JSP: The Complete Reference

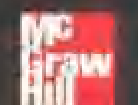
软件开发技术丛书

JSP技术大全

(美) Phil Hanna 著

闻道工作室 译

机械工业出版社
China Machine Press



Education

精彩好书见www.book118.com

软件开发技术丛书

JSP技术大全

（美） Phil Hanna 著

闻道工作室 译

精彩好书见www.book118.com

JSP是全新的网络服务器端编程环境 JSP充分利用了Java的强大功能，是一种优秀的服务器端技术

本书是JSP技术完整的参考手册，不仅适合初学者，对中高级服务器端编程人员更具参考价值

Phil Hanna: JSP: The Complete Reference (ISBN 0-07-212768-6).

Copyright © 2001 by the McGraw-Hill Companies, Inc.

Authorized translation from the English language edition published by McGraw-Hill, Inc.

All rights reserved. For sale in the People's Republic of China.

本书中文简体字版由机械工业出版社和美国麦格劳－希尔国际公司合作出版 未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2001-3933

图书在版编目（CIP）数据

JSP技术大全/（美）汉纳（Hanna, P.）著；闻道工作室译. –北京：机械工业出版社，2002.1

（软件开发技术丛书）

书名原文：JSP: The Complete Reference

ISBN 7-111-09333-X

Ⅰ. J… Ⅱ. ①汉… ②闻… Ⅲ. ①主页制作－手册 ②Java语言－程序设计－手册 Ⅳ. TP393.092-62

中国版本图书馆CIP数据核字（2001）第065418号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：冯晓陆

北京市密云县印刷厂印刷·新华书店北京发行所发行

2002年1月第1版第1次印刷

787mm × 1092mm 1/16 · 39印张

印数：0 001-5 000册

定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

精彩好书见www.book118.com

译 者 序

随着网络技术的发展以及电子商务和企业计算对网络资源需求的不断增加，客户端脚本语言applet已经不能满足人们的需求。它在浏览器的兼容性、安全性和下载时间过长方面的缺陷越来越突出。通用网关接口（CGI）在可扩展性上的缺点也使之不能成为当前服务器端技术的主流。JSP的出现弥补了以上的不足，它已被证明是一种优秀的服务器端技术，是开发Web应用非常好的基础

JavaServer Page（JSP）是使用Java代码动态生成HTML文档的Web页面模板 它运行于称为JSP容器的服务器端组件。JSP容器将JSP转换成等价的Java servlet JSP充分利用了Java的强大功能，真正带有数据库连接、网络访问和多线程操作 它是一种安全、健壮和平台独立的技术，把Java的强大功能引入电子商务和企业Web计算中。

本书是JSP技术的一本完整的参考手册，详细介绍了JSP技术的内容、操作环境及其和servlet、JavaBean及XML的关系以及协同使用方式。对一些如定制标签、数据库操作、测试和调试技术等高级主题也进行了深层次的探讨。并有一个贯穿全书的实例将各种技术结合起来，对读者理解和实际操作该技术有很大的助益。在附录中包含了servlet API、JSP API和HTTP 的参考。

本书内容丰富、讲解生动、图解和代码量大、结构严谨、语言清晰易懂，是当前网络服务器端的前沿技术书籍。不仅适合于JSP技术的初学者，对中高级Java服务器端编程人员更具参考价值。

精彩好书见www.book118.com

前　言

一门新技术的第一次高峰经常被其产生的期望所赶超，这使得第二次高峰得益于以前的经验和实际产生的效果。Java即是如此。客户端浏览器应用（applet）遇到了以下三方面的限制：

- 浏览器不兼容。
- 安全性矫枉过正。
- 由于下载时间过长导致的性能问题。

服务器端Java的出现改变了这一切。Java servlet和JavaServer Page（JSP）提供一种安全、健壮和独立的平台技术，把Java的强大功能引入到电子商务和企业Web计算中。事实表明，人们对JSP的研究兴趣正不断升温，对JSP技术水平的需求也越来越高。几乎所有的财富500强公司已经或准备采纳服务器端的Java应用。

本书目的是提供JSP技术的完整参考，开始是对Web编程环境和JSP元素的简单介绍，然后对高级主题进行深入讲解。

本书组织结构

本书由五部分组成，开始是高层次介绍，接着深入到各主题。

第一部分　Web编程环境

本书开始章节作为编程环境介绍Web、servlet、JSP和Web网络协议。

第1章　Web市场：本章将Web作为一个市场进行探讨，其目标、产品、服务及应用——如何演化到当前形式，将来发展的方向。介绍了Java及其在网络计算模型中的重要性。

第2章　Web应用演化：本章描述应用编程模型如何随Web的成熟而不断进化，以及每一阶段的经验如何引导下一步的需求。

第3章　超级文本传输协议介绍：本章介绍Web客户/服务器模型的底层语言，超级文本传输协议（HTTP）。它发展出对理解Web编程环境至关重要的基本概念。

第4章　servlet介绍：本章中解释了JSP和servlet之间的密切联系。给出了它们之间的共同特点，并图示说明。

第5章　JSP介绍：本章将JavaServer Page（JSP）作为一种服务器端脚本环境加以介绍，描述了servlet引擎及几个重要的示例。这里只讲到基本概念——第二部分对内容进行深入讲解。

第二部分　JSP元素

这部分介绍JSP的语法和语义，使读者熟悉创建工作代码所必须的技巧。内容包括基本语法、scriptlet、表达式、声明、包含文件、转发请求及指定页面行为。也涉及到如何开发定制标签。

第6章　JSP语法和语义：本章包含JavaServer Page的基本语法，给出如何将其并入HTML模板和Java代码。

精彩好书见www.book118.com

第7章　表达式和scriptlet：本章给出将Java代码段合并到JavaServer Page的基本模型。包含合法和非法的使用方式及如何由一至工作servlet的翻译器组成代码段。

第8章　声明：本章给出JSP中声明和Java代码的高级用法，包含最常用的三种声明，均提供实例。

第9章　请求发送：本章描述如何处理多服务器端组件的HTTP请求。这里给出包含其他文件的两种方法，解释为什么其中一种优于另一种，还给出如何使用<jsp:forward>行为将请求传至另一JSP进行处理。

第10章　Page伪指令：本章详细描述如何使用Page伪指令指定JavaServer Page的属性和行为。对每一属性给出完整实例。

第11章　JSP标签扩展：本章考虑JSP结构的扩展，特别是定义定制标签的能力。

第三部分　JSP行为

这部分讲述JSP如何与JDBC、JavaBean和其他主要的Java环境组件协同工作。包含了调试和发布的详细内容。

第12章　HTML窗体：本章讲述HTML窗体，servlet和JavaServer Page最常用的客户端。

第13章　数据库访问：大部分JSP页面都需要访问数据库，本章包含Java数据库连接的概念及如何在基于Web的应用中使用它。

第14章　会话和线程管理：HTTP是一个无状态协议，但JavaServer Page可以使用HTTP会话克服这一限制。本章讨论相关问题及对开发人员可利用的技术。

第15章　JSP和JavaBean：本章描述JavaBean及如何结合JavaServer Page使用它以分离事务逻辑为可重用组件。

第16章　JSP和XML：XML是作为结构化数据存储和交换的普遍语言而产生的。本章讲述JSP如何使用XML进行输入和输出。

第17章　JSP测试和调试：在编程过程中调试技术常被忽视，但却是不可或缺的知识。JavaServer Page提出了自己的观点。本章概括了可应用的一个基本方法学及可利用的工具。

第18章　发布Web应用：本章给出如何将JSP移出开发环境至产品Web环境。

第19章　事例分析：一个产品支持中心：本章通过管理一个技术支持中心的基于Web的系统将全书讨论的元素结合起来。

第四部分 JSP和其他Web组件

这部分给出使用了JavaServer Page的大量的上下文——其如何与servlet、applet、Perl 脚本、FTP、CGI、ASP和其他服务器端代理进行通信。

第20章　与其他客户端进行通信：虽然Web浏览器中HTML窗体是最常用的客户端环境，但JSP页面可用于支持其他任何支持HTTP协议的客户端，本章给出其实现方式。

第21章　与其他服务器通信：对前面几章的思想进行深入开发，本章讲述JSP组件如何访问其他协议，并讨论了JavaMail API。

附录

本书包含三个附录，为Servlet API、JSP API和HTTP 参考。

精彩好书见www.book118.com

目　录

译者序
前言

第一部分　Web编程环境

第1章　Web市场 …… 1
第2章　Web应用演化 …… 2
2.1　Web的产生 …… 2
2.2　Web编程模型的增长 …… 2
2.3　从客户端移向服务器端方案 …… 4
第3章　超文本传输协议介绍 …… 6
3.1　HTTP是什么 …… 6
3.1.1　Internet上请求文档的一种语言 …… 6
3.1.2　HTTP规范 …… 6
3.2　HTTP请求模型 …… 7
3.2.1　连接至Web服务器 …… 7
3.2.2　发送HTTP请求 …… 8
3.2.3　服务器接受请求 …… 9
3.2.4　来自服务器的HTTP响应 …… 9
3.3　实例 …… 10
3.4　小结 …… 13
第4章　servlet介绍 …… 14
4.1　servlet生命期 …… 14
4.1.1　init …… 15
4.1.2　service …… 15
4.1.3　destroy …… 16
4.2　例子：千米每公升到英里每加仑servlet …… 16
4.3　servlet类 …… 19
4.3.1　servlet …… 20
4.3.2　servlet请求 …… 22
4.3.3　servlet响应 …… 24
4.3.4　servlet上下文 …… 26
4.4　线程模型 …… 27
4.5　HTTP会话 …… 28
4.6　小结 …… 30
第5章　JSP介绍 …… 31
5.1　JSP工作方式 …… 31
5.2　一个基本例子 …… 32

第二部分　JSP 元 素

第6章　JSP语法和语义 …… 37
6.1　JSP开发模型 …… 37
6.2　JSP页面组件 …… 38
6.2.1　伪指令 …… 38
6.2.2　注释 …… 40
6.2.3　表达式 …… 40
6.2.4　scriptlet …… 41
6.2.5　声明 …… 42
6.2.6　隐含对象 …… 44
6.2.7　标准行为 …… 45
6.2.8　标签扩展 …… 47
6.3　一个完整实例 …… 47
6.3.1　Page伪指令 …… 51
6.3.2　<jsp:include>行为 …… 51
6.3.3　scriptlet …… 52
6.3.4　表达式 …… 52
6.3.5　一个声明 …… 53
6.4　小结 …… 54
第7章　表达式和scriptlet …… 55
7.1　表达式 …… 55
7.2　scriptlet …… 56
7.3　通过JSP容器处理表达式和scriptlet …… 57
7.3.1　HTML模板数据和表达式 …… 58
7.3.2　scriptlet内容 …… 58
7.3.3　容器生成的初始化和退出代码 …… 59

精彩好书见www.book118.com

7.4 隐含对象和JSP环境 ······60
7.4.1 Request ······61
7.4.2 Response ······62
7.4.3 PageContext ······62
7.4.4 Session ······63
7.4.5 Application ······64
7.4.6 Out ······64
7.4.7 Config ······65
7.4.8 Page ······65
7.4.9 Exception ······65
7.5 初始化参数 ······66
7.6 小结 ······67
第8章 声明 ······68
8.1 声明是什么 ······68
8.2 声明的基本用法 ······72
8.3 变量声明 ······72
8.4 方法定义 ······76
8.4.1 覆盖jspInit和jspDestroy ······80
8.4.2 隐含对象的访问 ······80
8.5 内部类 ······81
8.6 小结 ······83
第9章 请求发送 ······85
9.1 请求过程的剖析 ······85
9.2 包含其他资源 ······86
9.3 include伪指令 ······86
9.3.1 其工作方式 ······87
9.3.2 改变一个被包含文件的影响 ······88
9.3.3 使用include伪指令复制源码 ······88
9.4 <jsp:include>行为 ······90
9.5 使用哪种方法 ······100
9.6 转发请求 ······100
9.7 RequestDispatcher对象 ······103
9.8 模型1对比模型2 ······104
9.9 小结 ······105
第10章 Page伪指令 ······106
10.1 language ······106
10.2 extends ······107
10.2.1 JSP超类所需的接口 ······108
10.2.2 一个JSP超类例子 ······109
10.3 import ······112
10.4 session ······113
10.5 buffer和autoFlush ······113
10.6 isThreadSafe ······114
10.7 info ······114
10.8 contentType ······115
10.9 errorPage和isErrorPage ······115
10.10 小结 ······119
第11章 JSP标签扩展 ······121
11.1 为什么要定制标签 ······121
11.2 开发第一个定制标签 ······122
11.2.1 第1步——定义标签 ······122
11.2.2 第2步——创建TLD入口 ······123
11.2.3 第3步——编写标签处理器 ······124
11.2.4 第4步——将标签并入JSP页面 ······127
11.3 标签处理器工作方式 ······128
11.3.1 JSP容器的功能 ······128
11.3.2 标签处理器功能 ······129
11.4 标签库 ······130
11.4.1 标签库描述器 ······130
11.4.2 taglib伪指令 ······131
11.5 标签处理器API ······132
11.5.1 Tag接口 ······132
11.5.2 TagSupport类 ······133
11.6 标签处理器生命期 ······133
11.6.1 流线图 ······135
11.6.2 生成代码的一个例子 ······136
11.7 定义标签属性 ······141
11.8 体标签处理器API ······146
11.8.1 BodyContent ······147
11.8.2 BodyTag接口 ······148
11.8.3 BodyTagSupport类 ······148
11.9 体标签处理器生命期 ······149
11.10 定义脚本变量 ······151
11.10.1 TagExtraInfo类 ······152

精彩好书见www.book118.com

11.10.2 标签属性有效性检验 ……159
11.11 协作标签 ……159
11.11.1 使用Syntactic Scoping ……159
11.11.2 例子：switch标签 ……160
11.12 数据库查询例子的实现 ……167
11.12.1 所需标签 ……168
11.12.2 标签库描述器 ……168
11.12.3 标签处理器 ……169
11.13 小结 ……177

第三部分 JSP 行 为

第12章 HTML窗体 ……179
12.1 FORM元素 ……180
12.2 窗体输入元素 ……183
12.2.1 使用INPUT标签创建的元素 ……185
12.2.2 使用select和option创建的元素 ……194
12.2.3 textarea元素 ……195
12.3 窗体有效性检验 ……196
12.4 服务器端的窗体处理 ……198
12.5 小结 ……200
第13章 数据库访问 ……201
13.1 JDBC简介 ……201
13.1.1 基本JDBC操作 ……201
13.1.2 基本JDBC类 ……202
13.1.3 一个简单JDBC实例 ……204
13.2 JDBC驱动器 ……209
13.2.1 驱动器类型 ……209
13.2.2 JDBC-ODBC桥 ……209
13.2.3 注册一个驱动器 ……211
13.3 连接到一个数据库 ……212
13.4 语句接口 ……213
13.4.1 Statement ……214
13.4.2 PreparedStatement ……220
13.4.3 CallableStatement ……225
13.5 结果集 ……230
13.5.1 可滚动的结果集 ……233
13.5.2 可修改结果集 ……237
13.5.3 RowSet ……237
13.6 使用元数据 ……237
13.6.1 数据库元数据 ……238
13.6.2 ResultSetMetadata ……246
13.7 JDBC 2.0及以上版本中的新特性 ……247
13.8 小结 ……247
第14章 会话和线程管理 ……249
14.1 会话跟踪 ……249
14.1.1 隐藏域 ……250
14.1.2 URL重写 ……254
14.1.3 cookie ……256
14.2 会话API ……261
14.2.1 创建会话 ……261
14.2.2 从会话中保存和检索对象 ……263
14.2.3 销毁会话 ……264
14.2.4 修订后实例 ……264
14.2.5 会话捆绑侦听者 ……272
14.3 线程管理 ……278
14.4 servlet线程模型 ……289
14.4.1 缺省线程模型 ……289
14.4.2 单线程模型 ……290
14.5 多线程应用 ……291
14.6 应用考虑 ……295
14.7 小结 ……297
第15章 JSP和JavaBean ……298
15.1 JavaBean是什么 ……298
15.1.1 bean属性 ……298
15.1.2 持久性 ……300
15.2 JSP行为 ……305
15.2.1 <jsp:useBean> ……305
15.2.2 <jsp:setProperty> ……309
15.2.3 <jsp:getProperty> ……311
15.3 一个完整例子——带有bean的个性化风格 ……312
15.3.1 从Web得到天气数据 ……312
15.3.2 LyricNote入口 ……319
15.4 小结 ……323

精彩好书见www.book118.com

第16章 JSP和XML ……324
16.1 XML简介 ……324
16.1.1 XML解决的问题 ……325
16.1.2 XML语法 ……325
16.1.3 文档类型定义 ……326
16.2 XML解析器 ……328
16.2.1 文档对象模型 ……328
16.2.2 XML的简单API ……338
16.3 使用XSLT进行XSL转换 ……349
16.4 小结 ……352
第17章 JSP测试和调试 ……353
17.1 建立思想模型 ……353
17.2 独立测试 ……357
17.3 调试工具 ……358
17.3.1 捕获窗体参数 ……358
17.3.2 调试Web客户端 ……361
17.3.3 跟踪HTTP请求 ……367
17.4 小结 ……382
第18章 发布Web应用 ……383
18.1 Web应用环境 ……383
18.1.1 目录结构 ……383
18.1.2 资源映射 ……384
18.1.3 servlet上下文 ……386
18.2 Web存档文件 ……387
18.3 发布描述器：web.xml ……388
18.4 发布描述器示例 ……393
18.5 小结 ……395
第19章 事例分析：一个产品支持中心 ……396
19.1 过程流 ……396
19.2 数据模式 ……398
19.3 开发系统 ……398
19.4 模式－视图－控制器结构 ……399
19.4.1 模式类 ……400
19.4.2 视图类 ……464
19.4.3 控制器类 ……489
19.5 小结 ……502

第四部分 JSP和其他Web组件

第20章 与其他客户端进行通信 ……505
20.1 URL连接 ……505
20.1.1 URL类 ……505
20.1.2 URLConnection类 ……506
20.1.3 HttpURLConnection类 ……507
20.2 作为客户端的Java应用 ……508
20.2.1 JSP竞价服务器 ……508
20.2.2 竞价客户端应用 ……510
20.3 一个Java Applet客户端 ……512
20.3.1 Java插件 ……512
20.3.2 PriceQuoteApplet ……513
20.4 一个Perl 客户端 ……517
20.4.1 通用数据库选择服务器 ……518
20.4.2 Perl脚本 ……520
20.5 小结 ……523
第21章 与其他服务器通信 ……524
21.1 服务器端脚本环境 ……524
21.2 从一个JSP页面发送邮件 ……529
21.2.1 发送邮件的方法 ……529
21.2.2 在产品支持系统中的电子邮件通告 ……532
21.3 小结 ……535

第五部分 附 录

附录A servlet API版本2.3 ……537
附录B JSP API版本1.2 ……582
附录C HTTP参考 ……608

精彩好书见www.book118.com

第一部分 Web编程环境

第一部分对Web编程环境加以介绍，主要集中于商业和技术两方面，给出Web应用的演化过程，介绍了底层的客户端服务器体系结构和支持它的协议。最后介绍了JavaServer Page（JSP）。

第1章 Web 市 场

在罗马中心的太巴河附近坐落着罗马广场 两千年以前，罗马广场是罗马帝国的权利中心 这里是胜利凯旋之地，人们在这里交换货物和服务，新闻和观点是自由共享的（从中我们得出了单词“fornum”的一般含义） 虽然部分由石头、砖块和灰泥建成，但扩展它就使一种新技术成为可能：混凝土.

现在，Internet是全球电子市场．Internet正变成商业企业和私人顾客交换物品、服务和信息的主要中心。像罗马广场一样，由于技术的发展——新的计算机语言、网络标准的普及、低廉的硬件，使Internet的增长成为可能.

本书讲述JavaServer Page，一种将Web浏览器、Web服务器、数据库系统结合在一起，使其应用更容易开发、访问和发布的活跃技术。Java技术已证明在连接性、可靠性、可扩展性和安全性上是十分卓越的 此项技术比其他任何技术都更容易推动网络计算模型和全球电子市场的形成。

没有人可以准确预计出未来的趋势，即使是罗马广场最终也覆盖了草地，变成了il Campo Vaccino——牧场．但可以很保险的说，企业成功交换产品、服务、信息、思想的程度将一直取决于其接近市场的程度。

精彩好书见www.book118.com

第2章 Web应用演化

WWW最初并未设想为一个应用环境。然而今天，Web应用是Internet应用的基础——特别是电子商务的应用。本章追溯WWW、Web应用及相关技术的起源，为本书其余部分详细讲解各种技术打下铺垫。

2.1 Web的产生

WWW和其相关超文本传输协议（HTTP）1990年产生于欧洲粒子物理（CERN）实验室的工作。TimBerners-Lee将HTTP开发成一种发布文档的网络协议，并编写了第一个浏览器。在1991和1992年，此系统用于CERN和其他高能物理实验室及大学，并稳步流行起来。1993年，Mosaic浏览器的出现导致商业Web使用热潮。在5年时间里，全世界有超过650 000的Web服务器，并拥有上百万的用户。

2.2 Web编程模型的增长

将Web用做一种应用环境的思路已经发展很久了，技术的每一阶段都预示着新思路的广泛传播。第一个可操作模型只将Web服务用做文档请求服务。在此环境中，内容并不改变，除非是作者给出文档的一个新版本。客户端/服务器交互过程如图2-1所示。

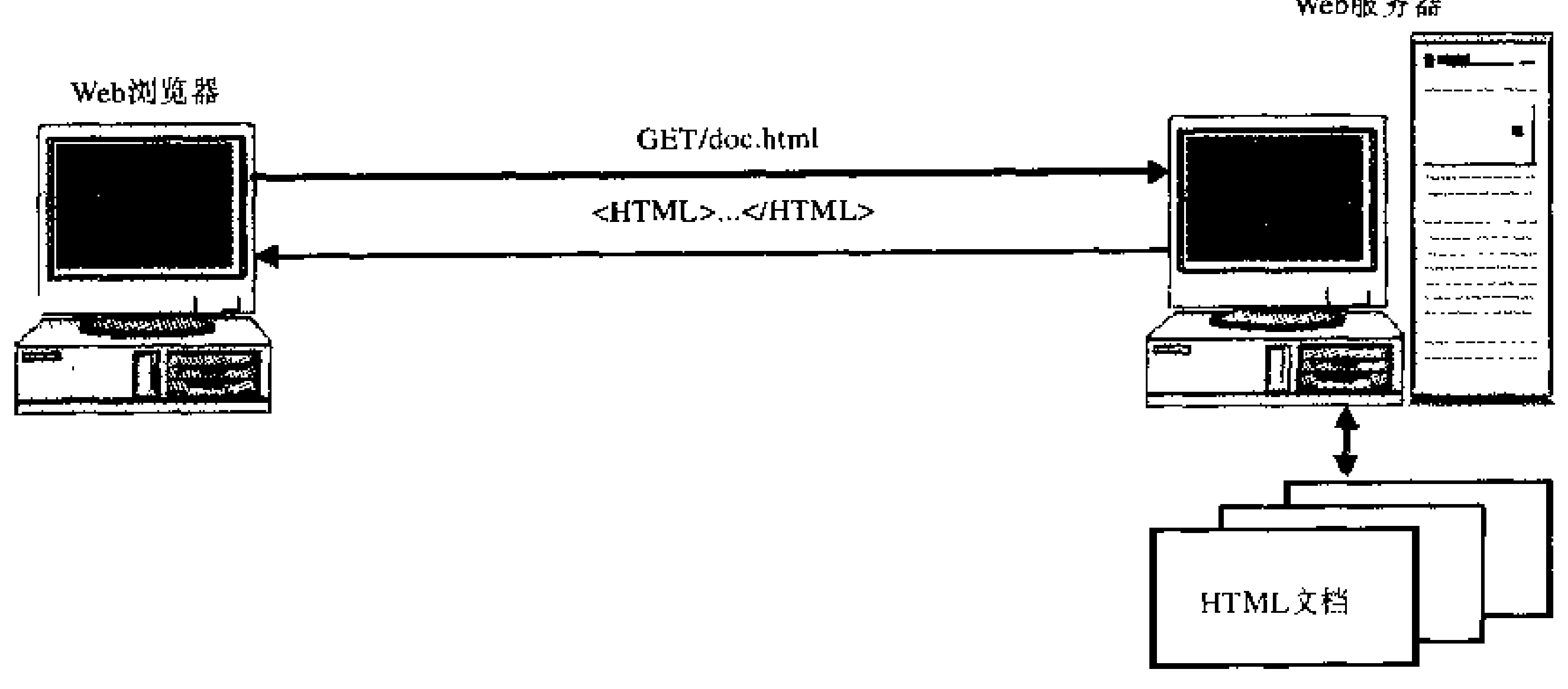


图2-1 静态文档服务模型

在此模型中HTTP是一个简单的请求/应答协议，浏览器请求一个文档（典型地使用一个GET命令），Web服务器以HTML数据流形式，在前面加上几个描述性头标返回文档。第3章详

精彩好书见www.book118.com

细解释HTTP。

很明显的是人们是否能够修改Web服务器处理的文档，就像Perl脚本这样的文本处理程序一样。Web浏览器不知道其间的差别，因为一个HTTP请求的结果仍然是一个HTML数据流。另外，浏览器可以发送多个请求——它可以发送参数，或是通过将其嵌入URL、或是通过发送一个带有请求的数据流，这表明HTTP请求可以被解释为数据库查询，查询结果可以被用于动态构建一个HTML文档。随着NCSA HTTPd的发展，Web服务器形成一种新的规范，命名为通用网关接口（CGI）。

一个CGI程序被Web服务器调用以响应各类请求。通常对特殊目录或文件名文档的请求带有特定扩展名，如.cgi。请求参数为关键字/取值对，请求头标为环境变量。程序读取这些参数和头标、手动执行应用任务（典型地访问一个数据库就是如此），然后生成一个HTTP响应。此响应被发回请求Web浏览器好像它是一个一般的静态文本。图2-2例示了过程流。

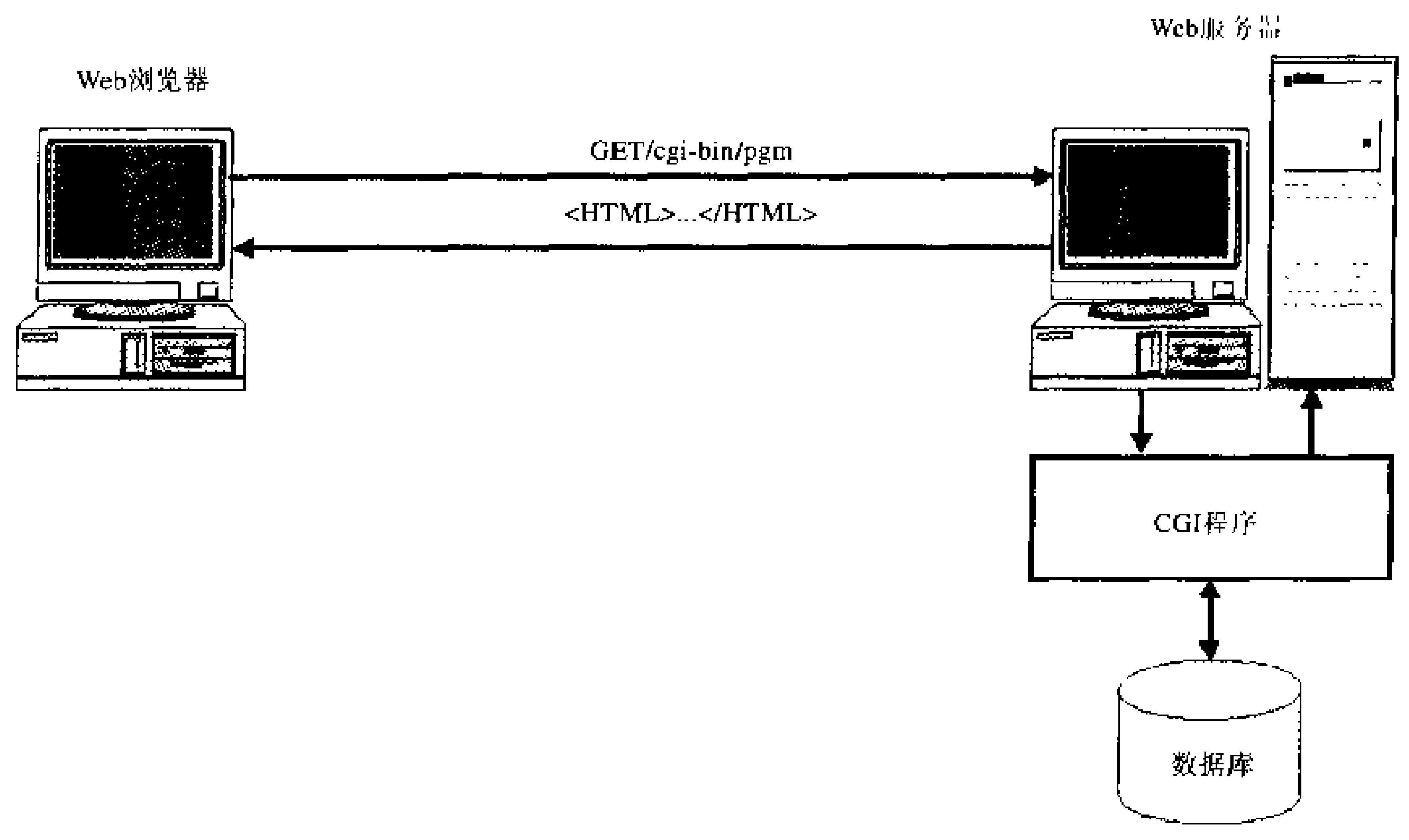


图2-2 CGI脚本生成动态内容

CGI很方便，但有一个最大的缺点。通常，CGI对每一HTTP请求产生一个新的进程[1]。当通信量很低时，这不会产生问题，但当通信级别增长时，就会造成大量的系统开销。由于此原因，通常CGI扩展性不好。

一个重大改进来自于Java Servlet API的1997版本以及随后产生的JavaServer Page（JSP）

1 对此有一些改进。如FastCGI，从一单一持续性进程中处理所有请求。

精彩好书见www.book118.com

API。这些相关技术将Java的强大功能带入了Web服务器，带有数据库连接、网络访问和多线程操作。值得注意的是，这是一个不同的处理模型。Servlet和JSP页面操作是保留在内存中的单一实例，使用多线程同步服务请求。如图2-3所示，servlet和JSP可以在高级、健壮应用中利用整个Java 2 企业版（J2EE）环境。

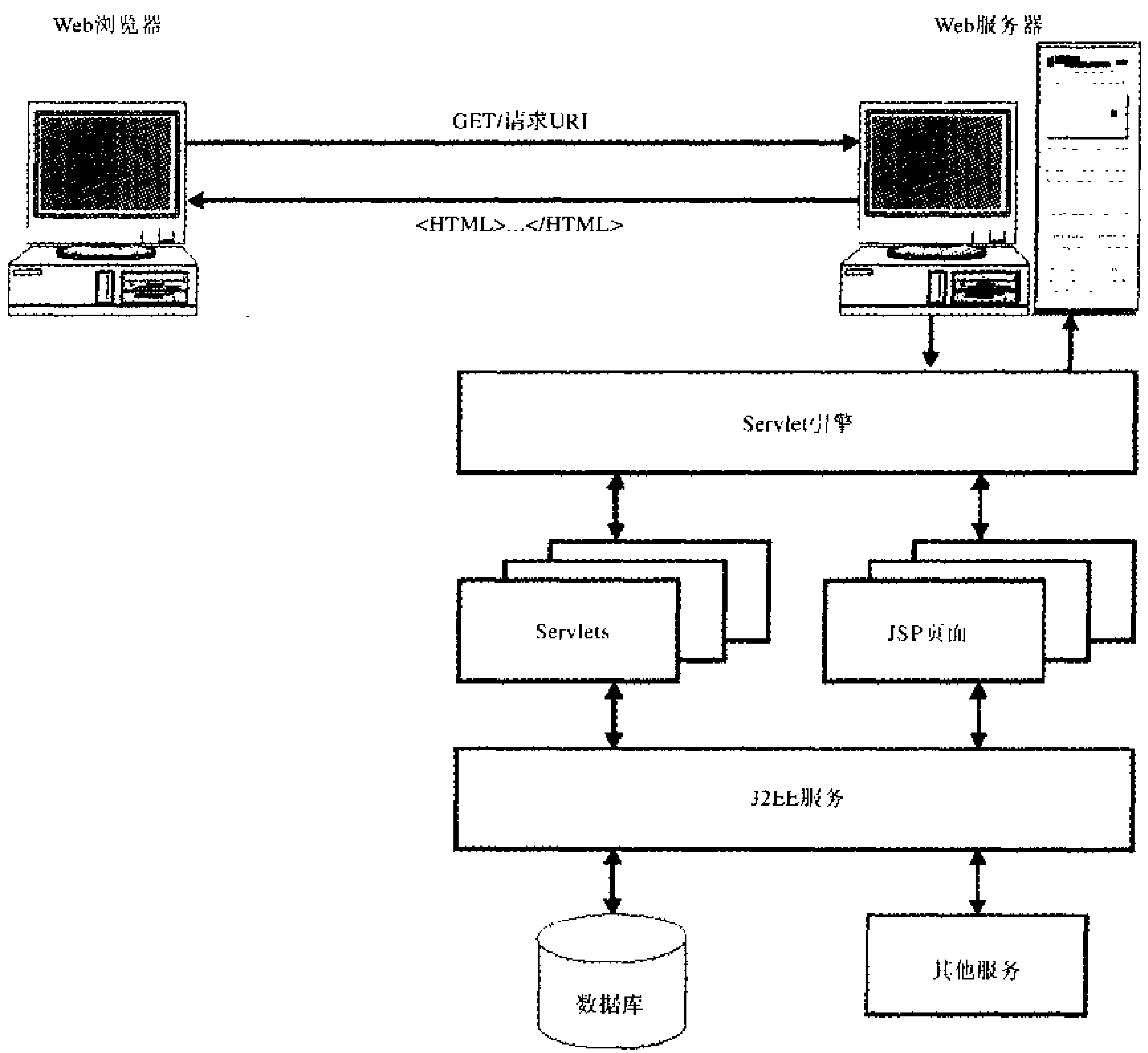


图2-3　servlets、JSP和J2EE的动态应用

2.3　从客户端移向服务器端方案

Web应用模型随着Web的成熟不断进化，每一阶段的经验都引导了下一步的需求。applet形式的客户端Java的第一波表面上很流行，但随着现实的介入有些令人失望。在浏览器之间发生的大量的不兼容，在慢速调制解调器上下载时间过长，限制applet使用的安全性约束。为此，

精彩好书见www.book118.com

applet发展很慢[1]，而服务器端Java则成为最大的发展领域。

服务器端Java没有applet环境的上述限制。不会发生浏览器的不一致性，因为浏览器并不需要带有一个Java虚拟机。浏览器只需要显示HTML，而这一点即使是最古老的浏览器也能够很好的实现。另外，不包含客户端设置及大的类文件下载。同样，安全性考虑也只受限于web服务器所处理的过程，这是典型地只在一个受控制的封闭环境中。

JSP证明是一种优秀的服务器端技术，是开发Web应用的好基础。本书其余章节将深入探讨JSP以证明这一点。

1 许多观察家相信客户端Java会和服务器端Java并驾齐驱。Java插件消除了浏览器的不兼容并允许使用Swing组件。另外，高速的Internet连接正在使下载因素变得越来越不重要。

精彩好书见www.book118.com

第3章 超文本传输协议介绍

本章介绍Web客户端/服务器模型使用的底层语言。HTTP发展出对理解Web编程环境重要的基本概念。这一章给出几个使用此语言进行通信的Web浏览器和服务器的实例。关于协议的附带详细内容可在附录C中找到。

3.1 HTTP是什么

超文本标记语言（HTML）是一种用来描述Web文档的内容的语言，超文本传输协议（HTTP）是一种用来描述如何在Internet上发送这些文档的语言。理解Web编程的关键是理解此协议和其操作环境。

3.1.1 Internet上请求文档的一种语言

HTTP指出了浏览器进行请求、服务器提供响应的规则。此规则集或协议，包含以下方式：

- 以名字请求一个文档。
- 在数据格式上达成一致。
- 判断用户是谁。
- 决定如何处理过期资源。
- 指出请求结果。

及其他有用函数。

HTTP像编写普通ASCII文本行一样由一组命令集组成。当使用一Web浏览器时，不必直接输入HTTP命令，而是当键入一个URL或点击一个超级链接时，浏览器将该行为转换为HTTP命令，此命令向URL中指定的服务器文档发出请求。Web服务器找到文档，将其发回至浏览器加以显示，并伴有其相关图形和其他超级链接。

3.1.2 HTTP规范

Internet标准通常在Internet工程任务制定组织（IETF）发布的RFC中指定。这些RFC被Internet研究发展机构广泛接受。因为它们是标准文档，故一般用正规语言编写，好像立法文档一样。这使得它们不适合做技术指南，但却是参考的重要标准。

RFC一旦被提出，就被编号且不会再改变。当一个标准被修改时，则给出一个新的RFC。作为标准，RFC在Internet上被广泛采用。一个优秀的在线资源是Brent Baccala的连接：An Internet Encyclopedia (http://www.freesoft.org/CIE)，它包含了大多数RFC的HTML版本，并提供一个全文搜索引擎。

处理HTTP的几个RFC为：

RFC 1945　HTTP版本1.0描述。

精彩好书见www.book118.com

RFC 2068 版本1.1的初步描述。

RFC 2616 1.1规范的修正版。

除非特别指定，本书使用RFC 2616中给出的HTTP 1.1标准。

3.2 HTTP请求模型

规范将HTTP作为无状态请求/响应协议加以描述，其基本操作如下：

1. 一个客户端应用，如Web浏览器打开至Web服务器的HTTP端口的一个套接字（缺省为80）。

2. 通过连接，客户端写一个ASCII文本请求行，后跟0或多个HTTP头标，一个空行和实现请求的任意数据。

3. Web服务器解析请求，定位指定资源。

4. 服务器将资源副本写至套接字，在此处由客户端加以读取。

5. 服务器关闭连接。

图3-1给出其基本操作。

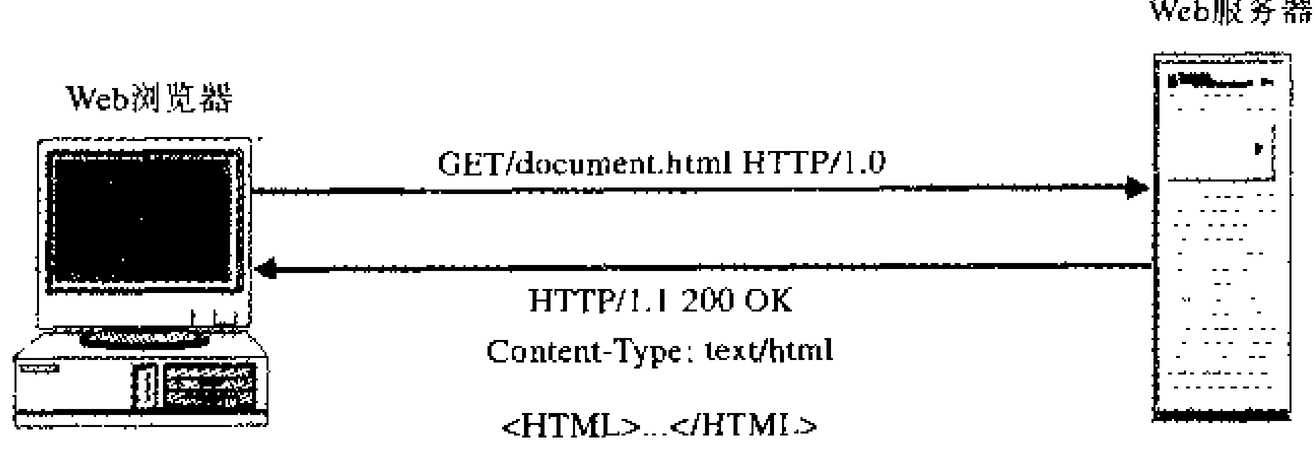


图3-1 HTTP基本操作

至关重要的一点为此模型是无状态的，表明在处理一个请求时，Web服务器并不记住来自同一客户端的前面请求的任意内容。协议是一个简单的请求（“请给我此文档”）和一个响应（“OK，给你”）。明显的，这对应用开发强加了限制，程序典型情况下需要大量的来回转换操作，复杂对象必须被初始化，并保持其状态。

此种方式需要服务器对一系列客户端请求所表示的会话设置一个标识，并要求客户端记住此标识，对每一次请求将其提供给服务器。此技术在第14章详细讲解。

下面用大量篇幅讨论其中的每一步骤。

3.2.1 连接至Web服务器

Web服务器操作侦听一特定的众所周知的端口号，通常为端口80上的请求，但也可以使用任意可用的端口。如果一个Web服务器侦听一不同的端口，指向此服务器的URL必须在服务器名称后紧跟一冒号和端口号，例如，

```
http://www.mycompany.com mypath.html
```

精彩好书见www.book118.com

指向运行于www.mycompany.com主机缺省端口80上的Web服务器的一个HTML文档。如果服务器运行于端口4311，则URL如下：

```
http://www.mycompany.com:4311/mypath.html
```

为什么还要这样麻烦的替代端口号呢，尤其是这样给出的URL语法很难看？因为这将允许多个服务器运行在单一主机上。一个具有不同功能的实验性Web服务器也许需要与主服务器共存。例如，Tomcat和JRun servlet引擎可以运行一个微型HTTP服务器以测试servlet和JSP页面。大多数Web服务器通过映射URL至不同的名空间提供隐藏这种替代语法的方式。

一个客户端，如一个Web浏览器，通过打开一个至Web服务器端口的TCP/IP套接字初始化一个HTTP请求，然后在套接字上打开输入和输出流。在Java术语中，这将等同于几行代码：

```
Socket socket   new Socket("www.mycompany.com", 80);
InputStream istream = socket.getInputStream();
OutputStream ostream = socket.getOutputStream();
```

需要打开套接字的参数是Web服务器主机名和端口号。服务器主机名从URL中抽取，而端口号或者为隐含，或者也从URL中抽取。输出流用于发送HTTP命令至Web服务器；输入流用于读取响应。

3.2.2 发送HTTP请求

一旦套接字连接完成，Web浏览器写入请求文档的一个HTTP命令。一个请求分4部分。

第一部分是请求行，由三个标记组成，用空格分隔：请求方法，请求URI和HTTP版本。下面给出典型的请求行：

```
GET /mypath.html HTTP/1.0
```

此例中请求方法是GET，URI为/mypath.html，HTTP版本为HTTP/1.0。

HTTP规范定义了8种可能的请求方法，在下表中列出。所有方法中，大部分请求使用GET或POST。这两种方法是本书惟一考虑的方法。

请求行第2个标记是请求同意资源标识符（Uniform Resource Identifter, URI）。它是文档或被请求的其他资源的URI。从实践出发，它指的是没有转发http://和主机名的URL。在例子http://www.mycompany.com/mypath.html中，请求URI是/mypath.html。

HTTP请求方法

方　法	描　　述
GET	检索URI中标识资源的一个简单请求
HEAD	与GET 方法相同。除了服务器并不返回请求文档。服务器只返回状态行和头标
POST	服务器接受被写入客户端输出流中数据的请求
PUT	服务器保存请求数据作为指定URI新内容的请求
DELETE	服务器删除URI中命名的资源的请求
OPTIONS	关于服务器支持的请求方法信息的请求
TRACE	Web服务器反馈HTTP请求和其头标的请求
CONNECT	已文档化但当前未实现的一个方法，预留做隧道代理

精彩好书见www.book118.com

请求行最后一个标记是HTTP版本。它表明客户端应用理解的HTTP规范的最高版本。允许值为HTTP/1.0和HTTP/1.1。

请求行之后是任意请求头标。它们是关键字/取值对，每行一对，关键字和取值用冒号（:）分隔。写完最后一个请求头标后，是一个空行，只发送回车符和退行。这将通知服务器以下不再有头标。即使不存在头标，也必须发送此空行，这样服务器不再查找其他头标。

请求头标进一步通知服务器关于客户端的功能和标识、典型的请求头标为：

User-Agent　客户端厂家和版本

Accept　客户端可识别的内容类型列表

Content-Length　附加到请求的数据字节数

请求和响应头标的完整列表见附录C。

对于HTTP POST 请求，请求可以包含数据。在后面12 章将看到如何使用POST传送HTML窗体域的取值。如果给出数据，最常使用的是Content-Type和Content-Length头标。

3.2.3 服务器接受请求

当客户端连接至Web服务器的侦听端口时，服务器接受连接并处理请求。在大多数情况下，是开始发送一个线程处理请求，这样就可以持续服务于多个请求。处理请求依据URI方式各有不同。如果URI表示一静态文档，服务器打开文档文件，准备将其内容拷备到客户端。如果URI是一程序名，如一个CGI脚本、servlet或JSP页面，并且服务器被配置可以处理这样的请求，服务器则准备调用程序或进程。

3.2.4 来自服务器的HTTP响应

无论服务器如何处理请求，结果都是一样的——一个HTTP响应。与一个请求类似，一个响应由4部分组成：状态行，0或多个响应头标，一个空行指明头标结束及组成请求的数据。

状态行由三个标记组成：

- HTTP版本。像客户端指明其可以理解的最高版本一样，服务器指出其能力。
- 响应代码。3位数字代码，指出请求成功或失败，如果失败，指出原因。HTTP状态码列表在附录C中可以找到。
- 一个可选响应描述。即为响应代码的可读性解释。

一个典型的HTTP响应状态行如下：

```
HTTP/1.0 200 OK
```

表明依据HTTP规范1.0级别对请求文档的成功检索。

状态行后面是响应头标，以空行做分隔符。像请求头标一样，它们指出服务器的功能，标识出响应数据的细节。附录C列出了有效的HTTP响应头标。

响应的最后一部分是所请求数据本身，典型的是一HTML文档或图像流。数据被发送后，服务器关闭其连接点。

精彩好书见www.book118.com

3.3 实例

下面看几个实例将更加清晰。当在浏览器地址栏键入一个URL或点击一个超级链接，一个GET请求的简单事件将会发生。如果打开URL http://www.lyricnote.com/simple.html，Web浏览器打开至主机www.lyricnote.com端口80的一个套接字连接，然后写入下列行：

```
GET /simple.html HTTP/1.0
```

接着是一个空行。Web服务器返回以下结果：

```
HTTP/1.1 200 OK
Date: Wed, 31 Jan 2001 03:55:43 GMT
Server: Apache/1.3.12 (Win32)
Content-Length: 241
Content-Type: text/html

<HTML>
<BODY>
<H3>Welcome</H3> to <b>The Lyric Note</b>,
the best Internet source for
<UL>
<LI>sheet music
<LI>musical instruments
<LI>books on musical topics
<LI>music software, and
<LI>musical gift items
</UL>
</BODY>
</HTML>
```

浏览器首先解析状态行，查看表明请求是否成功的状态代码。然后浏览器解析每一请求头标，头标告之以下为241个字节的HTML。浏览器读取HTML，依据HTML的语法和语义对其进行格式化，并在浏览器窗口中显示它，如图3-2所示：

一个HTML文档当被载入时可以包含对其他需要被载入的资源的引用。例如，图像经常被嵌入到具有HTML标签<img>的页面中。JavaScript文件或外部样式单也是需要的。Web浏览器（不是服务器）识别这些事件，对其他资源进行额外的请求。此过程循环多次。Web服务器并不读取其服务的整个HTML，识别出<img>标签，然后开始发送图像文件字节。Web服务器只是在一个操作中发回被请求的资源。如果几毫秒后浏览器请求了一个图像文件，服务器则在一单独操作中将之返回。Web浏览器在后台完成这一切，因此用户不知道几个请求被调用。

讨论前面的几个例子，假定打开http://www.lyricnote.com/compound.html，浏览器再次打开至www.lyricnote.com端口80的套接字连接，请求HTML文档：

```
GET /compound.html HTTP/1.0
```

响应结果如下：

```
HTTP/1.1 200 OK
Date: Tue, 30 Jan 2001 23:42:16 GMT
Server: Apache/1.3.12 (Win32)
```

精彩好书见www.book118.com


```
Content-Length: 380
Content-Type: text/html

<HTML>
<HEAD>
<LINK REL="stylesheet" HREF="lyricnote.css">
</HEAD>
<BODY>
<IMG SRC="images/logo.png">
<HR COLOR="#005A9C" ALIGN="LEFT" WIDTH="500">
<H3>Welcome</H3> to <b>The Lyric Note</b>,
the best Internet source for
<UL>
<LI>sheet music
<LI>musical instruments
<LI>books on musical topics
<LI>music software, and
<LI>musical gift items
</UL>
</BODY>
</HTML>
```

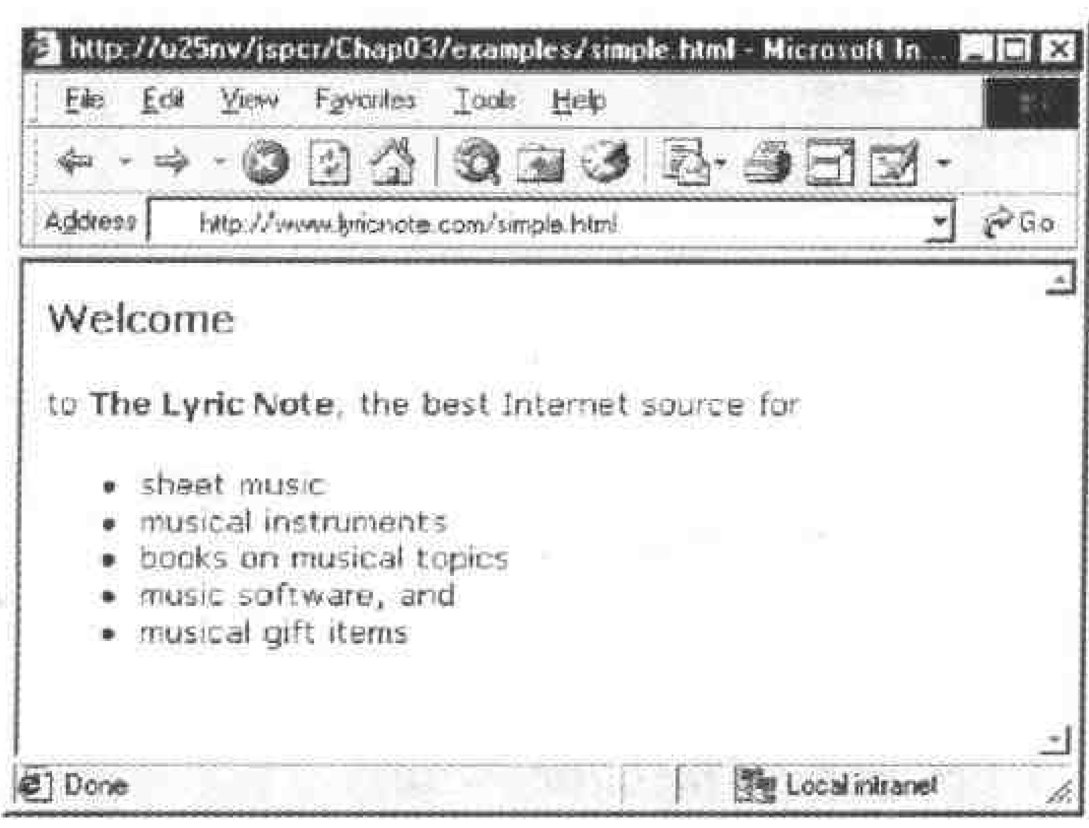


图3-2 简单的HTTP请求结果

当浏览器解析HTML时，它注意到样式单请求：

```
<LINK REL="stylesheet" HREF="lyricnote.css">
```

并进行第二次请求：

```
GET /lyricnote.css HTTP/1.0
```

Web服务器检索样式单，并将其返回至客户端。

精彩好书见www.book118.com


```
HTTP/1.1 200 OK
Date: Tue, 30 Jan 2001 23:42:27 GMT
Server: Apache/1.3.12 (Win32)
Connection: Keep-alive, close
Content-Length: 73
Content-Type: text/plain

h3 {
    font-size: 20px;
    font-weight: bold;
    color: #005A9C;
}
```

浏览器翻译该样式单，应用其字体大小，粗细和颜色样式到<H3>标签。接着，它遇到一个图像标签：

```
<IMG SRC="images/logo.png">
```

进行一次徽标请求：

```
GET /images/logo.png HTTP/1.0
```

使得Web服务器响应以图像数据流：

```
HTTP/1.1 200 OK
Date: Tue, 30 Jan 2001 23:42:44 GMT
Server: Apache/1.3.12 (Win32)
Connection: Keep-alive, close
Content-Length: 1280
Content-Type: text/plain

(Binary image data follows)
```

最后，浏览器完成整个页面，显示如图3-3所示：

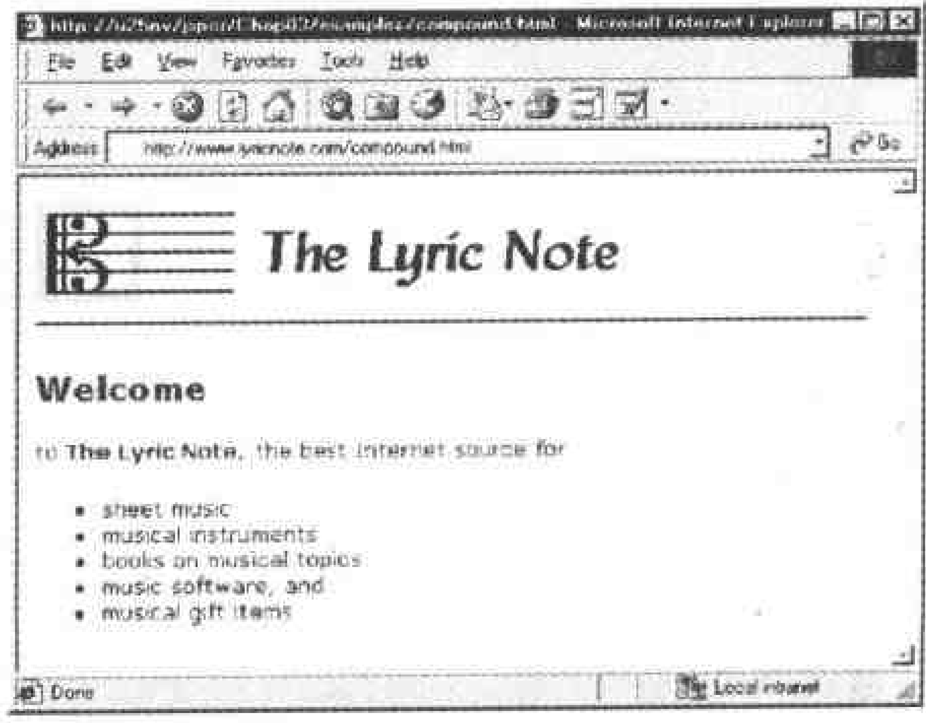


图3-3 对Compound请求结果

精彩好书见www.book118.com

3.4 小结

本章介绍HTTP中进行请求和返回响应的规则集。理解这些规则对正确的开发和发现并解决问题是至关重要的。重要的是明白HTTP是无状态的，即HTTP本身并不保留从一个请求到下一个请求的信息，JSP环境提供弥补这一点的正确方法。另外关键的一点是浏览器和服务器均可被功能类似的软件所替代；用其他语言编写的应用、applet程序可以作为客户端，诊断工具可以扮演服务器角色，因为它们所需做的就是提供浏览器和Web服务器使用的同样的HTTP请求和响应流。这些应用与实际所用的没有差别。在后面章节中会涉及此功能。

精彩好书见www.book118.com

第4章 servlet介绍

为理解JavaServer Page，有必要理解它们的底层技术Java servlet。servlet是通过动态生成Web主页扩展了Web服务器功能的Java类语言。一个称为servlet引擎的运行环境管理servlet的载入和载出，并结合Web服务器将请求导向servlet，并把输出发回Web客户端。

自从1997年问世以来，servlet已经成为服务器端Java编程的最主要环境，被广泛的使用于应用服务器。servlet有几个关键优点：

- 性能 原来的技术像CGI，典型情况是启动一个新进程处理每个进入请求。在Web只是学院和科学研究的知识库的年代，不会有很大的流量，此方法工作效果不错。相反，当第一次请求发送时，servlet被载入，并长久驻留在内存，servlet引擎装载servlet类的一个单一实例，并使用一个可利用的线程池向其发送请求。结果性能得到了显著提高。
- 简化 客户端Java applet运行于Web服务器提供的虚拟机上，这产生了兼容性问题，增加了复杂性，限制了applet提供的功能。servlet大大简化了这种情况，因为它们运行于在一受控服务器环境下的虚拟机中，只需要基本的HTTP与其客户端通信，不需要指定的客户端软件，即使旧的浏览器。
- HTTP会话 虽然HTTP服务器没有记住来自同一客户端以前请求内容的内嵌功能，servlet API提供了一个HttpSession类克服了此限制。
- 访问Java技术servlet作为Java应用，对Java的所有特性可以直接访问，如线程，网络访问和数据库连接。

JSP页面，自动被转换成servlet，也继承了所有这些优点。

本章阐述了servlet的工作方式，给出基本的servlet对象及其API。本章讨论servlet引擎，servlet生命期，servlet线程模式，servlet如何在请求之间维持一致状态。本章也包含一个servlet标注的实例。

4.1 servlet生命期

对应于客户端applet类似产品，servlet提供当在一个大的上下文中发生特定事件时调用的方法。此环境下编程包括编写预定义方法（有时称callback方法），它由一个管理程序在需要时调用。

例如，一个applet提供诸如init（）、start（）、paint（）、stop（）和destroy（）方法，由applet运行时环境响应用户的执行动作所调用。java.applet.Applet基类提供所有这些方法的缺省实现。你只需在所关心事件发生期间跳过这些方法。例如，如果需要创建GUI组件时，编写一个init（）方法。

类似地，servlet在由servlet引擎管理的一个请求和响应模型的上下文中执行操作。该引擎执行以下动作：

精彩好书见www.book118.com

- 当第一次被请求时载入一个servlet
- 调用servlet 的init（ ）方法
- 调用servlet的service（ ）方法处理任意数目的请求
- 当程序停止时，调用每个servlet的destroy（ ）方法

像applet一样，实现servlet调用方法有一些标准的基类javax.servlet.GenericServlet和javax.servlet.http.HttpServlet。servlet编程由这些类的子类组成，屏蔽了所需的方法以手工实现指定的任务。下面一节讲解每个servlet生命期中的方法。

4.1.1 init

当一个servlet的请求被servlet引擎收到后，它检查servlet是否被载入。如果未载入，servlet引擎使用类装载器得到所需的特定servlet类，然后调用其构造器取得servlet的一个实例。servlet被装载后，但在其服务于任何请求之前，servlet引擎以下列形式调用一个初始化方法：

```
public void init(ServletConfig config)
   throws ServletException
```

此方法只被调用一次，只是在servlet被放入服务之前。ServletConfig对象提供对servlet上下文（本章后面讨论）和servlet任意初始化参数代码的访问。为获得servlet上下文的引用，config对象必须被存储为一实例变量，此任务由GenericServlet中init（ServletConfig）方法实现。因此，在任意子类init（ ）方法中调用super.init(config)非常重要。

在init（ ）方法中，servlet可以执行任意必须的启动任务，如建立数据库连接。如果发生servlet不能处理请求的任意错误，应该产生溢出UnavailableException[1]。这将防止将请求导入servlet。

4.1.2 service

init（ ）方法成功完成后，servlet可以接受请求。缺省只有一个servlet实例被创建，servlet引擎在一单独线程中向实例发送每一请求。被调用的servlet方法如下：

```
public void service(
   ServletRequest request,
   ServletResponse response)
throws ServletException, IOException;
```

ServletRequest对象被servlet引擎创建，充做客户端和请求信息的包容器。它包含远程系统的标识请求参数和与请求相关的任意输入流。类似地，ServletResponse对象向servlet提供与其返回到初始请求器的结果进行通信的方式。它包含打开一个输出流的方法和指定内容类型和长度的方法。

尽管service（ ）方法很重要，但却很少被使用。原因是大部分servlet设计为在HTTP环境下

1 UnavailableException是可选地包含预计servlet不可利用的秒数的ServletException的子类。如果未指定，则假定servlet永久性不可利用。

精彩好书见www.book118.com

进行操作，在此环境下，有一个指定的javax.servlet.http包。大部分servlet并不直接扩展javax.servlet.GenericServlet，而是扩展其子类javax.servlet.http.HttpServlet。该子类提供与每一HTTP请求方法对应的指定方法：GET请求由doGet（ ）处理，POST请求由doPost（ ）处理等等。这些方法的符号使用请求和响应对象的HTTP指定版本：

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

HttpServlet中的service(Request,Response)方法把请求和响应对象置入其指定HTTP副本，然后调用service(HttpServletRequest，HttpServletResponse)，检验请求，并调用适当的doGet( )，doPost（ ）或其他方法。一个典型的HTTP servlet然后跳过这些补充方法中的一个或多个，而不是跳过service( )方法。

4.1.3 destroy

servlet规范允许servlet引擎在任意时刻卸载servlet。这样做以保存系统资源或准备停止servlet引擎。servlet引擎通过调用其destroy（ ）方法通知每一个即将发生的载入servlet。通过跳过destroy（ ）方法，可以释放在init（ ）期间分配的任意资源。

注意 *用户调用destroy（ ）实际上不会卸载servlet，只有servlet引擎可以实现此功能。*

4.2 例子：千米每公升到英里每加仑servlet

下面看一个简单的servlet。K2MServlet，如下所示，是一个创建燃油效率转换表的servlet，它根据英里每加仑表示出千米每公升。

```
package jspcr.servlets;

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

/**
* Prints a conversion table of miles per gallon
* to kilometers per liter
*/
public class K2MServlet extends HttpServlet
{
   private static final DecimalFormat FMT
      = new DecimalFormat("#0.00");

  private static final String PAGE_TOP = ""
      + "<HTML>"
```

精彩好书见www.book118.com


```
        + "<HEAD>"
        + "<TITLE>Fuel Efficiency Conversion Chart</TITLE>"
        + "</HEAD>"
        + "<BODY>"
        + "<H3>Fuel Efficiency Conversion Chart</H3>"
        + "<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>"
        + "<TR>"
        + "<TH>Kilometers per Liter</TH>"
        + "<TH>Miles per Gallon</TH>"
        + "</TR>"
        ;
  private static final String PAGE_BOTTOM = ""
        + "</TABLE>"
        + "</BODY>"
        + "</HTML>" ;

  public void doGet(
          HttpServletRequest request,
          HttpServletResponse response)
        throws ServletException, IOException
  {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println(PAGE_TOP);
        for (double kpl = 5; kpl <= 20; kpl += 1.0) {
          double mpg = kpl * 2.352146;
          out.println("<TR>");
          out.println("<TD>" + FMT.format(kpl) + "</TD>");
          out.println("<TD>" + FMT.format(mpg) + "</TD>");
          out.println("</TR>");
     }
     out.println(PAGE_BOTTOM);
   }
  }
```

开始，注意在程序头两个重要的语句：

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

这些语句向编译器标识这里使用的是从通用和指定HTTP servlet包中的类。import语句不是必须的，但使用它们可以在不指定全质名的情况下引用类。

下面是类声明：

```
public class K2MServlet extends HttpServlet
```

一个servlet需要实现javax.servlet.Servlet接口最小化状态。为简化servlet编写，servlet API提供了此接口，称为GenericServlet。它也提供指定HTTP子类HttpServlet，这是servlet最常用的

精彩好书见www.book118.com

基类。

```
public void doGet(
        HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, IOException
```

这里的servlet没有启动和终止行为的特定需求。因此只需屏蔽一个方法doGet（ ）。如果请求方法是GET，可从HttpServlet超类service（ ）方法中调用它。

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

在编写任何结果返回到客户端前，需要指定要发送的任意HTTP头标。这里惟一需要的是Content-Type，将其设置为text/html。

```
PrintWriter out = response.getWriter();
```

创建一个HTML页面包括编写与HTTP请求相关的一个输出流HTML语句。此输出流可使用getOutputStream（ ）或getWriter（ ）方法从response对象中获得。选取哪种方法取决于写入的是二进制流还是字符输出。重要的一点是servlet必须选择这些方法其中之一；但不能二者皆选。因为这里正在编写一般的HTML。这里使用getWriter（ ）获得一字符写入者。

其余的工作是打印HTML表格文本。方便起见，这里在静态字符串变量PAGE_TOP和PAGE_BOTTOM中将页面头和脚编码，在千米每公升的一定范围通过一个循环中打印表格本身。

```
out.println(PAGE_TOP);
for (double kpl = 5; kpl <= 20; kpl += 1.0) {
   double mpg = kpl * 2.352146;
   out.println("<TR>");
   out.println("<TD>" + FMT.format(kpl) + "</TD>");
   out.println("<TD>" + FMT.format(mpg) + "</TD>");
   out.println("</TR>");
}
out.println(PAGE_BOTTOM);
```

要运行此servlet，首先需要编译它。为保证编译成功，servlet API中的类必须在类路径下。这些类典型情况下在servlet引擎分布的一个JAR文件中可以找到。官方JAR文件也可以在Apache Jakarta Web站点http://jakarta.apache.org中找到。

接着，根据servlet引擎，需要在Web应用发布描述器/WEB-INF/web.xml中描述该servlet。对于一个简单servlet，描述器只包含一个<servlet> 标签，并带有其子<servlet-name>和<servlet-class>元素。这里描述器内容如下：

```
<?xml version="1.0" ?>
<web-app>
   ...
   <servlet>
      <servlet-name>K2M</servlet-name>
      <servlet-class>jspcr.servlets.K2MServlet</servlet-class>
   </servlet>
```

精彩好书见www.book118.com


```
    ...
</web-app>
```

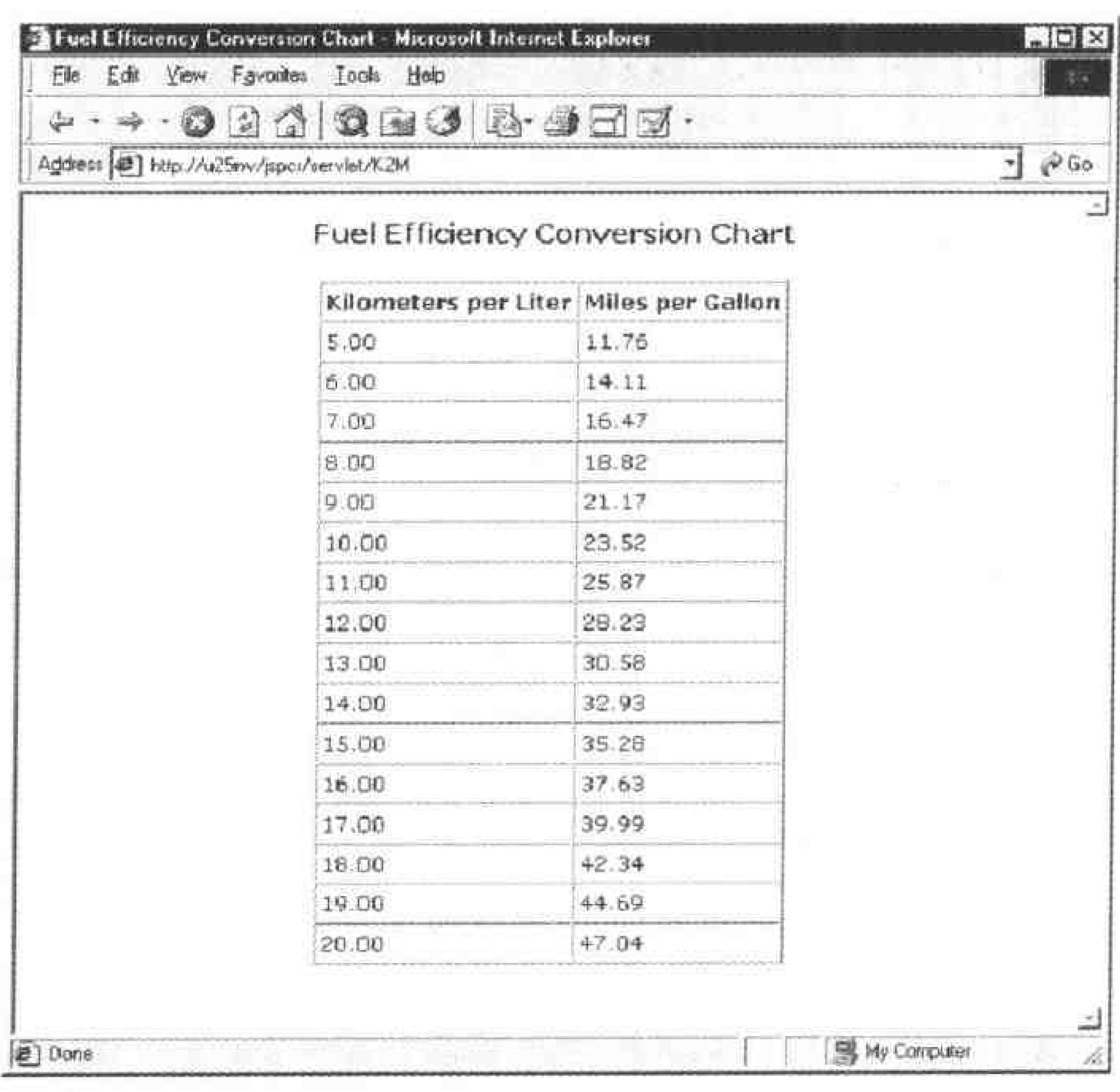
注意　web.xml中<servlet>表目必须在相对于其他元素的指定位置编码。细节请见第18章或参见web-app_2.2.DTD。

大多数情况下，修改web.xml文件需要在任何改变发生前重启servlet引擎。

最后，使用下列形式的URL调用此servlet：

```
http://<servername>/<webappname>/servlet/<servletname>
```

此servlet结果如下图所示：



Fuel Efficiency Conversion Chart

Kilometers per Liter	Miles per Gallon
5.00	11.76
6.00	14.11
7.00	16.47
8.00	18.82
9.00	21.17
10.00	23.52
11.00	25.87
12.00	28.23
13.00	30.58
14.00	32.93
15.00	35.28
16.00	37.63
17.00	39.99
18.00	42.34
19.00	44.69
20.00	47.04

千米每公升到英里每加仑的输出结果

4.3 servlet类

这一节概括javax. servlet和javax. servlet.http包中几个重要的类。servlet API的详细内容在附录A中可以找到。

精彩好书见www.book118.com

4.3.1 servlet

基本servlet抽象集是javax. servlet.Servlet接口，见表4-1。它规定了必须由servlet类实现，由servlet引擎识别和管理的方法集。其基本目标是提供生命期方法init（）、service（）和destroy（）。

servlet API提供Servlet接口的直接实现，称为GenericServlet，在表4-2中给出。此类提供除了service（）所有接口中方法的缺省实现。这意味着通过简单地扩展GenericServlet和编写一个定制service（）方法就可以编写一个基本的servlet。

表4-1　servlet接口中的方法

方　法	描　述
void init(ServletConfig config) throws ServletException	在servlet被载入后，并且在实施服务前由servlet引擎进行一次性调用。如果init( ) 产生溢出UnavailableException，则servlet退出服务。一个servlet应该提供保存config对象的某种方式以实现getServletConfig（）方法（见GenericServlet）
ServletConfig getServletConfig()	返回传递到servlet的init（）方法的ServletConfig对象
void service(ServletRequest request, ServletResponse response) throws ServletException, IOException	处理request对象中描述的请求，使用response对象返回请求结果
String getServletInfo()	返回描述servlet的一个字符串，趋向于需要提供人为可读性描述的管理工具所使用
void destroy()	当servlet将要卸载时由servlet引擎调用

表4-2　GenericServlet类中的方法

方　法	描　述
void destroy()	编写组成单词“destroy”的一个注册入口
String getInitParameter (String name)	返回具有指定名称的初始化参数值。通过调用config.getInitParameter(name)实现
Enumeration getInitParameter Names()	返回此servlet已编码的所有初始化参数的一个枚举类型值。调用config.getInit ParameterNames（）获得列表。如果未提供初始化参数，则返回一个空的枚举类型值（但不是null）
ServletConfig getServletConfig()	返回传递到init（）方法的ServletConfig对象
ServletContext getServletContext()	返回在config对象中引用的ServletContext
String getServletInfo()	返回空字符串（“”）
void init(ServletConfig config) throws ServletException	在一实例变量中保存config对象。编写组成单词“init”的注册入口，然后调用方法init（）
void init() throws ServletException	可以被跳过以处理servlet初始化。在config对象被保存后init（ServletConfig config）的结尾处自动被调用。servlet作者经常会忘记调用super.init（config）
void log(String msg,)	编写注册servlet的入口。为此调用servlet上下文的log（）方法。servlet的名字被加到消息文本的开头
void log(String msg, Throwable t)	编写一个入口和servlet注册的栈轨迹。此方法也是ServletContext中相应方法的一个副本

精彩好书见www.book118.com

（续）

方 法	描 述
abstract void service(Request request, Response response) throws ServletException, IoException	由servlet引擎调用为请求对象描述的请求提供服务。这是Generic Servlet中惟一的抽象方法。因此，它也是惟一必须被子类所覆盖的方法
String getServletName()	返回在Web应用发布描述器（web.xml）中指定的的servlet名字

除了Servlet接口，GenericServlet也实现ServletConfig，处理初始化参数和servlet上下文，提供对授权给传递到init（）中的ServletConfig对象的便利方法。

虽然servlet API允许扩展到其他协议，但当前版本只支持协议独立的servlet[1]和HTTP servlet。因为最终所有的servlet均在Web环境下实施操作，只有几个servlet直接扩展了GenericServlet。对servlet更一般的是扩展其指定HTTP子类HttpServlet。见表4-3。HTTP介绍见第3章。

HttpServlet通过调用指定到HTTP请求方法的方法实现service（）。亦即对于DELETE、HEAD、GET、OPTIONS、POST、PUT和TRACE，它分别调用doDelete（）、doHead（）、doGet（）、doOptions（）、doPost（）、doPut（）和doTrace（）。它也将这些方法使用的请求和响应对象置入其HTTP指定子类，本节后面会讲到这一点。

注意 处理GET、POST、PUT和DELETE 的方法缺省返回一个错误表明不支持请求方法，这样一个servlet需要覆盖其显示支持的这些方法。

表4-3 HttpServlet类中的方法

方 法	描 述
void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException	由servlet引擎调用处理一个HTTP GET请求。输入参数、HTTP头标和输入流（如果存在）可从request对象、response头标和response对象的输出流中获得
void doPost（HttpServletRequest request, HttpServletResponse response） throws ServletException, IOException	由servlet引擎调用处理一个HTTP POST请求。从获得参数、输入数据或返回响应的观点看，与doGet（）没有差别
void doPut（HttpServletRequest request, HttpServletResponse response） throws ServletException, IOException	由servlet引擎调用处理一个HTTP PUT请求。本方法中请求URI指出被载入的文件位置
void doDelete(HttpServletRequest request, HttpServletResponse response） throws ServletException, IOException	由servlet引擎调用处理一个HTTP DELETE请求，请求URI指出资源被删除

1 什么是协议无关的servlet？也许是一个根本不对请求提供服务的servlet，而只是简单地从其init（）方法启动后台线程并在destroy（）中杀死它们。它可用于模拟Windows NT和Unix的端口监督[控]进程。

精彩好书见www.book118.com

（续）

方 法	描 述
Void doOptions(HttpservletRequest request, HttpservletResponse response) throws ServletException, IOException	由Servlet引擎调用一个HTTP OPTIONS请求 返回一个Allow响应头标表明此servlet支持的HTTP方法。一个servlet不需要覆盖此方法，因为HttpServlet方法已经实现规范所需的功能
void doTrace(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException	由servlet引擎调用处理一个HTTP TRACE请求。使得请求头标被反馈成响应头标。一个servlet不需要覆盖此方法，因为HttpServlet方法已经实现HTTP规范所需的功能
void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException	service (Request request ,Response response)调用的一个立即方法，带有指定HTTP请求和响应。此方法实际上将请求导向doGet（）、doPost（）等等。不应该覆盖此方法
void service (Request request , Response response) throws ServletException, IOException	将请求和响应对象置入其指定HTTP子类，并调用指定HTTP的service（）方法

4.3.2 servlet请求

ServletResquest接口封装了客户端请求的细节。当前存在的通用版本是与协议无关的，并有一个指定HTTP的子接口。

协议无关版本见表4-4，具有方法：

- 找到客户端的主机名和IP地址
- 检索请求参数
- 取得和设置属性
- 取得输入和输出流

表4-4 ServletRequest类的方法

方 法	描 述
Object getAttribute (String name)	返回具有指定名字的请求属性，如果不存在则返回null。属性可由servlet引擎设置或使用setAttribute（）显式加入。setAttribute（）在连接一个RequestDispatcher对象时很有用
Enumeration getAttributeNames()	返回请求中所有属性名的枚举。如果不存在属性，则返回一个空的枚举
String getCharacterEncoding()	返回请求使用的字符编码
int getContentLength()	指定输入流的长度 如果未知，则返回 – 1
ServletInputStream getInputStream() throws IOException	返回与请求相关的（二进制）输入流。可以调用getInputStream（）或getReader（）方法之一，但不可同时调用

精彩好书见www.book118.com

（续）

方 法	描 述
String getParameter (String name)	返回指定输入参数，如果不存在，返回null
Enumeration getParameterNames()	返回请求中所有参数名的一个可能为空的枚举类型
String [] getParameterValues (String name)	返回指定输入参数名的取值数组，如果取值不存在，则返回null。它在参数具有多个取值的情况下十分有用（例如，HTTP检查框元素）
String getProtocol ()	返回请求使用协议的名称和版本
String getScheme()	返回请求URL的子串，但不包含第一个冒号前内容（例如，http）
String getServerName ()	返回处理请求的服务器的主机名
int getServerPort ()	返回接收主机正在侦听的端口号
BufferedReader getReader () throws IOException	返回与请求相关输入数据的一个字符解读器。此方法和getInputStream（）只可分别调用，不能同时使用
String getRemoteAddr ()	返回客户端主机的数字型IP地址
String getRemoteHost ()	如果知道，返回客户端主机名
void setAttribute (String name ,Object obj)	以指定名称保存请求中指定对象的引用
void remoteAttribute (String name)	从请求中删除指定属性
Locale getLocale()	如果已知，返回客户端第一现场，或者为null
Enumeration getLocales()	如果已知，返回客户端第一现场的一个枚举类型，否则返回服务器第一现场
boolean isSecure ()	如果请求使用了如HTTPS安全隧道，返回true
RequestDispatcher getRequestDispatcher (String name)	返回指定源名称的RequestDispatcher对象。关于请求发送的细节见第8章

HttpServletRequest子接口（见表4-5）加入方法以便处理

- 读取和写入HTTP头标
- 取得和设置cookies
- 取得路径信息
- 标识HTTP会话

表4-5 HttpServletRequest接口的方法

方 法	描 述
String getAuthType ()	如果servlet由一个鉴定方案所保护，如HTTP基本鉴定，则返回方案名称
String getContextPath ()	返回指定servlet上下文（web应用）的URL的前缀
Cookie [] getCookies ()	返回与请求相关cookie的一个数组
long getDateHeader (String name)	将输出转换成适合构建Date对象的long类型取值的getHeader（）的简化版
String getHeader (String name)	返回指定的HTTP头标值。如果其由请求给出，则名字应为大小写不敏感

精彩好书见www.book118.com

（续）

方　法	描　述
Enumeration getHeaderNames ()	返回请求给出的所有HTTP头标名称的枚举类型值
Enumeration getHeaders (String name)	返回请求给出的指定类型的所有HTTP头标的名称的枚举类型值。它对具有多取值的头标非常有用
int getIntHeader(String name)	将输出转换为int取值的getHeader（ ）的简化版
String getMethod ()	返回HTTP请求方法（例如GET、POST等等）
String getPathInfo ()	返回在URL中指定的任意附加路径信息
String getPathTranslated ()	返回在URL中指定的任意附加路径信息，被转换成一个实际路径
String getQueryString ()	返回查询字符串——即URL中"?"后面的部分
String getRemoteUser ()	如果用户通过鉴定，返回远程用户名，否则为null
String getRequestedSessionId ()	返回客户端返回的会话ID
String getRequestURI ()	返回URL中一部分，从"/"开始，包括上下文，但不包括任意查询字符串
String getServletPath ()	返回请求URI上下文后的子串
HttpSession getSession ()	调用getSession（true）的简便方法
HttpSession getSession (boolean create)	返回当前HTTP会话，如果不存在，则创建一个新的会话，create参数为true
Principal getPrincipal ()	如果用户通过鉴定，返回代表当前用户的java.security.Principal对象，否则为null
boolean isRequestedSessionIdFromCookie ()	如果请求的会话ID由一个Cookie对象提供，则返回true，否则为false
boolean isRequestedSessionIdFromURL ()	如果请求的会话ID在请求URL中解码，返回true，否则为false
boolean isRequestedSessionIdValid()	如果客户端返回的会话ID仍然有效，则返回true
boolean isUserInRole (String role)	如果当前已通过鉴定用户与指定角色相关，则返回true，如果不是或用户未通过鉴定，则返回false

4.3.3 servlet响应

servlet response对象的函数将一个servlet生成的结果传送到发出请求的客户端。ServletResponse操作主要是作为输出流及其内容类型和长度的包容器。它由servlet引擎创建，作为service（ ）方法的第2个参数被传递到servlet。

像servlet请求一样，servlet响应同时具有真正协议无关的类和一个指定HTTP版本。表4-6给出在通用版本中可利用的方法。

表4-6　ServletResponse接口的方法

方　法	描　述
void flushBuffer（ ） throws IOException	发送缓存到客户端的输出内容。因为HTTP需要头标在内容前被发送，调用此方法发送状态行和响应头标，以确认请求
int getBufferSize（ ）	返回响应使用的缓存大小。如果缓存无效，则返回0
String getCharacterEncoding（ ）	返回响应使用字符解码的名字。除非显示设置，否则为ISO-8859-1
Locale getLocale（ ）	返回响应使用的现场。除非用setLocale（ ）修改，否则缺省为服务器现场
OutputStream getOutputStream（ ） throws IOException	返回用于将返回的二进制输出写入客户端的流。此方法和getWriter（ ）方法二者只能调用其一

精彩好书见www.book118.com

（续）

方法	描述
Writer getWriter ( ) throws IOException	返回用于将返回的文本输出写入客户端的一个字符写入器。此方法和getOutputStream ( ) 二者只能调用其一
boolean isCommitted ( )	如果状态和响应头标已经被发回客户端，则返回true，在响应被确认后发送响应头标毫无作用
void reset ( )	清除输出缓存及任何响应头标。如果响应已得到确认，则引发事件IllegalStateException
void setBufferSize (int nBytes)	设置响应的最小缓存大小。实际缓存大小可以更大，可通过调用getBufferSize ( ) 得到。如果输出已被写入，则产生溢出IllegalStateException
void setContentLength (int length)	设置内容体的长度
void setContentType (String type)	设置内容类型。在HTTP servlet中即设置Content-Type头标
void setLocale (Locale locale)	设置响应使用的现场。在HTTP servlet中，将对Content-Type头标取值产生影响

指定HTTP的子接口HttpServletResponse加入表示状态码、状态信息和响应头标的方法（附录C详细描述HTTP响应头标）。例如用来发送cookie或将用户重定向到另一URL。此接口还负责对URL中写入一Web页面的HTTP会话ID进行解码。表4-7给出HttpServletResponse中的方法。

表4-7 HttpServletResponse接口的方法

方法	描述
void addCookie (Cookie cookie)	将一个Set-Cookie头标加入到响应
void addDateHeader (String name , long date) void setDateHeader (String name, long date)	使用指定日期值加入带有指定名字（或代换所有此名字头标）的响应头标的方法。长整型日期值适合于java.util.Date(long time)构造器
Void setHeader (String name, String value)	设置具有指定名字和取值的一个响应头标
void addIntHeader (String name , int value) void setIntHeader (String name , int value)	使用指定整型值加入带有指定名字的响应头标（或代换此名字所有头标）
boolean containsHeader (String name)	如果响应已包含此名字的头标，则返回true
String encodeRedirectURL (String url) String encodeURL (String url)	如果客户端不知道接受cookie，则向URL加入会话ID。第一种形式只对在sendRedirect ( ) 中使用的URL进行调用。其他被编码的URLs应被传递到encodeURL ( )
void sendError (int status) void sendError (int status , String msg)	设置响应状态码为指定值（可选的状态信息）。HttpServletResponse定义了一个完整的整数常量集合表示有效状态值
void setStatus (int status)	设置响应状态码为指定值。只应用于不产生错误的响应。而错误响应使用sendError ( )

精彩好书见www.book118.com

除了附加方法外，HttpServletResponse也定义了每一可能HTTP响应代码的整型常量。

4.3.4 servlet上下文

一个servlet上下文是servlet引擎提供用来服务于Web应用的接口。Web应用中servlet可以使用servlet上下文得到：

- 在调用期间保存和检索属性的功能，并与其他servlet共享这些属性。
- 读取Web应用中文件内容和其他静态资源的功能。
- 互相发送请求的方式。
- 记录错误和信息化消息的功能。

servlet上下文具有名字（它属于Web应用的名字），惟一映射到文件系统的一个目录。

一个servlet可以通过调用传递到init（）的ServletConfig对象的getServletContext方法得到servlet上下文的引用。如果servlet直接或间接调用子类GenericServlet，它可以使用继承的简化方法getServetContext（）[1]。

表4-8概括了ServletContext给出的方法。

表4-8 ServletContext接口的方法

方法	描述
Object getAttribute (String name) void setAttribute (String name, Object obj)	返回servlet上下文中具有指定名字的对象，或使用已指定名捆绑一个对象。从Web应用的标准观点看，这样的对象是全局对象，因为它们可以被同一servlet在另一时刻访问。或上下文中任意其他servlet访问
Enumeration getAttributeNames ()	返回保存在servlet上下文中所有属性名字的枚举值
ServletContext getContext (String uripath)	返回映射到另一URI的servlet上下文。在同一服务器上，URI必须是以"/"开头的绝对路径
String getInitParameter (String name)	返回指定上下文范围的初始化参数值。此方法与ServletConfig方法名称不一样。后者只应用于已编码的指定servlet。此方法应用于上下文中所有servlet的参数
Enumeration getInitParameterNames ()	返回（可能为空）指定上下文范围的初始化参数值名字的枚举值
int getMajorVersion () int getMinorVersion ()	返回此上下文中支持servlet API级别的最大和最小版本号
String getMimeType (String fileName)	返回指定文件名的MIME类型。典型情况是基于文件扩展名。而不是文件本身的内容（它可以不必存在）。如果MIME类型未知，可以返回null
RequestDispatcher getNameDispatcher (String name) RequestDispatcher getRequestDispatcher (String path)	返回具有指定名字或路径的servlet或JSP的RequestDispatcher。如果不能创建RequestDispatcher，返回null。如果指定路径，必须以"/"开头，并且是相对于servlet上下文的顶部

[1] 此JSP页面可以更容易，servlet上下文的一个引用可以被自动保存在隐含变量application中。

精彩好书见www.book118.com

（续）

方 法	描 述
String getRealPath (String path)	给定一个URI，返回文件系统中URI对应的绝对路径。如果不能进行映射，返回null
URL getResource(String path) InputStream getResourceAsStream (String path)	返回相对于servlet上下文或读取URL的输入流的指定绝对路径相对应的URL，如果这样的资源不存在，则返回null
String getServerInfo( )	返回servlet引擎的名称和版本号
void log(String message) void log(String message, Throwable t)	将一个消息写入servlet注册，如果给出Throwable参数，则包含栈轨迹
void removeAttribute (String name)	从servlet上下文中删除指定属性

4.4 线程模型

缺省情况，servlet引擎只载入一个servlet的单一实例，servlet予以服务的多个请求运行于单线程下，但却共享同样的实例以及同样的实例变量。这说明了几点，最值得注意的是实例变量不是线程安全的。例如，对于以下servlet：

```
package jspcr.servlets;

import java.io.*;
import java.sql.*;
import java.util.*;

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

/**
* Bad example! Don't try this at home.
*/
public class ColliderServlet extends HttpServlet
{
    private Connection con;

    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException
    {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con = DriverManager.getConnection("jdbc:odbc:usda");
            // ... run some lengthy database operation here
        }
```

精彩好书见www.book118.com


```
        catch (Exception e) {
            throw new ServletException(e.getMessage());
        }
    }
}
```

考虑两个请求到达同一线程，只间隔几百毫秒，这时会发生什么？第一个请求打开数据库连接，保存其引用至con实例变量，然后它使用连接执行一个表格修改或其他数据库操作。同时，第二个请求到达，打开另一连接，在同一个con实例变量中保存其引用。如果第一个操作完成，试图做另一个数据库操作，其不再拥有初始连接对象——它只知道第2个。然后当其试图使用第2个连接时就会发生错误。

实例变量同样类型的问题可能发生在调用其服务方法中其他方法的servlet中。如果这些方法试图访问服务方法中创建的已被保存在一个实例变量中的servlet请求、响应或任意对象，没有任何方式可以确保另一线程内的请求由于将其自身对象的引用保存到变量中而不打断该变量的连续性[1]。

最安全的方式是不仅使用实例变量，还要在服务方法中定义的局部变量。

单线程模型

虽然单实例多线程模型为缺省模型，一个servlet也可以通过实现SingleThreadModel改变此行为。此接口不包含方法，通知servlet引擎应该创建一个实例池，并分配每个进入请求到其自己的实例和线程。这将确保两个被同一实例处理的请求不会在其服务方法执行中被打断。实例变量一次只能被一个请求所影响。进而实现线程安全。注意，因为可能存在多实例，所以没有什么方式可以阻止它们在不同的线程中并发执行。如果它们访问像文件或数据库这样的外部资源，仍会产生矛盾。在SingleThreadModel中有几种方式可以解决用其他方式不能很好解决的问题。

4.5 HTTP会话

虽然导航一个Web页面有点像在客户端和服务器之间进行会话，大部分情况下并不是这样。典型情况，一个Web客户端请求一个HTML文档，其由服务器定位并传送回客户端。如果图像链接在HTML中，客户端（如果它是一个Web浏览器）为每一幅图像向服务器发出额外的请求。如果用户点击页面内的超级链接，客户端为此发出新的HTTP请求，但所有动作都是一次发出一个请求，在请求之间，服务器发生转移处理其他请求，并不会记得第一个客户端。不会发生命令和数据的来回交换。只有一个请求接着一个响应，然后断开链接[2]。

对静态文档的基本下载，这已经足够了。然而，像购物卡或循环搜索引擎需要保留与特殊客户端相关服务器上的动态对象。它可以接受这样的对象之间几个请求。这种情况下，需要跟

1 此特定问题可通过使用保存所有感兴趣对象的一个私有类，然后传递此类作为附加方法的一个参数（基本是一个数据结构）来解决。

2 HTTP 1.1提供了将连接持续几秒钟的方式。因此，例如HTML和相关图像可被有效地下载。这需要服务器和客户端均需要知道该方式并显式请求它。然而请求/响应协议本身都是一样的。

精彩好书见www.book118.com

踪对象所属的客户端行为。

解决此问题有几种方法。其中大部分是在服务器上包含对象本身，将其设置为要求客户端记住的惟一关键字。在每个与请求相关的结果中，客户端传回此关键字，使得服务器可以重新创建上下文。

这与电话预定机票有点类似。机票代理请求顾客的名字、地址、飞行信息，并将这些内容输入一个设置确认号码的应用数据入口，将确认号码返回给客户。如果后来客户需要收回或改变预定信息，可以给出确认好的号码，进而允许机票代理访问和修改原始记录。

如何使得客户端在需要时记住和给出关键字？存在以下几种方式：

- Cookie　服务器可以在其初始响应中发送一个Set-Cookie头标，并将会话ID作为cookie取值[1]。在随后结果中，客户端可以返回带有Cookie头标的取值。然而，某一用户可能选择关闭此浏览器的cookie功能，所以此技术并不确保有效。
- 将会话ID附加到URL　对于一个动态过程创建的Web页面中的超级链接，会话ID可以作为URL中一个请求参数被编码。这不需要使用cookie，但它需要每一个被点击的URL都被如此解码。如果没注意其中一个（这很容易发生），则会话链接失效。
- 隐藏域　如果应用由一系列使用submit按钮进行导航的HTML窗体组成，会话ID可被存储为一隐藏域，并通过request.getParameter（ ）进行检索。显然，只有窗体均为动态生成的时候，此方法才有效。

HttpSession接口

servlet API提供了一个与这些技术相关的简便的包容器称为HTTP Session。一个名为javax.servlet.http.HttpSession类似于哈希表的接口提供了setAttribute（ ）和getAttribute（ ）方法通过名字存储和检索对象。HttpSession提供了一个会话ID关键字，一个参与会话行为的客户端在同一会话中后面的请求中存储和返回它。servlet引擎查找适当的会话对象，并使之对当前请求可用。表4-9给出HttpSession中可利用的方法。

表4-9　HttpSession接口中的方法

方　法	描　述
Object getAttribute (String name) void setAttribute (String name , Object value) void removeAttribute (String name)	将会话中一个对象保存为指定名字，返回或删除前面保存的此名称对象
Enumeration getAttributeNames ()	返回捆绑到当前会话的所有属性名的枚举值
long getCreationTime() long getLastAccessedTime()	返回表示会话创建和最后访问日期和时间的一个长整型。该整型形式为java.util.Date（ ）构造器中使用的形式

1　Cookie是一个Web服务器发送的具有指定生命期的名称/取值对。客户端浏览器保存cookie并在每次进入浏览器从相同的域中请求一个页面时自动将其返回到服务器。cookie的细节可在RFC 2109规范中找到。

精彩好书见www.book118.com

（续）

方 法	描 述
String getId()	返回会话ID，servlet引擎设置的一个惟一关键字
int getMaxInactiveInterval()	如果没有与客户端发生交互，设置和返回会话存活的最大秒数
void setMaxInactiveInterval (int seconds)	
void invalidate()	使得会话被终止，释放其中任意对象
boolean isNew()	如果客户端仍未加入到会话，返回true。当会话首次被创建，会话ID被传入客户端，但客户端仍未进行包含此会话ID的第二次请求时，返回true

此API还提供一个HttpSessionBindingListener接口。实现此接口的对象必须提供valueBound( )和valueUnbound ( ) 方法，它们当对象加入到一个HttpSession或从中删除时被调用。

4.6 小结

Java servlet是Web服务器的扩展，允许Web内容在响应一个客户端请求时动态创建。它们由servlet引擎管理，servlet引擎将其载入并初始化，再将其传入许多请求提供服务。最后卸载它们。servlet在其他服务器端编程环境上有一些关键的优点：

- 因为它们是常驻内存，并可同时运行在多线程中，所以性能更佳。
- 因为除了一个Web浏览器之外不需要其他客户端软件，所以更简单。
- 会话跟踪。
- 访问Java技术，包括线程、网络和数据库连接。

servlet操作于一固定的生命周期中，提供servlet引擎进行初始化，处理请求和终止请求的回馈方法。其API提供两种线程模型：缺省为单实例运行多线程以及单线程模型。

在servlet API中基本类和接口为：

- Servlet接口，给出必须实现的回馈方法的描述。
- GenericServlet，实现Servlet接口方法的一个基类。
- HttpServlet，GenericServlet的一个指定HTTP子类。
- ServletRequest，封装客户端请求信息。
- ServletResponse，提供对返回到客户端结果输出流的访问。
- ServletContext接口，允许一组servlet在一个Web应用进行互操作。

Servlet是JSP页面的底层技术。理解它们对形成在JSP环境中开发和调试所需的思维模式是至关重要的。

精彩好书见www.book118.com

第5章 JSP 介 绍

JavaServer page（JSP）是使用Java代码动态生成HTML文档的Web页面模板。JSP运行于服务器端组件，称为JSP容器，它将JSP转换成等价的Java servlet。

正因为这样，servlet和JSP页面最终是相关的。虽然每项技术有其自身的发展，其中一个大部分也可能在另一个中。因为也是servlet，JSP页面具有servlet的所有优点：

- 比CGI具有更好的性能和扩展性，因为它们是常驻内存和多线程的。
- 不需要指定的客户端设置。
- 对HTTP会话提供嵌入式支持。使应用编写更容易。
- 对Java技术具有完全访问——网络式编程、线程和数据库连接——没有客户端applet的限制。

但除此之外，JSP页面还有其自身优点：

- 需要时自动重新编译。
- 因为它们存在于一般Web服务器文档空间中，定位JSP比定位servlet更加容易。
- 因为JSP是类似HTML的，因此与Web开发工具具有更大的兼容性。

本章将JSP作为一种服务器端脚本环境给出其简介，描述了JSP容器操作及一个完整实例，这里只给出了基本内容，第二部分第6章会对其进行深入讲解。

5.1 JSP工作方式

一个JSP页面存在三种形式：

- JSP源码　这是开发者实际编写的形式。它存在于一个文本文件中，扩展名为.jsp，由HTML模板代码、Java语言声明和JSP伪指令及描述如何生成Web页面对一特殊请求提供服务的动作混合而成。
- Java源码　JSP容器在需要时将JSP源码转换成等价的Java servlet源码。此源码典型情况保存在一个工作区，对调试很有帮助。
- 已编译Java类　与其他任意Java类相似，生成的servlet代码在.class文件中被编译成字节码，准备好被载入和执行。

JSP容器基于每个文件的时间戳自动管理每种形式的JSP页面。在一个HTTP请求的响应中，容器检查自从上次.java源码被编译后，.jsp源文件是否被修改。如果是，容器重新将JSP源转换成Java源并再次编译它。

图5-1给出JSP容器的工作过程。当对一个JSP页面的请求发出后，容器首先判断与.jsp文件对应的类的名字，如果该类不存在或比.jsp文件的老（意味着JSP源自从上次被编译后已经改变），然后容器为一个等价的servlet创建Java源码并编译它。如果servlet实例并未运行，容器载入该servlet类并创建一个实例。最后，容器发送一个线程在载入的实例中处理当前HTTP请求。

精彩好书见www.book118.com

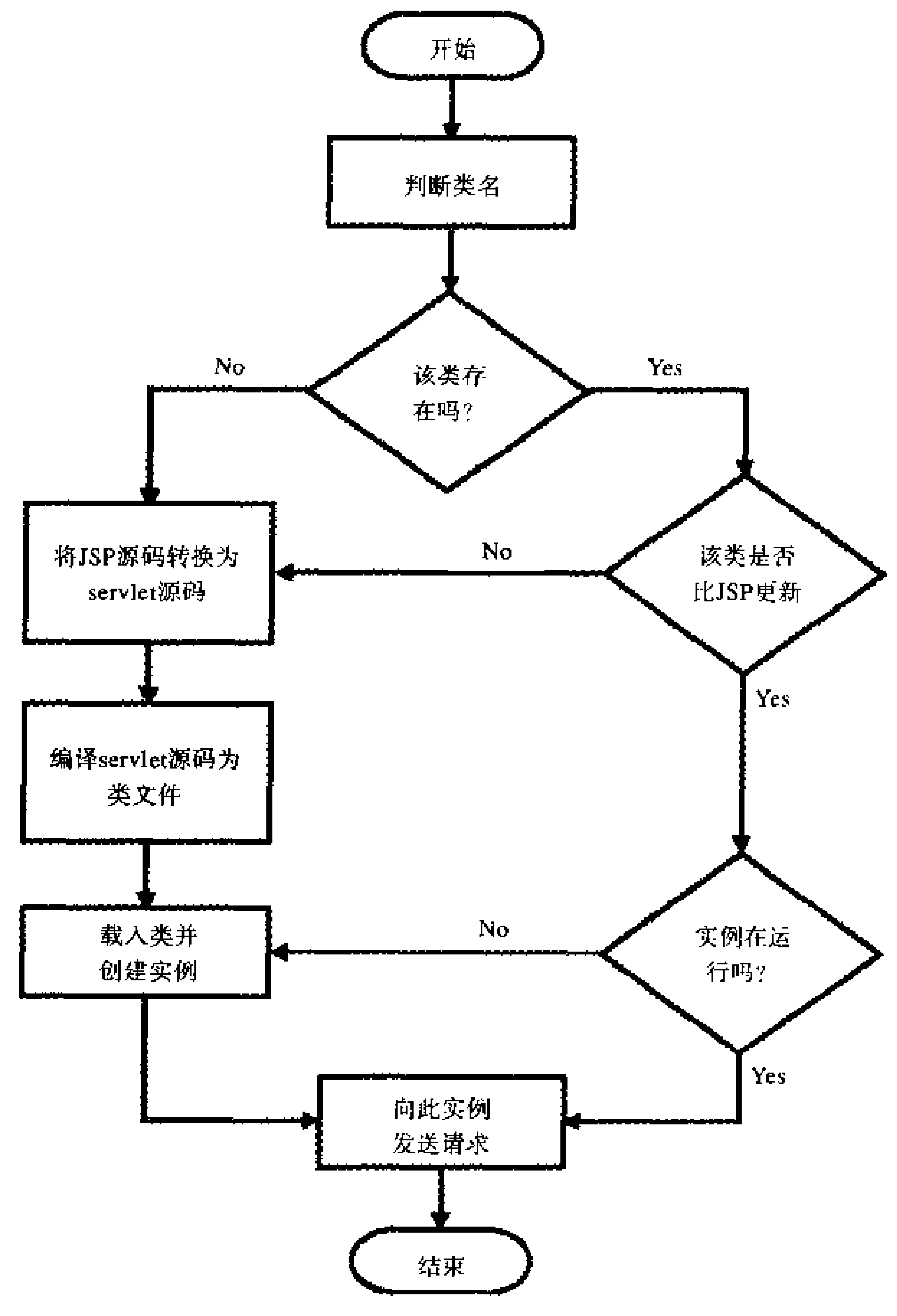


图5-1 JSP用来管理JSP转换的逻辑图

5.2 一个基本例子

为了说明JSP工作方式，下面使用与前一章相同的例子——将千米每公升转换成英里每加仑。以下为JSP页面：

```
<%@ page session="false" %>
<%@ page import="java.io.*,java.text.*,java.util.*" %>
<%-- Prints a conversion table of miles per gallon
```

精彩好书见www.book118.com


```
      to kilometers per liter --%>
<%!
   private static final DecimalFormat FMT
      = new DecimalFormat("#0.00");
%>
<HTML>
<HEAD>
<TITLE>Fuel Efficiency Conversion Chart</TITLE>
</HEAD>
<BODY>
<H3>Fuel Efficiency Conversion Chart</H3>
<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
<TR>
<TH>Kilometers per Liter</TH>
<TH>Miles per Gallon</TH>
</TR>
<%
   for (double kpl = 5; kpl <= 20; kpl += 1.0) {
      double mpg = kpl * 2.352146;
%>
<TR>
   <TD><%= FMT.format(kpl)%></TD>
   <TD><%= FMT.format(mpg)%></TD>
</TR>
<%
   }
%>
</TABLE>
</BODY>
</HTML>
```

将这段代码与第4章K2MServlet比较，首先注意的是JSP比较短——33行，而servlet有55行。另外，JSP看起来更像一个Web页面。大部分HTML像一般HTML一样是可识别的，对Java程序员也是如此。很明显，这里有某类循环，在循环内产生了表的各行。最后，特定字符集出现标记Java代码和HTML模板数据之间的边界。如果不明白其含义没有关系——在第6、7、8章会进行全面讲述。

如果从一个Web浏览器中调用此JSP页面，会看到如图5-2显示的表，正好与第4章产生的表一模一样。

为了使JSP到servlet的关系更加清晰，看一下由JSP容器产生的.java源码。此代码随着使用容器和其采纳的实现方法将有很大差别。以下代码列出JRun 3.0生成的源码（重新格式化以增加可读性）。

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
```

精彩好书见www.book118.com

Fuel Efficiency Conversion Chart

Kilometers per Liter	Miles per Gallon
5.00	11.76
6.00	14.11
7.00	16.47
8.00	18.82
9.00	21.17
10.00	23.52
11.00	25.87
12.00	28.23
13.00	30.58
14.00	32.93
15.00	35.28
16.00	37.63
17.00	39.99
18.00	42.34
19.00	44.69
20.00	47.04

图5-2 JSP页面千米每公斤到英里每加仑的输出结果

```
import javax.servlet.jsp.tagext.*;
import allaire.jrun.jsp.JRunJSPStaticHelpers;
import java.io.*;
import java.text.*;
import java.util.*;
public class jrun__Chap05__examples__K2M2ejsp18
   extends allaire.jrun.jsp.HttpJSPServlet
   implements allaire.jrun.jsp.JRunJspPage
{
   private ServletConfig config;
   private ServletContext application;
   private Object page = this;
   private JspFactory __jspFactory
      = JspFactory.getDefaultFactory();

public void _jspService(
      HttpServletRequest request,
      HttpServletResponse response)
   throws ServletException, java.io.IOException
{
   if (config == null) {
      config = getServletConfig();
      application = config.getServletContext();
   }
```

精彩好书见www.book118.com


```
    response.setContentType("text/html; charset=ISO-8859-1");
    PageContext pageContext = __jspFactory.getPageContext
       (this, request, response, null, false, 8192, true);
    JspWriter out = pageContext.getOut();

    try {
       out.print("\r\n<HTML>\r\n"
          + "<HEAD>\r\n"
          + "<TITLE>Fuel Efficiency Conversion Chart</TITLE>\r\n"
          + "</HEAD>\r\n"
          + "<BODY>\r\n"
          + "<H3>Fuel Efficiency Conversion Chart</H3>\r\n"
          + "<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>\r\n"
          + "<TR>\r\n<TH>Kilometers per Liter</TH>\r\n"
          + "<TH>Miles per Gallon</TH>\r\n</TR>\r\n");

    for (double kpl = 5; kpl <= 20; kpl += 1.0) {
       double mpg = kpl * 2.352146;
       out.print("\r\n<TR>\r\n <TD>");
       out.print(FMT.format(kpl));
       out.print("</TD>\r\n <TD>");
       out.print(FMT.format(mpg));
       out.print("</TD>\r\n</TR>\r\n");
    }
    out.print("\r\n</TABLE>\r\n</BODY>\r\n</HTML>\r\n");
  }
  catch (Throwable t) {
    if (t instanceof ServletException)
       throw (ServletException) t;
    if (t instanceof java.io.IOException)
       throw (java.io.IOException) t;
    if (t instanceof RuntimeException)
       throw (RuntimeException) t;
    throw JRunJSPStaticHelpers.handleException
       (t, pageContext);
  } finally {
    __jspFactory.releasePageContext(pageContext);
  }
}
private static final DecimalFormat FMT
   = new DecimalFormat("#0.00");
private static final String[] __dependencies__
   = {"/Chap05/examples/K2M.jsp",null};
private static final long[] __times__ = {980969842306L,0L};
public String[] __getDependencies()
{
   return __dependencies__;
}
```

精彩好书见www.book118.com


```
public long[] __getLastModifiedTimes()
{
    return __times__;
}
public int __getTranslationVersion()
{
        return 13;
    }
}
```

代码看上去有点机械，好像它是由一个计算机程序生成的（实际上是），但作为一个servlet仍是可识别的，特别是中间部分，与第4章K2MServlet源码有一点差别。

正如所见，建立此过程的思维模式是成功进行JSP开发和调试的关键。有了此基础，下面进入第二部分，深入探讨JSP元素。

精彩好书见www.book118.com

第二部分 JSP 元 素

下面6章讲述JSP语法和语义，给出创建代码所必须的技巧。内容包括基本语法、scriptlet、表达式、声明、文件包含、请求发送和指定页面行为。第二部分结束章节详细讲解JSP定制标签的内容。

第6章 JSP语法和语义

本章目的是要介绍在JavaServer Page中使用的基本组件，描述如何编写，解释其功能。本章首先给出JSP开发模型，然后介绍每个JSP元素、如何在整体设计中使用它们，最后解释每个元素使用方法并带有注释的例子。本章重点如下：

- 语法 用于表示元素的代码结构，以便JSP编译器可以识别它们。
- 语义 JSP容器中元素的含义——使用它们时发生的行为。

本章给出的每个JSP元素在第二部分其他章节中会详细阐述。

6.1 JSP开发模型

第5章提到JSP页面存在三种形式：

1) .jsp源文件，包含HTML语句和JSP元素。

2) servlet程序的Java源码。

3) 已编译的Java类。

为了理解JSP元素操作方式，如何创建这三种对象以及它们之间关系的思维模型是很重要的。首先，JSP开发人员编写一个.jsp源文件，将其保存在Web服务器或Web应用的文档文件系统的某处。从此观点讲，.jsp源文件与一般的HTML文件没有差别。其获得网络位置的URL是相同的，只是其文件名以.jsp而不是 .html结束。接着，当.jsp URL第一次被调用时，JSP容器读取此.jsp文件，解析其内容，生成等价的Java servlet源码。然后编译此servlet，创建一个.class文件。最后，JSP容器载入此servlet类，使用它服务于HTTP请求。中间一步（生成servlet源码）只有当.jsp文件已被修改时才会针对后续请求重复执行。

按此设计，JSP元素在下面两个操作步骤中会影响到JSP容器的操作方式：

- 转换时间 从一个.jsp文件生成Java servlet源码。
- 请求时间 调用servlet处理一个HTTP请求。

记住这个模型对理解JSP页面的语法单元和其功能很有帮助。

精彩好书见www.book118.com

6.2 JSP页面组件

.jsp文件可以包含JSP元素、固定模板数据或二者的任意结合。JSP元素指示JSP容器生成什么代码及其操作方式。这些元素有特定的开始和结束标记，使JSP编译器可对其识别。模板数据是JSP容器不可识别的所有其他代码。模板数据（通常为HTML）不做修改地加以传递，这样最后生成的HTML就会像在.jsp文件中编码中一样准确地包含模板数据。

JSP元素有以下3种：

- 伪指令
- 脚本元素、包含表达式、scriptlet和声明
- 动作

下面详细讲解每一个元素。

6.2.1 伪指令

伪指令是指示JSP容器生成什么代码的命令。其通用格式为：

```
<%@ directive-name [attribute="value" attribute="value" ...] %>
```

在“<%@”标签开始，“%>”标签之前可放置零或多个空格、制表符和换行字符。在伪指令名之后，在属性/取值对之间可以有一个或多个空格。惟一的限制是<%@开始标签必须与%>结束标签在同一物理文件中。

JSP 1.1规范描述了三种标准伪指令，并在JSP环境中都是兼容的。

- page
- include
- taglib

虽然规范声明在JSP 1.1环境中不能使用定制伪指令，但在后续规范中有可能包含用户自定义伪指令。

下面三节分别介绍这三种伪指令。

1. page伪指令

page伪指令用于指定整体JSP页面的属性。语法格式如下：

```
<%@ page [attribute="value" attribute="value" ...] %>
```

表6-1列出了其属性。

表6-1 page伪指令属性

属 性	取 值
language	在scriptlet、表达式和声明中使用的语言。在JSP 1.1中，此属性惟一有效值为java
extends	此JSP页面超类的全质名。它必须为实现HttpJspPage接口的类。JSP规范对不完全理解其隐意的情况下使用此属性提出警告
import	用逗号分隔列出一个或多个package.*名和/或全质类名。此列表用于在生成的Java servlet中创建相应的导入语句。以下包是自动包含的，不必被指出： java.lang.*

精彩好书见www.book118.com

(续)

属 性	取 值
	java.servlet.* java.servlet.jsp.* java.servlet.http.*
session	true 或false。指明JSP页面是否需要一个HTTP会话，如果为true，那么产生的servlet将包含创建一个HTTP会话（或访问一个HTTP会话，如果其已存在）的代码，缺省为true
buffer	指定输出缓存的大小。有效值是nnnkb或没有。这里，nnn为所分配缓存的kb字节数。缺省为8kb
autoflush	当缓存已满时被自动刷新，此值为true；产生缓存溢出时（异常部分抛出），则为false。缺省为true
isThreadSafe	如果页面可以处理来自多个线程的同步请求，则此值为true，如果不能，则为false。如果是false，生成的servlet表明它实现的是SingleThreadModel接口
info	页面getServletInfo（）方法返回的一个字符串
isErrorPage	如果此页面被用做另一个JSP的错误页面，则为true。这种情况下，页面可被指定为另一页面page伪指令中errorPage属性的取值。指定此属性为true，将使得exception隐含变量对此页面可用。缺省为false
errorPage	指定将被调用处理任意捕获溢出的另一个JSP页面的URL。其他JSP页面必须在其page伪指令中指定isErrorPage= “true”
contentType	指定将在生成servlet中使用的MIME类型和可选字符解码

一个文件中可以有多个page伪指令，其属性可集中指定应用于整个文件。但一个属性只能指定一次，import属性除外。

第10章深入讲解page伪指令。

2. include伪指令

include伪指令在转换时将另一文件的内容并入.jsp源输入流，与一个#include C预处理器伪指令非常类似。语法为：

```
<%@ include file="filename" %>
```

这里filename是依据当前servlet上下文解释的一个绝对或相对路径。例子如下：

```
<%@ include file="/header.html" %>
<%@ include file="/doc/legal/disclaimer.html" %>
<%@ include file="sortmethod" %>
```

include伪指令可与本章后面描述的<jsp:include>行为相对照，<jsp:include>行为生成请求时将另一个文件的输出合并到响应输出流中。其元素可被用于include标准头标和脚注或JSP页面中其他通用文本。第8章对这两种方法进行详细阐述。

3. taglib伪指令

taglib伪指令通过使用一个标签库，在当前页面中启用定制行为。该语法如下：

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

其属性在下面列出：

属 性	取 值
tagLibraryURI	一个标签库描述器的URL
tagPrefix	用于标识在页面后面部分使用定制标签的惟一前缀。

精彩好书见www.book118.com

例如，如果使用下面伪指令：

```
<%@ taglib uri="/tlds/FancyTableGenerator.tld" prefix="ft" %>
```

并且如果FancyTableGenerator.tld定义了一个名为table的标签，那么页面可以包含下述类型的标签：

```
<ft:table>
...
</ft:table>
```

JSP标签扩展在第11章详细描述。

6.2.2 注释

JSP规范在一个JSP页面中包含注释有两种方式：一种是只在JSP页面中可视的隐藏注释，一种是由页面生成的HTML或XML中包含的注释。前者类型语法为：

```
<%-- This is a hidden JSP comment --%>
```

后者为：

```
<!-- This is included in the generated HTML -->
```

当JSP编译器遇到一个JSP注释的开始标签<%--，它忽略文件中从这一点开始的所有内容直到找到相匹配的结束标签--%>。这表明JSP注释可用于屏蔽（或注释掉）JSP页面片段。这是一种暂时能屏蔽一个程序的部分而不会对源码做主要改动的永久性技术。另外，这也意味着JSP页面不能嵌套，因为一个内部注释的结束标签将被解释为外部注释的结尾标记。

另一种注释类型使用普通HTML或XML注释标签。此类注释经过响应输出流不会改变，被包含在生成的HTML中。它们在浏览器窗口中是可视的，但可通过调用View Source菜单选项查看。

如果一个注释的用途是使人们容易看懂程序，第二种类型没有第一种有用，原因有两个：它只在程序生成的HTML中看到，编程人员一般不会看到。然而，因为这些HTML注释是计算机生成的，它们可能不包含版本号、日期和其他在发现并修理程序故障里对技术支持人员有用的标识号。例如，一个JSP页面中包含下面3行：

```
<!--
Remote address was <%= request.getRemoteAddr() %>
-->
```

将会记录用户进行Web请求的远程地址而不会与输出混淆。如果应用某处出错，技术支持人员可指导用户查看生成的HTML源，指出标识出错的数据。

6.2.3 表达式

JSP提供一种简单方法访问可用的Java取值或其他表达式，并生成页面中HTML取值。语法为：

```
<%= exp %>
```

精彩好书见www.book118.com

这里exp为任意有效Java表达式。该表达式可以有任意数据值，只要它可以转换为一个字符串。这种转换通常通过生成out.print（ ）语句实现。例如，如下JSP代码：

```
The current time is <%= new java.util.Date() %>
```

生成servlet代码：

```
out.write("The current time is ");
out.print( new java.util.Date() );
out.write("\r\n");
```

提示 理解生成的代码可能会有助于提醒自己不要将分号放在表达式中。

第7章详细讨论表达式。

6.2.4 scriptlet

scriptlet是用于处理HTTP请求的一个或多个Java语句的集合。scriptlet语法如下：

```
<% statement; [statement; ...] %>
```

JSP编译器在_jspService（ ）方法主体中只简单不修改包含scriptlet内容。JSP页面可以包含任意数目的scriptlet。如果存在多个scriptlet，则每一个都附加到_jspService（ ）方法中，并按其编号排序。因此，一个scriptlet可以包含被括在大括号中的另一个scriptlet。考虑下面JSP页面，这是一个华氏温度到摄氏温度的转换表：

```
<%@ page import="java.text.*" %>
<TABLE BORDER=0 CELLPADDING=3>
<TR>
   <TH>Degrees<BR>Fahrenheit</TH>
   <TH>Degrees<BR>Celsius</TH>
</TR>
<%
   NumberFormat fmt = new DecimalFormat("###.000");
   for (int f = 32; f <= 212; f += 20) {
      double c = ((f - 32) * 5) / 9.0;
      String cs = fmt.format(c);
%>
   <TR>
      <TD ALIGN="RIGHT"><%= f %></TD>
      <TD ALIGN="RIGHT"><%= cs %></TD>
   </TR>
<%
   }
%>
</TABLE>
```

实例代码包含两个scriptlet：一个对应循环主体，一个对应大括号。在两个scriptlet之间是单一表格行的HTML标记，使用JSP表达式访问其取值。生成的servlet代码将scriptlet内容转换，其

精彩好书见www.book118.com

代码如下：

```
NumberFormat fmt = new DecimalFormat("###.000");
for (int f = 32; f <= 212; f += 20) {
   double c = ((f - 32) * 5) / 9.0;
   String cs = fmt.format(c);
   out.write("\r\n<TR>\r\n<TD ALIGN=\"RIGHT\">");
   out.print( f );
   out.write("</TD>\r\n");
   out.write("\r\n<TD ALIGN=\"RIGHT\">");
   out.print( cs );
   out.write("</TD>\r\n");
   out.write("</TR>\r\n");
}
```

输出如下：

华氏度	摄氏度
32	.000
52	11.111
72	22.222
92	33.333
112	44.444
132	55.556
152	66.667
172	77.778
192	88.889
212	100.000

第7章再进一步讲解scriptlet。

6.2.5 声明

像scriptlet一样，声明也包含Java语言语句，但有一点重要区别是：scriptlet代码成了_jspService ( ) 方法的一部分，而声明代码却是在_jspService ( ) 方法之外与生成的源文件合成一体。一个声明段的语法为：

```
<%! statement; [statement; ...] %>
```

声明段可用于声明类或实例变量、方法或内部类。与scriptlet不同，它们不可访问在下一段中描述的隐含对象。如果使用一个声明段声明一个需要使用请求对象的方法，则需要将对象作为参数传递给方法。

下面给出一个使用了声明段的JSP实例：

```
<%@ page
      errorPage="ErrorPage.jsp"
      import="java.io.*,java.util.*"
%>
```

精彩好书见www.book118.com


```
<%
     Enumeration enames;
     Map map;
     String title;

     // Print the request headers

     map = new TreeMap();
     enames = request.getHeaderNames();
     while (enames.hasMoreElements()) {
        String name = (String) enames.nextElement();
        String value = request.getHeader(name);
        map.put(name, value);
     }
     printTable(out, map, "Request Headers");

     // Print the session attributes

     map = new TreeMap();
     enames = session.getAttributeNames();
     while (enames.hasMoreElements()) {
        String name = (String) enames.nextElement();
        String value = "" + session.getAttribute(name);
        map.put(name, value);
     }
     printTable(out, map, "Session Attributes");

%>

<%-- Define a method to print a table --%>
<%!

     private static void printTable
        (Writer writer, Map map, String title)
     {
        // Get the output stream

        PrintWriter out = new PrintWriter(writer);

        // Write the header lines

        out.println("<TABLE BORDER=1 CELLPADDING=3>");
        out.println
           ("<TR><TH COLSPAN=2>" + title + "</TH></TR>");

        // Write the table rows

        Iterator imap = map.entrySet().iterator();
```

精彩好书见www.book118.com


```
            while (imap.hasNext()) {
               Map.Entry entry = (Map.Entry) imap.next();
               String key = (String) entry.getKey();
               String value = (String) entry.getValue();
               out.println("<TR>");
               out.println("<TD>" + key + "</TD>");
               out.println("<TD>" + value + "</TD>");
               out.println("</TR>");
            }

            // Write the footer lines

            out.println("</TABLE>");
            out.println("<P>");
         }
      %>
```

此JSP页面集合了两个表的数据：传递到请求对象的HTTP头标和会话属性。每一个输出都是格式化好的HTML表格。当然，表格可在迭代数据行的同时创建，但这需要复制格式化代码。可以使用一个私有静态方法printTable ( )，将其传递给包含关键字/取值对和表格标题的一个Map对象输出流的引用。

第8章将深入讨论声明。

6.2.6 隐含对象

虽然scriptlet、表达式、HTML模板数据均集成到_jspService ( ) 方法中，但JSP容器编写了方法本身的构架，初始化页面上下文及几个有用的变量。这些变量是在scriptlet和表达式中隐含有效的（但未声明）。可以像其他任何变量一样访问它们。但却不必事先声明。例如，传递到_jspService ( ) 方法的HttpServletRequest对象在名字请求下有效，如下面scriptlet所示：

```
<%
   String accountNumber = request.getParameter("acct");
   if (accountNumber == null) {
      // ... handle the missing account number problem
   }
%>
```

表6-2给出隐含变量的完整列表。

表6-2 隐含变量

变量名	取 值
request	正被服务的ServletRequest或HttpServletRequest
response	接收生成HTML输出的ServletResponse或HttpServletResponse
pageContext	此页面的PageContext对象。此对象是页面、请求、会话和应用的属性数据的主要存储位置
session	如果JSP页面使用了一个HttpSession，则它在名字session下可以利用
application	servlet上下文对象

精彩好书见www.book118.com

（续）

变量名	取值
out	用于生成输出HTML的字符输出流
config	此servlet上下文的ServletConfig对象
page	JSP页面本身的一个引用
exception	使得错误页面被调用的一个未捕获溢出。此变量只对带有isErrorPage="true"的页面可利用

还可以利用一个标签库创建另外的隐含变量。第11章对此进行讨论。

6.2.7 标准行为

行为是创建、修改或使用对象的高层JSP元素。与伪指令和脚本元素不同，行为使用严格的XML语法编码：

```
<tagname [attr="value" attr="value" ...] > ... </tag-name>
```

或者如果行为没有主体，则缩写形式为：

```
<tagname [attr="value" attr="value" ...] />
```

XML语法需要：

- 每一个标签必须具有匹配的结束标记或使用前面给出的/>简化形式。
- 属性取值必须放在引号里。
- 标签必须正确嵌套：<A><B>...</B></A>为合法，<A><B>...</A></B>非法。

在所有JSP 1.1兼容环境中，可利用的标准行为共有7种。在第15章对其进行详细描述。表6-3给出其语法。

表6-3 标准行为

标签名称	描述
<jsp:useBean>	声明一个Java Bean实例，使之与一变量名相关。语法为： <jsp:useBean id="name" [type="type"] [class="class"] [beanName="beanName"] [scope="page|request|session|application"]> ...</jsp:useBean>
<jsp:setProperty>	设置在前面<jsp:useBean>声明的Bean的一个或多个属性值。语法为： <jsp:setProperty name="id" prop-expression/> where prop-expression is one of the following: property="*" property="propName" property="propName" param="parameterName"

精彩好书见www.book118.com

（续）

<table>
<tr><th>标签名称</th><th>描　述</th></tr>
<tr><td></td><td>property="propName" value="value"
property="propName" value=<%= expression %></td></tr>
<tr><td><jsp:getProperty></td><td>返回一个Bean的指定属性值。语法为：
<jsp:getProperty name="id" property="name" /></td></tr>
<tr><td><jsp:include></td><td>调用另一资源，将其输出流并入JSP页面输出流。语法为：
<jsp:include page="URL" flush="true" />
or, if parameters need to be passed:
<jsp:include page="URL" flush="true">
<jsp:param ... />
<jsp:param ... />
...
<jsp:param ... />
</jsp:include></td></tr>
<tr><td><jsp:forward></td><td>将此HTTP请求转发置另一JSP页面或servlet进行处理。语法为：
<jsp:forward page="URL" />
or, if parameters need to be passed:
<jsp:forward page="URL">
<jsp:param ... />
<jsp:param ... />
...
<jsp:param ... />
</jsp:forward></td></tr>
<tr><td><jsp:param></td><td>将取值捆绑到一个名字，并将捆绑传至<jsp:include>或<jsp:forward>
调用的另一资源。语法为：
<jsp:forward>. Syntax is
<jsp:param name="name" value="value" /></td></tr>
<tr><td><jsp:plugin></td><td>用于生成下载Java插件相应的HTML链接
type="bean|applet"
code="objectCode"
codebase="objectCodebase"
{ align="alignment" }
{ archive="archiveList" }
{ height="height" }
{ hspace="hspace" }
{ jreversion="jreversion" }
{ name="componentName" }
{ vspace="vspace" }
{ width="width" }
{ nspluginurl="url" }
{ iepluginurl="url" } >
{ <jsp:params>
{ <jsp:param name="name" value="value" />
}+</jsp:params> }}</jsp:plugin></td></tr>
</table>

精彩好书见www.book118.com

6.2.8 标签扩展

除了表6-3列出的标准行为，JSP开发人员可以自己编写标签以扩展JSP的功能。第11章深入探讨标签扩展。

6.3 一个完整实例

下面的JSP页面实例融入了本章介绍的所有元素。页面名称为Echo.jsp。其惟一功能是向客户端浏览器传回一个HTML表格，其中包含浏览器发送的HTTP请求头标。程序列表如下：

```
<%@ page import="java.util.*" %>

<HTML>

<HEAD>
<TITLE>Echo</TITLE>
<STYLE>
<jsp:include page="style.css" flush="true"/>
</STYLE>
</HEAD>

<BODY>
<H3>HTTP Request Headers Received</H3>
<TABLE BORDER="1" CELLPADDING="4" CELLSPACING="0">
<%

   Enumeration eNames = request.getHeaderNames();
   while (eNames.hasMoreElements()) {
      String name = (String) eNames.nextElement();
      String value = normalize(request.getHeader(name));
%>
   <TR> <TD><%= name %></TD> <TD><%= value %></TD> </TR>
<%
   }
%>
</TABLE>
</BODY>
</HTML>
<%!
   private String normalize(String value)
   {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < value.length(); i++) {
       char c = value.charAt(i);
       sb.append(c);
       if (c == ';')
          sb.append("<BR>");
```

精彩好书见www.book118.com


```
    }
    return sb.toString();
  }
%>
```

当第一次调用Echo.jsp时，它创建了下列Java源码：

```
package Chap_00030_00035;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;

public class
   _0002fChap_00030_00035_0002fEcho_0002ejspEcho_jsp_5
   extends HttpJspBase
{

   // begin [file="Echo.jsp";from=(27,3);to=(39,0)]
   private String normalize(String value)
   {
      StringBuffer sb = new StringBuffer();
      for (int i = 0; i < value.length(); i++) {
         char c = value.charAt(i);
         sb.append(c);
         if (c == ';')
           sb.append("<BR>");
      }
      return sb.toString();
   }
   // end

   static {
   }

   public
      _0002fChap_00030_00035_0002fEcho_0002ejspEcho_jsp_5()
   {
   }
```

精彩好书见www.book118.com


```
private static boolean _jspx_inited = false;

public final void _jspx_init() throws JasperException
{
}

public void _jspService(
      HttpServletRequest request,
      HttpServletResponse response)
   throws IOException, ServletException
{
   JspFactory _jspxFactory = null;
   PageContext pageContext = null;
   HttpSession session = null;
   ServletContext application = null;
   ServletConfig config = null;
   JspWriter out = null;
   Object page = this;
   String _value = null;
   try {

        if (_jspx_inited == false) {
           _jspx_init();
           _jspx_inited = true;
        }

        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html");
        pageContext = _jspxFactory.getPageContext
           (this, request, response, "", true, 8192, true);

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        // HTML
        // begin [file="Echo.jsp";from=(0,32);to=(7,0)]
        out.write("\r\n\r\n");
        out.write("<HTML>\r\n\r\n");
        out.write("<HEAD>\r\n");
        out.write("<TITLE>Echo</TITLE>\r\n");
        out.write("<STYLE>\r\n");
        // end

        // begin [file="Echo.jsp";from=(7,0);to=(7,44)]
        {
```

精彩好书见www.book118.com


```
      String _jspx_qStr = "";
      out.flush();
      pageContext.include("style.css" + _jspx_qStr);
    }
    // end

    // HTML
    // begin [file="Echo.jsp";from=(7,44);to=(14,0)]
    out.write("\r\n");
    out.write("</STYLE>\r\n");
    out.write("</HEAD>\r\n\r\n");
    out.write("<BODY>\r\n");
    out.write("<H3>HTTP Request Headers Received");
    out.write("</H3>\r\n");
    out.write("<TABLE BORDER=\"1\"");
    out.write(" CELLPADDING=\"4\"");
    out.write(" CELLSPACING=\"0\">\r\n");
    // end

    // begin [file="Echo.jsp";from=(14,2);to=(19,0)]
    Enumeration eNames = request.getHeaderNames();
    while (eNames.hasMoreElements()) {
    String name = (String) eNames.nextElement();
    String value = normalize(request.getHeader(name));
    // end

    // HTML
    // begin [file="Echo.jsp";from=(19,2);to=(20,12)]
    out.write("\r\n <TR> <TD>");
    // end

    // begin [file="Echo.jsp";from=(20,15);to=(20,21)]
    out.print( name );
    // end

    // HTML
    // begin [file="Echo.jsp";from=(20,23);to=(20,33)]
    out.write("</TD> <TD>");
    // end

    // begin [file="Echo.jsp";from=(20,36);to=(20,43)]
    out.print( value );
    // end

    // HTML
    // begin [file="Echo.jsp";from=(20,45);to=(21,0)]
    out.write("</TD> </TR>\r\n");
    // end
```

精彩好书见www.book118.com


```
            // begin [file="Echo.jsp";from=(21,2);to=(23,0)]

            }
            // end

            // HTML
            // begin [file="Echo.jsp";from=(23,2);to=(27,0)]
            out.write("\r\n</TABLE>\r\n</BODY>\r\n</HTML>\r\n");
            // end

            // HTML
            // begin [file="Echo.jsp";from=(39,2);to=(40,0)]
            out.write("\r\n");
            // end
            }
            catch (Exception ex) {
               if (out.getBufferSize() != 0)
                  out.clear();
               pageContext.handlePageException(ex);
            }
            finally {
               out.flush();
               _jspxFactory.releasePageContext(pageContext);
            }
      }
}
```

下面按段分析该JSP页面和生成代码。

6.3.1 Page伪指令

JSP页面开始是一个Page伪指令，表明页面使用了java.util包：

```
<%@ page import="java.util.*" %>
```

此伪指令显示导入类列表在servlet源码中底部。

```
...
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;
```

6.3.2 <jsp:include>行为

页面使用一个样式单设置输出的外观和风格。样式单使用<jsp:include>行为进行声明。

```
<STYLE>
<jsp:include page="style.css" flush="true"/>
</STYLE>
```

<jsp:include>行为使得请求时读取下列样式单：

精彩好书见www.book118.com


```
body {
    color: #000000;
    background-color: #FEFEF2;
    font: Verdana 9pt;
};
```

6.3.3 scriptlet

页面上有两个scriptlet，在它们之前、之间和之后是定位的HTML模板数据。HTML数据

```
<HTML>

<HEAD>
<TITLE>Echo</TITLE>
...
```

通过写入下列语句不修改地传递：

```
out.write("\r\n");
out.write("<HTML>\r\n\r\n");
out.write("<HEAD>\r\n ");
out.write("<TITLE>Echo</TITLE>\r\n ");
...
```

然后第一个scriptlet被复制到此servlet中：

```
Enumeration eNames = request.getHeaderNames();
while (eNames.HasMoreElements()) {
    String name = (String) eNames.nextElement();
    String value = normalize(request.getHeaderName());
```

注意，在第2行代码段有一个大括号。相匹配的括号在第2个scriptlet给出。

6.3.4 表达式

在循环的每一步中，scriptlet从请求对象中抽取一个头标名和头标值。并不使用out.write ( )将之打印，页面设计人员选择将其置入HTML模式，使用JSP表达式标签，

```
%>
    <TR> <TD><%= name %></TD> <TD><%= value %></TD> </TR>
<%
```

生成下列servlet代码：

```
// HTML
// begin [file="Echo.jsp";from=(19,2);to=(20,12)]
out.write("\r\n <TR> <TD>");
// end

// begin [file="Echo.jsp";from=(20,15);to=(20,21)]
out.print( name );
```

精彩好书见www.book118.com


```
// end

// HTML
// begin [file="Echo.jsp";from=(20,23);to=(20,33)]
out.write("</TD> <TD>");
// end

// begin [file="Echo.jsp";from=(20,36);to=(20,43)]
out.print( value );
// end

// HTML
// begin [file="Echo.jsp";from=(20,45);to=(21,0)]
out.write("</TD> </TR>\r\n");
// end
```

6.3.5 一个声明

列出的头标值可能很长，使得表格宽度扭曲变形。解决此问题可以通过按分号浏览头标值，在出现分号的位置插入
标签。执行此功能的方法称为normalize ( )，在JSP文件的结尾处可以找到它：

```
<%!
   private String normalize(String value)
   {
      StringBuffer sb = new StringBuffer();
      for (int i = 0; i < value.length(); i++) {
         char c = value.charAt(i);
         sb.append(c);
         if (c == ';')
           sb.append("<BR>");
      }
      return sb.toString();
   }
%>
```

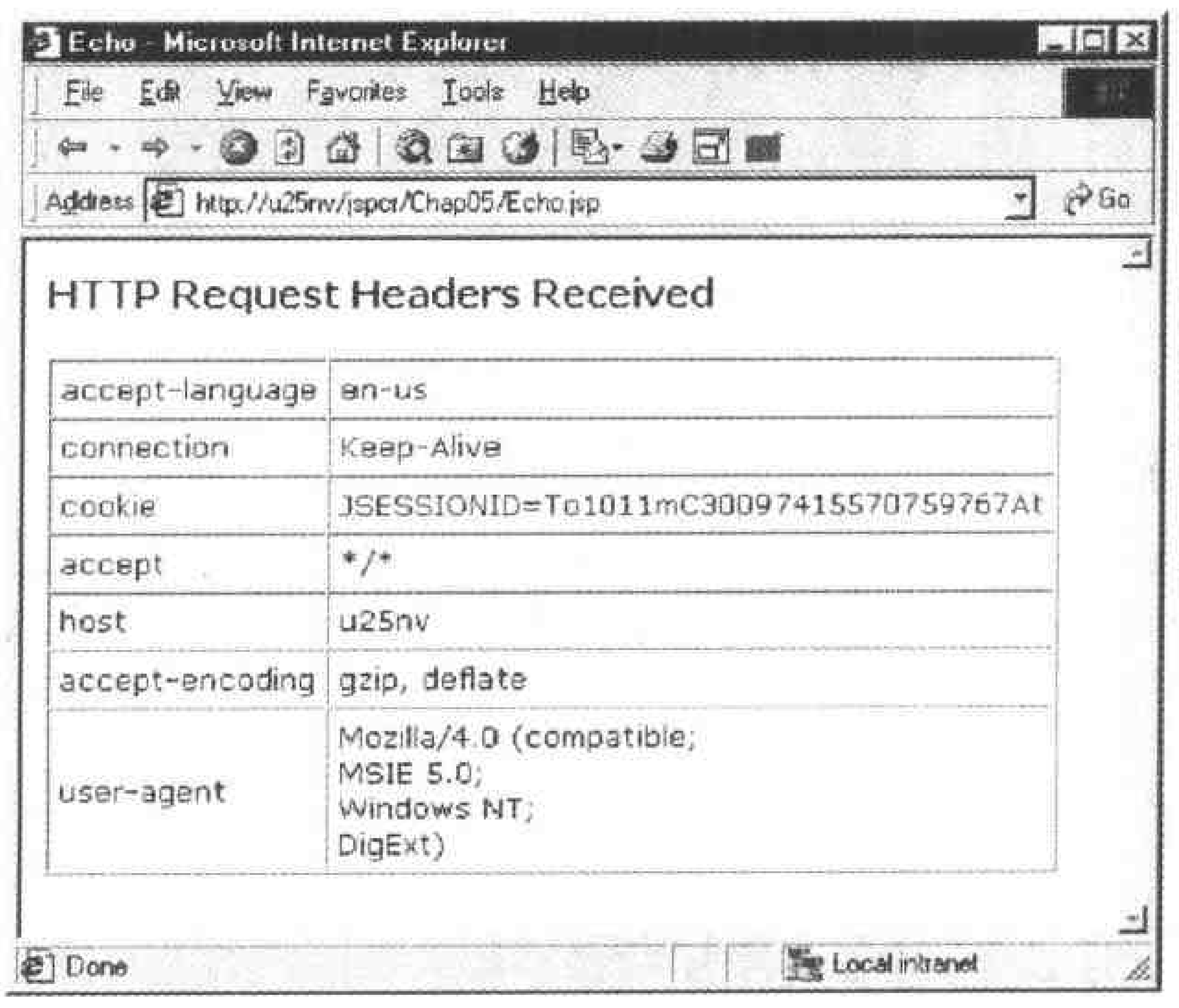
像对两个scriptlet一样，声明代码不修改传入生成的servlet中。但它不放在_jspService ( )方法中，而是在servlet开始附近，任何其他方法外将其编入类模块。

```
// begin [file="Echo.jsp";from=(27,3);to=(39,0)]
private String normalize(String value)
{
   StringBuffer sb = new StringBuffer();
   for (int i = 0; i < value.length(); i++) {
      char c = value.charAt(i);
      sb.append(c);
      if (c == ';')
        sb.append("<BR>");
```

精彩好书见www.book118.com


```
        }
        return sb.toString();
    }
    // end
```

输出结果如下图所示：



Echo.jsp的输出

6.4 小结

JSP开发环境提供了使用服务器端Java编程动态生成HTML页面的一种方式。其语法允许大部分HTML被直接编入，并有控制代码生成的Java代码标记段。它支持同时静态和动态地包含其他资源。JavaBean被完全整和到框架中，定制标签允许功能被封装，并可利用于非专业编程人员。

本章关键的一点是：JSP开发周期的思维模式对理解创建和调试应用是至关重要的。转换时（例如，静态资源通过<%@include%>伪指令被包含）及请求时（动态请求使用<jsp:include>）发生的行为使我们看清楚要使用的特性和使用它们的契机。第二部分其余章节将深入讨论此应用模式的每一特性。

精彩好书见www.book118.com

第7章　表达式和scriptlet

前面一章介绍了JSP语法和语义。语法并不很难学习，但掌握它并不表示已经知道了所需的一切。理解JSP需要构建一种其如何操作的思维模式——何时，以何方式生成Java源码、何时编译并载入类。

本章通过开发两个脚本元素阐述部分思维模式；表达式和scriptlet。可以看出JSP容器如何结合模板文本和JSP脚本元素生成一个处理用户请求的Java方法。本章还讨论了JSP页面如何访问其使用的Web环境，如何与产生的结果通信。

7.1　表达式

一个JSP表达式在JSP页面中是一个Java[1]语言表达式，由其所包围的HTML环境通过分隔符<%=和%>设置，如下所示：

```
<%= expression %>
```

例如，一个表达式可为基本数字型取值：

```
<B>Simple math:</B> 2 + 2 = <%= 2 + 2 %>
```

产生结果：

```
Simple math: 2 + 2 = 4
```

或包含方法调用的较复杂的表达式：

```
The Java virtual machine vendor is
<em><%= System.getProperty("java.vm.vendor") %></em>
```

产生结果为：

The Java virtual machine vendor is Sun Microsystems Inc.

一个表达式可以创建新的对象并表示它们。下面代码创建一个Date对象，将其传递到一个新SimpleDateFormat对象的format（ ）方法：

```
Today is
<%=
   new java.text.SimpleDateFormat("MMMM d, yyyy")
   .format(new java.util.Date())
%>
```

1　理论上，像在JSP规范中预想的，JSP页面可以用其他语言编写。因为这样编写已经约定俗成，所以Java是惟一的支持语言。这就是为什么将该技术称为JavaServer Page（JSP），而不是语言无关的Server Page（LISP）或任意旧式语言（AOLSP）。

精彩好书见www.book118.com

输出为：（当然为当天日期）

Today is June 28,2001

在分隔符〈%和%〉之间的Java表达式具有随意复杂性，惟一需要的是能够直接或间接通过调用其toString（ ）方法或String.valueof（ ）方法被表示成一个java.util.String。

注意 *表达式不能以分号结束。它们必须在等号和结束分号之间赋值语句的右边以合法的形式组成。*

7.2 scriptlet

scriptlet是嵌入到HTML页面的Java编程语句的集合。通过放置在<%和%>标记之间和周围的HTML区分。如下所示：

```
<% statement; [statement; …] %>
```

在<%之后，%>之前允许有空行，因此前面scriptlet也可以写为：

```
<%
statement;
[statement; …]
%>
```

下面的例子是使用一个scriptlet生成ASCII字符表的JSP页面：

```
<HTML>
<BODY>
<CENTER>

<H3>ASCII Table</H3>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0">
<%
   StringBuffer sb = new StringBuffer();
   sb.append("<TR>");
   sb.append("<TH WIDTH=40> </TH>");
   for (int col = 0; col < 16; col++) {
      sb.append("<TH>");
      sb.append(Integer.toHexString(col));
      sb.append("</TH>");
   }
   sb.append("</TR>");
   for (int row = 0; row < 16; row++) {
      sb.append("<TR>");
      sb.append("<TH>");
      sb.append(Integer.toHexString(row));
      sb.append("</TH>");
      for (int col = 0; col < 16; col++) {
         char c = (char)(row * 16 + col);
         sb.append("<TD WIDTH=32 ALIGN=CENTER>");
```

精彩好书见www.book118.com


```
            sb.append(c);
            sb.append("</TD>");
        }
        sb.append("</TR>");
    }
    out.println(sb);
%>
</TABLE>
</CENTER>
</BODY>
</HTML>
```

在第5行HTML后是scriptlet开始分隔符<%，然后是许多行Java代码，scriptlet结束分隔符%>，然后是关闭文档的HTML行。被调用时，此页面输出如图7-1所示。

下面一节讲述如何使用JSP容器处理这些脚本元素。

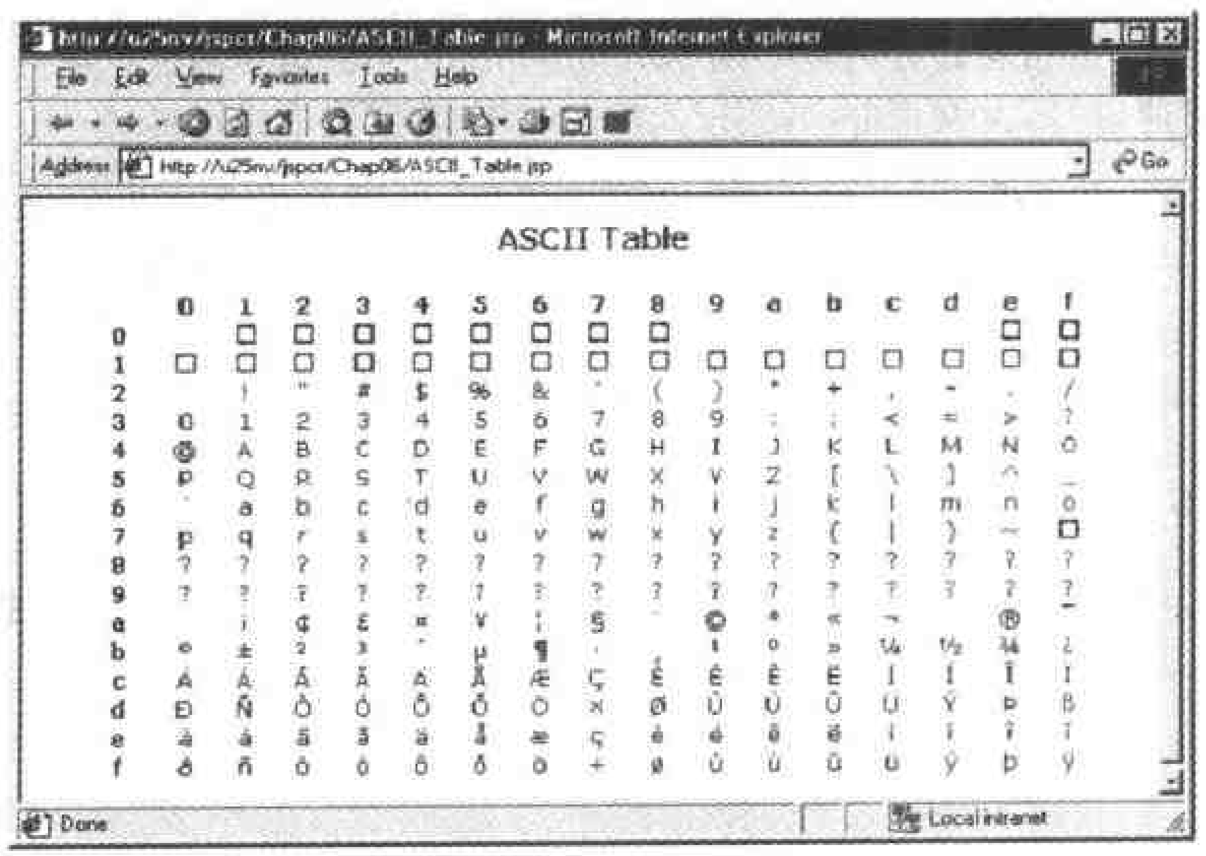


图7-1　JSP输出的ASCII-Table

7.3　通过JSP容器处理表达式和scriptlet

当遇到一个新的或已修改JSP页面，JSP容器解析它并创建等价的Java servlet[1]源码。表达式、scriptlet和页面中找到的HTML模板数据被JSP容器用来创建名为_jspService（）方法的Java源码。此方法对应于一个servlet的service（）方法，或更常用的doGet（）和doPost（）方法。_jspService（）由容器自动生成。JSP作者不必显式定义它。

生成的_jspService（）方法依据JSP页面的内容由3类语句组成：

1　Servlet在第4章已讨论。

精彩好书见www.book118.com

- 处理HTML模板数据和表达式的代码。
- 任意scriptlet内容。
- 容器产生的初始化和退出代码。

下面分别讲解，观察其处理方式。

7.3.1 HTML模板数据和表达式

JSP页面内不在一个JSP元素（伪指令、表达式、scriptlet和行为）中的任意字符可被认为是固定HTML模板的一部分。JSP容器创建将这些字符写入响应输出流的out.write（ ）或out.print（ ）语句。例如代码

```
<LI>Cash and Marketable Securities
```

被转换为：

```
out.write("<LI>Cash and Marketable Securities\r\n");
```

如果HTML模板需要包含任意字母<%字符串，则它们必须被特殊对待以避免与JSP容器混淆。JSP 1.1规范指出可通过写入<\%而非<%实现此功能。JSP容器生成代码将<%写入输出流。

注意 JSP容器典型情况对每一个未被打断的固定HTML数据生成一个长的out.write（ ）。本书实例从可读性考虑随意将长的字符串打断到多个out.write（ ）语句中。

除了固定HTML数据，模板也可以包含运行时用于求值并使用out.write（ ）语句打印出来的JSP表达式。在下一节讨论表达式。

7.3.2 scriptlet内容

在<%和%>之间的任意内容被原封不动地复制到_jspService（ ）方法中，因此JSP页面中下列行：

```
<TABLE BORDER=0>
<TR><TH>Celsius</TH><TH>Fahrenheit</TH></TR>
<%
   for (int c = 0; c <= 100; c += 10) {
      int f = 32 + 9*c/5;
      out.print("<TR><TD>" + c + "</TD>");
      out.print("<TD>" + f + "</TD></TR>");
   }
%>
</TABLE>
```

被JSP容器转换成_jspService（ ）方法中的下列行：

```
// HTML
// begin [file="c2f.jsp";from=(0,0);to=(2,0)]
out.write("<TABLE BORDER=0>\r\n");
out.write("<TR>");
out.write("<TH>Celsius</TH>");
```

精彩好书见www.book118.com


```
out.write("<TH>Fahrenheit</TH>");
out.write("</TR>\r\n");
// end
// begin [file="c2f.jsp";from=(2,2);to=(8,0)]

for (int c = 0; c <= 100; c += 10) {
   int f = 32 + 9*c/5;
   out.print("<TR><TD>" + c + "</TD>");
   out.print("<TD>" + f + "</TD></TR>");
}
// end
// HTML
// begin [file="c2f.jsp";from=(8,2);to=(10,0)]
out.write("\r\n</TABLE>\r\n");
// end
```

表格的HTML标记可在out.write（）语句中找到，方法主体中的scriptlet内容没有改变。

如果一个页面中有多个scriptlet，则按照遇到它们的次序将其复制，因此下列代码比较起来并无功能上的差别：

```
<%
    for (int i = 0; i < 10; i++) {
        out.println(i);
    }
%>
```

和

```
<% for (int i = 0; i < 10; i++) { %>
<%      out.println(i); %>
<% } %>
```

但在第2段代码中生成了几个新行字符（发生此情况是因为技术上新行被认为是固定HTML数据）。因为多个scriptlet被连在一起并放入同一方法，语法上讲单元可在一个scriptlet中开始，在另一个中完成。就像for语句中括号和大括号所示的那样。这也表示定义在任意scriptlet中的变量可被认为是_jspService（）方法的局部变量，并在从一个scriptlet或表达式到另一个时保留其值。

7.3.3 容器生成的初始化和退出代码

除了JSP作者编写的代码外，_jspService（）生成了开始和结束语句，以初始化和释放方法中所需的对象。生成的精确代码是与实现相关的，并指定于JSP容器厂家。在前面给出的摄氏到华氏的例子中，Tomcat生成了下列初始化和退出代码：

```
public void _jspService(
        HttpServletRequest request,
        HttpServletResponse response)
    throws IOException, ServletException
```

精彩好书见www.book118.com


```
    {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext
                (this, request, response, "", true, 8192, true);
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // ... your code appears here ...

        }
        catch (Exception ex) {
            if (out.getBufferSize() != 0)
                out.clearBuffer();
            pageContext.handlePageException(ex);
        }
        finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
}
```

可以看出在JSP作者加入的代码前创建了许多对象。这些对象的含义是下一节要讨论的内容。

7.4 隐含对象和JSP环境

在JSP页面中编写的scriptlet和表达式不会作为一个完整的程序存在，它们需要实施操作的环境。JSP容器提供此环境并通过调用隐含对象使之对页面作者可利用。这些对象由容器生成的语句在_jspService（）方法开始处创建并被设置成在所有JSP页面中都一样的名字。这些对象共有9个，如表7-1所示：

表7-1 在Scriptlet和表达式中可利用的隐含对象

对 象	描 述
request	传递到_jspService（）的HttpServletRequest对象
response	传递到_jspService（）的HttpServletResponse对象

精彩好书见www.book118.com

（续）

对　象	描　述
pageContext	访问页面、请求、会话或应用属性的一种方式
session	如果存在表示当前HttpSession对象
application	servlet上下文对象
out	JspWriter响应输出流对象
config	servlet配置对象
page	HSP类本身的当前实例的引用
exception	捕获的溢出（只在错误页面中有效）

可以像访问其他变量一样使用预定义名访问这些变量。在本章实例中用到了其中之一——JspWriter out变量。

```
<%
out.println("<B>out</B> is an <I>");
out.println(out.getClass().getName());
out.println("</I> object.");
%>
```

当运行于Tomcat下时输出为：

```
out is an org.apache.jasper.runtime.JspWriterImpl object.
```

当然这是厂家指定的。在JRun3.0下输出为：

```
out is an allaire.jrun.jsp.JRunJspWriter object.
```

JSP隐含对象服务于HTTP请求的上下文。下面各节详细讲述每一对象。

7.4.1 Request

request变量包含传递到生成的_jspService（）方法的第一个参数HttpServletRequest对象的引用。

此对象封装了由Web浏览器或其他客户端生成的HTTP请求的细节——参数、属性、头标和数据。其可用方法在表7-2中列出[1]。

表7-2　request对象的一些可用方法

方　法	描　述
String getHeader(String name)	返回指定的HTTP头标值，如果请求中不存在头标，则返回null
Enumeration getHeaderNames()	返回请求中出现的所有HTTP头标的一个枚举值
String getParameter(String name)	给定单值形式参数的名字，返回其取值
Enumeration getParameterNames()	返回传递到此请求所有形式的参数名字的枚举
HttpSession getSession(boolean creat)	返回当前HttpSession对象。如果不存在，依据create取值创建一个新的或是返回null

1　Servlet 2.2API中javax.servlet.http.HttpServletRequest和所有其他类的完整描述可在附录A中找到。

精彩好书见www.book118.com

7.4.2 Response

response对象提供对HTTP事务处理的另一端的访问。此对象封装了返回到HTTP客户端的输出，向页面作者提供设置响应头标和状态码的方式。它也包含访问响应输出流的方法，但JSP规范禁止直接访问此输出流。所有JSP响应输出必须使用out隐含变量写入。HttpServletResponse对象提供的方法包含表7-3中列出的属性。

表7-3 response对象的一些可用方法

方 法	描 述
boolean isCommitted()	返回指明HTTP响应是否已返回到客户端的一个标记
void setHeader(String name, String value)	设置一个HTTP头标为指定名字并取值
void setStatus(int sc)	设置HTTP状态为指定值

7.4.3 PageContext

JSP代码在一个环境层次结构中执行，如图7-2所示。例如，多个JSP页面可以服务于同一个简单的HTTP请求：一个产生头标信息，另一个生成具体的输出。类似的，多个HTTP请求也可以是一个大的HTTP会话的一部分，此会话从注册请求开始，接着是某用户选择请求，然后是将工作确认到数据库，最后在一个servlet上下文中的所有HTTP会话集和可以共享同一连接池或其他通用对象。

此层次结构中每一层都包含只应用于此层的属性。JSP规范提供PageContext对象跟踪4个层次上的属性。

- JSP页面
- HTTP请求
- HTTP会话
- 整个应用

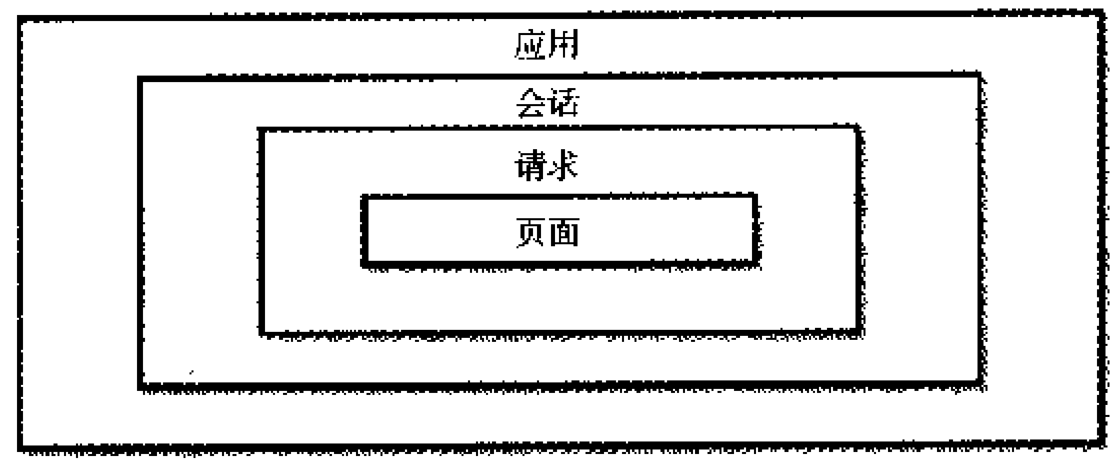


图7-2 JSP上下文层次

PageContext对象在_jspService（）方法开始时被自动初始化并设置成一名为pageContext的变量。此对象提供所有4个层次的属性查询和修改能力，如表7-4所示。它也提供转发请求到其他资源和包含其他资源输出的各种方法。

精彩好书见www.book118.com

表7-4 pageContext对象的一些可用方法

方 法	描 述
Object findAttribute(String name)	在页面、请求、会话和应用范围内寻找具有指定名字的属性对象，返回第一个找到的对象，如果没有，返回null
Object getAttribute(String name ,int scope)	在给定范围内返回具有指定名字的属性对象。scope参数值从PageContext类中的PAGE_CONTEXT、REQUEST_CONTEXT、SESSION_CONTEXT和APPLICATION_CONTEXT常量中选择
void removeAttribute(String name , int scope)	在给定范围内删除具有指定名的属性对象
void setAttribute(String name ,Object value, int scope)	在给定范围内将对象保存为命名属性。scope参数值从PageContext类中的PAGE_CONTEXT、REQUEST_CONTEXT、SESSION_CONTEXT和APPLICATION_CONTEXT常量中选择

7.4.4 Session

HTTP是一个无状态协议，这意味着它在从一个请求到另一个请求时不会记住前一个请求的一切，然而Web应用经常调用多个请求。例如，一个应用开始时可以是某种必须传播到几个其他Web页面的用户标识和有效性验证。此类应用的需求必须通过不同于Web服务器的其他方式实现。

依据应用需求，可以采纳以下几种方法实现此目的。第14章详细讨论了许多的方法，包括：

- 隐藏域
- Cookie
- URL 重写
- HTTP会话

这里只对最后一条感兴趣。一个HttpSession是一个类似于哈希表的与单一Web浏览器会话相关的对象。它存在于HTTP请求之间，可以存储任何类型的命名对象。缺省的，JSP容器在_jspService（）方法的开始创建一个HttpSession对象，或访问当前被激活的HttpSession对象。此对象被设置为一名为session的变量。

如果不需要在请求之间保留对象，可以通过在page伪指令中指定session=“false”关闭自动创建会话。这样做可以通过减少servlet引擎必须跟踪的对象数目提高性能。因为一个会话存活于其超时之前（典型为30min）或显式使之无效，性能的影响可能是很重要的。

表7-5概括了会话对象中几种有用的方法。

表7-5 session对象的可用方法

方 法	描 述
Object getAttribute(String name)	如果会话中存在，返回具有指定名字的对象
Enumeration getAttributeNames()	返回保存在会话中所有对象名字的枚举
String getId()	返回惟一会话ID。此ID必须由客户端（Web浏览器）在请求之间保存，并传递到JSP容器中以标识所需的会话
int getMaxInactiveInterval()	返回会话在用户请求之间处于活动状态的最大秒数。如果超出此秒数仍没有发生动作，JSP容器关闭此会话
void invalidate()	关闭会话，释放其所有对象
void setAttribute(String name, Object value)	将会话中一个对象保存为指定名字

精彩好书见www.book118.com

记住，pageContext对象也可以与session.getAttribute()和session.setAttribute()方法一样的方式取得并设置会话属性。

7.4.5 Application

application隐含对象封装了Web应用中所有servlet、JSP页面、HTML页面和其他资源的集合属性。此对象实现了javax.servlet.ServletContext在_jspService（）方法开始时自动被创建。它提供关于服务器版本、应用级初始化参数和应用内资源绝对路径的信息。此对象也提供注册信息的方式。其可用方法在表7-6中列出。

表7-6 application对象的可用方法

方 法	描 述
Enumeration getAttributeNames()	返回保存在servlet上下文中所有对象名字的一个枚举值
Object getAttribute(String name)	返回使用应用setAttribute（）方法保存的具有指定名字的对象
String getInitParameter(String name)	返回指定的应用级初始化参数值
Enumeration getInitParameterNames()	返回所有应用级初始化参数名字的一个枚举值
String getRealPath(String path)	如果可能，将Web应用上下文中一个路径转换为文件系统中绝对路径
URL getResource(String path)	返回映射到应用中指定路径的URL。路径必须以"/"开始，并且是相对于应用的根
InputStream getResourceAsStream(String path)	与getResource（）操作类似，但是返回一个至结果URL打开的输入流
void log(String msg)	将一个信息写入与应用相关的注册文件

像page、request和session隐含对象一样，application对象的属性也可以使用pageContext对象表示。

初始化参数在本章后面几节讨论。

7.4.6 Out

JSP页面的主要意图就是生成输出，并将其发回套接字连接的另外一端用户。正如本章前面所见，固定HTML模板数据和JSP表达式通过自动生成的out.write()和out.print（）方法调用被编写。在_jspService（）方法中开始时使用javax.servlet.jsp.JspWriter对象的引用初始化out变量。可以用这种方式生成所有输出或者在scriptlet中将输出显式写入out。因此，JSP页面

```
<%
    String[] colors = {"red", "green", "blue"};
    for (int i = 0; i < colors.length; i++) {
%>
<%= colors[i] %> <P>
<%
    }
%>
```

功能上等价于下面一个：

精彩好书见www.book118.com


```
<%
    String[] colors = {"red", "green", "blue"};
    for (int i = 0; i < colors.length; i++) {
      out.println(colors[i] + " <P>");
    }
%>
```

除了write（）方法可通用于所有java.io.Writer对象外，out对象提供查寻和表示输出缓冲区的方法，如表7-7所示：

表7-7 out对象可用方法

方　法	描　述
void flush ( )	强迫缓存数据被写入输出流
int getBufferSize( )	以字节数返回输出输出缓冲区大小，如果写人者不被调人缓存，则返回0
int getRemaining ( )	返回在缓存溢出发生前仍保留的字节数
void print(type value)	写入指定要素或对象属性的对象方法之一。在结尾处不加入新行
void println (type value)	类似于print（），但在结尾处加入新行

7.4.7 Config

除了通过application对象使应用级初始化参数可用，单独的servlet映射（JSP页面也可）也可以有初始化参数。config隐含对象提供了访问这些参数、servlet上下文（应用）和servlet名字的方法，具体见表7-8所示。

7.4.8 Page

page隐含对象是一个包含当前servlet接口引用的变量。基本上是this变量的别名。此对象典型情况对JSP页面写入者不可用。

7.4.9 Exception

被隐含exception变量引用的对象是通过JSP页面中一个catch块已经溢出但未捕获的java.lang.Throwable的任意实例。exception只有在<%@page%>伪指令具有属性isErrorPage="true"时才有效。其属性在第10章详细讨论。

表7-8 config对象可用方法

方　法	描　述
String getInitParameter(String name)	返回指定servlet初始化参数取值，如果命名的参数不存在，则返回null
Enumeration getInitParameterNames()	返回此servlet所有初始化参数名字的列表
ServletContext getServletContext()	返回servlet上下文的一个引用（同application隐含变量）
String name getServletName()	返回生成的servlet的名字

精彩好书见www.book118.com

7.5 初始化参数

初始化参数是可被JSP页面读取的外部名称/取值对。其使用方式与字符串常量相同。但有一些额外的优点。修改它们可以不用重新编译用到它们的程序。这使得初始化参数对保存安装和配置数据特别有用，如HTTP代理服务器名称，应用颜色设置或安装目录名称。

这些参数可在单独的JSP和servlet层次上或对一个应用的所有JSP页面上指定。对于后者，初始化参数在应用的web.xml[1]文件中声明。对于JSP和servlet层次的访问，则可通过向适当<servlet>元素加入一或多个<init-param>元素完成，如下所示：

```
<servlet>

   <servlet-name>Food</servlet-name>
   <jsp-file>/Chap07/examples/Food.jsp</jsp-file>

   <init-param>
      <param-name>DRIVER_NAME</param-name>
      <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
   </init-param>

   <init-param>
      <param-name>DATABASE_URL</param-name>
      <param-value>jdbc:odbc:usda</param-value>
   </init-param>

</servlet>
```

此例中，Food.jsp是一个访问营养信息数据库的JSP页面。不是包含JDBC驱动程序名字和数据库URL的硬编码值，JSP页面使用getInitParameter（ ）方法从初始化参数中得到这些值：

```
String driverName = getInitParameter("DRIVER_NAME");
if (driverName == null)

   throw new ServletException
   ("No DRIVER_NAME parameter was specified");

String databaseURL = getInitParameter("DATABASE_URL");
if (databaseURL == null)
   throw new ServletException
   ("No DATABASE_URL parameter was specified");

Class.forName(driverName);
Connection con = DriverManager.getConnection(databaseURL);
```

数据库访问参数可能在一个Web应用的几处位置用到，不必复制web.xml文件中的值，常用值可在应用层次上指定。使用<context-param>元素实现此功能：

1 web.xml文件和其他配置和发布问题在第18章讨论。

精彩好书见www.book118.com


```
<context-param>
    <param-name>DRIVER_NAME</param-name>
    <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>

<context-param>
    <param-name>DATABASE_URL</param-name>
    <param-value>jdbc:odbc:usda</param-value>
</context-param>
```

访问这些值的JSP代码几乎是一样的。只有应用对象的getInitParameter（）方法调用不同：

```
String driverName =
      application.getInitParameter("DRIVER_NAME");
   if (driverName == null)
      throw new ServletException
      ("No DRIVER_NAME parameter was specified");

String databaseURL =
      application.getInitParameter("DATABASE_URL");
   if (databaseURL == null)
      throw new ServletException
      ("No DATABASE_URL parameter was specified");

Class.forName(driverName);
Connection con = DriverManager.getConnection(databaseURL);
```

7.6 小结

JSP页面提供将Java代码嵌入到请求处理过程的两种方式。表达式和scriptlet。JSP表达式是生成一个字符串值的简单Java语言表达式（或者可被转换成一个字符串值）。表达式用<%=和%>分隔符封装。分隔符之间的部分均可作为out.print()或out.write()方法的参数。为此，表达式不能以分号结束。Scriptlet是设计在_jspService（）方法中实施操作的Java代码段，由<%和%>分隔符标记。在scriptlet中的编程声明可直接复制到生成servlet的Java代码。

为了将上述方法连接到JSP容器，JSP页面可以访问许多隐含对象。它们是具有预定义名的自动初始化对象。这些变量为：

- request
- response
- pageContext
- session
- application
- out
- config
- page
- exception

最后一个变量（exception）只对在页面伪指令中具有isErrorPage="true"属性的页面可用。

精彩好书见www.book118.com

第8章 声 明

前面一章讲述了JSP表达式和scriptlet，这两种元素类型与固定HTML模板数据一起共享一种通用环境——存在于生成的Java servlet的_jspService（）方法内。这对大多数请求过程已经足够了。_jspService（）方法强调对servlet功能的限制。本章介绍JSP声明，允许JSP作者编写操作于_jspService（）方法之外的Java代码。

8.1 声明是什么

类似于scriptlet，一个JSP声明由嵌入到HTML页面的Java源码组成。通过如下特定的打开和关闭标签将声明与页面其余部分区分开：

```
<%! java statements %>
```

声明的语法与scriptlet的语法一致，只有一点例外：开始分隔符为<%!而不是<%。

如同一个scriptlet，声明分隔符内的代码被不加修改地复制到生成的Java scriptlet。基本差别是代码放置位置：scriptlet被放置到_jspService（）方法内，而声明作为封装类的高层成员放在该方法外。理解这一点差别有助于了解JSP工作方式的思维模型，可以解释一些意外行为。

在何处生成声明代码?

如何生成声明代码的一个例子将使这一点更清晰。考虑下面使用scriptlet显示当前时间的JSP页面：

```
<%@ page import="java.text.*,java.util.*" %>
<%
   DateFormat fmt = new SimpleDateFormat("hh:mm:ss aa");
   String now = fmt.format(new Date());
%>
The time is <%= now %>
```

页面存储于文件ShowTimeS.jsp（S代表scriptlet）。调用此文件时，显示当前时间：

```
The time is 09:31:45 PM
```

如果用户刷新页面，时间如预想般递增：

```
The time is 09:31:48 PM
The time is 09:31:51 PM
The time is 09:31:53 PM
```

下面考虑使用一个声明而不是scriptlet编写的同样JSP。此页面名为ShowTimeD.jsp（D代表声明）：

精彩好书见www.book118.com


```
<%@ page import="java.text.*,java.util.*" %>
<%!
   DateFormat fmt = new SimpleDateFormat("hh:mm:ss aa");
   String now = fmt.format(new Date());
%>
The time is <%= now %>
```

ShowTimeS.jsp和ShowTimeD.jsp惟一的区别是ShowTimeD.jsp第2行以<%!开始而不是<%，表明这是一个声明而不是scriptlet。

调用ShowTimeD.jsp时，同样显示当前时间：

```
The time is 09:32:26 PM
```

但当刷新页面时，出现以下结果：

```
The time is 09:32:26 PM
The time is 09:32:26 PM
The time is 09:32:26 PM
```

时间没有变，为什么？答案可以在每个页面生成的servlet源码中找到。以下是scriptlet版本：

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.text.*;
import java.util.*;

public class ShowTimeS extends HttpJspBase
{
   static
   {
   }

   public ShowTimeS()
   {
   }

public void _jspService(
      HttpServletRequest request,
```

精彩好书见www.book118.com


```
        HttpServletResponse response)
    throws IOException, ServletException
  {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
    try {
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html;charset=8859_1");
        pageContext = _jspxFactory.getPageContext
            (this, request, response, "", true, 8192, true);

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        out.write("\r\n");
        DateFormat fmt = new SimpleDateFormat("hh:mm:ss aa");
        String now = fmt.format(new Date());
        out.write("\r\nThe time is ");
        out.print( now );
        out.write("\r\n");
    }
    catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    }
    finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

以下是声明版本：

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
```

精彩好书见www.book118.com


```
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.text.*;
import java.util.*;

public class ShowTimeD extends HttpJspBase
{
   DateFormat fmt = new SimpleDateFormat("hh:mm:ss aa");
   String now = fmt.format(new Date());

   static
   {
   }

   public ShowTimeD()
   {
   }

   public void _jspService(
         HttpServletRequest request,
         HttpServletResponse response)
      throws IOException, ServletException
   {
      JspFactory _jspxFactory = null;
      PageContext pageContext = null;
      HttpSession session = null;
      ServletContext application = null;
      ServletConfig config = null;
      JspWriter out = null;
      Object page = this;
      String _value = null;
      try {
         _jspxFactory = JspFactory.getDefaultFactory();
         response.setContentType("text/html;charset=8859_1");
         pageContext = _jspxFactory.getPageContext
            (this, request, response, "", true, 8192, true);

         application = pageContext.getServletContext();
         config = pageContext.getServletConfig();
         session = pageContext.getSession();
         out = pageContext.getOut();
```

精彩好书见www.book118.com


```
            out.write("\r\n");
            out.write("\r\nThe time is ");
            out.print( now );
            out.write("\r\n");
        }
        catch (Exception ex) {
            if (out.getBufferSize() != 0)
              out.clearBuffer();
            pageContext.handlePageException(ex);
            }
            finally {
               out.flush();
               _jspxFactory.releasePageContext(pageContext);
            }
     }
}
```

除了程序名，两个servlet的差别是两个脚本行的位置。在scriptlet版本中，它们包含在_jspService ( ) 方法的中间，fmt和now是此方法的局部变量。然而在声明版本中，它们出现在类中的第一项。这使得两个变量成为实例变量。当servlet第一次被创建时它们被初始化，并再也不变：可以看出这不但是我们不想要的，也是很危险的。

8.2 声明的基本用法

声明可以包含任何有效的Java代码，但在下列三种上下文中最常用：

- 变量声明 类和实例变量均可以被声明和初始化。
- 方法定义 重复的或过度复杂的scriptlet代码可以被重新构建到一个调用其他方法的主过程中。
- 内部类 可以定义附加类，并使之对scriptlet、表达式和其他声明代码可用。

本章其余部分详细讨论这些用法。

8.3 变量声明

如前面例子所示，声明可用于定义和初始化变量。这些变量将对scriptlet、表达式和其他声明可用。它们可以是类变量（标记为static关键字），如下所示：

```
<%!
   static final String[] COLORS = {
      "#CA9A26",
      "#3BF428",
      "#F7E339",
      "#FF40FF",
   };
%>
<%
```

精彩好书见www.book118.com


```
   for (int i = 0; i < COLORS.length; i++) {
      String color = COLORS[i];
%>
<DIV STYLE="background-color: <%= color%>;
            font-size: 12pt;
            font-weight: bold;">
   This is color <%= color %></DIV>
<% } %>
```

或实例变量，如下列文件vardec2.jsp所示：

```
<%! int count; %>
<%
   count = 0;
   for (int i = 0; i < 10; i++) {
%>
Request <%= Integer.toHexString(request.hashCode()) %>
count = <%= ++count %><BR>
<%
      Thread.sleep(250);
   }
%>
```

在这两种情况下，变量声明均作为封装类的高层成员被原封不动地复制到生成的servlet中。

线程安全和实例变量

例子vardec2.jsp中实例变量包含一点小的瑕疵。每次JSP服务于一个请求时，它将count变量设置为0，然后进入10次循环，增加计数并在显示请求对象杂乱代码的同时显示它。首次测试时，显示输出如图8-1所示。

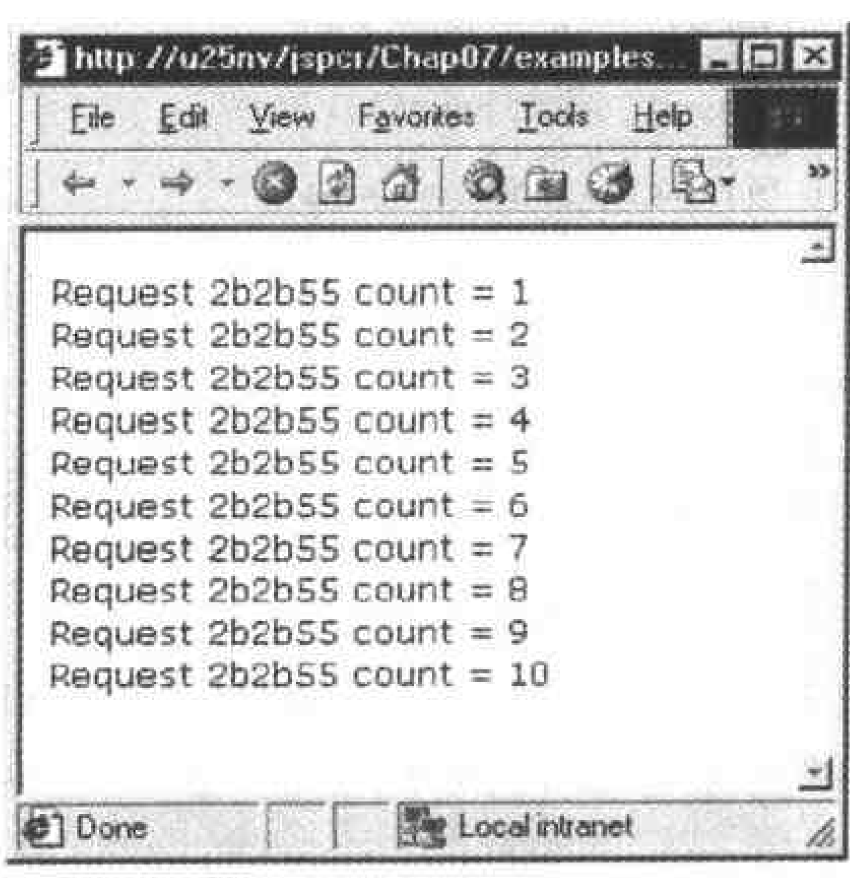


图8-1 vardec2.jsp的首次测试

精彩好书见www.book118.com

但当两个人同时请求该JSP页面时发生了什么呢？（现在应该知道为什么加入Thread.sleep(250)——考虑到冲突因素给出充分的延迟。）图8-2和图8-3显示两个请求被同时处理的情况。

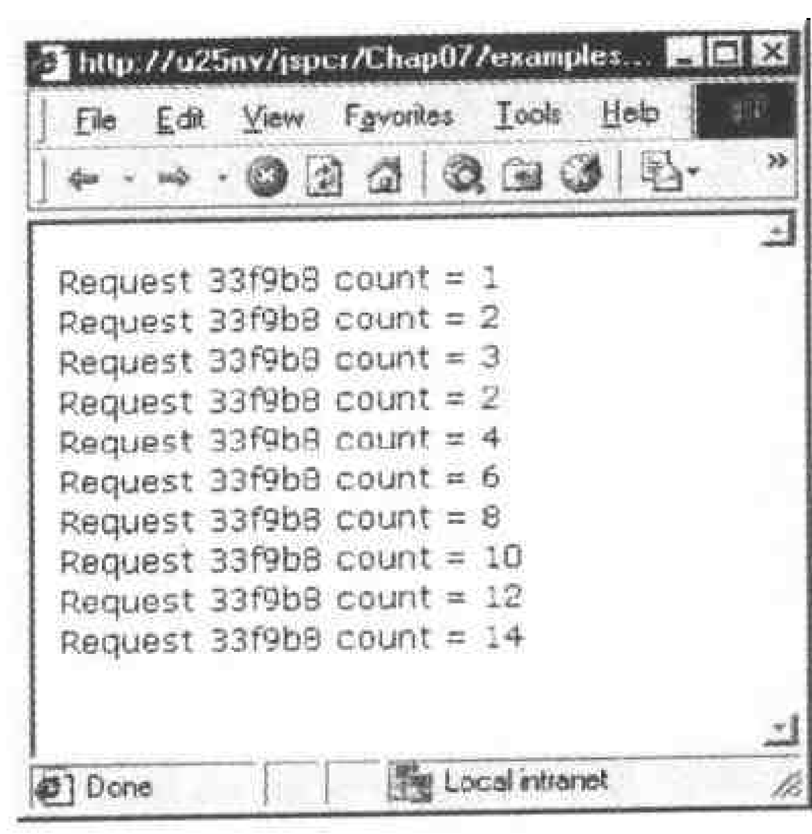


图8-2 vardec2.jsp处理请求33F9B8

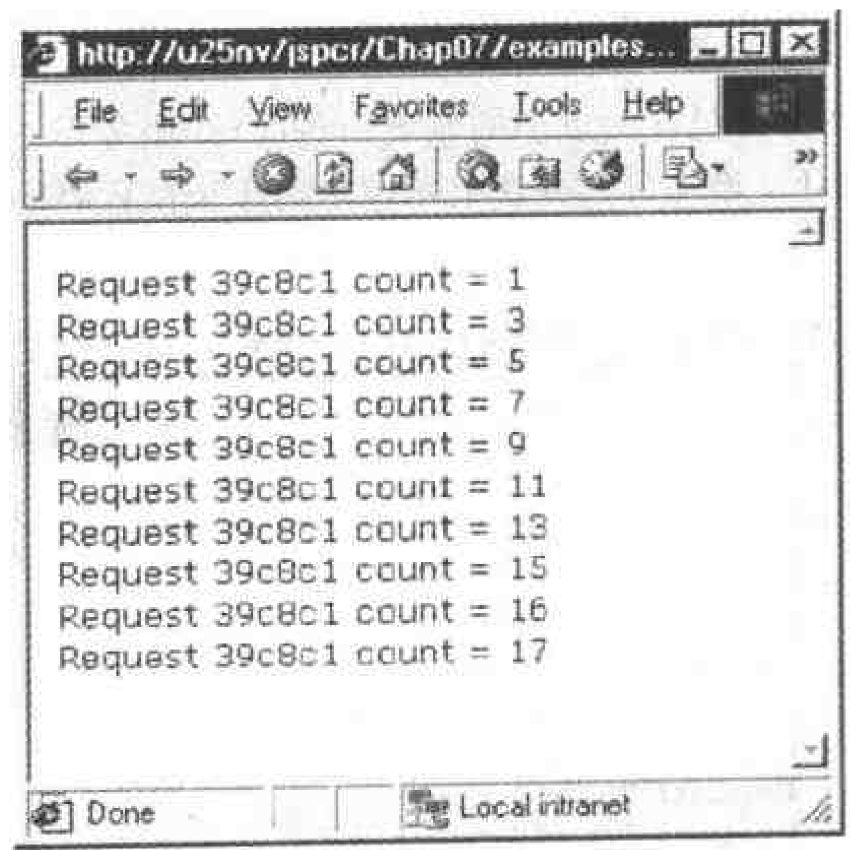


图8-3 vardec2.jsp处理请求39C8C1

第一个请求在前3行正常，然后计数降到2，对循环其余部分显示增量值为2。类似的，第二个请求从1开始，然后跳过所有奇数。一个对生成源码的检验说明了问题的所在：

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

精彩好书见www.book118.com


```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class vardec2 extends HttpJspBase
{

   // begin [file="vardec2.jsp";from=(0,3);to=(0,15)]
        int count;
        // end
        static
        {
        }

        public vardec2()
        {
        }

        public void _jspService(
              HttpServletRequest request,
              HttpServletResponse response)
           throws IOException, ServletException
        {
          JspFactory _jspxFactory = null;
          PageContext pageContext = null;
          HttpSession session = null;
          ServletContext application = null;
          ServletConfig config = null;
          JspWriter out = null;
          Object page = this;
          String _value = null;
          try {
             _jspxFactory = JspFactory.getDefaultFactory();
             response.setContentType("text/html;charset=8859_1");
             pageContext = _jspxFactory.getPageContext
                 (this, request, response, "", true, 8192, true);
```

精彩好书见www.book118.com


```
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            out.write("\r\n");

            count = 0;
            for (int i = 0; i < 10; i++) {
               out.write("\r\nRequest ");
               out.print( Integer.toHexString(request.hashCode()) );
               out.write("\r\ncount = ");
               out.print( ++count );
               out.write("<BR>\r\n");
               Thread.sleep(250);
               }
               out.write("\r\n");

            }
            catch (Exception ex) {
               if (out.getBufferSize() != 0)
                 out.clearBuffer();
               pageContext.handlePageException(ex);
            }
            finally {
               out.flush();
               _jspxFactory.releasePageContext(pageContext);
            }
      }
   }
```

问题的原因是在_jspService（）方法中count是一个实例变量，不是一个局部变量。JSP页面是作为servlet被编译，缺省地，作为一个单一实例运行，使用单线程处理每一请求。这种情况下，任何实例变量在所有请求处理线程间是自动共享的。在例子中，第一个请求到了循环3，但是然后处理第二个请求的线程进入_jspService（）方法，重置共享的count变量为0，随着循环继续，两个线程每125毫秒交替运行，并分别增加计数值。

第14章讨论了此问题，并给出几种解决方案。这里给出的结论是JSP页面中变量的声明最好只用于处理只读变量。

8.4 方法定义

声明一般用途是定义附加的方法。其语法与定义其他方法没有什么差别，只是使用了<%!

精彩好书见www.book118.com

和%>分隔符：

```
<%!
    public int sum(int a, int b)
    {
    return a + b;
    }
%>
```

像变量声明一样，方法定义作为_jspService（ ）方法外的高层成员被原封不动地复制到生成的servlet中。

```
public class methdef1 extends HttpJspBase
{
   // begin [file="methdef1.jsp";from=(0,3);to=(5,0)]

        public int sum(int a, int b)
        {
           return a + b;
        }
   // end

   // ...

   public void _jspService(
          HttpServletRequest request,
          HttpServletResponse response)
      throws IOException, ServletException
   {
      // ...
   }
}
```

一个JSP声明中典型的方法定义是针对由scriptlet产生的重新格式化的方法。考虑下面JSP页面，其在一个HTML表格中显示几个系统属性值：

```
<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
<%
   String[] propNames = {
      "java.awt.printerjob",
      "java.class.path",
      "java.class.version",
      "java.ext.dirs",
      "java.library.path",
   };
   for (int i = 0; i < propNames.length; i++) {
      String name = propNames[i];
      String value = System.getProperty(name);
```

精彩好书见www.book118.com


```
%>
<TR>
    <TD ALIGN=LEFT VALIGN=TOP><%= name %></TD>
    <TD ALIGN=LEFT VALIGN=TOP><%= value %></TD>
</TR>
<%
    }
%>
</TABLE>
```

此JSP页面输出如图8-4所示。此表格问题是几个取值太长，不带嵌入的空格。这意味着右边的表格单元太长，不能显示在窗口内。

解决此问题的一个简单方案是缩短属性值字符串。看起来另人讨厌的表格成员是由那些由分号分隔多个值的情况组成。在每个分号后加入一个
将其缩短，这样，列表就可以在多行显示，表格宽度需求不会超过最长列表表项。在scriptlet行内代码中实现此功能：

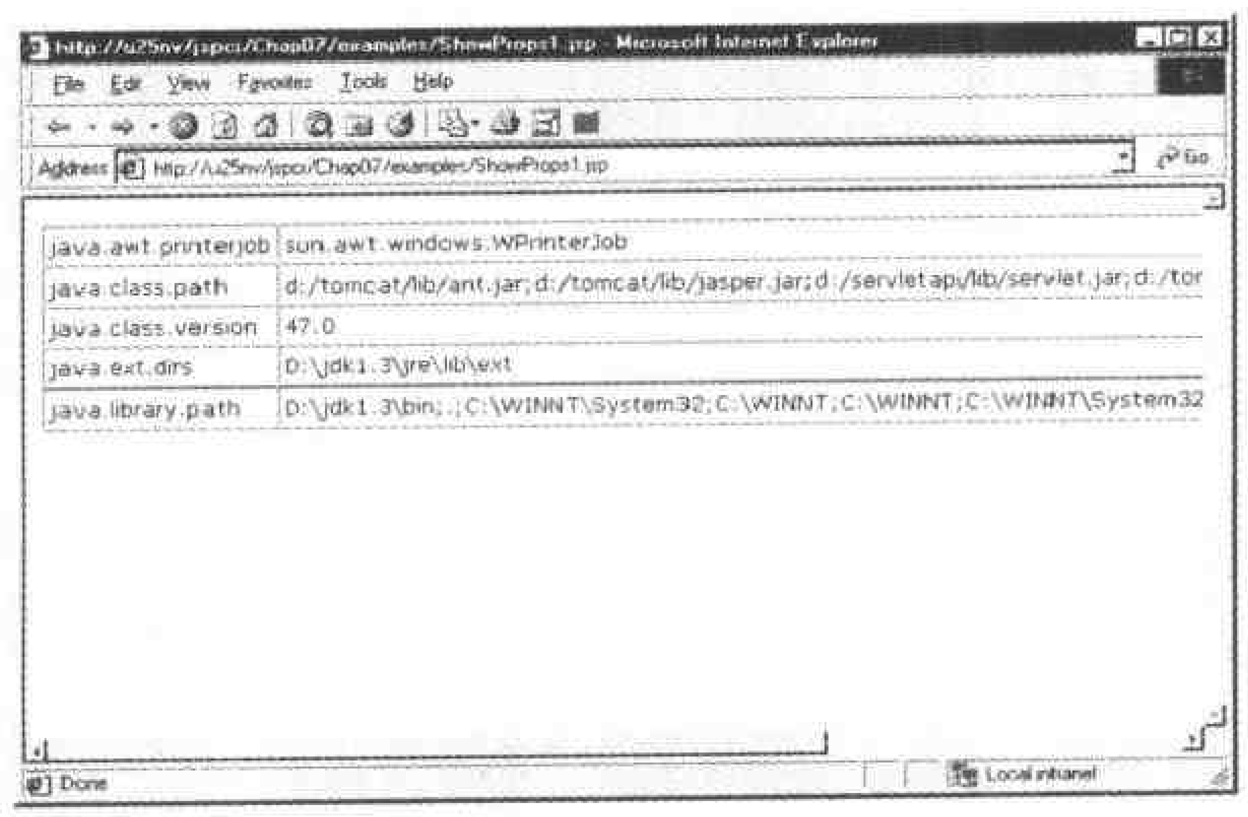


图8-4 具有宽单元的表格

但是一种更加可读的方案是应用了必要转换的normalize ( ) 方法。此方式下，<%=value%> 可简单写做<%=normalize(value)%>。下面是修改的JSP页面：

```
<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
<%
    String[] propNames = {
        "java.awt.printerjob",
        "java.class.path",
        "java.class.version",
        "java.ext.dirs",
        "java.library.path",
    };
```

精彩好书见www.book118.com


```
    for (int i = 0; i < propNames.length; i++) {
        String name = propNames[i];
        String value = System.getProperty(name);
%>
<TR>
    <TD ALIGN=LEFT VALIGN=TOP><%= name %></TD>
    <TD ALIGN=LEFT VALIGN-TOP><%= normalize(value) %></TD>
</TR>
<%
    }
%>
</TABLE>
<%!
    private static final String normalize(String s)
    {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            sb.append(c);
            if (c == ';')
                sb.append("<BR>");
        }
        return sb.toString();
    }
%>
```

这一次，当显示同样的属性时，表格更加适合窗口大小（见图8-5）。

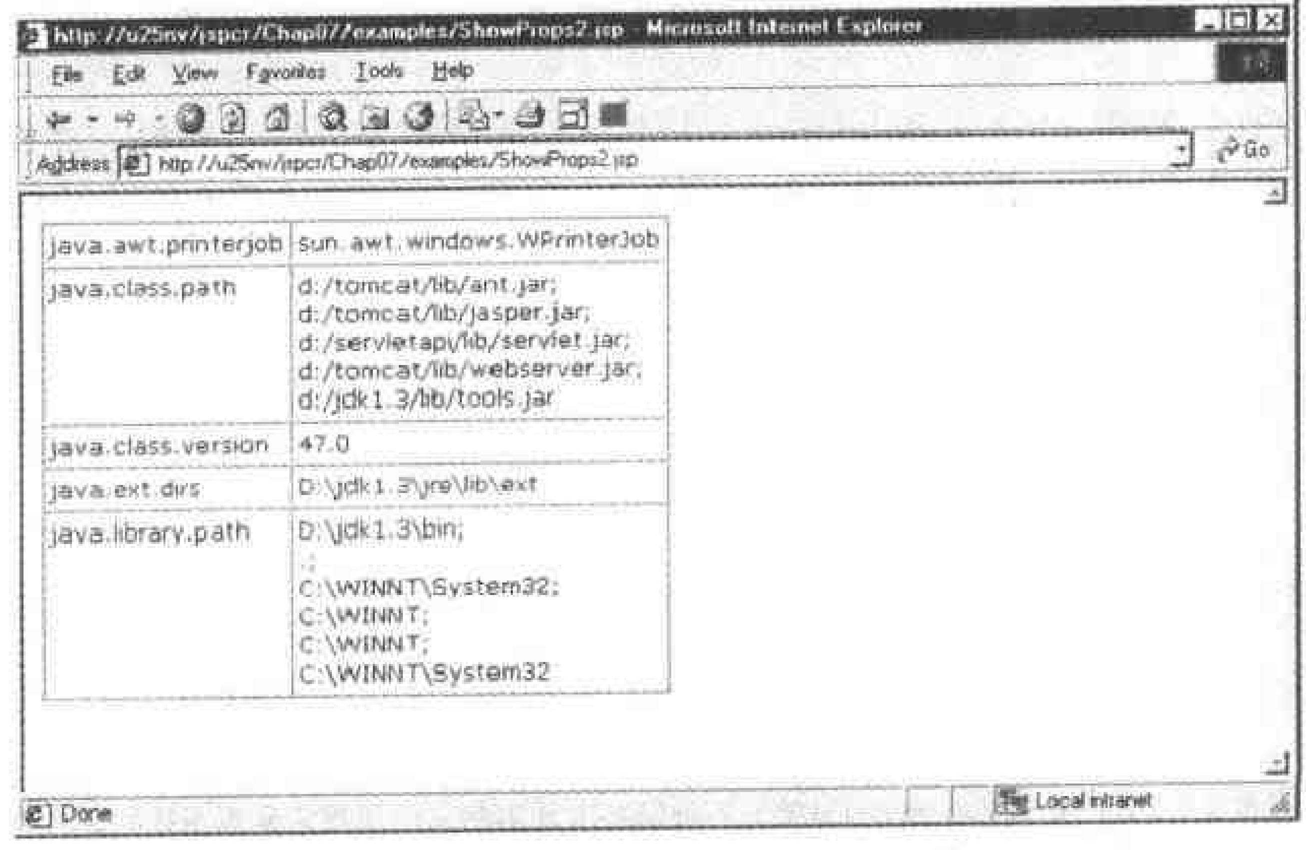


图8-5 具有标准化单元的相同表格

精彩好书见www.book118.com

8.4.1 覆盖jspInit和jspDestroy

在前面例子中，字符串表示可在scriptlet行内代码中实现，而不是通过方法调用。在一些情况下，这是不可能的。例如，当载入一个JSP页面时需要获得资源或启动线程，这些功能就应该在servlet的init（）和destroy（）方法的上下文中执行。

JSP 1.1规范禁止页面作者直接覆盖任何servlet生命期中的方法，包括init（）和destroy（）[1]。然而，有两种特殊的方法可以达到此目的，名字为jspInit（）和jspDestroy（）。这两种方法在init（）和destroy（）中自动调用，在父JSP页面实现中具有空定义。例如，在Tomcat内，基JSP类org.apache.jasper.runtime.HttpJspBase定义init（）和jspInit（）如下：

```
public final void init(ServletConfig config)
   throws ServletException
   {
      this.config = config;
      jspInit();
   }
public void jspInit()
{
}
```

它类似定义了destroy（）和jspDestroy（），如下：

```
public final void destroy()
{
   jspDestroy();
}
public void jspDestroy()
{
}
```

使用关键字final确保init（）和destroy（）本身不能被覆盖。反过来说，亦即jspInit（）和jspDestroy（）总是被调用。为了向JSP初始化段加入内容，应在一个JSP声明[2]中输入下列代码：

```
public void jspInit()
{
   TimerThread t = new TimerThread();
   t.start();
}
```

8.4.2 隐含对象的访问

与scriptlet和表达式不同，声明没有访问第7章讲述的隐含对象的权利。原因很明显，声明中

1 见JSP 1.1规范第3.1节。虽然某些servlet引擎并不强制要求此限制，但忽视它是不明智的。

2 十分奇怪，JSP 1.1规范并未提供从jspInit（）中产生溢出，即使init（）也可以这样做。在jspInit（）执行过程中如果JSP检测出一个致命错误时，应执行什么操作不是很明确。

精彩好书见www.book118.com

的方法定义在_jspService（ ）方法之外。因此，如果一个声明方法需要访问一个或多个这类对象，该对象必须从_jspService（ ）中传递过来。以下几种方式可以实现此功能：

- 作为个别参数传递对象。这很容易做，但如果需要多个参数就显得很笨拙。
- 作为参数传递pageContext隐含对象。从页面上下文中，所有其他变量均可以非直接地被访问。
- 作为单个参数传递包含所有感兴趣变量的一个结构，将会在下一节讲述。

第二种技术如下列代码所示：

```
<%@ page import="java.io.*,java.util.*" %>
<%!
   public void showSessionID(PageContext pc)
      throws IOException
   {
      JspWriter out = pc.getOut();
      HttpSession session = pc.getSession();
      Date created = new Date(session.getCreationTime());
      out.println("The session was created at " + created);
   }
%>
<%
   showSessionID(pageContext);
%>
```

showSessionID（ ）方法能够从页面上下文中提取JspWriter和HttpSession对象，并使用它们写入到当前输出流。

8.5 内部类

像其他Java类一样，JSP页面可以定义内部类。内部类对于运行后台进程或封装数据结构非常有用。使用正确时，它们可以保留一个Java程序的面向对象的字符，而这些字符有时在事件驱动环境里会丢失如JSP。

内部类作为包含隐含的和其他变量的一个数据结构可能很有用。其页面上下文充当了servlet上下文、会话、请求和页面中其他对象的一个包装器。它也包含针对附加的用户定义域的getAttribute（ ）和setAttribute（ ）方法，但它们必须是对象（不是如int和double的伪指令），被检索时必须置入适当类型。内部类是解决类型安全方式中同样问题的另一方案。此技术例子如下：

```
<%@ page import="java.io.*,java.util.*" %>
<%!
   /**
   * Inner class for passing parameters between methods
   */
   class Parameters {
      JspWriter out;
```

精彩好书见www.book118.com


```
        HttpSession session;
        String url;
    }

    public void showSessionID(Parameters parms)
        throws IOException
    {
        JspWriter out            = parms.out;
        HttpSession session      = parms.session;
        String url               = parms.url;
        Date created = new Date(session.getCreationTime());
        out.println("The session was created at " + created
            + "<P>");
        out.println("The url parameter was " + url);
    }
%>
<%
    Parameters parms = new Parameters();
    parms.out = out;
    parms.session = session;
    parms.url = request.getParameter("url");
    showSessionID(parms);
%>
```

生成的servlet在类的顶层和_jspService（）方法中scriptlet同时包含内部类和方法定义：

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.io.*;
import java.util.*;

public class PassInnerClass extends HttpJspBase
{
    // begin [file="PassInnerClass.jsp";from=(1,3);to=(22,0)]

        /**
         * Inner class for passing parameters between methods
```

精彩好书见www.book118.com


```
    */
    class Parameters {
        JspWriter out;
        HttpSession session;
        String url;
    }

    public void showSessionID(Parameters parms)
    throws IOException
    {
        JspWriter out            = parms.out;
        HttpSession session      = parms.session;
        String url               = parms.url;
        Date created = new Date(session.getCreationTime());
        out.println("The session was created at " + created
            + "<P>");
        out.println("The url parameter was " + url);
    }
// end

// ...

public void _jspService(
        HttpServletRequest request,
        HttpServletResponse response)
    throws IOException, ServletException
{
    // ...

    // begin [file="PassInnerClass.jsp";from=(23,2);to=(29,0)]

    Parameters parms = new Parameters();
    parms.out = out;
    parms.session = session;
    parms.url = request.getParameter("url");
    showSessionID(parms);

    // end
}
}
```

8.6 小结

第7章讲述了scriptlet和表达式，这一章介绍了第三种脚本元素，JSP声明。

像scriptlet，一个声明用于将Java语句嵌入到JSP页面内。两者关键的区别是JSP容器在生成

精彩好书见www.book118.com

的servlet中写入代码的位置。对于scriptlet，代码成为_jspService（ ）方法的一部分，而声明代码变成servlet类的高层代码。此差异对理解很重要，因为它影响到代码实施操作的上下文。

声明有三个基本用途：

- 变量声明　类和实例变量均可被定义，但同时必须小心变量的写权限。因为servlet缺省为多线程的。变量声明的最实际用途是针对静态final常量。
- 方法定义　利用JSP声明可以向生成的servlet加入附加方法。因为生成的代码不在_jspService（ ）方法中，它并没有权限访问隐含变量（request，response，out等等）。如果要使用它们，必须将之显式传递到方法中。声明可用于覆盖jspInit（ ）和jspDestroy（ ）方法。
- 内部类　声明给出编写内部类的简便方式。本章使用内部类作为在生成的servlet方法间传递变量集的一个数据结构。

精彩好书见www.book118.com

第9章　请求发送

具有多服务器组件处理的HTTP请求的大规模Web开发项目是人们所期望的，原因如下：

- 消除冗余性　一个Web站点的许多特征对所有页面是通用的，如头标和脚注、导航条目和叶脉内设计的其他元素。不用复制生成这些特性的HTML，能够一次写完就在许多地方使用它们很有用处。
- 内容和表示的分离　因为Java可用做JSP的任意一部分，可以很容易地完成生成信息和表示它们的代码。表示可能是读取数据库、执行计算和生成HTML表格。改变页面的逻辑和表示可能对以后是必须的。这样的代码很快就会变得很复杂。使这一切变得有意义的是从创建输出Web页面的JSP代码中分离出访问数据库和应用事务逻辑的纯粹的Java代码。

本章介绍允许转发请求和包含其他资源的内容和输出的JSP环境的特性。本章还讨论了RequestDispatcher类的工作方式，结束部分比较了两种JSP开发模型。

9.1　请求过程的剖析

处理servlet和JSP请求的servlet引擎可能是Web服务器本身的一部分（指in-process模式），或者是运行在一单独进程中。后一种情况下，Web服务器包含指向一个连接者的组件。connector解释servlet请求并通过与实现相关的协议[1]将之传递到servlet引擎。Web服务器像通常一样处理其他请求。图9-1阐述了这种out-of-process模式。

当servlet引擎收到一个请求，它将请求所有的细节汇编到一个HttpServletRequest对象。细节包括请求头标、URI、查询字符串和任意发送的参数等等。类似地，它初始化一个处理响应头标和响应输出流的HttpServletResponse对象，然后调用servlet的service（）方法（如果servlet是一个JSP，则为_jspService（）方法），向其传递两个对象的引用。如图9-2所示。

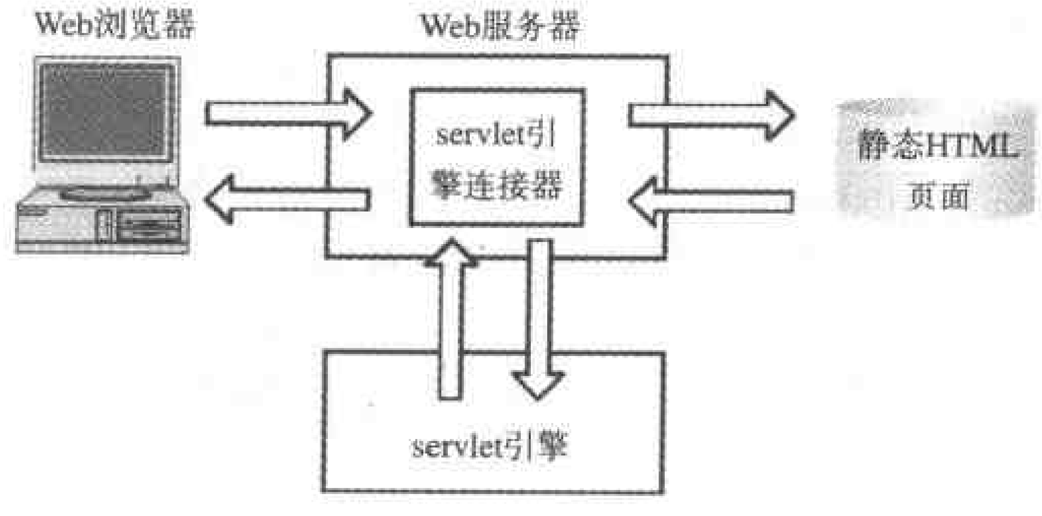


图9-1　out-of-process servlet引擎模型

1　JRun特性化其广泛用于Web服务器的连接者并采纳一种私有协议使请求与其servlet引擎通信。Tomcat使用一种称为ajp12的协议（最初为Apache JServ开发）在组件之间发送请求和响应。

精彩好书见www.book118.com

一个简单的JSP可以从请求对象中抽取所需内容。执行必要的计算和其他逻辑，然后使用响应对象创建输出。本章剩余部分介绍复杂的Web应用如何对这些请求和响应对象实施操作，并通过多个servlet或JSP传递它们。

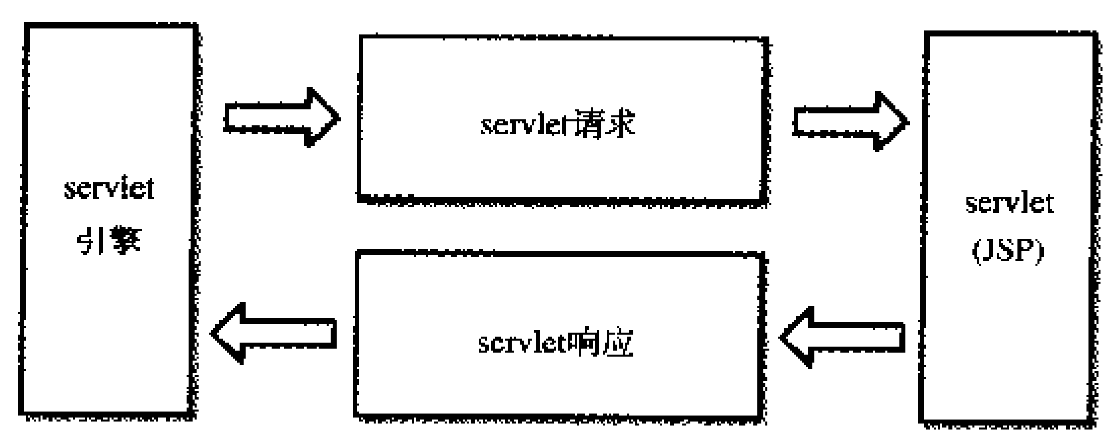


图9-2　将请求和响应对象传递到一个servlet的servlet引擎

9.2　包含其他资源

HTML本身没有直接在其输出中包含来自其他文件的数据的方式。这很不幸，因为大量的HTML标记对一个典型的Web站点中大量的页面都是通用的——公司标志、版权声明、导航、链接和其他特性。除了文本和图像为静态资源，也需要包含动态内容。JSP提供合并这些数据的两种方式：

- <%@ include%>伪指令　用于在JSP源码被转换成Java servlet源码和被编译前将静态文本复制到其中。典型情况下，文本为HTML代码，但它可以是在JSP页面内显示的任意内容。
- <jsp:include>行为　使得servlet引擎调用另一URL，生成带有最初JSP页面的输出。

在构建一种思维模式时关键的一点是<%@ include%>伪指令在编译时只执行一次，而<jsp:include>行为每次进行请求时都执行。下面两节讲解这两种JSP组件及其操作方式。

9.3　include伪指令

include伪指令的语法如下：

```
<%@ include file="filename" %>
```

被包含的文件名必须是相对的URL文档，即它只包含路径信息，没有协议或服务器信息。因此，只有当前servlet上下文中的资源可以用这种方式被包含。

如果文件名以“/”开始，它被认为是相对servlet上下文根的绝对路径。否则，文件名被认为是相对于当前JSP页面的。例如，如果一个Web应用有一个products子目录，products/search.jsp页面包含伪指令：

```
<%@ include file="/includes/header.inc" %>
```

那么被包含的文件是<path>/includes/header.inc，这里<path>是Web应用的mount点。如果伪指令为：

```
<%@ include file="includes/header.inc" %>
```

精彩好书见www.book118.com

那么，文件为<path>/products/includes/header.inc。

9.3.1 其工作方式

当遇到<%@ include%>伪指令，JSP容器读取指定文件，将其内容并入当前被解析的JSP源码中。例如，如果flavors.jsp包含

```
<H3>Flavors</H3>
Our most popular flavors are:
<%@ include file="flavor_list.html" %>
Try them all!
```

flavor_list.html包含

```
<OL>
<LI>Chocolate
<LI>Strawberry
<LI>Vanilla
</OL>
```

发送到Web服务器的HTML与flavors.jsp包含的完全一样：

```
<H3>Flavors</H3>
Our most popular flavors are:
<OL>
<LI>Chocolate
<LI>Strawberry
<LI>Vanilla
</OL>
Try them all!
```

可以看到Tomcat参考实现方案[1]生成的servlet源码中两个文件的交叉存取。

```
// begin [file="flavors.jsp";from=(0,0);to=(2,0)]
   out.write("<H3>Flavors</H3>\r\n");
   out.write("Our most popular flavors are:\r\n");
// end

// begin [file="flavor_list.html";from=(0,0);to=(5,0)]
   out.write("<OL>\r\n");
   out.write("<LI>Chocolate\r\n");
   out.write("<LI>Strawberry\r\n");
   out.write("<LI>Vanilla\r\n");
   out.write("</OL>\r\n");
// end

// begin [file="flavors.jsp";from=(2,38);to=(4,0)]
```

1 生成源码的实例为增加可读性重新进行了一些格式化。

精彩好书见www.book118.com


```
    out.write("\r\nTry them all!\r\n");
// end
```

与在注释中改变文件名不同，没有方式可以分辨出排序列表不是在原始JSP页面中被编码。从这一点上讲，<%@ include%>伪指令类似于C语言的#include预处理伪指令。

9.3.2 改变一个被包含文件的影响

如果flavor_list.html改动了会发生什么？JSP 1.1规范并没有规定要通知JSP容器一个包含文件已改变，但它不禁止这样做，并且一个健壮的JSP容器也应该这样做。JRun将附属名和最后修改时间并入生成的源码以便它可以判断文件何时超期：

```
private static final String[] __dependencies__ = {
   "/Chap09/examples/flavors.jsp",
   "/Chap09/examples/flavor_list.html",
   null
   };

private static final long[] __times__ = {
   958963142531L,
   958961380337L,
   0L
   };
```

关键一点是包含的文件是在编译时存在的文件，因为正好在此时处理<%@ include%>伪指令。这就是文件名为什么不能在运行时表示的原因。这也意味着包含的文件必须在编译时存在。

9.3.3 使用include伪指令复制源码

除了用于复制HTML，include伪指令还可用于像一个声明段一样包含Java源码。例如，常用的功能函数被保存在一个文件中，并使用include伪指令并入JSP页面。一个典型的例子是在HTML中过滤出带有特殊含义字符并用等价印刷体符号替代的函数：

```
<%!
public static final String webify(String s)
{
   StringBuffer sb = new StringBuffer();
   int n = s.length();
   for (int i = 0; i < n; i++) {
      char c = s.charAt(i);
      switch (c) {
         case '<': sb.append("&lt;"); break;
         case '>': sb.append("&gt;"); break;
         case '&': sb.append("&amp;"); break;
         case '"': sb.append("&quot;"); break;
```

精彩好书见www.book118.com


```
            default: sb.append(c); break;
         }
      }
      return sb.toString();
   }
%>
```

一旦定义好（被包含），此函数在包含的JSP页面的scriptlet和表达式中就可以使用了。

```
<%@ include file="webify.jsp" %>

Preformatted text can be coded with the
<%= webify("<PRE> and </PRE>") %> tags.
```

类似的，一个应用使用的常量可以在由include伪指令处理的JSP声明中被编码：

```
<%!
   static final int BORDER = 1;
   static final int CELLPADDING = 3;
   static final int CELLSPACING = 0;
   static final String[] COLORS = {"#C0C0C0", "#E0E0E0"};
%>
```

如果前面显示的声明保存在TableConstants.jsp文件中，那么一个JSP页面可以生成一个行之间交替背景颜色的表格，如下：

```
<%@ include file="TableConstants.jsp" %>

<TABLE BORDER="<%= BORDER %>"
       CELLPADDING="<%= CELLPADDING %>"
       CELLSPACING="<%= CELLSPACING %>"
       >
<%
   for (int i = 0; i < 10; i++) {
      int x = i+1;
      int xsq = x*x;
%>
<TR BGCOLOR="<%= COLORS[i % 2] %>">
   <TD><%= x %></TD> <TD><%= xsq %></TD>
</TR>
<%
   }
%>
</TABLE>
```

注意 对源码声明使用include伪指令时要注意两件事情：第一，JSP 1.1规范并不确保以这种方式包含代码的页面当发生改变时被注意到。第二，被包含代码使用了包含页面的名空间，因此必须要小心确保没有重复变量定义发生。

精彩好书见www.book118.com

9.4 <jsp:include>行为

对比于include伪指令，jsp:include行为在每次进行请求时被解释。此行为语法为：

```
<jsp:include page="resourcename" flush="true" />
```

被包含的资源名必须是相对的URL文档，即它只包含路径信息，以在include伪指令中个文件名同样的方式映射到当前servlet上下文中。如果文件名以“/”开始，它被认为是从servlet上下文根开始的一个路径。否则，文件名被认为是包含调用JSP的目录的相对路径。刷新属性（手工进行）用于指出是否在包含资源之前刷新输出JspWriter。在JSP 1.1中惟一有效值是true。

工作方式

<jsp:include>行为被JSP编译器解析，但不在编译时执行，它在请求时被转换成调用已命名资源的Java代码。资源可以是静态文本如一个HTML文件，或一个动态资源，如一个JSP页面或servlet。回到冰激凌滋味例子中，如果flavors.jsp包含

```
<H3>Flavors</H3>
Our most popular flavors are:
<jsp:include page="/servlet/FlavorList" flush="true" />
Try them all!
```

这里FlavorList是一个从数据库或其他动态资源中抽取喜欢滋味的servlet：

```
import java.io.*;
import java.net.*;
import java.sql.*;
import java.util.*;

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

/**
* Returns the current list of favorite flavors
*/
public class FlavorListServlet extends HttpServlet
{
   public static final String JDBC_DRIVER =
      "sun.jdbc.odbc.JdbcOdbcDriver";

   public static final String URL =
      "jdbc:odbc:IceCream";

   public void doGet(
         HttpServletRequest request,
         HttpServletResponse response)
      throws ServletException, IOException
```

精彩好书见www.book118.com


```
{
    PrintWriter out = response.getWriter();
    Connection con = null;
    try {

      // Connect to the ice cream database

      Class.forName(JDBC_DRIVER);
      con = DriverManager.getConnection(URL);

      // Run a query to get the top flavors

      Statement stmt = con.createStatement();
      String sql =
        "SELECT RANK, NAME"
        + " FROM flavors"
        + " WHERE (RANK <= 3)"
        + " ORDER BY RANK"
        ;

      ResultSet rs = stmt.executeQuery(sql);

      // Print as an ordered list

      out.println("<OL>");
      while (rs.next()) {
        int rank = rs.getInt(1);
        String name = rs.getString(2);
        out.println(" <LI>" + name);
      }
      out.println("</OL>");
    }
    catch (Exception e) {
      e.printStackTrace();
    }

    // Close the database

    finally {
      if (con != null) {
        try {
          con.close();
        }
        catch (SQLException ignore) {}
      }
    }
```

精彩好书见www.book118.com


```
    }
}
```

当调用flavors.jsp，产生输出如下：

```
<H3>Flavors</H3>
Our most popular flavors are:
<OL>
   <LI>Espresso Chip
   <LI>Orange Cream
   <LI>Peanut Butter
</OL>
Try them all!
```

结果HTML很类似，但底层机制完全不同。它可以在Tomcat生成的servlet源码中被看到：

```
// begin [file="flavors.jsp";from=(0,0);to=(2,0)]
   out.write("<H3>Flavors</H3>\r\n");
   out.write("Our most popular flavors are:\r\n");
// end

// begin [file="flavors.jsp";from=(2,0);to=(2,55)]
   {
      out.flush();
      pageContext.include("/servlet/FlavorList");
   }
// end

// begin [file="flavors.jsp";from=(2,55);to=(4,0)]
out.write("\r\nTry them all!\r\n");
// end
```

不是包含滋味次序列表，JSP调用pageContext.include（ ）方法运行访问数据库的servlet。servlet的输出被包含在JSP输出和JSP恢复控制中。include伪指令类似于C语言#include预处理伪指令，<jsp:include>行为更像一个C语言函数调用。

1. 限制

<jsp:include>行为调用的JSP页面可以访问所有对调用JSP可利用的隐含对象，包括response对象。它可以写入和刷新out对象，但它不能设置响应头标。例如，你不能指定一个不同的内容类型，也不能使用<jsp:include>行为处理带有WWW-Authenticate头标的验证，为什么？因为这太迟了——输出流在JSP被包含前已刷新，因此任何头标的表示已经写入客户端。

2. 运行时特性

因为一个<jsp:include>在运行时被执行，其指向的页面可以在运行表示时才提供，而不是被硬编码。下面JSP页面设计为一个HTTP servlet请求的理解示意。为防止过长，滚动属性名和取值列表，页面模拟了一个标签式的对话框，其属性被划分成逻辑组和用于选择显示哪一组页面上部的单选按钮。

精彩好书见www.book118.com


```
<%@ page import="java.util.*" %>
<%!
   // Table row colors

   static final String[] COLORS = {"#E0E0E0", "#F0F0F0"};

   // Array of tab codes, labels, and JSP's

   public static final String[][] TABS = {
      {"HD", "Headers", "ShowRequestHeaders.jsp"},
      {"PM", "Parameters", "ShowParameters.jsp"},
      {"SR", "ServletRequest Methods",
         "ShowServletRequestMethodValues.jsp"},
      {"HR", "HttpServletRequest Methods",
         "ShowHttpServletRequestMethodValues.jsp"},
   };
%>
<HTML>
<HEAD>
<TITLE>Show Request</TITLE>
</HEAD>
<BODY>
<H2>Show Request</H2>
<FORM>
<TABLE BORDER=0 CELLPADDING=3 CELLSPACING=0>
   <%-- Radio buttons for selecting the page --%>

   <TR>
      <TD ALIGN=LEFT>
<%
   String which = request.getParameter("which");
   if (which == null)
      which = TABS[0][0];
   String jspToRun = null;
   for (int i = 0; i < TABS.length; i++) {
      String tabCode = TABS[i][0];
      String tabLabel = TABS[i][1];
      String tabJSP = TABS[i][2];
      String CHECKED = "";
      if (which.equals(tabCode)) {
         CHECKED = "CHECKED";
         jspToRun = tabJSP;
      }
%>
   <INPUT NAME="which"
         TYPE="RADIO"
         VALUE="<%= tabCode %>"
         <%= CHECKED %>
```

精彩好书见www.book118.com


```
            onClick="this.form.submit()"
            ><%= tabLabel %>
<%
    }
%>


        <P>
        </TD>
    </TR>

    <TR>
        <TD ALIGN=CENTER VALIGN=TOP>

        <%-- Page showing details of the request --%>

        <jsp:include page="<%= jspToRun %>" flush="true" />

        <%-- Resulting table --%>

        <TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0 WIDTH=600>

        <TR>
            <TH COLSPAN=2 ALIGN=LEFT BGCOLOR="#000000">
                <FONT SIZE="+1" COLOR="#FFFFFF">
                <%= request.getAttribute("_table_title") %>
                </FONT>
            </TH>
        </TR>
        <TR>
            <TH WIDTH=200 ALIGN=LEFT>Name</TH>
            <TH WIDTH=400 ALIGN=LEFT>Value</TH>
        </TR>
        <%
            Map entries = (Map)
                request.getAttribute("_table_entries");
            Iterator iNames = entries.keySet().iterator();
            int row = 0;
            while (iNames.hasNext()) {
                String name = (String) iNames.next();
                Object value = entries.get(name);
        %>
        <TR BGCOLOR="<%= COLORS[row % 2] %>">
            <TD ALIGN=LEFT VALIGN=TOP><B><%= name %></B></TD>
            <TD ALIGN=LEFT VALIGN=TOP><%= value %></TD>
        </TR>
        <%
                row++;
            }
```

精彩好书见www.book118.com


```
            %>
            </TABLE>
            <P>
            </TD>
        </TR>

    </TABLE>
    </FORM>
    </BODY>
    </HTML>
```

显示可以利用的类别被编码成一个静态String数组。对每一类别存在两个字符的缩写：一个标签和抽取所需数据JSP页面的名字。共有4类属性：

- 请求头标
- 参数
- ServletRequest中方法
- HttpServletRequest中方法

字符串数组提供生成页面所需的所有信息。单选按钮被包含在一个自引用的HTML窗体中，在一个循环中生成。两个字符的缩写用做VALUE属性和可视文本的标签。当点击单选按钮，窗体被提交，按钮值提供了which参数的值。图9-3给出了原始显示，它是请求头标类别，当点击了另一按钮（例如，ServletRequest方法按钮），不同表格出现在表格体中（见图9-4）。

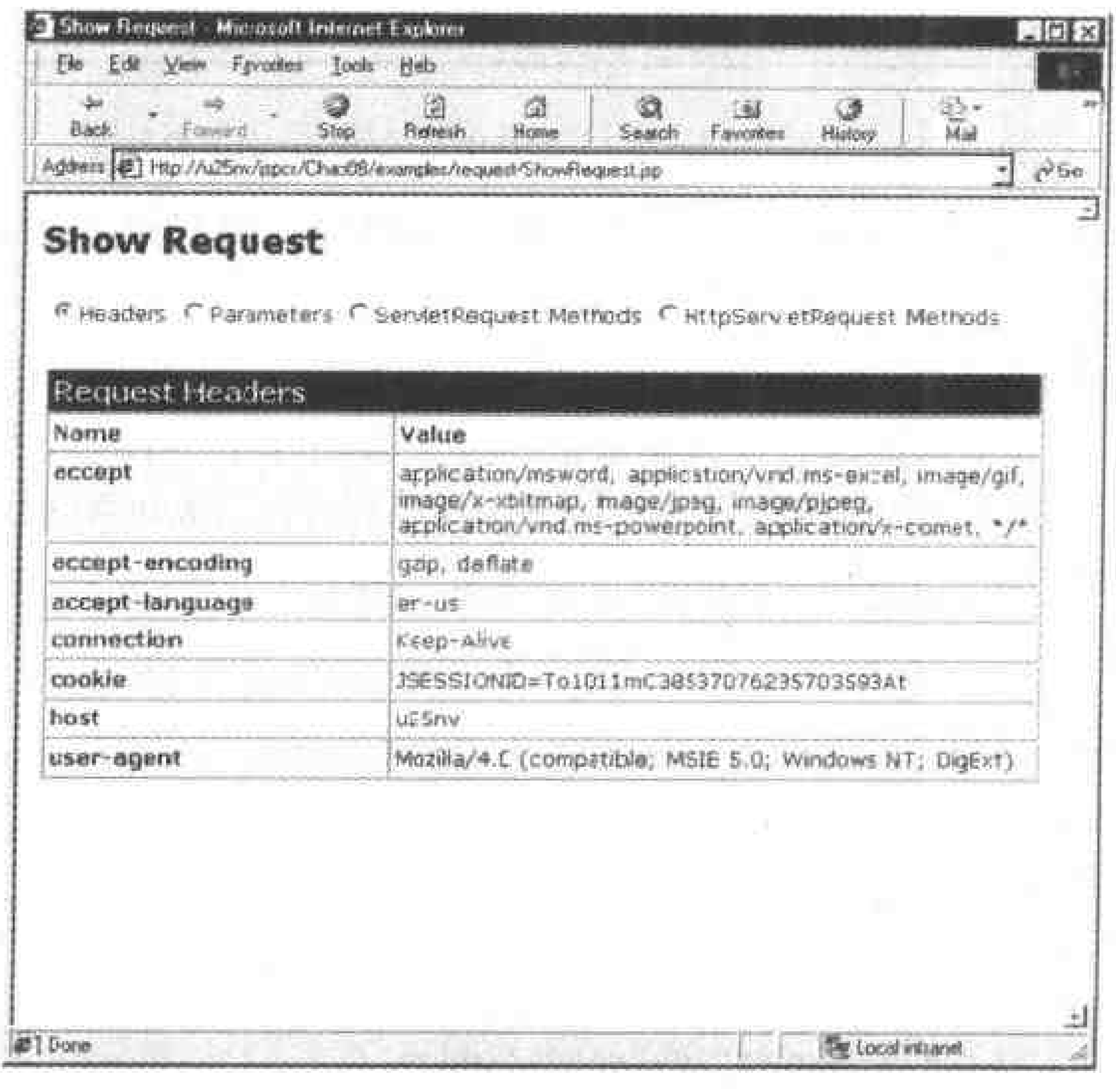
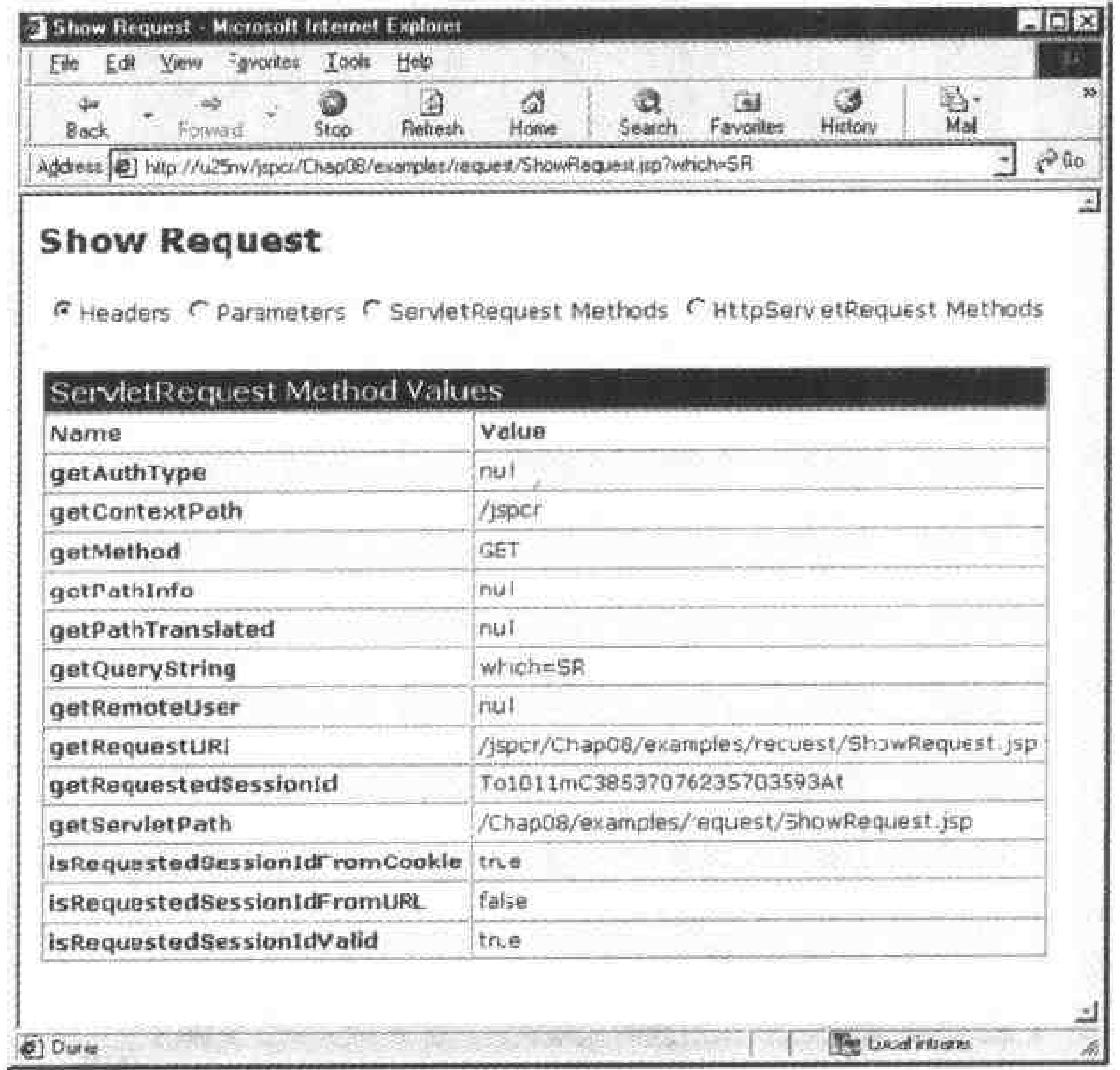


图9-3 ShowRequest.jsp 显示的HTTP请求头标

精彩好书见www.book118.com



Name	Value
getAuthType	null
getContextPath	/jspcr
getMethod	GET
getPathInfo	null
getPathTranslated	null
getQueryString	which=SR
getRemoteUser	null
getRequestURI	/jspcr/Chap08/examples/request/ShowRequest.jsp
getRequestedSessionId	To1011mC38537076235703593At
getServletPath	/Chap08/examples/request/ShowRequest.jsp
isRequestedSessionIdFromCookie	true
isRequestedSessionIdFromURL	false
isRequestedSessionIdValid	true

图9-4　ShowRequest.jsp 第3个标签中显示的Servlet请求方法

ShowRequest.jsp判断哪个单选按钮被点击，从字符串数组中选择相应的JSP文件名。然后此文件名被传入<jsp:include>行为的一个JSP表达式中：

```
<jsp:include page="<%= jspToRun %>" flush="true" />
```

生成页面的每一表格都创建属性名和取值列表，将其写入一个保存为一个请求属性的java.util.Map对象。表格头标字符串也被保存为一个请求属性。当被包含的JSP完成时，从请求属性中抽取映射并显示在一个HTML表格中。生成请求头标标签的JSP如下：

```
<%@ page import="java.util.*" %>
<%
   Enumeration eNames = request.getHeaderNames();
   if (eNames.hasMoreElements()) {
      String title = "Request Headers";
      Map entries = new TreeMap();
      while (eNames.hasMoreElements()) {
         String name = (String) eNames.nextElement();
         String value = request.getHeader(name);
         entries.put(name, value);
      }
```

精彩好书见www.book118.com


```
        request.setAttribute("_table_title", title);
        request.setAttribute("_table_entries", entries);
    }
%>
```

运行时信息选择被包含页面的功能是基于JSP的Web应用一个很有用的特性，因为它允许复杂过程构建在表格驱动逻辑上。

3. 向被包含JSP传递参数

可以通过<jsp:include>行为调用的JSP页面传递参数以提供额外定制信息。语法为：

```
<jsp:include page="pageName" flush="true">
<jsp:param name="parm1Name" value="parm1Value" />
<jsp:param name="parm2Name" value="parm2Value" />
</jsp:include>
```

向被包含的JSP传递参数与一般窗体参数一样，可以通过request.getParameter(name)检索。如果参数名与JSP正在使用的参数名一样，两个值都被传递，并可以使用getParameterValues(name)作为字符串数组进行检索。

下面JSP阐述了如何使用此技术。它两次包含同一页面，每次使用不同的参数。

```
<%
    // Diameter of the earth in kilometers

    int distance = 12756;
%>

<H4>Diameter of the Earth in SI (Metric) Units</H4>
<jsp:include page="ShowDiameter.jsp" flush="true">
    <jsp:param name="dist" value="<%= distance %>" />
    <jsp:param name="units" value="SI" />
</jsp:include>

<H4>Diameter of the Earth in U.S. Customary Units</H4>
<jsp:include page="ShowDiameter.jsp" flush="true">
    <jsp:param name="dist" value="<%= distance %>" />
    <jsp:param name="units" value="US" />
</jsp:include>
```

两个参数被传递：

- dist 每公里距离
- units 如果需要米单位，则为“SI”，否则为“US”

ShowDiameter.jsp页面检索公里距离，将之转换为一个整数，再找出等价的英里数。然后基于度量码单位按此单位参数进行传递，以SI或U.S.单位显示距离。

```
<%
    String dist = request.getParameter("dist");
```

精彩好书见www.book118.com


```
    int kilometers = Integer.parseInt(dist);
    double miles = kilometers / 1.609344;

    String units = request.getParameter("units");
    if (units.equals("SI")) {
    %> Diameter = <%= kilometers %> km <%
    }
    else {
    %> Diameter = <%= miles %> miles <%
    }
%>
```

图9-5显示了结果。

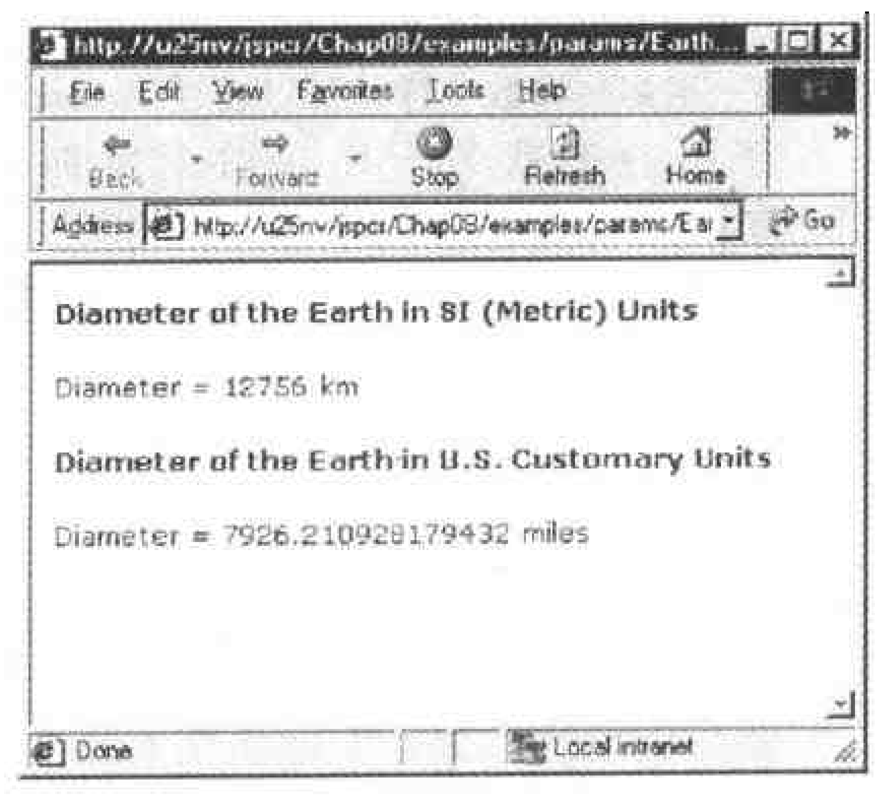


图9-5 以不同参数包含同一页面两次的一个JSP页面

4. 检索初始URI

当在一个<jsp:include>行为中调用页面时，它使用与包含页面相同的请求对象，即request.getRequestURI ( ) 和request.getServletPath ( ) 返回最初处理请求的页面路径，而不是当前页面的路径。然而，被包含页面的等价值可作为请求属性加以利用。在ShowPath1.jsp中示例如下：

```
<PRE>
In ShowPath1.jsp:

   request.getRequestURI()
      = <%= request.getRequestURI() %>

   request.getServletPath()
      = <%= request.getServletPath() %>
```

精彩好书见www.book118.com


```
</PRE>
<jsp:include page="ShowPath2.jsp" flush="true"/>
```

它包含的页面ShowPath2.jsp：

```
<PRE>
In ShowPath2.jsp:

   request.getRequestURI()
      = <%= request.getRequestURI() %>

   request.getServletPath()
      = <%= request.getServletPath() %>

   javax.servlet.include.request_uri
      = <%= request.getAttribute
      ("javax.servlet.include.request_uri") %>

   javax.servlet.include.servlet_path
      = <%= request.getAttribute
      ("javax.servlet.include.servlet_path") %>
</PRE>
```

两个页面的输出如下：

```
In ShowPath1.jsp:

   request.getRequestURI()
      = /jspcr/Chap09/examples/ShowPath1.jsp

   request.getServletPath()
      = /Chap09/examples/ShowPath1.jsp

In ShowPath2.jsp:

   request.getRequestURI()
      = /jspcr/Chap09/examples/ShowPath1.jsp

   request.getServletPath()
      = /Chap09/examples/ShowPath1.jsp

   javax.servlet.include.request_uri
      = /jspcr/Chap09/examples/ShowPath2.jsp

   javax.servlet.include.servlet_path
      = /Chap09/examples/ShowPath2.jsp
```

以此种方式检索的属性集显示在下表中。

精彩好书见www.book118.com

描述一个被包含的JSP页面的请求属性

属 性 名	等价方法
javax.servlet.include.request_uri	request.getRequestURI()
javax.servlet.include.context_path	request.getContextPath()
javax.servlet.include.servlet_path	request.getServletPath()
javax.servlet.include.path_info	request.getPathInfo()
javax.servlet.include.query_string	request.getQueryString()

9.5 使用哪种方法

include伪指令和<jsp:include>行为执行类似功能，各有优势。使用其中哪一种应该考虑包含是否需要在运行时进行。下面表格对比了两种选择：

规则	<%@ include%>	<jsp:include>
编译时间	较慢——资源必须被解析	稍快
执行时间	稍快	较慢——每次资源必须被解析
灵活性	较差——页面名字固定	更好——页面在运行时可以选择

9.6 转发请求

为实现将Web应用划分为内容和表示，JSP环境提供了<jsp:forward>行为，它允许从一个页面向另一个页面或servlet转发请求，语法为：

```
<jsp:forward page="page" />
```

这里，page是相对于当前页面的URI，或者是关于servlet上下文根的绝对URI。像<jsp:include>，<jsp:forward>行为对页面名字可以在运行时表示。类似地，它可以使用下列语法传递参数到新的JSP：

```
<jsp:forward page="page">
<jsp:param name="name_1" value="value_1" />
<jsp:param name="name_2" value="value_2" />
...
<jsp:param name="name_n" value="value_n" />
</jsp:forward>
```

当执行<jsp:forward>行为时，载入被命名页面，当前页面终止。新页面可以访问请求和响应对象，被期望创建所有的输出，因为转发页面不能编写任何输出。下述表格描述了使能或未使能输出缓存及缓存已满或未满时发生的动作。

缓存使能	缓存填充	动作
否	无	如果写入任何输出，则产生溢出IllegalStateException。
是	否	缓存在转发前被清除。
是	是	产生IllegalStateException溢出。

下列代码显示了请求转发的典型使用情况——分离内容和表示。第一个JSP页面是

精彩好书见www.book118.com

GetFoodGroups.jsp，它从USDA营养数据库中读取食物组列表。

```
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="java.io.*" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
<%@ page import="jspcr.forward.*" %>
<%
   // Load the driver class and establish a connection

   Class.forName
      ("sun.jdbc.odbc.JdbcOdbcDriver");

   Connection con = DriverManager.getConnection
      ("jdbc:odbc:usda");

   // Run a database query to get the list of food groups

   Statement stmt = con.createStatement();
   String sql =
        " SELECT FdGp_Cd, FdGp_Desc"
      + " FROM fd_group"
   ;
   ResultSet rs = stmt.executeQuery(sql);

   // Store the results as a list of FoodGroup objects

   List fglist = new ArrayList();
   while (rs.next()) {
      String code = rs.getString(1);
      String desc = rs.getString(2);
      FoodGroup fg = new FoodGroup(code, desc);
      fglist.add(fg);
   }

   rs.close();
   stmt.close();
   con.close();

   // Store the list as a request attribute

   request.setAttribute("jspcr.forward.FoodGroups", fglist);

   // Now forward the request

%><jsp:forward page="ShowFoodGroups.jsp" />
```

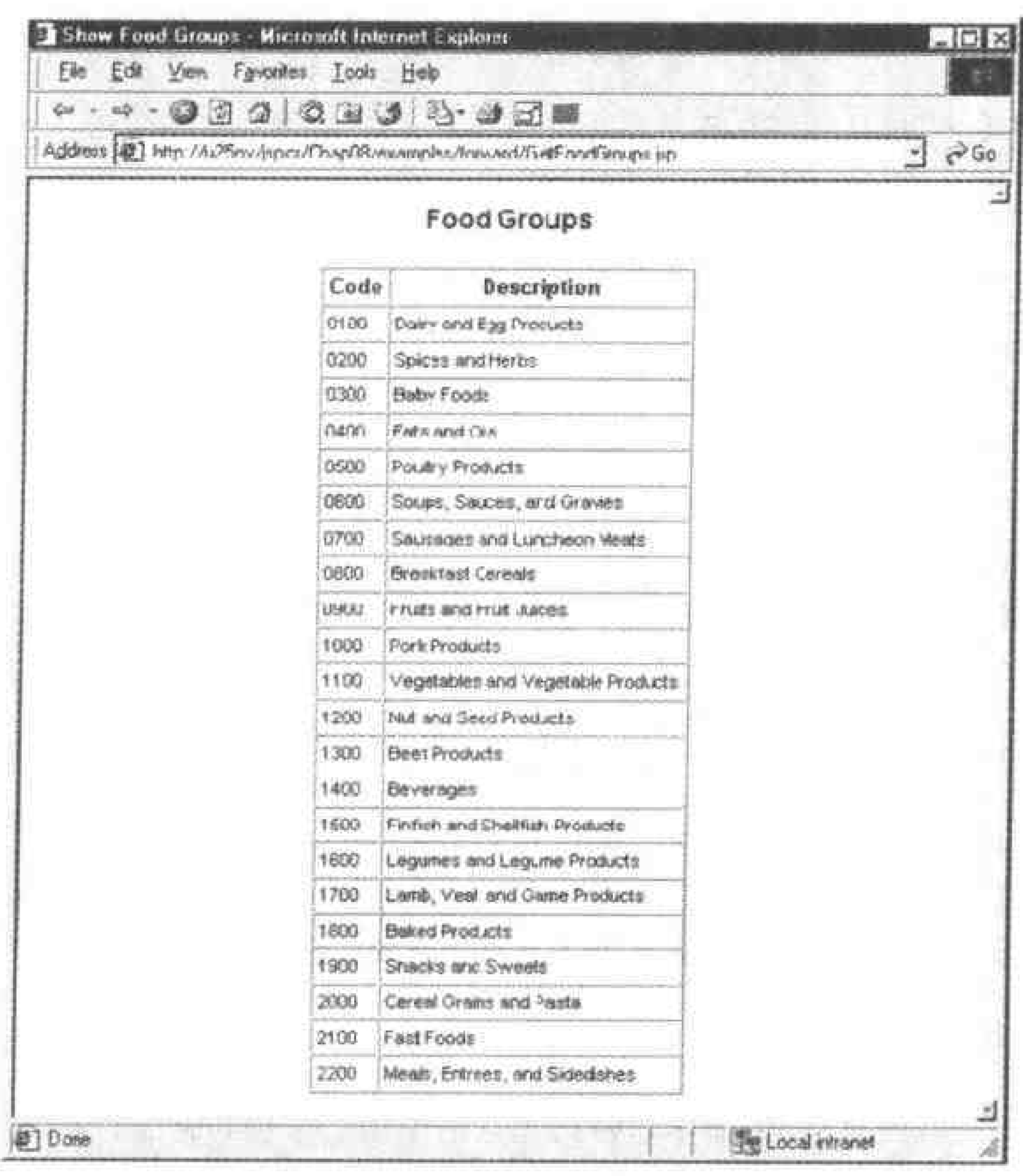
精彩好书见www.book118.com

当读取食物组记录时，将其存储在List结构中。列表被保存为请求的属性。当从数据库中抽取了所有记录，请求被转发到ShowFoodGroups.jsp，它检索列表并将其写成一HTML表格：

```
<%@ page import="java.io.*,java.util.*,jspcr.forward.*" %>

<HTML>
<HEAD>
<TITLE>Show Food Groups</TITLE>
<STYLE>
   body, td {
      background-color: #FFFFFF;
      font: 8pt Sans-Serif;
   }
</STYLE>
</HEAD>
<BODY>
<CENTER>
<H3>Food Groups</H3>

<%-- Get the list of FoodGroup objects
     that was created by database calls --%>

<%
   List fglist = (List) request.getAttribute
      ("jspcr.forward.FoodGroups");
   Iterator igroups = fglist.iterator();
%>

<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
<TR><TH>Code</TH><TH>Description</TH></TR>

<%-- Loop through the list and print each item --%>
<%
   while (igroups.hasNext()) {
      FoodGroup fg = (FoodGroup) igroups.next();
%>
<TR>
   <TD><%= fg.getCode() %></TD>
   <TD><%= fg.getDescription() %></TD>
</TR>
<%
   }
%>
</CENTER>
</BODY>
</TABLE>
```

精彩好书见www.book118.com

ShowFoodGroups.jsp的优势是可以单独被测试，而无需链接到数据库。可以编写一个用于测试的小的JSP，它只需熟悉List属性，ShowFoodGroups.jsp不知道它实际上并没有对数据库操作，结果如图9-6显示：



Code	Description
0100	Dairy and Egg Products
0200	Spices and Herbs
0300	Baby Foods
0400	Fats and Oils
0500	Poultry Products
0600	Soups, Sauces, and Gravies
0700	Sausages and Luncheon Meats
0800	Breakfast Cereals
0900	Fruits and Fruit Juices
1000	Pork Products
1100	Vegetables and Vegetable Products
1200	Nut and Seed Products
1300	Beef Products
1400	Beverages
1500	Finfish and Shellfish Products
1600	Legumes and Legume Products
1700	Lamb, Veal and Game Products
1800	Baked Products
1900	Snacks and Sweets
2000	Cereal Grains and Pasta
2100	Fast Foods
2200	Meals, Entrees, and Sidedishes

图9-6　一对JSP页面创建的食物组列表

9.7　RequestDispatcher对象

<jsp:include>和<jsp:forward>的底层机制都是javax.servlet.RequestDispatcher类。在前面一节食物组例子中，<jsp:forward>行为被Tomcat转换为下列代码：

```
if (true) {
   out.clear();
   String _jspx_qfStr = "";
   pageContext.forward("ShowFoodGroups.jsp" + _jspx_qfStr);
   return;
}
```

pageContext的Tomcat实现反过来调用一个RequestDispatcher处理转发：

精彩好书见www.book118.com


```
public void forward(String relativeUrlPath)
        throws ServletException, IOException
    {
        String path = getAbsolutePathRelativeToContext(relativeUrlPath);
        context.getRequestDispatcher(path).forward(request, response);
    }
```

创建RequestDispatcher有三种方式：

1）ServletContext.getRequestDispatcher(String path)

- 路径必须是相对上下文的绝对路径。
- 在另一servlet上下文的资源的发送器可以被创建，只要其上下文已知。上下文可通过context.getContext(otherContext)获得。

2）ServletContext.getNamedDispatcher(String name)

- 名字参数指向servlet别名，而不是一物理路径名。
- servlet可以使用config.getServletName()得到自己的名字。

3）ServletRequest. getRequestDispatcher(String name)

- 路径可以是相对上下文的绝对路径，或页面的相对路径。这是它与第一个方法的基本差别。

请求发送对比重定向

一个请求发送器的大部分工作也可以通过JSP和servlet写入"Moved Temporarily"或"Moved Permanently"状态码和在定位头标中写入的下一个JSP或servlet的URL完成。差别是重定向包括一个互相协作的客户端，而请求发送全部在服务器端处理，没有客户端交互。

9.8 模型1对比模型2

这些都是非常方便的特性。但如果只将其无目标地用于头标和脚注，则是未加以充分利用。实际上，它们可以是构建很好体系结构的一部分。如果你读过JSP新闻组，就经常会遇到模型1和模型2体系结构的引用，即在最初的JSP 0.92规范中引入的Web应用结构的两种不同方法。

在模型1应用中，JSP实现：

- 用户请求一个JSP页面
- JSP执行计算、数据库查询等等。
- JSP页面使用HTML表示其输出。

实现上述功能必要的Java代码能以scriptlet形式直接书写，或将其包含在JavaBean中。

模型2应用遵循MVC范例。MVC是一种面向对象的编程概念，此概念在Smalltalk语言中体现最显著。它将一个应用逻辑划分成三部分：

- 模型　是逻辑的“内部”表示法。它没有可视化输出并根本没有外部表示。正因为此，它可以很好地运行于servlet、单机版GUI、或一批测试程序。例如，象棋游戏的模型可以包含表示棋盘、每一子所代表的数字和规则编码的一个数组。
- 视图　是模型的表示层，很少或没有编程逻辑。它读取已经形成的结构并显示它们。在象

精彩好书见www.book118.com

棋例子中，视图是游戏的屏幕表示，可能带有交互的颜色和被吃掉的棋子。

• 控制器 提供用户输入和模型的方向。在象棋例子中，控制器为键盘。

在模型2 Web应用实例中，所有用户请求指向一个URL，servlet有时称之为发送器（控制器）。此servlet在请求的路径信息中查找其所执行任务的提示。可能有一个动作和JSP页面名字的表格来处理每一事件。行为处理组成了应用的模型。它们可以访问数据库或执行其他计算，然后形成结果的JavaBean或其他类，最后，调用JSP页面（视图）表示其输出。

哪一个模型更好呢？模型1更容易得以快速实现，但不可扩展。由于太多的内容被打包在一起，随着应用的增长，它会变得很笨拙。模型2可扩展性好一些，也允许专业人员编写应用的不同部分：

• Java程序员编写模型和控制器
• 用户接口专家编写不做任何事情只是显示输出的JSP页面。

9.9 小结

大量事实说明将一个HTTP请求的处理分片是很有益处的。JSP为此提供两种一般的功能：

• 使用<%@include%>或<jsp:include>包含其他资源。
• 使用<jsp:forward>转发请求

被包含资源可以是静态的（像HTML）或动态的（像JSP或servlet）。转发请求的功能提供了表格驱动应用的基础。

存在两种常用的开发体系结构。通常指模型1和模型2。模型1使用JSP页面接受用户输入，必要时访问数据库，并格式化其输出。模型2遵循MVC范例，允许复杂项目被分割，以适应专攻于其中一层或另一层的各个组。

精彩好书见www.book118.com

第10章 Page伪指令

JSP页面不但包含处理请求和生成响应的代码，还有发向JSP编译器的指令。这些指令称为伪指令。本章概述其中最常用的一个——Page伪指令。此伪指令提供设置影响到页面解释和执行方式的属性方式。语法如下：

```
<%@ page attribute="value" attribute="value" … %>
```

这里，属性如下：

```
language="scripting language"
extends="className"
import="importList"
session="true|false"
buffer="none|sizekb"
autoFlush="true|false"
isThreadSafe="true|false"
info="info_text"
contentType="ctinfo"
errorPage="error_url"
isErrorPage="true|false"
```

属性可按任意次序指定。在一个编译单元内（JSP页面和它使用include伪指令包含的任意文件）可以指定多个Page伪指令。如果使用了多个page伪指令，它们不能多次指定同一属性，但import属性除外。

本章其余部分详细讨论每一属性。

10.1 language

JSP体系结构允许其被扩展成服务器端脚本的通用框架。为此，它在page伪指令中支持language属性。指定值（缺省为Java）应用于所有声明、表达式和当前转换单元内的scriptlet，包括在include伪指令中指定的任意文件。所有JSP 1.1兼容的容器必须支持language属性的java取值。在 JSP 1.1规范中不支持其他语言，但个别JSP引擎是可以的。

虽然规范允许使用其他语言，但却强加了一些限制。语言必须支持Java运行时环境的程度到其允许访问标准隐含对象变量、JavaBean的get和set方法以及Java类的公有方法。

JRun 3.0支持java和javascript作为language属性的取值。当使用java时——显式或隐含指定——在JSP页面找到的scriptlet、表达式和声明像平时一样被复制到生成的servlet中。当指定了javascript时，仍然存在一个生成的Java servlet，但不包含javascript代码，而是此servlet初始化一个读取和解释最初JSP页面的脚本引擎。例如，如果JSP页面如下：

```
<%@ page language="java" %>
```

精彩好书见www.book118.com


```
<%
   int k = 10;
%>
k = <%= k %>
```

则生成的servlet包含语句：

```
out.print("\r\n");
   int k = 10;
out.print("\r\nk = ");
out.print(k);
out.print("\r\n\r\n");
```

这里把k作为一Java变量，对其赋值，并使用out JspWriter变量打印出来。

比较起来，如果同一JSP页面使用Javascript作为language属性的取值：

```
<%@ page language="javascript" %>
<%
   var k = 10;
%>
k = <%= k %>
```

那么生成的servlet初始化一个JRun特有的脚本引擎，调用其evaluate方法如下：

```
if (scriptEngine == null) {
   try {
      scriptEngine =
         ScriptEngineFactory.getScriptEngine("javascript");
      scriptEngine.init(pageContext);
   } catch (Exception e) {
      throw new ServletException
         ("Error initializing scripting engine.", e);
   }
}
if (request.getAttribute(SCRIPT_KEY) != null) {
   scriptEngine.init(
      pageContext,
      (String) request.getAttribute(SCRIPT_KEY),
   (String) request.getAttribute(DECLARATION_KEY));
}
scriptEngine.evaluate(pageContext);
```

显然，以一种JSP规范并未显式支持的语言编写的JSP页面很有可能在不同厂家的JSP容器之间不可移植。

10.2 extends

一般情况，JSP容器向从一个JSP页面生成的任意servlet提供其父类。但是，规范使你可以通过在page伪指令的extends属性中指定的全质名将链接的另一父类归为子类。这样做可以提供JSP页面家族的额外行为而无须在页面中对行为进行显式编码。

精彩好书见www.book118.com

JSP规范提示当使用此功能时要小心，因为它可能会阻止JSP容器提供厂家特有的性能和可靠性的增强。例如，JRun使用的标准JSP父类给出判断依赖性和最后修改时间的方法。类似的，Tomcat实现了保存指定类载入器引用的一个父类。如果使用一个不同的父类，它应该提供超出这些特性的重要功能。

10.2.1 JSP超类所需的接口

用做JSP页面超类的一个类，它必须实现以下接口：

- javax.servlet.jsp.JspPage 一个通用接口，不是HTTP使用所必须的。很少有servlet直接实现此接口。
- javax.servlet.jsp.HttpJspPage 用于HTTP协议下操作的JSP页面。此接口是JspPage的一个扩展。

这些接口定义了必须实现的三种方法，如表10-1所示：

表10-1 在JSP超类中需要声明的方法

方 法	描 述
public void jspInit()	当JSP页面载入时被servlet init（）方法自动调用。虽然必须实现此方法，但实现可以不必做任何事情。此方法被设计为JSP页面子类对它们所需执行的任意初始化工作进行覆盖的占位符
public void jspDestroy()	jspInit（）的反义，此方法当JSP页面卸载时被servlet的destroy（）方法自动调用
public void _jspService (request, response) throws ServletException ,IOException	此方法是JSP请求过程逻辑的核心。它不必在JSP页面内显式定义，因为这是JSP容器从JSP的scriptlet、表达式和伪指令中生成方法时的工作。此方法典型情况在JSP父类中声明为abstract

在_jspService方法中请求和响应参数的精确类型由其支持的协议规定。对于HTTP环境，类型为javax.servlet.http.HttpServletRequest和javax.servlet.http.HttpServletResponse。如果实现的是一不同协议，则需要定义在方法标记中使用的请求和响应类。

HttpJspPage扩展了JspPage，提供指定的HTTP行为。JspPage反过来扩展了javax.servlet .Servlet，定义了表10-2中列出的方法。

表10-2 javax.servlet.Servlet接口中的方法

方 法	描 述
public void init(ServletConfig config) throws ServletException	当servlet首次载入时，servlet容器调用的方法
public ServletConfig getServletConfig ()	返回servlet的配置对象，它管理servlet的初始化参数和上下文
public void service (ServletRequest request , ServletResponse response) throws ServletException ,IOException	servlet引擎所调用以服务于一个请求
public String getServletInfo()	返回servlet的一个描述。缺省返回空字符串
public void destroy()	当servlet被卸载时，由servlet引擎调用

精彩好书见www.book118.com

JSP超类必须符合并实现JSP协议。这需要：

- init（）方法必须调用jspInit（）。
- destroy（）方法必须调用jspDestroy（）。
- service（）方法必须将请求和响应参数置入其指定协议的类并调用_jspService（）。

实现必须是直接的，或者超类本身扩展提供实现的类，如javax.servlet.http.HttpServlet。

10.2.2 一个JSP超类例子

如果继续的话，要加倍小心和谨慎。这一节给出一个完整例子。假定已经有了访问通用数据库的一系列JSP页面。如果JSP页面不必载入JDBC驱动和建立数据库连接，则情况就更简单了。下面servlet既可以执行这些功能，又可用于JSP页面系列的父类。

```
package jspcr.page;

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

/**
* An example of a JSP superclass that can
* be selected with the <CODE>extends</CODE>
* attribute of the page directive. This servlet
* automatically loads the JDBC-ODBC driver class
* when initialized and establishes a connection
* to the USDA nutrient database.
*/
public abstract class NutrientDatabaseServlet
   extends HttpServlet
   implements HttpJspPage
{
   protected Connection con;

   /**
   * Initialize a servlet with the driver
   * class already loaded and the database
   * connection established.
   */
   public void init(ServletConfig config)
      throws ServletException
   {
      super.init(config);
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

精彩好书见www.book118.com


```
            con = DriverManager.getConnection("jdbc:odbc:usda");
        }
        catch (Exception e) {
            throw new UnavailableException(e.getMessage());
        }

        jspInit();
    }

    /**
    * Closes the database connection when
    * the servlet is unloaded.
    */
    public void destroy()
    {
        try {
            if (con != null) {
                con.close();
                con = null;
            }
        }
        catch (Exception ignore) {}

        jspDestroy();
        super.destroy();
    }

    /**
    * Called when the JSP is loaded.
    * By default does nothing.
    */
    public void jspInit()
    {
    }

    /**
    * Called when the JSP is unloaded.
    * By default does nothing.
    */
    public void jspDestroy()
    {
    }

    /**
    * Invokes the JSP's _jspService method.
    */
```

精彩好书见www.book118.com


```
    public final void service(
            HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException
    {
        _jspService(request, response);
    }

    /**
    * Handles a service request
    */
    public abstract void _jspService(
            HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException;
}
```

此例中，驱动器名字和数据库URL是硬编码的。在产品环境下，这些值应为可配置的参数。

为使用NutrientDatabaseServlet作为一个JSP超类，所需的就是使该类在JSP容器类目录下，并且使得JSP在page伪指令的extends属性里指定其全质名。如下所示：

```
<%@ page extends="jspcr.page.NutrientDatabaseServlet" %>

<%--
   This JSP page subclasses the NutrientDatabaseServlet
   parent class, which automatically loads the
   database driver and establishes the connection.
--%>

<%@ page import="java.io.*,java.sql.*" %>
<HTML>
<BODY>
<H3>Food Groups</H3>
<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
<TR><TH>Code</TH><TH>Description</TH></TR>
<%
   // Execute a query

   Statement stmt = con.createStatement();
   String sql = "SELECT * FROM FD_GROUP ORDER BY FDGP_DESC";
   ResultSet rs = stmt.executeQuery(sql);
   while (rs.next()) {
      String code = rs.getString(1);
      String desc = rs.getString(2);
%>
<TR>
```

精彩好书见www.book118.com


```
    <TD><%= code %></TD>
    <TD><%= desc %></TD>
</TR>
<%
    }

    // Close the database objects

    rs.close();
    stmt.close();
%>
</TABLE>
</BODY>
</HTML>
```

注意，JSP并不需要定义Connection对象。此JSP是超类的受保护变量，因而可访问其子类。

10.3 import

import属性用于描述JSP页面中使用类的全质名。这使得通过类名引用该类而无需加入包前缀成为可能。这是一个可选属性。

import属性值是一个包名和/或全质类名的逗号分隔的列表（每一个以字符串“.*”结束）。这些名字被直接转换到生成Java servlet中的import语句。其语法相当灵活。例如，为导入java.io、java.sql和java.util包中所有类，可以使用下列语句中的任意一个。

```
<%@ page import="java.io.*,java.sql.*,java.util.*" %>
```

或者为分行（因为新行记做字符串中的空格），

```
<%@ page import="
    java.io.*,
    java.sql.*,
    java.util.*
    "%>
```

或使用分开的page伪指令：

```
<%@ page import="java.io.*" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
```

除了空格上的差异，上述语句生成同样的Java代码：

```
import java.io.*;
import java.sql.*;
import java.util.*;
```

注意，导入类并不包含载入任何内容，它只是在Java方法中使用类名而无需指定其所属包的一种缩写方式。如果导入java.util.*，编码为：

精彩好书见www.book118.com


```
Vector names = new Vector();
```

代替：

```
java.util.Vector names = new java.util.Vector();
```

它只对Java编译器产生影响，而不是运行时的类图。你可以导入上千个类，但需要的只是实际运行时引用的。

缺省导入列表由4个包组成：

```
•java.lang
•javax.servlet
•javax.servlet.http
•javax.servlet.jsp
```

对这些包中的类不必进行导入声明，也不必写它们的全质名。

注意 *记住，import是page伪指令属性中可以惟一多次指定的属性。*

10.4 session

page伪指令的session属性指出页面是否需要一个HTTP会话。可能值有两个：

- 如果页面需要一个HTTP会话，则session="true"。这是缺省值
- 如果不需要HTTP会话，则session="false"。如果这样指定，session隐含变量为未定义，使用时将引起转换错误。

如果JSP页面不需要会话，从性能角度上说指定session="false"是有意义的，这样不必要的会话就不创建，从而防止消耗内存和CPU周期。

第14章详细讲述HTTP会话和会话管理。

10.5 buffer和autoFlush

buffer和autoFlush属性用于描述JSP采纳的输出缓存模型。buffer属性值可以为“none”，表明所有输出直接写入servlet响应对象的输出流，或者其值为以“kb”为后缀的一个整数值，后者情况下，输出被保存在内存中指定大小的缓存下。当缓存已满时，依据autoflush为“true”或“false”，输出或者被刷新或者产生缓存溢出。缺省缓存大小为8kb。表10-3总结了两个值结合起来产生的各种结果。

表10-3 Buffer和AutoFlush结合产生的影响。

Buffer	AutoFlush	影 响
none	true	字符一旦生成就被写入servlet响应输出流
none	false	非法结合。如果缓存没有起作用，则autoFlush="false"没有意义
8kb	true	使用8,192字节缓存。当缓存已满时，自动被刷新，这是缺省值。
8kb	false	使用8,192字节缓存。当缓存已满时，产生溢出。
sizekb	true	使用size为1,024字节倍数缓存。当缓存已满时，自动被刷新。
sizekb	false	使用size为1,024字节倍数缓存。当缓存已满时，产生溢出。

精彩好书见www.book118.com

10.6 isThreadSafe

缺省的，servlet引擎载入一个servlet的单个实例，并使用线程池服务于每个请求。此值表明两个或多个线程可以同时执行同一个servlet方法。如果servlet有实例变量，并且没有提供同步访问代码，则线程可能会冲突，并通过彼此对变量的访问形成接口。

servlet API提供解决此问题的一种方式——SingleThreadModel接口。此接口没有方法；它只是在servlet的每个实例需要一个专有线程时标记此servlet[1]。page伪指令的isThreadSafe属性提供使得SingleThreadModel与一个JSP页面相关的方式。

如果指定isThreadSafe="true"，则表明已经注意到了任何可能的线程冲突，因此JSP容器可以同时安全地向servlet发送多个请求。

```
<%@ page isThreadSafe="true" %>
```

生成下列类标记：

```
public class jrun__Chap10__examples__isThreadSafe__ex12ejsp25
   extends allaire.jrun.jsp.HttpJSPServlet
   implements allaire.jrun.jsp.JRunJspPage
```

如果取值为“false”，那么JSP容器生成实现SingleThreadModel的一个servlet：

```
<%@ page isThreadSafe="false" %>
```

生成

```
public class jrun__Chap10__examples__isThreadSafe__ex22ejsp25
   extends allaire.jrun.jsp.HttpJSPServlet
   implements allaire.jrun.jsp.JRunJspPage, SingleThreadModel
```

如果未指定，isThreadSafe取值为“true”。

注意 SingleThreadModel是一个受限值，因为它只阻止一个servlet中一个实例内的线程冲突。没有什么限制JSP容器载入一个servlet的多个实例，每一个对应一个专门线程。这种情况下，对外部资源如数据库和文件锁的竞争明显是无序的，仔细安排计划是惟一安全的设计方针。

10.7 info

page伪指令的info属性使用户可以指定JSP页面的描述性信息。例如：

```
<%@ page info="Shopping Cart Checkout Page" %>
```

此属性值被编译到类中，可利用servlet的getServletInfo( )方法获得。它允许servlet引擎在一个管理界面内对其servlet提供有用的描述。

---

1 第14章详细讨论线程发送问题。

精彩好书见www.book118.com

10.8 contentType

JSP页面一般生成HTML输出，但也可以产生其他内容类型。通过指定page伪指令的contentType="value"属性，可以向请求应用返回一个HTTP Content-Type头标。考虑下面给出简单的JSP页面：

```
<%@ page contentType="text/plain" %>
Hello, world!
```

在JRun下，HTTP请求和响应如下：

```
GET /jspcr/Chap10/examples/contentType/ex1.jsp HTTP/1.0

HTTP/1.1 200 OK
Date: Wed, 28 Jun 2000 05:36:33 GMT
Server: Apache/1.3.12 (Win32)
Set-Cookie: jsessionid=7179962170594302;path=/
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Connection: Keep-alive, close
Cache-Control: no-cache="set-cookie,set cookie2"
Content-Length: 17
Content-Type: text/plain

Hello, world!
```

如果未指定contentType属性，则请求和响应如下：

```
GET /jspcr/Chap10/examples/contentType/ex2.jsp HTTP/1.0

HTTP/1.1 200 OK
Date: Wed, 28 Jun 2000 05:40:15 GMT
Server: Apache/1.3.12 (Win32)
Set-Cookie: jsessionid=210659962170816163;path=/
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Connection: Keep-alive, close
Cache-Control: no-cache="set-cookie,set-cookie2"
Content-Length: 15
Content-Type: text/html; charset=ISO-8859-1

Hello, world!
```

除了内容类型，也可以使用下列语法指定字符集：

```
<%@ page contentType="type/subtype; charset=charset" %>
```

10.9 errorPage和isErrorPage

如果执行JSP页面时发生溢出，servlet引擎典型情况下向浏览器扔出一个栈轨迹。在开发阶

精彩好书见www.book118.com

段这对程序员很有帮助，但在商业Web应用中这是不应出现的。JSP提供一种简单且便利的解决方案，它需要结合两个属性使用：errorPage和isErrorPage。

JSP页面可以指出当产生一个不能捕获的溢出时显示一个专门的错误页面，

```
<%@ page errorPage="error_url" %>
```

这里，error_url是同一servlet上下文中另一JSP页面的ORL。此JSP页面必须在其page伪指令中使用下列属性

```
<%@ page isErrorPage="true" %>
```

一个错误页面可以通过exception隐含变量[1]访问溢出。它可以使用exception.getMessage（）抽出错误文本信息，在必要时显示或注册它。它也可以使用exception.printStackTrace（）生成一个栈轨迹。

页面不必很华丽，它只需报告溢出：

```
<%@ page isErrorPage="true" session="false"%>

<H3>Application Error</H3>

The error message is:

<B><%= exception.getMessage() %></B>
```

这作为一个占位符也许很合适，以便开发过程中将来可以修改。例如，加入一个公司徽标及简介，指示如何处理。

因为错误页面本身是一个JSP页面，它可以访问servlet上下文、会话、请求和其他servlet对象。这使得页面可能会捕获到诊断信息，并可能将其转发给技术支持人员。下面给出这样的一个错误页面的例子：

```
<%@ page isErrorPage="true" session="false"%>

<HTML>
<HEAD><TITLE>Tracking Error Page</TITLE></HEAD>
<BODY>
<CENTER>
<FONT SIZE="+3">
<B><I><U>Monolithic<BR>Technologies Corporation</U></I></B>
</FONT>
<P>
You found a bug we didn't know about:
<B><%= exception.getMessage() %></B>
<P>
```

1 这是JSP页面访问此变量的惟一环境。

精彩好书见www.book118.com


```
<%-- Create a form to submit to Tech Support --%>

<FORM ACTION="/send_diags.jsp">
<INPUT TYPE="submit" VALUE="Please click here">
<P>
to send this information
to our Technical Support department:
<P>

<%-- Supply date, time, and servlet name --%>

<%
   String dateTime = new java.util.Date().toString();
   String remoteAddr = request.getRemoteAddr();
   String servletContext = request.getContextPath();

%>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
   <TR>
   <TD><B>Date and Time:</B></TD>
   <TD><%= dateTime %>
   <INPUT TYPE="hidden"
         NAME="bug.dateTime"
         VALUE="<%= dateTime %>">
   </TD>
</TR>

<TR>
   <TD><B>Web Client:</B></TD>
   <TD><%= remoteAddr %>
   <INPUT TYPE="hidden"
          NAME="bug.remoteAddr"
          VALUE="<%= remoteAddr %>">
   </TD>
</TR>
<TR>
   <TD><B>Application:</B></TD>
   <TD><%= servletContext %>
   <INPUT TYPE="hidden"
          NAME="bug.servletContext"
          VALUE="<%= servletContext %>">
   </TD>
</TR>
</TABLE>

<%-- Include the stack trace as a hidden field --%>
```

精彩好书见www.book118.com


```
<INPUT TYPE="hidden" NAME="bug.stackTrace"
      VALUE="<%
java.io.PrintWriter pw = new java.io.PrintWriter(out);
exception.printStackTrace(pw);
      %>"

</FORM>
</CENTER>
</BODY>
</HTML>
```

页面名为TrackingErrorPage.jsp，显示了与溢出相关的错误信息以及日期、时间、客户端的IP地址和Web应用名，并给出一个按钮使用户能将此信息和栈轨迹转发到技术支持部门（使用其他JSP页面，这里并未显示）。此应用中JSP页面应该在其page伪指令中包含该错误页面的引用。

```
<%@ page errorPage="TrackingErrorPage.jsp" %>
```

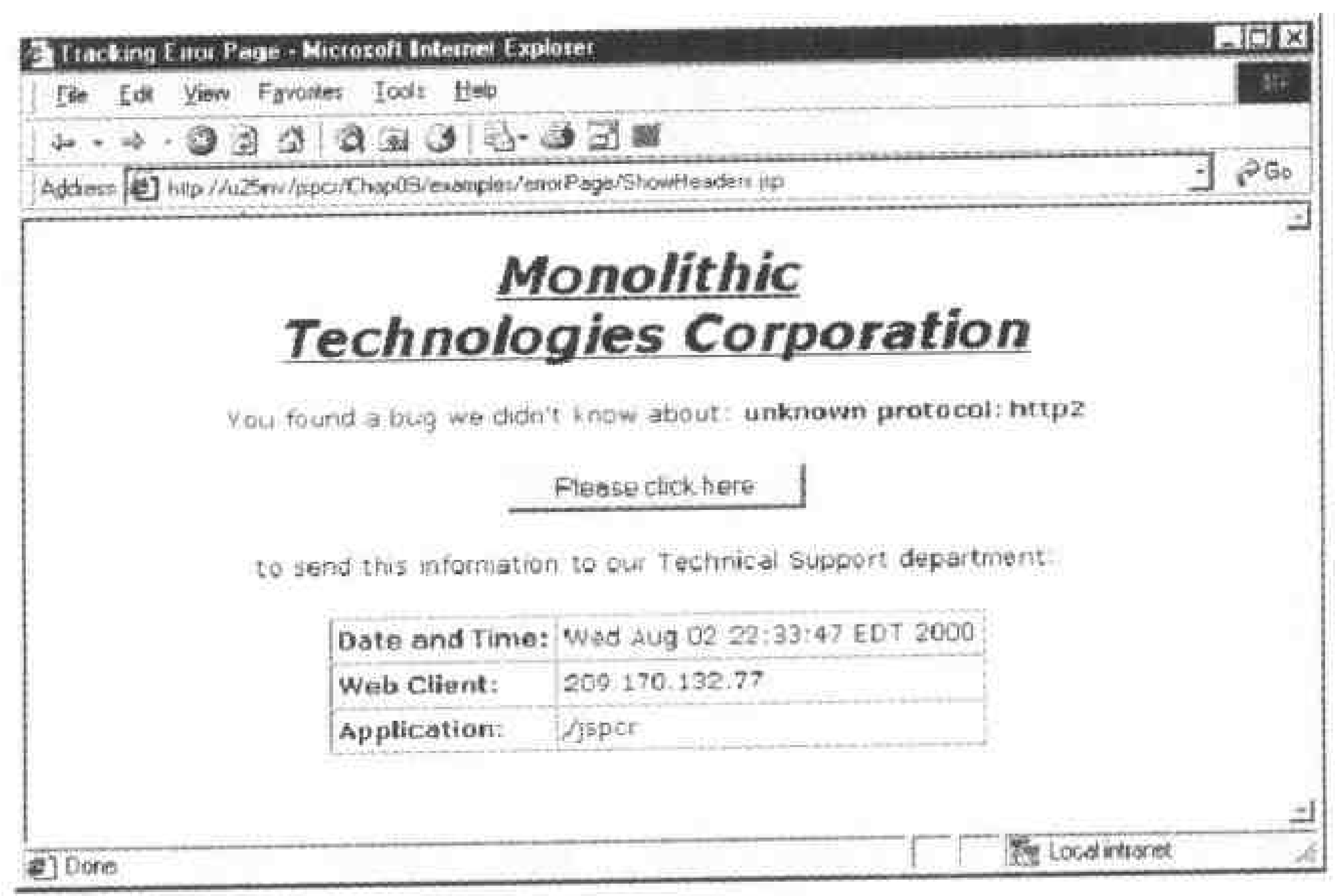
下图给出了一个使用该错误页面即应用JSP页面产生溢出的结果。

毫无疑问一般servlet也可以使用此功能。一个servlet所需做的就是仿效一个JSP生成的servlet所做的一切：

1) 在捕获所有溢出的一个try ...catch块中嵌入其doGet（ ）或doPost方法的主体。

2) 在catch块中，将溢出存储为名javax.servlet.jsp.jspException请求的一个属性。

3) 使用一个RequestDispatcher将请求转发到错误页面URL。



一个诊断错误页面

下面例子给出实现方式：

精彩好书见www.book118.com


```
package jspcr.page;

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

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

public class BuggyServlet extends HttpServlet
{
   public void doGet(
         HttpServletRequest request,
         HttpServletResponse response)
      throws ServletException, IOException
   {
      try {
         // ... body of servlet here
      }
      catch (Exception e) {
         // A servlet can use the JSP error page
         // mechanism by storing the exception
         // as a request attribute and forwarding
         // the request to the error page.

         request.setAttribute
            ("javax.servlet.jsp.jspException", e);

         getServletContext().getRequestDispatcher
            ("/Chap10/examples/errorPage/TrackingErrorPage.jsp")
            .forward(request, response);
      }
   }
}
```

10.10 小结

page伪指令使页面作者向JSP容器提供指令。本章给出每一可以被指定属性的操作描述：

- language　　脚本语言（缺省为java）。
- extends　　页面所特有的超类。
- import　　对生成的servlet可视的包和类。
- session　　是否创建一个HTTP会话对象。
- buffer　　输出缓存模型。
- autoFlush　　当缓存已满或产生溢出时是否刷新缓存。

精彩好书见www.book118.com

- isThreadSafe 是否实现SingleThreadModel。
- info 在一开发工具中显示页面的描述。
- contentType JSP响应使用的字符解码。
- isErrorPage 是否提供对隐含exception变量的访问。
- errorPage 处理不能捕获的溢出页面的URL。

精彩好书见www.book118.com

第11章　JSP标签扩展

JavaServer Page 1.1规范通过使开发人员能用定制标签扩展环境大大增强了JSP体系结构。定制标签是通过用户编写标签处理器返回的JSP页面语法和语义的类XML扩展。标签集合被组织成一个可打包为JAR文件的标签库，使其功能可以很容易地在任何JSP 1.1兼容的servlet引擎上分布和安装。

本章介绍定制标签，给出其任务和优点的概述，并给出一个如何编写和发布定制标签可扩展的，按步实现的例子，然后继续到标签库的细节部分，标签库描述器、标签扩展API和标签处理器。探讨了几个标签环境的例子。最后给出在第一个例子中数据库查询标签的实现。

11.1　为什么要定制标签

大部分程序员可以编写一般的HTML，大部分Web设计者会编写简单的JSP页面，但具有导航、浏览器检测、图形处理和窗体交互的一个好的HTML需要的是一个知识丰富的作者——一个专家[1]。同时，访问数据库、处理交互和使用套接字通信的Java编程已经超出了一个HTML作者的知识范畴。

定制标签为在两种专家之间进行交流提供了一种方式。Java程序员可以把应用功能打成包，而Web设计者可使用它作为构建模块。JavaBean也可用来封装代码，作为属性库它们是非常有用的。循环、嵌套或交互行为的概念很难用bean表达。定制标签向JSP开发提供了一种高层应用特有的方法。

例如，用定制标签编写的一个数据库查询代码如下：

```
<db:connect url="mydatabase">

   <db:runQuery>
      SELECT *
      FROM FD_GROUP
      WHERE FdGp_Desc LIKE '%F%'
      ORDER BY FdGp_Cd
   </db:runQuery>

   <table border="1" cellpadding="3" cellspacing="0">
      <tr><th>Food Group Code</th><th>Description</th></tr>
   <db:forEachRow>
      <tr>
         <td><db:getField name="FdGp_Cd"/></td>
```

1　进入http://www.cnn.com或http://www.msnbc.com查看HTML源码。这样的源码你能编写多少？

精彩好书见www.book118.com


```
            <td><db:getField name="FdGp_Desc"/></td>
        </tr>
    </db:forEachRow>
    </table>

</db:connect>
```

这里，connect，runQuery，forEachRow和getField是面向应用的定制标签。

上面例子中所有逻辑可以使用嵌入到JSP页面的scriptlet编写。例如<db:connect>的等价代码可以包括载入驱动类，打开到数据库的连接（可能是从池取得一个已存在连接），建立Statement和ResultSet对象和处理产生的任意溢出。也有可能在一个JavaBean中具体表现出大部分逻辑，但scriptlet代码仍需对结果集进行循环。它们都不如将逻辑打包到一个其功能可同时显现给Web设计者和servlet开发人员的类HTML标签集方便。

除了内容和表示的分离，定制标签的其他益处包括：

- 简单化 将一个复杂任务表示成具有各自属性和控制流子任务的并集相对比将之编写成晦涩的代码块，表达起来更容易。这样不但编码容易，而且容易理解。例如，在上面的数据库查询中，很容易正确猜出数据库连接的范围是什么，<db:runQuery>块创建了一个隐含的结果集，<db:forEachRow>对此结果集进行循环。
- 代码可重用 在一个Web应用中可能存在成百个数据库查询，如果不求助于使逻辑变得模糊的<%@include%>伪指令，共享scriptlet代码是很难的，而且有无法预料的负面影响。标签库则更容易将标准代码打包并在整个应用中共享它。
- 适合于编程工具 集成开发环境（IDE）只能将scriptlet块看作ASCII文本块。而定制标签，由于具有标签库描述器，可使之具有显示其描述信息，确认其属性等功能的开发工具所管理。

为更好地理解如何开发定制标签的思路，下面看一个简单的例子，按步骤进入定制标签的开发过程。

11.2 开发第一个定制标签

这里并不编写“Hello，World！”标签，而是开发一个很少用到的组件实例——检索Web服务器名字和版本的一个定制标签。此标签的实现和开发的其他所有标签一样，遵循下面4个基本步骤：

1) 定义标签。
2) 编写标签库描述器的接口。
3) 编写标签处理器。
4) 在JSP页面中使用标签。

11.2.1 第1步——定义标签

开始，需要明确定义标签的语法，包括回答下述问题：

精彩好书见www.book118.com

- 标签的名字是什么？后面会看到，总是通过名空间修饰词使用定制标签，因此不必使标签名全局惟一。
- 标签有什么属性？例如HTML<TABLE>标签可选属性为BORDER、CELLPADDING、CELLSPACING和WIDTH。定制标签可以定义任意数目的必需或可选属性，当执行标签时它们被传入标签处理器。
- 标签是否定义了脚本变量？例如，标准行为<jsp:useBean id="xyz" class="jspcr.beans.XYZBean">使得类型为jspcr.beans.XYZBean名为xyz的变量被定义。此变量然后对<jsp:getProperty>和<jsp:setProperty>动作及下面任意scriptlet或表达式中Java代码可用。定制标签可以用同样方式创建脚本变量。
- 标签在其开始和结束标志之间包含的体中是否执行特定动作？HTML<TABLE>标签针对的是在其终止</TABLE>结束标记前的表格行和表格单元。每一元素依赖于操作栈中其上面相关元素提供的信息。定制标签的应用可以具有协同完成某功能的嵌套标签特性。标签体也可包含标签操作的非JSP数据（如SQL语句）。

在第一个示例标签中，功能不多。调用标签的getWebServer，它没有属性，因为在不同的JSP页面不需要分别加以配置。标签并未定义脚本变量，只是返回包含Web服务器名字的字符串以替代getWebServer标签。最后，标签不需要考虑体，因为整个功能均包含在其开始标记中。

11.2.2 第2步——创建TLD入口

标签库描述器（TLD）是定义一系列相关标签名字和属性的XML文档。以下是对getWebServer示例标签使用的TLD：

```
<?xml version="1.0" ?>
<taglib>
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>diag</shortname>
   <tag>
      <name>getWebServer</name>
      <tagclass>jspcr.taglib.diag.GetWebServerTag</tagclass>
      <bodycontent>empty</bodycontent>
   </tag>
</taglib>
```

本章后面会详细介绍TLD，这里关键的一点是TLD映射标签名：

```
<name>getWebServer</name>
```

为一个全质类名：

```
<tagclass>jspcr.taglib.diag.GetWebServerTag</tagclass>
```

JSP容器在编译中遇到此标签时使用此映射创建相应的servlet代码。

将此文件命名为diagnostics.tld。对于此例，惟一需要担心的是将文件复制到正确的位置。TLD可以放在Web应用目录系统中的任意位置，但将之放在WEB-INF目录下更有意义，因为这

精彩好书见www.book118.com

样它就不会被直接公开的访问。依照惯例，TLD通常安装在目录/WEB-INF/tlds下。例如，如果一个Web应用名为test，那么diagnostics.tld应位于/test/WEB-INF/tlds。如果写为相对servlet上下文的一个URI，则为/WEB-INF/tlds/diagnostics.tld。

11.2.3 第3步——编写标签处理器

一个标签的行为在称为标签处理器的Java类中实现。JSP容器创建和保存标签处理器实例，这些类中的预定义方法从JSP页面生成的servlet直接调用。

在示例标签中，需要取得Web服务器的名字（例如Apache、Microsoft IIS、Netscape Enterprise等）。servlet API并不提供取得此信息的明显方式。请求对象对Web客户端内容比较了解，servlet上下文则知道servlet引擎的内容，但这些对象都不知道正在侦听端口80的软件产品。然而，此信息可由Web服务器本身当其向Web客户端发回HTTP响应时提供。这里采取的方案是在标签处理器中进行一个伪HTTP请求，然后从返回的HTTP头标中抽取服务器信息。

以下是标签处理器的完整源码：

```
package jspcr.taglib.diag;

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

/**
* Handler for the "getWebServer" tag
*/
public class GetWebServerTag extends TagSupport
{
   public int doStartTag() throws JspException
   {
      try {

         // Get the request object from the page context

         HttpServletRequest request =
            (HttpServletRequest) pageContext.getRequest();

         // Request information from web server

         URL url = new URL("http",
         request.getServerName(),
         request.getServerPort(),
         "/");
      URLConnection con = url.openConnection();
```

精彩好书见www.book118.com


```
            ((HttpURLConnection) con).setRequestMethod("OPTIONS");
            String webserver   con.getHeaderField("server");

            // Write it to the output stream

            JspWriter out = pageContext.getOut();
            out.print(webserver);
        }
        catch (IOException e) {
        throw new JspException(e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

下面详细分析该源码，查看其功能。

```
package jspcr.taglib.diag;
```

第一行标识了包名。不是必须在一个包中放人代码，但这将有助于组织起相关的类，形成更有意义的Javadoc文档。另外，一些JSP引擎不能对定制标签正确生成import语句，因此没有包名的类可能在生成servlet中引起编译错误。

```
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.net.*;
```

简单的标签处理器通常只需导入javax.servlet.jsp和javax.servlet.jsp.tagext包以及java.io.IOException类。这里需要的是javax.servlet.http中的HttpServletRequest类和java.net中的几个类。

```
public class GetWebServerTag extends TagSupport
```

一个标签处理器需要实现Tag接口或BodyTag接口，它们都在javax.servlet.jsp.tagext包中。BodyTag是Tag的子接口。标签作者可以随意直接利用这些接口，扩展缺省实现类TagSupport或BodyTagSupport的其中一个，覆盖任务所需手工加入的那些方法通常更加方便。示例标签并不支持标签体，因此这里只扩展TagSupport类。

```
public int doStartTag() throws JspException
```

调用此方法是在遇到开始标记时，其指定的任意属性已在标签处理器中设置完成之后，但在处理标签体之前。本例中，不存在体和属性，因此所有代码都包含在doStartTag（）方法中。注意，如果代码运行出错，该方法会产生JspException溢出。这里访问的是可能会产生java.io.IOException溢出的网络类，因此将整个方法放在一个try...catch块中，它将java.io.IOException溢出转换成一个JspException以便被JSP容器处理。注意，方法同时返回一个

精彩好书见www.book118.com

整数返回码。

```
HttpServletRequest request =
            (HttpServletRequest) pageContext.getRequest();
```

为向Web服务器发送一个HTTP请求，这里需要知道主机名和接收请求的端口号。此信息可在从pageContext对象中得到的请求对象中找到。机警的读者会注意到PageContext在此类中并未定义，因为它被定义为TagSupport超类的一个protected域，因此可以被如这里的子类所访问。此变量在调用TagSupport.setPageContext（ ）方法时doStartTag（ ）在调用前被设置[1]。

```
    URL url = new URL("http",
              request.getServerName(),
              request.getServerPort(),
              "/");
  URLConnection con = url.openConnection();
  ((HttpURLConnection) con).setRequestMethod("OPTIONS");
  String webserver = con.getHeaderField("server");
```

这里使用java.net.URL中的4个参数构造器以取得协议名、服务器名、端口号和路径，并从中得到一个URLConnection对象。因为这里实际上并不关心任何特殊文件的内容，因此指定的是OPTIONS方法而不是GET或POST。也可以使用HEAD，它基本与GET一样，但只返回头标。偶尔Web服务器也会告之不支持HEAD方法。OPTIONS应运行于任何与HTTP/1.1兼容的Web服务器（其最终意图就是要返回Web服务器支持的请求方法列表）。调用连接对象的getHeaderField（ ）方法使得请求被发送，响应中适当的HTTP头标被读取。

```
JspWriter out = pageContext.getOut();
out.print(webserver);
```

在webserver变量中捕获所需信息后，简单地将其写入当前servlet输出流。该输出流从页面上下文中得到。结果是JSP页面中使用的getServer标签被从HTTP请求中得到的服务器信息所替换。

```
return SKIP_BODY;
```

最后，退出该方法，返回在Tag接口中定义的一个整数常量SKIP-BODY。因为这里定义的标签没有体，因此不必对其评估，如果指定任何其他返回码，JSP页面会产生运行时溢出。

编译示例代码

现在完成了整个标签处理器，源码文件必须命名为GetwebServerTag.java，其编译类必须具有全质名jspcr.taglib.diag.GetWebServerTag。实现此功能的一种简易方式是在Web应用的/WEB-INF/classes目录下创建一个适当的目录集，

```
/WEB-INF/classes/jspcr/taglib/diag
```

将.java源文件放在diag目录中。在/WEB-INF/classes目录下使用下列命令编译程序：

---

1 读TagSupport和BodyTagSupport源码很有助益。它们是一些很小的类，对查看页面上下文和体内容变量的出处以及findAncestorWithClass的工作方式具有指导意义。该源码通常可在获得servlet.jar类的相同位置得到。

精彩好书见www.book118.com


```
javac jspcr/taglib/diag/GetWebServerTag.java
```

规定将servlet.jar文件放在类目录中某处。这里应将GetWebServerTag.class文件与GetWebServerTag.java放在同一目录下。如果不这样，应确保package语句输入正确。

11.2.4 第4步——将标签并入JSP页面

到此为止，可以使用标签了。下面JSP页面（ShowServer.jsp)给出了其使用方式：

```
<%@ taglib prefix="diag" uri="/WEB-INF/tlds/diagnostics.tld" %>

<HTML>

<HEAD>
<TITLE>Basic Example of a Custom Tag</TITLE>
</HEAD>

<BODY>
<H3>Basic Example of a Custom Tag</H3>
The web server is <diag:getWebServer/>
</BODY>

</HTML>
```

1. taglib伪指令

第一行包含了taglib伪指令：

```
<%@ taglib prefix="diag" uri="/WEB-INF/tlds/diagnostics.tld" %>
```

此伪指令在JSP页面中必须在其指向的任何定制标签使用之前出现，通常放置在页面的顶部。

2. 在JSP页面中如何使用标签

此Web页面其余部分是传统的HTML，除了指定定制标签的一行：

```
The Web server is <diag:getWebServer/>
```

当第一次调用ShowServer.jsp，JSP容器使用来自taglib伪指令的信息定位标签库描述器，并标识出在此页面中使用标签的位置。当生成的servlet收到一个请求，它产生下列HTML：

```
<HTML>

<HEAD>

<TITLE>Basic Example of a Custom Tag</TITLE>
</HEAD>

<BODY>
<H3>Basic Example of a Custom Tag</H3>

The web server is Apache/1.3.12 (Win32)
```

精彩好书见www.book118.com


```
</BODY>

</HTML>
```

当然，结果依赖于实际的Web服务器。如图11-1所示。

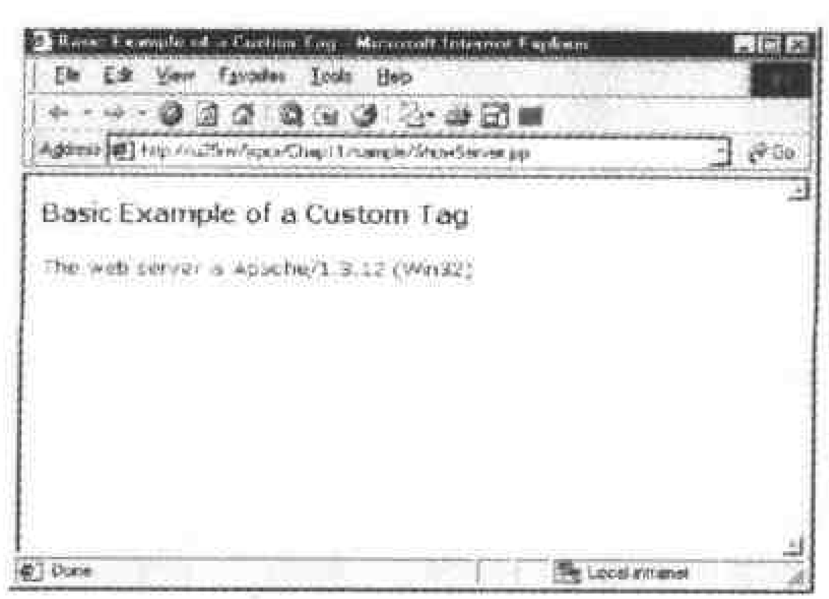


图11-1　使用定制标签标识Web服务器软件的一个JSP的输出

需要注意的是JSP页面中使用的定制标签必须符合严格的XML规则：

1) 所有标签必须完整。可通过一个匹配的结束标记：

```
<diag:name>
...
</diag:name>
```

2) 或在没有体的情况下通过缩写形式：

```
<diag:name/>.
```

3) 所有属性必须加引号，并为偶数个：

```
<diag:for id="I" start="1" end="10">
...
</diag:for>
```

4) 嵌套的标签不能重叠；下面为非法：

```
<diag:A>
<diag:B>
</diag:A>
</diag:B>
   is illegal.
```

5) 标签和属性名大小写敏感。

11.3　标签处理器工作方式

一个标签处理器是通过实现JSP容器调用的一系列预定义方法执行定制标签行为的一个Java类。本节中将给出标签处理器结构、其实现的接口、生命期和其如何使用属性和脚本变量，最后讲到协作及嵌套标签和它们如何交互。首先介绍JSP容器如何转换和调用JSP页面。

11.3.1　JSP容器的功能

前面介绍过一个JSP页面存在3种形式：

- .jsp文件　页面作者编写的最初的源文件，它可能包含HTML、scriptlet、表达式、声明、行为标记和伪指令。
- .java文件　与.jsp文件等价servlet的Java源码。此servlet由JSP容器生成。
- .class文件　生成的Java servlet的已编译形式。

精彩好书见www.book118.com

当一个HTTP客户端向JSP页面发出请求时，JSP容器检查.jsp和.java文件的修改日期。如果.java文件并不存在或比.jsp 文件旧（如果JSP页面已被修改时发生），JSP容器重新创建Java servlet并编译它。在此期间，发生下述转换：

- <%@page%>、<%@include%>和<%@taglib%>伪指令向JSP容器提供转换时信息。
- JSP表达式和HTML行在_jspService（）方法中依其出现的顺序被转换到out.print（）语句中。
- Scriptlet被原封不动地复制到_jspService（）。
- 声明被原封不动复制到_jspService（）外的源码中。
- 标准JSP行为，如<jsp:include>、<jsp:useBean>和<jsp:setProperty>被转换成执行其功能的运行时逻辑。
- 定制标签被扩展到调用其相应标签处理器中方法的Java语句中。

容器生成的标签相关代码

容器使用taglib伪指令定位标签库描述器（TLD）并将其和基于标签前缀页面中用到的定制标签相匹配。例如，如果伪指令为：

```
<%@ taglib prefix="db"
      uri="/WEB-INF/tlds/database.tld" %>
```

那么容器读取database.tld文件取得其描述的标签列表和与每一标签相关的标签处理器类的名字。当以后在具有名空间前缀的页面中遇到一个标签，

```
<db:connect url="mydatabase">
```

它查找与具有指定名字的标签前缀相关的一个标签库。容器使用在TLD中找到的标签结构信息生成一系列完成标签功能的Java语句。在前面给出的db:connect标签中，这些代码包含：

1) 创建connect标签处理器实例或从池中取得一个代码。

2) 将connect标签处理器一个引用传递到pageContext对象的代码。这是很有用的特性，因为它给出标签处理器对JSP页面的Request、Response、HttpSession、ServletContext和输出流对象的访问。这也意味着标签处理器可以在页面上下文惯例的任意层次上取得或设置属性。

3) 如果db:connect嵌套在另一定制标签中，将引用传递到父标签的代码。

4) connect标签处理器setUrl（）方法的调用，传递值为“mydatabase”。

5) 名为doStartTag（）方法的调用。connect标签处理器利用此方法执行当遇到开始标记时发生的任何行为（稍后再详述）。

11.3.2　标签处理器功能

在一个JSP页面的体中，一个定制标签如下：

```
<app:mail from="Accounting Manager" to="Staff" >
   <app:subject>Expense Reports</app:subject>
   Please be sure to submit all expense reports before
   the fifteenth day of the month to allow sufficient
   processing time. Thanks.
```

精彩好书见www.book118.com


```
</app:mail>
```

此标签组件包含：

- 具有0或多个属性的开始标记<app:mail...>。
- 结束标记</app:mail>。
- 开始和结束标记之间的行，称为标签体，包含一般文本或其他JSP语句[1]。

在将标签转换成servlet代码时，容器对每一组件调用标签处理器，使用pageContext对象共享处理器属性。这些方法的过程有时称为处理器的生命期。

在此工作中，处理器必须实现下面两个接口中的一个：

- javax.servlet.jsp.tagext.Tag 对于不在其体上进行操作的标签。
- javax.servlet.jsp.tagext.BodyTag 规定BodyTag是Tag子接口的标签。

这些接口指定了标签处理器必须提供的生命期方法。

该API还提供了两个支持类——TagSupport和BodyTagSupport——它们是上述两个接口的缺省实现。大部分标签扩展为这两个支持类而不是直接实现接口，虽然上述接口并不复杂。使用支持类的一个好处是可以只覆盖需要改变的方法，并允许支持类处理其他方法。另外，支持类可以保存页面上下文和体内容对象到保护变量中，这样子类就可以很简单地访问它们。

11.4 标签库

定制标签被实现和分布在一个称为标签库的结构中，有时称为taglib。一个标签库是类和以下元信息的集合：

- 标签处理器 实现定制标签功能的Java类。
- 标签附加信息 向JSP容器提供逻辑以确认标签属性和创建脚本变量的类。
- 标签库描述器（TLD） 描述单个标签和整个标签库属性的XML文档。

一个标签库组件可安装于JSP容器能访问的任意位置。标签处理器和标签附加元信息需要定位在JSP容器类载入器可找到的地方。标签库描述器可在URL指定的任意位置。然而在开发阶段，JSP 1.1规范要求JSP容器必须接受一个被打包成具有固定结构的JAR文件的标签库。在这样一个JAR文件中，类应位于一个目录树中，而此目录树根应与其包结构的根相匹配。TLD必须是/META-INF目录中名为taglib.tld的一个文件。这意味着要发布一个标签库，只需将其JAR文件复制到/WEB-INF/lib目录即可。或者将类解压到/WEB-INF/classes目录中，将TLD放在另一Web可访问的位置。典型情况是该目录名为/WEB-INF/tlds，这样做只是方便，不是必须的。

11.4.1 标签库描述器

JSP容器所需的标签库配置信息保存在标签库描述器中。一个TLD是一个描述库中每个标签及其标签处理器和属性以及版本和关于整个库标记信息的XML文档。

TLD元素

---

1 一个标签并不需要有体。标签可以只基于开始标签中指定的属性执行其功能。这种情况下，通常使用缩写的<tag.../>记号。

精彩好书见www.book118.com

标签库描述器的文档类型定义（DTD）可在http://java.sun.com/j2ee/dtds/Web-jsptaglibrary_1_1.dtd中找到。一个有效的TLD由一个简单的<taglib>元素组成，此元素可按固定次序包含如下一些子元素：

- tlibversion　是包含标签库版本号的必须元素。这是一个点式十进制数，最多为4组由小数点分隔的数字组成，如“1.0”或“1.3.045”。
- jspversion　是标记支持标签库所需的JSP规范的最低级别的可选元素。例如对JSP版本1.1，将为“1.1”。
- shortname　是标记标签库的缩写名。JSP编程工具可以使用该名字作为库中标签的缺省前缀。DTD规定此名字必须不包含空格并以希腊字母开始。然而空格的限制实际上普遍被忽视。shortname是必须元素。
- uri　是定义标识此库惟一URI的可选元素。典型URL位置来自可下载的taglib的最新版本的位置
- info　是给出标签库描述性信息的可选元素。它方便人们在JSP编程工具中进行观察。
- tag　在TLD中一或多个标签入口　它们描述了组成库的每个标签。

一个标签元素本身由至多6类子元素组成：

- name 在JSP页面中使用标签的名字。与标识标签库的名空间前缀一起，该名字是JSP容器惟一标识一个标签。
- tagclass 组成实现标签的标签处理器全质名的必须元素。
- teiclass　由标签使用的标签附加信息（TEI）类全质名组成的可选元素。TEI类给出关于标签处理器创建的脚本变量以及对标签属性执行的任意有效性验证的信息。
- bodycontent　可选地描述标签处理器如何使用其体内容。可能值为：

empty　标签体必须为空。

JSP　标签体由其他JSP元素组成。

tagdependent　标签体由标签本身解释，不带JSP转换。

- info　标签的人工可读可选的描述性信息。
- attribute　当在一个JSP页面中使用标签时可被编码的属性的可选信息。此入口在本章后面“定义标签属性”一节详细阐述。

11.4.2 taglib伪指令

taglib伪指令的目的是指定TLD的位置，设置在页面上与标签区分开来的一个短别名（前缀）。语法如下：

```
<%@ taglib prefix=" tag prefix" uri=" taglibURI" %>
```

这里两个属性为：

tag prefix　页面上惟一名字。用于标识库中的标签。例如，如果前缀为diag，那么页面上使用的此标签库中任何标签应写为<diag:xxx>，这里xxx是标签名。

taglibURI　标签库本身的URI。可以是以“/”开始的绝对路径名，“/”解释为像前面例子中一样的Web应用的根。或者是一个充作TLD符号名的URL。这种情况下，名字必须通过web.xml

精彩好书见www.book118.com

文件中的<taglib>入口被映射到实际的TLD。此方法在下一节讨论。

在web.xml文件中映射标签库

假定JAR文件包含类和版本号为taglib的3.8.2的TLD，该文件名为util_v3_8_2.jar，并发布在一个Web应用的/WEB-INF/lib目录下。Taglib伪指令如下可直接指向此目录：

```
<%@ taglib
      prefix="util"
      uri="/WEB-INF/lib/util_v3_8_2.jar"
%>
```

当然，当安装了版本3.8.3时，表明所有使用此标签库的JSP必须使用新的版本号升级。

一种替代方式是映射TLD的物理位置到可在taglib伪指令中使用的符号名。为此需要向Web应用的/WEB-INF/web.xml发布描述器加入一个<taglib>元素[1]。对于上一个例子，此元素结构如下：

```
<taglib>
   <taglib-uri> uri</taglib-uri>
   <taglib-location>
      /WEB-INF/lib/util_v3_8_2.jar
   </taglib-location>
</taglib>
```

这里uri可为任意有效URI，也许是一个类似文件的助记符，如/util-taglib，或可找到最新taglib版本的位置的URL。因此如下可为taglib伪指令编码：

```
<%@ taglib
      prefix="util"
      uri="http://www.vendor.com/taglibs/util"
%>
```

注意，URI不必指向一个实际文件。它是JSP容器可以在web.xml中查找实际文件位置的惟一标识符。还要注意，此映射技术只适用于以规定格式编码的JAR文件（TLD在/META-INF/taglib.tld)，一些JSP容器实现只对此格式表示认同。犹豫不定时，总是可以将JAR文件放在/WEB-INF/lib，将TLD放在/WEB-INF/tlds中，并在JSP页面中直接指向/WEB-INF/tlds/filename.tld。

11.5 标签处理器API

下面一节给出与Tag接口和TagSupport类相关的方法。

11.5.1 Tag接口

表11-1列出了必须被实现Tag接口类所支持的生命期方法。

1 它们为什么不能对此元素使用一个不同的名字呢？因为TLD文件已经具有带有完全不同含义的一个<taglib>元素。这样就不会给JSP作者产生混淆的机会。

精彩好书见www.book118.com

该接口还包含4个常量，表示doStartTag()和doEndTag（）方法的可能返回码：

- EVAL_BODY_INCLUDE 当doStartTag（）返回时，指明页面实现servlet应对标签体进行评估。
- SKIP_BODY 当doStartTag（）返回时，指明servlet应忽视标签体。
- EVAL_PAGE 当doEndTag（）返回时，指明页面其余部分应像通常一样被评估。
- SKIP_ PAGE 当doEndTag（）返回时，指明页面其余部分应被跳过。

表11-1 Tag接口中的方法

方 法	描 述
public void setPageContext (PageContext ctx)	生成的servlet在请求处理器执行其他任务前首先调用此方法，实现类应保存上下文变量以便它可以在标签生命期任意一点均可利用。从页面上下文中标签处理器可以访问所有的JSP隐含对象并在任意范围内设置和取得属性
public void setParent (Tag parent)	使一个标签可以找到操作栈中它上面的标签。在setPageContext后立即调用
public Tag getParent()	返回父标签
public int doStartTag() throws JspException	在设置了页面上下文、父和在开始标记中编码的任意属性后调用。返回码表明JSP实现servlet是否应评估标签体（EVAL_BODY_INCLUDE)或（SKIP_BODY)。此方法产生溢出JspException则表明一个致命错误
public int doEndTag() throws JspException	当遇到结束标记时调用。返回码表明JSP实现servlet是否应继续页面的其余部分（EVAL_PAGE）或（SKIP_ PAGE）。此方法产生溢出JspException则表明一个致命错误
public void release()	确保在页面退出前被调用。允许标签处理器释放其保留的任意资源，并重置其状态以便必要时再次使用

11.5.2 TagSupport类

javax.servlet.jsp.tagext. TagSupport 是实现Tag接口的具体类。除了此接口，TagSupport类还提供表11-2列出的另外一些方法。

扩展此类而不是直接实现接口通常是有益的。除了对所有必需方法提供缺省实现，保存pageContext变量、TagSupport还给出几个简便方法。findAncestorWithClass（）对支持嵌套标签十分有用。例如，一个outer标签可以将一系列对象作为实例变量管理，提供一个公有祖先使之可被inner标签访问。本章后面的数据库标签例子解释了这种技术。

11.6 标签处理器生命期

图11-2描述了一个标签处理器生命期的事件。流线图中显示的过程对应JSP容器在页面被转换成servlet时JSP容器为标签生成的Java代码。知道何时调用每个标签处理器方法以及页面和容器所处的状态是十分重要的。理解此协议有助于按预想进行编码。标签本身在运行时生成的servlet中并不存在这一点也很重要——标签被设置属性和调用标签处理器中方法的等价代码所替代。

下面考虑流线图的每一步。

精彩好书见www.book118.com

表11-2 TagSupport类中的附加方法

方 法	描 述
public static Tag findAncestorWithClass(Tag thisTag ,Class cls)	为所需的父标签处理器查找运行时标签栈。一个标签处理器可以提供其范围内子标签调用的方法
public void setId(String id) public String getId()	保存和检索在id属性中指定的名字
public void setValue(String name ,Object o) public Object getValue(String name)	在本地哈希表中给定名字下保存和检索取值
public void removeValue(String name)	从本地哈希表中删除给定名称的值
public Enumeration getValues()	返回哈希表中关键字的一个java.util.Enumeration

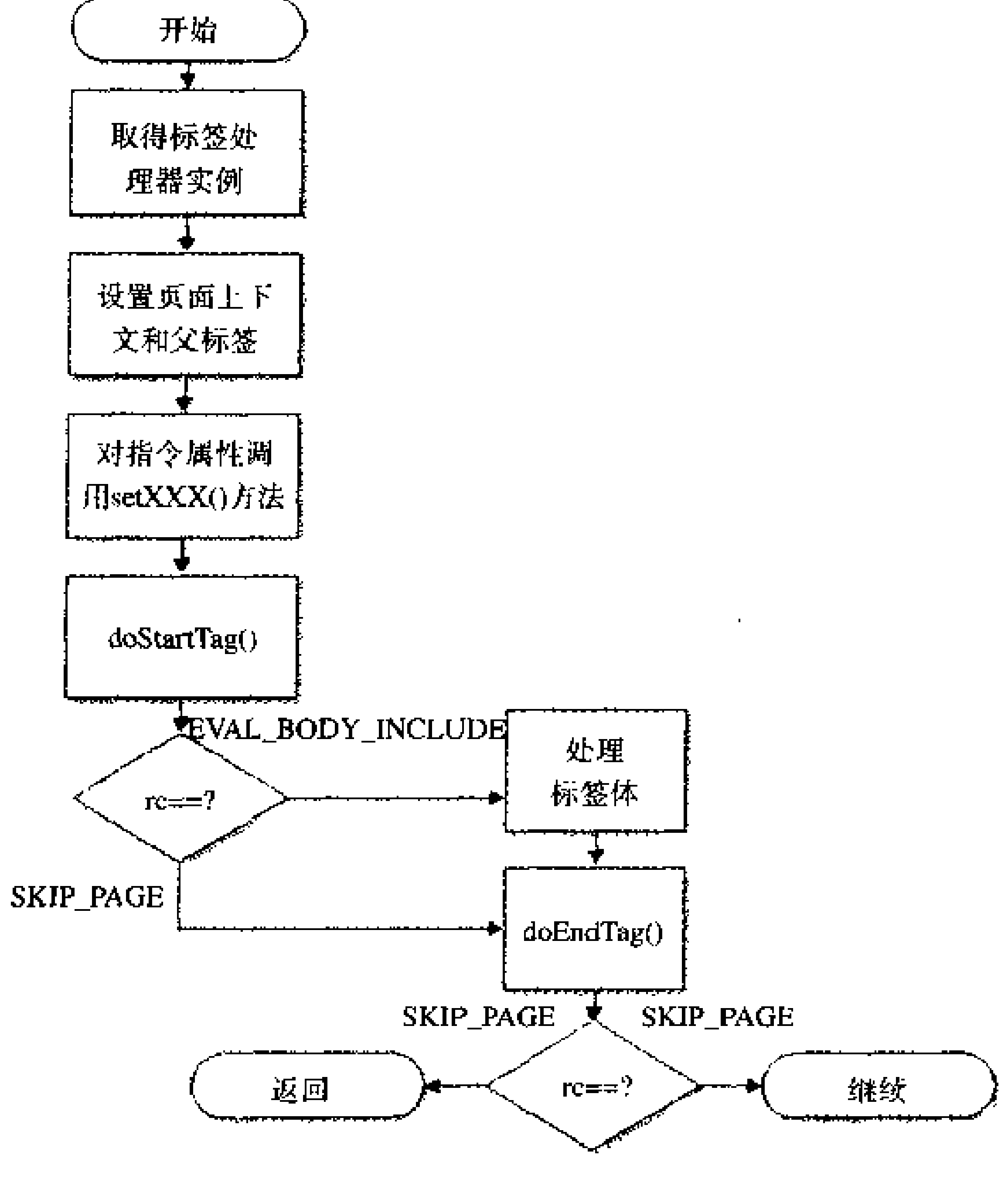


图11-2 标签处理器生命期流线图

精彩好书见www.book118.com

11.6.1 流线图

开始，生成的servlet需要创建标签处理器类的一个实例。实现方式通常是调用JSP容器一部分的工厂类的一个方法。工厂类包含一个标签处理器实例池以使其可重用不再处于激活状态的标签处理器。

接着，初始化标签处理器，使servlet获知其存在性。servlet通过调用标签处理器中的两个方法实现此过程：

set PageContext(Page PageContext Context ctx)对象包含所有JSP隐含对象的引用，并提供对页面属性、请求、会话和应用级别的访问。servlet调用此方法时，标签处理器应将上下文保存到一实例变量中以便它对所有处理器的方法均可利用。注意，TagSupport基类自动完成这一切。

setParent(Tag parent)　JSP页面内的标签可以嵌套，也就是被包含在另一标签的体内。调用set PageContext ( ) 后，servlet立即调用setParent ( )，如果存在，传递包含此标签的父标签引用。如果标签并未嵌套，则参数为空。对参数中标签的访问使得一个标签可以访问其父中的方法，这实际上是一种互操作行为。TagSupport类会自动保存此变量。

如果一个标签支持属性，属性的运行时取值通过处理器必需提供的setter方法被传入标签处理器。例如本章开始时数据库连接标签。

```
<db:connect url="mydatabase">
```

有一个属性，名为url，其标签处理器必需具有保存url属性值的如下方法：

```
public void setUrl(String value)
```

最有可能将该属性值保存在一个私有实例变量中。对开始标记中的编码为xxx的每一属性，生成的servlet有一个setXxx（value）方法调用。这些调用在setParent（）调用后立即执行。

在这一点上，开始调用标签处理器的doStartTag（）方法。页面上下文和父标签已经被设置，并已具备所有标签的属性。该方法可以读取这些变量，并执行实现标签功能所需的计算和操作。它可以通过设置页面上下文中的属性改变JSP页面内脚本变量的取值。这一点在11.10节“定义脚本变量”中会详细介绍。如果遇到任何致命错误，该方法应产生JspException。

doStartTag（）方法必需返回一个整型返回码，或为SKIP_BODY，或为EVAL_BODY_INCLUDE。如果doStartTag（）返回码为EVAL_BODY_INCLUDE，则正常处理标签体。如果返回码是SKIP_BODY，则初始JSP页面中直到此标签结束标记处的内容均被忽略。

注意　SKIP_BODY是TagSupport基类doStartTag（）的缺省返回码。TagSupport基类给出一个很少用到的实例，TagSupport可以在此处执行一些有用的功能而不是成为子类——可以使用它作为“注释”代码的定制标签处理器。如果在一个TLD中编写了下列入口

```
<tag>
   <name>skip</name>
   <tagclass>javax.servlet.jsp.tagext.TagSupport</tagclass>
   <bodycontent>JSP</bodycontent>
</tag>
```

精彩好书见www.book118.com

则可以使用下列代码围住一个JSP页面的任意部分[1]。

```
< prefix.skip>
...
</ prefix.skip>
```

那么它在运行时就不会被执行。

标签体被评估或忽视后，调用处理器的doEndTag（）方法。像doStartTag（）一样，此方法必需返回一整型码表明处理结果。如果值为EVAL_PAGE，页面的其余部分被评估；如果值为SKIP_PAGE，则servlet代码立即从_jspService（）中返回。

11.6.2 生成代码的一个例子

看一个例子会使生成的servlet和一个标签处理器之间的交互更加清晰。下面开发本章前面给出的getWebServer标签的增强版，使得可以指定任意头标名，而不是选择硬编码Server头标。为此，标签接受一个名为name的属性。下面一节给出关于长度的标签属性，但对于这个例子，所需的是在TLD中描述属性并使用标签处理器的setName（）方法使之可以同该属性通信。此标签称为getWebServerHeader。TLD需要加入一点内容：

```
<tag>
   <name>getWebServerHeader</name>
   <tagclass>jspcr.taglib.diag.GetWebServerHeaderTag</tagclass>
   <bodycontent>empty</bodycontent>
   <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>
```

name属性定义为必需域。如果需要，其值可由一个请求时间表达式提供，而不是编码为一个字面值。

不要惊讶，标签处理器几乎和getWebServer的一模一样。下面是getWebServerHeader标签处理器的源码：

```
package jspcr.taglib.diag;

import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.net.*;
```

1 实际不是任意部分，如果Scriptlet、表达式、标准行为和HTML模板在skip标签体中，它们将被抑制，但不能在一个scriptlet、表达式或声明中使用标签。

精彩好书见www.book118.com


```
/**
* Handler for the "getWebServerHeader" tag
*/
public class GetWebServerHeaderTag extends TagSupport
{
   private String name;

   /**
   * Sets the name property. A call to this method
   * is automatically generated by the JSP container
   * when a tag with the name attribute is used in
   * a JSP page.
   */
   public void setName(String name)
   {
      this.name = name;
   }

   public int doStartTag() throws JspException
   {
      try {

         // Get the request object from the page context

         HttpServletRequest request =
            (HttpServletRequest) pageContext.getRequest();

         // Request information from web server

         URL url = new URL("http",
                    request.getServerName(),
                    request.getServerPort(),
                    "/");
         URLConnection con = url.openConnection();
         ((HttpURLConnection) con).setRequestMethod("OPTIONS");

         // Extract the requested header

         String header = con.getHeaderField(name);

         // Write it to the output stream

         JspWriter out = pageContext.getOut();
         out.print(header);
      }
      catch (IOException e) {
```

精彩好书见www.book118.com


```
            throw new JspException(e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

主要差别是加入了name属性。这需要创建一个name变量和setName（）方法。然后替代：

```
String webserver = con.getHeaderField("server");
```

以

```
String header = con.getHeaderField(name);
```

这里name是JSP标签中的编码值。

在JSP页面中，使用旧的标签得到Web服务器产品名和新的标签得到Allow头标。因为标签处理器使用OPTIONS方法进行一个HTTP请求，服务器应返回一个列出其接受的请求方法的Allow头标。下面是改动后页面，名为ShowServerHeader.jsp。

```
<%@ taglib prefix="diag" uri="/WEB-INF/tlds/diagnostics.tld" %>

<HTML>

<HEAD>
<TITLE>Custom Tag with Attributes</TITLE>
</HEAD>

<BODY>
<H3>Custom Tag with Attributes</H3>

Request methods supported by this instance of
<diag:getWebServer/>
are
<H4><diag:getWebServerHeader name="allow"/></H4>
</BODY>

</HTML>
```

ShowServerHeader.jsp运行时产生输出如图11-3所示。

下面讨论JSP容器（此例为JRun 3.0)为ShowServerHeader.jsp生成的servlet_jspService（）方法的一部分。为清楚起见，源码被再次格式化并作一些小的改动。注意，可以不必这样写；以下为JSP容器基于JSP页面和TLD定义生成的代码：

```
PageContext pageContext = __jspFactory.getPageContext
   (this, request, response, null, true, 8192, true);
JspWriter out = pageContext.getOut();

try {
```

精彩好书见www.book118.com

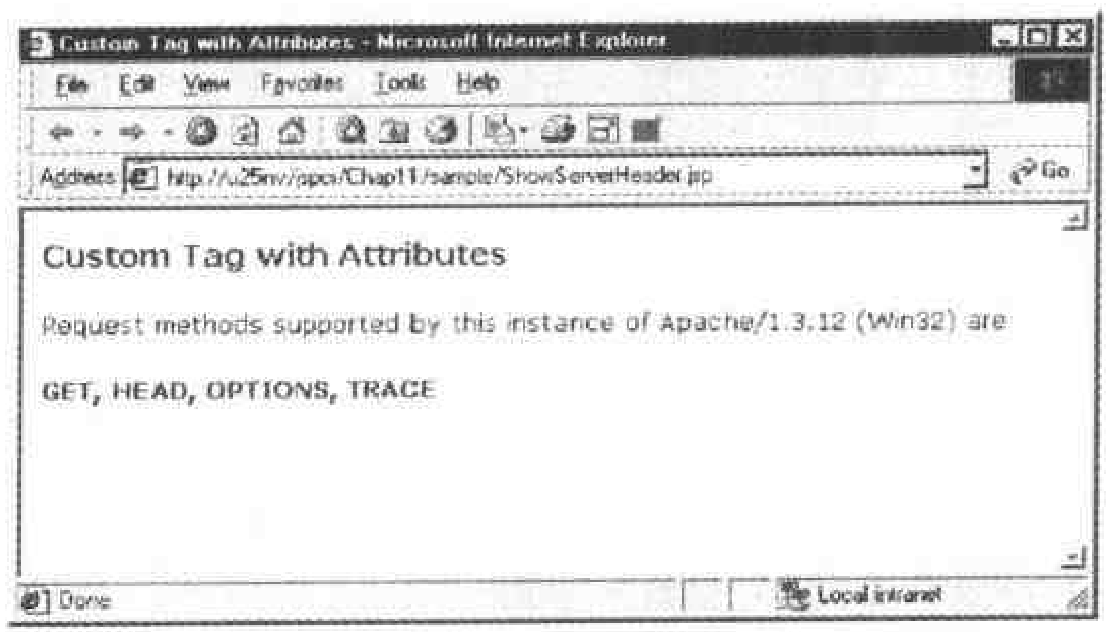


图11-3 输出改进的标签例子

```
out.print("\r\n\r\n"
    + "<HTML>\r\n\r\n"
    + "<HEAD>\r\n"
    + "<TITLE>Custom Tag with Attributes</TITLE>\r\n"
    + "</HEAD>\r\n\r\n"
    + "<BODY>\r\n"
    + "<H3>Custom Tag with Attributes</H3>\r\n\r\n"
    + "Request methods supported by this instance of"
    + "\r\n");
GetWebServerTag tag1 = (GetWebServerTag)
   JRunJSPStaticHelpers.createTagHandler
      (pageContext, "GetWebServerTag");

tag1.setPageContext(pageContext);
tag1.setParent(null);
tag1.doStartTag();

if (tag1.doEndTag() == Tag.SKIP_PAGE) {
   return;
}

out.print("\r\n"
    + "are\r\n"
    + "<H4>");

GetWebServerHeaderTag tag2 = (GetWebServerHeaderTag)
   JRunJSPStaticHelpers.createTagHandler
      (pageContext, "GetWebServerHeaderTag");

tag2.setPageContext(pageContext);
tag2.setParent(null);
```

精彩好书见www.book118.com


```
    tag2.setName("allow");
    tag2.doStartTag();

    if (tag2.doEndTag() == Tag.SKIP_PAGE) {
        return;
    }

    out.print("</H4>\r\n\r\n</BODY>\r\n\r\n</HTML>\r\n");

}
```

在_jspService（）开始附近，servlet创建和初始化其页面上下文：

```
PageContext pageContext = _jspFactory.getPageContext
    (this, request, response, null, false, 8192, true);
```

JspFactory有一个getPageContext（）方法，接收当前servlet、请求和响应对象错误页面URL（如果存在）及表明页面是否需要HTTP会话的标记、输出缓存大小和指出缓存是否被自动刷新的标记的引用。该方法返回封装所有这些对象的一个被初始化页面上下文。

```
JspWriter out = pageContext.getOut();
```

初始化页面上下文对象后，servlet使用它得到相应输出写入者的引用。如果需要，标签处理器可以使用同样的方法调用向页面进行输出。对于与其体内容进行交互的标签处理器问题有点复杂，在下一节再讨论。

```
GetWebServerTag tag1 = (GetWebServerTag)
    JRunJSPStaticHelpers.createTagHandler
        (pageContext, "GetWebServerTag");
```

打印完页面头标，servlet使用帮助类中一个静态方法创建标签处理器的实例。此帮助类可以使用标签处理器实例池或执行其他优化——JSP规范并未规定实现的方式。这给servlet引擎厂家展示其产品性能和功能的机会。

```
tag1.setPageContext(pageContext);
tag1.setParent(null);
```

如图11-2所示，生成的servlet然后调用标签处理器的setPageContext（）和setParent（）方法。这里不存在父标签，因此参数值为null。

```
tag1.doStartTag();
```

对于全面描述标签处理器的页面环境，调用处理器的doStartTag（）方法。注意，即使doStartTag（）返回了一个返回码，这里并不捕获返回码，原因是TLD指出getWebServer标签没有体（<bodycontent>empty</bodycontent>），因此没有处理此条件的条件代码生成。JSP容器能够优化代码而不是检测一个无意义的返回值。

```
if (tag1.doEndTag() == Tag.SKIP_PAGE) {
    return;
```

精彩好书见www.book118.com


```
}
```

doEndTag ( ) 方法返回EVAL_PAGE或SKIP_ PAGE。当看出SKIP_ PAGE只是使得_jspService()方法返回，两个返回值的作用就很清楚了。

打印完插入的HTML，servlet开始处理第2个标签：

```
GetWebServerHeaderTag tag2 = (GetWebServerHeaderTag)
    JRunJSPStaticHelpers.createTagHandler
        (pageContext, "GetWebServerHeaderTag");
tag2.setPageContext(pageContext);
tag2.setParent(null);
tag2.setName("allow");
tag2.doStartTag();
```

产生此标签处理和前一个的差别的原因是getWebServerHeader标签有一个name属性。这转换成在调用doStartTag ( ) 前调用标签处理器的setName ( ) 方法。结束标签处理方式相同。其返回码判断出是否从_jspService ( ) 方法退出还是继续。

11.7 定义标签属性

定制标签可以有任意多的属性，当在JSP页面中使用时表现为开始标签中的名字/取值对编码。例如，下列标签：

```
<opera:role name="Papageno" range="baritone"
            description="a bird-catcher"/>
```

有3个属性：name、range和description。属性可以是必需或可选的，其值可编码为字符串或在请求时使用JSP表达式提供（如果标签允许这样做）。

对标签提供的每一属性，其标签处理器必须给出两件事情：

- 保存属性的一个实例变量。
- 一个或多个setAttrname ( ) 方法，这里Attrname是第一个字母为大写的属性名。

对前面例子中的标签，标签处理器如下：

```
/**
* RoleTag
*/
public class RoleTag extends TagSupport
{
    // Three attributes:

    private String name;
    private String range;
    private String description;

    // ... and their setter methods:

    public void setName(String nameFromJSPTag)
```

精彩好书见www.book118.com


```
        {
            name = nameFromJSPTag;
        }

        public void setRange(String rangeFromJSPTag)
        {
            range = rangeFromJSPTag;
        }

        public void setDescription(String descriptionFromJSPTag)
        {
            description = descriptionFromJSPTag;
        }

        public int doStartTag() throws JspException
        {
            try {
                JspWriter out = pageContext.getOut();
                out.println("<TR>");
                out.println("<TD>" + name + "</TD>");
                out.println("<TD>" + range + "</TD>");
                out.println("<TD>" + description + "</TD>");
                out.println("</TR>");
            }
            catch (IOException e) {
                throw new JspException(e.getMessage());
            }
            return SKIP_BODY;
        }
    }
```

JSP容器在JSP servlet中生成获得定制标签中已编码的属性值的代码，并发送给标签处理器。实现此功能通过对每一属性调用setAttrname（）方法。其调用位置在页面上下文和父标签被设置之后，但在调用doStartTag（）之前。例如，如果一个JSP页面使用<opera:role>：

```
<%@ page session="false" %>
<%@ taglib prefix="opera" uri="/WEB-INF/tlds/opera.tld" %>

<HTML>
<HEAD><TITLE>The Magic Flute</TITLE></HEAD>

<BODY>
<H2>The Magic Flute</H2>
<H3>Dramatis Personae</H3>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR><TH>Role</TH><TH>Range</TH><TH>Description</TH>
```

精彩好书见www.book118.com


```
<opera:role name="Tamino" range="Tenor"
            description="an Egyptian prince"/>

<opera:role name="Pamina" range="Soprano"
            description="daughter of the Queen of the Night"/>

<opera:role name="Papageno" range="Baritone"
            description="a bird-catcher"/>

<opera:role name="Queen of the Night" range="Soprano"
            description="die Sternflammende Konigin"/>

<opera:role name="Sarastro" range="Bass"
            description="High Priest of Isis and Osiris"/>

</TABLE>
</BODY>
</HTML>
```

那么生成的servlet（再次使用JRun作为容器）将处理每一<opera:role>标签，代码如下：

```
RoleTag roleTag = (RoleTag)
   JRunJSPStaticHelpers.createTagHandler
   (pageContext, "RoleTag");
   roleTag.setPageContext(pageContext);
   roleTag.setParent(null);

   roleTag.setRange("Baritone");
   roleTag.setName("Papageno");
   roleTag.setDescription("a bird-catcher");

   roleTag.doStartTag();
```

属性setter方法是一个标签支持属性所需的惟一方法，但在TLD中可以指定更多的信息。在<tag>元素中，可以有任意数目下列形式的<attribute>元素：

```
<attribute>
   <name>attributeName</name>
   <required>true|false</required>
   <rtexprvalue>true|false</rtexprvalue>
</attribute>
```

这里只需要属性名；其他两个元素可选，缺省为false。

如果指定了<required>true</required>，那么属性在标签被用到的所有位置都要被编码，要不就产生一个致命的转换错误。另外，属性是可选的。标签处理器应该小心处理属性未指定的情况。在这种情况下，实例变量为null。

精彩好书见www.book118.com

如果<rtexprvalue>true</rtexprvalue>被指定，那么属性值可以使用一个请求时表达式指定。这种方式下编码的属性形式为：

```
attribute="<%= scriptlet_expression %>"
```

这里引号只包含JSP表达式。除了在运行时提供属性值，这样做还会保存表达式的类型。换句话说，

```
date="<%= new java.util.Date() %>"
```

结果是导致生成的servlet代码：

```
tag.setDate(new java.util.Date());
```

使得标签处理器的public void setDate(Date date)方法被调用，而不是public void setDate(String date)。

下面是有两个可选属性的一个定制标签的例子，每一属性使用请求时表达式指定。

```
<x:formattedDate date=" date" format=" format"/>
```

date属性在请求时由表达式java.util.Date对象指定。但格式可以是java.text.SimpleDateFormat或SimpleDateFormat使用的格式化字符串。其TLD如下：

```
<?xml version="1.0" ?>
<taglib>

   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>util</shortname>

   <tag>
      <name>formattedDate</name>
      <tagclass>jspcr.taglib.util.FormattedDateTag</tagclass>
      <bodycontent>empty</bodycontent>
      <info>
         Returns a date formatted using the specified format.
         If no date is specified, uses current date.
         Default date format is MM/dd/yyyy
      </info>

      <attribute>
         <name>date</name>
         <required>false</required>
         <rtexprvalue>true</rtexprvalue>
     </attribute>

      <attribute>
         <name>format</name>
         <required>false</required>
```

精彩好书见www.book118.com


```
        <rtexprvalue>true</rtexprvalue>
      </attribute>
    </tag>

</taglib>
```

下面是标签处理器：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.text.*;
import java.util.*;

/**
* FormattedDateTag
*/
public class FormattedDateTag extends TagSupport
{
    // The date attribute

   private Date date;
   public void setDate(Date date)
   {
      this.date = date;
   }
   // The format attribute

   private SimpleDateFormat format;
   public void setFormat(String fmtstr)
   {
      format = new SimpleDateFormat(fmtstr);
   }
   public void setFormat(SimpleDateFormat fmt)
   {
      format = fmt;
   }

   /**
   * Prints the date when the start tag is encountered
   */
   public int doStartTag() throws JspException
   {
      // Get date attribute, defaulting to current date
      Date date = this.date;
      if (date == null)
```

精彩好书见www.book118.com


```
            date = new Date();

         // Get date format attribute, defaulting
         // to month/day/year

         SimpleDateFormat format = this.format;
         if (format == null)
            format = new SimpleDateFormat("MM/dd/yyyy");

         // Format and print

         try {
            pageContext.getOut().print(format.format(date));
         }
         catch (IOException e) {
            throw new JspException(e.getMessage());
         }

         return SKIP_BODY;
      }
   }
```

注意，存在两个setFormat（）方法：一个针对java.text.SimpleDateFormat，另一个接收一个字符串并从中创建一个SimpleDateFormat。JSP容器生成servlet的方法依靠标签是否使用一个请求时表达式编码。下面是使用标签方式的例子：

```
<%@ page session="false" %>
<%@ page import="java.util.*,java.text.*" %>
<%@ taglib prefix="x" uri="/WEB-INF/tlds/util.tld" %>

<%
   Calendar gc = new GregorianCalendar(1931, 6, 25);
   Date then = gc.getTime();
   SimpleDateFormat fmt =
      new SimpleDateFormat("MMMMM d, yyyy");
%>
The date was

<x:formattedDate date="<%= then %>" format="<%= fmt %>" />.
```

当用于前面JSP页面时，输出如下：

```
The date was July 25, 1931.
```

11.8 体标签处理器API

简单的标签是在其开始标签中执行其功能的非常有用组件，但定制标签的真正强大之处是

精彩好书见www.book118.com

与其标签体交互的能力。这样可以使标签实现：

- 对其体文本进行后处理，也许将之分类、从中形成HTML表格或过滤出字符如“<”和“>”，用HTML安全的等价字符“<”和“>”替代。
- 定义新的隐含对象并对其创建脚本变量。
- 与嵌套标签合作执行复杂操作。

操作于其体上的标签是本章前面讨论的标签的扩展。它们实现了javax.servlet.jsp.tagext.Tag的一个子接口，称为javax.servlet.jsp.tagext.BodyTag。对TagSupport类也是如此，还存在BodyTag的一个基类实现，称为BodyTagSupport。

11.8.1 BodyContent

当JSP容器为具有体的标签生成代码时，它保存并恢复表示当前servlet输出写入者的对象。在处理标签体之前，创建一个新的输出写入者——它是BodyContent类的实例之一。当体被执行时，out脚本变量以及pageContext.getOut（）返回值均指向新的写入者对象。如果存在多层嵌套，写入者被保存在栈中，这样每一层都有自己的写入者。

BodyContent是javax.servlet.jsp.JspWriter的子类，但区别于其超类，其内容并不自动写入servlet输出流，而是积累在一字符串缓存的一定数量中。标签体完成后，初始JspWriter被恢复，但BodyContent对象仍在bodyContent中doEndTag（）变量可以利用。其内容可由其getString（）或getReader（）检索，并在必要时修改及写入恢复的JspWriter输出流中进而被嵌入页面输出。表11-3 列出BodyContent提供的附加的方法。

表11-3 BodyContent类的附加方法

方法	描述
public void flush() throws IOException	覆盖JspWriter.flush（）方法以便它总是产生溢出。刷新一个BodyContent写入者并不有效，因为它没有连接到将被写入的实际的输出流
public void clearBody()	重置BodyContent缓存为空，如果在doAfterBody（）中体被写入包围的写入者，这十分有用
public Reader getReader()	体内容执行后返回其读取者。此读取者被传递到可以处理java.io.Reader的其他类，如StreamTokenizer，FilterReader或一个XML解析器
public String getString()	体内容被执行后，返回包含它的一个字符串
public void writeOut(Writer w)	将体内容写入指定输出写入者
public JspWriter getEnclosingWriter()	返回栈中下一个更高的写入者对象（可能是另一BodyContent）

JSP容器为什么要为定制标签输出创建这样详细的结构呢？我们直到JSP容器允许输出被后处理和过滤。但还因为并不是所有的体内容均产生输出。例如，在上面的数据库查询中

```
<db:runQuery>
   SELECT *
   FROM FD_GROUP
```

精彩好书见www.book118.com


```
    WHERE FdGp_Desc LIKE '%F%'
    ORDER BY FdGp_Cd
</db:runQuery>
```

体根本不是HTML，而是表示一个SQL语句的字符串。它大概要使用BodyContent.getString( )方法读取并传递到其输出被写入Web页面的JDBC语句对象。这可能是自动的，因为BodyContent对象在缓存中保存其输出而不是写入它。

11.8.2 BodyTag接口

与其体内容进行交互标签的生命期有点复杂，因此在其标签处理器中就需要更多的方法。为此，存在一个Tag接口的扩展称为BodyTag，它继承Tag需要的所有方法，还加入了与体处理相关的三种新方法。表11-4描述了此接口。

除了三种新方法，BodyTag接口还定义了一个新的整型常量：

EVAL_BODY_TAG　当doStartTag（ ）返回时，使得新的BodyContent对象被创建并与此标签处理器相关联。当doAfterBody（ ）返回时，使得JSP servlet在修改完此标签控制的任意脚本变量后再次评估体。这使一个标签处理器有可能循环一系列元素，对每一元素进行体的评估。

表11-4　BodyTag接口中的方法

方　法	描　述
public void setBodyContent (BodyContentout)	在当前JspWriter已被写入，一个新的BodyContent写入者在被创建后由JSP servlet调用。它刚好发生在doStratTag（ ）之后
public void doInitBody() throws JspException	setBodyContent（ ）之后，体被评估前调用的生命期方法。如果多次评估体，此方法只调用一次
public int doAfterBody() throws JspException	体被评估后，BodyContent写入者仍处于激活状态时调用的生命期方法。此方法必需返回EVAL_BODY_TAG或SKIP_BODY。如果返回码是EVAL_BODY_TAG，体再次被评估，doAfterBody（ ）再次被调用

注意　除了SKIP_BODY，doStartTag（ ）还可以返回EVAL_BODY_INCLUDE或EVAL_BODY_TAG，它们都表明体应被处理。然而，实现BodyTag的标签处理器不能返回EVAL_BODY_INCLUDE，没有实现BodyTag的标签处理器不能返回EVAL_BODY_TAG。这些行为都会引起运行时溢出。

11.8.3 BodyTagSupport类

像Tag接口的情况一样，BodyTag有一个缺省实现类，称为javax.servlet.jsp.tagext.BodyTagSupport。此类扩展了TagSupport，但有一些轻微的差别。表11-5给出BodyTagSupport实现的公有方法。

体标签处理器可随意直接实现BodyTag接口，但BodyTagSupport通常是一种更方便的基类。

精彩好书见www.book118.com

表11-5 BodyTagSupport中的方法

方 法	描 述
public int doStartTag() throws JspException	覆盖TagSupport中的doStartTag（）方法，缺省返回EVAL_BODY_TAG而不是SKIP_BODY
public int doEndTag() throws JspException	调用TagSupport中的doEndTag()，返回结果
public void setBodyContent (BodyContent out)	在一保护成员变量bodyContent中保存新的体内容对象。子类可以直接访问此变量
public void doInitBody() throws JspException	缺省什么都不做。有可能在体被评估之前被需要执行初始化的子类所覆盖
public int doAfterBody() throws JspException	每次体被评估后由JSP servlet调用，体内容对象仍处于激活状态。此方法必需返回SKIP_BODY或EVAL_BODY_TAG，使得体再次被评估，doAfterBody（）再次被调用
public void release()	设置bodyContent变量为null，然后调用super.release（）。覆盖它的方法也必需调用super.release（）。否则，bodyContent将对垃圾集合不可利用
public BodyContent getBodyContent()	返回bodyContent变量。子类已经可以访问保护变量，但此方法允许无关的标签处理器类向此体内容发送输出
public JspWriter getPreviousOut()	在bodyContent变量上调用getEnclosingWriter（）并返回结果的简便方法

11.9 体标签处理器生命期

图11-4给出与其体交互的标签处理器的稍微复杂一点的生命期。

下面描述生命期流线图中的每一事件。

流线图

直到doStratTag（）前面几步与图11-2中并无差别。第一次不同是从此方法返回码的处理过程。体标签处理器中的doStratTag（）可能返回SKIP_BODY，它使得doEndTag（）产生分支，或EVAL_BODY_TAG，则开始一系列处理标签体的事件。

当从doStratTag（）返回EVAL_BODY_TAG，JSP servlet[1] 调用页面上下文的pushBody（）方法，执行下列三件事情：

1) 在栈中保存当前JspWriter。

2) 创建新的BodyContent对象，将其保存在页面上下文的页面范围内名为name的属性中。

3) 设置新的BodyContent对象为JSP页面隐含变量out。

之后，JSP servlet对新的写入者调用标签处理器的setBodyContent（）方法。

接着，调用标签处理器的doInitBody（）方法处理体被评估前所需的任意初始化。此初始化也可以在doStartTag（）中完成，但新的BodyContent那时将不可利用。如果标签中没有体，则

1 这一节中术语JSP servlet指由JSP容器基于JSP页面源码生成的servlet。

精彩好书见www.book118.com

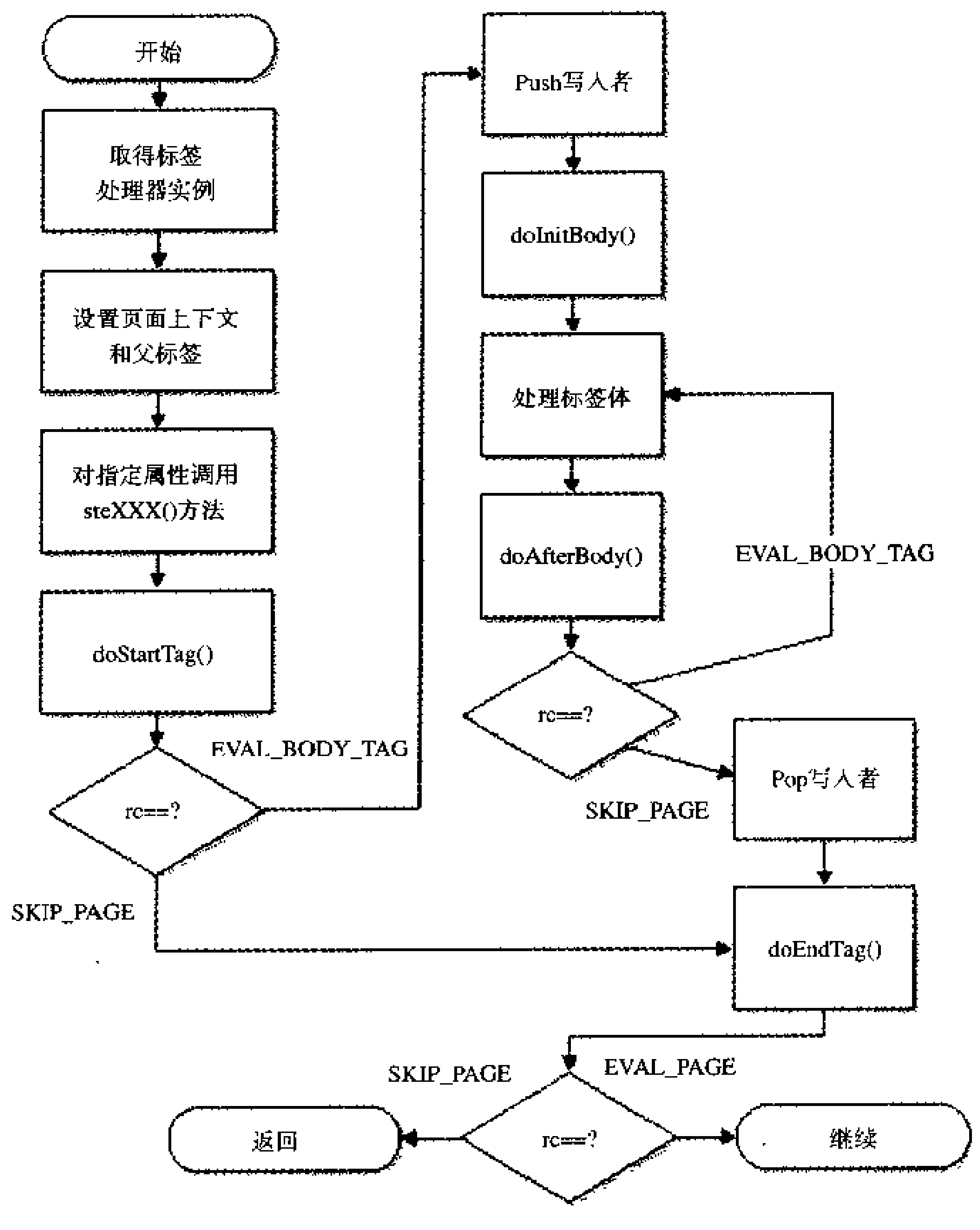


图11-4 体标签处理器生命期流线图

doInitBody()不被调用。如果它检测到任意致命错误，则产生JspException溢出。

到这里，servlet正常处理标签体，将输出写入BodyContent对象。此过程依赖于TLD中标签的< bodycontent >元素的取值。此元素有3个可能值：

- empty 标签体必须为空
- JSP scriptlet、表达式和模板HTML像通常一样评估。如果体范围内存在其它定制标签，也一样被评估。如果被使用在页面其它位置也一样。如果每一个都有实现BodyTag的标签处理器，则过程递归进行——当前BodyContent被推前，一个新的BodyContent被设置到内部标签等。

精彩好书见www.book118.com

- tagdependent 体内容被原封不动地写入BodyContent。scriptlet、表达式以初始JSP源码形式出现，而不是被JSP容器所解释。

处理完体之后，servlet调用标签处理器的doAfterBody( )方法。如果标签处理器要在此时将体内容写入包围的JspWriter，可如下实现：

```
JspWriter out = bodyContent.getEnclosingWriter();
out.println(bodyContent.getString());
bodyContent.clear();
```

或

```
JspWriter out = bodyContent.getEnclosingWriter();
bodyContent.writeOut(out);
bodyContent.clear();
```

如果体内容不太大，等到doEndTag ( ) 完成，再在一次操作中写入体内容可能更容易。

doAfterBody()方法返回两个可能返回码之一：

- SKIP_BODY 继续页面的其余部分
- EVAL_BODY_TAG 使得紧跟doAfterBody方法后体再次被评估，这是一个数组或枚举正在被处理，每一循环中数组或枚举中的下一元素正被设置到一脚本变量的典型情况。

当doAfterTag()最终返回SKIP_BODY，循环退出。体内容现在已完成，因此创建它的过程被反向：

1) 调用pageContent.popBody ( ) 及时检索前面的JspWriter。

2) 写入者被置回到out脚本变量。

最后，调用doEndTag ( ) 方法，允许标签处理器像输出流发回其内容。到这里，pageContext.getOut ( ) 指向最初的写入者，与标签处理前相同。然而，体内容在保护bodyContent变量中仍可利用。如下它被写入servlet输出流：

```
JspWriter out = pageContext.getOut();
out.println(bodyContent.getString());
```

或简单为

```
bodyContent.writeOut(pageContext.getOut());
```

doEndTag ( ) 应返回SKIP_PAGE，使得JSP页面的其余部分被忽视，或返回EVAL_PAGE，使得页面像通常一样被评估。

在给出一个体标签生成代码的详细例子前，首先需要理解标签处理器如何与脚本变量交互。

11.10 定义脚本变量

JSP页面作者对脚本变量很熟悉——它们通常是定义在一个scriptlet或<jsp:useBean>行为中的Java变量。例如，下面代码开始处的scriptlet：

```
<%
   String[] flavors = {"Chocolate", "Strawberry", "Vanilla"};
```

精彩好书见www.book118.com


```
    for (int i = 0; i < flavors.length; i++) {
%>
<LI>Flavor <%= i %> is <%= flavors[i] %>
<%
    }
%>
```

整型变量i和字符串数组变量flavors被定义，以后可用于页面上其他scriptlet和表达式。在此JSP页面：

```
<jsp:useBean id="m1" class="Meteor"/>
<jsp:setProperty name="m1"
      property="bane"
      value="The atmosphere"/>
Ahhhh! <%= m1.getBane() %>! Ahhhh!
```

<jsp:useBean>行为定义了类Meteor名为ml的变量。它被紧接着的<jsp:setpProperty>行为使用，并对最后一行的表达式可用。

定制标签也可以在其标签处理器中定义脚本变量，像前面例子一样，变量然后就可用于scriptlet、表达式和同一页面上的其他标签。定义这样变量的机制是TagExtraInfo类。

11.10.1 TagExtraInfo类

需要定义变量或对属性进行有效性检验的标签需要定义扩展TagExtraInfo类的一个类。此子类与TLD中定制标签相关：

```
   <tag>
      <name>mytag</name>
      <tagclass>mypackage.MyTagHandler</tagclass>
      <teiclass>mypackage.MyTagTEI</teiclass>
      ...
</tag>
```

TEI在JSP转换时起重要作用。当JSP解析器读取一个taglib伪指令，它载入相关的标签和TLD中每一标签的TEI类名，然后在解析标签时，调用其TEI中的方法取得脚本变量和有效性信息。通过在一个TEI子类中覆盖这些方法，一个标签作者可以创建变量并检验标签属性是否有效。表11-6列出TagExtraInfo类中可利用的方法。

主要关注的方法是getVariableInfo（ ）。此方法在页面转换时被JSP解析器调用，返回VariableInfo对象的一个数组。VariableInfo是一个只有4个域的数据结构：

- varName 被创建的变量名。
- className 变量的类的全质名。
- declare 一个布尔变量，如果JSP解析器应为变量创建一个实际的定义（与假想此类的变量已经在servlet前面部分被定义过相反），则为true。
- scope 整型，表明变量定义的位置（或被激活）。

精彩好书见www.book118.com

表11-6 TagExtraInfo类

方 法	描 述
public VariableInfo[] getVariableInfo(TagData data)	基于data参数中属性名和取值列表，构建描述名字、类型、存在性和创建脚本变量的范围的VariableInfo对象数组
public boolean isValid(TagData data)	在页面转换时由JSP解析器调用。给出属性名和取值列表，该方法可以分别和或相结合起来确认其有效性。如果属性有效，返回true，否则为false。缺省实现返回true
public void setTagInfo(TagInfo info)	设置被此类使用的TagInfo对象
public TagInfo getTagInfo()	返回此类使用的TagInfo对象

scope有三个可能值，每一个都由VariableInfo中定义的常量表示：

- AT_BEGIN 当遇到开始标签时定义变量并保持对页面其余部分可视。例如，下面为<jsp:useBean>定义的id变量的可视性。
- AT_END 结束标签后定义变量并保持对页面其余部分可视。
- NESTED 变量只在标签体范围内被定义。

1) 例子：enumerate标签

为阐述TEI类的用法，开发一个名为enumerate的标签，它循环java.util.Enumeration，使每一个元素像标签体中的脚本变量一样可以利用。以下是标签的TLD定义：

```
<tag>
   <name>enumerate</name>
   <tagclass>jspcr.taglib.util.EnumerateTag</tagclass>
   <teiclass>jspcr.taglib.util.EnumerateTEI</teiclass>
   <bodycontent>JSP</bodycontent>
   <info>
      Iterates tag body through an enumeration
   </info>

   <attribute>
      <name>enumeration</name>
      <required>true</required>
   <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

标签有一个必需属性，名为enumeration。此属性类型为java.util.Enumeration，因此其值必须由请求时表达式提供。标签处理器使用Enumeration.hasMoreElements（ ）和nextElement（ ）方法控制实际的循环：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
```

精彩好书见www.book118.com


```
import java.util.*;

/**
* EnumerateTag
*/
public class EnumerateTag extends BodyTagSupport
{
   // Enumeration attribute

   private Enumeration list;
   public void setEnumeration(Enumeration list)
   {
      this.list = list;
   }

   public int doStartTag() throws JspException
   {
      // Do not evaluate the body if the list is empty

      if (list.hasMoreElements()) {

         // Create a scripting variable named "element"
         // that contains the value of the current
         // element of the enumeration

         pageContext.setAttribute
            ("element", list.nextElement());

         return EVAL_BODY_TAG;
      }

      return SKIP_BODY;
   }

public int doAfterBody() throws JspException
{

      // Get next element. This will be assigned
      // to the scripting variable named "element"

      if (list.hasMoreElements()) {

         pageContext.setAttribute
            ("element", list.nextElement());
         return EVAL_BODY_TAG;
      }
```

精彩好书见www.book118.com


```
        // If no more elements, exit from the loop

        return SKIP_BODY;
    }

    public int doEndTag() throws JspException
    {
        // getOut() now refers to the original JspWriter

        try {
            bodyContent.writeOut(pageContext.getOut());
        }
        catch (IOException e) {
            throw new JspException(e.getMessage());
        }

        return EVAL_PAGE;
    }
}
```

注意，当处理enumeration的每一元素时，它作为名为element的属性被存储在页面上下文中。为确保当前元素像脚本变量一样可利用，这里采纳一个TEI类：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
* EnumerateTEI
*/
public class EnumerateTEI extends TagExtraInfo
{
    public VariableInfo[] getVariableInfo(TagData tagData)
    {
        return new VariableInfo[] {
            new VariableInfo(
                "element", // Variable name
                "java.lang.Object", // Class
                true,               // Create a declaration?
                VariableInfo.NESTED // Scope
            )
        };
    }
}
```

精彩好书见www.book118.com

这里getVariableInfo（ ）方法返回包含针对所需脚本变量VariableInfo对象的长度为1的数组。构造器声明：

- 变量名应为element。
- 其类为java.lang.Object。
- JSP解析器应为变量生成一个声明。
- 变量应在体评估过程期间对JSP页面可视，但其后不一定可视。

下面EnumTest.jsp页面给出活动中的标签：

```
<%@ page session="false" %>
<%@ page import="java.util.*" %>
<%@ taglib prefix="util" uri="/WEB-INF/tlds/util.tld" %>

<%--
      The scriptlet below loads the properties object.
      It could just as easily be loaded from a file.
--%>
<%
   Properties flavors = new Properties();
   flavors.setProperty("Vanilla", "The perennial favorite");
   flavors.setProperty("Chocolate", "Rich and smooth");
   flavors.setProperty("Strawberry", "Dazzling and fruity");
%>

<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR><TH>Flavor</TH><TH>Description</TH></TR>

<%--
      The enumerate tag will evaluate its body
      for each item in the properties object.
--%>

<util:enumerate enumeration="<%= flavors.propertyNames() %>">
<%
   String description = flavors.getProperty((String) element);
%>
<TR>
   <TD><%= element %></TD>
   <TD><%= description %></TD>
</TR>
</util:enumerate>

</TABLE>
```

enumerate标签出现在文件结尾附近。其值从flavors.propertyNames（ ）返回的java.util.Enumeration中设置。注意element没有可视声明——它是具有固定名字的一个隐含变量，

精彩好书见www.book118.com

类似于request、response、session和其他在JSP环境下定义的隐含变量。页面使用element变量两次，一次在scriptlet，被用于取得一个属性值，一次在一个JSP表达式中，其字符串值被打印在一个HTML表格中。

下面检验JRun为enumerate标签生成的servlet代码，将之与图11-4显示的流线图比较是很有益的：

```
EnumerateTag enumTag = (EnumerateTag)
   JRunJSPStaticHelpers.createTagHandler
      (pageContext, "EnumerateTag");

enumTag.setPageContext(pageContext);
enumTag.setParent(null);
enumTag.setEnumeration( flavors.propertyNames() );
```

JSP页面指定enumeration属性值如下：

```
enumeration="<%= flavors.propertyNames() %>"
```

它被传递到标签处理器，并带有对其setEnumeration（ ）方法的调用。接着，生成的servlet调用doStartTag（ ）并检验其返回码。doStartTag（ ）读取enumeration的第一个元素，并使用setAttribute("element",list.nextElement())将其存储在页面上下文中，

```
int rc = enumTag.doStartTag();
JRunJSPStaticHelpers.checkStartVal
   ("EnumerateTag", rc, BodyTag.EVAL_BODY_TAG, 24);
```

如果enumeration不为空，doStartTag（ ）返回EVAL_BODY_TAG，通过评估体触发该元素：

```
if (rc == BodyTag.EVAL_BODY_TAG) {

   out = pageContext.pushBody();

   enumTag.setBodyContent((BodyContent)out);
   enumTag.doInitBody();

   do {
      java.lang.Object element =
         (java.lang.Object)
            pageContext.getAttribute("element");
```

设置完嵌套的体内容并调用doInitBody（ ），servlet进入一个do while循环，循环的第一个语句是在doStartTag（ ）中刚被设置的element变量的getAttribute（ ）：

```
   out.print("\r\n");

   String description =
      flavors.getProperty((String) element);
```

精彩好书见www.book118.com


```
        out.print("\r\n<TR>\r\n <TD>");

        out.print(element);
        out.print("</TD>\r\n <TD>");

        out.print(description);
        out.print("</TD>\r\n</TR>\r\n");
    }
    while (enumTag.doAfterBody() == BodyTag.EVAL_BODY_TAG);
```

element变量然后被用来打印表格条目，然后调用doAfterBody（），doAfterBody（）重复取得下一元素并将其设置为pageContext中的element属性，只要元素可以利用，doAfterBody（）就返回EVAL_BODY_TAG，引发下一循环步，在循环中向element设置一新值：

```
        out = pageContext.popBody();
    }

    if (enumTag.doEndTag() == Tag.SKIP_PAGE) {
        if (true)
            return;
    }
```

最后，enumeration元素耗尽，doAfterBody（）返回SKIP_BODY，循环终止。前一个JspWriter从栈中弹出，doEndTag（）扔出结果，如图11-5所示。

2) 同步脚本变量

当在一个TEI中定义一个脚本变量，JSP容器生成servlet代码不但定义变量，而且使用标签处理器中的变量值将其同步。记得标签处理器使用pageContext.setAttribute（）设置所需值。生成的servlet代码有一个相的应pageContext.getAttribute（）语句在标签生命期中每一“do”方法后向脚本变量赋值。以这种方式修改哪一变量取决于在TEI中其定义的范围。表11-7给出了赋值操作：

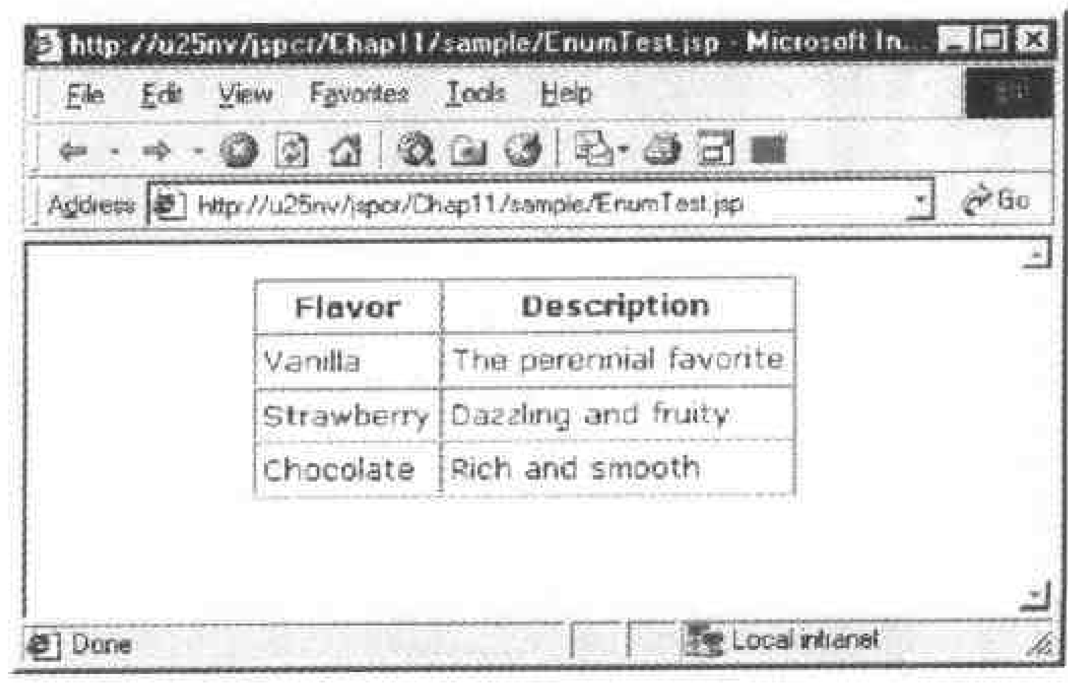


图11-5 enumerate标签测试的输出

精彩好书见www.book118.com

表11-7 脚本变量影响同步的范围

方 法	被同步变量的范围
doStartTag()	AT_BEGIN, NESTED
doInitBody()	AT_BEGIN, NESTED
doAfterBody()	AT_BEGIN, NESTED
doEndTag()	AT_BEGIN, NESTED, AT_END

11.10.2 标签属性有效性检验

除了定义脚本变量，TEI还提供public boolean isValid(TagData data)方法检验标签属性有效性。在此方法中，可以从data参数抽取属性名和取值列表，并检验其值是否有效。如果无效，可以返回false引发页面编译错误。例如，如果一个标签有几个属性，每一个都是可选的，但必需指定一个。不能单独使用TLD中的<attribute>元素指定此语义。isValid（ ）方法是实现此功能的惟一选择。

为导航属性列表，可以调用被传递到TagData参数中的isValid（ ）方法。表11-8列出一些可用方法。

isValid（ ）的最大缺点是不存在一种给出有意义的明显错误信息的方式。标签要不整个有效要不整个无效。

表11-8 TagData中一些可用方法

方 法	描 述
public void String getId()	如果为指定的，返回ID属性的名字
public Object getAttribute(String name)	给定一属性名，返回作为一个Object的属性值。如果属性值在转换时未知（也就是说，它被一请求时表达式指定），此方法返回TagData.REQUEST_TIME_VALUE
public String getAttributeString(String name)	给定一属性名，如果可能，返回作为java.lang.string的属性值
public Enumeration getAttributes()	返回标签属性名的枚举。和getAttribute（ ）一起使用，可以允许步进所有属性/取值对列表

11.11 协作标签

定制标签可以彼此交互执行有用的操作。一个常用的方案称为syntactic scoping。在其中标签处理器调用其父类中的方法。这一节给出此技术的扩展实例。

11.11.1 使用Syntactic Scoping

前面讲过标签可以嵌套。也就是说，一个标签可以用在另一个标签的体中。TagSupport类向标签处理器提供使用其findAncestor With Class( )方法查找其包围标签的标签处理器的方式。这是一个带有两个参数的静态方法——当前标签处理器的引用（this）和父标签有兴趣的类：

```
OuterTag ot = (OuterTag)
```

精彩好书见www.book118.com


```
    findAncestorWithClass(this, OuterTag.class);
if (ot == null)
    throw new JspException("No outer tag found");
```

一旦找到父标签，其所有的公有方法均可直接调用。下面一节阐述如何使用这种技术。

11.11.2 例子：switch标签

这里使用syntactic scoping来仿效Java语言的switch...case结构。这里需要三个标签：

- switch 其体定义switch逻辑的外部标签。此标签有一value属性，定义要被测试的条件和判断哪个case块应被执行。
- case 表示一种可能的case块的标签。这里给其两个属性：一个指定要匹配的精确值，另一个指定一个子字符串。第3个属性指定比较是否为大小写敏感。这里使用一个TEI类的isValid ( ) 方法检验前两个属性中的其中一个。
- default 如果其他case块均未通过执行块。

以下是所需的TLD接口：

```
<tag>
   <name>switch</name>
   <tagclass>jspcr.taglib.util.SwitchTag</tagclass>
   <bodycontent>JSP</bodycontent>
   <info>
      The enclosing tag for a switch/case block
   </info>

   <attribute>
      <name>value</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>

<tag>
   <name>case</name>
   <tagclass>jspcr.taglib.util.CaseTag</tagclass>
   <teiclass>jspcr.taglib.util.CaseTEI</teiclass>
   <bodycontent>JSP</bodycontent>
   <info>
      A case block to be included in the body of a switch
   </info>

   <attribute>
      <name>match</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
```

精彩好书见www.book118.com


```
    <attribute>
          <name>contains</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
       <name>caseSensitive</name>
       <required>false</required>
       <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

<tag>
    <name>default</name>
    <tagclass>jspcr.taglib.util.DefaultTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
       The default case included in the body of a switch
    </info>
</tag>
```

逻辑并不特别复杂。switch标签对其value属性和boolean型completed属性提供公有访问方法。completed属性跟踪一个case块是否已经匹配给定值并满足switch。下面是标签处理器：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
* A tag that emulates the switch ... case construct.
* Within the body of this statement there can be
* any number of case tag, including one default tag.
* The first one that matches the text is executed,
* and the rest are bypassed.
*/
public class SwitchTag extends TagSupport
{
   // The value attribute. This is the text that
   // case statements will compare to.
   private String value;
   public void setValue(String value)
      { this.value = value; }
   public String getValue()
      { return value; }
```

精彩好书见www.book118.com


```
    // A flag that indicates whether the switch statement
    // is complete. This happens when one of the case
    // statements matches the value and is executed.

    private boolean complete;
    public void setComplete(boolean complete)
       {this.complete = complete; }
    public boolean isComplete()
       { return complete; }

    /**
    * No real setup is required. All this method
    * needs to do is return EVAL_BODY_INCLUDE
    */
    public int doStartTag() throws JspException
    {
       return EVAL_BODY_INCLUDE;
    }
}
```

case标签也相当简单。它使用findAncestorWithClass（）找到其包围的switch标签。case标签首先调用switch标签的isComplete（）方法查看任何其他case是否已经满足switch。如果是，返回SKIP_BODY，这样其体不被执行。否则，它调用switch标签的getValue（）方法检索匹配字符串。如果匹配成功，case标签使用setComplete（true）满足switch并返回EVAL_BODY_INCLUDE。下面是标签处理器列表：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
* The body of this tag will be executed if it
* satisfies the condition specified in its attributes
* with respect to the value of the enclosing switch tag
*/
public class CaseTag extends TagSupport
{
   // Value of an exact string to be matched

   private String match;
   public void setMatch(String match)
      { this.match = match; }

   // Value of a substring that could be contained
   // in the switch tag's value
```

精彩好书见www.book118.com


```
private String contains;
public void setContains(String contains)
{ this.contains = contains; }

// Value of a boolean flag that indicates whether
// the match or comparison should be case sensitive.

private boolean caseSensitive;
public void setCaseSensitive(String flag)
{
   caseSensitive = new Boolean(flag).booleanValue();
}

public int doStartTag() throws JspException
{
   // Find the enclosing switch tag so that we
   // can call its methods

   SwitchTag switchTag = (SwitchTag)
      findAncestorWithClass(this, SwitchTag.class);

   // If the switch has already been satisfied,
   // skip the body of this statement

   if (switchTag.isComplete())
      return SKIP_BODY;

   // Test for an exact match, if the match attribute
   // was specified

   if (match != null) {

      String parentValue = switchTag.getValue();
      if (!caseSensitive)
         parentValue = parentValue.toUpperCase();

      String thisValue = match;
      if (!caseSensitive)

         thisValue = thisValue.toUpperCase();

      // If exact match, claim the switch

      if (parentValue.equals(thisValue)) {
         switchTag.setComplete(true);
         return EVAL_BODY_INCLUDE;
```

精彩好书见www.book118.com


```
         }

         // Otherwise, ignore the body

         return SKIP_BODY;
      }

      // Test for an substring match, if the contains attribute
      // was specified

      if (contains != null) {

         String parentValue = switchTag.getValue();
         if (!caseSensitive)
            parentValue = parentValue.toUpperCase();

         String thisValue = contains;
         if (!caseSensitive)
            thisValue = thisValue.toUpperCase();

         // If parent value contains this substring,
         // claim the switch

         if (parentValue.indexOf(thisValue) != -1) {
            switchTag.setComplete(true);
            return EVAL_BODY_INCLUDE;
            }

            // Otherwise, ignore the body

            return SKIP_BODY;
         }

         return SKIP_BODY;
      }
}
```

TEI检验match属性或contains属性之一是否已被指定，但不是两者均被指定：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
* Validates the attributes of a case tag
*/
public class CaseTEI extends TagExtraInfo
```

精彩好书见www.book118.com


```
{
    public boolean isValid(TagData tagData)
    {
        // The tag must contain either the match attribute
        // or the contains attribute, but not both.

        boolean noMatch =
            (tagData.getAttribute("match") == null);

        boolean noContains =
            (tagData.getAttribute("contains") == null);

            return (noMatch != noContains);
    }
}
```

default标签处理器工作方式类似于case，除了其匹配条件总是true。default工作方式并不完全类似于其Java副本，因为并未确保最后一定被执行，除非它最后被编码。下面是源码：

```
package jspcr.taglib.util;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
* The body of this tag will be executed if no other
* case tag has been encountered that satisfied
* the enclosing switch tag.
*/
public class DefaultTag extends TagSupport
{
    public int doStartTag() throws JspException
    {
        // Find the enclosing switch tag so that we
        // can call its methods

        SwitchTag switchTag = (SwitchTag)
            findAncestorWithClass(this, SwitchTag.class);

        // If the switch has already been satisfied,
        // skip the body of this statement

        if (switchTag.isComplete())
            return SKIP_BODY;

        // Otherwise, claim the switch

        switchTag.setComplete(true);
```

精彩好书见www.book118.com


```
        return EVAL_BODY_INCLUDE;
    }
}
```

结合起来，这些标签可以测试一个条件并执行所需块。下面JSP页面阐述了其使用过程：

```
<%@ page session="false" %>
<%@ taglib prefix="util" uri="/WEB-INF/tlds/util.tld" %>

<%
   String value = request.getParameter("value");
   if (value == null)
      value = "B";
%>
<H3>The value is <%= value %></H3>
<util:switch value="<%= value %>">

   <util:case match="A">
      <H3>The match="A" case block was selected</H3>
   </util:case>

   <util:case contains="B">
      <H3>The contains="B" case block was selected</H3>
   </util:case>

   <util:default>
      <H3>None of the case blocks were selected</H3>
   </util:default>

</util:switch>
```

当运行JSP页面，给定一个参数值为A，结果如图11-6所示。参数值为beauty（包含B，大小写不敏感），结果如图11-7所示。最后，如果值为C，不匹配任何case块，结果页面为图11-8所示。

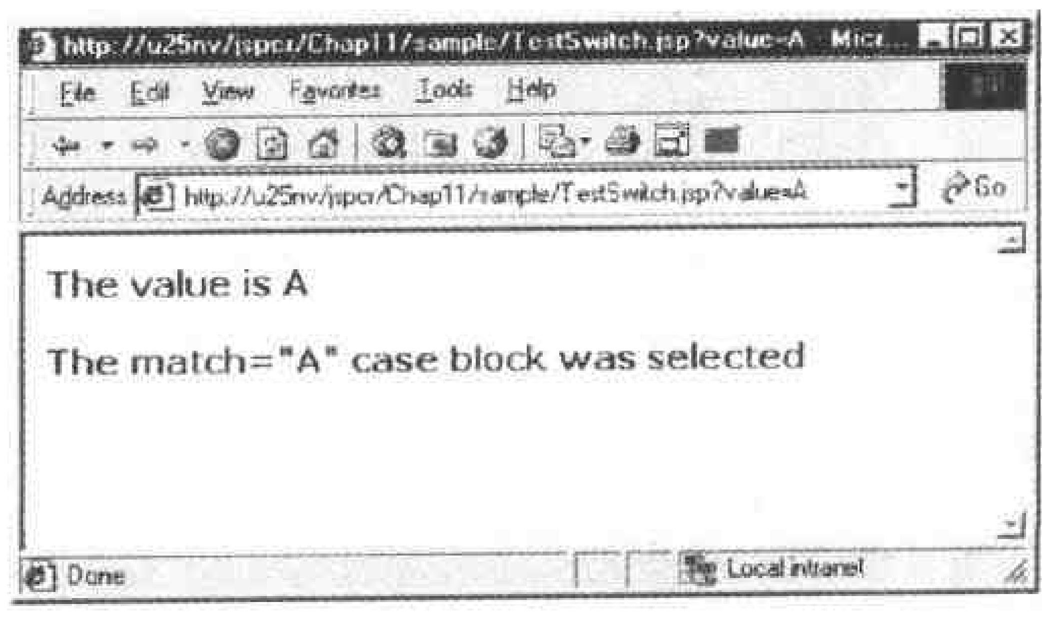


图11-6 value=A的switch测试

精彩好书见www.book118.com

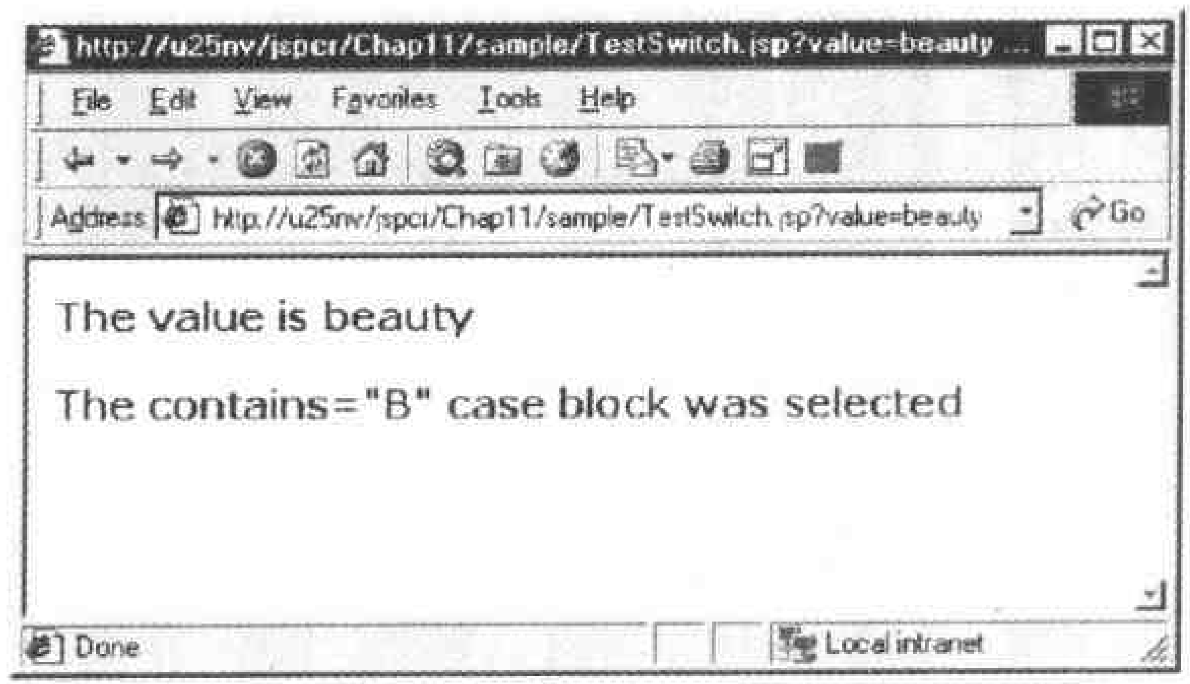


图11-7 value=beauty的switch测试

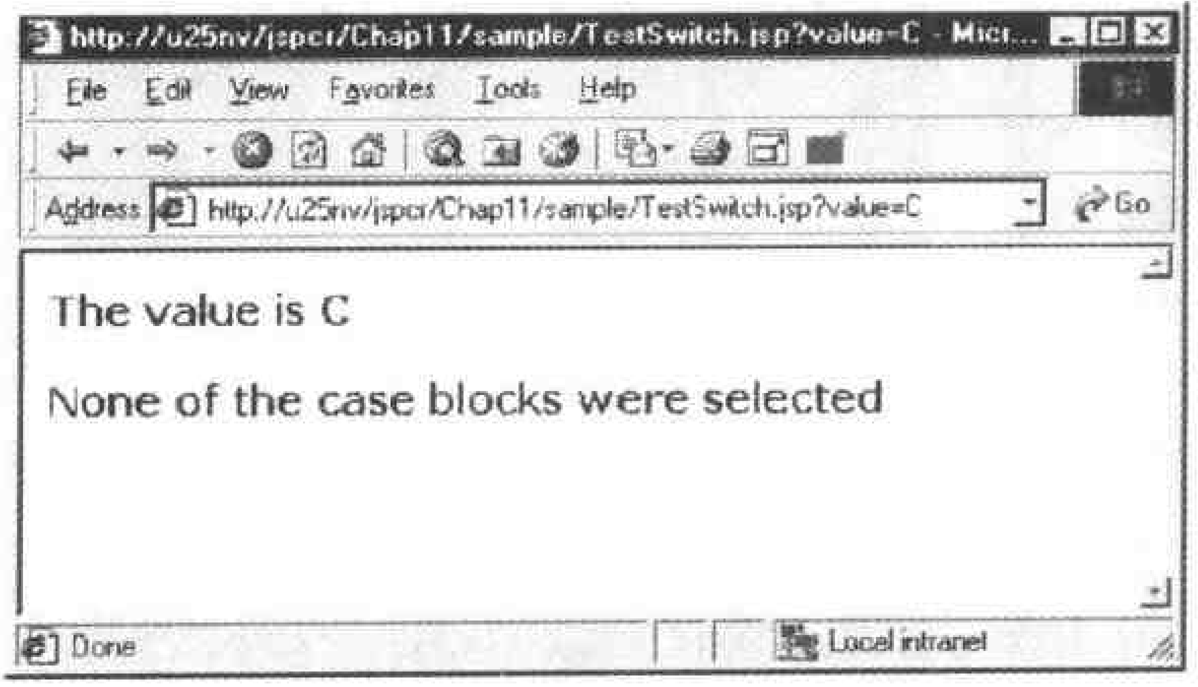


图11-8 value=C的switch测试

11.12 数据库查询例子的实现

本章最后是一个开始时给出的数据库查询例子的实现。

```
<db:connect url="mydatabase">

   <db:runQuery>
      SELECT *
      FROM FD_GROUP
      WHERE FdGp_Desc LIKE '%F%'
      ORDER BY FdGp_Cd
   </db:runQuery>

   <table border="1" cellpadding="3" cellspacing="0">
```

精彩好书见www.book118.com


```
    <tr><th>Food Group Code</th><th>Description</th></tr>
  <db:forEachRow>
    <tr>
      <td><db:getField name="FdGp_Cd"/></td>
      <td><db:getField name="FdGp_Desc"/></td>
    </tr>
  </db:forEachRow>
  </table>

</db:connect>
```

11.12.1 所需标签

存在4个协作标签：

connect　打开一个数据库连接并管理隐含Statement和ResultSet对象

runQuery　读取其体中的一个SQL语句，通知connect标签执行它

forEachRow　ResultSet上的一个循环

getField　检索命名域的当前值

11.12.2 标签库描述器

这些标签的TLD如下：

```
<tag>
   <name>connect</name>
   <tagclass>jspcr.taglib.jdbc.ConnectTag</tagclass>
   <bodycontent>JSP</bodycontent>
   <info>Opens a database connection and manages
         a Statement and ResultSet object</info>

   <attribute>
      <name>url</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>

</tag>
```

connect标签打开一个连接，并管理Statement和ResultSet对象。这些对象不是脚本变量；它们可通过库中其他标签访问。当遇到结束标签时连接被关闭。语法为：

<db:connect url="mydatabase">

驱动器类名可容易地作为另一属性加入，这里隐含了此过程以简化JSP。

```
<tag>
   <name>runQuery</name>
   <tagclass>jspcr.taglib.jdbc.RunQueryTag</tagclass>
```

精彩好书见www.book118.com


```
    <bodycontent>JSP</bodycontent>
    <info>Reads and executes the SQL statement
          in the tag body</info>

</tag>
```

runQuery标签从体中读取一个SQL语句并使用包围连接标签创建的语句对象执行它。它可能只用于connect标签的体中。runQuery标签的语法为：

<db:runQuery>sql statement</db:runQuery>

结果集也由connect标签管理。

```
<tag>
   <name>forEachRow</name>
   <tagclass>jspcr.taglib.jdbc.ForEachRowTag</tagclass>
   <bodycontent>JSP</bodycontent>
   <info>Iterates over the current result set</info>
</tag>
```

forEachRow标签循环当前结果集，以便getField标签可以访问其值。这只用在一个runQuery标签后一个connect标签的体中。语法为：

```
<db:forEachRow>
...
</db:forEachRow>
```

```
<tag>
   <name>getField</name>
   <tagclass>jspcr.taglib.jdbc.GetFieldTag</tagclass>
   <bodycontent>empty</bodycontent>
   <info>Retrieves a field
      from the current result set row</info>

   <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>

</tag>
```

getField标签从当前结果集行中读取一个域并返回该值的字符串形式。这只出现在forEachRow标签的体中。其语法为：

```
<db:getField name="fieldName"/>
```

11.12.3 标签处理器

这里要开发4个标签处理器。因为未定义脚本变量，故不需要任何TEI类。

精彩好书见www.book118.com

1. connect

connect标签将一个数据库URL作为一个属性，因此标签处理器需要一个实例变量和setUrl( )方法：

```
package jspcr.taglib.jdbc;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.sql.*;

/**
* ConnectTag
*/
public class ConnectTag extends TagSupport
{
   public static final String DRIVER_CLASS
      = "sun.jdbc.odbc.JdbcOdbcDriver";
   // =============================================
   // Tag attributes
   // =============================================

   private String url;

   public void setUrl(String url)
   {
      this.url = url;
   }
```

如果定义了Connection，Statement和ResultSet对象并提供每一个的公有访问方法，以及一个公有runQuery方法：

```
   // =============================================
   // JDBC objects managed by this tag
   // =============================================

   private Connection con;
   private Statement stmt;
   private ResultSet rs;

   public Connection getConnection() { return con; }
   public Statement getStatement() { return stmt; }
   public ResultSet getResultSet() { return rs; }

   /**
   * Runs a query
   * @param sql an SQL statement
   */
```

精彩好书见www.book118.com


```
public void runQuery(String sql)
   throws SQLException
{
   rs = stmt.executeQuery(sql);
}
```

整个数据库操作都保持在开始和结束标签之间，因此用两个生命期方法管理启动和退出：

```
// ============================================
// Lifecycle methods
// ============================================

/**
* Loads the driver class, opens a database
* connection, and creates a Statement object
*/
public int doStartTag() throws JspException
{
   con = null;
   try {
      Class.forName(DRIVER_CLASS);
      con = DriverManager.getConnection(url);
      stmt = con.createStatement();
   }
   catch (Exception e) {
      throw new JspException(e.getMessage());
   }
   return EVAL_BODY_INCLUDE;
}

/**
* Closes the connection and other JDBC objects
*/
public int doEndTag() throws JspException
{
   try {
      if (rs != null) {
         rs.close();
         rs = null;
      }
      if (stmt != null) {
         stmt.close();
         stmt = null;
      }
      if (con != null) {
         con.close();
         con = null;
      }
   }
```

精彩好书见www.book118.com


```
        catch (SQLException e) {
            throw new JspException(e.getMessage());
        }
        return EVAL_PAGE;
    }
}
```

2. RunQuery

runQuery标签做3件事情：

- 从体中抽取一个SQL语句。
- 找到包围的connect标签。
- 执行connect.runQuery（ ）方法。

源码如下：

```
package jspcr.taglib.jdbc;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.sql.*;
import java.util.*;

/**
* RunQueryTag
*/
public class RunQueryTag extends BodyTagSupport
{
    /**
    * Reads the SQL statement in the body of the tag
    * and asks the connect tag to execute it.
    */
    public int doEndTag() throws JspException
    {
        // Get the SQL to be run

        String sql = bodyContent.getString();
        if (sql == null)
            throw new JspException
            ("No SQL statement found in body of runQuery tag");

        sql = sql.trim();
        if (sql.equals(""))
            throw new JspException
            ("Empty SQL statement found in body of runQuery tag");

        // Locate the enclosing connect tag
```

精彩好书见www.book118.com


```
        ConnectTag connectTag = (ConnectTag)
            findAncestorWithClass(this, ConnectTag.class);
        if (connectTag == null)
            throw new JspException
            ("runQuery must be used in the body of a connect tag");

        // Tell the connect tag to run the query

        try {
            connectTag.runQuery(sql);
        }
        catch (SQLException e) {
            throw new JspException(e.getMessage());
        }

        // Normal return

        return EVAL_PAGE;
    }
}
```

3. ForEachRow

这是一个循环标签，类似于本章前面给出的enumerate标签。像runQuery，它首先使用findAncestorWithClass（）得到connect标签的引用。从connect标签处理器实例中，它使用getResultSet（）取得结果集，使用称为incrementRow（）的私有便捷方法将结果集步进到下一行。使用doStartTag（）或doAfterBody（）检测结果集的结尾并相应返回SKIP_BODY。

```
package jspcr.taglib.jdbc;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.sql.*;

/**
* ForEachRowTag
*/
public class ForEachRowTag extends BodyTagSupport
{
    private ConnectTag connectTag;

    /**
    * Sets up for the first iteration of the result set
    */
    public int doStartTag() throws JspException
    {
        connectTag = (ConnectTag)
```

精彩好书见www.book118.com


```
            findAncestorWithClass(this, ConnectTag.class);
        if (connectTag == null)
            throw new JspException
            ("forEachRow must be in the body of a connect tag");
        return incrementRow();
    }

    /**
    * After each row has been evaluated,
    * increment the result set and indicate
    * when end is reached.
    */
    public int doAfterBody() throws JspException
    {
    return incrementRow();
    }

    /**
    * When end tag is reached, dump the results
    */
    public int doEndTag() throws JspException
    {
        try {
            pageContext.getOut().print(bodyContent.getString());
        }
        catch (IOException e) {
            throw new JspException(e.getMessage());
        }
        return EVAL_PAGE;
    }

    /**
    * Convenience method for getting the next row.
    * Used by both <CODE>doStartTag</CODE>
    * and <CODE>doAfterBody</CODE>.
    * Returns EVAL_BODY_TAG if a row exists,
    * SKIP_BODY otherwise.
    */
    private int incrementRow() throws JspException
    {

        ResultSet rs = connectTag.getResultSet();

        if (rs == null)
            throw new JspException
            ("No result set found - no query has been run");

        // Get the next row or indicate that there are no rows
```

精彩好书见www.book118.com


```
        boolean hasNext = false;
        try {
            hasNext = rs.next();
        }
        catch (SQLException e) {
            throw new JspException(e.getMessage());
            }
            return (hasNext) ? EVAL_BODY_TAG : SKIP_BODY;
        }
}
```

4. GetField

像其副本runQuery和forEachRow，getField使用包围的connect标签中的公有方法。它从ResultSet中抽取指定域取值并将之发送到当前输出写入者。

```
package jspcr.taglib.jdbc;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.sql.*;

/**
* GetFieldTag
*/
public class GetFieldTag extends TagSupport
{
    private String name;

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

    /**
    * Returns the value of the specified
    * field in the result set as a string.
    */
    public int doEndTag() throws JspException
    {
        // Get the enclosing Connect tag

        ConnectTag connectTag = (ConnectTag)
          findAncestorWithClass(this, ConnectTag.class);
        if (connectTag == null)
           throw new JspException
           ("getField must be in the body of a connect tag");
```

精彩好书见www.book118.com


```
        // Get its current result set

        ResultSet rs = connectTag.getResultSet();
        if (rs == null)
           throw new JspException
           ("No result set exists - no query has been run");

        try {

           // Get the specified field and write it
           // to the output stream

           String value = rs.getString(name);
           JspWriter out = pageContext.getOut();
           out.print(value);
        }
        catch (SQLException e) {
           throw new JspException(e.getMessage());
        }
        catch (IOException e) {
           throw new JspException(e.getMessage());
        }

        // Normal completion

        return SKIP_BODY;
     }
}
```

当运行数据库查询时，产生结果如图11-9所示。

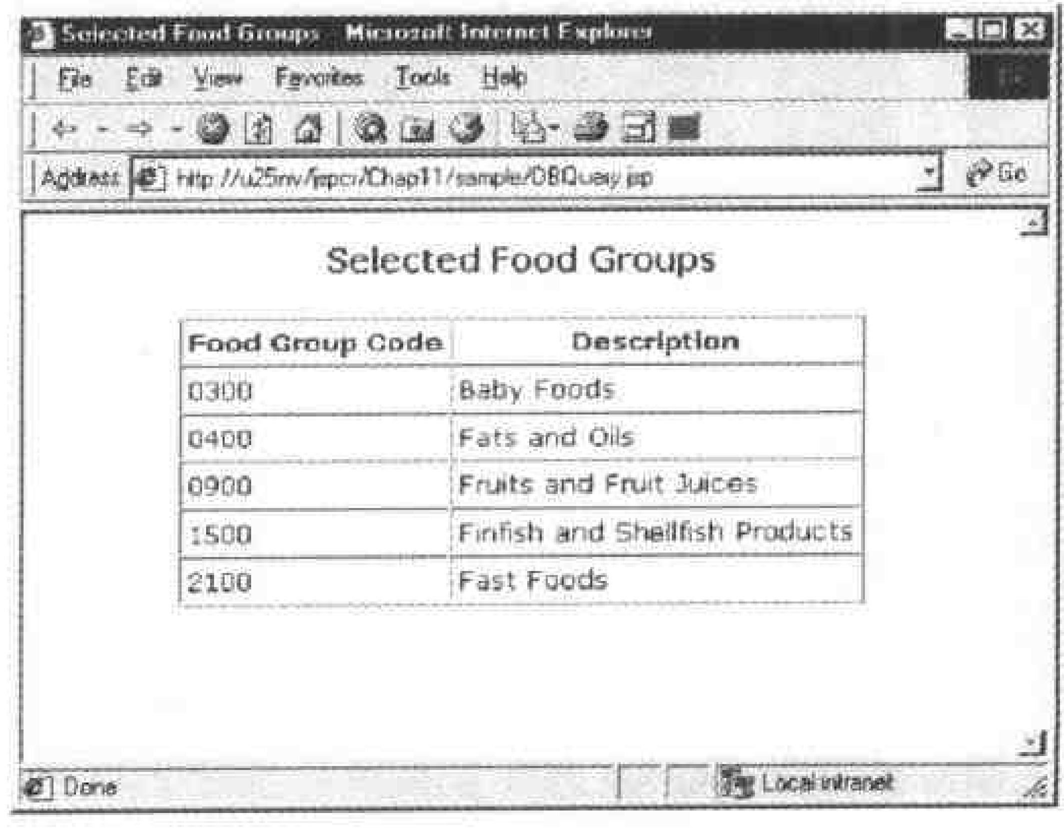


图11-9 数据库查询输出

精彩好书见www.book118.com

11.13 小结

定制标签是扩展JSP编程环境的一种非常好并且健壮的方法，允许开发小组向不熟悉Java编程的页面设计者提供其使用的指定应用的JSP标签的工具箱。标签的功能可通过称为标签处理器的一个Java类实现，它提供在标签生命期内各种位置JSP容器调用的方法。相关标签集可以协同完成复杂任务。标签处理器和配置信息集被打包在标签库中，它是一个厂家独立的结构，很方便发布。

精彩好书见www.book118.com

精彩好书见www.book118.com

第三部分 JSP 行 为

这一节讲述JSP如何与Java环境主要的组件协同工作。给出HTML窗体和JDBC数据库访问的背景知识，再讨论高级主题，如会话管理、线程、JavaBean和XML。第17章和第18章分别讲述调试和发布Web应用，第19章给出一个嵌入本书所有技术的完整的事例。

第12章 HTML 窗 体

大部分应用需要用户在Web环境某一点上进行输入。此输入通常来源于HTML窗体，像其文件副本一样，HTML窗体由一系列标签和以逻辑顺序排列的输入域组成。当用户填写了一个窗体，点击确认按钮后，输入域名字和取值被传入到相关处理Web服务器的程序中。图12-1给出一个典型的窗体。

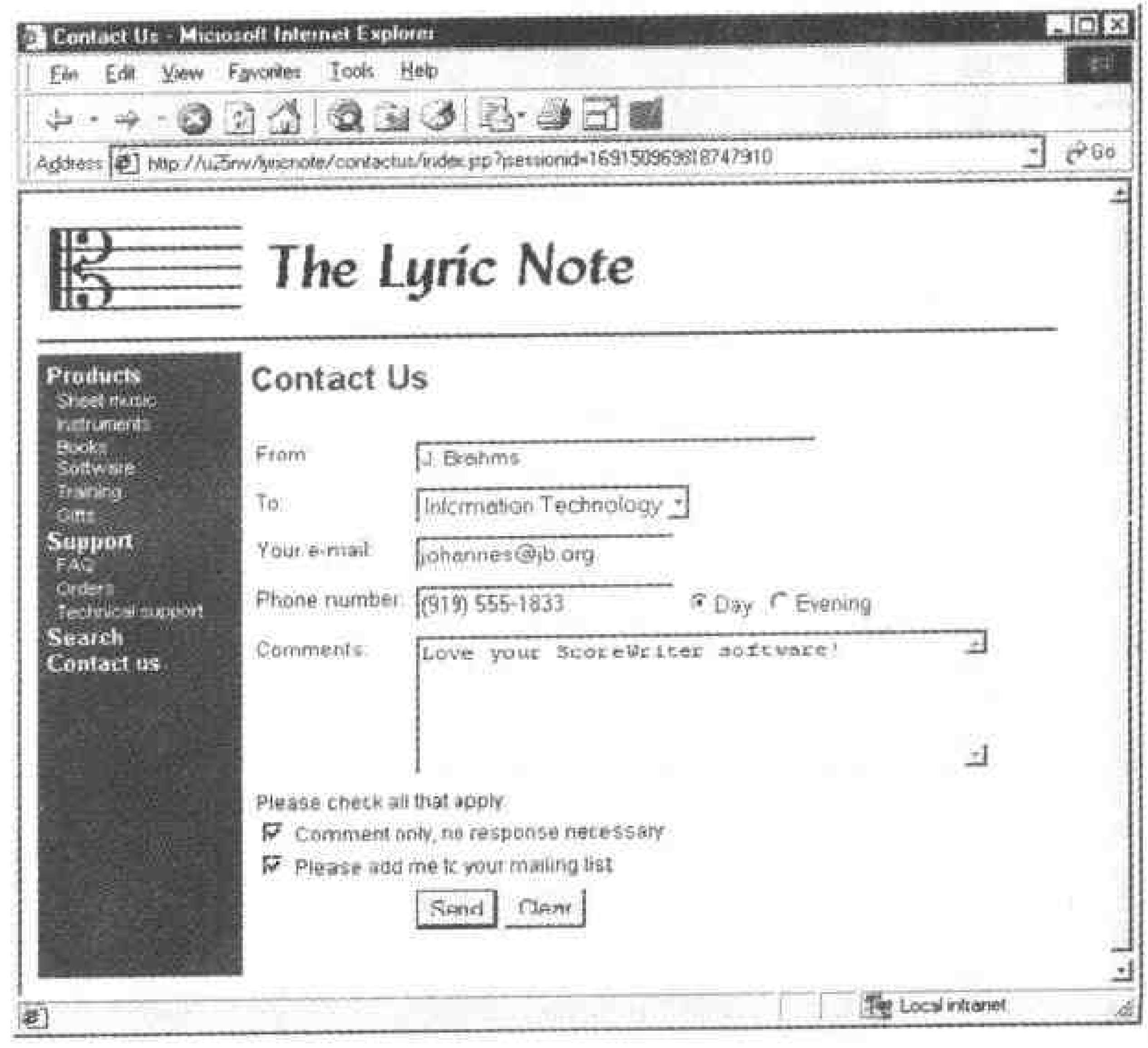


图12-1 用于反馈Web站点的HTML窗体

精彩好书见www.book118.com

HTML提供一个基本的元素或输入控制集，用于容纳大量的数据输入需求。集合包括：

- 文本输入元素　单行或多行输入的矩形框。
- 选择菜单　下拉式列表框中显示的选项列表。它可以有一个显示在屏幕上的外部窗体和与选择条目相关的内部编码值。
- 按钮　模拟控制面板上一个按钮的矩形控件。经常用于初始化一个命令，如确认窗体或清除输入域。
- 检查框　为检查或未检查，开或关的小方块。检查框可指定取值为是或否。
- 单选按钮　类似于检查框，单选按钮表明是或否，但它们通常用于互斥组，选择其中一个表明其他均未选择。
- 文件选择元素　为用户指定载入的文件名的控件。典型情况，此控件包含弹出文件选择对话框的一个浏览按钮。
- 隐藏元素　用于创建带有常量取值参数的不可视元素。

在一个HTML窗体中使用的元素集在HTML规范中被标准化和文档正规化。（在http:/www.w3.org/TR/html4/上可得到）此规范是由W3C制定的，当新的特性出现时，他们会进行定期地修改。正如预想的，并不是所有的浏览器都实现所有的特性，本章使用HTML4.01作为描述窗体工作方式的基础，但也只是给出广泛支持的大多数元素和属性。

本章讲述在HTML窗体中使用的元素，如何使用它们以及浏览器如何递交窗体。讨论了如何确认窗体及使之有效，然后大概讲解服务器端窗体处理。

12.1　FORM元素

FORM元素是HTML窗体后的基本结构，主要有3种用途：

- 将输入元素依照语法组织在一起。
- 标识处理窗体过程的服务器端程序。
- 指定发送的数据值及在哪个窗体中。

在HTML中用<FORM>标签表示FORM元素。此元素语法如下：

```
<FORM
    action="url"
    method="method"
    enctype="content type"
    accept-charset="charsets"
    accept="content types"
    name="form name">
...
</FORM>
```

属性和其取值在下一节给出。

FORM元素的属性

在下面列出的6个属性中，惟一需要的是action。实际上，其他属性和方法很少用到。以下

精彩好书见www.book118.com

是各属性的定义。

1. action属性

当窗体完成时，用户点击确认按钮，Web浏览器创建一个HTTP请求，将所有窗体数据打包，发送到某Web服务器上的一个程序。此程序在action属性中被指定。

Action属性值必须是一个HTTP URI。（见http://www.ietf.org/rfc/rfc2396.txt查询URI的正规定义）。其形式为：

```
[http://<servername>][/]<path>
```

为了确认窗体，Web浏览器打开一个至指定服务器的套接字连接（缺省服务器为下载HTML页面的服务器）。并使用指定路径进行HTTP请求，典型情况下路径指向一个servlet、JSP页面或CGI程序。此程序或者URI本身或者在一个输入流中接收HTTP请求和窗体数据，具体依据使用的HTTP方法（见下面method）。

可以在URI上指定一个查询字符串。这时查询字符串中编码的参数和在窗体主体中指定的参数合并。通常没有必要，因为一个隐藏域可以实现同样的功能。

2. method属性

HTTP协议提供许多请求类型用于文件传输、下载、删除和诊断操作。当然，在HTML窗体中只有GET和POST有效。method就是指定这两种方法中的一种。

HTTP GET或POST请求通常由Web服务器解释为在URI中命名文档的检索请求。当Web服务器已经配置成处理servlet、CGI程序或其他服务器端脚本环境时，它将对资源的请求解释为对这些程序调用的请求。这样程序产生的输出（典型为一HTML文档）被发送回请求者，就好象它是一个根据名字进行请求的静态文档一样。

在一个HTML窗体中使用GET和POST之间的差别是它们如何向服务器进程提供输入数据：

- GET　窗体值作为一个查询字符串被附加到URI中。
- POST　窗体值提供为输入流。

两种方法都可用，servlet API可以进行明智的选择，应该考虑几个特性。因为GET请求使得输入数据被附加到请求URI中，它们作为浏览器地址行上的名字/取值对和在Web服务器注册信息中是可视的，这使得GET对发送诸如密码等敏感数据不可取。另外，一些服务器和浏览器也对它们可以处理的URL的长度有些限制。还有，在HTTP规范中GET请求被认为是等幂的，这表明它们可以安全地被重复而没有无法预料的副作用。在一定环境下，这意味着服务器可以告诉客户端再次使用资源已存在的副本，而不是向其发送新副本。这不是窗体输入响应所想要的结果。为此，对请求方法，POST通常是一种更好的选择。

Method属性是可选的，如果未指定，缺省为GET。该属性值可以用大写或小写方式指定。

3. enctype属性

窗体的输入值可以有几种方式传输到服务器。将取值解码放到一个数据流中的方法称为content类型，当使用POST方法时在entype属性中指定。存在两种常用解码：

- `application/x-www-form-urlencoded`
- `multipart/form-data`

精彩好书见www.book118.com

Application/x-www-form-urlencoded 如果未指定(通常不指定)，enctype属性值为Application/x-www-form-urlencoded。（见http://www.freesoft.org/CIE/RFC/1738/index.htm上RFC1738，查阅URL解码的完整论述）。此解码技术包含下列步骤：

1) 将输入元素名字或取值中所有非空格特殊字符[1]替换成%xx，这里xx为相应ASCII字符码的两位16进制值。

2) 使用加号（+）替换任意空格。

3) 将每一名字和取值结果对用其间的（=）结合在一起。

4) 按在窗体中出现的次序连接结果name=value字符串，其间用&符号分隔。

服务器进程按相反步骤解码，从而恢复出原始的参数名和取值。

例如，如果一个窗体包含名为product的输入域，，其值为"Great Music @Home"，另一域为quantity，取值为3，解码字符串为：

```
product=Great+Music%40Home&quantity=3
```

URL解码的目的是将字符串附加到URL变得安全。如果空格、引号标记或其他分隔字符出现在URL中，它们可能会使处理其的程序发生混乱。

Multipart/form-data Multipart/form-data 是支持文件下载常用的一种新方法（RFC 2388(http://www.ietf.org/rfc/rfc2388.txt）给出HTML窗体对multipart/form-data的使用）。此解码中，每个输入域和其值都在输入流里自己的块中被发送，一个特定分隔字符串称为边界，标记每一块的开始和结束。边界是一个Web浏览器选择的伪随机的字符串，在Conyent-Type头标中指定。每一块都是一个或多个HTTP头标，接着是空行，然后是包含输入域取值的一行。域名在Content-Disposition头标中传递。

因此，使用前面的例子，如果窗体包含输入域product，取值为"Great Music @Home"，quantity，取值为3，浏览器将会创建一个HTTP请求，包含内容如下：

```
POST /someURI HTTP/1.0
Content-Type: multipart/form-data;boundary=7d025a324c0138
Content-Length: 178

--7d025a324c0138
Content-Disposition: form-data; name="product"

Great Music@Home
--7d025a324c0138
Content-Disposition: form-data; name="quantity"

3
--7d025a324c0138--
```

---

1 关于这些字符的定义有一些不同意见，RFC1738将其描述为不同于"$-_.+!*'(),"的任意非字母数字的字符。然而，Internet Explorer、Netscape Navigator和java.net.URLEncoder只限制排除"-_.*"。无论如何，因为编码和解码都由达成一致意见的程序进行，这不是一个问题。

精彩好书见www.book118.com

multipart/form-data解码的主要弊端是当前servlet API中并不直接支持它，也就是说，单个域名不能使用getParameterNames（）检索，其值不能使用getParameterValues（）读取。读取和解析输入流以获得信息是必须的。

4. acept-charset属性

在HTTP中使用字符集是用来将字节集转换为字符集的一系列规则。最常用的字符集是ISO-8859-1，ASCII的一个超级，额外映射了127-255范围的字节值。如果使用了accept-charset，它应该包含字符集取值列表，并由逗号或空格分隔。

accept-charset其目的是指明服务器程序可以解释和处理的字符集。实际上，此属性很少用到，并几乎被Internet Explorer和Netscape Navigator忽略。

5. accept属性

一个FORM标签可以指出其服务器端设计处理程序可以接受的内容类型。如果指定，accept属性必须包含内容类型的一个逗号分隔的列表，如text/html或image/jpg。这只是对浏览器的一种暗示，浏览器可以选择忽略它（大部分情况下是这样）。

6. name属性

窗体可以有一个名字，通过它窗体可以在文档其他位置<SCRIPT>段中被引用。例如，下列窗体：

```
<form
   method="post"
   action="diag/ShowParms.jsp"
   name="prodform"
   onsubmit="return checkform();"
<pre>
Product: <input type="text" name="product"
Quantity: <input type="text" name="quantity"
</pre>
</form>
```

这里checkform（）是一个确认窗体输入域有效性的JavaScript函数。它可以分别读取两个输入域的值，如document.prodform.product.value和document.prodform.quantity.value。

12.2 窗体输入元素

在〈FORM〉...〈/FORM〉标签体中，给出了每个数据域。HTML由每个域的描述性标记和创建所需控件的HTML标签组成。可视化设计通常由一个HTML表格完成，因此域标记和控件是水平和垂直分布的。下列HTML产生窗体如图12-1所示：

```
<form method="post" action="diag/ShowParms.jsp">
<table border "0" cellpadding="3" cellspacing="0">


   <tr valign="top">
      <td>From:</td>
      <td><input name="from" type="text" size=32></td>
```

精彩好书见www.book118.com


```
</tr>

<tr valign="top">
    <td>To:</td>
    <td>
        <select name="to" size=1>
            <option value="CS">Customer Service
            <option value="EX">Executive
            <option value="FI">Finance
            <option value="HR">Human Resources
            <option value="IT">Information Technology
            <option value="MK">Marketing
            <option value="FA">Facilities
            <option value="PC">Purchasing
            <option value="SP">Shipping
        </select>
    </td>
</tr>

<tr valign="top">
    <td>Your e-mail:</td>
    <td><input name="email" type="text" size=20></td>
</tr>

<tr valign="top">

    <td>Phone number:</td>
    <td>
    <input name="phone" type="text" size=20>
    <input name="dayphone" type="radio" value="1" checked>Day
    <input name="dayphone" type="radio" value="0">Evening
    </td>
</tr>

<tr valign="top">
    <td>Comments:</td>
    <td>
    <textarea name="comments" rows=5 cols=40></textarea>
    </td>
</tr>

<tr valign="top">
    <td colspan=2>
    <font size=-1>
    Please check all that apply: <br>
    <input name="category" type="checkbox" value="1">
```

精彩好书见www.book118.com


```
        Comment only, no response necessary <BR>
        <input name="category" type="checkbox" value="2">
           Please add me to your mailing list
        </font>
        </td>
    </tr>

    <tr valign="top">
        <td> </td>
        <td>
           <input type="submit" value="Send">
           <input type="reset" value="Clear">
        </td>
    </tr>

    <input type="hidden" name="remoteHost"
    value="209.170.132.238">
       <input type="hidden" name="userAgent"
    value="Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)">
    </table>

    </form>
```

在本节中我们使用此例中的窗体。

4个HTML标签集用来创建窗体输入元素：

- <INPUT> 用于几个特定元素类型的一般标签。
- <SELECT>和<OPTION> 用于创建菜单或下拉式列表框。
- <TEXTAREA> 用于多行文本输入。
- <BUTTON> 用于创建确认、重置和一般性用途的按钮。此标签仍未被广泛支持。为此，这里不予介绍。

12.2.1 使用INPUT标签创建的元素

HTML INPUT标签用于许多元素类型。它有大量的属性，其中许多只针对特定域类型。下面语法给出对大多数类型通用的属性。

```
<INPUT
     type="text | password | checkbox | radio | submit |
     reset | file | hidden | image | button"
     name="name"
     value="value"
     size="size">
```

这里属性定义如下：

- type= “type” 指出使用的特定域类型，如果未指定，缺省为文本类型。
- name= “name” 用于设置域标识符，以便它可以被脚本或样式单表示。它也是服务器程序

精彩好书见www.book118.com

检索域的名字。

- value= “value” 用来设置域的初始值。
- size= “size” 指出域可视宽度，单位为点或字符数（对于文本域）。

除了这些属性，INPUT标签可以指定当一定事件发生时调用浏览器中脚本行为的事件处理器。事件处理器属性值是脚本代码的片段，典型情况为JavaScript。事件处理器属性在所处理的事件后被命名，前缀为“on”：

- onfocus 当用户使用tabs键或点击鼠标到该域时发生，因此它接受键盘聚焦事件。
- onblur 当用户使用tabs键或点击鼠标离开该域时发生，因此它失去键盘聚焦事件。
- onselect 当某文本被选择时发生（Netscape Navigator不支持）。
- onchange 当用户改变控件值，然后通过在域外点击或跳格确认改变时发生。

当确认窗体时，浏览器抽取每个控件的名字和取值，依据在〈FORM〉标签中指定（或隐含）的解码类型转换它们，并将结果数据流发送到服务器进程。

下面各节讲解每个<INPUT>类型：

1. 文本

这是最简单和常用的<INPUT>类型，用于输入单行文本。其语法如下：

```
<INPUT
    type="text"
    name="name"
    value="value"
    size="size"
    maxlength="maxlength">
```

属性含义如下：

- type= “text” 指出这是一个文本控件。
- name= “name” 指定脚本来引用此控件的名字。它也是服务器端程序检索文本的名字。
- value= “value” 用来设置初始文本值。当窗体与窗体处理程序的输出在同一页面时，此属性很有用。
- size= “size” 指出显示宽度，如果未指定，浏览器选择缺省大小，但可能不合适。最好指定一个满意值。
- maxlength= “maxlength” 设置域中键入字符数的限制。

TEXT输入元素典型情况显示在矩形框中，例如，下列HTML：

```
<tr valign="top">
<td>From:</td>
<td><input name="from" type="text" size=32></td>
</tr>
```

结果如下：

From J Brahms

2. 密码

精彩好书见www.book118.com

文本控件的变体是密码控件。惟一的区别是用户在屏幕上键入的字符为不可见。代替的是掩码字符，如显示星号代替每个键入的字符。密码输入标签的语法如下：

```
<INPUT
      type="password"
      name="name"
      value="value"
      size="size"
      maxlength="maxlength">
```

其属性与在文本输入标签中具有同样的含义。

密码输入元素典型地作为矩形框显示。例如，一个技术支持应用中用到的下列HTML：

```
<tr valign="top">
   <td>Support ID:</td>
   <td><input name="suppid" type="password" size=10></td>
</tr>
```

输出结果如下：

Support ID:

注意 这是最小限度的安全窗体，设计只用来防止人眼看到在密码域中键入的内容。但传送到服务器进程的字符是用户原来键入的字符，并不是星号。使用密码控件并不能加密或隐藏域值，只是使其为不可视。

3. 检查框

检查框控件用来表示一个为真或假的选项。语法如下：

```
<INPUT
     type="checkbox"
     name="name"
     value="value"
     checked>
```

属性定义如下：

- type="checkbox" 指出这是一个检查框控件。
- name="name" 指定脚本用来引用此检查框的名字。它也是服务器端程序检索检查框值的名字。如果一组检查框表示同一域的多个值，并未互斥，则可以有相同的名字。
- value="value" 用来设置当此框被选中时返回的值。如果未指定，缺省值为两个字符的字符串on（即on，非真，1，yes或被检查）。
- checked 如果给出，则表明检查框具有初始选择状态。

检查框还支持一个额外的事件处理器属性：

- onclick 当用户点击检查框时发生。

并不支持onchange事件。

检查框通常表示为一个小的方型框，并具有出现或不出现的检验标记，表示出控件的布尔

精彩好书见www.book118.com

值。例如，如下HTML：

```
<tr valign="top">
    <td colspan=2>
    <font size=-1>
    Please check all that apply: <br>
    <input name="category" type="checkbox" value="1">
       Comment only, no response necessary <BR>
    <input name="category" type="checkbox" value="2">
       Please add me to your mailing list
    </font>
    </td>
</tr>
```

显示结果如下：

Please check all that apply:
☑ Comment only no response necessary
☑ Please add me to your mailing list

4. 单选按钮

单选按钮控件，像检查框一样，用于表示真或假的选项。不同的是，一组单选按钮在操作上是互斥的。当选中其中一个，任何具有同样名字属性的其他选项均被清除。从此意义上讲，此操作像汽车无线电上的按钮——当推进去一个，其他的就被弹出来。单选按钮语法如下：

```
<INPUT
     type="radio"
     name="name"
     value="value"
     checked>
```

属性定义如下：

• type= "radio"　指出这是一个单选按钮控件。
• name= "name"　指定脚本用来引用此单选按钮的名字。它也是服务器端程序检索单选按钮值的名字。如果一组单选按钮表示同一域的多个互斥值，则可以有相同的名字。
• value= "value"　指定当此按钮处于选择状态时窗体返回的值。这是所需属性。
• checked　如果给出，则表明此单选按钮是一组中最初被选择的。

像检查框一样，单选按钮支持onclick事件，但不支持onchange事件。

单选按钮通常表示为一个小圆圈，内中一点表示选中或未选中，表示出控件的布尔值。例如，如下HTML：

```
<tr valign="top">
     <td>Phone number:</td>
     <td>
     <input name="phone" type="text" size=20>
     <input name="dayphone" type="radio" value="1" checked>Day
     <input name="dayphone" type="radio" value="0">Evening
```

精彩好书见www.book118.com


```
    </td>
</tr>
```

显示结果如下：

Phone number (919)555-1833 ◉ Day ○ Evening

5. 确认

为了向服务器确认一个窗体，必须有一种方式表明用户已经输入完数据，这就是确认按钮的角色。确认按钮与其他控件不同，因为它通常不会向发送到服务器的数据流添加内容。确认按钮语法如下：

```
<INPUT
    type="submit"
    name="name"
    value="value">
```

其属性定义如下：

- type= "submit"　指出这是一个确认控件。
- name= "name"　指定脚本用来引用此确认按钮的名字。从服务器观点上讲它通常是不必要的。因为很明白，确认按钮必须被点击，否则不提交窗体。然而，如果在窗体中有几个确认按钮，每一个都有不同的值，那么此属性就很有用了。
- value= "value"　指定显示在按钮上的值。（如果也给出名字属性，则被窗体返回）。如果未指定，在Internet Explorer 5.x和Netscape Communicator 4.75中缺省为“Submit Query”。其他浏览器可能会给出不同的缺省值。

此控件支持onclick事件，但不支持onchange事件。

确认按钮通常表现为一个矩形按钮，并且上面具有value属性指定的文本。例如，如果一个确认按钮编码如下：

```
<input type="submit" value="Send">
```

输出结果如下：

Send

6. 重置

与确认很相近的是重置。它用于将窗体内所有控件设置为其初始状态。像确认按钮一样，它通常不会向发送到服务器的数据流填加内容。其语法如下：

```
<INPUT
    type="reset"
    value="value">
```

其属性定义如下：

- type= "reset"　指出这是一个重置控件。
- value= "value"　指定显示在按钮上的值。（如果也给出名字属性，则被窗体返回）。如果未指定，缺省为“Reset”。

精彩好书见www.book118.com

此控件支持onclick事件，但不支持onchange事件。

重置按钮通常表现为一个矩形按钮，并且上面具有value属性指定的文本。例如，如果一个重置按钮编码如下：

```
<input type="reset" value "Clear">
```

输出结果如下：

Clear

7. 文件

某些应用被上载到服务器的文件调用。例如，技术支持应用可能要处理用户发送的栈轨迹。广告牌系统可能接受文件提交。像这样应用结尾处出现的Web页面可以使用文件输入控件。语法如下：

```
<INPUT
    type="file"
    name="name"
    size="size">
```

其属性定义如下：

- type= "file"　指出这是一个文件控件。
- name= "name"　指定脚本用来引用此文件控件的名字。它也是服务器端程序所知域的名字，虽然可能格式不同，下面马上会讲到。
- size= "size"　指出文件名输入域的可视宽度。

文件控件通常表现为一个文本域，并且带有一个相关的浏览按钮。文件名直接在文本域中输入，或者是用户点击浏览按钮使用一个文件选择对话框：

Browse...

一个窗体要使用文件控件必须进行两处变动：

- 请求方法必须是POST。
- 解码类型（在〈FORM〉标签enctype属性中指定）必须是multipart/form-data。

如果这些条件不满足，控件仍可显示，仅被当作通常的文本输入域——发送到服务器的只是文件名。

服务器程序也必须进行许多重大改变。它必须能够使用本章前面讨论的multipart/form-data解码格式抽出文件内容及其他非文件参数值。

注意　主要的限制是Web受服务器可以载入文件的大小。加入这些限制的意图是防止使用大文件导致WEB服务器服务攻击瘫痪。

作为文件上载应用的例子，假定LyricNote.com发起一个竞赛，用户可上载音乐作品的MIDI[1]文件。竞赛的胜利者将收到LyricNote发送的音乐软件产品。图12-12显示了输入窗体。

1 MIDI（音乐设备数字接口）使用描述音符而不是实际录下音乐的一种文件格式。其内容由MIDI兼容的设备或媒体播放器重新创建。

精彩好书见www.book118.com

下述HTML生成窗体：

```
<form method="post"
      action="http://u25nv/lyricnote/servlet/midi_contest"
      enctype="multipart/form-data"
      >
<table border="0" cellpadding="3" cellspacing=0>
   <tr valign="top">
      <td colspan="2">
         Are you a budding composer?
         Submit a MIDI file of your own composition
         for a chance to win a copy of <B>ScoreWriter 4.5</B>.
         Click
         <A HREF="product/midi_contest/rules.html">here</A>
         for official rules.
      </td>
   </tr>

   <tr valign="top">
      <td>Your name:</td>
      <td><input name="name" type="text" size=32></td>
   </tr>

   <tr valign="top">
      <td>Your e-mail:</td>
      <td><input name="email" type="text" size=20></td>
   </tr>

   <tr valign="top">
      <td>Title of composition:</td>
      <td><input name="title" type="text" size=48></td>
   </tr>

   <tr valign="top">
      <td>MIDI file name:</td>
      <td><input type="file" name="midifile" size=32></td>
   </tr>

   <tr valign="top">
      <td> </td>
      <td>
         <input type="submit" value="Send">
         <input type="reset" value="Clear">
      </td>
   </tr>
```

精彩好书见www.book118.com


```
</table>
</form>
```

当窗体被确认时，浏览器生成一个HTTP请求，带有multipart/form-data格式的一个数据流，包含内容如下：

```
Content-Type: multipart/form-data;
    boundary=---------------------------7d01012174012c
Content-Length: 1507

-----------------------------7d01012174012c
Content-Disposition: form-data; name="name"

S. Vetter
-----------------------------7d01012174012c
Content-Disposition: form-data; name="email"

vetter@lyricnote.com
-----------------------------7d01012174012c
Content-Disposition: form-data; name="title"

It-B-Gone
-----------------------------7d01012174012c
Content-Disposition: form-data; name="midifile";
filename="C:\my_midi_files\Itbgon.mid"
Content-Type: audio/mid

MThd... (binary data not shown)
-----------------------------7d01012174012c--
```

可以看出给出了4个输入域，每个都由分界字符串分隔成自己的块。每一块都有指定域名的Content-Disposition头标。包含上载文件的最后一块还有一个在其Content-Disposition头标上的属性filename，给出客户端系统上的初始文件名，以及一个Content-Type头标，指出文件数据是二进制格式audio/mid。该服务器程序可以解析数据流并抽取域的取值和文件内容。

8. Hidden

并不是所有的输入均来自用户，至少不是直接来自用户。有些窗体可以使用硬编码或动态生成的常量。带有type="hidden"的INPUT元素可用于此目的。其语法如下：

```
<INPUT
type="hidden"
name="name"
value="value">
```

其属性含义如下：

- type="hidden" 指出这是一个隐藏控件。
- name="name" 指定脚本用来引用此控件的名字。它也是服务器程序用来检索文本的名字。

精彩好书见www.book118.com

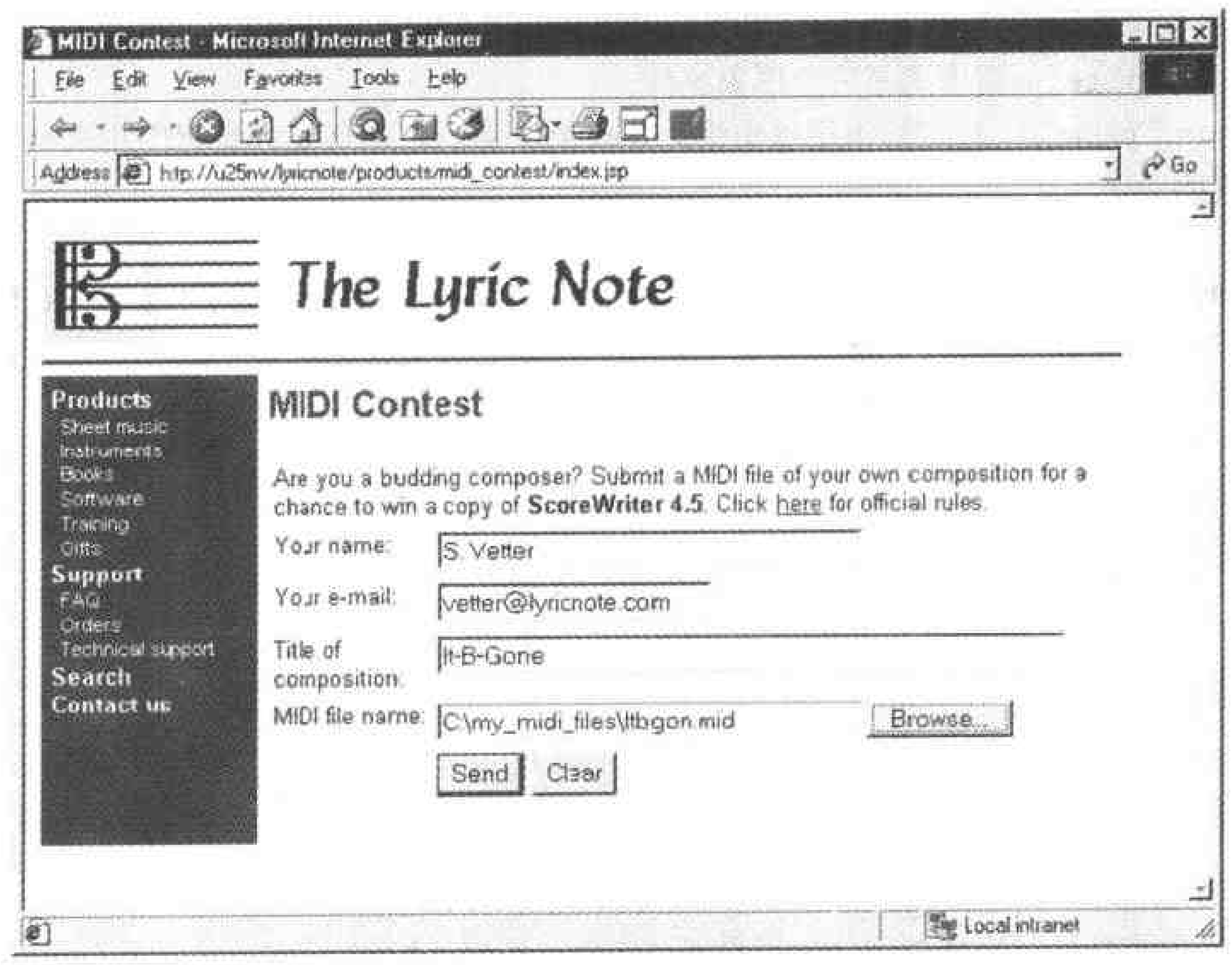


图12-2 包含一个文件上载输入域的窗体

• value= "value" 必需使用以设置一个初始文本值。

不出所料，隐藏域没有可视表示。其惟一目的是创建带有常量值的参数。它是一种事务处理代码，发送servlet用它作为其他类表格的关键字。更常见的是，隐藏域具有一个创建HTML的servlet或JSP页面动态生成的值。例如，

```
<input type="hidden" name="remoteHost"
   value="<%= request.getRemoteHost() %>">

<input type="hidden" name="userAgent"
   value="<%= request.getHeader("user-agent") %>">
```

作为窗体中的隐藏域保存Web服务器运行的IP地址或主机名以及一个表示浏览器软件和版本号的字符串。这些域对解决窗体问题的技术支持人员可能会有用。

9. image

一个image可使用户使用鼠标点击而不是使用键盘键入的输入域。其包含的信息是在图象上鼠标点击发生的位置。image输入类型可用于此目的。其语法如下：

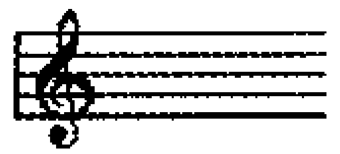
```
<INPUT
    type="image"
    name="name"
    src="imageurl">
```

精彩好书见www.book118.com

其属性含义如下：

- type= “image”　指出这是一个图象控件。
- name= “name”　指定脚本用来引用此控件的名字。它也是服务器程序用来检索鼠标点击位置的名字。
- src= “imageurl”　是图象文件的URL。

在数据流中为一个图象控件创建了两个参数，分别对应点击的x和y坐标。坐标以点为单位，并且相对于图象的左上角（0，0）。参数名组成是图象控件名加上“.x”或“.y”。例如，一个窗体包含一个音乐五线谱符号，提示用户点击所需音符：



如果图象输入控件编码如下：

```
<input type="image" name="staff" src="images/clef/tcstaff.png">
```

如果用户点击中线C上的音符C，位置大约是（108，40），结果数据流如下：

```
staff.x=108&staff.y=40
```

点击图象控件使得窗体被确认。用户不一定要点击确认按钮。

10. button

除了确认和重置按钮，存在一种通用的按钮输入类型，其语法如下：

```
<INPUT
    type="button"
    name="name"
    value="value">
```

属性定义如下：

- type= “button”　指出这是一个按钮控件。
- name= “name”　指定脚本用来引用此按钮的名字。
- value= “value”　按钮上必需显示的值。

要使用此控件，必需定义onclick事件处理器属性。然后一个JavaScript函数可以引用按钮名字和取值。

12.2.2　使用select和option创建的元素

select和option标签可结合在一起创建一个菜单项的滚动列表。参考图12-1中顾客反馈的例子，用户从一个下拉式列表中选择窗体的目标单元格。用于创建此列表的HTML如下：

```
<select name="to" size=1>
   <option value="CS">Customer Service
   <option value="EX">Executive
```

精彩好书见www.book118.com


```
    <option value="FI">Finance
    <option value="HR">Human Resources
    <option value="IT">Information Technology
    <option value="MK">Marketing
    <option value="FA">Facilities
    <option value="PC">Purchasing
    <option value="SP">Shipping
</select>
```

select标签的语法如下：

```
<select name="name" size="number"multiple> options </select>
```

这里属性含义为：

- name= “name” 设置服务器程序可引用此列表的名字。
- size= “number” 指出一次性可视的元素数，列表框的高度。如果number为1，列表为下拉式菜单。
- multiple 如果被指定，用户可以选择多个条目。

注意，< select >标签必需通过</ select >结束。

select列表的核心是具有相关值和描述的option标签集。这样的列表经常由数据库查询动态生成。option标签语法如下：

```
<option value="value" selected> text </option>
```

这里

- value= “value” 如果选中此条目，指定窗体返回值。如果未指定此属性，返回option标签体。
- selected 如果给出，预选择条目。

在开始和结束标签之间的文本（指体）是列表框中实际显示的内容。结束</option>标签不是必需的，常被省略。

当确认窗体时，被选中的条目值是与select元素相关的值。上一个例子中，如果用户选择列表中最后一个条目，数据流中返回值如下：

```
to=SP
```

如果在<select>标签上指定了multiple属性，，用户选择了多个条目，在与不同值相关的数据流中参数名会出现多次。因此，如果用户选择了shipping，还有finance和marketing，数据流如下：

```
to=FI&to=MK&to=SP
```

12.2.3 textarea元素

尽管text和password是单行输入域，textarea元素可以接受多行输入。这使得textarea元素对输入注释或可能多于一行的其他自由格式文本非常有用。高度和宽度均可指定，浏览器在必要时可加入滚动条。textarea语法如下：

```
<textarea name="name" rows="number" cols="number">
```

精彩好书见www.book118.com


```
... text ...
</textarea>
```

这里

- name= "name"　设置应被脚本和服务器程序知道的该域的名字。
- rows= "number"　指定文本区可视范围内的行数。这不是对列表框中实际行数的限制，它只是当滚动列表时一次显示的行数限制。
- cols= "number"　指定文本区可视宽度的字符长度。这不是对列表框中实际列数的限制，而只是其显示宽度。

12.3　窗体有效性检验

用户接口编程具有内在复杂性。进行输入域有效性检验的代码量经常比执行实际功能的代码还要大。必需域必需被检测以确保非空，可选域必需设置缺省值。所有的域都需要验证其是否输入了可接受的值或算法集。某些域的检验还依赖于其他域的取值。

所有有效性检验均可由服务器程序完成，从响应时间的观点来看，往返的网络流量使其显得太贵时。为此，有效性检验最好使用诸如JavaScript脚本语言在客户端进行。市面上有各种JavaScript书籍，因此这里对JavaScript不进行探讨，但给出一个完整的例子。

带有有效性检验的Contact Us窗体

下面向"Contact Us"窗体实例加入最小的有效性检验，如图12-1所示。需要做3件事情如下：

1) 设置触发器

首先，当窗体被确认时需要某些代码被强制执行。为此，设置窗体标签中的onsubmit属性：

```
<form method="post"
      action="diag/ShowParms.jsp"
      onsubmit="return validate(this);">
```

在onsubmit属性中指定的字符串在窗体确认前被求值，只有为true时窗体才被确认。理论上，有效性检验代码可以直接作为onsubmit属性值输入，但调用一个函数并返回其值会更简单。当有效性检验需求改变时，加入新代码会更容易。

2) 加入脚本块

为了将JavaScript语句嵌入生成的HTML，必须将之用<SCRIPT>...</SCRIPT>标签围住。为确保这些标签被载入并评估，最好将之放入HTML的<HEAD>...</HEAD>。

3) 编写有效性检验函数

第3步是编写有效性检验函数：

```
function validate(frm) {
   if (!hasData(frm.from.value)) {
      alert("Please type your name in the 'From:' box");
      return false;
   }
   return true;
```

精彩好书见www.book118.com


```
   }

   function hasData(s) {
      if (s == null)
         return false;
      var n = s.length;
      for (var i = 0; i < n; i++) {
         var c = s.charAt(i);
         switch (c) {
            case ' ':
            case '\t':
            case '\n':
               continue;
            default:
               return true;
         }
      }
      return false;
}
```

这里给出两个函数：

- validate(frm) 给定一个窗体引用，执行所有必需的有效性检验，如果没有错误发生，返回true。
- hasData(s) 如果指定字符串有至少一个非空格字符，则返回true的通用函数。

这里惟一需要检验的域是用户名。此例中，只需检验其为非空格。如果用户在域中什么也没输入，结果如图12-3所示。

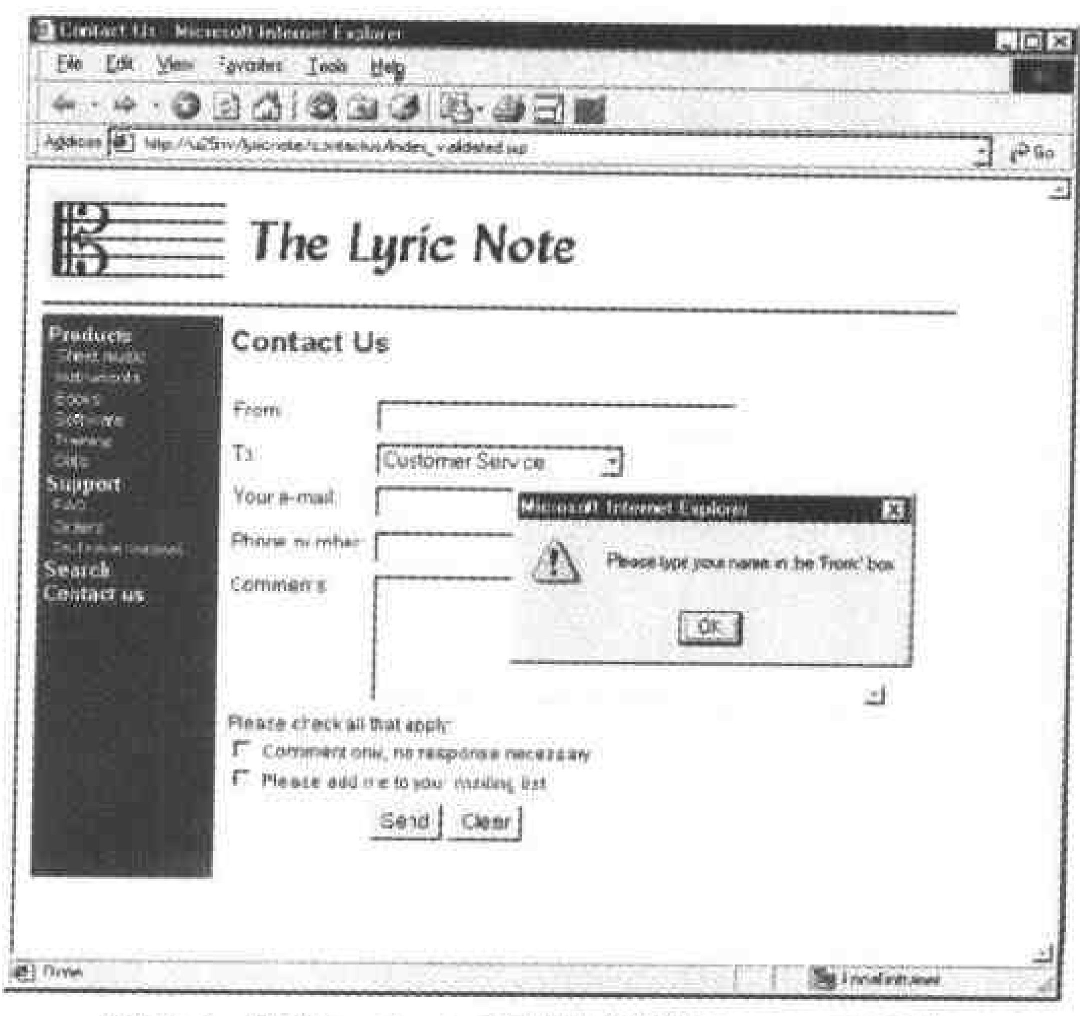


图12-3 具有JavaScript有效性检验的Contact Us窗体

精彩好书见www.book118.com

12.4 服务器端的窗体处理

许多处理窗体的模型都是在服务器端。第19章的事例深入探讨此问题。为了说明，开发一个只搜集请求参数并将之在可读的HTML表格中格式化的简单JSP页面。服务器程序为ShowParms.jsp，用于Contact Us实例：

```
<%@ page import="java.io.*,java.util.*" %>
<%@ taglib prefix="lyric" uri="/WEB-INF/tlds/taglib.tld" %>

<html>
<head>
<title>Parameter Values</title>
<base href="<lyric:baseURL/>">
<link rel="stylesheet" href="styles/diag.css">
</head>

<body>
<center>
<table border="1" cellpadding="3" cellspacing="1" width="600">
<tr>
<td colspan="2" align="center" class="header">
Parameter Values
</td>
</tr>
<tr><th>Name</th><th>Value</th></tr>
<%
    int currentRow = 0;
    Enumeration eNames = request.getParameterNames();
    while (eNames.hasMoreElements()) {
        String name = (String) eNames.nextElement();
        String[] values = request.getParameterValues(name);
        for (int i = 0; i < values.length; i++) {
            String value = values[i];
            currentRow++;
            String rowClass = "row" + (currentRow % 2);
%>
<tr valign="top">
<td align="right" class="<%= rowClass %>"><B><%= name %></B></td>
<td align="left" class="<%= rowClass %>"> <%= value %> </td>
</tr>
<%
        }
    }
%>
</table>
</center>
```

精彩好书见www.book118.com


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

程序开始有两个伪指令:

```
<%@ page import="java.io.*,java.util.*" %>
<%@ taglib prefix="lyric" uri="/WEB-INF/tlds/taglib.tld" %>
```

一个标签库被声明，设置前缀为“lyric”。这里只用到其中一个标签，返回Web应用的基本URL。当用于HTML< BASE >标签中时，此标签使得创建图象、样式单和应用中其他资源的相对和绝对链接更容易。

```
<base href="<lyric:baseURL/>">
```

程序的核心是调用getParameterNames( )，返回窗体域名的枚举并对这些名字并发进行循环，使用getParameterValues (Striing name) 检索其各自值。

```
Enumeration eNames = request.getParameterNames();
while (eNames.hasMoreElements()) {
    String name = (String) eNames.nextElement();
    String[] values = request.getParameterValues(name);
    ...
}
```

当图12-1中窗体被ShowParms.jsp确认时，结果如图12-4所示。

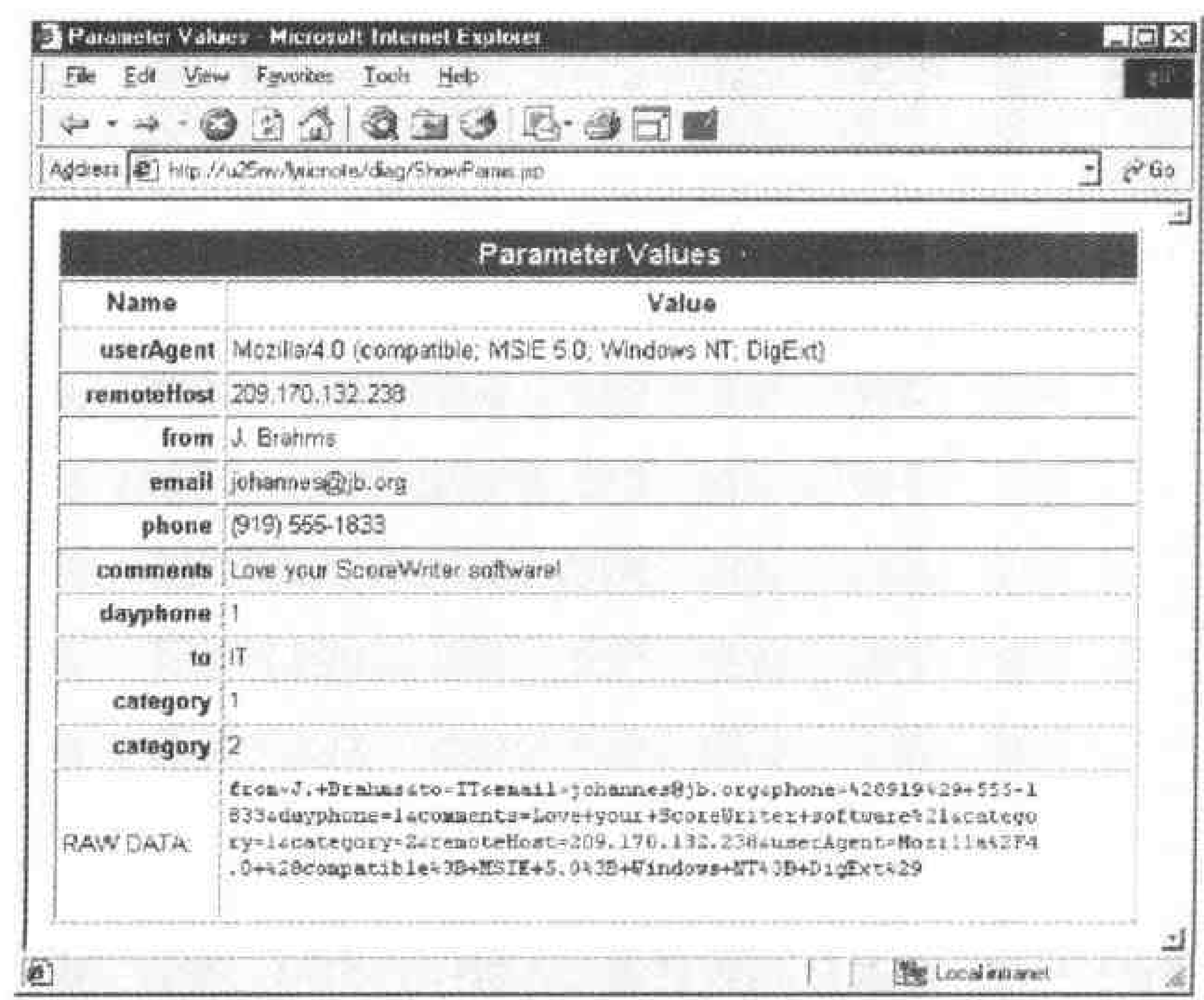


图12-4 从Contact Us窗体抽取的数据

精彩好书见www.book118.com

12.5 小结

HTML窗体提供了一个GUI环境，使用户很容易在服务器程序中进行工作和处理。对文本条目，菜单选择和Boolean选择存在许多控件，如检查框和单选按钮。使用一种备用的解码类型，窗体可以支持客户端的文件上载。在隐藏域中可以指定常量值。可以使用JavaScript编写的客户端函数对窗体进行有效性检验，节省了网络传输和处理时间。服务器程序，JSP页面或是servlet，可以使用servlet API检索参数名和取值并向确认窗体的浏览器返回结果。

精彩好书见www.book118.com

第13章 数据库访问

公司数据库是商业和大多数访问它的结果生成JSP页面的核心。在线零售商的Web站点可使其产品目录用于浏览。电影院的Web页面可以列出放映时间和电影信息。搜索引擎提示键盘键入内容，并返回匹配链接集合。

除了只读访问，许多JSP页面也可作为保存数据应用的前端。在一个购物卡收款功能中，按次序排列的条目列表必需被转换成其他系统处理的事务：定购、运输和结算。

Java使用JDBC[1]的技术处理数据库它是一种综合、通用方式。JDBC令各种使用SQL[2]的数据库管理系统进行通信成为可能。本章包含JDBC简介和基于Web的应用中使用的方式。内容包括JDBC驱动器、如何连接数据库，如何执行SQL语句和读取其结果。还描述了JDBC的健壮的事务处理及连接池机制。最后一节讨论了JDBC 2.0中的新特性和不能实现的情况。

13.1 JDBC简介

JDBC是Java程序和数据库管理系统之间的应用编程接口。像Oracle的Oracle Call Interface（OCI）或Microsoft的Open Database Connectivity（ODBC）一样，JDBC是一个调用级接口。这意味着程序可以使用方法或函数调用访问其特性，与被预编译器转换的嵌入SQL语句刚好相反。

程序员使用JDBC驱动器的Java类连接到一个数据库。存在上百个JDBC驱动器——至少有一个用于商业或共享数据库。一种特殊的JDBC驱动器，称为JDBC-ODBC桥，有可能把ODBC作为一种媒介，使得大量的ODBC驱动器可用于JDBC。

JDBC的最大优势是它对所有数据库管理系统提供一种标准接口。在一个Oracle数据库中的查询与在DB2、SQL Server或任何其他数据库中的查询区别很少或根本不需改变。通常保留下来的几点不同是数据类型名和对一些操作类型的支持。即使这些不同通常可以使用JDBC连接提供的元数据编程加以解决。

JDBC还简化从遗留系统到WEB应用的转换。嵌入式SQL产品，大约在1980年早期出现，大部分使用了可被JDBC调用复制的SQL语句和操作。在一批主框架COBOL应用中SQL的语法和语义在应用转换为Java时很少改变。

13.1.1 基本JDBC操作

使用JDBC并不难，依据要执行的任务，通常需要4个步骤：

1) 为DBMS载入一个JDBC驱动器。典型情况只调用指定驱动器类名的一个Class.forName（）

---

1 依据Sun Microsystems，JDBC并不是一个首字母缩写词。特别是，它并不代表Java数据库连接。

2 结构化查询语言（SQL）有很多书。最好的其中一本是SQL：完全参考，作者James R.Groff&Paul N.Weinberg，Osborne/McGraw-Hill出版，ISBN 0-07-211845-8。

精彩好书见www.book118.com

即可。

2) 使用此驱动器打开至一特定数据库的连接。这通过调用DriverManager类的getConnection(url)方法实现。url参数是指定使用的驱动器类和数据源的特定形式。

3) 通过连接发出SQL语句，一旦连接建立，它就可以创建Statement对象，通过该对象执行SQL命令。

4) 处理SQL操作返回的结果集。ResultSet接口提供步进每一行并对每一列取值的方法。

图13-1阐述了这4个步骤。

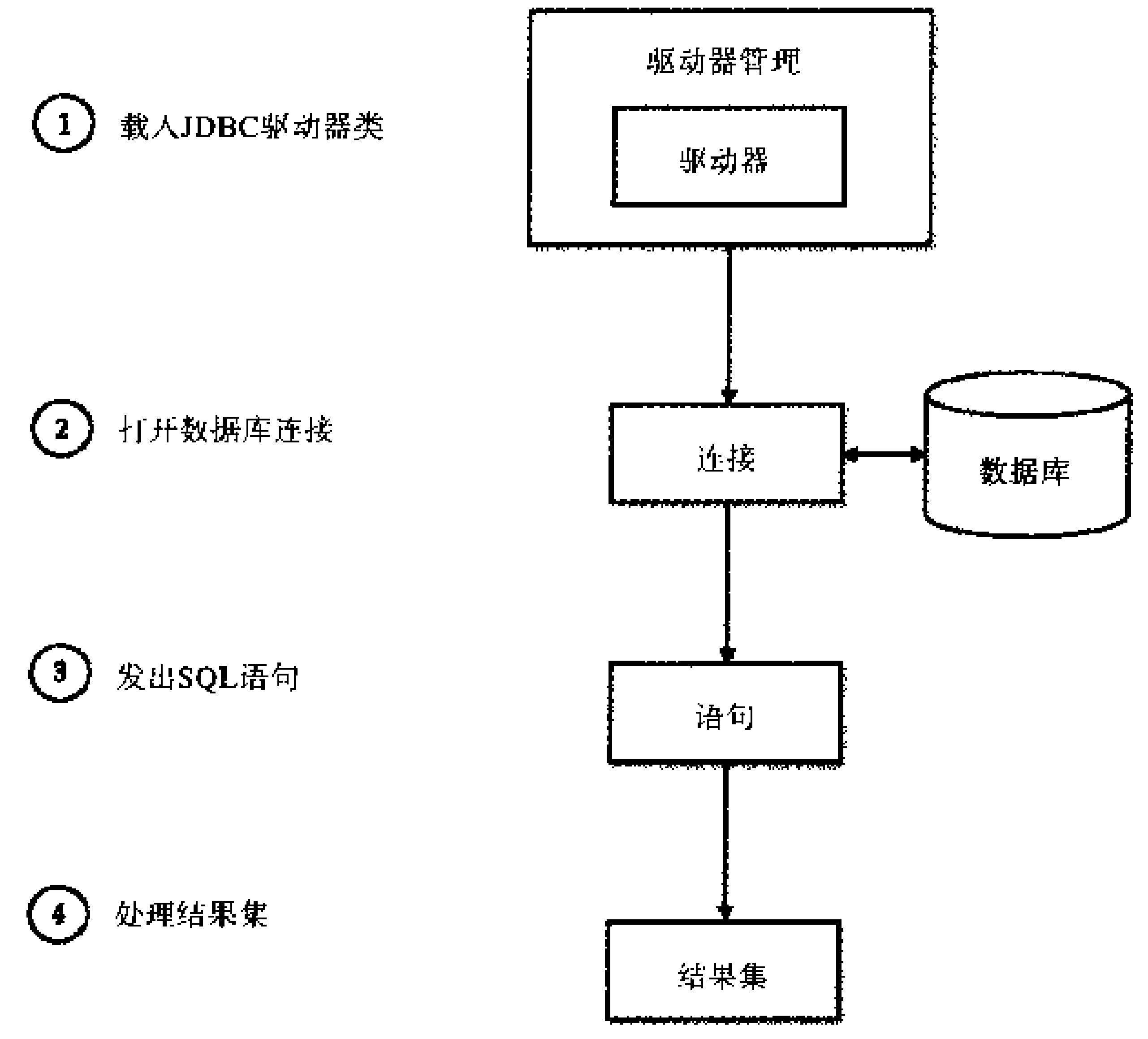


图13-1 基本JDBC操作中包含的4个步骤

使用JDBC 2.0，具有很大的灵活性。使用Java名称和目录接口（JNDI），一个应用可以通过一个名字服务中的名字查询DataSource对象，而不是硬编码驱动器类名和数据库URL。另外，JDBC 2.0结果集功能更多。它们可以按任意次序访问而不是按开始到结束顺序访问。它们可以被修改并将修改传递到下面的表格。它们也可以动态链接到其基本表格以便同时改变映射到的结果集。图13-2显示了JDBC 2.0数据库访问中包含的基本步骤。

13.1.2 基本JDBC类

JDBC接口包含在java.sql和javax.sql包中[1]。它主要由接口组成而不是集成类。因为每一厂家

1 javax.sql包含JDBC 2.0可选包API，以前称为JDBC 2.0标准扩展API。

精彩好书见www.book118.com

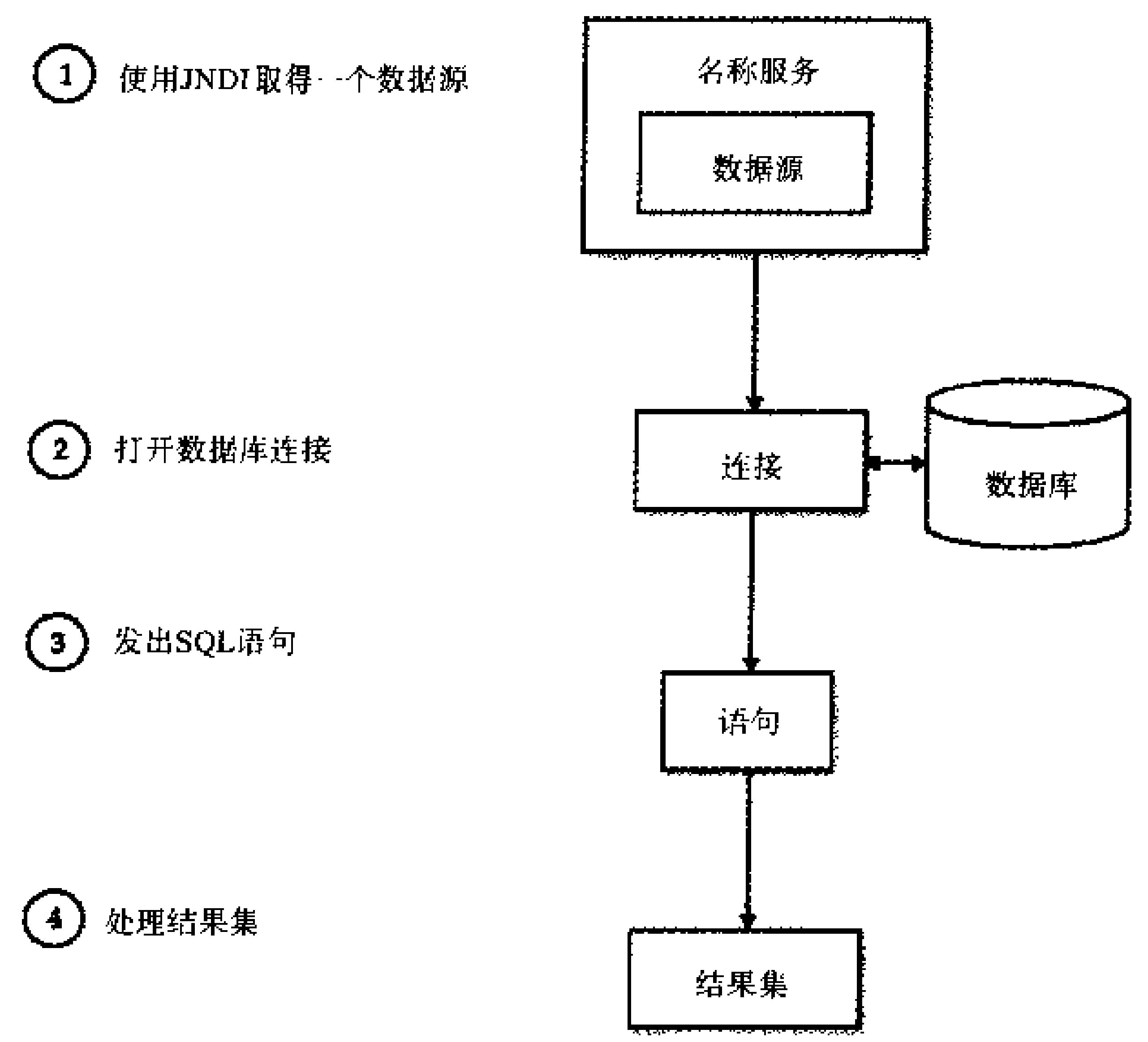


图13-2　使用JDBC 2.0和JNDI的数据库访问

都是其专有数据库协议所特有实现的。java.sql的核心API由16个接口、8个类和4个异常类型组成。可选包 API另外加入了12个接口和2个类。这些类中有许多只是与JDBC驱动器开发商相关的。更常用的是更小一部分子集，如下所述：

- Connection 数据库的当前连接。通过它Java程序可以读取和写入数据，以及开发数据库结构和功能。Connection对象通过JDBC 2.0中的DriverManager.getConnection（ ）或DataSource.getConnection（ ）调用创建。
- Statement 允许通过一个连接发送SQL语句并检索结果集和修改其产生的各行数据的对象。存在3类语句，每一个都是其祖先的特例。

1) Statement 用于执行静态SQL字符串。使用Connection.createStatement（ ）创建一个Statement。

2) PreparedStatement 使用预编译SQL的Statement的扩展，可能会动态设置输入参数。PreparedStatement对象常用于一个SQL插入操作的循环。使用Connection. prepareStatement (sqlstring)创建一个PreparedStatement。

3) CallableStatement 调用一存储过程的PreparedStatement。并不是所有的数据库管理系统都支持存储过程，但对支持的，CallableStatement提供标准调用语法。

- ResultSet 一个SQL查询或对某元数据函数的调用产生的表格行的排序集合。一个ResultSet更常用做一个Statement.executeQuery(sqlstring)方法调用的返回。JDBC API提供

精彩好书见www.book118.com

next（）方法循环ResultSet的行，getXXX（）方法抽取列值。这里XXX是Java代码类型。JDBC 2.0加入了许多用于随意访问和修改行的方法。

- DatabaseMetaData　包含提供数据库结构和功能信息的大量方法接口。通过Connection对象的getMetaData（）方法返回DatabaseMetaData对象。
- ResultSetMetaData　描述ResultSet列的接口，可通过调用结果集的getMetaData()方法获得。它包含描述列数以及每一列的名字显示尺寸、数据类型和类名的方法。
- DriverManager　注册JDBC驱动器并提供处理特殊JDBC URL连接的接口。惟一常用的方法是静态DriverManager.getConnection（），其3种形式的一种是返回捆绑到指定JDBC URL的一个当前Connection。
- SQLException　JDBC API使用的基本溢出类。SQLException带有提供任意厂家指定的错误代码的SQLState值的方法。如果产生多个溢出，它也可以被链接到另一SQLException。

JDBC API的主要特点之一是它应简单且容易掌握。在几天内就可以学会这7个类和3或4种其主要方法。这将使JDBC成为一种通用的技术。

13.1.3　一个简单JDBC实例

下面考虑一个用于JSP页面的JDBC实例。假想的LyricNote公司拥有一个内部雇员数据库，包含两个表格：departments和employees。使用如下SQL创建这些表格：

```
CREATE TABLE departments (
   deptno      char(2),
   deptname    char(40),
   deptmgr     char(4)
)
```

和

```
CREATE TABLE employees (
   deptno     char(2),
   empno      char(4),
   lname      char(20),
   fname      char(20),
   hiredate   date,
   ismgr      bit,
   deptno     char(2),
   title      char(50),
   email      char(32),
   phone     char(4)
)
```

示例JSP页面显示部门列表标识经理名字、电话号码和电子邮件地址。实现此列表的SQL如下：

```
SELECT D.deptname, E.fname, E.lname, E.title, E.email, E.phone
FROM departments D, employees E
WHERE D.deptmgr = E.empno
```

精彩好书见www.book118.com


```
ORDER BY D.deptname
```

D和E前缀是用于验证列名的伪表名，这样DBMS就可以区分哪个表提供哪一列。

完整JSP源码为：

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>

<HTML>
<HEAD>
<TITLE>Department Managers</TITLE>
</HEAD>
<BODY>
<img src="images/lyric_note.png" border=0><p>
<hr color="#000000">
<H2>Department Managers</H2>
<%
   String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
   String URL =
      "jdbc:idb:d:/lyricnote/WEB-INF/database/internal/db.prp";

   // Open a database connection
   Class.forName(DRIVER);
   Connection con = null;
   try {

      con = DriverManager.getConnection(URL);

      // Get department manager information

      String sql = ""
         + " SELECT D.deptname, E.fname, E.lname,"
         + " E.title, E.email, E.phone"
         + " FROM departments D, employees E"
         + " WHERE D.deptmgr = E.empno"
         + " ORDER BY D.deptname"
         ;

      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(sql);
%>
<DL>
<%
      while (rs.next()) {
         String dept = rs.getString(1);
         String fname = rs.getString(2);
         String lname = rs.getString(3);
         String title = rs.getString(4);
```

精彩好书见www.book118.com


```
            String email = rs.getString(5);
            String phone = rs.getString(6);

%>
<DT><B><%= dept %></B></DT>
<DD>
   <%= fname %> <%= lname %>, <%= title %><BR>
   (919) 555-0822 x<%= phone %>, <%= email %>
</DD>
<%
         }
         rs.close();
         rs = null;

         stmt.close();
         stmt = null;
      }
      finally {
         if (con != null) {
            con.close();
         }
      }
%>
</DL>
</BODY>
</HTML>
```

下面讨论每一节

开始是3个page伪指令：

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
```

这里显式要求不创建HTTP会话。应该在所有不需要访问会话的JSP页面中声明这一点。因为它可以节省需要建立和管理会话的服务器资源。

创建页面头标的HTML之后，一个scriptlet查询LyricNote内部数据库并显示结果。开始是两个定义JDBC驱动器名和数据库URL的字符串常量的声明。方便起见，将此信息独自保存在一个声明节中以便修改：

```
String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
String URL =
   "jdbc:idb:d:/lyricnote/WEB-INF/database/internal/db.prp";
```

此例使用InstantDB[1]数据库并连接到LyricNote内部数据库，其Properties文件为db.prp。

1 InstantDB是公开的，全部是由Java编写的关系型DBMS，可从http://instantdb.enhydra.org/index.html处得到，InstantDB有许多高级特性并支持JDBC 2.0。

精彩好书见www.book118.com

真正的工作开始于下一语句：

```
Class.forName(DRIVER);
Connection con = null;
try {
   con = DriverManager.getConnection(URL);
```

Class.forName（）调用使得JDBC驱动器类被载入。依据JDBC规范，驱动器应包含一个使得实例被创建和使用驱动器管理加以注册的静态初始化段。一些旧的驱动器不能实现此功能，如果这样，就需要调用驱动器类的newInstance（）方法。DriverManager类提供实际连接以响应对其静态getConnection（）方法的调用。

注意，用于保存连接的引用的con变量被声明且赋以一个null值。然后页面的其余部分被围在一个try{...}块中，如下：

```
finally {
      if (con != null) {
         con.close();
      }
   }
```

这样做的原因是一旦打开，连接需要关闭，哪怕是发生任何错误或产生溢出。通过finally{...}块使这一点得以确保。包含一个catch块是不需要的。这里缺省溢出处理器就足够了。

一旦建立连接，就可以运行SQL查询。为此，调用Connection对象的createStatement（）方法获得一个Statement对象，在其上可以调用executeQuery（）方法。

```
String sql = ""
   + " SELECT D.deptname, E.fname, E.lname,"
   + " E.title, E.email, E.phone"
   + " FROM departments D, employees E"
   + " WHERE D.deptmgr = E.empno"
   + " ORDER BY D.deptname"
   ;

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

executeQuery（）返回一个ResultSet。列表通过在一循环中调用其next（）方法简单读取该集合的每一行。

```
while (rs.next()) {
      String dept = rs.getString(1);
      String fname = rs.getString(2);
      String lname = rs.getString(3);
      String title = rs.getString(4);
      String email = rs.getString(5);
      String phone = rs.getString(6);
```

精彩好书见www.book118.com


```
%>
<DT><B><%= dept %></B></DT>
<DD>
   <%= fname %> <%= lname %>, <%= title %><BR>
   (919) 555-0822 x<%= phone %>, <%= email %>
</DD>
<%
      }
```

在循环中，使用ResultSet.getString(columnNumber）方法抽取每一列取值，然后格式化并打印单位名称、经理名字、头衔、电话号码和电子邮件。

最后，关闭所有创建的JDBC对象并设其引用为null，这样它们就成为搜集的垃圾。

```
      rs.close();
      rs = null;

      stmt.close();
      stmt = null;
```

Connection对象在前面讨论的finally{...}块中被关闭。

最终输出如图13-3所示。

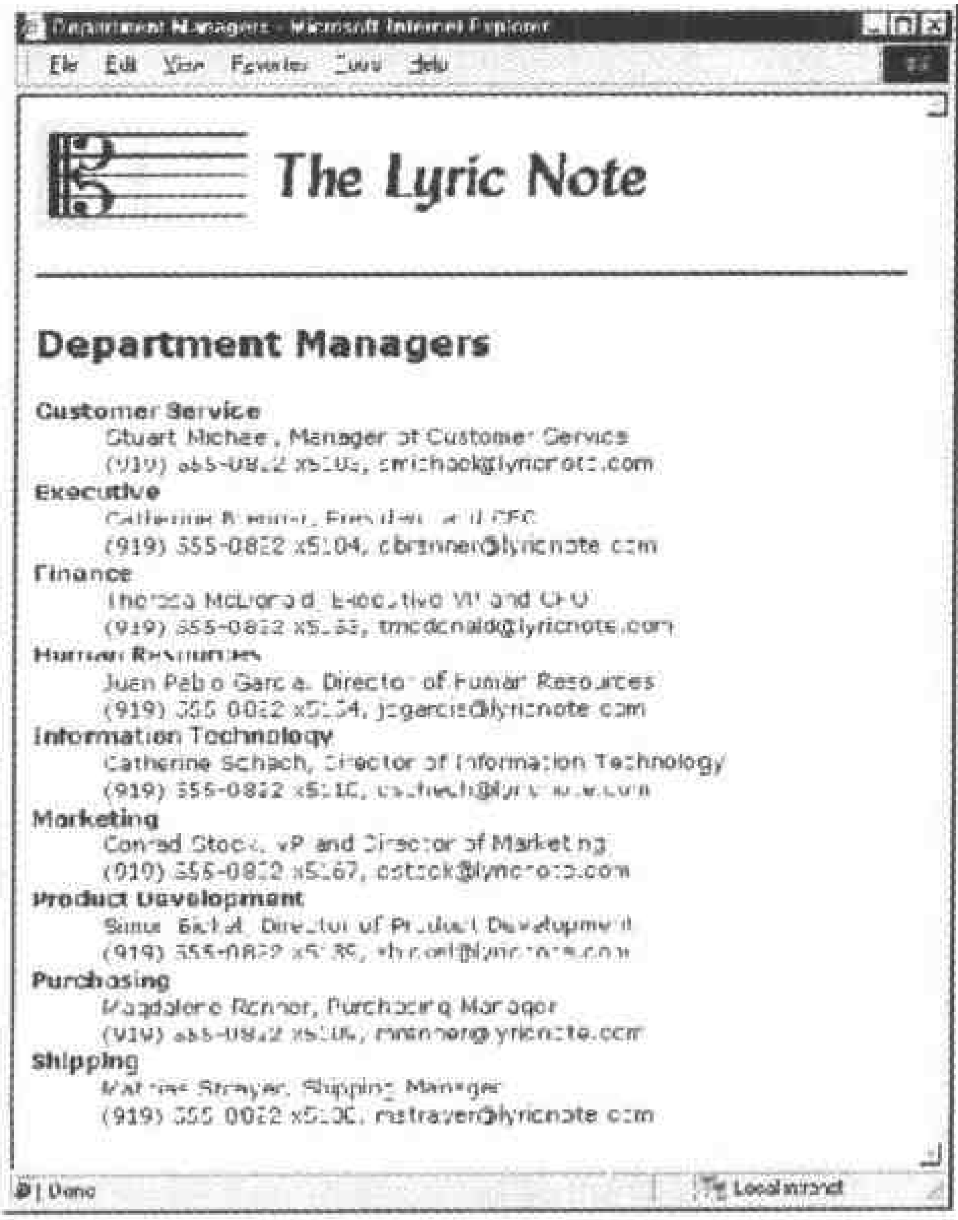


图13-3 简单JDBC例子的输出

精彩好书见www.book118.com

13.2 JDBC驱动器

为了从专有数据库协议角度说明程序，JDBC使用了由DriverManager类和一个或多个JDBC驱动器组成的中间层。一个driver是一个Java类，通常由实现java.sql.Driver接口的数据库厂家提供。驱动器的基本功能是连接到一个数据库并返回java.sql.Connection对象。

应用程序不直接调用驱动器，而是使用DriverManager注册，DriverManager判定一个特定连接请求的相应驱动器并通过它进行连接。

存在成百个驱动器，覆盖了所有数据库管理系统。其中大多数可从厂家的Web站点下载。在http://industry.java.sun.com/products/jdbc/drivers上可以找到一个搜索列表。

下面一节讨论4个JDBC驱动器类型，JDBC-ODBC桥的特殊情况以及注册一个驱动器的机制。

13.2.1 驱动器类型

JDBC规范依据驱动器结构将驱动器划分为4种类型，分别是：

- 类型1——JDBC-ODBC桥　此类型驱动器通过一个中间ODBC驱动器连接到数据库。这种方法有几个缺点，因此Sun只将之作为在没有其他驱动器可利用时的实验性和适当的选择。Microsoft和Sun都提供类型1 驱动器。
- 类型2——本地API，部分Java　类似于一个JDBC-ODBC桥，类型2驱动器使用本地方法调用厂家指定的API函数。这些驱动器也面对着与JDBC-ODBC桥同样的限制，因为它们需要在客户端系统上安装本地库文件。客户端为使用它们必需加以配置。
- 类型3——纯Java到数据库中间件　类型3驱动器使用至中间件服务器的网络协议进行通信。此中间件服务器反过来与一个或多个数据库管理系统通信。
- 类型4——纯Java直接到数据库　此类驱动器直接调用数据库管理系统使用的本地协议。

4种驱动器类型的体系结构都显示在图13-4中。

驱动器类型的区别是什么？从应用程序的观点看，分类不止是从系统结构方面。类型1和类型2驱动器需要本地代码被安装和在配置在客户端系统上。如果DBMS在一防火墙后，类型4驱动器就不是很适合。同样，每种驱动器都有其自己的性能特征。但应用程序接口在4种类型中是完全一样的。

13.2.2 JDBC-ODBC桥

类型1中JDBC-ODBC桥驱动器需要特别考虑。正如所见，使用它要考虑几个问题。首先JDBC-ODBC桥驱动器受限于底层的ODBC驱动器功能。ODBC驱动器是单线程的，因此在负载重情况下性能可能会很低。还有，它需要本地代码库JdbcOdbc.dll安装在客户端系统上。最后，任何使用情况它都需要配置一个ODBC数据源。这些限制使其不适合用于外部internet的applet。Sun推荐当没有其他JDBC驱动器可利用时出于实验性目的而使用它。

另一方面，JDBC-ODBC桥有几个明显的优点。因为JSP页面不是在applet环境中操作，它们没有这些限制。ODBC被广泛支持，因此使用桥使得访问数据源已经配置于其上的各种已存在系

精彩好书见www.book118.com

统成为可能。同样，使ODBC的数据库产品，如Microsoft Access和FoxBase，能被广泛利用。这些特性使JDBC-ODBC桥成为对低版本的Web应用和学习JDBC的可用工具的好选择。

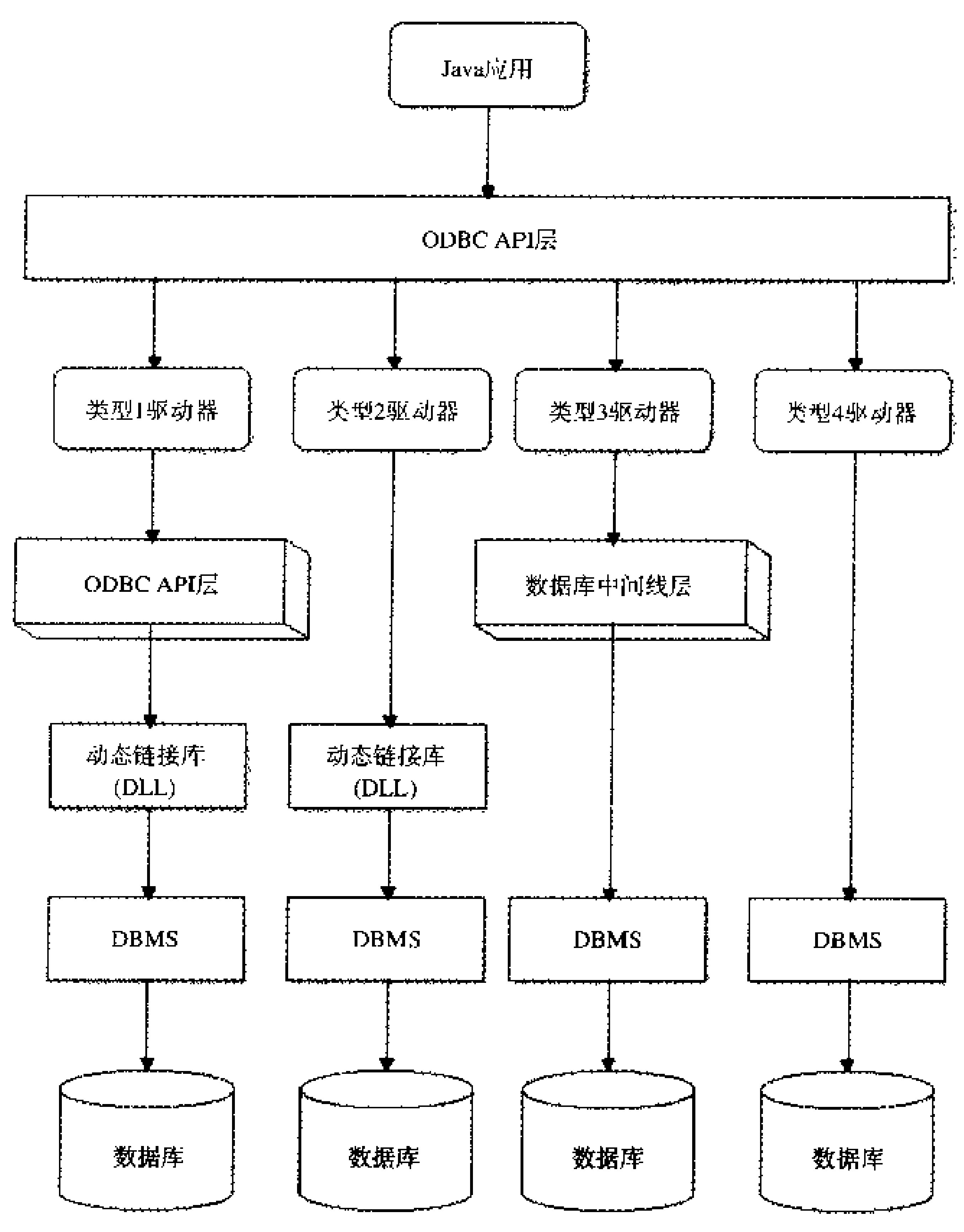


图13-4　4种JDBC驱动器类型体系结构

为在Java应用中使用JDBC-ODBC桥，必需配置一个适当的ODBC数据源。在Windows系统上，可通过控制面板上的ODBC数据源应用实现、数据源应配置为系统DSN，而不是用户DSN，因为JSP引擎典型地运行在系统用户配置文件下。如果使用Sun JVM，驱动器类名为sun.jdbc.odbc.JdbcOdbcDriver，对于Microsoft虚拟机，则为com.ms.jdbc.odbc.JdbcOdbcDriver。如果数据源名为dsn，则在getConnection（ ）语句中使用的数据库URL是jdbc:odbc:dsn。

Microsoft为其Access数据库产品以及dBase、Excel、FoxPro和大量的其他类型数据库提供

精彩好书见www.book118.com

ODBC驱动器，包括将一般文本文件作为简单数据库系统的文本驱动器。

13.2.3 注册一个驱动器

对于使用的一个JDBC驱动器，必需首先由驱动器管理者对其注册。实现这一点有几种方式，每一种都包含对DriverManager.registerDriver（ ）的调用。

最常用的方法是简单地载入驱动器类：

```
try {
    Class.forName("MyJdbcDriver");
}
catch (ClassNotFoundException e) {
    // Report the exception
}
```

以此方式载入的驱动器类应创建一个驱动器类实例并使用驱动器管理者注册它，使用逻辑类似下面代码：

```
static {
    PrintStream log = DriverManager.getLogStream();
    if (log != null)
        log.println("MyJdbcDriver class loaded");
    MyJdbcDriver driver = new MyJdbcDriver();
    try {
        DriverManager.registerDriver(driver);
    }
    catch (SQLException e) {
        if (log != null)
            log.println("Unable to register driver");
    }
}
```

一些旧的驱动器省略了此步骤，而是在构造器中进行注册。如果是那样，则需要使用下列方法创建驱动器的一个实例[1]：

```
try {
    Class.forName("MyJdbcDriver");
}
catch (ClassNotFoundException e) {
    // Report the exception
}
catch (InstantiationException e) {
    // Report the exception
}
catch (IllegalAccessException e) {
```

1 这样做很枯燥。可以简单地捕获Exception本身。但这样总是会使你可能对没有预计到的溢出应用错误的逻辑。

精彩好书见www.book118.com


```
    // Report the exception
}
```

驱动器注册的另外方法是将驱动器名放入jdbc.drivers系统属性中。它是驱动器类名的一个逗号分隔的列表，在其初始化过程中由DriverManager载入它。例如，如下可调用一个使用此方法的单机Java应用：

```
java -Djdbc.drivers=org.enhydra.instantdb.jdbc.idbDriver MyPGM
```

一些JDBC驱动器厂家，特别是Oracle，推荐显式创建一个驱动器实例并用驱动器管理者注册它：

```
DriverManager.registerDriver(
new oracle.jdbc.driver.OracleDriver());
```

JDBC 2.0允许通过使用一个JNDI服务提供者注册的DataSource对象进行连接。例如JRun 3.0，提供了在Web服务器层上定义JDBC数据源的方式，以及对连接的快速在线测试。这种方法的优点是驱动器类名和数据库URL被存储在名称服务中，而不是在应用程序中硬编码。只需要数据源名即可。与本章前面图13-3相关的示例JSP页面可以用下列代码代替其连接逻辑：

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup
   ("java:comp/env/jdbc/lyricnote_internal");

Connection con = null;
try {
   con = ds.getConnection();
   ...
}
finally {
   if (con != null)
      con.close();
}
```

注意　使用JDBC连接一个数据源的JSP页面必须导入javax.sql.*和javax.naming.*或其他以限定对InitialContext和DataSource的引用。

使用DataSource的另一好处是其他高级数据库特性如连接池和分布式事务处理可以由捆绑到名称服务的改变而整个改变，JSP源码无需改变。

13.3　连接到一个数据库

载入并注册一个驱动器后，它可以用来创建数据库连接。DriverManager实现此功能有如下3种方法：

```
getConnection(String url)
getConnection(String url, String userID, String password)
getConnection(String url, Properties prop)
```

精彩好书见www.book118.com

本质上，DriverManager使用同样的私有工作者方法处理每一方法。

驱动器管理者保留已注册驱动器列表，当调用其getConnection ( ) 方法时，它一次询问每一驱动器是否接受指定的URL。驱动器管理者通过调用驱动器的connect ( ) 方法实现此功能，如果驱动器不能接受URL，该方法返回null；如果可以，则返回一个激活的Connection对象。

前面提到，JDBC 2.0允许使用DataSource代替DriverManager建立连接。这种情况下，不使用URL参数，因为它被存储在名称服务中。

JDBC数据库URL

DriverManager.getConnection()的关键参数是一个JDBC URL，一个具有冒号分隔的3个组件的字符串：

```
<protocol>:<subprotocol>:<subname>
```

这里

- protocol 总是jdbc。
- subprotocol 标识使用的驱动器厂家指定的字符串。驱动器指出当其被驱动器管理者询问时是否能处理此subprotocol。例如，JDBC-ODBC桥使用保留值odbc作为其subprotocol。该值是惟一可用于所有驱动器厂家的值。Sun微系统是JDBC Subprotocol的正式注册者。
- subname 标识连接的特定数据库。该字符串包含驱动器需要用来标识数据库的内容。它也包含实际需要的连接参数。

JDBC URL例子如下：

```
jdbc:odbc:usda
```

表明JDBC-ODBC桥驱动器访问的一个ODBC数据源其名为usda。

```
jdbc:idb:c:/path/database.prp
```

InstantDB将subname解释为描述数据库位置和特性的一个属性文件。

```
"jdbc:oracle:thin:@"
    + "(DESCRIPTION="
    + "(ADDRESS=(HOST=u25nv)"
    + "(PROTOCOL=tcp)"
    + "(PORT=4311))"
    + "(CONNECT_DATA=(SID=music)))"
```

这是一个可能由Oracle客户端驱动器使用的长度连接的字符串。

像驱动器注册时一样,JDBC 2.0使用一个名称服务中的DataSource隐藏JDBC URL细节。

13.4 语句接口

SQL语言由从一个关系数据库中创建、表示和抽取数据的语句组成。JDBC提供这些SQL语句的面向对象的表示以封装其文本、执行状态和结果。理所当然，这种表示称为java.sql.Statement接口。语句对象发送SQL命令到数据库，其类型如下：

精彩好书见www.book118.com

- 数据定义命令，如CREATE TABLE或CREATE INDEX。
- 数据表示命令，如INSERT或UPDATE。
- 执行一个查询的SELECT语句。

数据表示命令返回被修改行的行数，而一个SELECT语句返回称为结果集的行的集合。

Statement接口有两个特殊子接口，扩展了其功能：使用预编译SQL的PreparedStatement，调用存储过程的CallableStatement。下面一节讨论这3种语句及其使用方式。

13.4.1 Statement

基本接口是java.sql.Statement。因为这是一个接口，它没有构造器，而是使用Connection.createStatement（）从连接对象获得。典型例子如下：

```
Connection con = null;
try {
   con = DriverManager.getConnection(URL);
   Statement stmt = con.createStatement();
   ...
   stmt.close();
}
finally {
   if (con != null)
      con.close();
}
```

JDBC 2.0引入了createStatement（）的另外一种形式，带有指出结果集是否可滚动以及是否同时影响到底层表格参数的改变。本章后面结果集一节详细描述了这些特性。

一旦创建语句，它就可用于执行命令。有4种方法可以实现：executeupdate、executeQuery、execute和executeBatch。使用哪种方法取决于要给出的结果：

- executeUpdate 趋向用于SQL INSERT、UPDATE或DELETE语句，或数据定义语句，如CREATE TABLE。它返回已修改行的行数。
- executeQuery 用于执行一个SQL SELECT语句并返回结果集。
- execute 可用于上述两种情况，但更趋向用于返回一个修改数量、多结果集或两者结合的语句。它返回一个指出其结果是否为一个修改数量或结果集的boolean型标记。另外此方法可用于导航结果。
- executeBatch 允许在一个批处理中执行多修改语句。修改数量返回到一个数组中。

下面的例子说明了这些方法。

1) executeUpdate方法

此例中，错误产品描述被一个executeUpdate方法调用的SQL UPDATE语句改正过来：

```
import java.sql.*;

public class UpdateExample
{
```

精彩好书见www.book118.com


```
    public static void main(String[] args)
        throws ClassNotFoundException, SQLException
    {
        String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
        String URL = "jdbc:idb:"
            + "D:/lyricnote/WEB-INF/database/products/db.prp";

        Class.forName(DRIVER);
        Connection con = null;
        try {
            con = DriverManager.getConnection(URL);
            Statement stmt = con.createStatement();
            int nRows = stmt.executeUpdate(
                " UPDATE products"
                + " SET description -"
                + "'Telemann: Concerto No. 1 in F for Two Horns'"
                + " WHERE itemcode = '022370'"
                );
            System.out.println(nRows + " rows updated");
            stmt.close();
        }
        finally {
            if (con != null)
                con.close();
        }
    }
}
```

成功执行时，程序打印“1 rows updated”。

2) executeQuery方法

为查看错误列表被改正，此例使用一个SELECT语句显示所有产品目录中名为Telemann协奏曲的音乐唱片的名字：

```
import java.sql.*;

public class QueryExample
{
    public static void main(String[] args)
        throws ClassNotFoundException, SQLException
    {
        String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
        String URL = "jdbc:idb:"
            + "D:/lyricnote/WEB-INF/database/products/db.prp";

        Class.forName(DRIVER);
            Connection con = null;
```

精彩好书见www.book118.com


```
            try {
                con = DriverManager.getConnection(URL);
                Statement stmt = con.createStatement();
                ResultSet rs = stmt.executeQuery(
                        " SELECT itemcode, description"
                    + " FROM products"
                    + " WHERE prodtype = 'SM'"
                    + " AND description like 'Telemann%'"
                    + " AND description like '%Concerto%'"
                );
                while (rs.next()) {
                    String itemCode = rs.getString(1);
                    String description = rs.getString(2);
                    System.out.println(itemCode + " " + description);
                }
                rs.close();
                stmt.close();
            }
            finally {
                if (con != null)
                    con.close();
            }
        }
    }
```

运行时，生成改正后输出：

```
022340 Telemann: Double Viola Concerto in G
022350 Telemann: Viola Concerto in G
022360 Telemann: Concerto for Horn Quartet
022370 Telemann: Concerto No. 1 in F for Two Horns
```

从结果集中取值的过程在本章后面介绍。

3) execute方法

虽然execute也可用于查询或修改，但严格说来，它用于返回多结果的操作。Statement接口提供判断返回内容和处理结果的方法。execute最常用于处理未知SQL字符串，下面例子中，从一个文件中读取和处理SQL语句：

```
import java.io.*;
import java.sql.*;

public class ExecuteExample
{
    public static void main(String[] args)
        throws ClassNotFoundException, SQLException, IOException
    {
        String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
```

精彩好书见www.book118.com


```
String URL = "jdbc:idb:"
   + "D:/lyricnote/WEB-INF/database/products/db.prp";
Class.forName(DRIVER);
Connection con = null;
try {
   con = DriverManager.getConnection(URL);
   Statement stmt = con.createStatement();

   // Read SQL statements from a file

   BufferedReader in =
      new BufferedReader(
      new FileReader("executeExample.sql"));

   while (true) {
      String line = in.readLine();
      if (line == null)
         break;

      // Execute statement

      boolean hasResultSet = stmt.execute(line);
      while (true) {

         if (hasResultSet) {
            ResultSet rs = stmt.getResultSet();
            System.out.println("Processing result set");
            // ... process result set
            rs.close();
         }
         else {
            int count = stmt.getUpdateCount();
            if (count == -1)
               break;
            System.out.println("Processing update count");
            // ... process update count
         }

         // See if there are any more results

         hasResultSet = stmt.getMoreResults();
      }
   }
   stmt.close();
      in.close();
   }
```

精彩好书见www.book118.com


```
            finally {
                if (con != null)
                    con.close();
            }
        }
    }
```

execute的初始返回代码是一个布尔值，如果语句执行生成了一个结果集，则为true，否则使用Statement.getUpdateCount（）获得修改计数。如果修改计数为-1，那么存在更多的结果集。同样，Statement.getMoreResults（）方法可被调用来对下一结果集或修改计数进行循环。其返回的布尔值含义与execute返回的一样。

4) ExecuteBatch方法

JDBC 2.0引入一组修改语句以批处理形式执行的功能。某些情况下，会显著提高性能。使用批处理修改在连接中使用方法为：

• clearBatch　重置批处理为空。
• addBatch　向批处理加入一个修改语句。
• executeBatch　确认批处理，并搜集修改计数。

并不是所有的驱动器都支持批处理修改，那些支持者通过从其DatabaseMetaData.supportsBatchUpdates（）方法中返回true表明这一点。

实现此功能的一个驱动器是Microsoft Access的JDBC-ODBC桥。下面例子中，LyricNote的composers Access数据库被修改，使表composers为至少为90岁的人组成。

```
import java.io.*;
import java.sql.*;
import java.util.*;

public class BatchUpdateExample
{
   public static void main(String[] args)
      throws ClassNotFoundException, SQLException, IOException
   {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = null;
      try {

         // Connect to the composers database

         con = DriverManager.getConnection
            ("jdbc:odbc:composers");
         Statement stmt = con.createStatement();

         // Clear the existing table and create a new one

         stmt.executeUpdate("DROP TABLE over90");
```

精彩好书见www.book118.com


```
stmt.executeUpdate(
        " CREATE TABLE over90"
      + " ("
      + " lastName VARCHAR(20),"
      + " firstName VARCHAR(20),"
      + " age INTEGER"
      + " )"
);

// Set up for handling all-or-nothing transaction

con.setAutoCommit(false);

// Add insert statements to a batch

stmt.clearBatch();

stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Rodrigo','Joaquin',99)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Gossec','Francois-Joseph',96)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Ruggles','Carl',96)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Widor','Charles-Marie',94)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Sibelius','Jean',93)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Copland','Aaron',91)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Auber','Daniel Francois',90)");
stmt.addBatch("INSERT INTO over90 VALUES"
    + "('Stravinsky','Igor',90)");

// Execute the batch and check the update counts

int[] counts = stmt.executeBatch();

boolean allGood = true;
for (int i = 0; i < counts.length; i++)
    if (counts[i] != 1)
        allGood = false;

// Commit or roll back the transaction

if (allGood) {
```

精彩好书见www.book118.com


```
                System.out.println
                    ("Transaction successful with "
                    + counts.length + " statements committed");
                con.commit();
            }
            else {
                System.out.println("Transaction failed");
                con.rollback();
            }

            // Done

            stmt.close();
        }
        finally {
            if (con != null)
                con.close();
        }
    }
}
```

设置连接的autoCommit标记使得我们可以确认或回滚整个批处理修改。

13.4.2 PreparedStatement

java.sql.PreparedStatement是使用预编译SQL的Statement的子接口。如果重复使用该语句，会导致性能的提高。PreparedStatement与Statement不同，其execute方法并不接受SQL字符串作为参数，而是在创建PreparedStatement时指定该SQL字符串。如下所示：

```
PreparedStatement pstmt = con.prepareStatement(sqlstring);
```

要执行的字符串可以包含两个可替代参数，在字符串中通过问号（？）的出现来表示。这些参数用作语句中的占位符，在执行前必须赋值。因此，API提供setXXX方法，这里XXX为Java的数据类型。

上述批处理修改实例，创建和载入年龄在90岁以上人的名单。其代码包含在循环体中执行的Preparedstatment，如下所示：

```
import java.io.*;
import java.sql.*;
import java.util.*;

public class PreparedStatementExample
{
   public static void main(String[] args)
      throws ClassNotFoundException, SQLException, IOException
   {
      String DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
```

精彩好书见www.book118.com


```
String URL = "jdbc:odbc:composers";

Connection con = null;
try {

   // Load the driver class

   Class.forName(DRIVER);

   // Connect to the database

   con = DriverManager.getConnection(URL);

   // Create the new table
   Statement stmt = con.createStatement();
   try {
   stmt.executeUpdate("DROP TABLE OVER90");
   }
   catch (SQLException ignore){}
   stmt.executeUpdate(
         " CREATE TABLE over90"
      + " ("
      + " lastName VARCHAR(20),"
      + " firstName VARCHAR(20),"
      + " age INTEGER"
      + " )"
   );
   stmt.close();
   stmt = null;

   // Prepare a statement to do inserts into the table

   PreparedStatement pstmt = con.prepareStatement(
      "INSERT INTO over90 VALUES(?, ?, ?)"
   );

   // Read composer names and ages from a file
   // that uses tabs to separate the fields

   BufferedReader in =
      new BufferedReader(
      new FileReader("over90.txt"));

   while (true) {
      String line = in.readLine();
      if (line == null)
```

精彩好书见www.book118.com


```
                break;

            // Split the line into the last name, first name
            // and age tokens

            StringTokenizer st = new StringTokenizer(line, "\t");
            if (st.countTokens() != 3)
                throw new IOException ("Expected 3 fields");
                String lastName = st.nextToken();
                String firstName = st.nextToken();
                int age = Integer.parseInt(st.nextToken());

                // Set the parameters in the prepared statement

                pstmt.setString(1, lastName);
                pstmt.setString(2, firstName);
                pstmt.setInt(3, age);

                // Update the record

                pstmt.executeUpdate();
                System.out.println(
                "Added record for " + firstName + " " + lastName);
            }

            in.close();

            pstmt.close();
            pstmt = null;

        }
        finally {
            if (con != null)
                con.close();
        }
    }
}
```

考虑代码中几个关键点。首先，语句创建时要带有可替换参数：

```
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO over90 VALUES(?, ?, ?)"
);
```

这里有3个问题要说明：一个是对表格中每一列，使用数字和字符串参数间没有差别，其余两个是简单编码，不需要有嵌入式引号或省略符号。

为在INSERT语句中使用从文件中读取的值，采纳setString（）和setInt（）方法：

```
pstmt.setString(1, lastName);
```

精彩好书见www.book118.com


```
pstmt.setString(2, firstName);
pstmt.setInt(3, age);
```

setXXX（）方法的第一个参数是列号，第一列为1，第二列为2，等等。第2个参数是插入值。

setXXX（）方法存在于所有的数据类型，以及两个特殊类型：setObject（）和setNull（）。使用setObject（）将类型转换为任意JDBC数据类型。它带有第3个参数：

```
pstmt.setObject(int column, Object value, int typeNumber)
```

这里typeNumber是定义在java.sql.Types中的静态整型常量。类似地，setNull（）可用来在参数中保存适当的null类型。

```
pstmt.setNull(int column, int typeNumber)
```

使用prepared语句避免动态语法错误

使用prepared的基本动机是其性能，另外还有一点益处。假定要制作一个运行LyricNote产品数据库查询的JSP页面。该页面包括一个输入搜索参数的窗体，此参数从请求参数中抽出，然后动态构建SQL语句。以下显示如何构建SQL的部分JSP页面：

```
ResultSet rs = stmt.executeQuery(
  " SELECT itemcode, description"
  + " FROM products"
  + " WHERE prodtype = 'SM'"
  + " AND description like '%" + searchFor + "%'"
  );
```

当使用该JSP页面搜索Stravinsky的工作时，返回结果如图13-5所示。

图13-5 QueryExample2.jsp的正常输出

精彩好书见www.book118.com

然而，如果要指定搜索Stravinsky的L'Histoire du Soldat，则得到图13-6中显示的错误屏幕。

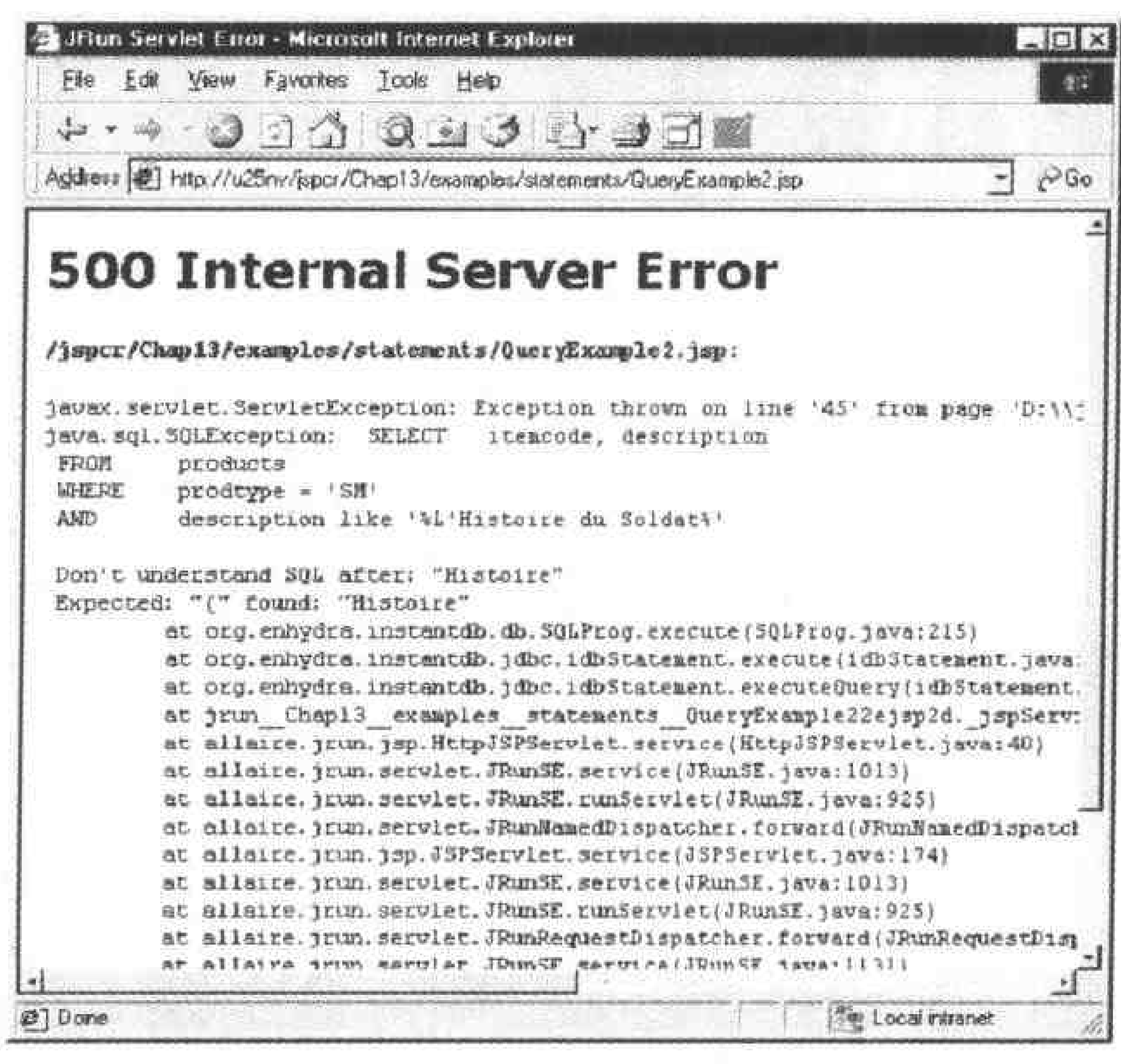


图13-6　由一非转义单引号引起的语法错误

发生什么了？答案可在错误信息中找到：

```
javax.servlet.ServletException
java.sqlSQLException: SELECT itemcode, description
 FROM products
 WHERE prodtype = 'SM'
 AND description like '%L'Histoire du Soldat%'

Don't understand SQL after: "Histoire"
```

单词L'Histoire有嵌入的单引号，因此当执行LIKE语句时，马上中断，将‘%L’视作其试图匹配的操作数。它下面的内容将被解析好象是它是SQL，因此引发错误。

问题可以通过扫描用户输入，查找嵌入的单引号，用其安全的替代字符替换。但实际上这比想象的复杂。此技术称为掩码字符，将随数据库和其SQL语法的不同而不断变化。JDBC体系的存在支持掩码字符方式，但也增加了用户必须处理内容的复杂性。

一个更简单和清晰的方式是使用带有一个替代参数的PreparedStatement处理它。需要改变的代码是：

精彩好书见www.book118.com


```
PreparedStatement pstmt = con.prepareStatement(
      " SELECT itemcode, description\n"
    + " FROM products\n"
    + " WHERE prodtype = 'SM'\n"
    + " AND description like ?"
);
pstmt.setString(1, "%" + searchFor + "%");
ResultSet rs = pstmt.executeQuery();
```

LIKE子句的操作数现在只是一个简单的问题标记，在运行时搜索参数动态加入。查询同时在任何输入类型下均可工作，而无论其在SQL中的含义。如图13-7所示。

13.4.3 CallableStatement

PreparedStatement的精致形式被收录在java.sql.CallableStatement。如果数据库支持它们[1]，此接口用于调用存储过程。例如，Oracle允许在PL/SQL中编写过程。在Microsoft Access中编写的查询可通过JDBC-ODBC桥作为存储过程被调用。

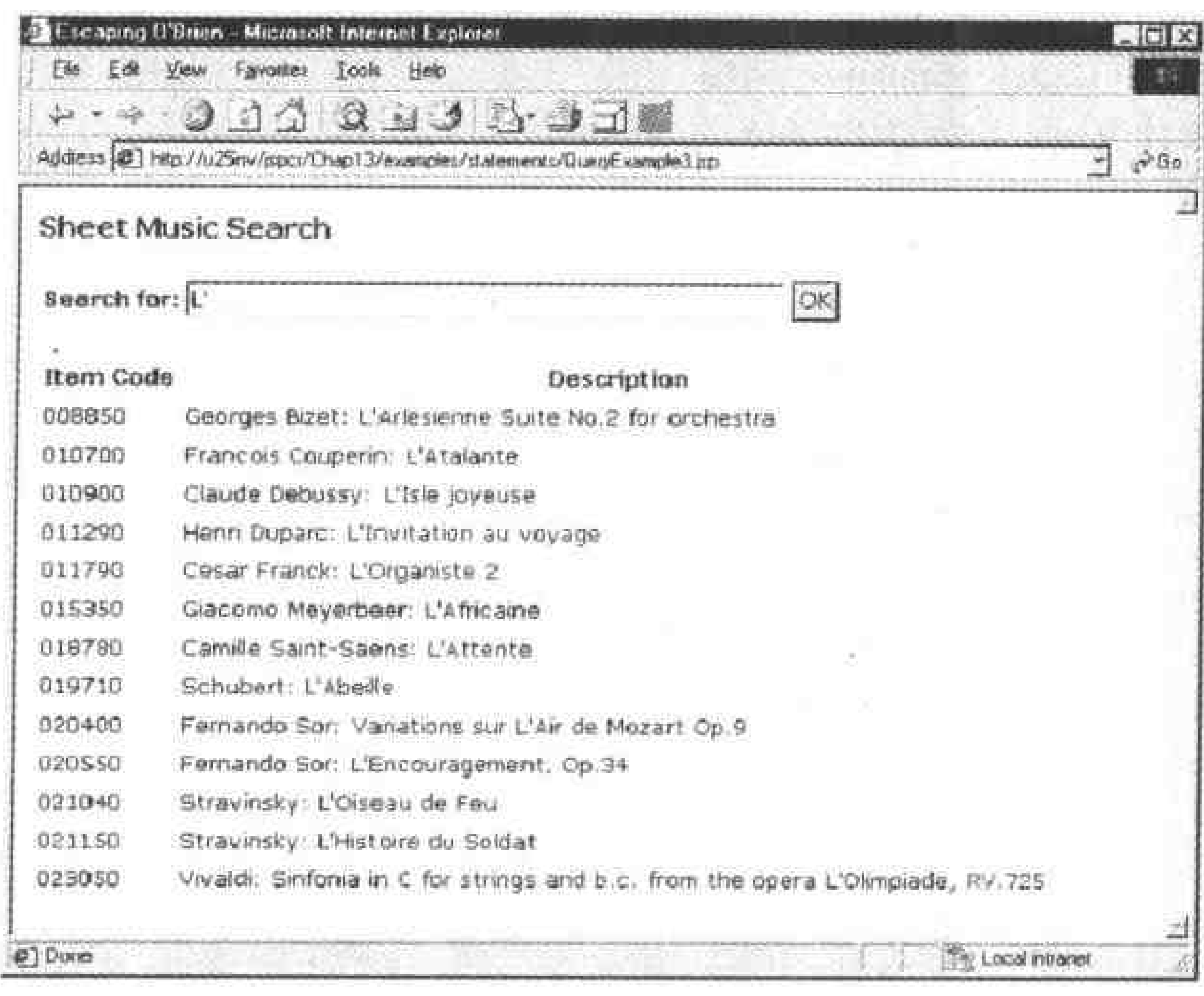


图13-7 将QueryExample2.jsp改变为可使用PreparedStatement后的输出

与其父接口PreparedStatement一样，使用一个预编译的显式命令字符串创建一个CallableStatement：

1 实际上很少有非商业性数据库支持存储过程。

精彩好书见www.book118.com


```
CallableStatement cstmt = con.prepareCall(escapeString);
```

它也使用问题标记指出可替代参数。CallableStatement使用的存储过程调用语法如下：

```
{? = call procedureName(?, ?, ..., ?)}
```

如果过程没有返回值，硬省略“? =”。类似地，如果没有输入参数，则不使用“(?, ?, …, ?)”。

因为CallableStatement扩展了PreparedStatement，它使用同样的方法设置可替代参数值：

```
String sql = "{call myproc(?, ?)}";
CallableStatement cstmt = con.prepareCall(sql);
cstmt.setString(1, "New York");
cstmt.setDouble(2, "19.73");
cstmt.executeQuery();
```

如果参数为OUT或INOUT，在调用被执行前必须使用CallableStatement.registerOutParameter( )注册此类型。其值可同样用PreparedStatement使用的getXXX( )方法检索。

Microsoft Access中的存储过程

Microsoft Access支持在SQL中编写的或按其设计意向开发的查询。这些查询可通过名字使用JDBC-ODBC桥和CallableStatement调用。图13-8显示了创建出生在指定年份之间的composer列表的查询的设计视图。开始和结束年份是查询的输入参数。

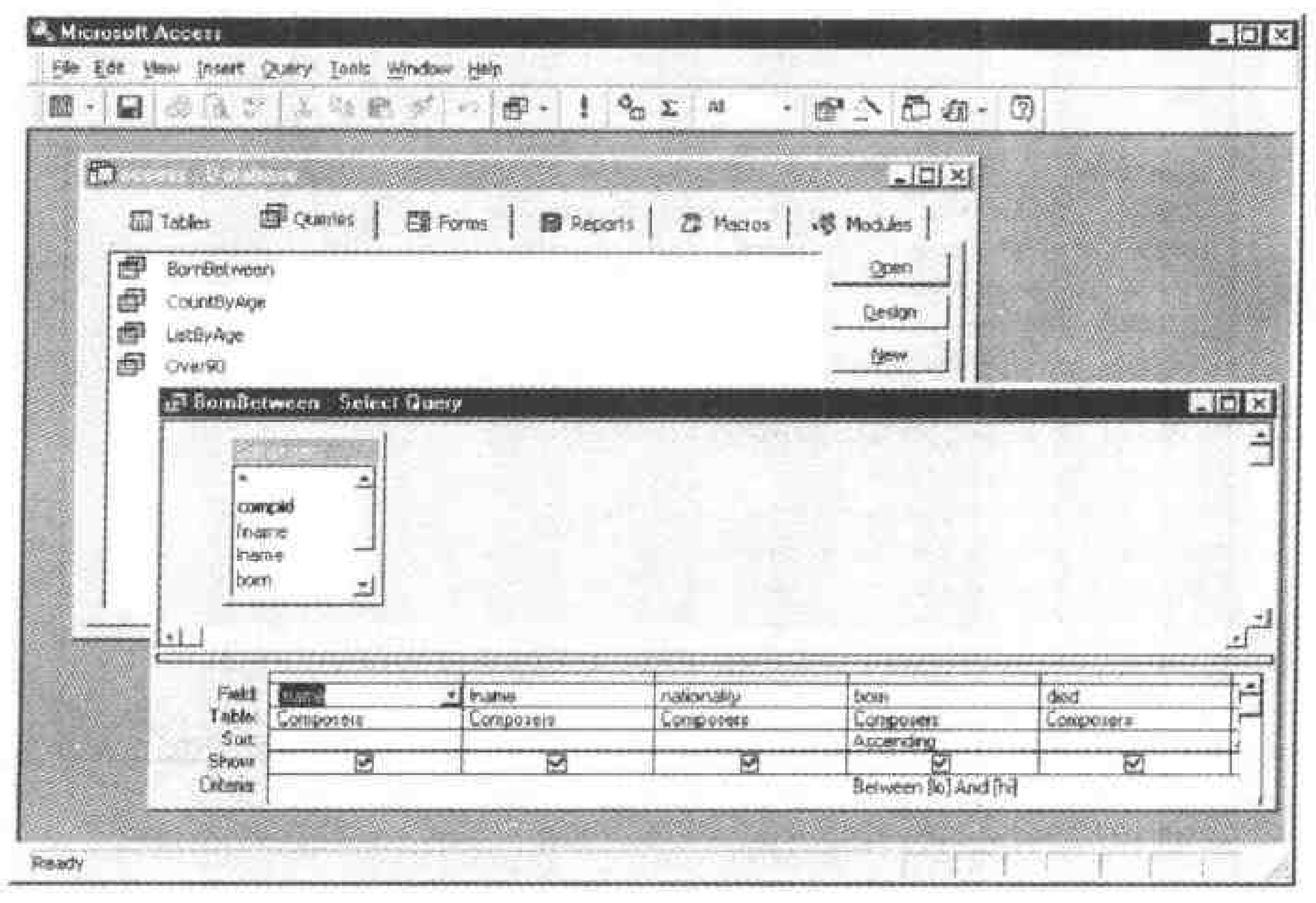


图13-8　BornBetween 查询的设计视图

当使用1891-1900作为年份区间运行时，选中了12个记录。结果如图13-9所示。

此查询可从JSP页面使用的CallableStatement运行，JSP页面执行的步骤如下所示：

精彩好书见www.book118.com

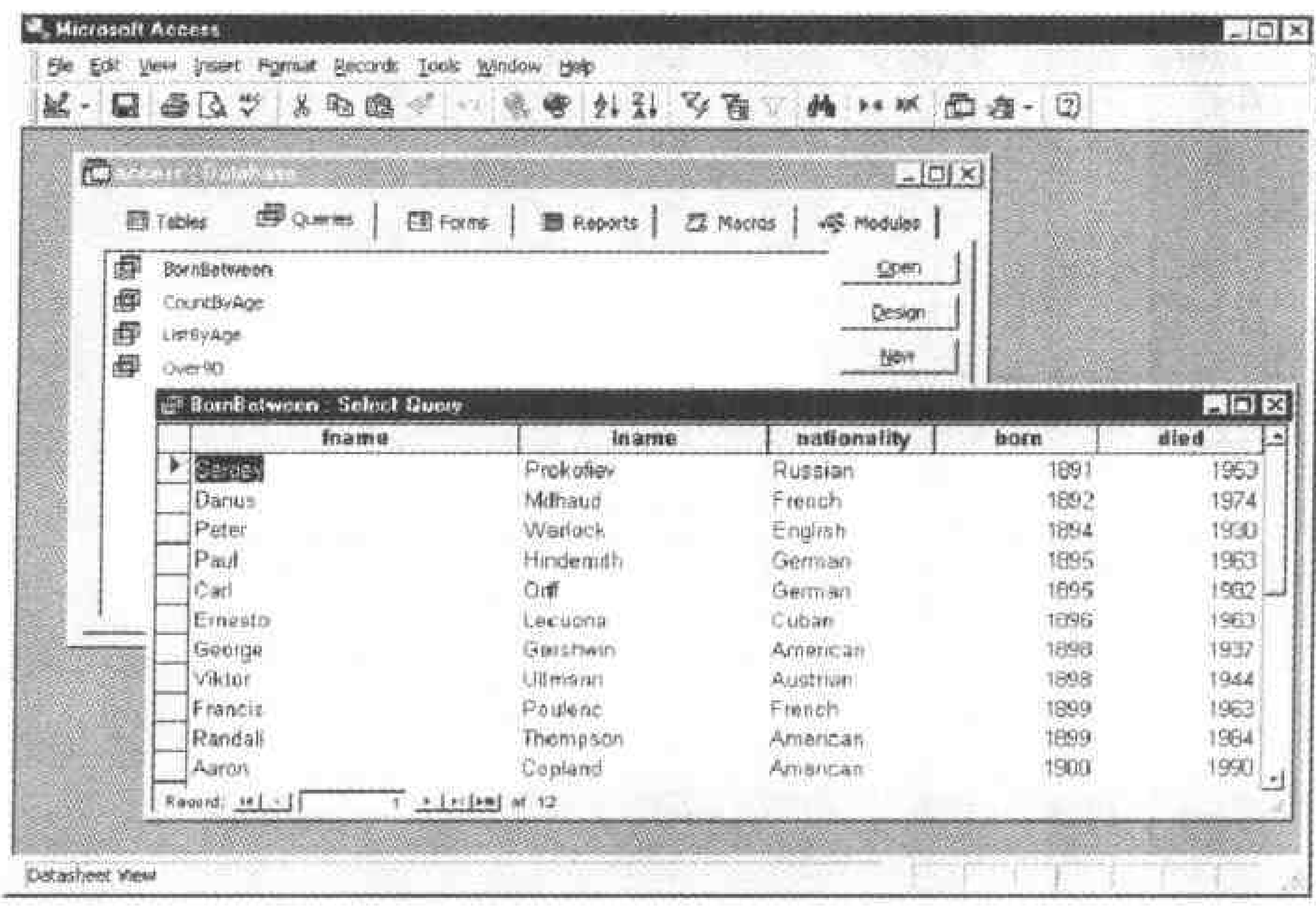


图13-9 BornBetween查询对1891-1900的结果

1) 提示在一HTML窗体中输入开始和结束年份。

2) 通过JDBC-ODBC桥连接到Access数据库。

3) 创建调用查询的CallableStatement。

4) 设置来自窗体取值的开始和结束年份参数。

5) 执行查询。

6) 在一个HTML表格中显示结果。

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%
   // Prompt for beginning and ending years

   String sLo = request.getParameter("lo");
   if (sLo == null)
      sLo = "";
   String sHi = request.getParameter("hi");
   if (sHi == null)
      sHi = "";
%>
   <H3>Select Composers by Year Born</H3>
   <FORM>
   <TABLE>
   <TR>
      <TD>Year range:
      <INPUT TYPE="TEXT" NAME="lo" SIZE=4 VALUE="<%= sLo %>">
      and
```

精彩好书见www.book118.com


```
            <INPUT TYPE="TEXT" NAME="hi" SIZE=4 VALUE="<%= sHi %>">
            <INPUT TYPE="SUBMIT" VALUE="Search">
         </TD>
      </TR>
      </TABLE>
      </FORM>
<%
      if (!sLo.equals("") && (!sHi.equals(""))) {

         int lo = Integer.parseInt(sLo);
         int hi = Integer.parseInt(sHi);

         // Load the driver

         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection con = null;
         try {

            // Connect to the composers database

            con = DriverManager.getConnection
               ("jdbc:odbc:lyricnote_internal");

            // Set up callable procedure

            String sql = "{call BornBetween(?, ?)}";
            CallableStatement cstmt = con.prepareCall(sql);
            cstmt.setInt(1, lo);
            cstmt.setInt(2, hi);
            ResultSet rs = cstmt.executeQuery();
%>
      <P>
      <TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
      <TR>
         <TH>Name</TH>
         <TH>Nationality</TH>
         <TH>Lived</TH>
      </TR>
<%
            // Print the result set

            while (rs.next()) {
               String fname = rs.getString(1);
               String lname = rs.getString(2);
               String nationality = rs.getString(3);
               int yearBorn = rs.getInt(4);
               int yearDied = rs.getInt(5);
%>
      <TR>
         <TD><%= fname %> <%= lname %></TD>
         <TD><%= nationality %></TD>
         <TD><%= yearBorn %>-<%= yearDied %></TD>
      </TR>
<%
```

精彩好书见www.book118.com


```
            }
%>
    </TABLE>
<%
            rs.close();
            rs = null;
            cstmt.close();
            cstmt = null;
        }
        finally {
            if (con != null) {
                con.close();
                con = null;
            }
        }
    }
%>
```

结果如图13-10所示。

图13-10 BornBetween查询对1891-1900年的操作基于Web的版本

精彩好书见www.book118.com

当然，因为查询本身是基于SQL的，难道不能使用一般的Statement在JSP页面中执行等价的SQL吗？也许可以，但不这样做有几点原因：

- 查询已经在本地Microsoft Access环境中编写和测试，可能开发了上百个查询，而很少为转换做过调整。
- 如果查询在其初始形式内被修改，改动将自动映射到基于Web的版本中。
- 查询可使用Access中给出的数据库特性，但并不一定在ODBC到JDBC-ODBC桥的层次上被支持。

13.5 结果集

一个结果集是表格行的排序列表，使用JDBC中的java.sql.ResultSet接口表示。结果集由executeQuery ( ) 或一些元数据方法调用产生。一旦创建，结果集中的数据可如下抽取出来：

1. 调用ResultSet.next ( ) 方法或JDBC 2.0提供的大量方法的集合中方法之一absolute ( )、relative ( )、next ( )、previous ( )、first ( )、last ( )、beforeFirst ( ) 或afterLast ( ) 移动到所需行。

2. 使用ResultSet.getXXX(columnNumber)或ResultSet.getXXX(columnName)检索所需列值，这里XXX为JDBC数据类型。

下面是一个简单例子，使用JSP页面搜索 LyricNote公司composer数据库中出生在Ireland的成员：

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<HTML>
<HEAD>
<TITLE>Irish Composers</TITLE>
</HEAD>
<BODY>
<H3>Irish Composers</H3>
<TABLE BORDER=0 CELLPADDING=3 CELLSPACING=1>
<%
   // JDBC driver name and database URL can be stored
   // in web.xml as context parameters so that they
   // do not have to be hard-coded.

   String DRIVER = application.getInitParameter("jdbc.driver");
   String URL = application.getInitParameter("jdbc.url.interna.");

   // Load the driver

   Class.forName(DRIVER);

   Connection con = null;
   try {
```

精彩好书见www.book118.com


```
        // Connect to the database

        con = DriverManager.getConnection(URL);
        Statement stmt = con.createStatement();

        // Create a query to select Irish composers

        String sql =
           "SELECT lname, fname, born, died"
           + " FROM composers"
           + " WHERE nationality = 'Irish'";

        // Execute the query to create a result set

        ResultSet rs = stmt.executeQuery(sql);

        // Loop through each row of the result set

        while (rs.next()) {
           /  Extract the two string values and two
           // integer values from the current row

           String lastName = rs.getString(1);
           String firstName = rs.getString(2);
           int born = rs.getInt(3);
           int died = rs.getInt(4);

           // Print a table row with the values

%>
<TR>
   <TD><%= firstName %> <%= lastName %></TD>
   <TD><%= born %>-<%= died %></TD>
</TR>
<%
        }

        // After last row is printed, close the result set
        // and the statement

        rs.close();
        stmt.close();
     }

     // Always close the connection
```

精彩好书见www.book118.com


```
    finally {
       if (con != null) {
          con.close();
          con = null;
       }
    }
%>
</TABLE>
</BODY>
</HTML>
```

当Statement执行查询时，创建一个ResultSet对象。JSP页面通过使用next（）方法读取每一行，然后使用getString（）或getInt（）抽取每一列值。结果如图13-11所示。

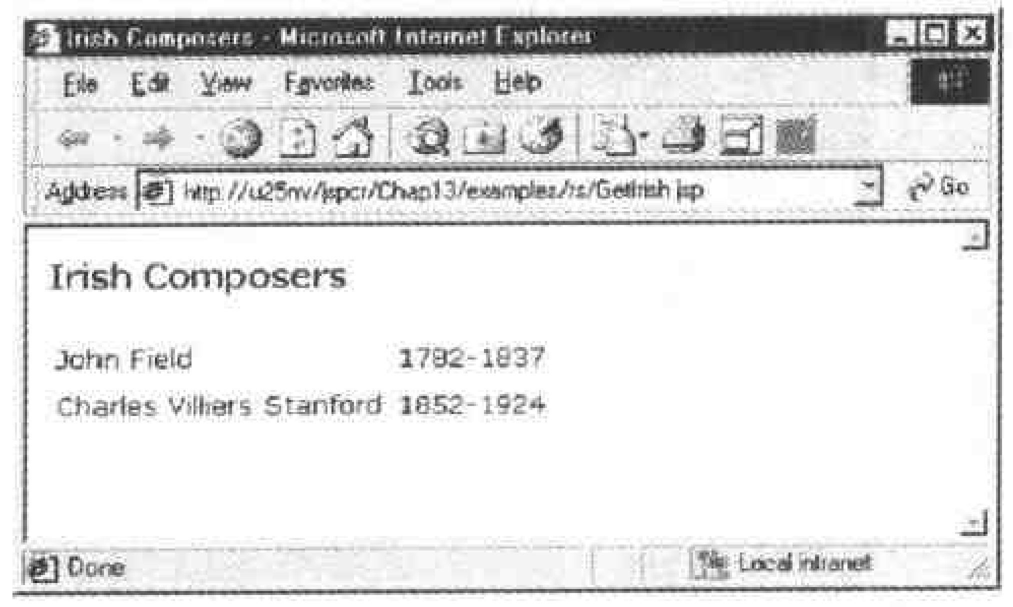


图13-11　结果集处理的一个简单实例

在一个ResultSet对象上可以调用大量getXXX（）方法。表13-1包含其完整列表。

每个getXXX（）方法存在两个版本：一个带有整型列号（1，2，...），一个带有一列名字符串。通过列号访问列可能更有效，虽然当域的次序改变时列名访问可能更容易。

表13-1　ResultSet提供的getXXX（）方法

方　法	描　述
getArray	返回一个SQL数组
getAsciiStream	返回一个打开的ASCII字符的java.io.InputStream。到ASCII（如果需要）的转换由JDBC驱动器处理
getBigDecimal	返回一个java.math.BigDecimal
getBinaryStream	返回一个打开的java.io.InputStream，在流上不进行转换
getBlob	返回一个java.sql.Blob（二进制大对象）
getBoolean	返回一个boolean值
getByte	返回一个单字节
getBytes	返回一个字节数组

精彩好书见www.book118.com

（续）

方法	描述
getCharacterStream	返回一个java.io.Reader字符流
getClob	返回一个java.sql.Clob（字符大对象）
getDate	返回一个java.sql.Date。注意，它是java.util.Date的子类
getDouble	返回一个双精度浮点值
getFloat	返回一个浮点值
getInt	返回一个整型值
getLong	返回一个长整型值
getObject	返回一个java.lang.Object
getRef	返回一个java.sql.Ref。这是SQL结构化类型值的引用
getShort	返回一个短整型值
getString	返回一个字符串
getTime	返回一个java.sql.Time值
getTimestamp	返回一个java.sql.Timestamp值，它包含毫秒为单位的时间

JDBC 2.0在结果集中引入了许多新的特性，在下面3节分别讨论。

13.5.1 可滚动的结果集

结果集原来只能沿一个方向（向前）导航，并只能开始在一个点（第一行）。有了JDBC 2.0，程序员有了更大的灵活性。游标（行指针）可以操作好象它是一个数组索引。存在向前和向后进行读取的方法，可以在任意行开始并测试当前游标位置。表13-2列出可利用的导航方法。

表13-2 JDBC 2.0对可滚动结果集的导航方法

方法	描述
boolean next()	提升游标到下一行
boolean previous()	将游标后移一行
boolean first()	将游标移到第一行
boolean last()	将游标移到最后一行
void beforeFirst()	将游标移到第一行前，通常伴随有next（）的调用
void afterLast()	将游标移到最后一行后，通常伴随有previous（）的调用
boolean absolute (int row)	将游标移到指定行，指定一个正数移动游标是相对于结果集尾。绝对值（-1）则与last（）相同
boolean relative(int row)	前移或后移游标指定行数
boolean isBeforeFirst()	如果游标在第一行前，则为true
boolean isAfterLast()	如果游标在最后一行后，则为true
boolean isFirst()	如果游标定位在第一行，则为true
boolean isLast()	如果游标定位在最后一行，则为true

为使用结果集，必须创建Statement对象带有指出请求特定功能的参数。为此，Connection.createStatement（）方法存在一种新的形式：

```
public Statement createStatement
(int resultSetType, int resultSetConcurrency)
throws SQLException
```

精彩好书见www.book118.com

这里resultSetType是用于滚动的类型，resultSetConcurrency指出是否修改结果集。两个参数值都取自ResultSet中的常量。如表13-3所示。

表13-3 可用于描述滚动结果集的ResultSet中的常量

常 量	含 义
TYPE_FORWARD_ONLY	JDBC 1.0风格导航，在其中游标开始在第一行，只能向前移动
TYPE_SCROLL_INSENSITIVE	使能所有游标定位的方法；结果集并未反映出底层表格中其他类型所做的改变
TYPE_SCROLL_SENSITIVE	使能所有游标定位的方法；结果集反映出底层表格中其他类型所做的改变
CONCUR_READ_ONLY	结果集未改变
CONCUR_UPDATABLE	可以加入和删除行，可以修改列

下述JSP页面是使用可滚动结果集显示潜在很长查询页面的例子。

```
<%@ page import="java.sql.*" %>
<%@ page import="java.text.*" %>
<%!
   public static final DecimalFormat PRICE_FMT
      = new DecimalFormat("$#,###.00");
%>
<HTML>
<HEAD>
<TITLE>Scrollable Example</TITLE>
</HEAD>
<BODY>
<IMG SRC="images/lyric_note.png" BORDER=0><P>
<HR COLOR="#000000">
<%
   // Get driver name and database URL from configuration
   // parameters stored in web.xml

   String DRIVER = application.getInitParameter("jdbc.driver");
   String URL = application.getInitParameter("jdbc.url");

   // Load the driver

   Class.forName(DRIVER);

   Connection con = null;
   Statement stmt = null;
   ResultSet rs = null;

   try {

      // Connect to the database

      con = DriverManager.getConnection(URL);
```

精彩好书见www.book118.com


```
        // Open a statement that supports scrollable result sets

        stmt = con.createStatement(
                        ResultSet.TYPE_SCROLL_INSENSITIVE,
                        ResultSet.CONCUR_READ_ONLY);

        // Execute the query

        rs = stmt.executeQuery(
                  " SELECT itemcode, price, description"
                + " FROM products"
                + " WHERE prodtype = 'IN'"
                + " ORDER BY description"
           );

        // Calculate number of rows

        rs.last();
        int nRows = rs.getRow();

        // Back up ten rows

        rs.relative(-10);

        // Now print last page of result set
%>
<H3>
   Musical Instruments
   - Items <%= rs.getRow() + 1 %> through <%= nRows %>
</H3>
<TABLE BORDER=1 CELLPADDING=3 CELLSPACING=0>
<TR><TH>Item</TH><TH>Price</TH><TH>Description</TH></TR>
<%
      while (rs.next()) {
         String itemcode = rs.getString(1);
         double price = rs.getLong(2) / 100.0;
         String description = rs.getString(3);
%>
<TR>
   <TD><%= itemcode %></TD>
   <TD ALIGN="RIGHT"><%= PRICE_FMT.format(price) %></TD>
   <TD><%= description %></TD>
</TR>
<%
      }
```

精彩好书见www.book118.com


```
        }
        finally {
            if (rs != null) { rs.close(); rs = null; }
            if (stmt != null) { stmt.close(); stmt = null; }
            if (con != null) { con.close(); con = null; }
        }
%>
</TABLE>
</BODY>
</HTML>
```

Statement对象被打开以便其创建的结果集可滚动，但不可修改。有了这些属性，ResultSet可以被告之其包含了多少行，而这在JDBC 1.0中是不可能的。通过定位游标到最后一行，发出一个relative（-10）方法调用，结果集中最后10行可被分离开并打印，图13-12显示出结果。

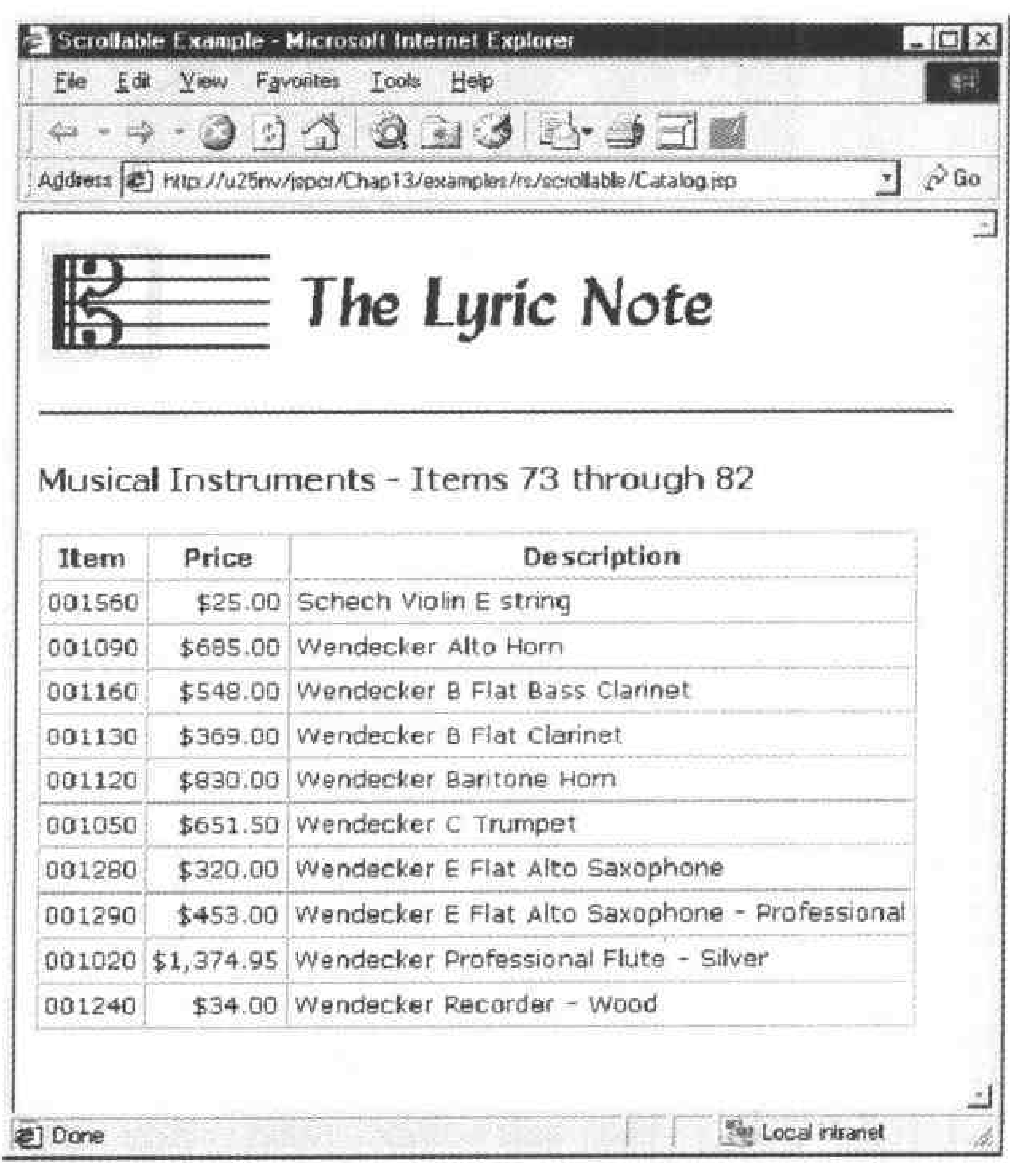


图13-12　显示使用可滚动结果集进行一个冗长查询的最后页面

精彩好书见www.book118.com

13.5.2 可修改结果集

有了JDBC 2.0，在结果集中修改列就成为可能，加入新行和删除现有行。每一情况中，底层表格中的相应行也跟着改变。

对于要修改的结果集，它必须由使用ResultSet.CONCUR_UPDATABLE创建的Statement对象所产生。JDBC 2.0给出updateXXX（）方法，这里XXX类似于存在的getXXX（）方法，是JDBC数据类型。这些方法带有列号或列名参数和一个取值参数，如下面例子所示：

```
double mySalary = rs.getDouble("SALARY");
mySalary *= 2.0;
rs.updateDouble("SALARY", mySalary);
rs.updateString("HOME_PHONE", unlisted);
rs.updateRow();
```

被修改值不会自动复制到底层表格中，除非调用updateRow( )。如果仍没有调用updateRow( )或在updateRow（）前隐含调用游标移动方法，可以使用ResultSet.cancelRowUpdates（）显式取消修改。

可以使用insertRow（）向结果集和底层表格加入新行。这包括一个特殊的游标位置，称为插入行。下面例子解释了其工作方式：

```
rs.moveToInsertRow();
rs.setString("employeeid", "M1205");
rs.setString("firstName", "Maria");
rs.setString("lastName", "Alicia");
rs.insertRow();
rs.moveToCurrentRow(); // Return to where we were
```

以类似方式，可以使用deleteRow（）删除结果集或底层表格中的行。为此，游标必须定位在要删除的行，如下所示：

```
rs.last(); // Delete the last row
rs.deleteRow();
```

13.5.3 RowSet

javax.sql包含有一个RowSet接口，它扩展了java.sql.ResultSet，因此它可以从它的数据库连接中分离出来。这对难维持一个连接和受内存限制的个人数字助手（PDA）应用很有益处。在出版时，RowSet仍处于开发阶段。Sun 微系统对此接口有3个早期的访问实现方案，可用于开发其功能：

- CachedRowSet 来自于一个JDBC结果集的序列化断连的RowSet。
- JdbcRowSet 也是来自于一个JDBC结果集的已连接RowSet，其行为符合JavaBean模型。
- WebRowSet 可将其内容写为XML文档的CachedRowSet的子类。

13.6 使用元数据

JDBC提供大量的元数据——数据的数据——对于数据库连接和结果集的集合。这一节给出

精彩好书见www.book118.com

这样的两个接口，如何获得其实例及其提供信息的高亮显示。

13.6.1 数据库元数据

关于JDBC连接的信息可使用Connection.getMetaData ( ) 获得。此方法返回java.sql.DatabaseMetaData的实例。java.sql.DatabaseMetaData是一个比java.sql或javax.sql包中任何其他接口或类中包含的方法都要多（总计149）的一个接口。这些方法描述了数据库支持的特性，其包含表格以及表格中的列。使用元数据，可以最小化SQL语言和数据库系统功能的差别。

查看一个DatabaseMetaData对象提供的所有信息可能具有指导作用。因为接口中有如此多的方法，为每一个的调用都进行手工编码是很枯燥的。为此，列出所有元数据方法清单，然后再调用每一方法并打印结果，使用这样的过程就比较容易了。下面JSP页面(MetadataExplorer.jsp)阐述了这种技术：

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
<%@ page import="java.lang.reflect.*" %>
<%
   // Get required driver name parameter

   String driverName = request.getParameter("driverName");
   if (driverName == null)
      driverName = "";
   driverName = driverName.trim();
   if (driverName.equals(""))
      throw new ServletException("No driverName parameter");

   // Get required database URL parameter

   String url = request.getParameter("url");
   if (url == null)
      url = "";
   url = url.trim();
   if (url.equals(""))
      throw new ServletException("No url parameter");

   // Get optional userID parameter

   String userID = request.getParameter("userID");
   if (userID == null)
      userID = "";
   userID = userID.trim();

   // Get optional password parameter
```

精彩好书见www.book118.com


```
String password = request.getParameter("password");
if (password == null)
   password = "";
password = password.trim();
// Load the driver

Class.forName(driverName);
Connection con = null;
try {

   // Open the database connection and get the metadata

   con = DriverManager.getConnection(url, userID, password);
   DatabaseMetaData md = con.getMetaData();

   // Use reflection to get a list of methods that the
   // metadata class supports. Select only public methods
   // that take no parameters and that return either
   // a string or a boolean.

   Class mdclass = md.getClass();
   Method[] methods = mdclass.getDeclaredMethods();
   Map methodMap = new TreeMap();

   for (int i = 0; i < methods.length; i++) {
      Method method = methods[i];

      // Public methods only

      if (!Modifier.isPublic(method.getModifiers()))
         continue;

      // with no parameters

      if (method.getParameterTypes().length > 0)
         continue;

      // that return String or boolean

      Class returnType = method.getReturnType();
      if ((returnType != java.lang.Boolean.TYPE) &&
         (returnType != java.lang.String.class))
         continue;

      // Add selected methods to sorted map
```

精彩好书见www.book118.com


```
            methodMap.put(method.getName(), method);
        }
%>
<HTML>
<HEAD>
<TITLE>Metadata Explorer</TITLE>
<LINK REL="stylesheet" HREF="style.css">
</HEAD>
<BODY>
<CENTER>
<H3>
Metadata Explorer for
<%= md.getDatabaseProductName() %>
<%= md.getDatabaseProductVersion() %>
<BR>
[<%= driverName %>]
</H3>
<TABLE BORDER=0 CELLPADDING=3 CELLSPACING=1>
<TR CLASS="header">
   <TH CLASS="header">Method</TH>
   <TH CLASS="header">Value</TH>
</TR>
<%
      // Generate the table

      int row = 0;
      Iterator im = methodMap.keySet().iterator();
      while (im.hasNext()) {
         String methodName = (String) im.next();
         Object methodValue = null;

         Method method = (Method) methodMap.get(methodName);

         // Invoke the method and get the result

         try {
            Object[] noParameters = new Object[0];
            methodValue = method.invoke(md, noParameters);
         }
         catch (Exception ignore) {}

         // Display the results

         row++;
         String rowClass = "row" + (row % 2);
%>
```

精彩好书见www.book118.com


```
<TR CLASS="<%= rowClass %>">
   <TD><%= methodName %></TD>
   <TD><%= formatLine(methodValue) %></TD>
</TR>
<%
      }
   }
   finally {
      if (con != null)
         con.close();
   }
%>
</TABLE>
</CENTER>
</BODY>
</HTML>
<%!
   /**
   * Formats an object in an HTML-friendly way,
   * making sure it doesn't exceed 48 characters
   * in width.
   */
   private static String formatLine(Object obj)
   {
      if (obj == null)
         return "";

      StringBuffer out = new StringBuffer();
      StringBuffer line = new StringBuffer();
      StringTokenizer st =
         new StringTokenizer(obj.toString(), ",;", true);

      while (st.hasMoreTokens()) {
         if (line.length() > 48) {
            out.append(line.toString());
            out.append("<BR>");
            line = new StringBuffer();
         }
         line.append(st.nextToken());
      }
      out.append(line.toString());

      return out.toString();
   }
%>
```

精彩好书见www.book118.com

此JSP页面设计为从一个提供驱动器名、JDBC URL、用户ID和密码域的HTML窗体发出调用。如图13-13所示。

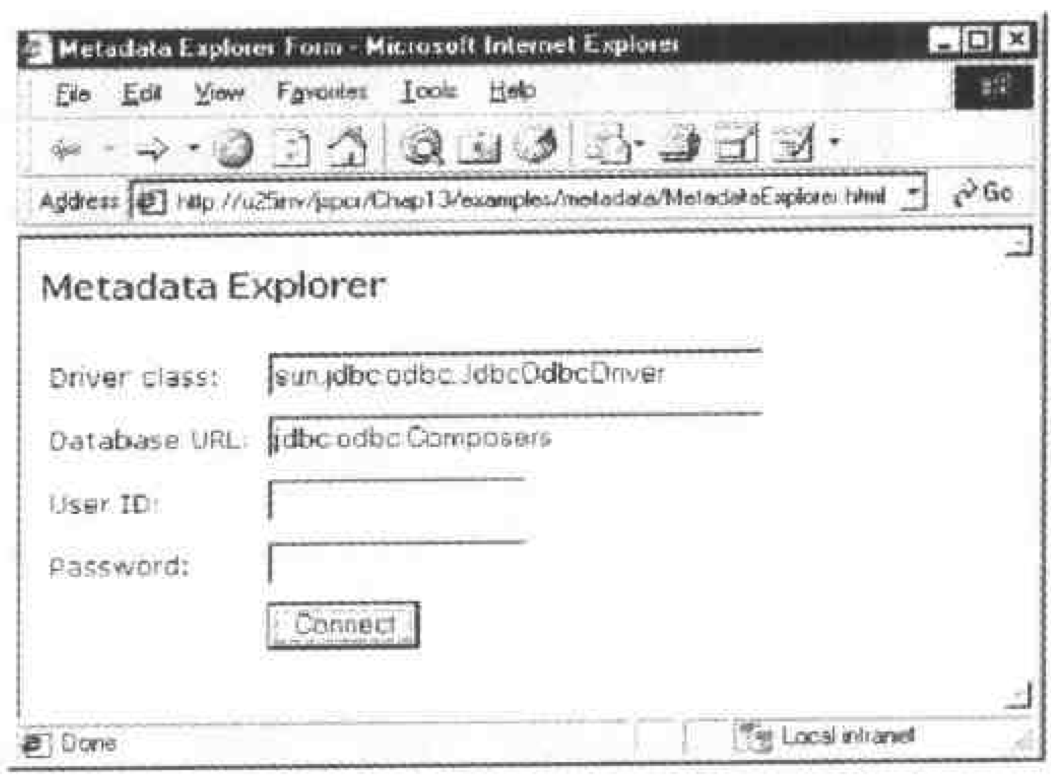


图13-13　MetadataExplorer.jsp的参数输入窗体

当运行在一个使用JDBC-ODBC桥的MicroSoft Access数据库中，MetadataExplorer产生输出部分如图13-14所示。完整列表包含在表13-4中。

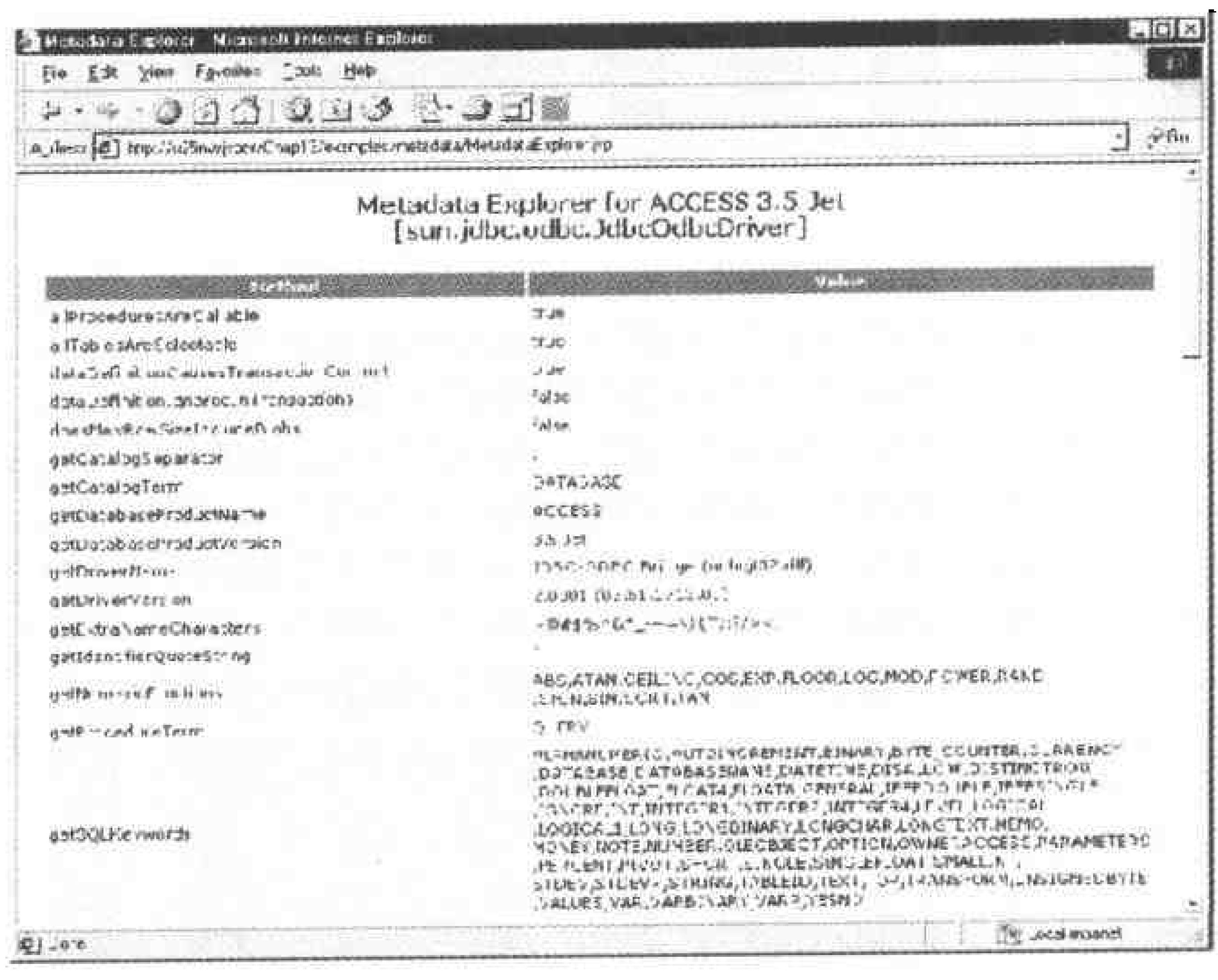


图13-14　使用MicroSoft Access数据库的MetadataExplorer

精彩好书见www.book118.com

表13-4 MicroSoft Access数据库中的元数据

方法	取值
allProceduresAreCallable	True
allTablesAreSelectable	True
dataDefinitionCausesTransactionCommit	True
dataDefinitionIgnoredInTransactions	False
doesMaxRowSizeIncludeBlobs	False
getCatalogSeparator	.
getCatalogTerm	DATABASE
getDatabaseProductName	ACCESS
getDatabaseProductVersion	3.5 Jet
getDriverName	JDBC-ODBC Bridge (odbcjt32.dll)
getDriverVersion	2.0001 (03.51.1713.00)
getExtraNameCharacters	~@#$%^&*_-+=\}{"';:?/><,
getIdentifierQuoteString	`
getNumericFunctions	ABS, ATAN, CEILING, COS, EXP, FLOOR, LOG, MOD, POWER, RAND, SIGN, SIN, SQRT, TAN
getProcedureTerm	QUERY
getSQLKeywords	ALPHANUMERIC, AUTOINCREMENT, BINARY, BYTE, COUNTER, CURRENCY, DATABASE, DATABASENAME, DATETIME, DISALLOW, DISTINCTROW, DOUBLEFLOAT, FLOAT4, FLOAT8, GENERAL, IEEEDOUBLE, IEEESINGLE, IGNORE, INT, INTEGER1, INTEGER2, INTEGER4, LEVEL, LOGICAL, LOGICAL1, LONG, LONGBINARY, LONGCHAR, LONGTEXT, MEMO, MONEY, NOTE, NUMBER, OLEOBJECT, OPTION, OWNERACCESS, PARAMETERS, PERCENT, PIVOT, SHORT, SINGLE, SINGLEFLOAT, SMALLINT, STDEV, STDEVP, STRING, TABLEID, TEXT, TOP, TRANSFORM, UNSIGNEDBYTE, VALUES, VAR, VARBINARY, VARP, YESNO
getSchemaTerm	\
getStringFunctions	ASCII, CHAR, CONCAT, LCASE, LEFT, LENGTH, LOCATE, LOCATE_2, LTRIM,

精彩好书见www.book118.com

(续)

方 法	取 值
	RIGHT, RTRIM, SPACE, SUBSTRING, UCASE
getSystemFunctions	CURDATE, CURTIME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, HOUR, MINUTE, MONTH, NOW, SECOND, WEEK, YEAR
getURL	jdbc:odbc:Composers
getUserName	admin
isCatalogAtStart	True
isReadOnly	False
nullPlusNonNullIsNull	False
nullsAreSortedAtEnd	False
nullsAreSortedAtStart	False
nullsAreSortedHigh	False
nullsAreSortedLow	True
storesLowerCaseIdentifiers	False
storesLowerCaseQuoted Identifiers	False
storesMixedCaseIdentifiers	False
storesMixedCaseQuoted Identifiers	True
storesUpperCaseIdentifiers	False
storesUpperCaseQuoted Identifiers	False
supportsANSI92EntryLevelSQL	True
supportsANSI92FullSQL	False
supportsANSI92IntermediateSQL	False
supportsAlterTableWith AddColumn	True
supportsAlterTableWith DropColumn	True
supportsBatchUpdates	True
supportsCatalogsInData Manipulation	True
supportsCatalogsInIndex Definitions	True
supportsCatalogsInPrivilege Definitions	False
supportsCatalogsInProcedure Calls	False
supportsCatalogsInTable	True

精彩好书见www.book118.com

（续）

方法	取值
Definitions	
supportsColumnAliasing	True
supportsConvert	True
supportsCoreSQLGrammar	False
supportsCorrelatedSubqueries	True
supportsDataDefinitionAndData ManipulationTransactions	True
supportsDataManipulation TransactionsOnly	False
supportsDifferentTable CorrelationNames	False
supportsExpressionsInOrderBy	True
supportsExtendedSQLGrammar	False
supportsFullOuterJoins	False
supportsGroupBy	True
supportsGroupByBeyondSelect	True
supportsGroupByUnrelated	False
supportsIntegrity EnhancementFacility	False
supportsLikeEscapeClause	False
supportsLimitedOuterJoins	False
supportsMinimumSQLGrammar	True
supportsMixedCaseIdentifiers	True
supportsMixedCaseQuoted Identifiers	False
supportsMultipleResultSets	False
supportsMultipleTransactions	True
supportsNonNullableColumns	False
supportsOpenCursorsAcross Commit	False
supportsOpenCursorsAcross Rollback	False
supportsOpenStatements AcrossCommit	True
supportsOpenStatements AcrossRollback	True
supportsOrderByUnrelated	False
supportsOuterJoins	True
supportsPositionedDelete	False
supportsPositionedUpdate	False
supportsSchemasInData	False

精彩好书见www.book118.com

（续）

方 法	取 值
Manipulation	
supportsSchemasInIndex Definitions	False
supportsSchemasInPrivilege Definitions	False
supportsSchemasInProcedure Calls	False
supportsSchemasInTable Definitions	False
supportsSelectForUpdate	False
supportsStoredProcedures	True
supportsSubqueriesIn Comparisons	True
supportsSubqueriesInExists	True
supportsSubqueriesInIns	True
supportsSubqueriesIn Quantifieds	True
supportsTableCorrelationNames	True
supportsTransactions	True
supportsUnion	True
supportsUnionAll	True
usesLocalFilePerTable	False
usesLocalFiles True	

13.6.2 ResultSetMetadata

DatabaseMetaData除了用于数据库连接，ResultSetMetadata也可以取得关于一个结果集列的信息。此接口由取得列号的方法——getColumnCount（ ）——和20个描述每一列的其他方法组成。

为得到一个ResultSetMetadata对象，程序调用ResultSet.getMetadata()方法，然后调用其getColumnCount（ ）方法，向其传递一个列号参数。对ResultSet来说，列号开始为1。

表13-5 给出ResultSetMetadata中可利用的方法

方 法	描 述
getColumnCount()	返回结果集中每一行中的列号
getCatalogName(int col)	返回从中抽取指定列的表的目录名
getColumnClassName(int col)	返回指定列的全质Java类型的名字
getColumnDisplaySize(int col)	返回指定列的最大显示宽度
getColumnLabel(int col)	返回指定列标签
getColumnName(int col)	返回指定列名字

精彩好书见www.book118.com

(续)

方 法	描 述
getColumnType(int col)	以与java.sql.Types对应的形式返回指定列类型
getColumnTypeName(int col)	返回字符串型列数据类型
getPrecision(int col)	返回小数位置数
getScale(int col)	返回小数点右边的数字
getSchemaName(int col)	返回列所在表的模式名
getTableName(int col)	返回列的底层表格的名字
isAutoIncrement(int col)	如果列自动编号，则为真
isCaseSensitive(int col)	如果列大小写敏感，则为真
isCurrency(int col)	如果列为一货币值，则为真
isDefinitelyWritable(int col)	如果对指定列的写入肯定成功，则为真
isNullable(int col)	返回指出列是否可以有null值的一个常量
isReadOnly(int col)	如果结果集为只读，则为真
isSearchable(int col)	如果此列可以使用在一where子句，则为真
isSigned(int col)	如果列值为有符号数字，则为真
isWritable(int col)	如果对指定列的写入可能成功，则为真

13.7 JDBC 2.0及以上版本中的新特性

JDBC 2.0最初指的是JDBC 2.0标准扩展API，现在被命名为JDBC 2.0可选包API。它包含在配有Java 2标准版本的JDBC 2.1核心API包中。本章讨论了其许多新特性，包括：

- DataSource JDBC驱动器名和URL可被保存在一名称服务中，并使用JNDI检索。
- 连接池 数据源提供者可以提供连接池，允许连接被激活和重复利用，通常会大幅度提高性能。此功能全部在名字服务中配置，对应用不需改变。
- 可滚动结果集 JDBC 1.0只允许通过在第一个记录开始的结果集向前导航。JDBC 2.0提供向前和向后导航的方法以及相对和绝对的游标定位。
- RowSets 断连的结果集可以操作以符合JavaBean模型。
- BatchUpdates 事务处理可以被分组并作为一个单元发送到数据库。

JDBC 3.0第一个公开草案发布于2000年9月。其新特性包括：

- 增强控件的确认/回滚事务处理边界。
- 连接池的配置。
- 对已有和可调用语句的参数更好的接口。

13.8 小结

几乎所有重要的JSP应用都需要访问数据库。Java提供一个标准的API称为JDBC。JDBC允许面向对象框架集中标准的SQL语句访问各种类型数据库系统。为使用JDBC，一个驱动器必须为数据库可利用。驱动器存在于所有的商业数据库，以及使用ODBC数据源的一个JDBC-ODBC桥。

精彩好书见www.book118.com

在JDBC中只有很少的关键对象，很容易掌握。Connection对象保持一个数据库的激活连接。3类Statement对象允许通过连接执行SQL语句并在一个ResultSet对象中捕获结果集。连接和结果集的大量信息可从DatabaseMetaData和ResultSetMetaData对象中得到。

JDBC已经有了几个版本，每一新版本都有一些增强特性，并承诺将继续作为Java编程主要的数据库访问技术。

精彩好书见www.book118.com

第14章 会话和线程管理

超文本传输协议（HTTP）最初的设计意图是在WWW上发布文档和图象。因此，它使用了一个相当简单的通信模型。客户端对文档进行请求，服务器响应以文档或错误代码，最终事务处理完成。服务器不会保留请求的任何信息，下一次客户端进行请求时，服务器没有方法可以将它与其他客户端区分开来。称此HTTP是无状态协议。

不幸的是，很少有应用适合这种简单的请求/响应模型。在大多数情况下，请求是有意义的工作。例如，应用可能有一个Web页面提示输入用户ID和密码，然后是搜索页面请求一个关键字在产品数据库中进行查询，再后面是匹配产品的列表、详细的产品信息页面，购物卡结算页面和订购总结页面。这些页面每一个都依赖于前一个页面，也依赖于服务器知道此时客户端应用的状态。更糟的是，应用客户端上的用户可能会前进或后退页面，或是完全进入另一Web页面，但却没有通知服务器会话已经终止或对任意一部分工作应如何继续。相关的难点是服务器进程运行时间过长——比Web服务器为保持合理的性能而等待的时间要长。

这些已经不是新问题，CGI程序和在线事务处理系统几年前就遇到了同样的问题。在这些环境下,应用的技术仍然是工作于servlet/JSP环境中，但Java servlet API有一种提供清晰、易用的内置机制方案：HTTP会话。

本章介绍使JSP模型适合应用模型的两个关键特征：会话管理和线程管理。讨论了会话跟踪的4种技术，基本集中在HTTP会话API，检验如何创建会话，其管理对象的方式以及如何终止，然后介绍多线程应用的Java嵌入式支持和可用的servlet线程模型。最后一节介绍关于对象生命期和可视化的应用问题。

14.1 会话跟踪

因为Web服务器在请求之间不会记住客户端，因此保持一个会话的惟一方式是客户端跟踪会话。实现此功能有两种基本方式：

- 客户端记住所有会话的相关数据并在必要时将之发回到服务器。
- 服务器保持所有数据，对其设置一个标识，让客户端记住该标识。

第一种方案实现比较简单，不需要服务器部分加入特殊功能。此方案需要来回传送大量的数据，可能会降低性能。另一问题是服务器端对象，如数据库和网络连接对每一请求必须被重新初始化。为此，此方案最适合于小批量数据的长期持久性发送。如用户优先权和帐号。

第二种方案功能更多，一旦服务器初始化了一个会话且客户端接受了它，服务器就可以构建复杂的，有效的对象并保存大量的数据，但只需一个关键字就可以区分会话。本章大部分集中讨论这一方案。

如何使客户端记住数据并将之返回到Web服务器？常用以下4种技术：

- 隐藏域

精彩好书见www.book118.com

- URL重写
- Cookie
- HTTP会话API

下面各节详细讨论每一技术。

14.1.1 隐藏域

HTML窗体支持HIDDEN类型的输入元素。隐藏域与HTTP请求中其他窗体参数一起被发送到Web服务器，但它们没有任何可视表示。它们只用于包含一个请求的字母或常量值。类似技术也用于CICS和主框架事务处理监视器提供事务处理代码。理论上，隐藏域可用于一般HTML Web页面，但如果要进行会话跟踪，它们就必须由类似CGI、servlet或JSP服务器进程创建的动态生成的Web页面。

隐藏域非常适合不需大量数据存储或对象初始化的来回的会话式应用。一个典型实例是Tomcat实例文件夹中包含的猜数游戏。此游戏选择1到100之间的一个随机整数，然后让用户猜它。每一次猜测后，游戏告诉用户每一次猜测值是否太小、太大或刚刚好。

下面JSP给出的是一个刚好相反的游戏：让用户考虑一个1到100之间的数，然后猜测它，依据是用户指出每一次猜测值是否太小，太大或刚刚好[1]。此JSP使用二进制搜索查找该数：

```
<%@ page session="false" %>
<H3>Number Guess Guesser</H3>
<%
   int wayLo = 1 - 1;
   int wayHi = 100 + 1;
   int state = 0;
   String parm = request.getParameter("state");
   if (parm != null)
      state = Integer.parseInt(parm);

   switch (state) {
      case 0: { // Initial screen
%>
<FORM>
Think of a number between
<%= wayLo + 1 %> and <%= wayHi - 1 %>,
and I'll try to guess it.<P>
Click OK when ready.<P>
<INPUT TYPE="submit" VALUE="OK">
<INPUT TYPE="hidden" NAME="lo" VALUE="<%= wayLo %>">
<INPUT TYPE="hidden" NAME="hi" VALUE="<%= wayHi %>">
<INPUT TYPE="hidden" NAME="numGuesses" VALUE="0">
```

1 如果你在一单独的窗口中和另外一人进行此游戏，可以在双方进行过程中看到彼此礼貌的评语。

精彩好书见www.book118.com


```
<INPUT TYPE="hidden" NAME="state" VALUE="1">
</FORM>
<%
            break;
        }
        case 1: { // First guess
            int numGuesses = 1 + Integer.parseInt
                (request.getParameter("numGuesses"));
            int lo = Integer.parseInt(request.getParameter("lo"));
            int hi = Integer.parseInt(request.getParameter("hi"));
            int guess = (hi + lo)/2;
%>
<FORM>
My first guess is <%= guess %>. How did I do?<P>
<INPUT TYPE="radio"
            NAME="result"
            VALUE="-1" onClick="submit()"> Too low
<INPUT TYPE="radio"
            NAME="result"
            VALUE="0" onClick="submit()"> Exactly right
<INPUT TYPE="radio"
            NAME="result"
            VALUE="1" onClick="submit()"> Too high
<P>
<INPUT TYPE="hidden" NAME="lo" VALUE="<%= lo %>">
<INPUT TYPE="hidden" NAME="hi" VALUE="<%= hi %>">
<INPUT TYPE="hidden" NAME="numGuesses" VALUE="<%= numGuesses %>">
<INPUT TYPE="hidden" NAME="state" VALUE="2">
</FORM>
<%
            break;
        }
        case 2: { // After first guess
            int numGuesses = 1 + Integer.parseInt
                (request.getParameter("numGuesses"));
            int lo = Integer.parseInt(request.getParameter("lo"));
            int hi = Integer.parseInt(request.getParameter("hi"));
            int result =
                Integer.parseInt(request.getParameter("result"));
            int guess = (hi + lo)/2;

            if (result < 0) {
                lo = guess;
                guess = (hi + lo)/2;
            }
            else if (result > 0) {
```

精彩好书见www.book118.com


```
                hi = guess;
                guess = (hi + lo)/2;
            }

            if (result != 0) {
%>
<FORM>
<%
    if (lo > wayLo)
        out.println(lo + " is too low.<BR>");
    if (hi < wayHi)
        out.println(hi + " is too high.<BR>");
    if ((hi - lo) > 1) {
%>
My next guess is <%= guess %>. How did I do?<P>
<INPUT TYPE="radio"
       NAME="result"
       VALUE="-1" onClick="submit()"> Too low
<INPUT TYPE="radio"
       NAME="result"
       VALUE="0" onClick="submit()"> Exactly right
<INPUT TYPE="radio"
       NAME="result"
       VALUE="1" onClick="submit()"> Too high
<P>
<INPUT TYPE="hidden" NAME="lo" VALUE="<%= lo %>">
<INPUT TYPE="hidden" NAME="hi" VALUE="<%= hi %>">
<INPUT TYPE="hidden" NAME="numGuesses" VALUE="<%= numGuesses %>">
<INPUT TYPE="hidden" NAME="state" VALUE="2">
</FORM>
<%
    }
    else {
        String[] text = {
            "Are we cheating?",
            "Did we forget our number?",
            "Perhaps we clicked the wrong button?",
            "What happened?",
            "What gives?",
        };
        String message = text[(int)(Math.random() * text.length)];
%>
<FORM>
<%= message %><P>
<INPUT TYPE="SUBMIT" VALUE="Start Over">
</FORM>
```

精彩好书见www.book118.com


```
<%
    }
        }
        else {
            numGuesses--;
%>
<FORM>
I win, and after only <%= numGuesses %> guesses!<P>
Do you want to try again?<P>
<INPUT TYPE="SUBMIT" VALUE="Start Over">
</FORM>
<%
            }
            break;
        }
    }
%>
```

JSP页面使用一个名为state的隐藏域来跟踪游戏中发生的事件。基于该state，它显示出相应的窗体：

- State 0 初始窗体解释该游戏和设置要用到的变量。包括state、猜测数、作为太低参数的最高值、作为太高参数的最低值。所有变量作为窗体中隐藏域保存。
- State 1 用户点击OK按钮后，程序检索太低和太高参数，使用两者的平均值作为下一次猜测数。窗体向用户显示3个单选按钮指出猜测是否太低、太高或刚好。低和高值、用户结果选择和猜测的累计次数再次作为隐藏保存。
- State 2 基于用户在单选按钮中指定的内容，程序使用新的上或下边界修改太低或太高值。如果猜测刚刚好，程序发出恭喜信息，提示是否再玩一次。同时，显示已知的上下边界和下一次猜测数。如图14-1所示。

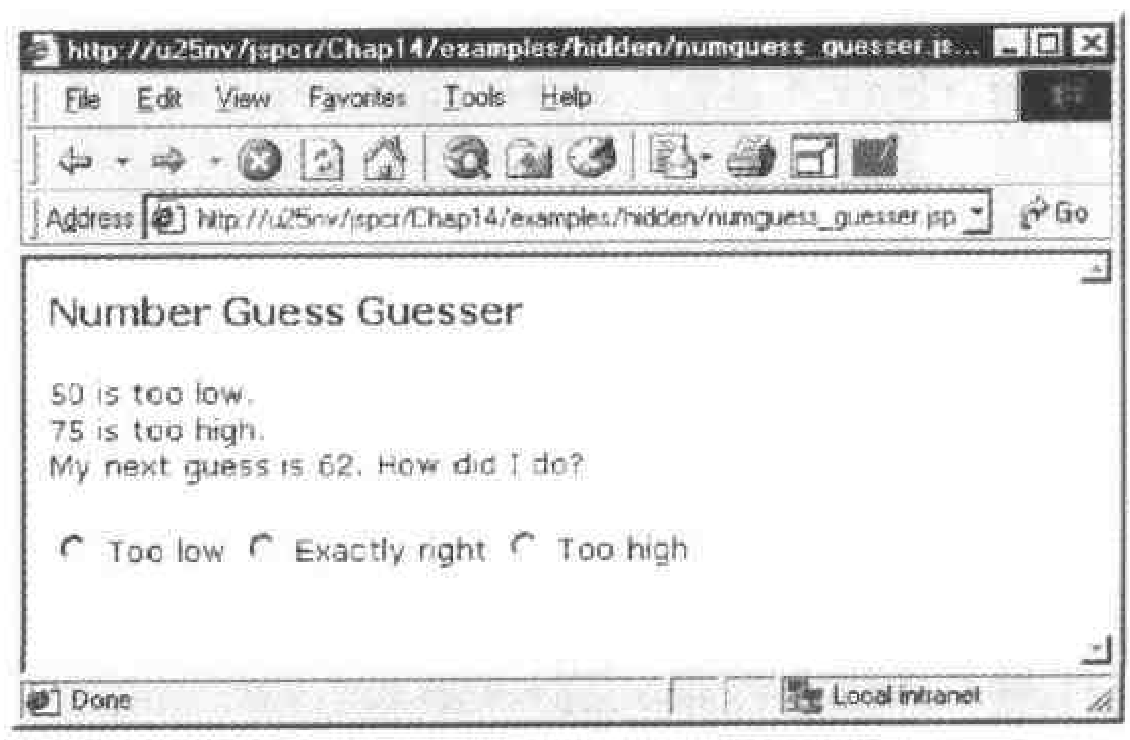


图14-1 数字猜测游戏

精彩好书见www.book118.com

具有隐藏域的程序只能用于HTML窗体中。如果用户点击超级链接，离开此页面，隐藏域失效，除非采纳了下一节描述的技术——URL重写。

14.1.2 URL重写

URL可以在后面附加参数，和服务器的请求一起发送。这些参数为名字/取值对，语法如下：

```
http://server/MyPage.jsp?name1=value1&name2=value2&...
```

JSP页面检索请求时，可如下读取参数值：

```
String value1 = request.getParameter("name1");
String value2 = request.getParameter("name2");
...
```

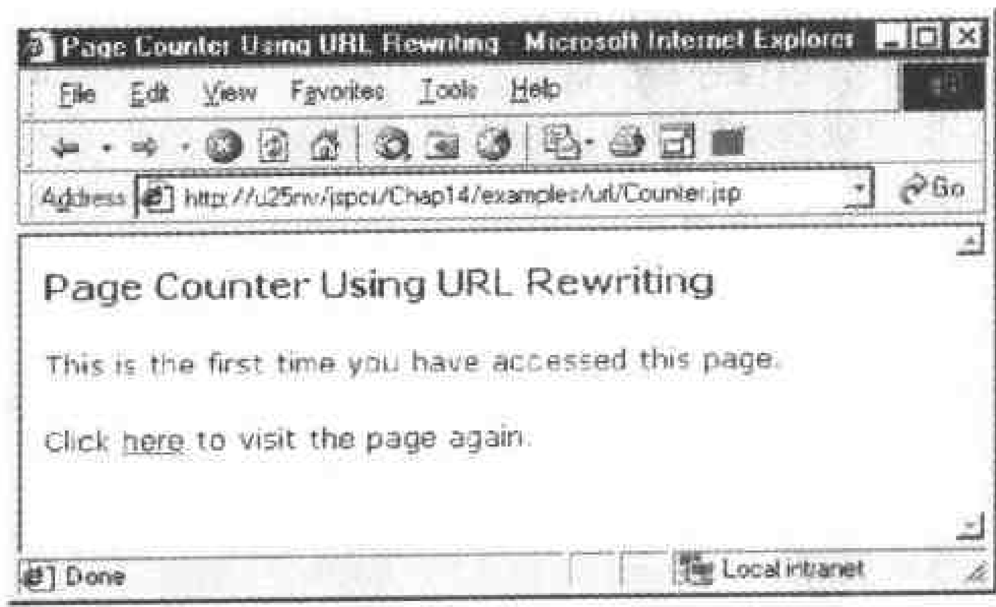
动态生成的Web页面可以利用此功能在作为超级链接写入页面的URL中保存会话数据。此功能允许客户端提醒保存所有必要值的服务器将服务器应用置入所需状态。

一个简单例子指出用户在当前会话期间访问一个页面的时间数的计数器。如下面列表所示：

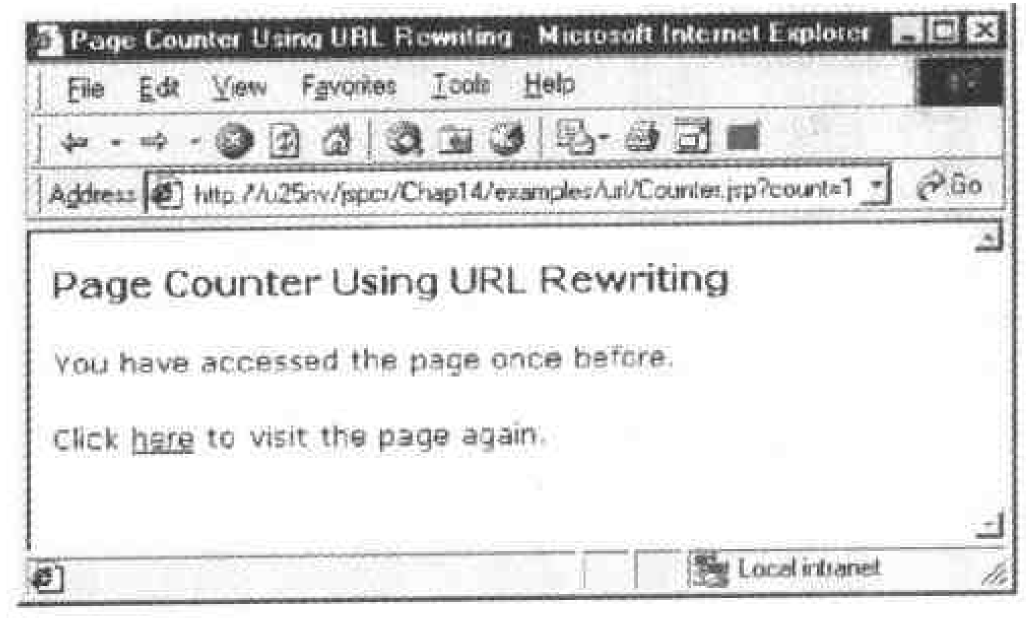
```
<%@ page session="false" %>
<HTML>
<HEAD>
<TITLE>Page Counter Using URL Rewriting</TITLE>
</HEAD>
<BODY>
<H3>Page Counter Using URL Rewriting</H3>
<%
   int count = 0;
   String parm = request.getParameter("count");
   if (parm != null)
      count = Integer.parseInt(parm);
   if (count == 0) {
%> This is the first time you have accessed this page. <%
   }
   else if (count == 1) {
%> You have accessed the page once before.<%
   }
   else {
%> You have accessed the page <%= count %> times before.<%
   }
%>
<P> Click
<A HREF="Counter.jsp?count=<%=count + 1 %>"
   >here</A> to visit the page again.
</BODY>
</HTML>
```

精彩好书见www.book118.com

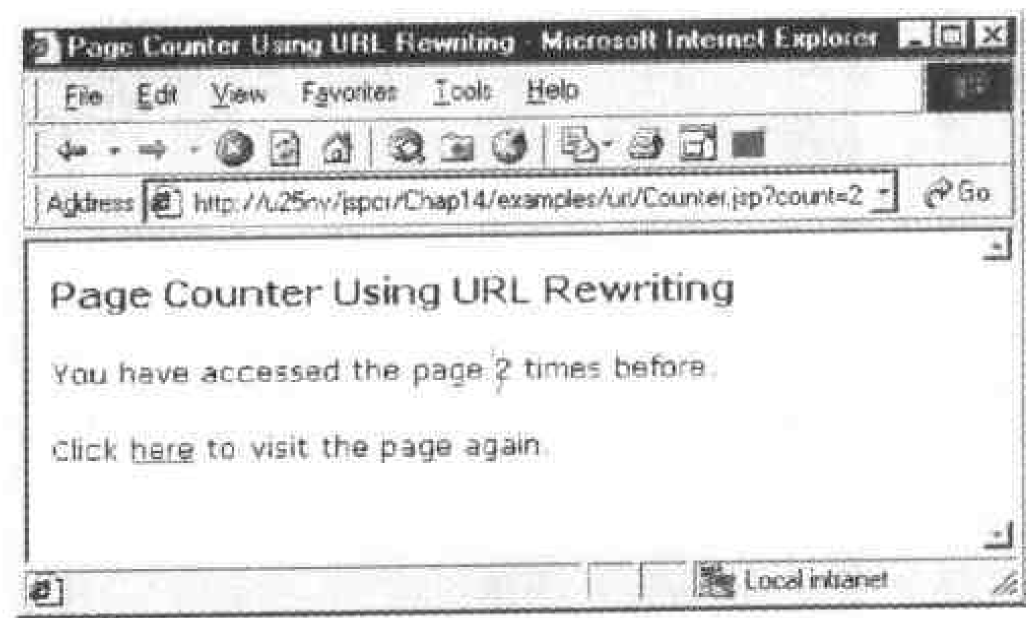
当用户只使用基本URL第一次请求页面时，count参数不存在，因此整数count变量设置为0：



页面的底部是再次调用同样counter.jsp页面的一个超级链接。但这一次，counter参数值比当前计数大1：



每次页面再被调用，计数被修改，同时显示信息发生变化。



此技术可确保在所有浏览器环境和安全设置中有效，这是它惟一的优点。如果保存大量的数据，此技术会降低性能。URL可能变得很大，有可能会超出Web服务器所接受的长度。另外，

精彩好书见www.book118.com

这样的URL是不安全的，在浏览器地址窗口和Web服务器注册中均可看到。需要页面上每个URL被重写必然会产生大量的冗长乏味的代码，很容易在过程中漏掉一个URL。不过，对于简单应用，URL重写是可信赖的，也很容易实现。

注意，通常不会手工向超级链接URL中附加参数。更常见的是使用HTTP会话 API执行URL重写。这样的话，只需附加一个会话ID。

14.1.3 cookie

持续性客户端数据存储最常用的技术包括HTTP cookie。一个cookie是一个小的、已命名数据元素。服务器使用Set-Cookie头标将之作为HTTP响应的一部分传送到客户端。客户端被要求保存cookie，在对同一服务器的后续请求使用一个cookie头标将之返回到服务器。连同名字和取值，cookie还包括：

- 终止日期，该时间后，客户端不在要求保留此cookie。如果未指定日期，一旦浏览器会话结束，则cookie终止。
- 域名，如servername.com，它限制了设置cookie有效的URL的子集。如果未指定，则返回所有请求初始Web服务器的cookie。
- 更深一步限制URL子集的路径名。
- secure属性，如果给出，表明连接使用了一个安全隧道，如SSL，则只返回cookie。

最初的cookie规范细节请见http://home.netscape.com/newsref/std/cookie_spec.html。

图14-2解释了如何设置cookie以及使用HTTP请求和响应对其检索。首先，Web浏览器从Web服务器请求一个页面，这时不包含cookie，当服务器响应以请求文档时，它发送一个Set-Cookie头标对名为language的cookie赋值为fr。此cookie有效期设为一年。浏览器读取此头标，抽出cookie信息，在其cookie缓存中存储名字/取值对，以及Web服务器的域和缺省路径。以后，当用

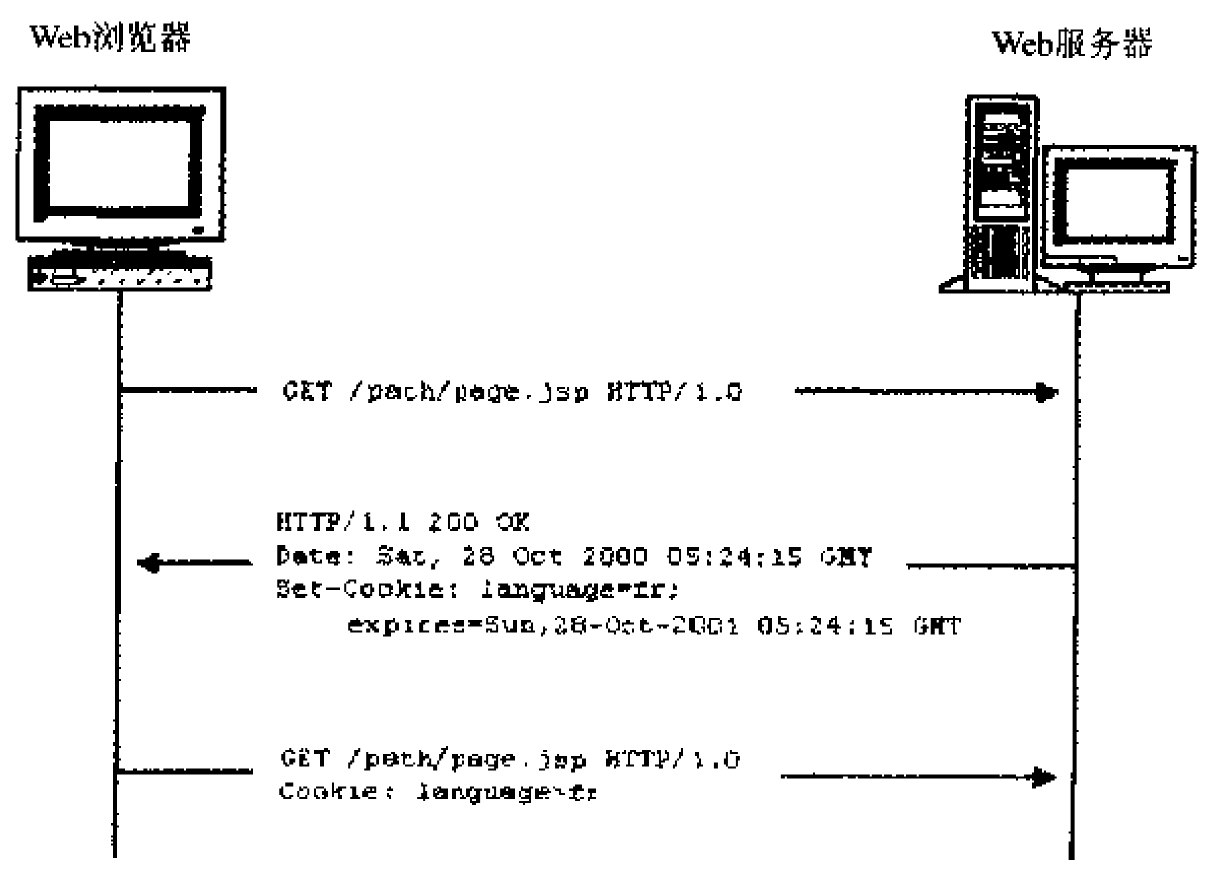


图14-2　设置和检索cookie的HTTP头标

精彩好书见www.book118.com

户再次访问页面时，浏览器识别出它以前从此服务器中收到一个cookie，且此cookie没有终止，因此将cookie发回服务器。

与其他持续性规划比较cookie的一个优点是在浏览器会话结束后，甚至是在客户端计算机重启后它仍可保留其值。这使得cookie非常适合于维持用户的选择，如语言。下面给出的应用使用户可以通过点击一个超级链接选择所需语言。选择使得两个cookie被发送到客户端：一个是语言，一个是国家。下一次用户访问站点时，浏览器会自动将cookie发回服务器，用户选中的语言在页面中被使用。

```
<%@ page session="false" %>
<%@ page import="java.util.*" %>
<jsp:include page="getLocale.jsp" flush="true"/>
<%
   ResourceBundle RB =
      (ResourceBundle) request.getAttribute("RB");
%>
<HTML>
<HEAD>
<TITLE>Using Cookies to Store Preferences</TITLE>
</HEAD>
<BODY>
<IMG SRC="images/lyric_note.png"><P>
<HR>
<jsp:include page="languageBar.jsp" flush="true"/>
<H3><%= RB.getString("greeting") %></H3>
</BODY>
</HTML>
```

主应用页面index.jsp使用<jsp:include>调用一个实用的JSP页面扫描存在cookie的请求头标并返回适当语言的一个资源包[1]。以下是实用页面getLocate.jsp：

```
<%@ page session="false" %>
<%@ page import="java.util.*" %>
<%
   // Look through cookies for language and country

   String language = null;
   String country = null;

   Cookie[] cookies = request.getCookies();
   if (cookies != null) {
```

1 一个java.util.ResourceBundle对象是程序以不同的语言检索消息和其他字符串的一种方式。这样程序就可用于多个现场，而无需做任何改动。ResourceBundle存在多个实现，其中最常用的一种使用一个一般的.properties文件存储消息文本。

精彩好书见www.book118.com


```
        for (int i = 0; i < cookies.length; i++) {
            Cookie cookie = cookies[i];
            String name = cookie.getName();
            if (name.equals("language"))
                language = cookie.getValue();
            if (name.equals("country"))
                country = cookie.getValue();
        }
    }

    // Get locale-specific resources

    Locale locale = null;
    if (language != null && country != null)
        locale = new Locale(language, country);
    if (locale == null)
        locale = Locale.getDefault();

    ResourceBundle RB = ResourceBundle.getBundle
        ("jspcr.sessions.welcome", locale);

    // Store the resource bundle as an attribute of the request

    request.setAttribute("RB", RB);
%>
```

getLocale.jsp使用request.getCookies（）取得请求中所有cookie的一个数组。它通过列表查找language和country cookie。如果getLocale知道它们，它为此语言和国家创建一个java.util.Locale。如果未找到，（第一次用户访问页面时的情况就是这样）则使用缺省现场。总之，它载入与此应用和现场相关的资源包，然后将该包保存为一个请求属性。getLocale.jsp从<jsp:include>返回后，index.jsp检索资源包并使用ResourceBundle.getString()得到被转换的文本。

index.jsp调用另一实用页面——languageBar.jsp创建语言选择超级链接。存储在每一超级链接中的是主页面URL（包括任意参数）以及语言和国家代码。以下是languageBar.jsp:

```
<%@ page session="false" %>
<%@ page import="java.util.*" %>
<%
    String thisURL = HttpUtils.getRequestURL(request).toString();
    thisURL = java.net.URLEncoder.encode(thisURL);

    Object[][] locales = {
        {new Locale("en", "US"), "English"},
        {new Locale("de", "DE"), "Deutsch"},
        {new Locale("es", "ES"), "Español"},
```

精彩好书见www.book118.com


```
        {new Locale("fr", "FR"), "Français"},
        {new Locale("it", "IT"), "Italiano"},
    };

    for (int i = 0; i < locales.length; i++) {
        Locale locale = (Locale) locales[i][0];
        String name = (String) locales[i][1];

        StringBuffer sb = new StringBuffer();
        if (i > 0)
        sb.append(" | ");
        sb.append("<A HREF=\"setPreferences.jsp?cameFrom=");
        sb.append(thisURL);
        sb.append("&language=");
        sb.append(locale.getLanguage());
        sb.append("&country=");
        sb.append(locale.getCountry());
        sb.append("\"");
        sb.append(">");
        sb.append(name);
        sb.append("</A>");
        out.println(sb);
    }
%>
```

超级链接调用setPreferences.jsp生成适当的语言和国家cookie，然后将其加入待发的响应头标。接下来，JSP重定向浏览器回到调用页面，如下所示：

```
<%@ page session="false" %>
<%
   // Set cookies for language and country

   final int ONE_YEAR = 60 * 60 * 24 * 365;
   String[] parms = { "language", "country" };
   for (int i = 0; i < parms.length; i++) {
      String name = parms[i];
      String value = request.getParameter(name);
      if (value != null) {
         Cookie cookie = new Cookie(name, value);
         cookie.setMaxAge(ONE_YEAR);
         response.addCookie(cookie);
      }
   }
// Redirect back to the calling JSP

   String cameFrom = request.getParameter("cameFrom");
   if (cameFrom == null)
```

精彩好书见www.book118.com


```
        cameFrom = request.getContextPath();
    response.sendRedirect(cameFrom);
%>
```

index.jsp最初出现在缺省现场，如图14-3所示。如果用户点击French超级链接，调用setPreferences.jsp页面重定向浏览器到index.jsp中，此时带有捆绑的cookie。结果显示页面的French版本。如图14-4所示。如果用户下次访问该站点，则记住此语言选择并应用它。

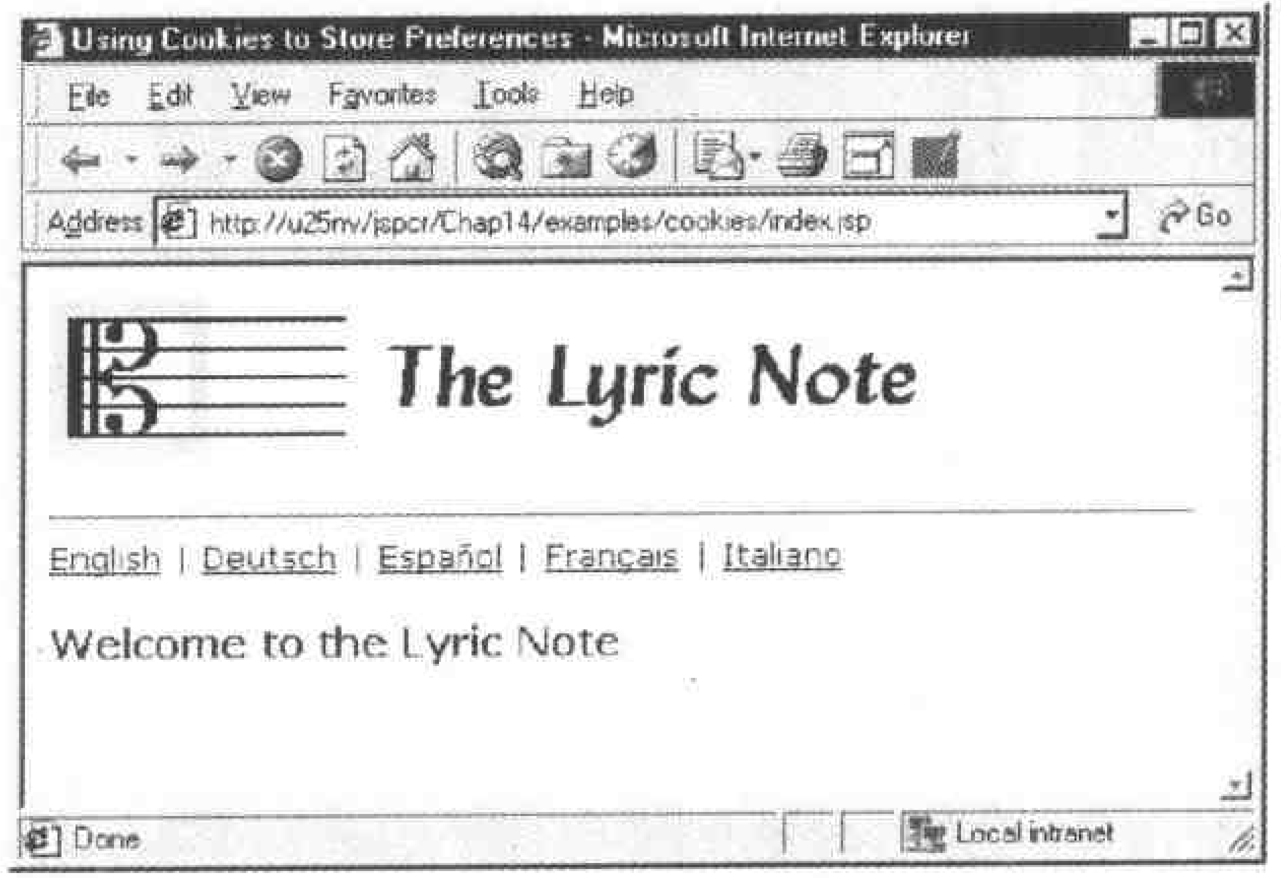


图14-3 显示语言选择条的LyricNote主页面

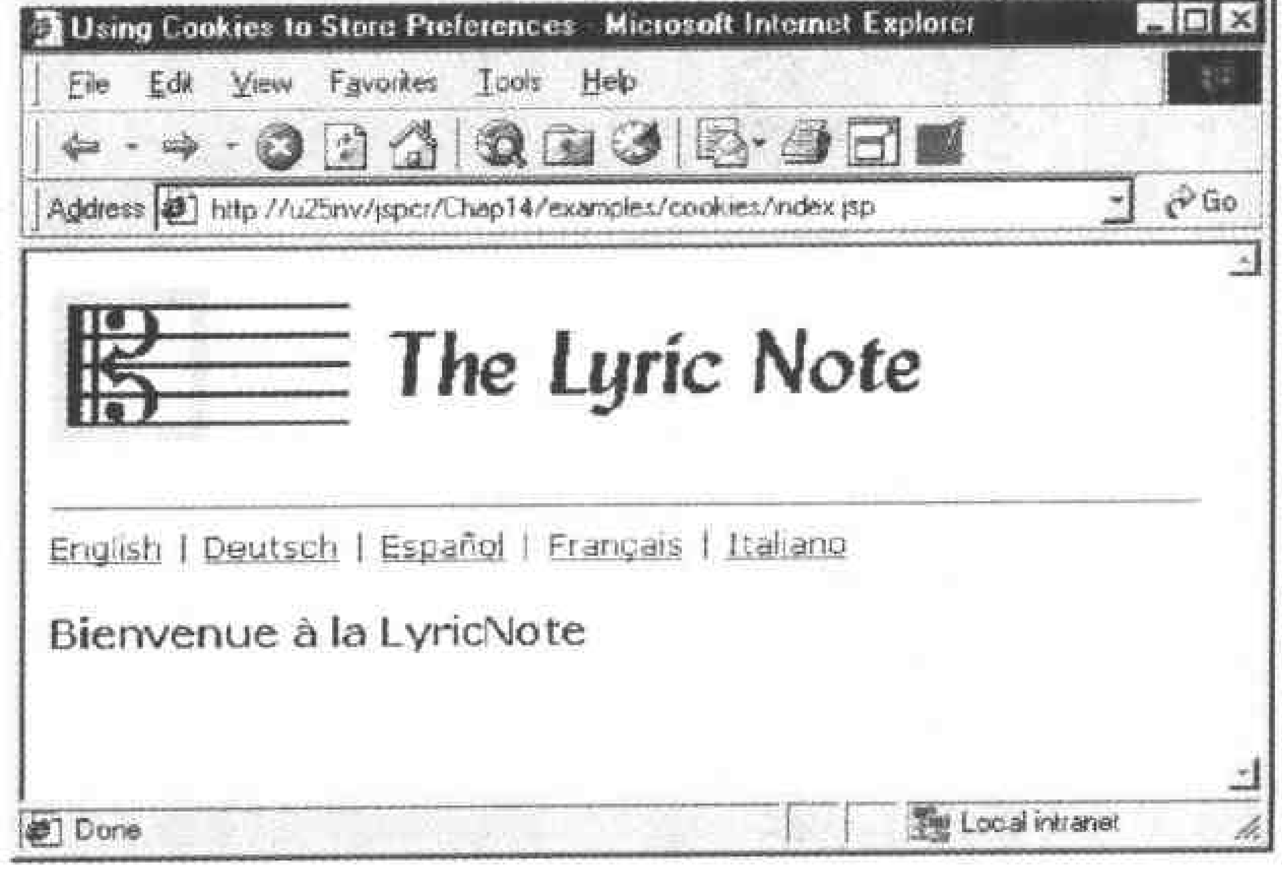


图14-4 LyricNote主页面的French版本

精彩好书见www.book118.com

Cookie的主要问题是用户可以由于私人原因而关闭其浏览器对cookie的支持。这意味着如果应用不使用cookie，则应用必须准备用其他方式完成工作。

14.2 会话API

到此为止，已经介绍了两种会话跟踪的通用方法，它们都要求客户端记住状态：

- 要求客户端保存所有会话数据并将其返回到每一请求的服务器。
- 要求客户端保存会话标识而让服务器处理工作。

第一种方法很容易实现，第二种方法通常提供了更多的功能和可扩展性，可以看出隐藏域、URL重写和cookie在某种程度上都可以用来支持两种方法。但大多数需要使用会话的JSP页面和servlet可以利用一种更高层的方法：HttpSession API。

Javax.servlet.http包中3个类组成了会话API：

- HttpSession　像一个Map或HashTable一样的接口，能够根据名字存储和检索对象。可通过调用HttpServletRequest.getSession（ ）创建一个会话，并保留它直至其超时或被参与一个会话的servlet关闭。携带会话标识的传入HTTP请求自动与会话相关。
- HttpSessionBindingListener　允许一个对象知道其何时被存储在一个会话中或从中删除的接口。此接口有两个回馈方法 ValueBound（ ）和valueUnbound（ ），对象必须实现它们以接收捆绑的标志信息。
- HttpSessionBindingEvent　传递到HttpSessionBindingListener的ValueBound（ ）和valueUnbound（ ）方法的一个事件对象。该事件具有返回会话的方法和侦听者捆绑到会话的名字。

14.2.1 创建会话

servlet通过调用HttpServletRequest中的getSession（ ）或getSession（boolean create）方法指出它要使用的会话。如下所示：

```
HttpSession session = request.getSession(true);
```

不带参数的getSession（ ）方法是调用getSession（true）的一种简便写法。Create参数指出如果会话不存在，servlet引擎是否应该创建一个新的会话。如果参数是false，则servlet只能对存在的会话实施操作。每种情况下，都要检验请求，查看它是否包含一个有效的会话ID，如果是，servlet容器返回会话对象的引用，然后用它来存储和检索会话属性。

在一个JSP页面中，会话创建是自动的，除非它在page伪指令中被禁止。在生成servlet中的_jspService（ ）方法的开头，创建和初始化PageContext对象，作为初始化的一部分，JspFactory.getPageContext（ ）方法调用request.getSession(true)。当它调用pageContext.getSession（ ）时，新创建或被访问的会话返回到生成的servlet中，然后会话作为隐含变量session对JSP页面的其余部分可以访问，如下所示：

```
public void _jspService(
        HttpServletRequest request,
```

精彩好书见www.book118.com


```
        HttpServletResponse response)
    throws ServletException, IOException
{
    PageContext pageContext = _jspFactory.getPageContext
        (this, request, response, null, true, 8192, true);

    JspWriter out = pageContext.getOut();
    HttpSession session = pageContext.getSession();
    ...
}
```

如果一个JSP页面不需要使用会话，它应该在page伪指令中禁止自动创建过程：

```
<%@ page session="false" %>
```

这使得当会话不必要时，将servlet引擎从必须创建和维护会话的义务中解脱出来。对不必要的会话的创建内存需求是很大的。

当首次创建会话时，客户端（Web浏览器）仍然不知道它，当会话ID被发送到客户端，客户在下一请求中端又将其发回时，就意味客户端连接了会话。一个servlet或JSP页面可以使用isNew（）方法检测出此过程是否发生：

```
HttpSession session = request.getSession();
if (session.isNew()) {
    // Create an empty shopping cart
}
```

如果会话新创建，客户端仍未被通知，或者如果客户端已经被通知，但选择不加入，则Session.isNew（）为真。

会话跟踪机制

servlet引擎试图使用cookie跟踪会话ID。在创建一个会话的servlet所编写的HTTP响应中，包含会话ID的一个Set-Cookie头标在一个名为JSESSIONID的 cookie中[1]。

```
Set-Cookie: JSESSIONID=rkbg6z27j1;Path=/jspcr
```

如果客户端接受cookie，客户端将在后续请求中返回它：

```
Cookie: JSESSIONID=rkbg6z27j1
```

如果是这样，客户端请求可能与会话相关而不必特别考虑servlet部分。如果客户端不接受cookie，则会话失效。为防止这一点，servlet API提供一种预防机制。如果cookie失败，它使用URL重写。对程序员来说这有点复杂，因为这意味着所有servlet编写的URL都必须附加会话ID。

如果客户端接受cookie，这样做就是不必要且很昂贵的，如果肯定已经确定cookie方法失败，就只有使用URL重写了。幸运的是，servlet API有封装此逻辑方法HttpServletResponse类的方法encodeURL（）和encodeRedirectURL（）在必要时向URL中加入会话ID：

1 这由Servlet 2.2 API规定。某些servlet引擎使用一个不同的值。

精彩好书见www.book118.com


```
String myURL = response.encodeURL("/servlet/nextServlet");
out.println("Click <A HREF='"
    + myURL + "'>here</A>"
    + " to continue");
```

对于传递到response.sendRedirect（）方法的URL应使用encodeRedirectURL（），其他则使用encodeURL（）。

当使用encodeURL（）方法时，当session.isNew（）为真时，会话ID总是被嵌入URL中。来自客户端的第一次响应后，servlet引擎判断会话ID是否在一个cookie中被返回。如果没有，servlet引擎继续向通过encodeURL（）传递的URL附加会话ID。否则，它切换到使用cookie，encodeURL（）返回未加改动的URL字符串。这使得程序员可以不必测试所有可能情况。

14.2.2　从会话中保存和检索对象

使用setAttribute（）方法将对象捆绑到一个会话：

```
session.setAttribute("jspcr.sessions.myapp.user", userID);
```

对象捆绑到会话的名字可以是任意惟一字符串。因为会话是在当前HTTP会话中的所有servlet和JSP页面之间共享，然而使用一个与其他应用不冲突的名字是很有意义的。最常见的是选择那些带有包名或servlet或JSP页面全质类名前缀的名字。

在会话中可保存任意类型对象，但是因为会话可能被序列化，最好是让会话对象实现java.io.Serializable。注意，只有对象可以被保存，面不是伪指令，如int、char或double。为保存这些伪指令，必须使用其对象包容器Integer、Character或Double。

使用getAttribute（）方法从一个会话中检索对象：

```
String userID = (String) session.getAttribute(
    "jspcr.sessions.myapp.user");
```

像Map或HashTable一样，一个会话只保存对象，因此对其进行检索时，必须将其置入适当的类型。包容器类中包含的伪指令必须由包容器提供的方法抽取：

```
Integer countObject = (Integer) getAttribute("count");
int count = countObject.intValue();
```

通常如果在会话中保存了属性，就可以知道它的名字、类型，可以直接以这种方式请求它，但也可以从getAttributeName（）方法中得到属性名列表：

```
out.println("Objects in this session:");
out.println("<PRE>");
Enumeration enames = session.getAttributeNames();
while (enames.hasMoreElements()) {
   String name = (String) enames.nextElement();
   Object value = session.getAttribute(name);
   out.println(name + " = " + value);
}
out.println("</PRE>");
```

精彩好书见www.book118.com

当不再需要对象时，使用removeAttribute（）将其从会话中删除：

```
session.removeAttribute("jspcr.sessions.myapp.user");
```

会话关闭，此动作自动发生，但一个属性需要在这之前被删除时就必须这样做了。

14.2.3 销毁会话

一旦被创建，会话通常持续到其超时或其被关闭。Timeout指会话保持为有效的请求之间的最大时间长度。这是一个很重要的因素，因为服务器没有什么方式可以知道一个客户是否经完成了对会话的工作，除了被显式地通知或等待固定长时间。

缺省超时间隔长度可以在发布描述器web.xml中设置：

```
<web-app>
   ...
   <session-config>
      <session-timeout> 30 </session-timeout>
   </session-config>
   ...
</web-app>
```

此间隔指定为分钟数，缺省为30。这里给出的值支持到应用中所有会话，除非它们分别覆盖它。

使用稀有资源如数据库连接的某些应用可以选择无限超时。这些应用可能使用setMaxInactiveInterval（）方法选择一个更短的时间周期：

```
session.setMaxInactiveInterval(180);
```

向setMaxInactiveInterval（）提供的参数是秒数[1]。上面例子使用了180秒或3分钟。当前值可从getMaxInactiveInterval（）中得到。如果指定了一个负值，则会话永不超时。

在某些情况下，应提供会话的肯定性结束。对此可使用invalidate（）方法：

```
session.invalidate();
```

此方法标记会话为未激活，解缚其捆绑的所有对象。例如，在购物卡应用中，使用一个会话保存排序的条目，结算逻辑将次序写入数据库后，会话应被销毁。这样如果用户再次购买条目项，旧的会话内容就不会存在。

14.2.4 修订后实例

会话API可以处理本章前面介绍的所有会话跟踪任务。在这一节，学习隐藏域、URL重写和cookie实例如何使用同一会话API方法实现。

1. *隐藏域实例——猜数游戏*

在隐藏域一节开发的猜数游戏可通过将所有隐藏域移到保存在一个HTTP会话的对象中而大

---

1 此API在这里有点不一致。为什么在发布描述器中使用分钟而在会话API中使用秒呢？

精彩好书见www.book118.com

大简化。此实例中对象是一个名为Parameters的内部类，它定义在JSP页面头部附近，但与一个定义外部类一样容易：

```
<%@ page session="true" %>
<H3>Number Guess Guesser</H3>
<%!
    public static final int WAY_LO = 0;
    public static final int WAY_HI = 101;
    public static final String PARMSKEY
        = "jspcr.sessions.numguess.parameters";

    // inner class containing state variables

    public class Parameters {
        int lo;
        int hi;
        int numGuesses;
        int state;
    }
%>
<%
    Parameters parms=(Parameters) session.getAttribute(PARMSKEY);
    if (parms == null) {
        parms = new Parameters();
        parms.state = 0;
        session.setAttribute(PARMSKEY, parms);
    }

    switch (parms.state) {
        case 0: { // Initial screen
%>
<FORM>
Think of a number between
<%= WAY_LO + 1 %> and <%= WAY_HI - 1 %>,
and I'll try to guess it.<P>
Click OK when ready.<P>
<INPUT TYPE="submit" VALUE="OK">
</FORM>
<%
        parms.lo = WAY_LO;
        parms.hi = WAY_HI;
        parms.numGuesses = 0;
        parms.state = 1;
        break;
    }
    case 1: { // First guess
```

精彩好书见www.book118.com


```
        parms.numGuesses++;
        int guess = (parms.hi + parms.lo)/2;
%>
<FORM>
My first guess is <%= guess %>. How did I do?<P>
<INPUT TYPE="radio"
        NAME="result"
        VALUE="-1" onClick="submit()"> Too low
<INPUT TYPE="radio"
        NAME="result"
        VALUE="0" onClick="submit()"> Exactly right
<INPUT TYPE="radio"
        NAME="result"
        VALUE="1" onClick="submit()"> Too high
</FORM>
<P>
<%
            parms.state = 2;
            break;
        }
        case 2: { // After first guess
            parms.numGuesses++;
            int result =
                Integer.parseInt(request.getParameter("result"));
            int guess = (parms.hi + parms.lo)/2;

            if (result < 0) {
                parms.lo = guess;
                guess = (parms.hi + parms.lo)/2;
            }
            else if (result > 0) {
                parms.hi = guess;
                guess = (parms.hi + parms.lo)/2;
            }

            if (result != 0) {
%>
<FORM>
<%
    if (parms.lo > WAY_LO)
        out.println(parms.lo + " is too low.<BR>");
    if (parms.hi < WAY_HI)
        out.println(parms.hi + " is too high.<BR>");
    if ((parms.hi - parms.lo) > 1) {
%>
My next guess is <%= guess %>. How did I do?<P>
```

精彩好书见www.book118.com


```
<INPUT TYPE="radio"
       NAME="result"
       VALUE="-1" onClick="submit()"> Too low
<INPUT TYPE="radio"
       NAME="result"
       VALUE="0" onClick="submit()"> Exactly right
<INPUT TYPE="radio"
       NAME="result"
       VALUE="1" onClick="submit()"> Too high
</FORM>
<%
   }
   else {
      String[] text = {
         "Are we cheating?",
         "Did we forget our number?",
         "Perhaps we clicked the wrong button?",
         "What happened?",
         "What gives?",
      };
      String message = text[(int)(Math.random() * text.length)];
      session.removeAttribute(PARMSKEY);
%>
<FORM>
<%= message %><P>
<INPUT TYPE="SUBMIT" VALUE="Start Over">
</FORM>
<%
   }
         }
         else {
            parms.numGuesses--;
%>
<FORM>
I win, and after only <%= parms.numGuesses %> guesses!<P>
Do you want to try again?<P>
<INPUT TYPE="SUBMIT" VALUE="Start Over">
</FORM>
<%
         session.removeAttribute(PARMSKEY);
         }
         break;
      }
   }
%>
```

精彩好书见www.book118.com

逻辑保持不变，但这里隐藏域被写入HTML窗体，其值保存在捆绑到会话的Parameter对象中。

2. URL重写实例——页面记数器

类似地，在URL重写一节开发的页面计数器可以使用一个HTTP会话保存记数变量。因为int是一个伪指令，使用Integer对象包容器并调用器intValue（）方法得到实际值：

```
<%@ page session="true" %>
<HTML>
<HEAD>
<TITLE>Page Counter Using HTTP Session</TITLE>
</HEAD>
<BODY>
<H3>Page Counter Using HTTP Session</H3>
<%
   if (session.getAttribute("count") == null)
      session.setAttribute("count", new Integer(0));

   int count=((Integer) session.getAttribute("count")).intValue();

   switch (count) {
      case 0:
%> This is the first time you have accessed this page. <%
         break;
      case 1:
%> You have accessed the page once before.<%
         break;
      default:
%> You have accessed the page <%= count %> times before.<%
         break;
   }

   session.setAttribute("count", new Integer(count+1));
%>
<P>
Click
<A HREF="<%= response.encodeURL("Counter.jsp") %>">here</A>
to visit the page again.
</BODY>
</HTML>
```

每次页面被刷新，计数增加并保存在一个新的Integer包容器的会话中。注意，用户点击重新显示页面的超级链接使用response.encodeURL（）以确保会话跟踪工作，而无论用户是否承认cookie。

3. cookie实例——语言选择

在cookie一节，学习了一个应用如何允许用户指出其语言选择，以便在此会话下的Web页面

精彩好书见www.book118.com

的其余部分可以在此语言下显示。应用使用了cookie这样选择就会在会话之间得以保持。如果不需要这种持续性，可以使用会话API完成同样的工作。

主页面（index.jsp）改动很少。它仍对消息文本使用资源包，但现在它将其作为一个会话属性，而不是一个请求属性。另外，页面伪指令现在加入了session="true"。

```
<%@ page session="true" %>
<%@ page import="java.util.*" %>

<%-- Get the appropriate resource bundle from the session --%>

<jsp:include page="getLocale.jsp" flush="true"/>
<%
   ResourceBundle RB = (ResourceBundle)
      session.getAttribute("RB");
%>

<HTML>
<HEAD>
<TITLE>Using Session API to Store Language Preference</TITLE>
</HEAD>
<BODY>
<IMG SRC="images/lyric_note.png"><P>
<HR>
<%-- Show a row of hyperlinks with language choices --%>

<jsp:include page="languageBar.jsp" flush="true"/>

<%-- Display greeting in appropriate language --%>

<H3><%= RB.getString("greeting") %></H3>

</BODY>
</HTML>
```

被包含模块——getLocale.jsp和setPreferences.jsp是真正发生变动的地方。setPreferences现在执行基于其收到的语言和国家参数进行适当资源包的真正载入动作。

```
<%@ page session="true" %>
<%@ page import="java.util.*" %>
<%
   // Get parameters for language and country

   String language = request.getParameter("language");
   String country = request.getParameter("country");

   // Get locale-specific resources
```

精彩好书见www.book118.com


```
    Locale locale = null;
    if (language != null && country != null)
        locale = new Locale(language, country);
    if (locale == null)
        locale = Locale.getDefault();

    ResourceBundle RB = ResourceBundle.getBundle
        ("jspcr.sessions.welcome", locale);

    // Store the resource bundle as an attribute in the session

    session.setAttribute("RB", RB);

    // Redirect back to the calling JSP

    String cameFrom = request.getParameter("cameFrom");
    if (cameFrom == null)
        cameFrom = request.getContextPath();

    cameFrom = response.encodeRedirectURL(cameFrom);

    response.sendRedirect(cameFrom);
%>
```

资源包被保存为一个会话属性，用户被重定向到最初页面。注意，“camefrom” URL通过encodeRedirectURL（ ）方法传递，传递时cookie被关闭。

getLocale.JSP页面现在可以对资源包在会话中查找，或是在没找到的情况下使用缺省包：

```
<%@ page session="true" %>
<%@ page import="java.util.*" %>
<%
    // Get the existing resource bundle from the session,
    // if one exists

    ResourceBundle RB = (ResourceBundle)
        session.getAttribute("RB");
    // If not, use the default resource bundle

    if (RB == null) {
        RB = ResourceBundle.getBundle("jspcr.sessions.welcome");
        session.setAttribute("RB", RB);
    }
%>
```

languageBar.jsp页面只需进行两处改动。因为它向主页面和setPreferences.jsp写入URL，因此

精彩好书见www.book118.com

需要通过response.encodeURL（）传递URL。，这样即使cookie被关闭，会话跟踪工作也能进行：

```
<%@ page session="true" %>
<%@ page import="java.util.*" %>
<%
   String thisURL = HttpUtils.getRequestURL(request).toString();
   // Encode the session ID into the URL, if necessary
   thisURL = response.encodeURL(thisURL);
   thisURL = java.net.URLEncoder.encode(thisURL);

   Object[][] locales = {
      {new Locale("en", "US"), "English"},
      {new Locale("de", "DE"), "Deutsch"},
      {new Locale("es", "ES"), "Español"},
      {new Locale("fr", "FR"), "Français"},
      {new Locale("it", "IT"), "Italiano"},
   };

   for (int i = 0; i < locales.length; i++) {

      Locale locale = (Locale) locales[i][0];
      String name = (String) locales[i][1];

      StringBuffer sb = new StringBuffer();
      if (i > 0)
         sb.append(" | ");
      sb.append("<A HREF=\"");

      // Encode the session ID into the generated URL

      StringBuffer sb2 = new StringBuffer();
      sb2.append("setPreferences.jsp?cameFrom=");
      sb2.append(thisURL);
      sb2.append("&language=");
      sb2.append(locale.getLanguage());
      sb2.append("&country=");
      sb2.append(locale.getCountry());
      String url = sb2.toString();
      url = response.encodeURL(url);

      sb.append(url);
      sb.append("\"");
      sb.append(">");
      sb.append(name);
      sb.append("</A>");
      out.println(sb);
   }
%>
```

精彩好书见www.book118.com

14.2.5 会话捆绑侦听者

会话API提供跟踪对象何时被加入和删除的方式。要收到这些事件通知的对象可以实现HttpSessionBindinglistener接口。实现类必须提供两个方法：

- `public void valueBound(HttpSessionBindingEvent event)`
- `public void valueUnbound(HttpSessionBindingEvent event)`

两个方法中，均收到一个HttpSessionBindingEvent的实例。事件参数具有检索会话和判断对象捆绑到会话的名字的方法。

通过会话捆绑侦听者得到的主要优点是它们可以释放其获得的资源。而不管客户端显式关闭应用或会话超时。这使得此接口对管理数据库连接非常有用。JDBC 2.0提供连接池，但许多驱动器还没有实现它。这样的话，一种替换方式是使用一个知道使自己断连的会话驻留的连接。

下面例子解释了该技术。BoundConnection是一个包围java.sql.Connection对象的包容器，实现了HttpSessionBindinglistener，因此，它可以在其不再使用时关闭连接。

```
package jspcr.jdbc;

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

/**
* A wrapper for a <CODE>Connection</CODE>
* object that is aware it is in an HTTP session.
* This enables it to shut down the connection
* when the session is destroyed.
*/
public class BoundConnection
   implements HttpSessionBindingListener, Serializable
{
   private transient Connection connection;

   /**
   * Creates a new <CODE>BoundConnection</CODE> object
   * for the specified connection.
   * @param con the connection
   */
   public BoundConnection(Connection con)
   {
      this.connection = con;
   }
```

精彩好书见www.book118.com


```
/**
* Returns the underlying connection
*/
public Connection getConnection()
{
   return connection;
}

/**
* Called when the <CODE>BoundConnection</CODE>
* is stored in an HTTP session
* @param event the binding event
*/
public void valueBound(HttpSessionBindingEvent event)
{
   trace("bound", event);
}

/**
* Called when the <CODE>BoundConnection</CODE>
* is removed from an HTTP session
* @param event the unbinding event
*/
public void valueUnbound(HttpSessionBindingEvent event)
{
   if (connection != null)
      try {
         connection.close();
         connection = null;
      }
      catch (SQLException e) {
         e.printStackTrace();
      }
   trace("unbound", event);
}

/**
* Prints a trace message
*/
private void trace(String s, HttpSessionBindingEvent event)
{
   HttpSession session = event.getSession();

   java.util.Date now =
```

精彩好书见www.book118.com


```
        new java.util.Date(System.currentTimeMillis());
    java.util.Date last =
        new java.util.Date(session.getLastAccessedTime());

    SimpleDateFormat fmt = new SimpleDateFormat("hh:mm:ss");
    StringBuffer sb;

    sb = new StringBuffer();
    sb.append("TRACE: ");
    sb.append(fmt.format(now));
    sb.append(" session ");
    sb.append(session.getId());
    sb.append(" last accessed time ");
    sb.append(fmt.format(last));
    System.err.println(sb.toString());

    sb = new StringBuffer();
    sb.append("TRACE: ");
    sb.append(fmt.format(now));
    sb.append(" session ");
    sb.append(session.getId());
    sb.append(" connection " );
    sb.append(s);
    System.err.println(sb.toString());
  }
}
```

BoundConnection构造器保存一个Connection为一个私有实例变量，通过GetConnection（ ）方法使其可利用。BoundConnection实现了两个HttpSessionBindingListener方法：

valueBound（ ）和valueUnbound（ ）。在每个方法中，它写入了跟踪消息，因此当连接被捆绑或释放时都存在记录。关键特性是valueUnBound（ ）方法，它关闭底层连接。

注意 BoundConnection对象实现了Serializable，因为会话可以被序列化，特别是在分布式应用中。这样就可以将Connection实例变量标记为transient，结果是servlet容器不会试图对其序列化。因此getConnection（ ）调用者需要检测返回值是否为null，并在必要时创建一个新的BoundConnection。

当会话开始时，给定一个新打开的数据库连接，使用BoundConnection的一个JSP页面就可以调用该对象。当BoundConnection保存在会话中，其valueBound（ ）方法被激发。在同一会话中的后续请求可以检索来自会话的BoundConnection并调用其getConnection（ ）方法以得到底层的java.sql.Connection。下面给出可重复使用的connect.jsp模块实现此逻辑。

```
<%@ page import="java.sql.*" %>
<%@ page import="jspcr.jdbc.*" %>
<%
   // If there is not already a connection bound to this
```

精彩好书见www.book118.com


```
    // session, create one

    if (session.getAttribute("bcon") == null) {

      String driver =
        application.getInitParameter("jdbc.driver");
      String url =
        application.getInitParameter("jdbc.url.internal");

      Class.forName(driver);
      Connection con = DriverManager.getConnection(url);

      // Bind the connection to this session

      BoundConnection bcon = new BoundConnection(con);
      session.setAttribute("bcon", bcon);

      // Set the timeout interval to three minutes

      session.setMaxInactiveInterval(180);
    }
%>
```

除了在必要时创建BoundConnection，connect.jsp设置会话超时时间间隔为3分钟。

下面给出的应用使用BoundConnection提供的重复数据库查询的快速访问。ComposerSearch.jsp提示输入国籍和世纪，然后搜索LyricNoye做曲者数据库并显示结果。它包含connect.jsp进行实际的连接和会话捆绑工作。

```
<%@ page session="true" %>
<%@ page import="jspcr.jdbc.*" %>
<%@ page import="java.sql.*" %>
<%
   // Get form parameters or use defaults

   String nationality = request.getParameter("nationality");
   if (nationality == null)
      nationality = "";

   String yearRange = request.getParameter("yearRange");
   if (yearRange == null)
      yearRange = "1901-2000";
%>
<HTML>
<HEAD>
<TITLE>Composer Search</TITLE>
</HEAD>
```

精彩好书见www.book118.com


```
<BODY>
<CENTER>
<H3>Composer Search</H3>
<FORM METHOD="POST">
<B>Nationality:</B>
<INPUT TYPE="TEXT" NAME="nationality" VALUE="<%= nationality %>">
<B>Century:</B>
<SELECT NAME="yearRange">
<%
   // Create the century option list

   for (int century = 16; century <= 20; century++) {
      int fromYear = (century - 1) * 100 + 1;
      int toYear = century * 100;
      StringBuffer sb = new StringBuffer();
      sb.append("<OPTION");
      if (yearRange.startsWith("" + fromYear))
         sb.append(" SELECTED");
         sb.append(" VALUE='");
         sb.append(fromYear);
         sb.append("-");
         sb.append(toYear);
         sb.append("'>");
         sb.append(century);
         sb.append("th Century</OPTION>");
         out.println(sb);
   }
%>
</SELECT>
<INPUT TYPE="SUBMIT" VALUE="Search">
</FORM>
<%
   // If values were entered in the form, display results

   if (!nationality.equals("")) {
%>

<%-- Get the bound connection --%>

<jsp:include page="connect.jsp" flush="true"/>

<TABLE BORDER=0 CELLPADDING=1 CELLSPACING=1>
<%

      BoundConnection bcon = (BoundConnection)
         session.getAttribute("bcon");
```

精彩好书见www.book118.com


```
        Connection con = bcon.getConnection();
        String sql = ""
           + " SELECT lname, fname, born, died"
           + " FROM composers"
           + " WHERE nationality = ?"
           + " AND ((born between ? and ?)"
           + " OR (died between ? and ?))"
           + " ORDER BY born, lname"
           ;
        PreparedStatement pstmt = con.prepareStatement(sql);

        int fromYear = Integer.parseInt(yearRange.substring(0, 4));
        int toYear = Integer.parseInt(yearRange.substring(5));

        pstmt.setString(1, nationality);
        pstmt.setInt(2, fromYear);
        pstmt.setInt(3, toYear);
        pstmt.setInt(4, fromYear);
        pstmt.setInt(5, toYear);
        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
           String lname = rs.getString(1);
           String fname = rs.getString(2);
           int born = rs.getInt(3);
           int died = rs.getInt(4);
%>
<TR>
   <TD><%= fname %> <%= lname %></TD>
   <TD><%= born %>-<%= died %></TD>
</TR>
<%
        }
        rs.close();
        pstmt.close();
%>
</TABLE>
<%
   }
%>
</CENTER>
</BODY>
</HTML>
```

为了访问会话驻留的连接，所有应用都必须检索其bcon会话属性，并将其置入一个BoundConnection，调用其getConnection（）方法。注意，没有必要显式关闭连接。此工作当3分钟到达时仍没有来自客户端的请求后自动完成。结果Web页面如图14-5所示，可以用于只有第

精彩好书见www.book118.com

一个查询需要的新连接的重复查询。

在System.err记录中的跟踪入口显示了HTTP会话中BoundConnection的生命期。

```
TRACE: 07:55:00 session 8720188469 last accessed time 07:55:00
TRACE: 07:55:00 session 8720188469 connection bound
TRACE: 07:59:38 session 8720188469 last accessed time 07:56:38
TRACE: 07:59:38 session 8720188469 connection unbound
```

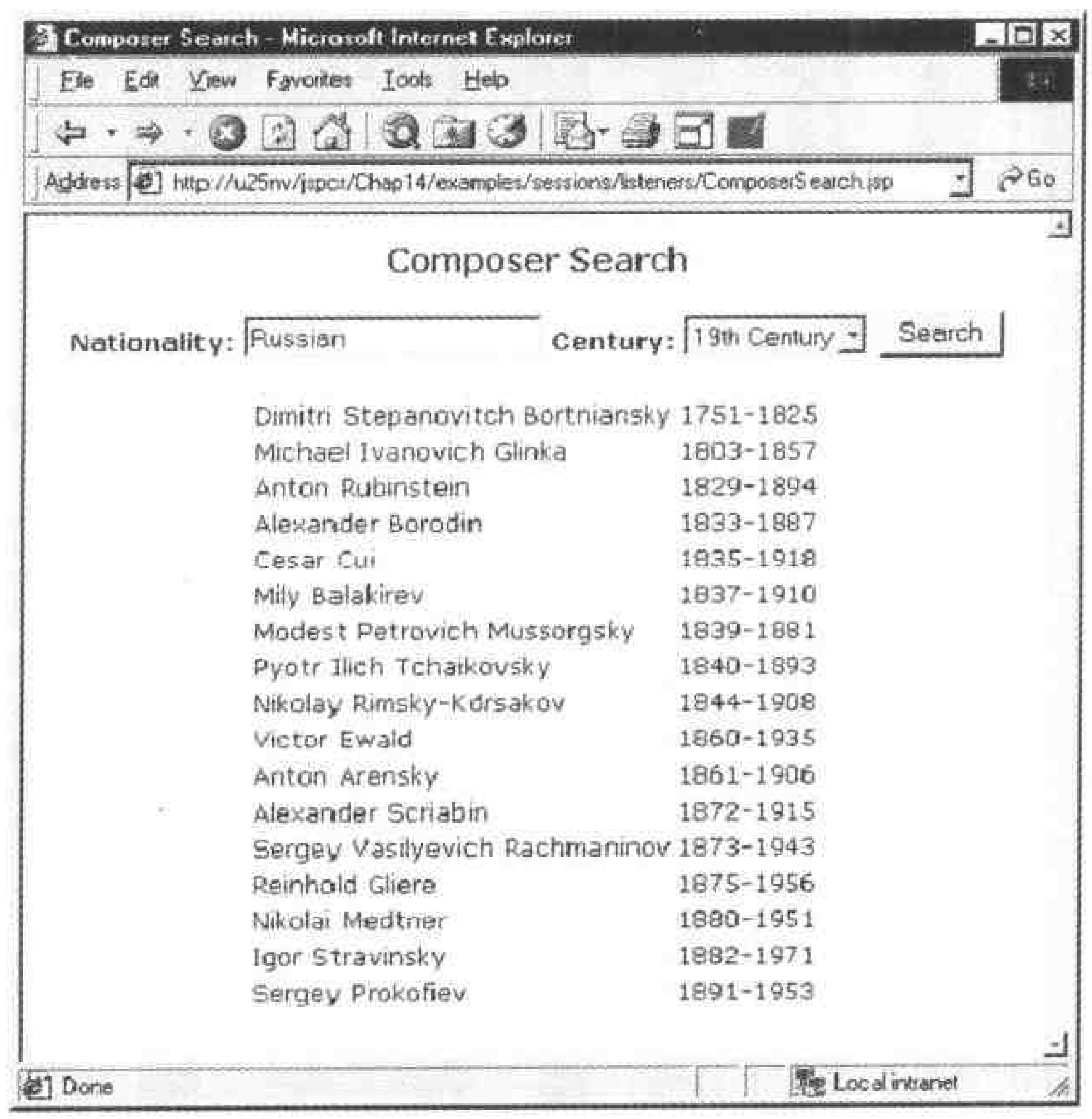


图14-5　使用BoundConnection的Web数据库查询

连接在07：55：00捆绑到会话并被使用一或多次，最后一次时间是07：56：38。3分钟之后，即07：59：38，会话超时，释放BoundConnection对象。这反过来使得valueUnBound（）方法被调用，它关闭了底层连接。

14.3　线程管理

Servlet和JSP页面比原来的服务器端技术有很大的优势。因为它们被载入内存，在多线程环境下作为单一实例运行。此方式还可以权衡。然而，多线程模型引入了简单模型不存在困难性。例如，如果一个servlet有实例变量，可以同时从不同请求中对其进行访问。如果两个请求写变量，

精彩好书见www.book118.com

则其值将不可预测。

幸运的是，因为servlet引擎是用Java编写的，它可以利用Java对多线程应用内置的支持。因此，下面学习一些基本的线程概念，检验两个servlet线程模型，再考虑一个高效的多线程应用。

线程概念

一个线程是具有自己的栈和程序计数的控制的单顺序流。使用多线程的程序表现为同时做多件事情。在同一进程中的同一时间，一个线程可以独立于其他线程进行操作，并共享所有的进程对象。

Web服务器本身就是可用线程的例子。一个简单的Web服务器操作如下：

1) 创建一个ServerSocket，调用其accept（）方法等待HTTP客户端请求。

2) 取得accept（）方法返回的客户端Socket对象，启动一个单独的线程处理其请求。

3) 返回到步骤1，在上一请求正被其他线程处理的同时接受更多的请求。

通常Java（不只是在JSP页面中）创建和使用多线程很容易。语言和类库理论上都是构建在带有线程的基础上。java.lang.Object，所有对象的最终基类，具有同步线程操作的方法，被每一个Java对象所继承。

线程由java.lang.Thead类的实例表示。一个新的线程对象直到其start（）方法被调用时才会与底层操作系统线程相关。start（）方法允许在启动线程前设置其字符特性（名字、优先权等）。调用start（）后，Java虚拟机创建一个操作系统线程，此线程开始执行线程的run（）方法。一个线程不断运行直到其run（）方法返回或调用其interrupt（）方法。

1) 创建可启动线程

启动新线程有3种技术可以利用。第一个是包含Thread子集并覆盖其run（）方法。然后此类中的对象就可以被分别创建和启动。下面的Example1.java阐述了这种技术。它使用Thread的子类调用CounterThread计数到8，打印线程名和每一循环的时间并等待循环之间随机时间长。

```
import java.text.*;
import java.util.*;

/**
* A class that demonstrates simple multithreading
*/
public class Example1
{
   public static void main(String[] args)
   {
      /**
      * Create, name, and start two counter threads
      */

      Thread t1 = new CounterThread();
      t1.setName("A");
      t1.start();
```

精彩好书见www.book118.com


```
        Thread t2 = new CounterThread();
        t2.setName("B");
        t2.start();
    }
}

/**
* A thread that counts to eight, waiting
* a random length of time between iterations.
*/
class CounterThread extends Thread
{
    /**
    * Date format used in message. Includes milliseconds.
    */

    public static final SimpleDateFormat FMT
        = new SimpleDateFormat("hh:mm:ss.SSS aa");

    /**
    * Starts the run method in a new thread
    */
    public void start()
    {
        System.out.println("Starting " + getName());
        super.start();
    }

    /**
    * Where the counter loop takes place.
    */
    public void run()
    {
        for (int i = 0; i < 8; i++) {
            try {
                sleep((long) (Math.random() * 500 + 100));
            }
            catch (InterruptedException e) {
                break;
            }
            System.out.println
                (FMT.format(new Date())
                + " Thread " + getName()
                + ": Count = " + i);
        }
```

精彩好书见www.book118.com


```
        System.out.println("Leaving " + getName());
    }
}
```

代码中主要行是两个CounterThread实例，A和B。下面程序的输出显示两个线程同时执行并在其循环中会偶尔交迭。

```
Starting A
Starting B
09:55:40.465 PM Thread B: Count = 0
09:55:40.545 PM Thread A: Count = 0
09:55:40.615 PM Thread B: Count = 1
09:55:40.846 PM Thread A: Count = 1
09:55:41.056 PM Thread B: Count = 2
09:55:41.346 PM Thread B: Count = 3
09:55:41.366 PM Thread A: Count = 2
09:55:41.687 PM Thread A: Count = 3
09:55:41.717 PM Thread B: Count = 4
09:55:41.847 PM Thread B: Count = 5
09:55:41.967 PM Thread B: Count = 6
09:55:42.017 PM Thread A: Count = 4
09:55:42.137 PM Thread B: Count = 7
Leaving B
09:55:42.268 PM Thread A: Count = 5
09:55:42.428 PM Thread A: Count = 6
09:55:42.848 PM Thread A: Count = 7
Leaving A
```

第二种技术拥有一个实现Runnable接口的类。此时，该类必须提供其自己的run（）方法并创建一个Thread对象执行实际工作。该类必须在Thread构造器中向自己传递一个引用（使用this变量）。下面Example2.java在操作中显示了这种技术。在模拟Example1后，它创建两个线程并向每一个传递其this变量。注意，两个线程可以同时运行同样的run（）方法：

```
import java.text.*;
import java.util.*;

/**
 * A class that demonstrates simple multithreading
 * using the Runnable interface.
 */
public class Example2 implements Runnable
{
    public static void main(String[] args)
    {
        new Example2();
    }
```

精彩好书见www.book118.com


```
    public Example2()
    {
        /**
        * Start two Runnable threads each using this run method.
        */

        Thread t1 = new Thread(this);
        t1.setName("A");
        t1.start();

        Thread t2 = new Thread(this);
        t2.setName("B");
        t2.start();
    }

    /**
    * Date format used in message. Includes miliseconds.
    */

    public static final SimpleDateFormat FMT
        = new SimpleDateFormat("hh:mm:ss.SSS aa");

    /**
    * Where the counter loop takes place.
    */
    public void run()
    {
        Thread t = Thread.currentThread();
        System.out.println("Starting " + t.getName());
        for (int i = 0; i < 8; i++) {
            try {
                t.sleep((long) (Math.random() * 500 + 100));
            }
            catch (InterruptedException e) {
                break;
            }
            System.out.println
                (FMT.format(new Date())
                + " Thread " + t.getName()
                + ": Count = " + i);
        }
        System.out.println("Leaving " + t.getName());
    }
}
```

Example2输出类似于Example1的输出：

精彩好书见www.book118.com


```
Starting main
Starting A
10:10:54.269 PM Thread A: Count = 0
10:10:54.299 PM Thread main: Count = 0
10:10:54.620 PM Thread main: Count = 1
10:10:54.690 PM Thread A: Count = 1
10:10:54.980 PM Thread main: Count = 2
10:10:55.180 PM Thread A: Count = 2
10:10:55.351 PM Thread A: Count = 3
10:10:55.461 PM Thread main: Count = 3
10:10:55.671 PM Thread A: Count = 4
10:10:55.811 PM Thread A: Count = 5
10:10:56.042 PM Thread main: Count = 4
10:10:56.272 PM Thread main: Count = 5
10:10:56.382 PM Thread A: Count = 6
10:10:56.753 PM Thread main: Count = 6
10:10:56.773 PM Thread A: Count = 7
Leaving A
10:10:56.943 PM Thread main: Count = 7
Leaving main
```

使用Runnable接口的一个弊端是它只有一个run（）方法的类，因此可能只执行一种后台操作，而无论其创建了多少进程。例如，一个进行动画模拟并侦听一个套接字或输入流的应用就不能通过实现Runnable完成。

Java 2引入了启动进程的第3种技术，java.util.Timer和java.util.TimerTask类。Timer类充作延迟或重复任务的调度程序。这些任务必须扩展TimerTask类并提供一个run（）方法。任务使用Timer.scheduleTask（）方法几种形式中的一种执行调度。与其他两种方法不同，TimerTask 的run（）方法正常情况并不包含一个执行循环，因为Timer可以自动调度重复任务的执行。下面Example3.java显示了使用Timer和TimerTask的例子。

```
import java.text.*;
import java.util.*;

/**
* A class that demonstrates simple multithreading
* using <CODE>java.util.Timer</CODE>
*/
public class Example3
{
   public static void main(String[] args)
   {
      /**
      * Create a timer to control the timer tasks
      */
      Timer timer = new Timer();
```

精彩好书见www.book118.com


```
        /**
        * Create two timer tasks and schedule
        * their execution at half-second intervals,
        * delaying the second one's start by 250 ms
        */
        TimerTask t1 = new CounterTimerTask("A");
        timer.schedule(t1, 0, 500);

        TimerTask t2 = new CounterTimerTask("B");
        timer.schedule(t2, 250, 500);
    }
}

/**
* A TimerTask that counts to eight
*/
class CounterTimerTask extends TimerTask
{
    /**
    * Date format used in message. Includes milliseconds.
    */

    public static final SimpleDateFormat FMT
        = new SimpleDateFormat("hh:mm:ss.SSS aa");

    private String name;
    private int counter;

    public CounterTimerTask(String name)
    {
        this.name = name;
        this.counter = 0;
    }

    /**
    * Where the counter loop takes place.
    */
    public void run()
    {
        if (counter == 0)
            System.out.println("Starting " + name);

        System.out.println
            (FMT.format(new Date())
            + " Thread " + name
```

精彩好书见www.book118.com


```
            + ": Count = " + counter);

        counter++;
        if (counter >= 8) {
            System.out.println("Leaving " + name);
            cancel();
        }
    }
}
```

CounterTimerTask对象跟踪其被调用的时间数并当其循环极限时调用其cancel ( ) 方法。因为Example3对每一任务使用固定的调度，计数信息在大约四分之一秒间隔内不断变化：

```
Starting A
10:57:44.209 PM Thread A: Count = 0
Starting B
10:57:44.460 PM Thread B: Count = 0
10:57:44.710 PM Thread A: Count = 1
10:57:44.961 PM Thread B: Count = 1
10:57:45.211 PM Thread A: Count = 2
10:57:45.461 PM Thread B: Count = 2
10:57:45.712 PM Thread A: Count = 3
10:57:45.962 PM Thread B: Count = 3
10:57:46.212 PM Thread A: Count = 4
10:57:46.463 PM Thread B: Count = 4
10:57:46.713 PM Thread A: Count = 5
10:57:46.963 PM Thread B: Count = 5
10:57:47.214 PM Thread A: Count = 6
10:57:47.464 PM Thread B: Count = 6
10:57:47.715 PM Thread A: Count = 7
Leaving A
10:57:47.965 PM Thread B: Count = 7
Leaving B
```

2) 同步线程

多线程应用经常有一些操作需要一次只被一个线程执行或是需要多线程协同工作。为此，必须存在保护关键代码端的一种方式，这样两个线程就不会同时运行它们。

为理解其中缘由，考虑下面程序的例子。它审查一个票据应用的发票号。使用的最后一个发票号保存在一个文本文件中。一个新的发票号在读取此文件，向其加入一个发票号并写回磁盘。程序开始有5个线程模拟多在线用户在随机时间访问发票号过程。此示例中的发票处理包括简单打印线程名及其已经设置的发票号。看下面代码是否存在故障：

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

精彩好书见www.book118.com


```
/**
* An illustration of a thread synchronization problem
*/
public class SynchTest implements Runnable
{
   public static void main(String args[])
   {
      new SynchTest();
   }

   /**
   * Creates a new SynchTest object that starts
   * five invoice handling threads.
   */
   public SynchTest()
   {
      Thread[] threads = {
         new Thread(this, "A"),
         new Thread(this, "B"),
         new Thread(this, "C"),
         new Thread(this, "D"),
         new Thread(this, "E"),
      };
      for (int i = 0; i < threads.length; i++)
         threads[i].start();
   }
   /**
   * Simulates handling ten invoices. This method
   * will be run by each of the five threads.
   */
   public void run()
   {
      try {
         for (int i = 0; i < 10; i++) {
            handleInvoice();
            Thread.sleep((long) (Math.random()*500));
         }
      }
      catch (InterruptedException ignore) {
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }

   /**
```

精彩好书见www.book118.com


```
     * The invoice handling method (with a subtle bug)
     */
    public void handleInvoice()
       throws IOException
    {
       Thread t = Thread.currentThread();

       // Get the last used invoice number from invoice.dat

       BufferedReader in =
          new BufferedReader(
          new FileReader("invoice.dat"));
       int invoiceNumber = Integer.parseInt(in.readLine());
       in.close();

       // Add 1 to get the current invoice number

       invoiceNumber++;
       System.out.println
          (t.getName() + " handles invoice " + invoiceNumber);

       // Update the invoice number

       PrintWriter out =
          new PrintWriter(
          new FileWriter("invoice.dat"));
       out.println(invoiceNumber);
       out.flush();
       out.close();
    }
}
```

程序可能无故障运行许多次，对每一线程设置了连续的发票号。但过一会，输出可能如下：

```
A handles invoice 68401
B handles invoice 68402
C handles invoice 68403
D handles invoice 68404
E handles invoice 68405
E handles invoice 68406
D handles invoice 68407
A handles invoice 68408
D handles invoice 68409
B handles invoice 68410
B handles invoice 68411
E handles invoice 68412
C handles invoice 68412
```

精彩好书见www.book118.com


```
B handles invoice 68413
D handles invoice 68414
E handles invoice 68415
B handles invoice 68416
B handles invoice 68417
A handles invoice 68418
```

发票号68412在列表中出现两次，同时被线程E和C设置，发生什么事件了呢？

问题在于：从读取发票号到其被重新写回invoice.dat文件期间，可能有另一个线程执行同样的方法读取文件并取得旧的号码。此线程然后可能增加它并修改文件，然后它可能就会拥有一个重复的发票号。

为防止这一点，Java提供在所有线程注意的一个对象上使用排它锁的方式，这种锁机制称为同步，由关键字synchronized激活。代码中每一块均可使用下列语法同步：

```
synchronized (object) {
// code to be synchronized
}
```

这里object是任意对象的引用。整个方法的同步可以通过使用synchronized关键字作为一种方法修改器实现，例如：

```
public synchronized void myMethod() {
   // code to be synchronized
}
```

其功能与下面代码等价：

```
public void myMethod() {
   synchronized(this) {
      // code to be synchronized
   }
}
```

当一个线程遇到synchronized块，它首先试图得到指定对象上的锁。如果该线程成功，即执行块并释放锁。如果该线程不能得到锁，则等待直到锁可利用，再获得锁本身，执行块并释放锁。Java虚拟机确保这些操作一次只被一个线程执行。

在发票处理例子中，重复发票问题可以通过同步handleInvoice（）方法消除[1]：

```
public synchronized void handleInvoice() throws IOException
{
// Read the file, increment the invoice number,
// and update the file.
...
}
```

1 当然，同步对运行在Java虚拟机外的其他Java类和某些进程修改invoice.dat文件不起作用。该例子假定你对该文件具有排它性控制。

精彩好书见www.book118.com

为了提高性能，重要的是不要同步任何多余的代码。因为这将强迫线程使单个文件一直走过同步段。不必同步整个handleInvoice（），而只是从文件被打开读取到写完关闭的代码就可以了。

14.4 servlet线程模型

servlet API利用了Java内置的对多线程的支持以确保可响应的请求处理和高效吞吐率。实现过程中，它提供了线程使用方式的灵活性。请求被发送到一个或多个线程的过程称为servlet线程模型。可以选择以下两种模型：

- 多个线程运行单个servlet实例，这是缺省线程模型。
- 多实例，每个都运行在自己的线程内。这称为单线程模型。

下面考虑每一模型操作的含义。

14.4.1 缺省线程模型

在缺省模型中，只载入servlet（或JSP）的单一实例[1]。servlet引擎维护线程池，当请求到达时将其设置到请求。每个线程运行适当的服务方法，典型为doget（）或dopost（）。在高峰活动期，许多请求可能通过同一个servlet方法在同时运行，但因为每个线程都有其自己的指令指针和本地变量栈，请求间不会发生冲突。图14-6阐述了缺省模型，显示了由3个线程处理的3个请求。

缺省模型提供好的吞吐率，但存在一些限制。因为只有一个servlet实例，任意实例变量只存在一个复本。如果考虑不谨慎，代码允许写入变量，一个线程覆盖了另一个线程所需的值。例如，在图14-6中，请求1、2、3同时运行，如果它们都在doGet（）方法中写入一个实例变量，再读取它，写入和读取就可能发生重叠。还有，如果doGet（）或doPost（）方法调用子过程，它必须将所有必要对象作为参数传递，因为从其被写入直到子过程读取它们为止，不能依赖实例变量保留其值。

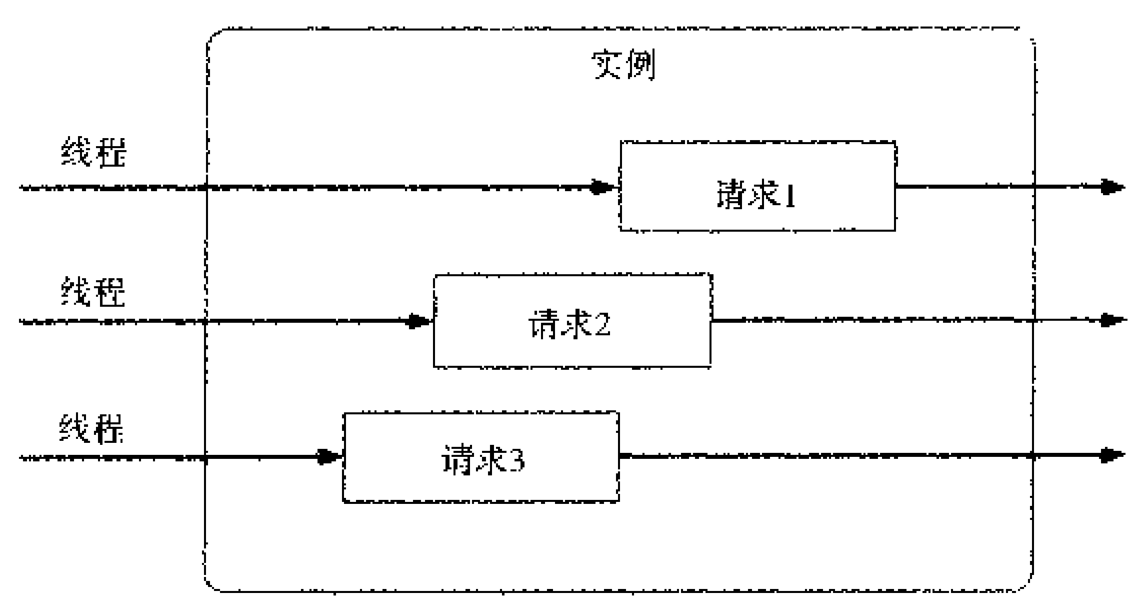


图14-6 缺省线程模型——带有多线程的一个实例

1 技术上讲，一个实例对应一个servlet名字。几个servlet名字可以和web.xml发布描述器中的相同的servlet类相连接。细节见第18章。

精彩好书见www.book118.com

由于此原因，尽量避免使用实例变量，除非它们是只读的。这很像一个限制，但它只是一个不同的观点。毕竟工作的真正单元是请求，而不是servlet实例。任何类型的对象均可以完全线程安全的方式保存为请求属性：

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

        ...
        openConnection(request);
        runQuery(request);
        ...
}

public void openConnection(HttpServletRequest request)
    throws SQLException
{
    request.setAttribute
        ("connection", DriverManager.getConnection(...));
}
public void runQuery(HttpServletRequest request)
    throws SQLException
{
    Connection con = (Connection)
        request.getAttribute("connection");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT ...");
    request.setAttribute("resultSet", rs);
}
```

同时可以在一个servlet方法中同步关键代码段，但必须小心以避免同步太多，对性能造成负面影响。

14.4.2 单线程模型

缺省模型的替代品是单线程模型。在此环境下，servlet引擎确保一次只有一个请求运行servlet实例的服务方法。为使用此模型，servlet必须实现SingleThreadModel接口。此接口中没有方法，它只是将servlet标记为需要此线程处理。在JSP页面中，利用page伪指令选择此模型：

```
<%@ page isThreadSafe="false" %>
```

这使得生成的servlet指定其实现SingleThreadModel。

一次只有一个线程可以执行单线程servlet的doGet（）或doPost（）方法。这意味着实例变量是线程安全的。但servlet引擎可在满足一定性能的前提下随意创建多个servlet实例。此操作模

精彩好书见www.book118.com

型的阐述如图14-7所示。它显示请求3运行前一直等待直到请求1全部完成，但请求2像其他请求一样同时运行在一个不同实例中。

SingleThreadModel是混乱的根源。可以发现粘贴到Java新闻组的信息说明网络或数据库连接没有正确分离，尽管实际上它们使用在实现SingleThreadModel的一个servlet中。原因很简单：确保线程安全性需要做的是对servlet实例本身。但是，因为存在多个实例，外部资源的同时访问并不受保护。

单线程模型的优点很少。如果给定计划和明智的同步策略，通常缺省模型是更好的选择。

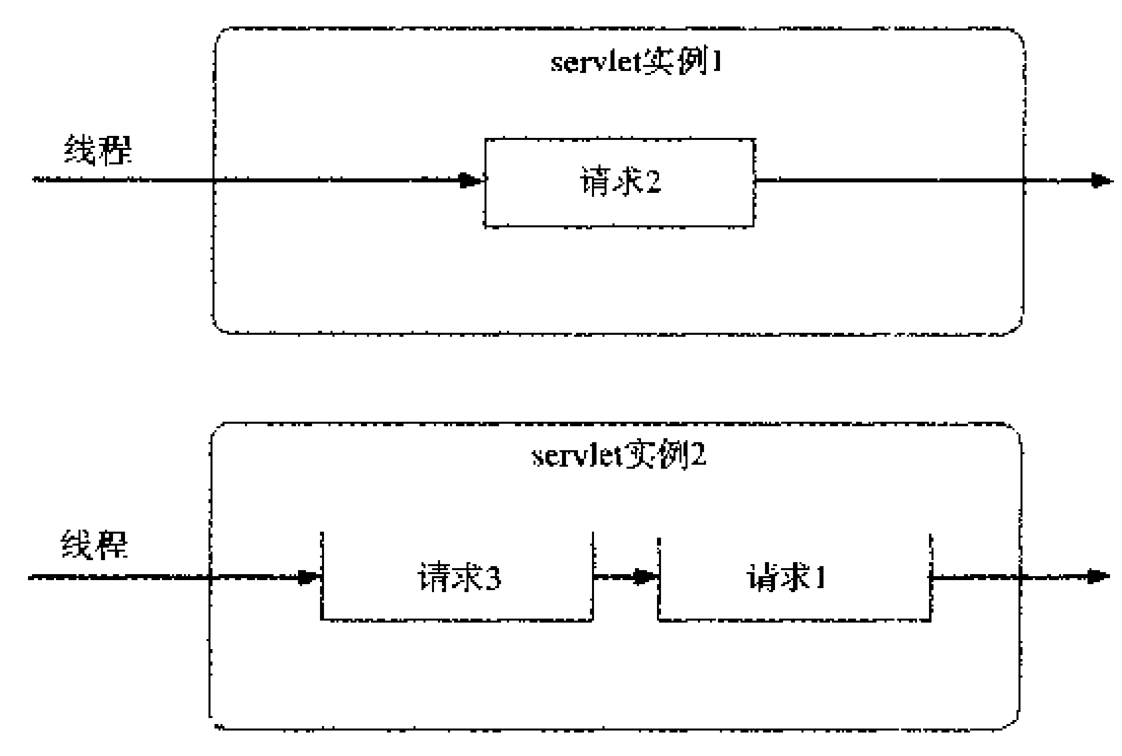


图14-7 在单线程模型中多个servlet实例

14.5 多线程应用

一些多线程特别有用的存在于服务器端环境。其中一个是自动开始的后台进程，并不捆绑到任何特定请求，类似于一个Unix例程或一个Windows NT服务。这可以在servlet环境中使用下列技术实现：

1) init ( ) 方法启动一个线程执行所需任务。包括打开一个套接字服务于请求或周期性地读取具有动态修改数据的Web页面，如股票报价或新闻标题。在servlet上下文中保持一个线程的引用。

2) 对命令，管理任务和状态报告严格使用doGet ( ) 方法。通常需要给出可以关闭或重启后台线程的命令。

3) 在destroy ( ) 方法中，释放任意使用的资源后关闭线程。

servlet (JSP) 可设计为发布描述器中的load-on-startup。，因此servlet引擎一运行它就开始运行，而无需注册用户。因为servlet引擎本身典型情况可作为一个例程或服务运行，因此可以将这样的进程写为Java servlet。

多线程另一点有益的用法是处理长时间运行的请求，如复杂的数据分析或其他类似工程的请求。如果一个客户端请求一个时间负载相当重的服务器进程，则客户端和servlet引擎都将等待

精彩好书见www.book118.com

直到其完成。也还有其他情况存在。

考虑假脱机打印的工作方式。一个用户可以在一个会话处理应用中点击一个打印按钮，当打印请求排队时可能只是轻微停顿一下。通常接收请求的打印机信息会出现，可能是一个标识打印请求的作业ID，然后缓冲的输出排队等待打印机可以利用。其他的一些任务可能包括监视打印队列状态，保存和释放作业，改变优先权，取消作业等等。

一个JSP页面可进行类似操作。不是在当前线程中运行一个复杂任务，它可以通过启动一个后台线程运行，并将线程的引用保存为一个会话属性。用户可以收到请求已经排队和正在被服务的通告。JSP页面也可以给出显示请求状态和用户可以保存、释放、取消请求的界面。当请求完成时，JSP页面可以给出查看结果的一个超级链接。被强大的JSP页面处理的长时间的请求甚至会使其结果被电邮给用户。

带有状态信息的长时间运行请求

此技术的一个变体可向用户提供一个状态条以指出请求正被处理，请求完成时则以结果替换它。此技术的关键是使用一个后台会话范围内的线程执行工作并使用<META HTTP-EQUIV="REFRESH"> HTML标签使得Web浏览器自动监视状态。

在下面例子中，用户注册，利用费时的模拟数据库操作请求验证。当处理验证请求时，用户会看到“Authenticating, please wait...”信息，请求完成时，此信息替换为验证结果。同时servlet引擎可自由处理其他请求而无需等待验证请求结束。

在此会话中客户端和服务器之间的基本转换如下：

```
Client: Please authenticate me. UserID=MyUserID, password=MyPassword
Server: OK, call me back in two seconds for status.
Client: (two seconds later) Are you done?
Server: No, call me back in two seconds for status
Client: (two seconds later) Are you done?
Server: No, call me back in two seconds for status
Client: (two seconds later) Are you done?
Server: Yes, you are authenticated (or not).
```

用户不必手工请求被修改的状态。HTML给出周期性修改页面的客户端自动方式。

```
<META HTTP-EQUIV="REFRESH" CONTENT="seconds; URL=url">
```

在一个HTML文档中此<META>标签的出现使得浏览器等待指定秒数，然后重定向到指定URL。此特性通常用于指出一个Web站点已经移动，用户将被自动导向新位置。

以下JSP使用一个模拟数据库验证延迟的后台工作线程实现了此协议：

```
<%--
  Authenticator.jsp

    A JSP page that displays status messages during a
    long-running request and does not tie up server
    resources waiting for the request to complete.
```

精彩好书见www.book118.com


```
--%>
<%

   // See if there is an authentication worker thread running

   WorkerThread worker = (WorkerThread)
      session.getAttribute("worker");

   // If not, create a new one and start the authentication

   if (worker == null) {
      String userID = request.getParameter("userID");
      String password = request.getParameter("password");
      worker = new WorkerThread(userID, password);
      session.setAttribute("worker", worker);
   }

   // Now display either the "please wait" screen
   // or the "user authenticated" screen

   if (!worker.isDone()) {
      String url = HttpUtils.getRequestURL(request).toString();
      url = response.encodeURL(url);
%>
<HTML>
<HEAD>
<TITLE>Please Wait</TITLE>
<META HTTP-EQUIV="REFRESH" CONTENT="2; URL=<%= url %>">
</HEAD>
<BODY>
Authenticating, please wait...
</BODY>
</HTML>
<%
   }
   else {
%>
<HTML>
<HEAD><TITLE>Done</TITLE></HEAD>
<BODY>
Authentication complete.
<%= worker.isAuthenticated() ? " You pass!" : " You fail!" %>
</BODY>
</HTML>
<%
      // Done with worker
```

精彩好书见www.book118.com


```
         session.invalidate();
      }
   %>
   <%!
   /**
   * A background thread that performs a potentially
   * long-running task (authentication from a database).
   */
   public class WorkerThread implements Runnable
   {
      private boolean done;
      private boolean authenticated;
      private Thread kicker;

      public WorkerThread(String userID, String password)
      {
         done = false;
         authenticated = false;
         kicker = new Thread(this);
         kicker.start();
      }

      public boolean isDone()
      {
         return done;
      }

      public boolean isAuthenticated()
      {
         return authenticated;
      }

      public void run()
      {
         // Do the work here

         try {

            // Pretend to do something that takes five seconds

            for (int i = 0; i < 5; i++)
               Thread.sleep(1000);

            // Randomly authenticate 80% of all users
```

精彩好书见www.book118.com


```
            authenticated   (Math.random() > 0.2);

             / We are done

            done   true;
         }
         catch (InterruptedException ignore) {}
         finally {
            kicker = null;
         }
      }
   }
   %>
```

14.6 应用考虑

JSP环境提供将应用字符特性映射到HTTP环境的大量的方案，主要考虑的是对象范畴，也就是其中属性有效的时间周期。页面上下文定义了四种范畴：

- 页面
- 请求
- 会话
- 应用

每一范畴都有其生命期并可在其中存储属性。特定范畴内的对象对同一servlet上下文中的JSP页面和servlet均可访问。开发商的任务就是选择匹配对象使用需求的对象范畴。

页面范畴等价于单一JSP页面中的_jspService（）方法的生命期。例如，用户ID字符串可如下给定其页面范畴：

```
pageContext.setAttribute
   ("userID", userID, PageContext.PAGE_SCOPE);
```

或简单为：

```
pageContext.setAttribute("userID", userID);
```

存在检索此对象相应的getAttribute（）方法。

当对象可简单地作为Java变量访问时为什么还要在页面上下文中使用页面范畴存储此对象呢？页面的主要上下文是一个JSP定制标签，它使用页面上下文在标签处理器和JSP页面之间进行通信。JSP定制标签的详细内容查看第11章。

请求范畴几乎与页面范畴相同，但它包含了由<jsp:include>或<jsp:forward>调用的其他JSP页面或servlet。在请求中属性可如下直接设置：

```
request.setAttribute("userID", userID);
```

或利用页面上下文：

```
pageContext.setAttribute
```

精彩好书见www.book118.com


```
("userID", userID, PageContext.REQUEST_SCOPE);
```

两个方法调用的影响是一样的；pageContext.setAttribute（ ）方法简单调用request.setAttribute（ ）。请求范畴适用于与单一请求相关的对象，可在servlet中设置并用于RequestDispatcher前导请求的JSP页面。

会话范畴可被标识有同样会话ID并与具有此ID的一个激活状态的HttpSession相关的多个请求所使用。在一个会话对象中属性可直接设置如下：

```
session.setAttribute("userID", userID);
```

或利用页面上下文

```
pageContext.setAttribute
("userID", userID, PageContext.SESSION_SCOPE);
```

当下面三种需求出现时，会话范畴可适用：

- 应用需要多个HTTP请求。
- 数据需要在请求之间保留。
- 一个或多个服务器端对象必须在请求间保存为一特定状态。

如本章讨论过的，当不需要HTTP会话时，可用隐藏域、URL重写和cookie替代它。

应用范畴是一个Web应用中所有servlet和JSP页面的通用名空间。它在请求间自动持续，不需要会话。在web.xml发布描述器中可使用<context-param>在应用范畴内设置静态初始化参数：

```
<context-param>
   <param-name>jdbc.driver</param-name>
   <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>

<context-param>
   <param-name>jdbc.url</param-name>
   <param-value>jdbc:odbc:composers</param-value>
</context-param>
```

这些静态参数可在servlet或JSP页面中使用servlet上下文的getInitParameter（ ）方法进行检索：

```
String driver = application.getInitParameter("jdbc.driver");
String url = application.getInitParameter("jdbc.url");
```

可使用像其他3种范畴同样的方法在应用范畴内保存和检索对象：

```
pageContext.setAttribute
("userID", userID, PageContext.APPLICATION_SCOPE);
```

或

```
application.setAttribute("userID", userID);
```

应用范畴对需要在请求之间持续的对象，对应用的所有用户均可视的对象或需要在其他servlet和JSP页面之间被共享的对象最有用。

精彩好书见www.book118.com

14.7 小结

Web应用模型并不自动映射到HTTP协议。HTTP是无状态的，不会记住一次请求到下一次请求的客户端的任何信息。Web浏览器环境也引入了复杂性——在应用中，每个页面均依赖于其祖先，但用户可以按随意次序浏览页面并离开应用而没有给出其执行信息。

问题的解决方案是让客户端（浏览器）记住一些细节并在每次进行请求时提醒服务器。这可能包括要求客户端管理所有数据，但实际上更常见的是客户端只记住一个标识，让服务器使用此标识检索数据库数据的其余部分。这种虚拟转换（虚拟是因为不包含持续的连接）通常称为会话。

本章介绍了会话管理的4种技术：

- 在一个HTML窗体中的隐藏域 这是简单用法，只能在一个HTML窗体中被传送。如果用户点击一个超级链接，则隐藏域（因此会话）失效。
- URL重写 包括将会话标识附加到JSP页面生成的所有URL中。从性能考虑，此技术可能很昂贵。
- Cookie 小的已命名数据元素被发送到客户端并当页面再次访问时返回到服务器。Cookie的优点是它们可存在任意长时间，即使是在客户端计算机已经关机之后。缺点是由于私人原因有些用户会关闭Cookie的支持。
- 会话API servlet引擎可以创建HttpSession对象，它作为在同一应用中来自同一客户端的请求之间可持续的已命名对象库。客户端使用cookie或URL重写记住会话ID，servlet引擎判断客户端接受并相应采取这些技术中的哪一种。

可编程关闭会话或依据一个可配置非活动周期使其超时。会话API向对象提供其何时捆绑或从会话中解缚的方式。

因为JSP页面运行于一个纯Java环境，因此可完全访问Java对多线程应用的支持。本章讨论了基本的线程概念，如何创建、启动和同步线程，然后考虑两种servlet线程模型。给出两个使用多线程的Web应用实例。

即使在最初的HTTP协议中并未设计动态内容，协议证明有能力对其进行扩展。建立在此灵活性上，会话API提供了在Web应用环境中进行HTTP工作的框架。

精彩好书见www.book118.com

第15章 JSP和JavaBean

在硬件和电子技术中,通过组件工程使大量的高级特性得以实现。不是从最初的电路开始，过程以一种新颖的方式将测试过的模块集成以创建高层次的功能。基于组件的编程将该思想扩展到软件领域。在Java世界里，即为JavaBean。

本章介绍JavaBean编程模型和bean如何将其属性暴露给使用它们的类。给出了在JSP页面中使用bean的接口。本章最后是一个完整的例子，说明了JSP页面中可定制天气预报bean的操作过程。

15.1 JavaBean是什么

bean的定义很广泛。bean简单说是满足两项需求的Java类：

- 它包含一个0参数构造器。
- 它利用Serializable或Externalizable使之可持续。

由此定义，大多数类都是bean，或经过一点改动可改为bean。运行bean的上下文除了Java虚拟机外不再需要其他环境。这就允许正确购建的bean用于任何Java环境——applet、servlet、JSP页面或单机Java应用。

15.1.1 bean属性

另外，大多数bean都有属性，Properties是bean用来提供读取或/和写入方法的属性。所有对属性的访问都必须通过这些方法实现，底层的数据域（如果存在）是私有的。JavaBean编程模型的一部分用于这些方法的名称约定。除非通过一个BeanInfo类进行特别指定，属性的读方法是名为get <PropertyName>（ ）的公有方法。这里<PropertyName>是第一个字母转换为大写的属性名。类似的，如果存在，写方法名为set <PropertyName>（ ）。

下面例子是名为Mortgage的JavaBean。它封装了描述抵押借款的参数：

```
package jspcr.beans.mortgage;

import java.io.*;

public class Mortgage implements Serializable
{
   private double principal;
   private double rate;
   private int term;

   /**
   * Returns the principal.
```

精彩好书见www.book118.com


```
*/
public double getPrincipal()
{
   return principal;
}

/**
* Sets the principal.
* @param principal the principal.
*/
public void setPrincipal(double principal)
{
   this.principal = principal;
}

/**
*  Returns the annual interest rate
*/
public double getRate()
{
   return rate;
}

/**
* Sets the interest rate.
* @param rate the annual interest rate as a percentage.
*/
public void setRate(double rate)
{
   this.rate = rate;
}

/**
* Returns the term in months.

*/
public int getTerm()
{
   return term;
}

/**
* Sets the term.
* @param term the term in months.
*/
public void setTerm(int term)
{
   this.term = term;
```

精彩好书见www.book118.com


```
    }

    /**
     * Returns the amortization factor, the amount of the
     * monthly payment that will pay all principal and
     * interest within the specified period of time.
     */
    public double getPayment()
    {
       if (rate == 0)
          throw new IllegalArgumentException
          ("No interest rate specified");

       double mrate = rate / 1200.0;

       double fv = Math.pow((1 + mrate), term);
       double numer = principal * mrate * fv;
       double denom = fv - 1.0;
       return round(numer / denom);
    }

    /**
     * Utility method that rounds a currency amount to
     * the nearest 1/100 of the currency unit.
     */
    public static final double round(double x)
    {
       return ((double) ((long) (x * 100.0 + 0.5))) / 100.0;
    }
}
```

Mortgagebean有4个属性：

- principal 借贷钱数。
- rate 表示为百分比的年息。例如百分之6为6.0。
- term 借贷偿还的月数。
- payment 严格只读属性。

注意，前3个属性存在私有数据域，第4个没有，它以命令形式计算。这里强调的是get和set方法的存在性。这些方法是bean向外部显示的惟一渠道。

参与bean状态改变的类可以实现许多EventListener接口中的一个。当一个类注册为事件侦听者，bean中发生相关事件时它收到反馈信息。这使得bean可能互相协作完成大的任务。AWT和Swing GUI体系结构充分利用了这种事件模型。然而，服务器端环境下使用的bean主要倾向于用做属性库，典型情况并不实现对事件侦听者的支持。

15.1.2 持久性

由于上述情形，一个bean必须提供其处于激活状态之间状态持久性的某种方式，甚至是在

精彩好书见www.book118.com

不同的Java虚拟机环境下。实现此功能可通过让bean实现Serializable或Externalizable接口之一。

Serialization指的是将对象转换为可在文件中保存或在网络上传输的字节流的过程。从字节流中序列化对象的相反过程称为deserialization。Java API为此提供ObjectOutputStream和ObjectInputStream类。

一个对象通过使用输出流的writeObject（ ）方法被写入ObjectOutputStream实现其序列化。例如，包含Mortgage bean的程序可如下将其序列化：

```
OutputStream fileOut = new FileOutputStream("mortgage.ser");
ObjectOutputStream objOut = new ObjectOutputStream(fileOut);
objOut.writeObject(mortgageBean);
objOut.flush();
fileOut.close();
```

后来，此程序或其他程序就可以从mortgage.ser文件中使用ObjectInputStream的readObject（ ）方法重新组建Mortgage bean：

```
InputStream fileIn = new FileInputStream("mortgage.ser");
ObjectInputStream objIn = new ObjectInputStream(fileIn);
Mortgage mortgageBean = null;
try {
   mortgageBean = (MortgageBean) objIn.readObject();
}
catch (ClassNotFoundException e) {
   // handle exception
}
```

在这两个代码段中，只序列化了一个对象，它被序列化到一个文件中。序列化多个对象也一样容易，只是对ObjOut.writeObject（ ）的多次调用。只要重组对象的程序知道对象和其类型的正确号码，它就可以通过多次调用objIn.readObject（ ）读取它们。类似的，返回流不必是一个文件——它可能是一个套接字、一个字节数组或一个打孔卡。只要虚拟机支持它们。

注意：Mortgage类不需要对其序列化做任何工作，它只需实现Serializable接口。正常情况，servlet引擎在其被终止时考虑需要存储会话的所有逻辑和应用bean并当其重新启动时恢复它们。

然而，存在溢出情况，并不是所有的对象都被序列化。例如，一个数据库连接不能被放入延迟的动画中并在以后被唤醒。显然的它应处于和数据库管理系统中的相应对象建立通信的状态。同时，线程连接至底层的操作系统线程，不能简单地被分解和重组。这种情况下，包含非序列化对象的对象必须提供其再次被连接或重启的方式。另外，它必须对具有关键字transient的对象声明其变量引用以防止正常序列化过程试图对其进行处理。

包含transient对象的一个类应通过提供具有下述标志的方法对其进行恢复：

```
private void readObject(ObjectInputStream in)
   throws IOException
```

在readObject（ ）方法中，类首先调用defaultReadObject（ ）恢复序列化的数据域，然后执行初始化过渡域所需的逻辑。典型情况下，此恢复过程通过调用的另一方法实现，也可在构造

精彩好书见www.book118.com

器中调用，以避免重复代码。

一个例子对阐述可能会有益处：

```
import java.io.*;
public class CounterBean implements Runnable, Serializable
{
   private transient Thread thread;
   private int count;

   /**
   * Creates a new CounterBean, initializes its count
   * to zero, and starts the counting thread
   */
   public CounterBean()
   {
      count = 0;
      start();
   }

   /**
   * Restores this CounterBean from an object stream
   * and restarts the thread
   */
   private void readObject(ObjectInputStream in)
      throws IOException
   {
      try {

         // Call this first to restore all the
         // non-transient fields

         in.defaultReadObject();
      }
      catch (ClassNotFoundException e) {
         throw new IOException(e.getMessage());
      }

      // Restart the thread

      start();
   }

   /**
   * Provides a means to shut down the thread
   */
   public void interrupt()
   {
      if (thread != null)
         thread.interrupt();
```

精彩好书见www.book118.com


```
    }

    /**
    * Starts the counting thread. This method
    * is called both from the constructor when
    * the object is first created and from the
    * readObject method when the object is deserialized.
    */
    private void start()
    {
        if (thread == null) {
            thread = new Thread(this);
            thread.setPriority(Thread.MIN_PRIORITY);
            thread.start();
        }
    }

    /**
    * Increments and prints the value of the counter
    * every second.
    */
    public void run()
    {
        try {
            for (;;) {
                Thread.sleep(1000);
                count++;
                System.out.println(count);
            }
        }
        catch (InterruptedException e) {}
    }
}
```

CounterBean类使用后台线程每隔一秒打印一个记数值。此类是序列化的，但线程不是。因此存在当类被反序列化时重启线程的逻辑。注意，线程变量使用关键字transient加以声明。该类有两个入口点：其构造器和readObject（）方法，两个入口点均调用start（）创建和启动一个处理记数逻辑的线程。线程继续执行直到它被中断。

使用此记数器bean的程序可以由其构造器对其加以调用、序列化，然后用显示获得其停止信息的线程对其进行反序列化：

```
import java.io.*;

/**
* A class that uses CounterBean. If a serialized
* version exists, it will deserialize that. Otherwise,
* it will create a new CounterBean.
*/
```

精彩好书见www.book118.com


```
public class CBTest
{
   public static void main(String[] args)
      throws IOException, ClassNotFoundException
   {
      CounterBean bean = null;

      // If a serialized version exists, load it

      File file = new File("counter.ser");
      if (file.exists()) {
         System.out.println("Deserializing bean");
         FileInputStream fileIn = new FileInputStream(file);
         ObjectInputStream objIn = new ObjectInputStream(fileIn);
         bean = (CounterBean) objIn.readObject();
         objIn.close();
         fileIn.close();
      }

      // Otherwise, create a new bean

      else {
         System.out.println("Creating new bean");
         bean = new CounterBean();
      }

      // Let the bean run until the user presses the
      // enter key

      BufferedReader in =
         new BufferedReader(
         new InputStreamReader(
         System.in));
      System.out.println("Press Enter to terminate program");
      in.readLine();
      bean.interrupt();

      // Serialize the bean

      System.out.println("Serializing bean");
      FileOutputStream fileOut = new FileOutputStream(file);
      ObjectOutputStream objOut = new ObjectOutputStream(fileOut);
      objOut.writeObject(bean);
      objOut.flush();
      objOut.close();
      fileOut.close();
   }
}
```

除了实现Serializable，bean还可以对其持久性规划实现Externalizable。Externalizable对象使

精彩好书见www.book118.com

用其选择格式处理其本身来自对象流的读和写。它为此必须提供readExternal(ObjectInput)和writeExternal(ObjectOutput)方法。但是，因为Serializable对象可以使用任意所需格式给出其自己的readObject ( ) 方法，Externalizable在这一点没有多少优势，并未广泛使用。

15.2 JSP行为

正如所见，JavaBean也是Java类，因此也可以在使用scriptlet、声明和表达式的JSP页面中被创建和表示。JSP规范对在更高层次上进行脚本化的bean提供特殊支持。此支持包含下述三种标准行为：

<jsp:useBean> 对声明、实例和初始化bean。

<jsp:setProperty> 对设置bean属性。

<jsp:getProperty> 对检索bean属性值。

下面各节分别讨论这些行为。

15.2.1 <jsp:useBean>

<jsp:useBean>标签创建或反序列化一个bean并将之与脚本变量相关。语法如下：

```
<jsp:useBean id="name" scope="scope" typespec />
```

或：

```
<jsp:useBean id="name" scope="scope" typespec >
 <body>
</jsp:useBean>
```

这里name、scope和typespec定义如下：

1. <jsp:useBean> id属性

在id属性中指定名字用作指定范围内bean对象属性名的标识符，在JSP页面内声明为一个Java脚本变量。因为该值是一个脚本变量，大小写敏感，必须符合对标识符的Java命名规则。该值用在<jsp:setProperty>和<jsp:getProperty>行为的name属性中指出行为应用中可能的bean。

2. <jsp:useBean> scope属性

scope指定bean存在的名空间。它与PageContext对象获得的范围相同。可能值为：

- PAGE 对当前JSP页面有效。这是缺省值。
- REQUEST 对JSP页面的其余部分和通过<jsp:forward>或<jsp:include>行为服务于此请求的任意其他资源有效。
- SESSION 在任意JSP页面或HTTP会话中的servlet执行期间有效。
- APPLICATION 在此Web应用中所有JSP页面或servlet中有效（servlet context）。

上述范围内的bean均可使用pageContext变量的getAttribute ( ) 和setAttribute ( ) 方法访问。在request、session或application名空间中的bean可以分别在servlet中使用ServletRequest、HttpSession和ServletContext类的getAttribute ( ) 和setAttribute ( ) 方法访问。

3. <jsp:useBean> 类型规范

精彩好书见www.book118.com

类型规范由class、type和beanName属性的结合组成。这些属性的结合允许灵活实施useBean行为。至少必须指定class和type中的一个。如果指定了class，则不能使用beanName。有效结合方式为：

- 只有type
- 只有class
- type和class
- type和beanName

类型规范允许创建新的bean以及序列化的bean再次使用或已存在bean合并到JSP主框架。深一层功能产生于<jsp:useBean>可以包含JSP代码体这一事实。如果bean在当前请求中新被例示，只有在<jsp:useBean>标签体中的代码（一个或多个<jsp:setProperty>行为和scriptlet）被执行。

<jsp:useBean>标签的准确操作最好用图形来解释。本节中的流线图描述了此标签在前面列出的四种情况中如何被评估。

只指定type属性　bean可以实现几个接口，只有对给定JSP页面中的bean是重要的。所需接口可以在type属性中键入。当只指定type，JSP引擎定义该类型的一个脚本变量，然后在指定范围内寻找名字匹配给定id的一个属性。如果JSP引擎找到一个匹配对象，将该对象置入指定类型并将其值赋给变量。否则，如果不能找到对象，JSP引擎产生溢出InstantiationException。此过程如图15-1所示。

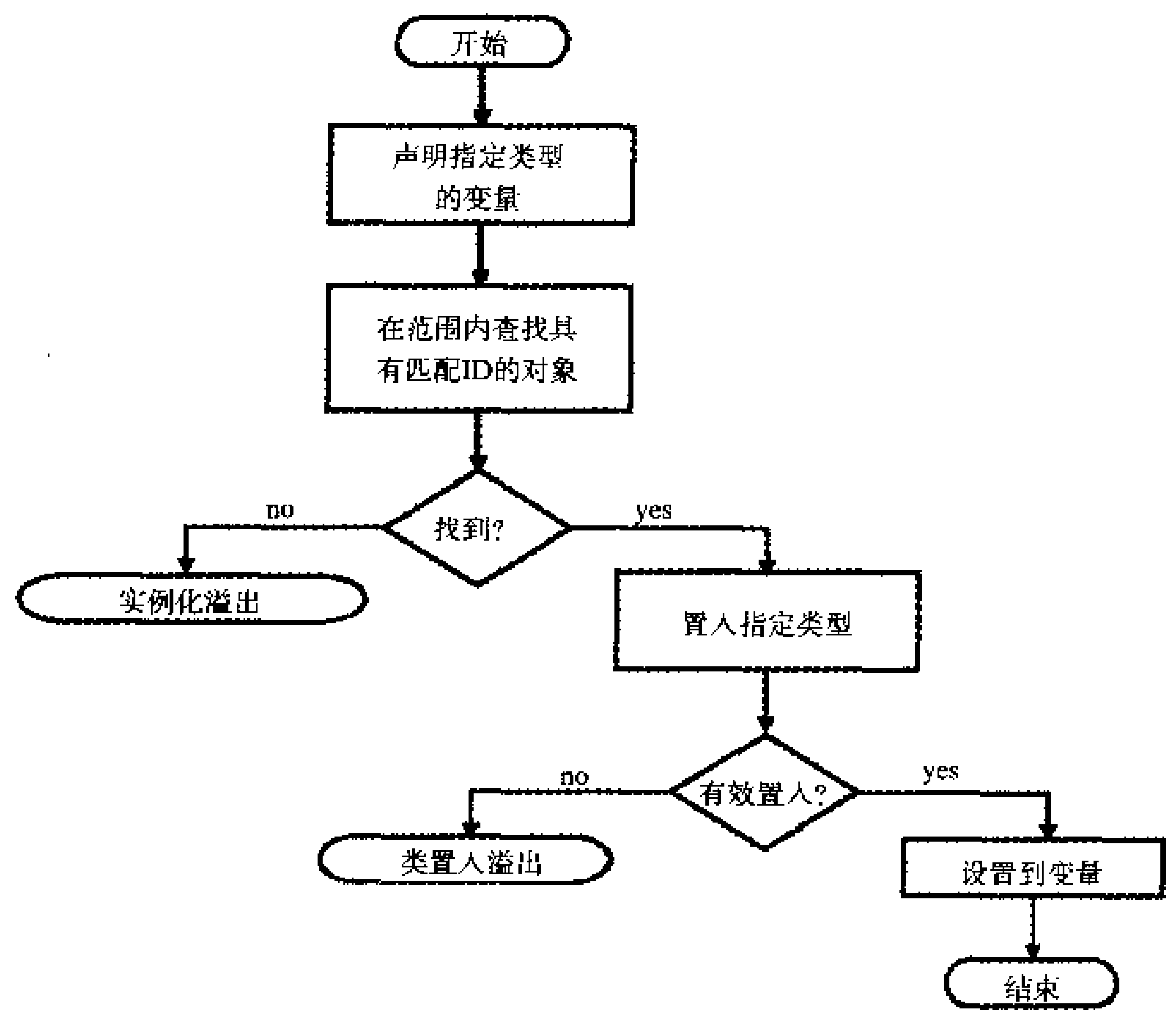


图15-1　当只给定type，<jsp:useBean>过程流线图

精彩好书见www.book118.com

只指定class属性 如果需要一个特殊的bean类，应指定class属性。JSP引擎再次声明该类的变量，并在指定范围内查找具有匹配名字的属性。如果找到，JSP引擎将对象置入指定类，并将其值赋给变量。否则，JSP引擎创建指定类的一个新对象，并将其值赋给变量，在范围内设其为一个属性。如果<jsp:useBean>标签有一个体，那么体被评估。此过程如图15-2所示。

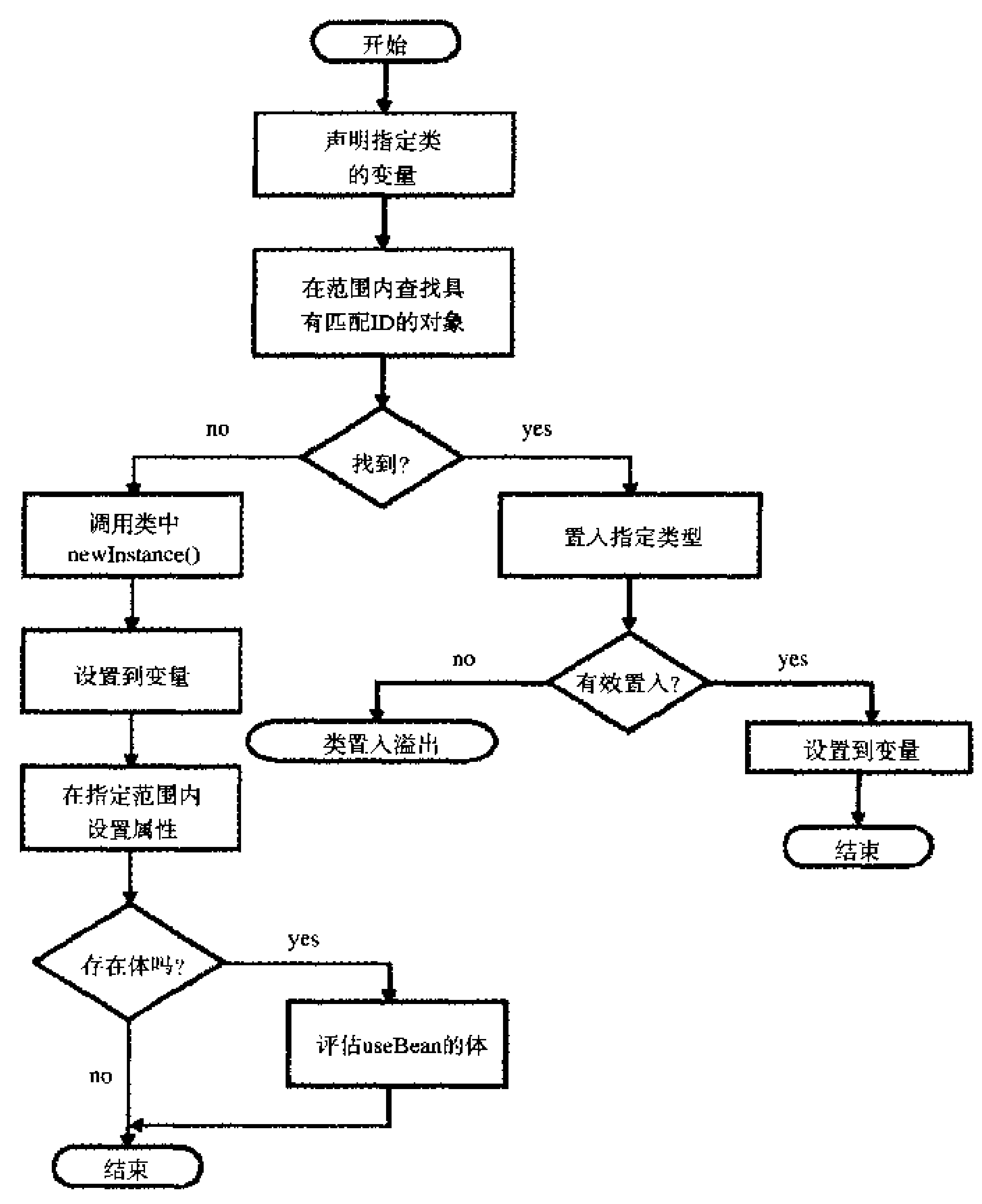


图15-2 当只给定class，<jsp:useBean>过程流线图

指定type和class 此情况与期望特定bean类很相近，只是使用了一个特殊接口。这种情况下，过程与前面的情况相同，只除了变量类型和置入操作使用类型不同，但不是类。算法如图15-3所示。

精彩好书见www.book118.com

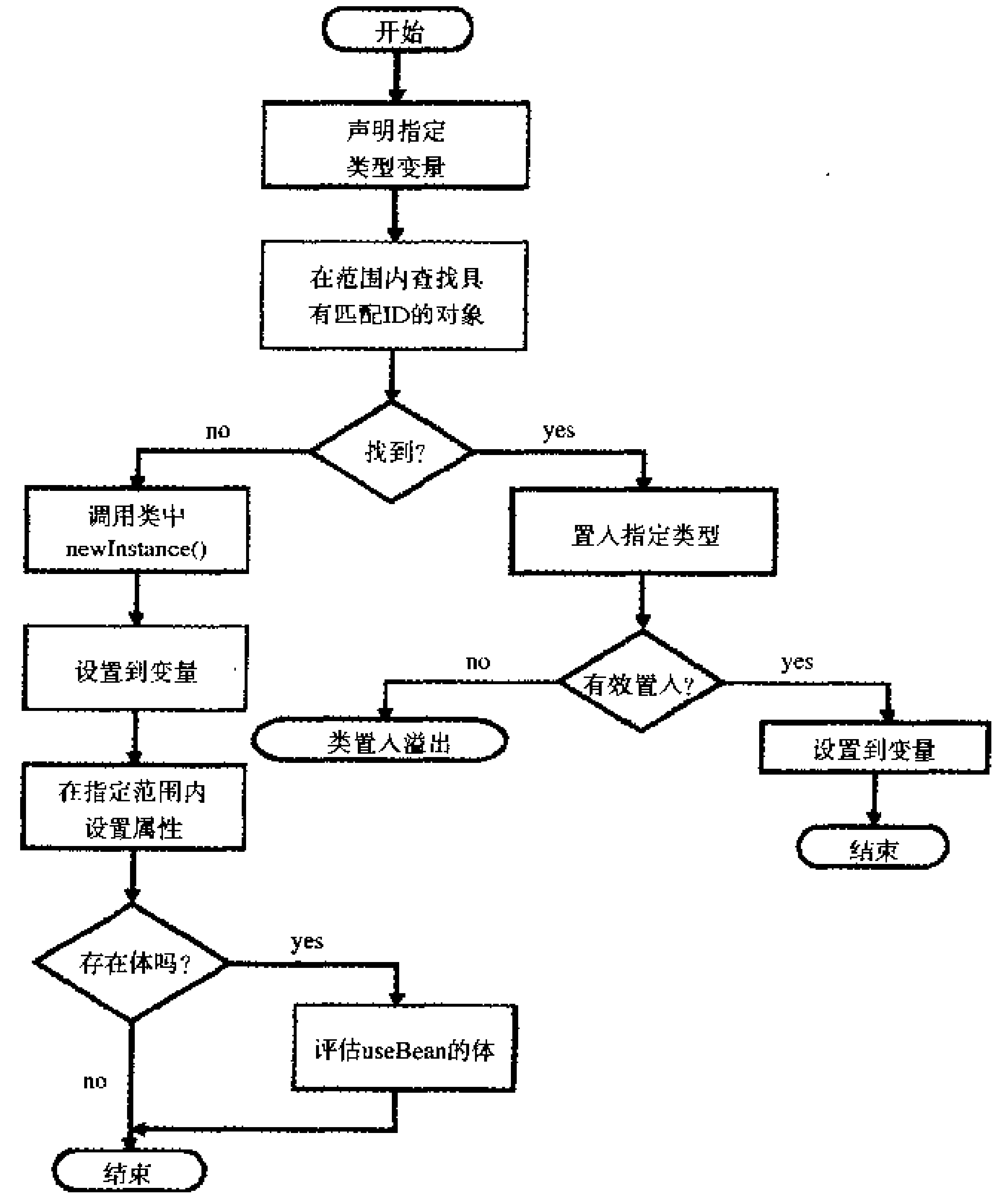


图15-3 当只给定class，<jsp:useBean>过程流线图

指定type和beanName 当已序列化的bean被导入JSP环境（如本章前面给出的Mortgage或CounterBean），应使用type和beanName属性。BeanName必须是Beans.instantiate(ClassLoader loader ,String name)使用的形式。名字首先被转换成一个文件名，如下：

- 句点被转换成"/"
- 后面附加了.ser。

例如，jspcr.beans.mortgage.Mortgage被转换成jspcr/beans/mortgage/Mortgage.ser。如果类载入器可以通过名字找到一个文件，则将其反序列化以获得对象。否则，初始名字被当作一个类名，类载入器试着创建命名类的一个实例。两种情况下，新生成的bean设置为指定id的一个脚本变量并保存为适当范围内的一个属性。过程如图15-4所示。

精彩好书见www.book118.com

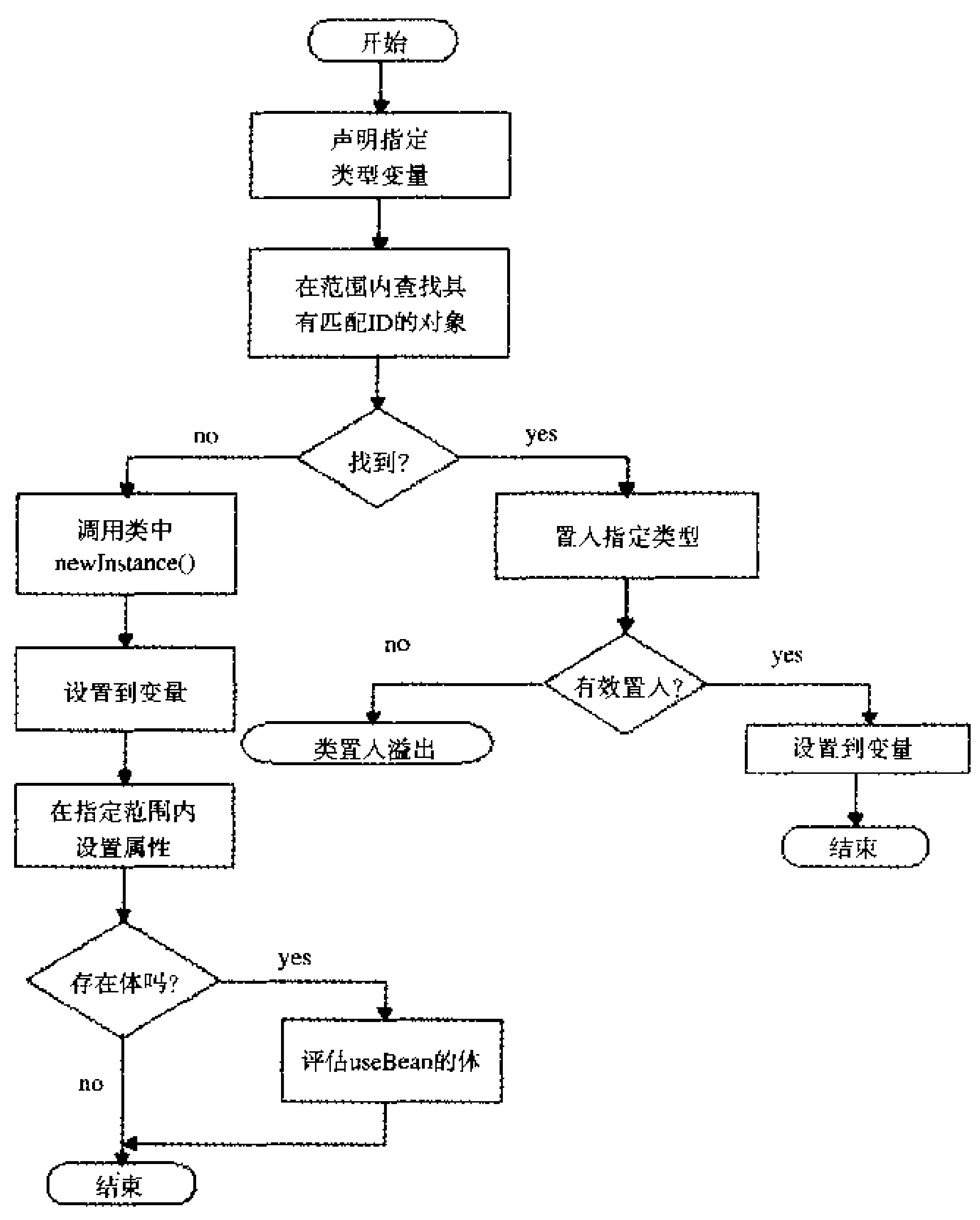


图15-4 当给定type和beanName时，<jsp:useBean>过程流线图

15.2.2 <jsp:setProperty>

<jsp:setProperty>行为基于JSP页面中的值设置bean属性值。语法可为下列4种形式的任意一种：

```
<jsp:setProperty name="name" property="property" value="value"/>
```

或

```
<jsp:setProperty name="name" property="property" param="param"/>
```

或

精彩好书见www.book118.com


```
<jsp:setProperty name="name" property="property"/>
```

或

```
<jsp:setProperty name="name" property="*" />
```

或

这里name、property、param和value在下面各节描述。

1. <jsp:setProperty> name属性

name属性标识其属性应设置的bean。该名字必须作为<jsp:useBean>标签的id属性被指定过。

2. <jsp:setProperty> property属性

一旦标识了特定的bean，则必须指定要设置属性名或属性。这是property属性的任务。该属性可为一属性名或特定值*。如果此属性为一名字，那么此bean必须有一个相应的属性设置方法。例如，在Mortgate bean中，存在setPrincipal ( )、setRate ( ) 和setTerm ( ) 方法，分别用于设置principal、rate和term属性。

如果属性值为*，那么此bean的设置属性列表与当前请求中的参数列表比较，一旦匹配出现，则使用相应的请求参数调用set方法。例如，如果HTML窗体包含下列域：

```
<FORM ACTION="DoMortgage.jsp">
Principal: <INPUT TYPE="TEXT" NAME="principal"><BR>
Annual Interest Rate: <INPUT TYPE="TEXT" NAME="rate"><BR>
Term in months: <INPUT TYPE="TEXT" NAME="term"><BR>
<INPUT TYPE="SUBMIT">
</FORM>
```

且DoMortgage.jsp有下列行为：

```
<jsp:useBean id="loan" class="jspcr.beans.mortgage.Mortgage">
<jsp:setProperty name="loan" property="*"/>
</jsp:useBean>
```

那么下列setProperty行产生同样的影响：

```
<jsp:setProperty
        name="loan"
        property="principal"
        value="<%= request.getParameter("principal") %>"/>

<jsp:setProperty
        name="loan"
        property="rate"
        value="<%= request.getParameter("rate") %>"/>

<jsp:setProperty
        name="loan"
        property="term"
        value="<%= request.getParameter("term") %>"/>
```

精彩好书见www.book118.com

其好处是代码行更少，包含错误的机会也就更少。

这也是通过指定属性名，而不是value或param属性得到的效果。这种情况下，设置属性值来自相应的请求参数。

注意 *如果property属性为*，或者如果value或param均未指定，请求参数为null或者空字符串，那么相应的bean属性不会改变。*

3. <jsp:setProperty> param属性

当请求参数与bean属性名不同时，可使用<jsp:setProperty>中的param属性映射请求参数到bean属性。如果指定了param属性，具有相应名字的请求参数被设置为在property属性中命名的属性[1]。

4. <jsp:setProperty> value属性

value属性指定了设置给bean属性的值。如果属性缺省，那么如前所述，使用相应的请求参数值。否则，可以形式<%=expression%>的字符串或JSP表达式指定该值。后者的语法称为请求时属性表达式。

如果取值是一个字母字符串，那么bean属性必须具有java.lang.String类型或为伪指令类型（boolean、byte、char、double、int、float、long）或相应的对象包容器类型（Boolean、Byte、Char、Double、Integer、Float、Long）。使用对象包容器valueOf方法可将某些字符串转换为非字符串类型。例如，Mortgate属性rate，带有setter方法：

```
public void setRate(double x)
{
   this.rate = x;
}
```

可以如下被调用：

```
<jsp:setProperty name="loan" property="rate" value="8.75"/>
```

然后JSP转换器创建设置属性的代码如下：

```
loan.setRate(Double.valueOf("8.75").doubleValue());
```

如果使用一个请求时表达式给出取值：

```
<jsp:setProperty name="loan" property="rate"
value="<%= LIBOR.getSixMonthLiborRate() + 0.05 %>" />
```

那么JSP转换器使用自测查找属性类型并将表达式置入此类型：

```
loan.setRate((double) (LIBOR.getSixMonthLiborRate() + 0.05));
```

15.2.3 <jsp:getProperty>

可以使用<jsp:getProperty>行为检索bean属性值。此行为形式为：

---

[1] 正常情况下，在将窗体变量名设置为bean属性时可以有多种选择，因为窗体很少在应用间共享。因此在窗体中像在bean中只简单地使用同样的名字时会给自己增加很多麻烦。

精彩好书见www.book118.com


```
<jsp:getProperty name "name" property."property"/>
```

这里name是带有相应id属性的bean，property是属性名。属性名必须是一个字母字符串，而非请求时表达式。当<jsp:getProperty>标签在运行时被评估，相应bean属性值被转换成一个字符串并写入JSP输出流中。

15.3 一个完整例子——带有bean的个性化风格

现在将所有元素放在一起，考虑一个完整实例。许多Web站点根据用户的意见定制页面的内容和外观。这种个性化可能采取对产品和与用户前面已经表示出兴趣的相关信息的超级链接的形式。例如，MSNBC显示本地天气和用户选择的股票价格。其后面隐藏的思想是如果信息总是在主页面上，那么用户就会很少会离开站点去查找这些信息。

这里（LyricNote Web设计者）已经决定向主页面加入此类信息。加入的第一片是可由用户定制报告出指定本地区域内天气的本地天气信息行。

15.3.1 从Web得到天气数据

这里要做的第一件事是天气信息源。在美国，它很容易可以从国家天气服务中心——国家海洋和大气管理组织得到。其Web页面提供天气数据、预报、当前条件、天气图、风暴预测和丰富的其他天气信息。虽然这些Web页面设计为通过用户的Web浏览器访问，但也很容易作为Java程序的URL输入流被读取。

为此，创建一个天气Observation bean。Observation使用机场代码作为报告此区域当前天气状况的国家天气服务Web站点的关键字。此bean有5个属性，在下表中列出：

当此bean解析国家天气服务Web页面时这些属性应在其内部设置，这些属性大部分都只有get方法。惟一可设置的属性是机场代码。调用setAirportCode（）使得此bean取得最后的读取并修改它的其他属性。

从机场代码中，此bean构建具有机场天气读取信息的特定Web页面的URL，然后打开URL输入流，解析HTML文件取得所需的特定属性：位置、时间和温度。

天气观测bean的属性

属 性	访 问	描 述
airportCode	read/write	进行大气观测的机场的3字符代码。设置该属性指导bean进入天气Web站点取得最后的读取信息
URL	read-only	包含此机场读取信息的国家天气服务Web页面的URL
location	read-only	机场名
time	read-only	观测数据和时间
temperature	read-only	摄氏温度

注意 这是链表中一个不可靠的链接。由外部资源产生的Web页面明显要发生变化，因此用于解析它的bean必须周期性地进行查阅和修改。当页面是基于格式为确定已知的内部Web站点时，这种“Web 采矿”技术非常有用。然而，许多政府和商业Web站点是计算机

精彩好书见www.book118.com

生成的并以相当正规的格式给出信息。

```
package jspcr.beans.weather;

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

/**
* A bean that extracts weather information from
* the U. S. National Weather Service web site
*/
public class Observation implements Serializable
{
   /**
   * The base URL for National Weather Service data
   */
   private static final String BASEURL =
      "http://weather.noaa.gov/weather/current";

   /**
   * Date format used for parsing observation time
   */
   private static final SimpleDateFormat DATEFMT =
      new SimpleDateFormat("MMM dd, yyyy - hh:mm aa zzz");

   /**
   * Airport code
   */
   private String airportCode;

   /**
   * Full name of location
   */
   private String location;

   /**
   * Time of observation
   */
   private Date time;

   /**
   * Temperature in degrees Celsius
   */
   private Double temperature;

   // ==========================================
```

精彩好书见www.book118.com


```
// Bean accessor methods
// ===========================================================

/**
* Returns the airport code
*/
public String getAirportCode()
{
   return airportCode;
}

/**
* Sets the airport code, which
* causes the bean to be reloaded.
* @param airportCode the airportCode.
*/
public void setAirportCode(String airportCode)
   throws IOException
{
   this.airportCode = airportCode;
   loadFromURL(getURL());
}

/**
* Returns the location.
*/
public String getLocation()
{
   return location;
}

/**
* Sets the location.
* @param location the location.
*/
protected void setLocation(String location)
{
   this.location = location;
}

/**
* Returns the time.
*/
public Date getTime()
{
   return time;
}
```

精彩好书见www.book118.com


```
/**
* Sets the time.
* @param time the time.
*/
protected void setTime(Date time)
{
this.time = time;
}

/**
* Returns the temperature.
*/
public double getTemperature()
{
   return (temperature == null)
             ? 0
             : temperature.doubleValue();
}

/**
* Sets the temperature.
* @param temperature the temperature.
*/
protected void setTemperature(double temperature)
{
   this.temperature = new Double(temperature);
}

/**
* Returns the URL of the NWS web page that contains
* current weather conditions at the airport
*/
public URL getURL() throws MalformedURLException
{
   StringBuffer sb = new StringBuffer();

   sb.append(BASEURL);
   sb.append("/K");
   sb.append(airportCode.toUpperCase());
   sb.append(".html");

   return new URL(sb.toString());
}

// ===========================================
// Web page parsing routines
// ===========================================

/**
```

精彩好书见www.book118.com


```
 * Loads the weather data from a URL
 */
protected void loadFromURL(URL url)
    throws IOException
{
    load(url.openStream());
}

/**
 * Parses an HTML input stream to extract a weather
 * observation. Note that this uses heuristics to
 * determine where each data element can be found
 * in the HTML. As such, it is subject to change.
 */
protected void load(InputStream stream) throws IOException
{
    location = null;
    time = null;
    temperature = null;

    BufferedReader in =
        new BufferedReader(
        new InputStreamReader(stream));

    for (;;) {
        String line = in.readLine();
        if (line == null)
            break;

        if (location == null)
            parseLocation(line);
        if (time == null)
            parseTime(line);
        if (temperature == null)
            parseTemperature(line);
    }
    in.close();
}

/**
 * Searches the current line for the location
 */
protected void parseLocation(String line)
{
    final String TOKEN1 = "<TITLE>";
    final String TOKEN2 = "-";
    final String TOKEN3 = "</TITLE>";
```

精彩好书见www.book118.com


```
    int p = line.indexOf(TOKEN1);
    if (p != -1) {
       p += TOKEN1.length();
       p = line.indexOf(TOKEN2, p);
       if (p != -1) {
          p += TOKEN2.length();
          int q = line.indexOf(TOKEN3);
          if (q != -1) {
             String token = line.substring(p, q).trim();
             StringTokenizer st =
                new StringTokenizer(token, ",");
             token = st.nextToken();
             token = st.nextToken();
             setLocation(token);
          }
       }
    }
}

/**
* Searches the current line for the time
*/
protected void parseTime(String line)
{
    final String TOKEN1 = "<OPTION SELECTED>";
    final String TOKEN2 = "<OPTION>";

    int p = line.indexOf(TOKEN1);
    if (p != -1) {
       p += TOKEN1.length();
       int q = line.indexOf(TOKEN2, p);
       if (q != -1) {
          String token = line.substring(p, q).trim();
          Date date = DATEFMT.parse
             (token, new ParsePosition(0));
          if (date != null)
             setTime(date);
       }
    }
}

/**
* Searches the current line for the temperature
*/
protected void parseTemperature(String line)
{
    final String TOKEN1 = "(";
    final String TOKEN2 = "C)";
```

精彩好书见www.book118.com


```
        int q = line.lastIndexOf(TOKEN2);
        if (q != -1) {
            int p = line.lastIndexOf(TOKEN1);
            if (p != -1) {
                p += TOKEN1.length();
                String token = line.substring(p, q).trim();
                try {
                    setTemperature(Double.parseDouble(token));
                }
                catch (NumberFormatException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

此bean的核心是load ( ) 方法。它解析查询位置、时间和温度的输入流。图15-5给出了典型的国家天气服务Web页面，显示了North Carolina的Raleigh-Durham国际机场的天气条件。

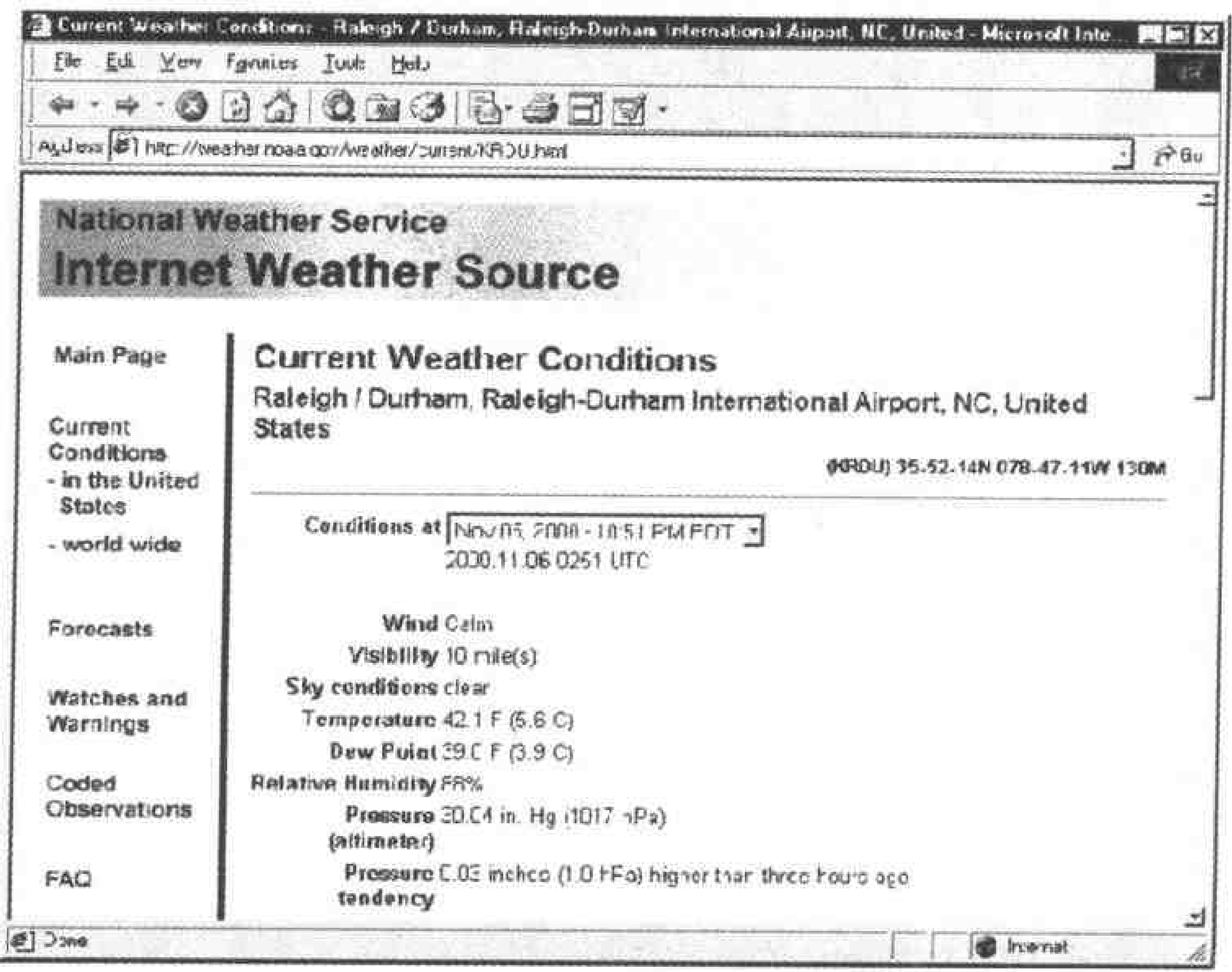


图15-5　Raleigh-Durham国际机场的国家天气服务Web页面

精彩好书见www.book118.com

如果可以找到抽取信息的可靠规则，则Web页面的HTML就会包括所需的所有信息。location属性证明非常简单，因为它包含在HTML的<TITLE>...</TITLE>标签内：

```
<HTML>
<HEAD>
<TITLE>Current Weather Conditions - Raleigh / Durham,
Raleigh-Durham International Airport, NC,
United States </TITLE>
```

（在HTML中标题只有一行，为了增加可读性这里将其分行。）parseLocation（ ）方法测试当前行查看它是否包含<TITLE>和</TITLE>标记。如果有，标记之间的文本被抽出，第2个逗号分隔的域被用做位置名。

time属性定位在文档中惟一的<OPTION SELECTED>和<OPTION>标签之间：

```
<TD><FONT FACE="Arial,Helvetica"><FORM>
<SELECT><OPTION SELECTED> Nov 05, 2000 - 10:51 PM EDT
<OPTION> Nov 05, 2000 - 09:51 PM CDT
<OPTION> Nov 05, 2000 - 08:51 PM MDT
<OPTION> Nov 05, 2000 - 07:51 PM PDT
<OPTION> Nov 05, 2000 - 06:51 PM ADT
<OPTION> Nov 05, 2000 - 05:51 PM HDT
</SELECT><BR> 2000.11.06 0251 UTC
</FORM></FONT></TD>
```

（再次增加可读性，将HTML换行。最初的HTML只有一行。）这里使用SimpleDateFormat中的parse（ ）方法将日期和时间转换成一个java.util.Date。

temperature属性在Web页面中以下列形式出现：

```
<TR VALIGN=TOP>
<TD ALIGN=RIGHT BGCOLOR="#FFFFFF"><B><FONT COLOR="#0000A0">
<FONT FACE="Arial,Helvetica">Temperature</FONT></FONT></B></TD>
<TD><FONT FACE="Arial,Helvetica"> 42.1 F (5.6 C)
</FONT></TD>
</TR>
```

此属性要分隔开有点困难，因为关键字Temperature出现在前面一行。这里采用一种更简单的方案，在(...C)中查找第一行结尾并解析括号内的摄氏温度的字符串。

该类的其余部分主要包括每一属性的适当的get和set方法。

15.3.2 LyricNote入口

现在已经完成了隐藏查找Web页面和解析HTML的各种杂乱细节的一个bean，并给出了给定机场代码下取得当前天气条件的一种方式。此bean缩减了调用的复杂性处理功能，该功能调用比调用Math.cos（ ）求一个三角余弦更复杂。现在可以将此bean放入Web页面入口的工作中。

策略是使用一个持续的cookie记录用户的位置选择（机场代码）。每次显示Web页面时，从cookie中检索机场代码并用于初始化一个Observation bean。输出的位置、时间和温度显示在页

精彩好书见www.book118.com

面顶部标志符中的一行。另外，该行还包括使用户可以选择新的机场代码的一个超级链接。以下是一部分JSP：

```
<%@ page session="false" %>
<HTML>
<HEAD>
<TITLE>LyricNote Portal</TITLE>
<LINK REL="stylesheet" HREF="style.css">
</HEAD>
<BODY>
<IMG SRC="images/lyric note.png">
<HR COLOR="#000000">

<%-- Get weather cookie --%>

<%
   String airportCode = "RDU";
   Cookie[] cookies = request.getCookies();
   if (cookies != null) {
      for (int i = 0; i < cookies.length; i++) {
         Cookie cookie = cookies[i];
         if (cookie.getName().equals("airportCode")) {
            airportCode = cookie.getValue();
            break;
         }
      }
   }
%>

<%-- Get the weather observation bean for that location --%>

<jsp:useBean id="wobs" class="jspcr.beans.weather.Observation">
<jsp:setProperty
   name="wobs"
   property="airportCode"
   value="<%= airportCode %>"/>
</jsp:useBean>

<%-- Show weather information --%>
<SPAN CLASS="whiteOnBlue"> Weather </SPAN>
<SPAN CLASS="blueOnWhite">
<jsp:getProperty name="wobs" property="location"/>
<jsp:getProperty name="wobs" property="time"/>
<jsp:getProperty name="wobs" property="temperature"/> C&deg;
</SPAN>
<A CLASS="whiteOnBlue" HREF="AirportSelection.html">
```

精彩好书见www.book118.com


```
 Select City 
</A>
<HR COLOR="#000000">

<%-- Show the rest of the web page --%>

</BODY>
</HTML>
```

对HTTP请求返回的cookie数组扫描名字为airportCode的一个cookie。如果存在此cookie，则其值可替换为缺省机场代码（对Raleigh/Durham North Carolina 为RDU），接着声明Observation bean：

```
<jsp:useBean id="wobs" class="jspcr.beans.weather.Observation">
```

并用选择的机场代码对其初始化：

```
<jsp:setProperty
   name="wobs"
   property="airportCode"
   value="<%= airportCode %>"/>
```

这里要做的就是抽取该bean的属性并将其写入输出流：

```
<jsp:getProperty name="wobs" property="location"/>
<jsp:getProperty name="wobs" property="time"/>
<jsp:getProperty name="wobs" property="temperature"/> C&deg;
```

图15-6显示结果。

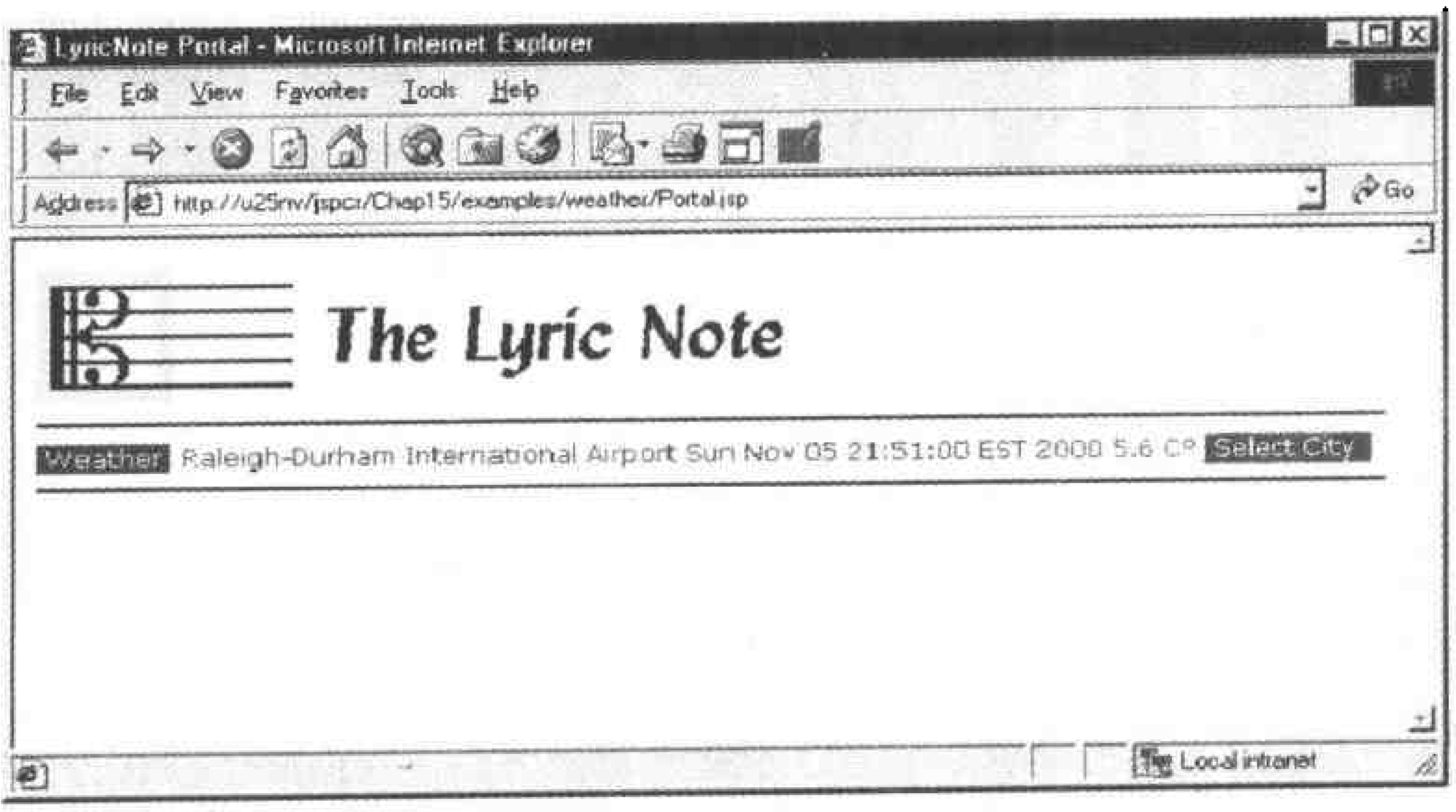


图15-6　带有缺省天气信息的LyricNote入口Web页面

精彩好书见www.book118.com

如果用户点击Select City超级链接，则出现天气数据已知的机场列表，如图15-7所示。这不是一个JSP页面，只是包含适当cookie设置的JSP的简单HTML窗体。

图15-7 机场选择页面

窗体form属性指向SetAirportCode.jsp。这是一个非可视JSP页面，将新的机场代码cookie发送到用户并重定向到入口页面，如下列代码所示。

```
<%@ page session="false" %>
<%
   String airportCode = request.getParameter("airportCode");
   if (airportCode != null) {
      Cookie cookie = new Cookie("airportCode", airportCode);
      final int ONE_YEAR = 60 * 60 * 24 * 365;
      cookie.setMaxAge(ONE_YEAR);
      response.addCookie(cookie);
   }
   response.sendRedirect("Portal.jsp");
%>
```

如果Web用户刚好定位在Roswell，New Mexico（或对此地天气条件感兴趣），并在机场选择框中选择适当行，机场代码cookie值被设置为ROW。通过LyricNote入口的所有后续请求使得Observation bean检索Roswell工业航空中心机场的天气条件（图15-8）。

精彩好书见www.book118.com

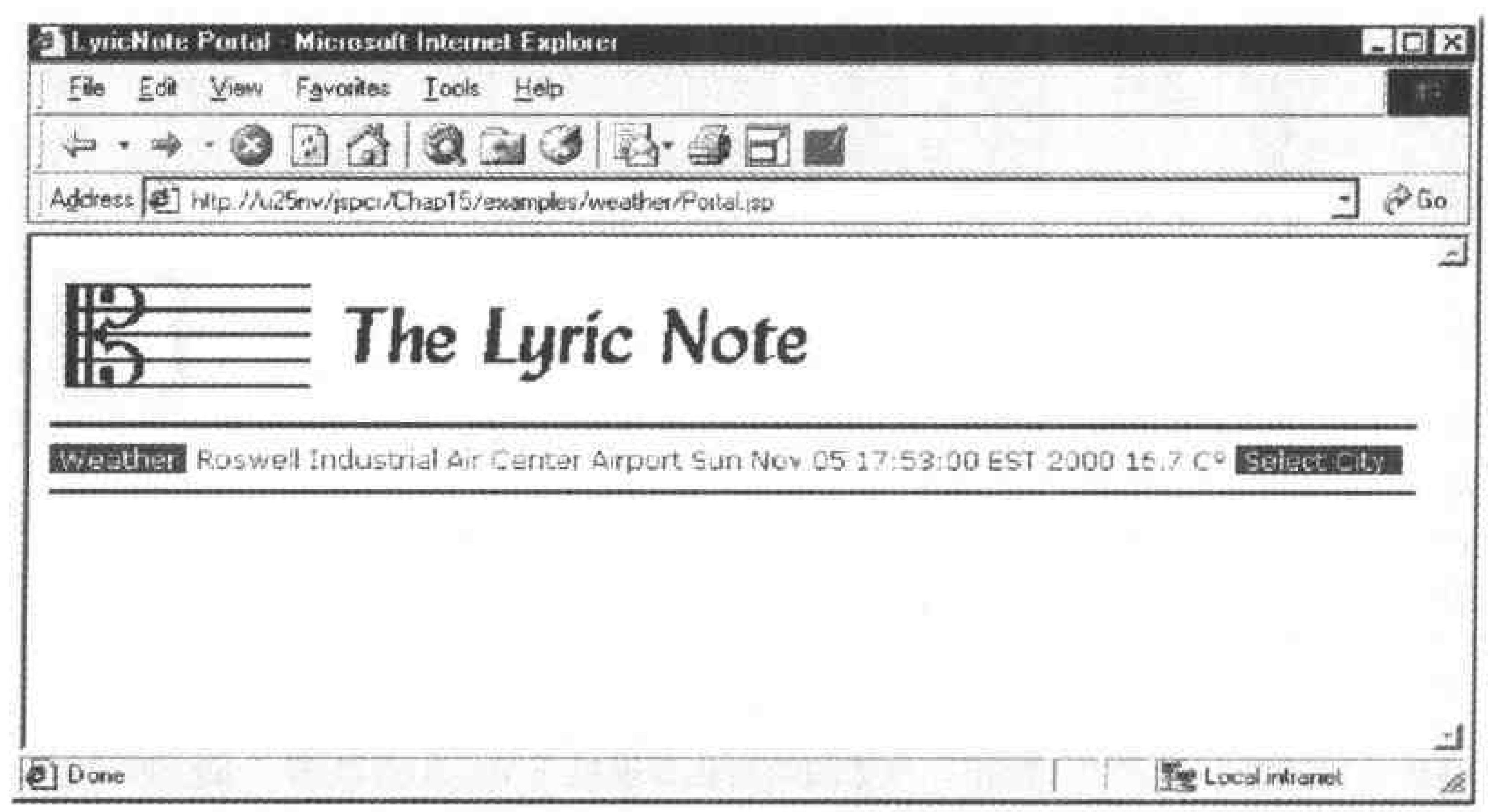


图15-8 带有已修改天气信息的LyricNote入口Web页面

15.4 小结

在Java体系结构中基于组件的编程使用了JavaBean。bean编程模型使用公有的get和set方法访问私有的bean属性，使用事件侦听者将其他类链接到一个bean的状态。并使用序列化使bean具有持续性。

JSP编程环境提供声明和访问bean的标准标签：

- `<jsp:useBean>`
- `<jsp:setProperty>`
- `<jsp:getProperty>`

useBean标签可以声明、例示和初始化一个bean。此标签具有各种属性结合，使得bean可以从存在的名空间中被抽取出来，创建一个新的实例或从一序列化的对象中被保存。所支持的名空间为页面范畴、请求范畴、会话范畴或应用范畴。

setProperty和getProperty用于从一个bean中保存或检索属性。除了字母字符串值，属性设置可来自于窗体参数和请求时表达式。

在JSP环境中JavaBean具有突出的优点。可以分别独立对其进行编码和测试，然后即可重复用于applet、servlet和单机应用上下文。JavaBean也减少了暴露在JSP scriptlet和声明中的Java代码量。结合cookie、定制标签和其他高级JSP技术，JavaBean给出开发实用Web应用一种可靠的，丰富的基础方案。

精彩好书见www.book118.com

第16章 JSP 和 XML

自从1996年在W3C出现其雏形以及在1998年被采用作为W3C推荐产品，XML（可扩展标记语言）已经成为结构化数据存储和交互的通用语言。XML用于Web站点内容管理、商业数据交换及体系结构、经济报表和音乐等大量的应用。除此之外，XML工具和扩展正将其方式渗透到所有编程观念中。

本章介绍将XML嵌入到Web应用的几种方式。简介后，讨论两种XML解析器模型，然后是XSL转换。这三种技术在同一HTML创建任务中以三种解决方案分别予以讲解。

16.1 XML简介

XML是描述带有用户定义标记标签集的结构化数据的系统。XML本身不是一种语言，而是定义特定意图语言的系统。它粗看起来像HTML，使用预定义<UL>，<LI>，<TABLE>和其他标签标记Web文档的各节。差别是HTML具有固定的标签集，而XML使用户可以设计描述数据所需的任意标签集。

例如，一首钢琴曲可如下表示：

```
<?xml version="1.0"?>
<song>
   <title>The Birds</title>
   <words-by>Hilaire Belloc</words-by>
   <music-by>Benjamin Britten</music-by>

   <track name="Voices">
      <time-signature>2/2</time-signature>
      <tempo>Andante con moto</tempo>
      <measure>
         <rest duration="1"/>
      </measure>
      <measure>...</measure>
   </track>

      <track name="Piano">
         <time-signature>2/2</time-signature>
         <tempo>Andante con moto</tempo>
         <measure>
            <note duration="8" value="e" octave="2"/>
            <note duration="8" value="g#" octave="2"/>
            <note duration="8" value="c#" octave="3"/>
            <note duration="8" value="g#" octave="3"/>
```

精彩好书见www.book118.com


```
            <note duration="2" value="f#" octave="3"/>
        </measure>
        <measure>...</measure>
    </track>
</song>
```

注意，歌曲的这种表示完全是结构化的——它不必理会歌曲如何表现。实际上，同一个XML文档也可用来生成打印的歌曲样单和在MIDI播放器中合成音调。

16.1.1 XML解决的问题

早期的文本处理格式经常不会区分内容和表示。例如，RTF具有类似于表格和列表的结构化数据编码以及字体和图像编码。HTML也遇到了同样的问题。类似于<table>、<tr>和<td>具有指定宽度和长度属性的元素通常用于对Web页面的物理布局产生影响，而不是以列表式风格将相关条目分组。

此方法的问题是当需要新的输出格式时，包含在文档中的格式化信息变得毫无用处。更糟的是，最初设计来传送结构化信息的标签只是因为其边界影响会被误用，如使用<UL>产生缩进。

比较起来，XML则完全基于结构。特定数据元素可被清晰标识并通过文本搜索应用抽取出来。如果一个XML文档需要交付给Web浏览器，可使用XSL样式单编程将其转换为HTML。如果文档需要用于一个事务处理系统，可通过XML解析器解析它，该解析器抽取指定域完成事物处理。XML文档可做为一个树型结构加以浏览或将其压缩成关系型数据库表格。只要使用文档的应用知道其编写语言，应用就可以找到并抽取所需数据。

16.1.2 XML语法

XML很简单，其语法的规则也容易学习。XML文档由元素组成，每一元素都有一个开始标签，一个体和一个结束标签，如下所述：

```
<tempo>Andante con moto</tempo>
```

开始标签<tempo>由括在小于和大于符号中的一个标签名组成。结束标签</tempo>与开始标签相同，但是带有一个正斜杠后跟一个大于符号。体的组成是开始标签和结束标签之间的内容，可以包含一般的文本或其他XML元素。

开始标签可以包含属性，它是开始标签内名字后面的name=“value”对编码，但位于结束的大于符号之前：

```
<track name="voices"> ... </track>
```

XML中的属性必须括在单引号或双引号中。

如果一个元素体为空，可以使用开始和结束标签的缩写形式。在此形式中，正斜杠从结束标签被移到开始标签中正好位于结束大于符号前，然后省略体和结束标签。因此，下述两种形式功能上是等价的：

精彩好书见www.book118.com


```
<rest duration="1"></rest>
<rest duration="1"/>
```

元素可以互相嵌套任意深度：

```
<song>
   <track>
      <measure>...</measure>
   </track>
</song>
```

但其结束标签必须以与开始标签出现的顺序刚好相反的次序出现。也就是说元素不能重叠。下述语句为非法：

```
<B><I>Do not do this!</B></I>
```

因此，合法格式的XML文档只有一个外部元素，称为文档元素，其中包含任意数目的正确嵌套的内部元素。

对XML语法的完整细节，请参考XML 1.0规范第二版，可在W3C 2000年10月6日的推荐版本中找到。此文档网址为http://www.w3.org/TR/REC-xml。

16.1.3 文档类型定义

XML不只是自由格式的标签组。很明显，使用XML文档进行输入的应用需要知道其可以包含的元素，这些元素被嵌套或重复的方式以及允许的属性等等。同时，生成XML文档（以及使用文本编辑器人为合成的文档）的应用需要知道同样的结构化信息。这就是文档类型定义（DTD）的功能。

DTD允许在特定文档类型中出现标签和属性的定义。例如，memo文档的DTD可定义为<memo>、<from>、<to>、<subject>、<text>和<paragraph>标签，指出<paragraph>元素只能出现在<text>元素中并需要<from>和<to>，而其余部分是可选的。生成memo文档的应用能确定它们仅生成语法正确版本。验证工具可以读取人为生成的memo文档并判断它们是否符合语法。这意味着接收端的应用可以通过对文档的正确理解并正确处理它而得以信任。

XML文档指出其使用的DTD以及在文档元素前使用一个<!DOCTYPE>标签立即知道DTD的位置：

```
<?xml version="1.0"?>
<!DOCTYPE song SYSTEM "song.dtd">
<song>
...
</song>
```

DTD也可以被嵌入到文档本身：

```
<?xml version="1.0"?>
<!DOCTYPE song [
...
]>
```

精彩好书见www.book118.com


```
<song>
...
</song>
```

或保留在一公有库中：

```
<?xml version="1.0"?>
<!DOCTYPE song PUBLIC publicid URL>
<song>
...
</song>
```

但一个DTD不是必须的。如果给出，文档就必须符合它。在XML规范的语言中，如果它符合语法规则（所有元素均被封闭起来，没有嵌入元素，所有属性均在引号中），则称文档是规范格式。如果文档包含一个DTD，且为规范格式并符合DTD，则称其有效。

这里是song文档类型的DTD如下所示：

```
<!ELEMENT song (title?,words-by?,music-by?,track+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT words-by (#PCDATA)>
<!ELEMENT music-by (#PCDATA)>
<!ELEMENT track (time-signature|tempo|measure)*>
<!ATTLIST track
   name CDATA #IMPLIED>
<!ELEMENT time-signature (#PCDATA)>
<!ELEMENT tempo (#PCDATA)>
<!ELEMENT measure (note|rest)+>
<!ELEMENT note EMPTY>
<!ATTLIST note
   duration CDATA #IMPLIED
   value CDATA #IMPLIED
   octave (1|2|3|4|5|6|7|8) #REQUIRED>
<!ELEMENT rest EMPTY>
<!ATTLIST rest
   duration CDATA #IMPLIED>
```

一个DTD由元素和属性列表组成。每一元素定义给出元素名、其后为可以包含的元素、它们的次序、是否需要以及是否可以重复。此描述可采取以下几种形式：

- 一般文本，表示为（#PCDATA），对应解析字符数据。
- 可接受的子元素按次序排列，由逗号分隔。
- 由逻辑OR符号“|”分隔的互斥元素。
- 子元素和括起的子元素列表，后跟一个重复计数：“？”，意即0或出现一次。“*”代表0或多次，“+”代表1或多次。
- 不能包含体的元素声明为EMPTY。

例如，<song>元素允许包含可选的<title>、<words-by>和<music-by>元素，后面跟一个或多个<track>元素。

精彩好书见www.book118.com


```
<!ELEMENT song (title?,words-by?,music-by?,track+)>
```

注意，如果给出子元素，它必须以指定次序只出现一次。<measure>元素定义为至少包含一个<note>或<rest>元素，后跟<note>或<rest>的任意重复次数。

```
<!ELEMENT measure (note rest)+>
```

<time-signature>元素可以只包含一般文本：

```
<!ELEMENT tempo (#PCDATA)>
```

元素具有的属性在包含元素名的<!ATTLIST>标签中列出，后跟每个属性的三组标记，指出其属性名、类型和缺省值。例如，<note>元素如下描述为具有可选duration和value属性，以及一个必须取值为整数1到8的octave属性。

```
<!ATTLIST note
   duration CDATA #IMPLIED
   value CDATA #IMPLIED
   octave (1|2|3|4|5|6|7|8) #REQUIRED>
```

如果DTD语法看起来有点复杂，不要担心。除非你是一个文档定义专家，否则很少会编写它。本节给出的非正式描述只是要有阅读DTD的基本能力。对于严格定义，请查阅XML规范。

注意 *阅读DTD的能力对理解web.xml发布描述器很有用。web.xml的结构和内容定义在servlet API规范中列出的web-app_2_2.dtd（或后续版本）。如果需要知道定义一个servlet初始化参数的位置，例如，DTD显示它们必须在<servlet>块中被编码，刚好在<servlet-class>之后，<load-on-startup>之前。*

16.2 XML解析器

为使用应用中的XML文档，需要对其进行解析。一个XML解析器读取文档并将其分隔称为开始标签、属性、体内容和结束标签。解析器具有应用编程接口，使用户可以不必自己去解释输入流的复杂性而轻松抽取所需元素。

存在两种常用的XML解析器模型：

- DOM 文档对象模型。
- SAX XML的简单API。

下面各节讨论每一模型。

16.2.1 文档对象模型

文档对象模型（DOM）是内存中文档的W3C标准表示法。与文本字符串不同，DOM将文档表示成一个节点树。该树可按任意次序遍历。节点可被增加和删除并且已修改DOM树可被保存为新文档。

DOM规范有不同的版本，由级别指定。DOM级别1是核心特性集，提供创建和访问文档元素的方式。DOM级别2，2000年11月13日作为W3C推荐产品被批准，增加了对名空间的支持。

精彩好书见www.book118.com

DOM不只是一个标准，也是一个应用编程接口（API）。W3C发布了组成org.w3c.dom包的一系列接口。然后不同的厂家提供实现这些接口的解析。流行的DOM解析器包括Xerces，来自Apache Software Foundation和JAXP，来自Sun微系统。

DOM API由4种类和接口组成：

- 节点
- 节点集
- 元数据
- 溢出

1. 节点接口

DOM中最有趣的基本单元是节点。一个XML文档中的所有内容——每一元素、开始标签中的属性、注释、严肃文本和整个文档都是节点。表16-1列出节点接口中的方法。

表16-1 节点接口中的方法

方法	描述
Node appendChild(Node newChild) throws DOMException	增加一个新节点到当前节点
Node cloneNode(boolean deep)	制作节点的复本。如果deep为真，递归复制此节点下的所有子树。否则，复制当前节点
namedNodeMap getAttributes()	如果节点是一个Element，返回此节点的已命名属性，否则，返回null
NodeList getChildNodes()	返回所有下一级子节点列表
Node getFirstChild()	返回第一个子节点。如果节点不是一个Element，则为null
Node getLastChild()	返回最后一个子节点。如果节点不是一个Element，则为null
Node getNextSibling()	返回同父的下一个子节点。如果节点不是一个Element，则为null
String getNodeName()	对已命名节点类型，如Element、Attr和Entity，返回节点名。对未命名节点类型，如Text、CDATAsection和Comment，分别返回#text,#cdata-section和#comment
int getNodeType()	返回指出节点特定类型的一个整数常量。返回值是节点接口中定义的下述常量之一： ATTRIBUTE_NODE CDATA_SECTION_NODE COMMENT_NODE DOCUMENT_FRAGMENT_NODE DOCUMENT_NODE DOCUMENT_TYPE_NODE ELEMENT_NODE ENTITY_NODE ENTITY_REFERENCE_NODE NOTATION_NODE PROCESSING_INSTRUCTION_NODE TEXT_NODE
String getNodeValue()	对Attribute和Text类型节点，返回其文本，否则返回null

精彩好书见www.book118.com

（续）

方 法	描 述
Document getOwnerDocument()	对此节点出现的文档返回Document节点
Node getParentNode()	如果这是一个Document、DocumentFragment或Attr节点，返回上一级父节点，否则返回null。仍未加入文档的新节点也可以有null父节点
Node getPreviousSibling()	返回同父的前一个子节点。如果父节点不是一个Element，则返回null
boolean hasChildNodes()	如果此节点有一个非空子节点列表，则返回true
Node insertBefore(Node child, Node beforeNode)	在指定节点前插入一个新的子节点。beforeNode可以为空，这种情况下，子节点被附加到列表结尾
Node removeChild (Node child) throws DOMException	从子节点列表中删除指定节点。如果该节点不是当前节点的子节点则产生溢出
Node replaceChild(Node newChild, Node oldChild) throws DomException	删除oldChild，用newChild替换它。如果该节点不是当前节点的子节点则产生溢出
void setNodeValue(String value) throws DOMException	设置当前节点值

在DOM级别2中，节点有两个新方法——isSupported（ ）和hasAttributes（ ），并包含了方法normalize（ ），它是Element接口的前面部分。

节点有13个指定子接口对应于在XML文档中出现的特殊节点类型。这些接口在表16-2中列出。

表16-2 针对特定节点类型的节点子接口

接 口	描 述
Attr	一个Element节点的属性。Attr具有检索属性名字和取值的方法。在DOM级别2中，Attr接口包括getOwnerElement（ ）方法
CDATASection	包围在XML文档中屏蔽语法<![CDATA[...]]>中的一个文本节点。CDATA段原封不动地被解析，不经过评估。它们允许文档内容包含字符和字符串，否则，将其解释为XML
CharacterData	三种包含文本的节点类型：Text、Comment和CDATASection的一个常用超级接口。它给出了取得和设置字符内容以及判断数据长度的方法
Comment	包含XML注释的节点。Comment值并不包含<!--和-->分隔符，只是注释文本，且包括空格
Document	将XML文档表示为一个整体的Document节点。在DOM实例中只有一个Documet节点。DOM级别2中加入了Document接口名空间的支持
DocumentFragment	用于构建一个Document节点子树的暂时节点。此接口没有方法
DocumentType	表示文档开始时<!DOCTYPE>元素的节点。此接口提供取得DTD名字、实体和在DTD中定义的注释方法
Element	Node最常用的子接口。表示一个XML开始标签、体和结束标签。除了从Node、Element中继承方法外，它还有设置和检索出现在开始标签中属性的方法。DOM级别2中Element包含大量的名空间已知的方法

精彩好书见www.book118.com

(续)

接 口	描 述
Entity	在XML文档中使用的一个外部组件，如图像文件。DOM级别1对此节点类型只提供最小支持
EntityReference	未评估实体的引用（名字或指针）。此接口只充作文档中的占位符，未定义方法
Notation	表示在DTD中声明的注释。注释描述了外部实体的格式
ProcessingInstruction	XML文档中指定的处理器指令。处理指令采用于语法<?<target>[<data>]?>
Text	包含元素体字符内容的Text节点

2. 节点集接口

各种DOM API方法返回节点集，为排序的列表，或者为节点名映射。表示这些集合有两个接口：NodeList和NamedNodeMap，分别如表16-3和表16-4所示：

表16-3 NodeList接口定义的方法

方 法	描 述
int getLength()	返回列表中节点号
Node item(int n)	返回列表中第n个节点，这里节点编号为0，1...

表16-4 NamedNodeMap接口定义的方法

方 法	描 述
int getLength()	返回列表中节点号
Node item(int n)	返回列表中第n个节点，这里节点编号为0，1...
Node getNamedItem(String name) void setNamedItem(Node item) void removeNamedItem(String name)	取得、设置或删除具有指定名的节点。如果集合中节点并不存在，getNamedItem（）返回null

DOM级别2中NamedNodeMap接口支持名空间中合格的条目名。

3. 节点元数据

XML特性可能是指定版本的。为判定DOM配置，DOM有一个接口使用户可以查询其支持的特性。此接口名为DOMImplementation。当前它只有一个方法：

```
boolean hasFeature(String feature, String version)
```

如果指定特性的指定级别被支持，则返回true。DOM级别2向DOMImplementation加入两个新方法以支持创建文档。

4. 溢出

DOM定义了一个溢出类名为DOMException。它是RuntimeException的子类。这意味着编译器不需要产生此溢出的方法被声明或在try/catch块中包含它。

5. DOM使用

典型的面向DOM的应用是创建一个DOM解析器实例，然后使它解析一个XML文档源创建

精彩好书见www.book118.com

DOM树。一旦创建了树，应用就可以浏览它，检验并抽取所需的内容。

示例解析器的含义是实现所特有的。可以直接使用new操作符创建。更高级别的方法是使用Sun微系统定义的Java API的XML（JAXP）包。在JAXP下，应用创建DocumentBuilderFactory的实例，可选地设置其namespaceAware和/或validating属性，然后使用该工厂实例得到一个解析器的实例。工厂实例找到匹配所需特性的DOM解析器类。

因此，为解析一个带有DOM的XML文档，应用可以执行如下：

```
DocumentBuilderFactory factory =
   DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(fileName);
Element root = document.getDocumentElement();
```

下面看一个在JSP页面中使用DOM进行XML解析的例子。下面XML文档选自LyricNote产品目录，可能是数据库查询的结果。它由产品代码标识的乐器列表组成。对每一乐器，文档包含了价格、手工质量，厂家名和产品描述。以下列表为缩写，完整列表有82个入口。

```
<?xml version="1.0"?>

<!DOCTYPE products PUBLIC "-//jspcr//products//EN"
   "http://u25nv/jspcr/Chap16/examples/products/products.dtd">

<products>

   <product code="001000">
      <product-type>IN</product-type>
      <price>537.00</price>
      <on-hand>48</on-hand>
      <manufacturer>Clemens-Altman</manufacturer>
      <description>Silver Flute - Student</description>
   </product>

   <product code="001010">
      <product-type>IN</product-type>
      <price>876.00</price>
      <on-hand>83</on-hand>
      <manufacturer>Gabriel</manufacturer>
      <description>Silver Flute</description>
   </product>

   ...

   <product code="001790">
      <product-type>IN</product-type>
      <price>165.50</price>
```

精彩好书见www.book118.com


```
        <on-hand>94</on-hand>
        <manufacturer>Roush and Sons</manufacturer>
        <description>Cello case (1/2 size)</description>
    </product>

</products>
```

JSP解析此文档并只抽取厂家为Clemens-Altman的产品。它在一个HTML表格中排列此子集，列对应产品代码、描述和价格。因为文档由产品元素集组成，一个逻辑方案是解析文档并将其转换为Product对象集合。Product对象的域对应每一产品块中的XML元素。使用Product对象帮助解析很有意义。因为DOM创建了一个树，可以简单定位每个<product>元素，创建一个Product对象并调用其load（ ）方法，将DOM元素作为参数传递。

下面给出了Product对象。除了其load（ ）方法外，Product对每一私有域还包含get和set方法。

```
package jspcr.xml.samples;

import org.w3c.dom.*;
public class Product
{
   private String code;
   private String productType;
   private double price;
   private int onHand;
   private String manufacturer;
   private String description;

   /**
   * Load the product data from a DOM element
   */
   public void load(org.w3c.dom.Element element)
   {
      code = element.getAttribute("code");
      for (Node node = element.getFirstChild();
            node != null;
            node = node.getNextSibling())
      {
         // Select only element nodes

         if (node.getNodeType() != Node.ELEMENT_NODE)
            continue;

         String tagName = node.getNodeName();

         // product-type

         if (tagName.equals("product-type")) {
```

精彩好书见www.book118.com


```
                String text = node.getFirstChild().getNodeValue();
                productType = text.trim();
            }

            // price

            else
            if (tagName.equals("price")) {
                String text = node.getFirstChild().getNodeValue();
                price = Double.parseDouble(text.trim());
            }

            // on-hand

            else
            if (tagName.equals("on-hand")) {
                String text = node.getFirstChild().getNodeValue();
                onHand = Integer.parseInt(text.trim());
            }

            // manufacturer

            else
            if (tagName.equals("manufacturer")) {
                String text = node.getFirstChild().getNodeValue();
                manufacturer = text.trim();
            }

            // description

            else
            if (tagName.equals("description")) {
                String text = node.getFirstChild().getNodeValue();
                description = text.trim();
            }
        }
    }
    // Not shown here - get and set methods
}
```

• 代码域很容易得到，因为它是产品元素的一个属性。需要调用元素的getAttribute（“code”）方法。其他域有点复杂，因为它们的值在产品子元素下文本节点内。这里的方案是遍历产品元素的子节点，将每一节点名与所需的域名进行比较。对此循环可采取几种方式：调用产品元素上的getChildNodes（ ）方法，返回一个NodeList对象。NodeList有一个getLength（ ）方法，会告之节点计数或是item（int index）方法，返回列表内在指定索引位置的节点。

精彩好书见www.book118.com

• 调用产品元素的getFirstChild ( ) 方法，然后依次调用每一子元素的getNextSibling ( ) 方法直到其返回为null。

这里的代码使用了第二种方法。

为取得文本节点值，可以利用每一数据元素没有子元素，只是解析字符数据这一事实。因此，可以调用每一数据元素的getFirstChild ( ) 方法，取得一个文本节点。文本自身身可从getNodeValue ( ) 方法得到。借助于XML已知的产品元素，可以解析产品目录XML文档并执行查询。以下为JSP页面：

```
<%@ page session="false" %>
<%@ page import="java.io.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.text.*" %>
<%@ page import="javax.xml.parsers.*" %>
<%@ page import="jspcr.xml.samples.*" %>
<%@ page import="org.w3c.dom.*" %>
<%@ page import="org.xml.sax.*" %>
<%
   long stime = System.currentTimeMillis();
%>
<HTML>
<HEAD>
<TITLE>(DOM) Clemens-Altman Musical Instruments</TITLE>
</HEAD>
<BODY>
<CENTER>
<H3>Clemens-Altman Musical Instruments</H3>
<H4>(Powered by DOM Level 1)</H4>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR>
   <TH>Product Code</TH>
   <TH>Description</TH>
   <TH>Price</TH>
</TR>
<%
   // Get a new document builder

   DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
   DocumentBuilder builder = factory.newDocumentBuilder();

   // Define the input source to be an XML document named
   // "instruments.xml" in the same directory as this JSP

   StringBuffer requestURL = HttpUtils.getRequestURL(request);
   URL jspURL = new URL(requestURL.toString());
```

精彩好书见www.book118.com


```
URL url = new URL(jspURL, "instruments.xml");
InputSource is = new InputSource(url.openStream());

// Load the document

Document document = builder.parse(is);
Element root = document.getDocumentElement();
root.normalize();

// Define currency formatter

NumberFormat fmt = NumberFormat.getCurrencyInstance();

// Select product code, description, and price
// where manufacturer = "Clemens-Altman"

for (
   Node node = root.getFirstChild();
   node != null;
   node = node.getNextSibling())
{
   // Ignore everything but product elements

   if (node.getNodeType() != Node.ELEMENT_NODE)
      continue;

   Element productElement = (Element) node;
   if (!productElement.getTagName().equals("product"))
      continue;

   // Load the product object

   Product product = new Product();
   product.load(productElement);

   // See if the manufacturer is "Clemens-Altman"

   String text = product.getManufacturer();
   if (!text.equals("Clemens-Altman"))
      continue;

   // Get the product code, price, and item name

   String code = product.getCode();
   String description = product.getDescription();
   double price = product.getPrice();
```

精彩好书见www.book118.com


```
%>
<TR>
   <TD><%= code %></TD>
   <TD><%= description %></TD>
   <TD ALIGN="RIGHT"><%= fmt.format(price) %></TD>
</TR>
<%
   }
%>
</TABLE>
<P>
<%
   long etime = System.currentTimeMillis();
   double elapsed = (etime - stime)/1000.0;
%>
<EM>Elapsed time: <%= elapsed %> seconds</EM>
</CENTER>
</BODY>
</HTML>
```

生成表格头标后，JSP页面使用JAXP方法创建DOM DocumentBuilder的一个实例：

```
DocumentBuilderFactory factory =
   DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

产品目录XML文档在一个与JSP页面相同目录下的名为instruments.xml的文件中。可以对JSP URL使用HttpUtils getRequestURL（）方法定位此文件，然后使用两个参数URL构造器得到一个XML文件的URL：

```
StringBuffer requestURL = HttpUtils.getRequestURL(request);
URL jspURL = new URL(requestURL.toString());
URL url = new URL(jspURL, "instruments.xml");
InputSource is = new InputSource(url.openStream());
```

InputSource是包装字节流、字符流或文件名的简便类。

使用文档构造器和定义的输入源，下面准备解析：

```
Document document = builder.parse(is);
Element root = document.getDocumentElement();
```

根<products>元素可从document.getDocumentElement（）方法得到。

现在，遍历<products>元素的下一级子元素搜索<product>元素。虽然此XML文档暗示不会发现任何其他元素，但实际不是这样。文本节点分隔了每个<product>块。

```
for (
      Node node = root.getFirstChild();
      node != null;
      node = node.getNextSibling())
```

精彩好书见www.book118.com

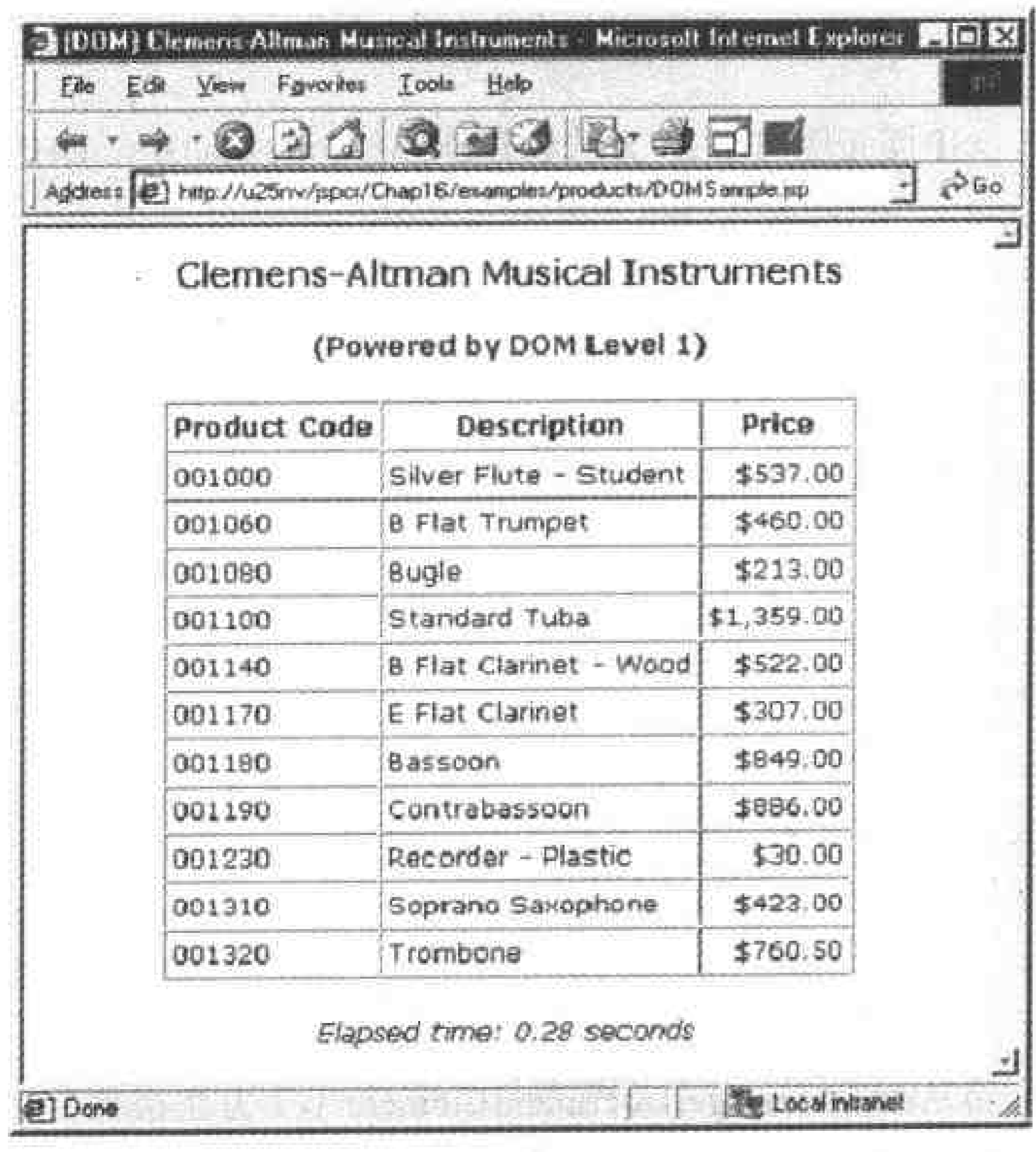

```
    {
        if (node.getNodeType() != Node.ELEMENT_NODE)
            continue;
        Element productElement = (Element) node;
        if (!productElement.getTagName().equals("product"))
            continue;
```

一旦找到一个<product>元素，就可以创建一个Product对象，并使其指导子元素查找所需内容：

```
Product product = new Product();
product.load(productElement);
```

这时，可以从Product对象中得到所需的一切。可以判断出其厂家是否为Clemens-Altman，并在得到肯定的情况下用表格打印它。结果如图16-1所示。

将DOM做解析模型的主要优点是它提供对文档结构各个部分的随意访问。但这也可能是它的最大弊端——在通过DOM API访问其任意部分前都必须读取和解析整个文档。对于大文档，负载是相当沉重的。



Product Code	Description	Price
001000	Silver Flute - Student	$537.00
001060	B Flat Trumpet	$460.00
001080	Bugle	$213.00
001100	Standard Tuba	$1,359.00
001140	B Flat Clarinet - Wood	$522.00
001170	E Flat Clarinet	$307.00
001180	Bassoon	$849.00
001190	Contrabassoon	$886.00
001230	Recorder - Plastic	$30.00
001310	Soprano Saxophone	$423.00
001320	Trombone	$760.50

图16-1 使用XML DOM解析器的产品目录搜索

16.2.2 XML的简单API

Simple API for XML (SAX)提供解析的各种方法。不是从XML文档中创建树，当解析事件

精彩好书见www.book118.com

发生时，SAX解析器通过文件读取并通知注册的侦听者。这些事件包括：

- 文档的开始。
- 读取新元素开始处的开始标签。
- 读取元素结尾处的结束标签。
- 读取元素体中的文本。
- 读取注释。
- 到达文档的结尾。

SAX接口为这些事件定义了方法。要处理特殊事件的应用可以实现一个或多个这些方法，然后注册为文档的处理器。当事件发生时，调用处理器的方法，并带有当前正被解析元素的取值。这使得SAX对需要很少或根本不用文档上下文筛选类型的应用非常适合。

像DOM一样，SAX API正不断演化。SAX 1.0产生于1998年5月XML-DEV邮件列表上的设计讨论。SAX 2.0规范发布于2000年5月。虽然SAX不是一种官方W3C规范，但已被广泛接受并在大多数解析器中伴随DOM被提供。实际上，DOM解析器经常是构建在SAX解析器之上，DOM的JAXP DocumentBuilder接口使用了几个SAX类。

1. SAX解析器

基本SAX接口是Parser。SAX API的实现提供了实现Parser的具体类。此接口定义了注册各种要使用处理器类的方法，并定义了两种形式的parse（ ）方法，如表16-5所示。

表16-5 SAX解析器接口中的方法

方 法	描 述
void parse(InputSource is) throws SAXException , IOException	使得解析器开始解析指定输入源提供的文档
void parse (String systemId) throws SAXException , IOException	使得解析器开始解析在指定系统ID中引用的文档。该系统ID可能是文件名或完全解析的URL
void setDocumentHandler (DocumentHandler handler)	为此解析器注册一个文档处理器
void setDTDHandler (DTDHandler handler)	为此解析器注册一个DTD处理器
void setEntityResolver(Entity Resolver resolver)	为此解析器注册一个实体解析器。可使用Entity Resolver以定制标签的方式定位外部实体
void setErrorHandler (Errorhandler handler	为定制的错误处理注册错误处理器
void setLocale (Locale locale) throws SAXException	指出错误和警告使用的现场

2. 处理器

有4个接口处理解析事件：

精彩好书见www.book118.com

- DocumentHandler 定义了对文档的开始和结尾、每一XML元素的开始和结尾、文档的文本、空白、注释和处理指令的回调方法。
- ErrorHandler 定义致命、可恢复和警告错误的回调。
- EntityResolver 允许外部实体的定制处理、如文档类型定义。
- DTDHandler收到文档类型定义中注释声明和未解析实体声明的通告。

以上最常采用的只有第1个，DocumentHandler。应用可以实现此接口，然后使用setDocumentHandler（）对解析器注册自己并开始接收回调。DocumentHandler中的方法在表16-6中给出。

除了4个处理器接口，SAX API还提供缺省实现——名为HandlerBase——对4个接口均满足。典型情况，应用将HandlerBase划为子类并只实现一些必需的方法。通常只包括startElement（）、characters（）和endElement（）方法。

表16-6 DocumentHandler定义的方法

方法	描述
void characters(char[] ch ,int start ,int len) throws SAXException	此方法调用针对XML解析器，读取元素文本中的字符数据。字符数据被传递到从start开始长度为len的ch数组中。简便起见，java.lang.String构造器使用这样同样的3个域
void endDocument() throws SAXException	当解析器完成对文档的解析时调用
void endElement(String name) throws SAXException	在当前元素的结束标签被解析时调用。标签名作为参数传递
void ignorableWhitespace(char[] ch , int start ,int len) throws SAXException	此方法调用针对XML解析器，读取无意义的字符数据。字符数据被传递到从start开始长度为len的ch数组中
void ProcessingInstruction (String target ,String data) throws SAXException	当遇到处理指令时调用
void setDocumentLocator(Locator locator)	通知文档处理器解析期间使用的Locator。Locator提供解析错误发生时可用的行和列号信息
void startDocument() throws SAXException	当解析器开始解析一个新文档时调用
void startElement(String name , AttributeList attrs) throws SAXException	当解析器遇到一个新元素开始标签时调用。被传递的参数包括标签名和名字/取值属性对

3. SAX用法

典型的面向SAX的应用创建SAX解析器的实例为解析文档处理器，然后调用parse（）方法开始解析并回调。在这一节，开发与DOM中相同的产品目录实例。仍然使用Product对象，但只

精彩好书见www.book118.com

保存产品属性，不进行任何解析：

```
<%@ page session="false" %>
<%@ page import="java.io.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.text.*" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.xml.parsers.*" %>
<%@ page import="jspcr.xml.samples.*" %>
<%@ page import="org.xml.sax.*" %>
<%
   long stime = System.currentTimeMillis();
%>
<HTML>
<HEAD>
<TITLE>(SAX 1.0) Clemens-Altman Musical Instruments</TITLE>
</HEAD>
<BODY>
<CENTER>
<H3>Clemens-Altman Musical Instruments</H3>
<H4>(Powered by SAX 1.0)</H4>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR>
   <TH>Product Code</TH>
   <TH>Description</TH>
   <TH>Price</TH>
</TR>
<%
   // Get a new SAX parser

   SAXParserFactory factory = SAXParserFactory.newInstance();
   SAXParser parser = factory.newSAXParser();

   // Define the input source to be an XML document named
   // "instruments.xml" in the same directory as this JSP

   StringBuffer requestURL = HttpUtils.getRequestURL(request);
   URL jspURL = new URL(requestURL.toString());
   URL url = new URL(jspURL, "instruments.xml");
   InputSource is = new InputSource(url.openStream());

   // Parse the input source

   parser.parse(is, new ProductParser(out));
%>
</TABLE>
```

精彩好书见www.book118.com


```
<P>
<%
   long etime = System.currentTimeMillis();
   double elapsed = (etime - stime)/1000.0;
%>
<EM>Elapsed time: <%= elapsed %> seconds</EM>
</CENTER>
</BODY>
</HTML>
<%!

// Inner class that parses the XML input source

class ProductParser extends HandlerBase
{

   private Product product;
   private StringBuffer buffer;
   private JspWriter out;
   private NumberFormat fmt;

   public ProductParser(JspWriter out)
   {
      this.out = out;
      buffer = new StringBuffer();
      fmt = NumberFormat.getCurrencyInstance();
   }
   /**
   * Called when a start tag is encountered
   */
   public void startElement(String name, AttributeList attrs)
      throws SAXException
   {
      if (name.equals("product")) {
         product = new Product();
         product.setCode(attrs.getValue("code"));
      }
      buffer = new StringBuffer();
   }

   /**
   * Accumulates characters from text nodes
   */
   public void characters(char[] ch, int start, int len)
      throws SAXException
```

精彩好书见www.book118.com


```
    {
       buffer.append(ch, start, len);
    }

    /**
    * Called when an end tag is encountered
    */
    public void endElement(String name)
       throws SAXException
    {
       String text = buffer.toString().trim();
       if (name.equals("price"))
          product.setPrice(Double.parseDouble(text));
       else if (name.equals("manufacturer"))
          product.setManufacturer(text);
       else if (name.equals("description"))
          product.setDescription(text);
       else if (name.equals("product")) {
          if (product.getManufacturer().equals("Clemens-Altman")) {
             try {
                String[] lines = {
                   "<TR>",
                   "<TD>", product.getCode(), "</TD>",
                   "<TD>", product.getDescription(), "</TD>",
                   "<TD ALIGN='RIGHT'>",
                   fmt.format(product.getPrice()), "</TD>",
                   "</TR>",
                   };
                   for (int i = 0; i < lines.length; i++)
                      out.println(lines[i]);
                }
                catch (IOException e) {
                   throw new SAXException(e.getMessage());
                }
             }
          }
       }
}
%>
```

像DOMBuilderFactory和DOMBuilder一样，SAXParserFactory和SAXParser可以通过JAXP调用：

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

记住，SAX技术要实现DocumentHandler（或扩展HandlerBase），并为有关的解析事件方法

精彩好书见www.book118.com

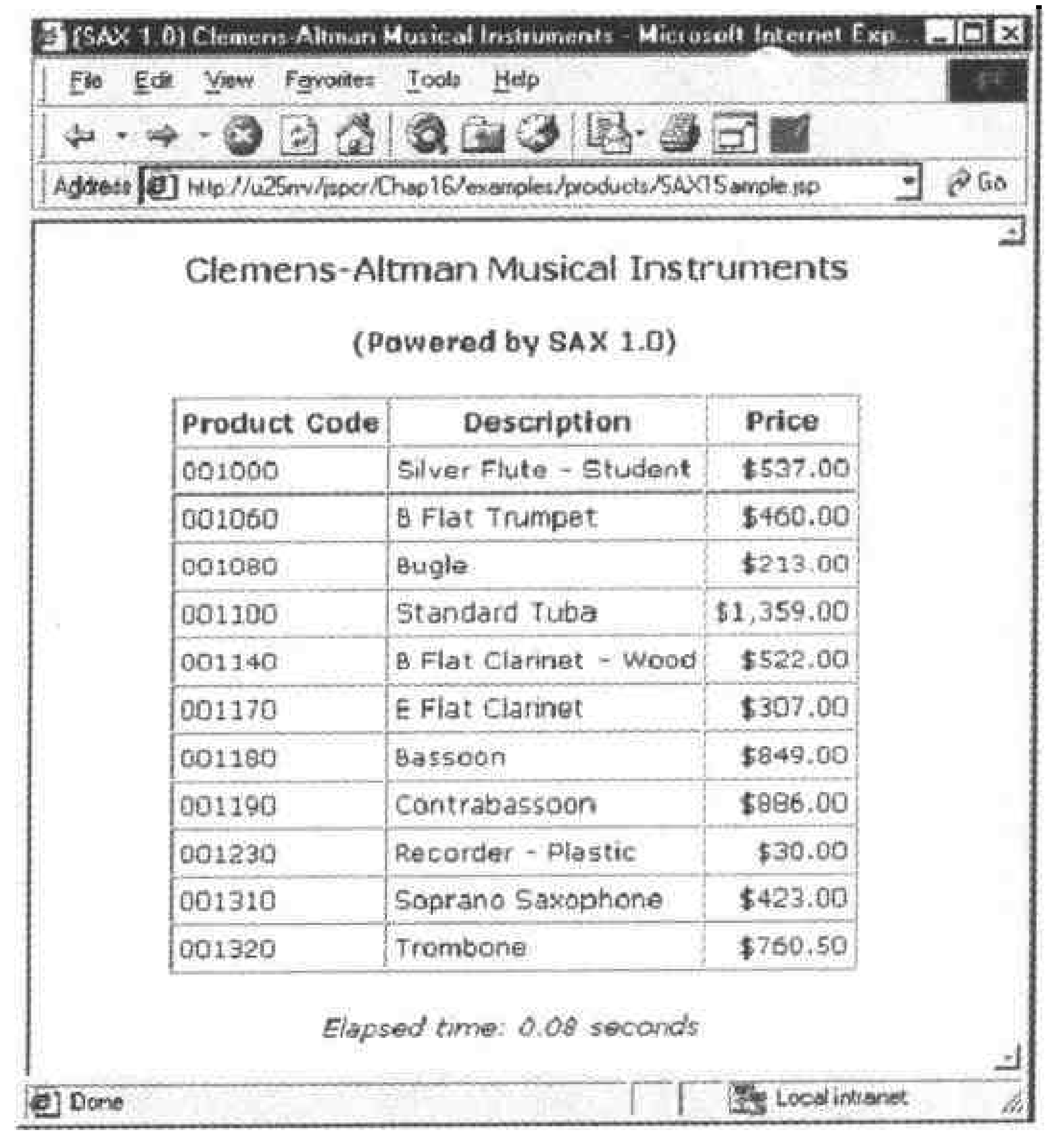
提供回调。可以使用内部类完成此功能。JAXP的parse（）方法将这里的ProductParser类注册为文档处理器，然后开始解析：

```
parser.parse(is, new ProductParser(out));
```

ProductParser类对下列3个事件有兴趣：

- startElement 在一个新<product>元素的开始，需要创建一个Product对象并在其中保存code属性。
- characters 像文本文件一样，可将其积累在StringBuffer。
- endElement 在元素的结尾，如果它是产品域之一，从StringBuffer中设置其值。如果它是<product>元素的结尾，可以在HTML表格中打印产品，清除缓存并等待要解析的下一个产品。

结果HTML表格显示如图16-2所示。注意和图16-1 DOM版本的运行时间的差别。



Product Code	Description	Price
001000	Silver Flute - Student	$537.00
001060	B Flat Trumpet	$460.00
001080	Bugle	$213.00
001100	Standard Tuba	$1,359.00
001140	B Flat Clarinet - Wood	$522.00
001170	E Flat Clarinet	$307.00
001180	Bassoon	$849.00
001190	Contrabassoon	$886.00
001230	Recorder - Plastic	$30.00
001310	Soprano Saxophone	$423.00
001320	Trombone	$760.50

图16-2 使用XML SAX 1.0解析器的产品目录搜索

4. SAX 2.0

更新的SAX规范在1998年5月被采纳。SAX 2.0中主要对名空间增强支持。名空间是带有一般前缀以和不带前缀具有相同名字的其他标签区别开的标签组。这样就允许XML标签包被定义而无需担心与类似名称的标签相混淆。

精彩好书见www.book118.com

在API术语中，SAX 2.0不支持Parser、DocumentHandler和AttributeList接口，取代的分别是已知名空间的XMLReader、ContentHandler和Attributes接口。

注意 正如所料，并不是所有的XML产品都紧跟最新的DOM和SAX产品级别。在2000年11月，除了在早期的访问包中，JAXP仍未支持SAX 2.0。Xerces-J 1.2.1版支持SAX 2.0，但并不是通过JAXP的Xerces版本，后者仍只是针对SAX 1.0，在Sun、W3C和xml.apache.org Web站点上查阅更新版本。

在SAX 2.0中，产品目录的JSP页面如下：

```
<%@ page session="false" %>
<%@ page import="java.io.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.text.*" %>
<%@ page import="java.util.*" %>
<%@ page import="jspcr.xml.samples.*" %>
<%@ page import="org.xml.sax.*" %>
<%@ page import="org.xml.sax.helpers.*" %>
<%
   long stime = System.currentTimeMillis();
%>
<HTML>
<HEAD>
<TITLE>(SAX 2.0) Clemens-Altman Musical Instruments</TITLE>
</HEAD>
<BODY>
<CENTER>
<H3>Clemens-Altman Musical Instruments</H3>
<H4>(Powered by SAX 2.0)</H4>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR>
   <TH>Product Code</TH>
   <TH>Description</TH>
   <TH>Price</TH>
</TR>
<%
   // Get a new SAX parser

   XMLReader parser = new org.apache.xerces.parsers.SAXParser();
   DefaultHandler handler = new ProductParser(out);
   parser.setContentHandler(handler);
   parser.setErrorHandler(handler);

   // Define the input source to be an XML document named
   // "instruments.xml" in the same directory as this JSP

   StringBuffer requestURL = HttpUtils.getRequestURL(request);
   URL jspURL = new URL(requestURL.toString());
```

精彩好书见www.book118.com


```
    URL url = new URL(jspURL, "instruments.xml");
    InputSource is = new InputSource(url.openStream());

    // Parse the input source

    parser.parse(is);
%>
</TABLE>
<P>
<%
    long etime = System.currentTimeMillis();
    double elapsed = (etime - stime)/1000.0;
%>
<EM>Elapsed time: <%= elapsed %> seconds</EM>
</CENTER>
</BODY>
</HTML>
<%!

// Inner class that parses the XML input source

class ProductParser extends DefaultHandler
{
   private Product product;
   private StringBuffer buffer;
   private JspWriter out;
   private NumberFormat fmt;

   public ProductParser(JspWriter out)
   {
      this.out = out;
      buffer = new StringBuffer();
      fmt = NumberFormat.getCurrencyInstance();
   }

   /**
   * Called when a start tag is encountered
   */
   public void startElement(
         String namespaceURI,
         String localName,
         String qName,
         Attributes attrs)
      throws SAXException
   {
      if (qName.equals("product")) {
         product = new Product();
         product.setCode(attrs.getValue("code"));
```

精彩好书见www.book118.com


```
      }
      buffer = new StringBuffer();
   }

   /**
   * Accumulates characters from text nodes
   */
   public void characters(char[] ch, int start, int len)
      throws SAXException
   {
      buffer.append(ch, start, len);
   }

   /**
   * Called when an end tag is encountered
   */
   public void endElement(
         String namespaceURI,
         String localName,
         String qName)
      throws SAXException
   {
      String text = buffer.toString().trim();
      if (qName.equals("price"))
         product.setPrice(Double.parseDouble(text));
      else if (qName.equals("manufacturer"))
         product.setManufacturer(text);
      else if (qName.equals("description"))
         product.setDescription(text);
      else if (qName.equals("product")) {
         if (product.getManufacturer().equals("Clemens-Altman")) {
            try {
               String[] lines = {
                  "<TR>",
                  "<TD>", product.getCode(), "</TD>",
                  "<TD>", product.getDescription(), "</TD>",
                  "<TD ALIGN='RIGHT'>",
                  fmt.format(product.getPrice()), "</TD>",
                  "</TR>",
               };
               for (int i = 0; i < lines.length; i++)
                  out.println(lines[i]);
            }
            catch (IOException e) {
               throw new SAXException(e.getMessage());
            }
         }
      }
```

精彩好书见www.book118.com


```
    }
}
%>
```

主要差别在于调用解析器的方式中：

```
XMLReader parser = new org.apache.xerces.parsers.SAXParser();
DefaultHandler handler = new ProductParser(out);
parser.setContentHandler(handler);
parser.setErrorHandler(handler);
```

以及回调方法的特征：

```
public void startElement(
      String namespaceURI, String localName, String qName,
      Attributes attrs)
   throws SAXException
...
public void endElement(
      String namespaceURI, String localName, String qName)
   throws SAXException
```

如图16-3所示，结果输出比在SAX 1.0中要快一点。

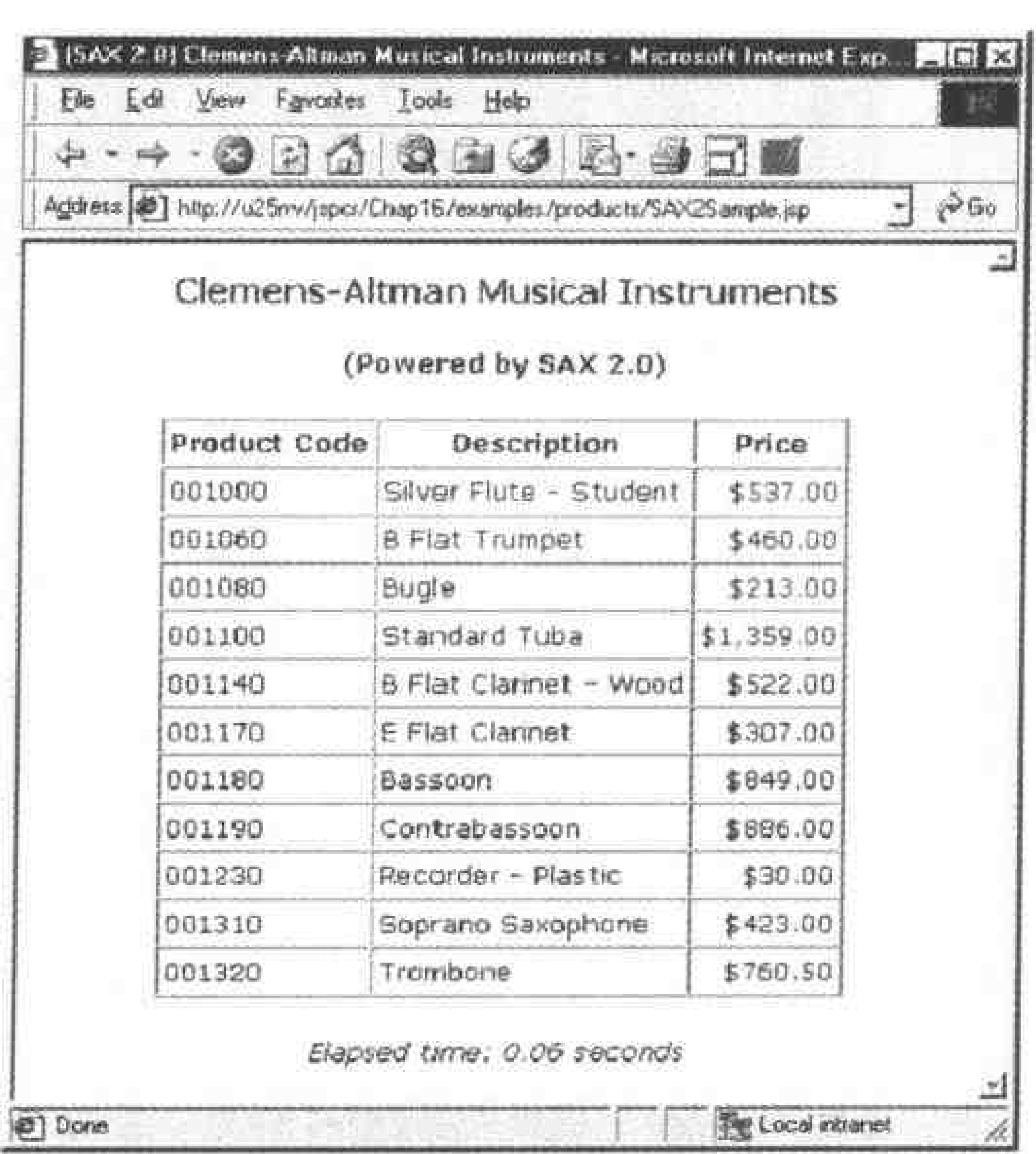


图16-3 使用XML SAX 2.0解析器的产品目录搜索

精彩好书见www.book118.com

SAX比DOM具有更多优势。它简单易学，使用内存更小并且不需要载入整个文档。几乎所有的XML解析器都有SAX接口；却很少有DOM接口。SAX还非常适合读取格式很乱的文档（如大多数HTML）。

16.3 使用XSLT进行XSL转换

前面提到，XML设计纯粹是为了标识文档结构而不是文档外观。很明显， XML和HTML很相近并且XML文档可被转换成HTML。进行转换时，可能要加入样式信息，这就是可扩展样式单语言（XSL）的任务。

XSL是用来设计样式单的一种语言。XSL样式单系统地描述了哪个格式化元素被应用到XML源文档中产生所需HTML输出的哪个元素。不奇怪，XSL样式单本身就是一个XML文档。

虽然XSL最初的意图是为样式单设计，很明显它也可以应用于一般的XML结构转换。此操作由一个XSL转换处理器（XSLT）执行。XSLT在W3C1999年11月发布的一个推荐产品中定义（见http://www.w3.org/TR/xslt.html）。流行的XSLT处理器可从Apache软件基地（Xalan）、Microsoft（MSXML）、Michael Kay（Saxon）和James Clark（XT）得到。

XSLT是一个广泛的主题，有许多书籍和文章都提到它。本书只给出一个基本介绍，其作用只是为了让用户读懂一个XSLT样式单。

XSLT使用称为XSL样式单的XML文档描述其所修改的内容和如何修改。在样式单中是一个或多个模板，用来标识要进行转换的特定XML元素，然后给出一个字母集合和嵌套的XSL语句指出输出的格式。关键的XSLT指令在表16-7中列出：

表16-7 主要的XSLT指令

指令	描述
<xsl:stylesheet>	XSL样式单中最外面的文档元素。必需属性是xmlns:xsl（XSL标签名空间）和version
<xsl:template>	标识一个模板块。可选属性为match，它指出模板匹配的XML元素。表示匹配值有很多方式。细节请查阅XSLT规范
<xsl:apply-templates>	使得处理器查看要匹配的其他元素。可选属性是select，其指定在与<xsl:template>match属性相同的语言中元素的子集
<xsl:value-of>	使得处理器替换指定元素值，可选属性是select，其操作与在<xsl:apply-templates>中相同

XSLT行为

我们可以在XSLT中完成在阐述DOM和SAX中用到的相同例子，以下是XSL样式单：

```
<?xml version="1.0"?>

<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">

<xsl:output method="html"/>
```

精彩好书见www.book118.com


```
<xsl:template match="/">
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR>
    <TH>Product Code</TH>
    <TH>Description</TH>
    <TH>Price</TH>
</TR>
<xsl:apply-templates
    select="//product[manufacturer='Clemens-Altman']"/>
</TABLE>
</xsl:template>
<xsl:template match="//product">
<TR>
    <TD><xsl:value-of select="@code"/></TD>
    <TD><xsl:value-of select="description"/></TD>
    <TD ALIGN="RIGHT">
        <xsl:value-of select="price"/>
    </TD>
</TR>
</xsl:template>

</xsl:stylesheet>
```

文档元素匹配/模板，因此在HTML表格任一边使用的HTML被编码到此模板体中。代替表格，这里有一个至XSLT处理器的回调：

```
<xsl:apply-templates
    select="//product[manufacturer='Clemens-Altman']"/>
```

select属性值指出从文档根开始的下一级，并且具有匹配Clemens-Altman值的厂家属性的任意产品元素。

因此，当文档被解析时，匹配规则的每一产品元素被传递到//product模板。此模板表示表格中的一行，它加入了<TR><TD>...</TD></TR>标签并用文档元素文本填充到其中：

```
<TD><xsl:value-of select="@code"/></TD>
```

前面一行在当前节点抽取名字为code的属性的取值。

```
<TD><xsl:value-of select="description"/></TD>
```

这一行抽出了<description>标签的文本值。

JSP页面很简单。除了生成外面的HTML，所有的工作就是创建XSLT处理器的一个实例并启动它：

```
<%@ page session="false" %>
<%@ page import="java.io.*" %>
<%@ page import="java.net.*" %>
```

精彩好书见www.book118.com


```
<%@ page import="org.xml.sax.*" %>
<%@ page import="org.apache.xalan.xslt.*" %>
<%
   long stime = System.currentTimeMillis();
%>
<HTML>
<HEAD>
<TITLE>(XSLT) Clemens-Altman Musical Instruments</TITLE>
</HEAD>
<BODY>
<CENTER>
<H3>Clemens-Altman Musical Instruments</H3>
<H4>(Powered by XSLT)</H4>
<%
   // Create an instance of the XSLT processor

   XSLTProcessor p = XSLTProcessorFactory.getProcessor();

   // Create the XML input and XSL URL's

   StringBuffer requestURL = HttpUtils.getRequestURL(request);
   URL jspURL = new URL(requestURL.toString());
   URL inURL = new URL(jspURL, "instruments.xml");
   URL xslURL = new URL(jspURL, "XSLTSample.xsl");

   // Process the stylesheet

   p.process(
      new XSLTInputSource(inURL.openStream()),
      new XSLTInputSource(xslURL.openStream()),
      new XSLTResultTarget(out)
      );
   out.flush();
%>
<P>
<%
   long etime = System.currentTimeMillis();
   double elapsed = (etime - stime)/1000.0;
%>
<EM>Elapsed time: <%= elapsed %> seconds</EM>
</CENTER>
</BODY>
</HTML>
```

结果如图16-4所示。

这个例子在服务器上运行XSL转换。在Web上传递XML和样式单以及在客户端上进行转换

精彩好书见www.book118.com

也是可能的。IE5 5.0提供了对XML和XSL样式单的直接支持。不幸的是，当Microsoft加入这些特性时，XSLT规范只是测试版形式。从此，XSLT发生了变化，许多特性已与Microsoft版本不兼容。有望在将来解决这些困难。

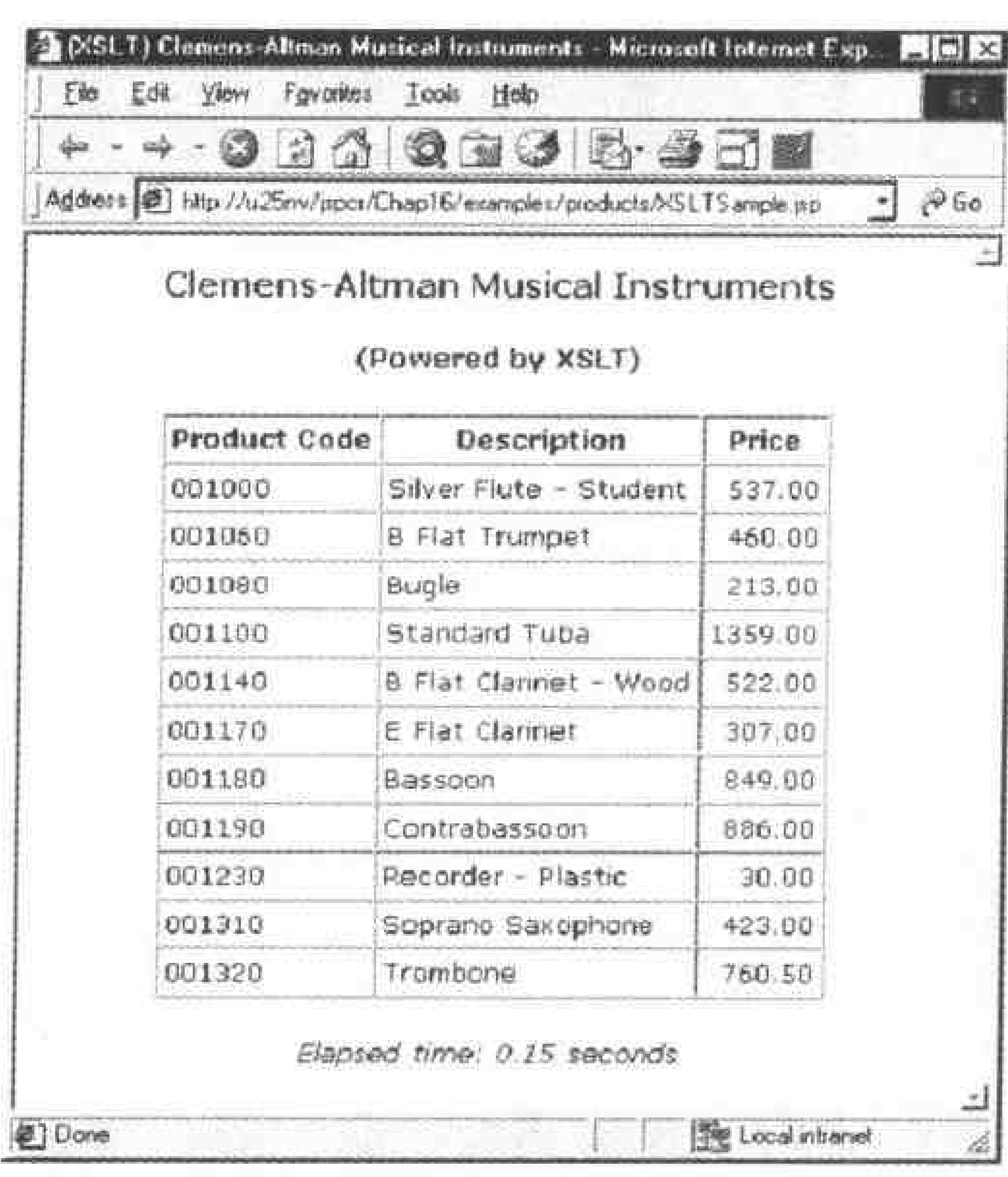


图16-4 XSLT生成的产品目录HTML

16.4 小结

XML正成为结构化数据存储和交互的通用语言，只需使用可读的文本文件和简单的语法规则。XML获取的不但是数据，还有元数据以及关于数据结构的信息。成百个应用正在被编写或转换为使用XML作为其输入和/或输出。XML规范以及其相关技术由W3C予以管理，通常作为W3C被引用。

为读取XML，需要一个解析器。常用有两种基本的解析器模型：

- 文档对象模型（DOM）将XML文档模拟成节点树。DOM API给出以任意次序导航一个DOM树的方法：向前、向后、通过兄弟节点。
- Java的简单API（SAX）在已注册处理器中调用回调方法的事件驱动的解析器模型。

可以使用XSLT处理器和XSL样式单转换XML。

毫无疑问，在将来XML应用将成倍增加，JSP能成为这些应用的一种使能技术。

精彩好书见www.book118.com

第17章 JSP测试和调试

调试技术在编程过程中常被忽视，但却是应用开发中必不可少的。编程是系统化的，代码可以很容易从其他程序移植过来，而调试常被作为可能会也可能不会解决问题的反复过程。

Web应用环境给出其独有的特殊性。因为应用可分为服务器和客户端组件，请求处理包括多个协作进程。错误的结果很难被再现，特别是如果错误是间歇性出现的话。

本章可以看出测试和调试也可以像开发一样系统化。本章概括了可应用的基本测试和调试技术，并开发了几个可用工具。

17.1 建立思想模型

系统化调试的关键是理解应用设计的工作方式。这意味着要知道包含的组件、其交互方式及其预计行为。这样才能分离出错误组件并判断出错误的原因。

转换和编译

例如，知道JSP页面存在三种形式，如图17-1所示：

1) JSP源码 这是开发商创建的一个.jsp文件，包含scriptlet、表达式、伪指令和HTML模板数据。

2) 生成的servlet源码 当一个JSP页面首次被请求或其.jsp文件发生变化后被请求时，JSP容器将其转换成一个等价的Java servlet。

3) 已编译servlet类 JSP页面被转换成servlet源码后，它被编译产生一个Java .class文件。

错误可能发生在过程的任意位置。仔细检验错误信息和中间过程存在的漏洞会有助于查找错误的起因。

图17-1 JSP页面的三种形式

注意 仔细读取错误信息很重要，但它们不出现就不能进行读取。IE5当发生一个级别为500的错误时缺省替换其本身的错误页面。大概这是为了保护终端用户免受栈跟踪的威胁，但这样使开发商变得很难办。可以通过选择工具Internet选项菜单条目关闭此特性，然后选择高级标签。在树状浏览选择下有一个名为“Show friendly HTTP error messages”的检查框，如果不选择此选项，就可以看到全部栈跟踪及一个servlet溢出发送的所有数据。

例如，假定有一个名为timer的定制标签跟踪体执行所花费的时间。该标签处理器是一个名

精彩好书见www.book118.com

为TimerTag的类，在其doStartTag（）和doEndTag（）方法中可取得当前系统时间并创建带有结果的一个脚本变量。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.net.*;
import java.util.*;

/**
* A tag handler for a custom tag that keeps track
* of how long its body takes to execute
*/
public class TimerTag extends TagSupport
{
   private long startTime;
   private long endTime;

   /**
   * Starts the timer
   */
   public int doStartTag() throws JspException
   {

      startTime = System.currentTimeMillis();
      return EVAL_BODY_INCLUDE;
   }

   /**
   * Stops the timer and calculates the elapsed time
   * in seconds. This is stored as a page context
   * attribute using the ID variable name
   */
   public int doEndTag() throws JspException
   {
      endTime = System.currentTimeMillis();
      double elapsed = (endTime - startTime)/1000.0;
      pageContext.setAttribute(getId(), new Double(elapsed));
      return EVAL_PAGE;
   }
}
```

典型用法可以看出其创建一个JDBC连接所花费的时间，如下列JSP代码所示：

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ taglib prefix="debug" uri="/WEB-INF/tlds/debug.tld" %>
```

精彩好书见www.book118.com


```
<debug:timer id="t1">
<%
   Connection con = null;
   try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      con = DriverManager.getConnection("jdbc:odbc:usda");
   }
   finally {
      if (con != null)
         con.close();
   }
%>
</debug:timer>
Connecting to the database took <%= t1 %> seconds.
```

当在JRun 3.0下运行此JSP页面时，工作正常，但在Tomcat 3.2下，则产生下列错误信息：

```
Error: 500
Location: /jspbr/Chap17/examples/Timer.jsp
Internal Servlet Error:
org.apache.jasper.JasperException:
Unable to compile class for JSP
D:\tomcat\work\localhost_8080%2Fjspbr\_0002fChap_00031_0003/
_0002fexamples_0002fTimer_0002ejspTimer_jsp_0.java:70:
Class Chap_00031_00037.examples.TimerTag not found.
      TimerTag _jspx_th_debug_timer_0 = new TimerTag();
      ^
```

仔细查看该信息，会发现几个错误特性的线索。首先，它给出了JSP页面的位置。这说明Tomcat能够找到JSP源码。接着，信息文本说明Jasper（Tomcat JSP转换器 ）不能编译一个JSP servlet类并给出生成的.java源文件的名字。这意味着从.jsp到.java的转换已完成并调用了Java编译器，但是失败了。因此可以知道这不是一个运行时错误，不是JDBC-ODBC驱动器产生的问题，因此一定是生成的servlet源码的编译错误。

将出错组件隔离开后，就明白错误信息其余部分的含义了。生成的servlet在第70行创建了一个TimeTag（ ）的实例并将其存储在_jspx_th_debug_timer_0变量中，就是这一行产生了错误信息“Class Chap_00031_00037.examples.TimerTag not found”。因此Java编译器（不是JSP转换器或servlet引擎）不能找到一个类。如果能够指出原因，目的就达到了。

编译器由于几个原因可能会找不到一个类。该类没有被编译过或其.class文件在类路径中不存在。但是要仔细检查出错信息以指出不同的原因。仔细注意编译器正在查找的类：Chap_00031_00037.examples.TimerTag。该包的名字来自何处？回到标签处理器源码，可以看出没有package语句并且在TLD中全质类名只是TimerTag。这导致类未被找到——编译器没有在正确的名字下查找它。

为什么编译器查找具有包名的一个类呢？因为这是编译问题，需要查看.java文件判断出问题来源。此文件位置与servlet引擎无关，但从错误信息中可以分辨出它在Tomcat根的work子目

精彩好书见www.book118.com

录下，沿着Web应用子目录一直下去，就会找到servlet源文件。开始几行显示了问题所在：

```
package Chap_00031_00037.examples;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
...
import org.apache.jasper.JasperException;
import java.sql.*;

public class ... extends HttpJspBase {
   ...
}
```

生成的servlet有一个package语句以及大量的import语句。从一般的Java知识来看，如果一个import语句给出了名字的其余部分，引用的类不会带有其全质包名。如果没有一个导入包且包含引用的类名，编译器假定它在与正被编译的类相同的包中。因此，生成的servlet的第70行（错误信息指出）引用的TimerTag类在每一导入包中被查找，且均未发现，然后就被认为它是servlet自身包Chap_00031_00037.examples中的一个类。这样就清楚了。

但还有两个问题：

- 为什么在JRun中工作？
- 如何解决问题？

如果看了生成servlet的JRun版本，问题就很容易回答：

```
// Generated by JRun, do not edit

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import allaire.jrun.jsp.JRunJSPStaticHelpers;
import java.sql.*;

public class jrun__Chap17__examples__Timer2ejspla
   extends allaire.jrun.jsp.HttpJSPServlet
   implements allaire.jrun.jsp.JRunJspPage
{
   ...
}
```

JRun JSP转换器并不生成一个package语句，因此生成的servlet在缺省的未命名包中，也就是标签处理器所在的同样包，因此当servlet使用未被采纳的类名时没有冲突：

```
TimerTag timer__4_1 = (TimerTag)
   JRunJSPStaticHelpers.createTagHandler
      (pageContext, "TimerTag");
```

精彩好书见www.book118.com

现在，怎么在Tomcat下解决问题呢？一种方式是为TimerTag提供一个import语句，这样Java编译器不会试图将之与其他包相联系。不必访问生成的servlet，只访问JSP源码即可。在JSP中加入下列语句：

```
<%@ page import="TimerTag" %>
```

执行以上命令并不是一种满意的方案，因为它必须在每一个使用此标签的JSP页面中执行。除了必须执行此命令的问题，对程序员来说为什么导入此类也不是很清楚——对其并不存在可视化引用。

一个更好的解决方案是对标签处理器指定一个包名。如果全质类名是jspcr.debug.TimerTag，则第70行将变成：

```
jspcr.debug.TimerTag _jspx_th_debug_timer_0
    = new jspcr.debug.TimerTag();
```

这样就不会模糊了。

注意　*隔离编译何运行时错误的一种有益的方式是预编译JSP页面。当一个带有请求参数“jsp_precompile”的JSP页面被调用时，JSP 1.1规范需要相应的JSP容器执行此操作。JSP容器将JSP页面转换为servlet源码并编译此servlet，但并不使其服务于请求。不必从一个浏览器执行此过程。可以从简单创建请求（包括“jsp_precompile”参数）的一个URL并调用其openStream（）方法的一批Java应用中执行此过程。*

17.2　独立测试

使发生错误的组件分离开是调试的关键。使这一过程简单易行也很重要。通过流程图识别出应该发生什么和实际正在发生什么是有可能的。当问题区域被分离开后，亲自测试发生错误的组件，验证操作的每一步应该是可能的。

为此，需要从已知状态开始，如果已经改动了几段代码，重新编译了某些bean，并修改了发布描述器，也许能找到更好的方案，但由于部分初始化和剩余类的作用，可能会看不出什么变化。可以如下避免上述情况：

- 删除转换JSP servlet和类的旧的复本　JSP文件只有当它比其相应的servlet和类文件新时才会被转换，但其依赖的模块可能会在不激活其重新转换的情况下发生变化。例如，包含在<%@include%>伪指令中的文件变化就不会确保包含的JSP页面被重新转换。某些JSP容器是这样，但规范并未指明。
- 删除序列化会话　某些JSP容器在关机时将会话保存到持久存储体，然后当JSP容器再次启动时重新装载它们。例如，JRun将已序列化会话写入/WEB-INF/session目录。如果改变了类并重新编译它们，当再次恢复servlet引擎时，可能拥有的是非序列化类的一个旧版本。
- 重新启动Web服务器和servlet引擎　在所有情况中这不是严格必须的，此步骤使你准确执行了所有的初始化工作。例如，对web.xml和标签库描述器的改变只有在启动时才能检测到。

一旦确定了应用状态，就可以向其提供已知的输入流并处理它。Servlet注册向主要的汇集点提供了来自servlet引擎和每一个servlet和JSP页面的信息。可以使用log（）方法向注册写入一

精彩好书见www.book118.com

个信息[1]，就像使用System.out.println（ ）一样。特别是在JSP页面中，很容易加入一些注册信息、测试以及加入一些基于测试结果的深入信息等等。向servlet注册写入信息比写入servlet输出流提供了一种更好的执行轨迹。后者可能在分析它之前就被中断或消失。

17.3 调试工具

大部分商业集成化开发环境（IDEs）都提供某种调试器使你可以步进执行一个Java类，检验并可能改变变量的取值。JDK包含一个命令行调试器名为jdb，它或多或少地执行以上功能。这些工具可能会有用，但用于调试JSP代码时有一些弊端。

首先，JSP页面与其等价的字节代码并不存在紧密的映射关系。它们可能由scriptlet、伪指令、表达式、HTML和定制标签组成。如果对按行进行跟踪有兴趣，就需要使用生成的servlet源码，而不是.jsp文件进行调试。

另外，JSP类被载入并运行于由可能链接到Web服务器的servlet引擎控制的单独的虚拟机中。为了调试单独的类，必须以调试模式启动整个servlet引擎。需要验证所有同样的类路径入口都是激活的并使用相同的端口等等。你甚至可以指出其执行方式。这使得调试环境与真正的运行时环境有很大差别。还有，在两种环境之间微小的差异可能会导致出现与正在被调试的问题无关的超时和竞争条件。

实际上，按行调试所做的工作很少，你不能用它来处理log（ ）方法（或System.out.println（ ））。在一个断点检验的任何变量都很容易被写入servlet注册信息。可以在任意点停止执行并通过简单扔出一个溢出产生栈轨迹。给定JSP自动编译和浏览器刷新按钮功能，可能会使进行带有新信息的几个循环比在调试模式下启动IDE并引入servlet引擎更快。

在这一节，将学习三种开发工具，对于HTTP请求处理环境，它们具有更少干涉和更好的适应性。结合log()方法进行跟踪，它们都能帮助隔离错误并校验调试。

17.3.1 捕获窗体参数

当使用HTML窗体发送请求参数到一个JSP或servlet时，明显需要测试的是能够知道其发送的参数及其取值。这通常是不明显的。如果一个<SELECT>元素允许多个选择，那么请求参数值是什么？如果一个TEXT输入元素未被填充，它被作为一个空白传递还是null？并未指定VALUE属性的检查框又如何？

发现以上取值的一种容易方式是使用捕获请求参数并作为表格形式的名字/取值对显示它们调试的JSP页面。下面JSP页面（Echo.jsp）给出其方式：

```
<%@ page session="false" %>
<%@ page import="java.util.*" %>
<HTML>
<HEAD><TITLE>Form Parameters</TITLE></HEAD>
```

1 这是ServletContext类中的方法，但它也可作为演绎出大部分JSP页面的GenericServlet的便捷方法被利用。log（ ）对System.out.println（ ）是可选的，因为它是厂家无关的，由所有的servlet容器提供。

精彩好书见www.book118.com


```
<BODY>
<H3>Form Parameters</H3>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR><TH WIDTH=200>Name</TH><TH WIDTH=200>Value</TH></TR>
<%
   Enumeration enames = request.getParameterNames();
   while (enames.hasMoreElements()) {
      String name = (String) enames.nextElement();
      String[] values = request.getParameterValues(name);
      if (values != null) {
         for (int i = 0; i < values.length; i++) {
            String value = values[i];
%><TR><TD><%= name %></TD><TD><%= value %></TD></TR><%
         }
      }
   }
%>
</TABLE>
</BODY>
</HTML>
```

Echo.jsp从请求对象中取得所有参数名的列表，然后循环列表并打印每一参数的名字和取值。惟一的难是参数可能有多个取值。例如，检查框组可以有同样的名字但却有不同的value属性。Servlet API考虑了这一点，提供了请求对象中的getParameterValues（）方法返回一个取值数组。

图17-2显示了带有各种类型输入元素的HTML窗体。生成该窗体的JSP页面如下：

```
<%@ page session="false" %>
<HTML>
<HEAD>
<TITLE>Job Application</TITLE>
</HEAD>
<BODY>
<H3>Please Indicate Your Qualifications</H3>

<FORM ACTION="/dailyplanet/apphandler.jsp" METHOD="POST">

<INPUT
   TYPE="hidden"
   NAME="locale"
   VALUE="<%= request.getLocale() %>"
   >
<TABLE BORDER="0" CELLPADDING="3" CELLSPACING="0">
   <TR>
      <TD>
         <INPUT TYPE="checkbox" NAME="speed">
         Faster than a speeding bullet
```

精彩好书见www.book118.com


```
            <BR>
            <INPUT TYPE="checkbox" NAME="power">
            More powerful than a locomotive
            <BR>
            <INPUT TYPE="checkbox" NAME="flight">
            Able to leap tall buildings with a single bound
            <BR>
         </TD>
      </TR>
      <TR><TD>Name: <INPUT TYPE="text" NAME="name">
      <INPUT TYPE="submit" VALUE="Submit"></TD></TR>
   </TABLE>

   </FORM>
   </BODY>
   </HTML>
```

此窗体使作业请求者使用一个检查框集描述其资历。另外，用户的现场作为一个隐藏域被捕获，因此响应可以用户指定的语言发送。通常，此窗体由/dailyplanet/apphander.jsp处理。为正确处理输入，apphandler.jsp需要知道请求参数被发送的格式。没有取得该值，怎样才能知道检查框的缺省格式，因为它们没有指定VALUE属性。

这很容易判别，通过替换ACTION="Echo.jsp"为ACTION="dailyplanet.jsp"，可以捕获窗体的输出并使用几个不同的浏览器及取值的组合测试它。如果对图17-2中的窗体取值执行此过程，将得到如图17-3的表格。

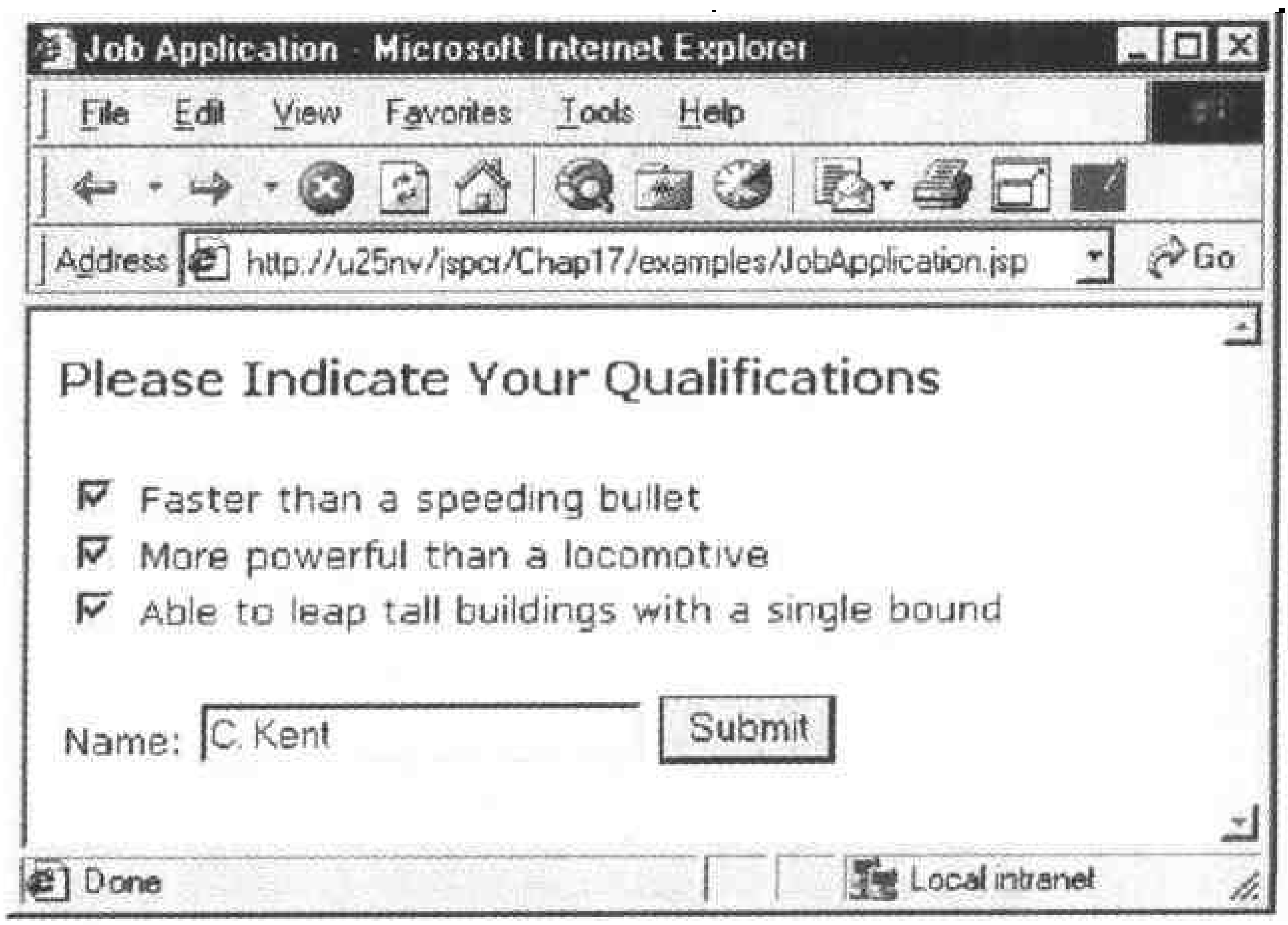


图17-2 在线作业应用

精彩好书见www.book118.com

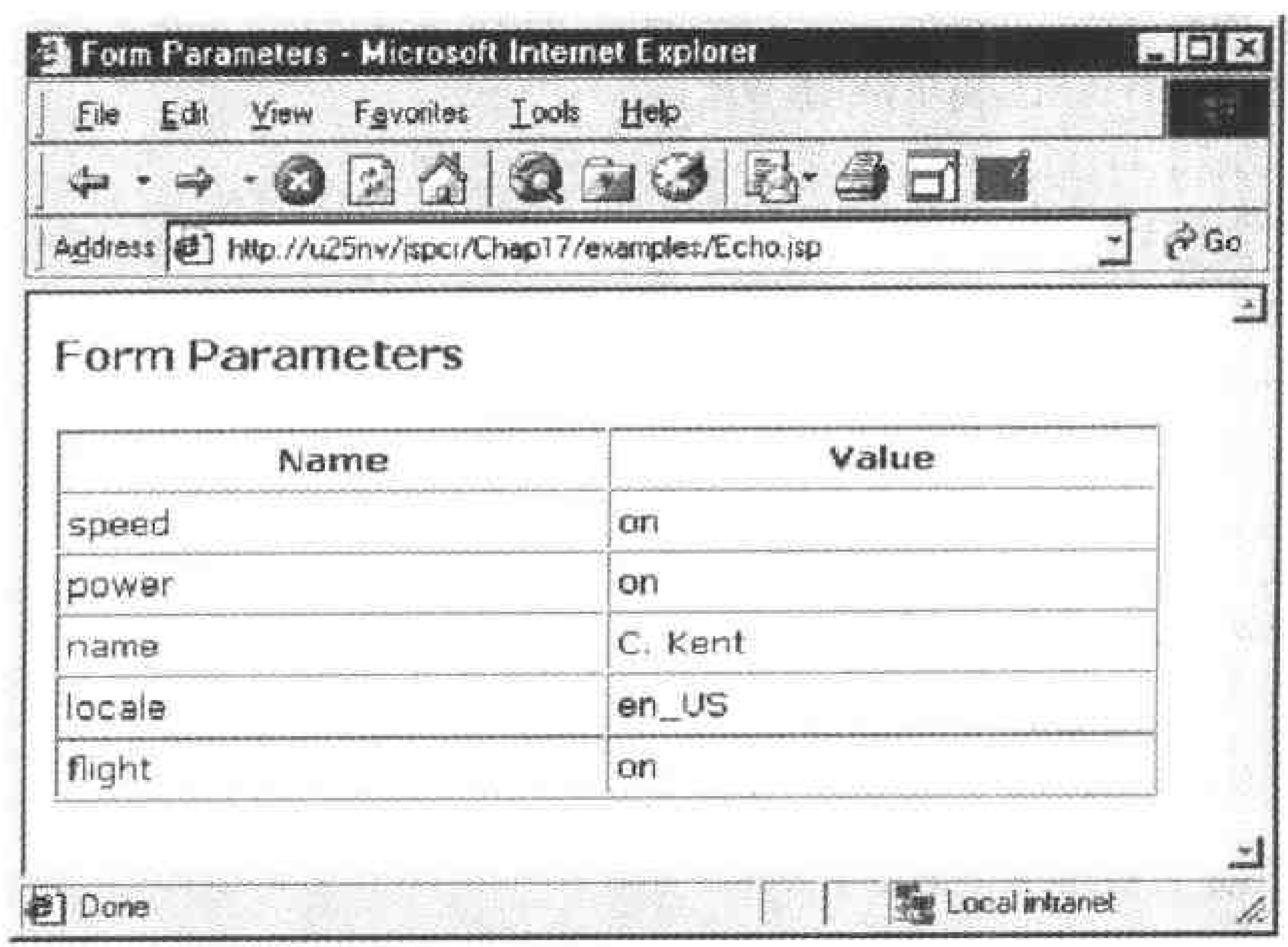


图17-3　来自在线作业应用的窗体参数

Echo.jsp可以被加强以显示关于请求的更多信息，如请求头标、cookie和请求属性。Echo.jsp的主要优点是它不需要对处理窗体的服务器端组件进行改动。一切改动就是在HTML文档或确认窗体的JSP页面中改动<FORM>元素的一行。

17.3.2　调试Web客户端

Echo.jsp服务器使你可以看出Web客户端输出的内容。事务处理的另一方面是正在处理的servlet或JSP页面的响应方式。当独立查看其输入和输出而不是在Web浏览器将其封装后再查看，这时候调试服务器端组件就会很容易。

这比预想的要容易。Web服务器不需要Web浏览器，而只需要以一般ASCII形式产生一个HTTP请求即可。在端口80上被调用的Telnet非常适合此项工作：

```
% telnet www.lyricnote.com 80
Trying...
Connected to www.lyricnote.com.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
...
```

不幸的是，在Windows系统上的缺省Telnet客户端是基于GUI的，对此用途使用起来很笨拙。在请求被处理后，GUI窗口不会滚动并自动清除其文本。Windows客户端不能正确处理Unix的行末转换。

精彩好书见www.book118.com

复制带有一个单机控制台模式Java应用的HTTP请求功能很容易。下面列出的WebClient.java是一个设计为从命令行调用的简单Web客户端：

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

public class WebClient
{
   /**
   * Mainline.
   * Reads command line parameters and creates a new
   * <CODE>WebClient</CODE> object.
   */
   public static void main(String[] args)
      throws Exception
   {
      String host = "localhost";
      int port = 80;

      for (int i = 0; i < args.length; i++) {
         String arg = args[i];
         if (arg.startsWith("-")) {
            if (arg.equals("-host")) {
               if (++i >= args.length)
                  throw new RuntimeException
                  ("no argument for " + arg);
               host = args[i];
            }
            else if (arg.equals("-port")) {
               if (++i >= args.length)
                  throw new RuntimeException
                  ("no argument for " + arg);
               try {
                  port = Integer.parseInt(args[i]);
               }
               catch (NumberFormatException e) {
                  throw new RuntimeException
                  ("Invalid port number [" + args[i] + "]");
               }
            }
            else {
               System.out.println("Invalid argument: " + arg);
               showUsage();
               System.exit(0);
            }
```

精彩好书见www.book118.com


```
        }
        else {
            showUsage();
            System.exit(0);
        }
    }

    new WebClient(host, port);
}

/**
* Displays the calling syntax
*/
public static void showUsage()
{
    String[] text = {
        "usage: java WebClient"
        + " [-host <hostName>]"
        + " [-port <portNumber>]",
    };
    for (int i = 0; i < text.length; i++)
        System.out.println(text[i]);
}

/**
* Creates and runs the web client
* @param host the HTTP server
* @param port the server port number
* @exception IOException if a socket error occurs
*/
public WebClient(String host, int port)
    throws IOException
{
    int contentLength = 0;

    // Open a socket to the web host

    Socket socket = new Socket(host, port);

    // Read input from user and echo it to web host

    BufferedReader in =

        new BufferedReader(
        new InputStreamReader(System.in));
```

精彩好书见www.book118.com


```
PrintWriter out =
   new PrintWriter(socket.getOutputStream());

// First line - request

String line = in.readLine();
out.println(line);

// Header lines

for (;;) {

   // Read and echo the line

   line = in.readLine();
   if (line == null)
      throw new IOException("Unexpected EOF");
   line = line.trim();
   out.println(line);

   // End of headers

   if (line.equals(""))
      break;

   // Otherwise, this is a header

   int p = line.indexOf(": ");
   if (p == -1)
      throw new IOException
      (line + " is not a valid header line");

   String name = line.substring(0, p).trim();
   String value = line.substring(p+1).trim();

   if (name.equalsIgnoreCase("Content-Length")) {
      try {
         contentLength = Integer.parseInt(value);
      }
      catch (NumberFormatException e) {
         throw new IOException
         ("Invalid content length " + value);
      }
   }
}
```

精彩好书见www.book118.com


```
// Read <contentLength> bytes of content

if (contentLength > 0) {
   StringBuffer sb = new StringBuffer();
   for (;;) {
      line = in.readLine();
      if (line == null)
         break;
      sb.append(line);
      int len = sb.length();
      if (len < contentLength)
         continue;
      if (len > contentLength)
         sb.setLength(contentLength);
      break;
   }

      // Write data to output stream

      out.print(sb.toString());
   }

   out.flush();

   // The server is now working on the request.
   // Read its output and dump to stdout
   in =
      new BufferedReader(
      new InputStreamReader(
      socket.getInputStream()));

   out = new PrintWriter(System.out);

   for (;;) {
      line = in.readLine();
      if (line == null)
         break;
      out.println(line);
   }

   // Close files

   in.close();
   out.close();
   socket.close();
```

精彩好书见www.book118.com


```
    }
}
```

WebClient的调用语法为：

```
java WebClient [-host <hostName>] [-port <portNumber>]
```

启动时，它打开至指定主机（缺省为localhost）的一个套接字连接，然后等待HTTP请求和从键盘输入的可选头标。用户输入的每一行都通过套接字被发送。当输入一空行时输入终止。不包含其他头标的至HTTP服务器的此信号将被发送，然后请求完成。例外情况是使用一个POST请求发送的数据。

请求被发送和处理后，服务器发回一个响应，它被反馈到控制台。一个典型的交互行为为：

```
D:\jspcr\Chap17\examples>java WebClient
POST /jspcr/Chap17/examples/Echo.jsp HTTP/1.0
Content-type: application/x-www-form-urlencoded
Content-length: 53

speed=on&power=on&flight=on&name=C.+Kent&locale=en_US
HTTP/1.1 200 OK
Date: Tue, 05 Dec 2000 03:59:35 GMT
Server: Apache/1.3.12 (Win32)
Connection: Keep-alive, close
Content-Length: 434
Content-Type: text/html; charset=ISO-8859-1

<HTML>
<HEAD>
<TITLE>Form Parameters</TITLE>
</HEAD>
<BODY>
<H3>Form Parameters</H3>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
<TR><TH WIDTH=200>Name</TH><TH WIDTH=200>Value</TH></TR>

<TR><TD>flight</TD><TD>on</TD></TR>

<TR><TD>speed</TD><TD>on</TD></TR>

<TR><TD>power</TD><TD>on</TD></TR>

<TR><TD>name</TD><TD>C. Kent</TD></TR>

<TR><TD>locale</TD><TD>en_US</TD></TR>

</TABLE>
</BODY>
</HTML>
```

精彩好书见www.book118.com

17.3.3 跟踪HTTP请求

为了有效地诊断一个Web应用，必须能够监视其进行请求和接收响应的方式。已经知道Java类可以同时充作Web客户端和服务器。在这一节，将开发一个监视工具执行两方面的功能，并充作客户端和服务器之间的中间人。如图17-4所示。当此跟踪器工具被插入一个Web应用，其服务器组件侦听HTTP请求，注册其头标，然后将其发送到真正的Web服务器。然后其客户端组件接收Web服务器的响应，注册头标并向客户端发回响应。客户端和服务器都不知道此循环中跟踪器的存在。

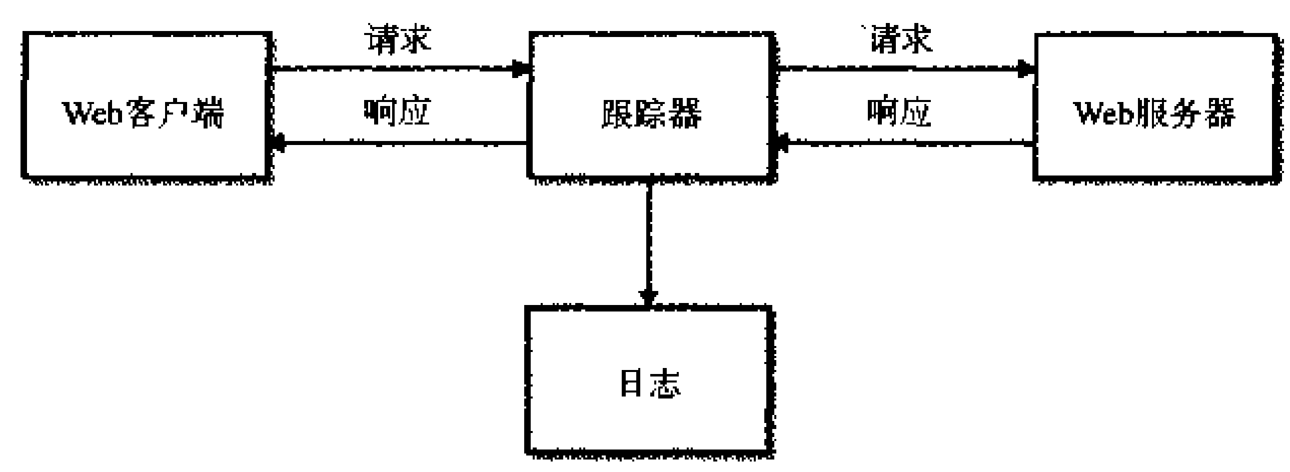


图17-4 Web应用HTTP跟踪器的配置

工具由两个主要组件组成：

- 侦听HTTP请求的Web服务器代理。
- 将客户端请求复制到服务器以及将响应复制回客户端，并在两个方向上均注册头标的请求处理器。

下面列出了第一个组件（Tracer.java）。

```
package http;

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

/**
* Acts as a proxy web server, capturing requests
* and responses and echoing the headers to a
* log stream.
*/
public class Tracer extends Thread implements Logger
{
   public static final int DEFAULT_PORT = 8601;

   private String host;
   private int port;
   private int tracerPort;
```

精彩好书见www.book118.com


```
    private PrintWriter logWriter;

    // -------------------------------------------
    // Class methods
    // ===========================================

    /**
    * Mainline
    */
    public static void main(String[] args)
       throws IOException
    {
       String opt_host = null;
       String opt_port = null;
       String opt_tracerPort = null;
       String opt_log = null;

       try {
          for (int i = 0; i < args.length; i++) {
             String arg = args[i];

             if (!arg.startsWith("-"))
                throw new IllegalArgumentException
                ("Unknown argument ["
                + arg + "]. Use -h for help");

             // -h for help

                String keyword = arg.substring(1);
                if (keyword.equals("h") ||
                   keyword.equals("help"))
                {
                   showUsage();
                   return;
                }

                // -host <hostname>

                if (keyword.equals("host")) {
                   if (++i >= args.length)
                      throw new IllegalArgumentException
                      (arg + " but no argument");
                   opt_host = args[i];
                }
                else
```

精彩好书见www.book118.com


```
        // -port <hostname>

        if (keyword.equals("port")) {
           if (++i >= args.length)
              throw new IllegalArgumentException
              (arg + " but no argument");
           opt_port = args[i];
        }
        else

        // -tracerPort <hostname>

        if (keyword.equals("tracerPort")) {
           if (++i >= args.length)
              throw new IllegalArgumentException
              (arg + " but no argument");
           opt_tracerPort = args[i];
        }
        else

        // -log <filename>

        if (keyword.equals("log")) {
           if (++i >= args.length)
              throw new IllegalArgumentException
              (arg + " but no argument");
           opt_log = args[i];
        }
        else
           throw new IllegalArgumentException
           ("Unrecognized option " + arg);
    }

    // Verify that there is no port conflict

    int testTracerPort = (opt_tracerPort == null)
       ? DEFAULT_PORT
       : Integer.parseInt(opt_tracerPort);

    int testHostPort = (opt_port == null)
       ? RequestHandler.DEFAULT_PORT
       : Integer.parseInt(opt_port);

    if (testTracerPort == testHostPort)
       throw new IllegalArgumentException
       ("Cannot assign port and tracerPort both to "
```

精彩好书见www.book118.com


```
                + testHostPort);
        }
        catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
            return;
        }

        // Create the tracer

        Tracer tracer = new Tracer();

        // Set its properties, if any

        if (opt_host != null)
            tracer.setHost(opt_host);

        if (opt_port != null)
            tracer.setPort(Integer.parseInt(opt_port));

        if (opt_tracerPort != null)
            tracer.setTracerPort
                (Integer.parseInt(opt_tracerPort));

        if (opt_log != null)
            tracer.setLogWriter(new FileWriter(opt_log));

        tracer.start();
    }

    /**
    * Displays calling syntax
    */
    public static final void showUsage()
    {
        String[] text = {
            "",
            "usage: java http.Tracer [options]",
            "",
            "where options are:",
            "",
            "-host <hostName> "
                + "(defaults to "
                + RequestHandler.DEFAULT_HOST + ")",
            "-port            <hostPort> "
                + "(defaults to "
            + RequestHandler.DEFAULT_PORT + ")",
                "-tracerPort <localPort> "
```

精彩好书见www.book118.com


```
         + "(defaults to "
            + DEFAULT_PORT + ")",
         " log          <fileName> "
            + "(defaults to System.out)",
      };
      for (int i = 0; i < text.length; i++)
         System.out.println(text[i]);
   }

   // ==========================================
   // Instance methods
   // ==========================================

   public void run()
   {
      // Set defaults if not otherwise specified

      if (tracerPort == 0)
         tracerPort = DEFAULT_PORT;

      if (logWriter == null)
         logWriter = new PrintWriter(System.out);

      // Start proxy server

      try {
         log("M: Opening tracer server on tracerPort "
            + tracerPort);
         ServerSocket server = new ServerSocket(tracerPort);

         // Loop forever

         for (;;) {

            // Wait for connection

            log("M: Waiting for connections");
            Socket client = server.accept();
            log("M: Connection received from " + client);

            // Dispatch it to a request handler thread

            RequestHandler rh = new RequestHandler(client);
            rh.setLogger(this);
            if (host != null)
               rh.setHost(host);
```

精彩好书见www.book118.com


```
                if (port != 0)
                    rh.setPort(port);
                rh.start();
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    // ==========================================
    // Implementation of Logger
    // ==========================================

    /**
    * Writes a message to the log
    * @param message the message
    */
    public synchronized void log(String message)
    {
        logWriter.println(message);
        logWriter.flush();
    }

    // ==========================================
    // Accessors
    // ==========================================

    /**
    * Returns the host.
    */
    public String getHost()
    {
        return host;
    }

    /**
    * Sets the host.
    * @param host the host.
    */
    public void setHost(String host)
    {
        this.host = host;
    }

    /**
```

精彩好书见www.book118.com


```
* Returns the port.
*/
public int getPort()
{
   return port;
}

/**
* Sets the port.
* @param port the port.
*/
public void setPort(int port)
{
   this.port = port;
}

/**
* Returns the tracerPort.
*/
public int getTracerPort()
{
   return tracerPort;
}

/**
* Sets the tracerPort.
* @param tracerPort the tracerPort.
*/
public void setTracerPort(int tracerPort)
{
   this.tracerPort = tracerPort;
}

/**
* Returns the logWriter.
*/
public Writer getLogWriter()
{
   return logWriter;
}

/**
* Sets the logWriter.
* @param logWriter the logWriter.
*/
public void setLogWriter(Writer logWriter)
```

精彩好书见www.book118.com


```
        throws IOException
    {
        this.logWriter = new PrintWriter(logWriter);
    }
}
```

Tracer主线是解析命令行，支持4个选项：

- -host <hostname> 目标Web服务器主机的名字。如果未指定，缺省为localhost。
- -port <portnumber> 目标WEB服务器上的端口号。缺省为80，这是缺省的HTTP端口号。
- -tracePort <portnumber> 跟踪器本身运行的本地端口号。缺省为8601。此端口号必须包含于要被跟踪的URL中。如果HTML窗体有http://www.lyricnote.com/search/ProductSearch.jsp的ACTION属性，那么它应该被改为http://www.lyricnote.com:8601/search/ProductSearch.jsp。这是连接跟踪器至任意应用的惟一必须要做的改变。
- -log <filename> 要写入HTTP头标的文件的名字。如果此选项未指定，日志信息写入System.out。确认命令行选项有效后，代码创建一个Tracer对象，设置其属性并启动它。

run（）方法创建了java.net.ServerSocket，开始侦听客户端连接。当收到连接时，run（）创建请求处理线程处理该事务。下面马上检验该组件。对每一步骤都编写了日志信息。

Tracer和RequestHandler都编写了日志信息。因为可以通过一个命令行选项重定向日志，每一组件都需要一个日志输出流的句柄。可通过定义Tracer实现的Logger接口完成此功能。RequestHandler作为Logger传递一个指针到Tracer。为弄清哪一组件发送信息，以C开始的信息表示客户端，S表示服务器，M表示跟踪器中间人。

工具的第2个组件RequestHandler充作至目标Web服务器的一个WEb客户端，向服务器传递请求行、请求头标和从真正的Web客户端得到的任意请求数据流、以及其发送的注册头标。然后RequestHandle转向、复制响应行、响应头标和响应数据到客户端。

```
package http;

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

/**
* A proxy HTTP server that handles a single request
*/
public class RequestHandler extends Thread
{
   public static final String DEFAULT_HOST = "localhost";
   public static final int DEFAULT_PORT = 80;

   private Socket client;
   private Logger logger;
   private String host;
   private int port;
```

精彩好书见www.book118.com


```
// ==========================================
// Constructors
// ==========================================

/**
* Creates a new <CODE>RequestHandler</CODE>
* for the specified client
*/
public RequestHandler(Socket client)
{
   this.client = client;
}

// ==========================================
// Instance methods
// ==========================================

/**
* Copies the request from the client to the server
* and copies the response back to the client.
*/
public void run()
{
   try {

      // Open a socket to the web server

      if (host == null)
         host = DEFAULT_HOST;
      if (port <= 0)
         port = DEFAULT_PORT;

      Socket server = new Socket(host, port);

      // Open I/O streams to the client

      InputStream cin =
         new BufferedInputStream(client.getInputStream());
      OutputStream cout =
         new BufferedOutputStream(client.getOutputStream());

      // Open I/O streams to the server

      InputStream sin =
         new BufferedInputStream(server.getInputStream());
```

精彩好书见www.book118.com


```
OutputStream sout =
   new BufferedOutputStream(server.getOutputStream());

// Copy request line and headers from client to server,
// echoing to logger if specified. Stop after the
// first empty line (end of headers)

int contentLength = 0;
StringBuffer sb = new StringBuffer();
for (;;) {

   // Read a byte from client
   // and copy it to server

   int c = cin.read();
   sout.write(c);

   // Ignore CR at end of line

   if (c == '\r')
      continue;

   // If LF, process the line

   if (c == '\n') {
      String line = sb.toString();
      sb = new StringBuffer();

      // Log the line

      logger.log("C: " + line);

      // If this is an empty line,
      // there are no more headers

      if (line.length() == 0)
         break;

      // If it is a content length header,
      // save the content length

      int p = line.indexOf(":");
      if (p != -1) {
         String key = line.substring(0, p).trim();
         String value = line.substring(p+1).trim();
         if (key.equalsIgnoreCase("content-length"))
```

精彩好书见www.book118.com


```
                contentLength = Integer.parseInt(value);
            }
        }

        // Otherwise, append char to string buffer

        else
            sb.append((char) c);
    }
    sout.flush();

    // If content length was specified, read input stream
    // and copy to server

    if (contentLength > 0) {
        for (int i = 0; i < contentLength; i++) {
            int c = cin.read();
            sout.write(c);
        }
        sout.flush();
    }

    // Echo the response back to the client

    sb = new StringBuffer();
    for (;;) {

        // Read a byte from server
        // and copy it to client

        int c = sin.read();
        cout.write(c);

        // Ignore CR at end of line

        if (c == '\r')
            continue;

        // If LF, process the line

        if (c == '\n') {
            String line = sb.toString();
            sb = new StringBuffer();

            // Log the line
```

精彩好书见www.book118.com


```
                logger.log("S: " + line);

                // If this is an empty line,
                // there are no more headers

                if (line.length() == 0)
                   break;
             }

             // Otherwise, append char to string buffer

             else
                sb.append((char) c);
          }
          cout.flush();

          // Copy remaining bytes to client

          int bytesCopied = 0;
          for (;;) {
             int c = sin.read();
             if (c == -1)
                break;
             cout.write(c);
             bytesCopied++;
          }
          if (bytesCopied > 0)
             cout.flush();

          // Close streams and sockets

          cin.close();
          cout.close();
          client.close();

          sin.close();
          sout.close();
          server.close();
       }
       catch (IOException e) {
          e.printStackTrace();
       }
   }

   // ==========================================
   // Accessors
```

精彩好书见www.book118.com


```
// ============================================.

/**
 * Returns the client.
 */
public Socket getClient()
{
    return client;
}

/**
 * Returns the logger.
 */
public Logger getLogger()
{
    return logger;
}

/**
 * Sets the logger.
 * @param logger the logger.
 */
public void setLogger(Logger logger)
{
    this.logger = logger;
}

/**
 * Returns the host.
 */
public String getHost()
{
    return host;
}

/**
 * Sets the host.
 * @param host the host.
 */
public void setHost(String host)
{
    this.host = host;
}

/**
 * Returns the port.
```

精彩好书见www.book118.com


```
     */
    public int getPort()
    {
       return port;
    }

    /**
    * Sets the port.
    * @param port the port.
    */
    public void setPort(int port)
    {
       this.port = port;
    }
}
```

RequestHandle的核心是其run（）方法，它打开至Web服务器的客户端套接字，然后打开套接字的输入流和输出流。同时，它为Web客户端打开输入流和输出流，然后run（）方法读取请求行和请求头标，查找标志头标结束的一个空行。当每一头标被读取时，它被注册并传递到Web服务器。如果找到一个Content-Length头标，记录下其取值。头标末尾空行后，如果内容长度非0，请求处理器从客户端输入流中读取指定长度字节并将其复制到服务器。对逆向的服务器响应也重复同样的过程，但却忽略了Content-Length头标，读取并复制服务器输出直至文件结束。

使用Tracer工具的例子是HTTP认证。此任务的许多工作既发生在浏览器，也发生在服务器范围内。检查一下HTTP头标会使这一点更清楚。

HTTP基本认证工作如下：

- Web用户请求由HTTP基本认证保护的文档。
- Web浏览器格式化HTTP请求，并将其发送到Web服务器。
- 服务器拒绝请求，设置状态码为401（需要认证）并发送一个WWW-Authenticate头标指出认证类型和范围。
- 浏览器得到401响应码，搜索其缓存看用户是否在会话期间已经注册到此范围内。如果用户没有注册，浏览器提示用户输入用户ID和口令。
- 从此提示下或浏览器会话缓存中得到的信任信息编码为Base64[1]，再次传送最初的请求，这次带有Authorization头标。
- 服务器见到Authorization头标，检验用户是否被授权接收此文档，然后返回此文档或者是又一个401响应行。

以下是Tracer对此过程所打印的内容[2]：

1 Base64编码将一个字节流转换为可读的ASCII字符，这样字节中的控制字符就不会干扰服务器操作。RFC 2068描述了该算法。然而，这不是加密，只是一种很容易逆向操作的字符转换。为此，HTTP基本认证没有特殊的安全性，应该使用在安全风险可接受的内部应用中。

2 为增加可读性将记录再次格式化。

精彩好书见www.book118.com


```
M: Opening tracer server on tracerPort 8601
M: Waiting for connections
M: Connection received from Socket
   [addr=ppp-1-247.dialup.lyricnote.com/209.165.213.47,
   port=1180,localport=8601]
M: Waiting for connections
```

跟踪器服务器通过在端口8601上打开服务器套接字开始启动。它阻塞于服务器套接字的accept ( ) 方法，等待客户端的连接。一旦收到连接，启动请求处理器，跟踪其服务器恢复侦听其他客户端请求。

```
C: GET /logviewer/index.jsp HTTP/1.1
C: Accept: application/msword, application/vnd.ms-excel, ...
C: Accept-Language: en-us
C: Accept-Encoding: gzip, deflate
C: User-Agent: Mozilla/4.0 (compatible; Windows NT 4.0)
C: Host: u25nv:8601
C: Connection: Keep-Alive
C:
```

请求处理器读取请求行和6个头标，将它们反馈到Web服务器。

```
S: HTTP/1.1 401 Authorization Required
S: Date: Tue, 05 Dec 2000 22:28:22 GMT
S: Server: Apache/1.3.12 (Win32)
S: WWW-Authenticate: Basic realm="Servlet Administrators"
S: Keep-Alive: timeout=15, max=100
S: Connection: Keep-Alive
S: Transfer-Encoding: chunked
S: Content-Type: text/html; charset=iso-8859-1
S:
```

Web服务器拒绝请求，返回一个401状态码（需要认证）以及WWW-Authenticate头标指出认证类型是Basic，范围是 “Servlet Administrators”。Web浏览器提示用户对此范围输入用户ID（“wolfgang”）和口令（“papageno”），重新发送请求：

```
M: Connection received from Socket
   [addr=ppp-1-247.dialup.lyricnote.com/209.165.213.47,
   port=1184,localport-8601]
M: Waiting for connections

C: GET /logviewer/index.jsp HTTP/1.1
C: Accept: application/msword, application/vnd.ms-excel, ...
C: Accept-Language: en-us
C: Accept-Encoding: gzip, deflate
C: User-Agent: Mozilla/4.0 (compatible; Windows NT 4.0)
C: Host: u25nv:8601
C: Connection: Keep-Alive
C: Authorization: Basic d29sZmdhbmc6cGFwYWdlbm8=
```

精彩好书见www.book118.com


```
C:

S: HTTP/1.1 200 OK
S: Date: Tue, 05 Dec 2000 22:28:37 GMT
S: Server: Apache/1.3.12 (Win32)
S: Connection: Keep-alive, Keep-Alive
S: Content-Length: 142
S: Keep-Alive: timeout=15, max=100
S: Content-Type: text/html; charset=ISO-8859-1
S:
```

这次，请求包含带有Base64-编码的信任信息的Authorization头标。这次服务器接收了请求并发回文档。

17.4 小结

测试和调试是应用开发中不可或缺少的部分。编程是系统化的，遵从于设计模式，而调试是随意的，通过实验和改变错误实现。本章重点介绍系统化调试方法的两个关键方面：

- 建立组件及其交互的思想模型。
- 分隔发生错误的组件。

给出了此方法中3种可用工具：

- Echo.jsp 捕获由HTML窗体产生参数的JSP页面。
- WebClient 模拟Web浏览器的单机应用。
- Tracer 中途截取并注册HTTP请求的单机应用。

对测试进行充分的思考和设计，调试就可能像应用开发一样系统化。

精彩好书见www.book118.com

第18章 发布Web应用

安装和配置Web应用很早就已经成为厂家的任务。例如，Apache Jserv使用.properties文件和包含Apache伪指令扩展集的文件来配置其servlet层和其属性。JRun的早期版本带有大量的.properties文件用于指出存在的服务器数量、它们使用的端口、类路径、可识别的servlet别名、预载入的servlet等等。JSWDK参考实现为此使用了一种定制的XML格式。

存在一定的差异性是不可避免的，因为servlet引擎出现带有其本身特性在各种不同实现之后，但配置任务的某部分是受限制的。描述servlet及其交互的一部分相当规则，可以被标准化，从而带来更多好处。这一点在Servlet 2.2 API中与Web应用介绍相关部分说得很清楚。

本章介绍Web应用的结构以及如何将其移出开发环境，进入产品环境。

18.1 Web应用环境

映射到Web服务器名空间的一般区域的协作资源集被Servlet 2.2和JSP 1.1 API规范称为Web应用。此集合可以包含servlet、JSP页面、HTML文件、图形、支持类和配置数据。

例如，LyricNote Web站点可以包含以下几个Web应用：

- products　包含产品目录数据库、图像、搜索引擎、针对顾客订购的购物卡应用和描述产品种类的Web页面。
- support　此应用向顾客提供JSP页面以报告出问题和请求的疑问，故障跟踪servlet、生成电子邮件的servlet和与基础知识进行交互的类。
- internal　给出MIS应用，如会议室预定，作业邮寄和公司时事通讯，是servlet、JSP页面和一般HTML的混合物。

18.1.1 目录结构

Web应用具有所有适应的servlet引擎都理解的规定的目录结构。此结构如图18-1所示。

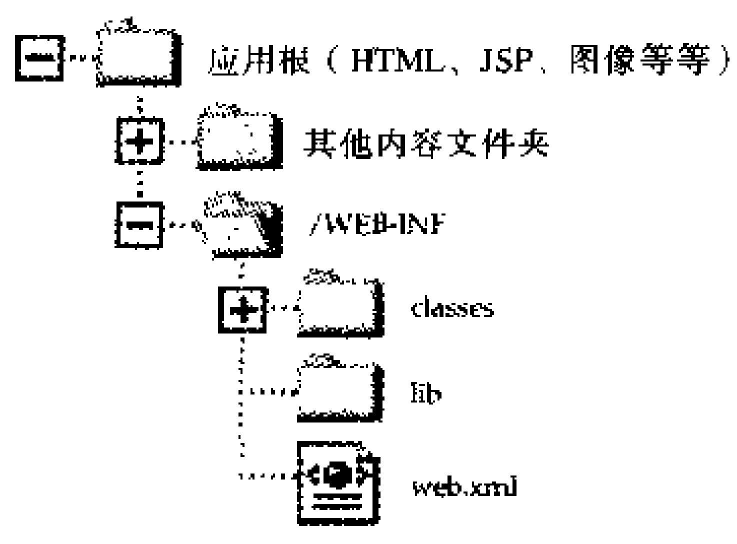


图18-1 Web应用目录结构

精彩好书见www.book118.com

最上层，即应用根，包含HTML文档、JSP页面、图形和其他组成应用内容的任意资源。根下可以有包含应用的任意数目的子目录，与一个Web服务器的文档树中的文件夹很相似。

根目录还包含一个名为WEB-INF的特殊目录。此目录及其子目录对应用用户不可视。它们包含servlet、类、.jar文件和组成应用的可操作部分的配置数据。在WEB-INF中有3个记录入口：

- classes　此目录包含servlet和其他类。这些类自动被servlet类载入器所发现，好像它们位于应用的类路径中。Classes可以有对应于包结构的子目录，与一个类路径中任何其他目录一样。
- lib　类似于classes，但包含.jar文件。此目录任意.jar文件下的类都是自动可利用于类载入器，不需要在某类路径下显式列出。
- web.xml　这是被称为发布描述器的XML文档。它有与厂家无关的严格定义的结构，用于配置组成Web应用的servlet和其他资源。本章后面会详细讨论web.xml。

虽然Servlet API规范并未定义任何特殊资源，在WEB-INF下还可以包含其他文件和子目录。常用的一个子目录是tlds，它包含JSP定制标签的标签库描述器。因为此子目录中的入口对应用类可视，但对用户不可视，WEB-INF常用于指定厂家的功能。例如，JRun创建了一个子目录jsp，包含了Java源码和从JSP页面生成servlet的已编译类。JRun还可以创建一个名为sessions的子目录，它保存当servlet引擎关闭时仍处于激活状态的任何序列化版本HTTP会话中。通常，WEB-INF适合于在Web应用中要用到的对用户的直接访问隐藏起来的任意数据。

18.1.2　资源映射

Web服务器有一个基本包含HTML文件的文档根目录。例如，在Apache中，它是<apache root>/htdocs。Microsoft Internet information Server（IIS）则使用inetpub/wwwroot。当在Web浏览器中点击URL，浏览器将其分解到它的服务器和路径组件，并生成一个对服务器指定资源的HTTP请求。当Web服务器收到请求时，从请求头标中抽取路径，将其转换成相对文档根目录的一个路径。例如，如果URL是：

```
http://www.lyricnote.com/products/index.html
```

浏览器打开至www.lyricnote.com主机的HTTP连接，并发送带有下列行的请求：

```
GET /products/index.html HTTP/1.0
```

如果Web服务器是Apache，安装于/usr/local/Apache，那么被发回的文件是：

```
/usr/local/Apache/htdocs/products/index.html
```

如果服务器是Microsoft IIS，安装于c:\inetpub,那么被请求的文件是：

```
c:\inetpub\wwwroot\products\index.html
```

正如所见，Web应用也可以有文档根，但此根可以在文件系统的任何位置。当servlet引擎识别出servlet或JSP页面的HTTP请求时，它从URL中抽出Web应用名并映射其余部分到

精彩好书见www.book118.com

URL所指应用的一个资源。例如，如果一个URL为：

```
http://www.lyricnote.com/products/contest/rules.jsp
```

那么servlet引擎为/contest/rules.jsp创建一个请求并将其传递给products Web应用[1]。Servlet的处理情况类似。一个如下URL：

```
http://www.lyricnote.com/products/servlet/Counter
```

作为Counter servlet的一个请求被传递到products Web应用[2]。

应用中servlet或JSP页面中使用的指向另一应用中资源的URL不使用应用名。例如，如果rules.jsp页面需要动态包含Counter servlet的输出，它使用如下语句：

```
<jsp:include page="/servlet/Counter" flush="true"/>
```

而不是语句：

```
<jsp:include page="/products/servlet/Counter" flush="true"/>
```

同样的情况也应用于<jsp:include>、<%@include%>伪指令以及创建RequestDispatcher对象的<%@taglib%>伪指令和方法所使用的URL。解释这些相对URL的规则是：

- 如果URL以“/”开头，它被解释成相对于文档根目录。
- 如果不以“/”开头，它被解释成相对当前JSP页面。

然而这样有一点困难。在一个JSP页面中用作超级链接、窗体行为、样式单链接或图形资源的URL被浏览器解释，而不是服务器。如果LyricNote主页面是一个带有链接到/products/contest/rules.jsp的JSP页面，那么链接就不能使用products应用名硬编码。为什么呢？因为应用名不是必须进入products。这完全依赖于系统管理员选择安装Web应用的位置。它可能作为product_test或staging_area或其他位置被登录，只有当应用真正运行时，一个JSP页面才能知道从request.getContextPath()方法中得到的名字。

可以有几种方式避开此问题。JSP页面可以将每个URL写为连接request.getContextPath（）与URL其余部分的JSP表达式，这样代码不必要地产生了散乱。一个更明智的方法是使用HTML<BASE>元素设置页面的一个上下文：

```
<HTML>
<HEAD>
<BASE HREF="http://www.lyricnote.com/products/">
</HEAD>
<BODY>
<A HREF="contest/rules.jsp">View the contest rules</A>
</BODY>
</HTML>
```

---

1 严格地说，URL中组成Web应用名的部分可以是多层字符串，如/products/test/Wednesday，但更常用的是使用一个单一记号。

2 发布描述器的<servlet-mapping>元素可用于将任何URL子字符串映射到一特定的servlet。下面给出的是缺省映射。

精彩好书见www.book118.com

使用<BASE>语句，Web浏览器可以解释任何相对于HREF属性的URL。这样，contest/rules.jsp就变成了http://www.lyricnote.com/products/contest/rules.jsp。

当然，这样还不够，还需要在运行时指出BASE HREF。HREF属性需要成为一个完整的URL，而不是来自服务器的一个绝对路径，因此需要做许多工作。如果连接使用的是SSL，URL可能会以https开头，端口号可能存在，不只是缺省端口80。幸运的是，request对象可以提供所有这些信息。下面scriptlet以一种常用的方式解决了问题：

```
<%
   String scheme = request.getScheme();
   String server = request.getServerName();
   int port = request.getServerPort();
   String path = request.getContextPath();

   StringBuffer sb = new StringBuffer();
   sb.append(scheme);
   sb.append("://");
   sb.append(server);
   if ((port != -1) && (port != 80)) {
      sb.append(":");
      sb.append(port);
   }
   sb.append(path);
   sb.append("/");
   String baseURL = sb.toString();
%>
<BASE HREF="<%= baseURL %>">
```

18.1.3 servlet上下文

在一个Web应用中，servlet和JSP页面可以通过称为servlet上下文的常用对象共享数据和功能。它是实现了javax.servlet.ServletContext接口的对象。Servlet上下文服务于下列用途：

- 对象共享　servlet和JSP页面都可以在servlet上下文通过名字保存对象，因此它们可以被其他servlet和JSP页面检索。这使得servlet和JSP页面的持续性可以和应用一样长。
- 初始化参数　应用范围内使用的常量可以在发布描述器中被指定并通过servlet上下文中的方法进行访问。这将允许配置细节——如数据库URL和驱动器类名——在任何已编译的Java代码外被指定。
- 请求发送　servlet可以发送请求到其他的servlet和JSP页面或者是在当前输出流中包含它们的输出。Servlet上下文提供使用路径或servlet名创建请求发送的方法。
- 消息注册　servlet上下文可以访问servlet注册，可用于同厂家无关的方式编写消息。

Servlet上下文提供的方法的完整描述请见表4-8。

在一个JSP页面中，servlet上下文对象在应用隐含对象中是自动可利用的。在servlet中，可通过getServletContext（）方法得到。

精彩好书见www.book118.com

原书缺页

精彩好书见www.book118.com


```
adding: sounds/BIRDS.MID(in = 3494) (out= 1138)(deflated 67%)
adding: sounds/ITBGON.MID(in = 975) (out= 478)(deflated 50%)
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/lib/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/web.xml(in = 46) (out= 37)(deflated 19%)
```

关于jar工具的完整细节查看JDK文档。

结果文件products.war准备发布。所有的Servlet 2.2兼容的servlet引擎都需要直接接受一个.war文件并构建相应的Web应用。正如所料，安装文件的特定方式是厂家指定的。Tomcat，允许简单地将.war文件安装到<tomcat_home>/webapps目录。当再次启动Tomcat时，.war文件被解包，并确认有效，新的应用就可以利用了。通常商业servlet引擎为此提供一个GUI管理工具。JRun在其管理控制台中为从一个.war文件发布应用提供发布向导，如图18-3所示。

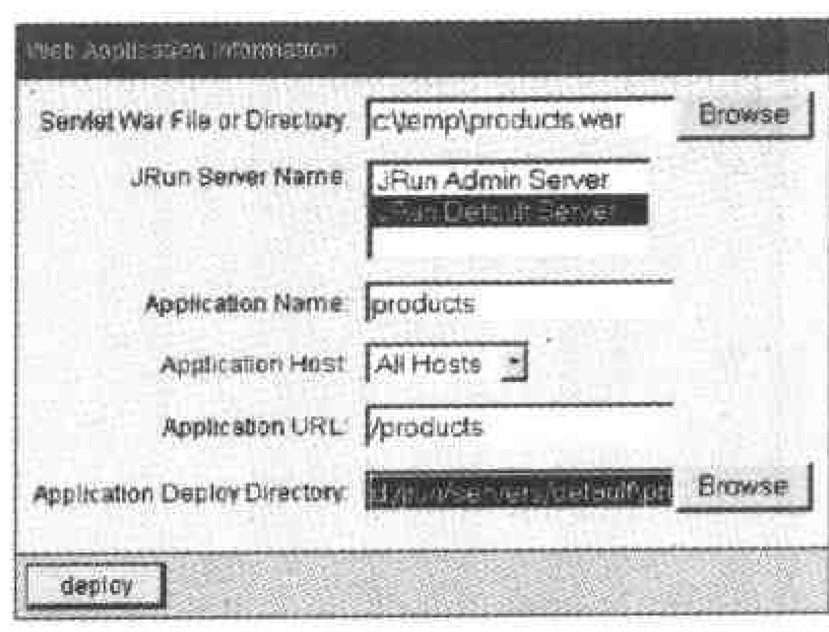


图18-3 JRun发布向导

18.3 发布描述器：web.xml

WEB-INF目录下的web.xml文件称为发布描述器。这是指定Web应用配置的格式定义严格的XML文档。结合其他内容，它可以用来描述：

- Servlet别名，映射和初始化参数。
- 会话超时限制。
- 整个应用可利用的全局参数。
- 安全配置。
- Mime类型。

因为此文件是一个XML文档，其格式可用一个文档类型定义或称DTD描述。其DTD称为web-app_x.y.dtd，这里x，y是servlet API规范的版本，如2.2。它由Sun微系统发布，可从http://java.sun.com/j2ee/dtds/web-app_2_2.dtd下载。

最简单的web.xml文件如下：

```
<?xml version="1.0"?>
```

精彩好书见www.book118.com


```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
</web-app>
```

在<web-app>体中是描述应用配置的其他元素。下表列出了可用元素。

注意　在<web-app>体中使用的任何元素都必须按排列的次序被准确指定。如果允许一个元素多次重复出现，它们必须一起出现，不能和其他元素混淆。例如，所有的<servlet>元素都必须出现在<servlet-mapping>元素之前。

web.xml发布描述器的内容

元　素	内　容
<web-app>	这是最上层元素，可以包含下述任意子元素。次序如下： <icon>（可选） <display-name>（可选） <description>（可选） <distributable>（可选） <context-param>（0或多个） <servlet>（0或多个） <servlet-mapping>（0或多个） <session-config>（可选） <mime-mapping>（0或多个） <welcome-file-list>（可选） <error-page>（0或多个） <taglib>（0或多个） <resource-ref>（0或多个） <security-constraint>（0或多个） <login-config>（可选） <security-role>（0或多个） <env-entry>（0或多个） <ejb-ref>（0或多个）
<icon>	允许开发者指定JPEG或GIF格式的图标文件应用中的相对位置。这些图标可被一个GUI管理用于标识应用，它可以包含下列元素之一或全部： <small-icon>（可选） <large-icon>（可选）
<small-icon>	一个16×16图标图像文件名
<large-icon>	一个32×32图标图像文件名
<display-name>	管理工具使用的父元素（<web-app>或<servlet>）的缩写名
<description>	管理工具使用的父元素的描述。此元素可在文件中几个不同的上下文中出现
<distributable>	如果指定，表明此应用设计为运行于多个分布式servlet容器中。
<context-param>	定义一个应用范围内的初始化参数。包含下列子元素： <param-name>（必须）

精彩好书见www.book118.com

（续）

元 素	内 容
	<param-value>（必须） <description>（可选）
<param-name>	参数名
<param-value>	参数值
<servlet>	定义一个servlet和其所有相关的配置，可以按次序包含下列子元素： <icon>（可选） <servlet-name>（必须） <display-name>（可选） <description>（可选） <servlet-class>或<jsp-file>（必须指定其中一个） <init-param>（0或多个） <load-on-startup>（可选） <security-role-ref>（0或多个）
<servlet-name>	Servlet容器已知的一个servlet的名字。注意，带有不同servlet名字的多个<servlet>元素可以指定相同的servlet类。这种情况下，servlet引擎创建servlet的多个实例。一个servlet可以调用GenericServlet或ServletConfig中的getServletName（）方法判断其名字
<servlet-class>	Servlet类的全质域名
<jsp-file>	一个JSP文件相对于Web应用根的完整路径
<init-param>	定义一个servlet初始化参数。包含下列子元素： <param-name>（必须） <param-value>（必须） <description>（可选）
<load-on-startup>	如果指定，此元素指出当servlet引擎启动时应预先载入servlet。这表明servlet的init（）方法将被调用，然后请求可利用servlet。如果servlet存在，此元素值（如果给出）可以指定此servlet启动的相对次序的一个整数值
<servlet-mapping>	指定应该被映射到servlet名字的URL模式。必须包含下列子元素： <servlet-name>（必须） <url-pattern>（必须）
<url-pattern>	指出将被调用的servlet的URL的一个子字符串必须匹配的模式。此模式可以包含*字符
<session-config>	定义会话配置参数。可以包含<session-timeout>子元素
<session-timeout>	在HTTP会话终止前servlet引擎允许处于非激活状态的缺省分钟数
<mime-mapping>	定义一个文件扩展名隐含的MIME类型。必须包含下列元素： <extension>（必须） <mime-type>（必须）
<extension>	指出其类型的文件名后缀。例如，png用来表示一个可移殖的网络图形文件
<mime-type>	与特定文件扩展名相关的MIME类型。例如，image/png用来指出一个可移殖的网络图形文件
<welcome-file-list>	0或多个<welcome-file>元素的列表当向一个为目录的URL发出请求时，servlet引擎依次试验每个欢迎文件

精彩好书见www.book118.com

（续）

元　素	内　容
<welcome-file>	如果在URL中未指定文件名，用于在一个目录中为请求提供服务的缺省文件。例子如index.html或index.jsp
<taglib>	用于定义映射到一个JSP标签库的URL。必须包含下列子元素： <taglib-uri>（必须） <taglib-location>（必须）
<taglib-uri>	在一个<%@taglib%>伪指令的uri属性中使用的字符型字符串。虽然常在这里指定一个URL或路径，不必指向一个真正的Web资源。它只充做JSP页面用来将标签库伪指令映射到标签库描述器（TLD）的惟一标识
<taglib-location>	可以发现标签库描述器（TLD）的Web应用根目录的相对URL。例如，/WEB-INF/tlds/mytags.tld
<error-page>	将一个HTTP响应代码或溢出类型映射到一个servlet、JSP页面或当发生错误时缺省调用的HTML文件。包含下列子元素： <error-code>或<exception-type>（必须选择其中一个） <location>（必须）
<error-code>	映射到错误页面的一个HTTP响应代码
<exception-type>	一个全质Java溢出类名
<location>	用做一个错误页面的servlet、JSP页面或HTML文件的URL
<resource-ref>	包含设置一个J2EE资源工厂的信息。可包含下列子元素： <description>（可选） <res-ref-name>（必须） <res-type>（必须） <res-auth>（必须）
<res-ref-name>	指出资源工厂引用的名字
<res-type>	指出与一个资源工厂相关的数据源的Java类名
<res-auth>	指出资源工厂提供证书的源。存在两个可能值：SERVLET——Web应用编程给出该值，CONTAINER——容器给出的证书
<security-constraint>	定义应用于1个或多个资源集合的安全性约束。可以包含下列子元素： <web-resource-collection>（1个或多个） <auth-constraint>（可选） <user-data-constraint>（可选）
<web-resource-collection>	定义Web应用中使用了安全性约束的资源集。可以包含下列子元素： <web-resource-name>（必须） <description>（可选） <url-pattern>（0或多个） <http-method>（0或多个）
<web-resource-name>	引用一个Web资源的名字
<http-method>	一个HTTP方法类型（例如，GET、POST等）
<user-data-constraint>	指出在应用间进行转换的数据的保护方式。可以包含下列子元素： <description>（可选） <transport-guarantee>（必须）
<transport-guarantee>	允许值为NONE——应用不需传输确认。INTEGRAL——数据在转换过程中不能改变。CONFIDENTIAL——数据在转换过程中不能读取

精彩好书见www.book118.com

（续）

元　素	内　容
<auth-constraint>	指定在一个<security-constraint>元素中集中对待的角色名列表。可以包含下列子元素： <description>（可选） <role-name>（0或多个）
<role-name>	用于标识已认证用户可以被注册角色的名字。此值与允许不同角色用户按条件执行一个servlet的一部分的request.isUserInRole()方法中指定的值相同
<login-config>	指定注册配置类型。可以包含下列子元素： <auth-method>（可选） <realm-name>（可选） <form-login-config>（可选）
<realm-name>	在HTTP基本认证中使用的域名
<form-login-config>	指定在基于窗体的注册中使用的资源。必须包含下列子元素： <form-login-page>（必须） <form-error-page>（必须）
<form-login-page>	指出提示用户输入用户名和口令的资源（HTML文件、JSP页面、servlet）名。此页面必须符合下列需求： 1) 窗体必须使用METHOD="POST" 和ACTION="j_security_check" 2) 用户名域必须命名为j_username 3) 口令域必须命名为j_password
<form-error-page>	指定当基于窗体的注册未成功时显示的资源（HTML文件、JSP页面、servlet）名
<auth-method>	指定使用的认证方法。合法值有4个： BASIC DIGEST FORM CLIENT-CERT 不是所有的servlet引擎都支持以上所有方法
<security-role>	声明在一个<security-constraints>元素中使用有效的安全角色名。可以包含下列子元素： <description>（可选） <role-name>（必须）
<security-role-ref>	创建角色名和其别名之间的映射。可以包含下列子元素： <description>（可选） <role-name>（必须） <role-link>（必须） 这样将允许servlet在request.isUserInRole（）方法中使用角色链接并使该名字等价于实际角色名。因此，如果应用被修改使用了一个不同的角色名，servlet不需改动
<role-link>	一个servlet用来引用一个实际角色名的符号名
<env-entry>	用于定义J2EE环境入口。可以包含下列子元素： <description>（可选）

精彩好书见www.book118.com

(续)

元　素	内　容
	<env-entry-name>（必须）
	<env-entry-value>（可选）
	<env-entry-type>（必须）
<env-entry-name>	相对于JNDI java:comp/env上下文的J2EE环境入口名
<env-entry-value>	J2EE环境入口值
<env-entry-type>	必须为下面值之一：
	java.lang.Boolean
	java.lang.String
	java.lang.Integer
	java.lang.Double
	java.lang.Float
<ejb-ref>	定义一个企业Java Bean（EJB）的引用。可以包含下列子元素：
	<description>（可选）
	<ejb-ref-name>（必须）
	<ejb-ref-type>（必须）
	<home>（必须）
	<remote>（必须）
	<ejb-link>（可选）
<ejb-ref-name>	一个EJB引用的JNDI名
<ejb-ref-type>	EJB的Java类
<home>	EJB本地接口类的全质名
<remote>	EJB远程接口类的全质名
<ejb-link>	在一个链接到EJB的拥有J2EE的应用中EJB的名字

18.4　发布描述器示例

表18-1看起来有点望而生畏，但幸运的是，大多数发布描述器只使用可能元素的一小部分。下面给出了典型的web.xml文件：

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

    <context-param>
        <param-name>JDBC.DRIVER</param-name>
        <param-value>
            org.enhydra.instantdb.jdbc.idbDriver
```

精彩好书见www.book118.com


```
        </param-value>
    </context-param>

    <context-param>
        <param-name>JDBC.URL</param-name>
        <param-value>
            jdbc:idb:d:/lyricnote/WEB-INF/db.prp
        </param-value>
    </context-param>

    <servlet>
        <servlet-name>Sample</servlet-name>
        <servlet-class>
            jspcr.servlets.SampleServlet
        </servlet-class>
        <init-param>
            <param-name>message</param-name>
            <param-value>Hello, world</param-value>
        </init-param>
    </servlet>

    <servlet>
        <servlet-name>daytime</servlet-name>
        <servlet-class>
            jspcr.services.daytime
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

</web-app>
```

此发布描述器包括4个元素：两个上下文参数和两个servlet声明。上下文参数定义对所有Web应用中servlet和JSP页面可利用的常量。这里定义了JDBC驱动器类和数据库URL。Servlet和JSP页面可以使用servlet上下文的getInitParameter（）方法检索这些值。因为这类信息经常变动，典型情况，将随着不同的安装而发生改变。能够描述它而不在Java类中进行硬编码是很方便的。

定义了两个servlet。一个是Sample，指向jspcr.servlets.SampleServlet类，并有一个初始化参数。允许使用如下所示的URL调用该servlet：

```
http://www.lyricnote.com/products/servlet/Sample
```

不需指定完整的servlet类名。第二个servlet名为daytime，使用<load-on-startup>元素使得适当servlet在引擎启动时被预先载入。

Servlet 2.2 API规范还提供其他的示例发布描述器。同样，大部分servlet引擎使用此文件示例的描述器。

精彩好书见www.book118.com

18.5 小结

随着Servlet 2.2规范的出现，Web应用发布已经标准化并与厂家无关。此规范描述了一个包含Web内容标准的目录结构，以及配置信息和类目录。配置在名为web.xml的称为发布描述器的XML文档中被指定。目录结构在web压缩（.war）文件格式中被镜像。发布Web应用，将其从servlet引擎移到另一个中比安装.war文件并调用servlet引擎的发布工具操作更为便利。

精彩好书见www.book118.com

第19章 事例分析：一个产品支持中心

这里假想的Internet音乐商店，LyricNote.com，售卖各种音乐产品：音乐唱片、有关音乐的书籍、礼物和音乐软件。对这些产品的支持包括电话定购、检测定购状态、解决付款问题和对软件提供技术支持。最后一项是本章讨论的焦点。

在这个事例分析中，开发管理产品支持中心是基于Web的系统。系统用户可以报告和跟踪产品缺陷及其记录注释并将其路由到适当的组织。

为清晰阐述，此应用不包括所有的有效性检验、用户控件或一个真实系统可能会有的管理报告。基本给出了本书讲过的各种技术并为进一步开发提供了模式。

19.1 过程流

开始考虑一下系统进行操作的环境。过程流如图19-1所示。

当顾客打电话报告软件问题时，电话中心代理回复该电话。如果问题与软件不相关，电话中心代理将电话路由到售卖中心或顾客服务中心。否则，代理检验顾客是否为授权提供支持，意即顾客是指定产品的有效购买者。代理创建一个问题报告并告诉顾客产品支持中心的电话。

对报告有缺陷的产品，问题报告被路由到产品支持专家。每个产品支持专家都有打开的问题报告的一个序列，当收到新的问题时，专家打电话给顾客以获得更详细的信息，试图判断出这是顾客问题或是代码问题。顾客问题包括缺乏必须的软件或硬件或是安装产品不正确。在这些情况下，产品支持人员帮助顾客解决问题到尽可能的程度，然后结束问题报告。

如果问题与代码相关，可能其他顾客也会遇到它并已经是固定存在的。如果是这样，问题在知识库中被归档，产品支持人员可从中通过适当的关键字进行搜索。补丁或解决问题所需要的过程通过电子邮件被发送到顾客，或通过Web使其可以得到。

如果问题在知识库中未发现，它被路由到作为产品基本支持被列出的开发者。开发者分析问题并试图再造它。产品可能工作正常，正如所设计的一样，这种情况下，缺陷被再次路由回产品支持，并标记“不是一个故障”。否则，开发者隔离故障并研究出一个解决方案。对该解决方案进行单元测试后，开发者将问题路由到质保中心。问题报告可被修改以表明如何再造该问题以及在何处得到解决问题所需的代码补丁。

产品的质保支持人员从开发者处收到问题报告，对解决方案进行测试。这是一个集成测试，检验新的代码对已存在系统的影响。如果方案引入了其他问题或不能通过所有已建立的测试情况，问题被再次路由到开发者。否则，方案被路由到产品支持处，这样就可以与顾客取得联系并向其提供新的代码。

在任意点上，系统都可以查询一个特定问题的状态，向其加入注释并将问题路由到下一地点。最有效的情况是每次路由都通过发送到新的问题所有者的电子邮件完成。这里没有开发这一部分，其内容请参阅讨论Java Mail API，在第21章。

精彩好书见www.book118.com

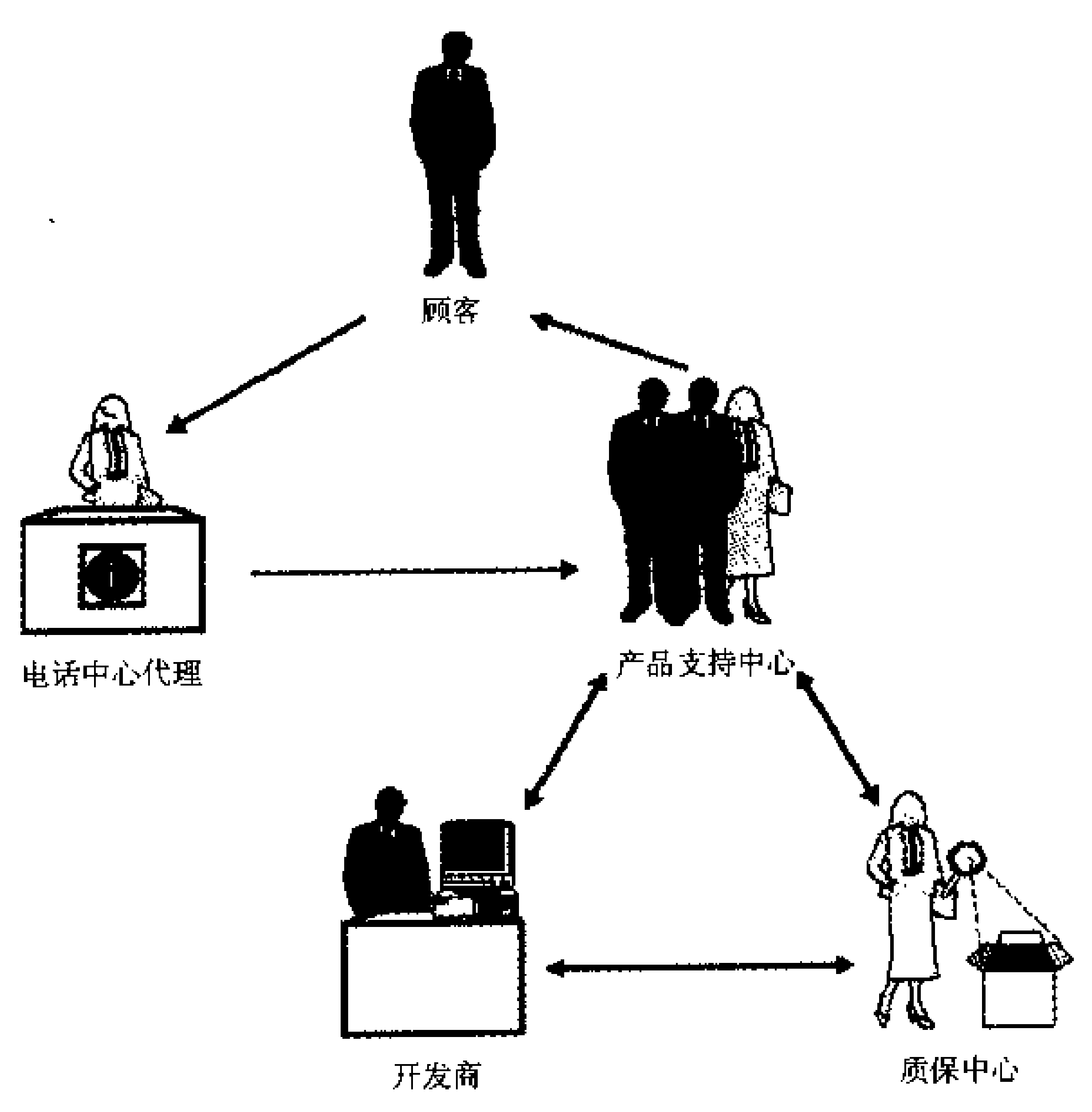


图19-1 产品支持中心过程流

总的来说，系统用户及其功能如下：

电话中心代理	检测用户授权 输入新的问题 查询已存在问题状态
产品支持中心	接收电话中心的问题报告 查阅产品的突出问题 与顾客接洽 修改问题状态 加入注释 路由问题到开发者
开发者	接收来自产品支持的问题报告 查阅产品的突出问题 分析问题并开发解决方案 向问题报告加入注释 路由问题到质保中心
质保中心	接收来自开发者的问题报告和解决方案

精彩好书见www.book118.com

执行集成测试
向问题报告加入注释
路由已解决的问题到产品支持中心
如果测试失败，将问题路由回开发者

另外，管理者可以在任意时刻查阅问题状态并访问显示质量的统计报告，如在队列中的时间、每个产品的故障报告以及开发者的突出故障。本应用中不包含这些报告，但可在问题数据库中开发。

19.2 数据模式

下表描述了包含所有记录和跟踪问题所需数据的数据库表格。

产品支持应用的数据模式

表 名	描 述	域
customer	购买了LyricNoye产品的顾客列表	顾客ID 顾客名 电话
Product	产品及其支持人员的列表	产品ID 产品名 产品支持人员 首要开发者 首要测试者
custprod	指出哪个顾客购买了哪个产品的顾客/产品对列表	顾客ID 产品ID 购买时间
Problems	已报告问题的主要记录	问题ID 描述 严重程度（1=高、2=中等，3=低） 报告时间 解决时间（如果已被解决） 顾客ID 产品ID
Problog	在已报告问题的生命期中事件的记录	问题ID 时间戳 事件ID 注释
employees	系统的用户，包括电话中心代理、产品支持中心、开发者和测试者	雇员ID 名字 其他域（这里未用）

19.3 开发系统

JSP是一种很方便的开发环境。必要时页面会自动被编译，URL很容易映射到目录的位置。

精彩好书见www.book118.com

在一个JSP页面中，可以使用HTML和Java的任意混合，这将大大增加灵活性。

不幸的是，这些优点意味着一般的JSP应用扩展性不好。随着嵌入JSP页面的Java代码增多，跟踪它也越来越困难。与可以被编译和单元测试的Java类不同，JSP scriptlet代码很难从其容器中分离出来。保持具有大量Java代码段的HTML的一致性也很困难。你可能开始使用bean完成了大部分工作，然后发现它们并不很正确，这样就导致将一段错误的Java代码埋葬在HTML中。如果应用随意利用<%@include%>伪指令，这些问题就会混在一起。

在更大的应用中将代码分隔成带有清晰功能的组件是一种更好的方式。对于这里的产品支持系统，使用了模式-视图-控制器（MVC）设计。

19.4 模式－视图－控制器结构

MVC后隐藏的思想是一个系统的可视方面应该从内部工作中分离出来，而内部工作应依次从开始和控制内部工作的机制中分离出来。MVC结构首先由Smalltalk及其从业者所采纳。但现在已是一种广泛使用的设计模式。图19-2阐述了MVC的工作方式。

模式指管理系统抽象内部状态和操作的代码。它处理数据库访问和大部分事务逻辑。模式没有可视组件，而是给出应用其他部分可以访问的应用编程接口。这使得编写一个从简单命令行接口测试和调试模式的驱动器程序成为可能。

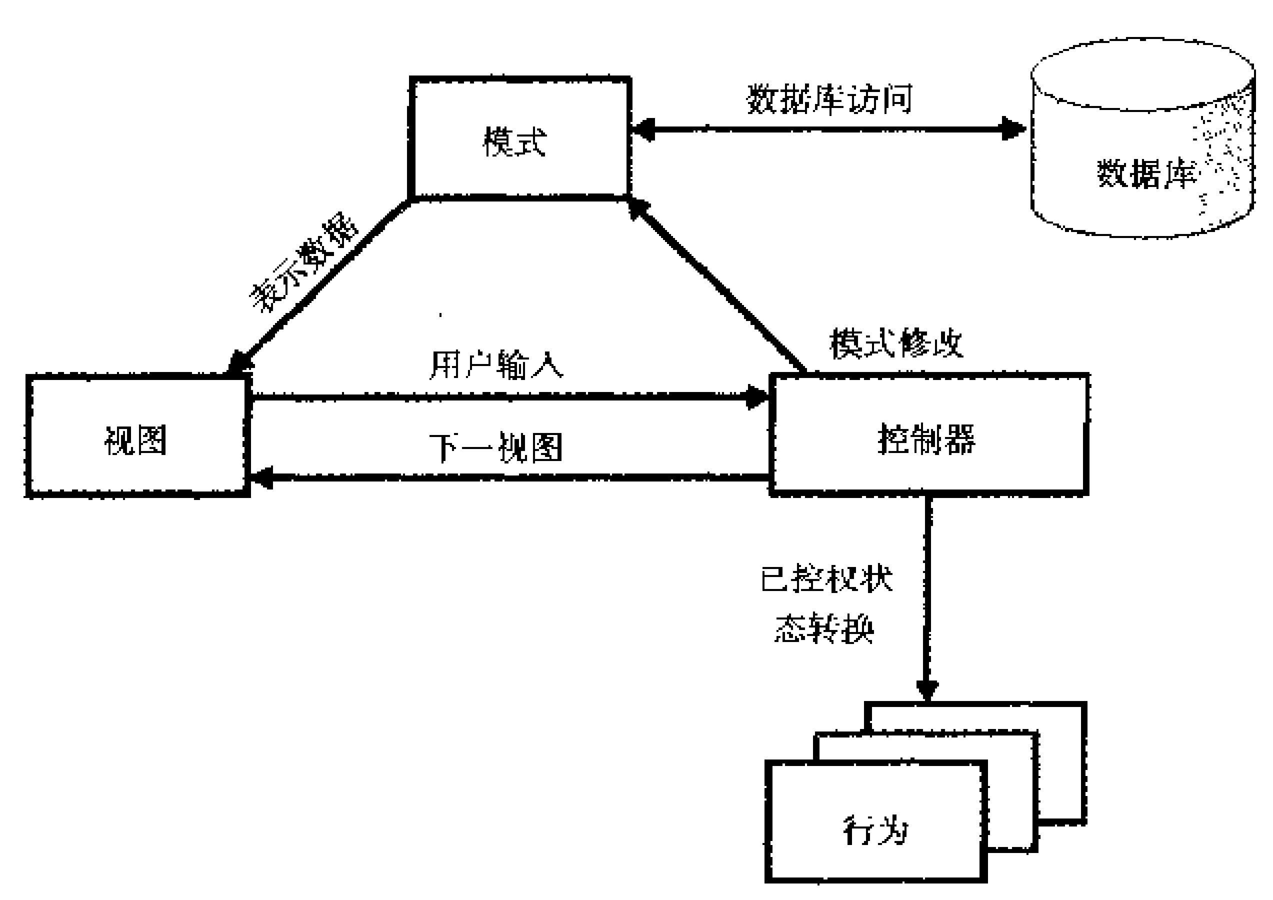


图19-2　模式-视图-控制器结构

例如，在一个象棋游戏中，模式可能由表示棋子的对象和保存它们的一个简单的8*8数组组成。模式可以具有指出其移动次序，评价一次给定的移动是否合法以及从一个数组元素到另一个数组元素移动棋子的方法。模式没有提供棋盘的任何可视表示的代码。

视图是系统的表示层。它不进行数据库访问，也不包含事务逻辑。视图所具有的一点非可视代码被限制于表示逻辑，如要显示的对象数组的循环。通过设计，一个模式可以与多个视图

精彩好书见www.book118.com

相关，可能是一个图形用户接口（GUI）或一个打印的报告。例如，一个基于Web的两人游戏可以对每个游戏者拥有一个视图，却都隶属于同一个模式。这不需要对模式做任何改变，因为模式并不知道其显示的方式。

在象棋游戏中，视图可能包含画出棋盘和棋子、列出移动、显示时钟的代码。它不知道任何象棋的规则、棋子的次序甚至是棋子的位置。视图将调用模式中的方法跟踪所有这些行为。即使是GUI代码也常常是很复杂的，所有视图都必须严格考虑其可视方面。测试和调试很直接，因为模式的stub版本可用于实现视图的上述方面。

控制器是依据用户输入表示模式的过程。基于当前视图、模式的状态和用户采取的行为，控制器调用模式的API修改模式状态并选择下一视图。粗略的讲，控制器处理用户输入，而视图处理用户的输出[1]。象棋游戏可以有两个控制器——一个传送游戏者的移动，一个选择计算机的移动。

在上述产品支持系统中，模式由一般的Java类（不是servlet和面向网络的）组成。简单的JSP页面被用作视图，控制器是带有某些支持类的单个servlet。

19.4.1　模式类

下面开始讨论产品支持系统模式。它由3个类的集合组成：

- 表示事务对象的类，与数据库中的表格大致对应。
- 应用容器和接口类。
- 测试框架。

这些类集中在一起组成了com.lyricnote.support.model包。除了维护应用状态，模式还包含所有访问数据库的代码，使用一个感知会话的包装类以确保数据库资源被正确管理。

1) 事务对象

6个类封装了在数据模式中使用的事务实体：

- `com.lyricnote.support.model.Customer`
- `com.lyricnote.support.model.Product`
- `com.lyricnote.support.model.CustomerProduct`
- `com.lyricnote.support.model.Employee`
- `com.lyricnote.support.model.Problem`
- `com.lyricnote.support.model.ProblemLog`

下面各节列出每个类的源码并描述了其操作。

Customer类　包中的第一个类是Customer类，它表示购买了LyricNote产品并符合产品支持条件的一个人或公司。它包含顾客名和电话号码，以及惟一的顾客标识符。标识符由顾客名的前4个字母，后跟顾客姓的第一和最后一个字母组成，结束是一个两位的惟一数字型后缀。该类在如下列表中给出。

```
package com.lyricnote.support.model;
```

1　控制器的角色有时可由一个更简单的模式-视图体系结构中的视图处理。

精彩好书见www.book118.com


```
import java.io.*;
import java.sql.*;
import java.util.*;

/**
* A person or company that has bought LyricNote products.
*/
public class Customer implements Serializable
{
   private String customerID;
   private String name;
   private String phone;

   /**
   * Factory method to create a customer record
   * from the current row of a result set.
   * @param rs a result set from the customer table
   * @exception SQLException if a database error occurs
   */
   public static Customer load(ResultSet rs)
      throws SQLException
   {
      Customer customer = new Customer();
      String value = null;

      value = rs.getString(1);
      if (value != null)
         customer.setCustomerID(value);

      value = rs.getString(2);
      if (value != null)
        customer.setName(value);

      value = rs.getString(3);
      if (value != null)
         customer.setPhone(value);

      return customer;
  }

  /**
  * Returns the object as a CSV string
  */
  public String toString()
  {
```

精彩好书见www.book118.com


```
        StringBuffer sb = new StringBuffer();

        if (getCustomerID() != null)
            sb.append(Util.quote(getCustomerID()));

        sb.append(",");
        if (getName() != null)
            sb.append(Util.quote(getName()));

        sb.append(",");
        if (getPhone() != null)
            sb.append(Util.quote(getPhone()));

        return sb.toString();
    }

    // =================================================
    // Property accessor methods
    // =================================================

    /**
    * Returns the customerID.
    */
    public String getCustomerID()
    {
        return customerID;
    }

    /**
    * Sets the customerID.
    * @param customerID the customerID.
    */
    public void setCustomerID(String customerID)
    {
        this.customerID = customerID;
    }

    /**
    * Returns the name.
    */
    public String getName()
    {
        return name;
    }

    /**
```

精彩好书见www.book118.com


```
     * Sets the name.
     * @param name the name.
     */
    public void setName(String name)
    {
       this.name = name;
    }

    /**
     * Returns the telephone number
     */
    public String getPhone()
    {
       return phone;
    }

    /**
     * Sets the telephone number.
     * @param phone the phone.
     */
    public void setPhone(String phone)
    {
       this.phone = phone;
    }
}
```

除了每一属性的getter和setter方法，Customer类还包括一个返回以逗号分隔值的字符串的toString（）方法以及一个load（）类方法。load（）方法从一个SQL结果集的customer表格行中创建一个Customer对象。此方法用于模式应用类以简化从数据库中检索Customer对象的集合。

toString（）方法中的Util.quote（）方法在本章后面"应用对象"节讨论。

Product类 接下来是标识product表格一行的Product类。其属性由惟一产品标识符、产品名、产品的基本支持人员的雇员数目、其首要开发者和其首要测试者组成。该类在下面列出：

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.*;
import java.util.*;

/**
* A software product supported by the Product
* Support system.
*/
public class Product implements Serializable
{
   private String productID;
```

精彩好书见www.book118.com


```
private String name;
private String productSupport;
private String developer;
private String tester;

/**
* Factory method to create a product record
* from the current row of a result set.
* @param rs a result set from the product table
* @exception SQLException if a database error occurs
*/
public static Product load(ResultSet rs)
   throws SQLException
{
   Product product = new Product();
   String value = null;

   value = rs.getString(1);
   if (value != null)
      product.setProductID(value);

   value = rs.getString(2);
   if (value != null)
      product.setName(value);

   value = rs.getString(3);
   if (value != null)
      product.setProductSupport(value);

   value = rs.getString(4);
   if (value != null)
      product.setDeveloper(value);

   value = rs.getString(5);
   if (value != null)
      product.setTester(value);

   return product;
}

/**
* Returns the object as a CSV string
*/
public String toString()
{
   StringBuffer sb = new StringBuffer();
```

精彩好书见www.book118.com


```
        if (getProductID() != null)
            sb.append(Util.quote(getProductID()));

        sb.append(",");
        if (getName() != null)
            sb.append(Util.quote(getName()));

        sb.append(",");
        if (getProductSupport() != null)
            sb.append(Util.quote(getProductSupport()));

        sb.append(",");
        if (getDeveloper() != null)
            sb.append(Util.quote(getDeveloper()));

        sb.append(",");
        if (getTester() != null)
            sb.append(Util.quote(getTester()));

        return sb.toString();
    }

    // ==========================================
    // Property accessor methods
    // ==========================================

    /**
    * Returns the product ID.
    */
    public String getProductID()
    {
        return productID;
    }

    /**
    * Sets the product ID.
    * @param product the product ID.
    */
    public void setProductID(String productID)
    {
        this.productID = productID;
    }

    /**
    * Returns the product name.
```

精彩好书见www.book118.com


```
*/
public String getName()
{
    return name;
}

/**
* Sets the product name.
* @param name the product name.
*/
public void setName(String name)
{
    this.name = name;
}

/**
* Returns the productSupport ID.
*/
public String getProductSupport()
{
    return productSupport;
}

/**
* Sets the productSupport ID.
* @param productSupport the productSupport.
*/
public void setProductSupport(String productSupport)
{
    this.productSupport = productSupport;
}

/**
* Returns the developer ID.
*/
public String getDeveloper()
{
    return developer;
}

/**
* Sets the developer ID.
* @param developer the developer.
*/
public void setDeveloper(String developer)
{
```

精彩好书见www.book118.com


```
        this.developer = developer;
    }

    /**
    * Returns the tester ID.
    */
    public String getTester()
    {
        return tester;
    }

    /**
    * Sets the tester ID.
    * @param tester the tester.
    */
    public void setTester(String tester)
    {
        this.tester = tester;
    }
}
```

像Customer类一样，Product对每一属性包含getter和setter方法以及定制的toString（ ）方法和从一个结果集行中载入一个Pruduct对象的工厂方法。

CustomerProduct类　电话中心代理的关键任务是检验顾客授权，即报告问题的人应是一个有效顾客并已购买了指定产品。这通过custprod表格中链接顾客和产品的记录的存在性指出。对应此表格的类是CustomerProduct，如下所示：

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.*;

/**
* A customer/product pair whose existence indicates
* that the customer bought the specified product.
*/
public class CustomerProduct implements Serializable
{
    private String customerID;
    private String productID;
    private Date datePurchased;

    /**
    * Factory method to create a customer/product record
```

精彩好书见www.book118.com


```
 * from the current row of a result set.
 * @param rs a result set from the customer table
 * @exception SQLException if a database error occurs
 */
public static CustomerProduct load(ResultSet rs)
   throws SQLException
{
   CustomerProduct custprod = new CustomerProduct();

   custprod.setCustomerID(rs.getString(1));
   custprod.setProductID(rs.getString(2));
   custprod.setDatePurchased(rs.getDate(3));

   return custprod;
}

/**
* Returns the object as a CSV string
*/
public String toString()
{
   StringBuffer sb = new StringBuffer();

   sb.append(getCustomerID());
   sb.append(",");
   sb.append(getProductID());
   sb.append(",");
   sb.append(Util.dateFormat(getDatePurchased()));

   return sb.toString();
}

// ========================================
// Property accessor methods
// ========================================

/**
* Returns the customerID.
*/
public String getCustomerID()
{
   return customerID;
}

/**
* Sets the customerID.
```

精彩好书见www.book118.com


```
v* @param customerID the customerID.
     */
    public void setCustomerID(String customerID)
    {
        this.customerID = customerID;
    }

    /**
     * Returns the productID.
     */
    public String getProductID()
    {
        return productID;
    }

    /**
     * Sets the productID.
     * @param productID the productID.
     */
    public void setProductID(String productID)
    {
        this.productID = productID;
    }

    /**
     * Returns the datePurchased.
     */
    public Date getDatePurchased()
    {
        return datePurchased;
    }

    /**
     * Sets the datePurchased.
     * @param datePurchased the datePurchased.
     */
    public void setDatePurchased(Date datePurchased)
    {
        this.datePurchased = datePurchased;
    }
}
```

CustomerProduct具有包含顾客ID、产品ID和产品被购买的日期的域。此信息来自产品注册卡或手工输入，这里没有列出。

像Customer和Product类一样，CustomerProduct具有getter和setter方法、创建逗号分隔值字符串的toStrong（）方法和从数据库载入对象的工厂方法。

精彩好书见www.book118.com

Employee类　与每一产品相关的是产品支持人、首要开发者和首要测试者。这些雇员的信息包含在employee表格中，并封装在Employee类中，如下所示：

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.*;

/**
* A LyricNote employee that uses the Product Support system.
*/
public class Employee implements Serializable
{
   private String employeeID;
   private String name;
   private Date dateHired;
   private boolean isManager;
   private String departmentID;
   private String title;
   private String email;
   private String phone;

   /**
   * Factory method to create an employee record
   * from the current row of a result set.
   * @param rs a result set from the employee table
   * @exception SQLException if a database error occurs
   */
   public static Employee load(ResultSet rs)
      throws SQLException
   {
      Employee employee = new Employee();

      employee.setEmployeeID(rs.getString(1));
      employee.setName(rs.getString(2));
      employee.setDateHired(rs.getDate(3));
      employee.setIsManager(rs.getBoolean(4));
      employee.setDepartmentID(rs.getString(5));
      employee.setTitle(rs.getString(6));
      employee.setEmail(rs.getString(7));
      employee.setPhone(rs.getString(8));

      return employee;
   }

   /**
```

精彩好书见www.book118.com


```
 * Returns the object as a CSV string
 */
public String toString()
{
   StringBuffer sb = new StringBuffer();
   sb.append(getEmployeeID());
   sb.append(",");
   sb.append(Util.quote(getName()));
   sb.append(",");
   sb.append(Util.dateFormat(getDateHired()));
   sb.append(",");
   sb.append(getIsManager());
   sb.append(",");
   sb.append(getDepartmentID());
   sb.append(",");
   sb.append(Util.quote(getTitle()));
   sb.append(",");
   sb.append(getEmail());
   sb.append(",");
   sb.append(getPhone());

   return sb.toString();
}

// ==========================================
// Property accessor methods
// ==========================================

/**
 * Returns the employee ID.
 */
public String getEmployeeID()
{
   return employeeID;
}

/**
 * Sets the employee ID.
 * @param employeeID the employee ID.
 */
public void setEmployeeID(String employeeID)
{
   this.employeeID = employeeID;
}

/**
 * Returns the employee name.
 */
```

精彩好书见www.book118.com


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

/**
* Sets the employee name.
* @param name the employee name.
*/
public void setName(String name)
{
   this.name = name;
}

/**
* Returns the dateHired.
*/
public Date getDateHired()
{
   return dateHired;
}

/**
* Sets the dateHired.
* @param dateHired the dateHired.
*/
public void setDateHired(Date dateHired)
{
   this.dateHired = dateHired;
}

/**
* Returns the isManager flag.
*/
public boolean getIsManager()
{
   return isManager;
}

/**
* Sets the isManager flag.
* @param isManager the isManager flag.
*/
public void setIsManager(boolean isManager)
{
   this.isManager = isManager;
}
```

精彩好书见www.book118.com


```
/**
* Returns the department ID.
*/
public String getDepartmentID()
{
    return departmentID;
}

/**
* Sets the department ID.
* @param departmentID the department ID.
*/
public void setDepartmentID(String departmentID)
{
    this.departmentID = departmentID;
}

/**
* Returns the title.
*/
public String getTitle()
{
    return title;
}

/**
* Sets the title.
* @param title the title.
*/
public void setTitle(String title)
{
    this.title = title;
}

/**
* Returns the email.
*/
public String getEmail()
{
    return email;
}

/**
* Sets the email.
* @param email the email.
*/
public void setEmail(String email)
{
```

精彩好书见www.book118.com


```
        this.email = email;
    }

    /**
    * Returns the phone.
    */
    public String getPhone()
    {
        return phone;
    }

    /**
    * Sets the phone.
    * @param phone the phone.
    */
    public void setPhone(String phone)
    {
        this.phone = phone;
    }
}
```

与其他表格不同，employee表格不止在产品支持系统中使用。正因为这样，它包含了比在产品支持中使用的更多域，如下所示：

- employeeID　惟一的4位的雇员号码
- name　雇员名字
- dateHired　雇员雇佣时间
- isManager　一个布尔变量，如果雇员是经理，则其值为真。
- departmentID　雇员所属单位的代码
- title　工作题目
- email　电子邮件地址
- phone　电话分机

Employee类（如下所示）封装了employee表格中的一行。

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.*;

/**
* A LyricNote employee that uses the Product Support system.
*/
public class Employee implements Serializable
{
```

精彩好书见www.book118.com


```
private String employeeID;
private String name;
private Date dateHired;
private boolean isManager;
private String departmentID;
private String title;
private String email;
private String phone;

/**
* Factory method to create an employee record
* from the current row of a result set.
* @param rs a result set from the employee table
* @exception SQLException if a database error occurs
*/
public static Employee load(ResultSet rs)
   throws SQLException
{
   Employee employee = new Employee();
   employee.setEmployeeID(rs.getString(1));
   employee.setName(rs.getString(2));
   employee.setDateHired(rs.getDate(3));
   employee.setIsManager(rs.getBoolean(4));
   employee.setDepartmentID(rs.getString(5));
   employee.setTitle(rs.getString(6));
   employee.setEmail(rs.getString(7));
   employee.setPhone(rs.getString(8));
   return employee;
}

/**
* Returns the object as a CSV string
*/
public String toString()
{
   StringBuffer sb = new StringBuffer();

   sb.append(getEmployeeID());
   sb.append(",");
   sb.append(Util.quote(getName()));
   sb.append(",");
   sb.append(Util.dateFormat(getDateHired()));
   sb.append(",");
   sb.append(getIsManager());
   sb.append(",");
   sb.append(getDepartmentID());
   sb.append(",");
   sb.append(Util.quote(getTitle()));
   sb.append(",");
   sb.append(getEmail());
   sb.append(",");
   sb.append(getPhone());
```

精彩好书见www.book118.com


```
        return sb.toString();
    }

    // ============================================
    // Property accessor methods
    // ============================================

    /**
    * Returns the employee ID.
    */
    public String getEmployeeID()
    {
        return employeeID;
    }

    /**
    * Sets the employee ID.
    * @param employeeID the employee ID.
    */
    public void setEmployeeID(String employeeID)
    {
        this.employeeID = employeeID;
    }

    /**
    * Returns the employee name.
    */
    public String getName()
    {
        return name;
    }

    /**
    * Sets the employee name.
    * @param name the employee name.
    */
    public void setName(String name)
    {
        this.name = name;
    }

    /**
    * Returns the dateHired.
    */
    public Date getDateHired()
    {
        return dateHired;
    }

    /**
```

精彩好书见www.book118.com


```
 * Sets the dateHired.
 * @param dateHired the dateHired.
 */
public void setDateHired(Date dateHired)
{
   this.dateHired = dateHired;
}

/**
 * Returns the isManager flag.
 */
public boolean getIsManager()
{
   return isManager;
}

/**
 * Sets the isManager flag.
 * @param isManager the isManager flag.
 */
public void setIsManager(boolean isManager)
{
   this.isManager = isManager;
}

/**
 * Returns the department ID.
 */
public String getDepartmentID()
{
   return departmentID;
}

/**
 * Sets the department ID.
 * @param departmentID the department ID.
 */
public void setDepartmentID(String departmentID)
{
   this.departmentID = departmentID;
}

/**
 * Returns the title.
 */
public String getTitle()
{
   return title;
}

/**
 * Sets the title.
```

精彩好书见www.book118.com


```
     * @param title the title.
     *
     public void setTitle(String title)
     {
        this.title = title;
     }

     /**
     * Returns the email.
     */
     public String getEmail()
     {
        return email;
     }

     /**
     * Sets the email.
     * @param email the email.
     */
     public void setEmail(String email)
     {
        this.email = email;
     }

     /**
     * Returns the phone.
     */
     public String getPhone()
     {
        return phone;
     }
     /**
     * Sets the phone.
     * @param phone the phone.
     */
     public void setPhone(String phone)
     {
        this.phone = phone;
     }
}
```

Employee类有前面讲过的toString ( ) 和load ( ) 方法。

Problem类 系统的核心是报告的问题集。在数据库中，每个问题由两类记录组成：一类表示问题整体，另一类表示问题从其报告到其被解决的生命期中的每一事件。静态问题数据包含在Problem类中，如下所示：

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.ResultSet;
```

精彩好书见www.book118.com


```
import java.sql.SQLException;
import java.util.*;

/**
* A software problem supported by the Problem
* Support system.
*/
public class Problem implements Serializable
{
   private String problemID;
   private String description;
   private int severity;
   private java.util.Date dateReported;
   private java.util.Date dateResolved;
   private String customerID;
   private String productID;

   /**
   * Factory method to create a problem record
   * from the current row of a result set.
   * @param rs a result set from the problem table
   * @exception SQLException if a database error occurs
   */
   public static Problem load(ResultSet rs)
      throws SQLException
   {
      Problem problem = new Problem();

      problem.setProblemID(rs.getString(1));
      problem.setDescription(rs.getString(2));
      problem.setSeverity(rs.getInt(3));
      problem.setDateReported(rs.getTimestamp(4));
      problem.setDateResolved(rs.getTimestamp(5));
      problem.setCustomerID(rs.getString(6));
      problem.setProductID(rs.getString(7));

      return problem;
   }

   /**
   * Returns the object as a CSV string
   */
   public String toString()
   {
      StringBuffer sb = new StringBuffer();
```

精彩好书见www.book118.com


```
        sb.append(getProblemID());
        sb.append(",");
        sb.append(getDescription());
        sb.append(",");
        sb.append(getSeverity());
        sb.append(",");
        sb.append(Util.dateTimeFormat(getDateReported()));
        sb.append(",");
        sb.append(Util.dateTimeFormat(getDateResolved()));
        sb.append(",");
        sb.append(getCustomerID());
        sb.append(",");
        sb.append(getProductID());

        return sb.toString();
    }

    /**
    * Closes the problem
    */
    public void close()
    {
        setDateResolved(Util.toTimestamp(new Date()));
    }

    // ==========================================
    // Property accessor methods
    // ==========================================

    /**
    * Returns the problemID.
    */
    public String getProblemID()
    {
        return problemID;
    }

    /**
    * Sets the problemID.
    * @param problemID the problemID.
    */
    public void setProblemID(String problemID)
    {
        this.problemID = problemID;
    }
```

精彩好书见www.book118.com


```
/**
 * Returns the description.
 */
public String getDescription()
{
    return description;
}

/**
 * Sets the description.
 * @param description the description.
 */
public void setDescription(String description)
{
    this.description = description;
}

/**
 * Returns the severity.
 */
public int getSeverity()
{
    return severity;
}

/**
 * Sets the severity.
 * @param severity the severity.
 */
public void setSeverity(int severity)
{
    this.severity = severity;
}

/**
 * Returns the dateReported.
 */
public java.util.Date getDateReported()
{
    return dateReported;
}

/**
 * Sets the dateReported.
 * @param dateReported the dateReported.
 */
```

精彩好书见www.book118.com


```
public void setDateReported(java.util.Date dateReported)
{
    this.dateReported = dateReported;
}

/**
* Returns the dateResolved.
*/
public java.util.Date getDateResolved()
{
    return dateResolved;
}

/**
* Sets the dateResolved.
* @param dateResolved the dateResolved.
*/
public void setDateResolved(java.util.Date dateResolved)
{
    this.dateResolved = dateResolved;
}

/**
* Returns the customerID.
*/
public String getCustomerID()
{
    return customerID;
}

/**
* Sets the customerID.
* @param customerID the customerID.
*/
public void setCustomerID(String customerID)
{
    this.customerID = customerID;
}

/**
* Returns the productID.
*/
public String getProductID()
{
    return productID;
}
```

精彩好书见www.book118.com


```
    /**
    * Sets the productID.
    * @param productID the productID.
    */
    public void setProductID(String productID)
    {
        this.productID = productID;
    }
}
```

一个Problem对象由下列域组成：

- problemID 惟一问题标识符，由系统设定。
- description 为在GUI中显示，问题的基本描述。
- severity 电话中心代理考虑问题对顾客重要性的估计。1=高，2=中等，3=低。
- dateReported 问题报告到电话中心的日期和时间。
- dateResolved 如果问题被解决，此域包含其被解决的日期和时间。否则值为null。
- customerID 8个字符的顾客标识符。
- productID 惟一产品标识符。

ProblemLog类 在问题生命期中的事件被表格problog中的行所模拟。该表格包含下面的列：

- problemID 问题标识符，以允许将表problog和表problem连接起来。
- logtime 数据库系统生成的事件戳。它结合问题ID组成了问题记录入口的惟一关键字。
- eventID 3个字符的代码，指出问题记录事件的性质。事件ID代码从下述列表中获得：

COM——注释

RPS——路由到产品支持中心。

RPD——路由到产品开发者。

RQA——路由到质保中心。

CNB——结束——不是一个故障。

CCP——结束——顾客的问题。

CFX——结束——已解决。

- comments制作此记录入口的人输入的注释。

系统使用ProblemLog类表示problog表格中的一行，如下所示：

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.*;

/**
* An update to a reported problem.
```

精彩好书见www.book118.com


```
*/
public class ProblemLog implements Serializable
{
   private String problemID;
   private java.util.Date logTime;
   private String eventID;
   private String comments;

   /**
   * Factory method to create a problem record
   * from the current row of a result set.
   * @param rs a result set from the problem table
   * @exception SQLException if a database error occurs
   */
   public static ProblemLog load(ResultSet rs)
      throws SQLException
   {
      ProblemLog probLog = new ProblemLog();

      probLog.setProblemID(rs.getString(1));
      probLog.setLogTime(rs.getTimestamp(2));
      probLog.setEventID(rs.getString(3));
      probLog.setComments(rs.getString(4));

      return probLog;
   }

   /**
   * Returns the object as a CSV string
   *
   public String toString()
   {
      StringBuffer sb = new StringBuffer();

      sb.append(getProblemID());
      sb.append(",");
      sb.append(Util.dateTimeFormat(getLogTime()));
      sb.append(",");
      sb.append(getEventID());
      sb.append(",");
      sb.append(getComments());

      return sb.toString();
   }

   // ========================================
```

精彩好书见www.book118.com


```
// Property accessor methods
// ==========================================

/**
* Returns the problemID.
*/
public String getProblemID()
{
    return problemID;
}

/**
* Sets the problemID.
* @param problemID the problemID.
*/
public void setProblemID(String problemID)
{
    this.problemID = problemID;
}

/**
* Returns the logTime.
*/
public java.util.Date getLogTime()
{
    return logTime;
}

/**
* Sets the logTime.
* @param logTime the logTime.
*/
public void setLogTime(java.util.Date logTime)
{
    this.logTime = logTime;
}

/**
* Returns the eventID.
*/
public String getEventID()
{
    return eventID;
}

/**
```

精彩好书见www.book118.com


```
     * Sets the eventID.
     * @param eventID the eventID.
     */
    public void setEventID(String eventID)
    {
        this.eventID = eventID;
    }

    /**
     * Returns the comments.
     */
    public String getComments()
    {
        return comments;
    }

    /**
     * Sets the comments.
     * @param comments the comments.
     */
    public void setComments(String comments)
    {
        this.comments = comments;
    }
}
```

2) 应用对象

事物对象表示系统所知的每个实体。因模式起见，其他对象作为一个整体表示应用。在此范畴下的类包括：

- `com.lyricnote.support.model.Model`
- `com.lyricnote.support.model.WebModel`
- `com.lyricnote.support.model.Util`

下面各节列出其中每个类的源码并描述其操作。

Model类　在系统操作期间，事务对象驻留在名为Model的应用容器类中。此类给出了API允许控制器操作它使视图从中抽取数据。每个用户会话存在一个Model对象，因此模式状态是线程安全的。

Model是一个相当大的类。下面一次列出一段，这样就可以详细讨论它。

```
package com.lyricnote.support.model;

import java.io.*;
import java.sql.*;
import java.util.*;
```

精彩好书见www.book118.com


```
/**
 * The model component in the Model-View-Controller architecture
 * of the product support application. The model is designed
 * to be used in a dedicated HTTP session with a single user,
 * however, there is no HTTP-specific code. This allows the
 * model to be tested by a batch driver.
 */
public class Model implements Serializable
{
   // Configuration fields

   private String problemIDFile;
   private String jdbcDriver;
   private String databaseURL;
   private transient Connection con;

   // Customer fields

   private List customers;
   private String customerID;

   // Product fields

   private List products;
   private String productID;

   // Problem fields

   private List problems;
   private String problemID;

   // Problem log fields

   private List problemLogs;
```

Model包含了表示应用状态的实例变量，这些变量分成了下述类型：

- 配置域　这些域包括包含下一个可利用的问题ID号，用来访问数据库的JDBC驱动器的名字、数据库URL和数据库连接对象的文件名。
- 顾客域　模式支持对顾客名字的字母搜索。最近的搜索结果保存在Customer对象的一个java.util.List中。此列表作为一个属性给出，并使用getCustomers（）方法得到。另外，模式还拥有一个customerID属性，它为需要它的几个方法提供隐含的ID。
- 产品域　像顾客域一样，对当前产品搜索结果和当前产品ID也存在实例变量。
- 问题域　模式对Problem对象和当前被选中的问题也具有实例变量。
- 问题记录域　同时，对与当前问题相关的ProblemLog对象列表也有一个java.util.List。

精彩好书见www.book118.com


```
// ===================================================
// Configuration and database methods
// ===================================================

/**
* Assigns a globally unique problem ID
*/
public static synchronized String assignProblemID
   (String problemIDFile)
{
   String id = null;
   try {

      // Read the next available ID

      BufferedReader in =
         new BufferedReader(
         new FileReader(problemIDFile));
      id = in.readLine();
      in.close();

      // Increment it and rewrite the file

      String prefix = id.substring(0, 1);
      int suffix = Integer.parseInt(id.substring(1));
      suffix++;
      String newID = "0000000" + String.valueOf(suffix);
      newID = newID.substring(newID.length() - 7);
      newID = prefix + newID;

      PrintWriter out =
         new PrintWriter(
         new FileWriter(problemIDFile));
      out.println(newID);
      out.flush();
      out.close();
   }
   catch (IOException e) {
      e.printStackTrace();
   }
   finally {
      return id;
   }
}

/**
```

精彩好书见www.book118.com


```
 * Creates a new connection using the currently
 * specified JDBC driver and URL
 * @exception SQLException if the connection fails
 * or if it already exists
 */
public void connect()
   throws SQLException
{
   if (isConnected())
      throw new SQLException("Already connected");

   // Verify that the driver and URL have been specified

   if (jdbcDriver == null)
      throw new SQLException("No jdbcDriver property");

   if (databaseURL == null)
      throw new SQLException("No databaseURL property");

   // Load the driver

   try {
      Class.forName(jdbcDriver).newInstance();
   }
   catch (ClassNotFoundException e) {
      throw new SQLException
      (jdbcDriver + " class could not be loaded");
   }

   // Open the connection

   con = DriverManager.getConnection(databaseURL);
}

/**
* Closes the current connection
*/
public void disconnect()
{
   // Close the connection

   if (con != null) {
      try {
         con.close();
         }
```

精彩好书见www.book118.com


```
            catch (SQLException ignore) {}
            finally {
                con = null;
            }
        }
    }

    /**
     * Returns true if there is an active connection
     */
    public boolean isConnected()
    {
        return (con != null);
    }

    /**
     * Returns the jdbcDriver.
     */
    public String getJdbcDriver()
    {
        return jdbcDriver;
    }

    /**
     * Sets the jdbcDriver.
     * @param jdbcDriver the jdbcDriver.
     */
    public void setJdbcDriver(String jdbcDriver)
    {
        this.jdbcDriver = jdbcDriver;
    }

    /**
     * Returns the databaseURL.
     */
    public String getDatabaseURL()
    {
        return databaseURL;
    }

    /**
     * Sets the databaseURL.
     * @param databaseURL the databaseURL.
     */
    public void setDatabaseURL(String databaseURL)
    {
```

精彩好书见www.book118.com


```
        this.databaseURL = databaseURL;
    }

    /**
     * Returns the problemIDFile.
     */
    public String getProblemIDFile()
    {
        return problemIDFile;
    }

    /**
     * Sets the problemIDFile.
     * @param problemIDFile the problemIDFile.
     */
    public void setProblemIDFile(String problemIDFile)
    {
        this.problemIDFile = problemIDFile;
    }
```

模式包含一个处理数据源的方法集。第一个是assignProblemID（String problemIDFile）。这是一个从文件中读取下一个可以得到的问题ID，然后使用一个增加数重写该文件的类方法。该方法是同步的，因此生成ID是惟一的[1]。

数据库连接由三种方法管理：

- `void connect()`
- `void disconnect()`
- `boolean isConnected()`

connect（）方法使用模式的JdbcDriver和databaseURL属性打开数据库的一个JDBC连接。这些属性从web.xml发布描述器中指定的上下文参数设置。disconnect（）方法关闭连接，isConnected（）方法给出一种测试数据库连接是否存在的方式。

注意 connect（）和disconnect（）方法提供连接数据库的能力，但它们并不选择何时及如何连接。实际上模式本身没有处理此功能的逻辑。这是控制器对象的任务，下面将会提到。

```
    // ==========================================================
    // Customer methods
    // ==========================================================

    /**
     * Returns the customer object corresponding to
```

1 文件名被存储为一个实例变量，但使用它的方法是一个类方法。这就是它必须被传递为一个参数的原因。因为模式可从Web或命令行测试shell中运行。每一情况文件可能处于不同的位置。

精彩好书见www.book118.com


```
 * the current customer ID
 * @exception SQLException if a database error occurs
 */
public Customer getCustomer()
   throws SQLException
{
   // Verify that a connection exists

   if (!isConnected())
      throw new SQLException("No connection");

   // Verify that there is a current customer ID

   if (customerID == null)
      throw new SQLException("No customer ID");

   PreparedStatement pstmt = null;
   ResultSet rs = null;
   Customer customer = null;

   try {

      // Prepare the query SQL

      pstmt = con.prepareStatement
      ("select * from customers where customerID = ?");
      pstmt.setString(1, customerID);

      // Execute the query

      rs = pstmt.executeQuery();
      if (rs.next())
      customer = Customer.load(rs);
   }
   finally {
      if (rs != null)
         rs.close();
      if (pstmt != null)
         pstmt.close();
   }

   // Return the customer

   return customer;
}
```

精彩好书见www.book118.com


```
/**
 * Returns the current customer search results
 */
public List getCustomers()
{
    return customers;
}

/**
 * Uses the specified customer search argument to query
 * the database for matching customers. Creates a list
 * of customer objects.
 * @param searchArgument the search argument
 * @exception SQLException if a database error occurs
 */
public void customerSearch(String searchArgument)
    throws SQLException
{
    // Verify that a connection exists and that
    // the search argument has been specified

    if (!isConnected())
        throw new SQLException("No connection");

    PreparedStatement pstmt = null;
    ResultSet rs = null;
    customers = null;
    try {

        // Prepare the query SQL

        pstmt = con.prepareStatement(
            "select *"
            + " from customers"
            + " where name like ?"
            + " order by name"
        );
        searchArgument = searchArgument.trim();
        searchArgument = "%" + searchArgument + "%";
        pstmt.setString(1, searchArgument);

        // Execute the query and copy the results
        // to a List

        rs = pstmt.executeQuery();
        customers = new LinkedList();
```

精彩好书见www.book118.com


```
            while (rs.next()) {
                customers.add(Customer.load(rs));
            }
        }
        finally {
            if (rs != null)
                rs.close();
            if (pstmt != null)
                pstmt.close();
        }
    }

    /**
    * Returns the customerID.
    */
    public String getCustomerID()
    {
        return customerID;
    }

    /**
    * Sets the customerID.
    * @param customerID the customerID.
    */
    public void setCustomerID(String customerID)
    {
        this.customerID = customerID;
}
```

当前顾客ID具有get和set方法以及一个从数据库中检索具有此ID的Customer对象的方法。getCustomer ( ) 方法在从结果集中抽取Customer对象中阐述了Customer.load ( ) 方法。customerSearch ( ) 方法从customer表格中选择其name域匹配指定搜索参数的Customer对象。结果java.util.List被保存为一个实例变量，可使用getCustomers ( ) 检索。

```
    // ==========================================
    // Product methods
    // ==========================================

    /**
    * Returns the product object corresponding to
    * the current product ID
    * @exception SQLException if a database error occurs
    */
    public Product getProduct()
        throws SQLException
    {
```

精彩好书见www.book118.com


```
    // Verify that a connection exists

    if (!isConnected())
       throw new SQLException("No connection");

    // Verify that a current product ID exists

    if (productID == null)
       throw new SQLException("No product ID");

    PreparedStatement pstmt = null;
    ResultSet rs = null;
    Product product = null;

    try {

       // Prepare the query SQL

       pstmt = con.prepareStatement
       ("select * from products where productID = ?");
       pstmt.setString(1, productID);

       // Execute the query

       rs = pstmt.executeQuery();
       if (rs.next())
          product = Product.load(rs);
    }
    finally {
       if (rs != null)
          rs.close();
       if (pstmt != null)
          pstmt.close();
    }

       // Return the product

       return product;
}

/**
* Returns the current product search results
*/
public List getProducts()
{
```

精彩好书见www.book118.com


```
        return products;
    }

    /**
    * Uses the specified product search argument to query
    * the database for matching products. Creates a list
    * of product objects.
    * @param searchArgument the search argument
    * @exception SQLException if a database error occurs
    */
    public void productSearch(String searchArgument)
        throws SQLException
    {

        // Verify that a connection exists and that
        // the search argument has been specified

        if (!isConnected())
            throw new SQLException("No connection");

        PreparedStatement pstmt = null;
        ResultSet rs = null;
        products = null;

        try {

            // Prepare the query SQL

            pstmt = con.prepareStatement(
                "select *"
                + " from products"
                + " where name like ?"
                + " order by name"
            );
            searchArgument = searchArgument.trim();
            searchArgument = "%" + searchArgument + "%";
            pstmt.setString(1, searchArgument);

            // Execute the query and copy the results
            // to a List

            rs = pstmt.executeQuery();
            products = new LinkedList();
            while (rs.next())
                products.add(Product.load(rs));
```

精彩好书见www.book118.com


```
        }
        finally {
            if (rs != null)
                rs.close();
            if (pstmt != null)
                pstmt.close();
        }
    }

    /**
    * Returns the productID.
    */
    public String getProductID()
    {
        return productID;
    }

    /**
    * Sets the productID.
    * @param productID the productID.
    */
    public void setProductID(String productID)
    {
        this.productID = productID;
    }
```

与顾客方法对应，产品方法取得和设置当前产品ID，检索相应的Product对象，选择匹配一个搜索字符串的产品并检索选中的产品。

```
    // =============================================
    // Customer/product methods
    // =============================================

    /**
    * Returns a list of CustomerProduct objects
    * for the current customer.
    * @exception SQLException if a database error occurs
    */
    public List getCustomerProducts()
        throws SQLException
    {
        // Verify that a connection exists

        if (!isConnected())
            throw new SQLException("No connection");

        // Verify that a current customer ID exists
```

精彩好书见www.book118.com


```
        if (customerID == null)
            throw new SQLException("No customer ID");

        PreparedStatement pstmt = null;
        ResultSet rs = null;
        List list = null;

        try {

            // Prepare the query SQL

            pstmt = con.prepareStatement(
                "select *"
                + " from custprod"
                + " where customerID = ?"
                + " order by datePurchased desc"
            );
            pstmt.setString(1, customerID);

            // Execute the query and populate the list

            rs = pstmt.executeQuery();
            list = new LinkedList();
            while (rs.next())
                list.add(CustomerProduct.load(rs));
        }
        finally {
            if (rs != null)
                rs.close();
            if (pstmt != null)
                pstmt.close();
        }

        // Return the list

        return list;
    }
```

当顾客ID被选中并保存在模式中时，可以在custprod表中搜索顾客购买的产品。CustomerProblem对象的搜索结果列表按购买时间降序排列被保存并返回给调用者。

```
    // ========================================
    // Employee methods
    // ========================================

    /**
     * Returns the employee object corresponding to
```

精彩好书见www.book118.com


```
     * the specified employee ID
     * @param employeeID the employee ID
     * @exception SQLException if a database error occurs
     */
    public Employee getEmployee(String employeeID)
       throws SQLException
    {
       // Verify that a connection exists

       if (!isConnected())
          throw new SQLException("No connection");

       PreparedStatement pstmt = null;
       ResultSet rs = null;
       Employee employee = null;

       try {

          // Prepare the query SQL

          pstmt = con.prepareStatement
          ("select * from employees where employeeID = ?");
          pstmt.setString(1, employeeID);

          // Execute the query

          rs = pstmt.executeQuery();
          if (rs.next())
             employee = Employee.load(rs);
       }
       finally {
          if (rs != null)
             rs.close();
          if (pstmt != null)
             pstmt.close();
       }

       // Return the employee

       return employee;
    }
```

可以通过调用getEmployee（），并向其传递雇员ID从数据库中检索Employee对象。这对显示Product对象中3个支持ID的雇员名很有用。

```
    // ==========================================
    // Problem methods
```

精彩好书见www.book118.com


```
// ==========================================

/**
 * Factory method to create a new problem record
 * and add it to the database
 */
public void newProblem() throws SQLException
{
   if (getCustomerID() == null)
      throw new SQLException
      ("No customer ID");

   if (getProductID() == null)
      throw new SQLException
      ("No product ID");

      Problem problem = new Problem();

      String fileName = getProblemIDFile();
      problemID = assignProblemID(fileName);
      problem.setProblemID(problemID);
      problem.setDescription("");
      problem.setSeverity(2);
      problem.setDateReported(new java.util.Date());
      problem.setCustomerID(getCustomerID());
      problem.setProductID(getProductID());

      // Add to database

      PreparedStatement pstmt = null;
      try {
         pstmt = con.prepareStatement
         ("insert into problems values(?, ?, ?, ?, ?, ?, ?)");
         pstmt.setString(1, problemID);
         pstmt.setString(2, problem.getDescription());
         pstmt.setInt(3, problem.getSeverity());
         pstmt.setTimestamp
            (4, Util.toTimestamp(problem.getDateReported()));
         pstmt.setNull(5, Types.TIMESTAMP);
         pstmt.setString(6, problem.getCustomerID());
         pstmt.setString(7, problem.getProductID());
         pstmt.executeUpdate();
      }
      finally {
      if (pstmt != null)
         pstmt.close();
   }
```

精彩好书见www.book118.com


```
}

/**
 * Updates the problem record in the database
 * @param problem the problem object
 * @exception SQLException if a database error occurs
 */
public void updateProblem(Problem problem)
   throws SQLException
{
   // Verify that a connection exists

   if (!isConnected())
      throw new SQLException("No connection");

   PreparedStatement pstmt = null;
   try {

      // Prepare the query SQL

      pstmt = con.prepareStatement
      ( " update problems"
      + " set"
      + " description = ?,"
      + " severity = ?,"
      + " dateResolved = ?"
      + " where problemID = ?"
      );
      pstmt.setString(1, problem.getDescription());
      pstmt.setInt(2, problem.getSeverity());
      if (problem.getDateResolved() != null)
         pstmt.setTimestamp(3,
            Util.toTimestamp(problem.getDateResolved()));
      pstmt.setString(4, problem.getProblemID());

      // Execute the update

      pstmt.executeUpdate();
   }
   finally {
      if (pstmt != null)
         pstmt.close();
   }
}

/**
```

精彩好书见www.book118.com


```
 * Returns the problem object corresponding to
 * the current problem ID
 * @exception SQLException if a database error occurs
 */
public Problem getProblem()
   throws SQLException
{
   // Verify that a connection exists

   if (!isConnected())
         throw new SQLException("No connection");

   // Verify that a current problem ID exists

   if (problemID == null)
      throw new SQLException("No problem ID");

   PreparedStatement pstmt = null;
   ResultSet rs = null;
   Problem problem = null;

   try {

      // Prepare the query SQL

      pstmt = con.prepareStatement
      ("select * from problems where problemID = ?");
      pstmt.setString(1, problemID);

      // Execute the query

      rs = pstmt.executeQuery();
      if (rs.next())
         problem = Problem.load(rs);
   }
   finally {
      if (rs != null)
         rs.close();
      if (pstmt != null)
         pstmt.close();
   }

   // Return the problem

   return problem;
}
```

精彩好书见www.book118.com


```
/**
 * Returns the current problem search results
 */
public List getProblems()
{
   return problems;
}

/**
 * Uses the specified customer ID to query
 * the database for problems for that customer.
 * Creates a list of problem objects.
 * @exception SQLException if a database error occurs
 */
public void customerProblemsSearch(String customerID)
   throws SQLException
{
   // Verify that a connection exists

   if (!isConnected())
      throw new SQLException("No connection");

   PreparedStatement pstmt = null;
   ResultSet rs = null;
   problems = null;

   try {

      // Prepare the query SQL

      pstmt = con.prepareStatement
      ("select * from problems where customerID = ?");
      pstmt.setString(1, customerID);

      // Execute the query and copy the results
      // to a List

      rs = pstmt.executeQuery();
      problems = new LinkedList();
      while (rs.next())
         problems.add(Problem.load(rs));
   }
   finally {
      if (rs != null)
```

精彩好书见www.book118.com


```
            rs.close();
         if (pstmt != null)
            pstmt.close();
      }
   }

   /**
   * Uses the specified product ID to query
   * the database for problems for that product.
   * Creates a list of problem objects.
   * @exception SQLException if a database error occurs
   */
   public void productProblemsSearch(String productID)
      throws SQLException
   {

      // Verify that a connection exists

      if (!isConnected())
         throw new SQLException("No connection");

      PreparedStatement pstmt = null;
      ResultSet rs = null;
      problems = null;

      try {

         // Prepare the query SQL

         pstmt = con.prepareStatement
         ("select * from problems where productID = ?");
         pstmt.setString(1, productID);

         // Execute the query and copy the results
         // to a List

         rs = pstmt.executeQuery();
         problems = new LinkedList();
         while (rs.next())
            problems.add(Problem.load(rs));
      }
      finally {
         if (rs != null)
            rs.close();
         if (pstmt != null)
            pstmt.close();
```

精彩好书见www.book118.com


```
        }
    }

    /**
    * Returns the problemID.
    */
    public String getProblemID()
    {
        return problemID;
    }

    /**
    * Sets the problemID.
    * @param problemID the problemID.
    */
    public void setProblemID(String problemID)
    {
        this.problemID = problemID;
}
```

newProblem（）方法创建当前顾客和产品的一个新问题记录，对之初始化并将其加入数据库，并使用updateProblem（）方法修改它。存在通过顾客和产品检索问题的方法，它们把结果保存在可使用getProblem（）检索的一个java.util.List中。

```
    // ==============================================
    // ProblemLog methods
    // ==============================================

    /**
    * Adds a new problem log entry
    * @param log a problem log object
    * @exception SQLException if a database error occurs
    */
    public void addProblemLog(ProblemLog log)
        throws SQLException
    {
        // Verify that a connection exists

        if (!isConnected())
            throw new SQLException("No connection");

        PreparedStatement pstmt = null;

        try {

            // Prepare the insert SQL
```

精彩好书见www.book118.com


```
            pstmt = con.prepareStatement
            ("insert into problog values(?, ?, ?, ?)");
            pstmt.setString(1, log.getProblemID());
            pstmt.setTimestamp(2, Util.toTimestamp(log.getLogTime()));
            pstmt.setString(3, log.getEventID());
            pstmt.setString(4, log.getComments());

            // Execute the statement

            pstmt.executeUpdate();
        }
        finally {
            if (pstmt != null)
                pstmt.close();
        }
    }

    /**
    * Uses the specified problem ID to query
    * the database for problem log entries for
    * that problem.
    * Creates a list of problem log objects.
    * @exception SQLException if a database error occurs
    */
    public void problemLogSearch(String problemID)
        throws SQLException
    {
        // Verify that a connection exists

        if (!isConnected())
            throw new SQLException("No connection");

        PreparedStatement pstmt = null;
        ResultSet rs = null;
        problemLogs = null;

        try {

            // Prepare the query SQL

            pstmt = con.prepareStatement
            ("select * from problog where problemID = ?");
            pstmt.setString(1, problemID);
```

精彩好书见www.book118.com


```
            // Execute the query and copy the results
            // to a List

            rs = pstmt.executeQuery();
            problemLogs = new LinkedList();
            while (rs.next())
               problemLogs.add(ProblemLog.load(rs));
         }
         finally {
            if (rs != null)
               rs.close();
            if (pstmt != null)
               pstmt.close();
         }
      }

      /**
      * Returns the problemLogs.
      */
      public List getProblemLogs()
      {
         return problemLogs;
      }
   }
```

最后，模式还拥有对一个问题加入记录入口的方法，进而可以对已存在问题搜索记录入口并检索搜索结果。

WebModel类 如果仔细看Model类，会注意到它不包括已知Web的方法。这是故意的。为了进行测试，要使用一个简单的命令行视图运行模式，这样在模式中就不会用到javax.servlet或javax.servlet.http类。当在Web浏览器中运行它时，模式就要利用和其环境有关的知识了。为此，使用Model的一个子类实现这一点。

WebModel实现有3种方法：

- void init (ServletContext context) 从web.xml发布描述器中抽取应用参数。包括JDBC驱动器名、数据库URL和包含下一个可利用的产品ID的文件名。使用web.xml指定这些值使配置产品支持应用对不同环境很容易做到。
- void valueBound (HttpSessionBindingEvent event) 是组成HttpSessionBindingListener接口的两个方法之一。在这里，此方法什么都不做。但必须实现它以满足编译器需要。
- void valueUnbound (HttpSessionBindingEvent event) 是两个HttpSessionBindingListener方法中的另一个。在此方法中执行当会话超时或无效时关闭数据库连接的重要功能。

WebModel类如下：

```
package com.lyricnote.support.model;

import javax.servlet.*;
```

精彩好书见www.book118.com


```
import javax.servlet.http.*;
import java.sql.SQLException;

/**
* HTTP-specific subclass of Model. Implements session
* binding and unbinding. Allows the database connection
* to be disconnected when the session times out or is
* invalidated.
*/
public class WebModel
   extends Model
   implements HttpSessionBindingListener
{
   /**
   * Initializes the database connection
   */
   public void init(ServletContext context)
      throws ServletException
   {
      // Set the model's JDBC driver property
      // from an application-scoped value
      // in web.xml

      String jdbcDriver =
         context.getInitParameter("jdbcDriver");
      if (jdbcDriver == null)
         throw new ServletException
         ("No jdbcDriver property specified");
      setJdbcDriver(jdbcDriver);

      // Do likewise for the database URL

      String databaseURL =
         context.getInitParameter("databaseURL");
      if (databaseURL == null)
         throw new ServletException
         ("No databaseURL property specified");
      setDatabaseURL(databaseURL);

      // and the problem ID assignment file

      String problemIDFile =
         context.getInitParameter("problemIDFile");
      if (problemIDFile == null)
         throw new ServletException
         ("No problemIDFile property specified");
```

精彩好书见www.book118.com


```
        setProblemIDFile(problemIDFile);

        // Connect to the database

        try {
            connect();
        }
        catch (SQLException e) {
            throw new ServletException(e.getMessage());
        }
    }

    /**
    * Called when the model is bound to a session
    */
    public void valueBound(HttpSessionBindingEvent event)
    {
    }

    /**
    * Called when the model is removed from a session
    */
    public void valueUnbound(HttpSessionBindingEvent event)
    {
        disconnect();
    }
}
```

Util类 应用对象类的最后一个是Util，一个提供各种支持方法的实用类。这些方法包括：

- dateFormat（ ）将Date对象转换成格式化的日期字符串。
- dateTimeFormat（ ）将Date对象转换成格式化的日期和时间字符串。
- toTimestamp（ ）将Date对象转换成一个java.sql.Timestamp，以便它可以在一个PreparedStatement.setTimestamp（ ）方法中使用。
- quote（ ）如果字符串包含任何嵌入的逗号，则用引号标记将其括起来。它可被toString（ ）方法ini Customer、Product和其他事务对象类使用以确保逗号分隔取值格式的使用安全性。
- isClosingEvent（ ） 如果指定的事件ID表明问题是已解决的其中一个，则该方法返回true。

下面是Util类的列表：

```
package com.lyricnote.support.model;

import java.text.*;
import java.sql.Timestamp;
import java.util.*;
```

精彩好书见www.book118.com


```
/**
 * Utility methods used in the model package
 */
public class Util
{
   private static final SimpleDateFormat DATE_FORMAT =
      new SimpleDateFormat("yyyy-MM-dd");

   private static final SimpleDateFormat DATE_TIME_FORMAT =
      new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

   /**
    * Formats a date using the default JDBC format
    */
   public static String dateFormat(Date d)
   {
      return d == null ? "" : DATE_FORMAT.format(d);
   }

   /**
    * Formats a timestamp using the default JDBC format
    */
   public static String dateTimeFormat(Date d)
   {
      return d == null ? "" : DATE_TIME_FORMAT.format(d);
   }

   /**
    * Converts a java.util.Date to a java.sql.Timestamp
    */
   public static Timestamp toTimestamp(Date d)
   {
      return (d == null)
         ? null
         : new Timestamp(d.getTime());
   }

   /**
    * Encloses a string in quotation marks
    * if it contains a comma.
    * @param s the string
    */
   public static String quote(String s)
   {
      if (s != null) {
         if (s.indexOf(",") > -1) {
```

精彩好书见www.book118.com


```
                StringBuffer sb = new StringBuffer();
                sb.append('"');
                sb.append(s);
                sb.append('"');
                s = sb.toString();
            }
        }
        return s;
    }

    /**
    * Returns true if the specified event ID
    * represents a "close" action
    * @param eventID the event ID
    */
    public static final boolean isClosingEvent(String eventID)
    {
        return (
            eventID.equals("CNB") ||
            eventID.equals("CCP") ||
            eventID.equals("CFX"));
    }
}
```

3) 测试框架

MVC结构的最大益处之一是每个组件都可以分别测试。在开发期间，能够单元测试模式特别有用，并且其API可作为方法被加入和修改，在这一节，介绍一个实现此功能的命令行shell。

Shell类　test.Shell类是一个充作控制器和com.lyricnote.support.model.Model视图的单机Java应用。像Unix shell或Windows命令提示一样，Shell类提示输入命令，执行它们并显示出结果。这些命令的语法是相应的调用模式中方法的Java的语法以及一些列出可利用方法、显示帮助文本和类似控制功能的命令。Shell为实现模式的每一部分都提供了一种简单的方式，而没有带来Web环境中GUI复杂性的增加。

像Model类一样，Shell相当长，下面一次讨论一段：

```
package test;

import com.lyricnote.support.model.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import java.sql.*;
import java.util.*;

/**
* An interactive shell for testing the product support
```

精彩好书见www.book118.com


```
* application model.
*/
public class Shell
{
   private static String PROMPT = "SHELL> ";
   private Model model;
   private InputStream stream;
   private boolean interactive;

   // ============================================
   // Class methods
   // ============================================

   /**
   * Mainline
   */
   public static void main(String[] args)
      throws Exception
   {
      Shell shell = new Shell(new Model());
      shell.run();
   }

   /**
   * Displays help text for this shell
   */
   protected static void help()
   {
      String[] text = {
         "",
         "Invoke a method by name,"
         + " or any of the following commands:",
         "",
         "quit - exits from the shell",
         "help - displays this help text",
         "methods - displays a list of model methods",
         "include <filename> - executes an included file",
         "",
      };
      for (int i = 0; i < text.length; i++)
         System.out.println(text[i]);
   }

   /**
   * Extracts a quoted string argument value
   * from a method call
```

精彩好书见www.book118.com


```
    */
    protected static String getArgument(String line)
    {
        String arg = null;
        int p = line.indexOf("(\"");
        if (p != -1) {
            p += 2;
            int q = line.indexOf("\")", p);
            if (q != -1) {
                arg = line.substring(p, q);
            }
        }
        return arg;
    }

    // ==========================================
    // Constructors
    // ==========================================

    /**
    * Creates a new Shell with input from System.in
    * @param model the model to be used
    */
    public Shell(Model model)
    {
        this(model, System.in);
    }

    /**
    * Creates a new Shell with input from
    * the specified input stream
    * @param model the model to be used
    * @param stream the input stream
    */
    public Shell(Model model, InputStream stream)
    {
        this.model = model;
        this.stream = stream;
        this.interactive = (stream == System.in);
    }
```

Shell使用一个简单的main（）方法创建模式和shell的实例，然后调用此shell的run（）方法。命令输入源最初是System.in，但正如所见，它也可以是保存在一个文件中并使用include命令处理的一个命令集。

```
    // ==========================================
```

精彩好书见www.book118.com


```
// Main read/execute loop
// ==========================================

/**
* Runs the shell
*/
public void run() throws Exception
{
   // Open a line reader over the input stream

   BufferedReader in =
      new BufferedReader(
      new InputStreamReader(stream));

   // Read and execute each line

   while (true) {

      if (interactive)
         System.out.print(PROMPT);

      String line = in.readLine();
      if (line == null)
         break;

      // Parse and execute the command

      try {

         if (line.equals("quit"))
            break;
         else if (line.startsWith("get"))
            doGet(line);
         else if (line.startsWith("set"))
            doSet(line);
         else if (line.startsWith("customerSearch"))
            customerSearch(line);
         else if (line.startsWith("productSearch"))
            productSearch(line);
         else if (line.startsWith("productProblemsSearch"))
            productProblemsSearch(line);
         else if (line.startsWith("customerProblemsSearch"))
            customerProblemsSearch(line);
         else if (line.startsWith("problemLogSearch"))
            problemLogSearch(line);
```

精彩好书见www.book118.com


```
            else if (line.startsWith("help"))
                help();
            else if (line.startsWith("methods"))
                methods();
            else if (line.startsWith("inc"))
                include(line);
            else if (line.startsWith("is"))
                doGet(line);
            else if (line.startsWith("connect"))
                doConnect();
            else if (line.startsWith("disconnect"))
                doDisconnect();
            else if (line.startsWith("newProblem"))
                doNewProblem();

            // none of the above

            else
                System.out.println
                    ("Unrecognized command [" + line + "]");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    in.close();
    if (interactive)
        doDisconnect();
}
```

run ( ) 方法在输入流上打开一个字符行读取器并开始提示输入以及执行命令。解析器是一个检验指定方法名及调用执行它们包装方法的if语句的简单列表。对开头时带有get、is或set Shell的命令，使用映射找到模式中相应的getter或setter方法并调用它们。如果遇到任何错误，将其写入System.out。

下面是该类的其余部分，由从主循环中调用的方法组成（按希腊字母列出）：

```
/**
 * Invokes the customer problems search method
 */
protected void customerProblemsSearch(String line)
    throws Exception
{
    String arg = getArgument(line);
    model.customerProblemsSearch(arg);
}
```

精彩好书见www.book118.com


```
/**
* Invokes the customer search method
*/
protected void customerSearch(String line)
    throws SQLException
{
    String arg = getArgument(line);
    model.customerSearch(arg);
}

/**
* Invokes the connect command
*/
protected void doConnect()
{
    try {
        System.out.println("Connecting...");
        model.connect();
        System.out.println("Connected");
    }
    catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}

/**
* Invokes the disconnect command
*/
protected void doDisconnect()
{
    System.out.println("Disconnecting...");
    model.disconnect();
    System.out.println("Disconnected");
}

/**
* Executes a "get" method
*/
protected void doGet(String line) throws Exception
{
    if (!interactive)
        System.out.println(line);

    // Get the read method name

    int p = line.indexOf("(");
```

精彩好书见www.book118.com


```
        if (p == -1)
            p = line.length();
        String readMethodName = line.substring(0, p).trim();

        // Lookup the read methods to see
        // if this one is found

        BeanInfo bi =
            Introspector.getBeanInfo(model.getClass());

        PropertyDescriptor[] pds =
            bi.getPropertyDescriptors();

        for (int i = 0; i < pds.length; i++) {
            PropertyDescriptor pd = pds[i];
            Method method = pd.getReadMethod();
            if (method != null) {
                String name = method.getName();
                if (name.equals(readMethodName)) {

                    // This method is the read method
                    // for this property.
                    // Invoke it and print the result

                    Object[] args = {};
                    Object result = method.invoke(model, args);
                    System.out.println(result);
                    return;
                }
            }
        }
        throw new IllegalArgumentException
        ("No " + readMethodName + " method found");
    }

    /**
    * Invokes the newProblem method
    */
    protected void doNewProblem() throws Exception
    {
        model.newProblem();
        System.out.println(model.getProblemID());
    }

    /**
    * Executes a "set" method
```

精彩好书见www.book118.com


```
*/
protected void doSet(String line) throws Exception
{
   if (!interactive)
      System.out.println(line);

   // Line should look like this:
   //
   // setSearchArgument("value")

   int p = line.indexOf("(");
   if (p == -1)
      throw new IllegalArgumentException
      ("No open parenthesis found");

   int q = line.indexOf(")", p);
   if (q == -1)
      throw new IllegalArgumentException
      ("No close parenthesis found");

   String writeMethodName = line.substring(0, p);
   String argument = line.substring(p+1, q).trim();

   // Argument must be a quoted string

   if (!(argument.startsWith("\"") &&
         argument.endsWith("\"")))
      throw new IllegalArgumentException
      ("Argument must be a quoted string");

   // Strip off the quotes

   argument = argument.substring(1, argument.length()-1);

   // Find the set method and execute it

   BeanInfo bi = Introspector.getBeanInfo(model.getClass());

   PropertyDescriptor[] pds = bi.getPropertyDescriptors();

   for (int i = 0; i < pds.length; i++) {
      PropertyDescriptor pd = pds[i];
      Method method = pd.getWriteMethod();
      if (method != null) {
         String name = method.getName();
         if (name.equals(writeMethodName)) {
```

精彩好书见www.book118.com


```
                    // This method is the write method
                    // for this property

                    Object[] args = { argument };
                    Object result = method.invoke(model, args);
                    return;
                }
            }
        }

        throw new IllegalArgumentException
        ("No " + writeMethodName + " method found");
    }

    /**
    * Runs a subshell for the file specified in
    * the include statement.
    * @param line an "include <path>/file" statement
    */
    protected void include(String line) throws Exception
    {
        if (!interactive)
            System.out.println(line);

        try {

            // Get the name of the file to be included

            StringTokenizer st = new StringTokenizer(line);
            st.nextToken();
            if (!st.hasMoreTokens())
                throw new IllegalArgumentException
                ("No file name specified for include");
            String fileName = st.nextToken();

            // Verify that file exists

            File file = new File(fileName);
            if (!file.exists())
                throw new IllegalArgumentException
                (fileName + " not found");

            // Run the subshell
```

精彩好书见www.book118.com


```
            System.out.println("Including " + fileName);
            new Shell(this.model, new FileInputStream(file)).run();
            System.out.println("Done including " + fileName);
        }
        catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    /**
    * Shows the public methods available in the model
    */
    protected void methods()
    {
        // Get the list of declared methods

        Class cls = Model.class;
        Method[] methods = cls.getDeclaredMethods();
        System.out.println(methods.length + " methods:");

        // Print the list

        for (int i = 0; i < methods.length; i++) {
            Method method = methods[i];
            String name = method.getName();
            Class[] parameterTypes = method.getParameterTypes();
            StringBuffer sb = new StringBuffer();
            sb.append(name);
            sb.append("(");
            for (int j = 0; j < parameterTypes.length; j++) {
                Class parmClass = parameterTypes[j];
                if (j > 0)
                    sb.append(",");
                sb.append(parmClass.getName());
            }
            sb.append(")");
            String s = sb.toString();
            System.out.println(" " + s);
        }
    }

    /**
```

精彩好书见www.book118.com


```
    * Invokes the problemLogSearch method
    */
    protected void problemLogSearch(String line)
        throws Exception
    {
        String id = getArgument(line);
        model.problemLogSearch(id);
    }
    /**
    * Invokes the productProblemsSearch method
    */
    protected void productProblemsSearch(String line)
        throws Exception
    {
        String id = getArgument(line);
        model.productProblemsSearch(id);
    }

    /**
    * Invokes the productSearch method
    */
    protected void productSearch(String line) throws Exception
    {
        String arg = getArgument(line);
        model.productSearch(arg);
    }
}
```

为获知该shell的用处，下面看看其行为。当调用该shell类，其main（）方法创建了Shell的实例并将其传递给Model一个新实例。如果键入help命令，会看到下面帮助文本：

```
P:\classes\test>java -classpath .. test.Shell
SHELL> help

Invoke a method by name, or any of the following commands:

quit       - exits from the shell
help       - displays this help text
methods    - displays a list of model methods
include <filename> - executes an included file

SHELL>
```

开始，先来看看哪些Model方法可以调用。键入methods命令，显示如下：

```
SHELL> methods
33 methods:
    connect()
```

精彩好书见www.book118.com


```
    customerProblemsSearch(java.lang.String)
    customerSearch(java.lang.String)
    disconnect()
    getProblemID()
    newProblem()
    productProblemsSearch(java.lang.String)
    productSearch(java.lang.String)
    addProblemLog(com.lyricnote.support.model.ProblemLog)
    assignProblemID(java.lang.String)
    getCustomer()
    getCustomerID()
    getCustomerProducts()
    getCustomers()
    getDatabaseURL()
    getEmployee(java.lang.String)
    getJdbcDriver()
    getProblem()
    getProblemIDFile()
    getProblemLogs()
    getProblems()
    getProduct()
    getProductID()
    getProducts()
    isConnected()
    problemLogSearch(java.lang.String)
    setCustomerID(java.lang.String)
    setDatabaseURL(java.lang.String)
    setJdbcDriver(java.lang.String)
    setProblemID(java.lang.String)
    setProblemIDFile(java.lang.String)
    setProductID(java.lang.String)
    updateProblem(com.lyricnote.support.model.Problem)
SHELL>
```

如何得到的该信息呢？看看实现methods命令的Shell方法。会发现它调用了Model类的getDeclaredMethods（）方法，然后打印出结果数组。这里显示了可以在shell中调用的每个方法的列表。

在开发期间，当向模式中加入新的方法，会看到它们被自动加入到列表中。在Shell类中，需要加入的内容是run（）方法中的if语句和简单调用模式方法并打印结果的子过程。大略看来，甚至不必对属性的getter和setter方法执行这些动作。

回到shell会话，就会知道没有一个数据库连接所做的事情很有限，因此下面调用报告连接是否已建立的模式方法：

```
SHELL> isConnected();
false
```

精彩好书见www.book118.com


```
SHELL>
```

当shell见到以is或get开始的命令，它将其解释成模式的属性访问器方法的调用。shell在其doGet（ ）方法中处理这样的解释过程。doGet（ ）使用JavaBean自测程序得到Model类中getter方法的列表，然后将命令行中的方法名与getter方法的名字比较进行匹配操作。当shell发现一个匹配的方法时，调用该方法并返回结果。

类似的方案可用于setter方法。任何以set开始的命令被发送到doGet（String line）方法，它从命令行中抽出参数，进入同一个自测程序找到适当的set方法，并调用带有命令行参数的方法。

这成为取得和设置模式属性的一种固有方式。当调用shell的isConnected（ ）方法时，它将调用传递给模式的isConnected（ ）方法，该方法报告没有连接发生。

为创建连接，模式需要JDBC驱动器名和数据库URL。设置这些属性，然后调用connect（ ）方法：

```
SHELL> setJdbcDriver("org.enhydra.instantdb.jdbc.idbDriver");
SHELL> setDatabaseURL("jdbc:idb:D:/jspcr/Chap19/database/db.prp");
SHELL> connect();
Connecting...
Connected
SHELL> isConnected();
true
SHELL>
```

这次，isConnected（ ）方法指出连接可以利用。如果正在测试的是数据库过程，则应该调用disconnect（ ），然后是quit命令：

```
SHELL> disconnect();
Disconnecting...
Disconnected
SHELL> quit
```

反复测试连接，键入很长一串命令有点乏味。为此，shell给出一个include命令。它允许从一个文件中读取shell命令，并在一个subshell下执行。此功能是递归的，因此包含的模块本身可以包含其他模块。

一个马上可以使用的包含模块是执行数据库连接。它实质上被所有测试使用。此模块包含执行连接需要键入的3个命令：

```
setJdbcDriver("org.enhydra.instantdb.jdbc.idbDriver");
setDatabaseURL("jdbc:idb:D:/jspcr/Chap19/database/db.prp");
connect();
```

现在可以启动一个shell，简单调用外部命令集：

```
P:\classes\test>java -classpath .. test.Shell
SHELL> include connect.inc
setJdbcDriver("org.enhydra.instantdb.jdbc.idbDriver");
setDatabaseURL("jdbc:idb:D:/jspcr/Chap19/database/db.prp");
Connecting...
Connected
```

精彩好书见www.book118.com


```
SHELL> isConnected();
true
```

注意，当被读取时，包含模块的内容被反馈到控制台。

连接建立起来后，就可以继续测试模式的任意部分。复制GUI应用执行的步骤序列，查看结果是否如预料的一样。下面搜索ScoreWriter产品，并通过其问题报告查找问题：

```
SHELL> productSearch("Score");
SHELL> getProducts();
[023500,ScoreWriter,0040,0140,0070]
SHELL> productProblemsSearch("023500");
SHELL> getProblems();
[G0000179,Can't get triplets to work,3,2001-01-14 18:40:39,
2001-01-14 18:42:53,WAGNER01,023500]
SHELL> problemLogSearch("G0000179");
SHELL> getProblemLogs();
[
G0000179,2001-01-14 18:41:09,RPS,They just don't work!,
G0000179,2001-01-14 18:42:20,COM,Told customer to try F5,
G0000179,2001-01-14 18:42:53,CCP,That did it]

SHELL> quit
Disconnecting...
Disconnected

P:\classes\test>
```

调用productSearch（ ）方法，然后是getProducts（ ）方法，会看到匹配搜索参数的产品列表（这里为一个元素列表）。对象表示为列出产品ID，其名字、产品支持人员的雇员ID、首要开发者和首要测试者的逗号分隔取值的字符串。然后productProblemsSearch（ ）方法找出对ScoreWriter所报告的问题列表（这里也只是一个元素列表）。最后，problemLogSearch（ ）和getProblemLogs（ ）方法对所选中的问题取得记录人口[1]。

4) 使用模式

模式组件的开发已经完成。在开发的初期，可能按前后顺序编写Model和Shell类，这样的话，模式的每一部分都可分别被测试。当发现故障时，可以返回到shell将其再现，而不必启动和停止Web服务器或在servlet记录中查询调试人口。这为编写应用其余部分打下可靠的基础。

当然。设计模式有多种方式。可以不必使用诸如Model类的应用容器。可以直接将事务对象作为一个HtpSession中的JavaBean处理。然而在这一节你看到的是一个满足各种类型应用需求的可行设计，却没有带来控制器和视图内不当的复杂性的增加。

19.4.2 视图类

模式可被捆绑到一个单机Java应用，然而，产品支持系统至少被4种角色的用户访问：电话中心代理、产品支持专家、开发商和测试者。另外，管理者可能需要进行质量统计，如队列中

1 列出的结果为适应行宽被重新格式化。

精彩好书见www.book118.com

一个问题等待时间的平均长度、需要解决问题的顾客反馈的平均数目和一个特定产品突出缺陷的数目。为此，最好的系统操作环境可能是公司的intranet，并且表示层由JSP页面组成。在这一节，会看到如何使用JSP页面作为捆绑模式的视图。

在系统中存在3个常用的入口点：

- 通过顾客　接到顾客电话后，电话中心代理首先通过顾客名字利用customer表的一个字母搜索查找顾客的ID，从匹配列表中选中一个顾客后，代理查看顾客详细信息，包括此顾客已购买的产品和此顾客报告问题的历史记录。在这里，电话中心代理可以输入一个新的问题报告或提供已存在问题的状态。
- 通过产品　产品支持人员、开发商、测试者针对的都是特定的产品而不是顾客。因此，其系统最初的视图是通过产品。它们可以使用product表的字母搜索按名字查找产品并从中看到突出问题的列表。
- 通过问题　任何系统用户可能已经知道针对特定缺陷的问题的ID。如果它们需要修改问题记录，则可以使用提示输入特定问题ID的窗体。

这3种情况中，应用最终都要显示一个特定问题详细信息的视图。从中，用户可以修改问题描述和严重程度，并将问题路由到另一部门或结束问题。最后的JSP视图是一个显示应用了哪种行为的确认屏幕。

图19-3解释了此应用流程。每个矩形框都表示一个特定的JSP页面。圆圈表示控制行为，下面会讲到。现在记住，控制行为是引起模式改变并使得下一视图被显示的起因。从JSP视图到控制行为的箭头标注了用户对于视图所采取行为的类型：从列表中选择、输入一个搜索参数或点击一个确认按钮。

下面详细讨论每个JSP视图页面。然面，首先需要查看某些支持类。

1. 支持页面

代码中的两段都是JSP视图页面。不是在每个JSP文件中复制它们，而是将其保存在单独的文本文件中并使用<%@include%>指令包含它们[1]。

InitModel.jsp　每个JSP视图都需要将模式声明为一个会话bean并确保数据库连接可以利用。这个常用的功能通过在视图页面中包含下列代码完成：

```
<%-- Define and initialize the model --%>

<jsp:useBean
   id="model"
   scope="session"
   class="com.lyricnote.support.model.WebModel">
<% model.init(application); %>
</jsp:useBean>

<%-- Provide an alias for the controller servlet --%>
```

1 为什么是include伪指令而不是<jsp:include>呢？因为被包含的一段定义和使用的被包含模块和包含模块都要通用的常量。

精彩好书见www.book118.com

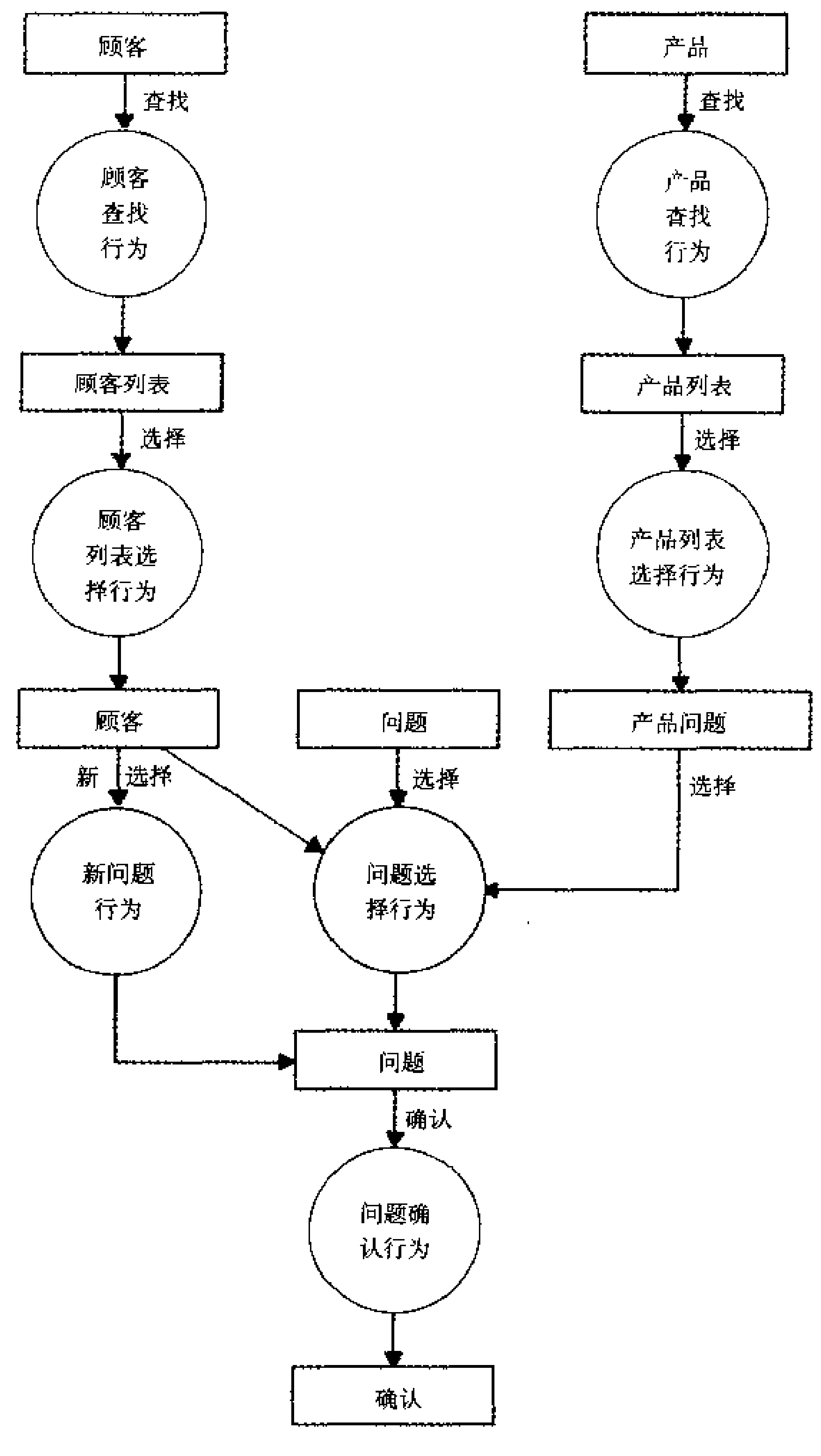


图19-3 视图/控制器交互图解

```
<%
    String BASEURL = request.getContextPath();
    String CONTROLLER = BASEURL + "/servlet/controller";
%>
```

精彩好书见www.book118.com

initModel.jsp中此段代码做3件事情：

- 声明具有会话范围的名为model的JavaBean，并使com.lyricnote.support.model.WebModel对象初始化。
- 调用模式的init（ ）方法从web.xml中抽取应用变量并建立一个会话已知的数据库连接。此方法只在模式第一次捆绑到会话时被调用一次。
- 初始化指定应用基本URL和控制器servlet名字的常量。这些常量在URL和视图其他位置的窗体行为属性中使用。

Banner.jsp 对于所有JSP页面中的常用风格和外观，使用了一个标准的头标。此头标保存在名为Banner.jsp的文件中，包含一个包含公司徽标和标准导航链接的HTML表格。

```
<TABLE BORDER=0 CELLSPACING=3 CELLPADDING=3 WIDTH=500>
<TR>
   <TD><IMG SRC="<%= BASEURL %>/images/logo.jpg"></TD>
</TR>
<TR>
<TD CLASS="menucell" ALIGN="RIGHT">
   <A CLASS="menuitem"
      HREF="<%= BASEURL %>/Problems.jsp">Problems</A>

   <SPAN CLASS="menuitem">|</SPAN>

   <A CLASS="menuitem"
      HREF="<%= BASEURL %>/Products.jsp">Products</A>

   <SPAN CLASS="menuitem">|</SPAN>

   <A CLASS="menuitem"
      HREF="<%= BASEURL %>/Customers.jsp">Customers</A>
</TR>
</TABLE>
```

InitModel.jsp和Banner.jsp都不是设计为用户直接调用。为防止这一点，它们被保存在/WEB-INF文件夹内。这使得它们对应用本身可视，但对Web用户不可视。

ErrorPage.jsp 最后一个支持模块是错误页面。如下所示：

```
<%@ page session="false" %>
<%@ page import="java.io.PrintWriter" %>
<%@ page isErrorPage="true" %>

<HTML>
<HEAD>
<TITLE>Error Page</TITLE>
<LINK REL="stylesheet" HREF="style.css">
</HEAD>
<BODY>
```

精彩好书见www.book118.com


```
<H3>Error</H3>
The following error occurred:
<PRE>
<%
   exception.printStackTrace(new PrintWriter(out));
%>
</PRE>
</BODY>
</HTML>
```

ErrorPage.jsp只用于显示任何未捕获溢出的栈轨迹。在系统开发期间这很有用，但在产品系统中可能应由某种更友好的页面所替换。

2. JSP视图页面

使用图19-3作为视图页面的映射，下面分别看一看9个JSP视图页面：

- Customers.jsp 提示顾客输入搜索参数。
- CustomersList.jsp 从顾客搜索结果列表中选择。
- Customer.jsp 单个顾客记录的详细信息视图。
- problem.jsp 单个问题记录的详细信息视图。
- Confirm.jsp 问题修改后显示的确认屏幕。
- products.jsp 提示输入产品搜索参数。
- poductsList.jsp 从产品搜索结果列表中选择。
- ProductProblems.jsp 从产品的问题列表中选择。
- Problems.jsp 提示输入产品ID。

Customers.jsp 电话中心代理的最初的入口点典型情况下是顾客的搜索视图。如下所示：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Customer Search</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Customer Search</H3>
<FORM
   METHOD="POST"
   ACTION="<%= CONTROLLER %>/Customers/Search">
```

精彩好书见www.book118.com


```
<B>Customer name</B>:
<INPUT TYPE="TEXT" NAME="customerSearchArgument" SIZE="20">
<INPUT TYPE="SUBMIT" VALUE="Search">
</FORM>

</BODY>
</HTML>
```

包含模式的初始化和标题代码后，此JSP页面使用一个HTML窗体提示输入一个顾客名搜索字符串。例如，如果电话中心代理正在与一个名为Eleanor Wagner的顾客通话，代理搜索包含字母W的名字，如图19-4所示。当代理点击搜索按钮时，窗体被确认，一个执行搜索的servlet被调用（在本章后面控制器类一节讨论）。

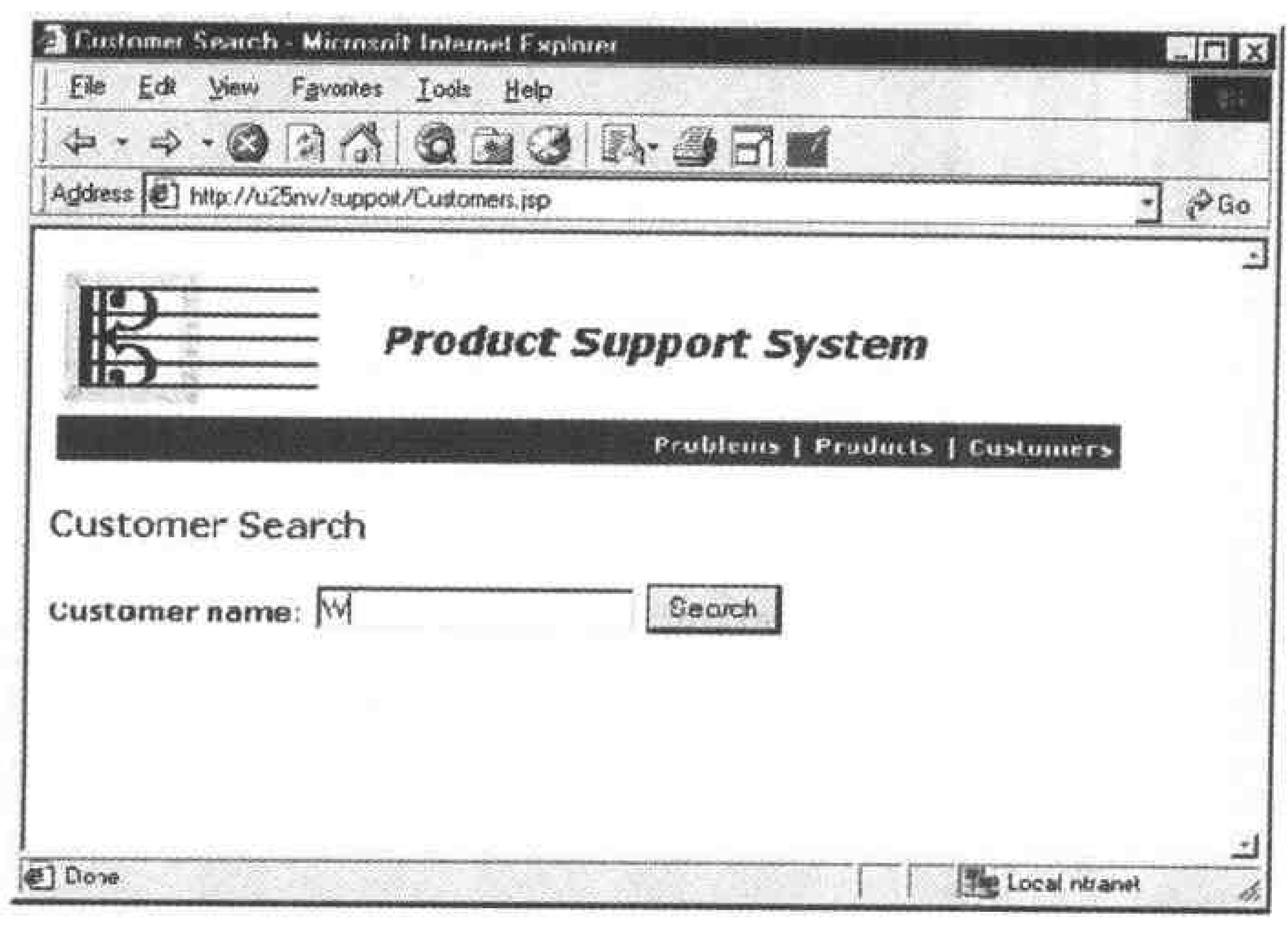


图19-4 顾客搜索页面

CustomersList.jsp 通过顾客名搜索的结果被保存在模式的一个java.util.List中，CustomersList.jsp页面抽取列表并将顾客ID显示为一个超级链接。其JSP源码如下：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="java.util.*" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
```

精彩好书见www.book118.com


```
<HEAD>
<TITLE>Customers List</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Customers List</H3>
<TABLE BORDER=0 CELLSPACING=5 CELLPADDING=0>
   <TR>
      <TH ALIGN=LEFT>Customer ID</TH>
      <TH ALIGN=LEFT>Customer Name</TH>
   </TR>

<%
   List list = model.getCustomers();
   if (list != null) {
      Iterator it = list.iterator();
      while (it.hasNext()) {
         Customer customer = (Customer) it.next();

         // Get the customer select URL

         String customerID = customer.getCustomerID();
         String selectURL = CONTROLLER +
            "/CustomersList/Select?customerID="
            + customerID;
%>
   <TR>
      <TD><A HREF="<%= selectURL %>"><%= customerID %></A></TD>
      <TD><%= customer.getName() %></TD>
   </TR>
<%
      }
   }
%>
</TABLE>

</BODY>
</HTML>
```

带有名字包含字母W的顾客列表的Web页面如图19-5所示。

Customer.jsp　从列表中选择Eleanor Wagner后，电话中心代理查看顾客详细信息页面，如图19-6所示。此页面有3部分：

• 左上角　包含顾客ID、名字和电话号码。

精彩好书见www.book118.com

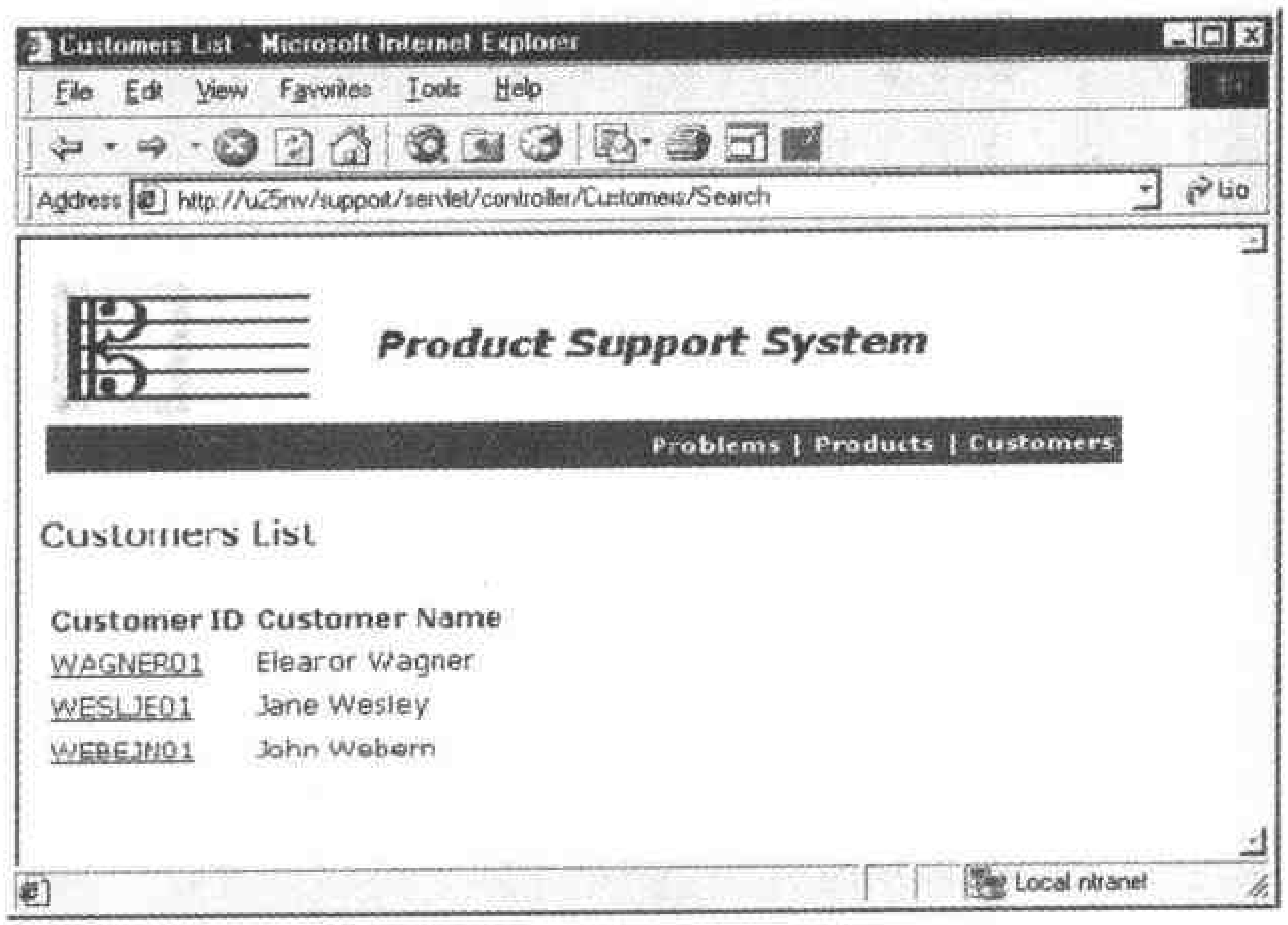


图19-5 顾客搜索结果页面

• 右上角 是一个顾客购买的产品列表。此列表按购买日期次序降序排列，产品名字是用来报告一个新问题的超级链接。
• 底部 是顾客报告的问题历史记录。这里，会看到一个已经被解决的问题。

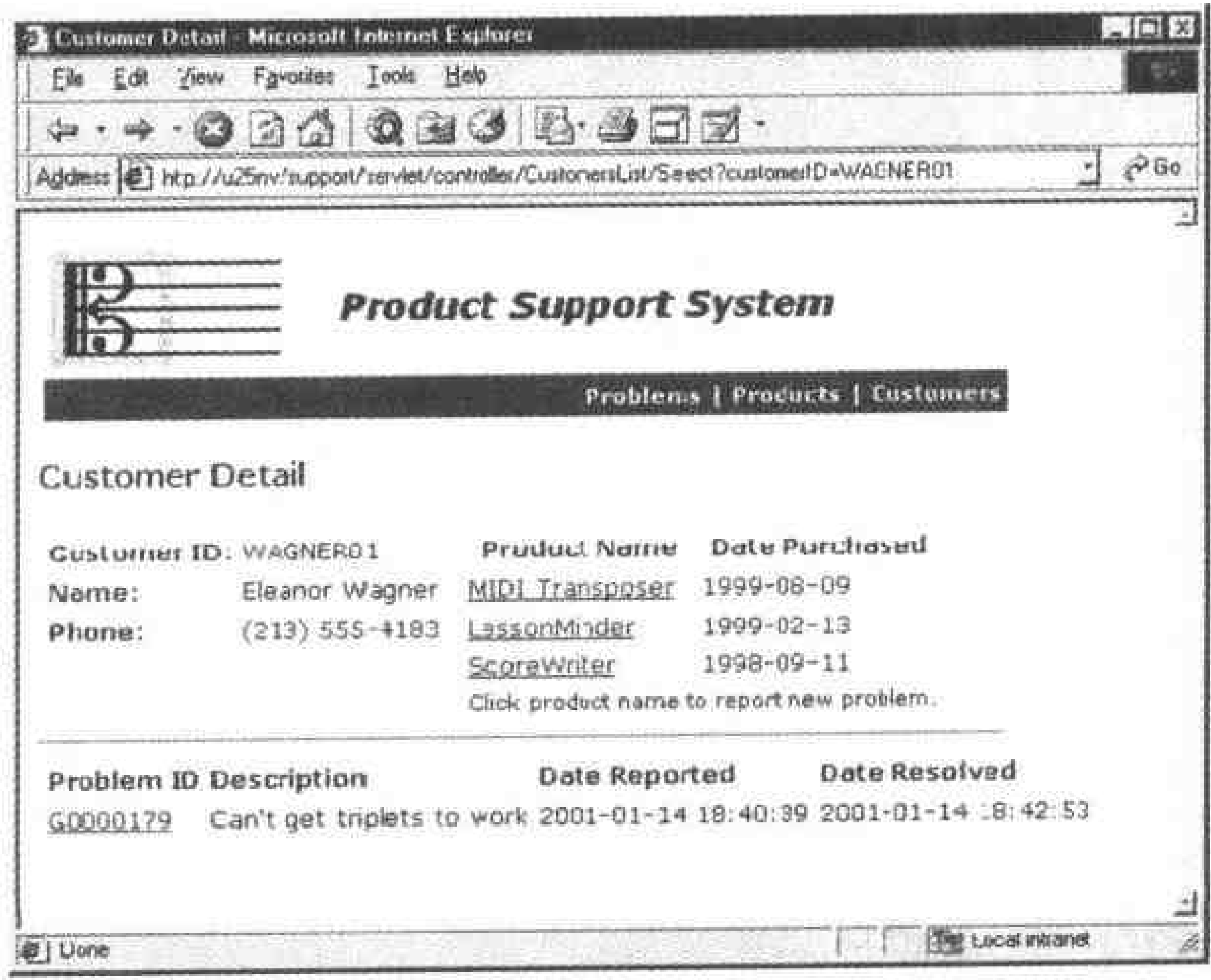


图19-6 顾客详细信息页面

精彩好书见www.book118.com

产品列表使电话中心代理可以判断出Wagner被授权提供支持的产品。

Customer.jsp源码如下：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="java.util.*" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Customer Detail</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<% Customer customer = model.getCustomer(); %>

<H3>Customer Detail</H3>

<%-- Customer information and products purchased --%>

<TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0>
<TR>

   <%-- Left side --%>

   <TD VALIGN=TOP>
   <TABLE BORDER=0 CELLSPACING=5 CELLPADDING=0>
   <TR>
      <TD><B>Customer ID:</B></TD>
      <TD><%= customer.getCustomerID() %></TD>
      <TD ROWSPAN=3>
      </TD>
   </TR>
   <TR>
      <TD><B>Name:</B></TD>
      <TD><%= customer.getName() %></TD>
   </TR>
   <TR>
      <TD><B>Phone:</B></TD>
      <TD><%= customer.getPhone() %></TD>
   </TR>
```

精彩好书见www.book118.com


```
    </TABLE>
    </TD>

    <%-- Right side --%>

    <TD VALIGN=TOP>
    <TABLE BORDER=0 CELLSPACING=5 CELLPADDING=0>
       <TR>
          <TH>Product Name</TH>
          <TH>Date Purchased</TH>
       </TR>
<%
    List products = model.getCustomerProducts();
    if ((products != null) && (products.size() > 0)) {
       Iterator it = products.iterator();
       while (it.hasNext()) {
          CustomerProduct custprod =
             (CustomerProduct) it.next();
          model.setProductID(custprod.getProductID());
          Product product = model.getProduct();
          String productName = product.getName();
          String datePurchased =
             Util.dateFormat(custprod.getDatePurchased());
          String NEW_URL =
             CONTROLLER + "/Customer/NewProblem"
             + "?customerID=" + custprod.getCustomerID()
             + "&productID-" + custprod.getProductID() ;
%>
          <TR>
             <TD>
             <A HREF-"<%= NEW_URL %>"><%= productName %></A>
             </TD>
             <TD><%= datePurchased %></TD>
          </TR>
<%
          }
%>
          <TR>
             <TD CLASS="fineprint" COLSPAN=2>
                Click product name to report new problem.
             </TD>
          </TR>
<%
    }
%>
    </TABLE>
```

精彩好书见www.book118.com


```
      </TD>
   </TR>
   </TABLE>

   <HR WIDTH=506 ALIGN=LEFT>

   <%-- Problems Reported --%>

   <TABLE BORDER=0 CELLSPACING=5 CELLPADDING=0>

      <TR>
         <TH ALIGN=LEFT>Problem ID</TH>
         <TH ALIGN=LEFT>Description</TH>
         <TH ALIGN=LEFT>Date Reported</TH>
         <TH ALIGN=LEFT>Date Resolved</TH>
      </TR>
   <%
      List list = model.getProblems();
      if (list != null) {
         Iterator it = list.iterator();
         while (it.hasNext()) {
            Problem problem = (Problem) it.next();

            // Create the problem select URL

            String problemID = problem.getProblemID();
            String selectURL = CONTROLLER +
               "/Problems/Select?problemID="
               + problemID;

            String problemDescription = problem.getDescription();

            // Get the reported and resolution dates

            String dateReported =
               Util.dateTimeFormat(problem.getDateReported());
            String dateResolved =
               Util.dateTimeFormat(problem.getDateResolved());
   %>
      <TR>
         <TD><A HREF="<%= selectURL %>"><%= problemID %></A></TD>
         <TD><%= problemDescription %></TD>
         <TD><%= dateReported %></TD>
         <TD><%= dateResolved %></TD>
      </TR>
   <%
```

精彩好书见www.book118.com


```
            }
        }
    %>
    </TABLE>

    </BODY>
    </HTML>
```

此JSP源码基本由HTML组成，并在与模式进行交互的scriptlet和表达式中包含一点Java。使用模式的getCustomer（）方法从模式中检索出当前的Customer对象，这使得JSP表达式可以获得并填充顾客ID、名字和电话域。对于右边的产品列表，JSP调用模式的getCustomerProducts（）方法，返回CustomerProduct对象的列表。该模式被用来依次获得每个产品的名字和购买日期。类似技术可用来获得在页面底部显示的顾客前面问题部分的列表。

Problem.jsp　Ms.Wagner报告她对ScoreWriter产品有问题。当她试图输入Db，软件反而用C#替代它。电话中心代理点击ScoreWriter超级链接，产生了如图19-7所示的JSP视图页面。

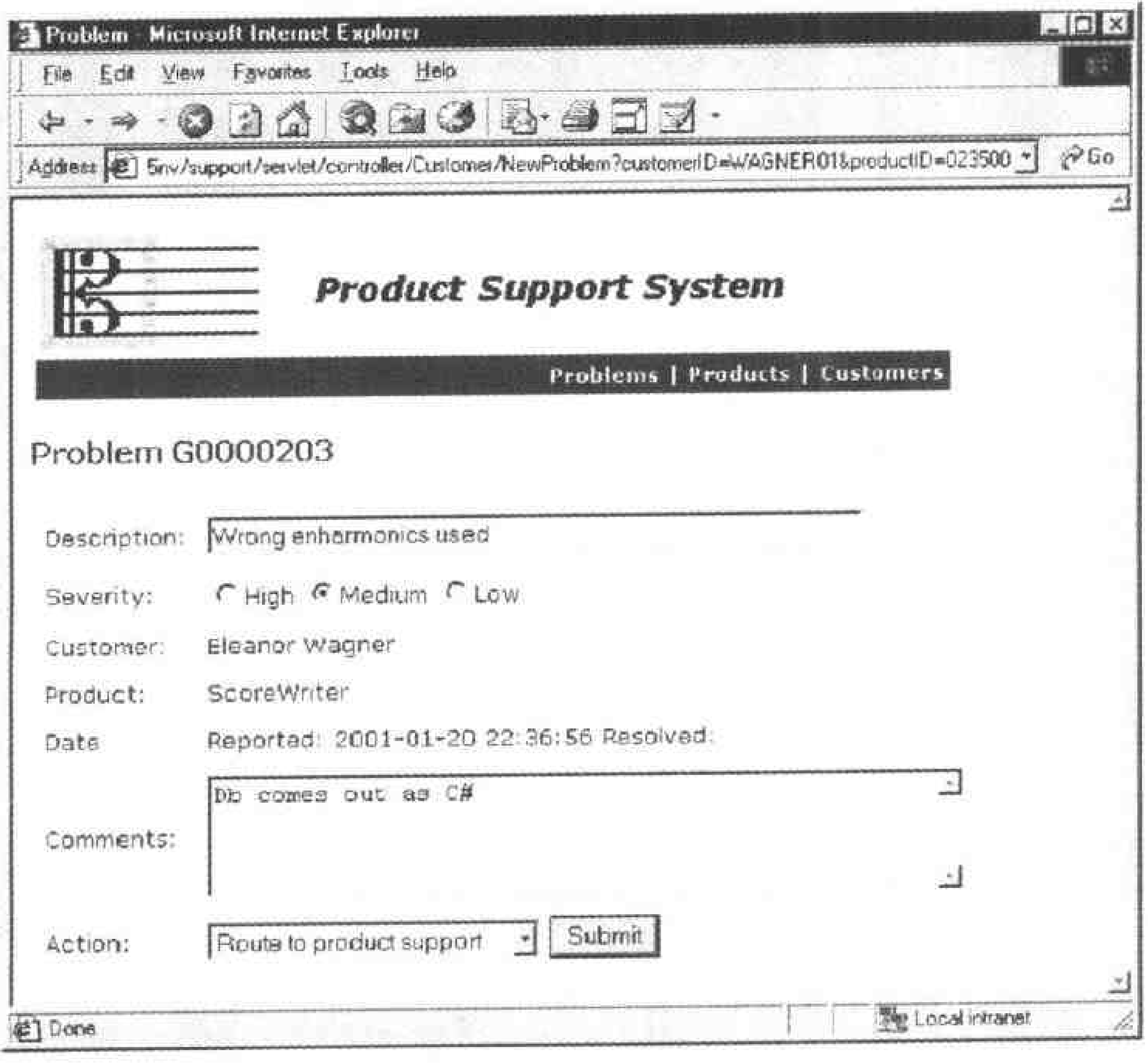


图19-7　问题细节页面

精彩好书见www.book118.com

输入问题描述、严重程度和顾客的注释后，代理点击确认按钮创建问题记录。

以下是problem.jsp源码：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="java.util.*" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Problem</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<%
   // Retrieve the problem from the model

   Problem problem = model.getProblem();

   // Get the customer

   model.setCustomerID(problem.getCustomerID());
   Customer customer = model.getCustomer();

   // Get the product

   model.setProductID(problem.getProductID());
   Product product = model.getProduct();

   // Determine the severity

   int severity = problem.getSeverity();
   String checked1 = (severity == 1) ? "CHECKED" : "";
   String checked2 = (severity == 2) ? "CHECKED" : "";
   String checked3 = (severity == 3) ? "CHECKED" : "";
%>

<H3>Problem <%= problem.getProblemID() %></H3>

<FORM METHOD="POST" ACTION="<%= CONTROLLER %>/Problem/Submit">
<INPUT TYPE="HIDDEN" NAME="problemID"
```

精彩好书见www.book118.com


```
   VALUE="<%= problem.getProblemID() %>">
<TABLE BORDER=0 CELLSPACING=5 CELLPADDING=3>
<TR>
   <TD>Description:</TD>
   <TD>
      <INPUT
         NAME="description"
         TYPE="text"
         VALUE="<%= problem.getDescription() %>"
         SIZE="50"
         >
   </TD>
</TR>
<TR>
   <TD>Severity:</TD>
   <TD>
      <INPUT
         NAME="severity"
         TYPE="radio"
         VALUE="1"
         <%= checked1 %>
         >High
      <INPUT
         NAME="severity"
         TYPE="radio"
         VALUE="2"
         <%= checked2 %>
         >Medium
      <INPUT
         NAME="severity"
         TYPE="radio"
         VALUE="3"
         <%= checked3 %>
         >Low
   </TD>
</TR>
<TR>
   <TD>Customer:</TD>
   <TD><%= customer.getName() %></TD>
</TR>

<TR>
   <TD>Product:</TD>
   <TD><%= product.getName() %></TD>
</TR>
<TR>
```

精彩好书见www.book118.com


```
      <TD>Date</TD>
      <TD>
         Reported:
         <%= Util.dateTimeFormat(problem.getDateReported()) %>
         Resolved:
         <%= Util.dateTimeFormat(problem.getDateResolved()) %>
      </TD>
   </TR>
   <TR>
      <TD>Comments:</TD>
      <TD>
   <TEXTAREA NAME="comments" COLS="50" ROWS="4">
   </TEXTAREA>
      </TD>
   </TR>
   <TR>
      <TD>Action:</TD>
      <TD>
      <SELECT NAME="eventID">
         <OPTION VALUE="COM">Comment
         <OPTION VALUE="RPS">Route to product support
         <OPTION VALUE="RPD">Route to development
         <OPTION VALUE="RQA">Route to test
         <OPTION VALUE="CNB">Closed - not a bug
         <OPTION VALUE="CCP">Closed - customer problem
         <OPTION VALUE="CFX">Closed - fixed
      </SELECT>
      <INPUT TYPE="SUBMIT" VALUE="Submit">
      </TD>
   </TR>
   </TABLE>
   </FORM>

   <%
      // Get log entries for this problem

      model.problemLogSearch(problem.getProblemID());
      List problemLogs = model.getProblemLogs();
      if (problemLogs.size() > 0) {
   %>
   <H4>Problem History</H4><P>
   <TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0">
   <TR>
      <TH>Time</TH>
      <TH>Event Code</TH>
```

精彩好书见www.book118.com


```
        <TH>Comments</TH>
    </TR>
    <%
            Iterator it = problemLogs.iterator();
            while (it.hasNext()) {
                ProblemLog log = (ProblemLog) it.next();
    %>
    <TR>
        <TD><%= Util.dateTimeFormat(log.getLogTime()) %></TD>
        <TD><%= log.getEventID() %></TD>
        <TD><%= log.getComments() %></TD>
    </TR>
    <%
            }
    %>
    </TABLE>
    <%
        }
    %>

    </BODY>
    </HTML>
```

同样，该JSP页面基本由HTML组成，带有一些访问Problem对象、其顾客ID、产品ID、描述和严重程度的Java。记录问题人口列表来自模式的problemLogSearch（ ）方法。

Confirm.jsp 代理确认问题记录后，产生一个确认页面（图19-8），列出针对问题的ID、描述、严重程度、顾客注释和问题路由。代理在电话里向顾客给出问题ID，通知她ScoreWriter产品支持人员将会向其回电。确认源码如下：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Confirmation</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Confirmation</H3>
<TABLE BORDER=0 CELLPADDING=3 CELLSPACING=0>
<TR>
```

精彩好书见www.book118.com


```
    <TD>Problem ID:</TD>
    <TD><%= request.getParameter("problemID") %></TD>
</TR>
<TR>
    <TD>Description:</TD>
    <TD><%= request.getParameter("description") %></TD>
</TR>
<TR>
    <TD>Severity:</TD>
    <TD><%= request.getParameter("severity") %></TD>
</TR>
<TR>
    <TD>Comments:</TD>
    <TD><%= request.getParameter("comments") %></TD>
</TR>
<TR>
    <TD>EventID:</TD>
    <TD><%= request.getParameter("eventID") %></TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

注意　系统加入有用的功能是问题被路由到的雇员发出的电子邮件通告。在第21章介绍其实现方式。

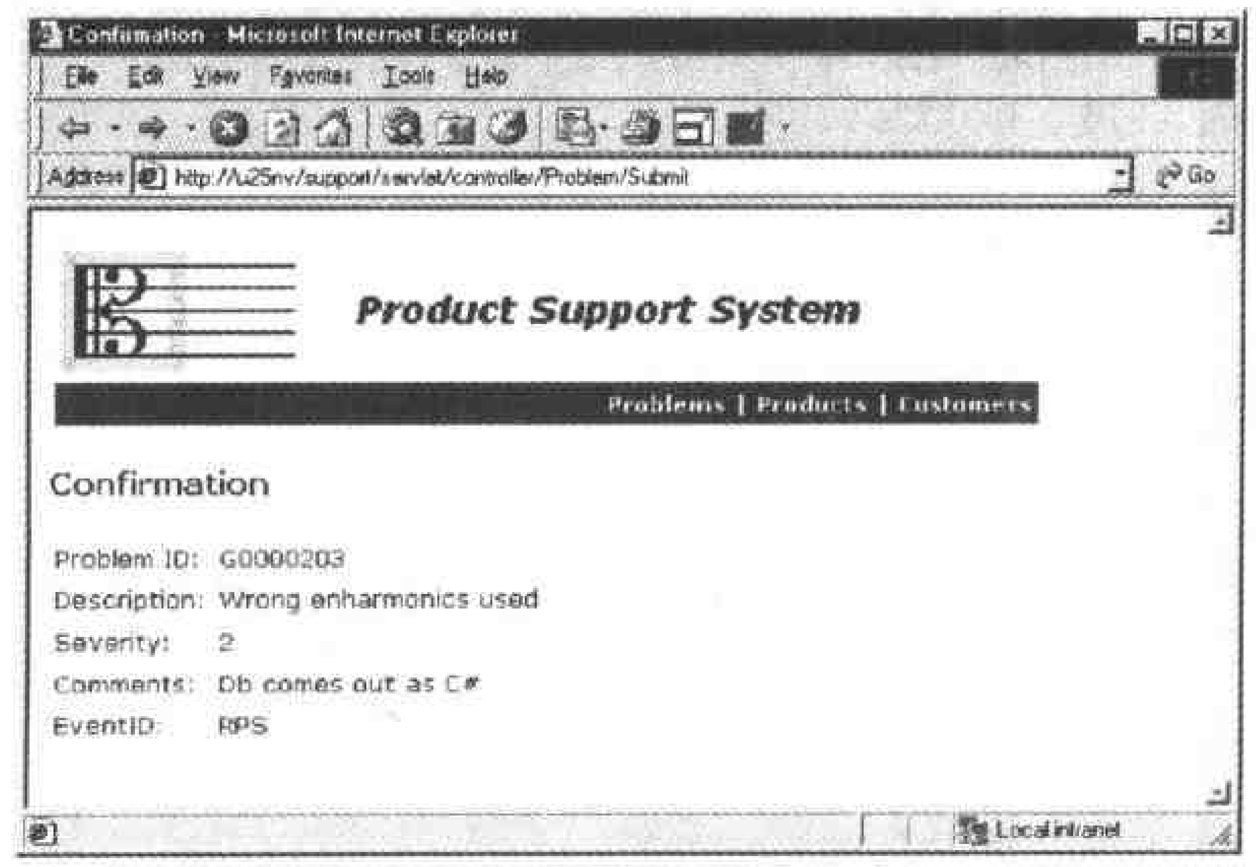


图19-8　确认页面

Products.jsp　其他用户，如产品支持人员、开发商或测试者可能通过查找一特定产品启动应

精彩好书见www.book118.com

用。像顾客搜索页面一样，也有一个产品搜索JSP视图页面，如下：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Product Search</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Product Search</H3>
<FORM
   METHOD="POST"
   ACTION="<%= CONTROLLER %>/Products/Search">
<B>Product name</B>:
<INPUT TYPE="TEXT" NAME="productSearchArgument" SIZE="20">
<INPUT TYPE="SUBMIT" VALUE="Search">
</FORM>

</BODY>
</HTML>
```

像顾客搜索页面一样，Products.jsp使用一个HTML窗体提示输入一个搜索字符串。图19-9解释了名字包含字母S的产品的搜索。

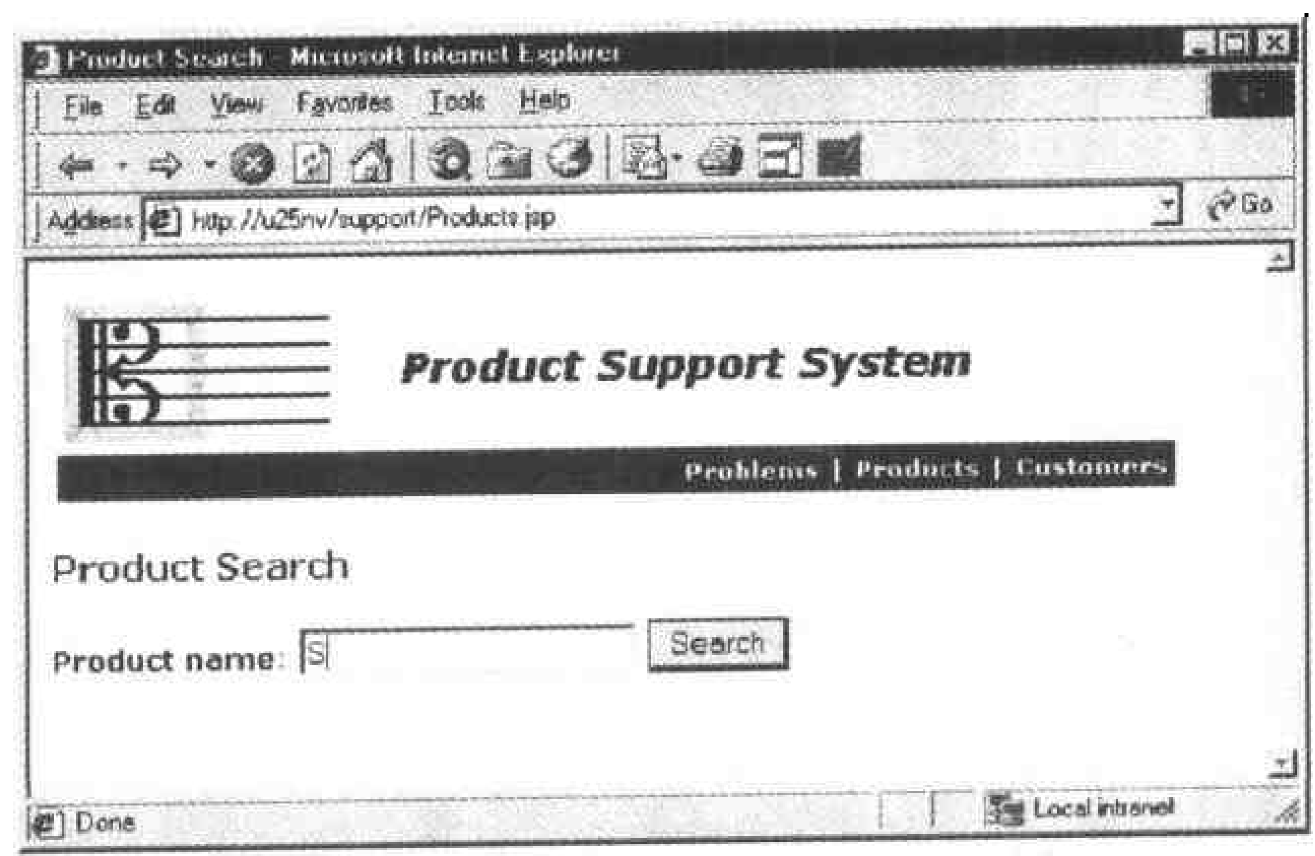


图19-9 产品搜索页面

精彩好书见www.book118.com

PoductsList.jsp 如图19-10所示，两个产品名包含字母S：Music Teacher Studio和ScoreWriter。PoductsList.jsp页面显示了两个产品的ID、产品名和设置的支持人员名字。PoductsList.jsp如下：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="java.util.*" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Products List</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Products List</H3>
<TABLE BORDER=0 CELLSPACING=5 CELLPADDING=0>
   <TR>
      <TH ALIGN=LEFT>Product ID</TH>
      <TH ALIGN=LEFT>Product Name</TH>
      <TH ALIGN=LEFT>Support</TH>
      <TH ALIGN=LEFT>Developer</TH>
      <TH ALIGN=LEFT>Tester</TH>
   </TR>

<%
   List list = model.getProducts();
   if (list != null) {
      Iterator it = list.iterator();
      while (it.hasNext()) {
         Product product = (Product) it.next();

         // Get the product select URL

         String productID = product.getProductID();
         String selectURL = CONTROLLER +
            "/ProductsList/Select?productID="
            + productID;

         String productName = product.getName();

         // Get the names of the product support,
         // developer, and tester employees

         String productSupport = product.getProductSupport();
```

精彩好书见www.book118.com


```
            String productSupportName =
                model.getEmployee(productSupport).getName();

            String developer = product.getDeveloper();
            String developerName =
                model.getEmployee(developer).getName();

            String tester = product.getTester();
            String testerName =
                model.getEmployee(tester).getName();
%>
    <TR>
        <TD><A HREF="<%= selectURL %>"><%= productID %></A></TD>
        <TD><%= productName %></TD>
        <TD><%= productSupportName %></TD>
        <TD><%= developerName %></TD>
        <TD><%= testerName %></TD>
    </TR>
<%
        }
    }
%>
</TABLE>

</BODY>
</HTML>
```

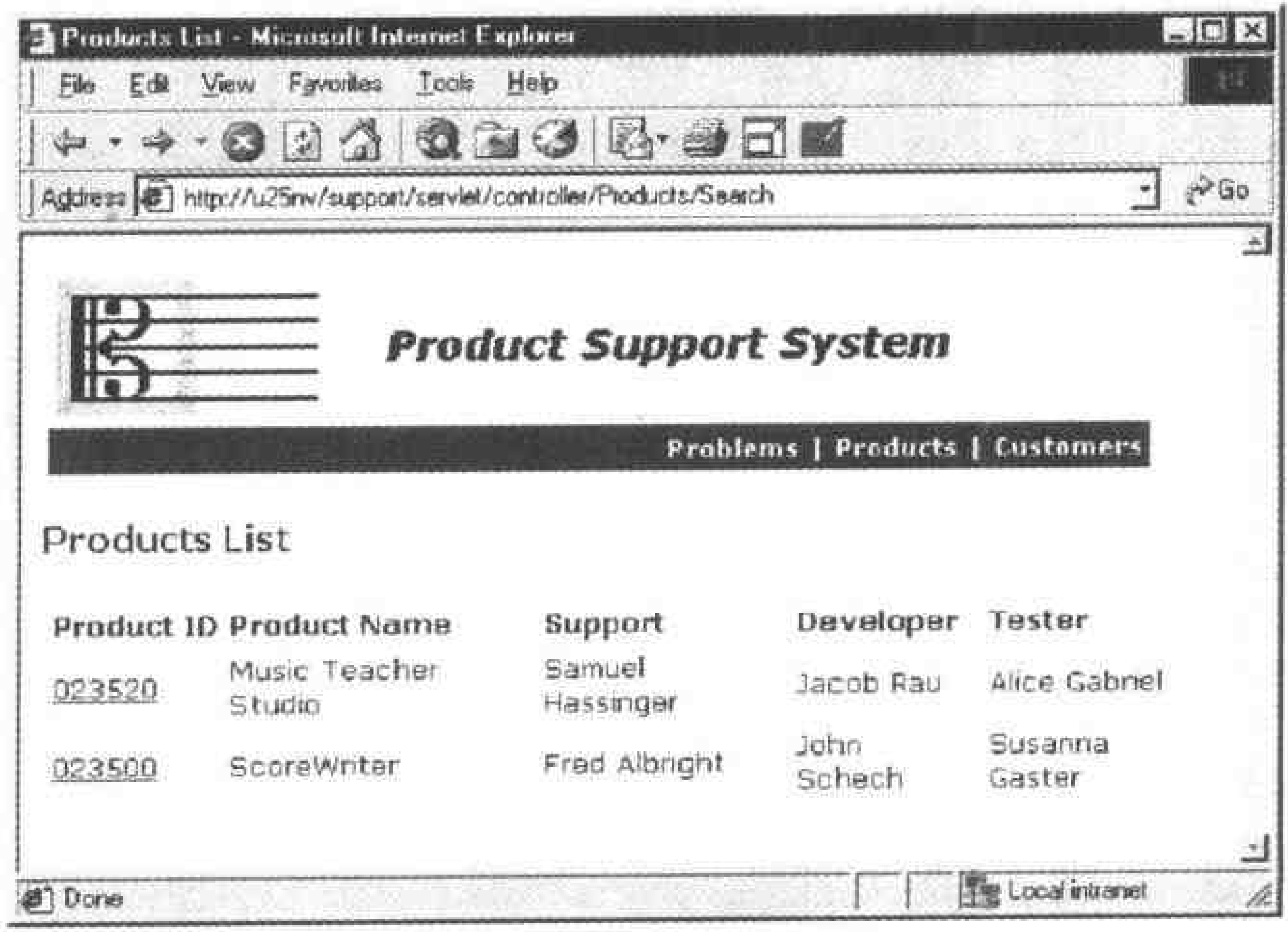


图19-10 产品列表页面

精彩好书见www.book118.com

对Product对象中三个雇员ID中的每一个，JSP都显示出相应的雇员名。它从模式的getEmployee（）方法中得到Employee对象，然后调用Employee.getName（）方法得到名字。

ProductProblems.jsp 如果选择了ScoreWriter链接，则显示此产品的问题列表。如图19-11所示。

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="java.util.*" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>

<HTML>
<HEAD>
<TITLE>Problems by Product</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Problems by Product</H3>
<%
   Product product = model.getProduct();
   String productID = product.getProductID();
   String productName = product.getName();
%>
<B>Product:<B> <%= productID %> - <%= productName %>
<TABLE BORDER=0 CELLSPACING=5 CELLPADDING=0>
   <TR>
      <TH ALIGN=LEFT>Problem ID</TH>
      <TH ALIGN=LEFT>Description</TH>
      <TH ALIGN=LEFT>Date Reported</TH>
      <TH ALIGN=LEFT>Date Resolved</TH>
   </TR>
<%
   List list = model.getProblems();
   if (list != null) {
      Iterator it = list.iterator();
      while (it.hasNext()) {
         Problem problem = (Problem) it.next();

         // Create the problem select URL

         String problemID = problem.getProblemID();
         String selectURL = CONTROLLER +
            "/Problems/Select?problemID="
```

精彩好书见www.book118.com


```
            + problemID;

        String problemDescription = problem.getDescription();

        // Get the reported and resolution dates

        String dateReported =
           Util.dateFormat(problem.getDateReported());
        String dateResolved =
           Util.dateFormat(problem.getDateResolved());
%>
   <TR>
      <TD><A HREF="<%= selectURL %>"><%= problemID %></A></TD>
      <TD><%= problemDescription %></TD>
      <TD><%= dateReported %></TD>
      <TD><%= dateResolved %></TD>
   </TR>
<%
      }
   }
%>
</TABLE>

</BODY>
</HTML>
```

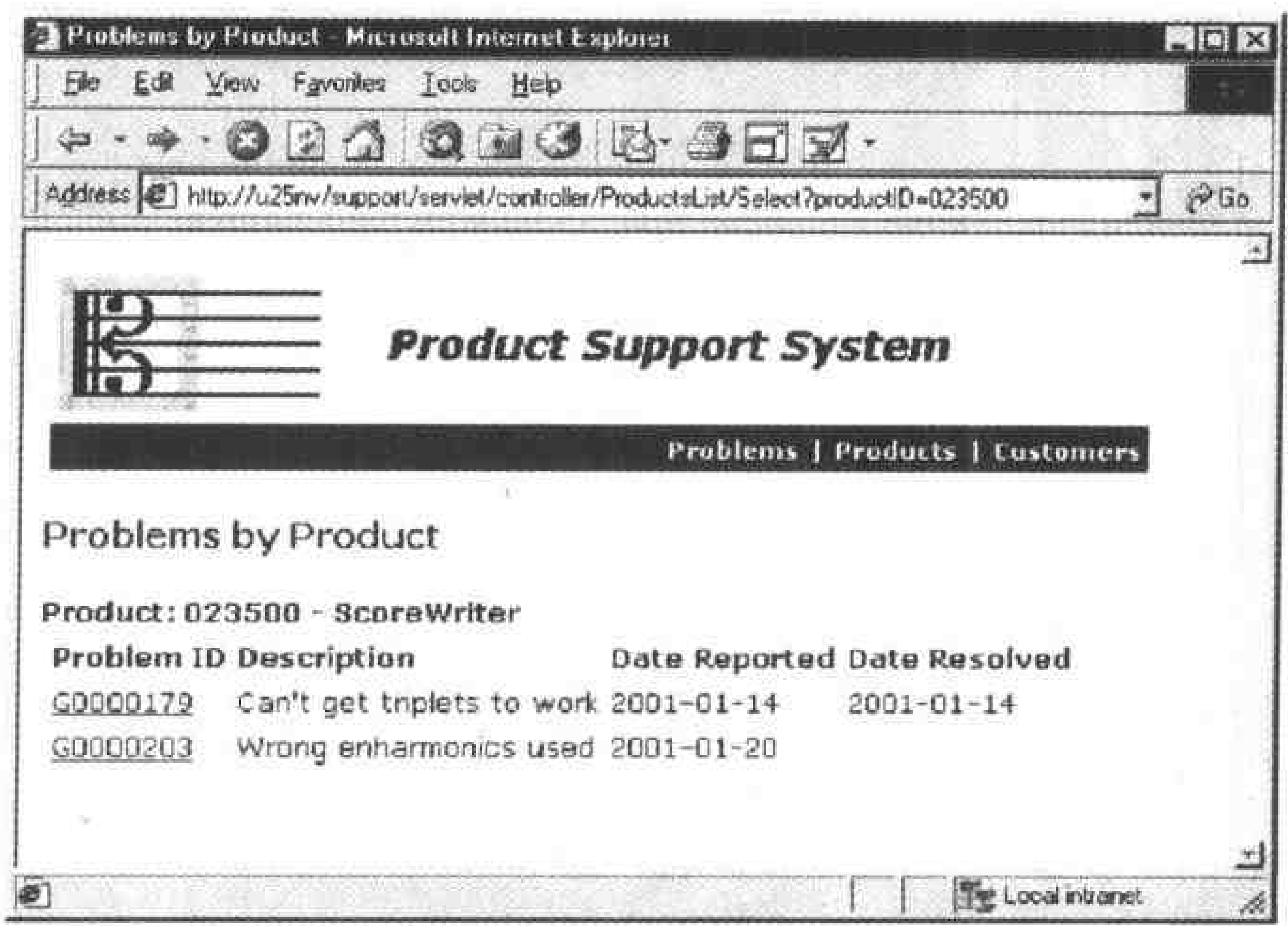


图19-11 产品问题页面

精彩好书见www.book118.com

当ProductProblems.jsp列表显示时，可以从模式中获得当前的Product对象以及此产品问题列表。

当ScoreWriter的产品支持人员Fred Albright打电话给顾客询问问题的细节时，他通过点击超级链接从列表中选择问题号码，然后看到该问题的一个已修改版本（图19-12），他在注释段输入与顾客洽谈的结果并确认问题修改。新的确认如图19-13所示：

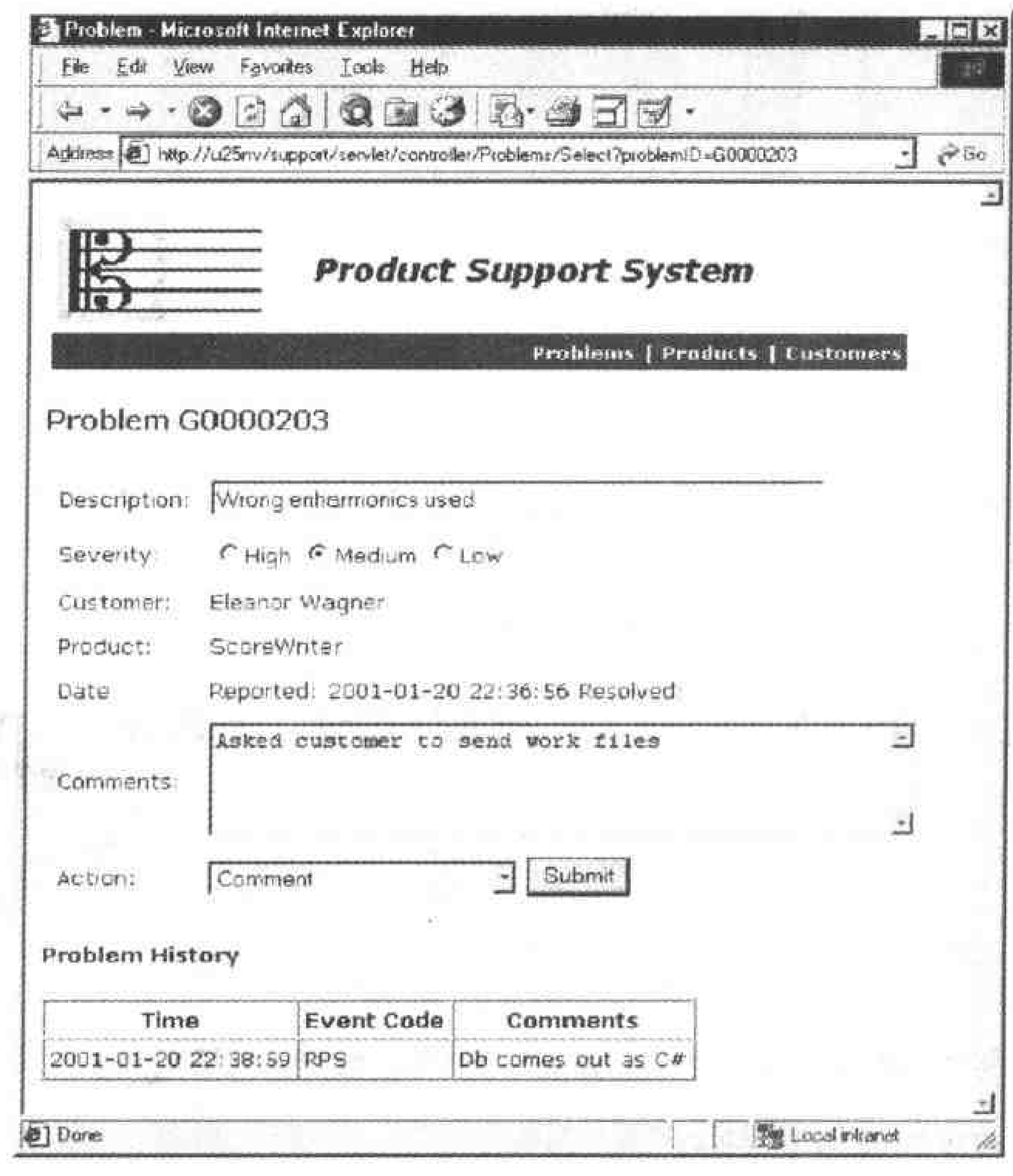


图19-12　已修改问题细节页面

Problems.jsp　如果问题ID已知，用户可以使用图19-4显示的JSP视图直接选择问题。Problems.jsp源码类似于顾客和产品搜索页面：

```
<%@ page session="true" %>
<%@ page errorPage="/ErrorPage.jsp" %>
<%@ page import="com.lyricnote.support.model.*" %>

<%@ include file="/WEB-INF/InitModel.jsp" %>
```

精彩好书见www.book118.com


```
<HTML>
<HEAD>
<TITLE>Problem Selection</TITLE>
<LINK REL="stylesheet" HREF="<%= BASEURL %>/style.css">
</HEAD>
<BODY>

<%@ include file="/WEB-INF/Banner.jsp" %>

<H3>Problem Selection</H3>
<FORM
   METHOD="POST"
   ACTION="<%= CONTROLLER %>/Problems/Select">
<B>Problem ID</B>:
<INPUT TYPE="TEXT" NAME="problemID" SIZE="8">
<INPUT TYPE="SUBMIT" VALUE="select">
</FORM>

</BODY>
</HTML>
```

对问题进一步深入，Fred Albright判断出它是一个用户输入错误。他对此问题输入注释信息（图19-15），并结束问题。确认屏幕如图19-16所示。

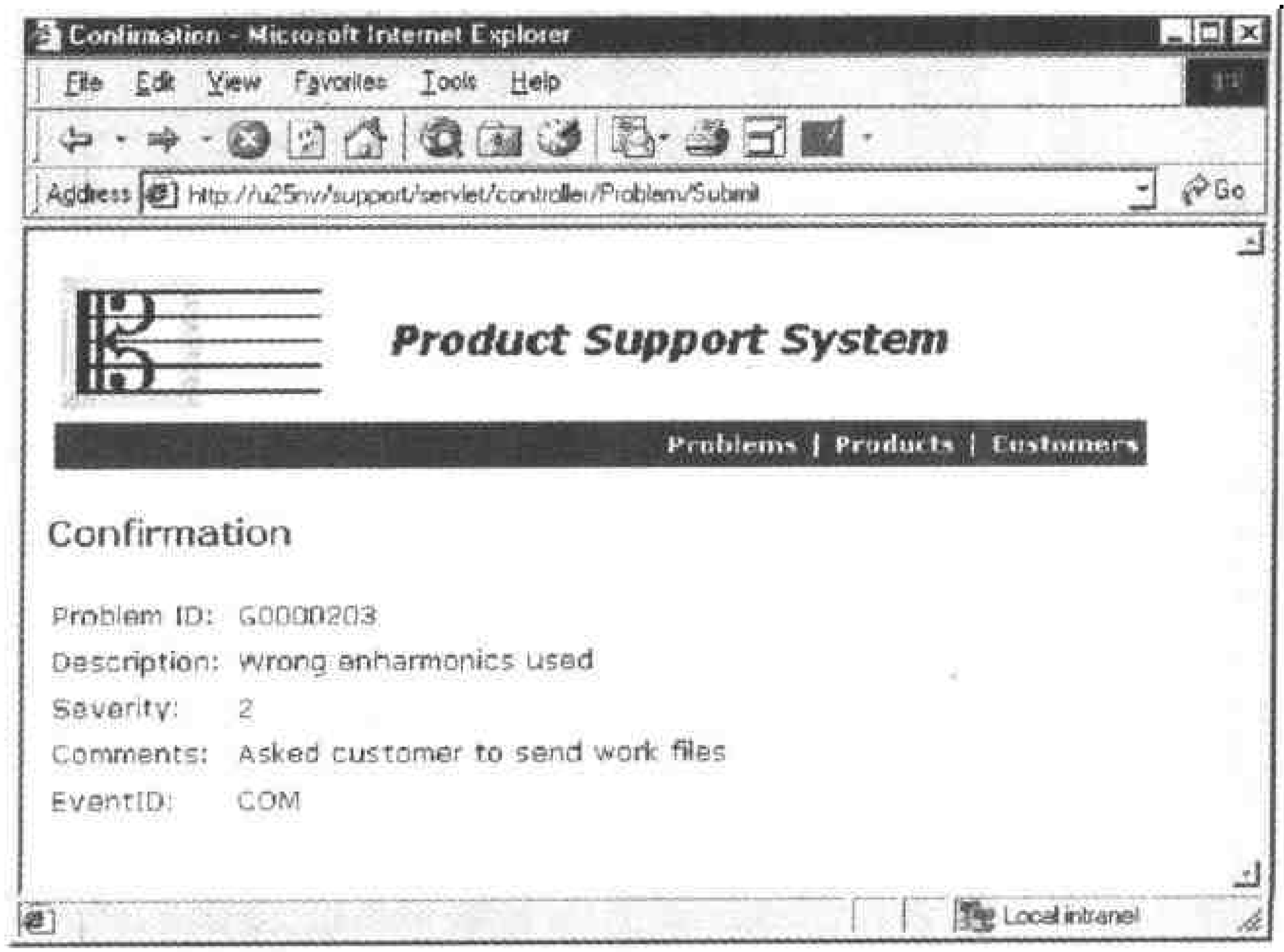


图19-13 新的确认页面

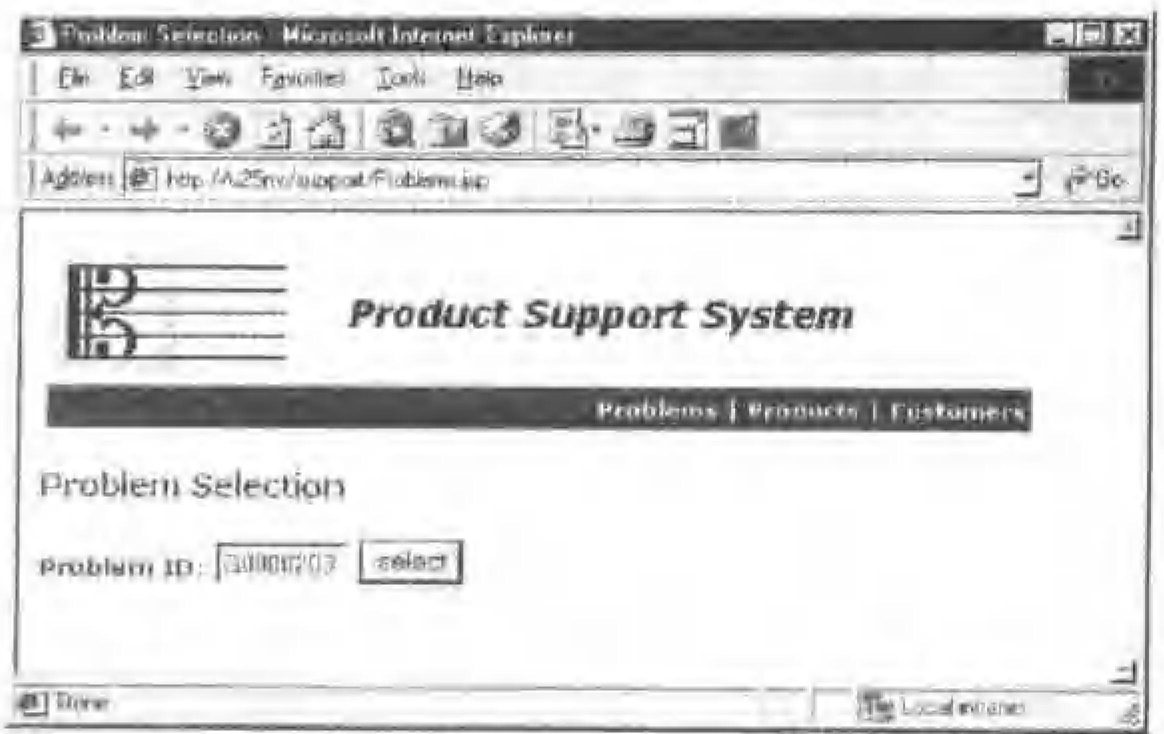


图19-14　问题选择页面

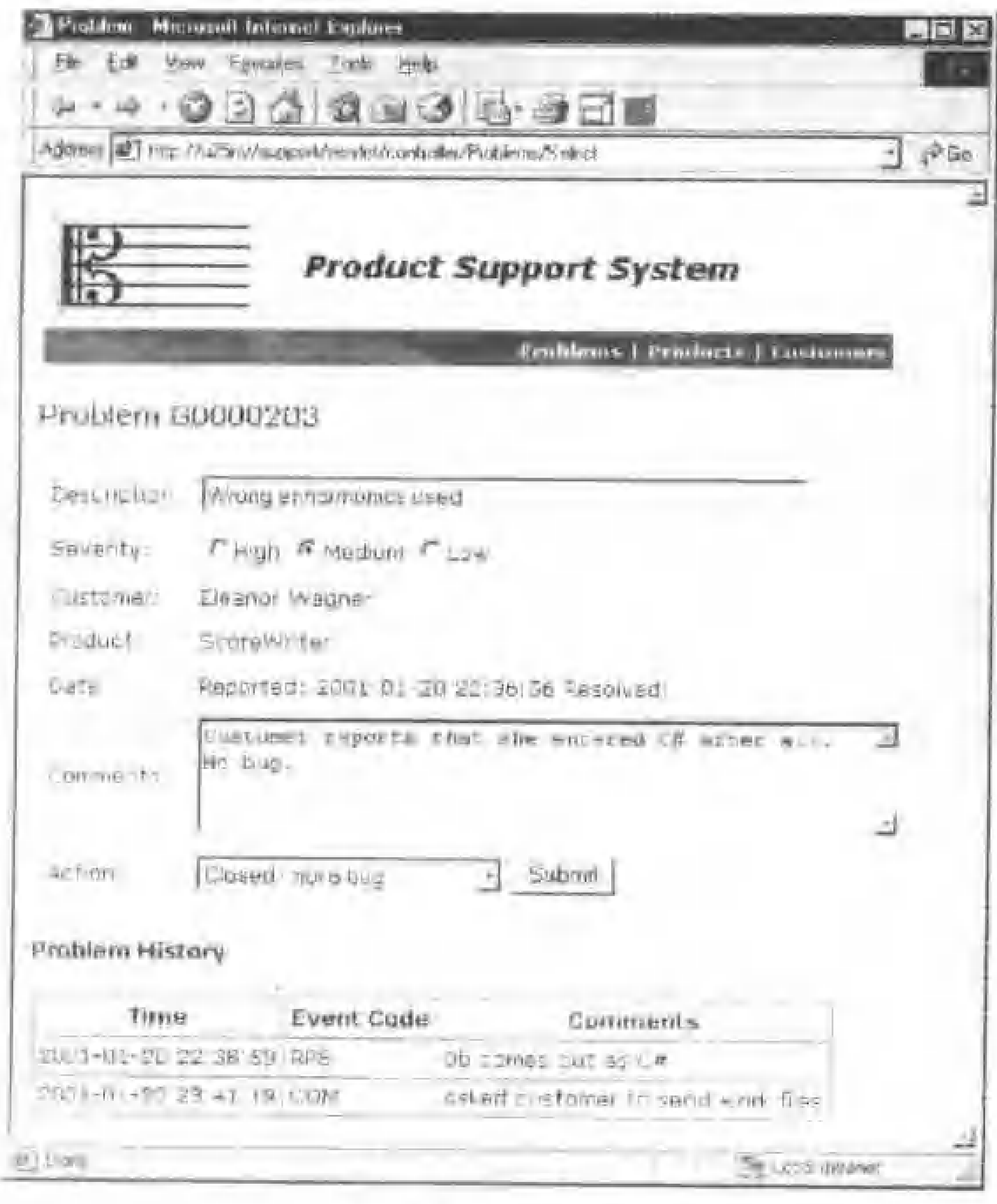


图19-15　最终问题修改页面

精彩好书见www.book118.com

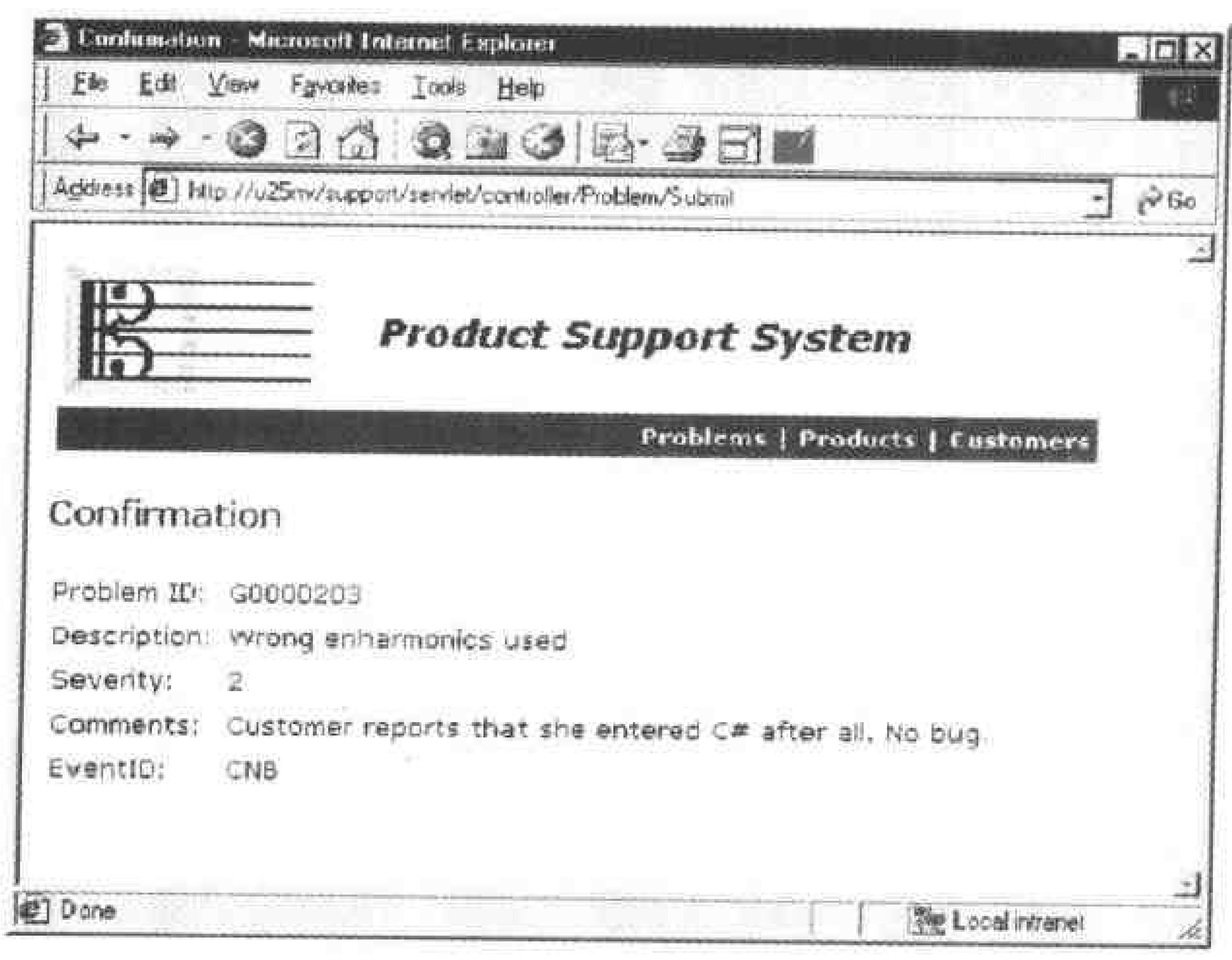


图19-16 最终确认页面

19.4.3 控制器类

开发的最后一个组件是控制器，即依据用户输入对模式进行操作并选择下一视图的系统部分。在产品支持系统中，此功能由一单独的servlet执行，名为ControllerServlet。

控制器功能可通过使用小的、定制的行为类以处理每一状态转换，不是在servlet中硬编码的每一行实现，这样就可以一次构建一个模块。下面讨论控制器授权给行为类使用的机制。

每次调用控制器servlet，它都需要知道两件事情：

- 当前视图是什么？
- 用户从此视图选择的行为是什么？

可能已经注意到，所有的JSP视图都调用在URL中带有附加路径信息的控制器servlet。此路径信息包含当前视图的名字和描述用户选择行为的关键字。例如，在Customer.jsp页面中，当用户输入一个搜索字符串，点击搜索按钮，窗体被确认，执行带有路径信息/Customers/Search的控制器servlet。一些视图页面可以有多个可能的用户行为。在Customer.jsp页面中，用户可以点击产品名报告新问题或选择要修改的已存在问题。在任何事件中控制器都接受视图名和行为关键字，并将它们结合在一起，并扩展关键字Action。结果就生成了可以处理状态转换类的名字。

1. 行为基类

下面是抽象Action类列表：

```
package com.lyricnote.support.controller;

import com.lyricnote.support.model.*;
```

精彩好书见www.book118.com


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

/**
* The base class for all state transitions
*/
public abstract class Action
{
   protected HttpServletRequest request;
   protected HttpServletResponse response;
   protected ServletContext application;
   protected Model model;

   /**
   * Executes the action. Subclasses should override
   * this method and have it forward the request to the
   * next view component when it completes processing.
   */
   public abstract void run()
      throws ServletException, IOException;

   /**
   * Sets the request.
   * @param request the request.
   */
   public void setRequest(HttpServletRequest request)
   {
      this.request = request;
   }

   /**
   * Sets the response
   * @param response the response
   */
   public void setResponse(HttpServletResponse response)
   {
      this.response = response;
   }

   /**
   * Sets the servlet context.
   * @param application the application.
   */
   public void setApplication(ServletContext application)
   {
```

精彩好书见www.book118.com


```
        this.application = application;
    }

    /**
    * Sets the model.
    * @param model the model.
    */
    public void setModel(Model model)
    {
        this.model = model;
    }
}
```

Action包含servlet请求和响应的实例变量、servlet上下文和模式本身。除了这些变量getter和setter方法，还有一个抽象方法run（）。此方法是惟一必须被每个行为控制器实现的方法。run（）方法调用模式方法实现转换，然后创建一个请求发送器并发送请求到下一视图。

2. 控制器servlet

控制器servlet是所有状态转换的驱动器。它维护已被调用的行为类的每个会话中的缓存。当进行一个请求时，servlet检查会话行为映射，查看该类的实例变量是否已被载入。如果未载入，它从路径信息中抽取视图名和行为关键字，并结合它们并对结果附加Action以得到行为类名。然后它载入类并创建一个实例，将其保存在会话行为映射中。确保一个模式已经存在后，servlet在行为对象中设置请求、响应、应用和模式的属性并调用其run（）方法。

ControllerServlet如下：

```
package com.lyricnote.support.controller;

import com.lyricnote.support.model.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
* The controller component of the Model-View-Controller
* architecture for the LyricNote problem reporting system
*/
public class ControllerServlet extends HttpServlet
{
    /**
    * Handles an HTTP GET request
    */
    public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
```

精彩好书见www.book118.com


```
    throws ServletException, IOException
{
    doPost(request, response);
}

/**
* Handles an HTTP POST request
*/
public void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, IOException
{
    HttpSession session = request.getSession();
    Map actionMap = (Map) session.getAttribute("actionMap");
    if (actionMap == null) {
        actionMap = new HashMap();
        session.setAttribute("actionMap", actionMap);
    }
    ServletContext context = getServletContext();
    try {

        // Get the state and event from the path info

        String pathInfo = request.getPathInfo();
        if (pathInfo == null)
            throw new ServletException
            ("Invalid internal state - no path info");

        // Load the action object that handles
        // this state and event

        Action action = (Action) actionMap.get(pathInfo);
        if (action == null) {

            // This is the first time the servlet has seen
            // this action. Get the state and event name
            // from pathInfo.

            StringTokenizer st =
                new StringTokenizer(pathInfo, "/");

            if (st.countTokens() != 2)
                throw new ServletException
                ("Invalid internal state - invalid path info ["
                + pathInfo + "]");
```

精彩好书见www.book118.com


```
            String state = st.nextToken();
            String event = st.nextToken();

            // Form the class name from the state and event

            String className =
                "com.lyricnote.support.controller."
                + state + event + "Action";

            // Load the class and create an instance

            try {
                Class actionClass = Class.forName(className);
                action = (Action) actionClass.newInstance();
            }
            catch (ClassNotFoundException e) {
                throw new ServletException
                ("Could not load class " + className
                + ": " + e.getMessage());
            }
            catch (InstantiationException e) {
                throw new ServletException
                ("Could not create an instance of "
                + className + ": " + e.getMessage());
            }
            catch (IllegalAccessException e) {
                throw new ServletException
                (className + ": " + e.getMessage());
            }

            // Cache the instance in the action map

            actionMap.put(pathInfo, action);
        }

        // Ensure that a model exists in the session.

        Model model = (Model) session.getAttribute("model");
        if (model == null)
            throw new ServletException
            ("No model found in session");

        // Now execute the action. The action should perform
        // a RequestDispatcher.forward() when it completes
```

精彩好书见www.book118.com


```
            action.setRequest(request);
            action.setResponse(response);
            action.setApplication(context);
            action.setModel(model);
            action.run();
        }
        catch (ServletException e) {

            // Use the JSP error page for all servlet errors

            request.setAttribute("javax.servlet.jsp.jspException", e);
            RequestDispatcher rd =
               context.getRequestDispatcher("/ErrorPage.jsp");

            if (response.isCommitted())
               rd.include(request, response);
            else
               rd.forward(request, response);
        }
    }
}
```

接下来的一节介绍在产品支持系统中使用的每个行为类。

3. 行为类

图19-3显示了从一个视图到另一个视图发生的7个状态转换过程。这些转换对应下列行为类：

- CustomersSearchAction
- CustomersListSelectAction
- CustomersNewProblemAction
- ProductsSearchAction
- ProductsListSelectAction
- ProblemsSelectAction
- ProblemSubmitAction

CustomerSearchAction类 调用此类中的run ( ) 方法获得Customer.jsp中的顾客搜索参数并调用模式中的customersearch ( ) 方法，然后它发送请求到显示搜索结果的JSP视图页面。CustomerSearchAction如下所示：

```
package com.lyricnote.support.controller;

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

/**
* Searches the database for customers matching the
```

精彩好书见www.book118.com


```
 * customer search argument
 */
public class CustomersSearchAction extends Action
{

   /**
    * Executes the action
    */
   public void run() throws ServletException, IOException
   {
      // Perform search

      String arg = request.getParameter("customerSearchArgument");
      if (arg != null) {
         arg = arg.trim();
         if (!arg.equals("")) {
            try {
               model.customerSearch(arg);
            }
            catch (SQLException e) {
               throw new ServletException(e.getMessage());
            }
         }
      }

      // Forward to customer list JSP

      final String next = "/CustomersList.jsp";
      RequestDispatcher rd =
         application.getRequestDispatcher(next);
      if (rd == null)
         throw new ServletException
         ("Could not find " + next);
      rd.forward(request, response);
   }
}
```

CustomerListSelectAction类 用户通过点击携带CustomerList视图名字和Select行为关键字的控制器中的超级链接选择列表中的一个顾客。另外，超级链接URL具有附加为一查询字符串的顾客ID。从此信息中，CustomerListSelectAction类执行下述操作：

- 从URL中抽取顾客ID参数并保存在模式中。
- 调用模式的顾客问题搜索方法。
- 发送请求到顾客细节JSP视图。

该行为类代码如下：

```
package com.lyricnote.support.controller;
```

精彩好书见www.book118.com


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

/**
* Gets detailed information for this customer
*/
public class CustomersListSelectAction extends Action
{
   /**
   * Executes the action
   */
   public void run() throws ServletException, IOException
   {
      // Get customer ID and store it in the model

      String customerID = request.getParameter("customerID");
      if (customerID == null)
         throw new ServletException
         ("No customer ID specified");
      model.setCustomerID(customerID);

      // Get the list of problems for this customer

      try {
         model.customerProblemsSearch(customerID);
      }
      catch (SQLException e) {
         throw new ServletException(e.getMessage());
      }

      // Forward to customer detail JSP

      final String next = "/Customer.jsp";
      RequestDispatcher rd =
         application.getRequestDispatcher(next);
      if (rd == null)
         throw new ServletException
         ("Could not find " + next);
      rd.forward(request, response);
   }
}
```

CustomerNewProblemAction类　如前面提到的，Customer视图有两种可能行为：创建一个新

精彩好书见www.book118.com

问题或修改已存在问题。新问题行为由下列行为类处理：

```
package com.lyricnote.support.controller;

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

public class CustomerNewProblemAction extends Action
{
   /**
   * Executes the action
   */
   public void run() throws ServletException, IOException
   {

      // Get the customer ID and product ID

      String customerID = request.getParameter("customerID");
      if (customerID == null)
         throw new ServletException
         ("No customer ID");

      String productID = request.getParameter("productID");
      if (productID == null)
         throw new ServletException
         ("No product ID");

      // Create a new problem

      try {
         model.setCustomerID(customerID);
         model.setProductID(productID);
         model.newProblem();
      }
      catch (SQLException e) {
         throw new ServletException(e.getMessage());
      }

      // Forward to problem detail JSP

      final String next = "/Problem.jsp";
      RequestDispatcher rd =
         application.getRequestDispatcher(next);
      if (rd == null)
```

精彩好书见www.book118.com


```
            throw new ServletException
            ("Could not find " + next);
        rd.forward(request, response);
    }
}
```

此行为类执行下述操作:

- 从视图生成的请求中检索顾客ID和产品ID参数。
- 创建并初始化一个新Problem对象，实现方式通过调用模式中的newProblem（ ）工厂方法，该方法设置惟一的问题ID并在数据库中写入初始记录。
- 发送请求到问题细节视图。

ProductsSearchAction类 像顾客搜索行为一样，产品搜索行为从请求中接受一个搜索字符串并调用模式中的一个搜索方法，然后发送请求到ProductsList.jsp视图。

```
package com.lyricnote.support.controller;

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

/**
* Searches the database for products matching the
* product search argument
*/
public class ProductsSearchAction extends Action
{
    /**
    * Executes the action
    */
    public void run() throws ServletException, IOException
    {

        // Perform search

        String arg = request.getParameter("productSearchArgument");
        if (arg != null) {
            arg = arg.trim();
            if (!arg.equals("")) {
                try {
                    model.productSearch(arg);
                }
                catch (SQLException e) {
                    throw new ServletException(e.getMessage());
                }
```

精彩好书见www.book118.com


```
            }
        }

        // Forward to product list JSP

        final String next = "/ProductsList.jsp";
        RequestDispatcher rd =
            application.getRequestDispatcher(next);
        if (rd == null)
            throw new ServletException
            ("Could not find " + next);
        rd.forward(request, response);
    }
}
```

ProductsListSelectAction类　当选中一个产品ID时，此行为类将其保存在模式中并调用模式的产品问题搜索。然后请求被发送到ProductProblems.jsp视图，显示结果。

```
package com.lyricnote.support.controller;

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

/**
* Searches the database for products matching the
* product search argument
*/
public class ProductsListSelectAction extends Action
{
    /**
    * Executes the action
    */
    public void run() throws ServletException, IOException
    {
        // Get product ID and store it in the model

        String productID = request.getParameter("productID");
        if (productID == null)
            throw new ServletException
            ("No product ID specified");
        model.setProductID(productID);

        // Get the list of problems for this product

        try {
```

精彩好书见www.book118.com


```
            model.productProblemsSearch(productID);
        }
        catch (SQLException e) {
            throw new ServletException(e.getMessage());
        }

        // Forward to product problems JSP

        final String next = "/ProductProblems.jsp";
        RequestDispatcher rd =
            application.getRequestDispatcher(next);
        if (rd == null)
            throw new ServletException
            ("Could not find " + next);
        rd.forward(request, response);
    }
}
```

ProblemsSelectAction.jsp类 3个应用入口点——顾客、产品和问题——都使用一个通用的问题选择行为，如下所示：

```
package com.lyricnote.support.controller;

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

/**
* Sets the current problem ID
*/
public class ProblemsSelectAction extends Action
{
    /**
    * Executes the action
    */
    public void run() throws ServletException, IOException
    {
        String problemID = request.getParameter("problemID");
        if (problemID != null) {
            problemID = problemID.trim();
            if (!problemID.equals("")) {
                model.setProblemID(problemID);
            }
        }

        // Forward to problem JSP
```

精彩好书见www.book118.com


```
        final String next = "/Problem.jsp";
        RequestDispatcher rd =
           application.getRequestDispatcher(next);
        if (rd == null)
           throw new ServletException
           ("Could not find " + next);
        rd.forward(request, response);
    }
}
```

此行为类简单将问题ID保存在模式中并发送请求到问题细节页面。

ProblemSubmitAction类　所需的最后一个行为类是从问题细节页面接受问题修改的一个行为类。此类任务为：

- 从请求中检索数据条目域。
- 从模式中检索当前Problem对象并修改其属性。如果事件ID指出问题应结束，则调用Problem对象的close（）方法。
- 修改数据库中此问题记录。
- 在问题记录中加入一个条目。
- 发送请求到下一视图——确认。

```
package com.lyricnote.support.controller;

import com.lyricnote.support.model.*;
import java.io.*;
import java.sql.SQLException;
import javax.servlet.*;
import javax.servlet.http.*;

/**
* Submits a problem update
*/
public class ProblemSubmitAction extends Action
{
   /**
   * Executes the action
   */
   public void run() throws ServletException, IOException
   {
      // Get the parameters

      String problemID = request.getParameter("problemID");
      String description = request.getParameter("description");
      String severity = request.getParameter("severity");
      String comments = request.getParameter("comments");
```

精彩好书见www.book118.com


```
            String eventID = request.getParameter("eventID");

            try {

               // Get the problem object from the model

               model.setProblemID(problemID);
               Problem problem = model.getProblem();

               // Update the problem object

               problem.setDescription(description);
               problem.setSeverity(Integer.parseInt(severity));
               if (Util.isClosingEvent(eventID))
                  problem.close();
               model.updateProblem(problem);

               // Add a problem log record

               ProblemLog log = new ProblemLog();
               log.setProblemID(problemID);
               log.setLogTime(new java.util.Date());
               log.setEventID(eventID);
               log.setComments(comments);
               model.addProblemLog(log);
            }
            catch (SQLException e) {
               throw new ServletException(e.getMessage());
            }

            // Forward to confirmation JSP

            final String next = "/Confirm.jsp";
            RequestDispatcher rd =
               application.getRequestDispatcher(next);
            if (rd == null)
               throw new ServletException
               ("Could not find " + next);
            rd.forward(request, response);
         }
      }
```

19.5 小结

这一章将本书讨论过的元素结合到基于Web的系统的管理产品支持中心中。该系统支持下述过程流：

精彩好书见www.book118.com

- 有问题的顾客拨打一个免费号码，与电话中心代理通话。
- 代理检验顾客是否被授权对指定产品提供支持，如果授权，则输入问题报告向顾客给出确认号码并路由问题到产品支持中心。
- 产品支持中心打电话给顾客判断这是代码问题还是顾客问题。如果问题证明是代码问题，当前并没有可利用的解决方案，则问题被路由到开发商。
- 负责的开发商分析问题。如果问题不是一个故障，开发商再次路由问题到产品支持中心以通知顾客。如果问题是一个故障，开发商编码并单元测试一个方案，并将问题路由到质保中心。
- 质保中心执行集成测试。如果方案需要进一步改动，则问题被再次路由回开发商。否则，它被发送到产品支持中心，在这里，方案被发送到顾客，问题结束。

支持该系统所需的数据模式由表示顾客、产品、顾客/产品对、雇员、问题报告和问题记录条目的关系数据库表组成。

采纳的系统结构称为模式－视图－控制器（MVC）。由3个组件构成：

- 模式　应用的内部工作，包含数据库访问和事务逻辑。没有可视代码——可通过一个简单的命令行驱动器操作。
- 视图　从模式中检索数据并为用户的交互行为进行显示的表示层。
- 控制器　接受用户输入，对模式进行操作以改变其状态并显现下一视图的组件。

最终系统通过将代码分隔成可被单独测试的并使复杂性降为最低。这样就会产生一个健壮的、特性完全的容易扩展的应用。

精彩好书见www.book118.com

精彩好书见www.book118.com

第四部分　JSP和其他Web组件

JavaServer页面不只是Web应用的一个表示层；它们也可以充做其他应用的客户端或服务器。下面两章介绍JavaServer页面使用大的上下文——其如何与Java应用、applet、Perl脚本、邮件服务器和其他服务器端代理进行通信。

第20章　与其他客户端进行通信

Web浏览器是最常用的JSP客户端，但并不是惟一的客户端。随着越来越多的应用成为Web的应用，网络资源在各种类型的系统中均成为重要的组件。Internet可以充做发送源数据，不需指出其表示方式的通信链接。例如，最新的货币兑换率可用于批处理计算。交互式旅游预约可以是可执行信息系统的一部分。当前天气条件、新闻标题及股票价格可以嵌入到其他应用的小信息窗口中——甚至是在顾客的电子产品中。

可以使用HTTP协议的任何程序充做一个JSP客户端。使用java.net包中的基类，应用可以进行HTTP请求并读取经过，就好象它们是一个文件的内容一样。另外，文件内容是动态的，可通过请求参数加以控制。

在这一章，将讲述3类非浏览器JSP客户端的开发：

- Java应用
- Java applet
- Perl 脚本

每个例子都包括JSP服务器和可以利用该服务器的客户端。首先介绍一下基本技术。

20.1　URL连接

成为服务器的关键为一个输入流。Java类库提供3个实现此功能的类：

- `java.net.URL`
- `java.net.URLConnection`
- `java.net.HttpURLConnection`

20.1.1　URL类

惟一资源定位器（URL）是网络上可利用对象的惟一地址，也是必须用对象实施操作的协议指示[1]。这些协议包括ftp、http、gopher、mailto、news和其他在指定应用中使用的协议。

1 完整的URL规范为RFC 1738，可在http://www.freesoft.org/CIE/RFC/1738/index.htm中找到。

精彩好书见www.book118.com

本章考虑的URL使用超文本传输协议（HTTP）。

一个HTTP URL由5部分组成：

```
<scheme>://<host>[:<port>]/<path>[?<query string>]
```

模式对未加密传输为http，对使用诸如安全套接字层（SSL）加密技术的传输则为https。主机部分是网络主机的全质域名，可表示为按点划分的十进制IP地址。端口号可选，如果未指定缺省为80。路径是主机HTTP服务器中文档空间的地址。通常结构化成一个目录树。URL还可以包含在查询字符串中编码的请求参数。

Java.net.URL类是URL的一个面向对象的包容器。它提供从字符串中构建URL，并访问前面给出的每个部分的方法。另外，Java.net.URL类有两个允许访问URL指定资源的内容，并在某种情况下可以进行修改的重要方法。这些方法在表20-1中列出。

许多情况下，从一个远程网络资源中进行读取像下面一样简单：

```
URL url = new URL("http://servername/path/filename");
InputStream in = url.openStream();
int c;
while ((c = in.read()) != -1) {
    // ... do something with this byte
}
```

表20-1 Java.net.URL中一些可用方法

方法	描述
public URLConnection OpenConnection ( )	进行URL表示的远程对象连接
public final InputStream OpenStream ( )	打开一个URLConnection，为读取其内容创建一个InputStream。这是调用openConnection().getInputStream ( ) 的一种便捷方法

20.1.2 URLConnection类

实现此功能的底层类是java.net. URLConnection。它是抽象类，其子类表示了程序和远程网络资源之间的连接。使用一个URLConnection包括以下4步：

1) 调用URL的openConnection ( ) 方法创建URLConnection对象。

2) 对指定任务手工配置该连接。包括指出连接是否应用作输入、输出或同时用于两者，以及设置指出内容类型和其他头标的请求属性。

3) 进行连接，通常是隐含的（虽然connect ( ) 方法可用于此目的）。

4) 远程资源成为可利用的。连接可提供资源内容和使用它发送的任意响应头标。

最常用的一些方法在表20-2中列出。

表20-2 java.net. URLConnection中的可用方法

方法	描述
public void SetRequestProperty	设置一个通用的请求头标，如连接的一个HTTP头标。例子是key为Content-Length，并带有表示请求数据部分的字节数的整数值

精彩好书见www.book118.com

（续）

方　法	描　述
(String key , String value)	
public OutputStream getOutputStream()	返回将数据写入连接的输出流。如HTTP POST请求中的窗体参数
public InputStream getInputStream()	返回从连接中读取数据的输入流。调用此方法的程序基于某种性能考虑，典型情况将结果打包到一个被缓存的流中

20.1.3 HttpURLConnection类

URLConnection是一个抽象的，协议固有的类。所以它没有公有的构造器。它只能通过调用URL的openConnection（ ）方法创建。此调用返回的实际对象是一个处理URL协议的指定协议的子类。本章实例中，此子类为java.net. HttpURLConnection。

HttpURLConnection本身是抽象的。Java虚拟机厂家提供一个实际的实现类。通过调用openConnection()返回对象的getClass（ ）方法可以看出此类的内容：

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

public class ShowConnectionClass
{
   public static void main(String[] args)
      throws IOException
   {
      URL url = new URL("http://www.ibm.com");
      URLConnection con = url.openConnection();
      Class conClass = con.getClass();
      System.out.println
         ("Connection class is " + conClass.getName());
   }
}
```

使用Sun JVM，此程序输出为：

```
Connection class is sun.net.www.protocol.http.HttpURLConnection
```

一个HttpURLConnection除了提供URLConnection的功能外，还有如下3个其他功能：

- 指定请求方法的一种方式。
- 直接访问HTTP响应代码。
- 给出HTTP响应代码助记名的常量集。

表20-3给出了几个不错的方法。

表20-3　java.net. HttpURLConnection中的可用方法。

方　法	描　述
public int getResponseCode()	从响应的第一行抽取HTTP响应代码。例如，如果响应行为HTTP/1.0 404 Not found，此方法返回404

精彩好书见www.book118.com

（续）

方 法	描 述
Public String getResponseMessage()	返回响应代码后响应行的其余部分。例如，如果响应行为HTTP/1.0 404 Not found，此方法返回Not found
public static void setFollowRedirects (boolean set)	如果设置为真，使得301（永久移动）或302（暂时移动）响应码自动生成发送地址的另一请求
public void setRequestMethod (String method)	指定使用的HTTP请求方法：GET、POST、HEAD、OPTIONS、PUT、DELETE或TRACE

在下面一节，会看到如何通过URL连接查看JSP页面，就好象它是一个可编程的文件。

20.2 作为客户端的Java应用

本书设想的Internet音乐公司，其域名为LyricNote.com，和由一打折产品购买服务中心管理的竞价系统中的其他在线零售商结合在一起。顾客可以询问服务中心一种特定音乐设备的价格，然后服务中心向参与其供应商的Web站点查询最佳价格。每个供应商以服务中心规定的标准XML文档格式将竞价返回。

20.2.1 JSP竞价服务器

因为竞价请求每次都不一样，LyricNote.com不能只是简单地返回一个静态XML文档，而是使用JSP页面从数据库的查询经过中动态生成XML。JSP页面(PriceQuote.jsp)列表如下：

```
<%@ page
      session="false"
      import="java.sql.*,java.text.*"
      contentType="text/xml"
%><%
   // Define constants for JDBC driver name and
   // database URL

   String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
   String DB_URL = "jdbc:idb:"
      + "D:/lyricnote/WEB-INF/database/products/db.prp";

   // Get the product search argument and desired quantity

   String product = request.getParameter("product");
   if (product == null)
      throw new ServletException("No product specified");

   String qstring = request.getParameter("quantity");
   if (qstring == null)
```

精彩好书见www.book118.com


```
        throw new ServletException("No quantity specified");

    int quantity = 0;
    try {
        quantity = Integer.parseInt(qstring);
    }
    catch (NumberFormatException e) {
        throw new ServletException("Quantity not numeric");
    }

    // Load the driver

    Class.forName(DRIVER);

    // Create a connection

    Connection con = null;
    try {
        con = DriverManager.getConnection(DB_URL);

        // Create a select statement

        PreparedStatement pstmt = con.prepareStatement
        (
        " select itemcode, price, description"
        + " from products"
        + " where prodtype = 'IN'"
        + " and description like ?"
        + " and onhand >= ?"
        );

        // Supply values for substitution parameters

        pstmt.setString(1, "%" + product + "%");
        pstmt.setInt(2, quantity);

        // Execute the query

        ResultSet rs = pstmt.executeQuery();

        // Create the XML

%><?xml version="1.0"?>
<price-quote>
   <supplier>LyricNote.com</supplier>
   <date><%=
```

精彩好书见www.book118.com


```
        new SimpleDateFormat("yyyy-MM-dd")
        .format(new java.util.Date())
        %></date>
  <%

        while (rs.next()) {
           String itemCode    = rs.getString(1);
           double price       = rs.getDouble(2) / 100;
           String description = rs.getString(3);
  %> <item
        code="<%= itemCode %>"
        price="<%= new DecimalFormat("###.00").format(price) %>"
        description="<%= description %>"/>
  <%
        }
  %></price-quote><%
     }
     finally {
        if (con != null)
           con.close();
     }
  %>
```

JSP页面抽出产品搜索参数以及来自请求参数中所要求的质量，然后打开至LyricNote产品数据库连接，搜索匹配项，结果被写为XML。

奇数缩进模式（在%><%和%></price-quote>%<中带有紧接的分隔符）用来防止换行字符和其他无关的空格被写入输出流。本章后面的例子给出实现此功能的另一种技术。

20.2.2 竞价客户端应用

这里只考虑客户端Java应用的简单版本——阐述如何进行连接，而不是如何处理数据。

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

/**
* An example of a Java application that acts as a JSP client.
*/
public class PriceQuoteReader
{
   public static void main(String[] args)
   {
      // Define the supplier URL and the two search arguments.
      // These are hard-coded for the purposes of this example.

      String supplier =
```

精彩好书见www.book118.com


```
            "http://www.lyricnote.com/PriceQuote.jsp";
        String product = "Clarinet";
        int quantity = 3;

        // Append the search arguments to the URL so that
        // they will be recognized as parameters to an
        // HTTP GET request

        StringBuffer sb = new StringBuffer();
        sb.append(supplier);
        sb.append("?product=");
        sb.append(URLEncoder.encode(product));
        sb.append("&quantity=");
        sb.append(URLEncoder.encode(String.valueOf(quantity)));
        String supplierURL = sb.toString();

        try {

           // Now create the URL instance

           URL url = new URL(supplierURL);

           // and open its input stream

           InputStream stream = url.openStream();

           // Read each line of the XML that is returned.
           // All this example does it print the results;
           // the real application would do something more
           // useful

           BufferedReader in =
              new BufferedReader(
              new InputStreamReader(stream));

           while (true) {
              String line = in.readLine();
              if (line == null)
                 break;
              System.out.println(line);
           }
           in.close();
        }
        catch (IOException e) {
           e.printStackTrace();
        }
    }
}
```

精彩好书见www.book118.com

PriceQuoteReader对供应商URL、产品搜索字符串和请求的质量使用硬编码值。它创建一个包含搜索字符串和质量的请求参数的URL，然后调用URL.openStream（）初始化连接，然后读取和打印结果XML文档。如下：

```
<?xml version="1.0"?>
<price-quote>
   <supplier>LyricNote.com</supplier>
   <date>2001-01-25</date>
   <item
      code="001130"
      price="369.00"
      description="Wendecker B Flat Clarinet"/>
   <item
      code="001140"
      price="522.00"
      description="Clemens-Altman B Flat Clarinet - Wood"/>
   <item
      code="001150"
      price="417.00"
      description="Gabriel E Flat Clarinet - Wood"/>
   <item
      code="001160"
      price="548.00"
      description="Wendecker B Flat Bass Clarinet"/>
   <item
      code="001170"
      price="307.00"
      description="Clemens-Altman E Flat Clarinet"/>
</price-quote>
```

此XML包含供应商名字和请求时间，后跟每个匹配产品元素——这里黑管模式至少有三种产品可以购买。

20.3 一个Java Applet客户端

服务中心的主要应用具有实时搜索指定站点的功能。为此，它使用了一个Java applet。

先介绍一点背景，Applet是Java技术应用的第一次高峰，是一种在Web页面中嵌入小的交互式GUI应用的方式。在广泛应用之前，就得到了大量的关注。一部分问题是不同的浏览器提供不同的支持层次，并随着时间其差别增大。Sun微系统对此问题提出了一种可行性方案，称为Java插件。

20.3.1 Java插件

插件是一个管理其本身Java虚拟机的指定浏览器的嵌入式对象。这表明它并不依赖浏览器所提供的支持层次。插件的早期版本来自于将<APPLET>标签转换成其浏览器指定的副本

精彩好书见www.book118.com

<OBJECT>（对Microsoft Internet Explorer）和<EMBED>（对Netscape Navigator）的HTML转换器。

JSP引入了在一个Web页面中使用插件的简单方式，名为<jsp:plugin>行为。当在一个JSP页面中使用<jsp:plugin>时，在输出的HTML文档中依据浏览器它被<object>或<embed>标签替换。此标签的语法令人望而生畏，如下所示：

```
<jsp:plugin
      type="bean|applet"
      code="<classname>"
      codebase="<codebase>"
      align="<alignment>"
      archive="<archiveList>"
      height="<height>"
      hspace="hspace>"
      jreversion="<jreversion>"
      name="componentName>"
      vspace="vspace>"
      width="width>"
      nspluginurl="url>"
      iepluginurl="url>"
>
<jsp-params>
      <jsp:param name="name>" value="value>"/>
      ...
      <jsp:param name="name>" value="value>"
</jsp-params>
<jsp-fallback>Arbitrary text</jsp-fallback>
</jsp-plugin>
```

JSP规范描述了大部分属性，定义方式“像在HTML规范中定义的一样”。例外情况如下：

- type　可为bean或applet。
- jreversion　指出所需的Java运行时环境（JRE）的版本，缺省为“1.1”。
- nspluginurl　如果需要，下载插件的Netscape版本的URL。
- iepluginurl　如果需要，下载插件的Internet Explorer版本的URL。

<jsp-params>节指出使用的applet参数，<jsp-fallback>包含在不能启动applet的情况下显示的文本。

幸运的是，只需要其中的一小部分属性，type、code、codebase、height和width。

对applet使用插件的好处是它会带来一个稳定的、可预知的，可能支持Java 2的JVM。这表明applet可对其GUI使用Java基础类（Swing）。

20.3.2 PriceQuoteApplet

下面列表是询问竞价服务器的applet源码：

精彩好书见www.book118.com


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;

public class PriceQuoteApplet extends JApplet
{
   private JTextField txtProduct;
   private JTextField txtQuantity;
   private JButton btnPost;
   private JTextArea txtOutput;

   /**
   * Creates the GUI components
   */
   public void init()
   {
      Container content = getContentPane();
      content.setLayout(new BorderLayout());
      JPanel pnl;

      // Top row

      pnl = new JPanel();
      pnl.add(new JLabel("Product:"));
      pnl.add(txtProduct = new JTextField(12));
      pnl.add(new JLabel("Quantity:"));
      pnl.add(txtQuantity = new JTextField(4));
      pnl.add(btnPost = new JButton("POST"));
      content.add(pnl, BorderLayout.NORTH);

      // Results panel

      pnl = new JPanel();
      pnl.add(txtOutput = new JTextArea(12, 40));
      content.add(pnl, BorderLayout.CENTER);

      // Start listening for button clicks

      btnPost.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event)
         {
            try {
               doPost();
            }
```

精彩好书见www.book118.com


```
            catch (IOException e) {
               txtOutput.setText(e.getMessage());
            }
         }
      });
   }

   /**
   * Makes the request and writes the XML results
   * to the output text area
   */
   public void doPost() throws IOException
   {
      // Extract the parameters from the GUI

      String product = txtProduct.getText().trim();
      String quantity = txtQuantity.getText().trim();

      // Create POST data using the search arguments
      // as request parameters

      StringBuffer sb = new StringBuffer();
      sb.append("product=");
      sb.append(URLEncoder.encode(product));
      sb.append("&quantity=");
      sb.append(URLEncoder.encode(String.valueOf(quantity)));
      String postData = sb.toString();

      // Create the URL from which the quote will be read

      URL supplierURL = new URL(getCodeBase(), "PriceQuote.jsp");

      // Open a URLConnection instance

      HttpURLConnection con = (HttpURLConnection)
         supplierURL.openConnection();

      // Set up for writing and reading

      con.setDoOutput(true);
      con.setDoInput(true);
      con.setUseCaches(false);

      // Tell the server that the input stream
      // contains POST data and give it the length
```

精彩好书见www.book118.com


```
        con.setRequestProperty(
            "Content-type",
            "application/x-www-form-urlencoded");

        con.setRequestProperty(
            "Content-length",
            String.valueOf(postData.length()));

        // Open an output stream for the connection
        // and write the POST data to it

        OutputStream out = con.getOutputStream();
        out.write(postData.getBytes());
        out.flush();

        // Open an input stream and copy the results
        // into the output text area

        BufferedReader in =
            new BufferedReader(
            new InputStreamReader(
            con.getInputStream()));

        txtOutput.setText("");

        while (true) {
            String line = in.readLine();
            if (line == null)
                break;
            txtOutput.append(line);
            txtOutput.append("\n");
        }

        // Close the output and input streams

        in.close();
        out.close();
    }
}
```

此applet组成为产品类型和质量的两个文本域，一个初始化搜索的按钮以及一个显示结果的文本域。这不是一本关于Swing的书籍，因此这里不涉及如何构建GUI的细节。这里感兴趣的方法是doPost（），原因有两个：

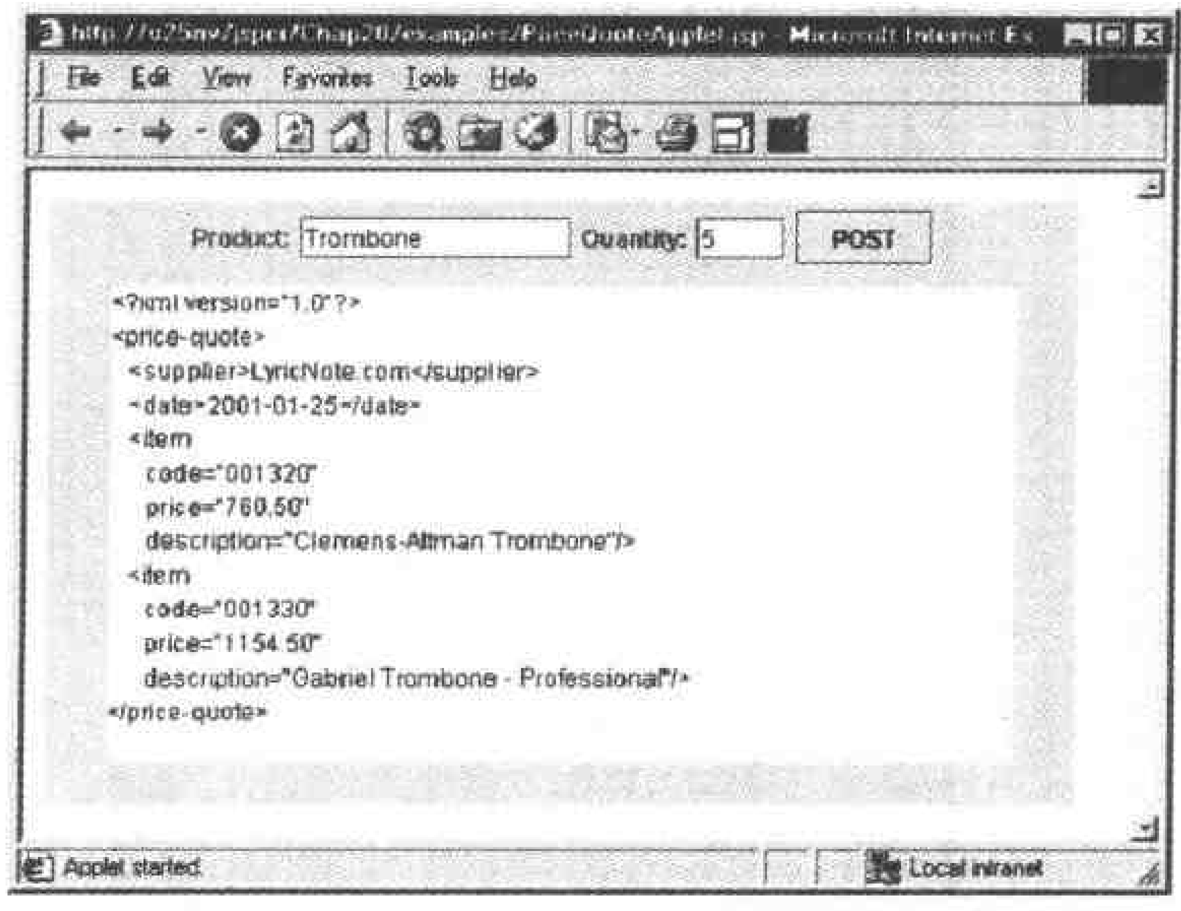
- 它显示出applet如何连接到URL输入流。
- 它使用HTTP POST方法实现目的。

精彩好书见www.book118.com

POST与GET方法不同，因为请求参数作为请求体给出，而不是附加到URL中。doPost（）使用第12章（HTML窗体）描述的URL解码机制将请求体构建成名字/取值对，然后打开至JSP页面的一个URLConnection，并配置其内容类型和长度，最后写入请求体。doPost（）读取结果并在输出文本域中显示它们。

包含此applet的JSP页面如下所示。下图给出输出结果。

```
<%@ page session="false" %>
<jsp:plugin
   type="applet"
   code="PriceQuoteApplet.class"
   codebase="."
   width="500"
   height="300"
   jreversion="1.2"
   >
</jsp:plugin>
```



运行PriceQuoteApplet的结果

20.4 一个Perl 客户端

因为在JSP服务器和其客户端之间的通信使用的是HTTP，甚至可以不用Java编写客户端。例如，Perl是一种广泛使用的脚本语言，特别是在CGI应用、文本处理和系统管理中。带有优秀的套接字的支持，Perl可以很容易地访问用JSP编写的服务器应用。

通过提供更好的数据库支持JSP可以扩展Perl的一个区域。因为实际上所有的数据库系统都可以通过某种形式的JDBC驱动器访问并且Perl可以从JSP URL连接中进行读取，Perl应用

精彩好书见www.book118.com

可以将JSP作为数据库访问的中间件使用。

20.4.1 通用数据库选择服务器

这里列出的JSP页面提供对任意JDBC可访问的数据库进行SQL SELECT查询的方式。它接受带有3个参数的HTTP请求：

- DRIVER JDBC 驱动器类名。
- URL JDBC数据库URL。
- QUERY 要执行的SELECT语句。

结果写入的格式为以tab键分隔取值格式，在第一行带有列名。

注意 此JSP页面阐述了防止JSP元素引入的无关空格干扰严格输出格式的另一种方式。实现方式是通过调用response.reset（ ）在写入真正文本的第一行前清除输出缓存。注意，reset（ ）还清除写入的任意头标和状态码。为此，Servlet 2.3规范加入了一个resetBuffer（ ）方法。

```
<%@ page session="false" %>
<%@ page import="java.io.*" %>
<%@ page import="java.sql.*" %>
<%
   Connection con = null;
   try {

      // Get the driver name and database URL parameters

      String driver = request.getParameter("DRIVER");
      if (driver == null)
         throw new ServletException
         ("No driver class name specified");

      String url = request.getParameter("URL");
      if (url == null)
         throw new ServletException
         ("No url class name specified");

      // Get the SELECT statement to be executed

      String query = request.getParameter("QUERY");
      if (query == null)
         throw new ServletException
         ("No QUERY parameter specified");

      // Verify that it is a SELECT statement

      query = query.trim();
```

精彩好书见www.book118.com


```
if (!query.toUpperCase().startsWith("SELECT"))
   throw new ServletException
   ("Only SELECT statements are valid");

// Make sure the driver is loaded

Class.forName(driver);

// Open the connection

con = DriverManager.getConnection(url);

// Compile the query statement. If it is invalid,
// a SQLException will be thrown.

PreparedStatement stmt = con.prepareStatement(query);

// Execute the query

ResultSet rs = stmt.executeQuery();

// Reset the response buffer to eliminate
// any nonsignificant whitespace

response.reset();

// Write the column headings

response.setContentType("text/tab-separated-values");

ResultSetMetaData rmd = rs.getMetaData();
int nColumns = rmd.getColumnCount();
StringBuffer buffer = new StringBuffer();
for (int i = 0; i < nColumns; i++) {
   int col = i+1;
   if (i > 0)
      buffer.append("\t");
   buffer.append(rmd.getColumnName(col));
}
out.println(buffer.toString());

// Write the data from the result set

while (rs.next()) {
   buffer = new StringBuffer();
   for (int i = 0; i < nColumns; i++) {
```

精彩好书见www.book118.com


```
            int col = i+1;
            if (i > 0)
               buffer.append("\t");
            buffer.append(rs.getString(col));
         }
         out.println(buffer.toString());
      }

      // Done

      rs.close();
      stmt.close();
   }
   finally {
      if (con != null) {
         try {
            con.close();
         }
         catch (SQLException ignore){}
      }
   }
%>
```

20.4.2 Perl脚本

Perl的非官方格言是"实现一件事有很多方式"。正是这样，任何Perl脚本都是实现同一任务许多可行方法的其中一种。

这里用来访问JSP数据库服务器的脚本是GetBooks.pl。它在端口80上打开至LyricNote Web服务器的一个套接字，然后进行带有3个参数的一个HTTP GET请求，其中包括要执行的SQL查询。

```
#! perl -w

# ==============================================
# Program: GetBooks
#
# Description:
#
# Sample Perl script that sends a database
# query to the SQLSelect.jsp
# ==============================================

use strict;
use IO::Socket;
```

精彩好书见www.book118.com


```
my $hostName = "u25nv";
my $hostPort = "80";

# Open a socket to the host

my $socket = new IO::Socket::INET(
       PeerAddr => $hostName,
       PeerPort => $hostPort,
       Proto => "tcp"
       );

# Set autoflush on

my $saveSelect = select $socket;
$| = 1;
select $saveSelect;

# Create the command

my $cmd = "";
$cmd .= "DRIVER=" . encode("org.enhydra.instantdb.jdbc.idbDriver");
$cmd .= "&URL=" . encode("jdbc:idb:D:/lyricnote/WEB-INF/
database/products/db.prp");
$cmd .= "&QUERY=" . encode(<<EOF);
SELECT itemCode, description
FROM PRODUCTS
WHERE PRODTYPE='BK'
EOF

my $cmdLength = length($cmd);

# Send the HTTP request

print $socket (<<EOF);
POST /jspcr/Chap20/examples/SQLSelect.jsp HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: $cmdLength

$cmd
EOF

# Read back the status code

my $line = <$socket>;
my ($httpVersion, $status) = split(/\s+/, $line);
if ($status != 200) {

   # Handle the error ...
```

精彩好书见www.book118.com


```
    }
    else {

        # Skip the rest of the headers and display the results.
        # End of headers is signaled by a blank line.

        my $inData = 0;
        while (<$socket>) {
            chomp;

            ($inData == 0) && do {
                $inData = 1 unless (/\S/);
                next;
            };

            ($inData == 1) && do {
                print "$_\n";
                next;
            };
        }
    }

    # Done

    $socket->close();

    # Subroutine to URL-encode a parameter string

    sub encode {
        my $s = shift;
        $s =~ s/([^A-Za-z0-9 ])/"%" . sprintf("%02X", ord($1))/eg;
        $s =~ s/ /+/g;
        return $s;
    }
```

这里，GetBooks.pl请求LyricNote产品数据库使用的InstantDB数据库驱动器，请求每个与音乐相关书籍的条目代码和描述。下面是结果：

```
itemcode    description
000030      Dorothy Wendecker: Bartok in New York
000040      Conrad Stock: Beethoven and the Weather
000120      Louis Krouse: The Bad Tsar
000150      Alice Gabriel: Did Salieri Do the Deed?
000160      John Glass: Stravinsky and the 20th Century Ballet
000170      Gray Raphael: Vox Humana
000200      Nicholas Thiers: Oh Boy! Oboe!
```

精彩好书见www.book118.com


```
000220 Douglas Benton: Some Kind of Brass
000240 Theresa McDonald: The Lyric Viola
000270 Violet Barber: Who's Afraid of the Twelve Tone Row?
000280 Rita Fall: What's My Melodic Line
000290 Mary Wright: More Ballet Bloopers
000330 Anna Maria Pontius: Purcell Mania
```

20.5 小结

虽然最常用的JSP客户端是一个Web浏览器，任何使用HTTP协议的程序都可以充做一个客户端。本章介绍了三种可选择的客户端：

- 请求动态创建的XML文档的单机Java应用。
- 使用HTTP POST访问JSP服务器的Java applet。
- 使用JSP作为其数据库服务器的Perl脚本。

HTTP通信链接所提供的灵活性使得基于JSP的组件可以并入到各种可能类型的应用中。

精彩好书见www.book118.com

第21章　与其他服务器通信

前面一章讲述不同Web浏览器的客户端可以访问JSP页面。将HTTP作为一种通用技术，Java应用、applet和用其他语言编写的程序可以使用JSP页面作为任何类型系统的非可视化组件。

反过来也成立——JSP页面可以用作其他服务器的客户端。在这一章，将介绍两种此类的服务器环境以及JSP页面如何与之进行互操作。

21.1　服务器端脚本环境

JSP和servlet提供Internet上的动态内容，但许多技术也可以做到这一点：

- CGI　通用网关接口驱动上千个交互式Web站点，并通常带有访问数据库和其他系统资源的Perl脚本。
- ASP　Microsoft的动态服务页面广泛使用于服务器环境，它使开发商可以将HTML和脚本命令混合在一起以提供动态内容。
- PHP　PHP是一种跨平台开放源，使用嵌入的HTML和Perl类似语言的服务器脚本环境。
- Cold Fusion　Allaire的Cold Fusion是一个服务器端的应用环境。它使用一种基于标签的服务器脚本语言，称为CFML，以执行数据库访问和生成Web输出。

找到使用以上多个技术或全部技术的组织通常并不容易，因为企业计算资源常常是分布的，并满足不同方案的分区或分部门的需要。不管JSP或其他技术是多么优秀，也很难有组织地形成一种通用的应用环境。遗留应用转换是很麻烦的。然而，转换它们也可能是不必要的，因为在许多情况下，这些应用可以互操作。再一次说明，关键的基础结构是HTTP协议，一种可用于所有这些环境的通用传输机制。

使用HTTP进行互操作

互操作需求产生的因素是相关公司的联合行为。一组公司可能属于使用加强的会计系统的父公司。子公司可能需要递交其预算数字、售卖信息和工资数据。它们也可能需要从父公司下载价格，折扣率和上级的指示。这些公司可以使用不同的应用系统，特别地如果它们是结合在一个合并或收购的企业中。

本书假想的LyricNote.com就是这样一个公司。它的商品以美元和加拿大元定价，它的父公司来保证兑换率。使用的兑换率每天都会变，父公司利用一个所有子公司使用的CGI程序使之可利用。LyricNote.com的产品目录Web应用需要使用此兑换率信息。对JSP页面，这意味着从URL连接中进行读取。

从远程网络资源进行读取

回忆第20章，java.net.URL和java.net.URLConnection均提供读取一个远程网络资源生成的输入流方法。对一个GET请求，此基本技术为：

精彩好书见www.book118.com


```
URL url = new URL("http://servername/path?parm=value");
URLConnection con = url.openConnection();
InputStream in = con.getInputStream();
```

或使用java.net.URL中的一种便捷方法：

```
URL url = new URL("http://servername/path?parm=value");
InputStream in = url.openStream();
```

如果需要在打开输入流之前进一步配置连接，最好选择第一个方法。如果需要发送请求头标或需要使用HTTP POST方法，就是这种情况。如下所示：

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

public class PostRateRequest
{
   public static void main(String[] args)
      throws Exception
   {
      // Set up the two request parameters

      String postData = "c1=USD&c2=CAD";

      // Open the URL connection for reading and writing

      URL url = new URL(
         "http://u25nv/cgi-bin/currency/GetRate.cgi");
      URLConnection con = url.openConnection();
      con.setDoOutput(true);
      con.setDoInput(true);

      // Set request headers for content type and length

      con.setRequestProperty(
         "Content-type",
         "application/x-www-form-urlencoded");

      con.setRequestProperty(
         "Content-length",
         String.valueOf(postData.length()));

      // Issue the POST request

      OutputStream out = con.getOutputStream();
      out.write(postData.getBytes());
      out.flush();
```

精彩好书见www.book118.com


```
            // Read the response

            InputStream in = con.getInputStream();
            while (true) {
               int c = in.read();
               if (c == -1)
                  break;
               System.out.print((char) c);
            }
            System.out.flush();

            // Done

            in.close();
            out.close();
         }
      }
```

LyricNote.com使用此技术从其父公司Web站点中读取美元到加拿大元的兑换率。下面给出的CatalogSearch.jsp列出了同时具有美元和加拿大元的产品价格。它从前面例子使用的CGI货币程序中得到兑换率。并将该程序嵌入其jspInit（）方法。图21-1显示出结果。

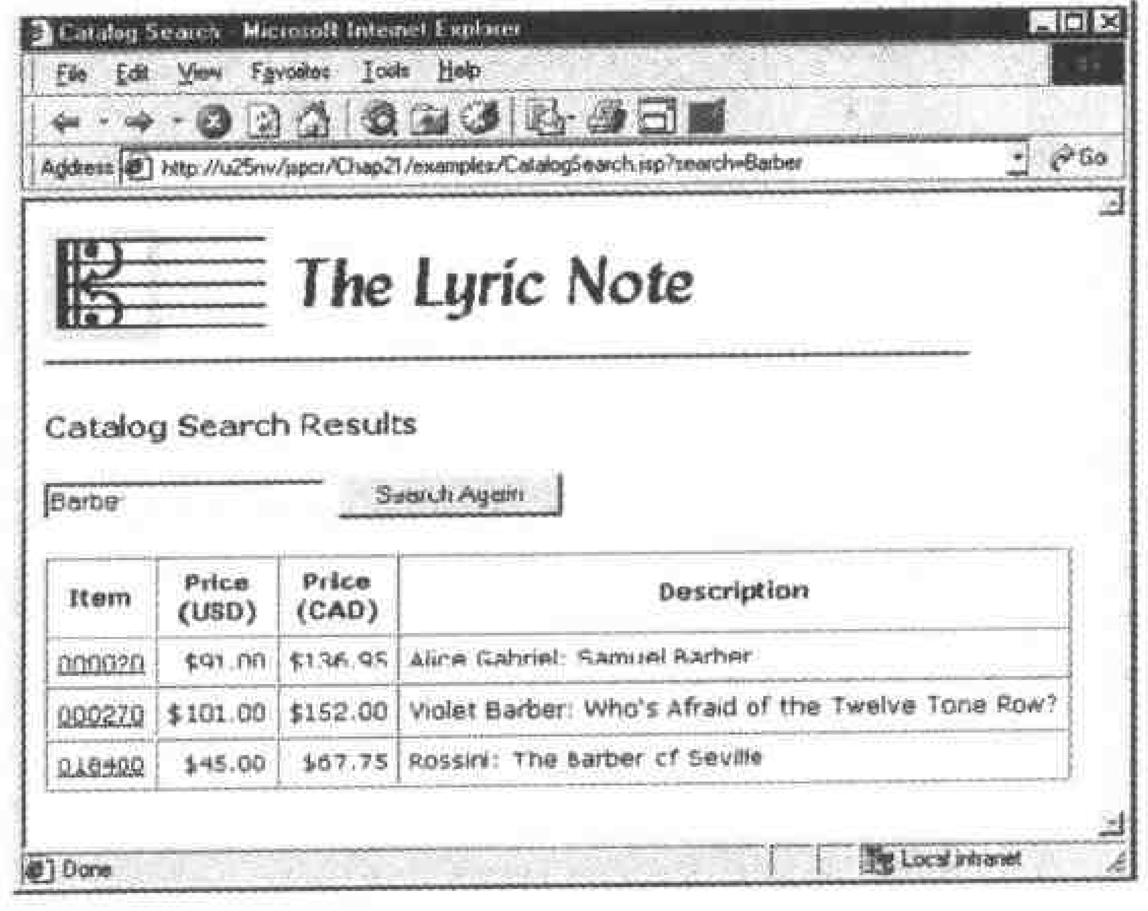


图21-1 显示从CGI资源中获得的USD-CAD兑换率的目录搜索输出。

在jspInit（）中，JSP页面对父公司提供的USD-CAD兑换率使用了HTTP GET请求。它读取单行结果，并将其分成两行。在后面，从数据库中读取价格（全美分格式）并转换成两位小数点的美元格式。然后可应用兑换率得到等价的加拿大元，误差在5美分之内。

精彩好书见www.book118.com


```
<%@ page session="false" %>
<%@ page import="java.io.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.text.*" %>
<%!
   private static double EXCHANGE_RATE;

   // Get the US to Canadian dollar exchange rate

   public void jspInit()
   {
      try {
         URL url = new URL(
            "http://u25nv/cgi-bin/currency/GetRate.cgi"
            + "?c1=USD"
            + "&c2=CAD");
         BufferedReader in =
            new BufferedReader(
            new InputStreamReader(
            url.openStream()));
         String line = in.readLine();
         in.close();
         EXCHANGE_RATE = Double.parseDouble(line);
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
%>
<%
   // Get search string

   String search = request.getParameter("search");
   if (search == null)
      search = "";
   search = search.trim();

   // Connect to database

   String DRIVER = "org.enhydra.instantdb.jdbc.idbDriver";
   String DB_URL = "jdbc:idb:" +
      "D:/lyricnote/WEB-INF/database/products/db.prp";

   Class.forName(DRIVER);
   Connection con = null;
   try {
      con = DriverManager.getConnection(DB_URL);

      // Create a query using the search string
```

精彩好书见www.book118.com


```
        PreparedStatement stmt = con.prepareStatement
           ("select itemcode, price, description"
           + " from products"
           + " where description like ?");
        stmt.setString(1, "%" + search + "%");

        // Run the query and display the results

        ResultSet rs = stmt.executeQuery();
%>
<HTML>
<HEAD>
<TITLE>Catalog Search</TITLE>
</HEAD>
<BODY>
<IMG SRC="images/lyric_note.png">
<HR WIDTH="500" ALIGN="LEFT" COLOR="#005A9C">
<H3>Catalog Search Results</H3>
<FORM>
<INPUT TYPE="TEXT" NAME="search" VALUE="<%= search %>">
<INPUT TYPE="SUBMIT" VALUE="Search Again">
</FORM>
<TABLE BORDER="1" CELLPADDING="5" CELLSPACING="0">
<TR>
   <TH>Item</TH>
   <TH>Price<BR>(USD)</TH>
   <TH>Price<BR>(CAD)</TH>
   <TH>Description</TH>
</TR>
<%
        NumberFormat fmt = NumberFormat.getCurrencyInstance();
        while (rs.next()) {
           String itemCode = rs.getString(1);
           double price = rs.getDouble(2) / 100;
           double price_c =
              ((long)(price * EXCHANGE_RATE * 20 + 0.5)) / 20.0;
           String description = rs.getString(3);
%>
<TR>
   <TD><A HREF="productDetail.jsp?itemCode=<%= itemCode %>"
         ><%= itemCode %></A></TD>
   <TD ALIGN="RIGHT"><%= fmt.format(price) %></TD>
   <TD ALIGN="RIGHT"><%= fmt.format(price_c) %></TD>
   <TD><%= description %></TD>
</TR>
<%
        }
%>
```

精彩好书见www.book118.com


```
</TABLE>
</BODY>
</HTML>
<%
    }
    finally {
        if (con != null)
            con.close();
    }
%>
```

21.2 从一个JSP页面发送邮件

第19章的产品支持系统分析示例掩盖了一个程序上的难点。当问题路由到支持人员时，此人怎么知道呢？开发商和测试者可以读取其支持的产品问题的当前列表，但他们不知道一个新的问题何时出现，除非他们碰巧正在查找它。当顾客在等待时，通过某种主动的过程通知支持人员特别重要。这时电子邮件就起作用了。

从程序中通过电子邮件进行通知一般不是一个好主意，特别是如果使用了硬编码地址，它不可避免地要过时。然而在这里，支持人员的标识和其电子邮件地址可从产品支持数据库中获得。因为电子邮件是一种很熟悉的机制，其基础结构已经存在。这是对该问题一种很好的解决方案。

21.2.1 发送邮件的方法

从应用中发送电子邮件有几种可选方式。这一节考虑下面3种方式：

- 使用套接字的SMTP。
- sun.net.smtp.SmtpClient类。
- JavaBeanMail API。

1. SMTP

最简单的方式是使用TCP/IP套接字上的简单邮件传输协议（SMTP）。追溯到1982年，SMTP是最老的Internet协议之一。它采用一个小的文本命令集以提供电子邮件参数：

HELO 标识发送者区域。

MAIL 标识发送者。

RCPT 标识接收者，可以使用多个RCPT命令。

DATA 标识消息体的开头。直到下一行“.”前为止，所有内容均是体的一部分。

QUIT 终止会话。

一个SMTP会话的例子如下所示（客户端发送的行由boldface指出）：

```
220 pluto.lyricnote.com ESMTP Sendmail 8.9.3/8.9.3;
   Mon, 29 Jan 2001 06:58:24 -0500 (EST)
HELO lyricnote.com
250 pluto.lyricnote.com Hello dialup.rdu.lyricnote.com
   [209.170.132.190], pleased to meet you
```

精彩好书见www.book118.com


```
MAIL FROM: phanna@lyricnote.com
250 phanna@lyricnote.com... Sender ok
RCPT TO: phanna@lyricnote.com
250 phanna@lyricnote.com... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
SUBJECT: Mail Test
This is a test of the mail system
This is only a test
Beeeeeeeeeeeeeeeeeeeeeeeeep
This concludes the test of the mail system
.
250 UAA07253 Message accepted for delivery
QUIT
221 pluto.lyricnote.com closing connection
```

完整的SMTP协议规范在RFC 821中，可在http://www.freesoft.org/CIE/RFC/821/index.htm上找到。

如例子所示，SMTP邮件可通过打开一个至邮件主机的java.net.Socket并使用其输入和输出流被简单发送。这种基于套接字的方法的优点是很容易实现，但当加入如附件等内容时，它就会变得很复杂。为此，很少有应用直接使用它。

2. sun.net.smtp.SmtpClient类

另一个方案是使用Sun微系统的Java运行时环境所提供的sun.net.smtp.SmtpClient。此类是原始SMTP套接字协议的一个面向对象的包容器。前面例子的SmtpClient版本如下：

```
import java.io.*;
import sun.net.*;
import sun.net.smtp.*;

public class MailTest
{
   public static void main(String[] args)
      throws Exception
   {
      SmtpClient client = new SmtpClient("mail.lyricnote.com");

      client.from("phanna@lyricnote.com");
      client.to("phanna@lyricnote.com");

      PrintStream out = client.startMessage();
      out.println("SUBJECT: Mail test");
      out.println("This is a test of the mail system");
      out.println("This is only a test");
      out.println("Beeeeeeeeeeeeeeeeeeeeeeeeep");
      out.println("This concludes the test of the mail system");
      client.closeServer();
   }
}
```

精彩好书见www.book118.com

此方法比使用套接字简单一些。它有一个主要的缺陷：sun.net.*类并未归档，正不断发生变化。Sun允许使用它们（实际上，Sun JRE没有它们就不能工作），但提出警告它们可能在没有通知的情况下改变或由新版本所替代。

3. JavaMail API

第3种方法是JavaMail API。JavaMail是以抽象方式模拟邮件组件的API集合——POP、SMTP、IMAP和其他邮件协议的可插入式体系结构。JavaMail对JDK 1.1.x和更高版本适用，是Java 2企业版（J2EE）的必需组件。

注意 JavaMail同时包括发送和接收邮件的类。这里只考虑发送方。JavaMail类和完整文档可从http://java.sun.com/products/javamail处下载。

对前面给出的同一个例子的JavaMail版本如下：

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class JavaMailTest
{
   public static void main(String[] args)
      throws Exception
   {
      // Create a session with the LyricNote mail host

      Properties props = new Properties();
      props.put("mail.host", "mail.lyricnote.com");
      Session mailSession = Session.getInstance(props, null);

      // Create address objects for the sender and receiver

      Address fromUser =
         new InternetAddress("phanna@lyricnote.com");
      Address toUser =
         new InternetAddress("phanna@lyricnote.com");

      // Create the message body

      Message body = new MimeMessage(mailSession);
      body.setFrom(fromUser);
      body.setRecipient(Message.RecipientType.TO, toUser);
      body.setSubject("Mail Test");
      body.setContent(
         "This is a test of the mail system\n"
         + "This is only a test\n"
         + "Beeeeeeeeeeeeeeeeeeeeeeeeeeeeeep\n"
         + "This concludes the test of the mail system",
```

精彩好书见www.book118.com


```
            "text/plain");

        // Send the message

        Transport.send(body);
    }
}
```

典型情况，使用JavaMail发送邮件包括7个步骤：

1) 使用Session.getInstance（）创建至邮件主机的一个会话。

2) 使用new InternetAddress（）创建发送者和接收者地址对象。

3) 使用new MimeMessage（Session session）创建一个消息体。

4) 使用Message对象的setFrom（）和setRecipient（）指定地址。

5) 使用setSubject（）指定主题。

6) 使用setContent（）指定消息体和解码类型。

7) 使用Transport.send(message)发送消息。

第一步创建一个新的Session对象。Session的作用是连接到邮件主机。可通过调用静态方法Session.getInstance(Properties props,Authenticator auth)获得此对象的一个新实例。此方法提供属性最小情况下必须包含邮件主机。如果不需要确认，auth参数可以为null。

第二步将发送者和接收者表示为InternetAddress对象。此类依据RFC 822（"ARPA Internet文本消息的格式标准"，可从http://www.freesoft.org/CIE/RFC得到）模型地址。

接着，使用newMimeMessage（）创建一个消息体对象。此类表示一个多部分的Internet邮件消息，包括其内容和头标。构造器带有Session对象的一个指针，因此消息可以与邮件主机和其他会话参数相关。

第四步分别使用setFrom（Address fromUser）和setRecipient（Message.RecipientType type，Address toUser）设置消息体的from和recipient属性。type参数用于区分TO、CC，和BCC接收者。

第五步可选，使用消息对象的setSubject（string subject）方法设置邮件消息的主题。

第六步使用消息对象的setContent（Object body，String type）方法创建实际的消息文本。body参数指定文本，type指出MIME类型（通常是text/plain）。

最后，使用静态Transport.send(Message message)方法发送邮件。Transport是具体实现由邮件服务提供者提供的抽象类，如Sun的smtp.jar文件。

21.2.2 在产品支持系统中的电子邮件通告

回到本书的问题。产品支持系统路由问题到客户时需要通告支持人员、开发商和测试者。此路由发生在模式组件中，具体是其addProblemLog（）方法（见第19章的com.lyricnote.support.Model类列表）。路由事件ID如下：

RPS 路由到产品支持人员。

RPD 路由到产品开发商。

精彩好书见www.book118.com

RQA 路由到测试者。

在addProblemLog（）中。可以判断事件ID是否为以上三种。如果是，调用一个新的模式方法notifySupport（ProblemLog log），如下：

```
/**
* Sends email to the appropriate support person
*/
public void notifySupport(ProblemLog log)
    throws SQLException, IOException
{
    // Get the problem object

    String problemID = log.getProblemID();
    setProblemID(problemID);
    Problem problem = getProblem();

    // Create the subject line from the problem ID
    // and problem description

    StringBuffer sb = new StringBuffer();
    sb.append("Problem ID: ");
    sb.append(problemID);
    sb.append(" ");
    sb.append(problem.getDescription());
    String subject = sb.toString();

    // Get the product object. We need this to find out
    // the support ID's and the corresponding e-mail
    // addresses

    String productID = problem.getProductID();
    setProductID(productID);
    Product product = getProduct();

    // Determine the appropriate party to receive the mail

    String employeeID = null;
    String eventDescription = null;
    String eventID = log.getEventID();
    if (eventID.equals("RPS")) {
        employeeID = product.getProductSupport();
        eventDescription = "ROUTED TO PRODUCT SUPPORT";
    }
    else
    if (eventID.equals("RPD")) {
        employeeID = product.getDeveloper();
```

精彩好书见www.book118.com


```
                eventDescription = "ROUTED TO DEVELOPMENT";
            }
            else
            if (eventID.equals("RQA")) {
                employeeID = product.getTester();
                eventDescription = "ROUTED TO TEST";
            }
            else
                return;

            eventDescription += "\r\n";
            eventDescription += log.getComments();

            // Lookup that person's email address

            Employee employee = getEmployee(employeeID);
            String email = employee.getEmail();

            // Send mail to the party

            Address fromUser = new InternetAddress
                ("support@lyricnote.com", "Product Support System");
            Address toUser = new InternetAddress
                (email, employee.getName());

            Properties props = new Properties();
            props.put("mail.host", "mail.lyricnote.com");
            Session mailSession = Session.getInstance(props, null);
            Message body = new MimeMessage(mailSession);

            try {
                body.setFrom(fromUser);
                body.setRecipient(Message.RecipientType.TO, toUser);
                body.setSubject(subject);
                body.setContent(eventDescription, "text/plain");
                Transport.send(body);
            }
            catch (MessagingException e) {
                throw new IOException(e.getMessage());
            }
    }
```

图21-2显示了这种行为新的特性。MIDI转换器产品报告出一个问题，电话中心代理填写问题报告并选择行为“路由到产品支持”。当代理点击确认按钮时，问题被加入数据库，调用模式的addProblemLog（）方法。因为事件是RPS，调用notifySupport（）方法，最后如图21-3显示的消息被发送到产品支持人员。

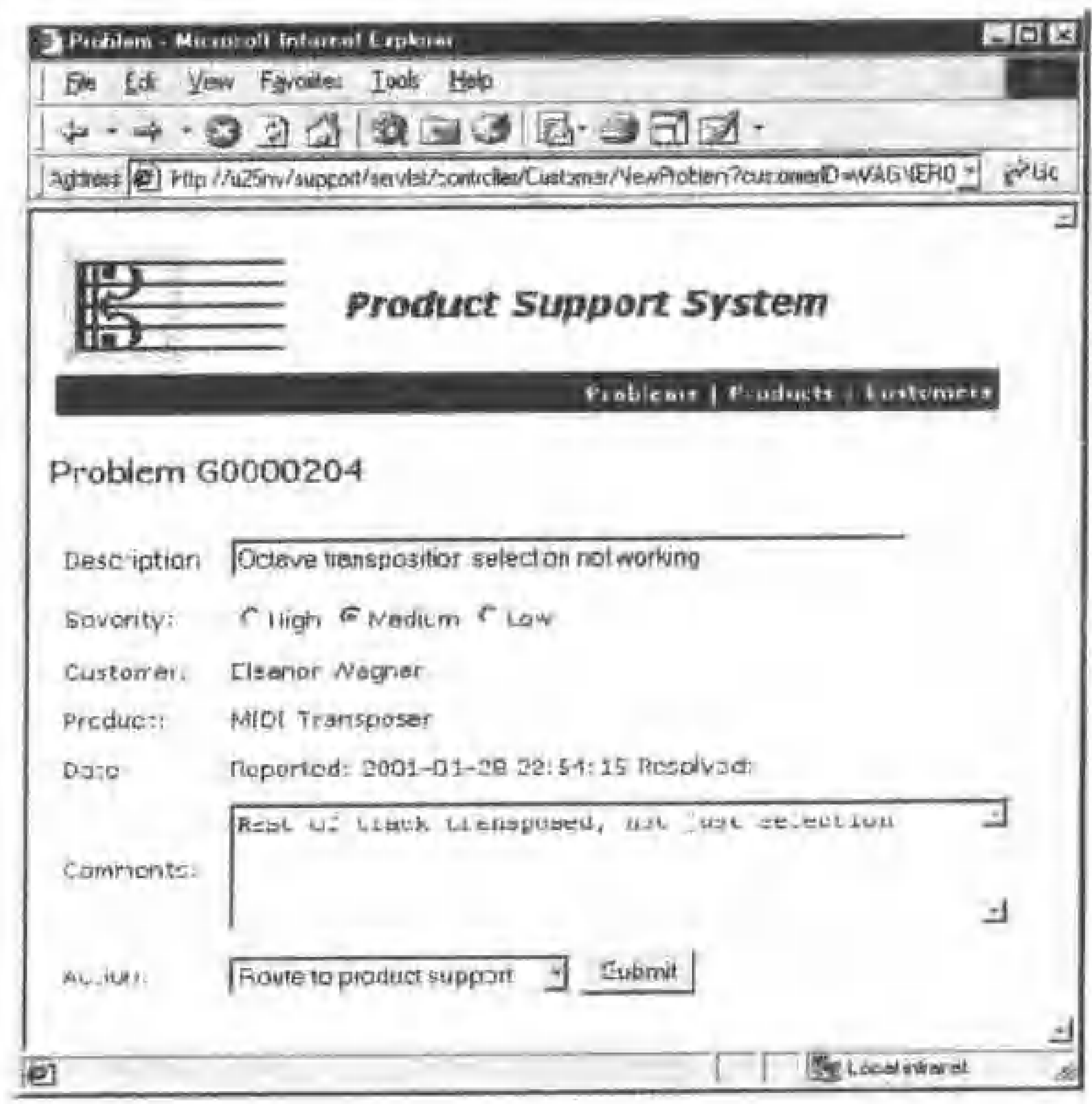


图21-2 包含至产品支持中心路由的问题报告

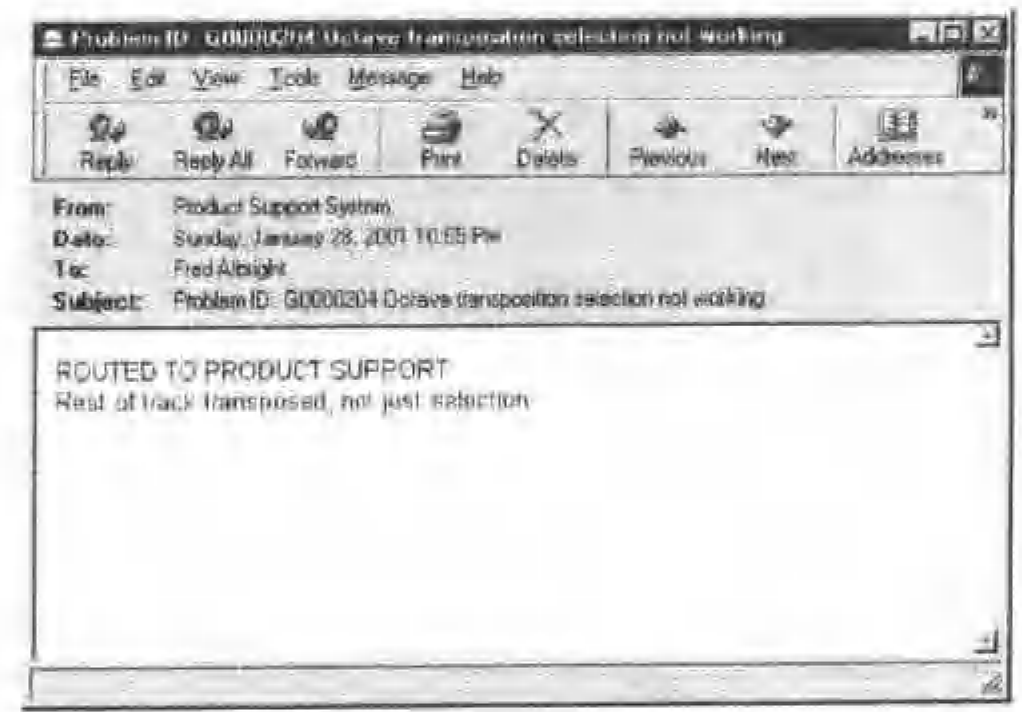


图21-3 发送到产品支持人员的电子邮件消息

21.3 小结

正像Web浏览器的程序可以是JSP客户端一样，JSP页面也可以是其他服务器的客户端。本章

精彩好书见www.book118.com

讨论了这样的两个环境：

- 从一个CGI服务器获得数据。
- 使用JavaMail API发送邮件。

JSP、servlet、CGI、PHP和ASP以及其他的服务器端脚本环境都使用一种通用技术向其客户端发送内容：HTTP协议。java.net包提供了URL和URLConnection类，它们具有通过HTTP访问资源的方法。

JavaMail API是一种模拟邮件系统各个部分的可扩展和可升级的体系结构。特定邮件协议，如POP、SMTP和IMAP的实现可随意利用。

精彩好书见www.book118.com

第五部分 附 录

本书的这一部分包含有关servlet API、JSP API和HTTP协议的参考材料。

附录A servlet API 版本2.3

此附录描述位于如下两个servlet包的每个类：

- javax.servlet 非任何具体协议指定的类。
- javax.servlet.http HTTP协议指定的类

对于每个类，包括如下小节：

- 类名。
- 上下文（全称、类型、超类和实现的接口）。
- 类描述。
- 类中每个方法的详细信息。

注意 这里描述的类和方法是基于servlet 2.3 规范的最终公开草案。尽管此最终草案非常接近于官方的规范，但是，仍然有可能被修改。如果有任何疑问，请参考此规范的最新版本，位于http://java.sun.com/products/servlet/index.html。

许多类和方法描述是不受赞成的。这意味着，不推荐使用它们，而且在以后的版本中，可能不再支持它们。

javax.servlet 包

Filter

全称：javax.servlet.Filter

类型：接口

实现此Filter接口的类根据请求、响应或者它们二者实施过滤操作。这些类实现doFilter()来完成此操作。

方法：

doFilter

```
public void doFilter(
      ServletRequest request,
      ServletResponse response,
```

精彩好书见www.book118.com


```
        FilterChain chain)
    throws IOException, ServletException
```

一旦请求经由过滤器链传递，则servlet引擎就调用此方法。

getFilterConfig

```
public FilterConfig getFilterConfig()
```

• 返回此过滤器的FilterConfig。

setFilterConfig

```
public void setFilterConfig(FilterConfig filterConfig)
```

设置FilterConfig 对象。当处理完此Filter时，servlet引擎也使用null调用该方法。

FilterChain

全称：javax.servlet.FilterChain

类型：接口

父接口　无

提供一个过滤器调用链视图。过滤器使用FilterChain激活链中的下一个过滤器。

方法:

doFilter

```
public void doFilter(
        ServletRequest request,
        ServletResponse response)
    throws IOException, ServletException
```

激活链中下一个过滤器或者位于链结尾处的资源。

FilterConfig

全称：javax.servlet.FilterConfig

类型：接口

在初始化期间向过滤器传递信息。

方法：

getFilterName

```
public String getFilterName()
```

返回过滤器的名称。

getInitParameter

```
public String getInitParameter(String name)
```

返回指定参数的值，如果该参数不存在返回null。

getInitParameterNames

精彩好书见www.book118.com


```
public Enumeration getInitParameterNames()
```

返回servlet 初始化参数名称的一个枚举。如果初始化参数不存在，则返回一个空枚举。

getServletContext

```
public ServletContext getServletContext()
```

返回调用者运行于其中的ServletContext。

GenericServlet

全称：javax.servlet.GenericServlet
类型：抽象类
实现：javax.servlet.Servlet
javax.servlet.ServletConfig
java.io.Serializable

servlet的一种基类，它不使用由HTTP协议指定的功能。GenericServlet实现所有servlet的基本功能：

初始化
请求处理
中止

惟一必须覆盖的方法是service()，它实际上处理请求。HTTP servlet的基类HttpServlet是一种更常用的servlet超类。

构造器：

GenericServlet

```
public GenericServlet()
```

一个什么都不作的空构造器。任何servlet初始化都应该在init()方法中实现。

方法：

destroy

```
public void destroy()
```

当卸载servlet时，由servlet引擎调用。servlet作者可以覆盖此方法来释放任何被分配的资源。在GenericServlet中，此方法只不过记录它被执行的事实。

getInitParameter

```
public String getInitParameter(String name)
```

给定一个初始化参数名称，返回此参数的值。如果此参数不存在，则返回null。

getInitParameterNames

```
public Enumeration getInitParameterNames()
```

返回此servlet存在的全部初始化参数名称的枚举值。

getServletConfig

精彩好书见www.book118.com


```
public ServletConfig getServletConfig()
```

返回与此servlet相关的ServletConfig对象。

getServletContext

```
public ServletContext getServletContext()
```

返回与此servlet相关的ServletContext对象。

getServletInfo

```
public String getServletInfo()
```

返回关于此servlet的标识信息，如作者、版本和版权。

getServletName

```
public String getServletName()
```

返回此servlet实例的名称。

init

```
public void init() throws ServletException
```

一个调用super.init(config)的便利方法。

init

```
public void init(ServletConfig config)
   throws ServletException
```

servlet容器调用该方法来表明：此servlet将被置于服务中。

log

```
public void log(String msg)
```

向servlet日志文件写入指定的消息。

log

```
public void log(String message, Throwable t)
```

• 向servlet日志文件写入指定的消息。位于Throwable的消息也写入日志文件。

service

```
public abstract void service(
        ServletRequest req,
        ServletResponse res)
throws ServletException, IOException
```

由servlet容器调用该方法，以便允许此servlet处理请求。

RequestDispatcher

全称：javax.servlet.RequestDispatcher

类型：接口

精彩好书见www.book118.com

一个用于从客户端向位于服务器的任何资源传递请求的对象。

方法：

forward

```
public void forward(
        ServletRequest request,
        ServletResponse response)
    throws ServletException, IOException
```

从一个servlet向另一个servlet、JSP页面或者HTML文档传递请求。只有在已经提交响应之前，才可以调用此方法。

include

```
public void include(
        ServletRequest request,
        ServletResponse response)
    throws ServletException, IOException
```

引用其他servlet、JSP页面和位于当前输出缓冲区的HTML文档的内容。此被引用的资源可能没有设置任何响应头标。

Servlet

全称：javax.servlet.Servlet

类型：接口

定义所有servlet必须实现的方法。servlet API在GenericServlet和HttpServlet类中提供此接口的具体实现。

方法：

destroy

```
public void destroy()
```

当自服务中取出servlet时，servlet容器调用该方法。

getServletConfig

```
public ServletConfig getServletConfig()
```

返回与此servlet相关的ServletConfig对象。

getServletInfo

```
public String getServletInfo()
```

返回关于servlet的标识信息。

init

```
public void init(ServletConfig config) throws ServletException
```

当将servlet置入服务中时，由servlet容器调用。init方法必须在此servlet接受任何请求之前正常结束。

精彩好书见www.book118.com

service

```
public void service(
        ServletRequest req,
        ServletResponse res)
    throws ServletException, IOException
```

servlet容器调用该方法来处理请求。

ServletConfig

全称：javax.servlet.ServletConfig
类型：接口
在初始化期间，被servlet引擎用于向servlet传递信息。
方法：

getInitParameter

```
public String getInitParameter(String name)
```

返回指定初始化参数的值，或者返回null，如果此参数不存在。

getInitParameterNames

```
public Enumeration getInitParameterNames()
```

返回servlet的初始化参数名称的枚举值，或者返回一空枚举值，如果任何参数名称都不存在。

getServletContext

```
public ServletContext getServletContext()
```

返回与此servlet相关的ServletContext。

getServletName

```
public String getServletName()
```

按照在发布描述器web.xml中的记录，返回此servlet实例的名称。

ServletContext

全称：Javax.servlet.ServletContext
类型：接口
定义一方法列表，servlet可以将此列表用于与其servlet容器的通信。Servlet上下文可以保存可为此应用中全部servlet所使用的属性。
方法：

getAttribute

```
public Object getAttribute(String name)
```

返回拥有指定名称的应用级属性，或者返回null，如果此属性不存在。此对象必须强制转换

精彩好书见www.book118.com

为适当的类型。

getAttributeNames

```
public Enumeration getAttributeNames()
```

在servlet上下文中返回此属性名称的枚举值。

getContext

```
public ServletContext getContext(String uripath)
```

返回位于同一个服务器中的另一个URL的servletContext对象。此路径必须以"/"开始，且将被解析为相对于Web服务器的文档根目录。

getInitParameter

```
public String getInitParameter(String name)
```

返回指定的初始化参数。

getInitParameterNames

```
public Enumeration getInitParameterNames()
```

返回servlet上下文的初始化参数名称的枚举值，如果初始化参数不存在，返回空的枚举值。

getMajorVersion

```
public int getMajorVersion()
```

返回位于此servlet API 版本号的小数点左边的整数。

getMineType

```
public String getMimeType(String file)
```

返回指定文件的MIME类型，如果不识别此MIME类型返回null。

getMinorVersion

```
public int getMinorVersion()
```

返回位于此servlet API 版本号的小数点右边的整数。

getNamedDispatcher

```
public RequestDispatcher getNamedDispatcher(String name)
```

返回指定servlet的RequestDispatcher，如果无法返回此RequestDispatcher返回null。

getRealPath

```
public String getRealPath(String path)
```

在当前servlet上下文中给定一个URI，将此路径为它所指向的绝对文件名。

getRequestDispatcher

```
public RequestDispatcher getRequestDispatcher(String path)
```

返回指定资源的RequestDispatcher，或返回null，如果无法创建此RequestDispatcher。此路径名必须以"/"开始，且被解析为相对于servlet上下文的根目录。

精彩好书见www.book118.com

getResource

```
public URL getResource(String path) throws MalformedURLException
```

返回指定资源的URL，如果无法创建此资源返回null。此路径名必须以“/”开始，且被解析为相对于servlet上下文的根目录。

getResourceASStream

```
public InputStream getResourceAsStream(String path)
```

返回指定资源的InputStream，如果无法创建此资源返回null。此路径名必须以“/”开始，且被解析为相对于servlet上下文的根目录。

getResourcePaths

```
public Set getResourcePaths()
```

返回一组指向保存于此Web应用中的资源的路径。此路径名必须以“/”开始，且被解析为相对于servlet上下文的根目录。

getServerInfo

```
public String getServerInfo()
```

返回servlet引擎的名称和版本号。此返回值的形式为servername/version_number。

[不推荐使用]getServlet

```
public Servlet getServlet(String name) throws ServletException
```

因为安全性的原因，不再支持此方法。参见HttpSessionListener以获取一种替换的方法。

getServletContextName

```
public String getServletContextName()
```

返回Web应用的名称，对应于该ServletContext在发布描述器web.xml中的display-name元素。

[不推荐使用]getServletNames

```
public Enumeration getServletNames()
```

因为安全性的原因，不再支持此方法。参见HttpSessionListener以获取一种替换的方法。

[不推荐使用]getServlets

```
public Enumeration getServlets()
```

[不推荐使用]log

```
public void log(Exception exception, String msg)
```

不再支持此方法。请使用log(Exception exception,String msg)作替换。

log

```
public void log(String msg)
```

向servlet日志中写入指定的信息。

log

精彩好书见www.book118.com


```
public void log(String message, Throwable throwable)
```

向servlet日志中写入指定的信息和给定的Throwable的堆栈跟踪。

removeAttribute

```
public void removeAttribute(String name)
```

从servlet上下文中去掉拥有指定名称的属性。

setAttribute

```
public void setAttribute(String name, Object object)
```

在该servlet上下文中，以指定的属性名称保存对象。

ServletContextAttributeEvent

全称：javax.servlet.ServletContextAttributeEvent

类型：类

扩展：javax.servlet.ServletContextEvent

用于通知对web应用的servlet上下文的属性进行的修改。

构造器：

ServletContextAttributeEvent

```
public ServletContextAttributeEvent(
    ServletContext source,
    String name,
    Object value)
```

从指定名称和值的指定上下文来构造ServletContextAttributeEvent。

方法：

getName

```
public String getName()
```

返回被修改的属性的名称。

getValue

```
public Object getValue()
```

返回被添加、删除或替换的属性的值。该值取决于属性是被添加、修改还是删除。对于被修改或删除，返回的是原属性的值。对于添加，返回的是新属性的值。

ServletContextAttributesListener

全称：javax.servlet.ServletContextAttributesListener

类型：接口

父接口：java.util.EventListener

实现此接口的类将接受对servlet上下文的属性列表的修改通知。

精彩好书见www.book118.com

方法：

attributeAdded

```
public void attributeAdded(ServletContextAttributeEvent scab)
```

当向servlet上下文中添加新的属性时调用此方法。

attributeRemoved

```
public void attributeRemoved
    (ServletContextAttributeEvent scab)
```

当从servlet上下文中删除一个现有属性时调用此方法。

attributeReplaced

```
public void attributeReplaced
(ServletContextAttributeEvent scab)
```

当在servlet上下文中替换一个现有属性时调用此方法。

ServletContextEvent

全称：javax.servlet.ServletContextEvent

类型：类

扩展：java.util.EventObject

通知对servlet上下文的修改的事件类。

构造器：

ServletContextEvent

```
public ServletContextEvent(ServletContext source)
```

从指定的上下文创建ServletContextEvent。

方法：

getServletContext

```
public ServletContext getServletContext()
```

返回被修改的ServletContext。

ServletContextListener

全称：javax.servlet.ServletContextListener

类型：接口

父接口：java.util.EventListener

实现此接口的类将接受对servlet上下文属性列表的修改通知。

方法：

contextDestroyed

```
public void contextDestroyed(ServletContextEvent sce)
```

精彩好书见www.book118.com

当将要关闭此servlet上下文时调用此方法。

ContextInitialized

```
public void contextInitialized(ServletContextEvent sce)
```

当Web应用准备好处理请求时调用此方法。

ServletException

全称：javax.servlet.ServletException

类型：类

扩展：java.lang.Exception

一个通用的servlet异常。

构造器：

ServletException

```
public ServletException()
```

创建一个新的servlet异常。

ServletException

```
public ServletException(String message)
```

使用指定的消息创建一个新的servlet异常。

ServletException

```
public ServletException(String message, Throwable rootCause)
```

创建一个包含消息和根异常的新servlet异常。

ServletException

```
public ServletException(Throwable rootCause)
```

创建一个包含根异常的新servlet异常。

方法：

getRootCause

```
public Throwable getRootCause()
```

返回当前servlet异常的根异常。

ServletInputStream

全称：javax.servlet.Servlet InputStream

类型：抽象类

扩展：java.io.InputStream

servlet用于从客户端请求中读二进制数据的一种输入流。通常使用ServletRequest.getInputStream()方法得到。

精彩好书见www.book118.com

方法：

readLine

```
public int readLine(byte[] b, int off, int len)
throws IOException
```

从指定的偏移处开始，一次从输入流中读出一行。向数组中读入字节，直到它读入指定个数的字节或遇到一个换行符（也读入数组中）。如果在要读入最大字节数之前就到达文件结尾，则返回-1。

ServletOutputStream

全称：javax.servlet.ServletOutputStream

类型：抽象类

扩展：java.oi.OutputStream

一种向客户端发送二进制数据的输出流。通常使用ServletResponse.getOutputStream()方法得到。

方法：

Print

```
public void print(boolean b) throws IOException
```

向客户端写入布尔型值，但在结尾处没有回车换行符。

Print

```
public void print(char c) throws IOException
```

向客户端写入字符型值，但在结尾处没有回车换行符。

Print

```
public void print(double d) throws IOException
```

向客户端写入双精度型值，但在结尾处没有回车换行符。

Print

```
public void print(float f) throws IOException
```

向客户端写入浮点型值，但在结尾处没有回车换行符。

Print

```
public void print(int i) throws IOException
```

向客户端写入整型值，但在结尾处没有回车换行符。

Print

```
public void print(long l) throws IOException
```

向客户端写入长整型值，但在结尾处没有回车换行符。

Print

精彩好书见www.book118.com


```
public void print(String s) throws IOException
```

向客户端写入字符串型值，但在结尾处没有回车换行符。

Println

```
public void println() throws IOException
```

向客户端写入一个回车换行符。

Println

```
public void println(boolean b) throws IOException
```

向客户端写入布尔型值，后跟一个回车换行符。

Println

```
public void println(char c) throws IOException
```

向客户端写入字符型值，后跟一个回车换行符。

Println

```
public void println(double d) throws IOException
```

向客户端写入双精度型值，后跟一个回车换行符。

Println

```
public void println(float f) throws IOException
```

向客户端写入浮点型值，后跟一个回车换行符。

Println

```
public void println(int i) throws IOException
```

向客户端写入整型值，后跟一个回车换行符。

Println

```
public void println(long l) throws IOException
```

向客户端写入长整型值，后跟一个回车换行符。

Println

```
public void println(String s) throws IOException
```

向客户端写入字符串型值，后跟一个回车换行符。

ServletRequest

全称：javax.servlet.ServletRequest

类型：接口

表示客户端向servlet请求的一种接口。Servlet引擎创建ServletRequest对象，并将它作为参数传递给servlet的service方法。拥有获取参数名称、值、属性和输入流的方法。

方法：

精彩好书见www.book118.com

getAttribute

```
public Object getAttribute(String name)
```

返回指定属性的值，如果指定名称的属性不存在返回null。

getAttributeNames

```
public Enumeration getAttributeNames()
```

返回此请求可用的属性名称的枚举值，如果此请求不包含属性，返回空枚举值。

getCharacterEncoding

```
public String getCharacterEncoding()
```

返回在此请求体中使用的字符编码的名称，如果此请求没有指定一种字符编码返回null。

getContentLength

```
public int getContentLength()
```

返回请求体的长度，如果不知道此长度返回 – 1。在HHTP servlet中，它与CGI变量CONTENT_LENGTH的值一致。

getContentType

```
public String getContentType()
```

返回请求体的MIME类型，如果不知道此类型返回null。在HHTP servlet中，它与CGI变量CONTENT-TYPE的值一致。

getInputStream

```
public ServletInputStream getInputStream() throws IOException
```

作为二进制数据来获取请求体。可能调用getInputStream()或getReader()方法读请求体，但是不会同时调用此两种方法。

getLocale

```
public Locale getLocale()
```

在指定的情况下，返回客户端将在其中接受内容的首选Locale。否则，返回服务器的缺省地区。

getLocales

```
public Enumeration getLocales()
```

按照用户选择的顺序，返回Locale对象的枚举值，如果客户端声明没有首选的地区，返回包含一个Locale的枚举值，即服务器的缺省地区。

getParameter

```
public String getParameter(String name)
```

返回请求参数的String值，如果此参数不存在为null。

getParameterMap

精彩好书见www.book118.com


```
public Map getParameterMap()
```

返回此请求参数的java.util.Map。

getParameterNames

```
public Enumeration getParameterNames()
```

返回此请求参数名称的java.util.Enumeration。如果此请求没有参数，返回空枚举值。

getParameterValues

```
public String[] getParameterValues(String name)
```

返回包含给定请求参数的所有值的String对象数组，如果此参数不存在返回null。

getProtocol

```
public String getProtocol()
```

返回请求所使用的协议的名称和版本号。返回值的形式为protocol/majorVersion.minorVersion。在HTTP servlet中，它与CGI变量SERVER_PROTOCAL一致。

getReader

```
public BufferedReader getReader() throws IOException
```

以字符数据的形式获取请求体。可能调用getInputStream()或getReader()，但是不会同时调用此两种方法。

[不推荐使用]getRealPath

```
public String getRealPath(String path)
```

不再支持此方法。

getRemoteAddr

```
public String getRemoteAddr()
```

返回发送请求的客户端的互联网协议（IP）地址。对于HTTP servlet，它与CGI变量REMOTE_ADDR相同。

getRemoteHost

```
public String getRemoteHost()
```

返回发送请求的客户端的名称，或客户端的IP地址，如果不能确定此名称。对于HTTP servlet，它与CGI变量REMOTE_HOST相同。

getRequestDispatcher

```
public RequestDispatcher getRequestDispatcher(String path)
```

返回位于指定路径的资源的RequestDispatcher对象。此方法与ServletContext.getRequestDispatcher()方法之间的差别在于，此方法可以使用相对路径。

getScheme

```
public String getScheme()
```

精彩好书见www.book118.com

返回用于发出此请求的模式。

getServerName

```
public String getServerName()
```

返回接受请求的服务器主机名称。在HTTP servlet中，它与CGI变量SERVER_NAME相同。

getServerPort

```
public int getServerPort()
```

返回此请求所发往的端口号。在HTTP servlet中，它与CGI变量SERVER_PORT相同。

isSecure

```
public boolean isSecure()
```

如果通过安全信道（如https）发出此请求，并返回true。

removeAttribute

```
public void removeAttribute(String name)
```

从此请求中删除命名的属性。

setAttribute

```
public void setAttribute(String name, Object o)
```

以给定的名称将一个属性绑定至此请求。

setCharacterEncoding

```
public void setCharacterEncoding(String env)
throws UnsupportedEncodingException
```

指定在此请求体中使用的字符编码方式。必须在读此请求参数或输入数据被读之前调用此方法。

ServletRequestWrapper

全称：javax.servlet.ServletRequestWrapper

类型：类

实现：javax.servlet.ServletRequest

ServletRequest的一种实现，可以通过对ServletRequest类进行派生，来扩展servlet引擎的实现类。

构造器：

ServletRequestWrapper

```
public ServletRequestWrapper(ServletRequest request)
```

为指定的请求对象创建ServletRequest适配器。

方法：

getAttribute

精彩好书见www.book118.com


```
public Object getAttribute(String name)
```

返回指定属性的值，如果拥有指定名称的属性不存在返回null。

getAttributeNames

```
public Enumeration getAttributeNames()
```

返回可用于此请求的属性的名称枚举值，如果此请求不包含属性，返回空枚举值。

getCharacterEncoding

```
public String getCharacterEncoding()
```

返回用于该请求体字符编码方式的名称，如果该请求没有指定字符编码返回null。

getContentLength

```
public int getContentLength()
```

返回请求体的长度，如果不知道此长度返回 -1。在HTTP servlet中，它与CGI变量CONTENT_LENGTH相同。

getContentType

```
public String getContentType()
```

返回请求体的MIME类型，如果不知道此类型返回null。在HTTP servlet中，它与CGI变量CONTENT_TYPE相同。

getInputStream

```
public ServletInputStream getInputStream() throws IOException
```

以二进制数据的形式获取请求体。可能调用getInputStream()或getReader()方法来读此请求体，但是不可能同时调用此两种方法。

getLocale

```
public Locale getLocale()
```

在指定的情况下，返回客户端将在其中接受内容的首选Locale。否则返回服务器的缺省地区。

getLocales

```
public Enumeration getLocales()
```

按照用户选择的顺序，返回Locale对象的枚举值，如果客户端声明没有首选的地区，返回包含一个Locale的枚举值，即服务器的缺省地区。

getParameter

```
public String getParameter(String name)
```

返回请求参数的String值，如果此参数不存在则为null。

getParameterMap

```
public Map getParameterMap()
```

精彩好书见www.book118.com

返回此请求参数的java.util.Map。

getParameterNames

```
public Enumeration getParameterNames()
```

返回此请求参数名称的java.util.Enumeration。如果此请求没有参数，返回空枚举值。

getParameterValues

```
public String getParameterValues(String name)
```

返回包含给定请求参数所有值的String对象数组，如果此参数不存在，返回null。

getProtocol

```
public String getProtocol()
```

返回请求所使用协议的名称和版本号。返回值的形式为protocol/majorVersion.minorVersion。在HTTP servlet中，它与CGI变量SERVER_PROTOCOL一致。

getReader

```
public BufferedReader getReader() throws IOException
```

以字符数据的形式获取请求体。可能调用getInputStream()或getReader()，但是不会同时调用此两种方法。

getRealPath

```
public String getRealPath(String path)
```

返回getRealPath(String path)。

getRemoteAddr

```
public String getRemoteAddr()
```

返回发送请求客户端的互联网协议（IP）地址。对于HTTP servlet，它与CGI变量REMOTE_ADDR相同。

getRemoteHost

```
public String getRemoteHost()
```

返回发送请求客户端的名称，如果不能确定此名称，返回客户端的IP地址。对于HTTP servlet，它与CGI变量REMOTE_HOST相同。

getRequest

```
public ServletRequest getRequest()
```

返回被封装的请求对象。

getRequestDispatcher

```
public RequestDispatcher getRequestDispatcher(String path)
```

返回位于指定路径资源的RequestDispatcher对象。此方法与ServletContext.getRequestDispatcher()方法之间的差别在于，此方法可以使用相对路径。

精彩好书见www.book118.com

getScheme

```
public String getScheme()
```

返回用于发出此请求的模式。

getServerName

```
public String getServerName()
```

返回接受请求服务器的主机名称。在HTTP servlet中，它与CGI变量SERVER_NAME相同。

getServerPort

```
public int getServerPort()
```

返回此请求所发往的端口号。在HTTP servlet中，它与CGI变量SERVER_PORT相同。

isSecure

```
public boolean isSecure()
```

如果通过安全信道（如https）发出此请求，并返回true。

removeAttribute

```
public void removeAttribute(String name)
```

从此请求中返回属性的名称。

setAttribute

```
public void setAttribute(String name, Object o)
```

以给定的名称将一个属性绑定至此请求。

setCharacterEncoding

```
   public void setCharacterEncoding(String enc) throws
UnsupportedEncodingException
```

指定在此请求体中使用的字符编码方式。必须在读此请求参数或输入数据被读之前调用此方法。

setRequest

```
public void setRequest(ServletRequest request)
```

设置请求对象。

ServletResponse

全称：javax.servlet.ServletResponse

类型：接口

封装一个请求所产生的全部响应信息，包括响应头标、状态码和输出流。HttpServletResponse扩展了此接口，以提供相关的HTTP特性。

方法：

flushBuffer

精彩好书见www.book118.com


```
public void flushBuffer() throws IOException
```

导致将缓存内容写入客户端，从而提交响应。

getBufferSize

```
public int getBufferSize()
```

返回响应所使用的缓存的实际大小。如果缓存被关闭，则返回零。

getCharacterEncoding

```
public String getCharacterEncoding()
```

返回此响应所设置的字符编码方式的名称。

getLocale

```
public Locale getLocale()
```

返回响应所使用的地区。

getOutputStream

```
public ServletOutputStream getOutputStream()
    throws IOException
```

返回此响应的ServletOutputStream。如果已经调用了此响应的getWriter()方法，就不能再调用此方法。

getWriter

```
public PrintWriter getWriter() throws IOException
```

返回此响应的PrintWriter。如果已经调用了此响应的getOutputStream()方法，就不能再调用此方法。

isCommitted

```
public boolean isCommitted()
```

如果已提交了此响应，则返回true，它意味着响应已经写完了它的状态码与头标。

reset

```
public void reset()
```

清除响应缓存中的所有现存数据，包括状态码和头标。如果响应已经被提交，则抛出一个IllegalStateException异常。

resetBuffer

```
public void resetBuffer()
```

清除响应缓存中的所有现存数据。此方法与reset()方法的区别在于：它不清除状态码和头标，如果响应已经被提交，则抛出一个IllegalStateException异常。

setBufferSize

```
public void setBufferSize(int size)
```

设置响应体的首选缓存大小。Servlet引擎使用的缓存至少与请求的大小相同。可以使用

精彩好书见www.book118.com

getBufferSize()方法检索实际的缓存大小。必须在向缓存中写任何内容之前调用此方法，否则将抛出一个IllegalStateException异常。

setContentLength

```
public void setContentLength(int len)
```

向客户端说明向响应中写入内容的长度。

setContentType

```
public void setContentType(String type)
```

设置内容的类型。

setLocale

```
public void setLocale(Locale loc)
```

设置响应的地区。必须在getWriter()方法之前调用此方法。缺省值为服务器所使用的地区。

ServletResponseWrapper

全称：javax.servlet.ServletResponseWrapper

类型：类

实现：javax.servlet.ServletResponse

实现ServletResponse的子类的基类。其缺省行为是在servlet引擎的ServletResponse实现类中激活相应的方法。

构造器：

ServletResponseWrapper

```
public ServletResponseWrapper(ServletResponse response)
```

创建指定响应对象的ServletResponseWrapper。

方法：

flushBuffer

```
public void flushBuffer() throws IOException
```

导致将缓存内容写入客户端，从而提交响应。

getBufferSize

```
public int getBufferSize()
```

返回响应所使用缓存的实际大小。如果缓存被关闭，则返回零。

getCharacterEncoding

```
public String getCharacterEncoding()
```

返回为此响应所设置的字符编码方式的名称。

getLocale

```
public Locale getLocale()
```

精彩好书见www.book118.com

返回响应所使用的地区。

getOutputStream

```
public ServletOutputStream getOutputStream()
    throws IOException
```

返回此响应的ServletOutputStream。如果已经调用了此响应的getWriter()方法，就不能再调用此方法。

getResponse

```
public ServletResponse getResponse()
```

返回被封装的ServletResponse对象。

getWriter

```
public PrintWriter getWriter() throws IOException
```

返回此响应的PrintWriter。如果已经调用了此响应的getOutputStream()方法，就不能再调用此方法。

isCommitted

```
public boolean isCommitted()
```

如果已提交了此响应，则返回true，它意味着响应已经写完了它的状态码与头标。

reset

```
public void reset()
```

清除响应缓存中的所有现存数据，包括状态码和头标。如果响应已经被提交，则抛出一个IllegalStateException异常。

resetBuffer

```
public void resetBuffer()
```

清除响应缓存中的所有现存数据。此方法与reset()方法的区别在于：它不清除状态码和头标。如果响应已经被提交，则抛出一个IllegalStateException异常。

setBufferSize

```
public void setBufferSize(int size
```

设置响应体的首选缓存大小。Servlet引擎使用的缓存至少与请求的大小相同。可以使用getBufferSize()方法获取实际的缓存大小。必须在向缓存中写入任何内容之前调用此方法，否则将抛出一个IllegalStateException异常。

setContentLength

```
public void setContentLength(int len)
```

向客户端说明向响应中写入内容的长度。

setContentType

```
public void setContentType(String type)
```

精彩好书见www.book118.com

设置内容的类型。

setLocale

```
public void setLocale(Locale loc)
```

设置响应的地区。必须在getWriter()方法之前调用此方法。缺省值为服务器所使用的地区。

setResponse

```
public void setResponse(ServletResponse response)
```

保存被封装的Response对象的一个引用。

SingleThreadModel

全称：javax.servlet.SingleThreadModel

类型：接口

一种可以由servlet实现的接口，它向servlet引擎说明：不能并发地使用多线程访问service()方法。这确保servlet每次只处理一个请求。此方法不包含任何方法；它只不过是表明一种希望其行为的标记。

注意 *尽管它要求在其自己的service()方法之内只使用单一的servlet实例，但是，它并没有禁止多实例对外部资源的并发访问。*

方法：

SingleThreadModel没有定义任何方法；它不过是一种标记接口。

UnavailableException

全称：javax.servlet.UnavailableException

类型：类

扩展：javax.servlet.ServletException

ServletException类的一种子类，当servlet无法再处理请求时，无论是暂时的，还是永久的，都抛出此异常。

构造器：

[不推荐使用]UnavailableException

```
public UnavailableException(int seconds, Servlet servlet, String msg)
```

不再支持。

[不推荐使用]UnavailableException

```
public UnavailableException(Servlet servlet, String msg)
```

不再支持。

UnavailableException

```
public UnavailableException(String msg)
```

精彩好书见www.book118.com

用一种表示servlet永远不可使用的消息创建一个新的异常。

UnavailableException

```
public UnavailableException(String msg, int seconds)
```

用指定的错误消息为servlet创建一个新的异常，此消息说明该servlet暂时不可使用。它接受一个整数，该整数表示此servlet的预计不可使用的秒数。

方法：

[不推荐使用] getServlet

```
public Servlet getServlet()
```

不再支持。

getUnavailableSeconds

```
public int getUnavailableSeconds()
```

返回servlet的预计的不可使用时间长度，以秒为单位，或返回一个负数，如果永远不可使用或不可使用时间不可确定。

isPermanent

```
public boolean isPermanent()
```

如果servlet永远不可使用，则返回true。

Javax.servlet.http包

Cookie

全称：javax.servlet.http.Cookie

类型：类

实现：java.lang.Cloneable

cookie是一种由servlet发送给请求者的主键/值的小型集合。要求请求者（通常为Web浏览器）局域化地保存此信息，并且在下一次请求同一URL时返回它。

Servlet引擎可以使用cookie保存特定客户端的独特会话信息。这种用法对servlet作者而言是透明的。也可以分别使用HttpServletResponse.addCookie()和HttpServletRequest.getCookies()方法显式地发送和接受cookie。

注意　*用户可以拒绝接受cookie，因此应用程序应该处理这种情形。*

构造函数：

Cookie

```
public Cookie(String name, String value)
```

用指定的名称和值创建新的cookie。

方法：

精彩好书见www.book118.com

clone

```
public Object clone()
```

返回cookie的一个拷贝。

GetComment

```
public String getComment()
```

返回cookie的注释。

GetDomain

```
public String getDomain()
```

返回cookie的域名。

GetMaxAge

```
public int getMaxAge()
```

返回cookie在被删除之前应该保存的最大秒数。注意，这是相对于setMaxAge()方法被调用的时间，而不是相对于当前时间。

GetName

```
public String getName()
```

返回cookie的名称。注意，不存在setName 方法；必须在构造函数中设置cookie的名称。

GetPath

```
public String getPath()
```

返回可以找到cookie的路径 在该路径或其子目录中，对任意URL的请求都导致返回cookie。参见RFC 2109以获取有关cookie路径的详细信息。

GetSecure

```
public boolean getSecure()
```

如果用户代理（浏览器）使用一种安全性协议返回cookie，则该方法返回true。

GetValue

```
public String getValue()
```

返回cookie的值。

GetVersion

```
public int getVersion()
```

返回cookie的协议版本：

0 原始Netscape规范

1 RFC 2109规范

SetComment

```
public void setComment(String purpose)
```

精彩好书见www.book118.com

用指定的字符串设置注释字段。

SetDomain

```
public void setDomain(String pattern)
```

设置cookie的域。域可用于在特定的寻址模式中限制cookie对服务器子集的可见性。域名在保存时被转化为小写形式。如果没有指定域，只向发送它的服务器返回cookie。参见RFC 2109以获取详细信息。

SetMaxAge

```
public void setMaxAge(int expiry)
```

指定cookie应该保持的时间长度，以秒为单位。正或零值要求浏览器在指定的时间间隔之后删除cookie。负值要求浏览器只在当前浏览器实例的持续期间保持活动的。

SetPath

```
public void setPath(String uri)
```

指定cookie应该在其中可见的路径。例如，如果指定了路径/servlet/abc，那么，对任何包含该路径（如/servlet/abc/def）的URL请求返回cookie。如果没有指定路径，那么假设路径为/。路径必须包含设置cookie的servlet。请参见RFC 2109以获取有关cookie路径的详细信息。

SetSecure

```
public void setSecure(boolean flag)
```

通知用户代理（浏览器）是否使用一种安全协议返回cookie。

SetValue

```
public void setValue(String newValue)
```

设置cookie的值为指定的字符串。

SetVersion

```
public void setVersion(int v)
```

设置cookie的协议版本：

0 原始Netscape 规范

1 RFC 2109规范

HttpServlet

全称：javax.servlet.http.HttpServlet

类型：抽象类

扩展：javax.servlet.GenericServlet

实现：java.io.Serializable

运行于HTTP环境的servlet的抽象基类。HttpServlet是GenericServlet的瘦扩展，此GenericServlet为HTTP的GET、POST、PUT、DELETE、HEAD、OPTIONS和TRACE请求提供

精彩好书见www.book118.com

指定的方法。Service()方法确定HTTP请求类型并调用适当的方法。

典型的HttpServlet子类将覆盖doGet()、doPost()或此二者，但是不会覆盖service()。

构造函数：

HttpServlet

```
public HttpServlet()
```

缺省（空）构造函数。不执行任何操作。所有的初始化都应该在从GenericServlet继承的init()方法中进行。

方法：

doDelete

```
protected void doDelete(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

处理HTTP的DELETE请求。如同doPut()，此种类型的请求通常是由Web浏览器直接发起的。

doGet

```
protected void doGet(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

处理HTTP的GET请求。缺省情况下，仅返回一个表示servlet不处理GET方法的错误。覆盖doGet()的servlet。作者通常执行以下步骤：

1) 读并处理HttpServletRequest参数。

2) 在HttpServletResponse对象中，通过调用getWriter()或getOutputStream()得到一个输出流。

3) 在响应对象中设置Content-Type头标。

4) 写输出的HTML页面。

doHead

```
protected void doHead(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

处理HTTP的HEAD请求。它在功能上与GET请求类似，除了它不返回响应体，而只返回状态和头标。

doOptions

```
protected void doOptions(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

精彩好书见www.book118.com

处理HTTP的OPTIONS请求并返回HTTP服务器支持的方法列表。通常不覆盖此方法。

doPost

```
protected void doPost(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

处理HTTP的POST请求。缺省情况下，仅返回一个表示servlet不处理POST方法的错误。覆盖doPost()的servlet，作者通常执行以下步骤：

1) 读并处理HttpServletRequest参数。

2) 在HttpServletResponse对象中，通过调用getWriter()或getOutputStream()得到一个输出流。

3) 在响应对象中设置Content-Type头标。

4) 写输出的HTML页面。

doPut

```
protected void doPut(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

处理HTTP的PUT请求。通过调用请求对象的getRequestURI()方法，可以找到要写入的资源名称，而且可以从请求对象的输入流中读资源数据本身。HTML表单不支持PUT方法；这种类型的请求通常不是由Web浏览器直接发起。

doTrace

```
protected void doTrace(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

处理HTTP的TRACE请求并回送请求头标。通常不覆盖此方法。

getLastModified

```
protected long getLastModified(HttpServletRequest req)
```

返回请求对象上一次被修改的时间（单位微秒，自1970年1月1日以来的时间），或者返回－1，如果不知道此时间。缺省实现总是返回-1。

service

```
protected void service(
        HttpServletRequest req,
        HttpServletResponse resp)
    throws ServletException, IOException
```

HTTP请求的主入口点。该方法确定请求方法（GET、POST等）并向适当的处理方法（doGet()、doPost()等）分派请求。以GET方法的情形为例，它试图确定资源自上一次请求以来

精彩好书见www.book118.com

是否已经被修改。如果没有，它仅返回一个HTTP的NOT_MODIFIED状态行。通常不覆盖此方法。

service

```
public void service(
        ServletRequest req,
        ServletResponse res)
    throws ServletException, IOException
```

转换协议中立的请求至HTTP请求的便利方法，如可能，接下来调用此HTTP指定的service()方法。

HttpServletRequest

全称：javax.servlet.Http.HttpServletRequest

类型：接口

父接口：javax.servlet.ServletRequest

封装所有相关HTTP请求的信息：其参数、属性、头标和输入数据。

方法：

getAuthType

```
public String getAuthType()
```

如果服务器使用一种类似BASIC或SSL的认证模式，返回这种模式的名称，否则返回null。

getContextPath

```
public String getContextPath()
```

返回请求URI的一部分，该URI指定servlet上下文（应用）。此路径以一个“/”字符开始，但是不以“/”字符结束。

getCookies

```
public Cookie[] getCookies()
```

返回包含客户端所发送请求的全部Cookie对象的数组，如果没有发送任何Cookie返回null。

getDateHeader

```
public long getDateHeader(String name)
```

给定一个请求头标的名称，转换相应的头标值为Date对象，即返回一种长整型值（从1970年1月1日以来的时间，以微秒计）。如果不存在指定的头标，则返回-1。

getHeader

```
public String getHeader(String name)
```

返回指定请求头标的字符串值，如果在请求中没有找到此指定的头标返回null。

getHeaderNames

```
public Enumeration getHeaderNames()
```

精彩好书见www.book118.com

返回在此请求中找到的所有头标名称的枚举值。如果不存在头标，则依据servlet引擎的不同，返回null或空枚举值。

getHeaders

```
public Enumeration getHeaders(String name)
```

对于在一个请求中可能多次出现的头标，此方法将返回该头标值的一种枚举值。

getIntHeader

```
public int getIntHeader(String name)
```

给定一个请求的头标名称，转换相应的头标值为一个整数，并返回此整型值。如果指定的请求头标不存在，则返回-1。

getMethod

```
public String getMethod()
```

返回包含在请求中第一行中的HTTP方法，如GET或POST。

getPathInfo

```
public String getPathInfo()
```

返回位于servlet名称之后的请求URL的字符子串，如果不存在附加的路径信息返回null。与CGI变量PATH_INFO相同。

getPathTranslated

```
public String getPathTranslated()
```

返回位于servlet名称之后的请求URL的字符子串，此URL已经转化为实际文件系统的路径，如果不存在附加的路径信息返回null。与CGI变量PATH_TRANSLATED相同。

getQueryString

```
public String getQueryString()
```

返回位于“？”之后的请求URL字符子串，如果不存在查询字符串返回null。通常只在GET请求中查找。与CGI变量QUERY_STRING相同。

getRemoteUser

```
public String getRemoteUser()
```

如果HTTP认证有效且用户已经登录，返回用户名。否则返回null。与CGI变量REMOTE_USER相同。

getRequestedSessionId

```
public String getRequestedSessionId()
```

返回由客户端返回的会话ID的值。通常与当前会话ID的值相同，但是可能涉及过期旧会话ID的值。如果请求没有指定一个会话ID，则返回null。

getRequestURI

精彩好书见www.book118.com


```
public String getRequestURI()
```

如果存在，则返回以协议名为开头（如http://）的请求URL的字符子串，并且扩展至查询字符串（以“？”开头），但不包含查询字符串。

getRequestURL

```
public StringBuffer getRequestURL()
```

重构用于请求的整个URL。包括协议、服务器名、端口号（如果为非缺省值）和文件名。不包括查询字符串。

getServletPath

```
public String getServletPath()
```

返回这个请求的URL的一部分，此URL调用servlet。它包括servlet的名称或至servlet的路径，但是不包括任何额外的路径信息或查询字符串。与CGI变量SCRIPT_NAME的值相同。

getSession

```
public HttpSession getSession()
```

返回HttpSession.get Session(true)值的便利方法。

getSession

```
public HttpSession getSession(boolean create)
```

返回当前的HttpSession对象，或创建一个新的会话（如果参数create为true）。返回值依赖于会话是否已经存在和create参数为true还是false：

存在会话	create	返回值
false	false	null
false	true	新的会话
true	false	现存的会话
true	true	现存的会话

getUserPrincipal

```
public Principal getUserPrincipal()
```

如果已经完成了用户认证，返回用户的java.security.Principal对象。否则，此方法返回null。

isRequestedSessionIdFromCookie

```
public boolean isRequestedSessionIdFromCookie()
```

如果不是作为请求URL的一部分被发送，而是从Cookie中接受此请求的会话ID，则返回true。

[不推荐使用]isRequestedSessionIdFromUrl

```
public boolean isRequestedSessionIdFromUrl()
```

不再支持。使用isRequestedSessionIdFromURL作替换。

isRequestedSessionIdFromURL

精彩好书见www.book118.com


```
public boolean isRequestedSessionIdFromURL()
```

被请求的会话ID来自于请求URL的一部分，而不是作为Cookie进行发送，则返回true。

isRequestedSessionIdValid

```
public boolean isRequestedSessionIdValid()
```

如果请求指定一个有效、活动的会话ID，则返回true。

isUserInRole

```
public boolean isUserInRole(String role)
```

如果在发布描述器的指定逻辑"角色"中包括了被认证的用户，则返回true。

HttpServletRequestWrapper

全称：javax.servlet.http.HttpServletRequestWrapper

类型：类

扩展：javax.servlet.ServletRequestWrapper

实现：javax.servlet.http.HttpServletRequest

此类是HttpServletRequest的一种具体实现，可以用多种servlet引擎中立的方式覆盖HttpServletRequest，以便向请求对象提供附加的功能。缺省情况下，请参看servlet引擎指定的相应方法。

构造器：

HttpServletRequestWrapper

```
public HttpServletRequestWrapper(HttpServletRequest request)
```

创建一种封装给定请求的请求对象。

方法：

getAuthType

```
public String getAuthType()
```

如果服务器使用一种类似如BASIC或SSL的认证模式，返回这种模式的名称，否则返回null。

getContextPath

```
public String getContextPath()
```

返回请求URI的一部分，该URI指定servlet上下文（应用）。此路径以一个"/"字符开始，但是不以"/"字符结束。

getCookies

```
public Cookie getCookies()
```

返回包含客户端所发送请求的全部Cookie对象的数组，如果没有发送任何Cookie返回null。

getDateHeader

```
public long getDateHeader(String name)
```

精彩好书见www.book118.com

给定一个请求头标的名称，转换相应的头标值为Date对象，即返回一种长整型值（从1970年1月1日以来的时间，以微秒计）。如果不存在指定的头标，则返回-1。

getHeader

```
public String getHeader(String name)
```

返回指定请求头标的字符串值，如果在请求中没有找到此指定的头标返回null。

getHeaderNames

```
public Enumeration getHeaderNames()
```

返回在此请求中找到的所有头标名称的枚举值。如果不存在头标，则依据servlet引擎的不同，返回null或空枚举值。

getHeaders

```
public Enumeration getHeaders(String name)
```

对于在一个请求中可能多次出现的头标，此方法将返回该头标值的一种枚举值。

getIntHeader

```
public int getIntHeader(String name)
```

给定一个请求的头标名称，转换相应的头标值为一个整数，并返回此整型值。如果指定的请求头标不存在，则返回-1。

getMethod

```
public String getMethod()
```

返回包含在请求中第一行中的HTTP方法，如GET或POST。

getPathInfo

```
public String getPathInfo()
```

返回位于servlet名称之后的请求URL的字符子串，如果不存在附加的路径信息返回null。与CGI变量PATH_INFO相同。

getPathTranslated

```
public String getPathTranslated()
```

返回位于servlet名称之后的请求URL的字符子串，此URL已经转化为实际文件系统的路径，如果不存在附加的路径信息返回null。与CGI变量PATH_TRANSLATED相同。

getQueryString

```
public String getQueryString()
```

返回位于“？”之后的请求URL的字符子串，如果不存在查询字符串返回null。通常只在GET请求中查找。与CGI变量QUERY_STRING相同。

getRemoteUser

```
public String getRemoteUser()
```

精彩好书见www.book118.com

如果HTTP认证有效且用户已经登录，返回用户名。否则返回null。与CGI变量REMOTE_USER相同。

getRequestedSessionId

```
public String getRequestedSessionId()
```

返回由客户端返回的会话ID的值。通常与当前会话ID的值相同，但是可能涉及过期的旧会话ID的值。如果请求没有指定一个会话ID，则返回null。

getRequestURI

```
public String getRequestURI()
```

如果存在，则返回以协议名为开头（如http://）的请求URL的字符子串，并且扩展至查询字符串（以“?”开头），但不包含查询字符串。

getRequestURL

```
public StringBuffer getRequestURL()
```

重构用于请求的整个URL。包括协议、服务器名、端口号（如果为非缺省值）和文件名。不包括查询字符串。

getServletPath

```
public String getServletPath()
```

返回这个请求的URL的一部分，此URL调用servlet。它包括servlet的名称或至servlet的路径，但是不包括任何额外的路径信息或查询字符串。与CGI变量SCRIPT_NAME的值相同。

getSession

```
public HttpSession getSession()
```

一种返回HttpSession.getSession(true)的值的便利方法。

getSession

```
public HttpSession getSession(boolean create)
```

返回当前HttpSession对象，或创建一个新的会话（如果参数create为true）。返回值依赖于会话是否已经存在和create参数为true还是false：

存在会话	Create	返回值
False	False	null
False	True	新的会话
True	False	现存的会话
True	True	现存的会话

getUserPrincipal

```
public Principal getUserPrincipal()
```

如果已经完成了用户认证，返回用户的java.security.Principal对象。否则，此方法返回null。

精彩好书见www.book118.com

isRequestedSessionIdFromCookie

```
public boolean isRequestedSessionIdFromCookie()
```

如果不是作为请求URL的一部分而得以发送，而是从Cookie中接受此请求会话ID，则返回true。

isRequestedSessionIdFromUrl

```
public boolean isRequestedSessionIdFromUrl()
```

关于被封装的请求对象，返回该对象不被支持的isRequestedSessionIdFromUrl()方法的值。

isRequestedSessionIdFromURL

```
public boolean isRequestedSessionIdFromURL()
```

被请求的会话ID来自于请求URL的一部分，与Cookie进行发送的相反，则返回true。

isRequestedSessionIdValid

```
public boolean isRequestedSessionIdValid()
```

如果请求指定一个有效、活动的会话ID，则返回true。

isUserInRole

```
public boolean isUserInRole(String role
```

如果在发布描述器的指定逻辑“角色”中包括了被认证的用户，则返回true。

HttpServletResponse

全称：javax.servlet.hhtp.HttpServletResponse

类型：接口

父接口：javax.servlet.ServletResponse

封装一个HTTP请求所产生有关响应的全部信息，包括响应头标、状态码和输出流。

方法：

addCookie

```
public void addCookie(Cookie cookie)
```

为指定的Cookie写一个Set-Cookie头标。

addDateHeader

```
public void addDateHeader(String name, long date)
```

为一个可以拥有多个值的HTTP头标写入日期头标。

addHeader

```
public void addHeader(String name, String value)
```

为一个可以拥有多个值的HTTP头标写入通用头标。

addIntHeader

```
public void addIntHeader(String name, int value)
```

精彩好书见www.book118.com

为一个可以拥有多个值的HTTP头标写入整数头标。

containsHeader

```
public boolean containsHeader(String name)
```

如果响应已经包含一种拥有指定名称的头标，则返回true。

[不推荐使用]encodeRedirectUrl

```
public String encodeRedirectUrl(String url)
```

不再支持。

encodeRedirectURL

```
public String encodeRedirectURL(String url)
```

通过在计划用于sendRedirect()的URL中随意地追加经编码的会话ID作为参数，以支持会话跟踪。如果客户端支持cookie，则没有必要这样做。Servlet引擎进行此项决策；对需要经由此方法完成输出的URL进行过滤往往是安全的。

[不推荐使用]encodeUrl

```
public String encodeUrl(String url)
```

不再支持。

encodeURL

```
public String encodeURL(String url)
```

如果需要，通过在指定的URL中随意地追加经编码的会话ID作为参数，以支持会话跟踪。如果客户端支持cookie，则没有必要这样做。Servlet引擎进行此项决策；对需要由此方法完成输出的URLs进行过滤往往是安全的。

sendError

```
public void sendError(int sc) throws IOException
```

设置HTTP状态码为指定值。在调用此方法之后，响应对象被提交；向此响应对象的任何进一步输入都无效。

sendError

```
public void sendError(int sc, String msg) throws IOException
```

设置HTTP状态码为指定值且设置状态消息。在调用此方法之后，响应对象被提交；向此响应对象的任何进一步输入都无效。

sendRedirect

```
public void sendRedirect(String location) throws IOException
```

设置HTTP状态码为302（暂时性传递），且用指定值写一个Location头标。通常用户引擎（Web浏览器）将解析此响应并自动请求新的URL。

setDateHeader

精彩好书见www.book118.com


```
public void setDateHeader(String name, long date)
```

用指定的名称和正确格式化的日期值写一个响应头标。

setHeader

```
public void setHeader(String name, String value)
```

用指定的名称和值写一个响应头标。

setIntHeader

```
public void setIntHeader(String name, int value)
```

用指定的名称和一个整型值写一个响应头标。

setStatus

```
public void setStatus(int sc)
```

设置此响应的状态码。

[不推荐使用]setStatus

```
public void setStatus(int sc, String sm)
```

不再支持。

HttpServletResponseWrapper

全称：javax.servlet.http.HttpServletResponseWrapper

类型：类

扩展：javax.servlet.ServletResponseWrapper

实现：javax.servlet.http.HttpServletResponse

此类是HttpServletResponse的一种具体实现，可以扩展HttpServletResponse以便允许定制响应对象。缺省情况下，关于该类中的方法，请参考servlet引擎的实现类中的对应方法。

构造器：

HttpServletResponseWrapper

```
public HttpServletResponseWrapper(HttpServletResponse response)
```

创建一个封装指定响应的响应适配器。

方法：

addCookie

```
public void addCookie(Cookie cookie)
```

为指定的Cookie写一个Set-Cookie头标。

addDateHeader

```
public void addDateHeader(String name, long date)
```

为一个可以拥有多个值的HTTP头标写入日期头标。

addHeader

精彩好书见www.book118.com


```
public void addHeader(String name, String value)
```

为一个可以拥有多个值的HTTP头标写入通用头标。

addIntHeader

```
public void addIntHeader(String name, int value)
```

为一个可以拥有多个值的HTTP头标写入整数头标。

containsHeader

```
public boolean containsHeader(String name)
```

如果响应已经包含一种拥有指定名称的头标，则返回true。

encodeRedirectUrl

```
public String encodeRedirectUrl(String url)
```

在servlet引擎指定的类中调用不被支持的encodeRedirectUrl()方法。

encodeRedirectURL

```
public String encodeRedirectURL(String url)
```

通过在计划用于sendRedirect()的URL中随意地追加经编码的会话ID作为参数，以支持会话跟踪。如果客户端支持cookie，则没有必要这样做。Servlet引擎进行此项决策；对需要经由此方法完成输出的URL进行过滤往往是安全的。

encodeUrl

```
public String encodeUrl(String url)
```

在servlet引擎指定的类中调用不被支持的encodeUrl()(string url)方法。

encodeURL

```
public String encodeURL(String url)
```

如果需要，通过在指定的URL中随意地追加经编码的会话ID作为参数，以支持会话跟踪。如果客户端支持cookie，则没有必要这样做。Servlet引擎进行此项决策；对需要由此方法完成输出的URLs进行过滤往往是安全的。

sendError

```
public void sendError(int sc) throws IOException
```

设置HTTP状态码为指定值。在调用此方法之后，响应对象被提交；向此响应对象的任何进一步输入都无效。

sendError

```
public void sendError(int sc, String msg) throws IOException
```

设置HTTP状态码为指定值且设置状态消息。在调用此方法之后，响应对象被提交；向此响应对象的任何进一步输入都无效。

sendRedirect

精彩好书见www.book118.com


```
public void sendRedirect(String location) throws IOException
```

设置HTTP状态码为302（暂时性传递），且用指定值写一个Location头标。通常用户引擎（Web浏览器）将解析此响应并自动请求新的URL。

setDateHeader

```
public void setDateHeader(String name, long date)
```

用指定的名称和正确格式化的日期值写一个响应头标。

setHeader

```
public void setHeader(String name, String value)
```

用指定的名称和值写一个响应头标。

setIntHeader

```
public void setIntHeader(String name, int value)
```

用指定的名称和一个整型值写一个响应头标。

setStatus

```
public void setStatus(int sc)
```

设置此响应的状态码。

setStatus

```
public void setStatus(int sc, String sm)
```

在servlet引擎指定的类中调用不被支持的setStatus(int sc,String sm)方法。

HttpSession

全称：javax.servlet.http.HttpSession

类型：接口

HttpSession是对象的一种用名称引用仓库，此对象属于用户浏览器会话。在用户请求之间，这种仓库在服务器端仍然保持有效。每个会话拥有一个由服务器分派的惟一会话ID，客户端对此服务器保持跟踪并用后续的请求进行回传。

通过调用HttpServletRequest.getSession(true)或HttpServletRequest.getSession()方法创建会话。然后会话ID通过cookie或作为生成的URL的一个参数被传递给客户端。此会话一直被看作是“新”的，直至客户端与之连接，直到客户端在后续请求中回传会话ID。isNew()方法可以用于确定这一点。

对象都是使用setAttribute()方法保存于会话中，并且可以使用getAttribute()方法检索。如果会话中的一个对象实现了HttpSessionBindingListener接口，无论它是被绑定至一会话，还是从一会话中被解除绑定，都会得到通知。

方法：

getAttribute

精彩好书见www.book118.com


```
public Object getAttribute(String name)
```

如果会话中存在此对象，则返回拥有指定名称的对象。如果此对象不存在返回null。

getAttributeNames

```
public Enumeration getAttributeNames()
```

返回被绑定至此会话的所有对象的名称枚举值。

getCreationTime

```
public long getCreationTime()
```

返回创建会话的时间，自1970年1月1日以来的时间，以微秒计。

getId

```
public String getId()
```

返回会话的标识符。

getLastAccessedTime

```
public long getLastAccessedTime()
```

返回会话上次被访问的时间，自1970年1月1日以来的时间，以微秒计。

getMaxInactiveInterval

```
public int getMaxInactiveInterval()
```

返回此会话在请求之间可以保持为活动的最大秒数。如果超过此时间间隔，允许servlet引擎中止该会话。

注意 某些servlet引擎错误地将此值当作微秒或分钟数。在其正确的情况下，应该检查此方法的操作。

[不推荐使用]getSessionContext

```
public HttpSessionContext getSessionContext()
```

不再支持。

[不推荐使用]getValue

```
public Object getValue(String name)
```

不再支持。请使用getAttribute(String name)方法代替。

[不推荐使用]getValueNames

```
public String getValueNames()
```

不再支持。请使用getAttributenames()方法代替。

invalidate

```
public void invalidate()
```

关闭会话，调用任何绑定至此会话的HttpSessionBindingListener对象的valueUnbound()方法。

精彩好书见www.book118.com

isNew

```
public boolean isNew()
```

如果已经创建了一个会话，但客户端还没有提出一个拥有此会话ID的请求，则返回true。

[不推荐使用]putValue

```
public void putValue(String name, Object value)
```

不再支持。请使用setAttribute(String name, Object value)方法代替。

removeAttribute

```
public void removeAttribute(String name)
```

在会话中删除一个拥有指定名称的对象引用。如果此对象实现了HttpSesseionBindingListener接口，则servlet引擎调用其valueUnbound()方法。如果在此会话中不存在指定的值，则取消上述删除操作。

[不推荐使用]removeValue

```
public void removeValue(String name)
```

不再支持。请使用removeAttribute(String name)方法代替。

setAttribute

```
public void setAttribute(String name, Object value)
```

在会话中以指定的名称保存一个对象引用。如果此对象实现了HttpSessionBindingListener接口，则servlet引擎调用其valueBound()方法。

setMaxInactiveInterval

```
public void setMaxInactiveInterval(int interval)
```

指定此会话可以在请求之间保持有效的最大秒数。如果超过此时间间隔，允许servlet引擎中止该会话。

注意 *某些servlet引擎错误地将此值当作微秒或分钟数。在其正确的情况下，应该检查此方法的操作。*

HttpSessionActivationListener

全称：javax.servlet.http.HttpSessionActivationListener

类型：接口

通过实现该接口，对象可以得到注册，以便接收激活有关会话和被动化事件的通知。

方法：

sessionDidActivate

```
public void sessionDidActivate(HttpSessionEvent se)
```

一旦会话被激活，将调用此方法。

精彩好书见www.book118.com

sessionWillPassivate

```
public void sessionWillPassivate(HttpSessionEvent se)
```

当会话将要被动化时，将调用此方法。

HttpSessionAttributesListener

全称：javax.servlet.http.HttpSessionAttributesListener
类型：接口
父接口：java.util.EventListener
通过实现此接口，对象可以注册接收添加或删除属性的事件通知。
方法：

attributeAdded

```
public void attributeAdded(HttpSessionBindingEvent se)
```

当已经向会话中添加了一个属性时，将调用此方法。

attributeRemoved

```
public void attributeRemoved(HttpSessionBindingEvent se)
```

当已经从会话中删除了一个属性时，将调用此方法。

attributeReplaced

```
public void attributeReplaced(HttpSessionBindingEvent se)
```

当在会话中已经更换了一个属性时，将调用此方法。

HttpSessionBindingEvent

全称：javax.servlet.http.HttpSessionBindingEvent
类型：类
扩展：javax.servlet.http.HttpSessionEvent

被作为参数传递给HttpSessionBindingListener的方法valueBound()和valueUnbound()的一种事件对象。通过使用该事件的方法，HttpSessionBindingListener可以获取它被绑定的名称和对HttpSession本身的一个引用。

构造器：

HttpSessionBindingEvent

```
public HttpSessionBindingEvent
   (HttpSession session, String name)
```

为指定的会话创建一个新的HttpSessionBindingEvent对象。其中的name参数表示此会话所绑定的侦听对象的名称。

HttpSessionBindingEvent

```
public HttpSessionBindingEvent
```

精彩好书见www.book118.com


```
(HttpSession session, String name, Object value)
```

为指定的会话创建一个新的HttpSessionBindingEvent对象。其中的name参数表示此会话所绑定的侦听对象的名称，value参数则包含名称的值。

方法：

getName

```
public String getName()
```

返回会话所了解该对象的名称。

getSeesion

```
public HttpSession getSession()
```

返回侦听者对象所绑定或解除绑定的会话。

getValue

```
public Object getValue()
```

返回被添加、修改或删除属性的名称。

HttpSessionBindingListener

全称：javax.servlet.http.HttpSessionBindingListener

类型：接口

父接口：java.util.EventListener

实现此种接口的对象在它们被绑定至HttpSession或从HttpSession解除绑定时会得到通知。此对象必须提供valueBound()和valueUnbound()方法，它们都包含一个HttpSessionBindingEvent参数，该参数允许此对象确定其名称和它所属于的会话。

方法：

valueBound

```
public void valueBound(HttpSessionBindingEvent event)
```

当一个对象被绑定至会话时，将调用此方法。

valueUnbound

```
public void valueUnbound(HttpSessionBindingEvent event)
```

当从会话中解除一个对象的绑定时，将调用此方法。

HttpSessionContext

全称：javax.servlet.http.HttpSessionContext

类型：接口

以前用于servlet间的直接通信。现在不推荐使用该类。

方法：

[不推荐使用]getIds

精彩好书见www.book118.com


```
public Enumeration getIds()
```

不再支持。

[不推荐使用]getSession

```
public HttpSession getSession(String sessionId)
```

不再支持。

HttpSessionEvent

全称：javax.servlet.http.HttpSessionEvent
类型：类
扩展：java.util.EventObject
在Web应用程序中表示改变会话的事件通知。
构造器：

HttpSessionEvent

```
public HttpSessionEvent(HttpSession source)
```

从指定的源创建一个新的会话事件。
方法：

getSession

```
public HttpSession getSession()
```

返回一个被修改会话的引用。

HttpSessionListener

全称：javax.servlet.http.HttpSessionListener
类型：接口
当创建会话或会话无效化时，实现HttpSessionListener的类会接收到通知。
方法：

sessionCreated

```
public void sessionCreated(HttpSessionEvent se)
```

当创建一个新的会话时，调用此方法。

SessionDestroyed

```
public void sessionDestroyed(HttpSessionEvent se)
```

当无效化一个会话时，调用此方法。

HttpUtils

全称：javax.servlet.http.HttpUtils

精彩好书见www.book118.com

类型：类

一种提供用于HTTP servlet方法的工具类。

构造器：

HttpUtils

```
public HttpUtils()
```

创建一个新的HttpUtils对象。

方法：

getRequestURL

```
public static StringBuffer getRequestURL
    (HttpServletRequest req)
```

返回用于指定请求的完整URL。包括协议、服务器名称、端口号（如果为非缺省端口）和文件名。不包含请求字符串。

ParsePostData

```
public static Hashtable parsePostData
(int len, ServletInputStream in)
```

读指定长度的servlet请求输入流，并且通过调用parseQueryString()方法将其解析为主键/取值对。

ParseQueryString

```
public static Hashtable parseQueryString(String s)
```

给定一个包含编码的URL参数和值的请求字符串。返回一个哈希表包括解析的名字和值。在此哈希表中，参数名为主键，相应的值则为字符串数组。如果此参数只出现一次，则数组的长度为1，否则，在数组中存在多个条目。参见java.net.URLEncoder以了解编码是如何实现的。

精彩好书见www.book118.com

附录B JSP API 版本1.2

此附录描述位于如下两个JSP包的每个类：

- javax.servlet.jsp JavaServer Page 基类。
- javax.servlet.jsp.tagext JSP定制标记。

对于每个类，包括如下小节：

- 类名。
- 上下文（全称、类型、超类和实现的接口）。
- 类描述。
- 类中每个方法的详细信息。

注意 这里描述的类和方法是基于JSP 1.2 规范的最终公开草案。尽管此最终草案非常接近于官方的规范，但是，仍然可能被修改。如果有任何疑问，请参考此规范的最新版本，位于http://java.sun.com/products/jsp。

Javax.servlet.jsp包

HttpJspPage

全称：javax.servlet.jsp.HttpJspPage

类型：接口

父接口 javax.servelet.jsp.JspPage

JspPage的一种子接口，此JspPage是由JSP引擎生成的HTTP指定类所实现的。JSP引擎将自动创建此页面包含所定义的全部scriptlet代码的一种_jspService()方法。JSP作者不应该覆盖此方法。

方法：

_jspService

```
public void _jspService(
        HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, IOException
```

JSP页面的体。自从其由JSP容器生成的servlet代码所定义以来，JSP作者必须不定义此方法。_jspService正是运行scriptlet和产生HTML模板输出的地方。

JspEngineInfo

全称：javax.servelt.jsp.JspEngineInfo

精彩好书见www.book118.com

类型：抽象类

一种提供有关JSP引擎信息的类。JspFactory.getEngineInfo()方法返回此类的一个实例。

注意 此类主要是设计为JSP引擎开发人员所使用。

构造器：

JspEngineInfo

```
public JspEngineInfo()
```

创建一个新的JspEngineInfo对象。

方法：

getSpecificationVersion

```
public abstract String getSpecificationVersion()
```

返回由JSP引擎所支持的JSP规范版本。

JspException

全称：javax.servlet.jsp.JspException

类型：类

扩展：java.lang.Exception

通用的JSP异常基类。定制标签类的许多方法会抛出该异常。

构造器：

JspException

```
public JspException()
```

创建一个新JspException，不包含相关的错误消息。

JspException

```
public JspException(String msg)
```

使用指定的消息创建一个新的JspException。

JspException

```
public JspException(String message, Throwable rootCause)
```

用指定的消息创建一个新的JspException，并且将指定的根异常与之相关联。

JspException

```
public JspException(Throwable rootCause)
```

创建一个与指定的根异常相关的新JspException。

方法：

getRootCause

```
public Throwable getRootCause()
```

精彩好书见www.book118.com

返回引起此JspException的异常。

JspFactory

全称：javax.servlet.jsp.JspFactory

类型：抽象类

一种提供工厂方法的类，用于创建支持JSP环境所必需的对象。包括分配缺省JspFactory的静态方法。

注意　该类主要是设计为JSP引擎开发人员所使用。

构造器：

JspFactory

```
public JspFactory()
```

创建一个新的JspFactory对象。

方法：

getDefaultFactory

```
public static synchronized JspFactory getDefaultFactory()
```

返回当前注册的JspFactory对象。

getEngineInfo

```
public abstract JspEngineInfo getEngineInfo()
```

返回此JSP实现的JspEngineInfo对象。

getPageContext

```
public abstract PageContext getPageContext(
    Servlet servlet,
    ServletRequest request,
    ServletResponse response,
    String errorPageURL,
    boolean needsSession,
    int buffer,
    boolean autoflush)
```

返回PageContext对象。调用此方法导致PageContext.initialize()方法被激活，并导致下列属性被设置：

- 请求的servlet。
- 请求servlet的ServletConfig。
- ServletRequest对象。
- ServletResponse对象。
- JSP错误页面的URL，如果曾指定了一个此种页面。
- JSP是否需要HTTP会话。

精彩好书见www.book118.com

- 缓存的大小。
- 缓存在溢出时是否应该自动被刷新。

当调用releasePageContext()方法时，这些资源得以释放。

注意 对该方法的调用应该由JSP引擎自动产生，而不应该由JSP作者编制。

releasePageContext

```
public abstract void releasePageContext(PageContext pc)
```

释放PageContext，包括在调用getPageContext()方法时获得的任何资源。

注意 对该方法的调用应该由JSP引擎自动产生，而不应该由JSP作者编制。

setDefaultFactory

```
public static synchronized void setDefaultFactory(JspFactory factory)
```

设置缺省的JspFactory对象。应该只由JSP引擎本身调用。

JspPage

全称：javax.servlet.jsp.JspPage

类型：接口

父接口：javax.servlet.Servlet

一个由JSP引擎生成的类所实现的Servlet子接口。jspInit()和jspDestroy()方法可以由JSP作者进行覆盖，以便实现Servlet的init()和destroy()方法所完成的工作。

方法：

jspDestroy

```
public void jspDestroy()
```

当生成的JSP servlet被销毁时，激活的一种方法。如果被用到，必须在JSP声明中定义它。应该覆盖此方法以替代destroy()方法。

jspInit

```
public void jspInit()
```

当生成的JSP servlet被初始化时，激活的一种方法。如果被用到，必须在JSP声明中定义它。应该覆盖此方法以替代init()方法。

JspTagException

全称：javax.servlet.jsp.JspTagException

类型：类

扩展：javax.servlet.jsp.JspException

JspException的一个子接口，在标签处理器中用于表示一种致命错误。

构造器：

精彩好书见www.book118.com

JspTagException

```
public JspTagException()
```

不使用任何相关消息创建一个新的JspTagException。

JspTagException

```
public JspTagException(String msg)
```

使用指定的消息创建一个新的JspTagException。

方法：

JspWriter

全称：javax.servlet.jsp.JspWriter

类型：抽象类

扩展：java.io.Writer

用于写JSP输出的java.io.Writer的一种子类。其作用基本上等同于java.io.PrintWriter。通过调用底层servlet的getWriter()方法，该类由生成的_jspService()方法进行实例化，这导致对getOutputStream()方法的调用为非法。

隐式变量out就是该类的一个实例。

方法：

clear

```
public abstract void clear() throws IOException
```

清除页面缓存。如果缓存已经被清除，则抛出一个IOException（例如全部的缓存数据已经写入输出流）。

clearBuffer

```
public abstract void clearBuffer() throws IOException
```

清除页面缓存。不抛出一个IOException。

close

```
public abstract void close() throws IOException
```

刷新并关闭流。

flush

```
public abstract void flush() throws IOException
```

刷新输出流。

getBufferSize

```
public int getBufferSize()
```

返回实际使用的缓存大小。

getRemaining

精彩好书见www.book118.com


```
public abstract int getRemaining()
```

返回缓存中没有使用的字节数。

isAutoFlush

```
public boolean isAutoFlush()
```

返回一种表示JSP的autoFlush标志是否设置的标记。

newLine

```
public abstract void newLine() throws IOException
```

输出系统的line.separator字符串。

Print

```
public abstract void print(boolean b) throws IOException
```

打印一个布尔型值。

Print

```
public abstract void print(char c) throws IOException
```

打印一个字符型值。

Print

```
public abstract void print(char[] s) throws IOException
```

打印一个字符数组。

Print

```
public abstract void print(double d) throws IOException
```

打印一个双精度的浮点数。

Print

```
public abstract void print(float f) throws IOException
```

打印一个单精度浮点数。

Print

```
public abstract void print(int i) throws IOException
```

打印一个整型值。

Print

```
public abstract void print(long l) throws IOException
```

打印一个长整型值。

Print

```
public abstract void print(Object obj) throws IOException
```

使用其toString()方法打印一个对象。

Print

精彩好书见www.book118.com


```
public abstract void print(String s) throws IOException
```

打印一个字符串。

Println

```
public abstract void println() throws IOException
```

打印系统的line.separator字符。

Println

```
public abstract void println(boolean x) throws IOException
```

打印一个布尔型值，其后跟一个换行符。

Println

```
public abstract void println(char x) throws IOException
```

打印一个字符值，其后跟一个换行符。

Println

```
public abstract void println(char[] x) throws IOException
```

打印一个字符数组，其后跟一个换行符。

Println

```
public abstract void println(double x) throws IOException
```

打印一个双精度浮点数，其后跟一个换行符。

Println

```
public abstract void println(float x) throws IOException
```

打印单精度浮点数，其后跟一个换行符。

Println

```
public abstract void println(int x) throws IOException
```

打印一个整型数，其后跟一个换行符。

Println

```
public abstract void println(long x) throws IOException
```

打印一个长整型数，其后跟一个换行符。

Println

```
public abstract void println(Object x) throws IOException
```

打印一个对象，其后跟一个换行符。

Println

```
public abstract void println(String x) throws IOException
```

打印一个字符串，其后跟一个换行符。

精彩好书见www.book118.com

PageContext

全称：javax.servlet.jsp.PageContext

类型：抽象类

PageContext是一个包装对象，封装了处理请求的JSP单一调用的全部细节。它包括初始化和释放会话、输出、请求和响应对象的方法。它还在各种JSP访问的名空间中提供设置与获取属性的方法。

在其getPageContext()方法被调用时，PageContext对象由JSPFactory进行创建并初始化，当其releasePageContext()方法被调用时，得以释放。这两种方法调用都是由JSP引擎生成的代码自动执行的。

构造器：

PageContext

```
public PageContext()
```

创建一个新的PageContext对象。

方法：

findAttribute

```
public abstract Object findAttribute(String name)
```

在页面、请求、会话和应用程序范围（按该顺序）搜索指定的属性，返回第一个配匹的值。如果在任何位置都不存在此属性，则返回null。

forward

```
public abstract void forward(String relativeUrlPath)
throws ServletException, IOException
```

调用与此servlet的RequestDispatcher相关的forward()方法。参见javax.servlet.RequestDispatcher以了解详细信息。

getAttribute

```
public abstract Object getAttribute(String name)
```

返回位于页面范围的指定属性，如果此属性不存在返回null。

getAttribute

```
public abstract Object getAttribute(String name, int scope)
```

返回位于确定范围的指定属性，如果此属性不存在返回null。范围由以下常数确定：

```
PageContext.PAGE_SCOPE
PageContext.REQUEST_SCOPE
PageContext.SESSION_SCOPE
PageContext.APPLICATION_SCOPE
```

getAttributeNamesInScope

```
public abstract Enumeration
```

精彩好书见www.book118.com


```
getAttributeNamesInScope(int scope)
```

返回位于指定范围的一个属性名称的枚举值。参见getAttribute(String name, int scope)以了解范围值的列表。

getAttributesScope

```
public abstract int getAttributesScope(String name)
```

返回指定名称的第一个属性的范围。参见getAttribute(String name, int scope)以了解范围值的列表。

getException

```
public abstract Exception getException()
```

返回被传递给ErrorPage的Exception对象。

getOut

```
public abstract JspWriter getOut()
```

返回此响应的JspWriter。

getPage

```
public abstract Object getPage()
```

返回与此PageContext相关的servlet。

getRequest

```
public abstract ServletRequest getRequest()
```

返回与此PageContext相关的ServletRequest。

getResponse

```
public abstract ServletResponse getResponse()
```

返回与此PageContext相关的ServletResponse。

getServletConfig

```
public abstract ServletConfig getServletConfig()
```

返回与此PageContext相关的ServletConfig。

getServletContext

```
public abstract ServletContext getServletContext()
```

返回与此PageContext相关的ServletContext。

getSession

```
public abstract HttpSession getSession()
```

返回此请求的HttpSession，如果会话不存在返回null。

handlePageException

```
public abstract void handlePageException(Exception e)
```

精彩好书见www.book118.com


```
    throws ServletException, IOException;
```

用于处理一种由当前页面抛出未被处理的异常。如果存在一个活动的异常，则调用此方法。

注意 相比之下，handlePageException(Throwable t)方法更通用，因此应该用来替代此方法。

handlePageException

```
public abstract void handlePageException(Throwable t)
    throws ServletException, IOException
```

处理一种由当前页面抛出未被处理的异常。如果存在一个活动的异常，则调用此方法。

include

```
public abstract void include(String relativeUrlPath)
throws ServletException, IOException
```

调用与此servlet的RequestDispatcher相关的include()方法。请参见javax.servlet.RequestDispatcher以了解更多信息。

initialize

```
public abstract void initialize(
    Servlet servlet,
    ServletRequest request,
    ServletResponse response,
    String errorPageURL,
    boolean needsSession,
    int bufferSize,
    boolean autoFlush)
throws IOException, IllegalStateException,
    IllegalArgumentException
```

保存servlet、request、response、errorPageURL、needsSession、bufferSize和autoFlush属性，并将适当的隐式变量提供给JSP。此方法由getPageContext()方法调用，而不应该由JSP作者直接调用。

popBody

```
public JspWriter popBody()
```

恢复此前pushBody()方法所保存的JspWriter，并且更新隐式变量out和PageContext的“输出”属性的值。

pushBody

```
public BodyContent pushBody()
```

保存当前的JspWriter，并且创建新的BodyContent对象，让它成为PageContext的“输出”属性和隐式变量out的值。

release

精彩好书见www.book118.com


```
public abstract void release()
```

执行与initialize相反的工作，释放PageContext及其需要的资源。该方法由releasePageContext()方法调用，而不应该由JSP作者直接调用。

removeAttribute

```
public abstract void removeAttribute(String name)
```

在页面、请求、会话和应用程序的范围（按该顺序）搜索指定的属性，并且删除第一个配匹的属性。

removeAttribute

```
public abstract void removeAttribute(String name, int scope)
```

删除与指定的名称和范围相关的属性。范围选择由下列常数确定：

```
PageContext.PAGE_SCOPE
PageContext.REQUEST_SCOPE
PageContext.SESSION_SCOPE
PageContext.APPLICATION_SCOPE
```

setAttribute

```
public abstract void setAttribute
   (String name, Object attribute)
```

使用可以在其他范围中设置属性的页面scope.setAttribute(String name, object o, int scope)来设置指定的属性。

setAttribute

```
public abstract void setAttribute
   (String name, Object o, int scope)
```

使用指定的名称和范围设置属性。范围选择由下列常数确定：

```
PageContext.PAGE_SCOPE
PageContext.REQUEST_SCOPE
PageContext.SESSION_SCOPE
PageContext.APPLICATION_SCOPE
```

Javax.servlet.jsp.tagext包

BodyContent

全称：javax.servlet.jsp.tagext.BodyContent

类型：抽象类

扩展：javax.servlet.jsp.JspWriter

BodyContent是javax.servlet.jsp.JspWriter的子类，与其超类的差异在于它的内容没有自动地输出至servlet的输出流。相反，这些内容被累积至一个字符串缓存中。在完成标签体之后，原始

精彩好书见www.book118.com

的JspWriter被保存，但是，在bodyContent变量的doEndTag()方法中，BodyContent对象仍然可见。可以使用其getString()或getReader()方法获取该对象的内容，可以按照需要进行修改，并且写入被保存的JspWriter输出流，以合并至页面输出。

方法：

clearBody

```
public void clearBody()
```

重置BodyContent缓存为空。如果在doAfterBody()方法中标签体被写入闭合的写入者，则该方法可能很有用。

flush

```
public void flush() throws IOException
```

覆盖JspWriter.flush()方法，以便它总是抛出一个异常。刷新一个BodyContent写入者是无效的，因为它并没有与一个可以写入的输出流连接。

getEnclosingWriter

```
public JspWriter getEnclosingWriter()
```

返回位于堆栈中下一个较高位的写入者对象（可能是另一个BodyContent）。

getReader

```
public abstract Reader getReader()
```

在体内容已经评估之后，返回此体内容的一个读程序。该读程序可以传递给能处理java.io.Reader的其他类，如StreamTokenizer、FilterReader或XML解析器。

getString

```
public abstract String getString()
```

在体内容评估之后，返回包含此体内容的字符串。

writeOut

```
public abstract void writeOut(Writer out) throws IOException
```

向指定的输出写入者写体内容。

BodyTag

全称：javax.servlet.jsp.tagext.BodyTag

类型：接口

父接口：javax.servlet.jsp.tagext.IterationTag

IterationTag接口的一种扩展，它包含必须与体处理一同实现的新方法。

方法：

doInitBody

```
public void doInitBody() throws JspException
```

精彩好书见www.book118.com

一种生命周期方法，在setBodyContent()方法之后，且刚好在体评估之前被调用。如果体需要多次评估，但只调用此方法一次。

setBodyContent

```
public void setBodyContent(BodyContent b)
```

在当前JspWriter已经被“推出”且创建了一种新的BodyContent写入者之后，由JSP servlet激活。此过程刚好发生在doStartTag()之后。

BodyTagSupport

全称：javax.servlet.jsp.tagext.BodyTagSupport

类型：类

扩展：javax.servlet.jsp.tagext.TagSupport

实现：javax.servlet.jsp.tagext.BodyTag

一种实现了BodyTag全部方法的有用基类。标签处理器可以扩展BodyTagSupport，并且只覆盖那些需要修改的方法。

构造器：

BodyTagSupport

```
public BodyTagSupport()
```

创建一个新的BodyTagSupport对象。

方法：

doAfterBody

```
public int doAfterBody() throws JspException
```

在每个体评估的结尾处激活此方法。缺省情况下返回Tag.SKIP_BODY。如果覆盖此方法，则应该返回Tag.SKIP_BODY或IterationTag.EVAL_BODY_AGAIN。

doEndTag

```
public int doEndTag() throws JspException
```

在定制标签的结尾处激活此方法。缺省返回EVAL_PAGE，也可以覆盖此方法，以返回SKIP_PAGE。

doInitBody

```
public void doInitBody() throws JspException
```

一种生命期方法，在setBodyContent()方法之后，且刚好在体评估之前被调用。如果体需要多次评估，但此方法只调用一次。缺省地，此实现不执行任何操作。

doStartTag

```
public int doStartTag() throws JspException
```

请参见位于Tag接口的doStartTag()方法。此实现返回EVAL_BODY_BUFFERED。

精彩好书见www.book118.com

getBodyContent

```
public BodyContent getBodyContent()
```

返回当前的BodyContent对象。

getPreviousOut

```
public JspWriter getPreviousOut()
```

返回环境JspWriter。

release

```
public void release()
```

释放标签处理器的状态。

setBodyContent

```
public void setBodyContent(BodyContent b)
```

保存BodyContent对象的一个引用。

IterationTag

全称：javax.servlet.jsp.tagext.IterationTag

类型：接口

父接口：javax.servlet.jsp.tagext.Tag

Tag接口的一种扩展，它定义了标签体重复评估的语法。

方法：

doAfterBody

```
public int doAfterBody() throws JspException
```

一种生命期方法，在体已经评估之后，且BodyContent写入者仍为活动状态时被调用。此方法必须返回EVAL_AGAIN或返回SKIP_BODY。如果返回代码为EVAL_BODY_AGAIN，则体被重新评估且再次调用doAfterBody()方法。

PageData

全称：javax.servlet.jsp.tagext.PageData

类型：抽象类

一种可以在TLD中作为JSP页面的验证程序而列表显示的类。它提供一种与JSP页面对应的XML文档的阅读方法。

构造器：

PageData

```
public PageData()
```

创建一个新的PageData对象。

精彩好书见www.book118.com

方法：

getInputStream

```
public abstract InputStream getInputStream()
```

作为XML文档返回该JSP页面。

Tag

全称：javax.servlet.jsp.tagext.Tag
类型：接口
一组必须由定制标签处理器实现的生命期方法。
方法：

goEndTag

```
public int doEndTag() throws JspException
```

当遇到结束标签时调用此方法。返回码表示JSP实现servlet是否应该处理余下的页面，即EVAL_PAGE或SKIP_PAGE。此方法可以抛出一个表示致命性错误的JspException。

doStartTag

```
public int doStartTag() throws JspException
```

在已经设置了页面上下文、父标签和开始标签所表示的任何属性之后调用，返回码表示JSP实现servlet是否应该评估此标签体，即EVAL_BODY_INCLUDE或BodyTag.EVAL_BODY_BUFFERED表示肯定，SKIP_BODY则表示否定。此方法可以抛出一个表示致命性错误的JspException。只有在标签处理器实现了BodyTag时，BodyTag.EVAL_BODY_BUFFERED才有效。

getParent

```
public Tag getParent()
```

返回父标签（最靠近的包围标签处理器），如果不存在父标签返回null。

release

```
public void release()
```

确保在退出页面之前调用此方法。允许标签处理器释放所占用的任何资源，并重置其状态，以便在需要时可以被复用。

setPageContext

```
public void setPageContext(PageContext pc)
```

在要求处理程序进行任何操作之前，生成的servlet首先调用此方法。实现类应该保存此上下文变量，以便在标签的生命期中任何时刻都可以得到此上下文变量。从页面的上下文处，标签处理器可以访问全部的JSP隐式对象，并且可以在任何位置获取或设置属性。

setParent

```
public void setParent(Tag t)
```

精彩好书见www.book118.com

设置父标签。允许标签处理器在计算堆栈中找到其上层的标签。在setPageContext方法之后紧接着就调用此方法。

TagAttributeInfo

全称：javax.servlet.jsp.tagext.TagAttributeInfo

类型：类

一种描述有关标签属性信息的类。此标签在转换时可以用到。

构造器：

TagAttributeInfo

```
public TagAttributeInfo(
String name,
boolean required,
String type,
boolean reqTime)
```

创建一个新的TagAttributeInfo。计划只在TagLibrary对象中调用此构造器。

方法

canBeRequestTime

```
public boolean canBeRequestTime()
```

如果此属性可以保持一个请求时值，则返回真。

getIdAttribute

```
public static TagAttributeInfo getIdAttribute
(TagAttributeInfo[] a)
```

一种工具方法，用于搜索属性名为“id”的TagAttributeInfo对象数组。

getName

```
public String getName()
```

返回属性名。

getTypeName

```
public String getTypeName()
```

返回作为字符串的属性类型。

isRequired

```
public boolean isRequired()
```

如果需要该属性，则返回true。

toString

```
public String toString()
```

以字符串形式的返回该对象。

精彩好书见www.book118.com

TagData

全称：javax.servlet.jsp.tagext.TagData
类型：类
实现：java.lang.Cloneable
包含关于标签属性的转换时信息。设计只为JSP容器所使用。
构造器：

TagData

```
public TagData(Hashtable attrs)
```

从一个哈希表创建新的TagData对象。

TagData

```
public TagData(Object[][] atts)
```

从一个二维属性/值对的数组创建新的TagData对象。
方法：

getAttribute

```
public Object getAttribute(String attName)
```

返回拥有指定名称的属性。如果该值必须在请求时指定，也可以返回REQUEST_TIME_VALUE，如果在标签中未指定此属性返回null。

getAttributes

```
public Enumeration getAttributes()
```

返回属性的一个枚举值。

getAttributeString

```
public String getAttributeString(String attName)
```

以字符串形式返回属性的值。

getId

```
public String getId()
```

如果指定了id属性的值，则返回该值。

setAttribute

```
public void setAttribute(String attName, Object value)
```

设置一个属性的值为指定值。

TagExtraInfo

全称：javax.servlet.jsp.tagext.TagExtraInfo
类型：抽象类

精彩好书见www.book118.com

对于需要定义变量或需要依据其属性执行验证的标签而言，必须扩展该TagExtraInfo类。该子类与位于标签库描述器的定制标签相关联。

构造器：

TagExtraInfo

```
public TagExtraInfo()
```

创建一个新的TagExtraInfo对象。

方法：

getTagInfo

```
public final TagInfo getTagInfo()
```

返回此类的TagInfo。

getVariableInfo

```
public VariableInfo getVariableInfo(TagData data)
```

基于属性名列表和data参数的值，构造一个VariableInfo对象数组，用于描述属性的名称、类型、存在性和每个要创建的脚本变量的范围。

isValid

```
public boolean isValid(TagData data)
```

如果在TagData参数中所指的属性有效，则返回true。

setTagInfo

```
public final void setTagInfo(TagInfo tagInfo)
```

设置该类的TagInfo。

TagInfo

全称：javax.servlet.jsp.tagext.TagInfo

类型：类

位于标签库描述器Tag元素的对象表示。

构造器：

TagInfo

```
public TagInfo(
    String tagName,
    String tagClassName,
    String bodycontent,
    String infoString,
    TagLibraryInfo taglib,
    TagExtraInfo tagExtraInfo,
    TagAttributeInfo attributeInfo)
```

从JSP1.1格式的标签库描述器创建一个新的TagInfo对象。

精彩好书见www.book118.com

TagInfo

```
public TagInfo(
    String tagName,
    String tagClassName,
    String bodycontent,
    String infoString,
    TagLibraryInfo taglib,
    TagExtraInfo tagExtraInfo,
    TagAttributeInfo attributeInfo,
    String displayName,
    String smallIcon,
    String largeIcon,
    TagVariableInfo tvi)
```

从JSP1.2格式的标签库描述器创建一个新的TagInfo对象

方法：

getAttributes

```
public TagAttributeInfo[] getAttributes()
```

返回描述该标签属性的数组，如果属性不存在返回null。

getBodyContent

```
public String getBodyContent()
```

返回该标签的bodycontent属性，如同在标签库描述器中所指定的。

getDisplayName

```
public String getDisplayName()
```

返回该标签的displayName属性，如同在标签库描述器中所指定的。

getInfoString

```
public String getInfoString()
```

返回该标签的info元素，如同在标签库描述器中所指定的。

getLargeIcon

```
public String getLargeIcon()
```

返回该标签的大图标路径，如同在标签库描述器中所指定的。

getSmallIcon

```
public String getSmallIcon()
```

返回该标签的小图标路径，如同在标签库描述器中所指定的。

getTagClassName

```
public String getTagClassName()
```

返回标签处理器类的名称。

精彩好书见www.book118.com

getTagExtraInfo

```
public TagExtraInfo getTagExtraInfo()
```

返回标签扩展信息类的名称。

getTagLibrary

```
public TagLibraryInfo getTagLibrary()
```

返回该标签TagLibraryInfo对象的一个引用。

getTagName

```
public String getTagName()
```

返回标签名。

getTagVariableInfos

```
public TagVariableInfo getTagVariableInfos()
```

返回与TagInfo相关的TagVariableInfo对象。

getVariableInfo

```
public VariableInfo getVariableInfo(TagData data)
```

返回该标签VariableInfo对象的一个引用。

isValid

```
public boolean isValid(TagData data)
```

返回相关TagExtraInfo类的isValid()方法的计算结果。

setTagExtraInfo

```
public void setTagExtraInfo(TagExtraInfo tei)
```

保存该标签TagExtraInfo的一个引用。

setTagLibrary

```
public void setTagLibrary(TagLibraryInfo tl)
```

设置TagLibraryInfo属性。

toString

```
public String toString()
```

为了诊断，以字符串的形式返回该对象。

TagLibraryInfo

全称：javax.servlet.jsp.tagext.TagLibraryInfo

类型：抽象类

一种封装与taglib指令相关的信息及其底层的标签库描述器（TLD）的类。

方法：

精彩好书见www.book118.com

getInfoString

```
public String getInfoString()
```

从TLD返回info属性。

getPrefixString

```
public String getPrefixString()
```

返回分派给taglib指令的前缀。

getReliableURN

```
public String getReliableURN()
```

从TLD返回reliableURL属性。

getRequiredVersion

```
public String getRequiredVersion()
```

返回JSP容器所需要的版本。

getShortName

```
public String getShortName()
```

从TLD返回短名称属性。

getTag

```
public TagInfo getTag(String shortname)
```

返回与给定标签名相关的TagInfo对象。

getTags

```
public TagInfo[] getTags()
```

返回在这个标签库中定义的全部标签的TagInfo数组对象。

getURI

```
public String getURI()
```

从taglib指令返回uri属性的值。

TagLibraryValidator

全称: javax.servlet.jsp.tagext.TagLibraryValidator

类型：抽象类

一种与位于TLD中JSP页面相关联的验证器类。验证器对表示JSP页面的XML文档实施验证操作。

构造器：

TagLibraryValidator

```
public TagLibraryValidator()
```

精彩好书见www.book118.com

创建一个新的TagLibraryValidator。

方法：

getInitParameters

```
public Map getInitParameters()
```

返回初始化参数。

release

```
public void release()
```

释放用于验证器的验证数据。

setInitParameters

```
public void setInitParameters(Map map)
```

向验证器提供初始化的主键/值参数。

validate

```
public String validate
   (String prefix, String uri, PageData page)
```

验证JSP页面。如果页面有效，则返回null。

TagSupport

全称：javax.servlet.jsp.tagext.TagSupport

类型：类

扩展：javax.servlet.jsp.tagext.IterationTag

java.io.Serializable

Tag接口的一种具体实现。Tag处理器可以扩展该类，并且只实现那些需要修改的方法。

构造器：

TagSupport

```
public TagSupport()
```

创建一个新的TagSupport对象。

方法：

doAfterBody

```
public int doAfterBody() throws JspException
```

在评估标签体之后激活此方法。

DoEndTag

```
public int doEndTag() throws JspException
```

当遇到结束标签时调用此方法。返回码表示JSP实现servlet是否应该处理余下的页面，即EVAL_PAGE或SKIP_PAGE。此方法可以抛出一个表示致命性错误的JspException。TagSupport

精彩好书见www.book118.com

实现返回EVAL_PAGE。

doStartTag

```
public int doStartTag() throws JspException
```

在已经设置了页面上下文、父标签和开始标签所表示的任何属性之后调用，返回码表示JSP实现servlet是否应该评估此标签体，即EVAL_BODY_INCLUDE或BodyTag.EVAL_BODY_BUFFERED表示肯定，SKIP_BODY则表示否定。此方法可以抛出一个表示致命性错误的JspException。只有在标签处理器实现了BodyTag时，BodyTag.EVAL_BODY_BUFFERED才有效。TagSupport实现返回SKIP_BODY。

findAncestorWithClass

```
public static final Tag findAncestorWithClass
(Tag from, Class klass)
```

在父标签堆栈中搜索指定类的最接近的标签处理器。这允许一种“内部”标签访问其包围标签的信息。

getId

```
public String getId()
```

返回该标签id属性的值。

getParent

```
public Tag getParent()
```

返回该标签处理器实例的直接父标签。

getValue

```
public Object getValue(String k)
```

返回保存于该标签处理器内拥有给定名称的对象。

getValues

```
public Enumeration getValues()
```

返回保存于该标签处理器中值的名称枚举值。

release

```
public void release()
```

确保在退出页面之前调用此方法。允许标签处理器释放所占用的任何资源，并重置其状态，以便在需要时可以被复用。

removeValue

```
public void removeValue(String k)
```

对于保存在该标签处理器中拥有指定名称的值，如果存在，则删除它。

setId

精彩好书见www.book118.com


```
public void setId(String id)
```

设置该标签的id属性。

setPageContext

```
public void setPageContext(PageContext pageContext)
```

在要求处理程序进行任何操作之前，生成的servlet首先调用此方法。实现类应该保存此上下文变量，以便在标签的生命期中任何时刻都可以得到此上下文变量。从页面的上下文处，标签处理器可以访问全部的JSP隐式对象，并且可以在任何位置获取或设置属性。

SetParent

```
public void setParent(Tag t)
```

设置父标签。允许标签处理器在计算堆栈中找到其上层的标签。在setPageContext方法之后紧接着就调用此方法。

SetValue

```
public void setValue(String k, Object o)
```

以指定的名称在标签处理器中保存该对象。

TagVariableInfo

全称：javax.servlet.jsp.tagext.TagVariableInfo

类型：类

一种封装自标签库中提取的标签变量信息的类。

构造器：

TagVariableInfo

```
public TagVariableInfo(
    String nameGiven,
    String nameFromAttribute,
    String className,
    boolean declare,
    int scope)
```

创建一种新的TagVariableInfo对象。

方法：

getClassName

```
public String getClassName()
```

返回位于TLD中<variable-class>元素的值。

getDeclare

```
public boolean getDeclare()
```

返回位于TLD中<declare>元素的值。

精彩好书见www.book118.com

getNameFromAttribute

```
public String getNameFromAttribute()
```

返回位于TLD中<name-from-attribute>元素的值。

getNameGiven

```
public String getNameGiven()
```

返回位于TLD中<name-given>元素的值。

getScope

```
public int getScope()
```

返回位于TLD中<scope>元素的值。

TryCatchFinally

全称：javax.servlet.jsp.tagext.TryCatchFinally

类型：接口

一种可以由标签处理器实现的补充接口，以便允许在该标签引用的catch和finally语句块中调用这些标签处理器。

方法：

doCatch

```
public void doCatch(Throwable t) throws Throwable
```

在评估一个标签体时，如果出现一个异常，则在catch语句块中激活此方法。

doFinally

```
public void doFinally()
```

在评估一个标签体时，如果出现一个异常，则在finally语句块中激活此方法。

VariableInfo

全称：javax.servlet.jsp.tagext.VariableInfo

类型：类

一种数据结构，它提供有关创建定制标签的脚本变量的配置信息。主要用于TagExtraInfo子类的getVariableInfo()方法。

构造器：

VariableInfo

```
public VariableInfo(
    String varName,
    String className,
    boolean declare,
    int scope)
```

精彩好书见www.book118.com

创建一个新的VariableInfo对象。其参数如下所示：

- varName　　将要创建变量的名称
- className　　变量类的全质名称
- declare　　一个布尔型值，如果生成的servlet应该包含变量声明，则其值为true。
- scope　　一个表示变量范围的整数。可能为AT_BEGIN、AT_END或NESTED。

方法：

getClassName

```
public String getClassName()
```

返回类名。

getDeclare

```
public boolean getDeclare()
```

返回布尔型属性，它表示变量是否应该被声明。

getScope

```
public int getScope()
```

返回表示变量范围的整数。

getVarName

```
public String getVarName()
```

返回变量名。

精彩好书见www.book118.com

附录C HTTP参考

本附录由两个表组成，其中一个描述超文本传输协议（HTTP）的响应码，另一个描述HTTP头标。关于HTTP的更多信息，请参考规范RFC 2616。

HTTP响应码

响应码由三位十进制数字组成，它们出现在由HTTP服务器发送的响应的第一行。响应码分五种类型，由它们的第一位数字表示：

- 1xx：**信息** 请求收到，继续处理。
- 2xx：**成功** 行为被成功地接受、理解和采纳。
- 3xx：**重定向** 为了完成请求，必须进一步执行的动作。
- 4xx：**客户端错误** 请求包含语法错误或者请求无法实现。
- 5xx：**服务器错误** 服务器不能实现一种明显无效的请求。

下表显示每个响应码及其含义：

响应码	含义
100	继续
101	分组交换协议
200	OK
201	被创建
202	被采纳
203	非授权信息
204	无内容
205	重置内容
206	部分内容
300	多选项
301	永久地传送
302	找到
303	参见其他
304	未改动
305	使用代理
307	暂时重定向
400	错误请求
401	未授权
402	要求付费
403	禁止
404	未找到
405	不允许的方法

精彩好书见www.book118.com

(续)

响应码	含义
406	不被采纳
407	要求代理授权
408	请求超时
409	冲突
410	过期的
411	要求的长度
412	前提不成立
413	请求实例太大
414	请求URI太大
415	不支持的媒体类型
416	无法满足的请求范围
417	失败的预期
500	内部服务器错误
501	未被使用
502	网关错误
503	不可用的服务
504	网关超时
505	HTTP版本未被支持

HTTP头标

头标由主键/值对组成。它们描述客户端或者服务器的属性、被传输的资源以及应该如何实现连接。存在四种不同类型的头标：

- 通用头标　既可用于请求，也可用于响应，并且是作为一个整体而不是特定资源与事务相关联。
- 请求头标　允许客户端传递关于自身的信息和希望的响应形式。
- 响应头标　服务器用于传递自身信息和响应。
- 实体头标　定义被传送资源的信息。既可用于请求，也可用于响应。

头标是以如下所示单一文本行形式发送的

<name>: <value><CRLF>

其中

name 是头标名，它是大小写敏感的；

value 是头标的值；

CRLF 是回车/换行符。

注意，在头标名与头标值之间，存在一个冒号和一个或者多个用作分隔的空格。

JSP页面可以使用其request对象的getHeader()方法读HTTP头标，还可以使用response.setHeader().java.net写入它们。在一个URL流中，URLConnection提供类似的方法访问头标。

精彩好书见www.book118.com

下表描述在HTTP/1.1中用到的头标：

头 标	描 述
Accept	定义客户端可以处理的媒体类型，按优先级排序。在一个以逗号为分隔的列表中，可以定义多种类型。可以使用通配符。例如： Accept: image/jpeg, image/pjpeg, image/png, */*
Accept-Charset	定义客户端可以处理的字符集，按优先级排序。在一个以逗号为分隔的列表中，可以定义多种类型。可以使用通配符。例如： Accept-Charset : iso-8859-1, * , utf-8
Accept-Encoding	定义客户端可以理解的编码机制。例如： Accept-Encoding : gzip, compress
Accept-Language	定义客户端乐于接受的自然语言列表。例如： Accept-Language : en, de
Accept-Ranges	一个响应头标，它允许服务器指明：将在给定的偏移和长度处，为资源组成部分的接受请求。该头标的值被理解为请求范围的度量单位。例如： Accept-Ranges : bytes Accept-Ranges : none
Age	允许服务器规定自服务器生成该响应以来，所经过的时间长度，以秒为单位。该头标主要用于缓存响应。例如： Age : 30
Allow	一个响应头标，它定义一个由位于请求URI中的资源所支持的HTTP方法列表。例如： Allow : GET , HEAD, PUT
Authorization	一个响应头标，用于定义访问一种资源所必需的授权（域和被编码的用户ID与口令）。例如： Authorization : Basic YXV0aG9yOnBoaWw=
Cache-Control	一个用于定义缓存指令的通用头标。例如： Cache-Control: max-age=30
Connection	一个用于表明是否保持socket连接为开放的通用头标。例如： Connection : close Connection : keep-alive
Content-Base	一种定义基本URI的实体头标，为了在实体范围内解析相对URLs。如果没有定义Content-Base头标，解析相对URLs，使用Content-Location URI（存在且绝对）或使用URI请求。例如：Content-Base:Http://www.lyricnote.com
Content-Encoding	一种介质类型修饰符，标明一个实体是如何编码的（zipped,compressed,等。）例如： Content-Encoding: gzip
Content-Language	用于指定在输入流中数据的自然语言类型。例如： Content-Language: en
Content-Length	指定包含于请求或响应中数据的字节长度。例如： Content-Length: 382
Content-Location	指定包含于请求或响应中的资源的定位（URI）。如果是一绝。对URL它也作为被解析实体的相对URL的出发点。例如： Content-Location: http://www.lyricnote.com/newsletter
Content-MD5	实体的一种MD5摘要，用作校验和。发送方和接受方都计算MD5摘要。接收方将其计算的值与在此头标中传递的值进行比较。例如： Content-MD5: <base64 of 128 bit MD5 digest>

精彩好书见www.book118.com

(续)

头 标	描 述
Content-Range	随部分实体一同发送；标明被插入节的低位与高位字节偏移。也标明此实体的总长度。例如： Content-Range: 1001-2000/5000
Content-Type	标明发送或者接收的实体的MIME类型。例如： Content-Type: text/html
Date	发送HTTP消息的日期。例如： Date: Mon, 06 Mar 2000 18 :42:51 GMT
ETag	一种实体头标，它向被发送的资源分派一个惟一的标识符。对于可以使用多种URL请求的资源，ETag可以用于确定实际被发送的资源是否为同一资源。例如： ETag： "208f-419e-30f8dc99"
Expires	指定实体的有效期。例如： Expires: Mon, 05 Dec 2008 12:00:00 GMT
From	一种请求头标，给定控制用户代理的人工用户的电子邮件地址。例如： From: webmaster@lyricnote.com
Host	被请求资源的主机名（以及可选的端口号）。对于使用HTTP/1.1的请求而言，此域是强制性的。例如： Host: www.lyricnote.com
If-Modified-Since	如果包含了GET请求，导致该请求条件性地依赖于资源上次修改日期。如果出现了此头标，并且自指定日期以来，此资源已经被修改，应该返回一个304响应代码。例如： If-Modified-Since: Web, 01 Mar 2000 12:00:00 GMT
If-Match	如果包含于一个请求，指定一个或者多个实体标记（见ETag）。只发送其ETag与列表中标记匹配的资源。例如： If-Match: "208f-419e-30f8dc99"
If-None-Match	如果包含于一个请求，指定一个或者多个实体标记（见ETag）。只有当资源的ETag不与列表中的任何一个条目匹配，操作才执行。例如： If-None-Match: "208f-419e-30f8dc99"
If-Range	指定资源的一个实体标记（见ETag），客户端已经拥有此资源的一个拷贝。必须与Range头标一同使用。如果此实体自上次被客户端检索以来，还不曾被修改过，那么，服务器将只发送指定的范围，否则，它将发送整个资源。例如： Range: bytes=0-499 If-Range: "208f-419e-30f8dc99"
If-Unmodified-Since	与If-Modified-Since相似，不过是在相反的意义上。只有自指定的日期以来，被请求的实体还不曾被修改过，才会返回此实体。例如： If-Unmodified-Since: Wed 01 Mar 2000 12:00:00 GMT
Last-Modified	指定被请求资源上次被修改的日期和时间。例如： Last-Modified: Wed, 08 Mar 2000 12:00:00 GMT
Location	对于一个已经移动的资源，用于重定向请求者至另一个位置。与状态编码302（暂时移动）或者301（永久性移动）配合使用。例如： Location: http://www2.lyricnote.com/index.jsp
Max-Forwards	一个用于TRACE方法的请求头标，以指定代理或网关的最大数目，该请求通过网关才得以路由。在通过请求传递之前，代理或网关应该减少此数目。例如： Max-Forwards: 3

精彩好书见www.book118.com

(续)

头　标	描　　述
Pragma	一个通用头标，它发送实现相关的信息。例如： Pragma: no-cache
Proxy-Authenticate	类似于WWW-Authenticate，但是有意请求只来自于请求链（代理）的下一个服务器的认证。例如： Proxy-Authenticate: Basic realm-Admin
Proxy-Authorization	类似于授权，但并非有意传递任何比在即时服务器链中更进一步的内容。例如： Proxy-Authorization: Basic YXV0aG9yOnBoaWw=
Public	列表显示服务器所支持的方法集。例如： Public: OPTIONS, MGET, MHEAD, GET, HEAD
Range	指定一种度量单位和一个部分被请求资源的偏移范围。例如： Range:bytes=206-5513
Referer	一种请求头标域，标明产生请求的初始资源。对于HTML表单，它包含此表单的Web页面的地址。例如： Referer: http://www.lyricnote.com/product/search.html
Retry-After	一种响应头标域，由服务器与状态编码503（无法提供服务）配合发送，以标明再次请求之前应该等待多长时间。此时间既可以是一种日期，也可以是一种秒为单位的数目。例如： Retry-After: 18 Retry-Atfer: Thu, 09 Mar 2014 16:45:15 GMT
Server	一种标明Web服务器软件及其版本号的头标。例如： Server:Apache/1.3.12 (Win32)
Transfer-Encoding	一种通用头标，标明对应该被接受方反向的消息体实施变换的类型。例如： Transfer-Encoding:chunked
Upgrade	允许服务器指定一种新的协议或者新的协议版本，与响应编码101（切换协议）配合使用。例如： Upgrade:HTTP/2.0
User-Agent	定义用于产生请求的软件类型（典型的，如Web浏览器）。例如： User-Agent : Mozilla / 4.0 (compatible ; MSIE 5.5 ; Windows NT ; DigExt) User-Agent : Mozilla / 4.7 [en] (WinNT ; I)
Vary	一个响应头标，用于表示：使用服务器驱动的协商从可用的响应表示中选择响应实体。例如： Vary : *
Via	一个包含所有中间主机和协议的通用头标，用于满足请求。例如： Via : 1.0 fred.com , 1.1 wilma.com
Warning	用于提供关于响应状态补充信息的响应头标。例如： Warning : 99 www.lyricnote.com Piano needs tuning
www-Authenticate	一个提示用户代理提供用户名和口令的响应头标。与状态编码401（未授权）配合使用。希望响应一个授权头标。例如： www-Authenticate : Basic realm = lyricnote_mgmt

精彩好书见www.book118.com

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝ＪＳＰ技术大全
作者＝
页数＝６１２
ＳＳ号＝１０４４０８１３
出版日期＝
ｗｗｗ．ｍｙｃｏｄｅｓ．ｎｅｔ

精彩好书见www.book118.com

封面页
书名页
版权页
前言页
目录页
译者序
前言
第一部分 Web编程环境
第1章 Web市场
第2章 Web应用演化
2.1 Web的产生
2.2 Web编程模型的增长
2.3 从客户端移向服务器端方案
第3章 超文体传输协议介绍
3.1 HTTP是什么
3.1.1 Internet上请求文档的一种语言
3.1.2 HTTP规范
3.2 HTTP请求模型
3.2.1 连接至Web服务器
3.2.2 发送HTTP请求
3.2.3 服务器接受请求
3.2.4 来自服务器的HTTP响应
3.3 实例
3.4 小结
第4章 servlet介绍
4.1 servlet生命期
4.1.1 init
4.1.2 service
4.1.3 destroy
4.2 例子：千米每公升到英里每加仑servlet
4.3 servlet类
4.3.1 servlet
4.3.2 servlet请求
4.3.3 servlet响应
4.3.4 servlet上下文
4.4 线程模型
4.5 HTTP会话
4.6 小结
第5章 JSP介绍
5.1 JSP工作方式
5.2 一个基本例子
第二部分 JSP 元素
第6章 JSP语法和语义
6.1 JSP开发模型
6.2 JSP页面组件
6.2.1 伪指令
6.2.2 注释
6.2.3 表达式
6.2.4 scriptlet
6.2.5 声明
6.2.6 隐含对象
6.2.7 标准行为
6.2.8 标签扩展

精彩好书见www.book118.com

6.3 一个完整实例
6.3.1 page伪指令
6.3.2 <jsp:include>行为
6.3.3 scriptlet
6.3.4 表达式
6.3.5 一个声明
6.4 小结
第7章 表达式和scriptlet
7.1 表达式
7.2 scriptlet
7.3 通过JSP容器处理表达式和scriptlet
7.3.1 HTML模板数据和表达式
7.3.2 scriptlet内容
7.3.3 容器生成的初始化和退出代码
7.4 隐含对象和JSP环境
7.4.1 Request
7.4.2 Response
7.4.3 PageContext
7.4.4 Session
7.4.5 Application
7.4.6 Out
7.4.7 Config
7.4.8 Page
7.4.9 Exception
7.5 初始化参数
7.6 小结
第8章 声明
8.1 声明是什么
8.2 声明的基本用法
8.3 变量声明
8.4 方法定义
8.4.1 覆盖jspInit和jspDestroy
8.4.2 隐含对象的访问
8.5 内部类
8.6 小结
第9章 请求发送
9.1 请求过程的剖析
9.2 包含其他资源
9.3 include伪指令
9.3.1 其工作方式
9.3.2 改变一个被包含文件的影响
9.3.3 使用include伪指令复制源码
9.4 <jsp:include>行为
9.5 使用哪种方法
9.6 转发请求
9.7 RequestDispatcher对象
9.8 模型1对比模型2
9.9 小结
第10章 Page伪指令
10.1 language
10.2 extends
10.2.1 JSP超类所需的接口
10.2.2 一个JSP超类例子

精彩好书见www.book118.com

10.3 import
10.4 session
10.5 buffer和autoFlush
10.6 isThreadSafe
10.7 info
10.8 contentType
10.9 errorpage和isErrorpage
10.10 小结
第11章 JSP标签扩展
11.1 为什么要定制标签
11.2 开发第一个定制标签
11.2.1 第1步--定义标签
11.2.2 第2步--创建TLD入口
11.2.3 第3步--编写标签处理器
11.2.4 第4步--将标签并入JSP页面
11.3 标签处理器工作方式
11.3.1 JSP容器的功能
11.3.2 标签处理器功能
11.4 标签库
11.4.1 标签库描述器
11.4.2 taglib伪指令
11.5 标签处理器API
11.5.1 Tag接口
11.5.2 TagSupport类
11.6 标签处理器生命期
11.6.1 流线图
11.6.2 生成代码的一个例子
11.7 定义标签属性
11.8 体标签处理器API
11.8.1 BodyContent
11.8.2 BodyTag接口
11.8.3 BodyTagSupport类
11.9 体标签处理器生命期
11.10 定义脚本变量
11.10.1 TagExtraInfo类
11.10.2 标签属性有效性检验
11.11 协作标签
11.11.1 使用Syntactic Scoping
11.11.2 例子：switch标签
11.12 数据库查询例子的实现
11.12.1 所需标签
11.12.2 标签库描述器
11.12.3 标签处理器
11.13 小结
第三部分 JSP 行为
第12章 HTML窗体
12.1 FORM元素
12.2 窗体输入元素
12.2.1 使用INPUT标签创建的元素
12.2.2 使用select和option创建的元素
12.2.3 textarea元素
12.3 窗体有效性检验
12.4 服务器端的窗体处理

精彩好书见www.book118.com

１２．５　小结
第１３章　数据库访问
１３．１　ＪＤＢＣ简介
１３．１．１　基本ＪＤＢＣ操作
１３．１．２　基本ＪＤＢＣ类
１３．１．３　一个简单ＪＤＢＣ实例
１３．２　ＪＤＢＣ驱动器
１３．２．１　驱动器类型
１３．２．２　ＪＤＢＣ－ＯＣＢＣ桥
１３．２．３　注册一个驱动器
１３．３　连接到一个数据库
１３．４　语句接口
１３．４．１　Ｓｔａｔｅｍｅｎｔ
１３．４．２　ＰｒｅｐａｒｅｄＳｔａｔｅｍｅｎｔ
１３．４．３　ＣａｌｌａｂｌｅＳｔａｔｅｍｅｎｔ
１３．５　结果集
１３．５．１　可滚动的结果集
１３．５．２　可修改结果集
１３．５．３　ＲｏｗＳｅｔ
１３．６　使用元数据
１３．６．１　数据库元数据
１３．６．２　ＲｅｓｕｌｔＳｅｔＭｅｔａｄａｔａ
１３．７　ＪＤＢＣ　２．０及以上版本中的新特性
１３．８　小结
第１４章　会话和线程管理
１４．１　会话跟踪
１４．１．１　隐藏域
１４．１．３　ｃｏｏｋｉｅ
１４．２　会话ＡＰＩ
１４．２．１　创建会话
１４．２．２　从会话中保存和检索对象
１４．２．３　销毁会话
１４．２．４　修订后实例
１４．２．５　会话捆绑侦听者
１４．３　线程管理
１４．４　ｓｅｒｖｌｅｔ线程模型
１４．４．１　缺省线程模型
１４．４．２　单线程模型
１４．５　多线程应用
１４．６　应用考虑
１４．７　小结
第１５章　ＪＳＰ和ＪａｖａＢｅａｎ
１５．１　ＪａｖａＢｅａｎ是什么
１５．１．１　ｂｅａｎ属性
１５．１．２　持久性
１５．２　ＪＳＰ行为
１５．２．１　＜ｊｓｐ：ｕｓｅＢｅａｎ＞
１５．２．２　＜ｊｓｐ：ｓｅｔＰｒｏｐｅｒｔｙ＞
１５．２．３　＜ｊｓｐ：ｇｅｔＰｒｏｐｅｒｔｙ＞
１５．３　一个完整例子－－带有ｂｅａｎ的个性化风格
１５．３．１　从Ｗｅｂ得到天气数据
１５．３．２　ＬｙｒｉｃＮｏｔｅ人口
１５．４　小结

精彩好书见www.book118.com

第１６章　ＪＳＰ和ＸＭＬ
１６．１　ＸＭＬ简介
１６．１．１　ＸＭＬ解决的问题
１６．１．２　ＸＭＬ语法
１６．１．３　文档类型定义
１６．２　ＸＭＬ解析器
１６．２．１　文档对象模型
１６．２．２　ＸＭＬ的简单ＡＰＩ
１６．３　使用ＸＳＬＴ进行ＸＳＬ转换
１６．４　小结
第１７章　ＪＳＰ测试和调试
１７．１　建立思想模型
１７．２　独立测试
１７．３　调试工具
１７．３．１　捕获窗体参数
１７．３．２　调试Ｗｅｂ客户端
１７．３．３　跟踪ＨＴＴＰ请求
１７．４　小结
第１８章　发布Ｗｅｂ应用
１８．１　Ｗｅｂ应用环境
１８．１．１　目录结构
１８．１．２　资源映射
１８．１．３　ｓｅｒｖｌｅｔ上下文
１８．２　Ｗｅｂ存档文件
１８．３　发布描述器：ｗｅｂ．ｘｍｌ
１８．４　发布描述器示例
１８．５　小结
第１９章　事例分析：一个产品支持中心
１９．１　过程流
１９．２　数据模式
１９．３　开发系统
１９．４　模式－视图－控制器结构
１９．４．１　模式类
１９．４．２　视图类
１９．４．３　控制器类
１９．５　小结
第四分部　ＪＳＰ和其他Ｗｅｂ组件
第２０章　与其他客户端进行通信
２０．１　ＵＲＬ连接
２０．１．１　ＵＲＬ类
２０．１．２　ＵＲＬＣｏｎｎｅｃｔｉｏｎ类
２０．１．３　ＨｔｔｐＵＲＬＣｏｎｎｅｃｔｉｏｎ类
２０．２　作为客户端的Ｊａｖａ应用
２０．２．１　ＪＳＰ竞价服务器
２０．２．２　竞价客户端应用
２０．３　一个Ｊａｖａ　Ａｐｐｌｅｔ客户端
２０．３．１　Ｊａｖａ插件
２０．３．２　ＰｒｉｃｅＱｕｏｔｅＡｐｐｌｅｔ
２０．４　一个Ｐｅｒｌ　客户端
２０．４．１　通用数据库选择服务器
２０．４．２　Ｐｅｒｌ脚本
２０．５　小结
第２１章　与其他服务器通信

精彩好书见www.book118.com

２１．１　服务器端脚本环境
２１．２　从一个ＪＳＰ页面发送邮件
２１．２．１　发送邮件的方法
２１．２．２　在产品支持系统中的电子邮件通告
２１．３　小结
第五部分　附录
附录Ａ　ｓｅｒｖｌｅｔ　ＡＰＩ版本２．３
附录Ｂ　ＪＳＰ　ＡＰＩ版本１．２
附录Ｃ　ＨＴＴＰ参考
附录页