



(美)Marty Hall, Larry Brown著 赵学良 译

Servlet与JSP核心编程 (第2版)

Core Servlets and JavaServer Pages:
Volume 1: Core Technologies
2nd Edition



Java 技术丛书

Servlet 与 JSP 核心编程

(第 2 版)

(美) Marty Hall
Larry Brown 著
赵学良 译

内 容 简 介

本书由浅入深，全面而深入地介绍了 servlet 和 JSP 技术。本书重点介绍核心技术，同时对相关的内容，如 Web 服务器的配置、安装和应用，数据库的安装和配置等都做了详细明了的介绍。本书的例子简练但真实，将复杂的任务拆分成多个步骤逐一介绍，大大减轻了读者阅读的负担。另外，和其他同类书籍不同的是，本书引导读者根据实际需要取长补短，同时，还基于实际的应用给出大量的提示。

本书叙述详尽、条理清晰。对于初学者来说是一本不可多得的入门书籍，经验丰富的 servlet 和 JSP 开发人员也可以通过阅读本书得到巩固和提高。

Simplified Chinese edition copyright © 2004 by PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.

Original English language title from Proprietor's edition of the Work.

Original English language title: Core Servlets and JavaServer Pages: Volume 1: Core Technologies, 2nd Edition by Marty Hall, Larry Brown, Copyright © 2004

EISBN: 0-13-009229-0

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Sun Microsystems, Inc.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).

本书中文简体翻译版由 Pearson Education 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字：01-2003-6859

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

Servlet 与 JSP 核心编程(第 2 版) / (美) 霍尔(Hall, M.), (美) 布朗(Brown, L.) 著; 赵学良译.
—北京: 清华大学出版社, 2004

(Java 技术丛书)

书名原文: Core Servlets and JavaServer Pages: Volume 1: Core Technologies, 2nd Edition

ISBN 7-302-08627-3

I. S… II. ①霍… ②布… ③赵… III. ① JAVA 语言—程序设计 ②JAVA 语言—主页制作—程序设计
IV. ①TP312 ②TP393.092

中国版本图书馆 CIP 数据核字(2004)第 043775 号

出 版 者: 清华大学出版社 地 址: 北京清华大学学研大厦

http://www.tup.com.cn 邮 编: 100084

社 总 机: 010-62770175 客户服务: 010-62776969

文稿编辑: 汤涌涛

封面设计: 陈刘源

印 刷 者: 北京牛山世兴印刷厂

装 订 者: 北京国马印刷厂

发 行 者: 新华书店总店北京发行所

开 本: 185×260 印 张: 31 字 数: 754 千字

版 次: 2004 年 6 月第 1 版 2004 年 6 月第 1 次印刷

书 号: ISBN 7-302-08627-3/TP·6184

印 数: 1~3500

定 价: 59.00 元

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话: (010)62770175-3103 或 (010)62795704。

译者序

以前在写译者序时，我总是会写很多，包括我的感想、阅读本书的体会与经验，以及本书的优点和特点，等等。但这一次不同了，首先，它的前身已经得到读者的广泛认同，作为计算机书籍，销量达到10万册绝对可以算是十分畅销了。故而，不用担心由于读者对它不了解而错过这本优秀的图书。如果再花时间去介绍它的优点与特点，那就有画蛇添足之嫌，而且，由于读者甚众，长篇大论的译者序会浪费读者的时间，积少成多，聚沙成塔，累积起来这个时间可能会比较可观。俗话说“一寸光阴一寸金”（可惜呀！有价无市），浪费读者的时间可谓是罪莫大焉。

再回到翻译上来，制作中译本就是为了节省读者学习的时间，降低读者学习的难度，减轻读者的负担（包括经济上），而通畅、准确的译文更是能让学习达到事半功倍的效果。但这个良好氛围的建立是需要多方共同努力的。作为出版社，应该及时、有选择、高效率、高质量地引进读者最需要的图书，清华大学出版社在这一方面的成就有目共睹。作为译者，应该准确、负责、高效地完成书籍的翻译工作，虽然不一定要殚精竭虑，但尽心尽力，以为读者负责的心态努力做到自己的最好却是份内的事。而作为读者，并不是被动地选择和接受，事实上，只有读者才是推动图书出版工作的主要和惟一的动力。读者的反馈对于确定图书的种类、提高图书的质量有着极为重要的作用，公正而客观的评价对于提高图书的质量是必不可少的。所以希望读者能够到本书的信息网站上（见封底）多发表自己的观点和意见，读者、译者和编审人员的良好互动才能创建良好的出版环境。

参与本书翻译的还有王永、张阳，他们分别在计算机应用和开发领域拥有丰富的经验。他们的参与对本书大有裨益。

感谢我妈妈，她对我无微不至的照顾使我能够将更多的时间投入到本书的修饰和润色中。当然还要感谢清华大学出版社为我们及时、高效、准确地做好图书的引进和出版工作。本书的编辑汤涌涛女士为本书投入了大量的时间，她丰富的经验及敬业的精神对本书的完善起了至关重要的作用。

赵学良

序

设想您的公司希望在线销售产品。您有一个数据库，其中存储着每项商品的价格及库存状况。但是，您的数据库不能处理 HTTP——Web 浏览器使用的语言，它也不能输出 HTML——Web 浏览器需要的格式。您该怎么办？用户决定购买什么之后，您如何收集这些信息呢？如果您想依据访问者的偏好与兴趣定制您的网站，应该从何处入手呢？如果您希望在用户购物时跟踪用户选购的物品清单，要用什么技术来实现呢？如果您的网站越来越受欢迎，您可能希望对页面进行压缩，以降低对带宽的占用。怎样完成这项工作才能使近 30% 的访客(其浏览器不支持压缩)依旧能够访问您的网站呢？所有这些情况中，您都需要某种程序作为浏览器和服务器端资源的媒介。本书介绍如何使用 Java 平台完成这项工作。

“等一下，”您可能会问，“你不是已经编写了一本介绍这些内容的书吗？”是的。在 2000 年 5 月，Sun Microsystems 和 Prentice Hall 出版了 Marty 的第二版 *Core Servlets and JavaServer Pages*。这本书的成功远远超出了每个人的预期，总销量近 10 万册，并被翻译成保加利亚语、简体中文、繁体中文、捷克语、法语、德语、以色列语(希伯来语)、日语、韩语、波兰语、俄语和西班牙语，并进入亚马逊 2001 年计算机编程书籍排名的前五名。而且，大量的培训请求令 Marty 应接不暇，他对向行业中的开发人员讲授培训课程亦是乐此不疲。尽管对大部分请求都不得不婉言谢绝，他还是在澳大利亚、加拿大、日本、波多黎各、菲律宾和美国众多的地点开设了 servlet 和 JSP 的短期课程。

此后，servlet 和 JSP 的应用持续高速增长。Java 2 平台已经成为开发电子商务应用、动态 Web 站点，以及利用 Web 的应用和服务的首选技术。servlet 和 JSP 依旧是这个平台的基础——它们提供 Web 客户程序与服务器端应用之间的链接。几乎所有主要的 Web 服务器，无论它们运行在 Windows、Unix(包括 Linux)、MacOS、VMS 之上，还是大型机操作系统之上，都支持 servlet 和 JSP 技术(自身支持或通过插件)。通过简单的配置，就可以在 Microsoft IIS，Apache Web Server，IBM WebSphere，BEA WebLogic，Oracle9i AS 以及其他众多服务器上运行 servlet 和 JSP。商业或开放源码的 servlet 和 JSP 的性能已经得到显著的提高。

然而，这个领域依旧处于不断的演进过程中。例如：

- Sun 已经不再提供 servlet 和 JSP 的官方参考实现。取而代之的是 Apache Tomcat，这是一个开放源码的产品，由许多不同组织形成的一个小组进行开发。因此，我们在本书中详细介绍 Tomcat 的配置与使用。
- 除 Tomcat 以外，本书首次面市之前十分流行的服务器，现在已逐渐退出了人们的视野。因此，我们不再介绍它们，而是选择 Macromedia JRun 和 Caucho Resin 进行介绍。

- servlet 规范 2.4 版在 2003 年晚期发布。在这份规范中加入并更改了许多 API。因此，我们对本书进行了升级，使得本书介绍的内容与这些 API 协调一致。
- JSP 规范 2.0 版也已发布(同样是在 2003 年晚期)。这个版本使得我们可以使用简短的表达式语言(expression language)访问 bean 的属性和集合中的元素。因此，本书中，我们既介绍传统的脚本，也介绍 JSP 2.0 表达式语言的应用。
- JDBC 的两个新版本业已发布，它们提供许多有用的新特性。因此，我们将在这些新特性的上下文中介绍数据库访问。
- MySQL 事实上已经成为流行的免费数据库。因此，我们在本书中将介绍如何下载、配置和使用 MySQL(当然，我们还介绍 Oracle9i 和 Microsoft Access)。

专注于服务器端技术的 Java 社团行动频繁。为了反映出这些最新的行动，本书已经彻头彻尾地进行了重写。许多新的能力已经加入进来。经验丰富的开发人员 Larry Brown 参与到本书的编写工作中，同时带来了他的专业经验，尤其在数据库应用领域。同时，本书对许多难学的课程进行了详细的解释。许多技术如今已经大不相同。

新版本对 servlet 和 JSP 进行了完全且及时的介绍。我们希望读者能够从中受益。

谁应该阅读本书

本书瞄准两个主要的读者群体。

主要的读者是对 Java 编程语言有基本的了解，但对服务器端应用了解甚少，甚至完全一无所知的开发人员。对于这些读者而言，几乎本书的全部内容都很有用；惟一的例外是 JSP 2.0 表达式语言(如果读者使用的服务器只兼容 JSP 1.2，则不适用)，在任何现实的应用中，几乎都会用到本书中每个章节介绍的内容。

第二个群体是那些熟悉基本的 servlet 和 JSP 开发，并希望学习如何使用我们刚才介绍的新功能的读者。如果您属于这种读者，那么您可以跳过本书的许多章节，集中学习 servlet 2.4，JSP 2.0 或 JDBC 3.0 提供的新功能。

尽管对于有经验的 servlet 和 JSP 程序员，还有这个领域的初学者，这本书都很适用，但我们还是假定读者对 Java 编程有基本的了解。您不必是专业的 Java 开发人员，但如果他对 Java 编程语言一无所知，那么这本书就不太适合您。毕竟，servlet 和 JSP 是 Java 编程语言的一种应用。如果您不知道这种语言，就没有办法应用它。因此，如果您对基本的 Java 开发没有任何概念，那么请从诸如 *Thinking in Java*, *Core Java*, 或 *Core Web Programming* 等书籍开始学起。在掌握了基本的知识之后，再回来阅读本书。

第二卷

本书的第一卷集中在核心技术上：介绍那些可能应用到每个现实项目中的 servlet 和 JSP 功能。第二卷着重介绍高级功能：不常使用但对复杂应用颇具价值的特性。

这些主题包括 servlet 和 JSP 过滤器、声明或规划 Web 应用的安全性、定制标签库、JSP 标准标签库(JSTL)、Apache Struts、JavaServer Faces(JSF)、Java Architecture for XML Binding(JAXB)、数据库连接共享(pooling)、高级 JDBC 特性，以及 Ant 在部署过程中的应用。

有关第二卷的出版日期，可以到本书的网站 <http://www.coreservlets.com/> 查询。

本书的特色

本书有 5 项重要的特征，使它有别于许多其他类似书籍。

- **综合介绍 servlet 和 JSP。** 这两项技术密切相关；应该同时学习和应用这两种技术。
- **真实的代码。** 完整的、能够工作且文档详尽的程序是学习的根本；本书中我们提供了许多这类程序。
- **逐步骤的指示。** 本书用真实的例子，将复杂的任务拆分成多个简单的步骤，加以阐述。
- **服务器的配置和使用细节。** 我们提供了大量具体的例子，帮助读者快速掌握所讲述的内容。
- **设计策略。** 我们基于最佳的方案和实践，给出了大量基于实践的提示。

综合介绍 servlet 和 JSP

《Servlet 与 JSP 核心编程》的主要观点之一是，要同时学习(和使用！)servlet 和 JSP，不要将二者分离开来。毕竟，它们不是两项完全不同的技术：JSP 不过是编写 servlet 的另一种不同的方式而已。如果您不了解 servlet 编程，那么，即使存在比 JSP 更好的选择，您也不能使用 servlet，不能使用 MVC 构架来集成 servlet 和 JSP，不能理解复杂的 JSP 构造，同时您也不能理解 JSP 脚本(scripting)元素如何工作(由于它们实际上就是 servlet 代码)。如果您不理解 JSP 开发，那么当 JSP 比 servlet 技术更为适用时，您就不能使用 JSP，不能使用 MVC 构架，同时，即使几乎完全是静态 HTML 的页面，您依旧要借助 print 语句。

servlet 和 JSP 是一体的！请同时学习这二者。

真实的代码

确实，小段的代码对于介绍概念来说很有用。本书含有大量这类代码。但是，要真正理解如何使用各种技术，您还需要在完整的可工作程序的上下文中检查这些技术。程序并不需要很大，只要是完整且能够运行的程序就可以。我们提供大量此类程序，这些程序都有较好的文档，并且可以无限制地使用，请到 <http://www.coreservlets.com> 获得相关内容。

逐步骤的指示

在 Marty 还是计算机科学研究生的时候(远在 Java 出现以前)，一位讲授算法的教授在课堂上当众宣布他是一个逐步骤指示的信徒。Marty 为此感到迷惑不解：其他人不这样吗？一点也不。确实，大多数讲师都会以这种方式解释简单的任务，但这位教授把它上升到理论的高度，并说：“首先做这个，然后做那个”，依次类推。其他的讲师不用这种方式解释事物；教科书也不以这种方式讲授课程。但是，这种方法对 Marty 帮助很大。

如果这种方法甚至能够用于理论性学科，那么它也应该很适合于本书介绍的任务。

服务器的配置和使用细节

当 Marty 初学服务器端编程时，他收集了许多书籍、正式规范和一些在线文档。它们几乎无一例外地说：“由于这项技术是可移植的，所以，您需要阅读服务器的相关文档，了解如何执行 servlet 或 JSP 页面”。啊哈！他甚至根本无从着手。在徘徊良久之后，他下载了一个服务器。他编写了一些代码，但怎么编译这些代码呢？编译后又应该放在什么地方呢？如何调用这些代码呢？在什么地方能找到相关的帮助呢？

servlet 和 JSP 代码是可移植的。API 是标准化的。但服务器的结构和组织却不是标准化的。Tomcat 中代码应该存放的目录就不同于 JRun 中的存放目录。Resin 中 Web 应用的设置就不同于其他服务器。这些细节很重要。

在此，我们并不是说这本书专门针对某种特定的服务器。我们只是想告诉读者，当某个主题需要与服务器相关的知识时，本书会针对特定的服务器进行论述。此外，具体的例子很有帮助。因此，当我们介绍的主题需要与服务器相关的信息时，比如存储 Web 应用的目录，我们会首先说明服务器遵循的通用模式。然后，我们给出 3 种最为流行且可以免费用于桌面开发的服务器的详尽细节，它们是 Apache Tomcat、Macromedia JRun 和 Caucho Resin。

设计策略

当然，了解 API 提供的能力很有价值。同时，语法细节也确实很重要。但是，您还是需要了解总体的概况。什么时候特定的方案是最好的呢？为什么？应该注意什么？servlet

和 JSP 技术并非完美；怎样设计系统才能最大限度地发挥它们的长处，避免它们的缺点？什么策略才能简化项目的长期维护？应该避免什么方案？

我们对 servlet 和 JSP 技术并不陌生。我们已经使用它们好多年了。并且，我们已经从数百个读者和 Marty 培训课程的学生那里得到了反馈。因此，我们不是仅仅向您展示如何使用单独的特性，我们还解释了如何在系统的总体设计中使用这些特性，并给出了最佳做法和策略。

本书的组织方式

本书由 3 部分组成：servlet 技术、JSP 技术和相关的支持技术。

第一部分：servlet 技术

- 下载和配置免费的服务器
- 建立开发环境
- 部署 servlet 和 JSP 页面：一些选项
- Web 应用中项目的组织
- 构建基本的 servlet
- 了解 servlet 的生命周期
- 应对多线程问题
- servlet 和 JSP 页面的调试
- 表单参数的读取
- 处理缺失和异常的数据
- 应对不完全的表单提交
- 使用 HTTP 请求报头
- 压缩页面
- 根据浏览器的类型以及用户如何到达当前页面对页面进行定制
- 操纵 HTTP 状态代码和响应报头
- 请求的重定向
- 用 servlet 构建 Excel 表格
- 由 servlet 生成定制 JPEG 图像
- 向用户发送增量更新
- 处理 cookie
- 记录用户的偏好
- 跟踪会话
- 浏览器会话和服务器会话之间的不同

- 累积用户的所购商品
- 实现购物车

第二部分：JSP 技术

- 了解对 JSP 技术的需求
- 评估从 JSP 页面中调用 Java 代码的策略
- 用传统的 JSP 脚本元素调用 Java 代码
- 使用预定义 JSP 变量(隐式对象)
- 用 page 指令控制代码结构
- 在 JSP 页面中生成 Excel 表格
- 控制多线程行为
- 在请求期间包括页面
- 在编译期间包括页面
- 使用 JavaBean 组件
- 自动设定 bean 的属性
- 共享 bean
- 用 MVC 构架集成 servlet 和 JSP 页面
- 使用 RequestDispatcher
- MVC 数据共享选项的比较
- 使用 JSP 2.0 表达式语言(expression language)访问 bean
- 使用统一的语法访问数组元素、List 项和 Map 条目
- 使用表达式语言的运算符

第三部分：支持技术

- 用 JDBC 访问数据库
- 简化 JDBC 的使用
- 使用预编译(参数化)查询
- 执行存储过程
- 事务的控制
- 使用 JDO 和其他对象-关系映射
- 配置 Oracle, MySQL 和 Microsoft Access 对 JDBC 的支持
- 创建 HTML 表单
- 审视所有合法的 HTML 表单元素
- 用定制 Web 服务器调试表单

核心方法

尤其要注意“核心方法”中的内容。它们给出那些必须使用或几乎总是使用的技术。

本书的网站

网站 <http://www.coreservlets.com> 为本书提供支持。这个免费的网站提供：

- 本书中使用的所有源代码；这些代码可以免费下载，并可以自由地使用它们。
- 本书正文中提到的所有 URL 链接。
- servlet 和 JSP 软件的最新下载站点。
- 书籍的打折信息。

另外，还提供本书的附加内容、更新和新闻。

目 录

第 1 章 Servlet 和 JSP 技术概述	1
1.1 servlet 的功用	1
1.2 要动态构建网页的原因	2
1.3 servlet 代码初探	3
1.4 Servlet 相对于“传统” CGI 的优点	4
1.5 JSP 的作用	6
第 I 部分 Servlet 技术	
第 2 章 服务器的安装和配置	11
2.1 下载和安装 Java 软件开发工具包	12
2.2 为桌面计算机下载服务器	13
2.3 服务器的配置	15
2.4 配置 Apache Tomcat	15
2.5 配置 Macromedia JRun	20
2.6 配置 Cauchy Resin	24
2.7 建立开发环境	25
2.8 测试系统的设置	28
2.9 实现简化的部署方法	35
2.10 默认 Web 应用的部署目录：汇总	37
2.11 Web 应用：预览	40
第 3 章 servlet 基础	48
3.1 servlet 的基本结构	49
3.2 生成纯文本的 servlet	50
3.3 生成 HTML 的 servlet	51
3.4 servlet 的打包	53
3.5 简单的 HTML 构建工具	54
3.6 servlet 的生命周期	56
3.7 SingleThreadModel 接口	62
3.8 servlet 的调试	65

第 4 章 客户请求的处理：表单数据	68
4.1 表单数据的作用	68
4.2 在 servlet 中读取表单数据	69
4.3 示例：读取 3 个参数	72
4.4 示例：读取所有参数	74
4.5 参数缺失或异常时默认值的应用	77
4.6 过滤字符串中的 HTML 特殊字符	85
4.7 根据请求参数自动填充 Java 对象：表单 bean	90
4.8 当参数缺失或异常时重新显示输入表单	96
第 5 章 客户请求的处理：HTTP 请求报头	104
5.1 请求报头的读取	104
5.2 制作所有请求报头的表格	106
5.3 了解 HTTP 1.1 请求报头	108
5.4 发送压缩 Web 页面	111
5.5 区分不同的浏览器类型	114
5.6 依据客户的到达方式定制页面	116
5.7 标准 CGI 变量的访问	119
第 6 章 服务器响应的生成：HTTP 状态代码	124
6.1 状态代码的指定	125
6.2 HTTP 1.1 状态代码	126
6.3 将用户重定向到浏览器相关页面的 servlet	130
6.4 各种搜索引擎的一个前端	132
第 7 章 服务器响应的生成：HTTP 响应报头	138
7.1 在 servlet 中设置响应报头	138
7.2 理解 HTTP 1.1 响应报头	139
7.3 构建 Excel 电子表格	144
7.4 servlet 状态的持续以及页面的自动重载	145
7.5 使用 servlet 生成 JPEG 图像	154
第 8 章 cookie 管理	162
8.1 cookie 的优点	162
8.2 cookie 存在的一些问题	164
8.3 cookie 的删除	166

8.4 cookie 的发送和接收	166
8.5 使用 cookie 检测初访者	169
8.6 使用 cookie 属性	171
8.7 区分会话 cookie 与持续性 cookie	173
8.8 基本的 cookie 实用程序	175
8.9 实际使用 cookie 实用程序	177
8.10 修改 cookie 的值：记录用户的访问计数	179
8.11 使用 cookie 记录用户的偏好	180
第 9 章 会话跟踪	185
9.1 会话跟踪的需求	185
9.2 会话跟踪基础	187
9.3 会话跟踪 API	189
9.4 浏览器会话与服务器会话	191
9.5 对发往客户的 URL 进行编码	192
9.6 显示客户访问计数的 servlet	193
9.7 累计用户数据的列表	195
9.8 拥有购物车和会话跟踪功能的在线商店	198
第 II 部分 JSP 技术	
第 10 章 JSP 技术概述	215
10.1 对 JSP 的需求	216
10.2 JSP 的好处	216
10.3 JSP 相对于竞争技术的优势	217
10.4 对 JSP 的误解	219
10.5 JSP 页面的安装	221
10.6 基本语法	223
第 11 章 用 JSP 脚本元素调用 Java 代码	226
11.1 模板文本的创建	226
11.2 在 JSP 中调用 Java 代码	226
11.3 限制 JSP 页面中 Java 代码的量	227
11.4 JSP 表达式的应用	230
11.5 示例：JSP 表达式	232
11.6 servlet 和 JSP 页面的对比	234
11.7 编写 scriptlet	235

11.8 scriptlet 示例.....	237
11.9 使用 scriptlet 将 JSP 页面的某些部分条件化	238
11.10 使用声明	240
11.11 声明的例子	241
11.12 使用预定义变量.....	243
11.13 JSP 表达式、scriptlet 和声明的比较	244
第 12 章 控制所生成的 servlet 的结构: JSP page 指令	249
12.1 import 属性.....	249
12.2 contentType 和 pageEncoding 属性	252
12.3 条件性地生成 Excel 电子表格.....	253
12.4 session 属性.....	255
12.5 isELIgnored 属性.....	256
12.6 buffer 和 autoFlush 属性	256
12.7 info 属性.....	257
12.8 errorPage 和 isErrorPage 属性	257
12.9 isThreadSafe 属性	259
12.10 extends 属性	260
12.11 language 属性	261
12.12 指令的 XML 语法.....	261
第 13 章 在 JSP 页面中包含文件和 applet.....	262
13.1 在请求期间包含页面: jsp:include 动作.....	262
13.2 在页面转换期间包含文件: include 指令	266
13.3 使用 jsp:forward 转发请求	270
13.4 包含使用 Java 插件的 applet.....	271
第 14 章 JavaBean 组件在 JSP 文档中的应用	279
14.1 使用 bean 的原因	279
14.2 bean 是什么	280
14.3 bean 的应用: 基本任务	281
14.4 示例: StringBean.....	284
14.5 设置 bean 的属性: 高级技术	286
14.6 共享 bean	291
14.7 共享 bean 的 4 种方式: 示例	294

第 15 章 servlet 和 JSP 的集成：模型-视图-控制器构架.....	303
15.1 MVC 的需求	303
15.2 用 RequestDispatcher 实现 MVC.....	305
15.3 MVC 代码汇总	309
15.4 目的页面中相对 URL 的解释	310
15.5 MVC 的应用：银行账户余额.....	311
15.6 3 种数据共享方式的对比.....	317
15.7 从 JSP 页面转发请求.....	322
15.8 包含页面	323
第 16 章 简化对 Java 代码的访问：JSP 2.0 表达式语言.....	325
16.1 应用 EL 的驱动力.....	325
16.2 表达式语言的调用.....	327
16.3 阻止表达式语言的求值.....	327
16.4 阻止标准脚本元素的使用.....	330
16.5 访问作用域变量.....	330
16.6 访问 bean 的属性.....	332
16.7 访问集合	337
16.8 引用隐式对象.....	339
16.9 表达式语言中运算符的应用.....	341
16.10 表达式的条件求值.....	344
16.11 表达式语言其他功能概览.....	346
第III部分 支持技术	
第 17 章 数据库访问：JDBC	351
17.1 JDBC 应用概述.....	352
17.2 基本 JDBC 示例.....	358
17.3 用 JDBC 实用工具简化数据库访问	364
17.4 使用预备语句.....	374
17.5 创建可调用语句.....	377
17.6 使用数据库事务.....	382
17.7 使用 ORM 框架将数据映射到对象.....	386
第 18 章 配置 MS Access, MySQL 和 Oracle9i.....	392
18.1 配置 Microsoft Access 与 JDBC 的使用	392

18.2 MySQL 的安装和配置.....	396
18.3 Oracle9i 数据库的安装和配置.....	398
18.4 通过 JDBC 连接来测试数据库.....	420
18.5 建立 music 表.....	426
第 19 章 HTML 表单的创建和处理	431
19.1 HTML 表单如何传输数据	432
19.2 FORM 元素	435
19.3 文本控件	439
19.4 按钮	443
19.5 复选框和单选按钮	447
19.6 组合框和列表框	450
19.7 文件上传控件	453
19.8 服务器端图像映射	455
19.9 隐藏域	458
19.10 控件组	458
19.11 制表次序	460
19.12 用于调试的 Web 服务器	460
附录 服务器的组织与结构	467

第 1 章 Servlet 和 JSP 技术概述

本章的主题：

- servlet 的作用
- 动态构建 Web 页面
- servlet 代码初探
- servlet 和其他技术的对比
- JSP 的作用

servlet 和 JSP 技术已经成为开发在线商店、交互式 Web 应用、以及其他动态网站的首选技术。原因是什么呢？本章首先对这项技术进行高层的概括，并给出它被广泛采用的一些原因，之后给出编程技术的具体细节。

1.1 servlet 的功用

servlet 是运行在 Web 服务器或应用服务器上的 Java 程序，它是一个中间层，负责连接来自 Web 浏览器或其他 HTTP 客户程序的请求和 HTTP 服务器上的数据库或应用程序。servlet 的工作是执行下面的任务，如图 1.1 所示。

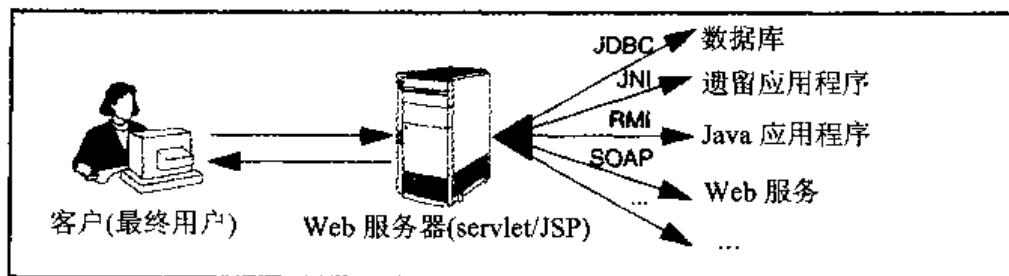


图 1.1 Web 中间件的作用

(1) 读取客户发送的显式数据。

最终用户一般在网页的 HTML 表单中输入这些数据。然而，数据还有可能来自于 applet 或定制的 HTTP 客户程序。servlet 如何读取这些数据在第 4 章中论述。

(2) 读取由浏览器发送的隐式请求数据。

图 1.1 中显示了一条从客户端到 Web 服务器(servlet 和 JSP 所在的层)的单箭头，但实际上从客户端传送到 Web 服务器的数据有两种，它们分别为用户在表单中输入的显式数据，以及后台的 HTTP 信息。两种数据都很重要。HTTP 信息包括 cookie、浏览器所能识别的媒体类型和压缩模式等；这部分内容在第 5 章详细论述。

(3) 生成结果。

这个过程可能需要访问数据库、执行 RMI 或 EJB 调用、调用 Web 服务，或者直接计算得出对应的响应。实际的数据可能存储在关系型数据库中。该数据库可能

不理解 HTTP，或者不能返回 HTML 形式的结果，所以 Web 浏览器不能直接与数据库进行会话。即使它能够做到这一点，为了安全上的考虑，我们也不希望让它这样做。对于大多数其他应用程序，也存在类似的问题。因此，我们需要 Web 中间层从 HTTP 流中提取输入数据，与应用程序会话，并将结果嵌入到文档中。

(4) 向客户发送显式数据(即文档)。

这个文档可以用各种格式发送，包括文本(HTML 或 XML)，二进制(GIF 图)，甚至可以是建立在其他底层格式之上的压缩格式，如 gzip。但是，到目前为止，HTML 是最常用的格式，故而 servlet 和 JSP 的重要任务之一就是将结果包装到 HTML 中。

(5) 发送隐式的 HTTP 响应数据。

图 1.1 中显示了一条从 Web 中间层(servlet 或 JSP 页面)到客户端的单箭头。但是，实际发送的数据有两种：文档本身，以及后台的 HTTP 信息。同样，两种数据对开发来说都是至关重要的。HTTP 响应数据的发送过程涉及告知浏览器或其他客户程序所返回文档的类型(例如 HTML)，设置 cookie 和缓存参数，以及其他类似的任务。这些任务在第 6 章和第 7 章详细论述。

1.2 动态构建网页的原因

当 Marty 编写完 *Core Servlets and JavaServer Pages* 的第一版时，许多对软件不甚了了的朋友和亲人询问他的书是讲什么的。Marty 向他们详细介绍 Java、面向对象的编程以及 HTTP 的技术细节，但他们很快就不知所云，一头雾水。最后，他们会不耐烦地问，“噢，这么说你这本书介绍的是如何制作网页，是这样吗？”

“不对！”，正确的答案是，“这本书介绍的是如何编写产生网页的程序。”

“哈！为什么要等到客户请求页面的时候，才让程序去构建结果呢？为什么不将网页提前做好呢？”

是的，预先建立的文档可以满足客户的许多请求，服务器无需调用 servlet 就可以处理这些请求。然而，许多情况下静态的结果不能满足要求，我们需要针对每个请求生成一个页面。实时构建网页的理由有很多种：

- 网页基于客户发送的数据。

例如，搜索引擎生成的页面，以及在线商店的订单确认页面，都要针对特定的用户请求而产生。在没有读取到用户提交的数据之前，我们不知道应该显示什么。要记住，用户提交两种类型的数据：显式(即 HTML 表单的数据)和隐式(即 HTTP 请求的报头)。两种输入都可用来构建输出页面。基于 cookie 值针对具体用户构建页面的情况尤其普遍。

- 网页由频繁改变的数据导出。

如果页面需要根据每个具体的请求做出相应的更改，当然需要在请求发生时构建响应。但是，如果页面周期性地改变，我们可以用两种方式来处理它：周期性地在服务器上构建新的网页(和客户请求无关)，或者仅仅在用户请求该页面时再构建。具体应该采用哪种方式要依具体情况而定，但后一种方式常常更为方便，因

为它只需简单地等待用户的请求。例如，天气预报或新闻网站可能会动态地构建页面，也有可能会返回之前构建的页面(如果它还是最新的话)。

- 网页中使用了来自公司数据库或其他服务器端数据源的信息。

如果数据存储在数据库中，那么，即使客户端使用动态 Web 内容，比如 applet，我们依旧需要执行服务器端处理。想像一下，如果一个搜索引擎网站完全使用 applet，那么用户将会看到：“正在下载 50TB 的 applet，请等待！”显然，这很愚蠢；这种情况下，我们需要与数据库进行会话。从客户端到 Web 层再到数据库(三层结构)，要比从 applet 直接到数据库(二层结构)更灵活，也更安全，而性能上的损失很少甚至没有。毕竟数据库调用通常是对速度影响最大的步骤，因而，经过 Web 服务器不会带来性能上的明显降低。实际上，三层结构常常更快，因为中间层(middle tier)可以执行高速缓存(caching)和连接共享(connection pooling)。

理论上讲，servlet 并非只用于处理 HTTP 请求的 Web 服务器或应用服务器，它同样可以用于其他类型的服务器。例如，servlet 能够嵌入到 FTP 或邮件服务器中，扩展它们的功能。而且，用于会话启动协议(Session Initiation Protocol, SIP)服务器的 servlet API 最近已经被标准化(参见 <http://jcp.org/en/jsr/detail?id=116>)。但在实践中，servlet 的这种用法尚不流行，在此，我们只论述 HTTP servlet。

1.3 servlet 代码初探

此时，尚不适合深入地钻研 servlet 的语法。但不要急，本书之后的内容将会对 servlet 进行详尽的介绍。现在，我们先快速地观察一个简单的 servlet，以对它的复杂程度有一个基本的了解。

清单 1.1 给出一个简单的 servlet，它向客户端输出一个简单的 HTML 页面。图 1.2 是它生成的结果。

第 3 章对这段代码做了详尽的说明，现在需要注意下面 4 点：

- 它是常规的 Java 代码。

这段代码用了新的 API，但不涉及新的语法。

- 它有我们不熟悉的重要语句。

servlet 和 JSP API 不属于 Java 2 平台标准版(Java 2 Platform, Standard Edition, J2SE)；它们是单独的规范(同时也属于 Java 2 平台企业版——J2EE)。

- 它对标准的类(HttpServletRequest)进行了扩展。

servlet 为应对 HTTP 提供了大量的基础结构。

- 它覆盖 override 了 doGet 方法。

servlet 用不同的方法响应不同类型的 HTTP 命令。

清单 1.1 HelloServlet.java

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

public class HelloServlet extends HttpServlet {
```

```

public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +"
        " \"Transitional//EN\">\n";
    out.println(docType +
        "<HTML>\n" +
        "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
        "<BODY BGCOLOR=\"#FDF5E6\">\n" +
        "<H1>Hello</H1>\n" +
        "</BODY></HTML>");
}
}

```

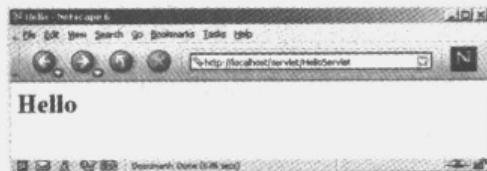


图 1.2 HelloServlet 的结果

1.4 Servlet 相对于“传统” CGI 的优点

和传统 CGI 及许多类 CGI 技术相比, Java servlet 效率更高、更易用、更强大、更容易移植、更安全、也更廉价。

1.4.1 效 率

应用传统的 CGI, 针对每个 HTTP 请求都要启动一个新的进程。如果 CGI 程序自身相对比较简单, 那么启动进程的开销会占用大部分执行时间。而使用 servlet, Java 虚拟机会一直运行, 并用轻量级的 Java 线程处理每个请求, 而非重量级的操作系统进程。类似地, 应用传统的 CGI 技术, 如果存在对同一 CGI 程序的 N 个请求, 那么 CGI 程序的代码会载入内存 N 次。同样的情况, 如果使用 servlet 则启动 N 个线程, 但仅仅载入 servlet 类的单一副本。这种方式减少了服务器的内存需求, 通过实例化更少的对象从而节省了时间。最后, 当 CGI 程序结束对请求的处理之后, 程序结束。这种方式难以缓存计算结果, 保持数据库连接打开, 或是执行依靠持续性数据(persistent data)的其他优化。然而, servlet 会一直停留在内存中(即使请求处理完毕), 因而可以直接存储客户请求之间的任意复杂数据。

1.4.2 便 利

servlet 提供大量的基础构造, 可以自动分析和解码 HTML 的表单数据, 读取和设置 HTTP 报头, 处理 cookie, 跟踪会话, 以及其他此类高级功用。而在 CGI 中, 大部分工作都需要我们自己来完成。另外, 如果您已经了解了 Java 编程语言, 为什么还要学习 Perl 呢? 您已经承认应用 Java 技术编写的代码要比 Visual Basic, VBScript 或 C++ 编写的代码更可

靠，且更易重用，为什么还要倒退回去选择那些语言来开发服务器端的程序呢？

1.4.3 强 大

servlet 支持常规 CGI 难以实现或根本不能实现的几项功能。servlet 能够直接与 Web 服务器对话，而常规的 CGI 程序做不到这一点，至少在不使用服务器专有 API 的情况下是这样。例如，与 Web 服务器的通信使得将相对 URL 转换成具体的路径名变得更为容易。多个 servlet 还可以共享数据，从而易于实现数据库连接共享和类似的资源共享优化。servlet 还能维护请求之间的信息，使得诸如会话跟踪和计算结果缓存等技术变得更为简单。

1.4.4 可 移 植 性

servlet 使用 Java 编程语言，并且遵循标准的 API。所有主要的 Web 服务器，实际上都直接或通过插件支持 servlet。因此，为 Macromedia JRun 编写的 servlet，可以不经任何修改地在 Apache Tomcat，Microsoft Internet Information Server(使用单独的插件)，IBM WebSphere，iPlanet Enterprise Server，Oracle9i AS，或者 StarNine WebStar 上运行。它们是 Java 2 平台企业版的一部分(J2EE，参见 <http://java.sun.com/j2ee/>)，所以，对 servlet 的支持越来越普遍。

1.4.5 廉 价

对于开发用的网站、低容量或中等容量网站的部署，有大量免费或极为廉价的 Web 服务器可供选择。因此，通过使用 servlet 和 JSP，我们可以从免费或廉价的服务器开始，在项目获得初步的成功后，再移植到具有更高性能或高级管理工具的昂贵的服务器上。这与其他 CGI 方案形成了鲜明的对比，这些 CGI 方案在初期都需要为购买专利软件包投入大量的资金。

价格和可移植性在某种程度上是互相关联的。例如，Marty 记录了所有通过电子邮件向他发送问题的读者的所在国。印度接近列表的顶端，可能仅次于美国。Marty 曾在马尼拉讲授过 JSP 和 servlet 培训课程，那儿也对 servlet 和 JSP 技术抱有很大的兴趣。

那么，为什么印度和菲律宾都对这项技术这么感兴趣呢？我们推测答案可能分为两部分。首先，这两个国家都拥有大量训练有素的软件开发人员。其次，这两个国家的货币对美元的汇率都极为不利。因此，从美国公司那里购买专用 Web 服务器会消耗掉项目的大部分前期资金。

但是，使用 servlet 和 JSP，他们能够从免费的服务器开始：Apache Tomcat(独立使用，或嵌入到常规的 Apache Web 服务器或 Microsoft IIS 中)。项目取得成功之后，他们可以转移到性能更高、管理更容易，但需要付费的服务器，如 Cauchy Resin。他们的 servlet 和 JSP 不需要重新编写。如果他们的项目变得更庞大，他们或许希望转移到分布式(集群)环境。没有问题：他们可以转而使用 Macromedia JRun Professional，该服务器支持分布式应用(Web 农场)。同样，他们的 servlet 和 JSP 没有任何部分需要重写。如果项目变得极为庞大，错综复杂，他们或许希望使用 Enterprise JavaBeans(EJB)来封装他们的商业逻辑。因此，他们可以切换到 BEA WebLogic 或 Oracle9i AS。同样，不需要对 servlet 和 JSP 做出更改。最后，

如果他们的项目变得更庞大，他们或许会将它从 Linux 转移到运行 IBM WebSphere 的 IBM 大型机上。他们还是不需要做出任何更改。

1.4.6 安全

传统 CGI 程序中主要的漏洞来源之一就是，CGI 程序常常由通用的操作系统外壳(shell)来执行。因此，CGI 程序必须仔细地过滤掉那些可能被外壳特殊处理的字符，如反引号和分号。实现这项预防措施的难度可能超出我们的想像，在广泛应用的 CGI 库中，不断发现由这类问题引发的弱点。

问题的第二个来源是，一些 CGI 程序用不自动检查数组和字符串边界的语言编写而成。例如，在 C 和 C++ 中，可以分配一个 100 个元素的数组，然后向第 999 个“元素”写入数据——实际上是程序内存的随机部分，这完全合法。因而，如果程序员忘记执行这项检查，就会将系统暴露在蓄意或偶然的缓冲区溢出攻击之下。

servlet 不存在这些问题。即使 servlet 执行系统调用(例如使用 `Runtime.exec` 或 `JNI`)激活本地操作系统上的程序，它也不会用到外壳来完成这项任务。当然，数组边界的检查以及其他内存保护特性是 Java 编程语言的核心部分。

1.4.7 主流

虽然存在许多很好的技术，但是，如果提供商不支持它们，或开发人员不知道如何使用这些技术，那么它们的优点又如何体现呢？servlet 和 JSP 技术得到服务器提供商的广泛支持，包括 Apache, Oracle, IBM, Sybase, BEA, Macromedia, Caucho, Sun/iPlanet, New Atlanta, ATG, Fujitsu, Lutris, Silverstream, World Wide Web Consortium(W3C)，以及其他服务器。存在几种低廉的插件，通过应用这些插件，Microsoft IIS 和 Zeus 也同样支持 servlet 和 JSP 技术。它们运行在 Windows, Unix/Linux, MacOS, VMS 和 IBM 大型机操作系统之上。它们是 Java 编程语言的最流行应用。它们可能是开发中型或大型 Web 应用的最流行选择。它们用在航空业(用于联合航空公司和达美航空公司的大部分网站)、电子商务(ofoto.com)，在线银行(First USA Bank, Banco Popular de Puerto Rico)、Web 搜索引擎/门户(excite.com)、大型金融网站(American Century Investments)、以及成百上千您日常光顾的其他网站。

当然，仅仅是流行并不能证明技术的优越性。有很多反面的例子。但我们的立场是：服务器端 Java 并非一项新的、未经证实的技术。

1.5 JSP 的作用

为了简化起见，某种程度上，可以将 servlet 看作是含有 HTML 的 Java 程序；将 JSP 看作是含有 Java 代码的 HTML 页面。

例如，将之前列出的 servlet(清单 1.1)与下面的 JSP 页面进行比较(清单 1.2)，就会发现后者更像是普通的 HTML 页面。有趣的是，尽管二者存在明显的差异，但实际上二者是相同的。JSP 文档只不过是编写 servlet 的另一种方式。JSP 页面会被翻译成 servlet，servlet 会被编译，在请求期间运行的就是 servlet。

清单1.2 Store.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3D//DTD HTML 4.0 Transtional//EN">
<HTML>
<HEAD><TITLE>Welcome to our Store</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>Welcome to our Store</H1>
<SMALL>Welcome,
<!--User name is "New User" for first-time visitors -->
<%= coreservlets.Utils.getUserNameFromCookie(request) %>
To access your account settings, click
<A HREF="Account-Settings.html">here.</A></SMALL>
<P>
Regular HTML for rest of online store's Web page
</BODY></HTML>
```

接下来的问题是，如果 JSP 技术和 servlet 技术从根本上讲功能相同，那么使用哪种技术还重要吗？答案是：“重要！”问题不在于功能，而在于方便性、易用性和可维护性。例如，所有 Java 编程语言能够完成的任务，都能够用汇编语言来完成。这是否意味着选择哪种语言没有什么差别呢？恐怕不是。

本书第 10 章开始详细论述 JSP。但现在最好先讲述一下 servlet 和 JSP 是如何结合在一起的。JSP 注重简化 HTML 的创建和维护。servlet 最适合于调用商业逻辑，执行复杂操作。一个简短的规则是：对于面向处理(processing)的任务，servlet 是最佳选择，而对于面向表示(presentation)的任务，JSP 是最佳选择。对于某些请求，servlet 是正确的选择。对于其他请求，JSP 则更好一些。而对于另外的请求，单独使用 servlet 和单独使用 JSP 都非最佳，将二者结合起来使用才是最好的方法(参见第 15 章)。重要的是，在整个项目中我们需要 servlet 和 JSP：几乎没有项目全部使用 servlet 或 JSP。我们需要将二者结合起来使用。

好！就先讲到这。请翻到下一章，继续学习！



第 I 部分：servlet 技术

- 第 2 章
服务器的安装和配置
- 第 3 章
servlet 基础
- 第 4 章
客户请求的处理：表单数据
- 第 5 章
客户请求的处理：HTTP 请求报头
- 第 6 章
服务器响应的生成：HTTP 状态代码
- 第 7 章
服务器响应的生成：HTTP 响应报头
- 第 8 章
cookie 管理
- 第 9 章
会话跟踪

第 2 章 服务器的安装和配置

本章的主题：

- 安装和配置 Java
- 下载并安装服务器
- 配置开发环境
- 测试安装是否正确
- servlet 和 JSP 部署工作的简化
- Tomcat, JRun 和 Resin 中文件的位置
- 将项目组织成 Web 应用

开始学习具体的 servlet 和 JSP 技术之前，首先需要获取恰当的软件，并了解如何使用这些软件。运行 servlet 和 JSP(JavaServer Page)需要一系列的相关软件，作为引导性的章节，本章将详细说明如何获取、配置、测试和使用所有软件的免费版本。开始的设置包括 7 步，概括如下：

(1) **下载并安装 Java 软件开发工具包(SDK)。**

这一步包括下载 Java 2 平台标准版(J2SE)，并相应地设置 PATH。2.1 节介绍这部分内容。

(2) **下载一个服务器。**

这一步涉及获取实现了 Servlet 2.3(JSP 1.2)或 Servlet 2.4(JSP 2.0)的服务器。2.2 节介绍这部分内容。

(3) **配置该服务器。**

这一步包括告诉服务器 SDK 的安装位置，将端口改为 80，并做出可能的、与服务器相关的定制。2.3 节中概括了一般的方式，2.4 节到 2.6 节分别给出针对 Apache Tomcat, Macromedia JRun 和 Cauchy Resin 的细节。

(4) **安装开发环境。**

这一步包括设置 CLASSPATH，使之包括开发目录的顶层目录，以及包含 servlet 和 JSP 类的 JAR 文件。2.7 节中介绍这部分内容。

(5) **测试设置是否正确。**

这一步包括检查服务器的主页，试验一些简单的 JSP 页面和 servlet。2.8 节中介绍这部分内容。

(6) **实现一种简单的部署方法。**

这一步需要选择从开发目录向服务器的部署区域复制资源的方案。2.9 节中介绍这部分内容。

(7) **创建定制的 Web 应用。**

这一步包括为应用程序创建单独的目录、修改 web.xml 以及为 servlet 赋予自定义的 URL。这一步可以推迟到对基本的 servlet 和 JSP 开发比较熟悉之后。2.11 节中介绍这部分内容。

2.1 下载和安装 Java 软件开发工具包

您可能早已安装了 Java 平台，如果尚未安装，那么第一步就是安装 Java 平台。当前版本的 servlet 和 JSP API 需要 Java 2 平台(标准版，J2SE，或企业版，J2EE)。如果不准备使用 J2EE 的特性，如 Enterprise JavaBeans(EJB)或 Java 消息服务(Java Messaging Service, JMS)，我们建议使用标准版。服务器会提供向 Java 2 标准版加入 servlet 和 JSP 支持所需的类。

但我们需要 Java 的哪个版本呢？这要依使用哪种 servlet/JSP API，以及使用的是与 J2EE 完全兼容的应用服务器(例如 WebSphere, WebLogic 或 JBoss)，还是独立的 servlet/JSP 容器(例如 Tomcat, JRun 或 Resin)而定。如果从零做起，我们推荐使用 Java 的最新版本(1.4)；这样可以得到最佳的性能，并保证与将来的版本兼容。但是，如果您希望了解最低的支持版本，下面是一个简略的汇总。

- servlet 2.3 和 JSP 1.2(独立服务器)。Java 1.2 或更新的版本。
- J2EE 1.3(包括 servlet 2.3 和 JSP 1.2)。Java 1.3 或更新的版本。
- servlet 2.4 和 JSP 2.0(独立服务器)。Java 1.3 或更新的版本。
- J2EE 1.4(包括 servlet 2.4 和 JSP 2.0)。Java 1.4 或更新的版本。

本书以 Java 1.4 为例进行介绍。

对于 Solaris, Windows 和 Linux，可以到 <http://java.sun.com/j2se/1.4/> 获取 Java 1.4，<http://java.sun.com/j2se/1.3/> 获取 Java 1.3。要确保下载的是 SDK(软件开发工具包)，而非仅仅是 JRE(Java Runtime Environment, Java 运行环境)——JRE 只执行已经编译好的 Java 类文件，不提供编译器。对于其他平台，首先检查这些平台是否预装了 Java 2 的实现，就如同 Mac OS X 一样。如果没有，请到 <http://java.sun.com/cgi-bin/java-ports.cgi> 查询 Sun 提供的第三方 Java 实现的清单。

您使用的 Java 实现应该拥有完整的配置操作指南，但关键是设置环境变量 PATH(不是 CLASSPATH)，使之指出包括 java 和 javac 的目录，一般是 *java_install_dir/bin*。例如，如果您使用 Windows 操作系统，并将 SDK 安装在 C:\j2sdk1.4.1_01 中，您可能需要将下面的行加入到 C:\autoexec.bat 文件中。切记，autoexec.bat 只在系统启动时执行。

```
set PATH=C:\j2sdk1.4.1_01\bin;%PATH%
```

如果您想下载已经配置好的 autoexec.bat 文件，包含 PATH 设定以及本章论述的其他设定，可以到 <http://www.coreservlets.com/>，转到相应的源代码档案文件后，选择第 2 章。

在 Windows NT/2000/XP 中，您还可以在 My Computer 上右击鼠标，选择 Properties，然后选择 Advanced，再选择 Environment Variables(【我的电脑】 | 【属性】 | 【高级】 | 【环境变量】)。更新 PATH 的值，然后单击 OK 按钮。

在 Unix 中(Solaris, Linux 等)，如果 SDK 安装在 /usr/j2sdk1.4.1_01 中，并且您使用 C 外壳，您需要将下面的内容放入到 .cshrc 文件中。

```
setenv PATH /usr/j2sdk1.4.1_01/bin:$PATH
```

重启(Windows；如果交互式地设定变量，则不需重启)，或注销后再次登录(Unix)之后，

打开一个 DOS 窗口(Windows)或外壳(Unix)，输入 `java -version` 和 `javac -help` 来检验 Java 的设置是否正确。两种情况下，您都应该看到一段真实的结果，而不应该是有关未知命令的错误消息。相应地，如果您使用集成开发环境(Integrated Development Environment, IDE)，比如 Borland JBuilder, Eclipse, IntelliJ IDEA，或 Sun ONE Studio，您应该试着编译并运行一个简单的程序，证实 IDE 确实知道 Java 的安装位置。

2.2 为桌面计算机下载服务器

第二步是下载实现了 servlet 2.3 规范(JSP 1.2)或 servlet 2.4 规范(JSP 2.0)的服务器(经常称为“servlet 容器”或“servlet 引擎”)，用于您的桌面计算机。实际上，我们一般会在桌面计算机上同时安装 3 种服务器(Apache Tomcat, Macromedia JRun, 和 Caucho Resin)，并在所有的服务器上测试应用程序，以了解与跨平台部署相关的问题，并防止我们不小心应用了不可移植的特性。在本书中，我们将给出每种服务器的细节。

不管最终部署在什么服务器上，我们都希望至少在桌面计算机上有一个供开发用的服务器。即使部署服务器就在隔壁，并与您的计算机之间存在高速网络连接，您依旧希望在开发时使用运行在自己桌面计算机上的服务器。对于开发来说，即使内联网上客户不能访问的测试服务器，使用起来也要比您的桌面计算机上安装的服务器要麻烦得多。与每次测试都需部署到远程的服务器相比，在桌面计算机上运行部署服务器，部署工作在很多方面都得到了简化。具体原因如下：

- 测试更为快速。

在桌面计算机上安装服务器之后，就不再需要使用 FTP 或其他上载程序。对更改的测试越困难，测试的次数就会越少。不经常测试可能会使错误得不到改正，从长远来看，会降低开发的效率与速度。

- 易于调试。

在桌面计算机上运行服务器时，许多服务器在普通窗口中显示标准输出，与之相对应的是，部署服务器的标准输出几乎不是隐藏，就是只能在执行结束后查看日志文件。因此，在桌面服务器上，简单的 `System.out.println` 语句成为跟踪和调试的有效工具。

- 易于重启。

在开发过程中，经常需要重启服务器，或者重新载入 Web 应用。例如，服务器一般只在启动时，或给出服务器专有命令重新载入 Web 应用时，才会读取 `web.xml` 文件(参见 2.11 节)。因此，每次修改完 `web.xml` 之后，一般都必须重启服务器或重新载入 Web 应用。即使服务器为重新载入 `web.xml` 提供交互式的方法，诸如清除会话数据、重置 `ServletContext`、以及 `servlet` 或 JSP 页面间接使用的类文件(例如，`bean` 或工具类)发生改变后需要进行的替换操作等，这些任务有可能依旧需要重启服务器。一些老一些的服务器也需要重启，因为它们实现的 `servlet` 重新载入(`servlet reloading`)不稳定或不可靠。(一般来说，服务器只将 `servlet` 对应的类实例化一次，然后将该实例保存在内存中，供不同的请求使用。对于具有 `servlet` 重新载入功能的服务器，如果与内存中的 `servlet` 相对应的磁盘上的类文件发生改变后，

它会自动替换内存中的 servlet。)另外,一些用于部署的服务器推荐完全禁止 servlet 重新载入功能,以提高系统的性能。因此,在鼠标一点,无须征得其他开发人员的允许,就可以重启服务器并重新载入 Web 应用的环境中进行开发,会更有效率。

- **基准更为可靠。**

对于短期运行的程序,即使在最好的情况下,依旧难以取得精确的计时结果。在负载极为沉重和多样的多用户系统上,得出的运行基准就更为不可靠。

- **完全控制。**

作为开发人员,您可能不是运行测试或部署服务器的系统的管理员。也许,每次您希望重启服务器时,都不得不请求某个系统管理员。或者,在开发周期最为紧要的时刻,远程系统有可能当机进行系统升级。

- **易于安装。**

下载和配置服务器一般可以在一个小时内完成。通过在桌面计算机上运行服务器,而非运行远程服务器,您可能在进行开发的第一天就能省下这些时间。

如果能够在桌面计算机上运行实际部署中使用的服务器,那当然是最好不过了。因此,如果您的部署服务器是 BEA WebLogic, IBM WebSphere, Oracle9i AS 等,并且您购买的许可允许您同时在桌面计算机上运行该服务器,当然可以这样做。但是, servlet 和 JSP 的优点之一就是您不是必须这样做;您可以在一种服务器上进行开发,并将开发结果部署到另外的服务器上。

下面列出的是一些可以用于桌面开发的最为流行的免费服务器。所有的情况下,免费的版本都是以独立 Web 服务器的形式运行。大多数情况下,如果希望使用可以与常规 Web 服务器(如 Microsoft IIS, iPlanet/Sun ONE Server, Zeus 或 Apache Web Server)进行集成的部署版本,您要为之付费。然而,这些服务器无论是作为 servlet 和 JSP 引擎应用到常规 Web 服务器中,还是作为完全独立的 Web 服务器来使用,对于开发来说,在性能上的差异完全可以忽略。请到 <http://java.sun.com/products/servlet/industry.html> 查看更为完整的支持 servlet 和 JSP 的服务器和服务器插件的清单。

- **Apache Tomcat**

Tomcat 5 是 servlet 2.4 和 JSP 2.0 规范的官方参考实现。Tomcat 4 是 servlet 2.3(JSP 1.2)的官方参考实现。这两种版本在开发过程中都可以作为独立服务器使用,或在部署过程中插入到标准的 Web 服务器中,供具体的项目使用。和所有的 Apache 产品一样, Tomcat 完全免费,并提供完整的源代码。在所有的服务器中,它对最新 servlet 和 JSP 规范的支持最好。而商业服务器一般文档更齐全、明晰,更易于配置,速度也更快。要下载 Tomcat,先到 <http://jakarta.apache.org/tomcat/>,进入执行程序下载一节,并选取 Tomcat 的最新版本。

- **Macromedia JRun**

JRun,作为 servlet 和 JSP 引擎,既能够以独立的模式用在开发过程中,也可以在部署过程中,插入到大多数常见的商业 Web 服务器中。用于开发目的时,它是免费的,但在部署它之前需要购买许可。那些觉得 Tomcat 难以管理的开发人员,常常会选择 JRun。详细情况,请访问 <http://www.macromedia.com/software/jrun/>。

- **Caucho 的 Resin**

Resin 是一个快速的 servlet 和 JSP 引擎，对 XML 提供广泛的支持。和 Tomcat 与 JRun 一道，它是商业 Web 托管公司提供 servlet 和 JSP 支持时，最常使用的 3 种服务器之一。对于开发和非商业部署来说，完全免费。详细情况，请访问 <http://caucho.com/products/resin/>。

- **New Atlanta 的 ServletExec**

ServletExec 是另一种广泛应用的 servlet 和 JSP 引擎，既可以独立运行，用于开发工作，也可以在部署过程中插入到 Microsoft IIS，Apache 和 Sun ONE 服务器中。ServletExec 提供免费下载，并可免费使用，但一些高性能功能和管理工具是禁用的，如果想启用这些功能，则需要购买许可。作为独立的桌面开发服务器时，我们选用 ServletExec Debugger 配置。详细信息，请访问 <http://www.newatlanta.com/products/servletexec/>。

- **Jetty**

Jetty 是一个开放源码的服务器，它支持 servlet 和 JSP 技术，并可免费用于开发和部署。它常常用作完全独立的服务器(而非集成到不支持 Java 的 Web 服务器中)，即使对于部署也是如此。详细信息，请访问 <http://jetty.mortbay.org/jetty/>。

2.3 服务器的配置

下载并安装好 Java 平台和支持 servlet 和 JSP 的服务器之后，我们需要对服务器进行配置，使之能够在我们的系统上运行。配置内容涉及下面的通用步骤；下面 3 节分别给出配置 Tomcat，JRun 和 Resin 的具体细节。

请注意，这些说明都是针对独立运行且用于桌面开发的 Web 服务器。在部署时，一般会将服务器作为插件，集成到传统 Web 服务器中，如 Apache 或 IIS。这种配置超过了本书所讲述的范围；可以使用服务器提供的向导，或阅读提供商的文档从中获取配置的指导。

(1) 确定 SDK 的安装目录。

要编译 JSP 页面，服务器需要知道 Java 类的位置，Java 编译器要使用这些类(例如 javac 或 jikes)。针对大多数服务器，要么服务器的安装向导检测出 JDK 的所在目录，要么我们需要设定 JAVA_HOME 环境变量，使之指向这个目录。JAVA_HOME 应该列出 SDK 安装目录的根目录，而非 bin 子目录。

(2) 指定端口。

大多数服务器都预先配置为使用非标准端口，这是为了避免与现有使用 80 端口的服务器发生冲突。如果没有服务器使用 80 端口，为了方便起见，可以将新安装的服务器设为使用该端口。

(3) 执行服务器专有的定制工作。

不同的服务器这些设定也各不相同。要仔细阅读服务器的安装说明。

2.4 配置 Apache Tomcat

在所有流行的 servlet 和 JSP 引擎中，Tomcat 最难配置。和大多数其他服务器相比，

Tomcat 的变动也最为频繁。Tomcat 的发布更为频繁，每个版本在安装和配置上都有较大的变动。因此，为了处理 Tomcat 的新版本，我们在 <http://www.coreservlets.com/> 上维护一个不断更新的 Web 页面，专门介绍 Tomcat 的安装和配置。我们的在线 Tomcat 配置页面包括您需要编辑的 3 个重要文件，autoexec.bat, server.xml 和 web.xml。如果您使用 Tomcat 4.1.24 之后的版本，可以到网站查阅最新的详细信息。下面的指示都是根据 4.1.24 版本做出的。

第一步是从 <http://jakarta.apache.org/tomcat/> 下载 Tomcat 的压缩文件。单击 Binaries，并选择最新的发布版本。假定您使用 JDK 1.4，请选择“LE”版本(如 tomcat-4.1.24-LE-jdk14.zip)。接下来，将文件解压缩到选定的位置。唯一的限定是，该位置不能限制写访问：Tomcat 在运行时需要创建临时文件，因此，启动 Tomcat 的用户必须能够对 Tomcat 的安装位置拥有写权限。解压缩 Tomcat 会生成类似 C:\jakarta-tomcat-4.1.24-LE-jdk14 的顶层目录(此后称为 *install_dir*)。下载并解压缩 Tomcat 文件之后，服务器的配置涉及下面的这些步骤。我们在下面给出简短的总结，然后在接下来的小节中提供相关的细节。

(1) **设置 JAVA_HOME 变量。**

设置这个变量，使之列出 SDK 的安装根目录。

(2) **指定服务器的端口。**

编辑 *install_dir/conf/server.xml*，并将 Connector 元素中 port 属性的值从 8080 改为 80。

(3) **启用 servlet 重新载入功能。**

向 *install_dir/conf/server.xml* 加入 DefaultContext 元素，告诉 Tomcat 如果 servlet 载入到服务器的内存中之后，磁盘上对应的类文件发生更改，则重新载入相关的 Servlet。

(4) **启用 ROOT 上下文。**

为了启用默认的 Web 应用，将 *install_dir/conf/server.xml* 中下面的行取消注释。

```
<Context path="" docBase="ROOT" debug="0"/>
```

(5) **开启 servlet 调用器。**

为了能够在不更改 web.xml 文件的情况下运行 servlet，某些版本的 Tomcat 要求将 *install_dir/conf/web.xml* 中的 /servlet/* servlet-mapping 元素取消注释。

(6) **增加 DOS 的内存限制。**

在较早期版本的 Windows 中，需要告诉操作系统为环境变量保留更多的空间。

(7) **设置 CATALINA_HOME。**

可选地，将 CATALINA_HOME 环境变量设为 Tomcat 的安装目录。

下面的小节分别详细说明每个步骤。请注意，这一节介绍 Tomcat 作为独立服务器在 servlet 和 JSP 开发上的应用。如果将 Tomcat 作为 servlet 和 JSP 容器，集成到常规 Web 服务器(例如，使用 Apache Web 服务器的 mod_webapp)中，则需要完全不同的配置。有关 Tomcat 在部署中的应用，请访问 <http://jakarta.apache.org/tomcat/tomcat-4.1-doc/>。

2.4.1 设置 JAVA_HOME 变量

最关键的 Tomcat 设置是 JAVA_HOME 环境变量——如果这项设置不正确，则 Tomcat 找不到 javac 使用的那些类，从而也就不能处理 JSP 页面。这个变量应该列出 SDK 安装目

录的根目录，而非 bin 目录。例如，如果您使用的是 Windows，并且将 SDK 安装到 C:\j2sdk1.4.1_01，那么在 C:\autoexec.bat 中可以加入下面的行。切记 autoexec.bat 只在系统启动时执行。

```
set JAVA_HOME=C:\j2sdk1.4.1_01
```

在 Windows NT/2000/XP 上，也可以在 My Computer(【我的电脑】)上有击鼠标，选择 Properties(【属性】)，之后选择 Advanced(【高级】)，之后选择 Environment Variables(【环境变量】)。然后，您可以输入 JAVA_HOME 值，并单击 OK(【确定】)。

在 Unix 上(Solaris, Linux, MacOS X, AIX 等)，如果 SDK 安装在/usr/local/java1.4 中，并且您使用的是 C 外壳，可以将下面的行加入到.cshrc 文件中。

```
setenv JAVA_HOME /usr/local/java1.4
```

和在操作系统内全局地设置 JAVA_HOME 环境变量相比，一些开发人员更喜欢编辑 Tomcat 的启动脚本，在其中设置变量。如果您倾向于这种策略，可以编辑 *install_dir/bin/catalina.bat*(Windows)并将下面的行插入到文件的顶部，在开头的一系列注释之后。

```
set JAVA_HOME=C:\j2sdk1.4.1_01
```

在修改 catalina.bat 之前，一定要先制作 catalina.bat 的备份副本。Unix 用户要对 catalina.sh 做出类似的更改。

2.4.2 指定服务器的端口

2.2 节列出的大多数免费服务器都使用非标准的默认端口，以避免与其他使用标准端口(80)的 Web 服务器发生冲突。Tomcat 也不例外：它默认使用 8080 端口。然而，如果您以独立模式使用 Tomcat(即作为完整的 Web 服务器，而非仅仅作为 servlet 和 JSP 引擎集成到另外的 Web 服务器中)，并且没有其他的运行服务器永久占用 80 端口，那么使用 80 端口会更方便。如果使用 80 端口，那么在浏览器中输入 URL 时不用总是使用端口号。然而，要注意，Unix 中，在 80 端口或其他小于 1024 的端口号上启动服务，必须具有管理员权限。在桌面计算机上，您可能拥有这类权限，但在部署服务器上，则不一定会拥有这些权限。此外，许多 Windows XP Professional 系统上，Microsoft IIS 已经注册了 80 端口；要在 80 端口上运行 Tomcat，则需要禁用 IIS。可以通过 Control Panel(【控制面板】)的 Administrative Tools/Internet Information Services(【管理工具】|【Internet Information Services】)永久禁用 IIS。

修改端口号涉及编辑 *install_dir/conf/server.xml*，将 Connector 元素的 port 属性从 8080 改为 80，并重新启动服务器。请用 Tomcat 的安装位置替换 *install_dir*。例如，如果您下载的是 Java 1.4 版本的 Tomcat 4.1.24，并将其解压缩到 C:\目录中，那么需要编辑的是 C:\jakarta-tomcat-4.1.24-LE-jdk14\conf\server.xml。

在 Tomcat 中，元素最初的样子可能如下：

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"  
port="8080" minProcessors="5" maxProcessors="75"  
... />
```

我们应该将它改成下面这样:

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
    port="80" minProcessors="5" maxProcessors="75"
    ... />
```

请注意,对于 Tomcat 的不同版本,这项元素也会稍有不同。找出正确条目最简单的方式是在 server.xml 中查找 8080; 8080 在非注释内容中应该只出现一次。在修改 server.xml 之前,要先保存它的备份,以防您会犯错误使得服务器不能运行。并且,要记住 XML 是大小写敏感的,因此,不能将 port 替换为 Port,或是将 Connector 替换为 connector。

2.4.3 启用 servlet 重新载入功能

下一步是告诉 Tomcat,检查与被请求的 servlet 相对应的类文件的修改日期,重新载入那些载入到服务器内存之后,类文件发生过改变的 servlet。在部署环境下,这对性能会有影响,所以默认情况下,它是关闭的。然而,如果不开启开发服务器的这个选项,那么对于那些已载入到服务器内存的 servlet,每次重新编译之后,都得重新启动服务器或重新载入 Web 应用。

要打开 servlet 重新载入功能,需要编辑 *install_dir/conf/server.xml*,向主 Service 元素中加入 DefaultContext 子元素,并将 reloadable 属性设为 true。最简单的方式是找到下面的注释:

```
<!-- Define properties for each web application. ...
... -->
```

紧随其后插入下面的行:

```
<DefaultContext reloadable="true"/>
```

同样,在做出这项修改之前,一定要保存 server.xml 的备份副本。

2.4.4 启用 ROOT 上下文

ROOT 上下文是 Tomcat 中默认的 Web 应用;在您初次涉足 servlet 和 JSP 技术时,使用它比较方便(尽管您稍微熟练之后就会使用自己的 Web 应用——参见 2.11 节)。在 Tomcat 4.0 以及 Tomcat 4.1 的某些版本中,默认的 Web 应用已经启用。但在 Tomcat 4.1.24 中,默认情况下它是禁用的。如果要启用它,请将 *install_dir/conf/server.xml* 中下面的行取消注释。

```
<Context path="" docBase="ROOT" debug="0"/>
```

2.4.5 开启 servlet 调用器

servlet 调用器(invoker servlet)允许在不修改 Web 应用的 WEB-INF/web.xml 文件的情况下,运行 servlet。您只需要将 servlet 复制到 WEB-INF/classes 目录中,使用 URL *http://host/servlet/ServletName*(默认 Web 应用)或 *http://host/webAppPrefix/servlet/ServletName*(定制 Web 应用)就可以使用它。首次接触 servlet 和 JSP,甚至在真实项目的最初开发阶段,使用 servlet 调用器都极为方便。但是,随着本书论述的逐渐深入,您会了解到,

在部署期间，一般不会将它打开。在 Apache Tomcat 4.1.12 之前，默认情况下调用器还是启用的。然而，一个与之相关的安全缺陷最近被揭示出来，servlet 调用器可以用来查看由 JSP 页面生成的 servlet 源代码。虽然大多数情况下，这无关紧要，但它有可能会向外界泄露专有代码，因此，Tomcat 4.1.12 中，调用器默认情况下是禁用的。我们猜想 Jakarta 项目将会很快修复这个问题，并在将来发布的 Tomcat 程序中重新启用 servlet 调用器。然而，在学习时，您几乎肯定希望启用它。不过要确保只在不能被外界访问的桌面开发计算机上启用它。

如果要启用 servlet 调用器，需要将 *install_dir/conf/web.xml* 文件中下面的 *servlet-mapping* 元素取消注释。要注意，文件名是 *web.xml*，不是 *server.xml*，并且不要将这个 Tomcat 专有的 *web.xml* 文件与每个 Web 应用放入到 WEB-INF 目录中的标准文件相混淆。

```
<servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

2.4.6 增加 DOS 的内存限制

如果您使用的是 Windows 的早期版本(比如 Windows 98/Me 或更早)，可能需要更改服务器的启动和停止脚本的 DOS 内存设置。如果在启动服务器时得到“Out of Environment Space”错误消息，您需要右击 *install_dir/bin/startup.bat*，选择 Properties(【属性】)，选择 Memory(【内存】)，将 Initial Environment(【初始环境】)项从 Auto(【自动】)改为至少 2816。对 *install_dir/bin/shutdown.bat* 重复相同的过程。

2.4.7 设置 CATALINA_HOME

有些情况下，让 CATALINA_HOME 环境变量指向 Tomcat 的安装根目录对于系统的设置也很有帮助。这个变量向服务器标识出各种 Tomcat 文件的位置。然而，如果您不复制服务器的启动和停止脚本，而只是使用快捷方式(在 Unix 上称为“symbolic links”)，那么您不必设置这个变量。2.9 节给出得到更多有关使用这些快捷方式的信息。

2.4.8 测试服务器的基本设置

要验证您是否成功地完成了 Tomcat 的配置工作，可以双击 *install_dir/bin/startup.bat* (Windows) 或执行 *install_dir/bin/startup.sh*(Unix/Linux)。之后，打开浏览器，输入 *http://localhost/*(如果您没将端口改为 80，则使用 *http://localhost:8080/*)，应该看到类似图 2.1 的内容。关闭服务器可以通过双击 *install_dir/bin/shutdown.bat*(Windows) 或执行 *install_dir/bin/shutdown.sh*(Unix)。如果 Tomcat 不能运行，请到 *install_dir/bin* 目录中，输入 *catalina run*；这样可以阻止 Tomcat 启动另外的单独窗口，从而我们可以看到诸如端口被占用，或 JAVA_HOME 定义不正确等错误消息。

定制完开发环境之后(参见 2.7 节)，一定要执行 2.8 节中列出的详尽测试。

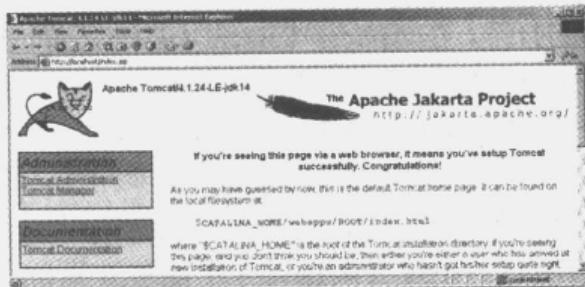


图 2.1 Tomcat 的主页

2.5 配置 Macromedia JRun

要在桌面计算机上使用 JRun，第一步是到 <http://www.macromedia.com/software/jrun/>，下载 JRun 用于开发的免费版本，并运行安装向导。大部分的配置设定都是在安装期间指定。我们希望指定的设定主要有 7 项。下面对它们进行了简短的总结；有关的细节在随后的小节中给出。

(1) 序列号。

如果用作免费的开发服务器，可以不填。

(2) 用户限制。

您可以限定只有您的账户才能使用 JRun，或者向系统中的任何人开放。

(3) SDK 的安装位置。

给出 SDK 的安装根目录，而非 bin 子目录。

(4) 服务器的安装目录。

大多数情况下，我们接受默认值。

(5) 管理员的用户名和密码。

在后期的附加定制过程中，将会需要这些信息。

(6) 自动启动功能。

在开发过程中，我们不希望 JRun 自动启动。特别地，在 Windows 中，不应该将 JRun 作为 Windows 服务。

(7) 服务器端口。

您可能希望将它从 8100 改为 80。

2.5.1 JRun 的序列号

以开发模式使用 JRun 时(即只接受来自本机的请求时)，不需要序列号。因此，除非您将服务器用作完全的部署版本，否则当提示输入序列号时，可以保持序列号为空。在需要的时候，无需重新安装服务器，就可以将开发模式的 JRun 升级成部署版本。参见图 2.2。

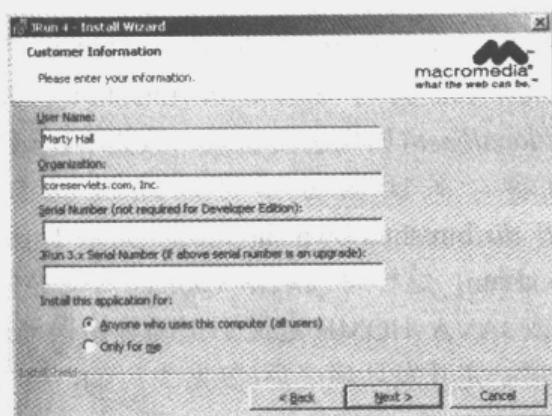


图 2.2 使用免费开发版本的 JRun 可以省略序列号

2.5.2 JRun 的用户限制

在安装 JRun 时，安装程序会询问是否希望系统上的所有用户都可以使用该服务器，或是仅仅限于您的账户。参见图 2.2。请根据需要进行选择。

2.5.3 Java 的安装位置

安装向导将会查找 Java 的安装，并将其根目录作为默认的选择。如果这项选择指出的是系统上 Java 的最新版本，则接受默认值。然而，如果安装向导找出了 Java 的旧版本，则要选择 Browse，给出另外的位置。这种情况下，一定要确保给出的是根目录的位置，而不是 bin 子目录。同样，要确保指出的是完整 SDK 的位置(在 Java 1.3 和更早版本中称为“JDK”），而不是 JRE(Java 运行环境)的位置——JRE 目录缺乏编译 JSP 页面所需的类。参见图 2.3。

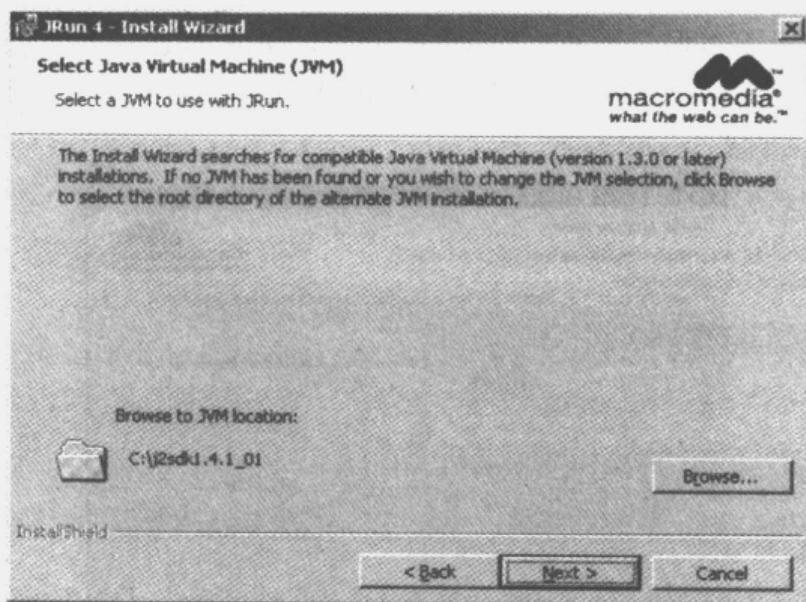


图 2.3 确保 JVM 的位置指向安装最新 Java 版本的根目录

2.5.4 服务器的安装位置

对于这个选项，您可以选择任何目录。大多数用户只是简单地接受默认值，在 Windows 上，是 C:\JRun4。

2.5.5 管理员用户名和密码

安装向导会提示您输入用户名和密码。您可以输入任意的值，但要记住您所指定的内容；之后，定制服务器时需要用到这些信息。参见图 2.4。

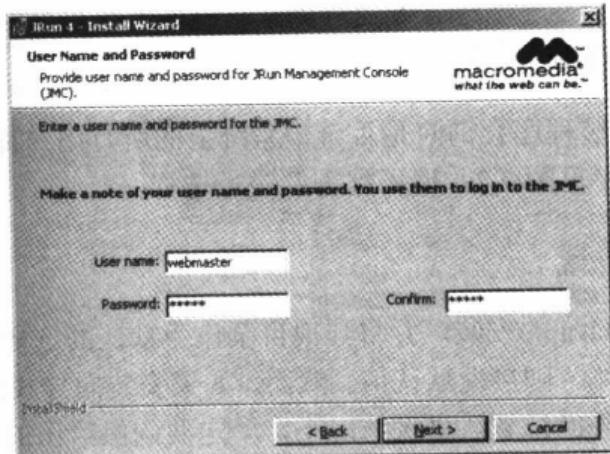


图 2.4 牢记管理员的用户名和密码

2.5.6 自动启动功能

在将 JRun 用作开发服务器时，手动启动和停止 JRun 要比让操作系统自动启动 JRun 更为方便。因此，当提示您是否希望 JRun 作为 Windows 服务时，不要选择此项。参见图 2.5。

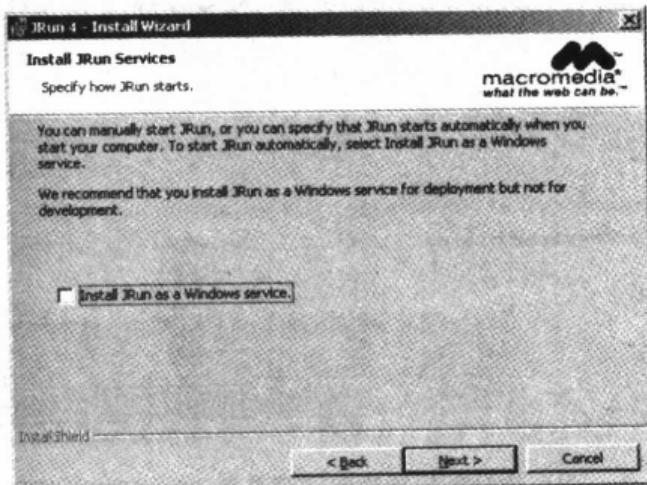


图 2.5 不要将 JRun 安装为 Windows 服务

2.5.7 服务器端口

完成安装之后，到 Start(【开始】)菜单，选择 Programs(【程序】)，选择 Macromedia JRun 4，并选择 JRun Launcher。选择 admin 服务器，单击 Start，对 default 服务器重复相同的工作。参见图 2.6。

接下来，或者打开浏览器并输入 URL <http://localhost:8000/>，或者到 Start(【开始】)菜单，选择 Programs(【程序】)，选择 Macromedia JRun 4，并选择 JRun Management Console。这两种方式都会调出一个 Web 页面，提示您输入用户名和密码。输入您在安装过程中指定的值，然后选择左窗格 default 服务器下的 Services。这项操作将会产生类似图 2.7 的结果。下面，选择 WebService，将端口从 8100 改为 80，点击 Apply，之后停止并重新启动该服务器。

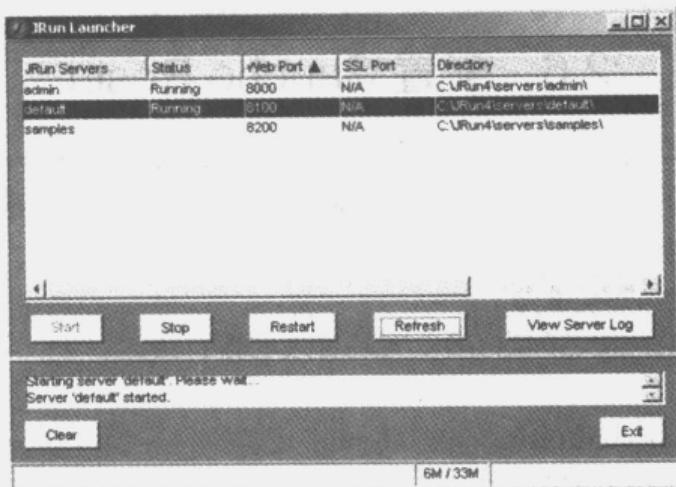


图 2.6 使用 JRun Launcher 启动和停止管理服务器和默认服务器

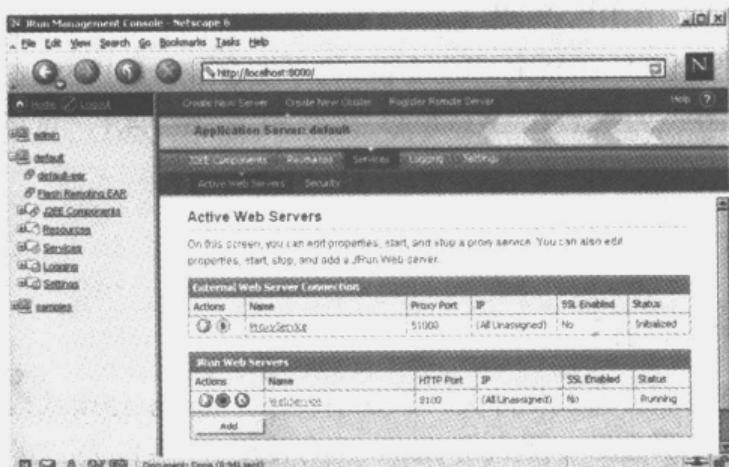


图 2.7 选择 default 服务器下的 Services 之后, 选择 WebService, 编辑默认的 JRun 服务器的端口

2.5.8 测试服务器的基本设置

如果要验证 JRun 的配置是否正确, 可以通过 Start(【开始】)菜单, 选择 Programs(【程序】), 选择 Macromedia JRun 4, 之后, 打开 JRun Launcher。如果您更改了服务器的端口, 那么服务器可能已经在运行中。(请注意, 除非想要修改附加的服务器选项, 否则不需要启动 admin 服务器。)打开浏览器, 输入 <http://localhost/>(如果您未将端口改为 80, 则输入 <http://localhost:8100/>)。您看到的内容应该与图 2.8 类似。在 JRun Launcher 中单击 Stop 可以停止服务器。

在定制完开发环境之后(参见 2.7 节), 一定要执行 2.8 节中列出的更为详尽的测试。

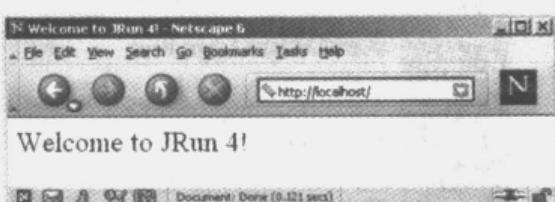


图 2.8 JRun 的主页

2.6 配置 Caucho Resin

要在您的桌面计算机上运行 Resin，首先要从 <http://caucho.com/products/resin/> 下载 Resin 的压缩文件，将它解压缩到选定的位置中(此后称为 *install_dir*)。完成这些工作后，配置服务器涉及两个简单的步骤。

(1) **设置 JAVA_HOME 变量。**

设置这个变量，使之列出 SDK 安装目录的根目录。

(2) **指定端口。**

编辑 *install_dir/conf/resin.conf*，将 http 元素的 port 属性从 8080 改为 80。

细节在随后的小节中给出。

2.6.1 设置 JAVA_HOME 变量

最重要的设置是 JAVA_HOME 环境变量。这个变量应该指出 SDK 安装目录的根目录。详细情况参见 2.4 节，如果您在 Windows 上使用 Java 1.4.1，可以将下面的行加入到 C:\autoexec.bat 中。

```
set JAVA_HOME=C:\j2sdk1.4.1_01
```

2.6.2 指定 Resin 的端口

为了避免与已有的服务器发生冲突，默认情况下，Resin 使用端口 8080。然而，如果您不会同时运行另外的服务器，那么将 Resin 改为使用 80 端口(标准的 HTTP 端口)可能更为方便。要完成这项更改，需要编辑 *install_dir/conf/resin.conf*，将<http port='8080'>改为<http port='80'>。

2.6.3 测试服务器的基本设置

要检验 Resin 的配置是否成功，可以双击 *install_dir/bin/httpd.exe*。之后，打开浏览器输入 <http://localhost/>(如果没有将端口改为 80，则输入 <http://localhost:8080/>)。此时应该看到类似图 2.9 的内容。在启动服务器时，一个小型的对话框会弹出来，可以选择对话框上的 Stop，停止服务器的运行。

在定制完开发环境之后(参见 2.7 节)，一定要执行 2.8 节中列出的详尽测试。



图 2.9 Resin 的主页

2.7 建立开发环境

配置并测试完服务器之后，是不是就万事大吉了呢？不是，还没有全部完成。这只是完成了本地部署环境而已。您还需要建立个人开发环境。否则，您将不能编译您编写的 servlet 和辅助 Java 类。配置开发环境分为下面几步：

(1) **创建开发目录。**

选定一个位置，用于开发 servlet、JSP 文档和支持类。

(2) **设置 CLASSPATH。**

告诉编译器 servlet 和 JSP JAR 文件的相关信息，以及开发目录的位置。初学者遇到的大部分常见问题都是由于这个变量设置不正确引起的。

(3) **创建启动和停止服务器的快捷方式。**

确保可以方便地启动和停止服务器。

(4) **标记或安装 servlet 和 JSP 的 API 文档。**

您将会频繁地参考这份文档，因此，一定要能够方便地访问它。

随后的小节详细地介绍这些步骤。

2.7.1 创建开发目录

您应该做的第一件事是创建一个目录，之后开发的 servlet 和 JSP 文档就放置在这个目录中。这个目录可以在您的主目录(如 Unix 上的~/ServletDevel)中，或是位于方便的普通位置(如 Windows 上的 C:\ServletDevel)。但不要将它放在服务器的安装目录中。

最后，您可能在这个开发目录中组织不同的 Web 应用(它们都遵循通用的结构——参见 2.11 节)。但是，对于开发环境的初始测试，您可以将 servlet 直接放入开发目录中(无包装的 servlet)，或是放入名称与 servlet 的包名相同的子目录中。编译之后，只需将生成的类文件复制到服务器默认的 Web 应用中。

许多开发人员将他们所有的代码都放在服务器的部署目录中(参见 2.10 节)。我们极不赞成这种做法，并推荐使用 2.9 节中介绍的方案。尽管在部署目录中开发，由于不需要复制文件，在一开始可能显得要方便些，但在以后的开发工作中，它会将事情搞得极为复杂。混淆开发和部署位置，使得从正在测试的版本中分离出一个可以运行的版本很困难，且难以在多个服务器上进行测试，同时还使组织工作变得极为复杂。另外，由于您的桌面计算机毫无疑问并非最终的部署服务器，因此无论如何您最终必须开发一个好的系统用于部署。

警告

不要将服务器的部署目录作为开发场所。应该在单独的目录中进行开发。

2.7.2 设置 CLASSPATH

由于 servlet 和 JSP 不是 Java 2 平台标准版的一部分，因此，您必须告诉编译器 servlet 类的位置。服务器虽然已经知道 servlet 类，但用于开发的编译器可能尚不知道。因此，如

果在未设置 CLASSPATH 的情况下尝试编译 servlet，标签库或其他使用 servlet API 的类可能不能通过编译，编译器会给出错误信息，说明存在未知的类。servlet JAR 文件的精确位置，每种服务器都各不相同。大多数情况下，您可以在 *install_dir/lib* 目录中查找。或阅读服务器的文档找出这个位置。找到 JAR 文件后，将它的位置加入 CLASSPATH 中。下面是 servlet JAR 文件在一些常见开发服务器中的位置。

- Tomcat
install_dir/common/lib/servlet.jar
- JRun
install_dir/lib/jrun.jar
- Resin
install_dir/lib/jsdk23.jar

除 servlet 的 JAR 文件之外，您还需要将开发目录放入 CLASSPATH 中。尽管对于简单的无包装 servlet 来说，这并非必需，但是在您稍微熟练之后，几乎肯定会使用包。如果文件属于包，并且使用了同一包中的其他类，那么在编译这个文件时，需要 CLASSPATH 中包括位于包层次顶部的目录。这种情况下，这个目录就是我们在第一小节中论述的开发目录。servlet 的初级程序员最容易犯的错误可能就是忘记这项设置。

核心方法

一定要将开发目录加入 CLASSPATH 中，否则，如果包中的 servlet 使用了同一包中其他的类，在编译时会得到“Unresolved symbol” 错误消息。

最后，您还应该在 CLASSPATH 中包括“.”(当前目录)。否则，您将只能编译那些在开发目录顶层的无包装的类。

下面是设定 CLASSPATH 的几种具有代表性的方法。它们假定您的开发目录是 C:\ServletDevel(Windows) 或 /usr/ServletDevel(Unix)，使用的服务器是 Tomcat 4。请将 *install_dir* 替换为服务器实际的安装位置。一定要注意文件名的大小写：它们是大小写敏感的(即使在 Windows 平台上！)。如果 Windows 路径中包含空格(例如，C:\Documents and Settings\Your Name\My Documents\...)，要将它括在双引号中。请注意，这些例子仅仅代表设定 CLASSPATH 的一种方式。例如，您还可以创建一个脚本，在调用 javac 时指定 -classpath 选项的值。另外，许多 Java 集成开发环境都有一个全局或项目专有的设置，可以达到同样的目的。由于这些设置完全依赖于具体的 IDE，故而在此不做讨论。

- Windows 95/98/Me

将下面的行放入 C:\autoexec.bat 中(注意，这些内容应该放到一行中，中间没有空格——在此将它们打断是为了易于阅读)。

```
set CLASSPATH=.;  
          C:\ServletDevel;  
          install_dir\common\lib\servlet.jar
```

- Windows NT/2000/XP

可以和上面一样使用 autoexec.bat，或者在 My Computer(【我的计算机】)上右击

鼠标，选择 Properties(【属性】)，然后选择 System(【系统】)，再选择 Advanced(【高级】)，然后选择 Environment Variables(【环境变量】)。之后，根据前面列出的内容输入 CLASSPATH 的值，单击 OK。

- Unix(C 外壳)

将下面的行放到.cshrc 中。(同样，在实际的文件中应该是一行，中间没有空格。)

```
setenv CLASSPATH .:  
    /usr/ServletDevel:  
    install_dir/common/lib/servlet.jar
```

2.7.3 创建启动和停止服务器的快捷方式

在开发过程中，我们经常需要重新启动服务器。因此，在主开发目录或桌面上放置启动和停止服务器的快捷方式能够极大地方便开发工作。在开发过程中就会感受到创建快捷方式带来的便利。

例如，对于安装在 Windows 上的 Tomcat，定位到 *install_dir/bin* 目录，在 startup.bat 上右击鼠标，选择 Copy(【复制】)。然后，转到开发目录，在窗口中右击鼠标，选择 Paste Shortcut(【粘贴快捷方式】)(而非 Paste(【粘贴】))。对 *install_dir/bin/shutdown.bat* 重复相同的过程。有些用户喜欢将快捷方式放在桌面上或 Start(【开始】)菜单中。如果将快捷方式放在这些地方，您甚至能够在快捷方式上右击鼠标，选择 Properties(【属性】)，然后通过在“Keyboard shortcut(【键盘快捷方式】)”文本框中键入一个键，创建键盘快捷方式。通过这些设置，您就能够在键盘上按下 Control-Alt-SomeKey 启动和停止服务器。

在 Unix 上，可以使用 ln -s 创建 startup.sh, tomcat.sh(这个文件是必需的，尽管我们不会直接调用这个文件)和 shutdown.sh 的符号链接(symbolic link)(对应于 Windows 的快捷方式)。

对于安装在 Windows 上的 JRun，首先到 Start(【开始】)菜单，选择 Programs(【程序】)，选择 Macromedia JRun 4，在 JRun Launcher 图标上右击鼠标，选择 Copy(【复制】)。然后，转到开发目录中，在窗口中右击鼠标，选择 Paste Shortcut(【粘贴快捷方式】)(而非 Paste(【粘贴】))。如果愿意的话，可以对 JRun Management Console 重复相同的过程。没有单独的停止图标，JRun Launcher 既可以启动服务器，也可以停止服务器。

对于安装在 Windows 上的 Resin，在 *install_dir/bin/httpd.exe* 上右击鼠标，选择 Copy(【复制】)。然后，转到开发目录，在窗口中右击鼠标，选择 Paste Shortcut(【粘贴快捷方式】)(而非 Paste(【粘贴】))。Resin 没有提供单独的停止图标；调用 httpd.exe 会弹出一个含有 Quit 按钮的窗口，可以用它来停止服务器。

2.7.4 标记或安装 servlet 和 JSP 的 API 文档

没有认真的程序员会在不参考 Java 1.4 或 Java 1.3 API 文档的情况下，开发通用 Java 应用程序，同样，没有认真的程序员会不参考 javax.servlet 包中类的 API，开发 servlet 和 JSP 页面。下面对 API 的出现地点进行了汇总。(请注意，<http://wwwcoreservlets.com> 中的源代码档案文件，除了提供所有例子的源代码之外，还列出了本书中引用的所有 URL 的最新链接。)

- <http://java.sun.com/products/jsp/download.html>
可以从这个网站下载 servlet 2.4(JSP 2.0) API 或 servlet 2.3(JSP 1.2) API 的 Javadoc 文件。您可能会发现这份 API 如此有用，除了在线浏览以外，值得下载到本地。然而，一些服务器捆绑了这份文档，因此，在下载之前应该首先检查它是否已经存在于您的计算机中。(参见接下来的一项)。
- **本地服务器**
一些服务器捆绑了 servlet 和 JSP 的 Javadoc。例如，在 Tomcat 中，您可以到默认的主页(<http://localhost/>)，单击 Tomcat Documentation，之后选择 Servlet/JSP Javadocs，就可以看到这份 API。或者选择 *install_dir/webapps/tomcat-docs/catalina/docs/api/index.html*，这样即使不运行 Tomcat，您依旧能够访问这些文档。然而，JRun 和 Resin 都没有捆绑这份 API。
- <http://java.sun.com/products/servlet/2.3/javadoc/>
这个网站允许在线浏览 servlet 2.3 的 API。
- http://java.sun.com/j2ee/sdk_1.3/techdocs/api/
可以在这个地址浏览 Java 2 平台企业版(J2EE)1.3 版的全部 API，包括 servlet 2.3 和 JSP 1.2 包。
- <http://java.sun.com/j2ee/1.4/docs/api/>
可以在这个地址浏览 Java 2 平台企业版(J2EE)1.4 版的全部 API，包括 servlet 2.4 和 JSP 2.0 包。

2.8 测试系统的设置

在开发自己的 servlet 和 JSP 页面时，首先要确保 SDK、服务器和开发环境都配置正确。对配置的检验可以分为 3 步，随后给出这 3 步的汇总；后面的小节给出更为详细的信息。

(1) 检验 SDK 的安装。

先要确保 java 和 javac 能够正常工作。

(2) 检查服务器的基本配置。

试着分别访问服务器的主页，一个简单的用户自定义 HTML 页面，以及一个简单的用户自定义 JSP 页面。

(3) 编译并部署一些简单的 servlet。

试验一个基本的无包装 servlet，一个使用包的 servlet，以及一个使用包和实用工具(辅助)类的 servlet。

2.8.1 检验 SDK 的安装

打开一个 DOS 窗口(Windows)或外壳(Unix)，输入 `java -version` 和 `javac -help`。两种情况下，您都应该看到实际的结果，而不应该是有关未知命令的错误消息。相应地，如果您使用集成开发环境(IDE)，则编译和运行一个简单的程序，来证实 IDE 知道 Java 的安装位置。如果任意一项测试失败，请回顾 2.1 节，并仔细检查随同 SDK 的安装指导。

2.8.2 检查服务器的基本配置

首先，启动服务器，并访问标准主页(<http://localhost/>，如果未将端口改为 80，则为 <http://localhost:port/>)。如果失败，则要回顾 2.3~2.6 节中介绍的指导信息，并仔细检查服务器的安装指导。

在检验完服务器能够运行之后，还要确保能够安装和访问简单的 HTML 和 JSP 页面。这项测试如果成功，则能够说明两件事。首先，能够成功访问 HTML 页面说明您了解哪个目录保存 HTML 和 JSP 页面。其次，能够成功访问新建的 JSP 页面说明 Java 编译器(而不仅仅是 Java 虚拟机)配置正确。

最终，您肯定希望创建和使用自己的 Web 应用(参见 2.11 节)，但针对最初的测试，我们推荐使用默认的 Web 应用。尽管 Web 应用都遵循共同的目录结构，但是默认 Web 应用的精确位置却和具体服务器有关。可以检查服务器的文档，获得说明性的指导，在此，我们汇总了 Tomcat、JRun 和 Resin 中默认 Web 应用的位置。在我们列出 *SomeDirectory* 的位置，您可以使用任意的目录名。(但不允许使用 WEB-INF 或 META-INF 作为目录名。对于默认的 Web 应用，您还必须避免使用与任何现有 Web 应用的 URL 前缀相匹配的目录名，比如 samples 或 examples。)如果您在本地计算机上运行测试，您可以在 URL 中 host 的位置使用 localhost。

- Tomcat HTML/JSP 目录
install_dir/webapps/ROOT
(或 *install_dir/webapps/ROOT/SomeDirectory*)
- JRun HTML/JSP 目录
install_dir/servers/default/default-ear/default-war
(或 *install_dir/servers/default/default-ear/default-war/SomeDirectory*)
- Resin HTML/JSP 目录
install_dir/doc
(或 *install_dir/doc/SomeDirectory*)
- 对应的 URL
http://host>Hello.html
(或 *http://host/SomeDirectory>Hello.html*)
http://host>Hello.jsp
(或 *http://host/SomeDirectory>Hello.jsp*)

在首次测试时，我们建议只是将 Hello.html(清单 2.1，图 2.10)和 Hello.jsp(清单 2.2，图 2.11)拖到相应的位置中。现在，先不要担心 JSP 文档。我们会在后面对此进行介绍。这些文件的代码，和本书中所有的代码一样，都在 <http://www.coreservlets.com> 上提供。该网站还包括本书中引用的所有 URL 链接、更新、辅助材料、培训课程的信息、以及其他 servlet 和 JSP 资源。它还包括一个经常更新的页面，介绍 Tomcat 的配置(由于 Tomcat 比其他服务器的变动更为频繁。)

如果 HTML 文件和 JSP 文件都不能工作(例如，得到 File Not Found—404—错误)，可能是将文件复制到错误的目录中，或者 URL 拼写错误(例如，Hello.jsp 中使用了小写的 h)。

如果 HTML 文件能够工作,但 JSP 文件失败,则可能是指定了不正确的 SDK 根目录(例如, JAVA_HOME 变量设置错误),应该回顾 2.7 节的内容。

清单 2.1 Hello.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>HTML Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>HTML Test</H1>
Hello.
</BODY></HTML>
```

清单 2.2 Hello.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JSP Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>JSP Test</H1>
Time: <%= new java.util.Date() %>
</BODY>
</HTML>
```



图 2.10 Hello.html 的结果

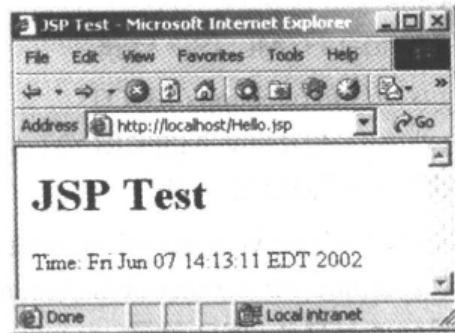


图 2.11 Hello.jsp 的结果

2.8.3 编译并部署一些简单的 servlet

好的,现在开发环境已经就绪。至少我们这样认为。但最好还是证实一下这个假设。下面给出的 3 个测试 servlet 可以帮助我们完成此项检验。

1. 测试 1: 不使用包的 servlet

第一个试用的 servlet 很基本:没有包,没有实用工具(辅助)类,只是简单的 HTML 输出。您可以不用编写自己的测试 servlet,而是从 <http://www.coreservlets.com/> 提供的本书源代码档案文件中下载 HelloServlet.java(清单 2.3)。同样,不要关注这个 servlet 的工作方式——在下一章会详细介绍——在此只是对您的设置进行测试。如果得到编译错误,则检查 CLASSPATH 设定(2.7 节)——您极有可能在列出包含 servlet 类的 JAR 文件的位置时犯错(如 servlet.jar)。

编译完 HelloServlet.java,将 HelloServlet.class 放入到适当的位置(一般是服务器默认 Web 应用的 WEB-INF/classes 目录)。可以从服务器文档中得出这个位置,或者查看下面的

列表，它汇总了 Tomcat，JRun 和 Resin 中对应的位置。然后，可以用 URL `http://host/servlet/HelloServlet`(如果没有按 2.3 节所述改变端口号，则使用 `http://host:port/servlet/HelloServlet`)访问该 servlet。如果在桌面系统上运行服务器，则用 `localhost` 替代 `host`。得到的内容应该与图 2.12 类似。如果访问这个 URL 失败，但对服务器自身的测试成功，那么极有可能是将类文件放错了目录。

注意，在 URL 中使用了 servlet(不是 servlets!)，尽管实际上并不存在名为 servlet 的目录。.../servlet/*ServletName* 形式的 URL 只是针对特殊 servlet 的一条指令(称为 servlet 调用器)，用以运行指定名称的 servlet。servlet 本身的代码可以存放在服务器通常用来存放 servlet 的任何位置(通常在.../WEB-INF/classes 中存储单个的类文件，.../WEB-INF/lib 中存储包含 servlet 的 JAR 文件)。使用这种默认的 URL 在初期开发过程中比较方便，但在准备好部署之后，几乎肯定会禁用这种功能，转而为每个 servlet 注册单独的 URL。详细信息请参见 2.11 节。实际上，服务器并非必须支持这类默认 URL，一些高端应用服务器，最突出的是 BEA WebLogic，就不提供此类支持。

- Tomcat 中存储 Java 的.class 文件的目录

install dir/webapps/ROOT/WEB-INF/classes

(注意：Tomcat 的许多版本中，必须手动地创建 classes 目录。)

- JRun 中存储 Java 的 .class 文件的目录

install_dir/servers/default/default-ear/default-war/WEB-INF/classes

- #### • Resin 中存储 Java 的 .class 文件的目录

install_dir/doc/WEB-INF/classes

- ## ● 对应的 URI

HelloServlet

清单2.3 HelloServlet.java

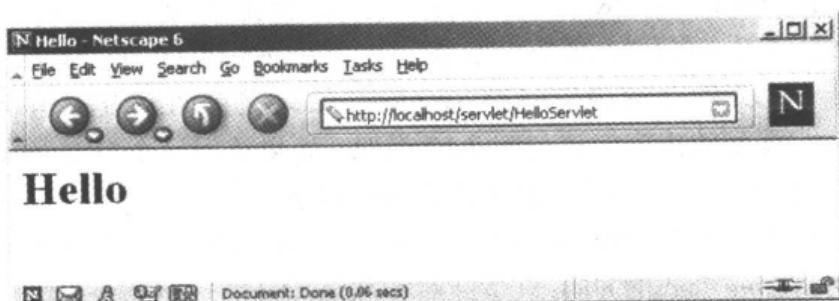


图 2.12 `http://localhost/servlet/HelloServlet` 的结果

2. 测试 2：使用包的 servlet

第二个试用的 servlet 使用包，但没有使用实用工具类。在 Java 编程语言中，包(package)是阻止类名冲突的标准机制。有 3 条规则需要牢记：

(1) 在代码中插入包声明。

如果类属于某个包，它的源代码中必须将“`package packageName;`”作为第一个非注释行。

(2) 使用与包名匹配的目录。

如果类属于某个包，它必须在与其包名匹配的目录中。无论是在开发位置还是在部署位置，类文件都必须如此。

(3) Java 代码中在包后使用圆点。

不管在 Java 代码中，还是在 URL 中，当您引用包中的类时，在包名和类名之间使用圆点，而非斜杠。

同样，可以不用自己编写测试 servlet，而是从 <http://www.coreservlets.com> 提供的本书的源代码档案文件中下载 `HelloServlet2.java`。由于这个 servlet 在 coreservlets 包中，因此，在开发过程中和部署到服务器时，都应该将它放在 coreservlets 目录中。如果发生编译错误，则应该回头检查 CLASSPATH 设定(2.7 节)——您极有可能会忘记包括“.”(当前目录)。编译完 `HelloServlet2.java` 后，找出服务器存储不属于定制 Web 应用 servlet 的目录(这个目录在默认的 Web 应用中通常是 WEB-INF/classes 目录)，将 `HelloServlet2.class` 放到该目录的 coreservlets 子目录中。可以检查服务器的文档找出这个位置，或者参见下面的列表，它汇总了 Tomcat，JRun 和 Resin 中的这个位置。现在，您可以简单地将类文件从开发目录复制到部署目录，2.9 节提供了简化这个过程的一些选项。

将 servlet 放置到正确的目录之后，使用 URL `http://localhost/servlet/coreservlets.HelloServlet2` 访问它。要注意，URL 中包名和 servlet 名之间使用的是圆点，不是斜杠。您看到的内容应该类似于图 2.13。如果此项测试失败，不是您输入了错误的 URL(例如，没有注意大小写)，就是将 `HelloServlet2.class` 放错了位置(例如，没有放入 coreservlets 子目录，而是直接放在服务器的 WEB-INF/classes 目录中)。

- Tomcat 中存储打包 Java 类的目录

`install_dir/webapps/ROOT/WEB-INF/classes/coreservlets`

- JRun 中存储打包 Java 类的目录

`install_dir/servers/default/default-ear/default-war/WEB-INF/classes/coreservlets`

- Resin 中存储打包 Java 类的目录
install_dir/doc/WEB-INF/classes/coreservlets
- 对应的 URL
http://host/servlet/coreservlets.HelloServlet2

清单2.4 coreservlets.HelloServlet2.java

```
package coreservlets;

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

/** Simple servlet for testing the use of packages.*/

public class HelloServlet2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>Hello (2)</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1>Hello (2)</H1>\n" +
                    "</BODY></HTML>");
    }
}
```

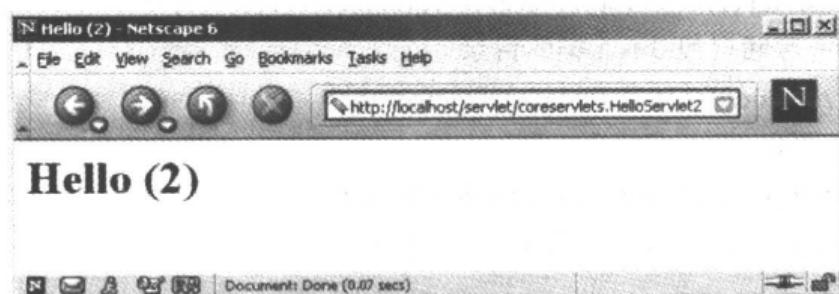


图 2.13 *http://localhost/servlet/coreservlets.HelloServlet2* 的结果
(注意, 包名和类名之间是圆点, 不是斜杠)

3. 测试 3: 使用包和实用工具类的 servlet

为检验服务器和开发环境的配置, 应该测试的最后的 servlet 既使用了包, 又使用了实用工具类。清单 2.5 列出了 HelloServlet3.java, 它使用 ServletUtilities 类(清单 2.6)简化 HTML 页面中 DOCTYPE(指定 HTML 的版本——用在使用 HTML 验证器的时候)和 HEAD(指定标题)部分的生成。页面的这两部分比较有用(实际上是技术上的需要), 但用 servlet 的 println 语句生成它们就显得冗长乏味。同样, 可以在 *http://www.coreservlets.com/* 找到它的源代码。

由于 servlet 和实用工具类都在 coreservlets 包中，因此，应该将它们都放在 coreservlets 目录中。如果发生编译错误，则应该回头检查 CLASSPATH 设定(2.7 节)——您极有可能忘记了包括顶层的开发目录。前面曾提到过，但我们还要再次强调：如果编译的包中使用了同一包中，或任何其他用户自定义(非系统)包中的其他类，必须保证 CLASSPATH 包括包层次的顶层目录。这项要求不仅限于 servlet；这是 Java 平台上包的通用工作方式。不过，许多 servlet 开发人员不知道这个情况，这是初级开发人员最常遇到的问题之一。此外，我们之后将会看到，如果您希望在 JSP 页面中使用自己编写的实用工具类，必须将它们都放入包内，因此，实际上您编写的所有辅助类(以及大多数 servlet)都应该放入包中。现在，您可能已经熟悉了使用包的过程。

警告

CLASSPATH 必须包括开发的顶层目录。否则，当试图编译的 servlet 属于包，且使用了包中用户定义的类时，会产生“unresolved symbol”错误消息。

编译完 HelloServlet3.java(会自动引起 ServletUtilities.java 的重新编译)之后，找出服务器存储不属于定制 Web 应用的 servlet 的目录(默认的 Web 应用中，这个目录通常是 WEB-INF/classes 目录)，将 HelloServlet3.class 和 ServletUtilities.class 放到它的 coreservlets 子目录中。可以检查服务器的文档找出这个位置，或者参见下面的列表，它汇总了 Tomcat, JRun 和 Resin 中使用的相关位置。之后，用 URL <http://localhost/servlet/coreservlets.HelloServlet3> 访问这个 servlet。您得到的内容应该与图 2.14 类似。

- Tomcat 中存储打包 Java 类的目录
install_dir/webapps/ROOT/WEB-INF/classes/coreservlets
- JRun 中存储打包 Java 类的目录
install_dir/servers/default/default-ear/default-war/WEB-INF/classes/coreservlets
- Resin 中存储打包 Java 类的目录
install_dir/doc/WEB-INF/classes/coreservlets
- 对应的 URL
<http://host/servlet/coreservlets.HelloServlet3>

清单 2.5 coreservlets/HelloServlet3.java

```
package coreservlets;

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

/** Simple servlet for testing the use of packages
 * and utilities from the same package.
 */

public class HelloServlet3 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
```

```

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Hello (3)";
        out.println(ServletUtilities.headWithTitle(title) +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1>" + title + "</H1>\n" +
                    "</BODY></HTML>");
    }
}

```

清单2.6 coreservlets/ServletUtilities.java(节选)

```

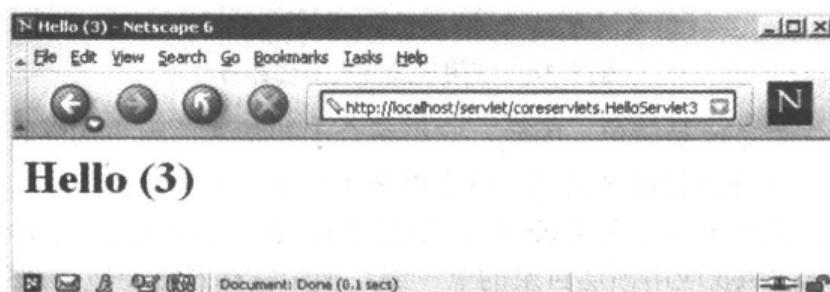
package coreservlets;

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

/** Some simple timesavers. Note that most are static methods.*/

public class ServletUtilities {
    public static final String DOCTYPE =
        "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
        "Transitional//EN\"";
    public static String headWithTitle(String title) {
        return(DOCUMENT + "\n" +
               "<HTML>\n" +
               "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
    }
    ...
}

```

图 2.14 *http://localhost/servlet/coreservlets.HelloServlet3* 的结果

2.9 实现简化的部署方法

现在您已经拥有了一个开发目录。您能够编译使用包或不使用包的 servlet。您知道 servlet 类应该放在哪个目录中。您知道用来访问它们的 URL(至少是默认的 URL; 在 2.11 节, 您将会了解到如何定制这个地址)。但是, 如何将类文件从开发目录移动到部署目录中呢? 每次都手动地复制每个文件十分乏味且易于出错。在您开始使用 Web 应用之后(参见 2.11 节), 复制单个的文件则更为棘手。

有几种方法可以简化这个过程。此处列出了最为流行的方法。如果您初次接触 servlet 和 JSP, 可能会选择第一个选项, 并在对开发过程熟练之前一直使用它。请注意, 我没有

列出将代码直接放置到服务器的开发目录这一选项。尽管它是初学者最常使用的选择之一，但由于它远不能满足高级任务的要求，因此我们推荐您在一开始就避免使用这种方法。

- (1) 复制成快捷方式或符号链接。
- (2) 使用 javac 的-d 选项。
- (3) 由 IDE 完成部署工作。
- (4) 使用 ant 或类似工具。

随后的小节给出这 4 种方案的细节。

2.9.1 复制成快捷方式或符号链接

在 Windows 中，转到服务器的默认 Web 应用，在 classes 目录上右击鼠标，选择 Copy(【复制】)。然后转到开发目录中，右击鼠标，选择 Paste Shortcut(【粘贴快捷方式】)(而非 Paste(【粘贴】))。现在，在编译完无包装的 servlet 后，只需将类文件拖放到快捷方式上。如果您在包中进行开发，可以使用鼠标的右键将整个目录(例如，coreservlets 目录)拖到快捷方式上，松开鼠标按键，选择 Copy(【复制】)。图 2.15 给出了相应的范例设置，它简化了本章的示例在 Tomcat、JRun 和 Resin 上的测试工作。在 Unix 上，可以将符号链接(用 ln -s 创建)用作 Windows 中的快捷方式。

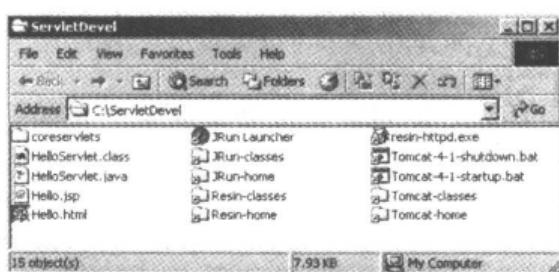


图 2.15 使用快捷方式简化部署

这种方式的优点之一是简单。因此，它很适合于初学者，因为初学者需要集中精力学习 servlet 和 JSP，不应该将过多的精力花在部署工具上。另一项优点是，通过对这种方案稍加变化，就能够适用于自定义 Web 应用(参见 2.11 节)。只需创建主 Web 应用目录的快捷方式(一般在默认 Web 应用顶层目录的上一级)，使用鼠标右键将包含您的 Web 应用的目录拖到这个快捷方式上，并选择 Copy(【复制】)，每次都复制整个 Web 应用。

这种方式的缺点之一是，在使用多个服务器的情况下需要重复复制。例如，我们在开发系统上安装 3 种不同的服务器(Tomcat、JRun 和 Resin)，需要定期在 3 种服务器上测试我们的代码，此时就需要重复复制。另一个缺点是，这种方式将 Java 源代码文件和类文件都复制到服务器中，但只有类文件是需要的。这在桌面服务器上可能不是什么问题，但是，在“真实”的部署服务器上，我们不希望包括源代码文件。

2.9.2 使用 javac 的-d 选项

默认情况下，Java 编译器(javac)将类文件放在产生它们的源代码文件所在的目录中。然而，javac 提供一个选项(-d)，我们可以用这个选项为类文件指定不同的位置。只需指定类文件的顶级目录——javac 会自动将打包后的类放到与包名匹配的子目录中。以 Tomcat

为例，可以用下面的方式编译 HelloServlet2(清单 2.4，2.8 节)(添加换行符是为了清晰起见，实际应用时要略去)。

```
javac -d install_dir/webapps/ROOT/WEB-INF/classes  
HelloServlet2.java
```

甚至可以编写 Windows 批处理文件或 Unix 外壳脚本或别名，使形如 `servletc` 的命令扩展成 `javac -d install_dir/.../classes`。可以在 <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javac.html> 得到-d 选项以及其他 javac 选项的详细信息。

这种方式的优点之一是它不需要手动地复制类文件。此外，由于 javac 会自动将类文件放到与包匹配的子目录中，不同包中的类可以使用相同的命令。

这种方式的主要缺点是它只适用于 Java 类文件；不能用它来部署 HTML 和 JSP 页面，更不用说整个 Web 应用。

2.9.3 由 IDE 完成部署工作

大多数专业的 servlet 和 JSP 开发环境(例如，IBM WebSphere Studio Application Developer, Sun ONE Studio, Borland JBuilder, Eclipse)都提供相应的选项，可以指定将项目的类文件部署到什么地方。之后，IDE 编译项目时，会自动将类文件部署到正确的位置(包专用的子目录和所有子目录)。

这种方式的优点之一是，至少在某些 IDE 中，它可以部署 HTML 和 JSP 页面，甚至整个 Web 应用，不仅仅是 Java 类文件。缺点为：它是 IDE 专有的技术，因而不能跨系统移植。

2.9.4 使用 ant 或类似工具

ant 是由 Apache 基金会开发的类似于 Unix make 实用程序的工具。然而，ant 用 Java 编程语言编写(从而也就具有了 Java 的可移植性)，并号称比 make 更易用，也更强大。许多 servlet 和 JSP 开发人员使用 ant 进行编译和部署。在 Tomcat 的用户和开发 Web 应用的人群中，ant 的使用尤为普遍(参见 2.11 节)。ant 的使用在本书第二卷中论述。

有关 ant 使用的一般信息，可以参见 <http://jakarta.apache.org/ant/manual/>。
<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/appdev/processes.html> 对 ant 在 Tomcat 中的应用给出了具体的指导。

这种方式的主要优点是灵活：ant 很强大，足以处理方方面面的任务，从编译 Java 源代码，复制文件，到生成 Web 档案(WAR)文件(参见 2.11 节)。ant 的缺点是学习它的使用比较困难；与本节中的其他技术相比，ant 的学习曲线比较陡峭。

2.10 默认 Web 应用的部署目录：汇总

下面的小节汇总了在 Apache Tomcat, Macromedia JRun 和 Caucho Resin 中部署和访问 HTML 页面、JSP 页面、servlet 和实用工具类的方式。这份汇总假定您在默认 Web 应用中部署文件，已经将端口号改为 80(参见 2.3 节)，并且通过默认的 URL(即 `http://host/servlet/ServletName`)访问 servlet。2.11 节说明如何部署用户自定义的 Web 应用，以及如何定制 URL。但您可能会希望从默认 Web 应用开始，来证实这一切确实能够正确工

作。附录给出了 Tomcat、JRun 和 Resin 中默认 Web 应用和定制 Web 应用所使用目录的统一汇总。

如果您使用的服务器就在您的桌面计算机上运行，那么可以在本节中每个 URL 中的 host 部分使用 localhost。

2.10.1 Tomcat

1. HTML 和 JSP 页面

- 主位置

install_dir/webapps/ROOT

- 对应的 URL

http://host/SomeFile.html

http://host/SomeFile.jsp

- 更具体的位置(任意子目录)

install_dir/webapps/ROOT/SomeDirectory

- 对应的 URL

http://host/SomeDirectory/SomeFile.html

http://host/SomeDirectory/SomeFile.jsp

2. 单个 servlet 和实用工具类文件

- 主位置(未打包的类)

install_dir/webapps/ROOT/WEB-INF/classes

- 对应的 URL(servlet)

http://host/servlet/ServletName

- 更具体的位置(包中的类)

install_dir/webapps/ROOT/WEB-INF/classes/packageName

- 对应的 URL(包中的 servlet)

http://host/servlet/packageName.ServletName

3. 捆绑在 JAR 文件中的 servlet 和实用工具类文件

- 位置

install_dir/webapps/ROOT/WEB-INF/lib

- 对应的 URL(servlet)

http://host/servlet/ServletName

http://host/servlet/packageName.ServletName

2.10.2 JRun

1. HTML 和 JSP 页面

- 主位置

install_dir/servers/default/default-ear/default-war

- 对应的 URL
http://host/SomeFile.html
http://host/SomeFile.jsp
- 更具体的位置(任意子目录)
install_dir/servers/default/default-ear/default-war/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.html
http://host/SomeDirectory/SomeFile.jsp

2. 单个 servlet 和实用工具类文件

- 主位置(未打包的类)
install_dir/servers/default/default-ear/default-war/WEB-INF/classes
- 对应的 URL(servlet)
http://host/servlet/ServletName
- 更具体的位置(包中的类)
install_dir/servers/default/default-ear/default-war/WEB-INF/classes/packageName
- 对应的 URL(包中的 servlet)
http://host/servlet/packageName.ServletName

3. 捆绑在 JAR 文件中的 servlet 和实用工具类文件

- 位置
install_dir/servers/default/default-ear/default-war/WEB-INF/lib
- 对应的 URL(servlet)
http://host/servlet/ServletName
http://host/servlet/packageName.ServletName

2.10.3 Resin

1. HTML 和 JSP 页面

- 主位置
install_dir/doc
- 对应的 URL
http://host/SomeFile.html
http://host/SomeFile.jsp
- 更具体的位置(任意子目录)
install_dir/doc/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.html
http://host/SomeDirectory/SomeFile.jsp

2. 单个 servlet 和实用工具类文件

- 主位置(未打包的类)
install_dir/doc/WEB-INF/classes
- 对应的 URL(servlet)
http://host/servlet/ServletName
- 更具体的位置(包中的类)
install_dir/doc/WEB-INF/classes/packageName
- 对应的 URL(包中的 servlet)
http://host/servlet/packageName.ServletName

3. 捆绑在 JAR 文件中的 servlet 和工具类文件

- 位置
install_dir/doc/WEB-INF/lib
- 对应的 URL(servlet)
http://host/servlet/ServletName
http://host/servlet/packageName.ServletName

2.11 Web 应用：预览

到此为止，我们都是用服务器的默认 Web 应用测试我们的 servlet。大多数服务器都会预先安装一个默认的 Web 应用，在大多数服务器中，可以用 *http://host/servlet/ServletName* 或 *http://host/servlet/packageName.ServletName* 形式的 URL，调用默认 Web 应用中的 servlet。在学习 servlet 的阶段，使用默认 Web 应用和 URL 十分方便；在您首次实践本书介绍的技术时，您可能只想使用这些默认的东西。因而，如果您初次涉足 servlet 和 JSP 开发，请先跳过这一节，等对这些技术熟悉之后，再来阅读本节的内容。

然而，在掌握了 servlet 和 JSP 的基本知识，并准备开发实际的应用时，您就不希望再使用默认的应用，而是想建立自己的 Web 应用。Web 应用在本书第二卷中详细论述，本节提供其基本情况的快速浏览。

核心方法

在初学时，请使用默认 Web 应用和默认的 servlet URL。对于正式的应用，则应使用定制的 Web 应用和 URL(在部署描述文件 web.xml 中指定)。

大多数服务器(包括本书中作为示例来使用的 3 种服务器)都有服务器专有的管理控制台，通过它可以从 Web 浏览器中创建和注册 Web 应用。第二卷将对这些控制台进行论述；现在，我们仅仅使用基本的手动方式，这种方式在所有的服务器上几乎都是相同的。下面的列表汇总这些步骤；随后的小节给出每个步骤的细节。

(1) 仿照默认 Web 应用的目录结构，创建一个目录。

HTML(还有 JSP)文档放在顶层目录中，web.xml 文件放在 WEB-INF 子目录，servlet 和其他类或者放在 WEB-INF/classes 中，或者放在 WEB-INF/classes 目录中与包名

匹配的子目录中。

(2) **更新 CLASSPATH。**

将 webAppDir/WEB-INF/classes 加入到 CLASSPATH 中。

(3) **将 Web 应用注册到服务器中。**

告知服务器 Web 应用的目录(或根据它创建的 JAR 文件)在什么位置, 调用这个应用时应该在 URL 中使用什么前缀(参见下一项)。例如, 在 Tomcat 中, 只需将 Web 应用的目录拖放到 *install_dir/webapps* 中, 之后重启服务器。目录的名称就是 Web 应用的前缀。

(4) **使用指定的 URL 前缀调用 Web 应用的 servlet 或 HTML/JSP 页面。**

对于没有打包的 servlet, 调用时用默认 URL *http://host/webAppPrefix/servlet/ServletName*, 打包的 servlet 用 *http://host/webAppPrefix/servlet/packageName.ServletName*, 位于 Web 应用顶层目录中的 HTML 页面用 *http://host/webAppPrefix/filename.html*。

(5) **为所有自己编写的 servlet 指定自定义的 URL。**

使用 web.xml 的 servlet 和 servlet-mapping 元素给每个 servlet 赋予形如 *http://host/webAppPrefix/someName* 的 URL。

2.11.1 创建 Web 应用的目录结构

要制作 Web 应用, 首先要在开发目录中创建一个目录。新目录应该遵循与默认 Web 应用相同的通用布局。

- HTML 以及 JSP 文档放在顶层目录中(或除 WEB-INF 之外的任何子目录中)。
- web.xml 文件(有时称为“部署描述文件”)放在 WEB-INF 子目录中。
- servlet 和其他类要么放在 WEB-INF/classes 中, 要么放在 WEB-INF/classes 中与包名匹配的子目录中(更为通用)。

制作这样一个目录, 最简单的方式是复制现有的 Web 应用, 比如在 Tomcat 中, 您可以将 ROOT 目录复制到您的开发目录, 将它重命名为 testApp, 从而得到类似 C:\Servlets+JSP\testApp 的目录。如同默认 Web 应用, 我们强烈建议不要在服务器的 Web 应用目录中直接进行开发工作。请在独立的目录中进行开发, 在准备好可以进行测试时, 再去部署它。最简单的部署是简单地将目录复制到服务器的标准位置, 2.9 节给出其他几种可选的方法。

2.11.2 更新 CLASSPATH

在 2.7 节中曾提到过, CLASSPATH 需要包含.class 文件的顶层目录。不管是否使用定制 Web 应用, 都必须如此, 因而要将 webAppDir/WEB-INF/classes 加入到 CLASSPATH 中。

2.11.3 将 Web 应用注册到服务器中

这一步告知服务器 Web 应用的目录(或根据它创建的 JAR 文件)在什么位置, 调用这个应用时应该在 URL 中使用什么前缀(参见下一小节)。不同的服务器提供各种专有的机制完

成这项注册，许多服务器使用交互式管理控制台。但是，在大多数服务器上，简单地将 Web 应用的目录拖进某个标准位置，之后重启服务器，也能够完成 Web 应用的注册。这种情况下，Web 应用目录的名称就是 URL 的前缀。下面列出本书使用的 3 种服务器中用于存储 Web 应用目录的标准位置。

- Tomcat 中 Web 应用的自动部署目录

install_dir/webapps

- JRun 中 Web 应用的自动部署目录

install_dir/servers/default

- Resin 中 Web 应用的自动部署目录

install_dir/webapps

例如，我们创建名为 testApp 的目录，它的结构如下：

- testApp/Hello.html

2.8 节中的 HTML 示例文件(清单 2.1)。

- testApp/Hello.jsp

2.8 节中的 JSP 示例文件(清单 2.2)。

- testApp/WEB-INF/classes/HelloServlet.class

2.8 节中无包装 servlet 的示例文件(清单 2.3)。

- testApp/WEB-INF/classes/coreservlets/HelloServlet2.class

2.8 节中第一个打包 servlet 的示例文件(清单 2.4)。

- testApp/WEB-INF/classes/coreservlets/HelloServlet3.class

2.8 节中第二个打包 servlet 的示例文件(清单 2.5)。

- testApp/WEB-INF/classes/coreservlets/ServletUtilities.class

HelloServlet3 使用的实用工具类(清单 2.6)。

WAR 文件

Web 档案(Web Archive)文件提供一种将 Web 应用捆绑到单个文件中的便利方式。使用单个大文件，相比许多小文件，更易于将 Web 应用在服务器之间转移。

WAR 文件实际上是以.war 为扩展名的 JAR 文件，我们使用通常的 jar 命令来创建它。例如，如果要将整个 Web 应用 testApp 捆绑到 WAR 文件 testApp2.war 中，我们只需要切换到 testApp 目录，并执行下面的命令。

```
jar cvf testApp2.war *
```

还有一些选项，可以用到高级应用中(我们在本书第二卷进行论述)，但对于简单的 WAR 文件来说，这就够了！

同样，精确的部署细节与服务器相关，但大多数服务器中，都可以简单地将 WAR 文件拖到自动部署目录中，WAR 文件的主文件名成为 Web 应用的前缀。例如，可以将 testApp2.war 拖到放置 testApp 的目录中，重启服务器，之后仅仅将 URL 中的 testApp 改为 testApp2，就可以调用用于测试的资源，如图 2.16 到图 2.20 所示。

2.11.4 使用 URL 前缀

在使用 Web 应用时，所有的 URL 中都要用到一个特殊的前缀。例如：

- 调用未打包的 servlet 用默认 URL:

http://host/webAppPrefix/servlet/ServletName

- 调用打包的 servlet 用:

http://host/webAppPrefix/servlet/packageName.ServletName

- 调用注册后的 servlet(参见下一小节)用:

http://host/webAppPrefix/customName

- 调用位于 Web 应用顶层目录的 HTML 页面用:

http://host/webAppPrefix/filename.html

- 调用子目录中的 HTML 页面用:

http://host/webAppPrefix/subdirectoryName/filename.html

- 和 HTML 页面放在同一目录中的 JSP 页面使用相同的方式进行调用(那些扩展名为.jsp 而不是.html 的文件除外)。

大多数 servlet 都允许我们自由地选择前缀，但是，默认情况下，目录名(或 WAR 文件的主文件名)是 Web 应用的前缀。例如，我们将 testApp 目录复制到相应的 Web 应用目录(Tomcat 和 Resin 中为 *install_dir/webapps*, JRun 中为 *install_dir/servers/default*)中，并重启服务器。之后，我们就可以使用除了在主机名之后加入 testApp 之外，其他部分与 2.8 节中完全相同的 URL 来访问这些资源。参见图 2.16 至图 2.20。

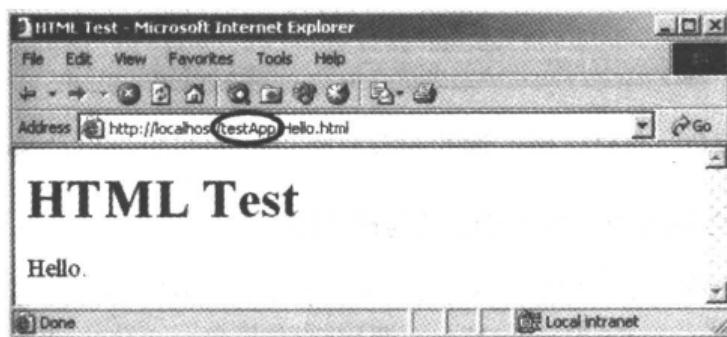


图 2.16 调用 Web 应用中的 Hello.html

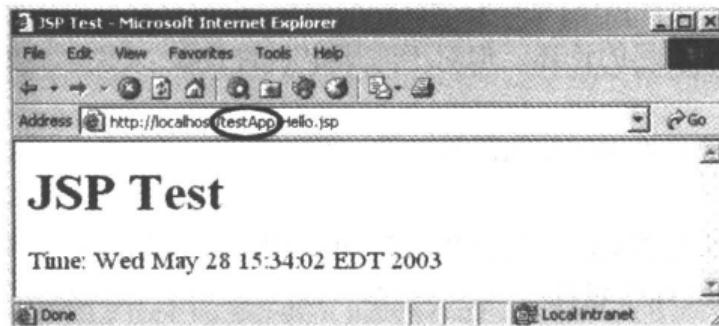


图 2.17 调用 Web 应用中的 Hello.jsp

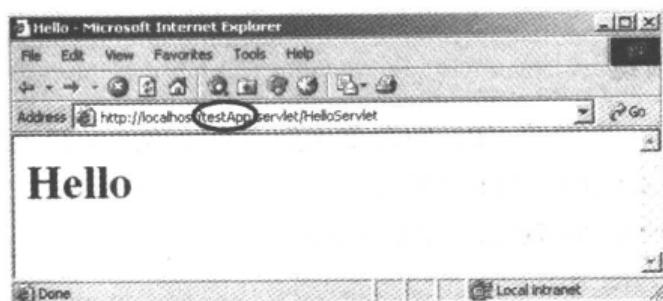


图 2.18 用默认 URL 调用 Web 应用中的 HelloServlet.class

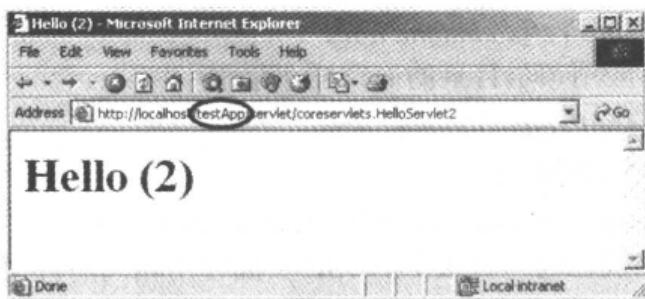


图 2.19 用默认 URL 调用 Web 应用中的 HelloServlet2.class

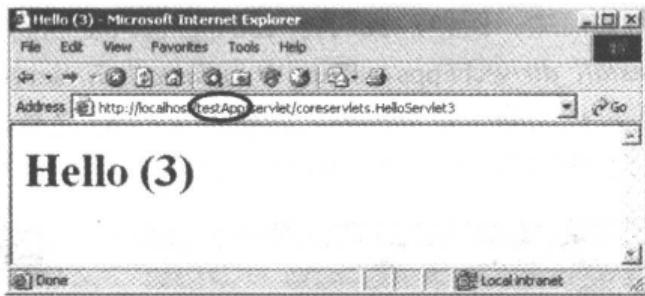


图 2.20 用默认 URL 调用 Web 应用中的 HelloServlet3.class

2.11.5 为自己编写的 servlet 分配定制的 URL

在最初的开发过程中，采用将 servlet 拖到 WEB-INF/classes 目录中，并立即用 `http://host/webAppPrefix/servlet/ServletName` 调用它的方式，十分方便。然而，对于重大应用的部署，我们总是希望自己定义 URL。

web.xml(部署描述文件)中的 servlet 和 servlet-mapping 元素可以用来指定 URL。本书第二卷将对 web.xml 进行详尽的论述，但对于注册定制的 URL 来说，我们只需知道 5 点：

- **文件的位置**
该文件总是放置在 WEB-INF 中。
- **基本格式**
该文件的开头是 XML 标头和 DOCTYPE 声明，并且含有一个 web-app 元素。
- **如何为 servlet 赋予名称**
使用 servlet 及其子元素 servlet-name 和 servlet-class。
- **如何为已赋予名称的 servlet 指定 URL**
使用 servlet-mapping 及其子元素 servlet-name 和 url-pattern。

- 何时读取 web.xml 文件的内容

服务器只在启动时读取 web.xml 文件。

1. 部署描述文件的位置

web.xml 文件总是放置在 Web 应用的 WEB-INF 目录中。这是各个服务器之间惟一通用的位置；其他位置(例如 Tomcat 中的 *install_dir/conf*)都是非标准的服务器扩展，应该避免使用它们。

2. 基本格式的定义

与 servlet 2.3(JSP 1.2)和 servlet 2.4(JSP 2.0)都兼容的 web.xml 文件具有如下的基本格式：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

</web-app>
```

专门用于 servlet 2.4(JSP 2.0)的部署描述文件在本书第二卷论述。

3. servlet 的命名

命名 servlet，我们需要用到 web-app 内 servlet 元素的 servlet-name(可以选取任意名称)和 servlet-class(完全限定类名)子元素。例如，将 Servlet2 赋予 HelloServlet2 的做法如下：

```
<servlet>
  <servlet-name>Servlet2</servlet-name>
  <servlet-class>coreservlets.HelloServlet2</servlet-class>
</servlet>
```

4. URL 的指定

要将 URL 赋予已命名的 servlet，需要使用 servlet-mapping 元素的 servlet-name(之前指定的名称)和 url-pattern(URL 前缀，由斜杠开始)子元素。例如，要将 URL *http://host/webAppPrefix/servlet2* 赋予已命名为 Servlet2 的 servlet，做法如下：

```
<servlet-mapping>
  <servlet-name>Servlet2</servlet-name>
  <url-pattern>/servlet2</url-pattern>
</servlet-mapping>
```

要注意，必须将所有的 servlet 元素放置在任何 servlet-mapping 元素之前，它们不能相互混合。

5. 部署描述文件的读取

许多服务器具有“热部署”能力或方法，可以交互式地重启 Web 应用。例如，JRun 会自动重启那些 web.xml 文件发生改变的 Web 应用。然而。默认情况下，服务器启动后 web.xml 文件是只读的。因此，除非您使用服务器专有的特性，否则每次修改完 web.xml 文件还是得重新启动服务器。

6. 示例

清单 2.7 给出 Web 应用 testApp 中完整的 web.xml 文件。这个文件放置在 testApp 的 WEB-INF 目录中, testApp 目录已复制到服务器的 Web 应用目录中(例如 Tomcat 和 Resin 中为 *install_dir/webapps*, JRun 中为 *install_dir/servers/default*), 服务器也已经重新启动过。图 2.21 到图 2.23 展示出, 应用注册的 URL 对这 3 个示例 servlet 的调用。

清单 2.7 WEB-INF/web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>Servlet1</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Servlet2</servlet-name>
    <servlet-class>coreservlets.HelloServlet2</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Servlet3</servlet-name>
    <servlet-class>coreservlets.HelloServlet3</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servlet1</servlet-name>
    <url-pattern>/servlet1</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Servlet2</servlet-name>
    <url-pattern>/servlet2</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Servlet3</servlet-name>
    <url-pattern>/servlet3</url-pattern>
  </servlet-mapping>
</web-app>
```



图 2.21 用定制 URL 调用 HelloServlet

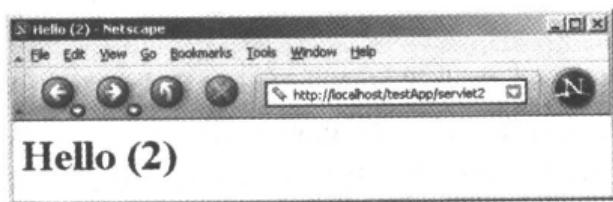


图 2.22 用定制 URL 调用 HelloServlet2



图 2.23 用定制 URL 调用 HelloServlet3

第3章 servlet 基础

本章的主题：

- servlet 的基本结构
- 生成纯文本的简单 servlet
- 生成 HTML 的 servlet
- servlet 和包
- 协助构建 HTML 的一些实用工具类
- servlet 的生命周期
- 多线程问题的处理
- 与 servlet 进行交互式对话的工具
- servlet 的调试策略

如第 1 章所述，servlet 是运行在 Web 服务器或应用服务器上的程序；它担当 Web 浏览器或其他 HTTP 客户程序发出的请求，与 HTTP 服务器上的数据库或应用程序之间的中间层。它们的工作是执行图 3.1 所示的任务。

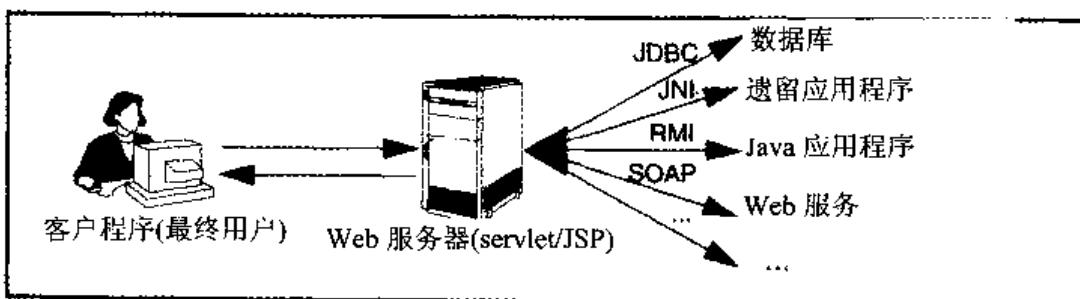


图 3.1 Web 中间件的作用

(1) 读取客户程序发送的显式数据。

最终用户一般在 Web 页面上的 HTML 表单中输入这类数据。然而，这类数据也可能来自于 applet 或定制的 HTTP 客户程序。

(2) 读取浏览器发出的隐式 HTTP 请求数据。

图 3.1 中有一条从客户到 Web 服务器(servlet 和 JSP 页面在这一层执行)的单向箭头，但实际上存在两种类型的数据：最终用户在表单中输入的显式数据和后台的 HTTP 信息。两种类型的数据对开发工作都至关重要。HTTP 信息包括 cookie、媒体类型和浏览器能够识别的压缩模式等。这部分内容在第 5 章进行论述。

(3) 生成结果。

这个过程可能需要与数据库进行对话、执行 RMI 或 CORBA 调用、调用 Web 服务或直接计算得出响应。实际的数据可能在关系型数据库中。这没有什么问题，但您的数据库可能不理解 HTTP，或者不能以 HTML 的形式返回结果，故而 Web 浏览器不能直接与数据库对话。大多数其他应用程序也是如此。您需要 Web 中间

层从 HTTP 流中解析出输入数据，与应用程序进行对话，并将结果嵌入到文档中。

(4) 向客户发送显式数据(即文档)。

这种文档可以用各种不同的形式发送，包括文本(HTML 或 XML)、二进制(GIF 图像)、Excel，甚至是层叠在其他底层格式上的压缩格式，如 gzip。

(5) 发送隐式的 HTTP 响应数据。

图 3.1 中有一条从 Web 中间层(servlet 或 JSP 页面)到客户的单向箭头，但实际发送的数据有两种：文档本身、以及后台的 HTTP 信息。两种类型的数据对开发工作都至关重要。发送 HTTP 响应数据时，需要告知浏览器或其他客户程序返回文档的类型(如 HTML)、设置 cookie 并缓存参数、以及其他类似任务。这些任务在第 6 章~第 8 章中进行介绍。

原则上，servlet 并不仅仅限于处理 HTTP 请求的 Web 服务器或应用服务器，它同样可以用于其他类型的服务器。例如，servlet 可以嵌入到 FTP 或邮件服务器中，扩展它们的功能。然而，实际上这种用法并不多见，本书中我们只论述 HTTP servlet。

3.1 servlet 的基本结构

清单 3.1 给出一个基本的 servlet，它处理 GET 请求。GET 请求是浏览器请求的常见类型，用来请求 Web 页面。用户在地址栏中输入 URL、点击 Web 页面内的链接、或提交没有指定 METHOD 或指定 METHOD="GET" 的 HTML 表单时，浏览器都会生成这个请求。servlet 还可以容易地处理 POST 请求(提交 METHOD="POST" 的 HTML 表单时，会生成 POST 请求)。HTML 表单的使用细节以及 GET 和 POST 之间的区别，请参见第 19 章。

清单3.1 ServletTemplate.java

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

public class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        // Use "request" to read incoming HTTP headers
        // (e.g., cookies) and query data from HTML forms.

        // Use "response" to specify the HTTP response status
        // code and headers (e.g., the content type, cookies).

        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser
    }
}
```

servlet 一般扩展 HttpServlet，并依数据发送方式的不同(GET 或 POST)，覆盖 doGet 或 doPost 方法。如果希望 servlet 对 GET 和 POST 请求采用同样的行动，只需要让 doGet 调用 doPost，反之亦然。

`doGet` 和 `doPost` 都接受两个参数: `HttpServletRequest` 和 `HttpServletResponse`。通过 `HttpServletRequest`, 可以得到所有的输入数据; 这个类提供相应的方法, 通过这些方法可以找出诸如表单(查询)数据、HTTP 请求报头和客户的主机名等信息。通过 `HttpServletResponse` 可以指定输出信息, 比如 HTTP 状态代码(200, 404 等)和响应报头(`Content-Type`, `Set-Cookie` 等)。最重要的是, 通过 `HttpServletResponse` 可以获得 `PrintWriter`, 用它可以将文档内容发送给客户。对于简单的 servlet, 大部分工作都花在用 `println` 语句生成期望的页面上。表单数据、HTTP 请求报头、HTTP 响应和 cookie, 这些内容在随后的章节中进行论述。

由于 `doGet` 和 `doPost` 抛出两种异常(`ServletException` 和 `IOException`), 所以必须在方法声明中包括它们。最后, 您还必须导入 `java.io(PrintWriter 等)`、`javax.servlet(HttpServletRequest 等)` 和 `javax.servlet.http(HttpServletRequest 和 HttpServletResponse)` 中的类。

然而, 并不需要记住方法的签名以及这些输入语句。只需下载 <http://www.coreservlets.com> 上源代码档案文件中的模板, 并从它开始编写自己的 servlet。

3.2 生成纯文本的 servlet

清单 3.2 列出一个输出纯文本的简单 servlet, 图 3.2 是它的输出。在我们继续讲述下面的内容之前, 花一些时间回顾一下安装、编译和运行这个简单 servlet 的过程是值得的。第 2 章对这一过程做了更为详尽的描述。

第一, 一定要检验下述基本情况:

- 服务器已按照 2.3 节的描述, 正确地安装。
- 开发环境中的 CLASSPATH 指出了必需的 3 项(servlet JAR 文件、顶层的开发目录和“.”), 如 2.7 节所述。
- 2.8 节列出的所有测试用例都能够成功执行。

第二, 输入“`javac HelloWorld.java`”, 或让开发环境编译这个 servlet(例如, 单击 IDE 的 Build 按钮, 或从 emacs 的 JDE 菜单中选择 Compile)。这一步将编译您的 servlet, 创建 `HelloWorld.class`。

第三, 将 `HelloWorld.class` 移动到服务器中默认 Web 应用存储 servlet 的目录。不同的服务器, 这个位置也会有所不同, 但一般的形式是 `install_dir/.../WEB-INF/classes`(详细情况参见 2.10 节)。Tomcat 中, 我们使用 `install_dir/webapps/ROOT/WEB-INF/classes`; JRun 中为 `install_dir/servers/default/default-ear/default-war/WEB-INF/classes`; Resin 中为 `install_dir/doc/WEB-INF/classes`。同时, 可以使用 2.9 节中介绍的技术, 自动将类文件放置到合适的位置。

最后是调用这个 servlet。最后这一步既可以使用默认 URL `http://host/servlet/ServletName`, 也可以使用 `web.xml` 文件中定义的定制 URL, 如 2.11 节所述。在最初的开发过程中, 几乎可以肯定, 使用默认 URL 比较方便, 这样每次测试新的 servlet 时可以不用编辑 `web.xml`。然而, 在部署实际的应用时, 一般总要禁用默认 URL, 并在 `web.xml` 中指定明确的 URL(参见 2.11 节)。实际上, 服务器并非必需支持默认 URL, 几种服务器, 最为突出的就是 BEA WebLogic, 就不支持默认 URL。

图 3.2 展示出用默认 URL 访问 servlet 的情况，服务器在本地计算机上运行。

清单3.2 HelloWorld.java

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

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```



图 3.2 http://localhost/servlet>HelloWorld 的结果

3.3 生成 HTML 的 servlet

大多数 servlet 生成 HTML，而非前述例子中的纯文本。要生成 HTML，需要在刚才介绍的过程中加入下面 3 步：

- (1) 告知浏览器，即将向它发送 HTML。
- (2) 修改 println 语句，构建合法的 Web 页面。
- (3) 用形式语法验证器(formal syntax validator)检查生成的 HTML。

第一步通过将 HTTP Content-Type 响应报头设为 text/html 来完成。一般而言，报头使用 HttpServletResponse 的 setHeader 方法来设置，但由于设置内容的类型是一项十分常见的任务，因而，HttpServletResponse 提供特殊的 setContentType 方法，专门用于这种目的。指明 HTML 的方式是使用 text/html 类型，因此，代码应该如下：

```
response.setContentType("text/html");
```

尽管 HTML 是 servlet 创建的最常见的文档类型，但是，servlet 创建其他类型文档的情况也很多见。例如，使用 servlet 生成 Excel 表格(内容类型 application/vnd.ms-excel——参见 7.3 节)、JPEG 图像(内容类型 image/jpeg——参见 7.5 节)和 XML 文档(内容类型 text/xml)的情况也十分常见。同时，一般很少使用 servlet 生成格式相对固定的 HTML 页面(即每次请求，页面的布局改动很小)；这种情况下 JSP 常常更为方便。JSP 在本书第 II 部分进行论述(从第 10 章开始)。

如果您尚不熟悉 HTTP 响应报头，不要担心；第 7 章将对它们进行论述。然而，您现在就应该注意：需要在 PrintWriter 实际返回任何内容之前，设置响应报头。这是由于 HTTP 响应由状态行、一个或多个报头、一个空行和实际的文档以此次序构成。报头的出现次序并不重要，servlet 会缓冲报头数据，将它们一次发送到客户端，因此，即使在设定报头之后，依旧可以设置状态代码(属于返回内容的第一行)。但是，servlet 不是一定要缓冲文档本身，因为对于篇幅较长的页面，用户或许只希望看到部分结果。servlet 引擎可以缓冲部分输出，但并未规定缓冲区的大小。可以使用 HttpServletResponse 的 getBufferSize 方法确定这个大小，或使用 setBufferSize 指定这个大小。也可以在缓冲区填满，要发往客户时，对报头进行设置。如果不确定缓冲区是否已经发送出去，可以使用 isCommitted 方法来检查。即使如此，最佳的方案还是将 setContentType 行放在任何使用 PrintWriter 的行之前。

警告

必须在传送实际的文档之前设定内容的类型。

在编写构建 HTML 文档的 servlet 时，第二步是用 `println` 语句输出 HTML(不再是纯文本)。清单 3.3 列出了 `HelloServlet.java`，在 2.8 节中，我们曾用它来验证服务器是否运转正常。如图 3.3 所示，浏览器按照 HTML 来格式化得到的结果，而非按照纯文本。

清单3.3 HelloServlet.java

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

/** Simple servlet used to test server.*/

public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" + "
            " \"Transitional//EN\\\">\\n";
        out.println(docType +
                    "<HTML\\n\" + "
                    "<HEAD><TITLE>Hello</TITLE></HEAD>\\n\" + "
                    "<BODY BGCOLOR=\"#FDF5E6\\\">\\n\" + "
                    "<H1>Hello</H1>\\n\" + "
                    "</BODY></HTML>\"");
    }
}
```

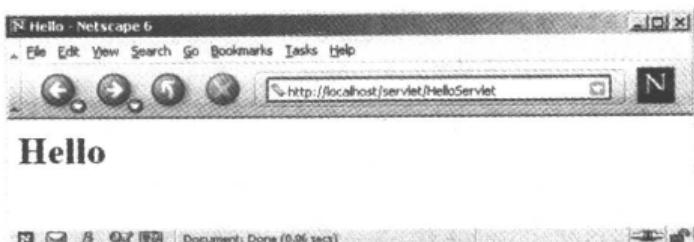


图 3.3 <http://localhost/servlet>HelloServlet> 的结果

最后一步是核实生成的 HTML 中，不存在有可能会在不同的浏览器上引起不可预期结果的语法错误。参见 3.5 节中有关 HTML 验证器的论述。

3.4 servlet 的打包

在产品开发过程中，多个程序员可能为同一服务器开发 servlet。因此，将所有的 servlet 放到同一目录中将会生成数目庞大且难以管理的类，如果两个开发人员为 servlet 或实用工具类命名时，不经意间选择了相同的名称，还会导致命名冲突。现在，通过 Web 应用(参见 2.11 节)，可以将内容划分到多个单独的目录中，每个目录有自己的一套 servlet、实用工具类、JSP 页面和 HTML 文件，避免了这个问题。然而，由于单个 Web 应用也可能比较庞大，因此，我们依旧需要采用 Java 中用以避免命名冲突的标准解决方案：包。此外，后面您将会看到，JSP 页面使用的定制类一定要放在包中。或许您早已养成了这种习惯。

在将 servlet 放到包中时，需要执行下面两个额外的步骤。

(1) 将文件放到与预定的包名匹配的子目录中。

例如，对于本书中接下来出现的 servlet，我们都使用 coreservlets 包。因此，类文件需要放置在名为 coreservlets 的子目录中。不管使用什么样的操作系统，都要注意包名和目录名的大小写。

(2) 在类文件中插入包语句。

例如，如果想将类放到名为 somePackage 的包中，那么该类必须在 somePackage 目录中，并且文件中第一行非注释语句应该为：

```
package somePackage;
```

例如，清单 3.4 给出 HelloServlet 类的一个变体，它在 coreservlets 包中，相应地，位于 coreservlets 目录中。按照 2.8 节讲述的内容，对于 Tomcat，这个类文件应该放在 *install_dir/webapps/ROOT/WEB-INF/classes/coreservlets* 中； JRun 中为 *install_dir/servers/default/default-ear/default-war/WEB-INF/classes/coreservlets* ； Resin 中为 *install_dir/doc/WEB-INF/classes/coreservlets*。其他服务器的安装位置也基本类似。

图 3.4 展示出用默认 URL 访问这个 servlet 的情况。

清单3.4 coreservlets/HelloServlet2.java

```
package coreservlets;
import java.io.*;
```

```

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

/** Simple servlet for testing the use of packages.*/

public class HelloServlet2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>Hello (2)</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1>Hello (2)</H1>\n" +
                    "</BODY></HTML>");
    }
}

```

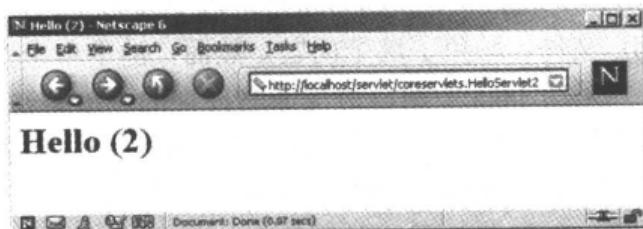


图 3.4 <http://localhost/servlet/coreservlets.HelloServlet2> 的结果

3.5 简单的 HTML 构建工具

您可能早已了解, HTML 文档的结构如下:

```

<!DOCTYPE ...>
<HTML>
<HEAD><TITLE>...</TITLE>...</HEAD>
<BODY ...>...</BODY>
</HTML>

```

在使用 servlet 构建 HTML 时, 您可能会略去这个结构的某些部分, 尤其是 DOCTYPE 行, 因为虽然 HTML 规范需要它, 但几乎所有主流的浏览器都忽略它。我们极不赞成这种做法。DOCTYPE 行的长处是, 它告诉 HTML 验证器您使用的是 HTML 的哪个版本, 从而验证器知道应该按照哪种规范对文档进行检查。这些验证器对调试很有价值, 能够帮助您捕获那些您的浏览器可以推测出来, 但其他浏览器在显示时可能会有困难的 HTML 语法错误。

由万维网联盟(<http://validator.w3.org/>)和 Web Design Group(<http://www.htmlhelp.com/tools/validator>)提供的诸多验证器应用最为广泛。您可以向它们提交一个 URL, 之后, 它们会读取相应的页面, 根据正式的 HTML 规范检查页面的语法, 并将发现的任何错误报告给

您。由于从客户的角度，生成 HTML 的 servlet 和常规 Web 页面没有什么不同，因此，可以用正常的方式对它进行验证，除非它返回结果时需要用到 POST 数据。由于 GET 数据就附加在 URL 上，我们甚至可以向验证器发送包括 GET 数据的 URL。如果 servlet 只能在公司防火墙的内部访问，可以运行它，将 HTML 保存到磁盘，然后使用验证器的文件上载选项。

核心方法

使用 HTML 验证器检查由您的 servlet 所生成的页面的语法。

毫无疑问，用 `println` 语句生成 HTML 有些笨重，尤其是那些冗长乏味的行，如 DOCTYPE 声明。有些人编写很长的 HTML 生成实用工具程序，然后，在编写 servlet 时使用这些实用工具程序，以此来解决这个问题。我们对这类扩展库的有效性持怀疑态度。首先且最重要的是，编程生成 HTML 的不便使 JSP(参见第 10 章)解决的主要问题之一。其次，用来生成 HTML 的例程可能十分笨重，一般并不支持所有的 HTML 属性(样式表的 CLASS 和 ID，JavaScript 事件处理器，表格单元的背景色等)。

尽管全功能 HTML 生成库的价值尚存在疑问，但是，如果您发现自己多次重复相同的构造，同样可以创建简单的实用工具类，简化这些构造。毕竟，您使用的是 Java 编程语言；不要忘记标准的面向对象编程原则：重用代码，而不要重复代码。重复相同的或近乎相同的代码意味着，在您所采用的方式发生变化时(这几乎是不可避免的)，必须更改许多位于不同位置的代码。

标准 servlet 中，Web 页面中的两部分内容(DOCTYPE 和 HEAD)一般不会发生改变，因而可以归结到一个简单的实用工具文件中。清单 3.5 就是这样一个文件，清单 3.6 列出 HelloServlet 类的又一变体，它使用这个实用工具类。贯穿本书，我们还将加入另外几个实用工具类。

```
清单3.5 coreservlets/ServletUtilities.java

package coreservlets;

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

/** Some simple timesavers. Note that most are static methods. */

public class ServletUtilities {
    public static final String DOCTYPE =
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +\n" +
        "Transitional//EN\">";

    public static String headWithTitle(String title) {
        return(DOCUMENT + "\n" +
            "<HTML>\n" +
            "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
    }
    ...
}
```

清单 3.6 coreservlets/HelloServlet3.java

```

package coreservlets;

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

/** Simple servlet for testing the use of packages
 * and utilities from the same package.
 */

public class HelloServlet3 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Hello (3)";
        out.println(ServletUtilities.headWithTitle(title) +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1>" + title + "</H1>\n" +
                    "</BODY></HTML>");
    }
}

```

编译完 HelloServlet3.java 之后(会导致自动编译 ServletUtilities.java)，需要将这两个类文件移动到服务器默认部署位置(.../WEB-INF/classes；详细信息请回顾 2.8 节)中的 coreservlets 子目录中。如果在编译 HelloServlet3.java 时产生“Unresolved symbol”错误，请返回检查 2.7 节中介绍的 CLASSPATH 设定，尤其是有关在 CLASSPATH 中包括顶层开发目录的部分。图 3.5 展示出用默认 URL 调用该 servlet 的结果。

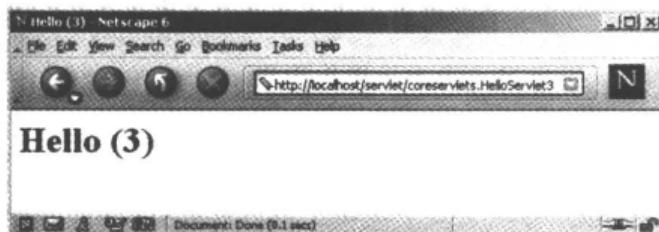


图 3.5 http://localhost/servlet/coreservlets.HelloServlet3 的结果

3.6 servlet 的生命周期

在 1.4 节中，我们提到过，服务器只创建每个 servlet 的单一实例，每个用户请求都会引发新的线程——将用户请求交付给相应的 doGet 或 doPost 进行处理。现在，我们更为详细地介绍 servlet 的创建和销毁方式，以及调用各种方法的方式和时间。我们首先在此进行汇总，然后在随后的小节中详细阐述。

首次创建 servlet 时，它的 init 方法会得到调用，因此，init 是放置一次性设置代码的地方。在这之后，针对每个用户请求，都会创建一个线程，该线程调用前面创建的实例的 service 方法。多个并发请求一般会导致多个线程同时调用 service(尽管可以实现特殊的接口

(SingleThreadModel)——规定任何时间只允许单个线程运行)。之后，由 service 方法依据接收到的 HTTP 请求的类型，调用 doGet, doPost, 或其他 doXxx 方法。最后，如果服务器决定卸载某个 servlet，它会首先调用 servlet 的 destroy 方法。

3.6.1 service 方法

服务器每次接收到对 servlet 的请求，都会产生一个新的线程，调用 service 方法。service 方法检查 HTTP 请求的类型(GET, POST, PUT, DELETE 等)并相应地调用 doGet, doPost, doPut, doDelete 等方法。GET 请求起因于正常的 URL 请求，或没有指定 METHOD 的 HTML 表单。POST 请求起因于特别将 POST 列为 METHOD 的 HTML 表单。其他 HTTP 请求都由定制客户生成。如果您不熟悉 HTML 表单，请参见第 19 章。

如果您需要在 servlet 中等同地处理 POST 和 GET 请求，您可能会倾向于不去实现 doGet 和 doPost 方法，而是直接覆盖 service 方法。这不是一个好的思想。取而代之，只需让 doPost 调用 doGet(或相反)即可，如下所示。

```
public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet code
}

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

虽然这种方式要多出几行代码，但相对于直接覆盖 service，它有好几项优点。首先，之后(或许在子类中)您还可以加入 doPut, doTrace 等，支持其他 HTTP 请求方法。直接覆盖 service 则排除了这种可能性。其次，您还可以通过添加 getLastModified 方法，加入对修改日期的支持，如清单 3.7 所示。由于 getLastModified 由默认的 service 方法调用，所以覆盖 service 方法也就失去了这个选项。最后，service 提供对 HEAD, OPTION 和 TRACE 请求的自动支持。

核心方法

如果您的 servlet 需要等同地处理 GET 和 POST，可以让 doPost 方法调用 doGet，或者相反。不要覆盖 service 方法。

3.6.2 doGet, doPost 和 doXxx 方法

这些方法才是 servlet 的主体。99%的时间，您只关心 GET 和 POST 请求，因而，您可以覆盖 doGet 和/或 doPost。如果愿意，也可以覆盖 DELETE 请求的 doDelete、PUT 请求的 doPut、OPTIONS 请求的 doOptions 以及 TRACE 请求的 doTrace。然而，要记住您已经拥有对 OPTIONS 和 TRACE 的自动支持。

通常情况下，不需实现 doHead 以处理 HEAD 请求(HEAD 请求规定，服务器应该只返

回正常的 HTTP 报头，不含与之相关联的文档)。由于系统会自动调用 `doGet`，并用生成的状态行和报头设定来应答 HEAD 请求，故而，一般不需要实现 `doHead`。有时，为了能够更快地生成对 HEAD 请求的响应(例如来自定制客户的请求，只需要 HTTP 报头，不需构建实际的文档输出)，会实现 `doHead` 方法。

3.6.3 init 方法

大多数时候，您的 servlet 只需处理单个请求的数据，`doGet` 或 `doPost` 是生命周期中惟一需要的方法。然而，有时候您希望在 servlet 首次载入时，执行复杂的初始化任务，但并不想每个请求都重复这些任务。`init` 方法就专为这种情况设计；它在 servlet 初次创建时被调用，之后处理每个用户的请求时，则不再调用这个方法。因此，它主要用于一次性的初始化，和 applet 的 `init` 方法相同。servlet 一般在用户首次调用对应 servlet 的 URL 时创建，但也可以指定 servlet 在服务器启动后载入(参见本书第二卷中关于 `web.xml` 文件的章节)。

`init` 方法的定义如下：

```
public void init() throws ServletException {
    // Initialization code...
}
```

`init` 方法执行两种类型的初始化：常规初始化，以及由初始化参数控制的初始化。

1. 常规初始化

第一种类型的初始化中，`init` 只是创建或载入在 servlet 生命期内用到的一些数据，或者执行某些一次性的计算。如果您熟悉 applet，这项任务类似于 applet 调用 `getImage` 通过网络载入图像文件：这个操作只需执行一次，故而由 `init` 触发。本书给出的 servlet 示例中，`init` 完成的初始化任务既有为即将处理的请求建立数据库连接共享，也有将数据文件载入 `HashMap`。

清单 3.7 列出的 servlet 使用 `init` 完成两件事。

首先，它构建一个由 10 个整数组成的数组。由于这些数字都来源于复杂的计算，我们不希望每次请求都重复这些计算。因此，`doGet` 查找 `init` 中计算得出的值，而不去每次都生成它们。这项技术的结果参见图 3.6。

其次，由于这个 servlet 的输出不会改变——除非服务器重新启动，因此，`init` 还存储了一个页面修改日期，由 `getLastModified` 方法使用。这个方法应该返回以毫秒表达的修改时间，起始时间为 1970 年，这是标准的 Java 日期。这个时间将被自动转换成适用于 `Last-Modified` 报头的 GMT 日期。更重要的是，如果服务器接收到条件 GET 请求(指定客户只想要那些自特定日期以来发生改变的页面)，系统会将指定的日期与由 `getLastModified` 返回的日期进行比较，只返回那些在指定的日期之后发生改变的页面。浏览器常常会针对存储在缓存中的页面生成这些条件化请求，因此，支持条件请求可以帮助用户(他们能够更快得到结果)、降低服务器的负载(您可以发送更少的完整文档)。由于 `Last-Modified` 和 `If-Modified-Since` 报头只使用整秒，因此 `getLastModified` 方法应该舍到整数秒。

清单 3.7 coreservlets/LotteryNumbers.java

```
package coreservlets;
```

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

/** Example using servlet initialization and the
 * getLastModified method.
 */

public class LotteryNumbers extends HttpServlet {
    private long modTime;
    private int[] numbers = new int[10];

    /** The init method is called only when the servlet
     * is first loaded, before the first request is processed.
     */

    public void init() throws ServletException {
        // Round to nearest second (i.e., 1000 milliseconds)
        modTime = System.currentTimeMillis()/1000*1000;
        for(int i=0; i<numbers.length; i++) {
            numbers[i] = randomNum();
        }
    }

    /** Return the list of numbers that init computed. */
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Your Lottery Numbers";
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    "<B>Based upon extensive research of " +
                    "astro-illogical trends, psychic farces, " +
                    "and detailed statistical claptrap, " +
                    "we have chosen the " + numbers.length +
                    " best lottery numbers for you.</B>" +
                    "<OL>");
        for(int i=0; i<numbers.length; i++) {
            out.println(" <LI>" + numbers[i]);
        }
        out.println("</OL>" +
                   "</BODY></HTML>");
    }

    /** The standard service method compares this date against
     * any date specified in the If-Modified-Since request header.
     * If the getLastModified date is later or if there is no
     * If-Modified-Since header, the doGet method is called
    
```

```

* normally. But if the getLastModified date is the same or
* earlier, the service method sends back a 304 (Not Modified)
* response and does <B>not</B> call doGet. The browser should
* use its cached version of the page in such a case.
*/
public long getLastModified(HttpServletRequest request) {
    return(modTime);
}

// A random int from 0 to 99.

private int randomNum() {
    return((int)(Math.random() * 100));
}
}

```

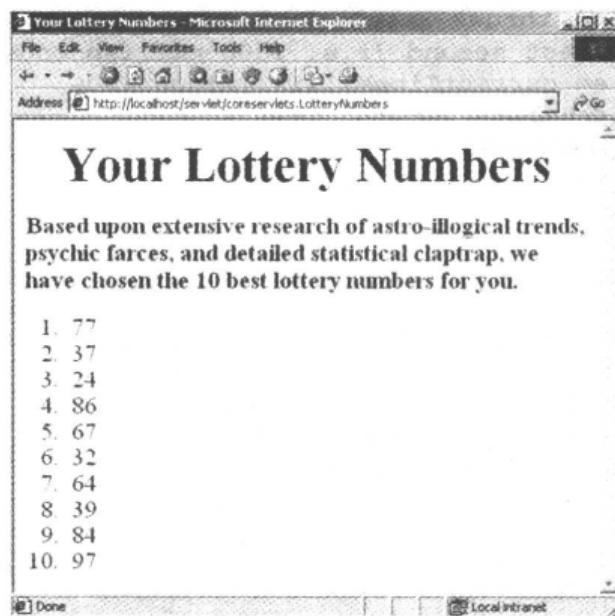


图 3.6 LotteryNumbers servlet 的结果

图 3.7 和图 3.8 给出用稍有不同的 If-Modified-Since 日期请求同一 servlet 得到的结果。为了设定请求报头和查看响应报头，我们使用 WebClient——一个 Java 应用程序，使用它可以交互式地设定 HTTP 请求、提交它们并查看“原始”的结果。WebClient 的代码在本书主页(<http://wwwcoreservlets.com/>)上的源代码档案文件中可以找到。

2. 由初始化参数控制的初始化

前面的例子中，init 方法计算出由 doGet 和 getLastModified 方法使用的某种数据。尽管这种类型的常规初始化十分普遍，但是，使用初始化参数控制初始化也很常见。要理解 init 参数的动机，您需要了解什么样的人可能希望对 servlet 的行为方式进行定制。他们主要可以分为 3 组：

- (1) 开发人员
- (2) 最终用户
- (3) 部署人员

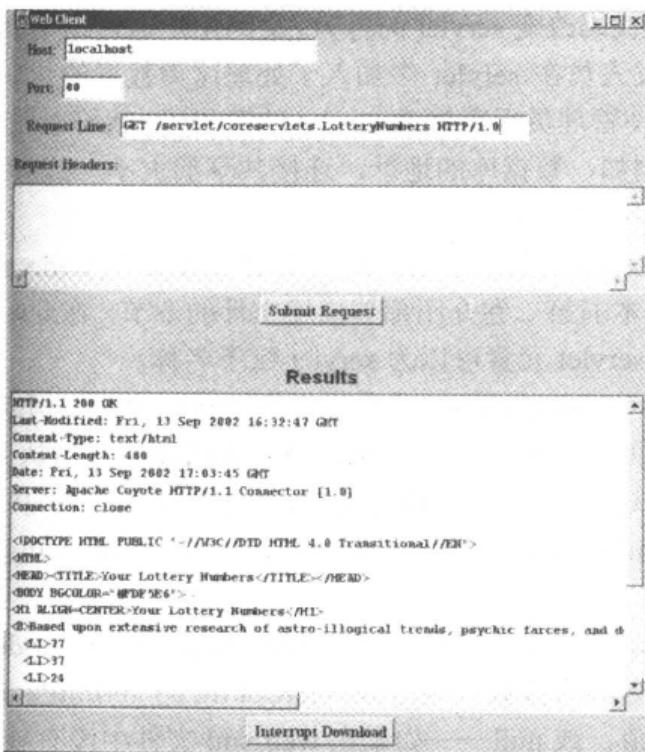


图 3.7 两种情况下对 LotteryNumbers servlet 的访问会得到正常的响应(客户端会接收到文档): 非条件性 GET 请求, 或指定的日期在 servlet 初始化之前的条件性请求。WebClient 程序(在此用来交互式地连接到服务器)的代码可以在 <http://wwwcoreservlets.com>/本书的源代码档案文件中找到

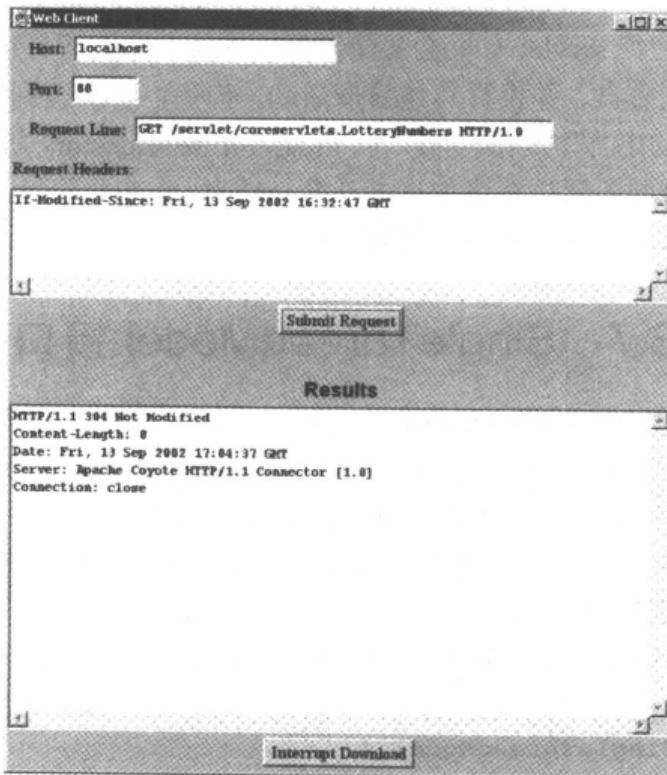


图 3.8 有一种情况对 LotteryNumbers servlet 的访问会得到 304(没有修改过)响应, 没有实际的文档: 接收到条件性 GET 请求, 且指定的日期就是 servlet 初始化的日期, 或之后

开发人员通过改变代码改变 servlet 的行为。最终用户通过向 HTML 表单提供数据改变 servlet 的行为(假定开发人员在 servlet 中加入了处理这类数据的代码)。但部署人员怎么办呢？需要有一种方式，使管理员无需修改 servlet 的源代码，就可以将 servlet 在机器间移动，以及改变特定的参数(例如，数据库的地址，连接共享的大小，或者数据文件的位置)。init 参数的目的就是为了提供这种能力。

由于 servlet 初始化参数的使用非常依赖于部署描述文件(web.xml)，所以，我们将 init 参数的细节和例子放到本书第二卷介绍部署描述文件的章节。在此，仅仅给出简略的预览。

- (1) web.xml 中的 servlet 元素可以为 servlet 赋予名称。
- (2) web.xml 中的 servlet-mapping 元素可以为 servlet 指定一个定制 URL。在使用 init 参数时，不要使用形如 `http://.../servlet/ServletName` 的默认 URL。实际上，尽管在最初的开发中这些默认 URL 极为便利，但在部署中几乎从不使用。
- (3) 通过向 web.xml 的 servlet 元素添加 init-param 子元素，可以指定初始化参数的名称和值。
- (4) 在 servlet 的 init 方法中，调用 `getServletConfig`，获取 `ServletConfig` 对象的引用。
- (5) 以 init 参数的名称为参数，调用 `ServletConfig` 的 `getInitParameter` 方法。返回值就是 init 参数的值，或 null——如果在 web.xml 文件中没有找到这个 init 参数。

3.6.4 destroy 方法

服务器可能会决定移除之前载入的 servlet 实例，或许因为服务器的管理员要求它这样做，或许由于服务器长时间空闲。但是，在服务器移除 servlet 的实例之前，它会调用 servlet 的 `destroy` 方法，从而使得 servlet 有机会关闭数据库连接、停止后台运行的线程、将 cookie 列表和点击计数写入到磁盘、并执行其他清理活动。但是，要意识到 Web 服务器有可能崩溃(还记得加利福尼亚的电力中断事件吗？)。因此，不要将 `destroy` 机制作为向磁盘上保存状态的惟一机制。如果服务器执行诸如点击计数，或对 cookie 值(表示特殊的访问)的列表进行累加等活动，应该主动地定期将数据写到磁盘上。

3.7 SingleThreadModel 接口

通常情况下，系统只生成 servlet 的单一实例，之后，为每个用户请求创建新的线程。这意味着，如果新的请求到来，而前面的请求依旧在执行，那么多个线程可能会并发地访问同一个 servlet 对象。因此，`doGet` 和 `doPost` 方法必须小心地同步对字段和其他共享数据(如果有的话)的访问，因为多个线程可能会同时对数据进行访问。要注意，多个线程并不共享局部变量，因此不需要特别的保护。

原则上，可以让 servlet 实现 `SingleThreadModel` 接口，阻止多线程访问，如下所示。

```
public class YourServlet extends HttpServlet
    implements SingleThreadModel {
    ...
}
```

如果实现了这个接口，系统会保证不会有多个请求线程同时访问该 servlet 的单个实例。

大多数情况下，系统将所有的请求排队，一次只将一个请求转给单个 servlet 实例。但是，服务器也可以创建由多个实例组成的池，同一时间每个实例都能够处理请求。不管通过哪种方式，您都勿需担心对 servlet 常规字段(实例变量)的同时访问。然而，您依旧必须同步对类变量或存储在 servlet 之外的共享数据的访问。

尽管原则上 SingleThreadModel 能够阻止并发访问，实际上，这通常是不好的选择，原因有二。

首先，如果您的 servlet 被频繁访问，那么同步对 servlet 的访问对性能会造成极大的损害(等待时间)。在 servlet 等待 I/O 时，服务器不能处理同一 servlet 的挂起请求。因此，在使用 SingleThreadModel 方式之前一定要慎重考虑。相应地，应该考虑只同步操作共享数据的那部分代码。

SingleThreadModel 的第二个问题来源于，规范允许服务器使用多个实例来处理请求，来替代对单个实例的请求进行排队的方案。只要每个实例同一时间只处理一个请求，实例共享的方式就满足规范的要求。但这不是一个好的方案。

一方面，假定您使用常规的非静态实例变量(字段)存储共享数据。当然，SingleThreadModel 会阻止并发的访问，但这样做是把孩子和洗澡水同时泼掉了：每个 servlet 实例都拥有实例变量的单独副本，因此，数据也就不能正确共享。

另一方面，假定您使用静态实例变量存储共享数据。SingleThreadModel 的共享池方式也就没有任何优点；多个请求(使用不同实例)依旧会并发地访问静态数据。

有时候 SingleThreadModel 依旧比较有用。例如，如果每个请求都需要重新初始化实例变量(比如，实例变量仅仅用来简化方法间的通信)，就可以使用 SingleThreadModel。但是，SingleThreadModel 的问题是如此严重，以至于 servlet 2.4(JSP 2.0)规范中明确地反对使用这种方式。使用明确的 synchronized 块要好得多。

警告

不要让高流量的 servlet 实现 SingleThreadModel。在其他情况下使用时也要极为小心。对于产品级的代码，明确的代码同步要好一些。servlet 规范的 2.4 版本不赞成使用 SingleThreadModel。

以清单 3.8 中的 servlet 为例，它试图为每个客户指定一个惟一的用户 ID(更确切地说，在服务器重启之前是惟一的)。它使用实例变量(字段)nextID 跟踪接下来应该指定哪个 ID，并使用下面的代码输出该 ID。

```
String id = "User-ID-" + nextID;
out.println("<H2>" + id + "</H2>");
nextID = nextID + 1;
```

现在，假定您十分小心地测试这个 servlet。您将它放在 coreservlets 子目录中，编译它，并将 coreservlets 目录复制到默认 Web 应用的 WEB-INF/classes 目录中(参见 2.10 节)。启动服务器。用 <http://localhost/servlet/coreservlets.UserID> 重复地访问这个 servlet。每次访问它，您都会得到一个不同的值(图 3.9)。故而可以得出代码是正确的，对吗？大错特错！只在多个客户同时访问这个 servlet 的时候才会产生问题。即使在这种情况下，也是偶尔发生。少

数情况下, 第一个客户代码读取了 nextID 字段后, 在递增该字段之前, 它的执行权可能被其他线程抢占。之后, 第二个客户有可能会读取该字段, 从而得到与第一个客户相同的值。这会产生大麻烦! 例如, 由于用户 ID 产生过程中的竞争条件, 可能会使现实世界中的电子商务应用程序偶尔会用错误的客户信用卡支付客户的订购商品。

现在, 如果您熟悉多线程编程, 那么对存在的问题应该十分明白。问题是, 正确的解决方案是什么呢? 下面给出 3 种可选方案。

(1) 减少竞争。

移除代码片段中的第 3 行, 将第一行改为下面的形式。

```
String id = "User-ID-" + nextID++;
```

这种方案降低了产生不正确答案的可能性, 但并没有完全消除这种可能。很多情况下, 降低错误答案出现的可能性是一件坏事情, 不是一件好事: 它只是意味着问题更难以在测试中发现, 而在真正应用之后更易于发生。

(2) 使用 SingleThreadModel。

将 servlet 的定义做如下修改:

```
public class UserIDs extends HttpServlet
    implements SingleThreadModel {
```

这样能够奏效吗? 如果服务器通过排队所有的请求来实现 SingleThreadModel, 那么, 是的, 它能够工作。但如果存在大量并发访问, 这种方式会导致性能上的巨大损失。更为不利的是, 如果服务器通过产生 servlet 的多个实例来实现 SingleThreadModel, 这种方式会完全失败, 因为每个实例都会拥有自己的 nextID 字段。两种服务器实现方式都是合法的, 因此, 这种“解决方案”不是根本的解决方案。

(3) 明确地同步对代码的访问。

使用 Java 编程语言的标准同步构造。在首次访问共享数据之前开启一个 synchronized 块, 完成对数据的最后更新之后并闭这个块, 如下所示。

```
synchronized(this) {
    String id = "User-ID-" + nextID;
    out.println("<H2>" + id + "</H2>");
    nextID = nextID + 1;
}
```

这项技术告诉系统, 一旦一个线程进入上面的代码块(或任何其他用同一对象引用标记的 synchronized 段)中, 则直到第一个线程退出为止, 不允许任何其他线程进入该代码段。Java 编程语言中经常使用这种解决方案。它也适用于现在这种情况。忘记易于出错且性能低下的 SingleThreadModel 吧! 用正确的方式处理竞争条件。

清单 3.8 coreservlets/UserIDs.java

```
package coreservlets;

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

```

/** Servlet that attempts to give each user a unique
 * user ID. However, because it fails to synchronize
 * access to the nextID field, it suffers from race
 * conditions: two users could get the same ID.
 */

public class UserIDs extends HttpServlet {
    private int nextID = 0;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Your ID";
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<CENTER>\n" +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1>" + title + "</H1>\n");
        String id = "User-ID-" + nextID;
        out.println("<H2>" + id + "</H2>");
        nextID = nextID + 1;
        out.println("</BODY></HTML>");
    }
}

```

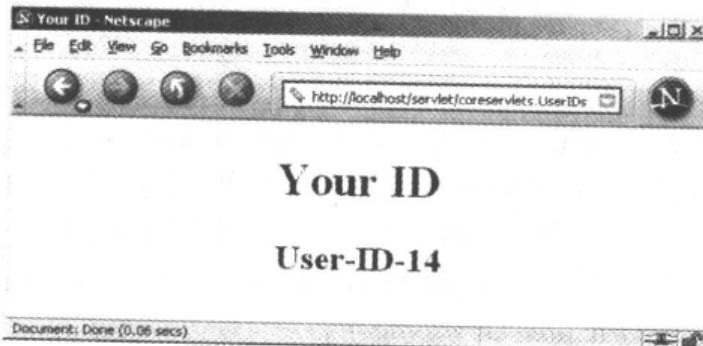


图 3.9 UserIDs servlet 的结果

3.8 servlet 的调试

毫无疑问，您在编写 servlet 时不会犯任何错误。但是，您的同事们可能会偶而犯错，您可以将这个建议传递给他们。认真地讲，由于不能直接执行 servlet，所以调试 servlet 需要一定的技巧。您只能通过 HTTP 请求触发它们的执行，并且它们是由 Web 服务器执行的。这种远程执行使得难以插入断点或读取调试信息，以及对栈进行跟踪。因而，servlet 的调试方式与那些常规开发中使用的手段稍有不同。下面是帮助调试 servlet 的 10 种通用策略。

(1) 使用打印语句。

如果您在桌面计算机上运行服务器，大多数服务器提供商都会让服务器弹出一个窗口，显示标准输出(即 `System.out.println` 语句的输出)。“什么？”，您可能会质疑，“您不会是在提倡类似旧式风格的打印语句吧？”嗯，是的。有许多复杂的调试技术，如果您熟悉它们，可以尽量使用它们。但是，仅仅收集基本的程序运作信息，就可能给调试工作带来令人惊奇的帮助。`init` 方法好像工作不正常吗？只需插入一个打印语句，重启服务器，检查打印语句是否显示在标准输出窗口中。也许您在声明 `init` 时犯了错，故而您定义的版本没有得到调用！如果产生 `NullPointerException`，可以插入几个打印语句，找出哪一行代码生成了这个错误、该行的哪个对象为 `null`。如果拿不定主意，请收集更多的信息。

(2) 使用集成在 IDE 中的调试器。

许多集成开发环境(IDE)都提供复杂的调试工具，可以集成到您的 servlet 和 JSP 容器中。IDE 的企业版，比如 Borland JBuilder, Oracle JDeveloper, IBM WebSphere Studio, Eclipse, BEA WebLogic Studio, Sun ONE Studio 等，一般都允许您插入断点、跟踪方法调用等。有些服务器甚至允许您连接到远程系统上运行的服务器。

(3) 使用日志文件。

`HttpServlet` 类有一个 `log` 方法，它允许您向服务器的日志文件写入信息。阅读日志文件中的调试信息，可能要比直接从窗口中观察它们(前两种方式即是如此)要麻烦一些，但是，即使在远程服务器上运行，依旧可以使用日志文件；而在这种情况下，打印语句几乎没有作用，只有高级 IDE 才支持远程调试。`log` 方法有两种变体：一个接受 `String` 参数，另一个接受 `String` 和 `Throwable`(`Exception` 类的先辈)参数。日志文件的精确位置与服务器相关，但一般都明确地记录在文档中，或可以在服务器安装目录的子目录中找到。

(4) 使用 Apache Log4J。

`Log4J` 是 Apache Jakarta 项目的一个包，Apache Jakarta 项目还管理着 Tomcat(本书中使用的一种服务器)和 Struts(一种 MVC 框架，在本书第二卷论述)。使用 `Log4J`，您半永久性地在代码中插入调试语句，使用基于 XML 的配置文件来控制在请求时间哪些语句应该调用。`Log4J` 快速、灵活、方便，并且越来越流行。详细信息，请参见 <http://jakarta.apache.org/log4j/>。

(5) 编写独立的类。

好的软件设计需要遵循的基本原则是将经常使用的代码放到单独的函数或类中，从而勿需总是重写这些代码。这个原则在编写 `servlet` 时甚至更为重要，因为这些单独的类常常可以独立于服务器进行测试。您甚至可以编写带有 `main` 方法的测试例程，它可以用来为您的例程产生成百上千的测试用例——如果需要手动地在浏览器中提交每个测试用例，您常常做不到这一点。

(6) 预先做好数据缺失或异常的准备。

如果从客户那里读取数据(第 4 章)，切记要检查数据是否为 `null` 或空字符串。在处理 HTTP 请求报头时(第 5 章)，切记报头是可选的，因而在特定的请求中可能为 `null`。在处理直接或间接来自于客户的数据时，一定要考虑到各种可能性，数据是

- 否含有输入错误或被完全省略。
- (7) **检查 HTML 源代码。**
如果您在浏览器中看到的结果比较奇怪,请从浏览器菜单中选择 View Source(【源文件】)。有时,一个小小的 HTML 错误,比如将</TABLE>写成了<TABLE>,会使页面的大部分内容不能显示。更好的方式是,可以使用正式的 HTML 验证器对 servlet 的输出进行验证。这种方式的论述请参见 3.5 节。
- (8) **单独检查请求数据。**
servlet 从 HTTP 请求中读取数据,构造响应,并将它发送回客户。如果这个过程发生错误,您肯定希望检查到底是由于客户发送了错误的数据,还是 servlet 对它的处理不正确。EchoServer 类——在第 19 章中论述,允许您提交 HTML 表单,得出到达服务器的精确数据。这个类不过是一个简单的 HTTP 服务器,它针对所有的请求,构造一个 HTML 页面,列出发送的内容。<http://wwwcoreservlets.com/> 提供全部的源代码。
- (9) **单独检查响应数据。**
单独检查完请求数据之后,您可能会希望单独检查响应数据。WebClient 类——在 3.6 节中的 init 示例中论述,允许您交互式地连接到服务器,发送自定义 HTTP 请求数据,并查看返回的所有数据——HTTP 响应报头以及其他所有内容。同样,可以从 <http://wwwcoreservlets.com/> 下载相关的源代码。
- (10) **停止和重启服务器。**
一般认为服务器应该将 servlet 保存在内存中,处理不同的请求,而非每次执行时都要重新载入它们。然而,大多数服务器支持一种开发模式,在这种模式下,当与 servlet 相关联的类文件发生改变后,它们应该自动重新载入。然而,有时服务器会不知所措,尤其是在您仅仅改变底层的类,没有改变顶层 servlet 的时候。因此,如果看起来您做出的更改没有相应反映在 servlet 的行为上,请重新启动服务器。类似地,init 方法仅仅在 servlet 初次载入时运行,web.xml 文件(参见 2.11 节)在 Web 应用载入后是只读的(尽管许多服务器都提供自定义的扩展,可以重新载入它),某些 Web 应用倾听程序(参见第二卷)仅仅在服务器首次启动时被触发。所有这些情况下,重启服务器都会简化调试工作。

第 4 章 客户请求的处理：表单数据

本章的主题：

- 单个请求参数的读取
- 全部请求参数的读取
- 缺失或异常数据的处理
- 过滤请求参数中的特殊字符
- 用请求参数的值自动填充数据对象
- 表单提交不完整的应对

之所以需要动态构建 Web 页面，主要动机之一是，只有这样才能根据用户的输入生成相应的结果。本章说明如何访问这些输入数据(4.1 节~4.4 节)。同时，本章还说明在某些期望的参数缺失时，如何使用默认值(4.5 节)；如何滤除请求数据中的<和>以避免 HTML 结果混乱(4.6 节)；如何创建能够根据请求数据自动填充的“表单 bean”(4.7 节)；以及，在所需的请求参数缺失时，如何重新显示表单，并将缺失的值突出显示(4.8 节)。

4.1 表单数据的作用

如果您曾使用过搜索引擎、访问过在线书店、在 Web 上跟踪证券行情或查阅过网站的机票报价，那么，您可能看到过一些奇怪的 URL，例如 `http://host/path?user=Marty+Hall&origin=bwi&dest=sfo`。问号后面的部分(即 `user=Marty+Hall&origin=bwi&dest=sfo`)称为表单数据(form data)(或查询数据，query data)，它是服务器端的程序从 Web 页面获取信息时所采用的最常见的方式。表单数据可以跟在问号后附加到 URL 的结尾(如上所示)，GET 请求即使用这种方式；表单数据还可以在单独的行中发送到服务器，POST 请求即为如此。如果您不熟悉 HTML 表单，可以参考第 19 章，其中详细介绍了如何构建收集和传输这类数据的表单。下面是一些基本的知识。

(1) 使用 FORM 元素创建 HTML 表单。

使用 ACTION 属性指定对结果进行处理的 servlet 和 JSP 页面的地址；可以使用绝对或相对 URL。例如：

```
<FORM ACTION="...">...</FORM>
```

如果省略 ACTION，那么数据将提交给当前页面对应的 URL。

(2) 使用输入元素收集用户数据。

将这些元素放在 FORM 元素的起始标签和结束标签之间，并为每个输入元素赋予一个 NAME。文本字段是最常用的输入元素；它们的创建方式如下：

```
<INPUT TYPE="TEXT" NAME="...">
```

(3) 在接近表单的尾部放置提交按钮。例如：

```
<INPUT TYPE="SUBMIT">
```

按下这个按钮时，浏览器会调用表单的 ACTION 指定的 URL。对于 GET 请求，URL 的结尾会附加一个问号和多个名/值对，其中，名称来自于 HTML 输入元素的 NAME 属性，值来自于终端用户。对于 POST 请求，所发送的数据相同，不过是在请求中单独的行发送，而不是附加在 URL 上。

从表单数据中提取所需的信息，一般是服务器端编程中最冗长乏味的工作之一。

第一，在使用 servlet 技术之前，一般必须采用一种方式读取 GET 请求的数据(传统 CGI 中，一般通过 QUERY_STRING 环境变量)，采用另一种不同的方式读取 POST 请求的数据(传统 CGI 中，通过读取标准输入)。

第二，必须在&符号处将名/值对切分出来，然后将参数的名称(等号的左方)和参数的值(等号的右方)分离开来。

第三，必须对值进行 URL 解码：解开浏览器对某些字符的编码。在发送字母数字字符时，浏览器不做任何改动，但会将空格转换成加号，其他字符转换成%XX，其中 XX 是该字符的 ASCII 码(或 ISO Latin-1)值，以十六进制表示。例如，如果用户在 HTML 表单中名为 users 的文本字段中输入“~hall, ~gates, and ~mcnealy”，该数据在发送时的形式是“users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy”，服务器端的程序必须从中重建出原来的数据。

最后，第 4 条理由是：由于有时参数的值可能省略(例如，“param1=val1¶m2=¶m3=val3”)，或者同一参数多次出现(例如，“param1=val1¶m2=val2¶m1=val3”)，故而采用传统的服务器端技术对表单数据进行分析时，分析代码需要对这些情况进行特殊处理，从而使得这个过程变得冗长笨重。

幸运的是，servlet 可以帮助我们完成大部分冗长笨重的分析工作。这就是下一节的主题。

4.2 在 servlet 中读取表单数据

servlet 提供的美妙特性之一就是：所有这种形式的分析工作都能自动完成。调用 request.getParameter 可以得到表单参数的值。如果参数多次出现，还可以调用 request.getParameterValues；如果希望得到当前请求中所有参数的完整列表，可以调用 request.getParameterNames。极少数情况下，可能需要读取原始的请求数据自己对它进行分析，此时可以调用 getReader 或 getInputStream。

4.2.1 单个值的读取：getParameter

读取请求(表单)的参数时，只需调用 HttpServletRequest 的 getParameter 方法，提供大小写敏感的参数名作为方法的参数。只要所提供的参数名与 HTML 源代码中出现的参数名完全相同，就可以得到与终端用户的输入完全一致的结果；任何必需的 URL 解码工作都已自动完成。不同于 servlet 技术的许多替代技术，不管数据是由 GET(即用在 doGet 方法中)发送，还是由 POST(即用在 doPost 中)发送，都可以用完全相同的方式使用 getParameter；servlet 知道客户使用的是哪种请求方法，并自动使用恰当的方法读取数据。如果参数存在但没有相应的值(即用户在提交表单时没有填写对应的文本字段)，则返回空的 String；如果没有这样的参数，则返回 null。

参数名对大小写敏感，因此，`request.getParameter("Param1")` 和 `request.getParameter("param1")` 不可以互换。

警告

提供给 `getParameter` 和 `getParameterValues` 的值是大小写敏感的。

4.2.2 多个值的读取：`getParameterValues`

如果同一参数名有可能在表单数据中多次出现，则应该调用 `getParameterValues`(返回字符串的数组)，而非 `getParameter`(返回单个字符串，对应参数首次出现时的值)。对于不存在的参数名，`getParameterValues` 的返回值为 `null`，如果参数只有单一的值，则返回只有一个元素的数组。

在此，如果您是 HTML 表单的设计者，最好保证每个文本字段、复选框或其他用户界面元素都有一个惟一的名称。这样，只使用简单的 `getParameter` 方法就能够完成所有的任务，完全避免使用 `getParameterValues`。然而，由于我们常常需要编写 servlet 和 JSP 来处理别人编写的 HTML 表单，因此，我们必须能够处理所有可能出现的情况。此外，多选列表框(即设置了 `MULTIPLE` 属性的 HTML `SELECT` 元素；详细信息请参见第 19 章)会为列表框中每个选定的元素，重复使用参数名。因此，多个值的情况不是总能避免。

4.2.3 参数名的查找：`getParameterNames` 和 `getParameterMap`

大多数 servlet 都会寻找一系列特定的参数名；大多数情况下，如果 servlet 不知道参数名，也就不知道如何处理它。因此，最基本的工具应该是 `getParameter`。但是，得出参数名的完整列表有时比较有用。对完整列表的基本应用是调试，但偶而也会在参数名变化很大的应用程序中使用这种列表。例如，名称本身或许能够告诉系统如何处理参数(如 `row-1-col-3-value`)；系统可能将参数名作为数据库中的列名，构建对数据库的更新；或者，服务器可能查找几个特定的名称，之后，将其余的名称传递给其他应用程序。

`getParameterNames` 以 `Enumeration` 的形式返回这种列表，其中的每一项都可以转换成 `String`，并可以用在 `getParameter` 或 `getParameterValues` 调用中。如果当前请求中没有参数，`getParameterNames` 返回空的 `Enumeration`(不是 `null`)。要注意，`Enumeration` 只是一个接口，它保证实际的类实现了 `hasMoreElements` 和 `nextElement` 方法；它并不保证具体的实现会采用某种特定的底层数据结构。同时，由于某些通用数据结构(尤其是散列表)打乱了元素的次序，因此，您不应该指望，`getParameterNames` 返回的参数会遵照它们在 HTML 表单中的次序。

警告

不要指望 `getParameterNames` 会以任何特定的次序返回参数名。

`getParameterNames` 的替代方案是 `getParameterMap`。这个方法返回一个 `Map`：参数名(字符串)是表的键，参数的值(字符串数组，与 `getParameterNames` 返回值相同)是表的值。

4.2.4 原始表单数据的读取及对上载文件的分析：getReader 或 getInputStream

除了读取个别的表单参数之外，我们还可以调用 `HttpServletRequest` 提供的 `getReader` 或 `getInputStream` 直接访问查询数据，之后，使用获得的流，分析原始的输入。但要注意，如果以这种方式读取数据，不能保证可以同时使用 `getParameter`。

对于常规的参数，读取原始数据不是一个好主意，因为这些输入既没有经过分析(分离成针对每个参数的项)，也没有经过 URL 解码(将加号变成空格，将`%XX`替换成 16 进制值`XX`对应的原始 ASCII 或 ISO Latin-1 字符)。但是，两种情况下需要读取原始的输入。

第一种情况，当数据不是由 HTML 表单提交，而是来自于定制的客户程序时，可能需要自己读取和分析这些数据。最常见的定制客户程序是 applet；applet-servlet 之间这种类型的通信在本书第二卷进行论述。

第二种情形，当数据来自于上载的文件时，可能需要自己读取数据。HTML 支持一种 FORM 元素(`<INPUT TYPE="FILE">`)，它允许客户将文件上载到服务器。遗憾的是，servlet 的 API 没有定义任何机制来读取这类文件。因此，我们需要用第三方提供的库来完成这项工作。应用最广泛的库中包括来自于 Apache Jakarta 项目的库。详细信息参见 <http://jakarta.apache.org/commons/fileupload/>。

4.2.5 多字符集输入的读取：setCharacterEncoding

默认情况下，`request.getParameter` 使用服务器的当前字符集解释输入。要改变这种默认行为，需要使用 `ServletRequest` 的 `setCharacterEncoding` 方法。但是，如果输入中使用了多种字符集，应该怎么办呢？这种情况下，不能简单地用普通的字符集名调用 `setCharacterEncoding`。之所以有这种限制，是因为 `setCharacterEncoding` 必须在访问任何请求参数之前调用，同时，多数情况下，我们需要使用请求中的参数(例如复选框)来确定字符集。

因而，我们只有两种选择：按照某个字符集读取参数，然后将它转换到另外的字符集；或者使用某些字符集提供的自动检测特性。

对于第一种选择，首先要读取感兴趣的参数，使用 `getBytes` 提取出原始的字节数据，之后将这些字节连同期望字符集的名称一同传递给 `String` 的构造函数。下面是将参数转换成日语的一个例子。

```
String firstNameWrongEncoding = request.getParameter("firstName");
String firstName =
    new String(firstNameWrongEncoding.getBytes(), "Shift_JIS");
```

对于第二种选择，需要使用一种支持从默认字符集进行检测和转换的字符集。Java 所支持的字符集的完整列表可以在 <http://java.sun.com/j2se/1.4.1/docs/guide/intl/encoding.doc.html> 获得。例如，如果允许输入既可以是英语，也可以是日语，则要使用下面的语句：

```
request.setCharacterEncoding("JISAutoDetect");
String firstName = request.getParameter("firstName");
```

4.3 示例：读取 3 个参数

清单 4.1 给出一个简单的 servlet——ThreeParams，它读取表单参数 param1，param2 和 param3，并将它们的值放在项目列表中。尽管与响应相关的设置需要在开始生成内容之前设定(参见第 6 章和第 7 章)，但并不要求在代码中特定的地方读取请求参数。因而，我们在准备好使用它们时才去读取这些参数。同时，请回顾一下，由于 ThreeParams 类在 coreservlets 包中，因此，它部署在 Web 应用(本例中为默认 Web 应用)中 WEB-INF/classes 目录下的 coreservlets 子目录中。

后面我们可以看到，同样的功能如何使用 JSP 来完成会得到极大简化，这个 servlet 就是一个绝佳的例子。等同的 JSP 版本参见 11.6 节。

清单 4.1 ThreeParams.java

```
package coreservlets;

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

/** Simple servlet that reads three parameters from the
 * form data.
 */

public class ThreeParams extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading Three Request Parameters";
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +"
            " \"Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                    "<UL>\n" +
                    " <LI><B>param1</B>: "
                    + request.getParameter("param1") + "\n" +
                    " <LI><B>param2</B>: "
                    + request.getParameter("param2") + "\n" +
                    " <LI><B>param3</B>: "
                    + request.getParameter("param3") + "\n" +
                    "</UL>\n" +
                    "</BODY></HTML>");
    }
}
```

清单 4.2 给出一个 HTML 表单，它收集用户的输入，并将其发送到这个 servlet。由于使用斜杠开头的 ACTION URL(/servlet/coreservlets.ThreeParams)，我们可以将这个表单安装

到默认 Web 应用的任何位置；也可以将该 HTML 表单移到别的目录或者将该 HTML 表单连同 servlet 一同移到其他的计算机，所有情况下都不需要对 HTML 表单或 servlet 做任何改动。以斜杠开头的表单 URL 可以增进可移植性，这一通用准则甚至在使用自定义 Web 应用时也完全适用，但需要在 URL 中包括 Web 应用的前缀。有关 Web 应用的详细信息，参见 2.11 节。当然，其他一些编写 URL 的方式也可以简化可移植性问题，但最重要的关键还是使用相对 URL(不含主机名)，而非绝对 URL(即 `http://host/...`)。如果使用绝对 URL，那么，在将 Web 应用从一台计算机移动到另外的计算机时，必须对表单进行编辑与修改。由于开发和部署几乎肯定不会在同一台计算机上进行，所以应该严格避免使用绝对地址。

核心方法

在表单中，一定要使用相对 ACTION URL，不要使用绝对 ACTION URL。

清单4.2 ThreeParamsForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Collecting Three Parameters</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Collecting Three Parameters</H1>

<FORM ACTION="/servlet/coreservlets.ThreeParams">
    First Parameter: <INPUT TYPE="TEXT" NAME="param1"><BR>
    Second Parameter: <INPUT TYPE="TEXT" NAME="param2"><BR>
    Third Parameter: <INPUT TYPE="TEXT" NAME="param3"><BR>
    <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</BODY></HTML>
```

回想一下，各个服务器中默认 Web 应用的位置都各不相同。HTML 表单一般放置在顶层目录中，或者除 WEB-INF 之外的子目录中。如果我们将 HTML 页面放在 form-data 子目录中，并从本地计算机对它进行访问，那么，在本书使用的 3 种示例服务器中，完整的安装位置如下：

- Tomcat 中的位置

install_dir/webapps/ROOT/form-data/ThreeParamsForm.html

- JRun 中的位置

install_dir/servers/default/default-ear/default-war/form-data/ThreeParamsForm.html

- Resin 中的位置

install_dir/doc/form-data/ThreeParamsForm.html

- 对应的 URL

http://localhost/form-data/ThreeParamsForm.html

图 4.1 给出的 HTML 表单中，用户已经输入了 3 个著名 Internet 人物的主目录名。也许有人会不同意这种说法，好吧！只有其中的两个比较著名^①，但此处的关键是，代字符(~)是一个非字母数字字符，在表单提交时，浏览器会对它进行 URL 编码。图 4.2 展示出这个

^① 毕竟，Gates 没有那么著名(作者在此有可能指盖茨对 Internet 的不重视，或者说的是反话。——译者注)。

servlet 的结果；注意，地址栏中是经过 URL 编码的值，但输出中却是表单域中原来的值：getParameter 总会返回与最终用户的输入完全相同的值，不管它们使用何种方式通过网络传送。

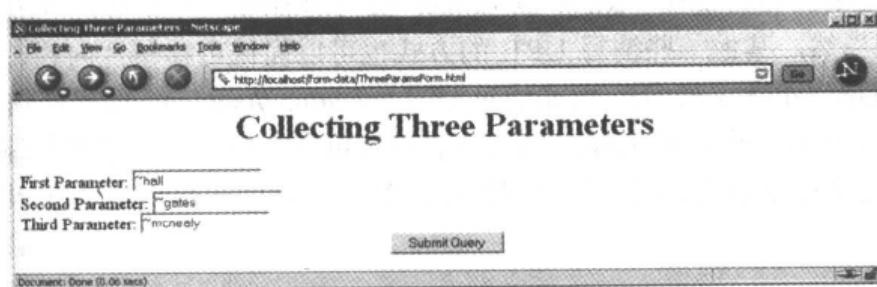


图 4.1 参数处理 servlet 的前端界面

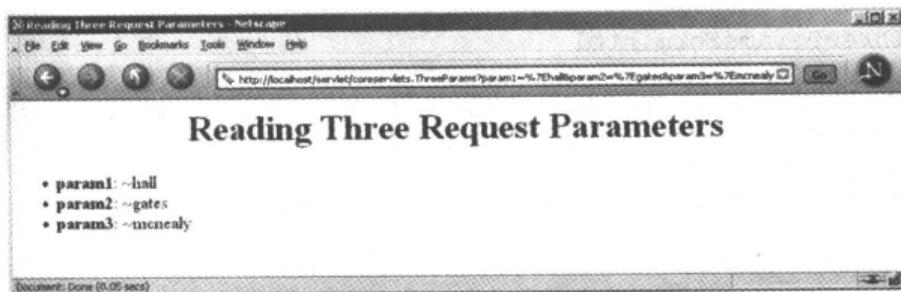


图 4.2 参数处理 servlet 的结果：请求参数已经自动经过 URL 解码

4.4 示例：读取所有参数

前面的例子从表单数据中根据预先指定的参数名，提取参数的值。它还假定每个参数有且仅有一个值。下面给出的例子查找发送来的所有参数名，并将它们的值放到一个表格中。它突出显示那些参数值缺失的参数，以及拥有多个值的参数。虽然在产品 servlet 中很少使用这种方式(如果不知道表单参数的名称，那么，就很有可能根本不知道应该用它们来做什么)，但对于调试来说却极为有用。

首先，这个 servlet 用 HttpServletRequest 的 getParameterNames 方法查出所有的参数名。getParameterNames 方法返回一个 Enumeration，其中包含参数名，次序未定。接下来，servlet 用标准的方式依次迭代 Enumeration，使用 hasMoreElements 确定什么时候停止，使用 nextElement 获取每个参数名。由于 nextElement 返回 Object，故而，这个 servlet 将结果转换成 String，并传递给 getParameterValues，得到一个字符串数组。如果这个数组中只有一项，且为空字符串，那么该参数没有值，该 servlet 生成斜体表示的“*No Value*”项。如果该数组中含有多项，那么该参数有多个值，该 servlet 将这些值显示在项目列表中。否则，将单一的值直接放入表格中。

这个 servlet 的源代码列在清单 4.3 中；清单 4.4 给出前端的 HTML 代码，可以用它来试验这个 servlet。图 4.3 和图 4.4 相应地展示出 HTML 前端和这个 servlet 的结果。

清单 4.3 ShowParameters.java

```
package coreservlets;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Shows all the parameters sent to the servlet via either
 * GET or POST. Specially marks parameters that have
 * no values or multiple values.
 */

public class ShowParameters extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        String title = "Reading All Request Parameters";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                    "<TR BGCOLOR=#FFAD00>\n" +
                    "<TH>Parameter Name<TH>Parameter Value(s)" );
        Enumeration paramNames = request.getParameterNames();
        while(paramNames.hasMoreElements()) {
            String paramName = (String)paramNames.nextElement();
            out.print("<TR><TD>" + paramName + "\n<TD>");
            String[] paramValues =
                request.getParameterValues(paramName);
            if (paramValues.length == 1) {
                String paramValue = paramValues[0];
                if (paramValue.length() == 0)
                    out.println("<I>No Value</I>");
                else
                    out.println(paramValue);
            } else {
                out.println("<UL>");
                for(int i=0; i<paramValues.length; i++) {
                    out.println("<LI>" + paramValues[i]);
                }
                out.println("</UL>");
            }
        }
        out.println("</TABLE>\n</BODY></HTML>");
    }

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

要注意到，这个 servlet 中的 `doPost` 方法只是简单地调用 `doGet`。这是因为我们希望它既能够处理 GET，也能够处理 POST 请求。如果希望在 HTML 界面如何向 servlet 发送数据方面获得某种程度的灵活性，那么，这是一种不错的标准做法。有关为什么让 `doPost` 调用 `doGet`(或者相反)要优于直接覆盖 `service` 的讨论，参见 3.6 节中对于 `service` 方法的论述。和所有含有密码域的表单一样，清单 4.4 中的 HTML 表单应用 POST(详细情况参见第 19 章)。但是，`ShowParameters` servlet 并非专门针对特定的前端，<http://www.coreservlets.com/> 上的源代码档案文件中包括一个类似的 HTML 表单，但这个表单使用 GET。

清单4.4 ShowParametersPostForm.html

```
!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>A Sample FORM using POST</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>
<FORM ACTION="/servlet/coreservlets.ShowParameters"
      METHOD="POST">
  Item Number: <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Description: <INPUT TYPE="TEXT" NAME="description"><BR>
  Price Each: <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
  <HR>
  First Name: <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Last Name: <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Middle Initial: <INPUT TYPE="TEXT" NAME="initial"><BR>
  Shipping Address:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Credit Card:<BR>
  &nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
                  VALUE="Visa">Visa<BR>
  &nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
                  VALUE="MasterCard">MasterCard<BR>
  &nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
                  VALUE="Amex">American Express<BR>
  &nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
                  VALUE="Discover">Discover<BR>
  &nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
                  VALUE="Java SmartCard">Java SmartCard<BR>
  Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  Repeat Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER><INPUT TYPE="SUBMIT" VALUE="Submit Order"></CENTER>
</FORM>
</BODY></HTML>
```

A Sample FORM using POST

Item Number: 123-A
Description: Wild Wonder Widget
Price Each: \$456.78

First Name: Sam
Last Name: Palmsano
Middle Initial: A
Shipping Address:
One Microsoft Way
Redmond, WA 98052

Credit Card:
 Visa
 MasterCard
 American Express
 Discover
 Java SmartCard

Credit Card Number: _____
Repeat Credit Card Number: _____

图 4.3 为 ShowParameter servlet 收集数据的 HTML 表单

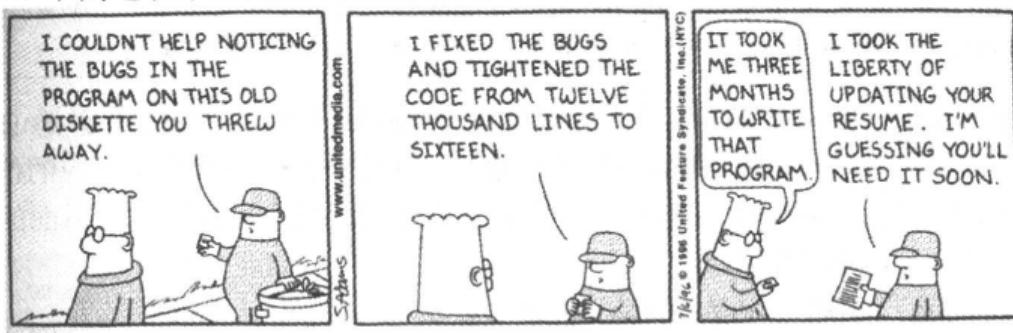
Reading All Request Parameters

Parameter Name	Parameter Value(s)
cardNum	• 3.1415927 • 3.1415927
cardType	Java SmartCard
price	\$456.78
initial	No Value
itemNum	123-A
address	One Microsoft Way Redmond, WA 98052
description	Wild Wonder Widget
firstName	Sam
lastName	Palmsano

图 4.4 ShowParameters servlet 的结果

4.5 参数缺失或异常时默认值的应用

在线求职服务最近已经变得越来越普及。声名卓著的网站向求职者提供服务，让他们的技能广为人知；同时也向雇主提供服务，使他们可以得到大量候选雇员的信息。本节提供一个完成这类网站部分功能的 servlet：根据用户提交的数据创建在线简历。现在的问题是：如果用户没有提供必需的信息，那么该 servlet 应该怎么处理这种情况？这个问题有两个答案：使用默认值，或者重新显示这个表单(提示用户缺失的值)。本节阐述默认值的应用；4.8 节阐述表单的重新显示。



翻印获得 United Feature Syndicate, Inc 的许可

在分析请求中的参数时，需要检查下面这三种情况：

(1) 参数的值为 null。

如果表单中没有期望名称的文本字段或其他元素，那么，期望的参数名根本不会出现在请求中，从而，对 request.getParameter 的调用返回 null。如果最终用户使用了不正确的 HTML 表单，或使用了包含 GET 数据的 URL 书签，但在制作 URL 书签之后，参数名发生变化，都会发生返回值为 null 的情况。为了避免 NullPointerException，必须在使用由 getParameter 返回的字符串之前，检查它是否

为 null。

(2) 参数的值为空字符串。

如果表单提交时相关的文本字段为空，则对 `request.getParameter` 的调用会返回空字符串(即：“””。要检查字符串是否为空字符串，可以使用 `equals` 方法与“”比较，或将字符串的长度与 0 比较。不要使用`==`运算符；Java 编程语言中，`=`总是用来测试两个参数是否为同一对象(位于相同的存储单元)，而非两个对象的相似性。为了安全起见，最好调用 `trim`，移除用户可能输入的任何空格，因为大多数情况下，一般会将纯空格当作缺失数据。因此，对缺失值的测试可能如下：

```
String param = request.getParameter("someName");
if ((param == null) || (param.trim().equals(""))) {
    doSomethingForMissingValues(...);
} else {
    doSomethingWithParameter(param);
}
```

(3) 参数的值为非空字符串，但格式错误。

错误格式的定义和具体的应用相关：您或许期望某些文本字段只包含数字值，或者某些文本字段有且只有 7 个字符，又或者某些文本字段只包含单个字母。

要注意，虽然在客户端可以使用 JavaScript 进行验证，但这并不能取代在服务器上执行这类检查的必要性。毕竟，服务器端的应用程序由您负责，而表单常常由另外的开发者或组织负责。即使他们未能检测出每种类型的不合法输入，您也不希望自己编写的应用程序崩溃。此外，客户还可能使用自己的 HTML 表单，或手动编辑含有 GET 数据的 URL，或禁用 JavaScript。

核心方法

在设计 servlet 时，要使之能够优雅地处理参数缺失(null 或空字符串)或格式不正确等情况。在测试 servlet 时，既要使用预期格式的数据，也要使用缺失或异常数据。

清单 4.5 和图 4.5 中展示的 HTML 表单是简历处理 servlet 的前端。如果不熟悉 HTML 表单，请参阅第 19 章。这个表单使用 POST 提交数据，并收集各个参数名所对应的值。在此，最需要了解的事情是，servlet 如何处理缺失和异常数据。这个过程汇总在下面的列表中。清单 4.6 给出 servlet 的完整代码。

- `name, title, email, languages, skills`

这些参数分别描述简历的各个组成部分。缺失的值应该由指定给参数的默认值替换。servlet 使用 `replaceIfMissing` 方法完成这项任务。

- `fgColor, bgColor`

这些参数指定页面的前景色和背景色。如果缺失这些值将会导致前景色为黑色，背景色为白色。servlet 还是使用 `replaceIfMissing` 方法完成这项任务。

- `headingFont, bodyFont`

这些参数分别指定标题和主文本使用的字体。如果缺失这个值或者这个值为“`default`”，则以 `sans-serif` 字体显示标题和主文本，如 Arial 或 Helvetica。servlet 使用 `replaceIfMissingOrDefault` 方法完成这项任务。

- **headingSize, bodySize**

这些参数分别指定主标题和正文文本的点数。子标题会以比主标题稍小一些的大小显示。如果未给出相应的值，或给出非数字值，则以默认大小(标题 32，正文 18)显示相应的内容。servlet 通过对 Integer.parseInt 的调用，以及捕获 NumberFormatException 的 try/catch 块，来处理这种情况。

清单4.5 SubmitResume.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Free Resume Posting</TITLE>
<LINK REL=STYLESHEET
      HREF="jobs-site-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>hot-computer-jobs.com</H1>
<P CLASS="LARGER">
To use our <I>free</I> resume-posting service, simply fill
out the brief summary of your skills below. Use "Preview"
to check the results, then press "Submit" once it is
ready. Your mini-resume will appear online within 24 hours.</P>
<HR>
<FORM ACTION="/servlet/core servlets.SubmitResume"
      METHOD="POST">
<DL>
<DT><B>First, give some general information about the look of
your resume:</B>
<DD>Heading font:
      <INPUT TYPE="TEXT" NAME="headingFont" VALUE="default">
<DD>Heading text size:
      <INPUT TYPE="TEXT" NAME="headingSize" VALUE=32>
<DD>Body font:
      <INPUT TYPE="TEXT" NAME="bodyFont" VALUE="default">
<DD>Body text size:
      <INPUT TYPE="TEXT" NAME="bodySize" VALUE=18>
<DD>Foreground color:
      <INPUT TYPE="TEXT" NAME="fgColor" VALUE="BLACK">
<DD>Background color:
      <INPUT TYPE="TEXT" NAME="bgColor" VALUE="WHITE">

<DT><B>Next, give some general information about yourself:</B>
<DD>Name: <INPUT TYPE="TEXT" NAME="name">
<DD>Current or most recent title:
      <INPUT TYPE="TEXT" NAME="title">
<DD>Email address: <INPUT TYPE="TEXT" NAME="email">
<DD>Programming Languages:
      <INPUT TYPE="TEXT" NAME="languages">

<DT><B>Finally, enter a brief summary of your skills and
experience:</B> (use &lt;P&gt; to separate paragraphs.
      Other HTML markup is also permitted.)
<DD><TEXTAREA NAME="skills"
      ROWS=10 COLS=60 WRAP="SOFT"></TEXTAREA>
</DL>
<CENTER>
      <INPUT TYPE="SUBMIT" NAME="previewButton" Value="Preview">
```

```

<INPUT TYPE="SUBMIT" NAME="submitButton" Value="Submit">
</CENTER>
</FORM>
<HR>
<P CLASS="TINY">See our privacy policy
<A HREF="we-will-spam-you.html">here</A>.</P>
</BODY></HTML>

```



图 4.5 简历预览 servlet 的前端

清单4.6 SubmitResume.java

```

package coreservlets;

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

/** Servlet that handles previewing and storing resumes
 * submitted by job applicants.
 */

public class SubmitResume extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        if (request.getParameter("previewButton") != null) {
            showPreview(request, out);
        } else {
            storeResume(request);
            showConfirmation(request, out);
        }
    }
}

```

```
/** Shows a preview of the submitted resume. Takes
 * the font information and builds an HTML
 * style sheet out of it, then takes the real
 * resume information and presents it formatted with
 * that style sheet.
 */

private void showPreview(HttpServletRequest request,
                        PrintWriter out) {
    String headingFont = request.getParameter("headingFont");
    headingFont = replaceIfMissingOrDefault(headingFont, "");
    int headingSize =
        getSize(request.getParameter("headingSize"), 32);
    String bodyFont = request.getParameter("bodyFont");
    bodyFont = replaceIfMissingOrDefault(bodyFont, "");
    int bodySize =
        getSize(request.getParameter("bodySize"), 18);
    String fgColor = request.getParameter("fgColor");
    fgColor = replaceIfMissing(fgColor, "BLACK");
    String bgColor = request.getParameter("bgColor");
    bgColor = replaceIfMissing(bgColor, "WHITE");
    String name = request.getParameter("name");
    name = replaceIfMissing(name, "Lou Zer");
    String title = request.getParameter("title");
    title = replaceIfMissing(title, "Loser");
    String email = request.getParameter("email");
    email =
        replaceIfMissing(email, "contact@hot-computer-jobs.com");
    String languages = request.getParameter("languages");
    languages = replaceIfMissing(languages, "<I>None</I>");
    String languageList = makeList(languages);
    String skills = request.getParameter("skills");
    skills = replaceIfMissing.skills, "Not many, obviously.");
    out.println
        (ServletUtilities.DOCTYPE + "\n" +
         "<HTML><HEAD><TITLE>Resume for " + name + "</TITLE>\n" +
         makeStyleSheet(headingFont, headingSize,
                       bodyFont, bodySize,
                       fgColor, bgColor) + "\n" +
         "</HEAD>\n" +
         "<BODY>\n" +
         "<CENTER>\n" +
         "<SPAN CLASS=\"HEADING1\">" + name + "</SPAN><BR>\n" +
         "<SPAN CLASS=\"HEADING2\">" + title + "<BR>\n" +
         "<A HREF=\"" + email + "\">" + email +
         "</A></SPAN>\n" +
         "</CENTER><BR><BR>\n" +
         "<SPAN CLASS=\"HEADING3\">Programming Languages" +
         "</SPAN>\n" +
         makeList(languages) + "<BR><BR>\n" +
         "<SPAN CLASS=\"HEADING3\">Skills and Experience" +
         "</SPAN><BR><BR>\n" +
         skills + "\n" +
         "</BODY></HTML>");

}

/** Builds a cascading style sheet with information
 * on three levels of headings and overall
```

```

* foreground and background cover. Also tells
* Internet Explorer to change color of mailto link
* when mouse moves over it.
*/



private String makeStyleSheet(String headingFont,
                             int heading1Size,
                             String bodyFont,
                             int bodySize,
                             String fgColor,
                             String bgColor) {
    int heading2Size = heading1Size*7/10;
    int heading3Size = heading1Size*6/10;
    String styleSheet =
        "<STYLE TYPE=\"text/css\">\n" +
        "<!--\n" +
        ".HEADING1 { font-size: " + heading1Size + "px;\n" +
        "            font-weight: bold;\n" +
        "            font-family: " + headingFont +
        "               Arial, Helvetica, sans-serif;\n" +
        "}\n" +
        ".HEADING2 { font-size: " + heading2Size + "px;\n" +
        "            font-weight: bold;\n" +
        "            font-family: " + headingFont +
        "               Arial, Helvetica, sans-serif;\n" +
        "}\n" +
        ".HEADING3 { font-size: " + heading3Size + "px;\n" +
        "            font-weight: bold;\n" +
        "            font-family: " + headingFont +
        "               Arial, Helvetica, sans-serif;\n" +
        "}\n" +
        "BODY { color: " + fgColor + ";\n" +
        "       background-color: " + bgColor + ";\n" +
        "       font-size: " + bodySize + "px;\n" +
        "       font-family: " + bodyFont +
        "          Times New Roman, Times, serif;\n" +
        "}\n" +
        "A:hover { color: red; }\n" +
        "-->\n" +
        "</STYLE>";
    return(styleSheet);
}

/** Replaces null strings (no such parameter name) or
 * empty strings (e.g., if textfield was blank) with
 * the replacement. Returns the original string otherwise.
 */
private String replaceIfMissing(String orig,
                               String replacement) {
    if ((orig == null) || (orig.trim().equals("")))
        return(replacement);
    else
        return(orig);
}

// Replaces null strings, empty strings, or the string

```

```
// "default" with the replacement.  
// Returns the original string otherwise.  
  
private String replaceIfMissingOrDefault(String orig,  
                                         String replacement) {  
    if ((orig == null) ||  
        (orig.trim().equals("")) ||  
        (orig.equals("default"))) {  
        return(replacement);  
    } else {  
        return(orig + ", ");  
    }  
}  
  
// Takes a string representing an integer and returns it  
// as an int. Returns a default if the string is null  
// or in an illegal format.  
  
private int getSize(String sizeString, int defaultSize) {  
    try {  
        return(Integer.parseInt(sizeString));  
    } catch(NumberFormatException nfe) {  
        return(defaultSize);  
    }  
}  
  
// Given "Java,C++,Lisp", "Java C++ Lisp" or  
// "Java, C++, Lisp", returns  
// "<UL>  
//   <LI>Java  
//   <LI>C++  
//   <LI>Lisp  
// </UL>"  
  
private String makeList(String listItems) {  
    StringTokenizer tokenizer =  
        new StringTokenizer(listItems, ", ");  
    String list = "<UL>\n";  
    while(tokenizer.hasMoreTokens()) {  
        list = list + " <LI>" + tokenizer.nextToken() + "\n";  
    }  
    list = list + "</UL>";  
    return(list);  
}  
  
/** Show a confirmation page when they press the  
 * "Submit" button.  
 */  
  
private void showConfirmation(HttpServletRequest request,  
                           PrintWriter out) {  
    String title = "Submission Confirmed.";  
    out.println(ServletUtilities.headWithTitle(title) +  
                "<BODY>\n" +  
                "<H1>" + title + "</H1>\n" +  
                "Your resume should appear online within\n" +  
                "24 hours. If it doesn't, try submitting\n" +  
                "again with a different email address.\n" +
```

```

        "</BODY></HTML>");

}

/** Why it is bad to give your email address to
 * untrusted sites.
 */

private void storeResume(HttpServletRequest request) {
    String email = request.getParameter("email");
    putInSpamList(email);
}

private void putInSpamList(String emailAddress) {
    // Code removed to protect the guilty.
}
}
}

```

该 servlet 获取所有字体和颜色参数的有意义的值之后，它会根据这些信息构建一份层叠样式表(Cascading Style Sheet, CSS)。HTML 4.0 网页中，样式表是指定字体、字号、颜色、缩进和其他格式化信息的标准方式。样式表一般存放在独立的文件中，这样同属一个站点的多个网页可以共享同一个样式表。但就当前这个例子来讲，直接使用 STYLE 元素将样式信息嵌入到页面中更为方便。有关样式表的更多信息，参见 <http://www.w3.org/TR/REC-CSS1>。

创建完样式表之后，该 servlet 依次将求职者的姓名、职位和电子邮件居中安置在页面的顶部。这些行使用标题字体，电子邮件地址放置在 mailto:超文本链接中，这样，未来的雇主可以通过在邮件地址上单击，直接联系求职者。languages 参数中指定的编程语言由 StringTokenizer 来分析(假定使用空格或逗号分隔语言名)，并放在“Programming Languages”标题之下的项目列表中。最后，来自于 skills 参数的文本放置在页面的底部，“Skills and Experience”标题后面。

图 4.6 和图 4.7 分别给出提供和省略所需参数时的结果。图 4.8 是单击 Submit 的结果。

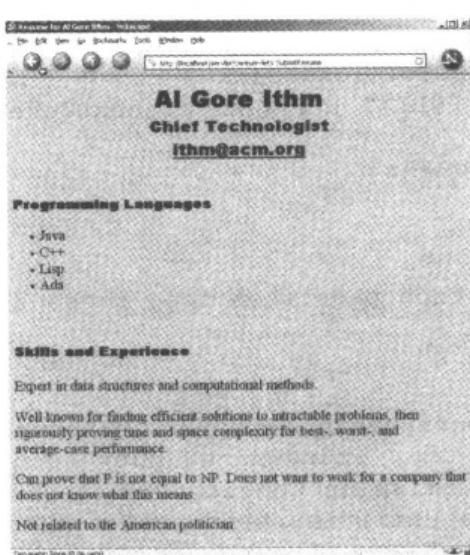


图 4.6 预览：提交的简历含有所需的表单数据

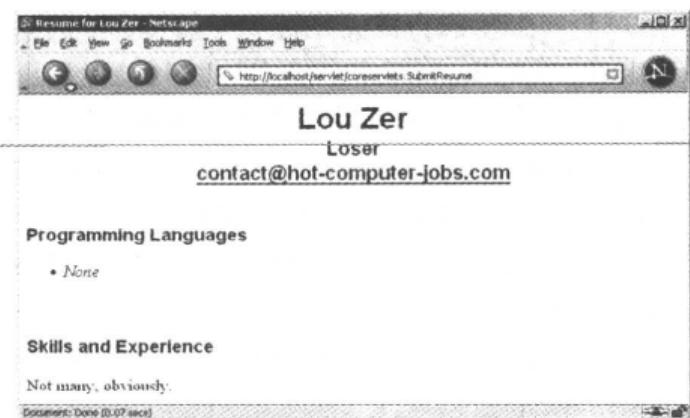


图 4.7 预览：提交的数据中缺失很多所需的数据(默认值取代省略的值)

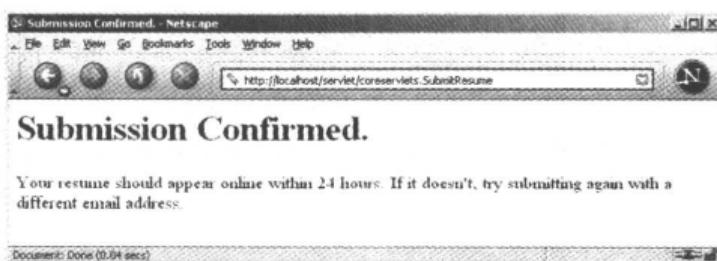


图 4.8 将简历提交给数据库之后的结果

4.6 过滤字符串中的 HTML 特殊字符

一般地，如果 servlet 希望生成含有诸如<或>等字符的 HTML，只需简单地使用标准的 HTML 字符实体——<或>。类似地，如果 servlet 要在 HTML 属性值中使用双引号或&符号，只需使用"或&。如果不做这些替换，最终生成的 HTML 代码将会不正确，因为<或>常常被解释为 HTML 标记中标签的一部分，属性值中的双引号可能会被解释为属性值的结尾，而&符号根本不允许出现在属性值中。大多数情况下，找出特殊的字符和使用标准的 HTML 替换都比较容易。然而，有两种情况，手动完成这种替换不那么容易。

第一种情况，手动转换比较困难，这发生在字符串来源于程序摘录，或者其他已经按照某种标准格式进行组织的来源时。这种情况下，手动地检查并更改所有的特殊字符单调乏味，而忘记转换哪怕一个特殊字符都会导致 Web 页面中某些部分缺失，或者格式不正确。

第二种情况，手动转换难以完成，这发生在字符串来源于 HTML 表单数据的时候。在此，转换绝对必须在运行期间执行，因为在编译期间当然不可能知道查询数据。如果用户偶然或蓄意地输入 HTML 标签，那么，生成的 Web 页面将会包含虚假的 HTML 标签，这可能会产生完全不可预期的结果(HTML 规范规定了浏览器应该如何处理合法的 HTML；但并没有说明 HTML 含有不正确的语法时应该做什么)。

核心方法

如果您需要读取请求参数，并将它们的值显示在生成的页面中，则必须过滤出那些特殊的 HTML 字符。不这样做可能会导致输出中缺失某些部分，或者某些部分格式错误。

对于外界能够访问的网页，如果没有过滤字符串中的特殊字符，可能会使得您的页面成为跨站点脚本攻击(cross-site scripting attack)的工具。这种情况下，恶意的程序员将 GET 参数嵌入到指向您的 servlet(或任何其他服务器端程序)的 URL 中。这些 GET 参数扩展成利用浏览器已知 bug 的 HTML 标签(一般为<SCRIPT>)。因而，攻击者能够将代码嵌入到指向您网站的 URL 之中，然后只散布这个 URL，而非恶意 Web 页面自身。通过这种方式，攻击者能够更容易地隐藏自己的身份，同时还可以利用信任关系，使得用户认为脚本来源于可靠的来源(您的组织)。有关这个问题更详细的信息，参见 <http://www.cert.org/advisories/CA-2000-02.html> 和 <http://www.microsoft.com/technet/security/topics/ExSumCS.asp>。

4.6.1 执行过滤的代码

替换字符串中的<，>，"，和&比较简单，多种不同的方式都能完成这项任务。但重要的是，要牢记 Java 字符串是不可改变的(即不能修改)，因此，重复的字符串拼接操作需要复制许多字符串片段，并在使用后废弃。例如，考虑下面这两行代码：

```
String s1 = "Hello";
String s2 = s1 + " World";
```

由于 s1 是不可修改的，第二行代码首先要生成 s1 的副本，将" World"追加到该副本，然后这个副本就被废弃。为了避免生成和复制这些临时对象所带来的开销，在循环中执行重复性的拼接操作时，应该使用可以改变的数据结构；StringBuffer 是通常的选择。

核心方法

如果您在循环中执行字符串拼接操作，不要使用 String，而应使用 StringBuffer。

清单 4.7 给出一个静态的 filter 方法，它使用 StringBuffer 高效地将输入的字符串复制到过滤后的版本，在这个过程中，替换了 4 个特殊的字符。

清单 4.7 ServletUtilities.java (节选)

```
package coreservlets;

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

public class ServletUtilities {
    ...

    /** Replaces characters that have special HTML meanings
     * with their corresponding HTML character entities.
     */
    // Note that Javadoc is not used for the more detailed
    // documentation due to the difficulty of making the
    // special chars readable in both plain text and HTML.
    //
    // Given a string, this method replaces all occurrences of
    // '<' with '&lt;', all occurrences of '>' with
    // '&gt;', and (to handle cases that occur inside attribute
```

```

// values), all occurrences of double quotes with
// " and all occurrences of & with &.
// Without such filtering, an arbitrary string
// could not safely be inserted in a Web page.

public static String filter(String input) {
    if (!hasSpecialChars(input)) {
        return(input);
    }
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for(int i=0; i<input.length(); i++) {
        c = input.charAt(i);
        switch(c) {
            case '<': filtered.append("<"); break;
            case '>': filtered.append(">"); break;
            case '\"': filtered.append("\""); break;
            case '&': filtered.append("&"); break;
            default: filtered.append(c);
        }
    }
    return(filtered.toString());
}

private static boolean hasSpecialChars(String input) {
    boolean flag = false;
    if ((input != null) && (input.length() > 0)) {
        char c;
        for(int i=0; i<input.length(); i++) {
            c = input.charAt(i);
            switch(c) {
                case '<': flag = true; break;
                case '>': flag = true; break;
                case '\"': flag = true; break;
                case '&': flag = true; break;
            }
        }
    }
    return(flag);
}
}

```

4.6.2 示例：用来显示代码片断的 servlet

以清单 4.8 中的 HTML 表单为例，它收集 Java 编程语言的一个代码片断，将它发送给清单 4.9 中的 servlet 进行显示。

如果用户输入常规的输入，结果是正确的，如图 4.9 所示。但是，如图 4.10 所示，当输入中包含特殊字符时，如<和>，结果就不可预料了。由于 HTML 规范并没有说明这种情况下应该怎么办，所以不同的浏览器可能会给出不同的结果，但大多数浏览器会认为“if(a<b){”中的“<b”开启一个 HTML 标签。但是，由于不能识别 b 之后的字符。浏览器会忽略它们，直到下一个>出现为止，也就是</PRE>标签的结尾。因此，不仅大部分代码片断消失，而且浏览器也不解释</PRE>标签，所以代码片断之后的文字显示也不正常——使用等宽字体且禁用自动换行。

清单 4.10 给出的 servlet, 除了在显示请求参数的值之前, 过滤掉其中的特殊字符以外, 功能上与前一个 servlet 完全相同。清单 4.11 给出一个向它发送数据的 HTML 表单(除 ACTION URL 之外, 这个表单与清单 4.8 中的表单完全相同)。图 4.11 展示出, 在接收到前一个 servlet 未能处理好的输入时, 得出的结果: 此时没有问题。

清单4.8 CodeForm1.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Submit Code Sample</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1 ALIGN="CENTER">Submit Code Sample</H1>
<FORM ACTION="/servlet/coreservlets.BadCodeServlet">
    Code:<BR>
    <TEXTAREA ROWS="6" COLS="40" NAME="code"></TEXTAREA><P>
    <INPUT TYPE="SUBMIT" VALUE="Submit Code">
</FORM>
</CENTER></BODY></HTML>
```

清单4.9 BadCodeServlet.java

```
package coreservlets;

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

/** Servlet that reads a code snippet from the request
 * and displays it inside a PRE tag. Fails to filter
 * the special HTML characters.
 */
public class BadCodeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Code Sample";
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +"
            "Transitional//EN\\\">\\n";
        out.println(docType +
                    "<HTML\\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\\n" +
                    "<BODY BGCOLOR=\\\"#FDF5E6\\\">\\n" +
                    "<H1 ALIGN=\\\"CENTER\\\">" + title + "</H1>\\n" +
                    "<PRE>\\n" +
                    getCode(request) +
                    "</PRE>\\n" +
                    "Now, wasn't that an interesting sample\\n" +
                    "of code?\\n" +
                    "</BODY></HTML>");

    }

    protected String getCode(HttpServletRequest request) {
```

```

        return(request.getParameter("code"));
    }
}

```

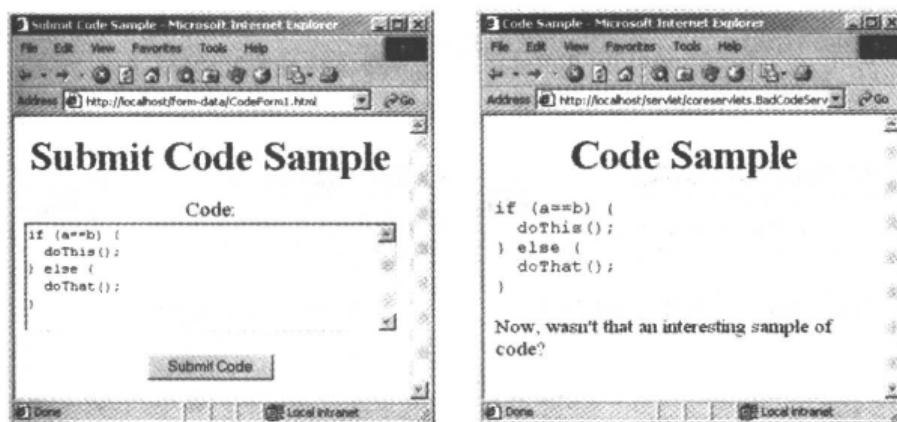


图 4.9 BadCodeServlet: 请求参数中没有特殊字符时结果正常

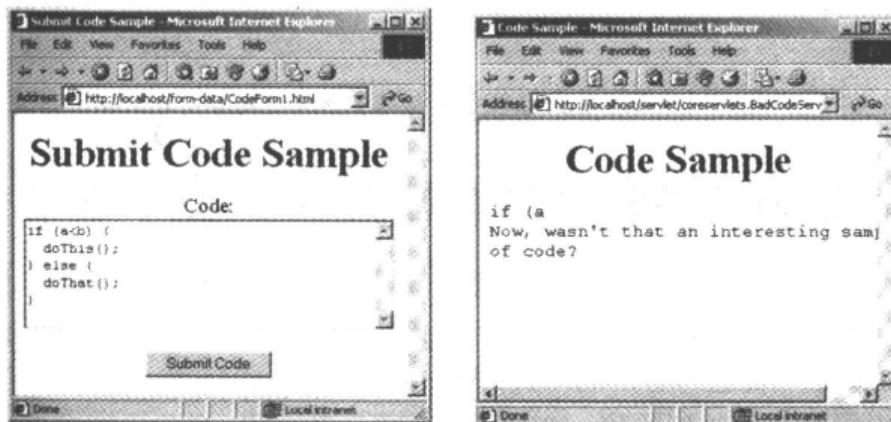


图 4.10 BadCodeServlet: 请求参数中含有特殊字符时结果中某些部分缺失或格式不正确

清单4.10 GoodCodeServlet.java

```

package coreservlets;

import javax.servlet.http.*;

/**
 * Servlet that reads a code snippet from the request
 * and displays it inside a PRE tag. Filters the special HTML characters.
 */

public class GoodCodeServlet extends BadCodeServlet {
    protected String getCode(HttpServletRequest request) {
        return(ServletUtilities.filter(super.getCode(request)));
    }
}

```

清单4.11 CodeForm2.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Submit Code Sample</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">

```

```

<CENTER>
<H1 ALIGN="CENTER">Submit Code Sample</H1>
<FORM ACTION="/servlet/coreservlets.GoodCodeServlet">
  Code:<BR>
  <TEXTAREA ROWS="6" COLS="40" NAME="code"></TEXTAREA><P>
  <INPUT TYPE="SUBMIT" VALUE="Submit Code">
</FORM>
</CENTER></BODY></HTML>

```

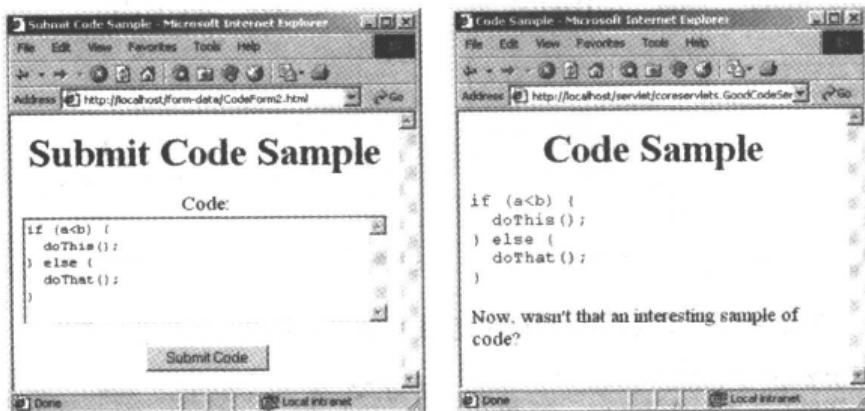


图 4.11 GoodCodeServlet：即使请求参数中含有特殊字符结果依旧正常

4.7 根据请求参数自动填充 Java 对象：表单 bean

`getParameter` 方法使得读取输入的请求参数更为容易：无论在 `doGet` 中，还是在 `doPost` 中，都使用同一个方法，返回的值已经自动完成了 URL 解码(即：转换成终端用户输入时的格式，而非在网络上传送所必需的格式)。但是，由于 `getParameter` 的返回值是 `String`，所以，如果希望得到其他类型的值，必须自己解析返回的值(当然也要检查缺失或异常数据)。例如，如果您希望得到 `int` 或 `double` 值，则必须将 `getParameter` 的结果传递给 `Integer.parseInt` 或 `Double.parseDouble`，并将相应的代码放在 `try/catch` 块中，检查 `NumberFormatException` 异常。如果您要处理许多请求参数，那么，这个过程可能会十分冗长。

例如，假定您有一个数据对象，其中含有 3 个 `String` 字段、2 个 `int` 字段、2 个 `double` 字段和 1 个 `boolean`。基于提交的表单数据填充这个对象，需要调用 `getParameter` 8 次，`Integer.parseInt` 和 `Double.parseDouble` 各 2 次，还需要一些专门的代码设置 `boolean` 标志。最好能够自动完成这项工作。

当前，在 JSP 中，使用 JavaBean 组件构架可以极大地简化读取请求参数、提取相应的值、并将结果存储到 Java 对象中的过程。这一过程在第 14 章详细论述。如果您不熟悉 bean 的概念，请参阅第 14 章中的内容。不过，它的中心思想是：普通的 Java 对象，如果它所属的类使用私有字段，且拥有遵循 `get/set` 命令约定的方法，则可以看作是 bean。方法名(除去单词“get”或“set”，并且首字母小写)称为属性(property)。例如，任意 Java 类，只要含有 `getName` 和 `setName` 方法，我们就称，它定义了一个拥有 `name` 属性的 bean。

如第 14 章所述，存在特殊的 JSP 语法(`jsp:setProperty` 调用中的 `property="*"`)，可以用来一举完成 bean 的填写工作。特别地，这项设置表明系统应该检查输入的所有请求参数，

并将它们传递给与请求中的参数名相匹配的 bean 属性。例如，如果将请求参数命名为 param1，那么该参数将会传递给对象的 setParam1 方法。另外，简单的类型转换可以自动完成。例如，如果有一个名为 numOrdered 的请求参数，同时对象拥有一个期望接收 int 的 setNumOrdered 方法(即该 bean 拥有一个类型为 int 的 numOrdered 属性)，请求参数 numOrdered 会自动转换成 int，生成的值会自动传递给 setNumOrdered 方法。

在此，如果在 JSP 中能够完成上述的工作，那么，在 servlet 中应该同样能够做到。毕竟，如第 10 章所述，JSP 页面实质上就是 servlet：每个 JSP 页面都要转换成 servlet，在请求期间运行的是 servlet。另外，在第 15 章我们将会看到，在复杂的情况下，最好组合使用 servlet 和 JSP，由 servlet 完成编程任务而 JSP 页面负责表示任务(presentation work)。因而，在 servlet 中能够比 JSP 页面更容易地读取请求参数真得十分重要。但不可思议的是，servlet 规范却没有提供这种功能：完成 property="*" 这一 JSP 过程的代码并没有通过一个标准的 API 暴露出来。

幸运的是，Apache 软件基金会的 Jakarta 通用包(Jakarta Commons Package，参见 <http://jakarta.apache.org/commons/>)得到广泛的应用，它提供的类使我们可以更容易地构建自动将请求参数和 bean 属性关联起来(即使用 setXxx 方法)的实用程序。下一小节介绍如何获得该通用包，但此处的要点是静态的 populateBean 方法接受一个 bean(即至少有一些方法遵循 get/set 命名约定的 Java 对象)和一个 Map 作为输入，将 Map 中所有值传递给 bean 中与相关的 Map 键名匹配的属性。这个实用程序还自动完成类型转换，当对应的请求参数格式异常时，不抛出异常，而是使用默认值(例如，数字值的默认值为 0)。

清单 4.12 提供一个实用工具类，它使用 Jakarta 通用库(Jakarta Commons)中的实用工具类，依据输入的请求参数自动填充 bean。要使用它，只需将 bean 和请求对象一同传递给 BeanUtilities.populateBean。这就够了。您希望将两个请求参数放入一个数据对象吗？没问题：只需一次方法调用。15 个请求参数连同类型转换的情况呢？也没有问题，还是同样的，只需一次方法调用。

清单 4.12 BeanUtilities.java

```
package coreservlets.beans;

import java.util.*;
import javax.servlet.http.*;
import org.apache.commons.beanutils.BeanUtils;

/** Some utilities to populate beans, usually based on
 * incoming request parameters. Requires three packages
 * from the Apache Commons library: beanutils, collections,
 * and logging. To obtain these packages, see
 * http://jakarta.apache.org/commons/. Also, the book's
 * source code archive (see http://www.coreservlets.com/)
 * contains links to all URLs mentioned in the book, including
 * to the specific sections of the Jakarta Commons package.
 */
public class BeanUtilities {
```

```

/** Examines all of the request parameters to see if
 * any match a bean property (i.e., a setXxx method)
 * in the object. If so, the request parameter value
 * is passed to that method. If the method expects
 * an int, Integer, double, Double, or any of the other
 * primitive or wrapper types, parsing and conversion
 * is done automatically. If the request parameter value
 * is malformed (cannot be converted into the expected
 * type), numeric properties are assigned zero and boolean
 * properties are assigned false: no exception is thrown.
*/
public static void populateBean(Object formBean,
                                HttpServletRequest request) {
    populateBean(formBean, request.getParameterMap());
}

/** Populates a bean based on a Map: Map keys are the
 * bean property names; Map values are the bean property
 * values. Type conversion is performed automatically as
 * described above.
*/
public static void populateBean(Object bean,
                                Map propertyMap) {
    try {
        BeanUtils.populate(bean, propertyMap);
    } catch(Exception e) {
        // Empty catch. The two possible exceptions are
        // java.lang.IllegalAccessException and
        // java.lang.reflect.InvocationTargetException.
        // In both cases, just skip the bean operation.
    }
}
}

```

4.7.1 BeanUtilities 的使用

清单 4.13 给出一个收集雇员保险信息的 servlet，它可能用来确定可行的保险计划和相关的费用。要执行这项任务，这个 servlet 需要用雇员的姓名和 ID(都是 String 类型)、子女个数(int)、以及是否已婚(boolean)等信息填写保险信息数据对象(InsuranceInfo.java，清单 4.14)。由于这个对象被表达为 bean，所以，可以使用 BeanUtilities.populateBean，通过一次方法调用，填写所需的信息。清单 4.15 给出了收集数据的 HTML 表单；图 4.12 和图 4.13 给出典型的结果。

清单4.13 SubmitInsuranceInfo.java

```

package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import coreservlets.beans.*;

/** Example of simplified form processing. Illustrates the

```

```

* use of BeanUtilities.populateBean to automatically fill
* in a bean (Java object with methods that follow the
* get/set naming convention) from request parameters.
*/



public class SubmitInsuranceInfo extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        InsuranceInfo info = new InsuranceInfo();
        BeanUtilities.populateBean(info, request);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN">\n";
        String title = "Insurance Info for " + info.getName();
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<CENTER>\n" +
                    "<H1>" + title + "</H1>\n" +
                    "<UL>\n" +
                    " <LI>Employee ID: " +
                    info.getEmployeeID() + "\n" +
                    " <LI>Number of children: " +
                    info.getNumChildren() + "\n" +
                    " <LI>Married?: " +
                    info.isMarried() + "\n" +
                    "</UL></CENTER></BODY></HTML>");
    }
}

```

清单4.14 InsuranceInfo.java

```

package coreservlets.beans;

import coreservlets.*;

/** Simple bean that represents information needed to
 * calculate an employee's insurance costs. Has String,
 * int, and boolean properties. Used to demonstrate
 * automatically filling in bean properties from request
 * parameters.
 */

public class InsuranceInfo {
    private String name = "No name specified";
    private String employeeID = "No ID specified";
    private int numChildren = 0;
    private boolean isMarried = false;

    public String getName() {
        return(name);
    }

    /** Just in case user enters special HTML characters,

```

```

    * filter them out before storing the name.
    */

public void setName(String name) {
    this.name = ServletUtilities.filter(name);
}

public String getEmployeeID() {
    return(employeeID);
}

/** Just in case user enters special HTML characters,
 * filter them out before storing the name.
 */

public void setEmployeeID(String employeeID) {
    this.employeeID = ServletUtilities.filter(employeeID);
}

public int getNumChildren() {
    return(numChildren);
}

public void setNumChildren(int numChildren) {
    this.numChildren = numChildren;
}

/** Bean convention: name getter method "isXxx" instead
 * of "getXxx" for boolean methods.
 */

public boolean isMarried() {
    return(isMarried);
}

public void setMarried(boolean isMarried) {
    this.isMarried = isMarried;
}
}

```

清单4.15 InsuranceForm.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Employee Insurance Signup</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1>Employee Insurance Signup</H1>

<FORM ACTION="/servlet/core servlets.SubmitInsuranceInfo">
    Name: <INPUT TYPE="TEXT" NAME="name"><BR>
    Employee ID: <INPUT TYPE="TEXT" NAME="employeeID"><BR>
    Number of Children: <INPUT TYPE="TEXT" NAME="numChildren"><BR>
    <INPUT TYPE="CHECKBOX" NAME="married" VALUE="true">Married?<BR>
    <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</CENTER></BODY></HTML>

```

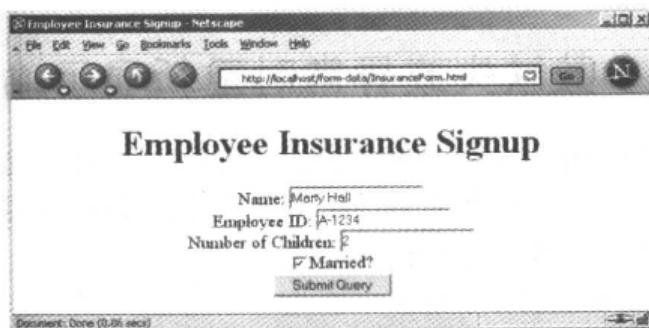


图 4.12 保险处理 servlet 的前端

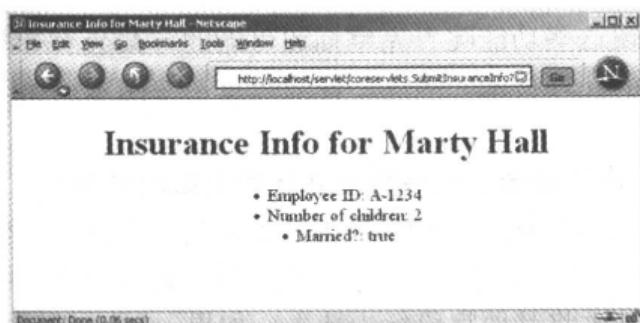


图 4.13 保险处理 servlet：通过使用 BeanUtilities.populateBean，请求数据的收集得到了极大的简化

4.7.2 Jakarta 通用包的获取和安装

BeanUtilities 类的大部分工作都由 Jakarta 通用库中的 BeanUtils 组件完成。这个组件执行必要的反射(reflection)(确定该对象拥有哪些可写 bean 属性——setXxx 方法)和类型转换(将 String 解析成 int, double, boolean, 或者其他基本类型或包装类型)。因而，除非安装 Jakarta 通用库中的 BeanUtils，否则 BeanUtilities 不能完成工作。但是，由于 BeanUtils 依赖于 Jakarta 通用库中另外两个组件——Collections 和 Logging，所以您必须下载和安装所有这 3 个组件。

要下载这些组件，首先到 <http://jakarta.apache.org/commons/>，在左边的栏中找到“Components Repository”标题，下载这 3 个组件 JAR 文件的最新版本。(我们的代码基于 BeanUtils 的 1.5 版本，但是，最近的版本几乎都相同。)下载这些组件最简单的方式也许是访问 <http://www.coreservlets.com/>，跳转到源代码档案文件的 Chapter 4，找出到这 3 个 JAR 文件的直接链接。

安装这 3 种组件时，最易于移植的方式是遵循下面的标准方案：

- 对于开发，将这 3 个 JAR 文件列在 CLASSPATH 中；
- 对于部署，将这 3 个 JAR 文件放在 Web 应用的 WEB-INF/lib 目录中。

在处理类似的 JAR 文件时(用在多个 Web 应用中)，许多开发人员使用服务器专有的、支持跨 Web 应用共享 JAR 文件的特性。例如，Tomcat 允许将通用 JAR 文件放在 *tomcat_install_dir/common/lib*。许多人在他们的开发计算机上使用另外的快捷方式——将这 3 个 JAR 文件拖入到 *sdk_install_dir/jre/lib/ext*。这样做使得这些 JAR 文件可以自动被开发环境和本地安装的服务器访问。这些都是比较有用的技巧，但要牢记 *your-web-app/WEB-INF/lib* 是部署服务器上惟一标准化的位置，

4.8 当参数缺失或异常时重新显示输入表单

当用户没有填写某些表单域时，有时需要使用默认值。但有时没有合理的默认值可用，则应该将表单重新显示给用户。常规的 HTML 表单不能完成这两项期望的功能。

- 用户不应该需要再次输入已经提供的值；
- 缺失的表单域应该突出标示出来。

4.8.1 重新显示的几种方案

那么，应该怎样实现这些功能呢？完整描述可能的方案会有一些复杂，并且需要用到本书第一卷中没有介绍的几项技术。因此，详细信息请参阅本书第二卷，此处只给出一个简短的介绍。

- **由同一 servlet 提供表单、处理数据并提供最后的结果。**

这个 servlet 首先寻找输入的请求数据：如果它没有找到任何数据，则提供一个空的表单。如果 servlet 找到不完整的请求数据，则提取出这些不完整的数据，将它放回到表单中，并将其他字段标为缺失。如果 servlet 得到所有的补充请求数据，则处理该请求并显示结果。表单省略 ACTION 属性，从而，表单提交时会自动发送到表单自身的 URL。这个方案是惟一一个我们已经介绍了所有必需技术的方案，因而也是本节将要阐述的方案。

- **由一个 servlet 提供表单；由第二个 servlet 处理数据并提供结果。**

由于这个方案将工作进行了划分，使得每个 servlet 可以更小，更易于管理，因此要优于第一个方案。但是，这种方案需要用到两种我们尚未介绍的技术，即：如何将控制从一个 servlet 传递到另一个；如何在 servlet 中访问另外的 servlet 中创建的用户专有数据。从一个 servlet 转到另外的 servlet 可以使用 response.sendRedirect(参见第 6 章)或 RequestDispatcher 的 forward 方法(参见第 15 章)。将数据从负责处理的 servlet，传递回显示表单的 servlet 时，最简单的方式是将它存储在 HttpSession 对象中(参见第 9 章)。

- **由一个 JSP 页面“手动地”提供表单；由一个 servlet 或 JSP 页面处理数据并提供结果。**

这是很好的方案，并被广泛采用。但是，它除了需要用到前面项目中提到的两项技术(如何将用户从数据处理 servlet 转到表单页面，以及如何用会话跟踪存储用户专有数据)之外，还需要 JSP 的相关知识。特别地，您需要了解，如何使用 JSP 表达式(参见第 11 章)或 `jsp:getProperty`(参见第 14 章)从数据对象中提取不完整数据，并将它放入到 HTML 表单中。

- **由一个 JSP 页面提供表单，用从数据对象获取的值自动填写表单中相应的字段；由一个 servlet 或 JSP 页面处理这些数据并提供最终的结果。**

这或许是所有方案中最好的一个。但是，除了前面项目提及的技术之外，它需要定制的 JSP 标签，模仿 HTML 表单元素，并自动使用指定的值。可以自己编写这些标签，也可以使用预制的版本，比如 JSTL 或 Apache Struts 中的那些标签(有关

定制标签、JSTL 和 Struts 的内容，参见本书第二卷)。

4.8.2 处理拍卖竞标的 servlet

我们以拍卖网站上处理竞标的 servlet 为例，说明表单重新显示的第一种方案。图 4.14 到图 4.16 展示出期望的结果：这个 servlet 首先显示一个空表单；如果提交的数据不完整，则重新显示表单，并将缺失的数据标识出来；在提交了全部的数据之后处理该请求。

要完成这种行为，这个 servlet(清单 4.16)执行下面这些步骤。

- (1) 根据请求数据填写 BidInfo 对象(清单 4.17)，利用 BeanUtilities.populateBean(参见 4.7 节)自动将请求中的参数名与 bean 的属性相匹配，并执行简单的类型转换。
- (2) 检查 BidInfo 对象是否完全为空(任何字段都为默认值，没有发生过更改)。如果为空，则调用 showEntryForm 显示初始的输入表单。
- (3) 检查 BidInfo 对象是否部分为空(某些字段发生了改变，但并非全部)。如果是这种情况，则调用 showEntryForm 显示输入表单，同时给出警告消息并将缺失的字段突出显示。用户已经输入数据的字段保持它们之前的值。
- (4) 检查 BidInfo 对象是否完全填写完毕。如果完全，则调用 showBid 对数据进行处理并提供结果。

图 4.14 servlet 的初始表单：它提供一个表单收集拍卖过程中的竞标数据

图 4.15 数据不完全的竞标 servlet。如果用户提供没有填写完整的表单，竞标 servlet 则重新显示该表单。用户之前填写的数据得到了维护，缺失的数据突出显示



图 4.16 数据完整的竞标 servlet：它提供了结果

清单4.16 BidServlet.java

```

package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import coreservlets.beans.*;

/** Example of simplified form processing. Shows two features:
 * <OL>
 *   <LI>Automatically filling in a bean based on the
 *       incoming request parameters.
 *   <LI>Using the same servlet both to generate the input
 *       form and to process the results. That way, when
 *       fields are omitted, the servlet can redisplay the
 *       form without making the user reenter previously
 *       entered values.
 * </UL>
 */

public class BidServlet extends HttpServlet {

    /** Try to populate a bean that represents information
     * in the form data sent by the user. If this data is
     * complete, show the results. If the form data is
     * missing or incomplete, display the HTML form
     * that gathers the data.
     */

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        BidInfo bid = new BidInfo();
        BeanUtilities.populateBean(bid, request);
        if (bid.isComplete()) {
            // All required form data was supplied: show result.
            showBid(request, response, bid);
        } else {
            // Form data was missing or incomplete: redisplay form.
            showEntryForm(request, response, bid);
        }
    }
}

```

```
}

/** All required data is present: show the results page. */

private void showBid(HttpServletRequest request,
                      HttpServletResponse response,
                      BidInfo bid)
    throws ServletException, IOException {
    submitBid(bid);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Bid Submitted";
    out.println
        (DOCTYPE +
         "<HTML>\n" +
         "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
         "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
         "<H1>" + title + "</H1>\n" +
         "Your bid is now active. If your bid is successful,\n" +
         "you will be notified within 24 hours of the close\n" +
         "of bidding.\n" +
         "<P>\n" +
         "<TABLE BORDER=1>\n" +
         " <TR><TH BGCOLOR=\"BLACK\"><FONT COLOR=\"WHITE\">" +
         bid.getItemName() + "</FONT>\n" +
         " <TR><TH>Item ID: " +
         bid.getItemId() + "\n" +
         " <TR><TH>Name: " +
         bid.getBidderName() + "\n" +
         " <TR><TH>Email address: " +
         bid.getEmailAddress() + "\n" +
         " <TR><TH>Bid price: $" +
         bid.getBidPrice() + "\n" +
         " <TR><TH>Auto-increment price: " +
         bid.isAutoIncrement() + "\n" +
         "</TABLE></CENTER></BODY></HTML>");
}

/** If the required data is totally missing, show a blank
 * form. If the required data is partially missing,
 * warn the user, fill in form fields that already have
 * values, and prompt user for missing fields.
 */
private void showEntryForm(HttpServletRequest request,
                           HttpServletResponse response,
                           BidInfo bid)
    throws ServletException, IOException {
    boolean isPartlyComplete = bid.isPartlyComplete();
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title =
        "Welcome to Auctions-R-Us. Please Enter Bid.";
    out.println
        (DOCTYPE +
         "<HTML>\n" +
         "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
```

```
"<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
"<H1>" + title + "</H1>\n" +
warning(isPartlyComplete) +
"<FORM>\n" +
InputElement("Item ID", "itemID",
    bid.getItemId(), isPartlyComplete) +
InputElement("Item Name", "itemName",
    bid.getItemName(), isPartlyComplete) +
InputElement("Your Name", "bidderName",
    bid.getBidderName(), isPartlyComplete) +
InputElement("Your Email Address", "emailAddress",
    bid.getEmailAddress(), isPartlyComplete) +
InputElement("Amount Bid", "bidPrice",
    bid.getBidPrice(), isPartlyComplete) +
checkbox("Auto-increment bid to match other bidders?",
    "autoIncrement", bid.isAutoIncrement()) +
"<INPUT TYPE=\"SUBMIT\" VALUE=\"Submit Bid\">\n" +
"</CENTER></BODY></HTML>");

}

private void submitBid(BidInfo bid) {
    // Some application-specific code to record the bid.
    // The point is that you pass in a real object with
    // properties populated, not a bunch of strings.
}

private String warning(boolean isFormPartlyComplete) {
    if(isFormPartlyComplete) {
        return("<H2>Required Data Missing! " +
            "Enter and Resubmit.</H2>\n");
    } else {
        return("");
    }
}

/** Create a textfield for input, prefaced by a prompt.
 * If this particular textfield is missing a value but
 * other fields have values (i.e., a partially filled form
 * was submitted), then add a warning telling the user that
 * this textfield is required.
 */
private String InputElement(String prompt,
                            String name,
                            String value,
                            boolean shouldPrompt) {
    String message = "";
    if (shouldPrompt && ((value == null) || value.equals("")))) {
        message = "<B>Required field!</B> ";
    }
    return(message + prompt + ":" + "\n" +
        "<INPUT TYPE=\"TEXT\" NAME=\"" + name + "\" " +
        "VALUE=\"" + value + "\"><BR>\n");
}

private String InputElement(String prompt,
                            String name,
                            double value,
```

```
        boolean shouldPrompt) {
    String num;
    if (value == 0.0) {
        num = "";
    } else {
        num = String.valueOf(value);
    }
    return(inputElement(prompt, name, num, shouldPrompt));
}

private String checkbox(String prompt,
                      String name,
                      boolean isChecked) {
    String result =
        prompt + ": " +
        "<INPUT TYPE=\"CHECKBOX\" NAME=\"" + name + "\"";
    if (isChecked) {
        result = result + " CHECKED";
    }
    result = result + "><BR>\n";
    return(result);
}

private final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
     \"Transitional//EN\">\n";
}
```

清单4.17 BidInfo.java

```
package coreservlets.beans;

import coreservlets.*;

/** Bean that represents information about a bid at
 * an auction site. Used to demonstrate redisplay of forms
 * that have incomplete data.
 */

public class BidInfo {
    private String itemID = "";
    private String itemName = "";
    private String bidderName = "";
    private String emailAddress = "";
    private double bidPrice = 0;
    private boolean autoIncrement = false;

    public String getItemName() {
        return(itemName);
    }

    public void setItemName(String itemName) {
        this.itemName = ServletUtilities.filter(itemName);
    }

    public String getItemID() {
        return(itemID);
    }

    public void setItemID(String itemID) {
        this.itemID = ServletUtilities.filter(itemID);
    }
}
```

```
}

public String getBidderName() {
    return bidderName;
}

public void setBidderName(String bidderName) {
    this.bidderName = ServletUtilities.filter(bidderName);
}

public String getEmailAddress() {
    return emailAddress;
}

public void setEmailAddress(String emailAddress) {
    this.emailAddress = ServletUtilities.filter(emailAddress);
}

public double getBidPrice() {
    return bidPrice;
}

public void setBidPrice(double bidPrice) {
    this.bidPrice = bidPrice;
}

public boolean isAutoIncrement() {
    return autoIncrement;
}

public void setAutoIncrement(boolean autoIncrement) {
    this.autoIncrement = autoIncrement;
}

/** Has all the required data been entered? Everything except
 * autoIncrement must be specified explicitly (autoIncrement
 * defaults to false).
 */
public boolean isComplete() {
    return (hasValue(getItemID()) &&
           hasValue(getItemName()) &&
           hasValue(getBidderName()) &&
           hasValue(getEmailAddress()) &&
           (getBidPrice() > 0));
}

/** Has any of the data been entered? */

public boolean isPartlyComplete() {
    boolean flag =
        (hasValue(getItemID()) ||
        hasValue(getItemName()) ||
        hasValue(getBidderName()) ||
        hasValue(getEmailAddress()) ||
        (getBidPrice() > 0) ||
        isAutoIncrement());
    return flag;
}
```

```
private boolean hasValue(String val) {  
    return(val != null) && (!val.equals(""));  
}  
}
```

第 5 章 客户请求的处理：HTTP 请求报头

本章的主题：

- HTTP 请求报头的读取
- 构建所有请求报头的表格
- 了解各种请求报头
- 压缩页面减少下载时间
- 区分不同的浏览器类型
- 依据客户的到达方式定制页面
- 标准 CGI 变量的访问

创建高效 servlet 的关键之一，就是要了解如何操纵超文本传输协议(HyperText Transfer Protocol, HTTP)。透彻地了解这个协议能够对提高 servlet 的性能和可用性起到立竿见影的效果。这一部分论述以请求报头的形式从浏览器发送到服务器的 HTTP 信息。本节讲述 HTTP 1.1 中最重要的请求报头、概括如何以及为什么要在 servlet 中使用这些请求报头。后面我们会看到，请求报头在 JSP 中的读取及应用与在 servlet 中相同。

请注意，HTTP 请求报头不同于前一章论述的表单(查询)数据。表单数据直接来源于用户的输入，对于 GET 请求，这些数据是 URL 的一部分，对于 POST 请求，这些数据在单独的行中。相应地，请求报头由浏览器间接地设定，并紧跟在初始的 GET 和 POST 请求行之后发送。例如，下面的例子给出了一个 HTTP 请求，它可能是用户向位于 <http://www.somebookstore.com/servlet/Search> 的 servlet 提交书籍搜索请求时产生的。这个请求包括 Accept, Accept-Encoding, Connection, Cookie, Host, Referer 和 User-Agent 报头，所有这些报头都有可能对 servlet 的运作至关重要，但它们都不能从表单数据中自动导出或演绎出来：如果 servlet 要用到这些信息，必须显式地读取请求报头。

```
GET/servlet/Search?keywords=servlets+jsp HTTP/1.1
Accept:image/gif,image/jpg,*/
Accept-Encoding:gzip
Connection:Keep-Alive
Cookie:userID=id456578
Host:www.somebookstore.com
Referer:http://www.somebookstore.com/findbooks.html
User-Agent:Mozilla/4.0 (compatible;MSIE 6.0;Windows NT 5.0)
```

5.1 请求报头的读取

报头的读取比较简单：只需用报头的名称为参数，调用 HttpServletRequest 的 getHeader 方法。如果当前的请求中提供了指定的报头，则这个调用返回一个 String，否则返回 null。在 HTTP 1.0 中，请求的所有报头都是可选的；在 HTTP 1.1 中，只有 Host 是必需的。因而，在使用请求报头之前一定要检查是否为 null。

核心方法

在使用 `request.getHeader` 返回的结果之前，一定要确保它不是 null。

报头名称对大小写不敏感。例如，`request.getHeader("Connection")` 完全等同于 `request.getHeader("connection")`。

尽管 `getHeader` 是读取输入报头的通用方式，但由于几种报头的应用太过普遍，故而 `HttpServletRequest` 为它们提供了专门的访问方法。下面是对这些方法的汇总。

- **getCookies**
`getCookies` 方法返回 Cookie 报头的内容，这些内容经分析后存储在由 `Cookie` 对象构成的数组中。第 8 章对这个方法进行详细的论述。
- **getAuthType 和 getRemoteUser**
`getAuthType` 和 `getRemoteUser` 方法对 `Authorization` 报头进行拆分，分解成它的各个构成部分。
- **getContentLength**
`getContentLength` 方法返回 `Content-Length` 报头的值(作为一个 int 值返回)。
- **getContentType**
`getContentType` 方法返回 `Content-Type` 报头的值(作为一个 String 返回)。
- **getDateHeader 和 getIntHeader**
`getDateHeader` 和 `getIntHeader` 方法读取指定的报头，然后分别将它们转换成 `Date` 和 `int` 值。
- **getHeaderNames**
除了查找特定的报头之外，您还可以使用 `getHeaderNames` 方法得到一个 `Enumeration`，枚举当前特定请求中所有的报头名称。5.2 节阐述这项功能。
- **getHeaders**
大多数情况下，每个报头名称在请求中只出现一次。然而，报头偶而也有可能出现多次，每次出现列出各自的值。`Accept-Language` 就是一例。您可以使用 `getHeaders` 获取一个 `Enumeration`，枚举报头每次出现所对应的值。

最后，除了查找请求的报头之外，您还可以获取主请求行自身的信息(即前面给出的示例中请求的第一行)，同样是使用 `HttpServletRequest` 提供的方法。下面是对 4 种主要方法的汇总：

- **getMethod**
`getMethod` 方法返回主请求方法(一般是 GET 或 POST，但也有可能是 HEAD, PUT 和 DELETE 方法)。
- **getRequestURI**
`getRequestURI` 方法返回 URL 中主机和端口之后，但在表单数据之前的部分。以 URL `http://randomhost.com/servlet/search.BookSearch?subject=jsp` 为例，`getRequestURI` 返回"/servlet/search.BookSearch"。
- **getQueryString**
`getQueryString` 方法返回表单数据。同样以 URL `http://randomhost.com/servlet/`

search.BookSearch?subject=jsp 为例, `getQueryString` 返回 "subject=jsp"。

- `getProtocol`

`getProtocol` 方法返回请求行的第三部分, 一般为 HTTP/1.0 或 HTTP/1.1。servlet 一般应该在指定专用于 HTTP 1.1 的响应报头(第 7 章)之前检查 `getProtocol`。

5.2 制作所有请求报头的表格

清单 5.1 列出的 servlet 只是简单地创建一个表格, 其中列出它接收到的所有报头, 以及与报头相关联的值。它通过调用 `request.getHeaderNames` 获取当前请求中所有报头的 `Enumeration`, 来完成这项任务。之后, 它循环迭代 `Enumeration`, 将报头名称放在左边的表格单元, 将 `getHeader` 的结果放在右边的表格单元。回顾一下, `Enumeration` 是 Java 中标准的接口; 它在 `java.util` 包中, 并且只包含两个方法: `hasMoreElements` 和 `nextElement`。

这个 servlet 还打印出主请求行的 3 个组成部分(方法、URI 和协议)。图 5.1 和图 5.2 列出 Netscape 和 Internet Explorer 中典型的结果。

清单 5.1 ShowRequestHeaders.java

```
package coreservlets;

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

/** Shows all the request headers sent on the current request.*/

public class ShowRequestHeaders extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Servlet Example: Showing Request Headers";
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<H1 ALIGN= CENTER>" + title + "</H1>\n" +
                    "<B>Request Method: </B>" +
                    request.getMethod() + "<BR>\n" +
                    "<B>Request URI: </B>" +
                    request.getRequestURI() + "<BR>\n" +
                    "<B>Request Protocol: </B>" +
                    request.getProtocol() + "<BR><BR>\n" +
                    "<TABLE BORDER=1 ALIGN= CENTER>\n" +
                    "<TR BGCOLOR=#FFAD00>\n" +
                    "<TH>Header Name<TH>Header Value");
        Enumeration headerNames = request.getHeaderNames();
        while(headerNames.hasMoreElements()) {
```

```

String headerName = (String)headerNames.nextElement();
out.println("<TR><TD>" + headerName);
out.println("    <TD>" + request.getHeader(headerName));
}
out.println("</TABLE>\n</BODY></HTML>");
}

/** Since this servlet is for debugging, have it
 * handle GET and POST identically.
 */
public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

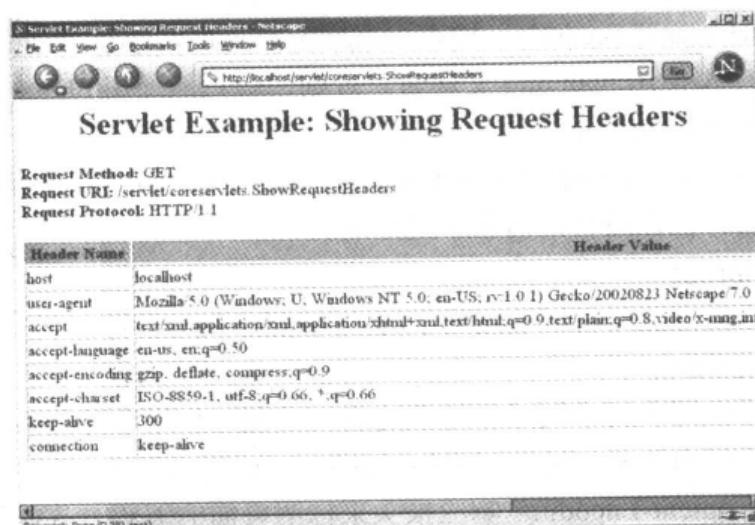


图 5.1 Windows 2000 上由 Netscape 7 发送的请求报头

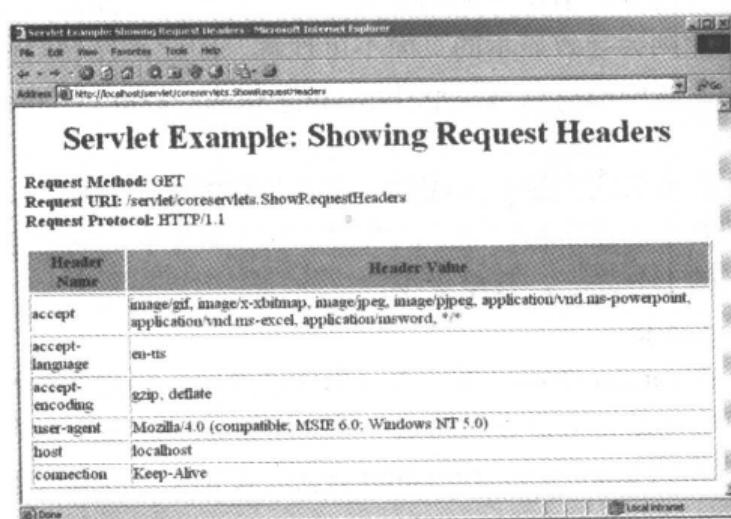


图 5.2 Windows 2000 上由 Internet Explorer 6 发送的请求报头

5.3 了解 HTTP 1.1 请求报头

对请求报头的访问，使得 servlet 能够执行许多优化，并提供大量特性，如果不能访问请求报头，要实现这些特性是不可能的。本节汇总了 servlet 最常使用的报头；这些报头以及其他报头的更多细节，请参阅 RFC 2616 中给出的 HTTP 1.1 规范。正式的 RFC 可以在许多地方得到；不过，最好先从 <http://www.rfc-editor.org/> 获取档案站点的当前清单。请注意，HTTP 1.1 支持的报头是 HTTP 1.0 的超集。

(1) Accept

这个报头指定浏览器或其他客户程序能够处理的 MIME 类型。那些能够以多种格式返回某种资源的 servlet，可以检查 Accept 报头来确定应该使用哪种格式。例如，PNG 格式的图像比 GIF 格式的图像在压缩上有一些优势，但不是所有的浏览器都支持 PNG 格式。如果您同时拥有同一图像的两种格式，您的 servlet 可以调用 `request.getHeader("Accept")`，检查 `image/png`，如果找到匹配项，则在生成的所有 IMG 元素中都使用 `blah.png` 文件名。否则，就只能使用 `blah.gif`。

关于常见 MIME 类型的名称及含义，请参见 7.2 节中的表 7.1。

要注意，Internet Explorer 5 和 6 存在一个 bug：在重新载入页面时，发送的 Accept 报头不正确。但在最初的请求中是正确的。

(2) Accept-Charset

这个报头标明浏览器可以使用的字符集(如 ISO-8859-1)。

(3) Accept-Encoding

这个报头详细列出客户端能够处理的编码类型。如果服务器接收到这个报头，则可以自由地使用任何一种所列出的格式对页面进行编码(一般为了降低传输时间)，同时发送 Content-Encoding 响应报头来标明页面的编码方式。这个编码类型完全不同于实际文档的 MIME 类型(在 Content-Type 响应报头中指明)，因为在浏览器确定如何处理内容之前这个编码是保留的。从另一方面说，如果使用了浏览器不理解的编码，则会导致显示的页面不可理解。因此，在使用任何类型的内容编码之前，一定要明确地检查 Accept-Encoding 报头。`gzip` 或 `compress` 是二种最常见的值。

在返回页面之前对它们进行压缩是一项有价值的服务，因为，一般说来，花在解码上的开销要小于在传输时间上的节省。在 5.4 节中，应用 gzip 压缩将下载时间减少到原来的十分之一，甚至更少。

(4) Accept-Language

这个报头，在 servlet 能够以多种语言生成结果时，列出客户程序首选的语言。这个报头的值应该是标准语言代码的一种，比如 `en`, `en-us`, `da` 等。详细信息请参见 RFC1766(首先到 <http://www.rfc-editor.org/> 获取 RFC 档案站点的最新列表)。

(5) Authorization

在访问密码保护的 Web 页面时，客户用这个报头来标识自己的身份。详细的信息，请参见本书第二卷中有关 Web 应用安全性的章节。

(6) Connection

这个报头标明客户是否能够处理持续性 HTTP 连接。持续性连接允许客户或其他浏览器在单个 socket 中读取多个文件(例如 HTML 文件及相关的几幅图像)，从而节省协商几个独立连接所需的开销。使用 HTTP 1.1 请求时，持续性连接是默认选项，如果使用旧式风格的连接，客户必须将这个报头的值指定为 close。在 HTTP 1.0 中，Keep-Alive 值的意思是应该使用持续性连接。

每个 HTTP 请求都会产生对 servlet 的新调用(即一个线程，调用 servlet 的 service 和 doXxx 方法)，不管请求是否为单独的连接。更确切地说，服务器只在读完 HTTP 请求之后，才会调用这个 servlet。这就意味着，servlet 需要与服务器进行协调，来处理持续性连接。所以，servlet 的工作只是使服务器能够使用持续性连接；servlet 通过设置 Content-Length 响应报头来做到这一点。详细信息，参见第 7 章。

(7) Content-Length

这个报头只适用于 POST 请求，用来给定 POST 数据的大小，以字节为单位。不需调用 request.getIntHeader("Content-Length")，只需简单地使用 request.getContentLength() 就可以得到这个报头的值；但是，表单数据的读取一般由 servlet 负责(参见第 4 章)，我们很少会显式地使用这个报头。

(8) Cookie

这个报头向服务器返回 cookie，这些 cookie 是之前由服务器发送给浏览器的。不要直接读取这个报头，因为这样做需要麻烦的低级分析；而要使用 request.getCookies。详细信息，请参见第 8 章。从技术上讲，cookie 不属于 HTTP 1.1。最初，它是 Netscape 的一项扩展，但现在已经得到广泛支持，包括 Netscape 和 Internet Explorer。

(9) Host

在 HTTP 1.1 中，浏览器和其他客户程序需要指定这个报头，它标明原始 URL 中给出的主机名和端口号。由于虚拟主机的广泛使用(一个计算机处理多个域名对应的网站)，服务器极有可能不能通过其他手段确定这个信息。这个报头不是 HTTP 1.1 新引入的，但在 HTTP 1.0 中它是可选的，并非是必需的。

(10) If-Modified-Since

这个报头标明，仅当页面在指定的日期之后发生更改的情况下，客户程序才希望获取该页面。如果没有更新的结果，则服务器发送 304 报头(Not Modified)。这个选项十分有用，因为使用它，浏览器可以缓存文档，只在它们发生更改时才通过网络重新载入它们。但是，servlet 不需要直接处理这个报头。取而代之，它们应该实现 getLastModified 方法，让系统自动处理修改日期。具体的例子，请参见 3.6 节中给出的彩票数字 servlet。

(11) If-Unmodified-Since

这个报头和 If-Modified-Since 正好相反：它规定仅当文档比指定的日期要旧时，操作才需要继续。一般来说，If-Modified-Since 用在 GET 请求中(“仅当文档比我缓存的版本要新时，才传送该文档”)，而 If-Unmodified-Since 用在 PUT 请求中(“仅当我生成这个文档之后，没有其他人对它做过更改时，才更新这个文档”)。这个

报头是 HTTP 1.1 新引入的报头。

(12) Referer

这个报头标明引用 Web 页面的 URL。例如，如果您在 Web 页面 1 单击指向 Web 页面 2 的链接，那么，在浏览器请求 Web 页面 2 时，就会将 Web 页面 1 的 URL 引入 Referer 报头。大多数主要的浏览器都会设置这个报头，因此，这是一种跟踪请求来源的有用方式。这项功能对于追踪将人们引到您的网站的广告客户很有帮助，可以依据引用网站的不同对内容做出细微的调整，可以确定用户是否首次访问您的应用，或者追溯网站的流量来自何方。对于最后一种情况，大多数人都依赖于 Web 服务器的日志文件，因为 Referer 一般都记录在日志文件中。尽管 Referer 报头十分有用，但也不要对它过度依赖，因为定制的客户程序可以轻易地篡改它。同时，要注意，由于 HTTP 最初的作者所犯的一个拼写错误，这个报头是 Referer，而不是期望的 Referrer。

最后要注意，某些浏览器(Opera)、广告过滤软件(Web Washer)和个人防火墙(Norton)屏蔽了这个报头。此外，即使在正常的情况下，仅当用户使用链接时才会设置这个报头。因而，不要忘记，任何情况下都要遵循对于所有报头都应该采用的方式：在使用报头之前检查它是否为 null。

详细情况以及具体的例子参见 5.6 节。

(13) User-Agent

这个报头标识生成请求的浏览器或其他客户程序，根据这个报头，可以针对不同类型的浏览器返回不同的内容。但是，在使用这个报头来应对 Web 浏览器时要注意，如果依赖于硬编码的浏览器版本清单及其相关特性，所产生的 servlet 代码不可靠且难以修改。只要有可能，就要尽量使用 HTTP 报头中给出的信息。例如，不要试图记住哪些平台上的哪种浏览器支持 gzip，而是简单地检查 Accept-Encoding 报头。

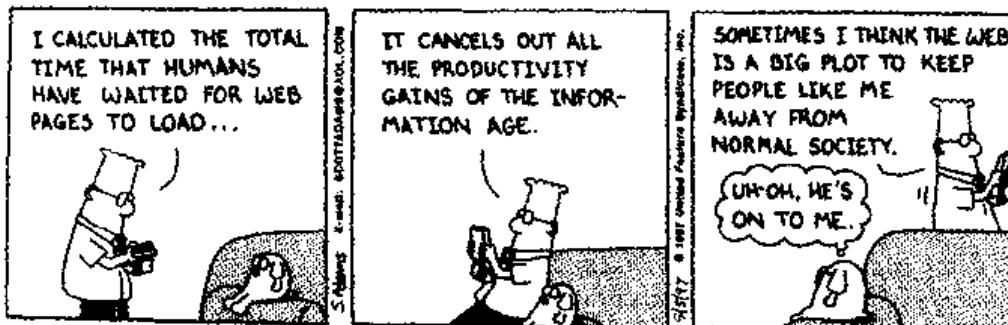
但是，User-Agent 报头对区分不同类别的客户程序却十分有用。例如，日本的开发人员可能会检查 User-Agent 是否为 Imode 移动电话(在这种情况下，他们会重定向到 chtml 页面)、Skynet 移动电话(在这种情况下，他们会重定向到 wml 页面)或 Web 浏览器(在这种情况下，他们会生成常规的 HTML)。

Internet Explorer 的大多数版本都会在 User-Agent 行中首先列出一个“Mozilla”(Netscape)版本，并以浏览器的实际版本作为补充。Opera 浏览器也会这样做。这种有意的混淆标识是为了与 JavaScript 兼容；JavaScript 的开发人员经常使用 User-Agent 报头来确定浏览器支持哪些 JavaScript 特性。因而，如果您希望将 Netscape 与 Internet Explorer 区分开来，您必须检查字符串“MSIE”或其他更为具体的内容，而非仅仅检查字符串“Mozilla”。还要注意，这个报头可以被容易地篡改，这使得那些根据这个报头来展示各种不同版本的浏览器的市场占有率的网站，也显得不那么可靠。

详细情况和具体的例子，参见 5.5 节。

5.4 发送压缩 Web 页面

gzip 文本压缩方案能够极大地减少 HTML(或纯文本)页面的大小。大多数最近的浏览器都知道如何处理 gzip 压缩后的内容，因此，服务器可以对文档进行压缩，使得在网络上传送的文档更小，之后浏览器会自动解压缩(不需要用户参与)并以正常的方式处理生成的结果。发送这种压缩后的内容，的确可以节省时间，因为在服务器端压缩文档所需的时间，以及之后在客户端解压缩所需的时间，一般会小于在下载上节省的时间，尤其是在使用拨号连接的情况下。



翻印获得 United Feature Syndicate, Inc 的许可

但是，尽管大多数最近的浏览器都支持这项功能，但并非全部。如果您将压缩后的内容发送给不支持这种功能的浏览器，那么浏览器根本不能显示该页面。幸运的是，支持这种功能的浏览器会设置 Accept-Encoding 请求报头，表示它们支持这种功能。支持对内容进行编码的浏览器包括 Netscape for Unix 的大多数版本，Internet Explorer for Windows 的大多数版本，以及 Windows 上 Netscape 4.7 及之后的版本。Netscape 在 Windows 上的早期版本和非 Windows 平台上的 Internet Explorer 一般不支持对内容进行编码。

清单 5.2 列出的 servlet 检查 Accept-Encoding 报头，向支持 gzip 编码的客户程序(由清单 5.3 中的 isGzipSupported 方法来决定)发送压缩的 Web 页面，向那些不支持这项功能的客户程序发送常规的 Web 页面。对结果(图 5.3)的压缩超过了 300 倍，并且，在使用拨号连接时能够数以 10 倍地提高速度。用 Netscape 和 Internet Explorer 在 28.8K 调制解调器连接上进行重复测试，压缩的页面平均的下载时间小于 5 秒，而非压缩页面一般都要花费超过 50 秒的时间。当使用更快的网络连接时，结果虽然没有那么惊人，但速度上的提高依旧很显著。Gzip 压缩是如此有用，因此我们后面提供一种过滤器，它可以对指定的 servlet 和 JSP 页面进行 gzip 压缩，但无需改变个别资源的实际代码。详细信息，参见本书第二卷中介绍 servlet 和 JSP 过滤器的章节。

重要提示

对于篇幅较长的文本页面，Gzip 压缩可以极大地降低下载时间。

由于对 gzip 格式的支持通过 `java.util.zip` 中的类内建在 Java 编程语言中，因此实现压缩比较简单直接。servlet 首先检查 Accept-Encoding 报头，检查它是否包含有关 gzip 的项。

如果支持，它使用 PrintWriter 封装 GZIPOutputStream，并指定 Content-Encoding 响应报头的值为 gzip。如果不支持 gzip，servlet 则使用正常的 PrintWriter，并省掉 Content-Encoding 报头。为了更容易地比较同一浏览器使用常规和压缩格式时的性能，我们还加入了一项功能，通过这项功能我们可以在 URL 尾部加上?disableGzip 来禁止压缩。

清单 5.2 LongServlet.java

```
package coreservlets;

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

/** Servlet with <B>long</B> output. Used to test
 * the effect of the gzip compression.
 */

public class LongServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");

        // Change the definition of "out" depending on whether
        // or not gzip is supported.
        PrintWriter out;
        if (GzipUtilities.isGzipSupported(request) &&
            !GzipUtilities.isGzipDisabled(request)) {
            out = GzipUtilities.getGzipWriter(response);
            response.setHeader("Content-Encoding", "gzip");
        } else {
            out = response.getWriter();
        }

        // Once "out" has been assigned appropriately, the
        // rest of the page has no dependencies on the type
        // of writer being used.
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
             \"Transitional//EN\\\">\\n";
        String title = "Long Page";
        out.println
            (docType +
             "<HTML>\\n" +
             "<HEAD><TITLE>" + title + "</TITLE></HEAD>\\n" +
             "<BODY BGCOLOR=\"#FDF5E6\\\">\\n" +
             "<H1 ALIGN=\"CENTER\\\">" + title + "</H1>\\n");
        String line = "Blah, blah, blah, blah, blah. " +
                     "Yadda, yadda, yadda, yadda.";
        for(int i=0; i<10000; i++) {
            out.println(line);
        }
        out.println("</BODY></HTML>");
        out.close(); // Needed for gzip; optional otherwise.
    }
}
```

清单5.3 GzipUtilities.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.zip.*;

/** Three small static utilities to assist with gzip encoding.
 * <UL>
 *   <LI>isGzipSupported: does the browser support gzip?
 *   <LI>isGzipDisabled: has the user passed in a flag
 *     saying that gzip encoding should be disabled for
 *     this request? (Useful so that you can measure
 *     results with and without gzip on the same browser).
 *   <LI>getGzipWriter: return a gzipping PrintWriter.
 * </UL>
 */

public class GzipUtilities {

    /** Does the client support gzip? */
    public static boolean isGzipSupported
        (HttpServletRequest request) {
        String encodings = request.getHeader("Accept-Encoding");
        return((encodings != null) &&
            (encodings.indexOf("gzip") != -1));
    }

    /** Has user disabled gzip (e.g., for benchmarking)? */
    public static boolean isGzipDisabled
        (HttpServletRequest request) {
        String flag = request.getParameter("disableGzip");
        return((flag != null) && (!flag.equalsIgnoreCase("false")));
    }

    /** Return gzipping PrintWriter for response. */
    public static PrintWriter getGzipWriter
        (HttpServletResponse response) throws IOException {
        return(new PrintWriter
            (new GZIPOutputStream
                (response.getOutputStream())));
    }
}
```

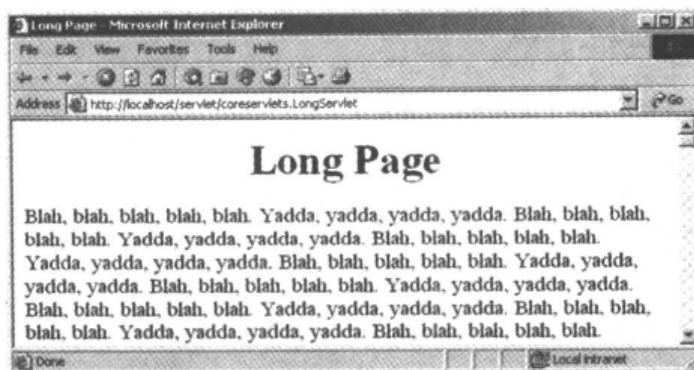


图 5.3 由于 Internet Explorer 6 的 Windows 版本支持 gzip，所以这个页面以压缩格式通过网络发送，并由浏览器自动重建，从而节省了大量的下载时间

5.5 区分不同的浏览器类型

User-Agent 报头标识发出请求的具体浏览器。尽管乍看起来这个报头的应用比较直观，但要注意几个细节：

- 仅仅在必需时才使用 User-Agent。

否则，您的代码将会包含浏览器的各种版本及其相关功能的表格，难以维护。例如，不要记录 Internet Explorer 5 的 Windows 版本支持 gzip 压缩但 MacOS 版本不支持，而是检查 Accept-Encoding 报头。不要记录哪种浏览器支持 Java，哪种不支持，而是使用 APPLET 标记，将代码放在<APPLET>和</APPLET>之间。

- 检查是否为 null。

不错，所有主要的浏览器版本都发送 User-Agent 报头。但是，这个报头并非 HTTP 1.1 规范的要求，一些浏览器允许您禁用它(例如 Opera)，而定制的客户程序(如 Web spider 和链接检验程序)可能根本就不使用这个报头。实际上，不管您正在处理哪种报头，只要使用 `request.getHeader` 返回的结果，就应该在使用前核实它不是 `null`。

- 区分 Netscape 和 Internet Explorer，要检查“MSIE”，而非“Mozilla”。

Netscape 和 Internet Explorer 都在该报头的开始处列出“Mozilla”，尽管 Mozilla 是类 Godzilla 的 Netscape 吉祥物。这个特征是为了与 JavaScript 兼容。

- 要注意，该报头可以假造。

一些浏览器允许用户修改这个报头的值。即使浏览器不允许这样做，用户总可以使用定制的客户程序。如果客户程序假造了这个报头，servlet 并不能区分这种情况。

清单 5.4 中的 servlet 向用户发送和浏览器相关的攻击性言论。为了简单起见，它假定用户使用的浏览器仅仅限于 Internet Explorer 和 Netscape 这两种浏览器。特别地，它假定任何 User-Agent 中包含“MSIE”的浏览器是 Internet Explorer，而任何不满足这个条件的浏览器则为 Netscape。图 5.4 和图 5.5 展示出相应的结果。

清单5.4 BrowserInsult.java

```
package coreservlets;

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

/** Servlet that gives browser-specific insult.
 * Illustrates how to use the User-Agent
 * header to tell browsers apart.
 */
public class BrowserInsult extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title, message;
        // Assume for simplicity that Netscape and IE are
        // the only two browsers
        String userAgent = request.getHeader("User-Agent");
        if ((userAgent != null) &&
            (userAgent.indexOf("MSIE") != -1)) {
            title = "Microsoft Minion";
            message = "Welcome, O spineless slave to the " +
                      "mighty empire.";
        } else {
            title = "Hopeless Netscape Rebel";
            message = "Enjoy it while you can. " +
                      "You <I>will</I> be assimilated!";
        }
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
             "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    message + "\n" +
                    "</BODY></HTML>");
    }
}
```

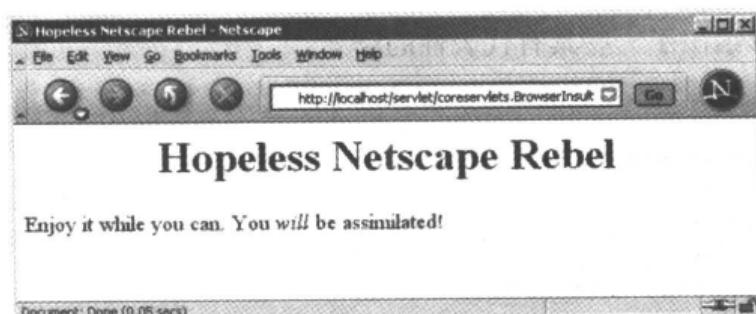


图 5.4 Netscape 用户浏览 BrowserInsult servlet 的结果

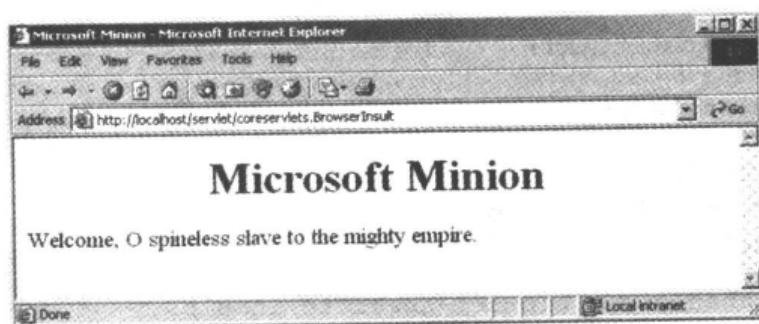


图 5.5 Internet Explorer 用户浏览 BrowserInsult servlet 的结果

5.6 依据客户的到达方式定制页面

Referer 报头指出，用户单击链接到达当前页面时所处页面的位置。如果用户直接输入页面的地址，那么浏览器就不会发送 Referer，同时 request.getHeader("Referer") 返回 null。

通过这个报头，您可以根据用户如何到达某个页面，对页面进行定制。例如，您可以使用这个报头完成下面的任务：

- 创建一个工作/职业网站，承袭链接到它的相关网站的外观感觉。
- 根据链接来自于防火墙内部还是外部，更改页面的内容。(但是，不要用这种技巧来保护您的应用，Referer 报头，和其他所有的报头一样，易于伪造。)
- 提供相应的链接，使用户能够返回之前的页面。
- 跟踪标题广告的有效性，或记录显示您的广告的不同网站的点击率。

清单 5.5 中列出的 servlet 使用 Referer 报头对它显示的图像进行定制。如果引用页的地址中包含字符串“JRun”，该 servlet 显示 Macromedia JRun 的 logo。如果地址中包含字符串“Resin”，该 servlet 显示 Cauchy Resin 的 logo。否则，servlet 显示 Apache Tomcat 的 logo。该 servlet 还显示引用页的地址。

清单 5.6 给出用来链接到这个 servlet 的 HTML 页面。我们创建 3 个相同的页面，分别命名为 JRun-Referer.html、Resin-Referer.html 和 Tomcat-Referer.html；该 servlet 使用引用页的名称(而非表单数据)，来区分这三者。回顾一下，HTML 页面放在 Web 应用的顶层目录中(或者其中的任意子目录)，而 servlet 的代码放在 WEB-INF/classes 中与包名匹配的子目录中。故而，以 Tomcat 和默认 Web 应用为例，HTML 页面放在 *install_dir/webapps/ROOT/request-headers/*，且使用形如 *http://hostname/request-headers/Xxx-Referer.html* 的 URL 进行访问。

图 5.6 到图 5.9 给出了一些具有代表性的结果。

清单 5.5 CustomizeImage.java

```
package coreservlets;

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

/** Servlet that displays referer-specific image.
 */

```

```

public class CustomizeImage extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String referer = request.getHeader("Referer");
        if (referer == null) {
            referer = "<I>none</I>";
        }
        String title = "Referring page: " + referer;
        String imageName;
        if (contains(referer, "JRun")) {
            imageName = "jrun-powered.gif";
        } else if (contains(referer, "Resin")) {
            imageName = "resin-powered.gif";
        } else {
            imageName = "tomcat-powered.gif";
        }
        String imagePath = "../request-headers/images/" + imageName;
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR="#FDF5E6">\n" +
                    "<CENTER><H2>" + title + "</H2>\n" +
                    "<IMG SRC=\"" + imagePath + "\">\n" +
                    "</CENTER></BODY></HTML>");
    }

    private boolean contains(String mainString,
                           String subString) {
        return (mainString.indexOf(subString) != -1);
    }
}

```

清单5.6 JRun-Referer.html(与Tomcat-Referer.html和Resin-Referer.html相同)

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Referer Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Referer Test</H1>
Click <A HREF="/servlet/core servlets.CustomizeImage">here</A>
to visit the servlet.
</BODY></HTML>

```

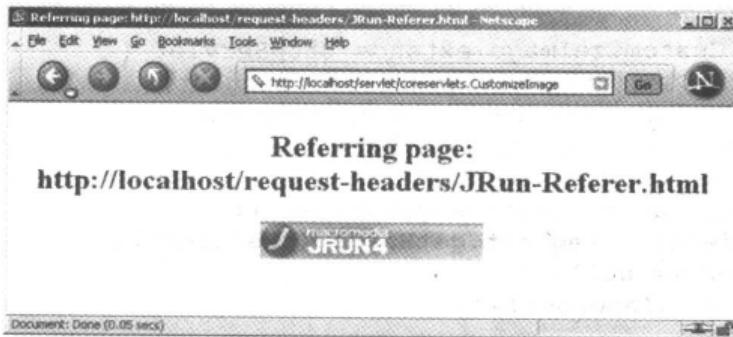


图 5.6 引用页的地址中含有字符串“JRun”时的 CustomizelImage servlet

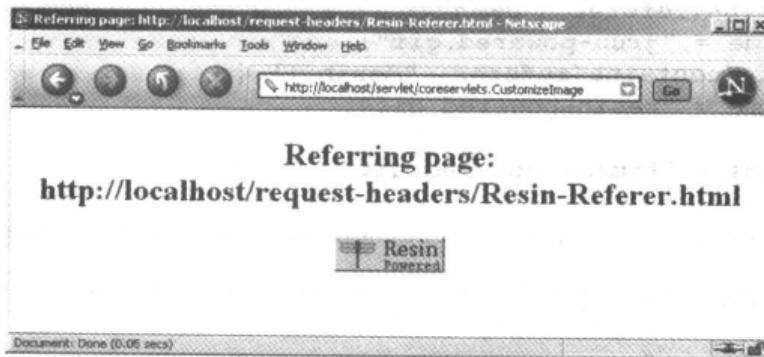


图 5.7 引用页的地址中含有字符串“Resin”时的 CustomizelImage servlet

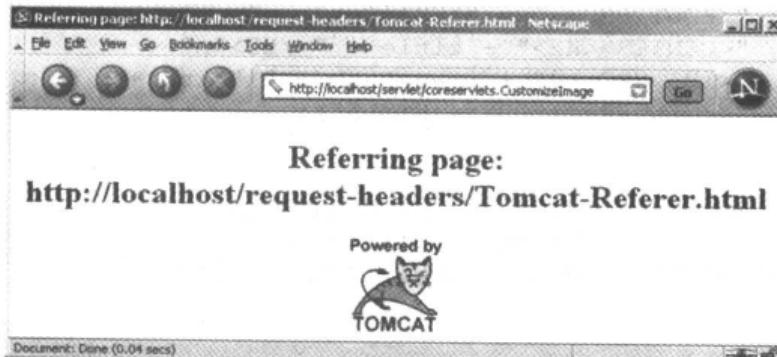


图 5.8 引用页的地址中既不含有“JRun”也没有“Resin”时的 CustomizelImage servlet

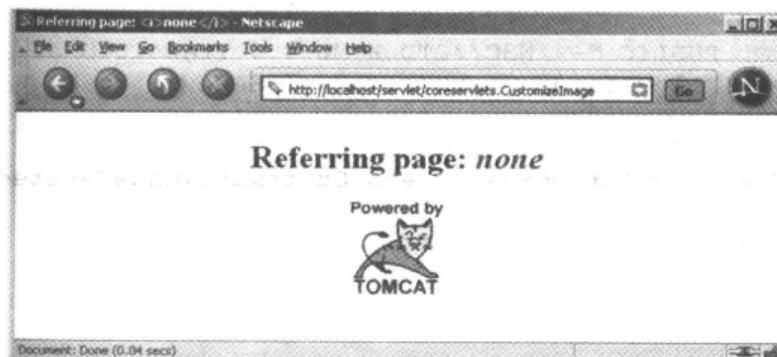


图 5.9 Referer 报头缺失时的 CustomizelImage servlet。在使用 Referer 报头时，一定要处理 getHeader 的结果为 null 的情况

5.7 标准 CGI 变量的访问

如果您在学习 servlet 之前，曾有过传统公共网关接口(Common Gateway Interface, CGI)编程的背景，那么，您可能习惯于“CGI 变量”的思想。存在一个有些混杂的集合，存储各种与当前请求相关的信息。有些基于 HTTP 请求中的行和报头(例如表单数据)，其他来自于 socket 本身(例如请求主机的名称和 IP 地址)，还有一些取自服务器的安装参数(例如：URL 到实际路径的映射)。

尽管将不同的数据来源区别对待会更有意义些，但有经验的 CGI 程序员可能觉得能够得到每个 CGI 变量在 servlet 中的等价物更为有用。如果您没有传统 CGI 方面的知识和经验，首先要多往好处想；servlet 比标准 CGI 更易于使用，更灵活，也更有效率。其次，浏览一下本节就可以了，只注意那些与输入的 HTTP 请求不直接相关的部分。特别地，要注意，您可以使用 `getServletContext().getRealPath` 得出 URI(URL 在主机和端口后的部分)的实际路径，还可以使用 `request.getRemoteHost` 和 `request.getRemoteAddress` 获取客户端的名称和 IP 地址。

5.7.1 servlet 中 CGI 变量的等价物

这个小节汇总了每个标准 CGI 变量的用途，以及从 servlet 中访问它的方式。我们假定 `request` 是提供给 `doGet` 和 `doPost` 方法的 `HttpServletRequest` 对象。

(1) AUTH_TYPE

如果提供了 `Authorization` 报头，这个变量给出指定的模式(basic 或 digest)。用 `request.getAuthType()` 访问它。

(2) CONTENT_LENGTH

仅适用于 POST 请求，这个变量存储发送数据的字节数，由 `Content-Length` 请求报头给出。技术上，由于 `CONTENT_LENGTH` CGI 变量是一个字符串，因此 servlet 中的等价物是 `String.valueOf(request.getContentLength())` 或 `request.getHeader("Content-Length")`。您可能只希望调用 `request.getContentLength()`，它返回一个整型数值。

(3) CONTENT_TYPE

`CONTENT_TYPE`，如果给出的话，指定附加数据的 MIME 类型。7.2 节中的表 7.1 列出了常见 MIME 类型的名称及含义。用 `request.getContentType()` 访问 `CONTENT_TYPE`。

(4) DOCUMENT_ROOT

`DOCUMENT_ROOT` 变量指定与 URL `http://host/` 对应的实际目录。用 `getServletContext().getRealPath("/")` 对它进行访问。同样，您可以使用 `getServletContext().getRealPath` 将任意 URI(即主机名和端口号之后的 URL 后缀) 映射到本地计算机的实际路径。

(5) HTTP_XXX_YYY

`HTTP_HEADER_NAME` 形式的变量是 CGI 程序访问任意 HTTP 请求报头的方式。

Cookie 报头成为 `HTTP_COOKIE`, User-Agent 成为 `HTTP_USER_AGENT`, Referer 成为 `HTTP_REFERER`, 依次类推。servlet 应该只使用 `request.getHeader`, 或 5.1 节中介绍的快捷方法。

(6) PATH_INFO

这个变量提供, servlet 地址之后, 查询数据之前, 附加到 URL 上的任何路径。以 `http://host/servlet/coreervlets.SomeServlet/foo/bar?baz=quux` 为例, 该路径信息是 `/foo/bar`。由于 servlet 不同于标准 CGI 程序, 能够直接与服务器对话, 所以, 它们不需要对路径信息进行特殊处理。路径信息可以作为常规表单数据的一部分进行发送, 之后用 `getServletContext().getRealPath` 进行转换。使用 `request.getPathInfo()` 访问 `PATH_INFO` 的值。

(7) PATH_TRANSLATED

`PATH_TRANSLATED` 给出映射到服务器实际路径的路径信息。同样, 由于 servlet 可以调用 `getServletContext().getRealPath` 将不完整的 URL 转换成实际的路径, 因此不需要对路径信息特殊对待。由于 CGI 程序的运行与服务器完全隔离, 因此这项转换在标准 CGI 程序中是不可能实现的。通过 `request.getPathTranslated()` 访问这个变量。

(8) QUERY_STRING

对于 GET 请求, 这个变量以单个字符串的形式给出附加的数据, 这些数据依旧保持 URL 编码之后的状态。在 servlet 中很少会用到原始数据; 相反, 一般都使用 `request.getParameter` 访问个别参数, 如 5.1 节所述。但是, 如果您确实需要原始数据, 那么, 可以使用 `request.getQueryString()` 得到它。

(9) REMOTE_ADDR

这个变量指出发出请求的客户机的 IP 地址, 类型为 `String`(例如“`198.137.241.30`”)。通过调用 `request.getRemoteAddr()` 对它进行访问。

(10) REMOTE_HOST

`REMOTE_HOST` 给出发出请求的客户机的完全限定域名(例如 `whitehouse.gov`)。如果不能确定域名, 则返回 IP 地址。您可以使用 `request.getRemoteHost()` 访问这个变量。

(11) REMOTE_USER

如果提供了 `Authorization` 报头, 并经过服务器自身的解码, 那么 `REMOTE_USER` 变量给出用户相关的部分, 它对于受保护网站的会话跟踪比较有用。用 `request.getRemoteUser` 访问它。在 servlet 中直接解码 `Authorization` 信息的内容, 参见本书第二卷中有关 Web 应用安全的章节。

(12) REQUEST_METHOD

这个变量规定 HTTP 请求的类型, 一般为 `GET` 或 `POST`, 但偶而也可能是 `HEAD`, `PUT`, `DELETE`, `OPTIONS` 或 `TRACE`。由于每个请求类型一般都由不同的 servlet 方法(`doGet`, `doPost` 等)处理, 因此 servlet 很少需要直接检查 `REQUEST_METHOD`。一个例外是 `HEAD`, 它由 `service` 方法自动处理, 返回 `doGet` 方法使用的任何报头和状态代码。通过 `request.getMethod()` 访问这个变量。

(13) SCRIPT_NAME

这个变量给出 servlet 的路径，相对于服务器的根目录。它可以通过 `request.getServletPath()` 访问。

(14) SERVER_NAME

`SERVER_NAME` 给出服务器计算机的主机名。可以使用 `request.getServerName()` 对它进行访问。

(15) SERVER_PORT

这个变量存储服务器正在侦听的端口。技术上，`servlet` 的等价物是 `String.valueOf(request.getServerPort())`，它返回 `String`。一般地，您只希望使用 `request.getServerPort()`，这个方法返回 `int`。

(16) SERVER_PROTOCOL

`SERVER_PROTOCOL` 标明在请求的行中使用的协议名称和版本(例如 `HTTP/1.0` 或 `HTTP/1.1`)。通过调用 `request.getProtocol()` 对它进行访问。

(17) SERVER_SOFTWARE

这个变量给出 Web 服务器的标识信息。通过 `getServletContext().getServerInfo()` 对它进行访问。

5.7.2 一个显示 CGI 变量的 servlet

清单 5.7 给出的 servlet 创建一个表格，列出除 `HTTP_XXX_YYY` 以外所有 CGI 变量的值，`HTTP_XXX_YYY` 不过是 5.3 中介绍的 HTTP 请求报头而已。图 5.10 给出了典型请求的结果。

清单5.7 ShowCGIVariables.java

```
package coreservlets;

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

/** Creates a table showing the current value of each
 *  of the standard CGI variables.
 */

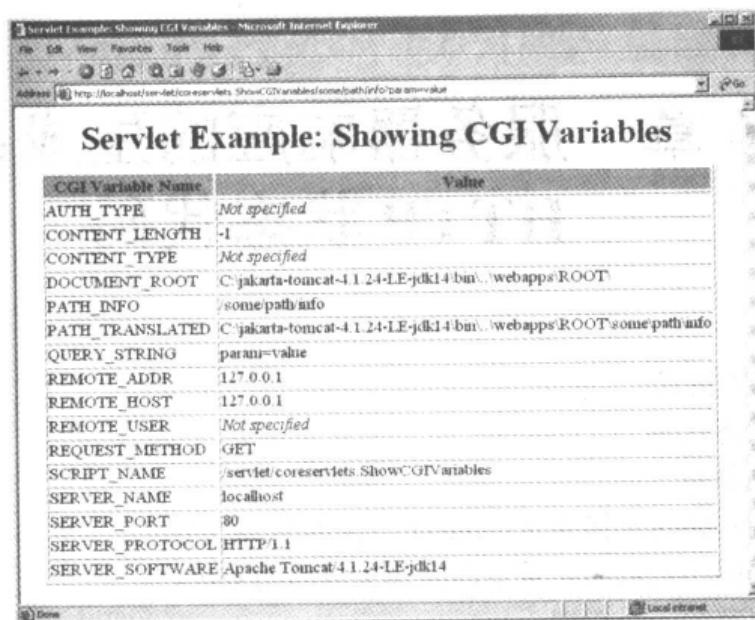
public class ShowCGIVariables extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String[][] variables =
        { { "AUTH_TYPE", request.getAuthType() },
          { "CONTENT_LENGTH",
            String.valueOf(request.getContentLength()) },
          { "CONTENT_TYPE", request.getContentType() },
          { "DOCUMENT_ROOT",
            getServletContext().getRealPath("/") },
          { "GATEWAY_INTERFACE", "CGI/1.1" },
          { "HTTP_ACCEPT", "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8" },
          { "HTTP_ACCEPT_CHARSET", "UTF-8;q=1.0,*;q=0.8" },
          { "HTTP_ACCEPT_ENCODING", "gzip;q=1.0,deflate;q=0.6,identity;q=0.3" },
          { "HTTP_ACCEPT_LANGUAGE", "zh-CN,zh;q=1.0,en;q=0.8" },
          { "HTTP_HOST", "www.coreervlets.com" },
          { "HTTP_REFERER", null },
          { "HTTP_USER_AGENT", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.122 Safari/537.36" },
          { "PATH_INFO", "/" },
          { "PATH_TRANSLATED", "C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\HelloWorld\index.html" },
          { "QUERY_STRING", null },
          { "REMOTE_ADDR", "127.0.0.1" },
          { "REMOTE_HOST", "127.0.0.1" },
          { "REMOTE_USER", null },
          { "REQUEST_METHOD", "GET" },
          { "SCRIPT_NAME", "index.html" },
          { "SERVER_NAME", "www.coreervlets.com" },
          { "SERVER_PORT", "80" },
          { "SERVER_SOFTWARE", "Apache/2.4.41 (Win32) OpenSSL/1.1.1g PHP/8.0.12" }
        };
        out.println("<table border='1'>");
        out.println("<tr><th>Variable</th><th>Value</th></tr>");
        for (String[] variable : variables) {
            out.println("<tr><td>" + variable[0] + "</td><td>" + variable[1] + "</td></tr>");
        }
        out.println("</table>");
    }
}
```

```
{ "PATH_INFO", request.getPathInfo() },
{ "PATH_TRANSLATED", request.getPathTranslated() },
{ "QUERY_STRING", request.getQueryString() },
{ "REMOTE_ADDR", request.getRemoteAddr() },
{ "REMOTE_HOST", request.getRemoteHost() },
{ "REMOTE_USER", request.getRemoteUser() },
{ "REQUEST_METHOD", request.getMethod() },
{ "SCRIPT_NAME", request.getServletPath() },
{ "SERVER_NAME", request.getServerName() },
{ "SERVER_PORT",
    String.valueOf(request.getServerPort()) },
{ "SERVER_PROTOCOL", request.getProtocol() },
{ "SERVER_SOFTWARE",
    getServletContext().getServerInfo() }
};

String title = "Servlet Example: Showing CGI Variables";
String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
     \"Transitional//EN\"\">\n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
    "<CENTER>\n" +
    "<H1>" + title + "</H1>\n" +
    "<TABLE BORDER=1>\n" +
    "  <TR BGCOLOR=\"#FFAD00\">\n" +
    "    <TH>CGI Variable Name<TH>Value");
for(int i=0; i<variables.length; i++) {
    String varName = variables[i][0];
    String varValue = variables[i][1];
    if (varValue == null)
        varValue = "<I>Not specified</I>";
    out.println("  <TR><TD>" + varName + "<TD>" + varValue);
}
out.println("</TABLE></CENTER></BODY></HTML>");
}

/** POST and GET requests handled identically. */

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



A screenshot of Microsoft Internet Explorer displaying a table titled "Servlet Example: Showing CGI Variables". The table lists various CGI variables and their values. The columns are "CGI Variable Name" and "Value".

CGI Variable Name	Value
AUTH_TYPE	Not specified
CONTENT_LENGTH	-1
CONTENT_TYPE	Not specified
DOCUMENT_ROOT	C:\jakarta-tomcat-4.1.24-LE-jdk14\bin\..\webapps\ROOT
PATH_INFO	/some/path/info
PATH_TRANSLATED	C:\jakarta-tomcat-4.1.24-LE-jdk14\bin\..\webapps\ROOT\some\path\info
QUERY_STRING	param1=value
REMOTE_ADDR	127.0.0.1
REMOTE_HOST	127.0.0.1
REMOTE_USER	Not specified
REQUEST_METHOD	GET
SCRIPT_NAME	/servlet/core servlets>ShowCGIVariables
SERVER_NAME	localhost
SERVER_PORT	80
SERVER_PROTOCOL	HTTP/1.1
SERVER_SOFTWARE	Apache Tomcat/4.1.24-LE-jdk14

图 5.10 典型请求的标准 CGI 变量

第 6 章 服务器响应的生成： HTTP 状态代码

本章的主题：

- HTTP 响应的格式
- 如何设置状态代码
- 状态代码的作用
- 进行重定向和跳转到错误页面(error page)的快捷方法
- 根据浏览器的不同将用户重定向到相关页面的 servlet。
- 各种搜索引擎的一个前端

前一章中我们已经看到，来自浏览器或其他客户程序的请求，由一个 HTTP 命令(一般为 GET 或 POST)、0 或多个请求报头(HTTP 1.1 中为一个或多个，因为 Host 是必需的)、一个空行以及一些查询数据(POST 请求)。典型请求的形式如下：

```
GET /servlet/SomeName HTTP/1.1
Host: ...
Header2: ...
...
HeaderN:
(Blank Line)
```

Web 服务器对请求的响应，一般由一个状态行、一些响应报头、一个空行和相应的文档构成。典型响应的形式如下：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
(Blank Line)
<!DOCTYPE ...>
<HTML>
<HEAD>...</HEAD>
<BODY>
...
</BODY></HTML>
```

状态行由 HTTP 版本(前面的示例中为 HTTP/1.1)、一个状态代码(整数，示例中为 200)、以及一段对应状态代码的简短消息(例子中为 OK)组成。大多数情况下，除了指定后面文档 MIME 类型的 Content-Type 报头之外，其他的报头都是可选的。虽然大多数响应都含有个文档，但也存在没有文档的情况。例如，对 HEAD 请求的响应就不应该包括文档，同时，大部分的状态代码是用来标明失败或重定向(因而要么不包括文档，要么只包括一个简短的错误消息文档)。

通过操作状态行和响应报头，servlet 可以执行多种重要任务。例如，它们可以将用户

转送到其他网站；标明附加的文档是图像、Adobe Acrobat 文件或 HTML 文件；告知用户访问该文档需要使用密码；依次类推。本章对最重要的代码进行汇总，并介绍使用它们能够完成哪些任务；随后的一章论述响应报头。

6.1 状态代码的指定

刚才已经介绍过，HTTP 响应的状态行由 HTTP 版本、一个状态代码和一段相关的消息组成。由于消息直接与状态代码相关，并且，HTTP 的版本是由服务器来决定的，故而，servlet 需要做的只是设置状态代码。系统自动设置的代码是 200，因此，一般地，servlet 根本不需要指定状态代码。如果确实希望设置状态代码，则可以使用 `response.setStatus`，`response.sendRedirect` 或 `response.sendError` 方法。

6.1.1 设置任意状态代码：`setStatus`

设置任意的状态代码，需要使用 `HttpServletResponse` 的 `setStatus` 方法。如果响应的状态代码比较特殊，并且伴有相关的文档内容，那么一定要在用 `PrintWriter` 实际返回任何内容之前调用 `setStatus`。之所以需要这样做，是因为 HTTP 请求由状态行、一个或多个报头、一个空行、以及实际的文档按照此处列出的次序组成。并没有相关的规范规定 servlet 一定要缓冲文档，因而，必须在使用 `PrintWriter` 之前设置状态代码，或者在设置状态代码时，仔细核实缓冲区尚未被清除，内容尚未实际发往浏览器。

核心方法

在向客户程序发送任何文档内容之前设置状态代码。

`setStatus` 方法以一个整数(状态代码，int 类型)为参数，但为了易读性，同时也是为了避免输入错误，尽量不要使用明确的数字，而要使用 `HttpServletResponse` 中定义的常量。每个常量的名字都来自于每个常量所对应的标准 HTTP 1.1 消息，全部大写并添加 SC 前缀 (Status Code，状态代码)，空格转化为下划线。因此，由于状态代码 404 所对应的消息是 Not Found，所以 `HttpServletResponse` 中与之等同的常量是 `SC_NOT_FOUND`。但有一个小小的例外：代码 302 所对应的常量源自 HTTP 1.0 定义的消息(Moved Temporarily)，而非 HTTP 1.1 消息(Found)。

6.1.2 设置 302 和 404 状态代码：`sendRedirect` 和 `sendError`

虽然设置状态代码的通用方法是直接调用 `response.setStatus(int)`，但是，`HttpServletResponse` 专为两种常见的情况提供了快捷方法。需要注意的是，这两个方法都抛出 `IOException` 异常，而 `setStatus` 不会。由于 `doGet` 和 `doPost` 方法已经抛出 `IOException`，因此，仅当将响应对象传递给其他方法时，才需要注意这种差异。

- `public void sendRedirect(String url)`

状态代码 302 命令浏览器连接到新的位置。`sendRedirect` 方法生成 302 响应以及 Location 报头，给出新文档的 URL。这个 URL 即可以是绝对 URL，也可以是相

对 URL; 系统在将它们放入 Location 报头之前, 自动将相对 URL 转换成绝对 URL。

- **public void sendError(int code, String message)**

状态代码 404 用于服务器没有找到文档的情况。sendError 方法发送状态代码(一般为 404)以及一小段简短的消息, 这段消息被自动安排到 HTML 文档中发送给客户。

设置状态代码并不意味着省略文档。例如, 绝大部分服务器都会自动为 404 响应生成一小段 File Not Found 消息, servlet 可以对这个响应进行定制。同样, 要牢记, 如果需要发送输出, 必须首先调用 setStatus 或 sendError。

6.2 HTTP 1.1 状态代码

本节介绍 servlet 与 HTTP 1.1 客户程序对话过程中可以使用的一些最重要的状态代码, 同时给出每种代码相关的标准消息。充分理解这些代码, 能够极大地提高 servlet 的功能, 因此, 您至少应该略读这些描述, 看看有哪些选项可供您支配。可以在准备使用这些功能时再回来仔细阅读相关的细节。

完整的 HTTP 1.1 规范在 RFC 2616 中给出。一般地, 可以到 <http://www.rfc-editor.org/>, 通过给出的最新 RFC 档案站点, 在线访问这些 RFC; 但由于这份 RFC 来自于万维网联盟(W3C), 因而, 您可以直接到 <http://www.w3.org/Protocols> 查阅相关文档。要注意那些 HTTP 1.1 中新加入的代码, 因为一些浏览器只支持 HTTP 1.0。应该只将新的代码发送给支持 HTTP 1.1 的客户程序, 可以通过检查 request.getRequestProtocol 检验客户程序是否支持 HTTP 1.1。

本节的剩余部分将讲述 HTTP 1.1 中可用的特定的状态代码。这些代码分为 5 类:

- **100-199**

100 到 199 间的代码都是信息性的, 标示客户应该采取的其他动作。

- **200-299**

200 到 299 间的值表示请求成功。

- **300-399**

300 到 399 间的值用于那些已经移走的文件, 常常包括 Location 报头, 指出新的地址。

- **400-499**

400 到 499 间的值表明由客户引发的错误。

- **500-599**

500 到 599 间的代码表示由服务器引发的错误。

HttpServletResponse 中代表各种代码的常量都来源于与代码相关的标准消息。在 servlet 中, 一般只通过这些常量引用状态代码。例如, 一般会使用 response.setStatus(response.SC_NO_CONTENT), 而非 response.setStatus(204), 因为后者对代码的阅读者来说比较难懂, 且容易产生输入错误。但要注意, 服务器可以对消息进行细微的更改, 客户程序应该只关注数字值。例如, 您或许会看到服务器返回状态行 HTTP/1.1 200 Document Follows, 而非 HTTP/1.1 200 OK。

(1) **100(Continue, 继续)**

如果服务器接收到值为 100-continue 的 Expect 请求报头, 这表示客户程序在询问

是否可以在随后的请求中发送附加文档。这种情况下，服务器应该以状态 100(SC_CONTINUE) 回应，告诉客户程序继续下去，或者使用 417(SC_EXPECTATION_FAILED) 告诉浏览器它不接受该文档。这是 HTTP 1.1 新引入的状态代码。

(2) 200(OK, 一切正常)

200(SC_OK) 表示一切正常；如果是 GET 和 POST 请求，则文档就跟在后面。对于 servlet，这是默认值；如果没有调用 setStatus，那么默认值就是 200。

(3) 202(Accepted, 已接受)

202(SC_ACCEPTED) 告诉客户，请求已经接受，但处理尚未完成。

(4) 204(No Content, 没有新文档)

状态代码 204(SC_NO_CONTENT) 要求浏览器继续显示之前的文档，因为没有新的文档。如果用户周期性地单击 Reload 按钮来重新载入页面，那么这个行为就比较有用，您可以确定前面的页面已经是最新的。

(5) 205(Reset Content, 重置内容)

值 205(SC_RESET_CONTENT) 表示没有新的文档，但浏览器应该重置文档视图。因此，这个状态代码用来指示浏览器清除表单的字段。它是 HTTP 1.1 新引入的状态代码。

(6) 301(Moved Permanently, 被永久移动)

301(SC_MOVED_PERMANENTLY) 状态表示所请求的文件已被移往别处；文档的新 URL 在 Location 响应报头中给出。浏览器应该依据这个链接跳转到新的 URL。

(7) 302(Found, 找到)

这个值类似于 301，只是原则上应该将 Location 报头给出的 URL 看作是非永久性的临时替代。实践中，大多数浏览器都等同地对待 301 和 302。注意：在 HTTP 1.0 中，该消息是 Moved Temporarily(被临时移动)，而非 Found；并且，HttpServletResponse 中对应的常量是 SC_MOVED_TEMPORARILY，而非预期的 SC_FOUND。

重点提示

表示 302 的常量是 SC_MOVED_TEMPORARILY，不是 SC_FOUND。

由于浏览器会自动转到 Location 响应报头中给出的 URL，故而状态代码 302 十分有用。要注意，浏览器立即重新连接到新的 URL；并不显示任何中间输出。这种行为将重定向(redirect)和刷新(refresh)区别开来。刷新时，浏览器会临时性地显示一个中间页面(详细情况参见下一章中有关 Refresh 报头的内容)。重定向时，由另外的网站生成最后的结果，而非该 servlet 自身。那么，为什么还要使用 servlet 呢？重定向主要用于下述任务：

- 计算目的地。

如果您预先知道用户的最终目的地，那么，使用超文本链接或 HTML 表单可以直接将用户转送到该处。但是，如果在确定应该到什么地方获取必需结果

时，需要检查相关的数据，那么重定向就显得十分有用。例如，您或许希望将用户转送到提供证券信息的标准网站，但在决定将用户转送到哪里(纽约证券交易所、纳斯达克或美国以外的网站)之前，首先需要检查证券代码。

- **跟踪用户的行为。**

如果直接将含有超文本链接(链接到其他网站)的页面发送给用户，则没有办法知道他们是否真的点击了该链接。但是，在分析提供给用户的不同链接的有用性时，这个信息可能会极为重要。因此，可以不将直接链接发送给客户，而是将指向您自己网站的链接发送给他们，这样，您的网站就可以先记录一些信息，再将他们重定向到实际的网站。例如，一些搜索引擎，就使用这种技巧来确定它们显示的结果中哪些最为常用。

- **执行边界效应。**

如果希望将用户转送到特定的站点，但在此之前需要在用户的浏览器中设置一个 cookie，应该怎么做呢？没问题：同时返回 Set-Cookie 响应报头(通过 response.addCookie——参见第 8 章)和 302 状态代码(通过 response.sendRedirect)。

302 状态代码十分有用，实际上，存在一个专门的方法，sendRedirect。与使用 response.setStatus(response.SC_MOVED_TEMPORARILY) 及 response.setHeader("Location", url) 相比，使用 response.sendRedirect(url) 有许多优点。首先，它更简短，使用也更容易。其次，使用 sendRedirect 时，servlet 自动构建含有该链接的页面，引导那些不能自动跟从重定向信息的老版本浏览器。最后，sendRedirect 还可以处理相对 URL，自动将它们转换成对应的绝对 URL。

技术上，一般认为仅当最初的请求为 GET 时，浏览器才应该自动执行重定向。详细信息，参见状态代码 307 的相关论述。

(8) 303(See Other, 检查其他文档)

303(SC_SEE_OTHER)状态类似于 301 和 302，除非初始的请求为 POST，否则应该用 GET 来读取新的文档(在 Location 报头中给出)。参见状态代码 307。这是 HTTP 1.1 新引入的状态代码。

(9) 304(Not Modified, 未发生更改)

在客户已经拥有缓存的文档时，它可以通过提供 If-Modified-Since 报头来执行条件请求，表示仅当文档在指定的日期之后发生改变时，才希望读取该文档。

304(SC_NOT_MODIFIED)表示缓存的版本是最新的，客户程序应该使用它。否则，服务器应该返回所请求的文档，并设置正常(200)状态代码。servlet 一般不应该直接设置这个状态代码。它们应该实现 getLastModified 方法，由默认的 service 方法基于这个修改日期处理条件请求。具体的例子参见 3.6 节中的 LotteryNumbers。

(10) 307(Temporary Redirect, 临时重定向)

浏览器对 307 状态的处理规则与 302 状态相同。之所以将值 307 引入到 HTTP 1.1 中，是因为，甚至在最初的消息是 POST 的情况下，许多浏览器依旧错误地跟随 302 响应中的重定向信息。浏览器应该只在接收到 303 响应状态时才跟从 POST 请求的重定向信息。引入这个新状态是为了去除二义性：如果接收到 303 响应，

则继续进行 GET 和 POST 请求的重定向；如果接收到 307 响应，对于 GET 请求的重定向，则继续进行，但对于 POST 请求的重定向，则不再继续下去。这是 HTTP 1.1 新引入的状态代码。

(11) **400(Bad Request, 错误请求)**

400(SC_BAD_REQUEST)状态表明客户请求中含有语法错误。

(12) **401(Unauthorized, 未授权)**

401(SC_UNAUTHORIZED)表示客户程序试图访问密码保护的页面，但在请求的 Authorization 报头中没有正确的身份标识信息。响应必须包括 WWW-Authenticate 报头。详细信息，参见本书第二卷中有关规划 Web 应用安全的章节。

(13) **403(Forbidden, 资源不可用)**

状态代码 403(SC_FORBIDDEN)表示服务器拒绝提供相关的资源，不管是否拥有授权。这个状态常常是由服务器上文件或目录的许可权限导致的结果。

(14) **404(Not Found, 未找到)**

404(SC_NOT_FOUND)状态告诉客户程序，在给定的地址找不到任何资源。这个值是标准的“no such page”响应。由于这个响应太过常用和有效，因此在 HttpServletResponse 类中为它提供了一个专门的方法：sendError("message")。与 sendStatus 相比，sendError 的优点是：使用 sendError 时，服务器自动生成显示错误消息的错误页面。404 错误不应该只是声明“对不起，找不到所需的页面”。相反，它们可以给出相关的信息，说明为什么找不到该页面，或者提供查找框或其他可供查看的位置。www.microsoft.com 和 www.ibm.com 等网站所提供的错误页面就是不错的例子——它们都很有用(要看到它们，只需杜撰一个上述网站中不存在的 URL)。实际上，有一个网站专门致力于好的、坏的、丑陋的和古怪的 404 错误页面：<http://www.plinko.net/404/>。我们还发现 <http://www.plinko.net/404/links.asp?type=cat&key=13>(有趣的 404 错误消息)十分有趣。

但遗憾的是，Internet Explorer 版本 5 及其随后版本的默认行为是忽略送回的错误页面，取而代之，它显示自己的静态(同时也没有什么用处)错误消息，即使这样做明显违背了 HTTP 规范。如果要关闭这项设置，请到 Tools(【工具】)菜单，选择 Internet Options(【Internet 选项】)，选取 Advanced(【高级】)标签，确保“Show friendly HTTP error messages(【显示友好的 HTTP 错误信息】)”复选框没有选择。很遗憾，只有极少数用户知道这项设置，因此，这项“特性”使得 Internet Explorer 的大部分用户看不到您返回的信息性消息。其他主要的浏览器和 Internet Explorer 版本 4 都可以正确地显示服务器生成的错误页面。

警告

默认情况下，Internet Explorer 版本 5 和其后的版本都忽略服务器生成的错误页面，这样做是不恰当的。

幸运的是，单独的 servlet 很少构建它们自己的 404 错误页面。更为常见的方式是为整个 Web 应用设置错误页面；详细信息参见 2.11 节。

(15) 405(Method Not Allowed, 方法不允许)

405(SC_METHOD_NOT_ALLOWED)值表示, 这个特定资源不允许使用该请求方法(GET, POST, HEAD, PUT, DELETE 等)。这是 HTTP 1.1 新引入的状态代码。

(16) 415(Unsupported Media type, 不支持的媒体类型)

值 415(SC_UNSUPPORTED_MEDIA_TYPE)表明, 服务器不知道如何处理请求附加文件的类型。这是 HTTP 1.1 新引入的状态代码。

(17) 417(Expectation Failed, 期望不能满足)

如果服务器接收到值为 100-continue 的 Expect 请求报头, 则表示客户程序在询问是否可以在后续的请求中发送附加的文档。这种情况下, 服务器要么用这个状态(417)回应, 告诉浏览器它不接受该文档, 要么使用 100(SC_CONTINUE)告诉客户程序继续进行。这是 HTTP 1.1 新引入的状态代码。

(18) 500(Internal Server Error, 服务器内部错误)

500(SC_INTERNAL_SERVER_ERROR)是通用的表示“服务器陷入混乱”的状态代码。它常常是由于 CGI 程序或(但愿不会如此!)servlet 崩溃或返回格式不正确的报头而造成的。

(19) 501(Not Implemented, 未实现)

501(SC_NOT_IMPLEMENTED)状态通知客户程序, 服务器不支持能够满足该请求的功能。它用于表示服务器不支持客户程序发送的命令, 比如 PUT。

(20) 503(Service Unavailable, 服务不可用)

状态代码 503(SC_SERVICE_UNAVAILABLE)表示, 由于维护工作或超负荷工作, 服务器不能做出响应。例如, 如果线程池或数据库连接池当前被全部占用, 那么 servlet 可能就要返回这个报头。服务器可以提供 Retry-After 报头, 告诉客户程序何时重试。

(21) 505(HTTP Version Not Supported, 不支持的 HTTP 版本)

505(SC_HTTP_VERSION_NOT_SUPPORTED)代码表示服务器不支持请求行中给出的 HTTP 版本。这是 HTTP 1.1 新引入的状态代码。

6.3 将用户重定向到浏览器相关页面的 servlet

我们在第 5 章中曾经论述过, User-Agent 请求报头指出生成请求的具体浏览器(或移动电话、其他客户程序), 并且, 大多数主要的浏览器在它们的 User-Agent 报头中含有字符串 Mozilla, 但只有 Microsoft Internet Explorer 包含字符串 MSIE。

清单 6.1 给出的 servlet 根据 User-Agent 请求报头的内容, 将 Internet Explorer 的用户转送到 Netscape 的主页, 将所有其他用户转送到 Microsoft 的主页。这个 servlet 通过使用 sendRedirect 方法向浏览器发送 302 状态代码和 Location 响应报头, 来完成这项任务。图 6.1 和图 6.2 分别给出 Internet Explorer 和 Netscape 所对应的结果。

清单 6.1 WrongDestination.java

```
package coreservlets;
```

```

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

/** Servlet that sends IE users to the Netscape home page and
 * Netscape (and all other) users to the Microsoft home page.
 */

public class WrongDestination extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        String userAgent = request.getHeader("User-Agent");
        if ((userAgent != null) &&
            (userAgent.indexOf("MSIE") != -1)) {
            response.sendRedirect("http://home.netscape.com");
        } else {
            response.sendRedirect("http://www.microsoft.com");
        }
    }
}

```

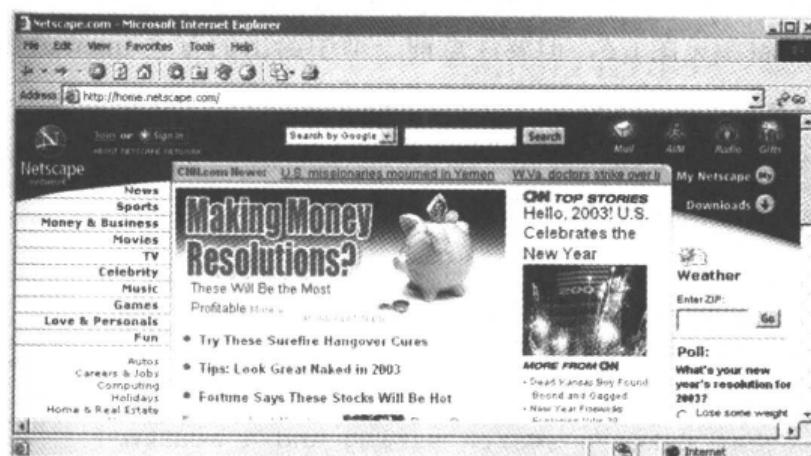


图 6.1 Internet Explorer 中访问 `http://host/servlet/coreservlets.WrongDestination` 的结果



图 6.2 Netscape 中访问 `http://host/servlet/coreservlets.WrongDestination` 的结果

6.4 各种搜索引擎的一个前端

假定您想制作一个提供“一站式搜索”的网站，允许用户勿需记住许多不同的 URL，就可以搜索任何最流行的搜索引擎。您希望让用户输入查询，选择搜索引擎，然后将用户转送到搜索引擎的结果页面。如果用户省略搜索关键字，或者没有选择搜索引擎，此时就不知道应该将用户重定向到哪个网站，因此，您需要显示一个错误页面，告知用户这种情况。

清单 6.2(SearchEngines.java)列出的 servlet，通过使用 302(Found)和 404(Not Found)状态代码完成这些任务——除 200 以外两个最常用的状态代码。302 代码可以通过 HttpServlet Response 的 sendRedirect 方法设置，404 用 sendError 指定。

这个应用程序中，servlet 构建一个 HTML 表单(参见图 6.3 和清单 6.5 中的源代码)，显示一个页面，让用户指定搜索字符串并选择使用的搜索引擎。用户提交表单以后，这个 servlet 从中提取出这两项参数，构造一个 URL，按照适合于所选搜索引擎的方式将参数嵌入到 URL 中(参见清单 6.3 和清单 6.4 中的 SearchSpec 和 SearchUtilities 类)，并将用户重定向该 URL(参见图 6.4)。如果用户没有选择搜索引擎或没有指定搜索项，错误页面会显示出来告知用户这种情况(参见图 6.5，但要注意前一节中给出的有关 Internet Explorer 对 404 状态代码处理的警告)。

清单 6.2 SearchEngines.java

```
package coreservlets;

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

/** Servlet that takes a search string and a search
 * engine name, sending the query to
 * that search engine. Illustrates manipulating
 * the response status code. It sends a 302 response
 * (via sendRedirect) if it gets a known search engine,
 * and sends a 404 response (via sendError) otherwise.
 */

public class SearchEngines extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String searchString = request.getParameter("searchString");
        if ((searchString == null) ||
            (searchString.length() == 0)) {
            reportProblem(response, "Missing search string");
            return;
        }
        // The URLEncoder changes spaces to "+" signs and other
        // non-alphanumeric characters to "%XY", where XY is the
        // hex value of the ASCII (or ISO Latin-1) character.
        // Browsers always URL-encode form values, and the
        // getParameter method decodes automatically. But since
```

```
// we're just passing this on to another server, we need to
// re-encode it to avoid characters that are illegal in
// URLs. Also note that JDK 1.4 introduced a two-argument
// version of URLEncoder.encode and deprecated the one-arg
// version. However, since version 2.3 of the servlet spec
// mandates only the Java 2 Platform (JDK 1.2 or later),
// we stick with the one-arg version for portability.
searchString = URLEncoder.encode(searchString);

String searchEngineName =
    request.getParameter("searchEngine");
if ((searchEngineName == null) ||
    (searchEngineName.length() == 0)) {
    reportProblem(response, "Missing search engine name");
    return;
}
String searchURL =
    SearchUtilities.makeURL(searchEngineName, searchString);
if (searchURL != null) {
    response.sendRedirect(searchURL);
} else {
    reportProblem(response, "Unrecognized search engine");
}
}

private void reportProblem(HttpServletRequest response,
                           String message)
throws IOException {
    response.sendError(response.SC_NOT_FOUND, message);
}
```

清单6.3 SearchSpec.java

```
package coreservlets;

/** Small class that encapsulates how to construct a
 * search string for a particular search engine.
 */

public class SearchSpec {
    private String name, baseURL;

    public SearchSpec(String name,
                      String baseURL) {
        this.name = name;
        this.baseURL = baseURL;
    }

    /** Builds a URL for the results page by simply concatenating
     * the base URL (http://...?someVar=") with the URL-encoded
     * search string (jsp+training).
     */

    public String makeURL(String searchString) {
        return(baseURL + searchString);
    }
}
```

```
public String getName() {
    return(name);
}
}

清单6.4 SearchUtilities.java

package coreservlets;

/** Utility with static method to build a URL for any
 *  of the most popular search engines.
 */

public class SearchUtilities {
    private static SearchSpec[] commonSpecs =
        { new SearchSpec("Google",
                         "http://www.google.com/search?q%"),
          new SearchSpec("AllTheWeb",
                         "http://www.alltheweb.com/search?q%"),
          new SearchSpec("Yahoo",
                         "http://search.yahoo.com/bin/search?p%"),
          new SearchSpec("AltaVista",
                         "http://www.altavista.com/web/results?q%"),
          new SearchSpec("Lycos",
                         "http://search.lycos.com/default.asp?query%"),
          new SearchSpec("HotBot",
                         "http://hotbot.com/default.asp?query%"),
          new SearchSpec("MSN",
                         "http://search.msn.com/results.asp?q%"),
        };
    public static SearchSpec[] getCommonSpecs() {
        return(commonSpecs);
    }
    /** Given a search engine name and a search string, builds
     *  a URL for the results page of that search engine
     *  for that query. Returns null if the search engine name
     *  is not one of the ones it knows about.
     */
    public static String makeURL(String searchEngineName,
                                 String searchString) {
        SearchSpec[] searchSpecs = getCommonSpecs();
        String searchURL = null;
        for(int i=0; i<searchSpecs.length; i++) {
            SearchSpec spec = searchSpecs[i];
            if (spec.getName().equalsIgnoreCase(searchEngineName)) {
                searchURL = spec.makeURL(searchString);
                break;
            }
        }
        return(searchURL);
    }
}
```

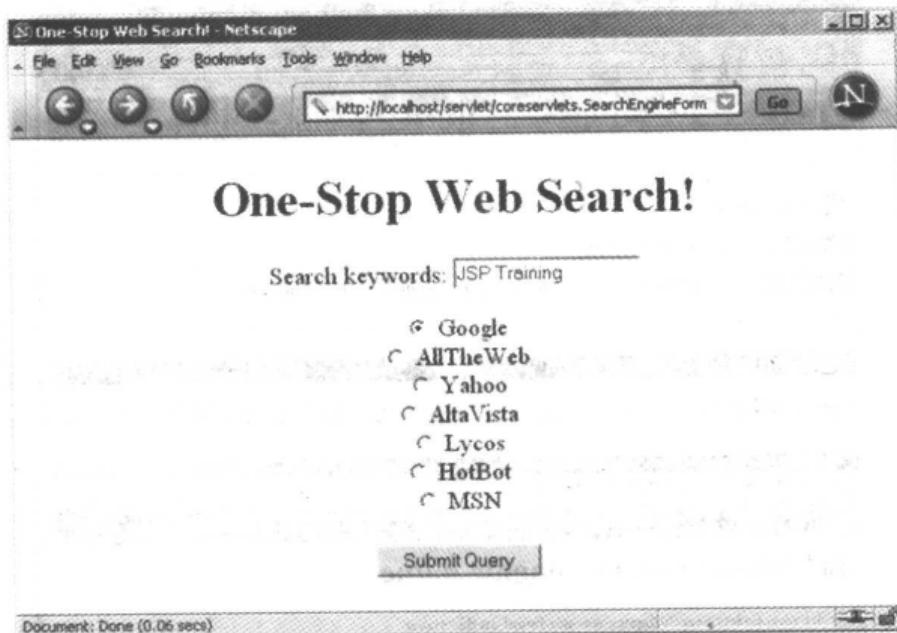


图 6.3 SearchEngines servlet 的前端。源代码参见清单 6.5

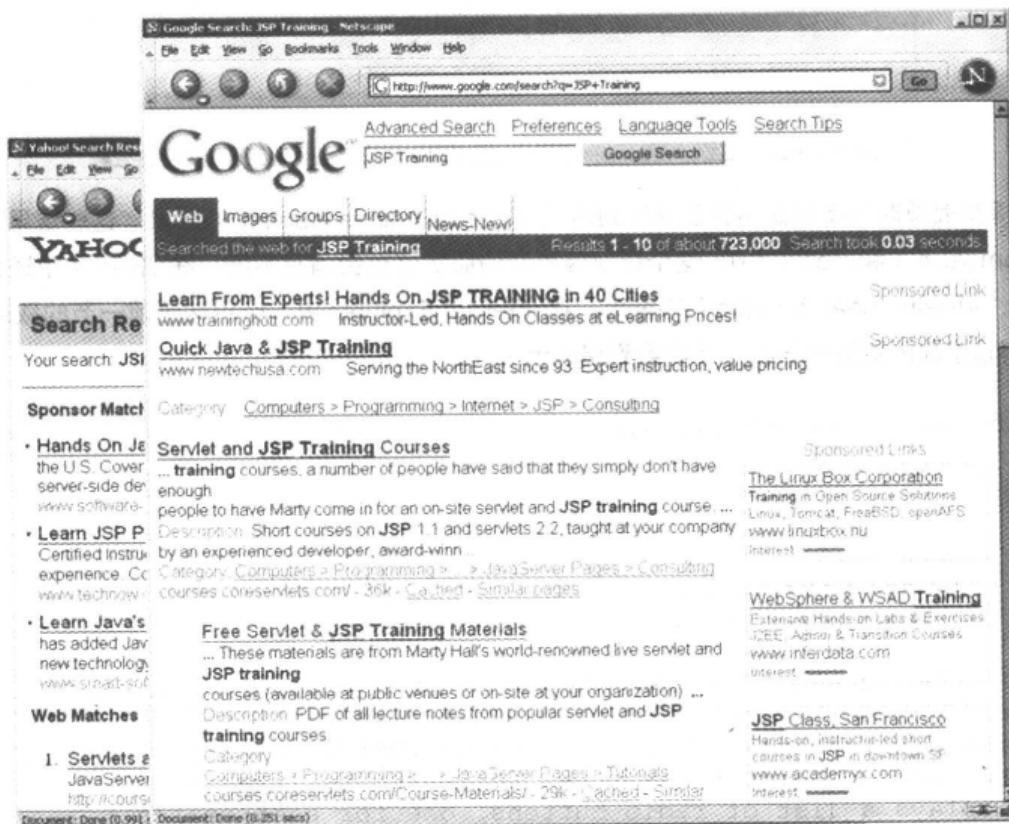


图 6.4 图 6.3 中的表单提交后 SearchEngines servlet 的结果。尽管表单提交给 SearchEngines servlet，但该 servlet 没有生成任何输出，最终用户看到的只是重定向的结果

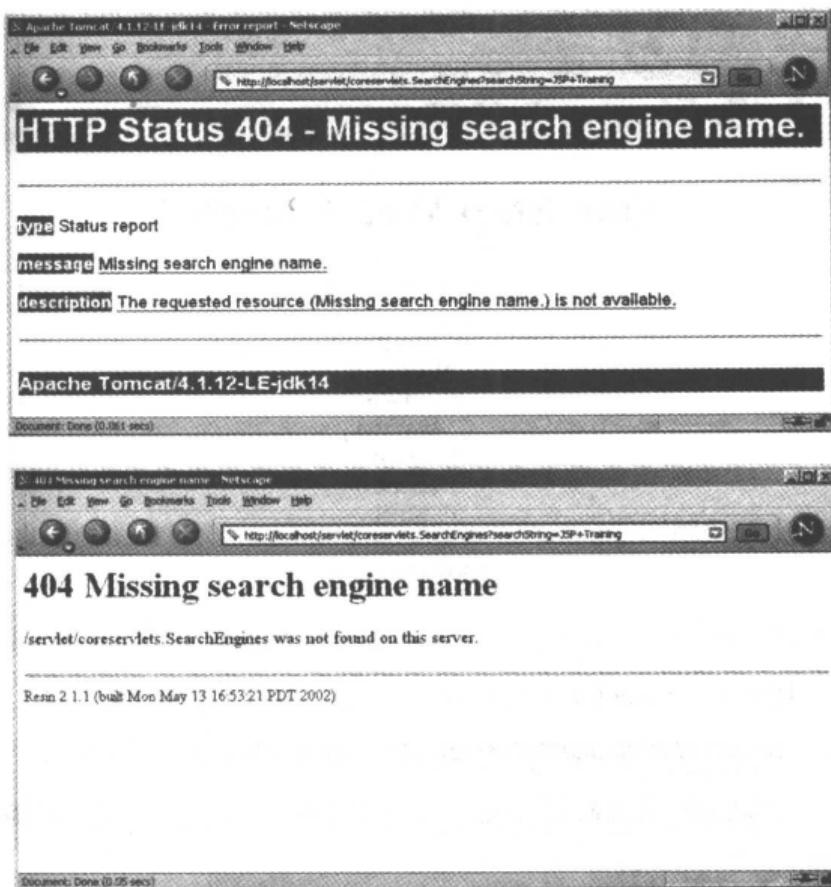


图 6.5 与提交表单时没有指定搜索引擎所对应的 SearchEngines servlet 的结果。这些结果分别来自于 Tomcat 4.1 和 Resin 4.0；这个结果因不同的服务器也会稍有不同，在 JRun 4 中还会错误地省略“Missing search string”字符串。在 Internet Explorer 中，必须按照前一节的叙述(参见 404 一项)修改浏览器的设置，才能看到错误消息

清单 6.5 SearchEngineForm.java

```
package coreservlets;

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

/** Servlet that builds the HTML form that gathers input
 * for the search engine servlet. This servlet first
 * displays a textfield for the search query, then looks up
 * the search engine names known to SearchUtilities and
 * displays a list of radio buttons, one for each search
 * engine.
 */

public class SearchEngineForm extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "One-Stop Web Search!";
        // ... (HTML code for the search form goes here)
    }
}
```

```
String actionURL = "/servlet/coreservlets.SearchEngines";
String docType =
    "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
     "Transitional//EN\">\n";
out.println
(docType +
"<HTML>\n" +
"<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
"<BODY BGCOLOR=\"#FDF5E6\">\n" +
"<CENTER>\n" +
"<H1>" + title + "</H1>\n" +
"<FORM ACTION=\"" + actionURL + "\">\n" +
" Search keywords: \n" +
" <INPUT TYPE=\"TEXT\" NAME=\"searchString\"><P>\n");
SearchSpec[] specs = SearchUtilities.getCommonSpecs();
for(int i=0; i<specs.length; i++) {
    String searchEngineName = specs[i].getName();
    out.println("<INPUT TYPE=\"RADIO\" " +
               "NAME=\"searchEngine\" " +
               "VALUE=\"" + searchEngineName + "\">\n");
    out.println(searchEngineName + "<BR>\n");
}
out.println
("<BR> <INPUT TYPE=\"SUBMIT\">\n" +
"</FORM>\n" +
"</CENTER></BODY></HTML>");
}
```

第 7 章 服务器响应的生成： HTTP 响应报头

本章的主题：

- HTTP 响应的格式
- HTTP 响应报头的设置
- 响应报头的用途
- 构建 Excel 电子表格
- 动态生成 JPEG 图像
- 向浏览器发送增量更新

前一章中已经提到过，Web 服务器的响应一般由状态行、一个或多个响应报头(其中之一必须为 Content-Type)、一个空行和关联的文档组成。要让 servlet 发挥最大的效力，您不仅需要知道如何生成文档，还需要知道如何有效地使用状态行和响应报头。

如前一章所述，HTTP 响应报头的设置，经常要和状态行中状态代码的设置结合起来。例如，所有“文档发生移动”的状态代码(300 到 307)都伴随一个 Location 报头，401(Unauthorized)代码经常包括伴随的 WWW-Authenticate 报头。然而，即使在设置正常的状态代码时，指定报头也可能起到十分重要的作用。响应报头可以用来：指定 cookie；提供页面的修改日期(用于客户端缓存)，指示浏览器在指定的时间间隔后重新载入页面；给出文件的大小使持续性 HTTP 连接的应用成为可能；指定生成文档的类型以及执行许多其他任务。本章展示如何生成响应报头，说明各种报头的用途，并给出几个具体的例子。

7.1 在 servlet 中设置响应报头

指定报头时，最通用的方式是使用 HttpServletResponse 的 setHeader 方法。这个方法接收两个字符串：报头的名称和报头的值。和设置状态代码一样，必须在返回实际的文档之前指定相关的报头。

- **setHeader(String headerName, String headerValue)**
这个方法将指定名称的响应报头设为给定的值。

除了通用的 setHeader 方法之外，HttpServletResponse 还有两个专门的方法，用来设置含有日期和整数的报头。

- **setDateHeader(String header, long milliseconds)**
这个方法省去将 Java 日期(自 1970 年以来的毫秒数，System.currentTimeMillis，Date.getTime 或 Calendar.getTimeInMillis 都返回这种类型的日期)转换成 GMD 时间字符串的麻烦。
- **setIntHeader(String header, int headerValue)**
这个方法可以省去在将整数插入到报头之前将 int 转换成 String 的不便。

HTTP 允许相同的报头名多次出现，有时，我们希望加入新的报头，而非替换已有的同名报头。例如，多个 `Accept` 和 `Set-Cookie` 报头分别指定所支持的不同 MIME 类型和不同 cookie，这种做法十分普遍。方法 `setHeader`, `setDateHeader` 和 `setIntHeader` 替换任何同名的已有报头，而 `addHeader`, `addDateHeader` 和 `addIntHeader` 等方法添加一个报头，不管是否已经存在同名的报头。如果确实需要知道是否已经设置了特定的报头，可以使用 `containsHeader` 进行检查。

最后，`HttpServletResponse` 还提供许多方便的方法来指定常用的报头。下面汇总了这些方法。

- `setContent-Type(String mimeType)`
这个方法设置 `Content-Type` 报头，大多数 servlet 都要用到这个方法。
- `setContentLength(int length)`
这个方法设置 `Content-Length` 报头，如果浏览器支持持续性(继续使用)HTTP 连接，这个报头十分有用。
- `addCookie(Cookie c)`
这个方法向 `Set-Cookie` 报头插入一个 cookie。由于响应中一般都会拥有多个 `Set-Cookie` 行，故而没有对应的 `setCookie` 方法。有关 cookie 的论述参见第 8 章。
- `sendRedirect(String address)`
如前一章所述，`sendRedirect` 方法将状态代码设为 302，同时设置 `Location` 报头。具体的例子参见 6.3 节和 6.4 节。

7.2 理解 HTTP 1.1 响应报头

下面对最常用的 HTTP 1.1 响应报头作了汇总。对这些报头的深刻理解能够增加您编写的 servlet 的效率，因此，您至少应该略读一下相关的叙述，看看可以使用哪些选项。在准备使用这些功能时，再回来阅读相关的细节。

这些报头是 HTTP 1.0 报头的超集。官方的 HTTP 1.1 规范在 RFC 2616 中给出。网络上的很多地方都提供这些 RFC；最好先到 <http://www.rfc-editor.org/> 获取档案站点的当前列表。报头名对大小写不敏感，但一般将每个单词的第一个字母大写。

如果 servlet 的行为依赖于只在 HTTP 1.1 中提供的响应报头，那么在编写时要极为小心，尤其是在您的 servlet 需要在 WWW 上运行，而非仅限于内联网的情况下，一些旧的浏览器只支持 HTTP 1.0。最好在使用 HTTP 1.1 专有的报头之前，用 `request.getRequestProtocol` 明确地检查 HTTP 的版本。

(1) Allow

`Allow` 报头指定服务器支持的请求方法(`GET`, `POST` 等)。`405(Method Not Allowed)` 响应需要用到这个报头。servlet 默认的 `service` 方法自动为 `OPTIONS` 请求生成这个报头。

(2) Cache-Control

这个报头告诉浏览器或其他客户，什么环境可以安全地缓存文档。它可能取下面这些值。

- **public**
文档可以缓存，即使按照正常的规则(例如密码保护的页面)，它不是可以缓存的文档。
- **private**
文档只适用于单个用户，只能存储在私有(非共享)缓存中。
- **no-cache**
文档不能被缓存(即不能用来满足后面的请求)。服务器还可以指定“`no-cache="header1, header2, ..., headerN"`”，来规定那些使用缓存的响应文档时应该略去的报头。浏览器一般不会缓存那些由含有表单数据的请求获取到的文档。但是，如果 servlet 为不同的请求生成不同的内容，即使在请求中没有表单数据的情况下也是如此，那么，告诉浏览器不要缓存响应就极为重要。由于老版本的浏览器使用 Pragma 报头完成这种任务，servlet 所采用的典型方法是设置所有的报头，如下面的例子所示。

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

- **no-store**
不要缓存文档，甚至不应该将它存储在磁盘上的临时目录中。这个报头的意图是防止由于疏忽而生成敏感数据的副本。
- **must-revalidate**
每次使用文档时，客户程序都必须联系原来的服务器(而非仅仅中间代理)，重新验证文档。
- **proxy-revalidate**
除了只适用于共享缓存以外，这个报头和 must-revalidate 相同。
- **max-age=xxx**
`xxx` 秒之后，应该将文档认作失效。这是 Expires 报头的方便替代，但只适用于 HTTP 1.1 客户程序。如果 max-age 和 Expires 在响应中同时存在，那么 max-age 优先。
- **s-max-age=xxx**
共享缓存应该在 `xxx` 秒之后将该文档认作失效。

Cache-Control 是 HTTP 1.1 新引入的报头。

(3) Connection

这个响应报头的 close 值，指示浏览器不要使用持续性 HTTP 连接。技术上，如果客户程序支持 HTTP 1.1 且没有指定 `Connection: close` 请求报头时(或当 HTTP 1.0 客户程序指定 `Connection: keep-alive` 时)，默认为持续性连接。但是，由于持续性连接需要 `Content-Length` 响应报头，所以 servlet 没有必要显式地使用 `Connection` 报头。如果不使用持续性连接，则可以省略 `Content-Length` 报头。

(4) Content-Disposition

使用 `Content-Disposition` 报头，可以要求浏览器询问用户，将响应存储在磁盘上给定名称的文件中。它的用法如下：

```
Content-Disposition: attachment; filename=some-file-name
```

当您向客户发送非 HTML 响应时(例如，7.3 节中的 Excel 电子表格或 7.5 节中的 JPEG 图像)，这个报头尤其有用。Content-Disposition 不是原始 HTTP 规范的一部分；它是后来在 RFC 2183 中定义的。再提一下，您可以到 <http://rfc-editor.org/> 按照给出的指示下载相关的 RFC 文档。

(5) **Content-Encoding**

这个报头标明页面在传输过程中所使用的编码方式。浏览器应该在决定如何处理文档之前先进行逆向编码。用 gzip 压缩文档能够极大地节省传输时间；具体的例子参见 5.4 节。

(6) **Content-Language**

Content-Language 报头表示文档所使用的语言。这个报头的值应该是标准的语言编码之一，比如 en, en-us, da 等。RFC1766 中含有语言编码的详细信息(可以在 <http://www.rfc-editor.org/> 列出的档案站点中在线访问 RFC 文档)。

(7) **Content-Length**

这个报头标明响应中的字节数。仅当浏览器使用持续性(继续使用)HTTP 连接时，才需要用到这个信息。有关如何确定浏览器何时支持持续性连接，参见 Connection 报头。如果希望在浏览器支持持续性连接时，您的 servlet 能够利用持续性连接的优点，那么，您的 servlet 应该将文档写入 ByteArrayOutputStream 中，完成后检查它的大小，将大小用 response.setContentType 填入 Content-Length 字段，之后用 byteArrayStream.writeTo(response.getOutputStream()) 发送这些内容。

(8) **Content-Type**

Content-Type 报头给出响应文档的 MIME(Multipurpose Internet Mail Extension，多用因特网邮件扩展)类型。设置这个报头是如此普遍，以至于在 HttpServletResponse 中为它提供了一个专门的方法：setContentType。正式注册的 MIME 类型的形式为 *mainType/subType*；未注册的 MIME 类型的形式为 *mainType/x-subType*。大多数 servlet 指定 text/html；但是，它们也可以指定其他类型。这个报头之所以重要，部分原因是由于 servlet 能够直接生成其他 MIME 类型(如本章的 Excel 和 JPEG 示例所示)，还有一部分原因是，servlet 还用作将其他应用程序连接到 Web 的“粘合代码”。假定您用 Adobe Acrobat 来生成 PDF 文件，用 GhostScript 生成 PostScript，用数据库应用程序来搜索建立索引后的 MP3 文件。您依旧需要 servlet 来应答 HTTP 请求，调用辅助应用程序，并设置 Content-Type 报头，即使 servlet 可能只是简单地将辅助应用程序的输入直接传递给客户程序。

除基本的 MIME 类型之外，Content-Type 报头还可以指定具体的字符编码。如果没有指定字符编码，默认为 ISO-8859_1(Latin)。例如，下面这一行语句指示浏览器将文档解释作 Shift_JIS(标准日语)字符集的 HTML。

```
response.setContentType("text/html; charset=Shift_JIS");
```

表 7.1 列出 servlet 最常使用的一些 MIME 类型。RFC 1521 和 RFC 1522 列出了更多常见的 MIME 类型(同样，参见 <http://www.rfc-editor.org/> 给出的 RFC 档案站点的

列表)。但是,新的 MIME 类型不断注册,所以最好查看动态列表。正式注册的类型列在 <http://www.isi.edu/in-notes/iana/assignments/media-types/media-types>。对于常见的未注册类型, <http://www.itsw.se/knbase/internet/mime.htm> 是一个不错的地方。

表 7.1 常见 MIME 类型

类 型	含 义
application/msword	Microsoft Word 文档
application/octet-stream	未识别或二进制数据
application/pdf	Acrobat(.pdf)文件
application/postscript	PostScript 文件
application/vnd.lotus-notes	Lotus Notes 文件
application/vnd.ms-excel	Excel 电子表格
application/vnd.ms-powerpoint	PowerPoint 演示文稿
application/x-gzip	Gzip 档案
application/x-java-archive	JAR 文件
application/x-java-serialized-object	序列化的 Java 对象
application/x-java-vm	Java 字节代码(.class)文件
application/zip	Zip 档案
audio/basic	.au 或.snd 格式的音频文件
audio/midi	MIDI 音频文件
audio/x-aiff	AIFF 音频文件
audio/x-wav	Microsoft Windows 音频文件
image/gif	GIF 图像
image/jpeg	JPEG 图像
image/png	PNG 图像
image/tiff	TIFF 图像
image/x-xbitmap	X Windows 位图图像
text/css	HTML 层叠样式表
text/html	HTML 文档
text/plain	纯文本
text/xml	XML
video/mpeg	MPEG 视频片断
video/quicktime	QuickTime 视频片断

(9) Expires

这个报头规定内容的过期时间,从而不再需要继续缓存。`servlet` 可以对那些更改相对频繁的文档使用这个报头,阻止浏览器显示过时的缓存值。此外,由于一些老版本的浏览器对 Pragma 的支持不太可靠(且根本不支持 Cache-Control),在过去,Expires 报头常常和一个日期一起,阻止浏览器执行缓存。但是,一些浏览器忽略 1980 年 1 月 1 日之前的日期,所以不要将 0 用作 Expires 报头的值。

例如，下面的语句指示浏览器不要缓存文档超过 10 分钟。

```
long currentTime = System.currentTimeMillis();
long tenMinutes = 10*60*1000; // In milliseconds
response.setDateHeader("Expires", currentTime + tenMinutes);
```

当然，同时还要检查 Cache-Control 报头的 max-age 值。

(10) Last-Modified

这是个十分有用的报头，它标明文件的最后修改时间。之后，客户程序就可以缓存该文档，并在后面的请求中用 If-Modified-Since 请求报头提供一个日期。这个请求可以看作是条件性 GET，只在 Last-Modified 日期比 If-Modified-Since 指定的日期要晚时才会返回文档。否则，服务器会返回 304(Not Modified)状态行，客户程序则使用缓存中的文档。如果您要显式地设置这个报头，可以使用 setDateHeader 方法省去自己格式化 GMT 日期字符串的麻烦。然而，大多数情况下，您只需要实现 getLastModified 方法(参见 3.6 节中的彩票数字 servlet)，由标准的 service 方法处理 If-Modified-Since 请求。

(11) Location

状态代码在 300 至 399 之间的所有响应都应该包括这个报头，它通知浏览器文档的地址。浏览器自动重新连接到这个地址，读取新的文档。这个报头，连同 302 状态代码，一般用 HttpServletResponse 的 sendRedirect 方法间接设置。具体的例子请参见 6.3 节和 6.4 节。

(12) Pragma

提供这个报头且将值设为 no-cache，指示 HTTP 1.0 客户不要缓存文档。但 HTTP 1.0 浏览器对这个报头的支持并不一致，因此常常通过将 Expires 设为已经过去的日期来完成这项任务。在 HTTP 1.1 中，Cache-Control: no-cache 是更为可靠的替代方法。

(13) Refresh

这个报头标明浏览器应该多长时间(以秒为单位)之后请求最新的页面。例如，要告诉浏览器 30 秒钟后请求新的副本，可以将它的值指定为 30。

```
response.setIntHeader("Refresh", 30);
```

要注意，Refresh 并不规定持续性更新；它只是指定下一次更新会在什么时候。因而，您必须在所有的后续响应中不断地提供 Refresh。这个报头极为有用，它可以使 servlet 快速地返回局部结果，同时，依旧使得客户能够在晚一些时候看到完整的结果。具体的例子参见 7.4 节。

除了让浏览器重新载入当前页面以外，我们还可以指定载入的页面。通过在刷新时间之后添加分号和一个 URL，可以完成这个任务。例如，要告诉浏览器在 5 秒钟后跳转到 *http://host/path*，只需使用下面的语句：

```
response.setHeader("Refresh", "5; URL=http://host/path/");
```

这项设置可以用来显示“splash screens(闪屏)”，在载入实际页面之前先简短地显示一幅引导性的图像或消息。

请注意，一般将下面的内容放入到 HTML 页面的 HEAD 节中，间接地设置这个报头，而不是作为显式的来自服务器的报头。

```
<META HTTP-EQUIV="Refresh"  
      CONTENT="5; URL=http://host/path/">
```

之所以采用这种用法是因为自动重载或转发常常是编写静态 HTML 页面的人员希望使用的方法。然而，servlet 直接设置这个报头更容易，也更清楚。

这个报头不是 HTTP 1.1 的正式部分，但却是 Netscape 和 Internet Explorer 都支持的扩展。

(14) Retry-After

这个报头可以和 503(Service Unavailable)响应结合使用，告诉客户程序多久之后可以重复它的请求。

(15) Set-Cookie

Set-Cookie 报头指定一个与页面相关联的 cookie。每个 cookie 都要求一个单独的 Set-Cookie 报头。servlet 不应该使用 response.setHeader("Set-Cookie", ...)，而应该使用 HttpServletResponse 中专用的 addCookie 方法。详细信息请参见第 8 章。技术上，Set-Cookie 不属于 HTTP 1.1。最初它是一项 Netscape 扩展，但现在已经被各种浏览器广泛支持，包括 Netscape 和 Internet Explorer。

(16) WWW-Authenticate

这个报头总是和 401(Unauthorized)状态代码一同使用。它告诉浏览器，客户应该在 Authorization 报头中提供哪种验证类型(BASIC 或 DIGEST)和域。有关 WWW-Authenticate 的使用，以及 servlet 和 JSP 页面可以使用的各种安全机制的讨论，参见本书第二卷有关 Web 应用安全的章节。

7.3 构建 Excel 电子表格

虽然 servlet 一般生成 HTML 输出，但它们不是必须这样做。HTTP 是 servlet 的基础，但 HTML 不是。现在，有时生成 Microsoft Excel 内容可能会比较有用，这样用户就可以将结果存入报告中，并且还可以使用 Excel 中内建的公式支持。Excel 至少接受 3 种不同格式的输出：用制表符分隔的数据、HTML 表格和本地二进制格式。

本节中，我们阐述如何使用制表符分隔的数据生成电子表格。第 12 章中，我们将介绍如何使用 HTML 表格格式构建 Excel 表格。格式不是问题，关键是使用 Content-Type 响应报头告知客户程序您正在发送电子表格。设置 Content-Type 报头使用快捷的 setContentType 方法，Excel 表格的 MIME 类型是 application/vnd.ms-excel。因而，要生成 Excel 电子表格，只需：

```
response.setContentType("application/vnd.ms-excel");  
PrintWriter out = response.getWriter();
```

之后，在这两行语句之间输出含有制表符(Java 字符串中为\t)的一些数据项。也就是说：没有 DOCTYPE、HEAD、BODY 这些 HTML 特有的内容。

清单 7.1 给出一个简单的 servlet，它构建一份比较苹果和橙子的 Excel 电子表格。要注

意， $=\text{SUM}(col:col)$ 合计 Excel 中特定范围的列。图 7.1 给出对应的结果。

清单7.1 ApplesAndOranges.java

```
package coreservlets;

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

/** Servlet that creates Excel spreadsheet comparing
 * apples and oranges.
 */

public class ApplesAndOranges extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("application/vnd.ms-excel");
        PrintWriter out = response.getWriter();
        out.println("\tQ1\tQ2\tQ3\tQ4\tTotal");
        out.println("Apples\t78\t87\t92\t29\t286");
        out.println("Oranges\t77\t86\t93\t30\t286");
    }
}
```

The screenshot shows a Microsoft Internet Explorer window displaying an Excel spreadsheet. The address bar shows the URL: http://localhost/servlet/coreservlets.ApplesAndOranges. The spreadsheet has columns labeled Q1, Q2, Q3, Q4, and Total. Row 1 contains the column headers. Rows 2 and 3 contain data for Apples and Oranges respectively. The data is as follows:

	A	B	C	D	E	F	G	H
1		Q1	Q2	Q3	Q4	Total		
2	Apples	78	87	92	29	286		
3	Oranges	77	86	93	30	286		
4								
5								
6								

图 7.1 安装有 Microsoft Office 的系统上，ApplesAndOranges servlet 在 Internet Explorer 中的结果

7.4 servlet 状态的持续以及页面的自动重载

假定您的 servlet 或 JSP 页面执行一项耗时较长的计算任务：比如说 20 秒或者更多。在这种情况下，完成计算后再将结果发送给用户是不合理的，到了那时客户可能已经放弃，并离开该页面，或者更坏，用户可能单击 Reload(重新载入)按钮重新开始了计算过程。为了处理耗时较长的请求，您需要下面这些功能：

- 一种跨请求存储数据的方式。

对于不专属于任意客户的数据，可以将其存储在 servlet 的字段(实例变量)中。对于用户专属的数据，可以将其存储在 HttpSession 对象中(参见第 9 章)。对于其他 servlet 和 JSP 页面需要用到的数据，可以将它存储ServletContext 中(参见第 14 章中介绍共享数据的小节)。

- 一种在请求发送给用户之后，保持运算继续进行的方式。

这个任务比较简单：只需启动一个线程(Thread)。系统为了应答请求而启动的线程，在响应完成之后自动结束，但其他的线程会保持运行。唯一的细微差别是：要将线程的优先级别设为一个较低的值，这样才不至于影响整个服务器的运行。

- 一种在更新的结果就绪后，使浏览器得到它的方式。

遗憾的是，由于浏览器并不维护一个与服务器之间的保持打开的连接，所以，服务器要想将新的结果主动地发送给浏览器不容易。取而代之，应该指示浏览器请求更新。这就是 Refresh 响应报头的用途。

查找公钥密码学使用的质数

下面给出一个具体例子，它可以提供由一些较大的、随机选定的质数构成的列表。您可能早已知道，大质数是大多数公钥密码系统，以及用在 Web 上的加密系统(例如在 SSL 和 X509 证书中的应用)的关键。对于很大的数字(例如 100 位)，寻找质数可能要花费一些时间，因此，这个 servlet 立即返回初始的结果，但其后会继续计算，并使用低优先级的线程，从而不会降低 Web 服务器的性能。如果计算尚未完成，服务器就会通过向浏览器发送 Refresh 报头，指示浏览器几秒钟后请求新的页面。

除了阐明 HTTP 响应报头(在这种情况下为 Refresh)的价值之外，这个例子展示出另外两种有价值的服务功能。首先，它展示出同一 servlet 可以处理多个同时发生的连接，每个连接都有自己的线程。因而，当一个线程正在忙于完成客户的计算时，其他客户能够连接上来，且依旧可以看到局部的结果。

其次，这个例子说明对于 servlet 来说，跨请求维护状态是多么容易，而在大多数竞争技术中，这项功能都比较难以实现(甚至在.NET 中——也许是除 servlet 以外最好的选择，也是如此)。由于只创建 servlet 的单个实例，每个请求只是产生一个新线程，调用 servlet 的 service 方法(该方法调用 doGet 或 doPost)。因此，共享的数据只需简单地放在 servlet 的常规实例变量(字段)中。因此，servlet 可以在浏览器重新载入页面的同时，访问正在进行的运算，并且能够保持最近 N 个请求结果的列表，如果新的请求指定了与最近请求相同的参数，则立即返回它们。当然，由设计者负责同步对共享数据的多线程访问这一普遍准则，依旧适用于 servlet。servlet 还可以将持续性数据存储在 ServletContext 对象中(通过 getServletContext 得到该对象)。ServletContext 提供 setAttribute 和 getAttribute 方法，使用这些方法您可以存储与指定键相关联的任意数据。将数据存入到实例变量，与存入到 ServletContext 的不同是：ServletContext 由 Web 应用中所有的 servlet 和 JSP 页面共享。

清单 7.2 给出主 servlet 类。首先，它接收一个请求，该请求指定了两个参数：numPrimes 和 numDigits。从用户那里收集这些值并发送到 servlet 这项工作一般通过简单的 HTML 表单来完成。清单 7.3 给出 HTML 表单的源代码，图 7.2 是相应的结果。接下来，通过简单的实用程序(应用 Integer.parseInt)将这些参数转换成整型数(参见清单 7.6)。随后，findPrimeList 方法将这些值与 ArrayList(存储最近完成的或正在进行的计算)进行匹配，检查是否存在一项之前的计算对应相同的两个值。如果找到，则使用之前计算出的值(PrimeList 类型)；否则，则创建新的 PrimeList，并在其中存入正在计算的 Vector，有可能会取代之前结果中最老的列表。接下来，检查该 PrimeList，确定查找所有质数的任务是否结束。如果尚未完成，则向客户程序发送 Refresh 报头，告诉客户程序 5 秒钟后再来请求最新的结果。

不管哪种方式，servlet 都会返回给客户程序由当前的值构成的项目列表。代表性的结果请参见图 7.3 到图 7.5。

清单7.2 PrimeNumberServlet.java

```
package coreservlets;

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

/** Servlet that processes a request to generate n
 * prime numbers, each with at least m digits.
 * It performs the calculations in a low-priority background
 * thread, returning only the results it has found so far.
 * If these results are not complete, it sends a Refresh
 * header instructing the browser to ask for new results a
 * little while later. It also maintains a list of a
 * small number of previously calculated prime lists
 * to return immediately to anyone who supplies the
 * same n and m as a recently completed computation.
 */

public class PrimeNumberServlet extends HttpServlet {
    private ArrayList primeListCollection = new ArrayList();
    private int maxPrimeLists = 30;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        int numPrimes =
            ServletUtilities.getIntParameter(request,
                                              "numPrimes", 50);
        int numDigits =
            ServletUtilities.getIntParameter(request,
                                              "numDigits", 120);
        PrimeList primeList =
            findPrimeList(primeListCollection, numPrimes, numDigits);
        if (primeList == null) {
            primeList = new PrimeList(numPrimes, numDigits, true);
            // Multiple servlet request threads share the instance
            // variables (fields) of PrimeNumbers. So
            // synchronize all access to servlet fields.
            synchronized(primeListCollection) {
                if (primeListCollection.size() >= maxPrimeLists)
                    primeListCollection.remove(0);
                primeListCollection.add(primeList);
            }
        }
        ArrayList currentPrimes = primeList.getPrimes();
        int numCurrentPrimes = currentPrimes.size();
        int numPrimesRemaining = (numPrimes - numCurrentPrimes);
        boolean isLastResult = (numPrimesRemaining == 0);
        if (!isLastResult) {
            response.setIntHeader("Refresh", 5);
        }
        response.setContentType("text/html");
```

```

PrintWriter out = response.getWriter();
String title = "Some " + numDigits + "-Digit Prime Numbers";
out.println(ServletUtilities.headWithTitle(title) +
    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
    "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
    "<H3>Primes found with " + numDigits +
    " or more digits: " + numCurrentPrimes +
    ".</H3>");
if (isLastResult)
    out.println("<B>Done searching.</B>");  

else
    out.println("<B>Still looking for " + numPrimesRemaining +
        " more<BLINK>...</BLINK></B>");  

out.println("<OL>");  

for(int i=0; i<numCurrentPrimes; i++) {
    out.println(" <LI>" + currentPrimes.get(i));
}
out.println("</OL>");  

out.println("</BODY></HTML>");  

}  

// See if there is an existing ongoing or completed  

// calculation with the same number of primes and number  

// of digits per prime. If so, return those results instead  

// of starting a new background thread. Keep this list  

// small so that the Web server doesn't use too much memory.  

// Synchronize access to the list since there may be  

// multiple simultaneous requests.  

private PrimeList findPrimeList(ArrayList primeListCollection,
                                int numPrimes,
                                int numDigits) {
    synchronized(primeListCollection) {
        for(int i=0; i<primeListCollection.size(); i++) {
            PrimeList primes =
                (PrimeList)primeListCollection.get(i);
            if ((numPrimes == primes.numPrimes()) &&
                (numDigits == primes.numDigits()))
                return(primes);
        }
        return(null);
    }
}
}
}

```

清单7.3 PrimeNumbers.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  

<HTML>  

<HEAD>  

    <TITLE>Finding Large Prime Numbers</TITLE>  

</HEAD>  

<BODY BGCOLOR="#FDF5E6">  

<CENTER>  

<H2>Finding Large Prime Numbers</H2>  

<BR><BR>  

<FORM ACTION="/servlet/core servlets.PrimeNumberServlet">  

    <B>Number of primes to calculate:</B>

```

```
<INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
<B>Number of digits:</B>
<INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
<INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>
</BODY></HTML>
```

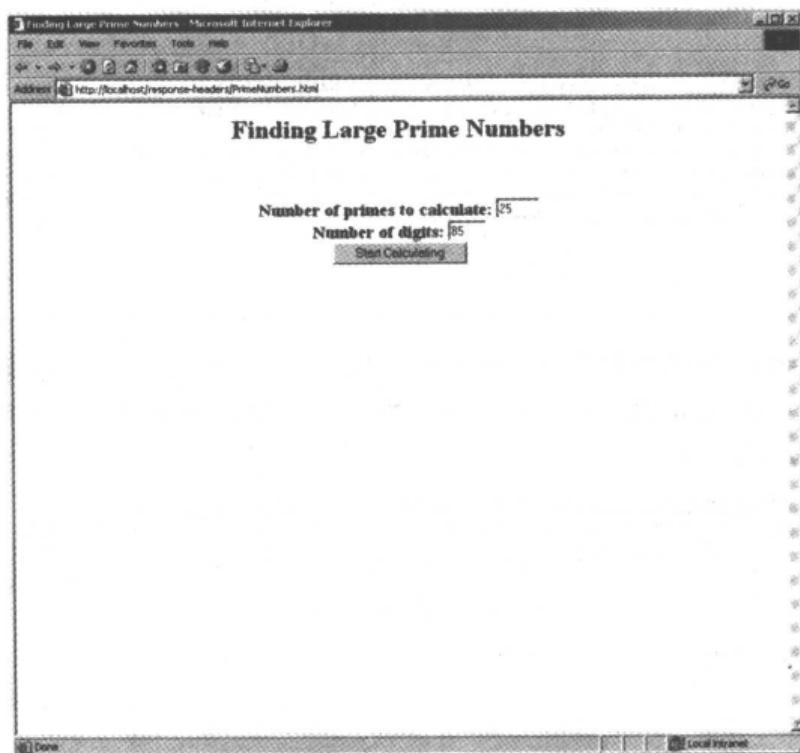


图 7.2 生成质数的 servlet 的前端

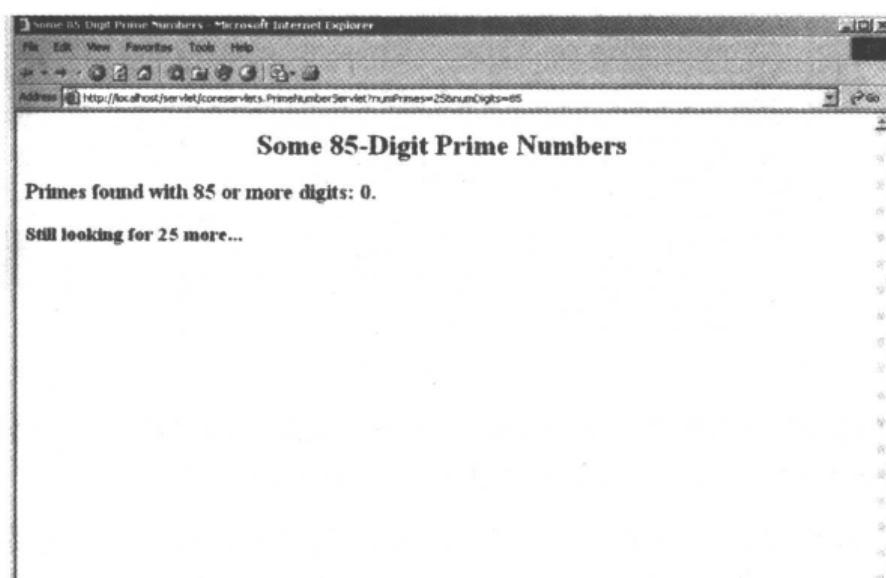


图 7.3 生成质数的 servlet 的初始结果。首先发送给浏览器一个快速结果，同时指示(在 Refresh 报头中)浏览器在 5 秒钟后重新连接获取更新

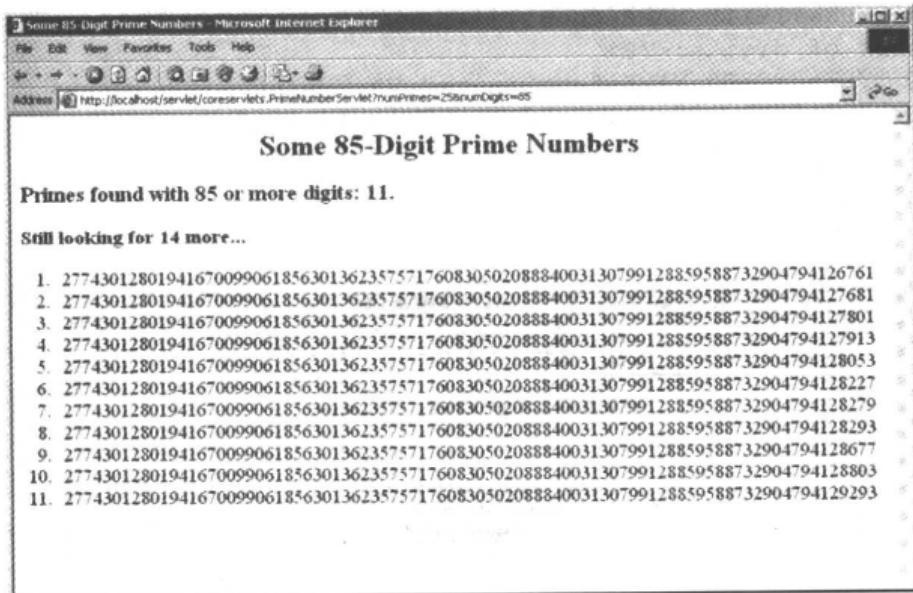


图 7.4 生成质数的 servlet 的中间结果。该 servlet 存储之前的计算结果，并通过比较请求参数(需要计算的质数的大小和数目)，将当前的请求与存储的值匹配。请求相同参数的其他客户程序看到的是同样的已经计算好的结果

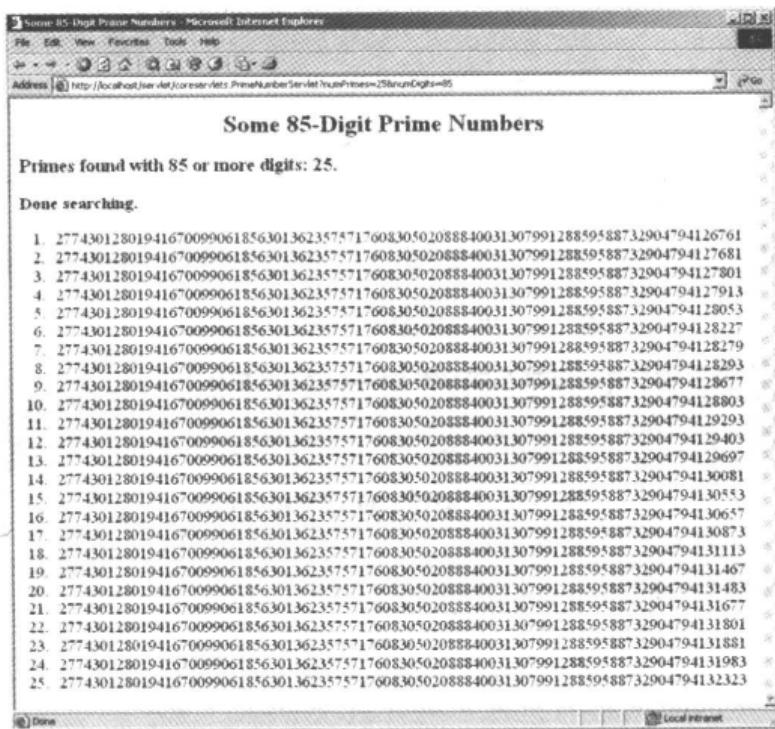


图 7.5 生成质数的 servlet 的最终结果。由于该 servlet 已经按照客户请求的数目计算出相应个数的质数，所以没有 Refresh 报头发送给浏览器，该页面也不再自动重新载入

清单 7.4(PrimeList.java) 和 清单 7.5(Primes.java) 给出该 servlet 使用的辅助代码。PrimeList.java 处理后台线程，为指定的一组值创建质数列表。这个例子的重点有两方面：这个 servlet 将数据存储在实例变量(或 ServletContext)中，跨请求维护数据；同时，这个 servlet 能够使用 Refresh 报头指示浏览器回来请求更新。但是，如果您关注质数生成的详细细节，Primes.java 还包含选择指定长度的随机数，然后在这个值及大于它的数值中寻找质数的底

层算法。它使用 BigInteger 类中的内建方法；确定数字是否为质数的算法是一个概率性的算法，因而也就有可能会出现错误。然而，发生错误的概率是可以指定的，我们可以使用错误值 100。假定大多数 Java 实现中使用的算法是 Miller-Rabin 测试，则可以证明，将一个合数(即非质数)错报成质数的可能性要小于 2^{100} 。这几乎肯定要小于确定性的算法中由于硬件错误或随机辐射引起不正确响应的可能性，因此可以将这种算法认为是确定性的。

清单7.4 PrimeList.java

```
package coreservlets;

import java.util.*;
import java.math.BigInteger;

/** Creates an ArrayList of large prime numbers, usually in
 * a low-priority background thread. Provides a few small
 * thread-safe access methods.
 */
public class PrimeList implements Runnable {
    private ArrayList primesFound;
    private int numPrimes, numDigits;

    /** Finds numPrimes prime numbers, each of which is
     * numDigits long or longer. You can set it to return
     * only when done, or have it return immediately,
     * and you can later poll it to see how far it
     * has gotten.
     */
    public PrimeList(int numPrimes, int numDigits,
                     boolean runInBackground) {
        primesFound = new ArrayList(numPrimes);
        this.numPrimes = numPrimes;
        this.numDigits = numDigits;
        if (runInBackground) {
            Thread t = new Thread(this);
            // Use low priority so you don't slow down server.
            t.setPriority(Thread.MIN_PRIORITY);
            t.start();
        } else {
            run();
        }
    }

    public void run() {
        BigInteger start = Primes.random(numDigits);
        for(int i=0; i<numPrimes; i++) {
            start = Primes.nextPrime(start);
            synchronized(this) {
                primesFound.add(start);
            }
        }
    }

    public synchronized boolean isDone() {
        return(primesFound.size() == numPrimes);
    }
}
```

```
}

public synchronized ArrayList getPrimes() {
    if (isDone())
        return(primesFound);
    else
        return((ArrayList)primesFound.clone());
}

public int numDigits() {
    return(numDigits);
}

public int numPrimes() {
    return(numPrimes);
}

public synchronized int numCalculatedPrimes() {
    return(primesFound.size());
}
}
```

清单 7.5 Primes.java

```
package coreservlets;

import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */

public class Primes {
    // Note that BigInteger.ZERO and BigInteger.ONE are
    // unavailable in JDK 1.1.
    private static final BigInteger ZERO = BigInteger.ZERO;
    private static final BigInteger ONE = BigInteger.ONE;
    private static final BigInteger TWO = new BigInteger("2");

    // Likelihood of false prime is less than 1/2^ERR_VAL.
    // Presumably BigInteger uses the Miller-Rabin test or
    // equivalent, and thus is NOT fooled by Carmichael numbers.
    // See section 33.8 of Cormen et al.'s Introduction to
    // Algorithms for details.
    private static final int ERR_VAL = 100;

    public static BigInteger nextPrime(BigInteger start) {
        if (isEven(start))
            start = start.add(ONE);
        else
            start = start.add(TWO);
        if (start.isProbablePrime(ERR_VAL))
            return(start);
        else
            return(nextPrime(start));
    }

    private static boolean isEven(BigInteger n) {
```

```
    return(n.mod(TWO).equals(ZERO));
}

private static StringBuffer[] digits =
{ new StringBuffer("0"), new StringBuffer("1"),
  new StringBuffer("2"), new StringBuffer("3"),
  new StringBuffer("4"), new StringBuffer("5"),
  new StringBuffer("6"), new StringBuffer("7"),
  new StringBuffer("8"), new StringBuffer("9") };

private static StringBuffer randomDigit(boolean isZeroOK) {
    int index;
    if (isZeroOK) {
        index = (int) Math.floor(Math.random() * 10);
    } else {
        index = 1 + (int) Math.floor(Math.random() * 9);
    }
    return(digits[index]);
}

/** Create a random big integer where every digit is
 * selected randomly (except that the first digit
 * cannot be a zero).
 */

public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
        if (i == 0) {
            // First digit must be non-zero.
            s.append(randomDigit(false));
        } else {
            s.append(randomDigit(true));
        }
    }
    return(new BigInteger(s.toString()));
}

/** Simple command-line program to test. Enter number
 * of digits, and the program picks a random number of that
 * length and then prints the first 50 prime numbers
 * above that.
 */

public static void main(String[] args) {
    int numDigits;
    try {
        numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
        numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
        start = nextPrime(start);
        System.out.println("Prime " + i + " = " + start);
    }
}
```

清单 7.6 ServletUtilities.java (节选)

```

package coreservlets;

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

/** Some simple timesavers. Note that most are static methods.
 */

public class ServletUtilities {
    // ...
    /** Read a parameter with the specified name, convert it
     * to an int, and return it. Return the designated default
     * value if the parameter doesn't exist or if it is an
     * illegal integer format.
    */

    public static int getIntParameter(HttpServletRequest request,
                                      String paramName,
                                      int defaultValue) {
        String paramString = request.getParameter(paramName);
        int paramValue;
        try {
            paramValue = Integer.parseInt(paramString);
        } catch(NumberFormatException nfe) { // null or bad format
            paramValue = defaultValue;
        }
        return(paramValue);
    }
}

```

7.5 使用 servlet 生成 JPEG 图像

虽然 servlet 通常生成 HTML 输出，但是，毫无疑问，它们还可以生成其他内容。例如，7.3 节中就给出一个构建 Excel 表格并将它们返回给客户程序的 servlet。在此，我们展示如何生成 JPEG 图像。

首先，我们要汇总一下在构建多媒体内容时，servlet 必须执行的两个主要步骤。

(1) 通知浏览器它们正在发送的内容类型。

servlet 通过使用 HttpServletResponse 的 setContentType 方法设置 Content-Type 响应报头，来完成这项任务。

(2) 以恰当的格式发送输出。

当然，不同的文档类型，这个格式也会有所不同，但大多数情况下发送的是二进制数据，而非发送 HTML 文档时所用的字符串。因此，servlet 一般使用 getOutputStream 方法获取原始的输出流，而非使用 getWriter 获取 PrintWriter。

将这两步放在一起，生成非 HTML 内容的 servlet 一般在 doGet 或 doPost 方法中含有如下的两段语句：

```

response.setContentType("type/subtype");
OutputStream out = response.getOutputStream();

```

这是构建非 HTML 内容时必需的两个常规步骤。下面，我们检查一下生成 JPEG 图像所需的两个具体步骤。

(1) 创建一个 `BufferedImage`。

通过调用 `BufferedImage` 的构造函数，给出宽度、高度、以及由 `BufferedImage` 类中定义的常量所定义的图像表示类型，创建一个 `java.awt.image.BufferedImage` 对象。表示类型并不重要，因为我们并不直接操纵 `BufferedImage` 中的二进制位，而且在转换到 JPEG 时大多数类型都产生相同的结果。我们使用 `TYPE_INT_RGB`。将这些工作放在一起，下面是一般的过程：

```
int width = ...;
int height = ...;
BufferedImage image =
    new BufferedImage(width, height,
                      BufferedImage.TYPE_INT_RGB);
```

(2) 在 `BufferedImage` 上绘制内容。

通过调用图像的 `getGraphics` 方法，将得到的 `Graphics` 对象转换成 `Graphics2D`，然后使用 Java 2D 丰富的绘画操作、坐标变换、字体设置和填充调色板执行具体的绘画。下面是一个简单的例子。

```
Graphics2D g2d = (Graphics2D)image.getGraphics();
g2d.setXXX(...);
g2d.fill(someShape);
g2d.draw(someShape);
```

(3) 设置 `Content-Type` 响应报头。

前面已经讨论过，完成这项任务使用 `HttpServletResponse` 的 `setContentType` 方法。JPEG 图像的 MIME 类型是 `image/jpeg`。因此，相应的代码如下：

```
response.setContentType("image/jpeg");
```

(4) 获取输出流。

前面论述过，发送二进制数据时，应该调用 `HttpServletResponse` 的 `getOutputStream` 方法，而非 `getWriter` 方法。例如：

```
OutputStream out = response.getOutputStream();
```

(5) 以 JPEG 格式将 `BufferedImage` 发送到输出流。

在 JDK 1.4 之前，自己完成这项任务需要做较多的工作。因此，大多数人会选择使用第三方的实用程序。然而，在 JDK 1.4 和以后的版本中，`ImageIO` 类极大地简化了这项任务。如果您使用的应用服务器支持 J2EE 1.4(包括 servlet 2.4 和 JSP 2.0)，您肯定拥有 JDK 1.4 或之后的版本。但是，独立的服务器并非一定要使用 JDK 1.4。因而，要清楚这种代码依赖于 Java 的版本。在使用 `ImageIO` 类时，只需传递给 `ImageIO` 的 `writer` 方法一个 `BufferedImage`、一个图像格式类型(`"jpg"`, `"png"`等——调用 `ImageIO.getWriterFormatNames` 可以得到完整的列表)、一个 `OutputStream` 或 `File`。除了要捕获必需的 `IOException` 之外，这就是整个过程。例如：

```
try {
```

```

        ImageIO.write(image, "jpg", out);
    } catch(IOException ioe) {
        System.err.println("Error writing JPEG file: " + ioe);
    }
}

```

清单 7.7 给出的 servlet 读取 message, fontName 和 fontSize 参数, 将它们传递给 MessageImage 实用工具类(清单 7.8)创建一个 JPEG 图像, 以指定的字体和大小显示该消息, 同时在主字符串后面以灰色、倾斜阴影显示该消息。如果用户点击了 Show Font List 按钮, 那么 servlet 就不再构建图像, 而是显示服务器支持的字体名称列表。

清单 7.7 ShadowedText.java

```

package coreservlets;

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

/** Servlet that generates JPEG images representing
 * a designated message with an oblique shadowed
 * version behind it.
 */

public class ShadowedText extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String wantsList = request.getParameter("showList");
        if (wantsList != null) {
            showFontList(response);
        } else {
            String message = request.getParameter("message");
            if ((message == null) || (message.length() == 0)) {
                message = "Missing 'message' parameter";
            }
            String fontName = request.getParameter("fontName");
            if ((fontName == null) || (fontName.length() == 0)) {
                fontName = "Serif";
            }
            String fontSizeString = request.getParameter("fontSize");
            int fontSize;
            try {
                fontSize = Integer.parseInt(fontSizeString);
            } catch(NumberFormatException nfe) {
                fontSize = 90;
            }
            response.setContentType("image/jpeg");
            MessageImage.writeJPEG
                (MessageImage.makeMessageImage(message,
                                              fontName,
                                              fontSize),
                 response.getOutputStream());
        }
    }

    private void showFontList(HttpServletRequest response)

```

```

        throws IOException {
    PrintWriter out = response.getWriter();
    String docType =
        "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
        "Transitional//EN\">\n";
    String title = "Fonts Available on Server";
    out.println(docType +
        "<HTML>\n" +
        "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
        "<BODY BGCOLOR=\"#FDF5E6\">\n" +
        "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
        "<UL>"));
    String[] fontNames = MessageImage.getFontNames();
    for(int i=0; i<fontNames.length; i++) {
        out.println(" <LI>" + fontNames[i]);
    }
    out.println("</UL>\n" +
        "</BODY></HTML>");
}
}

```

清单7.8 MessageImage.java

```

package coreservlets;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;

/** Utilities for building images showing shadowed messages.
 * <P>
 * Requires JDK 1.4 since it uses the ImageIO class.
 * JDK 1.4 is standard with J2EE-compliant app servers
 * with servlets 2.4 and JSP 2.0. However, standalone
 * servlet/JSP engines require only JDK 1.3 or later, and
 * version 2.3 of the servlet spec requires only JDK
 * 1.2 or later. So, although most servers run on JDK 1.4,
 * this code is not necessarily portable across all servers.
 */

public class MessageImage {

    /** Creates an Image of a string with an oblique
     * shadow behind it. Used by the ShadowedText servlet.
     */
    public static BufferedImage makeMessageImage(String message,
                                                String fontName,
                                                int fontSize) {

        Font font = new Font(fontName, Font.PLAIN, fontSize);
        FontMetrics metrics = getFontMetrics(font);
        int messageWidth = metrics.stringWidth(message);
        int baselineX = messageWidth/10;
        int width = messageWidth+2*(baselineX + fontSize);
        int height = fontSize*7/2;
    }
}

```

```
int baselineY = height*8/10;
BufferedImage messageImage =
    new BufferedImage(width, height,
                      BufferedImage.TYPE_INT_RGB);
Graphics2D g2d = (Graphics2D)messageImage.getGraphics();
g2d.setBackground(Color.white);
g2d.clearRect(0, 0, width, height);
g2d.setFont(font);
g2d.translate(baselineX, baselineY);
g2d.setPaint(Color.lightGray);
AffineTransform origTransform = g2d.getTransform();
g2d.shear(-0.95, 0);
g2d.scale(1, 3);
g2d.drawString(message, 0, 0);
g2d.setTransform(origTransform);
g2d.setPaint(Color.black);
g2d.drawString(message, 0, 0);
return(messageImage);
}

public static void writeJPEG(BufferedImage image,
                             OutputStream out) {
    try {
        ImageIO.write(image, "jpg", out);
    } catch(IOException ioe) {
        System.err.println("Error outputting JPEG: " + ioe);
    }
}

public static void writeJPEG(BufferedImage image,
                             File file) {
    try {
        ImageIO.write(image, "jpg", file);
    } catch(IOException ioe) {
        System.err.println("Error writing JPEG file: " + ioe);
    }
}

public static String[] getFontNames() {
    GraphicsEnvironment env =
        GraphicsEnvironment.getLocalGraphicsEnvironment();
    return(env.getAvailableFontFamilyNames());
}

/** We need a Graphics object to get a FontMetrics object
 * (an object that says how big strings are in given fonts).
 * But, you need an image from which to derive the Graphics
 * object. Since the size of the "real" image will depend on
 * how big the string is, we create a very small temporary
 * image first, get the FontMetrics, figure out how
 * big the real image should be, then use a real image
 * of that size.
 */
private static FontMetrics getFontMetrics(Font font) {
    BufferedImage tempImage =
        new BufferedImage(1, 1, BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = (Graphics2D)tempImage.getGraphics();
```

```
    return(g2d.getFontMetrics(font));
}
}
```

清单 7.9(图 7.6)给出用作该 servlet 前端的 HTML 表单。图 7.7~图 7.10 展示出一些可能的结果。为了简化试验过程，清单 7.10 给出一个交互式的应用程序，使用它可以在命令行指定消息和字体名称，将图像输出到文件。

清单 7.9 ShadowedText.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JPEG Generation Service</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">JPEG Generation Service</H1>
Welcome to the <I>free</I> trial edition of our JPEG
generation service. Enter a message, a font name,
and a font size below, then submit the form. You will
be returned a JPEG image showing the message in the
designated font, with an oblique "shadow" of the message
behind it. Once you get an image you are satisfied with,
right-click on it (or click while holding down the SHIFT key)
to save it to your local disk.
<P>
The server is currently on Windows, so the font name must
be either a standard Java font name (e.g., Serif, SansSerif,
or Monospaced) or a Windows font name (e.g., Arial Black).
Unrecognized font names will revert to Serif. Press the
>Show Font List" button for a complete list.

<FORM ACTION="/servlet/core servlets.ShadowedText">
<CENTER>
  Message:
  <INPUT TYPE="TEXT" NAME="message"><BR>
  Font name:
  <INPUT TYPE="TEXT" NAME="fontName" VALUE="Serif"><BR>
  Font size:
  <INPUT TYPE="TEXT" NAME="fontSize" VALUE="90"><P>
  <INPUT TYPE="SUBMIT" VALUE="Build Image"><P>
  <INPUT TYPE="SUBMIT" NAME="showList" VALUE="Show Font List">
</CENTER>
</FORM>

</BODY></HTML>
```

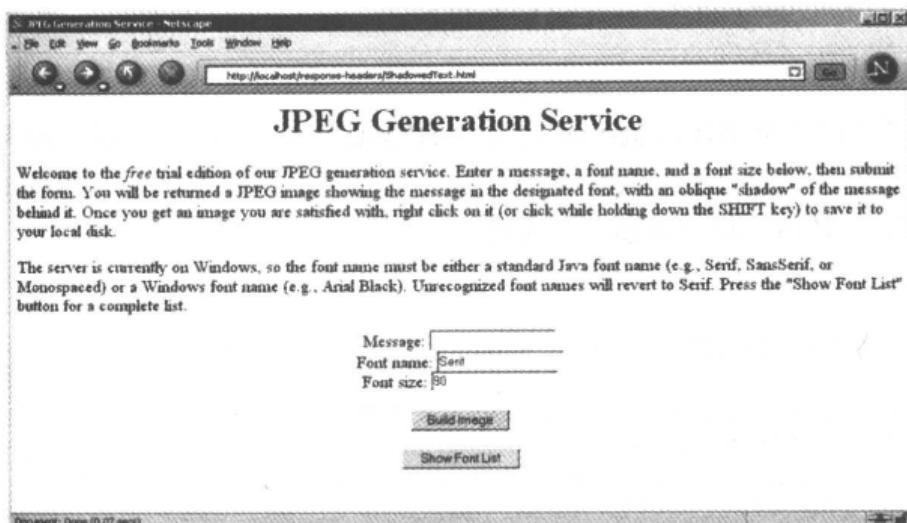


图 7.6 生成图像的 servlet 的前端

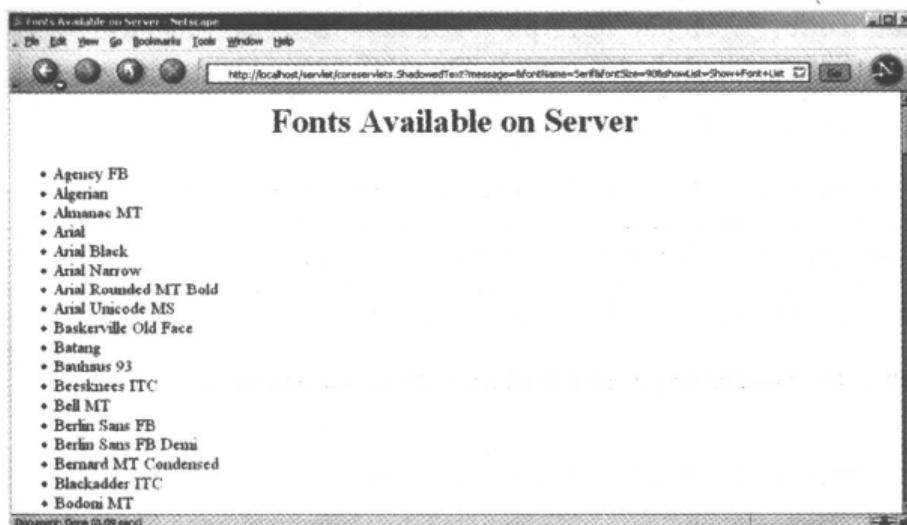


图 7.7 客户选择 Show Font List 时 servlet 的结果

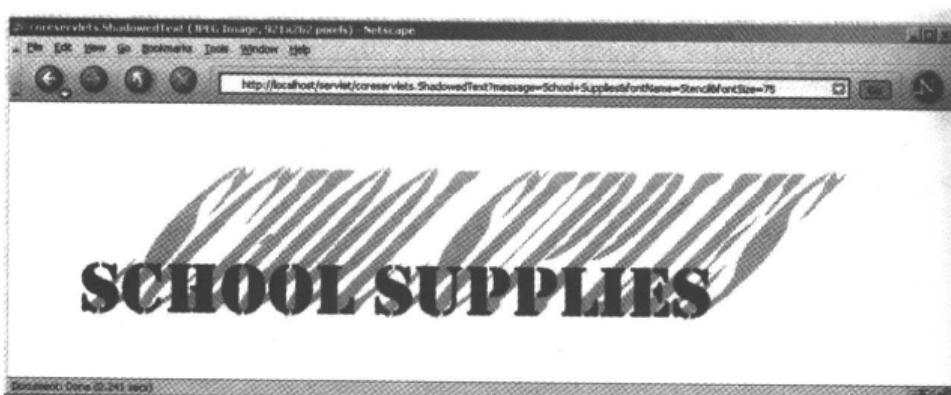


图 7.8 生成图像的 servlet 的一种可能结果。客户可以将图像存储成硬盘上的 somename.jpg，将它用在 Web 页面或其他应用程序中

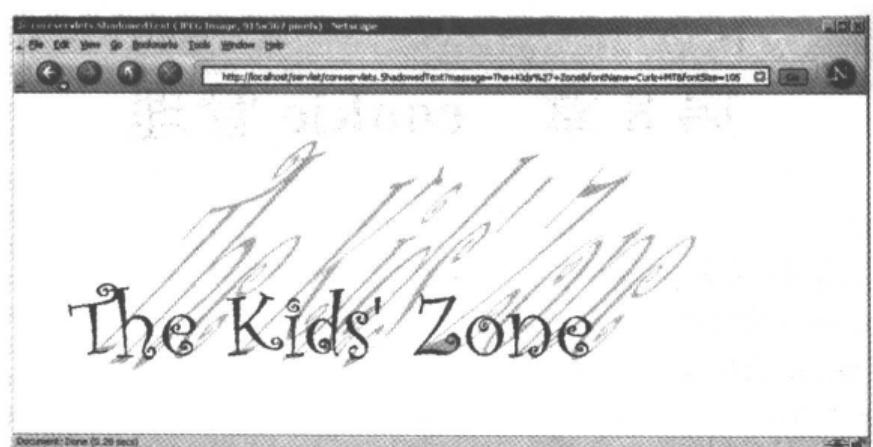


图 7.9 生成图像的 servlet 的第二种可能结果

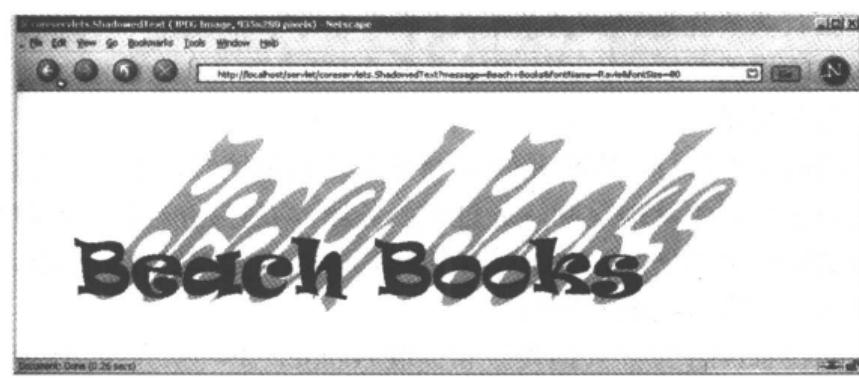


图 7.10 生成图像的 servlet 的第三种可能结果

清单7.10 ImageTest.java

```
package coreservlets;

import java.io.*;

public class ImageTest {
    public static void main(String[] args) {
        String message = "Testing";
        String font = "Arial";
        if (args.length > 0) {
            message = args[0];
        }
        if (args.length > 1) {
            font = args[1];
        }
        MessageImage.writeJPEG
            (MessageImage.makeMessageImage(message, font, 40),
             new File("ImageTest.jpg"));
    }
}
```

第 8 章 cookie 管理

本章的主题：

- cookie 的优点和缺点
- 输出 cookie 的发送
- 输入 cookie 的接收
- 跟踪重复用户
- cookie 属性的指定
- 会话 cookie 和持续性 cookie 之间的差异
- 用实用工具类简化 cookie 的使用
- cookie 值的修改
- 用户偏好的记录

cookie 是小段的文本信息，Web 服务器将它发送到浏览器，之后，在访问同一网站或域时，浏览器又将它原封不动地返回。通过让服务器读取它之前发送给客户端的信息，站点可以为访问者提供诸多便利，比如按照访问者之前的定制呈现该站点，或让身份可以验证的访问者进入，不需再次输入密码。

本章论述如何在 servlet 中显式地设置和读取 cookie，下一章介绍如何使用 servlet 会话跟踪 API(在后台可以使用 cookie 来实现)来关注用户对网站内不同页面的访问。

8.1 cookie 的优点

网站对 cookie 的利用方式一般有 4 种。下面，我们对这些优点进行汇总，在本节余下的部分中，给出相应的细节。

- 在电子商务会话中标识用户

这种类型的短期跟踪十分重要，为此，servlet 甚至专门在 cookie 的基础上构建了另外的 API 来完成这项任务。详细信息参见下一章。

- 记录用户名和密码

使用 cookie，可以让用户自动登录到站点，这为使用非共用计算机的用户提供了极大的便利。

- 定制站点

站点可以使用 cookie 来记录用户的偏好。

- 定向广告

站点可以使用 cookie，记录哪些主题引起特定用户的兴趣，并向他们显示与这些兴趣相关的广告。

8.1.1 在电子商务会话中标识用户

许多在线商店使用“购物车”来模拟现实世界中的购物环境，用户可以在商店中选择

商品，将它们加入到自己的购物车中，然后继续购物。由于每个页面发送之后，一般都会关闭 HTTP 连接，所以当用户选择新的商品加入到购物车中时，商店如何知道该用户就是将前面的商品放到购物车中的用户呢？持续性(继续使用)HTTP 连接并不能解决这个问题，因为持续连接一般只适用于在时间上极为接近的请求，比如浏览器对同一 Web 页面中图像的请求。此外，许多老版本的服务器和浏览器不支持持续性连接。但是，cookie 能够解决这个问题。实际上，这项功能十分有用，为此，servlet 甚至提供专门用于会话跟踪的 API，从而，servlet 和 JSP 的设计开发人员勿需直接操作 cookie 就能够利用它。会话跟踪在第 9 章论述。

8.1.2 记录用户名和密码

许多大型的站点，必须注册后才能使用它们的服务，但是，每次访问都要记住和输入用户名和口令，很不方便。对于那些对安全性要求比较低的站点，cookie 是个不错的选择。用户注册后，就会收到包含一个惟一用户 ID 的 cookie。客户后来重新连接时，这个用户 ID 会自动返回，服务器对它进行检查，确定它是否为注册用户且选择了自动登录，从而使用户勿需给出明确的用户名和密码，就可以访问服务器上的资源。站点可能还在数据库中存储着用户的地址、信用卡号码等，可以使用来自 cookie 的用户 ID 为键读取这些数据。使用这种方案，用户勿需每次都重新输入这些数据。

例如，当 Marty 到各个公司现场讲授 JSP 和 servlet 培训课程时，他一般会浏览 travelocity.com 和 expedia.com 获取航班信息。这两个网站在搜索航班时刻表时都要用到用户名和密码，但用户名中哪些字符合法，以及密码要求多少字符等方面规则各不相同。因而有一段时间，Marty 为了记住登录的用户名和密码而煞费周章。幸运的是，这两个网站都使用了前一段中讲述的 cookie 模式，简化了 Marty 从个人桌面计算机或膝上型计算机对这些网站的访问。

8.1.3 定制站点

许多“门户”网站都允许用户定制主页面的外观。它们允许用户选择希望看到哪个天气预报(是的，西雅图还在下雨)，选择应该显示哪些股票符号(是的，股票还在跌)，关注哪种体育战况(是的，Orioles 还在输)，搜索结果应该如何显示(是的，您希望每页显示多个结果)，依次类推。每次访问这些网站都必须对页面进行设置肯定会很不方便，因此，这些网站使用 cookie 记录用户的意愿。对于简单的设置，网站可以直接将页面的设置存储在 cookie 中完成定制。然而，对于更为复杂的定制，网站只需仅将一个惟一标识符发送给客户，由服务器端的数据存储与每个标识符对应的页面设置。

8.1.4 定向广告

许多由广告客户提供资金的网站，对显示“定向”(或“聚焦”)广告的收费要比显示“随机”广告多很多。广告客户一般愿意支付更多的费用，将他们的广告展示给那些对他们的产品目录感兴趣的受众。据报道，网站对定向广告的收费比随机广告多 30 倍。例如，如果您到一个搜索引擎上查找“Java Servlets”，那么，显示给您一个有关 servlet 开发环境

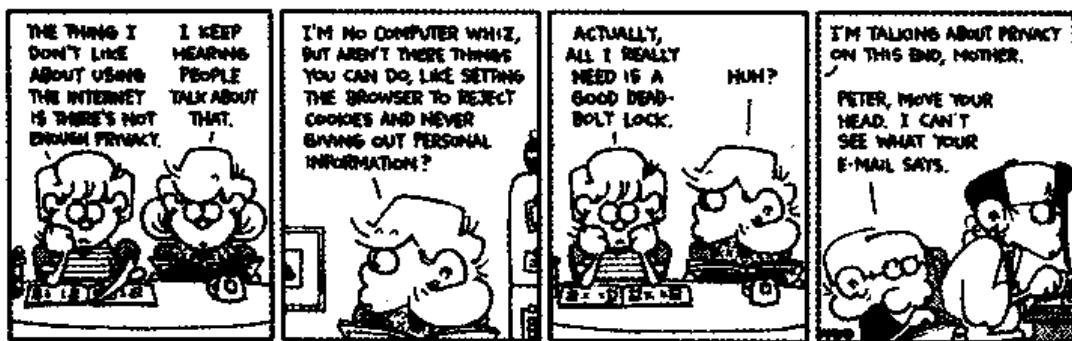
的广告，网站向广告客户的收费肯定要多于显示一个专门从事印度尼西亚线路的在线旅行社。如果搜索的是“Java Hotels”，情况将会相反。

您首次访问这些站点且未执行任何搜索时，由于尚不存在相关的 cookie，网站只能显示一个随机广告，如果您搜索的东西不和任何广告分类匹配时，也是如此。有 cookie 的情况下，通过记录之前的搜索，这些网站可以识别出您的兴趣。由于这种方式使得他们可以在用户访问主页和结果页面时显示定向广告，这几乎将他们的广告收入扩大了一倍。

8.2 cookie 存在的一些问题

向用户提供便利以及为网站的所有者增值是 cookie 背后的驱动力。尽管存在许多错误的认识，但 cookie 并不构成严重的安全威胁。cookie 从不会以任何方式得到解释或执行，因而也就不能插入病毒或攻击用户的系统。此外，由于浏览器一般对每个站点只接受 20 个 cookie，总共不超过 300 个，同时浏览器可以将每个 cookie 限制在 4K，因此 cookie 不能用来填满某人的硬盘或启动其他拒绝服务(denial-of-service, DoS)攻击。

但是，虽然 cookie 不能造成严重的安全威胁，但却有可能对隐私造成极大的威胁。



FOXTROT ©1998 Bill Amend. 经 UNIVERSAL PRESS SYNDICATE 授权翻印。版权所有

首先，一些人不喜欢搜索引擎可以记录他们之前搜索的内容。例如，他们可能搜索职位或敏感的健康数据，不希望同事或老板下次搜索时，看到有关的旗帜广告。此外，搜索引擎不需要使用旗帜广告：设计差劲的搜索引擎可能会显示一个文本区域，列出您最近的查询（“在哪个地方就职都行，除这个愚蠢公司之外！”；“我的 SARS 感染会伤害我的同事吗？”等）。您的同事在您的计算机上使用搜索引擎时，或者当您访问时站在您的背后观察屏幕，都可能会看到这些信息。

更坏的情况，两个网站可能通过从相同的第三方网站载入小幅的图像，共享用户的相关数据，由这个第三方网站使用 cookie 记录用户的数据，且与原来这两个网站共享数据。例如，假定 some-search-site.com 和 some-random-site.com 都想根据用户在 some-search-site.com 搜索的内容，显示来自 some-ad-site.com 的定向广告。如果用户搜索了“Java Servlets”，那么 some-search-site.com 的搜索引擎可能会返回含有下面图像链接的页面：

```
<IMG SRC="http://some-ad-site.com/banner?data=Java+Servlets" ...>
```

由于浏览器会建立到 some-ad-site.com 的 HTTP 连接，因此 some-ad-site.com 可以向浏

览器返回一个持续性 cookie。接下来，some-random-site.com 可能返回如下的图像链接：

```
<IMG SRC="http://some-ad-site.com/banner" ...>
```

由于浏览器会重新连接到 some-ad-site.com——之前浏览器从中得到 cookie 的网站，浏览器将返回前面接收到的 cookie。假定 some-ad-site.com 发送了一个惟一的 cookie 值，并且，在它的数据库中，将这个 cookie 值与“Java Servlets”搜索关联起来，那么即使用户第一次访问 some-random-site，some-ad-site 也能够返回定向标题广告。doubleclick.net 服务是这项技术最著名的早期应用。(但是，Netscape 和 Internet Explorer 的最近版本都提供一项不错的特性，使用它可以拒绝来自于所连接网站以外的 cookie，但并非将 cookie 完全禁用。参见图 8.1。)

如果您使用的邮件阅读器能够处理 HTML、支持 cookie 且与浏览器相关联，那么，这种将 cookie 与图像关联的技巧甚至可以通过电子邮件来加以利用。因此，人们能够向您发送需要载入图像的电子邮件，向这些图像附加 cookie，如果您后来访问他们的网站，就可以将您识别出来(电子邮件地址或其他各种信息)。

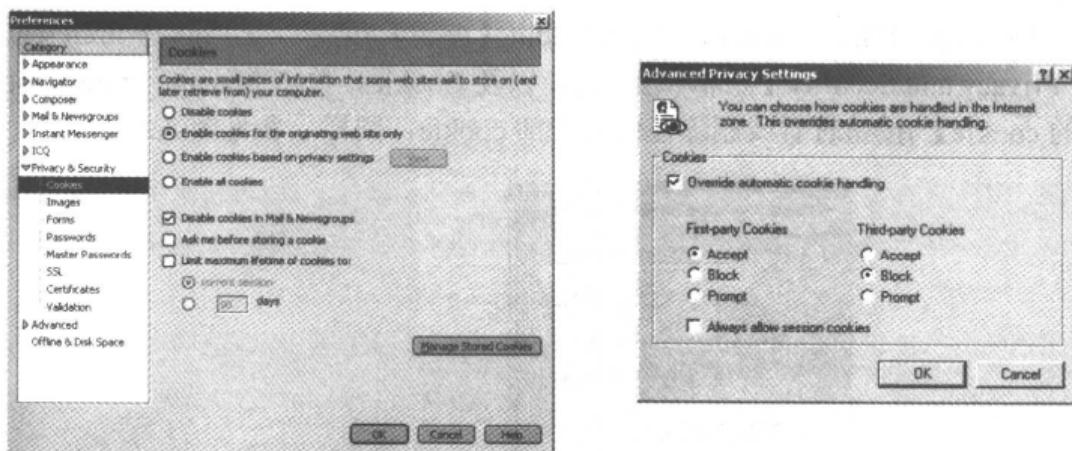


图 8.1 Netscape(左)和 Internet Explorer(右)中 cookie 的自定义设置

当站点依赖 cookie 处理过度敏感的数据时，第二个隐私问题就会出现。例如，某些大型的在线书店使用 cookie 记录用户的注册信息，并允许用户只输出少量的个人信息就可以下订单。由于它们并不真正显示用户的完整信用卡号码，并且只允许用户将书籍发送到输入完整的信用卡号码或用户名和密码时指定的地址。因此，使用您的计算机(或窃取了 cookie 文件)的人除了向您的地址发送大额的订单以外(订单可以拒绝)，不能对您造成其他伤害。但是，其他公司可能没有这么小心，能够使用其他人的计算机或 cookie 文件的攻击者有可能会访问到在线的个人信息。更为不利的是，有些不合格的站点可能会将信用卡或其他敏感信息直接嵌入到 cookie 中，而非使用无害的、只存在于服务器上的与实际用户相关联的标识符。由于在无人看管的情况下离开办公室的计算机时，大多数用户并不担心安全问题，不像将他们的信用卡遗留在办公桌上时那般紧张，故而这种嵌入方式十分危险。

这项讨论的要点有两个方面：

- (1) 由于现实的和可以预见的隐私问题，某些用户选择关闭 cookie。因此，即使只是使用 cookie 增加网站的附加值，也要尽可能地不依赖于它们。某些情况下，对 cookie 的依赖是难以避免的，但是，如果能够在禁用 cookie 的情况下也为用户提

供合理的功能，这样要好得多。

- (2) 作为 servlet 和 JSP 页面的设计开发人员，如果在 servlet 或 JSP 页面中用到 cookie，应该避免用 cookie 存储特别敏感的信息，因为，如果其他人访问到用户的计算机或 cookie 文件，则会将用户置于危险之中。

8.3 cookie 的删除

您可能会发现，定期删除 cookie(或起码是与 localhost 或服务器所在主机相关联的那些 cookie)会使得试验本章的例子更为容易。

要删除 Internet Explorer 中的 cookie，请选择 Tools(【工具】)菜单，并选择 Internet Options(【Internet 选项】)。如果要删除所有的 cookie，请单击 Delete Cookies(【删除 cookie】)。如果有选择地删除 cookie，请单击 Settings(【设置】)，然后选 View Files(【查看文件】)(cookie 文件的名称以 Cookie:开始，选择 View Files 之前，先选 Delete Files(【删除文件】)，更容易找出它们)。参见图 8.2。

要在 Netscape 中删除 cookie，请选择 Edit(【编辑】)菜单，之后选择 Preferences(【首选项】)，Privacy and Security(【隐私和安全】)，以及 Cookies。单击 Manage Stored Cookie(【管理存储的 cookie】)按钮查看或删除任何或全部 cookie。同样，参见图 8.2。

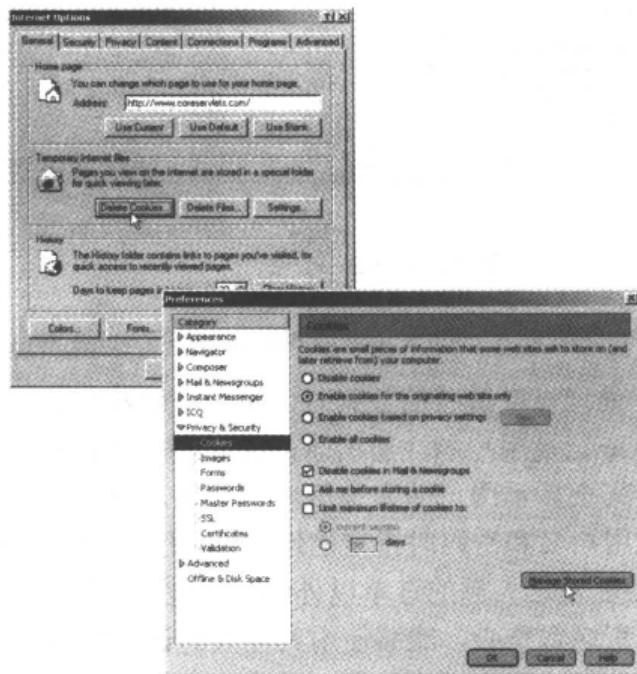


图 8.2 Internet Explorer 和 Netscape 中 cookie 的删除

8.4 cookie 的发送和接收

要将 cookie 发送到客户端，servlet 应该使用 Cookie 的构造函数，用指定的名称和值，创建一个或多个 cookie，用 cookie.setXxx(之后可以使用 cookie.getXxx 读取)设置任何可选的属性，并用 response.addCookie 将 cookie 插入到 HTTP 响应报头中。

要读取输入的 cookie, servlet 应该调用 `request.getCookies`, 这个方法返回 `Cookie` 对象的数组, 对应浏览器与您的网站关联的 cookie(如果请求中没有 cookie, 则为 null)。大多数情况下, servlet 应该循环处理这个数组, 调用每个 cookie 的 `getName` 方法, 直到找到名称与查找的名称相匹配的 cookie 为止, 然后调用该 cookie 的 `getValue` 方法, 检查与这个名称相关联的值。随后的小节中对每个主题进行详细论述。

8.4.1 向客户程序发送 cookie

发送 cookie 到客户程序涉及 3 个步骤(以下进行了汇总, 随后给出相应的细节)。

(1) **创建 Cookie 对象。**

调用 `Cookie` 的构造函数, 给出 cookie 的名称和 cookie 的值, 二者都是字符串。

(2) **设置最大时效。**

如果希望浏览器将 cookie 存储在磁盘上, 而非只是将它保持在内存中, 可以使用 `setMaxAge` 指定多长时间(以秒为单位)内 cookie 是合法的。

(3) **将 Cookie 放入到 HTTP 响应报头。**

使用 `response.addCookie` 完成这项任务。如果您忘记了这一步, 那么 cookie 不会发送给浏览器。

1. 创建 Cookie 对象

通过调用 `Cookie` 的构造函数可以创建一个 cookie, 这个方法接受两个字符串: cookie 的名称和 cookie 的值。不管是名称还是值都不应该含有空格或任何下列字符:

[] () = , " / ? @ : ;

例如, 如果要创建一个名为 `userID` 的 cookie, 并将它的值设为 `a1234`, 应该使用下面的语句:

```
Cookie c = new Cookie("userID", "a1234");
```

2. 设置最大时效

如果您创建一个 cookie, 并将它发送到浏览器, 默认情况下它是一个会话级别的 cookie: 存储在浏览器的内存中, 用户退出浏览器之后被删除。如果您希望浏览器将该 cookie 存储在磁盘上, 则需要使用 `maxAge`, 并给出一个以秒为单位的时间, 如下所示:

```
c.setMaxAge(60*60*24*7); // One week
```

由于可以使用会话跟踪 API(第 9 章)来简化大部分使用会话级 cookie 的任务, 在使用 `Cookie API` 时, 几乎总会用到 `setMaxAge` 方法。

将最大时效设为 0 则是命令浏览器删除该 cookie。

核心方法

创建 `Cookie` 对象后, 将 cookie 发送到客户程序之前, 一般应该调用 `setMaxAge` 方法。

要注意, `setMaxAge` 并非 `Cookie` 唯一可以修改的特性。其他一些较少使用的特性在 8.6

节中论述。

3. 将 Cookie 放入到 HTTP 响应报头

创建 Cookie 对象和调用 setMaxAge，只不过是操作服务器内存中的数据结构。并没有实际向浏览器发送任何内容。如果不将 cookie 发送到客户程序，那么它不会起到任何作用。虽然这是显而易见的，但初级开发人员常犯的一个错误就是创建和操作 Cookie 对象，但没有将它们发送到客户程序。

警告

创建和操作 Cookie 对象对客户程序没有任何影响。必须显式地用 response.addCookie 将 cookie 发送到客户端。

发送 cookie 需要使用 HttpServletResponse 的 addCookie 方法，将 cookie 插入到一个 Set-Cookie HTTP 请求报头中。由于这个方法并不修改任何之前指定的 Set-Cookie 报头，而是创建新的报头，因此我们将这个方法称为是 addCookie，而非 setCookie。同样，要记住，响应报头必须在任何文档内容发送到客户端之前设置。

下面就是一个具体的例子：

```
Cookie userCookie = new Cookie("user", "uid1234");
userCookie.setMaxAge(60*60*24*365); // Store cookie for 1 year
response.addCookie(userCookie);
```

8.4.2 从客户端读取 cookie

要将 cookie 发送到客户程序，首先要创建 Cookie，设置它的最大时效(一般情况下)，然后使用 addCookie 发送一个 Set-Cookie HTTP 响应报头。而要读取客户程序发送回来的 cookie，应该执行下述两项任务，下面首先对这两项任务进行汇总，在随后的小节中给出更为详细的信息。

(1) 调用 request.getCookies。

可以得到一个 Cookie 对象的数组。

(2) 对数组进行循环，调用每个 cookie 的 getName 方法，直到找到感兴趣的 cookie 为止。

之后，一般调用 getValue，以应用程序特有的方式使用这个值。

1. 调用 request.getCookies

要获取由浏览器发送来的 cookie，需要调用 HttpServletRequest 的 getCookies 方法。这个调用返回 Cookie 对象的数组，对应由 HTTP 请求中 Cookie 报头输入的值。如果请求中不含有 cookie，getCookies 应该返回 null。但是，要注意，老版本的 Apache Tomcat(3.x 版本)有一个 bug，它返回零长度的数组，而非 null。

2. 循环 Cookie 数组

有了 cookie 的数组之后，一般会对它进行循环，调用每个 Cookie 的 getName 方法，直到找出一个与您希望的名称相匹配的对象为止。回想一下，cookie 与您的主机(或域)相关，

而非您的 servlet(或 JSP 页面)。因而，尽管您的 servlet 可能发送单个 cookie，您可能会得回许多不相关的 cookie。找到感兴趣的 cookie 之后，一般要调用它的 getValue 方法，并根据得出的值完成一些具体的处理。例如：

```
String cookieName = "userID";
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for(int i=0; i<cookies.length; i++) {
        Cookie cookie = cookies[i];
        if (cookieName.equals(cookie.getName())) {
            doSomethingWith(cookie.getValue());
        }
    }
}
```

这个过程很普遍，因此在 8.8 节中，我们提供两个实用工具类，利用它们可以简化指定名称的 cookie 以及相应 cookie 值的获取任务。

8.5 使用 cookie 检测初访者

假定您想为首次到达的访问者显示一个突出的旗帜，告诉他们去注册。同时，又不希望向再次到来的访问者显示这个无用的旗帜，将显示搞得混乱。

cookie 是区分初访者和再访者的完美方式。检查唯一命名的 cookie 是否存在：如果存在，客户是再访者；如果不存在该 cookie，那么该访问者就是一个初来者。您应该设置一个输出 cookie，说明“这个用户已经来过这里”。

尽管这种思想比较直观简单，但还是要注意十分重要的一点：不能仅仅因为 cookie 数组中不存在特定的数据项就认为用户是个初来者。许多初级的 servlet 程序员都会错误地使用下面的方式：

```
Cookie[] cookies = request.getCookies();
if (cookies == null)
    doStuffForNewbie();           // Correct.
} else {
    doStuffForReturnVisitor();   // Incorrect.
}
```

这种做法是错误的！确实，如果 cookie 数组为 null，客户是一个初来者(至少就目前来说如此——这也可能是由于用户将 cookie 删除或禁用造成的结果)。但是，如果数组非 null，也不过是显示客户曾经到过您的网站(或域——参见下一节中的 setDomain)，并不能说明他们曾经访问过您的 servlet。其他 servlet、JSP 页面以及非 Java Web 应用都可以设置 cookie，依据路径的设置(见下一节中的 setPath)，其中的任何 cookie 都有可能返回给用户的浏览器。

清单 8.1 阐明了检查指定 cookie 的正确方法。图 8.3 和图 8.4 展示出初始访问和后续访问的结果。

清单 8.1 RepeatVisitor.java

```
package coreservlets;
import java.io.*;
```

```

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

/** Servlet that says "Welcome aboard" to first-time
 * visitors and "Welcome back" to repeat visitors.
 * Also see RepeatVisitor2 for variation that uses
 * cookie utilities from later in this chapter.
 */
public class RepeatVisitor extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        boolean newbie = true;
        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for(int i=0; i<cookies.length; i++) {
                Cookie c = cookies[i];
                if ((c.getName().equals("repeatVisitor")) &&
                    // Could omit test and treat cookie name as a flag
                    (c.getValue().equals("yes"))) {
                    newbie = false;
                    break;
                }
            }
        }
        String title;
        if (newbie) {
            Cookie returnVisitorCookie =
                new Cookie("repeatVisitor", "yes");
            returnVisitorCookie.setMaxAge(60*60*24*365); // 1 year
            response.addCookie(returnVisitorCookie);
            title = "Welcome Aboard";
        } else {
            title = "Welcome Back";
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                    "</BODY></HTML>");
    }
}

```



图 8.3 客户对 RepeatVisitor servlet 的初次访问



图 8.4 客户对 RepeatVisitor servlet 的后续访问

8.6 使用 cookie 属性

在将 cookie 加到输出报头之前，可以使用下面的 `setXxx` 方法设置 cookie 的各项特性，`Xxx` 是希望设置的属性名称。

尽管每个 `setXxx` 方法都有一个对应的 `getXxx` 方法来取出属性的值，但要注意，属性是从服务器发送到浏览器的报头的一部分；但它们不属于由浏览器返回给服务器的报头。因而，除了名称和值以外，cookie 属性只适用于从服务器输出到客户端的 cookie；服务器端来自于浏览器的 cookie 并没有设置这些属性。因而，不要期望通过 `request.getCookies` 得到的 cookie 中可以使用这些属性。这意味着，您不能仅仅通过设置 cookie 的最大时效，发出它，在随后请求的输入数组中查找适当的 cookie，读取它的值，修改它并将它存回 Cookie，从而实现不断改变的 cookie 值。每次都必须调用 `setMaxAge`(当然，还要将 Cookie 传递给 `response.addCookie`)。

下面就是设置 cookie 属性的方法。

- (1) `public void setComment(String comment)`

public String getComment()

这些方法指定或查找与 cookie 相关的注释。对于 0 版本的 cookie(参见随后的 `setVersion` 和 `getVersion`)，该注释纯粹用于信息目的，存在于服务器上，并不发送给客户程序。

- (2) `public void setDomain(String domainPattern)`

public String getDomain()

这些方法设置或读取 cookie 适用的域。一般地，浏览器只将 cookie 返回给主机名与发送该 cookie 的主机名完全相同的主机。例如，由 `bali.vacations.com` 发送来的 cookie，一般不会被浏览器返回给 `queensland.vacations.com` 的页面。如果网站希望这种行为，servlet 可以指定 `cookie.setDomain(".vacations.com")`。为了防止服务器设置适用于它们的域之外的主机的 cookie，指定的域必须满足下面的要求：它必须以点号开始(例如 `.coreservlets.com`)；对于非国家域，如 `.com`, `.edu`, `.gov` 等，它必须含有 2 个点号；同时，对于国家域名，如 `.co.uk` 和 `.edu.es`，它必须含有 3 个点号。

- (3) `public void setMaxAge(int lifetime)`

public int getMaxAge()

这些方法规定 cookie 多长时间之后过期。负值(默认值)表明 cookie 仅仅用于当前浏览会话(即直到用户退出浏览器为止)，并不存储到磁盘上。参见 `LongLivedCookie`

类(清单 8.4), 该类定义了 Cookie 的一个子类, 并自动将最长时效设为一年。指定 0 值则是指示浏览器删除该 cookie。

(4) **public String getName()**

getName 方法读取 cookie 的名称。名称和值实际上总是使用者真正关注的两件事。但由于名称是提供给 Cookie 的构造函数的, 因此没有 setName 方法; cookie 创建之后就不能更改它的名称。另一方面, getName 方法几乎用到服务器接收的每个 cookie 上。由于 HttpServletRequest 的 getCookies 方法返回 Cookie 对象的数组, 因此通常的做法是循环处理该数组, 调用 getName 直到找到特定的名称, 然后用 getValue 检查它的值。有关这个过程的封装, 参见清单 8.3 中的 getCookieValue 方法。

(5) **public void setPath(String path)**

public String getPath()

这些方法设置或取得 cookie 所适用的路径。如果没有指定一个路径, 浏览器只将该 cookie 返回给发送 cookie 的页面所在目录中或之下的 URL。例如, 如果服务器从 *http://ecommerce.site.com/toys/specials.html* 发送 cookie, 那么浏览器会在连接到 *http://ecommerce.site.com/toys/bikes/beginners.html* 时送回该 cookie, 但连接到 *http://ecommerce.site.com/cds/classical.html* 时则不会。setPath 方法可以指定更为通用的东西。例如, cookie.setPath("/")指定服务器的所有页面都应该收到该 cookie。指定的路径必须包括当前页面; 也就是说, 您可以指定比默认路径更为广义的路径。因此, 位于 *http://host/store/cust-service/request* 的 servlet 可以指定路径/*store/*(由于/*store/*包括/*store/cust-service/*), 但不能指定/*store/cust-service/returns/*(因为这个目录不包括/*store/cust-service/*)。

核心方法

如果要指定 cookie 适用于您的网站上所有的 URL, 使用 cookie.setPath("/")。

(6) **public void setSecure(boolean secureFlag)**

public boolean getSecure()

这一对方法设置或取得相应的布尔值, 表示 cookie 是否只能通过加密连接(即 SSL)发送。默认值是 false, 表示 cookie 适用于所有的连接。

(7) **public void setValue(String cookieValue)**

public String getValue()

setValue 方法指定与该 cookie 相关联的值; getValue 找出这个值。同样, 名称和值是您对 cookie 几乎总是关注的两部分, 尽管少数情况下会将名称用作布尔标志, 忽略它的值(即只关注指定名称的 cookie 存在与否)。然而, 由于 cookie 的值已提供了 Cookie 的构造函数, setValue 一般只用于改变输入 cookie 的值, 并将它们发送出去。具体的例子, 参见 8.10 节。

(8) **public void setVersion(int version)**

public int getVersion()

这些方法设置和取得 cookie 遵从的 cookie 协议版本。版本 0，默认值，遵循最初的 Netscape 规范(http://wp.netscape.com/newsref/std/cookie_spec.html)。版本 1 尚未被广泛支持，它依据 RFC 2109(可以从 <http://www.rfc-editor.org/> 列出的档案网站获取 RFC)。

8.7 区分会话 cookie 与持续性 cookie

本节通过对设置最大时效和未设置最大时效的 cookie 的行为，阐述 cookie 属性的使用。清单 8.2 列出了 CookieTest servlet，它执行下面两项任务：

- (1) 首先，该 servlet 设置 6 个输出 cookie。3 个没有明确设置时效(即默认情况下的负值)，意指它们只适用于当前浏览会话——直到用户重新启动浏览器为止。另外 3 个使用 setMaxAge 规定浏览器应该将它们写到磁盘上，并且应该保持一个小时，而不管用户通过哪种方式初始化新的浏览会话——重启浏览器还是重启计算机。
- (2) 其次，该 servlet 使用 request.getCookies 查找所有的输入 cookie，将它们的名称和值显示在一个 HTML 表格中。

图 8.5 展示出初次访问的结果，图 8.6 展示出紧随其后访问的结果，图 8.7 展示出用户重启浏览器之后访问的结果。

清单 8.2 CookieTest.java

```
package coreservlets;

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

/** Creates a table of the cookies associated with
 * the current page. Also sets six cookies: three
 * that apply only to the current session
 * (regardless of how long that session lasts)
 * and three that persist for an hour (regardless
 * of whether the browser is restarted).
 */

public class CookieTest extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        for(int i=0; i<3; i++) {
            // Default maxAge is -1, indicating cookie
            // applies only to current browsing session.
            Cookie cookie = new Cookie("Session-Cookie-" + i,
                                         "Cookie-Value-S" + i);
            response.addCookie(cookie);
            cookie = new Cookie("Persistent-Cookie-" + i,
                                "Cookie-Value-P" + i);
            // Cookie is valid for an hour, regardless of whether
            // user quits browser, reboots computer, or whatever.
            cookie.setMaxAge(3600);
            response.addCookie(cookie);
        }
    }
}
```

```

}
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String docType =
"<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
"Transitional//EN\">\n";
String title = "Active Cookies";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
    "<BODY BGCOLOR=#FDF5E6>\n" +
    "<H1 ALIGN=\\\"CENTER\\\">" + title + "</H1>\n" +
    "<TABLE BORDER=1 ALIGN=\\\"CENTER\\\">\n" +
    "<TR BGCOLOR=#FFAD00>\n" +
    " <TH>Cookie Name\n" +
    " <TH>Cookie Value");
Cookie[] cookies = request.getCookies();
if (cookies == null) {
    out.println("<TR><TH COLSPAN=2>No cookies");
} else {
    Cookie cookie;
    for(int i=0; i<cookies.length; i++) {
        cookie = cookies[i];
        out.println("<TR>\n" +
            " <TD>" + cookie.getName() + "\n" +
            " <TD>" + cookie.getValue());
    }
}
out.println("</TABLE></BODY></HTML>");
}
}

```

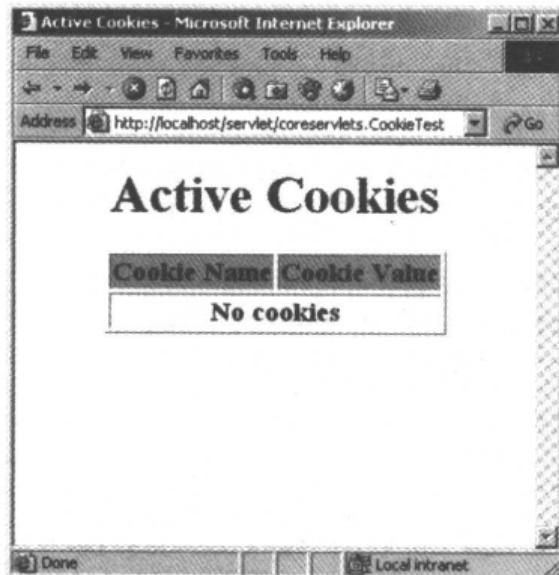


图 8.5 对 CookieTest servlet 的首次访问。访问该 servlet，退出浏览器，等一个小时，然后再次访问该 servlet，其结果相同

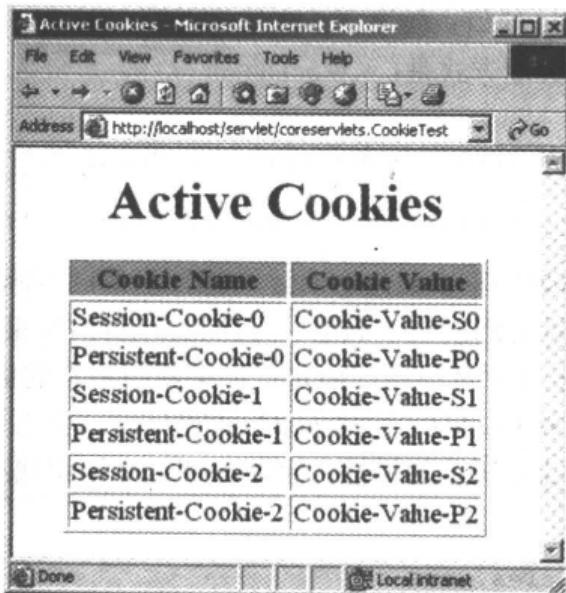


图 8.6 首次访问后一个小时之内，在同一个浏览器会话中(初次访问和此次访问之间浏览器保持打开)再次访问 CookieTest servlet 的结果

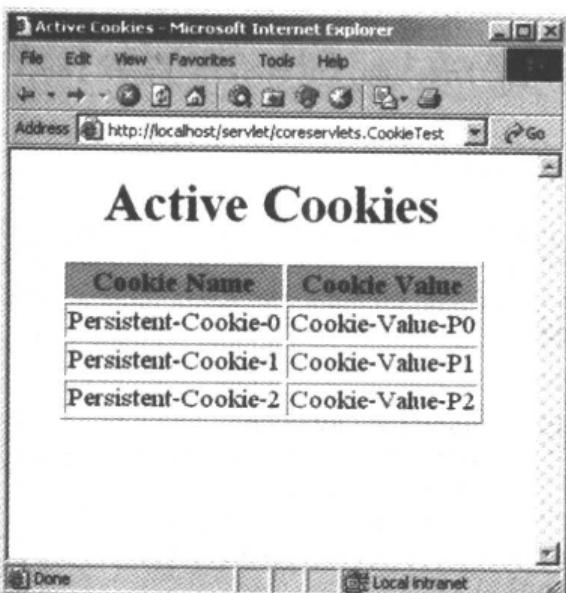


图 8.7 首次访问后一个小时之内，在不同的浏览器会话中(初次访问和此次访问之间浏览器重新启动过)再次访问 CookieTest servlet 的结果

8.8 基本的 cookie 实用程序

本节介绍一些简单但是有用 cookie 处理实用程序。

8.8.1 查找指定名称的 cookie

清单 8.3 给出 CookieUtilities 类中的两个静态方法，它们简化了给定名称的 cookie 或 cookie 值的获取。getCookieValue 方法遍历由所有 Cookie 对象组成的数组，返回名称与它

的输入相匹配的任何 Cookie 的值。如果不存在这类匹配，则返回指定的默认值。举个例子来说，我们处理 cookie 的典型方式是：

```
String color =
    CookieUtilities.getCookieValue(request, "color", "black");
String font =
    CookieUtilities.getCookieValue(request, "font", "Arial");
```

getCookie 方法也能够遍历整个数组，对比名称，但返回实际的 Cookie 对象，而非仅仅 cookie 的值。如果您不只是想读取它的值，而且还希望对 Cookie 做一些处理，则使用这个方法。getCookieValue 方法更常使用，但是，如果希望将新的值放入到 cookie 中，并将它再次送回时，可能会用 getCookie，而非 getCookieValue。但不要忘记，如果使用这种方式，那么必须重新指定 cookie 所有的属性，比如日期、路径和域：输入 cookie 中没有设置这些属性。

清单 8.3 CookieUtilities.java

```
package coreservlets;

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

/** Two static methods for use in cookie handling.
 */

public class CookieUtilities {

    /** Given the request object, a name, and a default value,
     * this method tries to find the value of the cookie with
     * the given name. If no cookie matches the name,
     * the default value is returned.
     */
    public static String getCookieValue
        (HttpServletRequest request,
         String cookieName,
         String defaultValue) {
        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for(int i=0; i<cookies.length; i++) {
                Cookie cookie = cookies[i];
                if (cookieName.equals(cookie.getName())) {
                    return(cookie.getValue());
                }
            }
        }
        return(defaultValue);
    }

    /** Given the request object and a name, this method tries
     * to find and return the cookie that has the given name.
     * If no cookie matches the name, null is returned.
     */
    public static Cookie getCookie(HttpServletRequest request,
```

```

        String cookieName) {
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for(int i=0; i<cookies.length; i++) {
        Cookie cookie = cookies[i];
        if (cookieName.equals(cookie.getName())) {
            return(cookie);
        }
    }
}
return(null);
}
}

```

8.8.2 创建长生存期的 cookie

清单 8.4 给出一个小型的类，如果希望客户退出浏览器时 cookie 自动保持一年，可以使用它来替代 Cookie。这个类(LongLivedCookie)只是对 Cookie 进行了扩展，自动调用 setMaxAge。

清单 8.4 LongLivedCookie.java

```

package coreservlets;

import javax.servlet.http.*;

/** Cookie that persists 1 year. Default Cookie doesn't
 * persist past current browsing session.
 */

public class LongLivedCookie extends Cookie {
    public static final int SECONDS_PER_YEAR = 60*60*24*365;

    public LongLivedCookie(String name, String value) {
        super(name, value);
        setMaxAge(SECONDS_PER_YEAR);
    }
}

```

8.9 实际使用 cookie 实用程序

清单 8.5 重写了清单 8.1 中的 RepeatVisitor servlet。新版本(RepeatVisitor2)和老版本的功能相同：它向初访者显示“Welcome Aboard”，向重访者显示“Welcome Back”。然而，它使用 8.8 节中的 cookie 实用程序，通过两种方式对代码进行了简化：

- (1) 这段代码不再调用 request.getCookies 并循环处理数组，检查每个名称，而是只调用 CookieUtilities.getCookieValue。
- (2) 不再创建 Cookie 对象，计算一年的秒数，之后调用 setMaxAge，而是只创建一个 LongLivedCookie 对象。

图 8.8 和图 8.9 给出了相应地结果。

清单 8.5 RepeatVisitor2.java

```

package coreservlets;

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

/** A variation of the RepeatVisitor servlet that uses
 * CookieUtilities.getCookieValue and LongLivedCookie
 * to simplify the code.
 */

public class RepeatVisitor2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        boolean newbie = true;
        String value =
            CookieUtilities.getCookieValue(request, "repeatVisitor2",
                                            "no");
        if (value.equals("yes")) {
            newbie = false;
        }
        String title;
        if (newbie) {
            LongLivedCookie returnVisitorCookie =
                new LongLivedCookie("repeatVisitor2", "yes");
            response.addCookie(returnVisitorCookie);
            title = "Welcome Aboard";
        } else {
            title = "Welcome Back";
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    "</BODY></HTML>");
    }
}

```



图 8.8 客户对 RepeatVisitor2 servlet 的首次访问



图 8.9 客户对 RepeatVisitor2 servlet 的后续访问

8.10 修改 cookie 的值：记录用户的访问计数

在前面给出的例子中，我们只在用户首次到访时发送一个 cookie。该 cookie 有了值之后，我们从不去改变它。由于 cookie 常常除了含有惟一的用户标识符之外没有其他内容，故而这种单一 cookie 值的方式十分常见：用户的实际数据存储在数据库中——用户标识符只是数据库的键而已。

但是，如果希望周期性地修改 cookie 的值应该怎么办呢？如何做到这一点呢？

- 要替换 cookie 之前的数据，需要发送相同的 cookie 名称，但要使用不同的 cookie 值。如果您想使用输入 Cookie 对象，那么不要忘记调用 response.addCookie；只是调用 setValue 是没有效果的。还需要调用 setMaxAge, setPath 等，重新应用所有的相关 cookie 属性——输入 cookie 中并没有指定 cookie 属性。由于要重新应用这些属性，因此重用输入 Cookie 对象可以节省的工作有限，故而许多开发人员不使用这种方式。
 - 要指示浏览器删除一个 cookie，只需使用 setMaxAge 将它的最大时效设为 0。

清单 8.6 给出的 servlet 跟踪每个客户对特定页面的访问次数。它通过生成名为 accessCount 的 cookie(它的值为实际的访问计数), 完成这项任务。在这个过程中, servlet 需要用相同的名称再次发送 cookie, 不断地替换该 cookie 的值。

图 8.10 给出一些典型的结果。

清单8.6 ClientAccessCounts.java

```

int count = 1;
try {
    count = Integer.parseInt(countString);
} catch(NumberFormatException nfe) { }
LongLivedCookie c =
    new LongLivedCookie("accessCount",
        String.valueOf(count+1));
response.addCookie(c);
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "Access Count Servlet";
String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +"
    "Transitional//EN\">\n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
    "<CENTER>\n" +
    "<H1>" + title + "</H1>\n" +
    "<H2>This is visit number " +
    count + " by this browser.</H2>\n" +
    "</CENTER></BODY></HTML>");
}
}

```

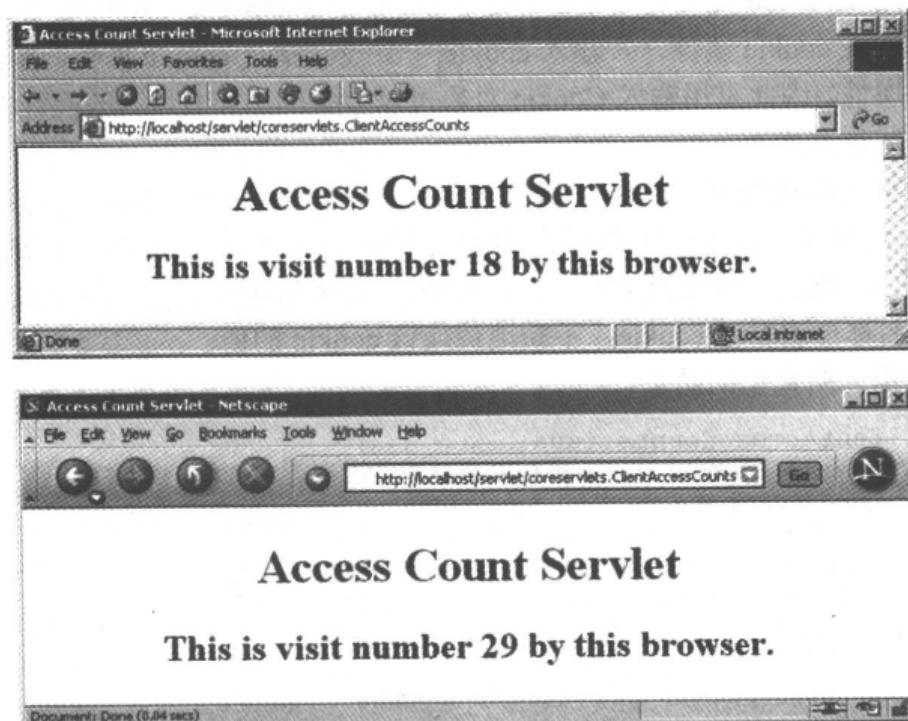


图 8.10 每个用户都会看到他们自己的访问计数。同样，Internet Explorer 和 Netscape 独立地维护 cookie，因此同一个用户在这两种浏览器中会看到独立的访问计数

8.11 使用 cookie 记录用户的偏好

cookie 最常见的应用之一就是用来“记录”用户的偏好。对于简单的用户设置，可以

直接将用户的偏好存储在 cookie 中。对于更为复杂的应用，一般在 cookie 中存储惟一的用户标识，而将实际的偏好存储在数据库中。

清单 8.7 给出的 servlet 创建具有下述特性的输入表单。

- 如果提交的表单不完整则重新显示该表单。

表单向第二个 servlet 发送数据(清单 8.8)，该 servlet 检查指定的请求参数是否缺失，然后将参数值存储在 cookie 中。如果没有参数缺失，第二个 servlet 显示参数的值。如果缺失了某个参数，第二个 servlet 将用户重定向到最初的 servlet，以便重新显示表单。最初的 servlet 通过从 cookie 中提取之前输入的值，维护这些数据。

- 表单记录之前输入的数据。

表单的字段由用户在最近的请求中输入的值来填充。

图 8.11 到图 8.13 给出一些典型的结果。

清单 8.7 RegistrationForm.java

```
package coreservlets;

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

/** Servlet that displays an HTML form to collect user's
 * first name, last name, and email address. Uses cookies
 * to determine the initial values of each of those
 * form fields.
 */

public class RegistrationForm extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String actionURL =
            "/servlet/coreservlets.RegistrationServlet";
        String firstName =
            CookieUtilities.getCookieValue(request, "firstName", "");
        String lastName =
            CookieUtilities.getCookieValue(request, "lastName", "");
        String emailAddress =
            CookieUtilities.getCookieValue(request, "emailAddress",
                                           "");
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        String title = "Please Register";
        out.println
            (docType +
             "<HTML>\n" +
             "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
             "<BODY BGCOLOR=#FDF5E6>\n" +
             "<CENTER>\n" +
             "<H1>" + title + "</H1>\n" +
             "<FORM ACTION=\"" + actionURL + "\">\n" +
```

```
"First Name:\n" +
" <INPUT TYPE=\"TEXT\" NAME=\"firstName\" \" +
" VALUE=\"\" + firstName + "\"><BR>\n" +
"Last Name:\n" +
" <INPUT TYPE=\"TEXT\" NAME=\"lastName\" \" +
" VALUE=\"\" + lastName + "\"><BR>\n" +
"Email Address: \n" +
" <INPUT TYPE=\"TEXT\" NAME=\"emailAddress\" \" +
" VALUE=\"\" + emailAddress + "\"><P>\n" +
"<INPUT TYPE=\"SUBMIT\" VALUE=\"Register\">\n" +
"</FORM></CENTER></BODY></HTML>");
```

清单8.8 RegistrationServlet.java

```
package coreservlets;

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

/** Servlet that processes a registration form containing
 * a user's first name, last name, and email address.
 * If all the values are present, the servlet displays the
 * values. If any of the values are missing, the input
 * form is redisplayed. Either way, the values are put
 * into cookies so that the input form can use the
 * previous values.
*/
public class RegistrationServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        boolean isMissingValue = false;
        String firstName = request.getParameter("firstName");
        if (isMissing(firstName)) {
            firstName = "Missing first name";
            isMissingValue = true;
        }
        String lastName = request.getParameter("lastName");
        if (isMissing(lastName)) {
            lastName = "Missing last name";
            isMissingValue = true;
        }
        String emailAddress = request.getParameter("emailAddress");
        if (isMissing(emailAddress)) {
            emailAddress = "Missing email address";
            isMissingValue = true;
        }
        Cookie c1 = new LongLivedCookie("firstName", firstName);
        response.addCookie(c1);
        Cookie c2 = new LongLivedCookie("lastName", lastName);
        response.addCookie(c2);
        Cookie c3 = new LongLivedCookie("emailAddress",
                                       emailAddress);
        response.addCookie(c3);
    }
}
```

```
response.addCookie(c3);
String formAddress =
    "/servlet/coreservlets.RegistrationForm";
if (isMissingValue) {
    response.sendRedirect(formAddress);
} else {
    PrintWriter out = response.getWriter();
    String docType =
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
        "Transitional//EN\\\">\\n";
    String title = "Thanks for Registering";
    out.println
        (docType +
        "<HTML>\\n" +
        "<HEAD><TITLE>" + title + "</TITLE></HEAD>\\n" +
        "<BODY BGCOLOR=\"#FDF5E6\\\">\\n" +
        "<CENTER>\\n" +
        "<H1 ALIGN=\"\" + title + "</H1>\\n" +
        "<UL>\\n" +
        " <LI><B>First Name</B>: " +
        firstName + "\\n" +
        " <LI><B>Last Name</B>: " +
        lastName + "\\n" +
        " <LI><B>Email address</B>: " +
        emailAddress + "\\n" +
        "</UL>\\n" +
        "</CENTER></BODY></HTML>");
}
}

/** Determines if value is null or empty. */
private boolean isMissing(String param) {
    return((param == null) ||
           (param.trim().equals(""))));
}
```

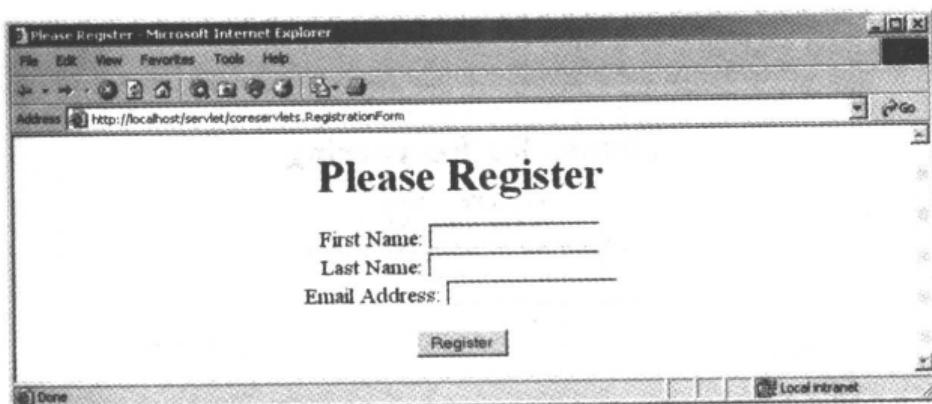


图 8.11 RegistrationForm servlet 的最初结果

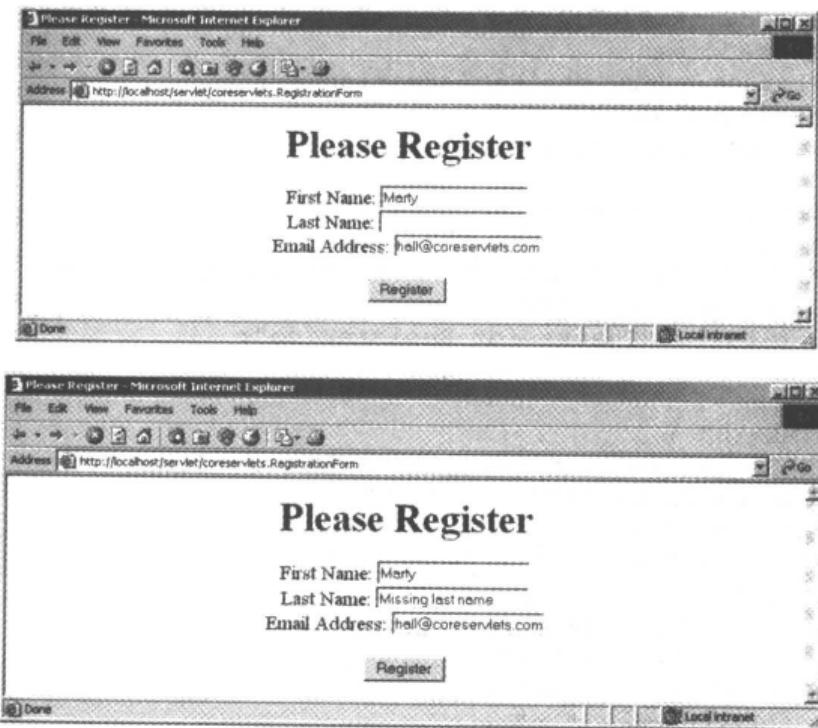


图 8.12 当输入表单没有填写完全时(上部), RegistrationServlet 将用户重定向到 Registration Form(下部)。RegistrationForm 使用 cookie 确定表单中已经填写的字段值

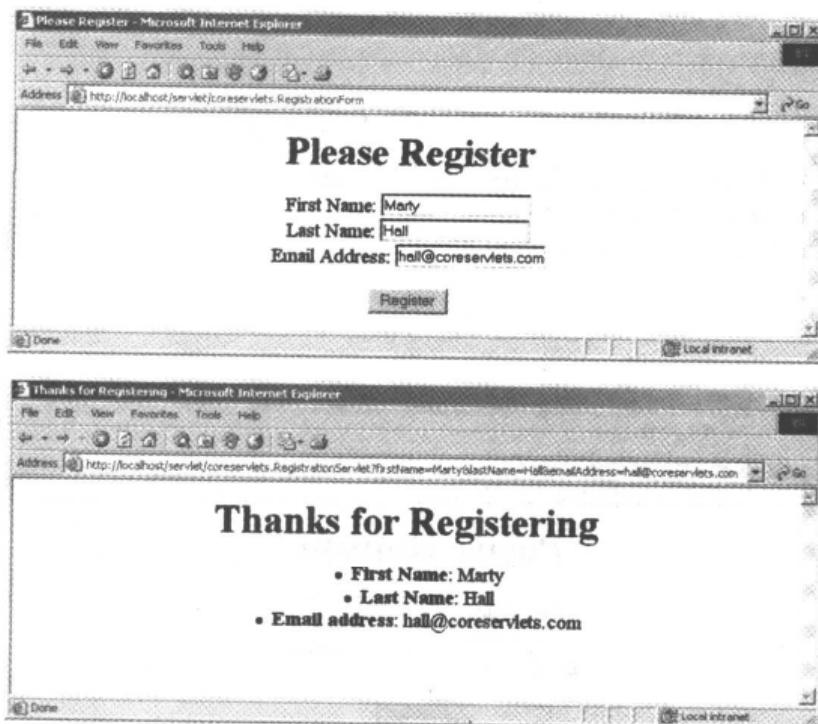


图 8.13 当输入表单填写完全时(上部), RegistrationServlet(下部)则显示请求参数的值。此处显示的输入表单(上部)还代表用户之后再次访问输入表单时表单的外观: 表单由最近使用过的值填充

第9章 会话跟踪

本章的主题：

- 从零开始实现会话跟踪(session tracking)
- 使用基本的会话跟踪
- 了解会话跟踪 API
- 区分服务器会话和浏览器会话
- URL 编码
- 不可变(immutable)对象和可变(mutable)对象的存储
- 跟踪用户的访问计数
- 累计用户购物
- 购物车的实现
- 构建在线商店

本章介绍servlet会话跟踪API，它可以在访问者浏览您的网站时跟踪具体用户的数据。

9.1 会话跟踪的需求

HTTP是“无状态”协议：客户程序每次读取Web页面，都打开到Web服务器的单独的连接，并且，服务器也不自动维护客户的上下文信息。即使那些支持持续性(继续使用)HTTP连接的服务器，尽管多个客户请求连续发生且间隔很短时它们会保持socket打开，但是，它们也没有提供维护上下文信息的内建支持。上下文的缺失引起许多困难。例如，在线商店的客户向他们的购物车中加入商品时，服务器如何知道购物车中已有何种物品呢？类似地，在客户决定结账时，服务器如何能确定之前创建的购物车中哪个属于此客户呢？这些问题虽然看起来十分简单，但是由于HTTP的不足，解答它们却异常复杂困难。

对于这个问题，存在3种典型的解决方案：cookie，URL重写和隐藏的表单域。下面的小节简短地概括了：如果自己实现会话跟踪(不使用内建的会话跟踪API)，这3种方式各需要些什么。

9.1.1 cookie

我们可以使用cookie存储购物会话的ID；在后续连接中，取出当前的会话ID，并使用这个ID从服务器上的查找表(lookup table)中提取出会话的相关信息。因此，实际上要用到两个表：将会话ID与用户关联起来的表和存储用户具体数据的表。例如，在最初的请求中，servlet可能会做类似下面这些工作：

```
String sessionID = makeUniqueString();
HashMap sessionInfo = new HashMap();
HashMap globalTable = findTableStoringSessions();
globalTable.put(sessionID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessionID);
```

```
sessionCookie.setPath("/");
response.addCookie(sessionCookie);
```

然后，在接下来的请求中，服务器可以使用globalTable散列表，将来自于JSESSIONID cookie中的会话ID与sessionInfo散列表中的用户具体数据关联起来。

以这种方式使用cookie是一种绝佳的解决方案，也是在处理会话时最常使用的方式。但是，servlet中最好有一种高级的API来处理所有这些任务，以及下面这些冗长乏味的任务。

- 从众多的其他 cookie 中(毕竟可能会存在许多 cookie)，提取出存储会话标识符的 cookie；
- 确定空闲会话什么时候过期，并回收它们；
- 将散列表与每个请求关联起来；
- 生成惟一的会话标识符。

9.1.2 URL 重写

采用这种方式时，客户程序在每个URL的尾部添加一些额外数据。这些数据标识当前的会话，服务器将这个标识符与它存储的用户相关数据关联起来。以 `http://host/path/file.html;jsessionid=a1234` 为例，`jsessionid=a1234` 作为会话的标识符附加在 URL 的尾部，因此 `a1234` 就是惟一标识与用户相关联的数据表的 ID。

URL 重写是比较不错的会话跟踪解决方案，即使浏览器不支持 cookie 或在用户禁用 cookie 的情况下，这种方案也能够工作。但是，如果您自己实现会话跟踪，URL 重写具有 cookie 所具有的同样缺点，也就是说，服务器端程序要做许多简单但是冗长乏味的处理任务。即使有高层的 API 可以为您处理大部分的细节，您仍须十分小心：每个引用您的站点的 URL，以及那些返回给用户的 URL(即使通过间接手段，比如服务器重定向中的 Location 字段)都要添加额外的信息。这种限制意味着，在您的站点上不能有任何静态 HTML 页面(至少静态页面中不能有任何链接到站点动态页面的链接)。因此，每个页面都必须使用 servlet 或 JSP 动态生成。即使所有的页面都动态生成，如果用户离开了会话并通过书签或链接再次回来，会话的信息也会丢失，因为存储下来的链接含有错误的标识信息。

9.2.3 隐藏的表单域

如第 19 章所述，HTML 表单中可以含有如下的条目：

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="a1234">
```

这个条目的意思是：在提交表单时，要将指定的名称和值自动包括在 GET 或 POST 数据中。这个隐藏域可以用来存储有关会话的信息，但它的主要缺点是：仅当每个页面都是由表单提交而动态生成时，才能使用这种方法。单击常规的(`<A HREF...>`)超文本链接并不产生表单提交，因此隐藏的表单域不能支持通常的会话跟踪，只能用于一系列特定的操作中，比如在线商店的结账过程。

9.2.4 servlet 中的会话跟踪

servlet 提供一种出色的会话跟踪解决方案：HttpSession API。这个高层接口构筑在 cookie

或URL重写之上。所有的服务器都需要支持使用cookie的会话跟踪，大多数服务器提供一项设置，可以全局地切换到URL重写。

不管采用哪种方式，servlet的开发设计人员都不需要为许多具体的实现细节操心，也不需要显式地操作cookie或附加到URL上的信息；他们自动获得一个区域，可以方便地存储与每个会话相关联的任意对象。

9.2 会话跟踪基础

在servlet中，会话的使用比较简单直接，它涉及4个基本的步骤。下面是简要的汇总；相应的细节随后给出。

(1) 访问与当前请求相关联的会话对象。

调用request.getSession获取HttpSession对象，该对象是一个简单的散列表，用来存储用户的相关数据。

(2) 查找与会话相关联的信息。

调用HttpSession对象的getAttribute，将返回值转换成恰当的类型，并检查结果是否为null。

(3) 存储会话中的信息。

使用setAttribute，设置需要存储的值以及相应的键。

(4) 废弃会话数据。

调用removeAttribute废弃指定的值。调用invalidate废弃整个会话。调用logout使客户退出Web服务器，并作废与该用户相关联的所有会话。

9.2.1 访问与当前请求相关联的会话对象

会话对象的类型是HttpSession，但它们基本上只是能够存储任意用户对象(每个都与一个键相关联)的散列表。通过调用HttpServletRequest的getSession方法访问HttpSession对象，如下所示：

```
HttpSession session = request.getSession();
```

在后台，系统从cookie或附加URL数据中提取出用户ID，之后以该ID为键，访问之前创建的HttpSession对象组成的表。但是，这些动作对程序员都是完全透明的：您只需调用getSession。如果在输入cookie或附加URL信息中找不到会话ID，系统则会创建一个新的空会话。同时，如果使用cookie(默认情况)，系统还会创建一个名为JSESSIONID的输出cookie，在其中存入唯一的值表示会话ID。因此，虽然您调用请求的getSession方法，但这种调用能够影响到后面的响应。因此，仅当设置HTTP响应报头合法的时候，才可以调用request.getSession，也就是在任何文档内容发送到(即刷新或交付)客户程序之前。

核心方法

只能在发送任何文档内容到客户程序之前调用 request.getSession。

现在，如果计划在不考虑已有数据的情况下，将数据加入到会话中，那么getSession()(或

等同的`getSession(true)`)是合适的方法调用，因为如果不存在会话，它会创建一个新的会话。然而，假使您只想打印出那些在会话中已经存在的信息，例如在电子商务站点“查看购物车”，那么在不存在会话时创建新会话则是一种浪费。此时，可以使用`getSession(false)`，如果不存在对应当前客户的会话时，它会返回`false`。下面是一个具体的例子。

```
HttpSession session = request.getSession(false);
if (session == null) {
    printMessageSayingCartIsEmpty();
} else {
    extractCartAndPrintContents(session);
}
```

9.2.2 访问与会话相关联的信息

`HttpSession`对象存在于服务器端；它们不在网络上来回传送；它们只是通过某种后台运作机制，比如cookie或URL重写，自动与客户关联在一起。这些会话对象拥有内建的数据结构(散列表)，其中可以存储任意数量的键和与键相关联的值。查找前些时候存储的值使用`session.getAttribute("key")`。这个方法的返回类型是`Object`，因此，我们可以对它执行类型转换，将它转换成会话中与这个属性名相关联的任何更为具体的数据类型。如果不存在这个属性，这个方法的返回值为`null`，因此，在调用与会话相关联的对象的方法之前，一定要检查它是否为`null`。

下面是一个具有代表性的例子。

```
HttpSession session = request.getSession();
SomeClass value =
    (SomeClass) session.getAttribute("someIdentifier");
if (value == null) { // No Such object already in session
    value = new SomeClass(...);
    session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

大多数情况下，我们都已预先知道具体的属性名，希望找出与该属性名相关联的值(如果存在的话)。但是，我们也可以调用`getAttributeNames`得出给定会话中的所有属性名，这个方法返回`Enumeration`。

9.2.3 将信息与会话关联起来

如前面的小节所述，`getAttribute`可以用来读取与会话相关联的信息。而如果要指定信息，则需使用`setAttribute`。如果希望在这些值存储到会话中之前执行边界效应(side effect)，只需让希望与会话关联起来的对象实现`HttpSessionBindingListener`接口。这样，每次调用`setAttribute`方法设置它们中间的任何一个对象时，它的`valueBound`方法都会被紧随其后调用。

但要注意，`setAttribute`会替换掉任何之前设定的值；如果想要在不提供任何替代的情况下移除某个值，则应使用`removeAttribute`。这个方法会触发所有实现了`HttpSessionBindingListener`接口的值的`valueUnbound`方法。

下面是一个向会话中添加信息的例子。可以用两种方式添加信息：添加一项新的会话

属性(如示例中粗体显示的行)或扩充已经存在于会话中的对象(如示例中最后一行)。9.7节和9.8节给出的例子(比较不可变和可变对象作为会话属性的应用)更为详实地展示出这两种方式之间的差异。

```
HttpSession session = request.getSession();
SomeClass value =
    (SomeClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
    value = new SomeClass(...);
    session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

通常地,会话属性的类型只要是Object就可以了(即它们可以是除null或基本类型,如int, double或boolean,之外的任何东西)。但是,某些应用服务器支持分布式Web应用,其中应用程序为物理服务器的集群所共享。这种情况要求会话跟踪依旧能够工作,因此,系统需要能够将会话对象在机器间转移。因此,如果您的程序运行在这种环境中,并将Web应用标记为可分布式执行,则必须满足额外的需求:会话的属性要实现Serializable接口。

9.2.4 废弃会话数据

使用完用户的会话数据之后,我们有3种选择:

- 只移除自己编写的 servlet 创建的数据。

我们可以调用 removeAttribute("key"), 废弃与指定键关联的值。这是最常见的方法。

- 删 除 整 个 会 话 (在 当 前 Web 应 用 中) 。

我们可以调用 invalidate, 将整个会话废弃掉。但要记住,这样做会丢失该用户所有的会话数据,而非仅仅由我们的 servlet 或 JSP 页面创建的会话数据。因此,Web 应用中所有的 servlet 和 JSP 页面都要适应 invalidate 有可能会被调用这种情况。

- 将 用户 从 系 统 中 注 销 并 删 除 所 有 属 于 他 (或 她) 的 会 话 。

最后,在支持 servlet 2.4 和 JSP 2.0 的服务器中,我们可以调用 logout, 将客户从 Web 服务器中注销,同时废弃所有与该用户相关联的会话(每个 Web 应用至多一个)。同样,由于这个动作会影响到我们自己的 servlet 以外的其他 servlet,因此,一定要与网站的其他开发者协调 logout 命令的使用。

9.3 会话跟踪 API

尽管会话的属性(即用户数据)是会话信息中我们最关注的部分,但是,其他信息有时也同样重要。下面对 HttpSession 类中的方法进行汇总。

(1) public Object getAttribute(String name)

这个方法从会话对象中提取出之前存储的值。如果没有值与给定的名称相关联,它返回 null。

(2) **public Enumeration getAttributeNames()**

这个方法返回会话中所有属性的名称。

(3) **public void setAttribute(String name, Object value)**

这个方法将值和名称关联起来。如果提供给 `setAttribute` 的对象实现了 `HttpSessionBindingListener` 接口，在它存储到会话中之后，它的 `valueBound` 方法会被调用。类似地，如果之前的值实现了 `HttpSessionBindingListener`，它的 `valueUnbound` 方法会被调用。

(4) **public void removeAttribute(String name)**

这个方法移除与指定名称关联的任何值。如果要移除的值实现了 `HttpSessionBindingListener` 接口，它的 `valueUnbound` 方法会被调用。

(5) **public void invalidate()**

这个方法将会话作废，并释放所有与之相关联的对象。使用这个方法时要小心：要牢记会话是与用户(即客户程序)相关联的，而不是单个的 servlet 或 JSP 页面。因此，如果作废一个会话，有可能会破坏掉其他 servlet 或 JSP 页面正在使用的数据。

(6) **public void logout()**

这个方法将客户从 Web 服务器中注销，并将与该客户相关联的所有会话作废。`logout` 的影响范围和身份验证的影响范围相同。例如，如果服务器实现了单一登录，那么调用 `logout` 则会将客户从服务器上的所有 Web 应用中注销，并且将与该客户相关联的所有会话(每个 Web 应用至多一个)作废。详细信息参见本书第二卷有关 Web 应用安全的章节。

(7) **public String getId()**

这个方法返回每个会话所对应的唯一标识符。这对于调试或记录，或少数情况下用编程的手段将值从内存移到数据库中(尽管某些 J2EE 服务器可以自动完成这项任务)，都很有用。

(8) **public boolean isNew()**

如果会话尚未和客户程序(浏览器)发生任何联系，则这个方法返回 `true`，这一般是因为会话是新创建的，不是由输入的客户请求所引起。对于预先存在的会话，该方法返回 `false`。提及这个方法的主要原因是让您尽量避免使用它：`isNew` 的用处可能远没有它乍看起来那么大。许多初级开发人员试图使用 `isNew` 来确定用户之前是否到过他们的 servlet(在会话的超时期之内)，他们编写如下的代码：

```
 HttpSession session = request.getSession();
 if (session.isNew()) {
    doStuffForNewbies();
 } else {
    doStuffForReturnVisitors(); // Wrong!
 }
```

这是错误的！的确，如果 `isNew` 返回 `true`，那么我们可以说，这应该是用户第一次到访(至少是在会话的超时期之内)。但如果 `isNew` 返回 `false`，只不过是说明他们之前曾经访问过该 Web 应用，并不代表他们曾访问过我们的 servlet 或 JSP 页面。

(9) public long getCreationTime()

这个方法返回会话首次构建的时间，格式为自 1970 年 1 月 1 日午夜(GMT)以来的毫秒数。如果要获取用于打印的值，可以将该值传递给 Date 的构造函数或 GregorianCalendar 的 setTimeInMillis 方法。

(10) public long getLastAccessedTime()

这个方法返回会话最后被客户程序访问的时间，格式为自 1970 年 1 月 1 日午夜(GMT)以来的毫秒数。

(11) public int getMaxInactiveInterval()**public void setMaxInactiveInterval(int seconds)**

这些方法读取或设置在没有访问的情况下，会话在被自动废弃之前应该保持多长时间，以秒为单位。负值表示会话从不超时。要注意，超时由服务器来维护，它不同于 cookie 的失效日期。首先，会话一般基于驻留内存的 cookie，不是持续性 cookie，因而也就没有截止日期。即使截取到 JSESSIONID cookie，并为它设定一个失效日期发送出去，浏览器会话和服务器会话也会截然不同。详细的区别参见下一节。

9.4 浏览器会话与服务器会话

默认地，会话跟踪(session-tracking)基于存储在浏览器内存中(而非写到磁盘上)的 cookie。因此，除非 servlet 显式地读取输入的 JSESSIONID cookie，设置最大时效和路径并将它发送回去，否则退出浏览器会造成会话中断：客户程序将不能再访问会话。但问题是服务器不知道浏览器已经关闭，因而服务器需要将会话维护在内存中，直到它处于非活动状态超过设定的间隔为止。

以到沃尔玛商店购物为例。您四处浏览，并将一些商品放在购物车中，之后，在您查看其他商品时会将购物车放在过道的一端。一个职员走上来，看到了购物车。他可以将其中的商品重新摆上货架吗？不能——您可能依旧在购物，并会很快回来寻找购物车。如果您意识到自己丢失了钱包，只好开着车回家去，这种情况下，应该怎么办呢？现在职员可以将您的购物车中的商品重新上架吗？同样不能——职员大概不会知道您已经离开了商店。因此，该职员应该怎么办呢？他可以密切注视着该购物车，如果在特定的一段时间内没有人去动它，那么他可以断定它被离弃，可以将商品拿出来。惟一的例外是，如果您将购物车交给他，并且告诉说“对不起，我将钱包忘在家里了，因此我必须离开。”

servlet 世界中类似的情形是：服务器试图确定是否可以丢弃客户的 HttpSession 对象。仅仅是由于客户当前并未使用该会话并不意味着服务器可以丢弃它。也许客户将很快回来(提交一个新请求)？如果客户退出浏览器，从而引起浏览器会话级别的 cookie 丢失，该会话也就事实上被中断。但是，如同您坐上汽车离开沃尔玛的情况一样，服务器并不知道客户退出了浏览器。因而，服务器依旧必须等一段时间，以确定会话是否被放弃。当客户访问的间隔时间超过 getMaxInactiveInterval 指定的时间之后，会话自动变为不活动状态。发生这种情况后，存储在 HttpSession 对象中的对象被移除(unbound，解除绑定)。然后，如果这些对象实现了 HttpSessionBindingListener 接口，它们会自动得到通知。“服务器会等待，

直至会话超时为止”这一规则的例外是 `invalidate` 或 `logout` 得到调用。这类似于您明确地告诉沃尔玛的职员，你要离开，因而服务器可以立即移除会话中的所有数据项，并销毁该会话对象。

9.5 对发往客户的 URL 进行编码

默认地，servlet 容器(引擎)使用 cookie 作为会话跟踪的底层机制。但假定您对服务器进行了重新配置，使之使用 URL 重写(URL rewrite)，代码应该做何更改呢？

好消息是：核心的会话跟踪代码不需做任何改动。

坏消息是：其他大部分代码都需要进行更改。特别地，如果您的页面中含有指向自身所在站点的链接时，必须显式地将会话数据添加到 URL。servlet API 提供将这项信息添加到任意指定 URL 的方法。问题是您必须调用这些方法：如果想让系统检查所有 servlet 和 JSP 页面的输出，找出哪些部分包含指向您自身站点的超链接并修改这些 URL，这在技术上是不可行的。您必须告诉它要修改哪个 URL。这项需求表示：如果使用 URL 重写进行会话跟踪，那么就不能拥有任何静态 HTML 页面，或者至少不能拥有引用自身站点的静态 HTML 页面。这对于很多应用都是一项沉重的负担，但少数情况下是值得的。

警告

如果您使用 URL 重写进行会话跟踪，您的大部分或全部页面都必须动态生成。您站点上的任何静态 HTML 页面都不能含有指向自身站点动态页面的链接。

两种情况下可能会用到引用自身站点的 URL。

第一种情况是在 servlet 生成的 Web 页面中含有嵌入的 URL。这种情况下，应该将这些 URL 传递给 `HttpServletResponse` 的 `encodeURL` 方法。这个方法确定当前是否在使用 URL 重写，仅在必需时附加会话信息；否则，不做任何更改直接返回传入的 URL。

下面是一个具体的例子：

```
String originalURL = someRelativeOrAbsoluteURL;
String encodedURL = response.encodeURL(originalURL);
out.println("<A HREF=\"" + encodedURL + "\">...</A>");
```

第二种情况是在 `sendRedirect` 调用中(即放入 Location 响应报头)。这种情况下，由于要根据不同的规则确定是否需要附加会话信息，因而不能使用 `encodeURL`。幸运的是，`HttpServletResponse` 提供 `encodeRedirectURL` 方法来处理这种情况。下面是一个具体的例子：

```
String originalURL = someURL;
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

如果认为自己的 Web 应用最终有可能使用 URL 重写来替代 cookie，那么最好预先规划，对引用自身站点的 URL 进行编码。

9.6 显示客户访问计数的 servlet

清单 9.1 给出一个简单的 servlet，它显示客户会话的基本信息。在客户处于连接状态时，该 servlet 使用 `request.getSession` 获取现存的会话，或在没有会话的情况下创建新的会话。之后，该 servlet 查找类型为 Integer 的 `accessCount` 属性。如果找不到这个属性，则使用 0 作为之前的访问计数。然后，对这个值进行递增，并用 `setAttribute` 与会话关联起来。最后，该 servlet 打印出一个小型的 HTML 表格，给出与会话相关的信息。

要注意，`Integer` 是一种 `immutable`(不可修改)的数据结构：构建后就不能更改。这意味着：每个请求都必须分配新的 `Integer` 对象，之后使用 `setAttribute` 来替换老的对象。下面的片断展示出存储不可变对象时会话跟踪所采用的一般方式。

```
HttpSession session = request.getSession();
SomeImmutableClass value =
    (SomeImmutableClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
    value = new SomeImmutableClass(...);
} else {
    value = new SomeImmutableClass(calculatedFrom(value));
}
session.setAttribute("someIdentifier", value);
doSomethingWith(value);
```

这种方式与下一节(9.7 节)采用可变(可修改)数据结构的方式形成对比。在那种方式中，仅当会话中不存在这种对象时才分配对象并调用 `setAttribute`。也就是说，每次都是改变对象的内容，会话维护对同一个对象的引用。

图 9.1 和图 9.2 分别给出初次访问这个 servlet，以及页面重新载入几次之后的结果。

清单9.1 ShowSession.java

```
package coreservlets;

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

/** Servlet that uses session-tracking to keep per-client
 * access counts. Also shows other info about the session.
 */

public class ShowSession extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        HttpSession session = request.getSession();
        String heading;
        Integer accessCount =
            (Integer)session.getAttribute("accessCount");
        if (accessCount == null) {
            accessCount = new Integer(0);
```

```

heading = "Welcome, Newcomer";
} else {
    heading = "Welcome Back";
    accessCount = new Integer(accessCount.intValue() + 1);
}
// Integer is an immutable data structure. So, you
// cannot modify the old one in-place. Instead, you
// have to allocate a new one and redo setAttribute.
session.setAttribute("accessCount", accessCount);
PrintWriter out = response.getWriter();
String title = "Session Tracking Example";
String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
    "Transitional//EN\">\n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
    "<CENTER>\n" +
    "<H1>" + heading + "</H1>\n" +
    "<H2>Information on Your Session:</H2>\n" +
    "<TABLE BORDER=1>\n" +
    "<TR BGCOLOR=\"#FFAD00\">\n" +
    "  <TH>Info Type<TH>Value\n" +
    "<TR>\n" +
    "  <TD>ID\n" +
    "  <TD>" + session.getId() + "\n" +
    "<TR>\n" +
    "  <TD>Creation Time\n" +
    "  <TD>" +
    new Date(session.getCreationTime()) + "\n" +
    "<TR>\n" +
    "  <TD>Time of Last Access\n" +
    "  <TD>" +
    new Date(session.getLastAccessedTime()) + "\n" +
    "<TR>\n" +
    "  <TD>Number of Previous Accesses\n" +
    "  <TD>" + accessCount + "\n" +
    "</TABLE>\n" +
    "</CENTER></BODY></HTML>");
```

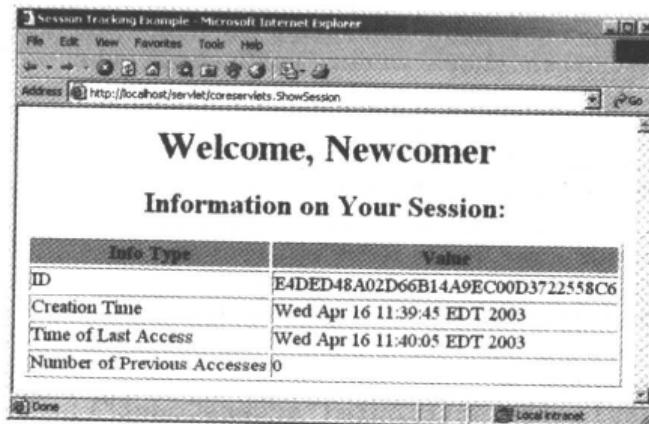


图 9.1 客户对 ShowSession servlet 的首次访问



图 9.2 对 ShowSession servlet 的第 12 次访问。该客户的访问计数与其他客户的访问次数无关

9.7 累计用户数据的列表

前一节(9.6 节)的例子在用户的 HttpSession 对象中存储用户专有的数据。所存储的对象是一种不可变的数据结构：不能被修改。因而，针对每个请求都要分配新的 Integer，并用 setAttribute 将新的对象存储到会话中，覆盖前面的值。

另一种常见的方式是使用可变的数据结构，比如数组、List、Map 或含有可写字段(实例变量)的应用程序专有数据结构。通过这种方式，除非首次分配对象，否则不需要调用 setAttribute。下面是一个基本的模板：

```
HttpSession session = request.getSession();
SomeMutableClass value=
    (SomeMutableClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
    value = new SomeMutableClass(...);
}
value.updateInternalState(...);
doSomethingWith(value);
```

可变数据结构最常用于维护与用户相关联的一组数据。本节中，我们提供一个经过简化的例子，其中维护一个基本的商品列表，存储每个用户的已购置商品。下一节中(9.8 节)，我们提供一个成熟的购物车的例子。这个例子中的大部分代码用来构建自动显示所购置商品的 Web 页面，以及购物车自身。虽然这些和具体应用相关的代码稍微有些复杂，但基本的会话跟踪却十分简单。虽然简单，但更有效的方法还是不受应用程序专有代码的影响，直接查看代码所使用的基本方法。这也正是此例的目的。

清单 9.2 列出的应用程序使用简单的 ArrayList(Java 2 平台中 Vector 的替代物)来跟踪每个用户已经购置的商品。除了查找或创建会话，并将新购置的商品(newItem 请求参数的值)插入会话中之外，这个例子输出一个项目列表，显示“手推车”(即 ArrayList)中有哪些商品。要注意到，输出这个列表的代码在 ArrayList 上进行了同步。这项预防措施是值得的。但要明白，使同步成为必需的环境非常罕见。由于每个用户拥有独立的会话，竞争条件发生的唯一方式是同一用户快速地连续提交两项购买请求。尽管不太可能发生，但的确存在这种可能性，因此同步措施是值得的。

清单 9.2 ShowItems.java

```

package coreservlets;

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

/** Servlet that displays a list of items being ordered.
 * Accumulates them in an ArrayList with no attempt at
 * detecting repeated items. Used to demonstrate basic
 * session tracking.
 */
public class ShowItems extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession();
        ArrayList previousItems =
            (ArrayList)session.getAttribute("previousItems");
        if (previousItems == null) {
            previousItems = new ArrayList();
            session.setAttribute("previousItems", previousItems);
        }
        String newItem = request.getParameter("newItem");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Items Purchased";
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1>" + title + "</H1>");
        synchronized(previousItems) {
            if ((newItem != null) &&
                (!newItem.trim().equals("")))
                previousItems.add(newItem);
            if (previousItems.size() == 0) {
                out.println("<I>No items</I>");
            } else {
                out.println("<UL>");
                for(int i=0; i<previousItems.size(); i++) {
                    out.println("<LI>" + (String)previousItems.get(i));
                }
                out.println("</UL>");
            }
        }
        out.println("</BODY></HTML>");
    }
}

```

清单 9.3 给出的 HTML 表单收集 newItem 参数的值，并将它们提交给 servlet。图 9.3

展示出表单的结果；图 9.4 和图 9.5 分别给出访问订单表单之前，以及访问几次之后，servlet 的结果。

清单9.3 OrderForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>Order Form</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1>Order Form</H1>
<FORM ACTION="servlet/coreservlets.ShowItems">
    New Item to Order:
    <INPUT TYPE="TEXT" NAME="newItem" VALUE="Yacht"><P>
    <INPUT TYPE="SUBMIT" VALUE="Order and Show All Purchases">
</FORM>
</CENTER></BODY></HTML>
```

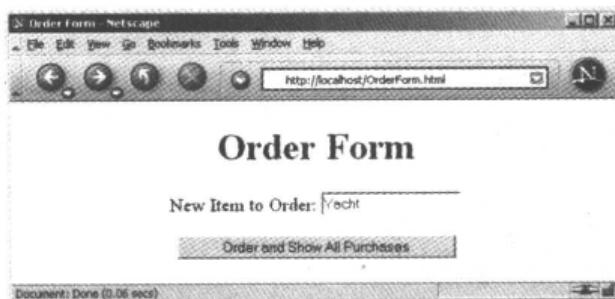


图 9.3 显示商品的 servlet 的前端



图 9.4 没有购买任何商品之前的显示商品的 servlet

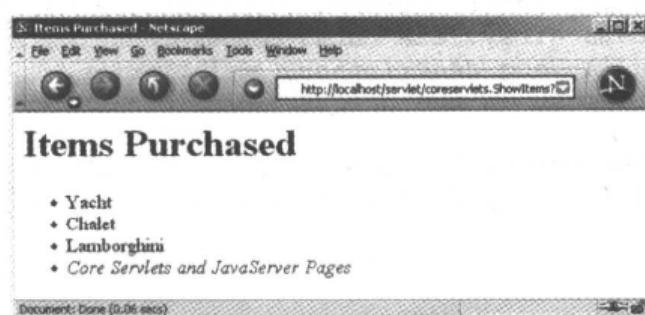


图 9.5 购买几项有价值的商品之后的显示商品的 servlet

9.8 拥有购物车和会话跟踪功能的在线商店

这一节给出一个扩充的例子，介绍如何使用会话跟踪构造在线商店。第一小节介绍如何构造显示待售商品的页面。每个展示页面的代码只是列出页面的标题和列在页面中的商品的标识符。之后，由父类中的方法根据存储在分类表中的物品描述自动构建实际的页面。第二小节给出处理订单的页面。它使用会话跟踪将购物车和每个用户关联起来，并允许用户修改之前选择的任何商品。第三小节提供购物车、表示单个物品和订单的数据结构、以及分类表的实现。

9.8.1 创建前端界面

清单 9.4 给出一个抽象基类，显示待售商品的 servlet 将它用作起点。它接受待售商品的标识符，在分类表中查找它们，并根据找出的描述和价格提供给用户一个订单页面。清单 9.5(结果在图 9.6 中给出)和清单 9.6(结果在图 9.7 中给出)展示出应用这个父类构造实际的页面是多么容易。

清单9.4 CatalogPage.java

```
package coreservlets;

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

/** Base class for pages showing catalog entries.
 * Servlets that extend this class must specify
 * the catalog entries that they are selling and the page
 * title <I>before</I> the servlet is ever accessed. This
 * is done by putting calls to setItems and setTitle
 * in init.
 */

public abstract class CatalogPage extends HttpServlet {
    private CatalogItem[] items;
    private String[] itemIDs;
    private String title;

    /** Given an array of item IDs, look them up in the
     * Catalog and put their corresponding CatalogItem entry
     * into the items array. The CatalogItem contains a short
     * description, a long description, and a price,
     * using the item ID as the unique key.
     * <P>
     * Servlets that extend CatalogPage <B>must</B> call
     * this method (usually from init) before the servlet
     * is accessed.
     */

    protected void setItems(String[] itemIDs) {
        this.itemIDs = itemIDs;
        items = new CatalogItem[itemIDs.length];
        for(int i=0; i<items.length; i++) {
```

```
    items[i] = Catalog.getItem(itemIDs[i]);
}
}

/** Sets the page title, which is displayed in
 * an H1 heading in resultant page.
 * <P>
 * Servlets that extend CatalogPage <B>must</B> call
 * this method (usually from init) before the servlet
 * is accessed.
*/
protected void setTitle(String title) {
    this.title = title;
}

/** First display title, then, for each catalog item,
 * put its short description in a level-two (H2) heading
 * with the price in parentheses and long description
 * below. Below each entry, put an order button
 * that submits info to the OrderPage servlet for
 * the associated catalog entry.
 * <P>
 * To see the HTML that results from this method, do
 * "View Source" on KidsBooksPage or TechBooksPage, two
 * concrete classes that extend this abstract class.
*/
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
if (items == null) {
    response.sendError(response.SC_NOT_FOUND,
                       "Missing Items.");
    return;
}
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
     \"Transitional//EN\\\">\\n";
out.println(docType +
            "<HTML>\\n" +
            "<HEAD><TITLE>" + title + "</TITLE></HEAD>\\n" +
            "<BODY BGCOLOR=\"#FDF5E6\\\">\\n" +
            "<H1 ALIGN=\"CENTER\\\">" + title + "</H1>");
CatalogItem item;
for(int i=0; i<items.length; i++) {
    out.println("<HR>");
    item = items[i];
    // Show error message if subclass lists item ID
    // that's not in the catalog.
    if (item == null) {
        out.println("<FONT COLOR=\"RED\\\">" +
                   "Unknown item ID " + itemIDs[i] +
                   "</FONT>");
    } else {
        out.println();
    }
}
```

```

String formURL =
    "/servlet/coreservlets.OrderPage";
// Pass URLs that reference own site through encodeURL.
formURL = response.encodeURL(formURL);
out.println(
    "<FORM ACTION=\"" + formURL + "\">\n" +
    "<INPUT TYPE=\"HIDDEN\" NAME=\"itemID\" " +
    " VALUE=\"" + item.getItemId() + "\">\n" +
    "<H2>" + item.getShortDescription() +
    " (" + item.getCost() + ")</H2>\n" +
    item.getLongDescription() + "\n" +
    "<P>\n<CENTER>\n" +
    "<INPUT TYPE=\"SUBMIT\" " +
    "VALUE=\"Add to Shopping Cart\">\n" +
    "</CENTER>\n<P>\n</FORM>");

}

}

out.println("<HR>\n</BODY></HTML>");
}
}
}

```

清单9.5 KidsBooksPage.java

```

package coreservlets;

/** A specialization of the CatalogPage servlet that
 * displays a page selling three famous kids-book series.
 * Orders are sent to the OrderPage servlet.
 */

public class KidsBooksPage extends CatalogPage {
    public void init() {
        String[] ids = { "lewis001", "alexander001", "rowling001" };
        setItems(ids);
        setTitle("All-Time Best Children's Fantasy Books");
    }
}

```

清单9.6 TechBooksPage.java

```

package coreservlets;

/** A specialization of the CatalogPage servlet that
 * displays a page selling two famous computer books.
 * Orders are sent to the OrderPage servlet.
 */

public class TechBooksPage extends CatalogPage {
    public void init() {
        String[] ids = { "hall001", "hall002" };
        setItems(ids);
        setTitle("All-Time Best Computer Books");
    }
}

```

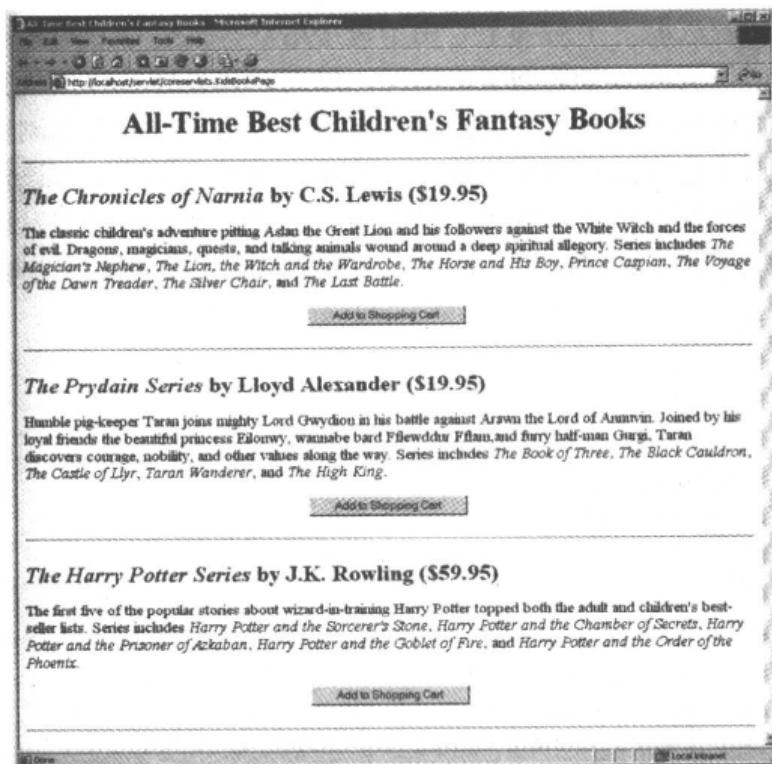


图 9.6 KidsBooksPage servlet 的结果

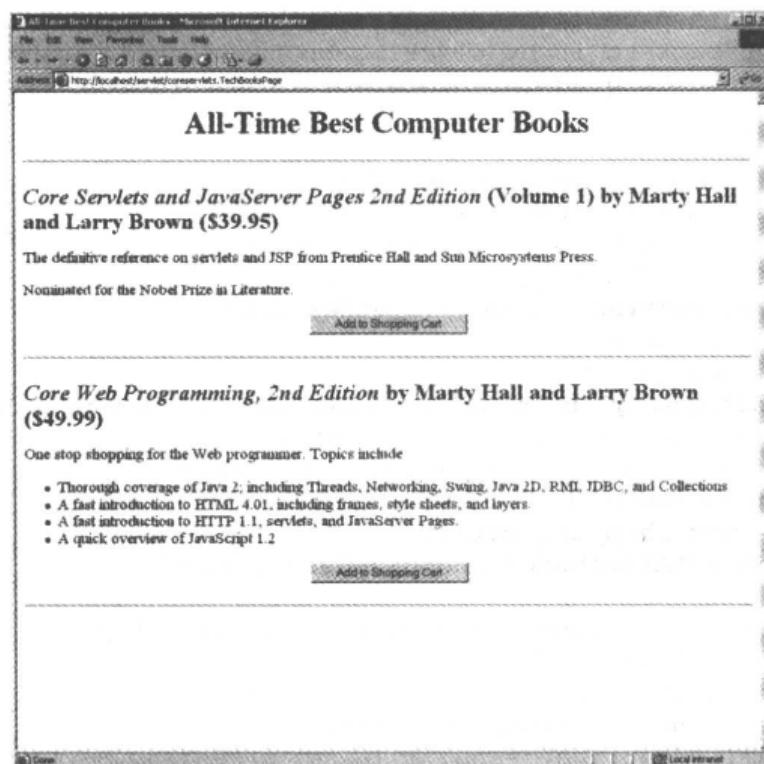


图 9.7 TechBooksPage servlet 的结果

9.8.2 订单的处理

清单 9.7 给出的 servlet 负责处理前一小节中列出的各种分类页面中发出的订单。它使

用会话跟踪为每个用户关联一个购物车。由于每个用户都拥有单独的会话，因此，不太可能发生在多个线程同时访问同一个购物车的情况。然而，如果您是位偏执狂，那么您有可能可以构想出几种有可能发生并发访问的情况，比如单个用户打开多个浏览器窗口，并快速地在多个浏览器窗口中连续地发送更新数据。因而，为了安全起见，该代码基于会话对象对访问做了同步。这种同步措施可以防止使用同一会话的其他线程并发地访问数据，同时依旧可以同步地处理来自于不同用户的请求。图 9.8 和图 9.9 给出一些典型的结果。

清单 9.7 OrderPage.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.text.*;

/** Shows all items currently in ShoppingCart. Clients
 * have their own session that keeps track of which
 * ShoppingCart is theirs. If this is their first visit
 * to the order page, a new shopping cart is created.
 * Usually, people come to this page by way of a page
 * showing catalog entries, so this page adds an additional
 * item to the shopping cart. But users can also
 * bookmark this page, access it from their history list,
 * or be sent back to it by clicking on the "Update Order"
 * button after changing the number of items ordered.
 */

public class OrderPage extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession();
        ShoppingCart cart;
        synchronized(session) {
            cart = (ShoppingCart)session.getAttribute("shoppingCart");
            // New visitors get a fresh shopping cart.
            // Previous visitors keep using their existing cart.
            if (cart == null) {
                cart = new ShoppingCart();
                session.setAttribute("shoppingCart", cart);
            }
            String itemID = request.getParameter("itemID");
            if (itemID != null) {
                String numItemsString =
                    request.getParameter("numItems");
                if (numItemsString == null) {
                    // If request specified an ID but no number,
                    // then customers came here via an "Add Item to Cart"
                    // button on a catalog page.
                    cart.addItem(itemID);
                } else {
                    // If request specified an ID and number, then
                    // customers came here via an "Update Order" button
                    // after changing the number of items in order.
                }
            }
        }
    }
}
```

```
// Note that specifying a number of 0 results
// in item being deleted from cart.
int numItems;
try {
    numItems = Integer.parseInt(numItemsString);
} catch(NumberFormatException nfe) {
    numItems = 1;
}
cart.setNumOrdered(itemID, numItems);
}

}

// Whether or not the customer changed the order, show
// order status.
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "Status of Your Order";
String docType =
    "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
    "<BODY BGCOLOR=#FDF5E6>\n" +
    "<H1 ALIGN=\\\"CENTER\\\">" + title + "</H1>");
synchronized(session) {
    List itemsOrdered = cart.getItemsOrdered();
    if (itemsOrdered.size() == 0) {
        out.println("<H2><I>No items in your cart...</I></H2>");
    } else {
        // If there is at least one item in cart, show table
        // of items ordered.
        out.println
            ("<TABLE BORDER=1 ALIGN=\\\"CENTER\\\">\n" +
            "<TR BGCOLOR=#FFAD00>\n" +
            " <TH>Item ID<TH>Description\n" +
            " <TH>Unit Cost<TH>Number<TH>Total Cost");
        ItemOrder order;
        // Rounds to two decimal places, inserts dollar
        // sign (or other currency symbol), etc., as
        // appropriate in current Locale.
        NumberFormat formatter =
            NumberFormat.getCurrencyInstance();
        // For each entry in shopping cart, make
        // table row showing ID, description, per-item
        // cost, number ordered, and total cost.
        // Put number ordered in textfield that user
        // can change, with "Update Order" button next
        // to it, which resubmits to this same page
        // but specifying a different number of items.
        for(int i=0; i<itemsOrdered.size(); i++) {
            order = (ItemOrder)itemsOrdered.get(i);
            out.println
                ("<TR>\n" +
                " <TD>" + order.getItemId() + "\n" +
                " <TD>" + order.getShortDescription() + "\n" +
                " <TD>" +
                formatter.format(order.getUnitCost()) + "\n" +
                " <TD>");

    }
}
```

```

    " <TD>" +
    "<FORM>\n" + // Submit to current URL
    "<INPUT TYPE=\"HIDDEN\" NAME=\"itemID\"\n" +
    "     VALUE=\"" + order.getItemId() + "\">\n" +
    "<INPUT TYPE=\"TEXT\" NAME=\"numItems\"\n" +
    "     SIZE=3 VALUE=\"" +
    order.getNumItems() + "\">\n" +
    "<SMALL>\n" +
    "<INPUT TYPE=\"SUBMIT\"\n" +
    "     VALUE=\"Update Order\">\n" +
    "</SMALL>\n" +
    "</FORM>\n" +
    " <TD>" +
    formatter.format(order.getTotalCost()));
}

String checkoutURL =
    response.encodeURL("../Checkout.html");
// "Proceed to Checkout" button below table
out.println(
    "</TABLE>\n" +
    "<FORM ACTION=\"\" + checkoutURL + "\">\n" +
    "<BIG><CENTER>\n" +
    "<INPUT TYPE=\"SUBMIT\"\n" +
    "     VALUE=\"Proceed to Checkout\">\n" +
    "</CENTER></BIG></FORM>");
}

out.println("</BODY></HTML>");
}
}

```

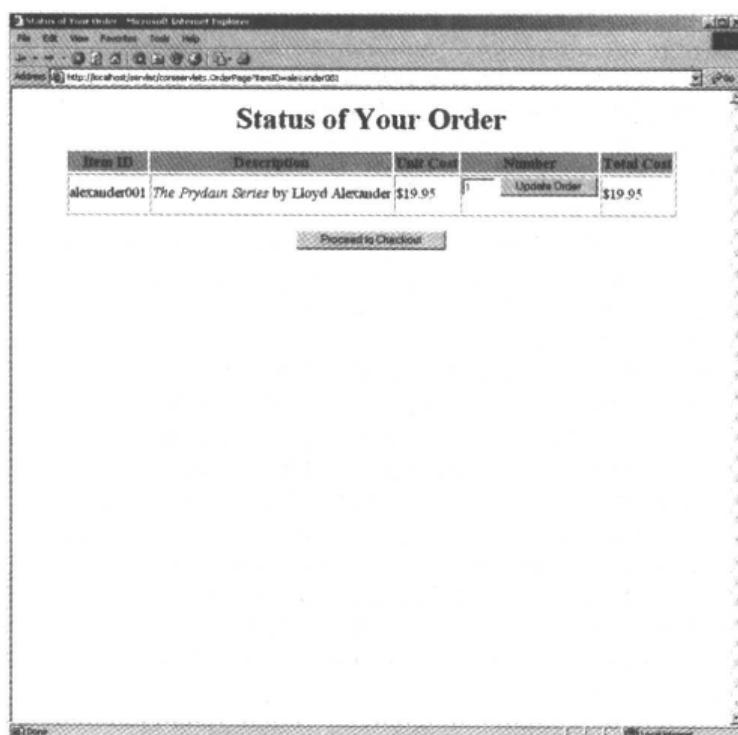


图 9.8 用户在 KidsBooksPage 中点击 “Add to Shopping Cart” 之后，OrderPage servlet 的结果

Item ID	Description	Unit Price	Quantity	Total
alexander001	<i>The Prydain Series</i> by Lloyd Alexander	\$19.95	4 <input type="button" value="Update Order"/>	\$79.80
rowling001	<i>The Harry Potter Series</i> by J.K. Rowling	\$39.95	1 <input type="button" value="Update Order"/>	\$39.95
lewis001	<i>The Chronicles of Narnia</i> by C.S. Lewis	\$19.95	1 <input type="button" value="Update Order"/>	\$19.95
hall001	<i>Core Servlets and JavaServer Pages 2nd Edition (Volume 1)</i> by Marty Hall and Larry Brown	\$39.95	52 <input type="button" value="Update Order"/>	\$2,077.40
hall002	<i>Core Web Programming, 2nd Edition</i> by Marty Hall and Larry Brown	\$49.99	23 <input type="button" value="Update Order"/>	\$1,149.77

图 9.9 对订单进行几次添加和更改之后，OrderPage servlet 的结果

清单9.8 Checkout.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>Checking Out</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Checking Out</H1>
We are sorry, but our electronic credit-card-processing
system is currently out of order. Please send a check
to:
<PRE>
    Marty Hall
    coreservlets.com, Inc.
    6 Meadowsweet Ct., Suite B1
    Reisterstown, MD 21136-6020
    410-429-5535
    hall@coreservlets.com
</PRE>
Since we have not yet calculated shipping charges, please
sign the check but do not fill in the amount. We will
generously do that for you.
</BODY></HTML>
```

9.8.3 背景：购物车和分类物品的实现

清单 9.9 列出购物车的实现。它只是简单地维护订单的 List，并提供添加和更新这些订单的方法。清单 9.10 给出处理单个分类商品的代码，清单 9.11 给出代表特定商品状态的类，清单 9.12 给出分类表的实现。

清单9.9 ShoppingCart.java

```
ItemOrder order;
for(int i=0; i<itemsOrdered.size(); i++) {
    order = (ItemOrder)itemsOrdered.get(i);
    if (order.getItemId().equals(itemID)) {
        if (numOrdered <= 0) {
            itemsOrdered.remove(i);
        } else {
            order.setNumItems(numOrdered);
        }
        return;
    }
}
ItemOrder newOrder =
    new ItemOrder(Catalog.getItem(itemID));
itemsOrdered.add(newOrder);
}
```

清单9.10 CatalogItem.java

```
package coreservlets;

/** Describes a catalog item for on-line store. The itemID
 * uniquely identifies the item, the short description
 * gives brief info like the book title and author,
 * the long description describes the item in a couple
 * of sentences, and the cost gives the current per-item price.
 * Both the short and long descriptions can contain HTML
 * markup.
 */

public class CatalogItem {
    private String itemID;
    private String shortDescription;
    private String longDescription;
    private double cost;

    public CatalogItem(String itemID, String shortDescription,
                       String longDescription, double cost) {
        setItemID(itemID);
        setShortDescription(shortDescription);
        setLongDescription(longDescription);
        setCost(cost);
    }

    public String getItemID() {
        return(itemID);
    }

    protected void setItemID(String itemID) {
        this.itemID = itemID;
    }

    public String getShortDescription() {
        return(shortDescription);
    }
```

```
protected void setShortDescription(String shortDescription) {
    this.shortDescription = shortDescription;
}

public String getLongDescription() {
    return(longDescription);
}

protected void setLongDescription(String longDescription) {
    this.longDescription = longDescription;
}

public double getCost() {
    return(cost);
}

protected void setCost(double cost) {
    this.cost = cost;
}
}
```

清单9.11 ItemOrder.java

```
package coreservlets;

/** Associates a catalog Item with a specific order by
 * keeping track of the number ordered and the total price.
 * Also provides some convenience methods to get at the
 * CatalogItem data without extracting the CatalogItem
 * separately.
 */

public class ItemOrder {
    private CatalogItem item;
    private int numItems;

    public ItemOrder(CatalogItem item) {
        setItem(item);
        setNumItems(1);
    }

    public CatalogItem getItem() {
        return(item);
    }

    protected void setItem(CatalogItem item) {
        this.item = item;
    }

    public String getItemID() {
        return(getItem().getItemID());
    }

    public String getShortDescription() {
        return(getItem().getShortDescription());
    }

    public String getLongDescription() {
```

```

        return(getItem().getLongDescription());
    }

    public double getUnitCost() {
        return(getItem().getCost());
    }

    public int getNumItems() {
        return(numItems);
    }

    public void setNumItems(int n) {
        this.numItems = n;
    }

    public void incrementNumItems() {
        setNumItems(getNumItems() + 1);
    }

    public void cancelOrder() {
        setNumItems(0);
    }

    public double getTotalCost() {
        return(getNumItems() * getUnitCost());
    }
}

```

清单9.12 Catalog.java

```

package coreservlets;

/** A catalog that lists the items available in inventory.
 */

public class Catalog {
    // This would come from a database in real life.
    // We use a static table for ease of testing and deployment.
    // See JDBC chapters for info on using databases in
    // servlets and JSP pages.
    private static CatalogItem[] items =
        { new CatalogItem
            ("hall001",
                "<I>Core Servlets and JavaServer Pages " +
                "2nd Edition</I> (Volume 1)" +
                " by Marty Hall and Larry Brown",
                "The definitive reference on servlets " +
                "and JSP from Prentice Hall and \n" +
                "Sun Microsystems Press.<P>Nominated for " +
                "the Nobel Prize in Literature.",
                39.95),
            new CatalogItem
            ("hall002",
                "<I>Core Web Programming, 2nd Edition</I> " +
                "by Marty Hall and Larry Brown",
                "One stop shopping for the Web programmer. " +
                "Topics include \n" +
                "<UL><LI>Thorough coverage of Java 2; " +
                "including Threads, Networking, Swing, \n" +
                "JavaBeans, and JSP"
            )
        };
}

```

```
"Java 2D, RMI, JDBC, and Collections\n" +
"<LI>A fast introduction to HTML 4.01, " +
"including frames, style sheets, and layers.\n" +
"<LI>A fast introduction to HTTP 1.1, " +
"servlets, and JavaServer Pages.\n" +
"<LI>A quick overview of JavaScript 1.2\n" +
"</UL>",
49.99),
new CatalogItem
("lewis001",
"<I>The Chronicles of Narnia</I> by C.S. Lewis",
"The classic children's adventure pitting " +
"Aslan the Great Lion and his followers\n" +
"against the White Witch and the forces " +
"of evil. Dragons, magicians, quests, \n" +
"and talking animals wound around a deep " +
"spiritual allegory. Series includes\n" +
"<I>The Magician's Nephew</I>,\n" +
"<I>The Lion, the Witch and the Wardrobe</I>,\n" +
"<I>The Horse and His Boy</I>,\n" +
"<I>Prince Caspian</I>,\n" +
"<I>The Voyage of the Dawn Treader</I>,\n" +
"<I>The Silver Chair</I>, and \n" +
"<I>The Last Battle</I>.",
19.95),
new CatalogItem
("alexander001",
"<I>The Prydain Series</I> by Lloyd Alexander",
"Humble pig-keeper Taran joins mighty " +
"Lord Gwydion in his battle against\n" +
"Arawn the Lord of Annuvon. Joined by " +
"his loyal friends the beautiful princess\n" +
"Eilonwy, wannabe bard Fflewddur Fflam," +
"and furry half-man Gurgi, Taran discovers " +
"courage, nobility, and other values along\n" +
"the way. Series includes\n" +
"<I>The Book of Three</I>,\n" +
"<I>The Black Cauldron</I>,\n" +
"<I>The Castle of Llyr</I>,\n" +
"<I>Taran Wanderer</I>, and\n" +
"<I>The High King</I>.",
19.95),
new CatalogItem
("rowling001",
"<I>The Harry Potter Series</I> by J.K. Rowling",
"The first five of the popular stories " +
"about wizard-in-training Harry Potter\n" +
"topped both the adult and children's " +
"best-seller lists. Series includes\n" +
"<I>Harry Potter and the Sorcerer's Stone</I>,\n" +
"<I>Harry Potter and the Chamber of Secrets</I>,\n" +
"<I>Harry Potter and the " +
"Prisoner of Azkaban</I>,\n" +
"<I>Harry Potter and the Goblet of Fire</I>, and\n" +
"<I>Harry Potter and the " +
"Order of the Phoenix</I>.\n",
59.95)
};
```

```
public static CatalogItem getItem(String itemID) {  
    CatalogItem item;  
    if (itemID == null) {  
        return(null);  
    }  
    for(int i=0; i<items.length; i++) {  
        item = items[i];  
        if (itemID.equals(item.getItemId())) {  
            return(item);  
        }  
    }  
    return(null);  
}
```


第 II 部分：JSP 技术

- 第 10 章
JSP 技术概述
- 第 11 章
用 JSP 脚本元素调用 Java 代码
- 第 12 章
控制所生成的 servlet 的结构：JSP page 指令
- 第 13 章
在 JSP 页面中包含文件和 applet
- 第 14 章
JavaBean 组件在 JSP 文档中的应用
- 第 15 章
servlet 和 JSP 的集成：模型-视图-控制器构架
- 第 16 章
简化对 Java 代码的访问：JSP2.0 表达式语言

第 10 章 JSP 技术概述

本章的主题：

- 对 JSP 技术的需求
- 评估 JSP 的好处
- JSP 与其他技术的比较
- 消除对 JSP 的误解
- JSP 页面的安装
- JSP 语法综述

应用 JavaServer Page(JSP)技术可以将常规的、静态的 HTML 与动态生成的内容混合起来。我们只需使用自己熟悉的 Web 页面构建工具，以正常的方式，编写常规 HTML；之后，将页面中生成动态内容的代码包括在特殊的标签中，这些标签大多数情况下以<%开始，以%>结束。

例如，清单 10.1(图 10.1)给出一个十分简单的 JSP 页面，它使用请求参数显示书的标题。要注意到，清单中大部分为标准的 HTML；动态代码总共只有半行，在清单中以粗体显示。

清单 10.1 OrderConfirmation.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Order Confirmation</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>Order Confirmation</H2>
Thanks for ordering <I><%= request.getParameter("title") %></I>!
</BODY></HTML>
```

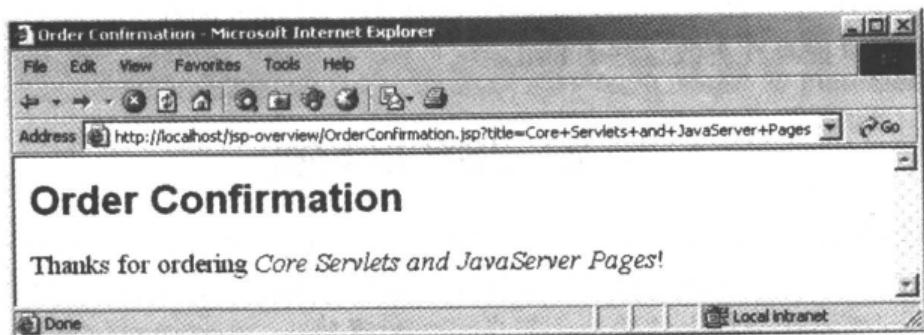


图 10.1 OrderConfirmation.jsp 的结果

我们可以将 servlet 看作是含有 HTML 的 Java 代码；可以将 JSP 可以看作是含有 Java 代码的 HTML。在此，虽然不管 servlet 还是 JSP 页面都非仅仅局限于使用 HTML，但它们

常常会使用 HTML，这种过度精简的描述是人们对这两项技术的通常看法。

尽管 JSP 页面和 servlet 之间存在巨大而且明显的差异，但实际上它们是一回事。JSP 页面最终要转换成 servlet，并进行编译，在请求期间执行的实际上是编译后的 servlet。因而，编写 JSP 页面实际上不过是另一种编写 servlet 的方式。

尽管 servlet 和 JSP 页面实际上是等同的，它们并非在所有的情况下都同样适用。将静态 HTML 从动态内容中分离开来比单独使用 servlet 有许多好处，同时，JSP 页面中使用的模式与竞争技术相比有好几项优点。本章说明使用 JSP 的理由，论述它的好处，消除一些误解，并展示如何安装和执行 JSP 页面，同时，对本书余下的部分中论述的 JSP 语法进行汇总。

10.1 对 JSP 的需求

“嘿！”您可能会说，“你花了好几章的内容赞美 servlet 的诸多优点。我喜欢 servlet。servlet 的编写比较方便，并且执行效率较高。它们使得请求参数的读取，以及建立自定义代码处理缺失和异常数据变得简单。它们能够容易地使用 HTTP 请求报头，并且能够灵活地操纵 HTTP 响应数据。它们能够基于 cookie 定制自身的行为，用会话跟踪 API 来跟踪用户的具体数据，以及使用 JDBC 与关系型数据库进行对话。我还需要什么呢？”

是的，这种观点很好。servlet 确实很有用，JSP 绝不是废弃它。但是，请检查一下 servlet 所擅长的主题。它们都是与编程(programming)或数据处理(data processing)相关的任务。而 servlet 并不擅长表示(presentation)。在需要生成输出的情况下，servlet 有下面的不足：

- **servlet 中 HTML 的编写和维护都比较困难。**

使用 print 语句生成 HTML 肯定不会容易：您必须使用括号和分号，必须在嵌入的双引号前插入反斜杠，还必须使用字符串拼接将内容组合在一起。此外，由于它看起来并不像 HTML，因此很不直观。将这种 servlet 风格与清单 10.1 对比就会发现二者之间的差别，您甚至很难注意到清单 10.1 列出的不是常规的 HTML 文档。

- **不能使用标准的 HTML 工具。**

所有那些您曾使用过的优秀的网站开发工具，在编写 Java 代码时没有用武之地。

- **非 Java 开发人员难以处理这些 HTML。**

如果将 HTML 嵌入到 Java 代码中，那么不了解 Java 编程语言的 Web 开发专业人员就难以检查和更改这些 HTML。

10.2 JSP 的好处

JSP 页面最终会转换成 servlet。因而，从根本上，JSP 页面能够执行的任何任务都可以用 servlet 来完成。然而，这种底层的等同性并不意味着 servlet 和 JSP 页面对于所有的情况都同等适用。问题不在于技术的能力，而是二者在便利性、生产率和可维护性上的不同。毕竟，在特定平台上能够用 Java 编程语言完成的事情，同样可以用汇编语言来完成，但是选择哪种语言依旧十分重要。

和单独使用 servlet 相比，JSP 提供下述好处：

- JSP 中 HTML 的编写与维护更为简单。

JSP 中可以使用常规的 HTML：没有额外的反斜杠，没有额外的双引号，也没有暗含的 Java 语法。

- 能够使用标准的网站开发工具。

例如，本书中大部分 JSP 页面，都是我们用 Macromedia Dreamweaver 编写的。即使那些对 JSP 一无所知的 HTML 工具，我们也可以使用，因为它们会忽略 JSP 标签(JSP tag)。

- 可以对开发团队进行划分。

Java 程序员可以致力于动态代码。Web 开发人员可以将精力集中在表示层(presentation layer)上。对于大型的项目，这种划分极为重要。依据开发团队的大小，及项目的复杂程度，可以对静态 HTML 和动态内容进行弱分离(weaker separation)和强分离(stronger separation)。

在此，这个讨论并不是让您停止使用 servlet，只使用 JSP。几乎所有的项目都会同时用到这两种技术。针对项目中的某些请求，您会选择使用 servlet；针对其他的请求，您可能会选择使用 JSP；对于另外的请求，您可能会在 MVC 构架(第 15 章)下组合使用这两项技术。我们总是希望用适当的工具完成相应的工作，仅仅是 servlet 并不能填满您的工具箱。

10.3 JSP 相对于竞争技术的优势

许多年前，本书的主作者(Marty)受到邀请，参加一个有关软件技术的小型(20 个人)研讨会。坐在 Marty 旁边的人是 James Gosling——Java 编程语言的发明者。隔几个位置，是来自华盛顿雷蒙德一家超大型软件公司的高级经理。在讨论过程中，研讨会的主席提出了 Jini 的议题，这在当时是一项新的 Java 技术。主席向该经理询问他的想法，这个经理回答说现在谈论它还为时过早，但看起来它好像是一个出色的想法。他继续说，他们会持续关注这项技术，如果这项技术变得流行起来，他们会遵循公司的“接受并扩充(embrace and extend)”的策略。此时，Gosling 随意地插话说“你的意思其实就是不接受且不扩充(disgrace and distend)。”

在此，Gosling 的抱怨显示出，他感到这个公司会从其他公司那里拿走技术，用于他们自己的目的。但你猜这次怎么样？这次鞋子穿在了另一只脚上。Java 社团没有发明这一思想——将页面设计成由静态 HTML 和用特殊标签标记的动态代码混合组成。ColdFusion 多年前就已经这样做了。甚至 ASP(来自于前述经理所在公司的一项产品)都在 JSP 出现之前推广了这种方式。实际上，JSP 不只采用了这种通用概念，它甚至使用许多和 ASP 相同的特殊标签。

因此，问题变成：为什么使用 JSP，而不使用其他技术呢？我们的第一反应是我们不是在争论所有的人应该做什么。其他这些技术中，有一些也很不错，在某种情况下也的确是合情合理的选择。然而，在其他情形中，JSP 明显要更好一些。下面给出几个理由。

10.3.1 与.NET 和 Active Server Page(ASP)相比

.NET 是 Microsoft 精心设计的一项技术。ASP.NET 是与 servlet 和 JSP 直接竞争的技术。

JSP 的优势体现在两个方面。

首先, JSP 可以移植到多种操作系统和 Web 服务器, 您不必仅仅局限于部署在 Windows 和 IIS 上。尽管核心.NET 平台可以在好几种非 Windows 平台上运行, 但 ASP 这一部分不可以。您不能期望可以将重要的 ASP.NET 应用部署到多种服务器和操作系统。对于某些应用, 这种差异没有什么影响。但有些应用, 这种差异却非常重要。

其次, 对于某些应用, 底层语言的选择至关重要。例如, 尽管.NET 的 C# 语言设计优良, 且和 Java 类似, 但熟悉核心 C# 语法和众多工具库的程序员很少。此外, 许多开发者依旧使用最初版本的 ASP。相对于这个版本, JSP 在动态代码方面拥有明显的优势。使用 JSP, 动态部分是用 Java 编写, 而非 VBScript 或其他 ASP 专有的语言, 因此 JSP 更为强劲, 更适合于要求组件重用的复杂应用。

当将 JSP 与之前版本的 ColdFusion 对比时, 您可能会得到相同的结论。应用 JSP, 您可以使用 Java 编写“真正的代码”, 不必依赖于特定的服务器产品。然而, 当前版本的 ColdFusion 满足 J2EE 服务器的环境, 允许开发者容易地混合使用 ColdFusion 和 servlet/JSP 代码。

10.3.2 与 PHP 相比

PHP(“PHP: Hypertext Preprocessor”的递归首字母缩写词)是免费的、开放源码的、HTML 嵌入其中的脚本语言, 与 ASP 和 JSP 都有某种程度的类似。JSP 的一项优势是动态部分用 Java 编写, 而 Java 已经在联网、数据库访问、分布式对象等方面拥有广泛的 API, 而 PHP 需要学习全新的、应用相对不广泛的语言。JSP 的第二项优势是, 和 PHP 相比, JSP 拥有极为广泛的工具和服务器提供商的支持。

10.3.3 与纯 servlet 相比

原则上, JSP 并没有提供 servlet 不能完成的功能。实际上, JSP 文档在后台被自动转换成 servlet。但是编写(和修改)常规的 HTML, 要比使用无数 println 语句生成 HTML 要方便得多。另外, 通过将表示与内容分离, 可以为不同的人分配不同的任务: 网页设计人员使用熟悉的工具构建 HTML, 要么为 servlet 程序员留出空间插入动态内容, 要么通过 XML 标签间接调用动态内容。

这是否表示您可以只学习 JSP, 将 servlet 丢到一边呢? 当然不是! 由于以下 4 种原因, JSP 开发人员需要了解 servlet:

- (1) JSP 页面会转换成 servlet。不了解 servlet 就无法知道 JSP 如何工作。
- (2) JSP 由静态 HTML、专用的 JSP 标签和 Java 代码组成。哪种类型的 Java 代码呢?
当然是 servlet 代码! 如果不了解 servlet 编程, 那么就无法编写这种代码。
- (3) 一些任务用 servlet 完成比用 JSP 来完成要好。JSP 擅长生成由大量组织有序的结构化 HTML 或其他字符数据组成的页面。servlet 擅长生成二进制数据, 构建结构多样的页面, 以及执行输出很少或者没有输出的任务(比如重定向)。
- (4) 有些任务更适合于组合使用 servlet 和 JSP 来完成, 而非单独使用 servlet 或 JSP。
详细情况参见第 15 章。

10.3.4 与 JavaScript 相比

JavaScript 和 Java 编程语言完全是两码事，前者一般用于在客户端动态生成 HTML，在浏览器载入文档时构建网页的部分内容。这是一项有用的功能，一般与 JSP 的功能(只在服务器端运行)并不发生重叠。和常规 HTML 页面一样，JSP 页面依旧可以包括用于 JavaScript 的 SCRIPT 标签。实际上，JSP 甚至能够用来动态生成发送到客户端的 JavaScript。因此，JavaScript 不是一项竞争技术，它是一项补充技术。

JavaScript 也可以用在服务器端，最引人注意的是 Sun ONE(以前的 iPlanet)、IIS 和 BroadVision 服务器。然而，Java 更为强大、灵活、可靠且可移植。

10.3.5 与 WebMacro 或 Velocity 相比

JSP 决非完美。许多人都曾指出过 JSP 中能够改进的功能。这是一件好事，JSP 的优势之一是该规范由许多不同公司组成的社团所控制。因此，在后续版本中，这项技术能够得到协调的改进。

但是，一些组织已经开发出了基于 Java 的替代技术，试图弥补这些不足。据我们的判断，这样做是错误的。使用扩充 JSP 和 servlet 技术的第三方工具，如 Apache Structs(参见本书第二卷)，是一种好的思路，只要该工具带来的好处能够补偿工具带来的额外复杂性。但是，试图使用非标准的工具代替 JSP 则不理想。在选择一项技术时，需要权衡许多方面的因素：标准化、可移植性、集成性、行业支持和技术特性。对于 JSP 替代技术的争论几乎只是集中在技术特性上，而可移植性、标准化和集成性也十分重要。例如，如 2.11 节所述，servlet 和 JSP 规范为 Web 应用定义了一个标准的目录结构，并提供用于部署 Web 应用的标准文件(.war 文件)。所有 JSP 兼容的服务器必须支持这些标准。我们可以建立过滤器(第二卷)作用到任意数目的 servlet 或 JSP 页面上，但不能用于非标准资源。Web 应用的安全设置也同样如此(参见第二卷)。

此外，业界对 JSP 和 servlet 技术的巨大支持使得这两项技术都有了巨大的进步，从而减轻了对 JSP 的许多批评。例如，JSP 标准标签库(第二卷)和 JSP 2.0 表达式语言(第 16 章)解决了两种最广泛的批评：缺乏良好的迭代结构；不使用显式的 Java 代码或冗长的 `jsp:useBean` 元素难以访问动态结果。

10.4 对 JSP 的误解

本节中，我们介绍一些对 JSP 技术最常见的误解。

10.4.1 忘记 JSP 技术是服务器端技术

本书的网站列出了主作者的电子邮件地址：hall@coreservlets.com。另外，Marty 为许多公司讲授 JSP 和 servlet 培训课程，还在许多公共聚会上讲授相关课程。因此，他收到了许多询问 servlet 和 JSP 问题的电子邮件。下面是他收到的一些典型问题(大部分问题不止一次地出现)。

- 我们的服务器正在运行 JDK 1.4。我如何将 Swing 组件用到 JSP 页面中呢？

- 我如何将图像放到 JSP 页面中？我不知道读取图像文件应该使用哪些 Java I/O 命令。
- Tomcat 不支持 JavaScript，当用户在图像上移动鼠标时，我如何使图像突出显示呢？
- 我们的客户使用不理解 JSP 的旧浏览器。我应该怎么做？
- 当我们的客户在浏览器中使用“View Source”(查看源代码)时，如何阻止它们看到 JSP 标签？

所有这些问题都基于浏览器对服务器端的过程有所了解的假定之上。但事实上浏览器并不了解服务器端的过程。因此：

- 如果要将使用 Swing 组件的 applet 放到网页中，重要的是浏览器的 Java 版本，和服务器的 Java 版本无关。如果浏览器支持 Java 2 平台，您可以使用正常的 APPLET(或 Java 插件)标签，即使在服务器上使用非 Java 技术也须如此。
- 您不需要 Java I/O 来读取图像文件，您只需将图像放在存储 Web 资源的目录中(即 WEB-INF/classes 向上两级的目录)，并输出一个正常的 IMG 标签。
- 您应该用 SCRIPT 标签，使用客户端 JavaScript 创建在鼠标下会更改的图像，这不会由于服务器使用 JSP 而改变。
- 浏览器根本不“支持”JSP——它们看到的只是 JSP 页面的输出。因此，如同对待静态 HTML 页面一样，只需确保 JSP 输出的 HTML 与浏览器兼容。
- 当然，您不需要采取什么措施来阻止客户看到 JSP 标签，这些标签在服务器上进行处理，发送给客户的输出中并不出现。

10.4.2 混淆转换期间和请求期间

JSP 页面需要转换成 servlet。servlet 在编译后，载入到服务器的内容中，初始化并执行。但是每一步发生在什么时候呢？要回答这个问题，要记住以下两点：

- JSP 页面仅在修改后第一次被访问时，才会被转换成 servlet 并进行编译；
- 载入到内存中、初始化和执行遵循 servlet 的一般规则。

表 10.1 列出一些常见的情形，讲述在该种情况下每一步是否发生。最常被误解的项已经突出标示出来。在参考该表时，要注意，由 JSP 页面生成的 servlet 使用 _jspService 方法 (GET 和 POST 请求都调用该函数)，不是 doGet 或 doPost 方法。同样，对于初始化，它们使用 jsplInit 方法，而非 init 方法。

表 10.1 各种情况下的 JSP 操作

	将 JSP 页面转换成 servlet	编译 servlet	将 servlet 载入到服务器内存中	调用 jsplInit	调用 _jspService
页面初次创建					
请求 1	有	有	有	有	有
请求 2	无	无	无	无	有

服务器重启后					
请求 3	无	无	有	有	有
请求 4	无	无	无	无	有
页面修改后					
请求 5	有	有	有	有	有
请求 6	无	无	无	无	有

10.4.3 认为只有 JSP 就足够

有一个由开发人员组成的小社团，他们对 JSP 是如此倾心，以至于干什么都使用 JSP。这些开发人员中大部分从未使用过 servlet；多数甚至从未使用过补充性的辅助类——针对每项任务他们都构建大型和复杂的 JSP 页面。

这样做是错误的。JSP 是一种优秀的工具，但它所处理的基本问题是表示(presentation)：用来解决创建和维护 HTML 来表示请求结果的困难。对于格式相对固定且含有许多静态文本的页面，JSP 是一种好的选择。但仅仅是 JSP 自己则不太适合于结构不固定的应用；也不适合于大部分由动态数据组成的应用；对于输出二进制数据，或操作 HTTP 但并不生成明确输出的应用(如 6.4 节中的搜索引擎 servlet)更是完全不适用。还有一些应用，单独使用 servlet 或 JSP 都不能很好地解决，而要组合使用这两项技术(参见第 15 章)。

JSP 是强大且应用广泛的工具。不过，有时其他工具更为适用。要为所做的事情选择正确的工具。

10.4.4 认为只有 servlet 就足够

在这个问题上，与 JSP 至上阵营对应的另一个极端是 servlet 至上阵营。这个阵营的追随者声称(比较正确)：JSP 页面实际上不过是经过装扮的 servlet，因此 JSP 页面不能完成 servlet 所不能完成的任何事情。据此，他们得出结论，您应该坚持使用 servlet，从而能够拥有对底层功能的完全访问，拥有完全的控制，同时能够看到发生的任何事。嗯，此前您听到过这类争论吗？它听起来很像那些声称“不要做一个懦弱的人，用汇编语言编写所有的代码”的人的立场。

是的，您能够用 servlet 来完成任何使用 JSP 完成的任务。但这两种做法有时并非同样方便。对于涉及大量静态 HTML 内容的任务，JSP 技术的应用(或组合使用 JSP 和 servlet)能够简化 HTML 的创建和维护，允许您使用行业中标准的网站创建工具，并且能够使您将工作在 Java 开发人员和 Web 开发人员之间进行划分，从而做到“分而治之”。

servlet 是强大且应用广泛的工具。不过，有时其他工具会更适合。要为所做的事情选择正确的工具。

10.5 JSP 页面的安装

servlet 需要您设置 CLASSPATH，使用包来避免命名冲突，将类文件安装在 servlet 专用的位置，同时需要使用专用的 URL，而 JSP 页面则不需要这些。JSP 页面可以和常规 HTML

页面、图像和样式表放在相同的目录中；还可以用与 HTML 页面、图像和样式表形式相同的 URL 访问它们。下面给出默认安装位置(即在没有使用定制 Web 应用时的位置)及与 URL 相关的几个例子。在我们列出 *SomeDirectory* 的位置，您可以使用任何喜欢的目录名，除了永远不允许将 WEB-INF 或 META-INF 用作目录名之外。在使用默认 Web 应用时，您还必须避免使用与任何其他 Web 应用的 URL 前缀相匹配的目录名。有关如何定义自己的 Web 应用，请参见 2.11 节。

10.5.1 Tomcat 中的 JSP 目录(默认 Web 应用)

- 主位置
install_dir/webapps/ROOT
- 对应的 URL
http://host/SomeFile.jsp
- 更具体的位置(任意子目录)
install_dir/webapps/ROOT/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.jsp

10.5.2 JRun 中的 JSP 目录(默认 Web 应用)

- 主位置
install_dir/servers/default/default-ear/default-war
- 对应的 URL
http://host/SomeFile.jsp
- 更具体的位置(任意子目录)
install_dir/servers/default/default-ear/default-war/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.jsp

10.5.3 Resin 中的 JSP 目录(默认 Web 应用)

- 主位置
install_dir/doc
- 对应的 URL
http://host/SomeFile.jsp
- 更具体的位置(任意子目录)
install_dir/doc/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.jsp

要注意，尽管 JSP 页面自身不需要专门的安装目录，JSP 页面调用的任何 Java 类依旧需要放在由 *servlet* 类使用的标准位置(例如 .../WEB-INF/classes/*directoryMatching*

PackageName, 参见 2.10 节)。还要注意, JSP 使用的 Java 类总是应该在包中; 这一点在后面的章节进一步讨论。

10.6 基本语法

下面对本书中使用的各种 JSP 构造进行简短的汇总。

10.6.1 HTML 文本

- **描述**
HTML 内容, 不加更改地传递给客户。
- **示例**
`<H1>Blah</H1>`
- **何处论述**
11.1 节

10.6.2 HTML 注释

- **描述**
HTML 注释, 发送给客户, 但不被浏览器显示。
- **示例**
`<!-- Blah -->`
- **何处论述**
11.1 节

10.6.3 模板文本

- **描述**
不加更改地发送给客户的文本。HTML 文本和 HTML 注释只是这种情况的特例。
- **示例**
除后续小节介绍的语法之外的所有内容。
- **何处论述**
11.1 节

10.6.4 JSP 注释

- **描述**
开发人员注释, 不发送到客户程序。
- **示例**
`<%-- Blah --%>`
- **何处论述**
11.1 节

10.6.5 JSP 表达式

- **描述**
表达式，每次请求页面时都计算值并发送到客户程序。
- **示例**
`<%= Java Value %>`
- **何处论述**
11.4 节

10.6.6 JSP Scriptlet

- **描述**
每次请求页面时执行的一个或多个语句。
- **示例**
`<% Java Statement %>`
- **何处论述**
11.7 节

10.6.7 JSP 声明

- **描述**
在页面转换成 servlet 时，成为类定义的一部分的字段或方法。
- **示例**
`<%! Field Definition %>`
`<%! Method Definition %>`
- **何处论述**
11.10 节

10.6.8 JSP 指令

- **描述**
servlet 代码的高层结构信息(page)、页面转换期间引入的代码(include)或所采用的定制标签库(taglib)。
- **示例**
`<%@ directive att="val" %>`
- **何处论述**
page: 第 12 章
include: 第 13 章
taglib 和 tag: 第二卷

10.6.9 JSP 动作

- **描述**

页面被请求时应该采取的动作。

- **示例**

```
<jsp:blah>...</jsp:blah>
```

- **何处论述**

`jsp:include` 及相关内容：第 13 章

`jsp:useBean` 及相关内容：第 14 章

`jsp:invoke` 及相关内容：第二卷

10.6.10 JSP 表达式语言的元素

- **描述**

简写的 JSP 表达式

- **示例**

```
 ${ EL Expression }
```

- **何处论述**

第 16 章

10.6.11 定制标签(定制动作)

- **描述**

对定制标签的调用。

- **示例**

```
<prefix:name>  
Body  
</prefix:name>
```

- **何处论述**

第二卷

10.6.12 转义的模板文本

- **描述**

需要特殊解释的文本。斜杠被移除，剩余的文本发送到客户端。

- **示例**

```
<\%  
%\\>
```

- **何处论述**

11.1 节

第 11 章 用 JSP 脚本元素调用 Java 代码

本章的主题：

- 静态和动态文本
- 动态代码和好的 JSP 设计
- 包对 JSP 辅助/实用工具类的重要性
- JSP 表达式
- JSP scriptlet
- JSP 声明
- 由 JSP 脚本元素生成的 servlet 代码
- scriptlet 和条件性文本
- 预定义变量
- servlet 和 JSP 对类似任务的处理

本章介绍在 JSP 页面内调用 Java 代码的“经典”方式。在 JSP 1(即 JSP 1.2 和之前的版本)和 JSP 2 中都可以使用这种方式。第 16 章介绍 JSP 表达式语言(expression language)，它为间接调用 Java 代码提供了一种简洁的机制，但仅限于 JSP 2.0 及以后的版本。

11.1 模板文本的创建

大多数情况下，JSP 文档的大部分由静态文本(一般是 HTML)构成，称之为模板文本(template text)。几乎从任何方面看，这种 HTML 都类似于普通的 HTML，它们遵循所有相同的语法规则，为处理该页面而创建的 servlet 只是将它们原封不动地传递给客户程序。这些 HTML 不仅看起来很普通，还可以使用任何已有的用于构建网页的工具创建它们。例如，本书中的许多 JSP 页面都是用 Macromedia Dreamweaver 创建的。

“模板文本会原封不动地直接传递”这一准则存在两个小的例外。第一，如果输出中含有<%或%>，需要在模板文本中使用<\%或%\>；第二，如果希望添加一段注释，使之出现在 JSP 页面中但不出现在结果文档中，需要使用

```
<%-- JSP Comment --%>
```

下面这种形式的 HTML 注释按照常规的方式传递给客户程序。

```
<!-- HTML Comment -->
```

11.2 在 JSP 中调用 Java 代码

如图 11.1 所示，在 JSP 中可以使用许多不同的方式生成动态内容。这些方式中的每一种都有其合理的应用范围。在决定哪种方式更为恰当时，项目的大小和复杂度是需要考虑的最重要的因素。但要知道，人们更容易犯将太多代码直接放在页面中的错误，这种情况

发生的机率要比其对立面大得多。尽管对于简单的应用，将少量的 Java 代码直接放到 JSP 页面中也工作得很好，但如果在 JSP 页面中使用大量复杂的 Java 代码块，造成的后果是难以维护，难以调试，难以重用，并且很难在开发团队的不同成员间划分工作。详细情况参见 11.3 节。不过，许多页面十分简单，图 11.1 中的前两种方式就十分奏效。本章就论述这两种方式。

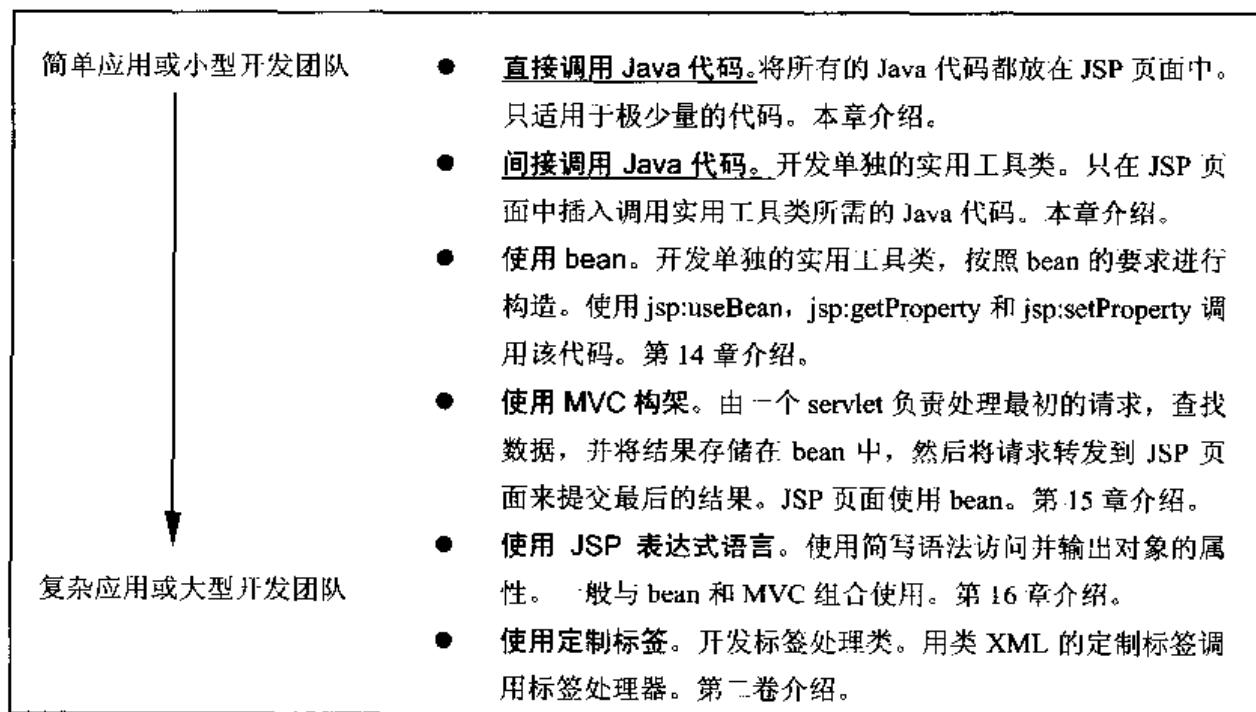


图 11.1 JSP 中调用动态代码的策略

JSP 脚本元素的类型

使用 JSP 脚本元素可以将 Java 代码插入到与 JSP 页面相对应的 servlet 中。脚本元素有 3 种形式：

- (1) 形如`<%= Java Expression %>`的表达式，它们在求值后插入到 servlet 的输出之中。
- (2) 形如`<% Java Code %>`的 scriptlet，它们将插入到 servlet 的 `_jspService` 方法(由 `service` 方法调用)中。
- (3) 形如`<%! Field/Method Declaration %>`的声明，它们将插入到 servlet 类的定义中，不属于任何已有的方法。

后面的小节将对这些脚本元素进行更为详细的叙述。

11.3 限制 JSP 页面中 Java 代码的量

假定需要调用 25 行的 Java 代码，那么您有两种选择：(1)将全部 25 行代码直接放在 JSP 页面中，或者(2)将 25 行代码放在单独的 Java 类中，将 Java 类放在 WEB-INF/classes/directory MatchingPackageName，使用一或两行基于 JSP 的 Java 代码调用它。哪种方式更好呢？第二种，当然是第二种，这是毫无疑问的！更多的代码，比如 50 行、100 行、500 行或 1000

行代码，都是如此。理由如下：

- **开发**

我们一般在面向 Java 的开发环境中(例如，IDE，如 JBuilder 或 Eclipse；或者代码编辑器，如 UltraEdit 或 emacs)编写常规的类。一般在面向 HTML 的环境中，如 Dreamweaver，编写 JSP。面向 Java 的开发环境一般在平衡圆括号、提供工具提示、检查语法、代码着色等方面做得更好。

- **编译**

如果要编译常规的 Java 类，只需在 IDE 中单击 Build 或调用 javac；要编译 JSP 页面，则必须将它放到正确的目录中，启动服务器，打开浏览器并输入相应的 URL。

- **调试**

在我们编写 Java 类或 JSP 页面时，语法上的错误在所难免。如果在常规的类定义中含有语法错误，编译器会马上告诉我们，还会给出代码中错误所在的行。如果在 JSP 页面中存在语法错误，服务器一般提示错误出现在 servlet(即由 JSP 转换成的 servlet)中哪一行。如果要在运行期间输出跟踪信息，对于常规的类，如果所使用的 IDE 没有提供更好的方法，也可以使用简单的 System.out.println 语句。在 JSP 中，虽然有些情况下可以使用打印语句，但这些打印语句显示在什么地方则因服务器而异。

- **工作划分**

许多大型的开发团队由精通 Java 语言的人员和精通 HTML 但对 Java 知之甚少，甚至一无所知的人员组成。直接出现在页面中的 Java 代码越多，Web 开发人员就越难以操作它。

- **测试**

假定希望制作一个 JSP 页面，输出 1 到指定的某个界限(包括在内)之间的随机整数。应该怎么做呢？是使用 Math.random，乘以这个区间，将结果转换成 int 并加 1 吗？听起来好像不错。但这样做正确吗？如果您直接在 JSP 页面中执行这些运算，那么，在测试时，必须一次一次地调用该页面，检查是否得到指定区间内的所有数字，且区间之外的数字没有出现。将 Reload 按钮点击了数十次之后，您就会对测试感到厌烦。但是，如果您在常规 Java 类的静态方法中完成这项任务，则可以编写测试例程，由它在循环中调用该方法，这样您就可以成百上千次地运行测试用例，没有任何问题。对于更复杂的方法，您可以保存输出，只要对方法做出修改，就用新的输出与之前保存的结果进行比较。

- **重用**

您将一些代码放到 JSP 页面中，后来，您发现需要在不同的 JSP 页面中完成相同的事情。应该怎么办呢？剪切、粘贴？嘘！以这种方式重复代码是万恶之源，因为如果您要对代码做出更改，那么必须更改很多不同地方的代码。解决代码重用的问题是面向对象编程的全部内容。不要仅仅因为使用 JSP 来简化 HTML 的生成，就忘记所有那些好的 OOP 基本准则。

“请等一下”，您说，“我的 IDE 使得开发、调试和编译 JSP 页面变得更为容易。”

不错。不存在硬性和快速的准则可以精确地确定直接在页面中放置多少 Java 代码才算太多。但是，没有 IDE 能够解决测试和重用问题，核心的通用设计策略应该是：把复杂的代码放在常规 Java 类中，保持 JSP 页面相对简单。

核心方法

JSP 页面中 Java 代码的数量应该有所限制。至少可以使用一些辅助类，然后在 JSP 页面中调用它们。经验更丰富之后，应该考虑使用 bean、MVC 和定制标签。

几乎所有经验丰富的开发人员都曾看到过这种毫无节制的使用，即 JSP 页面由许多行 Java 代码，加上很小的几段 HTML 组成。很明显这种做法不好：这样的页面难以开发、编译、调试，也难以在团队成员间分配工作以及重用。直接使用 servlet 要远远好于这种做法。但是，另外一些开发人员则有些反应过度，他们声称 JSP 页面中不应该直接出现任何 Java 代码。当然，对于某些项目，对内容和表示进行严格的分离，并强制执行任何 JSP 页面中都不出现 Java 语法的风格是值得的。但这样做并非总是必需(甚至会毫无益处)。

少数人走得更远，他们声称，所有应用中的所有页面都应该使用模型-视图-控制器(Model-View-Controller, MVC)构架，最好采用 Apache Struts 框架。就我们的观点，这也是一种反应过度。是的，MVC(第 15 章)是一个伟大的思想，并且，我们在实际的项目中常常会使用它。而且，Struts(第二卷)也的确是一个不错的框架；在本书即将出版之际，我们正在将它应用到一个大型项目上。这些方法更适用于中等(一般用 MVC)或高度复杂(Struts)的情况。

而简单的情况则需要简单的解决方案。以我们的观点，图 11.1 中所有的方案都有一个适用的位置：绝大部分依赖于应用的复杂度及开发团队的大小。同时，也要警告大家，初学者更容易犯这样的错误：在 JSP 页面中使用大量 Java 代码，从而使得页面难以管理。相对而言，他们不恰当地使用不必要的大型复杂框架的可能性要小得多。

使用包的重要性

不管编写什么样的 Java 类，类文件都要部署在 WEB-INF/classes/directory MatchingPackageName 中(或放入 WEB-INF/lib 目录中的 JAR 文件中)。不管该类是 servlet、常规辅助类、bean、定制标签处理器，还是其他什么东西都是如此。所有的代码都放在相同的位置。

但是，对于常规 servlet，由于可以使用分离的 Web 应用(参见 2.11 节)来避免与其他项目中的 servlet 发生名称冲突，因此有时使用默认的包也是合理的。然而，对于 JSP 所调用的代码，则一定要使用包。同时，由于在编写供 servlet 使用的实用工具类时，并不知道后来是否同样会在 JSP 中使用它，因而，所应采取的策略是：所有的类，不管是供 servlet 还是 JSP 页面使用，都应该使用包。

核心方法

将所有的类都放在包中。

为什么呢？要回答这个问题，请考虑下面的代码。该代码中可能包含 package 声明，也可能没有，但没有 import 语句。

```
...
public class SomeClass {
    public String someMethod(...) {
        SomeHelperClass test = new SomeHelperClass(...);
        String someString = SomeUtilityClass.someStaticMethod(...);
        ...
    }
}
```

现在的问题是，系统会认为 SomeHelperClass 和 SomeUtilityClass 在哪个包中呢？答案就是 SomeClass 所在的包。这个包由 package 声明给出。都是基本的 Java 语法。没有任何问题。那么，考虑下面的 JSP 代码：

```
...
<%>
SomeHelperClass test = new SomeHelperClass(...);
String someString = SomeUtilityClass.someStaticMethod(...);
%>
```

还是相同的问题：系统会认为 SomeHelperClass 和 SomeUtilityClass 在哪个包中呢？答案相同：当前类(由 JSP 页面转换而成的 servlet)所在的包。嗯，这是个好问题，可惜没人能够知道！JSP 规范并没有将这个包标准化。因此，不使用包的辅助类，在以这种方式使用时，只有在系统构建无包装的 servlet 时可以使用。但是服务器并不总是这样做，因而类似本示例的 JSP 代码可能会失败。更为不利的是，服务器有时确实会将 JSP 页面编译成无包装的 servlet。例如，Tomcat 的大多数版本都将 Web 应用顶层目录中的 JSP 页面编译成无包装的 servlet。问题是没有任何标准可以判断何时它们这样做，何时它们不这样做。如果 JSP 代码总是失败还好(找出问题之所在要容易一些。——译者注)。相反，如果它有时工作正常，有时失败，依赖于服务器，甚至依赖于 JSP 页面在哪个目录中。这可真是够令人头疼的！

如果想可靠、可移植，就要早做打算。无论什么情况，都使用包！

11.4 JSP 表达式的应用

JSP 表达式用来将值直接插入到输出中。它的形式如下：

```
<%= Java Expression %>
```

该表达式在求值、转换成字符串后，插入到页面中。求值是在运行期间执行(在页面被请求时)，因此可以访问到请求的所有信息。例如，下面的代码将会列出页面被请求的日期/时间。

```
Current time: <%= new java.util.Date() %>
```

11.4.1 预定义变量

为了简化这些表达式，我们可以使用多个预定义变量(或“隐式对象”)。这些变量并没有什么特别；系统只是告诉您，_jspService 方法(在由 JSP 页面生成的 servlet 中，它等同于 doGet 的方法)中的局部变量会使用什么名称。11.12 节更详尽地介绍这些隐式对象，但就表达式来说，最重要的一些对象是：

- request 对象，HttpServletRequest。
- response 对象，HttpServletResponse。
- session 对象，与请求相关联的 HttpSession(除非 page 指令的 session 属性将其禁用——参见 12.4 节)。
- out 对象，用来将输出发送到客户端的 Writer(JspWriter 类型的具有缓冲功能的版本)。
- application 对象，ServletContext。这是一个由 Web 应用中所有 servlet 和 JSP 页面共享的数据结构，对于存储共享数据也很有用。我们在有关 bean(第 14 章)和 MVC(第 15 章)的章节对它做进一步的论述。

下面是一个具体的例子：

```
Your hostname: <%= request.getRemoteHost() %>
```

11.4.2 JSP/servlet 的对应

现在，我们可以认为：JSP 表达式经求值后插入到页面的输出中。尽管事实的确如此，但有时理解后台的运作也很有帮助。

实际上十分简单，JSP 表达式基本上成为由 JSP 页面生成的 servlet 中的 print(或 write)语句。常规的 HTML 转换成 print 语句，同时用双引号将文本引起来，而 JSP 表达式转换成没有双引号的 print 语句。这些 print 语句没有放在 doGet 方法中，而是放在了一个新的方法_jspService 中，无论是 GET 和 POST 请求，service 方法都会调用_jspService 方法。例如，清单 11.1 给出一个小型的 JSP 示例，其中包括一些静态 HTML 和一个 JSP 表达式。清单 11.2 给出可能生成的_jspService 方法。当然，不同的提供商生成代码的方式也稍有不同，但优化(如从静态字节数组中读取 HTML)却十分通用。

同样，我们将 out 变量的定义做了极大的简化。JSP 页面中的 out 是一个 JspWriter，因此，我们必须对调用 getWriter 取得的 PrintWriter(稍微简单一些)进行适当的修改。因此，不要期望您使用的服务器能够生成和示例完全相似的代码。

清单 11.1 JSP 表达式示例：Random Number

```
<H1>A Random Number</H1>
<%= Math.random() %>
```

清单 11.2 典型的生成servlet的代码：Random Number

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, IOException {
    Response.setContentType("text/html");
    HttpSession session = request.getSession();
```

```

JspWriter out = response.getWriter();
out.println("<H1>A Random Number</H1>");
out.println(Math.random());
...
}

```

如果希望看到您的服务器生成的代码，必须要花一番心思才能找到它。实际上，一些服务器在成功完成编译之后会将源代码文件删除。下面汇总了 3 种常见的免费开发服务器中存储源代码文件的位置。

- Tomcat 自动生成的 servlet 源代码

install_dir/work/Standalone/localhost/_

(最后的目录是一个下划线。一般来说，位于 *install_dir*/work/Standalone/localhost/*webAppName*。Tomcat 的不同版本，这个位置也会稍有不同。)

- JRun 自动生成的 servlet 源代码

install_dir/servers/default/default-ear/default-war/WEB-INF/jsp

(一般来说，在 JSP 页面所属 Web 应用的 WEB-INF/jsp 目录中。然而，要注意 JRun 并不保存.java 文件，除非您将 *install_dir*/servers/default/SERVER-INF/default-web.xml 中的 keepGenerated 元素从 false 改为 true)。

- Resin 自动生成的 servlet 源代码

install_dir/doc/WEB-INF/work

(一般来说，在 JSP 页面所属 Web 应用的 WEB-INF/work 目录中。)

11.4.3 表达式的 XML 语法

XML 程序设计者可以使用下面的替代语法来书写 JSP 表达式。

<jsp:expression>Java Expression</jsp:expression>

JSP 1.2 及之后的版本中，只要程序设计者没有在同一页面中混合使用 XML 方式和标准的 JSP 方式(<%= ... %>)，就要求服务器支持这种语法。这意味着，要使用 XML 方式，那么整个页面都必须使用 XML 语法。在 JSP 1.2(但不包括 2.0)中，这项需求意味着我们必须将页面全部包括在一个 *jsp:root* 元素中。因此，多数开发人员除了生成 XML 文档(如 XHTML 或 SOAP)，或者 JSP 页面自身是某种 XML 处理过程(如 XSLT)的输出之外，都会坚持这种典型的语法。

要注意，XML 元素不同于 HTML 元素，它们对大小写敏感。因此，一定要使用小写的 *jsp:expression*。

11.5 示例：JSP 表达式

清单 11.3 给出一个 JSP 页面的例子，名为 Expressions.jsp。我们将该文件放置在 *jsp-scripting* 目录中，然后将整个目录从我们的开发目录复制到默认 Web 应用的顶层目录(一般地，WEB-INF 的上级目录就是 Web 应用的顶层目录)，并使用 URL <http://host/jsp-scripting/Expressions.jsp> 对它进行访问。图 11.2 和图 11.3 展示出一些典型的结果。

要注意，我们在 JSP 页面的 HEAD 部分中包括了 META 标签和一个样式表链接。包括这些元素是一种好的做法，但是，常规 servlet 生成的页面中经常会省略它们，原因有二。

第一，使用 servlet 时，生成所需的 `println` 语句十分冗长乏味。然而，使用 JSP，格式更为简洁，并且可以使用常用 HTML 构建工具的代码重用选项。这种便利性是我们选择使用 JSP 的重要因素。JSP 页面并不比 servlet 更强大(它们在后台就是 servlet)，但它们有时比 servlet 要方便得多。

第二，servlet 不能使用最简单的相对 URL(引用当前页面同一目录中的文件的 URL)，因为 servlet 的目录并不像常规 Web 页面那样映射到 URL。此外，服务器不应该允许客户直接访问 WEB-INF/classes(或 WEB-INF 中的任何目录)中的内容。因此，将样式表放在与 servlet 类文件相同的目录中是不可能的，即使您使用 web.xml 的 servlet 和 servlet-mapping 元素(参见 2.11 节)对 servlet 的 URL 进行定制也是如此。而从另一方面讲，JSP 页面安装在服务器中的常规 Web 页面层次中，只要客户程序直接访问 JSP 页面，而非间接通过 RequestDispatcher(参见第 15 章)，相对 URL 就能够正确解析。

因而，大多数情况下，样式表和 JSP 页面可以一同保存在相同的目录中。样式表的源代码，和本书中列出或引用的所有代码一样，可以在 <http://www.coreservlets.com> 找到。

清单 11.3 Expressions.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Expressions</TITLE>
<META NAME="keywords"
      CONTENT="JSP,expressions,JavaServer Pages,servlets">
<META NAME="description"
      CONTENT="A quick example of JSP expressions.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>JSP Expressions</H2>
<UL>
    <LI>Current time: <%= new java.util.Date() %>
    <LI>Server: <%= application.getServerInfo() %>
    <LI>Session ID: <%= session.getId() %>
    <LI>The <CODE>testParam</CODE> form parameter:
        <%= request.getParameter("testParam") %>
</UL>
</BODY></HTML>
```

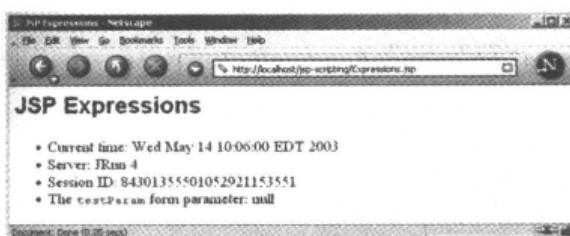


图 11.2 Expressions.jsp 的结果(使用 Macromedia JRun，省略 testParam 请求参数)

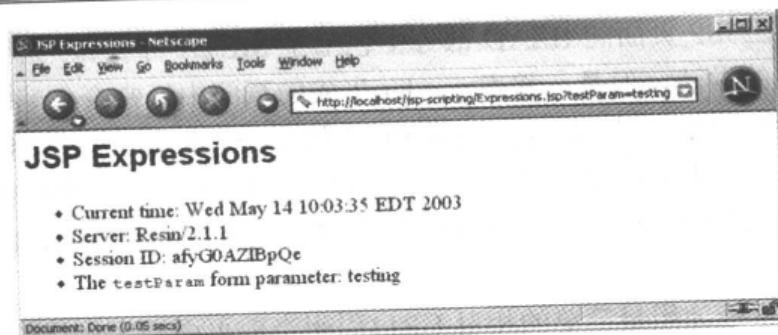


图 11.3 Expressions.jsp 的结果(使用 Caucho Resin, 指定 testing 作为 testParam 请求参数)

11.6 servlet 和 JSP 页面的对比

在 4.3 节中, 我们提供了一个 servlet 示例, 它输出 3 个指定表单参数的值。清单 11.4 重复了该 servlet 的代码。清单 11.5(图 11.4)用 JSP 对它进行了重写, 使用 JSP 表达式来访问表单的参数。JSP 版本明显更胜一筹: 更短、更简单、也更易于维护。

这并不是说所有的 servlet 都能如此整洁地转换成 JSP。JSP 适用于 HTML 页面的结构固定但许多地方的值需要动态计算的情况。如果页面的结果不断变动, 那么 JSP 就不那么适用。这种情况下, 有时使用 servlet 会更好。当然, 如果页面由二进制数据组成, 或仅有极少量的静态内容, 那么明显 servlet 会更好。此外, 有时答案并非是单独的 servlet 或 JSP, 而是这二者的结合。详细信息, 请参见第 15 章。

清单 11.4 ThreeParams.java

```
package coreservlets;

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

public class ThreeParams extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading Three Request Parameters";
        String docType =
            "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
            "Transitional//EN\">\n";
        out.println(docType +
                    "<HTML>\n" +
                    "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                    "<BODY BGCOLOR=#FDF5E6>\n" +
                    "<H1 ALIGN= CENTER>" + title + "</H1>\n" +
                    "<UL>\n" +
                    "  <LI><B>param1</B>: " +
                    + request.getParameter("param1") + "\n" +
                    "  <LI><B>param2</B>: " +
                    + request.getParameter("param2") + "\n" +
                    "  <LI><B>param3</B>: "
```

```

        + request.getParameter("param3") + "\n" +
        "</UL>\n" +
        "</BODY></HTML>");

    }
}

```

清单11.5 ThreeParams.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reading Three Request Parameters</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Reading Three Request Parameters</H1>
<UL>
    <LI><B>param1</B>: <%= request.getParameter("param1") %>
    <LI><B>param2</B>: <%= request.getParameter("param2") %>
    <LI><B>param3</B>: <%= request.getParameter("param3") %>
</UL>
</BODY></HTML>

```

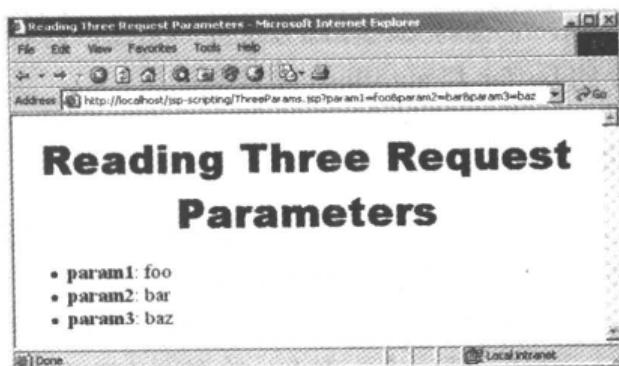


图 11.4 ThreeParams.jsp 的结果

11.7 编写 scriptlet

除了输出简单表达式的值之外，如果希望完成更为复杂的任务，则可以选择使用 JSP scriptlet，它可以将任意代码插入到 servlet 的 `_jspService` 方法中(由 `service` 方法调用)。scriptlet 的形式如下：

```
<% Java Code %>
```

scriptlet 可以访问到表达式能够访问的所有自动定义变量(`request`, `response`, `session`, `out` 等)。例如，如果您希望显式地将输出发送给合成的页面，可以使用 `out` 变量，如下面的例子所示：

```

<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);

```

```
<%>
```

在这个特定的例子中，使用 scriptlet 和 JSP 表达式的结合可以更为容易地完成相同的工作，如下所示：

```
<% String queryData = request.getQueryString(); %>
Attached GET data: <%= queryData %>
```

或者，您也可以使用单个 JSP 表达式，如下所示：

```
Attached GET data: <%= request.getQueryString() %>
```

一般地，scriptlet 可以完成单独使用表达式所不能完成的许多任务。这些任务包括设置响应报头和状态代码，调用边界效应，如写服务器日志或更新数据库，或执行含有循环、条件分支结构或其他复杂结构的代码。例如，下面的小片断指定将当前页作为 Microsoft Word 发送给客户程序，而非 HTML(默认)。由于 Microsoft Word 可以导入 HTML 文档，因此这项技术在实际的应用中十分有用。

```
<% response.setContentType("application/msword"); %>
```

重要的是，我们不必一定在 JSP 的最顶端设置响应报头或状态代码，即使这种功能好像违背了准则——这类响应数据需要在任何文档内容发往客户程序之前指定。在少量的文档内容之后再来设置报头和状态代码是合法的，因为由 JSP 生成的 servlet 使用 Writer 的一种特殊变体(JspWriter 类型)，它会对文档做部分缓冲。然而，这种缓冲行为可能会改变。参见第 12 章中 page 指令的 buffer 和 autoflush 属性的论述。

11.7.1 JSP/servlet 的对应性

了解 JSP scriptlet 与 servlet 代码之间的对应性比较容易。scriptlet 代码只是直接插入到 _jspService 方法中：不需字符串，不用 print 语句，不做任何更改。例如，清单 11.6 给出一个小型的 JSP 示例，它包括一些静态 HTML、一个 JSP 表达式和一个 JSP scriptlet。清单 11.7 给出可能生成的 _jspService 方法。要注意，对 bar 的调用(JSP 表达式)后面并没有加分号，但对 baz 的调用(JSP scriptlet)后面添加了分号。要记住，JSP 表达式包含 Java 值(不以分号结尾)，而大多数 JSP scriptlet 包含 Java 语句(必须以分号结束)。为了更容易判断何时使用分号，只需记住，表达式要放在 print 或 write 语句中，out.print(blah); 显然不合法。

另外，不同的提供商可能会以稍微不同的方式生成这段代码，同时，我们也将 out 变量(JspWriter 类型，不是调用 getWriter 得到的 PrintWriter，JspWriter 要比 PrintWriter 复杂一些)做了极大的简化。因此，不要期望您的服务器生成的代码会和此处列出的代码完全相同。

清单 11.6 JSP 表达式示例/Scriptlet

```
<H2>foo</H2>
<%= bar() %>
<% baz(); %>
```

清单 11.7 典型的生成servlet的代码：表达式/scriptlet

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
```

```

throws ServletException, IOException {
response.setContentType(text/html);
HttpSession session = request.getSession();
JspWriter out = response.getWriter();
out.println(<H2>foo</H2>);
out.println.(bar());
baz();
...
}

```

11.7.2 scriptlet 的 XML 语法

与`<% Java Code %>`等价的 XML 表达式是

```
<jsp:scriptlet>Java Code</jsp:scriptlet>
```

JSP 1.2 和之后的版本，只要程序设计者没有在同一页面中混合使用 XML 版本(`<jsp:scriptlet>...</jsp:scriptlet>`)和类 ASP 的版本(`<% ... %>`)，就要求服务器支持这种语法；如果使用 XML 版本，那么必须在整个页面中一致地使用 XML 语法。切记 XML 元素大小写敏感：一定要使用小写的 `jsp:scriptlet`。

11.8 scriptlet 示例

下面就举一个具体的例子，来说明有时单独使用 JSP 表达式太过复杂。清单 11.8 给出一个 JSP 页面，它使用 `bgColor` 请求参数来设置页面的背景色。简单地使用

```
<BODY BGCOLOR="<% request.getParameter("bgColor") %>">
```

则会违背读取表单数据的主要准则：总是要检查缺失或异常数据。因此，我们使用 `scriptlet` 来完成这项功能。JSP-Styles.css 被略去，这样，样式表就不会覆盖背景色。图 11.5、图 11.6 和图 11.7 分别展示出默认结果、背景为 C0C0C0 的结果，以及 papayawhip(这是一种古怪的 X11 色彩名，由于历史原因大多数浏览器依旧支持它)的结果。

清单 11.8 BGColor.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Color Testing</TITLE>
</HEAD>
<%
String bgColor = request.getParameter("bgColor");
if ((bgColor == null) || (bgColor.trim().equals("")))) {
    bgColor = "WHITE";
}
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Testing a Background of "<%= bgColor %>"</H2>
</BODY></HTML>

```

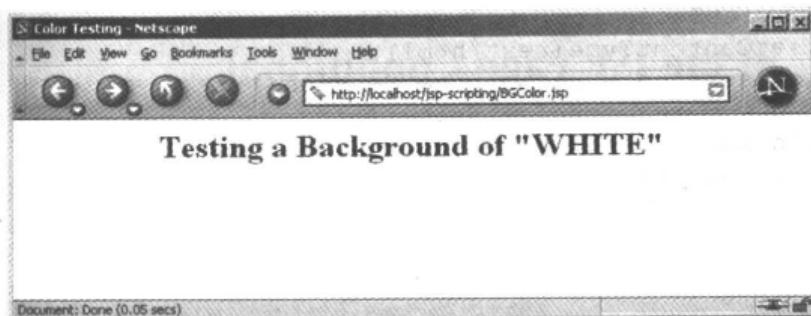


图 11.5 BGColor.jsp 的默认结果

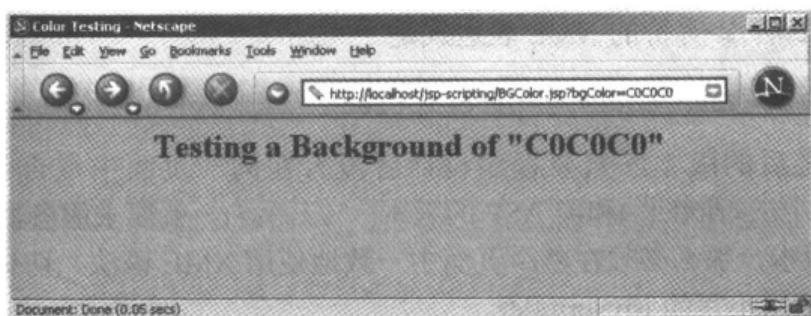


图 11.6 将 bgColor 设为 RGB 值 C0C0C0 访问 BGColor.jsp 的结果

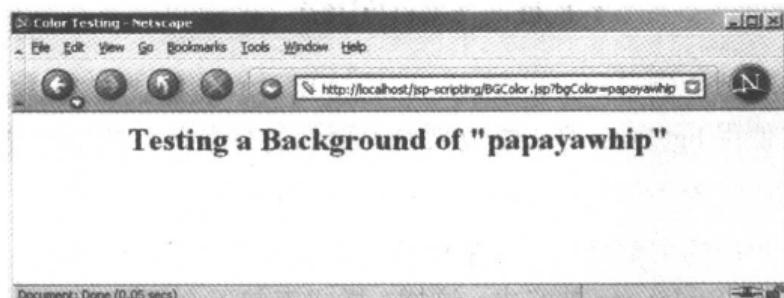


图 11.7 将 bgColor 设为 X11 色彩名 papayawhip 访问 BGColor.jsp 的结果

11.9 使用 scriptlet 将 JSP 页面的某些部分条件化

scriptlet 的另一种用途是条件性地输出 HTML 或其他不在任何 JSP 标签内的内容。这种用途的关键是(1)scriptlet 内的代码会原封不动地插入到由 JSP 页面生成的 servlet 的 _jspService 方法(为 service 方法所调用)中；(2)scriptlet 之前或之后的任何静态 HTML(模板文本)会转换成 print 语句。这种行为意味着：scriptlet 所包含的 Java 语句不一定要完整，开放的代码块会影响到 scriptlet 之外的静态 HTML 或 JSP。以清单 11.9 中的 JSP 片断为例，它混合了模板文本和 scriptlet。

清单 11.9 DayWish.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Wish for the Day</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
```

```

    TYPE="text/css">
</HEAD>
<BODY>
<% if (Math.random() < 0.5) { %>
<H1>Have a <I>nice</I> day!</H1>
<% } else { %>
<H1>Have a <I>lousy</I> day!</H1>
<% } %>
</BODY></HTML>

```

您可能会觉得粗体部分有些混乱。前几次看到拥有这种特征的结构时，的确会有这种感觉。“have a nice day” 和 “have a lousy day” 都不在 JSP 标签之内，但对于任何给定的请求，这二者中只有一个出现在输出中，这好像很奇怪。参见图 11.8 和图 11.9。

不要惊慌！我们只需遵循 JSP 代码转换成 servlet 代码的准则。想清楚 JSP 引擎如何将这个例子转换成 servlet 代码之后，我们就能够得出下面易于理解的结果。

```

if (Math.random() < 0.5) {
    out.println("<H1>Have a <I>nice</I> day!</H1>");
} else {
    out.println("<H1>Have a <I>lousy</I> day!</H1>");
}

```

此处的要点是前两个 scriptlet 中不是完整的语句，而是以悬挂花括号(指没有 “}” 与之匹配的 “{”)结尾的部分语句。这样做是为了将后面的 HTML 内容包括到 if 或 else 子句中。

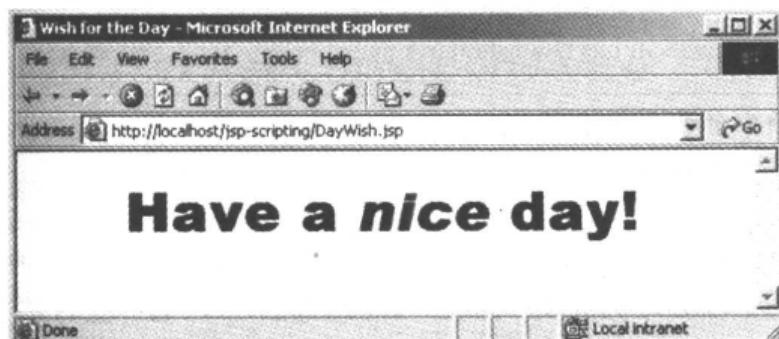


图 11.8 DayWish.jsp 的可能结果之一

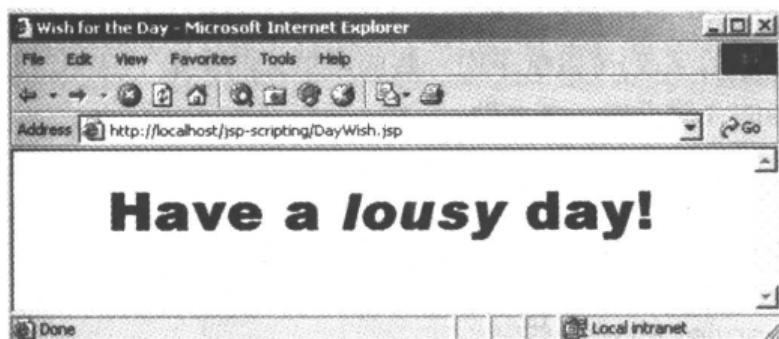


图 11.9 DayWish.jsp 的另一种可能结果

过度使用这种方式会使得 JSP 代码难以理解和维护。应该避免使用它来将大块的 HTML 条件化，同时要力图使 JSP 页面尽可能地集中在表示(HTML 输出)任务上。不过，一些情况下其他的方式没有吸引力。基本的例子是生成条目未定的列表或表格。当您显示

来自于数据库查询的数据时，常常会发生这种情况。有关 Java 代码访问数据库的细节，参见第 17 章。此外，即使您不使用这种方式，也肯定会在项目中看到使用它的例子，您需要理解它如何工作，以及为什么能够完成工作。

11.10 使用声明

使用 JSP 声明可以将方法或字段的定义插入到 servlet 类的主定义体中(位于对请求进行处理的 `_jspService` 方法之外)。声明的形式如下：

```
<%! Field or Method Definition %>
```

由于声明并不产生输出，所以，它们一般与 JSP 或 scriptlet 结合使用。基本上，JSP 声明可以包含字段(实例变量)定义、方法定义、内部类定义、甚至静态初始块：任何可以放在类定义中但在已有方法之外的内容。然而，在实践中，声明几乎总是包含字段或方法定义。

但要注意，不要使用 JSP 声明覆盖 servlet 的标准生命期方法(`service`, `doGet`, `init` 等)。由 JSP 页面转换而成的 servlet 已经使用了这些方法。由于对 `service` 方法的调用会自动分派给 `_jspService`(也就是表达式和 `scriptlet` 生成的代码放置的地方)，因而声明不需要访问 `service`, `doGet` 或 `doPost` 方法。但对于初始化和清理工作，可以使用 `jspInit` 和 `jspDestroy`(在由 JSP 转换而成的 servlet 中，标准的 `init` 和 `destroy` 方法保证会调用这两个方法)。

核心方法

对于 JSP 页面中的初始化和清理工作，可以使用 JSP 声明来覆盖 `jspInit` 或 `jspDestroy`，不要直接使用 `init` 或 `destroy`。

除了覆盖标准方法，如 `jspInit` 和 `jspDestroy`，JSP 声明在定义方法方面的功用是值得质疑的。请将这些方法移到单独的类(可能作为静态方法)中，这样更有利于它们的编写(这是由于您可以使用 Java 开发环境进行编写工作，而不是在类 HTML 环境中进行)、测试(不需要运行服务器)、调试(编译警告能够给出正确的行号；查看标准输出不需要使用其他技巧)和重用(许多不同的 JSP 页面可以使用同一个实用工具类)。但是，很快我们会看到，应用 JSP 声明定义实例变量(字段)可以赋予您一些应用单独的实用工具类不易完成的功能，它为跨请求持续性数据的存储提供一席之地。

核心方法

使用单独的 Java 类而非 JSP 声明来定义绝大多数的方法。

11.10.1 JSP/servlet 的对应性

JSP 声明生成放置在 servlet 类定义之中，`_jspService` 方法之外的代码。由于字段和方法可以以任意次序声明，因此，由声明产生的代码放在 servlet 顶部还是底部并不重要。以清单 11.10 给出的一小段 JSP 片断为例，其中包括一些静态 HTML、一个 JSP 声明和一个 JSP 表达式。清单 11.11 给出可能生成的 servlet。要注意，JSP 规范并没有定义由 JSP 页面

生成的 servlet 的名称，事实上，不同的服务器遵循不同的约定。此外，之前已经声明过，不同的提供商产生这段代码的方式也稍有不同，同时我们对 out 变量做了极大的简化(它是 JspWriter 对象，不是调用 getWriter 取得的 PrintWriter，JspWriter 比 PrintWriter 要复杂一些)。最后，servlet 从不会直接实现 HttpJspPage，而是会扩展一些提供商特有的已实现了 HttpJspPage 的类。因此，不要期望您使用的服务器产生的代码会和下面列出的完全一致。

清单 11.10 JSP 声明示例

```
<H1>Some Heading</H1>
<%!
    Private String randomHeading() {
        Return(<H2> + Math.random() + </H2>);
    }
%>
<%= randomHeading() %>
```

清单 11.11 典型的生成Servlet的代码：声明

```
public class xxxx implements HttpJspPage {
    private String randomHeading() {
        return(<H2> + Math.random() + </H2>);
    }
    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response)
        Throws ServletException, IOException {
        Response.setContentType(text/html);
        HttpSession session = request.getSession();
        JspWriter out = response.getWriter();
        out.println(<H1>Some Heading</H1>);
        out.println(randomHeading());
        ...
    }
    ...
}
```

11.10.2 声明的 XML 语法

与`<%! Field or Method Definition %>`等同的 XML 语法是：

```
<jsp:declaration>Field or Method Definition</jsp:declaration>
```

JSP 1.2 和之后的版本中，只要程序设计者没有在同一个页面混合 XML 版本(`<jsp:declaration> ... </jsp:declaration>`)和标准的类 ASP 版本(`<%! ... %>`)，就要求服务器支持这种语法。如果您要使用 XML 表单，那么整个页面都要遵循 XML 语法，因此，除非使用 XML，大多数开发人员都坚持使用传统语法。切记 XML 元素大小写敏感；一定要使用小写的 `jsp:declaration`。

11.11 声明的例子

在这个例子中，下面的 JSP 片断打印出自服务器启动以来(或 servlet 类更改并重新载入

后)当前页面被请求的次数。点击计数由两行代码实现!

```
<%! private int accessCount = 0; %>
Accesses to page since server reboot:
<%= ++accessCount %>
```

回顾一下,客户对同一 servlet 的多个请求只会产生多个线程,每一个线程都调用单一 servlet 实例的 service 方法。除非 servlet 实现了现在已经不赞成使用的 SingleThreadModel 接口(参见 3.7 节),否则多个请求并不会导致多个 servlet 实例的创建。因此,常规 servlet 的实例变量(字段)为多个请求共享,同时也不必将 accessCount 声明为 static。现在,高级读者可能会想知道刚才列出的片断是否线程安全;这段代码是否能够保证每个访问者都能得到惟一的计数?答案是否定的。原则上,非正常的情况下多个用户可能看到相同的值。对于访问计数,只要在长期运行中该计数是正确的,两个不同的用户偶尔看到相同的计数并不重要。但是,对于某些值,如会话标识符,值的惟一性是至关重要的。第 12 章有关 page 指令的 isThreadSafe 属性的论述中,给出一个和前面的片断类似的例子,但使用 synchronized 块来保证线程安全。

清单 11.12 给出完整的 JSP 页面。图 11.10 展示出一个具有代表性的结果。此时,在着手采用这种方式跟踪对页面的访问之前,需要注意许多事项。

首先,由于每次重启服务器,该计数都会重新开始,因此实际的点击计数并不能使用这种方式。实际的点击计数可能需要使用 jsplInit 和 jsplDestroy 在启动时读取之前的计数,并在服务器关闭时存储旧的计数。

其次,即使您使用 jsplDestroy,服务器也可能会出乎意料地崩溃(例如,当硅谷发生持续不断的电力罢工时)。因此,您还需要定期将点击计数写到磁盘上。

最后,一些高级服务器支持分布式应用,服务器的集群作为单个服务器出现在客户面前。如果您的 servlet 或 JSP 有可能需要支持这种方式的分布,那么就要预先规划,并且避免使用字段来存储持续性数据。应该转而使用数据库。(要注意,只要会话对象的值是 Serializable,那么它就能自动跨分布式应用共享。但会话的值为每个用户所特有,但在这种情况下我们需要的是不依赖于客户的数据。)

清单 11.12 AccessCounts.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Declarations</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>JSP Declarations</H1>
<%! private int accessCount = 0; %>
<H2>Accesses to page since server reboot:
<%= ++accessCount %></H2>
</BODY></HTML>
```

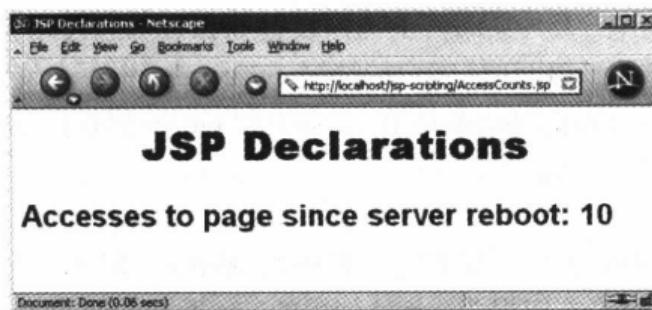


图 11.10 相同或不同的客户程序访问 AccessCounts.jsp9 次之后，再次访问它的结果

11.12 使用预定义变量

在为 servlet 编写 doGet 方法时，您可能会编写如下代码：

```
public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession();
    PrintWriter out = response.getWriter();
    out.println(...);
}
```

servlet API 告诉您 doGet 参数的类型，获取会话对象和输出器对象需要调用的方法，以及它们的类型。JSP 将方法名从 doGet 改为 _jspService，并使用 JspWriter 替代 PrintWriter。但思想是相同的。问题是，谁告诉你应该使用什么样的变量名呢？答案是，没人能够告诉！您可以按照自己的意愿任意选取。

为了更有效地使用 JSP 表达式和 scriptlet，您需要知道自动生成的 servlet 使用哪些变量名。这些内容需要从规范中获得。_jspService 自动定义了 8 个局部变量，有时也称为是“隐含对象”。这些变量并没有什么特别，它们只不过是局部变量的名称。注意，是局部变量，不是常量，也不是 JSP 保留字，并没有什么特殊。因此，如果您编写的代码不属于 _jspService 方法，那么就不能使用这些变量。特别地，由于 JSP 声明产生 _jspService 方法之外的代码，因此在声明中不能访问这些变量。类似地，在 JSP 页面调用的实用工具类中也不能使用它们。如果您需要某个独立的方法可以访问这些变量，那么就需要按照您在 Java 代码中经常采用的方式：传递该变量。

可用的变量是 request, response, out, session, application, config, pageContext 和 page。每个变量的详细信息随后给出。还有一个附加变量 exception，但仅仅在错误页面中才可用。这个变量在第 12 章介绍 errorPage 和 isErrorPage 属性的节中进行论述。

- Request

这个变量是与请求相关联的 HttpServletRequest。您可以通过它访问请求的参数、请求的类型(例如 GET 或 POST)和输入的 HTTP 报头(例如 cookie)。

- Response

这个变量是与发往客户的响应相关联的 HttpServletResponse。由于输出流(参见 out)

一般都会缓冲，在 JSP 页面的主体内设置 HTTP 状态代码和响应报头一般是合法的，尽管在已经有输出送往客户的情况下，servlet 中不允许设置报头和状态代码。但是，如果您将缓冲关闭(参见第 12 章中的 buffer 属性)，您必须在提供任何输出之前设置状态代码和报头。

- **Out**

这个变量是用来将输出发送到客户程序的 Writer。但是，为了在 JSP 页面中的各个地方都可以设置响应报头，此处使用的 out 不是标准的 PrintWriter，它对输出的内容进行缓冲(JspWriter 类型)。可以使用 page 指令(参见第 12 章)的 buffer 属性调整缓冲区的大小。由于 JSP 表达式会被自动放入到输出流中，从而很少需要直接引用 out，故而 out 变量几乎仅仅用在 scriptlet 中。

- **Session**

这个变量是与请求相关联的 HttpSession 对象。回顾一下，JSP 中会话是自动创建的，因此即使不存在对输入会话的引用，也存在这个变量。例外的情况是，如果使用 page 指令(第 12 章)的 session 属性禁用自动会话跟踪，则不存在这个变量。并且，这种情况下对 session 变量的引用在 JSP 页面转换成 servlet 时会引起错误。有关会话跟踪和 HttpSession 类的一般信息，参见第 9 章。

- **Application**

这个变量和 getServletContext 返回的类型相同，都为 ServletContext。servlet 和 JSP 页面可以在 ServletContext 对象中(而非实例变量中)存储持续性数据。ServletContext 拥有 setAttribute 和 getAttribute 方法，使用它们可以存储与指定键关联的任意数据。将数据存储在实例变量中或是存储在 ServletContext 中，二者之间的差异是：ServletContext 由 Web 应用中所有的 servlet 和 JSP 页面共享，而实例变量只能由存储数据的同一个 servlet 使用。

- **Config**

这个变量是该页的 ServletConfig 对象。原则上，您可以使用它来读取初始化参数，但在实践中，初始化参数在 jspInit 中读取，而非 _jspService。

- **PageContext**

JSP 引入一个名为 PageContext 的类，通过它可以访问页面的许多属性。PageContext 类拥有 getRequest, getResponse, getOut, getSession 等方法。pageContext 变量存储与当前页面相关联的 PageContext 对象的值。如果方法或构造函数需要访问多个与页面相关的对象，传递 pageContext 要比传递多个对 request, response, out 等的独立引用更容易。

- **Page**

这个变量不过是 this 的同义词，没有什么太大的用处。创建它是为了在脚本语言还不是 Java 的时代用作占位符。

11.13 JSP 表达式、scriptlet 和声明的比较

本节包含几个类似的例子，每个都生成 1~10 的随机整数。它们阐明 3 种 JSP 脚本元

素的典型应用方式之间的不同。所有的页面都使用清单 11.13 中定义的 randomInt 方法。

清单 11.13 RanUtilities.java

```
package coreservlets; // Always use packages!!

/** Simple utility to generate random integers.

public class RanUtilities {

    /** A random int from 1 to range (inclusive). */

    public static int randomInt(int range) {
        return(1 + ((int)(Math.random() * range)));
    }

    /** Test routine. Invoke from the command line with
     * the desired range. Will print 100 values.
     * Verify that you see values from 1 to range (inclusive)
     * and no values outside of that interval.
    */

    public static void main(String[] args) {
        int range = 10;
        try {
            range = Integer.parseInt(args[0]);
        } catch(Exception e) { // Array index or number format
            // Do nothing: range already has default value.
        }
        for(int i=0; i<100; i++) {
            System.out.println(randomInt(range));
        }
    }
}
```

11.13.1 示例 1：JSP 表达式

第一个例子的目标是输出一个项目列表，列出 1~10 的 5 个随机整数。由于本页的结构固定，且对于 randomInt 方法，我们使用单独的辅助类，因而 JSP 表达式就足够了。清单 11.14 给出相应的代码，图 11.11 给出典型的结果。

清单 11.14 RandomNums.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random Numbers</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Random Numbers</H1>
<UL>
    <LI><%= coreservlets.RanUtilities.randomInt(10) %>
    <LI><%= coreservlets.RanUtilities.randomInt(10) %>
```

```

<LI><%= coreservlets.RanUtilities.randomInt(10) %>
<LI><%= coreservlets.RanUtilities.randomInt(10) %>
<LI><%= coreservlets.RanUtilities.randomInt(10) %>
</UL>
</BODY></HTML>

```

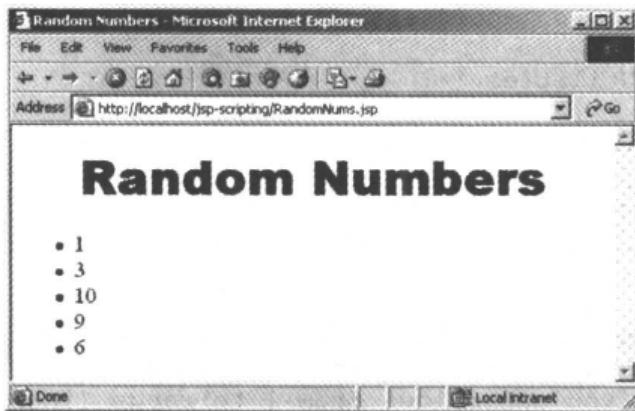


图 11.11 RandomNums.jsp 的结果。页面重新载入后会显示不同的值。

11.13.2 示例 2：JSP scriptlet

第二个例子的目标是生成一个 1~10 项的列表(随机选择)，每一项是 1~10 中的一个数字。由于列表的项数是动态的，因此需要使用 JSP scriptlet。但是，我们到底是用含有循环的单个 scriptlet 输出数字呢？还是应该使用 11.9 节中描述的悬挂花括号方式呢？在此应该做何选择并不明显：第一种方案产生更为精确的结果，但第二种方案将元素暴露给 Web 开发人员，他们可以修改项目列表的类型或插入附加的格式化元素。因此，我们介绍这两种方案。清单 11.15 列出第一种方案(单循环，其中使用预定义的 out 变量)，清单 11.16 列出第二种方案(将“静态”HTML 内容捕获到循环中)。图 11.12 和图 11.13 展示出一些典型的结果。

清单 11.15 RandomList1.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random List (Version 1)</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Random List (Version 1)</H1>
<UL>
<%
int numEntries = coreservlets.RanUtilities.randomInt(10);
for(int i=0; i<numEntries; i++) {
    out.println("<LI>" + coreservlets.RanUtilities.randomInt(10));
}
%>
</UL>
</BODY></HTML>

```

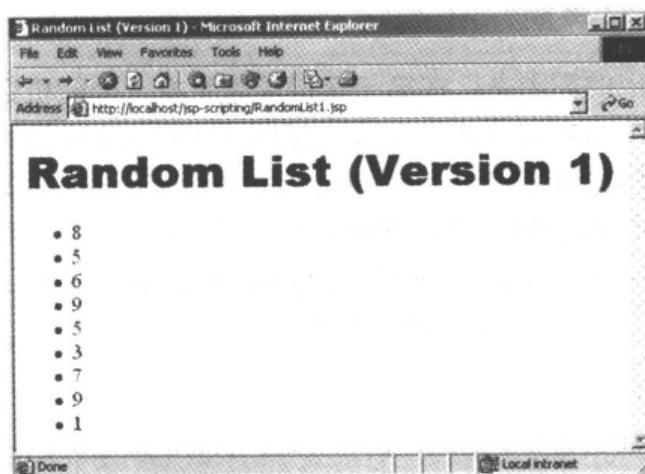


图11.12 RandomList1.jsp的结果。页面重新载入后会显示不同的值(和不同数目的列表项)

清单11.16 RandomList2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random List (Version 2)</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Random List (Version 2)</H1>
<UL>
<%>
int numEntries = coreservlets.RanUtilities.randomInt(10);
for(int i=0; i<numEntries; i++) {
<% } %>
<LI><%= coreservlets.RanUtilities.randomInt(10) %>
</UL>
</BODY></HTML>
```

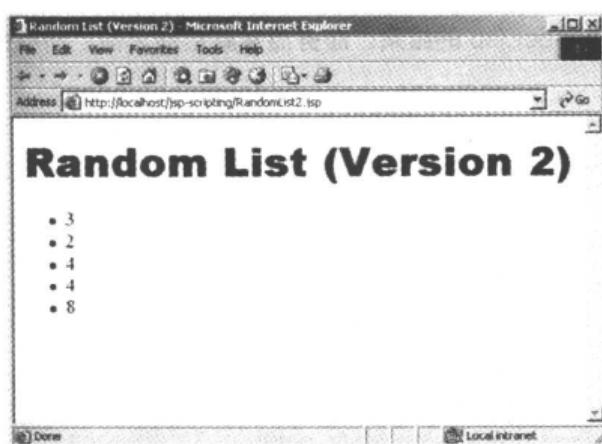


图11.13 RandomList2.jsp的结果。页面重新载入后会显示不同的值(和不同数目的列表项)

11.13.3 示例 3：JSP 声明

第三个例子中，相应的需求是：在第一次请求时生成一个随机数，然后向所有的用户显示同一数字，直至服务器重启为止。实例变量(字段)是完成这种持续性的自然之选。原因是实例变量只在对象构建时初始化，且 servlet 只构建一次，然后保存在内存中处理不同的请求，并不为每个请求分配新的实例。JSP 表达式和 scriptlet 只处理 _jspService 方法内的代码，因此它们不适合于这种情况。取而代之的是，此处需要使用 JSP 声明。清单 11.17 给出了相应的代码，图 11.14 展示出典型的结果。

清单 11.17 SemiRandomNumber.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Semi-Random Number</TITLE>
<LINK REL=STYLESCHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%!
private int randomNum = coreservlets.RanUtilities.randomInt(10);
%>
<H1>Semi-Random Number:<BR><%= randomNum %></H1>
</BODY>
</HTML>
```

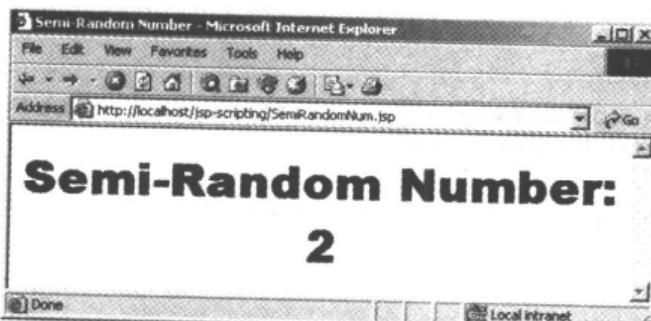


图 11.14 SemiRandomNumber.jsp 的结果。服务器重新启动之前，所有的客户都看到同样的结果

第 12 章 控制所生成的 servlet 的结构： JSP page 指令

本章的主题：

- page 指令的用途
- 指定导入类
- 指定页面的 MIME 类型
- 生成 Excel 电子表格
- 会话的共享
- 设置输出缓冲区的大小和行为
- 指定处理 JSP 错误的页面
- 控制线程的行为
- 使用指令的 XML 兼容语法

JSP 指令 (directive) 影响由 JSP 页面生成的 servlet 的整体结构。下面的模板给出指令的两种可能形式。属性值两边的双引号可以替换为单引号，但引号标记不能完全省略。如果要在属性值中使用引号，则要在它们之前添加反斜杠，'使用\'，"使用\"。

```
<% directive attribute="value" %>  
  
<% directive attribute1="value1"  
           attribute2="value2"  
           ...  
           attributeN="valueN" %>
```

在 JSP 中，主要有 3 种类型的指令：page，include 和 taglib。page 指令允许我们通过类的导入、servlet 超类的定制、内容类型的设置、以及诸如此类的事物来控制 servlet 的结构。page 指令可以放在文档中的任何地方；本章的主题就是它的应用。第二个指令，include，允许我们在 JSP 文件转换到 servlet 时，将一个文件插入到 JSP 页面中。include 指令应该放置在文档中希望插入文件的地方；第 13 章论述这部分内容。第三个指令，taglib，定义自定义的标记标签；本章第二卷对它进行长篇详细的论述，其中有好几章的内容都是关于定制标签库。

page 指令可以定义下面这些大小写敏感的属性（大致按照使用的频率列出）：import，contentType，pageEncoding，session，isELIgnored（只限 JSP 2.0），buffer，autoFlush，info，errorPage，isErrorPage，isThreadSafe，language 和 extends。本章随后的部分将逐一说明这些属性。

12.1 import 属性

我们可以使用 page 指令的 import 属性指定 JSP 页面转换成的 servlet 应该输入的包。

如 11.3 节所述, 以及图 12.1 所示, 使用独立的实用工具类可以使动态代码的编写、维护、调试、测试和重用更容易。

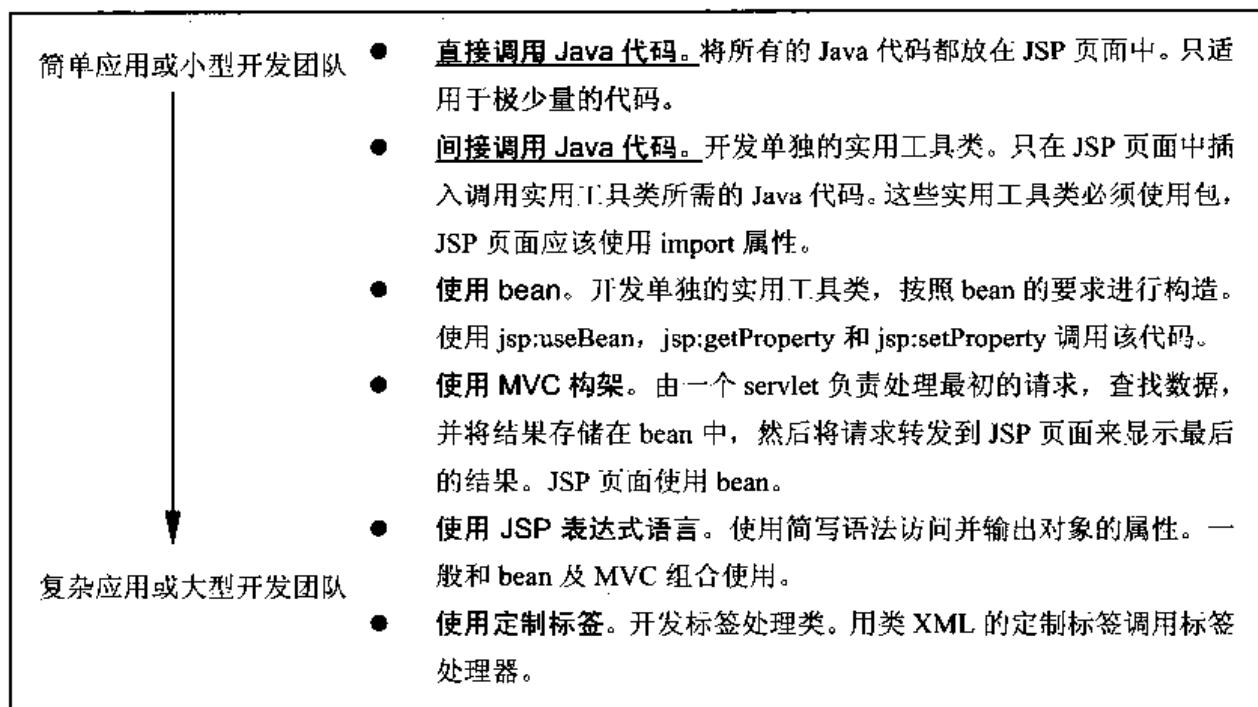


图 12.1 JSP 中调用动态代码的策略

在使用实用工具类时, 切记必须将它们放在包中。首先, 由于包有助于防止命名冲突, 因而, 包对于任何大型的项目都是一项好的策略。在 JSP 中, 包是绝对必需的。原因是, 如果没有使用包, 系统则认为您所引用的类与当前类在同一个包中。例如, 假定一个 JSP 页面包含下面的 scriptlet:

```
<% Test t = new Test(); %>
```

在此, 如果 Test 在某个输入包中, 则没有歧义。但是, 如果 Test 不在包中, 或者页面没有明确地导入 Test 所属的包, 那么系统将会认为 Test 就在这个自动生成的 servlet 所在的包中。但问题是自动生成的 servlet 所在的包是未知的! 服务器在创建 servlet 时, 常常会根据 JSP 页面所在的目录来决定它的包。别的服务器可能使用其他不同的方式。因此, 不能指望不使用包的类能够正常工作。对于 bean 也同样如此(第 14 章), 因为 bean 不过是遵循某些简单命名约定和结构约定的类。

核心方法

一定要将您的实用工具类和 bean 放在包中。

默认情况下, servlet 导入 java.lang.* , javax.servlet.* , javax.servlet.jsp.* , javax.servlet.http.* , 也许还包括一些服务器特有的包。编写 JSP 代码时, 绝不要依靠任何自动导入的服务器特有类。这样做会使得您的代码不可移植。

使用 import 属性时, 可以采用下面两种形式:

```
<%@ page import="package.class" %>
<%@ page import="package.class1,...,package.classN" %>
```

例如, 下面的指令表示: java.util 包中的所有类在使用时无需给出明确的包标识符。

```
<%@ page import="java.util.*" %>
```

import 是 page 的属性中惟一允许在同一文档中多次出现的属性。尽管 page 指令可以出现在文档中的任何地方, 但一般不是将 import 语句放在文档顶部附近, 就是放在相应的包首次使用之前。

要注意, 尽管 JSP 页面在服务器的常规 HTML 目录中, 但您所编写的由 JSP 页面使用的类必须放置在特殊的 Java 代码目录中 (例如, .../WEB-INF/classes/directory MatchingPackageName)。这些目录的相关信息参见 2.10 节和 2.11 节。

例如, 清单 12.1 给出一个页面, 它例举出前一章中的 3 种脚本元素的具体应用。该页面用到不属于标准 JSP 导入列表中的 3 个类: java.util.Date, coreservlets.CookieUtilities (参见 8.3 节) 和 coreservlets.LongLivedCookie (参见 8.4 节)。因而, 为了简化对这些类的引用, 这个 JSP 页面使用:

```
<%@ page import="java.util.*,coreservlets.*" %>
```

图 12.2 和图 12.3 给出一些典型的结果。

清单 12.1 ImportAttribute.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>The import Attribute</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>The import Attribute</H2>
<%-- JSP page Directive --%>
<%@ page import="java.util.*,coreservlets.*" %>
<%-- JSP Declaration --%>
<%!
private String randomID() {
    int num = (int)(Math.random()*10000000.0);
    return("id" + num);
}
private final String NO_VALUE = "<I>No Value</I>";
%>
<%-- JSP Scriptlet --%>
<%
String oldID =
    CookieUtilities.getCookieValue(request, "userID", NO_VALUE);
if (oldID.equals(NO_VALUE)) {
    String newID = randomID();
    Cookie cookie = new LongLivedCookie("userID", newID);
    response.addCookie(cookie);
}
%>
<%-- JSP Expressions --%>
```

```
This page was accessed on <%= new Date() %> with a userID
cookie of <%= oldID %>.
</BODY></HTML>
```

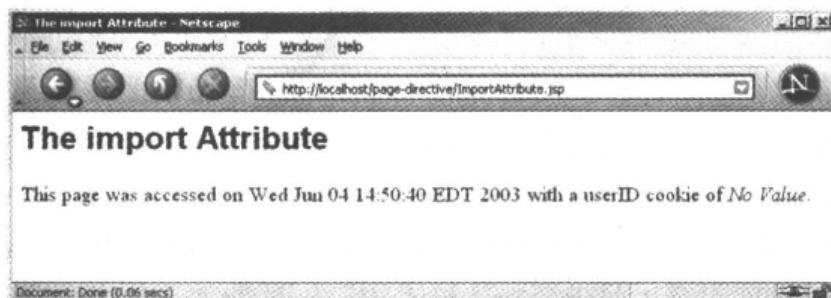


图 12.2 对 ImportAttribute.jsp 的首次访问

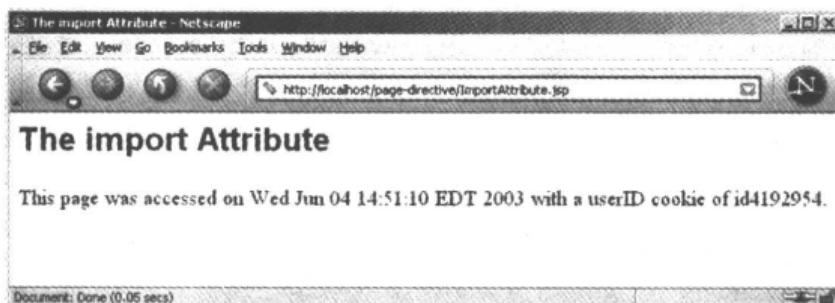


图 12.3 后续请求中对 ImportAttribute.jsp 的访问

12.2 contentType 和 pageEncoding 属性

`contentType` 属性设置 Content-Type 响应报头，标明即将发送到客户程序的文档的 MIME 类型。有关 MIME 类型的更多信息，参见 7.2 节中的表 7.1。

使用 `contentType` 属性时，可以采用下面两种形式：

```
<%@ page contentType="MIME-Type" %>
<%@ page contentType="MIME-Type; charset=Character-Set" %>
```

例如，指令

```
<%@ page contentType="application/vnd.ms-excel" %>
```

和下面的 scriptlet 所起到的作用基本相同。

```
<% response.setContentType("application/vnd.ms-excel"); %>
```

两种形式的第一点不同是，`response.setContentType` 使用明确的 Java 代码（这是一些开发人员力图避免使用的方式），而 `page` 指令只用到 JSP 语法。第二点不同是，指令被特殊处理，它们不是在出现的位置直接成为 `_jspService` 代码。这意味着 `response.setContentType` 能够有条件地调用，而 `page` 指令不能。条件性地设置内容的类型主要用在同一内容能够以多种不同的形式进行显示的情况下——具体的例子参见下一节。

不同于常规 servlet（默认的 MIME 类型为 `text/plain`），JSP 页面的默认 MIME 类型是 `text/html`（默认字符集为 ISO-8859-1）。因此，如果 JSP 页面以 Latin 字符集输出 HTML，

则根本无需使用 `contentType`。如果希望同时更改内容的类型和字符集，可以使用下面的语句：

```
<%@ page contentType="someMimeType; charset=someCharacterSet" %>
```

但是，如果只想更改字符集，使用 `pageEncoding` 属性更为简单。例如，日语 JSP 页面可以使用下面的语句：

```
<%@ page pageEncoding="Shift_JIS" %>
```

生成 Excel 电子表格

清单 12.2 给出的 JSP 页面使用 `contentType` 属性以及用制表符分隔的数据生成 Excel 输出。要注意，`page` 指令和注释都在底部，这样行尾的回车符不会显示在 Excel 文档中。

(要注意：JSP 不忽略空格——JSP 一般生成 HTML，浏览器会忽略 HTML 中的大部分空格，但 JSP 自身会维护空格并将它发送到客户端。)图 12.4 展示出在安装有 Microsoft Office 的系统上，Internet Explorer 显示的结果。

清单 12.2 Excel.jsp

```
First    Last      Email Address
Marty   Hall      hall@coreservlets.com
Larry   Brown     brown@coreservlets.com
Steve   Balmer    balmer@ibm.com
Scott   McNealy   mcnealy@microsoft.com
<%@ page contentType="application/vnd.ms-excel" %>
<%-- There are tabs, not spaces, between columns. --%>
```

The screenshot shows a Microsoft Internet Explorer window displaying an Excel spreadsheet. The address bar shows the URL: http://localhost/page-directive/Excel.jsp. The spreadsheet contains five rows of data, each with three columns: First Name, Last Name, and Email Address. The data is as follows:

	A	B	C	D	E	F	G
1	First	Last	Email Address				
2	Marty	Hall	hall@coreservlets.com				
3	Larry	Brown	brown@coreservlets.com				
4	Steve	Balmer	balmer@ibm.com				
5	Scott	McNealy	mcnealy@microsoft.com				
6							

图 12.4 Internet Explorer 中的 Excel 文档（Excel.jsp）

12.3 条件性地生成 Excel 电子表格

用 JSP 生成非 HTML 内容时，大多数情况都预先知道内容的类型。`page` 指令的 `contentType` 属性比较适合于这些情况：它不需要明确的 Java 语法，且可以出现在页面的任何地方。

但是，有时您也可能希望构建相同的内容，但根据情况的不同来改变列出的内容类型。例如，许多字处理和桌面出版系统能够输入 HTML 页面。因此，我们可以通过控制所发送的内容类型，使该页面出现在出版系统或浏览器中。类似地，Microsoft Excel 能够输入 HTML

中用 TABLE 标签表示的表格。根据这种功能，我们得出一种按照用户意愿返回 HTML 或 Excel 内容的简单方法，即：只使用 HTML 表格，当用户期望在 Excel 中查看结果时将内容类型设为 application/vnd.ms-excel。

遗憾的是，这种方式揭示出 page 指令的一项小的不足：属性的值不能在运行期间计算得出，也不能将 page 指令像模板文本一样条件性地插入到输出中。因而，下面的尝试不管 checkUserRequest 方法的结果如何，都会产生 Excel 内容。

```
<% boolean usingExcel = checkUserRequest(request); %>
<% if (usingExcel) { %>
<%@ page contentType="application/vnd.ms-excel" %>
<% } %>
```

幸运的是，根据条件设置内容的类型这个问题有一个简单的解决方案：只需使用 scriptlet 和常规的 servlet 方式——response.setContentType，如下面的片断所示：

```
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
    response.setContentType("application/vnd.ms-excel");
}
%>
```

例如，我们曾从事过一个项目，这个项目向授权用户显示财务（预算）信息。如果用户仅仅想要查看这些数据，则应该将它们显示在常规 Web 页面的表格中，如果用户希望将它放入到报告中时，则应放置在 Excel 电子表格中。在我们刚加入到这个项目中时，针对每个任务都有两套完全分离的代码。我们将它改为每种方式下都构建同样的 HTML 表格，且只改动内容的类型。它真的奏效。

清单 12.3 给出的页面就使用这种方式；图 12.5 和图 12.6 展示出相应的结果。当然，在实际的应用中，数据几乎肯定会来自于数据库。为了简单起见，我们在此使用静态的值。至于如何在 servlet 和 JSP 页面中与关系型数据库进行对话，请参见第 17 章。

清单 12.3 ApplesAndOranges.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Comparing Apples and Oranges</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<CENTER>
<H2>Comparing Apples and Oranges</H2>
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
    response.setContentType("application/vnd.ms-excel");
}
%>
<TABLE BORDER=1>
<TR><TH></TH>          <TH>Apples</TH>Oranges
```

```

<TR><TH>First Quarter <TD>2307 <TD>4706
<TR><TH>Second Quarter<TD>2982 <TD>5104
<TR><TH>Third Quarter <TD>3011 <TD>5220
<TR><TH>Fourth Quarter<TD>3055 <TD>5287
</TABLE>
</CENTER></BODY></HTML>

```

	Apples	Oranges
First Quarter	2307	4706
Second Quarter	2982	5104
Third Quarter	3011	5220
Fourth Quarter	3055	5287

图 12.5 ApplesAndOranges.jsp 的默认结果是 HTML 内容

	Apples	Oranges
First Quarter	2307	4706
Second Quarter	2982	5104
Third Quarter	3011	5220
Fourth Quarter	3055	5287

图 12.6 为 ApplesAndOranges.jsp 指定 format=excel 产生 Excel 表格

12.4 session 属性

session 属性控制页面是否参与 HTTP 会话。使用这个属性时，可以采用下面两种形式：

```

<%@ page session="true" %> <%-- Default --%>
<%@ page session="false" %>

```

true 值（默认）表示，如果存在已有会话，则预定义变量 session（类型为 HttpSession）应该绑定到现有的会话；否则，则创建新的会话并将其绑定到 session。false 值表示不自动创建会话，在 JSP 页面转换成 servlet 时，对变量 session 的访问会导致错误。

对于高流量的网站，使用 session="false" 可以节省大量的服务器内存。但要注意，session="false" 并不禁用会话跟踪——它只是阻止 JSP 页面为那些尚不拥有会话的用户创建新的会话。由于会话是针对用户，不是针对页面，所以，关闭某个页面的会话跟踪没有任何益处，除非有可能在同一客户会话中访问到的相关页面都关闭会话跟踪。

12.5 isELIgnored 属性

`isELIgnored` 属性控制的是：忽略 (`true`) JSP 2.0 表达式语言 (EL)，还是进行正常的求值 (`false`)。这是 JSP 2.0 新引入的属性；在只支持 JSP 1.2 及早期版本的服务器中，使用这项属性是不合法的。这个属性的默认值依赖于 Web 应用所使用的 `web.xml` 的版本。如果您的 `web.xml` 指定 `servlet 2.3` (对应 JSP 1.2) 或更早版本，默认值为 `true` (但变更默认值依旧是合法的，JSP 2.0 兼容的服务器中都允许使用这项属性，不管 `web.xml` 的版本如何)。如果您的 `web.xml` 指定 `servlet 2.4` (对应 JSP 2.0) 或之后的版本，那么默认值为 `false`。使用这个属性时，可以采用下面两种形式：

```
<%@ page isELIgnored="false" %>
<%@ page isELIgnored="true" %>
```

JSP 2.0 引入一种简洁的表达式语言，在 JSP 页面中可以用它来访问请求参数、cookie、HTTP 报头、bean 属性和 Collection 的元素。详细信息参见第 16 章。JSP EL 中，表达式的形式为 `${expression}`。一般地，这些表达式是方便的。但是，JSP 1.2 页面中也可能偶然会含有形如 `${...}` 的字符串，这种情况下会发生什么呢？在 JSP 2.0 中，这可能会引发问题。`isELIgnored="true"` 能够阻止这些问题。

12.6 buffer 和 autoFlush 属性

`buffer` 属性指定 `out` 变量 (类型为 `JspWriter`) 使用的缓冲区的大小。使用这个属性时，可以采用下面两种形式：

```
<%@ page buffer="sizekb" %>
<%@ page buffer="none" %>
```

服务器实际使用的缓冲区可能比指定的更大，但不会小于指定的大小。例如，`<%@ page buffer="32kb" %>` 表示应该对文档的内容进行缓存，除非累积至至少为 32KB、页面完成或明确地对输出执行清空 (例如使用 `response.flushBuffer`)，否则不将文档发送给客户。默认的缓冲区大小与服务器相关，但至少 8KB。如果要将缓冲功能关闭，应该十分小心；这样做要求设置报头或状态代码的 JSP 元素都要出现在文件的顶部，位于任何 HTML 内容之前。另一方面，有时输出内容的每一行都需要较长的生成时间，此时禁用缓冲或使用小缓冲区会更有效率；这样，用户能够在每一行生成之后立即看到它们，而不是等待更长的时间看到成组的行。

`autoFlush` 属性控制当缓冲区充满之后，是应该自动清空输出缓冲区 (默认)，还是在缓冲区溢出后抛出一个异常 (`autoFlush="false"`)。使用这个属性时，可以采用下面两种形式：

```
<%@ page autoFlush="true" %> <%-- Default --%>
<%@ page autoFlush="false" %>
```

在 `buffer="none"` 时，`false` 值是不合法的。如果客户程序是常规的 Web 浏览器，那么 `autoFlush="false"` 的使用极为罕见。但是，如果客户程序是定制应用程序，您可能希望确保

应用程序要么接收到完整的消息，要么根本没有消息。false 值还可以用来捕获产生过多数据的数据库查询，但是，一般说来，将这些逻辑放在数据访问代码中（而非表示代码）要更好一些。

12.7 info 属性

info 属性定义一个可以在 servlet 中通过 getServletInfo 方法获取的字符串。使用 info 属性时，采用下面的形式：

```
<%@ page info="Some Message" %>
```

12.8 errorPage 和 isErrorPage 属性

errorPage 属性用来指定一个 JSP 页面，由该页面来处理当前页面中抛出但未被捕获的任何异常（即类型为 Throwable 的对象）。它的应用方式如下：

```
<%@ page errorPage="Relative URL" %>
```

指定的错误页面可以通过 exception 变量访问抛出的异常。

isErrorPage 属性表示当前页是否可以作为其他 JSP 页面的错误页面。使用 isErrorPage 属性时，可以采用下面两种形式。

```
<%@ page isErrorPage="true" %>
<%@ page isErrorPage="false" %> <%-- Default --%>
```

举个例子，清单 12.4 给出的 JSP 页面基于距离和时间参数计算速度。该页面没有检查输入参数是否缺失或异常，因而在运行期间很容易发生问题。但是，这个页面指定 SpeedErrors.jsp（清单 12.5）来处理 ComputeSpeed.jsp 中发生的错误，因而用户不会接收到典型的传统 JSP 错误消息。要注意，SpeedErrors.jsp 放置在 WEB-INF 目录中。由于服务器禁止客户直接访问 WEB-INF，这种安排能够阻止客户无意间直接访问 SpeedErrors.jsp。发生错误时，SpeedErrors.jsp 由服务器访问，而非由客户程序访问；这类错误页面不会生成 response.sendRedirect 调用，客户只看到最初请求页面的 URL，看不到错误页面的 URL。

图 12.7 和图 12.8 分别展示出接收到有效或错误输入参数时的结果。

要注意，errorPage 属性指定页面专用的错误页面。如果要为整个 Web 应用指定错误页面，或者为应用中不同类型的错误指定错误处理页面，则需要使用 web.xml 中的 error-page 元素。详细信息参见本书第二卷。

清单 12.4 ComputeSpeed.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Computing Speed</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
```

```

<%@ page errorPage="/WEB-INF/SpeedErrors.jsp" %>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
    Computing Speed</TABLE>
<%!
// Note lack of try/catch for NumberFormatException if
// value is null or malformed.

private double toDouble(String value) {
    return(Double.parseDouble(value));
}
%>
<%
double furlongs = toDouble(request.getParameter("furlongs"));
double fortnights = toDouble(request.getParameter("fortnights"));
double speed = furlongs/fortnights;
%>
<UL>
    <LI>Distance: <%= furlongs %> furlongs.
    <LI>Time: <%= fortnights %> fortnights.
    <LI>Speed: <%= speed %> furlongs per fortnight.
</UL>
</BODY></HTML>

```

清单12.5 SpeedErrors.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Error Computing Speed</TITLE>
<LINK REL=STYLESCHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ page isErrorPage="true" %>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
    Error Computing Speed</TABLE>
<P>
ComputeSpeed.jsp reported the following error:
<I><%= exception %></I>. This problem occurred in the
following place:
<PRE>
<%@ page import="java.io.*" %>
<% exception.printStackTrace(new PrintWriter(out)); %>
</PRE>
</BODY></HTML>

```

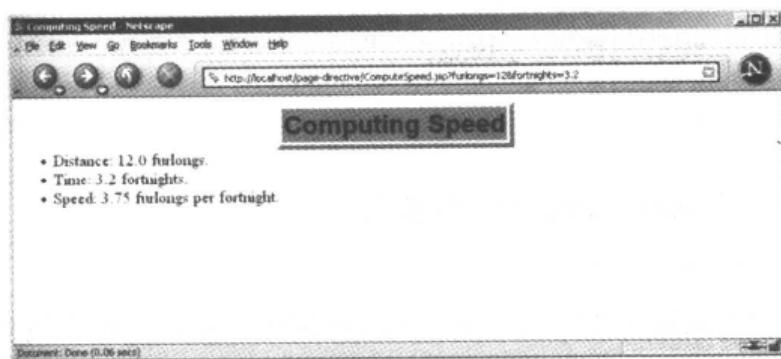


图 12.7 ComputeSpeed.jsp 接收到合法值时的结果

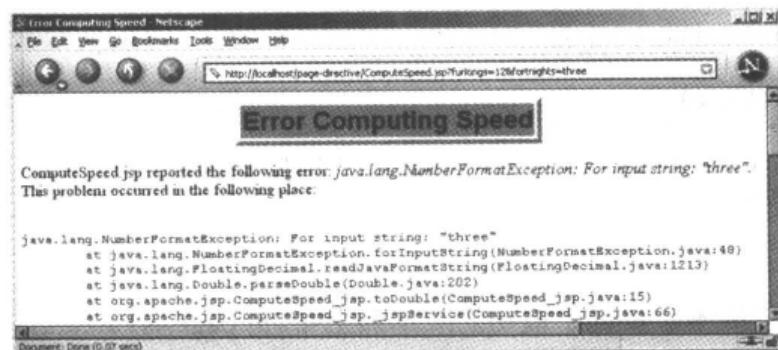


图 12.8 ComputeSpeed.jsp 接收到不合法值时的结果。注意: 地址栏中显示的是 ComputeSpeed.jsp 的 URL, 不是 SpeedErrors.jsp 所对应的 URL

12.9 isThreadSafe 属性

isThreadSafe 属性控制由 JSP 页面生成的 servlet 是允许并行访问（默认），还是同一时间不允许多个请求访问单个 servlet 实例（isThreadSafe="false"）。使用 isThreadSafe 属性时，可以采用下面两种形式：

```
<%@ page isThreadSafe="true" %> <%-- Default --%>
<%@ page isThreadSafe="false" %>
```

遗憾的是，阻止并发访问的标准机制是实现 SingleThreadModel 接口（3.7 节）。尽管在早期推荐使用 SingleThreadModel 和 isThreadSafe="false"，但最近的经验表明 SingleThreadModel 的设计很差，使得它基本上毫无用处。因而，应该避免使用 isThreadSafe，采用显式的同步措施取而代之。

警告

不要使用 isThreadSafe，应使用显式的同步措施。

为什么 isThreadSafe="false" 是一种坏的思想？为帮助理解，请考虑下面这段计算用户 ID 的代码片断。它并非线程安全，因为线程可能恰好会在读取 idNum 之后，但在更新 idNum 之前被别的线程抢先，从而造成两个用户拥有相同的用户 ID。

```
<%! private int idNum = 0; %>
```

```
<%
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
%>
```

这段代码应该使用 `synchronized` 块。结构如下：

```
synchronized(someObject) { ... }
```

它表示一旦某个线程进入这个代码块，则在该线程退出之前，没有其他线程能够进入同一代码块（或任何其他用同一个对象引用标记的块）。因而，前面的片断应该按照下面的方式编写：

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
    String userID = "userID" + idNum;
    out.println("Your ID is " + userID + ".");
    idNum = idNum + 1;
}
%>
```

采用显式同步的版本，要优于最初的版本加上`<%@ page isThreadSafe="false" %>`的做法，原因有二。

首先，如果页面被频繁访问，显式同步版本可能拥有更好的性能。理由是大部分 JSP 页面的瓶颈都不是 CPU，而是 I/O。因此，在系统等待 I/O 的同时（例如等待数据库的响应、EJB 调用的结果或者通过网络向用户发送输出时），应该做一些其他的事情。由于大多数服务器通过对请求进行排队，对它们进行顺序处理的方式实现 `SingleThreadModel`，因此，如果使用这种方式，高流量的 JSP 页面可能会慢很多。

更为不利的是，使用 `SingleThreadModel` 的版本甚至可能得不出正确的答案！服务器可以通过生成 `servlet` 的实例池来实现 `SingleThreadModel`（而不采取对请求进行排队的方式），只要没有实例被同时调用就可以了。当然，这会使得利用字段保存持续性数据的功用完全失效，因为每个实例会拥有不同的字段（实例变量），多个用户依旧有可能会取得相同的用户 ID。将 `idNum` 字段定义为 `static` 也不能解决这个问题：每个 `servlet` 实例的 `this` 引用都会不同，因此这种保护也会失效。

基本上，这些问题很难处理。放弃吧！忘掉 `SingleThreadModel` 和 `isThreadSafe="false"`，转而对您的代码采取明确的同步措施吧！

12.10 extends 属性

`extends` 属性指定 JSP 页面所生成的 `servlet` 的超类（superclass）。它采用下面的形式：

```
<%@ page extends="package.class" %>
```

这个属性一般为开发人员或提供商保留，由他们对页面的运作方式做出根本性的改变（如添加个性化特性）。一般人应该避免使用这个属性，除非引用由服务器提供商专为这种目的提供的类。

12.11 language 属性

从某种角度讲，language 属性的作用是指定页面使用的脚本语言，如下所示：

```
<%@ page language="cobol" %>
```

就现在来说，由于 Java 既是默认选择，也是惟一合法的选择，所以没必要再去关心这个属性。

12.12 指令的 XML 语法

如果您正在编写 XML 兼容的 JSP 页面，只要不在同一页面中混合使用 XML 语法和传统的语法，就可以使用指令的 XML 兼容语法。这些结构采用下面的形式：

```
<jsp:directive.directiveType attribute="value" />
```

例如：

```
<%@ page import="java.util.*" %>
```

的 XML 等价物是：

```
<jsp:directive.page import="java.util.*" />
```

第 13 章 在 JSP 页面中包含文件和 applet

本章的主题：

- 使用 `jsp:include` 在请求期间包含页面
- 使用`<%@ include ... %>`(`include` 指令)在页面转换期间包含文件
- 为什么 `jsp:include` 一般要好于 `include` 指令
- 使用 `jsp:plugin` 包含 Java 插件的 applet

在 JSP 中，主要有 3 种功能可以将外部内容包含到 JSP 文档中：

- **`jsp:include` 动作。** `jsp:include` 动作允许我们在请求期间将其他页面的输出包含进来。它的主要优点是：在被包含的页面发生更改时，无需对主页面做出修改。它的主要缺点是：它所包含的是次级页面的输出，而非次级页面的实际代码(`include` 指令)，所以，在被包含的页面中不能使用任何有可能总体上影响主页面的 JSP 构造。一般说来，它的优点要远远胜过它的缺点，这几乎注定它会比其他包含机制应用得更为普遍。`jsp:include` 的使用在 13.1 节中论述。
- **`include` 指令。** `include` 指令可以在主页面转换成 `servlet` 之前，将 JSP 代码插入其中。它的主要优点是功能强大，所包含的代码可以含有总体上影响主页面的 JSP 构造，比如字段的定义和内容类型的设定。它的主要缺点是难以维护：只要被包含的页面发生更改，就得更新主页面。`include` 指令的使用在 13.2 节论述。
- **`jsp:plugin` 动作。** 尽管本书主要论述服务器端的 Java，但同时也会照顾到客户端 Java(嵌入 Web 中的 applet)，因为客户端 Java 在实际应用中也能发挥一定的作用，尤其在公司内部的内联网上。`jsp:plugin` 元素可以将使用 Java 插件(Java Plug-in)的 applet 插入到 JSP 页面中。它的主要优点是：能够避免在 HTML 中编写冗长繁琐且易于出错的 `OBJECT` 和 `EMBED` 标签。它的主要缺点是：它适用于 applet，而 applet 的应用不太常见。`jsp:plugin` 的使用在 13.4 节中论述。

13.1 在请求期间包含页面：`jsp:include` 动作

假定您有一系列的页面，每一个都拥有同样的导航栏、联系信息或脚注。您应该怎样做呢？一种常见的“解决方案”是将相同的 HTML 片断剪切、粘贴到所有的页面中。这是一种坏的思想，因为，如果要对公共部分做出更动，必须更改所有用到它的页面。另一种通常的解决方案是使用某种服务器端包含机制，在页面被请求时将公共块插入到其中。这种通用方案是一种好的做法，但不同的服务器所采用的典型机制有所不同。`jsp:include` 是一种可移植的机制，它可以将下面列出的任何内容插入到 JSP 的输出中：

- HTML 页面的内容
- 纯文本文档的内容
- JSP 页面的输出
- `servlet` 的输出

`jsp:include` 动作在主页面被请求时，将次级页面的输出包含进来。尽管被包含页面的输出中不能含有 JSP，但这些页面可以是其他资源(这些资源有可能使用 servlet 或 JSP 创建输出)所产生的结果。也就是说，服务器按照正常的方式对指向被包含资源的 URL 进行解释，因而，这个 URL 可以是 servlet 或 JSP 页面。服务器以通常的方式运行被包含的页面，将产生的输出放入主页面中。这种行为和 RequestDispatcher 类(参见第 15 章)的 `include` 方法完全相同，在 `servlet` 希望完成这种类型的文件包含时，就是使用这个方法。

13.1.1 page 属性：指定所包含的页面

我们使用 `page` 属性指定被包含的页面，如下所示。这个属性是必不可少的；它应该是指向某种资源(我们希望将它的输出包含进来)的相对 URL。

```
<jsp:include page="relative-path-to-resource" />
```

如果相对 URL 不以斜杠开头，则将其解释为相对于主页面的位置。以斜杠开头的相对 URL 被解释为相对于 Web 应用的根目录(注意：不是相对于服务器的根目录)。举个例子，假定在 `headlines` Web 应用中有一个 JSP 页面，它所对应的 URL 是 `http://host/headlines/sports/table-tennis.jsp`。`headlines` Web 应用使用哪个目录，`table-tennis.jsp` 文件就在哪个目录的 `sports` 子目录中。接下来，考虑下面两行包含语句：

```
<jsp:include page="bios/cheng-yinghua.jsp" />  
<jsp:include page="/templates/footer.jsp" />
```

第一行语句，系统会在 `sports` 的 `bios` 子目录(即 `headlines` 应用所使用的主目录的 `sports/bios` 子子目录)中查找 `cheng-yinghua.jsp`。第二行语句，系统会在 `headlines` 应用的 `templates` 子目录中查找 `footer.jsp`，而不是在服务器根目录的 `templates` 子目录中进行查找。`jsp:include` 动作肯定不会让系统去查找当前 Web 应用以外的文件。如果记住系统如何解释以斜杠开头的 URL 有困难，可以记住这条规则：任何时候，如果是由服务器来处理它们，都按相对于当前 Web 应用进行解释；只有客户(浏览器)处理它们时，才会按相对于服务器的根目录来解释。例如，下面这行语句中的 URL：

```
<jsp:include page="/path/file" />
```

由于服务器对这个 URL 进行解析，浏览器不会看到它，因此是在当前 Web 应用的语境中进行解释。但下面这行语句中的 URL：

```
<IMG SRC="/path/file" ...>
```

因为由浏览器对它进行解析，因此会按相对于服务器的根目录进行解释。Web 应用的相关信息参见 2.11 节。

重点提示

对以斜杠开头的 URL，服务器和浏览器的解释是不同的。服务器总是相对于当前 Web 应用对它们进行解释。浏览器总是相对于服务器的根目录对它们进行解释。

最后，要注意，您可以将页面放在 WEB-INF 目录中。尽管客户对这个目录的直接访问是不允许的，但由于是服务器(而非客户程序)访问由 `jsp:include` 的 `page` 属性指出的文件，

因此，将被包含的页面放在 WEB-INF 目录中是完全可行的。实际上，我们推荐将被包含的页面放在 WEB-INF 目录中；这样可以防止客户出于偶然访问到这些页面(这种情况可能会比较糟糕，因为它们一般都不是完整的 HTML 文档)。

核心方法

如果要防止包含页面被客户单独访问，可以将它们放在 WEB-INF 目录或其子目录中。

13.1.2 XML 语法和 `jsp:include`

`jsp:include` 动作是我们看到的 JSP 构造中，第一个只有 XML 语法，没有等同“经典”语法的构造。如果您不熟悉 XML，请注意 3 件事：

- XML 元素名中可以含有冒号。

因此，不要被元素名是 `jsp:include` 吓倒。实际上，所有标准 JSP 元素的 XML 兼容版本都从 `jsp` 前缀(或命名空间)开始。

- XML 标签对大小写敏感。

在标准 HTML 中，您可以写成 BODY, body, 或 Body，都不要紧，但在 XML 中则不然。因此，一定要使用全部小写的 `jsp:include`。

- XML 标签必须显式关闭。

在 HTML 中，存在像 H1 这样的容器元素，它们拥有起始标签和结束标签(`<H1> ... </H1>`)，当然还存在像 IMG 或 HR 这样的独立元素，它们没有结束标签(`<HR>`)。

另外，HTML 规范规定某些容器元素(如 TR, P)的结束标签是可选的。但是，在 XML 中所有的元素都是容器元素，结束标签从来都是必不可少的。但是，为了方便起见，也可以将没有主体的片断，比如`<blah></blah>`，替换为`<blah/>`。因此，在使用 `jsp:include` 时，一定要包含结尾的斜杠。

13.1.3 flush 属性

除必需的 page 属性以外，`jsp:include` 还有一个次级属性：flush，如下所示。这个属性是可选的；它指定在将页面包含进来之前是否应该清空主页面的输出流(默认为 false)。但要注意，在 JSP 1.1 中，flush 是一项必需的属性，它的唯一合法值就是 true。

```
<jsp:include page="relative-path-to-resource" flush="true" />
```

13.1.4 新闻头条页面

作为 `jsp:include` 的典型应用，请考虑清单 13.1 中列出的新闻汇总页面。页面的开发人员变更文件 Item1.html 到 Item3.html(清单 13.2 到清单 13.4)中的新闻项时，不必更新主新闻页面。图 13.1 给出相应的结果。

要注意到：被包含的页面不是完整的 Web 页面。包含进来的文件可以是 HTML 文件、纯文本文件、JSP 页面、或 servlet(如果文件为 JSP 页面和 servlet，则包含进来的只是页面的输出，不是实际的代码)。但是，任何情况下，客户看到的都是合成后的结果。因此，如

果主页面和包含进来的内容中都含有诸如 DOCTYPE, BODY 等标签, 那么, 在客户看到的结果中这些标签将会出现两次, 这会使得最终的 HTML 不合法。在用到 servlet 和 JSP 时, 请务必查看生成的 HTML 源代码, 并将 URL 提交给 HTML 验证器(参见 3.5 节), 这是一个好习惯。在使用 `jsp:include` 时, 这个建议甚至更为重要, 因为初学者经常错误地将主页面和包含页面都设计成完整的 HTML 文档。

核心方法

不要将完整的 HTML 文档作为被包含页面。被包含页面中只能含有适合于出现在文件插入点处的 HTML 标签。

清单13.1 WhatsNew.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>What's New at JspNews.com</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    What's New at JspNews.com</TH>
  </TR>
<P>
Here is a summary of our three most recent news stories:
<OL>
  <LI><jsp:include page="/WEB-INF/Item1.html" />
  <LI><jsp:include page="/WEB-INF/Item2.html" />
  <LI><jsp:include page="/WEB-INF/Item3.html" />
</OL>
</BODY></HTML>
```

清单13.2 /WEB-INF/Item1.html

```
<B>Bill Gates acts humble.</B> In a startling and unexpected
development, Microsoft big wig Bill Gates put on an open act of
humility yesterday.
<A href="http://www.microsoft.com/Never.html">More details...</A>
```

清单13.3 /WEB-INF/Item2.html

```
<B>Scott McNealy acts serious.</B> In an unexpected twist,
wisecracking Sun head Scott McNealy was sober and subdued at
yesterday's meeting.
```

```
<A href="http://www.sun.com/Imposter.html">More details...</A>
```

清单13.4 /WEB-INF/Item3.html

```
<B>Larry Ellison acts conciliatory.</B> Catching his competitors
off guard yesterday, Oracle prez Larry Ellison referred to his
rivals in friendly and respectful terms.
```

```
<A href="http://www.oracle.com/Mistake.html">More details...</A>
```

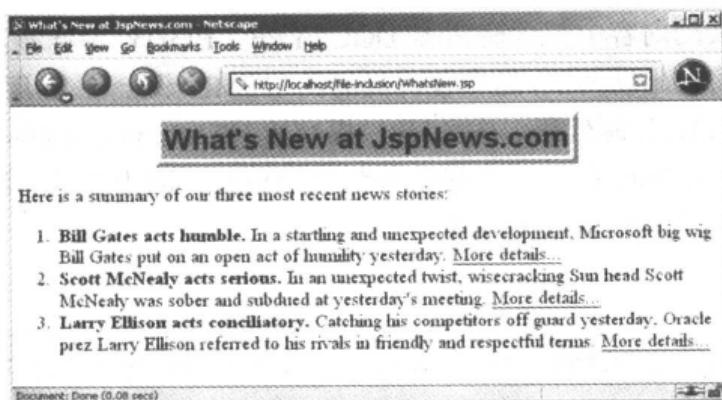


图 13.1 通过在请求期间将文件包含进来，可以在不更改主页面的情况下更新个别的文件

13.1.5 `jsp:param` 元素：增加请求参数

被包含页面与最初请求的页面使用相同的请求对象。因此，被包含页面看到的请求参数一般与主页面相同。如果希望增加或替换这些参数，可以使用 `jsp:param` 元素(它有两项属性，`name` 和 `value`)来完成。请考虑下面的代码片断：

```
<jsp:include page="/fragments/StandardHeading.jsp">
  <jsp:param name="bgColor" value="YELLOW" />
</jsp:include>
```

假定调用主页面的方式是通过 URL `http://host/path/MainPage.jsp?fgColor=RED`。下面的列表汇总了这种情况下各种 `getParameter` 调用的结果。

- 在主页面中(`MainPage.jsp`)(不管对 `getParameter` 的调用是发生在文件包含之前，还是之后)
 - ◆ `request.getParameter("fgColor")` 返回"RED"。
 - ◆ `request.getParameter("bgColor")` 返回 null。
- 在被包含页面中(`StandardHeading.jsp`)。
 - ◆ `request.getParameter("fgColor")` 返回"RED"。
 - ◆ `request.getParameter("bgColor")` 返回"YELLOW"。

如果 `jsp:param` 元素指定了主页面接收到的请求参数，那么由 `jsp:param` 指定的值仅在被包含文件中优先采用。

13.2 在页面转换期间包含文件：`include` 指令

`include` 指令可以在主 JSP 文档转换成 servlet 时(一般在它首次被访问时)，将文件包含到文档中。语法如下：

```
<%@ include file="Relative URL" %>
```

可以将 `include` 指令看作是一种预处理器：服务器将被包含文件的内容逐字节插入到主页面中，然后将产生的页面作为单个 JSP 页面进行处理。因此，`jsp:include` 和 `include` 指令之间根本性的不同在于它们被调用的时间：`jsp:include` 在请求期间被激活，而 `include` 指令在页面转换期间被激活。但是，这种差异的背后隐藏着更多不是特别直观的含意。我们将

它们汇总在表 13.1 中。

表 13.1 `jsp:include` 和 `include` 指令之间的差异

<code>jsp:include</code> 动作	<code>include</code> 指令
语法的基本形式	<code><jsp:include page="..." /></code>
包含动作的发生时间	请求期间
包含的内容	页面的输出
产生多少 servlet	两个(主页面和被包含页面都会成为独立的 servlet)
被包含页面中可否设置影响主页面的响应报头	不可以
被包含页面可否定义主页面使用的字段或方法	不可以
被包含页面发生更改时是否需要更新主页面	不需要
等同的 servlet 代码	<code>RequestDispatcher</code> 的 <code>include</code> 方法
何处论述	13.1 节
	13.2 节(本节)

`include` 指令包含的文件在页面转换期间插入进来，而非和 `jsp:include` 一样发生在请求期间，这一事实造成 `include` 指令和 `jsp:include` 在许多方面存在差异。然而，其中的两项差异十分重要：维护和能力。我们在下两小节中论述这两项内容。

13.2.1 `include` 指令的维护问题

由于包含发生在页面转换期间，因此造成的第一项差异就是：使用 `include` 指令的页面要比使用 `jsp:include` 的页面难维护得多。使用 `include` 指令(`<%@ include ... %>`)，如果包含的文件发生改变，那么，用到它的所有 JSP 页面可能都需要更新。服务器需要能够检测出 JSP 页面发生改变这种情况，并在处理接下来的请求之前，将它转换成新的 `servlet`。但遗憾的是，相关的规范只要求服务器能够检测出主页面什么时候发生了更改，并不要求它们能够检测出包含文件什么时候发生改变。虽然服务器可以支持某种机制，检测包含文件是否发生改变(并且重新编译 `servlet`)，但服务器并非必需这样做。实际上，几乎没有服务器支持类似的机制。因此，大多数服务器中，包含文件发生更改时，对于所有用到该文件的 JSP 文件，我们都得更新它们的修改日期。

这样做十分不方便；它会导致十分严重的维护问题，因此，仅当 `jsp:include` 不能满足要求时，我们才应该使用 `include` 指令。有些开发人员认为使用 `include` 指令生成的代码执行起来比使用 `jsp:include` 动作的代码更快。尽管原则上有可能的确如此，但性能上的差异很小，以致难以测量，同时，`jsp:include` 在维护上的优势十分巨大，当两种方法都可以使用时，`jsp:include` 几乎肯定是首选的方法。实际上，有些人员甚至因为 `include` 会带来维护上的繁重负担，而完全放弃使用它。这也许是一种过激反应，但是，至少，在确实需要 `include`

指令所提供的特别能力时，应该使用它。

核心方法

对于文件包含，应该尽可能地使用 `jsp:include`。仅在所包含的文件中定义了主页面要用到的字段或方法，或者所包含的文件设置了主页面的响应报头时，才应该使用 `include` 指令 (`<%@ include ... %>`)。

13.2.2 `include` 指令提供的其他能力

既然 `include` 指令产生难以维护的代码，为什么人们还要使用它呢？嗯，这就引出 `jsp:include` 和 `include` 指令的第二项差异。`include` 指令更为强大。`include` 指令允许所包含的文件中含有影响主页面的 JSP 代码，比如响应报头的设置和字段的定义。例如，假定 `snippet.jsp` 含有下面的代码：

```
<%! int accessCount = 0; %>
```

这种情况下，您可以在主页面中执行下面的任务：

```
<%@ include file="snippet.jsp" %> <%-- Defines accessCount --%>
<%= accessCount++ %>           <%-- Uses accessCount --%>
```

当然，使用 `jsp:include` 时这是不可能的，因为 `accessCount` 变量未定义；主页面不能成功地转换成 `servlet`。此外，即使主页面能够成功完成转换，不发生错误，也没有任何意义：`jsp:include` 包括的是辅助页面的输出，而 `snippet.jsp` 没有输出。

13.2.3 更新主页面

大多数服务器中，如果您使用 `include` 指令，并且更改了被包含的文件，都必须更新主页面的修改日期。某些操作系统提供相应的命令，可以直接更新修改日期，不必真的对文件进行编辑(如 Unix `touch` 命令)，但一种简单的可移植的做法是在顶级页面中写入一段 JSP 注释。当被包含的文件发生改变时更新这段注释。例如，我们可以将被包含文件的修改日期放在注释中，如下所示：

```
<%-- Navbar.jsp modified 9/1/03 --%>
<%@ include file="Navbar.jsp" %>
```

警告

如果您更改了被包含的文件，必须更新所有使用它的 JSP 文件的修改日期。

13.2.4 `include` 指令的 XML 语法

对于下面的语句：

```
<%@ include file="..." %>
```

XML 兼容的等价语句是：

```
<jsp:directive.include file="..." />
```

在使用这种形式时，主页面和被包含文件都必须完全使用 XML 兼容的语法。

13.2.5 示例：脚注重用

某种情况下，我们会采用 include 指令来替代 jsp:include，下面我们就举一个具体的例子来说明这种情况。假定有这样一个 JSP 页面：它产生含有简短脚注的 HTML 片断，其中含有访问计数和对当前页面最近访问的相关信息。清单 13.5 给出的就是这样一个页面。

在此，假定您有好几个页面都希望使用这种类型的脚注。您可以将这个脚注放在 WEB-INF(可以阻止客户直接访问这个文件)中，然后在希望使用它的页面中添加下面的语句：

```
<%@ include file="/WEB-INF/ContactSection.jsp" %>
```

清单 13.6 给出一个使用这种方式的页面；图 13.2 给出相应的结果。

“打住！”您可能会说，“是的，ContactSection.jsp 定义了一些实例变量(字段)。如果主页面用到了这些实例变量，我对你必须使用 include 指令表示赞同。但是，对于这种特定的情况，主页面并没有用到实例变量，因此应该使用 jsp:include，是不是这样？”这种想法是错误的！如果您在此使用 jsp:include，那么，所有使用这个脚注文件的页面将会显示相同的访问计数。而事实上，我们希望每个使用这个脚注的页面都维护不同的访问计数。我们不希望 ContactSection.jsp 成为单独的 servlet；我们希望 ContactSection.jsp 提供的代码成为由 JSP 页面(使用 ContactSection.jsp)产生的独立 servlet 的一部分。因此，我们需要使用 include 指令。

清单 13.5 ContactSection.jsp

```
<%@ page import="java.util.Date" %>
<%-- The following become fields in each servlet that
    results from a JSP page that includes this file. --%>
<%
private int accessCount = 0;
private Date accessDate = new Date();
private String accessHost = "<I>No previous access</I>";
%>
<P>
<HR>
This page &copy; 2003
<A HREF="http://www.my-company.com/">my-company.com</A>.
This page has been accessed <%= ++accessCount %>
times since server reboot. It was most recently accessed from
<%= accessHost %> at <%= accessDate %>.
<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
```

清单 13.6 SomeRandomPage.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Some Random Page</TITLE>
<META NAME="author" CONTENT="J. Random Hacker">
```

```

<META NAME="keywords"
      CONTENT="foo,bar,baz,quux">
<META NAME="description"
      CONTENT="Some random Web page.">
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
    Some Random Page</TABLE>
<P>
Information about our products and services.
<P>
Blah,blah,blah.
<P>
Yadda,yadda,yadda
<%@ include file="/WEB-INF/ContactSection.jsp" %>
</BODY></HTML>

```

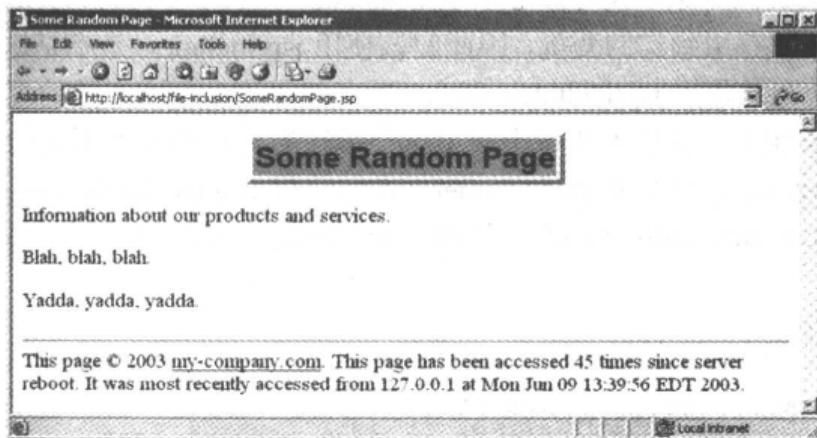


图 13.2 SomeRandomPage.jsp 的结果。它使用 include 指令，从而，它独立于任何其他使用 ContactSection.jsp 的页面来维护访问计数和最近访问页面的主机。

13.3 使用 jsp:forward 转发请求

除了可以使用 `jsp:include` 组合主页面和辅助页面的输出之外，我们还可以使用 `jsp:forward` 获取辅助页面的完整输出。例如，下面的页面随机地选择输出 `page1.jsp` 和 `page2.jsp`。

```

<% String destination;
if (Math.random() > 0.5) {
    destination = "/examples/page1.jsp";
} else {
    destination = "/examples/page2.jsp";
}
%>
<jsp:forward page="<% destination %>" />

```

如果使用 `jsp:forward`，则主页面不能含有任何输出。这就提出一个问题：那么使用 JSP

又有什么好处呢？答案是：没有任何好处！实际上，使用 JSP 对于处理这类情况反而不利，因为现实的情形会更复杂，而复杂的代码在 servlet 中要比在 JSP 页面中更容易开发和测试。我们推荐完全避免使用 `jsp:forward`。如果您希望执行类似于本示例的任务，请使用 `servlet`，由 `servlet` 调用 `RequestDispatcher` 的 `forward` 方法。详细信息参见第 15 章。

13.4 包含使用 Java 插件的 applet

Java 编程语言演化过程中的早期阶段，主要的应用领域是 `applet`(嵌入在 Web 页面中的 Java 程序，由 Web 浏览器执行)。另外，大多数浏览器支持最新版本的 Java。但是现在，`applet` 只不过是 Java 世界中很小的一部分，主流浏览器中支持 Java 2 平台(即 JDK 1.2-1.4)的只有 Netscape 6 和后续的版本。面对这种现实，`applet` 的开发人员有 3 项选择：

- 用 JDK 1.1 或 1.02(为了支持那些实在太老的浏览器)开发 `applet`;
- 让用户安装 Java 运行环境(JRE)1.4 版本，然后使用 JDK 1.4 开发 `applet`;
- 让用户安装任何版本的 Java 2 插件，然后使用 Java 2 开发 `applet`。

对于部署到公众环境中的 `applet`，一般会选择第一个选项，因为这个选项不需要用户安装任何专门的软件。应用这个选项不需要任何特殊的 JSP 语法：只需使用常规的 HTML `APPLET` 标签。但要牢记，`applet` 的 `.class` 文件需要放在能够被浏览器访问的目录中，不要放在 `WEB-INF/classes` 目录中，因为执行它们的是浏览器，不是服务器。但是，由于缺乏对 Java 2 平台的支持，使得这些 `applet` 要受到几项限制。

- 如果要使用 Swing，则必须通过网络发送 Swing 文件。这个过程极为耗时，而且，由于 Swing 依赖于 JDK 1.1，因此，如果浏览器是 Internet Explorer 3 和 Netscape 3.x 和 4.01-4.05(只支持 JDK 1.02)，则不能采用这种方式；
- 不能使用 Java 2D；
- 不能使用 Java 2 集合包；
- 代码的运行会更慢，因为这些代码由早期的编译器生成，而 Java 2 平台的大多数编译器相对于它们的 1.1 版本已经有了显著的提高。

因此，为企业内联网开发复杂 `applet` 的开发人员常常会从后两项中选取。

如果用户都拥有 Internet Explorer 6(或更新的版本)或 Netscape 6(或更新的版本)，第二个选项是最合适的。1.4 版本的 JRE 取代 Java 虚拟机(JVM)与这些浏览器绑定在一起。同样，使用这个选项不需要任何特殊的 JSP 语法：只需使用常规的 HTML 标签。还要切记，`applet` 的 `.class` 文件需要放在浏览器能够访问的目录中，不要放在 `WEB-INF/classes` 目录中，因为执行它们的是浏览器，不是服务器。

核心方法

不管您选择 `applet` 的哪种方式，`applet` 的 `.class` 文件都必须放在浏览器能够访问的目录中，不要将它们放在 `WEB-INF/classes` 目录中。使用它们的是浏览器，不是服务器。

但是，在大型的组织中，许多用户使用的是早期的浏览器，此时，第二项选项是不可行的。因此，为了解决这个问题，Sun 为 Netscape 和 Internet Explorer 开发了浏览器插件，

使用户可以在浏览器的许多不同版本中使用 Java 2 平台。您可以到 <http://java.sun.com/products/plugin/> 下载这个插件，同时，这个插件也和 JDK 1.2.2 及后续的版本捆绑在一起。由于这个插件很大(几兆)，故而不应该期望 WWW 上的用户会为了运行您的 applet 去下载并安装它。但从另一方面讲，对于快速的企业内联网，这是一种合理的方案，尤其是 applet 会自动提示缺少该插件的浏览器去下载这个插件。

遗憾的是，在某些浏览器中，常规的 APPLET 标签不能和插件协同工作，因为这些旧浏览器的设计就是：遇到 APPLET 标签时使用它们自己的内建虚拟机。因此，对于 Internet Explorer，您需要使用冗长麻烦的 OBJECT 标签；对于 Netscape，则要使用同样冗长的 EMBED。此外，由于您一般不知道哪种类型的浏览器会访问您的页面，因此，您的页面必须既包括 OBJECT，又包括 EMBED(将 EMBED 放在 OBJECT 的 COMMENT 部分中)，或者在请求发生时确定浏览器的类型，然后条件性地构建正确的标签。这个过程虽然比较直接简单，但却十分冗长麻烦，耗时良多。

`jsp:plugin` 元素指示服务器为使用插件的 applet 构建一个恰当的标签。这个元素并不向客户端添加任何 Java 功能。那它是怎么做到的呢？JSP 完全在服务器上运行；客户程序对 JSP 一无所知。`jsp:plugin` 元素只是简化了 OBJECT 或 EMBED 标签的生成任务。

重点提示

`jsp:plugin` 元素不向浏览器添加任何 Java 能力。它只是简化了笨重的 OBJECT 和 EMBED 标签的创建工作，Java 2 插件需要这两个标签。

在如何实现 `jsp:plugin` 上，服务器有一些回旋余地，但大多数服务器是通过简单地包含 OBJECT 和 EMBED 来实现它。要检查您的服务器如何转换 `jsp:plugin`，可以在页面中插入简单的 `jsp:plugin` 元素，并提供 type、code、width 和 height 属性，如下面的例子所示。然后，在浏览器中访问这个页面，并查看 HTML 源代码。例如，清单 13.7 给出 Tomcat 为下面的 `jsp:plugin` 元素生成的 HTML 代码。

```
<jsp:plugin type="applet"
             code="SomeApplet.class"
             width="300" height="200">
</jsp:plugin>
```

清单 13.7 Tomcat 为 `jsp:plugin` 生成的代码

```
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
       width="300" height="200"
       all-1_2_2-win.cab#Version=1,2,2,0">
  <param name="java_code" value="SomeApplet.class">
  <param name="type" value="application/x-java-applet;">
<COMMENT>
  <embed type="application/x-java-applet;" width="300" height="200"
         pluginspage="http://java.sun.com/products/plugin/"
         java_code="SomeApplet.class"
  >
<noembed>
</COMMENT>
</noembed></embed>
</object>
```

13.4.1 `jsp:plugin` 元素

使用 `jsp:plugin` 最简单的方式是只提供 4 项属性：`type`，`code`，`width` 和 `height`。将 `type` 属性设为 `applet`，并按照与 `APPLET` 元素完全相同的方式使用其他 3 项属性，但有两点例外：属性名大小写敏感，属性值必须用单引号或双引号引起。因此，我们可以将下面的语句：

```
<APPLET CODE="MyApplet.class"
          WIDTH=475 HEIGHT=350>
</APPLET>
```

替换为：

```
<jsp:plugin type="applet"
              code="MyApplet.class"
              width="475" height="350">
</jsp:plugin>
```

`jsp:plugin` 元素还有许多其他可选属性。大多数与 `APPLET` 元素的属性平行。下面是完整的列表。

- **type**
对于 `applet`，这个属性的值应该为 `applet`。但是，Java 插件还允许我们在 Web 页面中嵌入 JavaBean。这种情况下，就应该使用 `bean` 作为它的值。
- **code**
这个属性的用途与 `APPLET` 的 `CODE` 属性相同，它指定顶级的 applet 类文件(它扩展 `Applet` 或 `JApplet`)。
- **width**
这个属性的用途与 `APPLET` 的 `WIDTH` 属性相同，它指定应该为 applet 保留的宽度，以像素为单位。
- **height**
这个属性的用途与 `APPLET` 的 `HEIGHT` 属性相同，它指定应该为 applet 保留的高度，以像素为单位。
- **codebase**
这个属性的用途与 `APPLET` 的 `CODEBASE` 属性相同，它指定 applet 的根目录。`code` 属性就是相对于这个目录进行解析。和 `APPLET` 元素一样，如果省略这个属性，则当前页面所在的目录成为默认值。对于 JSP，这个默认位置是原来的 JSP 文件所在的目录，不是由 JSP 文件生成的 `servlet` 的位置(这个位置与具体系统相关)。
- **align**
这个属性的用途与 `APPLET` 和 `IMG` 的 `ALIGN` 属性相同，它指定 applet 在 Web 页面内的对齐方式。合法的值是 `left`, `right`, `top`, `bottom` 和 `middle`。
- **hspace**
这个属性的用途与 `APPLET` 的 `HSPACE` 属性相同，它指定 applet 左边和右边保留的空白区域，以像素为单位。

- **vspace**

这个属性的用途与 APPLET 的 VSPACE 属性相同, 它指定 applet 顶部和底部保留的空白区域, 以像素为单位。

- **archive**

这个属性的用途与 APPLET 的 ARCHIVE 属性相同, 它指定一个 JAR 文件, 类和图像应该从这个 JAR 载入。

- **name**

这个属性的用途与 APPLET 的 NAME 属性相同, 它指定一个名字, 用于 applet 之间的通信, 或用在脚本语言中(如 JavaScript)标识这个 applet。

- **title**

这个属性的用途与 APPLET(以及几乎 HTML 4.0 中的所有其他 HTML 元素)的 TITLE 属性(极少使用)相同, 它指定一个标题, 可以用于工具提示或索引。

- **jreversion**

这个属性标识所需的 Java 运行环境(JRE)的版本。默认是 1.2。

- **iepluginurl**

这个属性指定可以下载 Internet Explorer 插件的 URL。尚未安装插件的用户会得到相应的提示, 指示他们到这个位置下载插件。默认值会将用户导向 Sun 的网站, 但对于内联网应用来说, 您或许会希望将用户导向本地副本。

- **nspluginurl**

这个属性指定可以下载 Netscape 插件的 URL。默认值会将用户导向 Sun 的网站, 但对于内联网应用来说, 您或许会希望将用户导向本地副本。

13.4.2 **jsp:param** 和 **jsp:params** 元素

jsp:param 元素在 **jsp:plugin** 中的应用从某种意义上类似于 **PARAM** 在 **APPLET** 中的应用, 它用来指定一个名字和一个值, 在 **applet** 中可以用 **getParameter** 来访问它。但是, **jsp:param** 和 **PARAM** 之间存在两项主要的差异。首先, 由于 **jsp:param** 遵循 XML 语法, 因此, 属性名必须小写, 属性值必须引在单引号或双引号内, 同时元素必须以`>/`结束, 不能仅仅是`>`。其次, 所有 **jsp:param** 项都必须封闭在一个 **jsp:params** 元素内。

从而, 我们会将下面的语句:

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
<PARAM NAME="PARAM1" VALUE="VALUE1">
<PARAM NAME="PARAM2" VALUE="VALUE2">
```

替换为:

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">

<jsp:params>
    <jsp:param name="PARAM1" value="VALUE1" />
    <jsp:param name="PARAM2" value="VALUE2" />
</jsp:params>
```

```
</jsp:plugin>
```

13.4.3 jsp:fallback 引用

jsp:fallback 元素向不支持 OBJECT 或 EMBED 的浏览器提供一段替换性的文字。这个元素的使用方式几乎和 APPLET 元素内使用替换性文字的方式相同。我们可以将下面的语句：

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
<B>Error: this example requires Java.</B>
</APPLET>
```

替换为

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
<jsp:fallback>
<B>Error: this example requires Java.</B>
</jsp:fallback>
</jsp:plugin>
```

13.4.4 jsp:plugin 示例

清单 13.8 给出的 JSP 页面使用 **jsp:plugin** 元素产生一个 Java 2 插件项。清单 13.9 给出 applet 本身的代码(这个 applet 用到了 Swing, Java 2D, 以及清单 13.10 到 13.12 中的辅助类)。图 13.3 给出相应的结果。

清单 13.8 PluginApplet.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using jsp:plugin</TITLE>
<LINK REL=stylesheet
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
    Using jsp:plugin</TABLE>
<P>
<jsp:plugin type="applet"
            code="PluginApplet.class"
            width="370" height="420">
</jsp:plugin>
</CENTER></BODY></HTML>
```

清单 13.9 PluginApplet.java

```
import javax.swing.*;
```

```
/** An applet that uses Swing and Java 2D and thus requires
 * the Java Plug-In.
 */

public class PluginApplet extends JApplet {
    public void init() {
        WindowUtilities.setNativeLookAndFeel();
        setContentPane(new TextPanel());
    }
}
```

清单13.10 TextPanel.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** JPanel that places a panel with text drawn at various angles
 * in the top part of the window and a JComboBox containing
 * font choices in the bottom part.
 */

public class TextPanel extends JPanel
    implements ActionListener {
    private JComboBox fontBox;
    private DrawingPanel drawingPanel;

    public TextPanel() {
        GraphicsEnvironment env =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        String[] fontNames = env.getAvailableFontFamilyNames();
        fontBox = new JComboBox(fontNames);
        setLayout(new BorderLayout());
        JPanel fontPanel = new JPanel();
        fontPanel.add(new JLabel("Font:"));
        fontPanel.add(fontBox);
        JButton drawButton = new JButton("Draw");
        drawButton.addActionListener(this);
        fontPanel.add(drawButton);
        add(fontPanel, BorderLayout.SOUTH);
        drawingPanel = new DrawingPanel();
        fontBox.setSelectedItem("Serif");
        drawingPanel.setFontName("Serif");
        add(drawingPanel, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        drawingPanel.setFontName((String)fontBox.getSelectedItem());
        drawingPanel.repaint();
    }
}
```

清单13.11 DrawingPanel.java

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/** A window with text drawn at an angle. The font is
```

```
* set by means of the setFontName method.  
*/  
  
class DrawingPanel extends JPanel {  
    private Ellipse2D.Double circle =  
        new Ellipse2D.Double(10, 10, 350, 350);  
    private GradientPaint gradient =  
        new GradientPaint(0, 0, Color.red, 180, 180, Color.yellow,  
                          true); // true means to repeat pattern  
    private Color[] colors = { Color.white, Color.black };  
  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        Graphics2D g2d = (Graphics2D)g;  
        g2d.setPaint(gradient);  
        g2d.fill(circle);  
        g2d.translate(185, 185);  
        for (int i=0; i<16; i++) {  
            g2d.rotate(Math.PI/8.0);  
            g2d.setPaint(colors[i%2]);  
            g2d.drawString("jsp:plugin", 0, 0);  
        }  
    }  
  
    public void setFontName(String fontName) {  
        setFont(new Font(fontName, Font.BOLD, 35));  
    }  
}
```

清单13.12 WindowUtilities.java

```
import javax.swing.*;  
import java.awt.*;  
  
/** A few utilities that simplify using windows in Swing. */  
  
public class WindowUtilities {  
  
    /** Tell system to use native look and feel, as in previous  
     * releases. Metal (Java) LAF is the default otherwise.  
     */  
  
    public static void setNativeLookAndFeel() {  
        try {  
            UIManager.setLookAndFeel  
                (UIManager.getSystemLookAndFeelClassName());  
        } catch(Exception e) {  
            System.out.println("Error setting native LAF: " + e);  
        }  
    }  
    ...// See www.coreservlets.com for remaining code.  
}
```

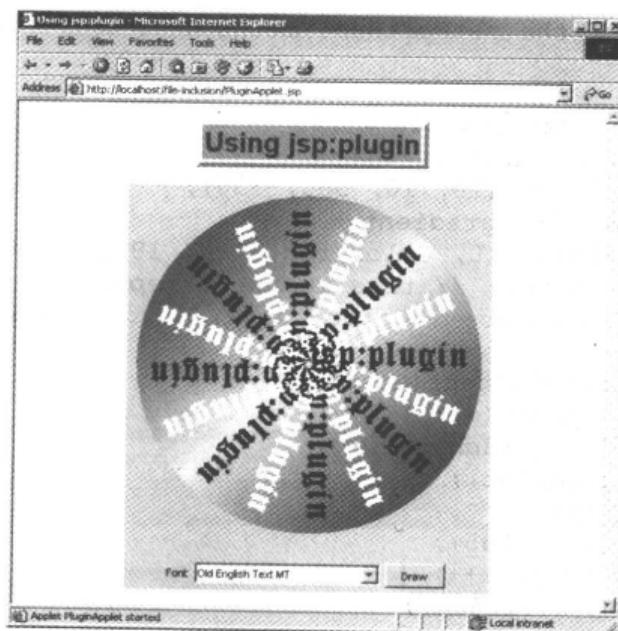


图 13.3 PluginApplet.jsp 在 Internet Explorer(安装有 JDK 1.4 插件)中的结果

第 14 章 JavaBean 组件在 JSP 文档中的应用

本章的主题：

- bean 的优点
- bean 的创建
- bean 类在服务器上的安装
- 访问 bean 属性
- 显式地设置 bean 的属性
- 根据请求参数自动设置 bean 的属性
- 多个 servlet 和 JSP 页面间共享 bean

本章论述向 JSP 页面插入动态内容的第三种通用策略(参见图 14.1)：借助于 JavaBean 组件。

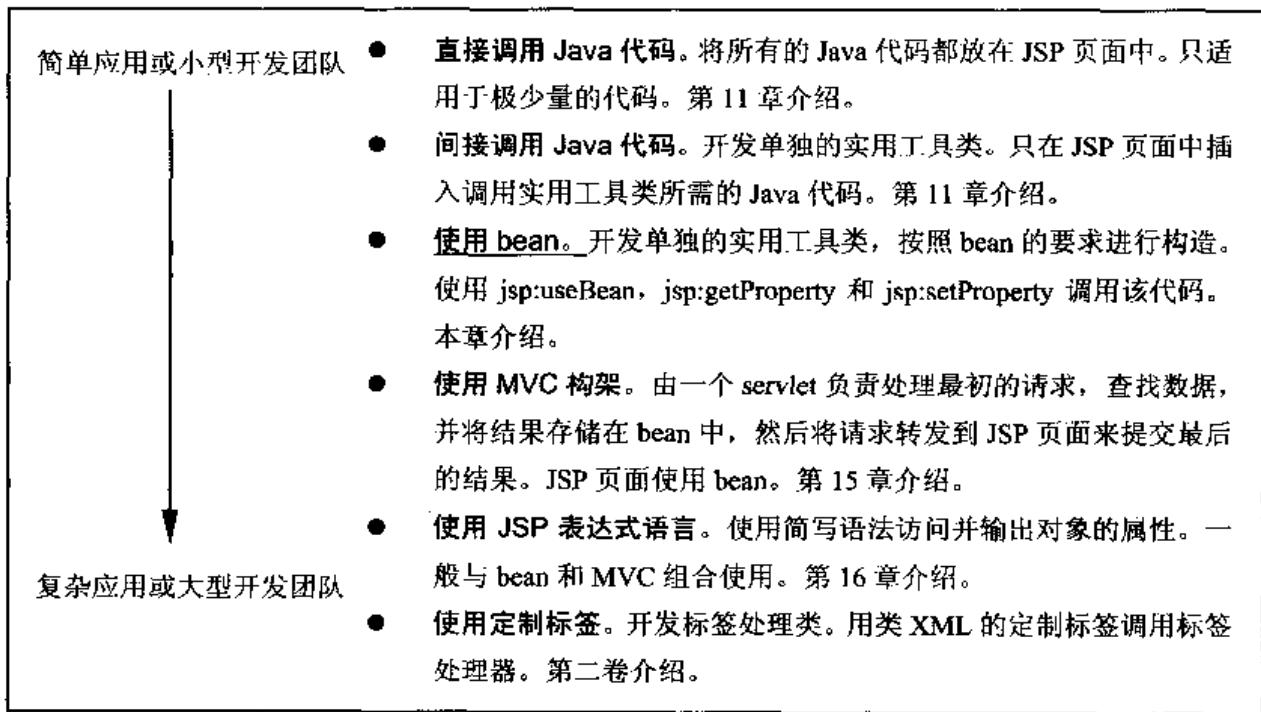


图 14.1 JSP 中调用动态代码的策略

14.1 使用 bean 的原因

您或许已经了解使用单独的 Java 类相对于直接在 JSP 页面中嵌入大量代码的优点。如 11.3 节所述，单独的类更易于编写、编译、测试、调试和重用。那么，bean 提供了什么其

他类型的类不能提供的东西呢？总体上讲，bean 不过是遵循某些简单约定的常规 Java 类，这些约定由 JavaBean 规范所定义；bean 并没有扩展特殊的类，也不在特殊的包中，同时，也没有使用特殊的接口。

尽管 bean 确实只是按照标准格式编写的 Java 类，但使用它们会有几种优点。一般地，可视化的处理工具和其他程序能够自动发现遵循这种格式的类的相关信息，无需用户编写任何代码就能够创建和操作这些类。尤其是在 JSP 中，相对于 scriptlet 和引用常规 Java 类的 JSP 表达式，JavaBean 组件的应用提供 3 项优点：

(1) 不需用到 Java 语法。

使用 bean，页面的创作者可以只使用 XML 兼容的语法操作 Java 对象：不需用到圆括号，分号和花括号，从而促进内容与表达之间的强分离，这对于拥有独立 Web 和 Java 开发人员的大型开发团队尤其有用。

(2) 对象的共享更为简单。

相比于使用等同功能的显式 Java 代码，使用 JSP bean 构造时，对象在多个页面或请求间的共享要容易得多。

(3) 请求参数与对象属性之间可以方便地对应起来。

JSP bean 极大地简化了读取请求参数，对字符串进行适当的转换，并将结果放入到对象中的过程。

14.2 bean 是什么

如我们所述，bean 不过是按照某种标准格式编写的 Java 类。全面地介绍 JavaBean 超出了本书的范畴，但就 JavaBean 在 JSP 中的使用来说，您只需了解下面列出的 3 点就足够了。如果希望更详细地了解 bean 的一般应用，请参考众多论述这一主题的相关书籍，或者参考 <http://java.sun.com/products/javabeans/docs/> 提供的文档和教程。

- bean 类必须拥有一个零参数的(默认)构造函数。

要满足这项要求，既可以显式地定义这样一个构造函数，也可以省略所有的构造函数(这样零参数的构造函数会被自动创建)。JSP 元素创建 bean 时，会调用默认构造函数。实际上，如第 15 章所见，实际的应用中经常会由 servlet 创建 bean，JSP 页面只是从中查询数据。这种情况下，不必要求 bean 必须要有零参数的构造函数。

- bean 类不应该有公开的实例变量(字段)。

要成为 JSP 可以访问的 bean，相应的类应该使用访问器方法(accessor method)取代对实例变量的直接访问。在面向对象的程序设计中，这是一项重要的设计原则，我们希望您在实践中已经遵循了这项原则。一般而言，访问器方法的使用可以让类的用户在不改变代码的情况下完成 3 件事：(a)对变量的值加以约束(如让 Car 类的 SetSpeed 方法拒绝负的速度)；(b)改变内部的数据结构(如在内部从英制单位改为公制单位，同时依旧保持 getSpeedInMPH 和 getSpeedInKPH 方法)；(c)在值发生改变时自动执行边界效应(比如：在调用 setPosition 时自动更新用户界面)。

- 持续性的值应该通过 getXxx 和 setXxx 方法来访问。

- 例如，如果您的 Car 类存储当前的乘客数，您可能会提供 `getNumPassengers`(没有参数且返回 int)和 `setNumPassengers`(接受 int 参数且返回 void 类型)方法。这种情况下，我们称 Car 类拥有一个名为 `numPassengers` 的属性(property)(注意，属性名中是小写的 n，但方法名中是大写的 N)。如果类拥有 `getXxx` 方法，但没有对应的 `setXxx` 方法，则称该类拥有一个只读属性 `xxx`。

这种命名约定的一个例外是布尔属性：我们可以使用方法 `isXxx` 检查它们的值。

例如，您的 Car 类可能拥有名为 `isLeased` 方法(没有参数且返回 boolean)和 `setLeased`(接受一个 boolean 且返回类型为 void)，一般称为 Car 类拥有一个布尔属性，名为 `leased`(同样要注意，属性名的第一个字母小写)。尽管我们可以使用 JSP scriptlet 或表达式访问类的任意方法，但是，访问 bean 的标准 JSP 动作只能使用那些遵循 `getXxx/setXxx` 或 `isXxx/setXxx` 命名约定的方法。

14.3 bean 的应用：基本任务

JSP 页面中，可以使用 3 种主要的构造来构建和操作 JavaBean 组件。

- **`<jsp:useBean`**

这个元素的最简单形式只是构建一个新的 bean。它的常规使用方式如下：

```
<jsp:useBean id="beanName"
              class="package.Class" />
```

如果提供 `scope` 属性(参见 14.6 节)，则 `jsp:useBean` 既可以构建新的 bean，也可以访问现存的 bean。

- **`<jsp:getProperty`**

这个元素读取或输出 bean 属性的值。用该元素读取属性是 `getXxx` 调用的简单记法。这个元素的使用如下：

```
<jsp:getProperty name="beanName"
                  property="propertyName" />
```

- **`<jsp:setProperty`**

这个元素修改 bean 的属性(即调用形如 `setXxx` 的方法)。它的常规使用方式如下：

```
<jsp:setProperty name="beanName">
                  property="propertyName"
                  value="PropertyValue" />
```

下面的小节详细叙述每个元素。

14.3.1 构建 bean: `jsp:useBean`

`jsp:useBean` 用以载入将要用在 JSP 页面中的 bean。由于 bean 可以直接利用 Java 类的可重用性，并且不会牺牲使用 JSP 所带来的便利(相对于单独使用 servlet 而言)，因此，bean 为我们提供一项十分有效的功能。

指定使用哪个 bean 的最简单形式是：

```
<jsp:useBean id="name" class="package.Class" />
```

这个语句一般表示“实例化由 class 指定的类，并将实例化后的对象绑定到 _jspService 中的变量，变量的名字由 id 指定”。但要注意，一定要使用完全限定类名——包括包名的类名。不管您是否使用<%@ page import... %>输入包，都要满足这个要求。

警告

`jsp:useBean` 的 class 属性必须使用完全限定类名。

例如，一般可以将 JSP 动作

```
<jsp:useBean id="book1" class="coreservlets.Book" />
```

认为是下面这段 scriptlet 的等价物。

```
<% coreservlets.Book book1 = new coreservlets.Book(); %>
```

14.3.2 bean 类的安装

bean 类的定义不能放在含有 JSP 文件的目录中，而应该放在安装 servlet 的目录中。但要牢记一定要使用包(详细信息参见 11.3 节)；因此，单个 bean 类的正确位置是 WEB-INF/classes/subdirectoryMatchingPackageName，如 2.10 节和 2.11 节所述。含有 bean 类的 JAR 文件应该放在 WEB-INF/lib 目录中。

核心方法

将所有的 bean 都放在包中。安装在通常用来存放 Java 代码的目录中：单个的类放在 WEB-INF/classes/subdirectoryMatchingPackageName 中，JAR 文件放在 WEB-INF/lib 中。

14.3.3 jsp:useBean 选项的使用：scope，beanName 和 type

尽管我们可以近似地将 `jsp:useBean` 看作等同于构建一个对象并将其绑定到本地变量，但是 `jsp:useBean` 的功能远不止此，它还提供其他一些功能更为强大的选项。如 14.6 节所示，我们可以指定 scope 属性将 bean 与其他页面关联起来(而非仅仅限于当前页面)。如果 bean 能够共享，那么获得现有 bean 的引用将会变得十分重要(而非每次都构造新的对象)。因此，`jsp:useBean` 动作规定，仅当不存在相同 id 和 scope 的 bean 时才实例化新的对象。

除使用 class 属性外，我们还可以使用 beanName。二者的不同是，beanName 既可以指向类，也可以指定含有序列化 bean 对象的文件。beanName 属性的值被传递给 `java.beans.Bean` 的 `instantiate` 方法。

大多数情况下，您希望本地变量和要创建的对象类型相同。但是，少数情况下，您或许希望所要声明的变量的类型是实际 bean 类型的超类，或是 bean 实现的接口。应该使用 type 属性来控制这种声明，如下所示：

```
<jsp:useBean id="thread1" class="mypackage.MyClass"
              type="java.lang.Runnable" />
```

这种应用导致类似下面的代码插入到 `_jspService` 方法中。

```
java.lang.Runnable thread1 = new myPackage.MyClass();
```

如果实际的类与 type 不兼容，则会产生 ClassCastException 异常。同样，如果这个 bean 业已存在，并且您只想使用现有的对象，而非创建新的对象，那么您可以只使用 type，省略 class。在使用 scope 属性共享 bean 时(如 14.6 节所述)，这样做比较有用。

要注意，由于 jsp:useBean 使用 XML 语法，所以它的格式与 HTML 语法存在三方面的不同：属性名大小写敏感，单引号和双引号都可以使用(但必须使用其中的一种)，标签的结束记为 />，不只是 >。前两项语法差异适用于所有形如 jsp:xxx 的 JSP 元素。第三项差异仅适用于元素是有单独起始和结束标签的容器。

几个字符序列在属性值中出现时需要特殊处理。要在属性值中获得'，使用\'。类似地，要获得"，使用\"；要获得\，使用\\；要获得%>，使用%\\>；要获得<%，使用<\\%。

14.3.4 访问 bean 的属性：jsp:getProperty

获得 bean 之后，就可以使用 jsp:getProperty 输出它的属性，jsp:getProperty 的 name 属性应该与 jsp:useBean 给定的 id 匹配，它的 property 属性指定需要的属性。

重点提示

在 jsp:useBean 中，bean 名由 id 属性给出。而在 jsp:getProperty 和 jsp:setProperty 中，由 name 属性给出。

除可以使用 jsp:getProperty 以外，您还可以使用 JSP 表达式，显式地调用对象的方法，变量的名称由 id 属性指定。例如，假定 Book 类有一个 String 类型的属性，名为 title，并且您已经用本节之前给出的 jsp:useBean 示例创建了名为 book1 的实例，那么，您可以用下面给出的两种方式将 title 属性的值插入到 JSP 页面中。

```
<jsp:getProperty name="book1" property="title" />  
<%= book1.getTitle() %>
```

对于这种情况，第一种方式比较好，因为这种语法更容易被那些不熟悉 Java 编程语言的 Web 设计人员所接受。如果您是用 jsp:useBean 创建对象，而非等同的 JSP scriptlet，则应该保持语法的一致性，用 jsp:getProperty 输出 bean 的属性，而不是使用等同的 JSP 表达式。但是，在使用循环、条件语句和没有表达成属性的方法时，就需要用到对变量的直接访问。

如果您不熟悉 bean 属性的概念，语句“这个 bean 拥有类型为 T 的属性 foo”的标准解释是“这个类有一个 getFoo 方法，它的返回类型为 T，并且它还有另外的方法 setFoo，它接受 T 为参数，保存它供之后用 getFoo 访问。”

14.4.5 简单 bean 属性的设置：jsp:setProperty

修改 bean 的属性一般使用 jsp:setProperty。这个动作有几种不同的形式，在最简单的形式中我们需要提供 3 个属性：name(应该与 jsp:useBean 给出的 id 匹配)，property(要更改的属性的名称)和 value(新的值)。在 14.5 节中，我们提供一些其他形式的 jsp:setProperty，它

们允许您自动将属性与请求参数关联。14.5 节中还说明如何将在请求期间计算得出的值(而非固定的字符串)作为属性的值，并讨论了类型转换约定(将字符串值提供给期望数字型、字符型或布尔型值的参数)。

`jsp:setProperty` 动作的一种替代方式是使用 scriptlet 显式地调用 bean 对象的方法。例如，给定 `book1` 对象(本节前面曾出现过)，可以使用下面两种形式修改 `title` 属性。

```
<jsp:setProperty name="book1"
                  property="title"
                  value="Core Servlets and JavaServer Pages" />
<% book1.setTitle("Core Servlets and JavaServer Pages"); %>
```

使用 `jsp:setProperty` 的优点是非程序开发人员可以更容易使用它，而直接访问对象则可以执行更为复杂的操作，比如根据条件设置值或调用对象 `getXxx` 或 `setXxx` 以外的其他方法。

14.4 示例：StringBean

清单 14.1 列出 `coreservlets` 包中一个简单的类——`StringBean`。由于这个类没有公开的实例变量(字段)，同时，它没有声明任何显式的构造函数，从而也就拥有一个默认的零参数构造函数，因此，它满足成为 bean 的条件。由于 `StringBean` 有一个 `getMessage` 方法，它返回 `String`，还有另一个方法 `setMessage`，它取 `String` 为参数，在 bean 的术语中，我们称这个类拥有一个名为 `message` 的 `String` 属性。

清单 14.2 给出使用 `StringBean` 类的一个 JSP 文件。这个 JSP 文件首先用 `jsp:useBean` 创建 `StringBean` 的一个实例，如下所示。

```
<jsp:useBean id="stringBean" class="coreservlets.StringBean" />
```

此后，可以通过两种方式将 `message` 属性插入到页面中。

```
<jsp:getProperty name="stringBean" property="message" />
<%= stringBean.getMessage() %>
```

可以用下面两种方式修改 `message` 属性。

```
<jsp:setProperty name="stringBean"
                  property="message"
                  value="some message" />
<% stringBean.setMessage("some message"); %>
```

请注意，我们不推荐在同一页面中混合使用显式 Java 语法和 XML 语法；这个例子只是为了说明这两种形式的效果相同。

核心方法

要尽可能避免混合使用 XML 兼容的 `jsp:useBean` 标签和含有显式 Java 代码的 JSP 脚本元素。

图 14.2 给出相应的结果。

清单14.1 StringBean.java

```
package coreservlets;

/** A simple bean that has a single String property
 * called message.
 */

public class StringBean {
    private String message = "No message specified";

    public String getMessage() {
        return(message);
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

清单14.2 StringBean.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using JavaBeans with JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
    <TR><TH CLASS="TITLE">
        Using JavaBeans with JSP</TABLE>
<jsp:useBean id="stringBean" class="coreservlets.StringBean" />
<OL>
    <LI>Initial value (from jsp:getProperty):
        <I><jsp:getProperty name="stringBean"
                           property="message" /></I>
    <LI>Initial value (from JSP expression):
        <I><%= stringBean.getMessage() %></I>
    <LI><jsp:setProperty name="stringBean"
                           property="message"
                           value="Best string bean: Fortex" />
        Value after setting property with jsp:setProperty:
        <I><jsp:getProperty name="stringBean"
                           property="message" /></I>
    <LI><% stringBean.setMessage("My favorite: Kentucky Wonder"); %>
        Value after setting property with scriptlet:
        <I><%= stringBean.getMessage() %></I>
</OL>
</BODY></HTML>
```

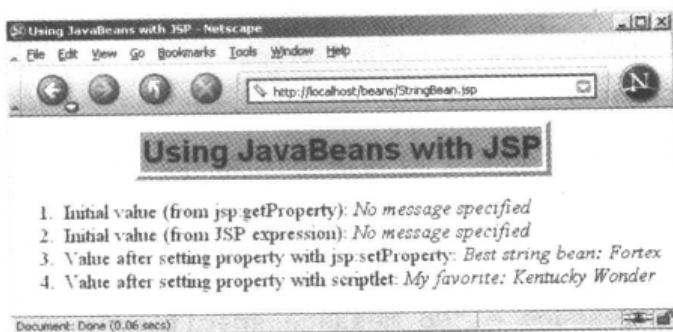


图 14.2 StringBean.jsp 的结果

14.5 设置 bean 的属性：高级技术

我们一般使用 `jsp:setProperty` 设置 bean 属性。这个动作最简单的形式需要用到 3 个属性：`name`(应该与 `jsp:useBean` 给出的 `id` 匹配)，`property`(要改变的属性的名称)和 `value`(新的值)。

例如，清单 14.3 中的 `SaleEntry` 类有一个 `itemID` 属性(`String` 类型)、一个 `numItems` 属性(`int` 类型)、一个 `discountCode` 属性(`double` 类型)以及两个只读属性，`itemCost` 和 `totalCost`(均为 `double` 类型)。清单 14.4 给出的 JSP 文件使用下面的方式构建 `SaleEntry` 的实例：

```
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
```

清单 14.5(图 14.3)给出用来收集请求参数的 HTML 表单。结果在图 14.4 给出。

bean 实例化之后，使用请求参数对 `itemID` 进行设置就比较简单直接了，如下所示。

```
<jsp:setProperty
    name="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />
```

可以看到，我们用 JSP 表达式作为 `value` 属性的值。大多数 JSP 属性值必须是固定的字符串，但 `jsp:setProperty` 的 `value` 属性允许给出的值为请求期间的表达式。如果表达式内部用到了双引号，则要回顾一下前面讲述过的内容：单引号可以替代属性值两边的双引号，在属性值中可以用\'和\"表示单引号或双引号。不管哪种情况，要点是：此处可以使用 JSP 表达式，但这样做需要用到显式 Java 代码。在某些应用中，避免这类显式代码是使用 bean 的首要原因。此外，如下一个例子所示，如果 bean 的属性不是 `String` 类型时，情况会变得极为复杂。下面两个小节将论述如何解决这些问题。

清单 14.3 SaleEntry.java

```
package coreservlets;

/** Simple bean to illustrate the various forms
 * of jsp:setProperty.
 */

public class SaleEntry {
    private String itemID = "unknown";
    private double discountCode = 1.0;
```

```
private int numItems = 0;

public String getItemID() {
    return(itemID);
}

public void setItemID(String itemID) {
    if (itemID != null) {
        this.itemID = itemID;
    } else {
        this.itemID = "unknown";
    }
}

public double getDiscountCode() {
    return(discountCode);
}

public void setDiscountCode(double discountCode) {
    this.discountCode = discountCode;
}

public int getNumItems() {
    return(numItems);
}

public void setNumItems(int numItems) {
    this.numItems = numItems;
}

// In real life, replace this with database lookup.
// See Chapters 17 and 18 for info on accessing databases
// from servlets and JSP pages.

public double getItemCost() {
    double cost;
    if (itemID.equals("a1234")) {
        cost = 12.99*getDiscountCode();
    } else {
        cost = -9999;
    }
    return(roundToPennies(cost));
}

private double roundToPennies(double cost) {
    return(Math.floor(cost*100)/100.0);
}

public double getTotalCost() {
    return(getItemCost() * getNumItems());
}
}
```

清单14.4 SaleEntry1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using jsp:setProperty</TITLE>
```

```

<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
    Using jsp:setProperty</TABLE>
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
<jsp:setProperty
    name="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />
<%
int numItemsOrdered = 1;
try {
    numItemsOrdered =
        Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="numItems"
    value="<%= numItemsOrdered %>" />
<%
double discountCode = 1.0;
try {
    String discountString =
        request.getParameter("discountCode");
    discountCode =
        Double.parseDouble(discountString);
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="discountCode"
    value="<%= discountCode %>" />
<BR>
<TABLE BORDER=1>
<TR CLASS="COLORED">
    <TH>Item ID<TH>Unit Price<TH>Number Ordered<TH>Total Price
<TR ALIGN="RIGHT">
    <TD><jsp:getProperty name="entry" property="itemID" />
    <TD>$<jsp:getProperty name="entry" property="itemCost" />
    <TD><jsp:getProperty name="entry" property="numItems" />
    <TD>$<jsp:getProperty name="entry" property="totalCost" />
</TABLE>
</CENTER></BODY></HTML>

```

清单14.5 SaleEntry1-Form.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Invoking SaleEntry1.jsp</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">

```

```

</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
    Invoking SaleEntry1.jsp</TABLE>
<FORM ACTION="SaleEntry1.jsp">
    Item ID: <INPUT TYPE="TEXT" NAME="itemID"><BR>
    Number of Items: <INPUT TYPE="TEXT" NAME="numItems"><BR>
    Discount Code: <INPUT TYPE="TEXT" NAME="discountCode"><P>
    <INPUT TYPE="SUBMIT" VALUE="Show Price">
</FORM>
</CENTER></BODY></HTML>

```

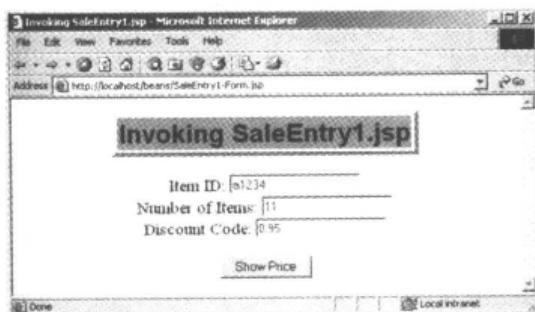


图 14.3 SaleEntry1.jsp 的前端界面

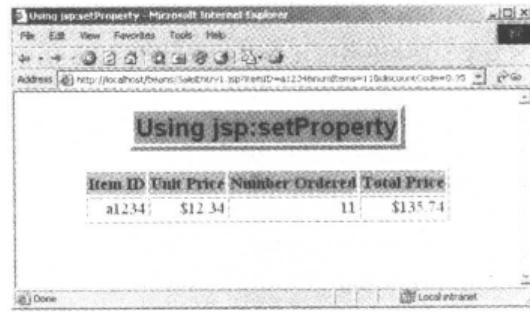


图 14.4 SaleEntry1.jsp 的结果

14.5.1 将单个属性与输入参数关联

itemID 属性的设置比较容易，因为它的值是 String。设置 numItems 和 discountCode 属性就要遇到更多的问题，因为它们的值必须是数字，而 getParameter 返回的是 String。下面是设置 numItems 所需的代码(稍显累赘)。

```

<%
int numItemsOrdered = 1;
try {
    numItemsOrdered =
        Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfc) {}
%>
<jsp:setProperty
    name="entry"
    property="numItems"
    value="<%=" numItemsOrdered %>" />

```

幸运的是，针对这个问题，JSP 有一个不错的解决方案。它允许将属性与请求参数关联，自动执行从字符串到数字、字符和布尔值的类型转换。我们可以不使用 value 属性，而是使用 param 指定一个输入参数，被指定的请求参数的值自动用作 bean 属性的值，由 String 到基本类型(byte, int, double 等)和包装类(Byte, Integer, Double 等)的类型转换都自动执行。如果请求中没有指定的参数，则不采取任何动作(系统并不传递 null 到相关联的属性)。从而，对 numItems 属性的设置可以简化为：

```

<jsp:setProperty
    name="entry"
    property="numItems"

```

```
param="numItems" />
```

如果请求参数的名称和 bean 属性的名称相同，还可以更进一步地简化这段代码。这种情况下可以省略 param，如下例所示。

```
<jsp:setProperty
    name="entry"
    property="numItems" /> <%-- param="numItems" is assumed. --%>
```

我们倾向于使用稍长一些的形式，明确地列出参数来。清单 14.6 给出 JSP 页面中用这种方式重写后的相关部分。

清单 14.6 SaleEntry2.jsp

```
...
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
<jsp:setProperty
    name="entry"
    property="itemID"
    param="itemID" />
<jsp:setProperty
    name="entry"
    property="numItems"
    param="numItems" />
<jsp:setProperty
    name="entry"
    property="discountCode"
    param="discountCode" />
...
...
```

14.5.2 将所有的属性与请求参数关联

将属性与请求参数关联可以省略许多简单内建类型的转换工作。JSP 允许我们将这个过程更进一步：将所有的属性与同名的请求参数关联起来。我们所要做的只是以“*”作为 property 参数的值。因此，清单 14.6 中全部 3 个 jsp:setProperty 语句可以替换为下面简单的行。清单 14.7 给出页面的相关部分。

```
<jsp:setProperty name="entry" property="*" />
```

清单 14.7 SaleEntry3.jsp

```
...
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
<jsp:setProperty name="entry" property="*" />
...
```

通过这种方式，我们可以定义简单的“表单 bean”，它的属性和请求参数对应，从而能自动填充。系统从请求参数开始查找匹配的 bean 属性，而非采用相反的方式。因此，对于没有请求参数与之匹配的属性，则不采取任何动作。这种行为意味着：表单 bean 不必一次填充完毕，相反，一次提交可以填写 bean 的部分属性，另一个表单填写其他属性，以此类推。但要使用这项功能，需要将 bean 在多个页面间共享。详细信息参见 14.6 节。最后要注意，servlet 也可以使用表单 bean——尽管只能通过使用某些定制工具来完成。详细信息

参见 4.7 节。

尽管这种方式比较简单，但要注意三点。

- **输入参数缺失时不采取任何动作。**

特别地，系统不会提供 null 作为属性的值。因此，在设计 bean 时，一般要为其设置可以识别的默认值，以便可以确定某个属性是否修改过。

- **自动类型转换并不能像手动类型转换那样能够防止不合法的值。**

实际上，尽管自动类型转换很方便，一些开发人员还是避免使用自动转换，而是将所有可以设置的 bean 属性定义为 String 类型，并使用显式的 try/catch 块来处理异常数据。至少，在使用自动类型转换时，应该考虑使用错误处理页面。

- **bean 属性的名称和请求参数大小写敏感。**

因此，属性名和请求参数的名字必须精确匹配。

14.6 共享 bean

到此为止，对于 `<jsp:useBean` 创建的对象，我们都将它们看作是 `_jspService` 方法中的局部变量来处理(`_jspService` 由页面生成的 `servlet` 的 `service` 方法调用)。虽然 bean 的确绑定到局部变量，但这并非惟一的行为。它们还存储在 4 个不同的位置——依 `<jsp:useBean` 的可选属性 `scope` 的值而定。

使用 `scope` 时，系统首先检查指定的位置是否存在指定名称的 bean。仅当系统找不到现有的 bean 时，才会创建新的 bean。这种行为使得 `servlet` 可以用下面的方式处理复杂的用户请求：建立 bean，将它们存储在 3 个标准共享位置中的某个位置(请求、会话或 `servlet` 的上下文)，然后将请求转发到几个适当 JSP 页面中的某个页面，提供适合于请求数据的结果。这种方式的详细信息参见第 15 章。

如下所述，`scope` 属性有 4 个可选值：`page`(默认)，`request`，`session` 和 `application`。

- `<jsp:useBean ... scope="page" />`

这是默认的值；完全省略 `scope` 属性也会得到相同的行为。`page` 作用域表示：在处理当前请求期间，除了要将 bean 对象绑定到局部变量外，还应该将它放在 `PageContext` 对象中。将对象存储在此处表示，`servlet` 可以通过调用预定义变量 `pageContext` 的 `getAttribute` 方法访问它。

由于每个页面和每个请求都有不同的 `PageContext` 对象，所以 `scope="page"`(或省略 `scope`)表示不共享 bean，也就是针对每个请求都创建新的 bean。

- `<jsp:useBean ... scope="request" />`

这个值表示：在处理当前请求期间，除了要将 bean 对象绑定到局部变量外，还应该将它放在 `HttpServletRequest` 对象中，从而可以通过 `getAttribute` 方法访问它。尽管乍看起来这个作用域好像也产生非共享 bean，但是，在使用 `jsp:include`(13.1 节)，`jsp:forward`(13.3 节)，或者 `RequestDispatcher` 的 `include` 或 `forward` 方法(第 15 章)时，两个 JSP 页面，或 JSP 页面和 `servlet` 将会共享请求对象。

使用 MVC(模型 2)构架时，经常会在请求对象中存储具体的值。详细信息参见第 15 章。

- <jsp:useBean ... scope="session" />

这个值表示，除了要将 bean 绑定到局部变量以外，还要将它存储到与当前请求关联的 HttpSession 对象中，我们可以使用 getAttribute 获取存储在 HttpSession 中的对象。

因此，使用这个作用域，JSP 页面可以容易地执行第 9 章中介绍的会话跟踪。

- <jsp:useBean ... scope="application" />

这个值表示，除了要将 bean 绑定到局部变量以外，还要将它存储在 ServletContext 中(通过预定义 application 变量或通过调用 getServletContext 获得)。ServletContext 由 Web 应用中多个 servlet 和 JSP 页面所共享。ServletContext 中的值可以用getAttribute 方法获取。

根据条件创建 bean

为了使 bean 的共享更为便利，两种情形下，我们可以有条件地对与 bean 相关的元素进行求值。

首先，仅当找不到相同 id 和 scope 的 bean 时，jsp:useBean 元素才会实例化新的 bean。如果存在相同 id 和 scope 的 bean，则只是将已有的 bean 绑定到相关的变量(由 id 指定)。

其次，我们也可以不使用

```
<jsp:useBean .../>
```

转而使用

```
<jsp:useBean ...> statements</jsp:useBean>
```

使用第二种形式的意义在于，jsp:useBean 的起始标签和结束标签之间的语句只在创建新的 bean 时执行，如果使用已有的 bean，则不执行。由于 jsp:useBean 调用默认(零参数)构造函数，因此，我们经常需要在 bean 创建之后修改它的属性。为了模拟构造函数，应该只在 bean 首次创建时执行这些修改，而不应该在访问现存(或更新后)的 bean 时执行。做到这一点并不困难：多个页面都可以在 jsp:useBean 的起始标签和结束标签之间包含 jsp:setProperty 语句：只有被第一个访问的页面才会执行这些语句。

例如，清单 14.8 给出一个简单的 bean，它定义了两个属性，accessCount 和 firstPage。accessCount 属性记录对一系列相关页面累积的访问计数，因此，针对所有的请求都要执行。firstPage 属性存储被访问的第一个页面的名称，因此应该在页面被首次访问时执行。为了区分，我们将更新 accessCount 属性的 jsp:setProperty 语句放在非条件性代码中，将设置 firstPage 的 jsp:setProperty 语句放在 jsp:useBean 的起始标签和结束标签之间。

清单 14.9 给出使用这种方式的 3 个页面中的第一个页面。<http://www.coreservlets.com/> 处的源代码档案还包含其他两个近乎相同的页面。图 14.5 给出典型的结果。

清单 14.8 AccessCountBean.java

```
package coreservlets;

/** Simple bean to illustrate sharing beans through
 * use of the scope attribute of jsp:useBean.
 */
```

```
public class AccessCountBean {  
    private String firstPage;  
    private int accessCount = 1;  
  
    public String getFirstPage() {  
        return(firstPage);  
    }  
  
    public void setFirstPage(String firstPage) {  
        this.firstPage = firstPage;  
    }  
  
    public int getAccessCount() {  
        return(accessCount);  
    }  
  
    public void setAccessCountIncrement(int increment) {  
        accessCount = accessCount + increment;  
    }  
}
```

清单14.9 SharedCounts1.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
<HTML>  
<HEAD>  
<TITLE>Shared Access Counts: Page 1</TITLE>  
<LINK REL=STYLESHEET  
      HREF="JSP-Styles.css"  
      TYPE="text/css">  
</HEAD>  
<BODY>  
<TABLE BORDER=5 ALIGN="CENTER">  
    <TR><TH CLASS="TITLE">  
        Shared Access Counts: Page 1</TH></TR>  
    <TR><TD>  
        <jsp:useBean id="counter"  
                    class="coreservlets.AccessCountBean"  
                    scope="application">  
        <jsp:setProperty name="counter"  
                      property="firstPage"  
                      value="SharedCounts1.jsp" />  
        </jsp:useBean>  
        Of SharedCounts1.jsp (this page),  
        <A HREF="SharedCounts2.jsp">SharedCounts2.jsp</A>, and  
        <A HREF="SharedCounts3.jsp">SharedCounts3.jsp</A>,   
        <jsp:getProperty name="counter" property="firstPage" />  
        was the first page accessed.  
        <P>  
        Collectively, the three pages have been accessed  
        <jsp:getProperty name="counter" property="accessCount" />  
        times.  
        <jsp:setProperty name="counter" property="accessCountIncrement"  
                      value="1" />  
    </TD></TR>  
</TABLE></BODY></HTML>
```

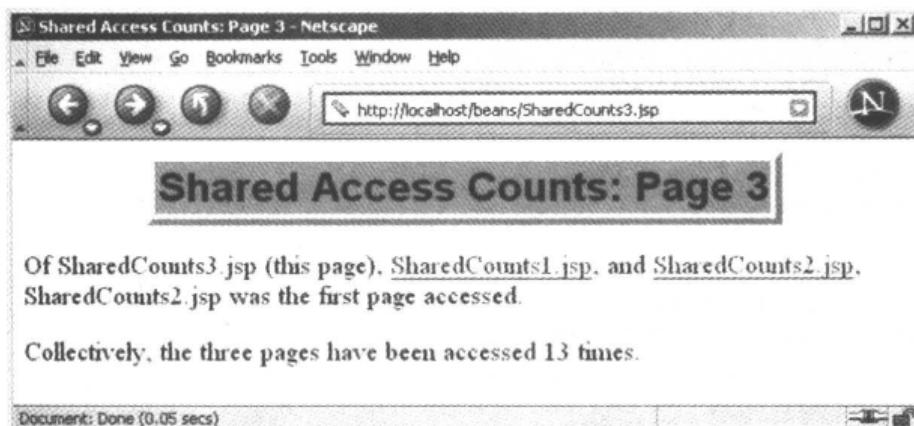


图 14.5 用户访问 SharedCounts3.jsp 的结果。用户访问的第一个页面是 SharedCounts2.jsp。在服务器上一次启动以来，且在图中显示的访问之前，SharedCounts1.jsp，SharedCounts2.jsp 和 SharedCounts3.jsp 总共被访问了 12 次

14.7 共享 bean 的 4 种方式：示例

本节中，我们给出一个扩展的例子，阐述 bean 应用的方方面面。

- 将 bean 用作可以脱离 JSP 页面单独进行测试的实用工具类；
- 使用非共享(作用域为页面)bean；
- 共享作用域为请求的 bean；
- 共享作用域为会话的 bean；
- 共享作用域为应用(即作用域为 ServletContext)的 bean。

在介绍示例之前，需要注意一件事。在不同的作用域中存储 bean 时，每个 bean 一定要使用不同的名字。否则，服务器会发生混乱并取回错误的 bean。

警告

不要为存储在不同位置的 bean 使用相同的 bean 名称。针对每个 bean，`jsp:useBean` 中的 id 都应使用惟一的值。

14.7.1 构建 bean 和 bean 测试器

bean 的基本应用是作为基本的实用工具(辅助)类。我们最强烈地再次说明：除了极短的片断以外，直接插入到 JSP 页面中的 Java 代码，在编写、编译、测试、调试和重用方面，都要比常规的 Java 类要难一些。

例如，清单 14.10 提供一个小型的 Java 对象，它代表一种食品，拥有两项属性：level 和 goesWith(即 4 个方法：getLevel，setLevel，getGoesWith 和 setGoesWith)。也许这个对象太简单了，以至于只查看源代码就足以得出它的实现是否正确。但是，很多时候您在事前都是这样想，但实际上还是会发现 bug。无论什么情况，复杂的类肯定需要测试，因此，清单 14.11 提供了一个测试例程。要注意到，这个实用工具类表示应用程序域的一个值，不依赖于任何 servlet 相关或 JSP 相关的类。因此，它的测试可以完全独立于服务器。清单

14.12 给出一些具有代表性的输出。

清单14.10 BakedBean.java

```
package coreservlets;

/** Small bean to illustrate various bean-sharing mechanisms.*/

public class BakedBean {
    private String level = "half-baked";
    private String goesWith = "hot dogs";

    public String getLevel() {
        return(level);
    }

    public void setLevel(String newLevel) {
        level = newLevel;
    }

    public String getGoesWith() {
        return(goesWith);
    }

    public void setGoesWith(String dish) {
        goesWith = dish;
    }
}
```

清单14.11 BakedBeanTest.java

```
package coreservlets;

/** A small command-line program to test the BakedBean.*/

public class BakedBeanTest {
    public static void main(String[] args) {
        BakedBean bean = new BakedBean();
        System.out.println("Original bean: " +
                           "level=" + bean.getLevel() +
                           ", goesWith=" + bean.getGoesWith());
        if (args.length>1) {
            bean.setLevel(args[0]);
            bean.setGoesWith(args[1]);
            System.out.println("Updated bean: " +
                               "level=" + bean.getLevel() +
                               ", goesWith=" + bean.getGoesWith());
        }
    }
}
```

清单14.12 BakedBeanTest.java的输出

```
Prompt> java coreservlets.BakedBeanTest gourmet caviar
Original bean: level=half-baked, goesWith=hot dogs
Updated bean: level=gourmet, goesWith=caviar
```

14.7.2 使用 scope="page"—不共享

在验证 bean 能够正常工作之后(且只能在此之后), 我们就可以将它用到 JSP 页面中。第一项应用是: 完全在对页面的单个请求中创建、修改和访问 bean。完成这项任务的步骤如下:

- **创建 bean:** 使用 `jsp:useBean`, 且 `scope="page"`(或根本不指定 `scope`, 因为 `page` 是默认值)。
- **修改 bean:** 使用 `jsp:setProperty`, 且 `property="*"`。然后, 提供与 bean 属性的名称相匹配的请求参数。
- **访问 bean:** 使用 `jsp:getProperty`。

清单 14.13 给出一个应用这 3 项技术的 JSP 页面。图 14.6 和图 14.7 说明这个 bean 只在页面的生命期内可用。

清单 14.13 BakedBeanDisplay-page.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Baked Bean Values: page-based Sharing</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Baked Bean Values: page-based Sharing</H1>
<jsp:useBean id="pageBean" class="coreservlets.BakedBean" />
<jsp:setProperty name="pageBean" property="*" />
<H2>Bean level:</H2>
<jsp:getProperty name="pageBean" property="level" /></H2>
<H2>Dish bean goes with:</H2>
<jsp:getProperty name="pageBean" property="goesWith" /></H2>
</BODY></HTML>
```

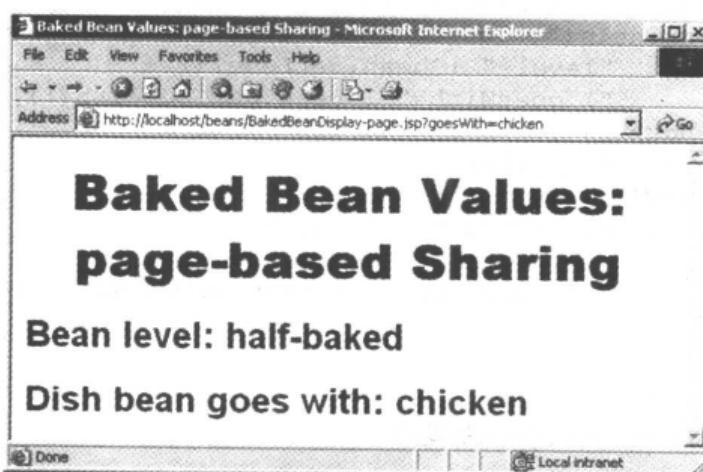


图 14.6 对 BakedBeanDisplay-page.jsp 的初始请求——BakedBean 属性在页面内持续

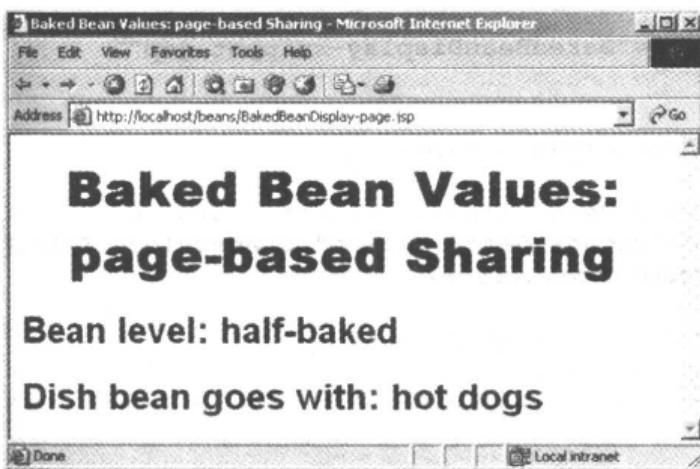


图 14.7 对 BakedBeanDisplay-page.jsp 的后续请求——BakedBean 属性不能跨请求持续

14.7.3 使用基于请求的共享

第二项应用是：在共享同一请求对象的两个不同页面中创建、修改和访问 bean。回顾一下，如果第二个页面由 `jsp:include`, `jsp:forward` 或 `RequestDispatcher` 的 `include` 或 `forward` 方法所调用，则第二个页面共享第一个页面的请求对象。要得到期望的行为，我们使用如下步骤：

- 创建 bean：使用 `jsp:useBean`，且 `scope="request"`；
- 修改 bean：使用 `jsp:setProperty`，且 `property="*"`。然后，提供与 bean 属性的名称相匹配的请求参数；
- 在第一个页面中访问 bean：使用 `jsp:getProperty`。然后，使用 `jsp:include` 调用第二个页面；
- 在第二个页面中访问 bean：用与第一个页面相同的 id 使用 `jsp:useBean`，同样 `scope="request"`。然后，使用 `jsp:getProperty`。

清单 14.14 和清单 14.15 给出应用上面四项技术的 JSP 页面。图 14.8 和图 14.9 阐明在第二个页面中可以访问这个 bean，但这个 bean 并不跨请求保存。

清单 14.14 BakedBeanDisplay-request.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Baked Bean Values: page-based Sharing</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Baked Bean Values: page-based Sharing</H1>
<jsp:useBean id="requestBean" class="coreservlets.BakedBean"
             scope="request"/>
<jsp:setProperty name="requestBean" property="*" />
<H2>Bean level:</H2>
<jsp:getProperty name="requestBean" property="level" /></H2>
<H2>Dish bean goes with:</H2>
```

```
<jsp:getProperty name="requestBean" property="goesWith" /></H2>
<jsp:include page="BakedBeanDisplay-snippet.jsp">
</BODY></HTML>
```

清单 14.15 BakedBeanDisplay-snippet.jsp

```
<H1>Repeated Baked Bean Values: request-based Sharing</H1>
<jsp:useBean id="requestBean" class="coreservlets.BakedBean"
    scope="request"/>
<H2>Bean level:
<jsp:getProperty name="requestBean" property="level" /></H2>
<H2>Dish bean goes with:
<jsp:getProperty name="requestBean" property="goesWith" /></H2>
```

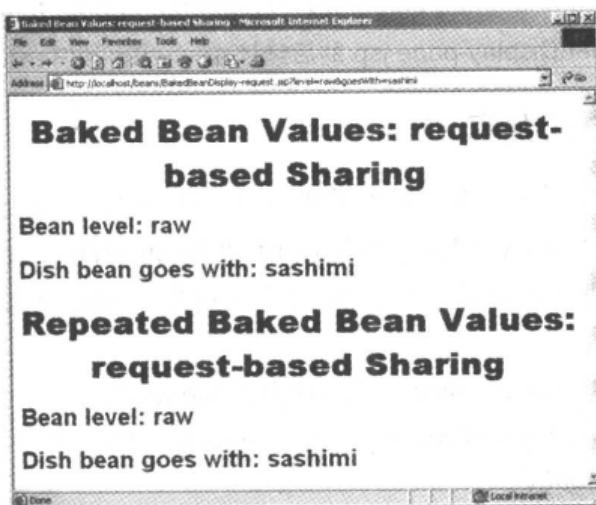


图 14.8 对 BakedBeanDisplay-request.jsp 的初始请求——BakedBean 属性持续到被包括页面中

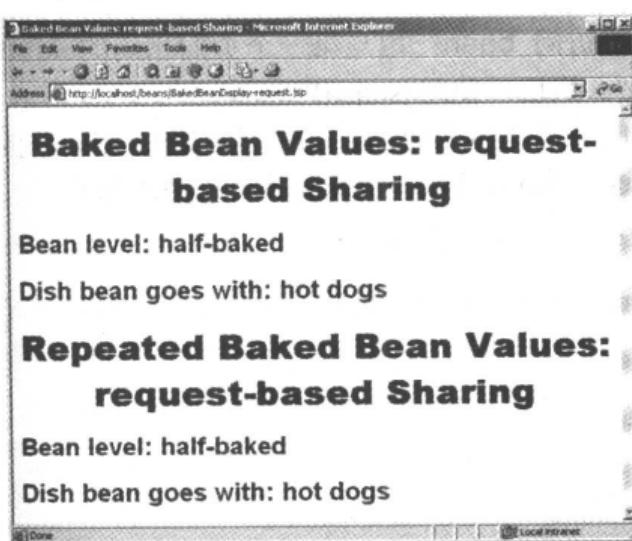


图 14.9 对 BakedBeanDisplay-request.jsp 的后续请求——BakedBean 属性不能跨请求持续

14.7.4 使用基于会话的共享

第三项应用涉及两个部分。首先，我们希望在页面内创建、修改和访问 bean。其次，如果同一客户返回到这个页面，他或她应该看到之前修改过的 bean。这是会话跟踪的典型

案例。因此，要获得期望的行为，我们使用如下步骤：

- **创建 bean:** 使用 `jsp:useBean`, 且 `scope="session"`;
- **修改 bean:** 使用 `jsp:setProperty`, 且 `property="*"`。然后, 提供与 bean 属性的名称相匹配的请求参数;
- **在初始的请求中访问 bean:** 在调用 `jsp:setProperty` 的同一请求中, 使用 `jsp:getProperty`;
- **之后再次访问 bean:** 在不包括请求参数的请求中(从而也就没有调用 `jsp:setProperty`), 使用 `jsp:getProperty`。如果这个请求来自于同一个客户(在会话超时之前), 则会看到之前修改的值。如果这个请求来自于不同的客户(或在会话超时之后), 则看到的是新创建的 bean。

清单 14.16 给出应用这些技术的一个 JSP 页面。图 14.10 给出最初的请求。图 14.11 和图 14.12 阐明, bean 在同一会话中可用, 但在其他会话中则不可用。要注意, 如果多个 JSP 页面中重复相同的 `jsp:useBean` 和 `jsp:getProperty` 代码, 那么, 我们应该得到类似的行为: 只要页面被同一客户访问, 之前的值就会保留。

清单 14.16 BakedBeanDisplay-session.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Baked Bean Values: page-based Sharing</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Baked Bean Values: page-based Sharing</H1>
<jsp:useBean id="sessionBean" class="coreservlets.BakedBean"
             scope="session"/>
<jsp:setProperty name=" sessionBean" property="*" />
<H2>Bean level:</H2>
<jsp:getProperty name=" sessionBean" property="level" /></H2>
<H2>Dish bean goes with:</H2>
<jsp:getProperty name=" sessionBean" property="goesWith" /></H2>
</BODY></HTML>
```

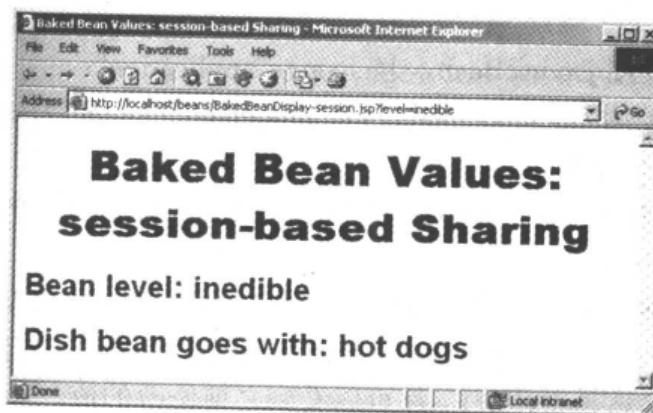


图 14.10 对 BakedBeanDisplay-session.jsp 的初始请求

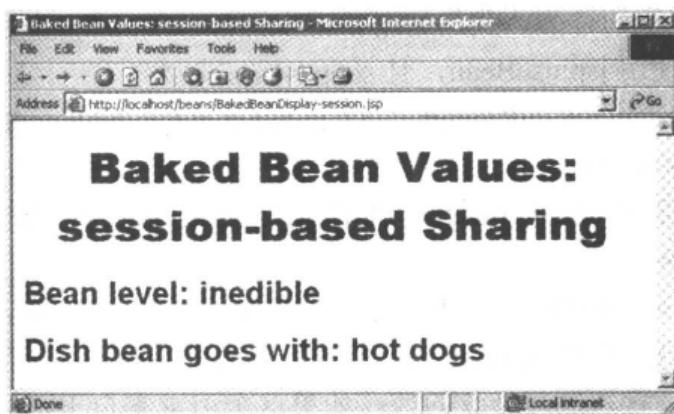


图 14.11 对 BakedBeanDisplay-session.jsp 的后续请求——如果请求来自于同一客户的同一会话，则 BakedBean 属性跨请求持续



图 14.12 对 BakedBeanDisplay-session.jsp 的后续请求——如果请求来自于不同的客户(此处即是)，或来自同一客户但处于不同的会话，则 BakedBean 属性不能跨请求持续

14.7.5 使用基于 ServletContext 的共享

第四项应用，也是最后的应用，也涉及两个部分。首先，我们希望在一个页面内创建、修改和访问 bean。其次，任何客户，如果之后再次访问这个页面，他或她应该看到之前修改过的 bean。除 ServletContext 以外，还有什么能够提供这种全局访问呢？因此，要获得期望的行为，我们使用如下步骤：

- 创建 bean：使用 `jsp:useBean`，且 `scope="application"`；
- 修改 bean：使用 `jsp:setProperty`，且 `property="*"`。然后，提供与 bean 属性的名称相匹配的请求参数；
- 在最初的请求中访问 bean：在调用 `jsp:setProperty` 的同一请求中，使用 `jsp:getProperty`；
- 之后再次访问 bean：在不包含请求参数的请求中(从而也就没有调用过 `jsp:setProperty`)，使用 `jsp:getProperty`。不管这个请求来自于同一个客户还是不同的客户(不管会话是否超时)，都可以看到之前修改的值。

清单 14.17 提供应用这些技术的一个 JSP 页面。图 14.13 给出初始的请求。图 14.14 和图 14.15 阐明，后面多个客户都可以访问到这个 bean。要注意，如果多个 JSP 页面重复

jsp:useBean 和 jsp:getProperty 代码，我们都会得到类似的行为。

清单 14.17 BakedBeanDisplay-application.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Baked Bean Values: page-based Sharing</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Baked Bean Values: page-based Sharing</H1>
<jsp:useBean id="applicationBean" class="coreservlets.BakedBean"
              scope="application"/>
<jsp:setProperty name=" applicationBean" property="*" />
<H2>Bean level:</H2>
<jsp:getProperty name=" applicationBean" property="level" /></H2>
<H2>Dish bean goes with:</H2>
<jsp:getProperty name=" applicationBean" property="goesWith" /></H2>
</BODY></HTML>
```

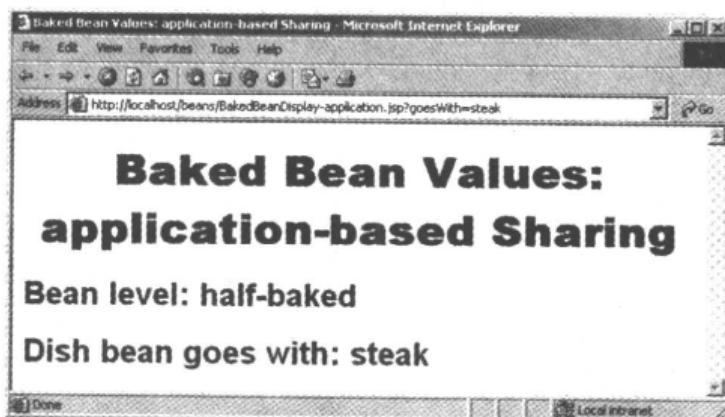


图 14.13 对 BakedBeanDisplay-application.jsp 的初始请求

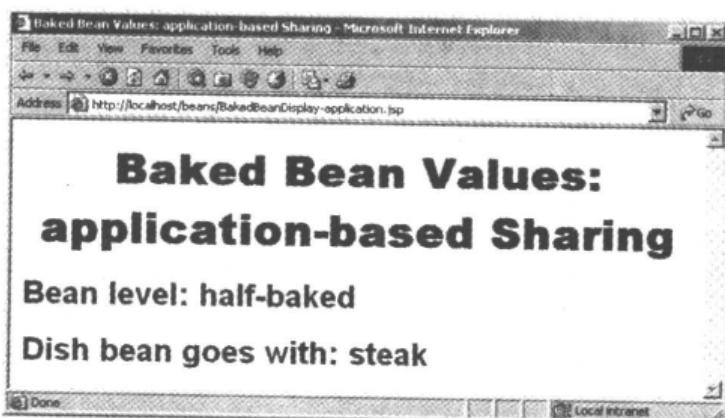


图 14.14 对 BakedBeanDisplay-application.jsp 的后续请求——BakedBean 属性跨请求持续

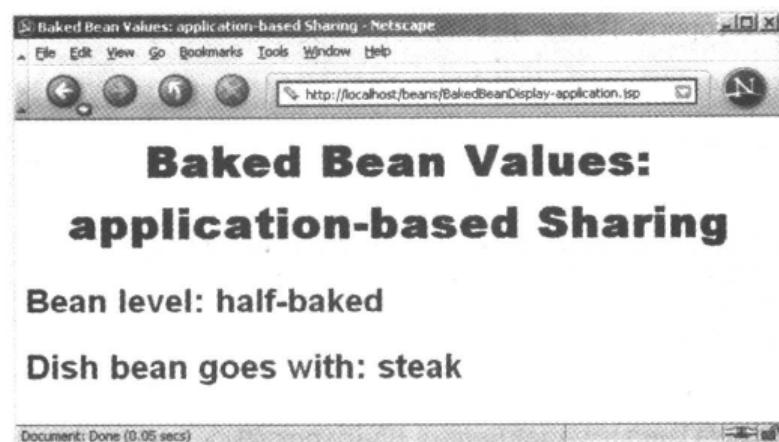


图 14.15 对 BakedBeanDisplay-application.jsp 的后续请求——BakedBean 属性跨请求持续，即使请求来自于不同的客户(本例)或处于不同的会话

第 15 章 servlet 和 JSP 的集成： 模型-视图-控制器构架

本章的主题：

- 模型-视图-控制器(MVC)的优点
- 用 RequestDispatcher 实现 MVC
- 从 servlet 向 JSP 页面转发请求
- 相对 URL 的处理
- 不同显示选项的选取
- 数据共享策略的对比
- 从 JSP 页面转发请求
- 用包含取代转发

servlet 擅长数据处理(processing)，如读取并检查数据，与数据库进行通信，调用商业逻辑，等等。JSP 擅长表示(presentation)，即构建 HTML 来表示请求的结果。本章描述如何组合使用 servlet 和 JSP 页面来解决问题，充分利用每项技术的强项。

15.1 MVC 的需求

如果您的应用需要大量的实际编程来完成任务，servlet 非常适合。如本书前面所述，servlet 能够操作 HTTP 状态代码和报头、使用 cookie、跟踪会话、跨请求保存信息、压缩页面、访问数据库、实时生成 JPEG 图像以及灵活高效地执行许多其他任务。但是，用 servlet 生成 HTML 十分冗长，并且难以修改。

这就是引入 JSP 的原因。如图 15.1 所示，JSP 可以将大部分的表示内容从动态内容中分离出来。通过这种方式，您可以用正常的方式编写 HTML，甚至可以使用 HTML 专有的工具，Web 内容的开发人员可以直接处理 JSP 文档。通过 JSP 表达式、scriptlet 和声明，我们可以向由 JSP 生成的 servlet 中插入简单的 Java 代码；通过 JSP 指令，我们可以控制页面的总体布局。对于更复杂的需求，可以将 Java 代码封装在 bean 中，甚至定义自己的 JSP 标签(tag)。

很好！现在我们已经拥有了所有需要的东西，是这样吗？嗯，不是，还不完全。JSP 文档背后的假定是它能够提供单一整体的表示。那么，如果要依据接收到的数据给出完全不同的结果时，应该怎么做呢？虽然脚本表达式、bean 和定制标签极为强大和灵活，但并不能克服 JSP 页面只能定义相对固定的高层页面外观这一局限。类似地，如果需要复杂的推理来确定适用于当前情形的数据，又该如何呢？JSP 在处理这种类型的商业逻辑方面十分薄弱。

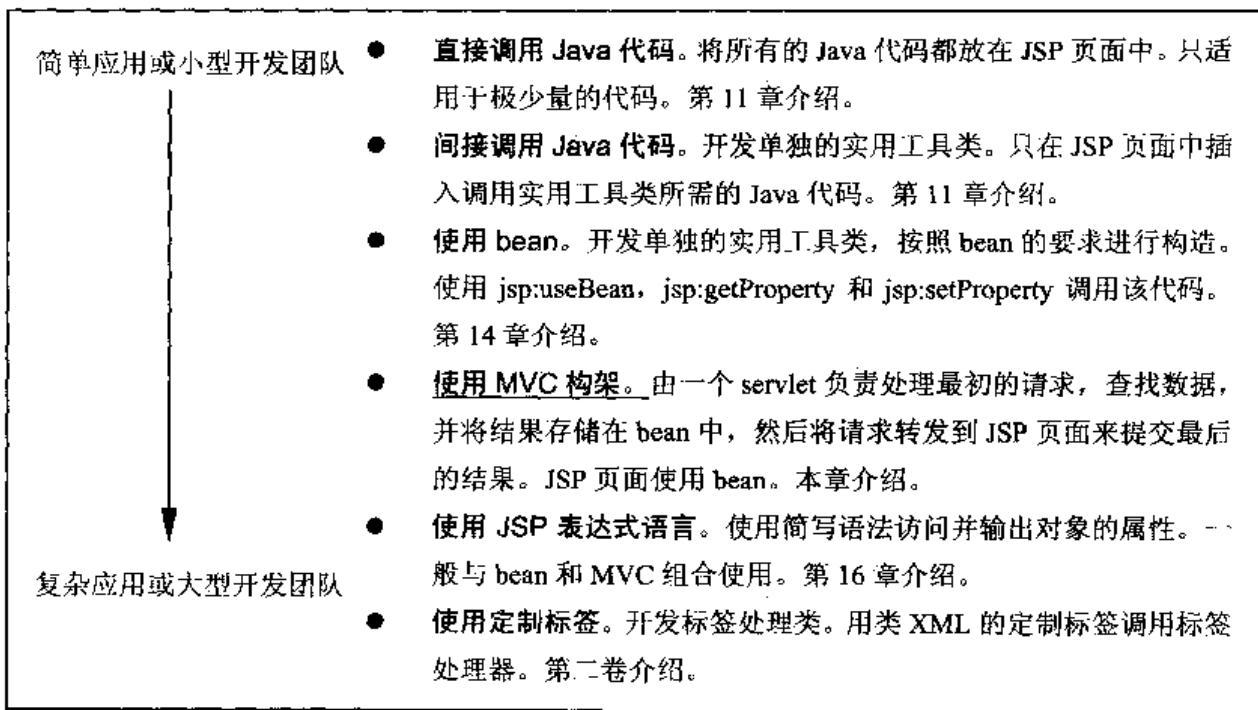


图 15.1 JSP 中调用动态代码的策略

解决这些问题的方案是既使用 servlet，也使用 JSP 页面。通过这种方式——也称为模型-视图-控制器(Model View Controller, MVC)或模型 2(Model 2)构架，可以使每项技术都集中发挥各自的长处。初始的请求由 servlet 来处理。servlet 调用商业逻辑和数据处理代码，并创建 bean 来表示相应的结果(即模型)。然后，servlet 确定哪个 JSP 页面适合于表达这些特定的结果，并将请求转发到相应的页面(JSP 页面即为视图)。由 servlet 确定哪个商业逻辑适用，应该用哪个 JSP 页面表达结果(servlet 就是控制器)。

15.1.1 MVC 框架

使用 MVC 方案的主要驱动力是将创建和操作数据的代码与表达数据的代码分离的愿望。实现这种表示层(presentation-layer)分离的基本工具在 servlet API 中是标准的，同时也是本章的主题。但是，在十分复杂的应用中，使用更为复杂精致的 MVC 框架有时会更有利。这些框架中最流行的是 Apache Struts，本书第二卷中详细论述这一主题。尽管 Struts 很有用且被广泛采用，但实现 MVC 方案并不一定要使用 Struts。对于简单和中等复杂的应用，使用 RequestDispatcher 从零开始实现 MVC 更为直观和灵活。不要强迫自己使用这种方式：应该从简单的方式做起。许多情形中，您的应用可能会在整个生命期内都坚持使用这种基本方案。即使以后您决定使用 Struts 或选择其他 MVC 框架，您的大部分投入也不白费，因为大部分工作也适用于复杂的框架。

15.1.2 构架或方案

术语“构架”常常含有“系统总体设计”之意。尽管许多系统从本质上讲确实是用 MVC 设计的，但并不需要仅仅为了使用 MVC 方案而重新设计整个系统。根本不需要这样做。

实际应用中，常常会用 `servlet` 处理某些请求，用 `JSP` 页面处理另外一些请求，由 `servlet` 和 `JSP` 结合起来共同处理另外的请求，如本章所述。不要以为，为了使用 `MVC` 方案就一定要重写整个系统的构架。应该先做起来，在应用中最适合使用这种构架的地方开始应用它。

15.2 用 `RequestDispatcher` 实现 `MVC`

`MVC` 最重要的概念是将商业逻辑层和数据访问层从表示层分离出来。它的结构也很简单，实际上，您可能早已熟悉了它的大部分内容。下面对所需的步骤进行简短的汇总；随后的小节提供相应的细节。

(1) 定义 `bean` 来表示数据。

从 14.2 节中，您已经知道，`bean` 不过是遵循几项简单约定的 Java 对象。第一步是定义 `bean` 来表示需要显示给用户的结果。

(2) 使用 `servlet` 处理请求。

大多数情况下，由 `servlet` 读取请求参数，如第 4 章所述。

(3) 填写 `bean`。

`servlet` 调用商业逻辑(与应用相关的代码)或数据访问代码(参见第 17 章)，获得最后的结果；然后将结果放置在第(1)步定义的 `bean` 中。

(4) 将 `bean` 存储到请求、会话或 `servlet` 的上下文中。

`servlet` 调用请求、会话或 `servlet` 上下文对象的 `setAttribute` 方法，存储表示请求结果的 `bean` 的引用。

(5) 将请求转发到 `JSP` 页面。

`servlet` 确定哪个 `JSP` 页面适合于当前的情形，并使用 `RequestDispatcher` 的 `forward` 方法将控制转移到那个页面。

(6) 从 `bean` 中提取数据。

`JSP` 页面使用 `jsp:useBean` 以及与第(4)步中的位置相匹配的 `scope`，访问 `bean`。之后，`JSP` 页面使用 `jsp:getProperty` 输出 `bean` 的属性。`JSP` 页面并不创建或修改 `bean`，它只是提取和显示由 `servlet` 创建的数据。

15.2.1 定义 `bean` 来表示数据

`bean` 是遵循几项简单约定的 Java 对象。这种情况下，由于都是 `servlet` 或其他 Java 例程(从不会是 `JSP` 页面)创建 `bean`，所以也就不再需要空(零参数)构造方法。因而，对象只需遵循正常的推荐准则：保持实例变量私有，且使用遵循 `get/set` 命名约定的访问器方法。

由于 `JSP` 页面只需访问 `bean`，不需创建或修改它们，因此，一种惯常的做法是定义值对象(`value object`)：仅用来表示结果的对象，只有很少甚至根本没有任何其他功能。

15.2.2 编写 `servlet` 处理请求

定义了 `bean` 类之后，接下来的任务是编写 `servlet` 来读取请求信息。由于在 `MVC` 构架中由 `servlet` 负责处理初始的请求，因此，我们可以分别使用第 4 章和第 5 章介绍的正常方式读取请求参数和请求报头。第 4 章的 `populateBean` 方法也可以使用，但要注意，这项技

术填写的是表单 bean(表示表单参数的 Java 对象), 不是结果 bean(表示请求结果的 Java 对象)。

尽管 servlet 使用常规的技术读取请求信息并生成数据, 但是, 它们并不使用常规的技术输出结果。实际上, 采用 MVC 方案时, servlet 并不创建任何输出, 输出完全由 JSP 页面来完成, 因而, servlet 并不会调用 response.setContentType, response.getWriter 或 out.println。

15.2.3 填写 bean

读取了表单参数之后, 可以根据它们来确定请求的结果。这些结果的确定完全与具体的应用相关。我们可以调用某种商业逻辑代码, 调用 Enterprise JavaBean 组件, 或者查询数据库。不管对数据进行怎样的处理, 都需要用到这些数据来填写值对象 bean(在第(1)步中定义)。

15.2.4 结果的存储

到此为止, 我们已经读入了表单信息, 已经创建了与请求相关的数据, 并将数据存储在 bean 中。现在, 我们需要将这些 bean 存储在 JSP 页面能够访问的位置。

servlet 主要在 3 个位置存储 JSP 页面所需的数据, 它们是 HttpServletRequest, HttpSession 和 ServletContext。这些存储位置对应 jsp:useBean 的 scope 属性的 3 种非默认值, 即 request, session 和 application。

- 存储仅由 JSP 页面在当前请求中使用的数据。

首先, servlet 依照下面的方式创建和存储数据:

```
ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
```

接下来, servlet 把请求转发给 JSP 页面, JSP 页面使用下面的语句检索数据。

```
<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="request" />
```

要注意, 请求的属性与请求的参数或请求的报头没有任何关系。请求的属性与来自客户的信息无关, 它们只是存储在散列表中的与具体应用相关的数据项, 附加在请求对象之上。这个散列表不过是用来存储数据, 使当前 servlet 和 JSP 页面都可以访问, 但是任何其他资源或请求都不能访问存储在其中的数据。

- 存储当前请求及同一客户的后续请求中由 JSP 页面使用的数据。

首先, servlet 以下面的方式创建和存储数据:

```
ValueObject value = new ValueObject(...);
HttpSession session = request.getSession();
session.setAttribute("key", value);
```

接下来, servlet 将请求转发给 JSP 页面, JSP 页面使用下面的语句检索数据。

```
<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="session" />
```

- 存储当前请求及任意客户的后续请求中由 JSP 页面使用的数据。

首先，servlet 以下面的方式创建和存储数据：

```
ValueObject value = new ValueObject(...);  
getServletContext().setAttribute("key", value);
```

接下来，servlet 将请求转发给 JSP 页面，JSP 页面使用下面的语句检索数据。

```
<jsp:useBean id="key" type="somePackage.ValueObject"  
scope="application" />
```

如 15.3 节所述，一般会对 servlet 代码进行同步，防止 servlet 和 JSP 页面之间数据被更改。

15.2.5 转发请求到 JSP 页面

转发请求使用 RequestDispatcher 的 forward 方法。RequestDispatcher 的获取需要调用 ServletRequest 的 getRequestDispatcher 方法并提供相对地址。我们可以指定 WEB-INF 目录中的地址；虽然 WEB-INF 中的文件不允许客户直接访问，但服务器可以将控制转移到那里。使用 WEB-INF 中的位置可以阻止客户无意中直接访问 JSP 页面(没有经过可以创建 JSP 所需数据的 servlet)。

核心方法

如果 JSP 页面只在由 servlet 生成的数据的上下文中才有意义，则可以将页面放在 WEB-INF 目录中。这样，servlet 可以将请求转发到该页面，但客户不能直接访问它们。

得到 RequestDispatcher 之后，则可以使用 forward 将控制转移到相关的地址，提供 HttpServletRequest 和 HttpServletResponse 作为参数。要注意，RequestDispatcher 的 forward 方法和 HttpServletRequest 的 sendRedirect 方法(7.1 节)有很大的不同。forward 不会像 sendRedirect 那样引入额外的响应/请求对。因而，在使用 forward 时，显示给客户的 URL 不会变化。

重点提示

使用 RequestDispatcher 的 forward 方法时，客户看到的是初始 servlet 的 URL，而非最终 JSP 页面的 URL。

例如，清单 15.1 给出 servlet 的一部分，它根据 operation 请求参数的值，将请求转发给 3 个不同 JSP 页面中的某一个。

清单 15.1 请求转发表示例

```
Request Forwarding Example  
public void doGet(HttpServletRequest request,  
                    HttpServletResponse response)  
throws ServletException, IOException {  
    String operation = request.getParameter("operation");  
    if (operation == null) {  
        operation = "unknown";
```

```
    }
    String address;
    if (operation.equals("order")) {
        address = "/WEB-INF/Order.jsp";
    } else if (operation.equals("cancel")) {
        address = "/WEB-INF/Cancel.jsp";
    } else {
        address = "/WEB-INF/UnknownOperation.jsp";
    }
    RequestDispatcher dispatcher =
        request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
}
```

1. 转发到静态资源

大多数情况下，我们将请求转发给 JSP 页面或其他 servlet。但是，有时，我们会希望将请求发送到静态 HTML 页面。例如，在电子商务网站，表示用户没有合法账户名的请求可能会转发给使用 HTML 表单收集必要信息的账户页面。对于 GET 请求，将请求转发到静态 HTML 页面完全合法，并且不需要特殊的语法，只需将 HTML 页面的地址作为 getRequestDispatcher 的参数。但是，由于转发的请求所使用请求方式与最初的请求相同，因此，不能直接将 POST 请求转发给常规的 HTML 页面。这个问题的解决方案是：只需将 HTML 页面的扩展名改为.jsp。对于 GET 请求，将 somefile.html 改为 somefile.jsp 并不会改变它的输出，但 somefile.html 不能处理 POST 请求，而 somefile.jsp 对于 GET 和 POST 请求都给出等同的响应。

2. 用重定向替代转发

标准的 MVC 方案使用 RequestDispatcher 的 forward 方法将控制从 servlet 转移到 JSP 页面。但是，在使用基于会话的数据共享时，有些情况下，使用 response.sendRedirect 会更合适。

下面是对 forward 的行为的汇总。

- 控制的转移完全在服务器上进行。不涉及任何网络数据流。
- 用户不会看到目的 JSP 页面的地址，而且，我们还可以将页面放在 WEB-INF 中，防止用户不经过建立数据的 servlet，直接访问这些页面。如果 JSP 页面只在由 servlet 生成的数据的上下文中才有意义，则更应该这样做。

下面是对 sendRedirect 的汇总。

- 控制的转移通过向客户发送 302 状态代码和 Location 响应报头来完成。转移需要另外的网络往返。
- 用户能够看到目的页面的地址，并可以记下来，独立地访问。如果将 JSP 设计为数据缺失时使用默认值，这种方式比较适用。例如，重新显示不完全的 HTML 表单，或汇总购物车的内容时，就会使用这种方式。所有的情况下，之前创建的数据都可以从用户的会话中提取出来，甚至对于不涉及 servlet 的请求，这些 JSP 页面也有意义。

15.2.6 从 bean 中提取数据

请求到达 JSP 页面之后，JSP 页面使用 `jsp:useBean` 和 `jsp:getProperty` 提取数据。一般说来，这种方式与第 14 章描述的方式完全相同。但有两处差异：

- JSP 页面从不创建对象。servlet，而非 JSP 页面，应该创建所有的数据对象。因而，为了保证 JSP 页面不会创建对象，应该使用

```
<jsp:useBean ... type="package.Class" />
```

代替

```
<jsp:useBean ... class="package.Class" />w
```

- JSP 页面不应该修改对象。因此，只应该使用 `jsp:getProperty`，不应该用到 `jsp:setProperty`。

所指定的 `scope` 应该与 servlet 使用的存储位置相匹配。例如，下面 3 种形式分别用于基于请求、基于会话和基于应用的共享。

```
<jsp:useBean id="key" type="somePackage.SomeBeanClass"
    scope="request" />
```

```
<jsp:useBean id="key" type="somePackage.SomeBeanClass"
    scope="session" />
```

```
<jsp:useBean id="key" type="somePackage.SomeBeanClass"
    scope="application" />
```

15.3 MVC 代码汇总

本节汇总在基于请求、基于会话和基于应用的 MVC 方案中所使用的代码。

15.3.1 基于请求的数据共享

对于基于请求的共享，servlet 将 bean 存储在 `HttpServletRequest` 中——只能为目的 JSP 页面所访问。

1. servlet

```
ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
RequestDispatcher dispatcher =
    request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

2. JSP 页面

```
<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="request" />
<jsp:getProperty name="key" property="someProperty" />
```

15.3.2 基于会话的数据共享

对于基于会话的共享, servlet 在 HttpSession 中存储 bean, 在同一客户的目的 JSP 页面或其他页面中都可以访问它。

1. servlet

```
ValueObject value = new ValueObject(...);
HttpSession session = request.getSession();
session.setAttribute("key", value);
RequestDispatcher dispatcher =
    request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

2. JSP 页面

```
<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="session" />
<jsp:getProperty name="key" property="someProperty" />
```

15.3.3 基于应用的数据共享

对于基于应用的共享, servlet 将 bean 存储在 ServletContext 中, Web 应用中任何 servlet 和 JSP 页面都可以访问它。为了保证 JSP 页面提取出的数据与 servlet 插入的数据完全相同, 应该采用下述的方式对代码进行同步。

1. servlet

```
synchronized(this) {
    ValueObject value = new ValueObject(...);
    getServletContext().setAttribute("key", value);
    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
    dispatcher.forward(request, response);
}
```

2. JSP 页面

```
<jsp:useBean id="key" type="somePackage.ValueObject"
    scope="application" />
<jsp:getProperty name="key" property="someProperty" />
```

15.4 目的页面中相对 URL 的解释

尽管 servlet 可以将请求转发给同一服务器上的任意位置, 但其过程与使用 HttpServletResponse 的 sendRedirect 方法大不相同。首先, sendRedirect 需要客户连接到新的资源, 而 RequestDispatcher 的 forward 方法完全在服务器上进行处理。其次, sendRedirect 不自动保留所有的请求数据; 而 forward 保留。最后, sendRedirect 产生不同的最终 URL, 而使用 forward 时维护最初 servlet 的 URL。

最后一点表示，如果目的页面使用图像和样式表的相对 URL，那么，这些 URL 应该相对于 servlet 的 URL 或服务器的根目录，不能相对于目的页面的实际位置。以下的样式表项为例：

```
<LINK REF=STYLE
      HREF="my-styles.css"
      TYPE="text/css">
```

如果通过转发请求访问含有这个项目的 JSP 页面，那么，my-styles.css 将会按照相对于初始 servlet 的 URL 进行解释，而不是相对于 JSP 页面自身的 URL，这几乎肯定会导致错误发生。针对这个问题，最简单的解决方案是给出样式表文件在服务器上的完整路径，如下所示。

```
<LINK REF=STYLE
      HREF="/path/my-styles.css"
      TYPE="text/css">
```

对于和中使用的地址，也需要用相同的方式来处理。

15.5 MVC 的应用：银行账户余额

本节中，我们将 MVC 方案用到显示银行账户余额的应用中。作为控制器的 servlet(清单 15.2)读取客户 ID，并将其传递给访问数据的代码，由访问数据的代码返回存储具体值的 bean: BankCustomer(清单 15.3)。然后，这个 servlet 将 bean 存储在 HttpServletRequest 对象中，在那里它可以被目的 JSP 页面访问，而其他页面则不能。如果客户的账户余额为负值，servlet 将请求转发到专为拖欠债务的客户设计的页面(清单 15.4，图 15.2)；如果客户的余额为正值但小于 10 000 美元，servlet 则转移到标准的余额显示页面(清单 15.5，图 15.3)；接下来，如果客户的余额大于等于 10 000 美元，servlet 将请求转发到专为重要客户保留的页面(清单 15.6，图 15.4)；最后，如果客户 ID 不可识别，则显示一个错误页面(清单 15.7，图 15.5)

清单 15.2 ShowBalance.java

```
package coreservlets;

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

/** Servlet that reads a customer ID and displays
 * information on the account balance of the customer
 * who has that ID.
 */

public class ShowBalance extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        BankCustomer customer =
            BankCustomer.getCustomer(request.getParameter("id"));
        String address;
```

```
if (customer == null) {
    address = "/WEB-INF/bank-account/UnknownCustomer.jsp";
} else if (customer.getBalance() < 0) {
    address = "/WEB-INF/bank-account/NegativeBalance.jsp";
    request.setAttribute("badCustomer", customer);
} else if (customer.getBalance() < 10000) {
    address = "/WEB-INF/bank-account/NormalBalance.jsp";
    request.setAttribute("regularCustomer", customer);
} else {
    address = "/WEB-INF/bank-account/HighBalance.jsp";
    request.setAttribute("eliteCustomer", customer);
}
RequestDispatcher dispatcher =
    request.getRequestDispatcher(address);
dispatcher.forward(request, response);
}
```

清单15.3 BankCustomer.java

```
package coreservlets;

import java.util.*;

/** Bean to represent a bank customer.*/

public class BankCustomer {
    private String id, firstName, lastName;
    private double balance;

    public BankCustomer(String id,
                        String firstName,
                        String lastName,
                        double balance) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.balance = balance;
    }

    public String getId() {
        return(id);
    }

    public String getFirstName() {
        return(firstName);
    }

    public String getLastName() {
        return(lastName);
    }

    public double getBalance() {
        return(balance);
    }

    public double getBalanceNoSign() {
```

```

        return(Math.abs(balance));
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    // Makes a small table of banking customers.

    private static HashMap customers;

    static {
        customers = new HashMap();
        customers.put("id001",
                      new BankCustomer("id001",
                                        "John",
                                        "Hacker",
                                        -3456.78));
        customers.put("id002",
                      new BankCustomer("id002",
                                        "Jane",
                                        "Hacker",
                                        1234.56));
        customers.put("id003",
                      new BankCustomer("id003",
                                        "Juan",
                                        "Hacker",
                                        987654.32));
    }

    /**
     * Finds the customer with the given ID.
     * Returns null if there is no match.
     */
}

public static BankCustomer getCustomer(String id) {
    return((BankCustomer)customers.get(id));
}
}

```

清单 15.4 NegativeBalance.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>You Owe Us Money!</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
    We Know Where You Live!</TABLE>
<P>
<IMG SRC="/bank-support/Club.gif" ALIGN="LEFT">
<jsp:useBean id="badCustomer"
              type="coreservlets.BankCustomer"
              scope="request" />

```

```

Watch out,
<jsp:getProperty name="badCustomer" property="firstName" />,
we know where you live.
<P>
Pay us the
$<jsp:getProperty name="badCustomer" property="balanceNoSign" />
you owe us before it is too late!
</BODY></HTML>

```

清单15.5 NormalBalance.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Your Balance</TITLE>
<LINK REL=STYLESCHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Your Balance</TABLE>
<P>
<IMG SRC="/bank-support/Money.gif" ALIGN="RIGHT">
<jsp:useBean id="regularCustomer"
             type="coreservlets.BankCustomer"
             scope="request" />
<UL>
  <LI>First name: <jsp:getProperty name="regularCustomer"
                                         property="firstName" />
  <LI>Last name: <jsp:getProperty name="regularCustomer"
                                         property="lastName" />
  <LI>ID: <jsp:getProperty name="regularCustomer"
                           property="id" />
  <LI>Balance: $<jsp:getProperty name="regularCustomer"
                                         property="balance" />
</UL>
</BODY></HTML>

```

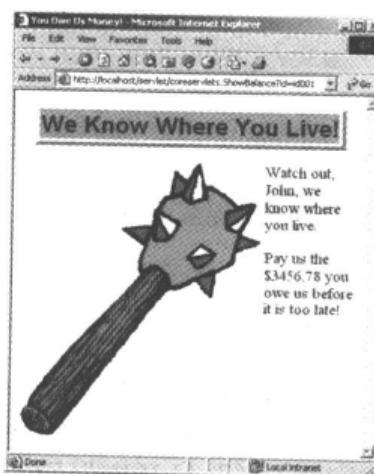


图 15.2 ShowCustomer servlet: ID 对应余额为负值的客户

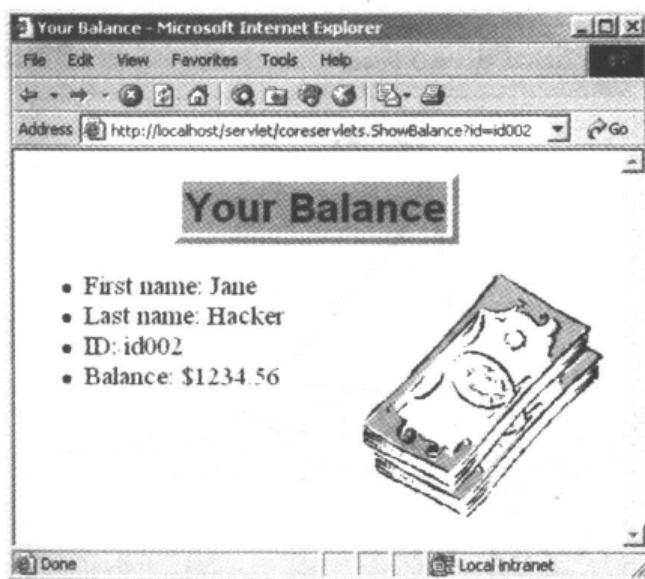


图 15.3 ShowCustomer servlet: ID 对应正常余额的客户

清单15.6 HighBalance.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Your Balance</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Your Balance</TABLE>
<P>
<CENTER><IMG SRC="/bank-support/Sailing.gif"></CENTER>
<BR CLEAR="ALL">
<jsp:useBean id="eliteCustomer"
            type="coreservlets.BankCustomer"
            scope="request" />
It is an honor to serve you,
<jsp:getProperty name="eliteCustomer" property="firstName" />
<jsp:getProperty name="eliteCustomer" property="lastName" />!
<P>
Since you are one of our most valued customers, we would like
to offer you the opportunity to spend a mere fraction of your
$<jsp:getProperty name="eliteCustomer" property="balance" />
on a boat worthy of your status. Please visit our boat store for
more information.
</BODY></HTML>
```

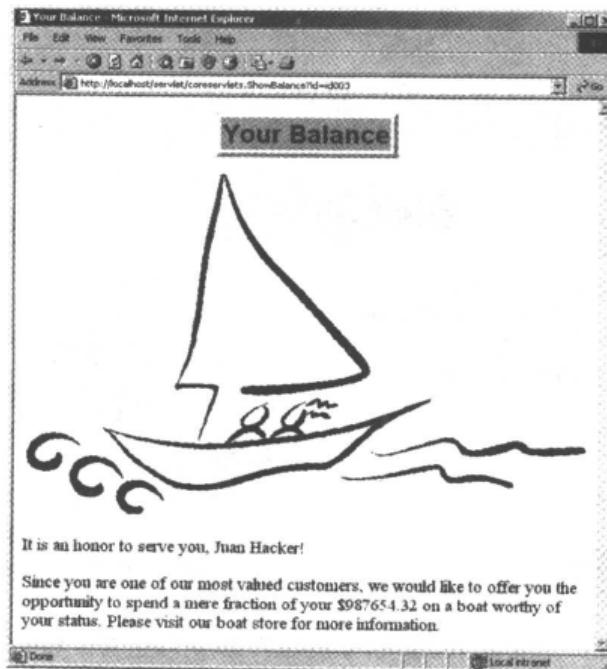


图 15.4 ShowCustomer servlet: ID 对应高余额的客户

清单 15.7 UnknownCustomer.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Unknown Customer</TITLE>
<LINK REL=stylesheet
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Unknown Customer</TH>
  </TR>
<P>
Unrecognized customer ID.
</BODY></HTML>
```

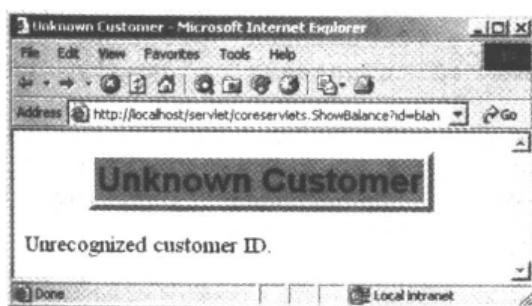


图 15.5 ShowCustomer servlet: 未知客户 ID

15.6 3 种数据共享方式的对比

在 MVC 方案中，由专门的 servlet 负责响应初始请求。这个 servlet 调用读取或创建商业数据的代码，将这些数据放在 bean 中，存储这个 bean，并将请求转发到提供结果的 JSP 页面。但是，这个 servlet 将 bean 存储在什么地方呢？

最常见的答案是，在请求对象中。这也是唯一只能被 JSP 页面所访问的位置。但是，有时，我们可能需要为同一客户保存结果(基于会话的共享)或存储整个 Web 应用广度的数据(基于应用的共享)。

本节针对每种方式都给出一个简短的例子。

15.6.1 基于请求的共享

在这个例子中，我们的目标是向用户显示一个随机数。由于针对每个请求都应该生成新的数字，因此基于请求的共享比较适合。

为了实现这种行为，我们需要一个 bean 来存储数字(清单 15.8)，一个 servlet 用随机数来填充 bean(清单 15.9)，以及一个 JSP 页面来显示结果(清单 15.10，图 15.6)。

清单 15.8 NumberBean.java

```
package coreservlets;

public class NumberBean {
    private double num = 0;

    public NumberBean(double number) {
        setNumber(number);
    }

    public double getNumber() {
        return num;
    }

    public void setNumber(double number) {
        num = number;
    }
}
```

清单 15.9 RandomNumberServlet.java

```
package coreservlets;

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

/** Servlet that generates a random number, stores it in a bean,
 * and forwards to JSP page to display it.
 */

public class RandomNumberServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
```

```

        HttpServletRequest response)
    throws ServletException, IOException {
NumberBean bean = new NumberBean(Math.random());
request.setAttribute("randomNum", bean);
String address = "/WEB-INF/mvc-sharing/RandomNum.jsp";
RequestDispatcher dispatcher =
    request.getRequestDispatcher(address);
dispatcher.forward(request, response);
}
}

```

清单 15.10 RandomNum.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random Number</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<jsp:useBean id="randomNum" type="coreservlets.NumberBean"
              scope="request" />
<H2>Random Number:
<jsp:getProperty name="randomNum" property="number" />
</H2>
</BODY></HTML>

```

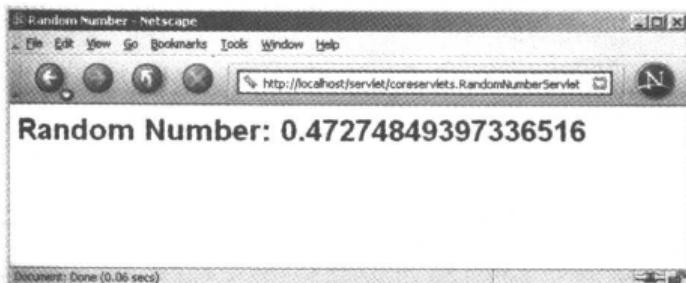


图 15.6 RandomNumberServlet 的结果

15.6.2 基于会话的共享

在这个例子中，我们的目标是显示用户的名和姓。如果用户没有告诉我们他的名字，我们希望使用他们之前给予我们的任何名字。如果用户没有显式地指定一个名字或找不到之前的名字，则显示一段警告。数据为每个客户而存储，因而基于会话的共享比较适合。

为了实现这种行为，我们需要一个 bean 来存储名字(清单 15.11)，一个 servlet 从会话中取得这个 bean 并用名和姓填充它(清单 15.12)，以及一个 JSP 页面来显示结果(清单 15.13，图 15.7 和图 15.8)。

清单 15.11 NameBean.java

```

package coreservlets;

public class NameBean {

```

```
private String firstName = "Missing first name";
private String lastName = "Missing last name";

public NameBean() {}

public NameBean(String firstName, String lastName) {
    setFirstName(firstName);
    setLastName(lastName);
}

public String getFirstName() {
    return(firstName);
}

public void setFirstName(String newFirstName) {
    firstName = newFirstName;
}

public String getLastname() {
    return(lastName);
}

public void setLastName(String newLastName) {
    lastName = newLastName;
}
}
```

清单15.12 RegistrationServlet.java

```
package coreservlets;

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

/** Reads firstName and lastName request parameters and forwards
 * to JSP page to display them. Uses session-based bean sharing
 * to remember previous values.
 */

public class RegistrationServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession();
        NameBean nameBean =
            (NameBean)session.getAttribute("nameBean");
        if (nameBean == null) {
            nameBean = new NameBean();
            session.setAttribute("nameBean", nameBean);
        }
        String firstName = request.getParameter("firstName");
        if ((firstName != null) && (!firstName.trim().equals(""))) {
            nameBean.setFirstName(firstName);
        }
        String lastName = request.getParameter("lastName");
        if ((lastName != null) && (!lastName.trim().equals(""))) {
            nameBean.setLastName(lastName);
        }
    }
}
```

```

    }
    String address = "/WEB-INF/mvc-sharing/ShowName.jsp";
    RequestDispatcher dispatcher =
        request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
}
}

```

清单 15.13 ShowName.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Thanks for Registering</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Thanks for Registering</H1>
<jsp:useBean id="nameBean" type="coreservlets.NameBean"
             scope="session" />
<H2>First Name:<br/>
<jsp:getProperty name="nameBean" property="firstName" /></H2>
<H2>Last Name:<br/>
<jsp:getProperty name="nameBean" property="lastName" /></H2>
</BODY></HTML>

```

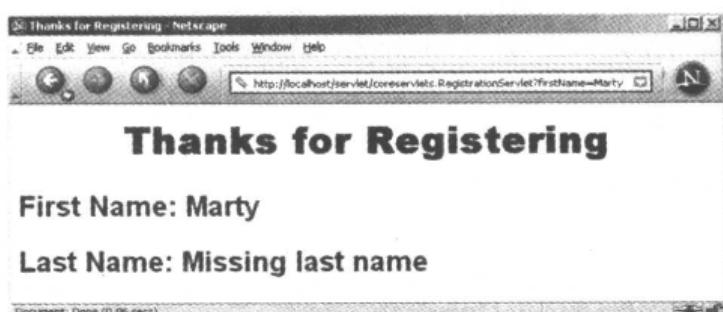


图 15.7 一个参数缺失或没有找到会话数据时 RegistrationServlet 的结果

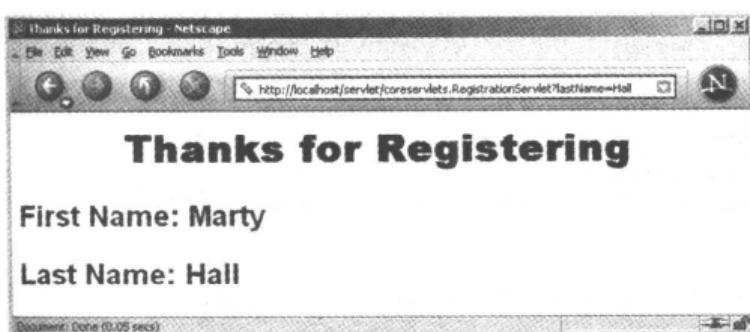


图 15.8 一个参数缺失且找到会话数据时 RegistrationServlet 的结果

15.6.3 基于应用的共享

在这个例子中，我们的目标是显示指定长度的质数。如果用户没有告诉我们期望的长

度，我们希望使用最近为任何用户计算的质数。数据在多个客户间共享，因此基于应用的共享比较适合。

为了实现这种行为，我们需要一个 bean 来存储质数(清单 15.14)——使用在 7.4 节中提供的 Primes 类，一个 servlet 来填写 bean 并将其存储在 ServletContext 中(清单 15.15)，以及一个 JSP 页面显示结果(清单 15.16，图 15.9 和图 15.10)。

清单 15.14 PrimeBean.java

```
package coreservlets;

import java.math.BigInteger;

public class PrimeBean {
    private BigInteger prime;

    public PrimeBean(String lengthString) {
        int length = 150;
        try {
            length = Integer.parseInt(lengthString);
        } catch (NumberFormatException nfe) {}
        setPrime(Primes.nextPrime(Primes.random(length)));
    }

    public BigInteger getPrime() {
        return(prime);
    }

    public void setPrime(BigInteger newPrime) {
        prime = newPrime;
    }
}
```

清单 15.15 PrimeServlet.java

```
package coreservlets;

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

public class PrimeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String length = request.getParameter("primeLength");
        ServletContext context = getServletContext();
        synchronized(this) {
            if ((context.getAttribute("primeBean") == null) ||
                (length != null)) {
                PrimeBean primeBean = new PrimeBean(length);
                context.setAttribute("primeBean", primeBean);
            }
            String address = "/WEB-INF/mvc-sharing>ShowPrime.jsp";
            RequestDispatcher dispatcher =
                request.getRequestDispatcher(address);
            dispatcher.forward(request, response);
        }
    }
}
```

```
    }
}
}
```

清单 15.16 ShowPrime.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>A Prime Number</TITLE>
<LINK REL=STYLESCHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>A Prime Number</H1>
<jsp:useBean id="primeBean" type="coreservlets.PrimeBean"
             scope="application" />
<jsp:getProperty name="primeBean" property="prime" />
</BODY></HTML>
```

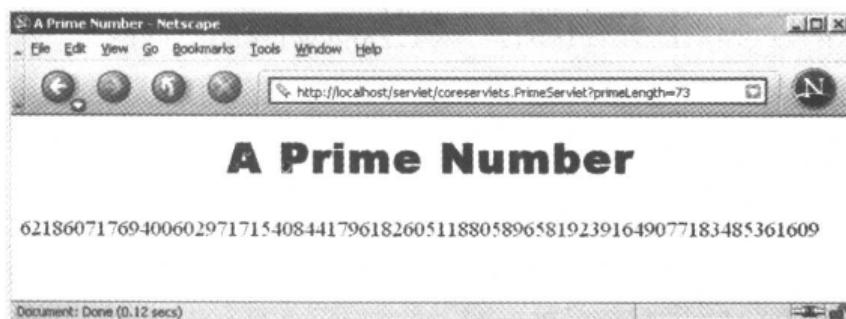


图 15.9 明确给定质数大小时 PrimeServlet 的结果：计算了指定大小的新质数

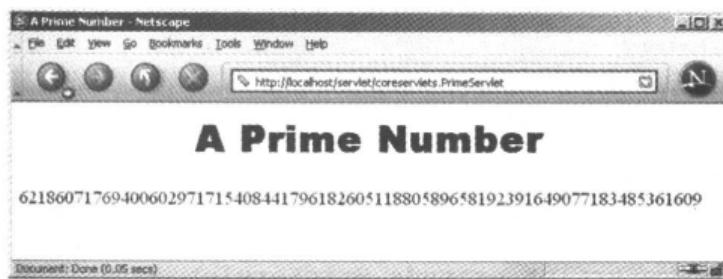


图 15.10 没有明确给定质数大小时 PrimeServlet 的结果：显示之前的数字，没有计算新的质数

15.7 从 JSP 页面转发请求

请求转发最常见的模式是：请求首先到达 servlet，然后由 servlet 将请求转发到 JSP 页面。之所以一般由 servlet 来处理初始的请求，是因为检查请求参数和建立 bean 需要大量的编程工作，在 servlet 中完成这些工作要比在 JSP 文档中要容易得多。目的页面一般为 JSP 文档的原因是：JSP 简化了 HTML 内容的创建过程。

但是，这是常用的方式并不表示它是完成任务的惟一方式。目的页当然可以是 servlet。类似地，JSP 页面也完全有可能将请求转发到其他地方。例如，有些 JSP 页面在正常情况

下提供特定类型的结果，但在接收到意外的值时，将请求转发到其他地方。

如果将请求发送到 servlet，而非 JSP 页面，则不需要对 RequestDispatcher 的应用做任何改动。但是，对于从 JSP 页面转发请求，存在特殊的语法支持。在 JSP 中，使用 `jsp:forward` 动作比将 RequestDispatcher 代码封装到 scriptlet 中的做法要更简单，也更容易。这个动作采用如下的形式：

```
<jsp:forward page="Relative URL"/>
```

由于 `page` 属性中可以含有 JSP 表达式，因此目的地可以在请求期间计算得出。例如，下面的代码将大约一半的访问者转发到 `http://host/examples/page1.jsp`，将其他访问者转发到 `http://host/examples/page2.jsp`。

```
<% String destination;
   if (Math.random() > 0.5) {
      destination = "/examples/page1.jsp";
   } else {
      destination = "/examples/page2.jsp";
   }
%>
<jsp:forward page="<%=\ destination %>" />
```

`jsp:forward` 动作，如同 `jsp:include`，可以使用 `jsp:param` 元素，为目的页面提供额外的请求参数。详细信息参见 13.2 节中有关 `jsp:include` 的论述。

15.8 包含页面

RequestDispatcher 的 `forward` 方法依靠目的 JSP 页面生成全部输出。不允许 servlet 生成任何自己的输出。

转发(forward)的一种替代方案是包含(include)。使用包含，servlet 可以将它自身的输出及一个或多个 JSP 页面的输出组合在一起。更常见的是，servlet 仍旧依靠 JSP 页面生成输出，但 servlet 调用不同的 JSP 页面生成最终页面中不同的部分。听起来熟悉吗？事实是，RequestDispatcher 的 `include` 方法就是 `jsp:include` 动作(13.1 节)在后台调用的代码。

在用 servlet 创建门户网站，由用户选择希望在页面上显示哪种内容时，这种方式最为常用。下面是一个具有代表性的例子。

```
String firstTable, secondTable, thirdTable;
if (someCondition) {
    firstTable = /WEB-INF/Sports-Scores.jsp;
    secondTable = /WEB-INF/Stock-Prices.jsp;
    thirdTable = /WEB-INF/Weather.jsp;
} else if (...) {...}
RequestDispatcher dispatcher =
    Request.getRequestDispatcher(/WEB-INF/Header.jsp);
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher(firstTable);
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher(secondTable);
dispatcher.include(request, response);
```

```
dispatcher =
    request.getRequestDispatcher(thirdTable);
dispatcher.include(request, response);
dispatcher =
    request.getRequestDispatcher(/WEB-INF/Footer.jsp);
dispatcher.include(request, response);
```

第 16 章 简化对 Java 代码的访问： JSP 2.0 表达式语言

本章的主题：

- 使用表达式语言的动力所在
- 调用表达式语言
- 停用表达式语言
- 阻止传统脚本元素的使用
- 表达式语言与 MVC 构架之间的关系
- 作用域变量的引用
- 访问 bean 属性、数组元素、List 元素和 Map 项
- 使用表达式语言运算符
- 表达式的条件求值

为了计算和输出存储在标准位置的 Java 对象的值，JSP 2.0 引入一种简捷的语言。表达式语言(Expression Language, EL)是 JSP 2.0 最重要的两项特性之一；另一个特性是用 JSP 语法(而非 Java 语法)定义定制标签的能力。本章论述表达式语言；定义新标签库的功能在第二卷论述。

警告

JSP 表达式语言不能用在只支持 JSP 1.2 或更早版本的服务器中。

16.1 应用 EL 的驱动力

如图 16.1 所示，JSP 页面可以使用许多不同的策略调用 Java 代码。最好、最灵活的选项之一是 MVC 方案(参见第 15 章)。在这种方案中，由一个 servlet 应答请求；调用恰当的商业逻辑或数据访问代码；将得出的结果放入 bean 中；然后，将 bean 存储在请求、会话或 servlet 上下文中；然后，将请求转发给 JSP 页面，将结果呈现出来。

这种方式的不便之处在于最后的步骤，即在 JSP 页面中呈现结果。我们一般使用 `jsp:useBean` 和 `jsp:getProperty`，但这些元素比较冗长笨拙。此外，`jsp:getProperty` 只支持对简单 bean 属性的访问；如果属性是集合或另外的 bean，那么，对“子属性”的访问则需要用到复杂的 Java 语法(这也是使用 MVC 模式时常常力图避免的东西)。

使用 JSP 2.0 表达式语言可以将表示层简化，它允许我们使用更简短、更易读的项，如下所示：

`${expression}`

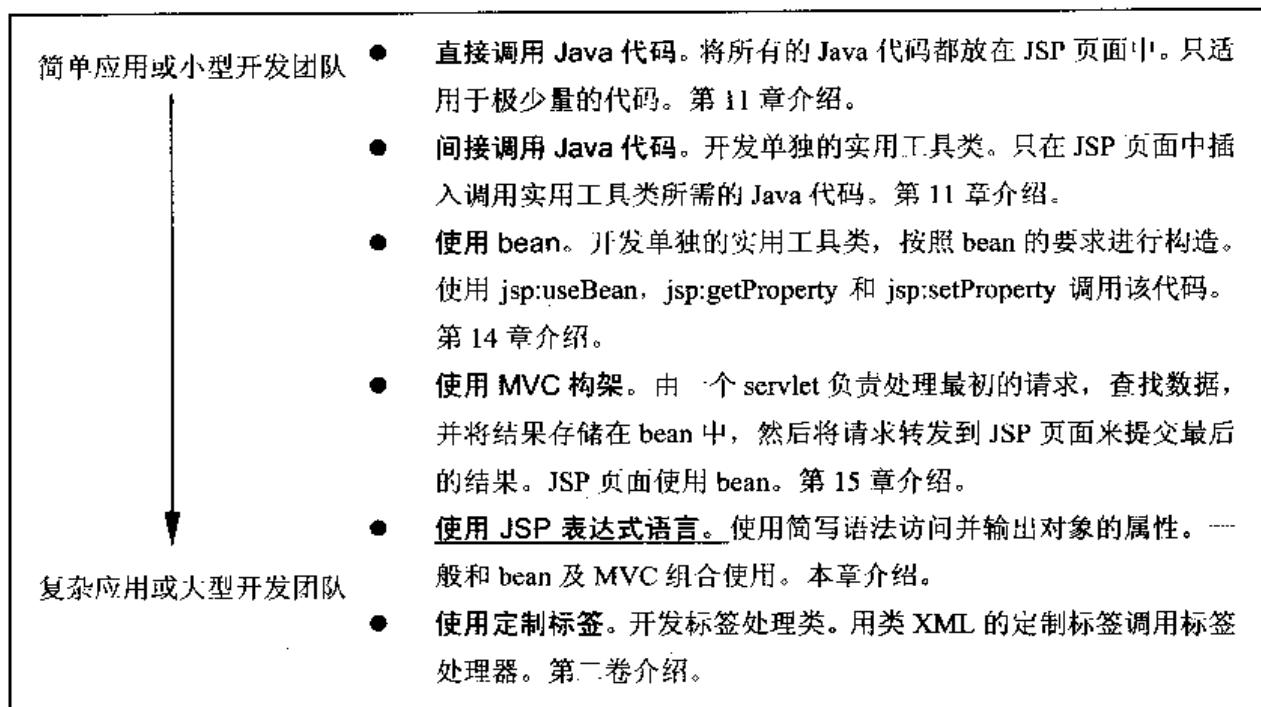


图 16.1 JSP 中调用动态代码的策略

来取代难以维护的 Java 脚本元素，或笨拙的 jsp:useBean 和 jsp:getProperty。

尤其重要的是，表达式语言支持下面的功能：

- **精确地访问存储对象；**
要输出“作用域变量”(用 setAttribute 存储在 PageContext, HttpServletRequest, HttpSession 或 ServletContext 中的对象)saleItem，我们使用 \${saleItem}。参见 16.5 节。
- **bean 属性的简略记法；**
要输出作用域变量 company 的 companyName 属性(即 getCompanyName 方法的结果)，我们使用 \${company.companyName}。而要访问作用域变量 company 的 president 属性的 firstName 属性，我们使用 \${company.president.firstName}。参见 16.6 节。
- **对集合元素的简单访问；**
要访问数组、List 或 Map 的元素，我们使用 \${variable[indexOrKey]}。如果索引或键所采用的形式可以满足合法 Java 变量名的要求，则 bean 的点号记法可以和集合的括号记法互换。参见 16.7 节。
- **对请求参数、cookie 和其他请求数据的简单访问；**
如果要访问标准的请求数据，我们可以使用几个预定义隐式对象。参见 16.8 节。
- **一组为数不多但有效的简单运算符；**
在 EL 表达式内操作对象时，可以使用任何算术、关系、逻辑或空值检查(empty-testing)运算符。参见 16.9 节。

- **条件性输出：**

在进行有选择地输出时，我们可以不必借助于 Java 脚本元素。取而代之，我们可以使用 `${test ? option1 : option2}`来完成这一功能。参见 16.10 节。

- **自动类型转换：**

表达式语言移除了大多数类型转换的需求，可以省略很多将字符串解析成数字的代码。

- **空值取代错误消息。**

大多数情况下，没有相应的值或 `NullPointerExceptions` 异常都会导致空字符串的出现，而非抛出异常。

16.2 表达式语言的调用

在 JSP 2.0 中，我们使用下面形式的元素调用表达式语言。

```
 ${expression}
```

这些 EL 元素可以出现在常规文本和 JSP 标签属性中，只要 JSP 标签的属性允许常规 JSP 表达式。如下所示：

```
<UL>
  <LI>Name: ${expression1}
  <LI>Address: ${expression2}
</UL>
<jsp:include page="${expression3}" />
```

在标签属性中使用表达式语言时，我们可以使用多个表达式(有可能与静态文本混合在一起)，结果被强行拼接成字符串。如下所示：

```
<jsp:include page="${expr1}blah${expr2}" />
```

本章将说明表达式语言的元素在普通文本中的应用。本书第二卷将会说明 EL 元素在标签属性中的应用，这些标签包括您自己编写的标签，以及由 JSP 标准标签库(JSTL)和 JavaServer Faces(JSF)库提供的标签。

转义特殊字符

如果希望 `${}`出现在页面的输出中，在 JSP 页面中需要使用`\${}`。如果希望在 EL 表达式中使用单引号或双引号，则需分别使用`\'`和`\\"`。

16.3 阻止表达式语言的求值

在 JSP 1.2 以及早期的版本中，形如 `${...}`的字符串没有特殊的含义。因此，完全可能出现这种情况，即之前创建的页面中含有 `${}`，而现在这个页面用在支持 JSP 2.0 的服务器中。这种情况下，我们需要停用该页面中的表达式语言。我们有如下 4 种选择：

- **停用整个 Web 应用中的表达式语言：**

我们使用指向 servlet 2.3(JSP 1.2)或更早版本的 `web.xml` 文件完成这项任务。详细

信息参见随后的第一小节。

- **停用多个 JSP 页面中的表达式语言:**

我们使用 web.xml 中的 `jsp-property-group` 元素指定相应的页面。详细信息参见随后的第二小节。

- **停用个别页面中的表达式语言:**

我们使用 `page` 指令的 `isELEnabled` 属性完成这项任务。详细信息参见随后的第三小节。

- **停用表达式语言的个别语句。**

对于需要跨多个 JSP 版本移植 JSP 1.2 页面(不修改 web.xml 文件)，我们可以将\$ 替换为$(对应\$的 HTML 字符实体)。如果 JSP 2.0 页面既含有表达式语言的语句，又含有字面\${字符串，我们可以使用\\$输出\${}。

切记，仅当页面中含有字符序列\${时，才需要用到这些技术。

16.3.1 停用整个 Web 应用中的表达式语言

如果 Web 应用的部署描述文件(即 WEB-INF/web.xml)引用的是 servlet 规范的 2.3 版本或更早的版本(即 JSP 1.2 或更早)，则 JSP 2.0 表达式语言自动在 Web 应用中停用。本书第二卷中会详尽地论述 web.xml 文件，本卷只在 2.11 节中提供了简要的介绍。例如，下面这个 web.xml(虽然没有任何具体的内容，但完全合法)与 JSP 1.2 兼容，同时也表示默认情况下应该将表达式语言停用。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

另一方面，下面的 web.xml 文件与 JSP 2.0 兼容，因而它规定默认情况下应该激活表达式语言。(所有这些 web.xml 文件，和本书中其他的代码示例一样，都可以从 <http://www.coreservlets.com>/本书的源代码档案中下载。)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  "http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">
</web-app>
```

16.3.2 停用多个 JSP 页面中的表达式语言

在部署描述文件指定 servlet 2.4(JSP 2.0)的 Web 应用中，我们使用 web.xml 文件的 `jsp-property-group` 元素的 `el-ignored` 子元素来指定应该忽略哪些页面内的表达式语言。下面给出的例子停用 legacy 目录下所有 JSP 页面的表达式语言。

```
<?xml version=1.0 encoding=ISO-8859-1?>
<web-app xmlns=http://java.sun.com/xml/ns/j2ee>
```

```
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=
    http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd
version=2.4>
<jsp-property-group>
    <url-pattern>/legacy/*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
</jsp-property-group>
</web-app>
```

`jsp-property-group` 元素在本书第二卷中详细论述。

16.3.3 停用个别页面中的表达式语言

如果要在个别页面内禁用 EL 求值，可以将 `page` 指令的 `isELEnabled` 属性设为 `false` 值，如下所示。

```
<%@ page isELEnabled="false" %>
```

要注意，`isELEnabled` 是 JSP 2.0 新引入的属性，不能在只支持 JSP 1.2 或更早版本的服务器中使用它。因此，我们不能使用这项技术让同一 JSP 页面不加修改地运行在新的和较老的服务器上。因此，`jsp-property-group` 元素一般是比 `isELEnabled` 属性更好的选择。

16.3.4 停用个别表达式语言语句

假定您希望将含有 `${}` 的 JSP 1.2 页面用到多个地方。特别地，您希望将它用在 JSP 1.2 Web 应用中，或是含有用到表达式语言的页面的 Web 应用中。您希望既不对页面做出任何修改，也不对 `web.xml` 文件做出任何修改，直接将该页面用到任何 Web 应用中。虽然这种情况不容易出现，但有时确实会发生，之前讨论的所有构造都不能达到这个目的。在这种情况下，只需将`$` 替换成相应的 HTML 字符实体(对应 ISO 8859-1 值`$(36)`)。因此，处理这种需求的办法是在整个页面内将`$` 替换为`$`。例如，

```
&#36;{blah}
```

将会可移植地向用户显示

```
 ${blah}
```

但要注意，字符实体是被浏览器转换成`$`，不是由服务器完成，因此，这项技术只能用在向 Web 浏览器输出 HTML 的情况下。

最后，假定您有一个 JSP 2.0 页面，它既含有表达式语言语句，又含有字面的 `${}` 字符串。这种情况下，只需在`$` 符号前面加上反斜杠。因而，

```
\${1+1} is ${1+1}
```

将会输出

```
\${1+1} is 2
```

16.4 阻止标准脚本元素的使用

JSP 表达式语言提供对作用域变量(存储在标准位置的 Java 对象)的简洁和易读访问。这个功能消除了大部分使用显式 Java 脚本元素(第 11 章介绍)的需求。实际上,一些开发人员更喜欢在他们的整个项目中都使用非传统脚本元素的方式。我们可以使用 `jsp-property-group` 的 `scripting-invalid` 子元素来强制执行这项约束。例如,下面的 `web.xml` 文件表示,在任何 JSP 页面中使用传统的脚本元素都会导致错误发生。

```
<?xml version=1.0 encoding=ISO-8859-1?>
<web-app xmlns=http://java.sun.com/xml/ns/j2ee
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation=
        http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd
    version=2.4>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <scripting-invalid>true</scripting-invalid>
    </jsp-property-group>
</web-app>
```

16.5 访问作用域变量

在使用 MVC 方案(第 15 章)时,由一个 `servlet` 调用创建数据的代码,之后使用 `RequestDispatcher.forward` 或 `response.sendRedirect` 将控制权转移给相应的 JSP 页面。为了让 JSP 页面能够访问到这些数据,该 `servlet` 需要使用 `setAttribute` 将数据存储在某个标准位置: `HttpServletRequest`, `HttpSession` 或 `ServletContext`。

处于这些位置的对象称作是“作用域变量”,使用表达式语言可以快速容易地访问这些对象。我们也可以将作用域变量存储在 `PageContext` 对象中,但由于 `servlet` 和 JSP 页面不共享 `PageContext` 对象,所以这样做没有什么用处。因此,作用域限于页面(`page-scoped`)的变量仅限于同一 JSP 页面之前存储的对象,不适合 `servlet` 存储的对象。

要输出作用域变量的值,只需在表达式语言元素中使用它的名字。例如,下面的语句

```
 ${name}
```

表示在 `PageContext`, `HttpServletRequest`, `HttpSession` 和 `ServletContext`(依照此处列出的次序)中查找名为 `name` 的属性。如果找到该属性,则调用它的 `toString` 方法并返回调用的结果。如果没有找到任何东西,则返回空字符串(不是 `null` 或错误消息)。因此,下面两个表达式是等同的。

```
 ${name}
<%= pageContext.getAttribute("name") %>
```

后者存在的问题是它过于冗长,并且需要显式的 Java 语法。当然,略去这些 Java 代码是可能的,但却需要用到更为冗长的 `jsp:useBean` 代码,如下所示。

```
<jsp:useBean id="name" type="somePackage.SomeClass" scope="...">
<%= name %>
```

此外，使用 `jsp:useBean` 时，还必须知道 `servlet` 使用的是哪个作用域，还必须知道属性的完全限定类名。这极为不方便，尤其是在 JSP 页面的创作者并非 `servlet` 的创作者的情况下。

16.5.1 属性名的选取

使用 JSP 表达式语言访问作用域变量时，所选取的属性名必须能够作为合法的 Java 变量名。因此，要避免使用那些可以出现在字符串中，但不允许出现在变量名中的字符，如小数点、空格、短划线等。

同时，还要避免使用与 16.8 节给出的与预定义属性名相冲突的属性名。

16.5.2 示例

为了说明对作用域变量的访问，`ScopedVars` `servlet`(清单 16.1)在 `HttpServletRequest` 中存储了一个 `String` 对象，在 `HttpSession` 中存储了另一个 `String` 对象，并在 `ServletContext` 中存储了一个 `Date` 对象。该 `servlet` 将请求转发给一个 JSP 页面，这个 JSP 页面使用 `${attributeName}` 访问并输出这些对象。

需要注意，不管这些属性存储在哪个作用域中，该 JSP 页面都使用同样的语法访问它们。因为 MVC 的 `servlet` 所存储的对象几乎总是使用惟一的属性名，因此，这是一项方便的特性。但是，使用重复的属性名在技术上是可行的，因此，一定要牢记，表达式语言以 `PageContext`, `HttpServletRequest`, `HttpSession` 和 `ServletContext` 的次序进行搜索。为了说明这一点，该 `servlet` 在 3 个共享作用域中都存储一个对象，3 种情况下都使用属性名 `repeated`。 `${repeated}` 返回的值(参见图 16.2)是以预先定义好的次序搜索各个作用域时找到的第一个属性的值(本例中这个作用域为 `HttpServletRequest`)。如果想限定搜索特定的作用域，请参阅 16.8 节。

清单 16.1 `ScopedVars.java`

```
package coreservlets;

/** Servlet that creates some scoped variables (objects stored
 * as attributes in one of the standard locations). Forwards
 * to a JSP page that uses the expression language to
 * display the values.
 */

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

public class ScopedVars extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        request.setAttribute("attribute1", "First Value");
        HttpSession session = request.getSession();
        session.setAttribute("attribute2", "Second Value");
        ServletContext application = getServletContext();
```

```

application.setAttribute("attribute3",
    new java.util.Date());
request.setAttribute("repeated", "Request");
session.setAttribute("repeated", "Session");
application.setAttribute("repeated", "ServletContext");
RequestDispatcher dispatcher =
    request.getRequestDispatcher("/el/scoped-vars.jsp");
dispatcher.forward(request, response);
}
}

```

清单16.2 scoped-vars.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Accessing Scoped Variables</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Accessing Scoped Variables
  </TH></TR>
  <TR><TD>
    <P>
      <UL>
        <LI><B>attribute1:</B> ${attribute1}
        <LI><B>attribute2:</B> ${attribute2}
        <LI><B>attribute3:</B> ${attribute3}
        <LI><B>Source of "repeated" attribute:</B> ${repeated}
      </UL>
    </TD>
  </TR>
</TABLE>
</BODY></HTML>

```

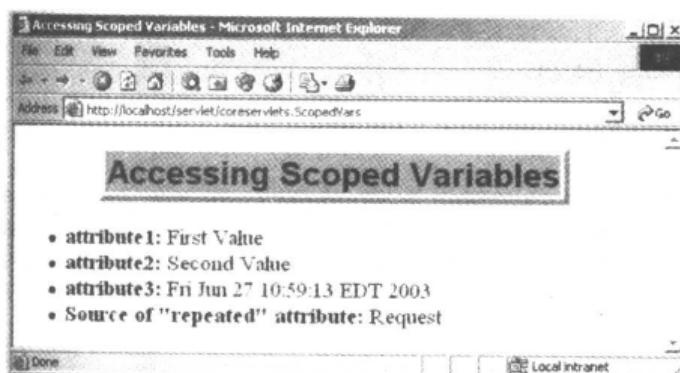


图 16.2 JSP 2.0 表达式语言简化了对作用域变量(作为属性存储在页面、请求、会话或 servlet 上下文中的对象)的访问

16.6 访问 bean 的属性

当只是使用 \${name} 时，系统查找名为 name 的对象，将它强制转换成 String 后返回。尽管这种行为比较方便，但是，少数情况下，我们可能需要它输出 MVC servlet 存储的实际对象，而不是我们通常所希望的那样输出对象的个别属性。

JSP 表达式语言为访问 bean 的属性提供一种简单但十分强大的点号记法。如果要返回某个作用域变量 customer 的 firstName 属性，只需使用 \${customer.firstName}。虽然这种形式看似十分简单，但是，为了支持这种行为，系统必须执行反射动作(分析对象的内在结构)。因此，假定对象属于 NameBean 类型且 NameBean 在 coreservlets 包中，如果要用显式的 Java 语法完成相同的事情，我们需要将

```
${customer.firstName}
```

替换为

```
<%@ page import="coreservlets.NameBean" %>
<%
NameBean person =
    (NameBean)pageContext.findAttribute("customer");
%>
<%= person.getFirstName() %>
```

此外，如果没有找到指定的属性，上面的代码中，使用 JSP 表达式语言的版本返回空字符串，而同样的情况下，使用脚本元素的版本需要另外的代码以避免 NullPointerException 异常。

对于表达式语言的应用，JSP 脚本元素并非唯一的替代方案。只要知道作用域和完全限定类名，我们就可以将下面的语句

```
${customer.firstName}
```

替换为

```
<jsp:useBean id="customer" type="coreservlets.NameBean"
               scope="request, session, or application" />
<jsp:getProperty name="customer" property="firstName" />
```

但是，表达式语言允许我们任意嵌套属性。例如，如果 NameBean 类有一个 address 属性(即 getAddress 方法)，它返回 Address 对象，而 Address 对象拥有 zipCode 属性(即 getZipCode 方法)，我们可以使用下面的简单形式访问这个 zipCode 属性。

```
${customer.address.zipCode}
```

jsp:useBean 和 jsp:getProperty 的组合不能在不使用显式 Java 语法的情况下完成同样的事情。

16.6.1 点号记法与数组记法的等同性

最后，要注意，使用表达式语言，我们可以用数组记法(方括号)替代点号记法。因此，我们可以将下面的语句

```
${name.property}
```

替换为

```
${name["property"]}
```

在访问 bean 的属性时，很少使用第二种形式。但是，它的确有两项优势。

首先，它允许我们在请求期间计算属性的名称。使用数组记法时，方括号内的值本身

就可以是一个变量；而使用点号记法时，属性名必须是字面值。

其次，数组记法允许我们使用那些不能成为合法属性名的值。在访问 bean 的属性时，这没有任何用处，但在访问集合(16.7 节)或请求报头(16.8 节)时，它很有用。

16.6.2 示例

为了说明 bean 属性的应用，请考虑清单 16.3 中给出的 BeanProperties servlet。这个 servlet 创建一个 EmployeeBean(清单 16.4)，将它以属性名 employee 存储在请求对象中，并将请求转发给 JSP 页面。

EmployeeBean 类拥有两个属性：name 和 company，分别为 NameBean(清单 16.5)和 CompanyBean(清单 16.6)对象。NameBean 有两个属性：firstName 和 lastName；CompanyBean 也有两个属性，分别为 companyName 和 business。JSP 页面(清单 16.7)使用下面的简单表达式语言访问这 4 个属性：

```
 ${employee.name.firstName}
 ${employee.name.lastName}
 ${employee.company.companyName}
 ${employee.company.business}
```

图 16.3 给出相应的结果。

清单 16.3 BeanProperties.java

```
package coreservlets;

/** Servlet that creates some beans whose properties will
 * be displayed with the JSP 2.0 expression language.
 */

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

public class BeanProperties extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        NameBean name = new NameBean("Marty", "Hall");
        CompanyBean company =
            new CompanyBean("coreservlets.com",
                           "J2EE Training and Consulting");
        EmployeeBean employee = new EmployeeBean(name, company);
        request.setAttribute("employee", employee);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/bean-properties.jsp");
        dispatcher.forward(request, response);
    }
}
```

清单 16.4 EmployeeBean.java

```
package coreservlets;

public class EmployeeBean {
```

```
private NameBean name;
private CompanyBean company;

public EmployeeBean(NameBean name, CompanyBean company) {
    setName(name);
    setCompany(company);
}

public NameBean getName() { return(name); }

public void setName(NameBean newName) {
    name = newName;
}

public CompanyBean getCompany() { return(company); }

public void setCompany(CompanyBean newCompany) {
    company = newCompany;
}
}
```

清单16.5 NameBean.java

```
package coreservlets;

public class NameBean {
    private String firstName = "Missing first name";
    private String lastName = "Missing last name";

    public NameBean() {}

    public NameBean(String firstName, String lastName) {
        setFirstName(firstName);
        setLastName(lastName);
    }

    public String getFirstName() {
        return(firstName);
    }

    public void setFirstName(String newFirstName) {
        firstName = newFirstName;
    }

    public String getLastName() {
        return(lastName);
    }

    public void setLastName(String newLastName) {
        lastName = newLastName;
    }
}
```

清单16.6 CompanyBean.java

```
package coreservlets;

public class CompanyBean {
```

```

private String companyName;
private String business;

public CompanyBean(String companyName, String business) {
    setCompanyName(companyName);
    setBusiness(business);
}

public String getCompanyName() { return(companyName); }

public void setCompanyName(String newCompanyName) {
    companyName = newCompanyName;
}

public String getBusiness() { return(business); }

public void setBusiness(String newBusiness) {
    business = newBusiness;
}
}

```

清单 16.7 bean-properties.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Accessing Bean Properties</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
    Accessing Bean Properties
</TABLE>
<P>
<UL>
    <LI><B>First Name:</B> ${employee.name.firstName}
    <LI><B>Last Name:</B> ${employee.name.lastName}
    <LI><B>Company Name:</B> ${employee.company.companyName}
    <LI><B>Company Business:</B> ${employee.company.business}
</UL>
</BODY></HTML>

```

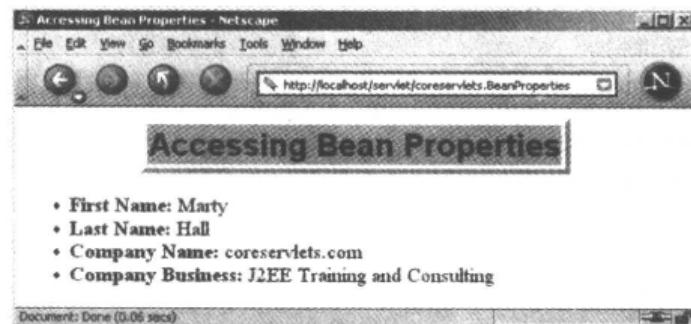


图 16.3 使用点号记法或数组记法访问 bean 的属性

16.7 访问集合

JSP 2.0 表达式语言允许我们使用相同的方式(数组记法)访问不同类型的集合。例如，无论作用域变量 `attributeName` 指向数组、`List` 或 `Map`，我们都使用下面的方式访问集合中的项：

```
${attributeName[entryName]}
```

如果作用域变量为数组，则 `entryName` 为索引，相应的值通过 `theArray[index]` 获取。例如，如果 `customerNames` 指向字符串数组，则下面的表达式会输出数组的第一项。

```
${customerNames[0]}
```

如果作用域变量为实现了 `List` 接口的对象，则 `entryName` 为索引，相应的值通过 `theList.get(index)` 获取。例如，如果 `supplierNames` 指向 `ArrayList`，则下面的表达式会输出 `ArrayList` 的第一项。

```
${supplierNames[0]}
```

如果作用域变量是实现了 `Map` 接口的对象，则 `entryName` 为键，相应的值通过 `theMap.get(key)` 获取。例如，如果 `stateCapitals` 指向 `HashMap`(键为美国州名，值为城市名)，则下面的表达式会返回"annapolis"。

```
${stateCapitals["maryland"]}
```

如果 `Map` 的键满足 Java 变量名的相关规定(比如不能有小数点、空格、短划线等字符)，则可以用点号记法来取代数组记法。此时，我们可以将前面的例子改写为：

```
${stateCapitals.maryland}
```

但要注意，数组记法允许我们在请求期间对键进行选取，而点号记法需要我们预先知道相应的键。

示例

为了说明 EL 表达式在访问集合方面的应用，`Collections servlet`(清单 16.8)创建一个字符串数组，一个 `ArrayList` 和一个 `HashMap`。然后，这个 `servlet` 将请求转发给 JSP 页面(清单 16.9，图 16.4)，由 JSP 页面使用统一的数组记法访问所有这 3 个对象中的元素。

由于纯数字型的值也可以作为 `bean` 的属性，因此，这个数组和 `ArrayList` 中的项都必须使用数组记法进行访问。但是，`HashMap` 中的项既可以使用数组记法进行访问，也可以使用点号记法。为了保持一致，我们统一采用数组记法，然而，清单 16.9 中下面形式的表达式

```
${company["Ellison"]}
```

可以替换为

```
${company.Ellison}
```

清单16.8 Collections.java

```

package coreservlets;

import java.util.*;

/** Servlet that creates some collections whose elements will
 * be displayed with the JSP 2.0 expression language.
 */

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

public class Collections extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        String[] firstNames = { "Bill", "Scott", "Larry" };
        ArrayList lastNames = new ArrayList();
        lastNames.add("Ellison");
        lastNames.add("Gates");
        lastNames.add("McNealy");
        HashMap companyNames = new HashMap();
        companyNames.put("Ellison", "Sun");
        companyNames.put("Gates", "Oracle");
        companyNames.put("McNealy", "Microsoft");
        request.setAttribute("first", firstNames);
        request.setAttribute("last", lastNames);
        request.setAttribute("company", companyNames);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/collections.jsp");
        dispatcher.forward(request, response);
    }
}

```

清单16.9 collections.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Accessing Collections</TITLE>
<LINK REL=stylesheet
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Accessing Collections
  </TH></TR>
</TABLE>
<P>
<UL>
  <LI>${first[0]} ${last[0]} (${company["Ellison"]})
  <LI>${first[1]} ${last[1]} (${company["Gates"]})
  <LI>${first[2]} ${last[2]} (${company["McNealy"]})
</UL>
</BODY></HTML>

```

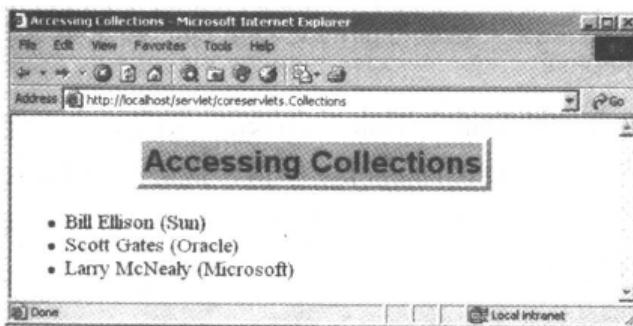


图 16.4 使用数组记法或点号记法访问集合。仅当键可以作为合法的 Java 变量名时才可以使用点号记法

16.8 引用隐式对象

大多数情况下，我们可以将 JSP 表达式与模型-视图-控制器构架(由 servlet 创建数据，由 JSP 页面表示数据，第 15 章)结合使用。这种情况下，JSP 页面一般只受 servlet 所创建的对象的影响，因此，通常的 bean 访问和集合访问机制就够用了。

但是，表达式语言并不仅仅局限于在 MVC 方案中的应用；如果服务器支持 JSP 2.0，并且 web.xml 文件指出 servlet 2.4，那么，表达式语言可以用在任何 JSP 页面中。为了使得表达式语言的使用更有效率，相应的规范定义了下面的隐式对象(implicit object)。

(1) pageContext

pageContext 对象引用当前页面的 PageContext。PageContext 类依次拥有 request, response, session, out 和 servletContext 属性(即 getRequest, getResponse, getSession, getOut 和 getServletContext 方法)。例如，下面的表达式输出当前的会话 ID。

```
 ${pageContext.session.id}
```

(2) param 和 paramValues

这些对象允许我们访问基本的请求参数值(param)或请求参数值的数组(paramValues)。因此，下面的表达式输出 custID 请求参数的值(如果当前请求中不存在这个参数，则返回空字符串，而非 null)。

```
 ${param.custID}
```

有关请求参数处理方面的更多信息，参见第 4 章。

(3) header 和 headerValues

这些对象分别访问 HTTP 请求报头的主要值以及全部值。回顾一下，如果点号后面的值不能作为合法的名称，则不能使用点号记法。因此，要访问 Accept 报头，我们可以使用

```
 ${header.Accept}
```

或

```
 ${header["Accept"]}
```

但要访问 Accept-Encoding 报头，则必须使用

```
 ${header["Accept-Encoding"]}
```

有关 HTTP 请求报头处理方面的更多信息，参见第 5 章。

(4) cookie

cookie 对象允许我们快捷地引用输入 cookie。但是，它的返回值是 Cookie 对象，不是 cookie 的值。如果要访问 cookie 的值，则需使用 Cookie 类标准的 value 属性(即 getValue 方法)。因此，下面两行代码都输出 userCookie 的值(如果没有找到这个 cookie，则输出空字符串)。

```
${cookie.userCookie.value}
${cookie["userCookie"].value}
```

有关 cookie 应用方面的更多信息，参见第 8 章。

(5) initParam

initParam 对象允许我们容易地访问上下文初始化参数(context initialization parameter)。例如，下面的代码输出初始化参数 defaultColor 的值。

```
${initParam.defaultColor}
```

有关初始化参数应用方面的更多信息，参见本书第二卷。

(6) pageScope, requestScope, sessionScope 和 applicationScope

这些对象允许我们限定系统应该在什么地方查找作用域变量。例如，在遇到下面的表达式

```
${name}
```

时，系统会依次在 PageContext, HttpServletRequest, HttpSession 和 ServletContext 中查找名为 name 的作用域变量，返回所找到的第一个匹配项。另外，如果遇到下面的表达式

```
${requestScope.name}
```

则系统只在 HttpServletRequest 中查找。

示例

清单 16.10 中给出的 JSP 页面使用隐式对象输出一个请求参数、一个 HTTP 请求报头、一个 cookie 值和服务器的相关信息。图 16.5 给出相应的结果。

清单 16.10 implicit-objects.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Using Implicit Objects</TITLE>
<LINK REL=stylesheet
      HREF="/e1/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Using Implicit Objects
  </TH>
</TABLE>
<P>
```

```
<UL>
<LI><B>test Request Parameter:</B> ${param.test}
<LI><B>User-Agent Header:</B> ${header["User-Agent"]}
<LI><B>JSESSIONID Cookie Value:</B> ${cookie.JSESSIONID.value}
<LI><B>Server:</B> ${pageContext.servletContext.serverInfo}
</UL>
</BODY></HTML>
```

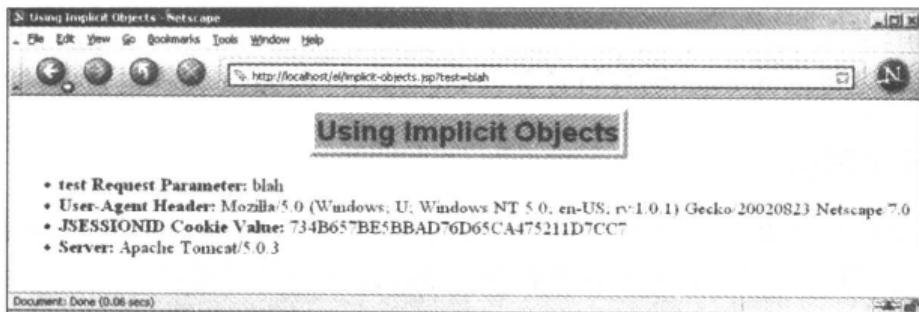


图 16.5 许多作用域变量是自动定义的。我们将这些预定义变量称为“隐式对象”

16.9 表达式语言中运算符的应用

JSP 2.0 表达式语言定义了许多算术、关系、逻辑和缺失值检测运算符。虽然使用运算符的代码常常会比等同的 Java 代码更简短，但要时刻牢记表达式语言的主要宗旨：提供对现有对象的简洁访问，经常用在 MVC 构架中。仅仅是将长而复杂的脚本元素替换成稍微简短一些的表达式语言元素并没有什么好处。这些方式都是错误的：复杂的商业逻辑或数据访问逻辑根本不应该放在 JSP 页面内。应该尽可能地将表达式语言的应用限制在表示逻辑(presentation logic)上；而商业逻辑应该存储在常规的 Java 类中，供 servlet 调用。

核心方法

应该将表达式语言的运算符用在面向表示逻辑(确定如何表达数据)的简单任务中。要避免在商业逻辑(创建并处理数据)中使用这些运算符。相反，应该将商业逻辑放在常规 Java 类中，然后从启动 MVC 过程的 servlet 中调用这些代码。

16.9.1 算术运算符

这些运算符一般用来对存储在 bean 中的值进行简单的操作。不要在复杂的算法中使用它们；而应将这类代码放在常规 Java 代码中。

(1) +和-

这二者是常规的加法和减法运算符，但有两点例外。首先，如果任一操作数为字符串，那么，在运行时，字符串会被自动解析成数值(但是系统并不自动捕获 NumberFormatException)。其次，如果任一操作数为 BigInteger 或 BigDecimal 类型，那么，系统会使用相应的 add 和 subtract 方法。

(2) *, /和 div

它们是常规的乘法和除法运算符，但有几点例外。第一，如同+和-运算符一样，

类型转换自动完成。第二，常规的算术运算符优先级依旧适用，因此，下面的语句

```
$ { 1 + 2 * 3 }
```

返回 7，不是 9。我们可以使用圆括号改变运算符的优先次序。第三，/ 和 div 运算符是等同的；提供这两者是为了与 XPath 和 JavaScript(ECMAScript)兼容。

(3) % 和 mod

%(或与之等同的 mod)运算符计算模数(余数)，和 Java 编程语言中的%相同。

16.9.2 关系运算符

这些运算符最常和 JSP 表达式语言的条件运算符(16.10 节)一同使用，或是用在应用布尔型属性值的定制标签中(比如 JSP 标准标签库 JSTL 中的那些循环标签——本书第二卷论述)。

(1) == 和 eq

这两个相等性运算符检查参数是否相等。但是，相比于 Java 的==运算符，它们更像 Java 的 equals 方法。如果两个操作数为同一对象，则返回 true。如果两个操作数为数值，则用 Java 的==运算符比较它们。如果任一操作数为 null，则返回 false。如果任一操作数为 BigInteger 或 BigDecimal，则用 compareTo 方法比较操作数。否则，用 equals 比较操作数。

(2) != 和 ne

这两个相等性运算符检查参数是否不同。同样，相比于 Java 的!=运算符，它们更像 Java 的 equals 方法的否定式。如果两个操作数为同一对象，则返回 false。如果两个操作数均为数值，则用 Java 的!=比较它们。如果任一操作数为 null，则返回 true。如果任一操作数为 BigInteger 或 BigDecimal，则用 compareTo 比较这些操作数。否则，用 equals 比较操作数，返回相反的结果。

(3) < 和 lt, > 和 gt, <= 和 le, >= 和 ge

它们都是标准的算术运算符，但有两点例外，首先，类型转换的执行同==和!=。其次，如果参数为字符串，则进行字面比较。

16.9.3 逻辑运算符

这些运算符用来组合关系运算符得出的结果。

&&, and, ||, or, ! 和 not

它们是标准的逻辑 AND, OR 和 NOT 运算符。它们将参数强制转换成 Boolean，并使用常规的 Java “短路”求值(一旦确定结果，立即停止测试)。&& 和 and 是等同的，|| 和 or 是等同的，! 和 not 是等同的。

16.9.4 空 运 算 符

empty

如果这个运算符的参数为 null、空字符串、空数组、空 Map 或空集合，则返回 true。

否则返回 false。

16.9.5 示例

清单 16.11 举例说明几个标准的运算符；图 16.6 给出相应的结果。

清单 16.11 operators.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>EL Operators</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    EL Operators
  </TH></TR>
  <P>
  <TABLE BORDER=1 ALIGN="CENTER">
    <TR><TH CLASS="COLORED" COLSPAN=2>Arithmetic Operators
        <TH CLASS="COLORED" COLSPAN=2>Relational Operators
    <TR><TH>Expression<TH>Result<TH>Expression<TH>Result
    <TR ALIGN="CENTER">
      <TD>\${3+2-1}<TD>\${3+2-1}    <%-- Addition/Subtraction --%>
      <TD>\${1<2}<TD>\${1<2}      <%-- Numerical comparison --%>
    <TR ALIGN="CENTER">
      <TD>\$("1"+2)<TD>\${"1"+2}    <%-- String conversion --%>
      <TD>\${"a"<"b"}<TD>\${"a"<"b"} <%-- Lexical comparison --%>
    <TR ALIGN="CENTER">
      <TD>\${1 + 2*3 + 3/4}<TD>\${1 + 2*3 + 3/4} <%-- Mult/Div --%>
      <TD>\${2/3 &gt;= 3/2}<TD>\${2/3 >= 3/2}       <%-- >= --%>
    <TR ALIGN="CENTER">
      <TD>\${3%2}<TD>\${3%2}          <%-- Modulo --%>
      <TD>\${3/4 == 0.75}<TD>\${3/4 == 0.75} <%-- Numeric = --%>
    <TR ALIGN="CENTER">
      <%-- div and mod are alternatives to / and % --%>
      <TD>\${(8 div 2) mod 3}<TD>\${(8 div 2) mod 3}
      <%-- Compares with "equals" but returns false for null --%>
      <TD>\${null == "test"}<TD>\${null == "test"}</TD>
    <TR><TH CLASS="COLORED" COLSPAN=2>Logical Operators
        <TH CLASS="COLORED" COLSPAN=2><CODE>empty</CODE> Operator
    <TR><TH>Expression<TH>Result<TH>Expression<TH>Result
    <TR ALIGN="CENTER">
      <TD>\${(1<2) && (4<3)}<TD>\${(1<2) && (4<3)} <%--AND--%>
      <TD>\${empty ""}<TD>\${empty ""} <%-- Empty string --%>
    <TR ALIGN="CENTER">
      <TD>\${(1<2) || (4<3)}<TD>\${(1<2) || (4<3)} <%--OR--%>
      <TD>\${empty null}<TD>\${empty null} <%-- null --%>
    <TR ALIGN="CENTER">
      <TD>\${!(1<2)}<TD>\${!(1<2)} <%-- NOT --%>
      <%-- Handles null or empty string in request param --%>
      <TD>\${empty param.blah}<TD>\${empty param.blah}</TD>
  </TABLE>
```

```
</BODY></HTML>
```

The screenshot shows a Microsoft Internet Explorer window with the title "EL Operators". The content consists of four tables demonstrating various EL operators:

Arithmetic Operators		Relational Operators	
Expression	Result	Expression	Result
<code> \${3+2-1}</code>	4	<code> \${1<2}</code>	true
<code> \${"1"+2}</code>	3	<code> \${"a"<"b"}</code>	true
<code> \${1 + 2*3 + 3/4}</code>	7.75	<code> \${2/3 >= 3/2}</code>	false
<code> \${3%2}</code>	1	<code> \${3/4 == 0.75}</code>	true
<code> \${(8 div 2) mod 3}</code>	1.0	<code> \${null == "test"}</code>	false

Logical Operators		empty Operator	
Expression	Result	Expression	Result
<code> \${1<2} && (4<3)}</code>	false	<code> \${empty ""}</code>	true
<code> \${1<2} (4<3)}</code>	true	<code> \${empty null}</code>	true
<code> \${!(1<2)}</code>	false	<code> \${empty paramblah}</code>	true

图 16.6 表达式语言定义一组运算符。使用它们时要谨慎：要从 servlet 中调用商业逻辑，而从非 JSP 页面调用

16.10 表达式的条件求值

JSP 2.0 表达式语言自身并不提供丰富的条件求值功能。这项功能由 JSP 标准标签库 (JSTL) 的 c:if 和 c:choose 标签，或其他定制标签库提供。(JSTL 和定制标签库的创建在本书第二卷介绍)。

但是，表达式语言支持基本的?:运算符，这个操作符的功能与 Java、C 和 C++ 语言中对应的运算符功能相同。例如，如果 test 的求值结果为 true，下面的语句

```
 ${ test ? expression1 : expression2 }
```

返回 expression1 的值；否则，返回 expression2 的值。一定要牢记，表达式语言的主要目标是简化表示逻辑；应该避免将这项技术用在商业逻辑中。

示例

清单 16.12 中的 servlet 创建两个 SalesBean 对象(清单 16.13)，并将请求转发给 JSP 页面(清单 16.14)进行表示(图 16.7)。但是，如果总的销售数值为负值，这个 JSP 页面希望使用边框为红色的表格单元。如果数值为负值，这个页面还希望使用白色的背景。实现这种行为是一种表示任务，因此，可以使用?:运算符。

清单 16.12 Conditionals.java

```
package coreservlets;

/** Servlet that creates scoped variables that will be used
 * to illustrate the EL conditional operator (xxx ? xxx : xxx).
 */
```

```

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

public class Conditionals extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        SalesBean apples =
            new SalesBean(150.25, -75.25, 22.25, -33.57);
        SalesBean oranges =
            new SalesBean(-220.25, -49.57, 138.25, 12.25);
        request.setAttribute("apples", apples);
        request.setAttribute("oranges", oranges);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/conditionals.jsp");
        dispatcher.forward(request, response);
    }
}

```

清单16.13 SalesBean.java

```

package coreservlets;

public class SalesBean {
    private double q1, q2, q3, q4;

    public SalesBean(double q1Sales,
                    double q2Sales,
                    double q3Sales,
                    double q4Sales) {
        q1 = q1Sales;
        q2 = q2Sales;
        q3 = q3Sales;
        q4 = q4Sales;
    }

    public double getQ1() { return(q1); }

    public double getQ2() { return(q2); }

    public double getQ3() { return(q3); }

    public double getQ4() { return(q4); }

    public double getTotal() { return(q1 + q2 + q3 + q4); }
}

```

清单16.14 conditionals.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Conditional Evaluation</TITLE>
<LINK REL=stylesheet
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>

```

```

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Conditional Evaluation
  </TABLE>
  <P>
  <TABLE BORDER=1 ALIGN="CENTER">
    <TR><TH>
      <TH CLASS="COLORED">Apples
      <TH CLASS="COLORED">Oranges
    <TR><TH CLASS="COLORED">First Quarter
      <TD ALIGN="RIGHT">${apples.q1}
      <TD ALIGN="RIGHT">${oranges.q1}
    <TR><TH CLASS="COLORED">Second Quarter
      <TD ALIGN="RIGHT">${apples.q2}
      <TD ALIGN="RIGHT">${oranges.q2}
    <TR><TH CLASS="COLORED">Third Quarter
      <TD ALIGN="RIGHT">${apples.q3}
      <TD ALIGN="RIGHT">${oranges.q3}
    <TR><TH CLASS="COLORED">Fourth Quarter
      <TD ALIGN="RIGHT">${apples.q4}
      <TD ALIGN="RIGHT">${oranges.q4}
    <TR><TH CLASS="COLORED">Total
      <TD ALIGN="RIGHT"
        BGCOLOR="${(apples.total < 0) ? "RED" : "WHITE"}">
        ${apples.total}
      <TD ALIGN="RIGHT"
        BGCOLOR="${(oranges.total < 0) ? "RED" : "WHITE"}">
        ${oranges.total}
    </TABLE>
  </BODY></HTML>

```

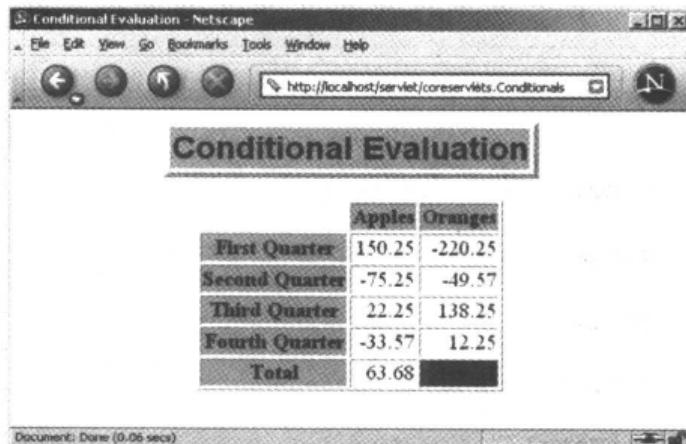


图 16.7 我们可以使用 C 风格的?:运算符条件性地输出不同的元素。但是，使用 JSP 标准标签库(JSTL)常常要好于使用这种类型的条件操作

16.11 表达式语言其他功能概览

本章总结了 JSP 2.0 表达式语言的大部分功能。但是，有两项功能我们在此没有讨论：表达式语言在标签库中的应用，以及表达式语言中函数的应用。由于它们要用到本书第二卷中介绍的技术，故而，我们将它们推迟到本书第二卷。

快速预览：表达式语言元素可以用在任何定制标签属性中，只要该属性允许使用请求期间的表达式(即属性的 attribute 元素指定 rexprvalue 为 true)。函数对应常规 Java 类中的 public static 方法，它使用标签库描述符(Tag Library Descriptor, TLD)文件的 function 元素来定义。

第III部分：支持技术

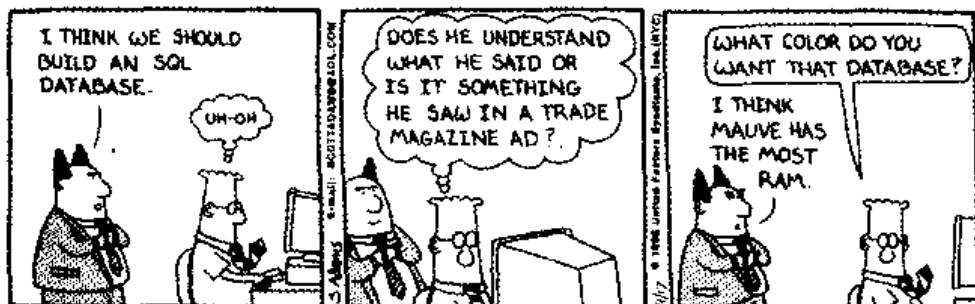
- 第 17 章
数据库访问：JDBC
- 第 18 章
配置 MS Access, MySQL 和 Oracle9i
- 第 19 章
HTML 表单的创建和处理

第 17 章 数据库访问：JDBC

本章的主题：

- 连接到数据库的 7 个基本步骤
- 简化 JDBC 的使用：一些实用工具
- 使用预编译(参数化)查询
- 创建和执行存储过程
- 通过事务更新数据
- 使用 JDO 和其他对象-关系(object-relational)的映射

JDBC 提供一套访问关系型数据库的标准库。通过 JDBC API，我们可以使用完全相同的 Java 语法访问大量各种各样的 SQL 数据库。要注意，虽然 JDBC API 标准化了数据库的连接方式，发送查询和提交事务的语法，以及表示结果的数据结构，但 JDBC 并不试图将 SQL 的语法也标准化，了解这一点比较重要。故而，我们依旧可以使用数据库提供商支持的任何 SQL 扩展。但是，由于大多数查询都遵循标准的 SQL 语法，因此，在使用 JDBC 的情况下，对代码做出很少的修改就可以更改数据库的主机、端口，甚至数据库提供商。



翻印获得 United Feature Syndicate 公司的许可

正式来讲，JDBC 不是一个首字母缩写词，因此，它不代表任何事情。“Java DataBase Connectivity”通常用作该名称的完整形式。

尽管完整的数据库编程教程超出了本章的范畴，我们还是在 17.1 节中介绍了应用 JDBC 的一些基础知识，我们假定您已经对 SQL 有所了解。

介绍完 JDBC 的基础知识之后，在 17.2 节中，我们提供一些访问 Microsoft Access 数据库的 JDBC 示例。

为了简化本章其余部分的 JDBC 代码，我们在 17.3 节中提供一些实用工具，用以创建到数据库的连接。

在 17.4 节中，我们论述预备语句(prepared statement)，它允许您多次执行类似的 SQL 语句；这要比每次都执行原始的查询更有效率。

在 17.5 节中，我们分析可调用语句(callable statement)。可调用语句允许您执行数据库存储过程或函数。

在 17.6 节中，我们介绍用以维护数据库完整性的事务管理(transaction management)。

通过在事务中执行对数据库的更改，能够确保发生问题时，数据库中的值会返回到它们最初的状态。

在 17.7 节中，我们简要地分析对象-关系映射(object-to-relational mapping, ORM)。ORM 框架为管理数据库中的信息提供完全的面向对象的方式。通过 ORM，我们不再需要直接使用 JDBC 和 SQL，只需简单地调用对象的方法。

高级的 JDBC 主题，包括使用定制 JSP 标签访问数据库，通过 JNDI 使用数据源，以及通过共享数据库连接来提高性能等，参见本书第二卷。有关 JDBC 更详细的信息，参见 <http://java.sun.com/products/jdbc/>、`java.sql` 的在线 API、或 <http://java.sun.com/docs/books/tutorial/jdbc/> 给出的 JDBC 教程。

17.1 JDBC 应用概述

本节中，我们介绍查询数据库的 7 个标准步骤。在 17.2 节中，我们给出两个简单的例子(一个命令行程序，一个 servlet)，说明查询 Microsoft Access 数据库时相应的步骤。

下面是一个汇总，细节随后给出。

(1) **载入 JDBC 驱动程序。**

如果要载入驱动程序，只需在 `Class.forName` 方法中指定数据库驱动程序的类名。这样做就自动创建了驱动程序的实例，并注册到 JDBC 驱动程序管理器。

(2) **定义连接 URL (connection URL)。**

在 JDBC 中，连接 URL 指定服务器的主机名、端口以及希望与之建立连接的数据
库名。

(3) **建立连接。**

有了连接 URL、用户名和密码，就可以建立到数据库的网络连接。连接建立之后，
就可以执行数据库查询，直到连接关闭为止。

(4) **创建 Statement 对象。**

创建 Statement 对象才能向数据库发送查询和命令。

(5) **执行查询或更新。**

有了 Statement 对象后，就可以使用 `execute`, `executeQuery`, `executeUpdate` 或
`executeBatch` 方法发送 SQL 语句到数据库。

(6) **结果处理。**

数据库查询执行完毕之后，返回一个 `ResultSet`。`ResultSet` 表示一系列的行和列，
我们可以调用 `next` 和各种 `getXxx` 方法对这些行和列进行处理。

(7) **关闭连接。**

在执行完查询且处理完结果之后，应该关闭连接，释放与数据库相关联的资源。

17.1.1 载入 JDBC 驱动程序

驱动程序是知道如何与实际的数据库服务器进行会话的软件部件。如果要载入驱动程序，只需载入相应的类；驱动程序类自身的一个 `static` 代码块自动生成驱动程序的实例，并将其注册到 JDBC 驱动程序管理器。为了使代码尽可能地灵活，我们要避免对类名的引用

进行硬编码(hard-coding)。17.3 节，我们提供了一个实用工具类，它从 Properties 文件载入驱动程序，从而程序中的类名不需硬编码。

这些需求提出两个有趣的问题。首先，如何在不生成类的实例的情况下载入它呢？其次，如何引用在代码编译时不知道其名称的类呢？两个问题的答案都是使用 Class.forName。这个参数接收一个表示完全限定类名(即包括包名的类名)的字符串，载入对应的类。这个调用有可能会抛出 ClassNotFoundException 异常，因而必须放在 try/catch 块中，如下所示。

```
try {
    Class.forName("com.microsoft.jdbc.MicrosoftDriver");
    Class.forName("com.oracle.jdbc.driver.OracleDriver");
    Class.forName("com.sybase.jdbc.SybDriver");
} catch(ClassNotFoundException cnfe) {
    System.err.println("Error loading driver: " + cnfe);
}
```

JDBC 方案的优点在于，任何情况下，数据库服务器不需做出任何更改。相反，JDBC 驱动程序(客户端)将用 Java 编程语言写就的调用转换成服务器所需的本地格式。这种方案意味着，我们首先得获得与所使用的数据库相对应的专有 JDBC 驱动程序，还需要检查提供商的文档，得出数据库驱动程序的完全限定类名。

原则上，CLASSPATH 内的任何类都可以使用 Class.forName。然而，实际上，大多数 JDBC 提供商都在 JAR 文件中发布它们的驱动程序。因此，在开发过程中要保证 CLASSPATH 设定中包括驱动程序 JAR 文件所在的路径。在 Web 服务器上部署时，需要将 JAR 文件放在 Web 应用的 WEB-INF/lib 目录(参见第 2 章)。但是，最好还是要与 Web 服务器的管理员核实一下。经常地，如果多个 Web 应用使用相同的数据库驱动程序，管理员会将 JAR 文件放置在服务器使用的公共目录中。例如，在 Apache Tomcat 中，多个应用公共的 JAR 文件可以放在 *install_dir/common/lib*。

重点提示

在部署具体的应用时，可以将 JDBC 驱动程序文件(JAR 文件)放在 WEB-INF/lib 目录中。但是，管理员有可能选择将 JAR 文件移到服务器上专门存储公共库文件的目录。

图 17.1 给出实现 JDBC 驱动程序的两种常见方案。第一种方案是 JDBC-ODBC 桥接器，第二种方案为纯 Java 实现。使用 JDBC-ODBC 桥接器方案的驱动程序称为 I 类驱动程序。由于许多数据库支持开放数据库互连(ODBC)访问，因此 JDK 中包括一个 JDBC-ODBC 桥接器，可以用来连接到数据库。但是，如果提供商提供纯 Java 驱动程序，应该尽可能地使用纯 Java 驱动程序，因为 JDBC-ODBC 驱动程序实现要比纯 Java 实现慢。我们将纯 Java 驱动程序称为 IV 类驱动程序。JDBC 规范还定义了其他两种类型的驱动程序，II 类和 III 类；但是，它们并不那么常见。有关驱动程序类型的详细信息，参见 <http://java.sun.com/products/jdbc/driverdesc.html>。

在本章开始给出的例子中，我们使用桥接器(包括在 JDK 1.4 中)连接到 Microsoft Access 数据库。后面的例子中，我们使用纯 Java 驱动程序连接到 MySQL 和 Oracle9i 数据库。

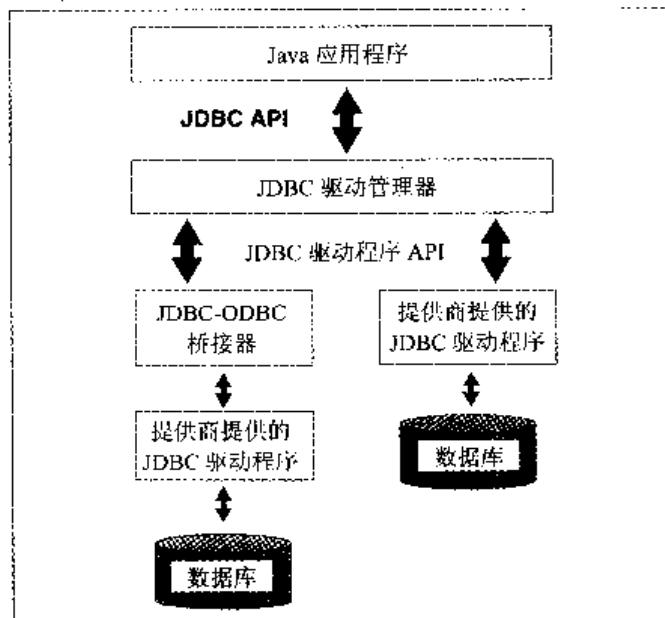


图 17.1 实现 JDBC 驱动程序的两种常见方案。JDK 1.4 包括一个 JDBC-ODBC 桥接器；但是，纯 JDBC 驱动程序(由提供商提供)性能更好

我们在 18.1 节提供了 Microsoft Access 驱动程序的相关信息。MySQL 驱动程序的信息在 18.2 节中提供，Oracle 驱动程序的信息在 18.3 节中提供。大多数其他数据库提供商为他们的数据库提供免费的 JDBC 驱动程序。上述驱动程序以及其他第三方驱动程序的最新列表，参见 <http://industry.java.sun.com/products/jdbc/drivers/>。

17.1.2 定义连接 URL

载入 JDBC 驱动程序之后，我们必须指定数据库服务器的位置。指向数据库的 URL 所使用的协议是 jdbc: 协议，并且嵌入服务器的主机名、端口和数据库名(或引用)。精确的格式定义在伴随特定驱动程序的文档中提供，此处仅仅是几个具有代表性的例子。

```

String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
                   ":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host +
                    ":" + port + ":" + "?SERVICENAME=" + dbName;
String msAccessURL = "jdbc:odbc:" + dbName;
  
```

17.1.3 建立连接

建立实际的网络连接时，需要将该 URL、数据库用户名和数据库密码传递给 DriverManager 类的 getConnection 方法，如下面的例子所示。要注意，getConnection 抛出 SQLException 异常，因此，我们需要使用 try/catch 块。因为下面步骤中的方法都抛出同样的异常，故而，我们一般使用单个 try/catch 块来处理所有的步骤，因此，我们将下面例子中的 try/catch 块都略去。

```
String username = "jay_debesee";
```

```

String password = "secret";
Connection connection =
    DriverManager.getConnection(oracleURL, username, password);

```

`Connection` 类还包括其他一些有用的方法，我们接下来对它们进行简短的介绍。前 3 种方法在 17.4 节~17.6 节中详细介绍。

- `prepareStatement`
创建预编译查询，提交给数据库。详细信息参见 17.4 节。
- `prepareCall`
访问数据库中的存储过程。详细信息参见 17.5 节。
- `rollback/commit`
控制事务管理。详细信息参见 17.6 节。
- `close`
终止打开的连接。
- `isClosed`
确定连接是否超时或被显式关闭。

建立连接的过程中，一个可选的部分是使用 `getMetaData` 方法查找数据库的相关信息。这个方法返回 `DatabaseMetaData` 对象，该对象拥有相应的方法，可以得出数据库自身的名称和版本(`getDatabaseProductName`, `getDatabaseProductVersion`)，或者 JDBC 驱动程序的名称和版本(`getDriverName`, `getDriverVersion`)。下面是一个具体的例子。

```

DatabaseMetaData dbMetaData = connection.getMetaData();
String productName =
    dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
    dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);

```

17.1.4 创建 Statement 对象

`Statement` 对象用来向数据库发送查询和命令。它由 `Connection` 的 `createStatement` 方法创建，如下所示。

```
Statement statement = connection.createStatement();
```

大部分数据库驱动程序，但并非全部，允许在同一连接中打开多个并行 `Statement` 对象。

17.1.5 执行查询或更新

有了 `Statement` 对象之后，就可以使用它的 `executeQuery` 方法发送 SQL 查询，`executeQuery` 返回 `ResultSet` 类型的对象。下面是一个例子。

```

String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);

```

下面列表汇总了 `Statement` 类中常用的方法。

- `executeQuery`

执行 SQL 查询并在 ResultSet 中返回数据。ResultSet 可能为空，但不会为 null。

- **executeUpdate**

用于 UPDATE, INSERT 或 DELETE 命令。返回受影响的行数，可以为 0。它还提供对数据定义语言(Data Definition Language, DDL)命令的支持，例如 CREATE TABLE, DROP TABLE 和 ALTER TABLE。

- **executeBatch**

将一组命令作为一个单元执行，返回一个数组，其中存储每个命令的更新计数。`addBatch` 可以向批量执行的命令组中添加命令。要注意，在开发 JDBC 兼容的驱动程序时，提供商并不是一定要实现这个方法。

- **setQueryTimeout**

指定驱动程序在抛出 SQLException 异常之前，等待处理结果的时间。

- **getMaxRows/setMaxRows**

确定 ResultSet 可能容纳的最大行数。超过的行将会在不给出任何警告的情况下丢弃。默认值为 0，表示没有限制。

除使用此处描述的方法发送任意命令之外，我们还可以使用 Statement 对象创建参数化查询(将值提供给预编译的固定格式查询)。详细信息参见 17.4 节。

17.1.6 结果处理

处理结果最简单的方式是使用 ResultSet 的 `next` 方法在表中移动，每次一行。在一行之内，ResultSet 提供各种 `getXxx` 方法，它们都以列名或列索引为参数，以各种不同的 Java 类型返回结果。例如，如果值应该为整数则使用 `getInt`, String 为 `getString`, 大多数其他数据类型也是如此。如果只是希望显示结果，大多数类型的列都可以使用 `getString`。但是，在使用参数为列索引(而非列名)的 `getXxx` 方法时，要注意列的索引从 1 开始(遵循 SQL 的约定)，不同于 Java 编程语言中数组、向量和大多数其他数据结构(都从 0 开始)。

警告

ResultSet 中行的第一列索引为 1，而非 0。

下面的例子打印 ResultSet 中所有行的前面两列，以及名(firstname)和姓(lastname)。

```
while(resultSet.next()) {
    System.out.println(resultSet.getString(1) + " " +
        resultSet.getString(2) + " " +
        resultSet.getString("firstname") + " " +
        resultSet.getString("lastname"));
}
```

我们建议在访问 ResultSet 的列时，不要使用列索引，而使用列名。使用这种方式，如果表的列结构发生改变，与 ResultSet 交互的代码不太容易出错。

核心方法

访问 ResultSet 中的数据时要使用列名，而非列索引。

JDBC 1.0 中，我们只能在 ResultSet 中向前移动；然而，在 JDBC 2.0 中，我们可以在 ResultSet 中向下(next)或向上(previous)移动，同样也可以移到特定的行(relative, absolute)。在本书第二卷中，我们提供几种定制标签，用以阐述 ResultSet 中的 JDBC 2.0 方法。

要明白，不管 JDBC 1.0 还是 JDBC 2.0，都没有提供确定驱动程序 JDBC 版本的直接机制。JDBC 3.0 通过在 DatabaseMetaData 类中增加 getJDBCMajorVersion 和 getJDBCMinorVersion 方法解决了这个问题。如果从提供商的文档中不能得出 JDBC 的版本，可以编写一个简短的程序获取一个 ResultSet，并在该 ResultSet 上试验 previous 操作。由于 resultSet.previous 只在 JDBC 2.0 及后面的版本中才可以使用，JDBC 1.0 驱动程序此时会抛出异常。18.4 节给出一个例子程序，它通过相关的测试(不太严格)确定数据库驱动程序的 JDBC 版本。

下面的列表汇总了 ResultSet 中一些有用的方法。

- **next/previous**
将 ResultSet 中的游标分别移动到下一行(任何 JDBC 版本)或前一行(JDBC 版本 2.0 及之后的版本)。
- **relative/absolute**
relative 方法将游标相对地移动特定数目的行，或正或负(向前或者向后)。absolute 方法将游标移到给定的行号。如果绝对值是负数，那么游标将相对于 ResultSet 的结尾进行定位(JDBC 2.0)。
- **getXxx**
返回 Xxx Java 类型(参见 java.sql.Types)的值，这个值来自于由列名或列索引指定的列。如果列的值为 SQL 中的 NULL 值，可以返回 0 或 null。
- **wasNull**
检查上面的 getXxx 读到的是否为 SQL 的 NULL 值。如果列的类型为基本类型(int, float 等)，且数据库中的值为 0，那么这项检查就很重要。由于数据库 NULL 也返回 0，所以 0 值和数据库的 NULL 不能区分开来。如果列的类型为对象(String, Date 等)，可以简单地将返回值与 null 比较。
- **findColumn**
返回 ResultSet 中与指定列名对应的索引。
- **getRow**
返回当前的行号。第一行从 1 开始(JDBC 2.0)。
- **getMetaData**
返回描述 ResultSet 的 ResultSetMetaData 对象。ResultSetMetaData 给出列的数目和名称。

getMetaData 方法尤为有用。仅仅有 ResultSet 的情况下，我们必须知道列的名称、数目和类型才能正确地对表进行处理。对于大部分固定格式的查询，这也是合理的预期情况。但是，对于某些特定的查询，如果能够动态地得出与结果相关的高级信息会更有效率。这就是 ResultSetMetaData 类的作用：通过它，我们能够确定 ResultSet 中列的数目、名称和类型。

下面列出 ResultSetMetaData 的一些实用方法。

- **getCount**
返回 ResultSet 中列的数目。
- **getColumnName**
返回列在数据库中的名称(索引从 1 开始)。
- **getColumnType**
返回列的 SQL 类型, 对应 java.sql.Types 中的项。
- **isReadOnly**
表示数据项是否为只读值。
- **isSearchable**
表明该列是否可以用在 WHERE 子句中。
- **isNullable**
表明该列是否可以存储 NULL。

ResultSetMetaData 并不包括有关行数的信息;然而,如果驱动程序与 JDBC 2.0 兼容,我们可以调用 ResultSet 的 last 将游标移到最后一行,然后调用 getRow 获取当前的行号。在 JDBC 1.0 中,确定行数的惟一方式是重复调用 ResultSet 的 next,直到它返回 false 为止。

重点提示

ResultSet 和 ResultSetMetaData 没有直接提供方法返回查询所返回的行数。然而,在 JDBC 2.0 中,可以调用 last 将游标定位到 ResultSet 的最后一行,然后调用 getRow 获取当前的行号。

17.1.7 关闭连接

显式地关闭连接使用下面的指令:

```
connection.close();
```

关闭连接还会关闭对应的 Statement 和 ResultSet 对象。

因为关闭连接的开销常常很大,故而,如果希望执行额外的数据库操作,应该延迟关闭数据库。实际上,重用已有的连接是一件重大的优化,JDBC 2.0 API 甚至专为获取共享连接定义了 ConnectionPoolDataSource 接口。连接共享在本书第二卷中进行论述。

17.2 基本 JDBC 示例

在这一节中,我们提供两个简单的 JDBC 示例,它们连接到 Microsoft Access Northwind 数据库(如图 17.2 所示),并执行一个简单的查询。Northwind 数据库包括在 Microsoft Office 的示例中。用 JDBC 访问 Microsoft Access Northwind 数据库时应该如何配置,请参见 18.1 节。

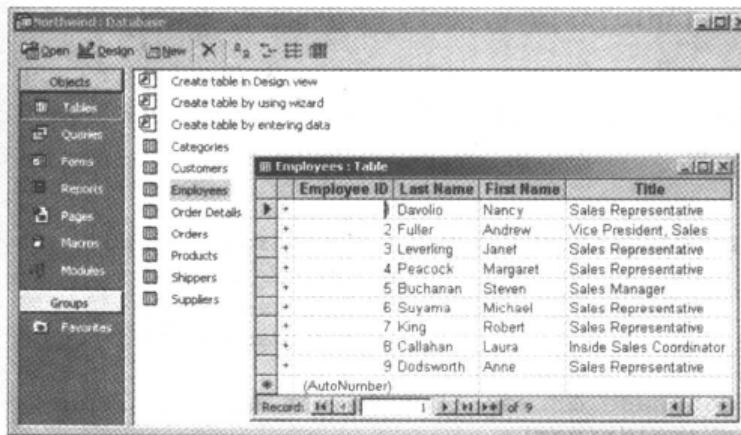


图 17.2 Microsoft Access Northwind 示例数据库。正在显示的是 Employees 表的前 4 列。有关如何使用这个数据库的信息参见 18.1 节

对于测试和试验来说，Northwind 数据库是一个好的选择，因为许多系统都安装了该数据库，而且用以连接到 Microsoft Access 的 JDBC-ODBC 桥接器已经绑定在 JDK 之中。但是，实际上我们一般不会使用 Microsoft Access 作为真正的在线数据库。对于产品来说，高性能的选择，如 MySQL(见 18.2 节)，Oracle9i(见 18.3 节)，Microsoft SQL Server，Sybase，或 DB2 都是更优的选择。

第一个例子，清单 17.1，提供一个独立的类，名为 NorthwindTest，该类遵照前面概括的 7 个步骤，显示查询 Employee 表的结果。

NorthwindTest 的结果列在清单 17.2 中。由于 NorthwindTest 在 coreservlets 包中，因此它在 coreservlets 子目录中。在编译该文件之前，首先要设置 CLASSPATH，使之包括含有 coreservlets 目录的目录。详细信息参见 2.7 节。设置好之后，在 coreservlets 子目录中运行 javac NorthwindTest.java(或在 IDE 中选择“build”或“compile”)编译该程序。运行 NorthwindTest 时，需要引用完整的包名 java coreservlets.NorthwindTest。

第二个例子，清单 17.3(NorthwindServlet)，从 servlet 中连接到数据库，以 HTML 表格的形式提供查询的结果。清单 17.1 和清单 17.3 都使用 JDBC-ODBC 桥接驱动程序，sun.jdbc.odbc.JdbcOdbcDriver，它包括在 JDK 之中。

清单 17.1 NorthwindTest.java

```
package coreservlets;

import java.sql.*;

/** A JDBC example that connects to the MicroSoft Access sample
 * Northwind database, issues a simple SQL query to the
 * employee table, and prints the results.
 */

public class NorthwindTest {
    public static void main(String[] args) {
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String url = "jdbc:odbc:Northwind";
        String username = ""; // No username/password required
        String password = ""; // for desktop access to MS Access.
    }
}
```

```

        showEmployeeTable(driver, url, username, password);
    }

    /**
     * Query the employee table and print the first and
     * last names.
     */

    public static void showEmployeeTable(String driver,
                                         String url,
                                         String username,
                                         String password) {
        try {
            // Load database driver if it's not already loaded.
            Class.forName(driver);
            // Establish network connection to database.
            Connection connection =
                DriverManager.getConnection(url, username, password);
            System.out.println("Employees\n" + "=====");
            // Create a statement for executing queries.
            Statement statement = connection.createStatement();
            String query =
                "SELECT firstname, lastname FROM employees";
            // Send query to database and store results.
            ResultSet resultSet = statement.executeQuery(query);
            // Print results.
            while(resultSet.next()) {
                System.out.print(resultSet.getString("firstname") + " ");
                System.out.println(resultSet.getString("lastname"));
            }
            connection.close();
        } catch(ClassNotFoundException cnfe) {
            System.err.println("Error loading driver: " + cnfe);
        } catch(SQLException sqle) {
            System.err.println("Error with connection: " + sqle);
        }
    }
}

```

清单 17.2 NorthwindTest 结果

Prompt> java coreservlets.NorthwindTest

```

Employees
=====
Nancy Davolio
Andrew Fuller
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth

```

第二个例子，NorthwindServlet(清单 17.3)，用来执行查询的信息取自 HTML 表单——NorthwindForm.html，参见清单 17.4。我们可以在将表单提交给 servlet 之前，将查询输入到表单的文本字段内。这个 servlet 从请求参数中读取驱动程序、URL、用户名、密码和查

询，并根据查询的结果生成一个 HTML 表格。该 servlet 还演示了如何使用 DatabaseMetaData 查找数据库的产品名称和产品版本。这个 HTML 表单在图 17.3 中给出；图 17.4 给出提交表单后的结果。这个例子中的 HTML 表单和 servlet 都在名为 jdbc 的 Web 应用之中。有关创建和使用 Web 应用的更多信息，参见 2.11 节。

清单 17.3 NorthwindServlet.java

```
package coreservlets;

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

/** A simple servlet that connects to a database and
 * presents the results from the query in an HTML
 * table. The driver, URL, username, password,
 * and query are taken from form input parameters.
 */

public class NorthwindServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +"
            "Transitional//EN\"\\n";
        String title = "Northwind Results";
        out.print(docType +
                  "<HTML>\\n" +
                  "<HEAD><TITLE>" + title + "</TITLE></HEAD>\\n" +
                  "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\\n" +
                  "<H1>Database Results</H1>\\n");
        String driver = request.getParameter("driver");
        String url = request.getParameter("url");
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        String query = request.getParameter("query");
        showTable(driver, url, username, password, query, out);
        out.println("</CENTER></BODY></HTML>");
    }

    public void showTable(String driver, String url,
                         String username, String password,
                         String query, PrintWriter out) {
        try {
            // Load database driver if it's not already loaded.
            Class.forName(driver);
            // Establish network connection to database.
            Connection connection =
                DriverManager.getConnection(url, username, password);
            // Look up info about the database as a whole.
            DatabaseMetaData dbMetaData = connection.getMetaData();
            out.println("<UL>");
            String productName =
```

```

        dbMetaData.getDatabaseProductName();
        String productVersion =
            dbMetaData.getDatabaseProductVersion();
        out.println(" <LI><B>Database:</B> " + productName +
            " <LI><B>Version:</B> " + productVersion +
            "</UL>");
        Statement statement = connection.createStatement();
        // Send query to database and store results.
        ResultSet resultSet = statement.executeQuery(query);
        // Print results.
        out.println("<TABLE BORDER=1>");
        ResultSetMetaData resultSetMetaData =
            resultSet.getMetaData();
        int columnCount = resultSetMetaData.getColumnCount();
        out.println("<TR>");
        // Column index starts at 1 (a la SQL), not 0 (a la Java).
        for(int i=1; i <= columnCount; i++) {
            out.print("<TH>" + resultSetMetaData.getColumnName(i));
        }
        out.println();
        // Step through each row in the result set.
        while(resultSet.next()) {
            out.println("<TR>");
            // Step across the row, retrieving the data in each
            // column cell as a String.
            for(int i=1; i <= columnCount; i++) {
                out.print("<TD>" + resultSet.getString(i));
            }
            out.println();
        }
        out.println("</TABLE>");
        connection.close();
    } catch(ClassNotFoundException cnfe) {
        System.err.println("Error loading driver: " + cnfe);
    } catch(SQLException sqle) {
        System.err.println("Error connecting: " + sqle);
    } catch(Exception ex) {
        System.err.println("Error with input: " + ex);
    }
}

private static void showResults(ResultSet results)
    throws SQLException {
    while(results.next()) {
        System.out.print(results.getString(1) + " ");
    }
    System.out.println();
}

private static void printUsage() {
    System.out.println("Usage: PreparedStatements host " +
        "dbName username password " +
        "vendor [print].");
}
}

```

清单17.4 NorthwindForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

```

<HTML>
<HEAD>
<TITLE>Simple Query Form</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>Query Input:</H2>
<FORM ACTION="/jdbc/servlet/coreservlets.NorthwindServlet"
      METHOD="POST">
<TABLE>
<TR><TD>Driver:
      <TD><INPUT TYPE="TEXT" NAME="driver"
                  VALUE="sun.jdbc.odbc.JdbcOdbcDriver" SIZE="35">
<TR><TD>URL:
      <TD><INPUT TYPE="TEXT" NAME="url"
                  VALUE="jdbc:odbc:Northwind" SIZE="35">
<TR><TD>Username:
      <TD><INPUT TYPE="TEXT" NAME="username">
<TR><TD>Password:
      <TD><INPUT TYPE="PASSWORD" NAME="password">
<TR><TD VALIGN="TOP">Query:
      <TD><TEXTAREA ROWS="5" COLS="35" NAME="query"></TEXTAREA>
<TR><TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="SUBMIT">
</TABLE>
</FORM>
</BODY></HTML>

```

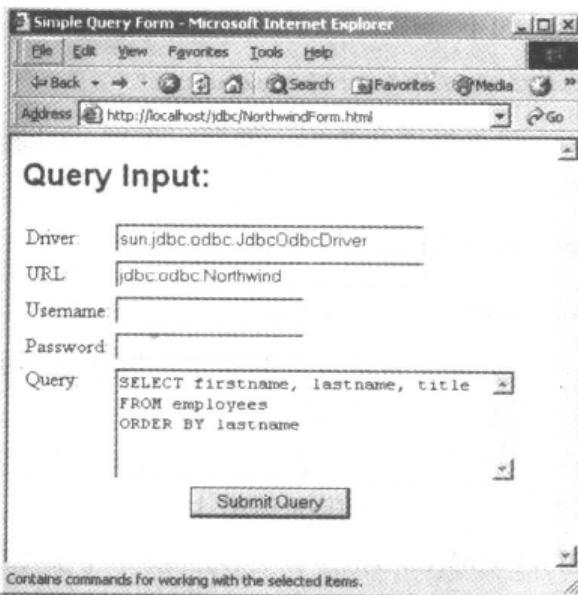


图 17.3 NorthwindForm.html：查询 Northwind 数据库的 servlet 的前端

在前述的例子中，HTML 表格由 servlet 内查询的结果产生。在本书第二卷中，我们提供各种定制标签，用以在 JSP 页面中根据查询结果生成 HTML 表格。此外，如果开发模型倾向于使用 JSP，那么，可以使用 JSP 标准标签库(JSP Standard Tag Library, JSTL)提供的 sql:query 动作来查询数据库，并将查询结果存储在作用域变量内，供 JSP 页面对其进行处理。JSTL 也在本书第二卷论述。



图 17.4 查询 Northwind 数据库的结果

17.3 用 JDBC 实用工具简化数据库访问

本节中，我们提供许多辅助类，贯穿本章都使用它们简化编码。这些类提供载入驱动程序和生成数据库连接的基本功能。

例如，DriverUtilities 类(清单 17.5)简化数据库连接 URL 的构建工作。比如，要构建 MySQL 中使用的 URL，形式如下

```
String url = "jdbc:mysql://host:3306/dbname";
```

我们首先要调用 loadDrivers 载入提供商数据。然后，调用 makeURL 构建该 URL，如下所示：

```
DriverUtilities.loadDrivers();
String url =
    DriverUtilities.makeURL(host, dbname, DriverUtilities.MYSQL);
```

其中，主机名、数据库名和提供商作为参数动态指定。通过这种方式，本章所有的例子都不需要对数据库的 URL 进行硬编码。更重要的是，我们可以简单地将数据库的信息添加到 DriverUtilities 中的 loadDrivers 方法(如果愿意的话，还可以提供指定驱动程序的常量)；之后，本章的所有例子都应该能够在我们使用的环境中运行。

另一个例子，ConnectionInfoBean 类(清单 17.9)提供一个实用方法——getConnection，它可以获取连接到数据库的 Connection。因此，在获取连接到数据库的连接时，可以将下面的代码：

```
Connection connection = null;
try {
    Class.forName(driver);
    connection = DriverManager.getConnection(url, username,
                                           password);
} catch (ClassNotFoundException cnfe) {
    System.err.println("Error loading driver: " + cnfe);
```

```

} catch(SQLException sqle) {
    System.err.println("Error connecting: " + sqle);
}

```

替换为:

```

Connection connection =
    ConnectionInfoBean.getConnection(driver, url,
                                      username, password);

```

如果在获取连接的过程中发生 SQLException，则返回 null。

在本节中，我们定义了 4 个实用工具类。

(1) DriverUtilities

这个类(清单 17.5)载入与各个数据库提供商相关的显式编码的驱动程序信息。之后，它提供相应的方法，获取提供商的驱动程序类(getDriver)，以及在给定主机名、数据库名和提供商的情况下创建一个 URL(makeURL)。我们提供 Microsoft Access、MySQL 和 Oracle 数据库的驱动程序信息，更新该类也比较容易。

(2) DriverUtilities2

这个类(清单 17.6)对 DriverUtilities(清单 17.5)进行扩展，并覆盖 loadDrivers 以便从 XML 文件中获取驱动程序的信息。drivers.xml(清单 17.7)给出一个具有代表性的 XML 文件。

(3) DriverInfoBean

DriverInfoBean 类(清单 17.8)封装特定提供商的驱动程序信息(供 DriverUtilities 使用，清单 17.5)。这个 bean 包含关键字(提供商名称)、驱动程序的简短描述、驱动程序类名、以及用来建立数据库连接的 URL。

(4) ConnectionInfoBean

这个类(清单 17.9)封装与特定数据库连接的相关信息。它封装了连接名、连接的简短描述、驱动程序类、连接到数据库的 URL、用户名和密码。另外，它还提供 getConnection 方法，使用这个方法可以直接得到连接到数据库的 Connection。

清单 17.5 DriverUtilities.java

```

package coreservlets;

import java.io.*;
import java.sql.*;
import java.util.*;
import coreservlets.beans.*;

/** Some simple utilities for building JDBC connections to
 * databases from different vendors. The drivers loaded are
 * hard-coded and specific to our local setup. You can
 * either modify the loadDrivers method and recompile or
 * use <CODE>DriverUtilities2</CODE> to load the driver
 * information for each vendor from an XML file.
 */

public class DriverUtilities {
    public static final String MSACCESS = "MSACCESS";
    public static final String MYSQL = "MYSQL";
}

```

```
public static final String ORACLE = "ORACLE";

// Add constant to refer to your database here ...

protected static Map driverMap = new HashMap();

/** Load vendor driver information. Here we have hard-coded
 * driver information specific to our local setup.
 * Modify the values according to your setup.
 * Alternatively, you can use <CODE>DriverUtilities2</CODE>
 * to load driver information from an XML file.
 */
<P>
* Each vendor is represented by a
* <CODE>DriverInfoBean</CODE> that defines a vendor
* name (keyword), description, driver class, and URL. The
* bean is stored in a driver map; the vendor name is
* used as the keyword to retrieve the information.
*<P>
* The url variable should contain the placeholders
* [$host] and [$dbName] to substitute for the <I>host</I>
* and <I>database name</I> in <CODE>makeURL</CODE>.
*/
public static void loadDrivers() {
    String vendor, description, driverClass, url;
    DriverInfoBean info = null;

    // MSAccess
    vendor = MSACCESS;
    description = "MS Access 4.0";
    driverClass = "sun.jdbc.odbc.JdbcOdbcDriver";
    url = "jdbc:odbc:[$dbName]";
    info = new DriverInfoBean(vendor, description,
                               driverClass, url);
    addDriverInfoBean(info);

    // MySQL
    vendor = MYSQL;
    description = "MySQL Connector/J 3.0";
    driverClass = "com.mysql.jdbc.Driver";
    url = "jdbc:mysql://[$host]:3306/[{$dbName}]";
    info = new DriverInfoBean(vendor, description,
                               driverClass, url);
    addDriverInfoBean(info);

    // Oracle
    vendor = ORACLE;
    description = "Oracle9i Database";
    driverClass = "oracle.jdbc.driver.OracleDriver";
    url = "jdbc:oracle:thin:@[$host]:1521:[{$dbName}]";
    info = new DriverInfoBean(vendor, description,
                               driverClass, url);
    addDriverInfoBean(info);

    // Add info on your database here...
}

/** Add information (<CODE>DriverInfoBean</CODE>) about new
```

```
* vendor to the map of available drivers.  
*/  
  
public static void addDriverInfoBean(DriverInfoBean info) {  
    driverMap.put(info.getVendor().toUpperCase(), info);  
}  
  
/** Determine if vendor is represented in the loaded  
 * driver information.  
 */  
  
public static boolean isValidVendor(String vendor) {  
    DriverInfoBean info =  
        (DriverInfoBean)driverMap.get(vendor.toUpperCase());  
    return(info != null);  
}  
  
/** Build a URL in the format needed by the  
 * database drivers. In building of the final URL, the  
 * keywords [$host] and [$dbName] in the URL  
 * (looked up from the vendor's <CODE>DriverInfoBean</CODE>)  
 * are appropriately substituted by the host and dbName  
 * method arguments.  
 */  
  
public static String makeURL(String host, String dbName,  
                             String vendor) {  
    DriverInfoBean info =  
        (DriverInfoBean)driverMap.get(vendor.toUpperCase());  
    if (info == null) {  
        return(null);  
    }  
    StringBuffer url = new StringBuffer(info.getURL());  
    DriverUtilities.replace(url, "[\$host]", host);  
    DriverUtilities.replace(url, "[\$dbName]", dbName);  
    return(url.toString());  
}  
  
/** Get the fully qualified name of a driver. */  
  
public static String getDriver(String vendor) {  
    DriverInfoBean info =  
        (DriverInfoBean)driverMap.get(vendor.toUpperCase());  
    if (info == null) {  
        return(null);  
    } else {  
        return(info.getDriverClass());  
    }  
}  
  
/** Perform a string substitution, where the first "match"  
 * is replaced by the new "value".  
 */  
  
private static void replace(StringBuffer buffer,  
                           String match, String value) {  
    int index = buffer.toString().indexOf(match);  
    if (index > 0) {
```

```
        buffer.replace(index, index + match.length(), value);
    }
}
}
```

在 DriverUtilities 中，每个提供商(Microsoft Access, MySQL 和 Oracle9i)的驱动程序信息都显式地编码在程序之中。如果您使用不同的数据库，则需要修改 DriverUtilities，使之包括您使用的驱动程序的信息，之后对代码重新编译。由于这种方式可能会不方便，所以我们在清单 17.6 中引入了第二个程序——DriverUtilities2，它从 XML 文件中读取驱动程序的信息。这样，如果要将新的数据库提供商加入到程序中，只需编辑这个 XML 文件。清单 17.7 给出 XML 文件的具体例子，drivers.xml。

在命令行应用程序中使用 DriverUtilites2 时，需要将驱动程序文件——drivers.xml，放在启动应用程序的工作目录中。此后，用完整的文件名(含路径)调用 loadDrivers。

对于 Web 应用，我们推荐将 drivers.xml 放在 WEB-INF 目录中。一般会将文件名作为 web.xml 中上下文相关的初始化参数(详细情况，参见本书第二卷介绍 web.xml 的章节)。同样，还要记住，在 servlet 中可以使用 getRealPath 得出相对于 Web 应用目录的文件的物理路径，如下面的代码段所示。

```
ServletContext context = getServletContext();
String path = context.getRealPath("/WEB-INF/drivers.xml");
```

JDK 1.4 中包括分析 XML 文档 drivers.xml 所需的所有类。如果您使用 JDK 1.3 或更早期的版本，可能需要下载和安装 SAX 或 DOM 分析程序。Apache 的 Xerces-J 是一个出色的分析程序，可以在 <http://xml.apache.org/xerces2-j/> 得到它。大多数 Web 应用服务器都绑定有 XML 分析程序，因此，您可能不需要下载 Xerces-J。检查提供商的文档以确定分析程序的位置，并在 CLASSPATH 中包括这个位置，以便能够编译您的应用。例如，Tomcat 4.x 在 install_dir/common/endorsed 目录中提供分析程序的 JAR 文件(xercesImpl.jar 和 xmlParserAPI.jar)。

要注意，如果在与 J2EE 1.4 完全兼容的服务器中使用 servlet 2.4(JSP 2.0)，则肯定可以访问到 JDK 1.4 或更新的版本。

清单 17.6 DriverUtilities2.java

```
package coreservlets;

import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import coreservlets.beans.*;

/** Extends <CODE>DriverUtilities</CODE> to support the
 * loading of driver information for different database vendors
 * from an XML file (default is drivers.xml). Both DOM and
 * JAXP are used to read the XML file. The format for the
 * XML file is:
 * <P>
 * <PRE>
 *   &lt;drivers&gt;
```

```
*      &lt;driver&gt;
*          &lt;vendor&gt;ORACLE&lt;/vendor&gt;
*          &lt;description&gt;Oracle&lt;/description&gt;
*          &lt;driver-class&gt;
*              oracle.jdbc.driver.OracleDriver
*          &lt;/driver-class&gt;
*          &lt;url&gt;
*              jdbc:oracle:thin:@[$host]:1521:[$dbName]
*          &lt;/url&gt;
*      &lt;/driver&gt;
*      ...
*  &lt;drivers&gt;
*  </PRE>
*  <P>
* The url element should contain the placeholders
* [$host] and [$dbName] to substitute for the host and
* database name in makeURL.
*/
public class DriverUtilities2 extends DriverUtilities {
    public static final String DEFAULT_FILE = "drivers.xml";

    /** Load driver information from default XML file,
     * drivers.xml.
     */
    public static void loadDrivers() {
        DriverUtilities2.loadDrivers(DEFAULT_FILE);
    }

    /** Load driver information from specified XML file. Each
     * vendor is represented by a <CODE>DriverInfoBean</CODE>
     * object and stored in the map, with the vendor name as
     * the key. Use this method if you need to load a
     * driver file other than the default, drivers.xml.
     */
    public static void loadDrivers(String filename) {
        File file = new File(filename);
        try {
            InputStream in = new FileInputStream(file);
            DocumentBuilderFactory builderFactory =
                DocumentBuilderFactory.newInstance();
            DocumentBuilder builder =
                builderFactory.newDocumentBuilder();
            Document document = builder.parse(in);
            document.getDocumentElement().normalize();
            Element rootElement = document.getDocumentElement();
            NodeList driverElements =
                rootElement.getElementsByTagName("driver");
            // Build DriverInfoBean for each vendor
            for(int i=0; i<driverElements.getLength(); i++) {
                Node node = driverElements.item(i);
                DriverInfoBean info =
                    DriverUtilities2.createDriverInfoBean(node);
                if (info != null) {
                    addDriverInfoBean(info);
                }
            }
        }
    }
}
```

```

    }
    } catch(FileNotFoundException fnfe) {
        System.err.println("Can't find " + filename);
    } catch(IOException ioe) {
        System.err.println("Problem reading file: " + ioe);
    } catch(ParserConfigurationException pce) {
        System.err.println("Can't create DocumentBuilder");
    } catch(SAXException se) {
        System.err.println("Problem parsing document: " + se);
    }
}

/** Build a DriverInfoBean object from an XML DOM node
 * representing a vendor driver in the format:
 * <P>
 * <PRE>
 *   &lt;driver&gt;
 *     &lt;vendor&gt;ORACLE&lt;/vendor&gt;
 *     &lt;description&gt;Oracle&lt;/description&gt;
 *     &lt;driver-class&gt;
 *       oracle.jdbc.driver.OracleDriver
 *     &lt;/driver-class&gt;
 *     &lt;url&gt;
 *       jdbc:oracle:thin:@[$host]:1521:[$dbName]
 *     &lt;/url&gt;
 *   &lt;/driver&gt;
 * </PRE>
 */
public static DriverInfoBean createDriverInfoBean(Node node) {
    Map map = new HashMap();
    NodeList children = node.getChildNodes();
    for(int i=0; i<children.getLength(); i++) {
        Node child = children.item(i);
        String nodeName = child.getNodeName();
        if (child instanceof Element) {
            Node textNode = child.getChildNodes().item(0);
            if (textNode != null) {
                map.put(nodeName, textNode.getNodeValue());
            }
        }
    }
    return(new DriverInfoBean((String)map.get("vendor"),
                             (String)map.get("description"),
                             (String)map.get("driver-class"),
                             (String)map.get("url")));
}
}

```

清单17.7 drivers.xml

```

<?xml version="1.0"?>
<!--
Used by DriverUtilities2. Here you configure information
about your database server in XML. To add a driver, include
a vendor keyword, description, driver-class, and URL.
For general use, the host and database name should not
be included in the URL; a special notation is required
-->

```

```
for the host and database name. Use [$host] as a
placeholder for the host server and {$dbName} as a placeholder
for the database name. Specify the actual host and database name
when making a call to makeUrl (DriverUtilities). Then, the
appropriate strings will be substituted for [$host]
and {$dbName} before the URL is returned.
-->
<drivers>
  <driver>
    <vendor>MSACCESS</vendor>
    <description>MS Access</description>
    <driver-class>sun.jdbc.odbc.JdbcOdbcDriver</driver-class>
    <url>jdbc:odbc:{$dbName}</url>
  </driver>
  <driver>
    <vendor>MYSQL</vendor>
    <description>MySQL Connector/J 3.0</description>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <url>jdbc:mysql://[$host]:3306/{$dbName}</url>
  </driver>
  <driver>
    <vendor>ORACLE</vendor>
    <description>Oracle</description>
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <url>jdbc:oracle:thin:@[$host]:1521:{$dbName}</url>
  </driver>
</drivers>
```

清单17.8 DriverInfoBean.java

```
package coreservlets.beans;

/** Driver information for a vendor. Defines the vendor
 * keyword, description, driver class, and URL construct for
 * connecting to a database.
 */

public class DriverInfoBean {
  private String vendor;
  private String description;
  private String driverClass;
  private String url;

  public DriverInfoBean(String vendor,
                        String description,
                        String driverClass,
                        String url) {
    this.vendor = vendor;
    this.description = description;
    this.driverClass = driverClass;
    this.url = url;
  }
  public String getVendor() {
    return(vendor);
  }

  public String getDescription() {
```

```
    return(description);
}

public String getDriverClass() {
    return(driverClass);
}

public String getURL() {
    return(url);
}
}
```

清单17.9 ConnectionInfoBean.java

```
package coreservlets.beans;

import java.sql.*;

/** Stores information to create a JDBC connection to
 * a database. Information includes:
 * <UL>
 *   <LI>connection name
 *   <LI>description of the connection
 *   <LI>driver classname
 *   <LI>URL to connect to the host
 *   <LI>username
 *   <LI>password
 * </UL>
 */

public class ConnectionInfoBean {
    private String connectionName;
    private String description;
    private String driver;
    private String url;
    private String username;
    private String password;

    public ConnectionInfoBean() { }

    public ConnectionInfoBean(String connectionName,
                            String description,
                            String driver,
                            String url,
                            String username,
                            String password) {
        setConnectionName(connectionName);
        setDescription(description);
        setDriver(driver);
        setURL(url);
        setUsername(username);
        setPassword(password);
    }

    public void setConnectionName(String connectionName) {
        this.connectionName = connectionName;
    }

    public String getConnectionName() {
```

```
    return(connectionName);
}

public void setDescription(String description) {
    this.description = description;
}

public String getDescription() {
    return(description);
}

public void setDriver(String driver) {
    this.driver = driver;
}

public String getDriver() {
    return(driver);
}

public void setURL(String url) {
    this.url = url;
}

public String getURL() {
    return(url);
}

public void setUsername(String username) {
    this.username = username;
}

public String getUsername() {
    return(username);
}

public void setPassword(String password) {
    this.password = password;
}

public String getPassword() {
    return(password);
}

public Connection getConnection() {
    return(getConnection(driver, url, username, password));
}

/** Create a JDBC connection or return null if a
 * problem occurs.
 */

public static Connection getConnection(String driver,
                                      String url,
                                      String username,
                                      String password) {
    try {
        Class.forName(driver);
        Connection connection =

```

```
        DriverManager.getConnection(url, username,
                                password);
    return(connection);
} catch(ClassNotFoundException cnfe) {
    System.err.println("Error loading driver: " + cnfe);
    return(null);
} catch(SQLException sqle) {
    System.err.println("Error connecting: " + sqle);
    return(null);
}
}
```

17.4 使用预备语句

如果需要多次执行类似的语句，那么，使用参数化(预备)语句要比每次都执行原始的查询更有效率。这种做法的主旨是，首先按照标准的格式创建参数化语句，在实际使用之前先发送到数据库进行编译。用问号表示语句中应该为具体的值所替换的位置。每次使用预备语句时，只需使用相应的 `setXxx` 调用(根据我们希望设置的项——使用以 1 为基的索引，以及参数的类型，如 `setInt`, `setString`)，替换语句中标记出来的参数。然后就可以和常规的语句一样，使用 `executeQuery`(如果希望返回 `ResultSet`)或 `execute`/`executeUpdate` 修改表中的数据。

例如，在 18.5 节中，我们创建一个 music 表，其中汇总各个古典作曲家的现有协奏曲唱片及价格。假定为了即将到来的销售旺季，您希望改变 music 表中所有唱片的价格，那么您可能会编写如下代码：

```
Connection connection =
    DriverManager.getConnection(url,username,password);
String template =
    UPDATE music SET price = ? WHERE id = ?;
PreparedStatement statement =
    Connection.prepareStatement(template);
Float[] newPrices = getNewPrices();
int[] recordingIDs = getIDs();
for(int i=0; i<recordingIDs.length, i++) {
    statement.setFloat(1,newPrices[i]); //price
    statement.setInt(2,recordingIDs[i]);//ID
    statement.execute();
}
```

依赖于服务器对预编译查询的支持，以及驱动程序处理原始查询的效率，预备语句在性能上的优势可能有很大的不同。例如，清单 17.10 中的类首先使用预备语句向数据库发送 100 个不同的查询，然后使用常规的语句重复同样的 100 个查询。一方面，在使用 PC，而且 PC 到 Oracle9i 数据库之间为快速局域网连接时，预备语句只花费原始查询所需时间的 62%，执行这 100 个查询的平均耗时为 0.61 秒，而常规语句的平均耗时为 0.99 秒(5 次平均)。另一方面，使用 MySQL(Connector/J 3.0)时，在快速局域网上，预备语句的耗时几乎等同于原始查询的耗时，仅仅节省了约 8% 的查询时间。要得到您所使用的环境中具体的性能表现，可以从 <http://wwwcoreservlets.com/> 下载 DriverUtilities.java，将您驱动程序的信息

息加入到其中，然后自己运行 PreparedStatements。有关 music 表的创建，参见 18.5 节。

但要注意：预备语句并不总是比普通的 SQL 语句执行得更快。性能上的提高可能依赖于您所执行的特定 SQL 命令。对 Oracle 中预备语句性能的更详细分析，参见 <http://www.oreilly.com/catalog/jorajdbc/chapter/ch19.html>。

然而，性能并非是预备语句的惟一优点。安全是预备语句的另一项优点。我们推荐在通过 HTML 表单接受用户输入，然后对数据库的值做出更新时，一定要使用预备语句或存储过程(参见 17.5 节)。相对于将用户输入的值经过转换拼接而形成 SQL 语句来说，我们强烈推荐这种方式。否则，聪明的攻击者可能提交特殊的表单值(类似于 SQL 语句的组成部分)，如果它们得到执行，那么，攻击者可能会对数据库做出不适当的访问和修改。我们经常将这种安全风险称为 SQL 注入攻击(SQL Injection Attack)。除了去除这类攻击风险之外，预备语句还能够正确地处理嵌入在字符串中的引号以及处理非字符数据(比如向数据库发送序列化后的对象)。

核心方法

从 HTML 表单接受数据时，为了避免 SQL 注入攻击，请使用预备语句或存储过程来更新数据库。

清单 17.10 PreparedStatements.java

```
package coreservlets;

import java.sql.*;
import coreservlets.beans.*;

/** An example to test the timing differences resulting
 * from repeated raw queries vs. repeated calls to
 * prepared statements. These results will vary dramatically
 * among database servers and drivers. With our setup
 * and drivers, Oracle9i prepared statements took only 62% of
 * the time that raw queries required, whereas MySQL
 * prepared statements took nearly the same time as
 * raw queries, with only an 8% improvement.
 */

public class PreparedStatements {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendor = args[4];
        // Use DriverUtilities2.loadDrivers() to load
        // the drivers from an XML file.
        DriverUtilities.loadDrivers();
        if (!DriverUtilities.isValidVendor(vendor)) {
            printUsage();
            return;
        }
        String driver = DriverUtilities.getDriver(vendor);
        String host = args[0];
```

```
String dbName = args[1];
String url =
    DriverUtilities.makeURL(host, dbName, vendor);
String username = args[2];
String password = args[3];
// Use "print" only to confirm it works properly,
// not when getting timing results.
boolean print = false;
if ((args.length > 5) && (args[5].equals("print"))) {
    print = true;
}
Connection connection =
    ConnectionInfoBean.getConnection(driver, url,
                                      username, password);
if (connection != null) {
    doPreparedStatements(connection, print);
    doRawQueries(connection, print);
}
try {
    connection.close();
} catch(SQLException sqle) {
    System.err.println("Problem closing connection: " + sqle);
}
}

private static void doPreparedStatements(Connection conn,
                                         boolean print) {
try {
    String queryFormat =
        "SELECT id FROM music WHERE price < ?";
    PreparedStatement statement =
        conn.prepareStatement(queryFormat);
    long startTime = System.currentTimeMillis();
    for(int i=0; i<100; i++) {
        statement.setFloat(1, i/4);
        ResultSet results = statement.executeQuery();
        if (print) {
            showResults(results);
        }
    }
    long stopTime = System.currentTimeMillis();
    double elapsedTime = (stopTime - startTime)/1000.0;
    System.out.println("Executing prepared statement " +
                       "100 times took " +
                       elapsedTime + " seconds.");
} catch(SQLException sqle) {
    System.err.println("Error executing statement: " + sqle);
}
}

public static void doRawQueries(Connection conn,
                                 boolean print) {
try {
    String queryFormat =
        "SELECT id FROM music WHERE price < ";
    Statement statement = conn.createStatement();
    long startTime = System.currentTimeMillis();
    for(int i=0; i<100; i++) {
```

```
ResultSet results =
    statement.executeQuery(queryFormat + i/4);
if (print) {
    showResults(results);
}
}
long stopTime = System.currentTimeMillis();
double elapsedTime = (stopTime - startTime)/1000.0;
System.out.println("Executing raw query " +
                    "100 times took " +
                    elapsedTime + " seconds.");
} catch(SQLException sqle) {
    System.err.println("Error executing query: " + sqle);
}
}

private static void showResults(ResultSet results)
throws SQLException {
while(results.next()) {
    System.out.print(results.getString(1) + " ");
}
System.out.println();
}

private static void printUsage() {
System.out.println("Usage: PreparedStatements host " +
                   "dbName username password " +
                   "vendor [print].");
}
}
```

前述的例子说明如何创建预备语句以及如何在命令行程序中设置该语句的参数。在进行 Web 开发时，您可能希望在 JSP 页面中向数据库提交预备语句。JSP 标准标签库(JSTL ——见本书第二卷)提供 `sql:query` 动作(定义提交给数据库的预备语句)和 `sql:param` 动作(为预备语句指定参数值)来完成这些任务。

17.5 创建可调用语句

通过 `CallableStatement`，我们可以执行数据库中的存储过程或函数。例如，在 Oracle 数据库中，我们可以用 PL/SQL 编写过程或函数，将它和数据表一同存储在数据库中。然后，我们可以创建到数据库的连接，通过 `CallableStatement` 执行存储的过程或函数。

存储过程有许多优点。例如，语法错误可以在编译时找出来，而非在运行期间；数据库过程的运行可能要比常规 SQL 查询快得多；程序员只需知道输入和输出参数，不需了解表的结构。另外，由于数据库语言能够访问数据库本地的一些功能(序列、触发器、多重游标)，因此，用它来编写存储过程可能要比使用 Java 编程语言要简易一些。

存储过程的缺点之一是需要学习新的数据库专有语言(但要注意，Oracle8i 数据库及其后续版本支持用 Java 编程语言编写的存储过程)。第二个缺点是存储过程的商业逻辑在数据库服务器上执行，而非客户机或 Web 服务器。而行业的发展趋势是尽可能多地将商业逻辑移出数据库，将它们放入 JavaBean 组件(或者，在大型的系统中，Enterprise JavaBean 组件)

中，在 Web 服务器上执行。在 Web 构架中采用这种方式的主要动机是，数据库访问和网络 I/O 常常是性能的瓶颈。

调用数据库中的存储过程涉及 6 个基本步骤，下面对它们进行逐一概括，并在随后的小节中详细介绍。

(1) 定义对数据库过程的调用。

如同预备语句，我们使用特殊的语法定义对存储过程的调用。过程的定义使用转义语法，由相应的? 定义输入和输出参数。

(2) 为这个过程准备 CallableStatement。

通过调用 prepareCall，从 Connection 获取 CallableStatement。

(3) 注册输出参数的类型。

在执行过程之前，必须声明每个输出参数的类型。

(4) 为输入参数提供值。

在执行过程之前，必须提供输入参数的值。

(5) 执行这个存储过程。

调用 CallableStatement 的 execute 方法执行数据库存储过程。

(6) 访问返回的输出参数。

依据输出的类型，调用对应的 getXxx 方法。

17.5.1 定义对数据库过程的调用

创建 CallableStatement 和创建 PreparedStatement 有些类似(参见 17.4 节)，二者都使用特殊的 SQL 转义语法，在语句执行之前要用相应的值替换其中的?。过程的定义一般有 4 种形式。

- 无参数过程

```
{ call procedure_name }
```

- 仅有输入参数的过程

```
{ call procedure_name(?, ?, ...) }
```

- 有一个输出参数的过程

```
{ ? call procedure_name }
```

- 既有输入参数又有输出参数的过程

```
{ ? = call procedure_name(?, ?, ...) }
```

在过程的 4 种形式中，procedure_name 都是数据库中存储过程的名称。同样，要注意，过程可能返回多个输出参数，并且，参数的索引值从输出参数开始。因此，前面最后一个例子中，第一个输入参数的索引值是 2(不是 1)。

重点提示

如果过程返回输出参数，那么输入参数的索引必须要考虑输出参数的数目。

17.5.2 为过程准备 CallableStatement

我们用 `prepareCall` 方法，从 `Connection` 获得 `CallableStatement`，如下所示。

```
String procedure = "{ ? = call procedure_name( ?, ? ) }";
CallableStatement statement =
    connection.prepareCall(procedure);
```

17.5.3 注册输出参数的类型

我们必须使用 `registerOutParameter` 注册每个输出参数的 JDBC 类型，如下所示：

```
statement.registerOutParameter(n, type);
```

其中 `n` 对应输出参数的序号(使用以 1 为基的索引)，`type` 对应 `java.sql.Types` 类中定义的常量(`Types.FLOAT`, `Types.DATE` 等)。

17.5.4 提供输入参数的值

在执行存储过程之前，我们需要调用与所要设置的项以及参数的类型相对应的 `setXxx`(例如 `setInt`, `setString`)，替换标记出来的输入参数。例如，下面的语句：

```
statement.setString(2, "name");
statement.setFloat(3, 26.0F);
```

将第一个输入参数设为一个 `String`，将第二个输入参数设为一个 `float`。切记，如果过程拥有输出参数，输入参数的索引从第一个输出参数开始计起。

17.5.5 执行这个存储过程。

要执行存储过程，只需调用 `CallableStatement` 对象的 `execute` 方法。例如：

```
statement.execute();
```

17.5.6 访问返回的输出参数。

如果过程返回输出参数，那么，在调用 `execute` 之后，可以通过调用 `getXxx(getDouble, getDate` 等等)访问每个对应的输出参数，其中的 `Xxx` 对应返回参数的类型。例如，

```
int value = statement.getInt(1);
```

返回第一个输出参数，类型为 `int`。

17.5.7 示例

在清单 17.11 中，`CallableStatements` 类演示 Oracle 存储过程(技术上，由于它返回一个值，应该称为是函数)的执行，这个存储过程是针对 `music` 数据表编写的(如何建立 `music` 表参见 18.5 节)。通过调用 `CallableStatements` 类并在命令行指定 `create`，可以在数据库中创建 `discount` 存储过程。这样做会调用 `createStoredFunction` 方法，这个方法将存储过程(一个长字符串)作为 SQL 更新提交给数据库。如果您有 Oracle SQL*Plus，则可以直接从 `discount.sql`(清单 17.12)载入这个过程。如何在 SQL*Plus 中运行 SQL 脚本参见 18.5 节。

存储过程 discount 修改 music 表中的 price 项。特别地，这个过程接受两个输入参数，composer_in(music 表中选定的作曲家)和 discount_in(价格折扣的百分比)。如果 discount_in 在 0.05~0.50 区间之外，则返回 -1；否则，存储过程返回数据表中被修改的行数。

清单 17.11 CallableStatements.java

```

package coreservlets;

import java.sql.*;
import coreservlets.beans.*;

/** An example that executes the Oracle stored procedure
 * "discount". Specifically, the price of all compositions
 * by Mozart in the "music" table are discounted by
 * 10 percent.
 * <P>
 * To create the stored procedure, specify a command-line
 * argument of "create".
 */

public class CallableStatements {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendor = args[4];
        // Change to DriverUtilities2.loadDrivers() to force
        // loading of vendor drivers from default XML file.
        DriverUtilities.loadDrivers();
        if (!DriverUtilities.isValidVendor(vendor)) {
            printUsage();
            return;
        }
        String driver = DriverUtilities.getDriver(vendor);
        String host = args[0];
        String dbName = args[1];
        String url =
            DriverUtilities.makeURL(host, dbName, vendor);
        String username = args[2];
        String password = args[3];

        Connection connection =
            ConnectionInfoBean.getConnection(driver, url,
                username, password);
        if (connection == null) {
            return;
        }

        try {
            if ((args.length > 5) && (args[5].equals("create"))) {
                createStoredFunction(connection);
            }
            doCallableStatement(connection, "Mozart", 0.10F);
        } catch (SQLException sqle) {
            System.err.println("Problem with callable: " + sqle);
        } finally {
    }
}

```

```
try {
    connection.close();
} catch(SQLException sqle) {
    System.err.println("Error closing connection: " + sqle);
}
}

private static void doCallableStatement(Connection connection,
                                         String composer,
                                         float discount)
throws SQLException {
CallableStatement statement = null;
try {
    connection.prepareCall("{ ? = call discount( ?, ? ) }");
    statement.setString(2, composer);
    statement.setFloat(3, discount);
    statement.registerOutParameter(1, Types.INTEGER);
    statement.execute();
    int rows = statement.getInt(1);
    System.out.println("Rows updated: " + rows);
} catch(SQLException sqle) {
    System.err.println("Problem with callable: " + sqle);
} finally {
    if (statement != null) {
        statement.close();
    }
}
}

/** Create the Oracle PL/SQL stored procedure "discount".
 * The procedure (technically, a PL/SQL function, since a
 * value is returned), discounts the price for the specified
 * composer in the "music" table.
 */
private static void createStoredFunction(
    Connection connection)
throws SQLException {
String sql = "CREATE OR REPLACE FUNCTION discount " +
            "(composer_in IN VARCHAR2, " +
            " discount_in IN NUMBER) " +
            "RETURN NUMBER " +
            "IS " +
            " min_discount CONSTANT NUMBER:= 0.05; " +
            " max_discount CONSTANT NUMBER:= 0.50; " +
            "BEGIN " +
            " IF discount_in BETWEEN min_discount " +
            "             AND max_discount THEN " +
            "     UPDATE music " +
            "     SET price = price * (1.0 - discount_in) " +
            "     WHERE composer = composer_in; " +
            "     RETURN(SQL%ROWCOUNT); " +
            " ELSE " +
            "     RETURN(-1); " +
            " END IF; " +
            "END discount;";
Statement statement = null;
```

```

try {
    statement = connection.createStatement();
    statement.executeUpdate(sql);
} catch(SQLException sqle) {
    System.err.println("Problem creating function: " + sqle);
} finally {
    if (statement != null) {
        statement.close();
    }
}
}

private static void printUsage() {
    System.out.println("Usage: CallableStatement host " +
        "dbName username password " +
        "vendor [create].");
}
}

```

清单17.12 discount.sql(Oracle的PL/SQL函数)

```

/*
 * Discounts the price of all music by the specified
 * composer, composer_in. The music is discounted by the
 * percentage specified by discount_in.
 *
 * Returns the number of rows modified, or -1 if the discount
 * value is invalid.
 */
CREATE OR REPLACE FUNCTION discount
    (composer_in IN VARCHAR2, discount_in IN NUMBER)
RETURN NUMBER
IS
    min_discount CONSTANT NUMBER:= 0.05;
    max_discount CONSTANT NUMBER:= 0.50;
BEGIN
    IF discount_in BETWEEN min_discount AND max_discount THEN
        UPDATE music
        SET price = price * (1.0 - discount_in)
        WHERE composer = composer_in;
        RETURN(SQL%ROWCOUNT);
    ELSE
        RETURN(-1);
    END IF;
END discount;

```

17.6 使用数据库事务

在更新数据库时，默认情况下，更改是永久性地写入(或提交)到数据库。然而，这种默认行为可以通过编写程序来关闭。在自动交付关闭的情况下，如果在更新时发生问题，则对数据库的每个更改都能够取消(或者说回退到最初的状态)。如果更新执行成功，那么，之后可以将这些更改永久性地提交给数据库。这种方式也称为事务管理。

事务管理可以帮助确保数据库中表的一致性。例如，假定您正在从储蓄账户向支票账

户传送资金。如果您首先从储蓄账户划出款项，然后将它存入支票账户。那么，如果在划出款项之后但在存入之前发生错误会发生什么问题呢？客户的账户将会损失钱，银行业务的管理者将会从中获益。另一方面，如果先存入支票账户，再从储蓄账户划出款项，且在存入之后但在划出款项之前发生错误，结果会怎样呢？客户的账户将会比实际的钱要多，银行的董事会会将所有的 IT 职员都解雇掉。重要的是，不管如何安排操作的次序，如果一项操作已提交，而另外的操作尚未提交，账户都会处于不一致状态。我们需要确保，要么所有的操作都发生，要么都不发生。这就是事务管理的内容。

数据库连接的默认设置为自动交付；即每个已执行的语句都自动提交给数据库。因此，为了进行事务管理，首先需要调用 `setAutoCommit(false)` 关闭连接的自动提交。

典型地，我们使用 `try/catch/finally` 块来正确地应对事务管理。首先，您应该记录自动提交的当前状态。然后，在 `try` 块中，调用 `setAutoCommit(false)` 并执行一系列的查询或更新。如果发生故障，则在 `catch` 块中调用 `rollback`；如果事务成功，则在 `try` 块的结尾调用 `commit`。不管哪种方式，都在 `finally` 块中重置自动提交的状态。

下面是这种事务管理方案的一个模板。

```
Connection connection =
    DriverManager.getConnection(url, username, password);
boolean autoCommit = connection.getAutoCommit();
Statement statement;
Try{
    connection.setAutoCommit(false);
    statement = connection.createStatement();
    statement.execute(...);
    statement.execute(...);

    ...
    connection.commit();
} catch(SQLException sqle) {
    connection.rollback();
} finally {
    statement.close();
    connection.setAutoCommit(autoCommit);
}
```

此处，从 `DriverManager` 获取连接的语句在 `try/catch` 块之外。这样，除非成功获取连接，否则不会调用 `rollback`。但是，`getConnection` 方法也会抛出 `SQLException` 异常，这个异常要么被外围的方法重新抛出，要么在单独的 `try/catch` 块中捕获。

清单 17.13 中，我们作为事务向向 `music` 表中添加唱片(有关 `music` 表的创建，参见 18.5 节)。为了将这项任务一般化，我们创建 `TransactionBean`——清单 17.14，其中，我们指定到数据库的连接，并且将 SQL 语句块以字符串数组的方式提交。然后，这个 `bean` 循环处理 SQL 语句的数组，执行每一个语句；并在有 `SQLException` 异常抛出时，执行 `rollback` 且将该异常重新抛出。

清单 17.13 Transactions.java

```
package coreservlets;

import java.sql.*;
import coreservlets.beans.*;
```

```
/** An example to demonstrate submission of a block of
 * SQL statements as a single transaction. Specifically,
 * four new records are inserted into the music table.
 * Performed as a transaction block so that if a problem
 * occurs, a rollback is performed and no changes are
 * committed to the database.
 */

public class Transactions {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendor = args[4];
        // Change to DriverUtilities2.loadDrivers() to load
        // vendor drivers from an XML file instead of loading
        // hard-coded vendor drivers in DriverUtilities.
        DriverUtilities.loadDrivers();
        if (!DriverUtilities.isValidVendor(vendor)) {
            printUsage();
            return;
        }
        String driver = DriverUtilities.getDriver(vendor);
        String host = args[0];
        String dbName = args[1];
        String url =
            DriverUtilities.makeURL(host, dbName, vendor);
        String username = args[2];
        String password = args[3];
        doTransactions(driver, url, username, password);
    }

    private static void doTransactions(String driver,
                                       String url,
                                       String username,
                                       String password) {
        String[] transaction =
        { "INSERT INTO music VALUES " +
          " ( 9, 'Chopin',      'No. 2 in F minor', 100, 17.99)",
          "INSERT INTO music VALUES " +
          " (10, 'Tchaikovsky', 'No. 1 in Bb minor', 100, 24.99)",
          "INSERT INTO music VALUES " +
          " (11, 'Ravel',        'No. 2 in D major', 100, 14.99)",
          "INSERT INTO music VALUES " +
          " (12, 'Schumann',     'No. 1 in A minor', 100, 14.99)" };
        TransactionBean bean = new TransactionBean();
        try {
            bean.setConnection(driver, url, username, password);
            bean.execute(transaction);
        } catch (SQLException sqle) {
            System.err.println("Transaction failure: " + sqle);
        } finally {
            bean.close();
        }
    }

    private static void printUsage() {
```

```
        System.out.println("Usage: Transactions host " +
                           "dbName username password " +
                           "vendor.");
    }
}
```

清单17.14 TransactionBean.java

```
package coreservlets.beans;

import java.io.*;
import java.sql.*;
import java.util.*;
import coreservlets.*;

/** Bean for performing JDBC transactions. After specifying the
 * the connection, submit a block of SQL statements as a
 * single transaction by calling execute. If an SQLException
 * occurs, any prior statements are automatically rolled back.
 */

public class TransactionBean {
    private Connection connection;

    public void setConnection(Connection connection) {
        this.connection = connection;
    }

    public void setConnection(String driver, String url,
                             String username, String password) {
        setConnection(ConnectionInfoBean.getConnection(
            driver, url, username, password));
    }

    public Connection getConnection() {
        return(connection);
    }

    public void execute(List list) throws SQLException {
        execute((String[])list.toArray(new String[list.size()]));
    }

    public void execute(String transaction)
        throws SQLException {
        execute(new String[] { transaction });
    }

    /** Execute a block of SQL statements as a single
     * transaction. If an SQLException occurs, a rollback
     * is attempted and the exception is thrown.
     */

    public void execute(String[] transaction)
        throws SQLException {
        if (connection == null) {
            throw new SQLException("No connection available.");
        }
        boolean autoCommit = connection.getAutoCommit();
```

```

try {
    connection.setAutoCommit(false);
    Statement statement = connection.createStatement();
    for(int i=0; i<transaction.length; i++) {
        statement.execute(transaction[i]);
    }
    statement.close();
} catch(SQLException sqle) {
    connection.rollback();
    throw sqle;
} finally {
    connection.commit();
    connection.setAutoCommit(autoCommit);
}
}

public void close() {
    if (connection != null) {
        try {
            connection.close();
        } catch(SQLException sqle) {
            System.err.println(
                "Failed to close connection: " + sqle);
        } finally {
            connection = null;
        }
    }
}
}

```

前述的例子演示了 bean 在向数据库提交事务中的应用。这种方式极适合于 servlet；但在 JSP 页面中，有时我们需要使用 JSP 标准标签库(JSTL)中的 sql:transaction 操作。JSTL 的详细介绍参见本书第二卷。

17.7 使用 ORM 框架将数据映射到对象

由于我们需要数据能够容易地从数据库到 Java 对象间来回移动，为了将对象映射到关系型数据库，众多的提供商开发了多种框架。由于面向对象的编程和关系型数据库一直存在不匹配性，故而，这是一种强大的能力：对象理解状态和行为，并与其他对象通过各种关系交织在一起；而关系型数据库则在表中存储信息，但一般通过主键关联起来。

表 17.1 汇总了几个流行的对象-关系映射(ORM)框架。当然还存在大量其他 ORM 框架。有关基于 Java 的 ORM 产品之间的比较，参见 <http://c2.com/cgi-bin/wiki?ObjectRelationalToolComparison>。ORM 框架和教程的其他优秀资源，请访问 <http://www.javaskyline.com/database.html>。(不要忘记，本书的源代码档案文件站点 <http://www.coreservlets.com/> 含有本书提及的所有 URL 的最新链接。)

表 17.1 对象-关系映射框架

框 架	URL
Castor	http://castor.exolab.org/
CocoBase	http://www.cocobase.com/
FrontierSuite	http://www.objectfrontier.com/
Kodo JDO	http://www.solarmetric.com/
ObjectRelationalBridge	http://db.apache.org/obj/
TopLink	http://otn.oracle.com/products/ias/toplink/

这类框架中多数(尽管不是全部)都支持 Java 数据对象(Java Data Object, JDO)API。JDO API 为管理映射到持续性存储介质(数据库)的对象提供完全的面向对象方案。ORM 及 JDO 的详细介绍超出了本书的范围。但我们会提供简短的汇总，介绍 JDO 所提供的功能。有关 JDO 的更多信息，参见 <http://java.sun.com/products/jdo/> 和 <http://jdocentral.com/> 处的在线资料。另外，还可以参考 Sameer Tyagi 等著的 *Core Java Data Objects*。

在 JDO 框架中，开发人员必须为映射到关系型数据库的每个 Java 类提供基于 XML 的元数据。元数据定义每个类中的持续性字段，以及每个字段相对于数据库的潜在角色(例如主键)。定义好每个 Java 类的元数据和源代码之后，开发人员必须运行框架的实用工具生成必需的 JDO 代码，来支持在持续性存储介质(数据库)中保存对象的字段。

JDO 框架的实用工具可以采用两种方式来修改 Java 类，使之支持持续性存储：第一种方式是在编译前修改源代码；第二种方式是在编译完源代码之后修改字节码(.class 文件)。第二种方式更常见，因为它简化了源代码的维护——开发人员看不到所生成的数据库代码。

用于 SQL 数据库的 JDO 实现中，框架的实用工具可以生成 JDO 持续性管理器所需的全部代码，包括向数据库插入(INSERT)新行，以及执行对数据做出持久性修改的 UPDATE 和 DELETE 操作。开发人员不需要编写任何 SQL 或 JDBC 代码；框架的实用工具生成所有必需的代码，持续性管理器生成所有与数据库之间必需的通信。框架建立之后，开发人员只需创建对象，并了解 JDO API。

清单 17.15 给出 Music.jdo，它说明在 SolarMetric's Kodo JDO 实现中如何定义 Music 类的元数据。XML 文件 Music.jdo 将 Music 类映射到数据库中的表，它使用 extension 元素(属性 key 为 table，属性 value 为 music)。数据库中不必事先存在名为 music 的表；实际上，Kodo 框架在数据库中创建持续性存储所需的全部表格，有时也可能使用稍加修改的表名。name 属性只是为框架定义一个映射。

这个.jdo 文件为必须保存在数据库中的类的每个字段指定一个 field 元素。每个 field 元素定义一个 extension 元素，将类中的字段映射到数据库表中的列，value 属性明确数据库列的名称。extension 元素与提供商相关，因此一定要参考 JDO 提供商的文档，找出 key 属性的正确值。

PersistenceManager 类提供对持续性存储介质(数据库)的访问。例如，清单 17.17 给出如何用 makePersistentAll 方法将字段(与新对象相关联)插入到持续性存储介质中。对持续性存储介质的更改被作为事务来管理，必须放在 Transaction 类的 begin 和 commit 调用之间。因此，要将与 Music 对象相关联的字段插入到数据库中，只需调用 Music 对象中相应的

`setKxx` 方法, 之后在事务内调用 `makePersistentAll` 方法。JDO 持续性管理器自动创建和执行将数据交付给数据库的 SQL 语句。类似地, 与 `Music` 对象相关联的字段的删除由 `PersistenceManager` 的 `makeDeletePersistent` 方法来处理。为了与持续性存储介质进行更为复杂的交互, JDO 提供 `Query` 类来执行查询, 并以对象集合(Collection)返回相应的结果。

最后, 数据库的位置、用户名、密码和其他系统专有的信息都从系统属性中读取(本例中由清单 17.18 指定)。

清单 17.15 Music.jdo

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"
"http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
  <package name="coreservlets.jdo">
    <class name="Music" >
      <extension vendor-name="kodo"
        key="table" value="music"/>
      <extension vendor-name="kodo"
        key="lock-column" value="none"/>
      <extension vendor-name="kodo"
        key="class-column" value="none"/>
      <field name="id" primary-key="true">
        <extension vendor-name="kodo"
          key="data-column" value="id"/>
      </field>
      <field name="composer">
        <extension vendor-name="kodo"
          key="data-column" value="composer"/>
      </field>
      <field name="concerto">
        <extension vendor-name="kodo"
          key="data-column" value="concerto"/>
      </field>
      <field name="available">
        <extension vendor-name="kodo"
          key="data-column" value="available"/>
      </field>
      <field name="price">
        <extension vendor-name="kodo"
          key="data-column" value="price"/>
      </field>
    </class>
  </package>
</jdo>
```

清单 17.16 Music.java

```
package coreservlets.jdo;

/** Music object corresponding to a record in a database.
 * A Music object/record provides information about
 * a concerto that is available for purchase and
 * defines fields for the ID, composer, concerto,
 * items available, and sales price.
```

```
*/  
  
public class Music {  
    private int id;  
    private String composer;  
    private String concerto;  
    private int available;  
    private float price;  
  
    public Music() {}  
  
    public Music(int id, String composer, String concerto,  
                int available, float price) {  
        setId(id);  
        setComposer(composer);  
        setConcerto(concerto);  
        setAvailable(available);  
        setPrice(price);  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public int getId() {  
        return(id);  
    }  
  
    public void setComposer(String composer) {  
        this.composer = composer;  
    }  
  
    public String getComposer() {  
        return(concerto);  
    }  
  
    public void setConcerto(String concerto) {  
        this.concerto = concerto;  
    }  
  
    public String getConcerto() {  
        return(composer);  
    }  
  
    public void setAvailable(int available) {  
        this.available = available;  
    }  
  
    public int getAvailable() {  
        return(available);  
    }  
  
    public void setPrice(float price) {  
        this.price = price;  
    }  
  
    public float getPrice() {  
        return(price);  
    }  
}
```

```

    }
}

```

清单 17.17 PopulateMusicTable.java

```

package coreservlets.jdo;

import java.util.*;
import java.io.*;
import javax.jdo.*;

/** Populate database with music records by using JDO.
 */

public class PopulateMusicTable {
    public static void main(String[] args) {
        // Create seven new music objects to place in the database.
        Music[] objects = {
            new Music(1, "Mozart", "No. 21 in C# minor", 7, 24.99F),
            new Music(2, "Beethoven", "No. 3 in C minor", 28, 10.99F),
            new Music(3, "Beethoven", "No. 5 Eb major", 33, 10.99F),
            new Music(4, "Rachmaninov", "No. 2 in C minor", 9, 18.99F),
            new Music(5, "Mozart", "No. 24 in C minor", 11, 21.99F),
            new Music(6, "Beethoven", "No. 4 in C", 33, 12.99F),
            new Music(7, "Liszt", "No. 1 in Eb major", 48, 10.99F)
        };

        // Load properties file with JDO information. The properties
        // file contains ORM Framework information specific to the
        // vendor and information for connecting to the database.
        Properties properties = new Properties();
        try {
            FileInputStream fis =
                new FileInputStream("jdo.properties");
            properties.load(fis);
        } catch(IOException ioe) {
            System.err.println("Problem loading properties file: " +
                               ioe);
        }
        return;
    }

    // Initialize manager for persistence framework.
    PersistenceManagerFactory pmf =
        JDOHelper.getPersistenceManagerFactory(properties);
    PersistenceManager pm = pmf.getPersistenceManager();

    // Write the new Music objects to the database.
    Transaction transaction = pm.currentTransaction();
    transaction.begin();
    pm.makePersistentAll(objects);
    transaction.commit();
    pm.close ();
}
}

```

清单 17.18 jdo.properties

```
# Configuration information for Kodo JDO Framework and
```

```
# MySQL database.
javax.jdo.PersistenceManagerFactoryClass=
    com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory
javax.jdo.option.RetainValues=true
javax.jdo.option.RestoreValues=true
javax.jdo.option.Optimistic=true
javax.jdo.option.NontransactionalWrite=false
javax.jdo.option.NontransactionalRead=true
javax.jdo.option.Multithreaded=true
javax.jdo.option.MsWait=5000
javax.jdo.option.MinPool=1
javax.jdo.option.MaxPool=80
javax.jdo.option.IgnoreCache=false
javax.jdo.option.ConnectionUserName: brown
javax.jdo.option.ConnectionURL: jdbc:mysql://localhost/csajsp
javax.jdo.option.ConnectionPassword: larry
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure=true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass=
    com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping=true
com.solarmetric.kodo.EnableQueryExtensions=false
com.solarmetric.kodo.DefaultFetchThreshold=30
com.solarmetric.kodo.DefaultFetchBatchSize=10
com.solarmetric.kodo.LicenseKey=5A8A-D98C-DB5F-6070-6000
```

第 18 章 配置 MS Access, MySQL 和 Oracle9i

本章的主题：

- 配置连接 Microsoft Access 的 DSN
- MySQL 的安装和配置
- 在 MySQL 中创建数据库和用户
- Oracle9i 的安装和配置
- 使用 Database Configuration Assistant 在 Oracle9i 中创建数据库
- 在 Oracle9i 中手动创建数据库
- 在 Oracle9i 中创建用户
- 通过 JDBC 连接对数据库进行测试
- 确定数据库驱动程序的 JDBC 版本
- 建立 music 示例表

在本章中，我们详细地介绍 3 种流行数据库，Microsoft Access, MySQL 和 Oracle9i，在和 JDBC 一同使用时，应该如何配置。

第一种数据库，Microsoft Access，由于 Java SDK(或 JDK)中已经包括了相应的 JDBC 驱动程序，并且许多开发人员都安装有 Access，故而它是练习和试验的绝佳选择。然而，在正式的应用中一般不太可能使用 Microsoft Access，因为它的设计目标就非针对处理大量的并发连接。有关 Microsoft Access 配置的详细信息，参见 18.1 节。

第二种数据库，MySQL，是产品级的数据库产品，并且可能是免费情况下最好的选择。在 18.2 节中，我们提供安装和配置 MySQL 的细节。另外，我们还提供相关的下载信息，并介绍如何在 Web 应用中使用相应的 MySQL JDBC 驱动程序。

第三种数据库，Oracle9i，虽然不是免费，但的确是极好的产品数据库。安装和配置 Oracle9i 的详细信息参见 18.3 节。安装和创建数据库的过程十分冗长。然而，Oracle9i 在行业中得到广泛应用，所以花一些时间来熟悉这个产品绝对是值得的。在完成安装和数据库的创建之后，我们介绍如何安装恰当的 Oracle JDBC 驱动程序，使之和各种版本的 Java SDK 协同使用。

最后，我们提供相关的程序，对数据库进行测试，以及载入本书中用到的示例数据表。

18.1 配置 Microsoft Access 与 JDBC 的使用

如果安装了 Microsoft Office，那么 Microsoft Access 和所需的开放数据库互连(Open DataBase Connectivity, ODBC)驱动程序可能已经安装在计算机之上。因此，尽管对于高端的产品级网站我们不推荐使用 Microsoft Access，但我们认为 Microsoft Access 对于学习和测试 JDBC 代码却是极好的。例如，第 17 章中的例子都连接到 Microsoft Access 预装的

Northwind 数据库。产品级网站应该使用更为健壮的产品，如 Oracle9i, DB2, Sybase, Microsoft SQL Server 或 MySQL。

在从 Java 平台连接到 Microsoft Access 数据库时，我们可以使用 JDK 中的 JDBC-ODBC 桥接器，sun.jdbc.odbc.JdbcOdbcDriver。这个桥接器允许 JDBC 使用 ODBC 与数据库进行通信，从而不需要到本地格式的驱动程序。但我们需要配置一个 ODBC 数据源名(Data Source Name, DSN)，将名称映射到物理数据库。

用于连接到 Microsoft Access 数据库的 URL 并不指定主机名，相反地，它指向一个 DSN，例如 `jdbc:odbc:dsn.`，其中的 `dsn` 是通过 ODBC DSN 向导指定的数据库名。要注意，Sun 的驱动程序，sun.jdbc.odbc.JdbcOdbcDriver，与 JDBC 2.0 并不完全兼容，因而并不支持 JDBC 2.0 引入的全部高级 JDBC 特性。但是，对于本章讨论的这些功能来说已经绰绰有余了。在 <http://industry.java.sun.com/products/jdbc/drivers/> 可以找到 Microsoft Access 的 JDBC 2.0 驱动程序。

为了使应用程序能够连接到服务器上的数据库，服务器上必须安装有 ODBC 3.x 版本。幸运的是，许多 Microsoft 产品都会安装 ODBC。如果没有 ODBC，单独将它安装到系统之上也很容易。ODBC 与微软数据访问组件(Microsoft Data Access Component, MDAC)绑定在一起。系统上应该安装的 MDAC 的正确版本，参见 <http://www.microsoft.com/data/download.htm>。

通过 ODBC 管理工具(ODBC Administration Tool)配置系统 DSN 需要 4 步，下面我们将它们进行简要的概括，随后的小节中对它们进行详细的介绍。

- (1) 从 ODBC Data Source Administrator(【ODBC 数据源管理器】)中选择 System DSN(系统 DSN)。

Data Source Administrator(数据源管理器)在系统的 Administrative Tools(【管理工具】)中，它允许我们创建新的数据源名。

- (2) 为新的系统 DSN 选取驱动程序。

多个 ODBC 数据库驱动程序都可以供新定义的 DSN 选择。但一般会选择 Microsoft Access Driver。

- (3) 选择数据源。

找到并选择计算机上的数据库文件作为 ODBC 连接的数据源。在此，还要指定数据源的名称，应用程序连接到 ODBC 驱动程序时要用到它。

- (4) 选择 OK(【确定】)接受新的 DSN。

选择 OK(【确定】)完成 ODBC 系统数据源的配置。此后，就可以通过 JDBC-ODBC 桥接器从 Java 中连接到这个数据源。

18.1.1 从 ODBC 数据源管理器中选择系统 DSN

在 Windows 2000 中，我们可以通过选择 Start(【开始】), Settings(【设置】)，然后选择 Control Panel(【控制面板】)。接下来，选择 Administrative Tools(【管理工具】)，然后选择 Data Sources(【数据源】)。最后，选择 System DSN(【系统 DSN】)标签，并选择 Add(【添加】)创建新的 DSN。Windows 的其他版本也都类似，例如，在 Windows XP 上，除了从 Start 菜单直接选取 Control Panel 之外，其他步骤完全相同。图 18.1 展示出 ODBC 数据源

管理器窗口中的系统 DSN 标签。

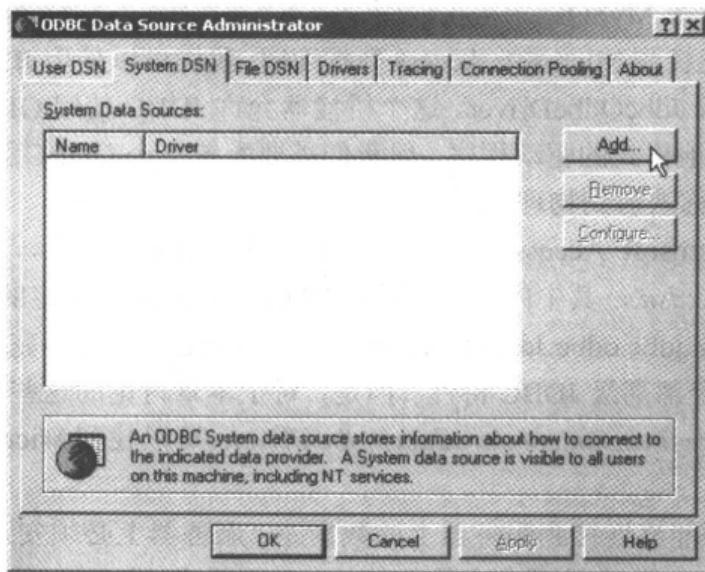


图 18.1 配置 ODBC 数据源时首先显示的窗口。选择 System DSN(【系统 DSN】) 标签，然后单击 Add(【添加】)按钮创建新的 DSN

18.1.2 为新的系统 DSN 选取驱动程序

在 Create New Data Source(【创建新数据源】)窗口中(图 18.2)，选择 Microsoft Access Driver(*.mdb)，之后选择 Finish(【完成】)。

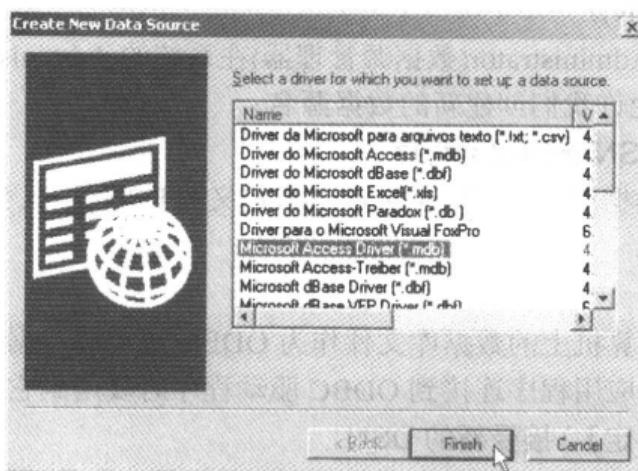


图 18.2 创建系统 DSN 时的第二个窗口。在继续进行配置之前，需要单击 Finish(【完成】)为数据源选取驱动程序

18.1.3 选择数据源

在 ODBC Microsoft Access Setup(【ODBC Microsoft Access 安装】)窗口中，输入数据源的名称(以及可选的描述)。这个 DSN 与用在 JDBC URL(jdbc:odbc:dsn)中的名称相同。例如，如果选择 Test 作为 DSN，则要将"jdbc:odbc:Test"作为 DriverManager.getConnection 的第一个参数。接下来，单击 Select(【选取】)按钮选取绑定到这个数据源名的物理数据库文

件, 如图 18.3 所示。完成这一步之后, 单击 OK(【确定】)按钮。

如果使用 Microsoft Access 自带的 Northwind 示例数据库, 这个数据库文件极有可能位于 C:\Program Files\Microsoft Office\Office\Samples\Northwind.mdb 或其他极为类似的目录, 这要依所安装的 Microsoft Access 版本而定。如果找不到 Northwind 文件, 可能需要打开 Microsoft Access, 并在打开的窗口中选择 Northwind Sample Database(【罗斯文示例数据库】), 来安装这个示例数据库。也可以从 <http://office.microsoft.com/downloads/2000/Nwind2k.aspx> 下载这个示例数据库。

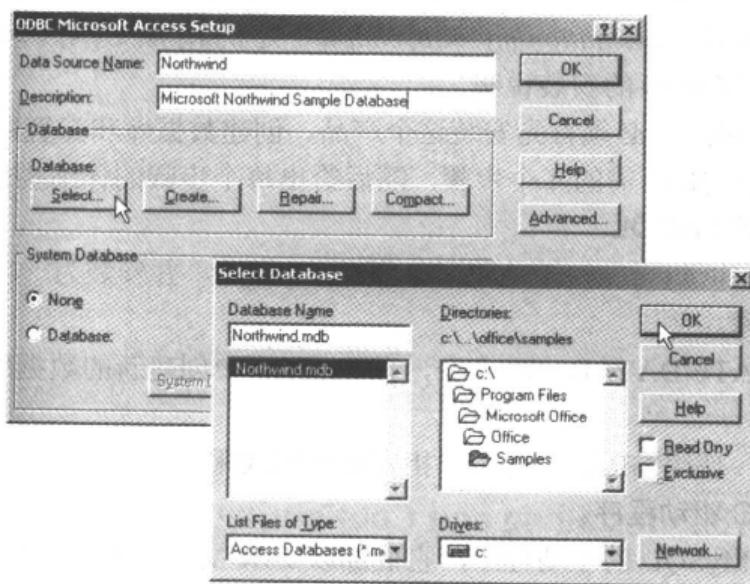


图 18.3 建立 Microsoft Access 数据库 DSN 的第三个窗口。指定数据源的名称(以及可选的描述), 之后选取绑定到数据源名的物理数据库文件

18.1.4 选择【确定】接受新的 DSN

此时, 新定义的 DSN 应该列在系统 DSN 标签中, 如图 18.4 所示。单击 OK(【确定】)结束配置。

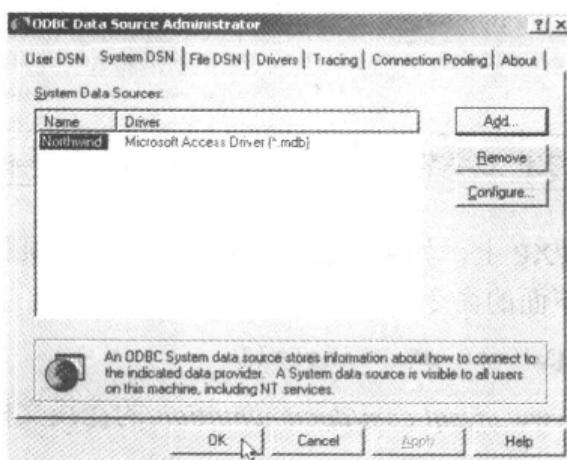


图 18.4 添加新的系统数据源之后弹出的第四个窗口。单击 OK(【确定】)接受这项更改

18.2 MySQL 的安装和配置

MySQL 是一种流行的开放源码数据库，它是免费的，且跨 Unix(Solaris, Linux 等), MacOS 和 Windows 等多种平台。本书交付印刷时，MySQL 的当前稳定版本是 4.0。MySQL 的 4.0 版本不支持存储过程和几项其他高级特性，但它是免费的，且拥有极高的性能。

下面是在 Windows 平台上下载和安装 MySQL 的细节。这些指示提供了 MySQL 的最简安装。对于安全问题(例如设置 root 密码)以及后期配置的指导原则，请参见 <http://www.mysql.com/documentation/mysql/bychapter/> 处的用法说明。在线文档还提供 Unix(包括 Linux)和 MacOS 的安装说明。

如果要使用 MySQL，必须首先安装这个产品、创建数据库和配置用户的权限。此处，我们概括了建立 MySQL 所需的 4 个步骤，随后给出每个步骤的详细描述。

(1) 下载并安装 MySQL。

从 <http://www.mysql.com/downloads/> 下载 MySQL，并安装为一项服务。

(2) 创建数据库。

输入 CREATE DATABASE 命令，在 MySQL 中创建新的数据库。

(3) 创建用户。

创建用户，使用 GRANT 为用户指定数据库权限。

(4) 安装 JDBC 驱动程序。

下载适合 MySQL 的驱动程序，它们都以 JAR 文件的形式提供。在开发过程中需要将 JAR 文件包括在 CLASSPATH 中。而在部署时则要将这个 JAR 文件放在 Web 应用的 WEB-INF/lib 目录中。

18.2.1 下载并安装 MySQL

可以从 <http://www.mysql.com/downloads/> 下载 MySQL。下载 mysql-4.0.xx-win.zip(或更新的版本)，解压缩，运行 setup.exe 程序安装 MySQL。我们推荐将 MySQL 安装到 C:\mysql 目录。要注意，在 Windows 上安装 MySQL 之前，所使用的登录用户必须拥有相应的管理权限。

警告

在 Windows NT/2000/XP 上安装 MySQL 时，必须拥有计算机的本地管理员权限。

在 Windows NT/2000/XP 上，如果要将 MySQL 配置为一项服务，请在 DOS 模式下从 C:\mysql\bin 目录中运行下面的命令。

```
c:\mysql\bin> mysqld-max-nt --install
```

详细的信息，参见 <http://www.mysql.com/documentation/> 的在线文档。

18.2.2 创建数据库

在创建数据库之前，必须启动 MySQL 服务器。可以在命令行输入 net start 命令启动这

项服务，如下所示。

```
C:\mysql\bin> net start MySql
```

如果服务器已经运行，系统会报告一个警告消息。

接下来，创建新数据库时，需要使用下面的命令，作为 root 用户启动 MySQL 监视器。

```
C:\mysql\bin> mysql.exe --user=root
```

然后，输入 CREATE DATABASE 命令创建数据库，如下所示。

```
mysql> CREATE DATABASE database_name;
```

database_name 就是希望创建的数据库的名字。我们为本书中的代码创建一个名为 csajsp 的数据库。要查看当前数据库的列表，可以输入下面的命令。

```
mysql> SHOW DATABASES;
```

如果您更喜欢图形界面，可以使用 MySQL Control Center 来管理您的服务器。MySQL Control Center 可以在 <http://www.mysql.com/downloads/mysqlcc.html> 找到。

18.2.3 创建用户

可以在创建用户的同时授予该用户特权。要准许用户从本地主机访问数据库，使用下面的命令：

```
mysql> GRANT ALL PRIVILEGES ON database.* TO user@localhost  
IDENTIFIED BY 'password';
```

database 是数据库的名字，*user* 是新用户的名称。要准许用户从其他客户机访问数据库，使用下面的命令：

```
mysql> GRANT ALL PRIVILEGES ON database.* TO user@"%"  
IDENTIFIED BY 'password';
```

其中 "@"%的作用是通配符，表示任何客户机对数据库的访问。如果创建新用户时发生问题，请检查是否作为 root 用户启动 MySQL 监视器。

18.2.4 安装 JDBC 驱动程序

常用来访问 MySQL 的 JDBC 驱动程序有两种：MySQL Connector/J 和 Caucho Resin 驱动程序。

MySQL 推荐使用 MySQL Connector/J 驱动程序，可以在 <http://www.mysql.com/products/connector-j/> 找到这个驱动程序。在我们的例子中，我们使用 Connector/J 驱动程序的 3.0 版本。这个驱动程序绑定在 JAR 文件中(mysql-connector-java-3.0.6-stable-bin.jar)，类名为 com.mysql.jdbc.Driver。使用 MySQL Connector/J 驱动程序时，相应的 URL 是 jdbc:mysql://host:3306/*dbName*，其中 *dbName* 是 MySQL 服务器中数据库的名字。

Caucho Resin 也在 <http://www.caucho.com/projects/jdbc-mysql/index.xtp> 提供一个 MySQL 驱动程序。这个驱动程序绑定在名为 caucho-jdbc-mysql-2.1.0.jar 的 JAR 文件中，类名为 com.caucho.jdbc.mysql.Driver。使用 Caucho Resin 驱动程序时，相应的 URL 是 jdbc:mysql-caucho://host:3306/*dbName*，同样，*dbName* 是 MySQL 服务器中数据库的名字。

因为 MySQL 与 ANSI SQL-92 不完全兼容, 故而两种驱动程序都非完全 JDBC 2.0 兼容。

在开发过程中要将 JAR 文件放在 CLASSPATH 中; 在部署时, 则要将 JAR 放在 Web 应用的 WEB-INF/lib 目录中。但是, 如果服务器上的多个应用都同时使用 MySQL 数据库, Web 管理员可能会选择将 JAR 文件移动到公共的 lib 目录中。例如, Tomcat 4.x 中, 多个应用使用的 JAR 文件可以放在 *install_dir/common/lib* 目录中。

18.3 Oracle9i 数据库的安装和配置

Oracle9i 数据库是高容量、产品级的数据库, 许多公司的 Internet 和 intranet 应用都使用它。Oracle9i 数据库提供产品级数据库服务器所应该提供的一切功能, 包括存储过程、视图、触发器、增强的安全性和数据恢复机制。

Oracle 提供 Oracle9i Database Release 2 的 3 种不同的版本, 如下所述。Oracle9i 包括一个庞大的产品家族(包括 Oracle9i Application Server 和 Oracle9i Developer Suite), 但在后文中, 我们都用“Oracle9i”指代 Oracle9i Database。

- 企业版

Oracle9i 企业版为 Internet 和 intranet 应用提供一个高效、可靠的解决方案。企业版适用于高容量的事务处理和数据仓库。企业版包括一个预先配置好的数据库、网络服务、数据库管理工具和实用程序。另外, 在企业版中, 许多产品选项可以获得许可。

- 标准版

Oracle9i 标准版是企业版按比例缩小的版本, 只许可不超过 4 个处理器的服务器使用。企业版适合于工作组、部门、intranet 和 Internet 应用。标准版包括一个预配置的数据库、网络服务、数据库管理工具和实用程序; 但标准版不支持企业版中的所有特性。

- 个人版

Oracle9i 个人版适用于单用户桌面环境。个人版倾向于教育目的, 它的年度许可费用具有很高的性价比。个人版支持企业版中的所有特性和选项, Oracle Real Application Clusters 例外。

有关 3 种 Oracle9i 数据库版本更详细的汇总, 请参见 http://otn.oracle.com/products/oracle9i/pdf/9idb_rel2_prod_fam.pdf。

如果要使用 Oracle9i, 必须首先安装该产品, 建立数据库, 并配置用户的权限。本节中, 我们提供 Oracle9i Release 2 在 Windows XP 上的下载与安装信息。对于其他平台, 可以在 <http://otn.oracle.com/docs/products/oracle9i/> 处找到平台相关的安装指示。下面, 我们概括安装 Oracle9i 所需的 4 个步骤, 随后给出每个步骤的详细描述。

- (1) 下载和安装 Oracle9i。

从 <http://otn.oracle.com/software/products/oracle9i/> 下载 Oracle9i Database Release 2, 并使用 Oracle Universal Installer 进行安装。

- (2) 创建数据库。

一般地, 在 Oracle9i 的安装过程中会创建一个数据库; 但是, 如果您使用的计算

机上已经安装了 Oracle9i, 您可以手动或使用 Database Configuration Assistant 创建新的数据库。

(3) **创建用户。**

要从 Web 应用中访问这个数据库, 需要创建新的用户, 并授予用户建立连接以及访问数据表的相关权限。

(4) **安装 JDBC 驱动程序。**

要从 Web 应用中访问 Oracle 数据库, 需要从 http://otn.oracle.com/software/tech/java/sqlj_jdbc/ 下载恰当的 JDBC 驱动程序。开发过程中要在 CLASSPATH 中包括相应的 JAR 文件。在部署时, 要将 JAR 文件放在 Web 应用的 WEB-INF/lib 目录中。

18.3.1 下载和安装 Oracle9i

我们可以从 <http://otn.oracle.com/software/products/oracle9i/> 下载 Oracle9i Database Release 2。下载 Oracle 的软件时需要注册; 但注册是免费的。如果计划将 Oracle9i 用作产品用途, 一定要仔细地阅读许可协议。Oracle 产品可以免费下载进行 30 天的评估。30 天之后, 必须购买许可。

在下面的说明中, 我们展示如何在 Windows XP 平台上安装 Oracle9i Database Release 2 个人版。其他平台的安装说明, 参见 <http://otn.oracle.com/docs/products/oracle9i/> 的文档。

Oracle9i Database Release 2 for Windows NT/2000/XP 打包在 3 个 ZIP 文件中: 92010NT_Disk1.zip(612 802 971 字节), 92010NT_Disk2.zip(537 604 934 字节) 和 92010NT_Disk3.zip(254 458 106 字节)。企业版、标准版和个人版都使用相同的安装文件。遵照下载页面给出的指示, 将这 3 个文件分别解压缩到对应的目录 Disk1, Disk2 和 Disk3。也可以不下载这个软件, 而在 <http://oraclestore.oracle.com/> 购买 CD 包。

Oracle 推荐的最低硬件要求如下: 奔腾 266, 256M 内存以及约 3G 的 NTFS 分区磁盘空间。精确的需求可以查看 http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/A95493-01/html/reqs.htm。

下面是在 Windows XP 计算机上的 C:\ 驱动器安装 Oracle9i Database Personal Edition 的指示。要执行这个安装过程, 执行安装的用户必须具有本地管理员权限。

警告

在 Windows NT/2000/XP 上安装 Oracle9i 时, 必须拥有计算机的本地管理员权限。

安装 Oracle9i 的步骤

(1) **启动 Oracle Universal Installer。**

可以通过 Disk1 目录中的 setup.exe 启动 Oracle Universal Installer 2.2。如果安装器不能启动, 可以试着运行 Disk1\install\win32 目录中的 setup.exe。启动安装器时, 将会暂时性地看到版本屏幕, 后面是欢迎屏幕, 如图 18.5 所示。单击 Next 按钮。

(2) 指定文件的位置。

在显示的第三个屏幕上(图 18.6)，可以指定安装程序的位置，以及安装 Oracle9i 的位置。接受默认值。Oracle Home(OraHome92)用在安装过程中创建的所有 Oracle 服务的名称中。单击 Next 继续。

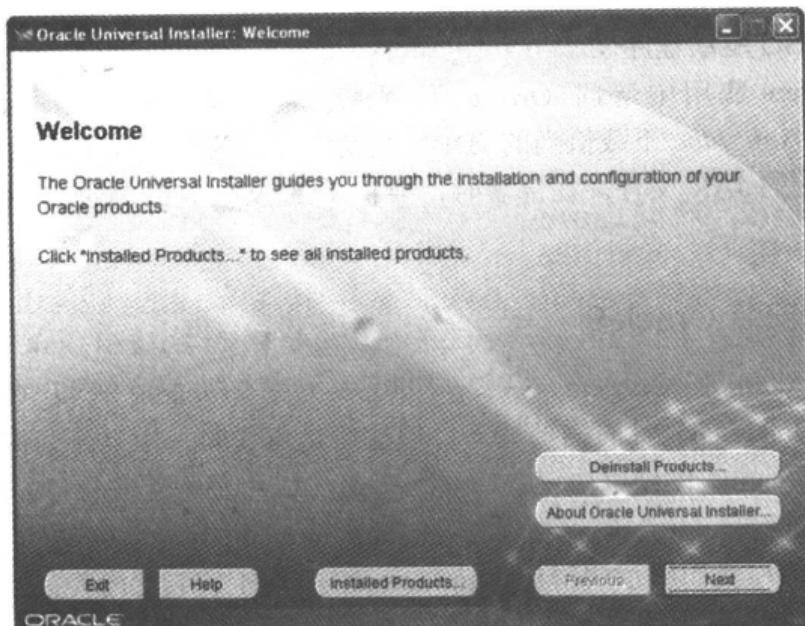


图 18.5 Oracle 的第二个安装窗口：欢迎信息

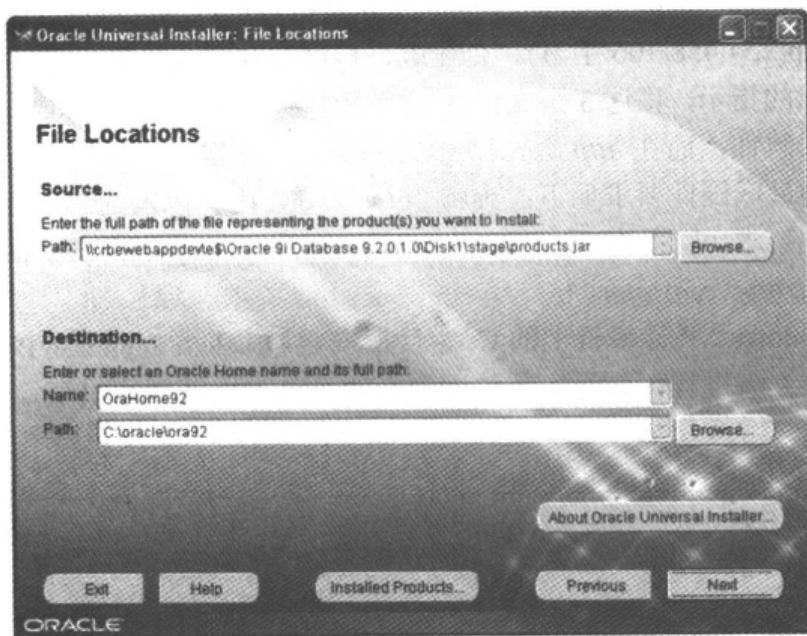


图 18.6 Oracle 的第三个安装窗口：汇总安装过程中源文件和目的文件的位置

(3) 选择安装的产品。

在显示的第四个屏幕上(图 18.7)，可以选择安装哪个产品。接受默认产品——Oracle9i Database，然后单击 Next。

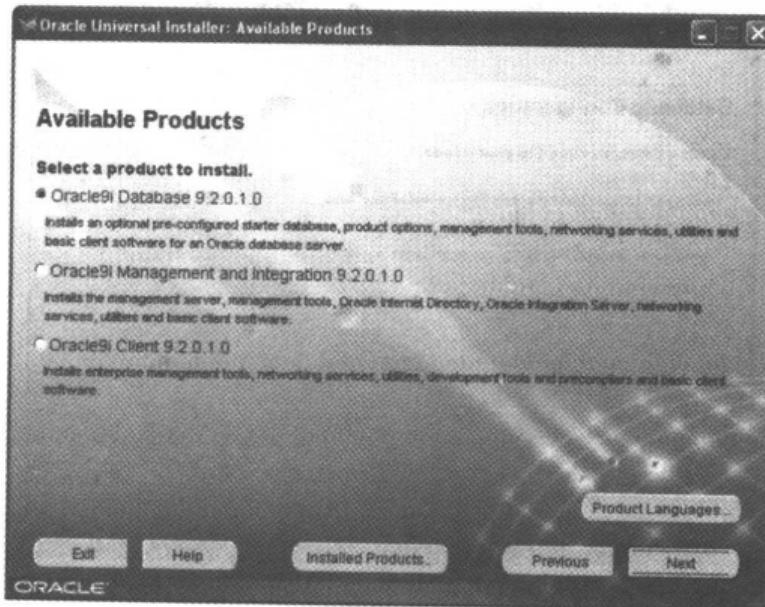


图 18.7 Oracle 的第四个安装窗口：选择安装的产品

(4) 选择安装类型。

在显示的第五个屏幕上(图 18.8)，可以选择安装数据库的哪种版本。在单用户环境中，我们推荐个人版(Personal Edition)，它在 Windows XP 上需要 2.53G 的磁盘空间。有关这 3 种版本更详细的信息，参见 http://otn.oracle.com/products/oracle9i/pdf/9idb_rel2_prod_fam.pdf。单击 Next。

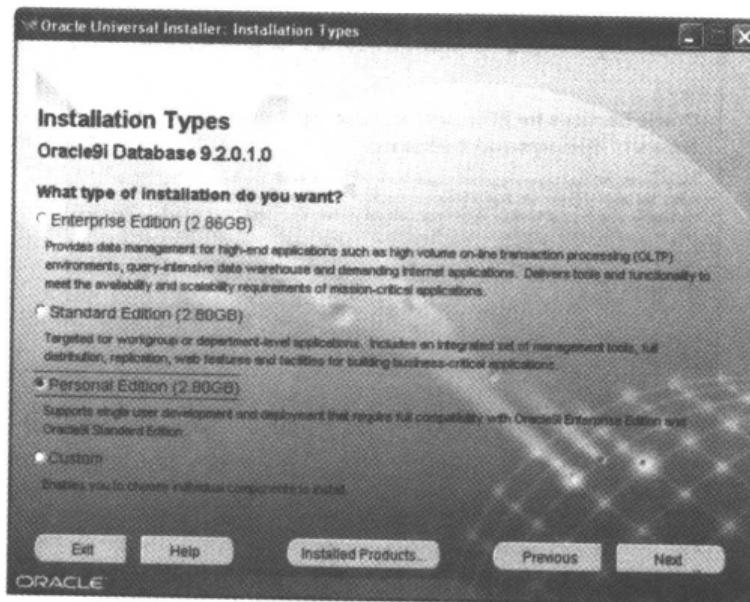


图 18.8 Oracle 的第五个安装窗口：选择安装的类型。对于单用户，选取个人版

(5) 选择数据库配置。

在显示的第六个屏幕上(图 18.9)，指定数据库配置。我们推荐默认的选择，General Purpose(常规用途)，这是因为对应这项选择的安装过程会自动创建一个起始数据库。接受默认的数据库配置，并单击 Next。

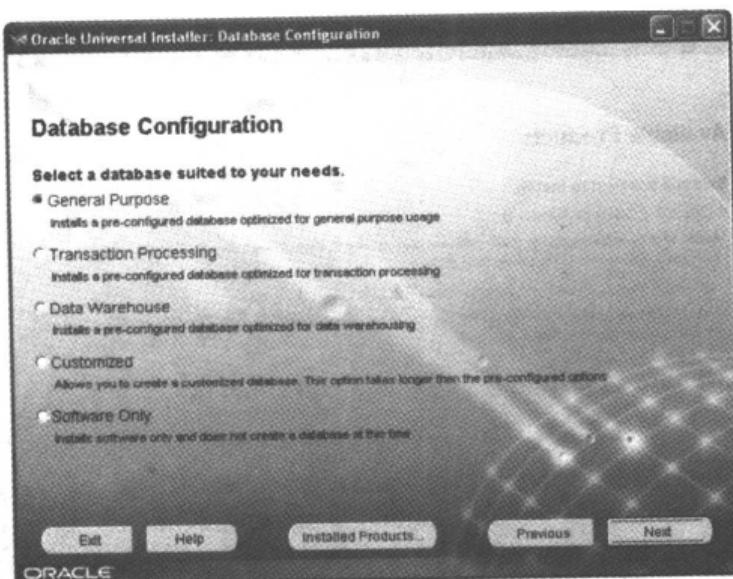


图 18.9 Oracle 的第六个安装窗口：选择数据库配置

(6) 指定 Oracle MTS Recovery Service 的端口。

在显示的第七个屏幕上(图 18.10)，指定 Oracle MTS Recovery Service 的端口，Oracle MTS Recovery Service 随同针对 Microsoft Transaction Server 的 Oracle 服务一同安装。这个服务处理由 Microsoft DTC 协调的分布式事务请求(MS DTC 暴露一系列 COM 对象，这些对象允许客户启动和参与跨多种连接的事务协调)。或许您不会使用这项功能，因此只需接受默认的端口号 2030，并单击 Next。

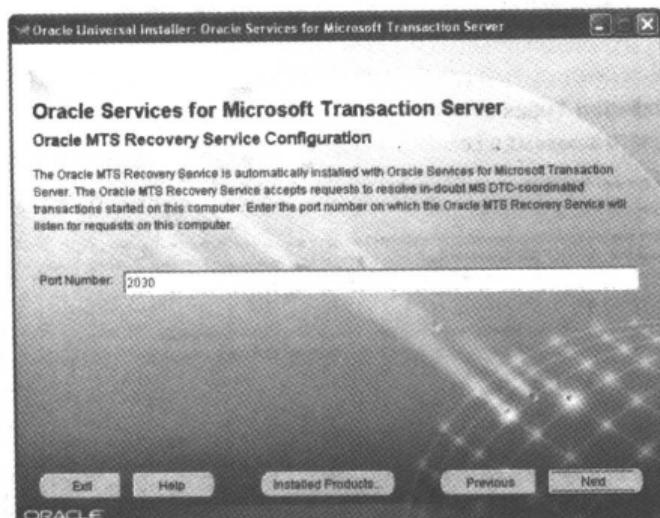


图 18.10 Oracle 的第七个安装窗口：指定 Oracle MTS Recovery Service 的端口

(7) 提供数据库系统标识(SID)。

在显示的第八个屏幕上(图 18.11)，我们必须惟一标识我们的数据库。Oracle 的配置和实用工具使用 SID 来标识进行操作的数据库。对于本书提供的 JDBC 示例，我们建议 Global Database Name 使用 csajspcoreservlets.com。输入这个名字可以自动生成 SID csajsp。单击 Next。

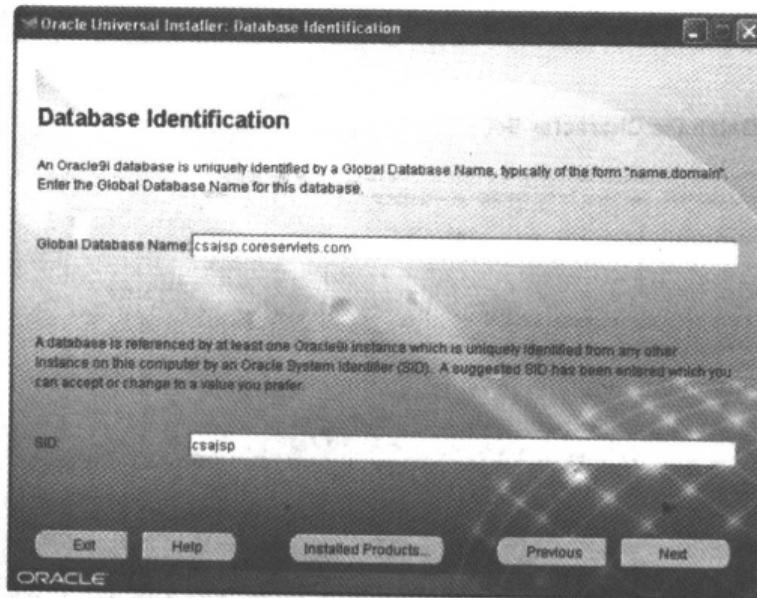


图 18.11 Oracle 的第八个安装窗口：指定全局数据库名称和 SID

(8) 指定数据库的位置。

在显示的第九个屏幕上(图 18.12)定义数据库的物理位置。在产品环境中，Oracle 推荐存储数据库的磁盘要不同于 Oracle9i 软件安装的磁盘。在开发环境中，您可能只有单个磁盘。我们使用推荐的默认位置，C:\oracle\oradata。单击 Next。

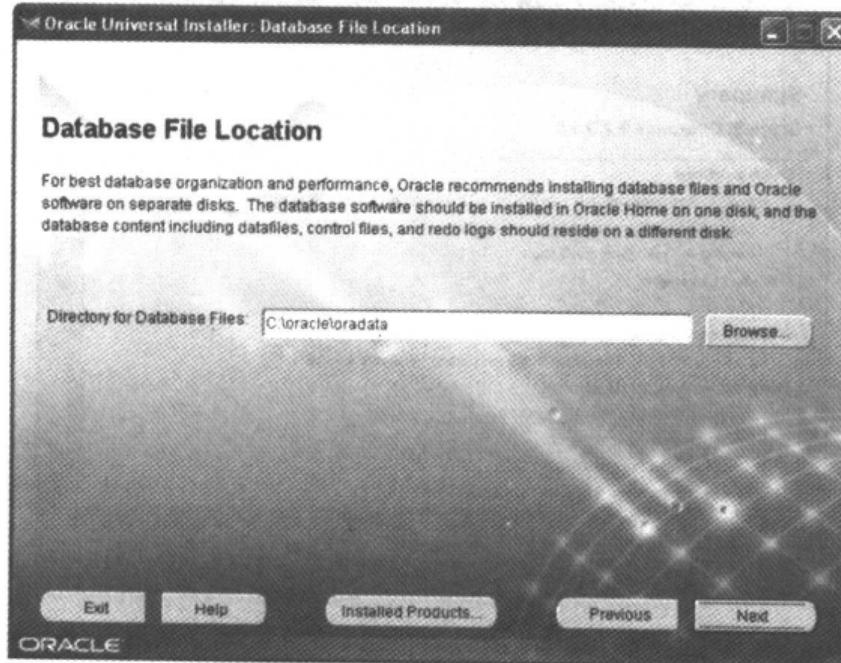


图 18.12 Oracle 的第九个安装窗口：指定数据库的物理位置

(9) 指定默认字符集。

在显示的第十个窗口中(图 18.13)，为数据库选择一种字符集。接受与操作系统的语言设置一致的默认字符集。

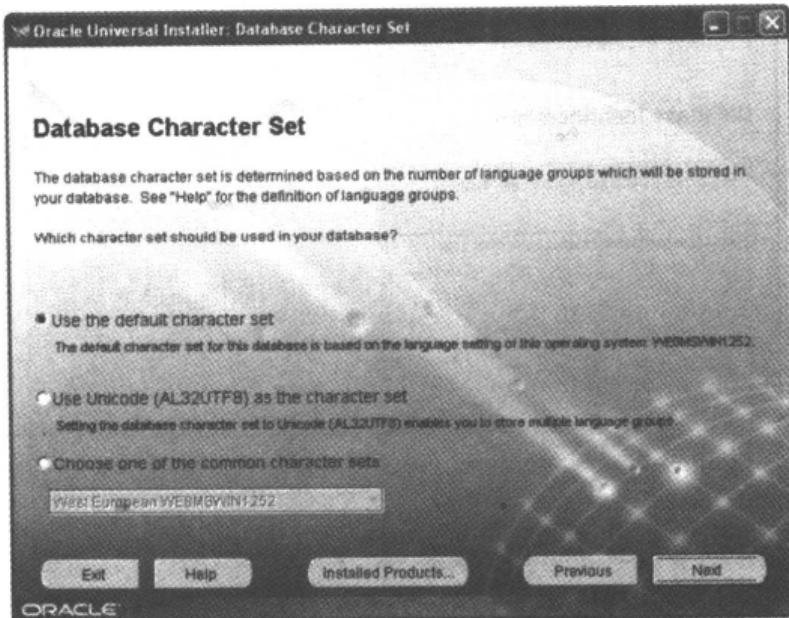


图 18.13 Oracle 的第十个窗口：选择数据库使用的默认字符集

(10) 检查产品列表。

显示的第 11 个屏幕汇总了哪些 Oracle 产品要安装到计算机上。在检查完这个列表之后，单击 Install 按钮。

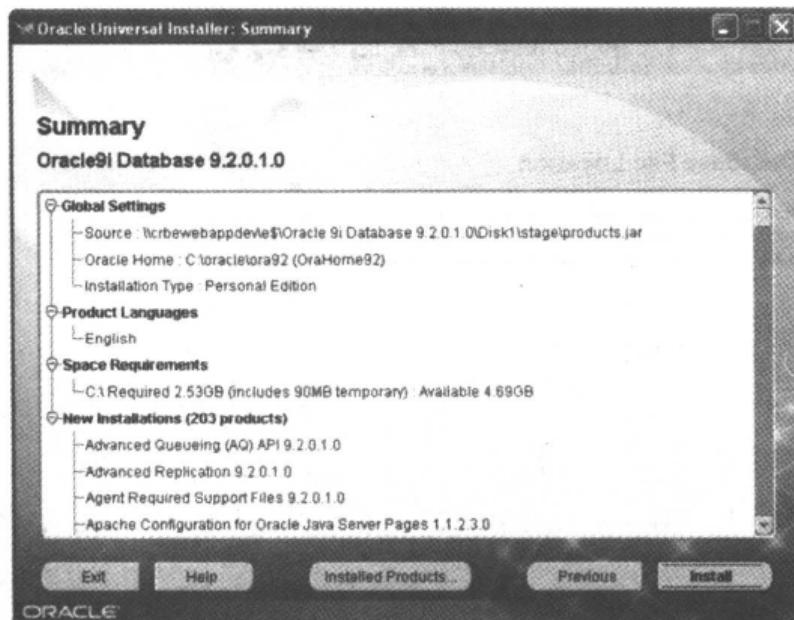


图 18.14 Oracle 的第十一个安装窗口：汇总要安装到计算机上的产品

(11) 安装 Oracle9i。

此时，Oracle Universal Installer 将要开始安装 Oracle9i。安装器(图 18.15)表示出安装的进程，并在每个组件安装时提供一段简短的消息。所有的安装活动都记录在位于 C:\Program Files\Oracle\Inventory\logs 的日志文件中。如果安装失败，可以检查这个日志文件，查出详细的信息。

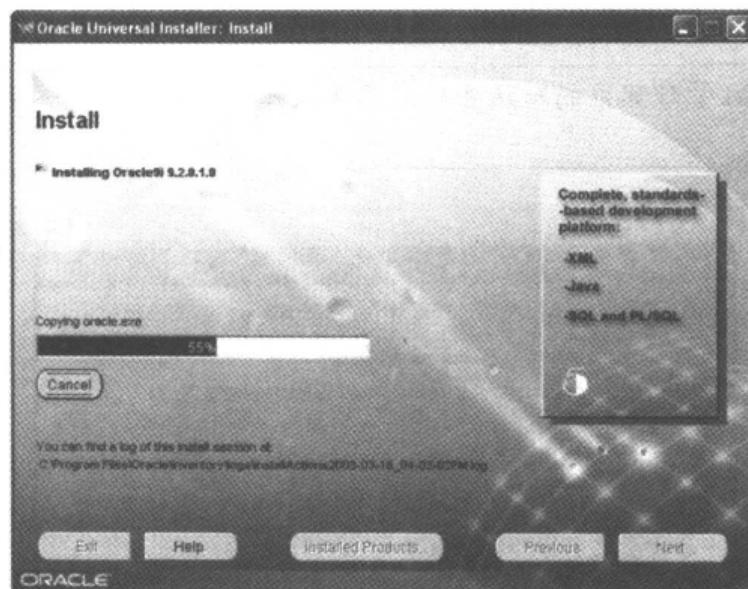


图 18.15 Oracle 的第十二个窗口：Oracle 组件安装期间

(12) 安装配置工具。

核心的 Oracle9i 软件安装完成后，还可以安装配置工具来管理数据库。我们推荐安装配置工具。单击 Next。工具安装的进程由 Oracle Universal Installer 标示出来，如图 18.16 所示。

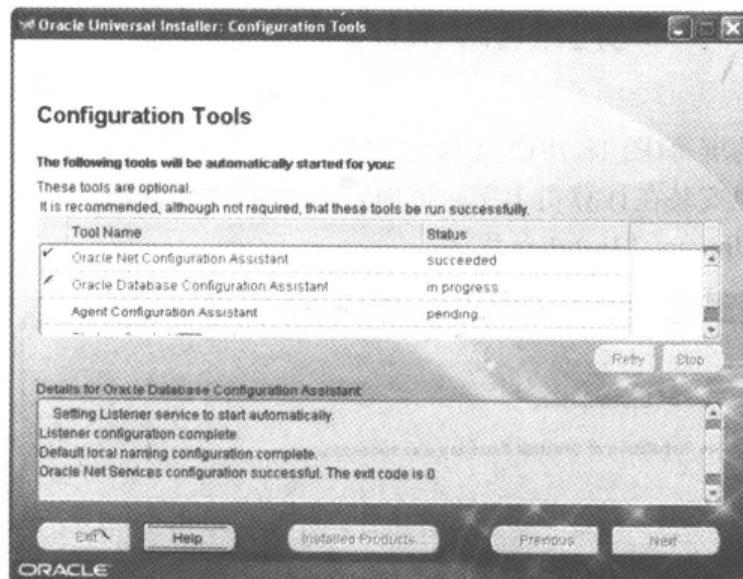


图 18.16 Oracle 的第十三个窗口：配置工具安装期间

(13) 指定密码。

配置工具安装完成后，Database Configuration Assistant 会提示输入新的 SYS 和 SYSTEM 密码，来管理数据库(图 18.17)。许多 Oracle 数据库产品使用的默认密码是：SYS 为 change_on_install，SYSTEM 为 manager。不要使用这些广为人知的密码。指定新的密码后，单击 OK。

警告

SYS 和 SYSTEM 管理账户的默认密码是广为人知的，为了安全地管理您的数据库，切记指定不同的密码。

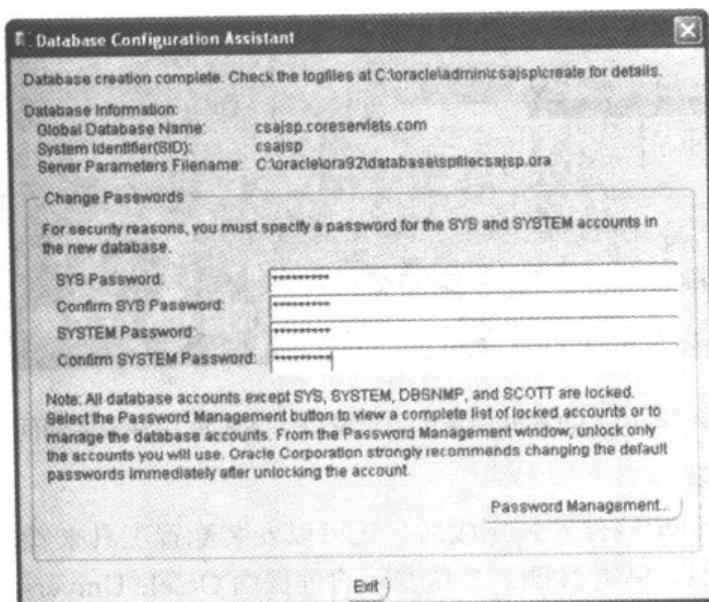


图 18.17 Oracle 的第十四个安装窗口：指定密码。使用 Database Configuration Assistant 为 SYS 和 SYSTEM 管理账户指定密码

(14) 完成安装。

最后显示的屏幕(图 18.18)是安装进程的结束。此时，Oracle9i Database Release 2 已经成功地安装在计算机上了，同时还创建了名为 csajsp 的数据库。单击 Exit 结束 Oracle Universal Installer 程序。

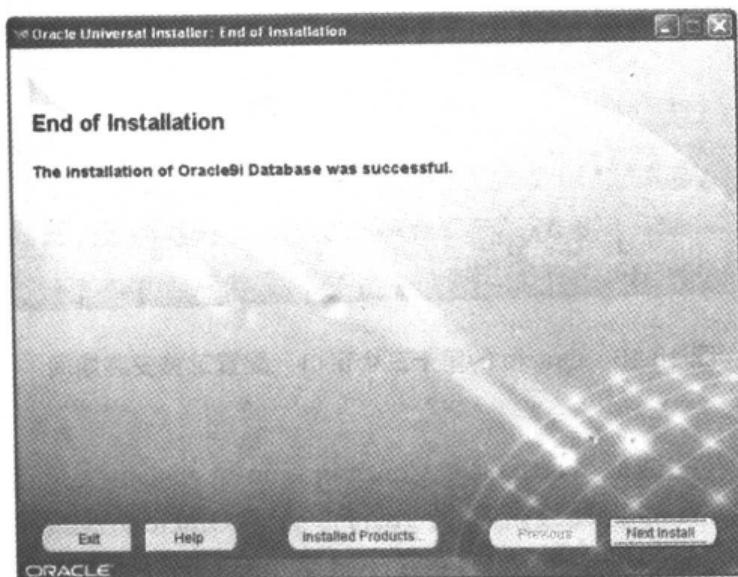


图 18.18 Oracle 的第十五个安装窗口：完成 Oracle9i 的安装

18.3.2 创建数据库

一般地，在 Oracle9i 的安装过程中就会创建起始数据库。但是，如果您使用的计算机已经安装了 Oracle9i，您可能会希望创建新的数据库。创建新的数据库有两种选择。第一种选择是使用 Oracle Database Configuration Assistant——图形化配置工具。第二种选择是手动地创建数据库。为了更好地理解 Oracle9i，我们提供创建新数据库的这两种方式。和安装 Oracle9i 相同，创建新的数据库必须拥有 Windows 的本地管理权限。

警告

在 Windows NT/2000/XP 上创建新的 Oracle9i 数据库时，您必须拥有计算机的本地管理员权限。

18.3.3 用 Configuration Assistant 创建数据库

创建新数据库的过程很复杂，因此 Oracle 强烈推荐使用 Database Configuration Assistant(DBCA)。下面是用 DBCA 创建数据库的步骤。

(1) 启动 Oracle Database Configuration Assistant。

Oracle9i 数据库的安装中包括 DBCA。在 Windows XP 上启动 DBCA 时，从 Start (【开始】)菜单开始，再选择 Programs(【程序】)，然后选 Oracle - OraHome92，然后选 Configuration and Migration Tools，最后是 Database Configuration Assistant。DBCA 启动时，会显示一个欢迎屏幕，如图 18.19 所示。单击 Next。

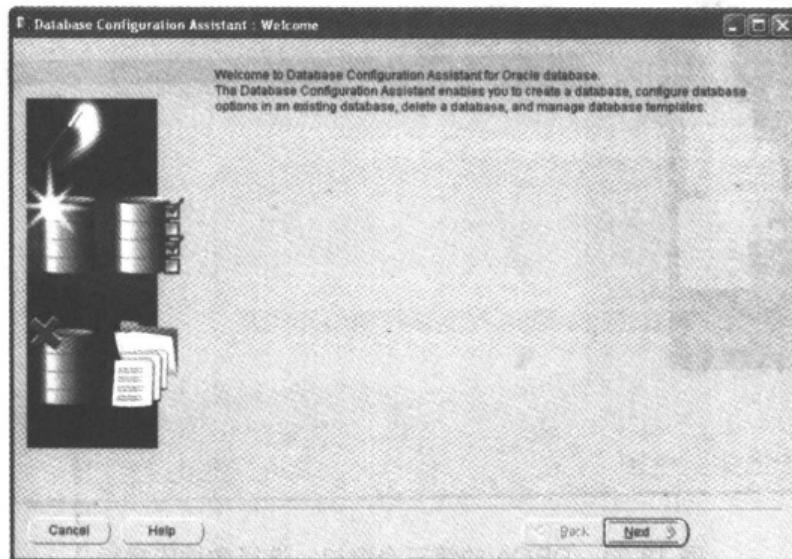


图 18.19 DBCA 的第一个窗口：欢迎信息

(2) 选择一项操作。

在显示的第二个屏幕上(图 18.20)，选择一项需要执行的操作。选择第一个选项：Create a database(创建数据库)。单击 Next。

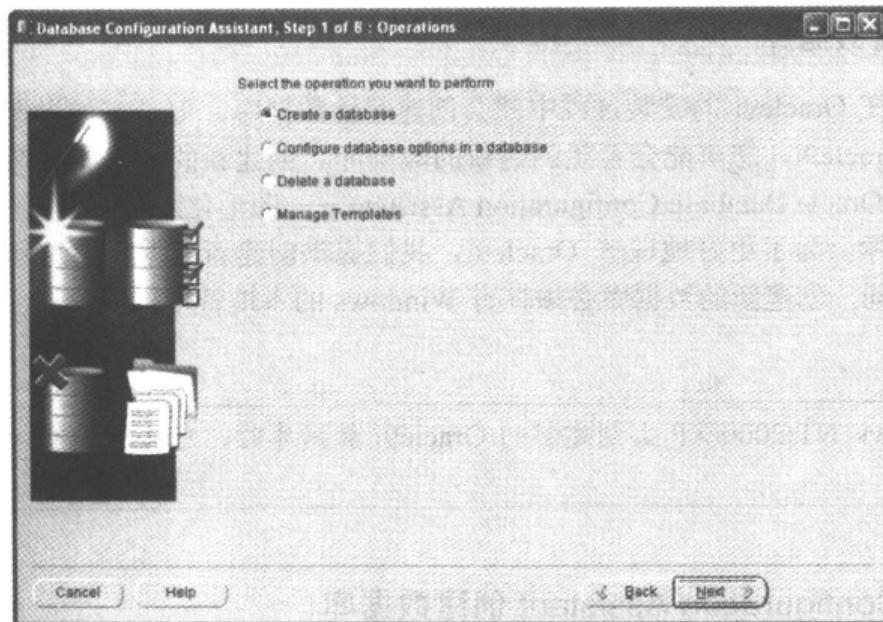


图 18.20 DBCA 的第二个窗口：选择执行一项操作

(3) 选择数据库模板。

在显示的第三个屏幕上(图 18.21)，选取用来创建数据库的模板。为新数据库选择模板之后，单击 Next。

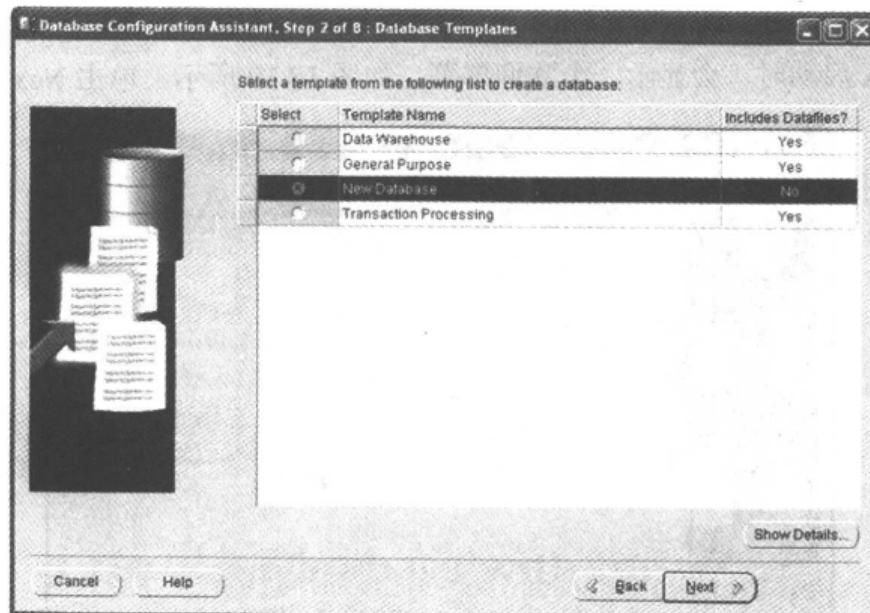


图 18.21 DBCA 的第三个窗口：选择数据库模板

(4) 提供数据库标识。

在显示的第四个屏幕上(图 18.22)，指定 Global Database Name 和 SID 标识新的数据库。Oracle 配置和实用工具使用 SID 来标识进行操作的数据库。对于本书中提供的 JDBC 示例，我们建议将 csajspcoreservlets.com 作为 Global Database Name。输入这个选择会自动生成 SID csajsp。单击 Next。

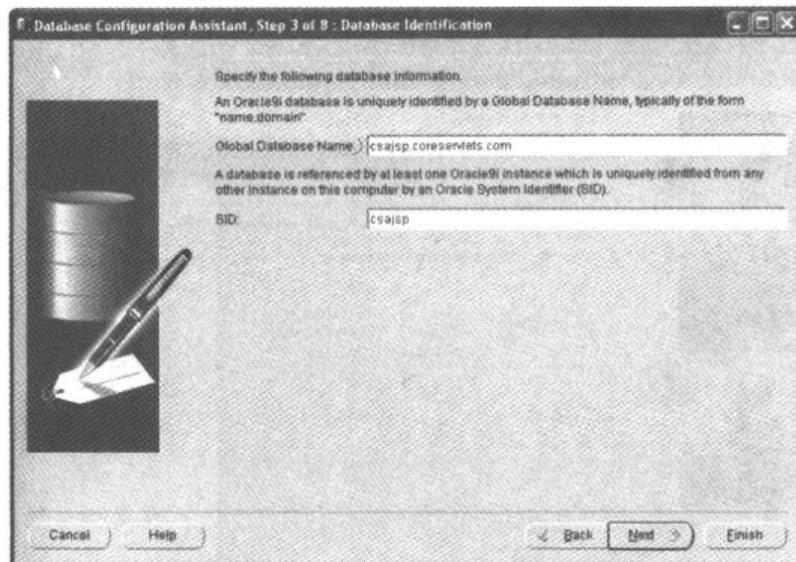


图 18.22 DBCA 的第四个窗口：指定全局数据库名称和 SID

(5) 选择需要安装的数据库特性。

在显示的第五个屏幕上(图 18.23)选择希望配置在数据库中的特性。如果为测试创建简单的数据库，不需要可选的特性，则不要选定任何特性。如果提示您是否删除关联的表空间(tablespace)，回答 Yes。同时，选择 Standard database features(标准数据库特性)按钮，不选定这 4 个选项。单击 Next。

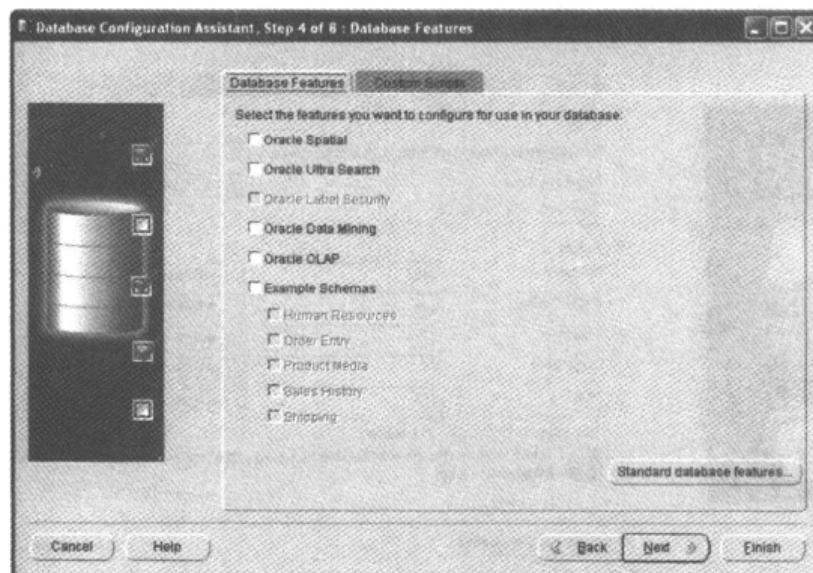


图 18.23 DBCA 的第五个窗口：选择要安装的数据库特性

(6) 选择数据库的连接选项。

在显示的第六个屏幕上(图 18.24)选择期望的数据库操作模式。选择 Dedicated Server Mode 选项。单击 Next。

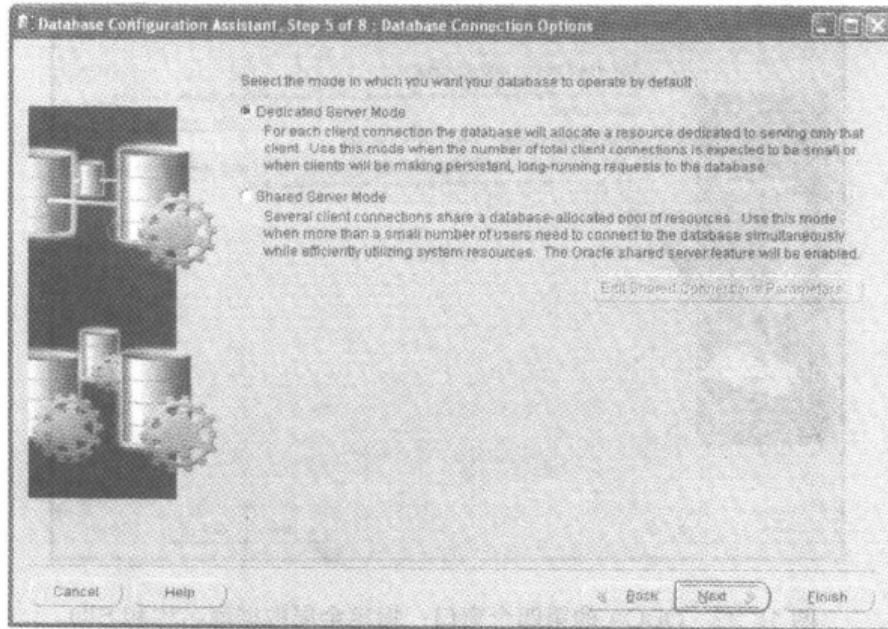


图 18.24 DBCA 的第六个窗口：选择数据库的操作模式

(7) 指定初始化参数。

在显示的第七个屏幕上(图 18.25)，可以对数据库进行定制。默认的参数就足够了，因此不需要改变标签中的任何设置。单击 Next。

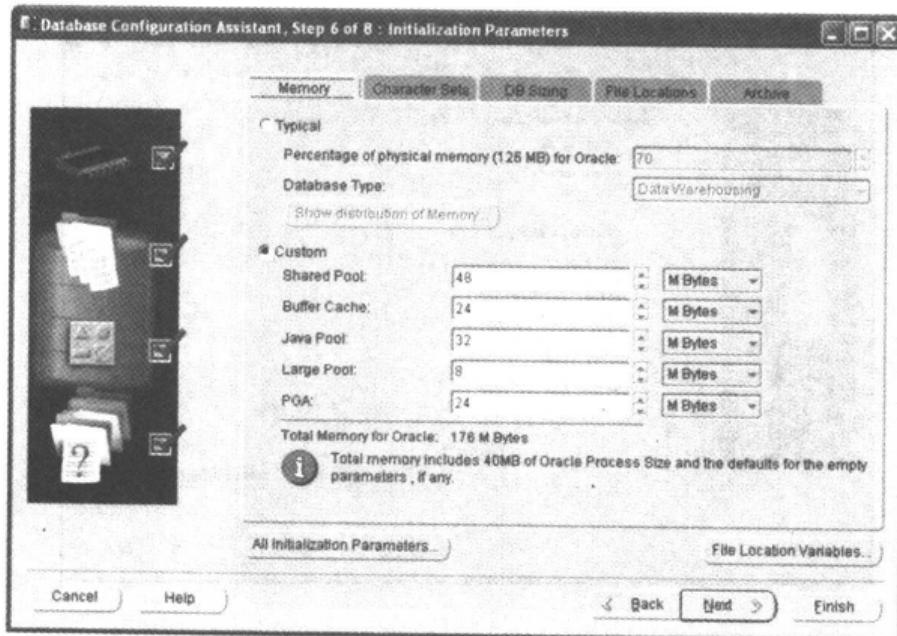


图 18.25 DBCA 的第七个窗口：指定数据库的初始化参数

(8) 指定存储参数。

在显示的第八个屏幕中(图 18.26)指定数据库创建时所需的参数。默认的存储文件和位置就足够了，不需要修改。单击 Next。

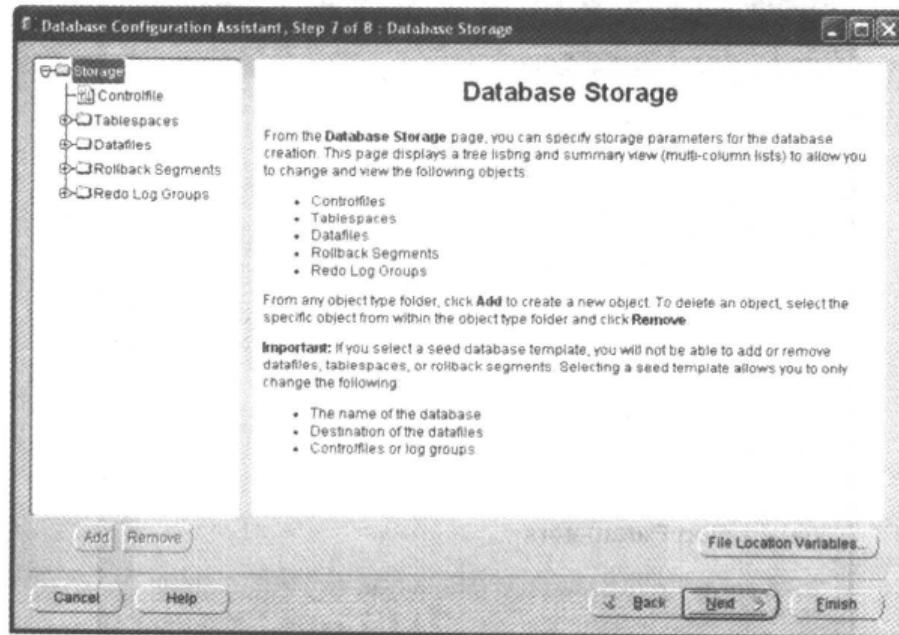


图 18.26 DBCA 的第八个窗口：指定数据库的存储参数

(9) 选择数据库的创建选项。

在显示的第九个屏幕中(图 18.27)指定创建数据库的选项。在此，由于只希望创建新的数据库，因此选择 Create Database 选项。单击 Next。

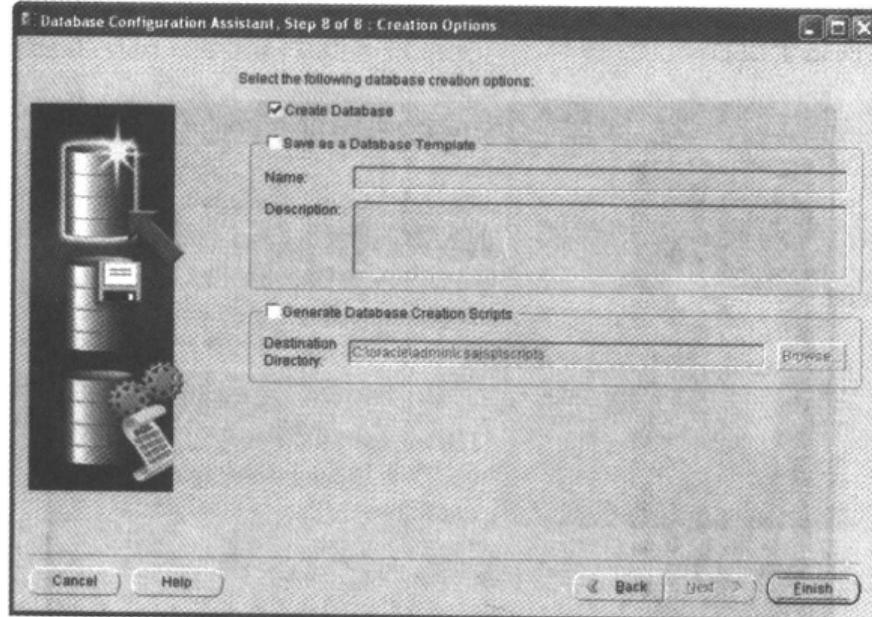


图 18.27 DBCA 的第九个窗口：选择创建数据库的选项

(10) 检查数据库的配置。

此时，DBCA 提供为创建数据库所选取的所有选项，如图 18.28 所示。检查完这些选项后，单击 OK。

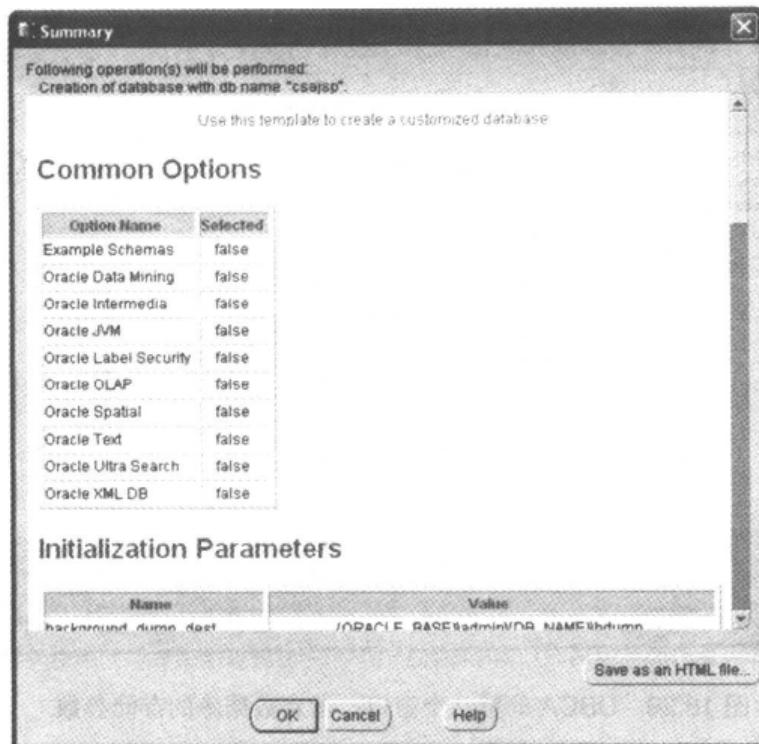


图 18.28 DBCA 的第十个窗口：在数据库创建之前汇总配置选项

(11) 监视数据库的创建过程。

显示的第十一个屏幕(图 18.29)标示出数据库创建过程中的各种活动。如果愿意，可以监视这个过程。

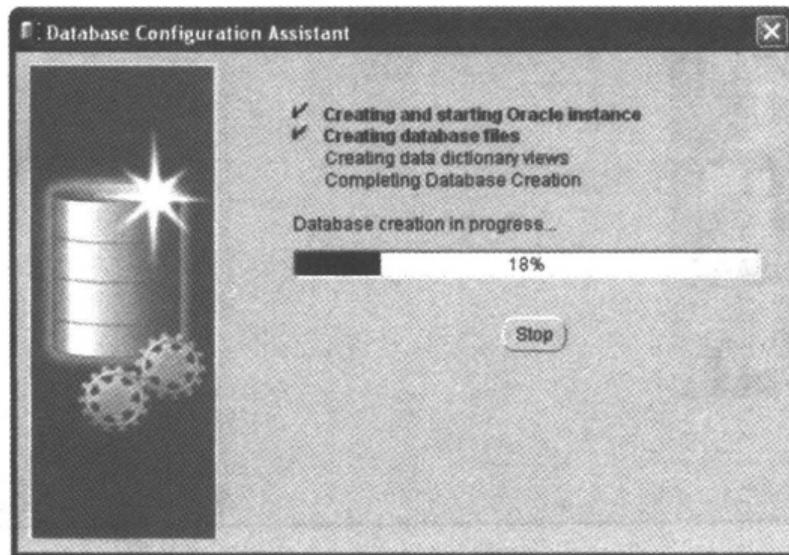


图 18.29 DBCA 的第十一个窗口：数据库的创建期间

(12) 指定密码。

数据库安装完成之后，Database Configuration Assistant 会提示您输入 SYS 和 SYSTEM 的密码，来管理数据库(图 18.30)。在指定新的密码之后，单击 OK 完成数据库的创建过程。

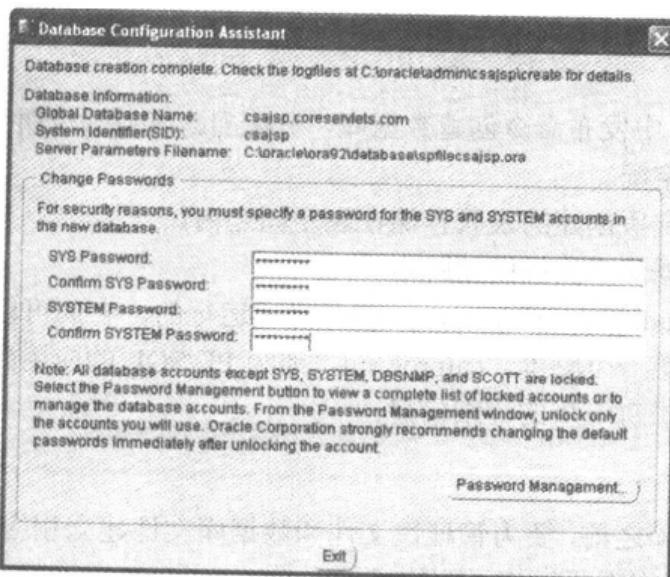


图 18.30 DBCA 的第十二个窗口：指定 SYS 和 SYSTEM 的管理密码

18.3.4 手动地创建数据库

我们一般使用 Database Configuration Assistant(在前一小节中做了介绍)创建新的数据库。但是，如果要更完全地控制整个过程，有时会手动地完成数据库的创建。这一节介绍手动过程。下面的列表简要地概括了手动创建 Oracle9i 数据库的步骤。详细的指示在列表之后给出。

(1) 建立数据库的目录。

在创建新的数据库之前，必须首先为管理性文件和数据库文件创建所需的目录。

(2) 创建初始化参数文件。

初始化参数文件是配置和启动数据库必需的文件。除了其他信息之外，参数文件还包含块大小和允许的进程数等信息。

(3) 创建密码文件。

含有用户身份验证信息的密码文件是数据库管理所必需的。管理员可以通过密码文件或 OS 的系统用户组进行身份验证。对于这项配置，我们使用密码文件。

(4) 为数据库创建服务。

在 Windows NT/2000/XP 上，数据库作为服务运行。采用这种方式才可以在管理员从系统中注销后，数据库不致被关闭。

(5) 声明 ORACLE_SID 值。

ORACLE_SID 是一个环境变量，声明运行 Oracle 工具(比如 SQL*Plus)时使用哪个数据库。

(6) 作为 SYSDBA 连接到 Oracle 服务。

要从 SQL*Plus 中管理和创建数据库，必须作为数据库系统的管理员(SYSDBA)连接到 Oracle 服务。

(7) 启动数据库实例。

启动实例以完成内存和进程的初始化，这样才能创建和管理数据库。在数据库实

例未启动的情况下，不能创建数据库。

(8) **创建数据库。**

在 SQL*Plus 中发布命令创建数据库，分配日志和临时文件。

(9) **创建用户表空间。**

数据库的用户所创建的表就存储在表空间之内。

(10) **运行构建数据库字典视图的脚本。**

两个脚本(catalog.sql 和 catproc.sql)是必须运行的，它们可以在数据库中建立视图和替代名。第一个脚本，catproc.sql，还为 PL/SQL 的应用配置数据库。

接下来，我们提供每一步的详细信息。

1. 建立数据库的目录

在创建新的数据库之前，要为管理性文件和数据库文件建立相应的目录。假定 Oracle9i 安装在 C:\ 驱动器，我们需要创建下面的目录：

```
C:\oracle\admin\csajsp  
C:\oracle\admin\csajsp\bdump  
C:\oracle\admin\csajsp\cdump  
C:\oracle\admin\csajsp\pfile  
C:\oracle\admin\csajsp\udump  
C:\oracle\oradata\csajsp
```

bdump 目录保存后台进程的报警和跟踪文件。cdump 目录存储 Oracle 服务器发生崩溃且不可恢复时的核心转储文件。udump 目录保存用户进程的跟踪文件。oradata\csajsp 目录含有物理数据库。

2. 创建初始化参数文件

启动数据库时，Oracle 必须读取一个初始化参数文件。这个文件中的参数会初始化 Oracle 实例的诸多内存和进程设置。初始化参数文件的标准命名约定是 initSID.ora，其中 SID 是数据库的系统标识符。

实例是内存以及与数据库相关联的后台进程的组合。实例的一个重要组成部分是 System Global Area(SGA，系统全局区)，它在实例启动时分配。SGA 是一段内存区域，存储和处理取自物理数据库的数据。

重点提示

Oracle 实例由内存结构和管理数据库的后台进程组成。初始化参数文件是启动实例所必需的。

实际上，在需要创建新的数据库时，大多数管理员都简单地复制和修改现有的参数文件。清单 18.1 提供一个具体的初始化参数文件，它在 Windows XP 上创建名为 csajsp 的数据库。这个文件需要放在 C:\oracle\admin\csajsp\pfile 目录。

有关初始化参数的更多信息，参见 http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96536/toc.htm 处 Oracle9i 数据库的在线参考。初始化参数文件中至少需要指定 background_dump_dest，compatible，control_files，db_block_buffers，db_name，

shared_pool_size 和 user_dump_dest。

清单18.1 initCSAJSP.ora(数据库初始化参数文件)

```
# Initialization parameter file for Oracle9i database
# on Windows XP.

# Database Identification
db_domain=coreservlets.com
db_name=csajsp

# Instance Identification
instance_name=csajsp

# Cache and I/O
db_block_size=8192
db_cache_size=25165824
db_file_multiblock_read_count=16

# Cursors and Library Cache
open_cursors=300

# Diagnostics and Statistics
background_dump_dest=C:\oracle\admin\csajsp\bdump
core_dump_dest=C:\oracle\admin\csajsp\cdump
timed_statistics=TRUE
user_dump_dest=C:\oracle\admin\csajsp\udump

# File Configuration
control_files=("C:\oracle\oradata\csajsp\CONTROL01.CTL",
               "C:\oracle\oradata\csajsp\CONTROL02.CTL",
               "C:\oracle\oradata\csajsp\CONTROL03.CTL")

# Job Queues
job_queue_processes=10

# MTS
dispatchers="(PROTOCOL=TCP) (SERVICE=csajspXDB)"

# Miscellaneous
aq_tm_processes=1
compatible=9.2.0.0.0

# Optimizer
hash_join_enabled=TRUE
query_rewrite_enabled=FALSE
star_transformation_enabled=FALSE

# Pools
java_pool_size=33554432
large_pool_size=8388608
shared_pool_size=50331648

# Processes and Sessions
processes=150

# Redo Log and Recovery
fast_start_mttr_target=300
```

```

# Security and Auditing
remote_login_passwordfile=EXCLUSIVE

# Sort, Hash Joins, Bitmap Indexes
pga_aggregate_target=25165824
sort_area_size=524288

# System Managed Undo and Rollback Segments
undo_management=AUTO
undo_retention=10800
undo_tablespace=undotbs

```

3. 创建密码文件

如果将初始化参数 REMOTE_LOGIN_PASSWORDFILE 设为 EXCLUSIVE，则必须创建密码文件以验证拥有 SYSDBA 特权的管理员。如果以 SYSDBA 特权连接到 Oracle 服务，那么，管理员可以没有任何限制地在数据库上执行任何操作。使用密码文件验证管理员可以提供最高级别的安全性。

ORAPWD 命令行工具可以用来创建密码文件。这个命令接受 3 个参数：FILE——指定密码文件的位置和文件名；PASSWORD——指定赋予给 SYS 用户(管理数据库)的密码；以及 ENTRIES——指定希望授予数据库管理 SYSDBA 特权的最大用户数(用户 SYS 已拥有 SYSDBA 特权)。

例如，下面的命令

```
Prompt> ORAPWD FILE="C:\oracle\ora92\DATABASE\PWDcsajsp.ora"
          PASSWORD=csajspDBA ENTRIES=5
```

创建密码文件 PWDcsajsp.ora，以 csajspDBA 作为管理数据库的 SYS 用户的密码。ENTRIES 值为 5 说明密码文件中总共有 5 个用户拥有 SYSDBA 特权。

按照约定，Oracle9i 中，密码文件放在 C:\oracle\ora92\DATABASE 目录中，密码文件的名字为 PWD`database`.ora，其中 `database` 是与密码文件相关联的数据库的名字(SID)。

4. 为数据库创建服务

在 Windows NT/2000/XP 上创建数据库之前，需要创建一个 Oracle 服务来运行这个数据库。创建 Oracle 服务可以避免管理员从计算机上注销时数据库进程会被终止。完成这一步骤使用 oradim 命令行工具。

假定数据库的 SID 为 csajsp，初始化参数文件是 initCSAJSP.ora，位于 C:\oracle\admin\csajsp\pfile 中，我们使用下面的命令创建 Oracle 服务。

```
Promtp> oradim -NEW -SID CSAJSP -STARTMODE MANUAL
          -PFILE "C:\oracle\admin\csajsp\pfile\initCSAJSP.ora"
```

这个命令创建服务 OracleServiceCSAJSP，配置为手工启动。但是，在首次创建时，服务应该启动。要检查服务是否已经启动，发布下面的命令。

```
Promtp> net start OracleServiceCSAJSP
```

如果希望在计算机重启后启动数据库服务，则要将服务的启动类型改为自动。如果要在 Windows XP 上更改启动类型，请到 Start(【开始】)菜单，然后选择 Control Panel(【控

制面板】), 再选择 Performance and Maintenance(【性能和维护】), 然后选择 Administrative Tools(【管理工具】), 再选择 Services(【服务】), 然后右击需要更改的服务, 选择 Properties(【属性】)。接下来, 在下拉列表框中更改启动的类型。

5. 声明 ORACLE_SID 值

ORACLE_SID 是一个环境变量, 各种 Oracle 工具都使用它来识别应该在哪个数据库上操作。如果要将 ORACLE_SID 设为 csajsp 数据库, 则需输入下面的命令:

```
Prompt> set ORACLE_SID=csajsp
```

要注意, 在等号(=)字符的前后没有空格。

6. 作为 SYSDBA 连接到 Oracle 服务

下一步是在创建新的数据库之前, 使用 SQL*Plus 以系统 DBA(SYSDBA)的身份连接到数据库服务。首先, 以 nolog 选项启动 SQL*Plus, 如下所示。

```
Prompt> SQLPLUS /nolog
```

然后, 以 SYSDBA 连接到 Oracle 服务, 使用的命令如下:

```
SQL> CONNECT SYS/password AS SYSDBA
```

其中 *password* 是之前创建密码文件时指定的 SYS 密码。注意: 设置 ORACLE_SID 环境变量之后, SQL*Plus 就能够自动确定应该连接到哪个数据库服务(本例中为 OracleServiceCSAJSP)。

7. 启动数据库实例

创建新的数据库时, Oracle 实例必须启动。如果只想启动实例, 但不安装数据库, 则在 SQL*Plus 中发布下面的命令。

```
SQL> STARTUP NOMOUNT  
PFILE="C:\oracle\admin\csajsp\pfile\initCSAJSP.ora"
```

PFILE 必须指出包含数据库初始化参数的文件。以 NOMOUNT 启动实例会创建 SGA 并启动后台进程; 但是, 数据库依旧不能访问。一般地, 只在创建和维护数据库时才会以 NOMOUNT 模式启动数据库。

8. 创建数据库

如果要创建新数据库, 可以在 SQL*Plus 中发布 CREATE DATABASE SQL 命令。清单 18.2 给出一个在 Windows NT/2000/XP 平台上创建 csajsp 数据库的 CREATE DATABASE 命令。要创建数据库, 只需在 SQL*Plus 中输入(剪切、粘贴)这个命令。或者可以在 SQL*Plus 中运行 create_csajsp.sql 来创建数据库。执行这个脚本使用下面的命令。

```
SQL> @create_csajsp.sql
```

要注意, 为了使 SQL*Plus 能够找到这个脚本, 有可能需要在@后指定完整的路径。

执行这个命令(或脚本)会在 C:\oracle\oradata\csajsp 目录中创建 csajsp 数据库, 并自动创建两个用户账户 SYS 和 SYSTEM, 来管理数据库。SYS 是数据库字典的所有者(数据库的结构和用户信息), SYSTEM 是 Oracle 工具所使用的附加表和视图的所有者。

如果数据库创建失败, 请检查报警日志文件, C:\oracle\admin\csajsp\bdump\alert_csajsp.log, 找出发生了什么错误。改正错误, 删除 C:\oracle\oradata\csajsp 目录中的所有文件, 重新执行这个命令。

清单18.2 create_csajsp.sql

```
/* SQL command to create an Oracle9i database named csajsp. */

CREATE DATABASE csajsp
  USER SYS IDENTIFIED BY csajspDBA
  USER SYSTEM IDENTIFIED BY csajspMAN
  LOGFILE
    GROUP 1 ('C:\oracle\oradata\csajsp\redo01.log') SIZE 100M,
    GROUP 2 ('C:\oracle\oradata\csajsp\redo02.log') SIZE 100M,
    GROUP 3 ('C:\oracle\oradata\csajsp\redo03.log') SIZE 100M
  MAXLOGFILES 5
  MAXDATAFILES 100
  MAXINSTANCES 1
  CHARACTER SET WE8MSWIN1252
  NATIONAL CHARACTER SET AL16UTF16
  DATAFILE 'C:\oracle\oradata\csajsp\system01.dbf'
    SIZE 325M REUSE
    AUTOEXTEND ON NEXT 10240K MAXSIZE UNLIMITED
    EXTENT MANAGEMENT LOCAL
  DEFAULT TEMPORARY TABLESPACE temp
    TEMPFILE 'C:\oracle\oradata\csajsp\temptbs01.dbf'
    SIZE 20M REUSE
    EXTENT MANAGEMENT LOCAL
  UNDO TABLESPACE undotbs
    DATAFILE 'C:\oracle\oradata\csajsp\undotbs01.dbf'
    SIZE 200M REUSE
    AUTOEXTEND ON NEXT 5120K MAXSIZE UNLIMITED;
```

9. 创建用户表空间

在将信息存入数据库之前, 必须为它们创建表空间, 否则不能存入数据。用户创建的所有表都放在表空间内。如果要创建用户表空间, 可以在 SQL*Plus 中输入下面的命令。

```
SQL> CREATE TABLESPACE users
  DATAFILE 'C:\oracle\oradata\csajsp\users01.dbf'
  SIZE 15M REUSE
  AUTOEXTEND ON NEXT 1280K MAXSIZE UNLIMITED
  EXTENT MANAGEMENT LOCAL;
```

这个命令创建初始大小为 15M 的表空间 users。数据在物理上存储在文件 users01.dbf 中。

10. 运行构建数据库字典视图的脚本

创建数据库的最后一步是运行 SQL*Plus 的 catalog.sql 和 catproc.sql 脚本。在 @ 后输入脚本的完整路径。

```
SQL> @C:\oracle\rdbms\admin\catalog.sql
SQL> @C:\oracle\rdbms\admin\catproc.sql
```

catalog.sql 脚本为数据字典表创建视图和替代名。catproc.sql 脚本完成使用 Oracle

PL/SQL 所需的一些设置和安装任务。这些脚本都生成大量的输出，可以忽略这些信息；针对在创建之前首先需要删除的表和视图，可能会产生一些错误消息。

18.3.5 创建用户

如果要从 Web 应用程序中访问数据库，首先需要创建新的用户，并赋予恰当的权限。在 SQL*Plus 中，输入下面的 CREATE USER 命令。

```
SQL> CREATE USER username IDENTIFIED BY password
      DEFAULT TABLESPACE users
      QUOTA UNLIMITED ON users
      TEMPORARY TABLESPACE temp;
```

其中 *username* 是新用户的登录名，*password* 是新用户的密码。默认表空间是这个用户创建的表存储的地方，QUOTA 子句授予用户对存储在 *users* 表空间中的信息的无限制使用。如果没有指定默认表空间的 QUOTA 子句，则用户不能创建任何表。同时还为用户赋予一个临时表空间，供相关的 SQL 查询用来对数据进行排序。

接下来，我们需要授予新用户连接到数据库服务和创建新表的权限。发布下面的 SQL*Plus 命令。

```
SQL> GRANT CREATE SESSION, CREATE TABLE
      TO username;
```

其中 *username* 是需要访问数据库的用户。授予 CREATE TABLE 特权还使得用户能够删除表。

18.3.6 安装 JDBC 驱动程序

在我们的 JDBC 示例中，我们使用 Oracle Thin JDBC 驱动程序，它可以建立到 Oracle 数据库服务器的直接 TCP 连接。Oracle JDBC 驱动程序可以从 http://otn.oracle.com/software/tech/java/sqlj_jdbc/ 下载。下载适当的版本，classes12.zip(如果使用 JDK 1.2 和 JDK 1.3)或 ojdbc14.jar(如果使用 JDK 1.4)，并将它放在 CLASSPATH 中供开发使用；在部署时放在 Web 应用的 WEB-INF/lib 目录。

如果 Web 服务器上多个应用都要访问 Oracle 数据库，Web 管理员可能会选择将这个 JAR 文件移到服务器的公共目录中。例如，在 Tomcat 中，多个应用使用的 JAR 文件可以放在 *install_dir/common/lib* 目录中。

如果您的 Web 应用服务器不能识别 WEB-INF/lib 目录中的 ZIP 文件，可以将文件的扩展名改为.jar；ZIP 和 JAR 压缩算法是兼容的(JAR 文件只是包括档案的元信息清单)。但是，某些开发人员会选择解压缩该文件，然后使用 jar 工具的 -0 命令选项创建非压缩的 JAR 文件。CLASSPATH 中支持压缩和非压缩 JAR 文件，但载入非压缩 JAR 文件中的类要更快一些。Java 档案工具的平台相关文档参见 <http://java.sun.com/j2se/1.4.1/docs/tooldocs/tools.html>。

最后还要注意，如果数据库传输中的安全同样重要，则要参考 http://download-west.oracle.com/docs/cd/B10501_01/java.920/a96654/advanc.htm，了解如何加密 JDBC 连接上的数据流。要加密从 Web 服务器到客户浏览器之间的数据流，可以使用 SSL(详细信息参见本书第二卷有关 Web 应用安全的章节)。

18.4 通过 JDBC 连接来测试数据库

安装和配置完数据库之后，您可能希望测试数据库的 JDBC 连通性。在清单中，我们提供了一个程序，它执行下面的数据库测试。

- 建立与数据库之间的 JDBC 连接，并报告产品的名称和版本；
- 创建简单的“authors”表，它含有 Core Servlets and JavaServer Pages, Second Edition 两个作者的 ID、名和姓；
- 查询“authors”表，汇总每个作者的 ID、名和姓；
- 执行不严格的测试，以确定 JDBC 的版本。使用所报告的 JDBC 版本时要小心：所报告的 JDBC 版本并不意味着：该驱动程序经过验证并支持这个版本的 JDBC 所定义的所有类和方法。

由于 TestDatabase 在 coreservlets 包中，因此它必须位于 coreservlets 子目录内。在编译这个文件之前，要对 CLASSPATH 进行设置，使之包括含有 coreservlets 目录的目录(也就是 coreservlets 的上级目录)。详细信息参见 2.7 节。此时，如果要编译这个程序只需在 coreservlets 子目录中运行 javac TestDatabase.java(或在 IDE 中选择 build 或 compile)。但要运行 TestDatabase，则需要指出完整路径，如下所示：

```
Prompt> java coreservlets.TestDatabase host dbName  
username password vendor
```

其中 *host* 是数据库服务器的主机名，*dbName* 是希望测试的数据库的名称，*username* 和 *password* 是访问数据库的用户的用户名和密码，而 *vendor* 是关键字，标识提供商的驱动程序。

这个程序使用第 17 章(清单 17.5)中的 DriverUtilities 类载入提供商驱动程序的信息，并创建用于连接数据库的 URL。当前，DriverUtilities 支持 Microsoft Access, MySQL 和 Oracle 数据库。如果您使用不同的数据库提供商，可能需要修改 DriverUtilities，加入提供商的相关信息。详细信息参见 17.3 节。

下面展示出 TestDatabase 在 MySQL 数据库 csajsp 上运行所生成的输出，它使用 MySQL Connector/J 3.0 驱动程序。

```
Prompt>java coreservlets.TestDatabase localhost  
Csajsp brown larry MYSQL  
Testing database connection ...  
  
Driver: com.mysql.jdbc.Driver  
URL:jdbc:mysql://localhost:3306/csajsp  
Username:brown  
Password:larry  
Product name:MySQL  
Product version:4.0.12-max-nt  
Driver name:MySQL-AB JDBC Driver  
Driver Version:3.0.6-stable (Date:2003/02/17 17:01:34 $,  
$Revision:1.27.2.1  
3$ )  
  
Creating authors table ... successful
```

```
Querying authors table ...
+-----+-----+-----+
| id   | first_name | last_name |
+-----+-----+-----+
| 1    | Marty      | Hall       |
| 2    | Larry      | Brown      |
+-----+-----+-----+
Checking JDBC version ...

JDBC Version: 3.0
```

有趣的是，和MySQL 4.0.12一同使用的MySQL Connector/J 3.0驱动程序报告JDBC版本3.0。但是，MySQL不与ANSI SQL-92完全兼容，这个驱动器也不可能通过JDBC 3.0的认证。因此，一定要仔细地检查提供商有关JDBC版本的文档，并在发布产品之前进行彻底的测试。

警告

DatabaseMetaData所报告的JDBC版本是非正式的。驱动程序不一定保证支持所报告的级别。请检查提供商的文档。

清单18.3 TestDatabase.java

```
package coreservlets;

import java.sql.*;

/** Perform the following tests on a database:
 * <OL>
 * <LI>Create a JDBC connection to the database and report
 *      the product name and version.
 * <LI>Create a simple "authors" table containing the
 *      ID, first name, and last name for the two authors
 *      of Core Servlets and JavaServer Pages, 2nd Edition.
 * <LI>Query the "authors" table for all rows.
 * <LI>Determine the JDBC version. Use with caution:
 *      the reported JDBC version does not mean that the
 *      driver has been certified.
 * </OL>
 */

public class TestDatabase {
    private String driver;
    private String url;
    private String username;
    private String password;

    public TestDatabase(String driver, String url,
                       String username, String password) {
        this.driver = driver;
        this.url = url;
        this.username = username;
        this.password = password;
    }
```

```
/** Test the JDBC connection to the database and report the
 * product name and product version.
 */
public void testConnection() {
    System.out.println();
    System.out.println("Testing database connection ...\\n");
    Connection connection = getConnection();
    if (connection == null) {
        System.out.println("Test failed.");
        return;
    }
    try {
        DatabaseMetaData dbMetaData = connection.getMetaData();
        String productName =
            dbMetaData.getDatabaseProductName();
        String productVersion =
            dbMetaData.getDatabaseProductVersion();
        String driverName = dbMetaData.getDriverName();
        String driverVersion = dbMetaData.getDriverVersion();
        System.out.println("Driver: " + driver);
        System.out.println("URL: " + url);
        System.out.println("Username: " + username);
        System.out.println("Password: " + password);
        System.out.println("Product name: " + productName);
        System.out.println("Product version: " + productVersion);
        System.out.println("Driver Name: " + driverName);
        System.out.println("Driver Version: " + driverVersion);
    } catch(SQLException sqle) {
        System.err.println("Error connecting: " + sqle);
    } finally {
        closeConnection(connection);
    }
    System.out.println();
}

/** Create a simple table (authors) containing the ID,
 * first_name, and last_name for the two authors of
 * Core Servlets and JavaServer Pages, 2nd Edition.
 */
public void createTable() {
    System.out.print("Creating authors table ... ");
    Connection connection = getConnection();
    if (connection == null) {
        System.out.println("failure");
        return;
    }
    try {
        String format =
            "(id INTEGER, first_name VARCHAR(12), " +
            "last_name VARCHAR(12))";
        String[] rows = { "(1, 'Marty', 'Hall')",
                         "(2, 'Larry', 'Brown')" };
        Statement statement = connection.createStatement();
        // Drop previous table if it exists, but don't get
        // error if not. Thus, the separate try/catch here.
    }
}
```

```
try {
    statement.execute("DROP TABLE authors");
} catch(SQLException sqle) {}
String createCommand =
    "CREATE TABLE authors " + format;
statement.execute(createCommand);
String insertPrefix =
    "INSERT INTO authors VALUES";
for(int i=0; i<rows.length; i++) {
    statement.execute(insertPrefix + rows[i]);
}
System.out.println("successful");
} catch(SQLException sqle) {
    System.out.println("failure");
    System.err.println("Error creating table: " + sqle);
} finally {
    closeConnection(connection);
}
System.out.println();
}

/** Query all rows in the "authors" table. */

public void executeQuery() {
    System.out.println("Querying authors table ... ");
    Connection connection = getConnection();
    if (connection == null) {
        System.out.println("Query failed.");
        return;
    }
    try {
        Statement statement = connection.createStatement();
        String query = "SELECT * FROM authors";
        ResultSet resultSet = statement.executeQuery(query);
        ResultSetMetaData resultSetMetaData =
            resultSet.getMetaData();
        int columnCount = resultSetMetaData.getColumnCount();
        // Print out columns
        String[] columns = new String[columnCount];
        int[] widths = new int[columnCount];
        for(int i=1; i <= columnCount; i++) {
            columns[i-1] = resultSetMetaData.getColumnName(i);
            widths[i-1] = resultSetMetaData.getColumnDisplaySize(i);
        }
        System.out.println(makeSeparator(widths));
        System.out.println(makeRow(columns, widths));
        // Print out rows
        System.out.println(makeSeparator(widths));
        String[] rowData = new String[columnCount];
        while(resultSet.next()) {
            for(int i=1; i <= columnCount; i++) {
                rowData[i-1] = resultSet.getString(i);
            }
            System.out.println(makeRow(rowData, widths));
        }
        System.out.println(makeSeparator(widths));
    } catch(SQLException sqle) {
        System.err.println("Error executing query: " + sqle);
    }
}
```

```
    } finally {
        closeConnection(connection);
    }
    System.out.println();
}

/** Perform a nonrigorous test for the JDBC version.
 * Initially, a last() operation is attempted for
 * JDBC 2.0. Then, calls to getJDBCMajorVersion and
 * getJDBCMinorVersion are attempted for JDBC 3.0.
 */
public void checkJDBCVersion() {
    System.out.println();
    System.out.println("Checking JDBC version ...\\n");
    Connection connection = getConnection();
    if (connection == null) {
        System.out.println("Check failed.");
        return;
    }
    int majorVersion = 1;
    int minorVersion = 0;
    try {
        Statement statement = connection.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
        String query = "SELECT * FROM authors";
        ResultSet resultSet = statement.executeQuery(query);
        resultSet.last(); // JDBC 2.0
        majorVersion = 2;
    } catch(SQLException sqle) {
        // Ignore - last() not supported
    }
    try {
        DatabaseMetaData dbMetaData = connection.getMetaData();
        majorVersion = dbMetaData.getJDBCMajorVersion(); // JDBC 3.0
        minorVersion = dbMetaData.getJDBCMinorVersion(); // JDBC 3.0
    } catch(Throwable throwable) {
        // Ignore - methods not supported
    } finally {
        closeConnection(connection);
    }
    System.out.println("JDBC Version: " +
        majorVersion + "." + minorVersion);
}

// A String of the form "| xxx | xxx | xxx |"

private String makeRow(String[] entries, int[] widths) {
    String row = "|";
    for(int i=0; i<entries.length; i++) {
        row = row + padString(entries[i], widths[i], " ");
        row = row + " |";
    }
    return(row);
}

// A String of the form "+-----+-----+-----+-----+
```

```
private String makeSeparator(int[] widths) {
    String separator = "+";
    for(int i=0; i<widths.length; i++) {
        separator += padString("", widths[i] + 1, "-") + "+";
    }
    return(separator);
}

private String padString(String orig, int size,
                        String padChar) {
    if (orig == null) {
        orig = "<null>";
    }
    // Use StringBuffer, not just repeated String concatenation
    // to avoid creating too many temporary Strings.
    StringBuffer buffer = new StringBuffer(padChar);
    int extraChars = size - orig.length();
    buffer.append(orig);
    for(int i=0; i<extraChars; i++) {
        buffer.append(padChar);
    }
    return(buffer.toString());
}

/** Obtain a new connection to the database or return
 * null on failure.
 */

public Connection getConnection() {
    try {
        Class.forName(driver);
        Connection connection =
            DriverManager.getConnection(url, username,
                                      password);
        return(connection);
    } catch(ClassNotFoundException cnfe) {
        System.err.println("Error loading driver: " + cnfe);
        return(null);
    } catch(SQLException sqle) {
        System.err.println("Error connecting: " + sqle);
        return(null);
    }
}

/** Close the database connection. */

private void closeConnection(Connection connection) {
    try {
        connection.close();
    } catch(SQLException sqle) {
        System.err.println("Error closing connection: " + sqle);
        connection = null;
    }
}

public static void main(String[] args) {
    if (args.length < 5) {
```

```

        printUsage();
        return;
    }
    String vendor = args[4];
    // Change to DriverUtilities2.loadDrivers() to
    // load the drivers from an XML file.
    DriverUtilities.loadDrivers();
    if (!DriverUtilities.isValidVendor(vendor)) {
        printUsage();
        return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
        DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];

    TestDatabase database =
        new TestDatabase(driver, url, username, password);
    database.testConnection();
    database.createTable();
    database.executeQuery();
    database.checkJDBCVersion();
}
private static void printUsage() {
    System.out.println("Usage: TestDatabase host dbName " +
        "username password vendor.");
}
}

```

18.5 建立 music 表

本书的 JDBC 示例都使用 Microsoft Access Northwind 数据库中的 Employees 表和自定义的 music 表，如表 18.1 所示。

表 18.1 music 表

ID	COMPOSER	CONCERTO	AVAILABLE	PRICE
1	Mozart	No. 21 in C# minor	7	24.99
2	Beethoven	No. 3 in C minor	28	10.99
3	Beethoven	No. 5 Eb major	33	10.99
4	Rachmaninov	No. 2 in C minor	9	18.99
5	Mozart	No. 24 in C minor	11	21.99
6	Beethoven	No. 4 in G	33	12.99
7	Liszt	No. 1 in Eb major	48	10.99

music 表汇总了价格以及现有的各个古典作曲家的协奏曲唱片。要在数据库创建 music 表，可以运行 CreateMusicTable.java 或 create_music_table.sql，下面的小节对此进行详细的说明。

18.5.1 使用 CreateMusicTable.java 创建 music 表

Java 程序 CreateMusicTable.java 列在清单 18.4 中，它的目的是创建 music 表。由于 CreateMusicTable.java 在 coreservlets 包中，因此，这个文件必须位于子目录 coreservlets 中。在编译这个文件之前，必须设置 CLASSPATH，使之包括含有 coreservlets 目录的目录(参见 2.7 节)，并在 coreservlets 子目录中运行 javac CreateMusicTable.java 编译这个程序。但是，如果要创建 music 表，在执行 CreateMusicTable 时必须指出完整的包名，如下面的命令所示：

```
Prompt> java coreservlets.CreateMusicTable host dbName  
username password vendor
```

其中 *host* 是数据库服务器的主机名，*dbName* 指出要将表载入到哪个数据库中，*username* 和 *password* 是配置成能够访问数据库的用户，*vendor* 是标识提供商驱动程序(MSACCESS, MYSQL, ORACLE)的关键字。因此，如果 MySQL 运行在本地主机，且含有一个名为 csajsp 的数据库，可能输入的命令是：

```
Prompt> java coreservlets.CreateMusicTable localhost  
CSAJSP brown larry MYSQL
```

其中 *brown* 是用户名，*larry* 是访问数据库的密码。

这个程序使用第 17 章中的两个类：清单 17.5 中的 DriverUtilities 和清单 17.9 中的 ConnectionInfoBean。DriverUtilities 载入驱动程序的信息，并创建到数据库的 URL。ConnectionInfoBean 存储到数据库的连接信息，并能够创建数据库连接。当前，DriverUtilities 支持 Microsoft Access, MySQL 和 Oracle 数据库。如果使用不同的数据库提供商，则必须修改 DriverUtilities 并加入提供商的具体信息。详细信息参见 17.3 节。

清单 18.4 CreateMusicTable.java

```
package coreservlets;  
  
import java.sql.*;  
import coreservlets.beans.*;  
  
/** Create a simple table named "music" in the  
 * database specified on the command line. The driver  
 * for the database is loaded from the utility class  
 * DriverUtilities.  
 */  
  
public class CreateMusicTable {  
    public static void main(String[] args) {  
        if (args.length < 5) {  
            printUsage();  
            return;  
        }  
        String vendor = args[4];  
        // Change to DriverUtilities2.loadDrivers() to  
        // load the drivers from an XML file.  
        DriverUtilities.loadDrivers();  
        if (!DriverUtilities.isValidVendor(vendor)) {  
            printUsage();  
        }  
        ConnectionInfoBean bean = new ConnectionInfoBean();  
        bean.setDriverName(vendor);  
        bean.setHost(args[0]);  
        bean.setPort(Integer.parseInt(args[1]));  
        bean.setDatabaseName(args[2]);  
        bean.setUsername(args[3]);  
        bean.setPassword(args[4]);  
        DriverUtilities.createTable(bean, "music");  
    }  
}  
  
class PrintUsage {  
    public static void main() {  
        System.out.println("Usage: java coreservlets.CreateMusicTable  
                           host dbName  
                           username password vendor");  
    }  
}
```

```
    return;
}
String driver = DriverUtilities.getDriver(vendor);
String host = args[0];
String dbName = args[1];
String url =
    DriverUtilities.makeURL(host, dbName, vendor);
String username = args[2];
String password = args[3];
String format =
    "(id INTEGER, composer VARCHAR(16), " +
    " concerto VARCHAR(24), available INTEGER, " +
    " price FLOAT)";
String[] rows = {
    "(1, 'Mozart',      'No. 21 in C# minor', 7, 24.99)",
    "(2, 'Beethoven',   'No. 3 in C minor', 28, 10.99)",
    "(3, 'Beethoven',   'No. 5 Eb major', 33, 10.99)",
    "(4, 'Rachmaninov', 'No. 2 in C minor', 9, 18.99)",
    "(5, 'Mozart',       'No. 24 in C minor', 11, 21.99)",
    "(6, 'Beethoven',   'No. 4 in G', 33, 12.99)",
    "(7, 'Liszt',        'No. 1 in Eb major', 48, 10.99)" };
Connection connection =
    ConnectionInfoBean.getConnection(driver, url,
                                     username, password);
createTable(connection, "music", format, rows);
try {
    connection.close();
} catch(SQLException sqle) {
    System.err.println("Problem closing connection: " + sqle);
}
}

/** Build a table with the specified format and rows. */
private static void createTable(Connection connection,
                                 String tableName,
                                 String tableFormat,
                                 String[] tableRows) {
try {
    Statement statement = connection.createStatement();
    // Drop previous table if it exists, but don't get
    // error if not. Thus, the separate try/catch here.
    try {
        statement.execute("DROP TABLE " + tableName);
    } catch(SQLException sqle) {}
    String createCommand =
        "CREATE TABLE " + tableName + " " + tableFormat;
    statement.execute(createCommand);
    String insertPrefix =
        "INSERT INTO " + tableName + " VALUES";
    for(int i=0; i
```

```

    private static void printUsage() {
        System.out.println("Usage: CreateMusicTable host dbName " +
                           "username password vendor.");
    }
}

```

18.5.2 使用create_music_table.sql创建music表

SQL脚本create_music_table.sql，在清单18.5中给出，它的目的是创建music表。如果数据库提供商提供运行SQL命令的工具，可以运行这个脚本创建music表。

对于MySQL数据库，可以运行MySQL监视器运行这个SQL脚本，如下所示。

```
mysql> SOURCE create_music_table.sql
```

有关启动MySQL监视器的详细信息，参见18.2节。如果这个脚本不在您启动MySQL监视器的目录中，必须指定脚本的完整路径。

对于Oracle数据库，可以运行SQL*Plus，使用下面两个命令之一执行该SQL脚本。

```
SQL> START create_music_table.sql
```

或

```
SQL> @create_music_table.sql
```

启动SQL*Plus的详细信息，参见18.3节。同样，如果这个脚本不在您启动SQL*Plus的目录中，必须指定脚本的完整路径。

清单18.5 create_music_table.sql

```

/*
 * SQL script to create music table.
 *
 * From MySQL monitor run:
 *   mysql> SOURCE create_music_table.sql
 *
 * From Oracle9i SQL*Plus run:
 *   SQL> START create_music_table.sql
 *
 * In both cases, you may need to specify the full
 * path to the SQL script.
 */

DROP TABLE music;
CREATE TABLE music (
    id INTEGER,
    composer VARCHAR(16),
    concerto VARCHAR(24),
    available INTEGER,
    price FLOAT);
INSERT INTO music
    VALUES (1, 'Mozart', 'No. 21 in C# minor', 7, 24.99);
INSERT INTO music
    VALUES (2, 'Beethoven', 'No. 3 in C minor', 28, 10.99);
INSERT INTO music
    VALUES (3, 'Beethoven', 'No. 5 Eb major', 33, 10.99);
INSERT INTO music
    VALUES (4, 'Rachmaninov', 'No. 2 in C minor', 9, 18.99);

```

```
INSERT INTO music
VALUES (5, 'Mozart', 'No. 24 in C minor', 11, 21.99);
INSERT INTO music
VALUES (6, 'Beethoven', 'No. 4 in G', 33, 12.99);
INSERT INTO music
VALUES (7, 'Liszt', 'No. 1 in Eb major', 48, 10.99);
COMMIT;
```

第 19 章 HTML 表单的创建和处理

本章的主题：

- 表单的数据提交
- 文本控件
- 按钮
- 复选框和单选按钮
- 组合框和列表框
- 文件上传控件
- 服务器端图像映射
- 隐藏域
- 控件组
- 制表次序
- 调试表单用的 Web 服务器

HTML 表单提供一种简单可靠的用户界面，可以用来收集来自于用户的数据，并将数据传输到 servlet 或其他服务器端的程序进行处理。本章中，我们介绍 HTML 4.0 规范所定义的标准表单控件。但在具体介绍每个控件之前，我们首先解释当发出 GET 或 POST 请求时，表单数据如何传输到服务器。

我们还提供一个小型的 Web 服务器，它能够帮助我们理解和调试 HTML 表单发送的数据。这个服务器只是读取浏览器发送给它的所有 HTTP 数据，然后返回一个 Web 页面，将这些行嵌入到 PRE 元素中。在本章的所有例子中，我们都使用这个服务器来展示 HTML 表单在提交时发送到服务器的表单控件数据。

要使用表单，需记住，应该将常规 HTML 文件放在什么地方才能使它们可以被 Web 服务器所访问。不同的服务器这个位置也不相同，参见第 2 章和附录的论述。下面，我们回顾一下 Tomcat，JRun 和 Resin 中，默认 Web 应用中 HTML 文件的位置。

(1) 默认 Web 应用：Tomcat

- 主位置
install_dir/webapps/ROOT
- 对应的 URL
http://host/SomeFile.html
- 更具体的位置(任意子目录)
install_dir/webapps/ROOT/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.html

(2) 默认 Web 应用：JRun

- 主位置
install_dir/servers/default/default-ear/default-war

- 对应的 URL
http://host/SomeFile.html
- 更具体的位置(任意子目录)
install_dir/servers/default/default-ear/default-war/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.html

(3) 默认 Web 应用: Resin

- 主位置
install_dir/doc
- 对应的 URL
http://host/SomeFile.html
- 更具体的位置(任意子目录)
install_dir/doc/SomeDirectory
- 对应的 URL
http://host/SomeDirectory/SomeFile.html

服务器的默认 Web 应用对于实践和学习来说比较有用,但在部署实际的应用时,几乎肯定会使用定制的 Web 应用;详细信息参见 2.11。

19.1 HTML 表单如何传输数据

HTML 表单允许我们在 Web 页面内创建各种用户界面控件,收集用户的输入。每个控件一般都有名称和值,名称在 HTML 中指定,值或者来自于用户输入,或者来自于 HTML 中指定的默认值。整个表单与某个程序的 URL 相关联,这个程序将会处理表单提交的数据,当用户提交表单时(一般通过点击某个按钮),控件的名称和值就以下面形式的字符串发送到指定的 URL。

`name1=value1&name2=value2&...&nameN=valueN`

这个字符串可以通过下面两种方式发送到指定的程序: GET 或 POST。第一种方法——HTTP GET 请求,将表单数据附加到指定 URL 的末尾,中间以问号分隔。第二种方法——HTTP POST,在 HTTP 请求报头和一个空行之后发送这些数据。在下面的例子中,我们明确地展示出 GET 和 POST 请求中数据如何发送到服务器。

例如,清单 19.1(HTML 代码)和图 19.1(典型的结果)给出一个含有两个文本字段的简单表单。生成这个表单的 HTML 元素在本章余下的部分中详细论述,但现在要注意几点。首先,要注意到,一个文本字段的名称为 `firstName`,另一个的名称为 `lastName`。其次,要注意,由于 GUI 控件是文本级别(嵌入式)的控件,因此我们需要使用明确的 HTML 格式对它们进行编排,以确保控件出现在描述它们的文本之后。最后,还要注意到 FORM 元素指定 `http://localhost:8088/SomeProgram` 作为数据将要发送到的 URL。

在提交表单之前,我们首先在本地计算机上的 8088 端口启动 EchoServer 服务器程序。EchoServer,如 19.12 节所述,是一个用于调试的小型 Web 服务器。不管指定什么样的 URL,或者向 EchoServer 发送什么数据,它仅仅是返回一个 Web 页面,其中列出浏览器发送的所

有 HTTP 信息。如图 19.2 所示，当第一个文本字段中填 Joe，第二个填 Hacker 时，提交这个表单，浏览器会请求下面这个 URL：`http://localhost:8088/SomeProgram?firstName=Joe&lastName=Hacker`。

清单 19.1 GetForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>A Sample Form Using GET</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>A Sample Form Using GET</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
First name:
<INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
Last name:
<INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
<INPUT TYPE="SUBMIT"> <!-- Press this button to submit form -->
</FORM>
</CENTER>
</BODY></HTML>
```

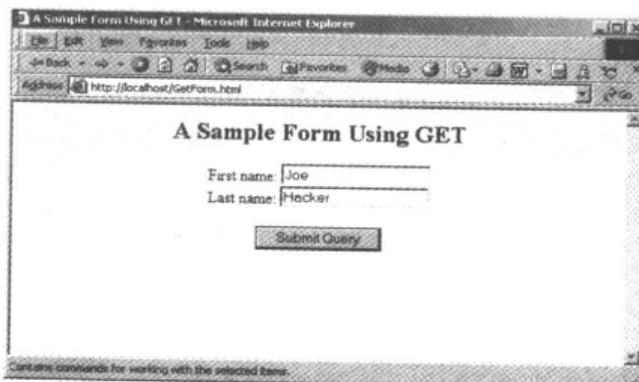


图 19.1 GetForm.html 的初始结果

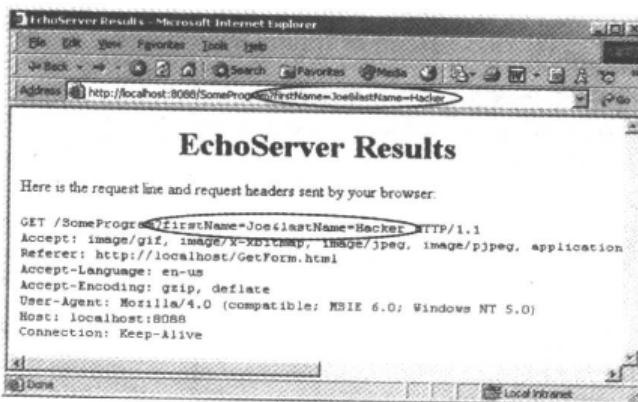


图 19.2 提交 GetForm.html 时 Internet Explorer 6.0 发送的 HTTP 请求

清单 19.2(HTML 代码)和图 19.3(典型结果)给出上面的例子使用 POST(而非 GET)的变种。如图 19.4 所示，文本字段分别为 Joe 和 Hacker 时提交表单，`firstName=Joe&lastName=Hacker`

Hacker 在单独的行中(HTTP 请求报头和一个空行之后)发往服务器。

这就是 HTML 表单的通用思想: GUI 控件收集用户的数据, 每个控件拥有名称和值, 在表单提交时, 将包含所有名/值对的字符串发送到服务器。在 servlet 中, 名称和值的提取比较简单: 这一主题在第 4 章已经介绍过。下面介绍常用的一些表单控件。

清单 19.2 PostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>A Sample Form Using POST</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>A Sample Form Using POST</H2>
<FORM ACTION="http://localhost:8088/SomeProgram"
      METHOD="POST">
  First name:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
  Last name:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT">
</FORM>
</CENTER>
</BODY></HTML>
```

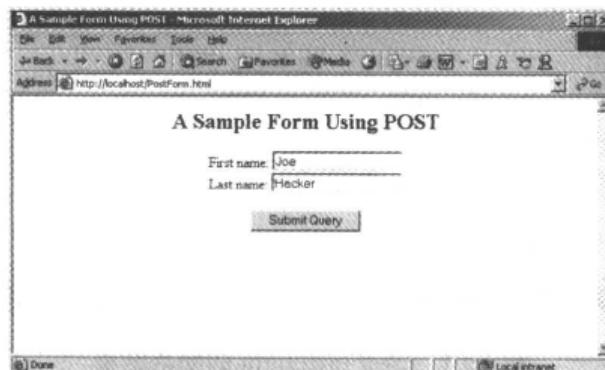


图 19.3 PostForm.html 的初始结果

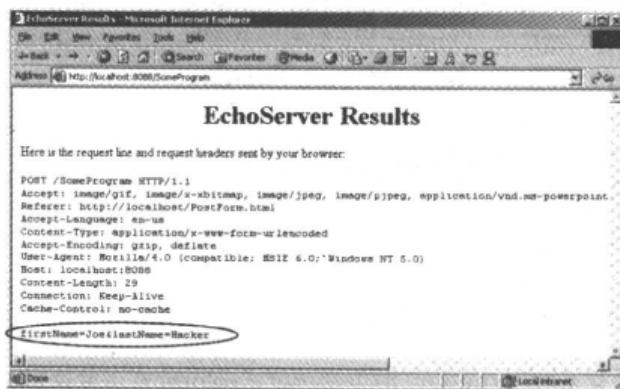


图 19.4 提交 PostForm.html 时由 Internet Explorer 6.0 发送的 HTTP 请求

19.2 FORM 元素

HTML 表单允许我们创建与特定 URL 相关联的一系列数据输入元素。在 HTML 源代码中，一般会为其中的每个元素赋予一个名称，而且，每个元素都有一个值，这个值或者来源于初始的 HTML，或者来自于用户的输入。在提交表单时，所有有效元素的名称和值都被收集到一个字符串中，名称和值之前为=，每个名/值对中间用&分隔。然后，这个字符串被传送到由 FORM 元素指定的 URL。这个字符串或者附加到 URL 上，中间用问号分隔；或者在 HTTP 请求报头和空行之后的单独的行中发送，这要依所使用的提交方法是 GET 还是 POST 而定。本节介绍 FORM 元素自身，它主要用来指定 URL 以及选定提交方法。随后介绍可以用在表单中的各种用户界面控件。

HTML元素: <**FORM ACTION="..." ...> ... </FORM>**

属性: ACTION, METHOD, ENCTYPE, TARGET, ONSUBMIT, ONRESET, ACCEPT, ACCEPT-CHARSET

FORM 元素为数据输入元素创建一块区域，并指定所收集的数据应该传输到哪个 URL。
例如：

```
<FORM ACTION="http://some.isp.com/someWebApp/SomeServlet">
    FORM Input elements and regular HTML
</FORM>
```

本节下面的部分说明适用于 FORM 元素的属性：ACTION, METHOD, ENCTYPE, TARGET, ONSUBMIT, ACCEPT 和 ACCEPT-CHARSET。要注意，我们不讨论那些同样适用于常规 HTML 元素的属性，如 STYLE, CLASS 和 LANG，我们只关注那些专用于 FORM 元素的属性。

(1) ACTION

ACTION 属性指定处理 FORM 数据的服务器端程序(如 <http://www.whitehouse.gov/servlet/schedule-fund-raiser>)。如果服务器端程序就在获取 HTML 表单的服务器，我们推荐这个动作使用相对 URL，而非绝对 URL。采用这种方式，将表单和 servlet 移到不同的主机时，勿需做任何编辑工作。由于我们经常在一台机器上进行开发和测试，之后部署到另外的机器上，因此这种考虑是重要的。例如：

```
ACTION="/servlet/schedule-fund-raiser"
```

核心方法

如果 servlet 或 JSP 页面和 HTML 表单位于同一服务器上，那么，在 ACTION 属性中应该使用相对 URL。

另外，还可以指定将表单数据发送给一个电子邮件地址(如 <mailto:audit@irs.gov>)。一些 ISP 不允许普通用户创建服务器端程序，或者他们额外收费才提供这项权限。在这种情况下，对于需要收集数据但不需要返回结果的页面(如接受产品的订单)，使用电子邮件发送数据比较方便。在使用 mailto URL 时，必须使用 POST 方法(参

见下面小节中的 METHOD)。

还要注意, ACTION 属性不是 FORM 元素必需的属性。如果省略了 ACTION, 表单数据将发送到表单自身的 URL。4.8 节给出使用这种自提交(self-submission)的具体例子。

(2) METHOD

METHOD 属性指定数据如何传输到 HTTP 服务器。在使用 GET 时, 数据附加在指定 URL 之后, 中间用问号分隔。具体的例子参见 19.1 节。GET 是默认的, 并且也是用户在地址栏中输入 URL 或在超链接上单击时, 浏览器使用的方法。在使用 POST 时, 数据在单独的行中发送。表单既可以使用 GET, 也可以使用 POST, 具体选择要依情况而定。

由于 GET 数据是 URL 的一部分, 所以 GET 的优点是可以完成下述工作:

- **保存表单提交的结果。**例如, 我们可以提交数据并记下生成的 URL, 然后通过电子邮件发送给同事, 或将它放入普通的超链接中。能够记下所生成页面的能力是 google.com, yahoo.com 和其他搜索引擎采用 GET 的主要原因。
 - **手动输入数据。**我们可以简单地输入 URL 并附加相应的数据来测试使用 GET 的 servlet 或 JSP。在最初的开发中, 这项功能十分方便。
- 由于 POST 数据不是 URL 的一部分, 所以 POST 的优点在于我们可以完成下述工作:
- **传送大量的数据。**许多浏览器限制 URL 不能超过几千个字符, 这使得 GET 不适用于表单必须发送大量数据的情况。由于 HTML 表单允许我们从客户机上载文件(参见 19.7 节), 因此发送几兆字节的数据十分平常。只有 POST 才可以用于这个任务。
 - **发送二进制数据。**空格、回车、制表符和许多其他字符在 URL 中是不合法的。如果您上载较大的二进制文件, 在传输之前编码所有字符, 并在另一端对它们进行解码, 都会是一件极为耗时的过程。
 - **旁人不能看到用户的机密数据。**HTML 表单允许我们创建密码域, 其中的数据在屏幕上用星号代替。然而, 如果数据以纯文本的形式显示在 URL 中, 使用密码域就毫无意义, 因为窥探者可以从用户的背后窥视, 或在用户离开计算机无人照看时查看浏览器的历史列表得知这些信息。但要注意, 仅仅是使用 POST 方法, 并不能阻止别人在网络连接上使用包嗅探器来侦探用户的机密数据。要防范这种类型的攻击, 需要使用 SSL(<https://>连接)加密网络的数据流。有关在 Web 应用中使用 SSL 的更多信息, 参见本书第二卷有关 Web 应用安全的章节。

在 servlet 中读取 GET 或 POST 数据时, 调用

```
request.getParameter("name")
```

其中 name 是 HTML 表单中输入元素的 NAME 属性的值。其他细节参见第 4 章。要注意, 如果需要的话, 我们可以使用 request.getInputStream 直接读取 POST 数据, 如下所示。

```

int length = request.getContentLength();
if (length > SOME_MAXSIZE) {
    throw new IOException("Possible denial of service attack");
}
byte[] data = new byte[length];
ServletInputStream inputStream = request.getInputStream();
int read = inputStream.readLine(data, 0, length);

```

(3) ENCTYPE

这个属性指定数据在传输前需要完成的编码方式。默认值为 application/x-www-form-urlencoded。依据万维网联盟的规定，这个编码是 UTF-8，只不过客户程序会将空格转换成加号(+)，其他非字母数字字符转换成百分号(%)后跟两个十六进制数字(代表浏览器字符集中的这个字符)。这些转换是在数据项的名称和值之间放入等号(=)、在名/值对之间放入&号之外进行的。

例如，图 19.5 给出 GetForm.html(清单 19.1)的一个版本，名字字段已经输入了“Larry (Java Hacker?)”。在图 19.6 中可以看到，在发送时这一项成为“Larry+%28Java+Hacker%3F%29”。这是因为空格变成了加号，28 是左圆括号的 ASCII 码值(十六进制)，3F 是问号的 ASCII 码值，29 是右圆括号的 ASCII 码值。要注意，除非指定其他方式，否则 POST 数据也按照 application/x-www-form-urlencoded 进行编码。

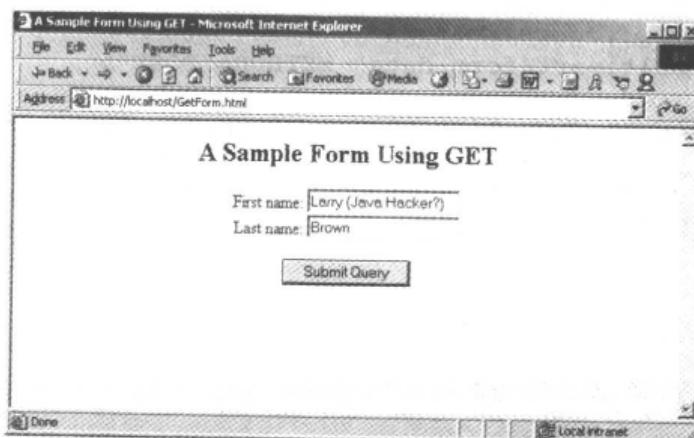


图 19.5 GetForm.html 的定制结果

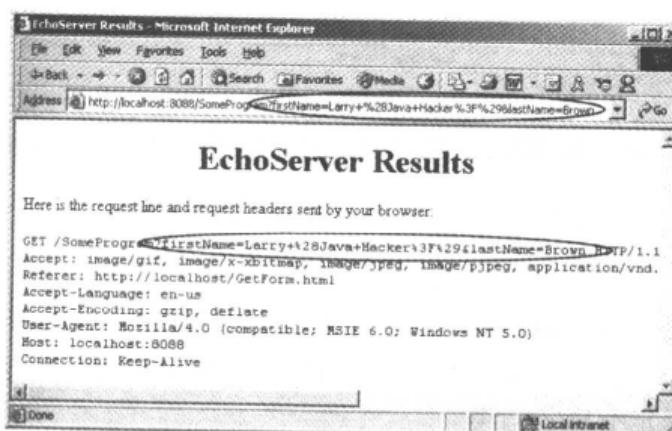


图 19.6 用图 19.5 所示的数据提交 GetForm.html 时由 Internet Explorer 6.0 发送的 HTTP 请求

大多数现代的浏览器都支持另外的 ENCTYPE: multipart/form-data。这种编码方式将每个字段作为 MIME 兼容文档的独立部分进行转换。使用这种 ENCTYPE 时, 方法类型必须指定为 POST。某些情况下, 这种编码方式可以使服务器端的程序对复杂数据的处理更为容易, 并且, 在使用文件上载控件发送整个文档(参见 19.7 节)时也需要用到它。例如, 清单 19.3 给出一个表单, 它与 GetForm.html(清单 19.1)的不同仅仅在于

```
<FORM ACTION="http://localhost:8088/SomeProgram">
```

改为

```
<FORM ACTION="http://localhost:8088/SomeProgram"
      ENCTYPE="multipart/form-data" METHOD="POST">
```

图 19.7 和图 19.8 给出相应的结果。

清单 19.3 MultipartForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Using ENCTYPE="multipart/form-data"</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>Using ENCTYPE="multipart/form-data"</H2>
<FORM ACTION="http://localhost:8088/SomeProgram"
      ENCTYPE="multipart/form-data" METHOD="POST">
  First name:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
  Last name:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT">
</FORM>
</CENTER>
</BODY></HTML>
```

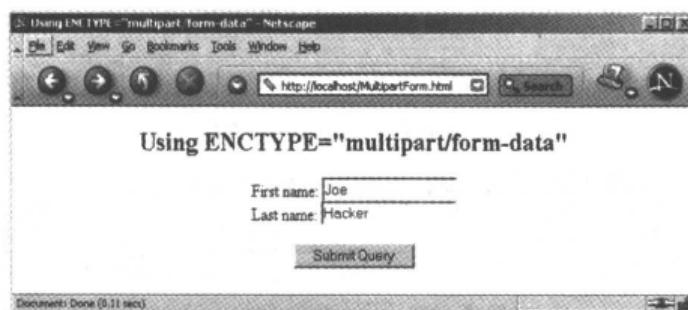


图 19.7 MultipartForm.html 的初始结果

(4) TARGET

具有框架功能的浏览器根据 TARGET 属性来确定应该用哪个框架单元来显示处理表单提交数据的 servlet、JSP 页面或其他程序返回的结果。默认是在包含提交表单的框架单元中显示结果。

(5) ONSUBMIT 和 ONRESET

JavaScript 使用这些属性附加应该在表单提交或重置时进行求值的代码。对于 ONSUBMIT，如果表达式的求值结果为 false，则不提交表单。这样，我们可以在提交表单中字段的值之前，在客户端调用 JavaScript 代码检查它们的格式，提示用户缺失或不合法的项。

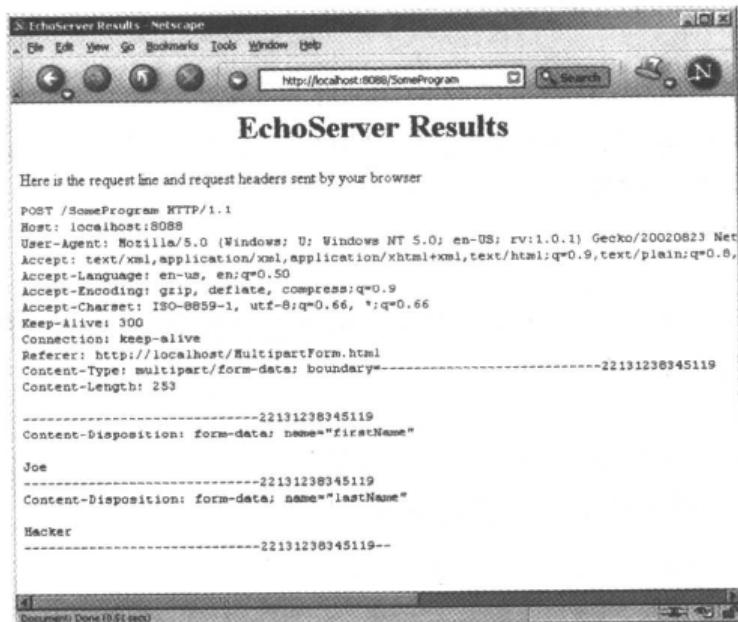


图 19.8 提交 MultipartForm.html 时由 Netscape 7.0 发送的 HTTP 请求

(6) ACCEPT 和 ACCEPT-CHARSET

它们是 HTML 4.0 新加入的属性，用来指定处理表单数据的 servlet 或其他程序必须接受的 MIME 类型(ACCEPT)和字符编码(ACCEPT-CHARSET)。客户还可以使用 ACCEPT 中列出的 MIME 类型限制文件上载元素向用户显示哪些文件类型。

19.3 文本控件

HTML 支持 3 种类型的文本输入元素：文本字段、密码域和文本区域。每种类型的控件都有一个给定的名字，这个名字对应的值取自控件的内容。在表单提交时(一般通过提交按钮来完成)，名字和值都一同发往服务器(参见 19.4 节)。

19.3.1 文本字段

HTML 元素：`<INPUT TYPE="TEXT" NAME="..." ...>`
(无结束标签)

属性：NAME(必需)，VALUE，SIZE，maxlength，onchange，onselect，onfocus，onblur，onkeydown，onkeypress，onkeyup

这个元素创建单行的输入字段，用户可以在其中输入文本，如前面清单 19.1、清单 19.2 和清单 19.3 所示。多行字段请参见后续小节中的 TEXTAREA。虽然为了清楚起见，我们

推荐明确地提供 TYPE，但 TEXT 是 INPUT 表单的默认 TYPE。应该牢记，浏览器的常规自动换行(word-wrapping)也适用于 FORM 元素，因此，要注意使用适当的 HTML 标记确保浏览器不会将描述性的文字与相关联的文本字段分开。

核心方法

尽可能地使用明确的 HTML 构造，将文本字段和它们的描述性文字组合在一起。

如果光标在文本字段中，且表单中有 SUBMIT 按钮(SUBMIT 按钮的细节参见 19.4 节)，用户按下回车时，Netscape 7.0 和 Internet Explorer 6.0 会提交表单。但是，HTML 规范并没有规定这种行为，其他浏览器的行为可能会有所不同。

警告

不要依赖于用户在文本字段中按下回车时浏览器会提交表单。表单中一定要包括明确地提交表单的按钮或图像映射。

为了阻止用户在文本字段中按下回车时浏览器自动提交表单，可以使用 BUTTON 输入控件并提供 onClick 事件处理器，来替代 SUBMIT 按钮。例如，用

```
<INPUT TYPE="BUTTON" VALUE="Check Values"
       onClick="submit()">
```

替代

```
<INPUT TYPE="SUBMIT">
```

提交表单。

下面的小节描述专门适用于文本字段的属性。适用于常规 HTML 元素(如 STYLE, CLASS, ID)的属性不做讨论。TABINDEX 属性——适用于所有表单元素，在 19.11 节中论述。

(1) NAME

NAME 属性在表单提交时标识文本字段。在标准 HTML 中，这个属性是必需的。由于数据总是以名/值对的形式发送到服务器，因此没有 NAME 的表单控件不会发送任何数据。

(2) VALUE

VALUE 属性指定文本字段的初始内容。在表单提交时，文本字段的当前内容会发送到相应的 URL；使用它们可以影响用户的输入。如果表单提交时文本字段为空，则表单数据仅仅由名字和等号构成(例如 name1=value1&textfieldname=&name3=value3)。

(3) SIZE

这个属性，基于文本字段所使用字体的平均字符宽度，指定文本字段的宽度。如果输入的文本超出这个大小，文本字段会自动滚动以容纳这些文本。如果用户输入的字符超出了 SIZE 的数目，或在使用不等宽字体时输入了 SIZE 个较宽的字符(如大写的 W)，都会发生这种情况。Netscape 和 Internet Explorer 6.0 自动在文本字

段中使用不等宽字体。遗憾的是，不能通过在 FONT 或 CODE 元素中嵌入 INPUT 元素来改变字体。但是，我们可以使用层叠样式表来改变输入元素的字体。例如，下面的样式表(在 HTML 页面的 HEAD 节中)将使得所有 INPUT 元素的文本显示为 12 点的 Futura(假定客户机上安装有 Futura 字体)。

```
<style type="text/css">
INPUT {
  font-size : 12pt;
  font-family : Futura;
}
</style>
```

核心方法

默认地，Netscape 和 Internet Explorer 用比例字体显示 INPUT 元素。如果要改变 INPUT 元素的字体，请使用样式表。

(4) MAXLENGTH

MAXLENGTH 给定文本字段允许的最大字符数。要注意区分这个数字和可视字符之间的差别，可视字符通过 SIZE 指定。但要注意，用户可以绕过这项设置的限制：对于 GET 请求，他们可以直接在 URL 中输入数据；而对于 POST 请求，他们可以编写自己的 HTML 表单。因此，服务器端的程序不应该依赖于请求中会包含恰当数量的数据。

(5) ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP

这些属性只适用于支持 JavaScript 的浏览器。它们指定，当修改发生后鼠标离开文本字段时、用户在文本字段中选取文本时、文本字段获得输入焦点时、文本字段失去输入焦点时以及单个键按下或释放时，相应地应该采取的行动。

19.3.2 密 码 域

HTML 元素：`<INPUT TYPE="PASSWORD" NAME="..." ...>`
(无结束标签)

属性：NAME(必需)，VALUE，SIZE，MAXLENGTH，
ONCHANGE，ONSELECT，ONFOCUS，ONBLUR，ONKEYDOWN，
ONKEYPRESS，ONKEYUP

密码域的创建和使用和文本字段相同，只不过用户输入文本时，输入并不回显；而是显示掩盖字符，一般为星号(参见图 19.9)。掩盖输入对于收集信用卡号码或密码等数据比较有用，这些情况下用户不希望靠近计算机的其他人看到这些数据。正常的、未经掩盖的文本(纯粹的文本)在表单提交时作为字段的值传送。

由于 GET 数据在问号后附加在 URL 上，所以密码域一定要使用 POST，这样才能使旁观者不能从显示在浏览器顶端的 URL 中得知未加掩盖的密码。另外，为了传输过程中数据的安全，还应该考虑使用 SSL 来加密数据。使用 SSL 的更多信息参见本书第二卷介绍 Web

应用安全的相关章节。

核心方法

为了保护用户的隐私，在创建含有密码域的表单时，一定要使用 POST。作为补充安全措施，在传输数据时要使用 https://(使用 SSL 来加密数据)。

NAME, VALUE, SIZE, MAXLENGTH, ONCHANGE, ONSELECT, ONFOCUS,
ONBLUR, ONKEYDOWN, ONKEYPRESS 和 ONKEYUP

密码域的属性和文本字段的属性使用方式相同。



图 19.9 通过<INPUT TYPE="PASSWORD" ...>创建的密码域

19.3.3 文本区域

HTML 元素: <TEXTAREA NAME="..."
ROW=xxx COLS=yyy> ...
</TEXTAREA>

属性: NAME(必需), ROWS(必需), COLS(必需), WRAP(非标准),
ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN,
ONKEYPRESS, ONKEYUP

TEXTAREA 元素创建多行文本区域，如图 19.10 所示。这个元素没有 VALUE 属性；它将起始标签和结束标签之间的文本作为文本区域的初始内容。就处理方式而言，<TEXTAREA ...> 和 </TEXTAREA> 之间的初始文本类似于现已废弃的 XMP 元素中的文本。也就是说，维护这段初始文本内的空格，除字符项 <, © 等按照正常的方式解释以外，起始标签和结束标签之间的 HTML 标记逐字取出。除非表单使用自定义的 ENCTYPE(参见 19.2 节)，否则，字符(包括那些由字符项生成的字符在内)在传输之前都按 URL 的方式进行编码。即空格成为加号，其他非字母数字字符成为%XX，其中 XX 为字符的十六进制数值。

(1) NAME

这个属性指定发送到服务器的名字。

(2) ROWS

ROWS 指定文本可视行的数目。如果输入了更多的文本，浏览器会为文本区域添加垂直滚动条。

(3) COLS

COLS 基于所使用字体中字符的平均宽度，指定文本区域的可视宽度。在 Netscape 7.0 和 Internet Explorer 6.0 中，如果单行的文本超出所允许的指定宽度，文本被自动换到下一行显示。但是，如果单个单词拥有超过指定宽度的字符，Internet Explorer 6.0 会将这个单词换到下一行显示，而 Netscape 7.0 则添加水平滚动条将单词保持在一行中。其他浏览器的行为可能会有所不同。

(4) ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN,

ONKEYPRESS 和 ONKEYUP

这些属性只适用于支持 JavaScript 的浏览器；它们指定特定条件发生时应该执行的代码。ONCHANGE 处理文本区域被更改之后输入焦点离开的情况，ONSELECT 描述用户选取文本区域内的文本时应该做些什么，ONFOCUS 和ONBLUR 指定文本区域获得或失去输入焦点时应该做什么，其他的属性确定键入单个键时应该做什么。

清单 19.4 创建一个文本区域，它有 5 个可视行，每行可以保存大约 30 个字符。结果在图 19.10 中给出。

清单 19.4 TEXTAREA 表单控件示例

```
<CENTER>
<P>
Enter some HTML:<BR>
<TEXTAREA NAME="HTML" ROWS=5 COLS=30>
Delete this text and replace
with some HTML to validate.
</TEXTAREA>
<CENTER>
```

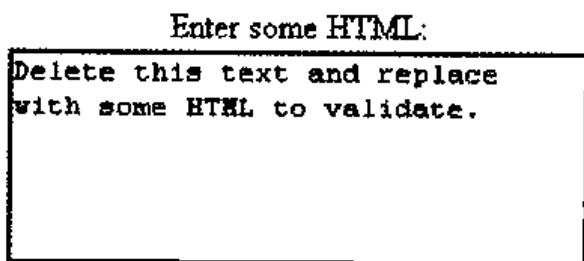


图 19.10 Netscape 7.0 中的文本区域

19.4 按 钮

按钮在 HTML 表单中用于两种目的：提交表单和将控件重置为 HTML 中指定的初始值。使用 JavaScript 的浏览器还可以将按钮用作第三种目的：触发任意 JavaScript 代码。

传统上，INPUT 元素创建的按钮都和 TYPE 属性 SUBMIT、RESET 或 BUTTON 一同使用。HTML 4.0 引入了 BUTTON 元素，Internet Explorer 6.0 和 Netscape 7.0 都支持它。这些新的元素允许我们创建多行标签、图像、其他字体或诸如此类的按钮。但是，早期的浏览器可能不支持 BUTTON 属性。

19.4.1 提 交 按 钮

1. HTML 元素：`<INPUT TYPE="SUBMIT" ...>`（没有结束标签）

属性：NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

单击提交按钮之后，表单会发送到由 FORM 的 ACTION 参数指定的 servlet 或其他服务器端程序。尽管可以用其他方式触发这个动作，比如在图像映射上单击，但是，大多数

表单至少会提供一个提交按钮。提交按钮，和其他表单控件一样，采用客户计算机操作系统的外观和风格，因而在不同的平台上看起来也会有所不同。图 19.11 给出 Windows 2000 专业版上由<INPUT TYPE="SUBMIT">创建的提交按钮。



图 19.11 使用默认标签的提交按钮

(1) NAME 和 VALUE

大多数输入元素都有一个名字和一个相关联的值。在表单提交时，有效元素的名字和值都会拼接到表单的数据字符串中。如果只是用提交按钮来驱动表单的提交，则按钮的名字可以省略，这对发送的数据字符串没有任何影响。如果提供名字，那么，只有实际被点击的按钮的名字和值才会被发送。这种功能使得您可以使用多个按钮，并检查哪个被按下。标签作为传输的值。提供明确的 VALUE 值可以改变默认的标签。

例如，清单 19.5 创建一个文本字段和两个提交按钮，如图 19.12 所示。如果选择第一个按钮，则发送到服务器的数据字符串将会是 Item=256MB+SIMM&Add=Add+Item+to+Cart。

清单 19.5 SUBMIT 输入控件示例

```
<CENTER>
Item:
<INPUT TYPE="TEXT" NAME="Item" VALUE="256MB SIMM"><BR>
<INPUT TYPE="SUBMIT" NAME="Add"
       VALUE="Add Item to Cart">
<INPUT TYPE="SUBMIT" NAME="Delete"
       VALUE="Delete Item from Cart">
</CENTER>
```

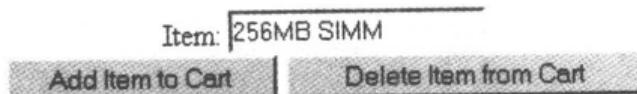


图 19.12 使用用户自定义标签的提交按钮

要注意，当表单数据提交给 servlet 时，对于没有按下的按钮，request.getParameter 返回 null。因而，可以简单地检查相应的参数是否为 null 来确定哪个按钮被选取，如下所示。

```
if (request.getParameter("Add") != null) {
    doCartAdditionOperation(...);
} else if (request.getParameter("Delete") != null) {
    doCartDeletionOperation(...);
}
```

(2) ONCLICK, ONDBLCLICK, ONFOCUS 和 ONBLUR

这些非标准的属性由具有 JavaScript 能力的浏览器用来将 JavaScript 代码和按钮关联起来。ONCLICK 和 ONDBLCLICK 代码在按钮压下后执行，ONFOCUS 代码在

按钮获得输入焦点时执行，ONBLUR 代码在按钮失去焦点时执行。如果附加到按钮上的代码返回 false，表单的提交会被取消。HTML 属性大小写不敏感，Java 程序员一般将这些属性称为 onClick, onDblClick, onFocus 和 onBlur。

2. HTML 元素: <BUTTON TYPE="SUBMIT" ...>

HTML 标记
</BUTTON>

属性: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

这种创建提交按钮的方式允许我们使用任意 HTML 标记作为按钮的内容。这个元素允许我们拥有多行的按钮标签、不同字体的按钮标签、图像按钮等。清单 19.6 给出几个例子，相应的结果在图 19.13 给出。

NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS 和 ONBLUR

这些属性的使用方式与它们在<INPUT TYPE="SUBMIT" ...>中的使用方式相同。

清单 19.6 ButtonElement.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>The BUTTON Element</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE">
<CENTER>
<H2>The BUTTON Element</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
<BUTTON TYPE="SUBMIT">Single-line Label</BUTTON>
  &nbsp; &nbsp;
<BUTTON TYPE="SUBMIT">Multi-line<BR>label</BUTTON>
<P>
<BUTTON TYPE="SUBMIT">
<B>Label</B> with <I>font</I> changes.
</BUTTON>
<P>
<BUTTON TYPE="SUBMIT">
<IMG SRC="images/Java-Logo.gif" WIDTH="110" HEIGHT="101"
      ALIGN="LEFT" ALT="Java Cup Logo">
Label<BR>with image
</BUTTON>
</FORM>
</CENTER>
</BODY></HTML>
```

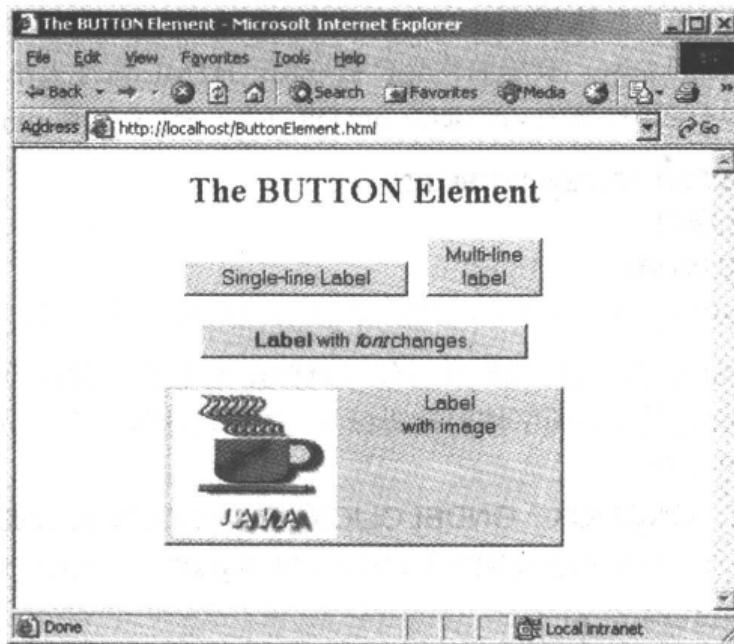


图 19.13 用 BUTTON 元素创建的提交按钮

19.4.2 重置按钮

1. HTML元素: <INPUT TYPE="RESET" ...> (没有结束标签)

属性: VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

重置按钮的用途是将所有控件的值都重置为 VALUE 参数指定的形式。它们的值从不作为表单的内容传输。

(1) VALUE

VALUE 属性指定按钮的标签; 默认为“Reset”。

(2) NAME

由于重置按钮在表单提交时并不向数据字符串加入任何内容, 在标准 HTML 中并不对它们命名。但是, JavaScript 允许使用 NAME 属性来简化对该元素的引用。

(3) ONCLICK, ONDBLCLICK, ONFOCUS 和 ONBLUR

这些非标准的属性由具备 JavaScript 能力的浏览器将 JavaScript 代码关联到按钮。ONCLICK 和 ONDBLCLICK 代码在按钮压下时执行, ONFOCUS 代码在按钮获得输入焦点时执行, ONBLUR 代码在按钮失去焦点时执行。HTML 属性大小写不敏感, JavaScript 程序员一般将这些属性称为 onClick, onDblClick, onFocus 和 onBlur。

2. HTML元素: <BUTTON TYPE="RESET" ...>

```
HTML标记
</HTML>
```

属性: VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

这种创建重置按钮的方式允许我们将任意 HTML 标记作为按钮的内容。所有属性的使用与<INPUT TYPE="RESET" ...>相同。

19.4.3 JavaScript 按钮

1. HTML 元素: <INPUT TYPE="BUTTON" ...> (没有结束标签)

属性: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

这个元素只能由支持 JavaScript 的浏览器所识别。它创建与 SUBMIT 或 RESET 按钮相同外观的按钮, 且允许创作者附加 JavaScript 代码到 ONCLICK, ONDBLCLICK, ONFOCUS 或 ONBLUR 属性。与 JavaScript 按钮相关联的名/值对在表单提交时并不作为数据的一部分传送。可以将任意代码关联到这种按钮, 但最通常的应用是在表单提交给服务器前检验所有输入元素的格式是否正确。例如, 下面这段 HTML 创建一个按钮, 在激活时, 调用 validateForm 函数。

```
<INPUT TYPE="BUTTON" VALUE="Check Values"
       onClick="validateForm()">
```

2. HTML 元素: <BUTTON TYPE="BUTTON" ...>

HTML 标记

```
</BUTTON>
```

属性: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

这种创建 JavaScript 按钮的方式允许我们使用任意 HTML 标记作为按钮的内容。所有属性的用法与<INPUT TYPE="BUTTON" ...>相同。

19.5 复选框和单选按钮

复选框和单选按钮允许用户在一组预定义的选项中进行选择。复选框可以单独选定或取消选定, 单选按钮可以归组到一起, 从而使用户只能选取该组内的单个成员。

19.5.1 复 选 框

HTML 元素: <INPUT TYPE="CHECKBOX" NAME="..." ...>

(没有结束标签)

属性: NAME (必需), VALUE, CHECKED, ONCLICK, ONFOCUS, ONBLUR

这个输入元素创建复选框, 表单提交时, 仅当复选框被选定的情况下, 它的名/值对才会传送。例如, 下面的代码会生成图 19.14 所示的复选框。

```
<P>
<INPUT TYPE="CHECKBOX" NAME="noEmail" CHECKED>
Check here if you do <I>not</I> want to
get out email newsletter
```

Check here if you do *not* want to get our email newsletter

图 19.14 HTML 复选框

要注意, 与复选框相关联的描述性文本是常规的 HTML, 需要小心, 以保证它出现在

复选框之后。因此，前述例子中的<P>保证复选框不出现在前面的段落。

核心方法

FORM 内段落的加载和换行与常规的段落相同。因此，一定要插入明确的 HTML 标记以保证输入元素与描述它们的文本在一起。

(1) NAME

这个属性提供发送到服务器的名字。对于标准的 HTML 复选框，NAME 属性是必需的，和 JavaScript 一同使用时才是可选的。

(2) VALUE

VALUE 属性是可选的，默认为 on。回顾一下，在表单提交时，仅当复选框被选取的情况下，相应的名字和值才会发送到服务器。例如，在前述的例子中，由于复选框被选取，所以 noEmail=on 将会加入到数据字符串中，但如果这个复选框未选定，则不会添加任何内容。因此，servlet、JSP 页面或其他服务器端程序经常只检查复选框名是否存在(即 request.getParameter 返回非 null 值)，而忽略它的值。

(3) CHECKED

如果提供 CHECKED 属性，那么在相关的 Web 页面载入时，复选框的初始状态为选定。否则它的初始状态为未选定。

(4) ONCLICK, ONFOCUS 和 ONBLUR

这些属性提供按钮单击、接收到输入焦点和失去焦点时相应需要执行的 JavaScript 代码。

19.5.2 单选按钮

HTML 元素：<INPUT TYPE="RADIO" NAME="..."
 VALUE="..." ...>(没有结束标签)

属性：NAME (必需), VALUE (必需), CHECKED, ONCLICK, ONFOCUS, ONBLUR

单选按钮不同于复选框的地方仅仅在于，在给定的组中一次只能选取单个单选按钮。我们通过为一组单选按钮提供相同的 NAME，表示它们属于一个组。同一时间，一组中只能有一个按钮被选取：已经有按钮被选取的情况下选取新的按钮，会取消之前的选定。被选定的按钮的值在表单提交时发送。尽管技术上单选按钮不需要相邻出现，但是我们推荐将单选按钮安排在一起。

清单 19.7 给出一个单选按钮的例子。由于输入元素如同正常的段落一样进行换行，DL 列表用来确保这些按钮在生成的页面中依次出现在前一项的下方，并且按照上面的标题进行缩进。图 19.15 展示出这个结果。在这种情况下，表单提交时，creditCard=java 将作为表单数据的一部分发送到服务器。

清单 19.7 单选按钮组示例

```
<DL>
  <DT>Credit Card:
  <DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="visa">
```

```
Visa
<DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="mastercard">
    Master Card
<DD><INPUT TYPE="RADIO" NAME="creditCard"
    VALUE="java" CHECKED>
        Java Smart Card
<DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="amex">
    American Express
<DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="discover">
    Discover
</DL>
```

Credit Card:

Visa
 Master Card
 Java Smart Card
 American Express
 Discover

图 19.15 HTML 中的单选按钮

(1) NAME

不同于大多数输入元素的 NAME 属性，这个 NAME 属性由多个元素共享。所有与同一名称相关联的单选按钮在逻辑上归为一组，从而在任何给定的时间只能有一个按钮被选定。要注意，属性值大小写敏感，因此下面 HTML 代码会产生两个不属于同一组的单选按钮。

```
<INPUT TYPE="RADIO" NAME="foo" VALUE="Value1">
<INPUT TYPE="RADIO" NAME="FOO" VALUE="Value2">
```

警告

要确保每个单选按钮的 NAME 属性与同属同一逻辑分组的其他组成员精确匹配，包括大小写。

(2) VALUE

VALUE 属性提供表单提交时与 NAME 一同传送的值。它并不影响单选按钮的外观。常规的文本和 HTML 标记围绕单选按钮放置，和复选框相同。

(3) CHECKED

如果提供 CHECKED 属性，那么在相关的 Web 页面载入时，单选按钮的初始状态为选定。否则，它的初始状态为未选定。

(4) ONCLICK, ONFOCUS 和 ONBLUR

这些属性指定按钮被单击、接收到输入焦点和失去焦点时应执行的相应的 JavaScript 代码。

19.6 组合框和列表框

SELECT 元素向用户提供一系列的选项。如果只能选取单项且没有指定可视大小，则在组合框中列出各个选项(下拉式菜单)供用户选择；当允许多重选择或指定可视大小时，则使用列表框。选项本身由嵌入到 **SELECT** 元素中的 **OPTION** 项指定。典型的格式如下：

```
<SELECT NAME="Name" ...>
<OPTION VALUE="Value1">Choice 1 Text
<OPTION VALUE="Value2">Choice 1 Text
...
<OPTION VALUE="ValueN">Choice 1 Text
</SELECT>
```

HTML 4.0 规范还定义了 **OPTGROUP**(只有一个属性 **LABEL**)，可以将 **OPTION** 元素装入其中创建层叠式菜单。

1. HTML 元素：<**SELECT** NAME="..." ...> ... </**SELECT**>

属性：NAME(必需)，SIZE，MULTIPLE，ONCLICK，ONFOCUS，ONBLUR，ONCHANGE

SELECT 创建一个组合框或列表框，供用户在多个选项中进行选取。我们使用 <**SELECT** ...> 和 </**SELECT**> 之间的 **OPTION** 元素指定每项选择。

(1) NAME

NAME 向 servlet、JSP 页面或其他服务器端程序标识该元素。

(2) SIZE

SIZE 给定可见行的数目。如果使用 SIZE，**SELECT** 菜单常表示为列表框，而非组合框。组合框一般是没有提供 SIZE 和 MULTIPLE 时由浏览器使用的表达方式。

(3) MULTIPLE

MULTIPLE 属性规定可以同时选择多个项。如果省略 MULTIPLE，则只允许选择一项。在 servlet 中，使用 `request.getParameterValues` 可以获取列表中被选择项的数组。例如，下面这段代码：

```
String[] listValues = request.getParameterValues("language");
if (listValues != null) {
    for(int i=0; i<listValues.length; i++) {
        String value = listValues[i];
        ...
    }
}
```

可以处理 language 列表中所有选定的值(参见清单 19.8)。要注意，返回数组中，值的次序可能与列表中值的显示次序不对应。

核心方法

如果 **SELECT** 列表允许多重选择，则应使用 `request.getParameterValues` 获取所有选定项组成的数组。

(4) ONCLICK, ONFOCUS, ONBLUR 和 ONCHANGE

这些非标准的属性为能够理解 JavaScript 的浏览器所支持。他们表示项目被点击、获得输入焦点、失去输入焦点和更改后失去焦点时，应该执行的相应代码。

2. HTML 元素: <OPTION ...> (结束标签可选)

属性: SELECTED, VALUE

这个元素指定菜单的选项；它只能用在 SELECT 元素中。

(1) VALUE

如果当前选项被选择，VALUE 给出值和 SELECT 菜单的 NAME 一同传送。这不是显示给用户的文本；显示给用户的文本由列在 OPTION 标签后单独的 HTML 标记给出。

(2) SELECTED

如果提供这个属性，SELECTED 指定在页面初次载入时，特定菜单项被选定。

清单 19.8 创建一个选择编程语言的菜单。由于只允许单项选择，且没有指定可视 SIZE，因此它被显示为组合框。图 19.16 和图 19.17 给出最初的外观和用户单击菜单激活它之后的外观。如果在表单提交时 Java 数据项是被选定的，那么 language=java 会发送到服务器端程序。需要注意的是，被传送的是 VALUE 属性，而非描述性的文本。

清单 19.8 SELECT 菜单示例

```
Favorite language:  
<SELECT NAME="language">  
  <OPTION VALUE="c">C  
  <OPTION VALUE="c++">C++  
  <OPTION VALUE="java" SELECTED>Java  
  <OPTION VALUE="lisp">Lisp  
  <OPTION VALUE="perl">Perl  
  <OPTION VALUE="smalltalk">Smalltalk  
</SELECT>
```



图 19.16 显示为组合框(下拉菜单)的 SELECT 元素

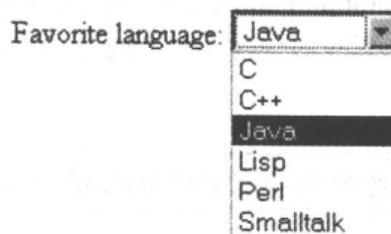


图 19.17 从 SELECT 菜单中选取选项

第二个例子给出显示为列表框的 SELECT 元素。如果在表单提交时多个项目均有效 (active)，则发送多个值，每个值都作为单独的项列出(重复 NAME)。例如，在清单 19.9 给出的例子中(图 19.18)，language=java&language=perl 会添加到发往服务器的数据中。正是

由于存在共享同一名字的多个项, 所以 servlet 的创作者除了要熟悉更为通用的 `getParameter` 方法外, 还需要熟悉 `HttpServletRequest` 的 `getParameterValues` 方法。详细信息参见第 4 章。

清单 19.9 允许多项选择的 SELECT 菜单

```
Languages you know:<BR>
<SELECT NAME="language" MULTIPLE>
<OPTION VALUE="c">C
<OPTION VALUE="c++">C++
<OPTION VALUE="java" SELECTED>Java
<OPTION VALUE="lisp">Lisp
<OPTION VALUE="perl" SELECTED>Perl
<OPTION VALUE="smalltalk">Smalltalk
</SELECT>
```

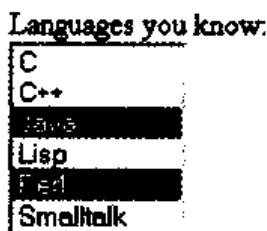


图 19.18 如果 `SELECT` 元素指定了 `MULTIPLE` 或 `SIZE`, 则会显示为列表框

3. HTML 元素: `<OPTGROUP ...> ... </OPTGROUP>`

属性: `LABEL`(必需)

Netscape 7.0 和 Internet Explorer 6.0 都支持这个元素, 使用它可以对菜单选项进行归组。它只能用在 `SELECT` 元素内。

`LABEL`

`LABEL` 给出显示的文本, 对应此菜单选项组。Netscape 和 Internet Explorer 以黑体显示这个标签, 使用倾斜字体。

清单 19.10 创建选择服务器端语言的菜单。此处, 菜单选项用 `OPTGROUP` 元素分类为两组。第一组为常见 servlet 语言, 第二组是常见 CGI 语言。图 19.19 显示出这个菜单, 并且, 常见 CGI 语言选择了 Java。对于这项选择, 发送到服务器的请求数据是 `language=java`。要清楚, Netscape 和 Internet Explorer 不会发送任何信息来标明选定了哪个 `OPTGROUP` 中的选项。从而, 所有的菜单选项都应该使用惟一的值, 不管处于哪个组。

核心方法

使用多个 `OPTGROUP` 时, 要确保所有 `OPTION` 的 `VALUE` 都使用惟一的名字。

清单 19.10 运用 `OPTGROUP` 元素将 `SELECT` 菜单分为两组

```
Server-side Languages:
<SELECT NAME="language">
<OPTGROUP LABEL="Common Servlet Languages">
  <OPTION VALUE="java1">Java
</OPTGROUP>
```

```
<OPTGROUP LABEL="Common CGI Languages">
<OPTION VALUE="c">C
<OPTION VALUE="c++">C++
<OPTION VALUE="java2">Java
<OPTION VALUE="perl">Perl
<OPTION VALUE="vb">Visual Basic
</OPTGROUP>
</SELECT>
```

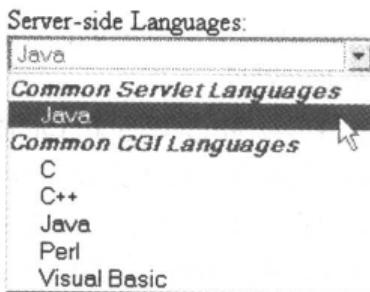


图 19.19 Netscape 7.0 中使用 OPTGROUP 对菜单选项进行归组的 SELECT 元素

19.7 文件上载控件

HTML元素: <INPUT TYPE="FILE" ...> (没有结束标签)

属性: NAME (必需), VALUE (忽略), SIZE, MAXLENGTH, ACCEPT, ONCHANGE, ONSELECT, ONFOCUS, ONBLUR (非标准)

这个元素生成一个文件名字段和一个 Browse 按钮。用户可以直接在文本字段中输入路径，或单击按钮调出文件选择对话框，交互式地选择一个文件路径。只要在初始的 FORM 声明中指定 ENCTYPE 为 multipart/form-data，在表单提交时，文件的内容就会一同传送。对于由多个部分组成的数据，还需要指定 POST 作为方法的类型。使用这个元素可以方便地创建用户支持页面，用户可以通过这种页面发送问题的描述，同时提供任何与问题有关的数据和配置文件。

核心方法

在含有文件上载控件的表单内，一定要指定 ENCTYPE="multipart/form-data" 和 METHOD="POST"。

遗憾的是，servlet API 没有为读取上载的文件提供高级的工具；我们必须调用 request.getInputStream，自己对请求进行分析。幸运的是，许多第三方软件库可以完成这项任务。最为流行的一种来自于 Jakarta 通用库，有关信息参见 <http://jakarta.apache.org/commons/fileupload/>。

(1) NAME

NAME 属性向服务器端程序标识该文本字段。

(2) VALUE

出于安全原因，这个属性是被忽略的；只有最终用户可以指定文件名。否则，恶

意的 HTML 创作者可以通过指定一个文件名，然后，使用 JavaScript 在页面载入时自动提交表单，从而偷窃客户的文件。

(3) SIZE 和 MAXLENGTH

SIZE 和 MAXLENGTH 属性与文本字段中相应属性的使用方式相同，它们分别指定可视字符的数目和最大允许字符的数目。

(4) ACCEPT

ACCEPT 应该是逗号分隔的 MIME 类型列表，用来限制可以使用的文件名。但是，只有极少数的浏览器支持这项属性。

(5) ONCHANGE, ONSELECT, ONFOCUS 和 ONBLUR

这些属性由支持 JavaScript 的浏览器用来指定发生下述事件：用户更改其中的内容后鼠标离开文本字段、用户在文本字段中选择文本、文本字段获得输入焦点、以及文本字段失去输入焦点时，相应地应该采取的动作。

清单 19.11 中的代码创建一个文件上载控件。图 19.20 给出最初的结果，图 19.21 给出 Browse 按钮激活时典型的弹出窗口。

清单 19.11 文件上载控件示例

```
<FORM ACTION="http://localhost:8088/SomeProgram"
      ENCTYPE="multipart/form-data" METHOD="POST">
Enter data file below:<BR>
<INPUT TYPE="FILE" NAME="fileName">
</FORM>
```

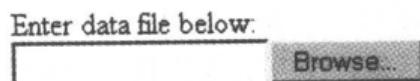


图 19.20 文件上载控件的初始外观

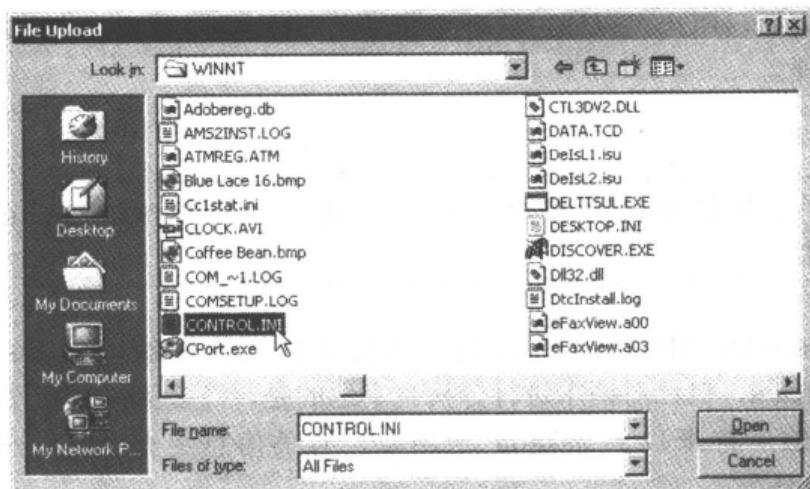


图 19.21 Windows 2000 Professional 上用户单击文件上载控件的 Browse 按钮时生成的文件选择对话框

19.8 服务器端图像映射

在标准的 HTML 中, MAP 元素可以将 URL 关联到图像的各个区域; 然后, 在单击图像的某个指定区域时, 浏览器载入相应的 URL。我们将这种形式的映射称为客户端图像映射(client-side image map)——在客户端决定联系哪个 URL, 没有服务器端程序的参与。HTML 还支持 HTML 表单中的服务器端图像映射(server-side image map)。使用这种映射时, 首先画出一幅图像; 当用户在图像上单击时, 点击的坐标发送到服务器端的程序。

客户端图像映射要简单一些, 也比服务器端图像映射更有效率; 如果仅仅希望将一系列固定的 URL 与某些预定义的图像区域关联起来, 应该使用这种方式。但是, 如果 URL 需要通过计算得出(如气象图)、区域频繁变化或请求中需要用到其他表单数据的情况下, 服务器端图像映射就比较适用。这一节论述完成服务器端图像映射的两种方式。

19.8.1 IMAGE——标准的服务器端图像映射

创建服务器端图像映射的常见方式是通过表单内的<INPUT TYPE="IMAGE" ...>元素。

HTML元素: <INPUT TYPE="IMAGE" ...> (没有结束标签)

属性: NAME(必需), SRC, ALIGN

这个元素显示一幅图像, 用户单击该图像时, 浏览器会将表单发送到由表单的 ACTION 指定的 servlet 或其他服务器端程序。名字本身并不会发送: 相反, 传送的是 *name.x=xpos* 和 *name.y=ypos*, 其中 *xpos* 和 *ypos* 是鼠标点击处相对于图像左上角的坐标。

(1) NAME

NAME 属性在表单提交时标识该文本字段。

(2) SRC

SRC 指定图像的 URL。

(3) ALIGN

ALIGN 属性与 IMG 元素的 ALIGN 属性拥有相同的选项(TOP, MIDDLE, BUTTON, LEFT 和 RIGHT)和默认值(BOTTOM), 并且使用的方式也相同。

清单 19.12 给出一个简单的例子, 其中, 表单的 ACTION 属性指定了 19.12 节中开发的 EchoServer。图 19.22 和图 19.23 展示出图像被单击之前和之后的结果。

清单 19.12 ImageMap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>The IMAGE Input Control</TITLE>
</HEAD>
<BODY>
<H1 ALIGN="CENTER">The IMAGE Input Control</H1>
Which island is Java? Click and see if you are correct.
<FORM ACTION="http://localhost:8088/GeographyTester">
<INPUT TYPE="IMAGE" NAME="map" SRC="images/indonesia.gif">
</FORM>
Of course, image maps can be implemented <B>in</B>
```

```
Java as well. :-)
</BODY></HTML>
```

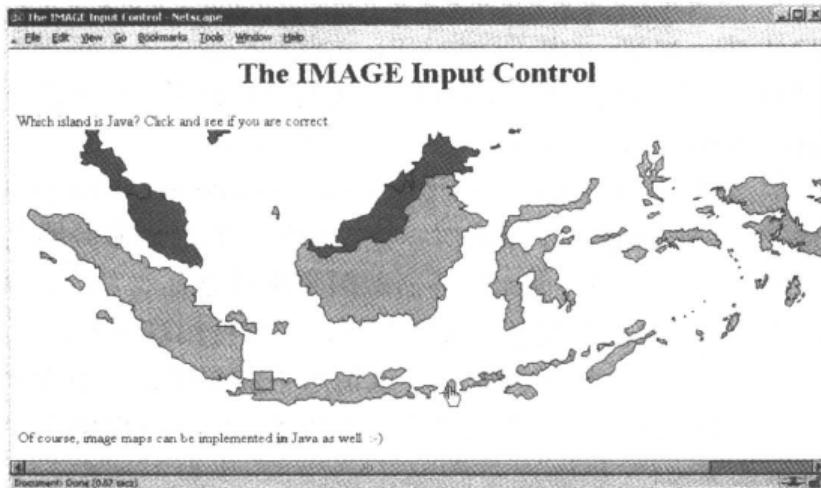


图 19.22 IMAGE 输入控件: NAME="map"

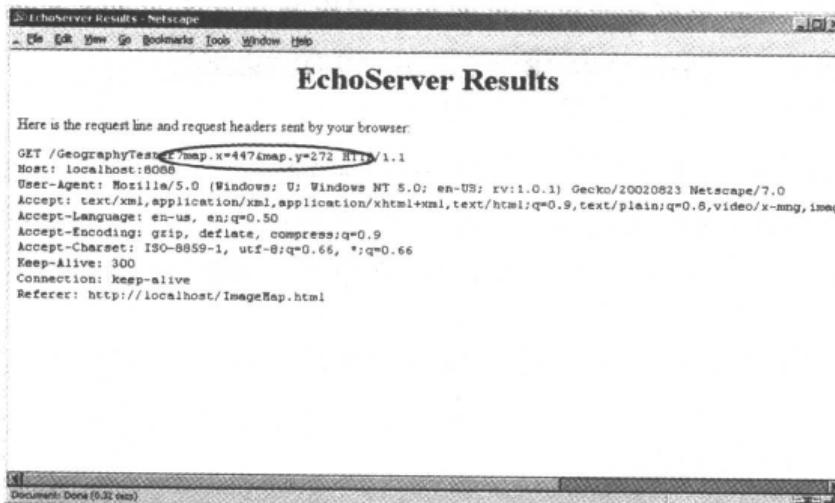


图 19.23 在(447, 272)处单击会提交表单且将 map.x=447&map.y=272 加入到表单数据中

19.8.2 ISMAP——另一种服务器端图像映射

ISMAP 是 IMG 元素的可选属性，它的工作方式与<INPUT TYPE="IMAGE" ...>类似。ISMAP 并不是一个 FORM 元素，但是，我们依旧可以用它来建立连接到 servlet 或其他服务器端程序的简单连接。如果含有 ISMAP 的图像在超链接中，那么在图像上单击会使得点击的坐标发送到指定的 URL。坐标由逗号分开，且以相对于图像左上角的像素数来表示。

例如，清单 19.13 将使用 ISMAP 属性的图像嵌入到超链接 <http://localhost:8088/ChipTester> 中，这个 URL 由 19.12 节中开发的小型 HTTP 服务器负责应答。图 19.24 给出初始的结果，它和没有 ISMAP 属性的图像在显示上完全相同。但是，鼠标在图像的向右 270 像素，左上角以下 189 像素压下时，浏览器会请求 URL <http://localhost:8088/ChipTester?270,189>(如图 19.25 所示)。

如果只是用服务器端图像映射选取一系列静态的指定 URL 中的某个 URL，那么，客

客户端 MAP 元素是更优的选择，因为，采用客户端 MAP 元素时，在确定使用哪个 URL 时不需要联系服务器。如果计划将其他输入元素和图像映射一同使用，那么 IMAGE 输入类型更为适用。但是，对于独立的图像映射，如果与 URL 相关联的区域不断变化或需要计算得出，那么，使用 ISMAP 的图像更合理。

清单 19.13 IsMap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>The ISMAP Attribute</TITLE>
</HEAD>
<BODY>
    <H1 ALIGN="CENTER">The ISMAP Attribute</H1>
    <H2>Select a pin:</H2>
    <A HREF="http://localhost:8088/ChipTester">
        <IMG SRC="images/chip.gif" WIDTH=495 HEIGHT=200 ALT="Chip"
            BORDER=0 ISMAP></A>
    </BODY></HTML>
```

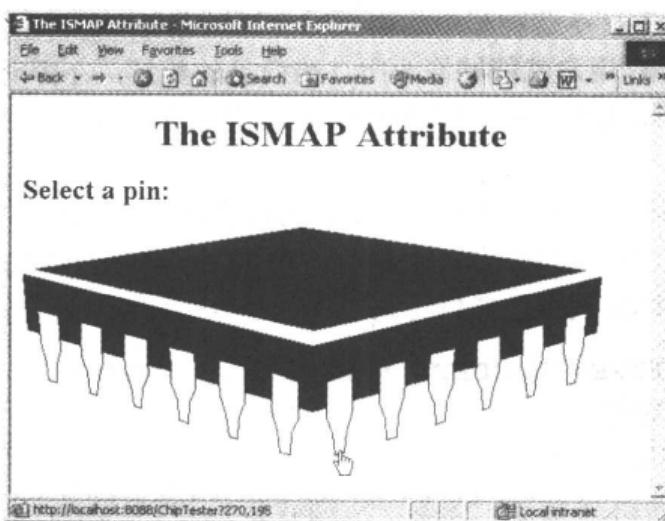


图 19.24 设置超链接中 IMG 元素的 ISMAP 属性可以改变图像被选取时的行为



图 19.25 当 ISMAP 图像被选取时，选择的坐标和 URL 一同发送

19.9 隐 藏 域

隐藏域并不影响提供给用户的页面的外观，而是存储不加改动发送回服务器的固定名字和值，不受用户输入的影响。隐藏域一般用于 3 种目的：

- **跟踪用户。**

用户在网站内到处浏览时，隐藏域内的用户 ID 可以用来跟踪用户访问了哪些页面，或标示用户已经做出的选择。实际上，servlet 的创作者一般依赖于 servlet 会话跟踪 API，而非在这种低的级别实现会话跟踪。会话跟踪的详细信息参见第 9 章。

- **为服务器端程序提供预定义输入。**

当多种静态 HTML 页面作为服务器上同一程序的前端界面时，预定义的隐藏域可以协助提供相关信息，标示请求来源于哪个页面。例如，在线商店可能向引导顾客到自己网站的那些人支付佣金。这种情况下，引导页面可以让访问者通过表单查找商店的目录，嵌入其中的隐藏域给出引导者的 ID。

- **在动态生成的页面中存储上下文相关的信息。**

例如，在表格中显示购物车中的商品时，可以在每一行放置隐藏域，标识特定的商品 ID。这样，用户就能够修改所订购商品的数量，在提交给服务器端程序时，隐藏域可以标识出那些经过修改的商品。而用户则不需要看到 HTML 页面中的商品 ID。

要注意，术语“隐藏”并不意味着字段不能为用户所发现，因为在 HTML 源代码中它们是完全可见的。由于没有可靠的方式来“隐藏”生成页面的 HTML，创作者必须当心，不要在隐藏域中嵌入密码或其他敏感信息。

HTML 元素：<INPUT TYPE="HIDDEN" NAME="..." VALUE="...>
(没有结束标签)

属性：NAME (必需)， VALUE

这个元素可以存储一个名称和一个值，但浏览器并不为它创建任何图形元素。名/值对在表单提交时添加到表单数据中。例如，在下面的例子中，itemID=brown001 总会与表单数据一同发送。

```
<INPUT TYPE="HIDDEN" NAME="itemID" VALUE="brown001">
```

19.10 控 件 组

HTML 4.0 定义了 FIELDSET 元素，以及与之相关联的 LEGEND，使用 FIELDSET 元素可以在外观上将表单内的控件进行分组。要注意，FIELDSET 元素只在 Netscape 6 以及之后的版本，Internet Explorer 6 以及之后的版本中能够起作用。

1. HTML 元素：<FIELDSET> ... </FIELDSET>

属性：没有

这个元素用作容器将相关的控件和(可选地)LEGEND 元素包围起来。除样式表、语言

等统一的属性外，它没有其他属性。清单 19.14 给出一个具体的例子，结果显示在图 19.26 中。

清单 19.14 Fieldset.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>Grouping Controls</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Grouping Controls</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
<FIELDSET>
<LEGEND>Group One</LEGEND>
Field 1A: <INPUT TYPE="TEXT" NAME="field1A" VALUE="Field A"><BR>
Field 1B: <INPUT TYPE="TEXT" NAME="field1B" VALUE="Field B"><BR>
Field 1C: <INPUT TYPE="TEXT" NAME="field1C" VALUE="Field C"><BR>
</FIELDSET>
<FIELDSET>
<LEGEND ALIGN="RIGHT">Group Two</LEGEND>
Field 2A: <INPUT TYPE="TEXT" NAME="field2A" VALUE="Field A"><BR>
Field 2B: <INPUT TYPE="TEXT" NAME="field2B" VALUE="Field B"><BR>
Field 2C: <INPUT TYPE="TEXT" NAME="field2C" VALUE="Field C"><BR>
</FIELDSET>
</FORM>
</BODY></HTML>
```

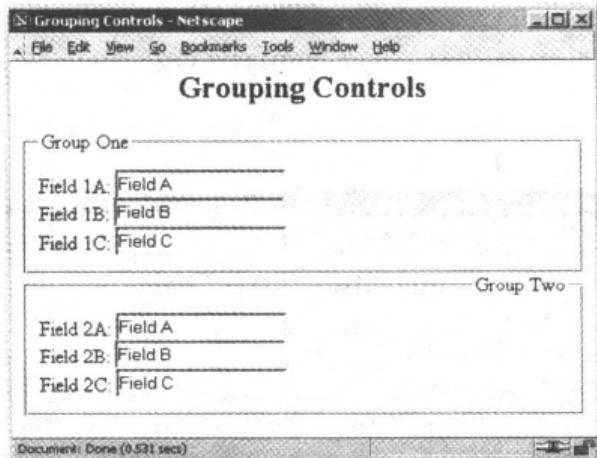


图 19.26 Netscape 7.0 中 FIELDSET 元素将相关控件在外观上组合起来

2. HTML 元素: <LEGEND> . . . </LEGEND>

属性: ALIGN

这个元素在围绕控件组的凹陷边上放置一个标签；它只能用在 FIELDSET 内。

ALIGN

这个属性控制标签的位置。合法的值是 TOP, BOTTOM, LEFT 和 RIGHT, 默认为 TOP。在图 19.26 中, 第一组使用默认的图注对齐方式, 第二组规定 ALIGN="RIGHT"。在 HTML 中, 样式表常常是控制元素对齐的更好方式, 因为它们允许将单个更改传播

到多个页面。

19.11 制表次序

HTML 4.0 定义了 TABINDEX 属性，它可以用在任何可见的 HTML 元素中。TABINDEX 值是一个整数，它控制敲击 TAB 键时元素接收到输入焦点的次序。

在清单 19.15 中，我们提供 3 个文本字段，field1，field2 和 field3。在此，TABINDEX 属性的设置是将制表次序定义为从 field1 到 field3，最后是 field2。HTML 页面显示在图 19.27 中。

一般说来，浏览器使用的隐含制表次序是自顶到底，从左到右。如果您的应用中非得使用非标准的次序，则应明确地声明制表次序。

清单 19.15 Tabindex.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Controlling TAB Order</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Controlling TAB Order</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
Field 1 (first tab selection):
<INPUT TYPE="TEXT" NAME="field1" TABINDEX=1><BR>
Field 2 (third tab selection):
<INPUT TYPE="TEXT" NAME="field2" TABINDEX=3><BR>
Field 3 (second tab selection):
<INPUT TYPE="TEXT" NAME="field3" TABINDEX=2><BR>
</FORM>
</BODY></HTML>
```

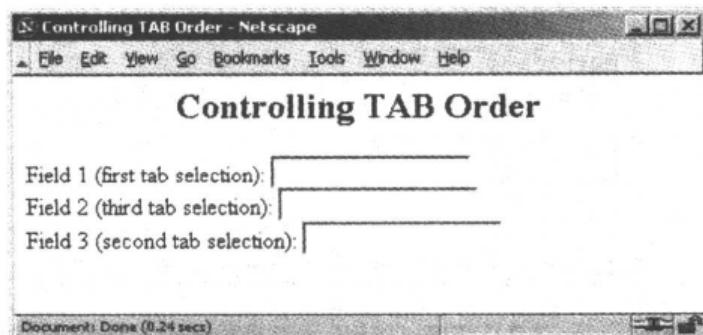


图 19.27 重复敲击 TAB 键输入焦点会在第一个文本字段、第三个文本字段和第二个文本字段之间按照这种次序循环(由 TABINDEX 指定的次序)

19.12 用于调试的 Web 服务器

这一节提供一个小型的“Web 服务器”，它有助于我们理解 HTML 表单的行为。本章之前的许多例子中都用到了它。这个服务器仅仅是读取浏览器发送给它的所有 HTTP 数据，然后返回一个 Web 页面，将这些行嵌入在 PRE 元素中。

这个服务器对于调试 servlet 也很有用。在出现错误时，首要的任务是确定问题出在收集数据上，还是出在对数据的处理上。3.6 节中的 WebClient 程序允许我们查看服务器端程序生成的原始数据；EchoServer 允许我们查看客户表单传来的原始数据。

在本地计算机上的 8088 端口启动 EchoServer，然后将表单的提交地址改为 `http://localhost:8088/`，这样就可以检查收集的数据是否以期望的格式进行组织。除了可以检查发送的表单数据之外，EchoServer 还显示浏览器发送的 HTTP 请求报头。

EchoServer

清单 19.16 提供最上层的服务器代码。我们一般从命令行运行 EchoServer，指定侦听的端口或接受默认的端口 8088。之后，EchoServer 重复地接受来自客户的 HTTP 请求，将所有发送给它的 HTTP 数据封装到 Web 页面内，返回给客户。大多数情况下，服务器会不断读取输入，直到读到空行为止(表示 GET、HEAD 或大多数其他类型 HTTP 请求的结束)。但是，在处理 POST 请求时，服务器会检查 Content-Length 请求报头，并读取空行后相应数目的字节。

清单 19.17 和清单 19.18 提供一些简化网络通信的实用工具类。EchoServer 就建立在这些类的基础之上。

清单 19.16 EchoServer.java

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

/** A simple HTTP server that generates a Web page showing all
 * the data that it received from the Web client (usually
 * a browser). To use this server, start it on the system of
 * your choice, supplying a port number if you want something
 * other than port 8088. Call this system server.com. Next,
 * start a Web browser on the same or a different system, and
 * connect to http://server.com:8088/whatever. The resultant
 * Web page will show the data that your browser sent. For
 * debugging in a servlet or other server-side program, specify
 * http://server.com:8088/whatever as the ACTION of your HTML
 * form. You can send GET or POST data; either way, the
 * resultant page will show what your browser sent.
 */

public class EchoServer extends NetworkServer {
    protected int maxRequestLines = 50;
    protected String serverName = "EchoServer";

    /** Supply a port number as a command-line
     * argument. Otherwise, use port 8088.
     */

    public static void main(String[] args) {
        int port = 8088;
        if (args.length > 0) {
            try {
                port = Integer.parseInt(args[0]);
            } catch(NumberFormatException nfe) {}
        }
    }
}
```

```
        }
        new EchoServer(port, 0);
    }

    public EchoServer(int port, int maxConnections) {
        super(port, maxConnections);
        listen();
    }

    /** Overrides the NetworkServer handleConnection method to
     *  read each line of data received, save it into an array
     *  of strings, then send it back embedded inside a PRE
     *  element in an HTML page.
    */

    public void handleConnection(Socket server)
        throws IOException{
        System.out.println
            (serverName + ": got connection from " +
             server.getInetAddress().getHostName());
        BufferedReader in = SocketUtil.getReader(server);
        PrintWriter out = SocketUtil.getWriter(server);
        String[] inputLines = new String[maxRequestLines];
        int i;
        for (i=0; i<maxRequestLines; i++) {
            inputLines[i] = in.readLine();
            if (inputLines[i] == null) // Client closed connection.
                break;
            if (inputLines[i].length() == 0) { // Blank line.
                if (usingPost(inputLines)) {
                    readpostData(inputLines, i, in);
                    i = i + 2;
                }
                break;
            }
        }
        printHeader(out);
        for (int j=0; j<i; j++) {
            out.println(inputLines[j]);
        }
        printTrailer(out);
        server.close();
    }

    // Send standard HTTP response and top of a standard Web page.
    // Use HTTP 1.0 for compatibility with all clients.

    private void printHeader(PrintWriter out) {
        out.println
            ("HTTP/1.0 200 OK\r\n" +
             "Server: " + serverName + "\r\n" +
             "Content-Type: text/html\r\n" +
             "\r\n" +
             "<!DOCTYPE HTML PUBLIC " +
             "\"-//W3C//DTD HTML 4.0 Transitional//EN\">\r\n" +
             "<HTML>\r\n" +
             "<HEAD>\r\n" +
             "  <TITLE>" + serverName + " Results</TITLE>\r\n" +
```

```
"</HEAD>\n" +
"\n" +
"<BODY BGCOLOR=\"#FDF5E6\">\n" +
"<H1 ALIGN=\"CENTER\">" + serverName +
" Results</H1>\n" +
"Here is the request line and request headers\n" +
"sent by your browser:\n" +
"<PRE>");

}

// Print bottom of a standard Web page.

private void printTrailer(PrintWriter out) {
    out.println
    ("</PRE>\n" +
     "</BODY>\n" +
     "</HTML>\n");
}

// Normal Web page requests use GET, so this server can simply
// read a line at a time. However, HTML forms can also use
// POST, in which case we have to determine the number of POST
// bytes that are sent so we know how much extra data to read
// after the standard HTTP headers.

private boolean usingPost(String[] inputs) {
    return(inputs[0].toUpperCase().startsWith("POST"));
}

private void readpostData(String[] inputs, int i,
                         BufferedReader in)
throws IOException {
    int contentLength = contentLength(inputs);
    char[] postData = new char[contentLength];
    in.read(postData, 0, contentLength);
    inputs[++i] = new String(postData, 0, contentLength);
}

// Given a line that starts with Content-Length,
// this returns the integer value specified.
private int contentLength(String[] inputs) {
    String input;
    for (int i=0; i<inputs.length; i++) {
        if (inputs[i].length() == 0)
            break;
        input = inputs[i].toUpperCase();
        if (input.startsWith("CONTENT-LENGTH"))
            return(getLength(input));
    }
    return(0);
}

private int getLength(String length) {
    StringTokenizer tok = new StringTokenizer(length);
    tok.nextToken();
    return(Integer.parseInt(tok.nextToken()));
}
```

清单 19.17 NetworkServer.java

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

/** A starting point for network servers. You'll need to
 * override handleConnection, but in many cases listen can
 * remain unchanged. NetworkServer uses SocketUtil to simplify
 * the creation of the PrintWriter and BufferedReader.
 */

public class NetworkServer {
    private int port, maxConnections;

    /** Build a server on specified port. It will continue to
     * accept connections, passing each to handleConnection until
     * an explicit exit command is sent (e.g., System.exit) or
     * the maximum number of connections is reached. Specify
     * 0 for maxConnections if you want the server to run
     * indefinitely.
    */

    public NetworkServer(int port, int maxConnections) {
        setPort(port);
        setMaxConnections(maxConnections);
    }

    /** Monitor a port for connections. Each time one is
     * established, pass resulting Socket to handleConnection.
    */

    public void listen() {
        int i=0;
        try {
            ServerSocket listener = new ServerSocket(port);
            Socket server;
            while((i++ < maxConnections) || (maxConnections == 0)) {
                server = listener.accept();
                handleConnection(server);
            }
        } catch (IOException ioe) {
            System.out.println("IOException: " + ioe);
            ioe.printStackTrace();
        }
    }

    /** This is the method that provides the behavior to the
     * server, since it determines what is done with the
     * resulting socket. <B>Override this method in servers
     * you write.</B>
     * <P>
     * This generic version simply reports the host that made
     * the connection, shows the first line the client sent,
     * and sends a single line in response.
    */
}
```

```
protected void handleConnection(Socket server)
    throws IOException{
    BufferedReader in = SocketUtil.getReader(server);
    PrintWriter out = SocketUtil.getWriter(server);
    System.out.println
        ("Generic Network Server: got connection from " +
         server.getInetAddress().getHostName() + "\n" +
         "with first line '" + in.readLine() + "'");
    out.println("Generic Network Server");
    server.close();
}

/** Gets the max connections server will handle before
 * exiting. A value of 0 indicates that server should run
 * until explicitly killed.
 */
public int getMaxConnections() {
    return(maxConnections);
}

/** Sets max connections. A value of 0 indicates that server
 * should run indefinitely (until explicitly killed).
 */
public void setMaxConnections(int maxConnections) {
    this.maxConnections = maxConnections;
}

/** Gets port on which server is listening. */
public int getPort() {
    return(port);
}

/** Sets port. <B>You can only do before "connect" is
 * called.</B> That usually happens in the constructor.
 */
protected void setPort(int port) {
    this.port = port;
}
}
```

清单19.18 SocketUtil.java

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

/** A shorthand way to create BufferedReader and
 * PrintWriters associated with a Socket.
 */

public class SocketUtil {
    /** Make a BufferedReader to get incoming data. */
    public static BufferedReader getReader(Socket s)
```

```
throws IOException {
    return(new BufferedReader(
        new InputStreamReader(s.getInputStream())));
}

/** Make a PrintWriter to send outgoing data.
 * This PrintWriter will automatically flush stream
 * when println is called.
 */

public static PrintWriter getWriter(Socket s)
    throws IOException {
    // Second argument of true means autoflush.
    return(new PrintWriter(s.getOutputStream(), true));
}
```

附录 服务器的组织与结构

本章的主题：

- 下载软件的 URL
- API 帮助文档的位置
- 服务器的配置
- 建立开发环境
- 默认的 Web 应用的目录
- 定制的 Web 应用的目录
- 自动生成的 servlet 代码的目录

本附录汇总 Tomcat, JRun 和 Resin 使用的各种文件和目录；同时介绍如何下载和配置这些服务器软件。

Tomcat

详细信息参见 2.4 节。<http://www.coreservlets.com/> 提供 Tomcat 最新的配置信息。

软件下载

- 访问 <http://jakarta.apache.org/tomcat/>。点击 Binaries，选取 Tomcat 5(servlet 2.4 和 JSP 2.0)或 Tomcat 4(servlet 2.3 和 JSP 1.2)的最新版本。将它们解压缩到选定的位置，后面的内容中，我们将它称为 *install_dir*。

记录 servlet 和 JSP API 文档的位置

Tomcat 捆绑了这份文档，并在服务器主页上提供相应的链接。我们可以将它在磁盘上的位置记录下来，这样，以后即使 Tomcat 没有运行，我们依旧可以访问这份文档。

Tomcat 4

- servlet 和 JSP API
install_dir/webapps/tomcat-docs/servletapi/index.html

Tomcat 5

- servlet API
install_dir/webapps/tomcat-docs/servletapi/index.html
- JSP API
install_dir/webapps/tomcat-docs/jspapi/index.html

服务器配置

- 设置 `JAVA_HOME` 变量。在 `JAVA_HOME` 变量中列出 Java 安装目录的根目录，注意，不是 `bin` 子目录。
- 指定服务器的端口。编辑 `install_dir/conf/server.xml`，将 `Connector` 元素的 `port` 属性从 8080 改为 80。
- 启用 `servlet` 重载。编辑 `install_dir/conf/server.xml`，将下面的内容添加到 `Service` 元素中：

```
<DefaultContext reloadable="true"/>
```
- 启用 `ROOT` 上下文。将 `install_dir/conf/server.xml` 下面的行解除注释：

```
<Context path="" docBase="ROOT" debug="0"/>
```

某些版本的 Tomcat 5 缺失尾部的斜杠；如果是这样，请加上它。
- 开启调用器 `servlet`。将 `install_dir/conf/web.xml` 文件中调用器 `servlet(invoker servlet)` 的 `servlet` 元素和 `servlet-mapping` 元素解除注释。

建立开发环境

- 创建开发目录。在这个目录中开发代码，在测试时复制到服务器的部署目录。
- 设置 `CLASSPATH`。让 `CLASSPATH` 包括 `install_dir/common/lib/servlet.jar`，主开发目录和“`.`”(当前工作目录)。
- 创建启动和停止服务器的快捷方式。在开发目录中，创建调用 `install_dir/bin/startup.bat` 和 `install_dir/bin/shutdown.bat` 的快捷方式。双击它们就可以启动和停止服务器。在 Unix/Linux 上使用 `startup.sh` 和 `shutdown.sh`。

使用默认 Web 应用

默认 Web 应用的主位置是 `install_dir/webapps/ROOT`。如果 `classes` 目录尚不存在，则应创建 `install_dir/webapps/ROOT/WEB-INF/classes`。使用默认 Web 应用之前必须启用 `ROOT` 上下文(参见前面有关服务器配置的节)。

无包装 `servlet`

- 代码：`install_dir/webapps/ROOT/WEB-INF/classes`
- URL：`http://host/servlet/ServletName`

打包的 `servlet`

- 代码：`install_dir/webapps/ROOT/WEB-INF/classes/packageName`
- URL：`http://host/servlet/packageName.ServletName`

打包的 `bean` 和实用工具类

- `install_dir/webapps/ROOT/WEB-INF/classes/packageName`

JAR 文件

- *install_dir/webapps/ROOT/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/webapps/ROOT*
- URL: *http://host/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/webapps/ROOT/directoryName*
- URL: *http://host/directoryName/filename*

使用定制 Web 应用

首先要在 *install_dir/webapps* 中为 Web 应用创建一个目录。这个目录中应该含有 WEB-INF 子目录, WEB-INF 中应该有 web.xml 文件(可以从 ROOT 复制), 并且还要有 WEB-INF/classes 子目录。除使用常规的目录以外, 我们还可以使用依照此种结构的 WAR 文件(实际上就是 JAR 文件, 不过是扩展名从.jar 改为.war)。下面我们使用 *webappName* 指代这个目录的名字(或 WAR 文件的名字, 去掉.war 扩展名)。详细信息, 参见 2.11 节。

无包装 servlet

- 代码位置: *install_dir/webapps/webappName/WEB-INF/classes*
- 默认 URL: *http://host/webappName/servlet/ServletName*
- 定制 URL: *http://host/webappName/AnyName*
(/AnyName 用 web.xml 中的 servlet 和 servlet-mapping 元素指定)

打包的 servlet

- 代码位置: *install_dir/webapps/webappName/WEB-INF/classes/packageName*
- 默认 URL: *http://host/webappName/servlet/packageName.ServletName*
- 定制 URL: *http://host/webappName/AnyName*
(/AnyName 用 web.xml 中的 servlet 和 servlet-mapping 元素指定)

打包的 bean 和实用工具类

- *install_dir/webapps/webappName/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/webapps/webappName/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/webapps/webappName*
- URL: *http://host/webappName/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/webapps/webappName/directoryName*

- URL: *http://host/webappName/directoryName/filename*

如何查看系统为 JSP 页面自动生成的代码

我们可以在下面的位置查看 Tomcat 根据 JSP 页面生成的 servlet 代码。

- 默认 Web 应用

install_dir/work/Standalone/localhost/_

- 定制 Web 应用

install_dir/work/Standalone/localhost/webAppName

JRun

详细信息参见 2.5 节。

软件下载

- 访问 <http://www.macromedia.com/software/jrun/>。根据给出的指示下载免费试用版本。

记录 servlet 和 JSP API 文档的位置

我们可以在线访问这些 API 文档；还可以将它们下载到本地计算机，以加快访问速度。

servlet 2.3 和 JSP 1.2

- 在线文档: <http://jakarta.apache.org/tomcat/tomcat-4.1-doc/servletapi/>
- 下载文档: <http://java.sun.com/products/jsp/download.html>

servlet 2.4 和 JSP 2.0

- servlet 2.4 文档(在线):
<http://jakarta.apache.org/tomcat/tomcat-5.0-doc/servletapi/>
- JSP 2.0 文档(在线):
<http://jakarta.apache.org/tomcat/tomcat-5.0-doc/jspapi/>
- servlet 2.4 和 JSP 2.0 文档(下载):
<http://java.sun.com/products/jsp/download.html>

服务器配置

运行安装向导并指定下述内容：

- **序列号**。如果用做免费的开发服务器，保持为空。
- **用户限制**。限制 JRun 仅能由当前账户使用，还是可以为系统上任何用户所用。
- **SDK 安装位置**。指定 Java 的根目录，注意，不是 bin 子目录。
- **服务器安装位置**。大多数情况下接受默认值。
- **管理员用户名和密码**。可以选取任意值，但要记下来，供以后使用。
- **自动启动功能**。切记，不要将 JRun 做为 Windows 服务。

完成安装之后，到 Start(【开始】)菜单，选择 Programs(【程序】)，选择 Macromedia JRun 4，并选择 JRun Launcher。选择 admin 服务器，单击 Start。接下来，打开浏览器，输入 URL <http://localhost:8000/>。使用在安装过程中指定的用户名和密码登录到服务器，然后选择左窗格中默认服务器下的 Services。接下来，选择 WebService，将端口从 8100 改为 80，单击 Apply，然后停止并重新启动服务器。

建立开发环境

- 创建开发目录。在这个目录中开发代码，在测试时复制到服务器的部署目录。
- 设置 CLASSPATH。让 CLASSPATH 包括 *install_dir/lib/jrun.jar*，主开发目录和“.”(当前工作目录)。
- 创建启动和停止服务器的快捷方式。到 Start(【开始】)菜单，选择 Programs(【程序】)，选择 Macromedia JRun 4，在 JRun Launcher 图标上右击，选择 Copy(【复制】)。然后，跳转到开发目录中，在窗口中右击，然后选择 Paste Shortcut(【粘贴快捷方式】)。不存在单独的关闭按钮；JRun Launcher 既可以启动服务器，也可以停止服务器。

使用默认 Web 应用

默认 Web 应用的主位置是 *install_dir/servers/default/default-ear/default-war/*。

无包装 servlet

- 代码：*install_dir/servers/default/default-ear/default-war/WEB-INF/classes*
- URL：<http://host/servlet/ServletName>

打包的 servlet

- 代码：*install_dir/servers/default/default-ear/default-war/WEB-INF/classes/packageName*
- URL：<http://host/servlet/packageName.ServletName>

打包的 bean 和实用工具类

- *install_dir/servers/default/default-ear/default-war/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/servers/default/default-ear/default-war/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置：*install_dir/servers/default/default-ear/default-war/*
- URL：<http://host/filename>

HTML 和 JSP 页面(子目录中)

- 代码位置：*install_dir/servers/default/default-ear/default-war/directoryName*
- URL：<http://host/directoryName/filename>

使用定制 Web 应用

首先要在 *install_dir/servers/default* 中创建 Web 应用的目录。这个目录中应该含有 WEB-INF 子目录, WEB-INF 目录中应该有 web.xml(可以从默认 Web 应用中复制), 以及 WEB-INF/classes 子目录。除使用常规目录以外, 我们还可以使用依照这种结构的 WAR 文件(实际上就是 JAR 文件, 不过将扩展名从.jar 改为.war)。下面我们使用 *webappName* 指代这个目录的名字(或者 WAR 文件的名字, 去掉.war 扩展名)。详细信息参见 2.11 节。

无包装 servlet

- 代码位置: *install_dir/servers/default/webappName/WEB-INF/classes*
- 默认 URL: *http://host/webappName/servlet/ServletName*
- 定制 URL: *http://host/webappName/AnyName*
(/AnyName 使用 web.xml 中的 servlet 和 servlet-mapping 元素指定)。

打包的 servlet

- 代码位置: *install_dir/servers/default/webappName/WEB-INF/classes/packageName*
- 默认 URL: *http://host/webappName/servlet/packageName.ServletName*
- 定制 URL: *http://host/webappName/AnyName*
(/AnyName 使用 web.xml 中的 servlet 和 servlet-mapping 元素指定)。

打包的 bean 和实用工具类

- *install_dir/servers/default/webappName/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/servers/default/webappName/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/servers/default/webappName*
- URL: *http://host/webappName/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/servers/default/webappName/directoryName*
- URL: *http://host/webappName/directoryName/filename*

如何查看系统为 JSP 页面自动生成的代码

我们可以在下面的位置查看 JRun 根据 JSP 页面生成的 servlet 代码。但是, 除非将 *install_dir/servers/default/SERVER-INF/default-web.xml* 中的 keepGenerated 元素从 false 改为 true, 否则, JRun 不会保存.java 文件。

- 默认 Web 应用
install_dir/servers/default/default-ear/default-war/WEB-INF/jsp
- 定制 Web 应用
install_dir/servers/default/webappName/WEB-INF/jsp

Resin

详细信息参见 2.6 节。

软件下载

- 访问 <http://caucho.com/resin/>。单击页面底部的下载链接，根据给出的指示下载软件。

记录 servlet 和 JSP API 文档的位置

我们可以在线访问这些 API 文档；还可以将它们下载到本地计算机，以加快访问速度。

servlet 2.3 和 JSP 1.2

- 在线文档：
<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/servletapi/>
- 下载文档：
<http://java.sun.com/products/jsp/download.html>

servlet 2.4 和 JSP 2.0

- servlet 2.4 文档(在线)：
<http://jakarta.apache.org/tomcat/tomcat-5.0-doc/servletapi/>
- JSP 2.0 文档(在线)：
<http://jakarta.apache.org/tomcat/tomcat-5.0-doc/jspapi/>
- servlet 2.4 和 JSP 2.0 文档(下载)：
<http://java.sun.com/products/jsp/download.html>

服务器配置

将 Resin 解压缩到选定的目录(此后称为 *install_dir*)，并执行下面两个步骤：

- (1) **设置 JAVA_HOME 变量。** 设置这个变量，使之列出 Java 安装目录的根目录，要注意，不是 bin 子目录。
- (2) **指定端口。** 编辑 *install_dir/conf/resin.conf*，将 http 元素的 port 属性从 8080 改为 80。

建立开发环境

- **创建开发目录。** 在这个目录中开发代码，在测试时复制到服务器的部署目录。
- **设置 CLASSPATH。** 让 CLASSPATH 包括 *install_dir/lib/jsdk23.jar*，主开发目录和“.”(当前工作目录)。
- **创建启动和停止服务器的快捷方式。** 在 *install_dir/bin/httpd.exe* 上右击，选择 Copy(【复制】)。然后，跳转到开发目录，在窗口内右击，选择 Paste Shortcut(【粘贴快捷方式】)。不存在单独的关闭按钮；调用 httpd.exe 会得到一个弹出窗口，上面有 Quit 按钮，使用这个按钮可以停止服务器。

使用默认 Web 应用

主位置是 *install_dir/doc*。

无包装 servlet

- 代码: *install_dir/doc/WEB-INF/classes*
- URL: *http://host/servlet/ServletName*

打包的 servlet

- 代码: *install_dir/doc/WEB-INF/classes/packageName*
- URL: *http://host/servlet/packageName.ServletName*

打包的 bean 和实用工具类

- *install_dir/doc/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/doc/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/doc*
- URL: *http://host/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/doc/directoryName*
- URL: *http://host/directoryName/filename*

使用定制 Web 应用

首先要在 *install_dir/webapps* 中创建 Web 应用的目录。这个目录中应该含有 WEB-INF 子目录, WEB-INF 目录中应该含有 web.xml(可以从默认 Web 应用中复制), 以及 WEB-INF/classes 子目录。除使用常规目录以外, 我们还可以使用依照这种结构的 WAR 文件(实际上就是 JAR 文件, 不过是扩展名从.jar 改为.war)。下面, 我们使用 *webappName* 指代这个目录名(或 WAR 文件的名字, 去掉.war 扩展名)。详细信息参见 2.11 节。

无包装 servlet

- 代码位置: *install_dir/webapps/webappName/WEB-INF/classes*
- 默认 URL: *http://host/webappName/servlet/ServletName*
- 定制 URL: *http://host/webappName/AnyName*
(/AnyName 使用 web.xml 中的 servlet 和 servlet-mapping 元素指定)。

打包的 servlet

- 代码位置: *install_dir/webapps/webappName/WEB-INF/classes/packageName*
- 默认 URL: *http://host/webappName/servlet/packageName.ServletName*
- 定制 URL: *http://host/webappName/AnyName*

(/AnyName 使用 web.xml 中的 servlet 和 servlet-mapping 元素指定)。

打包的 bean 和实用工具类

- *install_dir/webapps/webappName/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/webapps/webappName/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/webapps/webappName*
- URL: *http://host/webappName/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/webapps/webappName/directoryName*
- URL: *http://host/webappName/directoryName/filename*

如何查看系统为 JSP 页面自动生成的代码

我们可以在下面的目录中查看 Resin 根据 JSP 页面生成的 servlet 代码。

- 默认 Web 应用: *install_dir/doc/WEB-INF/work*
- 定制 Web 应用: *install_dir/webapps/webappName/WEB-INF/work*

(*/AnyName* 使用 web.xml 中的 servlet 和 servlet-mapping 元素指定)。

打包的 bean 和实用工具类

- *install_dir/webapps/webappName/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/webapps/webappName/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/webapps/webappName*
- URL: *http://host/webappName/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/webapps/webappName/directoryName*
- URL: *http://host/webappName/directoryName/filename*

如何查看系统为 JSP 页面自动生成的代码

我们可以在下面的目录中查看 Resin 根据 JSP 页面生成的 servlet 代码。

- 默认 Web 应用: *install_dir/doc/WEB-INF/work*
- 定制 Web 应用: *install_dir/webapps/webappName/WEB-INF/work*

(*/AnyName* 使用 web.xml 中的 servlet 和 servlet-mapping 元素指定)。

打包的 bean 和实用工具类

- *install_dir/webapps/webappName/WEB-INF/classes/packageName*

JAR 文件

- *install_dir/webapps/webappName/WEB-INF/lib*

HTML 和 JSP 页面(不在子目录中)

- 代码位置: *install_dir/webapps/webappName*
- URL: *http://host/webappName/filename*

HTML 和 JSP 页面(子目录中)

- 代码位置: *install_dir/webapps/webappName/directoryName*
- URL: *http://host/webappName/directoryName/filename*

如何查看系统为 JSP 页面自动生成的代码

我们可以在下面的目录中查看 Resin 根据 JSP 页面生成的 servlet 代码。

- 默认 Web 应用: *install_dir/doc/WEB-INF/work*
- 定制 Web 应用: *install_dir/webapps/webappName/WEB-INF/work*