使用的线型。边框相关属性有：

- border：适用于 CSS1，这是一个复合属性。用于设置对象的边框样式。可同时设置边框的粗细、线型和颜色。
- border-color：适用于 CSS1，用于设置对象边框的颜色。如果提供 4 个参数值，将按上、右、下、左的顺序依次设置四个边框的颜色。如果只提供 1 个，将用于设置四个边框的颜色。如果提供 2 个，则第一个用于设置上、下两个边框的颜色，第二个用于设置左、右两个边框的颜色。如果提供 3 个，则第一个用于设置上边框的颜色，第二个用于设置左、右两个边框的颜色，第三个用于设置下边框的颜色。
- border-style：适用于 CSS1，用于设置对象边框的线型。如果提供 4 个参数值，将按上、右、下、左的顺序依次设置四个边框的线型。如果只提供 1 个，将用于设置四个边框的线型。如果提供 2 个，则第一个用于设置上、下两个边框的线型，第二个用于设置左、右两个边框的线型。如果提供 3 个，则第一个用于设置上边框的线型，第二个用于设置左、右两个边框的线型，第三个用于设置下边框的线型。
- border-width：适用于 CSS1，用于设置对象边框的线宽。如果提供 4 个参数值，将按上、右、下、左的顺序依次设置四个边框的线宽。如果只提供 1 个，将用于设置四个边框的线宽。如果提供 2 个，则第一个用于设置上、下两个边框的线宽，第二个用于设置左、右两个边框的线宽。如果提供 3 个，则第一个用于设置上边框的线宽，第二个用于设置左、右两个边框的线宽，第三个用于设置下边框的线宽。
- border-top：适用于 CSS1，这是一个复合属性。用于设置对象上边框的样式。可同时设置边框的粗细、线型、颜色。

- border-top-color：适用于 CSS2，用于设置目标对象上边框的颜色。
- border-top-style：适用于 CSS2，用于设置目标对象上边框的线型。
- border-top-width：适用于 CSS1，用于设置目标对象上边框的线宽。
- border-right：适用于 CSS1，这是一个复合属性。用于设置对象右边框的样式。可同时设置边框的粗细、线型、颜色。
- border-right-color：适用于 CSS2，用于设置目标对象右边框的颜色。
- border-right-style：适用于 CSS2，用于设置目标对象右边框的线型。
- border-right-width：适用于 CSS1，用于设置目标对象右边框的线宽。
- border-bottom：适用于 CSS1，这是一个复合属性。用于设置对象下边框的样式。可同时设置边框的粗细、线型、颜色。
- border-bottom-color：适用于 CSS2，用于设置目标对象下边框的颜色。
- border-bottom-style：适用于 CSS2，用于设置目标对象下边框的线型。
- border-bottom-width：适用于 CSS1，用于设置目标对象下边框的线宽。
- border-left：适用于 CSS1，这是一个复合属性。用于设置对象左边框的样式。可同时设置边框的粗细、线型、颜色。
- border-left-color：适用于 CSS2，用于设置目标对象左边框的颜色。
- border-left-style：适用于 CSS2，用于设置目标对象左边框的线型。
- border-left-width：适用于 CSS1，用于设置目标对象左边框的线宽。

上面的边框属性中，边框颜色可以是任何有效的颜色值，而线宽可以是任何有效的长度值。线型可以是 none、hidden、dotted、dashed、solid、double、groove、ridge、inset、outset，分别对应于无边框、隐藏边框、点线边框、虚线边框、实线边框、双线边框、3D 凹槽边框、菱形边框、3D 凹边、3D 凸边等线型。虽然提供了这么多线型，但不是每个浏览器中都可看出全部线型的效果。

下面的页面代码分别使用了不同的边框效果，用于演示线型、线宽、颜色等属性控制边框后的效果，为了更好地看出边框效果，页面还用 CSS 设置了<div.../>元素的大小。

程序清单：codes\05\5.3\border.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>边框相关属性测试</title>
    <style type="text/css">
        /* 设置 div 元素的宽度和高度 */
        div {
            width:300px;
            height:40px;
        }
    </style>
</head>
<body>
<div style="border-width:1px; border-style:groove; border-color:#666666">
宽度为 1 的灰色凹槽边框
</div><br />
<div style="border-width:1px; border-style:dotted; border-color:#666666">
宽度为 1 的灰色点线边框
</div><br />
<div style="border-width:1px; border-style:dashed; border-color:#666666">
宽度为 1 的灰色虚线边框
</div><br />
```

```
<div style="border-width:5px; border-style:solid; border-color:#666666">
宽度为 5 的灰色实线边框
</div><br />
<div style="border-width:3px;border-style:solid;
    border-color:#cccccc #cccccc #444444 #444444;">
让四条边框颜色不同做出的立体效果
</div>
</body>
</html>
```

在浏览器中浏览该页面，将可以看到如图 5.12 所示效果。

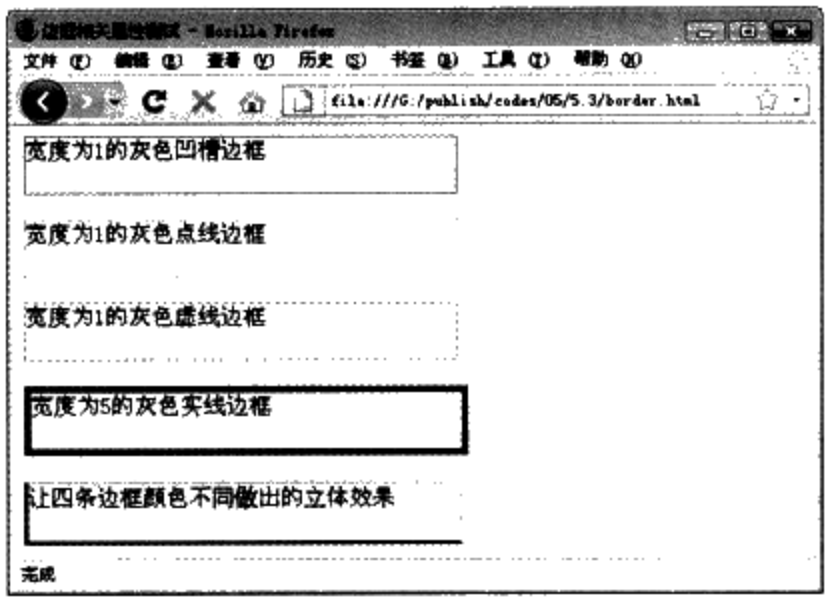


图 5.12　边框属性的效果

➤➤5.3.8　轮廓相关属性

轮廓相关属性主要用于让目标对象周围产生一圈“光晕”，这圈光晕不会占用页面实际的物理布局。通过轮廓相关属性，可设置该“光晕”的颜色、线宽、线型等属性。轮廓相关的属性有：

- outline：适用于 CSS2，这是一个复合属性。可全面设置目标对象轮廓的颜色、线宽、线型等属性。
- outline-color：适用于 CSS2：用于设置目标对象轮廓的颜色。
- outline-style：适用于 CSS2：用于设置目标对象轮廓的线型。
- outline-width：适用于 CSS2：用于设置目标对象轮廓的线宽。

将 5.3.7 中的实例代码稍作修改，修改页面中的 CSS 样式，将<style.../>中的 CSS 样式修改为如下形式：

程序清单：codes\05\5.3\outline.html

```
/* 设置 div 元素的宽度和高度，及其轮廓样式 */
div {
    width:300px;
    height:40px;
    outline-color:#c2c2c2;
    outline-style:solid;
    outline-width:10px;
}
```

在页面中定义了这些轮廓样式之后，在浏览器中浏览该页面，将可以看到如图 5.13 所示效果。

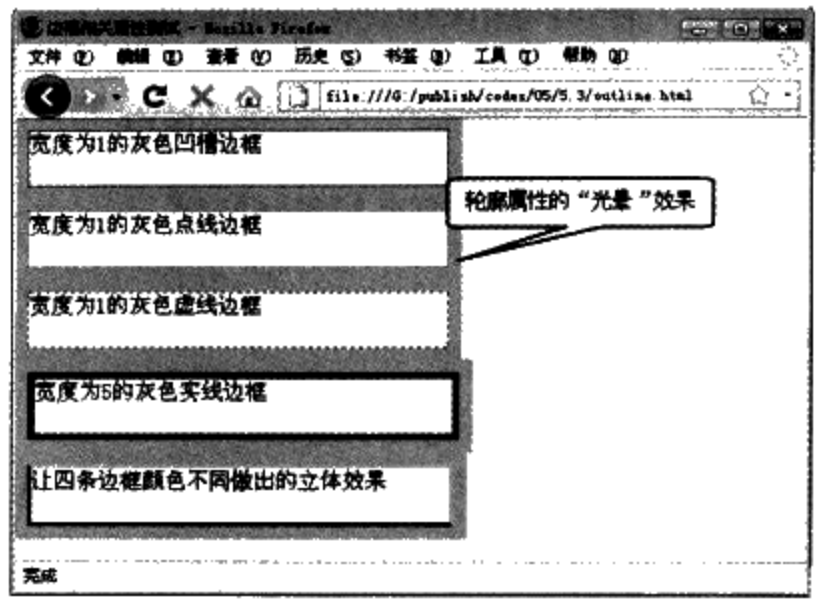


图 5.13　轮廓属性的效果

➤➤5.3.9　三个常用属性

下面介绍的三个常用属性，可用于设置目标对象的光标以及目标对象的显示与隐藏。这三个属性如下：

- display：适用于 CSS1 和 CSS2，用于设置目标对象是否及如何显示。该属性的常用值为 none，用于设置目标对象隐藏，一旦该对象隐藏，其占用的页面空间也会释放。如果没有为该属性指定值，则目标对象会显示出来。
- visibility：适用于 CSS2，用于设置目标对象是否显示。与 display 属性不同的是，通过该属性隐藏某个 XHTML 元素后，该元素占用的页面空间依然会被保留。该属性的两个常用值为 visible 和 hidden，分别用于控制目标对象的显示和隐藏。
- cursor：适用于 CSS2，用于设置目标对象上光标的形状。此属性的值可以是多个，多个值用英文逗号（,）分隔。该属性的常用值有 auto、crosshair、default、hand、move、help、text、wait、url (url)等，分别对应于自动光标、十字线光标、客户端光标、手形光标、十字箭头光标、帮助光标（带问号的光标）、文本光标、沙漏光标和自定义光标等。

看下面的页面代码：

程序清单：codes\05\5.3\other.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>其他属性测试</title>
    <style type="text/css">
    /* 设置 div 元素的宽度、高度、背景色和边框 */
    div{
        width:300px;
        height:40px;
        background-color:#dddddd;
        border:2px solid black;
    }
    </style>
</head>
<body>
<input type="button" value="隐藏"
    onclick="document.getElementById('test1').style.display='none'"/>
<input type="button" value="显示"
```

```
    onclick="document.getElementById('test1').style.display=''"/>
<div id = "test1">
使用 display 控制对象的显示和隐藏
</div>
<input type="button" value="隐藏"
    onclick="document.getElementById('test2').style.visibility ='hidden'"/>
<input type="button" value="显示"
    onclick="document.getElementById('test2').style.visibility ='visible'"/>
<div id = "test2">
使用 visibility 控制对象的显示和隐藏
</div>
<div style="cursor:wait;">
沙漏光标的效果
</div>
</body>
</html>
```

上面的页面中粗体字代码使用了 document.getElementyById("test1")，该方法属于 DOM 操作的一个方法，用于获取具有指定 id 属性值的 XHTML 元素。不仅如此，页面中四个按钮还指定了 onclick 属性，该属性值为一系列 JavaScript 语句，当用户单击这些按钮时，onclick 属性指定的一系列 JavaScript 脚本会获得执行。

在浏览器中浏览该页面，可看到如图 5.14 所示的效果。

在图 5.14 中看到，使用了 cursor:wait 属性后，当光标移动到目标对象上时，光标即变成沙漏形状。然后依次单击页面中两个“隐藏”按钮，将看到如图 5.15 所示的效果。

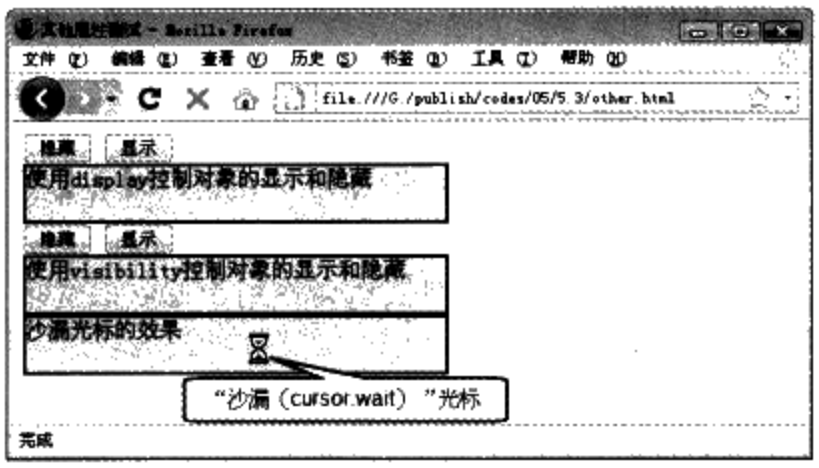


图 5.14　沙漏光标的效果

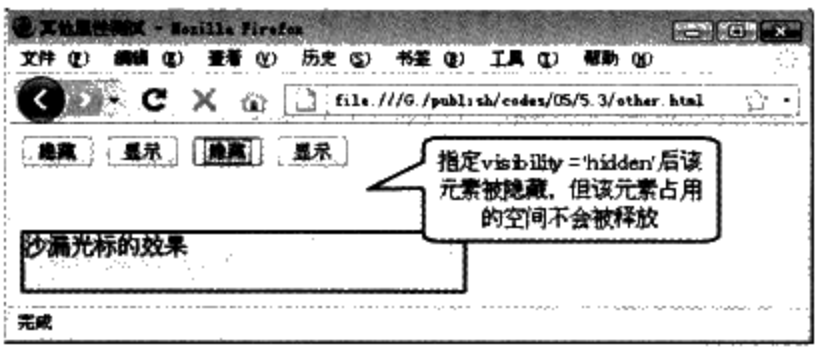


图 5.15　两种隐藏方式的效果

在图 5.15 中看到，对于使用 display 属性控制显示、隐藏的目标对象，一旦对象被隐藏，后面的按钮立即占据了它原来的位置。而使用 visibility 控制显示、隐藏的目标对象即使被隐藏，它所占据的页面空间依然被保留（沙漏光标效果所在的<div.../>并未上移）。

当然，本节并没有完全介绍 CSS 2 的全部样式。笔者所介绍的，都是 Ajax 应用中经常需要使用的 CSS 样式。至于一些特殊的效果，例如专属于某个浏览器的 CSS 样式，还有一些 CSS 滤镜样式，笔者并未全部介绍。本书所介绍的 CSS 样式是为本书的主题 Ajax 应用服务的，如果读者需要深入了解 CSS 样式，则应该登录 http://www.w3.org 站点查看 CSS 的相关规范。

5.4　选择器定义

上面介绍了 CSS 的各种基本属性，这些 CSS 属性主要用于控制 XHTML 元素如何显示。除此之外，还必须控制 CSS 样式应该应用到哪些 XHTML 元素。而 CSS 样式之所以能应用于 XHTML 文档的元素，都是因为选择器的作用，也就是选择器控制 CSS 样式对哪些 XHTML 元素起作用，而“属性名:属性值”则控制对 XHTML 元素起怎样的作用。

通常情况下，定义一个 CSS 样式时，总是按照如下格式：

```
Selector {property : value; ..... }
```

定义一个 CSS 样式时，大部分情况下只需指定一个 Selector（选择器），表示对符合该选择器规则的 XHTML 元素起作用。但也可能出现指定多个选择器的情形，如果有多个选择器，则该 CSS 样式可对多批 XHTML 元素起作用。同时指定多个选择器的语法格式如下：

```
Selector,Selector... {property : value; ..... }
```

也就是将多个选择器以英文逗号（,）隔开即可。

➤➤5.4.1 属性选择器

属性选择器是一种常用的 CSS 选择器，在前面的介绍中，如果有如下的 CSS 样式定义：

```
td {.....}
```

上面的 CSS 样式定义将会对页面中所有的<td.../>元素起作用，这就是属性选择器的一种形式。属性选择器一共有如下几种语法格式：

- Tag {..... }：指定该 CSS 样式对所有 Tag 元素起作用。
- Tag[attr] {..... }：指定该 CSS 样式对具有 attr 属性的 Tag 元素起作用。
- Tag[attr = value] { ..... }：指定该 CSS 样式对所有包含 attr 属性，且 attr 属性为 value 的 Tag 元素起作用。
- Tag [attr ~= value] { ..... }：指定该 CSS 样式对所有包含 attr 属性，且 attr 属性的值为以空格隔开的系列值，其中某个值为 value 的 Tag 元素其作用。
- Tag [attr |= value] { ..... }：指定该 CSS 样式对所有包含 attr 属性，且 attr 属性的值为以连字符分隔的系列值，且第一个值为 value 的 Tag 元素起作用。

上面这几个选择器作用的优先顺序是依次升高的，即对于包含 abc 属性的<div.../>元素，如果其属性值为 xyz，则以 div[abc=xyx]选择器定义的 CSS 样式会覆盖 div[abc]定义的 CSS 样式，而 div[abc=xyz]选择器定义的 CSS 样式中没有定义的属性，div[abc]选择器定义的 CSS 属性依然会作用于 abc 属性值为 xyz 的<div.../>元素。

注意：

上面这几种属性选择器并没有得到所有浏览器的广泛支持，只有第一种形式可以在所有浏览器中运行良好，而且最后 2 种 CSS 选择器在很多浏览器中都得不到很好的支持。如果需要让该 CSS 在所有浏览器中运行良好，建议仅使用第一种形式。

当然，这些属性选择器也并不是所有的浏览器都支持的，例如 Internet Explorer 6.0 将无法支持下述页面中包含的所有选择器。看下面的页面示例：

程序清单：codes\05\5.4\propertySelector1.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <title>属性选择器测试</title>
    <style type="text/css">
    /* 对所有 div 元素都起作用的 CSS 定义 */
    div {
        width:200px;
        height:30px;
        background-color:#dddddd;
```

```
        border:1px solid black;
    }
    /* 对有 id 属性的 div 元素起作用的 CSS 定义 */
    div[id] {
        background-color:#aaaaaa;
    }
    /* 对 id 属性值为 xx 的 div 元素起作用的 CSS 定义 */
    div[id=xx] {
        background-color:#888888;
    }
    </style>
</head>
<body>
<div>没有任何属性的 div 元素</div>
<div id="a">带 id 属性的 div 元素</div>
<div id="xx">id 属性值为 xx 的 div 元素</div>
</body>
</html>
```

上面的页面在 Firefox 中浏览将出现如图 5.16 所示的页面。

提 示:

在 Internet Explorer 6 中看不到该效果，因为 Internet Explorer 6 只支持第一种属性选择器。

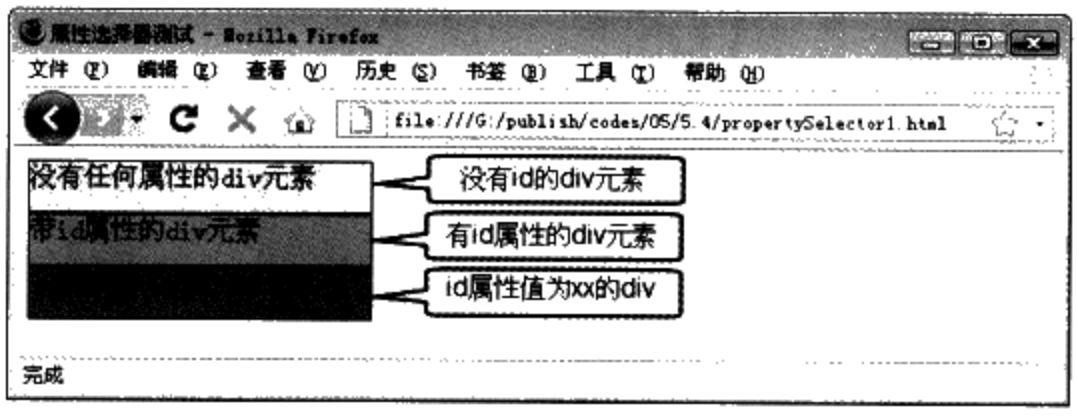


图 5.16 属性选择器的 CSS 样式

在图 5.16 中可看到，虽然 div[id=xx]选择器定义的 CSS 样式并没有定义其长、宽，但<div id="xx".../>元素依然具有指定高度、宽度，这表明该<div.../>元素的显示外观是多个 CSS 样式“叠加”作用的效果。

注意：

当多个 CSS 样式定义都可以对某个 XHTML 元素起作用时，该 XHTML 元素的显示外观将是多个 CSS 样式定义“叠加”作用的效果。如果多个 CSS 样式定义之间有冲突，则冲突属性以优先级更高的 CSS 样式定义取胜。

5.4.2 ID 选择器

这是一个精确控制的选择器，指定 CSS 样式定义将会作用于 XHTML 页面中具有指定 ID 的元素。ID 选择器的语法格式如下：

#idValue { ..... }：指定该 CSS 样式对 id 为 idValue 的 XHTML 元素起作用。

ID 选择器在 Internet Explorer 和 Firefox 中都有很好的支持。看下面的页面代码：

程序清单：codes\05\5.4\idSelector.html

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

```
<title>ID选择器测试</title>
    <style type="text/css">
    /* 对所有 div 元素都起作用的 CSS 定义 */
    div {
        width:200px;
        height:30px;
        background-color:#dddddd;
    }
    /* 对 id 为 xx 的元素起作用的 CSS 定义 */
    #xx {
        border:2px dotted black;
        background-color:#888888;
    }
    </style>
</head>
<body>
<div>没有任何属性的 div 元素</div>
<div id="xx">id 属性值为 xx 的 div 元素</div>
</body>
</html>
```

该页面在 Internet Explorer 中浏览的效果如图 5.17 所示。

图 5.17　ID 选择器的 CSS 样式效果

➤➤5.4.3　class 选择器

class 选择器指定 CSS 样式对具有指定 class 属性的元素起作用，class 选择器的语法格式如下：

Tag.classValue { ..... }：指定该 CSS 定义对 class 属性值为 classValue 的 Tag 元素起作用。此处的 Tag 可以省略，如果省略 Tag，则该 CSS 对所有的 class 属性为 classValue 的元素都起作用。

提　示：

为了让 XHTML 页面支持 class 选择器，W3C 组织规定几乎所有 XHTML 元素都可指定 class 属性，该属性唯一的作用正是让 class 选择器起作用。

下面的页面定义了 2 个 class 选择器，页面代码如下：

程序清单：codes\05\5.4\classSelector.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>class 选择器测试</title>
    <style type="text/css">
    /* 对所有 class 为 myclass 的元素都起作用的 CSS 定义 */
    .myclass {
        width:240px;
        height:40px;
```

```
        background-color:#dddddd;
    }
    /* 对 class 为 myclass 的 div 元素起作用的 CSS 定义 */
    div.myclass {
        border:2px dotted black;
        background-color:#888888;
    }
    </style>
</head>
<body>
<div class="myclass">class 属性为 myclass 的 div 元素</div>
<p class="myclass">class 属性为 myclass 的 span 元素</p>
</body>
</html>
```

在浏览器中浏览该页面，可看到如图 5.18 所示效果。

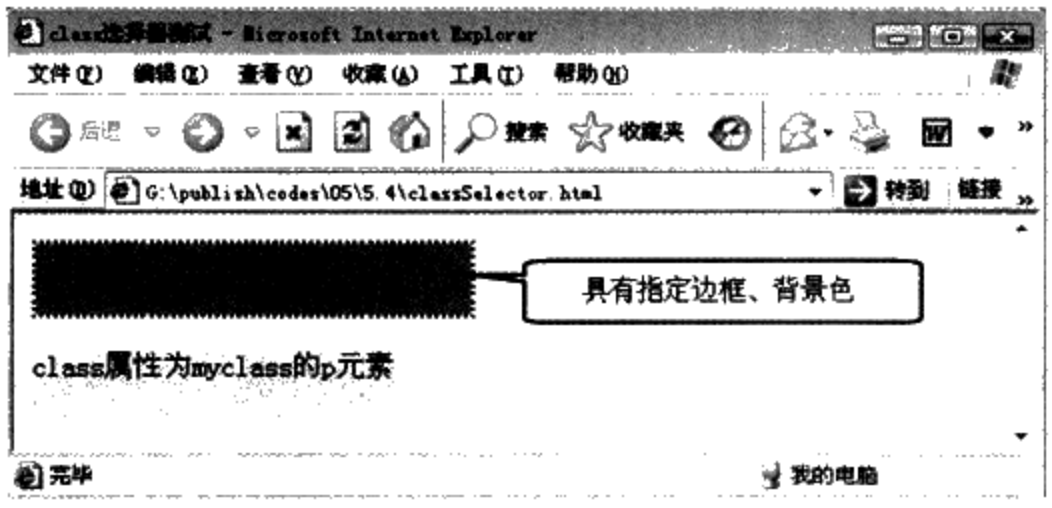


图 5.18　class 选择器的效果

正如在图 5.18 中所看到的，上面的页面中定义的 2 个 CSS 样式都可作用于<div.../>元素，因此该<div.../>元素的显示效果是两个 CSS 样式“叠加”的效果。从图 5.18 中可以看出，既指定标签又指定 class 值的选择器的优先级更高。

➤➤5.4.4　包含选择器和子元素选择器

包含选择器和子元素选择器都用于为目标元素指定一个父元素。但它们之间存在如下区别：对于包含选择器，只要目标元素位于父元素之内，即使是其“孙子元素”也可；对于子元素选择器，要求目标元素必须作为父元素的直接子元素才可。

包含选择器的语法格式如下：

- OuterTag InnerTag{ ..... }：只要 InnerTag 处于 OuterTag 之内，不管 InnerTag 处于 OuterTag 的几层之下，InnerTag 都可应用 CSS 样式。

包含选择器还有一种用法：第一个元素不是 XHTML 元素，只是一个选择器。此时将匹配位于外部选择器内部的嵌套选择器。

子元素选择器的语法格式如下：

- FatherTag>SonTag{ ..... }：SonTag 必须作为 FatherTag 的直接子元素，才可应用 CSS 样式。

看下面的页面代码：

程序清单：codes\05\5.4\childSelector.html

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

```
    <title>子元素选择器测试</title>
    <style type="text/css">
    /* 对所有的 div 元素起作用的 CSS 定义 */
    div {
        width:250px;
        height:35px;
        background-color:#dddddd;
    }
    /* 对处于 div 之内的 div 元素起作用的 CSS 定义 */
    div > div {
        width:200px;
        height:30px;
        border:2px dotted black;
        background-color:#999999;
    }
    </style>
</head>
<body>
<div>没有任何属性的 div 元素</div><br />
<div><div>处于 div 之内的 div 子元素元素</div></div>
</body>
</html>
```

上面的 div>div 选择器定义的 CSS 样式，应该对处于<div.../>元素之内的<div.../>元素起作用。图 5.19 显示了该页面的效果。

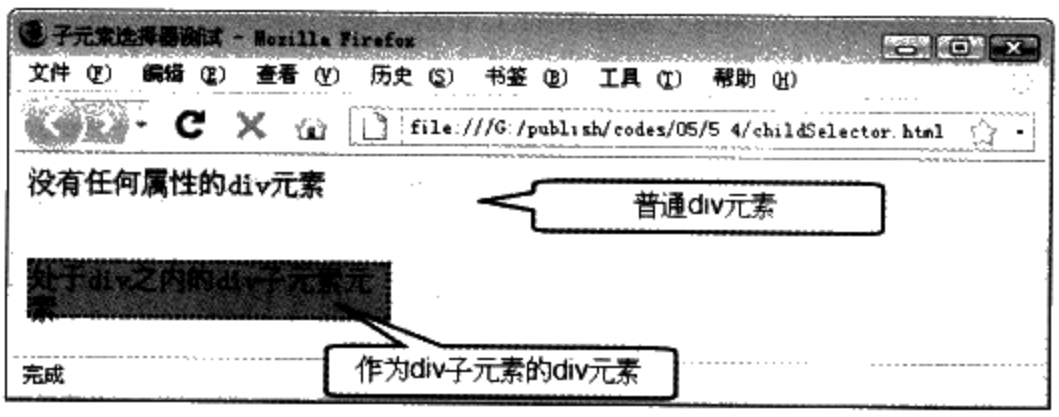


图 5.19 子元素选择器定义的 CSS 样式

➤➤5.4.5 超级链接相关选择器

超级链接相关选择器用于定义超级链接的样式。超级链接的默认样式都是蓝色，并加有下画线。通过 CSS 样式的设置可改变这种效果。下面是关于超级链接的四个选择器：

- a:link：设置超级链接在未被访问过时的 CSS 样式。
- a:hover：设置当鼠标悬停在超级链接上时的 CSS 样式。
- a:active：设置超级链接被用户激活（在鼠标点击与释放之间的事件）时的 CSS 样式。
- a:visited：设置已经访问过的超级链接的 CSS 样式。

这种超级链接相关的选择器在后面的示例中也有，此处不再给出具体的例子。

5.5 在脚本中修改显示样式

对于 Ajax 应用而言，更多的是需要在脚本中动态控制页面的显示效果。因此，Ajax 应用经常需要使用脚本动态地修改 XHTML 元素的 CSS 样式。

使用脚本动态设置 CSS 样式也非常简单，按如下步骤就可以动态修改目标元素的 CSS 样式：

（1）获取到需要设置 CSS 样式的目标元素，例如可用 getElementById()方法。

（2）修改目标元素的 CSS 样式。常有的方式有两种：

- 修改内联 CSS 属性值：使用如“obj.style.属性名=属性值”的 JavaScript 脚本即可。
- 修改 XHTML 元素的 class 属性值：使用如 “obj.className=class 选择器”的 JavaScript 脚本即可。

值得注意的是，脚本中的 CSS 属性名与页面中的静态 CSS 属性名并不完全相同。例如页面中静态 CSS 属性名为 color，脚本中该属性名还是 color；但静态 CSS 属性名为 background-color，脚本中该属性的属性名为 backgroundColor，相信学习 Java 的读者对这种命名方式相当熟悉：脚本中的 CSS 属性名是去掉原静态 CSS 属性名中的中画线（-），并将第一个单词的首字母小写，后面每个单词的首字母大写。如果静态 CSS 属性名没有包含中画线（-），则脚本中的 CSS 属性名与静态 CSS 属性名相同。

修改 XHTML 元素的 class 属性值应通过设置该元素的 className 属性完成，合法的 className 属性值是一个 class 选择器。

使用脚本修改目标对象的 CSS 样式值在 5.3.9 节中已经见到了范例，下面笔者将以几个简单的示范来结束本章的内容。

➤➤5.5.1 随机改变页面的背景色

改变页面的背景色是非常简单的事情，只要生成一个随机的 6 位数，并将该值赋给 body 元素的 CSS 属性 backgroundColor 即可。下面是随机改变背景色的页面代码：

程序清单：codes\05\5.5\randomBg.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>随机改变页面背景色</title>
    <script type="text/javascript">
    function changeBg()
    {
        //将背景色的值定义成空字符串
        var bgColor="";
        //循环 6 次，生成一个随机的六位数
        for (var i = 0 ; i < 6 ; i++)
        {
            bgColor += "" + Math.round(Math.random() * 9);
        }
        //将随机生成的背景颜色值赋给页面的背景色
        document.getElementById("test")
            .style.backgroundColor="#" + bgColor;
    }
    //为页面的单击事件绑定事件处理函数
    document.onclick = changeBg;
    </script>
</head>
<body id="test">
</body>
</html>
```

上面的代码先通过 getElementById 方法获取到页面的<body.../>元素，然后通过修改 body 元素的 style.backgroundColor 属性来改变页面的背景色。

➤➤5.5.2 卷帘效果

卷帘效果是利用在程序中动态修改目标对象高度完成的。在发出展开的信号后，目标对象的高度

随时间的流逝而变长；相反，在发出收起信号后，目标对象的高度随时间的流逝而变短。当然，为了使展开过程和收起过程不会互相影响，可为两个处理过程增加旗标控制。下面是卷帘效果的页面代码：

程序清单：codes\05\5.5\wind.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>卷帘效果</title>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
</head>
<body>
<input id="shen" type="button" value="展开"/>
<input id="shou" type="button" value="收起"/>
<div id="wind" style="background-color:#bbbbbb;width:200px;height:1px;">
</div>
<script type="text/javascript">
//控制展开的旗标
var shenflag = true;
//控制收起的旗标
var shouflag = false;
//控制卷帘展开的函数
function shen()
{
    if (shenflag)
    {
        //控制收起旗标为 false，即在展开过程中不受收起按钮的影响
        shouflag = false;
        var tm;
        var windHeight = document.getElementById("wind").style.height;
        //如果目标对象的高度小于 100px，则让其高度继续增大
        if (parseInt(windHeight.substring( 0, windHeight.
            indexOf("px"))) < 100)
        {
            //修改目标对象的高度为原有高度 + 2
            document.getElementById("wind").style.height =
                parseInt(windHeight.substring( 0,
                windHeight.indexOf("px"))) + 2 + "px";
            //每 50ms 将为目标对象的高度增加 2
            tm = setTimeout("shen()" , 50);
        }
        else
        {
            clearTimeout(tm);
            shouflag = true;
        }
    }
}
function shou()
{
    if (shouflag)
    {
        //控制展开旗标为 false，即在展开过程中不受收起按钮的影响
        shenflag = false;
        var tm;
        var windHeight = document.getElementById("wind").style.height;
        //如果目标对象的高度大于 3px，则让其高度继续减小
        if (parseInt(windHeight.substring( 0, windHeight.
```

```
            indexOf("px"))) > 3)
        {
            //修改目标对象的高度为原有高度 - 2
            document.getElementById("wind").style.height =
                parseInt(windHeight.substring( 0,
                windHeight.indexOf("px"))) - 2 + "px";
            //每隔 50ms 将目标对象的高度减 2
            tm = setTimeout("shou()" , 50);
        }
        else
        {
            clearTimeout(tm);
            shenflag = true;
        }
    }
}
//为两个按钮绑定事件处理函数
document.getElementById("shen").onclick=shen;
document.getElementById("shou").onclick=shou;
</script>
</body>
</html>
```

上面的简单代码实现了卷帘效果，在浏览器中浏览该页面，单击“展开”按钮，将看到卷帘缓缓展开；如果单击“收起”按钮，将看到卷帘缓缓收起。对于 Ajax 应用而言，将有大量的场景需要在脚本中控制目标对象的 CSS 样式。在脚本中控制 CSS 样式务必注意的是：脚本中的 CSS 属性名与静态的 CSS 属性名并不完全相同。

➤➤5.5.3 动态增加立体效果

立体效果是一种很简单的 CSS 效果，其原理是通过为其增加四个边框，其中左、上边框的颜色稍浅，而下、右边框的颜色稍深即可。本示例将这种立体效果定义成一个整体的 CSS 样式效果，然后通过鼠标事件激发，将整个 CSS 立体效果应用到指定的 XHTML 元素之上。页面代码如下：

程序清单：codes\05\5.5\wind.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>立体效果</title>
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <script type="text/javascript">
    function chg()
    {
        document.getElementById("up").className="solid";
    }
    </script>
    <style type="text/css">
    /* 对所有 class 属性值为 solid 的元素起作用的 CSS 定义 */
    .solid {
        width:160px;
        text-align:center;
        border-right: #222222 2px solid;
        border-top: #dddddd 2px solid;
        border-left: #dddddd 2px solid;
        border-bottom: #222222 2px solid;
        background-color: #cccccc;
```

```
    }
    </style>
</head>
<body>
    <input type="button" onclick='chg()' value="增加立体效果" />
    <div id="up">有立体效果的层</div>
</body>
</html>
```

在上面的页面代码中可看到，将目标元素的背景色设为#cccccc，而将上、左边框的颜色设为#dddddd，而将右、下边框的颜色设为#222222，这就可以产生立体效果。单击页面中的“增加立体效果”按钮，页面将出现如图 5.20 所示的效果。

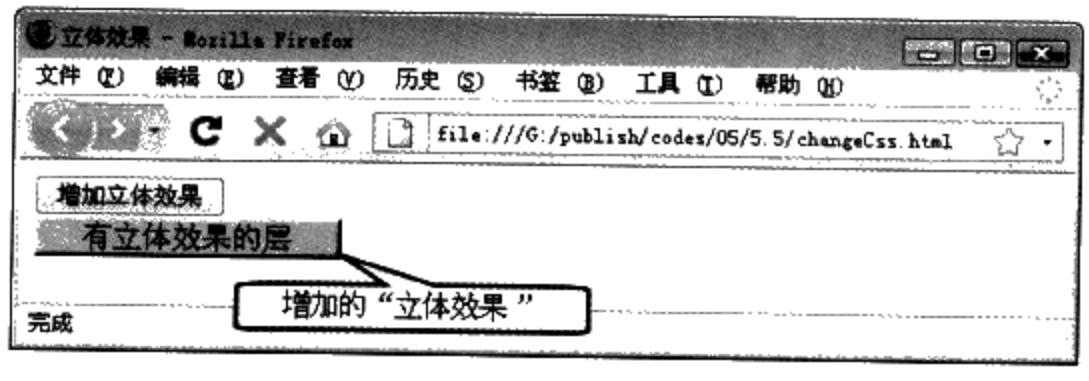


图 5.20 动态增加立体效果

5.6 本章小结

本章详细介绍了 CSS 的相关内容。本章在介绍 CSS 时，更侧重于介绍 Ajax 应用相关的 CSS，因此并未罗列出 CSS 那些琐碎的细节。

本章的知识比较简单，关键是掌握使用 CSS 的三种方式（外部、内部和内联），并掌握定义 CSS 样式的语法：Selector { ..... }，理解 Selector 决定了 CSS 样式对哪些 XHTML 元素起作用，而 CSS 属性定义决定了对 XHTML 元素起怎样的作用。本章所介绍的几种 Selector，需要重点掌握属性选择器、ID 选择器、class 选择器。本章还详细介绍了 CSS 常用的属性及其合法的属性值，这些内容很难一时间全部记住，读者可以在实践中慢慢掌握。

本章最后还详细介绍了如何在脚本中修改 CSS 样式（改变 style 和 className 属性），总结了静态 CSS 属性名和脚本中 CSS 属性名的区别和联系，并通过几个简单的示例示范了如何在脚本中修改 CSS 样式。这些内容是进行 Ajax 编程所必须掌握的技能。

CHAPTER

6

第6章
DOM 模型详解

本章要点

- DOM 模型简介
- DOM 模型的思想和作用
- DOM 和 XHTML 文档
- XHTML 元素在 DOM 模型中的实现类
- XHTML 元素之间的包含关系
- 访问 XHTML 元素的几种方法
- 修改 XHTML 元素的方法
- 增加 XHTML 元素的几种方法
- 删除 XHTML 元素的几种方法
- 传统 DHTML 对象模型
- DHTML 对象模型的包含关系
- window 对象的常用方法和功能
- document 对象的常用方法和功能
- 常用的两个范例
- 使用 DOM 模型来处理 XML 文档

DOM 是文档对象模型（Document Object Model）的简称。借助 DOM 模型，可以将一个结构化文档转换成 DOM 树，程序可以访问、修改树里的节点，也可以新增、删除树里的节点。在程序操作这棵 DOM 树时，结构化文档也会随之动态改变。

简单地说，DOM 采取直观、一致的方式将结构化文档进行模型化处理，形成一棵结构化的文档树，从而提供访问、修改该文档的简易编程接口。因此，一旦掌握了 DOM 编程模型，就拥有了使用 JavaScript 脚本动态修改 XHTML 页面的能力。在第 2 章已经提到，Ajax 编程的核心有两点：

- 通过 XMLHttpRequest 发送异步请求。
- 使用 DOM 动态加载服务器响应。

由此可见，掌握 DOM 模型是学习 Ajax 编程的基础。通过 DOM 技术，不仅可以操作 XHTML 文档的内容，包括新增节点、修改节点属性、删除节点等，而且还能操纵 XHTML 页面的风格样式。DOM 由 W3C 组织所倡导。因此，大多数浏览器都支持这项技术，DOM 目前比较常用的版本是 DOM 2。

实际上，DOM 模型不仅可以操作 XHTML 文档，还可操作 XML 文档。

6.1 DOM 模型概述

正如第 3 章介绍的，XHTML 文档只有一个根节点，而其他节点以根节点的子节点或孙子节点的形式存在，最终形成一个结构化文档。DOM 模型则用于导航、访问结构化文档的节点，并提供新增、修改、删除结构化文档的能力。

简单地说，DOM 提供了访问结构化文档的一种方式，但 DOM 并不是一种技术，它只是访问结构化文档（主要是 XML 文档和 XHTML 文档）的一种思想。基于这种思想，各种语言都有自己的 DOM 解析器。

DOM 解析器的作用就是完成结构化文档和 DOM 树之间的转换关系。通常来说，DOM 解析器解析结构化文档，就是将磁盘上的结构化文档转换成内存中的 DOM 树；而从 DOM 树输出 DOM 树，就是将内存中的 DOM 树转换成磁盘上的结构化文档。图 6.1 显示了这种转换关系。

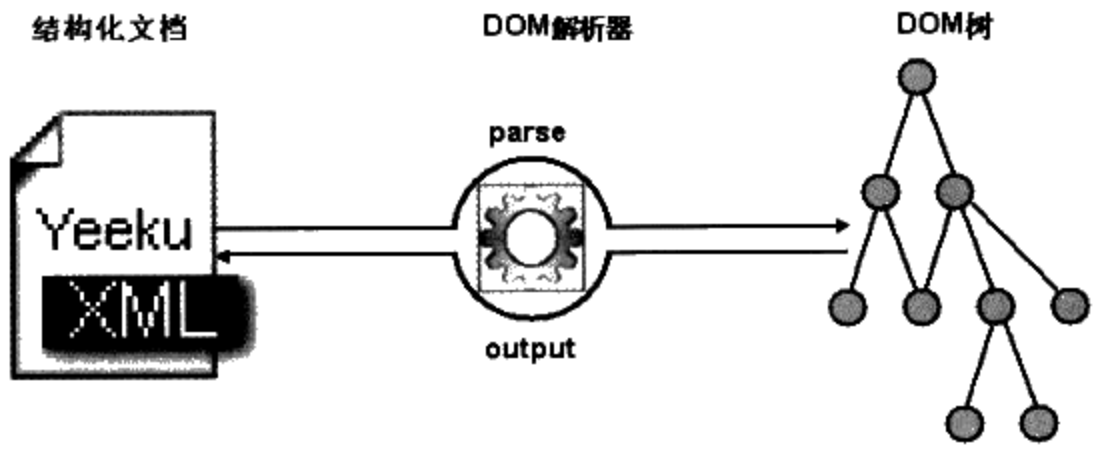


图 6.1 DOM 解析器的功能

对于支持 DOM 模型的浏览器而言，它们提供了更丰富的功能：当浏览器转入一个 XHTML 页面后，浏览器里已经得到了该 XHTML 文档对应的 DOM 树。在我们通过 JavaScript 脚本修改这棵 DOM 树时，浏览器里的 XHTML 页面会随之改变。

图 6.2 显示了 Eclipse 中设计 XHTML 文档对应的 DOM 视图。

在如图 6.2 所示的 DOM 树中，用光标选中 tr 元素时，左边 XHTML 文档中也以高亮的方式选中了对应的行。通过使用 DOM 模型，JavaScript 可以动态更新 XHTML 页面的内容。

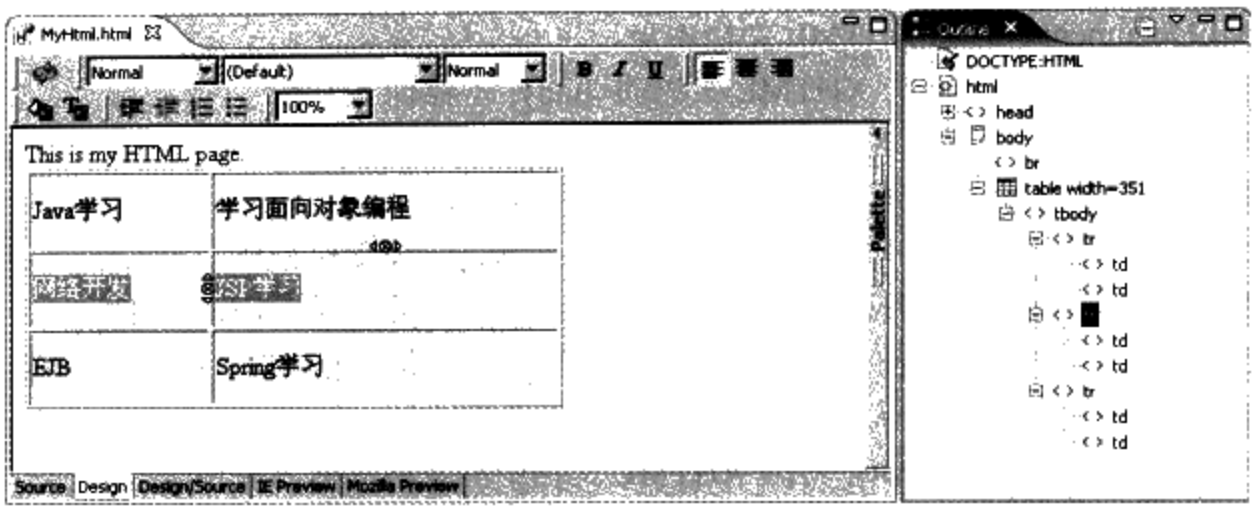


图 6.2　XHTML 文档与 DOM 树的对应

6.2　DOM 模型和 XHTML 文档

XHTML 文档是一种结构化文档。虽然早期的 HTML 文档有很多不规范的地方，但 HTML 规范版本是 XHTML，而 XHTML 是一个严格的结构化文件，遵守 XHTML 规范的 HTML 文档必须是一份格式良好的 XML 文档。

因为 XHTML 已经变成了规范的结构化文档，所以我们可以使用 DOM 来自由操作 XHTML 文档了——这也是 XHTML 规范取代 HTML 规范的重要原因。

➤➤6.2.1　XHTML 元素之间的继承图

DOM 为常用 XHTML 元素提供了一套完整的继承体系。从页面的 document 对象到每个常用的 XHTML 元素，DOM 模型都提供了对应的类，每个类都提供了相应的方法来操作 DOM 元素本身、属性及其子元素。DOM 模型允许以树的方式操作 XHTML 文档中的每个元素。

虽然 JavaScript 不是一门纯粹的面向对象的语言，但 DOM 还是为 XHTML 元素提供了一种简单继承关系。DOM 模型里 XHTML 元素的继承关系如图 6.3 所示。

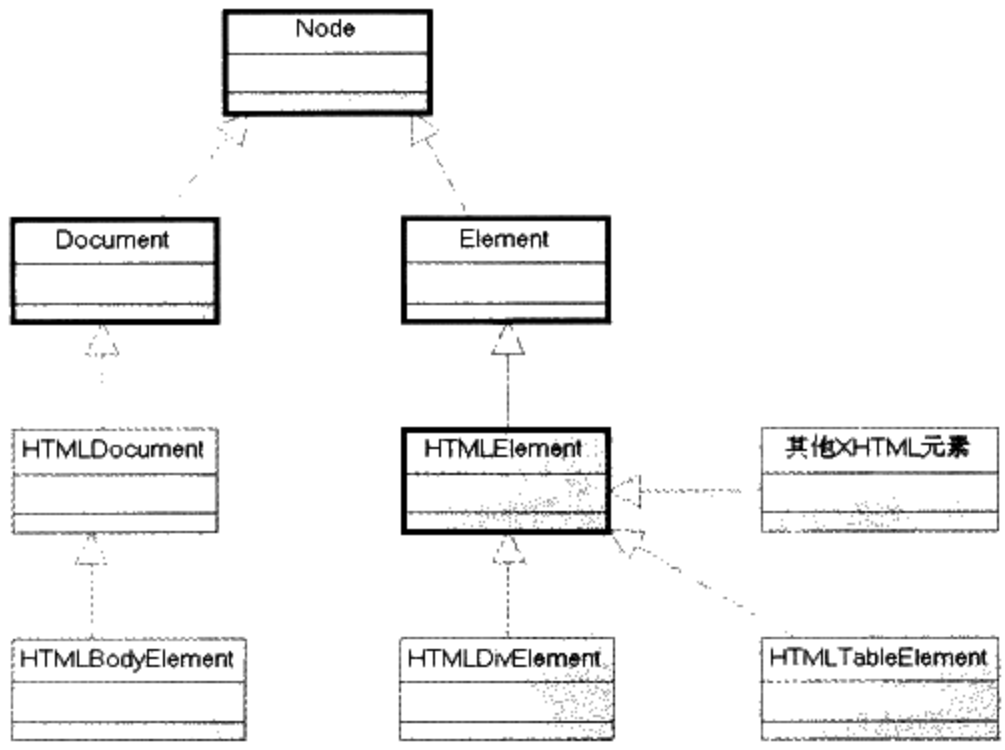


图 6.3　XHTML 元素的继承关系

在图 6.3 中粗线框框出的 4 个元素：Node、Document、Element、HTMLElement，都是普通 XHTML 元素的超类，不直接对应于 XHTML 页面控件，但它们所包含的方法也可被其他页面元素调用。除此之外，还有如下常用的 XHTML 元素。

- HTMLDocument：对应 XHTML 文档本身。
- HTMLBodyElement：对应 XHTML 文档的<body.../>控件。
- HTMLDivElement：对应 XHTML 文档中的普通<div.../>控件。
- HTMLFormElement：对应 XHTML 文档的表单控件。
- HTMLSelectElement：对应 XHTML 文档的列表框、下拉菜单控件。
- HTMLOptionElement：对应 XHTML 文档中列表框、下拉菜单的选项控件。
- HTMLInputElement：对应 XHTML 文档中的单行文本框、密码框、按钮等控件。
- HTMLTableElement：对应 XHTML 文档中的表格控件。
- HTMLTableCaptionElement：对应 XHTML 文档中的表格的标题控件。
- HTMLTableColElement：对应于 XHTML 文档中的表格的列控件。
- HTMLTableCellElement：对应于 XHTML 文档中的表格的单元格控件。
- HTMLOLElement：对应于 XHTML 文档的有序列表控件。
- HTMLULElement：对应于 XHTML 文档的无序列表控件。
- HTMLLIElement：对应于 XHTML 文档的列表项控件。

正如在第 3 章中所看到的，XHTML 元素之间的父子关系有比较严格的限制。例如，HTMLCellElement 通常只能作为 HTMLColElement 的子元素使用。下面介绍这种常用的包含关系。

➤➤6.2.2 XHTML 元素之间常见的包含关系

有些 XHTML 元素之间可以互相嵌套，例如<div.../>元素可以互相嵌套，但有些 XHTML 元素则不可互相嵌套，例如<td.../>元素只能作为<tr.../>元素的子元素、<option.../>元素只能作为<select.../>元素的子元素。图 6.4 描述了常用 XHTML 元素之间的包含关系。

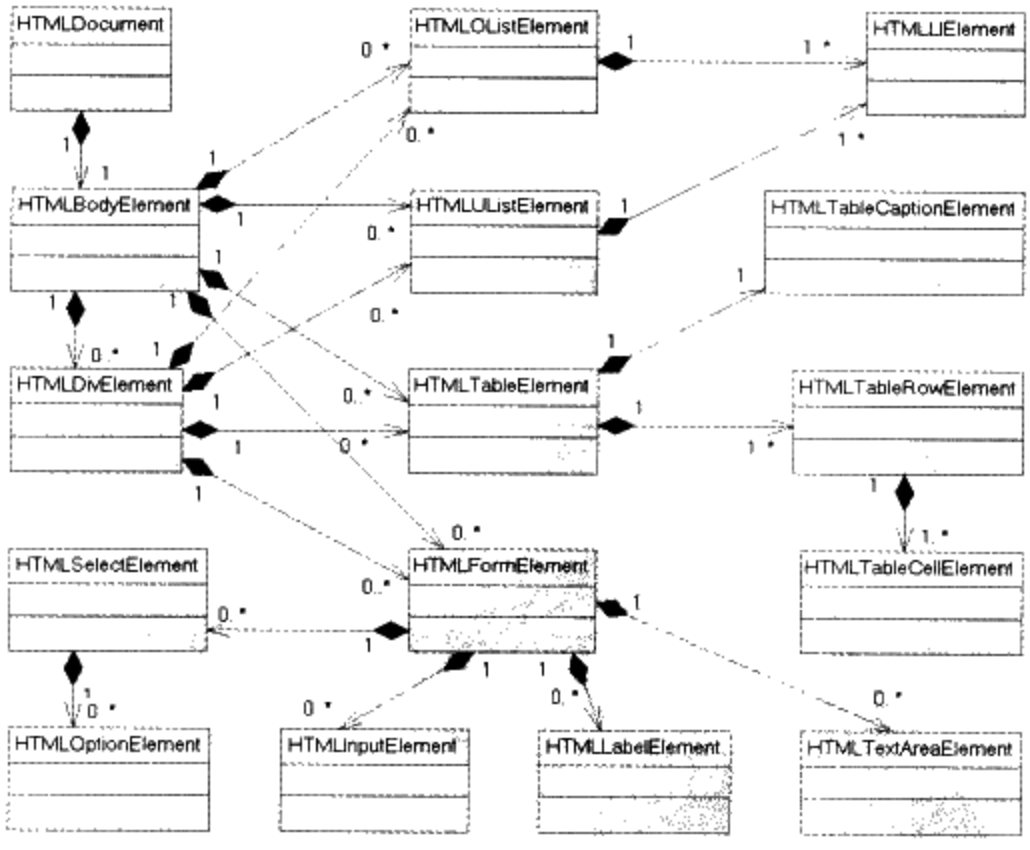


图 6.4 XHTML 元素之间的包含关系

从图 6.4 中可看出，HTMLDocument 对象作为整个 HTML 文档的最大的对象，里面可以包含一个 HTMLBodyElement 对象。HTML 文档中还有两个对象体系：表单对象和表格对象。表单对象里可以包含基本的输入对象，还可以包含<select.../>元素，<select.../>元素可以包含多个<option.../>元素。

表格对象可以包含标题（HTMLTableCaptionElement）控件，还可以包含多个表格行（HTML-TableColElement）控件，每个表格行又可以包含多个单元格（HTMLTableCellElement）控件。

注意：

使用 DOM 元素增加子元素时，必须注意元素之间合理的包含关系。例如，不要为<td.../>元素添加<tr.../>子元素，虽然语法上没有错误，但这种结构在 XHTML 文档上无法显示；在定义<table.../>元素时，至少要为其增加一个<tr.../>元素，否则该表格将没有任何显示。

6.3 访问 XHTML 元素

为了动态修改 XHTML 元素，必须能访问 XHTML 元素，DOM 提供了 2 种方式来访问 XHTML 元素：

- 根据 ID 来访问 XHTML 元素。
- 利用节点关系访问 XHTML 元素。

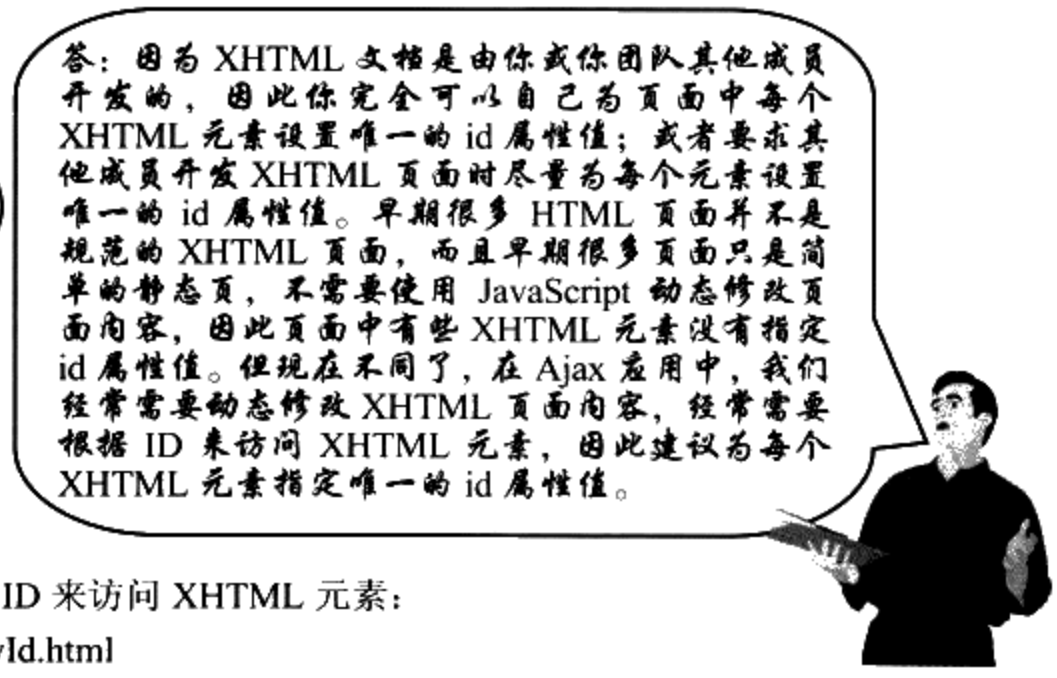
前一种方式简单易用，主要由 document 提供来的 getElementById()方法来完成；后一种方式则利用树节点之间的父子、兄弟关系来访问。

➤➤6.3.1 根据 ID 访问 XHTML 元素

根据 ID 访问 XHTML 元素要由如下方法实现：

- document.getElementById(idVal)：返回文档中 id 属性值为 idVal 的 XHTML 元素。

上面这个方法简单易用，只要被访问 XHTML 元素具有唯一的 id 属性，那么 JavaScript 脚本就可以方便地访问到该元素。



下面的页面代码示范了如何根据 ID 来访问 XHTML 元素：

程序清单：codes\06\6.3\AccessById.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 根据 ID 访问 XHTML 元素 </title>
    <script type="text/javascript">
        function accessById()
        {
            alert(document.getElementById("a").innerHTML + "\n"
                + document.getElementById("b").value);
        }
    </script>
</head>
<body>
    <div id="a">疯狂 Java 讲义</div>
```

```
    <textarea id="b" rows="3" cols='25'>轻量级 Java EE 企业应用实战</textarea>
    <input type="button" value="访问 2 个元素" onclick="accessById();" />
</body>
</html>
```

上面的页面中定义了一个 ID 为 a 的<div.../>元素，一个 ID 为 b 的<textarea.../>元素，页面中还定义了一个简单按钮，当用户单击该按钮时执行 accessById()函数，该函数只是弹出一个警告提示框，该提示框输出<div.../>元素的 innerHTML 属性和<textarea.../>元素的 value 属性。

在浏览器中浏览该页面，并单击页面中的“访问 2 个元素”按钮，可看到如图 6.5 所示警告框。

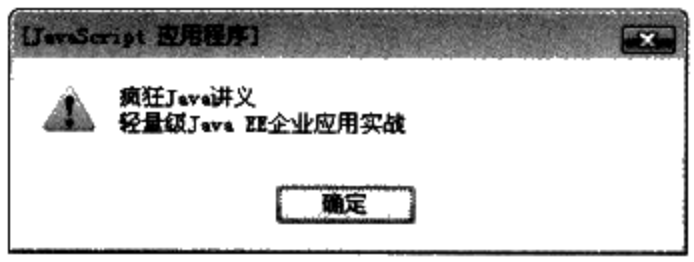
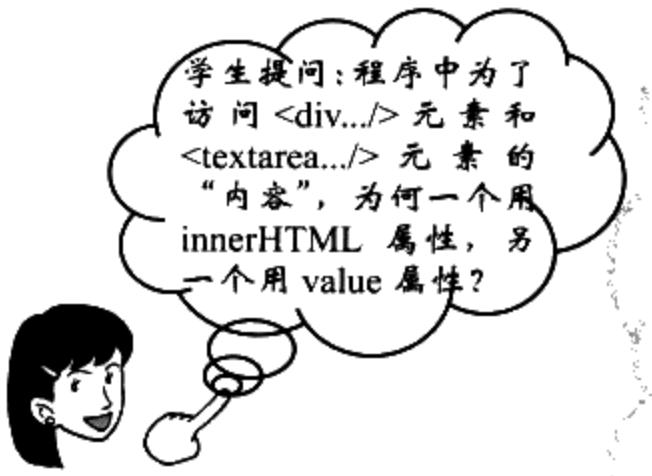


图 6.5　根据 ID 访问 XHTML 元素

从图 6.5 中可以看出，该警告框的内容正好是<div.../>元素和<textarea.../>元素的“内容”。由此可见，使用 document.getElementById()方法来访问 XHTML 元素非常简单。



答：DOM 模型扩展了 XHTML 元素，为几乎所有 XHTML 元素都新增了 innerHTML 属性，该属性代表该元素的“内容”——当某个元素的开始标签、结束标签之间都是字符串内容时（不包含其他子元素），JavaScript 子元素可通过它的 innerHTML 属性返回这些字符串内容。但<textarea.../>例外，因为它是一个表单域控件，它的开始标签和结束标签之间的内容是它的值，因此只能通过 value 属性来访问。不仅如此，还有<input.../>元素所生成的表单域控件，包括单行文本框、各种按钮等，它们的可视化文本都由 value 属性控制，因此也通过 value 来获取它们的“内容”。除此之外的其他 XHTML 元素，包括列表框、下拉菜单的列表项、<label.../>表单域、<button.../>按钮，都应通过 innerHTML 来获取它们的“内容”。

➤➤6.3.2　利用节点关系访问 XHTML 元素

一旦获取了某个 XHTML 元素，由于该元素实际上与 DOM 树的某个节点对应，因此我们完全可以利用节点之间的父子、兄弟关系来访问 XHTML 元素。

利用节点关系访问 XHTML 元素的属性和方法如下：

- Node parentNode：返回当前节点的父节点。只读属性。
- Node previousSibling：返回当前节点的前一个兄弟节点。只读属性。
- Node nextSibling：返回当前节点的后一个兄弟节点。只读属性。
- Node[] childNodes：返回当前节点的所有子节点。只读属性。
- Node[] getElementsByTagName(tagName)：返回当前节点的具有指定标签名的所有子节点。
- Node firstChild：返回当前节点的第一个子节点。只读属性。
- Node lastChild：返回当前节点的最后一个子节点。只读属性。

下面的页面代码示范了如何利用节点关系访问 XHTML 元素：

程序清单：codes\06\6.3\AccessByNodeRelation.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 根据节点关系访问 XHTML 元素 </title>
    <style type="text/css">
        /* 定义改变背景色的 CSS，表示被选中的项 */
        .selected {
            background-color:#6666ff
        }
    </style>
</head>
<body>
    <ol id="books">
        <li id="java">疯狂 Java 讲义</li>
        <li id="ssh">轻量级 Java EE 企业应用实战</li>
        <li id="ajax" class="selected">疯狂 Ajax 讲义</li>
        <li id="xml">疯狂 XML 讲义</li>
        <li id="ejb">经典 Java EE 企业应用实战</li>
        <li id="workflow">疯狂 Workflow 讲义</li>
    </ol>
    <input type="button" value="父节点"
        onclick="change(curTarget.parentNode);" />
    <input type="button" value="第一个"
        onclick="change(curTarget.parentNode.firstChild.nextSibling);" />
    <input type="button" value="上一个"
        onclick="change(curTarget.previousSibling.previousSibling);" />
    <input type="button" value="下一个"
        onclick="change(curTarget.nextSibling.nextSibling);" />
    <input type="button" value="最后一个"
        onclick="change(curTarget.parentNode.lastChild.previousSibling);" />
    <script type="text/javascript">
        var curTarget = document.getElementById("ajax");
        function change(target)
        {
            alert(target.innerHTML);
        }
    </script>
</body>
</html>
```

上面的页面代码定义 ID 为 ajax 的<li.../>元素为当前元素，页面中提供了 5 个按钮，分别访问当前元素的“父元素”和“第一个”、“上一个”、“下一个”、“最一个”等兄弟元素，页面中的粗体字代码是访问这些节点的关键代码。

例如访问当前节点的“上一个”兄弟节点，页面中使用 curTarget.previousSibling.previousSibling，程序中两次调用 previousSibling 属性，这并没有错误！读者可能感到疑惑了：这不是访问上两个兄弟节点吗？没错！确实是访问上两个节点！

需要指出的是<ol.../>节点一共包含 13 个子节点，而不是 6 个子节点！因为在每两个<li.../>节点之间都有一片“空白”（换行和空格），每片“空白”也将被当成<ol.../>元素的子节点。因为在我们使用 curTarget.previousSibling 访问当前节点的上一个节点时，实际上得到一个“空白”节点，而实际上我们需要访问上一个<li.../>节点，实际上是上两个节点。

注意：

对于 XHTML 页面而言，浏览器会将页面中的“空白”也当成文本节点，在我们使用 DOM 模型访问 XHTML 页面元素时必须小心处理。

在 Opera 浏览器中浏览该页面，并单击“上一个”按钮，将看到如图 6.6 所示提示框。

如果用 Firefox 浏览该页面，也可以看到与如图 6.6 所示完全相同的结果。但如果我们使用 Internet Explorer 6 浏览该页面，并单击“上一个”按钮，将看到弹出如图 6.7 所示提示框。

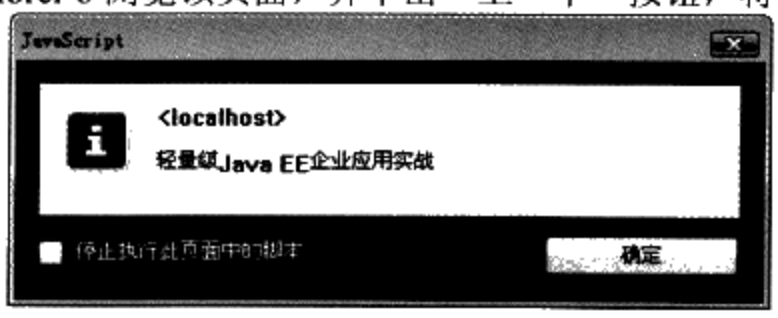


图 6.6　根据节点关系访问 XHTML 元素

图 6.7 Internet Explorer 6 根据节点关系访问 XHTML 元素

从图 6.7 可以看出，Internet Explorer 6 输出的实际上是上两个<li.../>元素的“内容”，也就是说：Internet Explorer 6 又没有把页面中的“空白”当成子元素！

笔者的机器上只安装了 Firefox 3、Opera 9.5 和 Internet Explorer 6 三个浏览器，大家可以发现 Firefox 3、Opera 9.5 保持一致，但 Internet Explorer 6 比较“与众不同”！实际上，将“空白”当成子元素处理是规范，其他浏览器如 Safari、Chrome 都会遵守该规范。

为了使 Ajax 应用能实现“跨浏览器”，必须考虑 Internet Explorer 6 的兼容性，因此应尽量少用这种方式来访问 XHTML 元素。

➤➤6.3.3　访问表单域控件

表单在 DOM 中由 HTMLFormElement 对象表示，该对象除了可调用前面介绍的基本属性和方法之外，还拥有如下几个常用属性：

- action：返回该表单的 action 属性值，该属性用于指定表单的提交地址。读写属性。
- elements：返回表单内全部表单域控件所组成的数组。使用该数组可以访问该表单内的任何表单控件。只读属性。
- length：返回表单内表单域的个数，该属性等于 elements.length 的值。只读属性。
- method：返回该表单的 method 属性，该属性通常有 POST 和 GET 两个值，默认采用 GET 方法。该属性用于确定表单发送请求的方式。读写属性。
- target：用于确定提交表单时的结果窗口，可以是_self、_parent、_top、_blank 等值。读写属性。

除此之外，Form 对象还有如下两个常用方法：

- reset()：重设表单，将所有的表单域的值设置为初始值。
- submit()：提交表单。

因为 HTMLFormElement 提供了 elements 属性返回表单内的全部表单控件，因此可通过该属性访问表单里的表单控件。看如下页面代码：

程序清单：codes\06\6.3\AccessFormElement.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 访问表单域控件 </title>
</head>
<body>
<form id="d" action="" method="get">
    <input name="user" type="text" /><br />
    <input name="pass" type="text" /><br />
    <select name="color">
        <option value="red">红色</option>
```

```
        <option value="blue">蓝色</option>
    </select><br />
    <input type="button" value="第一个"
        onclick="alert(document.getElementById('d').elements[0].value);" />
    <input type="button" value="第二个"
    onclick="alert(document.getElementById('d').elements['pass'].value);" />
    <input type="button" value="第三个"
        onclick="alert(document.getElementById('d').color.value);" />
</form>
</body>
</html>
```

上面的粗体字代码先访问页面的表单元素，再使用表单元素的 elements 属性来访问该表单内的表单域控件，例如 document.getElementById('d').elements[0]即访问该表单内的第一个表单域控件。

实际上，HTMLFormElement 的 elements 属性值并不是一个普通数组，而是一个 HTMLCollection 对象，该对象既可当成普通数组使用（即通过数字索引访问元素），也可通过关联数组来访问（即通过字符串索引来访问元素）。因此上面页面的第一行代码可通过 elements['pass']，即表单里 name 或 id 为 pass 的表单域控件来访问——如果表单内有多个表单域控件的 name 或 id 属性值为 pass，则 elements['pass'] 将再次返回一个 HTMLCollection 对象。

提 示：

不仅 HTMLFormElement 的 elements 属性值是 HTMLCollection 对象。实际上，XHTML 元素中许多可能返回对象数组的方法、属性值实际都得到的是一个 HTMLCollection 对象，例如前面介绍的 Node 所提供的 childNodes 等。

不仅如此，HTMLFormElement 还有个更有效率的方法，可用于访问表单域控件，语法如下：

- formObj.elementName：返回表单中 id 或 name 为 elementName 的表单域控件。

该方法也根据表单域控件的 id 或 name 属性来访问表单域控件，因此我们看到第三行粗体字代码使用 document.getElementById('d').color 来访问该表单里 id 或 name 为 color 的表单域控件。与前面类似的是，如果该表单里包含多个 id 或 name 为 color 的表单域控件，则 document.getElementById('d').color 将再次得到一个 HTMLCollection 对象。

➤➤6.3.4 访问列表框、下拉菜单的选项

HTMLSelectElement 代表一个列表框或下拉菜单，HTMLSelectElement 对象除了可使用普通 XHTML 元素的各种属性和方法外，还支持如下额外的属性：

- form：返回列表框、下拉菜单所在的表单对象。只读属性。
- length：返回列表框、下拉菜单的选项个数。该属性的值可通过增加或删除列表框的选项来改变。只读属性。
- options：返回列表框、下拉菜单里所有选项组成的数组。只读属性。
- selectedIndex：该属性返回下拉列表选中选项的索引，如果有多个选项被选中，则只返回第一个被选中选项的索引。读写属性。
- type：返回该下拉列表的类型，即是否允许多选。如果允许多选，则返回 select-multiple；如果不支持多选，则返回 select-one。

HTMLSelectElement 的 options 属性可直接访问该列表框、下拉菜单的所有列表项，传入指定索引即可访问指定列表项，语法格式如下：

- select.options[index]：返回列表框、下拉菜单的第 index+1 个选项。

列表框、下拉菜单的选项由 HTMLOptionElement 对象表示，除了前面介绍的普通属性之外，该对象还提供了如下几个常用属性：

- form：返回包含该选项所处列表框、下拉菜单的表单对象。
- defaultSelected：返回该选项默认是否被选中。只读属性。
- index：返回该选项在列表框、下拉菜单中的索引。只读属性。当然也可以通过增加或删除列表框的选项来改变该选项的索引值。
- selected：返回该选项是否被选中，通过修改该属性可动态改变该项是否被选中。
- text：返回该选项呈现出来的文本，就是<option>和</option>之间的文本。对 HTMLOptionElement 而言，该属性与 innerHTML 属性相同。
- value：返回每个选项的 value 属性，可以通过设置该属性来改变选项的 value 值。

下面的页面代码示范了访问列表框、下拉菜单中列表项的用法：

程序清单：codes\06\6.3\AccessSelect.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 访问列表项 </title>
</head>
<body>
    <select id="mySelect" name="mySelect" size="6">
        <option value="java">疯狂 Java 讲义</option>
        <option value="ssh">轻量级 Java EE 企业应用实战</option>
        <option value="ajax" selected="selected">疯狂 Ajax 讲义</option>
        <option value="xml">疯狂 XML 讲义</option>
        <option value="ejb">经典 Java EE 企业应用实战</option>
        <option value="workflow">疯狂 Workflow 讲义</option>
    </select><br />
    <input type="button" value="第一个"
        onclick="change(curTarget.options[0]);" />
    <input type="button" value="上一个"
        onclick="change(curTarget.options[curTarget.selectedIndex - 1]);" />
    <input type="button" value="下一个"
        onclick="change(curTarget.options[curTarget.selectedIndex + 1]);" />
    <input type="button" value="最后一个"
        onclick="change(curTarget.options[curTarget.length - 1]);" />
    <script type="text/javascript">
        var curTarget = document.getElementById("mySelect");
        function change(target)
        {
            alert(target.text);
        }
    </script>
</body>
</html>
```

上面的页面中粗体字代码是 HTMLSelectElement 对象访问各选项的方法，这些用法可以很好地跨浏览器，可以在各种浏览器中运行良好。

6.3.5 访问表格子元素

HTMLTableElement 代表表格，HTMLTableElement 对象除了可使用普通 HTML 元素的各种属性和方法外，还支持如下额外的属性：

- caption：返回该表格的标题对象。可通过修改该属性来改变表格的标题。
- HTMLCollection rows：返回该表格里的所有表格行，该属性会返回<thead.../>、<tfoot.../>和

<tbody.../>元素里的所有表格行。只读属性。

- HTMLCollection tBodies：返回该表格里所有<tbody.../>元素所组成的数组。
- tFoot：返回该表格里的<tfoot.../>元素。
- tHead：返回该表格里的所有<thead.../>元素。

在获得一个表格之后，完全可以通过上面提供的一系列属性来访问表格“内容”，例如 caption 属性返回该表格的标题，rows 属性返回该表格的全部表格行……与前面介绍的完全相似，如果需要访问表格的指定表格行，只需要使用如下格式即可：

- table.rows[index]：返回该表格第 index + 1 行的表格行。

HTMLTableRowElement 代表表格行，HTMLTableRowElement 对象除了可使用普通 HTML 元素的各种属性和方法外，还支持如下额外的属性：

- cells：返回该表格行内所有的单元格组成的数组。只读属性。
- rowIndex：返回该表格行在表格内的索引值。只读属性。
- sectionRowIndex：返回该表格行在其所在元素（<tbody.../>、<thead.../>等元素）的索引值。只读属性。

HTMLTableCellElement 代表单元格，HTMLTableCellElement 对象除了可使用普通 HTML 元素的各种属性和方法外，还支持如下额外的属性：

- cellIndex：返回该单元格在该表格行内的索引值。只读属性。

下面的代码示范了如何访问 HTML 表格的内容：

程序清单：codes\06\6.3\AccessTable.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 访问表格元素 </title>
</head>
<body>
<table id="d" border="1">
    <caption>疯狂 Java 体系</caption>
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>轻量级 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 Ajax 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 XML 讲义</td>
        <td>疯狂 Workflow 讲义</td>
    </tr>
</table>
<input type="button" value="表格标题"
    onclick="alert(document.getElementById('d').caption.innerHTML);" />
<input type="button" value="第一行、第一格"
    onclick="alert(document.getElementById('d').rows[0].cells[0].innerHTML);" />
<input type="button" value="第二行、第二格"
    onclick="alert(document.getElementById('d').rows[1].cells[1].innerHTML);" />
<input type="button" value="第三行、第二格"
    onclick="alert(document.getElementById('d').rows[2].cells[1].innerHTML);" />
</body>
</html>
```

6.4 修改 XHTML 元素

访问到指定 XHTML 元素之后，还可以对该元素进行修改，通过修改 XHTML 元素就可以实现动态更新 XHTML 页面的目的了。

修改节点通常是修改节点的内容，修改节点的属性，或者修改节点的 CSS 样式。总结起来一句话：XHTML 元素的所有读写属性都可被修改！一旦 XHTML 元素的属性值被修改，XHTML 页面上对应的内容也就随之改变。

修改 XHTML 元素通常通过修改如下几个常用属性来实现：

- innerHTML：大部分 XHTML 页面元素如<div.../>、<td.../>的呈现内容由该属性控制。
- value：少量表单控件如<input.../>、<textarea.../>的呈现内容由该属性控制。
- className：修改 XHTML 元素的 CSS 样式，该属性的合法值是一个 class 选择器名。
- style：修改 XHTML 元素的内联 CSS 样式。
- options[index]：直接对<select.../>元素的指定列表项赋值，可改变列表框、下拉菜单的指定列表项。

下面的示例代码演示了一个可编辑的表格，在页面中指定需要修改的表格行、列，然后输入要修改的值，即可动态修改单元格的内容。

程序清单：codes\06\6.3\changeValue.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title> 编辑表格值 </title>
</head>
<body>
改变第<input id="row" type="text" size="2" />行,
第<input id="cel" type="text" size="2" />列的值为:
<input id="celVal" type="text" size="16" /><br />
<input id="chg" type="button" value="改变" onclick="change();" />
<table id="d" border="1">
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>轻量级 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 Ajax 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 XML 讲义</td>
        <td>疯狂 Workflow 讲义</td>
    </tr>
</table>
<script type="text/javascript">
    function change()
    {
        var tb = document.getElementById("d");
        var row = document.getElementById("row").value ;
        row = parseInt(row);
        //如果需要修改的行不是整数，弹出警告
        if(isNaN(row))
        {
            alert("您要修改的行必须是整数");
```

```
            return false;
        }
        var cel = document.getElementById("cel").value ;
        cel = parseInt(cel);
        //如果需要修改的列不是整数，弹出警告
        if(isNaN(cel))
        {
            alert("您要修改的列必须是整数");
            return false;
        }
        //如果需要修改的行或者列超出了表格的行或列，弹出警告
        if (row > tb.rows.length ||
            cel > tb.rows.item(0).cells.length)
        {
            alert("要修改的单元格不在该表格内");
            return false;
        }
        //修改单元格的值
        tb.rows.item(row - 1).cells.item(cel - 1).innerHTML
            = document.getElementById("celVal").value;
    }
</script>
</body>
</html>
```

上面的程序中粗体字代码是关键代码：直接为单元格的 innerHTML 属性赋值即可修改该单元格的内容。图 6.8 显示了输入相应行、列，已经要改变的值，然后单击“改变”按钮后的页面效果。

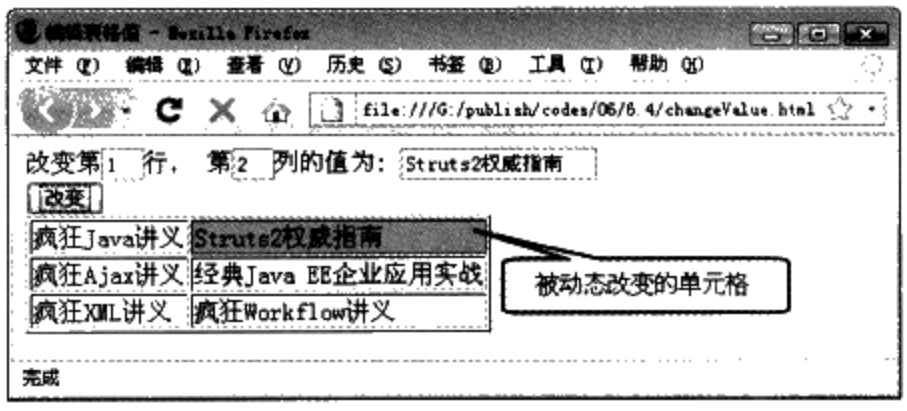


图 6.8 动态修改单元格的值

当然，要修改文本框里的值，直接修改该元素的 value 属性即可，并不需要使用 innerHTML 属性。关于元素的外观样式，可通过修改该元素的内联 CSS 样式实现，具体方法请参考第 5 章。

6.5 新增 XHTML 元素

JavaScript 脚本可以为 DOM 动态新增节点，程序为 DOM 树新增节点时，页面会动态地新增 XHTML 元素。当需要为页面新增 XHTML 元素时，需要按如下两个步骤操作：

（1）创建或复制节点。

（2）添加节点。

➤➤6.5.1 创建或复制节点

创建节点通常借助于 document 对象的 createElement 方法，语法如下：

- document.createElement(Tag)：创建 Tag 标签对应的节点。

下面的代码演示了如何创建一个节点：

程序清单：codes\06\6.5\createElement.html

```
<script>
    //创建一个新节点
    var a = document.createElement("div");
    //使用警告对话框输出节点
    alert(a);
</script>
```

在 Firefox 中浏览该页面，会弹出如图 6.9 所示的对话框。

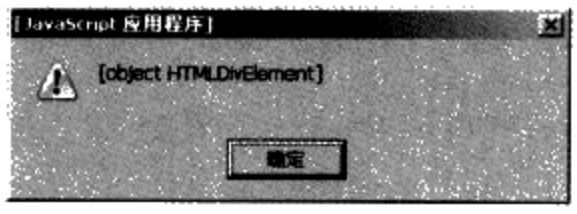


图 6.9　创建节点

当代码调用 document.createElement("div")创建节点后，将自动生成一个 HTMLDivElement 对象，该对象对应于 XHTML 文档中的<div.../>元素。因此，在创建元素时，传入的参数字符串并不是随意填写的，必须是一个合法标签名。再看下面的代码：

程序清单：codes\06\6.5\createElementError.html

```
<script>
    //创建一个新节点，传入不合法的标签名
    var a = document.createElement("divxxx");
    //使用警告对话框输出节点
    alert(a);
</script>
```

在浏览器中浏览该页面，将出现如图 6.10 所示的对话框。

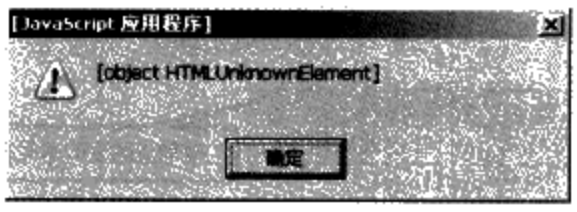


图 6.10　创建了一个未知的 XHTML 元素

注意：

调用 document.createElement()方法时，传入的参数必须是一个合法的 XHTML 标签。

创建一个节点的开销可能过大，实际上我们还可复制一个已有的节点，复制已有节点的系统开销略小。通过调用 Node 的 cloneNode()方法即可复制一个已有节点，该方法的语法格式如下：

- Node cloneNode(boolean deep)：复制当前节点。当 deep 为 true 时，复制当前节点的同时，复制该节点的全部后代节点；当 deep 为 false 时，仅复制当前节点。

如下代码示范了如何复制节点：

程序清单：codes\06\6.5\cloneElement.html

```
<ul id = "d">
    <li>疯狂 Java 讲义</li>
</ul>
<script type="text/javascript">
    //获取 ID 为 d 的节点
    var ul = document.getElementById("d");
    //复制 ul 的第二个子节点（不复制当前节点的后代节点）
    var ajax = ul.firstChild.nextSibling.cloneNode(false);
    //修改被复制的节点
```

```
    ajax.innerHTML = "疯狂 Ajax 讲义";
    //将复制的节点添加到页面中
    ul.appendChild(ajax);
</script>
```

上面的粗体字代码示范了如何复制新的节点，在浏览器中浏览该页面，可看到如图 6.11 所示效果。

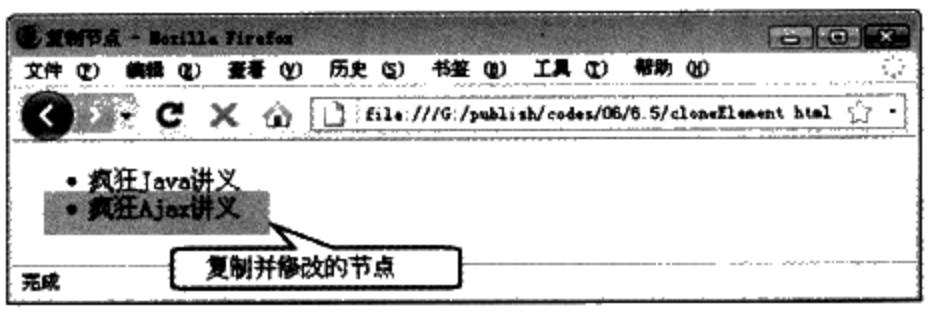


图 6.11　复制节点

➤➤6.5.2　添加节点

正如在 6.5.1 节所见到的：当一个节点创建成功后，一定要将该节点添加到 DOM 中才行。对于普通的节点，可采用 Node 对象的如下方法来添加节点：

- appendChild(Node newNode)：将 newNode 添加成当前节点的最后一个子节点。
- insertBofore(Node newNode, Node refNode)：在 refNode 节点之前插入 newNode 节点。
- replaceChild(Node newChild, Node oldChild)：将 oldChild 节点替换成 newChild 节点。

前面已经看到了 appendChild()方法的用法，下面仅对该代码进行简单修改，将原有的 appendChild()修改成 insertBefore()，修改后的关键代码如下：

程序清单：codes\06\6.5\insertChild.html

```
//将复制的节点插入 ul 的第一个子节点之前
ul.insertBefore(ajax, ul.firstChild);
```

在浏览器中浏览该页面可看到如图 6.12 所示的界面。

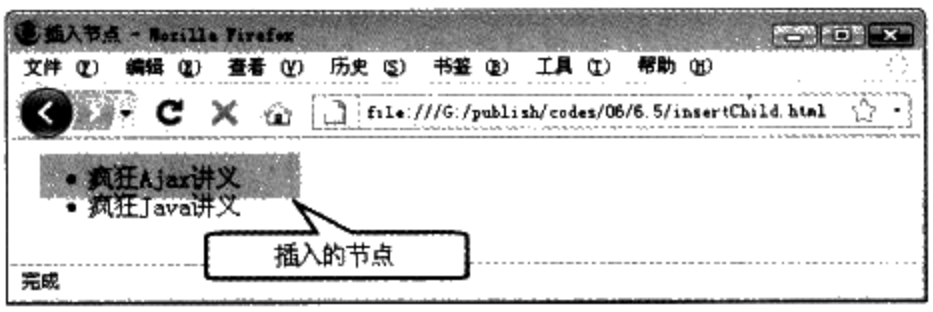


图 6.12　在指定节点之前插入节点

至于其他特殊的 XHTML 元素，则包含了更多的添加节点的方法。例如<select.../>则有更简单的方法来增加子节点，<table.../>、<tr.../>也有其他方法增加子节点。

➤➤6.5.3　为列表框、下拉菜单增加选项

为列表框、下拉菜单添加子节点，也就是为列表框、下拉菜单添加选项。添加选项有两种方法：

- 调用 HTMLSelectElement 的 add()方法添加选项。
- 直接为<select.../>的指定选项赋值。

HTMLSelectElement 包含如下方法用于添加新选项：

- add(HTMLOptionElement option, HTMLOptionElement before)：在 before 选项之前添加 option 选项。如果想将 option 选项添加在最后，将 before 指定为 null 即可；或者依然使用之前介绍的 appendChild(option)添加亦可。

下面的代码示范了通过这种方式来添加选项：

程序清单：codes\06\6.5\addOption.html

```
<body id="test">
<script>
    //创建<select.../>对象
    var a = document.createElement("select");
    //为<select.../>对象增加 10 个选项
    for (var i = 0 ; i < 10 ; i++)
    {
        //创建一个<option.../>元素
        var op = document.createElement("option");
        op.innerHTML = '新增的选项' + i;
        //将新的选项添加到列表框的最后
        a.add(op , null);
    }
    //设置列表框高度为 5
    a.size = 5;
    //将列表框增加成 body 元素的子节点
    document.getElementById("test").appendChild(a);
    </script>
</body>
```

在 Firefox 中浏览该页面，可看到如图 6.13 所示的效果。

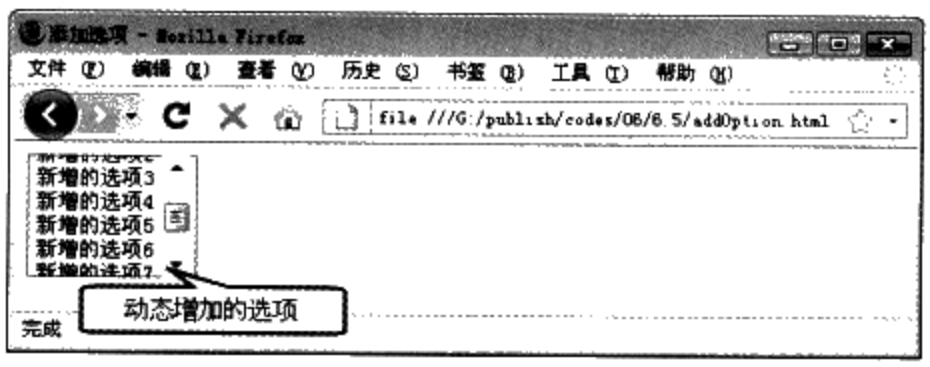


图 6.13　列表框动态增加选项

上面的页面程序在 Internet Explorer 6 中将出现错误，主要因为它不允许调用 add()方法时指定最后一个参数为 null。为了避免这种情况，我们可使用直接为指定选项赋值的方法来添加选项。

为指定选项赋值所支持的值必须是一个有效的选项，创建选项除了可使用前面所示的 createElement()方法之外，还可使用如下构造器：

- new Option(text, value, defaultSelected, selected)

该构造器有四个参数，这四个参数说明如下：

- text：该选项的文本，即该选项所呈现的“内容”。
- value：选中该选项的值。
- defaultSelected：设置默认是否选中该选项。
- selected：设置该选项当前是否被选中。

并不是每次构造该选项都需要指定四个参数，也可以只指定一个参数或者两个参数。如果构造 Option 对象时只指定了一个参数，则该参数是 Option 的 text 值；如果指定了两个参数，则第一个参数是 text，第二个参数是 value。

注意：

在 Internet Explorer 6 中运行时，如果直接为指定列表项赋值，则所赋值的<option.../>元素必须是通过 new Option()方法得到的，而不能是通过 document.createElementById("option")得到的。

下面的代码示范了利用第二种方法来为列表框、下拉菜单添加选项：

程序清单：codes\06\6.5\addOption2.html

```
<body id="test">
<script>
    //创建<select.../>对象
    var a = document.createElement("select");
    //为<select.../>对象增加 10 个选项
    for (var i = 0 ; i < 10 ; i++)
    {
        //创建一个<option.../>元素
        var op = new Option('新增的选项' + i , i);
        //直接为指定选项赋值
        a.options[i] = op;
    }
    //设置列表框高度为 5
    a.size = 5;
    //将列表框增加成 body 元素的子节点
    document.getElementById("test").appendChild(a);
    </script>
</body>
```

上面的页面代码既可以在 Firefox、Opera 中运行良好，也可以在 Internet Explorer 中运行良好。在 Internet Explorer 中浏览该页面，可看到如图 6.14 所示效果。

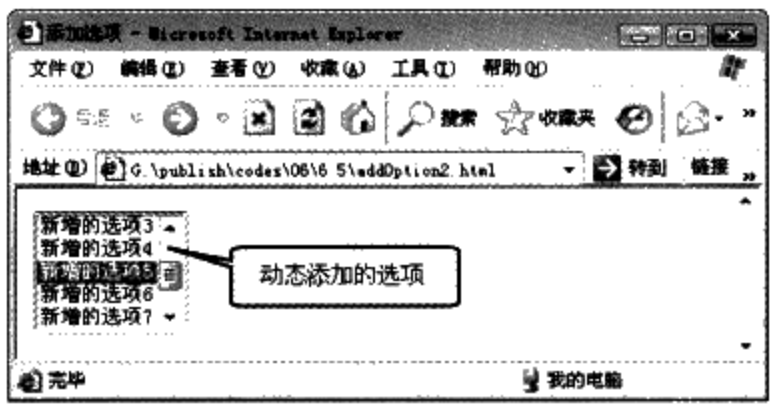


图 6.14　动态添加选项

➤➤6.5.4　动态添加表格内容

表格元素、表格行则另有增加子元素的方法。实际上，它们可以在添加子元素的同时创建这些子元素。也就是说，添加表格子元素时，往往无须使用 document 的 createElement()方法来创建节点。

HTMLTableElement 对象有如下方法：

- insertRow(index)：在 index 处插入一行。返回新创建的 HTMLTableRowElement。
- createCaption()：为该表格创建标题。返回新创建的 HTMLTableCaptionElement。如果该表格已有标题，则返回已有的标题对象。
- createTFoot()：为该表格创建<tfoot.../>元素。返回新创建的 HTMLTableFootElement。如果该表格已有<tfoot.../>元素，则返回已有的<tfoot.../>元素。
- createTHead()：为该表格创建<thead.../>元素。返回新创建的 HTMLTableHeadElement。如果该表格已有<thead.../>元素，则返回已有的<thead.../>元素。

HTMLTableRowElement 对象代表表格行，该对象包含如下方法用于插入单元格：

- insertCell(long index)：在 index 处创建一个单元格，返回新创建的单元格。

下面通过脚本在页面中动态生成一个表格：

程序清单：codes\06\6.5\addTd.html

```
<body id="test">
<script type="text/javascript">
    //创建一个表格对象
    var a = document.createElement("table");
    //设置表格的边框为 1
    a.border=1;
    var caption = a.createCaption();
    caption.innerHTML = "表格标题";
    //为表格循环插入 5 行
    for (var i = 0 ; i < 5 ; i++)
    {
        //插入行
        var tr = a.insertRow(i);
        //为每行循环插入 7 列
        for (var j = 0 ; j < 7 ; j++)
        {
            //循环插入 7 列
            var td = tr.insertCell(j);
            //设置每个单元格的内容
            td.innerHTML = "单元格内容 " + i + j;
        }
    }
    //将表格元素添加到 HTML 文档内
    document.getElementById("test").appendChild(a);
</script>
</body>
```

上面的代码中粗体字代码就是表格添加表格行、表格行添加单元格的关键代码，在浏览器中浏览该页面，可看到如图 6.15 所示效果。

图 6.15　脚本动态生成的表格

图 6.15 中的表格结构为 HTMLTableElement→HTMLRowElement→HTMLCellElement。每个表格元素包含若干个表格行子节点，每个表格行节点又包含若干个单元格子节点。整个表格看起来其实就是 DOM 树的子树。

6.6　删除 XHTML 元素

删除 XHTML 元素也是通过删除节点来完成的。对于普通的 XHTML 元素，可用通用方法来删除节点，而列表框、下拉菜单、表格，则有额外的方法来删除 XHTML 元素。

➤➤6.6.1　删除节点

删除节点通常借助于其父节点，Node 对象提供了如下方法来删除子节点：

➢ removeChild(oldNode)：删除 oldNode 子节点。

在从父节点中删除该子节点后，该子节点代表的内容也会消失。下面的代码通过控制 HTML 增加、删除节点来使页面中的表格出现、隐藏。代码如下：

程序清单：codes\06\6.6\removeChild.html

```
<body id="test">
    <input id="add" type="button" value="增加" disabled="disabled"
        onclick="add();"/>
    <input id="del" type="button" value="删除"
        onclick="del();"/>
    <div id="target" style="width:240px; height:50px; border:1px solid black">
        被控制的目标元素
    </div>
    <script type="text/javascript">
        //获取 body 元素
        var body = document.getElementById("test");
        //获取被控制的目标元素
        var target = document.getElementById("target");
        function add()
        {
            //添加目标元素
            body.appendChild(target);
            document.getElementById("add").disabled = "disabled";
            document.getElementById("del").disabled = "";
        }
        function del()
        {
            //删除目标元素
            body.removeChild(target);
            document.getElementById("del").disabled = "disabled";
            document.getElementById("add").disabled = "";
        }
    </script>
</body>
```

在浏览器中浏览该页面，可看到有“增加”和“删除”两个按钮，单击“删除”按钮，将看到如图 6.12 所示页面。

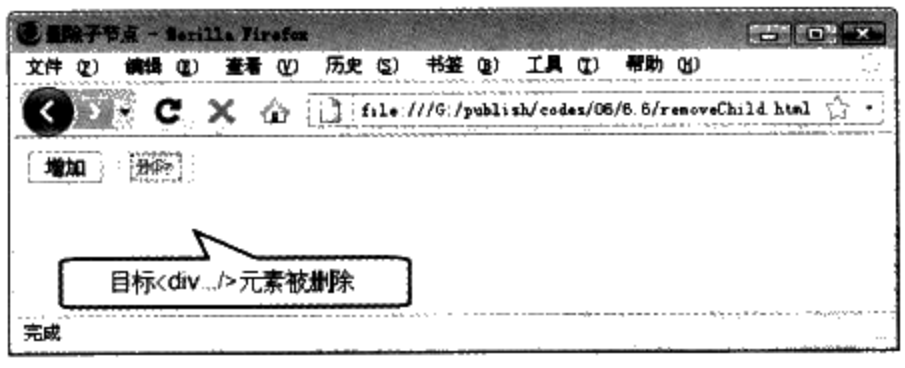


图 6.16 删除页面

与之对应的是，<select.../>元素和<table.../>元素也为删除子节点提供了更简便的操作方法。

➤➤6.6.2 删除列表框、下拉菜单的选项

删除列表框、下拉菜单的选项有两种方法：

➢ 利用 HTMLSelectElement 对象的 remove()方法删除选项。

➢ 直接将指定选项赋为 null 即可。

对于 HTMLSelectElement 对象而言，它提供了如下方法用于删除选项：

➢ remove(long index)：删除指定索引处的选项。

上面的方法中的 index 是需要删除选项所在的索引值。如果该索引值比下拉列表中的选项的最大索引还大，或者索引小于 0，则该方法不会删除任何选项。下面的页面演示了动态增加下拉列表的选项，并可以删除下拉列表的选项。页面代码如下：

程序清单：codes\06\6.6\removeOption.html

```
<body>
    <input id="opValue" type="text"/>
    <input id="add" type="button" value="增加" onclick="add();"/>
    <input id="del" type="button" value="删除" onclick="del();"/><br />
    <select id="show" size="8" style="width:120px;">
    </select>
<script>
    var show = document.getElementById("show");
    //增加下拉列表选项的函数
    function add()
    {
        //以文本框的值创建一个<option.../>元素
        var op = new Option(document.getElementById('opValue').value);
        //增加选项
        show.options[show.options.length] = op;
    }
    function del()
    {
        //如果有选项
        if (show.options.length > 0)
        {
            //删除最后的一个选项
            show.remove(show.options.length - 1 );
        }
    }
</script>
</body>
```

在浏览器中浏览该页面，在文本框中输入一个值，单击“添加”按钮就可将其添加到下拉列表中；如果单击“删除”按钮，将可删除最新增加的一个选项，浏览的效果如图 6.17 所示。

图 6.17　动态增加、删除选项

除此之外，直接将指定选项赋为 null 也可删除该选项，因此可以将上面程序中的 del()函数改为如下形式：

程序清单：codes\06\6.6\removeOption2.html

```
function del()
{
```

```
    //如果有选项
    if (show.options.length > 0)
    {
        //删除最后的一个选项
        show.options(show.options.length - 1 ) = null;
    }
}
```

两个页面的浏览效果完全相同。

提 示：

如果想删除某个列表框、下拉菜单的全部选项，没有必要采用循环的方式逐一删除每个选项，将列表框或下拉菜单 innerHTML 属性赋为 null，即可一次删除该列表框、下拉菜单的全部选项。

6.6.3 删除表格的行或单元格

删除表格的指定表格行使用 HTMLTableElement 对象的如下方法：

- deleteRow(long index)：删除表格中 index 索引处的行。

删除表格行的指定单元格使用 HTMLRowElement 对象的如下方法：

- deleteCell(long index)：删除某行 index 索引处的单元格。

下面的代码可以动态删除页面中的表格行，也可以动态删除表格中的单元格：

程序清单：codes\06\6.6\removeTable.html

```
<table id="test" border="1" width="400px">
    <caption>疯狂 Java 体系</caption>
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>轻量级 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 Ajax 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 XML 讲义</td>
        <td>疯狂 Workflow 讲义</td>
    </tr>
</table>
<script type="text/javascript">
    //获取目标表格
    var tab = document.getElementById("test");
    //删除行的函数
    function delrow()
    {
        if (tab.rows.length > 0)
        {
            //删除最后一行
            tab.deleteRow(tab.rows.length - 1);
        }
    }
    //删除目标表格的最后一格
    function delcell()
    {
        //获取表格的所有行
        var rowList = tab.rows;
        //获取表格的最后一行
        var lastRow = rowList.item(rowList.length - 1);
```

```
            if (lastRow.cells.length > 0)
            {
                //删除表格的最后一格
                lastRow.deleteCell(lastRow.cells.length - 1);
            }
        }
    </script>
    </body>
```

该页面在浏览器中浏览的效果如图 6.18 所示。

图 6.18 动态删除表格的行或列

6.7 传统 DHTML 模型

DOM 技术被 Internet Explorer 5 及以上版本的浏览器所支持，它采取一种非常直观且一致的方式来访问、导航和操作 XHTML 页面。通过 DOM 技术，不仅能够访问和更新页面的内容及结构，而且还能操纵文档的风格样式。

在 DOM 出现以前，JavaScript 采用传统的 DHTML 模型来访问、动态更新 XHTML 页面。在 DHTML 模型里，各元素之间有严格的包含关系，JavaScript 脚本可通过它们的 id 和 name 属性来访问它们。使用 DHTML 对象模型访问和更新 XHTML 页面时，不可避免地需要查询相关技术手册。因为 XHTML 元素很多，每个 XHTML 元素都有很多独有的属性、方法和事件。

相比之下，DOM 比 DHTML 对象模型功能更强大，它提供了对整个 XHTML 文档的访问模型，而不再局限于单一的 XHTML 元素。DOM 将文档转换为树形(Tree)结构，树的每个节点对应成 XHTML 元素。树形结构精确地描述了 XHTML 文档的元素间以及文本项间的相互关联，这种关联性包括父子节点关系、兄弟节点关系等。

在采用 DOM 技术访问和更新 XHTML 页面内容时，任何手册都可以放在一边。我们先查看一下 XHTML 源代码，推算出页面的 DOM 结构模型；然后，按照节点关系导航到指定节点，再修改其指定属性即可。

DHTML 对象模型则不包含 Tree 结构，因此不具备页面对象的相互导航功能。当我们从一个 XHTML 元素开始时，不能用父子节点、兄弟节点的关系来导航到指定元素。例如采用 DHTML 对象模型访问<table.../>中的指定单元格（<cell.../>）内容时，首先要确定单元格所在的行、列，然后再通过<table.../>元素提供的特殊方法来访问指定单元格。

注意：

虽然理论上 DOM 模型和 DHTML 模型存在一定的差异，但实际应用中我们不会区分哪种方式属于 DOM 模型，哪种方式属于 DHTML 模型。编写 JavaScript 脚本本来就是一件乏味、挫折感极强的事情：互相冲突的浏览器环境，缺乏有效的调试机制……所以在我们编写 JavaScript 脚本时，考虑最多的还是如何跨浏览器，如何保持高性能。

在 DHTML 模型里，window 对象代表整个窗口，该窗口可以是浏览器窗口，也可以只是浏览器页面内的一个 Frame。

DHTML 虽然没有提供一种完备的树形结构，却也提供了一种简单的方法来访问页面中各种子元素，这种访问主要借助于 DHTML 的包含关系来实现，DHTML 对象模型的包含关系如图 6.19 所示。

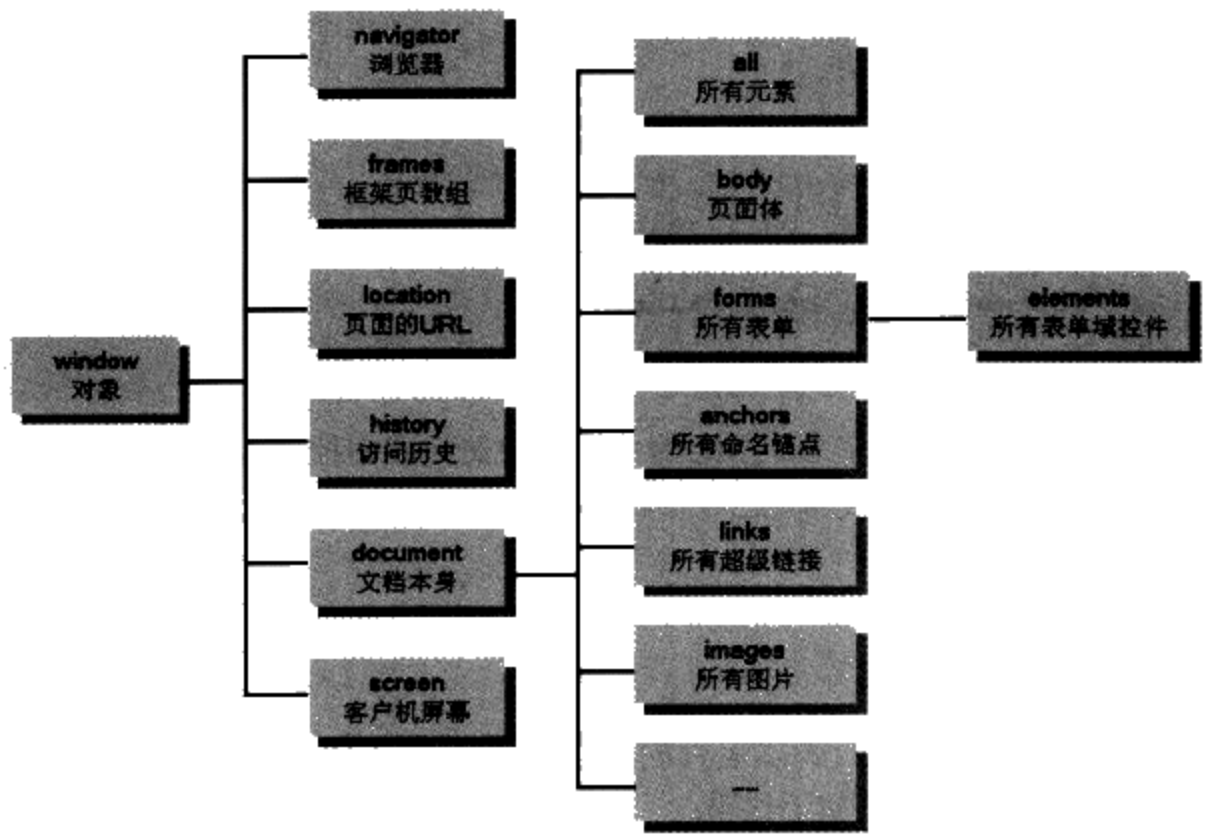


图 6.19　DHTML 包含关系图

如图 6.19 所示，DHTML 对象模型中，window 对象是整个对象模型的顶层对象，该对象包含一个 document 属性，该属性代表该窗口内的 HTML 文档，如果该窗口内有多个 Frame，则可使用 frames[] 方法依次访问该窗口的每个 Frame。

document 对象代表 HTML 文档本身，document 对象又包含了一系列的属性：forms、anchors、links、images……这些属性返回值以关联数组的形式存在，为了访问文档内指定控件，访问这些属性数组的指定元素即可，访问页面控件有如下三种语法：

- document.images[0]：返回页面内第一个图片元素。
- document.images[id]：返回页面内 id 或 name 为 id 的图片对象。
- document.images.id：返回页面内 id 或 name 为 id 的图片对象。

而 document 的 all 属性则比较独特，它返回该页面内所有控件。在早期的 DHTML 对象模型中，all 属性可作为 getElementById()方法的替代，使用 all 属性时一样可采用上面三种语法格式来访问页面里的控件。

如下代码示范了利用 DHTML 对象模型来访问页面控件：

程序清单：codes\06\6.6\dhtml.html

```
<body id="bd">
    <a href="http://www.leegang.org">疯狂 Java 联盟</a><br />
    <img id="lee" src="http://www.leegang.org/images/GreenPark/logo.jpg"
        alt="疯狂 Java 联盟" /><br />
    <form>
        <input type="text" name="user" value="文本框"/><br />
        <input type="button" id='bn' value="按钮"/>
    </form>
    <script type="text/javascript">
        //访问 body 元素
```

```
            alert(document.body.id);
            //访问第一个超级链接
            alert(document.links[0].href);
            //访问 id 或 name 为 lee 的图片
            alert(document.images['lee'].alt);
            //访问页面的第一个表单
            form = document.forms[0];
            alert(form.innerHTML);
            //访问表单里第一个元素
            alert(form.elements[0].value);
            //访问表单里 id 或 name 为 bn 的元素
            alert(form.elements['bn'].value);
            //下面的代码在 Internet Explorer 6 中可行
            alert(document.all['bn'].value);
        </script>
    </body>
```

上面的代码中粗体字代码示范了 DHTML 对象模型中访问页面控件的方法，页面中最后一行粗体字代码使用 document 的 all 属性，该属性已经不被 Firefox 3 支持，但在 Internet Explorer 6 中还可以运行良好。

6.8 使用 window 对象

window 对象是整个 JavaScript 脚本运行的顶层对象。在定义一个全局变量时，该变量是作为 window 对象的一个属性存在的。看如下代码：

程序清单：codes\06\6.8\windowVar.html

```
<script>
    //定义全局变量 a
    var a = 5;
    //判断 window 对象的属性 a 和全局变量 a 是否相等
    alert(window.a === a);
    //为 window 对象增加属性
    window.book = "疯狂 Ajax 讲义";
    //访问全局变量，将输出“疯狂 Ajax 讲义”
    alert(book);
</script>
```

我们将看到 window.a 和全局变量 a 是完全相等的，而且访问全局变量 book 时，实际上输出 window 的 book 属性值。

因此此处必须澄清一个概念：在定义了一个所谓的全局变量后，它仅仅在当前的 window 对象中具有全局性。如果在同一个页面里有多个 Frame，则意味着有多个 window 对象，且每个 window 中的全局对象不会互相影响。

实际上，不仅直接定义的全局变量将以 window 对象的属性存在，直接定义一个普通函数时，该函数也是作为 window 对象的方法存在的——也就是说，window 对象所包含的方法，JavaScript 脚本可以直接调用。window 提供了如下几个方法，这些方法可以在 JavaScript 脚本中直接使用：

- alert()、confirm()、prompt()：分别用于弹出警告对话框、确认对话框和提示输入对话框。
- close()：关闭窗口。
- focus()、blur()：让窗口获得焦点、失去焦点。
- moveBy()、moveTo()：移动窗口。
- open()：打开一个新的顶级窗口，用于装载新的 URL 所指向的地址，并可指定一系列的新属性，包括隐藏菜单等。

- print()：打印当前窗口或 Frame。
- resizeBy()、resizeTo()：重设窗口大小。
- scrollBy()、scrollTo()：滚动当前窗口中的 HTML 文档。
- setInterval()、clearInterval ()：设置、删除定时器。
- setTimeout()、clearTimeout()：也是设置定时器。推荐使用 setInterval()和 clearInterval ()。

除此之外，window 对象还提供了如下的常用属性，通过这些属性即可访问 window 对象包含的一系列的对象，例如 location、history 等：

- closed：该属性返回一个 boolean 值，用于判断该窗口是否处于关闭状态。
- defaultStatus、status：返回浏览器状态栏的文本。
- document：返回该窗口内装载的 XHTML 文档。
- frames[]：返回该窗口内包含的 Frame 对象，每个 Frame 对象又是一个 window 对象。
- history：返回该窗口的浏览历史。
- location：返回该窗口装载的 XHTML 文档的 URL。
- name：返回该窗口的名字。
- navigator：返回浏览当前页面的浏览器。
- parent：如果当前窗口是一个 Frame，则该属性返回包含本 Frame 的窗口，即该 Frame 的直接父窗口。
- screen：返回当前浏览者的屏幕对象。
- self：返回自身。
- top：如果当前窗口是一个 Frame，则该属性指向包含本 Frame 的顶级父窗口。

下面先看如何通过 status 改变窗口的状态栏文字。首先看如下的简单代码：

程序清单：codes\06\6.8\status.html

```
<script>
    //修改窗口的状态栏文字
    window.status="自定义状态栏文字";
</script>
```

如果在 Internet Explorer 中浏览该文档，将可以看到窗口的状态栏文字变成了“自定义状态栏文字”。如果结合定时器函数，将可以做出状态栏的动态文字效果。看下面的页面代码：

程序清单：codes\06\6.8\status2.html

```
<body onload="stack();">
<script type="text/javascript">
    //自定义的状态文字
    var statusText = "自定义的动画状态栏文字...";
    var out = "";
    //动画间隔时间
    var pause = 25;
    //动画宽度
    var animateWidth = 20;
    var position = animateWidth;
    var i = 0 ;
    function stack()
    {
        if (statusText.charAt(i) != " ")
        {
            out = "";
            //将 0 到 i-1 个字符拼成输出字符串
            for (var j=0; j<i; j++)
            {
```

```
            out += statusText.charAt(j);
        }
        //增加一定宽度空格
        for (j=i; j<position; j++)
        {
            out += " ";
        }
        //将第 i 个字符添加到输出字符串里去
        out += statusText.charAt(i);
        for (j=position; j<animateWidth; j++)
        {
            out += " ";
        }
        window.status = out;
        //如果后出来的字紧靠了前面字符串
        if (position == i)
        {
            animateWidth++;
            position = animateWidth;
            //i 加 1，对应为多出现一个字符
            i++;
        }
        else
        {
            position--;
        }
    }
    else
    {
        i++
    }
    if (i < statusText.length)
    {
        setTimeout("stack()",pause);
    }
}
</script>
</body>
```

如果在 Internet Explorer 中浏览该文档，将可看到如图 6.20 所示的状态栏效果。

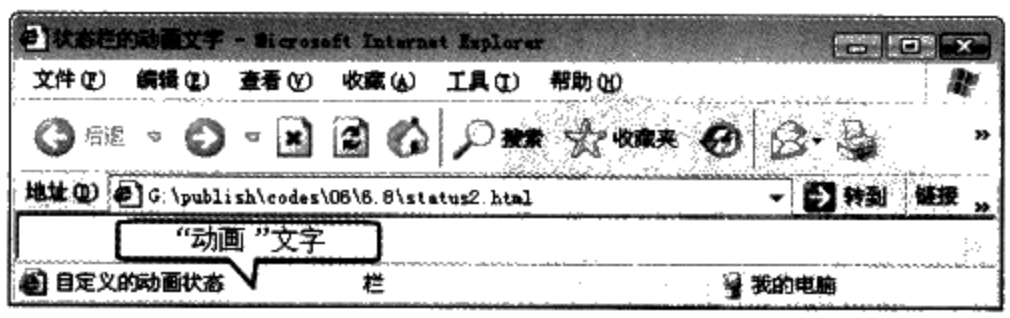


图 6.20　状态栏的动画文字

➤➤6.8.1　访问历史

window 的 history 属性是一个 History 对象，该对象表示当前窗口的浏览历史，支持如下几个方法：

- back()：后退到上一个浏览的页面，如果该页面是第一个打开，则该方法没有任何效果。
- foward()：前进到下一个浏览页面，前提是之前使用了 back 或 go 方法。
- go(intValue)：该方法可指定前进或后退多少个页面，其中的 intValue 控制前进、后退的页面数。其中 intValue 为正，表示前进；intValue 为负，表示后退。

由于 History 对象的使用非常简单，此处不再给出示范代码。

➤➤6.8.2 浏览器对象

window 对象有个 navigator 属性，该属性对应于 Navigator 对象，该对象代表浏览该页面所使用的浏览器，该对象在不同平台上的信息并不完全相同，但总包含了如下几个常用的属性：

- appName：返回该浏览器的内核名称。
- appVersion：返回该浏览器当前的版本号。
- platform：返回当前浏览器所在的操作系统。

当然，我们没有必要记住它到底包含了多少个属性，可以通过如下简单的代码测试它在对应平台下所包含的属性以及属性值。

程序清单：codes\06\6.8\navigator.html

```
<script>
alert(window.navigator);
var browser = "当前的浏览器信息是:\n";
//遍历该浏览器的全部属性
for(var propname in window.navigator)
{
    //将所有属性名、属性值连缀在一起
    browser += propname + ": " + window.navigator[propname] + "\n"
}
alert(browser);
</script>
```

在 Firefox 中浏览该页面，可看到如图 6.21 所示的对话框。

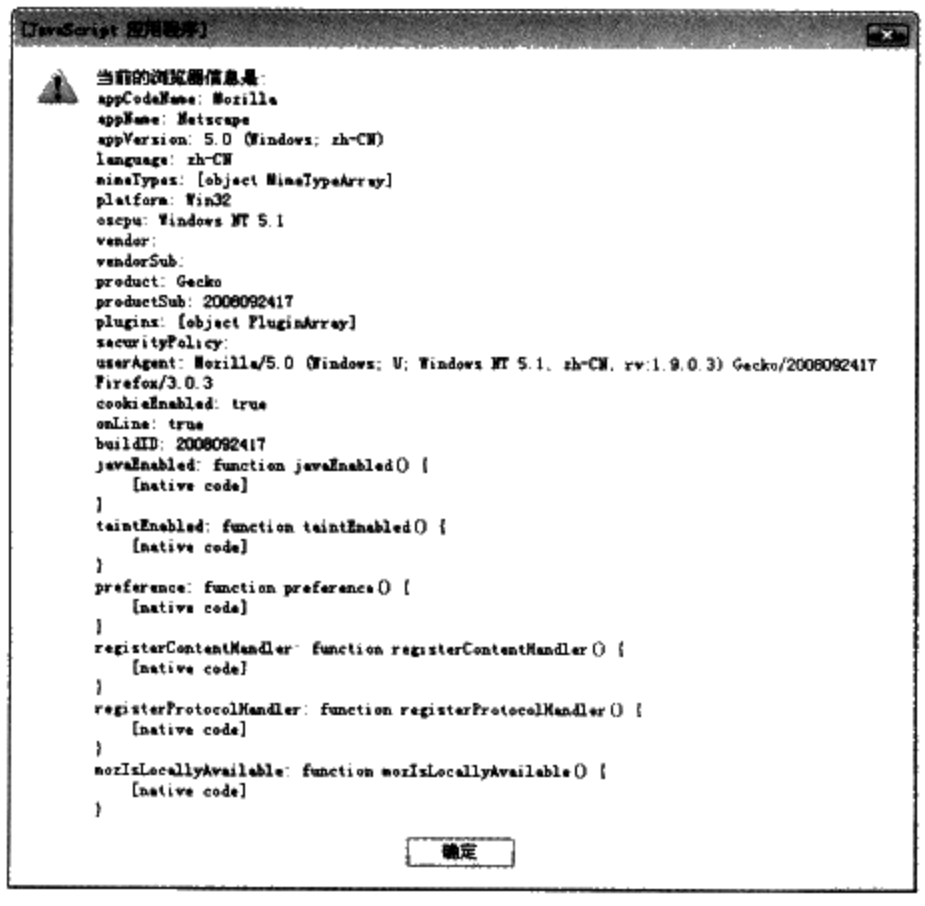


图 6.21 访问当前浏览器的信息

➤➤6.8.3 访问页面 URL

window 对象还包含了一个 location 属性，该属性可用于访问该窗口或 Frame 所装载文档的地址。location 对象还包含了如下几个常用属性：

- hostname：文档所在地址的主机名。
- href：文档所在地址的 URL 地址。
- host：文档所在地址的主机地址。
- port：文档所在地址的服务端口。
- pathname：文档所在地址的文件地址。
- protocol：装载该文档所使用的协议，例如 http:等。

下面的代码示范了访问 location 对象的常用属性：

程序清单：codes\06\6.8\location.html

```
<script>
    var loc = window.location;
    var locStr = "当前的 location 信息是:\n";
    //遍历 location 对象的全部属性
    for (var propname in loc)
    {
        locStr += propname + ": " + loc[propname] + "\n"
    }
    alert(locStr);
</script>
```

➤➤6.8.4 客户机屏幕信息

window 对象有一个 screen 属性，它返回当前浏览者的屏幕对象，可用于获取用户屏幕当前的大小、色深、屏幕分辨率等参数。该对象的属性也随不同的平台存在变化，但通常会包含如下属性：

- width：屏幕的横向分辨率。
- height：屏幕的纵向分辨率。
- colorDepth：当前屏幕的色深。

当然，通常没有必要记住该对象到底包含了多少属性，可通过如下简单代码测试该对象到底包含了多少属性：

程序清单：codes\06\6.8\screen.html

```
<script>
alert(window.screen);
var screen = "当前的屏幕信息是:\n";
//遍历 screen 对象的所有属性
for(var propname in window.screen)
{
    screen += propname + ": " + window.screen[propname] + "\n"
}
alert(screen);
</script>
```

在浏览器中浏览该页面，可看到如图 6.22 所示的对话框。

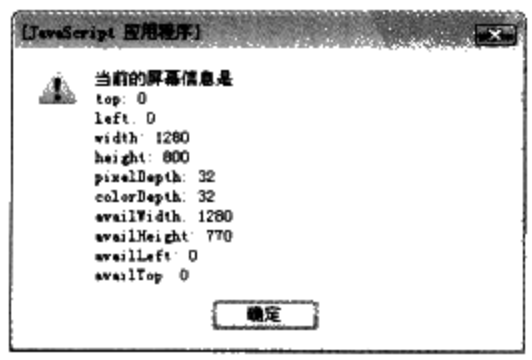


图 6.22 屏幕相关信息

➤➤6.8.5 弹出新窗口

window 的 open()方法用于打开一个新窗口。如果结合 screen 对象的属性，可将打开的窗口放大到满屏，形成满屏效果。看如下简单代码：

程序清单：codes\06\6.8\open.html

```
<script>
//获取当前屏幕的大小
width=window.screen.width;
height=window.screen.height;
//打开一个新的满屏窗口
window.open("status.html", "_blank", "left=0, top=0, width="+ width +
    ", height="+height+", toolbar = no, menubar = no, resize = no");
//关掉自身
window.close();
</script>
```

当然，这种 open 方法可以被像 Firefox 等一些浏览器禁用。大量的弹出窗口是对浏览者耐心的挑战，因此尽量少用弹出式窗口。

➤➤6.8.6 确认对话框和输入对话框

在前面 JavaScript 代码中大量使用了 alert()方法弹出对话框。实际上 window 对象还提供了两种对话框：用于取得用户确认（confirm）的确认对话框，用于获得用户输入（prompt）的输入对话框。

confirm()方法弹出一个确认对话框，返回一个 boolean 值。如果用户单击了“确定”按钮，将返回 true；如果用户单击了“取消”按钮，则返回 false。

看下面的简单代码，该代码使用 confirm 对话框取得用户确认，确认是否使用超级链接导航到下一个页面。

```
<a href="http://www.leegang.org"
    onClick="return confirm('请确认是否导航到疯狂 Java 联盟');">疯狂 Java 联盟</a>
```

在浏览器中浏览该页面，单击该页面中的超级链接，可看到如图 6.23 所示的对话框。

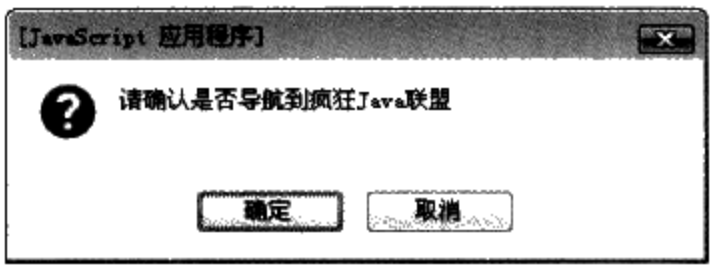


图 6.23 confirm 的确认对话框

prompt()方法则弹出一个输入对话框，该对话框可获取用户的输入，返回用户输入的内容。看下面的简单代码：

程序清单：codes\06\6.8\open.html

```
<body>
你的名字是：<span id="name"></span>
<script type="text/javascript">
    name = prompt("请输入你的名字：" ,"");
    document.getElementById("name").innerHTML = name;
</script>
</body>
```

在浏览器中浏览该页面，可看到如图 6.24 所示的对话框。

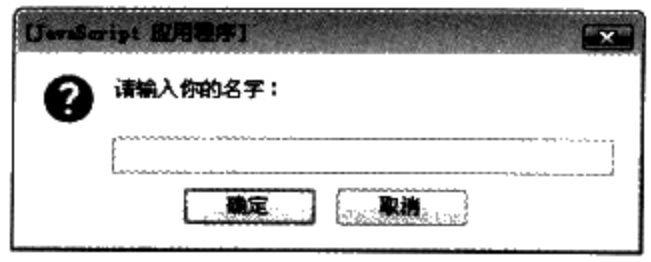


图 6.24　使用 prompt 对话框获取用户输入

➤➤6.8.7　使用定时器

window 提供了如下四个方法来支持定时器：

- setInterval("code" , interval)、clearInterval (timer)：设置、删除定时器。setInterval 设置每隔 interval 毫秒后重复执行 code。
- setTimeout("code;" , interval)、clearTimeout(timer)：也是设置定时器。推荐使用 setInterval() 和 clearInterval ()。setTimeout 设置在 interval 毫秒延迟后执行一次 code。

如果需要让一段代码、一个 JavaScript 函数以固定频率重复执行，则应该使用 setInterval()函数；如果需要让一段代码、一个 JavaScript 函数在指定延迟后仅仅执行一次，则应该使用 setTimeout()函数。

下面的页面示范了一个简单“动画”效果：

程序清单：codes\06\6.8\timer.html

```
<body onload="setTime();">
<span id="tm"></span>
<script type="text/javascript">
    //定义定时器变量
    var timer;
    //保存页面运行的起始时间
    var cur = new Date().getTime();
    function setTime()
    {
        //在 tm 元素中显示当前时间
        document.getElementById("tm").innerHTML =
            new Date().toLocaleString();
        //如果当前时间比起始时间大于 60 秒，停止定时器的调度
        if (new Date().getTime() - cur > 60 * 1000)
        {
            //清除 timer 定时器
            clearInterval(timer);
        }
    }
    //指定每隔 1000 毫秒执行 setTime()函数一次
    timer = window.setInterval("setTime();" , 1000);
</script>
</body>
```

从上面的页面代码中可以看出，setInterval()定时与 Java 的定时器基本相似，只是 setInterval()是控制一条或多条代码以指定时间间隔重复执行；而 Java 定时器控制事件监听器以指定时间间隔不断被触发。

实际上，上面的代码也可改为使用 setTimeout()方法来实现，看如下代码：

```
<body>
<span id="tm"></span>
<script type="text/javascript">
    //定义定时器变量
    var timer;
    //保存页面运行的起始时间
    var cur = new Date().getTime();
```

```
        function setTime()
        {
            //在 tm 元素中显示当前时间
            document.getElementById("tm").innerHTML =
                new Date().toLocaleString();
            //如果当前时间比起始时间小于等于 60 秒，执行定时器的调度
            if (new Date().getTime() - cur <= 60 * 1000)
            {
                //指定延迟 1000 毫秒后执行 setTime()函数。
                window.setTimeout("setTime();" , 1000);
            }
        }
        //直接调用 setTime()函数
        setTime();
    </script>
    </body>
```

上面的代码需要直接调用 setTime()函数，一旦 setTime()函数执行起来后，在一秒钟内，该函数将会重复执行——因为 setTime()函数的最后一行调用了 setTimeout("setTime();" , 1000);，该代码指定在 1 秒之后再次执行 setTime()函数。

提 示:

对于 setTimeout()和 setInterval()定时器的区别，我们可以举一个现实生活中的例子：假如有位先生希望周期性地和某位小姐约会，他有两种实现方式：第一种方式是制订一个约会时间表，比如每隔一天就约会一次，这样只需每次到时间进行约会即可。第二种方式需要先获取第一个约会，然后每次约会结束后再次约定下次约会的时间，这种方式需要每次约会结束时重新约定下次约会时间。setInterval()定时器采用的是第一种方式；而 setTimeout()则采用了第二种方式。

6.9 使用 document 对象

document 对象既是 HTMLDocument 类的一个实例，也是 DHTML 模型中的一个对象。因此，JavaScript 的 document 既可以作为 HTMLDocument 使用，也可以作为 DHTML 的 document 使用。该对象除了可使用标准 DOM 模型的方法，还可以使用如下几个常用方法：

- close()：结束一个通过 open 方法打开的 document 对象。
- open()：打开一个 document 对象。
- write()：向 document 对象中输出一条字符串，输完后不换行。
- writeln()：向 document 对象中输出一条字符串，输完后换行。

除此之外，还有如下常用属性：

- alinkColor、linkColor、vlinkColor、bgColor、fgColor：五个颜色属性，分别对应 HTML 文档中超级链接激活时的颜色、没有访问过的超级链接的颜色、访问过的超级链接的颜色、背景色和前景色。
- all：该属性返回该文档中的所有子元素。
- anchors：该属性返回文档中的所有命名锚点数组。
- applets：该属性返回文档中所有的 Applet 数组。
- cookie：该属性用于读写 HTTP cookie。
- documentElement：该属性返回文档的根元素。通常就是返回<html.../>元素。
- forms：该属性返回该 document 包含的全部表单数组。
- frames：该属性返回该 document 包含的全部 Frame 集合。
- images：该属性返回该 document 包含的全部图像数组。

- lastModified：该属性返回该 document 的最后修改时间。
- links：该属性返回该 document 内包含的全部超级链接的数组。
- location：该属性的作用类似于 URL。
- referrer：返回上一个页面的 URL，且上一个页面中的超级链接负责导航当前页面。
- scripts：该属性返回该 document 对象中包括的所有脚本的数组。
- styleSheets：该属性返回该 document 对象的全部 CSS 样式的数组。
- title：该属性返回 document 对象标题，就是<title>和</title>之间的部分。
- URL：该属性返回 document 所在 URL，该属性的值与 location 的 href 属性值相同。

6.9.1 动态页面

借助于 document 对象的 open 和 write 方法，可以动态生成一个页面，看下面的代码：

程序清单：codes\06\6.9\dynaDocument.html

```
<body>
<script>
//计数器
var n = 0;
var win = null;
//用于显示弹出窗口显示提示信息的函数
function show(msg)
{
    //判断弹出窗口是否为空
    if ((win == null) || (win.closed))
    {
        //打开一个新的弹出窗口
        win = window.open("","console","width=400,height=250,resizable");
        //将弹出窗口的文档打开成一个普通文档，而不是一个 text/html 文档
        win.document.open("text/plain");
    }
    //让弹出窗口得到焦点
    win.focus();
    //在弹出窗口装载的文档中输出信息
    win.document.writeln(msg);
    win.doucment.close();
}
</script>
<!-- 激发事件的按钮 -->
<input type="button" value="单击"
    onclick="show('您单击了按钮：' + ++n + '次。');">
</body>
```

上面的页面代码中粗体字代码可打开一个新的文档，每次调用 document.writeln()方法即可动态改变该文档的内容。在浏览器中浏览该页面，多次单击“单击”按钮，即可看到如图 6.25 所示窗口。

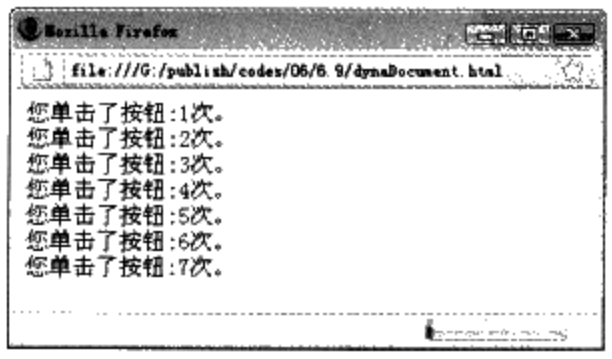


图 6.25 动态生成的文档

➤➤6.9.2 读写 Cookie

Cookie 是一些 name=value 对数据，这些数据可以由浏览器写入客户机硬盘，也可由浏览器从客户机硬盘读取。Cookie 通常用于持久记录客户的某些信息，比如客户的用户名及客户的喜好等，因而可把 Cookie 当成一种简单的数据持久化方法。

通常而言，读写 Cookie 都是由服务器程序（比如 JSP 页面或 Servlet 等）控制，但实际读写 Cookie 的依然是浏览器，而 JavaScript 一样可以控制浏览器读写 Cookie。

使用 JavaScript 控制浏览器写 Cookie 很简单，直接给 document.cookie 属性赋值即可，这个属性值必须为如下格式：

<name>=<value>

上面的各种<name>和<value>都可由开发者任意指定。除此之外，添加 Cookie 时还可指定如下几个属性：

- max-age：指定该 Cookie 存活的最长有效期。
- expires：指定 Cookie 的过期时间。
- path：指定该 Cookie 的路径。
- domain：指定该 Cookie 属于哪个域。
- secure：指定该 Cookie 的安全属性。

下面的代码通过 document.cookie 写入 Cookie，并指定该 Cookie 最长有效期为一年：

```
document.cookie = "name=yeeku; max-age=" + (60*60*24*365);
```

下面的代码通过 document.cookie 写入 Cookie，并指定该 Cookie 最长有效期为一年，而且该 Cookie 属于 leegang.org 域：

```
document.cookie = "name=yeeku; max-age=" + (60*60*24*365)
    + ";domain=leegang";
```

读取 Cookie 则略微复杂一点，需要先访问 document.cookie 属性，该属性返回一个字符串，然后使用 JavaScript 脚本分析该 Cookie 字符串即可。下面的代码示范了如何写入、读取 Cookie：

程序清单：codes\06\6.9\Cookie.html

```
<script>
function setCookie(name , value)
{
    //定义变量，保存当前时间
    var expdate = new Date();
    //将 expdate 的月份 + 1
    expdate.setMonth(expdate.getMonth() + 1);
    //添加 Cookie
    document.cookie = name + "=" + escape(value) ;
        + "; expires=" + expdate.toGMTString() + ";";
}
function getCookie(name)
{
    //访问 Cookie 的 name 开始处
    var offset = document.cookie.indexOf(name);
    //如果找到指定 Cookie
    if (offset != -1)
    {
        //从 Cookie 名后位置开始搜索
        offset += name.length + 1;
        //找到 Cookie 名后第一个分号（;）
        end = document.cookie.indexOf(";", offset) ;
        //如果没有找到分号
```

```
            if (end == -1)
            {
                end = document.cookie.length;
            }
            //截断字符串中 Cookie 的值
            return unescape(document.cookie.substring(offset, end));
        }
        else
        {
            return "";
        }
    }
    setCookie('user' , 'yeeku');
    alert(getCookie('user'));
    </script>
```

上面的页面代码的第一行粗体字代码用于添加 Cookie，第二行粗体字代码用于读取 Cookie，从页面代码中可以看出，添加 Cookie 就是为 document.cookie 属性赋值；读取 Cookie 就是截取 document.cookie 属性值的合适子串。

6.10 两个常用范例

下面介绍两个常用的动态脚本范例，这两个范例对于大部分的 Web 应用都是相当常见的，读者朋友可以参考这两个范例完成更加完善、实用的动态脚本。

➤➤6.10.1 可编辑表格

在 6.4 节介绍了一个可编辑的表格，这个可编辑的表格修改单元格值时，先要输入需要修改的单元格的坐标（单元格所在的行、列），其实这是相当不方便的。因为我们不可能让用户每次操作表格之前，先去设定要修改的单元格所在的行、列——这是相当不友好的用户界面。比较理想的表格是像 Excel 软件一样的表格编辑器。

理想的可编辑表格是：在用户双击某个单元格后，单元格的值即可手动输入，输入完后单元格的值就存在于该单元格内。如果该表格的数据连接着数据库，也可做到与数据库同步：每次用户修改完表格数据之后，即将表格对应于数据库的值修改到与表格中的显示一致。

如果不需要与数据库同步，则可借助于 JavaScript 的效果实现，其思路是：每次用户双击单元格之后，单元格的值被动态设置为空，并在单元格内即时插入一个文本框，用于接受用户输入。用户在单元格内的文本框内输入相应的值，且文本框失去焦点之后，将单元格的文本框清除，并将文本框的值作为前单元格的值显示出来，整个过程借助的是动态修改 HTML 元素的属性。

如果需要表格的值与数据库状态实时同步，则可借助 Ajax 技术实现。这种实现有三种解决策略：

- 第一种：每次单元格内可编辑的文本框失去焦点之时，使用 XMLHttpRequest 向服务器发送请求，将单元格修改后的数据发送到服务器，通知服务器完成数据的同步。这种方式最简单且容易实现，但效果不太好。因为如果用户频繁修改表格数据，则服务器的负担是比较重的。
- 第二种：每个单元格被修改后将其保留在一个数组内，用户修改了多个单元格后，一次性手动提交修改。这种方式需要在客户端开辟一个保存修改后值的单元格数组，但可降低服务器端的负担。这种方式还有个不好之处，是要用户手动提交，界面稍嫌烦琐——这种方式类似于 Excel 表格没有打开自动保存时的效果。
- 第三种：系统自动定时同步。每隔一段时间，系统自动启动 XMLHttpRequest 向服务器发送请求，完成表格数据与数据库数据的同步，这种方式类似于 Excel 表格的自动保存。

最理想的情况是第二种情况和第三种情况相结合的可编辑表格。因为现在还没有介绍

XMLHttpRequest 对象的使用。所以现在暂不介绍使用 XMLHttpRequest 发送请求并请求数据库同步的功能，此处的示例只是完成客户端的可编辑表格，代码如下：

程序清单：codes\06\6.10\editable_table.html

```
<body>
<table id="test" border="1" width="400px">
    <tr>
        <td>疯狂 Java 讲义</td>
        <td>轻量级 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 Ajax 讲义</td>
        <td>经典 Java EE 企业应用实战</td>
    </tr>
    <tr>
        <td>疯狂 XML 讲义</td>
        <td>疯狂 Workflow 讲义</td>
    </tr>
</table>
<script>
    //双击单元格后，在单元格中出现的输入文本框
    var tmpIn = document.createElement("input");
    tmpIn.type="text";
    //双击单元格后的当前单元格
    var curCell;
    //生成可编辑的单元格
    function edit(event)
    {
        //此处是为了兼顾 IE 浏览器和 Firefox 浏览器
        //因为它们所使用的事件机制不同
        //目标是获取被双击的单元格
        if( event == null )
        {
            curCell = window.event.srcElement;
        }
        else
        {
            curCell = event.target;
        }
        //将单元格的值填充到文本输入框的值
        tmpIn.value=curCell.innerHTML;
        //在文本框失去焦点时触发 end 函数
        tmpIn.onblur= end;
        //清空当前单元格的内容
        curCell.innerHTML = "";
        //将文本框添加到当前单元格内
        curCell.appendChild(tmpIn);
    }
    //编辑单元格结束后的函数
    function end()
    {
        //将文本框内的值赋给当前单元格
        curCell.innerHTML=tmpIn.value;
    }
    //为表格的双击事件绑定事件处理函数
    document.getElementById("test").ondblclick = edit;
</script>
</body>
```

通过上面的页面代码可以看出，实现可编辑表格的过程其实相当简单。过程是：双击单元格时，将单元格改变为一个单行文本框，文本框可接受用户输入；文本框失去焦点后，再将文本框的值赋给单元格即可。

提 示：

笔者在教学过程中，往往见到有些学员看到一个应用之后，“吓”得完全不敢动手。其实这只是一种假相，即使看起来非常复杂的应用，只要我们一步一步去实现，最终完全是可实现的。有时候，笔者不断地告诉学员，写程序真的是很“简单”的事情，只要你一行代码一行代码地积累，最终一定可以成为一个高手。为读者的情绪着想，笔者写书过程中宁愿舍弃一些代码的复用性，尽可能将代码简化简化再简化，让整个范例只突出想表达的主题，希望读者不要畏惧任何项目的代码，就笔者自己实现 Web 容器的经历而言，再大的项目都需要一行代码一行代码地积累，最终一定可以完成一个看起来非常“壮观”的应用。这就是所谓“积土成山，风雨兴焉”。

该页面在 Internet Explorer 和 Firefox 浏览器中都可以得到非常好的效果，该表格编辑后的效果如图 6.26 所示。

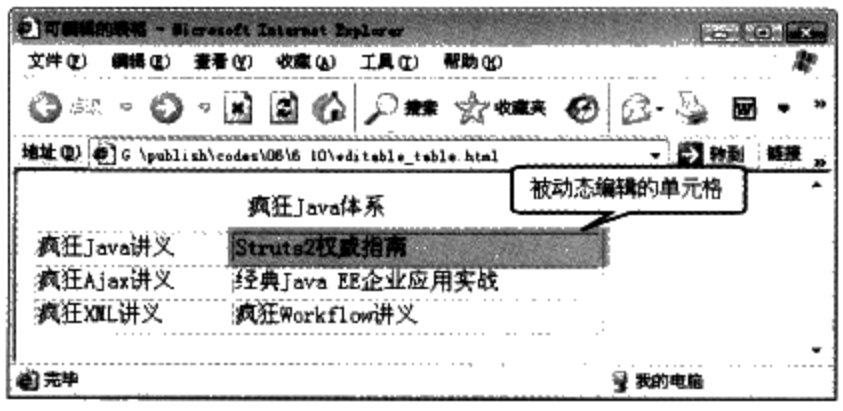


图 6.26　仿 Excel 的可编辑表格

➤➤6.10.2　导航菜单

这也是个非常常见的效果，我们经常在一些 Web 页面上看到这种导航菜单，它模仿了 Windows 窗口的菜单效果。单击菜单条上的菜单名，可以弹出一个菜单，菜单里包含了若干菜单项，每个菜单项都可以导航到其他的页面。

导航菜单的实现并不难，其技术核心是需要实现<div.../>的立体效果，当鼠标移动到菜单条上的菜单上时，该菜单应该出现立体效果：关于立体效果在第 5 章已经介绍过了，就是为目标对象增加边框，让左、上边框的颜色稍浅，而右、下边框的颜色稍深即可。

单击菜单后，应该出现对应的菜单，菜单出现的最直接做法是修改内联 CSS 样式的 display 属性，设置 display 属性为"none"将隐藏该菜单，设置 display 属性为""将显示该菜单。下面是该导航菜单的页面代码：

程序清单：codes\06\6.10\menu.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 导航菜单 </title>
    <style type="text/css">
    /* 设置菜单条的 CSS 样式 */
    #menubar {
        width: 780px;
        height: 32px;
        background-color: menu;
        padding-top: 3px;
    }
```

```
/* 设置菜单条里菜单的CSS样式 */
#menubar div {
    height: 22px;
    width: 60px;
    font-size: 10pt;
    background-color: menu;
    text-align: center;
    vertical-align: middle;
    padding-top: 6px;
    cursor: default;
    /* 控制div元素不换行 */
    float:left;
}
/* 当鼠标在菜单条上悬停时的立体"凸出"效果 */
.menuhover {
    border : solid 1px;
    border-color:#ffffff #333333 #333333 #ffffff;
}
/* 当鼠标在菜单条上悬停时的立体"凹下"效果 */
.menuclick {
    border : solid 1px;
    border-color: #333333 #ffffff  #ffffff #333333;
}
/* 菜单的CSS样式 */
.menu {
    z-index:100;
    border : solid 1px;
    border-color:#ffffff #333333 #333333 #ffffff;
    position:absolute;
    width:182px;
    background-color:menu;
    font-size:10pt;
    cursor:default;
}
/* 鼠标悬停在菜单项上时菜单项的高亮效果 */
.menuitemhover {
    width:180px;
    height:22px;
    background-color:#000055;
    color:#ffffff;
    padding-top: 6px;
    padding-left: 2px;
}
/* 普通菜单项的效果 */
.menuitem {
    width:180px;
    height:22px;
    padding-top: 6px;
    padding-left: 2px;
}
</style>
<script>
//用于保存上次出现的菜单
var lastMenu;
//用于保存上次单击的菜单标题
var lastMenuTitle;
//菜单悬停时的立体效果
function menuHover(event)
{
    var targetMenu;
```

```
    if (event.srcElement != null)
    {
        targetMenu = event.srcElement;
    }
    else
    {
        targetMenu = event.target;
    }
    //设置鼠标悬停在菜单上时的效果
    targetMenu.className = "menuhover";
}
//鼠标移出菜单时的恢复效果
function menuGotOut(event)
{
    var targetMenu;
    if (event.srcElement != null)
    {
        targetMenu = event.srcElement;
    }
    else
    {
        targetMenu = event.target;
    }
    //设置鼠标移出菜单时的恢复效果,
    //如果该菜单已经处于打开状态，则保持菜单标题的下凹效果
    if (targetMenu.className != "menuclick")
    {
        targetMenu.className = "";
    }
}
//菜单项悬停时的高亮
function itemHover(event)
{
    //获得需要出现高亮的菜单项
    var targetMenu;
    if (event.srcElement != null)
    {
        targetMenu = event.srcElement;
    }
    else
    {
        targetMenu = event.target;
    }
    //让菜单项出现高亮效果
    targetMenu.className = "menuitemhover";
}
//菜单项移出时的恢复，取消高亮
function itemGotOut(event)
{
    //获取鼠标移出的菜单项
    var targetMenu;
    if (event.srcElement != null)
    {
        targetMenu = event.srcElement;
    }
    else
    {
        targetMenu = event.target;
    }
    //取消菜单项上的高亮效果
```

```
        targetMenu.className = "menuitem";
    }
    //菜单条上的菜单标题被单击时的事件处理函数
    function showMenu(name,obj)
    {
        //将上次单击的菜单标题的凹下效果取消
        if ( lastMenuTitle != null ) lastMenuTitle.className = "";
        //将当前单击的菜单标题设置为凹下效果
        obj.className="menuclick";
        //关闭上次出现的菜单的显示
        if ( lastMenu != null ) lastMenu.style.display = "none"
        //获取当前需要显示的菜单
        var mu = document.getElementById(name);
        //显示该菜单标题对应的菜单
        mu.style.display="";
        //将当前对象赋为上次的菜单标题
        lastMenuTitle = obj;
        //将当前显示的菜单赋为上次的菜单
        lastMenu = mu;
    }
</script>
</head>
<body>
<div id="menubar">
    <!-- 菜单条上的“文件”菜单 -->
    <div onMouseOver="menuHover(event);" onMouseOut="menuGotOut(event);"
    onClick="showMenu('file' , this);">文件</div>
    <!-- 菜单条上的“编辑”菜单 -->
    <div onMouseOver="menuHover(event);" onMouseOut="menuGotOut(event);"
    onClick="showMenu('edit' , this);">编辑</div>
    <!-- 菜单条上的“查看”菜单 -->
    <div onMouseOver="menuHover(event);" onMouseOut="menuGotOut(event);"
    onClick="showMenu('view' , this);" >查看</div>
</div>
<!--“文件”菜单 -->
<div id="file" style="display:none;left:9px;" class="menu" name="menu">
    <!-- 下面是“文件”菜单的三个菜单项 -->
    <div class="menuitem" onMouseOver="itemHover(event);"
    onMouseOut="itemGotOut(event);" onClick="">新建</div>
    <div class="menuitem" onMouseOver="itemHover(event);"
    onMouseOut="itemGotOut(event);" onClick="">打开</div>
    <div class="menuitem" onMouseOver="itemHover(event);"
        onMouseOut="itemGotOut(event);" onClick="">保存</div>
</div>
<!--“编辑”菜单 -->
<div id="edit" style="display:none;left:69px;" class="menu" name="menu">
    <!-- 下面是“编辑”菜单的三个菜单项 -->
    <div class="menuitem" onMouseOver="itemHover(event);"
    onMouseOut="itemGotOut(event);"          onClick="">复制</div>
    <div class="menuitem" onMouseOver="itemHover(event);"
    onMouseOut="itemGotOut(event);"          onClick="">剪切</div>
    <div class="menuitem" onMouseOver="itemHover(event);"
    onMouseOut="itemGotOut(event);"          onClick="">粘贴</div>
</div>
<!--“查看”菜单 -->
<div id="view" style="display:none;left:129px;" class="menu" name="menu">
    <!-- 下面是“查看”菜单的三个菜单项 -->
    <div class="menuitem" onMouseOver="itemHover(event);"
        onMouseOut="itemGotOut(event);"     onClick="">放大</div>
    <div class="menuitem" onMouseOver="itemHover(event);"
```

```
        onMouseOut="itemGotOut(event);"     onClick="">缩小</div>
    <div class="menuitem" onMouseOver="itemHover(event);"
        onMouseOut="itemGotOut(event);"     onClick="">全屏</div>
</div>
</body>
</html>
```

从上面的代码中可以看出，菜单条是一个<div.../>元素，而每个菜单也是一个<div.../>元素。菜单开始时全部隐藏，当单击菜单条上的菜单时，对应的菜单出现。代码中使用了一个变量保存上一次操作出现的菜单，每个菜单出现之前，先将上次出现的菜单隐藏。导航菜单的效果如图 6.27 所示。

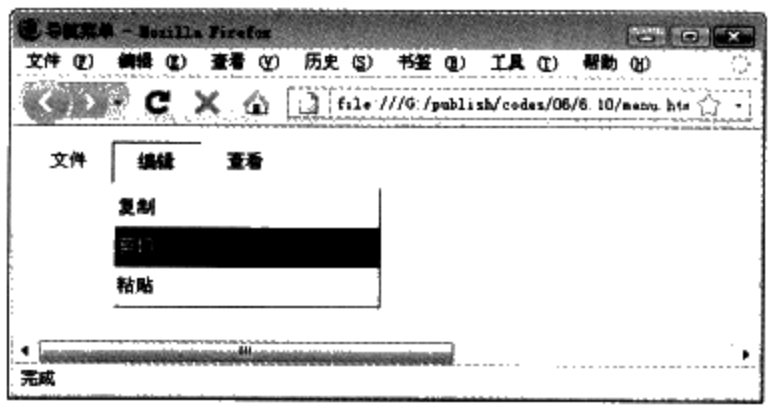


图 6.27 导航菜单的效果

从图 6.27 可看到，菜单是不透明的，如果放在页面中可能会将页面的其他内容挡住，为了避免这种情况，将该菜单的透明度设置为半透明即可。

6.11 DOM 模型和 XML 文档

XML 文档也是一种结构化文件，因而也可以通过 DOM 模型来操作。DOM 是以树形结构组织 XML 文档的每个节点，这个树形结构允许开发人员利用节点关系导航、访问指定节点。在开始解析 XML 文档之前，必须先装载整个文档，并构造对应的树形结构。

DOM 的解析处理具有如下优点：由于树在内存中是持久的，因此可以修改树的节点，对应为修改文档中元素的值，因此提供了灵活的修改。另外，需要对 XML 文档中的数据重复读取时，DOM 的优势非常明显，因为 DOM 一次性将整个 XML 文档全部读入内存，可以随机访问 XML 文档的每个元素。

但 DOM 解析器处理也存在一些问题：由于 DOM 需要在解析之前一次性加载整个 XML 文档，因此启动速度较慢，特别是文档较大时可能导致内存溢出。如果应用程序只需关注文档的一小部分，DOM 依然需要装载整个 XML 文档，不可避免地装载了那些永远不被使用的对象，这相当浪费。

➤➤6.11.1 使用 DOM 解析 XML 文档

在使用 DOM 解析 XML 文档时，使用的是 DOM 的思想，因而不可避免需要将 XML 文档转换成一棵 DOM 树。而 DOM 解析器在装入 XML 文档时，已经在内存中将 XML 文档的每个元素转换成了 DOM 树的每个节点。

对应下面这份简单的 XML 文档：

程序清单：codes\06\6.11\dom\student.xml

```
<?xml version="1.0" encoding="GBK"?>
<StudentInfo>
    <student>
        <name>张三</name>
        <sex>男</sex>
        <lesson>
            <lessonName>Spring 整合开发</lessonName>
            <lessonScore>85</lessonScore>
```

```
        </lesson>
        <lesson>
            <lessonName>轻量级 J2EE 应用开发</lessonName>
            <lessonScore>95</lessonScore>
        </lesson>
        <lesson>
            <lessonName>Ajax 应用开发</lessonName>
            <lessonScore>80</lessonScore>
        </lesson>
    </student>
    <student>
        <name>王小梅</name>
        <sex>女</sex>
        <lesson>
            <lessonName>Word 快速入门</lessonName>
            <lessonScore>80</lessonScore>
        </lesson>
        <lesson>
            <lessonName>Excel 应用</lessonName>
            <lessonScore>95</lessonScore>
        </lesson>
        <lesson>
            <lessonName>使用 OutLook</lessonName>
            <lessonScore>80</lessonScore>
        </lesson>
    </student>
</StudentInfo>
```

下面是解析该文档的 Java 程序代码：

程序清单：codes\06\6.11\dom\src\lee\DOMParserTest.java

```
public class DOMParserTest
{
    public static void main(String[] args)
    {
        DOMParserTest tp = new DOMParserTest();
        tp.parseXMLFile("student.xml");
    }
    /**
    * 解析文档
    * @param fileName
    */
    public void parseXMLFile(String fileName)
    {
        try
        {
            //构造 DOM 解析器的实例
            DOMParser parser = new DOMParser();
            //开始解析文档，将 XML 文件转换成 DOM 树存入内存
            parser.parse(fileName);
            //getDocument()获取 Document 对象
            Document doc = parser.getDocument();
            //获取 root 节点
            Element elmtInfo = doc.getDocumentElement();
            //getElementsByTagName()根据标签名获取子节点列表
            NodeList nlStudent = elmtInfo
                .getElementsByTagName("student");
            System.out.println("XML 文件开始解析");
            //循环输出每一个学生成绩
            for (int i = 0; i < nlStudent.getLength(); i++)
            {
```

```
            //当前 student 元素
            Element elmtStudent = (Element) nlStudent.item(i);
            //利用父子关系获取子节点
            NodeList nlCurrent = elmtStudent
                .getElementsByTagName("name");
            //读取到姓名节点的值
            System.out.println("姓名:" + nlCurrent.item(0)
                .getFirstChild().getNodeValue());
            //利用父子关系获取子节点
            nlCurrent = elmtStudent.getElementsByTagName("sex");
            //读取到性别节点的值
            System.out.println("性别:" + nlCurrent.item(0)
                .getFirstChild().getNodeValue());
            //再次获取多个课程节点
            nlCurrent = elmtStudent.getElementsByTagName("lesson");
            //遍历每个课程节点
            for (int j = 0; j < nlCurrent.getLength(); j++)
            {
                //获取第 i 个课程节点
                Element elmtLesson = (Element) nlCurrent.item(j);
                //获取 lessonName 节点的值
                NodeList nlLesson = elmtLesson
                    .getElementsByTagName("lessonName");
                System.out.print(nlLesson.item(0)
                    .getFirstChild().getNodeValue());
                System.out.print(":");
                //获取 lessonScore 节点的值
                nlLesson = elmtLesson
                    .getElementsByTagName("lessonScore");
                System.out.print(nlLesson.item(0)
                    .getFirstChild().getNodeValue());
                System.out.println();
            }
            System.out.println("--------------------");
        }
        System.out.println("XML 文件解析结束");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
  }
}
```

通过上面的代码可看出，DOM 解析器处理 XML 文档的方式类似于树的遍历。我们在代码中以遍历树的方式遍历每个节点时，DOM 解析器负责对应处理文档中的每个元素。

➤➤6.11.2 使用 DOM 解析器创建 XML

DOM 模型不仅可以用于解析 XML 文档，还可用于创建 XML 文档。使用 DOM 解析器创建 XML 文档的思路是先创建一个 Document 对象，Document 对象是顶层对象，可以获得一个根节点，根节点又可以增加任意多个子节点，而每个子节点又可以添加任意多个子节点。依此类推，最后形成一棵完整的 DOM 树，然后使用输出流将这颗 DOM 树输出即可。

下面是使用 DOM 生成 XML 文档的范例程序：

程序清单：codes\06\6.11\dom\src\lee\DOMGenerate.java

```
public class DOMGenerate
{
    //主方法，程序的入口
    public static void main( String[] args )
    {
        try
        {
            //创建 DocumentFactory 对象
            DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            //创建 Document 对象
            Document doc = db.newDocument();
            //创建根元素
            Element root = doc.createElement("Student");
            //创建 name 元素
            Element item = doc.createElement("name");
            //为 name 元素增加文本子节点
            item.appendChild(doc.createTextNode("张三"));
            //将 name 元素添加到根元素下
            root.appendChild(item);
            //创建 age 元素
            item = doc.createElement("age");
            //为 age 元素增加文本子元素
            item.appendChild(doc.createTextNode("28" ));
            //将 age 元素添加到根元素下
            root.appendChild(item);
            //创建 high 元素
            item = doc.createElement("high");
            //为 high 元素添加文本子元素
            item.appendChild(doc.createTextNode("1.72" ));
            //将 high 元素添加到根元素下
            root.appendChild(item);
            //创建 score 元素
            item = doc.createElement("score");
            //创建 Java 元素
            Element lesson = doc.createElement("Java");
            //为 Java 元素添加文本子元素
            lesson.appendChild(doc.createTextNode("95"));
            //将 Java 元素添加到 score 元素
            item.appendChild( lesson );
            //创建 Struts 元素
            lesson = doc.createElement("Struts");
            //为 Struts 元素添加文本子元素
            lesson.appendChild(doc.createTextNode("90"));
            //将 Struts 元素添加到 score 元素
            item.appendChild( lesson );
            //创建 Hibernate 元素
            lesson = doc.createElement("Hibernate");
            //为 Hibernate 元素添加文本子元素
            lesson.appendChild(doc.createTextNode("90"));
            //将 Hibernate 元素添加到 score 元素
            item.appendChild( lesson );
            //将 score 元素添加到根元素下
            root.appendChild( item );
            //为文档指定根元素
            doc.appendChild( root );
```

```
            //指定输出格式
            OutputFormat format = new OutputFormat(doc
                , "GBK" , true);
            StringWriter stringOut = new StringWriter();
            XMLSerializer serial = new XMLSerializer( stringOut, format );
            //将 DOM 树转换成字符串
            serial.asDOMSerializer();
            serial.serialize(doc.getDocumentElement());
            //创建文件输出流
            PrintStream ps = new PrintStream(new FileOutputStream("new.xml"));
            //输出 XML 文件
            ps.println(stringOut.toString());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面的程序中粗体字代码以 DOM 方式在内存中创建了一棵 DOM 树，程序先创建一个 Document，然后以 Document 来创建树中的其他节点，并将这些节点以树的形式组织起来。程序最后直接将该 DOM 输出到文件，即可得到如下 XML 文档：

```
<?xml version="1.0" encoding="GBK"?>
<Student>
    <姓名>张三</姓名>
    <age>28</age>
    <high>1.72</high>
    <score>
        <Java>95</Java>
        <Struts>90</Struts>
        <Hibernate>90</Hibernate>
    </score>
</Student>
```

6.12 本章小结

本章详细介绍了 DOM 模型的相关知识，包括利用 DOM 处理 XHTML 文档和 XML 文档。学习本章的基础是掌握 DOM 模型的思想，掌握结构化文件和 DOM 树之间的转化关系。本章的重点是通过 DOM 动态更新 XHTML 文档，包括访问 XHTML 元素的 5 种情况，修改 XHTML 元素，增加 XHTML 元素的 3 种情况和删除 XHTML 元素的 3 种情况。本章也介绍了传统 DHTML 对象模型，并介绍了传统 DHTML 模型里对象的包含关系。

除此之外，本章还介绍了 window 和 document 两个重要的对象。读者应该掌握这两个对象的常用方法和功能。

本章还介绍了两个相当常用的页面效果：仿 Excel 表格和仿 Windows 菜单，这两个页面效果既有很强的实用意义，也有很好的参考意义，希望读者认真体会。

➤➤本章练习

结合 CSS、XHTML、JavaScript 开发 Tab 页效果。

开发基于 HTML+JavaScript 技术的五子棋（需要结合第 7 章的事件编程知识）。

开发基于 HTML+JavaScript 技术的俄罗斯方块游戏（需要结合第 7 章的事件编程知识）。

CHAPTER 7

第7章 事件处理机制

本章要点

- 两种绑定事件处理函数的基本方法
- 事件处理函数中的 this 关键字
- 使用返回值改变事件的默认行为
- 在代码中触发事件
- Ajax 应用的 MVC 模式
- Internet Explorer 中绑定事件处理器的两种方法
- Internet Explorer 的事件对象
- Internet Explorer 的事件传播机制
- Internet Explorer 的事件重定向
- Internet Explorer 中取消事件的默认行为
- Internet Explorer 中捕获鼠标事件
- DOM 2 中绑定事件处理器的方法
- DOM 2 的事件对象
- DOM 2 的事件传播机制
- DOM 2 的事件转发
- DOM 2 中取消事件的默认行为

前一章已经介绍了 DOM 编程，通过使用 DOM 模型，开发者可以动态地改变 XHTML 页面内容。在前一章的许多案例中，为了让页面内容随用户单击鼠标而改变，我们在页面中为 XHTML 元素指定了 onclick 属性——实际上这已经用到了事件处理机制。

就像 AWT、Swing 界面编程的事件处理一样，若需要程序运行时能与用户交互，这时候就需要事件机制了。JavaScript 也提供了完备的事件机制，允许 XHTML 页面实现良好的用户交互。

事件机制使得客户端的 JavaScript 有机会被激活，从而得到调用。在一个 Web 页面装载之后，运行脚本的唯一机会，就是响应系统或者用户的动作。从第一个支持脚本编程的浏览器面世以来，简单的事件机制一直都是 JavaScript 脚本的一部分。现在，绝大部分浏览器都实现了强壮的事件模型，使脚本可以更加智能地处理事件。事件机制是 JavaScript 的重点部分，Ajax 应用对服务器状态的监控，动态加载服务器的数据等工作，都离不开 JavaScript 的事件机制。

7.1 基本事件模型

JavaScript 在浏览器中运行，浏览器中的 XHTML 页面主要由 DOM 组成，这些 DOM 对象也对应到各种 XHTML 元素，我们就可为这些 DOM 对象或 XHTML 元素绑定事件处理函数（或多条 JavaScript 脚本），当这些 DOM 对象或 XHTML 元素上发生某个动作时，这些事件处理函数（或多条 JavaScript 脚本）就会被激发，从而获得执行的机会。

为 XHTML 元素绑定处理函数有很多种不同的方法。事实上，不同的浏览器又提供了自己的绑定方法，这些绑定事件处理函数的方法很难说哪种更好，哪种更差。不同的绑定方法有不同的侧重点。为了保证更好的跨浏览器特性，通常推荐采用与浏览器无关的事件绑定方法。本节介绍的两种方式都可实现跨浏览器绑定。

➤➤7.1.1 绑定 XHTML 元素属性

常用的绑定事件处理器的方法是直接绑定到 XHTML 元素的属性，正如在前一章示例程序中所看到的：我们为多个 XHTML 元素指定了 onclick 属性值。

绑定到 XHTML 元素属性时，该属性值是一条或多条 JavaScript 脚本，多条脚本之间以英文分号分隔。

大部分表单的数据校验都会采用这种方式。在提交表单时，JavaScript 的事件处理程序将会对表单域进行校验，如果不能通过数据校验，则弹出警告对话框。

事件属性名称由事件类型前加一个“on”前缀构成，例如 onclick、ondblclick 等。这些属性的值也被称为事件处理器，因为它们指定了如何“处理”特定的事件类型。事件处理器属性的值是多条 JavaScript 脚本，最常见的值是一条调用某个 JavaScript 函数的语句。

下面介绍一个表单校验的示例程序，下面的 XHTML 页面包含三个表单域控件：用户名、密码、电邮。其中用户名、密码不能为空，而电子邮件不能为空，且必须满足电子邮件格式。在浏览器中浏览该页面，可看到如图 7.1 所示的效果。

图 7.1 数据校验的输入页面

当用户提交如图 7.1 所示表单时，JavaScript 的处理函数将自动触发，对上面的三个输入域进行输

入校验。为了绑定校验函数，只需要为该<form.../>的 onsubmit 属性指定合适的属性值即可，绑定数据校验函数的代码如下：

程序清单：codes\07\7.1\elementBind.html

```
<div align="center">
<h2>数据校验表单</h2>
<form method="post" onsubmit="return check(this);"
    name="register" action="#">
    用户名：<input type="text" name="user" /><br />
    密  码：<input type="password" name="pass" /><br />
    电  邮：<input type="text" name="email" /><br />
    <input type="submit" value="提交" />
</form>
</div>
```

上面的<form.../>元素的 onsubmit 属性为"return check()"，之所以使用 return 语句，是为了保证数据校验不能通过时拒绝表单提交。而 check()函数就是负责校验表单的函数，当该表单被提交时，该 check()函数就会被激发，从而获得执行的机会。

check()函数则使用 DOM 模型访问页面中的表单，并对表单里的表单控件进行校验，校验表单控件里的值是否符合业务要求。该页面的 JavaScript 片段如下：

程序清单：codes\07\7.1\elementBind.html

```
<script type="text/javascript">
//使用正则表达式截取空格
function trim(s)
{
    return s.replace( /^\s*/, "" ).replace( /\s*$/, "" );
}
//负责处理表单 submit 事件的函数
function check()
{
    //访问页面中第一个表单
    var form = document.forms[0];
    //错误字符串
    var errStr = "";
    //当用户名为空
    if (trim(form.user.value) == null || trim(form.user.value) == "")
    {
        errStr += "\n 用户名不能为空!";
        form.user.focus();
    }
    //当密码为空
    if (trim(form.pass.value) == null || trim(form.pass.value) == "")
    {
        errStr += "\n 密码不能为空!";
        form.pass.focus();
    }
    //当电子邮件为空
    if (trim(form.email.value) == null || trim(form.email.value) == "")
    {
        errStr += "\n 电子邮件不能为空!";
        form.email.focus();
    }
    //使用正则表达式校验电子邮件的格式是否正确
    if(!/^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/.test(trim(form.email.value)))
    {
        errStr += "\n 电子邮件的格式不正确!";
        form.email.focus();
```

```
        }
        //如果错误字符串不为空，表明校验出错
        if( errStr != "" )
        {
            //弹出出错信息
            alert(errStr);
            //返回 false，用于阻止表单提交
            return false;
        }
    }
    </script>
```

如果不输入用户名，不输入密码，并且输入的电子邮件格式不正确，则单击“提交”按钮将弹出如图 7.2 所示的对话框。

图 7.2　绑定 XHTML 元素属性

这种事件绑定方式简单易用，但绑定事件处理器时需要直接修改 XHTML 页面代码，因此存在如下几个坏处：

- 直接修改了 XHTML 元素属性，增加了页面逻辑的复杂度。
- 开发人员需要直接修改 XHTML 页面，不利于团队协作开发。

但这种绑定方式也有一个优点：在这种绑定方式中，XHTML 元素的 onclick 等属性值是一条或多条 JavaScript 语句，因此可以在调用 JavaScript 函数时传入参数，典型的就是传入 this、event 等有特殊意义的参数。关于事件处理函数中的关键字 this 的意义，请参看本书 7.1.3 节。

注意：

将事件处理函数绑定到 XHTML 元素的属性时，可以为函数传入参数，尤其是可传入 this、event 等具有特殊意义的参数。

➤➤7.1.2　绑定 DOM 对象的属性

直接绑定到 DOM 对象属性时，开发者无须修改 XHTML 元素的代码，而是将事件处理函数放在 JavaScript 脚本中绑定。

为了给特定的 XHTML 元素绑定事件处理函数，必须先在代码中获得需要绑定事件处理函数的 XHTML 元素对应的 DOM 对象，该 DOM 对象就是触发事件的事件源，然后给该 DOM 对象的 onclick 等属性赋值，其合法的属性值是一个 JavaScript 函数的引用。

值得指出的是：因为绑定到 DOM 对象属性时，该属性值只是一个 JavaScript 函数的引用，因此千万不要在函数后添加括号——一旦添加括号，那就变成了调用该函数，于是只是将该函数返回值赋给 DOM 对象的 onclick 等属性。

提 示:

与 AWT、Swing 事件编程机制相同的是，JavaScript 事件机制中也有事件源、事件监听器等概念。JavaScript 事件机制中的事件源就是各种 DOM 元素。由于 Java 是面向对象的编程语言，因此必须定义监听器来监听事件源；但 JavaScript 中函数可以独立存在，JavaScript 中事件监听器直接指定函数即可。

同样是如图 7.1 所示的页面，此时不再为<form.../>元素指定 onsubmit 属性，下面是表单的代码片段：

程序清单：codes\07\7.1\domBind.html

```
<form method="post" name="register" action="#">
    ...
</form>
```

上面的表单元素没有直接绑定事件处理函数，因此，我们需要通过绑定 DOM 对象属性来设置事件处理函数。只要在 JavaScript 脚本最后添加如下一行：

```
//为第一个表单的 onsubmit 绑定事件处理器
document.forms[0].onsubmit = check;
```

上面的粗体字代码中的 onsubmit 属性值只是 check 函数引用，没有在 check 函数后添加括号。在这种方式下，JavaScript 的事件绑定机制与 Java 事件绑定机制非常相似：事件源通常是一个可视化控件，事件监听器就是一个 JavaScript 函数（因为函数是 JavaScript 的一等公民）。

当采用这种方式为 DOM 对象绑定事件处理器函数时，开发者无须修改 XHTML 文档，只需在该页面中增加一行代码用于绑定即可。正如 4.1.2 节所介绍的，大部分时候我们都将 JavaScript 脚本写在单独的*.js 文件中，这样我们完全无须修改 XHTML 页面，只需修改*.js 文件即可，这样更有利于团队协作。

正如 Java 事件机制中不同可视化控件支持不同的事件类型，不同的 XHTML 控件也支持激发不同的事件。表 7.1 显示了所有标准 XHTML 文档控件支持的事件及对应描述。

表 7.1　标准 XHTML 文档元素所支持的事件

事件属性	对应的含义	支持该属性的 XHTML 标签
onabort	图片加载被中断	<img>
onblur	当某个 HTML 元素失去焦点时触发该事件，通常意味着用户已经激活了另一个 HTML 元素，通常对应用户单击了另一个 HTML 元素，或使用 Tab 切换了焦点	<button>、<input>、<label>、<select>、<textarea>、<body>
onchange	当表单域的值被修改时触发	<input>、<select>、<textarea>
onclick	单击某个标签时触发	大多数的可显示标签 IE 4.0 以后还支持<applet>和<font>
ondblclick	双击某个元素时触发	大多数的可显示标签 IE 4.0 以后还支持<applet>和<font>
onerror	图片加载出错时触发	<img>
onfocus	当某个标签得到焦点时触发，通常是用户单击，或者使用 Tab 键切换了焦点	<button>、<input>、<label>、<select>、 <textarea>、<body>
onkeydown	当焦点在当前元素上，按下键盘的某个键时触发	表单域控件标签和<body>标签
onkeypress	当焦点在当前元素上，单击键盘的某个键时触发	表单域控件标签和<body>标签
onkeyup	当焦点在当前元素上，并且松开了键盘的某个键时触发	表单域控件标签和<body>标签
onload	当某个对象被装载完毕时触发	<body>、<frameset>、<img>
onmousedown	当焦点停留在当前元素，并且按下鼠标键时触发	大多数可显示的标签 IE 4.0 以后的版本还支持<applet>和<font>
onmousemove	鼠标在当前元素上面，并且鼠标在当前元素上面移动时触发	大多数可显示的标签 IE 4.0 以后的版本还支持<applet>和<font>
onmouseout	当鼠标移出某个元素时触发，即鼠标一开始停留在元素上面	大多数可显示的标签 IE 4.0 以后的版本还支持<applet>和<font>
onmouseover	当鼠标移动到该元素上面时触发	大多数可显示的标签 IE 4.0 以后的版本还支持<applet>和<font>
onmouseup	当焦点在当前元素，并松开鼠标键时触发事件	大多数可显示的标签 IE 4.0 以后的版本还支持<applet>和<font>
onreset	当用户重置表单时触发	<form>
onresize	当窗口大小被改变时触发	<body>、<frameset>
onselect	当用户选择文本框或文本域的某段文字时触发	<input>、<textarea>
onsubmit	当表单提交时触发，通常通过单击表单的提交按钮触发	<form>
onunload	当某个对象从窗口或框架中卸载完毕时触发	<body>、<frameset>、<img>

➤➤7.1.3　事件处理函数和关键字 this

JavaScript 脚本通常处于 window 对象下运行，如果 JavaScript 脚本中使用 this 关键字，则通常引用到 window 本身。看如下的代码片段：

程序清单：codes\07\7.1\this.html

```
<input type="button" value="按钮" name="bn" onclick="showThisName();"/>
<script type="text/javascript">
    //为当前的浏览器窗口的 name 属性赋值
    window.name = "测试窗口";
```

```
    function showThisName()
    {
        //此时的 this 将引用到脚本所在的窗口
        alert(this.name);
    }
</script>
```

上面的代码中的 alert(this.name)中的 this 将引用窗口本身，当单击“按钮”按钮时，将弹出如图 7.3 所示的对话框。

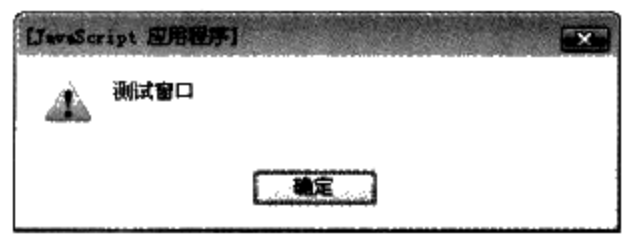


图 7.3　this 通常引用窗口本身

当我们为 XHTML 元素的 onclick 等属性指定一系列 JavaScript 脚本时，如果在这些 JavaScript 脚本中使用关键字 this，则该关键字引用该 XHTML 元素本身。

当我们为 DOM 对象的 onclick 等属性指定一个 JavaScript 函数引用时——由于 JavaScript 是一门动态语言，因此我们可以随时为某个对象添加属性和方法——这种方式可认为就是为该 DOM 对象增加了一个方法，因此，当我们在该函数中使用关键字 this 时，该 this 也是引用该 DOM 对象本身。

在一种极端的情况下，我们为 DOM 对象的 onclick 等属性指定的不是一个独立的 JavaScript 函数，而是某个对象的方法。例如如下代码：

```
//将对象 p 的 info 方法设置为按钮 bn3 的事件处理器
document.getElementById("bn3").onclick = p.info;
```

在上面的代码中，程序将按钮 bn3 的 onclick 属性赋值为对象 p 的 info 属性。在这种情况下，bn3 的 onclick 属性、对象 p 的 info 实际上引用了同一个函数，因此我们可以认为这行代码为按钮 bn3 增加了一个 onclick 方法。

注意：

JavaScript 的函数是一等公民，它永远独立而不从属于任何对象。即使我们将某个匿名函数定义为一个对象的方法，它依然是独立的。

因此当用户单击按钮 bn3 时，按钮 bn3 的 onclick 方法被触发——与对象 p 没有任何关系，所以 info()函数中 this 引用到按钮 bn3。

与此对应的是，如果我们直接调用对象 p 的 info()方法，则 info()函数中的 this 引用到对象 p。

看如下 XHTML 页面代码：

程序清单：codes\07\7.1\thisTest.html

```
<body>
    <input id="bn1" type="button" value="按钮 1" onclick="alert(this.value);"/>
    <input id="bn2" type="button" value="按钮 2" />
    <input id="bn3" type="button" value="按钮 3" />
    <script type="text/javascript">
        function test()
        {
            alert(this.value);
        }
        //将 test 函数设置为按钮 bn3 的事件处理器
        document.getElementById("bn2").onclick = test;
        //使用 JSON 格式定义了一个对象
        var p =
        {
            value: 'p 对象',
            info: function()
            {
                alert(this.value);
```

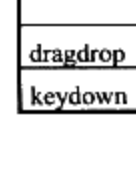
```
            }
        }
        //将对象 p 的 info 方法设置为按钮 bn3 的事件处理器
        document.getElementById("bn3").onclick = p.info;
        //直接调用对象 p 的 info()方法
        p.info();
    </script>
</body>
```

在上面的页面代码中，当我们单击“按钮一”、“按钮二”和“按钮三”中的任意一个时，三个按钮的事件处理函数中的 this 分别引用不同的按钮。只有最后一行代码直接调用对象 p 的 info()方法时，info()方法中的 this 才引用对象 p。

当用户单击按钮时，所触发的事件处理函数中 this 默认是引用到绑定该事件处理函数，如果我们确实需要让该 this 引用到原有的 JavaScript 对象，我们可以“曲线实现”，例如将代码改为如下形式：

```
document.getElementById("bn3").onclick = new function(){p.info();}
```

在上面的代码中将 onclick 属性赋值为一个新的匿名函数，该匿名函数里直接调用了对象 p 的 info()方法，这样 info()函数里的 this 总是引用对象 p。



提 示：

通过使用这种方式也可在绑定 DOM 对象属性时为事件处理函数传入参数，因为这种方式实质是重新构建一个匿名函数，该匿名函数里可以直接调用事件处理函数。

➤➤7.1.4 使用返回值改变默认行为

XHTML 元素大都包含了自己的默认行为。例如：单击超级链接将导致页面导航到超级链接所指的页面，单击表单的提交按钮将导致表单提交……但如果为这些元素增加对应的事件处理函数，将可以改变这种默认行为。例如阻止超级链接导航，看下面的代码：

```
<!-- 增加了事件处理函数的超级链接 -->
<a href="http://www.leegang.org" onclick="return false;">疯狂 Java 联盟</a>
```

上面的“疯狂 Java 联盟”超级链接绑定了 onclick 事件处理器。如果没有指定该事件处理函数，单击该链接时页面将导航到“疯狂 Java 联盟”站点。但该超级链接绑定了 onclick 事件处理代码，该代码简单地返回 false，这将阻止超级链接的导航功能。还有在 7.1.1 节所见的，如果表单提交事件的事件处理函数返回 false，也将阻止表单提交。

除了可以直接使用 return 返回值，还可以使用 confirm 提示框，单击该提示框的“确定”按钮将返回 true，而单击“取消”按钮则返回 false。看下面的代码：

```
<a href="http://www.leegang.org"
    onclick="return confirm('是否转入疯狂 Java 联盟？');">疯狂 Java 联盟</a>
```

当单击“疯狂 Java 联盟”超级链接时，将弹出如图 7.4 所示的对话框。

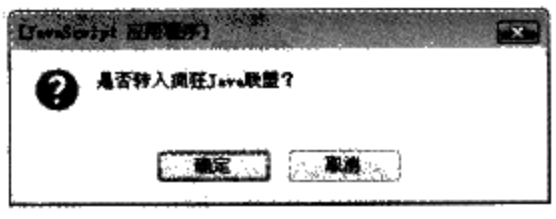


图 7.4 使用确认对话框确定是否导航

如果单击“确定”按钮，页面将导航到“疯狂 Java 联盟”站点；如果单击“取消”按钮，页面将不会导航。表 7.2 显示了 XHTML 元素的事件处理函数返回 false 时所产生的行为：

表 7.2 当事件处理函数返回 false 时，XHTML 元素的行为

事件处理属性	当事件处理函数返回 false 时产生的行为
click	对于单选框、复选框，将阻止选择该项；对于表单的提交按钮，阻止表单提交；对于表单的重置按钮，将阻止表单重置；对于超级链接，则阻止页面导航。
dragdrop	取消拖放事件
keydown	取消“按下键”事件

续 表

事件处理属性	当事件处理函数返回 false 时产生的行为
keypress	取消“单击键”事件
mousedown	取消鼠标按下的默认动作
mouseover	取消鼠标悬停的默认动作
submit	阻止表单的提交

➤➤7.1.5 在代码中触发事件

除了可以让用户动作、窗口动作等触发 JavaScript 中的事件外，JavaScript 还允许在脚本中触发事件，程序触发事件与用户动作触发事件的效果完全相同。所谓在脚本中触发事件，其关键就在于先获得该 DOM 对象或 XHTML 对象，然后在 JavaScript 脚本中调用该对象的方法即可。

对于 7.1.1 节所示应用，将按钮的 type 属性设置为 button，表明这个按钮没有提交表单的功能。下面是该表单的代码：

程序清单：codes\07\7.1\scriptTrigger.html

```
<form method="post" name="register" action="http://www.leegang.org">
    用户名：<input type="text" name="user" /><br />
    密  码：<input type="password" name="pass" /><br />
    电  邮：<input type="text" name="email" /><br />
    <input type="button" id="regist" value="提交"/>
</form>
```

注意：

上面的<form.../>元素内包含一个“提交”按钮，该按钮有个 id 属性，该属性不能为 submit 等值，否则下面的程序将出现错误。关于这个陷阱，笔者当年折腾了很久才发现这个陷阱。但如果该按钮不作为<form.../>元素的子元素则没有任何问题。

上面的按钮的类型是 button，表明单击该按钮时不会引发表单的提交。我们在脚本中为该按钮绑定事件处理函数，绑定事件处理函数的代码如下：

程序清单：codes\07\7.1\scriptTrigger.html

```
<script type="text/javascript">
//使用正则表达式截取空格
function trim(s)
{
    return s.replace( /^\s*/, "" ).replace( /\s*$/, "" );
}
//负责处理表单 submit 事件的函数
function check()
{
    //访问页面中第一个表单
    var form = document.forms[0];
    //错误字符串
    var errStr = "";
    //当用户名为空
    if (trim(form.user.value) == null || trim(form.user.value) == "")
    {
        errStr += "\n 用户名不能为空!";
        form.user.focus();
    }
    //当密码为空
    if (trim(form.pass.value) == null || trim(form.pass.value) == "")
    {
        errStr += "\n 密码不能为空!";
        form.pass.focus();
```

```
    }
    //当电子邮件为空
    if (trim(form.email.value) == null || trim(form.email.value) == "")
    {
        errStr += "\n 电子邮件不能为空!";
        form.email.focus();
    }
    //使用正则表达式校验电子邮件的格式是否正确
    if(!/^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/.test(trim(form.email.value)))
    {
        errStr += "\n 电子邮件的格式不正确!";
        form.email.focus();
    }
    //如果错误字符串不为空，表明数据校验出错
    if( errStr != "" )
    {
        //弹出出错提示
        alert(errStr);
    }
    //如果错误字符串为空，表明数据校验通过，可以提交表单
    else
    {
        //在代码中手动提交表单
        form.submit();
    }
}
//为第一个表单的 onclick 绑定事件处理器
document.getElementById("regist").onclick = check;
```

上面的程序中粗体字代码调用 form 的 submit()方法手动提交了表单——这就是在脚本中触发事件。表 7.3 显示了常见 XHTML 元素触发事件的方法：

表 7.3　常见 HTML 元素触发事件

触发事件的方法	所支持的 HTML 元素
click()	<input type="button">、<input type="checkbox">、<input type="reset">、<input type="submit">、<input type="radio">、<a>
blur()	<select>、<input>、<textarea>、<a>
focus()	<select>、<input>、<textarea>、<a>
select()	<input type="text">、<input type="password">、<input type="file">、<textarea>
submit()	<form>
reset()	<form>

答：其实这个问题比较容易混淆。其关键就在于 JavaScript 是一种动态语言，它允许动态地为对象增加属性和方法，前面第六章介绍过，访问表单域控件有一种简单的方法：formObj.elementName，其中 elementName 就是表单域的 id 或 name 属性值——这样可视为表单对象有一个 elementName 属性，也就是说：当表单 a 内包含 id 或 name 分别为 x、y 的两个表单域时，相当为该表单对象增加了 x、y 两个属性！

理解了上面的理论，我们就会明白：当我们指定<form.../>元素中<input.../>元素的 id 属性值为 submit 时，则<form.../>元素对应的 DOM 对象就增加了 submit 属性——这就覆盖了该对象中原有的 submit()方法，从而导致无法提交表单。

实际上，定义表单域控件的 name、id 属性时，这些属性值不应该和表单对象原有的方法名、属性名相同，否则这些表单域控件就会覆盖原有的方法、属性。

7.2 Ajax 应用的 MVC

起初，MVC 模式是针对相同的数据需要不同显示的应用而设计，其整体的结果如图 7.5 所示。

在经典的 MVC 模式中，事件由控制器处理，控制器根据事件的类型改变模型或视图，反之亦然。具体地说，模型维护一个视图列表，这些视图为获得模型变化通知，通常采用观察者模式登记给模型。模型发生改变时，向所有登记过的视图发送通知。接下来，视图从对应的模型中获得信息，然后更新自己。

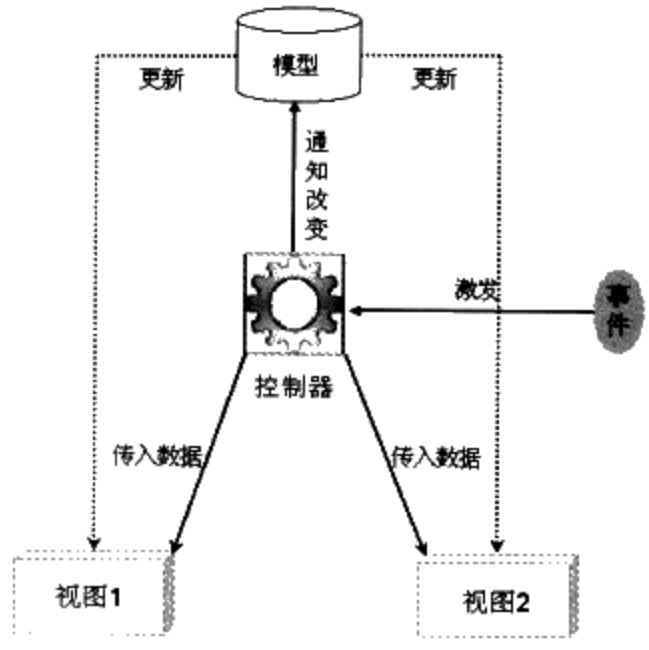


图 7.5 MVC 结构图

MVC 模式提供了分离的视图逻辑和业务逻辑，让应用具备良好的可扩展性。MVC 模式将应用的不同部分分成三组组件，每组组件具有相同的特征，有利于通过工程化、工具化产生管理程序代码。

在 MVC 架构的应用里，模型和视图不允许直接通信，模型更加专注于业务逻辑的实现，不需要与具体的视图技术耦合，从而可提供更好的可重用性；视图则无须调用业务逻辑组件的方法，仅从控制器读取相应的数据，从而避免了视图和业务逻辑直接耦合。视图仅仅显示从控制器传过来的简单数据，而不是直接调用业务方法。

因此，虽然在图 7.5 中显示了模型的改变将引起视图的更新，但这种更新依然需要通过控制器。若视图更新需要读取新数据，也是从控制器读取，而不是直接访问模型数据。

Java EE 应用的 MVC 架构不是本书的重点，本节并不打算在此处重点介绍（如需了解 Java EE 架构的更多知识，请参阅疯狂 Java 体系的《轻量级 Java EE 企业开发实战》一书）。本节将简单介绍 Ajax 应用对于传统 Java EE 应用架构的改变，以及 Ajax 应用的服务器端 MVC 到底需要多大的改变。图 7.6 显示了传统 Java EE 应用的架构。

而增加 Ajax 技术后，Ajax 技术带来的改变在哪里呢？前面已经介绍过了，Ajax 技术并不是要全面推翻 Java EE 应用的架构，而是对传统 Web 应用的完善。根据第二章 Ajax 初体验中介绍，Ajax 技术主要是在客户端浏览器和服务器端增加了薄薄一层，这一层就是 Ajax 引擎，Ajax 引擎负责异步发送请求，并负责解析服务器响应数据。

因为 Ajax 技术提供了动态更新页面显示的能力，因此 Ajax 应用通常不再使用 JSP 页面作为视图层，可以直接使用 XHTML 页面作为视图层。那么，传统的 JSP 页面是否还有用呢？正如第二章介绍的，当服务器需要生成大量的数据响应，并且响应的数据包含了丰富的显示格式时，应该考虑使用 JSP 生成响应，而不是使用控制器直接生成响应。因此，增加 Ajax 技术后的 Java EE 应用架构如图 7.7 所示。

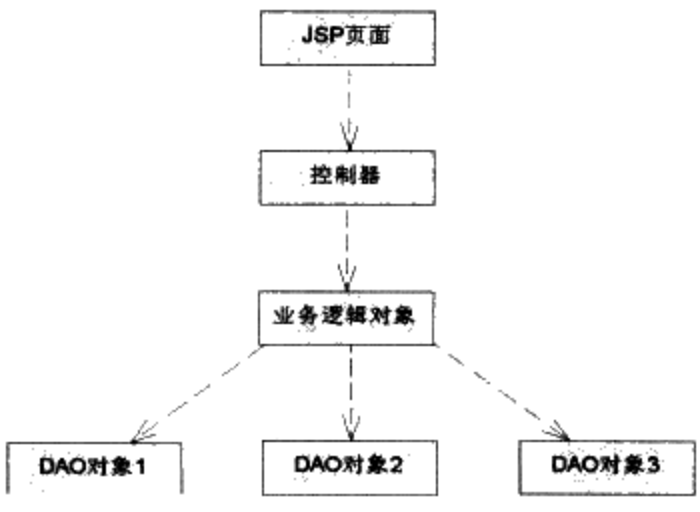


图 7.6 传统 Java EE 应用的架构

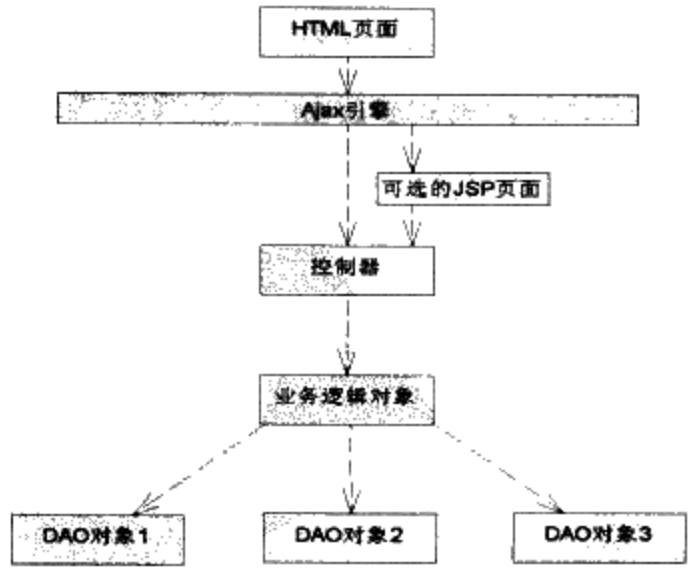


图 7.7 增加了 Ajax 技术的 Java EE 应用

可选的 JSP 页面到底是否需要，不可一概而论。通常的建议是，如果使用 XML 文档响应，因为响应中包含了大量的标签，建议采用 JSP 页面。而如果使用普通文本响应，因为响应简单，则通常建议直接使用控制器输出响应，不必拘泥于彻底的分层主义。

本节还将关注客户端 JavaScript 脚本的 MVC，通过对 JavaScript 应用 MVC 架构模式，让代码具有更好的可复用性和可扩展性。

回忆经典的 MVC 的由来，视图为用户提供可视的界面，以便用户触发事件，以事件驱动与控制器通信。视图在自身发生某些事件时，通知控制器，然后由控制器通知模型处理事件，该事件可能引发模型数据的改变，一旦模型数据改变，则控制器也会通知所有的视图做出对应的修改。

在纯粹的 Ajax 应用中，MVC 模型的三个部分可以有如下划分：

- 模型：由代表服务器响应的对象充当，模型负责从服务器读取数据，并负责通知控制器将数据更新（Ajax 应用中模型通常认为是 XMLHttpRequest 的 responseText 或 responseXML）。
- 视图：通常由 XHTML 页面的 DOM 元素充当，这些元素负责显示模型中的数据，并且让用户触发事件以驱动事件。
- 控制器：由 JavaScript 中的事件处理函数充当。事件处理函数控制负责响应视图的事件，并将模型的改变动态地加载到 XHTML 元素中。

在上面的三个角色的划分中，视图和控制器非常容易得到。但模型很容易和控制器混在一起，特别是当服务器的响应非常简单时，例如只是简单的文本字符串，如果要强行划分出模型将陷入彻底的分层主义。为了让 Ajax 应用保持较好的 MVC 架构，通常有如下三个建议：

- 将 JavaScript 脚本从 XHTML 文档里分离出来，用单独的*.js 文件保存。
- 不要将事件处理函数绑定到 XHTML 元素属性，而应该绑定到 DOM 对象的属性。
- 将动态更新 XHTML 页面的脚本分离处理，而不是直接混杂在事件处理函数里。

下面有一个简单的 XHTML 页面，该页面内只有一个按钮，单击该按钮会发送异步请求：

程序清单：codes\07\7.2\ajaxmvc\mvc.html

```
<body>
<!-- 定义一个简单的按钮 -->
<input type="button" id="test" value="测试" />
<div id="show">
</div>
<!-- 引入 JavaScript 函数 -->
<script src="mvc.js" type="text/javascript">
</script>
</body>
```

上面的页面代码中粗体字代码导入了 JavaScript 文件，这样就可以将 JavaScript 代码从该页面中分离出去。上面的页面中“测试”按钮上并未绑定任何事件处理函数，绑定事件处理函数被推迟到 JavaScript 脚本中完成。该应用的 JavaScript 脚本如下：

程序清单：codes\07\7.2\ajaxmvc\mvc.js

```
var objXMLHttp;
function createXMLHttpRequest()
{
    //对于 Mozilla、Firefox、Opera 等浏览器
    if (window.XMLHttpRequest)
    {
        objXMLHttp = new XMLHttpRequest();
    }
    //对于 Internet Explorer 浏览器
    else
    {
```

```
        var MSXML = ['MSXML2.XMLHTTP.5.0', 'MSXML2.XMLHTTP.4.0',
            'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
        for(var n = 0; n < MSXML.length; n ++)
        {
            try
            {
                //创建 XMLHttpRequest 对象
                objXMLHttp = new ActiveXObject(MSXML[n]);
                break;
            }
            catch(e)
            {
            }
        }
    }
}
createXMLHttpRequest();
//为 test 按钮绑定事件处理函数
document.getElementById("test").onclick=sendRequest;
function sendRequest()
{
    var url = "data";
    objXMLHttp.open("POST", url, true);
    //指定响应函数
    objXMLHttp.onreadystatechange = process;
    //发送请求
    objXMLHttp.send(null);
}
//process()是控制器函数
function process()
{
    //如果服务器响应到来
    if (objXMLHttp.readyState == 4 &&
        (objXMLHttp.status == 200 || objXMLHttp.status == 304))
    {
        //调用视图函数来加载服务器响应
        show(objXMLHttp.responseText);
    }
}
function show(content)
{
    //加载模型返回的数据
    document.getElementById("show").innerHTML=content;
}
```

从上面的粗体字代码可以看出，程序中绑定事件监听器使用了绑定到 DOM 对象的属性，控制器函数非常简单，仅仅读取模型数据（XMLHttpRequest 对象的 responseText 属性），并调用视图控制函数来加载服务器响应。在这种开发模式下，视图组件（XHTML）页面中没有 JavaScript 脚本，而且 JavaScript 脚本中各函数功能清晰，易于理解，是一种良好的开发模式。

7.3 Internet Explorer 的事件模型

除了在 7.1 节所介绍的事件模型外，不同浏览器还有自身所独有的事件机制，例如 Internet Explorer，就包含了不少独有的事件特性，这些事件特性包括 Internet Explorer 自己的事件绑定机制，以及独有的事件传播机制等。

➤➤7.3.1 使用 script for 绑定

除了 7.1 节所介绍的两种事件绑定方法外，Internet Explorer 还支持两种自己独有的事件绑定方法。这两种方法都不需要在 XHTML 元素中增加额外的属性，下面依次介绍 Internet Explorer 中的事件绑定方法：

在 Internet Explorer 4.0 以及更新的版本中，Microsoft 扩展了<script.../>元素，可以将它包含的脚本语句和某个元素的某个事件类型进行绑定，完成绑定的两个属性是 for 和 event。

注意：

此处介绍的是 Internet Explorer 独有的事件绑定机制，并没有被 W3C 组织规定为标准 HTML 的标签属性。

for 属性的值必须是 XHTML 文档中某个元素的 id 属性值，该属性值唯一标识了该 XHTML 元素。event 属性值是该元素所支持的事件名称，如 onmouseover、onclick 等。一旦为该 script 标签指定了这两个属性，就表明该标签内的所有脚本绑定了该元素的对应事件。在这种绑定机制下，事件处理的脚本语句并不在函数中，而是在<script.../>元素内，看如下的页面代码：

程序清单：codes\07\7.3\scriptfor.html

```
<!-- 简单的按钮 -->
<input type="button" id="bn1" name="bn1" value="单击我" />
<!-- 使用 script for 将下面的脚本绑定到 button1 按钮的 onclick 事件 -->
<script for="bn1" event="onclick" type="text/javascript">
    alert("您单击了我");
</script>
```

这种绑定机制比较独特，但可以正常运行。图 7.8 显示了这段脚本在 Internet Explorer 浏览器中单击按钮后弹出的对话框。

图 7.8 使用 script for 绑定事件

当然，<script.../>元素中的语句可以调用其他函数，包括从.js 文件中导入的函数。不方便的是，这种方式必须为每个元素的每个事件都建立一个<script for=""...../>标签。

值得注意的是：这种绑定方式仅支持 Internet Explorer 4.0 以及更新的版本，其他浏览器则没有实现这个特殊的<script.../>元素。因此应该尽量避免使用这种绑定方式，如果客户端的浏览器不是 Internet Explorer 4.0 或更新的版本，将不可避免地引起脚本错误。

➤➤7.3.2 使用 attachEvent 方法执行绑定

在 W3C 指定标准的事件模型之前，attachEvent()方法已经被实现了，并可被用于 Internet Explorer 5.0 以及更新的版本，该方法可以作用于浏览器中每个 XHTML 元素。这种绑定方式与 AWT、Swing 事件绑定机制非常相似。

attachEvent()方法的语法格式如下：

```
domObject.attachEvent("eventName", functionReference);
```

上面的语法格式中：eventName 的值是事件名称，例如 onmousedown。functionReference 的值是一个函数引用。

提 示：

与 AWT、Swing 事件绑定方法类似，AWT、Swing 事件绑定使用 addXxxListener(listener)，该方法只需要一个参数：listener，监听器实际起作用的部分就是它所包含的一个或多个事件处理方法。该方法之所以可以少一个参数：事件类型，是因为 addXxxListener(listener)方法名中的 Xxx 已经代表了事件类型。

下面的代码使用 attachEvent 方法完成了简单的事件绑定：

程序清单：codes\07\7.3\attachEvent.html

```
<body>
<input type="button" id="bn1" name="bn1" value="单击我" />
<script type="text/javascript">
    function test()
    {
        alert("单击按钮");
    }
    //使用 attachEvent 执行事件绑定
    document.getElementById("bn1").attachEvent("onclick" , test);
</script>
</body>
```

注意：

这种事件绑定机制只是 Internet Explorer 所提出的事实规范，并不是 W3C 制订的行业规范，Firefox 并不支持这种事件绑定机制，但 Opera 还是支持这种事件绑定机制的。

与前面介绍的事件绑定方式不同，前面介绍的绑定方式是直接将事件处理函数引用赋给 DOM 对象的 onclick 等属性，也就是说：一个 DOM 对象、一种事件最多只能绑定一个事件处理器，但采用 attachEvent() 方法绑定事件处理器时，一个 DOM 对象、一种事件可以绑定多个事件处理器。例如如下代码：

程序清单：codes\07\7.3\attachEvent2.html

```
<input type="button" id="bn1" name="bn1" value="单击我" />
<script type="text/javascript">
    function test()
    {
        alert("单击按钮");
    }
    function haha()
    {
        alert("haha 函数也被触发");
    }
    //使用 attachEvent 执行事件绑定
    document.getElementById("bn1").attachEvent("onclick" , test);
    //使用 attachEvent 执行事件绑定
    document.getElementById("bn1").attachEvent("onclick" , haha);
</script>
```

在 Internet Explorer 中浏览该页面时，单击“单击我”按钮，haha()函数先被触发（先绑定的后触发）；在 Opera 中浏览该页面时，单击“单击我”按钮，test()函数先被触发（先绑定的先触发）。

与 attachEvent()方法对应的是 detachEvent()方法，该方法用于删除一个事件处理器，其语法格式是：

```
domObject.detachEvent("eventName", functionReference);
```

执行该方法将会把一个已经注册的 functionReference 从 domObject 上删除，如果该函数还不曾注册给该对象，则执行该方法不会有任何作用。例如下面的两行代码：

```
//使用 attachEvent 执行事件绑定
document.getElementById("bn1").attachEvent("onclick" , test);
//使用 attachEvent 执行事件绑定
document.getElementById("bn1").detachEvent("onclick" , test);
```

上面的代码中粗体字代码先将 test()函数绑定到 bn1 按钮的 onclick 事件，后来又将该函数从该按钮的 onclick 事件中删除，这两行代码执行结束后，bn1 按钮上依然没有绑定事件处理函数。

➤➤7.3.3 访问事件对象

与 AWT、Swing 事件编程类似，事件对象封装了事件发生的详细信息，尤其是对鼠标、键盘事件。如果 JavaScript 脚本需要访问鼠标事件发生的位置、引发鼠标事件的鼠标键、引发键盘事件的键盘键，则需要访问事件对象。

Internet Explorer 中事件对象是个隐式可用的全局对象：event，当一个事件在浏览器中发生时，浏览器创建一个瞬态的事件对象，JavaScript 脚本通过 event 就可访问该对象。

Internet Explorer 中事件对象有如下常用的属性：

- type：返回发生的事件的类型，例如"click"等。
- srcElement：返回发生事件的 HTML 元素。
- clientX：返回发生鼠标事件在页面中的 X 坐标。
- clientY：返回发生鼠标事件在页面中的 Y 坐标。
- offsetX：返回发生鼠标事件位置相对于事件源的 X 坐标。
- offsetY：返回发生鼠标事件位置相对于事件源的 Y 坐标。
- button：对于鼠标事件有效，返回发生鼠标事件时所用的鼠标键。
- keyCode：对于键盘事件有效，返回发生键盘事件时所用的键盘键。
- altKey：返回 boolean 值，用以确定事件发生时是否按下了 Alt 组合键。
- ctrlKey：返回 boolean 值，用以确定事件发生时是否按下了 Ctrl 组合键。
- shiftKey：返回 boolean 值，用以确定事件发生时是否按下了 Shift 组合键。
- cancelBubble：阻止事件冒泡。关于事件传播请看 7.3.4 节内容。
- returnValue：返回事件处理函数的返回值。在整个事件传播链中，事件处理函数可以改变该值，当该值被设为 false 时，该事件的默认行为被取消。
- fromElement：对 mouseover 和 mouseout 两个事件有效，用于返回鼠标移出的 HTML 元素。
- toElement：对 mouseover 和 mouseout 两个事件有效，用于返回鼠标移入的 HTML 元素。

看下面的常用树形结构，该页面使用了 Internet Explorer 的全局事件对象，因而只能在 Internet Explorer 中浏览该树形结构。代码如下：

程序清单：codes\07\7.3\tree.html

```
<body>
<div id="root" class="outline">
    <img id="rootImage" class="outline" alt="rootImage"
        src="image/plus.gif" /> 我的电脑 </div>
<div id="rootDetails" style="display: none">
    <div id="child1" class="outline1" >
        <img id="child1Image" class="outline1" alt="child1Image"
            src="image/plus.gif" /> 本地磁盘 C: </div>
    <div id="child1Details" style="display: none">
        <div class="passage11">
            <img class="passage11" src="image/passage.gif" /> 文件一</div>
        <div class="passage11">
            <img class="passage11" src="image/passage.gif" /> 文件二</div>
    </div>
    <div id="child2" class="outline1">
        <img id="child2Image" class="outline1" alt="child2Image"
            src="image/plus.gif" /> 本地磁盘 D: </div>
    <div id="child2Details" style="display: none">
        <div class="passage11">
            <img class="passage11" src="image/passage.gif" /> 文件三</div>
        <div class="passage11">
            <img class="passage11" src="image/passage.gif" /> 文件四</div>
```

```
    </div>
    <div id="child3" class="outline1">
        <img id="child3Image" class="outline1" alt="child3Image"
            src="image/plus.gif" /> 本地磁盘 E: </div>
    <div id="child3Details" style="display: none">
        <div class="passage11">
            <img class="passage11" src="image/passage.gif" /> 文件五</div>
        <div class="passage11">
            <img class="passage11" src="image/passage.gif" /> 文件六</div>
    </div>
    <div class="passage1">
        <img class="passage1" src="image/passage.gif" />文件七</div>
</div>
</body>
```

这种树形结构在 Web 应用中相当常见，用途非常广泛。初学者往往觉得深不可测，其实相当简单，它主要利用了两个知识点：

- 使用 CSS 控制每个节点的缩进以及节点下内容的隐藏和显示。
- 使用 JavaScript 脚本监控鼠标单击事件。

看上面的程序中粗体字代码的 id 属性值，这些属性值之间有很直接的关系：树节点<div.../>元素的 id 为 xxx，则树节点前图片元素的 id 为 xxxImage，树节点下有一个对应于<div.../>元素的 id 为 xxxDetails。

对树形结构而言，每个树节点展开后下面将显示该节点的全部内容，即一个节点对应一个节点详细内容。也就是说，页面中每个非叶子节点都有如图 7.9 所示的结构。

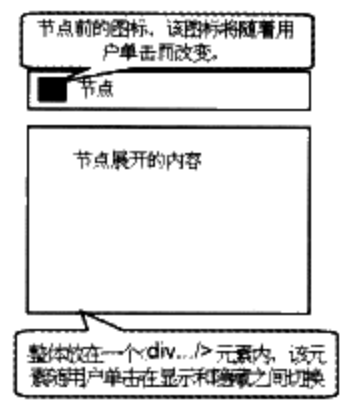


图 7.9 树形结构中非叶子节点的结构

默认情况下，节点展开的内容是隐藏的。单击鼠标时，JavaScript 控制节点展开的内容显示，再次单击鼠标时，节点展开的内容又隐藏，依次交替出现。

当用户单击节点<div.../>元素，或单击节点前的图标时，JavaScript 可通过隐式可用的 event 访问该事件对象，通过该事件对象即可访问到事件源本身——要么是树节点本身，要么是树节点前的图标，然后程序就可以根据该事件源来访问到要被操作的两个 XHTML 元素：

- 树节点前的图标。
- 树节点下“节点展开的内容”。

程序修改这两个 XHTML 元素的 CSS 样式即可实现想要的结果。下面是该页面所用的 JavaScript 代码：

程序清单：codes\07\7.3\tree.js

```
function clickHandler()
{
    //定义需要操作的 XHTML 元素 id
    var targetId;
    //定义需要被操作的 XHTML 元素
    var targetElement;
    //定义触发事件的事件源（其中 event 是隐式可用的全局对象）
    var srcElement = event.srcElement;
    //根据其 className 属性值判断它不是叶子节点，即该节点可以展开
    if (srcElement.className.substr(0,7) == "outline")
    {
        //如果事件源是树节点前的图片
        if (srcElement.id.indexOf("Image") > 0)
        {
            //获取该节点对应<div.../>元素的 id
```

```
        targetId = srcElement.id.substring(0
            , srcElement.id.length - 5) + "Details";
    }
    //如果事件源是树节点所在的<div.../>元素
    else
    {
        //获取该节点对应<div.../>元素的 id
        targetId = srcElement.id + "Details";
    }
    //找到对应的<div.../>元素
    targetElement = document.getElementById(targetId);
    //如果 targetElement 对象存在
    if (targetElement)
    {
        //如果该<div.../>元素处于“隐藏”状态
        if (targetElement.style.display == "none")
        {
            //显示该<div.../>元素
            targetElement.style.display = "";
        }
        else
        {
            //否则，隐藏该<div.../>元素
            targetElement.style.display = "none";
        }
    }
    //如果事件源是树节点前的图片
    if (srcElement.id.indexOf("Image") > 0)
    {
        //获取该节点前的<img.../>元素的 id
        targetId = srcElement.id;
    }
    //如果事件源是树节点所在的<div.../>元素
    else
    {
        //获取该节点前的<img.../>元素的 id
        targetId = srcElement.id + "Image";
    }
    //找到对应的<img.../>元素
    targetElement = document.getElementById(targetId);
    //如果该<img.../>元素中显示的图片是“加号”图片
    if (targetElement.src.indexOf("plus") >= 0)
    {
        //将<img.../>的“加号”图片换为“减号”图片
        targetElement.src = "image/minus.gif";
    }
    else
    {
        //否则，将<img.../>的“减号”图片换为“加号”图片
        targetElement.src = "image/plus.gif";
    }
  }
}
//为页面文档的 onclick 事件绑定事件处理函数
document.onclick = clickHandler;
```

上面的 JavaScript 代码主要就是根据 event 对象访问事件源，再根据事件源来访问需要被动态修改的两个对象（树节点前的图标、节点展开的内容），然后通过修改 src 属性和 CSS 样式属性来改变它们。

图 7.10 是树形结构的效果图。

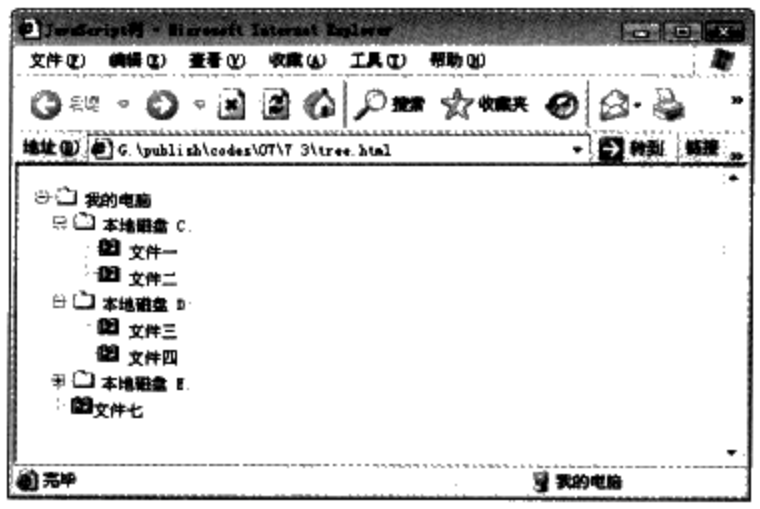


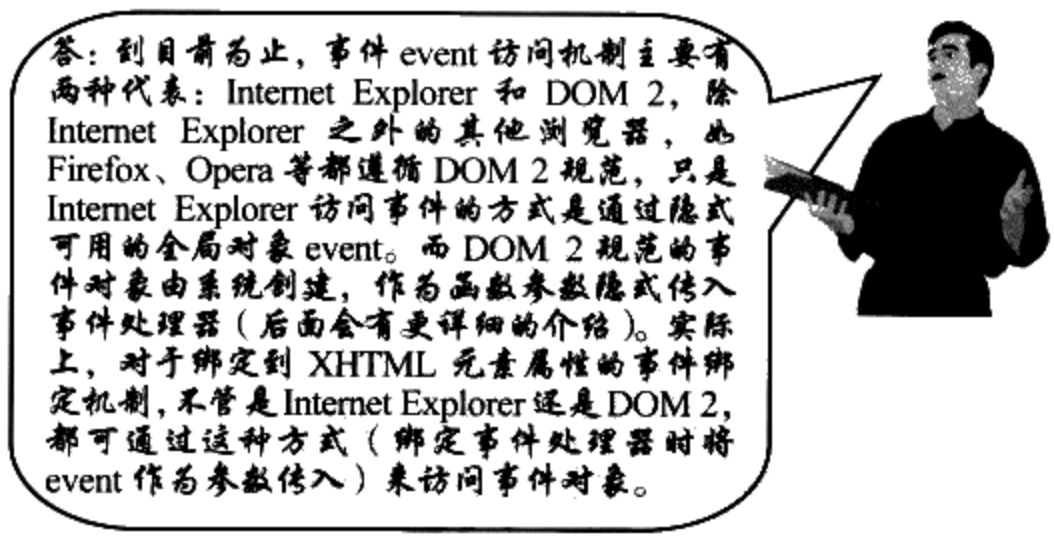
图 7.10 JavaScript 的树形结构

前面介绍的是将事件处理器绑定到 DOM 对象时获取事件的方式，如果绑定到 XHTML 元素属性，则还有另一种绑定方式：由于绑定到 XHTML 元素属性时可以传入参数，因此我们可以将 event 对象作为参数传入。看如下页面：

程序清单：codes\07\7.3\event2.html

```
<body>
<!-- 绑定事件处理器时，将 event 作为参数传入 -->
<button onclick="clickHandler(event);">按钮</button>
<script type="text/javascript">
    function clickHandler(evt)
    {
        //evt 参数由系统传入，evt 也可访问事件对象
        alert(evt.srcElement.innerHTML);
    }
</script>
</body>
```

上面的页面代码中粗体字代码为按钮绑定事件处理器时传入了一个 event 参数，该参数就是事件参数，将会传给 clickHandler()函数作为参数，因此在函数中也可通过 evt 参数来访问事件对象。



➤➤7.3.4 事件冒泡

当浏览者在页面上执行某个动作时，页面上实际上有多个元素可以响应该事件，假如我们单击页面的某个按钮，而该按钮又处于<div.../>元素之内，则实际上用户既单击了该按钮，也单击了该<div.../>元素。

Internet Explorer 中事件的传递方向是从事件发生的对象起，然后依次向该对象所在的父节点传递。因为这种传递方式是从下向上的传递，因此这种事件的传递方式也称为冒泡。并不是所有的事件都有

冒泡机制，Internet Explorer 中有些特殊的事件，例如表单提交、获得焦点等并不会冒泡。

冒泡的事件会沿着父节点一直向上，依次触发多个事件处理函数。但对于非冒泡的事件，则只在特定层次触发事件处理函数，而不会一直向上触发。看下面的简单代码：

程序清单：codes\07\7.3\bubble.html

```
<body onclick="gotClick('body元素');">
    <table onclick="gotClick('table元素');">
        <tr onclick="gotClick('tr元素');">
        <td onclick="gotClick('td元素');">
            <p onclick="gotClick('p元素');">
                <input type="button" value="单击我" onclick="gotClick('按钮');" />
            </p>
        </td>
        </tr>
    </table>
    <hr /><br />
<div id="results"> </div>
    <script type="text/javascript">
    function gotClick(who)
    {
        document.getElementById("results").innerHTML
            += who + " 被单击了 <br />";
    }
    </script>
</body>
```

上面的页面代码中定义了一个按钮，该按钮又处于一个<p.../>元素之内，该<p.../>元素又处于<td.../>元素之内，<td.../>元素又处于<tr.../>元素之内，<tr.../>元素又处于<table.../>元素之内，<table.../>元素又处于<body.../>元素之内，根据 Internet Explorer 的事件冒泡机制，当用户单击该按钮时，事件应该从按钮一直向上触发……浏览该页面，并单击该页面中的“单击我”按钮，将出现如图 7.11 所示的结果。

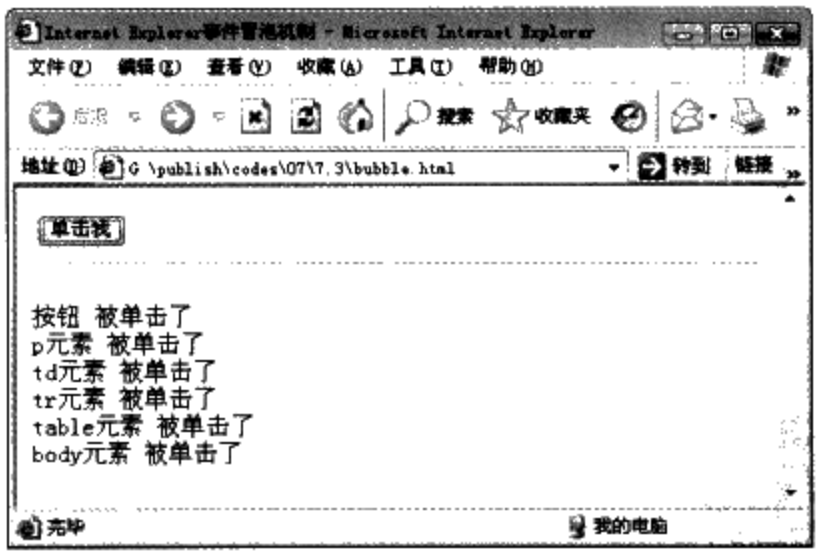


图 7.11　Internet Explorer 中事件冒泡效果

从图 7.11 可见，在按钮单击事件得到处理后，按钮的父节点元素 p 被单击的事件得到了处理，然后依次是元素 td、元素 tr、元素 table。事件从最直接的节点开始被触发，然后依次向其父节点上溯。

正如前面介绍的，并不是每个事件都可以冒泡。通常支持冒泡的事件有：onactivate、onafterupdate、onbeforeactivate、onbeforecopy、onbeforecut、onbeforedeactivate、onbeforeeditfocus、onbeforepaste、onbeforeupdate、oncellchange、onclick、oncontextmenu、oncontrolselect、oncopy、oncut、ondataavailable、ondatasetchanged、ondatasetcomplete、ondblclick、ondeactivate、ondrag、ondragend、ondragenter、ondragleave、ondragover、ondragstart、ondrop、onerrorupdate、onfocusin、onfocusout、onhelp、onkeydown、onkeypress、onkeyup、onlayoutcomplete、onmousedown、onmouesmove、onmouseout、onmouseover、

onmouseup、onmousewheel、onmove、onmoveend、onmovestart、onpaste、onresizeend、onresizestart、onrowenter、onrowsdelete、onrowsinserted、onselectstart 等。

如果想阻止冒泡，可修改 event 对象的 cancelBubble 属性，该属性的属性值默认是 true，也就是支持冒泡机制，将其设为 false 即可。

对上面冒泡测试的代码进行简单的修改，在“单击我”按钮上的 onclick 事件上添加 cancelBubble 的方法，即将该行代码修改为：

```
<!-- 阻止冒泡 -->
<input type="button" value="单击我"
    onclick="gotClick('按钮'); event.cancelBubble=true;" />
```

再次使用 Internet Explorer 浏览该页面，并单击“单击我”按钮，将出现如图 7.12 所示的界面。

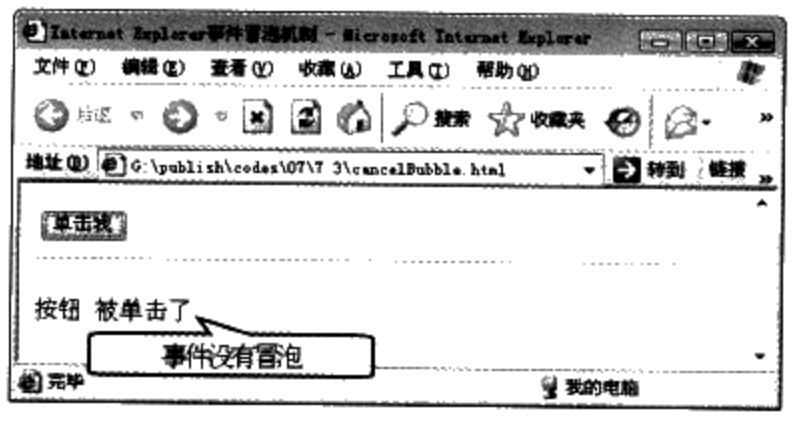


图 7.12　Internet Explorer 中阻止事件冒泡效果

7.3.5　重定向事件

事件冒泡机制严格从子节点向父节点上溯，将一路触发 DOM 树中所有的节点，这是 Internet Explorer 默认的事件触发机制。但在某些时候，我们不想让事件触发严格地按 DOM 树上溯，而是希望事件在不同节点之间跳跃，此时就可以借助 Internet Explorer 的事件重定向机制实现。

Internet Explorer 的事件重定向不仅可以将事件重定向到 DOM 树上的父节点，也可以转发到自己的子节点，甚至转发到根本不在 DOM 树上的其他节点。

Internet Explorer 的事件重定向通过 fireEvent 方法实现，该方法的语法格式是：

- target.fireEvent(String eventType , Event event)：将 event 事件重定向到 target 对象。

下面的代码示范了 Internet Explorer 事件重定向机制：

程序清单：codes\07\7.3\fireEvent.html

```
<!-- 整个 DOM 树都绑定了事件处理函数 -->
<body onclick="gotClick('body 元素');">
<table onclick="gotClick('table 元素');">
<tr onclick="gotClick('tr 元素');">
<td onclick="gotClick('td 元素');">
<p onclick="gotClick('p 元素');">
    <input type="button" value="单击我" onclick="gotClick('按钮');" />
</p>
</td>
</tr>
</table>
<input id="forward" type="button" value="被转发事件的按钮"
    onclick="gotClick('被转发的按钮')" />
<hr /><br />
<div id="results"> </div>
<script type="text/javascript">
//事件处理函数
function gotClick(who)
```

```
{
    document.getElementById("results").innerHTML +=
        who + " 被单击了 <br />";
    //取消事件的冒泡
    event.cancelBubble = true;
    //将事件重定向到 id 为 forward 的元素
    document.getElementById("forward").
        fireEvent("onClick" , event);
}
</script>
</body>
```

上面的程序中的 gotClick()函数的倒数第一行粗体字代码指定取消冒泡，倒数第二行粗体字代码指定将事件重定向到另一个按钮对象。

在 Internet Explorer 中浏览该页面，并单击“单击我”按钮，将出现如图 7.13 所示的界面。

图 7.13 事件转发测试

图 7.13 中的效果与我们的期望并不一致：“被转发的按钮 被单击了”，出现了很多次，难道该按钮真的被单击了多次？或者有多个事件被转发过来？事实是：当单击“单击我”按钮时，对应的事件处理函数显示了当前单击的按钮后，将事件转发到“被转发事件的按钮”，对应的事件处理函数启动，因而出现了相应的显示行，但请注意“被转发事件的按钮”按钮的事件处理函数的最后一行，它再次将事件转发到“被转发的事件按钮”，这是一个无限循环——当然，浏览器自动中止了这种无限循环。

如果不想出现这种无限循环，应该将“被转发的事件按钮”的事件处理函数更换成另一个事件处理函数，否则将形成无限循环。在页面脚本中增加如下函数：

程序清单：codes\07\7.3\fireEvent2.html

```
//用于处理“被转发事件的按钮”按钮的事件处理函数
function gotClick2(who)
{
    document.getElementById("results").innerHTML +=
        who + " 被单击了 <br />";
}
```

增加了上面所示的函数之后，可以为“被转发的事件按钮”按钮绑定事件处理函数，修改后代码如下：

```
<input id="forward" type="button" value="被转发事件的按钮"
    onclick="gotClick2('被转发的按钮');" />
```

gotClick2()函数最后已经没有了事件重定向，这样是否会出现期望的结果呢？在 Internet Explorer 中浏览该页面，并单击“单击我”按钮，出现如图 7.14 所示的界面。

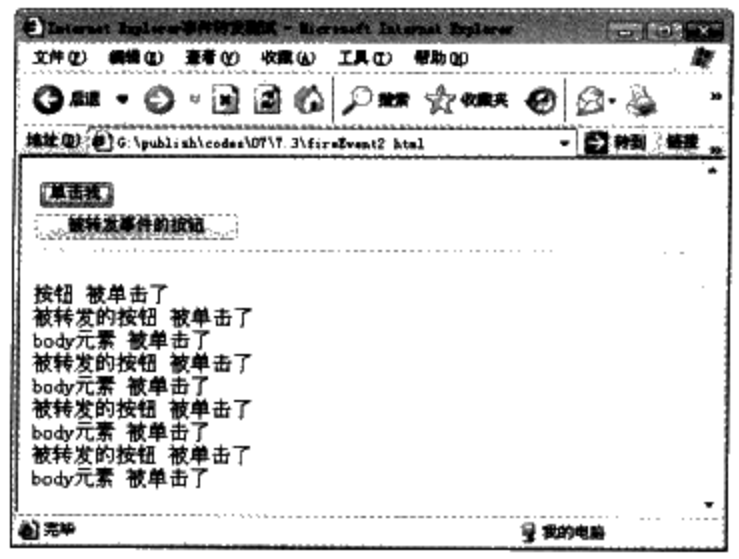


图 7.14　事件重定向

正如图 7.14 所显示的一样，当单击“单击我”按钮时，显示了“按钮 被单击了”，将事件转发到“被转发事件的按钮”，对应的事件处理函数显示了“被转发的按钮 被单击了”。注意：事件并没有在此处中止，事件将沿着该按钮所在的继承树上溯，于是就到了<body.../>元素。因而，<body.../>元素对应的事件处理函数被触发，打印出“body 元素 被单击了”，当然<body.../>元素的事件处理器（gotClick()函数）处理完事件后，事件被再次转发到“被转发事件的按钮”——这样就又进入了无限循环。为了避免这种情况，将“被转发事件的按钮”按钮的事件处理函数改为：

```
//用于处理“被转发事件的按钮”按钮的事件处理函数
function gotClick2(who)
{
    document.getElementById("results").innerHTML +=
        who + " 被单击了 <br />";
    //取消事件冒泡
    event.cancelBubble=true;
}
```

注意：

在执行事件转发时，必须要相当小心。如果想阻止事件冒泡，一定要记得将 event 的 cancelBubble 属性设置为 false 以阻止事件冒泡。

7.3.6　取消事件默认行为

前面介绍了一种取消事件默认行为的方式：让事件处理函数的返回值为 false 即可取消事件的默认行为。Internet Explorer 还提供了另一种方式来取消事件的默认行为：将事件对象的 returnValue 属性设为 false。

提 示：

前面介绍 returnValue 属性时已经提到，该属性返回事件处理函数的返回值。在整个事件传播链中，事件处理器可以改变该值，一旦该值被设为 false，该事件的默认行为被取消。

看如下页面代码：

程序清单：codes\07\7.3\returnValue.html

```
<body>
<!-- 增加了事件处理函数的超级链接 -->
<a id="wjc" href="http://www.leegang.org">疯狂 Java 联盟</a>
<script type="text/javascript">
    function clickHandler()
    {
```

```
        //使用确认对话框获取一个布尔值
        var ok = confirm('是否转入疯狂 Java 联盟? ');
        //修改 event 的 returnValue 属性值
        event.returnValue = ok;
    }
    document.getElementById("wjc").onclick=clickHandler;
</script>
</body>
```

如程序中粗体字代码所示，将一个布尔值赋给 event 的 returnValue，这样可修改 event 的 returnValue 属性值，一旦该属性值被设为 false，该事件的默认行为即被取消。

➤➤7.3.7 捕获鼠标事件

在实现某些用户界面时，我们需要处理鼠标拖动事件（例如实现下拉菜单或拖放行为），这就需要某个 XHTML 元素完全捕获鼠标事件，这样可以让事件完全不会发生冒泡。

提 示：

根据 DOM 2 事件模型，事件传播需要先后经过 2 个阶段：捕获阶段和冒泡阶段，如果捕获阶段已经阻止了事件传播，则事件传播根本不会发生冒泡。Internet Explorer 没有完全实现 DOM 2 事件模型，所以需要使用特殊方法来捕获鼠标事件。

Internet Explorer 事件模型的事件对象提供了如下两个方法来捕获事件、释放捕获：

- target.setCapture()：target 对象为安全捕获该事件。
- target.releaseCapture()：target 对象释放捕获。

这是所有 XHTML 元素的方法，一旦我们调用某元素的 setCapture()方法来捕获事件，该事件将被直接定位到该元素上、直接触发该元素上绑定的事件处理器，而根本不会发生事件冒泡！

注意：

这两个方法仅仅对鼠标事件有效，这些鼠标事件包括所有鼠标相关事件，如 mousedown、mouseup、mousemove、mouseover、mouseout、click 和 dblclick 等。

当某个元素调用了 setCapture()方法之后，在元素上发生的鼠标事件将直接定位到该元素，触发该元素上绑定的事件处理器，直到该元素调用 releaseCapture()方法或“鼠标捕获”被中断。浏览器失去焦点、弹出 alert()对话框或显示系统菜单都会导致“鼠标捕获”被中断。当该元素的“鼠标捕获”被中断时，该元素会触发“失去捕获”事件，程序可通过 onlosecapture 属性来监听该事件。

下面将会开发一个支持拖放效果的 XHTML 页面，该页面的 JavaScript 代码如下：

程序清单：codes\07\7.3\drag.js

```
function drag(target, event)
{
    //定义开始拖动时的鼠标位置（相对于 window 的坐标）
    var startX = event.clientX;
    var startY = event.clientY;
    //定义将要被拖动元素的位置（相对于 document 的坐标）
    //因为该 target 的 position 为 absolute，所以认为它所基于的坐标系是 document
    var origX = target.offsetLeft;
    var origY = target.offsetTop;
    //因为后面根据 event 的 clientX、clientY 来获取鼠标位置时，只能获取 window 坐标系
    //的位置，所以需要计算 window 坐标系和 document 坐标系的偏移
    //计算 window 坐标系和 document 坐标系之间的偏移
    var deltaX = startX - origX;
    var deltaY = startY - origY;
    //设置该元素直接捕获鼠标事件
```

```
        target.setCapture();
        //为该元素鼠标移动时绑定事件处理器
        target.attachEvent("onmousemove", moveHandler);
        //为鼠标松开时绑定事件处理器
        target.attachEvent("onmouseup", upHandler);
        //将“失去捕获”事件当成鼠标松开处理
        target.attachEvent("onlosecapture", upHandler);
        //阻止事件冒泡
        event.cancelBubble=true;
        event.returnValue = false;
        //鼠标移动的事件处理器
        function moveHandler()
        {
            //对于 IE 事件模型，获取事件对象
            var evt = window.event;
            //将被拖动元素的位置移动到当前鼠标位置
            //先将 window 坐标系位置转换成 document 坐标系位置，再修改目标对象的 CSS 位置
            target.style.left = (evt.clientX - deltaX) + "px";
            target.style.top = (evt.clientY - deltaY) + "px";
            //阻止事件冒泡
            evt.cancelBubble=true;
        }
        //鼠标松开的事件处理器
        function upHandler()
        {
            //对于 IE 事件模型，获取事件对象
            var evt = window.event;
            //取消该对象上绑定的事件处理器
            target.detachEvent("onlosecapture", upHandler);
            target.detachEvent("onmouseup", upHandler);
            target.detachEvent("onmousemove", moveHandler);
            //释放该对象的“事件捕获”
            target.releaseCapture();
            //阻止事件冒泡
            evt.cancelBubble=true;
        }
    }
```

上面的 JavaScript 代码主要定义了 drag()函数，将该函数绑定到任何 XHTML 元素的 onmousedown 属性，即可使 XHTML 元素可拖放。例如如下页面代码：

程序清单：codes\07\7.3\drag.html

```
<body>
<!-- 导入 JavaScript 脚本文件 -->
<script src="drag.js"></script>
<!-- 定义被拖放的元素 -->
<div style="position:absolute;
            left:120px;
            top:150px;
            width:250px;
            border:1px solid black;">
    <div style="background-color:#416ea5;
                width:250px;
                height:22px;
                cursor:move;
                font-weight:bold;
                border-bottom:1px solid black;"
        onmousedown="drag(this.parentNode, event);">
```

```
            可拖动标题
        </div>
        <p>可被拖动的窗口</p>
        <p>窗口内容</p>
</div>
<!-- 定义一个可拖动的图片，必须按住 Shift 才可拖动 -->
<img src="image/logo.jpg" alt="按住 Shift 可拖动"
    style="position:absolute;"
    onmousedown="if (event.shiftKey) drag(this, event);" />
</body>
```

在浏览器中浏览该页面，可看到如图 7.15 所示效果。

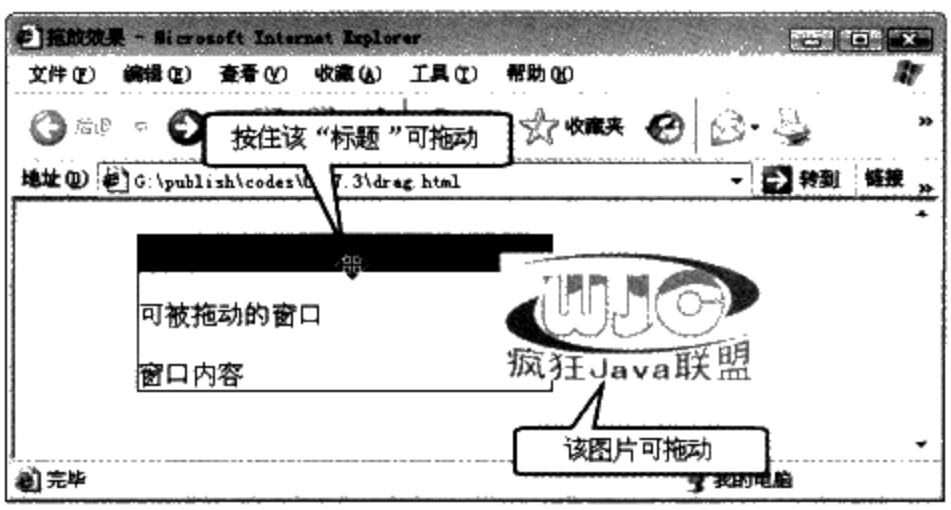


图 7.15 拖放效果

提 示：

此处介绍的拖放效果仅在 Internet Explorer 中有效，该页面大量使用了 Internet Explorer 特有的事件模型，包括访问 Internet Explorer 事件对象，取消事件默认行为，阻止事件传播等。如果需要考虑跨浏览器，必须先判断客户浏览器，再调用相应代码。

7.4 DOM 2 的事件模型

DOM 2 的事件模型是官方组织制订的事件模型，因此，它才是“正统”的事件模型， Firefox、Opera 等浏览器都支持这种事件模型。通常我们进行 JavaScript 编程时，至少需要考虑两种模型：一种是 Internet Explorer 模型，它是事实规范；一种是 DOM 2 模型，它是行业规范，Firefox、Opera 等其他浏览器都会遵守该规范。

➤➤7.4.1 绑定事件处理器

DOM 也提供了一种事件绑定机制，这种机制和 Internet Explorer 的 attachEvent()方法类似，但是有自己独特的语法。DOM 2 所提供的事件绑定方法是 addEventListener()，该方法的语法格式如下：

- objectTarget.addEventListener("eventType", handler, captureFlag)：该方法为 objectTarget 绑定事件处理器 handler，其第一个参数是事件类型字符串（将前面的事件属性去掉前缀“on”，例如 click、mousedown、keypress 等）；第二个参数是事件处理函数；第三个参数用于指定监听事件传播的哪个阶段（true 表示监听捕获阶段，false 表示监听冒泡阶段）。

下面的代码示范了使用 addEventListener()方法来绑定事件处理器：

程序清单：codes\07\7.4\addEventListener.html

```
<body>
<!-- 将测试的 div 元素 -->
<div id="test">
```

```
    <!-- div 元素的子元素：按钮 -->
    <input id="testbn" type="button" value="单击我" />
</div>
<hr /><br />
<div id="results"> </div>
<script type="text/javascript">
//事件处理函数
function gotClick1(event)
{
    //该事件处理函数简单输出事件的当前对象
    document.getElementById("results").innerHTML += "事件捕获阶段："
        + event.currentTarget + "<br />";
}
//事件处理函数
function gotClick2(event)
{
    //该事件处理函数简单输出事件的当前对象
    document.getElementById("results").innerHTML += "事件冒泡阶段："
        + event.currentTarget + "<br />";
}
//为 testbn 按钮绑定事件处理函数（捕获阶段）
document.getElementById("testbn").addEventListener("click",gotClick1, true);
//为 test 对象绑定事件处理函数（捕获阶段）
document.getElementById("test").addEventListener("click",gotClick1, true);
//为 testbn 按钮绑定事件处理函数（冒泡阶段）
document.getElementById("testbn").addEventListener("click",gotClick2, false);
//为按钮所在的 div 对象绑定事件处理函数（冒泡阶段）。
document.getElementById("test").addEventListener("click",gotClick2, false);
</script>
</body>
```

正如在上面的代码中所看到的，页面中的粗体字代码分别为按钮、<div.../>元素的 onclick 事件绑定了事件处理器，当浏览者单击“单击我”按钮时，由于该按钮处于<div.../>元素之内，所以该<div.../>元素也将被单击。

在 Firefox 中浏览该页面，并单击“单击我”按钮，将出现如图 7.16 所示的界面。

图 7.16　DOM 2 绑定事件处理器

上面的代码中为“单击我”按钮和所在的<div.../>元素分别绑定了两个事件处理函数。绑定两个事件处理函数时，最后一个参数不同，两个参数的值分别为 true 和 false，表明该元素在两个阶段都绑定对应的事件处理函数。

注意图 7.16 显示的效果：事件捕获阶段的两个事件处理函数先被触发，而事件冒泡阶段的两个事件处理函数后被触发。而且，在捕获阶段，先触发<div.../>元素；而在冒泡阶段，则先触发<input.../>元素。这与 DOM 2 的事件传播机制有关，关于 DOM 2 的事件传播机制可参阅 7.4.3 节的内容。

与 addEventListener()方法相对应，DOM 2 也提供了一个方法用于删除事件处理器，该方法为

removeEventListener，其语法格式如下：

- objectTarget.removeEventListener("eventType", handler, captureFlag)：该方法为 objectTarget 删除事件处理器 handler。其参数与 addEventListener()方法的三个参数完全类似，此处不再赘述。

➤➤7.4.2 访问事件对象

前面已经提到，DOM 2 事件模型与 Internet Explorer 事件模型访问事件对象的方式不同，在 DOM 2 的事件模型中，当浏览器检测到发生了某个事件时，将自动创建一个 Event 对象，并隐式地将该对象作为事件处理函数的第一个参数传入。

看下面的代码：

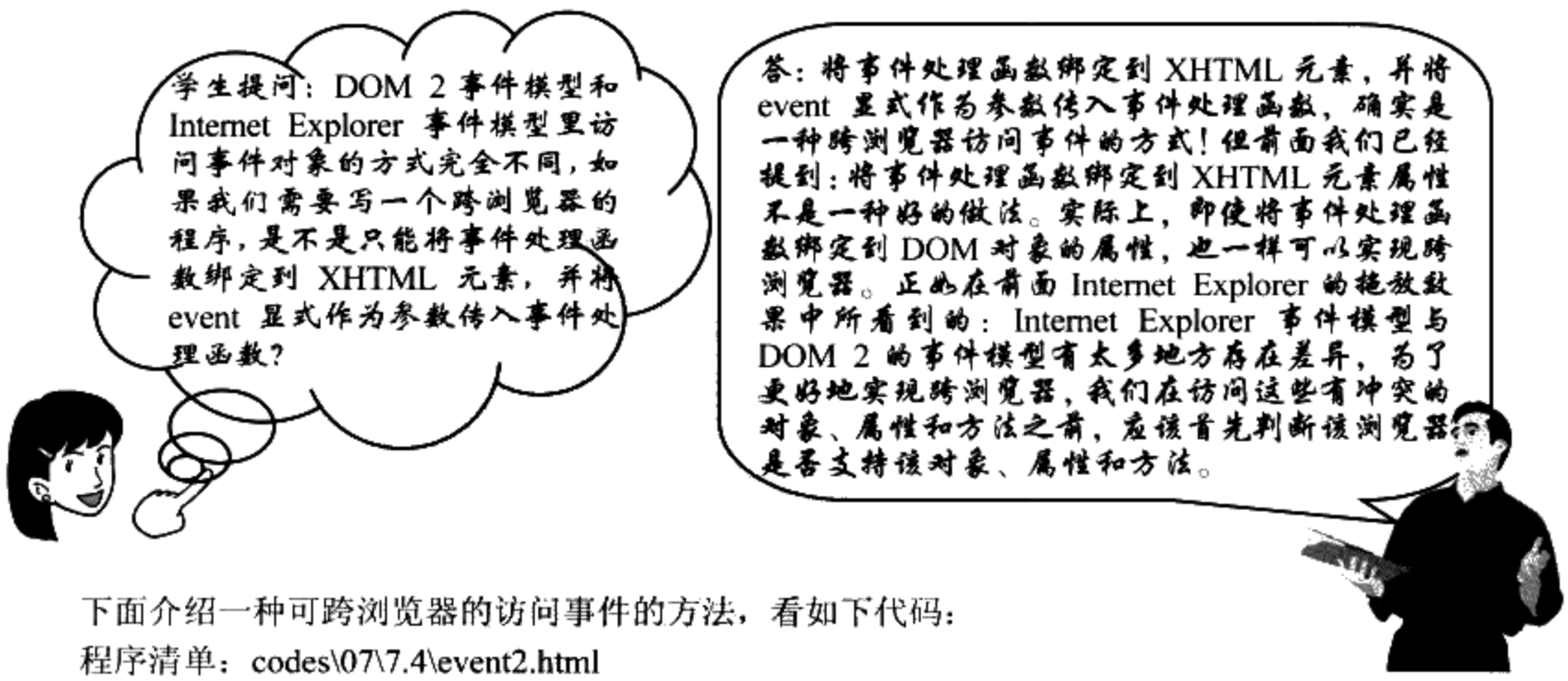
程序清单：codes\07\7.4\event.html

```
<body>
<button id="a">按钮</button>
<script type="text/javascript">
//定义一个形参 evt
function clickHandler(evt)
{
    //DOM 2 的事件对象将作为第一个参数传入 clickHandler 对象
    alert(evt.target.innerHTML);
}
//为按钮 a 绑定事件处理器
document.getElementById("a").onclick=clickHandler;
</script>
</body>
```

在上面的代码中我们看到，clickHandler()函数包含了一个 evt 参数，但该函数从未被显式调用，而是被绑定为按钮 a 的事件监听器。在 DOM 2 事件模型中，当用户单击该按钮时，浏览器会将该单击事件封装成 Event 对象，并将该对象传给 clickHandler()的 evt 参数。

图 7.17 访问 DOM 2 事件模型的事件对象

在浏览器中浏览该页面，并单击“按钮”按钮，可看到如图 7.17 所示对话框。



下面介绍一种可跨浏览器的访问事件的方法，看如下代码：

程序清单：codes\07\7.4\event2.html

```
<body>
<button id="a">按钮</button>
<script type="text/javascript">
```

```
//定义一个形参 evt
function clickHandler(evt)
{
    //对于 IE 浏览器，事件作为隐式可用的全局对象 event
    if (!evt)
    {
        evt = window.event;
    }
    //对于 DOM 2 事件模型，访问事件源用 target 属性
    if (evt.target)
    {
        alert(evt.target.innerHTML);
    }
    //对于 IE 浏览器
    else
    {
        alert(evt.srcElement.innerHTML);
    }
}
//为按钮 a 绑定事件处理器
document.getElementById("a").onclick=clickHandler;
</script>
</body>
```

上面的代码就是一份跨浏览器的代码，程序在使用属性、方法之前，总是先判断该属性、方法是否存在，从而就可以针对不同浏览器进行不同的处理。在 Firefox 中测试该页面，可看到如图 7.17 所示效果。在 Internet Explorer 中浏览该页面，单击“按钮”按钮，将可看到如图 7.18 所示效果。

图 7.18　跨浏览器地访问事件对象

注意：

上面的代码中 clickHandler()函数的形参名不要叫 event。如果该函数的形参名为 event，则该形参将会覆盖全局可用的 event 对象，从而可能导致在 Internet Explorer 中不能访问事件对象。

DOM 2 提供了一套完整的事件继承体系。DOM 2 的事件继承图如图 7.19 所示。

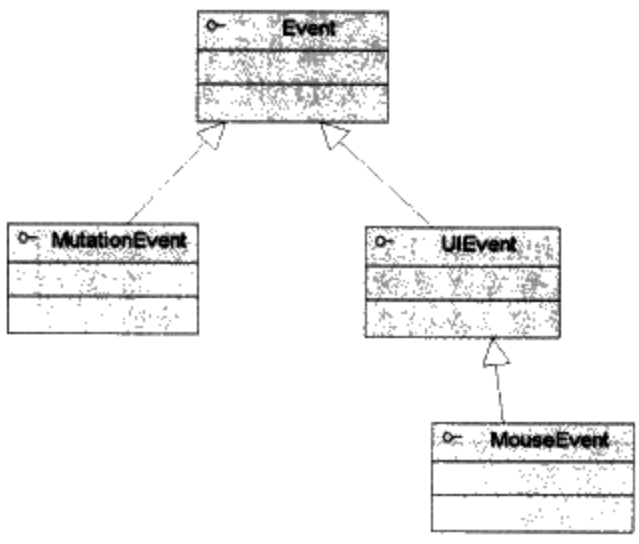


图 7.19　DOM 2 事件的继承树

DOM 2 事件模型中每个具体事件都是上面事件接口的一个实例。下面是具体的对应关系：

- Event：对应有 abort、blur、change、error、focus、load、reset、resize、scroll、select、submit、unload 等事件。
- MouseEvent：对应有 click、mousedown、mousemove、mouseout、mouseover、mouseup 等事件。
- UIEvent：对应有 DOMActivate、DOMFocusIn、DOMFocusOut 等事件。
- MutationEvent：对应有 DOMAttrModified、DOMCharacterDataModified、DOMNodeInserted、DOMNodeInsertedIntoDocument、DOMNodeRemoved、DOMNodeRemovedFromDocument、DOMSubtreeModified 等事件。

在 Event 接口里定义了如下属性：

- type：返回该事件的类型，该属性值与注册事件处理器时所用的事件类型字符串相同（例如

click、mouseover 等）。

- target：返回触发事件的事件源。
- currentTarget：返回事件当前所在的事件源。该属性值与 target 属性可以不同，如果在捕获或冒泡阶段处理该事件，则该属性值与 target 属性返回的对象并不相同。基本上，该属性可以代替事件处理器中的关键字 this。
- eventPhase：返回该事件正处在哪个阶段，可能的值有：Event.CAPTURING_PHASE（捕获阶段）、Event.AT_TARGET 或 Event.BUBBLING_PHASE（冒泡阶段）。
- timeStamp：该属性返回一个 Date 对象，代表事件的发生时间。
- bubbles：返回一个 boolean 值，用以表达该类事件是否支持冒泡。
- cancelable：返回一个 boolean 值，用以指定该事件是否有默认行为，且可以通过 preventDefault() 方法来取消该默认行为。

UIEvent 接口定义了如下两个属性：

- view：返回 window 对象，也就是发生该事件的窗口。
- detail：该属性返回一个数字，该数字可以提供一些附加意义。例如对 click、mousedown 和 mouseup 事件，event 属性返回 1 代表单击、2 代表双击、3 代表三击（鼠标每次单击都会产生一个事件，如果两次单击的时间足够接近，它们就会变成一次双击事件）。

MouseEvent 接口继承了 UIEvent，它不仅可以使用 Event 接口的所有属性，也可以访问 UIEvent 接口的全部属性，该接口里包含如下几个属性：

- button：返回一个数字代表触发事件的鼠标键。其中 0 代表鼠标左键，1 代表鼠标中键，2 代表鼠标右键。只有当浏览者改变了鼠标键状态时才可以访问该属性，例如 mousemove 事件就不可访问该属性。
- altKey、ctrlKey、metaKey、shiftKey：这四个属性都返回 boolean 值，用于显示发生该鼠标事件时，是否同时按下了 Alt、Ctrl、Meta 或 Shift 功能键。
- clientX、clientY：该属性返回鼠标事件的发生位置，该位置以浏览者的浏览器窗口作为坐标系。注意，该位置完全不考虑 document 的滚动位置，即使把浏览器滚动条拖到下面，只要鼠标事件在浏览器上方发生，clientY 属性依然是 0。DOM 2 并没有提供标准方法完成 window 坐标到 document 坐标之间的转换，所以开发者必须手动实现转换。在除 IE 之外的其他浏览器中，可以再为这两个属性分别添加 window.pageXOffset 和 window.pageYOffset 来完成 window 坐标到 document 坐标的转换。
- screenX、screenY：返回鼠标事件的发生位置，该位置以用户显示器作为鼠标位置的坐标系。当开发者试图在鼠标位置打开一个新的浏览器时，这两个属性比较有用。
- relatedTarget：返回该事件事件源的相关节点。对于 mouseover 事件，该属性值返回当鼠标划过某个 XHTML 元素之前离开的 XHTML 元素；对于 mouseout 事件，该属性值返回鼠标离开某个 XHTML 元素后立即进入的 XHTML 元素。其他事件通常没有该属性。

7.4.3 事件传播

DOM 2 中的事件先后沿两个方向传播：在第一个阶段，也就是前面提到的事件捕获阶段，事件从最顶层的对象依次向下传播，因此先触发顶层对象的事件处理函数，然后依次向下，直到传播到事件所发生的最底层对象；接着进入第二个阶段，也就是前面提到的事件冒泡阶段，事件传播从底层一直上溯，直到最顶层的对象。

整个 DOM 2 的事件传播可以分成两个阶段：捕获阶段和冒泡阶段。捕获阶段的事件触发器总是比冒泡阶段的触发器先触发：在事件捕获阶段，顶层对象的事件处理器先被触发；而事件冒泡阶段则是底层对象的事件触发器先被触发。

DOM 2 中的事件传播可以用图 7.20 表示。

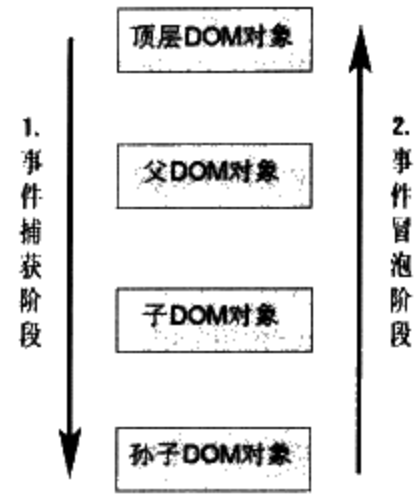


图 7.20 DOM 2 的事件传播模型图

前面的 addEventListener.html 页面和图 7.16 已经展示了 DOM 2 的事件传播机制，此处不再给出示例。

为了阻止事件传播，DOM 2 为 Event 对象提供了 stopPropagation 方法，该方法的语法格式如下：

- event. stopPropagation()：阻止 event 事件传播。如果我们在事件捕获阶段调用该方法阻止事件传播，则该事件根本不会进入事件传播阶段。

一旦调用该方法，则该事件的传播将被完全停止。值得注意的是 DOM 2 的事件传播顺序：在事件捕获阶段，事件从顶层对象传递到事件发生的目标对象；而在第二阶段，事件从事件发生的目标对象向上传播到顶层对象。

看如下代码：

程序清单：codes\07\7.4\stopPropagation.html

```
<body>
友情链接：<br />
<!-- 目标超级链接 -->
<a id="mylink" href="http://www.leegang.org">疯狂 Java 联盟</a>
<div id="show"></div>
<script type="text/javascript">
    //事件捕获阶段的处理函数
    function killClick1(event)
    {
        //取消默认事件的默认行为
        event.preventDefault();
        //阻止事件传播
        event.stopPropagation();
        document.getElementById("show").innerHTML +=
            '事件捕获阶段' + event.currentTarget + "<br />";
    }
    //事件冒泡阶段的处理函数
    function killClick2(event)
    {
        //取消事件的默认行为
        event.preventDefault();
        //阻止事件传播
        event.stopPropagation();
        document.getElementById("show").innerHTML +=
            '事件冒泡阶段' + event.currentTarget + "<br />";
    }
    //在事件捕获阶段，分别为超级链接对象、document 对象绑定事件处理函数。
    document.addEventListener("click", killClick1, true);
    document.getElementById("mylink").addEventListener(
        "click", killClick1, true);
    //在事件冒泡阶段，分别为超级链接对象、document 对象绑定事件处理函数。
    document.addEventListener("click", killClick2, false);
    document.getElementById("mylink").addEventListener(
        "click", killClick2, false);
</script>
```

根据上面的介绍，当事件传播处于捕获阶段时，从顶层对象向下传播，最先触发 document 对象的事件处理函数，该对象的事件处理函数阻止了事件传播，因而超级链接对象上的事件处理函数不会触发，事件传播更不会进入事件冒泡阶段。

在浏览器中浏览该页面，并单击“疯狂 Java 联盟”超级链接，可看到如图 7.21 所示结果。

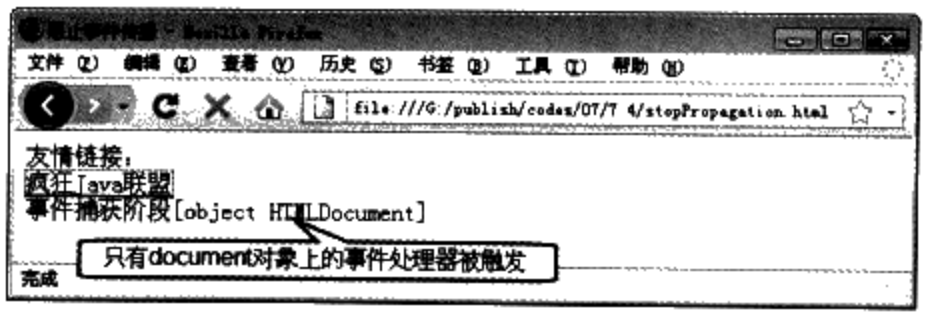
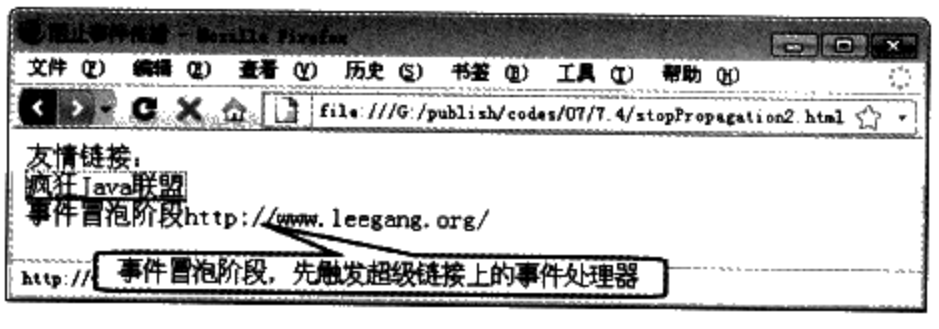


图 7.21　只有 document 对象的事件处理函数被触发

从图 7.21 可以看出，在事件捕获阶段，事件先触发 document，所以该对象上的事件处理器被触发，且阻止了事件传播，所以在图 7.21 中可看到：事件传播处于捕获阶段时，仅仅触发了 document 上的事件处理器。

如果将事件捕获阶段的事件处理函数注释掉，即只为 document 对象和超级链接对象的事件冒泡阶段绑定事件处理函数，此时超级链接对象上的事件处理函数将先被触发，然后阻止事件传播，而 document 对象上的事件处理函数将不会被触发。

在浏览器中浏览该页面，并单击“疯狂 Java 联盟”超级链接，可看到如图 7.22 所示结果。



·图 7.22　只有超级链接对象的事件处理函数被触发

提 示：

上面的程序还调用事件的 preventDefault()方法来取消事件的默认行为，关于取消事件默认行为的介绍参见本书 7.4.5 节。

DOM 2 实现了完备的事件处理机制，只要任何 XHTML 元素在捕获阶段捕获了事件，并阻止了该事件的继续传播，则事件不会进入事件传播阶段，这对实现鼠标拖放等效果已经足够。下面将实现一个跨浏览器的拖放效果，为了实现跨浏览器，程序在访问对象、属性或方法之前，总是先判断该浏览器是否支持该对象、属性或方法。代码如下：

程序清单：codes\07\7.4\drag.js

```
function drag(target, event)
{
    //定义开始拖动时的鼠标位置（相对 window 坐标）
    var startX = event.clientX;
    var startY = event.clientY;
    //定义将要被拖动元素的位置（相对于 document 坐标）
    //因为该 target 的 position 为 absolute，所以我们认为它所基于的坐标系是 document
    var origX = target.offsetLeft;
    var origY = target.offsetTop;
    //因为后面根据 event 的 clientX、clientY 来获取鼠标位置时，只能获取 window 坐标系
    //的位置，所以需要计算 window 坐标系和 document 坐标系的偏移
    //计算 window 坐标系和 document 坐标系之间的偏移
    var deltaX = startX - origX;
    var deltaY = startY - origY;
    //为被拖动对象的鼠标移动（mousemove）和鼠标松开（mouseup）注册事件处理器
    if (document.addEventListener)
```

```
{
    //DOM 2 事件模型
    //在事件捕获阶段绑定事件处理器
    document.addEventListener("mousemove", moveHandler, true);
    document.addEventListener("mouseup", upHandler, true);
}
else if (document.attachEvent)
{
    //IE 事件模型
    //设置该元素直接捕获该事件
    target.setCapture();
    //为该元素鼠标移动时绑定事件处理器
    target.attachEvent("onmousemove", moveHandler);
    //为鼠标松开时绑定事件处理器
    target.attachEvent("onmouseup", upHandler);
    //将失去捕获事件当成鼠标松开处理
    target.attachEvent("onlosecapture", upHandler);
}
//阻止事件传播
stopProp(event);
//取消事件默认行为
if (event.preventDefault)
{
    //DOM 2 事件模型
    event.preventDefault( );
}
else
{
    //IE 事件模型
    event.returnValue = false;
```

```
//鼠标移动的事件处理器
function moveHandler(evt)
{
    //对于 IE 事件模型，获取事件对象
    if (!evt) evt = window.event;
    //将被拖动元素的位置移动到当前鼠标位置
    //先将 window 坐标系位置转换成 document 坐标系位置，再修改目标对象的 CSS 位置
    target.style.left = (evt.clientX - deltaX) + "px";
    target.style.top = (evt.clientY - deltaY) + "px";
    //阻止事件传播
    stopProp(evt);
}
//鼠标松开的事件处理器
function upHandler(evt)
{
    //对于 IE 事件模型，获取事件对象
    if (!evt) evt = window.event;
    //取消被拖动对象的鼠标移动（mousemove）和鼠标松开（mouseup）的事件处理器
    if (document.removeEventListener)
    {
        //DOM 2 事件模型
        //取消在事件捕获阶段的事件处理器
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler, true);
    }
    else if (document.detachEvent)
    {
```

```
            //取消该对象上绑定的事件处理器
            target.detachEvent("onlosecapture", upHandler);
            target.detachEvent("onmouseup", upHandler);
            target.detachEvent("onmousemove", moveHandler);
            target.releaseCapture( );
        }
        //阻止事件传播
        stopProp(evt);
    }
    //阻止事件传播（该函数可以跨浏览器）
    function stopProp(evt)
    {
        //DOM 2 事件模型
        if (evt.stopPropagation)
        {
            evt.stopPropagation( );
        }
        else
        {
            //事件模型
            evt.cancelBubble = true;
        }
    }
}
```

上面的程序中粗体字代码用于对浏览器进行判断，判断浏览器到底是支持 DOM 2 事件模型，还是支持 Internet Explorer 事件模型，在确定浏览器支持某种事件模型后，然后才调用相应方法来处理鼠标的按下、拖动和松开事件。

该拖放效果所使用的 XHTML 页面与 7.3 节所使用的页面完全相同，在 Firefox 浏览器中浏览该页面，可看到如图 7.23 所示效果。

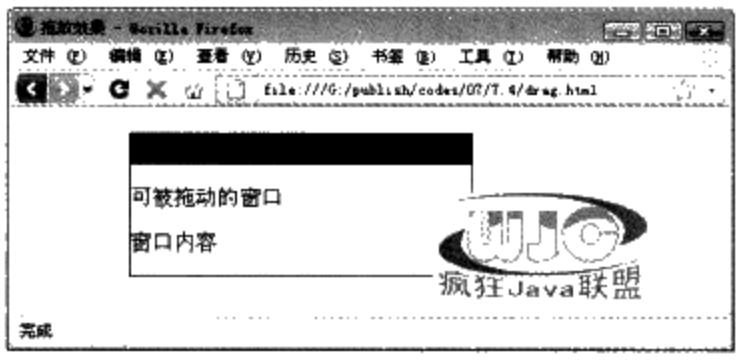


图 7.23　跨浏览器的拖放效果

➤➤7.4.4　转发事件

DOM 2 提供了 dispatchEvent 方法用于事件的转发，该方法属于 Node 对象，因此，DOM 2 的每个 Node 元素都可调用该方法，从而将事件直接转发到本节点。该方法的语法格式如下：

- target.dispatchEvent(Event event)：将 event 事件转发到 target 上。

与 Internet Explorer 事件模型里的重定向事件不同的是，dispatch()方法必须转发人工合成事件（Synthetic Event），不能直接转发系统创建的事件。

DOM 2 为创建人工合成事件提供了如下方法：

- document.createEvent(String type)：该方法创建一个事件对象，其中 type 参数指定事件类型，普通事件可使用 Events，UI 事件可使用 UIEvents，鼠标事件可使用 MouseEvents。

通过上面的方法得到一个事件后，可调用事件的如下方法来初始化：

- initEvent(String eventTypeArg,boolean canBubbleArg, boolean cancelableArg)：用于初始化一个

普通事件，第一个参数指定该事件类型，如 click 等，第二个参数指定该事件是否支持冒泡，第三个参数指定该事件是否有默认行为，且可通过 preventDefault()方法取消该默认行为。

- initUIEvent(String typeArg, boolean canBubbleArg, boolean cancelableArg, Window viewArg, long detailArg)：该方法的前 3 个参数与前一个方法中 3 个参数的意义完全相同。后两个参数与介绍 UIEvent 时的 view、detail 两个属性的意义相同。
- initMouseEvent(String typeArg, boolean canBubbleArg, boolean cancelableArg, AbstractView viewArg, long detailArg, long screenXArg, long screenYArg, long clientXArg, long clientYArg, boolean ctrlKeyArg,boolean altKeyArg, boolean shiftKeyArg, boolean metaKeyArg, unsigned short buttonArg, Element relatedTargetArg)：该方法里的前 5 个参数与上一个方法中 5 个参数的意义完全相同。后面的参数与介绍 MouseEvent 时各属性的意义相同。

看如下代码：

程序清单：codes\07\7.4\drag.js

```
<body>
<!-- 测试用的第一个按钮 -->
<input id="bn1" type="button" value="按钮 1" />
<!-- 测试用的第二个按钮 -->
<input id="bn2" type="button" value="按钮 2" />
<div id="show"></div>
<script type="text/javascript">
//第一个按钮被单击时的事件处理函数
function rd(evt)
{
    document.getElementById("show").innerHTML +=
        '事件冒泡阶段:' + evt.currentTarget.value + "被单击了<br />";
    //创建一个普通事件
    var e = document.createEvent("Events");
    //初始化事件对象，指定该事件支持冒泡，不允许取消默认行为
    e.initEvent("click", true, false);
    //将事件转发到按钮 bn2
    document.getElementById("bn2").dispatchEvent(e);
}
//第二个按钮被单击时的事件处理函数
function gotClick(evt)
{
    document.getElementById("show").innerHTML +=
        '事件冒泡阶段:' + evt.currentTarget.value + "<br />";
}
//分别为两个按钮绑定事件处理函数
document.getElementById("bn1").addEventListener("click", rd, false);
document.getElementById("bn2").addEventListener("click", gotClick, false);
</script>
</body>
```

上面程序中粗体字代码创建了人工合成事件，并将该事件初始化为 click 事件，然后将其转发到按钮 bn2，在浏览器中浏览该页面，并单击按钮“按钮 1”，可看到如图 7.24 所示结果。

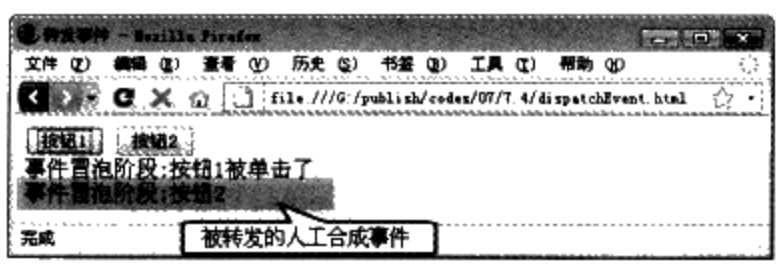


图 7.24　转发人工合成事件

➤➤7.4.5 取消事件的默认行为

DOM 2 也提供了取消事件默认行为的方法，DOM 2 中的事件对象都提供了 preventDefault()方法，该方法不需要参数，只要执行了给定事件的 preventDefault 方法，该事件的默认行为就将失效。该方法的语法格式如下：

- event.preventDefault()：取消 event 事件的默认行为。

下面的代码示范了如何利用 preventDefault()方法来取消事件的默认行为：

程序清单：codes\07\7.4\preventDefault.html

```
<body>
友情链接：<br />
<a id="mylink" href="http://www.leegang.org">疯狂 Java 联盟</a>
<script type="text/javascript">
function killClicks(event)
{
    //取消事件的默认行为
    event.preventDefault();
    alert("超级链接被单击");
}
//为按钮绑定事件处理函数（捕获阶段）
document.getElementById("mylink")
    .addEventListener("click", killClicks, true);
</script>
</body>
```

上面的代码使用 preventDefault 方法取消了事件的默认行为，值得注意的是：该方法虽然取消了事件的默认行为，但并未阻止事件处理函数的执行，也不会影响事件的传播。图 7.25 显示了单击超级链接的结果。

图 7.25 阻止事件的默认行为

如图 7.25 所示，用户单击该超级链接，页面弹出了警告对话框，但由于使用 preventDefault()方法取消了单击事件的默认行为，因此页面不会导航到“疯狂 Java 联盟”站点。

preventDefault()方法虽然取消了事件的默认行为，但不会阻止事件的传播，下面的代码为超级链接和 document 在事件传播阶段绑定了事件处理函数：

程序清单：codes\07\7.4\preventDefault2.html

```
<body>
友情链接：<br />
<a id="mylink" href="http://www.leegang.org">疯狂 Java 联盟</a>
<!-- 显示信息输出的 div 元素 -->
<div id="show"></div>
<script>
function killClicks(event)
{
    //取消事件的默认行为
    event.preventDefault();
    document.getElementById("show").innerHTML
        += "事件捕获阶段：" + event.currentTarget + "<br>";
}
```

```
//为 document 对象绑定事件处理函数
document.addEventListener("click", killClicks, true);
//为超级链接绑定事件处理函数
document.getElementById("mylink")
    .addEventListener("click", killClicks, true);
</script>
</body>
```

在浏览器中浏览该页面，并单击“疯狂 Java 联盟”超级链接，可看到如图 7.26 所示结果。

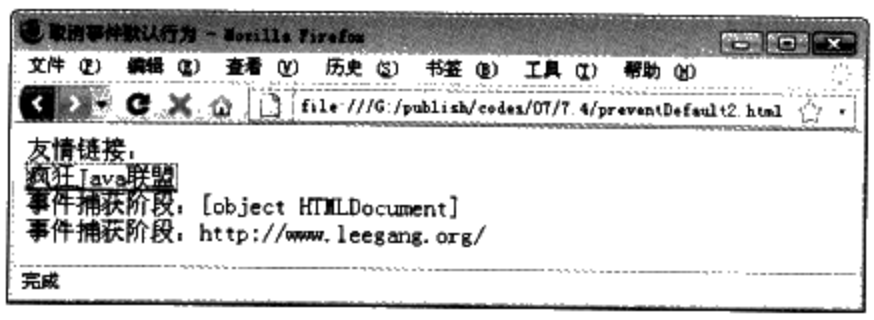


图 7.26　取消事件的默认行为并未阻止事件传播

7.5　本章小结

本章详细介绍了 JavaScript 的事件处理机制和基本的事件模型，包括两种绑定事件处理器的方式，通过返回值改变事件的默认行为，通过脚本手动触发事件等。因为事件处理器在 Ajax 应用中充当了非常重要的角色，所以本章还对 Ajax 应用进行了规范的 MVC 划分。

本章的难点是 Internet Explorer 和 DOM 2 两种事件模型。本章分别介绍了两种事件模型在实现机制上的详细差异：包括如何绑定事件处理器，如何访问事件对象，事件传播机制，如何阻止事件传播，如何取消事件的默认行为等。因为这两种事件模型的差异如此之大，如果我们需要编写跨浏览器的 JavaScript 代码，那就必须同时考虑这两种事件模型，先判断客户端浏览器支持哪种事件模型，再针对特定事件模型编写特定的代码。

➤➤本章练习

结合 7.3.3 节和 7.3.7 节的知识，开发一个可动态增加、删除节点，而且节点可拖动的树。

开发一个日期选择器。日期选择器是互联网页上的常用控件，它包含一个不可编辑的单行文本框和一个日期选择框（实质上是<div.../>元素），当用户选中某个日期时，该日期将自动填入上面的单行文本框中。图 7.27 显示了日期选择器的效果。

图 7.27　日期选择器

阅读 Prototype 库的源代码，尽可能理解源代码的作用，并为之尽可能多地添加注释。

CHAPTER

8

第8章
XMLHttpRequest 对象详解

本章要点

- XMLHttpRequest 对象的基本知识
- XMLHttpRequest 的常用属性
- XMLHttpRequest 的常用方法
- 发送 GET 请求
- 发送 POST 请求
- GET 请求和 POST 请求的区别
- 处理请求参数的编码问题
- 发送 XML 请求
- 处理服务器响应的时机
- 生成普通的文本响应
- 生成 XML 响应
- Ajax 的生命周期
- Ajax 应用的问题及解决方法

XMLHttpRequest 对象是整个 Ajax 技术中的核心，缺少了它，Ajax 的其余技术就无法成为一个有机的整体，将会土崩瓦解。Ajax 技术赖以存在的核心就是：异步发送请求。异步发送请求是根本，不刷新页面动态加载只是表面的现象。Ajax 技术离开了 XMLHttpRequest 对象，将失去与服务器异步通信的能力。

从第 2 章的介绍我们知道，在 Ajax 应用中我们以 XMLHttpRequest 对象异步发送请求，这种请求既可以是 GET 请求，也可以是 POST 请求，一样都可以发送请求参数。与传统 Web 应用中发送请求不同，Ajax 必须以编程方式来发送请求。在请求发送出去之后，服务器响应会在合适的时候返回，但客户端浏览器不会自动加载这种异步响应，程序必须先调用 XMLHttpRequest 对象的 responseText 或 responseXML 来获取服务器响应，再通过 DOM 操作将服务器响应动态加载到当前页面中。

8.1 XMLHttpRequest 对象概述

1999 年上半年，Microsoft 在 Internet Explorer 5.0 中首次使用了一种新技术，通过使用这种新技术，浏览者不用从当前 Web 页面跳转，或者使用表单提交来发送请求，而是可以直接在当前页面中发送请求到服务器，也可以从服务器读取数据。他们的这种技术的实现依赖于一个特殊的 ActiveX 对象，这个对象就是 XMLHTTP。

在此之前，能够做到直接与服务器通信的唯一技术是通过 iframe。这个功能相当重要：因为能减少无状态连接的痛苦，还可以减少下载冗余 HTML 代码，从而提高响应速度。

于是，Microsoft 干脆发布了 XMLHTTP 的 ActiveX 对象，让开发者也能异步与服务器交互。XMLHTTP 对象大受欢迎，到了 2000 年它几乎已经成为了事实上的标准。Mozilla 在这一年实现了具有相同接口的原生对象，称为 XMLHttpRequest 对象。后来 Opera、Safari 等浏览器也都相继实现了 XMLHttpRequest 对象。于是，XMLHttpRequest 就成了这个技术的正式名称。

关于 XMLHttpRequest 最通用的定义是：XMLHttpRequest 是一套可以在 JavaScript、VBScript、JScript 等脚本语言中使用的 API，它通过 HTTP 协议异步地向服务器发送请求，并可以获取从服务器返回的响应。XMLHttpRequest 的用处是：提供与服务器异步通信的能力。

根据 MSDN 的解释：XMLHTTP 提供客户端同 HTTP 服务器通讯的协议。客户端可以通过 XMLHTTP 对象向服务器发送请求，并使用微软 XML 文档对象模型（DOM）来处理服务器的响应。

XMLHttpRequest 也是 Ajax 技术中唯一一个尚未正式标准化的部分。不过，W3C 已经将其列入了工作草案，应该很快就会成为正式的标准。

8.2 XMLHttpRequest 的方法和属性

XMLHttpRequest 包含了一些基本的属性和方法，XMLHttpRequest 正是通过这些属性和方法与服务器通信的。Ajax 则依赖于 XMLHttpRequest 来完成与服务器的通信，XMLHttpRequest 是 Ajax 与服务器异步通信的核心。

通过 XMLHttpRequest 对象与服务器异步通信的能力，可避免每次发送请求对应一个页面的模式，从而允许在一个页面发送多次异步请求，每次只从服务器读取必需的信息。通过使用 Ajax，既可以减轻服务器的负担，又可以加快响应速度、缩短用户等待时间。

➤➤8.2.1 XMLHttpRequest 的方法

XmlHttpRequest 对象的方法并不多，下面是其基本的方法：

- abort()：停止发送当前请求。
- getAllResponseHeaders()：获取服务器返回的全部响应头。
- getResponseHeader("headerLabel")：根据响应头的名字，获取对应的响应头。

- open("method","URL"[,asyncFlag[,"userName"[, "password"]]])：建立与服务器 URL 的连接，并设置请求的方法，以及是否使用异步请求。如果远程服务需要用户名、密码，则提供对应的用户名和密码。
- send(content)：发送请求。其中 content 是请求参数。
- setRequestHeader("label", "value")：在发送请求之前，先设置请求头。

下面依次介绍这些常用方法的具体用法：

在请求被发送之后，getAllResponseHeaders 和 getResponseHeader 这两个方法，可用于获取服务器响应头。

虽然 getAllResponseHeaders 方法用于返回全部的响应头，但其返回值并不是一个数组，也不是一个对象，而是一个字符串：由所有响应头的“名:值”对所组成的字符串。即如下形式：

```
Server: Apache-Coyote/1.1 Content-Type: text/html;charset=GBK Content-Length: 38
    Date: Fri, 20 Oct 2006 08:07:59 GMT
```

具体有多少个响应头，则取决于生成响应的程序设置了多少。看下面的 XHTML 页面，该页面向服务器异步发送请求，处理响应时将所有的响应头全部输出。

程序清单：codes\08\8.2\getAllResponseHeaders\first.html

```
<body>
<select name="first" id="first" onchange="change(this.value);">
    <option value="1" selected="selected">中国</option>
    <option value="2">美国</option>
    <option value="3">日本</option>
</select>
<div id="output"></div>
<script type="text/javascript">
    //定义 XMLHttpRequest 对象
    var xmlrequest;
    //完成 XMLHttpRequest 对象的初始化
    function createXMLHttpRequest()
    {
        if(window.XMLHttpRequest)
        {
            //DOM 2 浏览器
            xmlrequest = new XMLHttpRequest();
        }
        else if (window.ActiveXObject)
        {
            // IE 浏览器
            try
            {
                xmlrequest = new ActiveXObject("Msxml2.XMLHTTP");
            }
            catch (e)
            {
                try
                {
                    xmlrequest = new ActiveXObject("Microsoft.XMLHTTP");
                }
                catch (e)
                {
                }
            }
        }
    }
    //事件处理函数，当下拉列表选择改变时触发该事件
```

```
function change(id)
{
    //初始化 XMLHttpRequest 对象
    createXMLHttpRequest();
    //设置请求响应的 URL
    var uri = "second.jsp?id=" + id;
    //打开与服务器响应地址的连接
    xmlrequest.open("POST", uri, true);
    //设置请求头
    xmlrequest.setRequestHeader("Content-Type"
    , "application/x-www-form-urlencoded");
    //设置处理响应的回调函数
    xmlrequest.onreadystatechange = processResponse;
    //发送请求
    xmlrequest.send(null);
}
//定义处理响应的回调函数
function processResponse()
{
    //响应完成且响应正常
    if (xmlrequest.readyState == 4)
    {
        if (xmlrequest.status == 200)
        {
            //信息已经成功返回，开始处理信息
            var headers = xmlrequest.getAllResponseHeaders();
            //通过警告框输出请求头
            alert("请求头的类型：" + typeof headers + "\n"
                + headers);
            //在页面中输出所有请求头
            document.getElementById("output").innerHTML = headers;
        }
        else
        {
            //页面不正常
            window.alert("您所请求的页面有异常。");
        }
    }
}
</script>
</body>
```

上面的代码中第一行粗体字代码指定当下拉列表框的所选项发生改变时将会触发 change()函数，该函数会向服务器 second.jsp 页面发送异步请求。系统采用 POST 方法发送请求。当请求得到响应后，使用 getAllResponseHeaders 方法获取所有的请求头，并将请求头以警告框、页面输出两种方式输出。

上面的页面中斜体字代码用于创建一个 XMLHttpRequest 对象，这段代码不一定十分标准，而且每次创建 XMLHttpRequest 对象的代码几乎完全相同，故读者无须过多关注斜体字代码。

上面的页面是级联下拉列表的实例，当选择不同国家时，该国家对应的城市将在下面显示，该应用的 second.jsp 页面的代码如下：

程序清单：codes\08\8.2\getAllResponseHeaders\second.jsp

```
<%-- 编译指定设置 JSP 页面的内容、字符集 --%>
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
int id = Integer.parseInt(request.getParameter("id"));
System.out.println(id);
switch(id)
```

```
{
    case 1:
%>
上海$广州$北京
<%
    break;
    case 2:
%>
华盛顿$纽约$加州
<%
    break;
    case 3:
%>
东京$大阪$福冈
<%
    break;
}
%>
```

上面的页面只是一个简单的 JSP 页面，该页面将会对客户端异步请求生成响应。客户端获取服务器响应时，并未处理响应数据，而是获取了响应头，响应头通过两种方式输出，一种是通过警告对话框，另一种是通过页面<div.../>元素加载。

图 8.1 获取全部响应头

在浏览器中浏览该页面，并改变下拉列表框的选中选项，将弹出如图 8.1 所示的警告框，该框的第一行是所获取全部响应头的类型。后面各行是响应头以及对应的值。

上面的程序发送请求时调用了 setRequestHeader()方法设置请求头。因为在发送 POST 请求时应设置对应的编码方式。XMLHttpRequest 提供的 open()和 send()方法主要用于发送请求，发送请求的相关知识将在 8.3 节介绍。

注意：

本章所介绍的都是完整的 Web 应用，因此读者需要掌握足够的 Java Web 知识，包括如何部署 Web 应用，开发 JSP、Servlet 程序。如果读者还没有这方面的知识，请先阅读疯狂 Java 体系的《轻量级 Java EE 企业应用实战》。

就笔者教过的很多学生来看，他们在调试 Ajax 应用时往往容易陷入一个怪圈：他们一直调试 XHTML 页面的 JavaScript 脚本，对服务器响应却不甚理会。实际上，我们调试 XHTML 页面的 JavaScript 脚本之前，应该先保证服务器响应正确，例如我们此处直接请求服务器的 second.jsp 页面，将可看到如图 8.2 所示页面。

图 8.2 先保证服务器响应正确

提示：

由于 JavaScript 代码本身缺乏有效的错误提示机制，如果我们不能先保证服务器响应完全正确，而是一开始就直接纠缠于 XHTML 页面的 JavaScript 代码，则可能因为服务器本身没有返回正确的响应，而将我们的调试引入歧途。

➤➤8.2.2 XMLHttpRequest 的属性

XMLHttpRequest 的属性也很简单，Ajax 技术通过 XMLHttpRequest 的这些简单属性实现与服务器的异步通信。

XMLHttpRequest 对象常用的属性有：

- onreadystatechange：该属性用于指定 XMLHttpRequest 对象状态改变时的事件处理函数。
- readyState：XMLHttpRequest 对象的处理状态。
- responseText：该属性用于获取服务器的响应文本。
- responseXML：该属性用于获取服务器响应的 XML 文档对象。
- status：该属性是服务器返回的状态码，只有当服务器的响应已经完成时，才会有该状态码。
- statusText：该属性是服务器返回的状态文本信息，只有当服务器的响应已经完成时，才会有该状态文本信息。

8.2.2.1 onreadystatechange 和 readyState 属性

XMLHttpRequest 的这些属性都非常有用，onreadystatechange 属性用于指定 XMLHttpRequest 对象的状态改变函数。当 XMLHttpRequest 对象的状态改变时，该函数将被触发。

提 示：

onreadystatechange 属性的作用与按钮对象的 onclick 属性一样，它们都是事件处理属性。也就是说，XMLHttpRequest 对象是事件源，它可以引发 readystatechange 事件，当程序将一个函数引用赋给 XMLHttpRequest 对象的 onreadystatechange 属性后，该函数即成为 XMLHttpRequest 对象的事件处理器，每次 XMLHttpRequest 发生 readystatechange 事件，都会触发监听该事件的事件处理器。

XMLHttpRequest 对象有如下几种状态：

- 0：XMLHttpRequest 对象还没有完成初始化。
- 1：XMLHttpRequest 对象开始发送请求。
- 2：XMLHttpRequest 对象的请求发送完成。
- 3：XMLHttpRequest 对象开始读取服务器的响应。
- 4：XMLHttpRequest 对象读取服务器响应结束。

XMLHttpRequest 对象的这几种状态信息可通过 readyState 属性读取。因此可以这样理解：每当 XMLHttpRequest 对象的 readyState 属性改变时，其 onreadystatechange 属性指定的方法都会被触发。

修改 8.2.1 节示例，将该示例中的 first.html 页面代码改为如下形式：

程序清单：codes\08\8.2\readyState\first.html

```
//XMLHttpRequest 对象状态改变时的事件处理函数
function processResponse()
{
    //输出 XMLHttpRequest 对象的状态。
    alert(xmlrequest.readyState);
    }
```

从上面的事件处理函数的代码可以看出，该函数仅仅用来监控 XMLHttpRequest 对象的 readyState 属性值的改变，当该属性值发生改变时，该事件监听函数输出 readyState 属性值。

该应用的 second.jsp 页面没有任何修改。

浏览该应用的 first.html 页面，然后改变其中下拉列表的选择，可看到依次弹出一系列的警告对话框。当弹出如图 8.3 所示对话框后，可看到服务器的控制台输出一个数字，该数字就是所选择国家的编号（second.jsp 页面的 System.out.println(id);代码输出该编号）。

图 8.3 XMLHttpRequest 的 readyState

服务器的控制台输出了请求参数，这表明当readyState状态为2时，服务器可以获取到XMLHttpRequest发送的请求参数。也就是说，readyState状态为2标志着XMLHttpRequest的请求发送完成。单击如图8.3所示的对话框中的“确定”按钮，还会依次弹出“3”、“4”两个对话框。当单击了“4”对话框中的“确定”按钮后，不会弹出任何的对话框。此时，服务器的响应已经完成。

注意：

这两个属性名在Internet Explorer中可以不区分大小写，但在其他浏览器中将严格区分大小写。只能写成onreadystatechange，而不能写成onReadyStateChange等形式。为了保证更好的跨浏览器效果，建议按严格的区分大小写的形式进行。

8.2.2.2 status和statusText属性

服务器的响应已经完成后，依然不能直接获取服务器响应。因为服务器响应也有很多种情况，看如图8.4所示的常见提示页面。

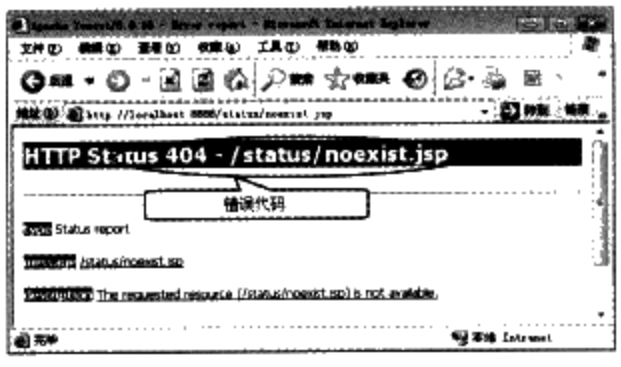


图8.4 系统错误提示页

图8.4是使用浏览器访问一个并不存在的资源时，服务器自动生成的错误提示页。在该页面的上面，有“HTTP Status 404”字符串，表明服务器响应的状态码为404，404表示资源不存在——即使资源不存在，服务器一样会生成响应。也就是说，即使程序判断XMLHttpRequest的readyState为4，服务器响应已经完成，但从服务器获取的响应信息依然有可能是错误的。

为了判断服务器的响应是否正确，可以检测XMLHttpRequest的status或statusText属性，将上面XHTML页面的回调函数改为如下形式：

程序清单：codes\08\8.2\status\first.html

```
//XMLHttpRequest 对象状态改变时的事件处理函数
function processResponse()
{
    //当服务器响应完成时
    if(xmlrequest.readyState == 4)
    {
        //输出服务器相应的状态码和状态提示
        alert(xmlrequest.status + "\n"
            + xmlrequest.statusText);
    }
}
```

上面的回调函数表明，当服务器响应完成时，将通过警告对话框输出服务器响应的状态码和状态提示。为了让服务器响应生成错误信息，将sencond.jsp页面修改成如下形式，该页面中的粗体字代码将引发空指针异常。

程序清单：codes\08\8.2\status\second.jsp

```
<%-- 编译指定设置 JSP 页面的内容、字符集 --%>
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
//定义一个空字符串。
```

```
String a = null;
//让下面的语句引发空指针异常
out.println(a.length);
%>
```

再次在浏览器中浏览 first.html 页面，并改变下拉选择列表的选项，从而向服务器发送请求，在浏览器中可看到如图 8.5 所示的对话框。

如果将服务器的响应页面 second.jsp 改回原来的形式，服务器即可生成正常响应。在浏览器中发送请求，将可看到如图 8.6 所示的警告框。

图 8.5　服务器内部错误的 status 和 statusText

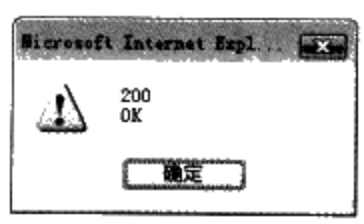


图 8.6　服务器正常响应的 status 和 statusText

通过检测 XMLHttpRequest 对象的 status 和 statusText 属性，即可判断服务器的响应是否正常。当服务器的响应正常时，JavaScript 才应该读取服务器响应信息，并将响应信息动态加载到目标页面。服务器常用的状态码及其对应的含义如下：

- 200：服务器响应正常。
- 304：该资源在上次请求之后没有任何修改，这通常用于浏览器的缓存机制。使用 GET 请求时尤其需要注意。
- 400：无法找到请求的资源。
- 401：访问资源的权限不够。
- 403：没有权限访问资源。
- 404：需要访问的资源不存在。
- 405：需要访问的资源被禁止。
- 407：访问的资源需要代理身份验证。
- 414：请求的 URL 太长。
- 500：服务器内部错误。

通过上面的介绍可以得到一个结论：如果想通过 JavaScript 获取服务器响应，必须先判断服务器响应是否完成。要判断服务器的响应是否完成，只需判断 XMLHttpRequest 对象的 readyState 属性即可，当 readyState 值为 4 时，代表响应完成；服务器响应完成后，还应判断服务器响应是否正确，判断服务器响应是否正确，可通过判断 XMLHttpRequest 对象的 status 属性进行。当 status 值为 200 时，服务器响应正确，否则响应不正常。

注意：

实际应用中往往也需要对服务器响应不正常的情况进行处理，例如弹出出错提示，告诉浏览者服务器响应出错。如果服务器响应出现了错误，但页面没有输出任何提示，则对浏览者来说是一个巨大的困惑。

8.3　发送请求

Ajax 与传统 Web 应用的第一个区别在于发送请求的方式：传统 Web 应用采用表单或请求某个资源的方式发送请求；而 Ajax 则采用异步方式在后台发送请求。下面我们将详细介绍 Ajax 发送请求的各种细节。

➤➤8.3.1 发送简单请求

所谓简单请求，指不包含任何参数的请求。这种请求通常用于自动刷新的应用，例如证券交易所的实时信息发送。这种请求通常用于公告性质的响应，公告性质的响应无须客户端的任何请求参数，而服务器将根据业务数据自动生成。对于简单请求，因为无须发送请求参数，因而采用 POST 和 GET 方式并没有太大区别。不管发送怎样的请求，XMLHttpRequest 都应该按如下步骤进行：

（1）初始化 XMLHttpRequest 对象。

（2）打开与服务器的连接。打开连接时，指定发送请求的方法：采用 GET 或 POST；指定是否采用异步方式。

（3）设置监听 XMLHttpRequest 状态改变的事件处理函数。

（4）发送请求。如采用 POST 方法发送请求，可发送带参数的请求。

下面的应用模拟了一个简单的证券价格公告牌，下面的代码是服务器的响应页面，该页面随机产生三个数字，将这三个数字以“$”符号隔开后发送到客户端。假设这三个数字就是对应的三个股票的报价。下面是服务器页面的代码：

程序清单：codes\08\8.3\simple\second.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java"%>
<%@ page import="java.util.Random"%>
<%
//创建伪随机器，以系统时间作为伪随机器的种子
Random rand = new Random(System.currentTimeMillis());
//生成三个伪随机数字，并以$符号隔开后发送到客户端
out.println(rand.nextInt(10) + "$" + rand.nextInt(10)
    + "$" + rand.nextInt(10));
%>
```

服务器响应生成三个随机数字，三个数字以$符号隔开，因此客户端页面只需要定时向服务器发送简单请求即可，这种请求无须任何请求参数。客户端页面代码如下：

程序清单：codes\08\8.3\simple\first.html

```
<body>
mysql 的虚拟股票价格：<div id="mysql" style="color:red;font-weight:bold;"></div>
tomcat 的虚拟股票价格：<div id="tomcat" style="color:red;font-weight:bold;"></div>
jetty 的虚拟股票价格：<div id="jetty" style="color:red;font-weight:bold;"></div>
<script type="text/javascript">
//XMLHttpRequest 对象
var xmlrequest;
//创建 XMLHttpRequest 对象的初始化函数
function createXMLHttpRequest()
{
    if(window.XMLHttpRequest)
    {
        //DOM 2 浏览器
        xmlrequest = new XMLHttpRequest();
    }
    else if (window.ActiveXObject)
    {
        // IE 浏览器
        try
        {
            xmlrequest = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e)
        {
```

```
            try
            {
                xmlrequest = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e){}
        }
    }
}
//用于发送简单请求的函数
function getPrice()
{
    //初始化 XMLHttpRequest 对象
    createXMLHttpRequest();
    var uri = "second.jsp";
    //打开与服务器的连接
    xmlrequest.open("POST", uri, true);
    //指定当 XMLHttpRequest 状态改变时的事件处理函数
    xmlrequest.onreadystatechange = processResponse;
    //发送请求
    xmlrequest.send(null);
}
//当 XMLHttpRequest 状态改变时，该函数将被触发
function processResponse()
{
    if(xmlrequest.readyState == 4)
    {
        if(xmlrequest.status == 200)
        {
            //将服务器响应以$符号分割成一个字符串数组
            var prices = xmlrequest.responseText.split("$");
            //将服务器的响应通过页面显示
            document.getElementById("mysql").innerHTML=prices[0];
            document.getElementById("tomcat").innerHTML=prices[1];
            document.getElementById("jetty").innerHTML=prices[2];
            //设置 1 秒钟后再次发送请求
            setTimeout("getPrice()", 1000);
        }
    }
}
//指定页面加载完成后执行 getPrice()函数
document.body.onload = getPrice;
</script>
</body>
```

上面的应用用于发送简单请求，请求不包含任何参数。发送请求时 open()方法的第一个参数决定了发送请求的方式，例如本应用指定以 POST 方式发送请求。

open 方法通常有三个参数：第一个参数指定发送请求的服务器资源的地址，第二个参数只能为 true 或 false，确定是否采用异步方式发送请求；第三个参数是发送的方法，只能是 POST 或 GET，通常建议采用 POST 方法发送请求。

注意：

open 方法在 Internet Explorer 中可以不区分大小写，但在其他浏览器如 Firefox 中严格区分大小写，为了保证应用可以跨浏览器，建议严格区分大小写。

该应用演示了一个自动刷新的页面，大约每隔 1 秒页面的股票报价会刷新一次，页面效果如图 8.7 所示。

图 8.7　模拟的股票报价

➤➤8.3.2　发送 GET 请求

通常而言，GET 请求用于从服务器上获取数据，而 POST 请求用于向服务器发送数据。GET 请求将所有请求参数转换成一个查询字符串，并将该字符串添加到请求的 URL 之后，因而可在请求的 URL 后看到请求参数名、请求参数值。如果将某个表单的 action 属性设置为 GET，则请求会将表单中各字段的名和值转换成字符串，并附加到 URL 之后。

POST 请求则通过 HTTP POST 机制，将请求的参数以及对应的值放在 HTML Header 中传输，用户看不到明码的请求参数值。

GET 请求传送的数据量较小，一般不能大于 2KB。POST 传送的数据量较大，通常认为 POST 请求参数的大小不受限制，但往往取决于服务器的限制。通常来说，POST 请求的数据量总比 GET 请求的数据量大。

当使用 Ajax 发送异步请求时，建议使用 POST 请求，而不是 GET 请求。发送 GET 方式请求有如下两个注意点：

- 通过 open 方法打开与服务器的连接时，设置使用 GET 方法。
- 如需要发送请求参数，应将请求参数转成查询字符串，并追加到请求 URL 之后。

下面的示例应用是个级联菜单的示范，但这个级联菜单与传统级联菜单有区别，区别在于：Ajax 的级联菜单无须一次将所有的菜单信息加载到页面中，而是每次改变父菜单时页面会异步地向服务器发送请求，然后再根据服务器响应来动态加载子菜单。

注意：

虽然 GET 请求的请求参数是附加在 URL 之后的，但使用 send 方法时，还是应该为 send 方法传入参数。如果调用 send 方法时无法发送请求参数，则使用 null 作为参数即可。如果直接使用 send()方法，则在 Internet Explorer 中可以运行，而在 Firefox 中将不能正常运行。

采用 GET 请求将父菜单的 ID 作为参数发送，下面是服务器的响应页面，此处并未让服务器响应页面从数据库读取——后台数据库访问可仿照传统 Java EE 架构。服务器响应页面的代码如下：

程序清单：codes\08\8.3\GET\second.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
//从服务器获取 id 参数
int id = Integer.parseInt(request.getParameter("id"));
//根据 id 的值，确定需要返回给客户端的信息，返回客户端的城市信息以$符号隔开
switch(id)
{
    case 1:
%>
上海$广州$北京
<%
    break;
    case 2:
```

```
%>
华盛顿$纽约$加洲
<%
    break;
    case 3:
%>
东京$大阪$福冈
<%
    break;
}
%>
```

上面的 JSP 页面作为服务器响应非常简单：该页面先读取请求参数，当请求 id 为 1 时，返回三个中国城市；当请求 id 为 2 时，返回三个美国城市；当请求 id 为 3 时，返回三个日本城市。客户端的 XHTML 页面则通过 XMLHttpRequest 向服务器发送请求，并将请求动态显示在 XHTML 文档中。下面是对应的 XHTML 页面的代码：

程序清单：codes\08\8.3\GET\first.html

```
<body>
<select name="first" id="first" size="3"
    onchange="change(this.value);">
    <option value="1" selected="selected">中国</option>
    <option value="2">美国</option>
    <option value="3">日本</option>
</select>
<select id="second" size="3"></select>
<script type="text/javascript">
    //定义 XMLHttpRequest 对象
    var xmlrequest;
    //完成 XMLHttpRequest 对象的初始化
    function createXMLHttpRequest()
    {
        if(window.XMLHttpRequest)
        {
            //DOM 2 浏览器
            xmlrequest = new XMLHttpRequest();
        }
        else if (window.ActiveXObject)
        {
            // IE 浏览器
            try
            {
                xmlrequest = new ActiveXObject("Msxml2.XMLHTTP");
            }
            catch (e)
            {
                try
                {
                    xmlrequest = new ActiveXObject("Microsoft.XMLHTTP");
                }
                catch (e)
                {
                }
            }
        }
    }
    //事件处理函数，当下拉列表选择改变时，触发该函数
    function change(id)
    {
        //初始化 XMLHttpRequest 对象
```

```
        createXMLHttpRequest();
        //设置请求响应的 URL
        var uri = "second.jsp?id=" + id;
        //打开与服务器资源的连接
        xmlrequest.open("GET", uri, true);
        //设置处理响应的回调函数
        xmlrequest.onreadystatechange = processResponse;
        //发送请求
        xmlrequest.send(null);
    }
    //定义处理响应的回调函数
    function processResponse()
    {
        //响应完成且响应正常
        if (xmlrequest.readyState == 4)
        {
            if (xmlrequest.status == 200)
            {
                //将服务器响应以$符号分隔成字符串数组
                var cityList = xmlrequest.responseText.split("$");
                //获取用于显示菜单的下拉列表
                var displaySelect = document.getElementById("second");
                //将目标下拉列表清空
                displaySelect.innerHTML = null;
                //以字符串数组的每个元素创建 option，并将这些选项添加到下拉列表中
                for (var i = 0 ; i < cityList.length ; i++)
                {
                    //创建一个<option.../>元素
                    var op = document.createElement("option");
                    op.innerHTML = cityList[i];
                    //将新的选项添加到列表框的最后
                    displaySelect.appendChild(option);
                }
            }
            else
            {
                //页面不正常
                window.alert("您所请求的页面有异常。");
            }
        }
    }
</script>
</body>
```

上面的页面中第一段粗体字代码用于发送 GET 方式的 Ajax 请求，第二段粗体字代码的实质是 DOM 操作，这段 DOM 操作用于动态加载服务器响应。

在浏览器中浏览该页面，并改变第一个下拉列表框的选中项，将可看到如图 8.8 所示效果。

图 8.8　使用 GET 异步请求的级联菜单

➤➤8.3.3　发送 POST 请求

正如前面所介绍的，POST 请求的适应性更广，可使用更大的请求参数，而且 POST 请求的请求参数通常不能直接看到。因此在使用 Ajax 发送请求时，尽量采用 POST 方式发送请求，而不是 GET 方式。发送 POST 请求通常需要如下三个步骤：

（1）使用 open 方法打开连接时，指定使用 POST 方式发送请求。

（2）设置正确的请求头，POST 请求通常应设置 Content-Type 请求头。

（3）发送请求，把请求参数转为查询字符串，将该字符串作为 send()方法的参数即可。

对于上面的应用，同样可以采用 POST 方式来发送请求，采用 POST 方式发送请求只需更改一个请求的发送方法。采用 POST 方式发送请求的方法如下：

程序清单：codes\08\8.3\POST\first.html

```
//事件处理函数，当下拉列表选择改变时，触发该函数
function change(id)
{
    //初始化 XMLHttpRequest 对象
    createXMLHttpRequest();
    //确定需要发送的 URL
    var uri = "second.jsp";
    //设置以 POST 方式发送请求，并打开连接
    xmlrequest.open("POST", uri, true);
    //设置 POST 请求的请求头
    xmlrequest.setRequestHeader("Content-Type"
        , "application/x-www-form-urlencoded");
    //确定 XMLHttpRequest 对象的状态改变时的回调函数
    xmlrequest.onreadystatechange = processResponse;
    //发送请求，在发送请求时发送请求参数
    xmlrequest.send("id="+id);
}
```

其余的部分则无须改变，应用的执行效果与采用 GET 方式发送请求的效果完全一样。事实上，即使采用 POST 方式发送请求，一样可以将请求参数附加在请求的 URL 之后。也就是说，采用如下代码一样可以发送 POST 请求：

```
//事件处理函数，当下拉列表选择改变时，触发该函数
function change(id)
{
    //初始化 XMLHttpRequest 对象
    createXMLHttpRequest();
    //确定需要发送的 URL
    var uri = "second.jsp"id=; + id
    //设置以 POST 方法发送请求，并打开连接
    xmlrequest.open("POST", uri, true);
    //设置 POST 请求的请求头
    xmlrequest.setRequestHeader("Content-Type"
        , "application/x-www-form-urlencoded");
    //确定 XMLHttpRequest 对象的状态改变时的回调函数
    xmlrequest.onreadystatechange = processResponse;
    //发送请求，在发送请求时发送请求参数
    xmlrequest.send(null);
}
```

注意：

即使使用 POST 方法发送请求，一样可以将请求参数附加到 URL 之后。类似于 GET 请求的是，即使没有请求参数，在调用 send()方法时，也一定要为 send()方法传入 null 参数，因为 Firefox 等浏览器要求调用 send()方法时必须传入参数。

8.3.4 发送请求时的编码问题

对于亚洲开发者而言，不可避免地需要发送包含非西欧字符的请求参数，而这些请求参数在传输过程中的编码将直接影响服务器的处理。如果发送的请求里不包含非西欧字符，将不会有任何问题。但一旦包含了非西欧字符的请求参数，则可能出现异常。

下面的简单应用通过文本输入框获取用户输入，然后分别使用两种方式发送请求，服务器负责获取用户请求，并将请求参数在控制台中输出。首先看服务器程序：

程序清单：codes\08\8.3\encoding\show.jsp

```
<%-- 指定服务器请求地址 --%>
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
//服务器页面获取用户的 value 请求参数，并将其在控制台输出
System.out.println(request.getParameter("value"));
%>
```

上面的服务器响应页面没有生成任何响应，而只是打印了 value 请求参数的值。本应用专门用于分析客户端请求参数。客户端页面提供了两种方法来发送请求，分别通过 GET 和 POST 方式发送请求。下面是客户端的页面代码：

程序清单：codes\08\8.3\encoding\first.html

```
<body>
<input id="test" name="test" type="text" size="90" />
<br />
<input type="button" value="GET 发送"
    onclick='getSend(document.getElementById("test").value)' />
<input type="button" value="POST 发送"
    onclick='postSend(document.getElementById("test").value)' />
<script type="text/javascript">
//保存 XMLHttpRequest 对象的变量
var xmlrequest;
function createXMLHttpRequest()
{
    if(window.XMLHttpRequest)
    {
        //DOM 2 浏览器
        xmlrequest = new XMLHttpRequest();
    }
    // IE 浏览器
    else if (window.ActiveXObject)
    {
        try
        {
            xmlrequest = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e)
        {
                try
                {
                xmlrequest = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e){}
        }
    }
}
//发送 POST 请求
function postSend(value)
{
    //初始化 XMLHttpRequest 对象
    createXMLHttpRequest();
    //服务器的请求 URL
    var uri = "show.jsp";
    //打开与服务器的连接，使用 POST 方式
    xmlrequest.open("POST", uri, true);
```

```
        xmlrequest.setRequestHeader("Content-Type"
            , "application/x-www-form-urlencoded");
        //发送请求
        xmlrequest.send("value="+value);
    }
    //发送 GET 请求
    function getSend(value)
    {
        //创建 XMLHttpRequest 对象
        createXMLHttpRequest();
        //设置请求的服务器程序 URL
        var uri = "show.jsp?value="+value;
        //打开连接，使用 GET 方式
        xmlrequest.open("GET", uri, true);
        //发送请求
        xmlrequest.send(null);
    }
    </script>
    </body>
```

上面的页面提供了两个按钮，这两个按钮分别用于发送 GET 请求和 POST 请求，单击两个按钮时分别触发 postSend()方法和 getSend()方法，两个请求都将使用文本页面中的文本框的值作为请求参数。图 8.9 是发送请求的页面。

在如图 8.9 所示页面中输入“疯狂 Java”字符串，分别采用 GET 方法和 POST 方法发送请求，在 Tomcat 的控制台中可看到如图 8.10 所示的页面。

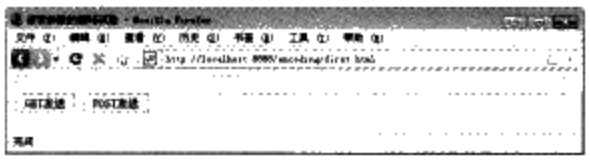


图 8.9　发送请求的页面

图 8.10　中文请求参数

从图 8.10 可以看出，当使用 GET 方式发送请求时，请求参数变成了乱码，编码问题也是 Ajax 应用必须处理的问题，对于 POST 请求很好处理，异步 POST 请求默认采用 UTF-8 字符集来编码请求参数，因此只需调用 HttpServletRequest 的 setCharacter("UTF-8")即可解决乱码问题。服务器采用如下代码即可获得正确的 POST 请求参数。

程序清单：codes\08\8.3\encoding\show.jsp

```
<%-- 指定服务器请求地址 --%>
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
//设置服务器解码所用的字符集
request.setCharacterEncoding("UTF-8");
//输出请求参数值
System.out.println(request.getParameter("value"));
%>
```

再次在如图 8.9 所示的文本框中输入中文字符，使用 POST 方法发送请求，可以看到 Tomcat 控制台中获得了正确的参数值；如果使用 GET 方法发送请求，请求参数值依然是乱码。

提　示：

在图 8.10 中可以看到，即使程序不调用 setCharacterEncoding("UTF-8");方法，Tomcat 控制台中依然可以看到正确的请求参数，这表明该服务器页面默认使用 UTF-8 字符集来解码请求参数！通常建议开发者调用 setCharacterEncoding("UTF-8");方法强制使用 UTF-8 字符集来解码请求参数。

前面已经介绍过了，GET 请求将请求参数和对应的值附加在请求的 URL 之后。对于中文的请求

参数值，将不再以中文的方式传递，而是转码成 URL 的格式。因此，将服务器页面修改成如下形式，对 GET 请求和 POST 请求分开处理：

程序清单：codes\08\8.3\encoding\show.jsp

```
<%-- 指定服务器请求地址 --%>
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
//处理 POST 请求
if (request.getMethod().equalsIgnoreCase("POST"))
{
    request.setCharacterEncoding("UTF-8");
    System.out.println(request.getParameter("value"));
}
//处理 GET 请求
else if (request.getMethod().equalsIgnoreCase("GET"))
{
    String tmp = request.getParameter("value");
    String a = new String(tmp.getBytes("ISO-8859-1") , "UTF-8");
    System.out.println(a);
}
%>
```

上面的代码先获取 value 请求参数，再将该参数按 ISO-8859-1 字符集编码成字节数组，然后按 UTF-8 字符集将该字节数组解码成字符串。再次在如图 8.9 所示页面中的文本框中输入“疯狂 Java”，单击“GET 发送”按钮，可看到控制台中出现如图 8.11 所示的输出。

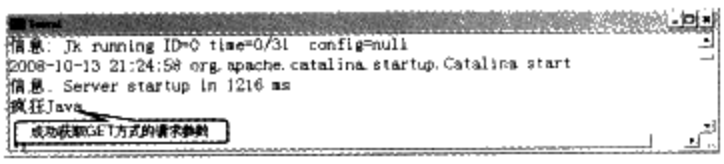


图 8.11 字符分析的 Tomcat 控制台

现在是否就高枕无忧了呢？事实上，问题依然没有得到解决。如果我们将浏览器更换成 Internet Explorer，在文本框中输入“疯狂 Java”，然后单击“GET 发送”，将看到 Tomcat 控制台中再次出现乱码。

相同的代码，使用不同浏览器将产生不同的结果，由此可见 GET 方式发送请求参数所用的字符集与客户端浏览器有关，不同浏览器在发送 GET 请求参数时使用了不同的字符集。而我们的服务器页面进行编码时始终采用 new String(tmp.getBytes("ISO-8859-1") , "UTF-8");进行解码，显然这个 UTF-8 字符集需要根据客户端浏览器发生改变。

假如我们将解码请求参数的代码改为：new String(tmp.getBytes("ISO-8859-1") , "GBK");，则在使用 Internet Explorer 发送非西欧字符请求参数时，可以在 Tomcat 控制台中看到正常字符——但使用 Firefox 发送非西欧字符请求参数的将看到乱码。

为了解决这个问题，唯一的做法是在发送请求的 XHTML 页面采用固定的 URL Encode，手动编码包含非西欧字符的请求参数，这样就可控制请求。即在客户端的 XHTML 页面采用如下 Java 代码：

```
//将包含非西欧字符的请求参数用 GBK 字符集进行编码
java.net.URLEncoder.encode(请求参数, "GBK")
```

然后再在服务器端采用如下代码：

```
//使用 GBK 字符集解码请求参数
new String(请求参数.getBytes("ISO-8859-1") , "GBK");
```

这个过程相当烦琐，而且需要 XHTML 页面中使用 Java API，相当不合时宜。因此通常建议发送 POST 请求，尽量少使用 GET 请求，理由如下：

- 处理 GET 请求的请求参数比较烦琐。

- 当请求参数包含的数据太多时，GET 请求可能丢失请求参数。
- 当两次 GET 请求的请求参数相同时，Internet Explorer 将直接使用服务器上次的缓存，根本不会重新发送请求，这对于自动刷新页面相当不利。

提 示：

对于 Internet Explorer 会“自作聪明”地缓存服务器响应的问题，开发 Ajax 应用时确实比较烦琐，不过我们也有一个办法对付：不管我们是否需要发送请求参数，发送 Ajax 请求时总是额外地增加一个参数：参数随意，参数值使用当前时间——这样就可以保证每次请求都有不同的请求参数，而 Internet Explorer 就会再次发送请求了。

➤➤8.3.5 发送 XML 请求

对于请求参数为一系列的 key-value 对的情形，笔者更加倾向于使用简单的 POST 请求。但对于某些极端的情形，请求参数特别复杂，可以考虑发送 XML 请求。XML 请求的实质还是 POST 请求，只是在发送请求的客户端页面将请求参数封装成 XML 字符串的形式，服务器端则负责解析该 XML 字符串。当然，服务器获取到该 XML 字符串后，可借助于 dom4j 或 JDOM 开源框架来解析。

下面对上面的级联菜单应用进行简单修改，修改后的级联菜单可以一次选取多个国家，如果一次选取了多个国家，则服务器返回多个国家对应的城市——请求参数采用 XML 文档发送。客户端页面需要增加一个 createXML()函数，该函数根据用户选取的国家，创建一个 XML 字符串，其代码如下：

程序清单：codes\08\8.3\xmlRequest\first.html

```
//定义创建 XML 文档的函数
function createXML()
{
    //开始创建 XML 文档，countrys 是根元素
    var  xml = "<countrys>" ;
    //获取 first 元素，并获取其所有的子节点（选项）
    var options = document.getElementById("first").childNodes;
    var option = null ;
    //遍历国家下拉列表的所有选项
    for (var i = 0 ; i < options.length; i ++)
    {
        option = options[i];
        //如果某个选项被选中
        if (option.selected)
        {
            //在 countrys 根节点下增加一个 country 子节点
            xml = xml + "<country>" + option.value + "<\/country>";
        }
    }
    //结束 XML 文档的根节点
    xml = xml + "<\/countrys>" ;
    //返回 XML 文档
    return xml;
}
```

该函数遍历 first 列表框的所有选项，对于已经选中的选项，则将其选项值添加为 XML 文档 countrys 节点的一个子节点，最后返回拼接的 XML 字符串。

注 意：

在生成 XML 字符串时，为了避免系统对斜线（/）进行特殊处理，所以使用了转义，即在斜线前增加反斜线（\）。

为了让 first 下拉列表可以支持多选，并且不是每次选中后都发送请求，将该页面的 HTML 代码进行简单修改，为 first 下拉列表增加 multiple="multiple"属性（该属性保证下拉列表可以多选），并增加一个按钮用于发送 Ajax 请求。修改后的列表框和按钮代码如下：

程序清单：codes\08\8.3\xmlRequest\first.html

```
<!-- 支持多选的列表框 -->
<select name="first" id="first"
    style="width:80px" size="3" multiple="multiple">
    <option value="1" selected="selected">中国</option>
    <option value="2">美国</option>
    <option value="3">日本</option>
</select>
<!-- 用于发送 Ajax 请求的按钮 -->
<input type="button" value="发送" onClick="send();" />
<!-- 被级联改变的列表框 -->
<select name="second" id="second" style="width:100px" size="5" />
</select>
```

单击“发送”按钮将触发 send()函数，该函数用于发送 XML 请求，代码如下：

程序清单：codes\08\8.3\xmlRequest\first.html

```
//定义发送 XML 请求的函数
function send()
{
    //定义请求发送的 URL
    var uri = "second.jsp";
    //打开与服务器的连接
    xmlrequest.open("POST", uri, true);
    //设置请求头
    xmlrequest.setRequestHeader("Content-Type"
        , "application/x-www-3form-urlencoded");
    //指定当 XMLHttpRequest 对象状态发生改变时触发 processResponse 函数
    xmlrequest.onreadystatechange = processResponse;
    //发送 XML 请求
    xmlrequest.send(createXML());
}
```

从上面的代码可以看出，发送的 XML 请求实际上依然是 POST 请求，只是请求参数不再以 param=value 的形式发送，而是直接采用 XML 字符串作为参数。这意味着服务器端不能直接获取请求参数，而是必须以流的形式获取请求参数。下面是处理 XML 请求的 JSP 页面代码：

程序清单：codes\08\8.3\xmlRequest\first.html

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%@ page import="java.io.*,org.dom4j.*,org.dom4j.io.XPPReader,java.util.*"%>
<%
//定义一个 StringBuffer 对象，用于接收请求参数
StringBuffer xmlBuffer = new StringBuffer();
String line = null;
//通过 request 对象获取输入流
BufferedReader reader = request.getReader();
//依次读取请求输入流的数据
while((line = reader.readLine()) != null )
{
    xmlBuffer.append(line);
}
//将从输入流中读取到的内容转换为字符串
String xml = xmlBuffer.toString();
//以 dom4j 开始解析 XML 字串串
```

```
Document xmlDoc = new XPPReader().read(
    new ByteArrayInputStream(xml.getBytes()));
//获得 countrys 节点的所有子节点
List countryList = xmlDoc.getRootElement().elements();
//定义服务器响应的结果
String result = "";
//遍历 countrys 节点的所有子节点
for(Iterator it = countryList.iterator(); it.hasNext();)
{
    Element country = (Element)it.next();
    //如果发送的该节点的值为 1，表明选中了中国
    if (country.getText().equals("1"))
    {
        result += "上海$广州$北京";
    }
    //如果发送的该节点的值为 2，表明选中了美国
    else if(country.getText().equals("2"))
    {
        result += "$华盛顿$纽约$加洲";
    }
    //如果发送的该节点的值为 3，表明选中了日本
    else if(country.getText().equals("3"))
    {
        result += "$东京$大阪$福冈";
    }
}
//向客户端发送响应
out.println(result);
%>
```

上面的 JSP 页面先从 HttpServletRequest 获得输入流，通过输入流取得请求字符串（格式为 XML 的字符串），然后使用 dom4j 来解析该 XML 字符串。

获取了 XML 字符串后，解析 XML 字符串的选择很多：SAX、DOM、dom4j、JDOM 和 JAXP 都可以胜任，这样就可以取得更多、更复杂的请求参数。关于如何使用 Java 程序解析 XML 文档的详细介绍请参阅疯狂 Java 体系的《疯狂 XML 讲义》。

在最极端的情形下，客户端的 XML 请求字符串可能不是直接生成，而是从已有的 XML 文档中取得，这需要借助于浏览器的 XML 解析器处理 XML 文档，然后发送解析得到 XML 内容。

8.4 处理服务器响应

完整的异步通信包括，发送请求、与服务器交互和获取服务器响应三个步骤。当服务器获得客户端请求参数后，开始处理。服务器处理结束并生成响应后，客户端必须获得服务器响应，并对服务器响应进行处理，将响应动态加载在当前页面上。

➤➤8.4.1 处理的时机

获得服务器的响应很简单，主要依赖于两个属性：responseText 和 responseXML。当然，并不是任意时刻调用 XMLHttpRequest 对象的这两个属性都可以获取服务器响应。如果服务器响应还没有结束，或者没有生成正确的响应，则调用这两个属性将不能取得正确的响应。

在 XMLHttpRequest 与服务器的通信过程中，XMLHttpRequest 的状态时刻监视着服务器的响应。XMLHttpRequest 对象有个 readyState 属性，该属性可动态监控服务器的响应，当 XMLHttpRequest 的 readyState 值为 4 时，表明服务器的响应已经完成。

服务器响应完成后，还需要判断响应是否正确，这可借助于 XMLHttpRequest 对象的 status 属性，

该属性代表服务器响应的状态码。不同的状态码及对应含义请参看 8.2.2 节。前面已经介绍过，服务器响应正确的状态码为 200。如果没有发送请求，而是直接从浏览器缓存读取响应，则状态码为 304。由此可见，当 readyState 等于 4，且 status 状态码为 200 或 304 时，客户端就可以开始处理服务器的响应了。因此在 Ajax 应用中常常可以见到如下代码片段：

```
//判断处理服务器响应的时机是否成熟
if (objXMLHttp.readyState == 4 &&
    (objXMLHttp.status == 200 || objXMLHttp.status == 304))
{
    ...
}
```

➤➤8.4.2 使用文本响应

一旦处理服务器响应的时机成熟，就可以获取服务器响应了。获取服务器响应主要借助于 XMLHttpRequest 对象的如下两个属性：

- **responseText**：生成普通文本响应。
- **responseXML**：生成 XML 响应。

第一个属性 responseText，用于生成文本响应，该属性将返回服务器响应的字符串。

使用文本响应适用于响应简单字符串的情形。在这种情形下，服务器响应是普通文本内容，基本不会包含太多格式。如果需要对文本字符串进行处理，则客户端需要自己分析该字符串，并将这段字符串解析成更复杂的表现形式。

关于使用文本响应的示例，可以参看 8.3 节的示例，上面的 JSP 示例页面都生成普通文本响应，而客户端根据$分解这些文件，然后以分解得到的字符串数组创建下拉列表。

➤➤8.4.3 使用 XML 响应

如果服务器需要生成特别复杂的响应，则可采用生成 XML 响应。生成 XML 响应需要借助于 XMLHttpRequest 的 resonseXML 属性，该属性生成一个 XML 文档对象。

几乎所有的浏览器都提供了解析 XML 文档对象的方法，借助于浏览器解析 XML 文档的能力，JavaScript 可以访问到 XML 文档的节点值，一旦访问到 XML 文档的节点值，就可以通过 DOM 动态加载到页面中显示出来。

还是上面级联菜单的示例，下面使用 XML 响应完成。下面的 JSP 页面不再生成普通文本，而是生成一个 XML 文档。代码如下：

程序清单：codes\08\8.4\xmlResponse\second.jsp

```
<%@ page contentType="text/xml; charset=GBK"%>
<%
//设置生成响应的编码方式
response.setContentType("text/xml; charset=UTF-8");
response.setHeader("Cache-Control","no-cache");
//输出 XML 文档的根元素
out.println("<citylist>");
int id = Integer.parseInt(request.getParameter("id"));
//对不同的参数，生成不同的 XML 文档
switch(id)
{
    case 1:
%>
<city>上海</city>
<city>广州</city>
<city>北京</city>
```

```
<%
    break;
    case 2:
%>
<city>华盛顿</city>
<city>纽约</city>
<city>加洲</city>
<%
    break;
    case 3:
%>
<city>东京</city>
<city>大阪</city>
<city>福冈</city>
<%
    break;
}
%>
</citylist>
```

客户端代码中的 JavaScript 部分负责解析 XML 文档，因为 XML 文档是结构化的文档，因而访问 XML 文档的节点比访问普通文本更简单。下面是客户端 JavaScript 处理 XML 响应的代码：

程序清单：codes\08\8.4\xmlResponse\ first.html

```
//处理服务器响应
function processResponse()
{
    //判断服务器响应是否完成
    if(xmlrequest.readyState == 4)
    {
        //判断服务器的响应是否成功
        if(xmlrequest.status == 200)
        {
            //获取服务器的 XML 响应
            var xmldoc = xmlrequest.responseXML
            //获取 XML 文档的 city 节点列表
            var cityList = xmldoc.getElementsByTagName("city");
            //获取用于显示下拉菜单的下拉框
            var displaySelect = document.getElementById("second");
            //清空列表框原有的选项
            displaySelect.innerHTML = null;
            //依次遍历多个 city 节点
            for (var i = 0 ; i < cityList.length ; i++)
            {
                //创建一个 option 节点
                option = document.createElement("option");
                //对于 IE 浏览器，DOM 节点内的文本内容直接使用 text 访问
                if (cityList[i].text)
                {
                    option.innerHTML = cityList[i].text;
                }
                //对于 DOM 2 浏览器，DOM 节点内的文本内容又是一个 Text 节点
                else
                {
                    option.innerHTML = cityList[i].firstChild.data;
                }
                //依次将多个 option 添加到 select 中
```

```
                displaySelect.appendChild(option);
            }
        }
    }
}
```

XMLHttpRequest 的 responseXML 属性在不同浏览器中返回的不一定是同一个对象，但总对应于浏览器内的 XML 文档对象，浏览器总提供了相应的方法来解析该 XML 文档。上面的程序中最后一段粗体字代码考虑到了不同浏览器处理 DOM 节点的差异，因此上面的程序可以实现跨浏览器，在 Firefox 中查看该页面可看到如图 8.12 所示效果。

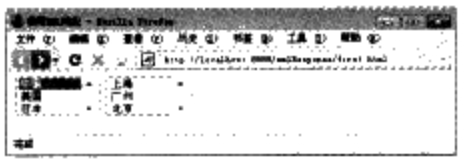


图 8.12　使用 XML 响应

注意：

使用 XML 响应要注意不同浏览器之间的差异，虽然各浏览器都实现了 DOM 规范，但它们在实现细节上依然存在一些差异。

➤➤8.4.4　使用 DOM 模型生成页面

一旦获取了服务器响应，不管是文本响应还是 XML 响应，JavaScript 都能解析响应内容，正如前面所介绍的，使用文本响应，从服务器获取的是普通字符串，这在所有浏览器中都是通用的；如果使用 XML 响应，则返回一个 XML 文档对象， XML 文档对象需要浏览器的 XML 解析器支持。

在使用文本响应和 XML 响应两个选择上，笔者更多倾向于选择文本响应，文本响应的字符串简单，无须生成额外的 XML 标签，服务器获得的是简单字符串，无须使用过多的解析手段，更容易实现跨浏览器的代码。如果使用 XML 响应，则必须借助于浏览器的 XML 解析器，这样将造成 JavaScript 脚本与浏览器的 XML 解析器 API 耦合，增大了跨浏览器的难度。

如果响应数据相当复杂，而且数据量相当大，使用 XML 文档当然有其优势：XML 文档是种结构化文档，解析 XML 文档的数据要比解析普通字符串的数据简单得多。一旦获得了 XML 格式的数据，就可以通过浏览器的 DOM 解析器来解析这些数据。

Ajax 技术的整个过程如下：XMLHttpRequest 发送请求，在与服务器交互中，其 readyState 状态可以监听到服务器的响应状态，当服务器的响应完成时，XMLHttpRequest 负责解析服务器响应，获取到响应后，解析出响应的数据，通过 DOM 操作来加载服务器响应。

8.5　XMLHttpRequest 对象的运行周期

整个 Ajax 技术紧紧围绕在 XMLHttpRequest 对象周围，了解 XMLHttpRequest 对象的运行周期，就是了解 Ajax 应用的运行。

（1）Ajax 应用总是从创建 XMLHttpRequest 对象开始，XMLHttpRequest 对象的作用如同名字所暗示的，允许通过客户端脚本来发送 HTTP 请求。Ajax 应用的第一步总是创建一个 XMLHttpRequest 实例，然后使用它来发送请求，这种请求可以是 GET 方式的，也可以是 POST 方式的。

（2）XMLHttpRequest 发送完之后，服务器的响应何时到达？应该何时处理服务器的响应呢？这需要借助于 JavaScript 的事件机制。还是回到 Ajax 的核心对象上来，Ajax 的核心对象是 XMLHttpRequest，它也是一个普通的 JavaScript 对象，就如一个普通按钮或一个普通文本框一样，可以触发事件；而 XMLHttpRequest 能触发的事件就是 onreadystatechange，当 XMLHttpRequest 对象的状态改变时，将触发其 onreadystatechange 事件。为 XMLHttpRequest 对象的 onreadystatechange 事件指定事件处理函数，该事件处理函数将可以在 XMLHttpRequest 状态改变时被触发，这个事件处理函数也称为回调函数。

（3）XMLHttpRequest 状态改变，且 readyState 为 4 时，即表明服务器的响应已经完成，此时可以开始处理服务器响应。

（4）通过 JavaScript 的事件机制，使用事件处理函数监听 XMLHttpRequest 状态的改变，当 XMLHttpRequest 的 readyState 为 4，且 status 为 200 时，事件处理函数处理服务器响应。

（5）进入事件处理函数后，XMLHttpRequest 依然不可或缺，事件处理函数必须借助于 XMLHttpRequest 来获取服务器响应，调用 responseText 方法或 responseXML 方法获取服务器的响应。至此，XMLHttpRequest 对象的运行周期结束。

（6）JavaScript 通过 DOM 操作将服务器响应动态地加载到 XHTML 页面中。

整个过程，从发送 HTTP 请求，到监控服务器的响应状态，到获取服务器响应数据，XMLHttpRequest 对象一直是 Ajax 技术的灵魂。

图 8.13 显示了 XMLHttpRequest 的运行周期。

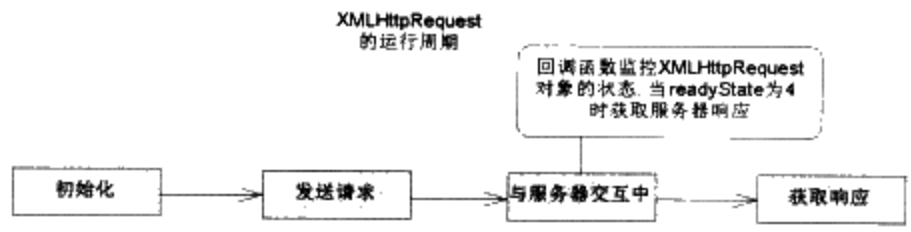


图 8.13　XMLHttpRequest 的运行周期

8.6　Ajax 必须解决的问题

虽然 Ajax 非常有用，但由于 JavaScript 本身存在跨浏览器问题，而且 XMLHttpRequest 对象在不同的浏览器中也有不同实现，因此 Ajax 技术的应用必须考虑跨浏览器的问题。此外还有 Ajax 的异步通信带来的安全性问题，以及大量 JavaScript 脚本运行时的性能问题，这些都需要 Ajax 开发者认真对待。

➤➤8.6.1　跨浏览器问题

Ajax 技术主要依赖于五种技术：XMLHttpRequest、JavaScript、DOM、XML 和 CSS。其中前三种是核心，后两种是可选的。当然，对于一个综合的 Ajax 应用，这五项技术一个都不能少。这五项技术中，除了 XML 和 CSS 两项技术大致上不存在浏览器差异外，其他三项技术都存在浏览器差异。

XMLHttpRequest 在不同的浏览器中的实现完全不同：Internet Explorer 采用 ActiveX Object 实现 XMLHttpRequest 对象，而 Firefox、Opera 等浏览器则采用不同的方式实现这个对象。即使是 Internet Explorer，不同版本也有不同的实现方式。

因此，在创建 XMLHttpRequest 对象时，必须尽量考虑到目前已经存在的各种浏览器，采用更通用的代码来创建 XMLHttpRequest 对象。下面的代码可以在不同的浏览器中创建 XMLHttpRequest 对象：

```
var objXMLHttp;
function createXMLHttpRequest()
{
    //对于 Firefox、Opera 等遵守 DOM 2 规范的浏览器
    if (window.XMLHttpRequest)
    {
        objXMLHttp = new XMLHttpRequest();
    }
    //对于 Internet Explorer 浏览器
    else
    {
        //将 Internet Explorer 的不同 XMLHTTP 实现声明成数组
        var MSXML = ['MSXML2.XMLHTTP.5.0',
```

```
            'MSXML2.XMLHTTP.4.0', 'MSXML2.XMLHTTP.3.0',
            'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
        //依次对每个 XMLHTTP 实现创建 XMLHttpRequest 对象
        for(var n = 0; n < MSXML.length; n ++)
        {
            try
            {
                objXMLHttp = new ActiveXObject(MSXML[n]);
                break;
            }
            catch(e)
            {
            }
        }
    }
}
```

即使采用上面的代码，依然不能保证整个 Ajax 应用一定可以支持所有浏览器。一旦推出新的浏览器，或者 XMLHttpRequest 有了新的实现，都将导致上面创建 XMLHttpRequest 对象的代码失效。但幸运的是，W3C 组织正在为 XMLHttpRequest 制订标准，一旦 XMLHttpRequest 形成了某种标准，事情就会简单起来。

此外，JavaScript 本身有很多不同的版本，DOM 也有几种不同的实现，所有的 JavaScript 脚本和 DOM 操作都需要得到浏览器的支持，虽然 JavaScript 和 DOM 都有行业标准，但因为其版本并不统一，因而浏览器对各脚本的支持也不尽相同。为了保证 Ajax 应用可以在不同的浏览器上运行，应该尽量避免使用特定浏览器才支持的方法和属性，以避免浏览器不支持。如果确实需要使用指定浏览器才支持的属性和方法，应先判断客户端浏览器是否支持该属性和方法。

➤➤8.6.2 安全性问题

Ajax 应用依然是一个基于 B/S 结构的应用，B/S 结构的应用总是面临着更多的安全性问题。除了传统的安全问题外，Ajax 应用还面临如下安全性问题：

- JavaScript 本身的安全性。
- 数据在网络上传输的安全。
- 客户端调用远程服务的安全。

8.6.2.1 JavaScript 本身的安全性

虽然 JavaScript 的安全性已逐步提高，例如对某些兼容性支持的放弃、受限功能、“同源”策略、安全区域以及签名脚本。

JavaScript 的跨浏览器和跨平台特性并不是真正的安全性问题。通过某些修改，在一台苹果电脑上的 Internet Explorer 中运行的脚本也可以在一台 PC 的 Netscape 上运行，因此一个恶意的脚本在 Netscape 上可能没有问题，但在 Internet Explorer、Opera 以及 Mozilla 上则可能引发安全隐患。

为了提高 JavaScript 的安全性，浏览器限制了 JavaScript 有很多事不能做：不能访问客户端文件、不能查询客户端网络连接、不能执行操作系统命令或程序。虽然有些时候缺少这些功能让程序开发非常困难，但提高了客户端 JavaScript 的安全。图 8.14 是当 JavaScript 试图访问客户端文件系统时 Internet Explorer 弹出的警告。

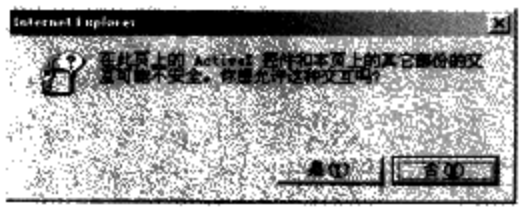


图 8.14 JavaScript 访问本地文件系统的警告

此外，JavaScript 还有许多受限功能：包括访问浏览器的历史记录、上载文件、提交、发送邮件、改变菜单栏等。

通常建议 JavaScript 只能读取和修改同源文档的属性，这种策略称为同源策略（Same Origin Policy，简称 SOP）。它也涵盖端口和协议，因此如果一个 JavaScript 脚本的源端口是 80 并且协议是 HTTP，那么就不能操作源端口为 21 及 FTP 协议的文档。通过这种限制，可以避免信息泄漏。假如有某个不怀好意的 Cracker，他可以通过运行恶意脚本来查看其他浏览器窗口中的有用信息，然后使用 XMLHttpRequest 请求或者其他方法将这些信息发送到自己的 Web 站点（也许他从其他浏览器窗口里获取的是信用卡的密码），那将是相当危险的事情。因此应该尽量限制使用 JavaScript 脚本访问其他文档的数据，这也是被大多数浏览器禁止的。

同源策略不仅应用于文档，而且应用于浏览器 Cookie 集。这可防止 Cracker 复制购物者浏览器的 Cookie，从而使网上商店认为他就是购物者本人。

大多数的策略都有例外，对于同源策略也是如此。这个策略对于拥有 UniversalBrowserRead 权限的脚本不起作用。拥有这个权限的脚本可以读取非同源的文档，拥有 UniversalBrowserWrite 权限则可以修改非同源脚本。使用这两个权限或者使用 UniversalBrowserAccess 权限就可以获得读写非同源文档的能力。

Internet Explorer 和 Netscape Navigator 在安全性上有各自不同的处理方法。Internet Explorer 通过区域来处理。Internet Explorer 5.5 将区域分为 Internet 和本地 Intranet，以及信任站点和受限站点。这四个等级使用户或管理员能够根据区域设置 JavaScript 所能进行的操作。可以通过指定每个站点的安全性来获得更好的控制。

Netscape Navigator 则使用数字签名脚本来管理安全性，用户可以针对 JavaScript 的每项功能对脚本实施限制。用签名脚本实现安全性看起来非常有效，但它并不能保证脚本不是恶意脚本。一个签名脚本只是确立了身份，而且也不能完全保证身份的正确。

有一项 JavaScript 安全性功能对所有浏览器都是相同的。JavaScript 在浏览器中运行于一个“沙盒”之中，就像 Java Applet 一样。本质上，JavaScript 在客户端浏览器中的功能是受限的。

8.6.2.2 数据在网络上传输的安全

当请求参数在网络上传输时，所有的代码都是以明码的方式传输，如果请求的只是一些不太重要的数据，采用普通的 HTTP 请求即可满足要求，但如果这些数据涉及特别机密的信息，例如信用卡的账户、口令等信息，则需要对这些数据进行加密。

一个正常的路由器不应该修改传输的数据包，除了数据包的包头和路由信息之外，路由器不会查看任何信息，但一个恶意的路由器则可能会读取传输的内容，从而获取用户的信用卡账户、口令等信息，甚至可能修改路由信息，将用户的请求重定向到一个恶意站点等。通常，普通的 HTTP 请求只适合于普通的页面浏览，对于机密的数据，则不应该使用 HTTP 请求完成。

与 HTTP 请求对应的是安全连接 HTTPS。这种传输协议是建立在安全 Socket 通信上的 HTTP 协议，它在明码的 HTTP 上增加了一层包装，使用公匙—密匙对加密传输的数据。HTTPS 协议由 Netscape 开发并内置于其浏览器中，用于对数据进行压缩和解压操作，并返回网络上送回的结果。HTTPS 实际上应用了 Netscape 的安全套接字层（SSL）作为 HTTP 应用层的子层。（HTTPS 使用端口 443，而不是像 HTTP 那样使用端口 80 来进行通信。）SSL 使用 40 位关键字作为 RC4 流加密算法，这对于商业信息的加密是合适的。HTTPS 和 SSL 支持使用 X.509 数字认证，如果有必要，用户可以确认发送者是谁。

当使用 HTTPS 传输数据时，恶意路由一样可以看到传输的数据内容，但因为数据已经进行了加密，因此即使被看到危险也不是很大。

HTTPS 是安全的超文本传送协议，就是使用 SSL 加密后的超文本传送协议，浏览器都可以支持这种协议下的网络文档，前提是具备对方提供的安全证书。

但使用 HTTPS 协议不得不考虑计算开销，HTTPS 需要使用 SSL 对数据加密和解密。这种计算上的开销，在客户端不会有大问题，因为客户端的闲置资源总是比较多。但对于服务器却不得不考虑，特别是对于一个高并发的大型应用。通常的做法是：仅使用 HTTPS 协议传输关键资源，普通资源则

采用普通的 HTTP 协议传输。

虽然如此，HTTPS 依然是网络上传输敏感数据的推荐解决方案，尽管使用 HTTPS 可能会使服务器的性能有所降低，但对于敏感的数据而言，以性能换取更高的安全性还是值得的。

8.6.2.3 客户端调用远程服务的安全

在传统的 Java EE 应用中，所有的请求都发送到控制器，而控制器负责权限检查控制，没有权限的请求将被拒绝。

对于 Ajax 应用而言，Ajax 的请求到底发送给谁？是否可以将业务逻辑层对象直接暴露给 Ajax 引擎调用呢？在前面的介绍中我们知道，Ajax 技术允许客户端完成部分服务器的工作，那么是否可以采用 JavaScript 来检查用户权限？如果用户拥有足够的权限，就允许他访问业务逻辑组件的服务，如果用户没有足够的权限，客户端脚本拒绝就拒绝他访问业务逻辑组件的服务。

使用客户端脚本控制权限不是一个好思路。Cracker 可以轻松绕过 JavaScript 的权限检查，然后直接调用业务逻辑组件的方法。如果非要让 Ajax 引擎访问业务逻辑组件，则建议将权限检查推后到业务逻辑组件中进行。

事实上，笔者依然坚持将 Ajax 的应用局限在 Java EE 应用的表现层，而不要扩散到其他层中。所有请求只能向应用的控制器请求，而不是直接访问业务逻辑组件。采用这种严格的分层，不仅更符合传统的 Java EE 应用的架构，也更利于安全的控制。

所有的 Ajax 请求都向控制器请求，控制器负责系统的安全。控制器负责检查调用者是否有访问资源的权限，而所有的业务逻辑组件隐藏在控制器的后面，这种策略能提供更好的安全性和解耦。

➤➤8.6.3 性能问题

这里所说的性能，主要是客户端 JavaScript 的运行性能，服务器端的响应性能在不在本节讨论范围之内。虽然很多人认为 JavaScript 主要在客户端运行，因此无须花太多时间关注 JavaScript 的运行性能。但作为一个负责任的程序员，还是应该尽量为浏览者节省资源。下面是关于客户端 JavaScript 性能的一些优化的小技巧：

- 关于 JavaScript 的循环，循环是一种常用的流程控制。JavaScript 提供了三种循环：for(; ;)、while( )、for(in)。在这三种循环中 for(in)的效率最差，因为它需要查询 Hash 键，因此应尽量少用 for(in)循环，for(; ;)和 while()循环的性能基本持平。当然，推荐使用 for 循环，如果循环变量递增或递减，不要单独对循环变量赋值，而应该使用嵌套的++或--运算符。
- 如果需要遍历数组，应该先缓存数组长度，将数组长度放入局部变量中，避免多次查询数组长度。
- 局部变量的访问速度要比全局变量的访问速度更快，因为全局变量其实是 window 对象的成员，而局部变量是放在函数的栈里的。
- 尽量少使用 eval，每次使用 eval 需要消耗大量时间。这时候使用 JavaScript 所支持的闭包可以实现函数模板。
- 尽量避免对象的嵌套查询，对于 obj1.obj2.obj3.obj4 这个语句，需要进行至少 3 次查询操作，先检查 obj1 中是否包含 obj2，再检查 obj2 中是否包含 obj3，然后检查 obj3 中是否包含 obj4……这不是一个好策略。应该尽量利用局部变量，将 obj4 以局部变量保存，从而避免嵌套查询。
- 使用运算符时，尽量使用+=、-=、*=、\=等运算符号，而不要直接进行赋值运算。
- 当需要将数字转换成字符时，采用如下方式："" + 1 。从性能上来看，将数字转换成字符时，有如下公式：("" +) > String() > .toString() > new String()。String()属于内部函数，所以速度很快，而.toString()要查询原型中的函数，所以速度逊色一些，new String()需要重新创建一个字符串对象，速度最慢。

- 当需要将浮点数转换成整型时，应该使用 Math.floor()或者 Math.round()。而不是使用 parseInt()，该方法用于将字符串转换成数字。而且 Math 是内部对象，所以 Math.floor()其实并没有多少查询方法和调用时间，速度是最快的。
- 尽量使用 JSON 格式来创建对象，而不是 var obj = new Object 方法。因为前者是直接复制，而后者需要调用构造器，因而前者的性能更好。
- 当需要使用数组时，也尽量使用 JSON 格式的语法，即直接使用如下语法定义数组：[aram,param,param,...]，而不是采用 new Array(param,param,...)这种语法。因为使用 JSON 格式的语法是引擎直接解释的，而后者则需要调用 Array 的构造器。
- 对字符串进行循环操作，例如替换、查找，应使用正则表达式。因为 JavaScript 的循环速度比较慢，而正则表达式的操作是用 C 写成的 API，性能比较好。

最后有一个基本的原则，对于大的 JavaScript 对象，因为创建时时间和空间的开销都比较大，因此应该尽量考虑采用缓存。例如 XMLHttpRequest，这个对象对于 Ajax 而言是个相当重要的对象，而且重用率极高，考虑使用池的技术管理。下面是使用池的技术管理 XMLHttpRequest 的示例代码：

程序清单：codes\08\8.6\xmlrequestPool.js

```
//使用 literal 语法定义一个对象 XMLHttp
var XMLHttp =
{
    //定义第一个属性，该属性用于缓存 XMLHttpRequest 对象的数组
    XMLHttpRequestPool: [],
    //对象的第一个方法，该方法用于返回一个 XMLHttpRequest 对象
    getInstance:function()
    {
        //从 XMLHttpRequest 对象池中取出一个空闲的 XMLHttpRequest
        for (var i = 0; i < this.XMLHttpRequestPool.length; i ++)
        {
            //如果 XMLHttpReuqest 的 readyState 为 0，或者为 4，
            //都表示当前的 XMLHttpRequest 对象为闲置的对象
            if (this.XMLHttpRequestPool[i].readyState == 0 ||
                this.XMLHttpRequestPool[i].readyState == 4)
            {
                return this.XMLHttpRequestPool[i];
            }
        }
        //如果没有空闲的，将再次创建一个新的 XMLHttpRequest 对象
        this.XMLHttpRequestPool[this.XMLHttpRequestPool.length]
            = this.createXMLHttpRequest();
        //返回刚刚创建的 XMLHttpRequest 对象
        return this.XMLHttpRequestPool[this.XMLHttpRequestPool.length - 1];
    },
    //创建新的 XMLHttpRequest 对象
    createXMLHttpRequest:function()
    {
        //对于 DOM 2 规范的浏览器
        if (window.XMLHttpRequest)
        {
            var objXMLHttp = new XMLHttpRequest();
        }
        //对于 Internet Explorer
        else
        {
            //将 Internet Explorer 内置的所有 XMLHTTP ActiveX 控件设置成数组
            var MSXML = ['MSXML2.XMLHTTP.5.0', 'MSXML2.XMLHTTP.4.0',
                'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
```

```
        //依次对 Internet Explorer 内置的 XMLHTTP 控件初始化，尝试创建 XMLHttpRequest 对象
        for(var n = 0; n < MSXML.length; n ++)
        {
            try
            {
                //如果可以正常创建 XMLHttpRequest 对象，使用 break 跳出循环
                var objXMLHttp = new ActiveXObject(MSXML[n]);
                break;
            }
            catch(e)
            {
            }
        }
    }
    //Mozilla 的某些版本没有 readyState 属性
    if (objXMLHttp.readyState == null)
    {
        //直接设置其 readyState 为 0
        objXMLHttp.readyState = 0;
        //对于那些没有 readyState 属性的浏览器，将 load 动作与下面的函数关联起来
        objXMLHttp.addEventListener("load", function ()
        {
            //当从服务器加载数据完成后，将 readyState 状态设为 4
            objXMLHttp.readyState = 4;
            if (typeof objXMLHttp.onreadystatechange == "function")
            {
                objXMLHttp.onreadystatechange();
            }
        }, false);
    }
    return objXMLHttp;
},
//定义对象的第三个方法：发送请求(方法[POST,GET]，地址，数据，回调函数)
sendRequest: function (method, url, data, callback)
{
    var objXMLHttp = this.getInstance();
    with(objXMLHttp)
    {
        try
        {
            //增加一个额外的 randnum 请求参数，用于防止 IE 缓存服务器响应
            if (url.indexOf("?") > 0)
            {
                url += "&randnum=" + Math.random();
            }
            else
            {
                url += "?randnum=" + Math.random();
            }
            //打开与服务器的连接
            open(method, url, true);
            //对于使用 POST 请求方式
            if (method == "POST")
            {
                // 设定请求头
                setRequestHeader('Content-Type',
                  'application/x-www-form-urlencoded');
                send(data);
```

```
            }
            //对于采用 GET 请求
            if (method == "GET")
            {
                send(null);
            }
            //设置状态改变的回调函数
            onreadystatechange = function ()
            {
                //当服务器的响应完成时，以及获得了正常的服务器响应
                if (objXMLHttp.readyState == 4 &&
                    (objXMLHttp.status == 200 ||
                        objXMLHttp.status == 304))
                {
                    //当响应时机成熟时，调用回调函数处理响应
                    callback(objXMLHttp);
                }
            }
        }
        catch(e)
        {
            alert(e);
        }
    }
};
```

上面的程序中粗体字代码使用了一个数组来缓存已有的 XMLHttpRequest 对象，该数组就是一个 XMLHttpRequest 对象池，每次需要发送请求时，将从 XMLHttpRequest 对象池中取出一个闲置的 XMLHttpRequest 对象，如果当前的对象池中没有闲置的 XMLHttpRequest 对象，则创建一个新的 XMLHttpRequest 对象。

每次使用该对象，只需要将上面的 JavaScript 代码包含在需要使用的页面中，然后直接通过如下方式发送请求：

```
//直接发送请求
XMLHttp.sendRequest("POST", url, data, callback);
```

其中 callback 是用于处理响应的回调函数，URL 是所请求的服务器 URL，而 data 是需要发送的请求参数数据，POST 是请求参数的发送方法。

8.7 本章小结

本章主要介绍了 Ajax 核心技术的 XMLHttpRequest 对象，介绍了 XMLHttpRequest 的创建，包括在不同浏览器中创建 XMLHttpRequest 对象的代码。本章详细介绍了 XMLHttpRequest 对象的常用属性，包括 onreadystatechange、readyState、status、statusText、responseText 和 responseXML，掌握这些属性是使用 XMLHttpRequest 对象的基础；本章也详细介绍了该对象的各种方法，这些方法也是 Ajax 编程的基础。

本章重点介绍了如何使用 XMLHttpRequest 发送请求，包括如何发送不带参数的简单请求，发送 GET 请求、POST 请求和 XML 请求；请求发送出去之后，程序还需要监控 XMLHttpRequest 对象来决定处理响应的时机；当服务器响应到来时，使用 XMLHttpRequest 对象的 responseText 或 responseXML 可获取服务器响应，再使用 DOM 操作将服务器响应动态加载在当前页面上。

CHAPTER 9

第9章 Prototype 库详解

本章要点

- Prototype 库的背景和相关知识
- 下载和安装 Prototype 库
- 使用 Prototype 对象了解 Prototype 库基本信息
- 使用 Prototype 库中的工具函数
- 掌握 Prototype 库提供的 JSON 支持
- 使用 Prototype 提供的自定义类或对象
- 使用 Prototype 为 JavaScript 系统类提供的扩展
- 掌握 Prototype 提供的 Ajax 支持
- 使用 Ajax.Request 类
- 使用 Form.request 方法
- 使用 Ajax.Responders 对象
- 使用 Ajax.Updater 类
- 使用 Ajax.PeriodicalUpdater 类

Prototype 库是属于 Ruby On Rails 一个 JavaScript 函数库，但因为它只是一个纯粹的 JavaScript 代码库，因此完全在其他的 Web 应用中使用。不管 Web 应用服务器端采用哪种编程语言：ASP、PHP、JSP、Ruby、Perl 等，客户端视图页面都可使用 Prototype 库。

Prototype 库只是一个 JavaScript 函数库，它提供了大量工具方法来简化 DOM 操作，还扩展了 Array、String 等内建类，并新增了一些类、对象。借助于 Prototype 函数库的帮助，开发者可以少写许多 JavaScript 代码。Prototype 库对 Ajax 也提供了良好的支持，Prototype 库封装了 XMLHttpRequest 对象，开发者无需完成创建 XMLHttpRequest 对象、发送异步请求等步骤，Ajax 开发者只需提供一个操作服务器响应的回调函数，剩下的事情都交给 Prototype 库搞定。

Prototype 库是个构思巧妙、兼顾标准，而且具有跨浏览器功能的 JavaScript 函数库。借助于 Prototype 库的帮助，开发者能轻松建立有高度互动的 Web2.0 特性的富客户端页面。即使从学习的角度来看，深入研究 Prototype 库的源代码，对于掌握 JavaScript 也有巨大的帮助。

Prototype 提供了简化 JavaScript 编程，但并未提供更多于界面相关的功能，Prototype 另一个关联项目：script.aculo.us 则提供了许多界面相关的函数和功能，读者可以登录 http://script.aculo.us/站点获取关于该项目更详细的信息。

9.1 Prototype 的下载和安装

Prototype 库只是一个 JavaScript 库，下载和安装 Prototype 库都是相当容易的事情，而且可以在任何 Web 应用中使用这个类库，不管使用 ASP、PHP、JSP，甚至是静态的 HTML 页面，都可使用 Prototype 库。

➤➤9.1.1 什么是 Prototype 库

Prototype.js 是由 Sam Stephenson 写的一个 JavaScript 函数库。这个类库最初是为了 Ruby On Rails 写成，但因为它只是一个 JavaScript 函数库，因而与任何的网页脚本没有任何关系，即使对于静态网页的编辑者，使用 Prototype.js 类库也可大量减少 JavaScript 代码量。因此得到迅速推广、应用。

Prototype.js 的使用相当方便，只要在网页中引入该 JavaScript 函数库即可。因此，有人戏称它为真正即插即用的软件。Prototype.js 的代码相当精简，代码加起来总共才 4000 多行，但代码实现的功能相当强大，它的主要功能可分为三个部分：

- 简单、易用的工具函数。
- 扩展了一些原有类，增加了一些自定义的对象。
- 简化 Ajax 开发的相关类。

方便的工具函数简化了原有 HTML 元素的访问，将开发者从重复的打字中解放出来，可以更简单地访问、操作 HTML 元素。

Ajax 扩展极好地丰富了系统类的功能，为 JavaScript 原有的对象、类增加了额外的方法，这些方法对于简化 Ajax 应用的开发也是非常有用的。

Ajax 的相关类更是大大简化了 Ajax 应用的开发，无需开发者创建 XMLHttpRequest 对象，打开与服务器连接等通用、繁琐的步骤。用户只需要指定请求发送的 URL，并为应用指定回调函数，剩下的事情由 Prototype.js 库完成。如果使用 Ajax.Updater 工具类，甚至可以无需指定回调函数。

➤➤9.1.2 下载 Prototype 库

Prototype 库是 Ruby On Rails 的一个相关项目，如果需要浏览该项目的完整情况，例如下载该项目的源代码等。可登陆 http://dev.rubyonrails.org/browser/spinoffs/prototype，在该站点可下载开发中的 Prototype.js 源代码。

对于普通的 Web 应用开发者而言，使用 Prototype 库的发布版是个不错的选择，Prototype 库的最

新发布版是1.6.0.3。登陆http://www.prototypejs.org/download站点，可看到一个Download the latest stable version-1.6.0.3 （September 29, 2008）的链接，不要单击该链接，用“目标另存为”的方式下载，下载得到一个prototype-1.6.0.3.js文件，这就是我们将要使用的Prototype库。

除此之外，在浏览器地址栏输入http://www.prototypejs.org/api，该页面是Prototype库的在线API文档，学习Prototype库就是掌握这些工具函数、对象的用法。在该页面也可下载Prototype库离线API文档的CHM文档或PDF文档。

如果有的读者为了体验Prototype.js开发中的版本，则应该登陆上面介绍的完整项目所在的位置，可以下载Prototype.js开发中的版本。

➤➤9.1.3 安装Prototype库

Prototype库是一个“即插即用”的JavaScript函数库，既不需要增加额外的环境变量，也不需要增加什么配置文件，只需要在HTML页面中导入Prototype函数库即可。

导入Prototype函数库，与我们开发HTML页面时导入自己的JavaScript函数库没有任何区别。为了在自己的JavaScript脚本中使用Prototype库的功能，应该在自己的JavaScript脚本之前导入Prototype.js类库。

一旦导入了Prototype库，开发者就可以在自己的脚本中使用Prototype库提供的功能。为了导入Prototype.js类库，应在HTML页面的开始位置增加如下代码：

```
<!-- 引入一个 JavaScript 函数库 -->
<script type="text/javascript" src="prototype-1.6.0.3.js">
</script>
```

上面导入Prototype库的代码可能会有小小的变化，如果prototype-1.6.0.3.js文件名被改名了，或者它与HTML页面并不是放在同一个路径下，则应该在上面代码的基础上作对应的修改，只要让src属性指向prototype-1.6.0.3.js脚本文件所在的位置即可。

➤➤9.1.4 使用Prototype对象

Prototype库提供了一个Prototype对象，该对象提供了如下几个属性

- Version：该属性返回Prototype库的版本号。
- Browser：用于判断用户的浏览器。该属性包含如下几个属性用于判断浏览器：
  - IE：如果用户浏览器为Internet Explorer，该属性返回true。
  - Opera：如果用户浏览器为Opera，该属性返回true。
  - WebKit：如果用户浏览器为Safari、Konqueror或Chrome等浏览器，该属性返回true。
  - Gecko：如果用户浏览器为Netscape家族的Mozilla、FireFox等，该属性返回true。
  - MobileSafari：如果用户浏览器为移动版Safari，该属性返回true。
- BrowserFeatures：用于判断用户浏览器是否支持指定特性。

还提供了如下两个方法：

- emptyFunction：该方法是一个空函数。该方法对应的函数为：function() { }。
- K：该方法是一个返回未做任何改变的参数本身。K方法对应的函数为：function(x) { return x }。

下面页面代码简单示范了Prototype对象的用法：

程序清单：codes\09\9.1\PrototypeTest.html

```
<body>
    <script src="../prototype-1.6.0.3.js" type="text/javascript">
    </script>
    <script type="text/javascript">
        document.writeln("Prototype 库的版本为：" + Prototype.Version + "<br />");
```

```
        document.writeln("客户端浏览器是否为Firefox:" + Prototype.Browser.Gecko + "<br />");
        //K 方法返回参数本身
        alert(Prototype.K("测试字符串"));
    </script>
</body>
```

在浏览器中浏览该页面，可以看到 Prototype 库的版本，如果使用 Firefox 浏览器浏览该页面，看到如图 9.1 所示界面。

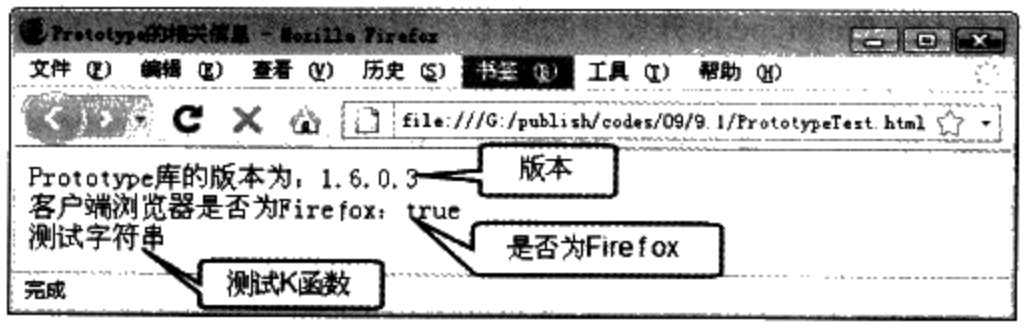


图 9.1　使用 Prototype 对象

9.2　Prototype 的工具函数

Prototype 库包含了很多有用的工具函数，这些函数对于简化页面 JavaScript 脚本，提高应用的开发速度非常有帮助，下面具体介绍这基本函数。

➤➤9.2.1　使用$()函数

前面的 Ajax 基础部分，为了访问一个 HTML 元素，我们需要使用 document.getElementById()方法，这个方法名如此之长，甚至可能引起拼写错误。Prototype 库提供了一个简化的访问方式，就是利用$()函数，该函数的语法格式如下：

- $(String tagName)：直接获取 id 为 tagName 的 HTML 元素。
- $(String tagName1, String tagName2)：获取 id 为 tagName1、tagName2 的 HTML 元素数组。

$()函数的功能比 document 对象的 getElementById()方法功能更加强大，它除了可接受一个 HTML 元素的 ID 之外，还可以接受多个 ID。如果为该函数传入多个 HTML 元素的 ID，该函数将返回多个 ID 所对应的 HTML 元素的数组。当然，大部分时候，我们使用$()函数，仅仅作为 document 的 getElementById()方法的替代品。看如下简单的代码：

程序清单：codes\09\9.2\$()Test1.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function clickHandler()
{
    //修改 out 对象的 innerHTML 属性
    $("out").innerHTML = "使用 Prototype.js"
}
</script>
<input onclick="clickHandler();" type="button" value="按钮"/>
<div id="out"></div>
</body>
```

上面的页面中，使用了$("out")，该方法可获取一个 ID 为"out"的 HTML 元素，在上面的页面代码中该对象为<div.../>元素，然后修改该<div.../>节点的 innerHTML 属性值为一个字符串常量。该页面执行的效果如图 9.2 所示。

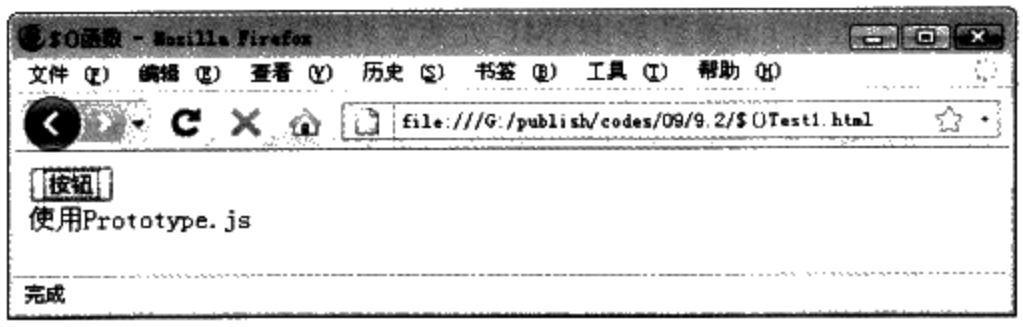


图 9.2　使用$()函数获取页面元素

因为$()函数底层就是采用 document.getElementById()方法实现，因此该函数与 getElementById()方法一样，有两点需要注意：

- 如果一个页面中有两个 ID 相同的 HTML 元素呢？$()函数只返回页面中第一个 HTML 元素。
- 对于 Internet Explorer 6.0 浏览器，$()函数不仅可以根据 ID 获取 HTML 元素，也可以根据 name 获取 HTML 元素。假设如下情况：页面中有一个 ID 为“a”的 HTML 元素，还有一个 name 为“a”的 HTML 元素，如果 HTML 页面中先出现的是 name 为“a”元素，在 Internet Explorer 6.0 中$(“a”)函数将返回 name 为“a”的 HTML 元素，而不是 ID 为“a”的 HTML 元素。

对于下面的代码：

程序清单：codes\09\9.2\$()Test2.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function clickHandler()
{
    //访问 a2 元素
    $("out").innerHTML += $("a2").value;
}
</script>
<input id="a1" name="a2" type="text" value="第一个文本框" /><br />
<input id="a2" name="a1" type="text" value="第二个文本框" /><br />
<input onclick="clickHandler();" type="button" value="显示" />
<div id="out" style="font-weight:bold;"></div>
</body>
```

上面页面的代码很简单，在页面中包含了两个文本框，第一个文本框的 name 属性为 a2，而 id 属性为 a1；第二个文本框的 name 属性为 a1，而 id 属性为 a2。

当我们在 Internet Explorer 6.0 中浏览该页面，单击“显示”按钮后将看到如图 9.3 所示的界面。

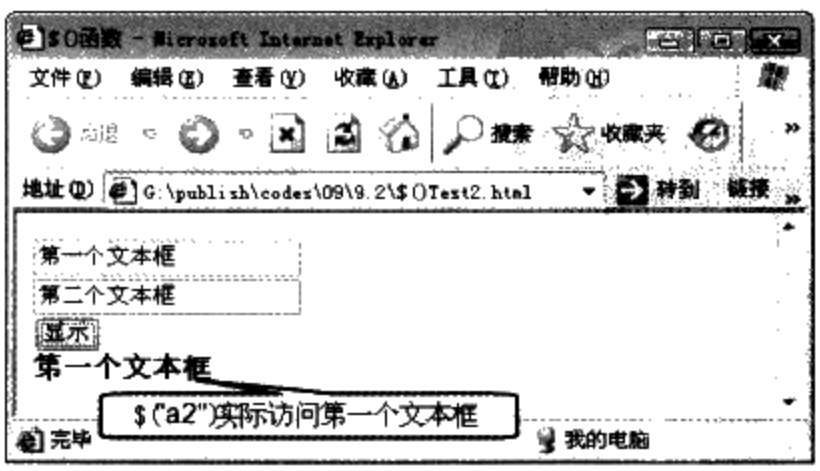


图 9.3　Internet Explorer 中使用$()函数

注意到图 9.3 中黑体部分，使用$()函数时，并没有如我们期望的获取第二个 id 为 a2 的元素，而是获取了第一个 name 为 a2 的元素。可见在 Internet Explorer6.0 中测试使用$()函数时，它不仅可以根

据 id 获取 HTML 元素，也可根据 name 获取 HTML 元素。对相同的页面，换成 Firefox 浏览器来浏览该页面，将看到如图 9.4 所示的的界面。

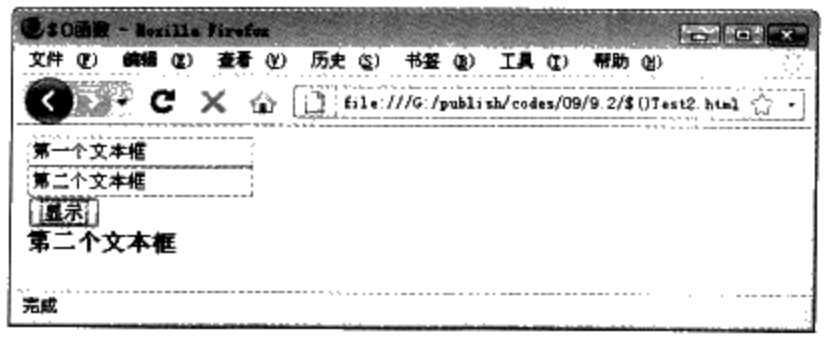


图 9.4　在 Firefox 中使用$()函数

对比图 9.3 和图 9.4 中黑体字部分，同样的代码，使用不同的浏览器查看时候将出现不同结果。导致这个问题的原因是：Internet Explorer 6.0 中的 getElementById()方法就可以根据 HTML 元素的 name 属性来获取，而 Prototype 库的$()函数借助 getElementById 方法实现。

注意：

为了避免在 DHTML 页面中访问 HTML 元素时引起错误，通常建议不要让多个 HTML 元素有相同的 id 属性，而且尽量让 HTML 元素的 id 和 name 属性保持一致。

如果为$()函数同时指定多个参数，则返回多个 HTML 元素的数组，因为 Prototype 扩展了系统原有的数组，因此访问该 HTML 元素数组也是非常方便的。看如下代码：

程序清单：codes\09\9.2\$()Test3.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function clickHandler()
{
    $("out").innerHTML += $("name1" , "name2");
}
</script>
<input id="name1" name="name1" type="text" value="第一个文本框"/><br />
<input id="name2" name="name2" type="text" value="第二个文本框"/><br />
<input onclick="clickHandler()" type="button" value="按钮"/>
<div id="out"></div>
</body>
```

上面的代码中使用了$("name1" , "name2")，这个函数将返回页面中 id 属性为 name1 和 name2 的两个 HTML 元素组成的数组。

除此之外，$()函数不仅可使用 HTML 元素的 id 属性值作为参数，还使用元素本身作为参数，这一点极大地提高了该函数的灵活性。

➤➤9.2.2　使用$$()函数

$$()函数和$()函数的功能相同，都可用于访问文档中的 HTML 元素，只是该元素的参数是一个或多个合法的 CSS 选择器，该函数则返回这些选择器所对应 HTML 元素所组成的数组。

看如下页面代码：

程序清单：codes\09\9.2\$$()Test.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
```

```
</script>
<span id="a">Struts2 权威指南</span>
<span>疯狂 Java 讲义</span>
<span class="pt9">轻量级 Java EE 企业应用实战</span>
<div class="pt9">疯狂 Ajax 讲义</div>
<hr />
<script type="text/javascript">
    //返回页面中所有<span.../>元素
    var eleList = $$("span");
    for (var i = 0; i < eleList.length; i++ )
    {
        document.writeln('$$("span")的' + i + '个元素为:'
            + eleList[i].innerHTML + "<br />");
    }
    //返回页面中 ID 为 a 的元素
    var eleList = $$("#a");
    for (var i = 0; i < eleList.length; i++ )
    {
        document.writeln('$$("#a")的' + i + '个元素为:'
            + eleList[i].innerHTML + "<br />");
    }
    //返回页面中 class 为 pt9 的元素
    var eleList = $$(".pt9");
    for (var i = 0; i < eleList.length; i++ )
    {
        document.writeln('$$(".pt9")的' + i + '个元素为:'
            + eleList[i].innerHTML + "<br />");
    }
</script>
</body>
```

上面页面中三次使用$$()函数来获取页面的 HTML 元素，$$("span")将获取页面中所有<span.../>元素，$$("#a")将获取页面中所有 id 为 a 的元素，$$(".pt9")将获取页面中所有 class 属性为 pt9 的元素。运行上面页面代码将看到如图 9.5 所示效果。

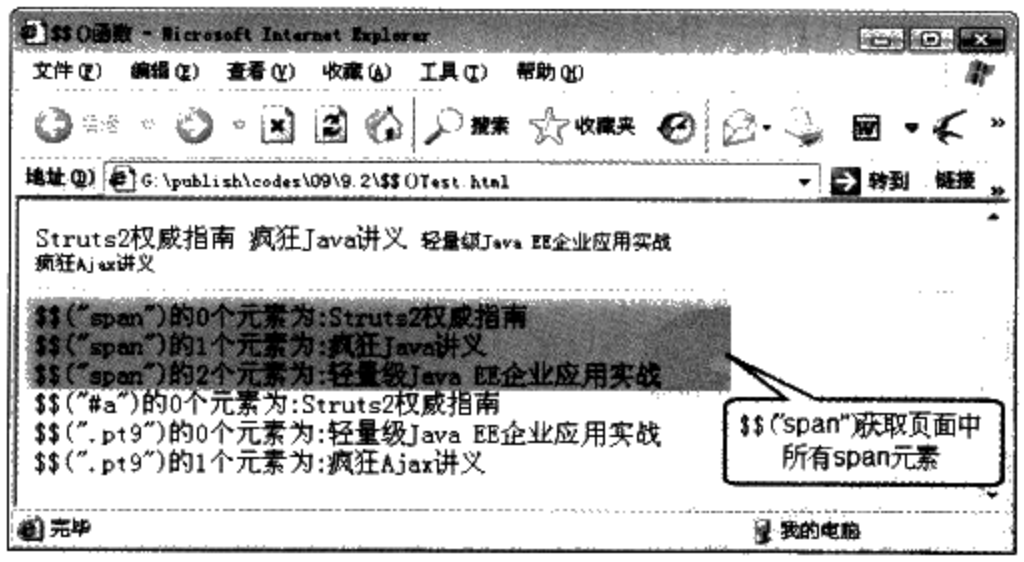


图 9.5 使用$$()函数

除了支持上面常用的选择器之外，该函数也支持如下更复杂的选择器：

```
//返回所有位于 id 为 java 元素之内、且具有 rel 属性的<a.../>元素。
$$('#java a[rel]');
//返回所有 href 属性为 http://www.crazyjava.org 的<a.../>元素。
$$('a[href="http://www.crazyjava.org"]');
//返回位于 id 为 crazyjava、leegang 的元素之内的所有<a.../>元素。
$$('#crazyjava a', '#leegang a');
```

➤➤9.2.3 使用$A()函数

$A()函数能把单个的集合对象转换成一个 Array 对象。结合被 Prototype.js 扩展后的 Array 类，能方便的把任何的可枚举列表转换成或拷贝到一个 Array 对象。比较常用的用途就是用于遍历 HTML 的节点列表。看如下代码：

程序清单：codes\09\9.2\$A()Test1.html

```
<body>
<div id="menuBar">
    <div>文件</div>
    <div>编辑</div>
    <div>查看</div>
</div>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//获得 id 为 menuBar 的元素，再获取该元素的所有 div 子元素，返回一个 HTMLCollection
var fileList = $("menuBar").getElementsByTagName("div");
document.writeln(fileList + "<br />");
//将 fileList 转换为一个数组
var fileArray = $A(fileList);
//依次输出数组的每个元素
for (var i = 0 ; i < fileArray.length ; i++)
{
    document.writeln(fileArray[i].innerHTML + "<br />");
}
</script>
</body>
```

在上面代码中可看出 menuBar 元素包含多个<div.../>子元素，如果直接获取 menuBar 元素内所有<div.../>子元素将返回一个 HTMLCollection 对象，程序使用$A()函数将该对象转换为一个数组。操作数组比操作 HTMLCollection 元素简单多了，所以程序可以通过遍历数组元素来访问每个<div.../>子元素。

$A()函数用于将一个参数转换成数组。如果传入的参数不是一个集合，而是一个普通变量，$A()函数将返回一个空数组，而不是只有一个元素的数组。

注意：

如果$A()函数的参数是一个普通变量，而不是一个集合，该函数将返回一个空数组，该数组不包含任何数组元素。

$A()函数还可以操作字符串，$A()函数将以字符串里每个字符作为数组元素。看下面的代码片段：

程序清单：codes\09\9.2\$A()Test2.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
var str = "crazyjava";
//使用$A 将一个字符串转换成字符数组
var strArray = $A(str);
//依次输出每个字符数组中的每个字符
for (var i = 0 ; i < strArray.length ; i++)
{
    document.writeln(strArray[i] + "<br />");
}
```

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

上面代码负责将“crazyjava”字符串转换成一个字符数组，字符数组里的每个数组元素将依次是c、r、a、z、y、j、a、v、a等字符。

➤➤9.2.4 使用$F()函数

$F()函数用于获取表单控件的值，比如input、textArea、selcet元素等，该函数既可用表单控件的id值作为参数，也可直接使用表单控件作为参数。

$F()函数甚至不要求表单控件处于<form.../>元素内，$F()函数一样可以获得该表单控件的值。$F()函数不管页面中有多个<form.../>元素，也不管该表单控件处于哪个<form.../>元素内，它总是返回页面中第一个满足要求的表单控件的值，看下面代码：

程序清单：codes\09\9.2\$F()Test1.html

```
<body>
<form id="form1" name="form1" action="#">
    <input type="text" id="text1" name="text1" />
</form>
<form id="form2" name="form2" action="#">
    <input type="text" id="text2" name="text2"/><br />
    <textarea cols="40" rows="2" id="text3" ></textarea>
</form>
<div id="show"></div>
<input type="button" onclick="show()" value="显示"/>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//事件处理函数，用于输出三个表单控件的值
function show()
{
    $("show").innerHTML = $F("text1") +
        "<br />" + $F("text2") +
        "<br />" + $F("text3");
}
</script>
</body>
```

与$()函数不同的是，$()函数是获取的HTML元素本身，而$F()函数用于获取表单控件的值，而不是表单域本身。可以这样说：$()函数返回的是一个HTML元素对象，而$F()返回的只是一个字符串值。在浏览器中浏览上面的页面，并为三个输入框中分别输入3个字符串，然后单击“显示”按钮，将出现如图9.6所示的界面。

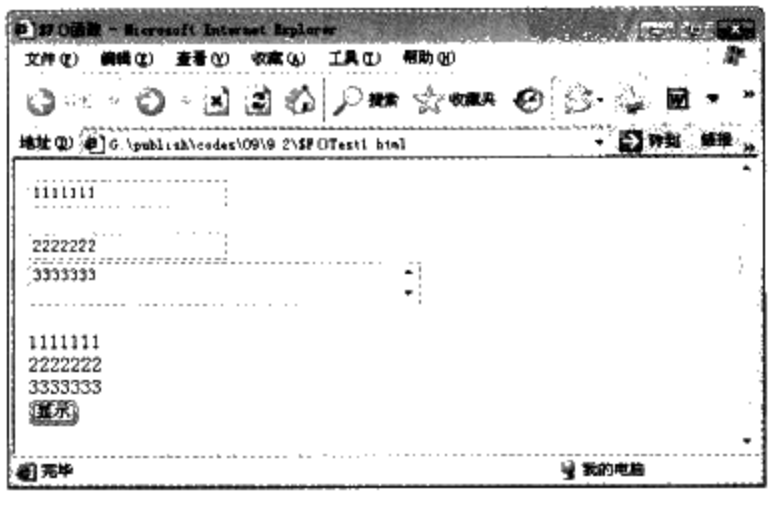


图9.6 使用$F()函数返回表单域的值

有一点$F()函数与$()函数是相同的：当在 Internet Explorer 中使用$F()函数时，该函数不仅可以根据 id 属性来获取表单控件的值，也可根据 name 属性来获取表单控件的值。如果表单控件的 id 属性和 name 属性不一致，则可能导致错误。

注意：

使用$F()函数和$()函数都应该让 HTML 元素的 id 属性和 name 属性保持一致。

9.2.5 使用$H()函数

$H()函数用于将 JavaScript 对象转换成 Hash 对象，Hash 类是 Prototype 库提供了一个类，它非常类似于 Java 语言里的 Map 数据结构，它总是由一系列的 key-value 对组成。Hash 对象里包含了一些方便的方法来操作 Hash 对象。对于如下代码片段：

程序清单：codes\09\9.2\$H()Test1.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
    // 将 navigator 对象里的属性名和属性值转换成对应的 Hash 对象
    var nav = $H(navigator);
    // 输出该 Hash 对象
    alert(nav.inspect());
</script>
</body>
```

上面程序中使用的 inspect()是 Hash 对象提供的一个工具方法，用于将该 Hash 对象的 key、value 全部输出，在浏览器中浏览该页面，将看到如图 9.7 所示的界面。

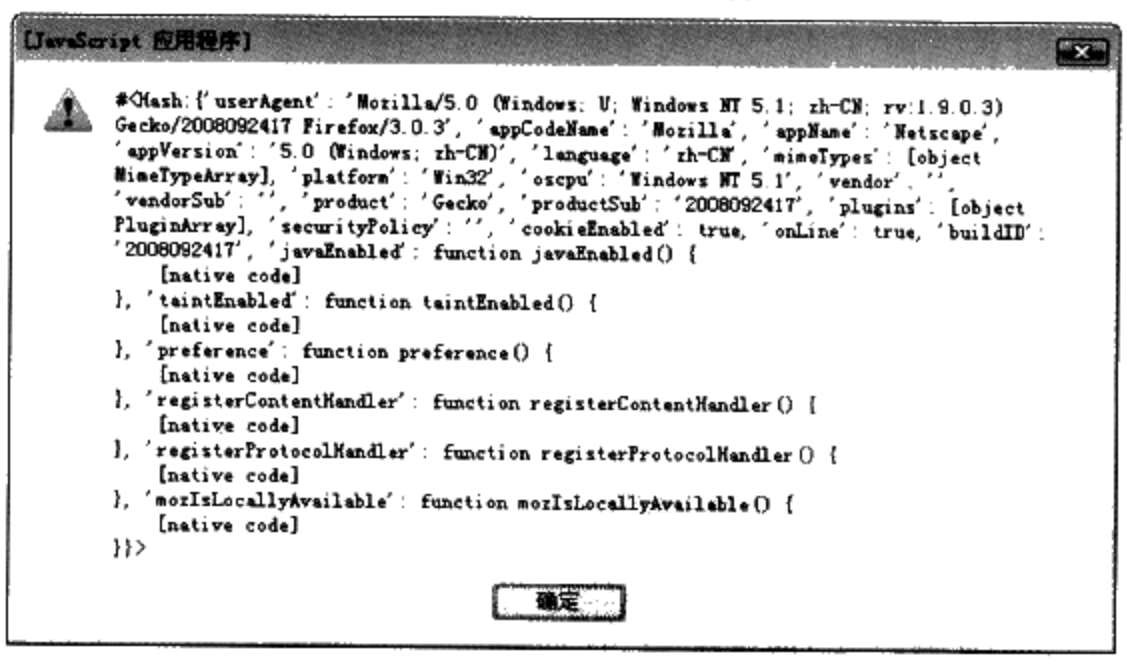


图 9.7 使用$H()函数

注意：

Internet Explorer 6 对 Hash 对象的支持不是太好，如果我们使用 Internet Explorer 6 浏览器来运行上面的程序将无法看到图 9.7 所示效果。因为应该尽量避免在 Internet Explorer 6 浏览器中使用$H()函数。

如果$H()函数的参数是不包含任何属性的 JavaScript 对象，$H()函数将返回一个空 Hash 对象。这里所说的空，并不是说该对象为 null，而是指它不包含任何 key-value 对。例如下面的代码片段：

程序清单：codes\09\9.2\$H()Test1.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
    //将空对象变成 Hash 对象
    var nav = $H({});
    //输出该 Hash 对象
    alert(nav.inspect());
</script>
</body>
```

上面的代码将{}变成一个 Hash 对象，该 Hash 对象并不会成为一个 null，而是一个不包含任何 key-value 对的 Hash 对象。

本章后面还有关于 Hash 对象更详细的介绍，读者可参照后面内容来学习$H()函数的用法。

➤➤9.2.6 使用$R()函数

$R()函数是一个省略的写法，用于构造一个 ObjectRange 对象，ObjectRange 也是 Prototype 库的一个自定义类，该类包含了一些简单的工具方法，可以很方便的枚举该对象里的元素。$R()函数只是 new ObjectRange(lowBound,upperBound,excludeBounds)的缩写形式。该函数的语法格式如下：

- $R(start, end[, exclusive = false])：start 指定 ObjectRange 的起始值，end 指定 ObjectRange 的结束值，exclusive 指定 ObjectRange 对象是否排除两个边界值，默认不排除。

对于如下代码片段：

程序清单：codes\09\9.2\$R()Test.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//构造一个 ObjectRange 对象
var range = $R(10 , 20 , false);
//遍历 ObjectRange 对象的每个元素
range.each(function(value, index)
{
    document.writeln(value + "<br />");
});
</script>
</body>
```

上面代码使用$R()函数构造了一个从 10 到 20 的 ObjectRange 对象，再使用 each()方法遍历该对象中的每个元素。each 方法来自于 Enumerable，它是 ObjectRange 对象的父类，该类提供了一些非常优雅的方法用于遍历列表中的元素。

提 示：

Java 程序员可能对 ObjectRange 对象的 each()方法感到困惑：这个 each()方法的参数不再是简单值，而是函数。但 Ruby 程序员可能会 each()方法发出会心一笑——这就是 Ruby 迭代器的处理方式。由于 Prototype 库最初是属于 Ruby On Rails 项目的，因此很多地方都有一股“Ruby 味道”。

➤➤9.2.7 使用 Try.these 函数

Try.these()函数允许传入一系列的函数作为参数，Try.these()函数将依次调用传入的一系列函数，找到第一个能成功返回值的函数，并将该函数的返回值作为 Try.these()函数的返回值。如果这一系列函数都没有返回值，Try.these()将会返回 undefined。

该函数主要为了满足 JavaScript 在不同浏览器上运行的需要，同样的 JavaScript 函数，在不同的浏览器中可能有不同的结果，甚至可能运行失败。为了解决 JavaScript 跨浏览器的问题，我们常常需要使用循环来依次调用每个函数，并使用 try...catch 来捕捉异常，假设对于如下常用的 JavaScript 代码片段，该代码片段常用于在 Internet Explorer 中创建的 XMLHttpRequest 对象。

```
//定义一个数组，保存 IE 中 XMLHttpRequest 的所有可能实现
var MSXML = ['MSXML2.XMLHTTP.5.0', 'MSXML2.XMLHTTP.4.0',
    'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
//遍历每个数组元素，试图创建 XMLHttpRequest 对象
for(var n = 0; n < MSXML.length; n ++)
{
    try
    {
        var objXMLHttp = new ActiveXObject(MSXML[n]);
        break;
    }
    catch(e)
    {
    }
}
```

上面代码需要依次创建 Internet Explorer 中的 XMLHttpRequest 对象，可以使用 Try.these 函数达到同样效果。

程序清单：codes\09\9.2\TrytheseTest1.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//使用 Try.these 创建 XMLHttpRequest 对象
//依次调用多个函数，直到遇到第一个函数的有返回值时，
//该返回值将作为 Try.these()函数的返回值
var objXMLHttp=Try.these(
    function() {return new XMLHttpRequest();},
    function() {return new ActiveXObject('MSXML2.XMLHTTP.5.0');},
    function() {return new ActiveXObject('MSXML2.XMLHTTP.4.0');},
    function() {return new ActiveXObject('MSXML2.XMLHTTP.3.0');},
    function() {return new ActiveXObject('MSXML2.XMLHTTP');},
    function() {return new ActiveXObject('Microsoft.XMLHTTP');}
);
alert(objXMLHttp);
</script>
</body>
```

上面程序中粗体字代码可在各种浏览器中创建 XMLHttpRequest 对象，程序将依次调用 Try.these()函数里多个函数，直到遇到第一个函数有返回值时，该返回值将作为 Try.these()函数的返回值。

Try.these()的参数是多个函数的引用，而不是调用函数。因此作为 Try.these()参数的多个函数后不能有括号。看如下代码片段：

程序清单：codes\09\9.2\TrytheseTest2.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//定义一个 aa 函数
function aa()
{
```

```
    return "hello";
}
//Try.these()的参数只是函数的引用，而不是调用函数
var b = Try.these(aa);
alert(b);
</script>
</body>
```

上面的代码是正确，但如果在 Try.these 中使用 aa()将变成错误的（除非调用 aa()再次返回一个函数）。一旦为 aa 函数增加了括号，表示调用该函数，而不是该函数的引用。

注意：

Try.these 函数的参数只能是函数引用。

9.3 Prototype 的 JSON 支持

前面已经提到：JSON 格式的数据交换具有轻量级、易理解、跨语言的优势，因此 JSON 对象常常被作为 Ajax 技术的数据交换格式，Prototype 库从 1.5 开始提供 JSON 支持，Prototype 库的 JSON 支持为 Date、Object、Array、Hash、Number 类增加了 toJSON()方法，toJSON()方法用于将这些对象转换成一个 JSON 格式的字符串。

除此之外，Prototype 库还为 String 类增加了如下三个与 JSON 相关的方法：

- isJSON()：该方法判断指定字符串是否为合法的 JSON 字符串。
- evalJSON([sanitize = false])：用于将指定字符串转换成 JSON 对象。
- toJSON()：用于将指定字符串转换成 JSON 字符串。

下面代码示范了 Prototype 库所提供的 JSON 支持：

程序清单：codes\09\9.3\JSONTest.html

```
<body>
<script src="../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
var date = new Date();
//将 Date 对象转化成 JSON 字符串
document.writeln("日期对应的 JSON 字符串为：" + date.toJSON() + "<br />");
var p = {
    name : "yeeku",
    age : 30
};
//将对象转换成 JSON 字符串
document.writeln("普通对象的 JSON 字符串为：" + Object.toJSON(p) + "<br />");
var books = ["疯狂 Java 讲义" , "疯狂 Ajax 讲义"];
//将数组转换成 JSON 字符串
document.writeln("数组的 JSON 字符串为：" + books.toJSON() + "<br />");
var hash = $H({name: 'yeeku', age : 30 });
//将 Hash 对象转换成 JSON 字符串
document.writeln("Hash 对象的 JSON 字符串为：" + hash.toJSON() + "<br />");
//将数值转换成 JSON 字符串
document.writeln("Hash 对象的 JSON 字符串为：" + (45).toJSON() + "<br />");
//下面四行代码测试怎样才算合法的字符串
document.writeln('"crazyjava".isJSON()的结果为：'
    + "crazyjava".isJSON() + "<br />");
document.writeln('"\"crazyjava\"".isJSON()的结果为：'
    + "\"crazyjava\"".isJSON() + "<br />");
```

```
document.writeln('"{age: 30}.isJSON()的结果为：'
    + "{age: 30}".isJSON() + "<br />");
document.writeln('"{\"age\": 30}".isJSON()的结果为：'
    + "{\"age\": 30}".isJSON() + "<br />");
//定义一个 JSON 格式的字符串
var str = '{"name": "yeeku", "age": 30}';
//将 JSON 格式的字符串转换成 JSON 对象
var author = str.evalJSON();
document.writeln('author 对象的 name 值为：' + author.name + "<br />");
document.writeln('author 对象的 age 值为：' + author.age + "<br />");
</script>
</body>
```

上面程序中粗体字代码示范了 Prototype 库提供的 JSON 支持，运行上面程序看到如图 9.8 所示效果。

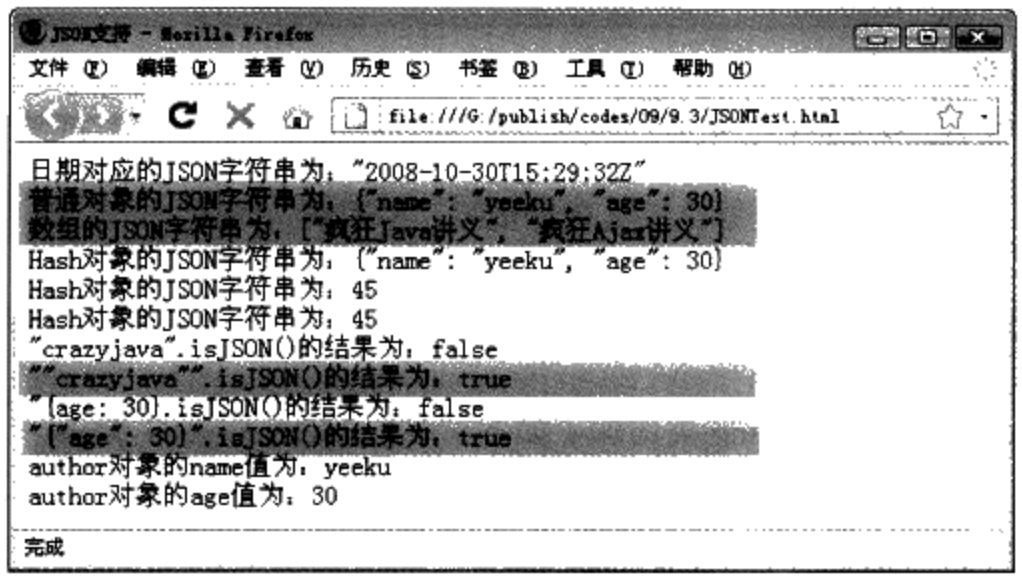


图 9.8　Prototype 的 JSON 支持

从图 9.8 的灰色覆盖区域可以看出：符合 JSON 格式的字符串要求所有 key 都处于引号之内，但从前面程序中可以看出，实际定义 JSON 对象时 key 则无需处于引号之内。

9.4　Prototype 的自定义对象和类

Prototype.js 提供了大量的自定义对象和类，这些自定义对象和类包含了更多有用功能和方法，这些对象和类简化 JavaScript 中 HTML 元素的操作，以及对表单元素的操作。借助于这些对象和类，可以大大减少自己的 JavaScript 代码编写。

➤➤9.4.1　使用 Element 对象

该类提供了一系列的方法用于简化 HTML 元素的操作，包括通过 CSS 改变 HTML 元素的外观，或直接提供一些方法为 HTML 元素提供动态显示效果。Element 类大致提供了如下常用方法：

- addClassName(element, className)：用于为某个 HTML 元素增加 CSS 样式。第一个参数 element，既可以是元素的 ID 属性，也可以是元素本身。 className 是 CSS 样式的名字。
- classNames(element)：用于返回某个 HTML 元素上所有的 CSS 样式（一个 HTML 元素可以对应多个 CSS 样式的）。参数 element 既可以是元素的 ID 属性，也可以是元素本身。
- cleanWhitespace(element)：如果该元素的某个子元素只有文本值，且文本值是空白，则该子元素被删除。参数 element 既可以是元素的 ID 属性，也可以是元素本身。
- empty(element)：判断某个元素是否为空，或者只包含空字符串。如果是，则返回真，否则返回假。参数 element 既可以是元素的 ID 属性，也可以是元素本身。
- getDimensions(element)：用于获取某个 HTML 元素的大小，返回值有两个属性，height 和 width。

参数 element 既可以是元素的 ID 属性，也可以是元素本身。

- getHeight(element)：用于返回某个 HTML 元素的高。参数 element 既可以是元素的 ID 属性，也可以是元素本身。
- getStyle(element, cssProperty)：用于返回某个 HTML 元素的内联 CSS 样式的属性值，如果没有指定内联的 CSS 样式，则返回 null。参数 element 既可以是元素的 ID 属性，也可以是元素本身。
- hasClassName(element, className)：用于判断某个 HTML 元素是否包含指定的 CSS 样式。如果包含制定的 CSS 样式，则返回 true，否则返回 false。参数 element 既可以元素的 ID 属性，也可以是元素本身。
- hide(elem1 [, elem2 [, elem3 [...]]])：同时隐藏多个 HTML 元素，通过设置 style.display='none' 来隐藏，因此会、释放元素在页面上所占的空间。传入的每个 element 既可以是元素的 ID 属性，也可以是元素本身。
- insert(element, {position: content})/insert(element, content)：该方法用于在 element 元素之前、之后、顶端、底端插入 content 元素。其中 position 可以为 before（之前）、after（之后）、top（顶端）和 bottom（底端）四个值，如果不指定 position，则 content 默认被插入在 element 的底端。
- makeClipping(element)：通过设定 overflow 的值设置内容溢出。element 参数既可以是元素 ID，也可以是元素本身。
- makePositioned(element)：将某个元素的内联 CSS 属性 style.position 设置为 relative。element 参数既可以是元素 ID，也可以是元素本身。
- remove(element)：从 document 对象中删除指定的 HTML 元素。element 参数既可以是元素 ID，也可以是元素本身。
- removeClassName(element, className)：为指定的 HTML 元素删除特定的 CSS 样式。element 参数既可以是元素 ID，也可以是元素本身。
- scrollTo(element)：将 window 滚动到对象所在的位置。element 参数既可以是元素 ID，也可以是元素本身。
- setStyle(element, cssPropertyHash)：为 HTML 元素设置内联 CSS 样式，element 参数既可以是元素 ID，也可以是元素本身；参数 cssPropertyHash 是个对象，该对象的属性名和属性值就是 CSS 样式名和值。
- show(elem1 [, elem2 [, elem3 [...]]])：将多个 HTML 元素在页面上显示出来，通过设置 style.display 为""完成，与 hide 正好相反。参数中的 element 既可以元素的 ID 属性，也可以是元素本身。
- toggle(elem1 [, elem2 [, elem3 [...]]])：控制传入的多个 HTML 元素整体显示、隐藏，即在显示和隐藏之间切换。传入的 element 既可以是元素本身，也可以是元素 ID。
- undoClipping(element)：恢复元素的 style.overflow 的值。传入的 element 既可以元素本身，也可以是元素的 ID 属性。与 makeClipping 方法相反。
- undoPositioned(element)：将 HTML 对象的 style.position 设置为""，element 既可以元素的 ID 属性，也可以是元素本身。与 makePositioned 方法相反。
- update(element, html)：设置给定 HTML 元素的 innerHTML 属性。如果 html 参数中包含<script>，那么它们不会被包含进去，但是会执行。
- visible(element)：判断某个 HTML 元素是否可见，如果该对象可见，则返回 true，否则返回 false。

下面给出一些关于这些方法的示范代码，但由于篇幅关系，此处不能将所有的代码一一给出示范

代码。先看 addClassName 方法，该方法用于为某个 HTML 元素增加 CSS 样式，看如下代码：

程序清单：codes\09\9.4\Element\changeCss.html

```
<body>
<input type="button" onclick='chg()' value="增加立体效果"/>
<div id=up >有立体效果的层</div>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function chg()
{
    //为 up 元素增加 solid 的 CSS 样式
    Element.addClassName("up","solid");
}
</script>
</body>
```

在浏览器中浏览该页面，如果单击“增加立体效果”按钮，将为 up 元素增加了立体效果，这种立体效果是由于 solid 的 CSS 样式表现出来的。下面代码片段示范了 toggle 方法的使用：

程序清单：codes\09\9.4\Element\toggle.html

```
<input type="button" onClick='chg()' value="切换显示"/>
<div id="div1">层 1</div>
<div id="div2">层 2</div>
<div id="div3">层 3</div>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function chg()
{
    //同时控制 div1 div2 div3 三个元素的显示和隐藏
    Element.toggle("div1");
    Element.toggle("div2");
    Element.toggle("div3");
}
</script>
```

如果在 HTML 文档中通过事件激发 chg()函数，div1、div2、div3 三个元素将一起显示、隐藏交替出现。

下面的代码片段示范了 makePositioned 和 undoPositioned 两个函数的作用，这两个函数分别用于设置，取消目标元素的 style.postion 属性为 relative。从前面关于 CSS 样式属性的介绍中知道，一旦设置 style.position 属性为 relative，则可直接通过设置 left、top 属性来改变 HTML 元素的位置。

程序清单：codes\09\9.4\Element\makePositioned.html

```
<body>
<input type="button" onclick='set()' value="设置 postion" />
<input type="button" onclick='unset()' value="取消 postion" />
<div id="div1" style="left:80px;top:20px;">
移动的层
</div>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function set()
{
    //设置起 style.postion 属性为 relative
    Element.makePositioned("div1");
```

```
}
function unset()
{
    //设置 style.position 属性为""
    Element.undoPositioned("div1");
}
</script>
</body>
```

在浏览器中浏览上面页面，一旦单击了“设置 position”按钮，“移动的层”元素的位置将发生改变，因为其 style.position 属性为 relative 时，可直接设置该元素的坐标。

Element 的 insert()方法可以非常方便地动态更新 HTML 页面，如下程序所示：

程序清单：codes\09\9.4\Element\

```
<body>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<div id="test" style="border:2px solid black;width:200px;">
    <div>test 元素的第一个子元素</div>
    <div>test 元素的最后一个子元素</div>
</div>
<input type="button" value="After 插入" onclick="Element.insert($('test')
    , {after: '<b>After 插入的内容</b><br />'});"><br />
<input type="button" value="Before 插入" onclick="Element.insert($('test')
    , {before:'<b>Before 插入的内容</b><br />'});"><br />
<input type="button" value="Top 插入" onclick="Element.insert($('test')
    , {top:'<b>Top 插入的内容</b><br />'});"><br />
<input type="button" value="Bottom 插入" onclick="Element.insert($('test')
    , {bottom:'<b>Bottom 插入的内容</b><br />'});"><br />
<input type="button" value="默认插入" onclick="Element.insert($('test')
    , '<b>默认插入的内容</b><br />');"><br />
</body>
```

上面程序中定义了 5 个按钮，这 5 个按钮分别用于将指定内容插入 id 为 test 的<div.../>元素的各个位置，在浏览器中浏览器该页面，并依次单击 5 个按钮将看到如图 9.9 所示效果。

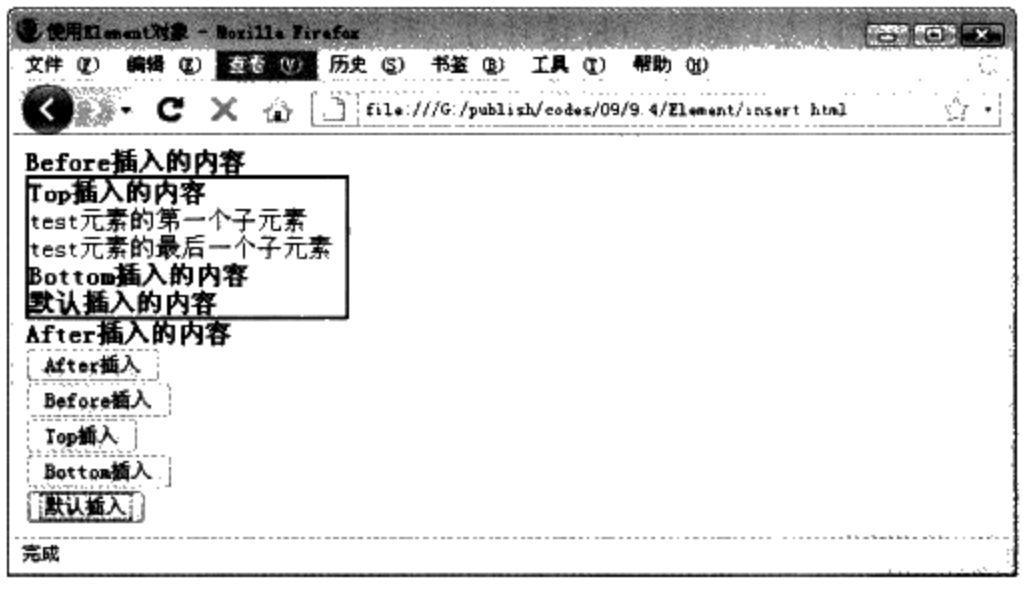


图 9.9　使用 Element 的 insert()方法

提　示：

Prototype 库的 Element 对象下包含了大量方法，本节仅仅介绍而来 Element 对象的常用方法。如果读者需要更全面了解 Element 对象所包含的方法，读者可以登录 http://www.prototypejs.org/api 了解更多方法的介绍。

➤➤9.4.2 使用 Element.Methods

Element.Methods 里包含的方法与 Element 里包含的方法大致相似，只是 Element.Methods 里定义的方法被直接混入了 DOM Element 元素，因此可以直接使用 DOM Element 元素来调用这些方法，例如我们可以 9.4.1 节中 changeCss.html 页面的 chg()函数修改为如下形式：

程序清单：codes\09\9.4\Element.Methods\changeCss.html

```
function chg()
{
    //为 up 元素增加 solid 的 CSS 样式
    $("up").addClassName("solid");
}
```

对比 9.4.1 节和 9.4.2 节的 changeCss.html 页面中的 chg()函数，我们发现这两个函数作用完全相同，只是写法上存在差异而已。相比之下，使用 Element.Methods 的方法更符合面向对象的特征。

基本上下面两种写法的作用完全一样：

```
//使用 Element 对象的 xxxMethod()方法操作 element 对象
Element.xxxMethod(element);
//用 element 调用 xxxMethod()方法
element.xxxMethod();
```

提 示：

因为使用 Element.Methods 里的方法会更符合面向对象的思维方式，因此通常推荐使用 Element.Methods 里的方法代替 Element 里的方法。

➤➤9.4.3 使用 Enumerable

Enumerable 提供了大量操作数组、集合和枚举的方法，Enumerable 通常不能直接创建对象，它是 Prototype 库的基础，它的方法将被混入（mixin）其他类、对象中，从而允许其他类或对象调用这些方法。

提 示：

Prototype 的作者是一个 Ruby fans，所以他设计 Prototype 时总是试图模仿 Ruby 的方式；而 Enumerable 正是对应于 Ruby 里的模块的概念，Enumerable 里提供了大量方法，但这些方法并不是专门为 Enumerable 设计的，而是打算混入（mixin）其他对象中，被其他对象所调用。在 Prototype 库中，Enumerable 的方法主要被混入了 Array、Hash 和 ObjectRange 类里。

还有一点需要指出，由于 Ruby 支持使用代码块作为参数，因此 Enumerable 里许多方法都可使用函数作为参数，这一点会在后面代码中看到示例。

- each(iterator)：遍历 List 对象中的每个元素，interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- all([iterator])：用于测试集合中所有元素是否满足某个条件。该函数会用指定的 iterator 函数依次测试每个集合元素，如果集合中任一元素在 iterator 函数测试中返回 false 或 null，那么这个函数返回 false，否则返回 true。如果没有给出 iterator，只要集合中任一个元素的值为 false 或 null，该函数返回 false。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- any(iterator)：用于测试集合中是否包含任一元素满足某个条件。该函数会用指定的 iterator 函数依次测试每个集合元素，如果集合中任一元素在 iterator 函数测试中返回 true，那么这个函数返回 true，否则返回 false。如果没有给出 iterator，该函数只要集合中有任一元素的值为 true，则该函数返回 true。interatr 是一个形如 function(value, index)的函数，该函数里 value 就

是 Enumerable 对象中的元素，而 index 是集合中元素的索引。

- collect(iterator)：使用 interator 函数依次处理集合中的每个元素，并将处理结果收集成一个数组对象，最后返回得到的数组。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- detect(iterator)：用于获取集合中第一个满足某个条件的元素。该函数会使 iterator 函数依次处理集合中的每个元素，如果遇到一个元素的值在 iterator 函数中返回 true，则返回该元素；如果一直没有出现该元素，则返回 null。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- entries()：直接将该集合转换成一个数组对象。
- find(iterator)：与 detect 的效果完全相同，Prototype 库推荐优先考虑使用 find()函数。
- findAll(iterator)：获取集合中所有满足条件的元素，使用 interator 函数依次处理集合中的每个元素，如果该元素在 iterator 函数中执行后返回 true，则该元素被收集到结果数组中，最后返回得到的数组。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- grep(pattern [, iterator])：找出集合中所有匹配 pattern 正则表达式的元素，返回所有满足匹配该正则表达式的元素。如果给出了 Iterator，则每个结果还要经过 Iterator 函数处理，最后返回由于 iterator 处理结果所组成的数组。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- include(obj)：判断集合是否包含指定对象。包含则返回 true，否则返回 false。
- inject(initialValue, iterator)：这是一个递归执行的函数，iterator 是一个形如 function(accumulator, value, index)的函数，iterator 总是将上一次执行的结果传给 accumulator 参数。第一次迭代时，accurmelator 等于 initialValue，最后返回 accumulator 的值。
- invoke(methodName [, arg1 [, arg2 [...]]])：依次使用每个集合元素作为参数调用 methodName 方法，调用这些方法时，会额外传入 arg1、arg2、arg3…等参数。最后返回所有元素调用 methodName 方法返回值组成的数组。
- map(iterator)：与 collect()相同。
- max([iterator])：如果该函数指定了 iterator 函数，则使用 iterator 函数依次处理集合中每个元素，最后处理结果的最大值；如果没有 iterator 函数，则直接返回集合中的最大值。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- member(obj)：与 include()相同。
- min([iterator])：如果该函数指定了 iterator 函数，则使用 iterator 函数依次处理集合中每个元素，最后返回处理结果的最小值；如果没有 iterator 函数，则直接返回集合中的最小值。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- partition([iterator])：该函数会调 iterator 函数依次处理集合中的每个元素，将所有处理结果为 true 的元素组成第一个数组返回，其余的作为第二个数组。如果没有 iterator 函数，则直接判断集合元素，所有为 true 的集合元素收集到第一个数组里返回，所有为 false 的集合元素收集到第二个数组里返回。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- pluck(propertyName)：返回集合所有元素的指定属性值所组成的数组。
- reject(iterator)：与 findAll()强好相反，获取集合中所有满足不某个条件的元素，使用 iterator 函数依次处理集合中的每个元素，如果该元素在 iterator 函数中执行后返回 false，则该元素

被收集到结果数组中，最后返回该数组。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。

- select(iterator)：与 findAll()函数相同。
- sortBy(iterator)：该函数使用 itertor 依次调用每个元素，最后根据返回值对集合元素排序，最后返回排序后的数组。interatr 是一个形如 function(value, index)的函数，该函数里 value 就是 Enumerable 对象中的元素，而 index 是集合中元素的索引。
- toArray()：与 entries()方法完全相同。
- zip(collection1[, collection2 [, ... collectionN [,transform]]])：将 collection1 到 collectionN 集合中相同索引的元素抽取出来，并将这些元素组成一个一维数组，最后返回多个一维数组组成二维数组。

下面也给出关于上面函数的测试，下面的代码是对 collect 方法的示范：

程序清单：codes\09\9.4\Enumerable\collect.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
var a = [20,30,40,50];
alert(a.collect(function (value , index)
{
    return value * index;
}));
</script>
```

collect()函数将会使用 function 处理 a 数组，collect()函数返回处理结果数组，运行上面程序返回的结果是 0, 30 , 80 , 150，就是 0*20 , 1 * 30 , 2*40 , 3 * 50 的结果。

下面是 each 函数的用法示范：

程序清单：codes\09\9.4\Enumerable\each.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//定义一个数组
var a = [20,30,40,50];
//each 自动遍历 a 数组的每个元素，并将元素、索引传入函数
a.each(function(value , index)
{
    $("show").innerHTML += "第" + index + "元素的值是： "
        + value + "<br />";
});
</script>
```

在浏览器中浏览页面，将看到一行一行的输出，这些就是遍历集合元素时产生的输出。

下面代码测试了 inject 函数：

程序清单：codes\09\9.4\Enumerable\inject.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
var a = [20,30,40,50];
var b = a.inject(5 ,function (acc , value, index)
{
    //访问每次递归时 acc 的值。
    document.writeln(acc + "<br />");
    return value * index + acc;
```

```
});
document.writeln(b + "<br />");
</script>
```

在浏览器中测试该页面，看到如图 9.10 所示效果。

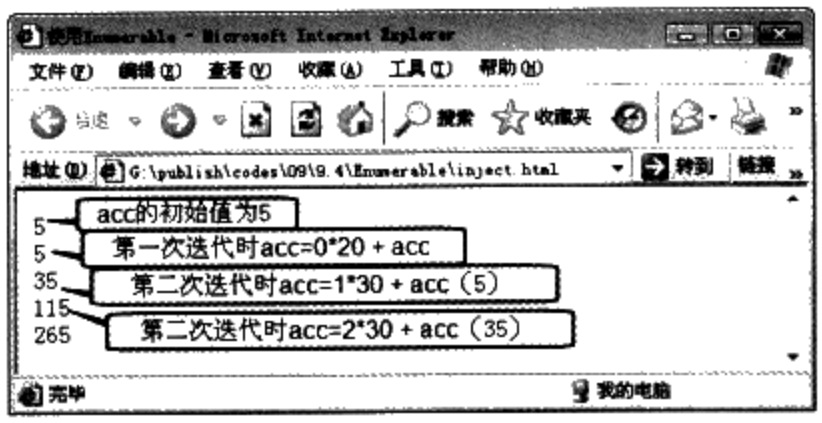


图 9.10　使用 inject()函数

对集合的第一个元素迭代时，5 作为 acc 值直接传入，iterator 函数返回 0 * 20 + 5，其结果等于 5；接着对第二个元素迭代，此时，acc 的值等于 iterator 函数上次返回值：5，此时 iterator 函数返回 1 * 30 + 5，其结果为 35。接着第三次迭代：2 * 40 + 35 = 115；接着第四次迭代：3 * 50 + 115 = 265，这就是最终的返回值。

下面代码测试了 zip 函数：

程序清单：codes\09\9.4\Enumerable\zip.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
document.writeln([1,2,3].zip([4,5,6], [7,8,9]).inspect() + "<br />");
document.writeln([1,2].zip([4,5,6], [7,8,9]).inspect() + "<br />");
document.writeln([1,2,3].zip([4,5,6], [7,8]).inspect() + "<br />");
</script>
```

该页面在浏览器中浏览将出现如图 9.11 所示。

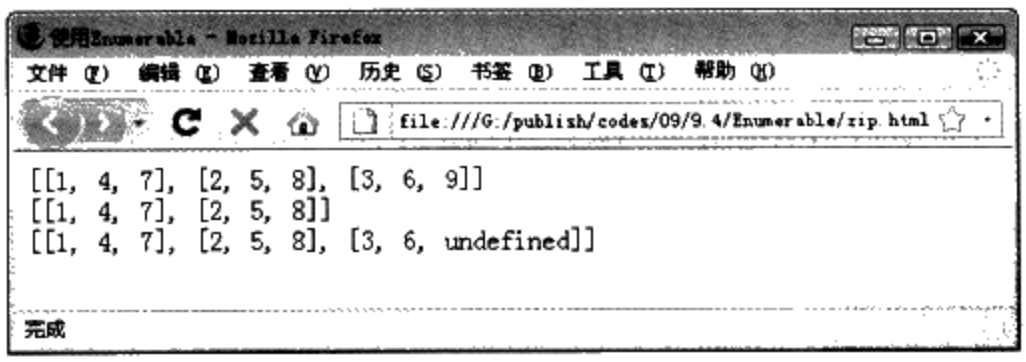


图 9.11　使用 zip 函数

从图 9.11 中可看出，zip()函数返回的二维数组的长度总是等于调用 zip()函数的集合的长度。该函数依次将被添加的集合里的每个元素抽取出来，然后根据 index 相同的规则，与调用 zip 方法的集合元素组成数组。

Enumerable 类的方法实在太多，此处由于篇幅问题不可能一一给出示范，希望读者参考上面的方法的介绍自行理解。

提 示：

关于 Enumerable 里的很多方法，笔者试图用语言表达时可能会产生一些歧义（包括读者阅读 Prototype 库的官方 API 时也会有相同的感觉），读者无需纠缠于字面意义上的说法，通过完全可以通过代码测试一下某个方法的功能，就可以面白这些方法的用途了。

➤➤9.4.4 使用 ObjectRange

ObjectRange 是 Prototype 库提供的一个自定义类，一个 ObjectRange 对象代表一个范围，例如从 2 到 9 之间的所有数组就是一个 ObjectRange 对象。创建 ObjectRange 有两种方法：

- 利用 ObjectRange 构造器。
- 利用$R()函数。

上面两种方法所支持的参数完全一样，两种方式创建的 ObjectRange 也完全一样。

ObjectRange 混入了 Enumerable 模块的方法，因此完全可以调用 Enumerable 模块里的方法。下面程序示范了 ObjectRange 的用法：

程序清单：codes\09\9.4\RangeObject\RangeObjectTest.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//创建一个 ObjectRange 对象
var range = new ObjectRange(2, 9 , true);
//编辑 ObjectRange 对象的元素
range.each(function(ele , index)
{
    document.writeln("索引" + index + "处的值为：" + ele + "<br />");
});
//使用$R()函数创建一个 ObjectRange 对象
var ra = $R('a' , 'e');
//将 ObjectRange 转换为数组
document.writeln($A(ra));
//判断该 ObjectRange 里是否包含'b'
alert(ra.include('b'));
</script>
```

注意：

当创建基于字符串的 ObjectRange 时需要特别小心，对于$R('a' , 'e')这种情况，当然是 a、b、c、d、e 这五个元素；但对于$R('ax' , 'ba')的情况将特别复杂，这个 ObjectRange 将由 ax、ay、az、a{……等很多字符串组成。

➤➤9.4.5 使用 Form.Element 操作表单控件

Form.Element 专门用于操作表单控件，使用它的方法可以激活表单控件，可以判断某个表单控件是否为空，可以清空系列表单控件，可以将表单控件的内容转换为查询字符串等。

Form.Element 包含了如下几个方法：

- clear(field)：清除传入 field 表单控件的值。field 表单控件既可以是表单控件的 id 属性值，也可以是表单控件本身。
- disable(element)：禁用某个表单控件。
- enable(element)：启用某个表单控件。
- present(field)：判断 field 表单控件是否有值。field 表单控件既可以是表单控件的 id 属性值，也可以是表单控件本身。即使表单控件的值是空白（例如空格、制表符等），也将返回 true。
- focus(field)：将焦点移动到指定表单控件。该表单控件既可以是表单控件的 id 属性值，也可以是表单控件本身。
- select(field)：对于像文本框、文本域一样的表单控件，该方法将可选中该表单控件内的文本。对于不是这类元素，该方法没有任何效果。

- activate(field)：同样可用于选中文本框、文本域内的文本，但该方法比 select 更多一个功能，如果目标元素不是文本框、文本域等控件，则把焦点到目标元素上，此时的作用与 focus 相同。
- getValue()：获取指定表单控件的值，该方法实际很少使用，因为使用$F()函数会更加简洁。
- serialize(element)：返回指定表单控件所转换的查询字符串。即返回形如'field=value'的字符串。

Form.Element 是 Prototype 库的 Field 发展起来的，因此 Form.Element 和 Field 实际上是同一个对象。而且 Form.Element 实际上还可作为模块使用，因此它的方法直接被混入（mixin）表单控件中，即表单控件可直接调用这些方法。举例来说，下面三行代码的效果完全一样：

```
//使用 Form.Element 的方法激活 myfield 控件
Form.Element.activate('myfield');
//使用 Field 的方法激活 myfield 控件
Field.activate('myfield');
//让表单控件直接调用 Form.Element 的方法
$('myfield').activate();
```

下面程序简单示范了 Form.Element 模块的方法：

程序清单：codes\09\9.4\Form.Element\Form.Element.html

```
<body>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<!-- 下面定义四个单行文本框，用于被下面的按钮事件操作 -->
<input id="text1" name="text1" type="text" /><br />
<input id="text2" type="text" /><br />
<input id="text3" type="text" /><br />
<input id="text4" type="text" value="crayjava.org"/><br />
<select size="3" id="st1">
    <option>疯狂 Java 讲义</option>
    <option>轻量级 Java EE 企业应用实战</option>
    <option>疯狂 Ajax 讲义</option>
</select><br />
<!-- 单击该按钮将清除第二个文本框的输入 -->
<input type="button" value="清除第二个表单控件的输入"
    onclick="Form.Element.clear('text2');"/><br />
<!-- 单击该按钮将校验第一个文本框的输入，
    当第一个文本框有输入时返回 true -->
<input type="button" value="校验第一个表单控件的输入"
    onclick="alert($('text1').present());"/><br />
<!-- 单击该按钮将会把焦点移到第三个输入框 -->
<input type="button" value="移动焦点到第三个输入框"
    onclick="Form.Element.focus('text3');"/><br />
<!-- 单击该按钮将选中第四个文本框内的文字 -->
<input type="button" value="选中第四个文本框的文本"
    onclick="Form.Element.select('text4');"/><br />
<!-- 单击该按钮让列表框获得焦点 -->
<input type="button" value="让下拉列表获得焦点"
    onclick="$('st1').activate();"/><br />
<!-- 单击该按钮会将第一个文本框的内容转换为查询字符串 -->
<input type="button" value="转换查询字符串"
    onclick="alert(Form.Element.serialize('text1'));"/><br />
<!-- 单击该按钮会将返回第一个文本框的值 -->
<input type="button" value="返回第一个表单控件的值"
    onclick="alert(Form.Element.getValue('text1'));"/><br />
</body>
</script>
</body>
```

上面的代码比较全面地示范了 Form.Element 里方法的用法，运行该页面即可看出 Form.Element 里方法的效果。

➤➤9.4.6 使用 Form 操作表单

Form 和 Form.Element 非常相似，区别只是 Form.Element 是操作指定指定表单控件，而 Form 是操作指定表单，或者该表单内的全部表单控件。Form 里大致包含如下方法：

- disable(form)：禁用指定表单内的所有表单控件。
- enable(form)：启用指定表单内的所有表单控件。
- focusFirstElement(form)：将焦点移动到指定表单中第一个可视的、有效的表单控件。
- findFirstElement(form)：返回表单中第一个有效的表单控件。
- getElements(form)：返回表单内的所有表单控件。该函数返回一个数组。
- getInputs(form [, typeName [, name]])：返回表单中所有的<input.../>元素，并可通过 typeName 和 name 属性对该元素进行过滤。值得注意的是，该函数返回的并不是一个数组，而是一个集合对象。
- reset(form)：重设表单。与调用表单对象的 reset()效果一样。
- request：该方法是发送异步请求的简单方法，它使用 Ajax.Request 对象发送异步请求，该方法会将表单内所有控件转为请求参数，并向表单 action 属性指定的 URL 发送异步请求。不仅如此，调用该方法时还可指定一个 options 参数，该参数将直接传给 Ajax.Request 对象的 options 参数。
- serialize(form)：返回指定表单内的所有表单控件所转换的查询字符串。即返回形如 'field1=value1&field2=value2&field3=value3' 的字符串。

下面代码示范了 Form 里方法的用法和功能：

程序清单：codes\09\9.4\Form\Form.html

```
<body>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<!-- 被操作的目标表单 -->
<form id="form1" name="form1" method="post">
    <!-- 定义四个单行文本框 -->
    <input id="text1" name="text1" type="text" /><br />
    <input id="text2" name="text2" type="text" /><br />
    <input id="text3" name="text3" type="text" /><br />
    <input id="text4" name="text4" type="text" value="xxxxxxxxx"/><br />
    <select id="st1" name="st1" size="3" >
        <option value="java">疯狂 Java 讲义</option>
        <option value="javaee">轻量级 Java EE 企业应用实战</option>
        <option value="ajax">疯狂 Ajax 讲义</option>
    </select><br />
</form>
<!-- 下面按钮将所有表单控件的转换成查询字符串 -->
<input type="button" value="转换查询字符串"
    onclick="alert($('form1').serialize());"/><br />
<!-- 下面按钮将返回表单的第一个有效表单控件 -->
<input type="button" value="返回第一个有效的表单控件"
    onclick="alert($('form1').findFirstElement());"/><br />
<!-- 下面按钮返回所有表单控件 -->
<input type="button" value="返回表单的全部表单控件"
    onclick="alert($('form1').getElements());"/><br />
<!-- 下面按钮返回所有 input 控件 -->
<input type="button" value="返回表单的全部 input"
```

```
    onclick="alert($('form1').getInputs());"/><br />
<!-- 下面按钮启用所有表单控件 -->
<input type="button" value="启用所有表单控件"
    onclick="$('form1').enable();"/><br />
<!-- 下面按钮禁用所有表单控件 -->
<input type="button" value="禁用所有表单控件"
    onclick="$('form1').disable();"/><br />
<!-- 下面按钮将焦点移动到第一个可视的表单控件 -->
<input type="button" value="将焦点移动到第一个可视的表单控件"
    onclick="$('form1').focusFirstElement();"/><br />
<!-- 下面按钮将重设表单 -->
<input type="button" value="重设表单"
    onclick="$('form1').reset();"/><br />
</body>
```

在浏览器中浏览该页面，并在表单控件中输入相应的值，则将看到如图 9.12 所示的页面。

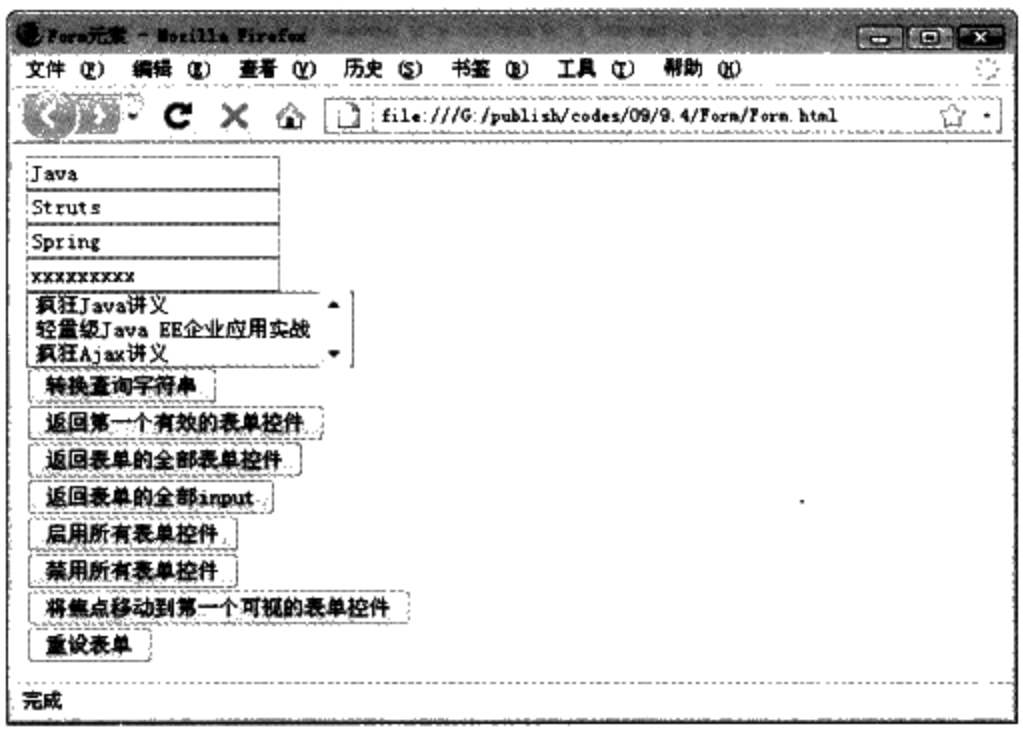


图 9.12　Form 的用法示范

如果单击其中的“转换查询字符串”，则可看到如图 9.13 所示的对话框。

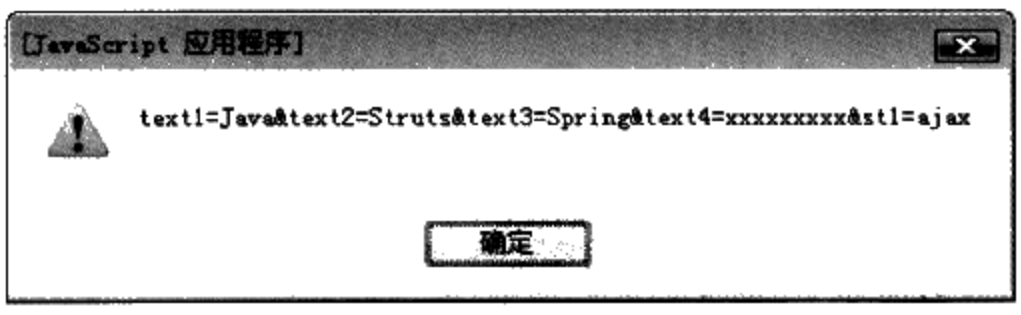


图 9.13　将所有表单控件转换成查询字符串

单击“使所有表单域失效”，则可看到所有表单域变为灰色，不可接收输入。当然，页面中每个按钮依次对应了 Form 的一个方法，读者可以自行单击每个按钮，查看对应的效果。

Form 的 request()方法可以非常方便地发送 Ajax 请求，本章后面内容有关于 request()方法更详细的讲解。

➤➤9.4.7　使用 Hash 对象

Hash 对象是一种类似于 Java 语言里 Map 的数据结构，它是一个 Key-Value 对的集合。Hash 对象的每个 Item 是一个包含两个元素的数组，前一个是 Key，后一个是 Value。每个 Item 都有两个属性：

key 和 value，分别代表 Hash 对象里每 Item 数组的前、后两个元素。该对象大致有如下方法：

- clone()：复制已有的 Hash 对象，返回复制产生的、新 Hash 对象。
- each(iterator)：遍历 Hash 对象里每个 key-value 对的迭代器，其中 iterator 是一个形-如 function(pair){}的函数，其中 pair 就是 Hash 对象的 key-value 对象。
- get(key)：根据 key 返回 value，与 Java Map 的 get()方法基本相似。
- inspect()：返回字符串显示 Hash 对象的 key-value 对，类似于 Java Map 的 toString()方法。
- keys()：返回 Hash 对象的全部 key 组成的数组。
- values()：返回 Hash 对象全部 value 组成数组。
- merge(otherHash)：将新的 Hash 对象合并到原有的 Hash 对象，返回新的 Hash 对象。
- set(key, value)：设置一对 key-value 对，类似于类似于 Java Map 的 put()方法。
- toObject()：将 Hash 对象转换成一个 JavaScript 对象，其实 Hash 对象和 JavaScript 都是 key-value 对，因此可以非常方便地相互转换。

提 示：

借助于$H()方法可将一个 JavaScript 对象转换成 Hash 对象，借助于该 toObject()方法则可将一个 Hash 对象转换成一个 JavaScript 对象。

- toQueryString()：将 Hash 对象转换为查询字符串，这种查询字符串以 'key1=value1&key2=value2 的形式出现。
- unset(key)：删除 key 所对应的 key-value 对，执行该方法后返回该 key 所对应的 value。
- update(object)：使用 object 更新当前 Hash 对象。如果 object 的某个属性名和 Hash 的某个 key 相同，则该 key 对应的 value 将被更新；如果 object 的属性名不和 Hash 对象的任何 key 相同，则该 object 的属性名、属性值将作为 key-value 对加入 Hash 对象中。

借助于$H 函数，可将一个普通对象转换成 Hash 对象，再借助于 Hash 对象的方法可以方便地访问该对象的全部属性和属性值。看如下的代码片段：

程序清单：codes\09\9.4\Hash\Hash.html

```
<body>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//定义一个 JavaScript 对象
var person =
{
    name:'yeeku',
    age:29
};
//将 JavaScript 对象转换成 Hash 对象
var h = $H(person);
//将 Hash 对象转换成查询字符串。输出：name=yeeku&age=29
document.writeln(h.toQueryString() + "<br />");
//测试 merge 方法
var person =
{
    name:'yeeku',
    age:29
};
var teacher =
{
    name:'yeeku.H.lee',
    gender:'male'
};
```

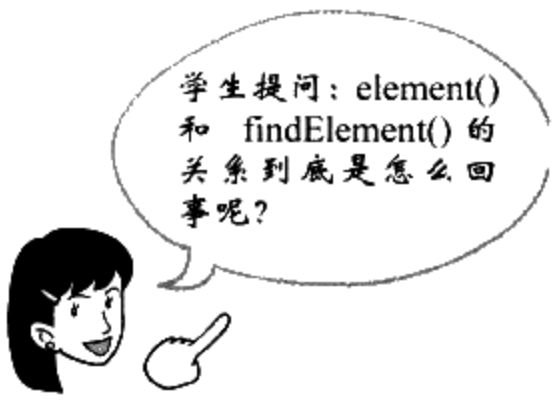
```
//将后一个 Hash 对象 merge 到前一个 Hash 对象中
var h = $H(person).merge($H(teacher));
//输出：name=yeeku.H.lee&age=29&gender=male
document.writeln(h.toQueryString() + "<br />");
//直接使用 JavaScript 对象来更新 Hash 对象
h.update({age : 30 , subject :"java"});
//将输出：name=yeeku.H.lee&age=30&gender=male&subject=java
document.writeln(h.toQueryString() + "<br />");
//遍历 Hash 对象的每个 key-value 对
h.each(function(pair)
{
    document.writeln(pair.key + "-->" + pair.value + "<br />");
});
</script>
</body>
```

上面程序分别使用 merge()和 update()两个方法来更新已有的 Hash 对象，从程序的粗体字代码可以看出，merge()和 update()方法的功能基本相似，区别只是 merge()方法使用另一个 Hash 对象来进行更新，而 update()方法则使用另一个 JavaScript 对象进行更新。

➤➤9.4.8 使用 Event

第 7 章介绍事件编程时候我们发现 JavaScript 事件模型是相当繁琐的，尤其是当我们需要跨浏览器时，Internet Explorer 浏览器和 DOM 2 规范浏览器的差异是如此之大，这种差异带给程序员巨大困扰。Prototype 为了消除这种差异提供了 Event 模块，该模块了包含了大量通用的方法来消除 JavaScript 事件编程的跨浏览器问题。

- element(event)：返回引发 event 事件的 DOM 元素。
- findElement(event, tagName)：该方法和 element()方法有点类似，只是 findElement()并不一定返回事件发生的事件源，它还可能返回该事件源 DOM 对象所在的容器。



答：大部分时候，我们需要根据事件来访问引发事件的 DOM 元素，但有时候我们需要访问该事件源所在容器，例如有如下代码<table><tr><td> <input type="button" value="click me!"/> </td></tr></table>，我们可以为代码中的按钮绑定一个事件监听器，当用户单击该按钮时，触发事件的事件源就是按钮对象，如果用 element()方法将会返回该按钮对象；但如果我们不是对该按钮对象感兴趣，而是对该按钮所在单元格（<tr.../>元素）感兴趣，则可以使用 findElement(event, "td")方法来获取，该方法将会依次搜索该按钮所在的容器，直到找到某个容器和 td 有相同的标签名。在极端的情况下，如果使用 findElement(event, "input")

- isLeftClick(event)：判断是否因为左键单击所引发的事件。
- observe(element, eventName, handler[, useCapture = false])：将 handler 注册成 element 的 eventName 事件的监听器，useCapture 指定 handler 是否在捕获阶段被触发。
- pointerX(event)：返回鼠标事件发生位置的 X 座标。
- pointerY(event)：返回鼠标事件发生位置的 Y 座标。
- stop(event)：停止 event 事件传播。

- stopObserving(element, eventName, handler[, useCapture = false])：取消 element 上绑定的 handler 监听器。

与前面许多模块类似的，Event 实际上作为模块使用，它所包含的方法已经被混入了 DOM 模型的 event 对象，因此可以直接使用 event 来调用上面这些方法。因此如下两行代码的效果完全一样：

```
//两种方法来获取 event 对象的事件源。
var element = Event.element(event);
var element = event.element();
```

如下程序示范了使用 Event 来简化事件编程：

程序清单：codes\09\9.4\Event\Event.html

```
<body>
<table border="1">
<tr><td>
<div><input id="ok" type="button" value="click me!"/></div>
</td></tr>
</table>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
Event.observe("ok" , "click" , function(event)
{
    alert("是否为左键事件：" + event.isLeftClick());
    alert("事件源：" + event.element().value);
    alert("最近的 td 元素：" + event.findElement("td").innerHTML);

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

上面页面代码不再使用原始的事件模型，而是使用基于 Event 提供的事件简化，这种简化消除了事件模型的浏览器差异，因而可以在所有浏览器上运行良好。

除此之外，Event 模块下还包含如下属性：

- KEY_BACKSPACE：该属性为常量 8，表明是退格键。
- KEY_TAB：该属性为常量 9，表明是 Tab 键。
- KEY_RETURN：该属性是常量 13，表明是回车键。
- KEY_ESC：该属性是常量 27，表明是 Esc 键。
- KEY_LEFT：该属性是常量 37，表明是向左箭头。
- KEY_UP：该属性是常量 38，表明是向上箭头。
- KEY_RIGHT：该属性是常量 39，表明是向右箭头。
- KEY_DOWN：该属性是常量 40，表明是向下箭头。
- KEY_DELETE：该属性是常量 46，表明是 Delete 键。

➤➤9.4.9 使用 Template

在有些时候，我们需要生成多个字符串，但这多个字符串中大量内容完全相同，只有少量关键部分发生改变，这时就可以借助于 Template 对象了。

创建 Template 对象可通过如下构造器完成：

- Tempate(pattern)：传入一个 pattern 字符串来创建 Template 对象，pattern 字符串中可使用#{var}形式的变量。

一旦得到了 Template 对象之后，程序就可以调用 Template 提供的如下方法来生成实际所需要的字

符串：

- evaluate(obj)：将 obj 对象的属性值合并到 Template 的 pattern 字符串中生成实际字符串。

如下代码示范了 Template 对象的用法：

程序清单：codes\09\9.4\Template\Template.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
objArr = [
    {book:'Struts2 权威指南' , author:"李刚"},
    {book:'疯狂 Java 讲义' , author:"李刚"},
    {book:'轻量级 Java EE 企业应用实战' , author:"李刚"}
]
var template = new Template("书名是#{book} , 作者是#{author}.");
for (var i = 0 ; i < objArr.length ; i++ )
{
    document.writeln(template.evaluate(objArr[i]) + "<br />");
}
</script>
```

上面程序中先定义了一个 Template 对象，该 Template 的 pattern 字符串里包含#{book}、#{author}两个变量，程序接着定义了 3 个 JavaScript 对象，每个对象都有 book、author 两个属性，运行上面程序，将看到如图 9.14 所示效果。

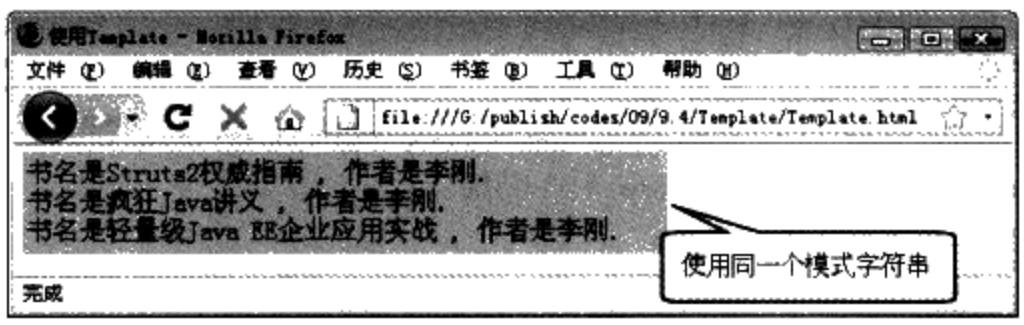


图 9.14　使用 Template

➤➤9.4.10　使用 Class

Class 是 Prototype 库为弥补 JavaScript 不支持面向对象而提供的对象，使用 Class 对象可以定义类，可以定义现有类的子类，可以动态为某个类动态添加方法。Class 对象主要提供了如下两个方法：

- create([superclass][, methods...])：定义一个新类，该新类继承 superclass 类里的所有方法。其中 methods 是一个 JavaScript 对象，该对象的所有方法将作为新建类的实例方法。如果子类覆盖了父类的实例方法，子类方法依然可以访问父类方法，为了访问父类方法，子类方法的方法签名的第一个形参应该是$super，然后就可使用$super 来代表父类中的同名方法了。
- addMethods(methods)：扩展已有的类，该方法为一个已有的类新增方法，或者覆盖原有的方法。一旦我们为某个类新增方法之后，该类的子类也会继承到该新方法，甚至那些之前创建的实例也会获得该新方法。

下面程序使用 Class 创建一个 Person 类，并为 Person 类定义了一个 Student 子类，程序如下：

程序清单：codes\09\9.4\Class\Class.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//定义一个新类
var Person = Class.create({
    //initialize 方法就是构造器
```

```
    initialize: function(name, age)
    {
        this.name = name;
        this.age = age;
    },
    //定义一个普通方法
    info: function()
    {
        alert(this.name + "的年龄是: " + this.age );
    }
});
//创建 Person 类的实例
var p = new Person('yeeku' , 30);
//调用方法
p.info();
//定义 Student 继承 Person
var Student = Class.create(Person ,
{
    //定义新的构造器，$super 形参代表父类同名方法
    initialize: function($super , name , age , grade)
    {
        $super(name, age);
        this.grade = grade;
    },
    //定义一个普通方法
    study: function()
    {
        //调用从 Person 继承到的 info()方法
        this.info();
        alert("我上 " + this.grade + " 年级");
    }
});
//创建 Student 对象
var s = new Student('wawa' , 8 , 3);
//调用方法
s.study();
</script>
```

执行上面程序，程序中 Student 类继承了 Person 类的 info()方法，由此可见：Prototype 库确实增加了一些“面向对象”的支持。

当使用 Class 对象创建了新类之后，新类里包含如下两个特殊属性：

- superclass：该属性引用该类的直接父类。
- subclasses：该属性的值是一个数组，该数组保存了该类的所有子类。

当使用这些类创建了 JavaScript 实例之后，这些实例都有一个 constructor 属性，该属性引用创建该实例的类。

➤➤9.4.11 两个常用的监听器

Prototype 库提供了两个常用监听器来监听表单以及表单控件，这两个常用监听器如下：

- Form.Observer(form , interval , callback)：如果表单 form 内任何表单控件的值发生改变，interval 秒后自动触发 callback 函数。该表单既然可以是表单的 id 属性，也可以是表单本身。
- Form.Element.Observer(element , interval , callback)：如果表单控件 element 的值发生改变，interval 秒后自动触发 callback 函数。该 element 既可以是表单控件的 id 属性，也可以是表单控件本身。

这两个监听器是专门为 Ajax 交互设计的，通过使用它们可以动态监听表单、表单控件里值的改变。如下程序所示：

程序清单：codes\09\9.4\Observer\Form.Observer.html

```
<body>
<form id="test" method="post" action="#">
用户名：<input type="text" id="user" name="user" />
密码：<input type="text" id="pass" name="pass" />
</form>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//为 test 表单绑定事件监听器
new Form.Observer("test", 1, function()
{
    //此处的 this.getValue()将返回目标表单的 serialize()
    alert(this.getValue());
});
</script>
</body>
```

上面代码为 test 表单注册了一个表单监听器，该监听器的使用了一个匿名函数，该匿名函数将在该表单内任何表单控件的值发生改变后、1 秒后自动触发。触发时，使用 getValue()方法返回该表单内的请求参数和请求值，以查询字符串的形式返回。图 9.15 显示了该表单内表单控件的值发生改变时弹出的对话框。

图 9.15　使用表单监听器

下面代码片段示范了监听表单控件的值改变：

程序清单：codes\09\9.4\Observer\Form.Element.Observer.html

```
<body>
用户名：<input type="text" id="user" name="user" />
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//为 user 表单控件绑定事件监听器
new Form.Element.Observer("user", 1, function()
{
    //此处的 this.getValue()将返回目标表单的 getValue()
    alert(this.getValue());
});
</script>
</body>
```

上面代码为 user 表单控件注册了一个表单监听器，该监听器的使用了一个匿名函数，该匿名函数将在 user 表单控件的值发生改变后、1 秒后自动触发。触发时，使用 getValue()方法返回该表单控件的值。此处只是返回表单控件的值，而不是查询字符串。图 10.16 显示了表单控件的值发生改变时弹出的对话框：

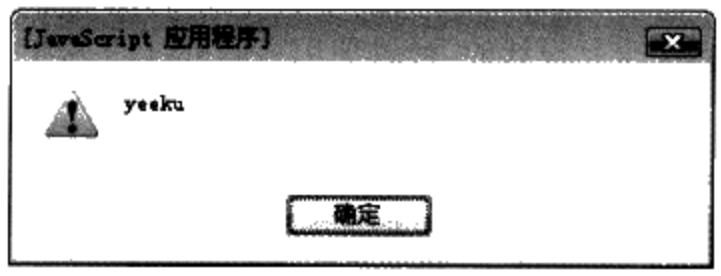


图 9.16　使用表单控件监听器

9.5　Prototype 常用的扩展

Prototype.js 不仅提供了一系列的自定义的类和对象，还对 JavaScript 中原有的类和对象进行扩展。这种扩展为原有的类和对象提供了更多方法，这种方法大大简化了原有对象的使用。下面依次介绍这些常用的扩展。

➤➤9.5.1　扩展 Array

Prototype 库扩展了 JavaScript 中数组，为这些数组提供了更多额外的方法。经 Prototype 扩展后数组都混入了 Enumerable 模块的一些方法。Prototype 为 Array 扩展了如下几个方法：

- clear()：清空该数组，即将数组的元素全部清空。
- clone()：复制一个新数组，复制的新数组和源数组有完全相同的数组元素。
- compact()：压缩数组，返回将源数组中的 null、undefined 等值删除后的新数组。该方法不会影响源数组。
- each(iterator)：遍历每个数组元素，iterator 是一个形如 function(ele , index){}函数，其中 ele、index 将依次等于每个数组元素、数组元素索引值。
- first()：返回数组的第一个元素。
- flatten()：用于将一个多维数组转换为一维数组。这种转换是基于深度优先法则，先将第一个数组中的每个元素都取出(如果第一个数组元素又是一个多维数组，也需要遍历该数组元素)。将所有数组元素取出组成一个一维数组。原数组不受任何影响。
- from(iterable)：该方法复制一个新数组，或者根据一个类似数组的集合来创建数组。该方法其实是$A()函数的别名。
- indexOf(value)：返回数组中某个元素的索引值，如果没有找到该元素，则返回-1。
- inspect()：以字符串形式返回每个数组元素，该方法类似 Java Arrays 提供的 toString()方法。
- last()：返回数组的最后一个元素。
- reduce()：简化数组。包含一个数组元素的数组将返回该数组元素；包含多个数组元素的数组将保持不变。
- reverse(true|false)：返回源数组每个元素反转后的数组。如果如果没有指定参数，或者参数为 true，则源数组也被反转。如果给出 false 参数，则不会反转源数组。
- size()：返回数组的长度，该方法从 Enumerable 混入，该方法返回值和数组原有的 length 属性返回值相同。
- uniq()：消除数组里重复的数组元素，如果数组里有多个数组元素完全相同，调用该方法后，这些数组元素将只剩下第一个元素。
- without(value1 [, value2 [, .. valueN]])：将 value1、value2...valueN 等元素从数组中删除，返回删除了这些目标元素后的新数组。

上面这些方法都比较简单，下面通过代码来示范 each、flatten()方法的用法：

程序清单：codes\09\9.5\Array\Array.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//定义第一个一维数组
var a = ['疯狂Java讲义','轻量级Java EE企业应用实战'];
//定义第二个一维数组
var b = ['wawa','anno'];
//将两个一维数组组成一个二维数组
var c = [a, b];
c.each(function(ele , index)
{
    document.writeln("第" + index + "个元素是：" + ele + "<br />");
});
//以一个二维数组和一个普通值形成三维数组
var d = [c , 'china']
//输出三维数组的长度
document.writeln("d数组的长度是：" + d.length + "<br />");
//将三维数组"压扁"成一维数组。
var e = d.flatten();
//输出一维数组后的长度（看到5）和数组元素
document.writeln(e.size() + e);
</script>
```

程序输出 d 数组的长度时将看到 2，这是正确的。因为该数组包含两个元素，第一个元素是个二维数组，第二个元素是一个普通值。

第二次输出 e 数组长度将看到 5，e 数组由 d 数组 flatten 而来，就是将一个三维数组“压扁”成一维数组，所以我们看到 e 数组长度为 5。在浏览器中浏览的长度和元素看到如图 9.17 所示的界面。

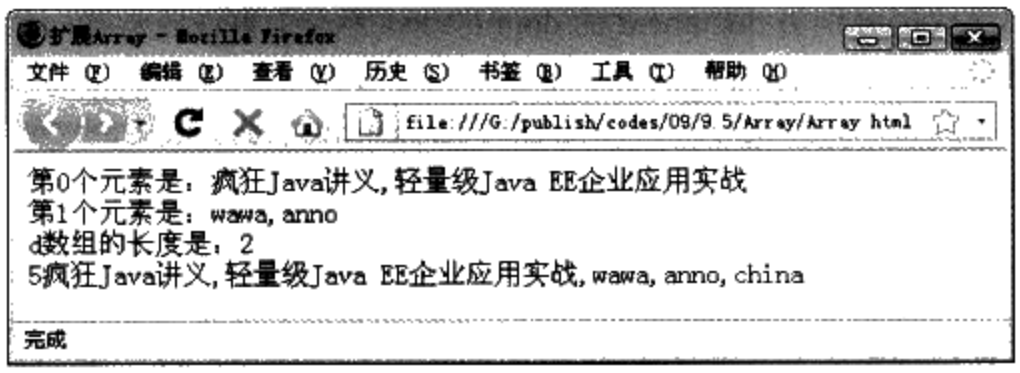


图 9.17 将三维数组“压扁”成一维数组

通过图 9.17 可看出 flatten()方法的转换过程，首先将 d 数组的第一个元素是 c，c 又是一个二维数组。根据深度优先法则，先一直找到 c 数组的第一个数组元素的第一个基本值"疯狂 Java 讲义"，然后是"轻量级 Java EE 企业应用实战"。然后再处理 c 数组第二个元素，即 b 数组，b 数组的两个值依次为 wawa、anno。最后的输出结果就形成了图 10.16 中所示的疯狂 Java 讲义,轻量级 Java EE 企业应用实战,wawa,anno,china。

➤➤9.5.2 扩展 document

Prototype 库扩展了原有的 document，扩展后的 document 新增了如下三个方法：

- fire(eventName[, memo])：用于在代码中为 document 触发人工合成的 eventName 事件。如果指定了第二个 memo 参数，事件处理函数可通过 event.memo 来访问该参数。该方法与 Element#fire()方法的功能相同，只是触发事件的事件源不同而已。
- observe(eventName, handler)：为 document 的 eventName 事件绑定监听器：handler。该方法与 Element#observe ()方法的功能相同，只是绑定的事件源不同而已。。
- stopObserving(eventName, handler[, useCapture = false])：取消为 document 的 eventName 绑定

的 handler 事件处理器。类似地，该方法的功能也类似于 Element 的 stopObserving()方法。

下面程序示范了使用 document 扩展后的方法：

程序清单：codes\09\9.5\document\document.html

```
<body>
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//为 document 的 loaded 事件绑定事件监听器
document.observe("dom:loaded" , function(event)
{
    $("show").innerHTML += ("页面装载完成 <br />");
});
//为 document 的 a:b 事件绑定事件监听器
document.observe("a:b" , function(event)
{
    $("show").innerHTML += ("myEvent 被触发了 <br />");
    //访问 event.memo.book 属性
    $("show").innerHTML += ("event.memo.book 属性值为："
        + event.memo.book);
});
</script>
<!-- 单击该按钮时触发 document 上的 a:b 事件 -->
<input type="button" value="单击我"
    onclick='document.fire("a:b" , {book:"疯狂 Ajax 讲义"});' />
<div id="show"></div>
</body>
```

上面程序中粗体字代码分别为 document 的 dom:loaded 绑定了事件监听器，还为 a:b 事件绑定了事件监听器，其中 dom:loaded 代表页面装载完成的事件，而 a:b 则完全是开发者自定义的“人工合成”事件。单击页面中“单击我”的按钮时，程序使用 document.fire()方法触发 a:b 事件——将会看到绑定的事件处理函数被执行。

注意：

使用 document、Element 的 observe()方法为“人工合成”事件绑定事件监听器时，该“人工合成”事件的事件名必须满足 xx:xx 的格式。document 的 dom:loaded 事件即代表页面装载完成事件。

9.5.3 扩展 String

扩展后的 String 对象包含了更多方法，扩展后的 String 用起来更加方便，扩展后的 String 增加了如下常用的方法：

- blank()：判断某个字符串是否为空。如果该字符串不包含任何字符，或只包含空白，则该方法返回 true，否则返回 false。
- camelize()：将以中划线分隔的字符串转换为“驼峰写法”。它可将 background-color 字符串转换成 backgroundColor 的形式，读者应该清楚该方法作用了，对了，在将 CSS 样式中的属性转换为脚本中的 CSS 属性时特别有用。
- capitalize()：将指定字符串的首字母大写，其他字母小写的方法。
- dasherize()：将字符串中的下划线（_）替换成中划线（-）。该方法常常和 underscore()方法结合使用。
- empty()：判断某个字符串是否严格为空。只有当该字符串不包含任何字符（连空白也不行）时该方法返回 true。

- endsWith(substring)：判断该字符串是否以 substring 结尾。
- escapeHTML()：返回将 HTML 字符转义后的字符串，即将小于符号（<）转换成<等。该方法不会影响源字符串。
- evalScripts()：执行在字符串中找到的所有脚本片段，即执行字符串中的<script.../>元素中脚本。该方法返回多个脚本返回值组成的数组。
- extractScripts()：返回字符串中所有的脚本片段，如果有多个脚本片段，以数组形式返回。该方法不会影响源字符串。
- gsub(pattern, replacement)：将字符串中匹配 pattern 正则表达式的部分替换成 replacement，replacement 既可是一个简单字符串，也可是一个形如 function(match){}的函数，如果是后者，该函数在替换 pattern 部分时，replacement 函数可以对匹配的子串进行适当处理。
- include(substring)：判字符串里是否包含 substring 子字符串。
- interpolate(object[, pattern])：以该字符串作为模板，并用 object 里对应的属性去填充该模板字符串，返回填充完成的字符串。
- scan(pattern, iterator)：扫描字符串中匹配 pattern 模式的子串，并将这些子串依次传入 iterator 函数。iterator 函数是一个形如 function(match){}的函数。
- startsWith(substring)：判断字符串是否以 substring 开头。
- strip()：删除字符串前后的空白。
- stripScripts()：返回删除了所有的脚本的新字符串，即删除<script.../>元素里的全部内容。该方法不会影响源字符串。
- stripTags()：返回删除了 HTML 和 XML 标记的新字符串。该方法不会影响源字符串。
- sub(pattern, replacement[, count = 1])：将字符串中匹配 pattern 正则表达式的部分替换成 replacement，replacement 既可是一个简单字符串，也可是一个形如 function(match){}的函数，如果是后者，该函数在替换 pattern 部分时，replacement 函数会进行适当处理。该函数与 gsub()的差别在于 count 参数，count 控制替换几次，而 gsub 的 g 是 global 的意思，也就是全部替换。
- succ()：返回该字符串的后一个字符串，例如"a".succ()返回"b"，"aa".succ()返回"ab"。
- times(count)：返回该字符串重复 count 后的新字符串。
- toArray()：将字符串转换成字符数组，每个字符都是一个数组元素。
- toQueryParams([separator = '&'])：把查询字符串转换成一个 JavaScript 对象。该方法还有一个别名：parseQuery。
- truncate([length = 30[, suffix = '...']])：用于将字符串截断到 length 个字符，并在后面追加 suffix 后缀。

提 示：

该函数在 HTML 页面上很常用，例如我们需要在某个表格内显示一段文本，如果这段文本太长无法显示，我们常常会将其截断到指定长度，并在后面追加省略号（...）。在这种需求下，使用 truncate()方法就可以一次满足该需求。

- underscore()：将“驼峰写法”的字符串转换为以下划线分割的字符串，例如"abcXyzGhi".underscore()将返回 abc_xyz_ghi"。该方法对源字符串没有影响。
- unescapeHTML()：escapeHTML()的反转。即将<等转换成小于符号（<）。该方法不会影响源字符串。

下面程序示范了 String 类新增方法用法：

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
```

```
    //用于示范删除 XML 和 HTML 标记
    var str1 ='<a>dfd</a>';
    //输出结果为 dfd
    alert("'<a>dfd</a>'.stripTags()的结果为："
        + str1.stripTags());
    //用于示范删除脚本
    var str2 ='<script>dfd<\/script>';
    //删除脚本后字符串为空字符串
    alert("'<script>dfd<\/script>'.stripScripts()的结果为："
        + str2.stripScripts());
    //示范将 HTML 标记转义
    var str3 ='<a>dfd</a>';
    //输出结果为&lt;a&gt;dfd&lt;/a&gt;
    alert("'<a>dfd</a>'.escapeHTML()的结果为："
        + str3.unescapeHTML());
    //示范反转义 HTML 标记
    var str4 ='&lt;a&gt;dfd&lt;/a&gt;';
    //输出结果为<a>dfd</a>
    alert("'&lt;a&gt;dfd&lt;/a&gt;'.unescapeHTML()的结果为："
        + str4.unescapeHTML());
    //示范执行字符串中的脚本
    var str5 ='<script>alert("====");<\/script>';
    //弹出警告对话框
    str5.evalScripts();
    //示范取得字符串中的脚本
    var str6 ='<script>alert("====");<\/script>'
        + '<script>alert("xxxx");<\/script>';
    //输出一个长度为 2 的字符串数组。
    alert(str6.extractScripts());
    var str7 = 'abc-xyz-ghi';
    var str8 = str7.camelize();
    alert("'abc-xyz-ghi'.camelize()的值为：" + str8);
    var str9 = str8.underscore().dasherize();
    //再次恢复'abc-xyz-ghi';
    alert(str9);
    var book = '轻量级 Java EE 企业应用实战';
    alert("'轻量级 Java EE 企业应用实战'.truncate(8 , '...')的值："
        + book.truncate(10 , '...'));
    var str10 = 'java struts hibernate';
    //用逗号（,）替换空白，将输出 java,struts,hibernate
    alert(str10.gsub('\\s+' , ','));
    //使用自定义替换，将所有单词首字母大写，并在单词后添加逗号（,）
    //将输出 Java, Struts, Hibernate
    alert(str10.gsub('\\w+' , function(match)
    {
        return match[0].capitalize() + ",";
    }));
    //将依次输出 java,struts,hibernate
    str10.scan("\\w+" , function(match)
    {
        alert(match);
    });
</script>
```

上面代码已经给出了各方法的操作结果，读者可直接看出各方法功能，此处不再赘述。

➤➤9.5.4 扩展 Function

Prototype 扩展了 Function 类，因为 JavaScript 里函数的复杂性，JavaScript 为函数增加了 bind()方法，从而可以将该函数限定在调用对象内执行。扩展后的 Function 主要增加了如下几个方法：

- argumentNames()：返回该函数所有形参组成的数组。
- bind(thisObj[, arg...])：将指定函数包装成另一个函数，从而将函数的执行范围限制在 thisObj 范围内——即函数内的 this 关键字总是引用 thisObject。
- bindAsEventListener(thisObj[, arg...])：这是 bind()方法的事件监听器版本，它可保证该函数被执行时候第一个参数可引用到触发该函数的事件。
- curry(arg...)：为该函数的部分形参传入值，从而得到部分形参已被赋值后的新函数（从而允许调用该新函数时无需为这些参数指定值）。
- defer(arg...)：调度该函数尽快执行。arg 参数将作为参数传给被调度的函数。因为该函数底层依赖于是 window.setTimeout()方法，故该方法返回执行 setTimeout()方法得到的定时器，从而允许调用 clearTimeout()方法来清除该定时器。
- delay(seconds[, arg...])：调度该函数于 second 秒后执行。arg 参数将作为参数传给被调度的函数。该方法与 defer()方法的用法和功能基本一样，只是该函数可以指定延迟 second 后执行。
- methodize()：将该函数包装到另外一个函数里，使得调用新函数时自动将 this 作为第一个参数传入原有的函数。
- wrap(wrapperFunction[, arg...])：返回对原函数进行包装后的新函数。

下面程序示范了 Function 扩展中较为难以理解的一些方法，程序如下：

程序清单：codes\09\9.5\Function\Function.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//定义一个 JavaScript 对象
var person = {
    name: 'yeeku',
    //虽然 info()函数定义在 person 对象里，它依然是一个独立的函数
    info: function()
    {
        alert(this.name);
    }
};
//看到输出'yeeku'
person.info();
//再顶一个 runFn 函数，该函数专门用于运行指定函数
function runFn(f)
{
    f();
}
window.name = '疯狂 Java 讲义';
//该函数所在范围是 window,所以 this.name 为'疯狂 Java 讲义'
runFn(person.info);
//下面代码先将 person.info 函数绑定给 person。
//这样无论以何种方式执行该函数，person.info 函数的 this 总是引用 person。
runFn(person.info.bind(person));
function test(name , age)
{
    alert("此人名为：" + name + "\n"
        + "年龄为：" + age);
}
//将 test()函数包转成 newTest 函数，test()函数的第一个参数已经有值
var newTest = test.curry('yeeku');
//将输出"此人名为：yeeku\n 年龄为：31
newTest(31);
//定义一个函数，函数 targe
var fn1 = function(target , name)
```

```
{
    target.name = name;
};
//执行 fn1.methodize();产生一个新函数
var fn2 = fn1.methodize();
//程序将 this（代表 window）传入 fn1。
fn2("疯狂 XML 讲义");
//输出"疯狂 XML 讲义"
alert(name);
</script>
```

上面程序中粗体字代码示范了 Function 所扩展的 bind()、curry()和 methodize()三个方法，使用这三方法后函数效果也在代码中有说明，此处不再赘述。

➤➤9.5.5 扩展 Number

Prototype 为 Number 扩展了一些新的工具函数，从而使得 JavaScript 中的数值变量、常量都可直接调用这些函数，从而极大了简化了 JavaScript 编程。Prototype 库为 Number 扩展的函数如下：

- abs()：求绝对值。
- ceil()：返回比当前数值大的最小整数。
- floor()：返回比当前数值小的最大整数。
- round()：四舍五入取整。
- succ()：取得当前数值+1 后的数值。
- times(iterator)：迭代器函数，iterator 是一个形如 function(num){}的函数，其中 num 将依次等于 0、1、2...一直到当前数值。
- toColorPart()：把当前数值转换为 16 进制数值的字符串形式。
- toPaddedString(length[, radix])：将数值转换为指定位数的字符串，其中 radix 还可指定该数值所使用的进制。

下面程序示范了如何使用 Prototype 为 Number 扩展的方法：

程序清单：codes\09\9.5\Number\Number.html

```
<script src="../../prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//下面输出 4。
alert((-4).abs());
//下面输出-2。
alert((-2.3).ceil());
//下面输出-2。
alert((-1.4).floor());
//下面输出 4。
alert((4.4).round());
//下面输出 2.2。
alert((1.2).succ());
//下面将依次输出 0、1、2
(3).times(function(num)
{
    alert(num);
});
//下面将输出'13'
alert((19).toColorPart());
//下面将输出'000022'
alert((12).toPaddedString(6 , 5));
</script>
```

上面程序中各粗体字代码都非常简单，只是 times()方法略微复杂，其实 times()方法是一个 Ruby 风格的迭代器，它会多次重复调用传给 times()方法的函数，函数的参数将自动从 0 一直迭代到当前数值。

9.6 Prototype 的 Ajax 支持

前面已经花了大量篇幅来介绍 Prototype 库作为 JavaScript 函数库的功能。事实上，Prototype 库对 JavaScript 提供那些扩展也是为了更好地简化 Ajax 应用的开发，除此之外，Prototype 库还为 Ajax 应用的开发提供了下面这些类。

➤➤9.6.1 使用 Ajax.Request 类

这个类对于简化 Ajax 应用的开发是相当重要的，也许在前面介绍 Ajax 应用时，已经有读者觉得笔者的代码写得相当繁琐、并且不断地违背 DRY（不要重复书写代码）的原则：总是不厌其烦地创建 XMLHttpRequest 对象，打开与服务器的连接，指定回调函数，然后发送请求参数……如此繁琐的过程，每次需要进行 Ajax 交互时都需要重复书写这段代码，的确是相当让人郁闷的事情。

借助于 Ajax.Request 类可以省却这些繁琐的过程，无需创建 XMLHttpRequest 对象，无需手动发送请求，只要简单地创建一个 Ajax.Request 对象，就可以完全异步请求的发送。Ajax.Request 类包含如下构造器：

Ajax.Request(URL , options)：创建一个 Ajax.Request 对象，对应于发送一次请求。参数 URL 是异步请求的地址。options 参数指定发送请求的各种详细选项，该对象支持如下几个通用属性：

- asynchronous：是否异步发送请求，该参数默认是 true。通常不应该改变该属性值。
- contentType：设置 Content-Type 请求头。
- encoding：指定请求内容编码所使用的字符集。该属性值默认是'UTF-8'。该属性通常无需改变。
- method：指定发送请求的方式。只能是 post 和 get，不能大写。
- parameters：指定请求参数。应采用 name1=value1&name2&value2 的形式。如果采用'get'方式发送请求，这些请求参数将被追加到 URL 字符串之后；如果采用其他方式发送请求，这些请求参数将被放入请求体中。
- evalJS：是否处理服务器响应里包含的 JavaScript 代码。该属性默认是 true。
- evalJSON：是否将服务器响应转换成 JSON 格式对象。该属性默认是 true。

大部分时候，我们只需要使用 Ajax.Request 的构造器就可以完成一次 Ajax 交互。下面以简单的代码来示范如何利用 Ajax.Request 完成交互。

下面代码示范了一个简单输入提示，当用户在单行文本框内输入相应的字符时，文本框的下面将出现相应的提示，该应用基于 Ajax.Request 对象完成。

下面是服务器页面代码（因为主要是示范 Ajax.Request 对象的用法，因此并未在服务器端并未来进行严格的分层），服务器页面以 JSP 页面充当，下面是服务器 JSP 页面代码：

程序清单：codes\09\9.6\Ajax.Request\tips.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
    //获取请求参数 favorite
    String hdchar = request.getParameter("favorite");
    System.out.println(hdchar);
    //如果请求参数是 apple 的前几个字符，则输出 apple
    if ("apple".startsWith(hdchar))
    {
        out.println("apple");
    }
    //如果请求参数是 banana 的前几个字符，则输出 banana
```

```
    else if("banana".startsWith(hdchar))
    {
        out.println("banana");
    }
    //如果请求参数是 peach 的前几个字符，则输出 peach
    else if("peach".startsWith(hdchar))
    {
        out.println("peach");
    }
    //否则将输出 other fruit
    else
    {
        out.println("other fruit");
    }
%>
```

这个页面代码是比较简单，它根据请求参数来匹配三个水果的名称，如果请求参数是三种水果名称的前面部分，则输出水果名，否则将输出 other fruit。

客户端的 HTML 页面则借助于 Ajax.Request 对象发送请求，下面是客户端 HTML 页面的代码：

程序清单：codes\09\9.6\Ajax.Request\index.html

```
<body>
<h3>请输入您喜欢的水果</h3>
<div style="width:280px;font-size:9pt">
输入 apple、banana、peach 可看到明显效果:</div>
<br />
<input id="favorite" name="favorite" type="text"
    onblur="$('tips').hide();"/>
<div id="tips" style="width:160px;border:1px solid menu;
    background-color:#ffffcc;display:none"></div>
<script src="prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
    //监控目标文本框输入文字发生改变的函数
    function searchFruit()
    {
        //请求的地址
        var url = 'tips.jsp';
        //将 favorite 表单域的值转换为请求参数
        var params = Form.Element.serialize('favorite');
        //创建 Ajax.Request 对象，对应于发送请求
        var myAjax = new Ajax.Request(
        url,
        {
            //请求方式：POST
            method:'post',
            //请求参数
            parameters: params,
            //指定回调函数
            onComplete: showResponse,
            //是否异步发送请求
            asynchronous:true
        });
    }
    //定义回调函数
    function showResponse(request)
    {
        //在提示 tip 元素中输出服务器的响应
        $('tips').innerHTML = request.responseText;
        //显示提示 tip 对象
```

```
            $('tips').show();
        }
        //为 favorite 表单域绑定事件处理函数
        new Form.Element.Observer("favorite", 1, searchFruit);
    </script>
    </body>
```

在上面代码中看到，使用 Ajax.Request 发送请求是如此简单，只需要创建一个 Ajax.Request 对象即可。创建该对象时，应指定发送请求的 URL。除此之外，options 对象可指定发送请求的相关选项：请求参数、是否异步、请求的发送方式和回调函数。

创建 Ajax.Request 对象直接使用它的构造器 Ajax.Request(URL , options)即可，其中 options 是一个匿名对象，该对象封装了发送请求的选项。 一旦创建了 Ajax.Request 对象，即完成了 Ajax 交互。

注意：

在 Ajax.Request 的 options 参数的 method 属性值不能是 GET，也不能是 POST。只能是 post 和 get。

上面的 HTML 页面中还使用了 Form.Element.Observer 类，用于为表单域的值改变时绑定事件处理函数。在浏览器中浏览该 HTML 页面，并在文本框中输入 a，将看到如图 9.18 所示的页面。

图 9.18 输入提示的效果

通过前面的介绍，已经知道 XMLHttpRequest 对象在发送请求的过程中的 readystate 属性有四个状态，Ajax 交互可以为这四个状态分别指定回调函数，就像上面程序的 options 参数指定了 onComplete 属性，该属性用于为 Ajax 交互指定回调函数。

类似于 onComplete 属性，options 对象一共可指定如下几种回调函数：

- onCreate：当 Ajax.Request 对象初始化完成后触发该属性指定的回调函数。
- onComplete：当 Ajax 请求交互完成后触发该属性指定的回调函数。
- onException：当 Ajax 交互过程中出现错误时触发该属性指定的回调函数。
- onFailure：当 Ajax 交互过程中完成，并返回了 statusCode，但 statusCode 的值不是 2xy 系列时将触发该属性指定的回调函数，该函数将会在 onComplete 属性指定的回调函数之前被触发。
- onInteractive：当 request 对象收到服务器部分响应，并未完全收到所有响应时触发该属性指定的回调函数。该属性指定的回调函数不能保证在合适的时候被触发。
- onLoaded：当底层 XMLHttpRequest 对象创建完成，连接被打开，并准备发送实际请求之时将触发该属性执行的回调函数。该属性指定的回调函数不能保证在合适的时候被触发。
- onLoading：当底层 XMLHttpRequest 对象创建完成，连接被打开后将触发该属性执行的回调函数。该属性指定的回调函数不能保证在合适的时候被触发。
- onSuccess：当 Ajax 交互过程中完成，并返回了 statusCode，且 statusCode 的值是 2xy 系列时将触发该属性指定的回调函数，该函数将会在 onComplete 属性指定的回调函数之前被触发。
- onUninitialized：当 XMLHttpRequest 刚刚创建时触发该属性指定的回调函数。该属性指定的

回调函数不能保证在合适的时候被触发。

- onXYZ：当 Ajax 交互过程中完成，并返回了 statusCode，且 statusCode 的值是 xyz 时触发该属性指定的回调函数，该函数将会在 onComplete 属性指定的回调函数之前被触发。一旦通过该属性指定了回调函数，onSuccess 和 onFailure 指定的回调函数将全部失效。

上面各属性的值大都是一个形如：function(response){}的函数，其中 response 代表服务器响应，它是一个 Ajax.Response 对象，通过该对象可取得服务器响应头和响应数据等。

提 示：

Ajax.Response 类由 Prototype 库提供，Prototype 库对 XmlHttpRequest 进行了简单包装，包装后就得到了这个 Ajax.Response 类。

Ajax.Response 类里包含如下常用的属性：

- status：该属性返回服务器响应的状态码。如 200、404 等。
- statusText：该属性返回服务器响应的状态字符串，该属性值等于 XMLHttpRequest.statusText。
- readyState：该属性值等于 XMLHttpRequest.readyState。
- responseText：该属性值等于 XMLHttpRequest.responseText。
- responseXML：如果服务器响应的 content-type 设置为 application/xml，则返回服务器响应的 XML 响应体。否则返回 null。
- responseJSON：如果服务器响应的 content-type 设置为 application/json，该属性获得服务器响应对应的 JSON 对象或数组。否则返回 null。
- headerJSON：如果 X-JSON 格式的相应头存在，则返回 JSON 对象或数组。否则返回 null。
- request：发送 Ajax 请求的对象。（就是 Ajax.Request 或 Ajax.Updater 的实例）。
- transport：该属性获取原生的 XmlHttpRequest 对象本身。

除此之外，还包含如下常用方法：

- getHeader(name)：获取 name 请求头的值，如果该请求头不存在则返回 null。 String or null Returns the value of the requested header if present. null otherwise. Does not throw errors on undefined headers like it's native counterpart does.
- getAllHeaders()：返回所有请求头里 name 和 value 连缀而成的字符串。
- getResponseHeader(name)：获取 name 响应头的值，如果该响应头不存在则返回 null。
- getAllResponseHeaders()：返回所有响应头里 name 和 value 连缀而成的字符串。

9.6.2 利用 Form.request 方法

Form.request()方法实际上是 Ajax.Request 的封装，该方法会将该表单所有表单控件转换为请求参数，默认向该表单 action 指定的 URL 发送异步请求。Form.request()方法完整的语法格式是：

Form.request([options])：直接以该表单内的所有表单控件作为参数发送请求。其中 options 参数用于指定额外的参数，options 对象可指定如下 2 个常用属性：

- method：该属性指定发送异步请求的方式，可以指定'get'或'post'。该属性值默认是'post'。
- parameters：该属性值可以一个 JavaScript 对象，用于添加额外的请求参数。

除此之外，options 对象里包含的各种属性名、属性值将被原封不动地传给 Ajax.Request 的 options 属性，因此一样可以指定 onComplete、onLoading 等属性。

下面以一个简单的登录应用来介绍 Form.request()方法的用法，这个登录应用的服务器处理页面代码如下：

程序清单：codes\09\9.6\Form.request\login.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
```

```
    //获取请求参数 user
    String user = request.getParameter("user");
    String pass = request.getParameter("pass");
    //实际应用中此处可调用业务组件的业务方法。但此处仅仅示范 Ajax 交互,
    //所以直接要求 user 为 yeeku, pass 为 123456 才可正常登录。
    if(user.equals("yeeku")
        && pass.equals("123456"))
    {
        out.println("登录成功! ");
    }
    else
    {
        out.println("登录失败! ");
    }
%>
```

该应用所使用的登录页面只需获取登录表单，并调用表单的 request()方法发送请求即可，下面是登录应用所使用的 HTML 页面代码：

程序清单：codes\09\9.6\Form.request\index.html

```
<body>
<h3>请输入用户名、密码登录</h3>
<form id="login" action="login.jsp" method="post">
用户名: <input name="user" type="text" /><br />
密码: <input name="pass" type="text" onclick="login();"/><br />
<input value="登录" type="button" onclick="login();"/>
</form>
<script src="prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//处理登录函数
function login()
{
    //使用 Form 的 request 发送异步请求
    $("login").request(
    {
        //指定回调函数
        onComplete: function(request)
        {
            alert(request.responseText);
        }
    });
}
</script>
</body>
```

上面代码中粗体字代码使用 Form.request()方法发送了异步请求，并指定当服务器响应完成时输出服务器响应。部署该应用，在浏览器中浏览该页面，并输入合适的用户名、密码后单击“登录”按钮，页面效果如图 9.19 所示效果。

图 9.19　使用 Form.request 发送异步请求

➤➤9.6.3 使用 Ajax.Responders 对象

这个对象用于注册 Ajax 事件监听器，该对象注册的 Ajax 事件监听器不管是哪个 XMLHttpRequest 在发生交互，都能被自动触发。因而，Ajax.Responders 注册的事件监听器是全局有效的，可用于监控整个 Ajax 的交互过程。Ajax.Responders 对象包含如下两个常用方法：

- register(handler)：注册一个全局 Ajax 事件处理器。该事件处理器是一个 JavaScript 对象，该对象指定监控 Ajax 交互的回调方法，如 onCreate,onComplete,onException 等。这些属性指定的函数将在相应事件发生时被触发。
- unregister(handler)：删除一个已经注册的 Ajax 事件处理器。

对前面 Ajax.Request 应用简单修改，在 HTML 页面中使用 Ajax.Responders 注册一个全局的 Ajax 事件处理器。增加的 JavaScript 代码如下：

程序清单：codes\09\9.6\Ajax.Responders\index.html

```
//定义 Ajax 的全局事件处理器
var myGlobalHandlers =
{
    //刚刚开始 Ajax 交互时触发该属性指定的函数。
    onCreate: function()
    {
        $('Loadingimg').show();
    },
    //交互失败时触发该属性指定的函数。
    onFailure:function()
    {
        alert('对不起!\n 页面加载出错!');
    },
    //交互成功时触发该属性指定的函数。
    onComplete: function()
    {
        if(Ajax.activeRequestCount == 0)
        {
            $('Loadingimg').hide();
        }
    }
};
//为 Ajax 事件绑定全局的事件处理器
Ajax.Responders.register(myGlobalHandlers);
```

上面代码为全局的 Ajax 事件注册了一个处理器，该处理器分别指定了三个回调函数，分别在刚开始 Ajax 交互时激发、Ajax 交互失败时激发和 Ajax 交互成功时触发。在 Ajax 交互完成时，还判断了 Ajax 对象的 activeRequestCount 属性，该属性为 0 时表明所有 Ajax 交互完成。Ajax.Responders 调用 register()方法注册了全局的事件处理器。

修改 Ajax.Request 应用的 HTML 页面，在单行文本框的后面添加 id 为 Loadingimg 的图片，该图片会在 Ajax 交互开始时显示，一旦 Ajax 交互结束该图片会自动隐藏。向服务器发送异步请求后，服务器响应没有完成时，页面效果如图 9.20 所示。

图 9.20 增加了 load 图片的效果

➤➤9.6.4 使用 Ajax 对象

Ajax 对象也是个全局对象，该对象可获取正处于 Ajax 请求的个数，也可以用于创建一个新的 XMLHttpRequest 对象。该对象具有常用属性一个，常用方法一个。Ajax 对象常用属性如下：

- activeRequestCount：处于 Ajax 交互过程中的 XMLHttpRequest 对象的个数。

在 9.6.3 节中已经使用了 activeRequestCount 属性，当该属性的值为 0 时，表明已经没有正处于交互的 XMLHttpRequest 对象。

Ajax 对象包含一个常用方法：

- getTransport()：该方法返回一个新的 XMLHttpRequest 对象。

➤➤9.6.5 使用 Ajax.Updater 类

这个类是对 Ajax.Request 类的简化，使用该类无需使用回调函数，该类可自动将服务器器响应显示在某个容器中。当服务器响应完成时，客户端 HTML 页面无需使用回调函数解析服务器响应，从而可以更进一步简化 Ajax 交互编程。

Ajax.Updater 类进一步简化了 Ajax 交互的过程。与 Ajax.Request 类似，每创建一个 Ajax.Updater 对象，就可完成一次的 Ajax 交互。但由于 Ajax.Updater 直接使用服务器响应来更新页面容器，因此创建 Ajax.Updater 时需要额外指定一个 containers 参数。Ajax.Updater 的构造函数如下：

- Ajax.Updater(containers , url , options)：创建一个 Ajax.Updater 对象，完成一次 Ajax 交互。其中 url 参数、options 参数与 Ajax.Request 对应构造参数的意义基本相同。containers 参数值有如下两种情形：
- 简单 HTML 元素：使用该 HTML 元素来显示服务器响应。
- JavaScript 对象：该对象是一个形如{complete:element,failure: element...}的对象。用于指定服务器响应完成、出错时使用各自的 HTML 元素来显示对应的提示信息。

与 Ajax.Request 对象相比，Ajax.Updater 的 options 参数额外多了如下两个属性：

- evalScripts：该属性指定是否检测、并执行服务器响应中的<script.../>元素。如果将该属性指定为 ture，则可在服务器响应中使用 JavaScript 脚本。该属性默认值是 false。
- insertion：默认情况下，Ajax.Updater 将会用服务器响应填充指定的 HTML 元素（也就是将目标元素的 innerHTML 设置为服务器相应数据）。为了改变这种默认行为，我们可以为 insertion 指定'top'、'bottom'、'before'和'after'四个字符串，分别代表将服务器响应插入目标元素顶端、尾部、前面、后面等四种情形。

注意：

因为 Ajax.Updater 类的构造参数 options 一样可以指定 onLoading、onLoaded、onInteractive 和 onComplete 等属性，所以 Ajax.Updater 一样可以指定回调函数。

下面的 HTML 代码是对前面输入提示的 HTML 页面的简单修改，修改后的输入提示 HTML 页面使用 Ajax.Updater 作为请求发送类。下面是 HTML 页面的代码：

程序清单：codes\09\9.6\Ajax.Updater\index.html

```
<body>
<h3>请输入您喜欢的水果</h3>
<div style="width:280px;font-size:9pt">
    输入 apple、banana、peach 可看到明显效果:</div>
<br />
<input id="favorite" name="favorite" type="text"
    onblur="$('tips').hide();"/>
<img id="Loadingimg" src="img/indicator.gif" style="display:none"/>
<div id="tips" style="width:160px;border:1px solid menu;
```

```
    background-color:#ffffcc;display:none;"></div>
<script src="prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//发送请求的函数
function searchFruit()
{
    //定义发送请求的服务器地址
    var url = 'tips.jsp';
    //取得请求参数字符串
    var params = $('favorite').serialize();
    //创建 Ajax.Updater 对象，对应于发送一次请求
    var myAjax = new Ajax.Updater(
        //指定 tips 作为服务器响应的容器
        'tips',
        url,
        {
        //发送请求的方法
        method: 'post',
        //请求参数
        parameters: params,
        //指定 Ajax 交互结束后的回调函数，匿名函数——显示 id 为 tips 的元素
        onComplete:function()
        {
            $('tips').show();
        }
    });
}
//为表单域绑定事件监听器
new Form.Element.Observer("favorite", 1, searchFruit);
//定义 Ajax 事件的全局处理器
var myGlobalHandlers =
{
    //当开始 Ajax 交互时，激发该函数
    onCreate: function()
    {
        $('Loadingimg').show();
    },
    //当 Ajax 交互失败后，激发该函数。
    onFailure:function()
    {
        alert('对不起!\n 页面加载出错!');
    },
    //当 Ajax 交互结束后，激发该函数。
    onComplete: function()
    {
        //如果所有 Ajax 交互都已完成，隐藏 Loadingimg 对象
        if(Ajax.activeRequestCount == 0)
        {
            $('Loadingimg').hide();
        }
    }
};
//为 Ajax 事件绑定全局事件处理器
Ajax.Responders.register(myGlobalHandlers);
</script>
</body>
```

在上面代码中，并没有直接修改 tips 元素的内容。只是在构造 Ajax.Updater 对象时，传入了 tips 参数：这表明 tips 元素会自动显示服务器响应，不管服务器是成功的响应，还是失败的响应。响应内容会自动显示在 tips 元素中。

除此之外，也可以采用如下方式来创建 Ajax.Updater 对象：

```
var myAjax = new Ajax.Updater(
//指定使用不同 HTML 元素分开显示成功的响应和失败的响应
{
    success:'tips',
    failure:'failure'
},
url,
{
    method: 'post',
    parameters: params,
    onLoading:function()
    {
        ...
    }
    onLoaded:function()
    {
        ...
        }
    //指定多个回调函数
    ...
});
```

上面代码创建 Ajax.Updater 对象时，依然是传入了三个构造参数。第一个构造参数是

```
{
    success:'tips',
    failure:'failure'
},
```

该参数表明，当服务器成功响应时，服务器响应在 id 为 tips 的元素中显示出来；当服务器响应失败时，服务器响应在 id 为 failure 的元素中显示出来。

下面示范一个服务器响应带 JavaScript 脚本的应用。服务器页面生成带脚本片段的代码，服务器页面代码如下。

程序清单：codes\09\9.6\Ajax.Updater2\a.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<script type="text/javascript">
    alert('服务器的警告对话框。');
</script>
疯狂 Java
```

上面服务器页面中会生成一段 JavaScript 脚本，这段脚本将作为 Ajax 响应被发送到客户端页面，客户端 HTML 页面使用一个按钮来发送 Ajax 请求，客户端的 HTML 页面代码如下：

程序清单：codes\09\9.6\Ajax.Updater2\index.html

```
<h3>请单击下面按钮</h3>
<!-- 下面按钮将发送 Ajax 请求 -->
<input type="button" value="发送" onclick="send();"/>
<div id="show"></div>
<script src="prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
function send()
{
    var url = 'a.jsp';
    var myAjax = new Ajax.Updater(
    //服务器响应在 show 元素中显示
    'show',
    url,
    {
```

```
            method: 'post',
            //请求参数为空
            parameters: null,
            //执行服务器响应中的脚本
            evalScripts:true
        });
    }
    </script>
    </body>
```

上面 send()用于发送异步请求，在 send()函数中创建 Ajax.Updater 对象时，options 参数中指定了 evalScripts:true，这意味着客户端将会执行服务器响应中的 JavaScript 脚本。在浏览器中浏览该，并单击“发送”按钮，将看到如图 9.21 所示的界面。

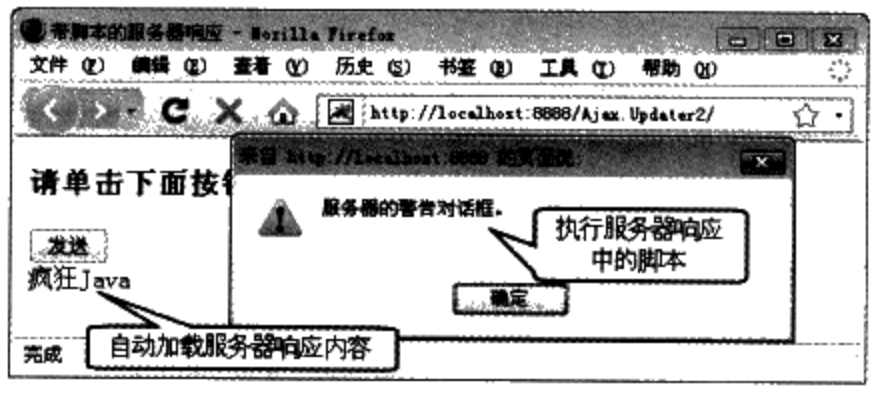


图 9.21　执行服务器响应中的脚本

上面示范应用的服务器页面直接输出了警告提示框，如果想生成能响应事件的按钮，则服务器页面可采用如下形式代码：

程序清单：codes\09\9.6\Ajax.Updater2\index.html

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<script type="text/javascript">
hi = function()
{
    alert("服务器按钮激发的单击事件");
}
</script>
<!-- 服务器生成一个按钮 -->
<input type="button" value="服务器按钮" onclick="hi();"/>
```

在这种情况下，服务器响应会在客户端页面上生成一个“服务器按钮”按钮，单击该按钮，页面将弹出一个对话框。

值得注意的是，hi 函数不能写成如下形式：

```
function hi()
{
    alert("服务器按钮激发的单击事件");
}
```

只能将 hi()函数写成匿名函数的形式，然后将该匿名函数赋给 hi 变量。后面的形式的脚本会被执行，这段脚本执行完后并不会创建一个名 hi 函数，它什么也不做。

➤➤9.6.6　使用 Ajax.PeriodicalUpdater 类

相信读者从名字上就可猜出该对象的大致用处。是的，它是一个周期性的 Ajax.Updater 类，它用于周期性地向服务器发送请求，并将服务器响应在客户端 HTML 页面的某个元素中显示出来。该类的用法与 Ajax.Updater 的用法非常相似。与 Ajax.Updater 相比，它的 options 可多指定如下两个属性：

- frequency：两次发送 Ajax 请求之间的时间间隔（单位为秒）。 如果两次服务器的两次请求

完全相同时，两次请求的发送间隔可能是 frequency * decay。该属性默认为 2。

- decay：如果服务器两次响应完全相同，则减慢发送请求的频率。如果有服务器响应有 n 次完全相同，则发送请求的时间间隔为 frequency * decay 的 n - 1 次方。该属性默认为 1。

除此之外，Ajax.PeriodicalUpdater 可使用如下两个属性：

- updater：最后一次发送 Ajax 请求所使用的 Ajax.Updater 对象。
- timer：周期性发送 Ajax 请求所使用的定时器对象。

下面示例是一个周期性的刷新价格的应用，该应用需要周期性地向服务器发送请求，服务器页面代码通过伪随机器生成价格，服务器页面代码如下：

程序清单：codes\09\9.6\Ajax.PeriodicalUpdater\price.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%
    //输出一个伪随机数
    out.println(Math.round(Math.random() * 10));
%>
```

客户端 HTML 页面通过 Ajax.PeriodicalUpdater 周期性地发送请求，代码如下：

程序清单：codes\09\9.6\Ajax.PeriodicalUpdater\index.html

```
<body>
<h3>今天的苹果价格为：</h3>
苹果价格为：<span style="color:red" id="price"></span>元/斤
<script src="prototype-1.6.0.3.js" type="text/javascript">
</script>
<script type="text/javascript">
//发送请求的服务器 URL
var url = 'price.jsp';
//创建 Ajax. PeriodicalUpdater 对象，周期性发送 Ajax 请求
var myAjax = new Ajax.PeriodicalUpdater(
    //price 用于显示服务器响应
    'price',
    url,
    {
        //定义发送请求的方法
        method: 'post',
        //定义请求参数为 null
        parameters: null,
        //发送请求的频率
        frequency:1
    });
</script>
</body>
```

该页面将每秒刷新一次，每次都将显示新的苹果价格。使用 Ajax.PeriodicalUpdater 对象，避免了需要使用定时器来发送请求，它是对 Ajax.Updater 类进一步简化的结果。

9.7 本章小结

本章详细介绍了 Prototype 库的各种知识，从 Prototype 库的下载和安装开始介绍，介绍了 Prototype 库的几个常用函数，还介绍了 Prototype.js 的自定义的类和对象，还介绍了 Prototype 库的常用扩展。本章重点介绍了 Prototype 库的 Ajax 功能，通过代码示范了如何使用 Ajax.Request、Ajax.Updater 和 Ajax.PeriodicalUpdater 发送异步请求，并介绍了 Ajax、Ajax.Responders 两个对象的用法。

下一章将介绍一个实际的应用：自动完成，当用户在文本框内输入一个字符时，系统会自动为用户补齐后面部分。下一章将通过该应用介绍如何利用 Prototype 库来简化 Ajax 编程。

CHAPTER
10

第10章
基于 Prototype 库的应用：自动完成

本章要点

- 基本的系统分析
- 设计 Hibernate 持久层
- 基于 HibernateDaoSupport 实现 DAO 组件
- 实现系统业务逻辑组件
- 使用 Spring 的 IoC 容器管理系统组件
- 设计 Servlet 提供 Ajax 响应
- 编写视图页面
- 使用 Ajax.Updater 发送请求
- 使用 Ajax.Request 发送请求
- 注册 Ajax 全局事件监听器

本章介绍的应用是对 9.6 节介绍的输入提示应用的改进，9.6 节的应用仅能提供给用户输入提示，用户不能直接选择提示，提示也是通过 JSP 页面直接生成的。本章将介绍一个更实际、更完整的 Java EE 应用。正如前面所介绍的，单纯的 Ajax 技术是没有价值的，本章介绍的应用将改进传统的 Java EE 应用，在传统 Java EE 应用的表层增加 Ajax 引擎，从而改善用户的体验。

本章的 Ajax 应用的持久层使用 Hibernate 技术；中间层使用 Spring 作为 IoC 容器，负责管理所有组件。而 Ajax 技术集中在表现层，用以改善用户体验。

10.1 应用的基本分析和设计

本应用模拟了一个用户购买笔记本的页面，用户在文本框中输入想购买的笔记本品牌，在用户输入笔记本品牌时，页面提供自动完成功能。假设用户输入“S”，则所有以 S 开头的笔记本品牌都会自动列在该单行文本框的下面，用户可以通过上、下按键进行选择。

一旦用户选择了某个笔记本品牌，在型号选择框的下面就会自动列出该品牌的所有笔记本型号。用户可以选择需要的笔记本型号，一旦选择了需要的笔记本型号，该笔记本型号的介绍就会呈现在用户眼前。整个过程采用异步请求完成，无须用户显式提交请求，而由 Ajax 引擎在后台自动完成请求的发送。

➤➤10.1.1 数据要求

考虑到上面购买笔记本的场景，可以确定数据的基本要求。整个应用需要两个表：brand 和 model，其中 brand 表用于保存笔记本品牌，而 model 表用于保存笔记本型号。brand 表中主要保存品牌的名称，model 表中主要保存型号名、型号介绍和对应的品牌。

model 表和 brand 表以外键约束关联。实际应用中，每个表可能需要保存更多、更复杂的内容。但因为本应用只是一个示范，为了突出应用的主题，数据库中只保存上面介绍的必要信息。

对于实际的项目而言，可能常常需要增加笔记本品牌，也需要增加笔记本型号。本应用只示范客户购买笔记本的场景，主要的数据操作为访问：根据品牌前缀查询品牌，根据品牌查询笔记本型号，根据型号查询笔记本详细信息。

根据实际情形来看，笔记本的品牌名肯定是不会重复的，因此应用为笔记本品牌名增加唯一约束；笔记本型号也是不会重复的，因此也为笔记本型号增加唯一约束。

➤➤10.1.2 数据表结构

根据上面的分析，可以为该系统建立数据库，数据库分成 2 个表：brand 和 model，brand 表存放笔记本品牌，而 model 表存放笔记本型号。brand 表有 2 列，b_id 列为逻辑主键，该列为自动增长列；b_name 列为品牌名，该列有唯一约束。brand 表的结构如图 10.1 所示。

model 表有 4 列，m_id 列为逻辑主键，该列为自动增加列；m_name 列为型号名，该列具有唯一约束；m_desc 列为型号描述；还有 b_id 列为外键列，该列参照 brand 表的 b_id 列。model 表的结构如图 10.2 所示。

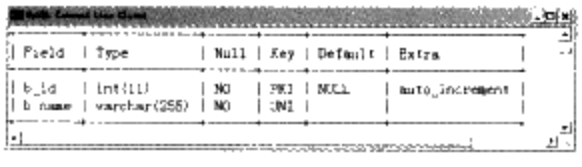


图 10.1 表 brand 的结构

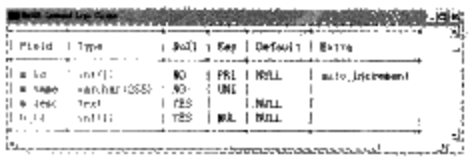


图 10.2 表 model 的结构

10.2 Domain Object 和持久层

本应用没有太复杂的业务逻辑，仅仅是单纯的数据访问操作，主要是各种数据查询，因此本应用

的 Domain Object 对象并未包含业务逻辑方法（看上去非常像贫血模式，关于架构模式的介绍可参考疯狂 Java 体系之《轻量级 Java EE 企业应用实战》）。Domain Object 对象仅仅是一些普通的 POJO，通过对这些 POJO 增加 Hibernate 映射，提供持久层访问的能力，并以 Hibernate Session 管理下的 Domain Object 为基础，提供 DAO 组件的实现。

➤➤10.2.1 Domain Object

本应用中包含两个 Domain Object，分别是 Brand 和 Model，分别代表笔记本的品牌和型号。当然，二者之间存在关联关系，借助于 Hibernate 的关联映射，Brand 拥有访问 Model 的能力。同时，Model 也拥有访问 Brand 的能力。下面是 Brand 的源代码：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\model\Brand.java

```
public class Brand implements Serializable
{
    private Integer id;
    //品牌名
    private String name;
    //该品牌对应的全部型号
    private Set<Model> models
        = new HashSet<Model>();
    //无参数的构造器
    public Brand()
    {
    }
    //省略所有属性的 setter 和 getter 方法
    ...
}
```

因为篇幅原因，上面的代码并未将所有的 setter 和 getter 方法列出，读者应该可以补齐这些 setter 和 getter 方法，这是 Hibernate 对 POJO 的要求。Brand 类对应的 Hibernate 配置文件如下：

程序清单：codes\10\InputTip\WEB-INF\src\Brand.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
    <!-- Hibernate 映射文件的根元素 -->
    <hibernate-mapping package="org.crazyjava.inputtip.model">
    <class name="Brand" table="brand">
        <!-- 映射标识属性 -->
        <id name="id" column="b_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射 name 普通属性 -->
        <property name="name" type="string"
            column="b_name" unique="true"/>
        <!-- 映射关联的所有 Model 实体 -->
        <set name="models" inverse="true">
            <key column="b_id"/>
            <one-to-many class="Model"/>
        </set>
    </class>
</hibernate-mapping>
```

借助于上面的配置文件，处于 Hibernate Session 管理下的 PO 具有操作数据库的能力。下面是 Model 类的源代码，此处同样省略了各属性的 setter 和 getter 方法：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\model\Model.java

```
public class Model implements Serializable
{
    //Model 的标识属性，该属性映射数据库表的主键
    private Integer id;
    //型号名
    private String name;
    //型号描述
    private String desc;
    //该型号所属的品牌
    private Brand brand;
    //无参数的构造器
    public Model()
    {
    }
    //省略所有属性的 setter 和 getter 方法
    ...
}
```

上面的 POJO 同样需要一个 Hibernate 映射文件才能具有操作数据库的能力，下面是 Model 类对应的 Hibernate 映射文件：

程序清单：codes\10\InputTip\WEB-INF\src\Model.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.inputtip.model">
    <class name="Model" table="model">
        <!-- 映射标识属性 -->
        <id name="id" column="m_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <property name="name" type="string"
            unique="true" column="m_name"/>
        <property name="desc" type="string" column="m_desc"/>
        <!-- 映射 Brand 关联的 Model 实体 -->
        <many-to-one name="brand"
            column="b_id" not-null="true"/>
    </class>
</hibernate-mapping>
```

现在已经完成了底层 Domain Object 的设计，并为其编写了对应的映射文件，借助于这些 Domain Object 以及对应的映射文件，可以很简单地实现系统所需的 DAO 组件。本应用在实现 DAO 组件时还借助了 Spring 的 HibernateDaoSupport 和 HibernateTemplate。

➤➤10.2.2 实现 DAO 组件

DAO 组件是 Java EE 应用里最缺乏变化的组件，也是最简单的组件，它们永远都是包含最基本的 CRUD（增加、查询、更新、删除）操作，不会有太多的变化。

根据 Java EE 应用面向接口编程的原则，每个 DAO 组件都由两个部分组成，一个是接口，另一个是接口的实现类。下面分别为 Brand 和 Model 提供对应的接口和实现类。BrandDao 组件的接口和实现类代码如下：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\dao\BrandDao.java

```
public interface BrandDao
{
    /**
     * 根据 ID 查找品牌
     * @param id 需要查找的品牌 ID
     */
    Brand get(Integer id);
    /**
     * 增加品牌
     * @param Brand 需要增加的品牌
     */
    void save(Brand Brand);
    /**
     * 修改品牌
     * @param Brand 需要修改的品牌
     */
    void update(Brand Brand);
    /**
     * 删除品牌
     * @param id 需要删除的品牌 ID
     */
    void delete(Integer id);
    /**
     * 删除品牌
     * @param Brand 需要删除的品牌
     */
    void delete(Brand Brand);
    /**
     * 查询全部品牌
     * @return 全部品牌
     */
    List<Brand> findAll();
}
```

上面的接口提供了一系列 CRUD 方法的规范，但并未给出任何实现，借助于 HiberateDaoSupport 和 HibernateTemplate 的支持，可以很方便地实现 DAO 组件，下面是 BrandDao 组件的实现类代码：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\dao\impl\BrandDaoImpl.java

```
public class BrandDaoHibernate
    extends HibernateDaoSupport implements BrandDao
    {
    /**
     * 根据 ID 查找品牌
     * @param id 需要查找的品牌 ID
     */
    public Brand get(Integer id)
    {
        //借助于 HibernateTemplate 的 get 方法返回特定主键对应的记录
        return (Brand)getHibernateTemplate()
            .get(Brand.class , id);
    }
    /**
     * 增加品牌
     * @param brand 需要增加的品牌
     */
    public void save(Brand brand)
    {
        //借助于 HibernateTemplate 的 save 方法增加记录
        getHibernateTemplate().save(brand);
    }
```

```
    /**
     * 修改品牌
     * @param brand 需要修改的品牌
     */
    public void update(Brand brand)
    {
        //借助于 HibernateTemplate 的 saveOrUpdate 更新记录
        getHibernateTemplate().saveOrUpdate(brand);
    }
    /**
     * 删除品牌
     * @param id 需要删除的品牌 ID
     */
    public void delete(Integer id)
    {
        //借助于 HibernateTemplate 的 delete 方法删除特定主键对应的记录
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 删除品牌
     * @param brand 需要删除的品牌
     */
    public void delete(Brand brand)
    {
        //借助于 HibernateTemplate 的 delete 方法删除指定实体
        getHibernateTemplate().delete(brand);
    }
    /**
     *查询全部品牌
     * @return 全部品牌
     */
    public List<Brand> findAll()
    {
        //借助于 HibernateTemplate 的 find()方法查询记录
        return (List<Brand>)getHibernateTemplate()
            .find("from Brand");
    }
}
```

HibernateDaoSupport 类提供了一个 getHibernateTemplate()方法，该方法可以方便地返回一个 HibernateTemplate 对象，该对象是 Spring 为简化 Hibernate 持久层访问提供的模板类，该类可以非常便捷地进行持久层访问。读者从代码中可以看出，很多 CRUD 方法的实现都只需要一行代码即可。这得益于 Spring 和 Hibernate 为我们提供的便捷。ModelDao 组件的接口的代码如下：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\dao\ModelDao.java

```
public interface ModelDao
{
    /**
     * 根据 ID 查找型号
     * @param id 需要查找的型号 ID
     */
    Model get(Integer id);
    /**
     * 增加型号
     * @param Model 需要增加的型号
     */
    void save(Model model);
    /**
     * 修改型号
```

```
        * @param Model 需要修改的型号
        */
        void update(Model model);
        /**
        * 删除型号
        * @param id 需要删除的型号 ID
        */
        void delete(Integer id);
        /**
        * 删除型号
        * @param Model 需要删除的型号
        */
        void delete(Model model);
        /**
        * 查询全部型号
        * @return 全部型号
        */
        List<Model> findAll();
        /**
        * 根据品牌查询型号
        * @param brand 需要查询的品牌
        * @return 该品牌对应的全部的型号
        */
        List<Model> findByBrand(String brand);
        /**
        * 根据型号名查询型号
        * @param model 需要查询的型号名
        * @return 该型号名对应的型号
        */
        Model findByModel(String model);
    }
```

ModelDao 的实现类 ModelDaoHibernate 的代码如下：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\dao\impl\ModelDaoHibernate.java

```
public class ModelDaoHibernate
    extends HibernateDaoSupport implements ModelDao
    {
    /**
    * 根据 ID 查找型号
    * @param id 需要查找的型号 ID
    */
    public Model get(Integer id)
    {
        return (Model)getHibernateTemplate()
            .get(Model.class , id);
    }
    /**
    * 增加型号
    * @param model 需要增加的型号
    */
    public void save(Model model)
    {
        getHibernateTemplate().save(model);
    }
    /**
    * 修改型号
    * @param model 需要修改的型号
    */
    public void update(Model model)
```

```
    {
    getHibernateTemplate().saveOrUpdate(model);
    }
    /**
     * 删除型号
     * @param id 需要删除的型号 ID
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 删除型号
     * @param model 需要删除的型号
     */
    public void delete(Model model)
    {
        getHibernateTemplate().delete(model);
    }
    /**
     * 查询全部型号
     * @return 返回全部型号
     */
    public List<Model> findAll()
    {
        return (List<Model>)getHibernateTemplate()
            .find("from Model");
    }
    /**
     * 根据品牌查询型号
     * @param brand 需要查询的品牌
     * @return 该品牌对应的全部的型号
     */
    public List<Model> findByBrand(String brand)
    {
        return (List<Model>)getHibernateTemplate()
            .find("from Model as m where m.brand.name=?" , brand);
    }
    /**
     * 根据型号名查询型号
     * @param model 需要查询的型号名
     * @return 该型号名对应的型号
     */
    public Model findByModel(String model)
    {
        List<Model> ml = (List<Model>)getHibernateTemplate()
            .find("from Model as m where m.name=?" , model);
        if (ml != null && ml.size() >= 0)
        {
            return ml.get(0);
        }
        return null;
    }
}
```

10.3 实现 Service 组件

Service 组件负责对外提供业务所需的逻辑方法，本应用的业务逻辑组件主要提供三个方法：根据

品牌前缀查询所有品牌，根据品牌名查询该品牌下的全部型号，根据型号名返回该型号的详细描述。

Service 组件同样遵循面向接口编程的原则，为每个业务逻辑组件编写接口和实现类，以实现 Web 层组件与业务逻辑层组件实现的分离。下面是业务逻辑组件的接口定义，该接口定义了该业务逻辑组件必须包含的三个方法：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\service\TipService.java

```
public interface TipService
{
    /**
     * 根据品牌前缀来返回所有对应的品牌名
     * @param prefix 品牌前缀
     * @return 该品牌前缀对应的所有品牌名
     */
    public List<String> getBrandsByPrefix(String prefix);
    /**
     * 根据品牌名返回该品牌下的型号名
     * @param brand 品牌名
     * @return 该品牌对应的全部型号名
     */
    public List<String> getModelsByBrand(String brand);
    /**
     * 根据型号名来返回该型号的描述
     * @param model 型号名
     * @return 该型号名对应的型号描述
     */
    public String getDescByModel(String model);
}
```

业务逻辑组件的实现依赖于 DAO 组件，DAO 组件提供了数据访问的能力，业务逻辑组件以 DAO 组件为基础实现三个业务逻辑方法，业务逻辑组件中 DAO 组件的实例化则依赖 Spring 的 IoC 容器管理，由 Spring 负责为业务逻辑组件注入 DAO 组件实例。下面是业务逻辑组件的实现代码：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\service\impl\TipServiceImpl.java

```
public class TipServiceImpl implements TipService
{
    //实现业务逻辑方法所依赖的两个 DAO 组件
    private BrandDao bd;
    private ModelDao md;
    //依赖注入两个 DAO 组件所必需的 setter 方法
    public void setBrandDao(BrandDao bd)
    {
        this.bd = bd;
    }
    public void setModelDao(ModelDao md)
    {
        this.md = md;
    }
    //根据品牌前缀返回所有以该前缀开始的名牌名
    public List<String> getBrandsByPrefix(String prefix)
    {
        //返回所有的品牌记录
        List<Brand> brands = bd.findAll();
        List<String> result = new ArrayList<String>();
        //遍历所有的品牌记录
        for (Brand brand : brands )
        {
            //如果品牌记录以前缀开始
            if (brand.getName().toUpperCase()
```

```
                .startsWith(prefix.toUpperCase()))
            {
                //将该品牌添加到结果集合里
                result.add(brand.getName());
            }
        }
        return result;
    }
    //根据品牌名返回该品牌的所有型号名
    public List<String> getModelsByBrand(String brand)
    {
        List<String> result = new ArrayList<String>();
        //根据品牌返回所有的型号实例
        List<Model> mlist = md.findByBrand(brand);
        //遍历型号集合，将每个型号的名字添加到结果集合里
        for (Model model : mlist )
        {
            result.add(model.getName());
        }
        return result;
    }
    //根据型号名，返回型号描述
    public String getDescByModel(String model)
    {
        //根据型号名返回特定的型号实体
        Model m = md.findByModel(model);
        //如果型号不为空，则返回型号描述
        if (m != null)
        {
            return m.getDesc();
        }
        return null;
    }
}
```

一旦完成了 DAO 组件和业务逻辑组件的编写，就可以将这些组件配置在 Spring 容器中，配置在 Spring 容器中的所有组件都会接受 Spring 容器管理，而 Web 层组件则无须与任何具体的组件耦合，它们只需通过 Spring 容器来获取业务逻辑组件。本应用基于 Spring 2.5 完成，故此采用 Schema 作为配置文件的语义约束：

程序清单：codes\10\InputTip\WEB-INF\applicationContext.xml

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 Spring 配置文件的 Schema 信息 -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <!-- 定义数据源 Bean，使用 C3P0 数据源实现 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
        destroy-method="close">
        <!-- 指定连接数据库的驱动 -->
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
        <!-- 指定连接数据库的 URL -->
        <property name="jdbcUrl" value="jdbc:mysql://localhost/tips"/>
        <!-- 指定连接数据库的用户名 -->
        <property name="user" value="root"/>
        <!-- 指定连接数据库的密码 -->
        <property name="password" value="32147"/>
```

```
        <!-- 指定连接数据库连接池的最大连接数 -->
        <property name="maxPoolSize" value="40"/>
        <!-- 指定连接数据库连接池的最小连接数 -->
        <property name="minPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的初始化连接数 -->
        <property name="initialPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的连接的最大空闲时间 -->
        <property name="maxIdleTime" value="20"/>
    </bean>
    <!-- 定义 Hibernate 的 SessionFactory -->
    <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- 依赖注入数据源，所注入的正是上面定义的 dataSource -->
        <property name="dataSource" ref="dataSource"/>
        <!-- mappingResouces 属性用来列出全部映射文件 -->
        <property name="mappingResources">
            <list>
                <!-- 以下用来列出 Hibernate 映射文件 -->
                <value>Brand.hbm.xml</value>
                <value>Model.hbm.xml</value>
            </list>
        </property>
        <!-- 定义 Hibernate 的 SessionFactory 的属性 -->
        <property name="hibernateProperties">
            <props>
                <!-- 指定数据库方言 -->
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLInnoDBDialect</prop>
                <!-- 是否根据需要每次自动创建数据库 -->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <!-- 显示 Hibernate 持久化操作所生成的 SQL -->
                <prop key="hibernate.show_sql">true</prop>
                <!-- 将 SQL 脚本进行格式化后再输出 -->
                <prop key="hibernate.format_sql">true</prop>
            </props>
        </property>
    </bean>
    <!-- 配置 brandDao 组件 -->
    <bean id="brandDao"
        class="org.crazyjava.inputtip.dao.impl.BrandDaoHibernate">
        <!-- 注入 SessionFactory 实例 -->
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
    <!-- 配置 modelDao 组件 -->
    <bean id="modelDao"
        class="org.crazyjava.inputtip.dao.impl.ModelDaoHibernate">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
    <!-- 配置 tipService 组件 -->
    <bean id="tipService"
        class="org.crazyjava.inputtip.service.impl.TipServiceImpl">
        <!-- 为 tipService 组件依赖注入 2 个 DAO 组件 -->
        <property name="brandDao" ref="brandDao"/>
        <property name="modelDao" ref="modelDao"/>
    </bean>
</beans>
```

现在已经完成了本应用的后台部分，tipService 组件提供的三个方法可完成本应用需要的三个业务逻辑功能：根据品牌前缀来获取以该前缀开始的所有品牌名，根据品牌名来获取该品牌所有的型号，

以型号名来获取该型号的详细描述。

可能有读者会注意到，TipService 组件的方法不具备事务性，Spring 容器中也没有为该组件的方法配置声明式事务管理。没错，因为笔者不想有太多额外的操作影响读者的阅读，因此并未为 TipService 组件配置事务管理。

注意：

如果需要在实际项目中使用该应用，则应该为 TipService 业务组件配置声明式事务管理。关于 Spring 声明式事务管理的介绍请参考疯狂 Java 体系的《轻量级 Java EE 企业应用实战》。

10.4 使用 Servlet 提供服务器响应

本应用需要向服务器异步发送三个请求，因此应该在服务器端提供三个 Servlet，每个 Servlet 负责处理一个用户请求，因为服务器的响应数据非常简单，因此本应用直接使用 Servlet 生成响应，而未使用 JSP 作为表现层。

每个 Servlet 都通过调用 TipService 业务逻辑组件来完成各自的响应，因为 TipService 被配置在 Spring 容器中，因此 Servlet 中无须手动创建 TipService 实例，而只需通过 Spring 容器来访问 TipService 实例即可，因此必须先获得 Spring 容器的引用。

通常都将 Spring 容器配置成随 Web 应用的启动而初始化，配置 Spring 容器随 Web 应用的初始化有两种方法：使用 load-on-startup Servlet 或 Listener。当然建议优先采用 Listener 配置，本应用即采用 Listener 来配置 Spring 容器的初始化。下面是为了配置 Spring 容器随 Web 应用启动而初始化在 web.xml 文件中增加的配置片段：

程序清单：codes\10\InputTip\WEB-INF\web.xml

```
<!-- 配置 Spring 配置文件的位置 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
<!-- 使用 ContextLoaderListener 初始化 Spring 容器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

一旦通过 web.xml 文件配置了 Spring 容器的自动初始化，Spring 容器就会随着 Web 应用的启动而完成初始化，初始化后的 Spring 容器被保存为一个 ServletContext 属性，Servlet 可以通过该属性来访问 Spring 容器。实际上，Spring 提供了一个工具类 WebApplicationContextUtils，该类能以更便捷的方法访问该 Web 应用中的 Spring 容器。

Servlet 一旦获得了 Spring 容器的引用，就可以访问到 Spring 容器中的任何组件。当然，Servlet 通常只需要访问 Service 组件，下面依次介绍三个 Servlet 的实现。

➤➤10.4.1 根据前缀查询品牌

根据品牌前缀查询品牌是通过调用 Service 组件的 getBrandsByPrefix 方法实现的，该方法将根据品牌名的前缀，返回所有以前缀开始的品牌名。下面是该 Servlet 的源代码。

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\servlet\GetBrandsServlet.java

```
public class GetBrandsServlet extends HttpServlet
{
    public void service(HttpServletRequest request ,
```

```
        HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        //通过 WebApplicationContextUtils 工具类获得 Spring 容器，
        //通过 Spring 容器访问容器中的业务逻辑组件
        TipService ts = (TipService)WebApplicationContextUtils
            .getWebApplicationContext(getServletContext())
            .getBean("tipService");
        //设置解码用的字符集，Ajax 的 POST 请求都采用 UTF-8 的编码集
        request.setCharacterEncoding("utf-8");
        //获取请求参数
        String prefix = request.getParameter("prefix");
        //调用业务逻辑组件的方法，返回品牌名组成的集合
        List<String> brands = ts.getBrandsByPrefix(prefix);
        String result = "";
        //遍历集合中的品牌名，将所有品牌名拼成一个字符串
        for (String brand : brands )
        {
            result += brand + "$";
        }
        //设置响应的文件头
        response.setContentType("text/html;charset=GBK");
        PrintWriter out =  response.getWriter();
        //输出响应
        out.println(result);
    }
}
```

该 Servlet 根据品牌名返回该品牌下所包含的全部型号，返回结果将所有型号名以$符号分隔开，该 Servlet 使用普通文本响应，不是 XML 响应。将该 Servlet 配置在 web.xml 文件中，以便被客户端调用，配置该 Sevlet 的配置片段如下：

程序清单：codes\10\InputTip\WEB-INF\web.xml

```
<!-- 定义 getBrands Servlet -->
<servlet>
    <servlet-name>getBrands</servlet-name>
    <servlet-class>org.crazyjava.inputtip.servlet.GetBrandsServlet</servlet-class>
</servlet>
<!-- 为该 Servlet 配置 URL -->
<servlet-mapping>
    <servlet-name>getBrands</servlet-name>
    <url-pattern>/getBrands.do</url-pattern>
</servlet-mapping>
```

完成了该 Servlet 的配置后，该 Servlet 即可被客户端请求，并根据请求生成响应。一旦客户端向 getBrands.do 发送请求，而且包含合适的请求参数，则服务器将生成类似 SONY$DELL$的字符串响应。

➤➤10.4.2 根据品牌查询型号

根据品牌查询型号是通过调用 Service 组件的 getModelsByBrand()方法实现的，该方法将根据品牌名返回该品牌下的所有型号名。下面是该 Servlet 的源代码：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\servlet\GetModelsServlet.java

```
public class GetModelsServlet extends HttpServlet
{
    public void service(HttpServletRequest request
        ,HttpServletResponse response)
        throws ServletException, java.io.IOException
```

```
    {
        //通过 WebApplicationContextUtils 工具类获得 Spring 容器，
        //通过 Spring 容器访问容器中的业务逻辑组件
        TipService ts = (TipService)WebApplicationContextUtils
            .getWebApplicationContext(getServletContext())
            .getBean("tipService");
        //设置解码用的字符集，Ajax 的 POST 请求都采用 UTF-8 的编码集
        request.setCharacterEncoding("utf-8");
        //获取请求参数
        String brand = request.getParameter("brand");
        //调用业务逻辑组件的方法，返回型号名组成的集合
        List<String> models = ts.getModelsByBrand(brand);
        String result = "";
        //遍历集合中的型号名，将所有型号名拼成一个字符串
        for (String model : models )
        {
            result += model + "$";
        }
        response.setContentType("text/html;charset=GBK");
        PrintWriter out =  response.getWriter();
        //输出响应
        out.println(result);
    }
}
```

类似地，该 Servlet 也是返回一个简单字符串，该字符串以$分隔指定品牌对应的所有型号名。将该 Servlet 配置在 web.xml 文件中，以便被客户端调用，配置该 Sevlet 的配置片段如下：

程序清单：codes\10\InputTip\WEB-INF\web.xml

```
<!-- 定义 getModels Servlet -->
<servlet>
    <servlet-name>getModels</servlet-name>
    <servlet-class>org.crazyjava.inputtip.servlet.GetModelsServlet</servlet-class>
</servlet>
<!-- 为该 Servlet 配置 URL -->
<servlet-mapping>
    <servlet-name>getModels</servlet-name>
    <url-pattern>/getModels.do</url-pattern>
</servlet-mapping>
```

完成了该 Servlet 的配置后，该 Servlet 即可被客户端请求，并根据请求生成响应。一旦客户端向 getModels.do 发送请求，而且包含合适的请求参数，则服务器将生成字符串响应，该字符串包含了该品牌下全部型号的型号名，型号名之间以$符号分隔。

➤➤10.4.3 根据型号查询详细信息

根据型号查询详细信息是通过 Service 组件的 getDescByModel()方法实现的，该方法将根据型号名，返回该型号对应的详细信息。下面是该 Servlet 的源代码：

程序清单：codes\10\InputTip\WEB-INF\src\org\crazyjava\inputtip\servlet\GetDetailServlet.java

```
public class GetDetailServlet extends HttpServlet
{
    public void service(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        //通过 WebApplicationContextUtils 工具类获得 Spring 容器，
        //通过 Spring 容器访问容器中的业务逻辑组件
```

```
            TipService ts = (TipService)WebApplicationContextUtils
                .getWebApplicationContext(getServletContext())
                .getBean("tipService");
            //设置解码用的字符集，Ajax 的 POST 请求都采用 UTF-8 的编码集
            request.setCharacterEncoding("utf-8");
            //获取请求参数
            String model = request.getParameter("model");
            response.setContentType("text/html;charset=GBK");
            PrintWriter out = response.getWriter();
            //输出响应
            out.println(ts.getDescByModel(model));
        }
    }
```

该 Servlet 根据型号名返回该型号的详细描述，因为详细描述无须被客户端再次处理，因此直接将详细信息输出成服务器响应。将该 Servlet 配置在 web.xml 文件中，以便被客户端调用，配置该 Sevlet 的配置片段如下：

程序清单：codes\10\InputTip\WEB-INF\web.xml

```
<!-- 定义 getDetail Servlet -->
<servlet>
    <servlet-name>getDetail</servlet-name>
    <servlet-class>org.crazyjava.inputtip.servlet.GetDetailServlet</servlet-class>
</servlet>
<!-- 为该 Servlet 配置 URL -->
<servlet-mapping>
    <servlet-name>getDetail</servlet-name>
    <url-pattern>/getDetail.do</url-pattern>
</servlet-mapping>
```

完成了该 Servlet 的配置后，该 Servlet 即可被客户端请求，并根据请求生成响应。一旦客户端向 getDetail.do 发送请求，而且包含合适的请求参数，则服务器将生成一个字符串响应，响应就是该型号对应的详细信息。

10.5 客户端 HTML 页面实现

本应用的静态页面并不包含太多、太复杂的表现效果，该页面主要包含两个单行文本输入框，第一个输入框用于输入笔记本的品牌名，第二个输入框用于输入笔记本的型号名。当然本应用无须等待用户手工输入完成，系统能提供自动完成。

当然，客户端的 HTML 页面还包含了几个隐藏的 HTML 元素，主要有四个隐藏的 HTML 元素，一个是品牌自动完成的提示<div.../>元素，一个是型号自动完成的提示<div.../>元素，还有一个是输出型号详细信息的<div.../>元素，还有一个用于标识 Ajax 交互的 loading 图片。下面是该静态 HTML 页面的源代码：

程序清单：codes\10\InputTip\index.html

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 输入提示示范 </title>
    <meta name="author" content="Yeeku.H.Lee" />
    <meta name="website" content="http://www.crazyjava.org" />
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <style type="text/css">
    /* 定义输入提示中被选中项的 CSS 样式 */
    .selectTip {
        background-color:#afeeee;
```

```
        border:1px solid gold;
    }
    /* 定义输入提示的 CSS 样式 */
    .tip {
        border:1px solid menu;
        background-color:#ffffcc;
        width:160px;
    }
    /* 定义显示 model 详细信息的 CSS 样式 */
    .detail {
        font-size:9pt;
        width:320px;
        border:1px solid menu;
        background-Color:#F0F8FF;
    }
    </style>
    <script src="common/prototype-1.6.0.3.js" type="text/javascript">
    </script>
</head>
<body>
<div id="loading" style="display:none;position:absolute;
    left:200px;top:60px;">
<img src="img/indicator.gif" alt="loading"/>
</div>
<h3>请输入想购买的笔记本品牌</h3>
<div style="width:300px;font-size:9pt">
输入 IBM,DELL,SONY,TOSHIBA,SAMSUNG 可看到明显效果:</div>
<input id="brand" name="brand" type="text"
    onblur="$('brandTip').hide();"/>
<div id="brandTip" class="tip" style="display:none;"></div>
<h3>请输入想购买的笔记本型号</h3>
<input id="model" name="model" type="text"
    onblur="$('modelTip').hide();"/>
<div id="modelTip" class="tip" style="display:none;"></div>
<div id="modelDetail" class="detail" style="display:none;"></div>
<script src="js/tip.js" type="text/javascript">
</script>
</body>
</html>
```

该页面的 HTML 元素并没有包含太多的事件处理函数，只在 brand 和 model 两个元素中增加了 onblur 事件处理器，由于这两个所谓的事件处理器是如此简单：仅包含一行简单代码，所以将该事件处理的代码直接设置成 onblur 属性值。

根据客户端 HTML 页面也应该遵循的 MVC 原则，HTML 代码中不应该绑定太多的事件处理函数，对于普通的 HTML 页面设计人员而言，他无须理会、也无法理会页面中的 HTML 元素应该包含怎样的行为。因此，将事件处理函数绑定到 HTML 元素既不利于业务逻辑和表现逻辑的分离，也不利于团队开发。

考虑一个 Ajax 应用的开发场景：静态的 HTML 页面交给前台的美工完成，Service 层下的 DAO 组件和业务逻辑组件则由专门程序开发，而 Web 层程序员则负责完成控制器 Servlet 和页面的 JavaScript 部分。为避免 Web 层程序员改动 HTML 页面，从而导致为 HTML 页面引入新的错误，Ajax 的开发者本不应该修改 HTML 代码，本应用正是基于该思路完成，所有的 JavaScript 代码以单独的文件存放，所有的事件处理函数也放在 JavaScript 代码文件中完成，而不是直接耦合在 HTML 页面代码中。图 10.3 显示了该静态 HTML 页面的浏览效果。

图 10.3 静态 HTML 页面的效果

此时的 HTML 页面，是一个静态的 HTML 页面，除了两行简单的事件处理代码，该页面几乎没

有任何的事件响应能力。下面借助于 Prototype 库为页面的 HTML 元素增加事件响应的能力。

10.6 增加 HTML 页面的事件响应能力

为了让页面中的 HTML 元素能响应用户动作，必须为页面中的 HTML 元素绑定相应的事件处理函数，考虑页面中需要响应的事件有：

- 用户在品牌输入框中输入，应激发服务器端的自动完成服务。
- 用户在品牌输入框中通过上、下键来选择合适的品牌。
- 用户在品牌输入框中单击回车键选中合适的品牌，应激发服务器端响应出该品牌下所有的型号。
- 用户在型号输入框中通过上、下键来选择合适的型号。
- 用户在型号输入框中单击回车键选中合适的型号，应激发服务器端响应该型号对应的详细描述。

对于使用上、下键来选择合适的品牌和型号的事件处理，它们可以共用同一个处理函数，其他的用户动作则都应该编写对应的事件处理函数。下面依次介绍这些事件处理函数。

➤➤10.6.1 实现品牌输入框的事件处理器

只要品牌输入框中的文字变化，都应该从服务器查询是否有适合的提示。对于这种要求，采用 Prototype 库的 Form.Element.Observer 十分合适。因此在 JavaScript 代码中使用如下代码为品牌输入框绑定事件处理函数。

```
//监听 brand 表单控件里 value 的改变
new Form.Element.Observer("brand", 1, searchBrand);
```

上面的事件绑定的含义是每隔一秒检测一次 brand 文本框的值是否改变，如果 brand 文本框的值发生了改变，即触发 searchBrand()函数；如果 brand 文本框的值没有任何改变，则无须触发任何动作。

可能读者已经想到：searchBrand 负责向服务器发送 Ajax 请求，这种 Ajax 请求并不需要周期发送，而且服务器响应不能直接输出到某个 HTML 元素内（服务器响应还需要在客户端重新解析），因此使用 Ajax.Request 发送 Ajax 请求。下面是 seachBrand 函数的代码：

程序清单：codes\10\InputTip\js\tip.js

```
function searchBrand()
{
    //发送请求的服务器 URL
    var url = 'getBrands.do';
    //生成请求参数
    var params = "prefix=" + $F('brand');
    //创建 Ajax.Request 对象，用于发送请求
    new Ajax.Request(
    url,
    {
        method:'post',
        //发送参数
        parameters: params,
        //指定回调函数
        onComplete: showResponse,
    });
}
```

使用 Ajax.Request 类允许使用回调函数，回调函数可以解析服务器响应，并将响应动态显示在 HTML 页面上，下面是回调函数 showResponse()的代码：

程序清单：codes\10\InputTip\js\tip.js

```
function showResponse(request)
{
```

```
    //获取服务器响应，并将响应字符串以$符号作为分隔符，分解成字符串数组
    var brandList = request.responseText.split("$");
    //获取品牌提示元素
    var bt = $("brandTip");
    //清空品牌提示元素的内容
    bt.innerHTML = "";
    //如果字符串数组的长度大于 1
    if ( brandList.length > 1)
    {
        //遍历品牌名，将每个品牌添加到 brandTip 元素中
        for(var i = 0 ; i < brandList.length - 1 ; i++)
        {
            var a = document.createElement("div");
            a.innerHTML = brandList[i];
            bt.appendChild(a);
        }
        //选中第一个品牌所在<div.../>元素
        bt.firstChild.className = "selectTip";
        //显示 brandTip 元素
        if( tip != $("brand").value)
        {
            bt.show();
        }
    }
    else
    {
        //隐藏 brandTip 元素
        bt.hide();
    }
}
```

当品牌输入框的文字改变时，searchBrand()会向服务器发送 Ajax 请求，当服务器响应生成后，JavaScript 会将响应输出在页面的 brandTip 元素中。图 10.4 显示了当用户在品牌输入框中输入 s 后的效果。

提示元素出现后，用户即可通过向上、向下键来选择相应品牌。当用户按下回车时，页面应该自动选中对应的品牌，并在下面的笔记本型号框中出现对应于该品牌的全部型号。

图 10.4　服务器生成自动完成

➤➤10.6.2　实现键盘事件的处理器

实际上，不管是品牌输入框，还是型号输入框，它们的键盘事件的处理都非常相似，如果单击向上键，则品牌、型号提示列表框的选中项上移；如果单击了向下键，则品牌、型号提示列表框的选中项下移。如果单击回车键，则将选中的品牌或型号输入到品牌输入框或型号输入框中。

因为两个单行文本输入框的事件处理函数大同小异，因此使用了同一个事件处理函数。为二者绑定事件处理函数可借助于 Prototype 库对 Event 的扩展，绑定键盘事件处理器的代码如下：

程序清单：codes\10\InputTip\js\tip.js

```
//为 brand、model 元素绑定 keydown 事件的处理器为 move，
//false 表示该事件处理器在冒泡阶段激发
Event.observe("brand", "keydown", move, false);
Event.observe("model", "keydown", move, false);
```

move 函数根据用户击键的键盘码来确定用户单击了哪个键，从而决定采用不同的处理策略。下面是 move 函数的代码：

程序清单：codes\10\InputTip\js\tip.js

```
function move(event)
{
    //获取用户单击事件的事件源：只有 brand 或 model 两个元素
    var srcEl = Event.element(event);
    //获取 brandTip 或 modelTip 元素，即获取品牌或模型提示所用的<div.../>元素
    var tipEl = $(srcEl.id + "Tip");
    //提示 DIV 元素处于显示状态才需要处理
    //如果提示 DIV 元素都没有出现，则用户单击向上、向下键都不会有任何反应
    if (tipEl.style.display == "" )
    {
        //向下键
        if(event.keyCode == 40 )
        {
            //如果提示框中有提示的品牌或提示型号
            if (tipEl.childNodes.length > 1)
            {
                //如果已经选中了最后一条
                if(tipEl.lastChild.className == "selectTip")
                {
                    //改变为选中第一条
                    tipEl.firstChild.className = "selectTip";
                    //让最后一条不再处于选中状态
                    tipEl.lastChild.className = "null";
                    return ;
                }
                //获取提示框的提示行数组
                var bList = tipEl.childNodes;
                //遍历提示<div.../>元素中的品牌或型号
                for (var i = 0 ; i < bList.length - 1 ; i ++)
                {
                    //如果第 i 个品牌或型号被选中
                    if (bList[i].className == "selectTip")
                    {
                    //将第 i+1 个品牌或型号选中，即向下移动
                    bList[i + 1].className = "selectTip";
                    //将原来的第 i 个品牌或型号改为不选中
                    bList[i].className = "null";
                    return ;
                    }
                }
            }
        }
        //向上键
        else if(event.keyCode == 38)
        {
            //如果提示框中有提示的品牌或提示型号
            if (tipEl.childNodes.length > 1)
            {
                //如果已经选中了第一条
                if(tipEl.firstChild.className == "selectTip")
                {
                    //改为选中最后一条
                    tipEl.lastChild.className = "selectTip";
                    //将第一条改为不选中
                    tipEl.firstChild.className = "null";
                    return ;
                }
                //获取提示框的提示行数组
                var bList = tipEl.childNodes;
                //遍历所有的品牌或型号
                for (var i = 1 ; i < bList.length ; i ++)
```

```
                {
                    //如果第 i 个品牌或型号为选中状态
                    if (bList[i].className == "selectTip")
                    {
                    //将第 i-1 个品牌或型号改为选中状态，即向上移动
                    bList[i - 1].className = "selectTip";
                    //将原来的第 i 个品牌或型号改为不选中
                    bList[i].className = "null";
                    return ;
                    }
                }
            }
        }
        //回车键
        else if(event.keyCode == 13)
        {
            //获取提示<div.../>元素的所有子元素
            var bList = tipEl.childNodes;
            //遍历提示<div.../>元素的全部子元素
            for (var i = 0 ; i < bList.length ; i ++)
            {
                //如果第 i 个元素当前处于选中状态
                if (bList[i].className == "selectTip")
                {
                    //将 tip 值、输入框的值改为选中状态下的品牌或型号
                    tip = srcEl.value = bList[i].innerHTML;
                    //隐藏提示
                    Element.hide(tipEl);
                    //如果是 brand 元素上发生了单击回车事件
                    if (srcEl.id == "brand")
                    {
                    //使 model 元素获得焦点
                    $("model").focus();
                    //调用 searchModel()，发送 Ajax 请求：显示当前品牌的全部型号
                    searchModel();
                    }
                    //如果 model 元素发生了单击回车事件
                    if (srcEl.id == "model")
                    {
                    //调用 getDetail()，发送 Ajax 请求：显示当前品牌的详细信息
                    getDetail();
                    }
                    return ;
                }
            }
        }
    }
}
```

上面的粗体字代码是实现对按键事件处理的关键代码，上面的代码能根据 brand 元素或 model 元素获取各自输入提示的<div.../>元素。因为 brand 元素的提示<div.../>元素的 id 为 brandTip，而 model 元素的提示<div.../>元素的 id 为 modelTip。

上面的代码中还使用了一个全局属性 tip，使用该属性是为了当用户单击回车时隐藏提示<div.../>元素。注意到 10.6.1 节的代码中提示<div.../>元素出现的时机是：只要 brand 文本框的文本发生了改变，且服务器生成了自动完成提示。如果没有采用全局属性 tip 控制，则只要用户单击了选中的品牌名，brand 元素的文本肯定发生改变，并且文本内容总对应于一条品牌提示，那将导致提示<div.../>元素重新出现。为了避免这个问题，笔者使用了一个 tip 全局属性，该属性保存了选中的提示品牌或选中的提示型号，如果品牌输入框的值与当前 tip 属性相同，则隐藏提示；如果型号输入框的值与当前 tip 属性相同，则隐藏提示。

➤➤10.6.3 根据品牌提示型号

一旦在品牌输入框中单击回车选中某个品牌，将发出异步的 Ajax 请求，该请求向服务器申请显示该品牌下的全部笔记本型号。发送该请求的函数为 searchModel()，该请求一样使用 Ajax.Request 完成，代码如下：

程序清单：codes\10\InputTip\js\tip.js

```
function searchModel()
{
    //发送请求的 URL
    var url = 'getModels.do';
    var params = "brand=" + $F('brand');
    //创建 Ajax.Request 对象，发送异步请求
    new Ajax.Request(
    url,
    {
        method:'post',
        //设置请求参数
        parameters: params,
        //指定回调函数
        onComplete: showModel,
    });
}
```

该请求发出后，得到服务器响应，服务器的响应是类似于 dddddd的字符串，回调函数负责解析该字符串，并显示在页面的提示框中。

下面是 shwoModel()回调函数的代码：

程序清单：codes\10\InputTip\js\tip.js

```
function showModel(request)
{
    //将服务器响应字符串以$作为分隔符分解成字符串数组
    var modelList = request.responseText.split("$");
    //获取型号提示的<div.../>元素
    var mt = $("modelTip");
    //清空型号提示的<div.../>的内容
    mt.innerHTML = "";
    //如果字符串数组长度大于 1
    if ( modelList.length > 1)
    {
        //遍历服务器提示的所有型号
        for(var i = 0 ; i < modelList.length - 1 ; i++)
        {
            //对每个型号创建<div.../>元素
            var a = document.createElement("div");
            //使用<div.../>显示当前型号的型号名
            a.innerHTML = modelList[i];
            //添加当前型号提示
            mt.appendChild(a);
        }
        //选中第一项
        mt.firstChild.className = "selectTip";
        //显示型号提示元素
        if( tip != mt.value)
        {
            mt.show();
        }
```

```
    }
    else
    {
        //隐藏型号提示元素
        mt.hide();
    }
}
```

一旦服务器生成了响应，该回调函数就负责将响应在 HTML 页面中动态显示，图 10.5 显示了用户型号提示的效果。

如图 10.5 所示是用户单击了一次向下键后的效果，model 输入框与 brand 输入框的向上键、向下键和回车事件共用同一个处理函数。因此用户已经可以通过向上、向下键来选择合适的型号。一旦用户在型号输入框中单击回车键，选中的型号将会自动输入到型号输入框中，并向服务器发送获取该型号详细信息的异步 Ajax 请求。

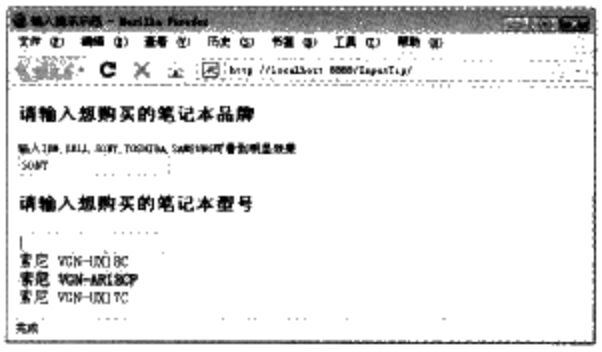


图 10.5　型号提示

➤➤10.6.4　根据型号显示描述

一旦用户选择了合适的型号，将发出异步 Ajax 请求，该请求负责根据型号名获取到该型号的详细描述。因为这种描述将直接在 HTML 元素中输出，无须使用任何回调函数解析服务器响应，因此使用 Ajax.Updater 可以更简单地完成异步请求的发送。该请求的函数代码如下：

程序清单：codes\10\InputTip\js\tip.js

```
function getDetail()
{
    //发送请求的 URL
    var url = 'getDetail.do';
    var params = "model=" + $F('model');
    //创建 Ajax.Updater 对象，发送异步请求
    new Ajax.Updater(
    //指定更新 modelDetail 元素
    'modelDetail',
    url,
    {
        method: 'post',
        parameters: params,
        //使用匿名函数作为回调函数，该函数控制详细信息<div.../>元素的出现
        onComplete:function()
        {
            $('modelDetail').show();
        }
    });
}
```

一旦用户选中了合适的型号，该型号对应的详细信息将出现在该页面的下面。图 10.6 显示了出现型号详细信息的效果。

经过这一系列的过程，该 Ajax 应用已经基本完成了，这种应用能极好地改善用户的浏览体验，界面呈现了相当的智能。但还可以增加一些改善，当页面向服务器发送请求时，应让用户知道请求已经发出，且正在等待服务器响应，如果服务器响应出错，也应该给用户一个简单的提示，这些都可以借助 Prototype 库的 Ajax.Responders 对象完成。

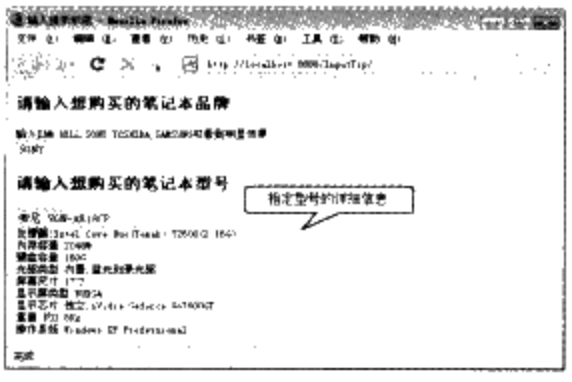


图 10.6　指定型号对应的详细信息

➤➤10.6.5 注册 Ajax 事件监听器

使用 Ajax.Responders 类可以为整个 Ajax 应用注册全局的事件监听器，用以提示用户 Ajax 交互当前的状态。为了检测到当前 Ajax 事件，必须先实现一个全局 Ajax 事件监听器。该监听器代码如下：

程序清单：codes\10\InputTip\js\tip.js

```
var myGlobalHandlers =
            {
    //在创建 XMLHttpRequest 对象时，触发该匿名函数
    onCreate: function()
    {
        //显示 loading 元素
        $('loading').show();;
    },
    //服务器响应出错时触发该匿名函数
    onFailure:function()
    {
        alert('对不起!\n 页面加载出错!');
    },
    //当 Ajax 交互完成时，触发该匿名函数
    onComplete: function()
    {
        //如果没有正在交互的 Ajax 请求
        if(Ajax.activeRequestCount == 0)
        {
                //隐藏 loading 元素
                $('loading').hide();
        }
    }
};
```

上面的代码主要用于检测是否有正在进行的 Ajax 请求，如果有正在进行的 Ajax 请求，则显示'loading'元素，否则隐藏该元素。该元素是一个标示 Ajax 交互的 GIF 动画图片，通过该图片告诉浏览者正在与服务器通信，如果服务器响应失败，也会弹出一个提示框。一旦实现了该 Ajax 事件的全局监听器，就可以使用 Ajax.Responders 来注册该监听器，代码如下：

```
//ajax 事件绑定
Ajax.Responders.register(myGlobalHandlers);
```

图 10.7 显示了客户端页面与服务器交互时出现的 loading 图片。

图 10.7 中的 loading 图片可以提示浏览者：目前正在与服务器交互，浏览者需要短暂的等待才会看到服务器响应，从而避免浏览者缺乏耐心而刷新页面。

图 10.7 Ajax 交互未完成时的页面效果

10.7 本章小结

本章以一个简单的自动完成页面示范了如何开发一个完整的 Ajax 应用，这个应用不是为了 Ajax 而 Ajax，它以传统的轻量级 Java EE 应用为蓝本，对传统的轻量级 Java EE 应用进行了改进，致力于改善用户的浏览体验。本章的应用示范了如何将传统轻量级 Java EE 应用与 Ajax 技术完美结合。

本章简单介绍了传统 Java EE 应用的 Domain Object 的设计，以及 DAO 组件、业务逻辑组件的实现，以结构良好、分层清晰的架构实现了应用的底层；本章还详细介绍了 Prototype 库的三个 Ajax 工具类的实际用途：Ajax.Request 用于发送支持回调的 Ajax 请求，Ajax.Updater 用于发送无须处理服务器响应的 Ajax 请求，Ajax.Responders 用于注册全局的 Ajax 事件监听器。

CHAPTER 11

第11章 jQuery 库详解

本章要点

- 理解 jQuery 的优雅的设计
- 下载和安装 jQuery
- 获取 jQuery 的工具函数
- jQuery 访问 DOM 对象支持的选择器和限定词
- jQuery 访问类数组对象的工具方法
- jQuery 过滤类数组的工具方法
- jQuery 提供的类 DOM 导航的工具方法
- jQuery 命名空间包含的方法
- jQuery 数据存储的方法
- jQuery 操作通用属性的方法
- jQuery 操作 CSS 样式的方法
- jQuery 更新 HTML 页面内容的方法
- jQuery 提供的事件编程支持
- jQuery 提供的动画效果方法
- 使用 load 方法发送异步请求
- 使用 jQuery.ajax 控制异步请求的各种选项
- 使用 get/post 方法发送异步请求

jQuery 库是另一个非常优秀的 JavaScript 库，与 Prototype 库有如下相似之处：①jQuery 也是一个纯粹的 JavaScript 代码库，完全可以在其他的 Web 应用中使用。②jQuery 也能兼顾主流浏览器标准，能跨浏览器运行。与 Prototype 库不同的是，jQuery 采用了另一种更优雅的解决方法，在使用 jQuery 库之后，开发者操作的对象不再是原始的 DOM 元素，而是 jQuery 对象。通过这种方式就使开发者无须理会不同浏览器处理 DOM 对象时存在的差异，而是直接以 jQuery 对象所支持的属性和方法操作 DOM 对象。

除此之外，jQuery 还提供了一些工具方法用来简化数组、字符串的操作。jQuery 库对 Ajax 也提供了良好的支持，使用 jQuery 同样无须手动创建 XMLHttpRequest 对象，只需指定发送请求的 URL 和处理服务器响应的回调函数即可，jQuery 将负责完成剩下的工作。

11.1 jQuery 入门

jQuery 与 Prototype 的最大区别在于设计思想不同，如果读者理解了 jQuery 的设计，那么使用 jQuery 就是比较简单的事情了。因此，下面将首先从 jQuery 的设计讲起。

➤➤11.1.1 理解 jQuery 的设计

几乎每个初次学习 jQuery 的读者都会发现，jQuery 也提供了一个$()函数，看上去和 Prototype 提供的$()几乎一样：都用于获取页面的 DOM 元素。看下面的 jQuery 入门示例：

程序清单：codes\11\11.1\jqueryQs.html

```
<div id="lee"></div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    var target = $("#lee")
    target.html("我要学习 jQuery")
          .height(60)
          .width(160)
          .css("border" , "2px solid black")
          .css("background-color" , "#ddddff")
          .css("padding" , 20);
</script>
```

上面的代码中第一行粗体字代码使用$()函数获取了一个 DOM 对象 target，后面的粗体字代码依次调用 height、width、css 等方法处理该对象，程序的运行结果如图 11.1 所示。

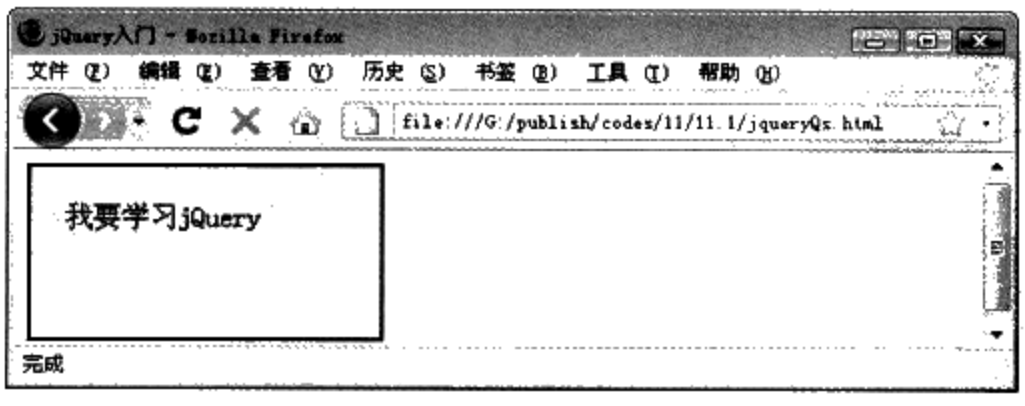


图 11.1 使用 jQuery 操作页面元素

看到这个运行结果，我们可以发现使用 jQuery 来动态操作 DHTML 页面非常简单。但读者很容易感到疑惑：程序中那些粗体字代码如何理解？

经过上面这个程序，我们应该可以明白jQuery的设计：使用jQuery之后，JavaScript操作的不再是HTML元素对应的DOM对象，而是包装DOM对象的jQuery对象。JavaScript通过调用jQuery对象的方法来改变它所包装的DOM对象的属性，从而实现动态更新HTML页面。由此可见，使用jQuery动态更新HTML页面只需如下两个步骤：

（1）获取jQuery对象。jQuery对象通常是对DOM对象的包装。

（2）调用jQuery对象的方法来改变自身。当jQuery对象被改变时，jQuery包装的DOM对象随之改变，HTML页面的内容也就随之更新了。

还有一点需要指出的是，jQuery很多改变自身属性的方法（类似于Java里的setter方法）都有返回值，就是返回该对象本身，因此可以连续多次调用改变自身属性的方法。例如在上面的程序中连续调用height、width、css方法来改变target对象。

以上就是使用jQuery的基本思路，开发者只要掌握两点即可：①获取jQuery对象；②jQuery有哪些可用的方法，这也是本章将要详细介绍的。当然，下面还是将从jQuery的下载和安装开始讲起。

➤➤11.1.2 下载和安装jQuery

由于jQuery也是一个纯粹的JavaScript库，因此下载和安装jQuery与Prototype并没有太大的不同之处。登录jQuery的官方站点http://jquery.com，在该站点可下载开发中的jQuery的最新版本。笔者成书之时，jQuery的最新版本是1.2.6，这是本书所使用的jQuery版本。

下载jQuery时有三个选项：

- Minified：该版本是去除注释后的jQuery库，文件体积较小，开发Ajax应用通常推荐使用该版本。
- Packed：该版本在Minified版本基础上做了进一步压缩，文件体积更小。但使用该版本需要客户端浏览器先执行解压缩。
- Uncompressed：该版本的jQuery库没有压缩，而且保留了注释，有兴趣研究jQuery源代码的读者可以下载该版本。

除此之外，在浏览器地址栏中输入http://docs.jquery.com/，可看到jQuery库的在线文档，主要包括Get Start（快速入门）和jQuery API Reference（API参考）两个部分，读者也可参考该文档来学习jQuery的用法。

与Prototype类似的是，jQuery库也不需要增加额外的环境变量、配置文件，只需要在HTML页面中导入jQuery的JavaScript文件即可。

一旦导入了jQuery库，开发者就可以在自己的脚本中使用jQuery库提供的功能。为了导入jQuery类库，应在HTML页面的开始位置增加如下代码：

```
<!-- 引入一个JavaScript函数库 -->
<script type="text/javascript"src="jquery-1.2.6.min.js">
</script>
```

上面的代码中src属性可能会有小小的变化，如果jquery-1.2.6.min.js文件名被改变了，或者它与HTML页面并不是放在同一个路径下，则应该在上述代码的基础上做对应的修改，让src属性指向

jquery-1.2.6.min.js 脚本文件所在的位置。

➤➤11.1.3 让 jQuery 与其他 JavaScript 库共存

如果读者需要让 jQuery 和其他 JavaScript 库（如 Prototype）共存，则有一个小小的问题需要解决：就是$()函数。由于 jQuery 里$()函数的功能很强大，而且返回一个 jQuery 对象，而 Prototype 的$()返回的是一个 DOM 对象，因此必然引起冲突。

为了解决 jQuery 和其他 JavaScript 库中$()函数的冲突，需要取消 jQuery 的$()函数，为此 jQuery 提供了如下方法：

```
//取消 jQuery 中的$()函数
jQuery.noConflict();
```

建议将上面的粗体字代码放在 JavaScript 代码的第一行，这行代码就会取消 jQuery 的$()函数——其实只是取消了 jQuery()函数的$()别名，因此我们依然可以使用 jQuery()来代替原来的$()。

除此之外，多次重复书写 jQuery()也是很烦琐的事情，jQuery 还允许开发者为 jQuery()指定自己的别名，如以下代码所示：

程序清单：codes\11\11.1\jqueryQs2.html

```
//给 jQuery()函数指定别名为 lee
var lee = jQuery.noConflict();
var target = lee("#lee")
target.html("我要学习 jQuery")
      .height(60)
      ...
```

正如在第一行粗体字代码中所见到的，程序为 jQuery 函数指定别名为 lee，这就允许程序后面使用 lee()函数来代替 jQuery()函数了，如程序中第二行粗体字代码所示。

通过这种方式，我们可以让 jQuery 和其他 JavaScript 库共存。

11.2 获取 jQuery 对象

从前面的介绍可以看出，使用 jQuery 的第一步就是获取 jQuery 对象，jQuery 库中获取 jQuery 对象主要有如下两种方式：

- 使用$()函数或用 jQuery 对象提供的、利用父子关系来返回的 jQuery 对象。
- jQuery 对象的调用方法改变自身后将再次返回该 jQuery 对象。

本节主要介绍第一种获取 jQuery 对象的方式，即使如此，$()获取 jQuery 对象的方式仍是如此之多，它既支持 CSS1-3 选择器来访问 DOM 元素，也支持使用 XPath 语法来访问 DOM 元素——$()函数会将这些 DOM 元素包装成 jQuery 对象后返回。

➤➤11.2.1 jQuery 核心函数

正如前面指出的，jQuery()函数是获取 jQuery 对象的根本途径，该函数主要有如下用法：

- jQuery(expression, [context])：该函数获取 expression 对应的 DOM 对象包装成的 jQuery 对象。其中 expression 既支持 CSS1-3 选择器，也支持 XPath 语法，功能非常丰富。由于 expression 表达式可能对应单个 DOM 元素，也可能对应多个 DOM 元素，因此该方法可能返回单个 jQuery 对象，也可能返回 jQuery 对象数组。context 是个可选参数，如果指定了该参数，表明仅获取 context 的子元素。
- jQuery(html, [ownerDocument])：该函数根据 html 参数（该参数是个 HTML 字符串）创建一个或多个 DOM 对象，返回包装这些 DOM 对象的 jQuery 对象。其中 ownerDocument 是可选

参数，指定使用 ownerDocument（document 对象）来创建 DOM 对象。

- jQuery(elements)：将一个或多个 DOM 元素包装为 jQuery 对象。elements 既可是单个的 DOM 对象，也可是多个 DOM 对象。该方法返回包装这些 DOM 对象的 jQuery 对象。
- jQuery(callback)：这种用法是$(document).ready()的缩写，其中 callback 指定一个函数，在页面加载完成时 callback 函数被激发。该函数返回页面 document 对象包装成的 jQuery 对象。

下面的代码示范了 jQuery 函数的四种用法：

程序清单：codes\11\11.2\$.html

```
<div id="lee"></div>
<div id="yeeku"></div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//获取所有<div.../>标签对应的 DOM 对象
$("div").append("新增的内容");
//使用 HTML 字符串创建一个 DOM 对象，并将其添加到 body 元素内
$("<input type='button' value='单击我'/>").appendTo(document.body);
//直接将一个 DOM 对象包装成 jQuery 对象
$(document.getElementById('lee'))
    .css("background-color" , "#aaffaa")
    .css("border" , "1px solid black");
//指定页面加载完成后执行指定函数
$(function()
{
    alert("页面加载完成");
});
</script>
```

在浏览器里浏览上述页面，可看到如图 11.2 所示结果。

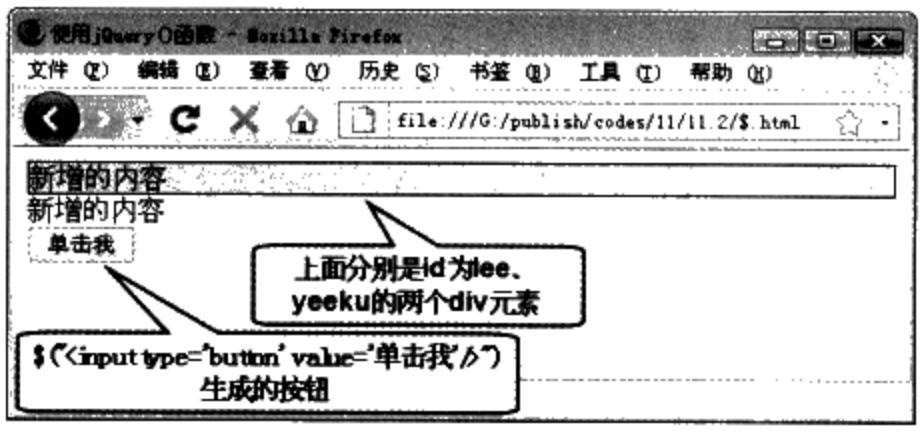


图 11.2 jQuery()函数的四种用法

上面的粗体字代码在使用$()函数获取了 jQuery 对象之后，还调用了 jQuery 对象的 appendTo、append 等方法，这些方法在后面会有更详细的介绍，此处不再赘述。值得指出的是，jQuery 的第一种用法 jQuery(expression, [context])，需要指定一个 expression，该表达式能支持的形式相当多，下面将详细介绍这些用法。

➤➤11.2.2 以 CSS 选择器访问 DOM 元素

本书第 5 章已经详细介绍了 CSS 所支持的各种常用选择器，每个 CSS 选择器可以对应一个或多个 HTML 元素，如果以该 CSS 选择器作为参数，$(selector)将可以获取该选择器对应的一个或多个 HTML 元素包装成的 jQuery 对象。

与前面的 CSS 选择器类似的是，$()可支持如下几种参数形式：

- #id：返回指定 id 对应的 HTML 元素包装成的 jQuery 对象。类似于 CSS 中 ID 选择器的功能。

- tagName：返回所有 tagName 标签对应的所有 HTML 元素包装成的 jQuery 对象数组。类似于 CSS 中属性选择器不指定任何属性的功能。
- tagName[attribute]：返回由 tagName 标签生成且包含 attribute 属性的所有 HTML 元素包装成的 jQuery 对象数组。以下几个都类似于 CSS 中属性选择器的功能。
- tagName[attribute=value]：返回由 tagName 标签生成且 attribute 属性等于 value 的所有 HTML 元素包装成的 jQuery 对象。
- tagName[attribute!=value]：返回由 tagName 标签生成且 attribute 属性不等于 value 的所有 HTML 元素包装成的 jQuery 对象。
- tagName[attribute^=value]：返回由 tagName 标签生成且 attribute 属性值以 value 开头的所有 HTML 元素包装成的 jQuery 对象。
- tagName[attribute$=value]：返回由 tagName 标签生成且 attribute 属性值以 value 结尾的所有 HTML 元素包装成的 jQuery 对象。
- tagName[attribute*=value]：返回由 tagName 标签生成且 attribute 属性值包含 value 的所有 HTML 元素包装成的 jQuery 对象。
- tagName[attributeFilter1][attributeFilter2]...：返回由 tagName 标签生成且具有 attributeFilter1、attributeFilter2 等任意一个属性特征的所有 HTML 元素包装成的 jQuery 对象。其中 attributeFilter1、attributeFilter2 支持前面任意一个有效的属性定义。
- .className：返回所有 class 属性为 className 的所有 HTML 元素包装成的 jQuery 对象。类似于 CSS 中 class 选择器的功能。
- *：返回所有 HTML 元素包装成的 jQuery 对象。这个较为少用。
- selector1, selector2...selectorN：同时指定多个选择器，返回匹配任何一个选择器的所有 HTML 元素包装成的 jQuery 对象。
- outerSelector innerSelector：返回 outerSelector 选择器之内的所有 innerSelector（不管处于多少层之内）对应的 HTML 元素包装成的 jQuery 对象。类似于 CSS 中的包含选择器。
- parentSelector>childSelector：返回直接位于 parentSelector 选择器之内第一层 childSelector 对应的 HTML 元素包装成的 jQuery 对象。类似于 CSS 中的子元素选择器。
- prevSelector+nextSelector：返回紧跟在 prevSelector 之后的第一个 nextSelector 对应的 HTML 元素包装成的 jQuery 对象。
- prevSelector~siblingsSelector：返回位于 prevSelector 之后的所有 siblingsSelector 对应的 HTML 元素包装成的 jQuery 对象。

注意：

上面的很多选择器都可同时匹配多个 HTML 元素，而使用上面这些选择器作为$()函数的参数时都将简单地返回 jQuery 对象。这是因为 jQuery 对象不只可以包装单个的 DOM 元素，也可包装多个 DOM 元素，此时的 jQuery 对象有点类似于数组。如果程序直接操作包装多个 DOM 元素的 jQuery 对象，那么这些 DOM 元素都会随之改变。

如下程序示范了上面几种选择器的用法：

程序清单：codes\11\11.2\selector.html

```
<ul>
    <li id="java">疯狂 Java 讲义</li>
    <li id="javaee" class="test">轻量级 Java EE 企业应用实战</li>
    <li id="ajax">疯狂 Ajax 讲义</li>
    <li id="xml">疯狂 XML 讲义</li>
    <li id="ejb">经典 Java EE 企业应用实战</li>
```

```
    <li><span id="workflow">疯狂 Workflow 讲义</span></li>
</ul>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//获取 id 为 java 的元素
$("#java").append("<b> 是 id 为 java 的元素</b>");
//获取所有包含 id 属性的<li.../>元素，为它们增加背景色
$("li[id]").css("background-color" , "#bbbbff");
//获取 class 属性为 test 的元素，并为它们增加边框
$(".test").css("border" , "3px dotted black");
//同时获取 id 为 xml、workflow 的元素
$("#xml,#workflow").append("<b>是 id 为 xml、workflow 其中之一的元素</b>");
//获取 ul 之内 id 为 workflow 的元素
$("ul #workflow").append("<br /><b>位于 ul 之内、id 为 workflow 的子元素</b>");
//获取 ul 之内 id 为 ajax 的直接子元素
$("ul>#ajax").append("<b>ul 之内、id 为 ajax 的子元素</b>")
    .css("border" , "2px solid black");
//获取 id 为 ajax 之后的所有 li 元素
$("#ajax~li").css("background-color" , "#ff5555");
</script>
</body>
```

上面的程序示范了$()所支持的几种选择器的用法，在浏览器中浏览该页面，将可看到如图 11.3 所示效果。

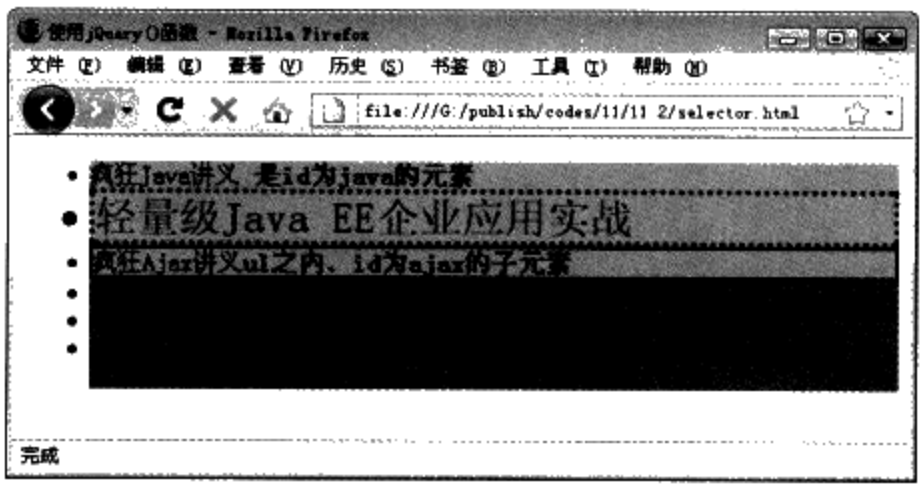


图 11.3　使用$()支持的各种选择器

经过上面的介绍，我们明白了$()函数的基本用法。从上面的介绍可以看出，$()函数支持的许多选择器都会一次返回多个 HTML 元素对应的 jQuery 对象，为此 jQuery 还支持在原有的 Selector 上增加额外的限定。

➤➤11.2.3　选择器的附加限定词

下面介绍的限定词通常会与前面介绍的选择器结合使用，前面的选择器往往可以匹配一个以上的 HTML 元素，而以下限定词则用于增加额外的限制。

- :first：返回匹配指定选择器第一个 HTML 元素包装成的 jQuery 对象。
- :last：返回匹配指定选择器最后一个 HTML 元素包装成的 jQuery 对象。
- :not(selector)：返回匹配指定选择器但去除 selector 选择器匹配的所有 HTML 元素包装成的 jQuery 对象。
- :even：返回匹配指定选择器的索引为奇数的 HTML 元素包装成的 jQuery 对象。元素索引是从 0 开始的。
- :odd：返回匹配指定选择器的索引为偶数的 HTML 元素包装成的 jQuery 对象。

- :eq(index)：返回匹配指定选择器的索引为 index 的 HTML 元素包装成的 jQuery 对象。
- :gt(index)：返回匹配指定选择器的索引大于 index 的所有 HTML 元素包装成的 jQuery 对象。
- :lt(index)：返回匹配指定选择器的索引小于 index 的所有 HTML 元素包装成的 jQuery 对象。
- :header：返回匹配指定选择器，且必须是 h1、h2、h3 之类的标题元素。
- :animated：返回匹配指定选择器，且当前没有执行动画效果的 HTML 元素包装成的 jQuery 对象。
- :contains(text)：返回匹配指定选择器，且包含 text 文本的 HTML 元素包装成的 jQuery 对象。
- :empty：返回匹配指定选择器，且不包含任何内容（包含字符串也不行）的 HTML 元素包装成的 jQuery 对象。
- :has(selector)：返回匹配指定选择器，且包含 selector 对应 HTML 元素的所有 HTML 元素包装成的 jQuery 对象。

注意：

:has(selector)限定不是返回 selector 所匹配的 HTML 元素，而是返回包含 selector 所匹配的元素的 HTML 元素。

- :parent：返回匹配指定选择器，且包含子元素或者文本的所有 HTML 元素包装成的 jQuery 对象。
- :hidden：返回匹配指定选择器，且当前不可见的 HTML 元素包装成的 jQuery 对象。
- :visible：返回匹配指定选择器，且当前可见的 HTML 元素包装成的 jQuery 对象。

下面的代码示范了以上限定词的用法：

程序清单：codes\11\11.2\restrict.html

```
<ul>
    <li id="java">疯狂 Java 讲义</li>
    <li id="javaee" class="test">轻量级 Java EE 企业应用实战</li>
    <li id="ajax">疯狂 Ajax 讲义</li>
    <li id="xml">疯狂 XML 讲义</li>
    <li id="ejb">经典 Java EE 企业应用实战</li>
    <li><span id="workflow">疯狂 Workflow 讲义</span></li>
</ul>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//访问 ul 元素下第一个 li 子元素。
$("ul>li:first").append("<b> 是 ul 元素之内第一个 li 子元素</b>");
//访问 ul 元素之内没有 id 属性的 li 子元素
$("ul>li:not([id])").append("<b> 是 ul 元素之内、没有 id 属性 li 子元素</b>");
//访问 ul 元素之内索引为奇数的 li 子元素，并为它们添加背景色
$("ul>li:even").css("background-color", "#ccffcc");
//访问 ul 元素之内索引大于 4 的 li 子元素（元素索引从 0 开始）
$("ul>li:gt(4)").append("<br/><b> 是 ul 元素之内、索引大于 4 的 li 子元素</b>")
    .css("border" , "1px dashed black");
//访问 ul 元素之内且包含 span 元素的 li 子元素
$("ul>li:has('span')").append(
    "<br/><b> 是 ul 元素之内、且包含 span 元素的 li 子元素</b>");
//访问 li 元素之内且可见的 span 子元素
$("li>span:visible").append(
    "<b> 是 li 元素之内，且可见的 span 子元素</b>")
    .css("background-color" , "#bbbbbb");
</script>
```

上面的程序是前面的选择器和限定词一起作用的效果，在浏览器中浏览上述页面，可看到如图 11.4 所示效果。

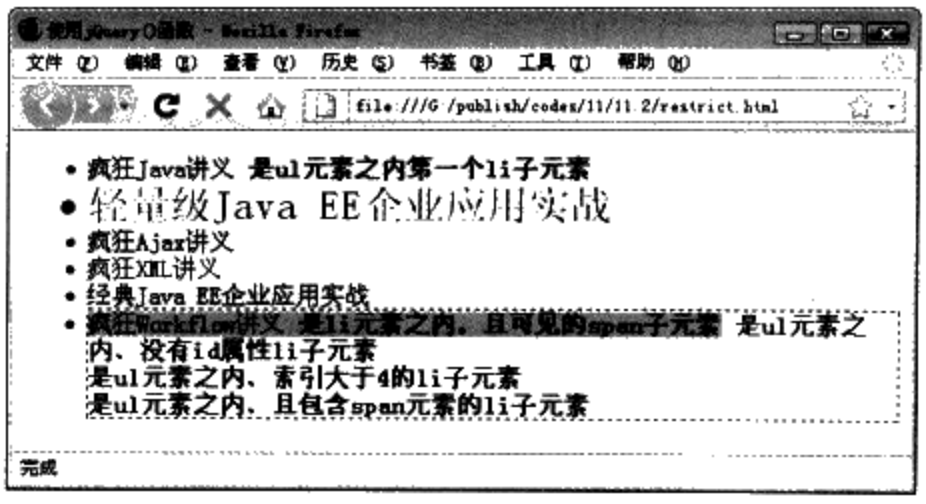


图 11.4　使用附加限定词的效果

下面的四个限定词中:first-child、:last-child 和:first、:last 等有点相似，但:nth-child 则比较有用：

- :nth-child(index/even/odd/equation)：返回匹配指定选择器的第 index 个 HTML 元素包装成的 jQuery 对象。该选项有如下四种用法：
  - :nth-child(n)：返回匹配指定选择器的第 n 个 HTML 元素包装成的 jQuery 对象，其中 n 为从 1 开始的元素索引。
  - :nth-child(even)：返回匹配指定选择器且索引为奇数的 HTML 元素包装成的 jQuery 对象。
  - :nth-child(odd)：返回匹配指定选择器且索引为偶数的 HTML 元素包装成的 jQuery 对象。
  - :nth-child(xn+m)：返回匹配指定选择器且索引为 xn+m 的 HTML 元素包装成的 jQuery 对象，其中 x、m 可变。例如 3n+1，则匹配索引为 1、4、7 等的元素。
- :first-child：返回匹配指定选择器的第一个 HTML 元素包装成的 jQuery 对象。
- :last-child：返回匹配指定选择器的最后一个 HTML 元素包装成的 jQuery 对象。
- :only-child：返回匹配指定选择器且是父元素中唯一的 HTML 元素（如果该父元素下有多个子元素则不会被匹配）的元素包装成的 jQuery 对象。

程序清单：codes\11\11.2\restrict2.html

```
<ul>
    <li id="java">疯狂 Java 讲义</li>
    <li id="javaee" class="test">轻量级 Java EE 企业应用实战</li>
    <li id="ajax">疯狂 Ajax 讲义</li>
    <li id="xml">疯狂 XML 讲义</li>
    <li id="ejb">经典 Java EE 企业应用实战</li>
    <li><span id="workflow">疯狂 Workflow 讲义</span></li>
</ul>
<span>疯狂 Java 联盟</span>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//访问页面中第 1、4、7...个 li 元素
$("li:nth-child(3n+1)").css("border" , "1px dashed black");
//访问页面中的 span 元素，且该 span 元素的父元素下仅包含该 span 元素
$("span:only-child()").append(" <b>是作为父元素唯一子元素的 span 元素</b>");
//测试:first 和:first-child 之间的关系
alert($("ul>li:first").html() == $("ul>li:first-child").html());
</script>
```

在浏览器中浏览上面的页面，将可以看到页面弹出对话框输出 true，这表明限定词:frist 和:first-child 的作用基本相似。在浏览器中可看到如图 11.5 所示效果。

图 11.5　限定词的效果

➤➤11.2.4　表单相关的选择器

下面各选择器专门用于匹配各种表单控件：

- :input：返回所有 input、textarea、select 和 button 元素包装成的 jQuery 对象。
- :text：返回所有 type="text"的 input 元素包装成的 jQuery 对象。
- :password：返回所有 type="password"的 input 元素包装成的 jQuery 对象。
- :radio：返回所有 type="radio"的 input 元素包装成的 jQuery 对象。
- :checkbox：返回所有 type="checkbox"的 input 元素包装成的 jQuery 对象。
- :submit：返回所有 type="submit"的 input 元素包装成的 jQuery 对象。
- :image：返回所有 type="image"的 input 元素包装成的 jQuery 对象。
- :reset：返回所有 type="reset"的 input 元素包装成的 jQuery 对象。
- :button：返回所有按钮元素（包括 type="button"的 input 元素）包装成的 jQuery 对象。
- :file：返回所有文件域包装成的 jQuery 对象。
- :hidden：返回所有不可见元素以及指定了 type="hidden"的 input 元素包装成的 jQuery 对象。

注意：

:hidden 选择器不仅可以匹配表单控件，而且可以匹配所有不可见的元素，包括<meta..../>等元素。

- :enabled：返回所有可用的（未指定 disabled="disabled"）的表单控件包装成的 jQuery 对象。
- :disabled：返回所有不可用的（指定了 disabled="disabled"）的表单控件包装成的 jQuery 对象。
- :checked：返回所有指定了 checked="checked"的表单控件包装成的 jQuery 对象。
- :selected：返回所有指定了 selected ="selected"的表单控件包装成的 jQuery 对象。

下面的程序示范了上述选择器的用法：

程序清单：codes\11\11.2\ formElement.html

```
<input id="user" type="text" /><br />
<input id="pass" type="password" /><br />
<textarea id="intro"></textarea><br />
<select id="gender">
<option value="male" selected="selected">男</option>
<option value="female">女</option>
</select><br />
<span style="display:none">疯狂 Java 讲义</span><br />
<input id="pass" type="checkbox" checked="checked" value="xx"/><br />
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
```

```
//获取所有的 input、textarea、button、select 元素
$(":input").val("test");
//获取所有指定了 selected="selected"的元素
$(":selected").css("border" , "2px dashed black");
//获取所有指定了 checked="checked"的元素
$(":checked").width(40).css("border" , "2px dotted black");
</script>
```

在浏览器里浏览该页面，可看到如图 11.6 所示效果。

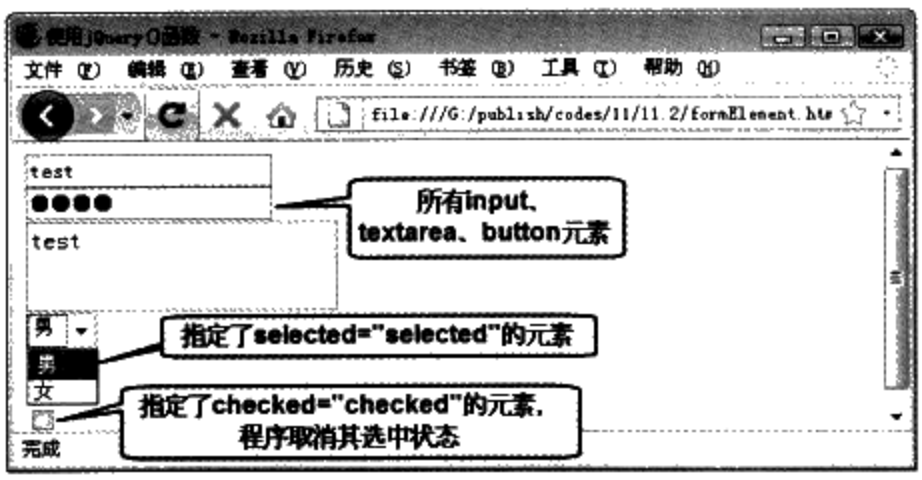


图 11.6　表单控件相关的选择器

11.3　jQuery 操作类数组的工具方法

很多时候，jQuery 的$()函数都返回一个类似数组的 jQuery 对象，例如$("div")将返回页面中所有<div.../>元素包装成的 jQuery 对象，这个 jQuery 对象实际上包含了多个<div.../>元素对应的 DOM 对象。在这种情况下，jQuery 提供了如下几个常用方法来操作类数组的 jQuery 对象：

- each(callback)：该方法是一个迭代器函数，它将使用 callback 函数迭代处理 jQuery 里包含的每个元素，在 callback 函数里使用 this 来代表当前正处理的 DOM 元素，如果想获取该 DOM 元素对应的 jQuery 对象，使用$(this)即可。callback 是一个形如 fn(i){}的函数，其中 i 代表 jQuery 里元素的索引，该索引从 0 开始。
- length：该属性返回 jQuery 里包含的 DOM 元素的个数。
- eq(position)：该方法返回 jQuery 里包含的第 position +1 个 DOM 元素包装成的 jQuery 对象。
- get()：该方法返回 jQuery 里包含的所有 DOM 元素组成的数组。
- get(index)：该方法返回 jQuery 里包含的第 index+1 个 DOM 元素（第一个元素的索引为 0）。

注意：

上面的 2 个方法非常重要，可以将 jQuery 对象再次恢复成 DOM 对象。根据前面的介绍我们知道，jQuery 的思路是把所有 DOM 对象包装成 jQuery 对象来处理，这种方式简单优雅，但总有些地方有失灵活。如果开发者需要操作 DOM 元素，则可通过这两个方法把 jQuery 对象转换成 DOM 对象。尤其需要指出的是，get()方法总是返回一个数组——即使原始的 jQuery 对象里只有一个 DOM 对象，调用 jQuery 对象的 get()方法也将返回一个长度为 1 的数组。

- index(subject)：该方法返回 jQuery 里 subject 的索引，其中 subject 既可以是 jQuery 里包含的多个 DOM 对象之一，也可以是任一 DOM 对象包装成的 jQuery 对象。

下面的程序示范了如何使用这些工具方法来操作类数组的 jQuery 对象：

程序清单：codes\11\11.3\arrayMethod.html

```
<div>
    <div id="java">疯狂 Java 讲义</div>
```

```
        <div id="javaee">轻量级 Java EE 企业应用实战</div>
        <div id="ajax">疯狂 Ajax 讲义</div>
        <div id="xml">疯狂 XML 讲义</div>
        <div id="ejb">经典 Java EE 企业应用实战</div>
        <div id="workflow">疯狂 Workflow 讲义</div>
    </div>
    <script type="text/javascript" src="../jquery-1.2.6.min.js">
    </script>
    <script type="text/javascript">
    //获取 div 之内所有的 div 元素，并迭代处理每个元素
    $("div>div").each(function(i)
    {
        this.innerHTML += " 添加的内容" + i;
    })
    //返回 div 之内的所有 div 元素的个数，下面将输出 6
    alert($("div>div").length);
    //获取 div 之内的第四个 div 元素包装成的 jQuery，下面将输出“疯狂 XML 讲义...”
    alert($("div>div").eq(3).html());
    //获取 div 之内的第二个 div 元素，下面将输出“轻量级 Java EE 企业应用实战...”
    alert($("div>div").get(1).innerHTML);
    //获取 id 为 java 的 div 元素。注意：$("#java").get()返回一个数组
    alert($("#java").get()[0].innerHTML);
    //所有 div 元素之内 id 为 ejb 的 div 的索引，下面将输出 4
    alert($("div>div").index($("#ejb").get(0)));
    </script>
```

上面的程序中粗体字代码已经清楚地列出了各工具方法的用法，并给出了各方法的输出结果和原因，读者可对照该程序来掌握这些方法的用法。

经过前面的介绍，我们知道 jQuery 对象和 DOM 对象之间可按如图 11.7 所示方式进行转换。

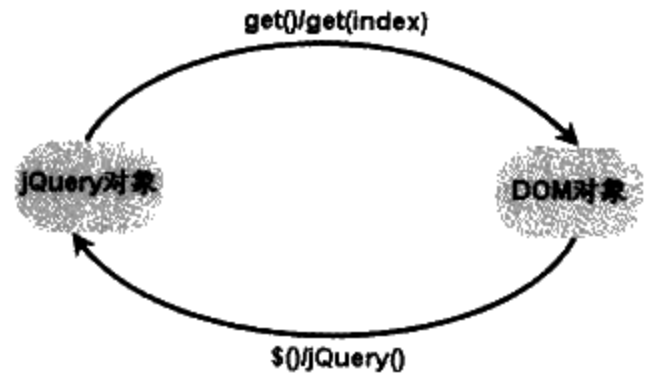


图 11.7　jQuery 对象和 DOM 对象的转换关系

➤➤11.3.1　过滤相关方法

下面是一组对类数组的 jQuery 对象进行过滤的方法，这些方法将会过滤到 jQuery 对象里包含的部分 DOM 对象。假如某个 jQuery 对象里包含了 5 个 DOM 对象，调用这些过滤相关方法之后，该 jQuery 对象里可能就只包含 3 个 DOM 对象了。

- filter(expr)：从 jQuery 对象里删除所有不匹配 expr 的 DOM 对象。其中 expr 可以是任意合法的选择器、限定词。
- filter(fn)：该方法是一个迭代器函数，它将使用 fn 函数迭代处理 jQuery 里包含的每个元素，在 fn 函数里使用 this 来代表当前正处理的 DOM 元素，如果想获取该 DOM 元素对应的 jQuery 对象，使用$(this)即可。fn 是一个形如 fn(i){}的函数，其中 i 代表 jQuery 里元素的索引，该索引从 0 开始。如果当前元素传入 fn 函数后返回 true，则该元素被保留；否则将被删除。
- is(expr)：用 expr 来检查该 jQuery 对象包含的元素集合，如果其中任意一个元素符合 expr 就返回 true。如果没有元素符合，或者表达式无效，就返回 false。其中 expr 可以是任意合法的选择器、限定词。
- map(callback)：该方法用于将 jQuery 对象里包含的一系列 DOM 对象转换成其他元素（这些元素既可包含原始 DOM 对象，也可不包含原始 DOM 对象）。callback 函数会依次处理 jQuery 里包含的每个 DOM 对象，每次函数执行的返回值将作为 jQuery 对象里包含的新元素。
- not(expr)：从 jQuery 对象里删除所有匹配 expr 的 DOM 对象。其中 expr 可以是任意合法的选

择器、限定词。该方法与 filter(expr)方法的效果完全相反。

- slice(start, end)：返回 jQuery 里索引从 start 开始到 end 结束的 DOM 元素组成的 jQuery 对象。

下面的程序示范了过滤相关方法的用法：

程序清单：codes\11\11.3\filter.html

```
<div>
    <div id="java">疯狂 Java 讲义</div>
    <div id="javaee">轻量级 Java EE 企业应用实战</div>
    <div id="ajax">疯狂 Ajax 讲义</div>
    <div id="xml">疯狂 XML 讲义</div>
    <div id="ejb">经典 Java EE 企业应用实战</div>
    <div id="workflow">疯狂 Workflow 讲义</div>
</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//对 div 之内的 div 元素进行过滤，必须满足 id 为 ajax
$("div>div").filter("#ajax").css("background-color" , "#aaaaaa");
//对 div 之内的 div 元素进行过滤，要求 div 内的字符串长度大于 8
$("div>div").filter(function()
{
    return this.innerHTML.length > 8;
}).css("border" , "1px solid black");
//对 div 之内的 div 元素进行过滤，必须满足 id 不为 ajax
$("div>div").not("#ajax").css("font-weight" , "bold");
//对 div 之内的 div 元素进行过滤，取出索引从 3 到 5 的元素
$("div>div").slice(3 , 5).height(40);
//将 div 之内的 div 元素映射成另一个数组，t 的值是[0,1,2,3,4,5]
$("div>div").map(function(i)
{
    return i;
});
</script>
```

上面的程序中不同功能的方法分别使用了不同的 CSS 样式，在浏览器中浏览该页面，可看到如图 11.8 所示效果。

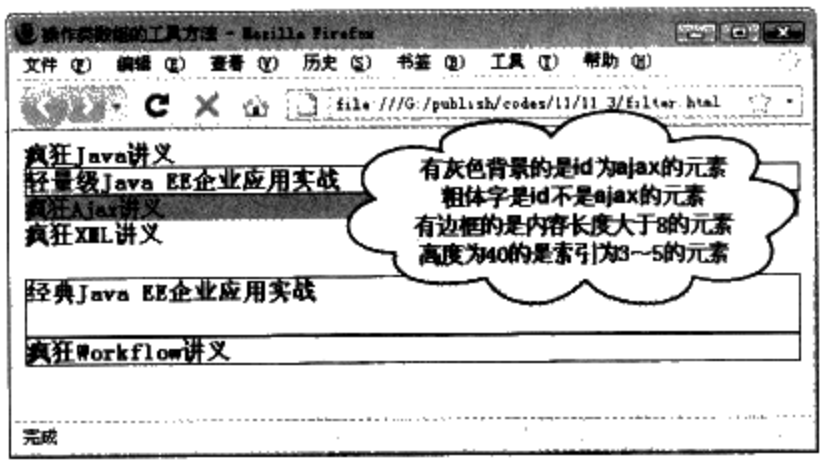


图 11.8　使用 jQuery 的过滤相关方法

➤➤11.3.2　仿 DOM 导航的相关方法

在 DOM 模型里，我们可以利用节点之间的父子关系进行导航，通过这种导航关系可以找到当前节点的兄弟节点、父节点、子节点等，DOM 模型的导航关系明了，但用起来依然比较烦琐。jQuery 进一步简化了这种导航关系，在 jQuery 中可以利用如下方法来找到当前 jQuery 对象（可能包含一个或多个 DOM 对象）的兄弟节点、父节点、子节点对应的 jQuery 对象。

- add(expr)：为原来的 jQuery 对象添加新的 DOM 元素，其中 expr 既可以是任何合法的选择器

或限定词；也可以是原始的 HTML 代码，（该方法将会把 HTML 代码转化为 DOM 对象后添加到 jQuery 里）；还可以是未经包装的 DOM 对象。

- children([expr])：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）之内的全部后代元素。如果指定了 expr 选择器，则只查找匹配 expr 选择器的后代元素。
- contents()：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）之内的全部内容，包括 DOM 元素和文本。
- find(expr)：查找处于当前 jQuery 对象（实际是该对象包含的 DOM 对象）之内能匹配 expr 选择器的所有后代元素。其中 expr 可以是任何合法的选择器。返回这些 DOM 元素包装成的 jQuery 对象。
- next([expr])：查找紧跟当前 jQuery 对象（实际是该对象包含的 DOM 对象）之后的元素。如果指定了 expr 选择器，则该元素必须匹配 expr 选择器。返回这些 DOM 元素包装成的 jQuery 对象。
- nextAll([expr])：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）之后的所有兄弟元素。如果指定了 expr 选择器，则只找出匹配 expr 选择器的兄弟元素。返回这些 DOM 元素包装成的 jQuery 对象。
- parent([expr])：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）的父元素。如果指定了 expr 选择器，则该父元素还必须匹配 expr 选择器。返回这些 DOM 元素包装成的 jQuery 对象。
- parents([expr])：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）的所有祖先元素。如果指定了 expr 选择器，则只找出匹配 expr 的祖先元素。返回这些 DOM 元素包装成的 jQuery 对象。
- prev([expr])：查找紧跟当前 jQuery 对象（实际是该对象包含的 DOM 对象）之前的元素。如果指定了 expr 选择器，则该元素必须匹配 expr 选择器。返回这些 DOM 元素包装成的 jQuery 对象。
- prevAll([expr])：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）之前的所有兄弟元素。如果指定了 expr 选择器，则只找出匹配 expr 选择器的兄弟元素。返回这些 DOM 元素包装成的 jQuery 对象。
- siblings([expr])：查找当前 jQuery 对象（实际是该对象包含的 DOM 对象）前后的所有兄弟元素。如果指定了 expr 选择器，则只找出匹配 expr 选择器的兄弟元素。返回这些 DOM 元素包装成的 jQuery 对象。

提 示：

siblings([expr]) 方法返回的结果相当于 prevAll([expr])和 nextAll([expr])方法返回结果的总和。

下面的程序示范了上面这些 DOM 导航相关方法的用法：

程序清单：codes\11\11.3\find.html

```
<div>
    <div id="java">疯狂 Java 讲义</div>
    <div id="javaee">轻量级 Java EE 企业应用实战</div>
    <div id="ajax">疯狂 Ajax 讲义</div>
    <div id="xml">疯狂 XML 讲义</div>
    <div id="ejb">经典 Java EE 企业应用实战</div>
    <div id="workflow">疯狂 Workflow 讲义</div>
</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//获取 div 之内的所有内容（包括节点和文本），实际返回 div 下的 6 个子 div
$("div").contents().css("background-color" , "#dddddd");
//获取 id 为 ajax 的节点的下一个兄弟节点
$("#ajax").next().css("border" , "2px dotted black");
```

```
//获取 id 为 ajax 的节点的上一个兄弟节点
$("#ajax").prev().css("border" , "2px solid black");
//获取 id 为 ajax 的节点的 id 为 java 的兄弟节点
$("#ajax").siblings("#java")
    .append("<b> 是 ID 为 ajax 的节点的兄弟节点（且其 id 为 java）</b>");
//取出所有 div 元素的父元素，将会输出 body 元素和一个 div 元素
$("div").parent().each(function()
{
    alert($(this).html());
});
</script>
```

在浏览器中浏览该页面，将可看到弹出2个对话框，分别输出<body.../>元素内容和包含6个<div.../>的父<div.../>的内容，并可以在浏览器中看到如图 11.9 所示效果。

图 11.9　使用 jQuery 的仿 DOM 导航方法

从上面的程序可以看出，使用 jQuery 的仿 DOM 导航方法可以更简单、更便捷地访问当前节点的兄弟节点、父节点和子节点，而且这些方法的返回值依然是 jQuery 对象，因此可以直接调用 jQuery 对象提供的工具方法。

➤➤11.3.3　链接方法

前面的过滤、导航等方法都会对原有的 jQuery 对象进行“破坏”——通常都会减去原 jQuery 对象中包含的部分 DOM 对象。下面的两个方法则可以不同方式找到前一次“破坏”操作之前的 jQuery 对象。

- andSelf()：该方法通常与前面的查找方法结合使用，作用是将查找之前的结果和查找之后的结果混合在一起。
- end()：该方法通常也是和前面的过滤、查找方法结合使用，用于将 jQuery 对象恢复到上一次执行过滤、查找方法之前的状态。

提　示：

end()方法的作用有点类似于“撤销”操作，在对某个 jQuery 对象调用 end()方法之后，该 jQuery 对象的状态将恢复到调用 end()前一个方法之前的状态。

如下的代码示范了这两个方法的用法：

程序清单：codes\11\11.3\undo.html

```
<div>
    <div id="java">疯狂 Java 讲义</div>
    <div id="javaee">轻量级 Java EE 企业应用实战</div>
    <div id="ajax">疯狂 Ajax 讲义</div>
    <div id="xml">疯狂 XML 讲义</div>
    <div id="ejb">经典 Java EE 企业应用实战</div>
    <div id="workflow">疯狂 Workflow 讲义</div>
</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
```

```
<script type="text/javascript">
//获取 id 为 ajax 的节点的下一个兄弟节点，再将 id 为 ajax 的节点与此连为一体
//实际返回 id 为 ajax 的节点以及 id 为 ajax 的下一个节点
$("#ajax").next().andSelf().css("border" , "2px solid black");
//先获取 ajax 节点的下一个节点，再使用 end()方法重新获取之前的 ajax 节点
$("#ajax").next().end().css("background-color" , "#ffaaaa");
</script>
```

11.4 jQuery 支持的方法

前面介绍的都是如何获取 jQuery 对象，一旦我们取得了 jQuery 对象，就可直接调用 jQuery 的方法来操作 DOM 了。jQuery 提供了大量方法来简化 DOM 操作，从而允许开发者以更一致、更精练的代码来动态改变 HTML 页面。

➤➤11.4.1 jQuery 命名空间的方法

jQuery 还提供了一个 jQuery 命名空间，开发者可以直接使用 jQuery 命名空间下的属性和方法。

提 示：

熟悉 Java 语法的读者可以把下面的工具方法当成 Java 的静态方法，开发者可以通过类名（jQuery 类）来调用这些工具方法。例如 jQuery.browser 返回系统浏览器相关属性。

- jQuery.browser：返回当前浏览器的相关信息，返回值是一个 JSON 格式的对象。
- jQuery.browser.version：返回用户浏览器的版本号。
- jQuery.boxModel：如果用户浏览器当前浏览的页面使用的是 W3C CSS Box Model 则返回 true，否则返回 false。
- jQuery.isFunction(obj)：测试 obj 是否为一个函数，如果是则返回 true。

除了上面这些基本的工具方法之外，jQuery 命名空间下还提供了如下工具方法，这些工具方法并不是用于操作 DOM，而是用于操作普通的字符串、数组和对象的，但这些方法对简化开发者的 JavaScript 编程一样大有裨益。

11.4.1.1 字符串、数组和对象相关工具方法

如下方法主要用于操作字符串、数组和对象：

- jQuery.trim(str)：截断字符串前后的空白。
- jQuery.each(object, callback)：该方法用于遍历 JavaScript 对象和数组（不是遍历 jQuery 对象）。其中 object 就是要遍历的对象或数组，callback 是一个形如 function(index, val){}的函数，其中 index 是对象的属性名或数组的索引，val 为对应的属性值或数组元素。如果想中途退出 each()遍历，则让 callback 函数返回 false 即可，这样其他返回值将被忽略。
- jQuery.extend(target, object1, [objectN])：将 object1、objectN 的属性合并到 target 对象里。如果 target 里有和 object1、objectN 同名的属性，则 object1、objectN 的属性值将覆盖 target 的属性值；如果 target 不包含 object1、objectN 里所包含的属性值，则 object1、objectN 的属性值将会新增到 target 对象里。
- jQuery.grep(array, callback, [invert])：该方法用于对 array 数组进行筛选。callback 是一个形如 function(val, index){}的函数，其中 index 是对象的属性名或数组的索引，val 为对应的属性值或数组元素。grep 将会依次将 array 数组元素的索引和值传入 callback 函数，如果 callback 函数返回 true 则保留该数组元素，否则删除数组元素。如果将 invert 指定为 true，则当 callback 函数返回 true 时反而会删掉该数组元素。
- jQuery.makeArray(obj)：将类数组对象（例如 HTMLCollection 对象）转换为真正的数组对象。类数组对象有 length 属性，其元素索引为 0 到 length-1。

- jQuery.map(array, callback)：该函数用于将 array 数组转换为另一个数组。callback 是一个形如 function(val, index){}的函数，其中 index 是数组的索引，val 为数组元素。map 将会依次将 array 数组元素的索引和值传入 callback 函数，每次传入 callback 函数的返回值将作为新数组的元素——这样就产生了一个新数组。
- jQuery.inArray(value, array)：返回 value 在 array 中出现的位置，如果 array 中不包含 value 元素，则返回-1。
- jQuery.merge(first, second)：合并 first、second 两个数组，将两个数组的元素合并到新数组里，并不会删除重复值。
- jQuery.unique(array)：删除 array 数组中的重复值。该 array 通常是 DOM 对象数组。

注意：

jQuery.unique(array)只对 DOM 对象数组起作用。如果是普通的字符串数组或数值型数组，则 juery.unique(array)函数不会有任何作用。

下面的代码示范了上述工具方法的用法：

程序清单：codes\11\11.4\tools.html

```
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//访问浏览器属性
for (var key in $.browser)
{
    document.writeln("当前浏览器的" + key
        + "属性为：" + $.browser[key] + "<br />");
}
//去除字符串前后的空白
document.writeln("$.trim('ddd');的结果是" + $.trim("ddd") + "<br />");
//遍历数组
$.each(["java" , "ajax" ,"java ee"] , function(index, val)
{
    document.writeln("['java' , 'ajax' ,'java ee']的第" + index
        + "个元素为:" + val + "<br />");
});
//以指定函数过滤数组
var grepResult = $.grep(["java" , "ajax" ,"java ee"] , function(val, index)
{
    //当数组元素的字符个数大于 5 时被保留
    return val.length > 5;
});
document.writeln("['java' , 'ajax' ,'java ee']里数组元素的字符个数大于 5 的还有："
    + grepResult + "<br />");
//以旧数组创建新数组
var mapResult = $.map(["java" , "ajax" ,"java ee"] , function(val, index)
{
    //将数组元素和索引值连缀在一起作为新的数组元素
    return val + index
});
document.writeln("以['java' , 'ajax' ,'java ee']创建的新数组为："
    + mapResult);
//创建 div 元素
var div = $("<div>aa<div>");
//以相同的两个 div 创建数组
var divArr = [div , div];
document.writeln("divArr.length 的值为：" + divArr.length + "<br />");
//执行$.unique 去除重复元素
```

```
document.writeln("$.unique(divArr).length 的结果为："
    + $.unique(divArr).length + "<br />");
var str = "aa";
//以两个相同的字符串创建数组
var strArr = [str , str];
document.writeln("strArr.length 的值为：" + strArr.length + "<br />");
//执行$.unique 不能去除重复元素
document.writeln("$.unique(strArr).length 的结果为："
    + $.unique(strArr).length + "<br />");
</script>
```

运行上面的程序将可看到如图 11.10 所示结果。

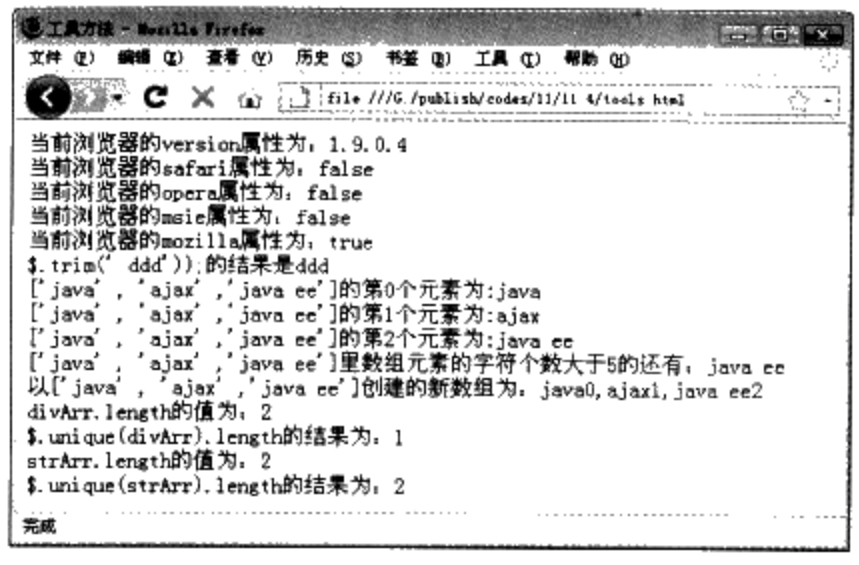


图 11.10　jQuery 命名空间下的工具方法

➤➤11.4.2　数据存储的相关方法

jQuery 允许把 jQuery 对象当做一个临时的“数据存储中心”，开发者能以 key-value 对的形式将数据存储到 jQuery 对象里，也可从 jQuery 对象里取出之前存储的数据，还可以删除之前存储的数据。存入 jQuery 对象里的数据既可以是基本类型值，也可以是数组、JavaScript 对象等。

- data(name)：获取 jQuery 对象里存储的 key 为 name 的数据。
- data(name, value)：向 jQuery 对象里存储 name:value 的数据对。
- removeData(name)：删除 jQuery 对象里存储的 key 为 name 的数据。

下面的代码示范了 jQuery 的数据存储相关方法：

程序清单：codes\11\11.4\data.html

```
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    var target = $("<div>java</div>");
    //向 jQuery 对象里添加 book 数据
    target.data("book" , "疯狂 Java 讲义");
    //访问 jQuery 对象里的 book 数据，将输出"疯狂 Java 讲义"
    alert(target.data("book"));
    //删除 jQuery 对象里的 book 数据
    target.removeData("book");
    //再次访问 jQuery 对象里的 book 数据，将输出"undefined"
    alert(target.data("book"));
</script>
```

➤➤11.4.3　操作属性的相关方法

下面这组方法是操作 DOM 对象属性的通用方法，可以操作 DOM 对象的通用属性，例如 title、alt、

src 等。

- attr(name)：访问 jQuery 对象里第一个匹配元素的 name 属性值。如果 jQuery 对象里包含的 DOM 对象都没有 name 属性，则该方法返回 undefined。name 可以是 title、alt、src、href 等属性。
- attr(properties)：用于为 jQuery 对象里的所有 DOM 对象同时设置多个属性值。其中 properties 是一个形如{name1:value1,name2:value2... }的 JavaScript 对象，例如{"src","logo.jpg"}。
- attr(key, value)：用于为 jQuery 对象里的所有 DOM 对象设置单个属性值。其中 key 是需要设置的属性名，value 是需要设置的属性值。
- attr(key, fn)：用于为 jQuery 对象里的所有 DOM 对象设置单个属性值，但不是直接给定属性值，而是提供 fn 函数，由 fn 函数来计算各元素的属性值。fn 是一个形如 function(index){}的函数，其中 index 代表各 DOM 元素在 jQuery 对象中的索引。
- removeAttr(name)：删除 jQuery 对象里所有 DOM 对象里的 name 属性的值。

下面的程序示范了动态改变页面中<img.../>元素 src 属性值的情况：

程序清单：codes\11\11.4\attribute.html

```
<body>
<img/><img/>
<div>
    <img/><img/><img/>
</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    //获取 body 下的 img 元素，并为这些 img 元素设置 src 属性值
    $("body>img").attr("src" , "logo.jpg")
        .attr("alt" , "疯狂 Java 联盟");
    //获取 div 下的 img 元素，并为这些 img 元素设置 src 属性值
    $("div>img").attr("src" , function(index)
    {
        return index + 1 + ".gif";
    });
</script>
</body>
```

上面的程序中两次使用了 jQuery 的 attr()方法，前一个方法为<img.../>元素的 src 属性设置固定值，后一个方法使用函数为<img.../>元素设置 src 属性值。在浏览器中浏览该页面，可看到如图 11.11 所示效果。

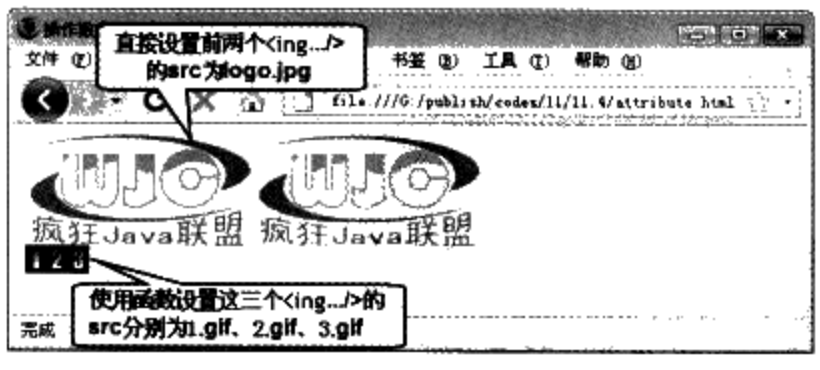


图 11.11　使用 attr()方法修改<img.../>元素的 src 属性

➤➤11.4.4　操作 CSS 属性的相关方法

jQuery 提供了如下操作 DOM 元素 CSS 样式的方法，包括直接访问、修改 DOM 元素的 class 属性值；除此之外，还提供了访问、修改 DOM 元素内联 CSS 属性值的方法。不仅如此，还提供了大量直接访问、修改 DOM 元素大小、位置的方法。

jQuery 提供的操作 CSS 属性的相关方法如下：

- addClass(class)：将指定的 CSS 定义添加到 jQuery 对象包含的所有 DOM 对象上。
- hasClass(class)：判断该 jQuery 对象是否包含至少一个具有指定的 CSS 定义的 DOM 对象。只要该 jQuery 对象里有一个 DOM 对象具有该 CSS 定义，则该方法返回 true，否则返回 false。
- removeClass(class)：删除 jQuery 对象所包含的所有 DOM 对象上的指定 CSS 定义。
- toggleClass(class)：如果 jQuery 对象包含的所有 DOM 对象上具有指定的 CSS 定义，则删除该 CSS 定义；否则添加该 CSS 定义。
- css(name)：返回该 jQuery 对象包含的第一个匹配的 DOM 对象上名为 name 的 CSS 属性值（也就是返回该 DOM 对象的 style.name 属性值）。如果在 jQuery 对象里找到的第一个 DOM 对象具有 style.name 属性值，则返回该值，否则返回 undefined。
- css(properties)：为 jQuery 对象包含的所有 DOM 对象同时设置多个 CSS 属性值（设置它们的内联 CSS 属性）。properties 是一个形如{key1:val1,key2:val2...}的对象，如{border:"1px solid black"}。
- css(name, value)：为 jQuery 对象包含的所有 DOM 对象设置单个 CSS 属性值（设置它们的内联 CSS 属性）。如 target.css("border" , "1px solid black");。
- offset()：获取 jQuery 对象包含的第一个匹配的 DOM 对象相对于该文档的位置。该方法返回一个形如{left:n,top:m}的对象。
- position()：获取 jQuery 对象包含的第一个匹配的 DOM 对象相对于其父元素的位置。该方法返回一个形如{left:n,top:m}的对象。
- scrollTop()：获取 jQuery 对象包含的第一个匹配的 DOM 对象的 scroll top 值（该属性值会考虑垂直滚动条里滑块的位置）。
- scrollTop(val)：设置 jQuery 对象里包含的所有 DOM 对象的 scroll top 值。
- scrollLeft()：获取 jQuery 对象包含的第一个匹配的 DOM 对象的 scroll left 值（该属性值会考虑水平滚动条里滑块的位置）。
- scrollLeft(val)：设置 jQuery 对象里包含的所有 DOM 对象的 scroll left 值。
- height()：返回 jQuery 对象里第一个匹配的元素的当前高度（以 px 为单位）。
- height(val)：设置 jQuery 对象里所有元素的高度，val 的单位为 px。
- width()：返回 jQuery 对象里第一个匹配的元素的当前宽度（以 px 为单位）。
- width(val) ：设置 jQuery 对象里所有元素的宽度，val 的单位为 px。
- innerHeight()：返回 jQuery 对象里第一个匹配的元素的内部高度（以 px 为单位），内部高度就是该元素的高度减去边框宽度，再减去垂直 padding 的大小。
- innerWidth()：返回 jQuery 对象里第一个匹配的元素的内部宽度（以 px 为单位），内部宽度就是该元素的宽度减去边框宽度，再减去水平 padding 的大小。
- outerHeight([options])：获取 jQuery 对象里第一个匹配元素的外部高度（包括边框和默认的 padding）。该方法对隐藏元素同样有效。如果 options 设为 true，则元素的页边距也会算在外部高度之内。
- outerWidth([options])：获取 jQuery 对象里第一个匹配元素的外部宽度（包括边框和默认的 padding）。该方法对隐藏元素同样有效。如果 options 设为 true，则元素的页边距也会算在外部宽度之内。

这些操作 DOM 元素 CSS 属性的方法比较简单清晰，而且前面的程序中也使用了部分操作 CSS 属性的方法，故此处的示例程序仅示范这里的部分方法：

程序清单：codes\11\11.4\css.html

```
<div id="test1">
整体添加 CSS 样式的元素
</div><br/>
<div id="test2">
```

```
采用 css(properties)方法添加 CSS 样式的元素
</div><br/>
<div id="test3" style="position:absolute;">
可以自由移动的元素
</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    //为 id 为 test1 的元素设置 class="text"
    $("#test1").addClass("test");
    //为 id 为 test2 的元素设置内联 CSS 样式
    $("#test2").css({border:"1px solid black" , color:"#888888"});
    //获取 id 为 test3 的元素
    var target = $("#test3")
        //设置背景色
        .css("background-color" , "#cccccc")
        .css("padding" , 10)
        //设置宽度
        .width(200)
        //设置高度
        .height(80)
        //设置位置
        .css("left" , 40)
        .css("top" , 64);
    //获取 target 的位置
    var posi = target.position();
    alert("target 的 X 坐标为:" + posi.left + "\n"
        + "target 的 Y 坐标为:" + posi.top);
</script>
```

在浏览器中浏览该页面，可看到如图 11.12 所示效果。

图 11.12　使用 jQuery 的 CSS 相关方法

➤➤11.4.5　操作元素内容的相关方法

jQuery 还提供了如下方法来访问或设置 DOM 元素的内容，包括访问或设置这些 DOM 元素的 innerHTML 属性、文本内容和 value 属性：

- html()：返回 jQuery 对象包含的第一个匹配的 DOM 元素的 HTML 内容（也就是返回其 innerHTML 属性值）。该方法不能在 XML 文档中使用，可以在 XHTML 文档中使用。
- html(val)：设置 jQuery 对象包含的所有 DOM 元素的 HTML 内容（也就是同时设置它们的 innerHTML 属性值）。该方法不能在 XML 文档中使用，可以在 XHTML 文档中使用。
- text()：返回 jQuery 对象包含的所有 DOM 元素的文本内容（会剔除该 DOM 元素里所有的 XML、HTML 标签）。该方法对 XML 文档和 XHTML 文档都有作用。
- text(val)：设置 jQuery 对象包含的所有 DOM 元素的文本内容。该方法对 XML 文档和 XHTML 文档都有作用。

- val()：返回 jQuery 对象包含的第一个匹配的 DOM 元素的 value 值，实际上就是返回表单控件的 value 属性值。该方法可返回字符串和数组（例如多选框和允许多选的下拉列表框）。
- val(val)：为 jQuery 对象包含的所有 DOM 元素设置单个 value 属性值。实际上就是设置表单控件的 value 属性值。
- val(Array<String>)：为 jQuery 对象包含的所有 DOM 元素设置多个 value 属性值。主要用于操作复选框和允许多选的下拉列表框。

下面的程序示范了操作元素内容的相关方法的用法：

程序清单：codes\11\11.4\content.html

```
<div></div><div></div>
<input id="book" name="book" type="text" /><br />
<input id="desc" name="desc" type="text" /><br />
<select id="gender">
    <option value="male">男人</option>
    <option value="female">女人</option>
</select><br />
<select id="publish" multiple="multiple">
    <option value="phei">电子工业出版社</option>
    <option value="tsinghua">清华大学出版社</option>
</select><br />
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//设置 body 下的 div 元素的内容
$("body>div").html("疯狂 Ajax 讲义");
//设置所有 input、select 和 textarea 的值
$(":input").val("疯狂 XML 讲义");
//为所有的<select.../>元素设置 value 值
$("select").val(["female" , "tsinghua" , "phei"]);
//仅获取 jQuery 元素的 text 部分，下面将输出“java:疯狂 Java 讲义”
alert($("<div>java:<span>疯狂 Java 讲义</span></div>").text());
</script>
```

在浏览器中浏览该页面，可看到如图 11.13 所示效果。

图 11.13　使用 jQuery 的操作元素内容的方法

➤➤11.4.6　操作 DOM 节点的相关方法

DOM 操作里最常见的操作就是对节点的操作，包括创建、复制节点、插入节点和删除节点等，而 jQuery 也提供了大量相关方法来简化对 DOM 节点的操作。

11.4.6.1 在指定节点内插入新节点

如下方法都用于在指定节点内添加新内容：

- append(content)：在 jQuery 对象包含的所有 DOM 节点内的尾部插入 content 所代表的内容，其中 content 既可以是 HTML 字符串，也可以是 DOM 元素，还可以是 jQuery 对象。

- appendTo(selector)：将当前 jQuery 对象包含的 DOM 元素添加到 selector 匹配的所有 DOM 的内部的尾端。

注意：

append()方法是在当前 jQuery 对象内部插入其他元素；而 appendTo()方法是将当前 jQuery 对象插入到其他元素内部。

- prepend(content)：在 jQuery 对象包含的所有 DOM 节点内的顶部插入 content 所代表的内容，其中 content 既可以是 HTML 字符串，也可以是 DOM 元素，还可以是 jQuery 对象。
- prependTo(selector)：将当前 jQuery 对象包含的 DOM 元素添加到 selector 匹配的所有 DOM 的内部的顶端。

下面的程序示范了这些方法的功能：

程序清单：codes\11\11.4\append.html

```
<div id="test1"></div>
<div id="test2" style="border:1px solid black;">id为test2的元素</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    //直接将一段HTML字符串添加到id为test1的元素的内部的尾端
    $("#test1").append("<b>疯狂XML讲义</b>");
    //创建一个<span.../>元素
    var span = document.createElement("span");
    span.innerHTML = "疯狂Java讲义"
    //将一个DOM元素添加到id为test1的元素的内部的顶端
    $("#test1").prepend(span);
    //将id为test1的元素添加到id为test2的元素内部的尾端
    $("#test1").appendTo("#test2");
</script>
```

在浏览器里浏览该页面，可看到如图 11.14 所示效果。

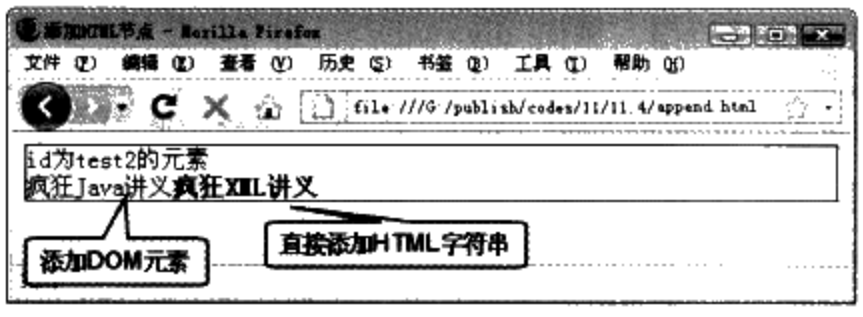


图 11.14　在节点内部插入内容

11.4.6.2　在指定节点外添加节点

如下方法用于在目标节点的前后添加新节点：

- after(content)：在该 jQuery 对象包含的所有 DOM 节点之后添加 content 对应的内容。其中 content 既可以是 HTML 字符串，也可以是 DOM 对象，还可以是 jQuery 对象。
- before(content)：在该 jQuery 对象包含的所有 DOM 节点之前添加 content 对应的内容。其中 content 既可以是 HTML 字符串，也可以是 DOM 对象，还可以是 jQuery 对象。
- insertAfter(selector)：将当前 jQuery 对象包含的所有 DOM 节点插入到 selector 匹配的所有节点之后。
- insertBefore(selector)：将当前 jQuery 对象包含的所有 DOM 节点插入到 selector 匹配的所有节点之前。

如下程序示范了以上几个插入方法的用法：

程序清单：codes\11\11.4\insert.html

```
<div id="test1" style="border:1px dotted black;">id 为 test1 的元素</div><br />
<div id="test2" style="border:1px solid black;">id 为 test2 的元素</div>
<hr />
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    //直接将一段 HTML 字符串插入到 id 为 test1 的元素的前面
    $("#test1").before("<b>疯狂 Ajax 讲义</b>");
    //直接将一段 HTML 字符串插入到 id 为 test1 的元素的后面
    $("#test1").after("<b>疯狂 XML 讲义</b>");
    //将 id 为 test2 的元素插入到 hr 元素之后
    $("#test2").insertAfter("hr");
</script>
```

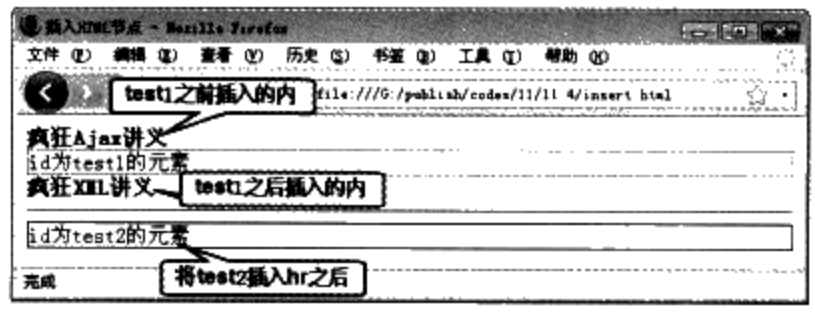


图 11.15　示范 jQuery 的插入方法

11.4.6.3　包裹

下面的方法还可以将当前 jQuery 对象里包含的 DOM 节点包裹起来，也就是在这些 DOM 节点之前插入开始标签，在其之后插入结束标签。

- wrap(node)：包裹当前 jQuery 对象包含的每个 DOM 节点。其中 node 既可以是有开始标签和结束标签的 HTML 字符串，例如 ；也可以是这些标签所对应的 DOM 元素。
- wrapAll(node)：包裹当前 jQuery 对象包含的所有 DOM 节点。其中 node 既可以是有开始标签和结束标签的 HTML 字符串，例如 ；也可以是这些标签所对应的 DOM 元素。

提 示：

wrap()和 wrapAll()的区别在于：wrap()是 jQuery 对象里的每个 DOM 元素都进行包裹，jQuery 对象里有几个元素就包裹几次；而 wrapAll()是将 jQuery 对象里的所有 DOM 元素当成一个整体进行包裹，不管 jQuery 对象里有多少个 DOM 元素都只包裹一次。

- wrapInner(node)：包裹当前 jQuery 对象包含的每个节点的内部成分——该方法不再包裹节点的全部，而是仅包裹节点内部的部分。

程序清单：codes\11\11.4\wrap.html

```
<span id="test1">id 为 test1 的元素</span><br />
<span id="test2">id 为 test2 的元素</span>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//在每个 span 元素之外再包裹一个带点线边框的 div 元素
$("span").wrap("<div style='border:1px dotted black'></div>");
//将每个 span 元素的内部成分再包裹一个灰色背景的 span 元素
$("span").wrapInner("<span style='background-color:#dddddd'></span>");
</script>
```

在浏览器中执行上述页面可得到如下页面代码：

```
<div style="border: 1px dotted black;"><span id="test1"><span style="background-color:
#dddddd;">
id为test1的元素</span></span></div><br>
<div style="border: 1px dotted black;"><span id="test2"><span style="background-color:
#dddddd;">
id为test2的元素</span></span></div>
```

11.4.6.4 替换

下面的方法用于替换 DOM 节点：

replaceWith(content)：将当前 jQuery 对象包含的所有 DOM 对象替换成 content。其中 content 既可以是 HTML 字符串，也可以是 DOM 对象，还可以是 jQuery 对象。

replaceAll(selector)：将当前 jQuery 对象包含的所有 DOM 对象替换成 selector 匹配的元素。

11.4.6.5 删除

下面的方法用于删除指定 DOM 节点：

empty()：删除当前 jQuery 对象包含的所有 DOM 节点里的内容（仅保留每个 DOM 节点对应的开始标签和结束标签）。

remove([expr])：删除当前 jQuery 对象包含的所有 DOM 节点。如果指定了 expr 选择器，则只删除 expr 选择器匹配的 DOM 节点。

11.4.6.6 复制

下面的方法用于复制 DOM 节点：

- clone([true])：复制当前 jQuery 对象里包含的所有 DOM 元素并且选中这些复制出来的副本。当程序需要把 DOM 文档中元素的副本添加到其他位置时，这个函数非常有用。其中 true 是可选的，如果指定了 true 则还会复制该 DOM 元素上的事件处理。

程序清单：codes\11\11.4\remove.html

```
<div><span id="test1">id为test1的元素</span>Java</div>
<span id="test2">id为test2的元素</span>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//将div元素内容全部清空
$("div").empty()
    //重新添加字符串
    .append("重新添加");
//删除所有id为test2的span元素
$("span").remove("#test2");
//取得页面中的div元素，并复制该元素
$("div").clone()
    //添加背景色
    .css("background-color" , "#cdcdcd")
    //添加到body元素尾部
    .appendTo("body");
</script>
```

在浏览器中浏览该页面，可看到如图 11.16 所示结果。

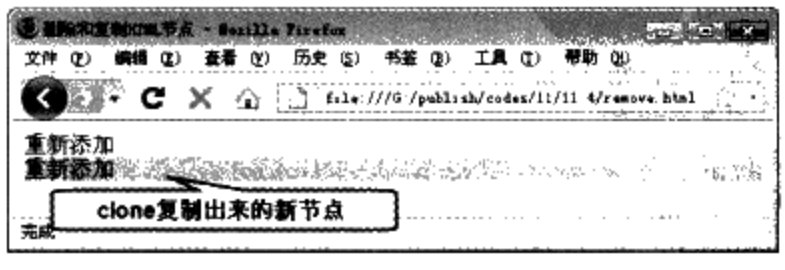


图 11.16 删除、复制节点

11.5 jQuery 事件相关方法

jQuery 也对 JavaScript 事件模型进行了简化，提供了一致的事件模型，从而允许开发者以更简洁的方式进行事件编程。jQuery 为事件编程提供了如下方法：

- ready(fn)：指定当该 jQuery 所对应的 DOM 对象装载完成时执行 fn()函数。该函数中可以指定一个参数，该参数即可代替原有的 jQuery 函数，这样即可避免命名冲突。

注意：

上面这个函数一个常用的场景是：$(document).ready(function(){...});，这行代码指定在页面装载完成时立即执行 ready()方法里指定的函数。值得指出的是，一定要保证在<body.../>元素中没有指定 onload 属性，否则不会触发$(document).ready()里指定的函数。同一个页面中可以无限次地调用$(document).ready()方法，多次注册的函数会按照（代码中的）先后顺序依次执行。

- bind(type, [data], fn)：为当前 jQuery 对象包含的所有 DOM 元素的 type 事件绑定事件处理函数 fn， fn 是一个形如 function(event){}的函数，其中 event 代表触发该函数的事件。data 是个可选参数，它是个形如{key1:val1,key2:val2...}的对象，函数 fn 可通过 event.data 来访问 data 对象，通过指定 data，可以向事件处理函数传入更多数据。

注意：

对于上面的函数 fn 而言，如果该函数想阻止该事件的默认行为，并阻止事件冒泡，则让该函数 return false 即可；如果只想取消默认行为，则调用 event 的 preventDefault()即可；如果只想阻止冒泡，则调用 evnet 的 stopPropagation()即可。

- one(type, data, fn)：该方法与 bind()方法的作用基本一致。与 bind 的区别是：无论如何，这个事件处理函数只会被执行一次。
- trigger(type, [data])：以编程方式触发当前 jQuery 对象包含的所有 DOM 对象上的 type 事件，该方法可以触发由 bind()绑定的自定义事件。除此之外，该函数也会导致 DOM 元素执行同名的事件动作。例如我们使用 trigger()触发一个表单的 submit 事件，则该表单将会被提交（如果要阻止这种默认行为，事件处理函数可返回 false）。data 是一个可选的数组类型的参数，该参数可以传给绑定在 DOM 对象上的事件处理函数。
- triggerHandler(type, [data])：该方法与 trigger 的作用基本相似，只是调用该方法来触发 type 事件时，不会导致 DOM 元素执行同名的事件动作。
- unbind([type], [fn])：这是 bind()方法的反方向操作，它用于从当前 jQuery 对象包含的每个 DOM 元素中删除绑定的事件处理函数。该函数的 type、fn 两个参数都是可选的，如果没有指定任何参数，则删除每个 DOM 元素上的所有事件处理函数。如果指定了 type，则只删除为 type 事件绑定的事件处理函数；如果指定了 fn，则只删除 DOM 上绑定的事件处理函数 fn。
- hover(over, out)：该方法为当前 jQuery 对象包含的每个 DOM 元素的 onmouseover、onmouseout 事件绑定事件处理函数。其中 over、out 都是函数，分别绑定到 onmouseover、onmouseout 事件作为事件处理函数。
- toggle(fn, fn2, [fn3,fn4,...])：为当前 jQuery 对象包含的每个 DOM 元素的 click 事件绑定多个事件处理函数。当用户第一次单击 DOM 元素时，系统触发函数 fn1，第二次单击时，系统触发函数 fn2……依此类推，直到所有函数都被依次触发一遍，此后如果用户再次单击，则系统又将触发函数 fn1——也就是从头再来。

下面的程序示范了上面几个方法的用法：

程序清单：codes\11\11.5\event.html

```
<input id="test1" type="button" value="单击我"/><br />
<input id="test2" type="button" value="切换右边复选框的勾选状态"/>
<input id="check" type="checkbox" value=""/><br />
<input id="test3" type="button" value="绑定 toggle 的按钮"/><br />
<div id="test4">
鼠标悬停、移除将触发指定函数
</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//指定页面加载完成时执行指定函数
$(document).ready(function()
{
    alert("页面加载完成！");
});
//为 id 为 test1 的元素的 click 事件绑定事件处理函数
$("#test1").bind("click" , {book:"疯狂 Ajax 讲义"} , function(event)
{
    alert("id 为 test1 的按钮被单击!\n" +
        "事件为: " + event +
        "\n 事件上 data 的 book 属性为: " + event.data.book);
});
//为 id 为 test2 的元素的 click 事件绑定事件处理函数
$("#test2").bind("click" , function(event)
{
    //使用代码触发 id 为 check 的元素的单击事件，而且执行默认行为
    $("#check").trigger("click");
});
//使用 toggle 为 id 为 test3 的元素绑定 3 个事件处理函数
$("#test3").toggle(
    function(event)
    {
        alert("3n 次被单击" + event);
    },
    function(event)
    {
        alert("3n + 1 次被单击");
    },
    function(event)
    {
        alert("3n + 2 次被单击");
    }
);
//使用 hover 为 id 为 test4 的元素绑定 2 个事件处理函数
//当鼠标移入该元素时触发第一个函数，移出该元素时触发第二个函数
$("#test4").css("border" , "1px solid black")
    .css("background-color" , "#cccccc")
    .width(200)
    .height(80)
    .hover(function(event)
    {
        alert("鼠标移入该元素之内！");
    },
    function()
    {
        alert("鼠标移出该元素！");
    }
);
</script>
```

上面的程序中分别示范了 bind、trigger、toogle 和 hover 等函数的用法，读者可以在浏览器中浏览该页面，并通过与页面中的按钮等元素交互来进一步掌握 jQuery 对事件编程的简化。

具体事件相关的方法

除了上面提到的通用事件编程相关函数之外，jQuery 还提供了与表 7.1 中各种事件属性相对应的许多方法，例如 blur、click 等——只要将表 7.1 中各事件属性的 on 前缀去掉，这些都可作为 jQuery 对象的方法。

程序清单：codes\11\11.5\concreteEvent.html

```
<input id="test1" type="button" value="单击我"/><br />
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
$("#test1").click(function(event)
{
    alert("id为test1的按钮被单击" + event)
})
.click();
```

上面的程序中对$("#test1")对象执行了两次 click 方法，第一次调用 click 方法时传入了一个匿名函数，这表明将该匿名函数绑定为$("#test1")的 click 事件处理函数；第二次调用 click 方法时没有传入任何参数，这表明将使用代码触发$("#test1")的 click 事件。

通过上面的程序不难看出，jQuery 提供的 click、blur、change、dblclick 等方法的用法非常简单，如果调用这些方法时指定了函数作为参数，则该函数将作为 jQuery 对象里 DOM 对象的事件处理函数；如果没有指定函数，就是通过代码触发 jQuery 对象里 DOM 对象的事件。

11.6 动画效果相关的方法

jQuery 还提供了一些效果相关方法，通过使用这些方法，我们可以非常方便地完成用户交互操作，从而简化 Ajax 系统开发。这些方法如下：

- show()：将该 jQuery 对象里包含的隐藏的 DOM 元素显示出来。
- show(speed, [callback])：将该 jQuery 对象里包含的隐藏的 DOM 元素以动画方式显示出来，其中 speed 既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表示毫秒数的整数，用于指定该动画的持续时间。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成，隐藏的 DOM 元素完全显示出来后将触发该函数。
- hide()：将该 jQuery 对象里包含的显示的 DOM 元素隐藏起来。
- hide(speed, [callback])：将该 jQuery 对象里包含的显示的 DOM 元素以动画方式隐藏起来，其中 speed 既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表示毫秒数的整数，用于指定该动画的持续时间。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成，显示的 DOM 元素完全隐藏起来后将触发该函数。
- toggle()：该方法是 show()、hide()两个方法的综合版本。如果当前 jQuery 对象里的 DOM 元素处于隐藏状态，就将它们显示出来；如果它们处于显示状态，就将它们隐藏起来。
- toggle(speed, callback)：该方法是 show(speed, [callback])、hide(speed, [callback])两个方法的综合版本。
- slideDown(speed, [callback])：该方法用于实现第 5 章介绍的卷帘效果。该方法将会不断增加当前 jQuery 对象所匹配 DOM 元素的高度，直至这些 DOM 元素完全显示出来。speed 指定该动画的持续时间，既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表

示毫秒数的整数。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成，所匹配的 DOM 元素完全显示出来后将触发该函数。

- slideUp(speed, [callback])：该方法用于实现第 5 章介绍的卷帘效果。该方法将会不断减小当前 jQuery 对象所匹配 DOM 元素的高度，直至这些 DOM 元素完全隐藏起来。speed 指定该动画的持续时间，既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表示毫秒数的整数。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成，所匹配的 DOM 元素完全隐藏起来后将触发该函数。
- slideToggle(speed, [callback])：该方法是 slideDown(speed, [callback])、slideUp(speed, [callback])的综合版本。如果当前 jQuery 匹配的元素处于隐藏状态，就使用“卷帘”动画将其显示出来；如果它们处于显示状态，就使用“卷帘”动画将它们隐藏起来。
- fadeIn(speed, [callback])：将 jQuery 对象匹配的 DOM 元素以“渐显”的方式显示出来（也就是不断调整透明度）。speed 指定该动画的持续时间，既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表示毫秒数的整数。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成，所匹配的 DOM 元素完全显示出来后将触发该函数。
- fadeOut(speed, [callback])：将 jQuery 对象匹配的 DOM 元素以“渐隐”的方式隐藏起来（也就是不断调整透明度）。speed 指定该动画的持续时间，既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表示毫秒数的整数。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成，所匹配的 DOM 元素完全隐藏起来后将触发该函数。
- fadeTo(speed, opacity, [callback])：将 jQuery 对象匹配的 DOM 元素的透明度调整到 opacity 值。opacity 是一个 0~1 的浮点数。speed 指定该动画的持续时间，既可以是"slow"、"normal"或"fast"三个字符串其中之一，也可以是一个表示毫秒数的整数。callback 是可选参数，用于指定一个回调函数，指定当动画执行完成后将触发该函数。

上面这些方法都代表了一些简单常用的效果，下面的程序示范了这些方法的用法：

程序清单：codes\11\11.6\effect.html

```
<input type="button" value="toggle" onclick="$('#test1').toggle(1000);"/><br />
<div id="test1">使用 toggle 控制的元素</div>
<input type="button" value="slide down" onclick="$('#test2').slideDown(1000);"/>
<input type="button" value="slide up" onclick="$('#test2').slideUp(1000);"/><br />
<div id="test2">使用 Slide 动画控制的元素</div>
<input type="button" value="fade in" onclick="$('#test3').fadeIn(1000);"/>
<input type="button" value="fade out" onclick="$('#test3').fadeOut(1000);"/><br />
<div id="test3">使用 Fade 动画控制的元素</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
```

在浏览器中运行上述页面，即可看到 toggle、slideUp、slideDown、fadeIn 和 fadeOut 等几种动画效果。通过使用上面的函数，开发者可以非常简便地开发出用户交互动画。

除了上面几个简单的动画效果方法之外，还有以下几个更复杂的方法，通过它们可以进行更复杂的控制：

- animate(params[,duration[,easing]][,callback])：该函数用于创建自定义动画。其中 params 是一个形如{prop1:endVal1,prop:endVal2...}的 JavaScript 对象，用于指定当前 jQuery 对象包含的 DOM 对象经过动画效果后的状态。duration 用于指定动画持续时间。easing 用于指定动画所使用擦除效果的名称（需要插件支持），jQuery 默认只支持"linear" 和 "swing"两个值。callback 指定动画结束后激发的回调函数。
- animate(params, options)：该函数是前一个函数的另一种形式。其中 params 与前一个函数完全相同。options 是一个用于指定复杂选项的 JavaScript 对象，可指定如下选项：

- duration：指定该动画的持续时间，可以是"slow"、 "normal"或"fast"三个字符串其中之一，也可以是表示动画持续时长的毫秒数，如 1000。
- easing：指定该动画所使用擦除效果的名称（需要插件支持），jQuery 默认只支持"linear" 和"swing"两个值。
- complete：指定在动画完成时激发所指定的函数。
- step：动画效果每改变一次将导致所指定的函数执行一次。
- queue：指定是否将该动画函数放入该对象的动画函数队列之后。

➢ stop([clearQueue], [gotoEnd])：停止当前 jQuery 对象里每个 DOM 元素上正在执行的动画。如果该 jQuery 对象上绑定了动画队列，且 clearQueue 没有指定为 true，则执行该方法后将立即执行当前动画的下一个动画。该函数可以指定 clearQueue、gotoEnd 两个可选的布尔类型的参数，其中 clearQueue 指定是否删除该 jQuery 对象上的动画队列；如果将 gotoEnd 设置为 true，则当前动画立即跳到最后一帧而结束，否则当前动画将停在当前帧而结束。

➢ queue()：返回当前 jQuery 对象里第一个匹配的 DOM 元素上的动画函数队列。

➢ queue(callback)：将 callback 动画函数添加到当前 jQuery 对象里所有 DOM 元素的动画函数队列的尾部。

➢ queue(queue)：用 queue 动画函数队列代替当前 jQuery 对象里所有 DOM 元素的动画函数队列。

➢ dequeue()：从当前 jQuery 对象里所有 DOM 元素的动画函数队列中删除一个动画函数。

下面的程序示范了使用 animate()方法来执行自定义动画：

程序清单：codes\11\11.6\complex.html

```
<input id="bn1" type="button" value="执行动画"/><br />
<div id="test1">使用 toggle 控制的元素</div>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    //为 id 为 bn1 的按钮绑定事件处理函数
    $("#bn1").click(function()
        {
            //为 id 为 test1 的元素指定自定义动画
            $("#test1").animate(
                //下面的 JavaScript 对象指定动画结束时目标元素的状态
                {
                    fontSize:"24pt",
                    width:"300px",
                    opacity:05
                },
                //下面的对象指定动画的详细选项
                {
                    duration:800,
                    easing:"swing",
                    step:function()
                    {
                        alert('step 回调！');
                    }
                }
            )
        }
    );
</script>
```

上面的程序中粗体字代码使用 animate 函数创建了一个自定义动画。使用 animate 函数创建自定义动画时有两点需要注意：

➢ options 中每个属性名都应该采用“驼峰”写法，即 font-size 应该写成 fontSize。

- options 中每个属性都应该是可以渐变的样式属性，如“height”、“top”或“opacity”等。如果指定一个 fontWeight 属性，那就不行了。

11.7 Ajax 相关方法

jQuery 的另一个吸引人的功能就是它所提供的 Ajax 支持，jQuery 提供了大量工具方法，这些工具方法可以帮助开发者完成 Ajax 开发的大量通用操作，开发者只需指定发送 Ajax 请求的 URL、回调函数即可——甚至连回调函数都可以省略。

➤➤11.7.1 两个工具方法

类似于 Prototype 库提供了 Form.serialize()函数，用于将整个表单转换成查询字符串，jQuery 也提供了如下两个类似的方法，但 jQuery 的这两个方法功能更强大，它们不仅可用于处理表单，也可用于处理一个或多个表单控件，下面是关于这两个方法的简要说明：

- serialize()：将该 jQuery 对象包含的表单或表单控件转换成查询字符串。
- serializeArray()：将 jQuery 对象包含的表单或表单控件转换为一个数组，每个数组元素都是形如{name:fieldName,value:fieldVal}的对象，其中 fieldName 是对应表单控件的 name 属性，fieldVal 是对应表单控件的 value 属性。

下面的页面程序示范了这两个工具方法的用法：

程序清单：codes\11\11.7\serialize.html

```
<form id="test">
用户名：<input id="user" name="user" type="text" /><br />
个人介绍：<textarea id="desc" name="desc"></textarea><br />
喜欢的图书：<select id="book" name="book">
    <option value="java">疯狂 Java 讲义</option>
    <option value="javaee">轻量级 Java EE 企业应用实战</option>
    <option value="ajax">疯狂 Ajax 讲义</option>
</select>
</form>
<button id="bn1">查询字符串</button>
<button id="bn2">查询 JSON 对象</button><hr />
<span id="show"></span>
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
    //为 id 为 bn1 的按钮绑定事件处理函数
    $("#bn1").click(function()
    {
        //将 id 为 test 的表单转换为查询字符串
        $("#show").html($("#test").serialize());
    });
    //为 id 为 bn2 的按钮绑定事件处理函数
    $("#bn2").click(function()
    {
        //将所有输入元素转换为数组
        var arr = $(":input").serializeArray();
        $("#show").empty();
        //遍历数组 arr
        for (var index in arr)
        {
            $("#show").append("第" + index + "表单控件名为：" 
                + arr[index].name + ",值为：" + arr[index].value + "<br />");
        }
    });
</script>
```

上面的页面中两个按钮分别使用了 serialize 和 serializeAll 来处理表单，在浏览器中浏览该页面，并单击第二个按钮，可看到如图 11.17 所示结果。

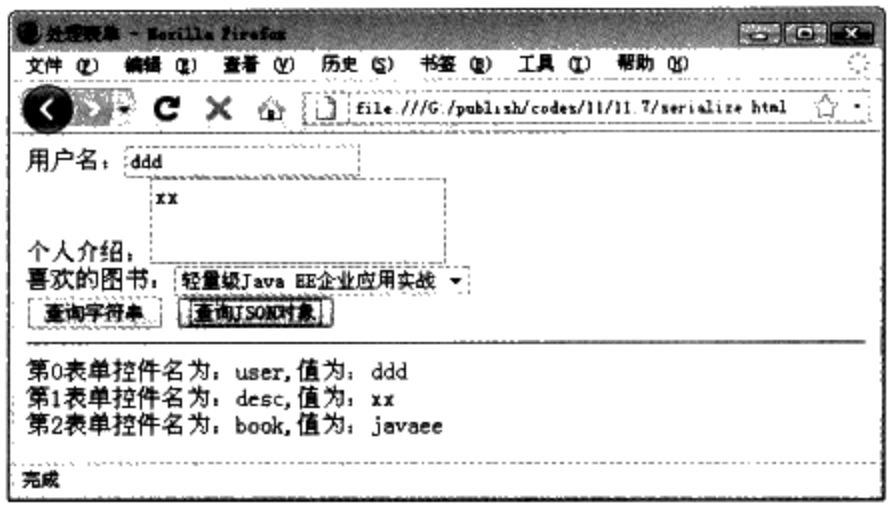


图 11.17　使用 serializeAll 处理表单控件

➤➤11.7.2　使用 load 方法

load 方法是一个非常便捷的 Ajax 交互方法，它向远程 URL 发送一个异步请求，甚至可以无须指定回调函数。load 方法的说明如下：

- load(url[,data][,callback])：向远程 url 发送异步请求，并直接将服务器响应插入当前 jQuery 对象匹配的 DOM 元素之内。其中 data 是一个形如{key1:val1,key2:val2...}的 JavaScript 对象，代表发送请求的请求参数。callback 指定交互 Ajax 成功后的回调函数。

下面的程序示范了如何使用 load 方法来进行 Ajax 交互：

程序清单：codes\11\11.7\load\index.html

```
<h3>请输入你的信息：</h3>
<form id="user">
    用户名:<input type="text" name="user" /><br />
    喜欢的图书:<select multiple="multiple" name="books">
        <option value="java">疯狂 Java 讲义</option>
        <option value="javaee">轻量级 Java EE 企业应用实战</option>
        <option value="ajax">疯狂 Ajax 讲义</option>
        <option value="xml">疯狂 XML 讲义</option>
    </select><br />
    <input id="load" type="button" value="Load"/>
</form><hr />
<div id="show"></div>
<script src="jquery-1.2.6.min.js" type="text/javascript">
</script>
<script type="text/javascript">
    //为 id 为 load 的按钮绑定事件处理函数
    $("#load").click(function()
    {
        //向 pro.jsp 发送 Ajax 请求，并自动加载服务器响应
        $("#show").load("pro.jsp" , $("#user").serializeArray());
    });
</script>
```

上面的程序中使用了 jQuery 的 serializeArray()方法来获取请求参数，使用了 load()方法来发送 Ajax 请求，没有指定回调函数，该页面将使用 id 为 show 的元素自动加载服务器的 HTML 响应。

该应用中处理服务器响应的 pro.jsp 页面代码如下：

程序清单：codes\11\11.7\load\pro.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
```

```
<%
//获取请求参数
String user = request.getParameter("user");
String[] books = request.getParameterValues("books");
//生成 HTML 字符串响应
out.println(user + ",您好，现在时间是:" + new java.util.Date());
out.println("<br />您喜欢的图书如下：");
out.println("<ol>");
for(int i = 0 ; i < books.length ; i++)
{
    out.println("<li>" + books[i] + "</li>");
}
out.println("</ol>");
%>
```

这是个非常简单的 JSP 页面，该 JSP 页面负责生成一个 HTML 字符串响应。当用户浏览前面的 index.html 页面，并单击“load”按钮时，将可看到 index.html 页面会自动加载 pro.jsp 页面响应，呈现如图 11.18 所示结果。

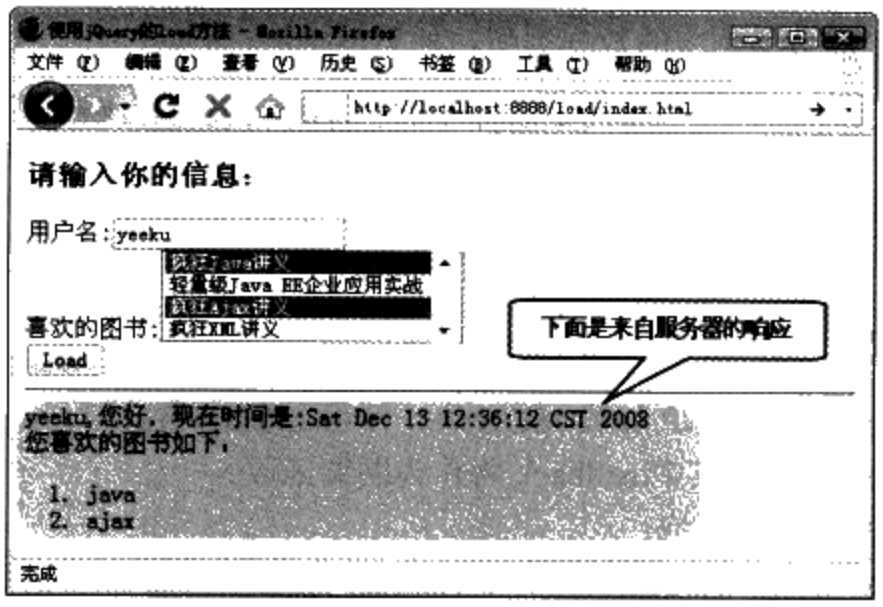


图 11.18 使用 load 方法进行 Ajax 交互

从上面的程序可以看出，使用 load 方法来发送 Ajax 请求简单便捷，开发者一样无须理会创建 XMLHttpRequest 的细节。如果开发者需要管理发送 Ajax 请求的细节，则可考虑使用 jQuery.ajax(options)方法，该方法则有点类似于 Prototype 库的 Ajax.Request，使用 jQuery.ajax 可获得 Ajax 交互的全部控制权。

➤➤11.7.3 使用 jQuery.ajax(options)方法

jQuery.ajax(options)既可以发送 GET 请求，也可以发送 POST 请求，甚至可以发送同步请求。通过使用该方法，开发者可以获得 Ajax 交互的全部控制权。

jQuery.ajax(options)是 jQuery Ajax 支持的底层实现，该方法返回其创建的 XMLHttpRequest 对象，大部分时候开发者无须理会它返回的 XMLHttpRequest 对象，但在特殊情况下可用于手动终止请求。该函数只需要一个 options 参数，该参数是一个形如{key1:val1,key2,val2...}的 JavaScript 对象，用于指定发送 Ajax 请求的各种选项，各选项的说明如下：

- async：指定是否使用异步请求，该选项默认是 true。
- beforeSend：指定发送请求之前将触发该选项指定的函数。通过指定该函数，可以在发送请求之前添加自定义的请求头。选项指定的函数是一个形如 function(xhr){...}的函数，其中 xhr 就是本次 Ajax 请求所使用的 XMLHttpRequest 对象。如果让该函数返回 false，即可取消本次 Ajax 请求。
- cache：如果该选项指定为 false，将不会从浏览器缓存里加载信息。该选项默认值为 true。如

果服务器响应是"script"，则该选项默认是 false。.

- complete：指定 Ajax 交互完成后的回调函数，该回调函数将在 success 或 error 回调函数之后被执行。该选项指定的函数是一个形如 function(xhr, textStatus){... }的函数，其中 xhr 是本次 Ajax 交互所使用的 XMLHttpRequest 对象，而 textStatus 则是服务器响应状态的描述信息。
- contentType：指定发送请求到服务器时所使用的内容编码类型。该选项的默认值是"application/x-www-form-urlencoded"，该默认值适合大多数应用场合。
- data：发送本次 Ajax 请求发送的请求参数。该选项既可使用 JavaScript 对象，也可以是查询字符串。如果指定 JavaScript 对象，系统会自动将其转换为查询字符串（除非将 processData 设为 false）。当指定该选项值为形如{key1:val1,key2:val2...}的对象，其中 valn 为数组时，系统将自动将其转换为多个请求参数，例如{foo:["bar1", "bar2"]}将会转换为'&foo=bar1&foo=bar2'。
- dataFilter：该选项执行一个回调函数，该回调函数将会对服务器响应进行预处理。该选项指定的函数是一个形如 function(data , type){...}的函数，其中 data 代表从服务器返回的响应，而 type 代表服务器响应的数据类型（也就是下面 dataType 选项指定的值）。服务器响应数据经过该选项指定的回调函数处理之后将会更加有序。
- dataType：指定服务器响应的数据类型。如果不指定，jQuery 将自动根据响应的 MIME 信息返回 responseXML 或 responseText，并将响应传给回调函数对应的参数。该选项支持如下值：
  - xml：返回可使用 jQuery 处理的 XML 文档。
  - html：返回 HTML 文本。该 HTML 文本里可以使用<script.../>标签包含 JavaScript 脚本。
  - script：返回 JavaScript 脚本，此时将禁止从浏览器缓存里加载信息。jQuery 将会自动执行服务器响应的 JavaScript 脚本。
  - json：返回一个符合 JSON 格式的字符串，jQuery 会将该响应转换成 JavaScript 对象。
  - jsonp：指定使用 JSONP 加载 JSON 块。使用 JSONP 格式时，应该在请求 URL 之后额外添加"?callback=?"，其中 callback 将作为回调函数。
  - text：返回普通文本响应。
- error：指定服务器响应出现错误的回调函数。该选项指定的函数是一个形如 function(xhr, textStatus, errorThrown){...}的函数，其中 xhr 是本次 Ajax 请求的 XMLHttpRequest 对象，textStatus 是关于错误的描述信息，errorThrown 是引起错误的错误对象。
- global：设置是否触发 Ajax 的全局事件处理函数，该选项默认是 true。
- ifModified：设置是否仅在服务器数据改变时获取新数据。系统将根据 HTTP 的 Last-Modified 响应头进行判断。该选项默认是 false。
- jsonp：该选项指定的值将会覆盖 JSONP 请求中的 callback 函数。也就是说，该选项指定的值将会覆盖查询字符串里的'callback=?'部分，即{jsonp:'onJsonPLoad'}将导致'onJsonPLoad=?'被传给服务器。
- password：指定密码。如果目标 URL 是需要安全授权的地址，则通过该选项指定密码。
- processData：指定是否需要处理请求数据。如果传给 data 选项的不是字符串，而是一个 JavaScript 对象，则 jQuery 将自动将其转换成查询字符串。如果不希望 jQuery 进行这种转换，则可将该选项指定为 false。
- scriptCharset：该选项仅对 dataType 是'jsonp'或'script'的情况有效。该选项设置系统使用给定的字符集来解释请求，仅当服务器响应和本地页面使用不同字符集时需要指定该选项。
- success：指定 Ajax 响应成功后的回调函数。该选项指定的函数是一个形如 function(xhr, textStatus){... }的函数，其中 xhr 是本次 Ajax 交互所使用的 XMLHttpRequest 对象，而 textStatus 则是服务器响应状态的描述信息。
- timeout：设置 Ajax 请求超时时长。

- type：设置发送请求的方式，最常用的两个值是"POST"和"GET"，该选项默认是"GET"。
- url：指定发送 Ajax 请求的目的 URL 地址。
- username：指定用户名。如果目标 URL 是需要安全授权的地址，则通过该选项指定用户名。
- xhr：该选项指定一个函数用于创建 XMLHttpRequest 对象。只有开发者想用自己的方式来创建 XMLHttpRequest 对象时才需要执行该选项。

通过指定上面这些选项，开发者可以全面控制 Ajax 请求的各种细节。但绝大部分情况下，开发者都不会使用 jQuery.ajax(options)来发送 Ajax 请求，而是使用另两个更简便的方法 getXxx 和 post。当开发者使用这两个方法来发送请求时，jQuery 将使用全局 Ajax 选项，为了设置全局 Ajax 选项，jQuery 提供了如下方法：

- jQuery.ajaxSetup(options)：为 jQuery 的 Ajax 交互设置全局选项，其中 options 参数和 jQuery.ajax(options)里的 options 参数的功能和意义完全一样。

➤➤11.7.4 使用 get/post 方法

jQuery 提供了如下几个简便方法来发送 GET 请求：

- jQuery.get(url, [data], [callback], [type])：向 url 发送异步的 GET 请求，其中 data 是一个 JavaScript 对象，用于指定请求参数；callback 指定服务器响应成功时的回调函数，该函数是一个形如 function(data , statusText){...}的函数，其中 data 是服务器响应，statusText 是服务器响应类型的描述信息；type 指定服务器响应数据的类型。
- jQuery.getJSON(url,[data],[callback])：该函数是前一个函数的 JSON 版本，相当于指定 type 参数为"json"。
- jQuery.getScript(url, [callback])：该函数是第一个函数的 Script 版本，相当于指定 type 参数为"script"。

下面的程序示范了使用 jQuery.get()方法来发送异步 GET 请求：

程序清单：codes\11\11.7\get\get.html

```
<h3>请输入你的信息：</h3>
<form id="user">
    用户名:<input type="text" name="user" /><br />
    喜欢的图书:<select multiple="multiple" name="books">
        <option value="java">疯狂 Java 讲义</option>
        <option value="javaee">轻量级 Java EE 企业应用实战</option>
        <option value="ajax">疯狂 Ajax 讲义</option>
        <option value="xml">疯狂 XML 讲义</option>
    </select><br />
    <input id="load" type="button" value="发送异步 GET 请求" />
</form><hr />
<div id="show"></div>
<script src="jquery-1.2.6.min.js" type="text/javascript">
</script>
<script type="text/javascript">
    //为 id 为 load 的按钮绑定事件处理函数
    $("#load").click(function()
    {
        //指定向 pro.jsp 发送请求，以 id 为 user 的表单里各表单控件作为请求参数
        $.get("pro.jsp" , $("#user").serializeArray() ,
        //指定回调函数
        function(data , statusText)
        {
            $("#show").append("服务器响应状态为：" + statusText + "<br />");
            $("#show").append(data);
```

```
        },
        //指定服务器响应为 html
        "html"
        );
    });
</script>
```

上面的页面代码使用了 jQuery.get()方法向 pro.jsp 发送异步 GET 请求，页面中的表单、表单控件、按钮等与前一个应用的基本相同，而且 pro.jsp 页面也相同，此处不再给出 pro.jsp 页面的代码。在浏览器中浏览该页面，并单击页面中发送请求的按钮，将可看到如图 11.19 所示页面。

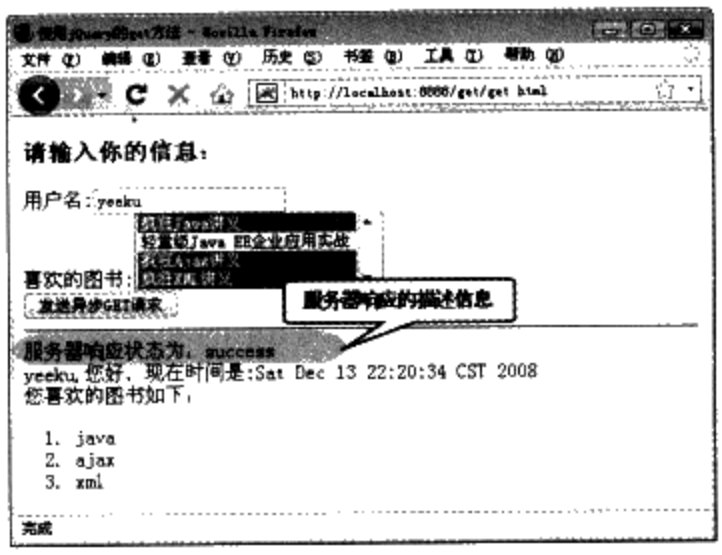


图 11.19　使用 get 方法发送请求

下面我们开发一个简单的范例程序，在该程序中我们让服务器直接生成 JavaScript 脚本响应，从而允许服务器的 JavaScript 脚本直接操作当前页面。该示例程序将使用 jQuery.getScript()方法发送请求，其 HTML 页面代码如下：

程序清单：codes\11\11.7\get\getScript.html

```
<ul style="display:none">
    <li></li>
    <li></li>
    <li></li>
    <li></li>
</ul>
<input id="get" type="button" value="getScript"/>
<div id="show"></div>
<script src="jquery-1.2.6.min.js" type="text/javascript">
</script>
<script type="text/javascript">
    //为 id 为 get 的按钮绑定事件处理函数
    $("#get").click(function()
    {
        $.getScript("script.jsp");
    });
</script>
```

上面的粗体字代码使用了 jQuery.getScript()方法发送异步 GET 请求，只指定向 script.jsp 发送请求，没有指定请求参数，也没有指定回调函数——这没有关系，script.jsp 页面将直接生成 JavaScript 响应，这些 JavaScript 脚本将直接修改当前 HTML 页面。下面是 script.jsp 页面的代码：

程序清单：codes\11\11.7\get\script.jsp

```
<%@ page contentType="text/javascript; charset=GBK" language="java" %>
$("ul>li").each(function(index)
{
    if(index % 2 == 0)
    {
        $(this).css("background-color" , "#ccffcc");
    }
    $(this).append("服务器响应" + index);
});
$("ul").slideDown(1000);
```

上面的 JSP 页面不再输出 HTML 标签，它包含的全部是 JavaScript 代码，这些 JavaScript 代码将作为响应传给客户端浏览器，并由客户端浏览器来解释执行。在浏览器中浏览前面的 getScript.html 页面，并

单击发送请求的按钮将看到如图 11.20 所示结果。

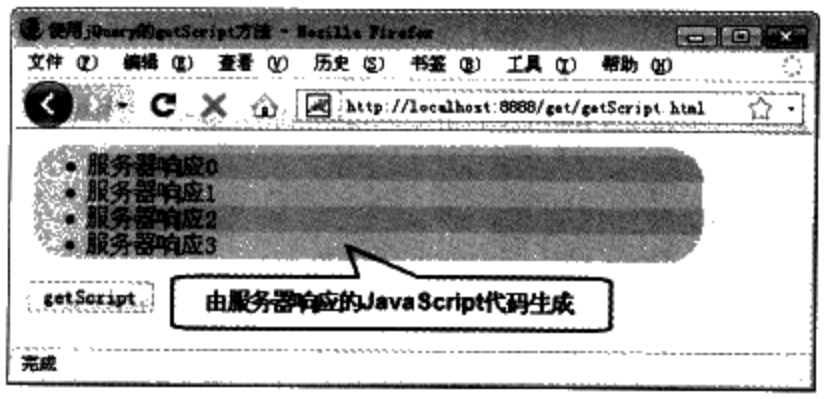


图 11.20 使用 getScript()方法获取 JavaScript 响应

jQuery 也提供了一个发送 POST 请求的方法，该方法与前面的 jQuery.get()方法并无太大的区别，甚至连参数、选项都完全相同，只是该方法发送的是异步的 POST 请求，关于该方法的详细说明如下：

- jQuery.post(url, [data], [callback], [type])：向 url 发送异步的 POST 请求。该方法中的各参数与 jQuery.get()方法中各参数的功能和意义完全相同。

因为 jQuery.get()和 jQuery.post()两个方法的用法、功能基本一样，它们的区别只是发送 GET 请求和 POST 请求的区别，故此处不再给出 jQuery.post()的例子。

提 示：

如果用户需要发送请求的请求参数量不是太大，通常使用 jQuery.get()方法即可；如果需要发送的请求参数量较大，建议使用 jQuery.post()方法发送请求。

11.8 扩展 jQuery 和 jQuery 插件

jQuery 还具有极好的可扩展性，如果开发者需要为 jQuery 增加新的函数和功能，可通过 jQuery 提供的如下两个方法来进行扩展：

- jQuery.fn.extend(object)：为所有 jQuery 对象扩展新的方法，其中 object 是一个形如{name1:fn1, name2,fn2...}的对象。经过这种扩展之后，所有 jQuery 对象都可使用 name1、name2 等方法。
- jQuery.extend(object)：为 jQuery 命名空间扩展新的方法，其中 object 是一个形如{name1:fn1, name2,fn2...}的对象。经过这种扩展之后，jQuery 命名空间下就会增加 name1、name2 等方法。

下面的程序示范了如何利用这两个方法来扩展 jQuery：

程序清单：codes\11\11.8\extend.html

```
用户名：<input name="name" type="text"/><br />
喜欢的颜色：<br />
红色：<input name="color" type="checkbox" value="red"/>
绿色：<input name="color" type="checkbox" value="green"/>
蓝色：<input name="color" type="checkbox" value="blue"/><br />
<input id="check" type="button" value="选中所有复选框"/>
<input id="uncheck" type="button" value="取消选中所有复选框"/><br />
<script type="text/javascript" src="../jquery-1.2.6.min.js">
</script>
<script type="text/javascript">
//为所有 jQuery 对象扩展新的方法
$.fn.extend(
{
    //为 jQuery 对象扩展 check 方法
    check: function() {
        //遍历 jQuery 里的每个 DOM 对象，指定其 checked 属性为 true
        return this.each(function()
        {
            this.checked = true;
        });
    },
    //为 jQuery 对象扩展 uncheck 方法
    uncheck: function()
    {
        //遍历 jQuery 里的每个 DOM 对象，指定其 checked 属性为 false
```

```
            return this.each(function()
            {
                this.checked = false;
            });
        }
    });
    $.extend(
    {
        //为 jQuery 命名空间扩展新方法
        test:function()
        {
            alert("为 jQuery 命名空间扩展的测试方法");
        }
    });
    //为 id 为 check 的按钮绑定事件处理函数
    $("#check").click(function()
    {
        $(":input").check();
    });
    //为 id 为 uncheck 的按钮绑定事件处理函数
    $("#uncheck").click(function()
    {
        $(":input").uncheck();
    });
    //调用为 jQuery 命名空间扩展的新方法
    $.test();
```

上面的程序中粗体字代码分别为 jQuery 对象、jQuery 命名空间扩展了新的方法，从而允许程序下面的粗体字代码使用这些新扩展的方法。通过这个程序可以看出，不论是第三方还是开发者，都可非常方便地扩展 jQuery，从而为 jQuery 引入新的功能。

实际上，由于 jQuery 越来越受欢迎，已经出现了大量的 jQuery 插件，这些插件极大地丰富了 jQuery 的功能。登录 http://plugins.jquery.com/站点即可看到一系列 jQuery 官方注册的 jQuery 插件，读者可以根据自己的需要选择合适的插件。

值得一提的是，jQuery 官方提供了一套优秀的界面库 jQueryUI，jQueryUI 的官方站点是 http://ui.jquery.com/，登录该站点即可看到这些丰富的 UI 效果，如果读者需要使用这些 UI 效果，可以在项目中使用 jQueryUI。

11.9 本章小结

本章详细介绍了另一个非常优秀的 JavaScript 库 jQuery，它提供了另一种优雅的设计思路：把 DOM 对象包装成 jQuery 对象进行处理，从而允许开发者不再面向 DOM 对象编程，而是面向 jQuery 对象编程。掌握这种设计思路之后，使用 jQuery 将非常简单，只要 2 步即可：①获取 jQuery 对象；②调用 jQuery 对象的方法。

本章的知识组织也遵循上面 2 个步骤：先介绍了获取 jQuery 对象的核心方法 jQuery()，并介绍了该方法支持的各种选择器和限定词。除此之外，还详细讲解了如何访问类数组的 jQuery 对象里包含的元素。本章剩下的部分则详细讲解了 jQuery 对象支持的各种方法：操作属性、操作 CSS 样式、动态更新 HTML 页面、事件编程支持、Ajax 支持等。

本章最后还详细介绍了如何扩展 jQuery，并通过示例进行了示范，并简要介绍了 jQuery 的各种插件以及 jQueryUI。

CHAPTER
12

第12章
基于jQuery的应用：电子相册系统

本章要点

- 实现系统的持久化类
- 映射 Hibernate 的持久化对象
- 扩展 HibernateDaoSupport 提供分页功能
- 基于 HibernateDaoSupport 实现 DAO 组件
- 在 Spring 容器中部署 DAO 组件
- 实现业务逻辑组件
- 部署业务逻辑组件
- 使用声明式事务机制为业务逻辑方法增加事务控制
- 使用 jQuery 的 post 方法发送异步 POST 请求
- 服务器端生成 JavaScript 更新 HTML 页面
- 处理异步请求时保存用户的浏览状态
- 使用 jQuery UI 对话框组件生成页面对话框
- 处理文件上传

本章将示范开发一个简单的电子相册系统，浏览者可以注册成本系统用户。注册用户可以选择上传相片并查看自己的相片，每个用户只能看到自己上传的相片。本系统将采用 jQuery 作为 Ajax 支持，主要使用 jQuery.post()方法发送异步 POST 请求，而且让服务器返回 JavaScript 脚本直接更新浏览器中的 HTML 页面。

除此之外，本系统还有一点值得读者注意，本系统解决了 Ajax 应用的防刷新问题。通常情况下，Ajax 应用使用浏览器的 JavaScript 来保存浏览状态，这样的后果是每当浏览者刷新页面时都会重置页面，之前的操作状态全部丢失！本应用则改变了这种做法，将浏览状态保存到 HttpSession 里，即使浏览者刷新页面，其浏览状态也不会丢失。当本系统的页面被加载完成时，JavaScript 将发送异步请求，请求将根据 HttpSession 里保存的浏览状态重新加载当前页面。

本系统还使用了 jQueryUI 的对话框。jQueryUI 对话框的用法非常简单，只要一行代码即可。在本章后面的代码中会看到 jQueryUI 对话框的用法示例。

12.1 实现持久层

本章中间层同样使用 Spring+Hibernate 来实现，Hibernate 负责持久层数据访问，依赖于持久化类操作底层数据库，而应用程序则可以面向对象的方式操作持久化对象。

➤➤12.1.1 实现持久化类

本应用需要两个表，这两个表分别用于存放用户信息和相片信息。用户信息表里主要保存了用户的用户名、密码等信息。对于一个实际使用的电子相册系统，可能还需要一些用户的详细资料、注册时间和最后访问时间等信息，但本范例不打算保存这些详细信息，这对于应用的实现没有丝毫影响。

本应用的相片表里则需要保存相片的标题、相片对应的文件名，以及该相片的属主。因此，用户表和相片表有主从表的关联关系，一个用户可对应于多张相片。相片所对应的持久化类如下：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\model\Photo.java

```
public class Photo
{
    //标识属性
    private Integer id;
    //该相片的名称
    private String title;
    //相片在服务器上的文件名
    private String fileName;
    //保存该相片所属的用户
    private User user;
    //无参数的构造器
    public Photo()
    {
    }
    //初始化全部属性的构造器
    public Photo(Integer id , String title , String fileName , User user)
    {
        this.id = id;
        this.title = title;
        this.fileName = fileName;
        this.user = user;
    }
    //省略所有属性的 setter 和 getter 方法
    ...
}
```

上面的 Photo 类中包含了一个 User 类型的属性，该属性指向另一个持久化类 User。从相片到用户

是多对一的关联，因此每个 Photo 都可访问对应的 User 实例。上面的持久化类省略了其他普通属性的 setter 和 getter 方法。

用户对应的持久化类的代码如下：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\model\User.java

```
public class User
{
    //标识属性
    private Integer id;
    //该用户的用户名
    private String name;
    //该用户的密码
    private String pass;
    //使用 Set 保存该用户关联的相片
    private Set<Photo> photos = new HashSet<Photo>();

    //无参数的构造器
    public User()
    {
    }
    //初始化全部属性的构造器
    public User(Integer id , String name , String pass)
    {
        this.id = id;
        this.name = name;
        this.pass = pass;
    }
    //省略其他属性的 setter 和 getter 方法
    ...
}
```

因为一个用户实例可以对应多个相片实例，因此用户持久化类里增加了一个 Set 属性，该属性用于保存当前用户关联的全部相片。上面的持久化类的代码省略了普通属性的 setter 和 getter 方法。

完成了上述持久化类的定义后，还应该增加 Hibernate 映射文件，Hibernate 需要映射文件才能明白持久化类和数据表、持久化类属性和数据列、持久化实例和数据记录之间的对应关系。一旦 Hibernate 明白了这种映射关系，程序就可通过持久化实例来操作底层数据库了。

下面是 Photo 实体对应的映射文件：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\model\Photo.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.album.model">
    <!-- 每个 class 元素映射一个持久化类 -->
    <class name="Photo" table="photo_table">
        <id name="id" type="int" column="photo_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="title" type="string"/>
        <property name="fileName" type="string"/>
        <!-- 映射和 User 实体的 N:1 关联 -->
        <many-to-one name="user" column="owner_id"
            class="User" not-null="true"/>
```

```
    </class>
</hibernate-mapping>
```

Photo 和 User 之间存在多对一的关联关系，所以上面的映射文件使用<many-to-one.../>元素来映射这种关联关系。

下面是 User 的映射文件：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\model\User.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
      PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.album.model">
    <!-- 每个 class 元素映射一个持久化类 -->
    <class name="User" table="user_table">
        <!-- 映射标识属性 -->
        <id name="id" type="int" column="user_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="name" type="string" unique="true"/>
        <property name="pass" type="string"/>
        <!-- 映射和 Photo 实体的 1:N 关联 -->
        <set name="photos" inverse="true">
            <key column="owner_id"/>
            <one-to-many class="Photo"/>
        </set>
    </class>
</hibernate-mapping>
```

因为 User 和 Photo 之间存在一对多的关联关系，因此上面的映射文件增加了<set.../>元素来映射 User 关联的多个 Photo 实体，并在<set.../>元素里使用<one-to-many.../>来映射 User 关联的 Photo 实体。

➤➤12.1.2 配置 SessionFactory

Hibernate 进行持久化操作需要两种 XML 文件：一种是进行数据访问的配置文件，用于指定 Hibernate 的全局属性，例如连接数据库所用的驱动、URL、用户名和密码等；另一种是 Hibernate 的映射文件，用于定义持久化类和数据表之间的映射关系。

前面我们已经提供了 Java 类和数据表之间的对应关系，但连接数据库的全局属性，例如数据库驱动、数据库服务的 URL、数据库用户名和密码等信息依然没有配置，这些通用的配置信息是通过配置 SessionFactory 来指定的。

本应用采用 Spring 管理应用的数据源、SessionFactory 等组件，因此程序可以直接在 Spring 配置文件中配置数据源、SessionFactory 等。下面是配置数据源、SessionFactory 的配置代码：

程序清单：codes\12\album\WEB-INF\applicationContext.xml

```
<!-- 定义数据源 Bean，使用 C3P0 数据源实现 -->
<bean id="dataSource" destroy-method="close"
    class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- 指定连接数据库的驱动 -->
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <!-- 指定连接数据库的 URL -->
    <property name="jdbcUrl"
        value="jdbc:mysql://localhost:3306/album"/>
    <!-- 指定连接数据库的用户名 -->
```

```
<property name="user" value="root"/>
<!-- 指定连接数据库的密码 -->
<property name="password" value="32147"/>
<!-- 指定连接数据库连接池的最大连接数 -->
<property name="maxPoolSize" value="40"/>
<!-- 指定连接数据库连接池的最小连接数 -->
<property name="minPoolSize" value="1"/>
<!-- 指定连接数据库连接池的初始化连接数 -->
<property name="initialPoolSize" value="1"/>
<!-- 指定连接数据库连接池的连接的最大空闲时间 -->
<property name="maxIdleTime" value="20"/>
</bean>
<!-- 定义 Hibernate 的 SessionFactory -->
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<!-- 依赖注入数据源，注入上面定义的 dataSource -->
<property name="dataSource" ref="dataSource"/>
<!-- mappingResouces 属性用来列出全部映射文件 -->
<property name="mappingResources">
    <list>
        <!-- 以下用来列出 Hibernate 映射文件 -->
        <value>org/crazyjava/album/model/User.hbm.xml</value>
        <value>org/crazyjava/album/model/Photo.hbm.xml</value>
    </list>
</property>
<!-- 定义 Hibernate 的 SessionFactory 的属性 -->
<property name="hibernateProperties">
    <props>
        <!-- 指定数据库方言 -->
        <prop key="hibernate.dialect">
            org.hibernate.dialect.MySQLInnoDBDialect</prop>
        <!-- 是否根据需要每次自动创建数据库 -->
        <prop key="hibernate.hbm2ddl.auto">update</prop>
        <!-- 显示 Hibernate 持久化操作所生成的 SQL -->
        <prop key="hibernate.show_sql">true</prop>
        <!-- 将 SQL 脚本进行格式化后再输出 -->
        <prop key="hibernate.format_sql">true</prop>
    </props>
</property>
</bean>
```

上面的配置文件配置了一个 sessionFactory Bean，配置该 Bean 时注入了前面配置的 dataSource，而 SessionFactory 是 Hibernate 进行持久化访问的根本，它是数据库编译后的内存镜像，SessionFactory 可产生 Session 对象，Hibernate 的持久化访问就是由 Session 来实现的。

12.2 实现 DAO 组件

本应用的持久层访问依然依赖于 DAO 组件，DAO 组件提供了数据库访问的能力，主要是对各自数据表的 CRUD 方法。

实现 DAO 组件除了依赖于 Hibernate 之外，还借助于 Spring 支持，Spring 提供了 HibernateDaoSupport 工具类，继承该类可以非常方便地实现 DAO 组件。

➤➤12.2.1 DAO 接口定义

DAO 接口仅仅定义 DAO 组件应该包含哪些方法，不会对这些方法提供实现。使用 DAO 接口的主要目的是为了实现更好的解耦。UserDao 接口的代码如下：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\dao\UserDao.java

```
public interface UserDao
{
    /**
     * 根据标识属性来加载 User 实例
     * @param id 需要加载的 User 实例的标识属性值
     * @return 指定标识属性对应的 User 实例
     */
    User get(Integer id);
    /**
     * 持久化指定的 User 实例
     * @param user 需要被持久化的 User 实例
     * @return User 实例被持久化后的标识属性值
     */
    Integer save(User user);
    /**
     * 修改指定的 User 实例
     * @param user 需要被修改的 User 实例
     */
    void update(User user);
    /**
     * 删除指定的 User 实例
     * @param user 需要被删除的 User 实例
     */
    void delete(User user);
    /**
     * 根据标识属性删除 User 实例
     * @param id 需要被删除的 User 实例的标识属性值
     */
    void delete(Integer id);
    /**
     * 查询全部的 User 实例
     * @return 数据库中全部的 User 实例
     */
    List<User> findAll();
    /**
     * 根据用户名查找用户
     * @param name 需要查找的用户的用户名
     * @return 查找到的用户
     */
    User findByName(String name);
}
```

PhotoDao 接口的代码如下：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\dao\PhotoDao.java

```
public interface PhotoDao
{
    //以常量控制每页显示的相片数
    final int PAGE_SIZE = 8;
    /**
     * 根据标识属性来加载 Photo 实例
     * @param id 需要加载的 Photo 实例的标识属性值
     * @return 指定标识属性对应的 Photo 实例
     */
    Photo get(Integer id);
    /**
     * 持久化指定的 Photo 实例
     * @param photo 需要被持久化的 Photo 实例
     * @return Photo 实例被持久化后的标识属性值
```

```
    */
    Integer save(Photo photo);
    /**
     * 修改指定的 Photo 实例
     * @param photo 需要被修改的 Photo 实例
     */
    void update(Photo photo);
    /**
     * 删除指定的 Photo 实例
     * @param photo 需要被删除的 Photo 实例
     */
    void delete(Photo photo);
    /**
     * 根据标识属性删除 Photo 实例
     * @param id 需要被删除的 Photo 实例的标识属性值
     */
    void delete(Integer id);
    /**
     * 查询全部的 Photo 实例
     * @return 数据库中全部的 Photo 实例
     */
    List<Photo> findAll();
    /**
     * 查询属于指定用户的相片，且进行分页控制
     * @param user 查询相片所属的用户
     * @param pageNo 需要查询的指定页
     * @return 查询到的相片
     */
    List<Photo> findByUser(User user , int pageNo);
}
```

这些接口定义了 DAO 组件应该实现的方法，但没有给出具体实现，具体的实现依赖于 DAO 接口的实现类。

➤➤12.2.2 完成 DAO 组件的实现类

DAO 组件的实现依赖于 Hibernate 框架，并借助于 Spring 提供的 HibernateDaoSupport 工具类，该工具类需要注入一个 SessionFactory 的引用，一旦获得了 SessionFactory 引用，就可以返回一个 HibernateTemplate 实例，HibernateTemplate 进行持久化访问非常简便，通常只需一行代码即可完成持久化访问。

本应用需要进行分页控制，而 Spring 提供的 HibernateDaoSupport 和 HibernateTemplate 都没有提供分页控制，因此本应用扩展了 HibernateDaoSupport 类，并提供了三个分页控制的方法，扩展后的 HibernateDaoSupport 子类代码如下：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\enhance\YeekuHibernateDaoSupport.java

```
public class YeekuHibernateDaoSupport extends HibernateDaoSupport
{
    /**
     * 使用 hql 语句进行分页查询操作
     * @param hql 需要查询的 hql 语句
     * @param offset 第一条记录的索引
     * @param pageSize 每页需要显示的记录数
     * @return 当前页的所有记录
     */
    public List findByPage(final String hql,
        final int offset, final int pageSize)
```

```
    {
        List list = getHibernateTemplate().executeFind(
            new HibernateCallback()
            {
                public Object doInHibernate(Session session)
                    throws HibernateException, SQLException
                {
                    List result = session.createQuery(hql)
                    .setFirstResult(offset)
                    .setMaxResults(pageSize)
                    .list();
                    return result;
                }
            });
        return list;
    }
    /**
     * 使用 hql 语句进行分页查询操作
     * @param hql 需要查询的 hql 语句
     * @param value 如果 hql 有一个参数需要传入，则 value 就是传入的参数
     * @param offset 第一条记录的索引
     * @param pageSize 每页需要显示的记录数
     * @return 当前页的所有记录
     */
    public List findByPage(final String hql , final Object value ,
        final int offset, final int pageSize)
    {
        List list = getHibernateTemplate().executeFind(
            new HibernateCallback()
            {
                public Object doInHibernate(Session session)
                    throws HibernateException, SQLException
                {
                    List result = session.createQuery(hql)
                    .setParameter(0, value)
                    .setFirstResult(offset)
                    .setMaxResults(pageSize)
                    .list();
                    return result;
                }
            });
        return list;
    }
    /**
     * 使用 hql 语句进行分页查询操作
     * @param hql 需要查询的 hql 语句
     * @param values 如果 hql 有多个参数需要传入，则 values 就是传入的参数数组
     * @param offset 第一条记录的索引
     * @param pageSize 每页需要显示的记录数
     * @return 当前页的所有记录
     */
    public List findByPage(final String hql, final Object[] values,
        final int offset, final int pageSize)
    {
        List list = getHibernateTemplate().executeFind(
            new HibernateCallback()
            {
                public Object doInHibernate(Session session)
                    throws HibernateException, SQLException
```

```
                {
                    Query query = session.createQuery(hql);
                    for (int i = 0 ; i < values.length ; i++)
                    {
                    query.setParameter( i , values[i]);
                    }
                    List result = query.setFirstResult(offset)
                    .setMaxResults(pageSize)
                    .list();
                    return result;
                }
            });
        return list;
    }
}
```

上面的 YeekuHibernateDaoSupport 类提供了三个分页查询的方法，这三个方法都利用了 HibernateTemplate 提供的 executeFind(HibernateCallback)方法，调用时需要传入一个 HibernateCallback 实例，而提供 HibernateCallback 时就可以获得 Hibernate Session，从而获得对 Hibernate 查询的全部控制权。

进行分页查询依赖于 Hibernate 提供的两个分页方法：

- setMaxResult(int PageSize)：该方法用于设置本次选择最多选出多少条记录。
- setFirstResult(int startResult)：该方法用于设置选出第一条记录的位置。

系统中所有 DAO 实现类都继承了上面的 YeekuHibernateDaoSupport 类，通过继承上面的 DAO 基类，实现 DAO 组件的方法将非常简洁，往往只需一条语句即可。下面是 UserDao 的实现类代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\dao\impl\UserDaoHibernate.java

```
public class UserDaoHibernate extends YeekuHibernateDaoSupport
    implements UserDao
{
    /**
     * 根据标识属性来加载 User 实例
     * @param id 需要加载的 User 实例的标识属性值
     * @return 指定标识属性对应的 User 实例
     */
    public User get(Integer id)
    {
        return (User)getHibernateTemplate()
            .get(User.class , id);
    }
    /**
     * 持久化指定的 User 实例
     * @param user 需要被持久化的 User 实例
     * @return User 实例被持久化后的标识属性值
     */
    public Integer save(User user)
    {
        return (Integer)getHibernateTemplate()
            .save(user);
    }
    /**
     * 修改指定的 User 实例
     * @param user 需要被修改的 User 实例
     */
    public void update(User user)
    {
        getHibernateTemplate()
            .update(user);
```

```
    }
    /**
     * 删除指定的 User 实例
     * @param user 需要被删除的 User 实例
     */
    public void delete(User user)
    {
        getHibernateTemplate()
            .delete(user);
    }
    /**
     * 根据标识属性删除 User 实例
     * @param id 需要被删除的 User 实例的标识属性值
     */
    public void delete(Integer id)
    {
        getHibernateTemplate()
            .delete(get(id));
    }
    /**
     * 查询全部的 User 实例
     * @return 数据库中全部的 User 实例
     */
    public List<User> findAll()
    {
        return (List<User>)getHibernateTemplate()
            .find("from User");
    }
    /**
     * 根据用户名查找用户
     * @param name 需要查找的用户的用户名
     * @return 查找到的用户
     */
    public User findByName(String name)
    {
        List<User> users = (List<User>)getHibernateTemplate()
            .find("from User u where u.name = ?" , name);
        if (users != null && users.size() == 1)
        {
            return users.get(0);
        }
        return null;
    }
}
```

下面是 PhotoDao 实现类的代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\dao\impl\PhotoDaoHibernate.java

```
public class PhotoDaoHibernate extends YeekuHibernateDaoSupport
    implements PhotoDao
{
    /**
     * 根据标识属性来加载 Photo 实例
     * @param id 需要加载的 Photo 实例的标识属性值
     * @return 指定标识属性对应的 Photo 实例
     */
    public Photo get(Integer id)
    {
        return (Photo)getHibernateTemplate()
            .get(Photo.class , id);
```

```
    }
    /**
     * 持久化指定的 Photo 实例
     * @param photo 需要被持久化的 Photo 实例
     * @return Photo 实例被持久化后的标识属性值
     */
    public Integer save(Photo photo)
    {
        return (Integer)getHibernateTemplate()
            .save(photo);
    }
    /**
     * 修改指定的 Photo 实例
     * @param photo 需要被修改的 Photo 实例
     */
    public void update(Photo photo)
    {
        getHibernateTemplate()
            .update(photo);
    }
    /**
     * 删除指定的 Photo 实例
     * @param photo 需要被删除的 Photo 实例
     */
    public void delete(Photo photo)
    {
        getHibernateTemplate()
            .delete(photo);
    }
    /**
     * 根据标识属性删除 Photo 实例
     * @param id 需要被删除的 Photo 实例的标识属性值
     */
    public void delete(Integer id)
    {
        getHibernateTemplate()
            .delete(get(id));
    }
    /**
     * 查询全部的 Photo 实例
     * @return 数据库中全部的 Photo 实例
     */
    public List<Photo> findAll()
    {
        return (List<Photo>)getHibernateTemplate()
            .find("from Photo");
    }
    /**
     * 查询属于指定用户的相片，且进行分页控制
     * @param user 查询相片所属的用户
     * @param pageNo 需要查询的指定页
     * @return 查询到的相片
     */
    public List<Photo> findByUser(User user , int pageNo)
    {
        int offset = (pageNo - 1) * PAGE_SIZE;
        //返回分页查询的结果
        return (List<Photo>)findByPage("from Photo b where b.user = ?"
            , user , offset , PAGE_SIZE);
    }
}
```

前面已经提到，上面的 DAO 组件都继承了 HibernateDaoSupport，因此必须为这些 DAO 组件注入 SessionFactory。Spring 为这种注入提供了方便，前面已经在 Spring 配置文件中配置了 SessionFactory，现在只需要在配置 DAO 组件时将 SessionFactory 注入 DAO 组件即可。下面是配置 DAO 组件的配置片段：

程序清单：codes\12\album\WEB-INF\applicationContext.xml

```
<!-- 配置 UserDao 组件 -->
<bean id="userDao"
    class="org.crazyjava.album.dao.impl.UserDaoHibernate">
    <!-- 注入 SessionFactory 引用 -->
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 配置 PhotoDao 组件 -->
<bean id="photoDao"
    class="org.crazyjava.album.dao.impl.PhotoDaoHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

实现了上面的 DAO 组件之后，程序即可以此 DAO 组件为基础，实现系统的业务逻辑组件。

12.3 实现业务逻辑层

业务逻辑组件依赖于底层的 DAO 组件，由 DAO 组件负责提供持久化访问功能，而业务逻辑组件则专注于提供业务逻辑功能。

➤➤12.3.1 实现业务逻辑组件

考虑到本应用的实际情况，客户端 JavaScript 代码需要访问如下几个方法：

- 处理用户登录：根据用户名和密码验证用户登录是否成功。
- 注册用户：增加一个新的系统用户。
- 增加相片：为特定的用户增加对应的相片。
- 通过用户获得指定页的所有相片。
- 验证某个用户名是否可用。

本系统的业务逻辑组件同样由接口和实现类两部分组成，不过业务逻辑组件的接口仅仅定义了上面 5 个方法，代码非常简单，故此处不再给出业务逻辑接口代码。

为了利用 Spring 的依赖注入将 DAO 组件注入业务逻辑组件，业务逻辑组件实现类应该为所依赖的 DAO 组件提供对应的 setter 方法；然后依赖于这些 DAO 组件来实现业务逻辑方法，下面是本系统中业务逻辑组件实现类的代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\serice\impl\AlbumServiceImpl.java

```
public class AlbumServiceImpl implements AlbumService
{
    //业务逻辑组件所依赖的 2 个 DAO 组件
    private UserDao ud = null;
    private PhotoDao  pd = null;
    //依赖注入 2 个 DAO 组件所需的 setter 方法
    public void setUserDao(UserDao ud)
    {
        this.ud = ud;
    }
    public void setPhotoDao(PhotoDao pd)
    {
        this.pd = pd;
    }
    /**
```

```
     * 验证用户登录是否成功
     * @param name 登录的用户名
     * @param pass 登录的密码
     * @return 用户登录的结果，成功返回 true，否则返回 false
     */
    public boolean userLogin(String name , String pass)
    {
        try
        {
            //使用 UserDao 根据用户名查询用户
            User u = ud.findByName(name);
            if (u != null && u.getPass().equals(pass))
            {
                return true;
            }
            return false;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            throw new AlbumException("处理用户登录出现异常！");
        }
    }
    /**
     * 注册新用户
     * @param name 新注册用户的用户名
     * @param pass 新注册用户的密码
     * @return 新注册用户的主键
     */
    public int registUser(String name , String pass)
    {
        try
        {
            //创建一个新的 User 实例
            User u = new User();
            u.setName(name);
            u.setPass(pass);
            //持久化 User 对象
            ud.save(u);
            return u.getId();
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
            throw new AlbumException("新用户注册出现异常！");
        }
    }
    /**
     * 添加相片
     * @param user 添加相片的用户
     * @param title 添加相片的标题
     * @param fileName 新增相片在服务器上的文件名
     * @return 新添加相片的主键
     */
    public int addPhoto(String user , String title  , String fileName)
    {
        try
        {
            //创建一个新的 Photo 实例
            Photo p = new Photo();
            p.setTitle(title);
```

```
            p.setFileName(fileName);
            p.setUser(ud.findByName(user));
            //持久化 Photo 实例
            pd.save(p);
            return p.getId();
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
            throw new AlbumException("添加相片过程中出现异常！");
        }
    }
    /**
     * 根据用户获得该用户的所有相片
     * @param user 当前用户
     * @param pageNo 页码
     * @return 返回属于该用户指定页的相片
     */
    public List<PhotoHolder> getPhotoByUser(String user , int pageNo)
    {
        try
        {
            List<Photo> pl = pd.findByUser(ud.findByName(user) , pageNo);
            List<PhotoHolder> result = new ArrayList<PhotoHolder>();
            for (Photo p : pl )
            {
                result.add(new PhotoHolder(p.getTitle() , p.getFileName()));
            }
            return result;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            throw new AlbumException("查询相片列表的过程中出现异常！");
        }
    }
    /**
     * 验证用户名是否可用，即数据库里是否已经存在该用户名
     * @param name 需要校验的用户名
     * @return 如果该用户名可用，则返回 true，否则返回 false
     */
    public boolean validateName(String name)
    {
        try
        {
            //根据用户名查询对应的 User 实例
            User u = ud.findByName(name);
            if (u != null)
            {
                return false;
            }
            return true;
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
            throw new AlbumException("验证用户名是否存在的过程中出现异常！");
        }
    }
}
```

从上面的程序可以看出，业务逻辑组件在返回 Photo 时并未直接返回 Photo 持久化类实例。根据 Java EE 规范，处于底层的 PO 实例不应该传到表现层。为了将相应的数据传到表现层，系统提供了一个简单的 VO 类（值对象），这个 VO 封装了 Photo 里的基本信息。

➤➤12.3.2 配置业务逻辑组件

到目前为止，已经完成了业务逻辑组件的实现，还应将业务逻辑组件配置在 Spring 容器中，让 Spring 的 AOP 机制为其提供声明式的事务管理，并由 Spring 为其注入 DAO 组件。

下面是 Spring 配置文件中配置业务逻辑组件并提供声明式事务管理的配置代码：

程序清单：codes\12\album\WEB-INF\applicationContext.xml

```
<!-- 配置 albumService 业务逻辑组件 -->
<bean id="albumService"
    class="org.crazyjava.album.service.impl.AlbumServiceImpl">
    <!-- 为业务逻辑组件注入 2 个 DAO 组件 -->
    <property name="userDao" ref="userDao"/>
    <property name="photoDao" ref="photoDao"/>
</bean>
<!-- 配置 Hibernate 的局部事务管理器，使用 HibernateTransactionManager 类 -->
<!-- 该类实现了 PlatformTransactionManager 接口，是针对 Hibernate 的特定实现-->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 配置 HibernateTransactionManager 时需要依赖注入 SessionFactory 的引用 -->
    <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
    <!-- 配置事务切面 Bean,指定事务管理器 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 用于配置详细的事务语义 -->
    <tx:attributes>
        <!-- 所有以'get'开头的方法是 read-only 的 -->
        <tx:method name="get*" read-only="true"/>
        <!-- 其他方法使用默认的事务设置 -->
        <tx:method name="*"/>
    </tx:attributes>
    </tx:advice>
    <aop:config>
    <!-- 配置一个切入点，匹配 lee 包下所有以 Impl 结尾的类执行的所有方法 -->
    <aop:pointcut id="leeService"
        expression="execution(* org.crazyjava.album.service.impl.*Impl.*(..))"/>
    <!-- 指定在 leeService 切入点应用 txAdvice 事务切面 -->
    <aop:advisor advice-ref="txAdvice"
        pointcut-ref="leeService"/>
</aop:config>
```

上面的配置文件中配置了 Hibernate 局部事务管理器 transactionManager，然后以该事务管理器为基础配置了一个事务切面 Bean，最后在<aop:config.../>元素中指定所有业务逻辑方法被调用时，该事务切面 Bean 都将起作用，这样就为所有业务逻辑方法都增加了事务控制。

12.4 实现客户端调用

本系统将采用 jQuery 的异步请求功能提供 Ajax 支持，而服务器响应则返回一段 JavaScript 脚本，JavaScript 脚本将会动态更新浏览者当前的 HTML 页面。本系统使用 Servlet 为异步请求提供响应，而 Servlet 则主动调用 Spring 容器中的业务逻辑组件来提供服务。

➤➤12.4.1 访问业务逻辑组件

本应用的所有业务逻辑组件都部署在 Spring 容器中，因此必须先初始化 Spring 容器才能访问业务逻辑组件，为了让 Spring 容器随 Web 应用的启动而初始化，我们在 Web 应用的 web.xml 文件中增加<listener.../>元素来配置 Listener。下面是 web.xml 文件中增加的内容：

程序清单：codes\12\album\WEB-INF\web.xml

```
<!-- 配置 Web 应用启动时加载 Spring 容器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

上面的 ContextLoaderListener 将在 Web 应用启动时自动初始化 Spring 容器，该 Listener 将会自动加载 Web 应用的 WEB-INF 路径下的 applicationContext.xml 文件。

Web 应用初始化 Spring 容器完成后，Web 应用中的 Servlet 可通过 WebApplicationContextUtils 工具类来获取 Spring 容器。为了让所有 Servlet 更好地访问 Spring 容器，程序提供了如下 Servlet 基类：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\web\base\BaseServlet.java

```
public class BaseServlet extends HttpServlet
{
    protected AlbumService as;
    //定义构造器，获得 Spring 容器的引用
    public void init(ServletConfig config)
        throws ServletException
    {
        super.init(config);
        ApplicationContext ctx = WebApplicationContextUtils
            .getWebApplicationContext(getServletContext());
        as = (AlbumService)ctx.getBean("albumService");
    }
}
```

上面的 BaseServlet 中包含了一个 as 实例属性，BaseServlet 的 init()方法负责初始化该 as 属性，初始化后的 as 属性就是 Spring 容器中的 albumService Bean，这样就使得 BaseServlet 的子类可直接通过 as 属性来访问 Spring 容器中的 albumService Bean，从而可调用该 Bean 里定义的业务逻辑方法。

➤➤12.4.2 处理用户登录

当浏览者浏览该系统而并未登录时，将看到系统首页包含了两个单行文本框，用于输入用户名和密码。图 12.1 显示了用户未登录的界面。

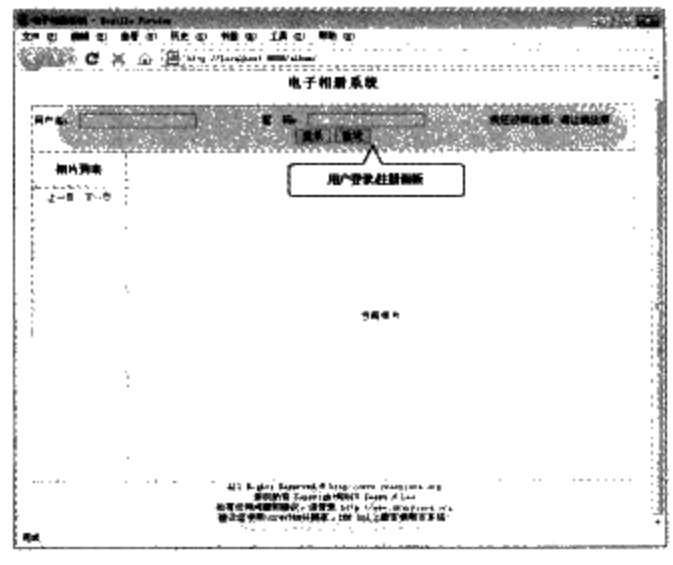


图 12.1 未登录时的界面

浏览者可以在用户名、密码输入框中输入登录的用户名、密码后单击“登录”按钮，单击“登录”按钮将触发 JavaScript 发送异步 POST 请求。下面是发送请求的 JavaScript 函数代码：

程序清单：codes\12\album\js\album.js

```
//处理用户登录的函数
function proLogin()
{
    //获取 user、pass 两个文本框的值
```

```
    var user = $.trim($("#user").val());
    var pass = $.trim($("#pass").val());
    if (user == null || user == ""
        || pass == null|| pass =="")
    {
        alert("必须先输入用户名和密码才能登录");
        return false;
    }
    else
    {
        //向 proLogin 发送异步 POST 请求
        $.post("proLogin", $('#user,#pass').serializeArray()
            , null , "script");
    }
}
```

上述登录页面里并未指定回调函数，而是指定服务器响应是 JavaScript 脚本，这样就可使用服务器响应脚本动态更新当前 HTML 页面。

上面的异步请求向 proLogin Servlet 发送，该 Servlet 将调用业务逻辑组件的 userLogin()方法来处理用户请求，并直接生成 JavaScript 脚本来更新 HTML 页面。下面是该 Servlet 的代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\web\ProLoginServlet.java

```
public class ProLoginServlet extends BaseServlet
{
    public void service(HttpServletRequest request
        , HttpServletResponse response)throws IOException,ServletException
    {
        String name = request.getParameter("user");
        String pass = request.getParameter("pass");
        response.setContentType("text/javascript;charset=GBK");
        //获取输出流
        PrintWriter out = response.getWriter();
        try
        {
            //清空 id 为 user、pass 的输入框的内容
            out.println("$('#user,#pass').val('');");
            if (name != null && pass != null
                && as.userLogin(name , pass))
            {
                HttpSession session = request.getSession(true);
                session.setAttribute("curUser" , name);
                out.println("alert('您已经登录成功！')");
                out.println("$('#noLogin').hide(500)");
                out.println("$('#hasLogin').show(500)");
                //调用获取相片列表的方法
                out.println("onLoadHandler();");
            }
            else
            {
                out.println("alert('您输入的用户名、密码不符，请重试！')");
            }
        }
        catch (AlbumException ex)
        {
            out.println("alert('" + ex.getMessage() + "请更换用户名、密码重试！')");
        }
    }
}
```

上面的 Servlet 的第一行粗体字代码指定服务器响应是 text/javascript，这表明服务器的响应是 JavaScript 脚本，而不是 HTML 代码。

上述 Servlet 使用 JavaScript 代码隐藏了 id 为 noLogin 的元素，并显示了 id 为 hasLogin 的元素——这样使得用户不再看到用户名、密码输入框，而是看到了登录后的操作菜单。

用户登录成功后，JavaScript 再次调用 onLoadHandler()方法，该方法负责获取当前用户指定页的相片列表。

用户一旦登录成功，页面上方的登录面板将立即隐藏，用户的控制面板将代替原来的登录面板，并在左边的相片列表框中列出当前用户的所有相片。而且相片框中的每个相片名都有对应的 JavaScript 处理函数，如果单击相片名，将显示出对应的相片。

如果用户输入了正确的用户名和密码，则单击“登录”按钮后，将可看到如图 12.2 所示的提示框。

用户一旦登录成功，系统将自动加载当前用户的所有相片，左边将会列出当前用户的所有相片。图 12.3 显示了用户登录成功后的界面。

图 12.2　登录成功

图 12.3　登录成功

用户注册和用户登录基本相似，用户单击图 12.1 所示页面中的“注册”链接，页面中将会显示“注册”按钮，用户单击“注册”按钮将触发 regist() JavaScript 函数，该函数负责向服务器发送异步 POST 请求，用于完成用户注册。由于用户注册、用户登录的处理流程基本相似，故此处不再赘述。

注意：

正如在前面的 ProLoginServlet 代码中所见，当用户登录成功后，系统会将当前用户名放入 HttpSession 中。当用户通过刷新按钮或 F5 刷新页面时，系统可以从 HttpSession 中取得当前浏览者的用户名，这样就可以避免在页面刷新时丢失浏览状态。而 ProRegistServlet 内也有类似的处理：在用户注册成功后，也会将当前用户名放入 HttpSession 中。

12.4.3　获得用户相片列表

获得用户的相片列表由函数 onLoadHandler()完成，该函数将通过 Servlet 调用 AlbumService 组件的 getPhotoByUser()方法来获取相片列表，由于系统需要不断获取最新的相片列表，所以 JavaScript 代码将周期性地调用 onLoadHandler()方法来获取最新的相片列表，下面是 onLoadHandler()函数的代码：

程序清单：codes\12\album\js\album.js

```
//周期性地获取当前用户当前页的相片
function onLoadHandler()
{
    //向 getPhoto 发送异步 GET 请求
    $.getScript("getPhoto");
    //指定 1 秒之后再次执行此方法
    setTimeout("onLoadHandler()", 1000);
}
```

上面的 onLoadHandler()函数向 getPhoto 发送异步 GET 请求，发送请求时也未指定回调函数，而是直接让服务器生成的 JavaScript 脚本动态更新当前 HTML 页面。下面是 getPhoto Servlet 的代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\web\GetPhotoServlet.java

```
public class GetPhotoServlet extends BaseServlet
```

```
{
    public void service(HttpServletRequest request
        , HttpServletResponse response)throws IOException,ServletException
    {
        HttpSession session = request.getSession(true);
        //从 HttpSession 中获取系统当前用户相片列表的当前页码
        String name = (String)session.getAttribute("curUser");
        Object pageObj = session.getAttribute("curPage");
        //如果 HttpSession 中的 curPage 为 null，则设置当前页为第一页
        int curPage = pageObj == null ? 1 :(Integer) pageObj;
        response.setContentType("text/javascript;charset=GBK");
        //获取输出流
        PrintWriter out = response.getWriter();
        try
        {
            List<PhotoHolder> photos = as.getPhotoByUser(name , curPage);
            //清空 id 为 list 的元素
            out.println("var list = $('#list').empty();");
            for (PhotoHolder ph : photos)
            {
                //将每个相片动态添加到 id 为 list 的元素中
                out.println("list.append(\"<div align='center'>" +
                    "<a href='javascript:void(0)' onclick=\\\"showImg('" +
                    ph.getFileName() + "');\\\">"+ ph.getTitle() + "</a></div>\");");
            }
        }
        catch (AlbumException ex)
        {
            out.println("alert('" + ex.getMessage() + "请重试！')");
        }
    }
}
```

从上面的代码中可以看出，该 Servlet 将会从 HttpSession 中读取 curUser、curPage 两个属性，其中 curUser 属性记录了浏览者的浏览状态：当前正在浏览哪一页！如果无法读到 curUser 属性，则系统默认加载第一页。从这个设计可以看出，本系统把用户正在浏览哪页的状态保存在服务器端，而不是浏览器中，这样设计可保证：即使用户刷新当前页面，用户的浏览状态也不会丢失。

将正在浏览的页码保存在服务器端还有一个方便之处：当系统需要进行翻页时，只要修改 HttpSession 里的 curPage 属性即可，无须进行额外处理。

onLoadHandler()函数是个周期性执行的函数，它会周期性地向服务器发送异步 GET 请求。这个函数在如下时候会获得执行机会：

- 用户登录成功后。
- 用户注册成功后。
- 页面加载完成时。

一旦 onLoadHandler()函数执行起来，它将每隔 1 秒执行一次，不断获取最新的相片列表。

➤➤12.4.4 处理翻页

正如前面所提到的，系统处理翻页操作比较简单，因为用户正在浏览的页码保存在 HttpSession 中，因此处理翻页只要修改 HttpSession 里的 curPage 属性即可。

当用户单击如图 12.3 所示页面左边的“上一页”、“下一页”链接时，系统将会触发翻页请求，翻页请求由如下 JavaScript 函数发送。

程序清单：codes\12\album\js\album.js

```
//处理翻页的函数
function turnPage(flag)
{
    $.getScript("turnPage?turn=" + flag);
}
```

处理翻页的 Servlet 是 turnPage，该 Servlet 类代码如下：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\web\TurnPageServlet.java

```
public class TurnPageServlet extends BaseServlet
{
    public void service(HttpServletRequest request
        , HttpServletResponse response)throws IOException,ServletException
    {
        String turn = request.getParameter("turn");
        HttpSession session = request.getSession(true);
        String name = (String)session.getAttribute("curUser");
        Object pageObj = session.getAttribute("curPage");
        //如果 HttpSession 中的 curPage 为 null，则设置当前页为第一页
        int curPage = pageObj == null ? 1 :(Integer) pageObj;
        response.setContentType("text/javascript;charset=GBK");
        PrintWriter out = response.getWriter();
        if (curPage == 1 && turn.equals("-1"))
        {
            out.println("alert('现在已经是第一页，无法向前翻页！')");
        }
        else
        {
            //执行翻页，修改 curPage 的值
            curPage += Integer.parseInt(turn);
            try
            {
                List<PhotoHolder> photos = as.getPhotoByUser(name , curPage);
                //翻页后没有记录
                if (photos.size() == 0)
                {
                    out.println("alert('翻页后找不到任何相片记录，系统将自动返回上一页')");
                    //重新返回上一页
                    curPage -= Integer.parseInt(turn);
                }
                else
                {
                    //把用户正在浏览的页码放入 HttpSession 中
                    session.setAttribute("curPage" , curPage);
                }
            }
            catch (AlbumException ex)
            {
                out.println("alert('" + ex.getMessage() + "请重试！')");
            }
        }
    }
}
```

从上面的代码可以看出：程序仅仅修改了 HttpSession 里 curPage 的属性值，程序将根据 turn 参数来决定向前翻页或向后翻页——当 turn 变量的值是 1 时，系统将执行向前翻页；当 turn 变量的值是-1 时，系统将执行向后翻页。

➤➤12.4.5 处理文件上传

很多人以为：Ajax 技术可以很方便地实现无刷新的文件上传。实际上，这里存在一个障碍：根据安全性需要，JavaScript 代码不能访问客户端文件系统（任何人都不希望浏览某个网页后，满磁盘都是蠕虫和木马吧？）。借助 Internet Explorer 里的 FSO（File System Object）对象，JavaScript 可以访问浏览者的文件系统，但这终究不是一个好主意：局限性太大，这种访问必须要得到用户的同意。如果 JavaScript 不能访问用户文件系统，那么 XMLHttpRequest 的请求参数就无法获得上传文件的文件内容，而只能获得上传文件的文件名。

XMLHttpRequest 只能将需要上传的文件名发送到服务器，但服务器获得了需要上传的文件名没有任何意义，因为服务器依然不能访问客户端的文件系统，这就是通过 Ajax 技术实现无刷新的文件上传的最大障碍。

本系统使用 jQueryUI 的对话框来进行文件上传，在上传成功后，将自动返回系统主界面。当用户单击如图 12.3 所示窗口中的“增加相片”链接时，将会触发如下 JavaScript 函数。

程序清单：codes\12\album\js\album.js

```
//打开上传窗口
function openUpload()
{
    $("#uploadDiv").show()
        .dialog(
        {
            modal: true,
            resizable: false,
            width: 428,
            height: 220,
            overlay: {opacity: 0.5 , background: "black"}
                });
}
```

上面的函数调用了 jQueryUI 的 dialog()方法，该方法将会在当前页面打开一个对话框，单击“增加相片”链接将可看到如图 12.4 所示对话框。

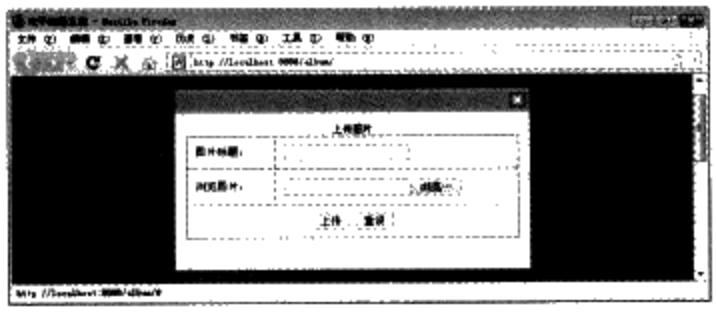


图 12.4 上传图片的对话框

用户单击如图 12.4 所示对话框中的“上传”按钮后，页面将发送同步 POST 请求，请求将提交到 proUpload Servlet，该 Servlet 负责处理文件的上传以及调用 AlbumService 的方法添加相片。上传还使用了另一个开源项目：commons-fileupload，因此应该将 commons-fileupload.jar 文件添加到应用的 WEB-INF/lib 路径下。下面是处理上传的 Servlet 的代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\web\ProUploadServlet.java

```
public class ProUploadServlet extends BaseServlet
{
    public void service(HttpServletRequest request ,
        HttpServletResponse response) throws IOException,ServletException
    {
        Iterator iter = null;
        String title = null;
        response.setContentType("text/html;charset=GBK");
        //获取输出流
        PrintWriter out = response.getWriter();
        out.println("<script type='text/javascript>'");
        try
```

```
{
    //使用 Uploader 处理上传
    FileItemFactory factory = new DiskFileItemFactory();
    ServletFileUpload upload = new ServletFileUpload(factory);
    List items = upload.parseRequest(request);
    iter = items.iterator();
    //遍历每个表单控件对应的内容
    while (iter.hasNext())
    {
        FileItem item = (FileItem) iter.next();
        //如果该项是普通表单域
        if (item.isFormField())
        {
            String name = item.getFieldName();
            if (name.equals("title"))
            {
                title = item.getString("GBK");
            }
        }
        //如果是需要上传的文件
        else
        {
            String user = (String)request.getSession()
                .getAttribute("curUser");
            String serverFileName = null;
            //返回文件名
            String fileName = item.getName();
            //取得文件后缀名
            String appden = fileName.substring(fileName.lastIndexOf("."));
            //返回文件类型
            String contentType = item.getContentType();
            //只允许上传 JPG、GIF、PNG 图片
            if (contentType.equals("image/pjpeg")
                || contentType.equals("image/gif")
                || contentType.equals("image/jpeg")
                || contentType.equals("image/png"))
            {
                InputStream input = item.getInputStream();
                serverFileName = String.valueOf(System.currentTimeMillis());
                FileOutputStream output = new FileOutputStream(
                    getServletContext().getRealPath("/")
                    + "uploadfiles\\" + serverFileName + appden);
                byte[] buffer = new byte[1024];
                int len = 0;
                while((len = input.read(buffer)) > 0 )
                {
                    output.write(buffer , 0 , len);
                }
                input.close();
                output.close();
                as.addPhoto(user , title , serverFileName + appden);
                response.sendRedirect("album.html?resultCode=0");
            }
            else
            {
                response.sendRedirect("album.html?resultCode=1");
            }
        }
```

```
            }
        }
        catch (FileUploadException fue)
        {
            fue.printStackTrace();
            response.sendRedirect("album.html?resultCode=2");
        }
        catch (AlbumException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面的程序中第一行粗体字代码调用 as 的 addPhoto()方法添加了一个新的相片，上面的 Servlet 在处理完用户上传请求后将重定向到 album.html，就是本系统使用的视图页面，重定向到 album.html 页面时还额外增加了一个 resultCode 参数，该参数标识了处理上传的效果。

在该 Servlet 处理上传成功后，将可看到如图 12.5 所示的对话框。

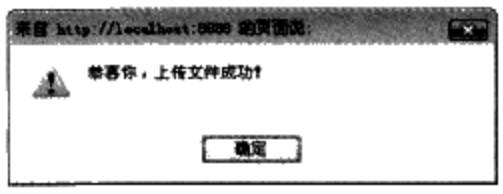
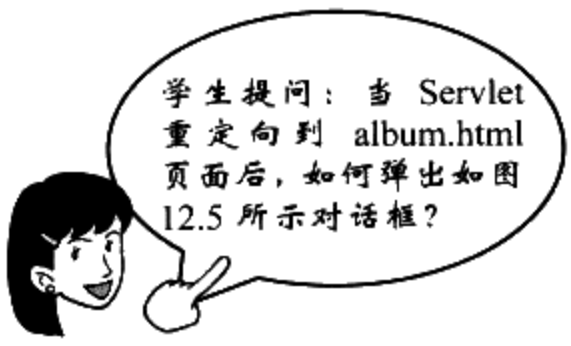
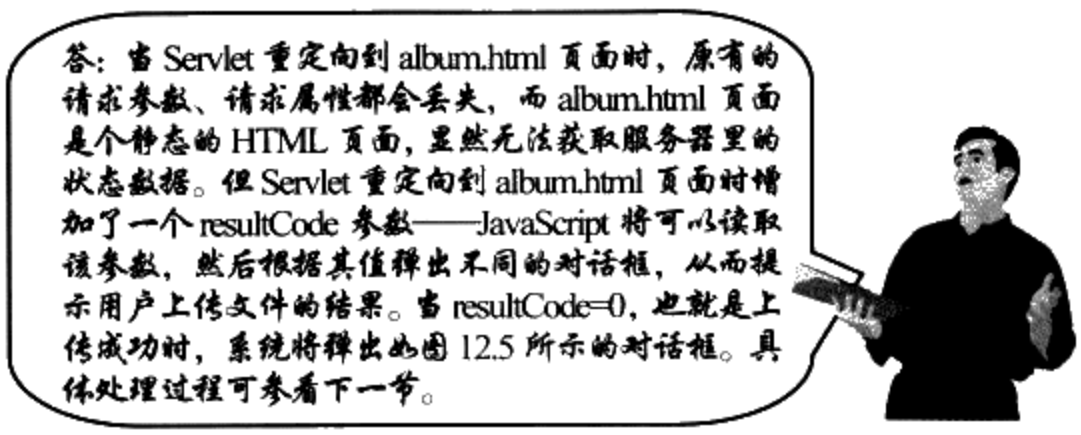


图 12.5　上传成功





如果单击图 12.5 中的“确定”按钮，系统将自动回到主页面。回到主页面后，可看到刚刚添加的相片已经被列在左边的相片列表中。单击任一相片标题，即可看到相片在右边显示出来。如图 12.6 所示为系统显示指定相片的效果。

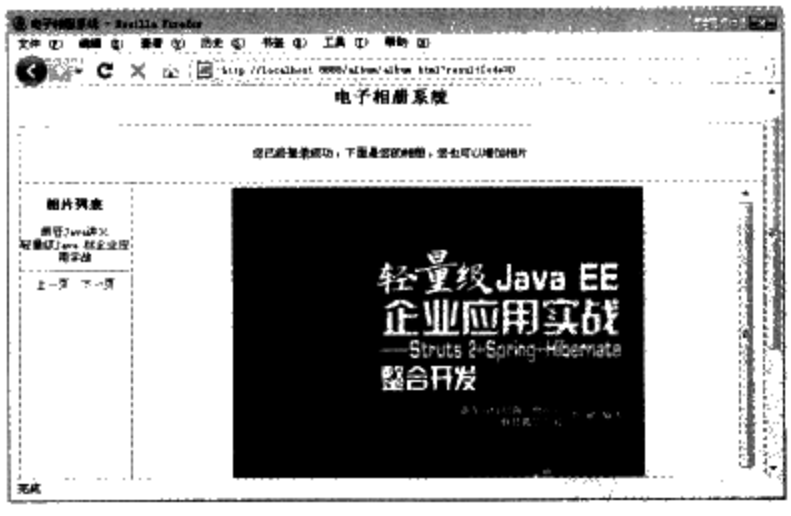


图 12.6　显示相片

➤➤12.4.6　页面加载时的处理

本系统将用户的浏览状态都放在 HttpSession 中保存，而不是直接放在客户端保存，这样可保证用户刷新页面时不会丢失浏览状态。程序通过如下代码指定页面加载后的行为：

程序清单：codes\12\album\js\album.js

```
$(document).ready(function(){
    $.getScript("pageLoad");
    //处理地址栏的 resultCode 参数
    var locationStr = document.location.toString();
    var resultIndex = locationStr.indexOf("resultCode");
    var resultCode = -1;
    if (resultIndex > 1)
    {
        resultCode = locationStr.substring(resultIndex + 11
            , resultIndex + 12);
        //根据不同的 resultCode，系统进行不同处理
        switch(resultCode)
        {
            case "0" :
                alert('恭喜你，上传文件成功！');
                $('#uploadDiv').dialog('close');
                break;
            case "1" :
                alert('本系统只允许上传 JPG、GIF、PNG 图片文件，请重试！');
                $('#title,#file').val('');
                break;
            case "2" :
                alert('处理上传文件出现错误，请重试！');
                $('#title,#file').val('');
                break;
        }
    }
});
```

从上面的粗体字代码可以看出，当页面加载完成后，JavaScript 将会向 pageLoad Servlet 发送异步 GET 请求，并让服务器响应的 JavaScript 脚本直接更新当前页面。下面是 pageLoad Servlet 的代码：

程序清单：codes\12\album\WEB-INF\src\org\crazyjava\album\web\PageLoadServlet.java

```
public class PageLoadServlet extends BaseServlet
{
    public void service(HttpServletRequest request
        , HttpServletResponse response)throws IOException,ServletException
    {
        response.setContentType("text/javascript;charset=GBK");
        //获取输出流
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession(true);
        String name = (String)session.getAttribute("curUser");
        //如果 name 不为 null，表明用户已经登录
        if (name != null)
        {
            //隐藏 id 为 noLogin 的元素（用户登录面板）
            out.println("$('#noLogin').hide()");
            //隐藏 id 为 hasLogin 的元素（用户控制面板）
            out.println("$('#hasLogin').show()");
            //调用获取相片列表的方法
            out.println("onLoadHandler();");
            //取出 HttpSession 中的 curImg 属性
            String curImg = (String)session.getAttribute("curImg");
            //重新显示用户正在浏览的相片
            if (curImg != null)
            {
                out.println("$('#show').attr('src' , 'uploadfiles/"
                    + curImg + "');");
```

```
        }
    }
}
```

从上面的代码中可以看出，该 Servlet 会读取 HttpSession 里的 curUser、cur Img 两个属性，其中 curUser 用于标识当前用户是否已经登录——如果用户已经登录，系统将隐藏登录面板，显示用户控制面板。curImg 则用于记录用户正在浏览的相片，如果该属性存在，则系统将根据它来加载相片。

12.5 本章小结

本章示范开发了一个简单的电子相册系统，系统中间层采用 Spring+Hibernate 提供服务，其中 Hibernate 负责访问持久层数据，而 Spring 则负责管理容器中的数据源、SessionFactory、DAO 组件、业务逻辑组件等，并负责管理各组件之间的依赖关系，Spring 的 AOP 机制还负责为业务逻辑组件提供事务控制。

本应用采用 jQuery 作为 Ajax 支持，本章的系统中发送异步请求时都没有指定回调函数，而是让服务器响应生成 JavaScript 脚本来更新当前 HTML 页面。除此之外，该示例应用还有一点需要指出：本应用将用户的浏览状态（当前用户名、正在浏览相册列表的哪页、正在浏览哪张相片）都放入了 HttpSession 里，这样就可避免用户刷新页面后丢失之前的浏览状态。本应用中还使用了 jQueryUI 的对话框组件来创建页面对话框，

➤➤本章练习

完善本电子相册系统，可以考虑从如下几个方面进行完善：

- 完善本系统的界面，让应用的用户界面更加美观。
- 增加相片管理功能，例如允许用户修改相片说明、删除相片。
- 增加权限控制（可考虑使用 Filter 实现），只有登录用户才可添加相片、管理相片。
- 增加相片分类，允许普通浏览者根据分类浏览其他用户的相片。
- 增加相片评论功能，允许浏览者对指定相片发表评论。

CHAPTER
13

第13章 DWR 框架详解

本章要点

- DWR 的介绍
- DWR 对处理类的要求
- 配置 DWR 配置文件
- 使用转换器
- 声明方法签名
- 使用创建器
- DWR 异步调用的通用配置
- 使用回调函数异步调用 Java 方法
- 支持更多选项的异步调用
- 使用 engine.js 核心 JavaScript 库
- 让处理类访问 Servlet API
- 让处理类读取其他 URL 的响应
- 异步调用 Spring 容器中 Bean 的方法
- 使用 Annotation 代替 dwr.xml 配置文件
- DWR 的异常处理
- 反向 Ajax 用法和机制
- 使用 ScriptSession 和 Util 操作 HTML 页面

DWR（Direct Web Remoting）是在 Apache 许可下的一个开源项目，它是一个非常专业的 Java EE Ajax 框架。通过使用 DWR 框架，可以将 Java 组件的方法直接暴露给 JavaScript 客户端。这种方式，非常类似于 Java 的远程方法调用，不同的是 DWR 中的客户端是 JavaScript 代码，而不是 Java 代码。

除此之外，DWR 还提供了一套 JavaScript 函数集，可用于可以简化 DOM 元素的操作，例如动态修改表格，动态修改列表框、下拉菜单等。

DWR 的专业还体现在与其他 Java EE 框架的整合上：DWR 可以与大名鼎鼎的 Spring 无缝整合，允许直接调用 Spring 容器中的 Bean，这是多么令人兴奋的事情啊。不仅如此，DWR 允许访问 Servlet API，还可以与 Struts、WebWork 和 JSF 等众多框架整合。这种整合非常有利于将 DWR 框架融入实际 Java EE 项目。

13.1 DWR 的下载和安装

DWR 的全称是 Direct Web Remoting，也就是直接 Web 远程调用。曾经有一个笑话：如果你流落荒岛，但是只能带一个 Ajax 框架，建议你选择 DWR。这个笑话虽然有些夸张，但从某个侧面反映了 DWR 框架在 Java EE 的 Ajax 领域的地位。

➤➤13.1.1 什么是 DWR

DWR 是一个 Java EE 领域的 Ajax 框架，通过 DWR 的帮助，可以帮助开发者更简单地开发出 Ajax 应用。通过 DWR 的帮助，开发者可以在在浏览器中的 Javascript 代码调用远程的 Java 方法，就像在这些 Java 方法就是在客户端定义的一样。

DWR 框架允许客户端 JavaScript 代码直接调用远程的 Java 方法，这种调用非常类似于 WebService 技术的 RPC（Remote Procedure Call，远程过程调用）调用，因此，DWR 也被称为 RPC 风格的 Ajax 框架。

DWR 框架主要包括如下两个部分：

- 客户端的 JavaScript，这部分代码使用客户端 JavaScript 可以直接调用远程服务器的 Java 方法。除此之外，DWR 还提供了一些方便的工具函数来简化 DOM 操作。
- 服务器上运行的 Servlet 负责处理用户请求，并将用户请求委托到实际 Java 对象进行处理，并负责把处理结果返回客户端。

DWR 会根据 Java 类动态地生成 JavaScript 代码，通过这些动态生成的 JavaScript 代码，将服务器端的 Java 方法直接暴露给客户端的 JavaScript，让用户产生一种错觉：通过使用 DWR 框架的帮助，客户端 JavaScript 代码直接调用远程 Java 方法。实际上，DWR 依然是依赖 XMLHttpRequest 对象和服务器进行通信。其基本原理是：当开发者直接调用远程 Java 方法时，DWR 会负责将这种调用转换成对应的 Ajax 请求，并使用 XMLHttpRequest 将请求发送到远程服务器端。当服务器处理完成后，DWR 负责将处理结果传回客户端的 JavaScript 代码。

在整个 Ajax 交互过程中，DWR 负责数据的传递和转换。

这种远程的调用从外表上看，非常类似于 RMI（Remote Method Invoke，远程方法调用）或 WebServices 的 RPC 机制。因此，DWR 的 Ajax 交互也被称为 RPC 风格的 Ajax 调用。而且，Java 的 RMI 调用是同步的，而 Ajax 调用是异步的。因此，使用 DWR 调用远程 Java 方法时，应该增加一个 JavaScript 的回调函数，当远程调用的结果成功返回时，这个回调函数会自动执行，回调函数负责将服务器返回的数据显示在当前页面上。

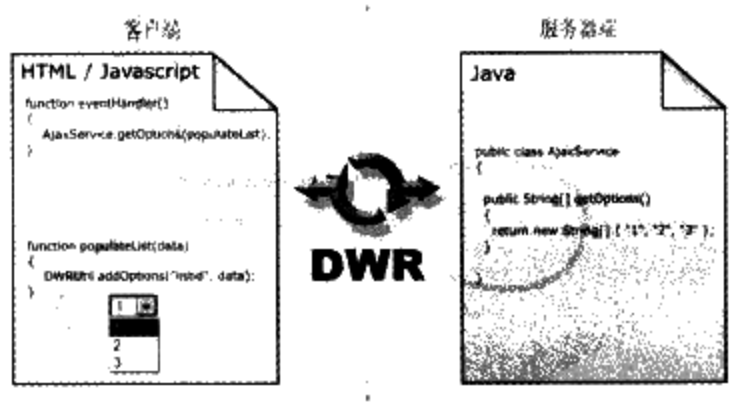


图 13.1 如使用 DWR 改变 Select 下拉框的选项

图 13.1 显示了如何通过 DWR 改变下拉菜单的选项。

DWR 为服务端的 AjaxService 类生成了一个相应的客户端 AjaxService 对象，从而将服务器端的 Java 对象暴露成客户端的 JavaScript 对象。当客户端 JavaScript 代码调用这个 JavaScript 对象时，DWR 负责处理整个远程调用的细节。

当处理远程方法返回值时，再次利用了 DWR 提供的工具对象 dwr.util，这个 JavaScript 对象主要用于简化 DOM 操作。实际上，DWR 提供了相当多的工具函数，用于简化 DOM 操作。除非是非常复杂的 DOM 操作，我们才需要自己使用原生的 JavaScript 操作 DOM。

提 示：

当然，DWR 框架不会与前面介绍的 JavaScript 函数库，例如 Prototype、JQuery 等冲突，因此完全可以在同一项目中混合使用 DWR 和 JQuery 等。这样即可利用 DWR 的 RPC 风格，又可利用 Prototype、JQuery 等库里的工具函数。

➤➤13.1.2 下载和安装 DWR

笔者写作本书之时，DWR 稳定的产品化版本是 2.0.5。该版本也是笔者所介绍的 DWR 的版本。下载和安装 DWR 请按如下步骤进行：

（1）登陆 http://directwebremoting.org/dwr/download 站点，下载 DWR 的最新版本。通常有如下三个下载选项供选择：

- JAR File: dwr.jar：该选项仅下载 DWR 的核心 JAR 文件。
- WAR File: dwr.war：该选项不仅下载了 DWR 的核心 JAR 文件，也包括了 DWR 的范例应用，还包括了运行 DWR 应用的各种依赖类库。
- Sources: dwr-2.0.rc1-src.zip：该选项用于下载 DWR 的源文件。

通常建议读者下载第三个选项，该选项不仅包含 DWR 的各种核心 JAR 文件，也包括 DWR 的范例应用。除此之外，该选项还包括 DWR 的源代码和 API 文档等相关内容。

将下载到的 zip 文件解压缩，该文件就是一个典型的 Web 结构，该文件夹包含如下有用的文件夹和文件：

- demo：该文件夹包含 DWR 示例应用所用的源代码。
- jar：该文件夹下包含 DWR 框架编译和运行时所依赖的第三方类库。
- java：该文件夹下包含 DWR 框架的源代码。
- javadoc：该文件夹下包含 DWR 框架的 API 文档。
- web：该文件夹就是 DWR 框架的示例应用。将该文件夹直接复制到 Web 服务器的自动部署目录下，即可运行该示例应用。
- dwr.jar：该文件是 DWR 框架的 JAR 包。
- LICENSE.txt 等授权、说明文档。

（2）将上面提到的 dwr.jar 文件复制到 Web 应用的 WEB-INF/lib 下。

提 示：

除此之外，DWR 运行时还依赖于 commons-logging-1.0.4.jar，因此还应该将 jar 文件夹下的 commons-logging-1.0.4.jar 文件复制到 Web 应用的 WEB-INF/lib 路径下。

（3）编写配置文件。成功安装 DWR 需要先修改 web.xml 文件，修改 web.xml 文件保证特定请求被转发给 DWR 的核心 Servlet 处理，而 dwr.xml 文件则负责定义 Java 类和 JavaScript 对象之间的对应关系。在 web.xml 文件中增加如下配置片段：

程序清单：codes\13\13.1\install\WEB-INF\web.xml

```
<!-- 配置 DWR 的核心 Servlet -->
<servlet>
    <!-- 指定 DWR 核心 Servlet 的名字 -->
    <servlet-name>dwr-invoker</servlet-name>
```

```
        <!-- 指定 DWR 核心 Servlet 的实现类 -->
        <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
        <!-- 指定 DWR 核心 Servlet 处于调试状态 -->
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <!-- 指定核心 Servlet 的 URL 映射 -->
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <!-- 指定核心 Servlet 映射的 URL -->
        <url-pattern>/leedwr/*</url-pattern>
    </servlet-mapping>
```

相信熟悉 Web 应用的读者可以明白上面配置片段的大致用处，它指定所有向/leedwr/*的请求，都将由 uk.ltd.getahead.dwr.DWRServlet 负责处理，该 Servlet 就是 DWR 框架的核心类。而配置该 Servlet 时的 debug 属性指定 DWR 是否处于调试状态，当 debug 状态时，DWR 会提供一个调试页面。如果作为产品发布时，该属性应该设置为 false。

注意：

一旦当成产品发布 DWR 应用时，该 debug 属性应该设置为 false。否则可能导致安全隐患。

除此之外，还必须增加一个 dwr.xml 文件，该文件负责定义 Java 类和 JavaScript 对象之间的对应关系。下面是一个简单的定义文件，定义了 java.util.Date 和 JavaScript 对象之间的对应关系。

程序清单：codes\13\13.1\install\WEB-INF\dwr.xml

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 DWR 配置文件的 DTD 等信息 -->
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr20.dtd">
<!-- DWR 配置文件的根元素是 dwr -->
<dwr>
    <allow>
        <!-- 使用 new 创建一个 JavaScript 对象，该对象名为 JDate -->
        <create creator="new" javascript="JDate">
            <!-- 创建 Jdate 对象使用的 Java 类为 java.util.Date -->
            <param name="class" value="java.util.Date"/>
        </create>
    </allow>
</dwr>
```

启动 Web 服务器，笔者以 Tomcat6.0.28（服务端口为 8888）为例，然后访问如下地址：

```
http://localhost:8888/install/leedwr/
```

其中 8888 是 Web 服务器的端口号，install 是该应用的 URL，leedwr 是配置 DWR 核心 Servlet 时指定的 URL。这样可以看到如图 13.2 所示的界面。

图 13.2 成功安装 DWR 后的提示界面

注意：

必须将 debug 属性设置成 true 才会看到如图 13.2 所示的界面。当设置 debug 属性为 true 时，表明 DWR 应用处于 debug 模式下，DWR 会为每一个远程调用类生成一个测试页面（如图 13.3 所示），这对于检查 DWR 的工作状态非常有用。

debug 模式还可以警告开发者一些可能存在的问题（如图 13.3 中红色字体部分）。因此，debug 模式不应该使用在实际部署环境里面，因为这些测试页面为攻击者提供了的大量信息。虽然，如果采用严格的安全控制系统，即使获得这些信息，网络上的 Cracker 应该不会有任何机会。但不要给 Cracker 任何机会，所以在发布 DWR 应用时，将 debug 设置为 false 是不错的选择。

提 示：

从图 13.2 可以看出，DWR 可以将任何 Java 类转换成一个 JavaScript 对象，不管这个 Java 类是系统提供的，还是用户自己开发的。当然，实际 Ajax 应用中通常会选择将自己的 Java 类转换成 JavaScript 对象。

图 13.2 显示的界面实际上是 DWR 提供的调试界面，该调试界面表明在已经成功产生了一个 JavaScript 对象，该对象名为 JDate。如果我们单击 JDate 链接，将看到如图 13.3 所示的界面。

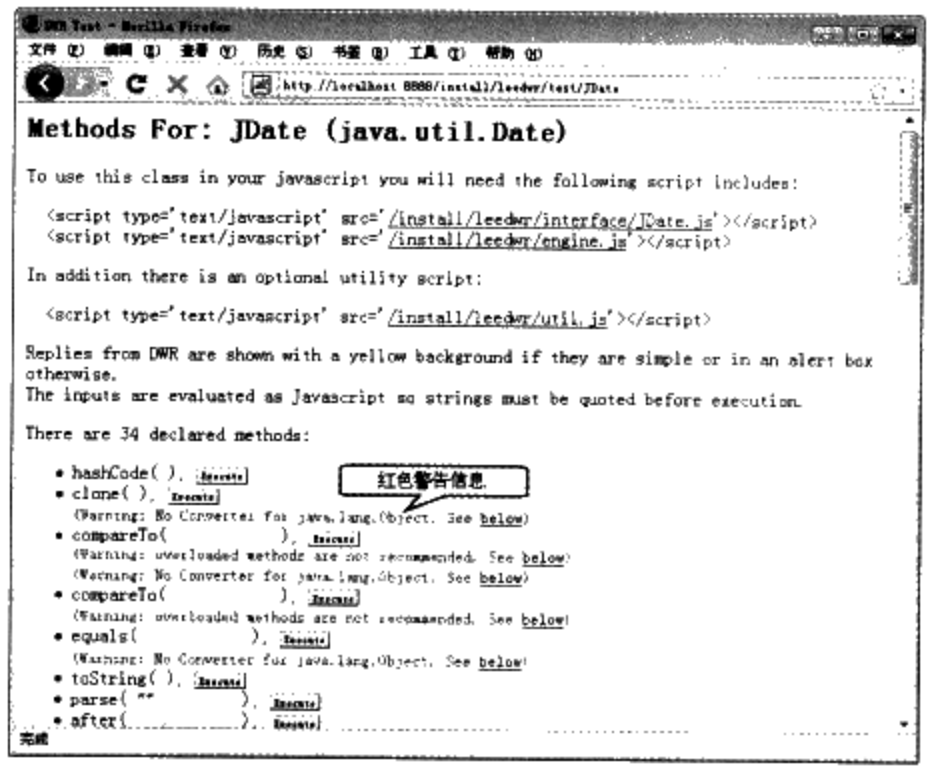


图 13.3 被成功导出的 JavaScript 对象

按图 13.3 中文本提示的，如果为了某个页面中使用 JDate 对象，只需在该页面中使用如下代码导入 JavaScript 代码库即可。

```
<script type='text/javascript' src='/install/leedwr/interface/JDate.js'></script>
```

可能读者已经注意到，从开始到现在，我们从未编写过任何 JDate.js 文件。但这个文件确实存在，DWR 动态生成了该 JavaScript 文件，这就是 DWR 的魅力。除此之外，DWR 还动态生成了另外两个工具 JavaScript 库，这两个 JavaScript 库也是 DWR 应用不可缺少的，因此应增加如下两行代码：

```
<script type='text/javascript' src='/install/leedwr/engine.js'></script>
<script type='text/javascript' src='/install/leedwr/util.js'></script>
```

在图 13.3 中可以看到 Date 类所包含的所有方法，从其父类、与 Date 类的所有方法，都被导出成 JDate 对象（JavaScript 对象）的方法。这可能引起一个问题，如果 Date 类包含一个方法，该方法名是 JavaScript 的关键字，这必然引起代码出错。因此，当定义一个服务器处理类时，处理类的方法名尽量不要使用 JavaScript 关键字作为方法名；而且由于 JavaScript 不支持方法重载，因此如果服务器处理类

的使用了方法重载，也将导致混乱。

> **注意：**
> 处理类中的方法名不要使用 JavaScript 关键字，也不要使用方法重载。

如果需要在应用中使用多个 dwr.xml 文件，则可以在配置 DWR 核心 Servlet 时增加如下的配置片段：

```
<init-param>
    <param-name>config*****</param-name>
    <param-value>WEB-INF/dwr.xml</param-value>
</init-param>
```

在这里 config*****意思是 param-name 要以字符串 config 开头。每次配置 DWR 的核心 Servlet 时，都可以使用这段配置片段，但每个 Servlet 定义只能有一段这样的配置片段。

下面的配置片段使用了多个 dwr.xml 配置文件，不同配置文件对应的 DWR 核心 Servlet 映射到不同的 URL

```
<servlet>
    <servlet-name>dwr-user-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
        <param-name>config-user</param-name>
        <param-value>WEB-INF/dwr-user.xml</param-value>
    </init-param>
</servlet>
<servlet>
    <servlet-name>dwr-admin-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
        <param-name>config-admin</param-name>
        <param-value>WEB-INF/dwr-admin.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>dwr-admin-invoker</servlet-name>
    <url-pattern>/dwradmin/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>dwr-user-invoker</servlet-name>
    <url-pattern>/dwruser/*</url-pattern>
</servlet-mapping>
```

正如上面配置文件见到的，定义多个 dwr.xml 文件通常有如下作用：

- 让多个远程调用类分开定义，将不同的远程类按功能模块定义在不同的配置文件中。
- DWR 可以使用 JAAS 的安全机制。将不同 DWR 核心 Servlet 的 URL 与特定用户权限关联起来，从而提供简单的权限控制。

如果在上面的 web.xml 文件中增加如下片段：

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>dwr-admin-collection</web-resource-name>
        <url-pattern>/dwradmin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
```

```
</security-constraint>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>dwr-user-collection</web-resource-name>
        <url-pattern>/dwruser/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>user</role-name>
    </auth-constraint>
</security-constraint>
```

上面的配置片段表明：只有 admin 角色的用户才可以访问/dwradmin/*下的资源，只有该角色才可以调用 dwr-admin.xml 配置文件中定义的远程类；而 user 角色的用户只可以访问/dwruser/*下的资源，也就是可以调用 dwr-user.xml 配置文件中定义的远程类。

13.2 使用 DWR

一旦我们在 Web 应用中成功安装了 DWR，接着就可以编写 Java 处理类了。开发这个 Java 类也非常简单，因为这个 Java 类几乎没有太多的额外要求。

➤➤13.2.1 编写处理类

使用 DWR 开发 Ajax 应用是非常简单的，但最简单的莫过于开发处理类：处理类完全就是一个 POJO（Plain Old Java Object，普通的传统 Java 对象），几乎不需要任何额外要求，只有如下两个要求：

- 远程处理类的方法名不要使用 JavaScript 保留字。
- 远程处理类的方法不要重载。

下面的应用将会编写一个足够复杂的 Java 处理类，它包含了大量方法，这些方法具有一定的代表性。该处理类的代码如下：

程序清单：codes\13\13.2\hellodwr\WEB-INF\src\lee\HelloDwr.java

```
public class HelloDwr
{
    //第一个简单的 hello 方法
    public String hello(String name)
    {
        return name + ", 您好！您已经开始了 DWR 的学习之旅，祝您学得开心...";
    }
    //使用一个 JavaBean 作为参数的方法
    public String sendObj(Person p )
    {
        return p.getName() + ", 您好！您已经学会了使用 JavaBean 参数...";
    }
    //返回 JavaBean 实例的方法
    public Person getBean (String name)
    {
        return new Person(name);
    }
    //返回一个普通的 Java 对象，Cat 对象为其属性提供 setter 和 getter 方法
    public Cat getObject(String name)
    {
        return new Cat("服务器端" + name);
    }
    //返回一个集合对象
    public List<Person> getPersonList()
    {
```

```
        List<Person> result = new ArrayList<Person>();
        result.add(new Person("集合 aaaa"));
        result.add(new Person("集合 bbbb"));
        result.add(new Person("集合 cccc"));
        return result;
    }
    //返回一个数组对象
    public Person[] getPersonArray()
    {
        Person[] result = new Person[3];
        result[0] = new Person("数组 aaaa");
        result[1] = new Person("数组 bbbb");
        result[2] = new Person("数组 cccc");
        return result;
    }
    //返回一个 Map 对象
    public Map<String, Person> getPersonMap()
    {
        //创建一个 Map 对象
        Map<String, Person> result = new HashMap<String, Person>();
        //填充 Map 对象的内容
        result.put("first" , new Person("Map aaaa"));
        result.put("second" , new Person("Map bbb"));
        result.put("third" , new Person("Map cccc"));
        //返回 Map
        return result;
    }
    //远程方法的参数是集合
    public String sendList(List<Person> pl)
    {
        String result = "";
        for (Person p : pl)
        {
            result += p.getName() + "<br />";
        }
        return result;
    }
    //远程方法的参数是不带泛型的集合
    public String sendListNoGeneric(List pl)
    {
        String result = "";
        for (Object p : pl)
        {
            result += ((Person)p).getName() + "<br />";
        }
        return result;
    }
    //远程方法的参数是集合
    public String sendMap(Map<String , Person> pmap)
    {
        String result = "";
        for (String key : pmap.keySet())
        {
            result += "键" + key + " 其值为: " +
                pmap.get(key).getName() + "<br />";
        }
        return result;
    }
}
```

上面的处理类有点繁琐，但这种繁琐是有价值的。因为它代表了处理类的如下几种情况：

- 远程 Java 方法的参数是字符串，返回值也是字符串。
- 远程 Java 方法的参数是 JavaBean 对象，返回值是字符串。
- 远程 Java 方法的参数是字符串，返回值是 JavaBean 实例。
- 远程 Java 方法的参数是字符串，返回值是 Java 对象，该对象的属性没有对应的 seter 和 getter 方法。
- 远程 Java 方法的参数是集合对象。
- 远程 Java 方法的参数是 Map 对象。
- 远程 Java 方法的返回值是集合对象。
- 远程 Java 方法的返回值是数组对象。
- 远程 Java 方法的返回值是 Map 对象。

这个处理类力求覆盖到参数、返回值是各种数据类型的情形。一旦完成了处理类的定义，就可以通过 DWR 配置文件定义处理类和 JavaScript 对象之间的关系。然后 DWR 将负责将 Java 类导出成 JavaScript 对象，暴露给客户端调用。

13.2.2 配置 DWR

DWR 配置文件大致上遵循如下格式：

```
<?xml version="1.0" encoding="GBK"?>
<!-- DWR配置文件的DTD信息，也可换成Schema定义 -->
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
    "http://getahead.ltd.uk/dwr/dwr20.dtd">
<!-- dwr是DWR配置文件的根元素 -->
<dwr>
    <!-- 只有当需要扩展DWR时，才需要init元素 -->
    <init>
        <!-- 下面的creator、converter可出现0次或多次 -->
        <creator id="..." class="..."/>
        <converter id="..." class="..."/>
    </init>
    <!-- allow元素是核心元素，用于定义Java类和JavaScript对象的对应关系 -->
    <allow>
        <!-- 下面的create、filter和convert可出现0次或多次 -->
        <create creator="..." javascript="..."
            scope="application | session | script | request | page">
            <param name=" " value="..."/>
            <include method="..."/>
            <exclude method="..."/>
        </create>
        <filter class="...">
            <param name="..." value="..."/>
        </filter>
        <convert converter="" match="" javascript="">
            <param name="" value=""/>
        </convert>
    </allow>
    <!-- signatures元素列出所有的方法声明 -->
    <signatures>
        列出所有的方法声明
    </signatures>
</dwr>
```

在上面通用的配置模板中，最重要的元素就是<allow.../>元素，如果一个 dwr.xml 文件中没有定义

<allow.../>元素，或者<allow.../>元素为空，则 DWR 将什么都干不了。<allow.../>元素里两个常用的元素是<create.../>和<convert.../>，其中<create.../>用于定义如何将一个 Java 类转换成一个 JavaScript 对象，而<convert.../>定义如何完成 Java 对象和 JavaScript 对象之间的转换。

通常有如下术语：方法参数会被 converted，远程 Java 类会被 create 成一个 JavaScript 对象。所以有一个叫 A 的 bean，它有一个方法叫 A.blah(B)，那么，应该为 A 指定一个 Creator（创建器），并为 B 指定一个 Converter（转换器）。

当使用<create.../>元素来创建一个 JavaScript 对象，首先应该创建 Java 对象，这个创建的 Java 对象可以被放入如下几个范围：script、page、request、session 和 application。相信读者对后四个范围都非常了解（如果对这四个范围还不了解，请先参阅疯狂 Java 体系的《轻量级 Java EE 企业应用实战》），如果指定 scope="script"，这意味着该 Java 对象仅被创建成 JavaScript 对象。

<init.../>元素是可选的，它用于声明新的创建器和新的转换器。大部分情况下，无需开发自己的创建器和转换器。但如果需要定义一个新的创建器和转换器，才需要在<init.../>部分定义。通常，DWR 已经提供了开发过程中需要的创建器和转换器。

<init.../>元素只是告诉 DWR 这些扩展类的存在，并定义如何使用这些扩展类，但它们依然还没有被使用。定义<creator.../>和<converter.../>元素时都应该指定 id 属性，以便后面使用。

因为处理类中方法的参数、返回值有大量非基本数据类型，因此必须告诉 DWR 如何处理这些复合类型的变量。为了处理这种 Java 对象和值和 JavaScript 变量之间的转换，DWR 提供了的大量的转换器，下面依次介绍这些转换器。

每个 Java 对象必须被暴露成一个 JavaScript 对象，才能在客户端 JavaScript 脚本中被调用，这个过程被成为“转换”，负责将一个 Java 对象转换成 JavaScript 对象的角色称为转换器，DWR 提供了大量的转换器，下面也将具体介绍这些转换器。

13.3 使用 DWR 的转换器

如果 Java 方法的参数不是基本数据类型，也不是字符串类型时，DWR 无法正常识别这种自定义类型的数据，必须在配置文件中定义转换器。转换器的作用是完成 Java 实例和 JavaScript 对象之间的相互转换，从而保证 Ajax 交互的正常通信。

➤➤13.3.1 基本转换器

为什么需要使用转换器，当我们声明一个方法时，该方法参数的类型是一个 Java 类型，但如何转换成对应的 JavaScript 类型呢？转换器用于负责定义 Java 类型和 JavaScript 类型之间的对应关系。

对于普通的情形，我们无需显式使用转换器，对于所有基本数据类型，包括 String 和 BigDecimal 等，DWR 已经提供了种简单对象的转换器，无需在 dwr.xml 中<allow>元素中使用<convert>元素定义。

DWR 默认支持的类型有：boolean、byte、short、int、long、float、double、char、java.lang.Boolean、java.lang.Byte、java.lang.Short、java.lang.Integer、java.lang.Long、java.lang.Float、java.lang.Double、java.lang.Character、java.math.BigInteger、java.math.BigDecimal 和 java.lang.String。

对于日期类型的数据，DWR 则提供了 Date 转换器，Date 转换器负责在 Javascript 的 Date 类型与 Java 中的 Date 类型(java.util.Date，java.sql.Date，java.sql.Times 或 java.sql.Timestamp)之间进行转换，DWR 默认支持 Date 转换器。

如果有一个 JavasSript 的字符串 (例如'2006-12-12') ，需要转换成 Java 的 Date 类型。默认提供了两个方法：在 Javascript 中用 Date.parse()把它解析成 JavaScript 的 Date 类型，然后由 DWR 的 DateConverter 来负责转换成 Java 的 Date 对象。或者直接将该字符串传递到服务器端，然后在服务器端使用 SimpleDateFormat 将该字符串转换成 Date 对象。

反过来，如果有个 Java 的 Date 对象，希望在 HTML 使用这个值。可以先用 SimpleDateFormat 把它转换成字符串，然后传递给客户端的 JavaScript 使用。也可以直接作为 Date 传给客户端，由 Date 转换器完成 Java Date 对象向 JavaScript 对象的转换，然后在客户端使用 JavaScript 完成格式化。第一种方式简单一些，并且无需使用 Date 转换器，但这种做会让在日期在览器上的显示逻辑受到限制——只是一个字符串，无法提供多样化的显示。第二个方法则更加灵活，客户端的 JavaScript 代码获得是 Date 对象，因此将可以提供更灵活的显示格式。

➤➤13.3.2 对象转换器

DWR 默认关闭了 Bean 和 Obect 转换器。Bean 转换器用于完成 JavaBean 对象和 JavaScript 对象之间的转换。Bean 转换器默认是关闭的，因此必须在 dwr.xml 文件中显式打开该转换器，才可以在 Java 方法中使用对应的 JavaBean 实例作为参数或返回值。

Object 转换器与 Bean 转换器非常类似，它们的作用完全相同，都是用于完成 Java 对象和 JavaScript 对象对象之间的转换。但 Object 转换器的功能更加强大，Object 转换器通过 Java 的反射来访问 Java 对象的属性，即 Object 转换器用于转换普通 Java 对象（该 Java 对象并未为每个属性提供 setter 和 getter 方法）。

提 示：

Bean 转换器和 Object 转换器的实质是相同的，区别只是两个转换访问 Java 对象里属性的方式不同，Bean 转换器使用 setter 和 getter 方法访问 Java 对象的属性，而 Object 转换器则使用反射来访问 Java 对象的属性。

为某个类单独打开一个 Bean 转换器，可以采用如下代码：

```
<!-- 对 lee.Person 类使用 Bean 转换器 -->
<convert converter="bean" match="lee.Person"/>
```

上面转换器表明 DWR 将使用 setter 和 getter 方法访问 lee.Person 对象的各属性。当使用 Bean 转换器来完成 Java 对象和 JavaScript 对象之间的转换时，该 Java 类应该符合 JavaBean 规范，要求每个属性必须有对应的 getter 和 setter 方法，其中 setter 有一个参数，并且该参数类型是 getter 方法的返回值类型，setter 方法返回值声明为 void。getter 方法没有任何参数。

如果同时需要转换某个包下的所有类，可用采用如下格式定义：

```
<convert converter="bean" match="lee.*"/>
```

上面的表明 lee 包下所有的类都是可以转换的。而下面配置将指定所有类都使用 Bean 转换器进行转换：

```
<convert converter="bean" match="*"/>
```

如果 Java 对象对应的类里属性没有对应的 setter 和 getter 方法，则应该考虑使用 Object 转换器，而不是 Bean 转换器。下面是本示例应用中的 Person 类代码：

程序清单：codes\13\13.2\hellodwr\WEB-INF\src\lee\Person.java

```
public class Person
{
    //私有属性
    private String name;
    public Person() {}
    //构造器
    public Person(String name)
    {
        this.name = name;
    }
    //name 属性的 setter 方法
    public void setName(String name)
```

```
    {
        this.name = name;
    }
    //name 属性的 getter 方法
    public String getName()
    {
        return name;
    }
}
```

如果远程方法的返回值是 JavaBean 实例，客户端 JavaScript 调用该方法后也返回一个 JavaScript 对象。默认情况下，该对象包含了 Java 对象的所有属性。

对于一个返回的 lee.Person 对象的方法，JavaScript 代码调用该方法后，将返回一个如下格式的 JavaScript 对象：

```
{name:'nameValue'}
```

如果某个方法包含了 lee.Person 类型的参数，则可以在 JavaScript 代码中通过如下代码调用该方法：

```
//调用远程方法
hello.sendObj({name:'nameValue'} , cb);
```

对照前面处理类的 public String sendObj(Person p)，第一个参数是 lee.Person 实例，这表明可以通过 JSON 格式创建一个 JavaScript 对象，然后可使用该对象作为调用参数。

在某些情况下，假如类 A 有 3 个属性：x、y 和 z，但希望 DWR 将 A 实例转换成 JavaScript 对象后只包含 x 和 y 属性，屏蔽 z 属性。可以利用 Bean 转换器的限制转换的功能，限制转换有两种方式：

- 黑名单方式：被黑名单列出的属性将不会被转换到 JavaScript 对象中。
- 白名单方法：没有被白名单列出的属性将不会转换到 JavaScript 对象中。

黑名单方式的语法格式如下：

```
<convert converter="bean" match="lee.Person ">
    <param name="exclude" value="property1, property2..."/>
</convert>
```

上面的表明 DWR 转换 lee.Person 实例时，将不会转换其中的 property1 和 property2 两个属性（当然，前提是 lee.Person 有这两个属性）。如果有更多的属性不希望 DWR 转换，则依次排列在后面，属性之间以英文逗号（,）隔开。

通常推荐使用 JavaBean 作为远程 Java 方法的参数和返回值。如果远程 Java 方法的参数或返回值不是 JavaBean 实例，而是普通 Object，则应该使用 Object 转换器。例如处理类中的 getObject 方法，其返回值是 Cat，则应该使用 Object 转换器。Cat 类的代码如下：

程序清单：codes\13\13.2\hellodwr\WEB-INF\src\lee\Cat.java

```
public class Cat
{
    //Cat 类的私有属性
    private String name;
    //构造器
    public Cat(String name)
    {
        this.name = name;
    }
}
```

对于 Cat 类的转换器在必须使用 Object 转换器，转换器的定义如下：

```
<!-- 对 lee.Cat 使用 Object 转换器 -->
<convert converter="object" match="lee.Cat">
    <!-- 指定 force="true"强制使用反射访问私有属性 -->
```

```
    <param name="force" value="true"/>
</convert>
```

与使用 Bean 转换器定义类似的是，match 属性一样支持通配符，因此也可以一次匹配某个包下的所有类，或应用中的全部类。

注意：

应该尽量变量使用 Object 转换器，因为 Object 转转器依赖于反射来访问 Java 对象的属性，因此将会引起部分性能下降。

13.3.3 数组转换器

默认情况下，数组转换器已经是打开的，不管数组元素是基本数据类型，还是字符串，或者是其他引用数据类型。DWR 都可支持转换。因此，不管远程 Java 方法中的参数是数组，还是 Java 方法返回值是数组，都不需在 dwr.xml 文件中增加额外的配置。

对于 Java 方法包含数组或者形参是数组的情形，需要注意 JavaScript 代码如何处理数组类型的返回值和请求参数。

如果远程 Java 方法的返回值是数组。例如返回如下数组：

```
//创建一个长度为 3 的数组
Person[] result = new Person[3];
//依次初始化三个数组元素
result[0] = new Person("数组 aaaa");
result[1] = new Person("数组 bbbb");
result[2] = new Person("数组 cccc");
return result;
```

如果远程 Java 方法返回该数组，JavaScript 客户端也将获得一个数组，数组长度也是 3，数组元素也是对象，是一个 JSON 格式的对象。因此，可以在客户端通过如下代码来访问返回值。

```
//下面的 data 是远程 Java 方法的返回值，
//data 是个数组，遍历数组。
for (var i = 0 ; i < data.length ; i ++)
{
    //依次访问数组元素，数组元素是 JSON 格式的对象，访问其 name 属性
    result += data[i].name + "<br />";
}
```

如果有一个这样的方法声明：

```
methodA(Person[] pArr)
```

上面方法声明中，方法参数是 Person 数组，该数组的每个元素是 Person 对象。则客户端也需要构造一个 JavaScript 数组作为参数。JavaScript 代码通过如下方式调用该方法：

```
//定义参数数组，数组每个元素都是 JSON 对象，该对象可转换成 lee.Person 实例
var args = [
    {name:"客户端 aaa"},
    {name:"客户端 bbb"},
    {name:"客户端 ccc"}
];
//调用远程 Java 方法，其中 cb 是回调函数
methodA(args,cb);
```

13.3.4 集合类型转换器

DWR 提供了两个集合转换器，两个转换器分别用于转换 Map 对象和 Collection 对象。DWR 默认

已经打开这两个集合转换器，因此无需在 dwr.xml 文件中配置。

如果需要显式配置，可采用如下语法格式：

```
<convert converter="collection" match="java.util.Collection"/>
<convert converter="map" match="java.util.Map"/>
```

这种转换器，支持 Collection 放 Map，也支持 Map 里放 Collection。就是说，转换器可以可以递归转换集合元素的内容。map 和 collection 转换器也有如下两点不足之处：

- 如果集合不使用泛型限制集合元素的类型，则 DWR 无法确定集合元素的类型。因此，这两个转换器都不能把集合元素转换成有意义的 JavaScript 对象。
- 不能明确指定集合的类型，只能使用基于接口的类型转换。

注意：

如果集合没有使用泛型来限制集合元素的类型，DWR 也提供了一定的补救措施，DWR 允许在 dwr.xml 中用<signatures.../>来声明集合元素的数据类型，从而可以让 DWR 能正确识别集合元素的类型。当然，使用泛型来限制集合元素类型将会更加简洁。

如果一个远程 Java 方法返回值是 Collection，客户端 JavaScript 调用该方法后将返回一个数组，这种情形，与调用远程 Java 方法返回数组的情形几乎完全一样。因此，远程 JavaScript 代码可通过访问数组的方式访问集合返回值。

如果一个远程 Java 方法包含 Collection 类型的形参，则客户端 JavaScript 代码只需提供一个数组参数即可调用该方法，数组元素对应集合元素。

如果远程 Java 方法返回值是 Map，客户端 JavaScript 代码调用该方法后将返回一个 JavaScript 对象，该对象的属性名是 Map 的 key，属性值是对应的 value。

对于如下的 Java 方法：

```
public Map<String, Person> getPersonMap()
{
    //创建一个 Map 对象
    Map<String, Person> result = new HashMap<String, Person>();
    //填充 Map 对象的内容
    result.put("first" , new Person("Map aaaa"));
    result.put("second" , new Person("Map bbb"));
    result.put("third" , new Person("Map cccc"));
    //返回 Map
    return result;
}
```

如果在客户端调用该方法，将返回一个 JSON 格式对象，该对象的具体值如下：

```
{
    first:{name:"Map aaaa"},
    second:{name:"Map bbbb"},
    third:{name:"Map cccc"},
}
```

如果某个远程 Java 方法包含 Map 类型的形参，则 JavaScript 客户端需要创建一个 JavaScript 对象作为实际参数传入。

13.4 方法声明定义

DWR 使用反射来确定 Java 实例和 JavaScript 对象之间的转换。有时候，参数类型信息并不十分明确，或者参数是一个集合对象，而且没有使用泛型来限制集合元素的类型。这就需要在 dwr.xml 文

件的<signatures.../>元素中明确指定这些类型。

方法声明放在<signatures.../>元素中定义，该元素的值是一个普通字符串。因为该元素的值通常可能包含一些特殊字符，为了避免这些特殊字符造成混淆，通常使用<![CDATA[...]]>将所有的字符串值包含起来。

假设有如下方法：

```
//远程方法的参数是不带泛型的集合
public void sendListNoGeneric(List pl)
{
    String result = "";
    for (Object p : pl)
    {
        result += ((Pereson)p).getName() + "<br />";
    }
    return result;
}
```

该方法的形参类型是一个普通 List，但 List 里的集合元素类型是不确定的，DWR 无法知道该 List 集合的元素类型。这就可以借助于 DWR 的<signatures.../>元素来定义了，signatures.../>元素里定义方法的语法格式非常类似于 JDK1.5 的泛型。如下配置片段：

```
<signatures>
 <![CDATA[
    import java.util.List;
    import lee.HelloDwr;
    import lee.Person;
    HelloDwr.sendListNoGeneric (List<Person>);
 ]]>
</signatures>
```

上面程序中的粗体字代码的 import 语句用于导入方法所在类型、形参类型等，配置中明确定义了一个 sendList()方法，该配置告诉 DWR ，sendList()方法的形参是一个 List，List 的集合元素是 Person 对象。

看到这种语法，可能有读者会猜测这是 JDK1.5 语法。其实不然，DWR 中有一个解析器专门处理这种方法声明，因此，即使使用 JDK1.4，上面这段配置依然可以正常工作。

使用方法声明时，应该使用 import 行来负责导入方法声明中所使用的类，import 的语法格式与 Java 代码中的 import 语法格式完全相同。类似于 Java 语言，DWR 也会默认导入 java.lang.*下所有类，因此无需为 java.lang.String 类导包。

<signatures.../>元素主要用于确定方法形参、返回值的类型，如果 DWR 可以通过反射来确定确定参数、返回值的类型，则无需配置<signatures.../>元素。

13.5 使用 DWR 的创建器

DWR 复杂把 Java 对象转换成客户端可以调用的 JavaScript 对象，为了将 Java 对象转换成 JavaScript 对象，DWR 提供了创建器，创建器负责将 Java 对象创建成 JavaScript 对象。

➤➤13.5.1 创建器的配置

DWR 使用 dwr.xml 文件的<create.../>元素来配置创建，<create.../>元素的大致格式如下：

```
<allow>
    <create creator="..." javascript="..." scope="...">
        <param name="..." value="..."/>
        <auth method="..." role="..."/>
        <exclude method="..."/>
        <include method="..."/>
```

```
    </create>
</allow>
```

create 元素用于创建一个 JavaScript 对象，create 元素中的很多属性和子元素都是可选的，但通常需要指定两个属性。javascript 属性用于指定所创建的 JavaScript 对象的名字，而 creator 则指定创建实例时所使用的创建器。

- DWR 默认提供如下几个创建器：
- new：该创建器使用 new 关键字创建 Java 实例。
- none：该创建器不创建任何实例。关于原因请参看后面介绍。
- scripted：使用脚本语言，如 BeanShell 或 Groovy 通过 BSF 创建实例。
- spring：直接使用 Spring 容器中的 Bean，用于 DWR 和 Spring 的整合。
- jsf：直接使用 JSF 的 Bean，用于 JSF 和 DWR 的整合。
- struts：直接使用 Struts 的 FormBean，用于 Struts 和 DWR 的整合。
- pageflow：直接访问 Weblogic 或 Beehive 的 PageFlow。
- ejb3：这是一个试验性的创建器，通过该创建器，DWR 可以直接使用 EJB3.0 的 Session Bean。到笔者成书之时，这个创建器依然没有产品版发布。建议谨慎使用。

最基本、最常用的创建器是 new。除此之外，用户也可以实现自己的创建起，如果用户需要实现自己的创建器，则必须在 init 部分注册该创建器。

对于 create 元素，有如下属性和子元素是通用的：

- javascript 属性：当远程 Java 对象被创建后，DWR 负责将该对象暴露成客户端的 JavaScript 对象，该属性指定这个 JavaScript 对象的名字。
- scope 属性：该属性非常类似于 Servlet 规范中的 scope。用于指定指定该 Java 实例的生命范围。只能选择 application、session、request 和 page 四个其中之一。该属性是可选的，默认是"page"。如果要选择 session 值，则需要浏览器支持 cookie，当前的 DWR 还不支持 URL 重写。
- param 子元素：被用来指定 Creaotr 的其他相关参数，不同的创建器需要指定的 param 是不一样的。例如，new 创建器则需要指定 class 参数，该参数确定创建 Java 实例所使用的 Java 类。不同创建器将在后面深入介绍。
- include 子元素：如果 Java 实例中包含 5 个方法，但只想有 3 个方法被暴露，则可以使用 include 元素指定这 3 个方法。一旦使用了 include 元素，Java 对象的其他方法默认将不被暴露。
- exclude 子元素：作用类似于 include 元素，但 exclude 元素用于指定不被暴露的方法。使用 exclude 元素，Java 对象的其他方法默认将被暴露出来。
- auth 子元素：该元素指定一个 Java EE 角色，一旦指定了<auth.../>元素，该 Java 对象的方法将处于被保护状态，只有具有<auth.../>指定的 Java EE 角色的用户才可以访问该对象的方法。

对于如下 lee.HelloDwr 类的创建配置，如果不希望将 abc()方法被暴露给 JavaScript 客户端，可以采用如下配置片段：

```
<create creator="new" javascript="hello">
    <param name="class" value="lee.HelloDwr"/>
    <exclude method="abc"/>
</create>
```

对于上面的配置片段，使用<exclude.../>子元素是黑名单配置方式，这表明 lee.HelloDwr 实例的方法默认将被暴露给 JavaScript 客户端，只有<exclude.../>元素指定的方法不会被暴露。

如果希望只暴露 lee.HelloDwr 实例的 abc()方法，其他方法都将被隐藏，则可采用如下配置片段：

```
<create creator="new" javascript="hello">
    <param name="class" value="lee.HelloDwr"/>
    <include method="abc"/>
</create>
```

对于上面配置片段，使用<include.../>子元素是白名单配置方式，表明 lee.HelloDwr 实例的方法默认将被隐藏，只有<include.../>元素指定的方法会被暴露给 JavaScript 客户端。

注意：

<exclude.../>和<include.../>实际是上黑名单和白名单的关系。其中<exclude.../>元素用于指定不希望被暴露的方法名单，而<include.../>元素指定希望被暴露的方法名单。相比之下，使用白名单的安全性更高一些。

<auth.../>子元素则用于指定 Java 对象的方法和 Java EE 角色的对应关系，只有该角色的用户才可访问该方法。这个角色通常需要 JAAS 支持。<auth.../>元素需要指定如下两个属性：

- method：该属性指定方法名，该方法名支持通配符。
- role：指定角色名。

如果希望只有 admin 角色才可以访问 lee.HelloDwr 下的 admin 方法，而可采用如下配置片段：

```
<create creator="new" javascript="hello">
    <param name="class" value="lee.HelloDwr"/>
    <auth method="admin" role="admin"/>
</create>
```

对于上面的配置片段，只有 admin 方法需要额外的 Java EE 角色认证。其他方法则一样无需认证。下面依次介绍常用的创建器的配置。

13.5.2 使用 new 创建器

new 创建器是最常用，也最简单的创建器，它使用 new 关键字创建一个 Java 实例，然后将该 Java 实例暴露给客户端 JavaScript 代码。

DWR 默认已启用了 new 创建器。如果开发中需要使用 new 创建器，无需在 init 部分指定 new 创建器。因为，DWR 默认已经添加了如下代码，

```
<creator id="new" class="uk.ltd.getahead.dwr.create.NewCreator"/>
```

当然，上面的配置片段无需出现在用户自定义的 dwr.xml 文件中，因为它被添加在 DWR 的内部 dwr.xml 文件中。

用 new 创建器有如下好处：

- 安全：new 创建器所创建对象的生存时间极短，因此多次调用中的状态不一致的机会很少。
- 内存消耗低：如果应用的用户量非常大，使用 new 创建器可以减少 JVM 的内存溢出的机会。

使用 new 创建器的配置语法非常简单，new 创建器需要指定一个 class 属性，该属性指定创建该 Java 对象的类（该类应该提供默认的构造器）。

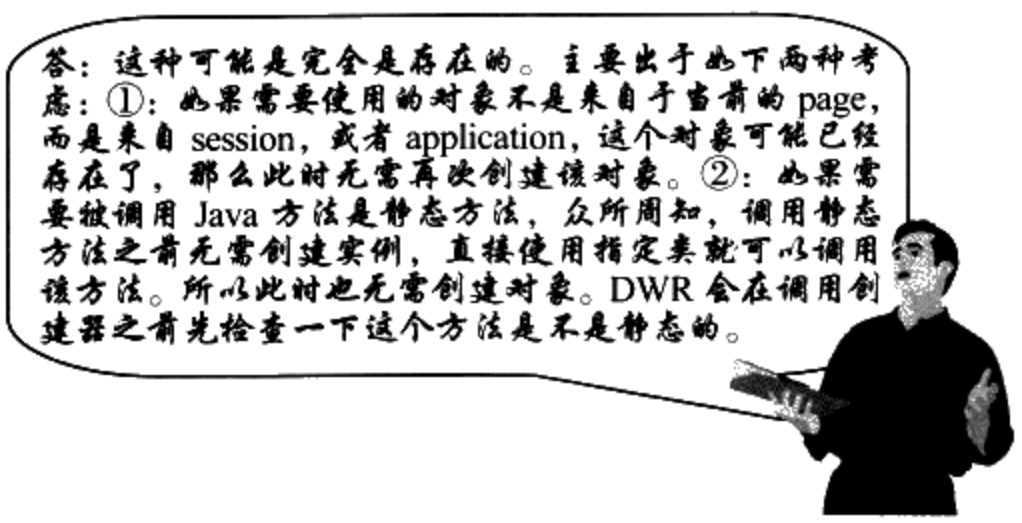
下面是使用 new 创建器的配置片段：

```
<!-- 使用 new 关键资创建一个 Java 实例
    指定创建的 JavaScript 对象名为 hello-->
<create creator="new" javascript="hello">
    <!-- 使用 class 属性指定创建该 Java 实例的实现类 -->
    <param name="class" value="lee.HelloDwr"/>
    ...
</create>
```

上面的配置片段的含义是：DWR 将使用无参数的构造器创建 lee.HelloDwr 的实例，并将该实例暴露给客户端 JavaScript 代码。lee.HelloDwr 实例被转换名为 hello 的 JavaScript 对象。这样，我们可以在 JavaScript 代码中通过 hello 对象调用 lee.HelloDwr 中的方法。

➤➤13.5.3 使用 none 创建器

none 创建器不创建任何对象。



当我们使用 none 创建器时，仍然需要使用<param.../>子元素，<param.../>子元素的 class 属性和 value 属性都不可少，通过这两个属性来指定 DWR 操作对象的类型。

➤➤13.5.4 使用 script 创建器

script 创建器用于通过动态语言，例如 BSF 等创建 Java 实例，然后将这些实例暴露成 JavsScript 对象。DWR 默认已经启用了 script 创建器，也就是说，在 DWR 默认的 dwr.xml 配置文件中已经导入了如下代码：

```
<creator id="script" class="uk.ltd.getahead.dwr.create.ScriptedCreator"/>
```

如下代码片段示范了使用 BSF 来创建 Bean，并将该 Bean 暴露成 JavaScript 对象：

```
<!-- 使用 script 创建器创建 Java 实例，暴露成 EmailValidator 对象 -->
<create creator="script" javascript="EmailValidator">
    <!-- 指定动态语言的类型 -->
    <param name="language" value="beanshell"/>
    <!-- 指定创建 Java 对象的脚本 -->
    <param name="script">
        import org.apache.commons.validator.EmailValidator;
        return EmailValidator.getInstance();
    </param>
</create>
```

script 创造器需要如下参数：

- language：指定创建 Java 对象的脚本语言，例如 beanshell，这是一个必需的属性。
- script：指定创建 Java 对象的脚本。该属性是一段可执行的脚本，随着 language 属性的不同，这段脚本的语法也是可变的。该属性和 scriptPath 必须出现其中之一。
- scriptPath：指定创建 Java 对象脚本源文件的全限定路径，该路径指定一个脚本文件，该文件里的脚本可创建一个 Java 对象，该属性与 script 必须出现其中之一。
- reloadable：当脚本改变时，是否自动重新加载，该属性是可选的，默认 true。
- class：该属性指定创建的对象的的类，该属性是可选的。

关于其他创建器，例如 spring 和 jsf 等，将在 DWR 整合部分介绍。

对于本应用而言，起 dwr.xml 配置文件的代码如下：

程序清单：codes\13\13.2\hellodwr\WEB-INF\dwr.xml

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 DWR 配置文件的 DTD 等信息 -->
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr20.dtd">
```

```
<!-- DWR 配置文件的根元素是 dwr -->
<dwr>
    <allow>
        <!-- 使用 new 关键资创建一个 Java 实例
            指定创建的 JavaScript 对象名为 hello-->
        <create creator="new" javascript="hello">
            <!-- 使用 class 属性指定创建该 Java 实例的实现类 -->
            <param name="class" value="lee.HelloDwr"/>
        </create>
        <!-- 对 lee.Person 类使用 Bean 转换器 -->
        <convert converter="bean" match="lee.Person"/>
        <!-- 对 lee.Cat 使用 Object 转换器 -->
        <convert converter="object" match="lee.Cat">
            <!-- 指定 force="true"强制使用反射访问私有属性 -->
            <param name="force" value="true"/>
        </convert>
    </allow>
    <signatures>
      <![CDATA[
        mport java.util.List;
        import lee.HelloDwr;
        import lee.Person;
        HelloDwr.sendListNoGeneric(List<Person>);
      ]]>
    </signatures>
</dwr>
```

13.6 调用服务器端的方法

正如前面提到的：从表面上来看，客户端 JavaScript 代码可以调用远程 Java 方法，但这只是一种假相。实际情况是：DWR 负责创建 Java 对象，并动态生成系列 JavaScript 脚本，并在 JavaScript 脚本中创建与 Java 对等的 JavaScript 对象，这个 JavaScript 对象里包含了对应 Java 对象的全部方法。

为了在客户端页面调用远程 Java 对象的方法，则必须在客户端页面中导入 JavaScript 脚本，DWR 的 JavaScript 脚本文件是动态生成的，直接在 Web 应用下找不到 DWR 动态生成的 JavaScript 脚本。

➤➤13.6.1 调用服务器端方法的通用配置

通常，DWR 会生成如下两个 JavaScript 文件：

- engine.js：DWR 的核心 JavaScript 文件，这个文件是不可缺少的。
- util.js：DWR 的工具 JavaScript 文件，在该文件内提供了一些工具方法。通过这些工具方法，可以简化 DOM 操作。

问题是：这两个文件是动态生成的，那么它们到底在什么地方呢？实际上，这两个文件生成的位置是变化的，它们的位置会随着 web.xml 文件中配置 uk.ltd.getahead.dwr.DWRServlet 的 URL 不同而不同。通常它们的实际位置是

```
uk.ltd.getahead.dwr.DWRServlet 的 URL/engine.js
uk.ltd.getahead.dwr.DWRServlet 的 URL/util.js
```

对于如下的配置片段：

```
<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <!-- 指定核心 Servlet 映射的 URL -->
    <url-pattern>/leedwr/*</url-pattern>
</servlet-mapping>
```

则 DWR 的两个 JavaScript 文件的位置在：

```
http://domainName:port/hellodwr/leedwr/engine.js
http://domainName:port/hellodwr/leedwr/util.js
```

除此之外，在 dwr.xml 文件中每使用一次 create 元素，将对应于创建一个 Java 对象，并将该对象暴露成 JavaScript 实例。DWR 也将动态生成一个 JavaScript 文件。这个 JavaScript 文件的位置在：

uk.ltd.getahead.dwr.DWRServlet 的 URL/interface/JavaScript 对象名.js

其中 interface 是固定的，而 JavaScript 对象名就是在 dwr.xml 配置文件中为 JavaScript 对象指定的对象名。对于采用上面的 web.xml 文件配置，以及采用如下 dwr.xml 配置片段：

```
<!-- 创建的 JavaScript 对象名为 hello -->
<create creator="new" javascript="hello">
    <param name="class" value="lee.HelloDwr"/>
</create>
```

因为上面创建的 JavaScript 对象名为 hello，因此 DWR 为其动态生成的 JavaScript 文件位置为：

```
http://domainName:port/hellodwr/leedwr/interface/hello.js
```

注意：

如果需要使用 DWR 生成的 JavaScript 脚本，必须先导入这些 JavaScript 脚本。其中 engine.js 是必需的，还有 JavaScript 对象所在的 JavaScript 文件也是必需的。而 util.js 脚本文件是可选的，当开发者需要使用 DWR 提供的工具函数时才需要导入该脚本。

一旦导入了 DWR 动态生成的 JavaScript 脚本，我们就可以使用异步方式来调用这些方法。所谓异步，就是需要在调用远程方法时传入回调函数。

当调用远程 Java 方法时候，总需要为该方法增加一个参数，传入的最后一个参数就是回调函数。回调函数当服务器响应完成时被触发，用于将服务器响应显示在当前页面中。

注意：

指定回调函数有两种做法：简单回调和使用 JSON 格式。如果使用简单回调，则调用远程方法的最后一个参数是函数引用——该函数就是回调函数；除此之外，还可以指定最后一个函数一个 JSON 格式对象，这种方式可指定更多的回调函数。

➤➤13.6.2 使用简单回调

使用简单回调，只要调用远程 Java 方法时额外增加一个参数即可，最后一个参数代表回调函数，这样能以异步方式调用远程 Java 方法，当远程 Java 方法响应结束后，最后一个参数指定的回调函数将会被触发。

假设有如下一个类，该类里仅包含一个简单的 hello()方法，例如如下 Hello 类：

```
//简单的处理类
public class Hello
{
    //仅包含一个简单的 hello 方法
    public String hello(String name)
    {
        ...
    }
}
```

一旦使用 DWR 将其暴露成 JavaScript 对象（假设被暴露成一个 hello 的 JavaScript 对象），则可通过如下代码来使用该 hello 对象。

```
<!-- 导入 DWR 的核心 JavaScript 代码，
    其中 leedwr 是 DWR 核心 Servlet 映射的 URL -->
```

```
<script type="text/javascript"  src="leedwr/engine.js">
</script>
<!-- 包含导出对象的 JavaScript 代码，其中 leedwr 是 DWR 核心 Servlet 映射的 URL -->
<script type="text/javascript"  src="leedwr/interface/hello.js">
</script>
//定义回调函数
function cb(data)
{
    alert(data);
}
//异步调用远程 Java 方法
hello.hello('yeeku', cb)
```

其中'yeeku'是传给远程 Java 方法 hello(String name)方法的 name 参数，而 çb 则是回调函数。当服务器响应完成时，回调函数将会自动启动。回调函数将总是一个遵循如下格式的函数：

```
function cbName(data)
{
    //data 就是服务器响应的数据
    ...
}
```

回调函数有一个参数：data，该参数就是服务器的响应，该参数既可以是字符串，也可以是对象，也可是 DOM 对象，也可以是数组，这取决于服务器端 Java 方法的返回值——DWR 转换器将会负责将该返回值转换成 JavaScript 对象，如字符串、Java 对象、数组等。

注意：

通常推荐将回调函数作为最后一个参数。通常也可以将回调函数作为第一个参数，但这可能导致错误，因此不推荐这样调用。

当然，回调函数也可写成匿名函数的形式，通过如下形式调用远程 Java 方法也完全没有问题：

```
//异步调用远程 Java 方法，回调函数是个匿名函数
hello.hello('yeeku', function(data)
{
    alert(data);
})
```

简单回调是最常见、最简单的调用方式，本示例应用的调用方法全部采用这种调用方式。当然，这个应用涉及的情形比较多，既有发送集合参数，也有发送 Map 参数，也有发送 JavaBean 参数；也包含了返回集合，返回 Map，返回 JavaBean 等多种情形。之所以把这个示例弄得这么复杂，笔者希望通过这个示例让读者理解各种情形。

下面是客户端调用远程 Java 方法的 JavaScript 代码。

程序清单：codes\13\13.2\hellodwr\hellodwr.js

```
//-------------发送简单字符串参数，返回普通字符串--------------
function sendMessage()
{
    //获取页面中 name 元素的值
    var name = document.getElementById("name").value;
    //调用远程方法，cb 是回调函数
    hello.hello(name , cb)
}
function cb(data)
{
    document.getElementById("show").innerHTML = data;
}
```

```
//------------发送一个 JavaBean 对象作为参数，返回普通字符串------------
function sendObject()
{
    var nameValue = document.getElementById("name").value;
    //调用远程方法，使用 JavaScript 对象作为参数
    hello.sendObj({name:nameValue} , cb);
}
//-----------------调用返回 JavaBean 方法-----------------
function getBean()
{
    var name = document.getElementById("name").value;
    //调用远程方法，beanCb 是回调函数
    hello.getBean(name , beanCb)
}
function beanCb(data)
{
    //服务器方法返回 JavaBean 对象，客户端的 data 是 JavaScript 对象
    document.getElementById("show").innerHTML =
        data.name + "，您好，您已经学会了使用 JavaBean 返回值";
}
//----------------调用返回 getObject 方法---------------
function getObject()
{
    var name = document.getElementById("name").value;
    //调用远程方法，objCb 是回调函数
    hello.getObject(name , objCb)
}
function objCb(data)
{
    //服务器方法返回非 JavaBean 式的对象，客户端的 data 是 JavaScript 对象
    document.getElementById("show").innerHTML =
        data.name + "，是从服务器返回的猫的名字";
}
//---------------调用返回集合的方法--------------
function getBeanList()
{
    //调用远程方法，listCb 返回回调函数
    hello.getPersonList(listCb);
}
//远程 Java 方法返回 List 对象，集合元素是 JavaBean 式的对象
//此处的 data 是 JavaScript 对象数组
function listCb(data)
{
    var result='';
    //遍历每个数组元素
    for (var i = 0 ; i < data.length ; i ++)
    {
        result += data[i].name + "<br />";
    }
    document.getElementById("show").innerHTML = result;
}
//---------------调用返回数组的方法--------------
function getBeanArray()
{
    hello.getPersonArray(arrayCb);
}
function arrayCb(data)
{
    var result = "";
    //下面的 data 是远程 Java 方法的返回值，
    //data 是个数组，遍历数组。
```

```
    for (var i = 0 ; i < data.length ; i ++)
    {
        //依次访问数组元素，数组元素是 JSON 格式的对象，访问其 name 属性
        result += data[i].name + "<br />";
    }
    document.getElementById("show").innerHTML = result;
}
//---------------调用返回 Map 对象的方法-------------
function getBeanMap()
{
    hello.getPersonMap(mapCb);
}
//远程 Java 方法返回 Map 对象，集合元素是 JavaBean 式的对象
//此处的 data 是 JavaScript 对象，且每个属性值都是 JavaScript 对象
function mapCb(data)
{
    var result='';
    for (var key in data)
    {
        result += "键为" + key + ",其值为:" + data[key].name + "<br>";
    }
    document.getElementById("show").innerHTML = result;
}

//---------------调用发送集合的方法-------------------
function sendBeanList()
{
    //创建 JavaScript 数组
    var args = [
        {name:"客户端 aaa"},
        {name:"客户端 bbb"},
        {name:"客户端 ccc"}
    ];
    //Java 方法需要 List 参数，以 JavaScript 数组作为参数调用远程方法
    hello.sendList(args , sendListCb);
}
function sendListCb(data)
{
    document.getElementById("show").innerHTML = data;
}
//---------------调用发送无泛型限制的集合-------------------
function sendBeanListNoGeneric()
{
    //创建 JavaScript 数组
    var args = [
        {name:"客户端 aaa"},
        {name:"客户端 bbb"},
        {name:"客户端 ccc"}
    ];
    //Java 方法需要 List 参数，以 JavaScript 数组作为参数调用远程方法
    hello.sendListNoGeneric(args , sendListCb);
}
//---------------调用发送 Map 的方法-----------------------
function sendBeanMap()
{
    //创建 JavaScript 对象
    var args = {
    first:{name:"客户端 aaa"},
```

```
        second:{name:"客户端 bbb"},
        third:{name:"客户端 ccc"}
        };
        //Java 方法需要 Map 参数，以 JavaScript 对象作为参数调用远程方法
        hello.sendMap(args , sendMapCb);
}
function sendMapCb(data)
{
    document.getElementById("show").innerHTML = data;
}
```

虽然上面的 JavaScript 代码比较繁琐，涉及的情况比较多，但客户端的 HTML 文档非常简单，它有一个 name 文本框，用于收集用户输入的用户名。然后提供了 10 个按钮，分别用于激发 10 种请求。HTML 页面的代码如下：

程序清单：codes\13\13.2\hellodwr\index.html

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> DWR 入门 </title>
    <meta name="author" content="Yeeku.H.Lee" />
    <meta name="website" content="http://www.crazyjava.org" />
    <meta http-equiv="Content-Type" content="text/html; charset=GBK" />
    <!-- 导入 DWR 为 hello 对象动态生成的 JavaScript 代码 -->
    <script type='text/javascript' src='leedwr/interface/hello.js'>
    </script>
    <!-- 导入 DWR 引擎的核心 JavaScript 代码库 -->
    <script type='text/javascript' src='leedwr/engine.js'></script>
    <!-- 导入开发者为本应用编写的 JavaScript 代码库 -->
    <script type='text/javascript' src='hellodwr.js'></script>
</head>
<body>
<h3>DWR 入门</h3>
请输入您的名字<input id="name" name="name" type="text"/><br>
<input type="button" value="发送简单请求" onclick="sendMessage();"/>
<input type="button" value="发送对象参数" onclick="sendObject();"/>
<input type="button" value="返回 JavaBean" onclick="getBean();"/><br />
<input type="button" value="返回 Object" onclick="getObject();"/>
<input type="button" value="返回 Bean 集合" onclick="getBeanList();"/>
<input type="button" value="返回 Bean 数组" onclick="getBeanArray();"/><br />
<input type="button" value="返回 Bean Map" onclick="getBeanMap();"/>
<input type="button" value="发送 Bean 集合" onclick="sendBeanList();"/>
<input type="button" value="发送不带泛型限制的 Bean 集合"
    onclick="sendBeanListNoGeneric();"/><br />
<input type="button" value="发送 Bean Map" onclick="sendBeanMap();"/>
<hr />
下面是服务器的回应:<br />
<div id= "show"></div>
</body>
</html>
```

上面代码中粗体字代码用于导入 JavaScript 代码文件，其中导入的第一个文件是 DWR 为 hello 对象动态生成的 JavaScript 代码库，第二个文件是 DWR 的核心 JavaScript 代码库。一旦导入了这两个 JavaScript 文件，就可以在我们的 JavaScript 文件中通过 hello 对象来调用远程 Java 方法，剩下的事情就简单了。

经过上面介绍和示例程序，我们可以看出 Java 类型和 JavaScript 类型之间存在如图 13.4 所示的转换关系。

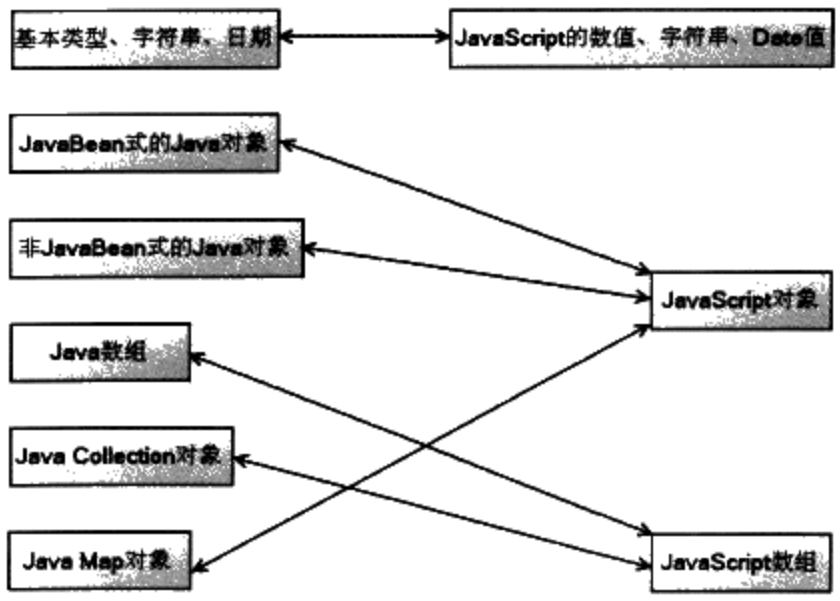


图 13.4　DWR 中 Java 类型和 JavaScript 类型的对应关系

➤➤13.6.3　使用 JSON 格式的回调

上面介绍的回调方式简单易用，但对于复杂的选项，则无法支持，例如指定超时时长，指定错误处理器。为了处理这种复杂的选项，可以使用 JSON 格式的回调。使用 JSON 格式的回调，可以指定更复杂的回调选项，例如超时时长和错误处理函数。

下面再定义一个非常简单的 Java 类，该类中包含了如下一个简单的方法：

程序清单：codes\13\13.6\json-callback\WEB-INF\src\lee\HelloDwr.java

```
public class HelloDwr
{
    //定义一个简单的方法
    public String hello(String name)
    {
        return name + ", 您好! 您正在学习使用 JSON 格式的回调...";
    }
}
```

在 dwr.xml 文件中配置该 Java 类，即可将该类的实例转换成客户端的 JavaScript 对象。在客户端调用 JavaScript 对象既可使用前面介绍的简单回调，也可以使用如下 JSON 格式的回调。

程序清单：codes\13\13.6\json-callback\hellodwr.js

```
//使用 JSON 对象指定各种回调选项
function sendMessage()
{
    var name = $("name").value;
    hello.hello(name ,
    {
        //指定回调函数
        callback:cb,
        //指定超时时长
        timeout:5000,
        //指定错误处理函数
        errorHandler:function(message) { alert("错误提示: " + message); },
        warningHandler:function(message) { alert("Oops: " + message); },
        textHtmlHandler: function(message) { alert("Oops: " + message); },
        exceptionHandler: function(message) { alert("Oops: " + message); },
        //指定发送请求的方式
        httpMethod:'POST',
        async:true,
        //指定发送异步请求的实现机制
```

```
        rpcType:dwr.engine.XMLHttpRequest,
        //指定发送请求之前的勾子函数
        preHook:function(){alert('远程调用之前...')},
        //指定发送请求之后的勾子函数
        postHook:function(){alert('远程调用之后...')}
    });
}
//回调函数
function cb(data)
{
    document.getElementById("show").innerHTML = data;
}
```

在上面的异步调用中，并不是简单地提供一个回调函数，而是提供了一个 JSON 格式的 JavaScript 对象，该 JSON 格式对象中指定回调函数、超时时长和错误处理函数等选项。上面几乎列出了 DWR2.0 调用所支持的全部选项（对于早期版本的选项则没有介绍），下面依次对这些选项进行解释：

- callback：该属性指定回调函数。回调函数只能有一个参数，该参数就是服务器的响应。
- timeout：该属性用于指定超时时长，该属性的值是一个整数值，单位是毫秒。
- errorHandler：该属性用于指定错误处理函数，对于 DWR1.x，如果服务端发生异常，该函数被激发；从 DWR2.0 开始服务端异常激发 exceptionHandler 函数。
- warningHandler：当因为浏览器的 bug 引起问题时，激发该函数，这个选项只有 DWR2.0 才支持。
- textHtmlHandler：当服务器响应不是正确 text/html 页面时，激发该函数。通常超时也会激发该函数。
- exceptionHandler：远程调用失败后激发该函数，通常当服务端处理异常，或者数据转换异常时会激发该函数。
- httpMethod：该属性只支持两个值：'GET'和'POST'，该属性用于指定发送请求的方法，分别对于 GET 请求和 POST 请求。
- async：该属性指定是否发送异步请求。该属性默认是 true，即使用异步请求。通常建议不要使用同步请求。
- rpcType：该属性指定远程调用的方式，该属性只支持三个值：dwr.engine.XMLHttpRequest、dwr.engine.Iframe 和 dwr.engine.ScriptTag。分别对应使用 XMLHttpRequest，Iframe 和 ScriptTag 实现远程调用。默认是 XMLHttpRequest 方式。
- preHook：该属性指定一个 Hook 函数，该函数将远程调用之前被激发。
- postHook：该属性指定一个 Hook 函数，该函数将远程调用之后被激发。

这种调用将更加灵活，但这种调用也有一个问题。假如有如下 Java 方法：

```
public String hello(Person p)
{
    ...
}
```

上面 Java 方法需要一个参数，但这个参数并非简单的字符串，而是一个对象。根据前面的介绍，为了传入 Java 对象，JavaScript 中需要创建一个 JSON 格式的对象。

如果 Person 类恰好有一个 callback 属性（当然这种概率不大，但也并不是完全没有可能），而且需要使用 JSON 格式的远程调用，将可能出现如下代码的调用。

```
hello.hello(
    {callback : yeeku},
    {callback : cb}
);
```

上面粗体字代码指定的两个对象几乎完全一样，它们都包含了 callback 属性，这样 DWR 会不会混淆呢？DWR 无法分辨 yeeku 和 cb 有什么区别，也许开发者心里清楚：yeeku 是 Person 对象的属性值，而 cb 是回调函数，但 DWR 无法分辨（正如笔者经常说的，千万不要让计算机迷惑，计算机一迷惑，就是我们错了）。

为了避免这种迷惑，Person 对象应该尽量避免使用 callback、timeout 和 errorHandler 等属性名。当然，DWR 还是有一定的“智能”的，它按如下规则搜索回调函数：

- 如果第一个参数或最后一个参数是一个函数，那么它就是回调函数。其他参数都是调用 Java 方法的参数。
- 如果最后一个参数是对象，且这个对象中有一个 callback 属性，而且该 callback 属性值是个函数，那么这个对象就是额外增加的 JSON 对象，其他参数都将传入 Java 方法作为参数。
- 如果第一个参数是 null ，则 DWR 认为没有回调函数。其他参数都将传入 Java 方法作为参数。尽管如此，DWR 还会检查最后一个参数是不是 null，如果是 null，则发出警告。
- 如果最后一个参数是 null，那么就没有 callback 函数。其他参数将被当成调用 Java 方法的参数。

注意：

永远将回调函数或指定回调选项的 JSON 对象作为最后一个参数。虽然也可以将它们作为第一个参数，但这很容易混淆，我们没有必要冒这个风险。DWR 之所以支持将回调函数、指定回调选项的 JSON 对象作为第一个参数，主要是出于兼容性考虑，DWR 并不推荐这种做法。

13.6.4 将客户端参数传递到回调函数

在一些特殊的情况下，回调函数不仅需要访问服务器返回的数据，还需要访问页面中的某个 JavaScript 变量。但回调函数默认只有一个参数，这该如何处理呢？为了解决这个问题，可以使用适配器设计模式，将只有一个参数的回调函数转换成有多个参数的函数，这种做法需要利用 JavaScript 的闭包（Closure）。

假如真正的回调函数为如下形式，该函数同时需要服务器数据和客户端数据：

程序清单：codes\13\13.6\dataFromBrowser\hellodwr.js

```
//真正的回调函数，同时需要服务器数据和客户端数据
function cb(fromServer , fromBrowser)
{
    var show = document.getElementById("show");
    show.innerHTML += "服务器数据：" + fromServer + "<br />";
    show.innerHTML += "浏览器变量：" + fromBrowser;
}
```

下面我们利用一个闭包函数来包装这个函数，将该函数包装成只要一个服务器参数的函数，如下代码所示：

程序清单：codes\13\13.6\dataFromBrowser\hellodwr.js

```
var fromBrowser = "客户端变量";
// 定义闭包函数来存储 fromBrowser 的引用，并调用 fromServer
var callbackAdapter = function(fromServer)
{
    cb(fromServer, fromBrowser);
};
```

于是我们就可把 callbackAdapter 当成回调函数，从而可以像平常一样来调用远程 Java 方法，只需将 callbackAdapter 当成回调函数即可。如下代码所示：

程序清单：codes\13\13.6\dataFromBrowser\hellodwr.js

```
function sendMessage()
{
```

```
    var name = $("name").value;
    //远程方法，callbackAdapter 包装了回调函数
    hello.hello(name , callbackAdapter);
}
```

上面代码中的 callbackAdapter 就是一个回调函数适配器，它负责将两个参数的回调函数转成只有一个参数的回调函数——只有一个参数的回调函数才符合 DWR 回调函数的要求。

归纳起来，使用 DWR 开发 Ajax 应用需要经过如下几个步骤：

（1）安装 DWR 框架。安装 DWR 框架需要如下几个步骤：①：将 DWR 所需要的 JAR 复制到 Web 应用 WEB-INF\lib 路径下。②：在 web.xml 文件中配置 DWR 的核心 Servlet。③：提供 dwr.xml 配置文件。

（2）开发 Java 类，Java 类里包含需要被远程调用的方法。

（3）在 dwr.xml 文件中使用创建器将 Java 类或对象创建成 JavaScript 对象。在 dwr.xml 文件中增加如下配置片段：

```
<create creator=" " javascript="jsObj">
    <param name=" " value=""/>
    ...
</create>
```

（4）进入 DWR 的测试页面查看转换得到的 JavaScript 对象。

（5）在 JSP、HTML 页面中通过 JavaScript 调用远程的 Java 方法。调用 Java 方法之前必须先导入必需的 JavaScript 库。

13.7 使用 engine.js

DWR 框架负责动态生成 JavaScript 库，它负责为每个导出的 JavaScript 对象生成 JavaScript 函数库。除此之外，它总会生成如下两个核心 JavaScript 函数库：①：engine.js 文件，这个文件是 DWR 的核心库，只要需要使用 DWR，就不能缺少该文件。②：util.js 文件，这个文件是 DWR 的工具类库。使用 util.js 库可以简化客户端的 DOM 操作，如果不想使用该文件提供的函数，则无需导入这个 JavaScript 库。

engine.js 对 DWR 非常重要，它是 DWR 的核心，只要用到 DWR 的地方就需要导入它。该函数库里有一个全局对象，该对象可用于设置 DWR 的一些全局属性，还可以实现批调用等功能。下面依次介绍 DWR 的这些全局选项。

➤➤13.7.1 设置调用顺序

Ajax 通常都是异步调用，因此，前面发送的请求，并不一定会在前面返回。也就是说，远程调用的服务器响应顺序与调用顺序往往并不相同。

通过调用 dwr.engine.setOrdered(boolean)方法，可严格限制请求的响应顺序，响应顺序将严格按发送顺序返回。为了达到这种效果，在旧的请求安全返回之前，DWR 不会发送新的请求。

DWR 默认并没有严格限制这种顺序，通过设置该属性为 true，可保证请求的发送顺序，与服务器的响应顺序完全一致。但这种做法会导致性能降低。

注意：

将这个属性设置为 true，将导致应用的性能下降。

➤➤13.7.2 设置全局超时时长

每次执行远程调用时，都可以通过 JSON 格式对象指定超时时长。但每次都指定超时时长是相当

繁琐的事情，DWR 的 dwr.engine 可设置全局的超时时长。可通过如下代码设置全局超时时长：

```
//设置全局超时时长
dwr.engine.setTimeout(5000);
```

上面的代码设置了一个全局超时时长，这个设置对所有的远程调用都有效，如果远程调用的响应时间超过了 5 秒，DWR 认为调用超时。

如果在执行远程调用时，再次通过 JSON 格式对象指定了超时时长，则全局设置被覆盖。

➤➤13.7.3 设置全局 Hook 函数

前面介绍使用 JSON 格式的回调参数时已经指出：通过 JSON 对象的选项可指定调用前的 Hook 函数，也可指定回调之后的 Hook 函数，通过这种方式的设置的 Hook 函数仅对单次调用有效。

如果想对所有异步调用设置通用的 Hook 函数，就可以通过 dwr.engine 对象的 setPreHook()和 setPostHook()两个方法实现。

通过 dwr.engine 设置的 Hook 函数是全局的，对所有的远程调用都有效。与单次调用时通过 postHook 和 preHook 指定回调函数类似，全局的 Hook 函数也有调用前 Hook 函数，调用后 Hook 函数。

设置全局的 Hook 函数可通过如下代码进行：

```
//设置全局的调用前 Hook 函数
dwr.engine.setPreHook(preFun);
//设置全局的调用后 Hook 函数
dwr.engine.setPostHook(postFun);
```

实际上，这种全局 Hook 函数非常有用。通常我们都希望异步调用的过程给用户某种方式的提示，这就是可以通过全局的 Hook 函数实现：当异步调用之前，让提示信息出现；当异步调用结束后，隐藏提示信息。

➤➤13.7.4 设置全局处理函数

通过 dwr.engine 可以设置如下几个全局处理器：

- 错误处理器：通过 setErrorHandler 方法设置，与前面介绍的 errorHandler 属性的意义相同。
- 警告处理器：通过 setWarningHandler 方法设置，与前面介绍的 warningHandler 属性的意义相同。
- 异常处理器：通过 setExceptionHandler 方法设置，与前面介绍的 exceptionHandler 属性的意义相同。
- 内容异常处理器：通过 setTextHtmlHandler 方法设置，与前面介绍的 textHtmlHandler 属性的意义相同。

这些处理器也可以在单次调用时重新设定，如果单次调用时设置了相应的处理器，则单次调用设置的处理器将会覆盖此处设置的全局处理器。

注意：

不要尝试设置全局回调函数——这没有意义，DWR 也不支持所谓的全局回调函数。

➤➤13.7.5 设置常用的全局选项

这些选项通常也可以在单次调用中设置。设置了全局选项后，可以避免每次调用都需要重复设置，因而设置全局选项可让代码更加简洁。如果需要单次调用时使用不同的设置，也可在单次调用时重新设置这些选项，单次调用时设置的选项将覆盖全局选项。

常用的全局选项以及对应的设置方法如下：

- 设置是否采用异步方式：默认是异步方式，可通过 dwr.engine.setAsync(flag)来改变默认方式，该方法只能使用 true 和 false 作为参数。

- 设置请求的方法：通过 dwr.engine. setHttpMethod (newmethod)方法来设置请求方法，该方法只接收两个值：POST 和 GET。
- 设置远程调用的方式：通过 dwr.engine.setRpcType 方法设置，该方法只接收 dwr.engine.XMLHttpRequest、dwr.engine.Iframe 和 dwr.engine.ScriptTag 三个值。通常使用 XMLHttpRequest 来实现远程调用，但如果浏览器禁用 ActiveX，则可使用 Iframe 代替，通常无需手动指定，DWR 会智能选择哪种实现方式。

13.7.6 批处理

当应用需要连续完成多次调用服务器方法时候，这样可能因为多次重复交互导致使用者等待，为了避免多次交互导致的性能下降，可使用 DWR 提供的批处理功能。

批处理可以一次完成多个远程调用，这样可以减少与服务器的交互次数，从而可以提交系统响应速度。

批处理通过 dwr.engine.beginBatch()方法开始，使用 dwr.engine.endBatch()方法结束。

一旦通过 dwr.engine.beginBatch()方法开始批处理后，即使我们发送了异步请求，但这些请求不会发送到服务器，而是在远程调用队列中等待，直到调用 dwr.engine.endBatch()来结束批处理，调用 dwr.engine.endBatch()方法之后，DWR 将会通过与服务器一次交互来完成全部调用。

注意：

不要忘了调用 endBatch()，否则所有的远程调用永远的处于列队中，永远都不会得到发送的机会。

另外还有有两个值得注意的地方：

- 不要在批处理中执行同步调用，否则将导致错误。
- 批处理中的远程调用，不能单独设置 hooks、timeouts 和 errorHandlers 等选项。这些选项都是对整批处理有效，而不是对单次调用生效。所以如果一个 batch 中有两个调用设置了不同的选项，最后一次的设置有效，其他设置都将被覆盖。

如果需要为批处理调用指定调用选项，通常建议在 endBatch()函数中设置批处理选项。如下面代码所示：

```
dwr.engine.beginBatch();
Remote.methodInBatch1(params, callback1);
Remote.methodInBatch2(params, callback2);
dwr.engine.endBatch({
    timeout:3000,
    errorHandler:function(msg){alert(messsage);}
});
```

上面的批处理将 method InBatch1 和 methodInBatch2 两个方法组织在一起，通过与服务器的一次交互完成两个方法的调用。

13.8 使用 util.js

util.js 中提供了一些工具函数，通过这些函数的帮助，我们可以以更简便的方式操作 DOM，这些工具函数主要用于简化 JavaScript 操作。

实际上 util.js 文件与 DWR 关系不是太大。如果喜欢，完全可以将该函数下载下来，然后可以在任何普通网页中使用，即使这个应用没有使用 DWR 亦可。util.js 中提供了 4 个基本的页面操作函数：getValue[s]()和 setValue[s]()可以操作除 table，list 和 image 之外的 HTML 元素，getText()可以操作 select

list。要修改页面的表格，可以使用 addRows()和 removeAllRows()。要修改列表（包括列表框和<ul.../>、<ol.../>等列表），则可使用 addOptions()和 removeAllOptions()函数。

下面详细介绍这些 util.js 中各函数的用法。

➤➤13.8.1 使用$()

看过前面 Prototype.库讲解的读者可能已经认识了这个函数。实际上，这个函数正是从 Prototype 库中借鉴过来的，它的作用也相似，该函数的作用与 document.getElementById("elementName")完全相同。

$()函数将根据指定 ID 查找文档中相应的 HTML 元素，该函数可以多个参数，如果传入多个 HTML 元素的 id 属性值作为参数，该函数将返回这些 id 对应元素的数组。

通常情况下，$()函数接受 HTML 元素的 id 属性值作为参数——这些参数都是字符串类型。如果传入该函数的参数不是字符串时，参数被不做任何处理直接返回。

从某种程度上讲，DWR 的$()函数功能更加强大，它可以在更多的浏览器上运行。

➤➤13.8.2 处理列表

此处所指的列表，不仅仅包括<select.../>元素产生的列表框、下拉菜单，还包括<ul.../>和<ol.../>列表。操作这种列表当然可直接通过 DOM 操作来完成，但通过 util.js 函数库将更加简单。

处理列表相关的函数主要由 dwr.util 对象的 removeAllOptions()和 addOptions()两个函数完成。其中 removeAllOptions()函数用于删除列表中的所有项，而 addOptions()则用于添加列表项，addOptions()共有如下 5 种形式：

- 字符串数组：dwr.util.addOptions(selectid, array)，第一个参数是列表元素的 id 属性值，第二个参数是字符串数组。该方法会为列表元素增加一系列选项，每个选项的文本和值都是字符串数组的元素。
- 对象数组：dwr.util.addOptions(selectid, data, prop)，第一个参数是列表元素的 id 属性值，第二个参数是个对象数组，第三个参数是数组元素的属性名。该方法会为列表元素增加一系列选项，每个选项的文本和值都是数组元素的 prop 属性值。
- 对象数组：dwr.util.addOptions(selectid, array, valueprop, textprop)，第一个参数是列表元素的 id 属性值，第二个参数是个对象数组，第三个参数是数组元素的属性名，第四个参数也是数组元素的属性名。该方法会为列表元素增加一系列选项，选项的文本是数组元素的 textprop 属性值，而选项值是数组元素的 valueprop 属性值。
- 对象：dwr.util.addOptions(selectid, mapObj, reverse)：第一个参数是列表元素的 id 属性值，第二个参数是个 JSON 格式的对象，第三个参数是个旗标，只能为 true 或 false。该方法会为列表元素增加一系列选项，选项的文本是 mapObj 对象的属性值，而选项值则是 mapObj 对象的属性名。reverse 旗标默认为 false，如果将该旗标设置为 true，各选项的文本是 mapObj 对象的属性名，而选项值是 mapObj 对象的属性值。
- 使用对象作为属性值的对象：dwr.util.addOptions(selectid, mapObj, valueprop, textprop) 用 mapObj 的每个属性值（每个属性值都是对象）创建一个选项。属性值对象的 valueprop 属性作为选项的 value，属性值对象的 textprop 属性作为选项的文本。

提 示：

第 3 个用法和第 5 个用法有点相似，它们都是根据对象数组来创建列表项。第 5 个方法可视为将 mapObj 对象的多个属性值合并在一起作为数组使用。

下面介绍的示例并没有使用 DWR，而是直接将 util.js 文件下载下来，并添加到 Web 页面中测试。下面程序先示范以字符串数组作为参数来添加列表项，看如下页面代码：

程序清单：codes\13\13.8\addOtions\StrArray.html

```
<body>
<select id="test"></select>
<input type="button" value="添加选项" onclick="add();">
<input type="button" value="删除选项" onclick="del();">
<script src="../util.js" type="text/javascript"></script>
<script type="text/javascript">
var strArr = ['请选择一本图书','疯狂 Java 讲义',
    '疯狂 Ajax 讲义','疯狂 XML 讲义'];
function add()
{
    //以字符串数组来为列表框添加列表项
    dwr.util.addOptions("test" , strArr);
}
function del()
{
    //删除下拉列表中的全部列表项。
    dwr.util.removeAllOptions("test");
}
</script>
</body>
```

在浏览器浏览该页面，单击“添加选项”按钮，将看到如图 13.5 所示的界面。

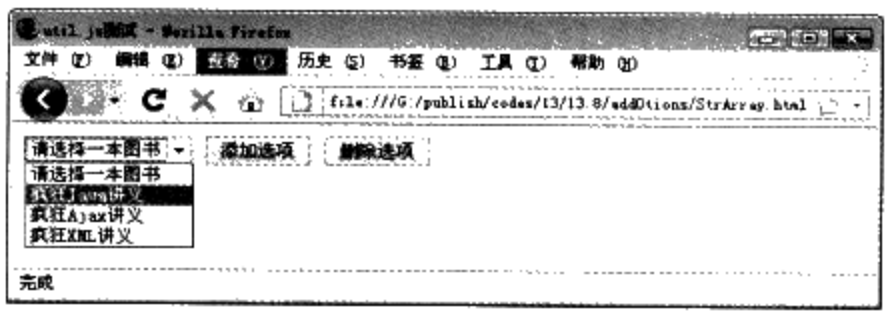


图 13.5 使用字符串数组为下拉菜单添加列表项

在上面的页面代码中，传给 addOptions()方法的参数是长度为 4 的字符串数组，因此每个选项的文本是数组元素，选项值也是数组元素。

下面页面代码使用对象数组作为下拉列表的选项，在这个示例中，每个选项的文本和值都是相同的。下面代码的 HTML 部分与前面的测试完全相同，此处不再列出，仅给出 JavaScript 部分。代码如下：

程序清单：codes\13\13.8\addOtions\ObjArray1.html

```
<script type="text/javascript">
//定义一个对象数组
var objArr = [
{name:'疯狂 Java 讲义'},
{name:'疯狂 Ajax 讲义'},
{name:'疯狂 XML 讲义'}];
function add()
{
    //以对象数组为列表框添加列表项
    dwr.util.addOptions("test" , objArr , 'name');
}
function del()
{
    //删除所有列表项
    dwr.util.removeAllOptions("test");
}
</script>
```

在浏览器中浏览该页面，并单击“添加选项”按钮，将看到如图 13.6 所示界面。

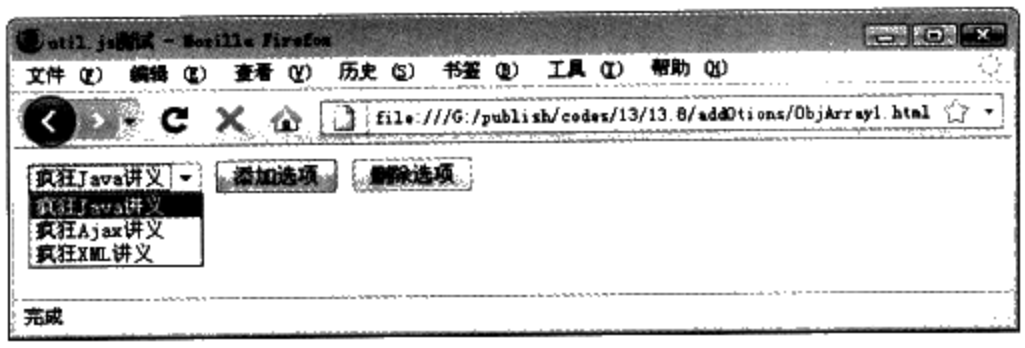


图 13.6　使用对象数组为下拉菜单添加列表项

上面页面中列表项的文本和值都是数组元素的 name 属性值，所以上面添加的各列表项的文本和值完全相同。

如果想让选项的文本和值使用数组元素的不同属性，则可使用四个参数的 addOptions 方法，看如下 JavaScript 代码（HTML 部分完全相同）。

程序清单：codes\13\13.8\addOtions\ObjArray2.html

```
<script type="text/javascript">
//定义一个对象数组
var objArr = [
{book:'疯狂 Java 讲义' , price:'99'},
{book:'疯狂 Ajax 讲义' , price:'79'},
{book:'疯狂 XML 讲义' , price:'69'}];
function add()
{
    //以对象数组为列表框添加列表项
    //以第三个参数指定的属性作为各列表项的值，
    //以第四个参数指定的属性作为个列表项的文本
    dwr.util.addOptions("test" , objArr , 'book' , 'price');
}
function del()
{
    //删除所有列表项
    dwr.util.removeAllOptions("test");
}
</script>
```

在浏览器中浏览该页面，单击“添加选项”按钮将可看到下拉菜单中增加了 3 个选项，选项的文本是 99、79 和 69，选项的值分别是“疯狂 Java 讲义”、“疯狂 Ajax 讲义”和“疯狂 XML 讲义”。

还有另一种使用单个对象添加选项的方法，对象的属性名作为列表选项的值，而对象的属性值作为列表选项的文本。当然，也可通过设置旗标为 true，使用属性名作为列表选项的文本，而属性值作为列表选项的值。看如下的 JavaScript 代码示例（HTML 部分完全相同）。

程序清单：codes\13\13.8\addOtions\Object.html

```
<script type="text/javascript">
var book = {
    name : "疯狂 Java 讲义",
    price : "99",
    publish : "电子工业出版社"
}
function add()
{
    //以对象为列表框添加列表项
    //对象的每个属性值将作为列表项的文本。
    //对象的每个属性名将作为列表项的值
    dwr.util.addOptions("test" , book);
}
function del()
{
```

```
    //删除下拉列表中的全部列表项
    dwr.util.removeAllOptions("test");
}
</script>
```

在浏览器中浏览该页面，并单击“添加选项”按钮，将看到如图 13.7 所示的界面。

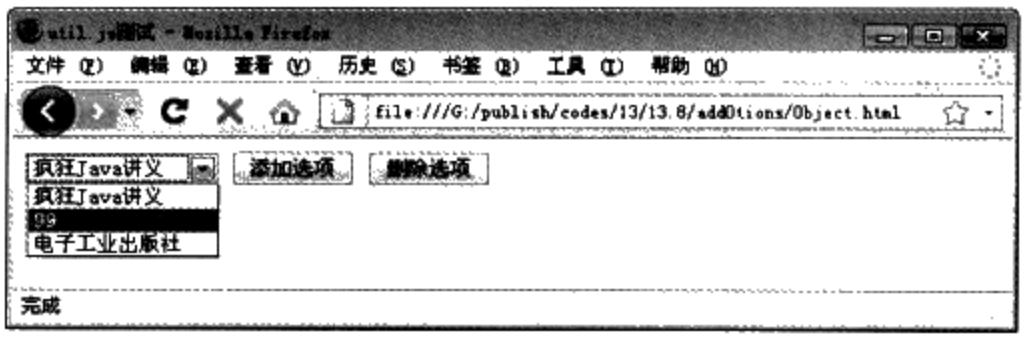


图 13.7　使用对象添加列表项

还有一种方式，也是使用对象来添加选项。在这种方式里，对象的每个属性值都是一个对象。看如下代码示例（HTML 部分完全相同）：

程序清单：codes\13\13.8\addOtions\MapObj.html

```
<script type="text/javascript">
//定义一个每个属性值都是对象的对象
var objMap = {
fisrt:{book:'疯狂 Java 讲义' , price:'99'},
second:{book:'疯狂 Ajax 讲义' , price:'79'},
third:{book:'疯狂 XML 讲义' , price:'69'},
fourth:{book:'轻量级 Java EE 企业应用实战' , price:'89'},
fifth:{book:'经典 Java EE 企业应用实战' , price:'89'}};
function add()
{
    //以对象添加列表项
    dwr.util.addOptions("test" , objMap , 'price' , 'book');
}
function del()
{
    //删除所有选项
    dwr.util.removeAllOptions("test");
}
</script>
```

这种用法是将 objMap 里的所有属性值拿出来创建列表项——每个属性值对应的对象创建一个列表项。在浏览器中浏览该页面，并单击“添加选项”按钮，将看到如图 13.8 所示的界面。

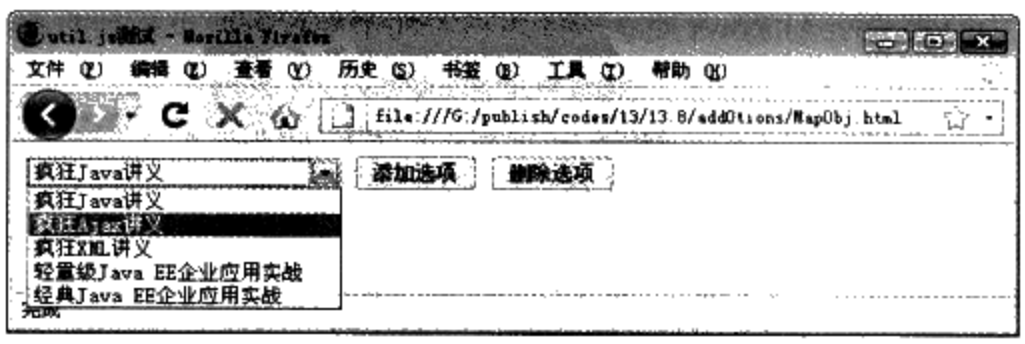


图 13.8　使用对象添加列表项

addOptions()不仅可用于操作<select.../>元素生成的列表框，也操作<ul.../>或<ol.../>元素对应的列表。在这种用法下，addOptions()仅仅支持前面 5 种语法的前 3 个，当使用 addOptions()方法来操作<ul.../>或<ol.../>的列表时，只能传入字符串数组或对象数组。如下代码所示：

程序清单：codes\13\13.8\addOtions\ol2.html

```
<body>
<ol id="test"></ol>
<input type="button" value="添加选项" onclick="add();">
<input type="button" value="删除选项" onclick="del();">
<script src="../util.js" type="text/javascript"></script>
<script type="text/javascript">
//定义一个对象数组
var objArr = [
{book:'疯狂 Java 讲义' , price:'99'},
{book:'疯狂 Ajax 讲义' , price:'79'},
{book:'疯狂 XML 讲义' , price:'69'}];
function add()
{
    //以对象数组为列表框添加列表项
    //以第三个参数指定的属性作为各列表项的文本
    //以第四个参数已经没有作用了
    dwr.util.addOptions("test" , objArr , 'book' , 'price');
}
function del()
{
    //删除所有列表项
    dwr.util.removeAllOptions("test");
}
</script>
</body>
```

在浏览器中浏览该页面，并单击“添加选项”按钮，将看到如图 13.9 所示的界面。

图 13.9　为<ol.../>元素添加列表项

➤➤13.8.3　处理表格

很多时候我们都需要动态操作 HTML 表格，例如为表格添加行，动态删除表格行等。dwr.util 提供了两个函数帮助我们处理 HTML 表格操作，这两个函数是 addRows()和 removeAllRows()，其中 addRows 用于向表格中添加行，而 removeAllRows 用于删除表格中的全部行。两个函数的语法格式如下：

- dwr.util.removeAllRows(tableId)：该函数只有一个参数，该参数是一个 HTML 表格元素的 id 属性值。为了更好地跨浏览器，此处最好使用<tbody.../>元素的 id 属性值。该函数将删除指定表格内的所有行。
- dwr.util.addRows(tableId , array , funArray , [option])：该函数的第一个参数与 removeAllRows 函数的参数相同；第二个参数是个数组，每个数组元素对应增加为表格增加一行；第三个参数是个函数数组，每个函数的返回值对应表格的一列。第四个参数是可选的选项，用于指定更复杂的选项。

下面先看一个简单的示例，该示例示范了如何根据一个字符串来生成表格：

程序清单：codes\13\13.8\addRows\Array.html

```
<body>
<table width="400" border="1">
```

```
<tr>
<th>城市</th>
<th>国家</th>
<th>洲</th>
</tr>
<tbody id="test"></tbody>
</table>
<input type="button" value="添加行" onclick="add();" />
<input type="button" value="删除行" onclick="del();" />
<script src="../util.js" type="text/javascript"></script>
<script type="text/javascript">
//定义一个字符串数组，每个数组元素对应表格一行
var rowArr = ['广州','华盛顿','伦敦'];
//定义一个函数数组，每个函数对于表格内的一列
var cellfuncs = [
function(data){
    return data;
},
function(data){
    if(data == '广州')return '中国';
    if(data == '华盛顿')return '美国';
    if(data == '伦敦')return '英国';
},
function(data){
    if(data == '广州')return '亚洲';
    if(data == '华盛顿')return '欧洲';
    if(data == '伦敦')return '欧洲';
}];
//为表格增加行
function add()
{
    dwr.util.addRows("test" , rowArr , cellfuncs);
}
//删除表格内的所有行
function del()
{
    dwr.util.removeAllRows("test");
}
</script>
</body>
```

上面代码中定义了一个函数数组，这个函数数组有 3 个函数成员。每个函数成员都是一个只有一个参数的函数，该函数的参数就是数组的元素。因此，如果使用一个包含三个元素的数组，使用有 4 个函数的数组，最终会添加 3 * 4 的表格内容。

在浏览器中浏览上面页面，并单击“添加行”按钮，将看到图 13.10 所示的界面。

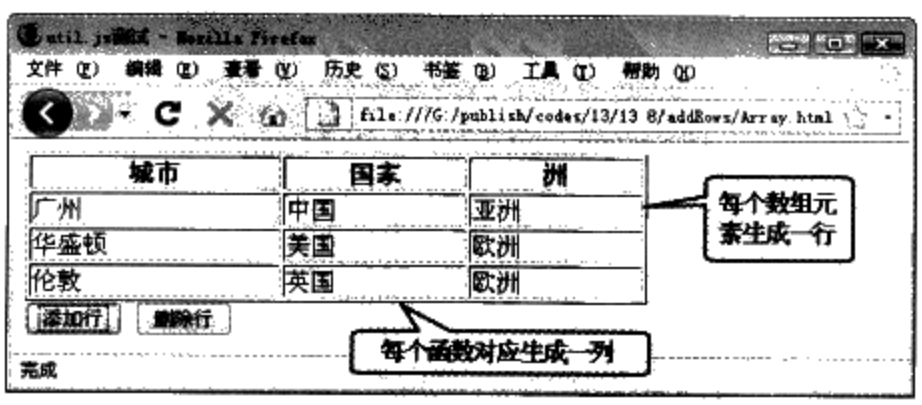


图 13.10　使用数组添加表格内容

表面上看起来，这种语法功能不够强大：甚至不能将一个二维数组的内容添加到表格中显示。那

只是一种错觉，实际上这种语法非常强大，它完全支持将二维数组的内容显示出来。看如下代码（因为 HTML 代码大同小异，此处没有列出）：

程序清单：codes\13\13.8\addRows\Array2.html

```
<script type="text/javascript">
//定义一个二维数组，用于在表格中输出
var rowArr = [['中国城市：','广州','上海','北京'],
    ['美国城市：','加州','华盛顿','纽约'],
    ['英国城市：','利物浦','伦敦','伯明翰']];
var cellfuncs = [
//表格函数，每个函数对应表格的一列。
//系列函数的 data 都是 rowArr 数组的元素——每个数组元素都是一维数组
function(data){
    return data[0];
},
function(data){
    return data[1];
},
function(data){
    return data[2];
},
function(data){
    return data[3];
}];
//添加表格行
function add()
{
    dwr.util.addRows("test" , rowArr , cellfuncs);
}
function del()
{
    dwr.util.removeAllRows("test");
}
</script>
```

在浏览器中浏览上面页面，并单击“添加行”按钮，将看到图 13.11 所示的界面。

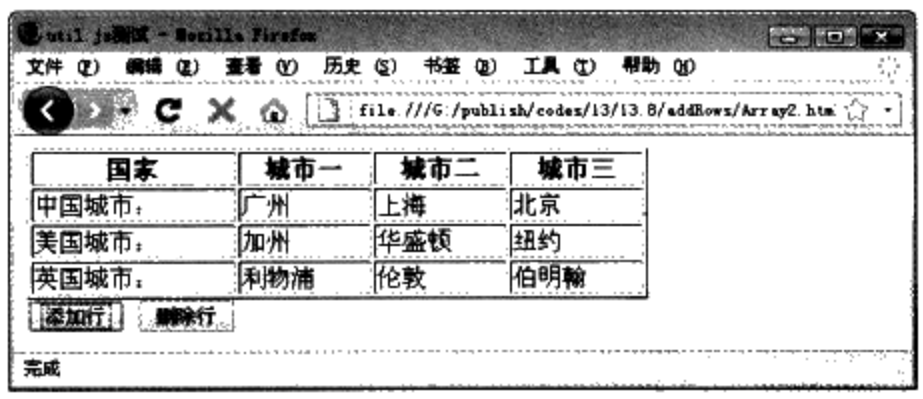


图 13.11　将二维数组添加到表格中显示

实际上，作为表格内容的数组不仅可以是普通字符串数组，可以是二维数组，也可以是 JSON 格式对象的数组。不管是那种数组，表格函数将依次访问每个数组内容。实际上有如下伪语法：

```
遍历内容数组的每个元素
{
    创建<tr> 标签
    遍历函数数组的每个函数
    {
        以当前数组元素的值，当前函数返回值创建一个单元格
    }
    创建</tr>标签
}
```

上面的伪语法就是 addRows()函数真实行为。实际上，作为表格内容的数组，也可以换成一个 JSON 格式的对象，addRows 将遍历对象的每个属性，每个属性对应为表格添加一行。

该函数还有一种更灵活的用法，这种用法使用了第四个参数，option。这个参数可指定如下两个选项：

- rowCreator：该选项指定一个函数，该函数重写了创建表格行的代码（例如可以增加 CSS 样式），该函数必须返回一个 HTMLTableRowElement 对象。该函数不能改变表格行的内容。
- cellCreator：该选项指定一个函数，该函数重写了创建单元格的代码（例如可以增加 CSS 样式），该函数必须返回一个 HTMLTableCellElement 对象。该函数不能改变单元格的内容。

在这两个属性指定的函数中，可以使用如下 5 个内建属性：

- rowData：就是当前表格行的数组元素或属性值，通过该属性可访问数组元素或 JSON 格式对象的属性值。对于同一行，所有单元格的 rowData 都是相同的。
- rowIndex：返回当前表格行 rowData 在数组的索引，如果表格内容是 JSON 格式对象，则返回其对应的属性名。
- rowNum：返回当前行在整个表格中的行索引值。
- data：当前单元格将显示的值，只能在 cellCreator 函数中使用。
- cellNum：返回当前单元格的列索引值，只能在 cellCreator 函数中使用。

注意：

使用这 5 个内建属性之时，一定不要忘记了在它们前面增加 options 前缀，也就是要使用 options.rowData 才可访问到当前表格行对应的数组元素或属性值。

下面的 JavaScript 代码稍微复杂一点，生成表格的内容使用一个 JSON 对象，addRows()函数将遍历该对象的每个属性，每个属性对应为表格添加一行。而且指定了 rowCreator 和 cellCreator 两个选项，让表格里奇、偶行的背景色不同，让单元格的前景色逐渐变淡。页面的 JavaScript 代码如下（HTML 部分没有给出）：

程序清单：codes\13\13.8\addRows\Advanced.html

```
<script type="text/javascript">
//提供一个 JSON 对象作为表格内容， JSON 对象的每个属性值都是数组
var rowArr = {中国城市:['广州','上海','北京'],
    美国城市:['加州','华盛顿','纽约'],
    英国城市:['利物浦','伦敦','伯明翰']
};
//表格函数数组，每个函数可访问 JSON 对象的属性值
var cellfuncs = [
function(data){
    return data[0];
},
function(data){
    return data[1];
},
function(data){
    return data[2];
}];
//创建表格的高级选项
var option =
{
    //指定 rowCreator 选项
    rowCreator:function(options)
    {
        var row = document.createElement("tr");
```

```
            //如果当前行索引为偶数，设置其背景色
            if(options.rowNum % 2 == 0)
            {
                row.style.backgroundColor = "#bbbbbb";
            }
            return row;
        },
        //指定 cellCreator 选项
        cellCreator:function(options)
        {
            var cell = document.createElement("td");
            //根据当前列索引设置前景色
            var index = options.cellNum * 80;
            cell.style.color = "rgb(" + index + ","
                + index + "," + index + ")";
            return cell;
        }
    };
    //添加表格行
    function add()
    {
        dwr.util.addRows("test" , rowArr , cellfuncs , option);
    }
    function del()
    {
        dwr.util.removeAllRows("test");
    }
    </script>
```

在浏览器中浏览该页面，并单击“添加行”按钮，将看到如图 13.12 所示的界面。

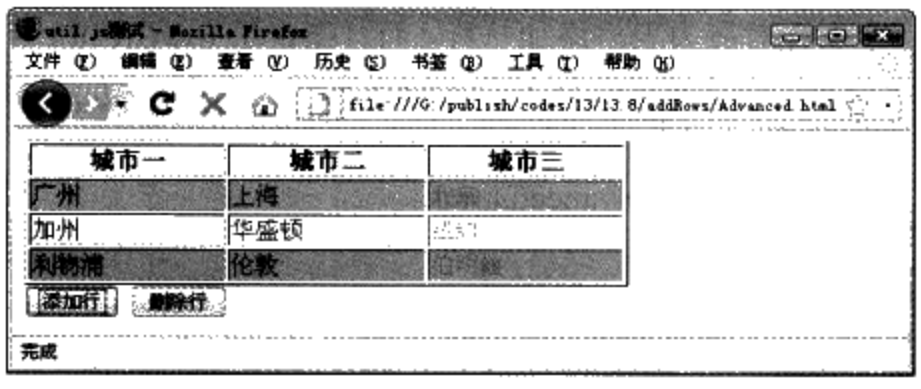


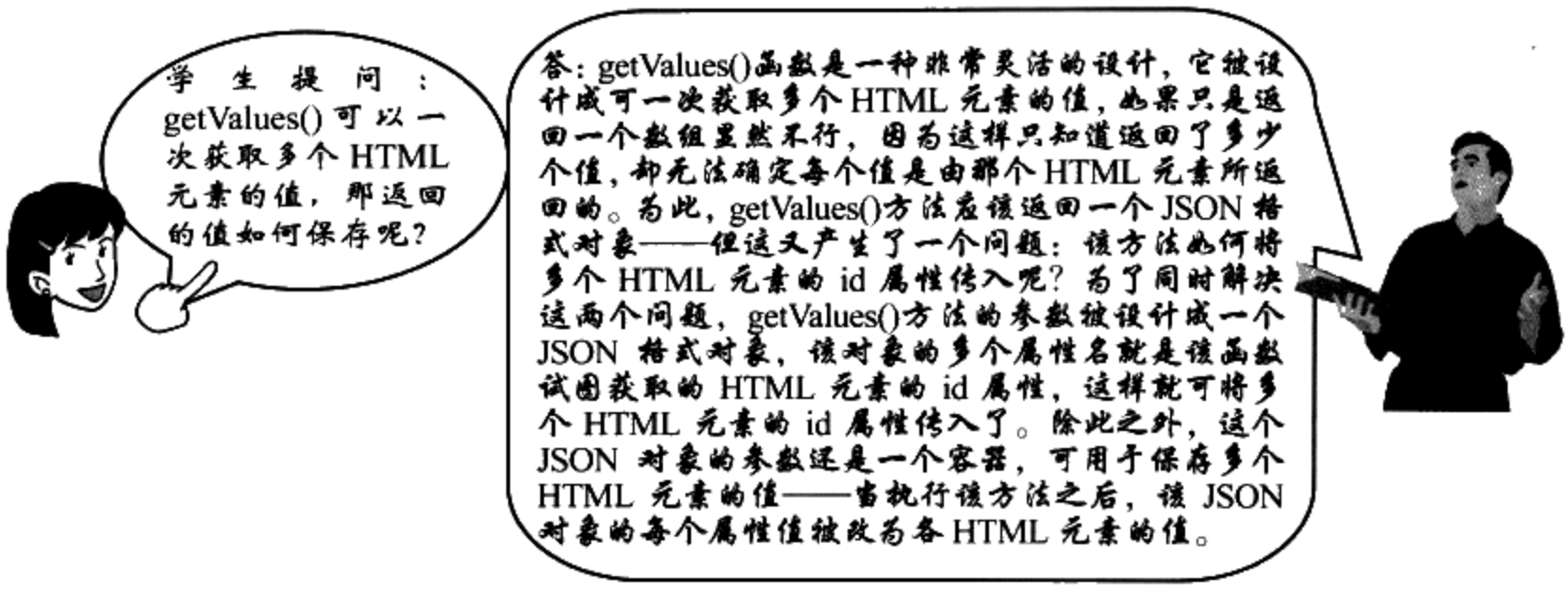
图 13.12　提供更多选项的创建表格行

➤➤13.8.4　访问 HTML 元素值

dwr.util 还提供了几个函数，这几个函数可用于简化访问和修改 HTML 元素的值。通过这几个函数的帮助，DOM 操作会更加简单。dwr.util 主要包含如下几个工具函数：

- getValue(elementId)：该函数用于返回 elementId 元素的值。
- getValues(obj)：该函数类似与 getValue，但它用于获得多个 HTML 元素的值。该函数的参数是一个 JSON 格式的对象，该对象由系列的 name/value 对组成。每个 name 都对应一个 HTML 元素，执行该方法后，obj 的属性值将修改为对应的 HTML 元素值。
- getText(selectId)：该函数类似于 getValue 函数，但这个函数只操作下拉列表。但用于返回选项的文本，而不是选项值。
- setValue(elementId, value)：该函数用于设置 elementId 元素的值。
- setValues(obj)：该函数的作用类似于 setValue 函数，但它接收一个 JSON 格式对象，该对象包含系列的 name/value 对，每个 name 都对应一个 HTML 元素。执行该方法后，HTML 元素

的值就被修改为 obj 对象里对应属性的值。



下面的简单代码测试了 getValue()和 setValue()函数的用法：

程序清单：codes\13\13.8\value\get-setValue.html

```
<body>
<script src="../util.js" type="text/javascript"></script>
<input id="test" type="text"/>
<input type="button" value="设置值"
    onclick="dwr.util.setValue('test' , '测试用的值');"/>
<input type="button" value="获取值"
    onclick="alert(dwr.util.getValue('test'));"/>
</body>
```

这个页面很简单，如果在浏览器中测试该页面，单击“测试值”按钮时，将看到文本框的值变为“测试用的值”；如果在文本框内输入任意字符串，单击“获取值”按钮，将看到弹出提示框，框内的内容正是文本框内的输入。

setValues()和 getValues()则使用 JSON 格式的对象作为参数，该对象的每个属性都是页面中 HTML 元素的 id 属性值。分别用于设置 HTML 元素值，访问 HTML 元素值。看如下代码：

程序清单：codes\13\13.8\value\get-setValues.html

```
<body>
<script src="../util.js" type="text/javascript"></script>
<script type="text/javascript">
function set()
{
    //定义了一个 JSON 格式的对象，该对象的每个属性名
    //对应页面中的一个 HTML 元素的 ID 属性值。
    var obj = {
        txt : '测试文本',
        pass: '12345678',
        area: 'Ajax 是一个有趣的技术\n 我们应该学习它',
        select: 'two'};
    //调用该方法后，页面中对应的 HTML 元素值将被设置为上面 JSON 对象的属性值
    dwr.util.setValues(obj);
}
function get()
{
    //定义了一个 JavaScript 对象。
    //该对象的属性名对应于页面中的 HTML 元素的 id 属性
    var obj = {
        txt : null,
```

```
            pass: null,
            area: null,
            select: null};
        //调用 getValues 方法后，obj 的属性值将被设置为页面中对应 HTML 元素的值
        dwr.util.getValues(obj);
        var result = "";
        //通过遍历 obj 对象的属性，输出 obj 对象的所有属性值。
        for (var name in obj)
        {
            result += obj[name] + "\n";
        }
        alert(result);
}
</script>
<input id="txt" type="text" /><br />
<input id="pass" type="password" /><br />
<textarea id="area" rows="2" cols="50"></textarea><br />
<select id="select">
    <option value="one">一</option>
    <option value="two">二</option>
    <option value="three">三</option>
</select>
<input type="button" value="设置值" onclick="set();"/>
<input type="button" value="获取值" onclick="get();"/>
</body>
```

在浏览器中浏览该页面，如果单击“设置值”按钮，将可看到如图 13.13 所示的界面。

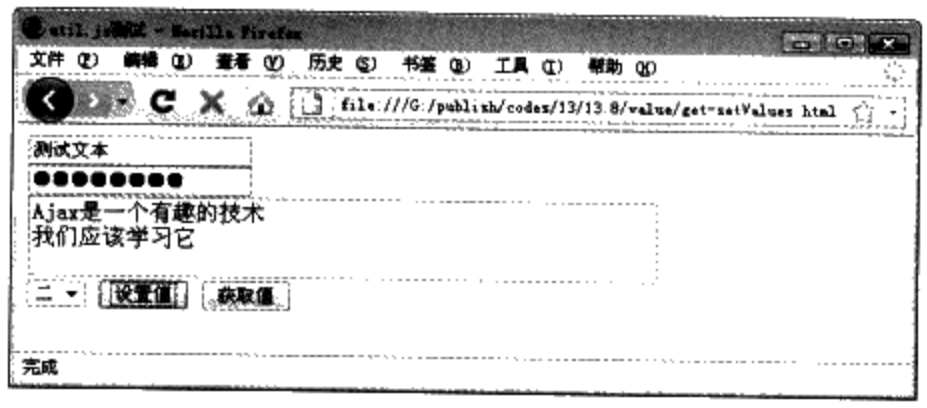


图 13.13　使用对象一次为多个 HTML 元素设置值

单击页面中的“获取值”按钮，将可看到如图 13.14 所示的警告框。

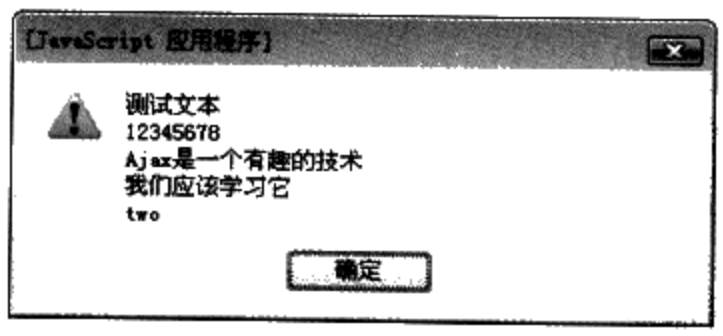


图 13.14　使用对象一次获取多个 HTML 元素的值

➤➤13.8.5　几个工具函数

从某种程度上来将，dwr.util 是个非常不错的 JavaScript 函数库，它提供了很多非常实用的功能。例如为调试增加的 toDescriptiveString()函数，还有处理表单提交按钮的键盘事件等。dwr.util 也许不如 JQuery 那么全面，不过它非常实用（实际上，DWR 完全可以与 JQuery 整合使用）。下面依次介绍这几个实用函数：

dwr.util.useLoadingMessage

用于当进行 Ajax 交互时设置用户的提示信息。该方法必须在页面加载后调用（不要在 onload()事件触发之前调用），因为该方法需要创建一个隐藏的<div.../>元素。

最简单的做法时在 onload 事件中调用 dwr.util.useLoadingMessage，示例代码如下：

```
<head>
<script type="text/javascript">
    //定义页面 load 时的初始化函数
    function init()
    {
        dwr.util.useLoadingMessage();
    }
</script>
    ...
</head>
<!-- 加载页面时，自动执行 init 方法 -->
<body onload="init();">
...
```

在某些情况下，例如需要保证 JavaScript 的 MVC 规则，不允许在 HTML 元素中手动添加控制代码，则可以通过如下方式来绑定 init 方法：

```
<script type="text/javascript">
function init() {
    dwr.util.useLoadingMessage();
}
if (window.addEventListener) {
    window.addEventListener("load", init, false);
}
else if (window.attachEvent) {
    window.attachEvent("onload", init);
}
else {
    window.onload = init;
}
</script>
```

这个函数非常简单，却不够灵活：提示信息总是红色的，总是英文提示信息，信息总是在右上角，而且这些都不允许用户自定义，DWR 可能在未来的版本会加强这个函数。

在现阶段，如果用户想实现自己的提示信息，或者假如提示图片，必须自己重新定义一个类似的方法。幸好这个方法的实现并不太难，它的思想是创建一个隐藏的<div.../>元素，当进行 Ajax 交互时，该元素出现；Ajax 交互结束时，该元素隐藏。

dwr.util.onReturn

用于处理表单元素的回车事件，很多时候，我们需要监控用户在文本框，或其他 HTML 元素中的回车事件，用以激发 JavaScript 函数。当然可以使用原生的事件处理机制来触发 JavaScript 函数。但不同浏览器的事件模型是不同的，比较麻烦。而 dwr.util.onReturn 则实现了跨浏览器的处理。

看如下简单代码片段：

程序清单：codes\13\13.8\onReturn.html

```
<body>
<script src="util.js" type="text/javascript"></script>
<script type="text/javascript">
function test()
{
    alert("单击了回车键");
}
</script>
```

```
    <input id="txt" name="txt" type="text"
        onkeypress="dwr.util.onReturn(event,test)"/>
    </body>
```

上面代码非常简洁，当用户在 txt 文本框元素内单击回车时将会触发 test()函数。

dwr.util.toDescriptiveString(elementId , level)

用于调试某个 HTML 元素，第一个参数是要调试的对象，第二个参数是可选的，用来指定内容深入的层次：0：单行调试，1：多行调试，但不深入子对象。2：多行调试，深入到第二层子对象。通常，第二个参数设定为 1 比较合适。

这个函数是 toString()函数的加强，它可比 toString 函数输出更多的信息。看如下的简单代码：

程序清单：codes\13\13.8\toDescriptiveString.html

```
<script type="text/javascript">
    var a = document.createElement("div");
    alert(dwr.util.toDescriptiveString(a , 1));
</script>
```

上面代码简单地创建了一个<div.../>元素，如果使用 toString 输出，则仅仅输出一个对象，看不到更详细的信息。使用了 toDescriptiveString 函数后，将可看到该对象更多详细信息。

dwr.util.selectRange(elementId , start , end)

用于在 elementId 输入框中选中文本，该函数选中从 start 到 end 的文本。看如下简单的代码：

程序清单：codes\13\13.8\selectRange.html

```
<script src="util.js" type="text/javascript"></script>
<input id="test" type="text" value="0123456789"/>
<input type="button" value="选中"
    onclick="dwr.util.selectRange('test',3,8)"/>
```

在上面代码中，包含了一个单行文本框，该单行文本框有 0123456789 的字符串，如果单击“选中”按钮，将通过 dwr.util.selectRange()函数来选中该文本框的第 3 个字符到第 8 个字符的文本。图 13.15 显示了这种效果。

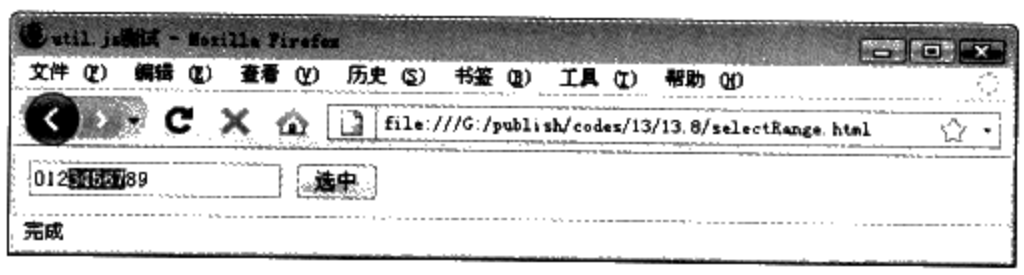


图 13.15 选中部分文本

13.9 整合第三方 Java EE 框架

DWR 是一个非常专业的 Java EE Ajax 解决方案，DWR 可以与很多 Java EE 框架整合，包括声名显赫的 Spring，这是一个非常实用的功能。也可与 Struts、JSF 和 WebWork 等 MVC 框架整合，但整合这些 MVC 框架是否具有真正大的意思？笔者将在后面进行深入的讨论，它还支持与 Hibernate 框架整合。

除此之外，由于 Ajax 技术主要是对传统 Web 应用的改进，因此少不了需要访问 Servlet API，DWR 当然不会没有这方面的支持。

➤➤13.9.1 访问 Servlet API

显而易见，Ajax 技术广泛应用于 Web 应用中，就少不了需要访问 Servlet API，例如一个处理登陆的服务器类，它应该将合法的用户名添加到 HttpSession 中。这就必然处理类可以访问 Servlet API，DWR

提供了两种方式访问 Servlet API。

使用 WebContext 类

DWR 提供了两个工具类：WebContext 和 WebContextFactory 类，这两个类可用于访问 Web 应用所在的 Servlet API。

其中 WebContextFactory 是产生 WebContext 工厂类，它提供了一个静态方法：get()，该方法可获得当前 Web 应用的 WebContext。一旦获得了 WebContext 类，就可以利用 WebContext 的如下几个方法：

- HttpServletRequest getHttpServletRequest()：该方法用于访问当前请求的 request 对象。
- HttpServletResponse getHttpServletResponse()：该方法用于访问当前请求对应的 response 对象。
- ServletConfig getServletConfig()：该方法用于访问 Web 应用的 ServletConfig 对象。
- ServletContext getServletContext()：该方法用于访问 Web 应用的 ServletContext 对象。
- HttpSession getSession()：该方法用于访问当前请求关联的 HttpSession 对象。
- HttpSession getSession(boolean create)：该方法用于访问当前请求关联的 HttpSession 对象。当当前请求没有关联的 HttpSession 时，如果 create 参数为 true，则创建一个新的 HttpSession 后返回；如果 create 参数为 false，则返回 null。

使用 WebContext 访问 Servlet API 简单易用，只要在处理类中通过这两个工具类访问 Web 应用的 Servlet API 即可。看如下处理类：

程序清单：codes\13\13.9\dwr-servlet1\WEB-INF\src\lee\AddSession.java

```
public class AddSession
{
    //该方法用于将 name 参数添加为一个 HttpSession 属性
    public void addSession(String name)
    {
        //通过 WebContextFactory 访问 Servlet API
        WebContextFactory.get().getSession(true)
            .setAttribute("user" , name);
    }
}
```

当上面的 addSession()方法被调用之后，本次会话 HttpSession 中即可多一个 user 属性，该属性值就是传入的参数。

这种使用 WebContextFactory 访问 Servlet API 的策略，优点是简单明了，开发者可以清楚地看到获得 Servlet API 的步骤；但缺点是处理类必须与 DWR API 耦合，引起代码污染。

直接访问 Servlet API

DWR 还提供了一种简化的策略，用以访问 Servlet API，在这种策略下，服务器处理类无需与 DWR API 耦合，降低了代码污染。在这种访问策略下，只要声明服务器处理方法时，额外增加需要访问 Servlet API 参数即可。看如下处理类：

程序清单：codes\13\13.9\dwr-servlet2\WEB-INF\src\lee\AddSession.java

```
public class AddSession
{
    //服务器处理方法中直接增加 HttpSession 参数，即可访问 HttpSession
    public void addSession(String name , HttpSession sess)
    {
        sess.setAttribute("user" , name);
    }
}
```

虽然服务器类的处理方法中增加了 HttpSession 参数，但将该服务器类暴露成 JavaScript 实例时，addSession()方法将没有 HttpSession 参数。就是说，依然使用如下方式来调用 addSession()方法：

```
//add 是 lee.AddSession 类转换成的 JavaScript 对象，
//调用 addSession 方法时，无需传入 HttpSession 参数。
add.addSession('yeeku');
```

从其他的 URL 读取数据

DWR 的 WebContext 对象还提供了如下方法：

- String forwardToString(java.lang.String url)：该方法可以用于读取 Web 应用中 url 资源，返回该资源所生成的响应。

借助于 WebContext 提供的 forwardToString()方法，可以非常方便地读取 Web 应用的其他资源。例如下面的 Java 处理类：

程序清单：codes\13\13.9\ReadOtherUrl\WEB-INF\src\lee\ReadOtherUrl.java

```
public class ReadOtherUrl
{
    //该方法用于将 name 参数添加为一个 HttpSession 属性
    public String read(String name)
        throws ServletException , IOException
    {
        WebContextFactory.get().getHttpServletRequest()
            .setAttribute("name" , name);
        return WebContextFactory.get()
            .forwardToString("/forward.jsp");
    }
}
```

上面服务器处理类可读取 Web 应用根路径下的 forward.jsp，并将该页面输出的响应作为本次异步调用的响应结果。

注意：

forwardToString()方法里指定的字符串总是以斜线（/）开头，因为该方法底层总是依赖于 HttpServletRequest 的 getRequestDispatcher ()方法。

本应用中 forward.jsp 页面是一个非常简单的页面，其页面代码如下：

程序清单：codes\13\13.9\ReadOtherUrl\forward.jsp

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 被转向的页面 </title>
    <meta name="website" content="http://www.crazyjava.org" />
</head>
<body>
```

```
<h3>name 请求参数的值：</h3>
<b>${requestScope.name}</b>
</body>
</html>
```

浏览器中以异步方式调用 lee.ReadOtherUrl 的实例的 read()方法后，forward.jsp 页面响应将会被发送到客户浏览器，HTML 页面使用如下简单代码来完成 Ajax 交互：

程序清单：codes\13\13.9\ReadOtherUrl\index.html

```
<body>
<script type='text/javascript' src='./leedwr/interface/read.js'></script>
<script type='text/javascript' src='./leedwr/engine.js'></script>
<script type='text/javascript' src='./leedwr/util.js'></script>
<script type='text/javascript' >
function cb(data)
{
    $("test").innerHTML = data;
}
</script>
<h3>读取其他 URL 的数据</h3>
请输入您的名字<input id="name" name="name" type="text"/><br />
<input type="button" value="发送请求"
    onclick="read.read(dwr.util.getValue('name') , cb);"/>
<hr />
<div id="test"></div>
</body>
```

单击页面的“发送请求”按钮，该页面将会发送一次异步请求，请求完成后看到如图 13.16 所示的效果。

图 13.16　读取其他 url 响应

➤➤13.9.2　整合 Spring

DWR 可以直接调用 Spring 容器中的 Bean，这种整合策略非常方便，根据 Java EE 的推荐架构，业务逻辑组件（通常也就是服务器处理类）通常需要依赖于更地层的 DAO 组件。如果不借助于 Spring 的 IoC 容器，业务逻辑组件与 DAO 组件的耦合又将降低到代码层次上。

如果 DWR 可以直接调用 Spring 容器中的业务逻辑 Bean，则所有的问题都可迎刃而解：Spring 负责为业务逻辑组件依赖注入 DAO 组件，并为业务逻辑组件提供声明式事务管理，还可在业务逻辑方法级别上增加基于 AOP 的权限控制（关于 AOP 的介绍请参阅疯狂 Java 体系的《轻量级 Java EE 企业应用实战》）……这一切都离不开 Spring 容器。

DWR 提供了一个 spring 的创建器，一旦使用了 spring 创建器，DWR 将负责搜索 Web 应用中的 Spring 容器，并将 Spring 容器中的 Bean 转换成一个浏览器中 JavaScript 可调用的对象。

下面定义一个简单的服务器处理类，该服务器处理类将被配置在 Spring 容器中，该 Java 类代码如下：

程序清单：codes\13\13.9\dwr-spring\WEB-INF\src\lee\HelloSpring.java

```
public class HelloSpring
{
    //服务器处理方法
    public String hello(String name)
    {
        return name + "您好，您已经会调用 Spring 中的 Bean 了...";
    }
}
```

上面的服务器处理类并未使用 Spring 的依赖注入（这不是本书的重点，读者可以在下一章看到更完整的案例），该处理类的处理方法只是返回一个简单字符串。

然后将该处理类部署在 Spring 容器中，下面是 Spring 配置文件的代码：

程序清单：codes\13\13.9\dwr-spring\WEB-INF\applicationContext.xml

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 Spring 配置文件的 Schema 信息 -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <!-- 定义了一个 bean -->
    <bean id="hello" class="lee.HelloSpring" />
</beans>
```

将上面 Spring 配置文件放在 Web 应用的 WEB-INF 路径下，Spring 框架默认加载该路径下的 applicationContext.xml 配置文件。

提 示：

在 DWR 的早期版本中，DWR 不允许将 Spring 配置文件放在 WEB-INF 路径下。最新版本的 DWR 已经改进了这个问题，因此完全可以将 Spring 配置文件放在 WEB-INF 路径下。

为了让 Spring 容器在 Web 应用启动时初始化，应该在 web.xml 配置文件中增加如下配置片段：

程序清单：codes\13\13.9\dwr-spring\WEB-INF\web.xml

```
<!-- 使用 ContextLoaderListener 来初始化 Spring 容器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

经过上面步骤，可以保证：Spring 容器会随 Web 应用的启动而初始化，而且服务器处理类被部署成 Spring 容器中的一个 Bean。下面介绍如何通过 dwr.xml 文件暴露该 Bean。在 dwr.xml 文件中使用 spring 创建器可将 Spring 容器中的 Bean 创建成一个 JavaScript 对象。下面是本应用中 dwr.xml 配置文件代码：

程序清单：codes\13\13.9\dwr-spring\WEB-INF\web.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
    "http://getahead.ltd.uk/dwr/dwr20.dtd">
<dwr>
    <allow>
        <create creator="spring" javascript="hello">
            <!-- 指定使用 Spring 容器中的 hello Bean -->
            <param name="beanName" value="hello"/>
```

```
        </create>
    </allow>
</dwr>
```

完成了上面定义后，Spring 容器的 hello Bean 将被创建成名为 hello 的 JavaScript 对象。可以在浏览器 JavaScript 代码中调用该对象。下面是调用 hello 对象的 JavaScript 代码。

程序清单：codes\13\13.9\dwr-spring\hellodwr.js

```
function sendMessage()
{
    //调用远程的 hello 方法，使用了 dwr.util 的 getValue 方法获取 HTML 元素的值
    hello.hello(dwr.util.getValue('name') , cb);
}
//回调方法
function cb(data)
{
    //使用 dwr.util 的 setValue 方法设置 HTML 元素的值
    dwr.util.setValue('show' ,data);
}
```

该应用的 HTML 页面非常简单，代码不再给出。在浏览器中浏览该页面，并激发调用远程方法的 sendMessage()方法，将可看到如图 13.17 所示的界面。

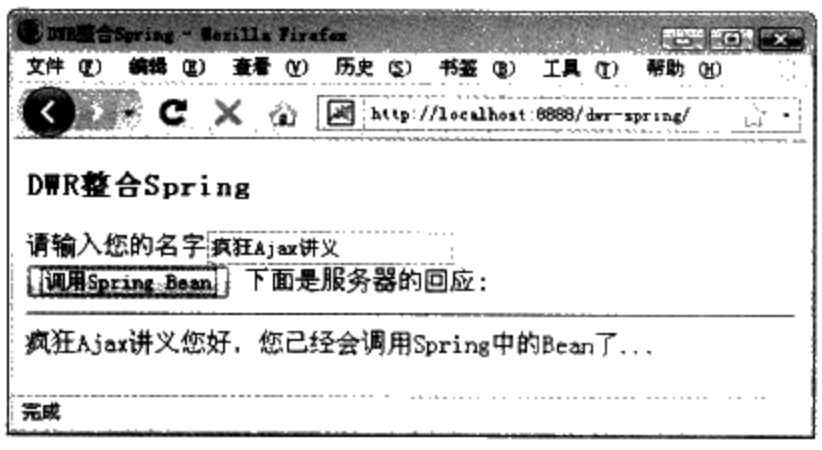


图 13.17　远程调用 Spring 容器中的 Bean

如果需要在在 dwr.xml 中强行指定需要使用哪些 Bean，可以使用 location*参数。这个 location*是个通配文件，它支持指定任意多个文件，只要参数以 location 开始并且唯一即可。例如：location-1、location-2。这些 location 被用做 Spring 的 ClassPathXmlApplicationContext 的参数，下面是这种用法的配置文件示例。

```
<allow> ...
    <create creator="spring" javascript="hello">
        <param name="beanName" value=" hello "/>
        <param name="location" value="beans.xml"/>
    </create>
</allow>
```

DWR 还提供了一种直接设置 BeanFactory（Spring 容器）的方法，SpringCreator 有一个静态的 setOverrideBeanFactory(BeanFactory)方法，通过该方法，可以使用编程方式直接设置 BeanFactory。

13.10　使用 DWR 注释

也许为顺应潮流（Struts 2、Spring、Hibernate 都允许使用注释来代替 XML 文件），DWR 也允许使用注释来代替 dwr.xml 文件。如果你愿意，甚至可以让注释和 dwr.xml 文件同时工作。

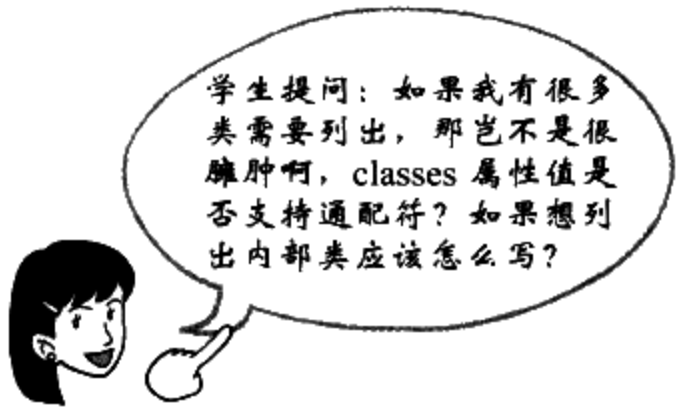
➤➤13.10.1 初始配置

为了使用 DWR 注释，必须在配置 DWR 核心 Servlet 时增加额外的配置参数：classes，该参数指定 DWR 自动加载哪些 Java 类——该属性值不仅需要列出所有服务器断的处理类，还需要列出所有作为 DTO（Data Transfer Object）使用的 Java 类。该属性值为多个 Java 类的全名，多个类名之间以英文逗号隔开。

本应用的的 web.xml 文件配置片段如下：

程序清单：codes\13\13.10\annotation\WEB-INF\web.xml

```
<!-- 配置 DWR 的核心 Servlet -->
<servlet>
    <!-- 指定 DWR 核心 Servlet 的名字 -->
    <servlet-name>dwr-invoker</servlet-name>
    <!-- 指定 DWR 核心 Servlet 的实现类 -->
    <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
    <!--  指定 DWR 核心 Servlet 处于调试状态 -->
    <init-param>
        <param-name>debug</param-name>
        <param-value>true</param-value>
    </init-param>
    <!-- 配置 DWR 自动加载哪些 Java 类 -->
    <init-param>
        <param-name>classes</param-name>
        <param-value>lee.HelloDwr,lee.Person</param-value>
    </init-param>
</servlet>
```



➤➤13.10.2 标注创建器和转换器

为了把一个 Java 类定义成可以远程访问的 Java 类，应该使用@RemoteProxy 和@RemoteMethod 注释，其中@RemoteProxy 用于标注一个 Java 类应转换成 JavaScript 对象，而@RemoteMethod 则用于标注一个方法应暴露成可远程方法的方法。

不管是采用 XML 配置文件的方式，还是采用 Annotation 来配置远程访问类，其实质完全一样，只是指定配置信息的方式不一样而已。使用@RemoteProxy 注释时可指定如下几个属性：

- creator：指定使用创建器的实现类。相当于 dwr.xml 文件中<create.../>元素的 creator 属性。只是该属性值必须指定创建器的实现类。
- Param[] creatorParams：为指定创建器指定必须的参数。相当于 dwr.xml 文件中<create.../>元素的<param.../>子元素。
- name：指定根据该 Java 类创建的 JavaScript 对象的名字。相当于 dwr.xml 文件中<create.../>元素的 name 属性。
- scope：指定将该 Java 类的实例放入哪个范围。相当于 dwr.xml 文件中<create.../>元素的 scope 属性。

下面我们定义一个简单的服务器类，并使用 Annotation 标注它，程序代码如下：

程序清单：codes\13\13.10\annotation\WEB-INF\src\lee\HelloDwr.java

```
@RemoteProxy(creator=NewCreator.class , name="hello")
public class HelloDwr
{
    //定义一个简单的方法
    public String hello(Person p)
    {
        return p.getName() + ", 您好！您正在学习使用 DWR 的注释...";
    }
}
```

上面文件的粗体字 Annotation 指定使用 new 创建器来转换该 Java 类，转换后得到的远程 JavaScript 对象名为 hello。

上面 hello()方法的参数是 Person 类，Person 类是一个 JavaBean 风格的类，如果使用 dwr.xml 文件管理配置信息，则必须使用<convert .../>元素配置转换。如果要采用 Annotation 来管理配置信息，应该使用@DataTransferObject 和@RemoteProperty 注视。其中@DataTransferObject 用于标柱一个需要被转换的 Java 类，该注释支持如下两个属性：

- converter：该属性指定使用怎样的转换器执行转换。相当于 dwr.xml 文件中<convert.../>元素的 converter 属性值。
- Param[] params：指定该转换器所需要的参数，相当于 dwr.xml 文件中<convert.../>元素的<param.../>子元素。

下面是本示例的 Person 类代码：

程序清单：codes\13\13.10\annotation\WEB-INF\src\lee\Person.java

```
@DataTransferObject
public class Person
{
    //使用 Annotation 标识下面两个属性是可转换的属性
    @RemoteProperty
    private String name;
    @RemoteProperty
    private int age;
    //无参数的构造器
    public Person()
    {
    }
    //省略 name、age 两个属性的 setter 和 getter 方法
    ...
}
```

通过使用 Annotation 来标注 Java 处理类、作为 DTO 使用的 Person 类（而且前面在 web.xml 文件中已经指定 DWR 自动加载这两个类），本应用不再需要 dwr.xml 配置文件，服务器开发工作已经完成。

客户端浏览器调用远程 Java 方法无需任何改变，故此处不再赘述。

13.11 异常处理

DWR 的异常处理是通过 dwr.engine 和单次调用时详细指定各种调用选项实现。

DWR 默认有自己的全局处理器，包括错误处理器：errorHandler、异常处理器：exceptionHandler 和警告处理器：warnningHandler。

通过调用 dwr.engine 的如下四个方法，可以改变系统默认的处理器：

- 错误处理器：通过 setErrorHandler 方法设置，与前面介绍的 errorHandler 属性的意义相同。
- 警告处理器：通过 setWarningHandler 方法设置，与前面介绍的 warningHandler 属性的意义相同。
- 异常处理器：通过 setExceptionHandler 方法设置，与前面介绍的 exceptionHandler 属性的意义相同。
- 内容异常处理器：通过 setTextHtmlHandler 方法设置，与前面介绍的 textHtmlHandler 属性的意义相同。

除此之外，DWR 也允许在单次调用和批量调用指定各种异常处理器，在单次调用中通过指定各种处理器选项来指定各种处理器，见如下示例代码。

```
hello.hello(params, {
    callback:function(data) { ... },
    errorHandler:function(errorString, exception) { ... }
});
```

在批量调用中可通过如下示例代码来设置批量调用的处理器：

```
dwr.engine.beginBatch();
hello.hello(params, function(data) { ... });
//其他的远程调用
//在 endBatch 中指定各种处理器选项。
dwr.engine.endBatch({
    errorHandler:function(errorString, exception) { ... }
});
```


提 示：

单次异步调用时指定异常处理器的实质就是前面介绍的 JSON 格式回调，通过使用 JSON 格式回调可以传入更多复杂的选项，从而可以对各种异常情况指定对应的处理函数。

DWR 可以转换异常，通过这种转换，可以将 Java 方法中的异常，变成 JavaScript 代码中的异常（不要尝试使用 try 块显式地捕捉异常，因为这种调用以异步方式进行）。

假如有如下的 Java 类：

```
public class HelloBean
{
    //下面方法抛出运行时异常：NullPointExcepion
    public String hello()
    {
        throw new NullPointerException("错误提示");
    }
}
```

在 JavaScript 可以通过如下方式来调用：

```
//定义一个错误处理函数
function errorHandler(msg)
{
    alert(msg);
}
//指定全局的错误处理函数
dwr.engine.setErrorHandler(errorHandler);
hello.hello(function(data) { alert(data); });
```

因为上面的 JavaScript 代码通过 dwr.engine 指定了全局错误处理函数，所有调用都会使用该错误处理函数，一旦服务器调用发生异常，都将触发该错误处理函数。

除此之外，当然也可在单次调用中通过 JSON 对象来指定错误处理函数，例如：

```
hello.hello(
```

```
{
    //指定回调函数。
    callback : function(data) { alert(data); },
    //指定错误处理函数。
    errorHandler:errorHandler
});
```

DWR 还提供了一个更强大的功能，可以将 Java 方法中的整个异常传给浏览器的 JavaScript 环境，而不是仅仅传递异常的 message 属性（Java 异常不仅仅包括 message 属性，还有更多丰富的信息，例如 SQLException 可包含错误号，SAX 异常可包含错误的行和列等等。）。这个功能原理非常简单，通过在 dwr.xml 文件中显式指定一个转换器，将 Java 异常转换成一个 JavaScript 对象，从而避免 Java 异常的信息丢失。

假如用户有一个自定义的异常类：

```
public class BusiException extends Exception
{
    public BusiException()
    {
    }
    public BusiException(String msg)
    {
        super(msg);
    }
}
```

对于 DWR 而言，上面的异常处理类与 JavaBean 类并没有太大区别。因此，可采用一个 bean 转换器将其转换成一个 JavaScript 对象。定义转换器的片段如下

```
<convert converter="bean" match="lee.BusiException"/>
```

一旦定义了这种转换后，当调用服务器中 Java 方法抛出的 lee.Business 异常时，该异常将被转换成一个 JavaScript 对象，该对象包含了该异常的更多详细信息。

JavaScript 代码中错误处理函数也应作出对应的修改：应为错误处理函数增加一个参数，通过该参数可访问原始的 Java 异常。

```
function errorHandler(msg , ex)
{
    //ex 是 Java 异常转换后对应的 JavaScript 对象
    ...
}
dwr.engine.setErrorHandler(errorHandler);
```

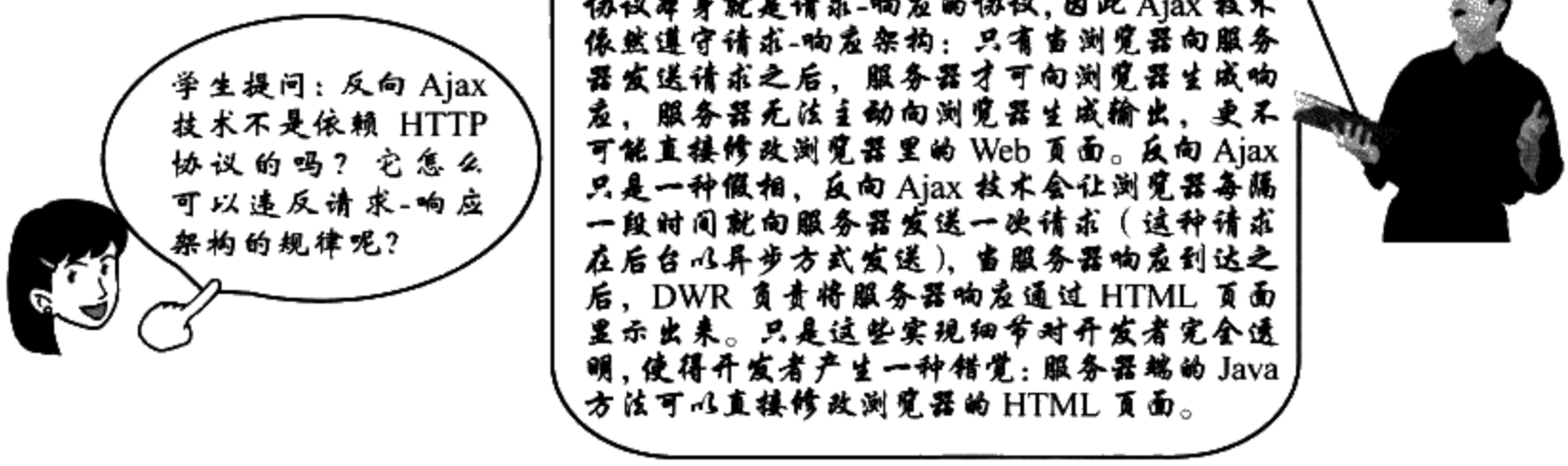
在上面的错误处理函数中，包含了两个参数：第一个参数是 Java 异常的 message 信息，而 ex 则是 Java 异常转换得到的 JavaScript 对象，通过访问 ex 对象的属性，可以访问原始 Java 异常的更多详细信息。

13.12 反向 Ajax

对于常规的 Web 应用和通常的 Ajax 应用，通常都是由浏览器发送请求，服务器响应，这种模式也被称为“请求-响应”模型。在这种模型下，浏览器每发出一个请求，服务器生成一个响应，服务器绝对不能主动向浏览器发送任何响应。

在反向 Ajax 模型里，服务器的 Java 方法可以“直接修改”浏览器里的 Web 页面，改变 Web 页面中组件的状态，服务器可以主动将最新的信息发送给浏览器。

➤➤13.12.1 配置使用反向 Ajax



为了使用反向 Ajax，需要修改两个地方来启动反向 Ajax：

- 修改 web.xml 文件中 DwrServlet 的配置，指定启动反向 Ajax。
- 修改 HTML 页面代码，指定该页面启用反向 Ajax 技术。

修改 web.xml 配置文件中 DwrServlet 的定义，为其增加 pollAndCometEnabled 属性，设置其值为 true。本应用的 web.xml 文件的配置如下：

程序清单：codes\13\13.12\ajaxChat\WEB-INF\web.xml

```
<?xml version="1.0" encoding="GBK"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
    <!-- 配置 DWR 的核心 Servlet -->
    <servlet>
        <!-- 指定 DWR 核心 Servlet 的名字 -->
        <servlet-name>dwr-invoker</servlet-name>
        <!-- 指定 DWR 核心 Servlet 的实现类 -->
        <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
        <!-- 指定 DWR 核心 Servlet 处于调试状态 -->
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
        <!-- 设置启用反向 Ajax 技术 -->
        <init-param>
            <param-name>pollAndCometEnabled</param-name>
            <param-value>true</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!-- 指定核心 Servlet 的 URL 映射 -->
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <!-- 指定核心 Servlet 映射的 URL -->
        <url-pattern>/leedwr/*</url-pattern>
    </servlet-mapping>
</web-app>
```

除此之外，还需要在使用反向 Ajax 技术的页面中设置启动反向 Ajax，通常可以在页面加载完成时通过 dwr.engine 来设置启用反向 Ajax。

下面是设置使用反向 Ajax 技术的代码片段（通过在 body 里设置 onload 属性让页面加载时使用反向 Ajax 技术）。

```
<!-- 本页面启用反向 Ajax -->
<body onload="dwr.engine.setActiveReverseAjax(true);">
```

提 示：

在 HTML 页面中启用反向 Ajax 技术的实质是启用一个定时器，这个定时器控制器周期性地向服务器发送请求，这就是反向 Ajax 实现的底层基础。

经过上面两个步骤的配置后，即可在该应用中使用反向 Ajax 技术了，这样就可以在 Java 代码中直接操作 HTML 页面了，在服务器端修改浏览器里的 HTML 页面。

➤➤13.12.2 在 Java 方法中操作 Web 页

为了在服务器端操作客户端 Web 页面，DWR 提供了一个 Util 的工具类，该工具类有两个常用的构造器：

- Util(java.util.Collection scriptSessions)：以一个 ScriptSession 集合构造 Util 实例，该实例可操作集合里多个 ScriptSession 实例所关联的浏览器里的 HTML 页面。
- Util(ScriptSession scriptSession)：以单个 ScriptSession 实例构造 Util 实例，该实例可操作 ScriptSession 所关联的浏览器里的 HTML 页面。

得到了 Util 实例后，Util 提供了如下系列方法用于操作 HTML 页面内容。

- addOptions：该方法是个重载的方法，包含多个操作选项，用于为列表元素添加选项。
- addRows：该方法是个重载的方法，包含多个操作选项，用于为表格增加表格行。
- cloneNode：该方法是个重载的方法，用于复制某个节点。
- removeAllOptions：该方法删除指定列表的全部选项。
- removeAllRows：该方法删除指定表格的所有行。
- removeClassName：删除指定 HTML 元素上的 class 属性，即通过 CSS 修改该元素的外观。
- removeNode：删除指定节点。
- setClassName：修改某个 HTML 元素的 class 属性，即通过 CSS 修改该元素的外观。
- setStyle：修改某个 HTML 元素的内联 CSS 样式。
- setValue：这是个重载的方法，用于为某个 HTML 元素设置值。
- setValues：该方法用于为多个 HTML 元素设置值。

为了创建 Util 对象，必须得到一个或多个 ScriptSession 对象，每个 ScriptSession 对象关联到一个浏览器。

提 示：

为了在服务器端通过 Java 来操作浏览器里的 HTML 页面，Java 程序必须获得该页面、或该应用关联的所有浏览器。一个 ScriptSession 对象就代表该页面、该应用所关联的浏览器会话。

为了获取 ScriptSession 实例，可以借助于 WebContext 的如下几个方法：

- getCurrentPage()：获取当前页面的 url。
- ScriptSession getScriptSession()：获取当前 WebContext 所关联浏览器会话。
- Collection getScriptSessionsByPage(java.lang.String url)：获取正在浏览 url 页面的所有浏览器会话。
- Collection getAllScriptSessions()：获取正在浏览当前应用的所有浏览器会话。

下面的应用是一个反向 Ajax 聊天室（部分参考了 DWR 官方示例的反向 Ajax 聊天室），其服务器

处理类的代码如下：

程序清单：codes\13\13.12\ajaxChat\WEB-INF\src\lee\JavaChat.java

```
public class JavaChat
{
    //保存聊天信息的 List 对象
    private LinkedList<ChatMsg> messages =
        new LinkedList<ChatMsg>();
    public void addMessage(String text)
    {
        if (text != null && text.trim().length() > 0)
        {
            messages.addFirst(new ChatMsg(text));
            //最多保留 10 条聊天记录
            while (messages.size() > 10)
            {
                messages.removeLast();
            }
        }
        WebContext wctx = WebContextFactory.get();
        //获取当前页面的 url
        String currentPage = wctx.getCurrentPage();
        //使用 utilThis 清楚 text 文本框的内容
        utilThis.setValue("text", "");
        //以当前 WebContext 关联的 ScriptSession 创建 Util
        Util utilThis = new Util(wctx.getScriptSession());
        //获取正在浏览当前页的所有浏览器会话
        Collection sessions = wctx.getScriptSessionsByPage(currentPage);
        //以 sessions 创建 Util 对象
        Util utilAll = new Util(sessions);
        //删除 chatlog 列表里的所有列表项
        utilAll.removeAllOptions("chatlog");
        //使用 messages 集合里集合元素的 text 属性为 chatlog 添加列表项
        utilAll.addOptions("chatlog" , messages , "text");
    }
}
```

从上面代码中看到，Java 类里操作浏览器中页面的关键类是 Util，创建 Util 对象时必须关联一个或多个浏览器会话——如果该 Util 关联一个浏览器会话，则 Util 只操作该浏览器里的 HTML 元素；如果该 Util 关联多个浏览器会话，则该 Util 同时操作多个浏览器里的 HTML 元素。

上面 JavaChat 类了还使用了 ChatMsg 类，使用该类的原因是因为 Util 的 addOptions()的参数必须是 JavaBean 集合，不支持直接使用字符串集合（笔者认为这相当令人遗憾）。ChatMsg 类仅仅只是字符串的包装而已，该类的代码如下：

程序清单：codes\13\13.12\ajaxChat\WEB-INF\src\lee\ChatMsg.java

```
public class ChatMsg
{
    //ChatMsg 包装的字符串
    private String text;
    public ChatMsg()
    {
    }
    public ChatMsg(String text)
    {
        this.text = text;
    }
    //省略 text 属性的 setter 和 gette 方法
    ...
}
```

13.12.3 在客户端调用反向 Ajax 方法

使用了反向 Ajax 技术后，由于 Java 方法将直接操作 HTML 页面的内容。因此无需使用回调函数，客户端 JavaScript 直接执行 Java 方法，该 Java 方法负责通知所有客户端浏览器更新内容。即可通过如下代码调用远程 Java 方法：

```
//调用远程 Java 方法，无需回调函数
chat.addMessage(dwr.util.getValue("text"));
```

该聊天室客户端 HTML 页面代码如下：

程序清单：codes\13\13.12\ajaxChat\java-chat.html

```
<!-- 本页面启用反向 Ajax -->
<body onload="dwr.engine.setActiveReverseAjax(true);">
<script type='text/javascript' src='./leedwr/engine.js'></script>
<script type='text/javascript' src='./leedwr/interface/chat.js'>
</script>
<script type='text/javascript' src='./leedwr/util.js'></script>
<script type="text/javascript">
function sendMessage()
{
    //调用远程 Java 方法，无需回调函数
    chat.addMessage(dwr.util.getValue("text"));
}
</script>
<h3>反向 Ajax 的聊天室</h3>
<div style="width:460px; height:200px;border:1px solid #999999">
<ul id="chatlog"></ul>
</div><br/>
输入您的聊天信息：
<input id="text" onkeypress="dwr.util.onReturn(event, sendMessage);"/>
<input type="button" value="发送" onclick="sendMessage()"/>
</body>
```

在浏览器中使用该聊天室进行聊天，将看到如图 13.18 所示的界面。

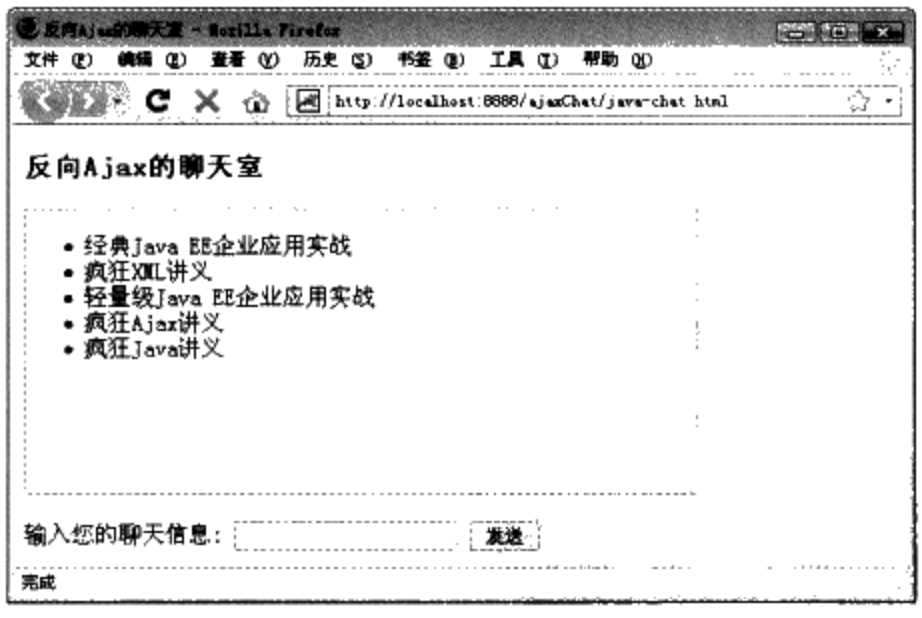


图 13.18 反向 Ajax 聊天室

纵观 DWR 的反向 Ajax 技术，不难发现一点：对开发者而言，反向 Ajax 技术消除了 JavaScript 代码的回调函数，允许在 Java 类中使用 Util 来操作 HTML 页面（当然，这是一种假相）。反向 Ajax 技术主要是为了迎合 Java 程序员的习惯：宁愿多写 Java 代码，尽量少些 JavaScript 代码。

就笔者来看，反向 Ajax 技术并不是十分必要的！对于一个成熟的 Web 程序员而言，JavaScript 编程也是一种基本功底。而且，借助于 dwr.util 对象的帮助，HTML 页面的 DOM 操作已经非常简单了。

13.13 本章小结

本章详细介绍了 DWR 的相关知识，本章所介绍的 DWR 是 2.0.5 版本，详细介绍了如何编写 DWR 的服务器处理类，怎样通过 DWR 配置文件将处理类创建成 JavaScript 实例，以及在客户端的 JavaScript 异步调用 Java 方法的方式。本章详细介绍了 DWR 支持的各种转换器、创建器，并介绍了如何在 dwr.xml 文件中声明方法签名。

本章重点介绍了如何使用 JavaScript 异步调用远程 Java 方法，包括简单回调、使用 JSON 对象指定更多回调选项、将客户端参数传入回调函数等。本章也详细介绍了 engine.js 和 util.js 两个函数库的用法。本章也详细讲解了 DWR 如何访问 Servlet API，如何让 DWR 和 Spring 整合。本章最后还介绍了 DWR2.0 的两个新特性：Annotation 和反向 Ajax 技术。

下一章将具体介绍如何在 Java EE 应用中使用 DWR 框架，下一章将介绍一个 DWR+Spring+Hibernate 的 Java EE 应用：即时消息系统，通过该系统示范如何将 DWR 框架有机地融入 Java EE 应用中。

CHAPTER 14

第14章 基于DWR的应用：即时消息系统

本章要点

- 分析、提取系统的 Domain Object
- 映射 Hibernate 的持久化对象
- 扩展 HibernateDaoSupport 提供分页功能
- 基于 HibernateDaoSupport 实现 DAO 组件
- 在 Spring 容器中部署 DAO 组件
- 实现业务逻辑组件
- 部署业务逻辑组件
- 使用声明式事务机制为业务逻辑方法增加事务控制
- 使用 AOP 机制为业务逻辑方法增加权限控制
- 使用 DWR 暴露 Spring 容器中的 Bean
- 处理用户登录、注册
- 处理发布新消息
- 处理获取消息列表、控制分页

本章将通过一个即时消息系统，示范如何将 DWR 和 Spring、Hibernate 两个框架整合起来，通过 DWR 直接调用 Spring 容器中的 Bean，让 DWR 和 Spring 无缝地整合在一起，而 Hibernate 则隐藏在 Spring 下面，提供持久层访问支持。

本章所实现的一样不是一个单纯的 Ajax 系统，可理解为原有 Java EE 应用增加了 Ajax 功能，或者说将 Ajax 技术有机地融入到了 Java EE 应用中。因此，系统采用轻量级 Java EE 架构，业务逻辑层封装了所有的业务逻辑方法，而 DAO 层封装了所有数据访问的原子方法。在这个应用里，MVC 框架消失了，因为系统让 DWR 直接访问 Spring 容器中的 Bean，DWR 代替了 MVC 框架。

由于应用缺少了 MVC 层框架，因此无法在 MVC 层进行权限检查，本应用将权限检查推迟到业务逻辑组件里进行，由于本应用使用 Spring 框架管理业务逻辑组件，因此可以借助 Spring AOP 框架来进行权限检查，从而可以将权限检查从业务逻辑代码中分离出来。

14.1 实现 Hibernate 持久层

本系统采用 Hibernate 完成持久层访问，Hibernate 持久层需要使用 POJO 和数据库映射文件。本系统一共有两个持久化类：User 和 Message。它们之间存在 1:N 的关系，即每个用户可发布多条消息，但每条消息只有一个发布者。

本系统要求用户必须登录后才可以发布消息、浏览消息。当用户进入系统时，如果用户没有登录，系统将弹出一个对话框，提示用户登录或注册。

➤➤14.1.1 Hibernate 持久层的 POJO

为了表示两个持久化实体之间的关联关系，可让两个 POJO 增加互相访问的属性字段。对于 1:N 的关联关系，1 的一端可增加 Set 属性，使用集合属性来保持关联的持久化实例，而 N 的一端则可以直接使用关联类属性来保持被关联实体。

下面是 User POJO 的源代码：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\model\User.java

```
public class User
{
    //标识属性
    private Integer id;
    //用户的用户名
    private String name;
    //用户的密码
    private String pass;
    //该用户所发布的全部消息
    private Set<Message> messages = new HashSet<Message>();
    //无参数的构造器
    public User() {}
    //初始化全部属性的构造器
    public User(Integer id , String name , String pass)
    {
        this.id = id;
        this.name = name;
        this.pass = pass;
    }
    //省略普通属性的 setter 和 getter 方法
    ...
    //messages 属性的 setter 和 getter 方法
    public void setMessages(Set<Message> messages)
    {
        this.messages = messages;
    }
```

```
    public Set<Message> getMessages()
    {
        return this.messages;
    }
}
```

本系统中 User 实体和 Message 实体之间存在 1:N 关系，因此上面的 User 类中使用 Set 集合来保存与 User 实体关联的全部 Message 实体，User 实体可以通过 getMessages()方法来获取该 User 发布的全部消息。

与之对应的是 Message 实体的 POJO，该实体需要保留一个属性，用以访问发布该 Message 的 User。Message 实体的 POJO 代码如下：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\model\Message.java

```
public class Message
{
    //标识属性
    private Integer id;
    //消息标题
    private String title;
    //消息内容
    private String content;
    //该消息的发布者
    private User user;
    //无参数的构造器
    public Message(){}
    //初始化全部属性的构造器
    public Message(Integer id , String title , String content)
    {
        this.id = id;
        this.title = title;
        this.content = content;
    }
    //省略了其他普通属性的 setter 和 getter 方法
    ...
    //user 属性的 setter 和 getter 方法
    public void setUser(User user)
    {
        this.user = user;
    }
    public User getUser()
    {
        return this.user;
    }
}
```

可能读者已经发现：User 和 Message 两个实体的属性都非常少，User 类仅有用户的用户名和密码两个属性，其他属性如性别、住址等详细信息都不存在。类似地，Message 也没有消息发布时间、发布者 IP 等信息。这是因为本章所示范的应用力求突出重点，放弃了无关紧要的枝节信息。毕竟，增加用户性别、住址等信息既不是本书的重点，也没有包含难以实现的难点，而且随着实际需求的不同，需要保留的信息也各不相同。因此该应用只保留了必要信息。

注意：

希望读者学习本系统时，多关注 Ajax 和 Java EE 的整合，不要纠缠于旁枝的细节。曾经有某网站专门发帖指出：实际系统不该使用明码来保存 User 密码，应该使用 MD5 之类的加密算法对密码加密。看到那个网站的发帖，让我感觉相当浅薄。原因如下：①本应用只是 Ajax 范例，加密密码既不是重点，也不是任何难点（随便增加个加密方法即可）。②即使在实际应用中，账户密码加密也不见得多么安全，因为如果让 cracker 物理接触到加密过的密码，他将有无数方法来侵入系统了。

在上面的两个实体中，可看到用户可以访问到其发布的全部消息，而由每条消息都可以访问其对应的发布者。但在实际应用中，往往不会通过用户访问其发布的全部消息。这是因为，当使用用户来访问消息时，所有的消息都保存在 Set 属性中，如果该用户发布的消息有百万条，则将导致系统一次加载百万条记录——这没有任何实际意义，因为任何页面都不可能一次显示百万条记录。一次加载百万条记录是非常耗内存的事情，因此不推荐从 1 的一端访问关联实体。

➤➤14.1.2 将 POJO 映射成持久化对象

为了让 POJO 具有持久层访问的能力，必须将 POJO 映射成持久化对象，为 POJO 增加相应的映射文件，映射文件指定了 POJO 与表之间的关联关系以及 POJO 属性和数据表字段之间的对应关系。

为了映射 1:N 关联，必须在 1 的一端增加<set.../>元素，该元素映射所有的关联实体；同时在 N 的一段增加<many-to-one.../>元素，该元素映射单个的关联实体。

下面是 User 持久化类的映射文件：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\model\User.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.message.model">
    <!-- 每个 class 元素映射一个持久化类 -->
    <class name="User" table="user_table">
        <!-- 映射标识属性 -->
        <id name="id" type="int" column="user_id">
            <generator class="identity"/>
        </id>
        <!-- 映射 name 普通属性 -->
        <property name="name" unique="true" type="string"/>
        <property name="pass" type="string"/>
        <!-- 映射 1:N 关联 -->
        <set name="messages" inverse="true">
            <key column="owner_id"/>
            <one-to-many class="Message"/>
        </set>
    </class>
</hibernate-mapping>
```

上面的映射文件将 User POJO 映射成持久化类，并指定了该持久化类和 user_table 表之间的对应关系。除此之外，上面的持久化类映射文件还使用<set.../>元素来映射和关联实体 Message 之间的 1:N 关系。

读者可能已经看到 set 元素中有 inverse="true"属性设置，增加该属性设置是为了保证更好的性能，对于 1:N 关联，通常推荐使用 N 的一端控制关联，而不要通过 1 的一端控制关联。

下面是 Message 持久化对象的映射文件：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\model\Message.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.message.model">
    <!-- 每个 class 元素映射一个持久化类 -->
    <class name="Message" table="message_table">
```

```
        <!-- 映射标识属性 -->
        <id name="id" type="int" column="message_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="title" type="string"/>
        <property name="content" type="string"/>
        <!-- 映射 N:1 关联实体 -->
        <many-to-one name="user" column="owner_id"
            not-null="true" class="User" lazy="false"/>
    </class>
</hibernate-mapping>
```

在 Message 实体的映射文件中，映射文件使用<many-to-one.../>元素映射关联实体，对于双向 1:N 关联，需要同时在<set.../>和<many-to-one.../>元素上指定 column 属性，用于指定外键列的列名，这两个 column 属性值应该相同，因为这两列指定的其实是同一列。在<many-to-one.../>元素中，增加了 not-null="true"属性，指定外键列不能为空，这要求每条消息必须有一个发布者，要求底层 message_table 表的 owner_id 不能为 null。

注意：

Hibernate 双向关联的两个实体中指定的外键列的列名必须相同。如果需要指定从表外键列不能为空，可以在<many-to-one.../>元素里指定 not-null="true"。

14.2 实现 DAO 组件

Java EE 应用中 DAO 组件的功能比较固定，总是用于提供对数据的 CRUD 操作，它大致上总包括如下几个方法：

- 创建实体，对应于增加记录。
- 根据主键获取实体，对应于根据主键加载一条记录。
- 删除指定实体，对应于根据主键删除记录。
- 根据指定主键删除实体，对应于根据主键删除记录。
- 修改实例，对应于修改指定记录。

除了上面的几个固定方法外，通常还包括了一系列的查询方法。

注意：

DAO 组件在功能上是模拟了一个轻量级的实体 EJB 的功能（当然实体 EJB 还包含了 Domain Object 的功能），但没有提供分布式应用的能力，也没有生命周期等功能。

本应用需要实现分页，DAO 组件应该可以根据指定的页码获得与之对应的记录。对于分页功能，在 Hibernate 中实现得很好，通过 Hibernate 中 Query 接口的如下两个方法，可以轻松实现分页：

- setFirstResult(int offset)：设置当前页需要的第一条记录。
- setMaxResults(int resultNum)：设置当前页显示的最多记录数。

但 Spring 提供的 HibenateTemplate 仅提供了 setMaxResults 方法，因而无法完成分页功能。为了实现分页功能，笔者扩展了 HibernateDaoSupport，为其增加了三个方法。

➤➤14.2.1 扩展 HibernateDaoSupport 来实现分页

学生提问：为什么不扩展 Hibernate Template 类来实现分页？扩展 HibernateDaoSup-p ort 是不是会引起一些混乱？

答：事实上，Spring 对 Hibernate 的封装确实是封装在 HibernateTemplate 类中的，不管是扩展 HibernateTemplate 还是扩展 HibernateDaoSupport，都可以实现分页。但我们都知道，HibernateDaoSupport 本身并不提供任何持久层访问操作，HibernateDaoSupport 的操作都是集中在 HibernateTemplate 中完成的。如果对 HibernateTemplate 进行扩展，但 HibernateDaoSupport 并不会返回扩展过的 HibernateTemplate 实例，依然返回原有的 HibernateTemplate 实例，这样将会导致扩展过的 HiberanteDaoSupport 依然返回原来的 HibernateTemplate。为了让 HiberanteDaoSupport 返回扩展后的 HibernateTemplate 实例，这样还是必须扩展 HibernateDaoSupport 类，重写它的 getHiberante Template 方法。考虑到这种情况，我们干脆直接扩展 HibernateDaoSupport，利用 Hibernate Template 的 executeFind()方法来增加三个分页查询的方法即可。

查看 HibernateDaoSupport 类的源代码，可看到如下三个方法：

```
//hibernateTemplate 类，就是 HibernateDaoSupport 用于操作数据库的 HibernateTemplate 实例
public final void setSessionFactory(SessionFactory sessionFactory)
{
    //依赖注入 sessionFactory 实例时，创建 HibernateTemplate 实例
    this.hibernateTemplate = createHibernateTemplate(sessionFactory);
}
//通过 Hibernate 的 SessionFactory 实例，创建一个 HibernateTemplate 实例
protected HibernateTemplate createHibernateTemplate(SessionFactory sessionFactory)
{
    return new HibernateTemplate(sessionFactory);
}
//该方法返回一个 HibernateTemplate 实例，该方法暴露给用户的 DAO 组件使用
public final HibernateTemplate getHibernateTemplate()
{
    return hibernateTemplate;
}
```

在上面的 HibernateDaoSupport 类中可看到，开发者无法重写 getHibernateTemplate()方法，除非重写 createHibernateTemplate()方法，让其返回一个扩展后的 HibernateTemplate 实例，然后再暴露一个方法，用于返回一个扩展后的 HibernateTemplate 实例——如此烦琐！

如果我们选择扩展 HibernateTemplate，则必然需要扩展 HibernateDaoSupport。相比之下，扩展 HibernateDaoSupport 则更加简洁，无须扩展 HibernateTemplate，只需要利用 HibernateTemplate 的 findQuery()方法进行分页查询即可。下面是扩展后 HibernateDaoSupport 类的代码：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\enhance\ YeekuHibernateDaoSupport.java

```
public class YeekuHibernateDaoSupport extends HibernateDaoSupport
{
    /**
    * 使用 HQL 语句进行分页查询操作
    * @param hql 需要查询的 HQL 语句
    * @param offset 第一条记录的索引
    * @param pageSize 每页需要显示的记录数
```

```
     * @return 当前页的所有记录
     */
    public List findByPage(final String hql,
        final int offset, final int pageSize)
    {
        List list = getHibernateTemplate().executeFind(
            new HibernateCallback()
            {
                public Object doInHibernate(Session session)
                    throws HibernateException, SQLException
                {
                    List result = session.createQuery(hql)
                    .setFirstResult(offset)
                    .setMaxResults(pageSize)
                    .list();
                    return result;
                }
            });
        return list;
    }
    /**
     * 使用 HQL 语句进行分页查询操作
     * @param hql 需要查询的 HQL 语句
     * @param value 如果 hql 有一个参数需要传入，则 value 就是传入的参数
     * @param offset 第一条记录的索引
     * @param pageSize 每页需要显示的记录数
     * @return 当前页的所有记录
     */
    public List findByPage(final String hql , final Object value ,
        final int offset, final int pageSize)
    {
        List list = getHibernateTemplate().executeFind(
            new HibernateCallback()
            {
                public Object doInHibernate(Session session)
                    throws HibernateException, SQLException
                {
                    List result = session.createQuery(hql)
                    .setParameter(0, value)
                    .setFirstResult(offset)
                    .setMaxResults(pageSize)
                    .list();
                    return result;
                }
            });
        return list;
    }
    /**
     * 使用 HQL 语句进行分页查询操作
     * @param hql 需要查询的 HQL 语句
     * @param values 如果 hql 有多个参数需要传入，则 values 就是传入的参数数组
     * @param offset 第一条记录的索引
     * @param pageSize 每页需要显示的记录数
     * @return 当前页的所有记录
     */
    public List findByPage(final String hql, final Object[] values,
        final int offset, final int pageSize)
    {
        List list = getHibernateTemplate().executeFind(
            new HibernateCallback()
```

```
            {
                public Object doInHibernate(Session session)
                    throws HibernateException, SQLException
                {
                    Query query = session.createQuery(hql);
                    for (int i = 0 ; i < values.length ; i++)
                    {
                    query.setParameter( i , values[i]);
                    }
                    List result = query.setFirstResult(offset)
                    .setMaxResults(pageSize)
                    .list();
                    return result;
                }
            });
        return list;
    }
}
```

➤➤14.2.2 实现 DAO 组件

每个 DAO 组件总会包括如下三个部分：

- DAO 接口：定义 DAO 组件所包含的方法
- DAO 实现类：实现 DAO 组件包含的方法。
- DAO 工厂：用于创建 DAO 实例。

因为扩展了 HibernateDaoSupport 支持类，所以 DAO 组件不再继承 HibernateDaoSupport，而是继承扩展后的子类 YeekuHibernateDaoSupport，使用该子类可以简单地实现分页。

对于使用 Spring 管理的 DAO 组件，Spring 容器就是 DAO 工厂。因此，无须编写额外的 DAO 工厂，只需要将 DAO 组件配置在 Spring 容器中即可。

下面是 Message DAO 组件接口的代码：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\dao\UserDao.java

```
public interface MessageDao
{
    /**
     * 根据主键加载消息
     * @param id 需要加载的消息 ID
     * @return 加载的消息
     */
    Message get(Integer id);
    /**
     * 保存消息
     * @param m 需要保存的消息
     * @retun 被保存消息的主键
     */
    int save(Message m);
    /**
     * 删除消息
     * @param m 需要被删除的消息
     */
    void delete(Message m);
    /**
     * 根据主键删除消息
     * @param id 需要删除的消息 ID
     */
    void delete(Integer id);
    /**
```

```
     * 修改消息
     * @param m 需要修改的消息
     */
    void update(Message m);
    /**
     * 查询指定页应该显示的消息（控制分页的查询方法）
     * @param pageNo 需要查询的页码
     * @return 查询到的消息集合
     */
    List findAllByPage(int pageNo);
}
```

上面的 DAO 接口里定义了 6 个方法，这些方法中除了一个 findAllByPage()方法其他都是标准的 DAO 方法。findAllByPage()是一个应用需要的 finder 方法，该方法是个典型的分页查询方法，需要接收一个 pageNo（第几页）参数，findAllByPage()方法将会返回指定页的所有消息实体。

Message DAO 组件的实现类代码如下：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\dao\impl\MessageDaoImpl.java

```
public class MessageDaoHibernate
    extends YeekuHibernateDaoSupport implements MessageDao
{
    //每页显示的消息数
    private int pageSize;
    //每页的消息数通过依赖注入管理
    public void setPageSize(int pageSize)
    {
    this.pageSize = pageSize;
    }
    /**
    * 根据主键加载消息
    * @param id 需要加载的消息 ID
    * @return 加载的消息
    */
    public Message get(Integer id)
    {
        return (Message)getHibernateTemplate().get(Message.class , id);
    }
    /**
     * 保存消息
     * @param m 需要被保存的消息
     * @retun 被保存消息的主键
     */
    public int save(Message m)
    {
        getHibernateTemplate().save(m);
        return m.getId();
    }
    /**
     * 删除消息
     * @param m 需要被删除的消息
     */
    public void delete(Message m)
    {
        getHibernateTemplate().delete(m);
    }
    /**
     * 删除消息
     * @param id 需要被删除的消息 ID
     */
    public void delete(Integer id)
```

```
    {
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 修改消息
     * @param m 需要被修改的消息
     */
    public void update(Message m)
    {
        getHibernateTemplate().saveOrUpdate(m);
    }
    /**
     * 查询指定用户指定页的消息
     * @param pageNo 需要查询的指定页
     * @return 查询到的消息集合
     */
    public List findAllByPage(int pageNo)
    {
        int offset = (pageNo - 1) * pageSize;
        return findByPage("from Message m order by m.id desc"
            , offset , pageSize);
    }
}
```

上面的 Message DAO 实现类继承了 YeekuHibernateDaoSupport 类，这样就可以非常简单地实现分页功能，正如上面的程序中粗体字代码所示，只要调用 YeekuHibernateDaoSupport 扩展的 findByPage() 方法即可完成分页查询。

由上面的程序可以看出，MessageDao 组件没有丝毫特别之处，实现 DAO 组件非常简单。实现 User DAO 组件同样简单，因此此处不再给出 User DAO 组件的实现。

因为在表现层执行分页时，每页显示的记录可能需要变化，因此将每页可显示的记录通过依赖注入管理，从而允许在配置文件中配置每页显示的记录数。

只要将这两个 DAO 组件配置在 Spring 容器中，Spring 容器就是 DAO 工厂，负责生成 DAO 组件的实例。下面是这两个 DAO 组件的配置片段。

```
<!-- 定义 userDao 组件 -->
<bean id="userDao"
    class="org.crazyjava.message.dao.impl.UserDaoHibernate">
    <!-- 为 userDao 组件注入 SessionFactory 实例 -->
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 定义 messsageDao 组件 -->
<bean id="messsageDao"
    class="org.crazyjava.message.dao.impl.MessageDaoHibernate">
    <!-- 为 messsageDao 组件注入 SessionFactory 实例 -->
    <property name="sessionFactory" ref="sessionFactory"/>
    <!-- 配置每页显示的消息条数 -->
    <property name="pageSize" value="2"/>
</bean>
```

将 DAO 组件配置在 Spring 容器中，就可以让 Spring 容器把 DAO 组件注入到业务逻辑组件中，避免让业务逻辑组件手动获取 DAO 组件，从而可以让业务逻辑组件和 DAO 组件分离。

14.3 实现业务逻辑组件

因为 DWR 可以和 Spring 框架无缝整合，可以直接将 Spring 容器中的 Bean 暴露成远程 JavaScript 对象，因此本系统将会把业务逻辑组件当成 DWR 的处理类。我们把这个处理类部署在 Spring 容器中，然后通过 spring 创建器将其创建成一个 JavaScript 对象，然后就可以在客户端 JavaScript 代码中调用该

Java 对象的方法。

➤➤14.3.1 业务逻辑组件的接口

此处业务逻辑组件所包含的方法，也就是客户端所需调用的方法，业务逻辑组件需要哪些业务逻辑方法，完全根据业务需要而定。而业务逻辑组件接口正用于定义该组件所包含的全部方法，该接口的代码如下：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\service\MessageManager.java

```
public interface MessageManager
{
    /**
     * 创建一条消息
     * @param title 新信息的标题
     * @param content 新消息的父节点
     * @param userId 创建消息的用户 Id
     * @param session 进行权限控制的 Session
     * @return 新创建消息的主键，如果创建失败，返回-1
     */
    int createMessage(String title , String content , int userId ,
        HttpSession session)throws MessageException;
    /**
     * 创建一个用户
     * @param user 新创建用户的用户名
     * @param pass 新创建用户的密码
     * @param session 新建用户后把用户名放入 HttpSession 中
     * @return 新创建用户的主键
     */
    int createUser(String user , String pass ,
        HttpSession session)throws MessageException;
    /**
     * 验证用户名是否可用，本系统不允许用户名重复
     * @param user 需要验证的用户名
     * @return 该用户名不重复、可用，返回 true，否则返回 false
     */
    boolean valid(String user)
        throws MessageException;
    /**
     * 验证用户登录是否成功
     * @param user 登录所用的用户名
     * @param pass 登录所用的密码
     * @param session 登录后需将用户名存入 session
     * @return 登录成功，返回登录用户 ID，否则返回-1
     */
    int login(String user , String pass
        , HttpSession session) throws MessageException;
    /**
     * 当前页面的浏览者是否登录
     * @param session 通过跟踪 Session 来判断用户是否登录
     * @return 已经登录返回当前用户 ID，否则返回-1
     */
    int isLogin(HttpSession session)
        throws MessageException;
    /**
     * 根据消息 ID 返回消息
     * @param id 消息 ID
     * @param session 进行权限控制的 Session
```

```
     * @return 指定 ID 对应的消息
     */
    MessageBean getMessage(int id
        , HttpSession session)throws MessageException;
    /**
     * 返回特定页面所有消息
     * @param pageNo 指定页码
     * @param session 进行权限控制的 Session
     * @return 指定页的全部消息
     */
    public List<MessageBean> getAllMessageByPage(int pageNo
        , HttpSession session)throws MessageException;
}
```

在上面的接口定义中，每个方法都抛出了 MessageException 异常，这是一个自定义异常，用于包装原始的异常信息。借助于 DWR 的异常转化机制，完全可以将 MessageException 异常转换成 JavaScript 对象，从而将更丰富的 Java 异常信息传递到客户端，浏览器端 JavaScript 代码里定义一个全局异常处理器，就可以访问到详细的异常信息。

从上面的粗体字代码可以看出，上面的业务逻辑方法中大都包含一个 HttpSession 类型的参数，因为这些方法实际是由 DWR 框架负责调用，因此 DWR 会传入 HttpSession 参数来调用这些方法。之所以为这些业务逻辑方法增加 HttpSession 类型的参数，主要是为了进行权限控制，本系统会将权限检查推迟到业务逻辑层进行控制。

➤➤14.3.2 业务逻辑组件的实现类

业务逻辑组件的实现类，实现了接口中定义的所有方法。因为业务逻辑组件的实现必须依赖于 DAO 组件，因此必须为依赖注入这些 DAO 组件提供相应的 setter 方法。下面是业务逻辑组件实现类的代码：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\service\impl\MessageManagerImp.java

```
public class MessageManagerImpl implements MessageManager
{
    //该业务逻辑组件所依赖的两个 DAO 组件
    private UserDao userDao;
    private MessageDao messDao;
    //依赖注入两个 DAO 组件所需的 setter 方法
    public void setUserDao(UserDao userDao)
    {
        this.userDao = userDao;
    }
    public void setMessDao(MessageDao messDao)
    {
        this.messDao = messDao;
    }
    /**
     * 创建一条消息
     * @param title 新信息的标题
     * @param content 新消息的父节点
     * @param userId 创建消息的用户 ID
     * @param session 进行权限控制的 Session
     * @return 新创建消息的主键，如果创建失败，返回-1
     */
    public int createMessage(String title , String content , int userId,
        HttpSession session)throws MessageException
    {
        try
        {
            User u = userDao.get(userId);
```

```
            if ( u != null)
            {
                Message m = new Message();
                m.setTitle(title);
                m.setContent(content);
                m.setUser(u);
                return messDao.save(m);
            }
            return -1;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("添加消息出现异常");
        }
    }
    /**
     * 创建一个用户
     * @param user 新创建用户的用户名
     * @param pass 新创建用户的密码
     * @param session 新建用户后把用户名放入 HttpSession 中
     * @return 新创建用户的主键
     */
    public int createUser(String user , String pass ,
        HttpSession session)throws MessageException
    {
        if (userDao.findByName(user) != null)
            throw new MessageException("该用户名已经存在");
        try
        {
            User u = new User();
            u.setName(user);
            u.setPass(pass);
            userDao.save(u);
            session.setAttribute("user" , u.getId());
            return u.getId();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("注册用户出现异常");
        }
    }
    /**
     * 验证用户名是否可用,本系统不允许用户名重复
     * @param user 需要验证的用户名
     * @return 该用户名不重复、可用，返回 true，否则返回 false
     */
    public boolean valid(String user)
        throws MessageException
    {
        try
        {
            if (userDao.findByName(user) == null)
                return true;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("验证用户名出现异常");
```

```
        }
        return false;
    }
    /**
     * 验证用户登录是否成功
     * @param user 登录所用的用户名
     * @param pass 登录所用的密码
     * @param session 登录后需将用户名存入 session
     * @return 登录成功，返回登录用户 ID，否则返回-1
     */
    public int login(String user , String pass
        , HttpSession session) throws MessageException
    {
        try
        {
            User u = userDao.findByName(user);
            if (u != null && u.getPass().equals(pass))
            {
                session.setAttribute("user" , u.getId());
                return u.getId();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("处理登录出现异常");
        }
        return -1;
    }
    /**
     * 当前页面的浏览者是否登录
     * @param session 通过跟踪 Session 来判断用户是否登录
     * @return 已经登录返回当前用户 ID，否则返回-1
     */
    public int isLogin(HttpSession session)
        throws MessageException
    {
        try
        {
            Object userId = session.getAttribute("user");
            if (userId != null)
            {
                return (Integer)userId;
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("验证用户是否登录出现异常");
        }
        return -1;
    }
    /**
     * 根据消息 ID 返回消息
     * @param id 消息 ID
     * @return 指定 ID 对应的消息
     */
    public MessageBean getMessage(int id
        , HttpSession session)throws MessageException
    {
```

```
        try
        {
            Message m = messDao.get(id);
            if (m != null)
            {
                return new MessageBean(0 , m.getTitle() ,
                    m.getContent() , m.getUser().getId() ,
                    m.getUser().getName());
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("获取消息内容出现异常");
        }
        return null;
    }
    /**
     * 返回特定页面所有消息
     * @param pageNo 指定页码
     * @param session 进行权限控制的 Session
     * @return 指定页的全部消息
     */
    public List<MessageBean> getAllMessageByPage(int pageNo
        , HttpSession session)throws MessageException
    {
        try
        {
            List ml = messDao.findAllByPage(pageNo);
            if (ml != null && ml.size() > 0)
            {
                List<MessageBean> result = new ArrayList<MessageBean>();
                for (Object obj : ml)
                {
                    Message me = (Message)obj;
                    result.add(new MessageBean(me.getId(),me.getTitle(),
                        null , 0 , me.getUser().getName()));
                }
                return result;
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new MessageException("获取消息列表出现异常");
        }
        return null;
    }
}
```

实现了上面的业务逻辑组件之后，将该组件配置在 Spring 容器中，并通过 DWR 的 spring 创建器将其创建成一个 JavaScript 对象。这样就可以让浏览器里的 JavaScript 代码直接调用该业务逻辑组件里的方法。

➤➤14.3.3 部署业务逻辑组件

部署业务逻辑组件非常简单，只要在 Spring 配置文件中使用<bean.../>元素配置该组件即可。除此之外，还必须为业务逻辑组件的业务逻辑方法提供事务管理。

本应用使用的是基于 Schema 的配置方式，在配置文件中使用 tx 和 aop 两个命名空间即可为业务

逻辑组件的所有业务方法配置事务管理。

下面是本应用中配置业务逻辑组件，以及为业务逻辑方法配置事务代理的配置片段。当然，整个应用的配置文件还包括 Hibernate 的 SessionFactory、数据源等。

程序清单：codes\14\message\WEB-INF\applicationContext.xml

```
<!-- 配置 messManager 业务逻辑组件 -->
<bean id="messManager"
    class="org.crazyjava.message.service.impl.MessageManagerImpl">
    <!-- 为业务逻辑组件注入 DAO 组件 -->
    <property name="userDao" ref="userDao"/>
    <property name="messDao" ref="messsageDao"/>
</bean>
<!-- 配置 Hibernate 的局部事务管理器，使用 HibernateTransactionManager 类 -->
<!-- 该类实现了 PlatformTransactionManager 接口，是针对 Hibernate 的特定实现-->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 配置 HibernateTransactionManager 时需要依赖注入 SessionFactory 的引用 -->
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 配置事务切面 Bean，指定事务管理器 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 用于配置详细的事务语义 -->
    <tx:attributes>
        <!-- 所有以 get 开头的方法是 read-only 的 -->
        <tx:method name="get*" read-only="true"/>
        <!-- 其他方法使用默认的事务设置 -->
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
<aop:config>
    <!-- 配置一个切入点，匹配 lee 包下所有以 Impl 结尾的类执行的所有方法 -->
    <aop:pointcut id="leeService"
        expression="execution(* org.crazyjava.message.service.impl.*Impl.*(..))"/>
    <!-- 指定在 leeService 切入点应用 txAdvice 事务切面 -->
    <aop:advisor advice-ref="txAdvice"
        pointcut-ref="leeService"/>
    <!-- 下面还有权限配置 -->
    ...
</aop:config>
```

配置文件中的斜体字代码配置了业务逻辑组件，并将 DAO 组件注入了业务逻辑组件。配置文件中的粗体字代码配置了一个事务管理器，这个事务管理器是针对 Hibernate 的局部事务管理的实现类。接下来的 tx 和 aop 两个空间用于为上面的业务逻辑组件配置声明式事务管理。

上面的业务逻辑组件仅仅实现了所需的业务功能，并未提供用户权限管理，因为本系统采用 DWR 作为前端框架，因此将用户权限检查推迟到业务逻辑层实现。

➤➤14.3.4 基于 AOP 的权限控制

Spring 的 AOP 机制提供了非常便捷的权限控制机制，开发者可以将权限检查逻辑集中管理，让权限管理逻辑横切多个业务逻辑方法。

Spring 支持使用 Annotation 风格的 AOP 管理机制，也提供了基于 XML Schema 配置风格的 AOP 管理机制，本应用将采用 XML Schema 配置风格的 AOP 管理机制。本应用的权限检查类只是一个普通 Java 类，无须使用 Annotation。代码如下：

程序清单：codes\14\message\WEB-INF\src\org\crazyjava\message\authority\AuthorityInterceptor.java

```
public class AuthorityInterceptor
{
    //进行权限检查的方法
    public Object authority(ProceedingJoinPoint jp)
        throws java.lang.Throwable
    {
        HttpSession sess = null;
        //获取被拦截方法的全部参数
        Object[] args = jp.getArgs();
        //遍历被拦截方法的全部参数
        for (int i = 0 ; i < args.length ; i++ )
        {
            //找到 HttpSession 类型的参数
            if (args[i] instanceof HttpSession) sess =
                (HttpSession)args[i];
        }
        //取出 HttpSession 里的 user 属性
        Integer userId = (Integer)sess.getAttribute("user");
        //如果 HttpSession 里的 user 属性不为 null，且大于 0
        if ( userId != null && userId > 0)
        {
            //继续处理
            return jp.proceed(args);
        }
        else
        {
            //如果还未登录，抛出异常
            throw new MessageException("您还没有登录，请先登录系统再执行该操作");
        }
    }
}
```

从上面的程序的粗体字代码可以看出：本应用权限控制逻辑十分简单，程序先访问到 HttpSession 对象，如果 HttpSession 的 user 属性不为 null，且大于 0，即表明该用户已经登录系统，程序可以继续向下执行；否则，系统抛出 MessageException 异常。

上面的 AuthorityInterceptor 类只是一个普通类，其 authority()方法也并无太大的特别之处，只是在方法里包含了一个 ProceedingJoinPoint 类型的参数而已。Spring 的 AOP 机制允许将该 Java 类配置成切面 Bean，下面是 Spring 配置文件中配置权限管理的片段：

程序清单：codes\14\message\WEB-INF\applicationContext.xml

```
<aop:config>
    <!-- 事务管理的配置 -->
    ...
    <!-- 将 authority 转换成切面 Bean
        切面 Bean 的新名称为 authorityAspect -->
    <aop:aspect id="authorityAspect" ref="authority">
        <!-- 定义一个 Around 增强处理，直接指定切入点表达式
            以切面 Bean 中的 authority() 方法作为增强处理方法 -->
        <aop:around pointcut=
            "execution(* org.crazyjava.message.service.impl.*Impl.*Message*(..))"
            method="authority"/>
    </aop:aspect>
</aop:config>
<!-- 定义一个普通 Bean 实例，该 Bean 实例将进行权限控制 -->
<bean id="authority"
    class="org.crazyjava.message.authority.AuthorityInterceptor"/>
```

利用 AOP 配置了上面的权限管理之后，用户只有在登录后才可调用业务逻辑组件中方法名里包含

Message 子串的方法。

14.4 调用业务逻辑组件

DWR 可以将 Spring 容器中的 Bean 暴露成远程 JavaScript 对象，这样就可以让浏览器端的 JavaScript 以异步方式来调用业务逻辑组件的方法。

➤➤14.4.1 将 Spring 容器中的 Bean 创建成 JavaScript 对象

为了将 Spring 容器中的 Bean 创建成 JavaScript 对象，必须使用 DWR 的 spring 创建器，该创建器需要指定 Spring 容器中 Bean 的名字。DWR 会自动搜索 Web 应用中的 Spring 容器，并将容器中的 Bean 创建成对应的 JavaScript 对象。

下面是将 Spring 中的 Bean 创建成 JavaScript 对象所使用的配置文件：

程序清单：codes\14\message\WEB-INF\dwr.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
"http://getahead.ltd.uk/dwr/dwr20.dtd">
<dwr>
    <allow>
        <!-- 使用 spring 创建器，创建一个名为 mm 的 JavaScript 对象 -->
        <create creator="spring" javascript="mm">
            <!-- 将 Spring 容器中的 messManager 创建成名为 mm 的对象 -->
            <param name="beanName" value="messManager"/>
            <!-- 使用 include 元素定义哪些方法将被暴露到客户端 -->
            <include method="createUser"/>
            <include method="createMessage"/>
            <include method="getAllMessageByPage"/>
            <include method="getMessage"/>
            <include method="isLogin"/>
            <include method="login"/>
            <include method="valid"/>
        </create>
        <!-- 定义 DWR 将 MessageBean 实例转换成 JavaScript 对象 -->
        <convert converter="bean"
            match="org.crazyjava.message.vo.MessageBean"/>
        <!-- 定义 DWR 将 MessageException 实例转换成 JavaScript 对象 -->
        <convert converter="bean"
            match="org.crazyjava.message.exception.MessageException"/>
    </allow>
</dwr>
```

在上面的配置文件中，使用了多个<include.../>元素，每个<include.../>元素对应一个 Java 方法，表示只有这些 Java 方法会被暴露给浏览器的 JavaScript 代码。除此之外，上面的配置文件的最后两行粗体字代码指定使用 bean 转换器来处理 MessageBean 和 MessageException 实例。

为了使用 DWR，当然需要在 web.xml 文件中配置 DWR 的核心 Servlet。除此之外，Spring 的容器也需要在 web.xml 文件中配置。下面是 web.xml 配置文件的代码：

程序清单：codes\14\message\WEB-INF\web.xml

```
<?xml version="1.0" encoding="GBK"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
    <!-- 配置 Web 应用启动时加载 Spring 容器 -->
```

```
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 配置 DWR 的核心 Servlet -->
    <servlet>
        <!-- 指定 DWR 核心 Servlet 的名字 -->
        <servlet-name>dwr-invoker</servlet-name>
        <!-- 指定 DWR 核心 Servlet 的实现类 -->
        <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
        <!-- 指定 DWR 核心 Servlet 处于调试状态 -->
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <!-- 指定核心 Servlet 的 URL 映射 -->
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <!-- 指定核心 Servlet 映射的 URL -->
        <url-pattern>/leedwr/*</url-pattern>
    </servlet-mapping>
</web-app>
```

完成了上面的定义后，可看到DWR的核心Servlet在leedwr处监听。因此，可以在leedwr下查看是否可以通过JavaScript访问Spring容器中的Bean。进入mm的调试控制台，将看到如图14.1所示的控制台页面。

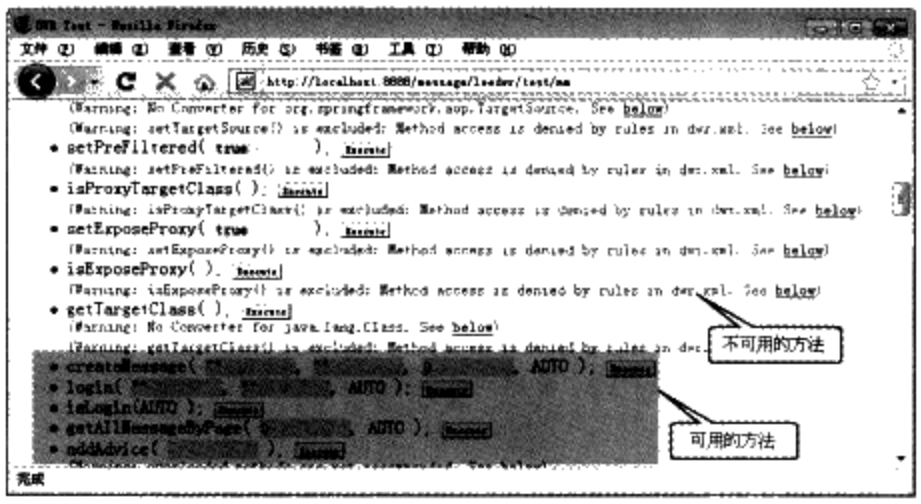


图14.1 DWR的调试控制台

在这个调试控制台中，可以看到所有可以被调用的Java方法下面没有特殊的说明，那些不可被调用的方法下有说明：access is denied by rules in dwr.xml，表明这些方法都不可通过JavaScript客户端调用。当然也可以通过该界面测试该Spring的业务逻辑组件是否正确。

在图14.1中可以看到，每个方法后都有一个“Execute”按钮，用于测试该方法，开发者可以在该按钮前输入合适的参数，然后单击“Execute”按钮来测试这些方法。

注意：

在该界面下所做测试的结果，会直接保存到底层数据库。例如在该界面下测试createUser()方法，将会导致系统增加一个用户。

14.4.2 处理用户登录

所有用户如需使用本系统，必须先登录。不管是发表消息，还是查看消息，都必须先登录系统。如果没有登录系统，将无法使用本系统。本系统除了在业务逻辑层进行了权限控制之外，也在JavaScript代码里进行了权限控制。

系统每次加载页面时都会判断用户是否已经登录，这就需要在页面加载时执行如下代码：

```
//判断用户是否已经登录
mm.isLogin(initCb);
```

调用 isLogin()业务逻辑方法时指定了一个回调函数 initCb，该函数用于根据用户是否登录决定下一步的操作。如果没有登录，系统将出现一个登录对话框；如果已经登录，则将 HttpSession 中的用户 ID 赋给 JavaScript 中的变量 curUser。下面是处理登录的回调函数代码：

程序清单：codes\14\message\mm.js

```
function initCb(data)
{
    //如果已经登录系统
    if (data > 0)
    {
        //将 HttpSession 中的 userId 赋给变量 curUser
        curUser = data;
        getAllMsgByPage();
    }
    //没有登录
    else
    {
        showDialog(login);
    }
}
```

在浏览器中第一次进入本系统可看到如图 14.2 所示的界面。

图 14.2 提示用户登录或注册

上面的代码中使用了 showDialog()函数来显示一个对话框，这是一个非常简单的函数，用于将指定对话框定位到屏幕中间，并通过修改其 CSS 样式让对话框显示出来。其代码如下：

程序清单：codes\14\message\mm.js

```
function showDialog(element)
{
    //定义 element 元素的位置
    element.style.top = document.documentElement.scrollTop
        + document.documentElement.clientHeight / 4 + "px";
    element.style.left = document.documentElement.scrollLeft
        + document.documentElement.clientWidth / 4 + "px";
    //让 element 元素显示出来
    element.style.display = "";
}
```

如果用户在对话框中输入用户名、密码，并单击“登录”按钮，将使 JavaScript 发送登录请求，

登录该系统的 JavaScript 函数如下：

程序清单：codes\14\message\mm.js

```
//------------用户登录------------
function userLogin()
{
    //必须输入用户名和密码才可以登录
    if (dwr.util.getValue("user") != null
        && dwr.util.getValue("user") != ""
        && dwr.util.getValue("pass") != null
        && dwr.util.getValue("pass") != "")
    {
        //调用 login 业务逻辑方法处理用户登录
        mm.login(dwr.util.getValue("user") ,
            dwr.util.getValue("pass") , loginCb);
    }
    else
    {
        alert("请先输入用户名和密码");
    }
}
```

如果用户名和密码匹配，将可看到如图 14.3 所示的对话框。如果用户单击该提示框中的“确定”按钮，如图 14.3 所示的对话框和用户登录的对话框同时消失，消息列表出现，用户将可以正常使用本系统。

如果用户输入的用户名和密码不匹配，也会弹出相应的提示框，提示用户输入的用户名和密码不对，登录对话框将不会消失。

图 14.3 登录成功

➤➤14.4.3 处理用户注册

如图 14.2 所示，该对话框既可用于注册，也可用于登录。因为笔者不想花太多时间去设计系统界面，因此登录、注册使用了同一个对话框。如果用户单击注册按钮，将触发注册事件，而注册事件将调用服务器的 createUser 方法，下面是处理用户注册的 JavaScript 函数。

程序清单：codes\14\message\mm.js

```
//------------用户注册------------
function userRegist()
{
    //如果用户名、密码不为空
    if (dwr.util.getValue("user") != null
        && dwr.util.getValue("user") != ""
        && dwr.util.getValue("pass") != null
        && dwr.util.getValue("pass") != "")
    {
        //调用 createUser()方法处理用户注册
        mm.createUser(dwr.util.getValue("user") ,
            dwr.util.getValue("pass") , registCb);
    }
    else
    {
        alert("请先输入用户名和密码");
    }
}
```

调用 createUser()方法时，使用了 registCb 回调函数，该函数将根据注册结果来提示用户。其代码如下：

程序清单：codes\14\message\mm.js

```
//-----------注册用的回调函数-----------
function registCb(data)
{
    //如果注册成功
    if (data > 0)
    {
        //将用户 ID 赋给变量 curUser
        curUser = data;
        alert("注册成功");
        login.style.display = "none";
        getAllMsgByPage();
    }
    else
    {
        alert("注册失败，请更换新的用户名和密码！");
    }
}
```

从上面的代码可以看出，用户注册和用户登录的处理也非常相似。实际也是这样，用户注册、登录共用了同一个对话框，只是注册要调用 createUser()业务逻辑方法，而登录调用 login()业务逻辑方法而已。用户注册成功后将看到如图 14.4 所示对话框。

图 14.4 注册成功

单击如图 14.4 所示中的“确定”按钮，注册成功的对话框和用户注册的对话框同时消失，系统消息列表将会在页面中加载出来。

➤➤14.4.4 处理消息发布

用户登录系统后，可以发布消息，发布消息先单击“发布消息”超级链接，一旦用户单击了该超级链接，将出现“发布新消息”对话框，如图 14.5 所示。

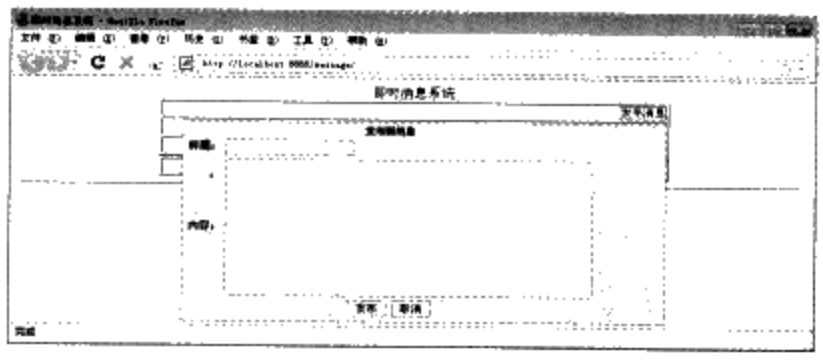


图 14.5 发布新消息

发布新消息的对话框可以选择关闭，如果用户单击“取消”按钮，发布消息的对话框将关闭。如果在标题文本框中输入消息标题，在消息内容中输入消息内容，然后单击“发布”按钮，将触发添加消息的函数，添加消息通过调用 createMessage()业务逻辑方法实现。下面是添加消息的 JavaScript 函数：

程序清单：codes\14\message\mm.js

```
//-------------用户发布新消息---------------
function addMsg()
{
    //如果用户已经登录
    if(curUser > 0)
    {
        //消息标题、消息内容不为空
        if (dwr.util.getValue("title") != null
            && dwr.util.getValue("title") != ""
            && dwr.util.getValue("content") != null
            && dwr.util.getValue("content") != "")
        {
            //调用 createMessage()方法来发布新消息
            mm.createMessage(dwr.util.getValue("title") ,
```

```
                dwr.util.getValue("content") , parseInt(curUser) , addMsgCb);
            dwr.util.setValue("title", "");
            dwr.util.setValue("content", "");
        }
        else
        {
            alert("请先输入消息标题和消息内容");
        }
    }
    else
    {
        alert("请先登录系统！");
    }
}
```

上面在添加消息时使用了 addMsgCb 回调函数，其代码如下：

程序清单：codes\14\message\mm.js

```
function addMsgCb(data)
{
    //如果发布新消息成功
    if (data > 0)
    {
        alert("添加消息成功");
    }
    else
    {
        alert("添加消息失败");
    }
    //隐藏发布新消息的对话框
    post.style.display='none';
}
```

如果没有特殊原因，发布新消息通常都应该成功，这样即可看到如图 14.6 所示的对话框。

一旦用户发布新消息成功，将可看到消息出现在当前页面中。这里存在一个问题：既然是即时消息系统，当然无须用户主动刷新页面，其他用户添加的消息也都应该在当前页面中“自动”出现。

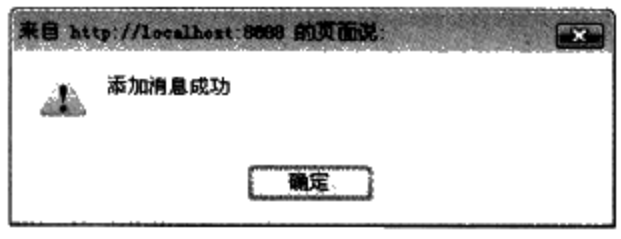


图 14.6　发布新消息成功

➤➤14.4.5　获取消息列表

为了让其他用户添加的消息自动出现在页面中，可以使用反向 Ajax 技术，这的确是反向 Ajax 大派用场的地方，但本应用并未使用 DWR 提供的反向 Ajax 功能。这是因为：使用反向 Ajax 就要在服务器端通过 Java 方法来操作 HTML 页面元素，而这一切都要通过 DWR 提供的 Util 类操作——实际上，Util 类操作 HTML 页面内容的能力十分有限，灵活性很差。例如，如果想填充一个表格内容，我们必须把要填充到表格中的数据定义成二维数组，用起来感觉灵活性不大。

即使不使用 DWR 的反向 Ajax 技术，本系统的显示页面一样可以获得最新的消息列表——通过定时的请求来实现。实际上，定时发送请求也是 DWR 实现反向 Ajax 的实质。

当然，获取消息列表的请求总需要第一次执行，此后就可定时执行。系统将在用户登录、注册后自动执行该请求。下面是发送请求获得消息列表的 JavaScript 函数：

程序清单：codes\14\message\mm.js

```
//----------获取系统全部消息列表----------
function getAllMsgByPage()
{
```

```
    mm.getAllMessageByPage(curPage , getMsgCb);
    //每 2 秒调用一次 getAllMsgByPage()方法来获取消息列表
    setTimeout("getAllMsgByPage();" , 2000);
}
```

该方法中包括了一个 getMsgCb()回调函数，该函数用于将服务器的响应添加到页面的表格中显示。其代码如下（该回调函数也是两个分页函数的回调函数）：

程序清单：codes\14\message\mm.js

```
//-----------获取消息列表的回调函数-----------
function getMsgCb(data)
{
    if (data == null && curPage != 1)
    {
        alert("后面已经没有记录了！");
        curPage--;
    }
    if (data != null)
    {
        var show = document.getElementById("show");
        //删除表格内全部内容
        dwr.util.removeAllRows("show");
        //遍历所有的消息列表
        for (var obj in data )
        {
            var row = document.createElement("tr");
            var title = data[obj]['title'];
            var owerName = data[obj]['owerName'];
            var msgId = data[obj]['id'];
            //添加单元格来显示消息发布者
            var cell = document.createElement("td");
            cell.innerHTML = owerName;
            row.appendChild(cell);
            //添加单元格来显示消息标题
            cell = document.createElement("td");
            cell.innerHTML = "<a href='#' onclick='getMsg("
                + msgId + ")'>" + title + "</a>";
            row.appendChild(cell);
            show.appendChild(row);
        }
    }
}
```

上面的代码中 getMsgCb()函数稍微有点复杂，它需要遍历服务器返回的对象数组，并根据该对象数组来创建表格。当程序获取系统中的所有消息之后，该回调函数即可将所有消息在页面中显示出来，如图 14.7 所示。

图 14.7　实时消息列表页

➤➤14.4.6　处理分页

本系统提供了分页功能，在配置文件中指定每页仅显示 2 条记录，这是为了在记录很少时也可看到分页效果。实际部署时，可将每页显示的消息条数设得更大。

分页功能涉及两个函数，一个用于发送“前一页”的请求，一个用于发送“后一页”的请求，两个请求函数共用了 getMsgCb()作为回调函数。下面是处理分页的两个请求函数的代码：

程序清单：codes\14\message\mm.js

```
//-------------上页函数-------------
function prePage()
{
    if (curPage <= 1)
    {
        alert("已经是第一页了，无法翻页");
    }
    else
    {
        //调用 getAllMessageByPage 方法来发送分页请求
        mm.getAllMessageByPage(--curPage , getMsgCb);
    }
    return false;
}
//-------------下页函数-------------
function nextPage()
{
    //调用 getAllMessageByPage 方法来发送分页请求
    mm.getAllMessageByPage(++curPage , getMsgCb);
    return false;
}
```

因为本系统并未保存消息的总页数，因此不可能在执行“后一页”请求时判断是否已经到了最后一页，所以只是发送了后一页请求，而 getMsgCb()回调函数会判断当前页是否已经为最后一页。如果 getMsgCb()发现下一页没有消息，将会提示当前页已经是最后一页了。

➤➤14.4.7 查看消息内容

查看消息内容通过单击消息标题实现，在获取消息列表的回调函数中，使用了如下代码：

```
cell.innerHTML = "<a href='#'onclick='getMsg(" + msgId + ")'>" + title + "</a>";
```

上面的表格内容为消息标题，在单击消息标题时将激发 getMsg()函数，并将消息 ID 作为参数传入。下面是 getMsg()函数的代码：

程序清单：codes\14\message\mm.js

```
//----------根据主键获取消息----------
function getMsg(data)
{
    //如果用户还未登录
    if (curUser > 0)
    {
        if (typeof data == 'number' && data > 0)
        {
            mm.getMessage(data , getMsgDetailCb);
        }
    }
    else
    {
        alert("请先登录系统！");
    }
    return false;
}
```

上面在调用服务器方法获得消息内容时，使用了 getMsgDetailCb()回调函数，该函数用于将服务器返回的 MessageBean 的内容显示在页面中，并通过对话框显示消息内容。其代码如下：

程序清单：codes\14\message\mm.js

```
//---------根据主键获取消息的回调函数-------
function getMsgDetailCb(data)
```

```
{
    if (data != null)
    {
        dwr.util.setValue('viewTitle' , data.title);
        dwr.util.setValue('viewAuthor' , data.owerName);
        dwr.util.setValue('viewContent' , data.content);
        showDialog(view);
    }
}
```

如果用户单击某条消息的标题，将可看到如图 14.8 所示的界面。

图 14.8　查看消息内容

➤➤14.4.8　页面加载函数

页面加载函数用于完成页面对话框的初始化，设置 DWR 的全局错误处理函数，以及验证用户是否登录等。该函数将会随页面加载而自动执行，其代码如下：

程序清单：codes\14\message\mm.js

```
function init()
{
    //初始化 3 个对话框元素
    login = $("login");
    post = $("post");
    view = $("view");
    //定义全局错误处理器
    dwr.engine.setErrorHandler(errHandler);
    mm.isLogin(initCb);
}
//指定页面加载时执行 init()函数
document.body.onload = init;
```

初始化函数中调用了 isLogin()业务逻辑方法来判断用户是否登录，并传入了 initCb()函数作为回调函数，该函数会根据 isLogin()方法的返回值来进行下一步处理，如果判断到用户还未登录，则显示登录、注册对话框，从而将看到如图 14.2 所示界面。

14.5　本章小结

本章开发了一个简单的即时消息系统，本系统采用了完善的轻量级 Java EE 应用架构，应用底层使用 Spring+Hibernate 整合，Spring 容器中的 Bean 对外提供业务逻辑功能。Spring 容器的 IoC 机制可实现 DAO 组件和业务逻辑组件的解耦，AOP 机制则负责为业务逻辑组件提供事务控制和权限控制。DWR 框架负责将 Spring 容器中的业务逻辑组件暴露给浏览器的 JavaScript 代码，使得浏览器端的 JavaScript 能以异步方式调用业务逻辑方法，并通过 DOM 操作将服务器响应动态加载出来。

除此之外，本章还介绍了基于 HibernateDaoSupport 来实现 DAO 组件，并详细介绍了如何扩展 HibernateDaoSupport 来实现分页查询。

➤➤本章练习

继续完善该即时消息系统，主要从如下三个方面进行完善：

- 完善系统界面，为其增加更多元素，使之更加美观。
- 为该系统增加类别管理，用户添加消息时应选择合适的类别。
- 为该系统增加消息回复，既可对消息主体进行回复，也可对消息回复进行回复。

CHAPTER 15

第15章 AjaxTags 框架详解

本章要点

- 了解 AjaxTags
- 下载和安装 AjaxTags
- 满足 AjaxTags 要求的服务器响应
- 使用 Servlet 生成响应
- 扩展 BaseAjaxServlet 生成响应
- 使用 AjaxXmlBuilder 帮助生成响应
- 常用 AjaxTags 标签的用法
- 了解 AjaxTags 的缺陷
- AjaxTags 的适用场景

AjaxTags 是由一系列的 JSP 标签组成，这些标签用以简化 Ajax 应用的开发。对很多开发者而言，开发 Ajax 应用时总是从创建 XMLHttpRequest 对象开始——这是相当糟糕的行为，你在重复发明轮子，实际上，那个轮子已经有人帮你做好了！通过前面的介绍，我们知道有两类选择，一种是使用 JavaScript 函数库来简化 Ajax 应用的开发，例如 Prototype.js 和 JQuery 库等；另外有一种是基于 RPC 的 Ajax 应用，例如使用 DWR 框架。

AjaxTags 则提供了更高层次的简化，AjaxTags 是以 Prototype 库和其扩展项目 Scriptaculous 库为基础的，它将常用的 Ajax 应用场景封装成简单的标签。通过使用 AjaxTags，Java EE 应用开发者可以像使用普通标签一样使用 AjaxTags，但这套标签库却可以为 Java EE 应用增加 Ajax 功能。虽然 AjaxTags 的灵活性相对较差，但对于大部分 Java EE 应用而言，常用的 Ajax 功能如自动完成、级联菜单等，AjaxTags 都有现成的实现，直接使用就可以了，没有理由拒绝 AjaxTags 的帮助。

15.1 AjaxTags 的下载和安装

AjaxTags 是属于 SourceForge 的一个开源项目，AjaxTags 以 Prototype 库及相关项目为基础，提供了一套简单的 JSP 标签库（早期的 AjaxTags 还需要 Struts 的支持，但今天的 AjaxTags 已经不再依赖于 Struts），通过这些标签库可非常简单的开发出 Ajax 应用。

➤➤15.1.1 什么是 AjaxTags

AjaxTags 是一套简单的 JSP 标签，这套标签以一些现有的开源项目为基础，AjaxTags 最核心的依赖是 Prototype.js 库，以及其相关项目：Scriptaculous.js。当然，JSTL 和 jakarta-commons 的一些包也是 AjaxTags 所必须的。除此之外，早期的 AjaxTags 还依赖于 Struts 框架，如果需要用 AjaxTags 的 displayTag 标签，则还需要 DisplayTag 框架。

AjaxTags 致力于解决 Java EE 应用开发者的软肋，厌倦了繁琐的 JavaScript 代码：赤裸裸地创建 XMLHttpRequest 对象，发送异步请求，当服务器响应到来之后，又要使用 DOM 操作来加载服务器响应，大量的 JavaScript 开发、调试，让人无需烦闷。

如果选择了 AjaxTags 标签库，开发者无需书写任何 JavaScript 代码，它以一种“非常 Java 的方式”来开发 Ajax 应用，开发者只需要提供满足 AjaxTags 所需格式的响应数据，然后在 JSP 页面中使用 AjaxTags 的标签——即使开发者不懂 JavaScript、不懂 DOM 操作也行。只要开发者能使用简单的 JSP 标签，开发者就可以使用 AjaxTags。它提供了一组简单的标签，这些标签封装了常用的 AjaxTags 功能。

因为 AjaxTags 是基于 JSP 标签库的，所以 AjaxTags 只能在 Java EE 应用中使用。虽然 Ajax 技术本身不局限于任何语言，但 AjaxTags 则只能用于 Java 应用。

➤➤15.1.2 下载和安装 AjaxTags

AjaxTags 是属于 SourceForge 的一个开源项目，登陆起 AjaxTags 的官方站点：http://ajaxtags.sourceforge.net/下载 AjaxTags 的最新版，目前 AjaxTags 的最新版本是 ajaxtags-1.3-beta-rc7，这个是 AjaxTags 目前较为稳定的产品版。

下载 AjaxTags 时有如下两个选项：

- Ajax Tag Library：此选项是 AjaxTags 项目发布文件。该选项对应的 zip 文件必须下载。
- Ajax Tags Demo：此选项是 AjaxTags 标签库的使用范例。下载该选项将得到 AjaxTags 示范应用。

将下载到的压缩文件解压缩，看到如下文件结构：

- docs：该路径下存放 AjaxTags 的各种文档，但没有包含 API 文档，AjaxTags 的 API 文档需要另行下载。
- lib：该路径下存放了编译和运行 AjaxTags 所依赖的第三方类库。用户可根据需要选择使用。

- web：该路径下存放了 AjaxTags 标签库所必须的各种图标文件、CSS 样式单和 JavaScript 库等。因此该路径下又包含 3 个路径：
- css：该路径下包含了使用 AjaxTags 必需的 CSS 样式单，这些样式单文件用于生成通用的 Ajax 效果。当然，用户也可以使用自己喜欢的 CSS 样式单。
- images：该路径包含使用 AjaxTags 必需的图标文件，这些图标文件用于生成通用的 Ajax 效果。当然，用户也可以使用自己喜欢的图片。
- js：该路径下包含了使用 AjaxTags 必须的 JavaScript 库文件，例如 prototype.js 等文件都可以在该路径下找到。
- ajaxtags-1.3-beta-rc7.jar：AjaxTags 的二进制类库。
- 其他授权等相关文档。

需要在一个 Web 应用中使用 AjaxTags，不再像传统的开源框架，只需要其 JAR 文件复制到 Web 应用的 WEB-INF\lib 下就可以的。因为 AjaxTags 是一个专注表现层的技术，因此，一个 Web 应用中如果需要使用 AjaxTags，则需要更多的东西。要在一个 Web 应用中使用 AjaxTags 请按如下步骤进行：

（1）将 AjaxTags 项目里 ajaxtags-1.3-beta-rc7.jar 文件复制到 Web 应用的 WEB-INF/lib 下。

（2）因为 AjaxTags 还需要第三方的类库的支持，至少需要 standard.jar、commons-lang-2.3.jar 两个文件复制到 Web 应用的 WEB-INF/lib 路径下。

（3）将 AjaxTags 项目解压缩路径下的 js 整个路径复制到 Web 应用的根路径下。当然，也可以放在 Web 应用里的任意路径，只要在 JSP 页面中能导入这些 JavaScript 文件即可。通常 AjaxTags 依赖于如下几个关键的 JavaScript 库。

- prototype.js 库。
- scriptaculous.js 库。
- overlibmws.js 库。
- ajaxtags.js、ajaxtags_controls.jsajaxtags_parser.js 库。

因此需要使用 AjaxTags 的 JSP 页面必须导入上面的 6 个 JavaScript 库。

（4）将 AjaxTags 项目解压缩路径下的 images 文件夹复制到 Web 应用的根路径下。当然，也可以放在 Web 应用的任意路径，只要在 JSP 页面中能访问该路径下的图片文件即可。如果程序开发者决定使用自己的图片，则可以不需要该步骤。但开发者则应该自己提供相应的图片，例如常用的 loading 图标。

（5）将 AjaxTags 项目解压缩路径下的 css 整个路径复制到 Web 应用根路径下。当然也可以放在 Web 应用的任意路径，理由与 images 路径相似。同样，开发也可以选择放弃 AjaxTags 默认的 CSS 样式，而是使用自己的 CSS 样式。

经过上面步骤，我们就可以在一个 JSP 页面中使用 AjaxTags 标签了，这下标签可以极大地简化 Ajax 应用的开发。因为我们需要使用 JSP 标签库，因此使用 AjaxTags 的页面不能是静态 HTML 页面，必须是动态的 JSP 页面。

注意：

因为只有在 JSP 页面中才可使用 JSP 标签库，因此使用 AjaxTags 标签的页面不能是静态 HTML 页面，必须是动态的 JSP 页面。

当然，即使有 AjaxTags 的帮助，服务器的响应还是必须由开发者提供。下面我们介绍如何开发服务器响应类。

15.2 AjaxTags 入门

前面我们已经介绍了 Ajax 的使用是相当简单的事情，完全不需要繁琐的创建 XMLHttpRequest 对

象，发送 Ajax 请求、也不需要编写回调函数，不需要解析服务器响应。整个 Ajax 交互过程完全由 AjaxTags 完成，应用开发者只需要编写服务器响应即可。服务器响应由处理请求的处理类负责生成。

➤➤15.2.1 编写处理类

这里说的处理类并不一定是一个完整的 Java 类，它可以是一个 JSP 页面，也可以是一个配置在 Web 应用中的 Servlet，也可以是 Struts 的 Action，更甚至是一个非 Java 的服务器组件。唯一的要求是它能响应用户的请求就可。当然，因为 AjaxTags 是一种高度封装的 Ajax 框架，因此处理类的返回值不能像之前那样随心所欲，处理类的返回值必须满足某种格式，服务器处理类的返回值必须满足如下 XML 文件格式：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- AjaxTags 服务器响应的根元素 -->
<ajax-response>
    <!-- AjaxTags 服务器响应的内容必须在 response 里 -->
    <response>
        <!-- 下面 item 节点，分别用于不同的选择 -->
        <item>
            <name>Record 1</name>
            <value>Val1</value>
        </item>
        <item>
            <name>Record 2</name>
            <value>Val2</value>
        </item>
        ...
    </response>
</ajax-response>
```

从上面代码可以看出，AjaxTags 要求的 XML 响应必须处于<ajax-response.../>元素的</response.../>子元素里，每个<item.../>元素负责生成一项响应。

AjaxTags 也允许使用普通文本响应，普通文本响应则应满足如下格式：

```
#普通文本响应的示范
#每项响应对应一行，每行的前面部分是 name，后面是 value，
#中间以英文逗号（,）隔开。
Record 1,Val1
Record 2,Val2
...
```

下面介绍的一个简单应用以自动完成应用为范例，建立一个对应的处理类，处理类以 JSP 来代替。下面是自动完成的处理 JSP 页面代码。这是一个简单的级联菜单应用，用户一旦选中第一个下拉列表框的某个选项值，下一个下拉列表框将随之变化。处理器类由一个 JSP 页面充当，该页面负责生成一个 XML 响应，而 XML 相应的格式则符合上面的格式。下面是该处理器页面的代码：

程序清单：codes\15\15-2\firstTags\res.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%@ page contentType="text/html; charset=GBK" language="java" %>
<%@ page import="java.util.*"%>
<%
//获取请求参数
int country = Integer.parseInt(request.getParameter("country"));
//设置响应内容和所使用的字符集
response.setContentType("text/xml; charset=UTF-8");
//控制响应不会在客户端缓存
response.setHeader("Cache-Control", "no-cache");
```

```
//用于控制服务器的响应
List<String> cityList = new ArrayList<String>();
//根据请求参数 country 来控制服务器响应
switch(country)
{
    //对于选择下拉框的“中国”选项
    case 1:
        cityList.add("广州");
        cityList.add("深圳");
        cityList.add("上海");
        break;
    //对于选择下拉框的“美国”选项
    case 2:
        cityList.add("华盛顿");
        cityList.add("纽约");
        cityList.add("加洲");
        break;
    //对于选择下拉框的“日本”选项
    case 3:
        cityList.add("东京");
        cityList.add("大阪");
        cityList.add("福冈");
        break;
}
%>
<ajax-response>
<response>
<%
//遍历集合，依次将城市添加到服务器响应的 item 项里。
for(String city : cityList)
{
%>
    <item>
        <name><%=city%></name>
        <value><%=city%></value>
    </item>
<%}%>
</response>
</ajax-response>
```

上面 JSP 页面的响应并不是 HTML 页面，而是生成一个 XML 页面，这个 XML 页面正好能满足 AjaxTags 标签的响应要求。该页面根据请求参数，依次将三个城市添加到 cityList 集合里，然后遍历该集合来生成一个满足要求的 XML 页面。

一旦服务器提供了满足要求的 XML 响应，就可以在 JSP 页面里使用标签来提供 Ajax 功能。

➤➤15.2.2 使用标签

在客户端页面使用 AjaxTags 的标签来生成 Ajax 应用，在客户端页面使用 AjaxTags 标签非常简单，而且“非常 Java”，开发者几乎感觉不到使用了 Ajax 的功能，实际上该页面已经具有了 Ajax 能力。在 JSP 页面使用 AjaxTags 应按如下步骤进行：

（1）在 JSP 页面中使用 taglib 指令导入 AjaxTags 标签库。

（2）在 JSP 页面中导入 AjaxTags 必需的 JavaScript 库、CSS 样式单等。

（3）使用 AjaxTags 标签。

下面使用 AjaxTags 的 select 标签的 JSP 页面代码：

程序清单：codes\15\15-2\firstTags\select.jsp

```
<!-- 导入 AjaxTags 标签库 -->
<%@ taglib uri="http://ajaxtags.org/tags/ajax" prefix="ajax"%>
<%@ page contentType="text/html;charset=GBK"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> AjaxTags 入门 </title>
    <!-- 导入 AjaxTags 所必须的 JavaScript 库 -->
    <script type="text/javascript" src="js/prototype.js"></script>
    <script type="text/javascript"
        src="js/scriptaculous/scriptaculous.js"></script>
    <script type="text/javascript"
        src="js/overlibmws/overlibmws.js"></script>
    <script type="text/javascript"
        src="js/ajax/ajaxtags.js"></script>
    <script type="text/javascript"
        src="js/ajax/ajaxtags_controls.js"></script>
    <script type="text/javascript"
        src="js/ajax/ajaxtags_parser.js"></script>
    <!-- 导入 AjaxTags 所必须的 CSS 样式单 -->
    <link rel="stylesheet" type="text/css" href="css/ajaxtags.css" />
</head>
<body>
<body>
国家:<select id="country" name="country">
    <option value="">选择一个国家</option>
    <option value="1">中国</option>
    <option value="2">美国</option>
    <option value="3">日本</option>
</select>
城市:<select id="city" name="city">
    <option value="">城市列表</option>
</select>
<!-- 使用 AjaxTags 的 select 标签 -->
<ajax:select
    baseUrl="res.jsp"
    source="country"
    target="city"
    parameters="country={country}" />
</body>
</html>
```

上面页面的第一段粗体字代码导入了 AjaxTags 标签库，指定标签库的 URI 和 prefix 等。第二段粗体字代码是使用 AjaxTaga 标签必须的导入的 JavaScript 库和 CSS 样式单文件。

上面的 JSP 页面除了导入了几个 JavaScript 库之外，完全不需要任何 JavaScript 代码，丝毫感受不到使用 Ajax 的痕迹，但因为使用了 AjaxTags 的 select 标签（该标签用于生成级联菜单的 Ajax 应用）。该页面也具有了级联菜单的 Ajax 功能。图 15.1 显示使用 AjaxTags 后该页面的级联菜单效果。

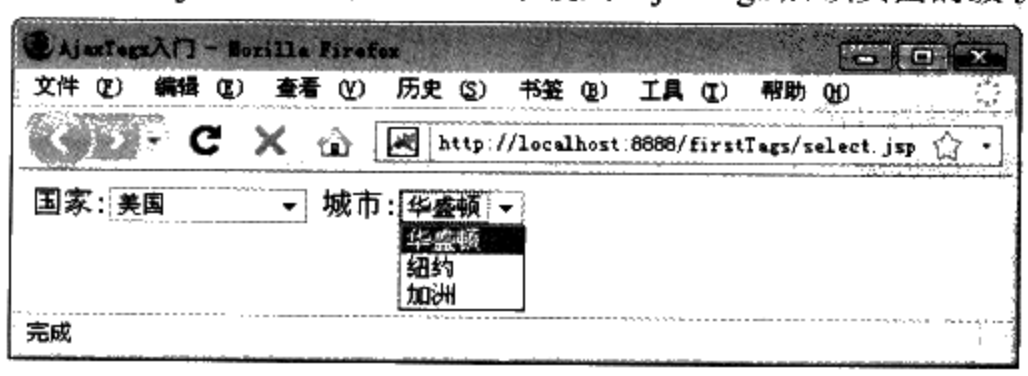


图 15.1　使用 AjaxTags 级联菜单的效果

通过上面示例可以看出，使用 AjaxTags 来开发 Ajax 应用是多么简单。

注意：

因为有些浏览器中在处理以 GET 方式发送的请求时存在一些问题，因此笔者修改了 ajaxtags.js 文件中的请求发送方式。关于请求的发送方式，笔者更倾向于使用 POST 请求，而 AjaxTags 默认的请求发送方式都是 GET 方式的。只需打开 ajaxtags.js 文件，将文件中 method: 'get'替换成 method: 'post'，这样可以改变 AjaxTags 的请求发送方式。

15.3 处理类的几种形式

前面已经介绍过了，AjaxTags 的处理类并不一定是一个真正的 Java 类，它可以有很多形式，如 JSP 页面、Servlet 等。甚至可以是一个非 Java 文件，唯一的要求是该处理类能返回一个满足 AjaxTags 格式要求的 XML 响应或文本响应。

为了简化生成满足要求的响应，AjaxTags 还提供了几个工具类，这些工具类可以简化处理类的实现，更方便地生成满足 AjaxTags 要求的响应。

下面依次介绍处理类的几种形式，介绍这几种形式时都以前面介绍的级联菜单应用为基础，应用使用 AjaxTags 标签的 JSP 页面没有太大改变，仅仅改变了 ajax:select 标签的 baseUrl 属性——这是因为采用不同处理类时，处理类在服务器端配置的 URL 不同。

➤➤15.3.1 使用普通 Servlet 生成响应

使用普通的 Servlet 来生成响应与前面介绍的 JSP 响应是类似的，因为 JSP 的实质就是 Servlet，一旦完成了 Servlet 的配置，可以将 baseURL 指定为该 Servlet 的 URL 地址，将请求将该 Servlet 发送，该 Servlet 负责生成 XML 响应，从而完成整个 Ajax 交互。下面负责响应的 Servlet 的源代码：

程序清单：codes\15\15-3\Servlet\WEB-INF\src\lee\SelectServlet.java

```
public class SelectServlet extends HttpServlet
{
    //普通 Servlet 的服务响应方法
    public void service(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException,IOException
    {
        //获得请求参数，即国家值
        int country = Integer.parseInt(request.getParameter("country"));
        //用于控制服务器的响应
        List<String> cityList = new ArrayList<String>();
        switch(country)
        {
            //对于选择国家下拉框的"中国"选项
            case 1:
                cityList.add("广州");
                cityList.add("深圳");
                cityList.add("上海");
                break;
            //对于选择国家下拉框的"美国"选项
            case 2:
                cityList.add("华盛顿");
                cityList.add("纽约");
                cityList.add("加洲");
                break;
            //对于选择国家下拉框的"日本"选项
            case 3:
```

```
            cityList.add("东京");
            cityList.add("大阪");
            cityList.add("福冈");
            break;
        }
        //该类用于辅助生成 XML 响应
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
        for (String city : cityList )
        {
            builder = builder.addItem(city , city);
        }
        System.out.println(builder);
        //设置响应的页面头
        response.setContentType("text/xml; charset=UTF-8");
        PrintWriter out =  response.getWriter();
        out.println(builder.toString());
    }
}
```

该 Servlet 的代码与 JSP 代码非常类似，它们都先获取请求参数 country，并根据请求参数值不同，生成包含三个城市的 List 对象。值得注意的是，Servlet 的代码并不需要像在 JSP 页面里一样，自己来控制输出<ajax-response>、<response>、<name>和<value>等标签，这系列的过程是通过 AjaxTags 提供的一个辅助类完成的，这个辅助类为 AjaxXmlBuilder。借助于 AjaxXmlBuilder 的帮助，程序开发者可以更便捷地生成 AjaxTags 所需格式的响应。关于 AjaxXmlBuilder 在下一接介绍。

编译该 Servlet，然后在 web.xml 文件中增加如下配置，增加了如下配置后，该 Servlet 就能处理客户端请求了。在 web.xml 文件中增加的配置片段如下：

程序清单：codes\15\15-3\Servlet\WEB-INF\web.xml

```
<!-- 配置 Servlet -->
<servlet>
    <!-- 配置该 Servlet 的名字 -->
    <servlet-name>select</servlet-name>
    <!-- 配置该 Servlet 的实现类 -->
    <servlet-class>lee.SelectServlet</servlet-class>
</servlet>
<!-- 配置 Servlet 映射的 URL -->
<servlet-mapping>
    <!-- 指定该 Servlet 的名字 -->
    <servlet-name>select</servlet-name>
    <!-- 指定该 Servlet 映射的 URL 地址 -->
    <url-pattern>/select</url-pattern>
</servlet-mapping>
```

一旦完成了该 Servlet 的映射，该 Servlet 就能在 select 的 URL 处提供响应。因此，只需要在使用 AjaxTags 标签的页面中通过如下代码来使用标签。

```
<ajax:select
    baseUrl="select"
    source="country"
    target="city"
    parameters="country={country}"/>
```

与之前直接使用 JSP 作为响应对比，修改了 ajax:select 标签的 baseUrl 属性，baseUrl 属性是 Ajax 请求发送的服务器 URL，此处改为 select，即表示向 SelectServlet 发送请求。关于 AjaxTags 标签的各属性的详细介绍，将在后面介绍。

➤➤15.3.2 使用 AjaxXmlBuilder 辅助类

AjaxXmlBuilder 是个工具类，它包含了一些工具方法，这些工具方法根据字符串、集合来生成满足 AjaxTags 要求的响应。借助于 AjaxXmlBuilder 的帮助，程序开发者无需手动输出<ajax-response>、<response>、<name>和<value>等标签，只需调用 AjaxXmlBuilder 的工具方法，它会自动生成这些标签，并将字符串或集合里的对象的值增加到该响应里。AjaxXmlBuilder 对象主要包含如下两个方法：

- addItem(String name, java. value)：每调用一次，为响应增加一个 item 节点，其中第一个参数就是 item 节点下 name 节点的值，第二个参数就是 item 节点下 value 节点的值。
- addItems(Collection collection, String nameProp, String valProp)：该方法遍历 collection 集合里的元素，collection 里的元素必须是对象，且必须包含 nameProp 和 valProp 两个属性。该方法为每个集合元素创建一个 item 节点，其中集合元素的 nameProp 属性值将作为 item 节点下 name 子节点的值；valProp 属性值将作为 item 节点下 value 子节点的值。

通过这两个方法，可以非常便捷地生成 AjaxTags 所需格式的响应。看下面的简单代码：

程序清单：codes\15\15.3\AjaxXmlBuilder\src\lee\Test.java

```
public class Test
{
    public static void main(String[] args)
    {
        //创建一个默认的 AjaxXmlBuilder 实例
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
        //采用循环依次为为响应添加 5 个 item 节点
        for (int i = 0 ; i < 5 ; i++ )
        {
            builder.addItem("name 值" + i , "value 值" + i);
        }
        //打印出 builder 本身所生成的 XML 响应
        System.out.println(builder.toString());
    }
}
```

该文件将生成一个满足 AjaxTags 响应格式的 XML 文件，该文件的的 ajax-response、response 节点都会默认包含，该文件包含了 5 个 item 节点。下面是该程序的输出结果：

```
<?xml version="1.0" encoding="UTF-8" ?>
<ajax-response>
    <response>
        <item>
            <name>name 值 0</name>
            <value>value 值 0</value>
        </item>
        <item>
            <name>name 值 1</name>
            <value>value 值 1</value>
        </item>
        <item>
            <name>name 值 2</name>
            <value>value 值 2</value>
        </item>
        <item>
            <name>name 值 3</name>
            <value>value 值 3</value>
        </item>
        <item>
            <name>name 值 4</name>
```

```
            <value>value 值 4</value>
        </item>
    </response>
</ajax-response>
```

提 示：

提示：上面的 XML 响应是笔者进行格式化后的效果，真实输出的 XML 响应这么美观，AjaxXmlBuilder 本身所生成的 XML 响应没有换行，没有缩进，只是系列的 XML 元素和内容。

AjaxXmlBuilder 除了可使用 addItem()方法来依次添加 item 节点之外，还可以 addItems()方法一次添加多个节点，该方法会自动遍历指定集合，为每个集合元素创建一个 item 节点。看下面的示例程序：

程序清单：codes\15\15.3\AjaxXmlBuilder\src\lee\Test2.java

```
public class Test2
{
    public static void main(String[] args)throws Exception
    {
        //构造一个集合对象
        List<Person> pl = new ArrayList<Person>();
        //向集合添加三个元素，每个元素都是 Person 实例
        pl.add(new Person("Jack" , "男"));
        pl.add(new Person("Rose" , "女"));
        pl.add(new Person("小强" , "男"));
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
        //调用 addItems()遍历 pl 集合元素，每个集合元素创建一个 item 节点
        builder.addItems(pl , "name" , "gender");
        //输出生成的 XML 响应
        System.out.println(builder.toString());
    }
}
```

该代码使用了 addItems()方法为 XML 响应添加三个 item 节点。每个节点都包含 name 和 value 子节点，name 节点的值为 Person 对象的 name 属性，而 value 节点的值为 Person 对象 gender 属性——前提是集合元素有 name 属性和 gender 属性。下面是 Person 类代码：

程序清单：codes\15\15.3\AjaxXmlBuilder\src\lee\Person.java

```
public class Person
{
    //name 属性
    private String name;
    //gender 属性
    private String gender;
    //无参数的构造器
    public Person(){}
    //初始化全部属性的构造器
    public Person(String name , String gender)
    {
        this.name = name;
        this.gender = gender;
    }
    //省略 name 和 gender 的 setter 和 getter 方法
    ...
}
```

运行 Test2 程序，将可看到如下输出：

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

```
    <response>
        <item>
            <name>Jack</name>
            <value>男</value>
        </item>
        <item>
            <name>Rose</name>
            <value>女</value>
        </item>
        <item>
            <name>小强</name>
            <value>男</value>
        </item>
    </response>
</ajax-response>
```

通过使用 AjaxXmlBuilder 的 builder.addItems(pl , "name" , "gender");方法，该 builder 将自动遍历 pl 集合中的元素，每个元素对应一个 item 节点。集合元素的 name 属性为每个 item 节点的 name 节点值，gender 属性为每个 item 节点的 value 节点值。

➤➤15.3.3 使用 BaseAjaxServlet 生成响应

这种方式是对使用 Servlet 作为响应的一种改进，BaseAjaxServlet 是 HttpServlet 的一个子类，因此该类可以直接配置在 web.xml 文件中来响应客户端请求。BaseAjaxServlet 也是个抽象类，但它已经重写了 HttpServlet 的 doPost()和 doGet()方法，程序员无需实现这两个方法，这两个方法用于对客户端提供响应。

BaseAjaxServlet 的子类也必须重写一个回调方法，getXmlContent(HttpServletRequest request, HttpServletResponse response)方法，该方法的返回一个 XML 字符串，该字符串被作为响应输出到浏览器。

程序清单：codes\15\15.3\BaseAjaxServlet\WEB-INF\src\lee\SelectServlet.java

```
//用于响应 Ajax 请求的 Servlet，继承 BaseAjaxServlet
public class SelectServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，该方法的返回值将作为 Ajax 请求的响应
    public String getXmlContent(HttpServletRequest request,
        HttpServletResponse response)
        throws Exception
    {
        //获取请求参数
        int country = Integer.parseInt(request.getParameter("country"));
        //初始化需要集合
        List<String> cityList = new ArrayList<String>();
        //针对不同的请求参数，采用不同的城市添加到集合中
        switch(country)
        {
            //对于选择国家下拉框的"中国"选项
            case 1:
                cityList.add("广州");
                cityList.add("深圳");
                cityList.add("上海");
                break;
            //对于选择国家下拉框的"美国"选项
            case 2:
                cityList.add("华盛顿");
                cityList.add("纽约");
                cityList.add("加洲");
                break;
            //对于选择国家下拉框的"日本"选项
```

```
            case 3:
                cityList.add("东京");
                cityList.add("大阪");
                cityList.add("福冈");
                break;
        }
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
        //遍历集合，将集合中的元素添加为 item 节点的子节点 name 和 value 的值。
        for (String city : cityList )
        {
            builder = builder.addItem(city , city);
        }
        //返回一个 XML 字符串。
        return builder.toString();
    }
}
```

这个 Servlet 虽然没有重写 doGet 和 doPost 方法，这也正是 BaseAjaxServlet 的用处，避免书写繁琐的 doGet 和 doPost 方法，而且无需手动获取输出流，只要返回一个 XML 字符串即可。而 BaseAjaxServlet 则使用 getXmlContent()的返回值作为响应。

BaseAjaxServlet 的子类依然是一个标准 Servlet，因此在 web.xml 文件中配置该 Servlet，使用该 Servlet 为 Ajax 请求提供响应等与普通 Servlet 的用法完全一样。

与使用普通 Servlet 相比，使用 BaseAjaxServlet 有如下几个方便之处：

- 重写 getXmlContent()方法可声明抛出 Exception 异常，因此在 getXmlContent()方法中处理异常更灵活。
- 继承 BaseAjaxServlet 来提供 Servlet 无需重写 doGet()、doPost()等方法，重写 getXmlContent()方法即可。

➤➤15.3.4 使用非 Java 响应

前面已经介绍了，AjaxTags 标签并不一定要求服务器采用 Servlet，或者 JSP 作为响应，甚至允许使用非 Java 的响应，例如直接使用静态的 XML 文件作为响应。虽然这种场景的应用不是特别广泛，但也不是完全没有用处，下面介绍使用静态的 XML 文件作为响应的情形。下面是静态 XML 文件的源代码：

程序清单：codes\15\15.3\non-java\res.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ajax-response>
    <response>
        <item>
            <name>广州</name>
            <value>广州</value>
        </item>
        <item>
            <name>深圳</name>
            <value>深圳</value>
        </item>
        <item>
            <name>上海</name>
            <value>上海</value>
        </item>
    </response>
</ajax-response>
```

读者可能已经看出来了：这个静态 XML 文件是当 country 请求参数为 1 时，上面的 Servlet 的响

应。没错，如果使用这个静态的 XML 文件作为相应，则不管在客户端选择哪个国家，都将输出中国的三个城市。虽然这并不太符合实际情况，但如果我们使用静态的响应，但由于静态响应完全不管浏览器发送怎样的请求参数，服务器总是生成完全相同的响应。

提 示:

因为上面的 XML 文件的声明部分指定了 encoding="UTF-8"，也就是这份 XML 文件应该使用 UTF-8 的字符集进行编码，所以保存这份文件时候应该选择使用 UTF-8 的字符集。

假设该文件的文件名为 res.xml，并将该文件放置在与 select.jsp 页面同一个路径下，然后在 select.jsp 页面中可通过如下标签来完成 Ajax 交互。

```
<ajax:select
    baseUrl="res.xml"
    source="country"
    target="city"
    parameters="country={country}"/>
```

上面 Ajax select 标签的 baseUrl 改为 res.xml，表示直接向静态的 XML 文件请求。不管请求参数是什么，请求总是得到相同的响应。图 15.2 显示了这种情况。

图 15.2 使用静态 XML 文件作为响应

15.4 使用 AjaxTags 标签

AjaxTags 提供的标签并不是特别多，但这些标签都用于解决常用的 Ajax 场景，例如自动完成，级联菜单等。使用 AjaxTags 的标签，可以让我们从繁琐的 JavaScript 处理中抽身而出，以非常 Java 的方式，优雅地完成 Ajax 应用。

➤➤15.4.1 使用自动完成标签

自动完成标签的完成的功能是类似于 Internet Explorer 中的文本框具有的功能，系统可以根据用户的输入，提示自动完成的选择。

假设输入城市，如果用户输入“广”，系统读取数据库，对比数据库的记录，找出所有以“广”开头的城市，例如“广岛”和“广州”，从而提供给用户选择。这种自动完成的 Ajax 应用，在第 10 章已经介绍过了。在那种示例应用中，大约包含 200 多行的 JavaScript 代码，下面我们以 AjaxTags 来完成这个 Ajax 应用。自动完成使用 ajax:autocomplete 标签，该标签有如下几个属性：

- var：这是一个可选属性，该属性定义了 autocomplete 标签创建的 JavaScript 对象名。通常无需指定该属性。
- attachTo：这是一个可选属性，该属性定义了前面的 var 对应的自动完成对象将应用到的对象。
- baseUrl：这是一个必需属性，该属性定义了 Ajax 发送请求的目标 URL。该属性指定的 URL 将返回一个典型的 AjaxTags 所需要的 XML 响应，响应的每个 item 节点的 name 值就是自动完成的提供给用户选择的一项。该属性支持表达式语言。

- source：这是一个必需属性，该属性指定的 HTML 元素的值一旦发生改变，将发送 Ajax 请求。通常，该属性的文本将作为自动完成的前缀部分。即，source 元素指定的 HTML 元素里的文本将被作为请求参数，伴随着 Ajax 请求一同发送，一旦该请求参数发送到服务器的 baseUrl，baseUrl 处的服务器响应将返回一个满足条件的 XML 响应。当然，也可以指定其他的请求参数。
- target：这是一个必需属性，该属性指定一个文本域，该文本框将显示自动完成的选择项对应的 value。如果用户无需使用额外的文本框来显示 value，则可将该参数设置为与 source 相同。
- parameters：这是一个必需属性，该属性指定了 Ajax 请求的请求参数。
- className：这是一个必需属性，该属性指定了自动完成所提供的下拉框的 CSS 样式单的名字。通常系统提供了该 CSS 样式单，但用户可以自定义自己的 CSS 样式单。
- indicator：这是一个可选属性，该属性指定一个 HTML 元素，该元素在该 Ajax 请求开始时出现，随着 Ajax 交互完成而隐藏。该元素可以通知用户 Ajax 交互的进度。
- minimumCharacters：这是一个可选属性，该属性指定自动完成最少所需的字符数。假设 source 指定一个文本框，如果 minimumCharacters 的属性值为 2，则至少要求 source 指定的文本框提供了 2 个字符的输入，系统才会提供自动完成的功能。
- appendSeparator：这是一个可选属性，一旦设置了该属性，target 属性指定的文本框的值不会被覆盖，而是在后面添加上自动完成的 value 节点的值。添加时将以 appendSeparator 属性指定的字符串作为分隔符。
- preFunction：这是一个可选属性，该属性指定了 Ajax 交互之前的自动执行的函数。
- postFunction ：这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。
- errorFunction：这是一个可选属性，该参数指定一个函数，该函数当服务器响应出错时被调用。
- parser：这是一个可选属性，该参数指定服务器响应的解析器，通常该解析器无需指定。除非用户需要自己完成特别的工作。默认解析器是 ResponseHtmlParser。

提 示：

AjaxTags 的很多标签都支持指定 preFunction、postFunction、errorFunction 三个属性，而且三个属性所指定函数的意义也完全相同，故以后介绍其他标签时将不再列出这这个属性。

为了更好地演示 Ajax 应用，首先定义了 Book 类，它是一个标准的 VO（Value Object，值对象），每个 Book 实例代表一本图书。Book 里包含了 name 和 publisher 两个属性。下面是该 Book 类的代码：

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\Book.java

```
public class Book implements Serializable
{
    //定义 name 属性
    private String name;
    //定义 publisher 属性
    private String publisher;
    //无参数的构造器
    public Book()        {}
    //初始化全部属性的构造器
    public Book(String name , String publisher)
    {
        this.name = name;
        this.publisher = publisher;
    }
    //省略了 name 和 publisher 的 setter 和 getter 方法
    ...
    //利用 ToStringBuilder 来重写 toString()方法
    public String toString()
    {
```

```
        return new ToStringBuilder(this)
            .append("书名：", name)
            .append("出版社：", publisher).toString();
    }
}
```

为了简单起见，我们不再使用复杂的持久层组件，不使用数据库来保存数据，不使用 DAO 组件来完成数据库访问。而是将所有的数据以静态属性的方式保存在业务逻辑组件中，业务逻辑组件不再需要依赖持久层组件。业务逻辑组件直接访问静态属性，从而提供业务逻辑的实现。考虑到后面的 AjaxTags 标签，此处的 BookService 组件包含了多个业务逻辑方法。下面是 BookService 类的源代码：

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\BookService.java

```
public class BookService
{
    static final List<Book> books = new ArrayList<Book>();
    //初始化一些图书作为试验数据
    static
    {
        books.add(new Book("疯狂 Java 讲义", "电子工业出版社"));
        books.add(new Book("轻量级 Java EE 企业应用实战", "电子工业出版社"));
        books.add(new Book("疯狂 Ajax 讲义", "电子工业出版社"));
        books.add(new Book("疯狂 XML 讲义", "电子工业出版社"));
        books.add(new Book("疯狂 Workflow 讲义", "电子工业出版社"));
        books.add(new Book("软件工程导论", "清华大学出版社"));
        books.add(new Book("Java 教程", "清华大学出版社"));
        books.add(new Book("Hibernate 持久化", "清华大学出版社"));
        books.add(new Book("Java 动画设计", "清华大学出版社"));
        books.add(new Book("Java 编程思想", "机械工业出版社"));
        books.add(new Book("Java 高级程序设计", "机械工业出版社"));
        books.add(new Book("Spring 项目指南", "机械工业出版社"));
        books.add(new Book("Java 项目指南", "机械工业出版社"));
    }
    /**
     * 根据出版社查询所出版的书
     * @param publisher 出版社。
     * @return 该出版社所出的全部书籍
     */
    public List<Book> getBooksByPublisher(String publisher)
    {
        List<Book> result = new ArrayList<Book>();
        System.out.println("出版社:" + publisher);
        for ( Book book : books)
        {
            if (book.getPublisher().equalsIgnoreCase(publisher))
            {
                result.add(book);
            }
        }
        return result;
    }
    /**
     * 根据书名前缀返回以该前缀开始的全部书籍
     * @param prefix 书名前缀。
     * @return 所有以 prefix 开头的书籍
     */
    public List<Book> getBooksByPrefix(String prefix)
    {
        List<Book> result = new ArrayList<Book>();
```

```
        for (Book book : books)
        {
            if (book.getName().toLowerCase()
                .startsWith(prefix.toLowerCase()))
            {
                result.add(book);
            }
        }
        return result;
    }
    /**
     * 返回全部的书籍
     * @return 所有书籍
     */
    public List<Book> getAllBooks()
    {
        return books;
    }
}
```

这个 BookService 业务逻辑组件将作为本节所有示例程序的业务逻辑组件。本示例程序在添加书籍实例时，并未考虑书籍的真实性，只是随便添加，作为应用的示范使用，请读者不要误会。本业务逻辑组件将所有的持久层数据作为组件属性保存，并未真正从数据库读取。下面是自动完成的 Servlet，该 Servlet 基于 BaseAjaxServlet 完成，该 Servlet 的代码如下：

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\BookService.java

```
public class AutocompleteServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，该方法的返回值作为 Ajax 请求的响应
    public String getXmlContent(HttpServletRequest request,
        HttpServletResponse response) throws Exception
    {
        //获得请求参数
        String prefix = request.getParameter("prefix");
        //创建业务逻辑组件的实例
        BookService service = new BookService();
        //返回以特定前缀开始的所有的书籍
        List list = service.getBooksByPrefix(prefix);
        //借助于 AjaxXmlBuilder 返回 XML 字符串
        AjaxXmlBuilder builder = new AjaxXmlBuilder();
        //以 list 集合来创建 XML 响应
        builder = builder.addItems(list, "name", "publisher");
        return builder.toString();
    }
}
```

该 Servlet 根据发送的请求参数，调用 getBooksByPrefix()业务方法，从所有的书籍中选择出所有书名以 prefix 开头的书籍。把该 Servlet 配置在 web.xml 文件中，该 Servlet 即能对 Ajax 请求提供响应。在 web.xml 文件中增加如下片段来完成 Servlet 的配置。

```
<!-- 配置 autocomplete Servlet -->
<servlet>
    <servlet-name>autocomplete</servlet-name>
    <servlet-class>lee.AutocompleteServlet</servlet-class>
</servlet>
<!-- 为 autocomplete Servlet 指定 URL -->
<servlet-mapping>
    <servlet-name>autocomplete</servlet-name>
```

```
    <url-pattern>/autocomplete</url-pattern>
</servlet-mapping>
```

为 AutocompleteServlet 指定了合适的 URL 之后，即可让该 Servlet 为 Ajax 请求提供响应了，即可在页面中使用 autocomplete 标签了，该标签提供了自动完成功能。下面是在页面中使用自动完成标签的代码片段：

程序清单：codes\15\15.4\TagsTest\autocomplete.jsp

```
<!--
根据 name 文本框的值改变来发送 Ajax 请求，
将 value 节点的值输入 publisher 文本框
Ajax 请求向 autocomplete 发送
自动完成的提示框的 CSS 样式为:autocomplete
当 Ajax 交互时显示 indicator 元素
至少需要 2 个字符才发送 Ajax 请求
-->
<ajax:autocomplete
    source="name"
    target="publisher"
    baseUrl="autocomplete"
    className="autocomplete"
    parameters="prefix={name}"
    indicator="indicator"
    minimumCharacters="2"/>
```

图 15.3 显示了自动完成的示范效果。

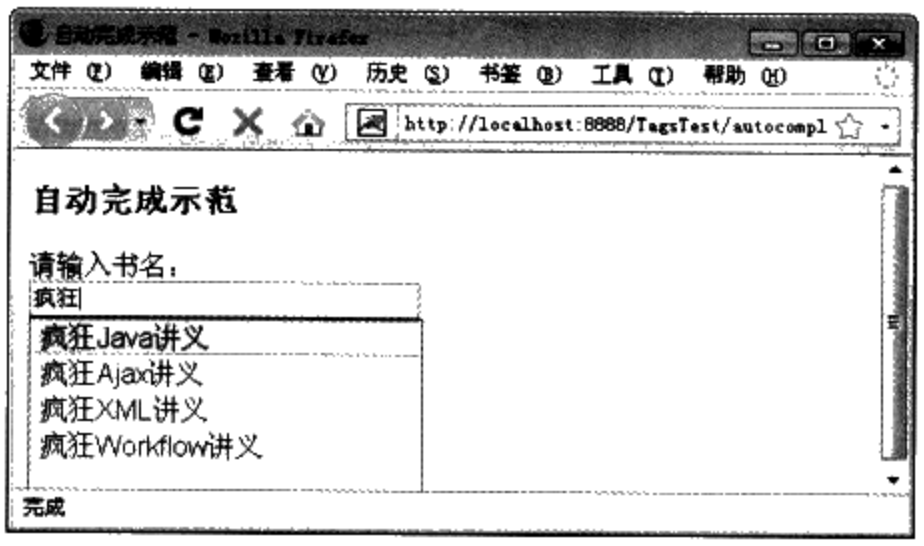


图 15.3 自动完成的效果

一旦用户选择了相应的书籍，该书籍的出版社将自动填写在 publisher 文本框内。

➤➤15.4.2 使用 area 标签

该标签允许在页面中开辟出一个单独区域，而该区域的超级链接请求等不会刷新整个页面，而是仅仅刷新该页面部分。该标签有如下几个属性：

- id：这是一个必需属性，该属性指定 area 的 ID，用于唯一标识该 area，该属性值将成为该 area 对应的<div.../>元素的 id 属性值。
- ajaxFlag：这是个可选属性，该属性指定该页面的其他部分是否忽略 Ajax 请求。该属性是控制局部刷新的关键，指定该属性为 true 则只刷新 area 标签包含的区域。
- style：这是一个可选属性，用于指定内联的 CSS 样式。
- styleClass：这是一个可选属性，用于指定 CSS 样式单名。
- ajaxAnchors：这是一个可选属性。

下面是一个使用 area 标签的示例代码片段：

程序清单：codes\15\15.4\TagsTest\pagearea.jsp

```
<jsp:useBean id="now" class="java.util.Date"/>
日期：${now}<p>
<ajax:area id="myarea" ajaxAnchors="true" ajaxFlag="myarea"
    style="background:#eeeeee;width:300px; height:80px;">
简单的页面 area
<br /><a href="pagearea.jsp">单击此处</a><br />
日期：${now}
</ajax:area>
```

页面中分别有两次输出当前时间的代码，页面中有两次输出 now 变量的代码。因为后一个日期的输出放在 ajax:area 标签内，如果单击“单击此处”的超级链接时，第二个时间会得到更新，第一个时间不会改变。因为该超级链接刷新 ajax:area 标签内的部分内容，不会刷新整个页面。如果单击了“单击此处”超级连接，我们看到第二个日期发生了改变，但第一个日期不会刷新。图 15.4 显示了 ajax:area 的局部刷新效果。

图 15.4　ajax:area 的局部刷新

➤➤15.4.3　使用 anchors 标签

这是一个超级链接标签，但该超级连接标签与普通的超级连接不同，该超级连接标签仅仅刷新页面的局部——刷新 ajax:area 标签指定的页面部分。因此，该标签必须与 ajax:area 一起使用。该标签有如下两个属性：

- target ：这是一个必需属性，该属性指定刷新的 ajax:area 元素的 ID 属性。
- ajaxFlag：这是个可选属性，该属性指定该页面的其他部分是否忽略 Ajax 请求。

通过使用 ajax:anchors 标签可在页面生成一个超级链接，该超级链接可控制页面的指定区域局部刷新，无需使用 ajax:area 中的超级链接来控制局部刷新。

程序清单：codes\15\15.4\TagsTest\anchors.jsp

```
<jsp:useBean id="now" class="java.util.Date"/>
日期：${now}<p>
<!-- 指定刷新 myarea 区域 -->
<ajax:anchors target="myarea" ajaxFlag="myarea">
    <a href="anchors.jsp">单击此处</a></ajax:anchors>
<ajax:area id="myarea" ajaxAnchors="true" ajaxFlag="myarea"
    style="background:#eeeeee;width:300px; height:80px;">
简单的页面 area
<br /><a href="pagearea.jsp">单击此处</a><br />
日期：${now}
</ajax:area>
```

上面页面中有两个超级链接，一个是页面范围的超级链接，一个是在 ajax:area 范围内的超级链接。两个超级链接都可以控制页面的局部刷新。

➤➤15.4.4 使用 callout 标签

该标签是一个服务器提示功能，这个功能在以前通常在客户端完成，当用户的鼠标移动到某个产品上面时，该产品上将出现一个提示框，但这个提示框的信息往往是写在客户端的。通过使用 callout 标签，可以让服务器响应来作为提示框的信息。从 AjaxTags 1.2 以后，该功能的实现依赖于 OverLIBMWS JavaScript 库，因此应将对应的 JavaScript 库复制到 Web 应用里。该标签支持如下几个属性：

- var：这是一个可选属性，该属性定义了 callout 标签创建的 JavaScript 对象名。通常无需指定该属性。
- attachTo：这是一个可选属性，必须先定义 var 属性才可指定该属性。
- baseUrl：这是一个必需属性，该属性指定发送 Ajax 请求的目标 URL。该属性指定的 URL 将返回一个标准的 HTML 页面。
- source：该属性指定的哪个 HTML 元素将触发服务器提示框，即指定哪个 HTML 元素触发 Ajax 请求。source 和 sourceClass 两个属性必须指定其中之一。
- sourceClass：该属性指定一类 HTML 元素将触发服务器提示框，即指定那些 HTML 元素是该 CSS 样式单，这些 HTML 元素将都可以触发 Ajax 请求。source 和 sourceClass 两个属性必须指定其中之一。
- parameters：伴随 Ajax 请求发送的请求参数。该属性的值支持一个特殊的变量 ajaxParameter，该变量代表发送请求的内容。
- title：这是一个可选属性，该属性指定信息提示框的标题。
- overlib：这是一个可选属性，该属性指定 OverLib 库的各种选项。
- preFunction、postFunction、errorFunction：三个指定回调函数的属性。

callout 标签所需要的 XML 响应只需要一个 item 节点，该节点的 name 节点值将作为提示的标题显示，而 value 节点值作为提示的内容显示。下面是示例应用的 Servlet。

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\CalloutServlet.java

```
public class CalloutServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法
    public String getXmlContent(HttpServletRequest request,
        HttpServletResponse response)throws Exception
    {
        //设置解析请求参数所用的字符集
        request.setCharacterEncoding("UTF-8");
        //获取请求参数
        String param = request.getParameter("book");
        System.out.println(param);
        //第一个参数是 name 节点的值，作为提示的标题。
        //第二个参数是 value 节点的值，作为提示的内容。
        AjaxXmlBuilder builder = new AjaxXmlBuilder().
            addItemAsCData( "提示标题",
            "<p>关于书籍:<b>" + param + "</b>的信息如下：<br>"
            + "服务器的提示信息 </p>");
        return builder.toString();
    }
}
```

将该 Servlet 配置在应用中，为了配置该 Servlet，在 web.xml 文件中增加如下片段：

```
<!-- 配置 callout Servlet -->
<servlet>
    <servlet-name>callout</servlet-name>
    <servlet-class>lee.CalloutServlet</servlet-class>
</servlet>
<!-- 为 callout Servlet 指定 URL -->
<servlet-mapping>
    <servlet-name>callout</servlet-name>
    <url-pattern>/callout</url-pattern>
</servlet-mapping>
```

该 Servlet 即可以响应用户的 Ajax 请求了，下面是页面中使用 callout 标签的代码片段：

程序清单：codes\15\15.4\TagsTest\callout.jsp

```
<div>当鼠标移动到下面的书名上时，可看到服务器提示:</div>
<div style="width: 400px; border:1px dashed #909090;">
疯狂 Java 体系图书:
<ul>
<li><span class="book">疯狂 Java 讲义</span></li>
<li><span class="book">轻量级 Java EE 企业应用实战</span></li>
<li><span class="book">疯狂 Ajax 讲义</span></li>
<li><span class="book">疯狂 XML 讲义</span></li>
</ul>
</div>
<ajax:callout
    baseUrl="callout"
    sourceClass="book"
    parameters="book={ajaxParameter}"
    title="书籍详细信息"/>
```

读者应该看到 parameters 属性的值为 book={ajaxParameter}，其中 ajaxParameter 是个特殊的变量，这个变量代表任何发送请求的 HTML 元素本身。当轻量级 Java EE 企业应用实战发送请求时，该变量的值就是轻量级 Java EE 企业应用实战。图 15.5 显示了这种服务器提示的效果。

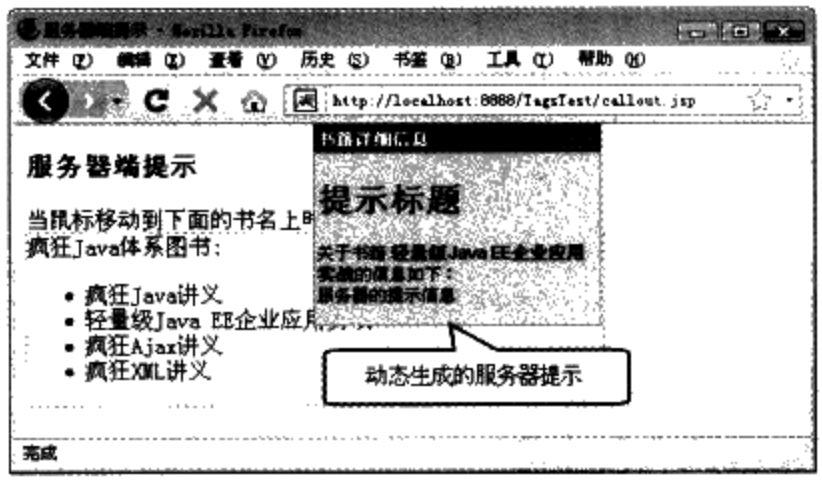


图 15.5　服务器提示

➤➤15.4.5　使用 htmlContent 标签

该标签能将服务器响应在当前页面的指定区域（通常是一个<div.../>元素）内显示出来。该标签不再需要 XML 响应，它只需要一个标准的 HTML 响应，这个 HTML 响应将以异步方式在指定页面加载出来。该标签有如下几个属性：

- var：这是一个可选属性，该属性定义了 htmlContent 标签创建的 JavaScript 对象名。通常无需指定该属性。
- attachTo：这是一个可选属性，必须先定义 var 属性才可指定该属性。

- baseUrl：这是一个必需属性，该属性指定发送 Ajax 请求的目标 URL。
- source：该属性指定的哪个 HTML 元素将触发服务器响应，即指定哪个 HTML 元素触发 Ajax 请求。source 和 sourceClass 两个属性必须指定其中之一。
- sourceClass：该属性指定哪一批 HTML 元素将触发服务器响应，即指定那些 HTML 元素是该 CSS 样式单，这些 HTML 元素将都可以触发 Ajax 请求。source 和 sourceClass 两个属性必须指定其中之一。
- target：这是一个必需属性，该属性指定了 HTML 响应的输出容器。
- parameters：这是一个可选属性，如果需要发送请求参数，则需要使用该属性。
- preFunction、postFunction、errorFunction：三个指定回调函数的属性。

下面是提供了 htmlContent 响应的 Servlet，该 Servlet 不再生成 XML 响应，而是生成 HTML 响应，该 Servlet 的代码如下：

程序清单：codes\15\15.4\TagsTest\htmlContent.jsp

```
public class HtmlContentServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，生成服务器响应。
    public String getXmlContent(HttpServletRequest request
        , HttpServletResponse response)throws Exception
    {
        //设置服务器解码方式
        request.setCharacterEncoding("UTF-8");
        //获取请求参数
        String publisher = request.getParameter("publisher");
        //创建业务逻辑组件实例
        BookService service = new BookService();
        //根据出版社获取所有的书籍
        List<Book> list = service.getBooksByPublisher(publisher);
        //开始拼接需要返回的字符串
        StringBuffer html = new StringBuffer();
        html.append("<h3>").append(publisher)
            .append("出版的书籍包括如下</h3>");
        for (Book book: list)
        {
            html.append("<li>")
                .append(book.getName()).append("</li>");
        }
        html.append("</ul>");
        html.append("<br/>");
        html.append("最后更新时间:" + new Date());
        return html.toString();
    }
}
```

正如代码中看到，该 Servlet 不再借助于 AjaxXmlBuilder 类来辅助生成 XML 响应，该 Servlet 不再返回一个 XML 文件，而是返回一个 HTML 文档。htmlContent 会将该响应直接输出在 HTML 文档的目标元素中。将该 Servlet 配置在 Web 应用中，在 web.xml 文件中增加如下片段：

```
<!-- 配置 htmlContent Servlet -->
<servlet>
    <servlet-name>htmlContent</servlet-name>
    <servlet-class>lee.HtmlContentServlet</servlet-class>
</servlet>
<!-- 为 htmlContent Servlet 指定 URL -->
<servlet-mapping>
    <servlet-name>htmlContent</servlet-name>
```

```
    <url-pattern>/htmlContent</url-pattern>
</servlet-mapping>
```

在页面中使用 ajax:htmlContent 标签，本页面中使用了两种方式来输出 htmlContent 内容，一种是采用超级链接的方式，一种是采用下拉俩表框的形式。代码片段如下：

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\HtmlContentServlet.java

```
选择出版社查看详细信息：<br />
<ul>
    <li><a href="#" class="publisher">电子工业出版社</a></li>
    <li><a href="#" class="publisher">清华大学出版社</a></li>
    <li><a href="#" class="publisher">机械工业出版社</a></li>
</ul>
<p>选择出版社查看详细信息：</p>
<select id="publishSelector" name="publishSelector">
    <option value="电子工业出版社">电子工业出版社</option>
    <option value="清华大学出版社">清华大学出版社</option>
    <option value="机械工业出版社">机械工业出版社</option>
</select>
<div style="position:absolute;left:300px;top:20px;background-color:#ffffaa"
    id="bookDesc" ></div>
<ajax:htmlContent
    baseUrl="htmlContent"
    sourceClass="publisher"
    target="bookDesc"
    parameters="publisher={ajaxParameter}"/>
<ajax:htmlContent
    baseUrl="htmlContent"
    source="publishSelector"
    target="bookDesc"
    parameters="publisher={publishSelector}"/>
```

页面中的超级链接和下拉框都可以激发 htmlContent 的效果，一旦用户单击了超级链接，或者用户选择了下拉列表，都可以看到该出版社所有的出版的图书。图 15.6 显示了 htmlContent 标签的效果。

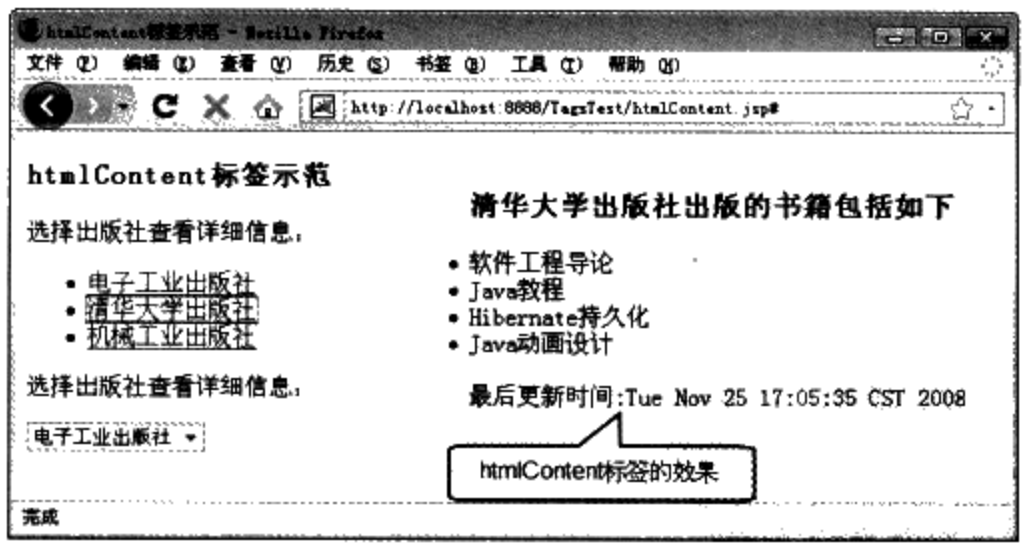


图 15.6　使用 htmlContent 输出 HTML 响应

➤➤15.4.6　使用 portlet 标签

该标签将在页面上生成一个 Portlet 区域，该区域的内容直接显示服务器端 HTML 响应，类似于 htmlContent 标签，该标签不需要 XML 响应，而是支持 HTML 响应。使用 ajax:portlet 标签，还可以定义该 Portlet 的内容是否支持周期性刷新。该标签包含如下几个属性：

- var：这是一个可选属性，该属性定义了 portlet 标签创建的 JavaScript 对象名。通常无需指定该属性。

- attachTo：这是一个可选属性，必须先定义 var 属性才可指定该属性。
- baseUrl：这是一个必需属性，该属性指定了 Ajax 请求发送的 URL。
- source：这是一个必需属性，该属性指定了 portlet 的 id 属性值。
- parameters：这是一个可选属性，该属性指定发送 Ajax 请求的请求参数。
- classNamePrefix：这是一个可选属性，该属性指定了 portlet's 的“Box”、“Tools”、“refresh”、“Size”、“Close”、“Title”、和“Content”元素的 CSS 样式名的前缀。
- title：这是一个必需属性，该属性指定 portlet 的标题。
- imageClose：这是一个可选属性，该属性指定关闭按钮的图标。
- imageMaximize：这是一个可选属性，该属性指定最大化按钮的图标。
- imageMinimize：这是一个可选属性，该属性指定最小化按钮的图标。
- imageRefresh：这是一个可选属性，该属性指定刷新按钮的图标。
- refreshPeriod：这是一个可选属性，该属性指定 Portlet 的内容刷新频率，即隔多少秒刷新一次。如果没有指定该属性，则 Portlet 的内容不会自动刷新，除非手动刷新。默认情况下，当页面加载时，portlet 的内容也会刷新，但如果设置了 executeOnLoad 为 false，则页面载入时，Portlet 的内容也不会刷新，除非手动刷新。
- executeOnLoad：这是一个可选属性，该属性指定当页面重载时，是否重新检索 Portlet 里的内容。默认是重新检索。
- expireDays：这是一个可选属性，该属性指定 cookie 持久保存的天数。
- expireHours：这是一个可选属性，该属性指定 cookie 持久保存的小时数。
- expireMinutes：这是一个可选属性，该属性指定 cookie 持久保存的分钟数。
- preFunction、postFunction、errorFunction：这三个属性用于指定 Ajax 交互的回调函数。

该标签需要的响应类似于 htmlContent 标签的响应——只是需要 HTML 响应即可。与 htmlContent 标签不同的是，portlet 标签会创建一个小窗口来装载 HTML 响应。

此处不再单独为该标签书写服务器处理类，而是直接向 htmlContent 发送 Ajax 请求，因此可以在页面中直接使用 portlet 标签，使用 portlet 标签的代码片段如下：

程序清单：codes\15\15.4\TagsTest\portlet.jsp

```
<ajax:portlet
    source="tsinghua"
    baseUrl='htmlContent'
    classNamePrefix="portlet"
    title="清华大学出版社 Portlet"
    imageClose="img/close.png"
    imageMaximize="img/maximize.png"
    imageMinimize="img/minimize.png"
    imageRefresh="img/refresh.png"
    parameters="publisher=清华大学出版社"
    refreshPeriod="5" />
<ajax:portlet
    source="phei"
    baseUrl='htmlContent'
    classNamePrefix="portlet"
    title="电子工业出版社 Portlet"
    imageClose="img/close.png"
    imageMaximize="img/maximize.png"
    imageMinimize="img/minimize.png"
    imageRefresh="img/refresh.png"
    parameters="publisher=电子工业出版社"
    refreshPeriod="5" />
```

上面的页面使用了两个 portlet，页面执行的效果如图 15.7 所示，其中清华大学出版社的 Portlet 已经被最小化了，因此看不到该 Portlet 的内容。

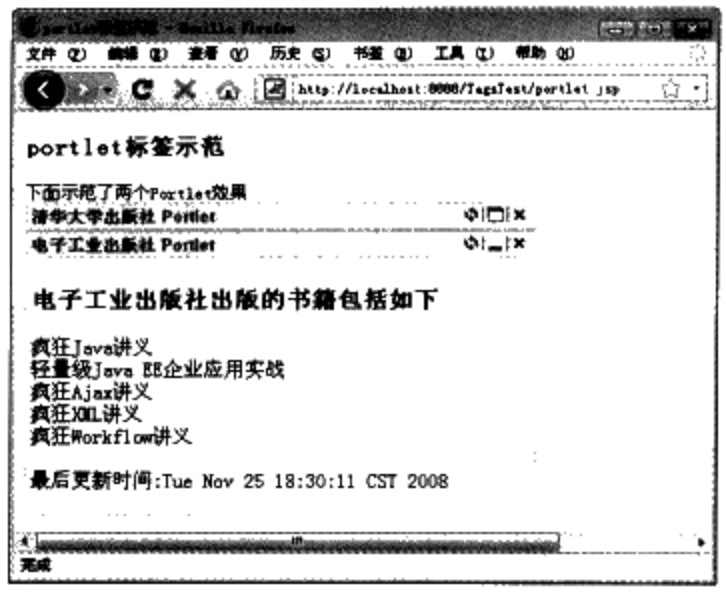


图 15.7　使用 portlet 标签生成 Portlet

➤➤15.4.7　使用 select 标签

select 标签就是在本章的第二节和第三节频繁使用的标签，它的主要作用是完成级联下拉框效果。所谓级联下拉框，根据第一个下拉框选择的值不同，第二个下拉框的选项可以动态改变。select 标签有如下几个属性：

- var：这是一个可选属性，该属性定义了 select 标签创建的 JavaScript 对象名。通常无需指定该属性。
- attachTo：这是一个可选属性，必须先定义 var 属性才可指定该属性。
- baseUrl：这是一个必需属性，该属性指定发送 Ajax 请求的目标 URL。
- source：这是一个必需属性，该属性指定了第一个下拉框的 id 属性，当该属性指定的下拉框改变时触发 Ajax 交互。
- target：这是一个必需属性，该属性指定了第二个下拉框的 id 属性，当 Ajax 响应完成后，AjaxTags 会根据响应来自动更新该属性指定的下拉框。
- parameters：这是一个可选属性，如果需要发送请求参数，则需要使用该属性。
- eventType：这是一个可选属性，该属性指定了源对象上触发请求的事件类型。
- executeOnLoad：这是一个可选属性。
- defaultOptions：这是一个可选属性，该属性是一系列以逗号（,）隔开的值，这些值将总是作为第二个下拉框的默认选项。
- preFunction、postFunction、errorFunction：三个用于指定 Ajax 回调的属性
- parser：这是一个可选属性，该参数指定一个解析服务器响应的解析器，通常该解析器无需指定。除非用户需要自己完成特别的工作。默认解析器是 ResponseHtmlParser。

因为在第二节和第三节已经大量使用了该标签，因此此处不再给出关于 select 标签的示范应用。值得注意的是：每个 select 标签只能指定一个源下拉框，一个目标下拉框，这往往不能满足实际的需要。例如一个常用场景：第一个下拉框是国家，第二个下拉框是省份，第三个下拉框是城市，每个下拉框的值都应该随前面下拉框的值的改变而改变。AjaxTags 的 select 能否完成这个需要呢？答案是肯定的，只要使用两个 select 标签即可。

对于第一个 select 标签，国家列表框是源列表框，省份列表框是目标列表框；对于第二个 select 标签，省份列表框是源列表框，城市列表框是目标列表框。

提　示：

使用 ajax:select 标签，可以很方便地实现多级联动列表框。每个 ajax:select 标签只能控制 2 个列表框的联动，为了控制多个列表框之间的联动，应该在 JSP 页面使用多个 ajax:select 标签。

➤➤15.4.8 创建 Tab 页

Tab 的创建依赖于两个标签：tabPanel 和 tab。一个 tabPanel 标签是一个整体的 Tab 框，每个 tab 则负责生成一个 Tab 页。因此，tabPanel 和 tab 两个标签通常一起使用。tabPanel 标签的包含如下几个属性：

- var：这是一个可选属性，该属性定义了 tabPanel 标签创建的 JavaScript 对象名。通常无需指定该属性。
- attachTo：这是一个可选属性，，必须先定义 var 属性才可指定该属性。
- panelStyleId：这是一个必需属性，指定 Tab 框的 id 属性值。
- contentStyleId：这是一个必需属性，指定 Tab 框里内容的 id 属性值。
- panelStyleClass：这是一个可选属性，该属性指定了 Tab 框整体使用的 CSS 样式。
- contentStyleClass：：这是一个可选属性，该属性指定 Tab 框里内容所使用的 CSS 样式。
- currentStyleClass：这是一个必需属性，该属性指定了激活状态下 Tab 页所使用的 CSS 样式。
- preFunction、postFunction、errorFunction：三个指定 Ajax 回调函数的属性。
- parser：这是一个可选属性，该参数指定一个解析服务器响应的解析器，通常该解析器无需指定。除非用户需要自己完成特别的工作。默认解析器是 ResponseHtmlParser。

tab 标签可指定如下几个属性：

- baseUrl：这是一个必需属性，该属性指定发送 Ajax 请求的目标 URL。
- caption：这是一个必需属性，该属性指定了该 Tab 页的标题。
- defaultTab：这是一个可选属性，该属性指定该页面是否作为 Tab 框默认的激活页。
- parameters：这是一个可选属性，如果加载该页面时需要发送请求参数，则需要指定该属性。

值得注意的是，tab 标签的 Ajax 响应无需使用 XML 响应，应该直接使用标准的 HTML 响应，tab 标签将把 HTML 内容直接输出在 Tab 页中。

下面应用示范一个简单的 Tab 效果，每个 Tab 页面的 baseUrl 都使用前面 htmlContent 中已经定义了的 htmlContent Servlet。下面是使用 tabPanel 和 tab 标签的代码片段：

程序清单：codes\15\15.4\TagsTest\tab.jsp

```
<!-- 使用 tabPanel 构建整体的 Tab 效果 -->
<ajax:tabPanel
    panelStyleId="tabPanel"
    contentStyleId="tabContent"
    panelStyleClass="tabPanel"
    contentStyleClass="tabContent"
    currentStyleClass="ajaxCurrentTab">
    <!-- tabPanel 的每个 tab 子标签对应一个 Tab 页 -->
    <ajax:tab caption="电子工业出版社"
        baseUrl="htmlContent"
        parameters="publisher=电子工业出版社"
        defaultTab="true"/>
    <ajax:tab caption="清华大学出版社"
        baseUrl="htmlContent"
        parameters="publisher=清华大学出版社"/>
    <ajax:tab caption="机械工业出版社"
        baseUrl="htmlContent"
        parameters="publisher=机械工业出版社"/>
</ajax:tabPanel>
```

上面代码将 ajax:tab 标签放在 ajax:tabPanel 内，从而形成一个整体的 Tab 框，每个 tab 标签对应一个 Tab 页，每个 Tab 页所显示的 URL 完全相同，但随请求参数的不同，因此每个 Tab 页的内容也不相同。图 15.8 显示了该 Tab 框的效果。

图 15.8　Tab 效果

➤➤15.4.9　使用 displayTag 标签

displayTag 标签需要依赖于 Apache 组织下的 DisplayTag 项目，AjaxTags 包装了 DisplayTag 项目，但它增加了 Ajax 机制，从而允许以 Ajax 的方式来排序，分页。由于该标签的核心是 DisplayTags 项目，开发者应该具有 DisplayTags 的相关知识。AjaxTags 中 displayTag 标签有如下几个属性：

- id：这是一个必需属性，该属性指定了 displayTag 对应的<div.../>元素的 id 属性值。
- ajaxFlag：指定是否忽略页面其他部分内容。该属性是进行 Ajax 局部刷新的关键，将该属性设为 true 将会使得 displayTag 生成的表格以异步方式进行排序、分页。
- style：指定 displayTag 生成表格的内联 CSS 样式。
- styleClass：这是一个可选属性，该属性指定 displayTag 的 CSS 样式。
- pagelinksClass：这是一个可选属性，该属性指定 DisplayTag 的分页导航条的 CSS 样式。
- tableClass：这是一个可选属性，该属性指定 DisplayTag 的 table 元素的 CSS 样式。
- columnClass：这是一个可选属性，该属性指定 DisplayTag 里的 table 里每列的 CSS 样式。
- baseUrl：这是一个可选属性，没有太大的作用。
- postFunction ：这是一个可选属性，该属性指定了 Ajax 交互完成后自动执行的函数。

为了使用 AjaxTags 的 displayTag 标签，必须先在 Web 应用中安装 DisplayTags 项目，安装 DisplayTags 请按如下步骤进行：

（1）将 displaytag-{version}.jar 文件复制到 Web 应用的 WEB-INF/lib 下。

（2）将 DisplayTag 所依赖的 JAR 文件复制到 Web 应用的 WEB-INF/lib 下。这个步骤对于 AjaxTags 而言，往往已经完成了，因此无需额外复制其他 JAR 包。

（3）如果需要使用 Display 的导出选项，这种导出选项非常有用，它可以将表格显示的内容直接导出成 XML 文档，Excel 文档等。为了支持导出选项，则应该在 web.xml 文件中配置如下

```
<!-- 配置 ResponseOverrideFilter -->
<filter>
    <filter-name>ResponseOverrideFilter</filter-name>
    <filter-class>org.displaytag.filter.ResponseOverrideFilter</filter-class>
</filter>
<!-- 指定 ResponseOverrideFilter 需要过滤 displaytag.jsp -->
<filter-mapping>
    <filter-name>ResponseOverrideFilter</filter-name>
    <url-pattern>/displaytag.jsp</url-pattern>
</filter-mapping>
```

（4）如果需要自定义 DisplayTag 的显示效果，则还需要增加一个 displaytag.properties 文件，关于该文件的各种属性以及具体含义请参考 DisplayTag 的官方文档。下面是本示例应用增加在 WEB-INF/classes 路径下的 displaytag.properties 文件。

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\displaytag.properties

```
sort.behavior=list
sort.amount=list
basic.empty.showtable=true
basic.msg.empty_list=找不到满足要求的结果
paging.banner.placement=bottom
paging.banner.some_items_found=查询到{0}条记录， 当前显示从{2}到{3}条记录.
export.types=csv excel xml
export.amount=list
export.csv=true
export.excel=true
export.pdf=true
export.xml=true
export.excel.include_header=true
```

因为该文件中包含了中文字符，因此还必须使用 native2ascii 命令将该属性文件转为国际化的属性文件。经过这 4 个步骤，该 Web 应用就可以支持 DisplayTag 标签了，也就可以使用 AjaxTags 的 displayTag 标签了，在页面中使用 displayTag 标签的代码如下：

程序清单：codes\15\15.4\TagsTest\displaytag.jsp

```
<jsp:useBean id="now" class="java.util.Date"/>
<!-- 直接初始化业务逻辑组件 -->
<jsp:useBean id="service" class="lee.BookService" />
下面的内容不刷新，页面只刷新表格内容<br/>
页面加载时间：${now}<br/>
<!-- 将 display 标签放在 ajax:display 标签内，使得 Display 标签
    能以 Ajax 方式进行排序，分页 -->
<ajax:displayTag id="displayTagFrame" ajaxFlag="displayAjax">
    ajax:displayTag 内的更新时间：${now}
    <display:table name="pageScope.service.allBooks" class="displaytag"
        pagesize="10" defaultsort="1" export="true"
        defaultorder="descending" id="row" excludedParams="ajax">
        <!-- 输出业务逻辑组件中的两列 -->
        <display:column property="name" title="书名"
            sortable="true" headerClass="sortable" />
        <display:column property="publisher" title="出版社"
            sortable="true" headerClass="sortable" />
    </display:table>
</ajax:displayTag>
```

在浏览器中浏览该页面，看到如图 15.9 的效果。

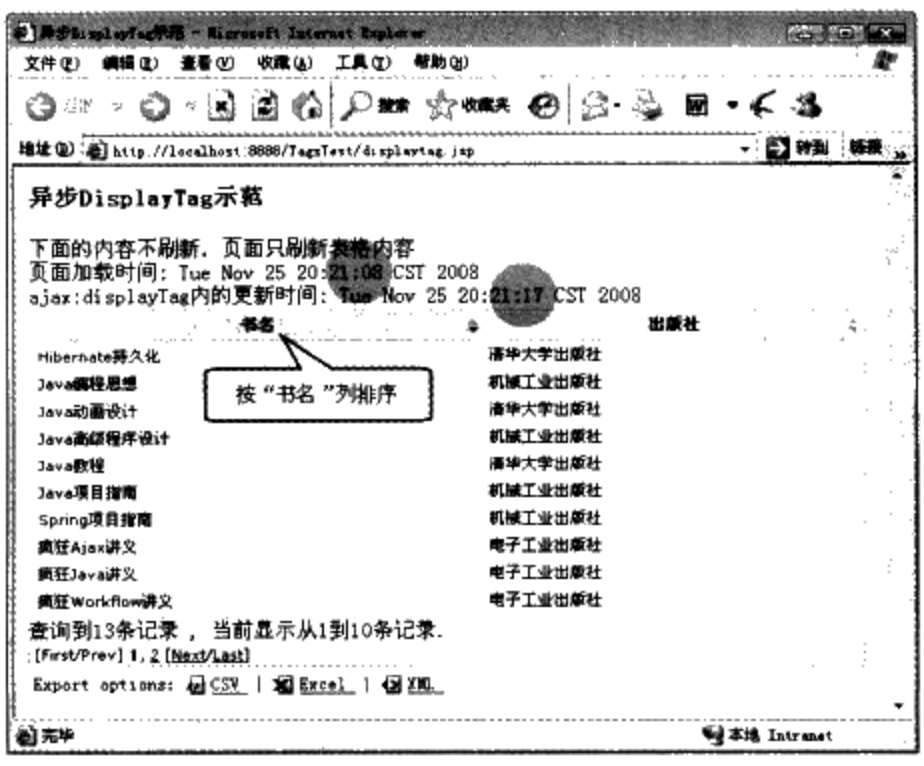
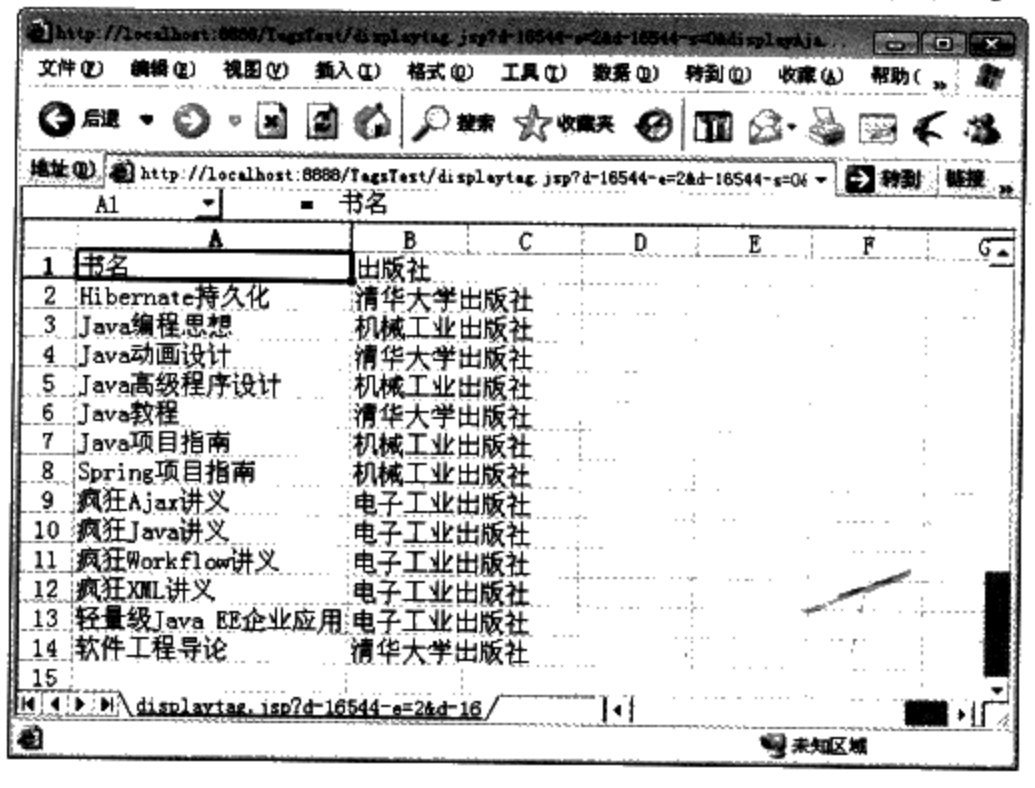


图 15.9　使用 AjaxTags 的 displayTag 标签

读者可看到页面中两次输出的时间并不相同，那是因为笔者已经单击了“书名”列，从而按“书名”排序了，只是这种排序是以 Ajax 方式进行的，因此只刷新表格部分，并未刷新整个页面内容，从而看到两个时间并不相同。

如果单击表格下面的分页导航，则可看到以 Ajax 方式完成分页。如果单击下面的导出 Excel 按钮，将可看到如图 15.10 所示的界面，这是 DisplayTag 的功能，与 Ajax 的 displayTag 并没有什么关系。



	A	B	C	D	E	F	G
1	书名	出版社					
2	Hibernate持久化	清华大学出版社					
3	Java编程思想	机械工业出版社					
4	Java动画设计	清华大学出版社					
5	Java高级程序设计	机械工业出版社					
6	Java教程	清华大学出版社					
7	Java项目指南	机械工业出版社					
8	Spring项目指南	机械工业出版社					
9	疯狂Ajax讲义	电子工业出版社					
10	疯狂Java讲义	电子工业出版社					
11	疯狂Workflow讲义	电子工业出版社					
12	疯狂XML讲义	电子工业出版社					
13	轻量级Java EE企业应用	电子工业出版社					
14	软件工程导论	清华大学出版社					
15							

图 15.10　导出 Excel 文档的效果

➤➤15.4.10　使用 tree 标签创建树

树是 HTML 页面中常用的控件，它可用于显示组织结构图，也可用于复杂的页面导航。当需要表现某个对象层次关系，也可考虑使用树。

AjaxTags 提供了 tree 标签来创建树，AjaxTags 创建的树是基于异步请求的树——当浏览者单击该树的某个节点时，系统会向服务器发送异步请求，从而动态地加载该节点的所有子节点。

AjaxTags 的 tree 标签可支持如下几个属性：

- baseUrl：指定发送 Ajax 请求的目标 URL。该 URL 的响应将作为子节点显示出来。
- parameter：必填参数，该属性指定发送 Ajax 请求的参数。
- styleId：必填参数，指定该 tree 的 ID。当页面初始化该树时，该属性指定的值会作为参数值发送到服务器。
- preFunction、postFunction、errorFunction：指定回调函数的三个属性。
- parser：指定解析服务器响应的解析器。该属性的默认值是 ResponseXmlToHtmlLinkListParser。
- collapsedClass：指定折叠节点前图标的 CSS 样式。
- expandedClass：指定展开节点前图标的 CSS 样式。
- treeClass：指定该树的 CSS 样式。
- nodeClass：指定节点的 CSS 样式。

为了使用 AjaxTags 的 tree 标签，AjaxTags 提供了 AjaxTreeXmlBuilder 工具类，它可以提供满足 tree 标签要求的响应，AjaxTreeXmlBuilder 工具类主要提供了如下三个重载的方法：

- addItem(String name, String value...)：以 name、value 添加一个树节点。
- addItem(TreeItem item)：以 TreeItem 对象作为参数添加一个树节点。
- addItems(Collection collection, String nameProp, String valueProp...)：遍历 collection 集合，每个

集合元素对应添加一个树节点。

TreeItem 实例用于封装一个树节点，通常包含了该树节点的文本、树节点的值、URL 以及是否展开等数据。AjaxTreeXmlBuilder 可通过添加 TreeItem 实例来添加一个树节点。

下面是为 tree 标签提供响应的 Servlet 类代码：

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\TreeServlet.java

```
public class TreeServlet extends BaseAjaxServlet
{
    public String getXmlContent(HttpServletRequest request,
            HttpServletResponse response) throws Exception
    {
        //获取 node 请求参数
        String node = request.getParameter("node");
        //创建业务逻辑组件的实例
        BookService service = new BookService();
        AjaxTreeXmlBuilder treeBuilder = new AjaxTreeXmlBuilder();
        //如果 node 参数值为"出版社"
        if (node.equals("出版社"))
        {
            //添加三个节点
            treeBuilder.addItem("电子工业出版社", "电子工业出版社", true, "#");
            treeBuilder.addItem("清华大学出版社", "清华大学出版社", true, "#");
            treeBuilder.addItem("机械工业出版社", "机械工业出版社", true, "#");
        }
        //如果 node 参数值为"电子工业出版社"
        else if (node.equals("电子工业出版社"))
        {
            //获取电子工业出版社的所有图书
            List<Book> books = service
                .getBooksByPublisher("电子工业出版社");
            for (Book book : books)
            {
                //创建 TreeItem 节点
                TreeItem item = new TreeItem(book.getName() ,
                    book.getName() , false);
                //设置该节点是叶子节点
                item.setLeaf(true);
                treeBuilder.add(item);
            }
        }
        else
        {
            //以 node 作为参数，获取指定出版社的全部图书
            List<Book> books = service.getBooksByPublisher(node);
            //遍历 books 集合，每个集合元素对应添加一个树节点
            treeBuilder.addItems(books , "name" , "name",
                "name", "name");
        }
        return treeBuilder.toString();
    }
}
```

定义上面 Servlet 之后，在 web.xml 文件中配置该 Servlet，配置该 Servlet 的配置片段如下：

```
<!-- 配置 tree Servlet -->
<servlet>
    <servlet-name>tree</servlet-name>
    <servlet-class>lee.TreeServlet</servlet-class>
```

```
    </servlet>
    <!-- 为 tree Servlet 指定 URL -->
    <servlet-mapping>
        <servlet-name>tree</servlet-name>
        <url-pattern>/tree</url-pattern>
    </servlet-mapping>
```

接下来就可以在 JSP 页面中使用 tree 标签来生成树了，下面是 JSP 页面中使用 tree 标签的代码片段：

程序清单：codes\15\15.4\TagsTest\tree.jsp

```
<ajax:tree
    baseUrl="tree"
    styleId="出版社"
    parameters="node={ajaxParameter}"
    nodeClass="nodeClass"
    expandedClass="expandedNode"
    collapsedClass="collapsedNode"
    treeClass="tree" />
```

在浏览器中浏览该页面将可看到如图 15.11 所示的效果。

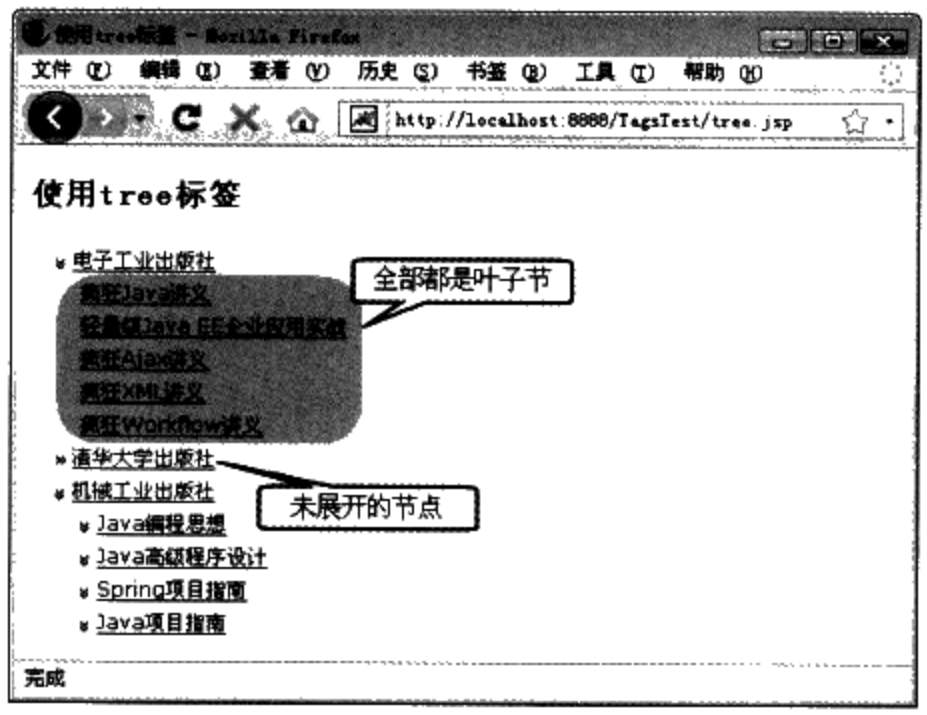


图 15.11　使用 tree 标签生成的树

➤➤15.4.11　使用 updateField 标签

这个标签也完成了一种常用的效果，当一个表单域的输入完成后，其他几个表单域的值是根据该表单域的值计算得到的，就可以采用该标签。大部分的时候，如果这种计算无需服务器数据参与，则这种计算可以在客户端通过 JavaScript 计算完成。但在某些情况下，例如商品的折扣，是可以通过后台程序设定的，则需要服务器数据的参与，因此应该使用该标签来完成该效果。updateFiled 标签有如下几个属性：

- var：可选属性，该属性定义了 updateField 标签创建的 JavaScript 对象名。通常无需指定该属性。
- attachTo：可选属性，必须指定了 var 属性才可指定该属性。
- baseUrl：必填属性，该属性指定发送 Ajax 请求的目标 URL。
- source：必填属性，该属性指定一个表单域，该表单域的值将作为请求参数，随 Ajax 请求向服务器发送。

- target：必填属性，该属性的值以逗号（,）隔开，指定了一系列的表单域，Ajax 响应的结果将在这些表单域中输出。
- action：必填属性，该属性指定的 HTML 元素能触发 onclick 事件，该事件将触发 Ajax 交互。
- parameters：可选属性，该属性指定需要发送到服务器端的请求参数。
- eventType：可选属性，该属性指定能触发 Ajax 请求的事件类型。
- preFunction：、postFunction、errorFunction：指定 Ajax 回调函数的三个属性。
- parser：可选属性，该参数指定解析服务器响应的解析器，默认采用 ResponseHtmlParser 解析器；如果使用 XML 相应，通常指定为 ResponseXmlParser 即可。

值得注意的是该标签的响应，它的响应一样是个标准的 AjaxTags 响应，该响应包含的 item 节点数，应与需要动态计算的表单域的数量相等。而且每个 item 节点的 name 节点值应与目标表单域 id 属性相同。

下面应用示范了通过一个初始价格，计算出五星级会员、四星级会员、三星级会员、二星级会员和一星级会员的会员价。计算打折价的 Servlet 代码如下：

程序清单：codes\15\15.4\TagsTest\WEB-INF\src\lee\CalDiscountServlet.java

```
public class CalDiscountServlet extends BaseAjaxServlet
{
    //重写 getXmlContent 方法，该方法返回的 XML 字符串作为响应
    public String getXmlContent(HttpServletRequest request
        , HttpServletResponse response)throws Exception
        {
        //price 为初始价
        double price = 0;
        //下面五个变量分别为不同级别会员的打折价
        double five = 0;
        double four = 0;
        double three = 0;
        double two = 0;
        double one = 0;
        //获取请求参数
        price = Double.parseDouble(request.getParameter("price"));
        //调用服务器计算
        five = price * 0.7;
        four = price * 0.8;
        three = price * 0.85;
        two = price * 0.9;
        one = price * 0.95;
        //构造响应的 XML 字符串，并返回
        return new AjaxXmlBuilder()
            .addItem("five", Double.toString(five))
            .addItem("four", Double.toString(four))
            .addItem("three", Double.toString(three))
            .addItem("two", Double.toString(two))
            .addItem("one", Double.toString(one))
            .toString();
    }
}
```

看到上面代码中 addItem 的两个参数，第一个参数分别为"five"、"four"等，这些参数并不是随意填写的，这些参数应与页面中需要通过服务器计算的表单域的 id 属性相同。即页面中的 five 表单域的值就等于 Double.toString(five)。

页面中使用 formupdate 标签来完成该 Ajax 交互，因为同时有 5 个表单域需要通过计算得到，因此 target 的值为以逗号（,）隔开的 5 个值。下面是页面中使用 formupdate 的代码片段：

程序清单：codes\15\15.4\TagsTest\updateField.jsp

```
请输入初始价格:<input type="text" id="price" name="price" /><br/>
<input id="action" type="button" value="计算"/><br />
五星级会员价：<input type="text" id="five"/><br />
四星级会员价：<input type="text" id="four" name="four"/><br />
三星级会员价：<input type="text" id="three" name="three"/><br />
二星级会员价：<input type="text" id="two" name="two"/><br />
一星级会员价：<input type="text" id="one" name="one"/><br />
<ajax:updateField
    baseUrl="calDiscount"
    source="price"
    target="five,four,three,two,one"
    action="action"
    parameters="price={price}"
    parser="new ResponseXmlParser()"/>
```

在页面中的初始价格文本框中输入 80，然后单击“计算”按钮，将出现如图 15.12 所示的界面。

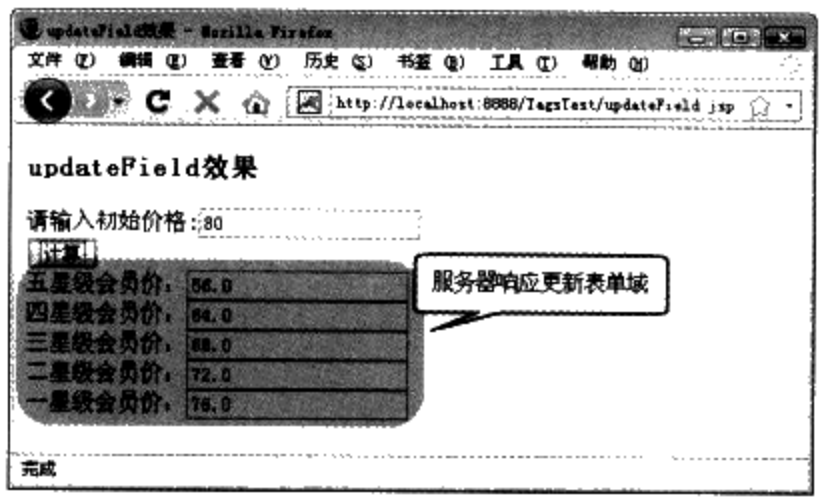


图 15.11　服务器响应更新表单域

上面介绍的这些是 AjaxTags 提供的常用标签，除此之外，AjaxTags 还提供 ajax:editor、ajax:toggle 两个标签，这两个标签的用法比较简单，故此处不再赘述。

15.5　关于 AjaxTags 的选择

正如前面介绍的，通过使用 AjaxTags 标签，完成一个 Ajax 应用是如此简单，对于常见的 Ajax 应用场景，AjaxTags 都提供了对应的封装，程序开发者只需要使用 JSP 标签，即可开发出功能完备的 Ajax 应用，但 AjaxTags 并不是万能的，有些时候，我们必须放弃 AjaxTags。选择更繁琐的开发方式。

➤➤15.5.1　AjaxTags 的优势和使用场景

AjaxTags 的优势相当明显，当 Ajax 技术开始面世时，笔者听到一种说法：Ajax 是对程序员的折磨，从而取悦用户。这种说法，在某种程度上是对的，但所有的技术都将以提高用户体验为最终目标，对于一个程序员而言，能带给用户更好的体验，就是最大的成就。

但 Ajax 技术的繁琐不言而喻，JavaScript 本身不是一门严格的语言，缺乏严格的调试机制，即使在底层所有响应完成后，程序员还必须在表现层完成异常繁琐的 DOM 更新，还必须应用 CSS 样式。如果在加上跨浏览器支持，向后兼容性等一系列的技巧，开发一个普通的 Ajax 页面就需要大量时间。

虽然已经有了大量 JavaScript 库，例如 Prototype.js 和 JQuery 等。即使使用这些 JavaScript 库，我们依然需要面对很多问题，我们依然需要动态更新 DOM，依然必须书写大量的 JavaScript 代码。

实际上，大量的 Ajax 应用场景重复出现，级联列表框、自动完成、页面提示……每个常用的 Ajax 交互需要我们花费大量的时间和精力。

AjaxTags 对这些常用的 Ajax 交互场景提供了包装，程序开发者几乎无需书写任何的 JavaScript 代码就可以完成一个专业的 Ajax 应用。特别是对于 Java EE 应用开发者，书写一个传统的 Servlet，并将该 Servlet 配置在 Web 应用中，然后在页面中使用 Ajax 标签即可完成一个 Ajax 应用，相当简单。

AjaxTags 最大的有点是简单，Java EE 应用开发者甚至可以无需了解 Ajax 技术细节，只需要会编写 Servlet，只需要会使用 JSP 标签，就可以开发出专家级的 Ajax 应用，这是 AjaxTags 提供的最大好处。AjaxTags 可以大大节省开发者的时间。

对于所有能使用 AjaxTags 情况，笔者推荐优先考虑使用 AjaxTags。毕竟使用 AjaxTags 既可以节省时间，也可以避免错误。AjaxTags 的更新非常快，经常有新的标签加入，即使每个 beta 版之间的标签数量，标签以用法也存在差异，希望读者在使用 AjaxTags 时，到官方站点看看，AjaxTags 的最新版包含了那些简便的标签。

➤➤15.5.2 AjaxTags 的缺点

AjaxTags 以简单、快捷的特性，方便了 Java EE 的 Ajax 开发者，但它也不是万能的，在某些情形下，它依然存在一些小小的缺陷。大致上 AjaxTags 存在如下缺陷：

- AjaxTags 只能在 Java EE 应用环境下使用，不能在其他的 Web 编程脚本，如 ASP、PHP 等脚本语言上使用。
- AjaxTags 的高度封装，虽然简化了 Ajax 的开发，但导致灵活性严重丧失，对于复杂的 Ajax 交互，使用 AjaxTags 完成更加繁琐。
- AjaxTags 对 Ajax 交互提供了封装，但没有提供了调试环境。整个 Ajax 交互不仅对普通浏览者透明，对于应用开发者也是透明的，调试难度相对较大。

虽然存在这些缺点，但 AjaxTags 的简单可以远远掩盖这些缺陷，对于能使用 AjaxTags 标签的地方，应该尽量考虑使用 AjaxTags。但如果需要对 AjaxTags 进行大量扩展，修改，则应该考虑使用其他技术。毕竟，AjaxTags 与其他 Ajax 技术并不是互斥的，例如 Prototype.js，本身就 AjaxTags 所依赖的技术。

15.6 本章小结

本章详细介绍了 AjaxTags 的使用，本章从 AjaxTags 的安装开始介绍，详细介绍 AjaxTags 所需的响应格式，并介绍了 AjaxTags 的处理类的编写和配置，以及 AjaxTags 处理类的几种形式。本章重点介绍了 AjaxTags 的常用标签，每个标签都详细介绍了它的用途和用法，并详细说明了各标签里所支持的全部属性，并结合示例程序示范了如何使用这些标签。本章最后还分析了 AjaxTags 标签的优点、缺点，及其合适的应用场景。

至此，本书关于 Ajax 的框架部分也介绍完成，关于这些框架的选择，就是“运用之妙，存乎一心”的事情了，笔者不能强行指定用户应该使用哪个框架，放弃哪个框架。但就笔者的经验而看， 通常需要使用一个专业的 Ajax 框架为主，以 JavaScript 库为辅是一个不错的选择。

CHAPTER
16

第 16 章
Ajax 实例：简易 Blog 系统

本章要点

- 实现 Hibernate 持久化类
- 实现系统 DAO 组件
- 在 Spring 容器中部署 DAO 组件
- 实现业务逻辑组件
- 部署业务逻辑组件
- 为业务逻辑方法配置事务管理
- 使用 DWR 暴露 Spring 容器中的 Bean
- 处理发布新 Blog、回复
- 处理查看 Blog 内容
- 处理获取 Blog 列表、控制分页
- 处理查看 Blog 回复、控制分页

本章将介绍一个基于 Ajax 技术的 Blog 系统，这个系统允许发表 Blog 文章，允许发表回复。本系统并不是一个完善的 Blog 系统，其功能非常简单：只能添加 Blog 文章和发表回复。而且本系统没有进行任何权限控制，任何用户都可以在上面发表 Blog 文章。笔者将在下一章的综合应用中介绍更完善的权限控制，让 Ajax 和 Spring 的 AOP 结合，提供用户的权限控制。读者可在本应用的基础上进一步完善，从而提供更丰富的功能。

该 Blog 系统的后台系统依然用 Hibernate 作为持久层的 ORM 框架，使用 Spring 作为中间层的 IoC 容器，将各组件以松耦合的方法组织在一起。DAO 对象的实现采用 Spring 的 DAO 支持。Ajax 引擎使用了 DWR，让 DWR 直接访问 Spring 容器中的业务逻辑组件，DWR 的这个功能非常实用，它可以利用 Spring 容器的强大功能。

16.1 实现 Hibernate 持久层

与前面的 Java EE 应用类似，本章一样采用 Hibernate 作为应用的持久层，从而可使用面向对象的方式操作关系数据库。

➤➤16.1.1 设计 Hibernate 的持久化类

本应用包含两个实体类：Blog 文章和文章回复，因为本应用没有处理用户管理，因此无须用户登录、用户权限检查等功能，所以本系统无须保存用户实体。如果需要扩展本系统，则可以考虑增加用户实体，那样就需要处理用户相关功能。

本系统设计的两个实体类保存了各自必需的信息，例如文章标题、文章内容等。Blog 文章回复则保存了回复者的用户名、回复内容、回复者的电子邮件等信息。

两个实体存在 1:N 关联，即一篇 Blog 文章对应 N 个回复。Hibernate 以面向对象的方式操作数据库，因此它可以理解这种关联。Blog 文章对应实体类的代码如下：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\model\Blog.java

```
public class Blog
{
    //标识属性
    private Integer id;
    //标题
    private String title;
    //内容
    private String content;
    //添加时间
    private Date addTime;
    //该 Blog 关联的全部评论
    private Set<Comment> comments
        = new HashSet<Comment>();
    //无参数的构造器
    public Blog(){}
    //初始化全部属性的构造器
    public Blog(int Integer , String title , String content , Date addTime)
    {
        this.id = id;
        this.title = title;
        this.content = content;
        this.addTime = addTime;
    }
    //省略普通属性的 setter 和 getter 方法
    ...
    //comments 属性的 setter 和 getter 方法
```

```
    public void setComments(Set<Comment> comments)
    {
        this.comments = comments;
    }
    public Set<Comment> getComments()
    {
        return this.comments;
    }
}
```

上面的 Blog 类中包含了一个 Set 类型的属性，该属性保留了该 Blog 文章所关联的一系列的评论实体 Comment。对于一篇 Blog 文章而言，Blog 文章可以包含多个回复，即存在 1:N 关联。由于我们已经在 Blog 类中提供了 getComments()方法，因此可以通过 Blog 实例直接获取该 Blog 对应的所有回复。

因为篇幅关系，上面的持久化类省略了 Blog 文章其他属性的 getter 和 setter 方法，这些 getter 和 setter 方法都非常简单，相信读者可以将这些方法补齐。Blog 文章回复对应的持久化类的代码如下：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\model\Comment.java

```
public class Comment
{
    //标识属性
    private Integer id;
    //发表评论的用户
    private String user;
    //发表评论的用户的 Email
    private String email;
    //发表评论的用户的 URL
    private String url;
    //评论内容
    private String content;
    //评论添加时间
    private Date addTime;
    //评论所关联的 Blog
    private Blog blog;
    //无参数的构造器
    public Comment(){}
    //初始化全部属性的构造器
    public Comment(int Integer , String user , String email ,
        String url , String content , Date addTime , Blog blog)
    {
        this.id = id;
        this.user = user;
        this.email = email;
        this.url = url;
        this.content = content;
        this.addTime = addTime;
        this.blog = blog;
    }
    //省略其他属性的 setter 和 getter 方法
    ...
    //blog 属性的 setter 和 getter 方法
    public void setBlog(Blog blog)
    {
        this.blog = blog;
    }
    public Blog getBlog()
    {
        return this.blog;
    }
}
```

Comment 和 Blog 之间存在 N:1 的关联关系，即每篇回复都有一篇与之对应的 Blog 文章，回复可以通过这种关联关系直接访问与之对应的 Blog 文章。在上面的代码中，回复持久化类里保存了一个 Blog 类型的属性，该属性就是该 Comment 所对应的 Blog 文章。

定义了上面两个 POJO 类之后，还需要为它们提供相应的映射文件。

➤➤16.1.2 完成映射文件

仅有上面所定义的 POJO 还不足以访问持久层数据库，Hibernate 需要 XML 映射文件，该映射文件定义了 POJO 类和数据表的对应关系。当开发者以面向对象的方式操作 POJO 时，Hibernate 通过这种对应关系将这种面向对象的操作转换为对数据库的操作。

本应用的 Blog 文章和回复之间是典型的 1:N 关联，Hibernate 完全可以理解关联映射。对于 1:N 关联，建议采用性能更好的双向关联，而且让 N 的一端来控制关联关系。1 的一端只能访问 N 的一端，但不能控制关联关系。

下面是 Comment 持久化类的映射文件，这个映射文件增加了<many-to-one.../>元素来映射关联实体：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\model\Comment.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.blog.model">
    <!-- 每个 class 元素映射一个持久化类 -->
    <class name="Comment" table="comment table">
        <!-- 映射标识属性 -->
        <id name="id" column="comment_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="user" type="string"/>
        <property name="email" type="string"/>
        <property name="url" type="string"/>
        <property name="content" type="text"/>
        <property name="addTime" type="java.util.Date" column="add_time"/>
        <!-- 映射该评论关联的 Blog 文章 -->
        <many-to-one name="blog" class="Blog"
            column="blog_id" not-null="true"/>
    </class>
</hibernate-mapping>
```

上面的映射文件完成了 Comment 类的映射，在这个映射文件中，<many-to-one.../>元素（粗体字部分）映射了该 Comment 对应的 Blog 实体。

注意：

因为我们需要让 addTime 属性映射成 MySQL 数据库中 datetime 类型的数据列，而 Hibernate 的 type 属性值不支持 datetime 值，故映射 addTime 属性时指定 type="java.util.Date" 属性，这样可直接指定该列数据类型是 datetime。

下面是 Blog 的映射文件，因为每篇 Blog 可对应多个 Comment 实体，因此 Blog 的映射文件中应增加<set.../>元素来映射关联实体，表明每个 Blog 可以关联多个 Comment 实体。

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\model\Blog.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.blog.model">
    <!-- 每个 class 元素映射一个持久化类 -->
    <class name="Blog" table="blog_table">
        <!-- 映射标识属性 -->
        <id name="id" column="blog_id">
            <!-- 指定主键生成器策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="title" type="string"/>
        <property name="content" type="text"/>
        <property name="addTime" type="java.util.Date" column="add_time"/>
        <!-- 映射该 Blog 文章关联的全部评论 -->
        <set name="comments" inverse="true">
            <!-- 映射外键列 -->
            <key column="blog_id"/>
            <one-to-many class="Comment"/>
        </set>
    </class>
</hibernate-mapping>
```

在进行双向的 1:N 关联时，两边都应该指定外键列的列名。因为两边指定的外键列实际上是同一列，因此两边指定的列名应该相等。对于 1:N 关联，通常不推荐使用 1 的一端控制关联，因此在上面的映射文件中，为 set 元素增加了 inverse=" true " 属性。

注意：

对于双向的 1:N 关联映射，两个映射文件的<many-to-one.../>元素和<key.../>元素都需要指定 column 属性，而且两个 column 属性必须相同，因为它们实际上都是映射从表的外键列。

完成上面的映射后，Hibnerate 就可以理解数据表和 POJO 之间的对应关系了，但 Hibernate 还不知道连接哪个数据库，以及连接数据库的 URL、用户名、密码等全局属性——因为还没有为 Hibernate 配置 SessionFactory 属性。

本应用将使用 Spring 管理所有的 DAO 组件，因此将 Hibernate 的 SessionFactory 放在 Spring 容器中管理，即在 Spring 配置文件中增加如下片段：

程序清单：codes\16\blog\WEB-INF\applicationContext.xml

```
<!-- 定义数据源 Bean，使用 C3P0 数据源实现 -->
<bean id="dataSource" destroy-method="close"
    class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- 指定连接数据库的驱动 -->
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <!-- 指定连接数据库的 URL -->
    <property name="jdbcUrl"
        value="jdbc:mysql://localhost:3306/blog"/>
    <!-- 指定连接数据库的用户名 -->
    <property name="user" value="root"/>
    <!-- 指定连接数据库的密码 -->
    <property name="password" value="32147"/>
    <!-- 指定连接数据库连接池的最大连接数 -->
    <property name="maxPoolSize" value="40"/>
    <!-- 指定连接数据库连接池的最小连接数 -->
```

```
        <property name="minPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的初始化连接数 -->
        <property name="initialPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的连接的最大空闲时间 -->
        <property name="maxIdleTime" value="20"/>
    </bean>
    <!-- 定义 Hibernate 的 SessionFactory -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- 依赖注入数据源，注入上面定义的 dataSource -->
        <property name="dataSource" ref="dataSource"/>
        <!-- mappingResouces 属性用来列出全部映射文件 -->
        <property name="mappingResources">
            <list>
                <!-- 以下用来列出 Hibernate 映射文件 -->
                <value>org/crazyjava/blog/model/Blog.hbm.xml</value>
                <value>org/crazyjava/blog/model/Comment.hbm.xml</value>
            </list>
        </property>
        <!--定义 Hibernate 的 SessionFactory 的属性-->
        <property name="hibernateProperties">
            <props>
                <!-- 指定数据库方言 -->
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLInnoDBDialect</prop>
                <!-- 是否根据需要每次自动创建数据库 -->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <!-- 显示 Hibernate 持久化操作所生成的 SQL -->
                <prop key="hibernate.show_sql">true</prop>
                <!-- 将 SQL 脚本进行格式化后再输出 -->
                <prop key="hibernate.format_sql">true</prop>
            </props>
        </property>
    </bean>
```

经过上面的配置，Hibernate 可以理解系统需要操作的数据库，以及连接数据库的 URL、用户名、密码等全局属性；通过使用前面的映射文件，Hibernate 也可以理解 POJO 和数据表之间的对应关系。这样就允许我们在 Java 程序里以面向对象的方式来进行持久化访问。

➤➤16.1.3 数据表的结构

对于采用 OOA、OOD 这种思路完成的应用，数据表的结构通常无须开发者手动建立，开发者只要设计了对应的 POJO，并定义了合适的映射文件，系统就可以自动生成所需的数据表，包括数据表的主、外键约束。

本应用中有两个实体类，两个实体类之间使用无连接表的 1:N 映射策略，系统将根据持久化类生成两个表，分别用于存放 Blog 文章和文章回复。

图 16.1 是用于存放 Blog 文章的表结构。

Blog 是 1:N 关联中 1 的一端，如果采用主从表表示这种关联关系，则 Blog 对应的数据表就是主表，该表中无须保留外键，外键将保存在从表一端。图 16.2 是 Comment 对应数据表的表结构。

```
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| blog_id   | int(11)      | NO   | PRI | NULL    | auto_increment |
| title     | varchar(255) | YES  |     | NULL    |                |
| content   | text         | YES  |     | NULL    |                |
| add_time  | datetime     | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
```

图 16.1 Blog 对应的表结构

```
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| comment_id | int(11)      | NO   | PRI | NULL    | auto_increment |
| user       | varchar(255) | YES  |     | NULL    |                |
| email      | varchar(255) | YES  |     | NULL    |                |
| url        | varchar(255) | YES  |     | NULL    |                |
| content    | text         | YES  |     | NULL    |                |
| add_time   | datetime     | YES  |     | NULL    |                |
| blog_id    | int(11)      | NO   | MUL |         |                |
+------------+--------------+------+-----+---------+----------------+
```

图 16.2 Comment 对应的表结构

两个表之间通过主、外键建立约束，comment_table 表中增加了 blog_id 列，该列参照 blog_table 表的 id 列。

16.2 实现 DAO 组件

本应用的持久层访问也是由 DAO 组件来完成的，DAO 组件封装了所有的数据库访问，让数据库访问与业务逻辑组件分离，从而将数据库访问逻辑和业务逻辑分离。本系统有两个 DAO 组件，这两个 DAO 组件都需要借助于 Spring 的 HibernateDaoSupport。DAO 接口里提供了各种方法定义，DAO 实现类则对这些方法提供了实现。

➤➤16.2.1 DAO 接口定义

Blog DAO 组件负责 Blog 文章的数据库访问操作，包括 Blog 记录的增加、删除、修改和查询等操作，下面是 Blog DAO 组件的代码：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\dao\BlogDao.java

```
public interface BlogDao
{
    /**
     * 根据主键加载 Blog
     * @param id 需要加载的 Blog ID
     * @return 加载的 Blog
     */
    Blog get(int id);
    /**
     * 保存 Blog
     * @param b 需要保存的 Blog
     */
    void save(Blog b);
    /**
     * 删除 Blog
     * @param b 需要删除的 Blog
     */
    void delete(Blog b);
    /**
     * 删除 Blog
     * @param id 需要删除的 Blog ID
     */
    void delete(int id);
    /**
     * 修改 Blog
     * @param b 需要修改的 Blog
     */
    void update(Blog b);
    /**
     * 查询指定页的 Blog
     * @param pageNo 需要查询的页码
     * @return 查询到的 Blog 集合
     */
    List findAllByPage(int pageNo);
    /**
     * 查询最新 Blog 文章
     * @return 最新 Blog 文章
     */
    Blog findLastest();
}
```

正如前面所见，BlogDao 接口里定义了一系列基本的 CRUD 方法，用于完成 Blog 记录的增加、删除、修改等操作。除此之外，还定义了两个简单的查询方法：findAllByPage()和 findLastest()，这两个方法分别用于查询所有的 Blog 文章和最新的 Blog 文章。查询所有的 Blog 文章时，提供了一个页码参数，用于控制分页。Comment DAO 接口的代码如下：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\dao\CommentDao.java

```
public interface CommentDao
{
    /**
     * 根据主键加载评论
     * @param id 需要加载的评论 ID
     * @return 加载的评论
     */
    Comment get(int id);
    /**
     * 保存评论
     * @param c 需要保存的评论
     */
    void save(Comment c);
    /**
     * 删除评论
     * @param c 需要删除的评论
     */
    void delete(Comment c);
    /**
     * 删除评论
     * @param id 需要删除的评论的 ID
     */
    void delete(int id);
    /**
     * 修改评论
     * @param c 需要修改的评论
     */
    void update(Comment c);
    /**
     * 根据 Blog ID 和页码来查找评论
     * @param blogId 评论所对应 Blog 文章的 ID
     * @param pageNo 查找指定页的 Blog
     * @return 查找到的评论集合
     */
    List findByBlogAndPage(int blogId , int pageNo);
}
```

CommentDao 接口里一样定义了增加、删除、修改回复的方法。除此之外，还定义了一个普通查询方法：findByBlogAndPage()，该方法根据 Blog ID 页码查询回复。

➤➤16.2.2 实现 DAO 组件

本系统的查询方法需要进行分页，Spring 的 HibenateTemplate 并未提供分页的实现。因此，笔者扩展了 Spring 的 HibernateDaoSupport 类，扩展 HibernateDaoSupport 得到的 YeekuHibernateDaoSupport 类与第 14 章提供的 YeekuHibernateDaoSupport 类完全相同，故此处不再赘述。

DAO 实现类继承了 YeekuHibernateDaoSupport 类，实现了各自的 DAO 接口。下面是 BlogDaoHibernate 实现类的代码：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\dao\impl\BlogDaoImpl.java

```
public class BlogDaoHibernate
    extends YeekuHibernateDaoSupport implements BlogDao
```

```
{
    //每页显示的消息数
    private int pageSize;
    //每页的消息数通过依赖注入管理
    public void setPageSize(int pageSize)
    {
        this.pageSize = pageSize;
    }
    /**
     * 根据主键加载 Blog
     * @param id 需要加载的 Blog ID
     * @return 加载的 Blog
     */
    public Blog get(int id)
    {
        return (Blog)getHibernateTemplate().get(Blog.class , new Integer(id));
    }
    /**
     * 保存 Blog
     * @param b 需要保存的 Blog
     */
    public void save(Blog b)
    {
        getHibernateTemplate().save(b);
    }
    /**
     * 删除 Blog
     * @param b 需要删除的 Blog
     */
    public void delete(Blog b)
    {
        getHibernateTemplate().delete(b);
    }
    /**
     * 删除 Blog
     * @param id 需要删除的 Blog 的 ID
     */
    public void delete(int id)
    {
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 修改 Blog
     * @param b 需要修改的 Blog
     */
    public void update(Blog b)
    {
        getHibernateTemplate().saveOrUpdate(b);
    }
    /**
     * 查询指定页的 Blog
     * @param pageNo 需要查询的页码
     * @return 查询到的 Blog 集合
     */
    public List findAllByPage(int pageNo)
    {
        int offset = (pageNo - 1) * pageSize;
        //返回分页查询的结果
        return findByPage("from Blog b order by b.id desc"
            , offset , pageSize);
```

```
    }
    /**
     * 查询最新 Blog 文章
     * @return 最新 Blog 文章
     */
    public Blog findLastest()
    {
        List l = getHibernateTemplate()
            .find("from Blog b where b.id >= (select max(b.id) from Blog as b)");
        //返回该集合里第一个 Blog 对象
        if (l != null && l.size() > 0 )
        {
            return (Blog)l.get(0);
        }
        return null;
    }
}
```

在上面的实现类中，除实现了 DAO 接口中定义的一系列方法外，还提供了一个依赖注入方法，这个依赖注入方法用于注入一个 pageSize 变量值，这个变量值就是每页显示的记录数。

CommentDaoHibernate 实现类与之类似，也提供了每页显示记录数的依赖注入方法。下面是 CommentDaoHibernate 实现类的代码：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\dao\impl\CommentDaoImpl.java

```
public class CommentDaoHibernate
    extends YeekuHibernateDaoSupport implements CommentDao
{
    //每页显示的消息数
    private int pageSize;
    //每页的消息数通过依赖注入管理
    public void setPageSize(int pageSize)
    {
        this.pageSize = pageSize;
    }
    /**
     * 根据主键加载评论
     * @param id 需要加载的评论的 ID
     * @return 加载的评论
     */
    public Comment get(int id)
    {
        return (Comment)getHibernateTemplate().get(Comment.class , new Integer(id));
    }
    /**
     * 保存评论
     * @param c 需要保存的评论
     */
    public void save(Comment c)
    {
        getHibernateTemplate().save(c);
    }
    /**
     * 删除评论
     * @param c 需要删除的评论
     */
    public void delete(Comment c)
    {
        getHibernateTemplate().delete(c);
    }
```

```
    /**
     * 删除评论
     * @param id 需要删除的评论的 ID
     */
    public void delete(int id)
    {
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 修改评论
     * @param c 需要修改的评论
     */
    public void update(Comment c)
    {
        getHibernateTemplate().saveOrUpdate(c);
    }
    /**
     * 根据 Blog ID 和页码来查找评论
     * @param blogId 评论所对应 Blog 文章的 ID
     * @param pageNo 需要查询的页码
     * @return 查找到的评论集合
     */
    public List findByBlogAndPage(int blogId , int pageNo)
    {
        int offset = (pageNo - 1) * pageSize;
        return findByPage("from Comment c where c.blog.id = ? order by c.id desc"
            , blogId , offset, pageSize);
    }
}
```

提供了这两个 DAO 组件的实现类后，必须将它们配置在 Spring 容器中。因为 HibernateDaoSupport 已经提供了 setSessionFactory 方法，这个方法用于为 DAO 组件依赖注入 SessionFactory 的引用，这两个 DAO 组件一旦获得了 SessionFactory，就具有了对数据库进行操作的能力。Spring 的 IoC 容器将负责为 DAO 组件注入 SessionFactory 引用。

➤➤16.2.3 配置 DAO 组件

提供了两个 DAO 组件的实现类后，将它们配置在 Spring 容器中，让 Spring 容器为其注入 SessionFactory 的引用，并将 DAO 组件注入到业务逻辑组件中。通过这种依赖注入，可以降低 DAO 组件和业务逻辑组件之间的耦合，从而允许业务逻辑组件与 DAO 组件彻底分离。

为了让 Spring 容器管理 DAO 组件的依赖关系，我们必须将 DAO 组件部署在 Spring 容器中，部署 DAO 组件的配置片段如下：

程序清单：codes\16\blog\WEB-INF\applicationContext.xml

```
<!-- 配置 BlogDao 组件 -->
<bean id="blogDao"
    class="org.crazyjava.blog.dao.impl.BlogDaoHibernate">
    <!-- 注入 SessionFactory 引用 -->
    <property name="sessionFactory" ref="sessionFactory"/>
    <!-- 指定每页显示的 Blog 数量 -->
    <property name="pageSize" value="3"/>
</bean>
<!-- 配置 CommentDao 组件 -->
<bean id="commentDao"
    class="org.crazyjava.blog.dao.impl.CommentDaoHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
    <property name="pageSize" value="3"/>
</bean>
```

前面在介绍 Hibernate 配置时已经配置了 SessionFactory，故此处配置 DAO 组件时可直接为之注入 SessionFactory 实例。除此之外，上面的配置文件中的粗体字代码还为 DAO 组件指定了每页显示的 Blog、Comment 数目。

16.3 实现业务逻辑组件

系统 DAO 组件开发完成之后，程序就可以开始在 DAO 组件基础上实现业务逻辑组件了，系统的业务逻辑组件只集中实现业务功能，数据访问的功能则交给 DAO 组件完成，这样就可将业务逻辑层与底层的数据访问分离。业务逻辑组件一样需要接口、实现类，并将其配置在 Spring 容器中。

➤➤16.3.1 业务逻辑组件的接口

业务逻辑组件里提供业务逻辑方法的定义，每个系统有多少个业务逻辑方法完全由业务需要决定。对本系统而言，大致包含如下业务逻辑方法：

- 发表 Blog 文章。
- 获取指定页的所有 Blog 文章。
- 发表回复。
- 根据指定 Blog 文章和页码获取回复列表。
- 根据 Blog ID 查看指定 Blog 的详细信息。
- 获取最新的 Blog 文章。

通过 BlogManager 接口定义上面的一系列业务逻辑方法，下面是该接口的代码：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\service\BlogManager.java

```
public interface BlogManager
{
    /**
     * 创建一篇新的 Blog
     * @param title Blog 的标题
     * @param content Blog 的内容
     * @return 新创建 Blog 的主键，如果创建失败，返回-1
     */
    int createBlog(String title , String content)
        throws BlogException;
    /**
     * 创建一个评论
     * @param user 发表评论的用户
     * @param email 发表评论用户的 Email
     * @param url 发表评论用户的 URL
     * @param content 发表评论的内容
     * @param blogId 发表评论所对应的 Blog ID
     * @return 新发表评论的主键
     */
    int createComment(String user, String email, String url,
                String content, int blogId)throws BlogException;
    /**
     * 返回指定页的 Blog 列表
     * @param pageNo 指定页面页码
     * @return 指定页的 Blog 列表
     */
    List<BlogBean> getAllBlogByPage(int pageNo)
        throws BlogException;
    /**
     * 返回指定 Blog 指定页所对应的评论
     * @param blogId 指定 Blog 的 ID
```

```
    * @param pageNo 指定页面页码
    * @return 指定 Blog 指定页码的所有评论
    */
    List<CommentBean> getCommentsByBlogAndPage(int blogId , int pageNo)
        throws BlogException;
    /**
    * 返回指定 Blog 文章
    * @param blogId 需要查询的 Blog 文章的 ID
    * @return blogId 对应的 Blog 文章
    */
    BlogBean getBlog(int blogId)
        throws BlogException;
    /**
    * 返回 ID 最大的 Blog 文章
    * @return ID 最大的 Blog 文章
    */
    BlogBean getNewestBlog()
        throws BlogException;
}
```

BlogManager 接口仅仅提供方法定义，并不提供方法的实现。业务逻辑接口仅仅指定了业务逻辑组件应该完成哪些事情，没有为这些业务逻辑功能提供具体实现，这些功能的具体实现由实现类完成。

16.3.2 业务逻辑组件的实现类

业务逻辑组件仅仅提供了业务逻辑方法的实现，不再包含数据库访问逻辑，数据库访问逻辑由 DAO 组件提供。因为业务逻辑功能的实现需要依赖于 DAO 组件，DAO 组件由 Spring IoC 容器注入到业务逻辑组件中，因此业务逻辑组件的实现类必须提供对应的 setter 方法，下面是业务逻辑组件实现类的代码：

程序清单：codes\16\blog\WEB-INF\src\org\crazyjava\blog\service\impl\BlogManagerImpl.java

```
public class BlogManagerImpl implements BlogManager
{
    private BlogDao blogDao;
    private CommentDao commentDao;
    //依赖注入业务逻辑组件所必需的 setter 方法
    public void setBlogDao(BlogDao blogDao)
    {
        this.blogDao = blogDao;
    }
    public void setCommentDao(CommentDao commentDao)
    {
        this.commentDao = commentDao;
    }
    /**
    * 创建一篇新的 Blog
    * @param title Blog 的标题
    * @param content Blog 的内容
    * @return 新创建 Blog 的主键，如果创建失败，返回-1
    */
    public int createBlog(String title , String content)
        throws BlogException
    {
        try
        {
            Blog b = new Blog();
            b.setTitle(title);
            b.setContent(content);
            b.setAddTime(new Date());
```

```
            blogDao.save(b);
            return b.getId();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new BlogException("保存 Blog 文章出错");
        }
    }
    /**
     * 创建一个评论
     * @param user 发表评论的用户
     * @param email 发表评论的用户的 Email
     * @param url 发表评论的用户的 URL
     * @param content 发表评论的内容
     * @param blogId 发表评论所对应的 Blog ID
     * @return 新发表评论的主键
     */
    public int createComment(String user, String email, String url,
            String content, int blogId)throws BlogException
    {
        Blog b = blogDao.get(blogId);
        if (b == null)
        {
            return -1;
        }
        try
        {
            Comment c = new Comment();
            c.setUser(user);
            c.setEmail(email);
            c.setUrl(url);
            c.setContent(content);
            c.setAddTime(new Date());
            c.setBlog(b);
            commentDao.save(c);
            return c.getId();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new BlogException("保存文章评论出错");
        }
    }
    /**
     * 返回指定页的 Blog 列表
     * @param pageNo 指定页面页码
     * @return 指定页的 Blog 列表
     */
    public List<BlogBean> getAllBlogByPage(int pageNo)
        throws BlogException
    {
        List<BlogBean> result = new ArrayList<BlogBean>();
        try
        {
            List bl = blogDao.findAllByPage(pageNo);
            for (Object o : bl)
            {
                Blog b = (Blog)o;
                result.add(new BlogBean(b.getId() ,
                    b.getTitle() , null , null));
```

```
            }
            return result;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new BlogException("获取文章标题列表出错");
        }
    }
    /**
     * 返回指定 Blog 指定页所对应的评论
     * @param blogId 指定 Blog 的 ID
     * @param pageNo 指定页面页码
     * @return 指定 Blog 指定页码的所有评论
     */
    public List<CommentBean> getCommentsByBlogAndPage(int blogId , int pageNo)
        throws BlogException
    {
        try
        {
            List cl = commentDao.findByBlogAndPage(blogId , pageNo);
            List<CommentBean> result = new ArrayList<CommentBean>();
            for (Object o : cl)
            {
                Comment c = (Comment)o;
                result.add(new CommentBean(c.getUser() , c.getEmail() ,
                    c.getUrl() , c.getContent() , c.getAddTime()));
            }
            return result;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new BlogException("获取文章评论列表出错");
        }
    }
    /**
     * 返回指定 Blog 文章
     * @param blogId 需要查询的 Blog 文章的 ID
     * @return blogId 对应的 Blog 文章
     */
    public BlogBean getBlog(int blogId)
        throws BlogException
    {
        try
        {
            Blog b = blogDao.get(blogId);
            return new BlogBean(b.getId() , b.getTitle() ,
                b.getContent() , b.getAddTime());
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new BlogException("获取指定 Blog 文章出错");
        }
    }
    /**
     *返回 ID 最大的 Blog 文章
     * @return ID 最大的 Blog 文章
     */
    public BlogBean getNewestBlog()
```

```
        throws BlogException
    {
        try
        {
            Blog b = blogDao.findLastest();
            if (b != null)
            {
                return new BlogBean(b.getId() , b.getTitle() ,
                    b.getContent() , b.getAddTime());
            }
            return null;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new BlogException("获取最新的 Blog 文章出错");
        }
    }
}
```

提供了业务逻辑组件的实现类后，还需要将该业务逻辑组件部署在 Spring 容器中，由 Spring 容器负责创建业务逻辑组件的实例，并管理业务逻辑组件的依赖关系。

➤➤16.3.3 配置业务逻辑组件

配置业务逻辑组件应为其注入 DAO 组件，下面是配置业务逻辑组件的代码片段：

程序清单：codes\16\blog\WEB-INF\applicationContext.xml

```
<!-- 配置 BlogManager 业务逻辑组件 -->
<bean id="blogManager"
    class="org.crazyjava.blog.service.impl.BlogManagerImpl">
    <!-- 为业务逻辑组件注入 2 个 DAO 组件 -->
    <property name="blogDao" ref="blogDao"/>
    <property name="commentDao" ref="commentDao"/>
</bean>
```

除此之外，还应为业务逻辑组件配置事务管理，本应用采用 Spring 提供的 tx 和 aop 两个命名空间来配置事务管理。下面是为业务逻辑组件配置事务管理的代码片段：

程序清单：codes\16\blog\WEB-INF\applicationContext.xml

```
<!-- 配置 Hibernate 的局部事务管理器，使用 HibernateTransactionManager 类 -->
<!-- 该类实现了 PlatformTransactionManager 接口，是针对 Hibernate 的特定实现-->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 配置 HibernateTransactionManager 时需要依赖注入 SessionFactory 的引用 -->
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<!-- 配置事务切面 Bean，指定事务管理器 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 用于配置详细的事务语义 -->
    <tx:attributes>
        <!-- 所有以 get 开头的方法是 read-only 的 -->
        <tx:method name="get*" read-only="true"/>
        <!-- 其他方法使用默认的事务设置 -->
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
<aop:config>
```

```
    <!-- 配置一个切入点，匹配 lee 包下所有以 Impl 结尾的类执行的所有方法 -->
    <aop:pointcut id="leeService"
        expression="execution(* org.crazyjava.blog.service.impl.*Impl.*(..))"/>
    <!-- 指定在 leeService 切入点应用 txAdvice 事务切面 -->
    <aop:advisor advice-ref="txAdvice"
        pointcut-ref="leeService"/>
</aop:config>
```

经过这些配置，本系统的后台部分已经开发完成了系统的业务逻辑组件可以正常对外提供业务功能了。接下来就需要将业务逻辑方法暴露给浏览器，允许浏览器通过 JavaScript 来调用业务逻辑方法。

16.4 整合 DWR 框架

为了在 Web 应用中正常使用 Spring 框架，应让 Spring 容器随 Web 应用的启动而初始化，可以选择在 web.xml 文件中配置使用 Listener 来初始化 Spring 容器。除此之外，还需在 web.xml 文件中配置 DWR 的核心 Servlet。

➤➤16.4.1 配置 web.xml 文件

因为 Listener 比 load-on-startup Servlet 的启动时机更早，因此推荐使用 Listener 来初始化 Spring 容器。下面是配置 Spring 容器随 Web 应用启动而初始化的代码片段：

程序清单：codes\16\blog\WEB-INF\web.xml

```
<!-- 配置 Web 应用启动时加载 Spring 容器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

在 web.xml 文件中增加上面的配置片段，即可让 Spring 容器随 Web 应用的启动而初始化，Spring 初始化时将自动加载位于 WEB-INF 下的 applicationContext 配置文件。

为了在 Web 应用中整合 DWR 框架，必须在 web.xml 文件中配置 DWR 的核心 Servlet，让该 Servlet 负责拦截特定的请求。下面是定义 DWR 的核心 Servlet 的配置片段：

程序清单：codes\16\blog\WEB-INF\web.xml

```
<!-- 配置 DWR 的核心 Servlet -->
<servlet>
    <!-- 指定 DWR 核心 Servlet 的名字 -->
    <servlet-name>dwr-invoker</servlet-name>
    <!-- 指定 DWR 核心 Servlet 的实现类 -->
    <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
    <!--指定 DWR 核心 Servlet 处于调试状态 -->
    <init-param>
        <param-name>debug</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>
<!-- 指定核心 Servlet 的 URL 映射 -->
<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <!-- 指定核心 Servlet 映射的 URL -->
    <url-pattern>/leedwr/*</url-pattern>
</servlet-mapping>
```

在 web.xml 文件中增加上面的配置片段，表示 DWR 所暴露的 JavaScript 对象都位于 leedwr 下。视图页面的 JavaScript 代码必须导入位于 leedwr 下的 JavaScript 库。

➤➤16.4.2 将 Spring 容器中的 Bean 转化成 JavaScript 对象

为了将 Spring 容器中的 Bean 转化成 JavaScript 对象，需要在 dwr.xml 文件中使用 spring 创建器，该创建器负责将 Spring 容器中的 Bean 创建成一个浏览器里的 JavaScript 对象。

除此之外，配置文件中还定义了一系列需要暴露的方法，定义需要暴露的方法使用<include.../>元素来指定，这种指定方式是典型的白名单方式，只有被<include.../>元素列出的方法才会被暴露给浏览器。除此之外，本应用还需要利用 DWR 的 Bean 转换器来转换 BlogBean、CommentBean 和 BlogExcetpion，这些都需要在 dwr.xml 文件中进行配置，下面是本应用里 dwr.xml 文件的代码：

程序清单：codes\16\blog\WEB-INF\dwr.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
    "http://getahead.ltd.uk/dwr/dwr20.dtd">
<dwr>
    <allow>
        <!-- 使用 spring 创建器，创建一个名为 bm 的 JavaScript 对象 -->
        <create creator="spring" javascript="bm">
            <!-- 将 Spring 容器中的 blogManager 创建成名为 bm 的对象 -->
            <param name="beanName" value="blogManager"/>
            <!-- 使用 include 元素定义哪些方法将被暴露到客户端 -->
            <include method="createComment"/>
            <include method="createBlog"/>
            <include method="getAllBlogByPage"/>
            <include method="getCommentsByBlogAndPage"/>
            <include method="getBlog"/>
            <include method="getNewestBlog"/>
        </create>
        <!-- 定义使用 Bean 转换器处理如下 Java 类 -->
        <convert converter="bean"
            match="org.crazyjava.blog.vo.BlogBean"/>
        <convert converter="bean"
            match="org.crazyjava.blog.vo.CommentBean"/>
        <convert converter="bean"
            match="org.crazyjava.blog.exception.BlogExcetpion"/>
    </allow>
</dwr>
```

上面的配置文件使用 spring 创建器创建了一个 bm 对象，该对象由 Spring 容器中的 blogManager 创建而来。浏览器中的 JavaScript 代码可通过 bm 对象来访问 blogManager 组件的方法。

经过了上面的步骤，下面就可以在 JavaScript 代码中直接调用 bm 对象的方法，也就是调用 blogManager 的方法了。前面在配置 DWRServlet 时，指定了 DWR 处于调试状态，因此可以看到 DWR 的调试控制台。在浏览器中输入调试控制台的地址，将可看到如图 16.3 所示的界面。

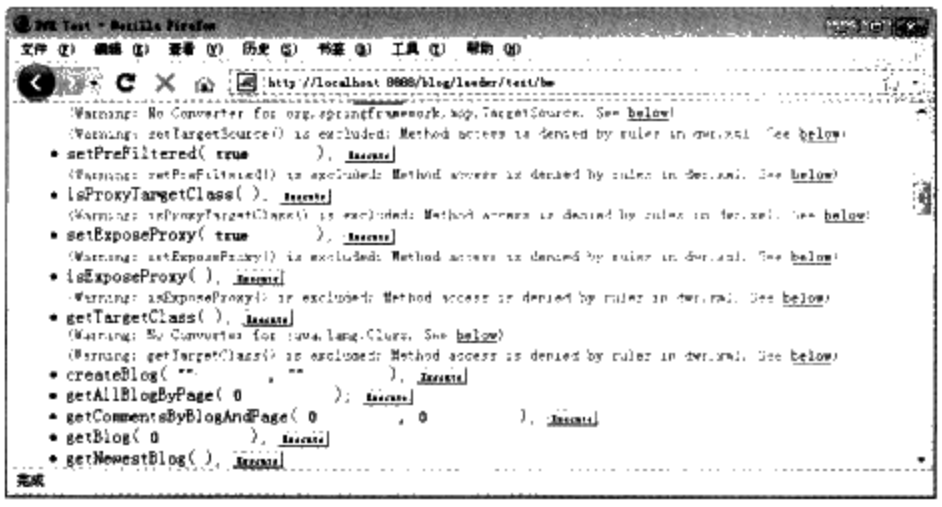


图 16.3 通过 DWR 控制台查看 bm 对象的方法

16.5 在客户端调用 JavaScript 对象

正如在图 16.3 中所见到的，bm 对象中包含了 5 个方法。该控制台还提示了如果需要使用 DWR 的 JavaScript 对象，应该在客户端导入哪些 JavaScript 代码库。在如图 16.3 所示页面的上部，有 3 个超级链接部分，其代码如下。

```
/blog/leedwr/interface/bm.js
/blog/leedwr/engine.js
/blog/leedwr/util.js
```

这表明应该在浏览器中的 JavaScript 代码中导入上面的三个 JavaScript 函数库。

➤➤16.5.1 获取 Blog 文章列表

为了在 HTML 页面左边显示 Blog 文章列表，必须调用业务逻辑组件的 getAllBlogByPage()方法，该方法根据指定页码获取 Blog 文章列表。

为了控制 Blog 文章列表的翻页，系统将 Blog 文章列表的当前页设置为一个全局变量。获取当前页的全部 Blog 文件的函数如下：

程序清单：codes\16\blog\bm.js

```
//获取所有的 Blog 文章
function getBlogList()
{
    bm.getAllBlogByPage(curBlogPage , blCb);
}
```

其中的 getAllBlogByPage()函数中有一个 curBlogPage 参数，该参数是一个全局变量，保存了用户正在浏览的当前页。blCb 是获取 Blog 文章列表的回调函数，该回调函数用于将 Blog 文章加载到页面中显示。下面是回调函数的代码：

程序清单：codes\16\blog\bm.js

```
//获取所有的 Blog 文章列表的回调函数
function blCb(data)
{
    //因为该函数还作为翻页函数的回调代码，所以需要判断是否已经是最后一页
    //如果当前页码不是第一页，而且当前页没有记录
    if ((curBlogPage > 1 ) &&
        (data == null || data == "undefined" || data.length < 1))
    {
        alert("已经是最后一页了，无法向后翻页");
        //自动回到上一页
        curBlogPage--;
        return false;
    }
    //当前页没有记录
    else if (data == null || data == "undefined" || data.length < 1)
    {
        alert("暂时还没有任何 Blog 文章");
        return false;
    }
    else
    {
        //用于获取显示 Blog 文章列表的 HTML 元素
        var listElement = $("blogList");
        //清空原有的 Blog 文章列表
        listElement.innerHTML="";
        //遍历 getAllBlogByPage 函数返回的每条记录
```

```
            for (var i = 0 ; i < data.length ; i++ )
            {
                //对于每条记录都需创建一个 DIV 元素
                var titleDiv = document.createElement("div");
                //设置显示每条记录的 DIV 元素的 HTML 部分
                titleDiv.innerHTML = '<a href="#" onClick="getBlog('
                    + data[i]['id'] + ');">' + data[i]['title'] + '</a>';
                listElement.appendChild(titleDiv);
            }
        }
    }
```

上面的回调函数也是翻页 Blog 文章列表的回调函数。

➤➤16.5.2 控制 Blog 文章列表的翻页

控制翻页比较简单，修改 JavaScript 的 curBlogPage 全局变量即可。修改 curBlogPage 变量减 1 就是前翻一页，修改 curBlogPage 变量加 1 就是后翻一页。下面是控制 Blog 文章列表向前翻页的代码：

程序清单：codes\16\blog\bm.js

```
//查看 Blog 文章列表的上一页函数
function blogPre()
{
    //如果当前页已经是第一页，无法翻页
    if (curBlogPage == 1)
    {
        alert("已经是第一页，无法向前翻页");
    }
    //执行翻页，修改 curBlogPage 减 1 后翻页
    else
    {
        bm.getAllBlogByPage(--curBlogPage , blCb);
    }
}
```

向前翻页先判断当前页码是否为 1，如果当前页码为 1，则表明无法翻页。否则修改 curBlogPage 变量减 1，再次执行请求完成翻页。下面是控制向后翻页的代码：

程序清单：codes\16\blog\bm.js

```
//查看 Blog 文章的下一页函数
function blogNext()
{
    bm.getAllBlogByPage(++curBlogPage , blCb);
}
```

因为本系统并未保存 Blog 文章列表的总页数，因此在翻页时无法判断当前页是否为最后一页，而是直接将当前页加 1，然后发送请求。

判断当前页是否为最后一页在回调函数中完成，如果在回调函数中获取的 Blog 文章列表为空，且不是第一页，则表明当前页已经是最后一页。如果在最后一页进行翻页，将看到如图 16.4 所示的警告框。

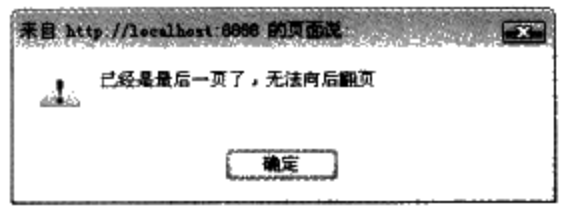


图 16.4 在 Blog 文章列表最后一页翻页的结果

➤➤16.5.3 页面加载时的动作

本系统控制当页面加载时，在页面左边加载当前所有 Blog 文章的标题，而右边显示当前最新的 Blog 文章。这个动作通过页面加载时的初始化函数控制。初始化函数需要完成两个动作：

- 加载当前最新的 Blog 文章。
- 加载当前页面的 Blog 文章列表。

其中第一个动作通过调用业务逻辑组件的 getNewestBlog()方法完成，而第二个方法通过上面定义的 getBlogList()函数完成。除此之外，初始化函数还设置了 DWR 的全局错误处理函数，初始化函数的代码如下：

程序清单：codes\16\blog\bm.js

```
//页面加载时自动执行的初始化函数
function init()
{
    dwr.engine.setErrorHandler(errHandler);
    getBlogList();
    bm.getNewestBlog(blogCb);
}
```

上面的代码中调用 getNewestBlog()方法时指定了 blogCb()回调函数，该函数将负责将返回的 Blog 文章在页面中显示出来。下面是 blogCb()回调函数的代码。

程序清单：codes\16\blog\bm.js

```
//获取 Blog 文章的回调函数
function blogCb(data)
{
    if (data != null && data != "undefined")
    {
        //curBlog 是个全局变量
        curBlog = data['id'];
        //将 Blog 文章中的内容在页面中显示出来
        dwr.util.setValue('addTime' , data['addTime'].toLocaleString());
        dwr.util.setValue('title' , data['title']);
        dwr.util.setValue('content' , data['content']);
    }
}
```

完成了上面的定义后，在浏览器中浏览系统的开发页面，将可看到如图 16.5 所示。

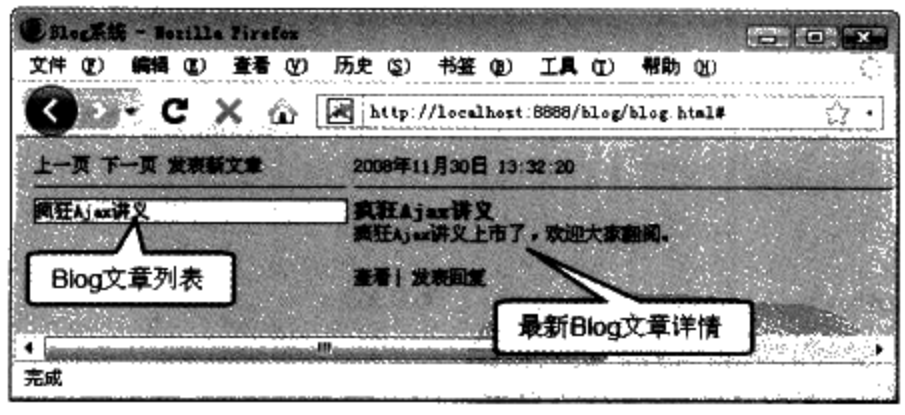


图 16.5　页面的初始化状态

正如在图 16.5 中所见到的，在页面左边看到的是当前页的 Blog 文章列表，右边看到的是最新 Blog 文章的详细内容。

➤➤16.5.4　查看评论

在图 16.5 中并未看到“疯狂 Ajax 讲义”这篇 Blog 文章的评论，本系统默认关闭 Blog 文章的评论显示，如果需要显示当前 Blog 文章的所有评论，则可单击“查看”超级链接，触发如下函数。

程序清单：codes\16\blog\bm.js

```
//查看回复的函数
```

```
function viewComment()
{
    bm.getCommentsByBlogAndPage(curBlog , curCommentPage , commentCb);
}
```

在上面的函数中，调用了业务逻辑组件的 getCommentsByBlogAndPage()方法，该方法根据指定页码指定 Blog lD 来获取该 Blog 当前页的所有回复。上面的代码中的 commentCb 是回调函数，该回调函数负责显示所有评论。与前面类似的是，该回调函数也是控制翻页的回调函数，因此有一些控制翻页的代码。下面是该回调函数的代码：

程序清单：codes\16\blog\bm.js

```
//查看回复的回调函数
function commentCb(data)
{
    //如果当前的记录为空，并且不是第一页
    if ((curCommentPage > 1 ) &&
        (data == null || data == "undefined" || data.length < 1))
    {
        alert("已经是最后一页了，无法向后翻页");
        //系统向前翻一页
        curCommentPage--;
        return false;
    }
    //如果没有回复
    else if (data == null || data == "undefined" || data.length < 1)
    {
        alert("暂时没有回复");
        return false;
    }
    //如果有回复
    else
    {
        //获取用于显示回复列表的 HTML 元素
        var comElement = $("comList");
        comElement.innerHTML="";
        //遍历当前页的所有回复
        for (var i = 0 ; i < data.length ; i++ )
        {
            //将回复的内容在 HTML 页面中显示出来
            var commentDiv = document.createElement("div");
            commentDiv.className="comment";
            commentDiv.innerHTML += data[i]['content'];
            commentDiv.innerHTML += "<br><br>发布者：<b><a href='mailto:"
                + data[i]['email'] + "'>" + data[i]['user'] + "</a> ("
                + data[i]['url'] + ") </b>—" + data[i]['addTime'].toLocaleString();
            comElement.appendChild(commentDiv);
            $("viewComment").style.display = "";
        }
    }
}
```

单击“查看”超级链接后，将可看到如图 16.6 所示的界面。

如图 16.6 所示，一旦浏览者单击了“查看”超级链接，将可看到当前 Blog 的所有回复列表，而且在查看回复时同样也可进行翻页控制。

图 16.6　查看 Blog 回复的界面

➤➤16.5.5　控制回复的翻页

系统提供了一个全局变量 curCommentPage，该变量用于保存当前所显示回复列表的页码。翻页就是通过修改该全局变量来控制的，将该变量值加 1 就是向后翻页，减 1 就是向前翻页。如果当前页已经是第一页，则取消翻页动作。翻页的回调函数也是使用上面定义的 commentCb 函数。控制向前翻页的代码如下：

程序清单：codes\16\blog\bm.js

```
//查看评论的上一页函数
function commentPre()
{
    //如果当前页是第一页
    if (curCommentPage == 1)
    {
        alert("已经是第一页，无法向前翻页");
    }
    else
    {
        //将 curCommentPage 变量减 1，然后请求获取指定页的所有回复列表
        bm.getCommentsByBlogAndPage(curBlog , --curCommentPage , commentCb );
    }
}
//查看评论的下一页函数
function commentNext()
{
    bm.getCommentsByBlogAndPage(curBlog , ++curCommentPage , commentCb );
}
```

与前面类似的是，系统并未保存 Blog 回复的总页数，所以无法知道当前页是否是最后一页，系统只是简单地将 curCommentPage 变量加 1，在回调函数中判断是否已经到达最后一页。如果单击如图 16.6 所示中的“下一页”超级链接，将可看到如图 16.7 所示的对话框。

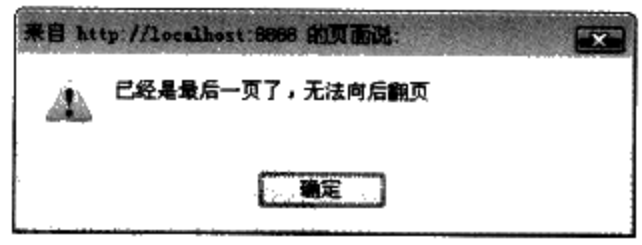


图 16.7　回复列表的最后一页进行翻页的提示

➤➤16.5.6　添加回复

系统默认不会显示添加回复的输入框，浏览者可通过单击“发表回复”的超级链接打开添加回复的输入框，单击“发表回复”链接将触发如下函数：

程序清单：codes\16\blog\bm.js

```
//显示添加回复的表格
function showAdd()
{
    //获取添加回复的 HTML 表格
    var addComment = $("addComment");
    //如果添加回复的表格被隐藏
```

```
    if (addComment.style.display == "none")
    {
        //显示添加回复的表格
        addComment.style.display = "";
    }
    else
    {
        //隐藏添加回复的表格
        addComment.style.display = "none";
    }
}
```

单击“发表回复”链接后可看到如图 16.8 所示的界面。

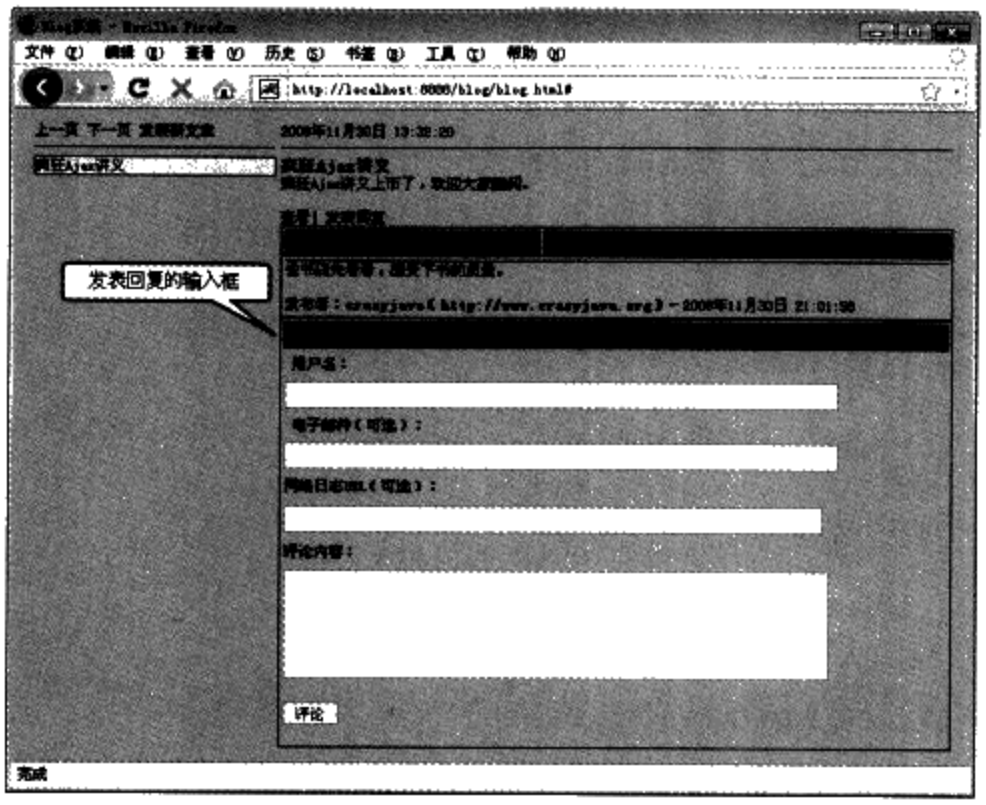


图 16.8　发表回复的输入框

正如在上面的代码中所看到的，单击“发表回复”链接可让发表回复输入框在显示和隐藏两种状态之间切换。如果在输入框中输入相应的内容，单击“评论”按钮，将触发如下函数：

程序清单：codes\16\blog\bm.js

```
//添加回复
function addComment()
{
    //curBlog 指定对哪篇文章进行回复
    if (curBlog < 0)
    {
        alert("请先选择需要回复的 Blog 文章");
        return false;
    }
    //获取用户名输入框的值
    var user = $("addUser").value;
    //获取电子邮件输入框的值
    var email = $("addEmail").value;
    //获取添加者 Blog 空间的地址
    var url = $("addUrl").value;
    var content = $("addContent").value;
    //要求必须输入用户名和 Blog 回复的内容
    if (user == null || user == "" || content == "" || content == null)
```

```
    {
        alert("请先输入用户名和您要评论的内容");
        return false;
    }
    //调用 createComment 业务逻辑方法来添加回复
    bm.createComment(user , email , url , content ,
        curBlog , addCommentCb);
}
```

上面的粗体字代码调用了 createComment()方法来添加评论，添加评论时指定了 addCommentCb()回调函数，该回调函数根据返回值判断添加回复是否成功。回调函数的代码如下：

程序清单：codes\16\blog\bm.js

```
//添加回复的回调函数
function addCommentCb(data)
{
    //返回值是添加回复的返回值（新创建回复的 ID），如果 ID 大于 0，表明添加成功
    if (data > 0)
    {
        //将各输入框清空
        $("addUser").value = "";
        $("addEmail").value = "";
        $("addUrl").value = "";
        $("addContent").value = "";
        //弹出提示框
        alert("添加评论成功");
    }
    else
    {
        //添加失败
        alert("添加评论失败");
    }
}
```

如果添加回复成功，将可看到如图 16.9 所示的提示框。

图 16.9　添加回复成功

➤➤16.5.7　查看 Blog 文章内容

在 16.5.1 节的代码中可以看到，在页面左边加载 Blog 文章列表时，每篇 Blog 文章标题都有一个超级链接，单击该文章标题将触发 getBlog(id)函数，该函数用于根据 Blog 文章 ID 获取文章的详细内容。getBlog()函数的代码如下：

程序清单：codes\16\blog\bm.js

```
//获取指定 ID 的 Blog 文章
function getBlog(id)
{
    //调用 getBlog 业务逻辑方法，获取 Blog 文章的详细内容
    bm.getBlog(id , blogCb);
    return false;
}
```

上述代码中的 blogCb 是回调函数，用于在页面中显示指定 Blog 文章的详细信息。该函数的代码在 16.5.3 节中已经给出，此处不再赘述。

➤➤16.5.8　添加新的 Blog 文章

添加新的 Blog 文章通过打开一个新的窗口完成。如果单击页面中的“发表新文章”链接，将触发

如下函数：

程序清单：codes\16\blog\bm.js

```
//打开添加 Blog 的窗口
function showAddBlog()
{
    //打开一个新的 JavaScript 窗口，在该窗口中加载 addBlog.html 页面
    win = window.open("addBlog.html","console","width=400,height=250");
    win.focus();
}
```

单击“发表新文章”链接后，可看到如图 16.10 所示的界面。

图 16.10　发表新 Blog 文章

如图 16.10 所示的界面中包含了“发表”按钮，单击该按钮将触发 addBlog()函数，该函数根据用户输入发表一篇新的 Blog 文章，代码如下：

程序清单：codes\16\blog\bm.js

```
//创建新 Blog 文章
function addBlog()
{
    //获取文章标题
    var blogTitle = $("blogTitle").value;
    //获取文章内容
    var blogContent = $("blogContent").value;
    //要求文章的标题和内容都不可为空
    if (blogTitle == null || blogTitle == "" ||
        blogContent == null || blogContent == "")
    {
        alert("新增 Blog 文章时，必须填写文章标题和文章内容");
    }
    else
    {
        //调用业务逻辑组件的方法添加新的 Blog 文章
        bm.createBlog(blogTitle , blogContent , addBlogCb);
    }
}
```

上面的粗体字代码调用了 createBlog()方法来新增 Blog 文章，调用该方法时指定了 addBlogCb()回调函数，该回调函数用于根据方法返回值判断添加 Blog 文章是否成功，下面是该函数的代码：

程序清单：codes\16\blog\bm.js

```
//创建新 Blog 文章的回调函数
function addBlogCb(data)
{
    //返回值是新添加的 Blog 文章的 ID，如果 ID 不是数字或者返回值不大于 0
```

```
        if (typeof data != "number" || data < 1)
        {
            alert("添加 Blog 文章失败");
        }
        //添加成功
        else
        {
            //将输入框清空
            $("blogTitle").value = "";
            $("blogContent").value = ""
            alert("添加文章成功");
            //关闭添加 Blog 文章的窗口
            window.close();
        }
    }
```

正常情况下，用户单击“发表”按钮，新的 Blog 文章添加完成，添加 Blog 文章的窗口消失，系统弹出如图 16.11 所示对话框。

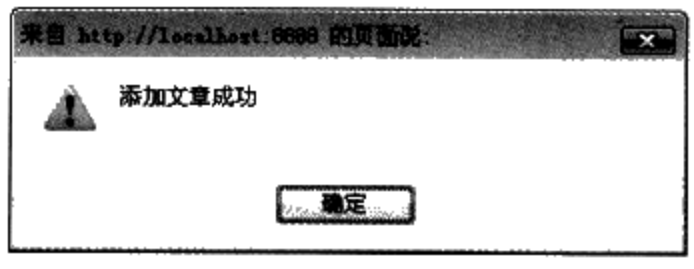


图 16.11　成功添加 Blog 文章

16.6　本章小结

本章通过一个 Blog 系统详细介绍了 Ajax 技术与实际 Java EE 应用的融合。本章使用了 DWR 作为 Ajax 引擎，让 JavaScript 直接访问 Spring 容器中的 Bean。应用的 DWR 框架取代了传统的 MVC 框架。通过 DWR 框架，可以让 JavaScript 调用 Spring 容器中 Bean 的方法。本章介绍了轻量级 Java EE 应用的开发过程，重点介绍了如何利用 DWR 来增加 Ajax 引擎，包括将 DWR 和 Spring 放在一起整合开发，并详细介绍了如何通过 DOM 操作来动态更新 HTML 页面。

接下来将介绍一个更复杂的 Ajax + Java EE 应用：电子拍卖系统。下一章所介绍的应用将更加完善，增加了 Ajax 引擎，并通过 Spring AOP 提供了完善的权限控制。

➤➤本章练习

继续完善该 Blog 系统，可以考虑从如下几个方面进行完善：

- 整合第 12 章的电子相册系统，允许 Blog 里增加相册系统。
- 为 Blog 系统添加用户管理（注册、登录等）。
- 为 Blog 建立分类管理，允许用户在添加 Blog 文章时选择分类。
- 为 Blog 系统建立圈子管理，允许用户加入指定圈子。
- 其他功能可参考 http://www.crazyjava.org 的 Blog 系统。

CHAPTER 17

第 17 章 电子拍卖系统

本章要点

- 传统 Java EE 应用的系统设计
- 分析、提取系统的 Domain Object
- 映射 Hibernate 的持久化对象
- 基于 HibernateDaoSupport 实现 DAO 组件
- 在 Spring 容器中部署 DAO 组件
- 实现业务逻辑组件
- 部署业务逻辑组件
- 使用声明式事务机制为业务逻辑方法增加事务控制
- 使用 AOP 机制为业务逻辑方法增加权限控制
- 利用 Spring 邮件抽象层发送竞价确认邮件
- 利用 Spring 任务调度处理拍卖到期的物品
- 使用 DWR 暴露 Spring 容器中的 Bean
- 使用 JavaScript 调用业务逻辑方法

本章所介绍的系统以一个轻量级 Java EE 应用为基础，在此基础上增加 Ajax 引擎，当用户刚进入系统时，需要下载整个应用的页面——可能需要比传统 Web 页面更多的时间。一旦下载完成，后面的用户交互将非常快捷。

该系统模拟了一个电子拍卖系统。注册用户可以发布拍卖物品，参与竞价。非注册用户可以浏览拍卖物品，浏览流拍物品。如果时间到了物品的拍卖期限，系统提供后台线程判断物品是流拍，还是被最高竞价者赢取。注册用户参与竞价后，系统会发送邮件通知竞价用户。Spring 的任务调度负责启动后台线程来修改物品状态；Spring 的邮件抽象层负责发送竞价通知邮件。

本系统使用 Hibernate 作为持久层的 O/R mapping 框架，使用 Spring 管理业务层组件和持久层组件。DWR 作为 Ajax 引擎的框架，直接调用 Spring 容器中的 Bean。整个应用的事务控制、权限控制都交给 Spring 的 AOP 机制完成。应用使用 DWR 作为系统的 Ajax 引擎，应用架构无须控制器，因此权限控制推迟到业务逻辑方法中完成。

17.1 总体说明和概要设计

Ajax 不能成为应用的全部，它只是应用的表现层部分。本章所介绍的不仅是一个基本的 Ajax 应用，而是轻量级 Java EE 应用与 Ajax 整合后的应用，从而提供更好的用户体验。

本应用的后台结构是一个完善的轻量级 Java EE 架构，应用架构中抛弃了控制器组件，改为使用 DWR 作为 Ajax 引擎，HTML 页面作为视图组件，浏览器里的 JavaScript 直接调用 Spring 容器中的业务逻辑组件，并负责在页面中显示业务逻辑组件的执行结果。

➤➤17.1.1 系统的总体架构设计

该系统采用 Java EE 的三层结构，分为表现层、业务逻辑层和数据服务层。多层结构将业务规则、数据访问等工作放到中间层处理，客户端不直接与数据库交互，而是通过控制器与中间层建立连接，再由中间层与数据库交互。系统的数据持久层使用 MySQL 数据库存放数据。

系统使用 HTML 页面作为表现层，浏览器里的 JavaScript 直接调用业务逻辑组件方法，并通过 DOM 操作将业务逻辑组件的方法执行结果动态显示在页面上。

系统中间层采用 Spring+Hibernate，为了更好地分离，中间层又可细分为如下几层：

- 业务逻辑层：负责实现业务逻辑，业务逻辑组件是 DAO 组件的门面。
- DAO 层：封装了数据的增、删、查、改等原子操作。
- Domain Object 层（领域对象层），通过实体/关系映射工具将领域对象映射成持久化对象，从而允许以面向对象方式操作数据库，本系统采用 Hibernate 作为 O/R mapping 框架。

Spring 框架贯穿整个中间层，Spring 可以管理持久化访问所需的数据源，也可以管理 Hibernate 的 SessionFactory，并可以管理业务逻辑组件和 DAO 组件之间的依赖关系。DWR 则是业务逻辑组件与浏览器端 JavaScript 之间的桥梁，通过 DWR 的中介，浏览器端 JavaScript 可以调用业务逻辑组件的方法。系统的总体架构如图 17.1 所示。

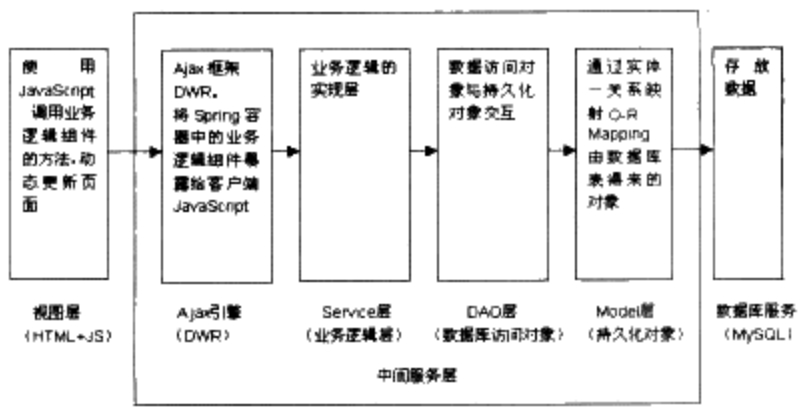


图 17.1 系统的总体设计图

17.1.2 数据库设计

本系统的 E-R 图如图 17.2 所示。

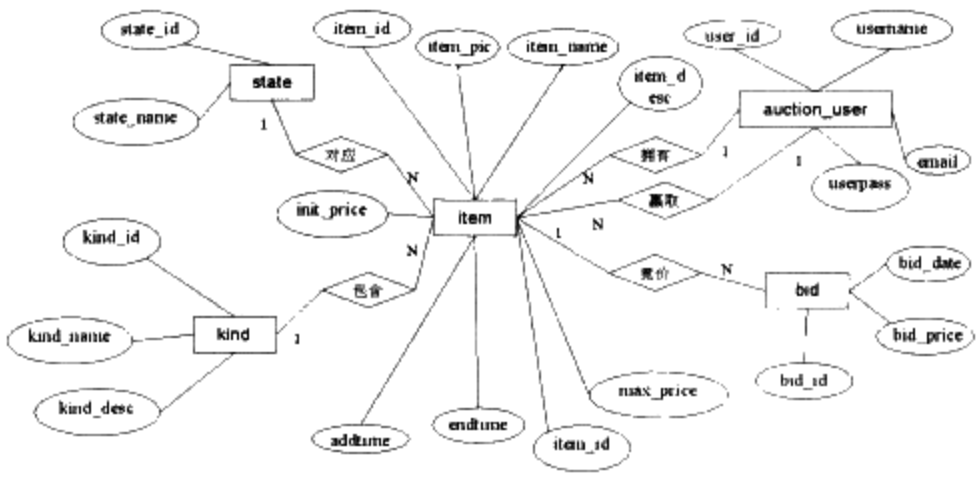


图 17.2 系统 E-R 图

本系统的数据库系统使用 MySQL 建立，包含 5 张数据表，分别用于存放上面 E-R 图中的 5 个实体。auction_user 表用于存放系统的注册用户信息，其表结构如图 17.3 所示。

Field	Type	Null	Key	Default	Extra
user_id	int(11)		PRI	NULL	auto_increment
username	varchar(50)		UNI		
userpass	varchar(50)				
email	varchar(100)				

图 17.3 auction_user 表

kind 表用于存放物品种类，其表结构如图 17.4 所示。

Field	Type	Null	Key	Default	Extra
kind_id	int(11)		PRI	NULL	auto_increment
kind_name	varchar(50)				
kind_desc	varchar(255)				

图 17.4 kind 表结构

item 表用于存放物品，其表结构如图 17.5 所示。

Field	Type	Null	Key	Default	Extra
item_id	int(11)	NO	PRI	NULL	auto_increment
item_name	varchar(255)	NO			
item_remark	varchar(255)	NO			
item_desc	varchar(255)	YES		NULL	
kind_id	int(11)	NO	MUL		
addtime	date	NO			
endtime	date	NO			
init_price	double	NO			
max_price	double	NO			
owner_id	int(11)	NO	MUL		
winer_id	int(11)	YES	MUL	NULL	
state_id	int(11)	NO	MUL		

图 17.5 item 表结构

state 表用于存放拍卖物品的状态，其表结构如图 17.6 所示。

Field	Type	Null	Key	Default	Extra
state_id	int(11)		PRI	NULL	auto_increment
state_name	varchar(10)	YES		NULL	

图 17.6　state 表结构

bid 表用于存放竞价记录，其表结构如图 17.7 所示。

Field	Type	Null	Key	Default	Extra
bid_id	int(11)		PRI	NULL	auto_increment
user_id	int(11)		MUL	0	
item_id	int(11)		MUL	0	
bid_price	double			0	
bid_date	date			0000-00-00	

图 17.7　bid 表结构

17.2　实现 Hibernate 持久化类

本系统打算使用贫血模型定义 Domain Object，系统 Domain Object 类就是持久化类，这些持久化类仅仅为各属性提供了必需的 setter 和 getter 方法，并未包含业务逻辑方法。所有的业务逻辑方法都由业务逻辑组件提供实现。

➤➤17.2.1　设计 Domain Object

本系统的开发并未完全按 OOA、OOD 的过程进行，而是采用了传统的信息化系统开发过程，先设计系统的数据库。因此在系统建模期间，已经得到了系统的 E/R 图，根据 E/R 图可以创建数据库的表，数据库表结构建立以后，可以根据表结构编写持久化对象。

虽然这个过程并不完全符合面向对象的设计过程，但因为数据库的建立对于企业信息应用非常重要，往往难以放弃分析系统的 E-R 关系图，因此 E-R 图的建立也是非常基础的部分。实际上 E-R 图也可用于辅助设计 Domain Object。

本系统一共有如下 5 个 Domain Object 对象，分别是：

- AuctionUser：对应注册用户，包括用户名、密码、Email 地址等信息。
- Kind：对应物品种类，包括种类名、种类描述等信息。
- State：对应物品的状态信息，包含状态名等信息。
- Item：对应物品，包含物品名、物品描述、物品备注、物品种类、物品状态等信息。
- Bid：对应竞价信息，包含竞价物品、参与竞价的用户、竞价价格等信息。

不仅如此，5 个 Domain Object 之间的关联关系比较多，它们之间具有如下关联关系：

- AuctionUser 与 Item 存在两种关系：所有者关系和赢取者关系。这两种关系都是 1－N 关系，即 AuctionUser 可以访问他所赢取的全部物品，也可以访问他所拥有的全部物品，因为 AuctionUser 通过 Set 类型的变量来分别保存他的赢取物品和所有物品。而 Item 里则保存 AuctionUser 的变量：分别是它对应的所有者和赢取者。
- Kind 和 Item 存在 1－N 关系，Kind 里以 Set 类型属性保存该种类下的全部物品，而 Item 里以 Kind 类型属性保存它所在的种类。
- State 和 Item 存在 1－N 关系，State 以 Set 类型属性保存该状态下的全部物品，而 Item 以 State 类型属性保存它所处的状态。
- Item 和 Bid 存在 1－N 关系，Item 以 Set 类型属性保存该物品的全部竞价，而 Bid 以 Item 类

型属性保存它对应的物品。

- User 和 Bid 也存在 1—N 关系，User 以 Set 类型属性保存该用户参与的全部竞价，而 Bid 以 User 类型属性保存参与竞价的用户。

图 17.8 显示了 5 个实体之间的关联关系。

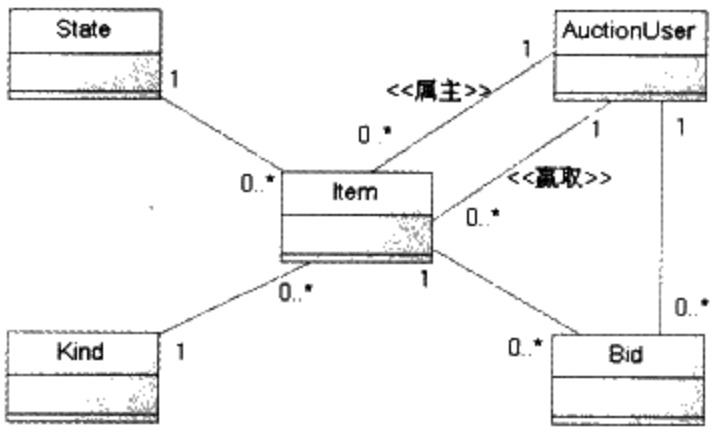


图 17.8　实体之间的关联关系

➤➤17.2.2　实现 Domain Object

各 Domain Object 之间存在的关联关系在图 17.8 上表现得非常清楚，其中 AuctionUser 和 Item 两个持久化类之间的关系尤为复杂，它们之间存在 2 种 1—N 关联，分别是所属和赢取两种关系。

下面是 AuctionUser 类的代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\model\AuctionUser.java

```
public class AuctionUser
{
    //标识属性
    private Integer id;
    //用户名属性
    private String username;
    //密码属性
    private String userpass;
    //电子邮件属性
    private String email;
    //根据属主关联的物品实体
    private Set<Item> itemsByOwner = new HashSet<Item>();
    //根据赢取者关联的物品实体
    private Set<Item> itemsByWiner = new HashSet<Item>();
    //该用户所参与的全部竞价
    private Set<Bid> bids = new HashSet<Bid>();
    //无参数的构造器
    public AuctionUser()
    {
    }
    //初始化全部基本属性的构造器
    public AuctionUser(Integer id , String username ,
        String userpass , String email)
    {
        this.id = id;
        this.username = username;
        this.userpass = userpass;
        this.email = email;
    }
    //下面省略普通属性的 setter 和 getter 方法
    ...
    //itemsByOwner 属性的 setter 和 getter 方法
```

```
    public void setItemsByOwner(Set<Item> itemsByOwner)
    {
        this.itemsByOwner = itemsByOwner;
    }
    public Set<Item> getItemsByOwner()
    {
        return this.itemsByOwner;
    }
    //itemsByWiner 属性的 setter 和 getter 方法
    public void setItemsByWiner(Set<Item> itemsByWiner)
    {
        this.itemsByWiner = itemsByWiner;
    }
    public Set<Item> getItemsByWiner()
    {
        return this.itemsByWiner;
    }
    //bids 属性的 setter 和 getter 方法
    public void setBids(Set<Bid> bids)
    {
        this.bids = bids;
    }
    public Set<Bid> getBids()
    {
        return this.bids;
    }
}
```

上面的 AuctionUser 里保留了 2 个 Set<Item>属性：itemsByOwner 和 itemsByWiner，其中 itemsByOwner 用于访问属于该用户的全部拍卖物品，而 itemsByWiner 用于访问该用户赢取的全部拍卖物品，这两个属性的关联实体都是 Item。此外，AuctionUser 里还有一个 bids 属性，该属性用于访问该用户参与的全部竞价记录。

AuctionUser 实体和 3 个关联实体存在 1－N 关系，因此需要在映射 AuctionUser 的映射文件中增加 3 个<set.../>元素，每个<set.../>元素映射一个关联实体，下面是 AuctionUser 类的映射文件：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\model\AuctionUser.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.auction.model">
    <class     name="AuctionUser" table="auction_user">
        <!-- 映射标识属性 -->
        <id    name="id" type="int" column="user_id">
            <!-- 指定主键生成策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="username" column="username" type="string"
            not-null="true" length="50" unique="true"/>
        <property name="userpass" column="userpass" type="string"
            not-null="true" length="50"/>
        <property name="email" column="email" type="string"
            not-null="true"       length="100"/>
        <!-- 映射该用户所拥有的全部 Item -->
        <set name="itemsByOwner" inverse="true">
```

```
            <key column="owner_id"/>
            <one-to-many class="Item"/>
        </set>
        <!-- 映射该用户所赢取的全部 Item -->
        <set name="itemsByWiner" inverse="true">
            <key column="winer_id"/>
            <one-to-many class="Item"/>
        </set>
        <!-- 映射该用户所参与的全部竞价 -->
        <set name="bids"  inverse="true">
            <key column="user_id" />
            <one-to-many class="Bid"/>
        </set>
    </class>
</hibernate-mapping>
```

本系统将把所有 1－N 关联都映射成双向关联，因此我们为 3 个<set.../>元素都指定 inverse="true"，这表明 AuctionUser 实体（1 的一端）不控制关联关系。根据 Hibernate 的建议：对于双向 1－N 关联，不要让 1 的一端控制关系，而应该让 N 的一端控制关联关系，这样可以保证更好的性能。

Item 类除了和 AuctionUser 之间存在 2 种 N－1 关联关系之外，还和 Bid 之间存在 1－N 关联关系，和 Kind 之间存在 N－1 关联关系，和 State 之间存在 N－1 关联关系。下面是 Item 类的代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\model\Item.java

```
public class Item
{
    //标识属性
    private Integer id;
    //物品 Remark
    private String itemRemark;
    //物品名称
    private String itemName;
    //物品描述
    private String itemDesc;
    //物品添加时间
    private Date addtime;
    //物品结束拍卖时间
    private Date endtime;
    //物品的起拍价
    private double initPrice;
    //物品的最高价
    private double maxPrice;
    //该物品的所有者
    private AuctionUser owner;
    //该物品所属的种类
    private Kind kind;
    //该物品的赢取者
    private AuctionUser winer;
    //该物品所处的状态
    private State itemState;
    //该物品对应的全部竞价记录
    private Set<Bid> bids = new HashSet<Bid>();
    //无参数的构造器
    public Item()
    {
    }
    //初始化全部基本属性的构造器
    public Item(Integer id , String itemRemark , String itemName ,
```

```
        String itemDesc , Date addtime , Date endtime ,
        double initPrice , double maxPrice , AuctionUser owner)
    {
        this.id = id;
        this.itemRemark = itemRemark;
        this.itemName = itemName;
        this.itemDesc = itemDesc;
        this.addtime = addtime;
        this.endtime = endtime;
        this.initPrice = initPrice;
        this.maxPrice = maxPrice;
        this.owner = owner;
    }
    //下面省略普通属性的 setter 和 getter 方法
    ...
    //owner 属性的 setter 和 getter 方法
    public void setOwner(AuctionUser owner)
    {
        this.owner = owner;
    }
    public AuctionUser getOwner()
    {
        return this.owner;
    }
    //kind 属性的 setter 和 getter 方法
    public void setKind(Kind kind)
    {
        this.kind = kind;
    }
    public Kind getKind()
    {
        return this.kind;
    }
    //winer 属性的 setter 和 getter 方法
    public void setWiner(AuctionUser winer)
    {
        this.winer = winer;
    }
    public AuctionUser getWiner()
    {
        return this.winer;
    }
    //itemState 属性的 setter 和 getter 方法
    public void setItemState(State itemState)
    {
        this.itemState = itemState;
    }
    public State getItemState()
    {
        return this.itemState;
    }
    //bids 属性的 setter 和 getter 方法
    public void setBids(Set<Bid> bids)
    {
        this.bids = bids;
    }
    public Set<Bid> getBids()
    {
```

```
        return this.bids;
    }
}
```

上面的 5 行粗体字代码定义了 Item 类对应的 4 个关联实体，其中 owner 引用该物品的所属者，winer 引用该物品的赢取者，这两个属性都引用到 AuctionUser 实体。Item 类和 AuctionUser、Kind、State 之间存在 N－1 关联，因此需要使用<many-to-one.../>元素来映射这些关联实体；Item 类和 Bid 之间存在 1－N 关联，因此需要使用<set.../>元素来映射 Bid。下面是 Item 类的映射文件：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\model\Item.hbm.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素 -->
<hibernate-mapping package="org.crazyjava.auction.model">
    <class name="Item" table="item">
        <!-- 映射标识属性 -->
        <id name="id" type="int" column="item_id">
            <!-- 指定主键生成策略 -->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 -->
        <property name="itemRemark" column="item_remark" type="string"
            not-null="true" length="255"/>
        <property name="itemName" column="item_name" type="string"
            not-null="true" length="255"/>
        <property name="itemDesc" column="item_desc" type="string"
            not-null="false" length="255"/>
        <property name="addtime" column="addtime" type="date"
            not-null="true" length="10"/>
        <property name="endtime" column="endtime" type="date"
            not-null="true" length="10"/>
        <property name="initPrice" column="init_price" type="double"
            not-null="true" length="12"/>
        <property name="maxPrice" column="max_price" type="double"
            not-null="true" length="12"/>
        <!-- 映射该 Item 所属的 AuctionUser -->
        <many-to-one name="owner" column="owner_id" class="AuctionUser"
            not-null="true" lazy="false"/>
        <!-- 映射该 Item 所属的 Kind -->
        <many-to-one name="kind" column="kind_id" class="Kind"
            not-null="true" lazy="false"/>
        <!-- 映射赢取该 Item 的 AuctionUser -->
        <many-to-one name="winer" column="winer_id" class="AuctionUser"
            not-null="false" lazy="false"/>
        <!-- 映射该 Item 所处的 State -->
        <many-to-one name="itemState" column="state_id" class="State"
            not-null="true" lazy="false"/>
        <!-- 映射该 Item 关联的全部竞价 -->
        <set name="bids" inverse="true">
            <key column="item_id"/>
            <one-to-many class="Bid"/>
        </set>
    </class>
</hibernate-mapping>
```

从上面的映射文件可以看出，我们为<many-to-one.../>元素指定了 lazy="false"，该属性将会取消延

迟加载，对于 N 对 1 的关联映射，N 的一端的实体只有一个关联实体，如果我们取消了延迟加载，那么系统加载 N 的一端的实体时，1 的一端的实体同时也会被加载出来——这不会有太大的问题，因为只是额外多加载了一条记录。

注意：

为<many-to-one.../>指定 lazy="false"不会有太大的性能下降。但对于<set.../>等映射集合的元素则尽量不要指定 lazy="false"。一旦为映射集合属性的元素指定了 lazy="false"，则意味着加载主表记录时，参照该记录的所有从表记录也会同时被加载出来，这会产生一个问题：我们无法预料有多少条从表记录参照该主表记录，同时加载这些从表记录可能引起巨大的性能下降。

本系统中还有 Bid、State 和 Kind 三个实体，但这三个实体都比 Item 和 AuctionUser 实体更简单。读者理解了 Item 和 AuctionUser 的映射关系，自然也就掌握了 Bid、State 和 Kind 的实现，故此处不再赘述。

17.3 DAO 层实现

本系统的后台完全采用轻量级 Java EE 应用的架构，系统持久层访问使用 DAO 组件完成。DAO 组件抽象出底层的数据访问，业务逻辑组件无须理会数据库访问的细节，只需专注于业务逻辑的实现即可。DAO 将数据访问集中在独立的一层，所有的数据访问都由 DAO 对象完成，将数据访问集中使得系统更具可维护性。

DAO 组件还有助于提升系统的可移植性，独立的 DAO 层使得系统能在不同的数据库之间轻易切换，底层的数据库实现对于业务逻辑组件完全透明，数据库移植时仅仅影响 DAO 层，不同数据库的切换不会影响业务逻辑组件，因此提高了系统的可移植性。

Spring 为 DAO 模式提供了更好的支持，通过 Spring 提供的 DAO 支持类 HibernateDaoSupport，应用可以更简单地实现 DAO 模式。

➤➤17.3.1 DAO 的基础配置

HibernateDaoSupport 需要容器注入一个 SessionFactory 引用，该类也提供了 setSessionFactory 方法，用于依赖注入 SessionFactory 属性。HibernateDaoSupport 一旦获得了 SessionFactory 的引用，就可以调用 getHibernateTemplate()方法获得 HibernateTemplate 的实例，HibernateTemplate 提供了大量便捷方法来简化数据库访问。

Spring 为整合 Hibernate 提供了 LocalSessionFactoryBean 类，可以将 Hibernate 的 SessionFactory 纳入其 IoC 容器内。使用 LocalSessionFactoryBean 配置 SessionFactory 之前，必须为其提供对应的数据源，SessionFactory 的相关配置片段如下：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\model\Item.hbm.xml

```
<!-- 定义数据源 Bean，使用 C3P0 数据源实现 -->
<bean id="dataSource" destroy-method="close"
    class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- 指定连接数据库的驱动 -->
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <!-- 指定连接数据库的 URL -->
    <property name="jdbcUrl"
        value="jdbc:mysql://localhost:3306/auction"/>
    <!-- 指定连接数据库的用户名 -->
    <property name="user" value="root"/>
    <!-- 指定连接数据库的密码 -->
    <property name="password" value="32147"/>
    <!-- 指定连接数据库连接池的最大连接数 -->
```

```
        <property name="maxPoolSize" value="40"/>
        <!-- 指定连接数据库连接池的最小连接数 -->
        <property name="minPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的初始化连接数 -->
        <property name="initialPoolSize" value="1"/>
        <!-- 指定连接数据库连接池的连接的最大空闲时间 -->
        <property name="maxIdleTime" value="20"/>
    </bean>
    <!-- 定义 Hibernate 的 SessionFactory -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- 依赖注入数据源，注入上面定义的 dataSource -->
        <property name="dataSource" ref="dataSource"/>
        <!-- mappingResouces 属性用来列出全部映射文件 -->
        <property name="mappingResources">
            <list>
                <!-- 以下用来列出 Hibernate 映射文件 -->
                <value>org/crazyjava/auction/model/AuctionUser.hbm.xml</value>
                <value>org/crazyjava/auction/model/Bid.hbm.xml</value>
                <value>org/crazyjava/auction/model/Item.hbm.xml</value>
                <value>org/crazyjava/auction/model/Kind.hbm.xml</value>
                <value>org/crazyjava/auction/model/State.hbm.xml</value>
            </list>
        </property>
        <!-- 定义 Hibernate 的 SessionFactory 的属性 -->
        <property name="hibernateProperties">
            <props>
                <!-- 指定数据库方言 -->
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLInnoDBDialect</prop>
                <!-- 是否根据需要每次自动创建数据库 -->
                    <prop key="hibernate.hbm2ddl.auto">update</prop>
                <!-- 显示 Hibernate 持久化操作所生成的 SQL -->
                <prop key="hibernate.show_sql">true</prop>
                <!-- 将 SQL 脚本进行格式化后再输出 -->
                <prop key="hibernate.format_sql">true</prop>
            </props>
        </property>
    </bean>
```

注意：

Hibernate 属性可以直接放在 LocalSessionFactoryBean bean 内配置，也可以放在 hibernate.cfg.xml 文件中配置。

17.3.2 实现 DAO 组件

为了实现 DAO 模式，系统至少需要如下三个部分：

- DAO 接口。
- DAO 接口的实现类。
- DAO 工厂。

对于采用 Spring 框架的应用而言，无须额外提供 DAO 工厂，因为 Spring 容器本身就是 DAO 工厂。此外，开发者需要提供 DAO 接口和 DAO 实现类。每个 DAO 组件都应该提供标准的新增、加载、加载和删除等方法，此外还需提供数量不等的查询方法。

如下是 AuctionUserDao 接口的源代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\dao\AuctionUserDao.java

```
public interface AuctionUserDao
{
    /**
     * 根据 ID 查找用户
     * @param id 需要查找的用户 ID
     */
    AuctionUser get(Integer id);
    /**
     * 增加用户
     * @param user 需要增加的用户
     */
    void save(AuctionUser user);
    /**
     * 修改用户
     * @param user 需要修改的用户
     */
    void update(AuctionUser user);
    /**
     * 删除用户
     * @param id 需要删除的用户 ID
     */
    void delete(Integer id);
    /**
     * 删除用户
     * @param user 需要删除的用户
     */
    void delete(AuctionUser user);
    /**
     * 查询全部用户
     * @return 获得全部用户
     */
    List<AuctionUser> findAll();
    /**
     * 根据用户名、密码查找用户
     * @param username 查询所需的用户名
     * @param pass 查询所需的密码
     * @return 指定用户名、密码对应的用户
     */
    AuctionUser findUserByNameAndPass(String username , String pass);
}
```

上面的 AuctionUser 接口里定义了增加、修改、根据主键加载、删除、根据主键删除等 5 个标准 DAO 方法，还定义了一个 findAll()方法，该方法用于返回全部的 AuctionUser 实例；此外还定义了一个 findUserByNameAndPass()方法，该方法根据用户名、密码查询 AuctionUser，由于本系统在映射 AuctionUser 的 username 属性时指定了 unique="true"，因此根据 username、pass 查询时不会返回 List，最多只会返回一个 AuctionUser 实例。

定义了上面的 AuctionUserDao 接口之后，下面就可以为该接口提供实现类了，代码如下：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\dao\impl\AuctionUserDaoHibernate.java

```
public class AuctionUserDaoHibernate
    extends HibernateDaoSupport implements AuctionUserDao
{
    /**
     * 根据 ID 查找用户
     * @param id 需要查找的用户 ID
```

```
     */
    public AuctionUser get(Integer id)
    {
        return (AuctionUser)getHibernateTemplate()
            .get(AuctionUser.class , id);
    }
    /**
     * 增加用户
     * @param user 需要增加的用户
     */
    public void save(AuctionUser user)
    {
        getHibernateTemplate().save(user);
    }
    /**
     * 修改用户
     * @param user 需要修改的用户
     */
    public void update(AuctionUser user)
    {
        getHibernateTemplate().saveOrUpdate(user);
    }
    /**
     * 删除用户
     * @param id 需要删除的用户 ID
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 删除用户
     * @param user 需要删除的用户
     */
    public void delete(AuctionUser user)
    {
        getHibernateTemplate().delete(user);
    }
    /**
     * 查询全部用户
     * @return 获得全部用户
     */
    public List<AuctionUser> findAll()
    {
        return (List<AuctionUser>)getHibernateTemplate()
            .find("from AuctionUser");
    }
    /**
     * 根据用户名、密码查找用户
     * @param username 查询所需的用户名
     * @param pass 查询所需的密码
     * @return 指定用户名、密码对应的用户
     */
    public AuctionUser findUserByNameAndPass(String username , String pass)
    {
        //执行 HQL 查询
        List<AuctionUser> ul = (List<AuctionUser>)getHibernateTemplate()
            .find("from AuctionUser au where au.username = ? and au.userpass = ?" ,
```

```
                new String[]{username , pass});
            //返回查询得到的第一个 AuctionUser 对象
            if (ul.size() == 1)
            {
                return (AuctionUser)ul.get(0);
            }
            return null;
        }
    }
```

上面的 AuctionUserDaoHibernate 类中除了粗体字方法稍稍复杂一点以外，其他大部分 DAO 方法体只需一行代码即可，这就是 HibernateTemplate 的功劳：HibernateTemplate 封装了大部分通用操作，开发者只需提供关键部分的代码即可。

比 AuctionUserDao 稍微复杂一点的是 ItemDao，下面是 ItemDao 接口的代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\dao\ItemDao.java

```
public interface ItemDao
{
    /**
     * 根据主键查找物品
     * @param itemId 待查找物品的 ID
     * @return 对应的物品
     */
    Item get(Integer itemId);
    /**
     * 保存物品
     * @param item 需要保存的物品
     */
    void save(Item item);
    /**
     * 修改物品
     * @param item 需要修改的物品
     */
    void update(Item item);
    /**
     * 删除物品
     * @param id 需要删除的物品 ID
     */
    void delete(Integer id);
    /**
     * 删除物品
     * @param item 需要删除的物品
     */
    void delete(Item item);
    /**
     * 根据产品分类，获取当前拍卖的全部商品
     * @param kindId 种类 ID
     * @return 该类的全部产品
     */
    List<Item> findItemByKind(Integer kindId);
    /**
     * 根据所有者查找处于拍卖中的物品
     * @param useId 所有者 ID
     * @return 指定用户处于拍卖中的全部物品
     */
    List<Item> findItemByOwner(Integer userId);
    /**
     * 根据赢取者查找物品
```

```
     * @param userId 赢取者 ID
     * @return 指定用户赢取的全部物品
     */
    List<Item> findItemByWiner(Integer userId);
    /**
     * 根据物品状态查找物品
     * @param stateId 状态 ID
     * @return 该状态下的全部物品
     */
    List<Item> findItemByState(Integer stateId);
}
```

类似地，ItemDaoHibernate 一样继承 HibernateDaoSupport 就能用简单的代码来实现该 DAO 组件的全部方法，下面是 ItemDaoHibernate 类的代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\dao\impl\ItemDaoHibernate.java

```
public class ItemDaoHibernate
    extends HibernateDaoSupport implements ItemDao
{
    /**
     * 根据主键查找物品
     * @param itemId 待查找物品的 ID
     * @return 对应的物品
     */
    public Item get(Integer itemId)
    {
        return (Item)getHibernateTemplate().get(Item.class , itemId);
    }
    /**
     * 保存物品
     * @param item 需要保存的物品
     */
    public void save(Item item)
    {
        getHibernateTemplate().save(item);
    }
    /**
     * 修改物品
     * @param item 需要修改的物品
     */
    public void update(Item item)
    {
        getHibernateTemplate().saveOrUpdate(item);
    }
    /**
     * 删除物品
     * @param id 需要删除的物品 ID
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(get(id));
    }
    /**
     * 删除物品
     * @param item 需要删除的物品
     */
    public void delete(Item item)
    {
```

```
        getHibernateTemplate().delete(item);
    }
    /**
     * 根据产品分类，获取当前拍卖的全部商品
     * @param kindId 种类 ID
     * @return 该类的全部产品
     */
    public List<Item> findItemByKind(Integer kindId)
    {
        return (List<Item>)getHibernateTemplate()
            .find("from Item as i where i.kind.id = ? and i.itemState.id = 1"
            , kindId);
    }
    /**
     * 根据所有者查找处于拍卖中的物品
     * @param useId 所有者 ID
     * @return 指定用户处于拍卖中的全部物品
     */
    public List<Item> findItemByOwner(Integer userId)
    {
        return (List<Item>)getHibernateTemplate()
            .find("from Item as i where i.owner.id = ? and i.itemState.id = 1"
            , userId);
    }
    /**
     * 根据赢取者查找物品
     * @param userId 赢取者 ID
     * @return 指定用户赢取的全部物品
     */
    public List<Item> findItemByWiner(Integer userId)
    {
        return (List<Item>)getHibernateTemplate()
            .find("from Item as i where i.winer.id = ? and i.itemState.id = 2"
            ,userId);
        }
    /**
     * 根据物品状态查找物品
     * @param stateId 状态 ID
     * @return 该状态下的全部物品
     */
    public List<Item> findItemByState(Integer stateId)
    {
        return (List<Item>)getHibernateTemplate()
            .find("from Item as i where i.itemState.id = ?" , stateId);
    }
}
```

与 AuctionUserDaoHiberante 类相似，ItemDaoHibernate 类也非常简单，几乎所有方法都只要一行代码即可实现。

借助于 Spring+Hibernate 的简化，开发者可以非常简便地实现所有 DAO 组件，系统中的 KindDao、BidDao、StateDao 并不会比 ItemDao、AuctionUserDao 复杂，故不再给出它们的实现。

➤➤17.3.3 部署 DAO 组件

所有继承 HibernateDaoSupport 的 DAO 实现类必须为其提供 SessionFactory 的引用，Spring 的 IoC 容器可以将 SessionFactory 注入到 DAO 组件中。

下面是本系统中部署 DAO 组件的配置代码，本系统以单独配置文件来部署 DAO 组件，这样可以

将不同组件放在不同配置文件中分开管理，从而避免 Spring 配置文件过于庞大。下面是部署 DAO 组件的配置文件：

程序清单：codes\17\auction\WEB-INF\daoContext.xml

```
<?xml version="1.0" encoding="GBK"?>
<!-- 指定 Spring 配置文件的 DTD 信息 -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <!-- 配置 daoTemplate，作为所有 DAO 组件的模板 -->
    <bean id="daoTemplate" abstract="true">
        <!-- 为 DAO 组件注入 SessionFactory 引用 -->
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
    <!-- 配置 stateDao 组件 -->
    <bean id="stateDao" parent="daoTemplate"
        class="org.crazyjava.auction.dao.impl.StateDaoHibernate"/>
    <!-- 配置 kindDao 组件 -->
    <bean id="kindDao" parent="daoTemplate"
        class="org.crazyjava.auction.dao.impl.KindDaoHibernate"/>
    <!-- 配置 auctionDao 组件 -->
    <bean id="auctionUserDao" parent="daoTemplate"
        class="org.crazyjava.auction.dao.impl.AuctionUserDaoHibernate"/>
    <!-- 配置 bidDao 组件 -->
    <bean id="bidDao" parent="daoTemplate"
        class="org.crazyjava.auction.dao.impl.BidDaoHibernate"/>
    <!-- 配置 itemDao 组件 -->
    <bean id="itemDao" parent="daoTemplate"
        class="org.crazyjava.auction.dao.impl.ItemDaoHibernate"/>
</beans>
```

上面的粗体字代码配置了一个 daoTemplate 抽象 Bean，它将作为系统中其他 DAO 组件的模板，daoTemplate 将作为其他 DAO 组件的配置模板，这样就可将 daoTemplate 的配置属性传递给其他 DAO Bean。

为了让其他 DAO 组件获得 daoTemplate 的配置属性，必须将其他 DAO 组件配置成 daoTemplate 的子 Bean，子 Bean 通过 parent 属性指定父 Bean，正如上面的配置文件中每个 DAO Bean 的粗体字代码都指定了 parent="daoTemplate"，即表明这些 DAO 组件将以 daoTemplate 作为模板。

注意：

上面各 DAO 实现类并未提供 setSessionFactory()方法，该方法由其父类 HibernateDaoSupport 提供，用于依赖注入 SessionFactory()引用。该配置文件中并未配置 SessionFactory bean，DataSource 和 SessionFactory 的配置在 applicationContext.xml 文件中。

17.4 业务逻辑层实现

本系统的规模不大，只涉及到 5 个 DAO 组件，分别对应 5 个持久化对象的访问。本系统使用一个业务逻辑对象即可封装 5 个 DAO 组件。对于只采用一个业务逻辑对象的情形，则 DWR 只需暴露一个 JavaScript 对象，而浏览器 JavaScript 即可通过该对象来调用其业务逻辑方法，从而对用户请求提供响应。如果系统中包含多个业务逻辑组件，则 DWR 需要暴露多个 JavaScript 对象。

➤➤17.4.1 设计业务逻辑组件

业务逻辑组件采用门面模式封装多个 DAO 组件，DAO 组件与业务逻辑组件之间的关系如图 17.9 所示。

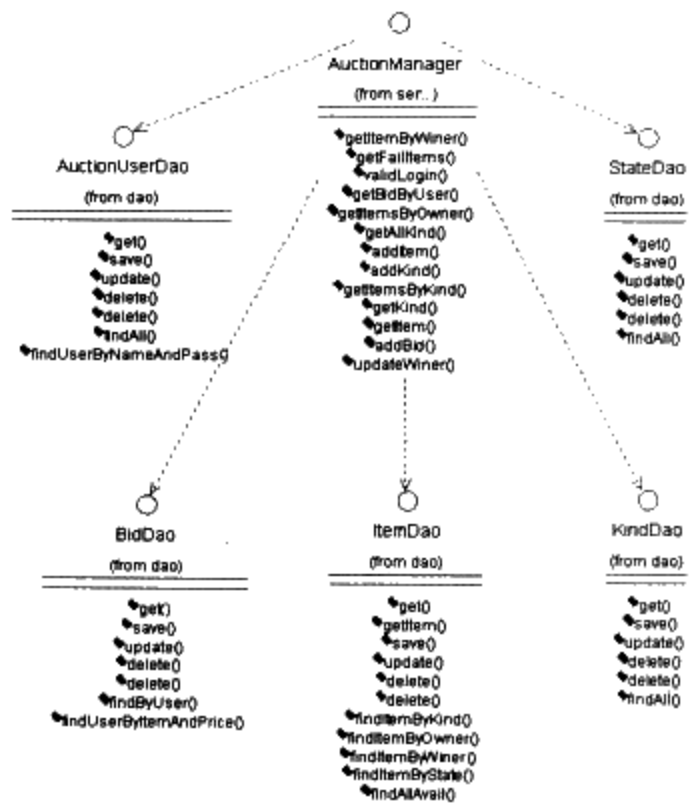


图 17.9　AuctionManager 与 DAO 组件接口的类图

AuctionManager 接口里定义了大量业务方法，这些业务方法的实现依赖于 DAO 组件。为了提供高层次的解耦，业务逻辑组件推荐使用接口分离的规则，将业务逻辑组件分成接口和相应的实现类两个部分。

AuctionManagerImpl 实现类实现了 AuctionManager 接口，实现了该接口中的所有方法。此外，AuctionManagerImpl 实现类比接口中的定义多了如下 5 个依赖注入的方法：

- setUserDao(AuctionUserDao dao)：为业务逻辑组件依赖注入 AuctionUserDao 的方法。
- setBidDao(BidDao dao)：为业务逻辑组件依赖注入 BidDao 的方法。
- setItemDao(ItemDao dao)：为业务逻辑组件依赖注入 ItemDao 的方法。
- setKindDao(KindDao dao)：为业务逻辑组件依赖注入 KindDao 的方法。
- setStateDao(StateDao dao)：为业务逻辑组件依赖注入 StateDao 的方法。

➤➤17.4.2　业务逻辑组件的异常处理

DWR 的功能非常强大，它支持将服务器端的异常信息传给浏览器，甚至可以将整个异常对象转换成 JavaScript 对象后传递给浏览器，从而能提供更丰富的异常信息。大部分时候，没有必要将整个异常对象传递到客户端，只需要将异常信息传递到客户端即可。

Spring 的异常处理哲学简化了异常的处理：所有的数据库访问异常都被包装了 Runtime 异常，DAO 组件中无须显式捕捉异常，所有的异常都被推迟到业务逻辑组件中捕捉。

业务逻辑组件中捕捉系统抛出的原始异常，这种原始异常不应该被客户端看到，甚至不应该在服务器端暴露出来。为了达到这个目的，可以使用 log4j 的日志功能：系统使用 log4j 记录业务逻辑方法中原始的异常信息，然后再抛出自定义异常。下面是本系统中自定义异常类的代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\exception\AuctionException.java

```
public class AuctionException extends Exception
{
    //定义一个无参数的构造器
    public AuctionException()
    {
    }
    //定义一个带 message 参数的构造参数
```

```
    public AuctionException(String message)
    {
        super(message);
    }
}
```

系统业务逻辑方法采用如下方式来处理逻辑：

```
try
{
    //完成业务逻辑
    ...
}
//捕捉异常
catch (Exception e)
{
    //通过日志记录异常
    log.debug(e.getMessage());
    //抛出新异常
    throw new AuctionException("底层业务异常,请重试");
}
```

当 JavaScript 调用业务逻辑方法抛出异常时，DWR 可以捕捉到这个异常信息，并将该信息传给浏览器 JavaScript 代码，可以通过设置 DWR 的全局错误处理器来处理该异常。

➤➤17.4.3 发送竞价通知邮件

本系统提供了邮件发送的功能，当用户竞价成功后，系统将发送邮件通知竞价用户。通知邮件的发送采用 Spring 的邮件抽象层，Spring 的邮件抽象中包含两个工具类：

- JavaMailSender。
- SimpleMailMessage。

本系统并未发送 MimeMessage 邮件，而是使用 SimpleMailMessage 来发送文本邮件。

提 示：

如果系统需要发送邮件的内容非常丰富，既包含文本内容，也包含图片内容，则可以考虑发送 MimeMessage 邮件。

为了让业务逻辑组件增加发送邮件的功能，必须为其增加如下两个方法：

```
//依赖注入 MailSender 实例的 setter 方法
public void setMailSender(MailSender mailSender)
{
    this.mailSender = mailSender;
}
//依赖注入 SimpleMailMessage 对象的 setter 方法
public void setMessage(SimpleMailMessage message)
{
    this.message = message;
}
```

上面两个方法是为业务逻辑组件注入 MailSender 和 Message 的 setter 方法，因此还必须在 Spring 配置文件中配置如下两个发送邮件的 Bean：

- mailSender。
- mailMessage。

在 Spring 配置文件中配置这两个 Bean 之后，还应该使用依赖注入将这两个 Bean 注入到业务逻辑组件中。下面是定义发送邮件的两个 Bean：

程序清单：codes\17\auction\WEB-INF\applicationContext.xml

```
<!-- 定义 JavaMailSenderImpl，它用于发送邮件 -->
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <!-- 指定发送邮件的 SMTP 服务器地址 -->
    <property name="host" value="smtp.163.com"/>
    <property name="javaMailProperties">
        <props>
            <prop key="mail.smtp.auth">true</prop>
            <prop key="mail.smtp.timeout">25000</prop>
        </props>
    </property>
    <!-- 指定登录邮箱的用户名、密码 -->
    <property name="username" value="spring_test"/>
    <property name="password" value="123abc"/>
</bean>
<!-- 定义 SimpleMailMessage Bean，它代表了一份邮件 -->
<bean id="mailMessage"
    class="org.springframework.mail.SimpleMailMessage">
    <property name="from" value="spring_test@163.com"/>
    <!-- 指定邮件标题 -->
    <property name="subject" value="竞价通知"/>
</bean>
```

➤➤17.4.4 实现业务逻辑层组件

业务逻辑层组件与具体的数据库访问技术分离，完全依赖于 DAO 组件。为了让业务逻辑组件与 DAO 组件实现分离，通常推荐面向接口编程。

此处的业务逻辑组件有一个值得注意的地方，那就是权限控制。因为取消了传统 Java EE 应用中的控制器，因此权限检查推迟到业务逻辑组件中进行。权限检查一般是通过跟踪 HttpSession 来完成，那些需要进行权限控制的业务逻辑方法都需要访问 HttpSession。

DWR 为访问这种 HttpSession 提供了简单的控制，当客户端 JavaScript 调用包含 HttpSession 参数的方法时，JavaScript 无须理会 HttpSession 参数，DWR 负责为该参数传入参数值。

本系统的权限使用了 Spring AOP，因此并未直接在业务逻辑方法中进行权限检查，权限检查通过拦截器完成。下面是 AuctionManagerImpl 的源代码：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\service\impl\AuctionManagerImpl.java

```
public class AuctionManagerImpl implements AuctionManager
{
    static Logger log = Logger.getLogger(
        AuctionManagerImpl.class.getName());
    //以下是该业务逻辑组件所依赖的 DAO 组件
    private AuctionUserDao userDao;
    private BidDao bidDao;
    private ItemDao itemDao;
    private KindDao kindDao;
    private StateDao stateDao;
    //业务逻辑组件发送邮件所依赖的两个 Bean
    private MailSender mailSender;
    private SimpleMailMessage message;
    //为业务逻辑组件依赖注入 DAO 组件所需的 setter 方法
    public void setUserDao(AuctionUserDao userDao)
    {
        this.userDao = userDao;
    }
```

```
public void setBidDao(BidDao bidDao)
{
    this.bidDao = bidDao;
}
public void setItemDao(ItemDao itemDao)
{
    this.itemDao = itemDao;
}
public void setKindDao(KindDao kindDao)
{
    this.kindDao = kindDao;
}
public void setStateDao(StateDao stateDao)
{
    this.stateDao = stateDao;
}
//为业务逻辑组件注入两个邮件发送 Bean 的 setter 方法
public void setMailSender(MailSender mailSender)
{
    this.mailSender = mailSender;
}
public void setMessage(SimpleMailMessage message)
{
    this.message = message;
}
/**
 * 根据赢取者查询物品
 * @param sess 用于获取赢取者的 HttpSession
 * @return 赢取者获得的全部物品
 */
public List getItemByWiner(HttpSession sess) throws AuctionException
{
    try
    {
        //从 HttpSession 中取出 userId 属性
        Integer winerId = (Integer)sess.getAttribute("userId");
        List items = itemDao.findItemByWiner(winerId);
        List result = new ArrayList();
        for (Iterator it = items.iterator() ; it.hasNext() ; )
        {
            Item item = (Item)it.next();
            ItemBean ib = new ItemBean();
            initItem(ib,item);
            result.add(ib);
        }
        return result;
    }
    catch (Exception e)
    {
        log.debug(e.getMessage());
        throw new AuctionException("查询用户所赢取的物品出现异常,请重试");
    }
}
/**
 * 查询流拍的全部物品
 * @return 全部流拍物品
 */
public List getFailItems() throws AuctionException
```

```
    {
        try
        {
            List items = itemDao.findItemByState(3);
            List result = new ArrayList();
            for (Iterator it = items.iterator() ; it.hasNext() ; )
            {
                Item item = (Item)it.next();
                ItemBean ib = new ItemBean();
                initItem(ib,item);
                result.add(ib);
            }
            return result;
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("查询流拍物品出现异常,请重试");
        }
    }
    /**
     * 根据用户名、密码验证登录是否成功
     * @param username 登录的用户名
     * @param pass 登录的密码
     * @param verCode 验证码
     * @param sess 跟踪用户登录的 HttpSession 对象
     * @return 登录是否成功
     */
    public boolean validLogin(String username , String pass ,
        String verCode , HttpSession sess) throws AuctionException
    {
        String rand = (String)sess.getAttribute("rand");
        if (!rand.equals(verCode))
        {
            throw new AuctionException("您输入的验证码不对,请重试");
        }
        try
        {
            AuctionUser u = userDao.findUserByNameAndPass(username , pass);
            if (u != null)
            {
                sess.setAttribute("userId" , u.getId());
                return true;
            }
            return false;
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("处理用户登录出现异常,请重试");
        }
    }
    /**
     * 查询用户的全部出价
     * @param sess 用于获取查询用户的 HttpSession
     * @return 用户的全部出价
     */
    public List getBidByUser(HttpSession sess) throws AuctionException
```

```
    {
        try
        {
            Integer userId = (Integer)sess.getAttribute("userId");
            List l = bidDao.findByUser(userId);
            List result = new ArrayList();
            for ( int i = 0 ; i < l.size() ; i++ )
            {
                Bid bid = (Bid)l.get(i);
                BidBean bb = new BidBean();
                initBid(bb, bid);
                result.add(bb);
            }
            return result;
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("浏览用户的全部竞价出现异常,请重试");
        }
    }
    /**
     * 根据用户查找目前仍在拍卖中的全部物品
     * @param sess 用于获取所属者的 HttpSession
     * @return 属于当前用户的、处于拍卖中的全部物品
     */
    public List getItemsByOwner(HttpSession sess) throws AuctionException
    {
        try
        {
            Integer userId = (Integer)sess.getAttribute("userId");
            List result = new ArrayList();
            List items = itemDao.findItemByOwner(userId);
            for (Iterator it = items.iterator() ; it.hasNext() ; )
            {
                Item item = (Item)it.next();
                ItemBean ib = new ItemBean();
                initItem(ib,item);
                result.add(ib);
            }
            return result;
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("查询用户所有的物品出现异常,请重试");
        }
    }
    /**
     * 查询全部种类
     * @return 系统中的全部种类
     */
    public List getAllKind() throws AuctionException
    {
        List result = new ArrayList();
        try
        {
            List kl = kindDao.findAll();
```

```
            for (Object o : kl )
            {
                Kind k = (Kind)o;
                result.add(new KindBean(k.getId(),
                    k.getKindName(), k.getKindDesc()));
            }
            return result;
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("查询全部种类出现异常,请重试");
        }
    }
    /**
     * 添加物品
     * @param name 物品名称
     * @param desc 物品描述
     * @param remark 物品备注
     * @param avail 有效天数
     * @param kind 物品种类
     * @param sess 跟踪添加者的 HttpSession
     * @return 新增物品的主键
     */
    public int addItem(String name , String desc , String remark ,
        double initPrice , int avail , int kind , HttpSession sess)
        throws AuctionException
    {
        try
        {
            Integer userId = (Integer)sess.getAttribute("userId");
            Kind k = kindDao.get(kind);
            AuctionUser owner = userDao.get(userId);
            //创建 Item 对象
            Item item = new Item();
            item.setItemName(name);
            item.setItemDesc(desc);
            item.setItemRemark(remark);
            item.setAddtime(new Date());
            Calendar c = Calendar.getInstance();
            c.add(Calendar.DATE , avail);
            item.setEndtime(c.getTime());
            item.setInitPrice(new Double(initPrice));
            item.setMaxPrice(new Double(initPrice));
            item.setItemState(stateDao.get(1));
            item.setKind(k);
            item.setOwner(owner);
            //持久化 Item 对象
            itemDao.save(item);
            return item.getId();
            }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("添加物品出现异常,请重试");
        }
    }
    /**
```

```
     * 添加种类
     * @param name 种类名称
     * @param desc 种类描述
     * @return 新增种类的主键
     */
    public int addKind(String name , String desc)
        throws AuctionException
    {
        try
        {
            Kind k = new Kind();
            k.setKindName(name);
            k.setKindDesc(desc);
            kindDao.save(k);
            return k.getId();
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("添加种类出现异常,请重试");
        }
    }
    /**
     * 根据产品分类，获取处于拍卖中的全部物品
     * @param kindId 种类ID
     * @return 该类的全部产品
     */
    public List getItemsByKind(int kindId) throws AuctionException
    {
        List result = new ArrayList();
        try
        {
            List items = itemDao.findItemByKind(kindId);
            for (Iterator it = items.iterator() ; it.hasNext() ; )
            {
                Item item = (Item)it.next();
                ItemBean ib = new ItemBean();
                initItem(ib , item);
                result.add(ib);
            }
            return result;
        }
        catch (Exception e)
        {
            log.debug(e.getMessage());
            throw new AuctionException("根据种类获取物品出现异常,请重试");
        }
    }
    /**
     * 根据种类ID获取种类名
     * @param kindId 种类ID
     * @return 该种类的名称
     */
    public String getKind(int kindId) throws AuctionException
    {
        try
        {
            Kind  k = kindDao.get(kindId);
```

```
            if (k != null)
            {
                return k.getKindName();
            }
            return null;
        }
        catch (Exception ex)
        {
            log.debug(ex.getMessage());
            throw new AuctionException("根据种类 ID 获取种类名称出现异常,请重试");
        }
    }
    /**
     * 根据物品 ID 获取物品
     * @param itemId 物品 ID
     * @return 指定 ID 对应的物品
     */
    public ItemBean getItem(int itemId)
        throws AuctionException
    {
        try
        {
            Item item = itemDao.get(itemId);
            ItemBean ib = new ItemBean();
            initItem(ib, item);
            return ib;
        }
        catch (Exception ex)
        {
            log.debug(ex.getMessage());
            throw new AuctionException("根据物品 ID 获取物品详细信息出现异常,请重试");
        }
    }
    /**
     * 增加新的竞价，并对竞价用户发邮件通知
     * @param itemId 物品 ID
     * @param bidPrice 竞价价格
     * @param sess 用于跟踪竞价用户的 HttpSession
     * @return 返回新增竞价记录的 ID
     */
    public int addBid(int itemId , double bidPrice , HttpSession sess)
        throws AuctionException
    {
        Integer userId = (Integer)sess.getAttribute("userId");
        try
        {
            AuctionUser au = userDao.get(userId);
            Item item = itemDao.get(itemId);
            if (bidPrice > item.getMaxPrice())
            {
                item.setMaxPrice(new Double(bidPrice));
                itemDao.save(item);
            }
            //初始化 Bid 对象
            Bid bid = new Bid();
            bid.setBidItem(item);
            bid.setBidUser(au);
            bid.setBidDate(new Date());
```

```
            bid.setBidPrice(bidPrice);
            //持久化 Bid 对象
            bidDao.save(bid);
            //准备发送邮件
            SimpleMailMessage msg = new SimpleMailMessage(this.message);
            msg.setTo(au.getEmail());
            msg.setText("Dear "
                + au.getUsername()
                + ", 谢谢你参与竞价，你的竞价的物品的是: "
                + item.getItemName());
            mailSender.send(msg);
            return bid.getId();
        }
        catch(Exception ex)
        {
            log.debug(ex.getMessage());
            throw new AuctionException("处理用户竞价出现异常,请重试");
        }
    }
    /**
     * 根据时间来修改物品的状态、赢取者
     */
    public void updateWiner()throws AuctionException
    {
        try
        {
            List itemList = itemDao.findItemByState(1);
            for (int i = 0 ; i < itemList.size() ; i++ )
            {
                Item item = (Item)itemList.get(i);
                if (!item.getEndtime().after(new Date()))
                {
                    //根据指定物品和最高竞价来查询用户
                    AuctionUser au = bidDao.findUserByItemAndPrice(
                        item.getId() , item.getMaxPrice());
                    //如果该物品的最高竞价者不为 null
                    if (au != null)
                    {
                        //将该竞价者设为赢取者
                        item.setWiner(au);
                        //修改物品的状态成为“被赢取”
                        item.setItemState(stateDao.get(2));
                    itemDao.save(item);
                    }
                    else
                    {
                        //设置该物品的状态为“流拍”
                        item.setItemState(stateDao.get(3));
                        itemDao.save(item);
                    }
                }
            }
        }
        catch (Exception ex)
        {
            log.debug(ex.getMessage());
            throw new AuctionException("根据时间来修改物品的状态、赢取者出现异常,请重试");
        }
```

```
    }
    /**
     * 将一个 Bid 对象转换成 BidBean 对象
     * @param bb BidBean 对象
     * @param bid Bid 对象
     */
    private void initBid(BidBean bb , Bid bid)
    {
        bb.setId(bid.getId().intValue());
        if (bid.getBidUser() != null )
            bb.setUser(bid.getBidUser().getUsername());
        if (bid.getBidItem() != null )
            bb.setItem(bid.getBidItem().getItemName());
        bb.setPrice(bid.getBidPrice());
        bb.setBidDate(bid.getBidDate());
    }
    /**
     * 将一个 Item PO 转换成 ItemBean 的 VO
     * @param ib ItemBean 的 VO
     * @param item Item 的 PO
     */
    private void initItem(ItemBean ib , Item item)
    {
        ib.setId(item.getId());
        ib.setName(item.getItemName());
        ib.setDesc(item.getItemDesc());
        ib.setPic(item.getItemRemark());
        if (item.getKind() != null)
            ib.setKind(item.getKind().getKindName());
        if (item.getOwner() != null)
            ib.setOwner(item.getOwner().getUsername());
        if (item.getWiner() != null)
            ib.setWiner(item.getWiner().getUsername());
        ib.setAddTime(item.getAddtime());
        ib.setEndTime(item.getEndtime());
        if (item.getItemState() != null)
            ib.setState(item.getItemState().getStateName());
        ib.setInitPrice(item.getInitPrice());
        ib.setMaxPrice(item.getMaxPrice());
    }
}
```

上面的业务逻辑组件中包含了多个业务逻辑方法，都需要通过 HttpSession 来获取系统中当前用户的 ID，当然前提是该用户 ID 存在，也就是用户已经成功登录。因此，调用这些方法之前应该先执行权限检查。但上面的业务逻辑方法中并未进行任何权限控制，这是因为本系统使用 AOP 来集中管理所有业务逻辑方法的权限控制。

上面的业务逻辑方法中，还有一个 updateWiner 方法，该方法拿物品的最后拍卖期限和当前时间进行比较，如果当前时间已经超过了该物品的最后拍卖期限，则修改该物品的状态为流拍或拍卖成功。如果拍卖成功，则还要修改该物品的赢取者。该方法并不由客户端直接调用，而是由 Spring 的任务调度机制负责调用。

将该业务逻辑组件部署在 Spring 容器中，并为该业务逻辑组件注入 5 个 DAO 组件，以及注入 2 个用于发送邮件的 Bean，下面是配置该业务逻辑组件的配置片段：

程序清单：codes\17\auction\WEB-INF\applicationContext.xml

```
<!-- 配置业务逻辑组件 -->
<bean id="auctionManager"
    class="org.crazyjava.auction.service.impl.AuctionManagerImpl">
    <!-- 为业务逻辑组件注入所需的 DAO 组件 -->
```

```
        <property name="userDao" ref="auctionUserDao"/>
        <property name="bidDao" ref="bidDao"/>
        <property name="itemDao" ref="itemDao"/>
        <property name="kindDao" ref="kindDao"/>
        <property name="stateDao" ref="stateDao"/>
        <property name="mailSender" ref="mailSender"/>
        <property name="message" ref="mailMessage"/>
    </bean>
```

➤➤17.4.5 业务层的权限控制

本系统的权限控制使用 AOP 集中管理，AOP 机制使用一个普通类作为拦截器，该拦截器负责拦截所有需要进行权限检查的方法，拦截器方法可以检查目标方法的参数，并根据方法参数的值判断客户端是否具有调用目标方法的权限。拦截器类代码如下：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\authority\AuthorityInterceptor.java

```
public class AuthorityInterceptor
{
    //进行权限检查的方法
    public Object authority(ProceedingJoinPoint jp)
        throws Throwable
    {
        HttpSession sess = null;
        //获取被拦截方法的全部参数
        Object[] args = jp.getArgs();
        //遍历被拦截方法的全部参数
        for (int i = 0 ; i < args.length ; i++ )
        {
            //找到 HttpSession 类型的参数
            if (args[i] instanceof HttpSession) sess =
                (HttpSession)args[i];
        }
        //取出 HttpSession 里的 userId 属性
        Integer userId = (Integer)sess.getAttribute("userId");
        //如果 HttpSession 里的 userId 属性不为 null，且大于 0
        if ( userId != null && userId > 0)
        {
            //继续处理
            return jp.proceed(args);
        }
        else
        {
            //如果还未登录，抛出异常
            throw new AuctionException("您还没有登录，请先登录系统再执行该操作");
        }
    }
}
```

实现了该拦截器后，可在 Spring 配置文件中配置该拦截器，让该拦截器对业务逻辑方法进行拦截。下面是 Spring 配置文件中配置权限检查拦截器的配置片段：

程序清单：codes\17\auction\WEB-INF\applicationContext.xml

```
<aop:config>
    ...
    <!-- 将 authority 转换成切面 Bean
        切面 Bean 的新名称为：authorityAspect -->
    <aop:aspect id="authorityAspect" ref="authority">
        <!-- 定义一个 Around 增强处理，直接指定切入点表达式
```

```
            以切面 Bean 中的 authority()方法作为增强处理方法 -->
        <aop:around pointcut=
            "execution(* org.crazyjava.auction.service.impl.*.getItemByWiner(..))
            or execution(* org.crazyjava.auction.service.impl.*.getBidByUser(..))
            or execution(* org.crazyjava.auction.service.impl.*.getItemsByOwner(..))
            or execution(* org.crazyjava.auction.service.impl.*.addItem(..))
            or execution(* org.crazyjava.auction.service.impl.*.addBid(..))"
            method="authority"/>
    </aop:aspect>
</aop:config>
<!-- 定义一个普通 Bean 实例，该 Bean 实例将进行权限控制 -->
<bean id="authority"
    class="org.crazyjava.auction.authority.AuthorityInterceptor"/>
```

上面的配置片段表示：AuthorityInterceptor 拦截器将负责拦截 getItemByWiner、getBidByUser 等方法，从而实现在调用这些方法之前进行权限检查。

➤➤17.4.6 业务层的任务调度

Spring 的任务调度抽象层简化了任务调度，本系统使用 JDK 的 Timer 支持完成任务调度，Spring 为 JDK 的 Timer 支持提供如下两个工具类：

- ScheduledTimerTask：该工具类可将继承 TimerTask 的任务包装成可调度的任务。
- TimerFactoryBean：该工具类实现了 InitializingBean 接口，用于启动线程。

借助于 Spring 任务调度的抽象，本系统只需要编写一个继承 TimerTask 的任务类即可，该任务类的源代码如下：

程序清单：codes\17\auction\WEB-INF\applicationContext.xml

```
public class CheckWiner extends TimerTask
{
    //该任务所依赖的业务逻辑组件
    private AuctionManager mgr;
    //依赖注入业务逻辑组件必需的 setter 方法
    public void setMgr(AuctionManager mgr)
    {
        this.mgr = mgr;
    }
    //该任务的执行体
    public void run()
    {
        try
        {
            mgr.updateWiner();
        }
        catch (AuctionException ae)
        {
            ae.printStackTrace();
        }
    }
}
```

该调度任务，仅仅在 run 方法内调用了 AuctionManager 的 updateWiner 方法——该方法根据时间修改物品的状态以及赢取者。

注意：

该任务调度类依赖于业务逻辑组件，因此必须为其提供相应的 setter 方法。

该任务类本身既不知道从何时开始调度，也不知道调度的频率，也没有启动调度线程——这些都通过 Spring 的配置完成，而不是以硬编码方式写在代码中。下面是本系统中关于任务调度的配置片段：

程序清单：codes\17\auction\WEB-INF\applicationContext.xml

```
<!-- 配置一个 TimerTask Bean -->
<bean id="checkWiner" class="org.crazyjava.auction.schedule.CheckWiner">
    <!-- 依赖注入业务逻辑组件 -->
    <property name="mgr" ref="auctionManager"/>
</bean>
<!-- 将 TimerTask Bean checkWiner 包装成可周期性执行的任务调度 Bean -->
<bean id="scheduledTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <!-- 指定调度频率和延迟 -->
    <property name="delay" value="0"/>
    <property name="period" value="86400000"/>
    <property name="timerTask" ref="checkWiner"/>
</bean>
<!-- 启动实际调度 -->
<bean id="timerFactory"
    class="org.springframework.scheduling.timer.TimerFactoryBean">
    <!-- 下面列出所有需要调用的任务调度 Bean -->
    <property name="scheduledTimerTasks">
        <list>
            <ref bean="scheduledTask"/>
        </list>
    </property>
</bean>
```

➤➤17.4.7 事务管理

每个业务逻辑方法都应该是一个整体，而不能只完成部分操作，因此我们应该为业务逻辑方法增加事务控制，Spring 的声明式事务管理极好地简化了事务控制，开发者实现业务逻辑方法时无须理会事务逻辑，直接在 Spring 配置文件中配置事务管理即可。

Spring 提供了 tx、aop 两个命名空间来简化事务控制，下面的配置片段用于为业务逻辑方法增加事务控制：

程序清单：codes\17\auction\WEB-INF\applicationContext.xml

```
<!-- 配置 Hibernate 的局部事务管理器，使用 HibernateTransactionManager 类 -->
<!-- 该类实现了 PlatformTransactionManager 接口，是针对 Hibernate 的特定实现-->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 配置 HibernateTransactionManager 时需要依赖注入 SessionFactory 的引用 -->
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 配置事务切面 Bean，指定事务管理器 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 用于配置详细的事务语义 -->
    <tx:attributes>
        <!-- 所有以'get'开头的方法是 read-only 的 -->
        <tx:method name="get*" read-only="true"/>
        <!-- 其他方法使用默认的事务设置 -->
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
<aop:config>
    <!-- 配置一个切入点，配置指定包下所有以 Impl 结尾的类执行的所有方法 -->
    <aop:pointcut id="leeService"
```

```
            expression="execution(* org.crazyjava.auction.service.impl.*Impl.*(..))"/>
        <!-- 指定在 leeService 切入点应用 txAdvice 事务切面 -->
        <aop:advisor advice-ref="txAdvice"
            pointcut-ref="leeService"/>
        ...
    </aop:config>
```

17.5 暴露业务逻辑方法

借助于 DWR 框架的帮助，我们可以将 Spring 容器中的 Bean 暴露给远程浏览器，从而允许 JavaScript 以异步方式调用 Spring 容器中 Bean 的方法。将业务逻辑组件暴露出来之后，还可以通过 DWR 的调试控制台测试业务逻辑方法。

➤➤17.5.1 初始化 Spring 容器

为了将业务逻辑组件暴露给浏览器 JavaScript，必须让 Spring 容器随 Web 应用的启动而初始化。Spring 容器初始化时，会将容器中所有 Bean 全部实例化。一旦 Spring 将容器中的 Bean 创建完成，DWR 即可将这些 Bean 暴露给浏览器 JavaScript。

为了让 Spring 容器随 Web 应用的启动而初始化，可以使用 Listener 配置。在 web.xml 文件中增加如下配置片段：

程序清单：codes\17\auction\WEB-INF\web.xml

```
<!-- 指定 Spring 配置文件的位置 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml,
        /WEB-INF/daoContext.xml</param-value>
</context-param>
<!-- 配置 Web 应用启动时加载 Spring 容器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

由于本应用将 DAO 组件、业务逻辑组件等分开配置，所以上面的配置片段指定了两个配置文件，两个配置文件都放在 WEB-INF 路径下，文件名分别是 applicationContext.xml 和 daoContext.xml。

增加了上面的配置片段后，Spring 容器会随 Web 应用的启动而初始化。Spring 容器初始化时，其中的 Bean 也会随之初始化完成。

➤➤17.5.2 配置 DWR 的核心 Servlet

DWR 的核心 Servlet 负责拦截用户的 Ajax 请求。DWR 的核心 Servlet 是 DWRServlet，它是客户端 JavaScript 和服务器端 Java 对象之间的桥梁。客户端的 Ajax 请求都交给 DWRServlet 处理，该 Servlet 根据请求调用服务器端 Java 对象的方法，并将调用的结果返回给 JavaScript 客户端。

配置 DWR 的核心 Servlet 与普通 Servlet 并无太大区别，只是将该 Servlet 的 URL 配置成支持通配地址即可。下面是配置 DWR 的核心 Servlet 的代码片段：

程序清单：codes\17\auction\WEB-INF\web.xml

```
<servlet>
    <!-- 指定 DWR 核心 Servlet 的名字 -->
    <servlet-name>dwr-invoker</servlet-name>
    <!-- 指定 DWR 核心 Servlet 的实现类 -->
    <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
```

```
        <!-- 指定 DWR 核心 Servlet 处于调试状态 -->
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <!-- 指定核心 Servlet 的 URL 映射 -->
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <!-- 指定核心 Servlet 映射的 URL -->
        <url-pattern>/leedwr/*</url-pattern>
    </servlet-mapping>
```

根据上面的配置片段，DWRServlet 将拦截来自/leedwr/* URL 下的所有 Ajax 请求。

➤➤17.5.3 暴露业务逻辑方法

DWR 使用 dwr.xml 文件来暴露业务逻辑组件，在 dwr.xml 文件中指定需要暴露的业务逻辑组件的名字，并指定转换得到的 JavaScript 对象的名字。

为了提供更好的安全性，本系统还使用了白名单来指定需要暴露的业务逻辑方法。下面是 dwr.xml 配置文件的代码：

程序清单：codes\17\auction\WEB-INF\dwr.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
    "http://getahead.ltd.uk/dwr/dwr20.dtd">
<dwr>
    <allow>
        <!-- 使用 spring 创建器，创建一个名为 am 的 JavaScript 对象 -->
        <create creator="spring" javascript="am">
            <!-- 将 Spring 容器中的 auctionManager 创建成名为 am 的对象 -->
            <param name="beanName" value="auctionManager"/>
            <!-- 使用 include 元素定义哪些方法将被暴露到客户端 -->
            <include method="getItemByWiner"/>
            <include method="getFailItems"/>
            <include method="validLogin"/>
            <include method="getBidByUser"/>
            <include method="getItemsByOwner"/>
            <include method="getAllKind"/>
            <include method="addItem"/>
            <include method="addKind"/>
            <include method="getItemsByKind"/>
            <include method="getKind"/>
            <include method="getItem"/>
            <include method="addBid"/>
            <include method="getKind"/>
        </create>
        <!-- 定义使用 Bean 转换器处理如下 Java 类 -->
        <convert converter="bean"
            match="org.crazyjava.auction.business.BidBean"/>
        <convert converter="bean"
            match="org.crazyjava.auction.business.ItemBean"/>
        <convert converter="bean"
            match="org.crazyjava.auction.business.KindBean"/>
        <convert converter="bean"
            match="org.crazyjava.auction.exception.AuctionException"/>
    </allow>
</dwr>
```

上面的配置文件中粗体字代码指定 DWR 将创建一个名为 am 的 JavaScript 对象，该对象使用 spring 创建器产生。并通过白名单的方式指定了一系列方法，这些方法都将被暴露给客户端。在配置文件下部，定义了四个 convert 元素，这四个元素指定了 BidBean、ItemBean、KindBean 和 AuctionException 四个 Java 实例可被转换成 JavaScript 对象。

经过上面的配置，已经成功将业务逻辑组件暴露给客户端。如果指定了 DWR 处于调试状态，则可在 DWR 的调试控制台测试业务逻辑方法。进入 am 对象的调试控制台，可看到如图 17.10 所示调试界面。

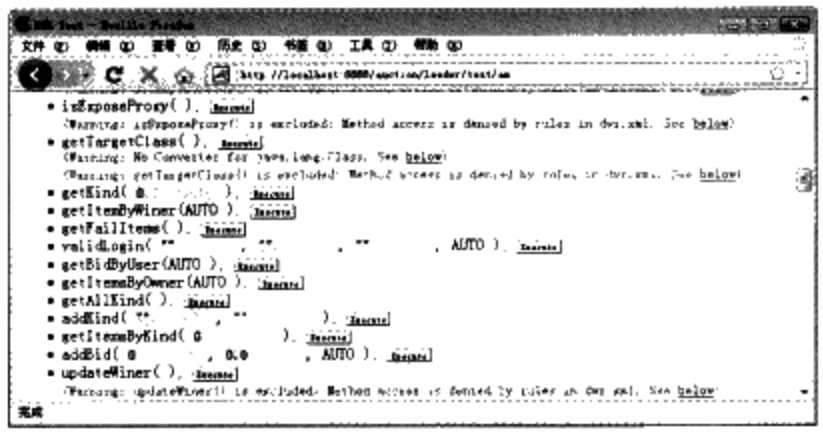


图 17.10　am 对象的调试控制台

从图 17.10 中可以看出，am 对象包含了多个方法，这些方法实际由业务逻辑组件提供实现。当浏览器 JavaScript 调用这些方法时，DWR 将委托给业务逻辑组件对应的方法。

开发者可以在如图 17.10 所示的界面中输入方法参数，然后单击方法后的“Execute”按钮，对业务逻辑方法进行测试。

17.6　调用业务逻辑方法响应用户请求

一旦完成了上面的定义，就可以在浏览器的 JavaScript 代码中调用上面的业务逻辑方法，JavaScript 以异步方式调用业务逻辑方法，并将方法返回值动态更新在 HTML 页面上，从而避免了刷新整个页面，从而提供与服务器异步交互的能力。

➤➤17.6.1　页面加载时的函数

页面加载时需设置 DWR 的全局错误处理函数，设置页面的 loadMessage，并初始化全局变量等。页面加载时执行的函数代码如下：

程序清单：codes\17\auction\am.js

```
function init()
{
    //初始化全局变量 curShow
    curShow = $("index");
    //设置全局错误处理函数
    dwr.engine.setErrorHandler(errHandler);
    //设置页面的 loadMesssage
    dwr.util.useLoadingMessage();
}
```

上面的第一行粗体字代码设置了一个 curShow 全局变量，用于保存用户当前浏览的页面内容。本应用是一个纯粹的 Ajax 应用，整个应用只有一个 HTML 页面，当用户处理不同业务时，页面将选择呈现不同的内容，而 curShow 全局变量保存的正是用户当前正在浏览的页面内容。图 17.11 显示了浏览者进入该系统看到的首页。

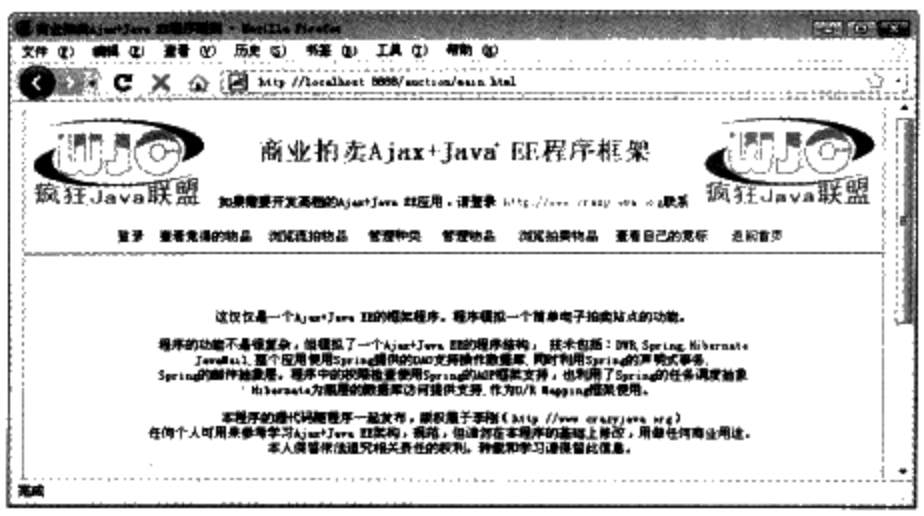


图 17.11　拍卖系统首页

上面的函数中使用了一个全局错误处理函数 errHandler。当调用服务器方法出现异常、错误时，该方法自动启动。该函数负责将业务逻辑方法的异常信息呈现给浏览者。该函数的代码如下：

```
//全局错误处理函数
function errHandler(msg)
{
    alert(msg);
}
```

在上面的错误处理函数中有一个 msg 参数，该参数就是服务器的错误提示。当调用服务器方法抛出异常时，该信息就是异常信息。

➤➤17.6.2　处理返回首页的请求

当用户进入系统的其他模块后，如果用户单击页面上方的“返回首页”链接，将触发 showIndex() 函数，该函数将让系统重新返回系统的首页。处理这个请求很简单，将用户正在浏览的内容隐藏，并显示首页内容即可。该函数的代码如下：

程序清单：codes\17\auction\am.js

```
//显示首页
function showIndex()
{
    //curShow 是个全局变量，该变量代表用户当前浏览的页面内容
    if (curShow != null)
    {
        //将当前内容隐藏
        curShow.style.display = "none";
    }
    //显示首页内容
    $("index").style.display = "";
    //将首页内容设为当前浏览内容
    curShow = $("index");
}
```

在上面的 showIndex 函数中，引用了一个全局变量 curShow，该变量代表了用户当前浏览的页面内容，页面加载时系统将会初始化该变量。

➤➤17.6.3　浏览所有流拍物品

浏览所有流拍物品需要调用业务逻辑组件的 getFailItems()方法，该方法无须执行权限检查。所有用户都可以直接调用。当用户单击页面上方的“浏览流拍物品”链接时，将发送获取所有流拍物品的请求，对应的函数如下：

程序清单：codes\17\auction\am.js

```
//浏览流拍物品
function viewFail()
{
    //定义一个客户端旗标，表明是获取流拍物品
    var flag = "fail";
    //使用闭包将服务器数据和浏览器数据同时传入回调函数
    var callbackAdapter = function(data)
    {
        viewFailCb(data, flag);
    };
    //调用 am 对象的 getFailItems 方法
    am.getFailItems(callbackAdapter);
}
```

在上面的代码中，实际的回调函数是 viewFailCb，该函数有两个参数：第一个参数是业务逻辑方法的返回值，后一个参数是旗标，用于指定是否为获取所有流拍物品。

上面的粗体字代码之所以弄得这么“复杂”，是因为浏览流拍物品和查看自己的物品使用了同一个表格来显示。当调用服务器业务逻辑方法后，回调函数不仅需要返回物品列表，还需要知道到底是显示流拍物品，还是显示自己的物品，其代码如下：

程序清单：codes\17\auction\am.js

```
function viewFailCb(data , flag)
{
    //如果服务器返回的数据不为空，且至少包含了一个物品
    if (data != null && data.length > 0)
    {
        //指定单元格函数，每个函数对应一列
        var cellfuncs = [
            //输出物品名
            function(data){
                return data['name'];
            },
            //输出物品种类
            function(data){
                return data['kind'];
            },
            //输出物品最大价格
            function(data){
                return data['maxPrice'];
            },
            //输出物品的备注
            function(data){
                return data['pic'];
            }
    ];
    //删除表格 viewFailBody 的全部行
    dwr.util.removeAllRows("viewFailBody");
    //使用 data 数组中的数据为 viewFailBody 增加表格行
    dwr.util.addRows("viewFailBody" , data , cellfuncs , option);
    //如果是浏览所有流拍物品
    if (flag == "fail")
    {
        //设置表格标题
        dwr.util.setValue("failAndOwnerTitle" , "所有流拍的物品");
        //浏览流拍物品，没有添加物品的超级链接
        $("addItemHref").style.display = "none";
```

```
        }
        //查看自己的拍卖物品
        if (flag == "owner")
        {
            dwr.util.setValue("failAndOwnerTitle" , "您当前的拍卖物品：");
            //查看自己的拍卖物品时，需要添加物品的超级链接
            $("addItemHref").style.display = "";
        }
        //隐藏当前显示的内容
        if (curShow != null)
        {
            curShow.style.display = "none";
        }
        $("viewFail").style.display = "";
        $("addItem").style.display = "none";
        curShow = $("viewFail");
    }
    //如果没有物品显示
    else if (flag == "owner")
    {
        alert("您暂时还没有拍卖物品！");
    }
    else if (flag == "fail")
    {
        alert("暂时没有流拍物品！");
    }
}
```

正如在代码中所见，这个回调函数不仅是浏览所有流拍物品的回调函数，也是浏览自己在拍卖中的物品的回调函数。因此，该函数需要处理两种情形。如果系统中包含流拍物品，将可看到如图 17.12 所示的界面。

图 17.12　查看流拍物品

上面的回调函数中调用 dwr.util 的 addRows()方法时还指定了一个 option 参数，该参数用于为 addRows()方法指定自定义的创建选项，option 参数是个全局变量，其值如下：

```
//指定自定义创建选项
var option =
{
    //指定 rowCreator 选项
    rowCreator:function(options)
    {
        var row = document.createElement("tr");
        //如果当前行索引为偶数，设置其背景色为白色
        if(options.rowNum % 2 == 0)
        {
            row.style.backgroundColor = "#ffffff";
        }
        //如果当前行索引为奇数，设置其背景色为淡绿色
        else
        {
```

```
            row.style.backgroundColor = "#e2ffe2";
        }
        return row;
    }
};
```

➤➤17.6.4 处理用户登录

该系统有一些功能只有登录用户才能完成，如果用户单击页面上方的“登录”超级链接，将触发showLogin()函数，该函数仅仅是显示登录窗口，并未处理实际登录，其代码如下：

程序清单：codes\17\auction\am.js

```
//显示登录表格
function showLogin()
{
    //隐藏当前显示的内容
    if (curShow != null)
    {
        curShow.style.display = "none";
    }
    //显示登录表格
    $("login").style.display = "";
    curShow = $("login");
}
```

该函数执行后可看到如图 17.13 所示的界面。

图 17.13 系统登录界面

在图 17.13 中可看到，系统登录使用了一个随机图形验证码。使用随机图形验证码是为了增加系统安全性，防止 Cracker 使用暴力破解。该图形验证码的实质就是一个 Servlet，与普通的 Servlet 不同的是，该 Servlet 不生成文本内容，而是生成图形内容，其代码如下：

程序清单：codes\17\auction\WEB-INF\src\org\crazyjava\auction\web\AuthImg.java

```
public class AuthImg extends HttpServlet
{
    //定义图形验证码中绘制字符的字体
    private final Font mFont =
        new Font("Arial Black", Font.PLAIN, 16);
    //定义图形验证码的大小
    private final int IMG_WIDTH = 100;
    private final int IMG_HEIGTH = 18;
    //定义一个获取随机颜色的方法
    private Color getRandColor(int fc,int bc)
    {
        Random random = new Random();
        if(fc > 255) fc = 255;
        if(bc > 255) bc = 255;
        int r = fc + random.nextInt(bc - fc);
```

```
        int g = fc + random.nextInt(bc - fc);
        int b = fc + random.nextInt(bc - fc);
        //得到随机颜色
        return new Color(r , g , b);
    }
    //重写 service 方法，生成对客户端的响应
    public void service(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        //设置禁止缓存
        response.setHeader("Pragma","No-cache");
        response.setHeader("Cache-Control","no-cache");
        response.setDateHeader("Expires", 0);
        response.setContentType("image/jpeg");
        BufferedImage image = new BufferedImage
            (IMG_WIDTH , IMG_HEIGTH , BufferedImage.TYPE_INT_RGB);
        Graphics g = image.getGraphics();
        Random random = new Random();
        g.setColor(getRandColor(200 , 250));
        //填充背景色
        g.fillRect(1, 1, IMG_WIDTH - 1, IMG_HEIGTH - 1);
        //为图形验证码绘制边框
        g.setColor(new Color(102 , 102 , 102));
        g.drawRect(0, 0, IMG_WIDTH - 1, IMG_HEIGTH - 1);
        g.setColor(getRandColor(160,200));
        //生成随机干扰线
        for (int i = 0 ; i < 80 ; i++)
        {
            int x = random.nextInt(IMG_WIDTH - 1);
            int y = random.nextInt(IMG_HEIGTH - 1);
            int xl = random.nextInt(6) + 1;
            int yl = random.nextInt(12) + 1;
            g.drawLine(x , y , x + xl , y + yl);
        }
        g.setColor(getRandColor(160,200));
        //生成随机干扰线
        for (int i = 0 ; i < 80 ; i++)
        {
            int x = random.nextInt(IMG_WIDTH - 1);
            int y = random.nextInt(IMG_HEIGTH - 1);
            int xl = random.nextInt(12) + 1;
            int yl = random.nextInt(6) + 1;
            g.drawLine(x , y , x - xl , y - yl);
        }
        //设置绘制字符的字体
        g.setFont(mFont);
        //用于保存系统生成的随机字符串
        String sRand = "";
        for (int i = 0 ; i < 6 ; i++)
        {
            String tmp = getRandomChar();
            sRand += tmp;
            //获取随机颜色
            g.setColor(new Color(20 + random.nextInt(110)
                ,20 + random.nextInt(110)
                ,20 + random.nextInt(110)));
            //在图片上绘制系统生成的随机字符
```

```
            g.drawString(tmp , 15 * i + 10,15);
        }
        //获取 HttpSesssion 对象
        HttpSession session = request.getSession(true);
        //将随机字符串放入 HttpSesssion 对象中
        session.setAttribute("rand" , sRand);
        g.dispose();
        //向输出流输出图片
        ImageIO.write(image, "JPEG", response.getOutputStream());
    }
    //定义获取随机字符串的方法
    private String getRandomChar()
    {
        //随机生成 0、1、2 之中的一个数字
        int rand = (int)Math.round(Math.random() * 2);
        long itmp = 0;
        char ctmp = '\u0000';
        switch (rand)
        {
            //生成大写字母
            case 1:
                itmp = Math.round(Math.random() * 25 + 65);
                ctmp = (char)itmp;
                return String.valueOf(ctmp);
            //生成小写字母
            case 2:
                itmp = Math.round(Math.random() * 25 + 97);
                ctmp = (char)itmp;
                return String.valueOf(ctmp);
            //生成数字
            default :
                itmp = Math.round(Math.random() * 9);
                return  itmp + "";
        }
    }
}
```

编写完该 Servlet 后，将该 Servlet 配置在 Web 应用中，配置该 Servlet 的配置片段如下：

程序清单：codes\17\auction\WEB-INF\web.xml

```
<!-- 配置图形验证码 Servlet -->
<servlet>
    <servlet-name>img</servlet-name>
    <servlet-class>org.crazyjava.auction.web.AuthImg</servlet-class>
</servlet>
<!-- 为图形验证码 Servlet 指定 URL -->
<servlet-mapping>
    <servlet-name>img</servlet-name>
    <url-pattern>/authImg</url-pattern>
</servlet-mapping>
```

经过上面的配置，指定了图形验证码的 Servlet 被映射在/authImg 处。在 HTML 页面中可通过如下方式显示图形验证码：

```
<img name="d" src="authImg" alt="验证码"/>
```

使用图形验证码与使用普通图片并没有太大的区别，唯一的区别是将原来指定普通图片的 src 属性值的地方改为指定图形验证码 Servlet 的 URL。

如果用户单击如图 17.13 所示中的“登录”按钮，将触发 submitLogin()函数，该函数先对 HTML

页面数据进行基本校验，然后异步调用服务器的 validLogin()方法，其代码如下：

程序清单：codes\17\auction\am.js

```
//提交登录
function submitLogin()
{
    //获取 loginUser 文本框中的输入值
    var loginUser = trim($("loginUser").value);
    //获取 loginPass 文本框中的输入值
    var loginPass = trim($("loginPass").value);
    //获取验证码输入框中的输入值
    var loginVer = trim($("vercode").value);
    var errStr = "";
    //进行基本校验，判断用户名、密码和图形验证码必须输入，且输入必须符合指定格式
    if (loginUser == null || loginUser == "")
    {
        errStr += "您没有输入用户名！\n";
    }
    else if (loginPass == null || loginPass == "")
    {
        errStr += "您没有输入密码！\n";
    }
    else if (loginVer == null || loginVer == "")
    {
        errStr += "您没有输入验证码！\n";
    }
    else if (loginUser.length >= 10 || loginUser.length <= 3)
    {
        errStr += "用户名的长度必须在 3 到 10 之间！\n";
    }
    else if (loginPass.length >= 10 || loginPass.length <= 3)
    {
        errStr += "密码的长度必须在 3 到 10 之间！\n";
    }
    else if (loginVer.length != 6)
    {
        errStr += "验证码长度必须为 6！\n";
    }
    //如果不能通过客户端校验，弹出出错提示
    if (errStr != "")
    {
        alert(errStr);
        return false;
    }
    //如果通过了客户端校验，调用远程 Java 方法
    else
    {
        am.validLogin(loginUser , loginPass , loginVer , submitLoginCb);
    }
}
```

在上面的函数中，使用了一个 trim()函数，用于去掉字符串前后的空格，该函数使用了正则表达式来去掉字符串前后的空格，其代码如下：

```
//使用正则表达式去掉字符串前后的空格
function trim(s)
{
    return s.replace( /^\s*/, "" ).replace( /\s*$/, "" );
}
```

处理用户登录的回调函数是 submitLoginCb，该函数根据登录处理结果提示用户登录是否成功，代码如下：

程序清单：codes\17\auction\am.js

```
//处理登录的回调函数
function submitLoginCb(data)
{
    //如果服务器响应数据为 true，表明登录成功
    if (data)
    {
        //清空用户输入的登录数据
        dwr.util.setValue("loginUser" , "");
        dwr.util.setValue("loginPass" , "");
        dwr.util.setValue("vercode" , "");
        alert("登录成功，你可以继续使用本系统");
        showIndex();
    }
    //登录失败
    else
    {
        alert("用户名和密码不符合，登录失败！");
    }
}
```

如果用户输入的验证码不符合，DWR 将自动把服务器的异常提示信息显示出来；如果登录的用户名与密码不符，系统将弹出错误提示。如果用户输入的用户名、密码匹配，则可以正常登录系统，看到登录成功的提示信息。

➤➤17.6.5 管理物品

管理物品模块用户只有登录后才可以操作，如果未登录用户单击页面上方的“管理物品”链接，则 Spring 的 AOP 机制将负责提示用户登录。

权限检查的执行过程是，客户端调用业务逻辑方法时，如果业务逻辑方法符合 Spring AOP 中指定的方法名，权限检查拦截器将检查目标方法的参数，如果调用方法时 HttpSession 没有 userId 属性，则拦截器将抛出异常，异常信息为“您还没有登录，请先登录系统再执行操作”，该异常信息将由 DWR 捕获，并通过警告框输出到客户端。

未登录用户单击“管理物品”链接时，将看到如图 17.14 所示的警告框。

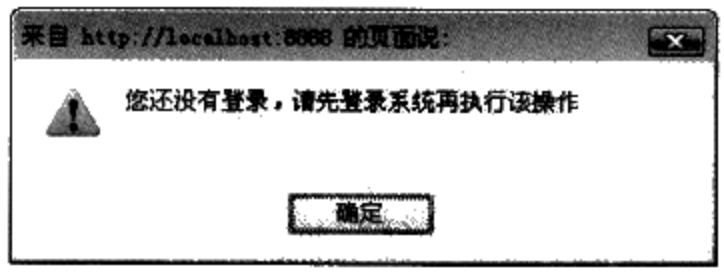


图 17.14 基于 AOP 的权限检查

用户单击“管理物品”链接时，将触发如下函数：

程序清单：codes\17\auction\am.js

```
//浏览自己的物品
function viewOwnerItem()
{
    var flag = "owner";
    //使用闭包将服务器数据和浏览器数据同时传入回调函数
    var callbackAdapter = function(data)
```

```
    {
        viewFailCb(data, flag);
    };
    am.getItemsByOwner(callbackAdapter);
}
```

管理物品与查看流拍物品使用了同一个回调函数，因此使用了闭包将旗标传给回调函数，从而让回调函数可以判断当前是处理查看用户自己的物品，还是处理查看流拍物品。如果用户成功登录，然后单击“管理物品”超级链接，将看到如图 17.15 所示的界面。

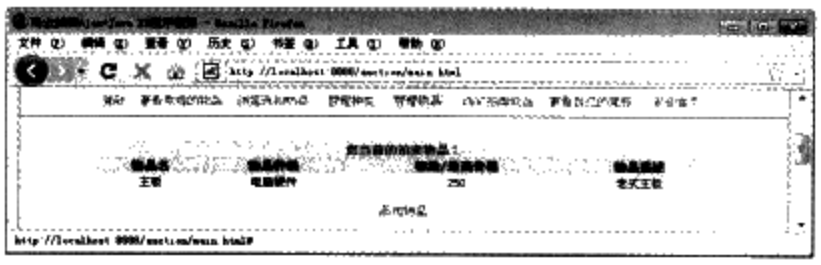


图 17.15　管理物品

在图 17.15 中可看到表格的下方有一个“添加物品”的链接，单击该链接将出现添加物品的输入框，单击该链接将触发如下函数。

程序清单：codes\17\auction\am.js

```
//显示添加物品的表单控件
function showAddItem()
{
    //获取所有物品种类
    am.getAllKind(showAddItemCb);
}
```

上面的函数并未告诉页面需要显示添加物品的表格，那是因为需要添加物品时，应该先加载物品种类，添加物品时应指定物品种类。因此，上面显示添加物品时调用了 getAllKind()业务逻辑方法，该方法将返回系统中所有的物品种类。

showAddItemCb 回调函数负责在页面中显示添加物品的表格，并将物品种类在页面中通过下拉菜单加载出来，其代码如下：

程序清单：codes\17\auction\am.js

```
//显示添加物品的回调函数
function showAddItemCb(data)
{
    if (data != null && data.length > 0)
    {
        //从 addItemCatalog 的 select 列表中删除所有选项
        dwr.util.removeAllOptions("addItemCatalog");
        //将 data 中的数据项添加为 addItemCatalog 的 select 列表中的选项
        dwr.util.addOptions("addItemCatalog" , data ,
            'id' , 'kindName');
        $("addItem").style.display = "";
    }
    //必须有物品种类，才可以添加物品
    else
    {
        alert("还没有物品种类，请先添加物品种类");
    }
}
```

单击“添加物品”的链接后，将看到如图 17.16 所示的界面。

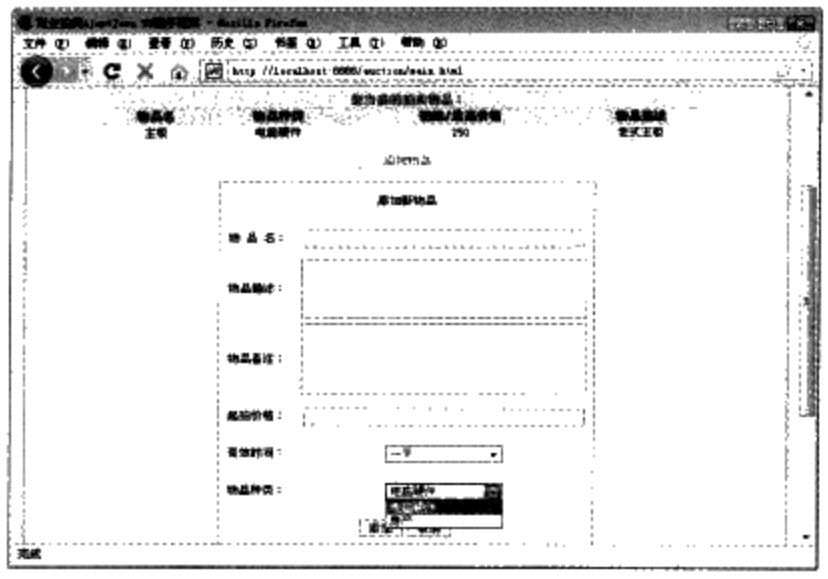


图 17.16　添加物品的输入框

从如图 17.16 所示的下拉菜单中可以看到当前系统中包含的所有物品种类，用户可以在对应的文本框中输入物品信息，然后单击“添加”按钮，此时将触发如下函数。

程序清单：codes\17\auction\am.js

```
function addItem()
{
    //先通过输入框获取添加物品的各项基本信息
    var addItemName = trim($("addItemName").value);
    var addItemDesc = trim($("addItemDesc").value);
    var addItemPic = trim($("addItemPic").value);
    //获取物品的初始价格
    var addItemPrice = trim($("addItemPrice").value);
    //获取物品的拍卖时间
    var addItemAvail = trim(dwr.util.getValue("addItemAvail"));
    var addItemCatalog = trim(dwr.util.getValue("addItemCatalog"));
    var errStr = "";
    //进行客户端输入校验
    if (addItemName == null || addItemName == "")
    {
        errStr += "您没有输入物品名！\n";
    }
    else if (addItemPrice == null || addItemPrice == "")
    {
        errStr += "您没有输入物品的起拍价\n";
    }
    else if (addItemName.length >= 10 || addItemName.length < 2)
    {
        errStr += "物品名的长度必须在 2 到 10 之间！\n";
    }
    if (isNaN(addItemPrice))
    {
        errStr += "物品的起拍价必须是数字！\n";
    }
    //如果客户端输入校验不能通过
    if (errStr != "")
    {
        //输出校验提示信息
        alert(errStr);
        return false;
    }
    //如果通过了客户端输入校验
    else
```

```
    {
        //调用 addItem 业务方法来处理添加物品
        am.addItem(addItemName, addItemDesc, addItemPic,
            parseFloat(addItemPrice), addItemAvail, addItemCatalog, addItemCb);
    }
}
```

用户单击“添加”按钮后，如果通过了客户端输入校验，将调用 addItem()业务逻辑方法，该方法负责添加新物品。添加物品的回调函数如下：

程序清单：codes\17\auction\am.js

```
function addItemCb(data)
{
    //添加物品后方法返回新增物品的主键
    if (data != null && typeof data == "number" && data > 0)
    {
        alert("添加物品成功");
        //取消添加
        cancelItem();
        //查看自己所有处于拍卖中的物品
        viewOwnerItem();
    }
    else
    {
        alert("添加物品失败");
    }
}
```

在上面的回调函数中，使用了 cancelItem 函数，该函数是取消添加物品的函数。用户单击如图 17.16 所示中的“取消”按钮将触发该函数。当用户添加完物品后，调用该函数清空各表单控件内的值，并隐藏添加新物品的输入框。该函数的代码如下：

程序清单：codes\17\auction\am.js

```
//取消添加物品
function cancelItem()
{
    //将添加新物品的各输入控件清空
    dwr.util.setValue("addItemName" , "");
    dwr.util.setValue("addItemDesc" , "");
    dwr.util.setValue("addItemPrice" , "");
    dwr.util.setValue("addItemPic" , "");
    //隐藏添加物品的页面组件
    $("addItem").style.display = "none";
}
```

一旦成功添加新物品，系统将弹出提示框告诉用户添加新物品成功。添加的新物品将立即在页面上方显示出来——这是因为 addItemCb()函数再次调用了 viewOwnerItem()函数，该函数负责把当前用户处于拍卖状态下的物品显示出来。

➤➤17.6.6 管理物品种类

如果用户单击页面上方的“管理种类”链接，将进入管理物品种类模块。管理物品种类会先看到系统包含的所有物品种类。用户单击“管理种类”链接时，将触发如下 JavaScript 函数：

程序清单：codes\17\auction\am.js

```
//查看全部物品种类
function viewCatalog()
```

```
{
    //调用 getAllKind 业务逻辑方法，获取当前所有种类
    am.getAllKind(viewCatalogCb);
}
```

调用 getAllKind 的业务逻辑方法时，使用了 viewCatalogCb 回调函数，该回调函数将负责将系统当前的所有种类在页面中显示出来，其代码如下：

程序清单：codes\17\auction\am.js

```
//查看种类的回调函数
function viewCatalogCb(data)
{
    //如果可以正常获取种类信息
    if (data != null && data.length > 0)
    {
        //指定表格函数数组，每个函数对应一列
        var cellfuncs = [
            //输出种类名
            function(data){
            return data['kindName'];
            },
            //输出种类描述
            function(data){
                return data['kindDesc'];
            }
        ];
        //删除 viewCatalogBody 表格中的所有表格行
        dwr.util.removeAllRows("viewCatalogBody");
        //将 data 中的数据添加到 viewCatalogBody 表格中
        dwr.util.addRows("viewCatalogBody" , data , cellfuncs , option);
        //隐藏当前显示的内容
        if (curShow != null)
        {
            curShow.style.display = "none";
        }
        //将种类表格显示出来
        $("viewCatalog").style.display = "";
        $("addKind").style.display = "none";
        curShow = $("viewCatalog");
    }
    else
    {
        alert("暂时还没有种类信息！");
    }
}
```

回调函数将 getAllKind 方法返回的内容动态显示在页面中，让用户可以查看系统当前的物品种类。如果系统已包含了物品种类，则用户单击“管理种类”链接将看到如图 17.17 所示的界面。

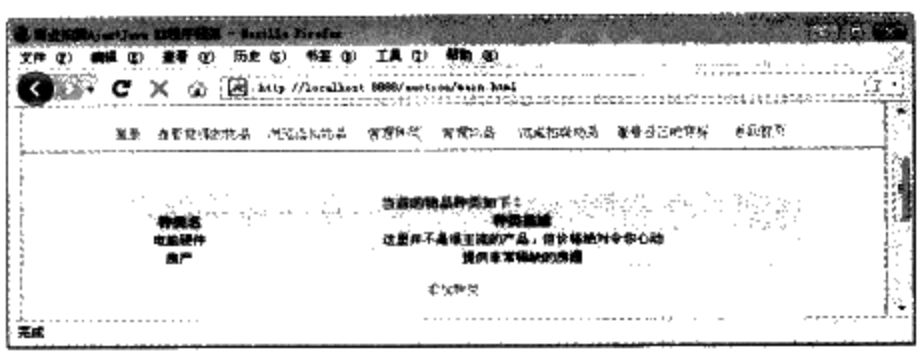


图 17.17　查看物品种类

在如图 17.17 所示的物品种类的下方可看到有个“添加种类”的链接，单击该链接将出现添加物品种类的输入框，显示添加种类的 JavaScript 事件处理器是直接写在 HTML 页面中的。“添加种类”链接的代码如下：

```
<a href="#" onclick="$('addKind').style.display = '';return false;">添加种类</a>
```

单击“添加种类”链接后，将看到如图 17.18 所示的界面。

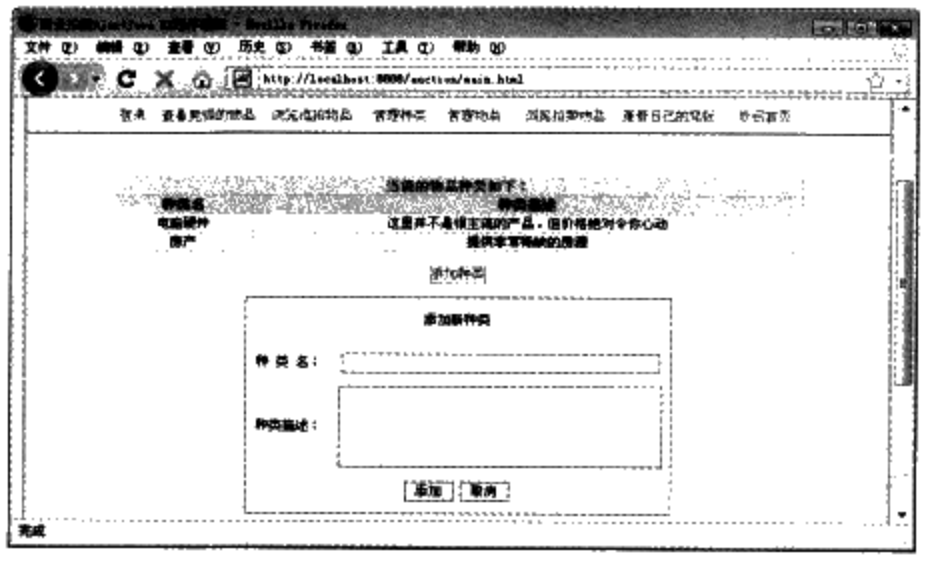


图 17.18　添加种类

在如图 17.18 所示的“添加新种类”的输入框中可看到两个按钮：“添加”和“取消”，这两个按钮分别用于添加物品种类和取消添加两个动作。单击“添加”按钮将触发添加种类函数，而单击“取消”按钮将触发如下函数。

程序清单：codes\17\auction\am.js

```
//取消添加种类
function cancelCatalog()
{
    //将用户输入的种类名、种类描述清空
    dwr.util.setValue("addKindName" , "");
    dwr.util.setValue("addKindDesc" , "");
    //隐藏添加种类的输入框
    $("addKind").style.display = "none";
}
```

如果用户单击“取消”按钮，对应的 JavaScript 函数将清空页面的种类名和种类描述。如果用户单击“添加”按钮，将触发 addCatalog 函数，其代码如下：

程序清单：codes\17\auction\am.js

```
function addCatalog()
{
    //获取种类名
    var addKindName = trim($("addKindName").value);
    //获取种类描述
    var addKindDesc = trim($("addKindDesc").value);
    var errStr = "";
    //进行客户端输入校验
    if (addKindName == null || addKindName == "")
    {
        errStr += "您没有输入种类名！\n";
    }
    else if (addKindDesc == null || addKindDesc == "")
    {
        errStr += "您没有输入种类描述！\n";
```

```
    }
    else if (addKindName.length >= 10 || addKindName.length < 2)
    {
        errStr += "种类名的长度必须在 2 到 10 之间! \n";
    }
    else if (addKindDesc.length <=8)
    {
        errStr += "种类描述的长度必须大于 8 个字符! \n";
    }
    //如果不能通过客户端输入校验
    if (errStr != "")
    {
        alert(errStr);
        return false;
    }
    //如果可以通过客户端输入校验
    else
    {
        //调用 addKind 业务逻辑方法来处理添加种类
        am.addKind(addKindName , addKindDesc , addKindCb);
    }
}
```

当用户单击了“添加”按钮后，JavaScript 先对用户的输入进行基本校验，判断用户输入是否满足要求。如果用户输入满足要求，则调用服务器端的 addKind()业务逻辑方法来添加种类。其中 addKindCb 是回调函数，该回调函数根据方法返回值判断添加种类是否成功，代码如下：

程序清单：codes\17\auction\am.js

```
//添加种类的回调函数
function addKindCb(data)
{
    //方法返回值是新增种类的主键，如果主键大于 0，表明添加种类成功
    if (data != null && typeof data == "number" && data > 0)
    {
        alert("添加种类成功");
        //取消添加
        cancelCatalog();
        //重新查看系统中所有的物品种类
        viewCatalog();
    }
    else
    {
        alert("添加种类失败");
    }
}
```

当用户添加种类成功后，系统将弹出提示框，告诉用户添加种类成功，并将添加种类的输入框隐藏，然后重新查看系统种类。这样使得添加种类成功后，可以立即在页面上看到新增的物品种类。

➤➤17.6.7 查看竞得物品

查看竞得物品需要用户登录，权限检查部分已经交给 Spring 的 AOP 来完成，JavaScript 代码部分无须太多额外的处理。

当用户单击页面上方的“查看竞得的物品”链接时，如果用户还未登录系统，将弹出提示信息，告诉用户应先登录系统。如果用户已经登录了本系统，则将触发如下函数：

程序清单：codes\17\auction\am.js

```
//浏览赢取的物品
function viewSuccess()
{
    //调用 getItemByWiner 业务逻辑方法获取当前用户赢取的物品
    am.getItemByWiner(viewSuccessCb);
}
```

上面的 JavaScript 函数调用了 getItemByWiner()业务逻辑方法来获取用户赢取的全部物品，其中viewSuccessCb()函数是回调函数，该函数负责将当前用户赢取的物品在页面中显示出来，其代码如下：

程序清单：codes\17\auction\am.js

```
//浏览当前用户赢取物品的回调函数
function viewSuccessCb(data)
{
    //如果调用 getItemByWiner 方法返回值不为空，且获取了至少一个物品
    if (data != null && data.length > 0)
    {
        //表格函数数组，每个函数对应一列
        var cellfuncs = [
            //输出物品名
            function(data){
                return data['name'];
            },
            //输出物品种类
            function(data){
                return data['kind'];
            },
            //输出物品的赢取价格
            function(data){
                return data['maxPrice'];
            },
            //输出物品的备注
            function(data){
                return data['pic'];
            }
        ];
        //删除 viewSuccBody 的所有表格行
        dwr.util.removeAllRows("viewSuccBody");
        //将 data 数据添加为 viewSuccBody 的表格行
        dwr.util.addRows("viewSuccBody" , data , cellfuncs , option);
        //隐藏当前显示内容
        if (curShow != null)
        {
            curShow.style.display = "none";
        }
        //显示查看赢取物品的表格
        $("viewSucc").style.display = "";
        curShow = $("viewSucc");
    }
    else
    {
        alert("暂时没有赢取任何物品！");
        }
}
```

如果当前用户已经赢取了某个物品，则当用户单击“查看竞得的物品”链接时，将看到如图 17.19 所示的界面。

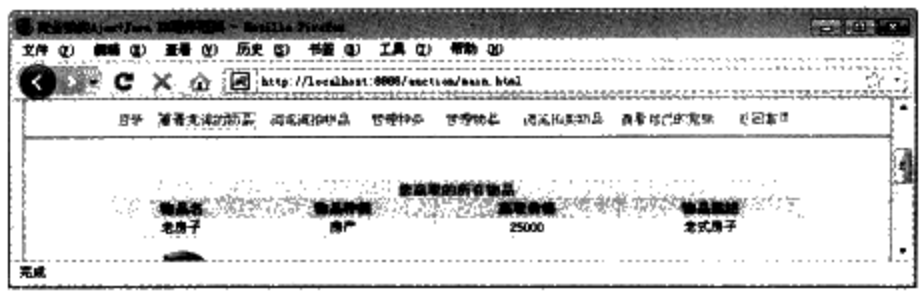


图 17.19　查看竞得的物品

➤➤17.6.8　查看自己的竞价记录

如果当前用户已经登录，则还可以查看到自己的所有竞价记录。用户单击“查看自己的竞标”链接将触发如下函数：

程序清单：codes\17\auction\am.js

```
//浏览自己的竞价
function viewBid()
{
    //调用 getBidByUser()业务逻辑方法来获取当前用户的竞价记录
    am.getBidByUser(viewBidCb);
}
```

业务逻辑组件的 getBidByUser()方法用于获取当前用户的所有竞价记录。如果用户没有登录，Spring AOP 框架中的拦截器会阻止对该方法的调用。如果用户已经登录，则可调用该业务逻辑方法。其中 viewBidCb 是回调函数，该函数负责将用户的所有竞价动态显示在页面中，其代码如下：

程序清单：codes\17\auction\am.js

```
//浏览当前用户全部竞价的回调函数
function viewBidCb(data)
{
    //如果返回值不为空，且至少获取了一次竞价记录
    if (data != null && data.length > 0)
    {
        //表格函数数组，每个函数对应一列
        var cellfuncs = [
            //输出竞价的物品
            function(data){
                return data['item'];
            },
            //输出竞价的价格
            function(data){
                return data['price'];
            },
            //输出竞价时间
            function(data){
                    return data['bidDate'].toLocaleString();
            },
            //输出竞价用户
            function(data){
                    return data['user'];
            }
        ];
        //删除 viewBidBody 表格中的所有表格行
        dwr.util.removeAllRows("viewBidBody");
        //将 data 中的数据添加为 viewBidBody 的表格行
        dwr.util.addRows("viewBidBody" , data , cellfuncs , option);
        //隐藏当前显示的内容
        if (curShow != null)
```

```
        {
            curShow.style.display = "none";
        }
        //显示查看竞价的表格
        $("viewBid").style.display = "";
        curShow = $("viewBid");
    }
    //如果没有获取对应的竞价记录
    else
    {
        alert("您暂时还没有任何竞价记录！");
    }
}
```

如果用户没有登录，则查看竞价记录时将被拦截器拦截，并弹出提示，告诉用户必须先登录系统。如果用户已经登录系统，则查看竞价记录时将看到如图 17.20 所示的界面。

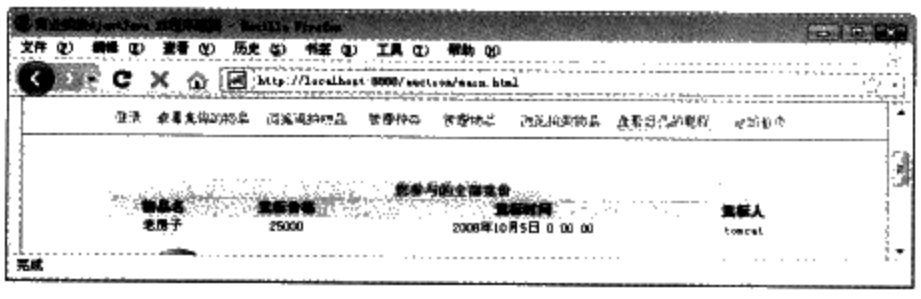


图 17.20　查看竞价记录

➤➤17.6.9　浏览拍卖物品

用户可以浏览当前正在拍卖的物品，浏览正在拍卖的物品时，必须先选择想浏览的物品种类。系统通过一个列表框来显示系统中所有的物品种类，当用户单击某个物品种类时，该种类下的所有物品将显示在页面上。当用户单击“浏览拍卖物品”链接时，将进入浏览物品的模块，应该先获取当前物品种类。单击该链接时将触发如下函数：

程序清单：codes\17\auction\am.js

```
//浏览拍卖中的物品
function viewInBid()
{
    //调用 getAllKind 业务逻辑方法获取所有的物品种类
    am.getAllKind(viewInBidCb);
}
```

其中的 viewInBidCb 函数是回调函数，负责将系统中的所有种类显示在下拉列表中，代码如下：

程序清单：codes\17\auction\am.js

```
//浏览拍卖中的物品的回调函数
function viewInBidCb(data)
{
    //如果当前物品种类不为空，且至少返回了一个物品种类
    if (data != null && data.length > 0)
    {
        //删除 selectBidCatalog 下拉列表中的所有选项
        dwr.util.removeAllOptions("selectBidCatalog");
        //删除拍卖物品表中的全部行
        dwr.util.removeAllRows("viewInBidBody");
        //将系统当前的所有物品种类添加在 selectBidCatalog 下拉列表中显示
        dwr.util.addOptions("selectBidCatalog", data, 'id', 'kindName');
        //隐藏当前显示的内容
        if (curShow != null)
```

```
        {
            curShow.style.display = "none";
        }
        //显示浏览拍卖物品的表格
        $("viewInBid").style.display = "";
        curShow = $("viewInBid");
    }
    else
    {
        alert("暂时没有任何物品");
        }
}
```

通过上面的代码可以看出，单击“浏览拍卖物品”链接时并未获取任何物品信息，而只是将当前的所有物品种类加载到了 selectBidCatalog 列表框中。如果用户单击“浏览拍卖物品”链接，将看到如图 17.21 所示的界面。

图 17.21　查看拍卖物品

正如在图 17.21 中所看到的效果，页面中并未显示任何物品，而仅仅在左边的列表框中显示了当前系统包含的所有物品种类。左边的列表框的代码如下：

```
<select id="selectBidCatalog" size="8" onchange="loadItemByCatalog(this.value);"></select>
```

通过上面的代码可以看出，当列表框的选中值发生改变时，将触发 loadItemByCatalog 函数，并将选中的物品种类 ID 作为参数传入 loadItemByCatalog 函数。其中 loadItemByCatalog 函数用于根据种类加载物品列表，其代码如下：

程序清单：codes\17\auction\am.js

```
//根据用户选择的种类加载物品
function loadItemByCatalog(cataId)
{
    //设置显示物品列表的标题
    dwr.util.setValue("bidCatalog" , dwr.util.getText("selectBidCatalog"));
    //删除物品表格中的所有表格行
    dwr.util.removeAllRows("viewInBidBody");
    //调用 getItemsByKind()业务逻辑方法
    am.getItemsByKind(cataId , loadItemByCatalogCb);
}
```

正如在上面的代码中所看到的，当单击列表框中任一种类时，将会把当前种类名加载到页面的物品列表的标题中，并调用服务器端的 getItemsByKind 方法，该方法根据种类 ID 获取所有物品。其中的 loadItemByCatalogCb 是回调函数，代码如下：

程序清单：codes\17\auction\am.js

```
//根据用户选择的种类加载物品的回调函数
function loadItemByCatalogCb(data)
{
```

```
//如果调用远程方法的返回值不为空，且获得了至少一个物品
if (data != null && data.length > 0)
{
    //遍历方法返回值中的每个对象
    for (var i = 0; i < data.length; i++ )
    {
        //创建一个表格行
        var tr = $("viewInBidBody").insertRow(i);
        //根据表格行的奇偶不同，设置表格行的背景色
        if(i % 2 == 0)
        {
                tr.style.backgroundColor = "#ffffff";
        }
        else
        {
            tr.style.backgroundColor = "#e2ffe2";
        }
        //插入多个单元格，将数据在表格中显示出来
        var td = tr.insertCell(0);
        td.innerHTML = '<a href="#" onClick="viewDetail('
            + data[i]['id'] +')">' + data[i]['name'] + '</a>';
        td = tr.insertCell(1);
        td.innerHTML = data[i]['addTime'].toLocaleString();
        td = tr.insertCell(2);
        td.innerHTML = data[i]['endTime'].toLocaleString();
        td = tr.insertCell(3);
        td.innerHTML = data[i]['maxPrice'];
        td = tr.insertCell(4);
        td.innerHTML = data[i]['owner'];
    }
}
else
{
    alert("该种类下暂时没有拍卖物品，请重新选择！");
}
}
```

如果用户单击的物品种类下没有对应的拍卖物品，系统将弹出提示框。如果用户单击的种类下有相应的拍卖物品，将看到如图 17.22 所示的界面。

图 17.22 查看某种类下的物品

正如在图 17.22 中所见到的：物品列表中的物品名是一个超级链接，单击该链接将显示对应物品的详细信息，并可对该物品竞价。

➤➤17.6.10 参与竞价

如果单击如图 17.22 所示中的物品名的链接，将触发 viewDetail()函数，并将该物品 ID 作为参数

传给该函数。viewDetail 函数的代码如下：

程序清单：codes\17\auction\am.js

```
//查看物品详细信息
function viewDetail(itemId)
{
    //调用 getItem 业务逻辑方法，根据物品 ID 获取该物品的详细信息。
    am.getItem(itemId , viewDetailCb);
}
```

一旦获取了物品的详细信息，viewDetailCb()回调函数就负责将物品的详细信息在页面中显示出来，其代码如下：

程序清单：codes\17\auction\am.js

```
//查看物品详细信息的回调函数
function viewDetailCb(data)
{
    //隐藏当前显示的内容
    if (curShow != null)
    {
        curShow.style.display = "none";
    }
    //curItem 是个全局变量，用于保存当前物品 ID
    curItem = data["id"];
    //将获取的物品信息在页面中显示出来
    dwr.util.setValue("detailName" , data["name"]);
    dwr.util.setValue("detailDesc" , data["desc"]);
    dwr.util.setValue("detailRemark" , data["pic"]);
    dwr.util.setValue("detailKind" , data["kind"]);
    dwr.util.setValue("detailOwner" , data["owner"]);
    dwr.util.setValue("detailInitPrice" , data["initPrice"]);
    dwr.util.setValue("detailMaxPrice" , data["maxPrice"]);
    dwr.util.setValue("detailStartTime" , data["addTime"].toLocaleString());
    dwr.util.setValue("detailEndTime" , data["endTime"].toLocaleString());
    //显示浏览物品详细信息的表格
    $("viewDetail").style.display = "";
    curShow = $("viewDetail");
}
```

上面的回调函数负责将单击物品的详细信息在页面中显示出来，并将当前物品 ID 设置为全局变量 curItem 的值。单击某个物品名时，将看到如图 17.23 所示的界面。

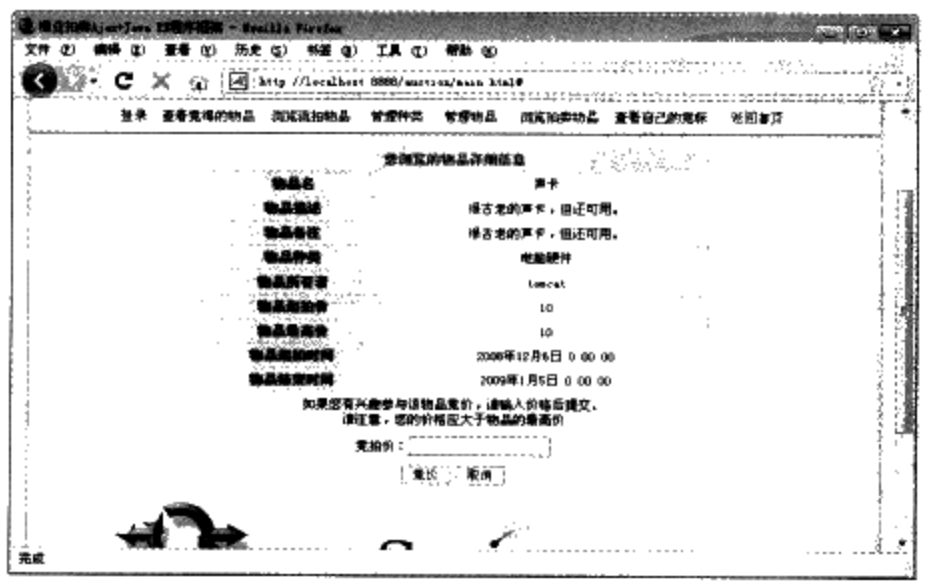


图 17.23　查看物品详细信息

在如图 17.23 所示的界面的下方，用户可以输入竞拍价格参与竞拍。如果用户单击“竞价”按钮，将触发如下函数。

程序清单：codes\17\auction\am.js

```
function submitBid()
{
    //获取竞拍价
    var bidPrice = trim(dwr.util.getValue("bidPrice"));
    //获取该物品的当前最高价
    var maxPrice = trim(dwr.util.getValue("detailMaxPrice"));
    var errStr = "";
    //进行客户端的输入校验
    if (bidPrice == null || bidPrice == "")
    {
        errStr += "您没有输入想竞价的价格！\n";
    }
    if (isNaN(bidPrice))
    {
        errStr += "竞价的价格必须是数字！\n";
    }
    //判断输入的竞拍价必须高于物品的当前最高价
    if (parseFloat(bidPrice) <= parseFloat(maxPrice))
    {
        errStr += "竞价的价格必须高于当前最高出价！\n";
    }
    //如果客户端输入校验失败，弹出出错提示
    if (errStr != "")
    {
        alert(errStr);
        return false;
    }
    //客户端输入校验通过
    else
    {
        if(curItem && curItem > 0)
        {
            //调用 addBid 业务逻辑方法处理添加竞价
            am.addBid(curItem , parseFloat(bidPrice) , submitBidCb);
        }
    }
}
```

submitBid 函数的处理过程非常简单，先进行必要的客户端输入校验，如果校验通过，则调用远程业务逻辑方法处理用户竞价。其中的 submitBidCb()是回调函数，该回调函数根据服务器响应确定用户竞价是否成功，代码如下：

```
//提交竞价回调函数
function submitBidCb(data)
{
    //如果方法返回值是数字，且大于 0，表明竞价成功
    if (data != null && typeof data == "number" && data > 0)
    {
        alert("竞价成功！");
        viewDetail(curItem);
        //清空竞价输入框
        dwr.util.setValue("bidPrice" , "");
    }
    else
```

```
        {
            alert("竞价失败！");
        }
    }
```

因为 addBid 业务逻辑方法的返回值是新增竞价的主键值，如果该方法的返回值为数字，且该数字大于 0，则表明添加竞价成功，系统提示竞价成功。

运行该系统，不难发现该系统还存在一个问题：用户刷新问题！每当浏览者单击浏览器的“刷新”按钮或单击 F5 键时，该应用的页面将会重新加载，用户之前操作的各种状态都将丢失——这是由 JavaScript 的本质决定的。为了更好地保存用户的操作状态，可以考虑将用户操作状态保存在 HttpSession 里——每次发送 Ajax 请求之时，将用户操作状态保存到 HttpSession 里，当用户刷新页面时，系统从 HttpSession 里读取操作状态，并将这些状态重新加载出来。

17.7 本章小结

本章介绍了一个完整的 Ajax+Java EE 项目，内容覆盖系统数据库设计，系统分析、设计，系统 DAO 层设计，业务逻辑层设计等。本章的应用是对传统 Java EE 应用的改进，而不是一个简单的 Ajax 应用。本应用还利用 Spring AOP 机制解决了 Ajax 应用的权限检查问题，应用将权限检查推迟到业务逻辑方法中进行，当权限检查失败时，业务逻辑层抛出一个 AuctionException 异常，该异常将由 DWR 框架传给浏览者。

此外，本章的应用还整合了 Spring 的邮件支持和任务调度，当用户竞价成功后，系统将向用户发送确认邮件；任务调度则周期性地检查系统中的拍卖物品，一旦发现某拍卖物品超过了拍卖的最后期限，系统会修改该物品的状态。

本章不再局限于单纯的 Ajax 知识的介绍，而是侧重于开发一个实际的应用，以及在实际应用中如何使用 Ajax：本系统将 DWR 和 Spring 无缝整合在一起，既充分利用了 Spring 容器的强大功能，也利用了 DWR 作为 Ajax 框架的便捷，为传统 Java EE 应用增加了 Ajax 支持，从而改善了用户体验。

➤➤本章练习

进一步完善本拍卖系统，读者可以考虑从如下几个方面完善：

- 增加刷新控制（用户发送请求时将客户端状态保存到 HttpSession 里，当用户刷新时重新加载 HttpSession 里的状态。）。
- 为系统增加用户注册功能，并允许每个用户单独开设网上商店。
- 增加物品图片，允许用户添加物品时上传多张物品图片。
- 为系统增加更细致的分类，以便用户更方便地查询商品。
- 为系统增加搜索功能。为商品列表增加分页。
- 控制用户竞价，禁止物品所有者对物品竞价。
- 用户赢取指定物品后，该用户可以对物品的所有者添加评价。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview®
www.broadview.com.cn

《疯狂 Ajax 讲义》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入地学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；

2. 本书作者的见面会信息或网络视频的沟通活动；

3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．短信

您只需编写如下短信： B08440+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2．电子邮件

您可以发邮件至jsj@phei.com.cn或editor@broadview.com.cn。

3．信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B08440+退订”至10666666789即可，或者编写邮件“B08440+退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn亦可取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

我们期待您的消息！
博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888
E-mail：jsj@phei.com.cn，editor@broadview.com.cn

反侵权盗版声明

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

疯狂源自梦想 技术成就辉煌

看得懂 学得会 做得出

不知道是否有人仔细研究过笔者写的书，书中的长句是很少的——因为很多语句自己都会反复地调整，有兴趣的读者可以仔细体会一下这个特点。此外，本书还有如下特点：

1. 通俗易懂，适合自学

该书第一版作为培训教材近2年了，在吸收大量学习者的学习体会和心得的基础上，本书重点讲解了学习过程中难以理解和掌握的知识点，降低了学习者的学习难度。

2. 知识丰富，内容全面

正如该书的第一版，书中知识非常全面：XHTML、CSS、JavaScript、DOM、Event机制、XMLHttpRequest、Prototype库、jQuery、DWR、AjaxTags等Ajax知识的相关内容，都可在本书中找到详细的讲解。

3. 深入实用，实践性强

本书并不是一本Ajax的入门图书，本书将Ajax技术融入轻量级Java EE开发，深入介绍了Ajax+ Java EE整合开发的方法和步骤，对实际企业开发具有极好的指导意义。

阅读此书有任何技术问题，都可以登录如下站点获得解决：
疯狂Java联盟：http://www.crazyjava.org

上架建议：Java>Ajax

网上订购：www.dearbook.com.cn
第二书店·第一服务

责任编辑：朱沭红
责任美编：李 玲

ISBN 978-7-121-08440-9
9 787121 084409 >

定价：69.00元（含光盘1张）