

开发专家

之 Sun ONE

# Tomcat与Java Web 开发技术详解

书赠光盘  
为书中范  
例文件及  
源代码

Sun Weiwei Li Hongcheng  
孙卫琴 李洪成  
飞思科技产品研发中心

编著  
监制

Sun  
ONE



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

JSP应用开发详解(第二版)  
EJB应用开发详解  
Java Web服务应用开发详解  
Java TCP/IP应用开发详解  
Java 2应用开发指南(第二版)  
J2EE应用开发(WebLogic + JBuilder)  
J2EE企业级应用开发  
J2EE技术参考手册  
J2ME技术参考手册  
JBuilder精髓  
Tomcat与Java Web开发技术详解



## 开发专家

之 Sun ONE

# SUN ONE

本书详细介绍了在最新Tomcat 5版本上开发Java Web 应用的各种技术。主要内容包括：Tomcat和Java Web开发的基础知识、Java Web开发的高级技术、Tomcat与当前其他通用软件的集成，以及Tomcat的各种高级功能。

书中内容注重理论与实践相结合，列举了大量具有典型性和实用价值的Web应用实例，并提供了详细的开发和部署步骤。

### 读者对象：

本书语言深入浅出，通俗易懂。无论对于Java Web 开发的新手还是行家来说，本书都是精通Tomcat技术和开发Java Web应用的必备的实用手册。



随书赠送光  
盘为书中范  
例源文件及  
相关软件

飞思在线：<http://www.fecit.cn>

ISBN 7-5053-9392-8



本书贴有激光防  
伪标志，凡没有  
防伪标志者，属  
于盗版图书。



总策划：郭晶  
执行策划：何郑燕  
责任编辑：陆舒敏  
封面设计：张跃

9 787505 393929 >

ISBN 7-5053-9392-8/TP · 5412

定价：45.00 元  
(含光盘)

开发专家之 Sun ONE

# Tomcat 与 Java Web 开发技术详解

孙卫琴 李洪成 编著

飞思科技产品研发中心 监制

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

本书详细介绍了在最新 Tomcat 5 版本上开发 Java Web 应用的各种技术。主要内容包括：Tomcat 和 Java Web 开发的基础知识，Java Web 开发的高级技术，Tomcat 与当前其他通用软件的集成，以及 Tomcat 的各种高级功能。

书中内容注重理论与实践相结合，列举了大量具有典型性和实用价值的 Web 应用实例，并提供了详细的开发和部署步骤。由于 Java Web 技术是 SUN 公司在 Java Servlet 规范中提出的通用技术，因此本书讲解的 Java Web 应用例子可以运行在任何一个实现 SUN 的 Servlet 规范的 Java Web 服务器上。随书配套光盘内容为书中范例源程序，以及本书涉及到的软件的安装程序。

本书语言深入浅出，通俗易懂。无论对于 Java Web 开发的新手还是行家来说，本书都是精通 Tomcat 技术和开发 Java Web 应用的必备的实用手册。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

### 图书在版编目 (CIP) 数据

Tomcat 与 Java Web 开发技术详解/孙卫琴，李洪成编著. —北京：电子工业出版社，2004.4  
(开发专家之 Sun ONE)

ISBN 7-5053-9392-8

I . T... II . ①孙...②李... III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2003) 第 107424 号

责任编辑：陆舒敏

印 刷：北京东光印刷厂

出版发行：电子工业出版社

北京海淀区万寿路 173 信箱 邮编：100036

经 销：各地新华书店

开 本：787×1092 1/16 印张：28.25 字数：723.2 千字

印 次：2004 年 4 月第 1 次印刷

印 数：5 000 册 定价：45.00 元（含光盘 1 张）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：010-68279077。质量投诉请发邮件至 zhts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

## 出版说明

“开发专家”是电子工业出版社计算机图书研发部长期以来精心培育的计算机科学技术类本版品牌。这个品牌是由多个专题系列组成的横向大系列，涵盖了计算机技术的各个方面，特别是一直受到极大关注的程序开发类系列，例如《Borland 开发专家》、《开发专家之数据库》、《开发专家之网络编程》、《开发专家之 Delphi》以及《开发专家之 Sun ONE》等。这些专题系列基于各自的角度，从纵向上包含了该专题的所有内容。因此，整个“开发专家”的品牌架构纵横交错，囊括了所有的计算机技术和所有的技术层面，海纳百川而又极具可扩展性。

“开发专家”的作者队伍主要依托于“飞思科技产品研发中心”。“飞思科技产品研发中心”是由专业的策划人员、权威的技术专家和资深的作者队伍共同构成的。在图书的出版上，形成了以研发为基础、以出版为中心、以服务为支持的专业化出版框架和流程。通过深入的市场调查和技术跟踪，在综合了技术需求和读者焦点等因素的基础上，形成各系列丛书的写作重点和大纲，然后聘请业界的最前沿学者进行写作。同时，策划工作全程介入写作进程，严格控制写作质量，用最专业的技术背景、最深刻的理论基础、最具代表性的案例、最能为专业读者接受的形式，为读者提供品质最佳的图书产品，体现了出版者和著作者的完美结合。

多年来，计算机图书研发部始终把创造社会效益摆在首位，秉承一切为国内计算机技术专业读者服务的精神，为推动国内 IT 技术发展、为体现国内技术的原创水平，穷尽所有的创意与努力，将出版者的命运与读者的支持紧紧地连在了一起。

在此，我们临出版之残酷竞争而不惧，旌旗猎猎而异军突起，这与广大读者的支持是分不开的。为使我们的脚步更坚实、使我们的队伍永葆活力和创造力，我们期待着您能为我们的前进贡献出您的意见和建议。同时，我们也在等待着您的加入。

我们的联系方式如下：

咨询电话：(010) 68134545 68131648

答疑邮件：[support@fecit.com.cn](mailto:support@fecit.com.cn)

网    址：<http://www.fecit.com.cn>   <http://www.fecit.net>

答疑网址：<http://www.fecit.com.cn/> “问题解答”专区

通用网址：计算机图书、FECIT、飞思教育、飞思科技、飞思

电子工业出版社计算机研发部

# 天子飞思

新世纪之初的北京，一群满怀共同理想的年轻人聚集在飞思教育产品研发中心的旗帜下，他们将新的希望和活力注入了中国IT教育产品开发领域。飞思人在为自己打造成为中国IT教育产品研发的精英团队而更加不懈努力。

21世纪的今天，飞思人在多元化教育产品的开发和出版等方面已经迈出了坚实的第一步，开拓出属于自己的一片天空，初步赢得了涓涓细流。

如今，本着教育为科技服务的宗旨，飞思科技产品研发中心以崭新的面貌等待您的支持与关注。

## 飞思人理念

我们经常感谢生活的慷慨，让我们这些原本并不同源的人得以同本，为了同一个梦想走到一起。

因为身处科技教育前沿，我们深感任重道远；因为伴随知识更新节奏，我们一刻不敢停歇。虽然我们年轻，但我们拥有：

“严谨、高效、协作”的团队精神

全方位、立体化的服务意识

实力雄厚的作者群和开发队伍

当然，最重要的是我们拥有：

恒久不变的理想和永不枯竭的激情和灵感

正因如此，我们敢于宣称：

飞思科技=丰富的内容+完美的形式



这也是我们共同精心培育的品牌 [www.feicit.com.cn](http://www.feicit.com.cn) 的承诺。

“问渠哪得清如许，为有源头活水来”。路再远，终需用脚去量；风景再美，终需自然抚育。

年轻的飞思人愿为清风细雨、阳光晨露，滋润您发芽、成长；更甘当坚实的铺路石，为您铺就成功之路。

# 前　　言

Jakarta Tomcat 服务器是在 SUN 公司的 JSWDK (JavaServer Web DevelopmentKit, SUN 公司推出的小型 Servlet/JSP 调试工具) 的基础上发展起来的一个优秀的 Java Web 应用容器, 它是 Apache-Jakarta 的一个子项目。Tomcat 被 JavaWorld 杂志的编辑选为 2001 年度最具创新的 Java 产品 (Most Innovative Java Product), 同时它又是 SUN 公司官方推荐的 Servlet/JSP 容器 (参见 <http://java.sun.com/products/jsp/tomcat/>), 因此它受到越来越多软件公司和开发人员的喜爱。Servlet 和 JSP 的最新规范都在 Tomcat 的新版本中得到了实现。

作为一个开放源码的软件, Tomcat 得到了开放源码志愿者的广泛支持, 它可以和目前大部分的主流 HTTP 服务器 (如 IIS 和 Apache 服务器) 一起工作, 而且运行稳定、可靠、效率高。

作者根据多年的 Java Web 开发经验, 详细阐明了在最新的 Tomcat 5.x 版本上开发 Java Web 应用涉及的各种技术, 并且介绍了如何将 Tomcat 和其他主流 HTTP 服务器集成并创建具有实用价值的企业 Java Web 应用的方案。

## 本书的组织结构和主要内容

本书的内容总体上分为两部分, 一部分介绍如何配置 Tomcat 服务器, 从而为 Java Web 应用创建高效的开发和运行环境; 还有一部分依据 SUN 的 Java Servlet 规范, 介绍了开发 Java Web 的各种技术。在组织结构上, 这两部分内容穿插在一起, 贯穿全书。本书按照由浅到深、前后呼应的顺序来安排内容。本书涉及以下内容。

### 1. Tomcat 的基础知识

包括如下内容:

- (1) Tomcat 服务器结构和安装步骤
- (2) Java Web 应用的结构和发布
- (3) 配置虚拟主机

### 2. Java Servlet、JavaServer Page 以及 Java Web 应用的基础知识

包括如下内容:

- (1) Servlet 的原理
- (2) 创建 Servlet 的基本步骤
- (3) JSP 语法
- (4) 在 Java Web 应用中访问数据库, 配置数据源
- (5) 在 Java Web 应用中访问 Java Bean, Java Bean 在不同范围内的生命周期
- (6) 在 Java Web 应用中使用 Session
- (7) 使用 ant 工具来管理 Web 应用

### 3. Java Web 开发高级技术

包括如下内容:

- (1) Servlet 过滤器

- (2) 创建自定义 JSP 标签
- (3) 网站的模板设计
- (4) 开发 Java Mail Web 应用，配置 Mail Session

#### 4. Tomcat 与当前其他通用软件的集成

包括如下内容：

- (1) Struts 和 MVC 设计模式
- (2) 使用 Log4J 进行日志操作
- (3) Tomcat 与 Jboss 服务器集成，创建 J2EE 应用
- (4) Tomcat 与 Apache SOAP 集成
- (5) Tomcat 与 Apache AXIS 集成
- (6) Tomcat 与其他 HTTP 服务器（如 Apache Web 服务器和 IIS Web 服务器）集成
- (7) Velocity 模板语言

#### 5. Tomcat 的高级功能

包括如下内容：

- (1) 持久性会话管理
- (2) Tomcat 的控制和管理平台
- (3) 安全域
- (4) Tomcat 阀
- (5) 创建嵌入式 Tomcat
- (6) 在 Tomcat 中配置 SSL

### 本书的范例程序

本书将 3 个 Web 应用的例子贯穿全书：

- helloapp 应用
- bookstore（网上书店）应用
- Java Mail Web（javamail）应用

#### 1. helloapp 应用

本书通过 helloapp 应用的例子来讲解 Java Web 开发的基础知识，比如，在第 2 章以 helloapp 应用为例，讲述了发布 Web 应用的步骤。在其他章节中，所有针对单个知识点的 Servlet 和 JSP 例子，都被发布到 helloapp 应用中。

#### 2. bookstore 应用

bookstore 应用是一个充分运用了所有 Java Web 开发技术的综合例子，它实现了一个网上书店，更加贴近于实际应用。为了便于读者循序渐进地掌握 Java Web 技术，在书中提供了 bookstore 应用的 4 个版本，它们分别侧重于某些技术。

(1) bookstore version0：通过这个例子读者可以进一步掌握 JSP 编程的技巧，能够灵活运用 Java Bean 和 Session，并掌握通过 JDBC 访问数据库的技术。

(2) bookstore version1：该例介绍如何在 Tomcat 中配置 JNDI DataSource，以及如何在 Web 应用中访问 JNDI DataSource。

(3) bookstore version2：使读者掌握创建 JSP 自定义标签的高级技术，并掌握对网页

进行模板设计的方法。

(4) bookstore version3：实现了基于 J2EE 架构的 bookstore 应用，并介绍了在 Jboss-Tomcat 的集成服务器上发布 J2EE 应用的方法。

### 3. javamail 应用

Java Mail Web 应用是一个基于 Web 的邮件客户程序，它向 Web 客户提供了访问 Mail 服务器上的邮件账号、进行收发信件和管理邮件夹等功能。通过这个例子，读者可以了解电子邮件的发送和接收协议，掌握 Java Mail API 的使用方法，以及通过 Java Mail API 创建 Java Mail Web 应用的过程。通过这个例子，读者还可以掌握在 Tomcat 中配置 Mail Session 的步骤，以及在 Web 应用中访问 Mail Session 的方法。

## 这本书是否适合你

本书面向所有打算或已经开发 Java Web 应用的读者。尽管本书在讲解 Java Web 技术时以 Tomcat 作为开发和运行平台，但由于 Java Web 技术是 SUN 公司在 Java Servlet 规范中提出的通用技术，因此本书讲解的应用例程可以运行在任何一个实现 SUN 的 Servlet 规范的 Java Web 服务器上。另一方面，由于 Tomcat 是 SUN 公司官方推荐的 Servlet/JSP 容器，因此在学习 Java Web 开发技术或进行实际的开发工作时，Tomcat 是首选的 Java Web 应用容器。

如果你是开发 Java Web 应用的新手，建议按照本书的先后顺序来学习。如果你已经在 Tomcat 上开发 Java Web 应用方面有丰富经验，则可以把本书作为实用的 Tomcat 技术参考资料。本书详细介绍了如何把 Tomcat 与当前其他通用的 Web 技术集成，以及 Tomcat 的各种高级功能。灵活运用本书介绍的 Tomcat 最新技术，将使 Java Web 应用开发更加得心应手。

实践是掌握 Java Web 技术最迅速、有效的办法。本书提供了大量典型的例子，在本书配套光盘上提供了完整的源代码，以及软件安装程序。本书所有程序都在 Tomcat 5.0.12 版本中测试通过，读者可以按照书上介绍的详细步骤亲自动手，在本地机器上配置 Tomcat 开发和运行环境，然后创建和发布 Java Web 应用。

## 光盘使用说明

本书配套光盘包含以下目录。

### 1. software 目录

在该目录下包含了本书涉及到的所有软件的最新版本的安装程序，包括：

- (1) JDK 的安装软件 (JDK1.4.2)
- (2) Tomcat 的安装软件 (Tomcat5.0.12)
- (3) MySQL 服务器的安装软件 (MySQL4.0.14)
- (4) Apache 服务器的安装软件 (Apache2.0.47)
- (5) Ant 的安装软件 (Ant1.5.4)
- (6) Apache SOAP 软件 (Apache SOAP2.3.1)
- (7) Apache AXIS 软件 (Apache AXIS1.1)
- (8) Log4J 软件 (Log4J1.2.8)

- (9) Struts 软件 (2003/09/03 发布)
- (10) Jboss 与 Tomcat 的集成软件 (Jboss-3.2.1\_tomcat-4.1.24)
- (11) Mail Server 的安装软件 (Merak Mail Server 试用版)
- (12) Velocity 软件 (Velocity 1.3.1)

## 2. lib 目录

在该目录下提供了运行与本书有关的 Java Web 应用, 以及配置 Tomcat 服务器时所需的 JAR 文件和 DLL 文件:

- (1) activation.jar (JavaBeans Activation Framework API)
- (2) mail.jar (Java Mail API)
- (3) log4j-1.2.8.jar (Log4J API)
- (4) soap.jar (SOAP API)
- (5) xerces.jar (Xerces API)
- (6) struts.jar (Struts API)
- (7) j2ee.jar (J2EE API)
- (8) isapi\_redirect.dll (Tomcat 与 IIS 集成的 JK 插件)
- (9) mod\_jk\_2.0.46.dll (Windows/NT 2000 下 Tomcat 与 Apache 集成的 JK 插件)
- (10) mod\_jk.so-ap2.0.46-rh72.46-rh72 (Linux 下 Tomcat 与 Apache 集成的 JK 插件)
- (11) jsp-api.jar (JSP API)
- (12) servlet-api.jar (Server API)
- (13) mysqldriver.jar (MySQL 驱动程序)

## 3. sourcecode 目录

在该目录下提供了本书所有的源程序, 每一章的源程序位于相应的 sourcecode/chapterX 目录下 (X 代表章节号)。bookstore 应用和 javamail 应用分别位于 sourcecode/bookstores 和 sourcecode/javamails 目录下。

本书在编写过程中得到了 Apache 软件组织和 SUN 公司在技术上的大力支持, 飞思科技产品研发中心负责监制工作, 此外曹文伟、潘玉峰和李红军为本书的编写提供了有益的帮助, 在此表示衷心的感谢! 尽管我们尽了最大努力, 但本书难免会有不妥之处, 欢迎各界专家和读者朋友批评指正, 我们的联系方式是:

电 话: (010) 68134545 68131648

电子邮件: support@fecit.com.cn tomcat\_author@yahoo.com.cn

飞思在线: <http://www.fecit.com.cn> <http://www.fecit.net>

答疑网址: <http://www.fecit.com.cn/question.htm>

通用网址: 计算机图书、飞思、飞思教育、飞思科技、FECIT

编 者

飞思科技产品研发中心

# 目 录

<b>第 1 章 Tomcat 简介</b> .....	1
1.1 Tomcat 与 Servlet 容器.....	1
1.2 Tomcat 的结构.....	2
1.3 Java Web 应用简介 .....	4
1.4 Tomcat 的工作模式.....	5
1.5 Tomcat 的版本.....	6
1.6 安装和配置 Tomcat 所需的资源.....	7
1.7 安装 Tomcat.....	7
1.8 测试 Tomcat 的安装.....	9
1.9 Tomcat 的运行脚本.....	11
1.10 小结.....	12
<b>第 2 章 创建和发布 Web 应用</b> .....	13
2.1 Tomcat 的目录结构.....	13
2.2 创建和发布 Web 应用.....	14
2.3 配置虚拟主机.....	25
2.4 小结.....	27
<b>第 3 章 Servlet 技术</b> .....	29
3.1 Servlet 简介 .....	29
3.2 Servlet API.....	29
3.3 Servlet 的生命周期 .....	32
3.4 HTTP 与 HttpServlet .....	33
3.5 创建 HttpServlet 的步骤 .....	37
3.6 ServletContext 和 Web 应用的关系.....	39
3.7 小结.....	42
<b>第 4 章 JSP 技术</b> .....	43
4.1 JSP 简介.....	43
4.2 JSP 语法.....	44
4.3 JSP 与 Cookie .....	50
4.4 转发 JSP 请求.....	53
4.5 JSP 异常处理.....	54
4.6 再谈部署 JSP.....	57
4.7 小结.....	57
<b>第 5 章 bookstore 应用简介</b> .....	59
5.1 bookstore 应用的软件结构 .....	59
5.2 浏览 bookstore 应用的 JSP 网页 .....	60

5.3 JavaBean 和实用类 .....	66
5.4 发布 bookstore 应用 .....	71
5.5 小结 .....	72
<b>第 6 章 访问数据库 .....</b>	<b>73</b>
6.1 安装和配置 MySQL 数据库 .....	73
6.2 通过 JDBC 访问数据库 .....	75
6.3 数据源 (DataSource) 简介 .....	89
6.4 配置数据源 .....	90
6.5 程序中访问数据源 .....	93
6.6 处理中文编码 .....	101
6.7 小结 .....	102
<b>第 7 章 Session 的使用与管理 .....</b>	<b>105</b>
7.1 Session 简介 .....	105
7.2 Session 范例程序 .....	107
7.3 Session 的跟踪 .....	111
7.4 Session 的持久化 .....	115
7.5 小结 .....	121
<b>第 8 章 访问 JavaBean .....</b>	<b>123</b>
8.1 JavaBean 简介 .....	123
8.2 JSP 访问 JavaBean 的语法 .....	124
8.3 JavaBean 的范围 .....	125
8.4 在 bookstore 应用中访问 JavaBean .....	128
8.5 小结 .....	134
<b>第 9 章 用 ant 工具管理 Web 应用 .....</b>	<b>135</b>
9.1 安装配置 ant .....	135
9.2 创建 build.xml 文件 .....	135
9.3 运行 ant .....	140
9.4 小结 .....	141
<b>第 10 章 Tomcat 的控制平台和管理平台 .....</b>	<b>143</b>
10.1 访问 Tomcat 的控制平台和管理平台 .....	143
10.2 Tomcat 的控制平台 .....	144
10.3 Tomcat 的管理平台 .....	148
10.4 小结 .....	150
<b>第 11 章 安全域 .....</b>	<b>151</b>
11.1 安全域概述 .....	151
11.2 为 Web 资源设置安全约束 .....	152
11.3 内存域 .....	158
11.4 JDBC 域 .....	160
11.5 DataSource 域 .....	162

11.6 在 Web 应用中访问用户信息 .....	165
11.7 小结 .....	166
<b>第 12 章 Tomcat 阀 .....</b>	<b>167</b>
12.1 Tomcat 阀简介 .....	167
12.2 客户访问日志阀 .....	167
12.3 远程地址过滤器 .....	169
12.4 远程主机过滤器 .....	170
12.5 客户请求记录器 .....	171
12.6 小结 .....	172
<b>第 13 章 Servlet 过滤器 .....</b>	<b>173</b>
13.1 Servlet 过滤器简介 .....	173
13.2 创建 Servlet 过滤器 .....	174
13.3 发布 Servlet 过滤器 .....	177
13.4 串联 Servlet 过滤器 .....	181
13.5 小结 .....	194
<b>第 14 章 自定义 JSP 标签 .....</b>	<b>195</b>
14.1 自定义 JSP 标签简介 .....	195
14.2 创建标签处理类 .....	196
14.3 创建标签库描述文件 .....	201
14.4 在 Web 应用中使用标签 .....	203
14.5 发布支持中、英文版本的 helloapp 应用 .....	206
14.6 小结 .....	208
<b>第 15 章 采用模板设计网上书店应用 .....</b>	<b>209</b>
15.1 如何设计网站的模板 .....	209
15.2 创建负责流程控制的 Servlet .....	210
15.3 创建模板标签和模板 JSP 文件 .....	212
15.4 修改 JSP 文件 .....	226
15.5 发布采用模板设计的 bookstore 应用 .....	227
15.6 小结 .....	231
<b>第 16 章 Struts 和 MVC 设计模式 .....</b>	<b>233</b>
16.1 MVC 设计模式简介 .....	233
16.2 Struts 实现的 MVC 设计模式 .....	234
16.3 创建采用 Struts 的 Web 应用 .....	238
16.4 运行 helloapp-struts 应用 .....	246
16.5 小结 .....	251
<b>第 17 章 使用 Log4J 进行日志操作 .....</b>	<b>253</b>
17.1 Log4J 简介 .....	253
17.2 Log4J 的基本使用方法 .....	257
17.3 在 helloapp 应用中使用 Log4J .....	262

17.4	小结.....	265
<b>第 18 章</b>	<b>Tomcat 与 Jboss 集成 .....</b>	<b>267</b>
18.1	安装 Jboss 和 Tomcat 整合服务器 .....	267
18.2	J2EE 体系结构简介 .....	268
18.3	创建 EJB 组件 .....	270
18.4	在 Web 应用中访问 EJB 组件 .....	275
18.5	发布 J2EE 应用 .....	277
18.6	小结.....	283
<b>第 19 章</b>	<b>开发 Java Mail Web 应用 .....</b>	<b>285</b>
19.1	E-mail 协议简介 .....	285
19.2	Java Mail API 简介 .....	287
19.3	Java Mail 应用程序开发环境 .....	288
19.4	创建 Java Mail 应用程序 .....	291
19.5	Java Mail Web 应用简介 .....	294
19.6	Java Mail Web 应用的程序结构 .....	295
19.7	在 Tomcat 中配置 Mail Session .....	318
19.8	发布和运行 javamail 应用 .....	321
19.9	小结.....	322
<b>第 20 章</b>	<b>Tomcat 与 Apache SOAP 集成 .....</b>	<b>323</b>
20.1	SOAP 简介 .....	323
20.2	建立 Apache SOAP 环境 .....	325
20.3	在 Tomcat 上发布 Apache-SOAP Web 应用 .....	326
20.4	创建 SOAP 服务 .....	327
20.5	管理 SOAP 服务 .....	329
20.6	创建和运行 SOAP 客户程序 .....	332
20.7	小结.....	335
<b>第 21 章</b>	<b>Tomcat 与 Apache AXIS 集成 .....</b>	<b>337</b>
21.1	建立 Apache AXIS 环境 .....	337
21.2	在 Tomcat 上发布 Apache-AXIS Web 应用 .....	337
21.3	创建 SOAP 服务 .....	338
21.4	管理 SOAP 服务 .....	340
21.5	创建和运行 SOAP 客户程序 .....	341
21.6	发布 JWS 服务 .....	344
21.7	小结.....	345
<b>第 22 章</b>	<b>Tomcat 与其他 HTTP 服务器集成 .....</b>	<b>347</b>
22.1	Tomcat 与 HTTP 服务器集成的原理 .....	347
22.2	在 Windows 下 Tomcat 与 Apache 服务器集成 .....	349
22.3	在 Linux 下 Tomcat 与 Apache 服务器集成 .....	353
22.4	Tomcat 与 IIS 服务器集成 .....	355

22.5 小结.....	362
<b>第 23 章 创建嵌入式 Tomcat 服务器.....</b>	<b>363</b>
23.1 将 Tomcat 嵌入 Java 应用.....	363
23.2 创建嵌入了 Tomcat 的 Java 示范程序.....	365
23.3 运行嵌入式 Tomcat 服务器.....	370
23.4 小结.....	372
<b>第 24 章 在 Tomcat 中配置 SSL.....</b>	<b>373</b>
24.1 SSL 简介.....	373
24.2 在 Tomcat 中使用 SSL.....	376
24.3 小结.....	380
<b>第 25 章 JSP 2.0 的新特征.....</b>	<b>381</b>
25.1 JSP 表达式语言.....	381
25.2 简单标签扩展.....	389
25.3 小结.....	393
<b>第 26 章 Velocity 模板语言.....</b>	<b>395</b>
26.1 安装 Velocity.....	395
26.2 Velocity 的简单例子.....	395
26.3 注释.....	399
26.4 引用.....	400
26.5 指令.....	405
26.6 其他特征.....	414
26.7 小结.....	415
<b>附录 A server.xml 文件.....</b>	<b>417</b>
A.1 配置 Server 元素 .....	420
A.2 配置 Service 元素 .....	420
A.3 配置 Engine 元素.....	420
A.4 配置 Host 元素 .....	421
A.5 配置 Context 元素 .....	422
A.6 配置 Connector 元素 .....	422
<b>附录 B web.xml 文件.....</b>	<b>425</b>
B.1 配置 Servlet 过滤器.....	427
B.2 配置 Servlet.....	428
B.3 配置 Servlet 映射.....	429
B.4 配置 Session.....	429
B.5 配置 Welcome 文件清单 .....	430
B.6 配置 Tag Library .....	430
B.7 配置资源引用 .....	430
B.8 配置安全约束 .....	431
B.9 配置安全验证登录界面 .....	432

B.10 配置对安全验证角色的引用 .....	432
<b>附录 C XML 简介 .....</b>	<b>433</b>
C.1 SGML、HTML 与 XML 的比较 .....	433
C.2 DTD 文档类型定义 .....	434
C.3 有效 XML 文档以及简化格式的 XML 文档 .....	435
C.4 XML 中的常用术语 .....	436

# 第1章 Tomcat简介

Jakarta Tomcat 服务器是在 SUN 公司的 JSWDK (JavaServer Web DevelopmentKit, 是 SUN 公司推出的小型 Servlet/JSP 调试工具) 的基础上发展起来的一个优秀的 Servlet/JSP 容器, 它是 Apache-Jakarta 软件组织的一个子项目。它不但支持运行 Servlet 和 JSP, 而且还具备了作为商业 Java Web 应用容器的特征。

作为一个开放源码的软件, Tomcat 得到了开放源码志愿者的广泛支持, 它可以和目前大部分的主流 HTTP 服务器 (如 IIS 和 Apache 服务器) 一起工作, 而且运行稳定、可靠、效率高。

Tomcat 服务器除了能够运行 Servlet 和 JSP, 还提供了作为 Web 服务器的一些特有功能, 如 Tomcat 管理和控制平台、安全域管理和 Tomcat 阀等。Tomcat 已成为目前开发企业 Java Web 应用的最佳选择之一。

本章简单介绍 Tomcat 作为 Servlet 容器的工作过程, 以及 Tomcat 的结构和工作模式, 最后介绍安装 Tomcat 服务器的步骤。

## 1.1 Tomcat 与 Servlet 容器

Jakarta Tomcat 服务器是一种 Servlet/JSP 容器。Servlet 是一种运行在支持 Java 语言的服务器上的组件。Servlet 最常见的用途是扩展 Java Web 服务器功能, 提供非常安全的、可移植的、易于使用的 CGI 替代品。它是一种动态加载的模块, 为来自 Web 客户的请求提供服务。它完全运行在 Java 虚拟机上。由于它在服务器端运行, 因此它的运行不依赖于浏览器。

Tomcat 作为 Servlet 容器, 负责处理客户请求, 把请求传送给 Servlet 并把结果返回给客户。Servlet 容器与 Servlet 之间的接口是由 Java Servlet API 定义的, 在 Java Servlet API 中定义了 Servlet 的各种方法, 这些方法在 Servlet 生命周期的不同阶段被 Servlet 容器调用; Servlet API 还定义了 Servlet 容器传递给 Servlet 的对象类, 如请求对象 ServletRequest 和响应对象 ServletResponse。

当客户请求访问某个 Servlet 时, Servlet 容器将创建一个 ServletRequest 对象和 ServletResponse 对象。在 ServletRequest 对象中封装了客户请求信息, 然后 Servlet 容器把 ServletRequest 对象和 ServletResponse 对象传给客户所请求的 Servlet。Servlet 把响应结果写到 ServletResponse 中, 然后由 Servlet 容器把响应结果传给客户。Servlet 容器响应客户请求的过程如图 1-1 所示。

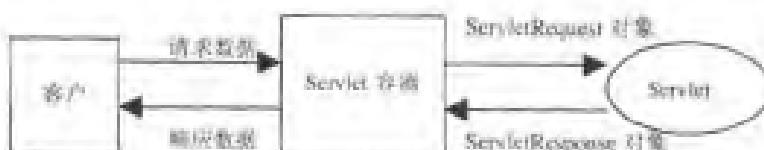


图 1-1 Servlet 容器响应客户请求的过程

本书将讨论 Tomcat 和 Java Web 开发的各种技术，如果要获取关于 Tomcat 的更多信息，可以访问 Tomcat 的主页 (<http://jakarta.apache.org/tomcat/index.html>)，如图 1-2 所示。



图 1-2 Tomcat 主页

## 1.2 Tomcat 的结构

Tomcat 服务器是由一系列可配置的组件构成，其中核心组件是 Catalina Servlet 容器，它是所有其他 Tomcat 组件的顶层容器。Tomcat 的组件可以在 <CATALINA\_HOME>/conf/server.xml 文件中进行配置，每个 Tomcat 组件在 server.xml 文件中对应一种配置元素。以下代码以 XML 的形式展示了各种 Tomcat 组件之间的关系：

```

<Server>
  <Service>
    <Connector />
    <Engine>
      <Host>
        <Context>
        </Context>
      </Host>
    </Engine>
  </Service>
</Server>

```

```
</Host>
</Engine>
</Service>
</Server>
```

在以上 XML 代码中，每个元素都代表一种 Tomcat 组件。这些元素可分为 4 类。

#### 1. 顶层类元素

顶层类元素包括`<Server>`元素和`<Service>`元素，它们位于整个配置文件的顶层。

#### 2. 连接器类元素

连接器类元素代表了介于客户与服务之间的通信接口，负责将客户的请求发送给服务器，并将服务器的响应结果传递给客户。

#### 3. 容器类元素

容器类元素代表处理客户请求并生成响应结果的组件，有 3 种容器类元素，它们是 Engine、Host 和 Context。Engine 组件为特定的 Service 组件处理所有客户请求，Host 组件为特定的虚拟主机处理所有客户请求，Context 组件为特定的 Web 应用处理所有客户请求。

#### 4. 嵌套类元素

嵌套类元素代表了可以加入到容器中的组件，如`<Logger>`元素、`<Valve>`元素和`<Realm>`元素，这些元素将在后面的章节做介绍。

下面，再对一些基本的 Tomcat 元素进行介绍。如果要了解这些元素的具体属性，可以参照本书附录 A（server.xml 文件）。

- `<Server>`元素

`<Server>`元素代表整个 Catalina Servlet 容器，它是 Tomcat 实例的顶层元素。`<Server>`元素中可包含一个或多个`<Service>`元素。

- `<Service>`元素

`<Service>`元素中包含一个`<Engine>`元素，以及一个或多个`<Connector>`元素，这些`<Connector>`元素共享同一个`<Engine>`元素。

- `<Connector>`元素

`<Connector>`元素代表和客户程序实际交互的组件，它负责接收客户请求，以及向客户返回响应结果。

- `<Engine>`元素

每个`<Service>`元素只能包含一个`<Engine>`元素。`<Engine>`元素处理在同一个`<Service>`中所有`<Connector>`元素接收到的客户请求。

- `<Host>`元素

一个`<Engine>`元素中可以包含多个`<Host>`元素。每个`<Host>`元素定义了一个虚拟主机，它可以包含一个或多个 Web 应用。

- `<Context>`元素

`<Context>`元素是使用最频繁的元素。每个`<Context>`元素代表了运行在虚拟主机上的单个 Web 应用。一个`<Host>`元素中可以包含多个`<Context>`元素。

**提示**

在本书中，Catalina 容器指的是<Engine>、<Host>或<Context>元素，它们都是容器类元素；Catalina Servlet 容器或者 Servlet 容器指的是<Server>元素，它代表了整个 Tomcat 服务器。

Tomcat 各个组件之间的嵌套关系如图 1-3 所示。

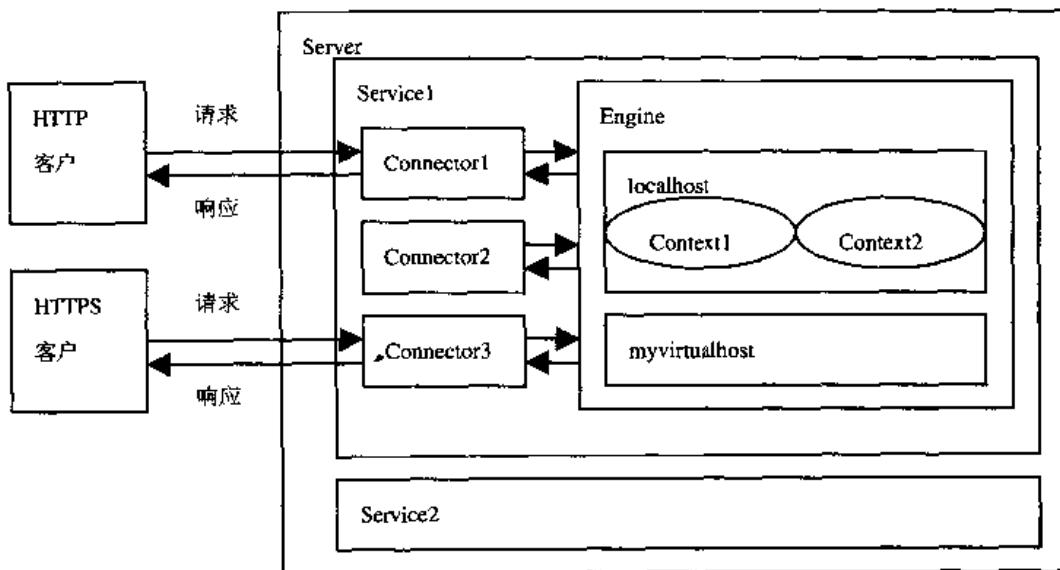


图 1-3 Tomcat 各个组件之间的嵌套关系

图 1-3 表明，Connector 负责接收客户的请求并向客户返回响应结果，在同一个 Service 中，多个 Connector 共享同一个 Engine。同一个 Engine 中可以有多个 Host，同一个 Host 中包含多个 Context。

### 1.3 Java Web 应用简介

Tomcat 服务器最主要的功能就是充当 Java Web 应用的容器。在 SUN 的 Java Servlet 规范中，对 Java Web 应用做了这样的定义：“Java Web 应用由一组 Servlet、HTML 页、类，以及其他可以被绑定的资源构成。它可以在各种供应商提供的实现 Servlet 规范的 Web 应用容器中运行。”在 Java Web 应用中可以包含如下内容：

- Servlet
- JSP
- 实用类
- 静态文档，如 HTML、图片等
- 客户端类
- 描述 Web 应用的信息（web.xml）

**提示**

可以说 Tomcat 服务器是 Servlet/JSP 容器，也可以说它是 Java Web 应用容器。这两种说法并不矛盾，因为构成 Java Web 应用的最主要的组件就是 Servlet 和 JSP。

Java Web 应用的主要特征之一就是它与 Context 的关系。每个 Web 应用有唯一的 Context。当 Java Web 应用运行时，Servlet 容器为每个 Web 应用创建唯一的 ServletContext 对象，它被同一个 Web 应用中所有的组件共享。

假定有两个 Web 应用分别为 helloapp 和 bookstore，两个客户分别访问如下 URL：

客户 1 访问的 URL 为：<http://localhost:8080/helloapp/index.htm>

客户 2 访问的 URL 为：<http://localhost:8080/bookstore/bookstore.jsp>

Tomcat 服务器的各个组件响应客户请求的过程如图 1-4 所示。在图 1-4 中，每个 Context 容器只对应一个 Java Web 应用。

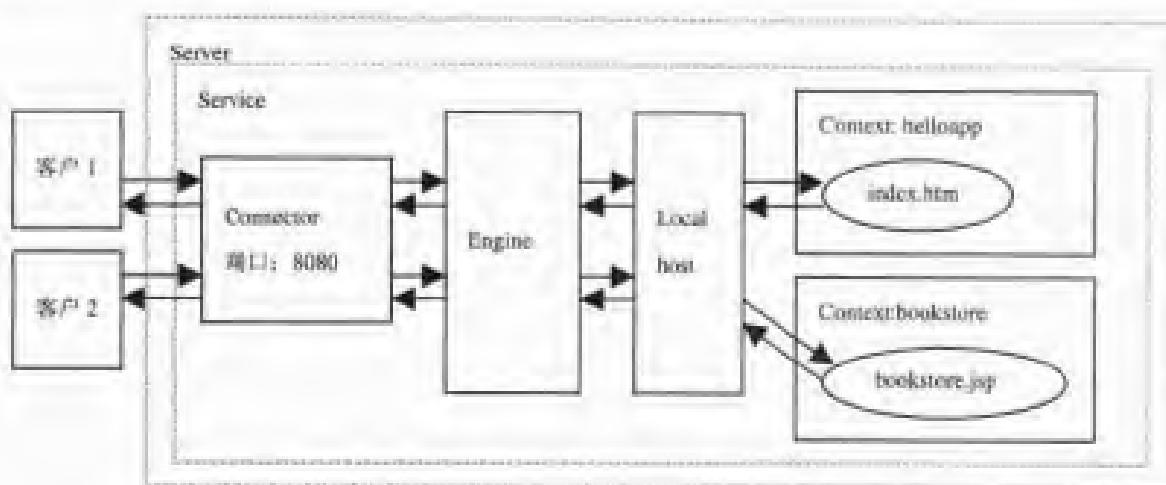


图 1-4 Tomcat 服务器的各个组件响应客户请求的过程

## 1.4 Tomcat 的工作模式

Tomcat 作为 Servlet 容器，有以下几种工作模式。

### 1. 独立的 Servlet 容器

在这种模式下，Tomcat 可以作为独立的 Java Web 服务器，Servlet 容器作为构成 Web 服务器的一部分而存在。独立的 Servlet 容器是 Tomcat 的默认模式。

### 2. 进程内的 Servlet 容器

Servlet 容器分为 Web 服务器插件和 Java 容器两部分。Web 服务器插件在其他 Web 服务器内部地址空间打开一个 Java 虚拟机（JVM, Java Virtual Machine），Java 容器在此 JVM 中运行 Servlet。如有客户端发出调用 Servlet 的请求，插件获得对此请求的控制并将它传递（使用 JNI 通信机制）给 Java 容器。进程内 Servlet 容器对于单进程、多线程的服务器非常合适，可以提供较高的运行速度，但缺乏伸缩性。



JNI (Java Native Interface) 指的是 Java 本地调用接口，通过这一接口，Java 程序可以和采用其他语言编写的本地程序进行通信。

### 3. 进程外的 Servlet 容器

Servlet 容器分为 Web 服务器插件和 Java 容器两部分。Web 服务器插件在其他 Web 服务器的外部地址空间打开一个 JVM，Java 容器在此 JVM 中运行 Servlet。如有客户端发出调用 Servlet 的请求，插件获得对此请求的控制并将它传递（采用 IPC 通信机制）给 Java 容器。进程外 Servlet 容器对客户请求的响应速度不如进程内容器，但进程外容器具有更好的伸缩性和稳定性。



IPC 是两个进程之间进行通信的一种机制。

Tomcat 既可作为独立的 Servlet 容器，也可和其他的 Web 服务器集成（当前支持 Apache、IIS 和 Netscape 服务器等）。作为进程内的 Servlet 容器或者进程外的 Servlet 容器，在本书的第 22 章将专门介绍 Tomcat 和其他 Web 服务器的集成方法。

## 1.5 Tomcat 的版本

随着 SUN 公司推出的 Servlet/JSP 规范的不断完善和升级，Tomcat 的版本也随之不断更新。Tomcat 和 Servlet/JSP 规范之间的版本对应关系参见表 1-1。

表 1-1 Tomcat 版本和 Servlet/JSP 规范

Servlet/JSP 规范	Tomcat 版本
2.4/2.0	3.0.x
2.3/1.2	4.1.27
2.2/1.1	3.3.1g

Tomcat 目前的最新版本是 Tomcat 5.0.x，它的前景被业界看好。Tomcat 4.1.x 是目前比较稳定和成熟的版本，已经被广泛用于实际 Web 应用的开发中。

下面分别介绍一下 Tomcat 5.0.x 和 Tomcat 4.1.x 的特点。

### 1. Tomcat 5.0.x

Tomcat 5.0.x 在 Tomcat 4.1 的基础上做了许多扩展和改进，它提供的新功能和新特性包括：

- 对服务器性能进一步优化，提高垃圾回收的效率
- 采用 JMX 技术来监控服务器的运行
- 提高了服务器的可扩展性和可靠性
- 增强了对 Tag Library 的支持
- 利用 Windows Wrapper 和 UNIX Wrapper 技术改进了与操作系统平台的集成
- 采用 JMX 技术来实现嵌入式的 Tomcat
- 提供了更完善的 Tomcat 文档



JMX( Java Management Extensions )是 SUN 提出的 Java 管理扩展规范，是一个为应用程序、设备和系统植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议。通过 JMX，用户可以灵活地开发无缝集成的系统、网络和服务管理应用。

## 2. Tomcat 4.1.x

Tomcat 4.0.x 完全废弃了 Tomcat 3.x 的架构，采用新的体系结构实现了 Catalina Servlet 容器。Tomcat 4.1.x 在 Tomcat 4.0.x 的基础上又进一步升级，它提供的新功能和新特性包括：

- 基于 JMX 的管理控制功能
- 实现了新的 Coyote Connector ( 支持 HTTP/1.1、AJP 1.3 和 JNDI )
- 重写了 Jasper JSP 编译器
- 提高了 Web 管理应用与开发工具的集成
- 提供客户化的 Ant 任务，使 Ant 程序根据 build.xml 脚本直接和 Web 管理应用交互

以上介绍中出现了很多新名词和术语，如 Tag Library、AJP 1.3 和 Ant，在本书后面章节会解释这些名词。

## 1.6 安装和配置 Tomcat 所需的资源

在安装和配置 Tomcat 之前，需要下载两个适用于操作系统的软件，参见表 1-2。

表 1-2 安装和配置 Tomcat 所需的软件

软件名称	下载地址	在本书配套光盘上的位置
Tomcat 5	<a href="http://jakarta.apache.org/builds/">http://jakarta.apache.org/builds/</a>	Windows NT/2000: software/jakarta-tomcat-5.0.12.zip 或 software/jakarta-tomcat-5.0.12.exe Linux: software/jakarta-tomcat-5.0.12.tar.gz
JDK 1.4	<a href="http://java.sun.com/j2se/1.4/">http://java.sun.com/j2se/1.4/</a>	Windows NT/2000: software/j2sdk-1_4_2-windows-i586.exe Linux: software/j2sdk-1_4_2-01-linux-i586.bin

对于 Windows 操作系统，Tomcat 5 提供了两种安装文件，一个文件为 jakarta-tomcat-5.x.exe，还有一个文件为 jakarta-tomcat-5.x.zip。jakarta-tomcat-5.x.exe 是可运行的安装程序，通过这个程序安装 Tomcat，会自动把 Tomcat 服务加入到 Windows 操作系统的服务中，并且在【开始】→【程序】菜单中加入 Tomcat 服务器管理菜单。在下面小节中介绍的安装方法采用的是 jakarta-tomcat-5.x.zip 文件，只要把它解压到本地硬盘即可。

## 1.7 安装 Tomcat

本节将把 Tomcat 安装为独立的 Web 服务器，也就是说，它的工作方式为第一种模式，即独立的 Servlet 容器。下面将分别介绍如何在 Windows NT/2000 以及 Linux 操作系统中安

装 Tomcat。

### 1. 在 Windows NT/2000 操作系统中安装 Tomcat

安装 Tomcat 之前，首先安装 JDK。假定把 JDK 安装在 C:\j2sdk1.4.2 目录下。

接下来，解压 Tomcat 压缩文件 jakarta-tomcat-5.x.zip，解压 Tomcat 的压缩文件的过程就相当于安装的过程。假定解压至 C:\jakarta-tomcat 目录。

然后，需要设定两个环境变量：JAVA\_HOME，它是 JDK 的安装目录； CATALINA\_HOME，它是 Tomcat 的安装目录。

在 Windows NT/2000 操作系统中设置环境变量的步骤如下。

步骤

(1) 打开“控制面板”，选择“系统”图标。

(2) 双击“系统”图标，运行 Windows NT/2000 系统程序，选择【高级】标签，如图 1-5 所示。

(3) 在图 1-5 中单击【环境变量】按钮，将会出现设置环境变量的窗口，如图 1-6 所示。



图 1-5 Windows NT/2000 系统程序



图 1-6 “环境变量”对话框

(4) 单击“系统变量”区域的【新建】按钮，将会出现“新建系统变量”对话框，在对话框中新建 JAVA\_HOME 环境变量，环境变量值为 C:\j2sdk1.4.2，如图 1-7 所示。



图 1-7 设置 JAVA\_HOME 环境变量

(5) 重复步骤(4), 新建 CATALINA\_HOME 环境变量, 环境变量值为 C:\jakarta-tomcat。

## 2. 在 Linux 操作系统中安装 Tomcat

安装 Tomcat 之前, 首先安装 JDK。假定把 JDK 安装在 /home/java/j2sdk1.4.2 目录下。接下来, 解压 Tomcat 压缩文件, 假定解压至 /home/tomcat 目录。

设定两个环境变量: JAVA\_HOME 和 CATALINA\_HOME。在 Linux 操作系统中设置环境变量, 需要根据 Linux 采用的 SHELL 类型, 采用不同的设置命令, 参见表 1-3 和表 1-4。

表 1-3 设置 JAVA\_HOME 环境变量

SHELL 类型	设置 JAVA_HOME 环境变量的命令
bash	JAVA_HOME=/home/java/j2sdk1.4.2; export JAVA_HOME
sh	setenv JAVA_HOME /home/java/j2sdk1.4.2

表 1-4 设置 CATALINA\_HOME 环境变量

SHELL 类型	设置 CATALINA_HOME 环境变量的命令
bash	CATALINA_HOME=/home/tomcat; export CATALINA_HOME
sh	setenv CATALINA_HOME /home/tomcat

为了便于编译和运行本书后面章节的 Java 程序, 可以把 Java 编译器和解释器所在的目录<JAVA\_HOME>/bin 添加到环境变量 PATH 中。

## 1.8 测试 Tomcat 的安装

要测试 Tomcat 的安装, 必须先启动 Tomcat 服务器。如果在 Windows NT/2000 下直接通过 Tomcat 5 的专门安装程序 jakarta-tomcat-5.x.exe 安装 Tomcat, 则可以从 Windows 的【开始】菜单中启动或关闭 Tomcat 服务器。此外, 还可以通过运行批处理文件启动 Tomcat 服务器。表 1-5 显示了在 Windows NT/2000 以及 Linux 下分别启动和关闭 Tomcat 服务器的批处理文件, 运行这些批处理文件, 即可启动或关闭 Tomcat 服务器。

表 1-5 启动和关闭 Tomcat 服务器的批处理文件

操作系统	启动	关闭
Windows NT/2000	<CATALINA_HOME>/bin/startup.bat	<CATALINA_HOME>/bin/shutdown.bat
Linux	<CATALINA_HOME>/bin/startup.sh	<CATALINA_HOME>/bin/shutdown.sh



如果在 Tomcat 5 安装后没有设定环境变量 CATALINA\_HOME, Tomcat 启动脚本会自动设置 CATALINA\_HOME, 其值为当前的 Tomcat 安装目录。

Tomcat 服务器启动后, 就可以通过浏览器访问以下 URL:

<http://localhost:8080/>

将出现如图 1-8 所示的页面。

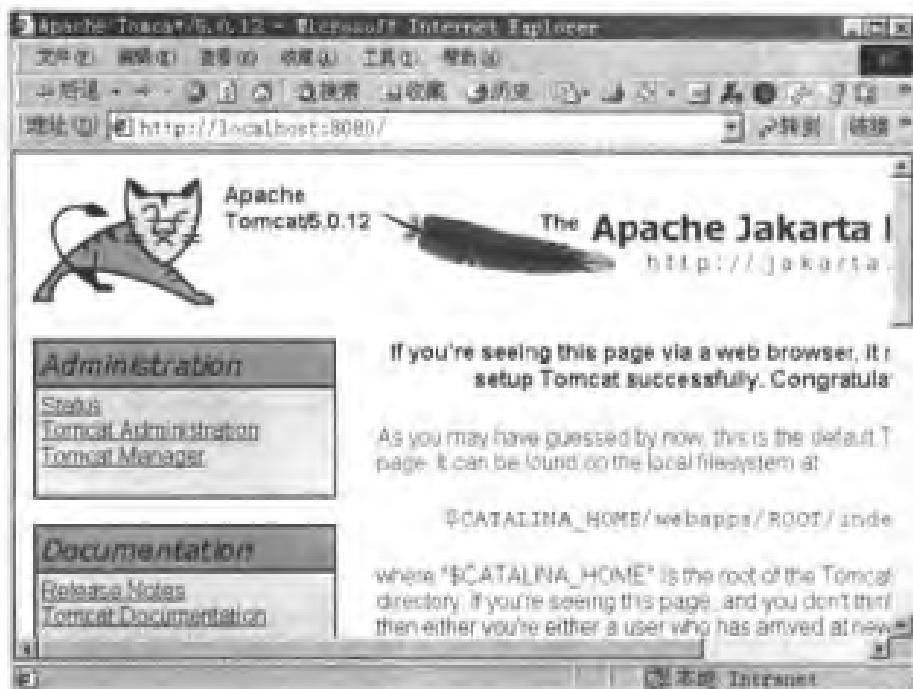


图 1-8 Tomcat 的默认主页

Tomcat 服务器采用的 HTTP 端口为“8080”，如果想采用默认的 HTTP 端口“80”，可以修改<CATALINA\_HOME>/conf/server.xml，将<Connector>元素的 port 属性值改为“80”，然后重启 Tomcat 服务器。修改成功后，通过 URL 路径 <http://localhost> 就可以访问如图 1-8 所示的页面。

#### 修改前的<Connector>元素

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="8080" minProcessors="5" maxProcessors="100"
    enableLookups="true" redirectPort="8443" acceptCount="100"
    debug="0" connectionTimeout="20000"
    disableUploadTimeout="true" />
</Connector>
```

#### 修改后的<Connector>元素

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 80 -->
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="80" minProcessors="5" maxProcessors="100"
    enableLookups="true" redirectPort="8443" acceptCount="100"
    debug="0" connectionTimeout="20000"
    disableUploadTimeout="true" />
</Connector>
```

接下来我们将通过运行 Tomcat 服务器提供的 JSP 例子，来测试 JDK 的安装是否成功。首先访问 URL 路径 <http://localhost:8080/jsp-examples/>，如图 1-9 所示。

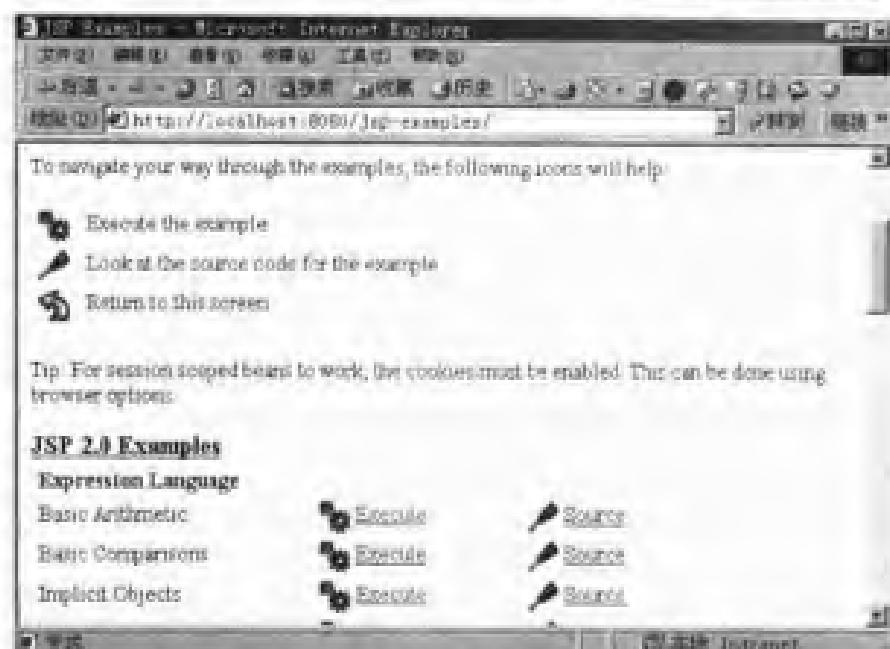


图 1-9 JSP 例子网页

执行 Date 的例子，将会出现类似如图 1-10 所示的结果，显示当前日期。

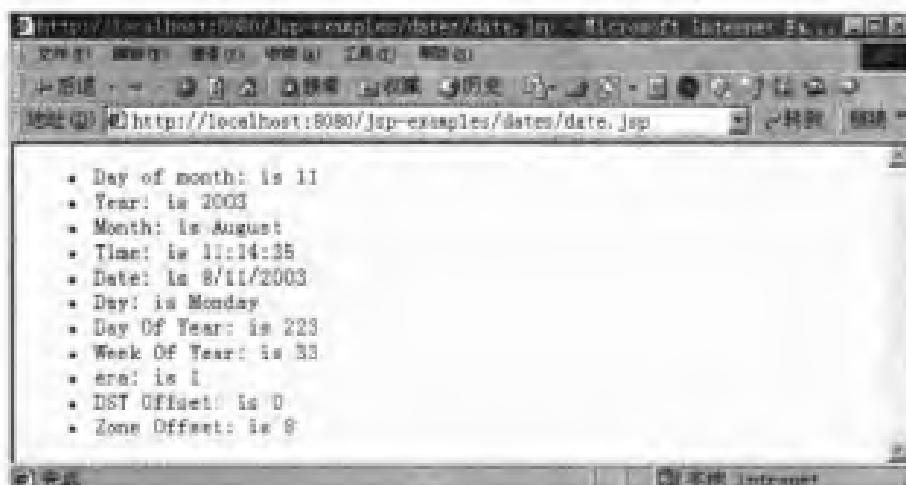


图 1-10 JSP 例子 Date 网页

如果 Date 的例子执行失败，再检查 JAVA\_HOME 环境变量的设置，应确保它的值与 JDK 安装目录一致。

## 1.9 Tomcat 的运行脚本

如果仔细研究一下 Tomcat 启动和关闭脚本（以 Windows NT/2000 操作系统为例），会发现 startup.bat 和 shutdown.bat 都执行同一目录下的 catalina.bat 脚本。catalina.bat 脚本允许输入命令行参数，catalina.bat 的使用方法参见表 1-6。

表 1-6 catalina.bat 的使用方法

命令行参数	描 述
start	在新的 DOS 窗口启动 Tomcat 服务器
run	在当前 DOS 窗口启动 Tomcat 服务器
debug	在调试模式下启动 Tomcat 服务器
embedded	在嵌入模式下启动 Tomcat 服务器
stop	关闭 Tomcat 服务器

执行 startup.bat 脚本，相当于执行了 catalina start 命令；执行 shutdown.bat 脚本，相当于执行了 catalina stop 命令；在开发和调试阶段，运行 catalina run 命令更有利于查看 Tomcat 服务器启动时的出错信息。在某些情况下，如果配置的 server.xml 文件有错（最常见的是语法错误，导致 org.xml.sax.SAXParseException 异常），可能会导致 Tomcat 服务器启动失败，而且没有在文件系统留下任何日志信息。如果运行 catalina start 命令，Tomcat 服务器在一个独立的 DOS 窗口中启动，一旦启动失败，这个 DOS 窗口就立刻自动关闭，程序运行中输出的出错信息也随之消失；如果运行 catalina run 命令，Tomcat 服务器在当前 DOS 窗口中启动，一旦启动失败，仅仅是 Tomcat 启动程序异常终止，在当前 DOS 窗口中仍保留了运行时的出错信息，便于查找启动失败原因。

还有一个值得注意的地方是，尽管在 catalina.bat 中也设置了 classpath，但是 Tomcat 的类装载器并不从这个 classpath 中加载类文件，因此修改 catalina.bat 中的 classpath 没有实际意义。关于 Tomcat 的类装载器可以参考 Tomcat 的有关文档：

<CATALINA\_HOME>/webapps/tomcat-docs/class-loader-howto.html



Tomcat 安装软件中附带了详细的文档，在安装好 Tomcat 以后，文档以 Web 应用的形式存放在<CATALINA\_HOME>/webapps/tomcat-docs 目录下。

## 1.10 小 结

本章介绍了 Jakarta Tomcat 服务器的概念和主要功能。我们简单讨论了 Java Web 应用，这是 Tomcat 服务器的工作核心。接着，讲解了在 Windows NT/2000 和 Linux 下安装 Tomcat 服务器的步骤，还介绍了测试 Tomcat 服务器是否安装成功的方法。

# 第 2 章 创建和发布 Web 应用

本章介绍如何在 Tomcat 上创建和发布 Web 应用。这里首先讲解 Tomcat 的目录结构以及 Web 应用的目录结构，接着介绍如何将 HTML、Servlet、JSP 和 Tag Library 部署到 Web 应用中，然后介绍把整个 Web 应用打包并发布的方法，最后介绍如何在 Tomcat 上配置虚拟主机。

本章侧重于讨论 Web 应用的结构和发布方法，所以没有对本章的 Servlet 和 JSP 的例子进行详细解释，关于 Servlet 和 JSP 的技术可以分别参考第 3 章和第 4 章的内容。

## 2.1 Tomcat 的目录结构

在 Tomcat 上发布 Web 应用之前，首先要了解 Tomcat 的目录结构。Tomcat 的目录结构参见表 2-1，这些目录都是<CATALINA\_HOME>的子目录。

表 2-1 Tomcat 的目录结构

目 录	描 述
/bin	存放 Windows 平台以及 Linux 平台上启动和关闭 Tomcat 的脚本文件
/conf	存放 Tomcat 服务器的各种配置文件，其中最重要的配置文件是 server.xml
/server	包含 3 个子目录：classes、lib 和 webapps
/server/lib	存放 Tomcat 服务器所需的各種 JAR 文件
/server/webapps	存放 Tomcat 自带的两个 Web 应用：admin 应用和 manager 应用
/common/lib	存放 Tomcat 服务器以及所有 Web 应用都可以访问的 JAR 文件
/shared/lib	存放所有 Web 应用都可以访问的 JAR 文件
/logs	存放 Tomcat 的日志文件
/webapps	当发布 Web 应用时，默认情况下把 Web 应用文件放于此目录下
/work	Tomcat 把由 ISP 生成的 Servlet 放于此目录下

从表 2-1 可以看出，在/server/lib 目录、/common/lib 和/shared/lib 目录下都可以放 JAR 文件，它们的区别在于：

- 在/server/lib 目录下的 JAR 文件只可被 Tomcat 服务器访问
- 在/shared/lib 目录下的 JAR 文件可以被所有的 Web 应用访问，但不能被 Tomcat 服务器访问
- 在/common/lib 目录下的 JAR 文件可以被 Tomcat 服务器和所有 Web 应用访问

此外，对于下面将要介绍的 Java Web 应用，在它的 WEB-INF 目录下，也可以建立 lib 子目录，在 lib 子目录下可以放各种 JAR 文件，这些 JAR 文件只能被当前 Web 应用访问。



在以上提到的 lib 目录下都只接受 JAR 文件，如果类压缩文件为 ZIP 文件，应该将它展开，重新打包为 JAR 文件再拷贝到 lib 目录中。如果直接将 ZIP 文件拷贝到 lib 目录，则会发现 Tomcat 服务器仍然找不到相关的类。打包命令参见 2.2.8 节。

## 2.2 创建和发布 Web 应用

Java Web 应用由一组静态 HTML 页、Servlet、JSP 和其他相关的 class 组成。每种组件在 Web 应用中都有固定的存放目录。Web 应用的配置信息存放在 web.xml 文件中。在发布某些组件（如 Servlet）时，必须在 web.xml 文件中添加相应的配置信息。

### 2.2.1 Web 应用的目录结构

Web 应用具有固定的目录结构，这里假定开发一个名为 helloapp 的 Web 应用。首先，应该在<CATALINA\_HOME>/webapps 目录下创建这个 Web 应用的目录结构，参见表 2-2。

表 2-2 Web 应用的目录结构

目 录	描 述
/helloapp	Web 应用的根目录。所有的 JSP 和 HTML 文件都存放在此目录下
/helloapp/WEB-INF	存放 Web 应用的发布描述文件 web.xml
/helloapp/WEB-INF/classes	存放各种 class 文件。Servlet 类文件也放于此目录下
/helloapp/WEB-INF/lib	存放 Web 应用所需的的各种 JAR 文件。例如，在这个目录下，可以存放 JDBC 驱动程序的 JAR 文件

从表 2-2 中，我们看到在 classes 以及 lib 子目录下，都可以存放 Java 类文件。在运行过程中，Tomcat 的类装载器先装载 classes 目录下的类，再装载 lib 目录下的类。因此，如果两个目录下存在同名的类，classes 目录下的类具有优先权。

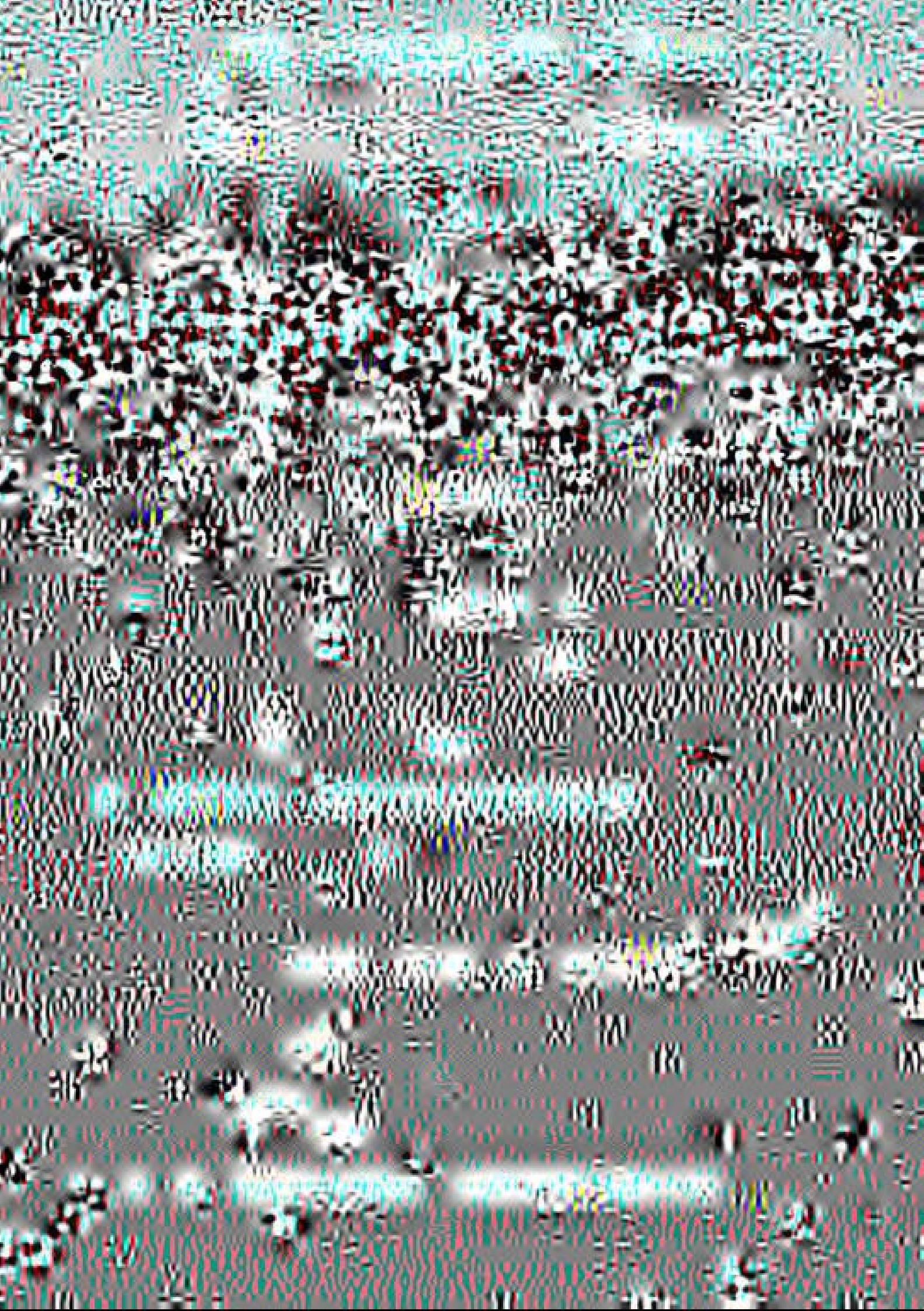
本章介绍的 helloapp 应用的目录结构如图 2-1 所示，helloapp 应用在 Windows 资源管理器中的展开如图 2-2 所示。

在 helloapp 应用中创建了如下组件：

- HTML 组件：index.htm
- JSP 组件：login.jsp 和 hello.jsp
- Servlet 组件：DispatcherServlet

这些组件之间的链接关系为：

index.htm → login.jsp → DispatcherServlet → hello.jsp



```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

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

以上 web.xml 文件的第一行指定了 XML 的版本和字符编码，第二行 DOCTYPE 指定文档类型，接下来声明了一个<web-app>元素，所有关于 Web 应用的配置元素都将加入到这个<web-app>元素中。

### 2.2.3 在 server.xml 中加入<Context>元素

<Context>元素是<CATALINA\_HOME>/conf/server.xml 中使用最频繁的元素，它代表了运行在<Host>上的单个 Web 应用。一个<Host>中可以有多个<Context>元素。每个 Web 应用必须有惟一的 URL 路径，这个 URL 路径在<Context>元素的 path 属性中设定。

例如，在名为“localhost”的<Host>元素中加入如下<Context>元素：

```
<!-- Define the default virtual host -->
<Host name="localhost" debug="0" appBase="webapps"
unpackWARs="true" autoDeploy="true">

  <!--
  -->
  <Context path="/helloapp" docBase="helloapp" debug="0"
reloadable="true"/>

</Host>
```

Context 元素的各个属性的说明参见表 2-3。

表 2-3 Context 元素的属性

属性	描述
path	指定访问该 Web 应用的 URL 入口
docBase	指定 Web 应用的文件路径，可以指定绝对路径，也可以指定相对于 Host 的 appBase 属性的相对路径（关于 Host 的 appBase 属性参见 2.3 节）。如果 Web 应用采用开放目录结构，则指定 Web 应用的根目录；如果 Web 应用是个 WAR 文件，则指定 WAR 文件的路径
reloadable	如果这个属性设为 true，Tomcat 服务器在运行状态下会监视在 WEB-INF/classes 和 WEB-INF/lib 目录下 class 文件的改动。如果监测到有 class 文件被更新，服务器会自动重新加载 Web 应用



在开发阶段，将 reloadable 属性设为 true，有助于调试 Servlet 和其他的 class 文件。但是由于这一功能会加重服务器的运行负荷，因此建议在 Web 应用的产品发布阶段，将这个属性设为 false。

## 2.2.4 部署 HTML 文件

在 helloapp 目录下加入 index.htm 文件，这个文件仅仅用来显示一串带链接的字符串“Welcome to HelloApp”，它链接到 login.jsp 文件。以下是 index.htm 文件的代码：

```
<html>
  <head>
    <title>helloapp</title>
  </head>
  <body>
    <p><font size="7">Welcome to HelloApp</font></p>
    <p><a href="login.jsp?language=English">English version </a>
  </body>
</html>
```

访问 index.htm 的 URL 为 <http://localhost:8080/helloapp/index.htm>，该页面的显示结果如图 2-3 所示。



图 2-3 index.htm

## 2.2.5 部署 JSP

接下来，创建两个 JSP 文件，其中一个是 login.jsp（参见例程 2-1），它显示登录页面，要求输入用户名和口令，这个页面链接到一个名为 DispatcherServlet 的 Servlet。还有一个 JSP 文件是 hello.jsp（参见例程 2-2），这个 JSP 被 DispatcherServlet 调用，显示 Hello 页面。JSP 的语法将在第 4 章详细讨论，本节侧重于介绍 JSP 的发布过程。这两个 JSP 文件都应放在 helloapp 目录下。

例程 2-1 login.jsp

---

```
<html>
  <head>
    <title>helloapp</title>
```

```
</head>
<body>
    <br>
    <form name="loginForm" method="post" action="dispatcher">
        <table>
            <tr>
                <td><div align="right">User Name:</div></td>
                <td><input type="text" name="username"></td>
            </tr>
            <tr>
                <td><div align="right">Password:</div></td>
                <td><input type="password" name="password"></td>
            </tr>
            <tr>
                <td></td>
                <td><input type="Submit" name="Submit" value="Submit"></td>
            </tr>
        </table>
    </form>
</body>
</html>
```

例程 2-2 hello.jsp

```
<html>
<head>
    <title>helloapp</title>
</head>
<body>
    <b>Welcome: <%= request.getAttribute("USER") %></b>
</body>
</html>
```

login.jsp 中生成了一个 loginForm 表单，它有两个字段：username 和 password。访问 login.jsp 的 URL 为 <http://localhost:8080/helloapp/login.jsp>，它生成的页面如图 2-4 所示。

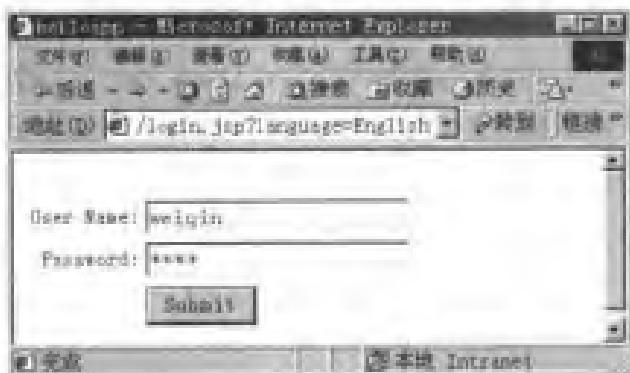


图 2-4 login.jsp 网页

## 2.2.6 部署 Servlet

下面，创建一个 Servlet 文件，名为 DispatcherServlet.java（参见例程 2-3），它调用 HttpServletRequest 对象的 getParameter 方法读取客户提交的 loginForm 表单数据，获取用户名和口令，然后将用户名和口令保存在 HttpServletRequest 对象的属性中，再把请求转发给 hello.jsp。

例程 2-3 DispatcherServlet.java

```
package mypack;

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

public class DispatcherServlet extends HttpServlet {

    private String target = "/hello.jsp";

    public void init(ServletConfig config)
        throws ServletException {
        super.init(config);
    }

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

        // If it is a get request forward to doPost()
        doPost(request, response);
    }

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

        // Get the username from the request
        String username = request.getParameter("username");
        // Get the password from the request
        String password = request.getParameter("password");

        // Add the user to the request
        request.setAttribute("USER", username);
        request.setAttribute("PASSWORD", password);
    }
}
```

```

// Forward the request to the target named
ServletContext context = getServletContext();

System.out.println("Redirecting to " + target);
RequestDispatcher dispatcher =
    context.getRequestDispatcher(target);
dispatcher.forward(request, response);
}

public void destroy() {
}
}

```

编译并发布 DispatcherServlet 的步骤如下。

### 步骤

(1) 编译 DispatcherServlet.java。编译时，需要将 Java Servlet API 的 JAR 文件 (servlet-api.jar) 设置为 classpath, servlet-api.jar 文件位于<CATALINA\_HOME>/common/lib 目录下。

(2) 把编译出来的 class 文件拷贝到 /helloapp/WEB\_INF/classes 目录下。DispatcherServlet.class 的存放位置为 /helloapp/WEB\_INF/classes/mypack/DispatcherServlet.



在本例中，声明将 DispatcherServlet 类放在包 mypack 下，所以应该在 /WEB\_INF/classes 目录下先创建子目录/mypack，然后在子目录下放 DispatcherServlet.class 文件。

(3) 接下来在 web.xml 中为 DispatcherServlet 类加上<servlet>和<servlet-mapping>元素。

<web-app>

```

<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>mypack.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/dispatcher</url-pattern>
</servlet-mapping>

```

</web-app>

<servlet>元素的属性描述参见表 2-4。

在本例配置中，没有为 DispatcherServlet 设置 load-on-startup 属性，因此当 Web 应用启动时，Servlet 容器不会加载这个 Servlet，只有当 Web 客户首次访问这个 Servlet 时才加载它。

表 2-4 &lt;servlet&gt;元素的属性

属性	说 明
<servlet-name>	定义 Servlet 的名字
<servlet-class>	指定实现这个 Servlet 的类
<init-param>	定义 Servlet 的初始化参数（包括参数名和参数值），一个<servlet>元素中可以有多个<init-param>
<load-on-startup>	指定当 Web 应用启动时，装载 Servlet 的次序。当这个值为正数或零，Servlet 将最先加载数值小的 Servlet，再依次加载其他数值大的 Servlet。如果这个值为负数或者没有设定，那么 Servlet 容器将在 Web 客户首次访问这个 Servlet 时加载它

<servlet-mapping>元素用来指定<servlet-name>和<url-pattern>映射。<url-pattern>是指访问 Servlet 的相对 URL 路径。

根据以上<url-pattern>属性，访问 DispatcherServlet 的 URL 为 http://localhost:8080/helloapp/dispatcher。DispatcherServlet 接受到客户请求后，再把请求转发给 hello.jsp，hello.jsp 生成的页面如图 2-5 所示。



图 2-5 DispatcherServlet 调用 hello.jsp 生成的网页

### 2.2.7 部署 JSP Tag Library

最后，在 Web 应用中加入 Tag Library（标签库）。Tag Library 向用户提供了自定义 JSP 标签的功能。我们将定义一个名为 mytaglib 的标签库，它包含了一个简单的 hello 标签，这个标签能够将 JSP 页面中所有的<mm:hello/>解析为字符串“hello”。

以下是创建和发布 mytaglib 标签库的步骤。

#### 步骤

(1) 编写用于处理 hello 标签的类 HelloTag.java，例程 2-4 列出了 HelloTag.java 的源代码。

例程 2-4 HelloTag.java

```
package mypack;
```

```
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;

public class HelloTag extends TagSupport
{
    public void HelloTag() { }

    // Method called when the closing hello tag is encountered
    public int doEndTag() throws JspException {

        try {

            // We use the pageContext to get a Writer
            // We then print the text string Hello
            pageContext.getOut().print("Hello");
        }
        catch (Exception e) {

            throw new JspTagException(e.getMessage());
        }
        // We want to return SKIP_BODY because this Tag does not support
        // a Tag Body
        return SKIP_BODY;
    }

    public void release() {

        // Call the parent's release to release any resources
        // used by the parent tag.
        // This is just good practice for when you start creating
        // hierarchies of tags.
        super.release();
    }
}
```

编译 HelloTag.java 时，需要将 jsp-api.jar 文件添加到 classpath 中，这个 JAR 文件位于 <CATALINA\_HOME>/common/lib 目录下。编译生成的 HelloTag.class 存放位置为 /WEB-INF/classes/mypack/HelloTag.class。

(2) 创建 Tag Library 的描述文件 mytaglib.tld 文件，在这个文件中定义 mytaglib 标签库和 hello 标签。这个文件存放位置为 /WEB-INF/mytaglib.tld。例程 2-5 列出了 mytaglib.tld 的源代码。

例程 2-5 mytaglib.tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```

<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>mytaglib</shortname>
  <uri>/mytaglib</uri>

  <tag>
    <name>hello</name>
    <tagclass>mypack.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Just Says Hello</info>
  </tag>

</taglib>

```

(3) 在 web.xml 文件中加入<taglib>元素, 例程 2-6 列出了修改后的 web.xml 文件。

例程 2-6 加入<taglib>元素的 web.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
'//Sun Microsystems, Inc./DTD Web Application 2.3//EN'
'http://java.sun.com/j2ee/dtds/web-app_2_3.dtd'>

<web-app>

  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>mypack.DispatcherServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/dispatcher</url-pattern>
  </servlet-mapping>

  <taglib>
    <taglib-uri>/mytaglib</taglib-uri>
    <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
  </taglib>

```

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

<taglib> 中包含两个属性<taglib-uri> 和 <taglib-location>。其中 <taglib-uri> 指定 Tag Library 标示符；<taglib-location> 指定 Tag Library 的描述文件（TLD）的位置。

(4) 在 hello.jsp 文件中加入 hello 标签。首先，在 hello.jsp 中加入引用 mytaglib 的 taglib 指令：

```
<%@ taglib uri="/mytaglib" prefix="mm" %>
```

以上 taglib 指令中，prefix 用来指定引用 mytaglib 标签库时的前缀，修改后的 hello.jsp 文件参见例程 2-7。

例程 2-7 加入 Tag 标签的 hello.jsp

```
<%@ taglib uri="/mytaglib" prefix="mm" %>
<html>
<head>
    <title>helloapp</title>
</head>
<b><mm:hello/> : <%= request.getAttribute("USER") %></b>
</body>
</html>
```

hello.jsp 修改后，再依次访问 index.htm → login.jsp → DispatcherServlet → hello.jsp，最后生成的网页如图 2-6 所示。



图 2-6 带 hello 标签的 hello.jsp 生成的网页

## 2.2.8 创建并发布 WAR 文件

Tomcat 既可以运行采用开放式目录结构的 Web 应用，也可以运行 WAR 文件。在本书配套光盘的 sourcecode/chapter2/helloapp 目录下提供了所有源文件，只要把整个 helloapp 目录拷贝到 <CATALINA\_HOME>/webapps 目录下，即可运行开放式目录结构的 helloapp 应用。

在 Web 应用的开发阶段，为了便于调试，通常采用开放式的目录结构来发布 Web 应用，这样可以方便地更新或替换文件。如果开发完毕，进入产品发布阶段，应该将整个 Web 应用打包为 WAR 文件，再进行发布。

在本例中，按如下步骤发布 helloapp。

### 步骤

(1) 进入 helloapp 应用的根目录<CATALINA\_HOME>/webapps/helloapp。

(2) 把整个 Web 应用打包为 helloapp.war 文件，命令如下：

```
jar cvf helloapp.war /*
```



**提示** 在 JDK 的 bin 目录下提供了打包程序 jar.exe。如果要展开 helloapp.war 文件，命令为：jar xvf helloapp.war。

(3) 把 helloapp.war 文件拷贝到<CATALINA\_HOME>/webapps 目录下。

(4) 删除原先的 helloapp 目录。

(5) 启动 Tomcat 服务器。

Tomcat 服务器启动时，会把 webapps 目录下的所有 WAR 文件自动展开为开放式的目录结构。所以服务器启动后，会发现服务器把 helloapp.war 展开到<CATALINA\_HOME>/webapps/helloapp 目录中。

## 2.3 配置虚拟主机

在 Tomcat 的配置文件 server.xml 中，Host 元素代表虚拟主机，在同一个 Engine 元素下可以配置多个虚拟主机。例如，有两个公司的 Web 应用都发布在同一个 Tomcat 服务器上，可以为每家公司分别创建一个虚拟主机，它们的虚拟主机名分别为：

www.mycompany1.com

www.mycompany2.com

这样当 Web 客户访问以上两个 Web 应用时，就好像这两个应用分别拥有各自的主机。此外，还可以为虚拟主机建立别名，例如，如果希望 Web 客户访问 www.mycompany1.com 或 mycompany1.com 都能连接到同一个 Web，那么可以把 mycompany1.com 作为虚拟主机的别名来处理。

下面讲解如何配置 www.mycompany1.com 虚拟主机。

### 步骤

(1) 打开<CATALINA\_HOME>/conf/server.xml 文件，会发现在<Engine>元素中已经有一个名为 localhost 的<Host>元素，可以在它的后面（即</Host>后面）加入如下<Host>元素：

```
<Host name="www.mycompany1.com" debug="0" appBase="C:\mycompany1">
  <unpackWARs="true" autoDeploy="true">
    <alias>mycompany1.com</alias>
    <alias>mycompany1</alias>
```

```
<Context path="/helloapp" docBase="helloapp" debug="0"
reloadable="true" />
```

</Host>

以上配置代码位于本书配套光盘的 sourcecode/chapter2/virtualhost-configure.xml 文件中。

<Host>元素的属性描述参见表 2-5。

表 2-5 <Host>元素的属性

属性	描述
name	指定虚拟主机的名字
debug	指定日志级别
appBase	指定虚拟主机的目录，可以指定绝对目录，也可以指定相对于<CATALINA_HOME>的相对目录。如果此项没有设定，默认值为<CATALINA_HOME>/webapps
unpackWARs	如果此项设为 true，表示将把 Web 应用的 WAR 文件先展开为开放目录结构后再运行。如果设为 false，则直接运行 WAR 文件
autoDeploy	如果此项设为 true，表示当 Tomcat 服务器处于运行状态时，能够监测 appBase 下的文件，如果有新的 Web 应用加入进来，则会自动发布这个 Web 应用
alias	指定虚拟主机的别名，可以指定多个别名
deployOnStartup	如果此项设为 true，则表示 Tomcat 服务器启动时会自动发布 appBase 目录下所有的 Web 应用。如果 Web 应用在 server.xml 中没有相应的<Context>元素，则将采用默认的 Context 配置。deployOnStartup 的默认值为 true

在<Host>的 deployOnStartup 属性为 true 的情况下，即使你没有在 server.xml 中为 helloapp 应用加入<Context>元素，Tomcat 服务器也可以自动发布和运行 helloapp 应用。在这种情况下，Tomcat 使用默认的 DefaultContext。关于 DefaultContext 的知识可以参考 Tomcat 文档：

[<CATALINA\\_HOME>/webapps/tomcat-docs/config/defaultcontext.html](http://tomcat.apache.org/tomcat-5.5-doc/config/defaultcontext.html)

(2) 把 helloapp 应用 (helloapp.war 文件或者是整个 helloapp 目录) 拷贝到 appBase 属性指定的目录 C:/mycompany1 下。

(3) 为了使以上配置的虚拟主机生效，必须在 DNS 服务器中注册以上的虚拟主机名和别名，使它们的 IP 地址都指向 Tomcat 服务器所在的机器。必须注册以下名字：

www.mycompany1.com

mycompany1.com

mycompany1

(4) 重启 Tomcat 服务器，然后通过浏览器访问：

<http://www.mycompany1.com/helloapp/index.htm>

如果返回正常的页面就说明配置成功。还可以通过虚拟机的别名来访问 helloapp 应用：

<http://mycompany1.com/helloapp/index.htm>

<http://mycompany1/helloapp/index.htm>

## 2.4 小结

本章通过 helloapp Web 应用例子，介绍了在 Tomcat 上创建和发布 Web 应用的步骤。通过本章内容，读者可以学会创建 Web 应用的目录结构，创建 web.xml 文件，并且能够把 HTML、Servlet、JSP 和 Tag Library 部署到 Web 应用中。此外，读者还可以掌握将整个 Web 应用打包并发布的方法。本章还介绍了配置虚拟主机的方法。

为了便于读者编译源程序，在本书配套光盘的 sourcecode/chapter2 目录下提供了编译本章 Java 程序的脚本 compile.bat，它的内容如下：

```
set catalina_home=C:\jakarta-tomcat
set path=%path%;C:\j2sdk1.4.2\bin

set currpath=\
if "%OS%" == "Windows_NT" set currpath=%~dp0%

set src=%currpath%helloapp\src
set dest=%currpath%helloapp\WEB-INF\classes
set classpath= %catalina_home%\common\lib\servlet-api.jar;
%catalina_home%\common\lib\jsp-api.jar

javac -sourcepath %src% -d %dest% %src%\mypack\DispatcherServlet.java
javac -sourcepath %src% -d %dest% %src%\mypack\HelloTag.java
```

运行这个脚本时，只要重新设置以上 Tomcat 目录和 JDK 的目录即可。

在 javac 命令中，-sourcepath 设定 Java 源文件的路径，-d 设定编译生成的类的存放路径。javac 命令的-classpath 参数可以设定 classpath 路径，如果此项没有设定，将参照环境变量 classpath 的设置。

# 第 3 章 Servlet 技术

本章介绍最主要的 Java Web 应用组件之一：Servlet。首先介绍 Java Servlet 的概念、生命周期和创建过程，然后讲述 Web 应用和 ServletContext 的关系。

## 3.1 Servlet 简介

Java Servlet 是与平台无关的服务器端组件，它可以运行在 Servlet 容器中。Servlet 容器负责 Servlet 和客户的通信以及调用 Servlet 的方法，Servlet 和客户的通信采用“请求/响应”的模式。

Servlet 可完成如下功能：

- 创建并返回基于客户请求的动态 HTML 页面
- 创建可嵌入到现有 HTML 页面中的部分 HTML 页面（HTML 片段）
- 与其他服务器资源（如数据库或基于 Java 的应用程序）进行通信
- 接收多个客户机的输入，并将结果广播到多个客户机上。例如，Servlet 可以实现支持多个参与者的游戏服务器
- 根据客户请求采用特定的 MIME（Multipurpose Internet Mail Extensions）类型对数据过滤，例如进行图像格式转换

## 3.2 Servlet API

Servlet 的框架是由两个 Java 包组成的：javax.servlet 和 javax.servlet.http。在 javax.servlet 包中定义了所有的 Servlet 类都必须实现或扩展的通用接口和类。在 javax.servlet.http 包中定义了采用 HTTP 协议通信的 HttpServlet 类。

Servlet 的框架的核心是 javax.servlet.Servlet 接口，所有的 Servlet 都必须实现这一接口。在 Servlet 接口中定义了 5 个方法，其中有 3 个方法代表了 Servlet 的生命周期：

- init 方法，负责初始化 Servlet 对象
- service 方法，负责响应客户的请求
- destroy 方法，当 Servlet 对象退出生命周期时，负责释放占用的资源

Servlet 的框架结构如图 3-1 所示。

从图 3-1 可以看出，GenericServlet 实现了 Servlet 接口，而 HttpServlet 扩展了 GenericServlet。当用户开发自己的 Servlet 类时，Servlet 类必须扩展以下两个类中的一个。

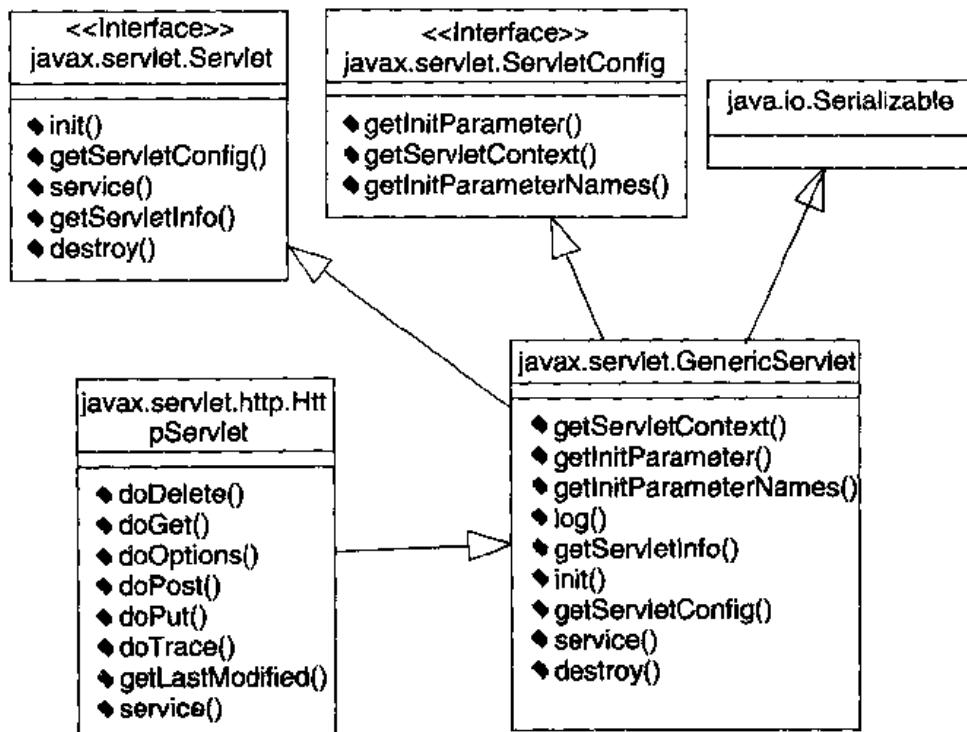


图 3-1 Servlet API 类框图

### ● 扩展 GenericServlet 类

如果 Servlet 类扩展了 GenericServlet 类，则必须实现 service 方法，因为 GenericServlet 类中的 service 方法被声明为抽象方法，该方法的声明形式如下：

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

service 方法有两个参数：ServletRequest 和 ServletResponse。Servlet 容器将客户的请求信息封装在 ServletRequest 对象中，传给 service 方法；service 方法将响应客户的结果通过 ServletResponse 对象传给客户。

### ● 扩展 HttpServlet 类

如果 Servlet 类扩展了 HttpServlet 类，通常不必实现 service 方法，因为 HttpServlet 类已经实现了 service 方法，该方法的声明形式如下：

```
protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException;
```

在 HttpServlet 的 service 方法中，首先从 HttpServletRequest 对象中获取 HTTP 请求方式的信息，然后再根据请求方式调用相应的方法。例如，如果请求方式为 GET，那么调用 doGet 方法；如果请求方式为 POST，那么调用 doPost 方法。



HTTP 的请求方式包括 DELETE、GET、OPTIONS、POST、PUT 和 TRACE，在 HttpServlet 类中分别提供了相应的方法，它们是 doDelete()、doGet()、doOptions()、doPost()、doPut() 和 doTrace()。

在 HttpServlet 的 service 方法中，也有两个参数：HttpServletRequest 和 HttpServletResponse，这两个类分别扩展了 ServletRequest 和 ServletResponse 类。

下面，再介绍一下 ServletRequest 和 ServletResponse 接口的作用以及它们的方法。

### ● ServletRequest 接口

ServletRequest 接口中封装了客户请求信息，如客户请求方式、参数名和参数值、客户端正在使用的协议，以及发出客户请求的远程主机信息等。ServletRequest 接口还为 Servlet 提供了直接以二进制数方式读取客户请求数据流的 ServletInputStream。ServletRequest 的子类可以为 Servlet 提供更多的与特定协议相关的信息。例如，HttpServletRequest 提供了读取 HTTP Head 信息的方法。

ServletRequest 接口提供的获取客户请求信息的部分方法参见表 3-1，可以参考 Servlet API 文档，获取关于 ServletRequest 更详细的信息。Servlet API 文档的地址为：

<CATALINA\_HOME>/webapps/tomcat-docs/servletapi/index.html

表 3-1 ServletRequest 接口的方法

方法名	描述
getAttribute	根据参数给定的属性名返回属性值
getContentType	返回客户请求数据 MIME 类型
getInputStream	返回以二进制数方式直接读取客户请求数据的输入流
getParameter	根据给定的参数名返回参数值
getRemoteAddr	返回远程客户主机的 IP 地址
getRemoteHost	返回远程客户主机名
getRemotePort	返回远程客户主机的端口
setAttribute	在 ServletRequest 中设置属性（包括属性名和属性值）

### ● ServletResponse 接口

ServletResponse 接口为 Servlet 提供了返回响应结果的方法。它允许 Servlet 设置返回数据的长度和 MIME 类型，并且提供输出流 ServletOutputStream。ServletResponse 子类可以提供更多和特定协议相关的方法。例如，HttpServletResponse 提供设定 HTTP HEAD 信息的方法。

ServletResponse 接口提供的输出 Servlet 响应结果的部分方法参见表 3-2，可以参考 Servlet API 文档，获取关于 ServletResponse 更详细的信息。

表 3-2 ServletResponse 的方法

方法名	描述
getOutputStream	返回可以向客户端发送二进制数据的输出流对象 ServletOutputStream
getWriter	返回可以向客户端发送字符数据的 PrintWriter 对象
getCharacterEncoding	返回 Servlet 发送的响应数据的字符编码
getContentType	返回 Servlet 发送的响应数据的 MIME 类型
setCharacterEncoding	设置 Servlet 发送的响应数据的字符编码
setContentType	设置 Servlet 发送的响应数据的 MIME 类型

### 3.3 Servlet 的生命周期

Servlet 的生命周期开始于被装载到 Servlet 容器中，结束于被终止或重新装入时。Servlet 的生命周期可以分为 3 个阶段：初始化阶段、响应客户请求阶段和终止阶段。在 javax.servlet.Servlet 接口中定义了 3 个方法 init()、service() 和 destroy()，它们将分别在 Servlet 的不同阶段被调用。

#### 1. 初始化阶段

在下列情形下 Servlet 容器装载 Servlet：

- Servlet 容器启动时自动装载某些 Servlet
- 在 Servlet 容器启动后，客户首次向 Servlet 发出请求
- Servlet 的类文件被更新后，重新装载 Servlet



Servlet 容器是否在启动时自动装载 Servlet，这是由在 web.xml 中为 Servlet 设置的<load-on-startup>属性决定的，参见附录 B（web.xml 文件）。

Servlet 被装载后，Servlet 容器创建一个 Servlet 实例并且调用 Servlet 的 init() 方法进行初始化。在 Servlet 的整个生命周期中，init 方法只会被调用一次。init 方法有两种重载形式：

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

在 Servlet 的初始化阶段，Servlet 容器会为 Servlet 创建一个 ServletConfig 对象，用来存放 Servlet 的初始化配置信息，如 Servlet 的初始参数。如果 Servlet 类覆盖了第一种带参数的 init 方法，应该先调用 super.init(config) 方法确保参数 config 引用 ServletConfig 对象；如果覆盖的是第二种不带参数的 init 方法，可以不调用 super.init() 方法，如果要在 init 方法中访问 ServletConfig 对象，可以调用 Servlet 类的 getServletConfig() 方法。

#### 2. 响应客户请求阶段

对于到达 Servlet 容器的客户请求，Servlet 容器创建特定于这个请求的 HttpServletRequest 对象和 HttpServletResponse 对象，然后调用 Servlet 的 service 方法。service 方法从 HttpServletRequest 对象获得客户请求信息并处理该请求，通过 HttpServletResponse 对象向客户返回响应结果。

#### 3. 终止阶段

当 Web 应用被终止，或 Servlet 容器终止运行，或 Servlet 容器重新装载 Servlet 的新实例时，Servlet 容器会先调用 Servlet 的 destroy 方法。在 destroy 方法中，可以释放 Servlet 所占用的资源。

## 3.4 HTTP与HttpServlet

Web服务器和浏览器通过HTTP在Internet上发送和接收消息。HTTP是一种基于请求/响应模式的协议。客户端发送一个请求，服务器返回对该请求的响应。HTTP使用可靠的TCP连接，默认端口是80。HTTP的第一个版本是HTTP/0.9，后来发展到了HTTP/1.0，现在最新的版本是HTTP/1.1，它在RFC 2616中定义。Tomcat 5版本默认情况下使用HTTP/1.1协议。关于HTTP/1.1的详细资料可以参考<http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf>。

在HTTP中，客户端/服务器之间的会话总是由客户端通过建立连接和发送HTTP请求的方式初始化，服务器不会主动联系客户端或要求与客户端建立连接。在会话开始后，浏览器或服务器都可以随时中断连接，例如，在浏览网页时可以随时点击【停止】按钮中断当前的文件下载过程，关闭与Web服务器的HTTP连接。

### 3.4.1 HTTP请求

HTTP请求由3个部分构成，分别是：

- 请求方法 URI 协议/版本
- 请求头（Request Header）
- 请求正文

下面是一个HTTP请求的例子：

```
GET /sample.jsp HTTP/1.1
Accept: image/gif, image/jpeg, /*
Accept-Language: zh-cn
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Accept-Encoding: gzip, deflate

userName=weiqin&password=1234
```

#### 1. 请求方法 URI 协议/版本

请求的第一行是“方法 URI 协议/版本”：

```
GET /sample.jsp HTTP/1.1
```

以上代码中“GET”代表请求方法，“/sample.jsp”表示URI，“HTTP/1.1”代表协议和协议的版本。

根据HTTP标准，HTTP请求可以使用多种请求方法。例如，HTTP 1.1支持7种请求方法：GET、POST、HEAD、OPTIONS、PUT、DELETE和TRACE。在Internet应用中，最常用的请求方法是GET和POST。

URI完整地指定了要访问的网络资源，通常只要给出相对于服务器的根目录的相对目录即可，因此总是以“/”开头。最后，协议版本声明了通信过程中使用的HTTP的版本。

## 2. 请求头 (Request Header)

请求头包含许多有关客户端环境和请求正文的有用信息。例如，请求头可以声明浏览器所用的语言，请求正文的长度等。

```
Accept: image/gif, image/jpeg, */*
Accept-Language: zh-cn
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Accept-Encoding: gzip, deflate
```

## 3. 请求正文

请求头和请求正文之间是一个空行（只有 CRLF 符号的行），这个行非常重要，它表示请求头已经结束，接下来的是请求的正文。请求正文中可以包含客户提交的查询字符串信息：

```
userName=weiqin&password=1234
```

在以上例子的 HTTP 请求中，请求的正文只有一行内容。当然，在实际应用中，HTTP 请求正文可以包含更多的内容。

### 3.4.2 HTTP 响应

与 HTTP 请求相似，HTTP 响应也由 3 个部分构成，分别是：

- 协议 状态代码 描述
- 响应头 (Response Header)
- 响应正文

下面是一个 HTTP 响应的例子：

```
HTTP/1.1 200 OK
Server: ApacheTomcat/5.0.12
Date: Mon, 6 Oct 2003 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 6 Oct 2003 13:23:42 GMT
Content-Length: 112
```

```
<html>
<head>
<title>HTTP 响应示例</title>
</head>
<body>
Hello HTTP!
</body>
</html>
```

#### 1. 协议 状态代码 描述

HTTP 响应的第一行类似于 HTTP 请求的第一行，它表示通信所用的协议是 HTTP 1.1，服务器已经成功地处理了客户端发出的请求（200 表示成功）：

HTTP/1.1 200 OK

## 2. 响应头 (Response Header)

响应头也和请求头一样包含许多有用的信息，例如服务器类型、日期时间、内容类型和长度等：

```
Server: ApacheTomcat/5.0.12
Date: Mon, 6 Oct 2003 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 6 Oct 2003 13:23:42 GMT
Content-Length: 112
```

## 3. 响应正文

响应正文就是服务器返回的 HTML 页面：

```
<html>
<head>
<title>HTTP 响应示例</title>
</head>
<body>
Hello HTTP!
</body>
</html>
```

响应头和正文之间也必须用空行分隔。

### 3.4.3 HttpServlet 的功能

在了解了具体的 HTTP 协议规范后，可以更好地理解 HttpServlet 的作用。它能够根据客户发出的 HTTP 请求，生成相应的 HTTP 响应结果。HttpServlet 首先必须读取 HTTP 请求的内容。Servlet 容器负责创建 HttpServletRequest 对象，并把 HTTP 请求信息封装到 HttpServletRequest 对象中，这大大简化了 HttpServlet 解析请求数据的工作量。如果没有 HttpServletRequest，HttpServlet 只能直接处理 Web 客户发出的原始的字符串数据，有了 HttpServletRequest 后，只要调用 HttpServletRequest 的相关方法，就可以方便地读取 HTTP 请求中任何一部分信息。HttpServletRequest 中读取 HTTP 请求信息的常用方法参见表 3-3。

表 3-3 HttpServletRequest 的常用方法

方 法	描 述
getCookies()	返回 HTTP 请求的 Cookies
getHeader(String name)	返回参数指定的 HTTP 请求的 Header 数据
getRequestURI()	返回 HTTP 请求 URI
getQueryString()	返回 HTTP 请求数据中的查询字符串
getMethod()	返回 HTTP 请求方法

Servlet 容器还向 HttpServlet 提供了 HttpServletResponse 对象，HttpServlet 可以通过它来生成 HTTP 响应的每一部分内容。HttpServletResponse 提供的生成响应数据 Header 的方法参见表 3-4。

表 3-4 HttpServletResponse 的常用方法

方 法	描 述
addCookie(Cookie cookie)	向 HTTP 响应中加入 Cookie
setHeader(String name, String value)	设置 HTTP 响应的 Header, 如果参数 name 对应的 Header 已经存在, 则覆盖原来的 Header 数据
addHeader(String name, String value)	向 HTTP 响应中加入 Header

**提示**

除了表 3-3 和表 3-4 列出的方法外, 在 HttpServletRequest 的父类 ServletRequest 中提供了读取客户请求的通用方法, 在 HttpServletResponse 的父类 ServletResponse 中提供了生成服务器响应的通用方法。

Servlet 容器响应 Web 客户请求流程的 UML 时序图如图 3-2 所示。

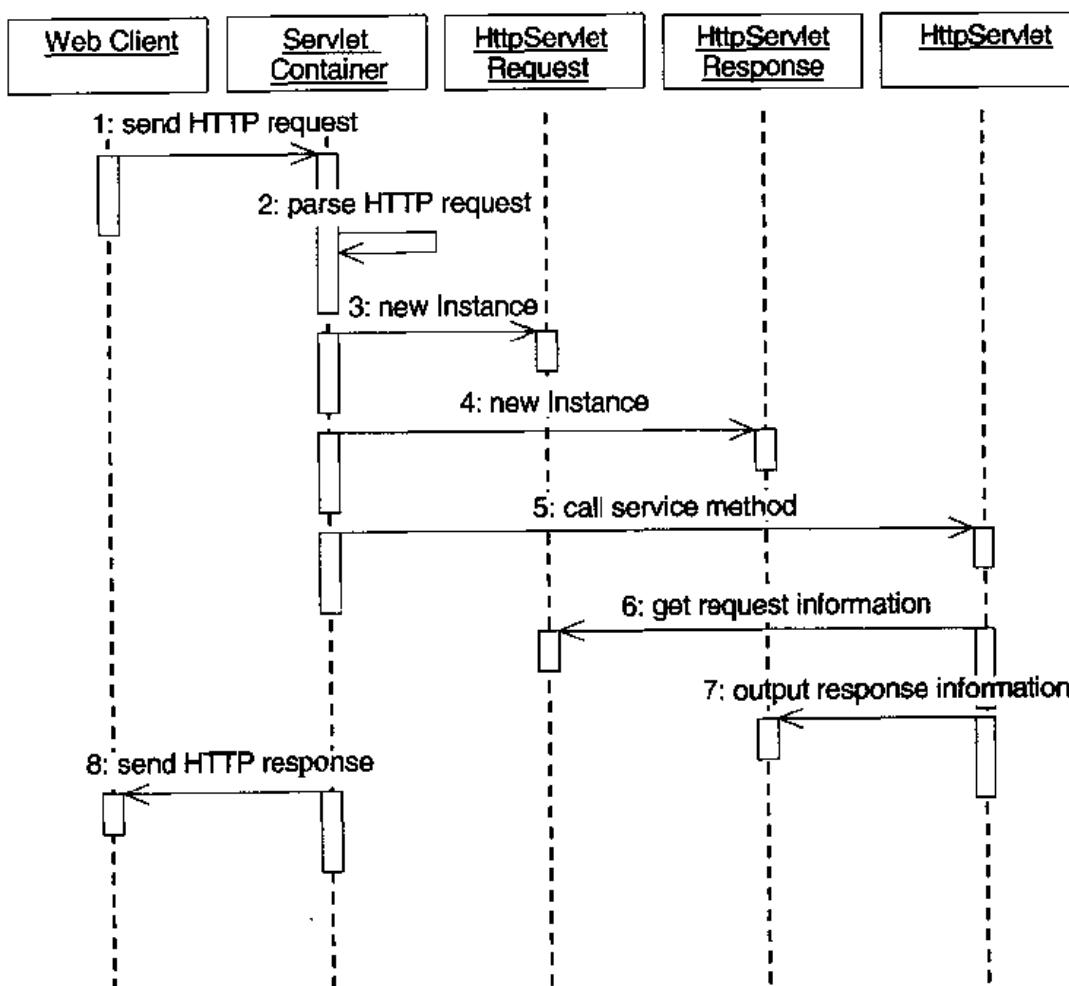


图 3-2 Servlet 容器响应 Web 客户请求的时序图

在本书中, 经常用 UML 时序图来直观地显示对象之间相互作用的流程。在时序图中, 从对象 A 到对象 B 的箭头, 表示 A 向 B 发送一条消息, B 接收到消息后, 将执行相关的操作, 因此也可以理解为 A 调用 B 的方法。例如图 3-2 中的步骤 5, 表示 Servlet 容器调用

HttpServlet 的 service 方法。对于步骤 2，箭头的起点和终点都指向 Servlet 容器，表示 Servlet 容器调用自身的方法来解析 HTTP 请求信息。

下面解释图 3-2 的时序图中的各个步骤的含义：

- 1: Web 客户向 Servlet 容器发出 HTTP 请求；
- 2: Servlet 容器解析 Web 客户的 HTTP 请求；
- 3: Servlet 容器创建一个 HttpServletRequest 对象，在这个对象中封装了 HTTP 请求信息；
- 4: Servlet 容器创建一个 HttpServletResponse 对象；
- 5: Servlet 容器调用 HttpServlet 的 service 方法，把 HttpServletRequest 和 HttpServletResponse 对象作为 service 方法的参数传给 HttpServlet 对象；
- 6: HttpServlet 调用 HttpServletRequest 的有关方法，获取 HTTP 请求信息；
- 7: HttpServlet 调用 HttpServletResponse 的有关方法，生成响应数据；
- 8: Servlet 容器把 HttpServlet 的响应结果传给 Web 客户。

## 3.5 创建 HttpServlet 的步骤

创建用户自己的 HttpServlet 类，通常涉及下列 4 个步骤。

### 步骤

(1) 扩展 HttpServlet 抽象类。

(2) 覆盖 HttpServlet 的部分方法，如覆盖 doGet() 或 doPost() 方法。

(3) 获取 HTTP 请求信息，例如通过 HttpServletRequest 对象来检索 HTML 表单所提交的数据或 URL 上的查询字符串。无论是 HTML 表单数据还是 URL 上的查询字符串，在 HttpServletRequest 对象中都以参数名/参数值的形式存放，可以通过以下方法检索参数信息：

- getParameterNames(): 返回一个 Enumeration 对象，它包含了所有的参数名信息
- getParameter(String name): 返回参数名 name 对应的参数值
- getParameterValues(): 返回一个 Enumeration 对象，它包含了所有的参数值信息

(4) 生成 HTTP 响应结果。通过 HttpServletResponse 对象可以生成响应结果。HttpServletResponse 对象有一个 getWriter() 方法，该方法返回一个 PrintWriter 对象。使用 PrintWriter 的 print() 或 println() 方法可以向客户端发送字符串数据流。

例程 3-1 提供了一个 Servlet 样例（HelloServlet.java）。

例程 3-1 HelloServlet.java

---

```

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

public class HelloServlet extends HttpServlet // 第一步：扩展 HttpServlet 抽象类.
{
    
```

```
//第二步：覆盖 doGet() 方法
public void doGet(HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException
{
    //第三步：获取 HTTP 请求中的参数信息
    String clientName=request.getParameter("clientName");
    if(clientName!=null)
        clientName=new String(clientName.getBytes("ISO-8859-1"),"GB2312");
    else
        clientName="我的朋友";

    // 第四步：生成 HTTP 响应结果
    PrintWriter out;
    String title="HelloServlet";
    String heading1="This is output from HelloServlet by doGet:";
    // set content type.
    response.setContentType("text/html;charset=GB2312");
    // write html page.
    out = response.getWriter();
    out.print("<HTML><HEAD><TITLE>" + title + "</TITLE>");
    out.print("</HEAD><BODY>");
    out.print(heading1);
    out.println("<h1><P> " + clientName + " : 您好</h1>");
    out.print("</BODY></HTML>");
    //close out.
    out.close();
}
}
```

上述 HelloServlet 类扩展 HttpServlet 抽象类，覆盖了 doGet 方法。在重写的 doGet 方法中，通过 getParameter 方法读取 HTTP 请求中的一个参数 clientName。在上述代码中，为了解决汉化问题，做了必要的转码工作。客户提交的查询数据采用默认的 ISO-8859-1 编码，应该把它转换为中文编码 GB2312：

```
//字符编码转换
clientName=new String(clientName.getBytes("ISO-8859-1"),"GB2312");
.....
//设置输出响应数据的字符编码
response.setContentType("text/html;charset=GB2312");
```

可以按 2.2.6 小节（部署 Servlet）的步骤编译并发布 HelloServlet，假定把 HelloServlet 发布到 helloapp 应用中，HelloServlet.class 的存放位置为：

```
<CATALINA_HOME>/webapps/helloapp/WEB-INF/classes/mypack/HelloServlet.class
在 web.xml 中为 HelloServlet 类加上如下<servlet>和<servlet-mapping>元素：
<servlet>
    <servlet-name>HelloServlet</servlet-name>
```

```
<servlet-class>mypack.HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

然后通过如下 URL 访问 HelloServlet：

<http://localhost:8080/helloapp/hello?clientName=小芳>  
HelloServlet 的输出结果如图 3-3 所示。



图 3-3 HelloServlet 的执行结果

## 3.6 ServletContext 和 Web 应用的关系

Servlet 容器在启动时会加载 Web 应用，并为每个 Web 应用创建唯一的 ServletContext 对象。可以把 ServletContext 看成是一个 Web 应用的服务器端组件的共享内存。在 ServletContext 中可以存放共享数据，它提供了 4 个读取或设置共享数据的方法，参见表 3-5。

表 3-5 ServletContext 的方法

方 法 名	描 述
setAttribute (String name, Object object)	把一个对象和一个属性名绑定，将这个对象存储在 ServletContext 中
getAttribute (String name)	根据给定的属性名返回所绑定的对象
removeAttribute (String name)	根据给定的属性名从 ServletContext 中删除相应的属性
getattributeNames ()	返回一个 Enumeration 对象，它包含了存储在 ServletContext 对象中的所有属性名

接下来将通过一个 CounterServlet 的例子（例程 3-2）来说明 ServletContext 和 Web 应用的关系。以下是源代码。

例程 3-2 CounterServlet.java

---

```
package mypack;
```

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

public class CounterServlet extends HttpServlet {

    private static final String CONTENT_TYPE = "text/html";

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

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

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

        //获得 ServletContext 的引用
        ServletContext context = getServletContext();
        // 从 ServletContext 读取 count 属性
        Integer count = (Integer)context.getAttribute("count");

        // 如果 count 属性还没有设置，那么创建 count 属性，初始值为 0
        // one and add it to the ServletContext
        if ( count == null ) {
            count = new Integer(0);
            context.setAttribute("count", new Integer(0));
        }

        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>WebCounter</title></head>");
        out.println("<body>");
        // 输出当前的 count 属性值
        out.println("<p>The current COUNT is : " + count + "</p>");
        out.println("</body></html>");

        // 创建新的 count 对象，其值增 1
        count = new Integer(count.intValue() + 1);
    }
}
```

```
// 将新的 count 属性存储到 ServletContext 中
context.setAttribute("count", count);
}

public void destroy() {
}
}
```

以上代码在 ServletContext 中设置了一个 Integer 类型的属性，属性名为 count。当该 Servlet 的 doGet 或 doPost 方法被调用时，它先从 ServletContext 中读取 count 属性，如果 count 属性不存在，那就在 ServletContext 中创建 count 属性。接下来向客户端输出当前 count 值，然后将 count 值加 1，再把新的 count 存储到 ServletContext 对象中。

可以按 2.2.6(部署 Servlet)小节的步骤编译并发布 CounterServlet，假定把 CounterServlet 发布到 helloapp 应用中，CounterServlet.class 的存放位置为：

```
<CATALINA_HOME>/webapps/helloapp/WEB-INF/classes/mypack/CounterServlet.class
```

在 web.xml 中为 CounterServlet 类加上如下<servlet>和<servlet-mapping>元素：

```
<servlet>
    <servlet-name>CounterServlet</servlet-name>
    <servlet-class>mypack.CounterServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>CounterServlet</servlet-name>
    <url-pattern>/counter</url-pattern>
</servlet-mapping>
```

我们按下面的步骤测试该 Servlet。

### 步骤

(1) 通过如下 URL 访问 CounterServlet：

```
http://localhost:8080/helloapp/counter
```

第一次访问该 Servlet 时，在浏览器上的 count 值为 0。

(2) 刷新该页面，会看到每刷新一次，count 值增加 1，假定最后一次刷新后 count 值为 5。

(3) 再打开一个新的浏览器，访问 CounterServlet，此时 count 值为 6。

(4) 重新启动 Tomcat 服务器，然后再访问 CounterServlet，count 值又被初始化为 0。

(5) 复制 helloapp 应用，改名为 helloapp1，再发布 helloapp1。通过不同的浏览器窗口分别访问 helloapp 以及 helloapp1 中的 CounterServlet，会发现这两个 Web 应用拥有各自独立的 count 属性。

通过上述实验，我们可以看到在 ServletContext 中设置的属性，在 Web 应用运行期间一直存在。当 Web 应用被关闭时，Servlet 容器会销毁 ServletContext 对象，存储在 ServletContext 对象中的属性自然也不复存在。不同 Web 应用的 ServletContext 各自独立。

## 3.7 小 结

本章介绍了 Java Servlet 的概念，代表 Servlet 生命周期的 3 个方法：init()、service() 和 destroy()。这里首先介绍了 Servlet API 中常用的一些接口和类，包括：Servlet 接口、GenericServlet 类、HttpServlet 类、ServletRequest 类和 ServletResponse 类；然后讲述了创建 Servlet 的步骤；最后通过 CounterServlet 例子说明了 Web 应用和 ServletContext 的关系。

# 第4章 JSP技术

JSP是Java Server Page的缩写，它是Servlet的扩展，其目的是简化建立和管理动态网站的工作。本章将介绍JSP的运行机制和语法，以及在JSP中使用Cookie的方法，本章还将讲述JSP的异常处理。

## 4.1 JSP简介

在传统的网页HTML文件(\*.htm, \*.html)中加入Java程序片段(Scriptlet)和JSP标签，就构成了JSP网页。Java程序片段可以操纵数据库、重新定向网页以及发送E-mail等，实现建立动态网站所需要的功能。所有程序操作都在服务器端执行，网络上传送给客户端的仅是得到的结果，这样大大降低了对客户浏览器的要求，即使客户浏览器端不支持Java，也可以访问JSP网页。

在JSP的众多优点之中，其中之一的是它能把HTML编码和业务逻辑有效地分离。通常，JSP负责生成动态HTML页面，业务逻辑由其他可重用的组件(如Servlet、Java Bean)和其他Java程序来实现，JSP可以通过Java程序片段访问这些业务组件。JSP访问服务器端可重用组件的模型如图4-1所示。

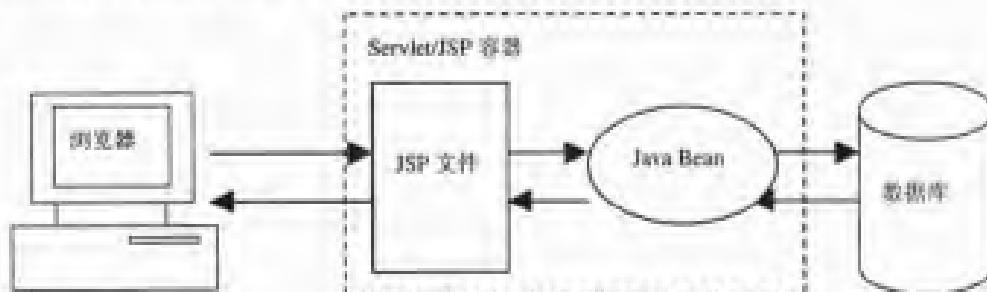


图4-1 JSP访问服务器端可重用组件

当Tomcat服务器接收到Web客户端的一个JSP文件请求时，它对JSP文件进行语法分析并生成Java Servlet源文件，然后对其进行编译。一般情况下，Servlet源文件的生成和编译仅在初次调用JSP时发生。如果原始的JSP文件被更新，Tomcat服务器将检测所做的更新，在执行它之前重新生成Servlet并进行编译。Tomcat服务器初次执行JSP的过程如图4-2所示。



Tomcat把由JSP生成的Servlet源文件和类文件放于<CATALINA\_HOME>/work目录下，通常情况下，如果修改了JSP文件，Tomcat会重新编译JSP，并把编译生成的新文件覆盖work目录下原来的旧文件。在少数情况下，如

如果更新 JSP 文件后，通过浏览器浏览看到的仍然是旧的网页，有可能 Tomcat 使用的还是 work 目录下旧的文件。因此可以删除 work 目录下的相关文件，这样能够确保 Tomcat 重新编译了修改后的 JSP 文件。

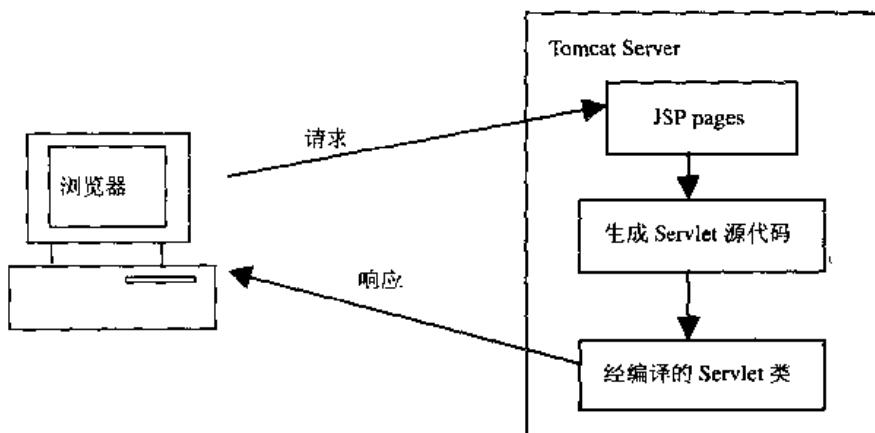


图 4-2 Tomcat 服务器初次执行 JSP 的过程

由 JSP 生成的 Servlet 类实现了 `javax.servlet.jsp.JspPage` 接口，该接口扩展了 `javax.servlet.Servlet` 接口。在 `javax.servlet.jsp.JspPage` 接口中定义了代表 JSP 生命周期的方法 `jspInit()` 和 `jspDestroy()`，类似于 `Servlet` 的 `init()` 和 `destroy()` 方法。

## 4.2 JSP 语法

JSP 文件（扩展名为 .jsp）可以包含如下内容：

- JSP 指令（或称为指示语句）
- JSP 声明
- Java 程序片段（Scriptlet）
- 变量数据的 Java 表达式
- 隐含对象

### 4.2.1 JSP 指令（Directives）

JSP 指令（在`<%@`和`%>`内的）用来设置和整个 JSP 网页相关的属性，如网页的编码方式和脚本语言等。JSP 指令的一般语法形式为：

`<%@ 指令名 属性="值" %>`

常用的 3 种指令为 `page`、`include` 和 `taglib`。下面分别讲述 `page` 和 `include` 指令，`taglib` 指令在本书第 14 章（自定义 JSP 标签）中讲解。

#### 1. page 指令

`page` 指令可以指定所使用的脚本语言、JSP 代表的 `Servlet` 实现的接口、`Servlet` 扩展的类以及导入的软件包。`page` 指令的语法形式为：

```
<%@ page 属性 1="值 1" 属性 2="值 2" %>
```



JSP 代表的 Servlet 指的是由 Servlet 容器编译 JSP 文件所生成的 Servlet 类。

page 指令的属性描述参见表 4-1。

表 4-1 page 指令的属性

page 指令的属性	描述	举例
language	指定文件中所使用的脚本语言。目前仅 java 为有效值和默认值。该指令作用于整个文件。当多次使用该指令时，只有第一次使用是有效的	<%@ page language="java" %>
method	指定 Java 程序片段 (Scriptlet) 所属的方法的名称。Java 程序片段会成为指定方法的主体。默认的方法是 service 方法。当多次使用该指令时，只有第一次使用是有效的。该属性的有效值包括 service、doGet 和 doPost 等	<%@ page method="doPost" %>
import	指定导入的 Java 软件包名或类名列表。该列表用逗号分隔。在 JSP 文件中，可以多次使用该指令来导入不同的软件包	<%@ page import="java.io.*;java.util.Hashable" %>
content_type	指定响应结果的 MIME 类型。默认 MIME 类型是 text/html。默认字符编码为 ISO-8859-1。当多次使用该指令时，只有第一次使用是有效的	<%@ page content_type="text/html; charset=GB2312" %>
session= "true false"	指定 JSP 页是否使用 Session，默认为 true	<%@ page session="true" %>
errorPage= "error_url"	指定当发生异常时，客户请求被重新定向到哪个网页	<%@ page errorPage="errorpage.jsp" %>
isErrorPage= "true false"	表示此 JSP 网页是否为处理异常的网页	<%@ page isErrorPage="true" %>

## 2. include 指令

JSP 可以通过 include 指令来包含其他文件。被包含的文件可以是 JSP 文件、HTML 文件或文本文件。如果被包含的是 JSP 文件，那么被包含的 JSP 文件中的 Java 程序片段也会被执行。

include 指令的语法为：

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

在开发网站时，如果多数 JSP 网页都包含相同的内容，可以把这部分相同的内容单独放到一个文件中，其他的 JSP 文件通过 include 指令将这个文件包含进来，这样做可以提高开发网站的效率，而且便于维护网页。

例如，开发网站时，通常希望所有的网页保持同样的风格，比如每个网页上都有相同的 logo。因此可以把生成 logo 的代码放在一个 JSP 文件中，其他的 JSP 文件通过 include 指令将它包含进来。

以下是一个网上书店 bookstore（参见第 5 章）的 logo 页，名为 banner.jsp：

```
<body>

<hr>
```

banner.jsp 负责显示网上书店的 logo，它不是一个完整的页面，在其他的网页中，都通过 include 指令将它包含进来。

网页 bookstore.jsp 用 include 指令将 banner.jsp 包含进来。以下是 bookstore.jsp 的代码：

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ include file="common.jsp" %>
<html>
<head><title>Bookstore</title></head>
<%@ include file="banner.jsp" %>

<center>
<p><b><a href="<%request.getContextPath()%>/catalog.jsp">查看所有书目</a></b></p>

<form action=bookdetails.jsp method="POST">
<h3>请输入查询信息</h3>
<b>书的编号:</b>
<input type="text" size="20" name="bookId" value="" ><br><br>
<center><input type=submit value="查询"></center>
</form>
</center>

</body>
</html>
```

在 bookstore.jsp 中包含了两个 JSP 文件，一个是 banner.jsp，还有一个是 common.jsp。common.jsp 将在第 5 章（bookstore 应用简介）中讲述。bookstore.jsp 生成的网页如图 4-3 所示。

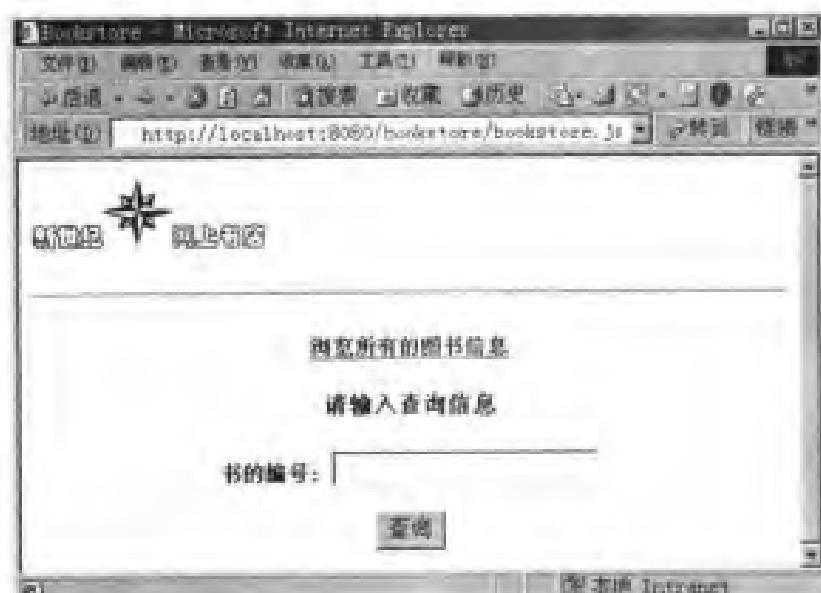


图 4-3 bookstore.jsp 网页

## 4.2.2 JSP 声明

JSP 声明（在`<%!`和`%>`内的）用于声明 JSP 代表的 Servlet 类的成员变量和方法。语法如下：

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

例如：

```
<%! int i=0; %>
<%! int a,b,c ;%>
<%! String h=new String("hello"); %>
<%!
public String f(int i){
    if(i<3) return "i<3";
    else return "i>=3";
}
%>
```

每个 JSP 声明只在当前 JSP 页面中有效，如果希望每个 JSP 页面中都包含这些声明，可以把它们写成单独的页面，然后用`<%@ include %>`指令把这个页面加到其他页面中。

## 4.2.3 Java 程序片段 (Scriptlet)

在第 3 章(Servlet 技术)中曾讲过当客户请求访问 Servlet 时，Servlet 容器会调用 Servlet 的 service 方法。在 JSP 文件中，可以在`<%` 和`%>`标记间直接嵌入任何有效的 Java 语言代码。这样嵌入的程序片段称为 Scriptlet。如果在 page 指令中没有指定 method 属性（参见 4.2.1 节），则生成的代码默认为 service 方法的主体。

例如以下 JSP 中定义了 3 个程序片段：

```
<% String gender="female"; if(gender.equals("female")){ %>
    She is a girl.
<% }else{ %>
    He is a boy.
<% }%>
```

以上代码等价于以下 Servlet 的 service 方法：

```
public void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    PrintWriter out = response.getWriter();
    String gender="female"; //局部变量
    if(gender.equals("female"))
        out.println("She is a girl.");
    else
        out.println("He is a boy.");
}
```

以上 JSP 的输出结果为“She is a girl.”。在这里，if 语句由 3 段`<%` 和`%>`代码构成，分段的 if 语句可以控制网页的输出结果。如果将以上代码改为：

```
<% String gender="male"; if(gender.equals("female")){ %>
    She is a girl.
<% }else{ %>
    He is a boy.
<% }%>
```

那么输出结果为：He is a boy.

再看下面这个例子：

```
<% int a=0;
while(a<3){
%>

a=<%=a%>

<%
a++;
} //end of while
%>
```

以上代码中 while 语句由两段<% 和 %>代码构成。分段的 while 语句可以控制循环输出。该 JSP 的输出内容为：a=0 a=1 a=2。

#### 4.2.4 变量数据的 Java 表达式

JSP 表达式标记为<%= 和 %>。该表达式的值会显示在网页上。int 或 float 类型的值都自动转换成字符串加以显示。以下是一个包含“<%! %>”、“<% %>”和“<%= %>”标记的 JSP 例子。

hitCounter.jsp

```
<html>
<head><title>Welcome Page</title></head>
<body>
<H1>You hit the page:
<%! int hitcount=1;%>
<% int count=0;
hitcount=count++;%>
<%= hitcount++ %>
times
</H1>
</body></html>
```

在 hitCounter.jsp 中，定义了两个变量：hitcount 和 count。hitcount 变量在 JSP 声明标记中定义，因此为类成员变量；count 变量在 Scriptlet 标记中定义，因此为局部变量。如果多次从客户端访问该 JSP，会发现每次的输出结果都如下所示：

You hit the page: 0 times

以上代码相当于以下的 Servlet 类：

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

```

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

public class hitCounterServlet extends HttpServlet {
    private int hitcount=1; //成员变量
    public void init() throws ServletException {
    }
    public void service(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
        int count=0; //局部变量

        hitcount=count++;
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Welcome Page</title></head>");
        out.println("<body>");
        out.println("<H1>You hit the page:"+(hitcount++)+" times<H1>"); //输出 hitcount 变量
        out.println("</body></html>");
    }
    /**Clean up resources*/
    public void destroy() {
    }
}

```

如果将 hitCounter.jsp 中<% 和 %>内容注释掉，则结果如下所示：

```

<html>
    <head><title>Welcome Page</title></head>
    <body>
        <H1>You hit the page:
        <%! int hitcount=1;%>
        <%-- 
        <% int count=0;
            hitcount=count++;%>
        --%>
        <%= hitcount++ %>
        times
        </H1>
    </body>
</html>

```

此时，hitcount 变量的输出值将会随着用户的访问次数从 1 开始逐渐递增。如果从多个浏览器窗口同时访问 hitCounter.jsp，则访问的都是同一个 hitcount 变量。

通过这个例子，我们可以理解 JSP 中类成员变量和局部变量的区别。对于非静态的类成员变量（也称为实例变量，如 hitcount 变量），每个 JSP 实例拥有一个实例变量，由于在 Servlet 容器内，一个 JSP 文件只对应一个 JSP 实例，因此也只有一个 hitcount 实例变量。

局部变量和实例变量有不同的生命周期，局部变量在一个方法中定义，当 Servlet 容器每次调用这个方法时，JVM（Java Virtual Machine，Java 虚拟机）将为局部变量分配内存，

从而创建一个新的局部变量。这个方法执行完毕后，JVM 就会销毁这个局部变量。

#### 4.2.5 隐含对象

在编写 JSP 程序时，可以直接使用 Servlet/JSP 容器提供的隐含对象。使用这些对象的引用变量时不需要做任何变量声明。所有的 JSP 隐含对象以及它们的类型参见表 4-2。

表 4-2 JSP 中的隐含对象

隐含对象	类 型
request	javax.servlet.HttpServletRequest
response	javax.servlet.HttpServletResponse
pageContext	javax.servlet.jsp.PageContext
application	javax.servlet.ServletContext
out	javax.servlet.jsp.JspWriter
config	javax.servlet.ServletConfig
page	java.lang.Object (相当于 Java 中的 this 关键字)
session	javax.servlet.http.HttpSession
exception	java.lang.Exception

例如，在 JSP 中可以直接通过 request 变量获取 HTTP 请求信息中的参数。

```
<%
    String username = request.getParameter("username");
    out.println(name);
%>
```

PageContext 和 JspWriter 类位于 javax.servlet.jsp 包中，它们的用法可以参考 JSP API：  
[<CATALINA\\_HOME>/webapps/tomcat-docs/jspapi/index.html](<CATALINA_HOME>/webapps/tomcat-docs/jspapi/index.html)

### 4.3 JSP 与 Cookie

Cookie 的英文原意是“点心”，它是用户访问 Web 服务器时，服务器在用户硬盘上存放的信息，好像是服务器送给客户的“点心”。

服务器可以根据 Cookie 来跟踪用户，这对于需要区别用户的场合（如电子商务）特别有用。不过，点心不是白吃的，服务器也可以据此收集用户的信息，因此为了保证安全，多数浏览器可以设置是否使用 Cookie。

为了测试本节程序，首先要确保浏览器启用了 Cookie。选择 IE 浏览器的【工具】→【Internet 选项】→【安全】选项卡，如图 4-4 所示。



图 4-4 IE 浏览器的 Internet 选项窗口

假定 Tomcat 服务器安装在本机，应该在图 4-4 中选择“本地 Intranet”图标，然后单击【自定义级别】按钮。在 Internet Explorer 中启用 Cookie 的界面如图 4-5 所示。



如果通过 IE 浏览器访问 Internet 上的 Web 站点，应该在图 4-4 中选择“Internet”图标，然后单击【自定义级别】按钮来设置 Cookie。



图 4-5 在 Internet Explorer 中设置 Cookie

一个 Cookie 包含一对 Key/Value。下面的代码生成一个 Cookie 并将它写到用户的硬盘上：

```
Cookie theCookie=new Cookie("cookieName","cookieValue");
```

```
response.addCookie(theCookie);
```

在 Cookie 的构造方法中，第一个参数是 Key，第二个参数是 Value。

如果服务器想从用户硬盘上获取 Cookie，可以使用下面方法从客户请求中取得所有的 Cookie：

```
Cookie cookies[] = request.getCookies();
```

然后调用 Cookie 的 getName 方法获得 Cookie 的 Key，调用 Cookie 的 getValue 方法获得 Cookie 的 Value。

此外，还可以通过 Cookie 的 setMaxAge(int expiry)方法来设置 Cookie 的有效期。超过参数 expiry 指定的时间（以秒为单位），Cookie 就会失效。

例程 4-1 是一个使用 Cookie 的例子。

例程 4-1 JspCookie.jsp

```
<html>
<head><title>jspCookie.jsp</title></head>
<body>

<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++)
{
%
>
<p>
<b>Cookie name:</b>
<%= cookies[i].getName() %>

<b>Cookie value:</b>
<%= cookies[i].getValue() %>
</p>
<p>
<b>Old max age in seconds:</b>
<%= cookies[i].getMaxAge() %>
<%
cookies[i].setMaxAge(60);
%
>
<b>New max age in seconds:</b>
<%= cookies[i].getMaxAge() %>
</p>
<%
}
%
>
<%
int count1 = 0;
int count2=0;
%
>
<%
```

```

response.addCookie(new Cookie(
    "cookieName" + count1++, "cookieValue" + count2++));

%>

</body>
</html>

```

JspCookie.jsp 首先读取客户端所有 Cookie 并把它们输出到网页上，然后修改 Cookie 的生命期，将修改前后 Cookie 的生命周期输出到网页上，最后创建新的 Cookie 并把它写到客户端。JspCookie.jsp 生成的网页如图 4-6 所示。

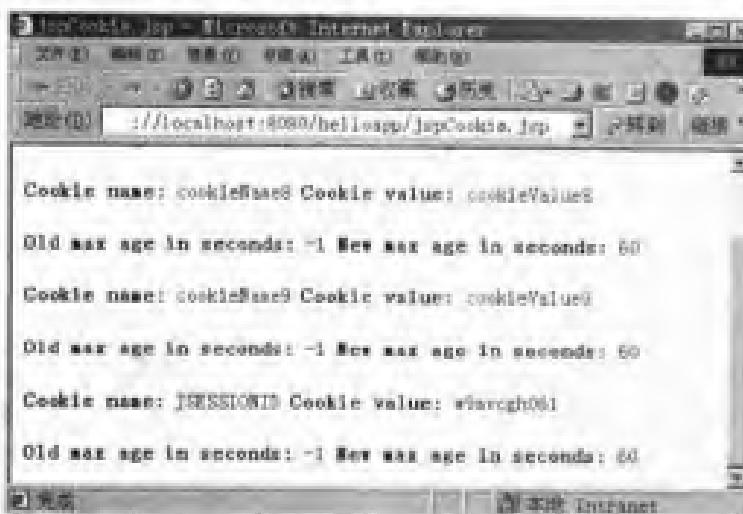


图 4-6 jspCookie.jsp 网页

在浏览 jspCookie.jsp 时，每次刷新页面，都会看到一个特殊的 Cookie: JSESSIONID。它代表了当前 Session 的 Session ID。第 7 章（Session 的使用与管理）将讲述 Session 的概念。

如果把 IE 浏览器的本地 Intranet 的安全级别设为禁用 Cookie，再运行 jspCookie.jsp 时将会出现异常。

## 4.4 转发 JSP 请求

<jsp:forward>标签用于将客户请求重定向到其他的 HTML 文件、JSP 文件或者 Servlet 文件。<jsp:forward>的语法为：

```
<jsp:forward page="重新定向的文件" />
```

<jsp:forward>标签从一个 JSP 文件向另一个文件传递包含用户请求的 request 对象。如果 JSP 文件中包含<jsp:forward>标签，那么这个 JSP 文件中的所有输出数据都不会被发送到客户端，并且<jsp:forward>标签以下的代码不会被执行。

例如下面的 JSP 程序 jspForward1.jsp 把客户请求转发给 jspForward2.jsp：

jspForward1.jsp

```
<html><head><title>Forward1 Page</title></head>
<body>
<p>
    This is output of jspForward1 before forward
</p>
<jsp:forward page="jspForward2.jsp" />
<p>
    This is output of jspForward1 after forward
</p>
</body></html>
```

### jspForward2.jsp

```
<html><head><title>Forward2 Page</title></head>
<body>
<p>
    hello, <%= request.getParameter("name")%>
</p>
</body></html>
```

在 jspForward1.jsp 中，<jsp:forward> 标签前后都试图输出一些字符串。假定把 jspForward1.jsp 和 JspForward2.jsp 都发布到 helloapp 应用中，访问 <http://localhost:8080/helloapp/jspForward1.jsp?name=weiqin>，生成的网页如图 4-7 所示。

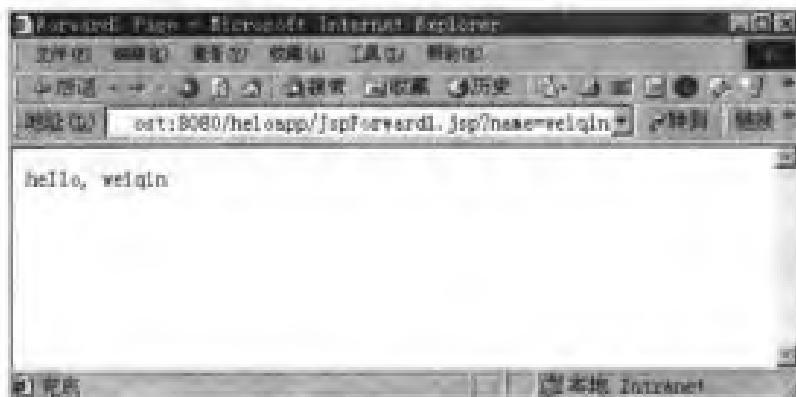


图 4-7 访问 jspForward1.jsp 生成的动态网页

由于 jspForward1.jsp 将请求转发给了 jspForward2.jsp，jspForward1.jsp 所有的数据输出都无效。此外，jspForward2.jsp 和 jspForward1.jsp 共享同一个 HttpServletRequest 对象，因此 jspForward2.jsp 可以通过 request.getParameter("name") 方法读取请求参数。

## 4.5 JSP 异常处理

像普通的 Java 程序一样，可以把异常引入到 JSP 中。如果在执行 JSP 的 Java 代码时发生异常，可以通过下面的指令将 HTTP 请求转发给另一个专门处理异常的网页：

```
<%@ page errorPage="errorpage.jsp" %>
```

在处理异常的网页中，应该通过如下语句将该网页声明为异常处理网页：

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

在处理异常的网页中可以直接访问 exception 隐含对象，获取详细的异常信息，例如：

```
<p>
```

```
    错误原因为：<% exception.printStackTrace(new PrintWriter(out));%>
```

```
</p>
```

下面创建一个可能会抛出异常的 JSP 网页 `jspSum.jsp`，在这个网页中读取客户请求中的两个参数 `num1` 和 `num2`，把它们转化为整数类型，再对其求和，最后把结果输出到网页上。将字符串转化为整数的代码如下：

```
private intToInt(String num){  
    return Integer.valueOf(num).intValue();  
}
```

如果客户输入的参数不能转化为整数，就会抛出 `NumberFormatException`，这时客户请求会转到 `errorpage.jsp`。

例程 4-2 和 4-3 分别是 `jspSum.jsp` 和 `errorpage.jsp` 的源代码。

例程 4-2 jspSum.jsp

---

```
<%@ page contentType="text/html; charset=GB2312" %>  
<%@ page errorPage="errorpage.jsp" %>  
  
<html><head><title>jspSum.jsp</title></head>  
<body>  
    <%!  
        private intToInt(String num){  
            return Integer.valueOf(num).intValue();  
        }  
    %>  
    <%  
        int num1=toInt(request.getParameter("num1"));  
        int num2=toInt(request.getParameter("num2"));  
    %>  
  
    <p>        运算结果为:<%=num1%>+<%=num2%>=<%=(num1+num2)%>  
    </p>  
</body></html>
```

---

例程 4-3 errorpage.jsp

---

```
<%@ page contentType="text/html; charset=GB2312" %>  
<%@ page isErrorPage="true" %>  
<%@ page import="java.io.PrintWriter" %>  
  
<html><head><title>Error Page</title></head>
```

---

```

<body>

    <p>
        你输入的参数（num1=<%=request.getParameter("num1")%>,
        num2=<%=request.getParameter("num2")%>）有错误
    </p>
    <p>
        错误原因为：<% exception.printStackTrace(new PrintWriter(out));%>
    </p>
</body></html>

```

假定把这两个 JSP 文件发布到 helloapp 应用，通过如下 URL 访问 jspSum.jsp：

<http://localhost:8080/helloapp/JspSum.jsp?num1=100&num2=200>

这时 jspSum.jsp 正常执行，生成的网页如图 4-8 所示。

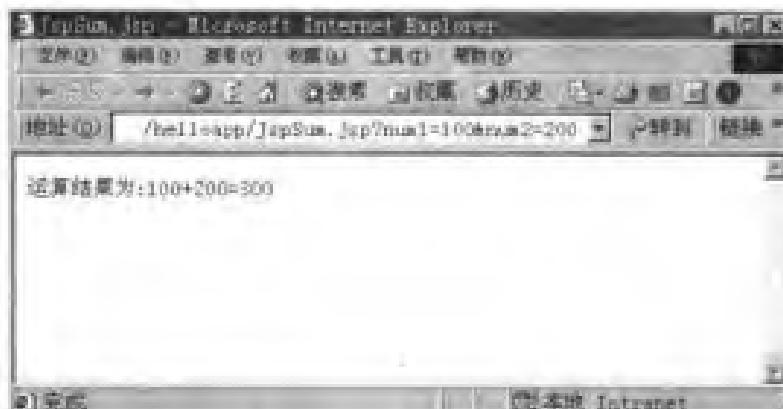


图 4-8 jspSum.jsp 正常执行时生成的网页

如果将 URL 中的参数 num2 的值改为字符串“two”，再访问 jspSum.jsp：

<http://localhost:8080/helloapp/JspSum.jsp?num1=100&num2=two>

由于 num2 参数不能转化为整数，就会抛出 NumberFormatException，这时客户请求会转到 errorpage.jsp，生成的网页如图 4-9 所示。



图 4-9 jspSum.jsp 出现异常时生成的网页

## 4.6 再谈部署 JSP

在第2章介绍了部署Java Web应用各个组件的方法。部署JSP很简单，例如，向helloapp应用中部署本章4.3节的jspCookie.jsp，只要把jspCookie.jsp文件拷贝到helloapp应用的根目录下即可。Web客户可以通过如下URL访问jspCookie.jsp：

<http://localhost:8080/helloapp/jspCookie.jsp>

部署Servlet时，必须在web.xml中加入<servlet>和<servlet-mapping>元素，其中<servlet-mapping>元素可以用来设定访问Servlet的<url-pattern>。事实上，也可以为JSP配置<servlet>和<servlet-mapping>元素，从而设定访问JSP的<url-pattern>。以下是在web.xml中配置jspCookie.jsp的代码：

```
<servlet>
    <servlet-name>jspCookie</servlet-name>
    <jsp-file>/jspCookie.jsp</jsp-file>
</servlet>

<servlet-mapping>
    <servlet-name>jspCookie</servlet-name>
    <url-pattern>/cookie</url-pattern>
</servlet-mapping>
```

在web.xml中加入了以上代码后，就可以通过<url-pattern>指定的URL来访问jspCookie.jsp：

<http://localhost:8080/helloapp/cookie>

## 4.7 小结

JSP是一种基于Java的脚本技术，它能将HTML编码从Web页面的业务逻辑中有效地分离出来。本章介绍了JSP的语法。JSP文件可以包含：JSP指令（或称为指示语句）、JSP声明、直接插入的Java代码（Scriptlet）、变量数据的Java表达式和JSP隐含对象。本章还讲解了Cookie的使用方法，以及JSP请求的转发和异常处理。本章没有覆盖JSP的所有技术，在本书的以下章中将详细介绍Session、访问JavaBean和自定义JSP标签的技术：

- 第7章：Session的使用与管理
- 第8章：访问JavaBean
- 第14章：自定义JSP标签

# 第 5 章 bookstore 应用简介

bookstore 应用是一个充分运用了本书所有 Web 技术的综合例子，它实现了一个网上书店，更加贴近于实际应用。本章将介绍 bookstore 应用的软件结构，各个 JSP 网页的功能，以及部分 Web 组件的实现。在本书的其他章节，还将陆续介绍 bookstore 应用涉及的 Web 技术。

本书一共提供了 4 个版本的 bookstore 应用，它们完成的功能和提供的客户界面都相同，但在实现方式上有差别。本章讲解的 bookstore 应用基于 version0，它的源文件在本书配套光盘上的位置为：sourcecode/bookstores/version0/bookstore。

## 5.1 bookstore 应用的软件结构

bookstore 应用是一个 Java Web 应用，采用典型的三层软件结构：

- 客户层：提供基于浏览器的客户界面，客户可以浏览 Web 服务器传过来的静态或动态 HTML 页面，客户可以通过动态 HTML 页面和 Web 服务器交互
- Web 服务器层：Servlet、JSP 和 JavaBean 组件运行在 Web 服务器上，JSP 负责生成动态 HTML 页面，JavaBean 负责访问数据库和事务处理。在 Web 服务器层还包括一些供 JSP 和 JavaBean 组件访问的实用类
- 数据库层：存放和维护 Web 应用的数据信息

bookstore 应用的软件结构如图 5-1 所示。

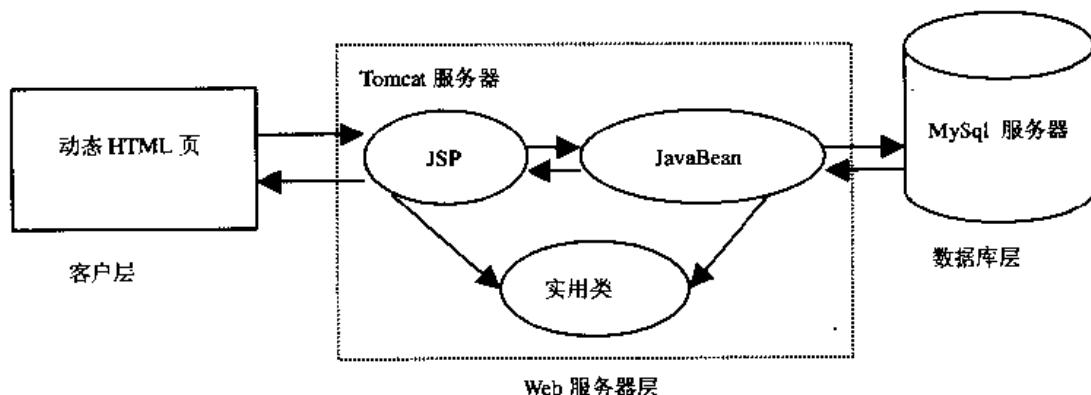


图 5-1 bookstore 应用的软件结构

### 5.1.1 Web 服务器层

开发 bookstore 应用最主要的工作就是开发 Web 服务器层组件。构成 Web 服务器层组

件的具体文件如表 5-1 所示。

表 5-1 bookstore 应用的文件清单

文件类别	文件名	描述
JSP	banner.jsp	网站的 logo
	common.jsp	包含了各个 JSP 网页的公共代码
	bookstore.jsp	网站的主页
	bookdetails.jsp	显示某本书的详细信息
	catalog.jsp	显示书店所有书目。客户可以将选购的书加入购物车
	showcart.jsp	显示客户购物车中的书，客户可以修改购物车的内容
	cashier.jsp	客户付款页面
	receipt.jsp	完成结账业务，结束当前购物交易，提供客户重新购物的链接
JavaBean	BookDB.java	访问数据库，查询书的信息，处理购书事务
	ShoppingCart.java	代表虚拟的购物车
实用类	BookDetails.java	代表具体的一本书，包含书的详细信息
	ShoppingCartItem.java	代表购物车中的一项购物条目

### 5.1.2 数据库层

bookstore 应用采用 MySQL 作为数据库服务器。网上书店中所有书的信息存放在数据库的 books 表中。books 表的字段说明参见表 5-2，在本书 6.1 节中，介绍了创建数据库 BookDB 和表 books 的步骤。

表 5-2 books 表的结构

字段	描述
id	书的 ID 号，它是 books 表的 primary key
name	作者姓名
title	书的名字
price	价格
yr	出版时间
description	书的描述信息
saleAmount	销售数量

## 5.2 浏览 bookstore 应用的 JSP 网页

下面从用户角度浏览 bookstore 应用的功能。bookstore 应用的站点导航图如图 5-2 所示，在这幅图上，显示了各个网页之间的链接关系。



图 5-2 bookstore 应用站点导航图

### 1. banner.jsp

banner.jsp 用于显示书店的 logo，它是所有网页的公共部分。在其他的 JSP 网页中，可以通过<%@ include>标记将 banner.jsp 包含进去。本书 4.2.1 节中介绍了 banner.jsp。

### 2. common.jsp

common.jsp 也是其他网页包含的公共部分，它通过<%@ page import>标记引入了 JSP 网页可能访问的 Java 类，通过<%@ page errorPage>标记指定异常处理页面，并且定义了一个 application 范围内的 Java Bean：

```
<jsp:useBean id="bookDB" scope="application" class="mypack.BookDB"/>
```

所有包含 common.jsp 的 JSP 网页都共享这个 bookDB 对象，它负责完成实际的数据库操作。

common.jsp 还提供了一个 convert(String s)方法来处理中文字符编码转换，它能够把采用 ISO-8859-1 编码的字符串转换为采用 GB2312 编码的字符串。关于编码转换的更多知识可以参考本书 6.6 节（处理中文编码）中的相关内容。

当 JSP 网页从数据库中取出数据，它使用的是 MySQL 数据库驱动程序默认的字符编码 ISO-8859-1，如果要正确地把它显示在网页上，则需要把字符编码转换为 GB2312。例如在 bookdetails.jsp 中为了显示书名，就调用了 convert 方法：

```
<p>书名: <%=convert(book.getTitle())%></p>
```

以下是 common.jsp 的源代码：

```
<%@ page import="mypack.*" %>
<%@ page import="java.util.Properties" %>
<%@ page errorPage="errorpage.jsp" %>

<jsp:useBean id="bookDB" scope="application" class="mypack.BookDB"/>

<%!
public String convert(String s){
try{
```

```
    return new String(s.getBytes("ISO-8859-1"),"GB2312");  
}catch(Exception e){return null;}  
}  
%>
```

### 3. bookstore.jsp

bookstore.jsp 是书店的首页，它提供了两个链接：可以通过“查看所有书目”链接进入 catalog.jsp，也可以输入书的编号，再单击【查询】按钮，转到 bookdetails.jsp 网页，查看某本书的详细信息。

通过 <http://localhost:8080/bookstore/bookstore.jsp> 进入网上书店的主页，如图 5-3 所示。

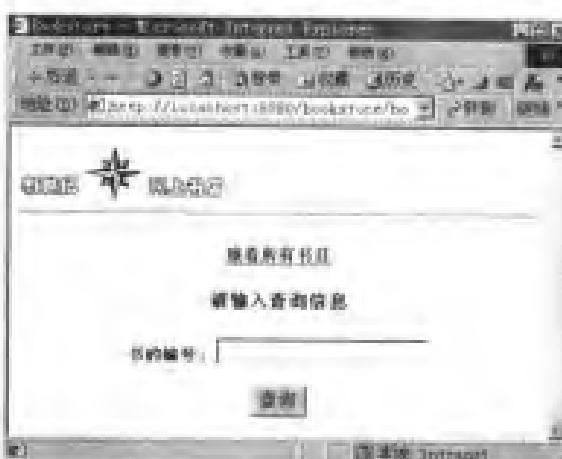


图 5-3 bookstore.jsp 网页

### 4. bookdetails.jsp

bookdetails.jsp 用于显示某一本书的详细信息，如图 5-4 所示。如果选择“加入购物车”链接，则会把这本书放到虚拟的购物车中，然后转到 catalog.jsp 网页；如果选择“继续购物”，则直接转到 catalog.jsp 网页。

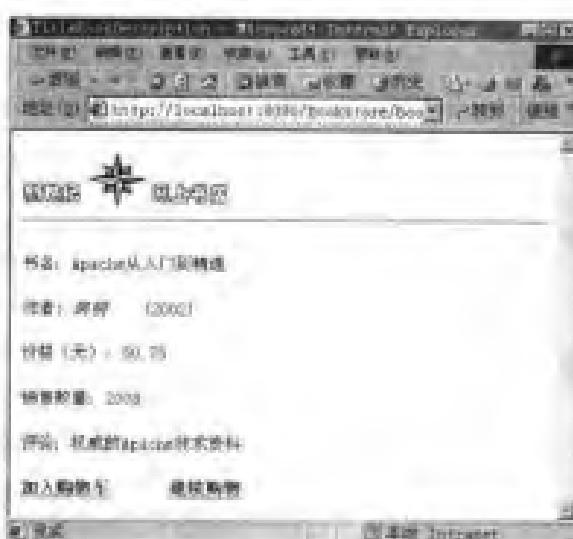


图 5-4 bookdetails.jsp 网页

如果客户查询的书的编号在数据库中不存在, bookdetails.jsp 将显示提示信息, 如图 5-5 所示。



图 5-5 书号不存在时的 bookdetails.jsp 网页

### 5. catalog.jsp

catalog.jsp 用于显示书店中所有的书目, 如图 5-6 所示, 客户可以在该网页上选购书。对于同一本书, 客户每选择一次“加入购物车”链接, 这本书的购买数量就会增加 1。每当客户将一本书加入购物车, 就会在网页上显示相应的提示信息。客户也可以选择“查看购物车”链接转到 showcart.jsp 网页, 或者选择“付账”链接转到 cashier.jsp 网页。

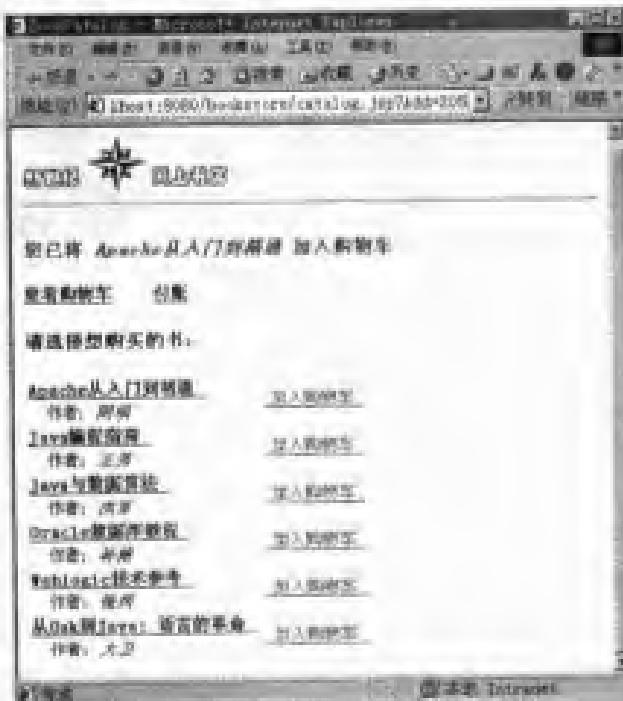


图 5-6 catalog.jsp 网页

### 6. showcart.jsp

showcart.jsp 用于显示客户的购物车中的信息, 如图 5-7 所示。客户可以通过选择“删除”链接从购物车中删除一本书, 也可以通过选择“清空购物车”链接删除所有购买的书。

目。showcart.jsp 中的“继续购物”链接转到 catalog.jsp，“付款”链接转到 cashier.jsp。



图 5-7 showcart.jsp 网页

### 7. cashier.jsp

cashier.jsp 提供了客户输入信用卡信息的表单，如图 5-8 所示。客户单击【提交】按钮，就会转到 receipt.jsp 网页。

您一共购买了5本书

您应支付的金额为223.75元

信用卡用户名

信用卡账号

提交

图 5-8 cashier.jsp 网页

### 8. receipt.jsp

receipt.jsp 结束当前的 Session，礼貌地和客户告别，并且提供了继续购物的链接，如图 5-9 所示。客户选择“继续购物”，又会转到 bookstore 的首页 bookstore.jsp。



图 5-9 receipt.jsp 网页

### 9. errorpage.jsp

errorpage.jsp 是异常处理页面，它将异常信息输出到网页上，以下是 errorpage.jsp 的源代码：

```
<%@ page import="java.io.*" %>
<!--设置中文输出-->
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page isErrorPage="true" %>

<html><head><title>Error Page</title></head>
<body>
<p>
    服务器端发生错误:<%= exception.getMessage() %>
</p>
<p>
    错误原因为：<% exception.printStackTrace(new PrintWriter(out));%>
</p>
</body></html>
```

当浏览器正在连接 bookdetails.jsp 网页时，如果 MySQL 服务器关闭，将会导致数据库连接异常，异常发生后，客户请求由 errorpage.jsp 处理，如图 5-10 所示。

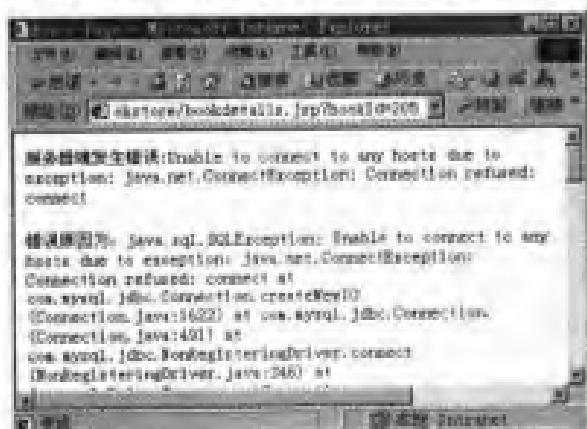


图 5-10 errorpage.jsp 网页

**提示**

在 Web 开发阶段，把程序中发生异常的堆栈信息输出到异常处理网页上，有助于开发人员跟踪和调试程序。当 Web 应用面向真正的 Web 客户时，应该提供更友好的异常处理网页，确保客户能够理解信息。

## 5.3 JavaBean 和实用类

在 bookstore 应用中，创建了以下类：

- BookDB.java
- BookDetails.java
- ShoppingCart.java
- ShoppingCartItem.java

这些类都放在 mypack 包下。

### 5.3.1 实体类

BookDetails 代表了具体的一本书，它的属性和 books 表中字段对应。例程 5-1 是 BookDetails.java 的源代码。

例程 5-1 BookDetails.java

```
package mypack;

public class BookDetails implements Comparable {
    private String bookId = null;
    private String title = null;
    private String name = null;
    private float price = 0.0F;
    private int year = 0;
    private String description = null;
    private int saleAmount;

    public BookDetails(String bookId, String name, String title,
                      float price, int year, String description, int saleAmount) {
        this.bookId = bookId;
        this.title = title;
        this.name = name;
        this.price = price;
        this.year = year;
        this.description = description;
        this.saleAmount = saleAmount;
    }

}
```

```
public String getTitle() {
    return title;
}

public float getPrice() {
    return price;
}

public int getYear() {
    return year;
}

public String getDescription() {
    return description;
}

public String getBookId() {
    return this.bookId;
}

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

public int getSaleAmount(){
    return this.saleAmount;
}

public int compareTo(Object o) {
    BookDetails n = (BookDetails)o;
    int lastCmp = title.compareTo(n.title);
    return (lastCmp);
}
```

### 5.3.2 购物车的实现

ShoppingCart.java 和 ShoppingCartItem.java 分别代表购物车和购物车中的条目，在一个购物车中可以包含多个购物车条目。购物车条目包含了客户购买同一样书的信息和数量。

例如，某个客户的购物车内包含如下内容：

- 《Java 编程指南》两本
- 《Apache 从入门到精通》三本

那么在 ShoppingCart 对象中应该包含两个 ShoppingCartItem 对象。ShoppingCartItem 有两个成员变量：

Object item;

```
int quantity;
```

item 变量代表客户购买的书，它引用 BookDetails 对象，quantity 表示书的数量。

对于以上的例子，第一个 ShoppingCartItem 的 item 变量指向代表《Java 编程指南》的 BookDetails 对象，quantity 为 2；第二个 ShoppingCartItem 的 item 变量指向代表《Apache 从入门到精通》的 BookDetails 对象，quantity 为 3。在这个购物模型中 ShoppingCart、 ShoppingCartItem 和 BookDetails 对象之间的关系如图 5-11 所示。

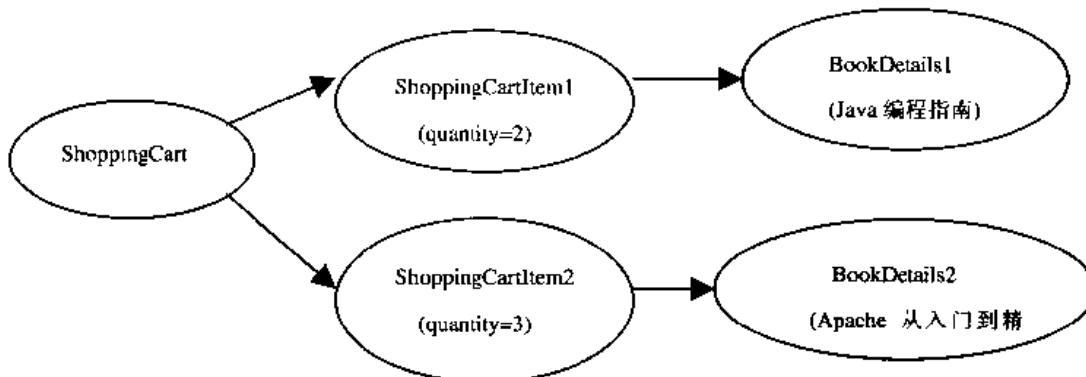


图 5-11 ShoppingCart、ShoppingCartItem 和 BookDetails 对象之间的关系

例程 5-2 是 ShoppingCartItem 的源代码。

例程 5-2 ShoppingCartItem.java

```
package mypack;

public class ShoppingCartItem {
    Object item;
    int quantity;

    public ShoppingCartItem(Object anItem) {
        item = anItem;
        quantity = 1;
    }

    public void incrementQuantity() {
        quantity++;
    }

    public void decrementQuantity() {
        quantity--;
    }

    public Object getItem() {
        return item;
    }
}
```

```

public int getQuantity() {
    return quantity;
}
}

```

ShoppingCart 将 ShoppingCartItem 存放在 HashMap 中，它提供了以下方法：

- public synchronized void add(String bookId, BookDetails book)

将一本书放入购物车中，如果这本书的 ShoppingCartItem 条目已经在购物车中存在，则将 ShoppingCartItem 的 quantity 加 1，否则创建这本书的 ShoppingCartItem 条目。

- public synchronized void remove(String bookId)

从购物车中删除一本书，即将这本书的 ShoppingCartItem 的 quantity 减 1。如果 ShoppingCartItem 的 quantity 小于或等于零，就把 ShoppingCartItem 从 ShoppingCart 中删除。

- public synchronized Collection getItems()

从购物车中返回所有的 ShoppingCartItem 对象的集合。

- public synchronized int getNumberOfItems()

返回购物车中所有书的数量。

- public synchronized double getTotal()

返回购物车中所有书的总金额。

例程 5-3 是 ShoppingCart 的源代码。

例程 5-3 ShoppingCart.java

```

package mypack;

import java.util.*;

public class ShoppingCart {
    HashMap items = null;
    int numberofItems = 0;

    public ShoppingCart() {
        items = new HashMap();
    }

    public synchronized void add(String bookId, BookDetails book) {
        if(items.containsKey(bookId)) {
            ShoppingCartItem scitem = (ShoppingCartItem) items.get(bookId);
            scitem.incrementQuantity();
        } else {
            ShoppingCartItem newItem = new ShoppingCartItem(book);
            items.put(bookId, newItem);
        }
        numberofItems++;
    }
}

```

```
public synchronized void remove(String bookId) {
    if(items.containsKey(bookId)) {
        ShoppingCartItem scitem = (ShoppingCartItem) items.get(bookId);
        scitem.decrementQuantity();

        if(scitem.getQuantity() <= 0)
            items.remove(bookId);

        numberOfItems--;
    }
}

public synchronized Collection getItems() {
    return items.values();
}

protected void finalize() throws Throwable {
    items.clear();
}

public synchronized int getNumberOfItems() {
    return numberOfItems;
}

public synchronized double getTotal() {
    double amount = 0.0;

    for(Iterator i = getItems().iterator(); i.hasNext(); ) {
        ShoppingCartItem item = (ShoppingCartItem) i.next();
        BookDetails bookDetails = (BookDetails) item.getItem();

        amount += item.getQuantity() * bookDetails.getPrice();
    }
    return roundOff(amount);
}

private double roundOff(double x) {
    long val = Math.round(x*100); // cents
    return val/100.0;
}

public synchronized void clear() {
    items.clear();
    numberOfItems = 0;
}
```

## 5.4 发布 bookstore 应用

在本书配套光盘的 sourcecode/bookstores/version0/bookstore 目录下提供了这个 Web 应用的完整的源代码，它的目录结构如图 5-12 所示。

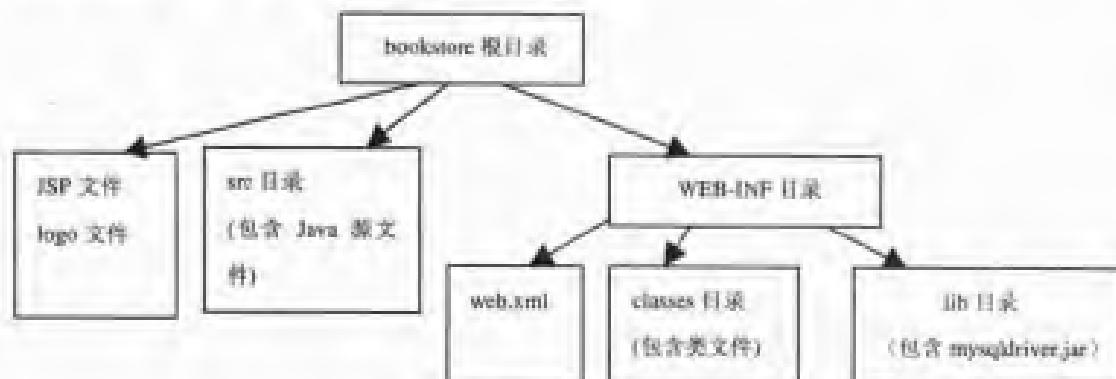


图 5-12 bookstore 应用的目录结构

bookstore 应用在 Windows 资源管理器中的展开图如图 5-13 所示。

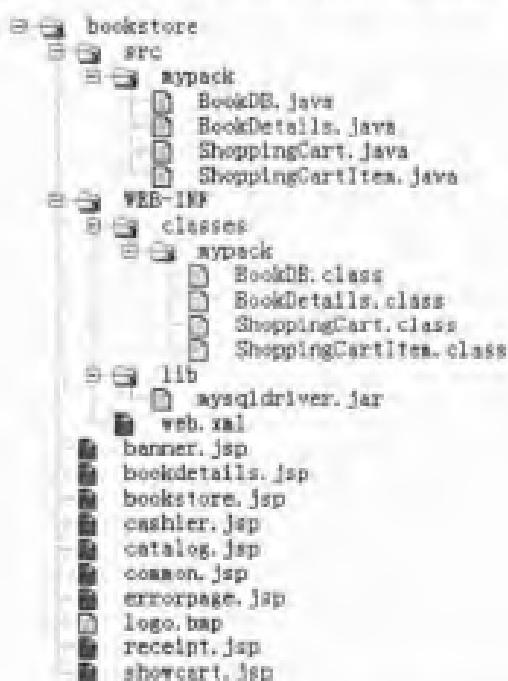


图 5-13 bookstore 应用在 Windows 资源管理器中的展开图

发布 bookstore version0 应用的步骤非常简单。

### 步骤

- (1) 把整个 bookstore 目录拷贝到<CATALINA\_HOME>/webapps 目录下。

(2) 参照 6.1 节(安装和配置 MySQL 数据库), 在 MySQL 中创建 BookDB 数据库。

**提示**

在 2.3 节(配置虚拟主机)中讲过, <Host>元素的 deployOnStartup 属性值默认为 true, 在这种情况下, Tomcat 服务器启动时能自动发布虚拟主机目录下所有的 Web 应用, 即使这个 Web 应用没有在 server.xml 中加入 <Context> 元素, 也可以被发布。

(3) 接下来就可以启动 Tomcat 服务器, 访问 bookstore 应用了。bookstore 应用使用的 Session 依赖于浏览器支持 Cookie, 因此访问 bookstore 应用时, 应该确保浏览器启用了 Cookie。关于 Session 的概念参见本书第 7 章(Session 的使用与管理)。

## 5.5 小结

本章介绍了 bookstore 应用的三层软件结构, 从浏览器端浏览各个 JSP 网页的功能, 还讲解了购物车的实现。在本书的以下章中还将陆续介绍 bookstore 应用涉及的 Web 技术:

- 第 6 章 访问数据库
- 第 8 章 访问 JavaBean
- 第 9 章 用 ant 工具管理 Web 应用
- 第 15 章 采用模板设计网上书店应用
- 第 18 章 Tomcat 与 Jboss 集成

本书一共提供了 4 个版本的 bookstore 应用, 它们完成的功能和提供的客户界面都相同, 但在实现方式上有差别, 这 4 个版本在本书配套光盘上的位置分别为:

- sourcecode/bookstores/version0/bookstore
- sourcecode/bookstores/version1/bookstore
- sourcecode/bookstores/version2/bookstore
- sourcecode/bookstores/version3/bookstore

其中 version1、version2 和 version3 都是对 version0 的改写, 它们的技术侧重点分别为:

- version1: 通过数据源访问数据库, 参见第 6 章
- version2: 采用模板设计 Web 应用, 参见第 15 章
- version3: 创建基于 J2EE 框架的 J2EE 应用, 参见第 18 章

# 第6章 访问数据库

本章首先介绍如何通过 JDBC API 访问数据库，包括 JDBC 驱动程序的概念、java.sql 包中常用类的使用方法以及访问数据库的步骤。接着介绍了数据源的概念以及如何在 Tomcat 中配置数据源，最后举例说明如何在 Web 应用中通过数据源访问数据库。

## 6.1 安装和配置 MySQL 数据库

本书中所有与数据库相关的内容都以 MySQL 作为数据库服务器。MySQL 是一个多用户、多线程的强壮的 SQL 数据库服务器。对 UNIX 和 OS/2 平台，MySQL 提供免费安装软件。在开发 Java Web 应用时，把 Tomcat 与 MySQL 组合，这是非常流行的服务器端软件的搭配方式。

假定 MySQL 安装后的根目录为<MYSQL\_HOME>，在<MYSQL\_HOME>/bin 目录下提供了 mysql.exe，它是 MySQL 的客户程序，它支持在命令行中输入 SQL 语句。MySQL 客户程序的界面如图 6-1 所示。MySQL 安装后，有一个初始用户账号 root，它的初始口令为空。MySQL 的安全验证机制要求用户为 root 账号重新设置口令。

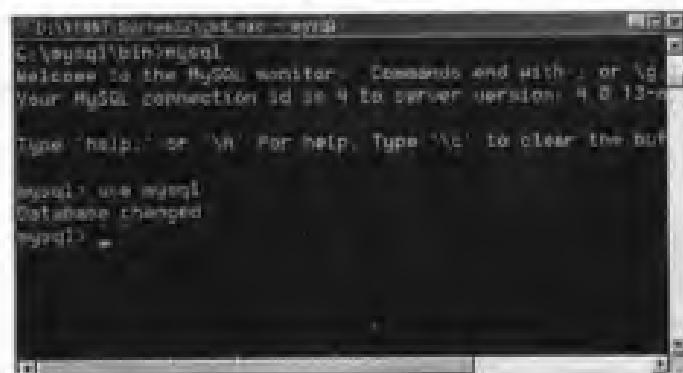


图 6-1 MySQL 的客户程序界面

在本章后面两节访问数据库的例子中，都以 BookDB 数据库为例，在 BookDB 中有一张表 books，在这张表中保存了书的信息。5.1.2 节已对 books 表中的各个字段做了说明。

下面来安装 MySQL 服务器，并且创建 BookDB 数据库。

首先，安装 MySQL 服务器，为 root 账号设置口令，创建新账号：dbuser，口令为：1234。具体步骤如下。

### 步骤

- (1) 在 <http://www.mysql.com> 站点下载 MySQL 的安装软件和 JDBC 驱动程序。此外，在本书配套光盘的 software 目录下也提供了 MySQL 安装软件。

mysql-4.0.14b-win.zip (Windows)

mysql-standard-4.0.14-pc-linux-i686.tar.gz (Linux)

在本书配套光盘的 lib 目录下提供了 MySQL 的驱动程序 mysqldriver.jar 文件。

(2) 安装 MySQL, 启动 MySQL 服务器。

(3) 打开 DOS 界面, 转到<MySQL\_HOME>/bin 目录。

(4) 在 DOS 命令行输入命令: mysql, 进入 MySQL 客户程序, 如图 6-1 所示。

(5) 进入 mysql 数据库, SQL 命令如下:

```
use mysql
```

(6) 为 root 账号重新设置口令, 新的口令为: 1234。SQL 命令如下:

```
UPDATE user SET Password=PASSWORD('1234')
```

```
WHERE user='root';
```

```
FLUSH PRIVILEGES;
```

(7) 退出 mysql 客户程序, SQL 命令如下:

```
exit
```

(8) 再以 root 账号进入 mysql 客户程序, DOS 命令如下:

```
mysql -u root -p
```

当系统提示输入口令时, 输入: “1234”。

(9) 进入 mysql 数据库, 创建一个新的账号: dbuser, 口令为: 1234, SQL 命令如下:

```
use mysql;
GRANT ALL PRIVILEGES ON *.* TO dbuser@localhost IDENTIFIED BY '1234' WITH
GRANT OPTION;
```

其次, 创建数据库 BookDB 和 books 表, 向 books 表中插入数据, 具体步骤如下(在本书配套光盘的 sourcecode/chapter6/目录下, 提供了创建 books 表的脚本文件 books.sql)。



(1) 创建数据库 BookDB, SQL 命令如下:

```
CREATE DATABASE BookDB;
```

(2) 进入 BookDB 数据库, SQL 命令如下:

```
use BookDB
```

(3) 在 BookDB 数据库中创建 books 表, SQL 命令如下:

```
CREATE TABLE books
(id VARCHAR(8)
PRIMARY KEY,
name VARCHAR(24),
title VARCHAR(96),
price FLOAT,
yr INT,
description VARCHAR(30),
saleAmount INT);
```

(4) 在 books 表中加入一些记录, SQL 命令如下:

```
INSERT INTO books VALUES('201', '王芳',
'Java 编程指南',
```

33.75, 1999, '让读者轻轻松松掌握 Java 语言', 1000);

```
INSERT INTO books VALUES('202', '张丙',
    'Weblogic 技术参考', 45.99,
    2002, '真的不错耶', 2000);
```

```
INSERT INTO books VALUES('203', '孙艳',
    'Oracle 数据库教程',
    40, 2003, '关于 Oracle 的最畅销的技术书', 2000);
```

```
INSERT INTO books VALUES('204', '大卫',
    '从 Oak 到 Java: 语言的革命',
    20.75, 1998, '很值得一看', 2000);
```

```
INSERT INTO books VALUES('205', '阿明',
    'Apache 从入门到精通',
    50.75, 2002, '权威的 Apache 技术资料', 2000);
```

```
INSERT INTO books VALUES('206', '洪军',
    'Java 与数据算法',
    54.75, 2002, '权威的 Java 技术资料', 2000);
```

## 6.2 通过 JDBC 访问数据库

JDBC 是 Java DataBase Connectivity 的缩写。java.sql 包提供了 JDBC API，程序员可以通过它编写访问数据库的程序。在 java.sql 包里定义了访问数据库的接口和类。

JDBC API 并不能直接访问数据库，它依赖于数据库厂商提供的 JDBC Driver (JDBC 驱动程序)。Java 程序和 JDBC 驱动程序的关系如图 6-2 所示。

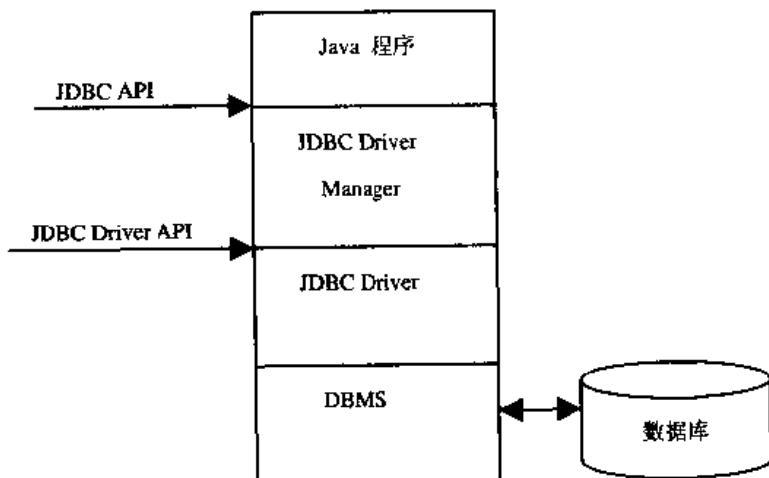


图 6-2 Java 程序与 JDBC 驱动程序

### 6.2.1 java.sql 包中的接口和类

在 java.sql 包中常用的接口和类包括：

- Driver 接口和 DriverManager 类
- Connection
- Statement
- PreparedStatement
- ResultSet

这些类之间的关系如图 6-3 所示。

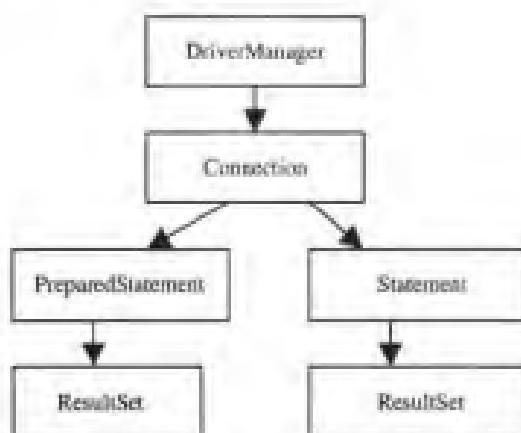


图 6-3 java.sql 包中的类之间的关系

#### 1. Driver 接口和 DriverManager 类

所有 JDBC 驱动程序都必须实现 Driver 接口，JDBC 驱动程序由数据库厂商提供。因此在编写访问数据库的 Java 程序时，必须把特定数据库的 JDBC 驱动程序包加入到 classpath 中。

DriverManager 用来建立和数据库的连接以及管理 JDBC 驱动程序。DriverManager 的常用方法说明参见表 6-1。

表 6-1 DriverManager 的常用方法

方 法	描 述
registerDriver (Driver driver)	在 DriverManager 中注册 JDBC 驱动程序
getConnection (String url, String user, String pwd)	建立和数据库的连接，返回 Connection 对象
setLoginTimeout (int seconds)	设定等待数据库连接的最长时间
setLogWriter (PrintWriter out)	设定输出数据库日志的 PrintWriter 对象

#### 2. Connection

Connection 代表 Java 程序和数据库的连接。Connection 的常用方法说明参见表 6-2。

表 6-2 Connection 的常用方法

方 法	描 述
getMetaData()	返回数据库的 MetaData 数据。MetaDta 数据包含了数据库的相关信息，例如当前数据库连接的用户名、使用的 JDBC 驱动程序、数据库允许的最大连接数，以及数据库的版本等。
createStatement()	创建并返回 Statement 对象。
prepareStatement(String sql)	创建并返回 PreparedStatement 对象。

### 3. Statement

Statement 用来执行静态的 SQL 语句。例如：对于 insert、update 和 delete 语句，可以调用 executeUpdate(String sql)方法；对于 select 语句，可以调用 executeQuery(String sql)方法，这个方法返回一个 ResultSet 对象，例如：

```
String sql="select id,name,title,price from books where name='Tom' and price=40";
ResultSet rs=stmt.executeQuery(sql); // stmt 为 Statement 对象
```

### 4. PreparedStatement

PreparedStatement 用来执行动态的 SQL 语句。在访问数据库时，可能会遇到这样的情况，某条 SQL 语句被多次执行，但每次的参数不同。例如：

```
select id,name,title,price from books where name='Tom' and price=40
select id,name,title,price from books where name='Mike' and price=30
select id,name,title,price from books where name='Jack' and price=50
```

以上 SQL 语句的格式为：

```
select id,name,title,price from books where name=? and price=?
```

在这种情况下，用 PreparedStatement 比 Statement 更方便，因为 PreparedStatement 允许 sql 语句中包含参数。

PreparedStatement 的使用步骤如下。



(1) 生成 PreparedStatement 对象。例如：以下 SQL 语句中 name 的值和 price 的值都用“?”代替，它们表示两个可被替换的参数：

```
String selectStatement = "select id,name,title,price from books where name = ? and price = ? ";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
```

(2) 调用 PreparedStatement 的 setXXX 方法，给参数赋值：

```
prepStmt.setString(1, name); //假定已定义 name 变量
prepStmt.setFloat(2, price); //假定已定义 price 变量
```

(3) 执行 SQL 语句：

```
ResultSet rs = prepStmt.executeQuery();
```

### 5. ResultSet

ResultSet 表示 select 语句查询得到的记录集合。ResultSet 的记录行号从 1 开始，一个 Statement 对象在同一时刻只能打开一个 ResultSet 对象。调用 ResultSet 的 next()方法，可以使游标定位到下一条记录。调用 ResultSet 的 getXXX()方法，可以取得某个字段的值。

ResultSet 的常用的 getXXX 方法见表 6-3。

表 6-3 ResultSet 的常用的 getXXX 方法

目 录	描 述
getString(int columnIndex)	返回指定字段的 String 类型的值，参数 columnIndex 代表字段的序号
getString(String columnName)	返回指定字段的 String 类型的值，参数 columnName 代表字段的名字
getInt(int columnIndex)	返回指定字段的 int 类型的值，参数 columnIndex 代表字段的序号
getInt(String columnName)	返回指定字段的 int 类型的值，参数 columnName 代表字段的名字
getFloat(int columnIndex)	返回指定字段的 float 类型的值，参数 columnIndex 代表字段的序号
getFloat(String columnName)	返回指定字段的 float 类型的值，参数 columnName 代表字段的名字

ResultSet 提供了 getString()、getInt() 和 getFloat() 等方法。应该根据字段的类型来确定调用哪种 getXXX() 方法。既可以通过字段的序号来指定字段，也可以通过字段的名字来指定字段。

例如对于以下的 select 查询语句，记录集合存放在 rs 变量中：

```
String sql=" select id,name,title,price from books where name='Tom' and price=40";
ResultSet rs=stmt.executeQuery(sql);
```

如果要访问 id 字段，可以用以下的语句：

```
rs.getString(1);
```

或者

```
rs.getString("id");
```

如果要访问 price 字段，可以用以下的语句：

```
rs.getFloat(4);
```

或者

```
rs.getFloat("price");
```

如果要取出 ResultSet 中所有记录，可以采用下面的循环语句：

```
while (rs.next())
{
    String col1 = rs.getString(1);
    String col2 = rs.getString(2);
    String col3 = rs.getString(3);
    float col4 = rs.getFloat(4);

    // 打印当前记录
    System.out.println("id=" + col1 + " name=" + col2 + " title=" + col3 + " price=" + col4);
}
```

## 6.2.2 编写访问数据库程序的步骤

在 Java 程序中，通过 JDBC 访问数据库包含以下步骤。



(1) 装载并注册数据库的 JDBC 驱动程序，其中 JDBC-ODBC Driver 是在 JDK 中自

带的，默认已经注册，所以不需要再注册。以下分别给出了装载 JDBC-ODBC 驱动程序，装载并注册 SQLServer 驱动程序、Oracle 驱动程序和 MySQL 驱动程序的代码：

```
//装载 JdbcOdbcDriver class
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver") ;

//装载并注册 SQLServer Driver
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
java.sql.DriverManager.registerDriver (new com.microsoft.jdbc.sqlserver.SQLServerDriver ()) ;

//装载并注册 OracleDriver
Class.forName("oracle.jdbc.driver.OracleDriver");
java.sql.DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ()) ;

//装载并注册 MySQLDriver
Class.forName("com.mysql.jdbc.Driver");
java.sql.DriverManager.registerDriver(new com.mysql.jdbc.Driver()); //不是必要步骤
```



在旧的 MySQL 驱动程序版本中，Driver 类为 org.gjt.mm.mysql.Driver。目前新的版本尽管保留了这个类，但提倡使用新的 Driver 类，即 com.mysql.jdbc.Driver。

有些驱动程序类在被加载的时候，能自动创建本身实例，然后调用 DriverManager.registerDriver() 方法注册自身。例如对于 MySQL 的驱动程序类 com.mysql.jdbc.Driver，在 JVM（Java 虚拟机）加载这个类时，会执行它的如下静态代码块：

```
// Register ourselves with the DriverManager
static {
    try {
        java.sql.DriverManager.registerDriver(new Driver());
    } catch (java.sql.SQLException E) {
        throw new RuntimeException("Can't register driver!");
    }
    ...
}
```

所以在程序中其实只要通过 Class.forName 方法加载 MySQL Driver 类即可，可以不必再注册驱动程序类。

### （2）建立与数据库的连接。

```
Connection con = java.sql.DriverManager.getConnection(dburl,user,password);
```

getConnection()方法中有3个参数，dburl 表示连接数据库的 JDBC URL，user 和 password 分别表示连接数据库的用户名和口令。

JDBC URL 的一般形式为：

```
jdbc:drivertype:driversubtype://parameters
```

drivertype 表示驱动程序的类型，driversubtype 是可选的参数，parameters 通常用来设

定数据库服务器的 IP 地址、端口号和数据库的名称。以下给出了几种常用的数据库的 JDBC URL 形式：

- 如果通过 JDBC-ODBC Driver 连接数据库，采用如下形式：

jdbc:odbc:datasource

- 对于 Oracle 数据库连接，采用如下形式：

jdbc:oracle:thin:@localhost:1521:sid

- 对于 SQLServer 数据库连接，采用如下形式：

jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=BookDB

- 对于 MySQL 数据库连接，采用如下形式：

jdbc:mysql://localhost:3306/BookDB

(3) 创建 Statement 对象，准备调用 SQL 语句：

```
Statement stmt = con.createStatement();
```

(4) 调用 SQL 语句：

```
String sql="select id,name,title,price from books where name='Tom' and price=40";
ResultSet rs=stmt.executeQuery(sql);
```

(5) 访问 ResultSet 中的记录集：

```
while (rs.next())
{
    String col1 = rs.getString(1);
    String col2 = rs.getString(2);
    String col3 = rs.getString(3);
    float col4 = rs.getFloat(4);

    //打印当前记录
    System.out.println("id="+col1+" name="+col2+" title="+col3+" price="+col4);
}
```

(6) 依次关闭 ResultSet、Statement 和 Connection 对象：

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

### 6.2.3 事务处理

在数据库操作中，一项事务是指由一条或多条对数据库更新的 SQL 语句所组成的一个不可分割的工作单元。只有当事务中的所有操作都正常完成，整个事务才能被提交到数据库；如果有一项操作没有完成，就必须撤销整个事务。

例如在银行转账事务中，假定张三从自己的账号上把 1000 元钱转到李四的账号上，相关的 SQL 语句如下：

```
update account set money=money-1000 where name='zhangsan';
update account set money=money+1000 where name='lisi';
```

这两条 SQL 语句必须作为一个完整的事务来处理，也就是说，只有当两条 SQL 语句都成功地执行，才能提交整个事务。只要有一条语句执行失败，整个事务必须撤销，否则

会导致数据库中的数据不一致。例如，假定第一条语句执行成功了，第二条语句却执行失败（例如执行第二条语句时数据库服务器刚好关机），张三户头上的钱少了1000元，李四户头上的钱却没多出来，那么银行转账事务就混乱了。

在 Connection 类中提供了3个控制事务的方法：

- setAutoCommit(boolean autoCommit): 设置是否自动提交事务
- commit(): 提交事务
- rollback(): 撤销事务

在 JDBC API 中，默认情况下为自动提交事务。也就是说，每一条对数据库更新的 SQL 语句代表一项事务，操作成功后，系统将自动调用 commit() 来提交，否则将调用 rollback() 来撤销事务。

在 JDBC API 中，可以通过调用 setAutoCommit(false) 来禁止自动提交事务。然后就可以把多条更新数据库的 SQL 语句作为一个事务，在所有操作完成后，调用 commit() 来进行整体提交。倘若其中一项 SQL 操作失败，就不会执行 commit()，而是产生相应的 SQLException，此时就可以在捕获异常的代码块中调用 rollback() 方法撤销事务。示例如下：

```

try {
    con = java.sql.DriverManager.getConnection(dbUrl, dbUser, dbPwd);
    //禁止自动提交，设置回滚点
    con.setAutoCommit(false);
    stmt = con.createStatement();
    //数据库更新操作 1
    stmt.executeUpdate("update account set money=money-1000 where name='zhangsan'");
    //数据库更新操作 2
    stmt.executeUpdate("update account set money=money+1000 where name='lisi'");
    con.commit(); //事务提交
} catch(Exception ex) {
    ex.printStackTrace();
    try {
        con.rollback(); //操作不成功则回滚
    } catch(Exception e){
        e.printStackTrace();
    }
} finally{
    try{
        stmt.close();
        con.close();
    } catch(Exception e){
        e.printStackTrace();
    }
}
}

```

#### 6.2.4 通过 JDBC 访问数据库的 JSP 范例程序

以下是一个 JSP 访问数据库的范例程序（例程 6-1），名为 DbJsp.jsp。在这个程序中，

建立了和数据库的连接，向 books 表中加入了一条记录，然后将 books 中所有记录以表格的方式显示出来，最后删除新加的记录。

例程 6-1 DbJsp.jsp

```
<!--首先导入一些必要的 packages-->
<%@ page import="java.io.*"%>
<%@ page import="java.util.*"%>
<!--告诉编译器使用 SQL 包-->
<%@ page import="java.sql.*"%>
<!--设置中文输出-->
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
    <title>DbJsp.jsp</title>
</head>
<body>
<%
//以 try 开始
try
{
    Connection con;
    Statement stmt;
    ResultSet rs;
    //加载驱动程序，下面的代码为加载 MySQL 驱动程序
    Class.forName("com.mysql.jdbc.Driver");
    //注册 MySQL 驱动程序
    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    //用适当的驱动程序连接到数据库
    String dbUrl = "jdbc:mysql://localhost:3306/BookDB?useUnicode=true&characterEncoding=GB2312";
    String dbUser="dbuser";
    String dbPwd="1234";
    //建立数据库连接
    con = java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);
    //创建一个 JDBC 声明
    stmt = con.createStatement();
    //增加新记录
    stmt.executeUpdate("INSERT INTO books (id,name,title,price) VALUES ('999','Tom','Tomcat Bible',44.5)");
    //查询记录
    rs = stmt.executeQuery("SELECT id,name,title,price from books");
    //输出查询结果
    out.println("<table border=1 width=400>");
    while (rs.next())
    {

```

```

String col1 = rs.getString(1);
String col2 = rs.getString(2);
String col3 = rs.getString(3);
float col4 = rs.getFloat(4);
//打印所显示的数据
out.println("<tr><td>" + col1 + "</td><td>" + col2 + "</td><td>" + col3 + "</td><td>" + col4 + "</td></tr>");
}
out.println("</table>");

//删除新增加的记录
stmt.executeUpdate("DELETE FROM books WHERE id='999'");

//关闭数据库连接
rs.close();
stmt.close();
con.close();
}

//捕获错误信息
catch (Exception e) {out.println(e.getMessage());}
%>
</body>
</html>

```

把 DbJsp.jsp 拷贝到 <CATALINA\_HOME>/webapps/bookstore 目录下，然后把 mysqldriver.jar 文件拷贝到 <CATALINA\_HOME>/common/lib 目录或 <CATALINA\_HOME>/webapps/bookstore/WEB-INF/lib 目录下。

访问 <http://localhost:8080/bookstore/DbJsp.jsp>，输出结果如图 6-4 所示。



图 6-4 DbJsp.jsp 生成的网页

### 6.2.5 在 bookstore 应用中通过 JDBC 访问数据库

BookDB.java 负责访问数据库，它提供了操纵数据库的所有方法，包括：

- public Collection getBooks(): 从 books 表中读取所有书的信息，放在 Collection 集合中
- public int getNumberOfBooks(): 从 books 表中获取所有书的销售数量
- public BookDetails getBookDetails(String bookId): 根据 bookId 读取某一本书的详细信息
- public void buyBooks(ShoppingCart cart): 根据购物车中的内容，更新 books 表。该方法调用 buyBook(String bookId, int quantity, Connection con) 方法，完成实际的 SQL 操作
- public void buyBook(String bookId, int quantity, Connection con): 完成实际购买书的 SQL 操作，执行的 SQL 语句为：update books set saleAmount=saleAmount+quantity where id=bookId

本书提供了 bookstore 应用的 4 种版本，其中 version1 的 BookDB 通过 DataSource 访问数据库，其他版本都通过 JDBC 访问数据库。下面讲述 version0 中 BookDB 的实现，相关的代码位于本书配套光盘的 sourcecode/bookstores/version0/bookstore/src/mypack 目录下。

在 BookDB 的构造方法中通过 Class.forName() 方法装载 MySQL 的 JDBC 驱动程序：

```
public BookDB () throws Exception{
    Class.forName("com.mysql.jdbc.Driver");
}
```

每次访问数据库时，都调用 BookDB 自身的 getConnection() 方法，在这个方法中建立和数据库的连接，并返回 Connection 对象：

```
public Connection getConnection()throws Exception{
    return java.sql.DriverManager.getConnection(dbUrl, dbUser, dbPwd);
}
```

当数据库访问结束后，应该依次关闭 ResultSet、PreparedStatement(或 Statement) 和 Connection 对象，从而释放数据库连接占用的资源。在 BookDB.java 中定义了 3 个方法分别关闭这 3 种对象：

```
closeResultSet(ResultSet rs)
closePrepStmt(PreparedStatement prepStmt)
closeConnection(Connection con)
```

为了确保数据库访问结束后， releaseConnection() 方法一定被执行，所有访问数据库的方法都采用如下结构：

```
Connection con=null;
PreparedStatement prepStmt=null;
ResultSet rs =null;
try {
    con=getConnection();
    //access database
```

```
....  
}finally{  
    closeResultSet(rs);  
    closePrepStmt(prepStmt);  
    closeConnection(con);  
}
```

例程 6-2 给出了 BookDB.java 的源代码。

例程 6-2 BookDB.java

```
/** access mysql database through JDBC Driver */  
package mypack;  
import java.sql.*;  
import javax.naming.*;  
import javax.sql.*;  
import java.util.*;  
  
public class BookDB {  
  
    private ArrayList books;  
    private String dbUrl = "jdbc:mysql://localhost:3306/BookDB";  
    private String dbUser="dbuser";  
    private String dbPwd="1234";  
  
    public BookDB () throws Exception{  
        Class.forName("com.mysql.jdbc.Driver");  
    }  
  
    public Connection getConnection()throws Exception{  
        return java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);  
    }  
  
    public void closeConnection(Connection con){  
        try{  
            if(con!=null) con.close();  
        }catch(Exception e){  
            e.printStackTrace();  
        }  
    }  
  
    public void closePrepStmt(PreparedStatement prepStmt){  
        try{  
            if(prepStmt!=null) prepStmt.close();  
        }catch(Exception e){  
            e.printStackTrace();  
        }  
    }  
}
```

```
        }

    }

    public void closeResultSet(ResultSet rs){
        try{
            if(rs!=null) rs.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public int getNumberOfBooks() throws Exception {
        Connection con=null;
        PreparedStatement prepStmt=null;
        ResultSet rs=null;
        books = new ArrayList();

        try {
            con=getConnection();
            String selectStatement = "select * " + "from books";
            prepStmt = con.prepareStatement(selectStatement);
            rs = prepStmt.executeQuery();

            while (rs.next()) {
                BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
                    rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
                books.add(bd);
            }
        } finally{
            closeResultSet(rs);
            closePrepStmt(prepStmt);
            closeConnection(con);
        }
        return books.size();
    }

    public Collection getBooks()throws Exception{
        Connection con=null;
        PreparedStatement prepStmt=null;
        ResultSet rs =null;
        books = new ArrayList();
        try {
            con=getConnection();
            String selectStatement = "select * " + "from books";

```

```
prepStmt = con.prepareStatement(selectStatement);
rs = prepStmt.executeQuery();

while (rs.next()) {

    BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
        rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
    books.add(bd);
}

}finally{
    closeResultSet(rs);
    closePrepStmt(prepStmt);
    closeConnection(con);
}

Collections.sort(books);
return books;
}

public BookDetails getBookDetails(String bookId) throws Exception {
    Connection con=null;
    PreparedStatement prepStmt=null;
    ResultSet rs =null;
    try {
        con=getConnection();
        String selectStatement = "select * " + "from books where id = ? ";
        prepStmt = con.prepareStatement(selectStatement);
        prepStmt.setString(1, bookId);
        rs = prepStmt.executeQuery();

        if (rs.next()) {
            BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
                rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
            prepStmt.close();

            return bd;
        }
        else {
            return null;
        }
    }finally{
        closeResultSet(rs);
        closePrepStmt(prepStmt);
        closeConnection(con);
    }
}
```

```
        }
    }

    public void buyBooks(ShoppingCart cart) throws Exception {
        Connection con=null;
        Collection items = cart.getItems();
        Iterator i = items.iterator();
        try {
            con=getConnection();
            con.setAutoCommit(false);
            while (i.hasNext()) {
                ShoppingCartItem sci = (ShoppingCartItem)i.next();
                BookDetails bd = (BookDetails)sci.getItem();
                String id = bd.getBookId();
                int quantity = sci.getQuantity();
                buyBook(id, quantity,con);
            }
            con.commit();
            con.setAutoCommit(true);
        } catch (Exception ex) {
            con.rollback();
            throw ex;
        }finally{
            closeConnection(con);
        }
    }

    public void buyBook(String bookId, int quantity,Connection con) throws Exception {
        PreparedStatement prepStmt=null;
        ResultSet rs=null;
        try{
            String selectStatement = "select * " + "from books where id = ? ";
            prepStmt = con.prepareStatement(selectStatement);
            prepStmt.setString(1, bookId);
            rs = prepStmt.executeQuery();

            if (rs.next()){
                prepStmt.close();
                String updateStatement =
                    "update books set saleamount = saleamount + ? where id = ?";
                prepStmt = con.prepareStatement(updateStatement);
                prepStmt.setInt(1, quantity);
                prepStmt.setString(2, bookId);
            }
        }
    }
}
```

```
prepStmt.executeUpdate();
prepStmt.close();
```

```
} finally {
    closeResultSet(rs);
    closePrepStmt(prepStmt);
}
```

## 6.3 数据源（DataSource）简介

JDBC 2.0 提供了 javax.sql.DataSource 接口，它负责建立与数据库的连接，在应用程序中访问数据库时不必编写连接数据库的代码，可以直接从数据源获得数据库连接。

### 1. 数据源和连接池

在 DataSource 中事先建立了多个数据库连接，这些数据库连接保存在连接池（Connect Pool）中。Java 程序访问数据库时，只需从连接池中取出空闲状态的数据库连接；当程序访问数据库结束，再将数据库连接放回连接池，这样做可以提高访问数据库的效率。试想如果 Web 应用每次接收到客户请求，都和数据库建立一个连接，数据库操作结束就断开连接，这样会耗费大量的时间和资源。因为数据库每次配置连接都要将 Connection 对象加载到内存中，再验证用户名和密码。

### 2. 数据源和 JNDI 资源

DataSource 对象是由 Tomcat 提供的，因此不能在程序中采用创建一个实例的方式来生成 DataSource 对象，而需要采用 Java 的另一个技术 JNDI（Java Naming and Directory Interface），来获得 DataSource 对象的引用。

可以简单地把 JNDI 理解为一种将对象和名字绑定的技术，对象工厂负责生产出对象，这些对象都和惟一的名字绑定。外部程序可以通过名字来获得某个对象的引用。

在 javax.naming 包中提供了 Context 接口，该接口提供了将对象和名字绑定，以及通过名字检索对象的方法。Context 中的主要方法描述参见表 6-4。

表 6-4 Context 接口的方法

方 法	描 述
bind(String name, Object object)	将对象与一个名字绑定
lookup(String name)	返回与指定的名字绑定的对象

外部应用程序访问对象工厂中的对象的过程如图 6-5 所示。

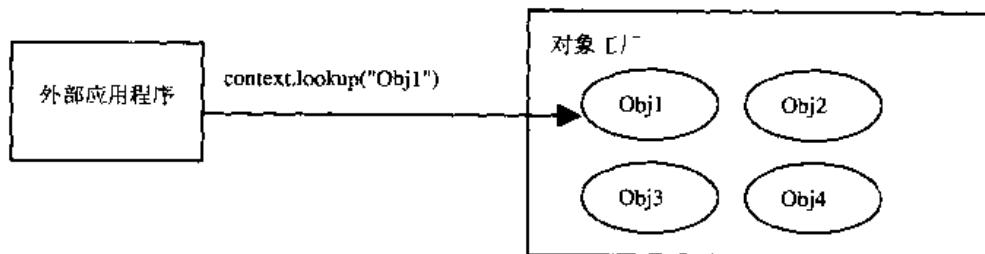


图 6-5 外部应用程序访问对象工厂中的对象

Tomcat 把 DataSource 作为一种可配置的 JNDI 资源来处理。生成 DataSource 对象的工厂为 org.apache.commons.dbcp.BasicDataSourceFactory。假定配置了两个 DataSource，一个名为 jdbc/BookDB，还有一个名为 jdbc/BankDB，bookstore 应用访问名为 jdbc/BookDB 的 DataSource 的过程如图 6-6 所示。

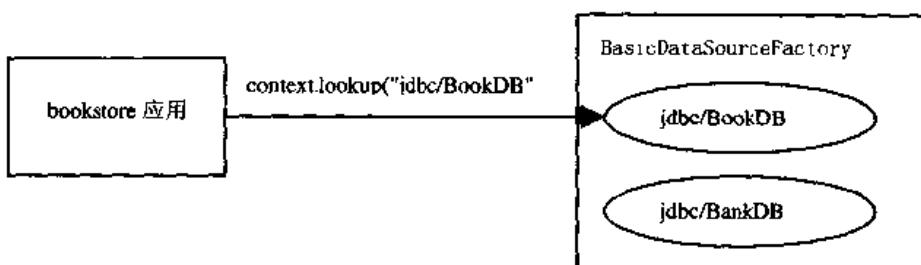


图 6-6 bookstore 应用程序访问 jdbc/BookDB 数据源

## 6.4 配置数据源

数据源的配置涉及修改 server.xml 和 web.xml 文件，在 server.xml 中加入定义数据源的元素<Resource>，在 web.xml 中加入<resource-ref>元素，声明该 Web 应用所引用的数据源。

### 6.4.1 在 server.xml 中加入<Resource>元素

<Resource>元素用来定义 JNDI Resource。在 Tomcat 中，DataSource 是 JNDI Resource 的一种。以下代码为 bookstore 应用定义了一个名为 jdbc/BookDB 的数据源。

```
<Context path="/bookstore" docBase="bookstore" debug="0"
reloadable="true">
```

```
<Resource name="jdbc/BookDB"
auth="Container"
type="javax.sql.DataSource" />
```

```
<ResourceParams name="jdbc/BookDB">
<parameter>
<name>factory</name>
```

```
<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>

<!-- Maximum number of dB connections in pool. Make sure you
    configure your mysqld max_connections large enough to handle
    all of your db connections. Set to 0 for no limit.
    -->
<parameter>
    <name>maxActive</name>
    <value>100</value>
</parameter>

<!-- Maximum number of idle dB connections to retain in pool.
    Set to 0 for no limit.
    -->
<parameter>
    <name>maxIdle</name>
    <value>30</value>
</parameter>

<!-- Maximum time to wait for a dB connection to become available
    in ms, in this example 10 seconds. An Exception is thrown if
    this timeout is exceeded. Set to -1 to wait indefinitely.
    Maximum time to wait for a dB connection to become available
    in ms, in this example 10 seconds. An Exception is thrown if
    this timeout is exceeded. Set to -1 to wait indefinitely.
    -->
<parameter>
    <name>maxWait</name>
    <value>10000</value>
</parameter>

<!-- MySQL dB username and password for dB connections -->
<parameter>
    <name>username</name>
    <value>dbuser</value>
</parameter>
<parameter>
    <name>password</name>
    <value>1234</value>
</parameter>

<!-- Class name for mm.mysql JDBC driver -->
<parameter>
    <name>driverClassName</name>
    <value>com.mysql.jdbc.Driver</value>
```

```

</parameter>

<!-- The JDBC connection url for connecting to your MySQL dB.
    The autoReconnect=true argument to the url makes sure that the
    mm.mysql JDBC Driver will automatically reconnect if mysqld closed the
    connection.  mysqld by default closes idle connections after 8 hours.
-->

<parameter>
    <name>url</name>
    <value>jdbc:mysql://localhost:3306/BookDB?autoReconnect=true</value>
</parameter>
</ResourceParams>

</Context>

```

在以上代码中，定义了<Resource>和<ResourceParams>元素，<Resource>的属性描述参见表 6-5。

表 6-5 <Resource>的属性

属性	描述
name	指定 Resource 的 JNDI 名字
auth	指定管理 Resource 的 Manager，它有两个可选值：Container 和 Application。Container 表示由容器来创建和管理 Resource，Application 表示由 Web 应用来创建和管理 Resource
type	指定 Resource 所属的 Java 类名

在<ResourceParam>元素中指定了配置 BookDB 数据源的参数，<ResourceParam>元素的参数说明见表 6-6。

表 6-6 <ResourceParam>的参数

参数	描述
factory	指定生成 DataSource 的 factory 的类名
maxActive	指定数据库连接池中处于活动状态的数据库连接的最大数目，取值为 0，表示不受限制
maxIdle	指定数据库连接池中处于空闲状态的数据库连接的最大数目，取值为 0，表示不受限制
maxWait	指定数据库连接池中的数据库连接处于空闲状态的最长时间(以毫秒为单位)，超过这一时间，将会抛出异常。取值为 -1，表示可以无限期等待
username	指定连接数据库的用户名
password	指定连接数据库的口令
driverClassName	指定连接数据库的 JDBC 驱动程序
url	指定连接数据库的 URL

#### 6.4.2 在 web.xml 中加入<resource-ref>元素

如果 Web 应用访问了由 Servlet 容器管理的某个 JNDI Resource，必须在 web.xml 文件中声明对这个 JNDI Resource 的引用。表示资源引用的元素为<resource-ref>，以下是声明引

用 jdbc/BookDB 数据源的代码：

```
<webapp>
<resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/BookDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
</webapp>
```

<resource-ref>的属性描述参见表 6-7。

表 6-7 <resource-ref>的属性

属性	描述
description	对所引用的资源的说明
res-ref-name	指定所引用资源的 JNDI 名字，与<Resource>元素中的 name 属性对应
res-type	指定所引用资源的类名字，与<Resource>元素中的 type 属性对应
res-auth	指定管理所引用资源的 Manager，与<Resource>元素中的 auth 属性对应

## 6.5 程序中访问数据源

javax.naming.Context 提供了查找 JNDI Resource 的接口，例如，可以通过以下代码获得 jdbc/BookDB 数据源的引用：

```
Context ctx = new InitialContext();
DataSource ds=(DataSource)ctx.lookup("java:comp/env/jdbc/BookDB");
```

得到了 DataSource 对象的引用后，就可以通过 DataSource 的 getConnection()方法获得数据库连接对象 Connection：

```
Connection con=ds.getConnection();
```

当程序结束数据库访问后，应该调用 Connection 的 close()方法，及时将 Connection 返回数据库连接池，使 Connection 恢复空闲状态。

### 6.5.1 通过数据源访问数据库的 JSP 范例程序

例程 6-3 是一个访问 jdbc/BookDB 数据源的例子，它与 DbJsp.jsp 完成的功能相同，两者的代码很相似，不同之处在于获得数据库连接的方式不一样。

例程 6-3 DbJsp1.jsp

---

```
<!--首先导入一些必要的 packages-->
<%@ page import="java.io.*"%>
<%@ page import="java.util.*"%>
<%@ page import="java.sql.*"%>
<%@ page import="javax.sql.*"%>
<%@ page import="javax.naming.*"%>
```

```
<!--设置中文输出-->
<%@ page contentType="text/html; charset=GB2312" %>
<html>
<head>
    <title>DbJsp1.jsp</title>
</head>
<body>
<%
//以 try 开始
try
{
    Connection con;
    Statement stmt;
    ResultSet rs;

//建立数据库连接
    Context ctx = new InitialContext();
    DataSource ds =(DataSource)ctx.lookup("java:comp/env/jdbc/BookDB");
    con = ds.getConnection();

//创建一个 JDBC 声明
    stmt = con.createStatement();
//增加新记录
    stmt.executeUpdate("INSERT INTO books (id,name,title,price) VALUES ('999','Tom','Tomcat
Bible',44.5)");
//查询记录
    rs = stmt.executeQuery("SELECT id,name,title,price from books");
//输出查询结果
    out.println("<table border=1 width=400>");
    while (rs.next())
    {
        String col1 = rs.getString(1);
        String col2 = rs.getString(2);
        String col3 = rs.getString(3);
        float col4 = rs.getFloat(4);

//convert character encoding
        col1=new String(col1.getBytes("ISO-8859-1"),"GB2312");
        col2=new String(col2.getBytes("ISO-8859-1"),"GB2312");
        col3=new String(col3.getBytes("ISO-8859-1"),"GB2312");

//打印所显示的数据
        out.println("<tr><td>" + col1 + "</td><td>" + col2 + "</td><td>" + col3 + "</td><td>" + col4 + "</td></tr>");
    }
    out.println("</table>");
```

```

//删除新增加的记录
stmt.executeUpdate("DELETE FROM books WHERE id='999'");

//关闭数据库连接
rs.close();
stmt.close();
con.close();
}

//捕获错误信息
catch (Exception e) {out.println(e.getMessage());}

%>
</body>
</html>

```

假定将 DbJsp1.jsp 发布到 bookstore 应用中，首先应该按照 6.4.1 和 6.4.2 小节的步骤修改 server.xml 和 bookstore 应用的 web.xml 文件。在本书配套光盘的 sourcecode/chapter6 目录下提供了两个文件：server\_modify.xml 和 web\_modify.xml 文件，这两个文件分别提供了修改 server.xml 和 web.xml 的代码。此外，应该将 MySQL 的 JDBC 驱动程序拷贝到 <CATALINA\_HOME>/common/lib 目录下。通过浏览器访问 DbJsp1.jsp：

<http://localhost:8080/bookstore/DbJsp1.jsp>

会看到生成的网页与 DbJsp.jsp 生成的网页一样，如图 6-4 所示。



如果直接通过 JDBC 访问数据库，则可以把 JDBC 驱动程序拷贝到 /bookstore/<WEB-INF>/lib 目录或者<CATALINA\_HOME>/common/lib 目录下；如果通过数据源访问数据库，由于数据源由 Servlet 容器创建并维护，所以必须把 JDBC 驱动程序拷贝到<CATALINA\_HOME>/common/lib 目录下，确保 Servlet 容器能访问驱动程序。关于这两个 lib 目录的区别，可以参见本书 2.1 节（Tomcat 的目录结构）。

## 6.5.2 在 bookstore 应用中通过数据源访问数据库

在 bookstore version0 版本中，对于每一个需要访问数据库的客户请求，BookDB.java 都会首先建立与数据库的连接，然后再完成相关的事务。频繁的连接数据库，会影响 Web 服务器的工作效率，降低服务器响应客户请求的速度。

为了提高访问数据库的效率，在 bookstore version1 版本中通过 JNDI DataSource 来访问数据库。对于每一个需要访问数据库的客户请求，BookDB.java 不必直接建立和数据库的连接，只需从 Servlet 容器提供的数据源的数据库连接池中取出一个空闲状态的连接。BookDB 通过数据源访问数据库的过程如图 6-7 所示。

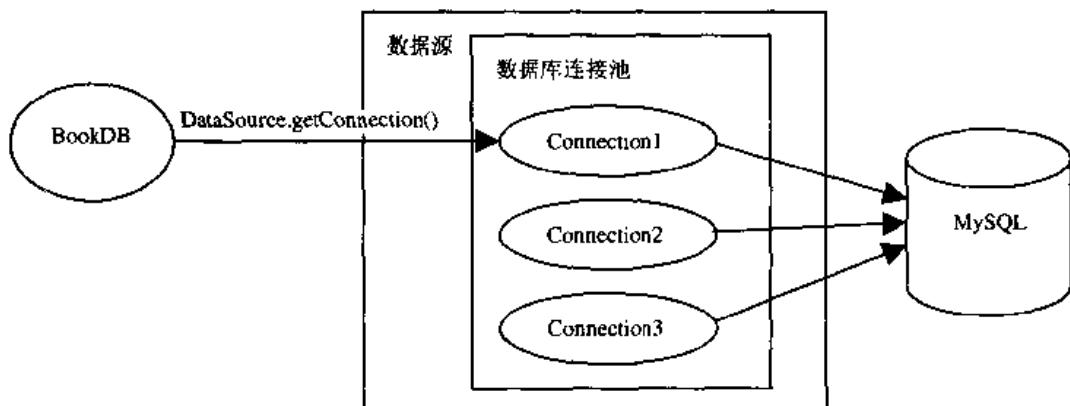


图 6-7 BookDB 通过数据源访问数据库

下面讲述 version1 中 BookDB 的实现，相关的代码位于本书配套光盘的 sourcecode/bookstores/version1/bookstore/src/mypack 目录下。

在 BookDB 的构造方法中通过 Context.lookup()方法获得 DataSource 的引用，并将它保存在成员变量 ds 中。

每次访问数据库时，都调用 BookDB 自身的 getConnection()方法，该方法从 DataSource 的数据库连接池中取出一个空闲状态的连接：

```

public Connection getConnection(){
    try{
        return ds.getConnection();
    }catch(Exception e){
        e.printStackTrace();
        return null;
    }
}

```

当数据库访问结束后，应该依次关闭 ResultSet、PreparedStatement(或 Statement)和 Connection 对象，从而使连接返回到数据库连接池。在 BookDB.java 中定义了 3 个方法分别关闭这 3 种对象：

```

closeResultSet(ResultSet rs)
closePrepStmt(PreparedStatement prepStmt)
closeConnection(Connection con)

```

为了确保数据库访问结束后，releaseConnection()方法一定被执行，所有访问数据库的方法采用如下结构：

```

Connection con=null;
PreparedStatement prepStmt=null;
ResultSet rs=null;
try {
    con=getConnection();
    //access database
    .....
}finally{

```

```

        closeResultSet(rs);
        closePrepStmt(prepStmt);
        closeConnection(con);
    }
}

```

例程 6-4 给出了 BookDB.java 的源代码。

例程 6-4 BookDB.java

---

```

package mypack;
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
import java.util.*;

public class BookDB {

    private ArrayList books;
    private DataSource ds=null;

    public BookDB () throws Exception{

        Context ctx = new InitialContext();
        if(ctx == null )
            throw new Exception("No Context");
        ds =(DataSource)ctx.lookup("java:comp/env/jdbc/BookDB");
    }

    public Connection getConnection()throws Exception{
        return ds.getConnection();
    }

    public void closeConnection(Connection con){
        try{
            if(con!=null) con.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public void closePrepStmt(PreparedStatement prepStmt){
        try{
            if(prepStmt!=null) prepStmt.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

```
        }

    }

    public void closeResultSet(ResultSet rs){
        try{
            if(rs!=null) rs.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public int getNumberOfBooks() throws Exception {
        Connection con=null;
        PreparedStatement prepStmt=null;
        ResultSet rs=null;
        books = new ArrayList();

        try {
            con=getConnection();
            String selectStatement = "select * " + "from books";
            prepStmt = con.prepareStatement(selectStatement);
            rs = prepStmt.executeQuery();

            while (rs.next()) {
                BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
                    rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
                books.add(bd);
            }
        } finally{
            closeResultSet(rs);
            closePrepStmt(prepStmt);
            closeConnection(con);
        }
        return books.size();
    }

    public Collection getBooks()throws Exception{
        Connection con=null;
        PreparedStatement prepStmt=null;
        ResultSet rs =null;
        books = new ArrayList();
        try {

```

```
con= getConnection();
String selectStatement = "select * " + "from books";
prepStmt = con.prepareStatement(selectStatement);
rs = prepStmt.executeQuery();

while (rs.next()) {

    BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
        rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
    books.add(bd);
}

}finally{
    closeResultSet(rs);
    closePrepStmt(prepStmt);
    closeConnection(con);
}

Collections.sort(books);
return books;
}

public BookDetails getBookDetails(String bookId) throws Exception {
    Connection con=null;
    PreparedStatement prepStmt=null;
    ResultSet rs =null;
    try {
        con= getConnection();
        String selectStatement = "select * " + "from books where id = ? ";
        prepStmt = con.prepareStatement(selectStatement);
        prepStmt.setString(1, bookId);
        rs = prepStmt.executeQuery();

        if (rs.next()) {
            BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
                rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
            prepStmt.close();

            return bd;
        }
        else {
            return null;
        }
    }
}
```

```
        }finally{
            closeResultSet(rs);
            closePrepStmt(prepStmt);
            closeConnection(con);
        }
    }

public void buyBooks(ShoppingCart cart) throws Exception {
    Connection con=null;
    Collection items = cart.getItems();
    Iterator i = items.iterator();
    try {
        con=getConnection();
        con.setAutoCommit(false);
        while (i.hasNext()) {
            ShoppingCartItem sci = (ShoppingCartItem)i.next();
            BookDetails bd = (BookDetails)sci.getItem();
            String id = bd.getBookId();
            int quantity = sci.getQuantity();
            buyBook(id, quantity,con);
        }
        con.commit();
        con.setAutoCommit(true);
    } catch (Exception ex) {
        con.rollback();
        throw ex;
    }finally{
        closeConnection(con);
    }
}

public void buyBook(String bookId, int quantity,Connection con) throws Exception {
    PreparedStatement prepStmt=null;
    ResultSet rs=null;
    try{
        String selectStatement = "select * " + "from books where id = ? ";
        prepStmt = con.prepareStatement(selectStatement);
        prepStmt.setString(1, bookId);
        rs = prepStmt.executeQuery();

        if (rs.next()) {
            prepStmt.close();
        }
    }
}
```

```
String updateStatement =  
        "update books set saleamount = saleamount + ? where id = ?";  
prepStmt = con.prepareStatement(updateStatement);  
prepStmt.setInt(1, quantity);  
prepStmt.setString(2, bookId);  
prepStmt.executeUpdate();  
prepStmt.close();  
}  
}  
}finally{  
    closeResultSet(rs);  
    closePrepStmt(prepStmt);  
}
```

在本书配套光盘的 `sourcecode/bookstores/version1/bookstore` 目录下提供了采用数据源访问数据库的 `bookstore` 应用，可以按以下步骤发布这个应用。

卷之三

- (1) 把整个 bookstore 应用拷贝到<CATALINA\_HOME>/webapps 目录下。
  - (2) 把 bookstore 目录下 server\_modify.xml 文件（包含数据源配置代码）内容拷贝到 server.xml 中。
  - (3) 把本书配套光盘的 lib 目录下的 mysqldriver.jar 文件拷贝到<CATALINA\_HOME>/common/lib 目录下。

## 6.6 处理中文编码

当通过 JDBC 或者是 DataSource 数据源(最终也是通过 JDBC), 从数据库中取出数据, 该数据的字符编码有可能和网页使用的编码不一致, 如果不进行相关的处理, 会导致在网页上出现乱码。例如, 在 DbJsp.jsp 网页中声明使用 GB2312 中文编码:

<%@ page contentType="text/html; charset=GB2312" %>

而本书使用的 MySQL 数据库的 JDBC 驱动程序(即本书配套光盘的 lib/mysql.jar)默认情况下采用 ISO-8859-1 编码。为了把从 MySQL 数据库中读出的数据正确地显示在网页上, 可以采用以下两种办法之一:

- 在设定连接数据库的 URL 时，指定字符编码。 DbJsp.jsp（参见例程 6-1）就采用了这种办法：

```
String dbUrl = "jdbc:mysql://localhost:3306/BookDB?useUnicode=true&characterEncoding=GB2312";
```

采用这种方式从数据库取出的数据，使用的字符编码为以上 dbUrl 中指定的 GB2312，这样就可以直接把它显示在网页上。

**提示**

对于不同的数据库，数据库 URL 中字符编码的设置形式可能不一样。

- 如果在设定连接数据库的 URL 时，没有设定字符编码，则首先应该知道 JDBC 驱动程序使用的默认字符编码，然后对从数据库中取出的数据进行字符编码转换。DbJsp1.jsp（参见例程 6-2）就采用了这种办法：

```
String col1 = rs.getString(1);
String col2 = rs.getString(2);
String col3 = rs.getString(3);
float col4 = rs.getFloat(4);
//convert character encoding
col1=new String(col1.getBytes("ISO-8859-1"),"GB2312");
col2=new String(col2.getBytes("ISO-8859-1"),"GB2312");
col3=new String(col3.getBytes("ISO-8859-1"),"GB2312");

//打印所显示的数据
out.println("<tr><td>" + col1 + "</td><td>" + col2 + "</td><td>" + col3 + "</td><td>" + col4 + "</td></tr>");
```

**提示**

不同数据库的驱动程序，以及同一数据库的不同版本的驱动程序，使用的默认字符编码都可能不一样。

## 6.7 小结

本章介绍了 Web 应用访问数据库的两种方法：一种方法是通过 JDBC API 访问数据库，还有一种方法是通过数据源访问数据库。这两种方法不同之处在于获得数据库连接的方式不一样。采用数据源，可以避免每次访问数据库都建立数据库连接，这样可以提高访问数据库的效率。在 java.sql 包中提供了访问数据库的 API，常用的类包括：Connection（代表数据库连接）、Statement（执行静态 SQL 语句）、PreparedStatement（执行动态 SQL 语句）和 ResultSet（代表 select 查询语句得到的记录集合）。

本章介绍了 bookstore 应用访问数据库的两种方法。bookstore version0 和 version1 中的 BookDB 的功能相同，代码和结构也很相似，它们的区别如下。

### 1. 构造方法的实现不一样

BookDB version0 的构造方法加载 MySQL 的 JDBC Driver：

```
public BookDB () throws Exception{
    Class.forName("com.mysql.jdbc.Driver");
}
```

BookDB version1 的构造方法获得 JNDI DataSource 的引用：

```
public BookDB () throws Exception{
    Context ctx = new InitialContext();
```

```
if(ctx == null )  
    throw new Exception("No Context");  
ds =(DataSource)ctx.lookup("java:comp/env/jdbc/BookDB");  
}
```

## 2. getConnection 方法的实现不一样

BookDB version0 的 getConnection 方法直接和数据库建立连接，然后返回连接对象：

```
public Connection getConnection()throws Exception{  
    return java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);  
}
```

BookDB version1 的 getConnection 方法从 DataSource 中获得空闲状态的连接：

```
public Connection getConnection()throws Exception{  
    return ds.getConnection();  
}
```

## 3. closeConnection 方法的作用不一样

两个 BookDB 的 closeConnection 方法的代码相同：

```
public void closeConnection(Connection con){  
    try{  
        if(con!=null) con.close();  
    }catch(Exception e){  
        e.printStackTrace();  
    }  
}
```

在 BookDB version0 中，con.close()方法将直接关闭和数据库的连接，而在 BookDB version1 中，con.close()方法仅仅是把数据库连接对象返回到数据库连接池中，使连接对象又恢复到空闲状态。

# 第 7 章 Session 的使用与管理

当客户访问 Web 应用时，在许多情况下，Web 服务器必须能够跟踪客户的状态。比如有若干客户各自以合法的账号登录到电子邮件系统，然后分别进行收信、写信和发信等一系列操作。在这过程中，假如某个客户请求查看收件箱，Web 服务器必须能判断出发出请求的客户的身份，这样才能返回与这个客户相对应的数据。

再比如，有许多客户在同一个购物网站上购物，Web 服务器为每个客户配置了虚拟的购物车（ShoppingCart）。当某个客户请求将一个商品放入购物车时，Web 服务器必须根据发出请求的客户的身份，找到该客户的购物车，然后把商品放入其中。

Web 服务器跟踪客户的状态通常有 4 种方法：

- 建立含有跟踪数据的隐藏表格字段
- 重写包含额外参数的 URL
- 使用持续的 Cookie
- 使用 Servlet API 中的 Session（会话）机制

本章将介绍如何通过 Session 来实现服务器对用户的状态的跟踪。本章将通过 JSP 例子讲解 Session 在 Web 应用中的使用方法，以及如何在 Tomcat 中管理 Session。

## 7.1 Session 简介

HTTP 是无状态的协议。例如，有多个客户同时访问 bookstore 应用的 catalog.jsp 组件，都把同一本书加入到他们各自的购物车，他们请求的 URL 地址相同：

`http://localhost:8080/bookstore/catalog.jsp?Add=205`

当 Servlet 容器接收到以上 HTTP 请求后，如何判断是哪个客户发出的请求，从而把这本书加入到与这个客户相对应的购物车中呢？

在 Java Servlet API 中引入 Session 机制来跟踪客户的状态。Session 指的是在一段时间内，单个客户与 Web 服务器的一连串相关的交互过程。在一个 Session 中，客户可能会多次请求访问同一个网页，也有可能请求访问各种不同的服务器资源。

例如在电子邮件应用中，从一个客户登录到电子邮件系统开始，经过收信、写信和发信等一系列操作，直至最后退出邮件系统，整个过程为一个 Session。再比如，在网上书店应用中，从一个客户开始购物，到最后结账，整个过程为一个 Session。

在 Servlet API 中定义了 `javax.servlet.http.HttpSession` 接口，Servlet 容器必须实现这一接口。当一个 Session 开始时，Servlet 容器将创建一个 HttpSession 对象，在 HttpSession 对象中可以存放客户状态的信息（例如购物车）。Servlet 容器为 HttpSession 分配一个唯一标识符，称为 Session ID。Servlet 容器把 Session ID 作为 Cookie 保存在客户的浏览器中。每次客户发出 HTTP 请求时，Servlet 容器可以从 HttpServletRequest 对象中读取 Session ID，然后根

据 Session ID 找到相应的 HttpSession 对象，从而获取客户的状态信息。Session 的运行机制如图 7-1 所示。

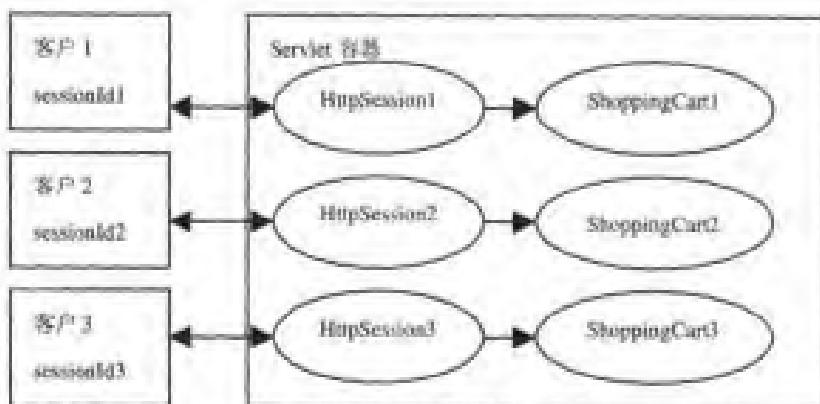


图 7-1 Session 的运行机制

图 7-1 中，每个客户都有惟一的 SessionID，它对应一个 HttpSession 对象，每个 HttpSession 对象都指向这个客户的 ShoppingCart 对象。



本书 4.3 节（JSP 与 Cookie）的 `jspCookie.jsp` 例子表明，Session ID 作为 Cookie 保存在客户的浏览器中。

`HttpSession` 接口中的方法描述参见表 7-1，这些方法提供了维护客户状态信息的功能。

表 7-1 HttpSession 接口

方 法	描 述
<code>getId()</code>	返回 Session 的 ID
<code>invalidate()</code>	使当前的 Session 失效，Servlet 容器会释放 HttpSession 对象占用的资源
<code>setAttribute(String name, Object value)</code>	将一对 name/value 属性保存在 HttpSession 对象中
<code>getAttribute(String name)</code>	根据 name 参数返回保存在 HttpSession 对象中的属性值
<code>getAttributeNames()</code>	以数组的方式返回 HttpSession 对象中所有的属性名
<code>isNew()</code>	判断是否是新创建的 Session。如果是新创建的 Session，返回 true，否则返回 false
<code>setMaxInactiveInterval()</code>	指定一个 Session 可以处于不活动状态的最大时间间隔，以秒为单位，如果超过这个时间，Session 自动失效。如果设置为负数，表示不限制 Session 处于不活动状态的时间
<code>getMaxInactiveInterval()</code>	读取当前 Session 可以处于不活动状态的最大时间间隔

默认情况下，JSP 网页都是支持 Session 的，也可以通过以下语句显示声明支持 Session：

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

当客户第一次访问 Web 应用中支持 Session 的某个网页时，就会开始一个新的 Session。接下来当客户浏览这个 Web 应用的不同网页时，始终处于同一个 Session 中。Session 拥有特定的生命周期。在以下情况中，Session 将结束生命周期，Servlet 容器会将 Session 所占

用的资源释放掉：

- 客户端关闭浏览器
- Session 过期
- 服务器端调用了 HttpSession 的 invalidate()方法

Session 过期是指当 Session 开始后，在一段时间内客户没有和 Web 服务器交互，这个 Session 会失效， HttpSession 类的 setMaxInactiveInterval()方法可以设置允许 Session 保持不活动状态的时间（以秒为单位），如果超过这一时间，Session 就会失效。

## 7.2 Session 范例程序

下面是一个简单的邮件系统，由 3 个 JSP 文件组成： maillogin.jsp（参见例程 7-1）、 mailcheck.jsp（参见例程 7-2）和 maillogout.jsp（参见例程 7-3）。它们位于本书配套光盘的 sourcecode/chapter7/mail1 目录下。

在 maillogin.jsp 提供了一个登录界面，要求用户输入用户名和口令，并且显示当前的 Session ID，参见例程 7-1。

例程 7-1 maillogin.jsp

---

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page session="true" %>
<html>
    <head>
        <title>helloapp</title>
    </head>

    <body bgcolor="#FFFFFF" onLoad="document.loginForm.username.focus()">

    <%
        String name="";
        if(!session.isNew()){
            name=(String)session.getAttribute("username");
            if(name==null)name="";
        }
    %>

    <p>欢迎光临邮件系统</p>
    <p>Session ID:<%=session.getId()%></p>
    <table width="500" border="0" cellspacing="0" cellpadding="0">
        <tr>
            <td>
                <table width="500" border="0" cellspacing="0" cellpadding="0">
                    <form name="loginForm" method="post" action="mailcheck.jsp">
                        <tr>
                            <td width="401"><div align="right">User Name:&nbsp;</div></td>
```

```
<td width="399"><input type="text" name="username"
value=<%=name%>></td>
</tr>
<tr>
<td width="401"><div align="right">Password:&nbsp;</div></td>
<td width="399"><input type="password" name="password"></td>
</tr>
<tr>
<td width="401">&nbsp;</td>
<td width="399"><br><input type="Submit" name="Submit" value="提交"></td>
</tr>
</form>
</table>
</td>
</tr>
</table>
</body>
</html>
```

mailcheck.jsp 从 HttpServletRequest 中读取用户账号，将用户名作为属性保存在 HttpSession 中，然后返回新邮件数目，在这里只是简单地返回一个固定的邮件数目。在 mailcheck.jsp 中还提供了到 maillogin.jsp 和 maillogout.jsp 的链接，参见例程 7-2。

例程 7-2 mailcheck.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page session="true" %>
<html>
<head><title>checkmail</title></head>
<body>

<%
String name=null;
name=request.getParameter("username");
if(name!=null)session.setAttribute("username",name);
%>

<a href="maillogin.jsp">登录</a>&nbsp;&nbsp;&nbsp;
<a href="maillogout.jsp">注销</a>&nbsp;&nbsp;&nbsp;
<p>当前用户为： <%=name%> </P>
<P>你的信箱中有 100 封邮件</P>

</body>
</html>
```

maillogout.jsp 调用 HttpSession 的 invalidate()方法结束当前的 Session，并且提供了到 maillogin.jsp 的链接，参见例程 7-3。

## 例程 7-3 maillogout.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ page session="true" %>
<html>
  <head>
    <title>maillogout</title>
  </head>
  <body>
    <%
      String name=(String)session.getAttribute("username");
      session.invalidate();
    %>

    <%=name%,再见！
    <p>
    <p>
      <a href="maillogin.jsp">重新登录邮件系统</a>&ampnbsp&ampnbsp&ampnbsp
    </body>
  </html>

```

假定把这 3 个 JSP 文件发布到 helloapp 应用中，按如下步骤来测试代码。



- (1) 访问 <http://localhost:8080/helloapp/maillogin.jsp>，输入用户名和口令，如图 7-2 所示。



图 7-2 maillogin.jsp 网页

- (2) 在 maillogin.jsp 中单击【提交】按钮，进入 mailcheck.jsp 页面，如图 7-3 所示。

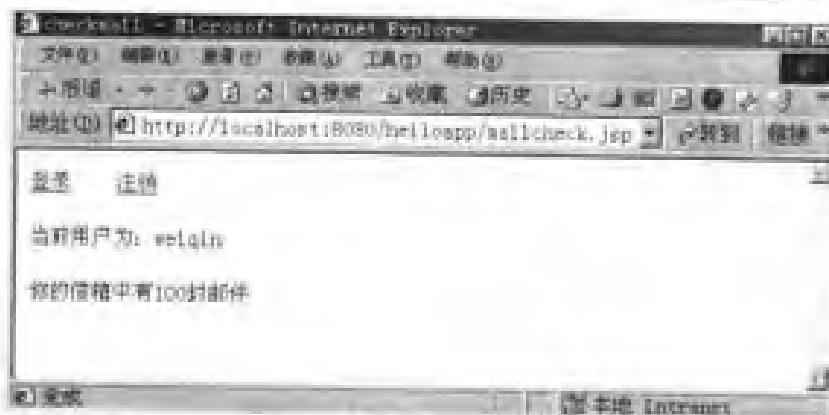


图 7-3 mailcheck.jsp 网页

(3) 在 mailcheck.jsp 页面中选中“登录”链接，再次进入 maillogin.jsp 页面。这时客户端仍然处于同一个 Session 中，调用 session.isNew()方法返回 false：

```
<%
String name="";
if(!session.isNew()){
    name=(String)session.getAttribute("username");
    if(name==null)name="";
}
%>
```

maillogin.jsp 接着调用 session.getAttribute("username")方法，获得当前 Session 中的用户信息。会看到 maillogin.jsp 将当前用户名显示在用户文本框中，而且显示的 Session ID 与第一次登录 maillogin.jsp 时显示的 ID 一样。

(4) 在 maillogin.jsp 中单击【提交】按钮，进入 mailcheck.jsp 页面，选择“注销”链接，进入 maillogout.jsp 网页，如图 7-4 所示。

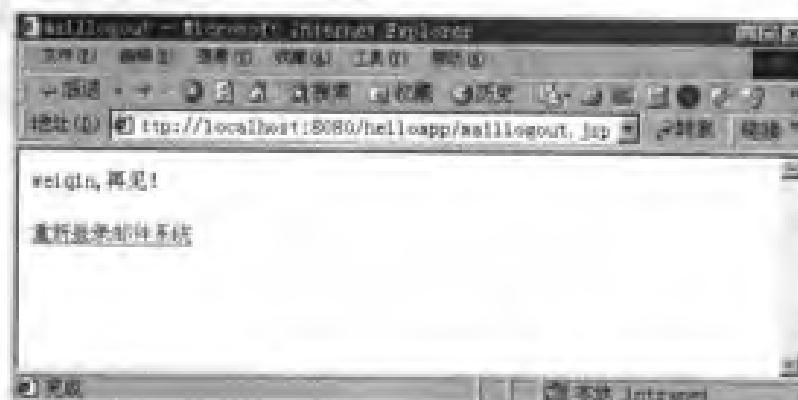


图 7-4 maillogout.jsp

(5) 在 maillogout.jsp 网页选择“重新登录邮件系统”链接，再次进入 maillogin.jsp 页面，这时由于在 maillogout.jsp 中已经调用 session.invalidate()方法，使上一个 Session 失效（与 Session 对应的 HttpSession 对象被销毁），因此再次进入 maillogin.jsp 页面时将开始一个新的 Session（Servlet 容器会创建与之对应的新的 HttpSession 对象）。在 maillogin.jsp 页面上会显示新的 Session ID。

(6) 打开两个浏览器，用不同的账号分别访问邮件系统，这时服务器将创建两个Session（每个Session有各自的 HttpSession 对象），如果两个浏览器同时访问相同的网页，则会看到在各个浏览器端显示的内容各自独立。

### 7.3 Session 的跟踪

通过本章前两节的介绍，我们已经知道 Servlet 容器通过在客户端浏览器中保存一个 Session ID 来跟踪 Session。如果客户浏览器支持 Cookie，就把 Session ID 作为 Cookie 保存在浏览器中，7.2 节介绍的邮件系统就是基于客户浏览器支持 Cookie 的。

下面开始在 IE 浏览器中禁止本地 Intranet 的 Cookie，在 IE 浏览器中选择【工具】→【Internet 选项】→【安全】→【本地 Intranet】→【自定义级别】选项，然后禁止 Cookie，如图 7-5 所示。



图 7-5 在 IE 浏览器中禁用 Cookie

接下来再访问 7.2 节的邮件系统，步骤如下。

#### 步骤

(1) 访问 <http://localhost:8080/helloapp/maillogin.jsp>，输入用户名和口令，参照 7.2 节图 7-2。

(2) 在 maillogin.jsp 中单击【提交】按钮，进入 mailcheck.jsp 页面，参照 7.2 节图 7-3。

(3) 在 mailcheck.jsp 页面中选择“登录”链接，再次访问 maillogin.jsp 页面。这时由于客户端浏览器中不存在作为 Cookie 的 Session ID，因此在客户请求中不包含 Cookie 信息，因为 Servlet 容器又创建了一个新的 Session，所以在 maillogin.jsp 页面上显示的是新的 Session ID，且用户文本框中的内容为空。

由此可以看出，如果客户浏览器禁止 Cookie，Servlet 容器无法从客户端浏览器中取得作为 Cookie 的 Session ID，也就无法跟踪客户状态，因此每次客户请求支持 Session 的 JSP 页面，Servlet 容器都会创建一个新的 Session，这样就失去了跟踪客户状态的功能。

在 Java Servlet API 中提出了跟踪 Session 的另一种机制，如果客户浏览器不支持 Cookie，Servlet 容器可以重写客户请求的 URL，把 Session ID 添加到 URL 信息中。HttpServletResponse 接口提供了重写 URL 的方法：

```
public java.lang.String encodeURL(java.lang.String url)
```

该方法的实现机制为：

- 先判断当前的 Web 组件是否启用 Session，如果没有启用 Session，例如在 JSP 中声明`<%@ page session="false" %>`，或者已经执行了`session.invalidate()`方法，那么直接返回参数 url
- 再判断客户浏览器是否支持 Cookie，如果支持 Cookie，就直接返回参数 url；如果不支持 Cookie，就在参数 url 中加入 Session ID 信息，然后返回修改后的 url

为了保证在浏览器不支持 Cookie 的情况下，也可以正常使用 Session，应该修改 7.2 节例子中所有的 URL 链接，例如把 mailcheck.jsp 中对 maillogin.jsp 的链接做如下修改：

修改前：

```
<a href="maillogin.jsp">
```

修改后：

```
<a href="<%="response.encodeURL("maillogin.jsp")%>">
```

例程 7-4、例程 7-5 和例程 7-6 分别是修改后的 maillogin.jsp、checkmail.jsp 和 maillogout.jsp 的代码。

例程 7-4 maillogin.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page session="true" %>
<html>
<head>
<title>helloapp</title>
</head>

<body bgcolor="#FFFFFF" onLoad="document.loginForm.username.focus()">

<%
String name="";
if(!session.isNew()){
    name=(String)session.getAttribute("username");
    if(name==null)name="";
}
%>
<p>欢迎光临邮件系统</p>
<p>Session ID:<%=session.getId()%></p>
<table width="500" border="0" cellspacing="0" cellpadding="0">
```

```

<tr>
  <td>
    <table width="500" border="0" cellspacing="0" cellpadding="0">
      <form name="loginForm" method="post"
        action="<%>response.encodeURL("mailcheck.jsp")%>">
        <tr>
          <td width="401"><div align="right">User Name:&nbsp;</div></td>
          <td width="399"><input type="text" name="username" value=<%>name%></td>
        </tr>
        <tr>
          <td width="401"><div align="right">Password:&nbsp;</div></td>
          <td width="399"><input type="password" name="password"></td>
        </tr>
        <tr>
          <td width="401">&nbsp;</td>
          <td width="399"><br><input type="Submit" name="Submit" value="提交"></td>
        </tr>
      </form>
    </table>
  </td>
</tr>
</table>

</body>
</html>

```

例程 7-5 checkmail.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ page session="true" %>
<html>
  <head><title>checkmail</title></head>
  <body>

    <%
    String name=null;
    name=request.getParameter("username");
    if(name!=null)session.setAttribute("username",name);
    %>

    <a href="<%>response.encodeURL("maillogin.jsp")%>">登录</a>&nbsp;&nbsp;&nbsp;
    <a href="<%>response.encodeURL("maillogout.jsp")%>">注销</a>&nbsp;&nbsp;&nbsp;
    <p>当前用户为: <%>name%> </P>
    <p>你的信箱中有 100 封邮件</P>

  </body>

```

```
</html>
```

### 例程 7-6 maillogout.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page session="true" %>
<html>
    <head>
        <title>maillogout</title>
    </head>
    <body>
        <%
            String name=(String)session.getAttribute("username");
            session.invalidate();
        %>
        <%=name%>,再见!
        <p>
        <p>
            <a href="<%="<%=response.encodeURL("maillogin.jsp")%>">">重新登录邮件系统</a>
        </body>
    </html>
```

修改后的源文件位于本书配套光盘的 sourcecode/chapter7/mail2 目录下，把它们发布到 helloapp 应用中，按如下步骤运行修改后的应用。

#### 步骤

(1) 访问 <http://localhost:8080/helloapp/maillogin.jsp>，服务器端的 maillogin.jsp 将执行以下代码：

```
<form name="loginForm" method="post"
      action="<%="<%=response.encodeURL("mailcheck.jsp")%>">">
```

以上代码的 response.encodeURL("mailcheck.jsp") 方法将返回包含 SessionID 的 URL 信息。可以选择 IE 浏览器的【查看】→【源文件】菜单，查看服务器传给客户端的源文件。此时可以看到以上代码生成的网页内容为：

```
<form name="loginForm" method="post"
      action="mailcheck.jsp;jsessionid=95493599415695F01771DB6F270F68EF">
```

(2) 在客户端 maillogin.jsp 网页上输入用户名和口令，参照 7.2 节图 7-2，然后单击【提交】按钮，此时客户请求的 URL 信息应该为以上包含 SessionID 的 URL，所以在访问 mailcheck.jsp 页面时，如图 7-6 所示，会发现在 IE 浏览器的地址一栏中的 URL 添加了 Session ID 信息：

<http://localhost:8080/helloapp/mailcheck.jsp;jsessionid=95493599415695F01771DB6F270F68EF>

(3) 在 mailcheck.jsp 页面中选中“登录”链接，再次进入 maillogin.jsp 页面。同样，Session ID 信息也添加到 URL 中，此时由于处于同一个 Session 中，在 maillogin.jsp 中显示当前 Session ID 和用户名。



图 7-6 mailcheck.jsp 网页

(4) 在 maillogin.jsp 中单击【提交】按钮，进入 mailcheck.jsp 页面，再选择“注销”链接，进入 maillogout.jsp 网页，然后在 maillogout.jsp 网页中选择“重新登录邮件系统”链接，再次进入 maillogin.jsp 页面。这时由于在 maillogout.jsp 中已经调用 session.invalidate() 方法，因此在 maillogout.jsp 中的 response.encodeURL 方法直接返回 maillogin.jsp：

```
session.invalidate();
```

```
.....
```

```
<a href="<% =response.encodeURL("maillogin.jsp") %>">>重新登录邮件系统</a>
```

所以从浏览器的地址栏可以看到此时访问 maillogin.jsp 的 URL 中没有添加 Session ID 信息。

## 7.4 Session 的持久化

当一个 Session 开始时，Servlet 容器会为 Session 创建一个 HttpSession 对象。Servlet 容器在某些情况下把这些 HttpSession 对象从内存中转移到文件系统或数据库中，在需要访问 HttpSession 信息时再把它们加载到内存中。这样做，有两个好处：

- 假定有一万个人同时在访问某个 Web 应用，Servlet 容器中会生成一万个 HttpSession 对象。如果把这些对象都一直存放在内存中，将消耗大量的内存资源，显然是不可取的。因此可以把处于不活动状态的 HttpSession 对象转移到文件系统或数据库中，这样可以提高对内存资源的利用率
- 假定某个客户正在一个购物网站上购物，他将购买的物品先放在虚拟的购物车（ShoppingCart）中，Servlet 容器将这个包含购物信息的购物车对象保存在 HttpSession 对象中。如果此时 Web 服务器忽然出现故障而终止，那么内存中的 HttpSession 对象连同客户的购物信息都会丢失。如果把 HttpSession 对象事先保存在文件系统或数据库中，当 Web 服务器重启后，还可以从文件系统或数据库中恢复 Session 数据

**提示** 把 HttpSession 对象保存到文件系统或数据库中的方法，采用了 Java 语言提供的对象序列化技术。如果把 HttpSession 对象从文件系统或数据库中恢复到内存中，则采用了 Java 语言提供的对象反序列化技术。

HttpSession 和 Session Store 的关系如图 7-7 所示。

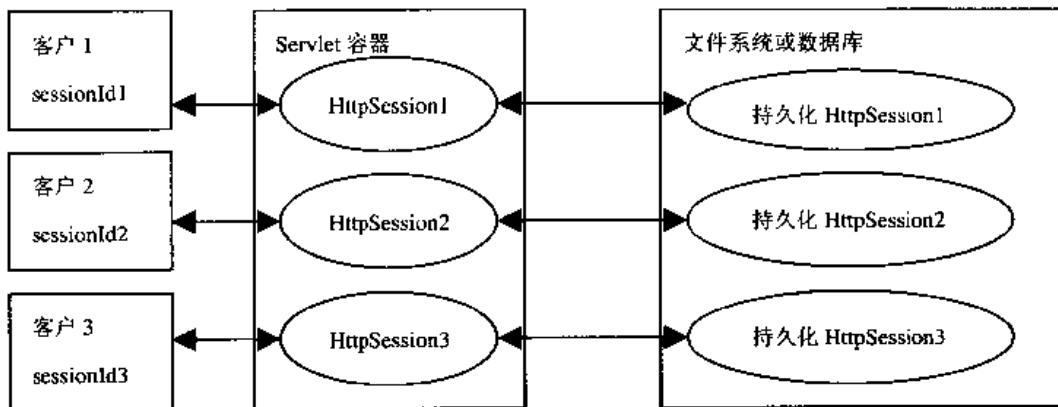


图 7-7 HttpSession 和 Session Store 的关系

Session 的持久化是由 Session Manager 来管理的。Tomcat 提供了两个实现类：

- org.apache.catalina.session.StandardManager
- org.apache.catalina.session.PersistentManager

### 1. StandardManager

StandardManager 是默认的 Session Manager。它的实现机制为：当 Tomcat 服务器关闭或重启，或者 Web 应用被重新加载时，会对在内存中的 HttpSession 对象进行持久化，把它们保存到文件系统中，默认的文件为：

<CATALINA\_HOME>/work/Catalina/hostname/applicationname/SESSIONS.ser

例如，对于以上邮件系统的例子，采用的就是默认的 StandardManager，如果在访问 mailcheck.jsp 时关闭 Tomcat 服务器，会在<CATALINA\_HOME>/work/Catalina/localhost/helloapp 目录下看到一个 SESSIONS.ser 文件，这个文件中保存了持久化的 HttpSession 对象的信息。

重启 Tomcat 服务器后，Tomcat 服务器把 SESSIONS.ser 中的持久化 HttpSession 对象加载到内存中，此时对客户端来说，依然处于同一个 Session 中。

### 2. PersistentManager

PersistentManager 能够把 Session 对象保存到 Session Store 中，它提供了比 StandardManager 更为灵活的 Session 管理功能，它具有以下功能：

- 当 Tomcat 服务器关闭或重启，或者 Web 应用被重新加载时，会对在内存中的 HttpSession 对象进行持久化，把它们保存到 Session Store 中
- 具有容错功能，及时把 Session 备份到 Session Store 中，当 Tomcat 服务器意外关闭后再重启时，可以从 Session Store 中恢复 Session 对象
- 可以灵活控制在内存中的 Session 数目，将部分 Session 转移到 Session Store 中

Tomcat 实现持久化 Session Store 的接口为 org.apache.Catalina.Store，目前提供了两个实现这一接口的类：org.apache.Catalina.FileStore 和 org.apache.Catalina.JDBCStore。

下面讨论如何配置 PersistentManager 以及两种持久化 Session Store。

### 7.4.1 配置 FileStore

FileStore 将 HttpSession 对象保存在一个文件中，这个文件的默认目录为 <CATALINA\_HOME>/work/Catalina/hostname/applicationname。每个 HttpSession 对象都会对应一个文件，它以 Session 的 ID 作为文件名，扩展名为 session。

假定为 helloapp 应用配置 FileStore，应该在 server.xml 文件中，helloapp 应用的<Context> 元素内，加入<Manager>元素：

```
<Context path="/helloapp" docBase="helloapp" debug="0"
reloadable="true">

    ...
    <Manager className="org.apache.catalina.session.PersistentManager" >
        debug=0;
        saveOnRestart="true";
        maxActiveSessions="-1";
        minIdleSwap="-1";
        maxIdleSwap="-1";
        maxIdleBackup="-1";
        <Store className="org.apache.catalina.session.FileStore" directory="mydir" />
    </Manager>

</Context>
```

在本书配套光盘的 sourcecode/chapter7/server\_modify\_filestore.xml 文件中提供了以上代码。<Manager>元素专门用于配置 Session Manager，它的<Store>子元素指定了实现持久化 Session Store 的类和存放 Session 文件的目录。<Manager>元素的属性说明参见表 7-2。

表 7-2 <Manager>元素的属性

方 法	描 述
className	指定 Session Manager 的类名
debug	设定 Session Manager 采用的跟踪级别
saveOnRestart	如果这个属性设为 true，表示当 Tomcat 服务器关闭时，所有的有效 HttpSession 对象都会保存到 Session Store 中。当 Tomcat 服务器重启时，会重新加载这些 Session 对象
maxActiveSessions	指定可以处于活动状态的 Session 的最大数目。如果超过这一数目，Tomcat 将把一些 Session 对象转移到 Session Store 中。如果这个属性为 -1，表示不限制可以处于活动状态的 Session 的数目
minIdleSwap	指定 Session 处于不活动状态的最短时间（以秒为单位）。超过这一时间，Tomcat 有可能把这个 Session 对象转移到 Session Store 中
maxIdleSwap	指定 Session 处于不活动状态的最长时间（以秒为单位）。超过这一时间，Tomcat 必须把这个 Session 对象转移到 Session Store 中
maxIdleBackup	指定 Session 处于不活动状态的最长时间（以秒为单位）。超过这一时间，Tomcat 将为这个 Session 对象在 Session Store 中进行备份。与 maxIdleSwap 不同，这个 Session 对象仍然存在于内存中

修改 server.xml 后，重启 Tomcat 服务器，然后访问 maillogin.jsp，再由 maillogin.jsp 进入 mailcheck.jsp，mailcheck.jsp 把用户名保存在 HttpSession 对象中。这时关闭 Tomcat 服务器，会在<CATALINA\_HOME>/work/Catalina/localhost/helloapp/mydir 目录下看到一个以当前 Session ID 命名的 Session 文件，例如：

2C442D3D212EF05588B63A4C7F7D4C37.session。

如果再重启 Tomcat 服务器，从刚才的 mailcheck.jsp 网页转到 maillogin.jsp，则会发现 maillogin.jsp 网页显示原来的 Session ID 和用户名，这是因为 Tomcat 服务器从 Session 文件中重新加载了原来的 HttpSession 对象。

#### 7.4.2 配置 JDBCStore

JDBCStore 将 HttpSession 对象保存在数据库的一张表中。Session 表中的字段描述参见表 7-3。

表 7-3 Session 表的结构

方 法	描 述
session_id	表示 Session ID
session_data	表示 Session 对象的序列化数据
app_name	表示 Session 所属 Web 应用的名字
session_valid	表示 Session 是否有效
session_inactive	表示 Session 可以处于不活动状态的最长时间
last_access	表示最近一次访问 Session 的时间

下面是在 MySQL 中创建 Session 表的步骤，假定表的名字为 tomcat\_sessions，这张表所在的数据库为 tomcatsessionDB。关于 MySQL 的用法，可以参考 6.1 节（安装和配置 MySQL 数据库）。在 6.1 节中，已经创建了一个 MySQL 账号：用户名为：dbuser，口令为：1234。下面的配置中会用到这一账号。

#### 步骤

(1) 在操作系统中打开 DOS 界面，转到<MYSQL\_HOME>/bin 目录。

在 DOS 命令行输入命令：mysql，进入 MySql 客户界面。

(2) 创建数据库 tomcatsessionDB，SQL 命令如下：

```
CREATE DATABASE tomcatsessionDB;
```

(3) 进入 tomcatsessionDB 数据库，SQL 命令如下：

```
use tomcatsessionDB;
```

(4) 在 tomcatsessionDB 数据库中创建 tomcat\_sessions 表，SQL 命令如下：

```
create table tomcat_sessions (
    session_id      varchar(100) not null primary key,
    valid_session   char(1) not null,
    max_inactive    int not null,
    last_access     bigint not null,
```

```

app_name      varchar(255),
session_data  mediumblob,
KEY kapp_name(app_name)
);

```

本书配套光盘的 sourcecode/chapter7/tomcat\_sessions.sql 文件为创建 tomcat\_sessions 表的 SQL 脚本。

tomcat\_sessions 表创建好以后, 为 helloapp 应用配置 JDBCStore, 在 server.xml 文件中, helloapp 应用对应的<Context>元素内, 加入以下<Manager>元素:

```

<Context path="/helloapp" docBase="helloapp" debug="0"
reloadable="true">
    .....
    <Manager className="org.apache.catalina.session.PersistentManager" >
        debug=99;
        saveOnRestart="true"
        maxActiveSessions="-1"
        minIdleSwap="-1"
        maxIdleSwap="-1"
        maxIdleBackup="-1"
        <Store className="org.apache.catalina.session.JDBCStore"
            driverName="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost/tomcatsessionDB?user=dbuser password=1234"
            sessionTable="tomcat_sessions"
            sessionIdCol="session_id"
            sessionDataCol="session_data"
            sessionValidCol="valid_session"
            sessionMaxInactiveCol="max_inactive"
            sessionLastAccessedCol="last_access"
            sessionAppCol="app_name"
            checkInterval="60"
            debug="99"
        />
    </Manager>
</Context>

```

在本书配套光盘的 sourcecode/chapter7/server\_modify\_jdbcstore.xml 文件中提供了以上配置代码。

为了确保 Tomcat 服务器能够访问 MySQL 数据库, 应该将 MySQL 的 JDBC 驱动程序 (本书配套光盘的 lib/mysqldriver.jar 文件) 拷贝到<CATALINA\_HOME>/common/lib 目录下。

<Store>子元素的属性描述参见表 7-4。

表 7-4 &lt;Store&gt;子元素的属性

方 法	描 述
className	指定 Session Store 的类名。
debug	设定 Session Store 采用的跟踪级别，可选的值范围是 0~99，在产品发布阶段，可以将 debug 设为 0，有助于提高服务器运行性能。
driverName	设定数据库驱动程序。
connectionURL	设定访问数据库的 URL。
sessionTable	设定存放 Session 对象的表。
sessionIdCol	设定在 Session 对象表中，表示 Session ID 的字段名。
sessionDataCol	设定在 Session 对象表中，表示 Session Data 的字段名。
sessionAppCol	设定在 Session 对象表中，表示 Session 所属的 Web 应用的名字的字段名。
sessionValidCol	设定在 Session 对象表中，表示 Session 是否有效的字段名。
sessionMaxInactiveCol	设定在 Session 对象表中，表示 Session 可以处于不活动状态的最长时间的字段名。
sessionLastAccessCol	设定在 Session 对象表中，表示最近一次访问 Session 的时间的字段名。
checkInterval	设定在 Session 对象表中，表示 Tomcat 定期检查 Session 状态的时间间隔的字段名。

server.xml 修改以后，重启 Tomcat 服务器，然后按以下步骤测试 JDBCStore。

### 步骤

(1) 通过浏览器访问 <http://localhost:8080/helloapp/maillogin.jsp>，在网页上显示当前 Session ID：5B26A887913492585F4AEF334BB98142。

(2) 让浏览器停留在 maillogin.jsp 网页上，然后在 Tomcat 的管理平台终止 helloapp 应用，终止 Web 应用的方法可以参考 10.3 节（Tomcat 的管理平台）；或者也可以执行 <CATALINA\_HOME>/bin/shutdown.bat 关闭 Tomcat 服务器。

(3) 运行 MySQL 的客户程序，输入如下 SQL 命令：

```
use tomcatsessionDB;
select session_id from tomcat_sessions;
```

会看到在 tomcat\_sessions 表中保存了一条 HttpSession 的记录，如图 7-8 所示。

```
F:\01\MySQL> use tomcatsessionDB;
Database changed
mysql> select session_id from tomcat_sessions;
+-----+
| session_id |
+-----+
| 5B26A887913492585F4AEF334BB98142 |
+-----+
1 row in set (0.00 sec)

mysql>
```

图 7-8 在 tomcat\_sessions 表中保存的当前 Session 记录

(4) 重新启动 helloapp 应用，然后刷新刚才的 maillogin.jsp 网页，会发现 Session ID 不变，该网页仍然处于原来的 Session 中。

## 7.5 小结

本章介绍了Session的概念，通过JSP例子讲解了如何将Session运用到Web应用中。然后讲解了如何配置Session Store。Tomcat采用Persistent Manager来管理持久化Session Store。Session Store目前有两种实现：FileStore和JDBCStore。FileStore将Session数据保存在文件系统中，JDBCStore将Session数据保存在数据库中。下一章将介绍如何在Java Web应用中访问JavaBean。

# 第8章 访问 JavaBean

本章介绍如何在 Java Web 应用中使用 JavaBean。首先介绍 JavaBean 的概念和创建方法，接着介绍 JSP 访问 JavaBean 的语法，然后通过例子来解释 JavaBean 在 Web 应用中的存在范围。最后讲解如何在 bookstore 应用中运用 JavaBean。

## 8.1 JavaBean 简介

JavaBean 是一种可重复使用、且跨平台的软件组件。JavaBean 可分为两种：一种是有用户界面（UI, User Interface）的 JavaBean；还有一种是没有用户界面，主要负责处理事务（如数据运算，操纵数据库）的 JavaBean。JSP 通常访问的是后一种 JavaBean。

JSP 与 JavaBean 搭配使用，有 3 个好处：

- 使得 HTML 与 Java 程序分离，这样便于维护代码。如果把所有的程序代码都写到 JSP 网页中，会使得代码繁杂，难以维护
- 可以降低开发 JSP 网页人员对 Java 编程能力的要求
- JSP 侧重于生成动态网页，事务处理由 JavaBean 来完成，这样可以充分利用 JavaBean 组件的可重用性特点，提高开发网站的效率

一个标准的 JavaBean 有以下几个特性：

- JavaBean 是一个公共的（public）类
- JavaBean 有一个不带参数的构造方法
- JavaBean 通过 getXXX 方法设置属性，通过 setXXX 方法获取属性



在 JavaBean 中除了可以定义 getXXX 方法和 setXXX 方法，也可以像普通 Java 类那样定义其他完成特定功能的方法。

以下是一个 JavaBean 的例子，名为 CounterBean。在 CounterBean 中定义了一个属性 count，还定义了访问这个属性的两个方法：getCount() 和 setCount()。

```
package mypack;
public class CounterBean{
    private int count=0;
    public CounterBean(){}
    public int getCount(){
        return count;
    }
    public void setCount(int count){
        this.count=count;
    }
}
```

假定把 CounterBean 类发布到 helloapp 应用中，它的存放位置是<CATALINA\_HOME>/webapps/helloapp/WEB-INF/classes/mypack/CounterBean.class。

## 8.2 JSP 访问 JavaBean 的语法

在 JSP 网页中，既可以通过程序代码来访问 JavaBean，也可以通过特定的 JSP 标签来访问 JavaBean。采用后一种方法，可以减少 JSP 网页中的程序代码，使它更接近于 HTML 页面。下面介绍访问 JavaBean 的 JSP 标签。

### 1. 导入 JavaBean 类

如果在 JSP 网页中访问 JavaBean，首先要通过<%@ page import>指令导入 JavaBean 类，例如：

```
<%@ page import="mypack.CounterBean" %>
```

### 2. 声明 JavaBean 对象

<jsp:useBean>标签用来声明 JavaBean 对象，例如：

```
<jsp:useBean id="myBean" class="mypack.CounterBean" scope="session" />
<jsp:useBean id="myBean_1" class="mypack.CounterBean" scope="session" />
```

上述代码声明了两个 JavaBean 对象：myBean 和 myBean\_1。<jsp:useBean>标签中 id 代表 JavaBean 对象的变量名，class 用来指定 JavaBean 的类名，scope 用来指定 JavaBean 对象的范围（参见 8.3 节）。如果在 scope 指定的范围内，该 JavaBean 对象不存在，则创建这个 JavaBean 对象，相当于执行以下 Java 语句：

```
CounterBean myBean=new CounterBean();
CounterBean myBean_1=new CounterBean();
```

如果在 scope 指定的范围内，该 JavaBean 对象已经存在，则<jsp:useBean>标签不会生成新的 JavaBean 对象，而是直接获得已经存在的 JavaBean 对象的引用。



在<jsp:useBean>标签中，指定 class 属性时，必须给出完整的 JavaBean 的类名（包括类所属的包的名字）。如果将以上的声明语句改为：

```
<jsp:useBean id="myBean" class=" CounterBean" scope="session" />
```

JSP 编译器会找不到 CounterBean 类，从而抛出 ClassNotFoundException。

### 3. 访问 JavaBean 属性

JSP 提供了访问 JavaBean 属性的标签，如果要将 JavaBean 的某个属性输出到网页上，可以用<jsp:getProperty>标签，例如：

```
<jsp:getProperty name="myBean" property="count" />
```

如果要给 JavaBean 的某个属性赋值，可以用<jsp:setProperty>标签，例如：

```
<jsp:setProperty name="myBean" property="count" value="0" />
```

### 8.3 JavaBean 的范围

在<jsp:useBean>标签中可以设置 JavaBean 的 scope 属性，scope 属性决定了 JavaBean 对象存在的范围。scope 的可选值包括 page、request、session 和 application。scope 的默认属性值为 page。下面用一个访问 CounterBean 的 JSP 例子（例程 8-1 Counter.jsp）来解释这 4 种 scope 属性值。假定把 Counter.jsp 发布到 helloapp 应用中。

在这个 JSP 例子中，首先声明了一个 CounterBean 对象 myBean：

```
<jsp:useBean id="myBean" scope="page" class="mypack.CounterBean" />
```

接着输出 myBean 的 count 属性值，并将 count 属性增 1：

```
Current count value is :<jsp:getProperty name="myBean" property="count" />
<jsp:setProperty name="myBean" property="count"
    value="<% =myBean.getCount() + 1 %>" />
```

接下来判断 myBean 对象存在于哪种 scope 中：

```
<%
CounterBean obj=null;
String scope=null;
obj=(CounterBean)request.getAttribute("myBean");
if(obj!=null)scope="request";

obj=(CounterBean)session.getAttribute("myBean");
if(obj!=null)scope="session";

obj=(CounterBean)application.getAttribute("myBean");
if(obj!=null)scope="application";

if(scope==null)scope="page";
%>
```

Counter.jsp 的源代码参见例程 8-1。

例程 8-1 Counter.jsp

---

```
<%@ page import="mypack.CounterBean" %>

<html>
<head>
<title>
Counter
</title>
</head>
<jsp:useBean id="myBean" scope="page" class="mypack.CounterBean" />
<body>
```

Current count value is :<jsp:getProperty name="myBean" property="count" />

```
<jsp:setProperty name="myBean" property="count"
    value="<%="<%=myBean.getCount()+1 %>" />

<%
CounterBean obj=null;
String scope=null;
obj=(CounterBean)request.getAttribute("myBean");
if(obj!=null)scope="request";

obj=(CounterBean)session.getAttribute("myBean");
if(obj!=null)scope="session";

obj=(CounterBean)application.getAttribute("myBean");
if(obj!=null)scope="application";

if(scope==null)scope="page";
%>

<p>scope=<%=scope%></p>
</body>
</html>
```

### 8.3.1 JavaBean 在 page 范围内

在这种情况下，客户每次请求访问 JSP 页面时，都会创建一个 JavaBean 对象。JavaBean 对象的有效范围是客户请求访问的当前 JSP 网页。JavaBean 对象在以下两种情况下都会结束生命期：

- 客户请求访问的当前 JSP 网页通过<forward>标记将请求转发到另一个文件
- 客户请求访问的当前 JSP 页面执行完毕并向客户端发回响应

在 Counter.jsp 中，myBean 对象的存在范围是 page：

```
<jsp:useBean id="myBean" scope="page" class="mypack.CounterBean" />
```

通过浏览器访问 <http://localhost:8080/helloapp/Counter.jsp>，将会看到 count 的值为 0。网页上的输出内容如下：

Current count value is :0

scope=page

多次刷新网页，count 的值始终为 0。这是因为在 page 范围内，每次访问 Counter.jsp，都会生成新的 CounterBean 对象，原有的 CounterBean 对象已经结束生命期。

### 8.3.2 JavaBean 在 request 范围内

在这种情况下，客户每次请求访问 JSP 页面时，都会创建新的 JavaBean 对象。JavaBean

对象的有效范围为：

- 客户请求访问的当前 JSP 网页
- 和当前 JSP 网页共享同一个客户请求的网页，即当前 JSP 网页中<%@ include>指令以及<forward>标记包含的其他 JSP 文件

当所有共享同一个客户请求的 JSP 页面执行完毕并向客户端发回响应时，JavaBean 对象结束生命周期。

JavaBean 对象作为属性保存在 HttpServletRequest 对象中，属性名为 JavaBean 的 id，属性值为 JavaBean 对象，因此也可以通过 HttpServletRequest.getAttribute()方法取得 JavaBean 对象，例如：

```
CounterBean obj=(CounterBean)request.getAttribute("myBean");
```

修改 Counter.jsp 中的<jsp:useBean>标签，将 scope 属性改为 request：

```
<jsp:useBean id="myBean" scope="request" class="mypack.CounterBean" />
```

通过浏览器访问 <http://localhost:8080/helloapp/Counter.jsp>，会看到网页上的输出内容如下：

```
Current count value is :0
```

```
scope=request
```

多次刷新网页，count 的值始终为 0。这是因为在 request 范围内，每次访问 Counter.jsp，都会生成新的 CounterBean 对象，原有的 CounterBean 对象已经结束生命周期。

### 8.3.3 JavaBean 在 session 范围内

在这种情况下，JavaBean 对象被创建后，它存在于整个 Session 的生存周期内（第 7 章对 HttpSession 进行了介绍），同一个 Session 中的 JSP 文件共享这个 JavaBean 对象。

JavaBean 对象作为属性保存在 HttpSession 对象中，属性名为 JavaBean 的 id，属性值为 JavaBean 对象。除了可以通过 JavaBean 的 id 直接引用 JavaBean 对象外，也可以通过 HttpSession.getAttribute()方法取得 JavaBean 对象，例如：

```
CounterBean obj=(CounterBean)session.getAttribute("myBean");
```

修改 Counter.jsp 中的<jsp:useBean>标签，将 scope 属性改为 session：

```
<jsp:useBean id="myBean" scope="session" class="mypack.CounterBean" />
```

通过浏览器访问 <http://localhost:8080/helloapp/Counter.jsp>，会看到网页上的输出内容如下：

```
Current count value is :0
```

```
scope=session
```

多次访问 Counter.jsp，会看到 count 的值不断递增。如果打开一个新的浏览器，count 的值又从零开始增长。这是因为在“session”范围内，CounterBean 对象存在于一个 Session 中。每打开一个浏览器，就会开始一个新的 Session。每个 Session 中拥有各自的 CounterBean 对象。

### 8.3.4 JavaBean 在 application 范围内

JavaBean 对象被创建后, 它存在于整个 Web 应用的生命周期内, Web 应用中的所有 JSP 文件都能共享同一个 JavaBean 对象。

JavaBean 对象作为属性保存在 application 对象中, 属性名为 JavaBean 的 id, 属性值为 JavaBean 对象, 除了可以通过 JavaBean 的 id 直接引用 JavaBean 对象外, 也可以通过 application.getAttribute()方法取得 JavaBean 对象, 例如:

```
CounterBean obj=(CounterBean)application.getAttribute("myBean");
```

修改 Counter.jsp 中的<jsp:useBean>标签, 将 scope 属性改为 application:

```
<jsp:useBean id="myBean" scope="application" class="mypack.CounterBean" />
```

通过浏览器访问 <http://localhost:8080/helloapp/Counter.jsp>, 会看到网页上的输出内容如下:

```
Current count value is :0
```

```
scope=application
```

多次访问 Counter.jsp, 会看到 count 的值不断递增。如果打开一个新的浏览器, count 的值在原有的基础上继续递增。这是因为在“application”范围内, CounterBean 对象存在于 helloapp 应用的整个生命期中。

## 8.4 在 bookstore 应用中访问 JavaBean

在 bookstore 应用中创建了两个 JavaBean: BookDB 和 ShoppingCart。本书 6.5.2 节(在 bookstore 应用中通过数据源访问数据库)介绍了 BookDB 的实现, 5.3.2 节(购物车的实现)介绍了 ShoppingCart 的实现。下面讲解 JSP 文件如何访问这两个 JavaBean。

### 8.4.1 访问 BookDB

BookDB 负责访问数据库, 处理查询 book 数据以及购书事务。在 common.jsp 中声明了 BookDB:

```
<jsp:useBean id="bookDB" scope="application" class="mypack.BookDB"/>
```

BookDB 的 scope 属性为 application, 这意味着整个 web 应用只有一个 BookDB 对象, 所有通过<% include>标记包含了 common.jsp 的其他 JSP 网页都可以访问这个 BookDB 对象。

例如在 bookdetails.jsp 中调用了 bookDB.getBook(bookId)方法, 根据 bookId 查找书的详细信息, 并把查询结果输出到网页上。

bookdetails.jsp 的源代码参见例程 8-2。

例程 8-2 bookdetails.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```

<%@ include file="common.jsp" %>
<%@ page import="java.util.*" %>

<html>
<head><title>TitleBookDescription</title></head>
<%@ include file="banner.jsp" %>
<br>
<%
    //Get the identifier of the book to display
    String bookId = request.getParameter("bookId");
    if(bookId==null)bookId="201";
    BookDetails book = bookDB.getBookDetails(bookId);
%>

<%
    if(book==null)
    {
%>
        <p>书号<%=bookId%>在数据库中不存在<p>
        <strong><a href="<%=request.getContextPath()%>/catalog.jsp">继续购物</a></strong>
<%
    return;
}
%>

<p>书名: <%=convert(book.getTitle())%></p>
作者: <em><%=convert(book.getName())%></em>&nbsp;&nbsp;
(<%=book.getYear()%>)<br>
<p>价格(元): <%=book.getPrice()%></p>
<p>销售数量: <%=book.getSaleAmount()%></p>
<p>评论: <%=convert(book.getDescription())%></p>

<p><strong><a href="<%=request.getContextPath()%>/catalog.jsp?Add=<%=bookId%>">
加入购物车
</a>&nbsp; &nbsp; &nbsp;
<a href="<%=request.getContextPath()%>/catalog.jsp">继续购物</a></p></strong>
</body>
</html>

```

#### 8.4.2 访问 ShoppingCart

ShoppingCart 代表虚拟的购物车, ShoppingCart 的作用和实际生活中的购物车很相似。例如: 顾客们到超市里去购物, 每位顾客都有各自的购物车(相当于服务器为每个客户创建不同的 ShoppingCart 对象), 当某个顾客的整个购物活动结束时, 超市就会收回购物车(相

当于服务器清除这个客户的 ShoppingCart 对象)。当这个客户下次到超市里去购物,他又会找到一个空的购物车进行购物活动(相当于服务器为这个客户创建新的 ShoppingCart 对象)。

在 catalog.jsp、showcart.jsp 和 cashier.jsp 中均访问 ShoppingCart 对象。以下是声明 ShoppingCart 的代码:

```
<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart"/>
```

ShoppingCart 的范围为 session。bookstore 应用中的 Session 代表了客户的一次购物活动,从选购书开始,到付账结束。ShoppingCart 对象保存在 Session 中,用来跟踪客户的购书信息(例如书名和购买数量)。当客户付账时,服务器端就可以根据 ShoppingCart 中的信息来计算客户应支付的金额。

### 1. catalog.jsp 访问 ShoppingCart

在 catalog.jsp 网页上,如果客户针对某本书选择“加入购物车”链接,catalog.jsp 就会把这本书的信息加入到和该客户 Session 对应的 ShoppingCart 中:

```
<%
    // Additions to the shopping cart
    String bookId = request.getParameter("Add");
    if (bookId != null) {
        BookDetails book = bookDB.getBookDetails(bookId);
        cart.add(bookId, book);
    }
%>
```

### 2. showcart.jsp 访问 ShoppingCart

showcart.jsp 用于显示客户购物车中的内容,并且提供了管理购物车的功能。showcart.jsp 从 ShoppingCart 对象中读取所有的 ShoppingCartItem 对象,然后从 ShoppingCartItem 对象中读取 BookDetails 对象,并且将这些数据输出到网页上。

如果客户在 showcart.jsp 网页选择“删除”链接,就会执行如下代码:

```
<%
    String bookId = request.getParameter("Remove");
    if (bookId != null) {
        cart.remove(bookId);
        BookDetails book = bookDB.getBookDetails(bookId);
    }
%>
```

showcart.jsp 的源代码参见例程 8-3。

例程 8-3 showcart.jsp

---

```
<%@ page contentType="text/html; charset=GB2312" %>

<%@ include file="common.jsp" %>
<%@ page import="java.util.*" %>

<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart"/>

<html>
```

```

<head><title>TitleShoppingCart</title></head>
<%@ include file="banner.jsp" %>
<%
    String bookId = request.getParameter("Remove");
    if (bookId != null) {
        cart.remove(bookId);
        BookDetails book = bookDB.getBookDetails(bookId);
    }
%>

<font color="red" size="+2">您删除了一本书: <em><%=convert(book.getTitle())%>
</em>
<br>&nbsp;<br>
</font>

<%
}

if (request.getParameter("Clear") != null) {
    cart.clear();
}
%>

<font color="red" size="+2"><strong>
清空购物车
</strong><br>&nbsp;<br></font>

<%
}
// Print a summary of the shopping cart
int num = cart.getNumberOfItems();
if (num > 0) {
%>

<font size="+2">您的购物车内有<%=num%>本书
</font><br>&nbsp;

<table>
<tr>
<th align=left>数量</th>
<th align=left>书名</th>
<th align=left>价格</th>
</tr>

<%
Iterator i = cart.getItems().iterator();
while (i.hasNext()) {
    ShoppingCartItem item = (ShoppingCartItem)i.next();

```

```
BookDetails book = (BookDetails)item.getItem();
%>

<tr>
<td align="right" bgcolor="#ffffff">
<%=item.getQuantity()%>
</td>

<td bgcolor="#ffffaa">
<strong><a href="<%=request.getContextPath()%>/bookdetails.jsp?bookId=<%=book.getBookId()%>">
<%=convert(book.getTitle())%></a></strong>
</td>

<td bgcolor="#ffffaa" align="right">
<%=book.getPrice()%>
</td>

<td bgcolor="#ffffaa">
<strong>
<a href="<%=request.getContextPath()%>/showcart.jsp?Remove=<%=book.getBookId()%>">删除
</a></strong>
</td></tr>

<%
// End of while
%>

<tr><td colspan="5" bgcolor="#ffffff"><br></td></tr>

<tr>
<td colspan="2" align="right" bgcolor="#ffffff">总额(元)</td>
<td bgcolor="#ffffaa" align="right"><%=cart.getTotal()%></td>
<td><br></td>
</td>

</table>

<p>&nbsp;<p>
<strong><a href="<%=request.getContextPath()%>/catalog.jsp">继续购物
</a>&nbsp;&nbsp;&nbsp;
<a href="<%=request.getContextPath()%>/cashier.jsp">付账</a>&nbsp;&nbsp;&nbsp;
<a href="<%=request.getContextPath()%>/showcart.jsp?Clear=clear">清空购物车</a></strong>
<%
} else {
```

```

%>

<font size="+2">您的购物车目前为空</font>
<br>&nbsp;<br>
<a href="<%=request.getContextPath()%>/catalog.jsp">继续购物</a>

<%
// End of if
}
%>

</body>
</html>

```

### 3. cashier.jsp 访问 ShoppingCart

cashier.jsp 中从 ShoppingCart 中获取客户购买书的总数量和总金额，然后输出到网页上，此外还提供了让客户输入信用卡账号的表单。

```

<p>您一共购买了<%=cart.getNumberOfItems() %>本书</p>
<p>您应支付的金额为<%=cart.getTotal() %>元</p>

```

cashier.jsp 的源代码参见例程 8-4。

例程 8-4 cashier.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>

<%@ include file="common.jsp" %>
<%@ page import="java.util.*" %>

<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart"/>

<html>
<head><title>TitleCashier</title></head>
<%@ include file="banner.jsp" %>
<p>您一共购买了<%=cart.getNumberOfItems() %>本书</p>
<p>您应支付的金额为<%=cart.getTotal() %>元</p>

<form action="<%=request.getContextPath()%>/receipt.jsp" method="post">
<table>
<tr>
<td><strong>信用卡用户名</strong></td>
<td><input type="text" name="cardname" value="guest" size="19"></td>
</tr>
<tr>
<td><strong>信用卡账号</strong></td>
<td><input type="text" name="cardnum" value="xxxx xxxx xxxx xxxx" size="19"></td>
</tr>
<tr>

```

```
<td></td>
<td><input type="submit" value="递交"></td>
</tr>
</table>
</form>
</body>
</html>
```

## 8.5 小结

JavaBean 是一种可重复使用、跨平台的软件组件。JavaBean 在 JSP 网页中有 4 种范围：page、request、session 和 application。本章通过例子解释了 scope 属性和 JavaBean 对象的生命周期的关系。最后讲解了 JavaBean 在 bookstore 中的运用，并在 bookstore 应用中创建了两个 JavaBean：BookDB 和 ShoppingCart。BookDB 的 scope 范围为 application，ShoppingCart 的 scope 范围为 session。

# 第 9 章 用 ant 工具管理 Web 应用

ant 工具是 Apache 的一个开源代码项目，它是一个优秀的软件工程管理工具。ant 类似于 make 工具，但克服了传统的 make 工具的缺点。传统的 make 往往只能在某一平台上使用，ant 本身用 Java 语言实现，并且使用 XML 格式的配置文件来构建工程，可以很方便地实现多平台编译，非常适合管理大型工程。

本章介绍了 ant 的安装和配置，并以 bookstore 应用为例，介绍了 ant 的使用方法。

## 9.1 安装配置 ant

ant 的下载地址为 <http://jakarta.apache.org/builds>，本书配套光盘的 software 目录下提供了 apache-ant-1.5.4-bin.zip 文件。获得了 ant 的压缩文件后，应该把它解压到本地硬盘。假设解压后 ant 的根目录为<ANT\_HOME>。

接下来需要在操作系统中设置如下环境变量：

- ANT\_HOME - ant 的安装目录
- JAVA\_HOME - JDK 的安装目录
- PATH - 把%ANT\_HOME%/bin 目录添加到 PATH 变量中，以便于从命令行下直接运行 ant

上述设置完成后，就可以使用 ant 了。

## 9.2 创建 build.xml 文件

用 ant 编译规模较大的工程非常方便，每个工程都对应一个 build.xml 文件，这个文件包含与这个工程有关的路径信息和任务，每个 build.xml 文件都包含一个 project 和至少一个 target 元素。target 元素中包含一个或多个任务元素，任务是一段可执行代码。ant 提供了内置任务集，用户也可以开发自己的任务元素。最常用的构建工程的 ant 内置任务描述参见表 9-1。

表 9-1 ant 内置任务

ant 任务	功能
property	设置 name/value 形式的属性
mkdir	创建目录
copy	拷贝文件和文件夹
delete	删除文件或文件夹
javac	编译 Java 源文件
war	为 Web 应用打包

例如，为 bookstore 应用创建一个 build.xml 文件，用于编译 bookstore 应用的 Java 源代码，并且将这个应用打包为 WAR 文件。假定 bookstore 应用的根目录为〈bookstore〉，参见本书配套光盘的 sourcecode/bookstores/version0/bookstore，在这个目录下已经包含如图 9-1 所示的文件目录结构。



图 9-1 bookstore 应用原有的文件目录展开图



为了便于读者直接运行 bookstore 应用，在 WEB-INF 子目录下已经提供了包含所有类文件的 classes 目录。进行本章实验时，可以先删除这个目录。

build.xml 中配置的任务负责在 bookstore 的根目录下建立 build 子目录，然后在 build 子目录下创建 web 应用。最后创建的 build 目录结构如图 9-2 所示。

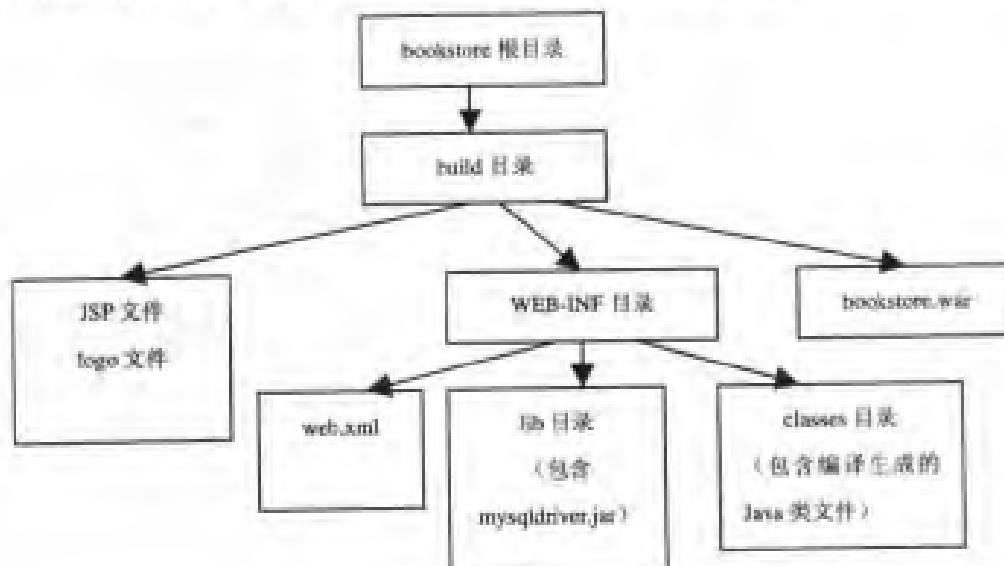


图 9-2 build.xml 配置的 build 目录结构

build 目录在 Windows 资源管理器中的展开图如图 9-3 所示。



图 9-3 build 目录在 Windows 资源管理器中的展开图

在 bookstore 根目录下提供了 build.xml 文件。例程 9-1 是 build.xml 的代码。

#### 例程 9-1 build.xml 文件

---

```

<project name="bookstore" default="about" basedir=".">
    <target name="init">
        <timestamp/>
        <property name="build" value="build" />
        <property name="src" value="src" />
        <property environment="myenv" />
        <property name="servletpath"
            value="${myenv.CATALINA_HOME}/common/lib/servlet-api.jar" />
        <property name="mysqlpath" value="WEB-INF/lib/mysqldriver.jar" />

        <mkdir dir="${build}" />
        <mkdir dir="${build}\WEB-INF" />
        <mkdir dir="${build}\WEB-INF\classes" />

        <copy todir="${build}" >
            <fileset dir="${basedir}" >
                <include name="*.jsp" />
                <include name="*.bmp" />
                <include name="WEB-INF/**" />
                <exclude name="build.xml" />
            </fileset>
        </copy>
    

```

---

```
</target>

<target name="compile" depends="init">
    <javac srcdir="${src}"
        destdir="${build}/WEB-INF/classes"
        classpath="${servletpath}:${mysqlpath}">
    </javac>
</target>

<target name="bookstorewar" depends="compile">
    <war warfile="${build}/bookstore.war" webxml="${build}/WEB-INF/web.xml">
        <lib dir="${build}/WEB-INF/lib"/>
        <classes dir="${build}/WEB-INF/classes"/>
        <fileset dir="${build}" />
    </war>
</target>

<target name="about" >
    <echo>
        This build.xml file contains targets for building bookstore web
        application
    </echo>
</target>

</project>
```

build.xml 的根元素是 project，它有 3 个属性：name、default 和 basedir。name 属性指定工程的名字，basedir 属性指定工程的基路径，如果设置为“.”，就表示工程的基路径为 build.xml 文件所在的路径。default 属性是必须给定的属性，它指定工程默认的 target 元素，运行 ant 时如果不指定 target，则使用 default 属性指定的 target。在本例中，默认的 target 为“about”：

```
<project name="bookstore" default="about" basedir=".">>
```

在本例中一共定义了 4 个 target：

```
<target name="init">
<target name="compile" depends="init">
<target name="bookstorewar" depends="compile">
<target name="about">
```

target 中的属性 depends 指定在执行本 target 之前必须完成的 target，例如 bookstorewar target 的 depends 属性为 compile，意思是在执行 bookstorewar target 之前，必须先执行 compile target。

### 1. init target

init target 完成初始化工作，它首先通过 property 任务来设置属性，一个工程可以设置很多属性，属性由名字和值构成：

```
<property name="build" value="build" />
```

```

<property name="src" value="src" />
<property environment="myenv" />
<property name="servletpath"
  value="${myenv.CATALINA_HOME}/common/lib/servlet-api.jar" />
<property name="mysqlpath" value="WEB-INF/lib/mysqldriver.jar" />

```

以上代码还设置了一个系统环境属性 myenv，通过它可以访问系统环境变量，例如，\${myenv.CATALINA\_HOME}代表了 CATALINA\_HOME 系统环境变量。

在 build.xml 文件的其他地方使用属性时的格式为 \${属性名}。

例如：

```
classpath="${servletpath}:${mysqlpath}"
```

对于以上代码，当 ant 运行时，会把属性名 servletpath 和 mysqlpath 对应的属性值替换到 classpath 的具体内容中。

init target 接下来通过 mkdir 任务创建 web 应用的目录结构：

```

<mkdir dir="${build}" />
<mkdir dir="${build}\WEB-INF" />
<mkdir dir="${build}\WEB-INF\classes" />

```

mkdir 任务的 dir 属性指定需要创建的目录，既可以指定绝对路径，也可以指定相对路径。如果路径内容以 “/”、“\” 或 “C:\” 之类开始，就表示绝对路径，否则表示相对路径。相对路径的基路径取决于 project 元素的 basedir 属性。

然后 init target 通过 copy 任务将 bookstore 基路径下相关的文件和目录拷贝到 build 目录下：

```

<copy todir="${build}" >
  <fileset dir="${basedir}" >
    <include name="*.jsp" />
    <include name="*.bmp" />
    <include name="WEB-INF/**" />
    <exclude name="build.xml" />
  </fileset>
</copy>

```

<copy>元素的 todir 属性指定把文件拷贝到哪个目录，<fileset>子元素的 dir 属性指定从哪个目录拷贝文件。<include>子元素指定需要拷贝哪些文件，<exclude>子元素指定不需要拷贝哪些文件。对于<include name="WEB-INF/\*\*" />，表示需要拷贝 WEB-INF 目录下所有的文件、子目录及子目录下的文件；如果是<include name="WEB-INF/\*.\*" />，表示只需要拷贝 WEB-INF 目录下所有的文件，不包含子目录以及子目录下的文件。

## 2. compile target

compile target 用来编译 Java 源程序：

```

<target name="compile" depends="init">
  <javac srcdir="${src}"
    destdir="${build}\WEB-INF\classes"
    classpath="${servletpath}:${mysqlpath}">
  </javac>
</target>

```

ant 的 javac 任务可以编译 Java 源程序, java 源文件放在 srcdir 属性指定的文件夹中, 生成的 CLASS 文件, 存放在 destdir 指定的文件夹中, 其目录结构与 package 语句一致。必须确保源文件的目录结构也与 package 语句相一致。

### 3. bookstorewar target

bookstorewar target 通过 war 任务为 bookstore 应用打包。

```
<target name="bookstorewar" depends="compile">
    <war warfile="${build}/bookstore.war" webxml="${build}/WEB-INF/web.xml">
        <lib dir="${build}/WEB-INF/lib"/>
        <classes dir="${build}/WEB-INF/classes"/>
        <fileset dir="${build}" />
    </war>
</target>
```

war 任务的 warfile 属性指定生成的 WAR 文件, webxml 属性指定 Web 应用的 web.xml 文件。<fileset dir="\${build}" /> 指定把 build 目录下所有的文件加入到包中。

### 4. about target

about target 中包含一个 echo 任务, 它的作用与 DOS 的 echo 命令相似, 用于向控制台回显文本。

```
<target name="about">
    <echo>
        This build.xml file contains targets for building bookstore web
        application
    </echo>
</target>
```

## 9.3 运行 ant

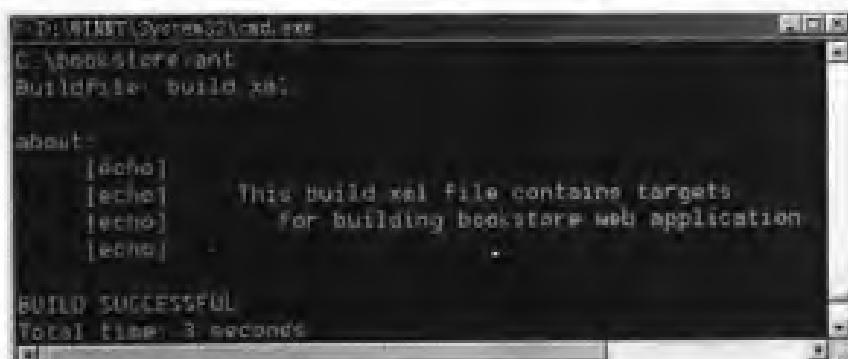
运行<ANT\_HOME>/bin/ant.bat 脚本, 可以直接运行 ant。如果不带任何参数, ant 会在当前路径下搜索 build.xml 文件, 如果找到, 就运行 project 元素的 default 属性指定的 target。在运行 ant 时, 也可以通过参数来指定 build.xml 文件和 target, 语法如下:

```
ant -buildfile <build-dir>/build.xml targetname
```

对于上面的例子, 假定已经把本书光盘 sourcecode/bookstores/version0/bookstore 目录拷贝到本地硬盘 C:\, 则在 DOS 命令行下用下面 3 种方式运行 ant 的效果是一样的:

- 先进入 C:\bookstore 目录, 再输入命令: ant
- 直接输入命令: ant -buildfile C:\bookstore\build.xml
- 直接输入命令: ant -buildfile C:\bookstore\build.xml about

build.xml 的默认 target 为 about, 对于以上 3 种方式都执行 about target, 在 DOS 界面上看到的输出结果如图 9-4 所示。



```

D:\WINNT\System32\cmd.exe
C:\bookstore\ant
buildfile: build.xml

about:
[echo]
[echo]      This build file contains targets
[echo]      For building bookstore web application
[echo]

BUILD SUCCESSFUL
Total time: 3 seconds

```

图 9.4 ant 执行 about target 的显示结果

如果要运行 bookstorewar target，可以采用以下两种方式：

- 先进入 C:\bookstore，再输入命令：ant bookstorewar
- 直接输入命令：ant -buildfile C:\bookstore\build.xml bookstorewar

以上两种方式都将执行 bookstorewar target。运行完毕，在 C:\bookstore\build 目录下将生成 bookstore.war 文件。如果要发布 bookstore 应用，只要把 bookstore.war 文件拷贝到 <CATALINA\_HOME>/webapps 目录下即可。

## 9.4 小结

ant 工具是 Apache 的一个开放源代码项目，它是一个优秀的软件工程管理工具。默认情况下，ant 运行时，在当前路径下搜索 build.xml 文件，如果找到，就运行 project 元素的 default 属性指定的 target。运行 ant 时，也可以通过参数来指定 build.xml 文件和 target。build.xml 文件包含与工程有关的路径信息和任务。在本章的例子中，介绍了 ant 提供的内置任务如 mkdir（创建目录）、javac（编译 Java 源程序）和 war（给 Web 应用打包）等的用法。

# 第 10 章 Tomcat 的控制平台和管理平台

在前面的章节已经讲过，可以通过 server.xml 文件来配置 Tomcat 服务器，通过 web.xml 文件来配置 web 应用。但这种配置方法不直观，比较复杂，对操作人员要求较高。为了简化配置的步骤，Tomcat 提供了基于 Web 方式的管理和控制平台，通过浏览器，用户可以很方便地配置 Tomcat 服务器，还可以管理运行在 Tomcat 服务器上的 Web 应用，如发布、启动、停止或删除 Web 应用，以及查看 Web 应用状态。

## 10.1 访问 Tomcat 的控制平台和管理平台

Tomcat 的控制平台和管理平台是 Tomcat 自带的两个 Web 应用，这两个 Web 应用分别位于以下目录中：

- <CATALINA\_HOME>/server/webapps/admin
- <CATALINA\_HOME>/server/webapps/manager

访问 Tomcat 的 admin 应用的 URL 为 <http://localhost:8080/admin>，访问 manager 应用的 URL 为 <http://localhost:8080/manager/html/>，访问这两个应用的登录页面分别如图 10-1 和图 10-2 所示。



图 10-1 admin 应用的登录页面

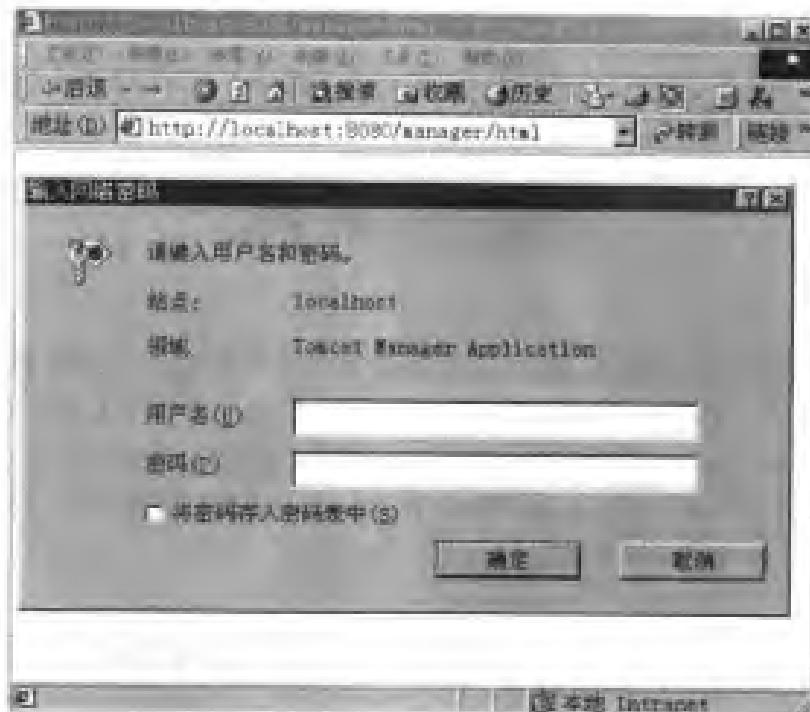


图 10-2 manager 应用的登录页面

安装好 Tomcat 后，并没有提供访问控制和管理应用的账号。因此，应该先添加具有控制和管理权限的账号。方法为打开<CATALINA\_HOME>/conf/tomcat-user.xml 文件，在该文件中添加如下内容：

```
<user username="admin" password="1234" roles="admin"/>
<user username="manager" password="1234" roles="manager"/>
```

上述代码分别创建了两个用户 admin 和 manager。admin 用户具有控制权限，manager 用户具有管理权限。这样，可以以 admin 用户身份登录 admin 应用，以 manager 用户身份登录 manager 应用。在本书第 11 章（安全域）中对如何配置安全验证信息进行了详细介绍。tomcat-user.xml 文件修改后，应该重启 Tomcat 服务器，文件修改才能生效。



如果在 Windows NT/2000 中直接通过 Tomcat 的安装程序（例如 jakarta-tomcat-5.0.12.exe）安装 Tomcat，在安装过程中会提示设定控制账号。

## 10.2 Tomcat 的控制平台

在 Tomcat 控制平台的登录窗口中输入用户名：admin，口令：1234，将登录控制平台，显示如图 10-3 所示的页面。

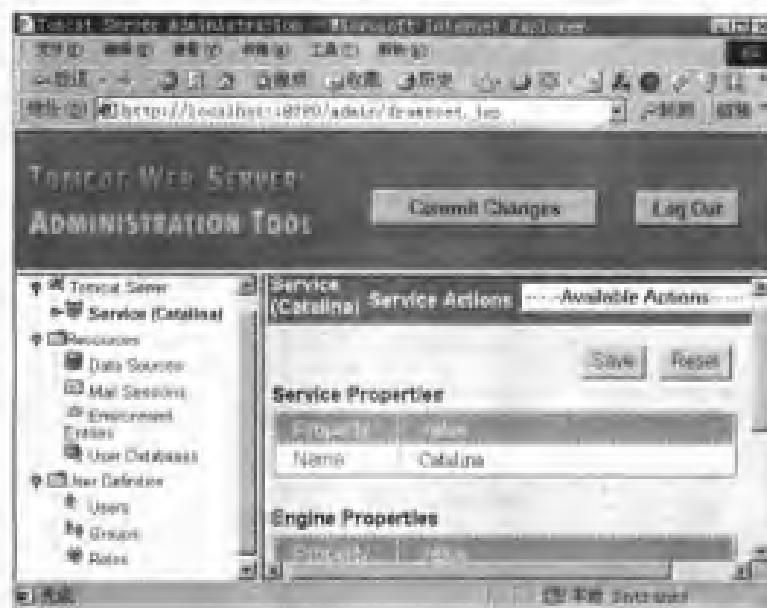


图 10-3 admin 应用主页

### 10.2.1 Tomcat 控制平台的功能

通过 Tomcat 控制平台，可以配置 Tomcat 服务器的各种元素。Tomcat 控制平台把所有可配置的信息分为 3 个目录：

- Tomcat Server 目录
- Resources 目录
- User Definition 目录

#### 1. Tomcat Server 目录

Tomcat Server 相当于 server.xml 中的<Server>元素，它下面包括许多子元素，这些子元素之间的嵌套关系和本书 1.2 节中介绍的各种组件的关系是一致的。

如果选中某个元素，在右边的窗口就会显示这个元素的所有属性，并且这些属性是可编辑的。例如，如果从左边的目录树中选择【Server】→【Service】→【Host】→【Context/bookstore】→【Resources】→【Data Source】，在右边的窗口中会显示在 bookstore 应用中配置的数据源 jdbc/BookDB，选中这个数据源，就会显示它的所有属性，如图 10-4 所示。

#### 2. Resources 目录

Resources 用于配置 Tomcat 的各种资源，在与 Tomcat Server 目录平级的 Resources 目录下配置的资源，可以被 Tomcat Server 中所有的 Web 应用访问，等价于在 server.xml 的<GlobalNamingResources>元素下配置资源。一共有 4 种资源：

- Data Source：代表 JNDI 数据源，参见第 6 章（访问数据库）
- Mail Session：代表 JNDI Mail Session 资源，参见第 19 章（开发 Java Mail Web 应用）
- Environment Entry：代表环境变量
- User Database：代表安全域中的用户数据库

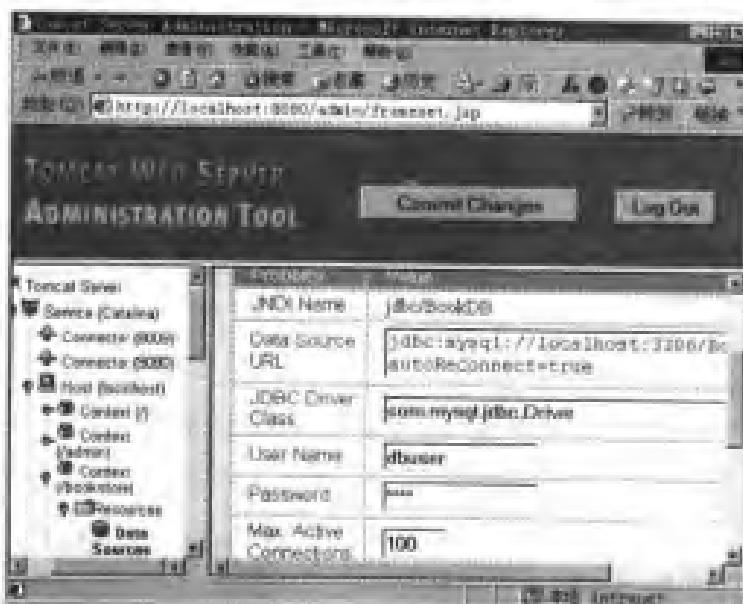


图 10-4 数据源 jdbc/BookDB 的属性

### 3. User Definition 目录

User Definition 用于配置安全域中的用户信息，可以定义用户、角色和组，参见第 11 章（安全域）。

#### 10.2.2 配置<Logger>元素

下面以配置<Logger>元素为例，说明如何操纵 Tomcat 的控制平台。<Logger>元素代表了 Catalina 容器的日志记录器，它可以嵌套在 Engine、Host 和 Context 这 3 种 Catalina 容器中。子类容器如果没有定义自己的<Logger>元素，将继承父类容器的<Logger>元素。

所有的<Logger>元素都包含以下属性：

- className：代表实现日志记录器的类的名字
- verbosity：指定日志记录器的日志级别。可选的值包括：0 (FATAL)、1 (ERROR)、2 (WARNING)、3 (INFO) 和 4 (DEBUG)。只有级别比 verbosity 的值小或者相等的日志信息才会被输出到日志设备上。例如，如果 verbosity 的值为 2 (WARNING)，那么 FATAL、ERROR、WARNING 类型的日志就会被输出到日志设备上，其他级别的日志被忽略

一共有 3 种类型的 Logger：FileLogger、SystemErrLogger 和 SystemOutLogger。

- FileLogger 的实现类为 org.apache.catalina.logger.FileLogger，它将日志信息保存到文件中，FileLogger 的属性参见表 10-1
- SystemErrLogger 的实现类为 org.apache.catalina.logger.SystemErrLogger，它把日志信息输出到 Tomcat 服务器指定的标准错误输出流中，在默认的 Tomcat 的启动脚本中，标准错误输出流指向<CTALINA\_HOME>/logs/catalina.out 文件
- SystemOutLogger 的实现类为 org.apache.catalina.logger.SystemOutLogger，它把日志信息输出到 Tomcat 服务器指定的标准输出流中。在默认的 Tomcat 的启动脚本

中，标准输出流指向<CATALINA\_HOME>/logs/catalina.out 文件

表 10-1 FileLogger 的属性

属性	描述
directory	指定日志文件的绝对目录或相对于<CATALINA_HOME>的相对目录。该属性的默认值为<CATALINA_HOME>/logs
prefix	指定日志文件名的前缀
suffix	指定日志文件名的扩展名
timestamp	如果设为 true，表示将把生成日志的时间和日期也保存到日志文件中

如果要为某种 Catalina 容器配置<Logger>元素，一种办法是直接修改 server.xml 文件，在这个 Catalina 容器元素下加入<Logger>元素，例如，以下是为 localhost 配置 FileLogger 的代码：

```
<Host name="localhost" debug="0" appBase="webapps"
      unpackWARs="true" autoDeploy="true">

    <Logger className="org.apache.catalina.logger.FileLogger"
          prefix="localhost_log." suffix=".txt"
          timestamp="true"/>

    <!-- 其他配置项 -->

</Host>
```

此外，也可以通过 Tomcat 的控制平台来配置<Logger>元素。以下是为 jsp-examples 应用配置 FileLogger 的步骤。

### 步骤

(1) 访问 <http://localhost:8080/admin>，输入用户名：admin，口令：1234，登录 Tomcat 的控制平台，从左边窗口的目录树中找到并选中 Context (/jsp-examples) 目录，然后从右边窗口的【Context Actions】下拉框中选中【Create New Logger】菜单，如图 10-5 所示。

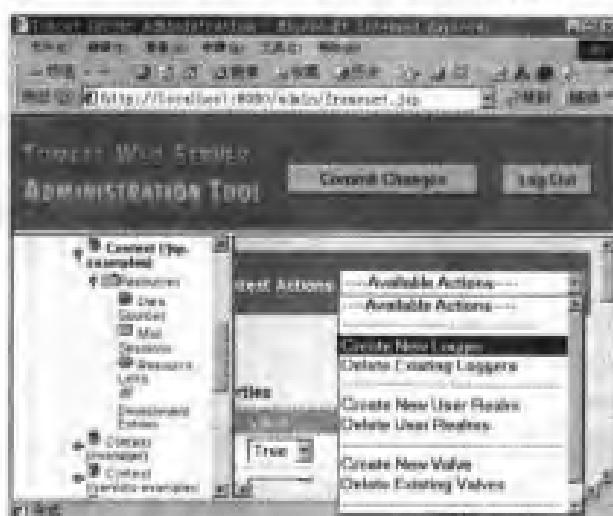


图 10-5 创建新的日志记录器

(2) 在配置 Logger 的窗口中为 Logger 的各个属性赋值, 如图 10-6 所示。

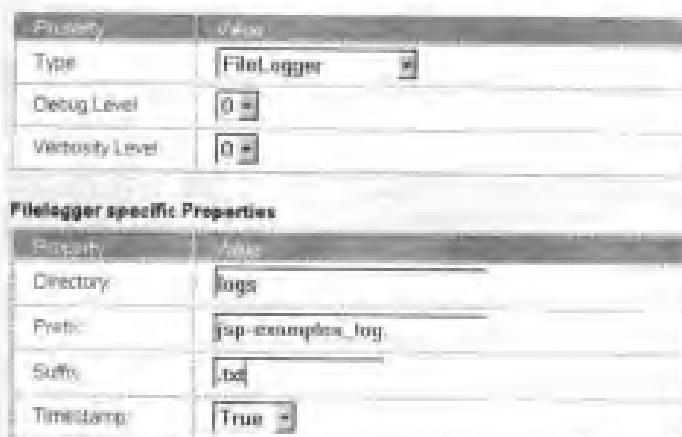


图 10-6 为 jsp-examples 应用设置 FileLogger

(3) 编辑完毕后, 单击【Save】和【Commit Changes】按钮, 这时配置才会真正生效。

(4) 重启 Tomcat 服务器, 此时在<CATALINA\_HOME>/logs 目录下会生成一个名为 jsp-examples\_log.2003-09-02.txt 的日志文件, 打开文件会看到如下日志内容:

```
2003-09-02 20:50:42 createObjectName with
StandardEngine[Catalina].StandardHost[localhost].StandardContext[/jsp-examples]
```

### 10.3 Tomcat 的管理平台

在 Tomcat 管理平台的登录窗口中输入用户名: manager, 口令: 1234, 将登录管理平台, 显示如图 10-7 所示的页面。

Path	Display Name	Running Sessions	Comments
/	Welcome to Tomcat	0	Start Stop Reload Deploy
/admin	Tomcat Administration Application	1	Start Stop Reload Deploy
/charmers		0	Start Stop Reload Deploy
/hellolog		0	Start Stop Reload Deploy

图 10-7 manager 应用的主页

从图 10-7 中可以看到，在 Tomcat 的管理平台上列出了所有的 Web 应用和它们的状态，并且提供了 Start、Stop、Reload 和 Undeploy 命令。通过这些命令，可以在 Tomcat 服务器始终处于运行状态下管理 Web 应用。这些命令的描述参见表 10-2。

表 10-2 Web 应用的管理命令

命令	描述
Start	启动 Web 应用
Stop	停止 Web 应用
Reload	停止 Web 应用，重新加载 Web 应用的各种组件，如 Servlet、JSP 和类文件，然后重新启动 Web 应用
Undeploy	卸载 Web 应用，并且删除<CATALINA_HOME>/webapps 目录下该 Web 应用的文件资源

Tomcat 的管理平台还可以发布 Web 应用，它提供了两种发布方式，如图 10-8 所示。第一种方式是发布位于<CATALINA\_HOME>/webapps 目录下的 Web 应用，还有一种方式是发布位于文件系统任意位置的 WAR 文件。

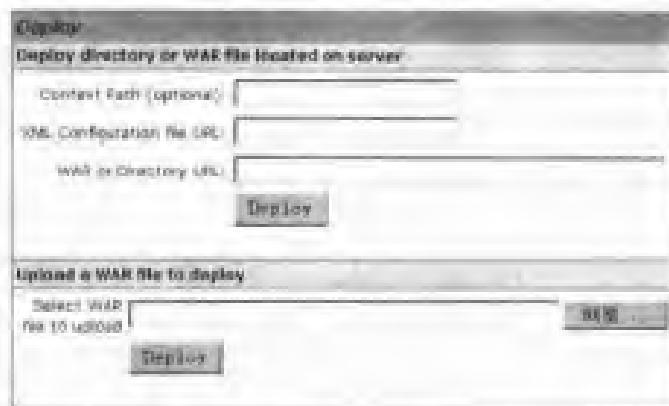


图 10-8 发布 Web 应用的表单

下面举例说明如何操纵 Tomcat 的管理平台。假定要发布一个名为 myhelloapp 的 Web 应用，可以把本书 2.2.8 小节生成的 helloapp.war 文件改名为 myhelloapp.war。

### 步骤

(1) 把 myhelloapp.war 拷贝到<CATALINA\_HOME>/webapps 目录下，接着在“Deploy directory or WAR file located on server”一栏中输入如图 10-9 所示的内容。



图 10-9 发布位于服务器上的 myhelloapp.war

然后单击【Deploy】按钮，这个 Web 应用就被发布了。可以访问 <http://localhost:8080/>

myhelloapp/index.htm，来验证发布是否成功。

(2) 选择 Undeploy 命令删除 myhelloapp 应用，这个命令执行后，会发现 <CATALINA\_HOME>/webapps 目录下的 myhelloapp.war 文件被删除。

(3) 把 myhelloapp.war 文件拷贝到文件系统的任意地方，比如 C:\myhelloapp.war，然后在“upload a war file to deploy”一栏中输入如图 10-10 所示内容。



图 10-10 发布位于文件系统的任意地方的 myhelloapp.war

然后单击【Deploy】按钮，这个 Web 应用就被发布了。可以访问 <http://localhost:8080/myhelloapp/index.htm>，来验证发布是否成功。如果发布成功，则会看到 Tomcat 把 myhelloapp.war 自动拷贝到<CATALINA\_HOME>/webapps 目录下。



通过以上两种方式发布 Web 应用，Tomcat 均不会修改 server.xml 文件，在 server.xml 文件中不会添加这个 Web 应用的<Context>元素。Tomcat 运行这个 Web 应用时，会采用默认的 Context 配置。

## 10.4 小结

本章介绍了 Tomcat 的基于 Web 方式的控制和管理平台。通过控制平台，可以配置 Tomcat 服务器以及 Web 应用；通过管理平台，可以在不重启 Tomcat 服务器的情况下，方便地发布、启动、停止或卸除 Web 应用。这里以配置日志记录器为例，讲解了控制平台的使用方法。

事实上，在配置 Tomcat 服务器时，无论是手工修改 server.xml 文件，还是通过控制平台来配置，都要求用户对各种配置元素的属性很了解。在本书的其他章节中，为了便于讲解这些配置元素的属性，仍然采用手工修改 server.xml 文件的方式来配置 Tomcat 服务器。

# 第 11 章 安 全 域

本章介绍如何通过安全域来保护 Web 应用的资源。首先介绍安全域的概念，接着介绍如何为 Web 资源配置安全约束，然后详细讲解配置内存域、JDBC 域和数据源域的步骤。

## 11.1 安全域概述

安全域是 Tomcat 服务器用来保护 Web 应用的资源的一种机制。在安全域中可以配置安全验证信息，即用户信息（包括用户名和口令）以及用户和角色的映射关系。每个用户可以拥有一个或多个角色，每个角色限定了可访问的 Web 资源。一个用户可以访问其拥有的所有角色对应的 Web 资源。Web 客户必须以某种用户身份才能登录 Web 应用系统，该客户只能访问与这种用户身份对应的 Web 资源。Web 客户、用户、角色和受保护 Web 资源的关系如图 11-1 所示。

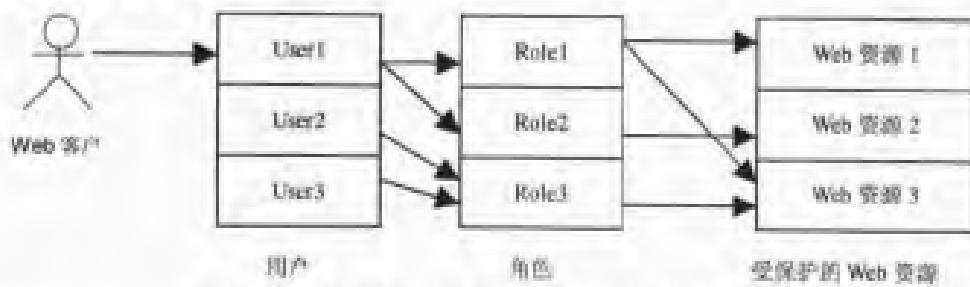


图 11-1 Web 客户、用户、角色和受保护 Web 资源的关系

根据图 11-1，假定 Web 客户以 User1 的身份登录 Web 应用，那么他拥有的角色是 Role1 和 Role2，角色 Role1 可以访问 Web 资源 1 和 Web 资源 3，角色 Role2 可以访问 Web 资源 2，所以用户 User1 可以访问 Web 资源 1、Web 资源 2 和 Web 资源 3。

安全域是 Tomcat 内置的功能，在 org.apache.catalina.Realm 接口中声明了把一组用户名、口令及所关联的角色集成到 Tomcat 中的方法。Tomcat5 提供了 4 个实现这一接口的类，它们分别代表了 4 种安全域类型，参见表 11-1。

表 11-1 安全域的类型

安全域类型	类 名	描 述
内存域	MemoryRealm	在初始化阶段，从 XML 文件中读取安全验证信息，并把它们以一组对象的形式存放在内存中
JDBC 域	JDBCRealm	通过 JDBC 驱动程序访问存放在数据库中的安全验证信息
数据源域	DataSourceRealm	通过 JNDI 数据源访问存放在数据库中的安全验证信息
JNDI 域	JNDIRealm	通过 JNDI provider 访问存放在基于 LDAP 的目录服务器中的安全验证信息

不管配置哪一种类型的安全域，都包含以下步骤。

### 步骤

- (1) 为 Web 资源设置安全约束，参见 11.2 节。
- (2) 在 conf/server.xml 文件中配置<Realm>元素，在这个元素中指定安全域的类名以及相关的属性。形式如下：

```
<Realm className="..." 实现这一安全域的类的名字"
      ... 其他属性 .../>
```

<Realm>元素可以嵌入到 3 种不同的 Catalina 容器元素中，这直接决定<Realm>的作用范围，例如，它可以决定哪些 Web 应用可以共享这一 Realm，参见表 11-2。

表 11-2 <Realm>元素在 server.xml 中的嵌入位置

嵌入位置	描述
嵌入到<Engine>元素中	所有虚拟主机上的所有 Web 应用共享这个 Realm。例外情况是在这个<Engine>下的<Host> 或 <Context> 元素下还定义了各自的 Realm 元素
嵌入到<Host>元素中	<Host>下的所有 Web 应用共享这个 Realm。例外情况是在这个<Host>下的<Context> 元素下还定义了各自的 Realm 元素
嵌入到 <Context>元素中	只有<Context>元素中的 Web 应用才能使用这个 Realm 元素

## 11.2 为 Web 资源设置安全约束

在上一节中，介绍过每种角色只能访问特定的 Web 资源。通过为 Web 资源设置安全约束，可以指定某种 Web 资源可以被哪些角色访问。例如，对于一个网上商店应用，假定它包含了/shopping、/order、/admin 等 URL 入口。其中客户可以访问/shopping，进行浏览商品、购物等活动；销售人员可以访问/order，管理订单信息；系统管理人员可以访问/admin，管理整个 Web 应用。公司里的小张是个销售人员，他负责管理订单信息，有时他也在网上购物。

在上述例子中，可以确定/shopping、/order、/admin 均为受保护的 Web 资源，它们分别只能被 customer、salesman 和 administrator 角色访问。用户小张拥有 customer 和 salesman 角色，如图 11-2 所示。

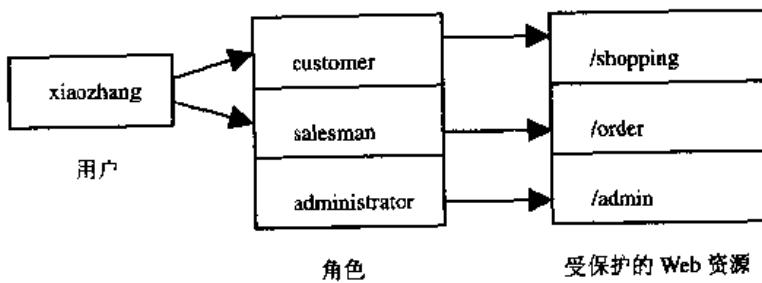


图 11-2 网上商店中受保护的 Web 资源



为 Web 资源设置安全约束时，只指定某种 Web 资源可以被哪些角色访问，并不涉及定义用户信息。用户信息以及用户和角色的映射关系是在配置 <Realm> 元素时设定的。

为 Web 资源设置安全约束，需要在 Web 应用的 web.xml 文件中加入 <security-constraint>、<login-config> 和 <security-role> 元素。

下面以 Tomcat 的控制平台应用（admin 应用）为例，讲解如何配置这些元素。此外还将为第 2 章介绍的 helloapp 应用设置安全约束。Tomcat 的 admin 应用的 web.xml 文件的位置为 <CATALINA\_HOME>/server/webapps/admin/web.xml。

本书配套光盘的 sourcecode/chapter11/web.xml 文件，是增加了安全约束配置代码的 helloapp 应用的配置文件，可以将这个文件覆盖原来的 web.xml 文件。

### 11.2.1 在 web.xml 中加入<security-constraint>元素

在 <security-constraint> 元素中指定受保护的 Web 资源以及所有可以访问该 Web 资源的角色。例如在 Tomcat 的 admin 应用中声明了如下安全约束：

```
<security-constraint>
    <display-name>Tomcat Server Configuration Security Constraint</display-name>
    <web-resource-collection>
        <web-resource-name>Protected Area</web-resource-name>
        <!-- Define the context-relative URL(s) to be protected -->
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.do</url-pattern>
        <url-pattern>*.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <!-- Anyone with one of the listed roles may access this area -->
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

以上安全约束代码指明：只有 admin 角色才能访问 admin 应用中 \*.jsp、\*.do 和 \*.html 资源。

再看一下 Tomcat 自带的另一个 Web 应用 jsp-examples 中的安全约束配置。这个应用的 web.xml 文件位于 <CATALINA\_HOME>/webapps/jsp-examples/WEB-INF 目录下：

```
<security-constraint>
    <display-name>Example Security Constraint</display-name>
    <web-resource-collection>
        <web-resource-name>Protected Area</web-resource-name>
        <!-- Define the context-relative URL(s) to be protected -->
        <url-pattern>/security/protected/*</url-pattern>
        <!-- If you list http methods, only those methods are protected -->
        <http-method>DELETE</http-method>
        <http-method>GET</http-method>
```

```

<http-method>POST</http-method>
<http-method>PUT</http-method>
</web-resource-collection>
<auth-constraint>
    <!-- Anyone with one of the listed roles may access this area -->
    <role-name>tomcat</role-name>
    <role-name>role1</role-name>
</auth-constraint>
</security-constraint>

```

以上的安全约束代码指明：只有 tomcat 和 role1 角色可以以 DELETE、GET、POST 或 PUT 方式访问 jsp-examples 应用中 URL 为 /security/protected/ 下的 Web 资源。

<security-constraint> 元素的各个属性的说明参见表 11-3。

表 11-3 <security-constraint> 元素的属性

属性	说明
<web-resource-collection>	声明受保护的 Web 资源
<web-resource-name>	标识受保护的 Web 资源
<url-pattern>	指定受保护的 URL 路径
<http-method>	指定受保护的 HTTP 方法，如 GET、POST 或 PUT。如果没有定义 HTTP 方法，那么所有 HTTP 方法都受到保护。如果某种 HTTP 方法受到保护，则表示当客户通过这种方法访问受保护的 Web 资源时，要求通过安全验证
<auth-constraint>	声明可以访问受保护资源的角色，可以包含多个<role-name>子元素
<role-name>	指定可以访问受保护资源的角色

下面，为 helloapp 加上安全约束，假定只有 friend 和 guest 角色可以访问 helloapp 应用下的所有 Web 资源，应该把这段代码加入到 helloapp 应用的 web.xml 的 <web-app> 元素中。

```

<security-constraint>
    <display-name>HelloApp Configuration Security Constraint</display-name>
    <web-resource-collection>
        <web-resource-name>Protected Area</web-resource-name>
        <!-- Define the context-relative URL(s) to be protected -->
        <url-pattern>/* </url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <!-- Anyone with one of the listed roles may access this area -->
        <role-name>friend</role-name>
        <role-name>guest</role-name>
    </auth-constraint>
</security-constraint>

```

## 11.2.2 在 web.xml 中加入<login-config>元素

接下来，在 web.xml 文件中再加入 <login-config> 元素，它指定当 Web 客户访问受保护的 Web 资源时，系统弹出的登录对话框的类型。例如在 Tomcat 的 admin 应用中定义了如

下<login-config>元素：

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Tomcat Server Configuration Form-Based Authentication Area</realm-name>
    <form-login-config>
        <form-login-page>/login.jsp</form-login-page>
        <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
</login-config>
```

<login-config>元素的各个属性的说明参见表 11-4。

表 11-4 <login-config>元素的属性

属性	说 明
<auth-method>	指定验证方法。它有 3 个可选值：BASIC（基本验证）、DIGEST（摘要验证）、FORM（基于表单的验证）
<realm-name>	指定安全域的名称
<form-login-config>	当验证方法为 FORM 时，配置验证网页和出错网页
<form-login-page>	当验证方法为 FORM 时，指定验证网页
<form-error-page>	当验证方法为 FORM 时，指定出错网页

在表 11-4 中提到 3 种验证方法：基本验证（Basic Authentication）、摘要验证（Digest Authentication）和基于表单的验证（Form Authentication）。下面分别介绍这 3 种验证方法。

### 1. 基本验证

如果 Web 应用采用基本验证，当客户访问受保护的资源时，浏览器会先弹出一个对话框，要求用户输入用户名和密码。如果客户输入的用户名和密码正确，Web 服务器就允许他访问这些资源；否则，在接连 3 次尝试失败之后，会显示一个错误消息页面。这个方法的缺点是把用户名和密码从客户端传送到 Web 服务器时，在网络上传送的数据采用 Base64 编码（全是可读文本），因此这种验证方法不是非常安全。

在 helloapp 应用的 web.xml 中加入如下<login-config>元素：

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>HelloApp realm</realm-name>
</login-config>
```

当访问 helloapp 应用时，浏览器端会先弹出一个对话框，要求输入用户名和密码，如图 11-3 所示。



图 11-3 基本验证的登录窗口

## 2. 摘要验证

摘要验证方法和基本验证的区别在于：前者不会在网络中直接传输用户密码，而是首先采用 MD5（Message Digest Algorithm）对用户密码进行加密，然后传输加密后的数据，这种验证方法显然更为安全。

在 helloapp 应用的 web.xml 中加入如下<login-config>元素：

```
<login-config>
    <auth-method>DIGEST</auth-method>
    <realm-name>HelloApp realm</realm-name>
</login-config>
```

当访问 helloapp 应用时，浏览器端将先弹出一个对话框，要求输入用户名和密码，如图 11-4 所示。

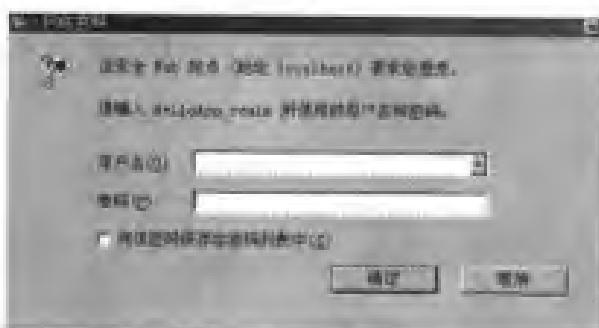


图 11-4 摘要验证的登录窗口

## 3. 基于表单的验证

基于表单的验证方法和基本验证方法的区别在于：前者使用自定义的登录页面来代替标准的登录对话框。在<form-login-config>元素中可以设定登录页面以及验证失败时的出错页面。

用户自定义的验证网页中必须提供一个登录表单，表单中和用户名对应的文本框必须命名为 j\_username，和密码对应的文本框必须命名为 j\_password，并且表单的 action 的值必须为 j\_security\_check。

例如为 helloapp 应用创建如下验证网页 usercheck.jsp，参见例程 11-1。

例程 11-1 usercheck.jsp

```
<html>
<head>
<title>Login Page for helloapp</title>
<body bgcolor="white">
<form method="POST" action=j_security_check>
<table border="0" cellspacing="5">
<tr>
    <th align="right">Username:</th>
    <td align="left"><input type="text" name="j_username"></td>
</tr>
<tr>
```

```

<th align="right">Password:</th>
<td align="left"><input type="password" name="j_password"></td>
</tr>
<tr>
<td align="right"><input type="submit" value="Log In"></td>
<td align="left"><input type="reset" value="reset"></td>
</tr>
</table>
</form>
</body>
</html>

```

再为 helloapp 应用创建一个错误处理页面 error.jsp，参见例程 11-2。

例程 11-2 error.jsp

```

<!–设置中文输出–>
<%@ page contentType="text/html; charset=GB2312" %>
<html><head><title>Error Page</title></head>
<body>
<p>
    请输入合法的用户名和口令
</p>
</body></html>

```

下面，在 helloapp 应用的 web.xml 中加入如下<login-config>元素：

```

<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>HelloApp realm</realm-name>
    <form-login-config>
        <form-login-page>/usercheck.jsp</form-login-page>
        <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
</login-config>

```

当访问 helloapp 应用时，浏览器端将会先显示 usercheck.jsp 网页，要求输入用户名和密码，如图 11-5 所示。



图 11-5 基于表单验证的登录窗口

如果在登录窗口中输入非法的用户名或口令，将会显示出错页面，如图 11-6 所示。



图 11-6 验证失败后的错误页面

### 11.2.3 在 web.xml 中加入<security-role>元素

最后，应该在 web.xml 中加入<security-role>元素，指明这个 Web 应用引用的所有角色名字。例如，在 admin 应用中声明引用了 admin 角色：

```
<security-role>
    <description>
        The role that is required to log in to the Administration Application
    </description>
    <role-name>admin</role-name>
</security-role>
```

在 helloapp 应用中引用了 guest 和 friend 角色，因此也要加入相应的<security-role>元素：

```
<!-- Security roles referenced by this web application -->
<security-role>
    <description>
        The role that is required to log in to the helloapp Application
    </description>
    <role-name>guest</role-name>
    <role-name>friend</role-name>
</security-role>
```

## 11.3 内存域

内存域是由 org.apache.catalina.realm.MemoryRealm 类来实现的。MemoryRealm 类从一个 XML 文件中读取用户信息。默认情况下，该 XML 文件为<CATALINA\_HOME>/conf/tomcat-users.xml。<role>元素用来定义角色，<tomcat-users>元素用来设定用户信息。以下是 tomcat-users.xml 文件中的代码：

```
<!--
  NOTE: By default, no user is included in the "manager" role required
  to operate the "/manager" web application. If you wish to use this app,
```

you must define such a user - the username and password are arbitrary.

```
-->
<tomcat-users>
    <role rolename="tomcat"/>
    <role rolename="role1"/>
    <role rolename="manager"/>
    <role rolename="admin"/>
    <user name="admin" password="" roles="admin" />
    <user name="manager" password="" roles="manager" />
    <user name="tomcat" password="tomcat" roles="tomcat" />
    <user name="role1" password="tomcat" roles="role1" />
    <user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

在以上 XML 文件中，定义了 4 种角色和 5 个用户（包括用户名和口令），还指定了每个用户拥有的角色。其中 admin 和 manager 用户是在第 10 章中定义的，他们分别拥有 admin 和 manager 角色。当用户登录到 Tomcat 的 admin 应用时，必须以 admin 用户身份登录，参见 10.1 节。

下面，在 tomcat-users.xml 中加入两个用户 xiaowang 和 xiaoming，他们分别拥有 friend 和 guest 角色。以下是修改后的 tomcat-users.xml 文件：

```
<tomcat-users>
    <role rolename="tomcat"/>
    <role rolename="role1"/>
    <role rolename="manager"/>
    <role rolename="admin"/>
    <role rolename="friend"/>
    <role rolename="guest"/>

    <user name="admin" password="1234" roles="admin" />
    <user name="manager" password="1234" roles="manager" />
    <user name="tomcat" password="tomcat" roles="tomcat" />
    <user name="role1" password="tomcat" roles="role1" />
    <user name="both" password="tomcat" roles="tomcat,role1" />
    <user name="xiaowang" password="1234" roles="friend" />
    <user name="xiaoming" password="1234" roles="guest" />
</tomcat-users>
```

接下来在 server.xml 中 helloapp 应用对应的<Context>元素内加入如下<Realm>元素：

```
<Realm className="org.apache.catalina.realm.MemoryRealm" />
```

下面总结配置 MemoryRealm 的步骤。

### 步骤

- (1) 按 11.2 节的步骤在 helloapp 应用的 web.xml 中配置安全约束，假定采用基本验证方法。
- (2) 在 tomcat-user.xml 文件中定义用户、角色以及两者的映射关系。
- (3) 在 server.xml 中加入相应的<Realm>元素，在本书配套光盘的 sourcecode/chapter11

/server\_modify\_realm.xml 文件中提供了<Realm>元素的代码。

(4) 重启 Tomcat 服务器, 然后访问 <http://localhost:8080/helloapp/index.htm>, 会看到浏览器端弹出一个安全验证窗口, 如图 11-3 所示。在安全验证窗口中输入用户名: xiaowang, 口令: 1234, 就可以通过安全验证, 然后访问 index.htm 文件。

## 11.4 JDBC 域

JDBCRealm 通过 JDBC 驱动程序访问存放在关系型数据库中的安全验证信息。JDBC 域使得安全配置非常灵活。当修改了数据库中的安全验证信息后, 不必重启 Tomcat 服务器, 因为数据库服务器和 Tomcat 服务器是相互独立的。

当用户第一次访问受保护的资源时, Tomcat 将调用 Realm 的 authenticate()方法, 该方法从数据库中读取最新的安全验证信息。

该用户通过验证后, 在用户访问 Web 资源期间, 用户的各种验证信息被保存在缓存中(对于 FORM 类型的验证, 这表示验证信息直到会话结束才失效; 对于 BASIC 类型的验证, 这表示验证信息直到浏览器关闭才失效)。因此, 如果此时对数据库中安全验证信息做了修改, 这种修改对正在访问 Web 资源的用户无效, 只有当用户再次登录时, 才会生效。

### 11.4.1 用户数据库的结构

必须在数据库中创建两张表: users 和 user\_roles, 这两张表包含了所有的安全验证信息。users 表用来定义用户信息, 包括用户名和口令, user\_roles 表用来定义用户和角色的映射关系。一个用户可以没有, 或者有一个或多个角色。这两张表的结构参见表 11-5 和表 11-6。

表 11-5 users

字段	字段类型	描述
user_name	varchar(15)	用户名
user_pass	varchar(15)	用户的口令

表 11-6 user\_roles

字段	字段类型	描述
user_name	varchar(15)	用户名
role_name	varchar(15)	该用户所拥有的角色

在这两张表中添加一些用户信息, 参见表 11-7 和 11-8。

表 11-7 users 的内容

user_name	user_pass
xiaowang	1234
xiaoming	1234

表 11-8 user\_roles 的内容

user_name	role_name
xiaowang	friend
xiaoming	guest

## 11.4.2 在 MySQL 中创建和配置用户数据库

下面以 MySQL 为例，创建以上用户数据库。关于 MySQL 服务器的用法可以参照 6.1 节（安装和配置 MySQL 数据库）中的内容。下面是创建和配置用户数据库的步骤。

### 步骤

(1) 在<MYSQL\_HOME>/bin 目录中启动 MySQL 客户程序，创建名为 tomcatusers 的用户数据库，SQL 命令如下：

```
create database tomcatusers;
```

(2) 将当前数据库设为 tomcatusers 数据库，SQL 命令如下：

```
use tomcatusers
```

(3) 创建 users 表，SQL 命令如下：

```
create table users(
    user_name varchar(15) not null primary key,
    user_pass varchar(15) not null
);
```

(4) 创建 user\_roles 表，SQL 命令如下：

```
create table user_roles(
    user_name varchar(15) not null,
    role_name varchar(15) not null,
    primary key(user_name,role_name)
);
```

(5) 向 users 表中加入用户数据，SQL 命令如下：

```
insert into users values("xiaowang","1234");
insert into users values("xiaoming","1234");
```

(6) 向 user\_roles 表中加入数据，SQL 命令如下：

```
insert into user_roles values("xiaowang","friend");
insert into user_roles values("xiaoming","guest");
```

在本书配套光盘的 sourcecode/chapter11/tomcatusers.sql 文件中提供了上述 SQL 脚本。

## 11.4.3 配置<Realm>元素

用户数据库创建好以后，应该把 MySQL 数据库的驱动程序拷贝到<CATALINA\_HOME>/common/lib 中，然后在 server.xml 中 helloapp 应用对应的<Context>元素内加入如下<Realm>元素：

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
```

```

driverName="com.mysql.jdbc.Driver" debug="99"
connectionURL="jdbc:mysql://localhost/tomcatusers"
connectionName="dbuser" connectionPassword="1234"
userTable="users" userNameCol="user_name" userCredCol="user_pass"
userRoleTable="user_roles" roleNameCol="role_name" />

```

如果在<Context>中已经有<Realm>元素，则应该把原来的<Realm>元素注释掉。Realm 元素的各个属性的说明参见表 11-9。

表 11-9 Realm 元素的属性

属性	描述
className	指定 Realm 的类名，在这里为 org.apache.catalina.realm.JDBCRealm
connectionName	用于建立数据库连接的用户名
connectionPassword	用于建立数据库连接的口令
connectionURL	用于建立数据库连接的数据库 URL
debug	设定跟踪级别
digest	设定存储口令时的加密方式
driverName	JDBC 驱动程序类的名字
roleNameCol	在 user_roles 表中代表角色的字段名
userCredCol	在 users 表中代表用户口令的字段名
userNameCol	在 users 和 user_roles 表中代表用户名字的字段名
userRoleTable	指定用户与角色映射关系的表
userTable	指定用户表

下面总结了配置 JDBCRealm 的步骤。

### 步骤

- (1) 按 11.2 节的步骤在 helloapp 应用的 web.xml 中配置安全约束，假定采用基本验证方法。
- (2) 在 MySQL 中创建 tomcatusers 数据库、users 表和 user\_roles 表。
- (3) 将 MySQL 的 JDBC 驱动程序拷贝到<CATALINA\_HOME>/common/lib 中。
- (4) 在 server.xml 中加入相应的<Realm>元素，在本书配套光盘的 sourcecode/chapter11/server\_modify\_realm.xml 文件中提供了<Realm>元素的代码。
- (5) 重启 Tomcat 服务器，然后访问 <http://localhost:8080/helloapp/index.htm>，会看到浏览器端弹出一个安全验证窗口，如图 11-3 所示。在安全验证窗口中输入用户名：xiaowang，口令：1234，就可以通过安全验证，然后访问 index.htm 文件。

## 11.5 DataSource 域

DataSourceRealm 和 JDBCRealm 很相似：两者都将安全验证信息存放在关系型数据库中，并且创建的用户数据库结构也相同。两者不同之处在于访问数据库的方式不一样：DataSourceRealm 通过 JNDI DataSource 来访问数据库，而 JDBCRealm 通过 JDBC 驱动程序

访问数据库。以下是配置 DataSourceRealm 的步骤。

步骤→

- (1) 按 11.2 节的步骤在 helloapp 应用的 web.xml 中配置安全约束，假定采用基本验证方法。
- (2) 参照 11.3 节的内容在 MySQL 中创建 tomcatusers 数据库、users 表和 user\_roles 表。
- (3) 将 MySQL 的 JDBC 驱动程序拷贝到<CATALINA\_HOME>/common/lib 中。
- (4) 参照 6.4 节(配置数据源)的内容，创建一个名为 jdbc/tomcatusers 的 DataSource，需要在 server.xml 中<GlobalNamingResources>元素下加入<Resource>和<ResourceParams>元素，在本书配套光盘的 sourcecode/chapter11/server\_modify\_datasource.xml 文件中提供了以下的配置代码：

```
<GlobalNamingResources>

    ...
    ...

    <Resource name="jdbc/tomcatusers"
              auth="Container"
              type="javax.sql.DataSource" />

    <ResourceParams name="jdbc/tomcatusers">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <parameter>
            <name>maxActive</name>
            <value>100</value>
        </parameter>

        <parameter>
            <name>maxIdle</name>
            <value>30</value>
        </parameter>

        <parameter>
            <name>maxWait</name>
            <value>10000</value>
        </parameter>

        <parameter>
            <name>username</name>
```

```
<value>dbuser</value>
</parameter>
<parameter>
<name>password</name>
<value>1234</value>
</parameter>

<parameter>
<name>driverClassName</name>
<value>com.mysql.jdbc.Driver</value>
</parameter>

<parameter>
<name>url</name>
<value>jdbc:mysql://localhost:3306/tomcatusers?autoReconnect=true</value>
</parameter>

</ResourceParams>

</GlobalNamingResources>
```

为 DataSourceRealm 配置 DataSource，有以下几个值得注意的地方：

第一，在 server.xml 中已经存在<GlobalNamingResources>元素，它用于配置 Tomcat 服务器范围内的 JNDI 资源。在目前的 Tomcat 版本中，DataSourceRealm 从<GlobalNamingResources>中寻找 DataSource，如果把 DataSource 配置在其他 Catalina 容器中（如 Engine、Host 或 Context），会出现 NameNotFoundException。异常的原因为找不到 JNDI DataSource，从日志文件中可以看到这一异常。

第二，如果使用低于 Tomcat 5.0.12 的其他版本，即使正确配置了 DataSourceRealm，也会出现找不到 JNDI DataSource 的异常。在 Tomcat 5.0.12 及以上版本中改正了这个缺陷。

第三，在 Web 应用中并不会访问这个 DataSource，所以无需在 web.xml 中加入<resource-ref>元素，声明对 DataSource 的引用。

(5) 在 server.xml 中加入如下<Realm>元素，在本书配套光盘的 sourcecode/chapter11/server\_modify\_realm.xml 文件中提供了<Realm>元素的代码。

```
<Realm className="org.apache.catalina.realm.DataSourceRealm" debug="99"
      dataSourceName="jdbc/tomcatusers"
      userTable="users" userNameCol="user_name" userCredCol="user_pass"
      userRoleTable="user_roles" roleNameCol="role_name"/>
```

<Realm>元素的各个属性的说明参见表 11-10。

(6) 重启 Tomcat 服务器，然后访问 <http://localhost:8080/helloapp/index.htm>，会看到浏览器端弹出一个安全验证窗口，如图 11-3 所示。在安全验证窗口中输入用户名：xiaowang，口令：1234，就可以通过安全验证，然后访问 index.htm 文件。

表 11-10 &lt;Realm&gt;元素的属性

属性	描述
className	指定 Realm 的类名，在这里为 org.apache.catalina.realm.DataSourceRealm
dataSourceName	设置数据源的 JNDI 名字
debug	设置跟踪级别
digest	设置存储口令时的加密方式
roleNameCol	在 user_roles 表中代表角色的字段名
userCredCol	在 users 表中代表用户口令的字段名
userNameCol	在 users 和 user_roles 表中代表用户名的字段名
userRoleTable	指定用户与角色映射关系的表
userTable	指定用户表

## 11.6 在 Web 应用中访问用户信息

当通过验证的用户访问 Web 资源时，`HttpServletRequest` 接口的 `getRemoteUser()`方法可以返回访问当前网页的用户名，否则 `getRemoteUser()`方法返回 `null`。例如以下 `welcome.jsp` 文件中调用 `request.getRemoteUser()`方法，将当前用户名的名字输出到网页上：

```
<html>
<head>
    <title>helloapp</title>
</head>
<body>
    <b>Welcome: <%= request.getRemoteUser() %></b>
</body>
</html>
```

运行这个例子时，可以先按照 11.3 节的步骤配置好 `MemoryRealm`，将 `welcome.jsp` 发布到 `helloapp` 应用中。然后访问 `http://localhost:8080/welcome.jsp`，会看到浏览器端弹出一个安全验证窗口，如图 11-3 所示。在安全验证窗口中输入用户名：`xiaowang`，口令：`1234`，就可以通过安全验证，然后将看到 `welcome.jsp` 生成的网页，如图 11-7 所示。

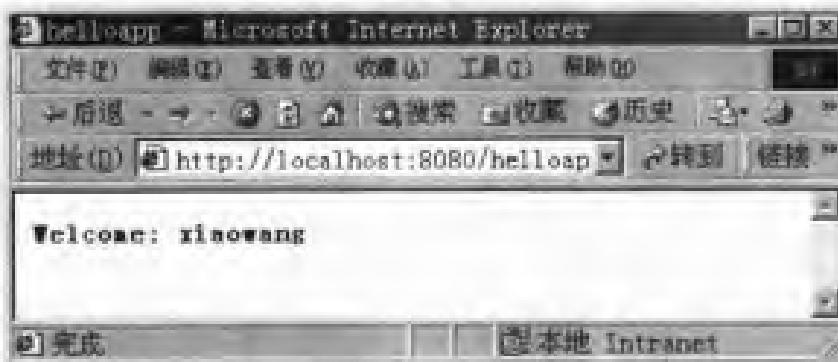


图 11-7 welcome.jsp 网页

## 11.7 小 结

安全域是 Tomcat 服务器用来保护 Web 应用的资源的一种机制。在安全域中可以配置安全验证信息，即用户信息（包括用户名和口令）以及用户和角色的映射关系。每个用户可以拥有一个或多个角色，每个角色限定了可访问的 Web 资源。安全域是 Tomcat 内置的功能，在 org.apache.catalina.Realm 接口中声明了把一组用户名、口令及所关联的角色集成到 Tomcat 中的方法。Tomcat5 提供了 4 个实现这一接口的类，它们分别代表了 4 种安全域类型：MemoryRealm、JDBCRealm、DataSourceRealm 和 JNDIRealm，这些安全域的差别在于存放安全验证信息的地点不一样。本章介绍了前 3 种安全域的配置，关于 JNDIRealm 的配置可以参考 Tomcat 的文档：

<CATALINA\_HOME>/webapps/tomcat-docs/realm-howto.html

# 第 12 章 Tomcat 阀

Tomcat 阀（Valve）用于对 Catalina 容器接收到的 HTTP 请求进行预处理。它是 Tomcat 专有的，目前还不能用于其他的 Servlet/JSP 容器。本章将介绍 Tomcat 阀的种类，还将详细介绍各种 Tomcat 阀的功能和使用方法。

## 12.1 Tomcat 阀简介

Tomcat 阀（Valve）能够对 Catalina 容器接收到的 HTTP 请求进行预处理。Tomcat 阀可以加入到 3 种 Catalina 容器中，它们是 Engine、Host 和 Context。Tomcat 阀在不同的容器中的作用范围参见表 12-1。

表 12-1 Tomcat 阀加入到 Catalina 容器中

容 器	描 述
Engine	加入到 Engine 中的 Tomcat 阀可以预处理该 Engine 接收到的所有 HTTP 请求
Host	加入到 Host 中的 Tomcat 阀可以预处理该 Host 接收到的所有 HTTP 请求
Context	加入到 Context 中的 Tomcat 阀可以预处理该 Context 接收到的所有 HTTP 请求

所有的 Tomcat 阀都实现了 org.apache.Catalina.Valve 接口或扩展了 org.apache.Catalina.valves.ValveBase 类。Tomcat 阀分为 4 种，分别是：

- 客户访问日志阀（Access Log Valve）
- 远程地址过滤器（Remote Address Filter）
- 远程主机过滤器（Remote Host Filter）
- 客户请求记录器（Request Dumper）

不管是配置哪一种 Tomcat 阀，都需要在 server.xml 中加入<Valve>元素，它的形式为：

```
<Valve className="... 实现这种阀的类的名字"
      ... 其他属性 .../>
```

## 12.2 客户访问日志阀

客户访问日志阀（Access Log Valve）能够将客户的请求信息写到日志文件中。这些日志可以记录网页的访问次数、用户的会话活动和用户验证信息等。客户访问日志阀可以加入到 Engine、Host 或 Context 容器中，记录所在容器接收的 HTTP 请求信息。客户访问日志阀的工作过程如图 12-1 所示。

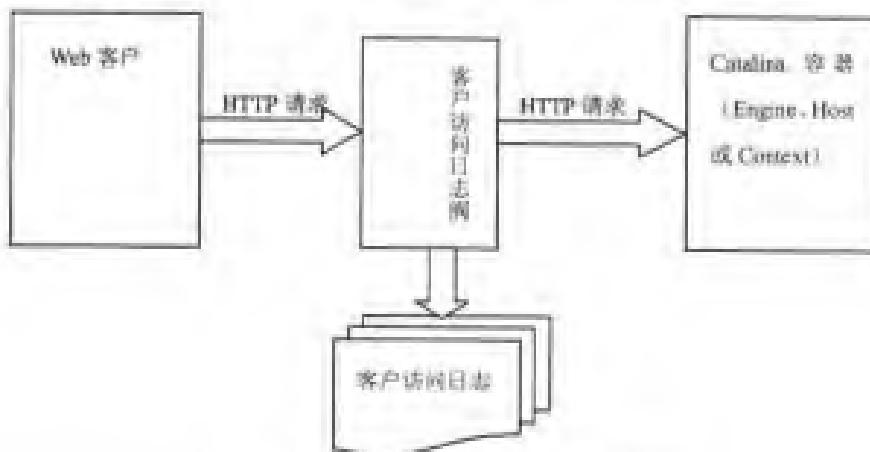


图 12-1 客户访问日志阀的工作过程

客户访问日志阀的<Valve>元素的属性描述参见表 12-2。

表 12-2 客户访问日志阀的&lt;Valve&gt;元素的属性

属性	描述
className	指定阀的实现类，这里为 org.apache.catalina.valves.AccessLogValve
directory	设定存放日志文件的绝对或相对于<CATALINA_HOME>的相对目录。该属性的默认值为 <CATALINA_HOME>/logs
pattern	设定日志的格式和内容
prefix	设定日志文件名前缀，默认值为 access_log
resolveHosts	如果设为 true，表示把远程 IP 地址解析为主机名；如果设为 false，表示直接记录远程 IP 地址。默认值为 false
suffix	设定日志文件的扩展名，默认值为 "

<Valve>元素的 pattern 属性用于设定日志的格式和内容，它有以下可选值：

- %a: 远程 IP 地址
- %A: 本地 IP 地址
- %b: 发送的字节数，不包括 HTTP Header。- 表示发送字节为零
- %B: 发送的字节数，不包括 HTTP Header
- %h: 远程主机名
- %H: 客户请求所用的协议
- %l: 远程逻辑用户名（目前总是返回-）
- %m: 客户请求所用的方法（GET、POST 等）
- %p: 接收到客户请求的本地服务器端口
- %q: 客户请求中的查询字符串（Query string），如果存在，以?开头
- %r: 客户请求的第一行内容（包括客户请求所用的方法以及请求的 URI）
- %s: 服务器响应结果中的 HTTP 状态代码
- %S: 用户 Session ID
- %t: 时间和日期
- %u: 经过安全验证的远程用户名。- 表示不存在远程用户名

- %U：客户请求的 URL 路径
- %v：本地服务器名

例如在 server.xml 中 localhost 的<Host>元素中加入如下<Valve>元素：

```
<Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
prefix="localhost_access_log." suffix=".txt" pattern="%h %l %u %t %r %s %b" resolveHosts="true" />
```

重启 Tomcat 服务器后，从浏览器端访问 helloapp 应用，此时在 <CATALINA\_HOME>/logs 目录下会生成一个 localhost\_access\_log.2003-08-24.txt 文件。文件内容如下：

```
sun - xiaowang [24/Aug/2003:13:37:34 0800] GET /helloapp/login.jsp HTTP/1.1 200 1029
sun - xiaowang [24/Aug/2003:13:38:14 0800] POST /helloapp/checker HTTP/1.1 200 383
```

可见，日志文件的内容是由 pattern 来指定的，上述文件中第二条日志记录和 pattern 格式的对应关系参见表 12-3。

表 12-3 日志记录和 pattern 格式的对应关系

pattern	日志内容	描述
%h	sun	远程客户主机名
%l	-	远程逻辑用户名
%u	xiaowang	经过安全验证的远程用户名
%t	[24/Aug/2003:13:38:14 0800]	时间和日期
%r	POST /helloapp/checker HTTP/1.1	客户请求的第一行内容
%s	200	服务器响应结果中的 HTTP 状态代码
%b	383	发送的字节数

pattern 属性的默认值为 common，它相当于“%h %l %u %t %r %s %b”的组合，所以上述<Valve>元素的 pattern 值也可以由 common 来代替，生成的日志文件和上面的例子一样。

```
<Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
prefix="localhost_access_log." suffix=".txt" pattern="common" resolveHosts="true" />
```

## 12.3 远程地址过滤器

远程地址过滤器（Remote Address Filter）可以根据远程客户的 IP 地址来决定是否接受客户的请求。在远程地址过滤器中，事先保存了一份被拒绝的 IP 地址清单和允许访问的 IP 地址清单，如果客户的 IP 地址在拒绝清单中，那么这个客户的请求不会被 Catalina 容器响应；如果客户的 IP 地址在允许访问清单中，那么这个客户的请求可以被 Catalina 容器响应。远程地址过滤器的工作过程如图 12-2 所示。

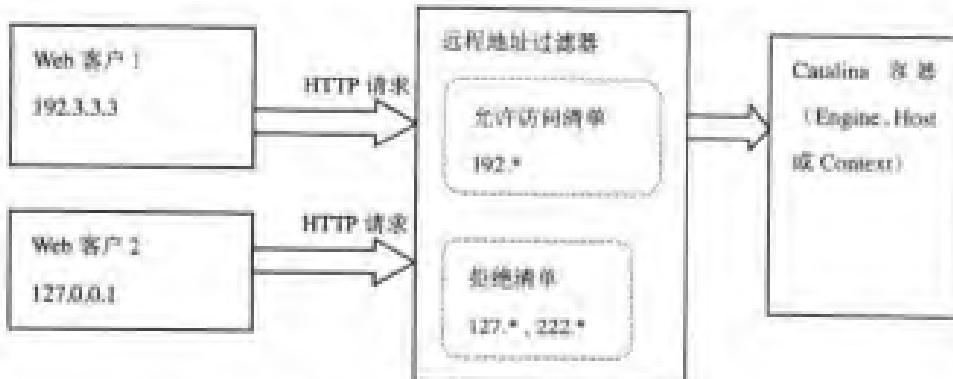


图 12-2 远程地址过滤器的工作过程

远程地址过滤器的<Valve>元素的属性描述参见表 12-4。

表 12-4 远程地址过滤器的&lt;Valve&gt;元素的属性

属性	描述
className	指定阀的实现类，这里为 org.apache.catalina.valves.RemoteAddrValve
allow	指定允许访问的客户 IP 地址。如果此项没有设定，表示只要客户 IP 地址不在 deny 清单中，就允许访问。多个 IP 地址以逗号隔开
deny	指定不允许访问的客户 IP 地址。多个 IP 地址以逗号隔开

例如，在 server.xml 中 localhost 的<Host>元素中加入如下<Valve>元素：

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
      deny="127.*, 222.*" />
```

以上代码表明，所有 IP 地址以 127 或 222 开头的客户都被拒绝访问 localhost 中的 Web 应用。

## 12.4 远程主机过滤器

远程主机过滤器（Remote Host Filter）可以根据远程客户的主机名，来决定是否接受客户的请求。在远程主机过滤器中，事先保存了一份被拒绝的主机名清单和允许访问的主机名清单，如果客户的主机名在拒绝清单中，那么这个客户的请求不会被 Catalina 容器响应；如果客户的主机名在允许访问清单中，那么这个客户的请求可以被 Catalina 容器响应。图 12-3 显示了远程主机过滤器的工作过程。

远程主机过滤器的<Valve>元素的属性描述参见表 12-5。

表 12-5 远程主机过滤器的&lt;Valve&gt;元素的属性

属性	描述
className	指定阀的实现类，这里为 org.apache.Catalina.valves.RemoteHostValve
allow	指定允许访问的客户主机名。如果此项没有设定，表示只要客户主机名不在 deny 清单中，就允许访问。多个主机名以逗号隔开
deny	指定不允许访问的客户主机名。多个主机名以逗号隔开

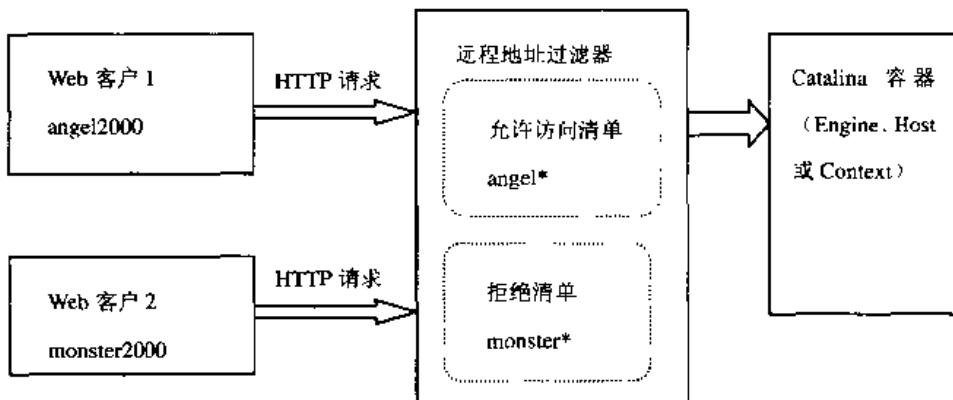


图 12-3 远程主机过滤器的工作过程

例如，在 server.xml 中 localhost 的<Host>元素中加入如下<Valve>元素：

```
<Valve className="org.apache.catalina.valves.RemoteHostValve"
      deny="monster*" />
```

以上代码表明，所有主机名中包含 monster 字符串的客户都被拒绝访问 localhost 中的 Web 应用。

## 12.5 客户请求记录器

客户请求记录器（Request Dumper）用于把客户请求的详细信息记录到日志文件中，这里的日志文件是在<Logger>元素中配置的（关于<Logger>元素的配置参见 10.2.2 节的内容），客户请求记录器是一个有效的跟踪工具，尤其是当 HTTP 请求中的 Header 或 Cookie 出了问题时，它可以跟踪客户请求的详细信息。客户请求记录器的工作过程如图 12-4 所示。

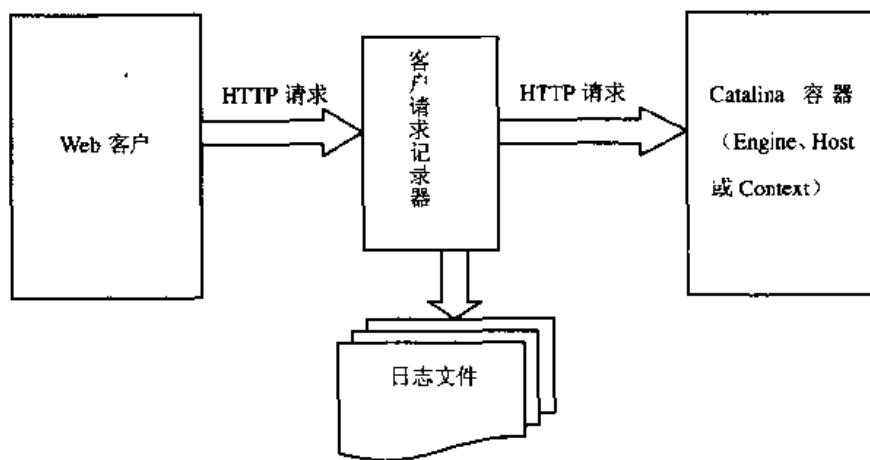


图 12-4 客户请求记录器的工作过程

假定在 server.xml 中 localhost 的<Host>元素中已经配置了如下<Logger>元素：

```
<Logger className="org.apache.catalina.logger.FileLogger"
      directory="logs" prefix="localhost_log." suffix=".txt"
      timestamp="true"/>
```

然后在 server.xml 中 localhost 的<Host>元素中加入如下<Valve>元素：

```
<Valve className="org.apache.catalina.valves.RequestDumperValve" />
```

重启 Tomcat 服务器后，从浏览器端访问 helloapp 应用，此时在<CATALINA\_HOME>/logs 目录下会生成一个 localhost\_log.2003-08-24.txt 文件，文件中内容如下：

```
2003-08-24 21:55:18 RequestDumperValve[localhost]: REQUEST URI  =/helloapp/index.htm
2003-08-24 21:55:18 RequestDumperValve[localhost]: authType=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: characterEncoding=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: contentLength=-1
2003-08-24 21:55:18 RequestDumperValve[localhost]: contentType=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: contextPath=/helloapp
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=accept=/*
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=accept-language=zh-cn
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=accept-encoding=gzip, deflate
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=user-agent=Mozilla/4.0 (compatible;
MSIE 5.01; Windows NT 5.0)
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=host=localhost:8080
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=connection=Keep-Alive
2003-08-24 21:55:18 RequestDumperValve[localhost]: header=authorization=Basic
eGhb3dhbmc6MTIzNA==
2003-08-24 21:55:18 RequestDumperValve[localhost]: locale=zh_CN
2003-08-24 21:55:18 RequestDumperValve[localhost]: method=GET
2003-08-24 21:55:18 RequestDumperValve[localhost]: pathInfo=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: protocol=HTTP/1.1
2003-08-24 21:55:18 RequestDumperValve[localhost]: queryString=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: remoteAddr=127.0.0.1
2003-08-24 21:55:18 RequestDumperValve[localhost]: remoteHost=sun
2003-08-24 21:55:18 RequestDumperValve[localhost]: remoteUser=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: requestedSessionId=null
2003-08-24 21:55:18 RequestDumperValve[localhost]: scheme=http
2003-08-24 21:55:18 RequestDumperValve[localhost]: serverName=localhost
2003-08-24 21:55:18 RequestDumperValve[localhost]: serverPort=8080
2003-08-24 21:55:18 RequestDumperValve[localhost]: servletPath=/index.htm
2003-08-24 21:55:18 RequestDumperValve[localhost]: isSecure=false
```

## 12.6 小结

Tomcat 阀（Valve）能够对 Catalina 容器接收到的 HTTP 请求进行预处理。Tomcat 阀可以加入到 3 种 Catalina 容器中，它们是 Engine、Host 和 Context。Tomcat 阀分为 4 种：客户访问日志阀（Access Log Valve）、远程地址过滤器（Remote Address Filter）、远程主机过滤器（Remote Host Filter）和客户请求记录器（Request Dumper）。配置 Tomcat 阀很简单，只要在 server.xml 的 Catalina 容器元素下加入<Valve>元素即可。在本书配套光盘的 sourcecode/chapter12 目录下的 server\_modify.xml 文件包含了本章的所有配置代码。

# 第 13 章 Servlet 过滤器

上一章介绍了 Tomcat 阀，它是 Tomcat 专有的预处理客户请求的工具。本章将介绍 Servlet 过滤器，它是在 Java Servlet 规范 2.3 中定义的，因此所有实现这一规范的 Servlet 容器都支持 Servlet 过滤器。

Servlet 过滤器能够对 Servlet 容器的请求和响应对象进行检查和修改。本章首先介绍 Servlet 过滤器的概念，然后介绍 Servlet 过滤器的创建和发布过程，最后讲解如何将 Servlet 过滤器串联起来工作。

## 13.1 Servlet 过滤器简介

Servlet 过滤器是在 Java Servlet 规范 2.3 中定义的，它能够对 Servlet 容器的请求和响应对象进行检查和修改。Servlet 过滤器本身并不生成请求和响应对象，它只提供过滤作用。Servlet 过滤器能够在 Servlet 被调用之前检查 Request 对象，修改 Request Header 和 Request 内容；在 Servlet 被调用之后检查 Response 对象，修改 Response Header 和 Response 内容。Servlet 过滤器负责过滤的 Web 组件可以是 Servlet、JSP 或 HTML 文件。Servlet 过滤器的过滤过程如图 13-1 所示。

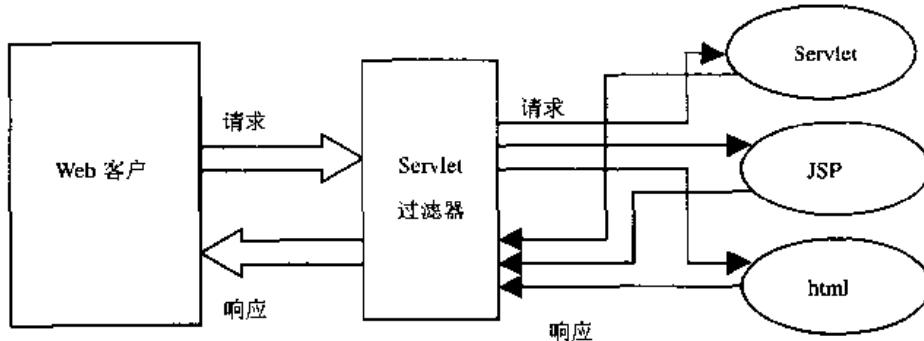


图 13-1 Servlet 过滤器的过滤过程

Servlet 过滤器具有以下特点：

- Servlet 过滤器可以检查和修改 HttpServletRequest 和 HttpServletResponse 对象
- 可以指定 Servlet 过滤器和特定的 URL 关联，只有当客户请求访问此 URL 时，才会触发过滤器工作
- Servlet 过滤器是 Java Servlet 规范 2.3 的一部分，因此所有实现 Java Servlet 规范 2.3 的 Servlet 容器都支持 Servlet 过滤器
- 独立的 Servlet 过滤器可以被串联在一起，形成管道效应，协同修改请求和响应对象

## 13.2 创建 Servlet 过滤器

所有的 Servlet 过滤器类都必须实现 javax.servlet.Filter 接口。这个接口含有 3 个过滤器类必须实现的方法：

- `init(FilterConfig)`: 这是 Servlet 过滤器的初始化方法，Servlet 容器创建 Servlet 过滤器实例后将调用这个方法。在这个方法中可以读取 web.xml 文件中 Servlet 过滤器的初始化参数
- `doFilter(ServletRequest, ServletResponse, FilterChain)`: 这个方法完成实际的过滤操作。当客户请求访问与过滤器关联的 URL 时，Servlet 容器将先调用过滤器的 `doFilter` 方法。`FilterChain` 参数用于访问后续过滤器
- `destroy()`: Servlet 容器在销毁过滤器实例前调用该方法，在这个方法中可以释放 Servlet 过滤器占用的资源

下面是一个过滤器的例子，这个名叫 NoteFilter 的过滤器可以拒绝列在黑名单上的客户访问留言簿，而且能将服务器响应客户请求所花的时间写入日志。例程 13-1 是 NoteFilter 的源代码。

例程 13-1 NoteFilter.java

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

public class NoteFilter implements Filter {
    private FilterConfig config = null;
    private String blackList=null;

    public void init(FilterConfig config) throws ServletException {
        this.config = config;
        blackList=config.getInitParameter("blacklist");
    }

    public void destroy() {
        config=null;
    }

    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException
    {
        String username =((HttpServletRequest) request).getParameter("username");
        if (username!=null )username=new String(username.getBytes("ISO-8859-1"),"GB2312");
        if (username!=null && username.indexOf(blackList) != -1 ) {
            response.setContentType("text/html;charset=GB2312");
        }
    }
}
```

```

PrintWriter out = response.getWriter();
out.println("<html><head></head><body>");
out.println("<h1>对不起,"+username + ",你没有权限留言 </h1>");
out.println("</body></html>");
out.flush();
return;
}

long before = System.currentTimeMillis();
config.getServletContext().log("NoteFilter:before call chain.doFilter()");
chain.doFilter(request, response);
config.getServletContext().log("NoteFilter:after call chain.doFilter()");
long after = System.currentTimeMillis();
String name = "";
if (request instanceof HttpServletRequest) {
    name = ((HttpServletRequest)request).getRequestURI();
}
config.getServletContext().log("NoteFilter:" +name + ":" + (after - before) + "ms");
}
}

```

当 NoteFilter 初始化时，它调用 config.getInitParameter("blacklist")方法，从 web.xml 文件中读取初始化参数 blacklist，这个参数表示被禁止访问留言簿的客户黑名单。

```

public void init(FilterConfig config) throws ServletException {
    this.config = config;
    blackList=config.getInitParameter("blacklist");
}

```

在 NoteFilter 的 doFilter()方法中首先从 request 对象中读取客户姓名，然后将客户姓名转换为中文字符编码。如果客户姓名中包含黑名单里的字符串，那么将直接向客户端返回一个拒绝网页。由于在这种情况下没有调用 chain.doFilter()方法，因此客户请求不会到达所访问的 Web 组件。

```

String username =((HttpServletRequest) request).getParameter("username");
if (username!=null )username=new String(username.getBytes("ISO-8859-1"),"GB2312");
if (username!=null && username.indexOf(blackList) != -1 ) {
    response.setContentType("text/html;charset=GB2312");
    PrintWriter out = response.getWriter();
    out.println("<html><head></head><body>");
    out.println("<h1>对不起,"+username + ",你没有权限留言 </h1>");
    out.println("</body></html>");
    out.flush();
    return;
}

```

假定姓名中包含“捣蛋鬼”字符串的客户将被禁止访问留言簿，并且留言簿由 NoteServlet 来实现(参见例程 13-2)，当名叫“捣蛋鬼 2000”的客户访问留言簿时，NoteFilter 的工作流程如图 13-2 所示。

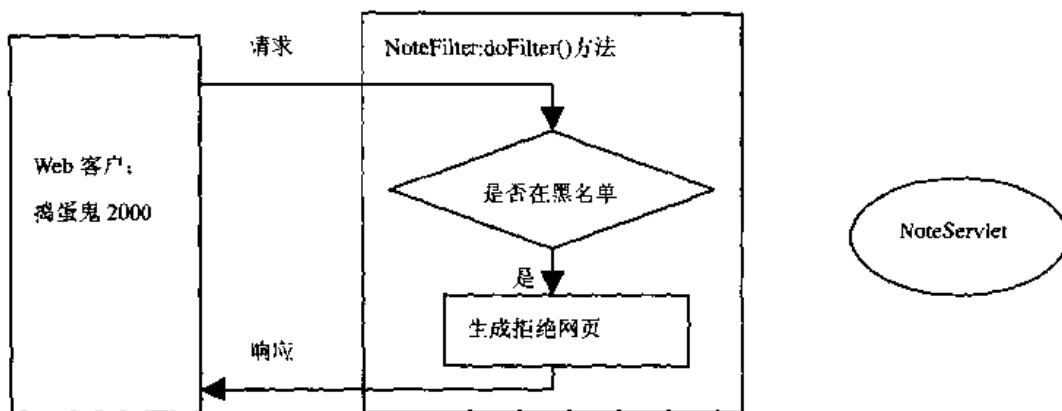


图 13-2 当客户请求被拒绝时的 NoteFilter 工作流程

如果客户名不在黑名单里，NoteFilter 的 doFilter 方法就会调用 chain.doFilter()方法，这个方法用于调用过滤器链中后续过滤器的 doFilter()方法。假如没有后续过滤器，那么就把客户请求传给相应的 Web 组件。在本例中，在调用 chain.doFilter()方法前后都生成了一些日志，并且记录了调用 chain.doFilter()方法前后的时间，从而计算出 Web 组件响应客户请求所花的时间。

```

long before = System.currentTimeMillis();
config.getServletContext().log("NoteFilter:before call chain.doFilter()");
chain.doFilter(request, response);
config.getServletContext().log("NoteFilter:after call chain.doFilter()");
long after = System.currentTimeMillis();
String name = "";
if (request instanceof HttpServletRequest) {
    name = ((HttpServletRequest)request).getRequestURI();
}
config.getServletContext().log("NoteFilter:" + name + ":" + (after - before) + "ms");
    
```

当名叫“小精灵”的客户访问留言簿时 NoteFilter 的工作流程如图 13-3 所示。

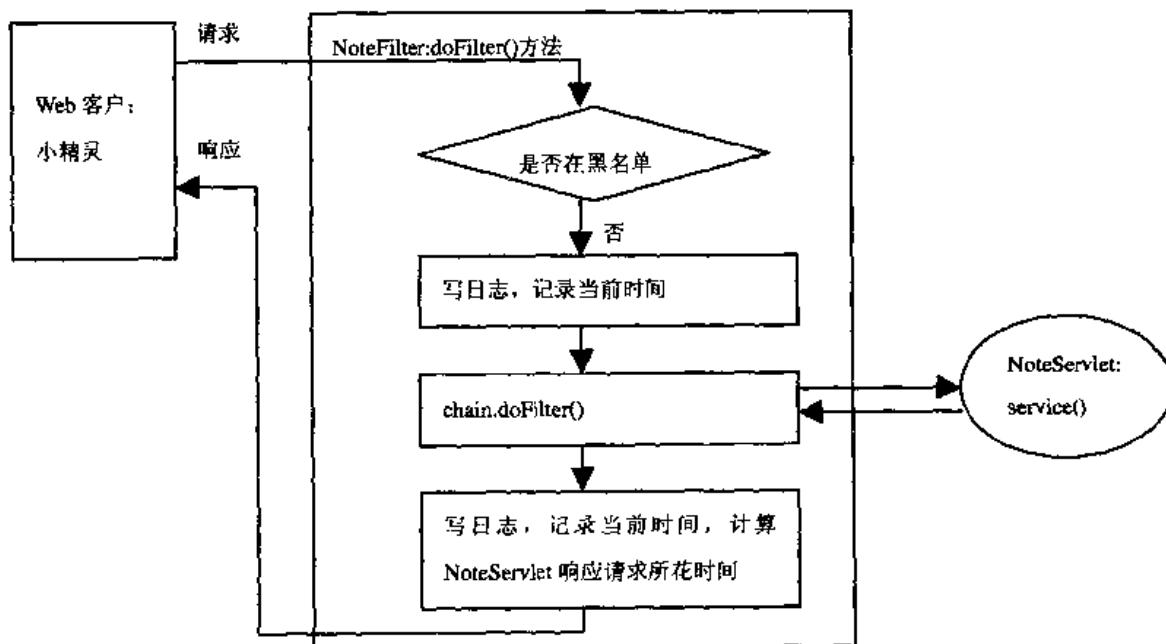


图 13-3 当客户请求被接受时的 NoteFilter 工作流程

### 13.3 发布 Servlet 过滤器

发布 Servlet 过滤器时，必须在 web.xml 文件中加入<filter>元素和<filter-mapping>元素。<filter>元素用来定义一个过滤器，如下所示：

```
<filter>
    <filter-name>NoteFilter</filter-name>
    <filter-class>NoteFilter</filter-class>
    <init-param>
        <param-name>blacklist</param-name>
        <param-value>捣蛋鬼</param-value>
    </init-param>
</filter>
```

以上代码中，<filter-name>属性指定过滤器的名字，<filter-class>指定过滤器的类名。<init-param>子元素为过滤器实例提供初始化参数，它包含一对参数名和参数值，在<filter>元素中可以包含多个<init-param>子元素。在这里定义了一个名为 blacklist 的参数，在 NoteFilter 中将读取这个参数，它表示禁止留言的客户黑名单。

<filter-mapping>元素用于将过滤器和 URL 关联，如下所示：

```
<filter-mapping>
    <filter-name>NoteFilter</filter-name>
    <url-pattern>/note</url-pattern>
</filter-mapping>
```

对于以上代码，当客户请求的 URL 和<url-pattern>指定的 URL（/note）匹配，将触发 NoteFilter 过滤器工作。这里的<filter-name>属性必须和<filter>元素中的<filter-name>属性一致。



如果希望 Servlet 过滤器过滤所有的 URL，可以把<url-pattern>的值设为“/\*”。

下面，创建一个 Servlet 类 NoteServlet，它实现一个简单的留言簿。它提供了一个表单，让客户输入姓名和留言，客户提交表单后，再将用户输入的信息显示在客户端的网页上。例程 13-2 是 NoteServlet 的源代码。

例程 13-2 NoteServlet.java

```
package mypack;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class NoteServlet extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=GB2312";
    public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException,
```

```

IOException {
    response.setContentType(CONTENT_TYPE);
    ServletOutputStream out = response.getOutputStream();
    out.println("<html>");
    out.println("<head><title>留言簿</title></head>");
    out.println("<body>");

    String username=request.getParameter("username");
    String content=request.getParameter("content");

    //转换为中文字符编码
    if(username!=null)username=new String(username.getBytes("ISO-8859-1"),"GB2312");
    if(content!=null)content=new String(content.getBytes("ISO-8859-1"),"GB2312");

    if(content!=null && !content.equals(""))
        out.println("<p>" +username+"的留言为: "+content+"</P>");

    out.println(" <FORM action='"+request.getContextPath()+" /note method=POST'>");

    out.println("<b>姓名:</b>");
    out.println("<input type=text size=10 name=username ><br>");
    out.println("<b>留言:</b><br>");
    out.println("<textarea name=content rows=5 cols=20 wrap></textarea><br>");
    out.println("<BR>");
    out.println("<input type=submit value=提交>");
    out.println("</form>");
    out.println("</body></html>");
}

public void destroy() {
}
}

```

假定把 NoteServlet 和 NoteFilter 发布到 helloapp 应用中，下面是发布和运行 NoteServlet 和 NoteFilter 的步骤。



(1) 编译 NoteServlet 和 NoteFilter 类，在编译时，需要将 Java Servlet API 的 JAR 文件 servlet-api.jar 设置为 classpath，servlet-api.jar 文件，位于<CATALINA\_HOME>/common/lib 目录下。编译生成的类的存放位置如下：

```

<CATALINA_HOME>/webapps/helloapp/WEB_INF/classes/mypack/NoteServlet.class
<CATALINA_HOME>/webapps/helloapp/WEB_INF/classes/NoteFilter.class

```

(2) 在 web.xml 中配置 NoteServlet 和 NoteFilter，应该先声明过滤器元素，再声明 Servlet 元素。此外，由于在 web.xml 中包含了中文字符“捣蛋鬼”，因此，应该把 web.xml 的字符编码设为 GB2312。以下是 web.xml 的代码：

```

<?xml version="1.0" encoding="GB2312"?>

<!DOCTYPE web-app PUBLIC
'-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
'http://java.sun.com/j2ee/dtds/web-app_2_3.dtd'>

<web-app>

    <filter>
        <filter-name>NoteFilter</filter-name>
        <filter-class>NoteFilter</filter-class>
        <init-param>
            <param-name>blacklist</param-name>
            <param-value>捣蛋鬼</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>NoteFilter</filter-name>
        <url-pattern>/note</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>NoteServlet</servlet-name>
        <servlet-class>mypack.NoteServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>NoteServlet</servlet-name>
        <url-pattern>/note</url-pattern>
    </servlet-mapping>

</web-app>

```

(3) 为了观察 NoteFilter 生成的日志，应该确保在 server.xml 的 localhost 对应的<host>元素中已经配置了如下<logger>元素：

```

<Logger className="org.apache.catalina.logger.FileLogger"
       directory="logs" prefix="localhost_log." suffix=".txt"
       timestamp="true"/>

```

(4) 启动 Tomcat 服务器，访问 <http://localhost:8080/helloapp/note>，将会显示留言簿表单，如图 13-4 所示。

此时，在<CATALINA\_HOME>/logs/localhost\_log.2003-08-25.txt 文件中生成如下日志：

```

2003-08-25 20:54:41 NoteFilter:before call chain.doFilter()
2003-08-25 20:54:41 NoteFilter:after call chain.doFilter()
2003-08-25 20:54:41 NoteFilter:/helloapp/note: 20ms

```

最后一行日志表明 Servlet 容器调用 NoteServlet 的 service 方法花了 20ms。



图 13-4 NoteServlet 的网页

(5) 在留言簿中输入姓名和留言，确保姓名不在黑名单中，如图 13-5 所示。然后单击【提交】按钮。



图 13-5 在留言簿中输入合法信息

NoteServlet 返回的网页如图 13-6 所示。



图 13-6 在留言簿中输入合法信息后的返回页面

(6) 在留言簿中输入客户名“捣蛋鬼 2000”并留言，如图 13-7 所示，然后单击【提交】按钮。

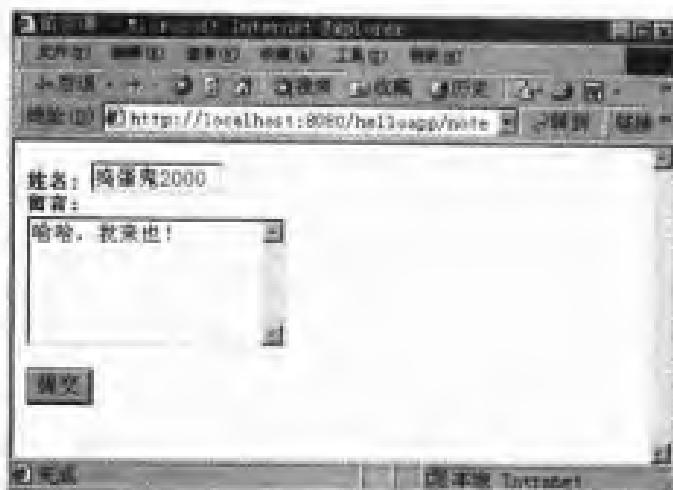


图 13-7 在留言簿中输入的客户姓名在黑名单中

此时，由于客户姓名在黑名单中，NoteFilter 将屏蔽客户的请求，返回的网页如图 13-8 所示。

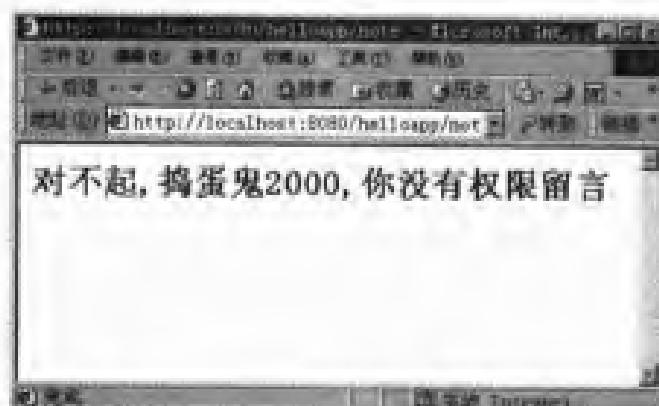


图 13-8 拒绝客户访问留言簿时的返回页面

## 13.4 串联 Servlet 过滤器

多个 Servlet 过滤器可以串联起来协同工作。Servlet 容器将根据它们在 web.xml 中定义的先后顺序，依次调用它们的 doFilter()方法。假定有两个 Servlet 过滤器串联起来，它们的 doFilter()方法均采用如下结构：

```
Code1: //表示调用 chain.doFilter()前的代码  
chain.doFilter();
```

```
Code2: //表示调用 chain.doFilter()后的代码
```

当客户请求访问与过滤器关联的 Servlet 时，这两个过滤器以及 Servlet 的工作流程如图 13-9 所示。

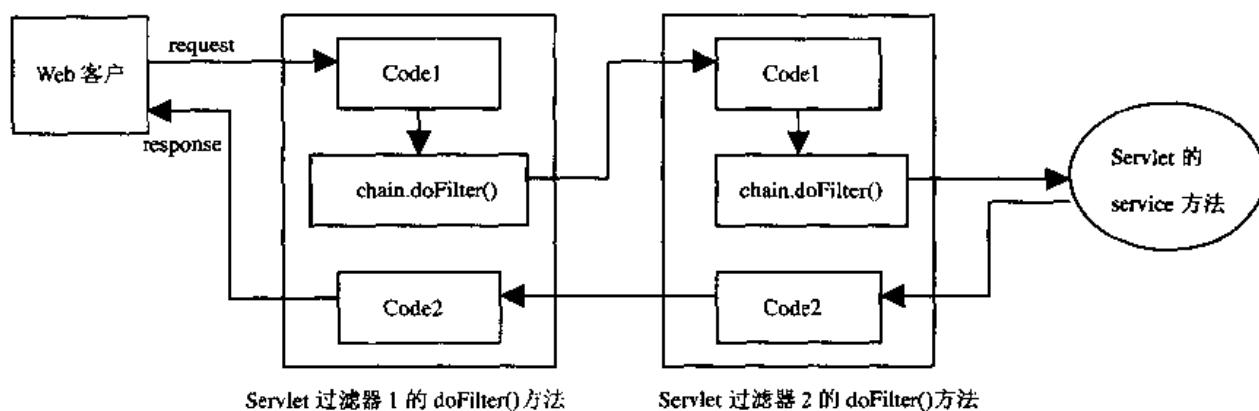


图 13-9 串联 Servlet 过滤器的工作流程

在 13.2 节，我们已经创建了一个 NoteFilter 过滤器，下面将创建第二个 Servlet 过滤器，名为 ReplaceTextFilter，它能够修改 NoteServlet 的响应结果，将新的字符串替换网页中特定的旧字符串，然后将修改后的网页返回给客户。为了实现这一过滤功能，一共创建了 3 个类：

- ReplaceTextStream：用户自定义的 ServletOutputStream，具有替换字符串的功能
- ReplaceTextWrapper：用户自定义的 HttpServletResponseWrapper
- ReplaceTextFilter：能够对输出网页内容进行字符串替换的 Servlet 过滤器

### 13.4.1 用户自定义的 ServletOutputStream

ReplaceTextStream 类是 ServletOutputStream 的子类，它封装了 Servlet 容器提供的 ServletOutputStream 对象，并且创建了一个 ByteArrayOutputStream，它是一个字节数组缓存。在构造 ReplaceTextStream 类的实例时，应该把 Servlet 容器提供的 ServletOutputStream 对象作为参数传给 ReplaceTextStream 对象：

```

private OutputStream intStream;
private ByteArrayOutputStream baStream;
private boolean closed = false;

private String oldStr;
private String newStr;

public ReplaceTextStream(OutputStream outStream,
                        String searchStr,
                        String replaceStr) {
    intStream = outStream;
    baStream = new ByteArrayOutputStream();
    oldStr = searchStr;
    newStr = replaceStr;
}

```

ReplaceTextStream 类的 `println(String s)` 方法把数据写到 `ByteArrayOutputStream`:

```
public void println(String s) throws IOException {
    s = s + "\n";
    byte[] bs = s.getBytes();
    baStream.write(bs);
}
```

ReplaceTextStream 的 `close()` 和 `flush()` 方法中均调用 `processStream()` 方法:

```
public void close() throws java.io.IOException {
    if (!closed) {
        processStream();
        intStream.close();
        closed = true;
    }
}

public void flush() throws java.io.IOException {
    if (baStream.size() != 0) {
        if (!closed) {
            processStream(); // need to synchronize the flush!
            baStream = new ByteArrayOutputStream();
        }
    }
}
```

在 `processStream` 方法中先对 `ByteArrayOutputStream` 的数据进行字符串替换, 然后将替换后的数据全部写到 `ServletOutputStream` 中。字符串替换操作是在方法 `replaceContent` 中完成的。

```
public void processStream() throws java.io.IOException {
    intStream.write(replaceContent(baStream.toByteArray()));
    intStream.flush();
}
```

`replaceContent` 方法负责替换字节数组中的字符串:

```
public byte[] replaceContent(byte[] inBytes) {

    String retVal = "";
    String firstPart = "";

    String tpString = new String(inBytes);
    String srchString = (new String(inBytes)).toLowerCase();

    int endBody = srchString.indexOf(oldStr);

    if (endBody != -1) {
        firstPart = tpString.substring(0, endBody);
        retVal = firstPart + newStr +
            tpString.substring(endBody + oldStr.length());
    }
}
```

```

    } else {
        retVal=tpString;
    }

    return retVal.getBytes();
}

```

ReplaceTextStream 类实现对输出流的字符串替换的流程如图 13-10 所示。

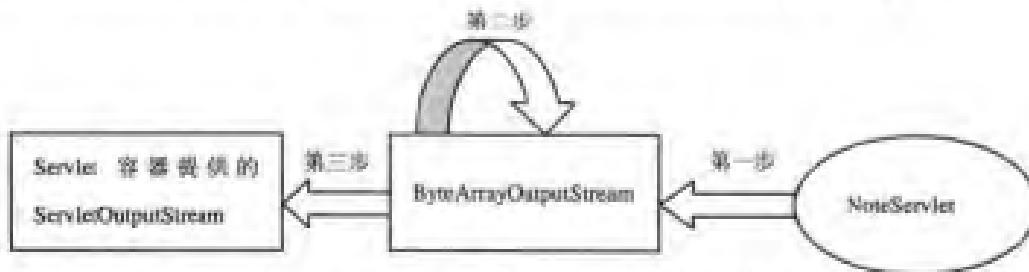


图 13-10 ReplaceTextStream 类替换输出流字符串的流程

图 13-10 中的每一步操作的说明参见表 13-1。

表 13-1 ReplaceTextStream 类替换输出流字符串的流程

步 骤	功 能	描 述
第一歩	把响应结果输出到缓存中	NoteServlet 调用 ReplaceTextStream 的 printin(String)方法，将所有的响应数据写到 ByteArrayOutputStream 中。
第二步	在缓存中替换字符串	ReplaceTextFilter 调用 ReplaceTextStream 的 close 方法，close 方法又调用 processStream 方法，在 processStream 方法又调用 replaceContent 方法替换 ByteArrayOutputStream 中的字符串。
第三步	把响应结果输出到真正的 Servlet 输出流中	在 processStream 方法中将替换后的 ByteArrayOutputStream 的数据写到 Servlet 容器提供的 ServletOutputStream 对象中。

例程 13-3 是 ReplaceTextStream 的源程序。

例程 13-3 ReplaceTextStream.java

---

```

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

class ReplaceTextStream extends ServletOutputStream {
    private OutputStream inStream;
    private ByteArrayOutputStream baStream;
    private boolean closed = false;
    private String oldStr;
    private String newStr;

    public ReplaceTextStream(OutputStream outStream, String searchStr,
                           String replaceStr) {

```

```
intStream = outStream;
baStream = new ByteArrayOutputStream();
oldStr = searchStr;
newStr = replaceStr;
}
public void write(int a) throws IOException{
    baStream.write(a);
}
public void println(String s) throws IOException{
    s=s+"\n";
    byte[] bs=s.getBytes();
    baStream.write(bs);
}
public void close() throws java.io.IOException {
    if (!closed) {
        processStream();
        intStream.close();
        closed = true;
    }
}
public void flush() throws java.io.IOException {
    if (baStream.size() != 0) {
        if (!closed) {
            processStream(); // need to synchronize the flush!
            baStream = new ByteArrayOutputStream();
        }
    }
}
public void processStream() throws java.io.IOException {
    intStream.write(replaceContent(baStream.toByteArray()));
    intStream.flush();
}
public byte[] replaceContent(byte[] inBytes) {
    String retVal="";
    String firstPart="";
    String tpString = new String(inBytes);
    String srchString = (new String(inBytes)).toLowerCase();
    int endBody = srchString.indexOf(oldStr);

    if (endBody != -1) {
        firstPart = tpString.substring(0, endBody);
        retVal = firstPart + newStr +
            tpString.substring(endBody + oldStr.length());
    } else {
        retVal=tpString;
    }
}
```

```
        }
        return retVal.getBytes();
    }
}
```

### 13.4.2 用户自定义的 HttpServletResponseWrapper

ReplaceTextWrapper 是 HttpServletResponseWrapper 的子类。HttpServletResponseWrapper 类位于 javax.servlet.http 包中，它实现了 HttpServletResponse 接口，可用来重新包装 HttpServletResponse 对象。

ReplaceTextWrapper 中封装了 Servlet 容器提供的 ServletResponse 对象，并且在构造方法中创建了一个 ReplaceTextStream 对象。ReplaceTextWrapper 为 NoteServlet 提供了 getOutputStream 方法，该方法返回用户自定义的 ReplaceTextStream 对象，而不是 Servlet 容器提供的 ServletOutputStream 对象，这样就可以保证 NoteServlet 调用 println 方法生成响应数据时，这些数据首先写到 ReplaceTextStream 的 ByteArrayOutputStream 中。以下是 ReplaceTextWrapper 的源程序：

```
class ReplaceTextWrapper extends HttpServletResponseWrapper {

    private ReplaceTextStream tpStream;
    public ReplaceTextWrapper(ServletResponse inResp, String searchText,
                             String replaceText) throws java.io.IOException
    {
        super((HttpServletResponse) inResp);
        tpStream = new ReplaceTextStream(inResp.getOutputStream(),
                                         searchText, replaceText);
    }

    public ServletOutputStream getOutputStream() throws java.io.IOException {
        return tpStream;
    }
}
```

### 13.4.3 创建对输出网页进行字符串替换的过滤器

ReplaceTextFilter 在初始化方法中调用 config.getInitParameter("search")方法读取被替换的字符串，调用 config.getInitParameter("replace")方法读取替换后的字符串。

```
private FilterConfig config = null;
private String searchStr=null;
private String replaceStr=null;
public void init(FilterConfig config) throws ServletException {
    this.config = config;
    searchStr=config.getInitParameter("search");
    replaceStr=config.getInitParameter("replace");
```

}

ReplaceTextFilter 在 doFilter 方法中首先创建了一个 ReplaceTextWrapper 对象，然后调用 chain.doFilter(request, myWrappedResp)方法，将 ReplaceTextWrapper 对象传给后续的 Servlet 过滤器或者 Servlet，在本例中后续组件为 NoteServlet。这样，当 NoteServlet 调用 response.getOutputStream() 方法来获取 ServletOutputStream 对象时，其实得到的是 ReplaceTextStream 对象。

在 doFilter 方法中，在调用 chain.doFilter(request, myWrappedResp)前后，都生成了一些日志，用于跟踪 doFilter()方法的执行流程。

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    ReplaceTextWrapper myWrappedResp = new ReplaceTextWrapper(response,
        searchStr, replaceStr);
    config.getServletContext().log("ReplaceTextFilter:before call chain.doFilter()");
    chain.doFilter(request, myWrappedResp);
    config.getServletContext().log("ReplaceTextFilter:after call chain.doFilter()");
    myWrappedResp.getOutputStream().close();
}
```

例程 13-4 是 ReplaceTextFilter 的源程序。

例程 13-4 ReplaceTextFilter.java

---

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

public class ReplaceTextFilter implements Filter {

    private FilterConfig config = null;
    private String searchStr=null;
    private String replaceStr=null;
    public void init(FilterConfig config) throws ServletException {
        this.config = config;
        searchStr=config.getInitParameter("search");
        replaceStr=config.getInitParameter("replace");
    }
    public void destroy() {
        config = null;
    }
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        ReplaceTextWrapper myWrappedResp = new ReplaceTextWrapper(response,
            searchStr, replaceStr);
        config.getServletContext().log("ReplaceTextFilter:before call chain.doFilter()");
        chain.doFilter(request, myWrappedResp);
        config.getServletContext().log("ReplaceTextFilter:after call chain.doFilter()");
    }
}
```

```

        myWrappedResp.getOutputStream().close();
    }
}

```

#### 13.4.4 ReplaceTextFilter 过滤器的 UML 时序图

在 NoteServlet 的 service 方法中, 从 HttpServletResponse 对象中获取 ServletOutputStream 对象, 然后调用 ServletOutputStream 对象的 println 方法生成响应数据:

```

public void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException,
IOException {
    response.setContentType(CONTENT_TYPE);
    ServletOutputStream out = response.getOutputStream();
    out.println("<html>");
    out.println("<head><title>留言簿</title></head>");
    out.println("<body>");
    .....
}

```

如果没有安装 ReplaceTextFilter 过滤器, NoteServlet 的 service 方法将从参数中获取 Servlet 容器提供的 HttpServletResponse 对象, 因此, response.getOutputStream 方法返回 Servlet 容器提供的 ServletOutputStream 对象。通过这个流输出数据时, 无法修改已输出的数据内容。在没有安装 ReplaceTextFilter 过滤器的情况下, NoteServlet 的 service 方法输出响应数据的 UML 时序图, 如图 13-11 所示。

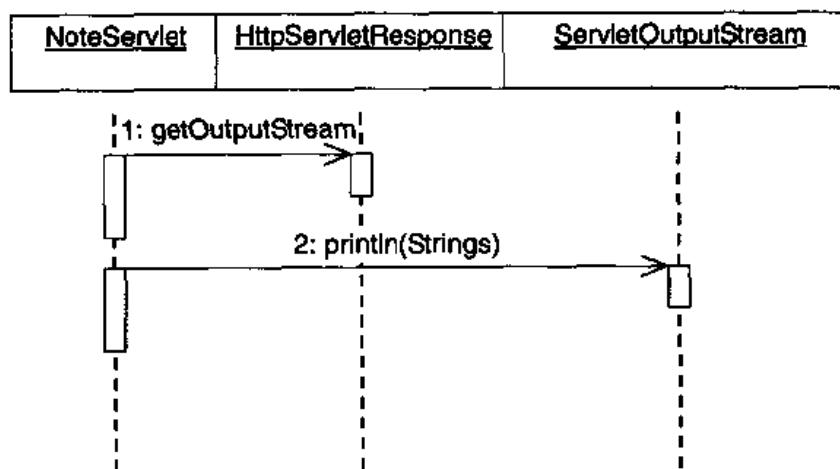


图 13-11 NoteServlet 输出响应数据的 UML 时序图（无 ReplaceTextFilter 时）

如果安装了 ReplaceTextFilter 过滤器, ReplaceTextFilter 把一个 ReplaceTextWrapper 对象作为参数传给 NoteServlet 的 service 方法。这样, response.getOutputStream 方法将返回 ReplaceTextStream 对象。通过这个用户自定义的流输出数据时, 可以修改已输出的数据内容。加入了 ReplaceTextFilter 后, NoteServlet 的 service 方法输出响应数据的 UML 时序图, 如图 13-12 所示。

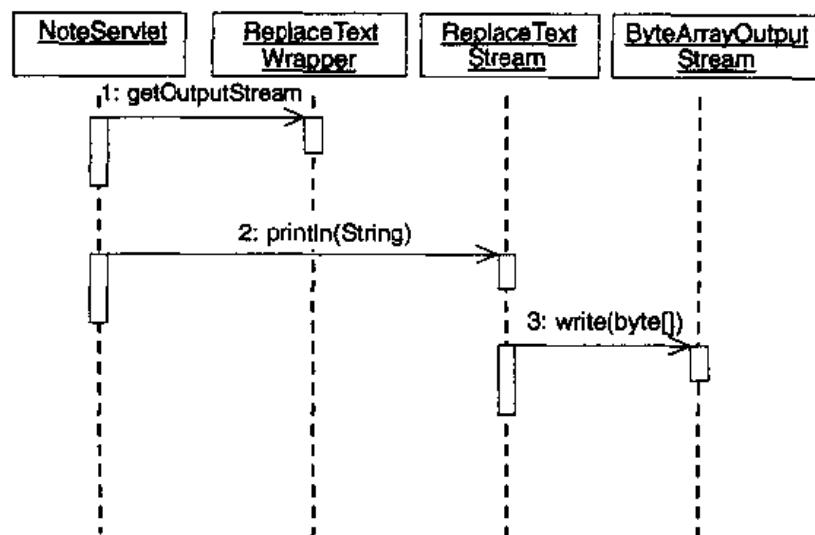


图 13-12 NoteServlet 输出响应数据的 UML 时序图（加入 ReplaceTextFilter 后）

在 ReplaceTextFilter 的 doFilter 方法中，最后一行代码为关闭 ReplaceTextStream 流，ReplaceTextFilter 关闭 ReplaceTextStream 的 UML 时序图，如图 13-13 所示。

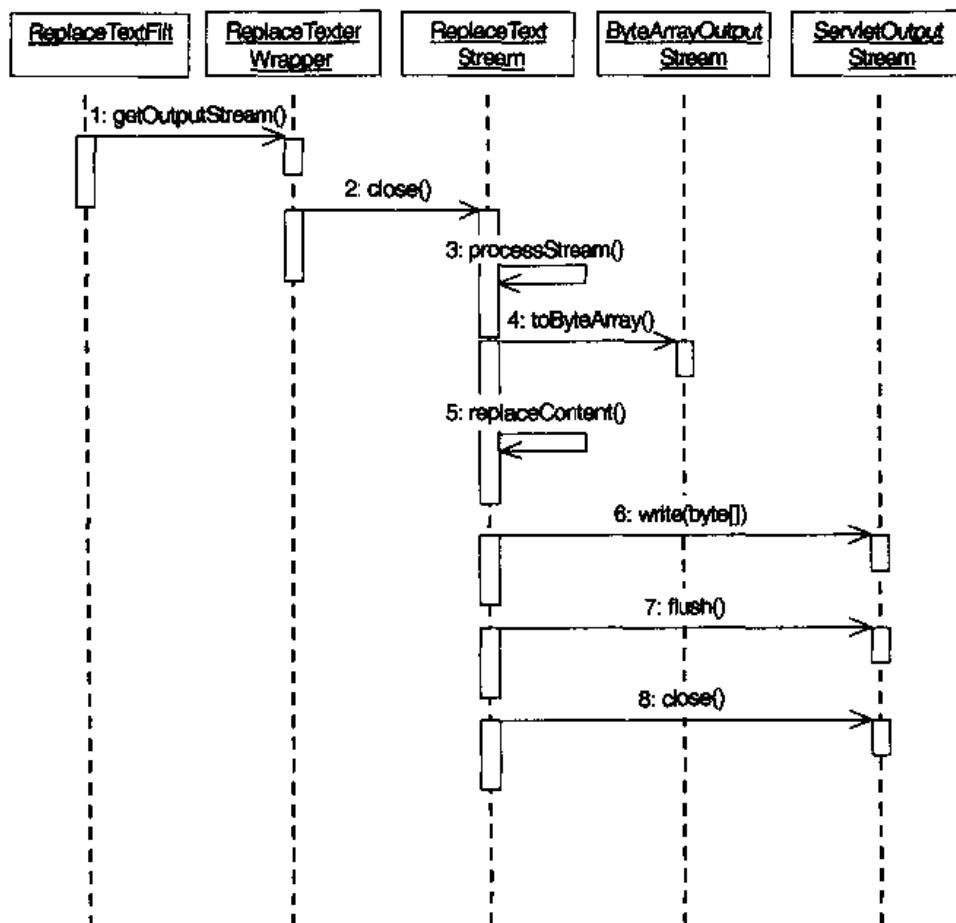


图 13-13 ReplaceTextFilter 关闭 ReplaceTextStream 的 UML 时序图

### 13.4.5 发布和运行包含 ReplaceTextFilter 过滤器的 Web 应用

假定把 ReplaceTextFilter 发布到 helloapp 应用中，以下是发布和运行 ReplaceTextFilter 的步骤。



(1) 编译 ReplaceTextStream、ReplaceTextWrapper 和 ReplaceTextFilter 类，在编译时，需要将 Java Servlet API 的 JAR 文件 servlet-api.jar 设置为 classpath，servlet-api.jar 文件位于 <CATALINA\_HOME>/common/lib 目录下。它们编译生成的类的存放位置如下：

```
<CATALINA_HOME>/webapps/helloapp/WEB_INF/classes/ ReplaceTextStream.class  
<CATALINA_HOME>/webapps/helloapp/WEB_INF/classes/ ReplaceTextWrapper.class  
<CATALINA_HOME>/webapps/helloapp/WEB_INF/classes/ ReplaceTextFilter.class
```

(2) 在 web.xml 中配置 ReplaceTextFilter，ReplaceTextFilter 的定义应该在 NoteFilter 定义之后。此外，由于在 web.xml 中包含了中文字符，因此，应该把 web.xml 的字符编码设为 GB2312。以下是 web.xml 的代码：

```
<?xml version="1.0" encoding="GB2312"?>  
  
<!DOCTYPE web-app PUBLIC  
        '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'  
        'http://java.sun.com/j2ee/dtds/web-app_2_3.dtd'>  
  
<web-app>  
  
    <filter>  
        <filter-name>NoteFilter</filter-name>  
        <filter-class>NoteFilter</filter-class>  
        <init-param>  
            <param-name>blacklist</param-name>  
            <param-value>捣蛋鬼</param-value>  
        </init-param>  
    </filter>  
  
    <filter-mapping>  
        <filter-name>NoteFilter</filter-name>  
        <url-pattern>/note</url-pattern>  
    </filter-mapping>  
  
    <filter>  
        <filter-name>ReplaceTextFilter</filter-name>  
        <filter-class>ReplaceTextFilter</filter-class>  
        <init-param>  
            <param-name>search</param-name>  
            <param-value>暴力</param-value>  
        </init-param>  
    </filter>
```

```

</init-param>
<init-param>
    <param-name>replace</param-name>
    <param-value>和平</param-value>
</init-param>
</filter>

<filter-mapping>
    <filter-name>ReplaceTextFilter</filter-name>
    <url-pattern>/note</url-pattern>
</filter-mapping>

<servlet>
    <servlet-name>NoteServlet</servlet-name>
    <servlet-class>mypack.NoteServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>NoteServlet</servlet-name>
    <url-pattern>/note</url-pattern>
</servlet-mapping>

</web-app>

```

以上配置为 ReplaceTextFilter 定义了两个初始化参数 search 和 replace，它们的参数值分别为“暴力”和“和平”。因此，当客户留言中包含“暴力”字符串，就会被 ReplaceTextFilter 过滤器替换为“和平”字符串。

(3) 为了观察 ReplaceTextFilter 生成的日志，应该确保在 server.xml 的 localhost 对应的<host>元素中已经配置了如下<logger>元素：

```

<Logger className="org.apache.catalina.logger.FileLogger"
        directory="logs" prefix="localhost_log." suffix=".txt"
        timestamp="true"/>

```

(4) 启动 Tomcat 服务器，访问 <http://localhost:8080/helloapp/note>，将会看到一个留言簿表单，如图 13-14 所示。

在<CATALINA\_HOME>/logs/localhost\_log.2003-08-25.txt 文件中生成如下日志：

```

2003-08-25 23:07:30 NoteFilter:before call chain.doFilter()
2003-08-25 23:07:30 ReplaceTextFilter:before call chain.doFilter()
2003-08-25 23:07:30 ReplaceTextFilter:after call chain.doFilter()
2003-08-25 23:07:30 NoteFilter:after call chain.doFilter()
2003-08-25 23:07:30 NoteFilter:/helloapp/note: 60ms

```

根据日志，可以看出 Servlet 容器先调用 NoteFilter 的 doFilter 方法，再调用 ReplaceTextFilter 的 doFilter 方法。当 ReplaceTextFilter 的 doFilter 方法执行完毕，流程将回到 NoteFilter 的 doFilter 方法中。

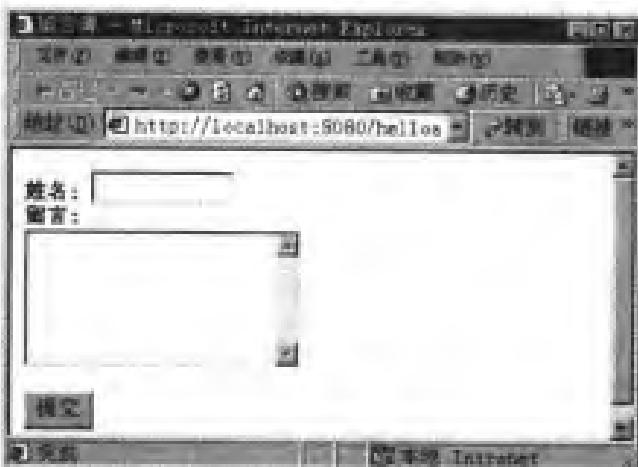


图 13-14 NoteServlet 的网页

(5) 在留言簿中输入用户名“捣蛋鬼 2000”，然后再留言，如图 13-15 所示，然后单击【提交】按钮。



图 13-15 在留言簿中输入的客户姓名在黑名单中

此时，由于用户姓名在黑名单中，NoteFilter 将屏蔽客户的请求，直接返回拒绝网页，如图 13-16 所示。

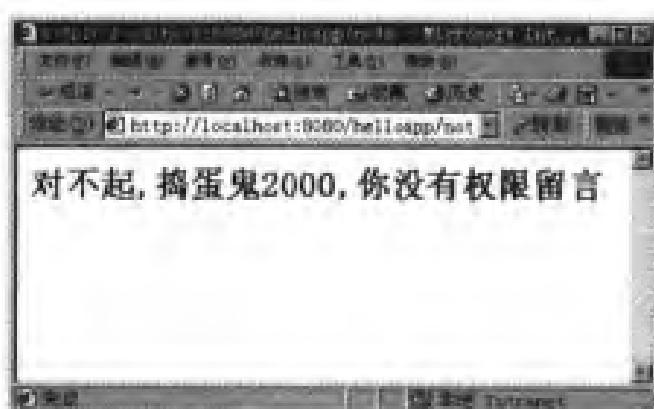


图 13-16 NoteFilter 拒绝客户访问留言簿时的返回页面

此时，在<CATALINA\_HOME>/logs/localhost\_log.2003-08-25.txt 文件中没有生成任何日志。在 NoteFilter 中检查出客户在黑名单中后，就结束客户请求，因此 ReplaceTextFilter 的 doFilter 方法没有被调用。

(6) 在留言簿中输入用户名“匆匆过客”，在留言中输入包含“暴力”的字符串，如图 13-17 所示，然后单击【提交】按钮。

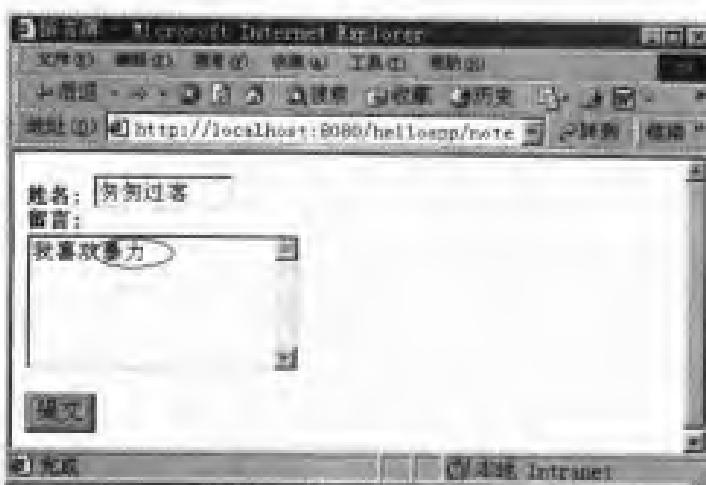


图 13-17 在留言簿中输入的留言包含被替换字符串

此时，由于客户提交的留言中包含“暴力”字符串，它将被 ReplaceTextFilter 替换为“和平”字符串，返回的网页如图 13-18 所示。

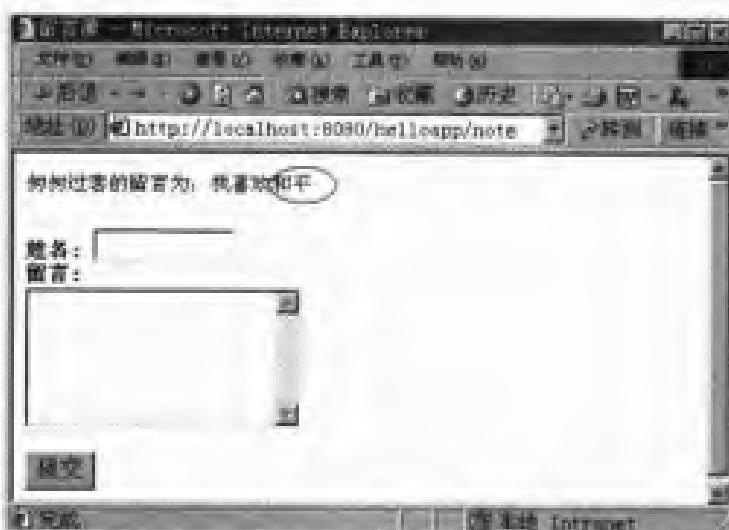


图 13-18 被 ReplaceTextFilter 替换字符串后返回的网页

在本书配套光盘的 sourcecode/chapter13/helloapp 目录下提供了包含以上所有文件的 helloapp 应用，如果要发布这个应用，只要把整个 helloapp 应用拷贝到 <CATALINA\_HOME>/webapps 目录下即可。

## 13.5 小 结

Servlet 过滤器是在 Java Servlet 规范 2.3 中定义的，它能够对 Servlet 容器的请求和响应对象进行检查和修改。所有实现 Java Servlet 规范 2.3 的 Servlet 容器都支持 Servlet 过滤器。本章介绍了 Servlet 过滤器的创建和发布方法，并且介绍了 Servlet 过滤器串联起来的工作流程。同时介绍了一个具有实际用途的 Servlet 过滤器例子 ReplaceTextFilter，它能够对 Servlet 的响应结果进行字符串替换。如果在实际应用中，希望对 Servlet 的响应结果进行特定的格式或数据转换，均可采用这种编程模型。

# 第 14 章 自定义 JSP 标签

JSP Tag Library 技术是在 JSP 1.1 版本中才出现的，它支持用户在 JSP 文件中自定义标签，这样可以使 JSP 代码更加简洁。这些可重用的标签能处理复杂的逻辑运算和事务，或者定义 JSP 网页的输出内容和格式。本章以 helloapp 应用为例，介绍自定义标签的创建过程以及在 Web 应用中的使用方法。

## 14.1 自定义 JSP 标签简介

下面将以 helloapp 应用为例，介绍如何创建和使用自定义 JSP 标签。这里将创建一个能替换 helloapp 应用中 JSP 网页的静态文本的标签，这个标签名为 message，它放在 mytaglib 标签库中。

如果在 hello.jsp 文件中使用 message 标签，先在 JSP 文件中引入这个标签库，如下所示：

```
<%@ taglib uri="/mytaglib" prefix="mm" %>
```

然后用<mm:message>标签替换 hello.jsp 中静态文本 Hello，如下所示：

修改前：

```
<b>Hello : <%= request.getAttribute("USER") %></b>
```

修改后：

```
<b><mm:message key="hello.hello" /> : <%= request.getAttribute("USER") %></b>
```

当客户访问 hello.jsp 网页时，message 标签的处理类会根据属性 key 的值从一个文本文档中找到与 key 匹配的字符串。假定这个字符串为 Hello，然后将这个字符串输出到网页上，因此对客户端来说，看到的网页界面和没有使用 message 标签的界面一样。对开发人员来说，采用 message 标签可以使 JSP 网页变得简洁并且易于维护。试想如果 JSP 网页的静态显示内容发生需求变更，网页维护人员无需修改这些 JSP 网页，只要修改一个存放网页静态文本的文件即可。

采用 message 标签，还可以方便地实现同一个 JSP 文件支持多种语言版本。message 标签可以根据客户选择的语言来输出相应的静态文本。使用不同语言的客户访问 hello.jsp 文件时，message 标签的处理过程如图 14-1 所示。

制作一个完整的 Tag Library 程序，包含 3 个步骤。



- (1) 创建标签的处理类 (Tag Handler Class)。
- (2) 创建标签库描述文件 (Tag Library Descriptor File)。
- (3) 在 JSP 文件中引入标签库。

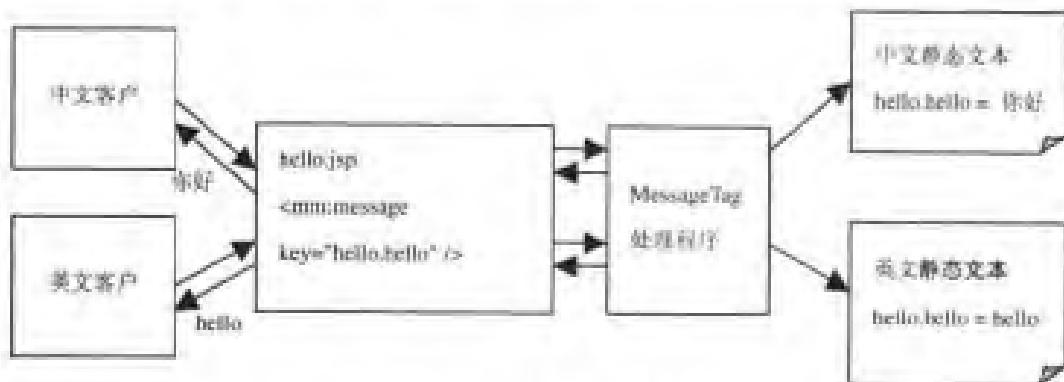


图 14-1 message 标签支持多种语言

## 14.2 创建标签处理类

JSP 容器编译 JSP 网页时，如果遇到自定义标签，就会调用这个标签的处理类（Tag Handler Class）。处理标签的类必须扩展 javax.servlet.jsp.TagSupport 类或者 javax.servlet.jsp.BodyTagSupport。本书介绍 TagSupport 类。

### 14.2.1 JSP Tag API

TagSupport 类的主要方法参见表 14-1。

表 14-1 TagSupport 类的主要方法

方 法	属 性
doStartTag	JSP 容器遇到自定义标签的起始标志时调用该方法
doEndTag	JSP 容器遇到自定义标签的结束标志时调用该方法
setValue(String k, Object v)	在标签处理类中设置 key/value
getValue(String k)	在标签处理类中根据参数 key 返回匹配的 value
removeValue(String k)	在标签处理类中删除 key/value
setPageContext(PageContext pc)	设置 PageContext 对象。该方法由 JSP 容器在调用 doStartTag 或 doEndTag 方法前调用
setParent(Tag t)	设置嵌套了当前标签的上层标签的处理类。该方法由 JSP 容器在调用 doStartTag 或 doEndTag 方法前调用
getParent()	返回嵌套了当前标签的上层标签的处理类

#### 1. parent 和 pageContext 属性

TagSupport 类有两个重要的属性：

- parent：代表嵌套了当前标签的上层标签的处理类
- pageContext：代表 Web 应用中的 javax.servlet.jsp.PageContext 对象

JSP 容器在调用 doStartTag 或 doEndTag 方法前，会先调用 setPageContext 和 setParent 方法，设置 pageContext 和 parent。在 doStartTag 或 doEndTag 方法中可以通过 getParent 方

法获取上层标签的处理类；在 TagSupport 类中定义了 protected 类型的 pageContext 成员变量，因此在标签处理类中可以直接访问 pageContext 变量。PageContext 类提供了保存和访问 Web 应用的共享数据的方法：

```
public void setAttribute(String name, Object value, int scope)
public Object getAttribute(String name, int scope)
```

在本章以及下一章的例子中，都会用到 PageContext 的以上方法。其中，scope 参数用来指定属性存在的范围，它的可选值包括：

- PageContext.PAGE\_SCOPE
- PageContext.REQUEST\_SCOPE
- PageContext.SESSION\_SCOPE
- PageContext.APPLICATION\_SCOPE



**提示** 在 TagSupport 的构造方法中不能访问 pageContext 成员变量，因为此时 JSP 容器还没有调用 setPageContext 方法对 pageContext 进行初始化。

## 2. 处理标签的方法

TagSupport 类提供了两个处理标签的方法：

```
public int doStartTag() throws JspException
public int doEndTag() throws JspException
```

当 JSP 容器遇到自定义标签的起始标志，就会调用 doStartTag()方法。doStartTag()方法返回一个整数值，用来决定程序的后续流程。它有两个可选值：Tag.SKIP\_BODY 和 Tag.EVAL\_BODY\_INCLUDE。

Tag.SKIP\_BODY 表示标签之间的内容被忽略。Tag.EVAL\_BODY\_INCLUDE 表示标签之间的内容被正常执行。例如对于以下代码：

```
<prefix: Mytag>
Hello Tag Library
.....
.....
</prefix:Mytag>
```

假若<Mytag>的 doStartTag()方法返回 Tag.SKIP\_BODY，Hello Tag Library 字符串不会显示在网页上；若返回 Tag.EVAL\_BODY\_INCLUDE，Hello Tag Library 字符串将显示在网页上。

当 JSP 容器遇到自定义标签的结束标志，就会调用 doEndTag()方法。doEndTag()方法也返回一个整数值，用来决定程序后续流程。它有两个可选值：Tag.SKIP\_PAGE 和 Tag.EVAL\_PAGE。Tag.SKIP\_PAGE 表示立刻停止执行 JSP 网页，网页上未处理的静态内容和 JSP 程序均被忽略，任何已有的输出内容立刻返回到客户的浏览器上。Tag.EVAL\_PAGE 表示按正常的流程继续执行 JSP 网页。

## 3. 用户自定义的标签属性

如果在标签中还包含自定义的属性，例如：

```
<prefix:Mytag attribute1="value1">
```

```
.....  
.....  
</prefix:Mytag>
```

那么在标签处理类中应该将这个属性作为成员变量，并且分别提供设置和读取属性的方法，假定以上 attribute1 为 int 类型，可以定义如下方法：

```
private int attribute1;  
public void setAttribute1(int value){  
    this.attribute1=value;  
}  
public int getAttribute1(){  
    return attribute1;  
}
```

## 14.2.2 创建标签处理类范例程序

下面将创建 message 标签的处理类 MessageTag.java。

### 1. 创建包含 JSP 网页静态文本的文件

首先将创建包含 JSP 网页静态文本的文件，这些文本以 key/value 的形式存放。在 messageresource.properties 文件中存放英文静态文本，在 messageresource\_ch.properties 文件中存放中文静态文本。这两个文件都存放于 helloapp 应用的 WEB-INF 目录下。以下是这两个文件的内容：

messageresource.properties 文件：

```
hello.title = helloapp  
hello.hello = Hello  
login.title = helloapp  
login.user = User Name  
login.password = Password  
login.submit = Submit
```

messageresource\_ch.properties 文件：

```
hello.title = helloapp 的 hello 页面  
hello.hello = 你好  
login.title = helloapp 的登录页面  
login.user = 用户名  
login.password = 口令  
login.submit = 提交
```

### 2. 在 Web 应用启动时装载静态文本

尽管装载静态文本的任务可以直接由标签处理类来完成，但是把初始化的操作安排在 Web 应用启动时完成，则更符合 Web 编程的规范。在本例中，将修改 DispatcherServlet 类的 init 方法，这个方法负责从 messageresource.properties 文件中读取英文静态文本，从 messageresource\_ch.properties 文件中读取中文静态文本，然后把它们装载到各自的 Properties 对象中，最后再把两个 Properties 对象作为属性保存到 ServletContext 中：

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

```

throws ServletException {
super.init(config);

Properties ps=new Properties();
Properties ps_ch=new Properties();
try{
    ServletContext context=config.getServletContext();
    InputStream in=context.getResourceAsStream("/WEB-INF/messageresource.properties");
    ps.load(in);
    InputStream in_ch=context.getResourceAsStream
    ("/WEB-INF/messageresource_ch.properties");
    ps_ch.load(in_ch);

    in.close();
    in_ch.close();

    context.setAttribute("ps",ps);
    context.setAttribute("ps_ch",ps_ch);
}catch(Exception e){
    e.printStackTrace();
}
}
}
}

```

为了保证在 Web 应用启动时就加载 DispatcherServlet，应该在 web.xml 中配置这个 Servlet 时设置 load-on-startup 属性：

```

<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>mypack.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

```

### 3. 创建 MessageTag.java

例程 14-1 是 MessageTag.java 的源程序。

例程 14-1 MessageTag.java

---

```

package mypack;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.http.HttpSession;
import java.util.*;
import java.io.*;

public class MessageTag extends TagSupport
{
    private String key=null;

```

```
public MessageTag() {
}
public String getKey(){
    return this.key;
}

public void setKey(String key){
    this.key=key;
}

// Method called when the closing hello tag is encountered
public int doEndTag() throws JspException {

    try {
        Properties ps=
        (Properties)pageContext.getAttribute("ps",pageContext.APPLICATION_SCOPE);
        Properties ps_ch=
        (Properties)pageContext.getAttribute("ps_ch",pageContext.APPLICATION_SCOPE);

        HttpSession session=pageContext.getSession();
        String language=(String)session.getAttribute("language");
        String message=null;
        if(language!=null && language.equals("Chinese")){
            message=(String)ps_ch.get(key);
            message=new String(message.getBytes("ISO-8859-1"),"GB2312");
        }
        else
            message=(String)ps.get(key);
        pageContext.getOut().print(message);
    }
    catch (Exception e) {
        throw new JspTagException(e.getMessage());
    }
    // We want to return SKIP_BODY because this Tag does not support
    // a Tag Body
    return SKIP_BODY;
}

public void release() {
    super.release();
}
```

MessageTag 包含一个成员变量 key，它与 message 标签的属性 key 对应。在 MessageTag 中定义了 getKey 和 setKey 方法：

```

private String key=null;
public String getKey(){
    return this.key;
}

public void setKey(String key){
    this.key=key;
}

```

在 MessageTag 的 doEndTag 方法中，首先从 pageContext 中读取包含静态文本的 Properties 对象。由于 DispatcherServlet 的 init 方法把 Properties 对象保存在 ServletContext 中，Properties 对象的存在范围是整个 Web 应用，所以在调用 pageContext.getAttribute 方法时设置了 pageContext.APPLICATION\_SCOPE 属性：

```

Properties ps=
(Properties)pageContext.getAttribute("ps",pageContext.APPLICATION_SCOPE);
Properties ps_ch=
(Properties)pageContext.getAttribute("ps_ch",pageContext.APPLICATION_SCOPE);

```

在 MessageTag 的 doEndTag 方法中，接着从 session 中读取当前客户使用的语言：

```

HttpSession session=pageContext.getSession();
String language=(String)session.getAttribute("language");

```

然后根据客户使用的语言，选择相应的 Properties 对象，从中读取 key 对应的静态文本。如果用户选择的是中文语言，还要进行字符编码转换处理。Properties 对象通过 InputStream 装载文件时，按照 ISO-8859-1 编码来读取数据，因此，应该把从 Properties 对象中取出的 message 转换为中文编码 GB2312。

```

String message=null;
if(language!=null && language.equals("Chinese")){
    message=(String)ps_ch.get(key);
    message=new String(message.getBytes("ISO-8859-1"),"GB2312");
}
else
    message=(String)ps.get(key);

pageContext.getOut().print(message);

```

### 14.3 创建标签库描述文件

标签库描述文件（Tag Library Descriptor）文件，简称 TLD，采用 XML 文件格式，定义了用户的标签库。TLD 文件中的元素可以分为 3 类：

- <taglib>：标签库元素
- <tag>：标签元素
- <attribute>：标签属性元素

### 1. 标签库元素<taglib>

标签库元素<taglib>用来设定标签库的相关信息，它的属性说明参见表 14-2。

表 14-2 <taglib>元素的属性

方 法	描 述
libversion	指定 Tag Library 的版本
jspversion	指定 JSP 的版本
shortname	指定 Tag Library 默认的前缀名(prefix)
uri	设定 Tag Library 的唯一访问标识符
info	设定 Tag Library 的说明信息

### 2. 标签元素<tag>

标签元素<tag>元素用来定义一个标签，它的属性说明参见表 14-3。

表 14-3 <tag>元素的属性

方 法	描 述
name	设定 Tag 的名字
tagclass	设定 Tag 的处理类
bodycontent	设定标签的主体(body)内容
info	设定标签的说明信息

<tag>元素的 bodycontent 属性有 3 个可选值：empty、JSP 和 tagdependent。empty 表示标签中没有 body；JSP 表示标签的 body 中可以加入 JSP 程序代码；tagdependent 表示标签中的内容由标签自己去处理。

假定用户定义了一个 sql 标签，它的 bodycontent 属性值为 tagdependent，以下 JSP 代码使用了 sql 标签：

```
<prefix:sql>
    select * from MemberDB where memberID<10
</prefix:sql>
```

这段代码中的 body 为一个 SQL 语句，它将由<prefix:sql>标签来处理，执行这个 SQL 语句。

### 3. 标签属性元素<attribute>

标签属性元素<attribute>用来定义标签的属性，<attribute>元素的属性说明参见表 14-4。

表 14-4 <attribute>元素的属性

方 法	描 述
name	属性名称
required	属性是否必须的，默認為 false
rteprvalue	属性值是否可以为 request-time 表达式

所谓的 request-time 表达式就是类似<%=...%>的表达式。当 rteprvalue（它是 return expression value 的缩写）取值为 true 时，说明属性值可以采用如下方式设置：

```
<% int num=1; %>
<prefix:MyTag attribute1="<% num %>" />
```

下面将创建标签库文件，名为 mytaglib.tld。在这个文件中定义了名为 mytaglib 的标签库，在这个库中定义了一个名为 message 的标签，这个标签有一个名为 key 的属性。例程 14-2 是 mytaglib.tld 文件的源代码。

例程 14-2 mytaglib.tld

---

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->

<taglib>
    <libversion>1.0</libversion>
    <jspversion>1.1</jspversion>
    <shortname>mytaglib</shortname>
    <uri>/mytaglib</uri>

    <tag>
        <name>message</name>
        <tagclass>mypack.MessageTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>produce message by key</info>
        <attribute>
            <name>key</name>
            <required>true</required>
        </attribute>
    </tag>

</taglib>
```

---

## 14.4 在 Web 应用中使用标签

如果 Web 应用中用到了自定义标签，则必须在 web.xml 文件中加入<taglib>元素，它用于声明所引用的标签所在的标签库：

```
<taglib>
    <taglib-uri>/mytaglib</taglib-uri>
    <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
</taglib>
```

<taglib>元素中的属性描述参见表 14-5。

表 14-5 &lt;taglib&gt;元素的属性

属性	说明
<taglib-uri>	设定 Tag Library 的惟一标识符，在 Web 应用中将根据这一标识符来引用 Tag Library
<taglib-location>	指定和 Tag Library 对应的 TLD 文件的位置

在 JSP 文件中需要加入<%@ taglib>指令来声明对标签库的引用，例如：

```
<%@ taglib uri="/mytaglib" prefix="mm" %>
```

uri 用来指定 Tag Library 的标识符，它必须和 web.xml 中的<taglib\_uri>属性保持一致。prefix 表示在 JSP 网页中引用这个标签库中的标签时的前缀，例如，以下代码表示引用 mytaglib 标签库中的<message>标签：

```
<title><mm:message key="hello.title" /></title>
```

下面将对第 2 章介绍的 helloapp 应用进行修改，使它支持中文和英文两个版本。

## 1. 修改 index.htm

我们将在 index.htm 文件中提供中文和英文版本的链接，以下是修改后的 index.htm 文件：

```
<html>
<head>
<title>helloapp</title>
</head>
<body >
<p><font size="7">Welcome to HelloApp</font></p>
<p><a href="login.jsp?language=English">English version </a>
<p><a href="login.jsp?language=Chinese">Chinese version </a>
</body>
</html>
```

## 2. 修改 login.jsp

在 login.jsp 中，首先加入 Java 代码，读取客户选择的语言种类，并把它作为 language 属性保存在 session 对象中。然后用<mm:message>标签替换 login.jsp 中所有的静态文本。以下是修改后的 login.jsp 文件：

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/mytaglib" prefix="mm" %>
<html>

<% String language=request.getParameter("language");
if(language==null)language="English";
session.setAttribute("language",language);
%>

<head>
<title><mm:message key="login.title" /></title>
</head>
<body >
<br>
```

```

<form name="loginForm" method="post" action="dispatcher">
    <table>
        <tr>
            <td><div align="right"><mm:message key="login.user" /></div></td>
            <td><input type="text" name="username"></td>
        </tr>
        <tr>
            <td><div align="right"><mm:message key="login.password" /></div></td>
            <td><input type="password" name="password"></td>
        </tr>
        <tr>
            <td></td>
            <td>
                <input type="Submit" name="Submit" value=<mm:message key="login.submit" />>
            </td>
        </tr>
    </table>
</form>
</body>
</html>

```

### 3. 修改 hello.jsp

用<mm:message>标签替换 hello.jsp 中所有的静态文本。如果用户选择的是中文语言，则还要进行字符编码转换处理。客户提交的表单数据采用默认的 ISO-8859-1 编码，应该把它转换为中文编码 GB2312。

以下是修改后的 hello.jsp 文件：

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/mytaglib" prefix="mm" %>
<html>
<head>
    <title><mm:message key="hello.title" /></title>
</head>
<body>

<%
String user=(String)request.getAttribute("USER");
String language=(String)session.getAttribute("language");
if(language!=null && language.equals("Chinese")){
user=new String(user.getBytes("ISO-8859-1"),"GB2312");
}
%>

<b><mm:message key="hello.hello" /> : <%= user %></b>
</body>
</html>

```

## 14.5 发布支持中、英文版本的 helloapp 应用

下面，按以下步骤发布并运行加入了 message 标签的 helloapp 应用。

### 步骤

(1) 编译 MessageTag.java 和修改后的 DispatcherServlet.java，编译时需要将 servlet-api.jar 和 jsp-api.jar 文件添加到 classpath 中，这两个 jar 文件都位于<CATALINA\_HOME>/common/lib 目录下。编译后生成的类的存放位置为：

```
<CATALINA_HOME>/webapps/helloapp/WEB-INF/classes/mypack/MessageTag.class  
<CATALINA_HOME>/webapps/helloapp/WEB-INF/classes/mypack/DispatcherServlet.class
```

(2) 将 messengeresource.properties 文件 和 messengeresource\_ch.properties 文件拷贝到 <CATALINA\_HOME>/webapps/helloapp/WEB-INF 目录下。

(3) 创建 mytaglib.tld 文件（参见 14.3 节例程 14-2），它的存放位置为：

```
<CATALINA_HOME>/webapps/helloapp/WEB-INF/mytaglib.tld
```

(4) 在 web.xml 文件中加入<taglib>，修改后的 web.xml 文件如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE web-app PUBLIC  
'-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'  
'http://java.sun.com/j2ee/dtds/web-app_2_3.dtd'>
```

```
<web-app>
```

```
    <servlet>  
        <servlet-name>dispatcher</servlet-name>  
        <servlet-class>mypack.DispatcherServlet</servlet-class>  
        <load-on-startup>1</load-on-startup>  
    </servlet>
```

```
    <servlet-mapping>  
        <servlet-name>dispatcher</servlet-name>  
        <url-pattern>/dispatcher</url-pattern>  
    </servlet-mapping>
```

```
    <taglib>  
        <taglib-uri>/mytaglib</taglib-uri>  
        <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>  
    </taglib>
```

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

(5) 按 14.4 节的相关内容修改 index.htm、login.jsp 和 hello.jsp 文件。

(6) 重启 Tomcat 服务器，访问 http://localhost:8080/helloapp/index.htm，生成的网页如

图 14-2 所示。

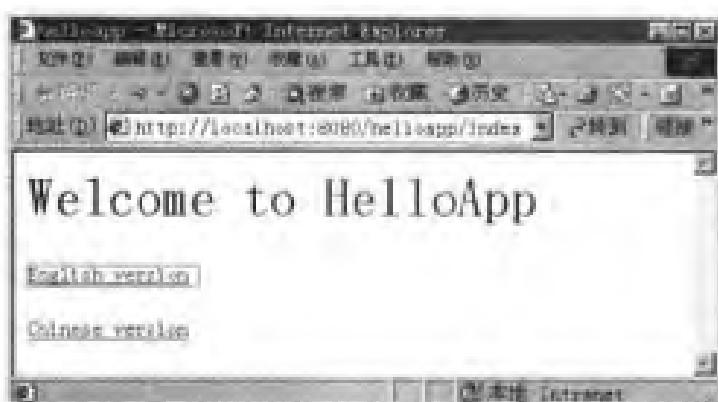


图 14-2 index.htm 网页

(7) 在 index.htm 网页上选择“Chinese version”链接，进入 login.jsp 页面，此时会看到 login.jsp 的 title 以及网页上的内容均为中文，如图 14-3 所示。

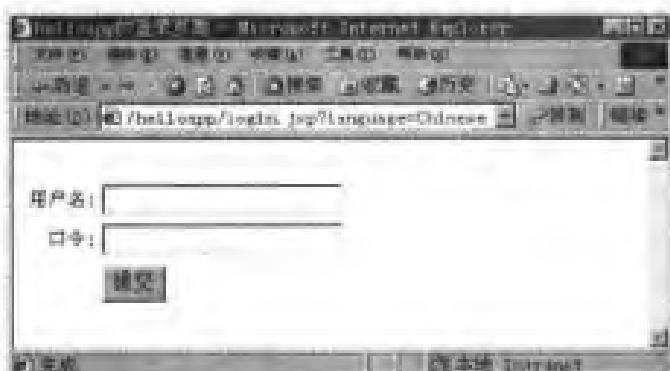


图 14-3 中文版本的 login.jsp 网页

(8) 在 login.jsp 中输入用户名和口令，然后单击【提交】按钮，此时客户请求由 DispatcherServlet 来处理，它把请求再转发给 hello.jsp，会看到 hello.jsp 网页的 title 和网页内容均为中文，如图 14-4 所示。



图 14-4 中文版本的 hello.jsp 网页

(9) 再次访问 <http://localhost:8080/helloapp/index.htm>，这次选择“English version”链接，接下来会看到 login.jsp 和 hello.jsp 的输出网页均为英文。

在本书配套光盘的 sourcecode/chapter14/helloapp 目录下提供了包含以上所有文件的 helloapp 应用。如果要发布这个应用，只要把整个 helloapp 应用拷贝到<CATALINA\_HOME>

/webapps 目录下即可。它的目录结构如图 14-5 所示。

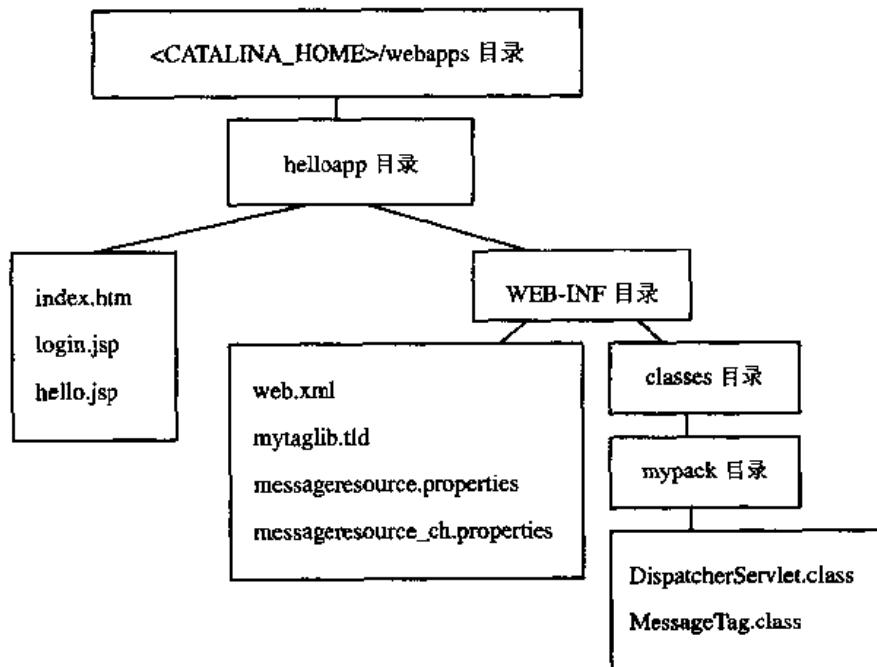


图 14-5 helloapp 应用的目录结构

## 14.6 小结

JSP Tag Library 技术是在 JSP 1.1 版本中才出现的，它支持用户在 JSP 文件中自定义标签，这样可以使 JSP 代码更加简洁。JSP 容器编译 JSP 网页时，如果遇到自定义标签，就会调用这个标签的处理类（Tag Handler Class）。处理标签的类必须扩展 javax.servlet.jsp.TagSupport 类或者 javax.servlet.jsp.BodyTagSupport，本书介绍了 TagSupport 类的用法。TagSupport 类提供了两个处理标签的方法：doStartTag() 和 doEndTag()。当 JSP 容器遇到自定义标签的起始标志，就会调用 doStartTag() 方法，当 JSP 容器遇到自定义标签的结束标志时，就会调用 doEndTag() 方法。在标签处理方法中，可以通过 TagSupport 的成员变量 pageContext 来访问 Web 应用的共享数据。

# 第 15 章 采用模板设计网上书店应用

在设计网站时，常常希望所有的网页保持同样的风格。本章以 bookstore 应用为例，介绍如何通过自定义 JSP 标签来为网站设计模板，所有显示在客户端的网页都通过模板来生成。采用这种办法来设计大型网站，可以提高网站的开发效率，使网页便于维护。假如网页的风格出现需求变更，不需要修改所有的网页，只要修改模板即可。通过本章内容，读者可以进一步掌握开发自定义 JSP 标签的技术。

## 15.1 如何设计网站的模板

在设计网站的模板时，首先应该找出所有网页在结构和内容上的相同之处，然后把这些相同之处定义在模板中，模板本身也常常是个 JSP 文件。在 bookstore 应用中，所有显示在客户端的网页都采用如下结构：

```
<html>
<head>
<title>
    <!-- title -->
</title>
</head>
<!-- banner -->
<!-- body -->
</body>
</html>
```

这些网页都包含 title、banner 和 body 三部分，通过 template.jsp 文件来定义这个基本框架，当客户请求某个网页时，template.jsp 文件中的自定义标签可以根据客户的请求，把具体的 title、banner 和 body 内容填充进去，生成客户所需的网页。

所有的客户请求都由 DispatcherServlet 类来处理，它把客户请求的具体 URL 保存在 HttpServletRequest 对象的属性中，再把客户请求转发给 template.jsp，template.jsp 从 HttpServletRequest 对象中读取客户请求的 URL，然后根据客户请求生成相应的网页。

当客户请求查看购物车内容时，服务器端的流程如图 15-1 所示。DispatcherServlet 将客户请求转发给 template.jsp，template.jsp 根据客户请求把相应的 title、banner.jsp 和 showcart.jsp 填充到模板中，生成客户需要的网页。

对 bookstore 应用进行模板设计，包含以下内容：

- 创建负责流程控制的 Servlet：DispatcherServlet
- 创建模板标签和模板 JSP 文件。这里将创建 4 个标签：`<definition>`、`<screen>`、`<insert>` 和`<parameter>`。`template.jsp` 和 `screendefinitions.jsp` 用来生成所有网页的模板

- 修改 bookstore 应用中原有的 JSP 文件，使它们仅负责生成模板中 body 部分的内容

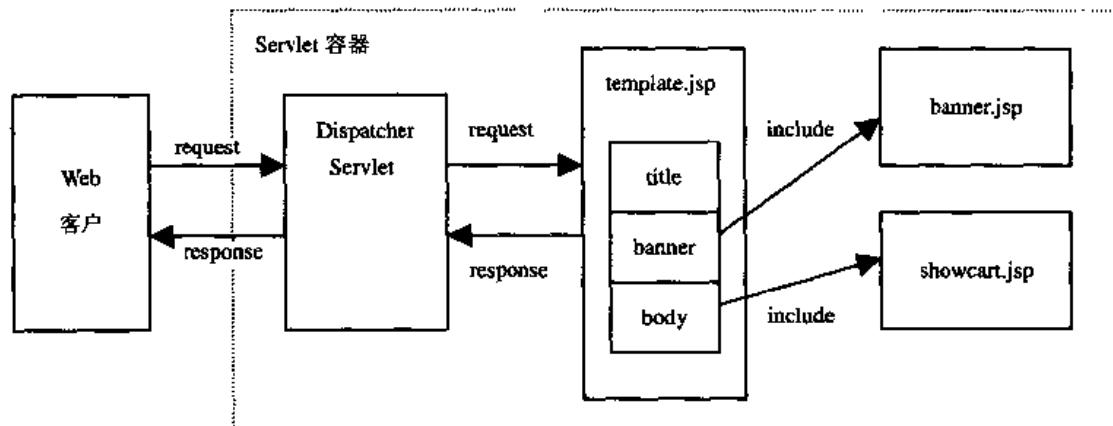


图 15-1 客户请求查看购物车内容时服务器端的流程

## 15.2 创建负责流程控制的 Servlet

所有的客户请求都由 DispatcherServlet 类来处理，它把客户请求的 URL 作为 selectedScreen 属性保存在 HttpServletRequest 对象中，然后再把请求转发给 template.jsp。template.jsp 文件中的自定义标签再从 HttpServletRequest 对象中读取 selectedScreen 属性，把具体的 title、banner 和 body 内容填充进去，生成客户需要的网页。DispatcherServlet 类的源程序如下：

```

package mypack;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class DispatcherServlet extends HttpServlet {
    public void service(HttpServletRequest request, HttpServletResponse response) {
        request.setAttribute("selectedScreen", request.getServletPath().substring(1));
        try {
            request.getRequestDispatcher("/template.jsp").forward(request, response);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

以上代码中获取客户请求 URL 的代码为：

`request.getServletPath().substring(1)`

如果客户请求的 URL 为 /bookdetails，调用 String 类的 substring(1)方法将返回 bookdetails 字符串。

为了能让所有的客户请求都首先由 DispatcherServlet 来处理，我们在 web.xml 文件中为 DispatcherServlet 类配置了多个<servlet-mapping>元素：

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>mypack.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/enter</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/catalog</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/bookdetails</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/showcart</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/cashier</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/receipt</url-pattern>
</servlet-mapping>
```

对于 bookstore 应用中原有的 JSP 文件，应该修改文件中的链接地址。例如，在 bookstore.jsp 文件中，有如下链接：

```
<p><b><a href="<% =request.getContextPath()%>/catalog.jsp">查看所有节目</a></b>
```

应该把这个链接改为：

```
<p><b><a href="<% =request.getContextPath()%>/catalog ">查看所有书目</a></b>
```

这样，当客户选择 “/catalog” 链接时，客户请求就会先被 DispatcherServlet 接收，而不是直接交给 catalog.jsp 文件处理。

原有的 JSP 文件和 DispatcherServlet 的<url-pattern>的对应关系如图 15-2 所示。

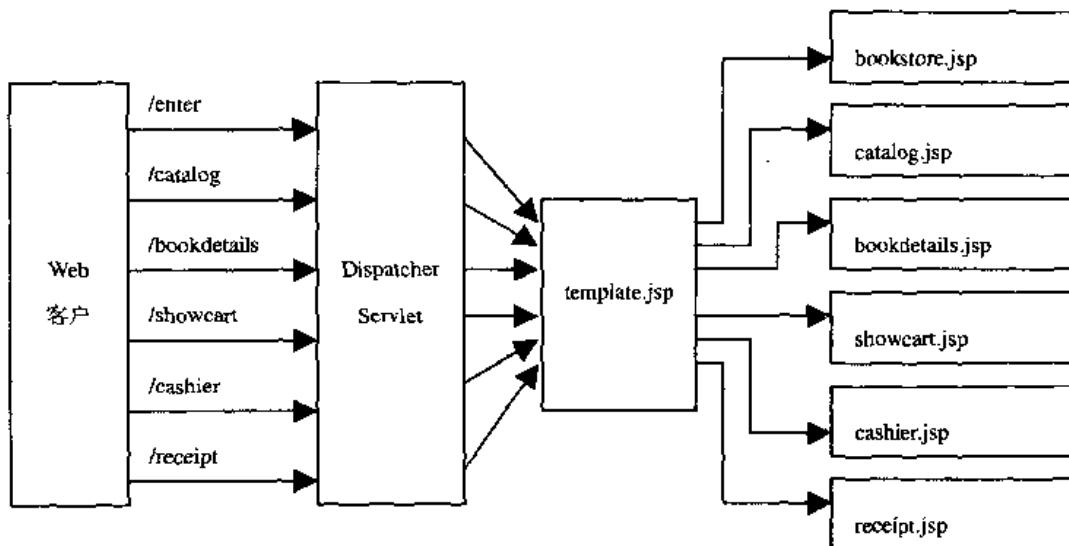


图 15-2 JSP 文件和 DispatcherServlet 的<url-pattern>的对应关系

### 15.3 创建模板标签和模板 JSP 文件

根据上一节已经知道，template.jsp 是生成所有网页的模板文件。以下是 template.jsp 的源代码：

```

<%@ taglib uri="/mytaglib" prefix="mm" %>
<%@ page errorPage="errorpage.jsp" %>
<%@ page import="java.util.*" %>
<%@ include file="screendefinitions.jsp" %>
<html>
<head>
<title>
    <mm:insert definition="bookstore" parameter="title"/>
</title>
</head>
<body>
    <mm:insert definition="bookstore" parameter="banner"/>
    <mm:insert definition="bookstore" parameter="body"/>
</body>
</html>
  
```

这个文件定义了所有网页的基本框架，由 title、banner 和 body 组成。<mm:insert>标签能根据客户请求，把相应的 title、banner 和 body 内容填充到模板中，形成客户所需的网页。

DispatcherServlet 把所有的客户请求都转发给 template.jsp，那么 template.jsp 是如何根据客户的请求来决定网页的 title、banner 和 body 的具体内容的呢？在 template.jsp 中通过<%@ include%>标记包含了 screendefinitions.jsp 文件，这个文件负责依据客户的请求决定网页的具体内容。例程 15-1 是 screendefinitions.jsp 的代码。

## 例程 15-1 screendefinitions.jsp

```

<mm:definition name="bookstore" screenId="<%=(String)request.getAttribute("selectedScreen")%>">
    <mm:screen id="enter">
        <mm:parameter name="title" value="Bookstore" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/bookstore.jsp" direct="false"/>
    </mm:screen>
    <mm:screen id="catalog">
        <mm:parameter name="title" value="BookCatalog" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/catalog.jsp" direct="false"/>
    </mm:screen>
    <mm:screen id="bookdetails">
        <mm:parameter name="title" value="TitleBookDescription" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/bookdetails.jsp" direct="false"/>
    </mm:screen>
    <mm:screen id="showcart">
        <mm:parameter name="title" value="TitleShoppingCart" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/showcart.jsp" direct="false"/>
    </mm:screen>
    <mm:screen id="cashier">
        <mm:parameter name="title" value="TitleCashier" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/cashier.jsp" direct="false"/>
    </mm:screen>
    <mm:screen id="receipt">
        <mm:parameter name="title" value="TitleReceipt" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/receipt.jsp" direct="false"/>
    </mm:screen>
</mm:definition>

```

在 screendefinitions.jsp 文件中，<mm:definition>标签定义了客户所请求的某个具体网页的内容。DispatcherServlet 类把客户请求的 URL 作为 selectedScreen 属性保存在 HttpServletRequest 对象中，在 screendefinitions.jsp 文件中又从 HttpServletRequest 对象中读取 selectedScreen 属性，把它赋值给<mm:definition>标签的 screenId 属性。

```

<mm:definition name="bookstore" screenId="<%=(String)request.getAttribute("selectedScreen")%>">

```

在<mm:definition>标签中还包含了许多<mm:screen>标签，这些标签用来预先定义各类网页的具体内容（包括 title、banner 和 body）。所有网页的 banner 都对应 banner.jsp 文件。body 是模板中的主体部分，它对应的文件可以是 bookstore.jsp、bookdetails.jsp、catalog.jsp、

showcart.jsp、cashier.jsp 和 receipt.jsp 中的一个。

在 template.jsp 和 screendefinitions.jsp 文件中一共用到了 4 个标签：`<definition>`、`<screen>`、`<insert>`和`<parameter>`。这些标签的描述参见表 15-1。

表 15-1 模板标签以及标签处理类

标 签	描 述	处理类和相关的类
parameter	定义网页中 title、banner 或 body 对应的内容	ParameterTag、Parameter
screen	预先定义某个网页的内容	ScreenTag
definition	根据客户请求决定相应网页的内容	DefinitionTag、Definition
insert	把客户请求的网页的各项内容填充到模板中，生成客户需要的网页	InsertTag

下面分别讲述这些标签的作用和标签处理类创建过程。

### 15.3.1 <parameter>标签和其处理类

在 screendefinitions.jsp 文件中，多次使用到了`<parameter>`元素，例如：

```
<cmm:screen id="enter">
    <cmm:parameter name="title" value="Bookstore" direct="true"/>
    <cmm:parameter name="banner" value="/banner.jsp" direct="false"/>
    <cmm:parameter name="body" value="/bookstore.jsp" direct="false"/>
</cmm:screen>
```

`<parameter>`元素定义了网页中的某一项内容，它有 3 个属性：

- `name`：指定在网页中所处的位置，取值范围是 title、banner 和 body
- `value`：设定具体的内容，如果 `name` 为 title，就直接给出字符串；如果 `name` 为 banner 或 body，就给出相应的 JSP 文件名
- `direct`：指定此项内容是否可以直接在网页上输出，如果 `name` 属性为 title，那么 `direct` 属性为 true；如果 `name` 属性为 banner 或 body，那么 `direct` 属性为 false

在`<insert>`标签的处理类 InsertTag 中，会根据`<parameter>`标签的 `direct` 属性来决定输出某项内容的方法，如果 `direct` 属性为 true，就直接输出 `value` 属性；如果 `direct` 属性为 false，此时 `value` 属性对应的是 JSP 文件，所以应该用 include 方法把这个 JSP 文件包含到 template.jsp 文件中。以下是 InsertTag 类中输出`<parameter>`的 `value` 的代码：

```
// if parameter is direct, print to out
if (direct || include && parameter != null)
    pageContext.getOut().print(parameter.getValue());
// if parameter is indirect, include results of dispatching to page
else {
    if ((parameter != null) && (parameter.getValue() != null))
        pageContext.include(parameter.getValue());
}
```

`<parameter>`标签采用 Parameter 类来保存所有属性，下面是 Parameter 的代码：

```
package mytaglib;
public class Parameter {
    private String name;
```

```

private boolean isDirect;
private String value;
public Parameter(String name, String value, boolean isDirect) {
    this.name = name;
    this.isDirect = isDirect;
    this.value = value;
}
public String getName() {
    return name;
}
public boolean isDirect() {
    return isDirect;
}
public String getValue() {
    return value;
}
}

```

<parameter>标签的处理类为 ParameterTag，它在 doStartTag()方法中构造一个 Parameter 对象，把<parameter>标签的 name、value 和 direct 属性保存在 Parameter 对象中。然后把 Parameter 对象保存到一个 ArrayList 对象中，这个 ArrayList 对象名为 parameters，以下是这段操作的代码：

```

if (paramName != null) {
    ArrayList parameters = ((TagSupport)getParent()).getValue("parameters");
    if (parameters != null) {
        Parameter param = new Parameter(paramName, paramValue, isDirect);
        parameters.add(param);
    }
}

```



java.util.ArrayList 是一个集合类，可以通过它的 add(Object obj)方法，向集合中添加对象。

以上代码中的 getParent 和 getValue 方法是在 TagSupport 类中定义的，ParameterTag 类继承了 TagSupport 类的这两个方法。getParent()方法返回上一层标签的处理类对象。从 screendefinitions.jsp 文件中可以看出，<parameter>标签嵌套在<screen>标签中，因此这里的 getParent 方法返回 ScreenTag 对象。

在 TagSupport 类中定义了 setValue(String key, Object obj)和 getValue(String key)方法，通过 setValue(String key, Object obj)方法，可以在标签处理对象中以 key/value 的方式保存一些共享数据，并且在需要的时候通过 getValue(String key)方法读取共享数据。

parameters 对象保存在 ScreenTag 中，所以以上代码调用 ScreenTag 的 getValue("parameters")方法取得 parameters 对象。

在 screendefinitions.jsp 文件中，每个<screen>标签包含 3 个<parameter>标签，它们分别代表了一个网页的 title、banner 和 body，当 3 个<parameter>标签的处理类都执行完毕，在

parameters 对象中保存了 3 个 Parameter 对象。ScreenTag、parameters 和 Parameter 对象之间的关系如图 15-3 所示。

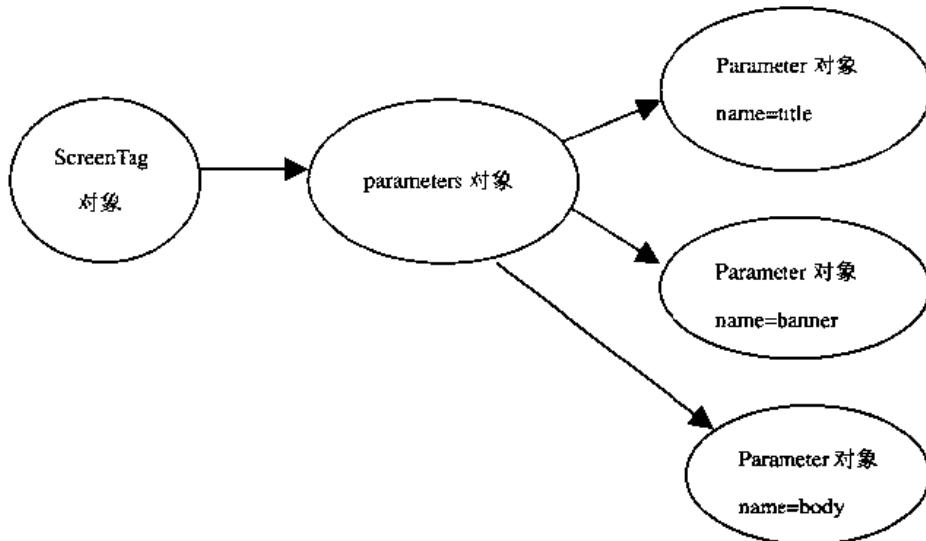


图 15-3 ScreenTag、parameters 和 Parameter 对象之间的关系

例程 15-2 是 ParameterTag.java 的源程序。

#### 例程 15-2 ParameterTag.jsp

```
package mytaglib;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.*;
import java.util.*;

public class ParameterTag extends TagSupport {
    private Tag parentTag = null;
    private String paramName = null;
    private String paramValue = null;
    private String isDirectString = null;

    public ParameterTag() {
        super();
    }

    public void setName(String paramName) {
        this.paramName = paramName;
    }

    public void setValue(String paramValue) {
        this.paramValue = paramValue;
    }

    public void setDirect(String isDirectString) {
        this.isDirectString = isDirectString;
    }

    public int doStartTag() {
```

```

boolean isDirect = false;

if ((isDirectString != null) &&
    isDirectString.toLowerCase().equals("true"))
    isDirect = true;

try {
    // retrieve the parameters list
    if (paramName != null) {
        ArrayList parameters = (ArrayList)((TagSupport)getParent()).getValue("parameters");
        if (parameters != null) {
            Parameter param = new Parameter(paramName, paramValue, isDirect);
            parameters.add(param);
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
return SKIP_BODY;
}

public void release() {
    parentTag = null;
    paramName = null;
    paramValue = null;
    isDirectString = null;
    super.release();
}
}

```

### 15.3.2 <screen>标签和处理类

在 screendefinitions.jsp 中一共定义了 6 个<screen>标签，分别代表了 6 个网页。<screen>标签是<definition>标签的子元素，同时它又是<parameter>标签的父元素：

```

<mm:definition name="bookstore" screenId="<%=(String)request.getAttribute("selectedScreen")%>">
    <mm:screen id="enter">
        <mm:parameter name="title" value="Bookstore" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/bookstore.jsp" direct="false"/>
    </mm:screen>
    .....
</mm:definition>

```

<screen>标签预定义了某个网页的各项内容，它有一个属性 id，用于标志网页名。<screen>标签的处理类为 ScreenTag，在 doStartTag 方法中，它首先创建了一个 ArrayList 对象，并且调用 setValue 方法把它作为属性保存起来：

```
setValue("parameters", new ArrayList());
```

这个 `ArrayList` 类型对象用来存放 `Parameter` 对象，`<screen>` 标签和 3 个子类标签 `<parameter>` 共享这个 `ArrayList` 类型对象。上一节已经讲过，`<parameter>` 标签的处理类把相应的 `Parameter` 对象都保存在这个 `ArrayList` 类型对象中（在 `ParameterTag` 类中把它命名为 `parameters`）。`ScreenTag` 和 `ParameterTag` 操纵 `parameters` 对象的时序图如图 15-4 所示。

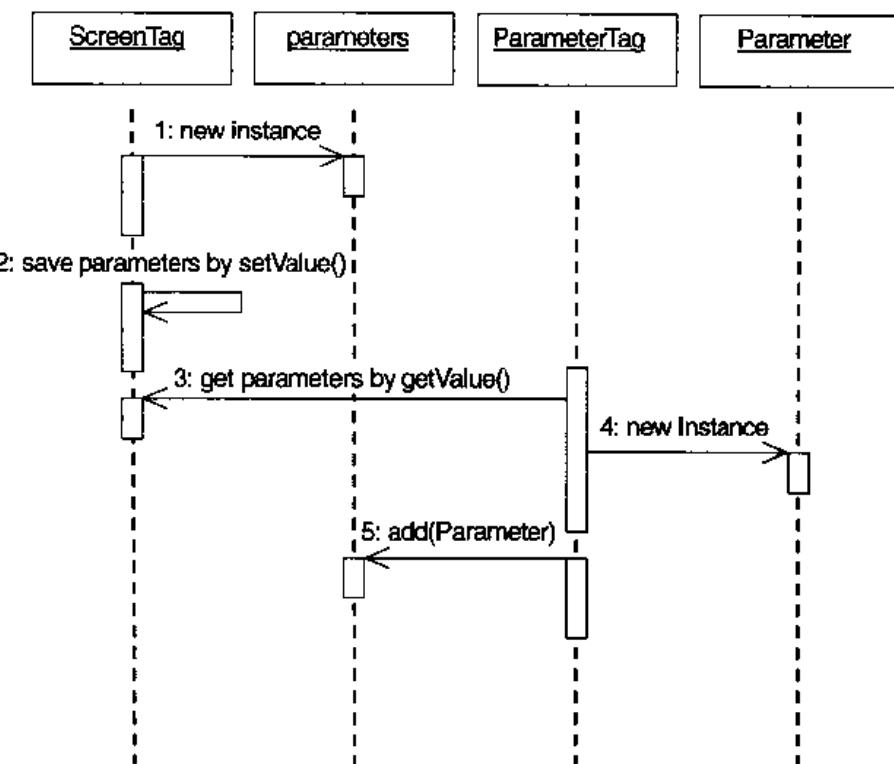


图 15-4 ScreenTag 和 ParameterTag 操纵 parameters 对象的时序图

在 `ScreenTag` 的 `doStartTag` 方法中，从 `PageContext` 中取得 `HashMap` 类型的 `screens` 对象，然后把 `parameters` 对象以 `key/value` 的形式保存在 `screens` 对象中。

```

HashMap screens = (HashMap) pageContext.getAttribute("screens",
pageContext.APPLICATION_SCOPE);
if (screens == null) {
    return SKIP_BODY;
}
else {
    if (!screens.containsKey(getId())) {
        screens.put(getId(), getValue("parameters"));
    }
    return EVAL_BODY_INCLUDE;
}
  
```

### 提示

`java.util.HashMap` 类提供了以 `key/value` 的方式保存对象的方法，可以通过 `put(Object key, Object value)` 方法保存对象，还可以通过 `get(Object key)` 方法取得被保存对象的引用。

在 screendefinitions.jsp 中一共定义了 6 个<screen>标签，它们都把各自的 parameters 对象保存在 screens 对象中，在保存时以<screen>标签的 id 属性作为 key。当 6 个<screen>标签的处理类都执行完毕，在 screens 对象中保存了 6 个 parameters 对象，pageContext、screens 和 parameters 对象之间的关系如图 15-5 所示。

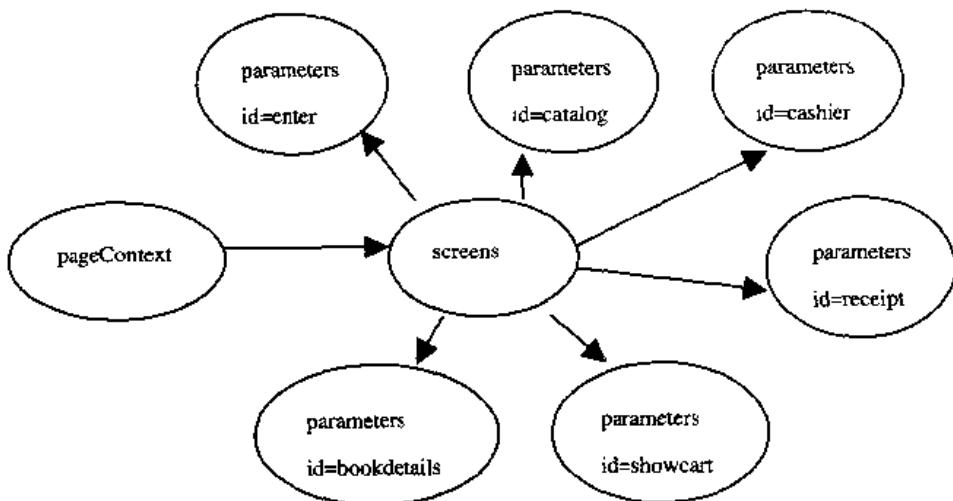


图 15-5 pageContext、screens 和 parameters 对象之间的关系

例程 15-3 是 ScreenTag.java 的源程序。

例程 15-3 ScreenTag.java

---

```

package mytaglib;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.*;
import java.util.*;

public class ScreenTag extends TagSupport {
    public ScreenTag() {
        super();
    }

    public int doStartTag() {
        setValue("parameters", new ArrayList());
        HashMap screens = (HashMap) pageContext.getAttribute("screens",
            pageContext.APPLICATION_SCOPE);
        if (screens == null) {
            return SKIP_BODY;
        }
        else {
            if (!screens.containsKey(getId())) {
                screens.put(getId(), getValue("parameters"));
            }
        }
        return EVAL_BODY_INCLUDE;
    }
}
  
```

```
        }
    }

    public void release() {
        super.release();
    }
}
```

### 15.3.3 <definition>标签和处理类

<definition>标签根据客户的请求，决定输出网页的内容：

```
<mm:definition name="bookstore" screenId="<%=(String)request.getAttribute("selectedScreen")%>">
    <mm:screen id="enter">
        <mm:parameter name="title" value="Bookstore" direct="true"/>
        <mm:parameter name="banner" value="/banner.jsp" direct="false"/>
        <mm:parameter name="body" value="/bookstore.jsp" direct="false"/>
    </mm:screen>
    .....
    .....
</mm:definition>
```

<definition>标签有两个属性，name 属性用来标识当前的 Definition 对象，screenId 属性用来指定网页的名字。screenId 属性值是由客户请求的 URL 决定的：

screenId="<%=(String)request.getAttribute("selectedScreen")%>"

<definition>标签采用 Definition 类来保存网页的各项内容，下面是 Definition 类的代码：

```
package mytaglib;
import java.util.HashMap;

public class Definition {
    private HashMap params = new HashMap();

    public void setParam(Parameter p) {
        params.put(p.getName(), p);
    }
    public Parameter getParam(String name) {
        return (Parameter) params.get(name);
    }
}
```

一个 Definition 对象代表了客户请求的一个网页，它包含 3 个 Parameter 对象，分别代表网页的 title、banner 和 body。这 3 个 Parameter 对象保存在一个 HashMap 类型的对象中，这个对象名为 params。Definition 对象、params 对象和 Parameter 对象之间的关系如图 15-6 所示。

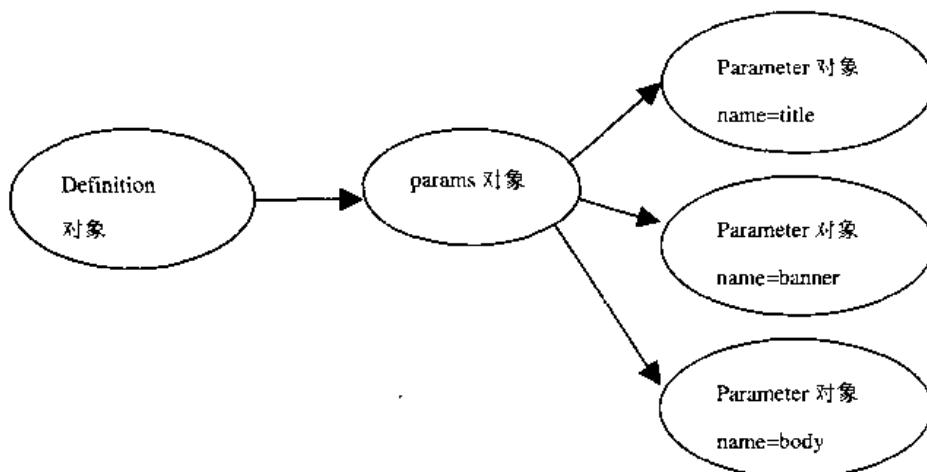


图 15-6 Definition 对象、params 对象和 Parameter 对象之间的关系

<definition>标签的处理类为 DefinitionTag，在 DefinitionTag 的 doStartTag 方法中，首先判断在 pageContext 对象中是否存在 screens 对象，如果不存在，就创建一个 screens 对象，把它保存在 pageContext 中。screens 对象的存在范围为 pageContext.APPLICATION\_SCOPE，它表示在整个 Web 应用中只有一个 screens 对象，可以被所有的 Web 组件共享，以下是这段操作的代码：

```

HashMap screens = null;
screens = (HashMap) pageContext.getAttribute("screens", pageContext.APPLICATION_SCOPE);
if (screens == null)
    pageContext.setAttribute("screens", new HashMap(), pageContext.APPLICATION_SCOPE);
    return EVAL_BODY_INCLUDE;
}
  
```

在上一节，已经讲过每个<screen>标签都会创建一个 parameters 对象，并把它保存在 screens 对象中。所以，screens 对象用来存放 bookstore 应用的所有网页的内容，参见 15.3.2 节的图 15-5。

在 DefinitionTag 的 doEndTag 方法中，首先从 pageContext 中取得 screens 对象，再从 screens 对象中取得与客户请求的 screenId 对应的 parameters 对象，在这个 parameters 对象中存放了客户请求的网页的内容。

```

screens = (HashMap) pageContext.getAttribute("screens", pageContext.APPLICATION_SCOPE);
if (screens != null) {
    parameters = (ArrayList) screens.get(screenId);
}
  
```

接下来把 parameters 的内容封装到一个 Definition 对象中：

```

Iterator ir = null;
if (parameters != null)
    ir = parameters.iterator();

while ((ir != null) && ir.hasNext())
    definition.setParam((Parameter) ir.next());
  
```

这样，这个 Definition 对象就代表了客户请求的网页的内容。然后把这个 Definition 对象保存到 pageContext 对象中：

```
// put the definition in the page context
pageContext.setAttribute(definitionName, definition);
```

DefinitionTag 创建 Definition 对象的时序图如图 15-7 所示。

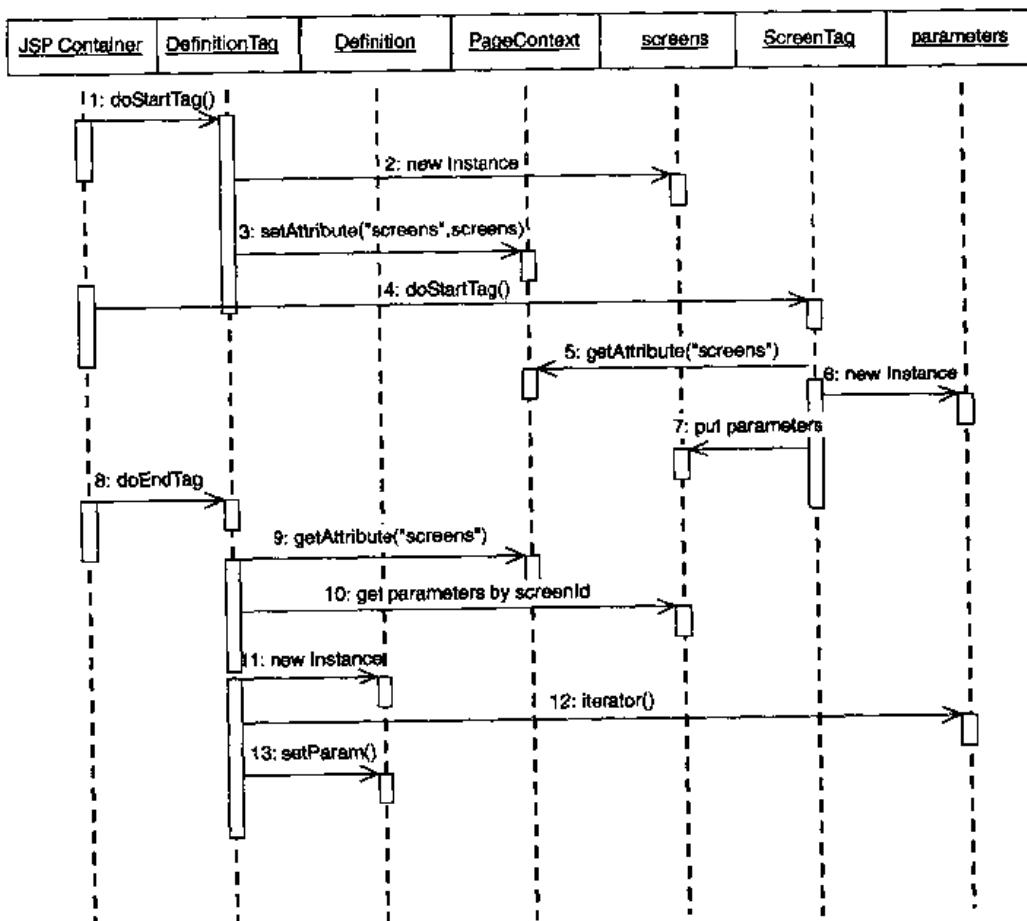


图 15-7 DefinitionTag 创建 Definition 对象的时序图

例程 15-4 是 DefinitionTag.java 的源程序。

例程 15-4 DefinitionTag.java

```
package mytaglib;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;
import java.util.*;

public class DefinitionTag extends TagSupport {
    private String definitionName = null;
    private String screenId;

    public DefinitionTag() {
        super();
    }

    public void setDefinitionName(String definitionName) {
        this.definitionName = definitionName;
    }

    public void setScreenId(String screenId) {
        this.screenId = screenId;
    }

    public String getDefinitionName() {
        return definitionName;
    }

    public String getScreenId() {
        return screenId;
    }

    public void doStartTag() throws JspTagException {
        PageContext pageContext = (PageContext) this.pageContext;
        screens screens = new screens();
        pageContext.setAttribute("screens", screens);
        screens.setDefinitionName(definitionName);
        screens.setScreenId(screenId);
    }

    public void doEndTag() throws JspTagException {
        PageContext pageContext = (PageContext) this.pageContext;
        screens screens = (screens) pageContext.getAttribute("screens");
        pageContext.removeAttribute("screens");
    }
}
```

```

    }

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

    public void setScreenId(String screenId) {
        this.screenId = screenId;
    }

    public int doStartTag() {
        HashMap screens = null;

        screens = (HashMap) pageContext.getAttribute("screens",
            pageContext.APPLICATION_SCOPE);
        if (screens == null)
            pageContext.setAttribute("screens", new HashMap(),
            pageContext.APPLICATION_SCOPE);
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspTagException {
        try {
            Definition definition = new Definition();
            HashMap screens = null;
            ArrayList parameters = null;
            TagSupport screen = null;

            screens = (HashMap) pageContext.getAttribute("screens",
                pageContext.APPLICATION_SCOPE);
            if (screens != null) {
                parameters = (ArrayList) screens.get(screenId);
            }

            Iterator ir = null;

            if (parameters != null)
                ir = parameters.iterator();

            while ((ir != null) && ir.hasNext())
                definition.setParam((Parameter) ir.next());

            // put the definition in the page context
            pageContext.setAttribute(definitionName, definition);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return EVAL_PAGE;
    }

    public void release() {
}

```

```
definitionName = null;
screenId = null;
super.release();
}
}
```

### 15.3.4 <insert>标签和处理类

在 template.jsp 文件中，<insert>标签能够将网页的某部分内容填充到模板中，生成客户需要的网页：

```
<html>
<head>
<title>
    <mm:insert definition="bookstore" parameter="title"/>
</title>
</head>
<body>
    <mm:insert definition="bookstore" parameter="banner"/>
    <mm:insert definition="bookstore" parameter="body"/>
</body>
</html>
```

<insert>标签有两个属性：definition 属性用来指定 Definition 对象，parameter 属性用来指定具体的 Parameter 对象。

<insert>标签的处理类为 InsertTag。在 InsertTag 的 doStartTag 方法中，首先根据<insert>标签的 definition 属性从 pageContext 对象中取得 Definition 对象，它包含了当前客户请求的网页内容信息：

```
definition = (Definition)pageContext.getAttribute(definitionName);
```

然后根据<insert>标签的 parameter 属性从 Definition 对象中取得 Parameter 对象，它包含了网页的某部分内容信息。这个 Parameter 对象可以代表网页的 title、banner 或 body：

```
if (parameterName != null && definition != null)
    parameter = (Parameter)definition.getParam(parameterName);
```

在 InsertTag 的 doEndTag 方法中，把 Parameter 的内容输出到网页上。如果 Parameter 的属性 direct 为 true，此时 Parameter 代表的是网页的 title，它包含的网页内容是文本，因此可以直接把它写到网页上；如果 direct 属性为 false，此时 Parameter 代表的是网页的 banner 或 body，Parameter 包含的网页内容是个 JSP 文件，所以应该通过 include 方法把这个 JSP 文件包含到 template.jsp 文件中：

```
if (directInclude && parameter != null)
    pageContext.getOut().print(parameter.getValue());
// if parameter is indirect, include results of dispatching to page
else {
    if ((parameter != null) && (parameter.getValue() != null))
        pageContext.include(parameter.getValue());
}
```

在 template.jsp 中包含了 3 个<insert>标签，它们分别负责生成网页的 title、banner 和 body。

InsertTag 类向 template.jsp 模板中插入网页内容的时序图如图 15-8 所示。

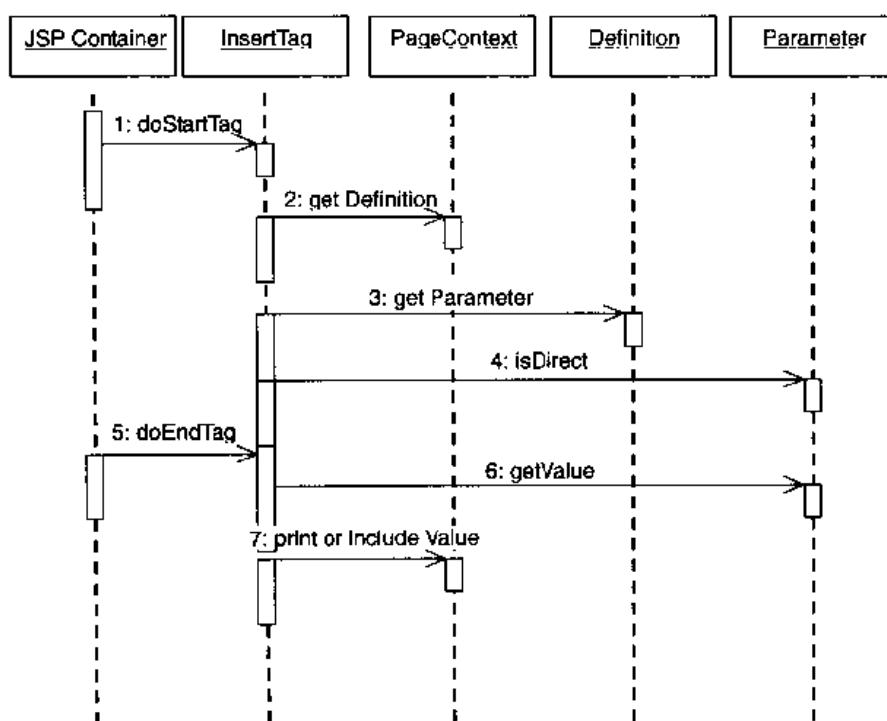


图 15-8 InsertTag 类向 template.jsp 模板中插入网页内容的时序图

例程 15-5 是 InsertTag.java 的源程序。

例程 15-5 InsertTag.java

---

```

package mytaglib;

import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;

public class InsertTag extends TagSupport {
    private boolean directInclude = false;
    private String parameterName = null;
    private String definitionName = null;
    private Definition definition = null;
    private Parameter parameter = null;

    public InsertTag() {
        super();
    }

    public void setParameter(String parameter) {
        this.parameterName = parameter;
    }
}
  
```

```
public void setDefinition(String name) {
    this.definitionName = name;
}
public int doStartTag() {
    // get the definition from the page context

    definition = (Definition)pageContext.getAttribute(definitionName);
    // get the parameter
    if (parameterName != null && definition != null)
        parameter = (Parameter) definition.getParam(parameterName);

    if (parameter != null)
        directInclude = parameter.isDirect();
    return SKIP_BODY;
}
public int doEndTag()throws JspTagException {
    try {
        // if parameter is direct, print to out
        if (directInclude && parameter != null)
            pageContext.getOut().print(parameter.getValue());
        // if parameter is indirect, include results of dispatching to page
        else {
            if ((parameter != null) && (parameter.getValue() != null))
                pageContext.include(parameter.getValue());
        }
    } catch (Exception ex) {
        throw new JspTagException(ex.getMessage());
    }
    return EVAL_PAGE;
}
public void release() {
    directInclude = false;
    parameterName = null;
    definitionName = null;
    definition = null;
    parameter = null;
    super.release();
}
}
```

## 15.4 修改 JSP 文件

现在我们已经知道, template.jsp 能根据客户的请求, 生成客户需要的网页。Bookstore 应用中原有的 JSP 文件不能构成独立的网页, 它们都只是作为 template.jsp 中 body 部分的

内容。因此，应该删除这些 JSP 文件中的 title、banner 内容。

此外，为了保证客户请求的 URL 都能由 DispatcherServlet 来接收，应该修改这些 JSP 文件中的 URL 链接，例如，在 bookstore.jsp 文件中，有如下链接：

```
<p><b><a href="<% =request.getContextPath()%>/catalog.jsp">查看所有书目</a></b>
```

应该把这个链接改为：

```
<p><b><a href="<% =request.getContextPath()%>/catalog">查看所有书目</a></b>
```

这样，当客户选择 “/catalog” 链接时，客户请求就会先被 DispatcherServlet 接收，而不是直接交给 catalog.jsp 文件处理。

下面是修改后的 bookstore.jsp 文件，粗体部分为修改的内容：

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

<%-
<html>
<head><title>Bookstore</title></head>
<% @ include file="banner.jsp" %>
--%>

<center>
<p><b><a href="<% =request.getContextPath()%>/catalog">查看所有书目</a></b>
```

**<FORM action=bookdetails method="POST">**

```
<h3>请输入查询信息</h3>
<b>书的编号:</b>
<input type="text" size="20" name="bookId" value="" ><br><br>
<center><input type=submit value="查询"></center>
</form>
</center>

<%-
</body>
</html>
--%>
```

除了修改 bookstore.jsp 文件外，还应该按照上述方法修改 catalog.jsp、bookdetails.jsp、showcart.jsp、cashier.jsp 和 receipt.jsp 文件。

## 15.5 发布采用模板设计的 bookstore 应用

现在，我们已经对原来的 bookstore 应用进行了改版，使它通过模板来生成网页。新建了如下文件：

- DispatcherServlet.java、
- template.jsp、screendefinitions.jsp
- Parameter.java 和 ParameterTag.java

- ScreenTag.java
- Definition.java 和 DefinitionTag.java
- InsertTag.java

此外,还对原有的 JSP 文件,包括 bookstore.jsp、catalog.jsp、bookdetails.jsp、showcart.jsp、cashier.jsp 和 receipt.jsp 文件进行了修改。

按如下步骤发布 bookstore 应用。

## 步骤

(1) 编译上述新建的 Java 文件。编译生成的 DispatcherServlet.class 存放目录为:

<CATALINA\_HOME>/webapps/bookstore/WEB-INF/classes/mypack

其他的标签处理类和相关的类都存放到以下目录:

<CATALINA\_HOME>/webapps/bookstore/WEB-INF/classes/mytaglib

(2) 所有的 JSP 文件的存放目录为:

<CATALINA\_HOME>/webapps/bookstore

(3) 在<CATALINA\_HOME>/webapps/bookstore/WEB-INF 目录下创建 mytaglib.tld 文件,用于定义<parameter>、<screen>、<definition>和<insert>标签。例程 15-6 是 mytaglib.tld 的源代码。

例程 15-6 mytaglib.tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>mytaglib</shortname>
    <uri>/mytaglib</uri>

    <tag>
        <name>definition</name>
        <tag-class>mytaglib.DefinitionTag</tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>name</name>
            <required>true</required>
            <rtpvalue>true</rtpvalue>
        </attribute>
        <attribute>
            <name>screenId</name>
            <required>true</required>
            <rtpvalue>true</rtpvalue>
        </attribute>
    </tag>
```

```
</tag>
<tag>
    <name>screen</name>
    <tag-class>mytaglib.ScreenTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>id</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
</tag>
<tag>
    <name>parameter</name>
    <tag-class>mytaglib.ParameterTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>name</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
    <attribute>
        <name>value</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
    <attribute>
        <name>direct</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
</tag>
<tag>
    <name>insert</name>
    <tag-class>mytaglib.InsertTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>definition</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
    <attribute>
        <name>parameter</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
</tag>
```

```
</taglib>
```

(4) 修改原来的 web.xml 文件，加入<servlet>、<servlet-mapping>和<taglib>元素，例程 15-7 是 web.xml 的源代码。

例程 15-7 web.xml

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

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>mypack.DispatcherServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/enter</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/catalog</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/bookdetails</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/showcart</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/cashier</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/receipt</url-pattern>
    </servlet-mapping>

<taglib>
    <taglib-uri>/mytaglib</taglib-uri>
```

```
<taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
</taglib>

</web-app>
```

(5) 重启 Tomcat 服务器，访问 <http://localhost:8080/bookstore/enter>，将会看到采用模板设计的 bookstore 应用和第 5 章介绍的 bookstore 应用提供同样的客户界面。

在本书配套光盘的 sourcecode/bookstores/version2/bookstore 目录下已经提供了现成的采用模板设计的 bookstore 应用。要发布这个应用，只要把整个 bookstore 目录拷贝到 <CATALINA\_HOME>/webapps 目录下即可。

## 15.6 小结

本章以 bookstore 应用为例，介绍如何通过自定义 JSP 标签来为网站设计模板，所有显示在客户端的网页都通过模板来生成。采用这种办法来设计大型网站，可以提高开发网站效率，使网页便于维护。对 bookstore 应用进行模板设计的思路是：所有的客户请求都先由 DispatcherServlet 类来处理，它再把客户请求转给 template.jsp。template.jsp 和 screendefinitions.jsp 是生成所有网页的模板，它们包含 4 个自定义 JSP 标签：`<definition>`、`<screen>`、`<insert>` 和 `<parameter>`。`<parameter>` 标签定义网页中 title、hanner 或 body 对应的内容；`<screen>` 标签预定义某个网页的内容；`<definition>` 标签根据客户请求决定相应网页的内容；`<insert>` 标签把客户请求的网页的各项内容填充到模板中，生成客户需要的网页。

通过本章的内容，读者也可以进一步掌握开发自定义 JSP 标签的技术。各个标签之间通过以下方法来共享数据：

- 设置 Web 应用中所有标签可以共享的数据：调用 PageContext 的 `setAttribute(String name, Object value, PageContext.APPLICATION_SCOPE)` 方法来保存共享数据
- 设置父类和子类标签可以共享的数据：调用标签处理类的 `setValue` 方法来保存共享数据，子类标签如果要访问父类标签的数据，则可以先通过标签处理类的 `getParent` 方法来获得父类标签处理类的引用，然后再调用父类标签处理类的 `getValue` 方法来访问共享数据

# 第 16 章 Struts 和 MVC 设计模式

Jakarta-Struts 是 Apache 软件组织提供的一项开放源码工程。它为 Java Web 应用提供了模型-视图-控制器（MVC）框架结构。

本章首先介绍 MVC 设计模式的基本概念，然后讲解 Struts 的框架和工作原理，最后以 helloapp 应用为例，介绍在 Web 应用中使用 Struts 的方法。

## 16.1 MVC 设计模式简介

MVC 是一种流行的软件设计模式，它把系统分为 3 个模块：模型（Model）、视图（View）和控制器（Controller）。各个模块的功能说明参见表 16-1。

表 16-1 MVC 的三个模块

MVC 模块	描述
模型	代表应用程序状态和业务逻辑
视图	提供可交互的客户界面，向客户显示模型数据
控制器	响应客户的请求，根据客户的请求来操纵模型，并把模型的响应结果经由视图展现给客户

各个模块间的相互作用情况如图 16-1 所示。客户可以从视图提供的客户界面上浏览数据或发出请求，客户的请求由控制器处理，它根据客户的请求调用模型的方法，完成数据更新，然后调用视图的方法将响应结果展示给客户。视图也可以直接访问模型，查询数据信息，当模型中数据发生变化时，它会通知视图刷新界面，显示更新后的数据。

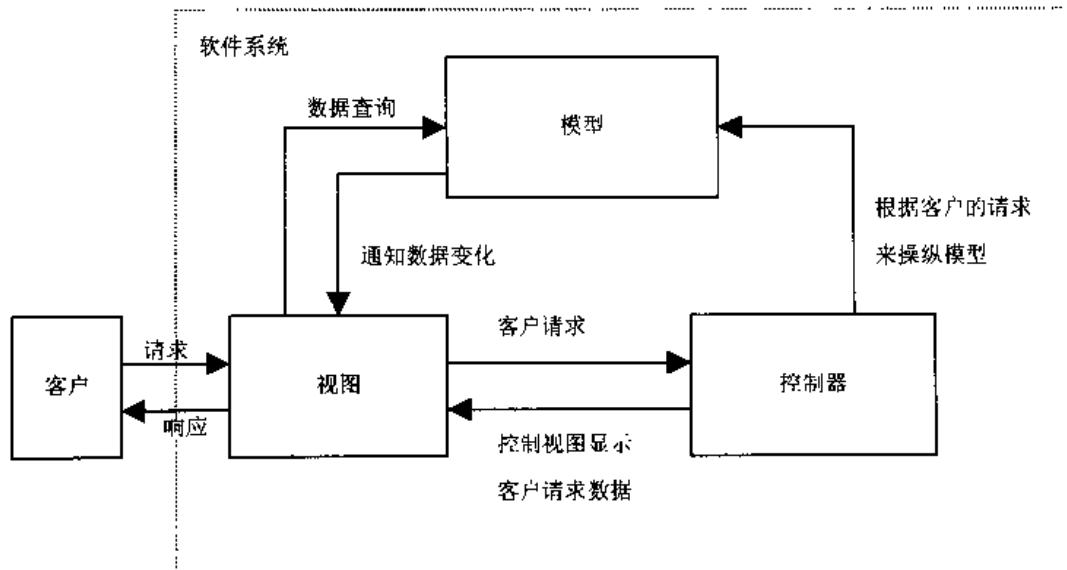


图 16-1 MVC 模型图

## 16.2 Struts 实现的 MVC 设计模式

Struts 把 MVC 设计模式运用到 Web 应用中, 它由一组相互协作的类、Servlet 及 JSP Tag Library 组成。基于 Struts 框架的 Web 应用程序基本上符合 JSP Model2 的设计标准, 可以说是 MVC 设计模式的一种变化类型。Struts 结构如图 16-2 所示。

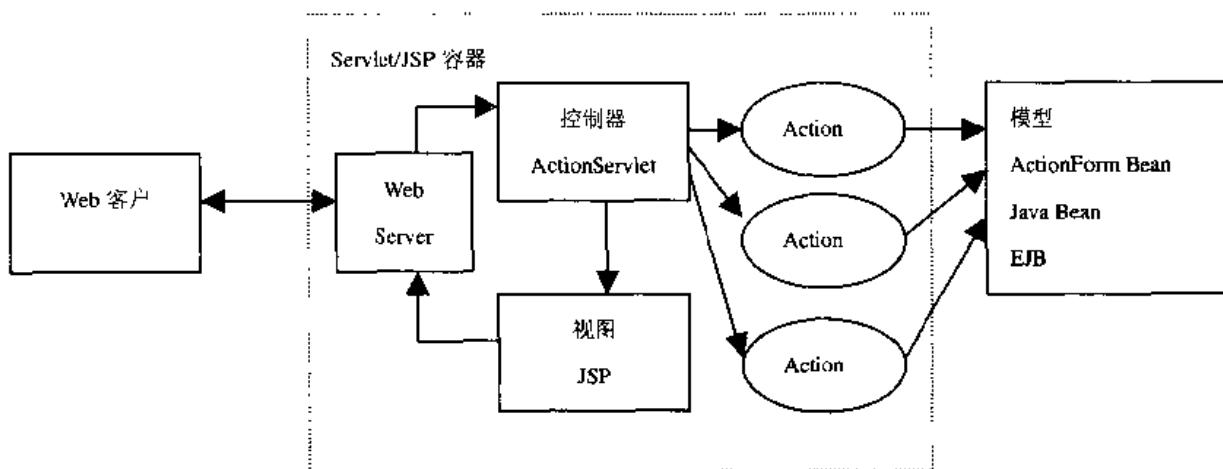


图 16-2 Struts 实现的 MVC 设计模式

在 Struts 框架中, 模型由 ActionForm Bean 和其他实现业务逻辑的 Java Bean 或 EJB 组件构成, 控制器由 ActionServlet 来实现, 视图由一组 JSP 文件构成。

### 16.2.1 视图 (View)

视图就是一组 JSP 文件。在这些 JSP 文件中没有业务逻辑, 也没有模型信息, 只有标签, 这些标签可以是标准的 JSP 标签或 Struts 标签库中的标签。

### 16.2.2 模型 (Model)

模型表示应用程序的状态和业务逻辑。ActionForm Bean 在会话级或请求级表示模型的状态, 而不是在持久级。它可以表示客户的表单数据, JSP 文件使用 Struts 标签读取来自 ActionForm Bean 的信息。对于大型应用, 业务逻辑通常由 Java Bean 或 EJB 组件实现。

在 ActionForm 类中提供了 validate()方法, 它用于对客户提交的表单数据进行验证 (Validation)。通常, 在 validate 方法中只是对表单数据进行一般性的语法或格式检查。validate()方法返回一个 ActionErrors 对象。如果表单验证失败, 在 ActionErrors 对象中就会包含一个或多个 ActionError 对象。如果 validate()方法返回 null 或者返回一个不包含 ActionError 的 ActionErrors 对象, 那么就表示表单验证成功。

### 16.2.3 控制器 (Controller)

控制器由 ActionServlet 类来实现，它是 Struts 框架中的核心组件。ActionServlet 继承了 javax.servlet.http.HttpServlet 类，它在 MVC 模型中扮演控制器的角色。ActionServlet 主要负责接收 HTTP 请求信息，根据配置文件 struts-config.xml 的配置信息，把请求转发给适当的 Action 对象。如果该 Action 对象还不存在，ActionServlet 会先创建这个 Action 对象。

按照 Servlet 规范，所有的 Servlet 必须在 Web 配置文件 web.xml 中声明。以下是对 ActionServlet 的标准声明形式：

```
<servlet>
    <servlet-name>action</servlet name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>detail</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

根据以上的配置，ActionServlet 在 Web 应用启动时就被加载并初始化，在 Web 应用中，所有 “\*.do” 形式的 URL 都由 ActionServlet 来处理。

### 16.2.4 Action 类

Action 类负责实现业务逻辑，更新模型的状态，并帮助控制应用程序的流程。通常把 Action 类也划分到 Model 层中。用户可以定义自己的 Action 类来完成实际的业务逻辑。

对于大型应用，Action 可以充当客户请求和业务逻辑处理之间的适配器（Adaptor），其功能就是将请求与业务逻辑分开，Action 根据客户请求调用相关的业务逻辑组件。业务逻辑由 Java Bean 或 EJB 来完成。Action 类应该侧重于控制应用程序的流程，而不是控制

应用程序的逻辑。通过将业务逻辑放在单独的 Java 包或 EJB 中，可以提高应用程序的灵活性和可重用性。

当 ActionServlet 控制器收到客户请求后，把请求转发到一个 Action 实例时。如果这个实例不存在，控制器会首先创建它，然后调用这个 Action 实例的 execute()方法。execute()方法的定义如下：

```
public ActionForward execute(ActionMapping mapping,  
    ActionForm form,  
    HttpServletRequest request,  
    HttpServletResponse response) throws IOException, ServletException ;
```

execute 方法的参数描述参见表 16-2。

表 16-2 Action 的 execute 方法的参数

参数	描述
ActionMapping	包含了这个 Action 在 struts-config.xml 文件中的描述信息（在<action>元素中定义）
ActionForm	包含了客户的表单数据
HttpServletRequest	当前的 HTTP 请求对象
HttpServletResponse	当前的 HTTP 响应对象

上面讲到一个客户请求是通过 ActionServlet 控制器处理和转发的。那么，控制器如何决定把客户请求转发给哪个 Action 对象呢？这就需要一些描述客户请求和 Action 映射关系的配置信息了。在 Struts 中，这些配置映射信息都存储在特定的 XML 文件（如 struts-config.xml）中。

这些配置信息在系统启动的时候被读入内存，供 Struts 在运行期间使用。在内存中，每一个<action>元素都对应一个 org.apache.struts.action.ActionMapping 类的实例。Action 的 execute 方法返回 ActionForward 对象，它封装了把客户请求再转发给其他 Web 组件的信息。用户定义自己的 Action 类时，必须覆盖 execute 方法。在 Action 类中该方法返回 null。

### 16.2.5 Struts 应用的流程

对于采用 Struts 框架的 Web 应用，在 Web 应用启动时就会加载并初始化控制器 ActionServlet，ActionServlet 从 struts-config.xml 文件中读取配置信息，把它们存放到 ActionMappings 对象中。

当 Servlet 容器接收到一个客户请求时，如果客户请求的 URL 为\*.do，那么首先由 ActionServlet 接收，ActionServlet 将执行如下流程。

#### 步骤

- (1) 如果 ActionForm 实例不存在，就创建一个 ActionForm 对象，把客户提交的表单数据保存到 ActionForm 对象中。
- (2) 根据配置信息决定是否需要表单验证。如果需要验证，就调用 ActionForm 的 validate()方法。
- (3) 如果 ActionForm 的 validate()方法返回 null 或返回一个不包含 ActionError 的

ActionErrors 对象，就表示表单验证成功。

(4) ActionServlet 根据配置信息决定将请求转发给哪个 Action。ActionServlet 创建一个 ActionMapping 对象，存放这个 Action 的配置信息。如果相应的 Action 实例不存在，就先创建这个实例，然后调用 Action 的 execute 方法。

(5) Action 的 execute 方法返回一个 ActionForward 对象，ActionServlet 再把客户请求转发给 ActionForward 对象指向的 JSP 组件。

(6) ActionForward 对象指向的 JSP 组件生成动态网页，返回给客户。

ActionServlet 响应客户请求的时序图如图 16-3 所示。

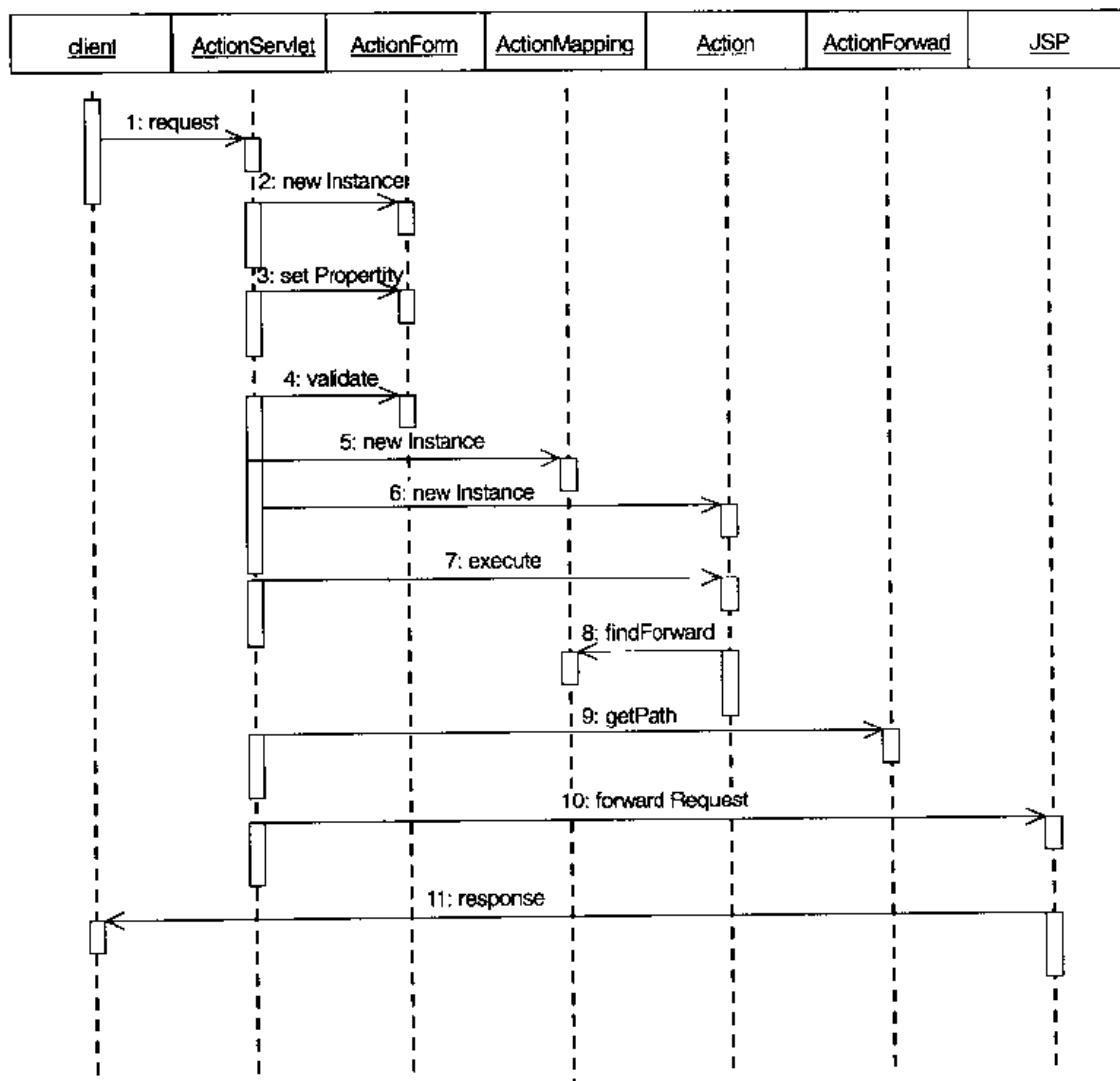


图 16-3 ActionServlet 响应客户请求的时序图

对于以上流程的步骤 (3)，如果 ActionForm 的 validate()方法返回一个包含一个或多个 ActionError 的 ActionErrors 对象，就表示表单验证失败，此时 ActionServlet 将直接把请求转发给包含客户提交表单的 JSP 组件。在这种情况下，不会再创建 Action 对象并调用 Action 的 execute 方法。

### 16.2.6 Struts 的包和标签库

整个 Struts 大约由 15 个包，近 200 个类组成，而且数量还在不断地扩展。Struts 主要的包的描述参见表 16-3。

表 16-3 Struts 主要的包

Struts 包	描 述
org.apache.struts.action	其中包含了控制整个 Struts 框架的核心类，比如 ActionServlet，以及 Action, ActionForm 和 ActionMapping 等
org.apache.struts.actions	主要作用是在客户的 HTTP 请求和业务逻辑处理之间提供特定适配器转换功能
org.apache.struts.config	提供对配置文件 struts-config.xml 中元素的映射
org.apache.struts.util	Struts 为了更好支持 web application 的应用，提供了一些常用服务的支持，比如 Connection Pool 和 Message Source
org.apache.struts.validator	Struts1.1 中增加了 validator，用于动态地配置对客户提交的 FORM 表单数据的验证

Struts 提供了一组可扩展的自定义标签库，可以简化创建用户界面的过程。目前包括：Bean 标签库、HTML 标签库、Logic 标签库、Nested 标签库和 Template 标签库。关于这些标签库的使用方法，可以从 <http://jakarta.apache.org> 查看关于 Struts 的文档。在本章的例子中，介绍了 Bean 标签库和 HTML 标签库中标签的用法。

## 16.3 创建采用 Struts 的 Web 应用

### 16.3.1 建立 Struts 的环境

为了把 Struts 运用到 Web 应用中，首先需要下载与操作系统对应的 Jakarta-Struts 二进制数文件，下载地址为 <http://jakarta.apache.org/builds>，本书配套光盘的 software 目录下提供了 jakarta-struts-20030903.zip 压缩文件。

以下是建立 Struts 环境的步骤。



- (1) 将 Struts 的压缩文件解压到本地硬盘，假定解压后 Struts 的根目录为 <Struts\_Home>。
- (2) 将 <Struts\_Home>/webapps/struts-blank.war 拷贝到 <CATALINA\_HOME>/webapps 目录下。
- (3) 将 struts-blank.war 文件改名为 helloapp-struts.war 文件。
- (4) 启动 Tomcat 服务器，当 Tomcat 服务器启动后，会看到 helloapp-struts.war 文件被自动解压到 <CATALINA\_HOME>/webapps/helloapp-struts 目录下。
- (5) 重启 Tomcat，访问 <http://localhost:8080/helloapp-struts/>，将看到如图 16-4 所示的

网页。

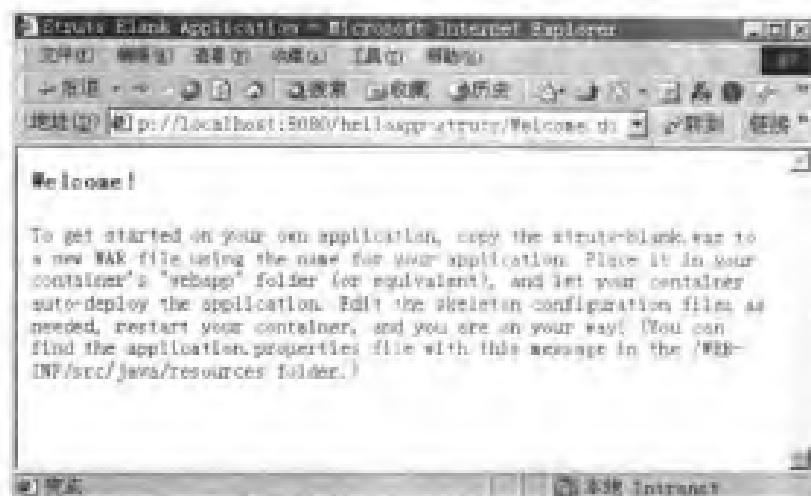


图 16-4 helloapp-struts 的默认主页

struts-blank.war 提供了基于 Struts 的 Web 应用的基本框架。在 helloapp-struts/WEB-INF 目录下，已经包含了 web.xml、struts-config.xml 和 Struts 的标签库描述文件(\*.tld)；在 helloapp-struts/WEB-INF/lib 目录下，包含了 Struts 的所有 JAR 文件。下面将在 struts-blank.war 的基础上，创建 helloapp-struts 应用。

### 16.3.2 创建视图

Struts 的视图是一组包含了 Struts 标签的 JSP 文件。在本例中，将对原来 helloapp 应用的 login.jsp 和 hello.jsp 进行改写，此外，还将创建一个 loginerror.jsp 文件。这 3 个 JSP 文件的关系如图 16-5 所示。

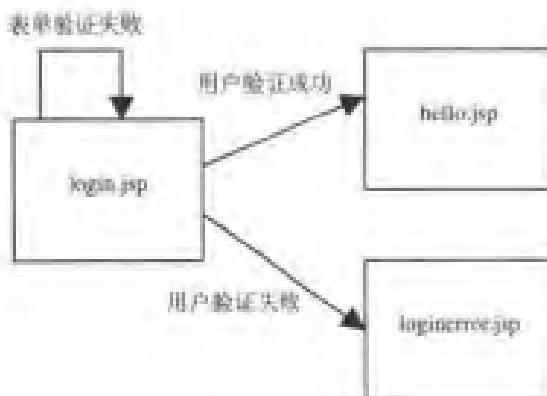


图 16-5 login.jsp、hello.jsp 和 loginerror.jsp 的关系

表单验证和用户验证在这个应用中有不同的含义，在讲解具体的程序时会解释它们。

#### 1. 修改 login.jsp 文件

以下是修改 login.jsp 文件的步骤。

**步骤**

(1) 首先，声明使用的 Struts 标签库：

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
```

第一个标签库为 struts-bean，在这个库中包含一个 message 标签，它可以从一个名为 application.properties 的文件中读取文本内容。第二个标签库为 struts-html，在这个库中包含一个 form 标签，它可以定义表单，与 ActionForm 类对应。

(2) 接下来，采用<bean:message>标签将 login.jsp 中所有的静态文本替换为动态文本。以 login.jsp 的 title 为例：

修改前：<title>helloapp</title>

修改后：<title><bean:message key="app.title"/></title>

<bean:message>标签的功能与第 14 章介绍的自定义标签<mm:message>很相似，它能够根据 key 属性从一个 application.properties 文件中读取对应的文本，并把文本内容输出到网页上。在本例中，包含这些文本信息的文件为：

```
<CATALINA_HOME>/webapps/helloapp-struts/WEB-INF/classes/resources/application.properties
```

在这个文件中加入如下内容：

```
app.title=helloapp
app.username= User Name
app.password=Password
app.hello=Welcome
app.loginerror=Your username or password are invalid.
app.loginagain=Login Again
error.username=<font color="red">Please input username</font>
```

(3) 接着，改写 login.jsp 中原来的 FORM 表单，原来 FORM 表单的代码为：

```
<form name="loginForm" method="post" action="dispatcher">
<table>
<tr>
<td><div align="right">User Name:</div></td>
<td><input type="text" name="username"></td>
</tr>
<tr>
<td><div align="right">Password:</div></td>
<td><input type="password" name="password"></td>
</tr>
<tr>
<td></td>
<td><input type="Submit" name="Submit" value="Submit"></td>
</tr>
</table>
</form>
```

采用<html:form>标签替换原有的标准<form>标签。此外，还将用<html:text>、<html:password>和<html:submit>标签替换原来标准的<input type="text">、<input type="

"password">和<input type="Submit">标签。

例程 16-1 是修改后的 login.jsp 文件。

例程 16-1 login.jsp

---

```

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
<head>
    <title><bean:message key="app.title"/></title>
</head>

<body>
<html:errors />
<html:form action="login.do" method="POST"      >
    <table >
        <tr>
            <td><bean:message key="app.username"/>:</td>
            <td><html:text property="username" /></td>
        </tr>
        <tr>
            <td><bean:message key="app.password"/>:</td>
            <td><html:password property="password" /></td>
        </tr>
        <tr>
            <td colspan="2" align="center"><html:submit /></td>
        </tr>
    </table>
</html:form>

</body>
</html>

```

---

<html:form>标签有一个属性 action，它的取值为 login.do。在上一节，已经讲过控制器 ActionServlet 在 web.xml 文件中的<servlet-mapping>配置为：

```

<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

所以，当客户提交 FORM 表单后，客户请求由 ActionServlet 来接收和处理。

在 login.jsp 中还加入了<html:errors>标签，它用于显示表单验证（FORM Validation）失败时的出错信息。

## 2. 修改 hello.jsp

修改完 login.jsp 后，对 hello.jsp 进行修改，主要是引入<bean:message>标签。以下是修

改后的 hello.jsp 的代码：

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
<head>
    <title><bean:message key="app.title" /></title>
</head>

<body>
<b><bean:message key="app.hello" />: <%= request.getAttribute("USER") %></b>
</body>
</html>
```

### 3. 创建 loginerror.jsp

当用户输入的用户名或口令没有通过用户验证时，就会转到 loginerror.jsp 网页，它显示登录失败信息，并提供了再次访问 login.jsp 的链接。以下是 loginerror.jsp 的代码：

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
<head>
    <title><bean:message key="app.title" /></title>
</head>

<body>
<b><bean:message key="app.loginerror" />
<a href="login.jsp"><bean:message key="app.loginagain" /></a></b>
</body>
</html>
```

### 16.3.3 创建模型

在 Struts 的 Model 层，可以创建 ActionForm Bean 或其他实现业务逻辑的 Java Bean 组件。在本例中，我们将创建自己的 ActionForm 类，名为 LoginForm，它的属性和 login.jsp 视图中的表单的输入参数对应。LoginForm 扩展了 ActionForm 类。

在 LoginForm 中定义了 username 和 password 两个属性，并且提供了 getXXX() 和 setXXX() 方法。例程 16-2 是 LoginForm.java 的源程序。

例程 16-2 LoginForm.java

```
package mypack;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;

public class LoginForm extends ActionForm {
```

```

private String password = null;
private String username = null;

// Password Accessors
public String getPassword() {
    return (this.password);
}
public void setPassword(String password) {
    this.password = password;
}

// Username Accessors
public String getUsername() {
    return (this.username);
}
public void setUsername(String username) {
    this.username = username;
}
// This method is called with every request. It resets the Form
// attribute prior to setting the values in the new request.
public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.password = null;
    this.username = null;
}

public ActionErrors validate(ActionMapping arg0, HttpServletRequest arg1){
    ActionErrors errors=new ActionErrors();
    if(username==null || username.equals("")) {
        errors.add("username",new ActionError("error.username"));
    }
    return errors;
}
}

```

在 LoginForm 中还提供了 validate 方法，它可以对客户提交的表单数据进行检查。在本例中，如果客户输入的 username 为空或为 null，就会创建一个 ActionError 对象，然后把它加入到 ActionErrors 对象中，最后返回 ActionErrors 对象。

编译这个类时，需要将 Struts 的 JAR 文件加入到 classpath 中，Struts 的 JAR 文件的位置为：

<CATALINA\_HOME>/webapps/helloapp-struts/WEB-INF/lib/struts.jar

发布这个 LoginForm 类时，应该在 struts-config.xml 文件的<form-beans>元素中嵌入如下<form-bean>元素：

<form-bean name="loginForm" type="mypack.LoginForm" />

struts-config.xml 文件的位置为：

<CATALINA\_HOME>/webapps/helloapp-struts/WEB-INF/struts-config.xml

### 16.3.4 创建 Action 类

下面创建一个名叫 LoginAction 的 Action 类。它的功能和原来的 DispatcherServlet 类基本相似。ActionServlet 将客户请求转发给 LoginAction，LoginAction 对客户输入的用户名和口令进行验证，然后再把请求转发给 hello.jsp 或 loginerror.jsp。例程 16-3 是 LoginAction.java 的源程序。

例程 16-3 LoginAction.java

```
package mypack;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class LoginAction extends Action {

    protected String checkUser(String username, String password) {

        String user = null;

        // You would normally do some real User lookup here, but
        // for this example we will have only one valid username "swq"
        if (username.equals("swq") && password.equals("1234")) {
            user = new String("Sun weiqin");
        }
        return user;
    }

    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {

        String user = null;

        // Default target to success
        String target = new String("success");

        if (user == null) {
```

```

// Use the LoginForm to get the request parameters
String username = ((LoginForm)form).getUsername();
String password = ((LoginForm)form).getPassword();

user = checkUser(username, password);

// Set the target to failure
if ( user == null ) {
    ((LoginForm)form).setUsername("username");
    ((LoginForm)form).setPassword("password");
    target = new String("failure");
}
else {
    request.setAttribute("USER", user);
}
// Forward to the appropriate View
return (mapping.findForward(target));
}
}

```

编译 LoginAction 时，需要将 Struts 的 JAR 文件加入到 classpath 中，Struts 的 JAR 文件的位置为：

<CATALINA\_HOME>/webapps/helloapp-struts/WEB-INF/lib/struts.jar

发布 LoginAction 时，应该在 struts-config.xml 文件的<action-mappings>元素中加入如下<action>元素：

```

<action path="/login" type="mypack.LoginAction"
name="loginForm" scope="session"
input="/login.jsp"
validate="true" >
<forward name="success" path="/hello.jsp" />
<forward name="failure" path="/loginerror.jsp" />

</action>

```

<action>的属性说明参见表 16-4。

表 16-4 <action>的属性

属性	描述
path	指定 Action 处理的 URL
type	指定 Action 的类名
name	指定 Action 处理的 ActionForm 名，与<form-bean>元素的 name 属性匹配
scope	指定 ActionForm 存在的范围
input	指定包含客户提交表单的网页，如果 ActionForm 的 validate 方法返回错误，则应该把客户请求转发到这个网页
validate	如果取值为 true，则表示 ActionServlet 应该调用 ActionForm 的 validate 方法

<action>元素的<forward>子元素用来指定当 Action 的 execute 方法执行完毕后，把客户请求再转发给哪个 Web 组件。

在 LoginAction 的 execute 方法中，首先从 LoginForm 中读取 FORM 表单数据：

```
String username = ((LoginForm)form).getUsername();
String password = ((LoginForm)form).getPassword();
```

然后调用 checkUser 方法验证用户名是否合法。在实际应用中，可能需要访问数据库来验证用户账号。本例中假定只有 swq 为合法用户，swq 的口令为 1234。如果用户验证失败，checkUser 方法返回 null。

```
user = checkUser(username, password);
```

接下来决定把客户请求转发给哪个客户组件，变量 target 的值和<action>元素的 forward 属性一致。如果用户验证失败，先把 LoginForm 中的 username 和 password 属性分别设为 username 和 password，然后给 target 赋值为 failure，此时请求转发给 loginerror.jsp；如果用户验证成功，就给 target 赋值为 success，此时请求转发给 hello.jsp。

```
// Set the target to failure
if ( user == null ) {
    ((LoginForm)form).setUsername("username");
    ((LoginForm)form).setPassword("password");
    target = new String("failure");
}
else {
    request.setAttribute("USER", user);
}
// Forward to the appropriate View
return (mapping.findForward(target));
```

## 16.4 运行 helloapp-struts 应用

按以上步骤创建好 helloapp-struts 应用后，就可以启动 Tomcat 服务器，运行 helloapp-struts 应用。在本书配套光盘的 sourcecode/chapter16/helloapp-struts 目录下，提供了这个应用的所有源文件，可以直接将整个 helloapp-struts 目录拷贝到<CATALINA\_HOME>/webapps 目录下以发布这个应用。

### 16.4.1 服务器端装载 login.jsp 的流程

Tomcat 服务器启动后，访问 <http://localhost:8080/helloapp-struts/login.jsp>，会看到如图 16-6 所示的网页。

当服务器端装载 login.jsp 网页时，流程如下。



(1) <bean:message>标签从 application.properties 文件中读取文本，并将相应的文本替换 login.jsp 中的<bean:message>标签代码。

(2) <html:form>标签查找在 session 范围中的 mypack.LoginForm Bean。如果存在这样的实例，就把 LoginForm 中的属性映射到 Form 表单的输入文本框中。由于此时还不存在 LoginForm 实例，所以忽略这项操作。

(3) 把 login.jsp 的视图呈现给客户。

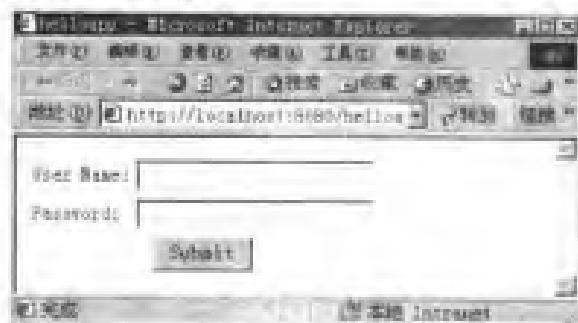


图 16-6 login.jsp 网页

#### 16.4.2 表单验证 (FORM Validation) 的流程

当浏览器端出现 login.jsp 的登录网页后，直接单击【Submit】按钮，会看到如图 16-7 所示的网页。

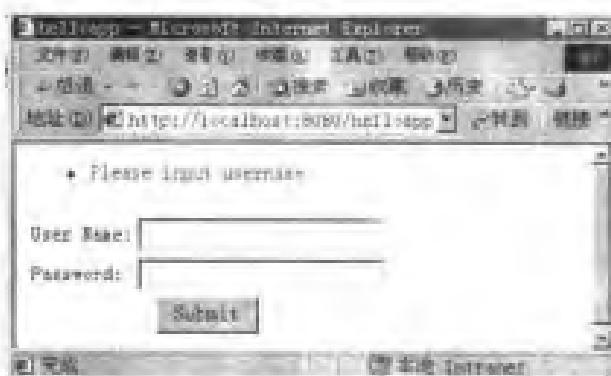


图 16-7 表单验证失败的网页

当客户提交 loginForm 表单时，服务器端对表单验证流程如下。

#### 步骤

(1) Servlet/JSP 容器在 web.xml 文件中寻找<url-pattern>为\*.do 的<servlet-mapping>元素入口，然后找到匹配的<servlet-name>。

```
<servlet-mapping>
<servlet-name> action</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

(2) Servlet/JSP 容器根据<servlet-name>从<servlet>元素中找到 ActionServlet 类，它是所有 struts 应用的控制类。

```
<servlet>
<servlet-name>action</servlet-name>
```

```
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

(3) ActionServlet 创建一个 LoginForm 对象，把客户提交的表单数据传给 LoginForm。这个 LoginForm 对象保存在 session 范围内。

(4) ActionServlet 调用 LoginForm 的 validate 方法：

```
public ActionErrors validate(ActionMapping arg0,HttpServletRequest arg1){
    ActionErrors errors=new ActionErrors();
    if(username==null || username.equals("")){
        errors.add("username",new ActionError("error.username"));
    }
    return errors;
}
```

(5) LoginForm 的 validate 方法返回一个 ActionErrors 对象，里面包含一个 ActionError 对象。

(6) ActionServlet 根据<action>元素的 input 属性，把客户请求转发给 login.jsp。

```
<action path="/login" type="mypack.LoginAction"
       name="loginForm" scope="session"
       input="/login.jsp"
       validate="true" >
    <forward name="success" path="/hello.jsp" />
    <forward name="failure" path="/loginerror.jsp" />
</action>
```

(7) login.jsp 中的<html:error>标签处理类获得 ActionError 对象，从 application.properties 文件中读取 key 为 error.username 对应的文本，然后把文本显示在网页上。在 application.properties 文件中，key 为 error.username 对应的文本为：

```
error.username=Please input username
```

### 16.4.3 用户验证失败的流程

接下来在 login.jsp 的 FORM 表单中输入用户名：abc，口令：1234，然后单击【Submit】按钮。当服务器端响应客户请求时，验证流程如下。

步骤

(1) Servlet/JSP 容器在 web.xml 文件中寻找<url-pattern>为\*.do 的<servlet-mapping>元素入口，然后找到对应的<servlet-name>。

```
<servlet-mapping>
<servlet-name> action</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

(2) Servlet/JSP 容器根据<servlet-name>从<servlet>元素中找到 ActionServlet 类，它是所有 struts 应用的控制类。

```
<servlet>
```

```
<servlet-name>action</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

(3) ActionServlet 创建一个 LoginForm 对象，把 HttpServletRequest 对象中表单数据传给 LoginForm，这个 LoginForm 对象保存在 session 范围内。

(4) ActionServlet 调用 LoginForm 的 validate 方法，这次 validate 方法返回的 ActionErrors 对象中不包含任何 ActionError 对象，表示表单验证成功。

(5) ActionServlet 在 struts-config.xml 文件中寻找<action>元素，查询条件是 path 属性为 /login：

```
<action path="/login" type="mypack.LoginAction" name="loginForm" scope="session"
input="/login.jsp">
<forward name="success" path="/hello.jsp" />
<forward name="failure" path="/login.jsp" />
</action>
```

(6) 如果 LoginAction 实例不存在，ActionServlet 就创建一个实例。ActionServlet 接着创建一个包含了以上<action>元素配置信息的 ActionMapping 对象，然后调用 LoginAction 的 execute 方法。

(7) LoginAction 在 execute 方法中执行相关的业务逻辑，由于没有通过用户验证，就把 LoginForm 的 username 和 password 属性分别设为 username 和 password。然后调用 ActionMapping.findForward 方法，参数为 failure。

```
if ( user == null ) {
    ((LoginForm)form).setUsername("username");
    ((LoginForm)form).setPassword("password");
    target = new String("failure");
}
```

(8) ActionMapping.findForward 方法从<action>元素中寻找 name 属性为 failure 的<forward>子元素。然后返回 ActionForward 对象，它代表的 URL 为 /loginerror.jsp。

(9) LoginAction 把 ActionForward 对象返回给 ActionServlet，ActionServlet 再把客户请求转发给 /loginerror.jsp。

(10) loginerror.jsp 的<bean:message>标签从 application.properties 文件中读取文本，并将相应的文本替换<bean:message>标签代码，最后生成动态网页，如图 16-8 所示。

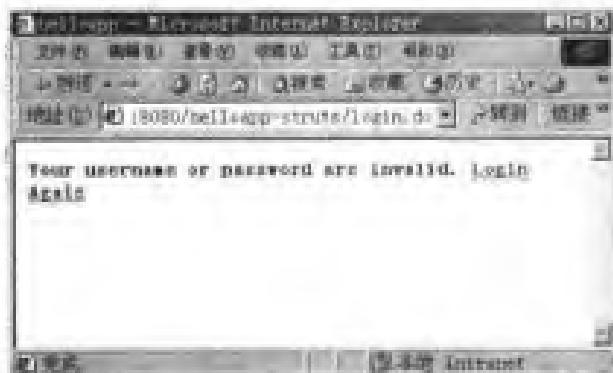


图 16-8 loginerror.jsp 网页

(11) 在 loginerror.jsp 网页中选择“Login Again”链接，客户请求转发到 login.jsp 网页。

(12) login.jsp 的<html:form>标签检查在 session 范围中的 mypack.LoginForm Bean，把 LoginForm 对象中的属性映射到 FORM 表单的输入文本框中。此时 username 的值为 username，password 的值为 password。因此，login.jsp 的网页如图 16-9 所示。

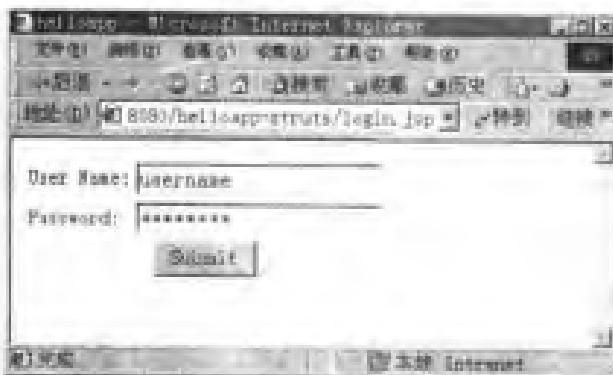


图 16-9 用户验证失败时的 login.jsp 网页

#### 16.4.4 用户验证成功的流程

接下来，在 login.jsp 的 FORM 表单中输入用户名：swq，口令：1234，然后单击【Submit】按钮。当服务器端响应客户请求时，验证流程如下。



(1) Servlet/JSP 容器在 web.xml 文件中寻找<url-pattern>为\*.do 的<servlet-mapping>元素入口，然后找到对应的<servlet-name>。

```
<servlet-mapping>
  <servlet-name> action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

(2) Servlet/JSP 容器根据<servlet-name>从<servlet>元素中找到 ActionServlet 类，它是所有 struts 应用的控制类。

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

(3) ActionServlet 创建一个 LoginForm 对象，把 HttpServletRequest 对象中表单数据传给 LoginForm。这个 LoginForm 对象保存在 session 范围内。

(4) ActionServlet 调用 LoginForm 的 validate 方法，validate 方法返回的 ActionErrors 对象中不包含任何 ActionError 对象，表示表单验证成功。

(5) ActionServlet 在 struts-config.xml 文件中寻找<action>元素，查询条件是 path 属性为/login：

```

<action path="/login" type="mypack.LoginAction" name="loginForm" scope=">session"
input="/login.jsp">
<forward name="success" path="/hello.jsp" />
<forward name="failure" path="/login.jsp" />
</action>

```

(6) 如果 LoginAction 实例不存在, ActionServlet 就创建一个实例。ActionServlet 接着创建一个包含了<action>描述信息的 ActionMapping 对象, 然后调用 LoginAction 的 execute 方法。

(7) LoginAction 在 execute 方法中执行相关的业务逻辑, 用户验证成功, 然后调用 ActionMapping.findForward 方法, 参数为 success。

(8) ActionMapping.findForward 方法从<action>元素中寻找 name 属性为 success 的<forward>子元素, 然后返回 ActionForward 对象, 它代表的 URL 为/hello.jsp。

(9) LoginAction 然后把 ActionForward 对象返回给 ActionServlet, ActionServlet 再把客户请求转发给/hello.jsp。

(10) hello.jsp 的<bean:message>标签从 application.properties 文件中读取文本, 并将相应的文本替换<bean:message>标签代码, 最后生成动态网页, 如图 16-10 所示。

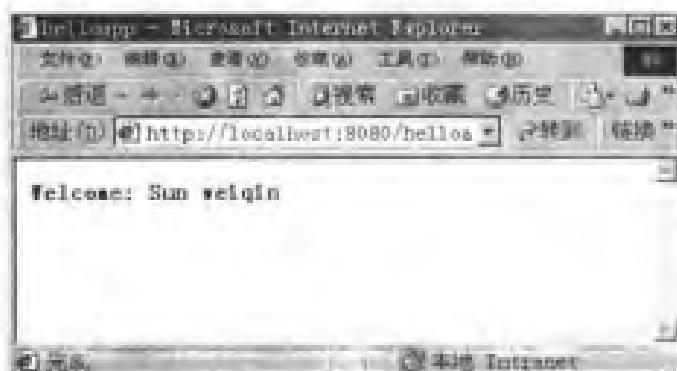


图 16-10 hello.jsp 网页

## 16.5 小结

Struts 把 MVC 设计模式运用到 Web 应用中, 它由一组相互协作的类(组件)、Servlet 以及 JSP Tag Library 组成。本章介绍了 Struts 的框架体系和工作流程。Struts 的最主要的组件包括:

- ActionServlet, 它担当控制器角色, 客户请求都通过 ActionServlet 来转发
- ActionForm, 它表示模型状态, 能够保持客户请求级或会话级的数据
- Action, 它负责处理具体的业务逻辑, 或者调用其他的 Java Bean 或 EJB 组件来完成业务逻辑

本章通过 helloapp-struts 应用介绍了在 Web 应用中使用 Struts 的方法。在运行 hello-struts 应用时, 还详细介绍了 Struts 的各个组件的工作流程, 帮助读者透彻地理解 Struts 的工作原理。

# 第 17 章 使用 Log4J 进行日志操作

Log4J 是 Apache 的一个开放源代码项目，它是一个日志操作包。通过使用 Log4J，可以指定日志信息输出的目的地，如控制台、文件、GUI 组件，甚至是套接口服务器、NT 的事件记录器和 UNIX Syslog 守护进程等；还可以控制每一条日志的输出格式。此外，通过定义日志信息的级别，能够非常细致地控制日志的输出。最令人感兴趣的是，这些功能可以通过一个配置文件来灵活地进行配置，而不需要修改应用程序的代码。

本章首先介绍 Log4J 的组成，接着介绍如何在程序中使用 Log4J。最后介绍如何在 Web 应用中通过 Log4J 生成日志。

## 17.1 Log4J 简介

在应用程序中输出日志有 3 个目的：

- 监视代码中变量的变化情况，把数据周期性地记录到文件中供其他应用进行统计分析工作
- 跟踪代码运行时轨迹，作为日后审计的依据
- 担当集成开发环境中的调试器的作用，向文件或控制台打印代码的调试信息

要在程序中输出日志，最普通的做法就是在代码中嵌入许多的打印语句，这些打印语句可以把日志输出到控制台或文件中。比较好的做法就是构造一个日志操作类来封装此类操作，而不是让一系列的打印语句充斥代码的主体。

在强调可重用组件开发的今天，除了自己从头到尾开发一个可重用的日志操作类外，Apache 为我们提供了一个强有力的现成的日志操作包 Log4J。Log4J 主要由三大组件构成：

- **Logger**: 负责生成日志，并能够对日志信息进行分类筛选，通俗地讲就是决定什么日志信息应该被输出，什么日志信息应该被忽略
- **Appender**: 定义了日志信息输出的目的地，指定日志信息应该被输出到什么地方，这些地方可以是控制台、文件、网络设备等
- **Layout**: 指定日志信息的输出格式

这 3 个组件协同工作，使得开发者能够依据日志信息类别去记录信息，并能够在程序运行期间，控制日志信息的输出格式以及日志存放地点。

一个 Logger 可以有多个 Appender，这意味着日志信息可以同时输出到多个设备上，每个 Appender 都对应一种 Layout，Layout 决定了输出日志信息的格式。

假定根据实际需要，要求程序中的日志信息既能输出到程序运行的控制台上，又能输出到指定的文件中，并且当日志信息输出到控制台时采用 SimpleLayout 布局，当日志信息输出到文件时采用 PatternLayout 布局，此时 Logger、Appender 和 Layout 3 个组件的关系如图 17-1 所示。

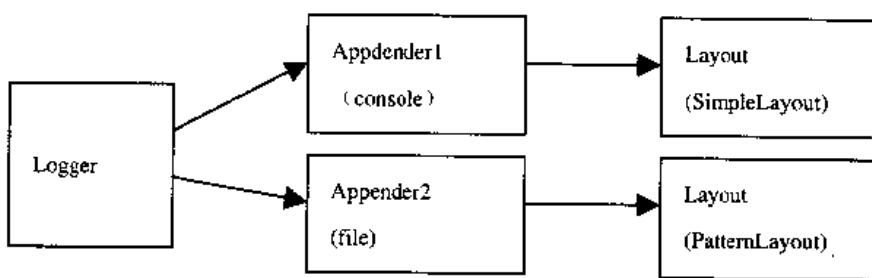


图 17-1 Logger、Appender 和 Layout 三个组件的关系

### 17.1.1 Logger 组件

Logger 是 Log4J 的核心组件，它代表了 Log4J 的日志记录器，它能够对日志信息进行分类筛选，就是决定什么日志信息应该被输出，什么日志信息应该被忽略。

Logger 组件由 org.apache.log4j.Logger 类来实现，它提供了如下方法：

```

package org.apache.log4j;
public class Logger {
    // Creation & retrieval methods:
    public static Logger getRootLogger();
    public static Logger getLogger(String name);
    // printing methods:
    public void debug(Object message);
    public void info(Object message);
    public void warn(Object message);
    public void error(Object message);
    public void fatal(Object message);

    // generic printing method:
    public void log(Priority p, Object message);
}
  
```

可以在 Log4J 的配置文件中配置自己的 Logger 组件，例如以下代码配置了一个 Logger 组件，名为 helloappLogger：

```
log4j.logger.helloappLogger=WARN
```

以上代码定义了一个 Logger 组件，名为 helloappLogger，并为它分配了一个日志级别（Priority），即“WARN”。一共有 5 种日志级别：FATAL、ERROR、WARN、INFO 和 DEBUG，其中 FATAL 的级别最高，接下来依次是 ERROR、WARN、INFO 和 DEBUG。

为什么要对日志分级别呢？试想一下，我们在写程序的时候，为了调试程序，会在很多容易出错的地方输出大量的日志信息。当程序调试完毕，不再需要输出这些日志信息了，那怎么办呢？以前的做法是把每个程序中输出日志信息的代码删除。对于大的应用程序，这种做法既费力又费时，几乎是不现实的。

Log4J 采用日志级别机制，简化了控制日志输出的步骤。获得了一个 Logger 的实例以后，可以调用以下方法之一输出日志信息：

- fatal(Object message): 输出 ERROR 级别的日志信息
- error(Object message): 输出 ERROR 级别的日志信息
- warn(Object message): 输出 WARN 级别的日志信息
- info(Object message): 输出 INFO 级别的日志信息
- debug(Object message): 输出 DEBUG 级别的日志信息
- log(Priority p, Object message): 输出参数 Priority 指定级别的日志信息

对于这些输出日志的方法，只有当它输出日志的级别大于或等于为 Logger 组件配置的日志级别时，这个方法才会被真正执行。对于以上配置的 helloappLogger，它的日志级别为 WARN，那么在程序中，它的 fatal()、error() 和 warn() 方法会被执行，而 info() 和 debug() 方法不会被执行。对于 log() 方法，只有当它的参数 Priority 指定的日志级别大于或等于 WARN 时，这个方法才会被执行。

假如不需要输出级别为 WARN 的日志信息，则可以把 helloappLogger 组件的级别调高，如调到 ERROR 或 FATAL 级别，这样 WARN 级别和以下级别的日志就不会输出了，这比修改源程序显然方便得多。

### 17.1.2 Appender 组件

Log4J 的 Appender 组件决定将日志信息输出到什么地方。目前，log4J 的 Appender 支持将日志信息输出到以下目的地：

- 控制台 (Console)
- 文件 (File)
- GUI 组件 (GUI component)
- 套接口服务器 (Remote socket server)
- NT 的事件记录器 (NT Event Logger)
- UNIX Syslog 守护进程 (Remote UNIX Syslog daemon)

一个 Logger 可以同时对应多个 Appender，也就是说，一个 Logger 的日志信息可以同时输出到多个目的地。例如，要为 helloappLogger 配置两个 Appender：一个是 file，一个是 console，则可以采用如下配置代码：

```
log4j.logger.helloappLogger=WARN,file,console
```

```
log4j.appender.file= org.apache.log4j.RollingFileAppender
log4j.appender.file.File=log.txt
```

```
log4j.appender.console= org.apache.log4j.ConsoleAppender
```

### 17.1.3 Layout 组件

Layout 组件用来决定日志的输出格式，它有以下几种类型：

- org.apache.log4j.HTMLLayout (以 HTML 表格形式布局)
- org.apache.log4j.PatternLayout (可以灵活地指定布局模式)

- org.apache.log4j.SimpleLayout (包含日志信息的级别和信息字符串)
- org.apache.log4j.TTCCLayout (包含日志产生的时间、线程和类别等信息)

SimpleLayout 仅输出日志信息级别和信息字符串, 例如, 要为名为 console 的 Appender 配置 SimpleLayout 布局, 可以采用如下配置代码:

```
log4j.appender.console.layout=org.apache.log4j.SimpleLayout
```

采用 SimpleLayout 布局, 从控制台看到的输出日志的形式如下:

```
WARN - This is a log message from the helloAppLogger
```

PatternLayout 可以让开发者依照 ConversionPattern 去定义输出格式。ConversionPattern 有点像 C 语言的 print 打印函数, 开发者可以通过一些预定义的符号来指定日志的内容和格式, 这些符号的说明参见表 17-1。

表 17-1 PatternLayout 的格式

符 号	描 述
%r	自程序开始后消耗的毫秒数
%t	表示日志记录请求生成的线程
%p	表示日志语句的优先级别
%r	与日志请求相关的类别名称
%c	日志信息所在的类名
%m%n	表示日志信息的内容

例如, 要为名为 file 的 Appender 配置 PatternLayout 布局, 可以采用如下配置代码:

```
log4j.appender.file.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.file.layout.ConversionPattern=%t %p - %m%n
```

采用以上 PatternLayout 布局, 从日志文件中看到的输出日志的形式如下:

```
THREAD-1 WARN - This is a log message from the helloAppLogger
```

以上日志内容中, “%t” 对应 “THREAD-1”, “%p” 对应 “WARN”, “%m%n” 对应后面具体的日志信息。

#### 17.1.4 Logger 组件的继承性

Log4J 提供了一个 root Logger, 它是所有 Logger 组件的“祖先”, 以下是配置 root Logger 的代码:

```
log4j.rootLogger=INFO,console
```

用户可以在配置文件中方便地配置存在继承关系的 Logger 组件, 凡是在符号 “.” 后面的 Logger 组件都会成为在符号 “.” 前面的 Logger 组件的子类。例如:

```
log4j.apache.helloappLogger=WARN
```

```
log4j.apache.helloappLogger.childLogger=file
```

对于以上配置代码, childLogger 就是 helloappLogger 的子类 Logger 组件。

Logger 组件的继承关系有以下特点:

- 如果子类 Logger 组件没有定义日志级别, 则将继承父类的日志级别
- 如果子类 Logger 组件定义了日志级别, 就不会继承父类的日志级别

- 默认情况下，子类 Logger 组件会继承父类所有的 Appender，把它们加入到自己的 Appender 清单中
- 如果把子类 Logger 组件的 additivity 标志设为 false，那么它就不会继承父类的 Appender。additivity 标志的默认值为 true

以上配置的 rootLogger、helloappLogger 和 childLogger 之间的继承关系如图 17-2 所示。

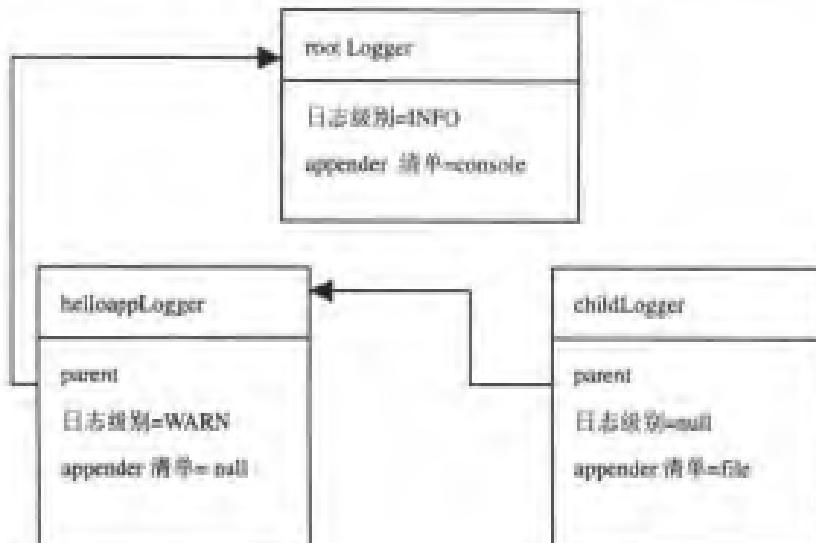


图 17-2 Logger 组件的继承关系

尽管 helloappLogger 没有配置 Appender，它继承了 rootLogger 的 console Appender。childLogger 继承了 helloappLogger 的日志级别 WARN，此外它还继承了 helloappLogger 从 rootLogger 继承而来的 console Appender。这 3 个 Logger 组件实际的日志级别和 Appender 清单参见表 17-2。

表 17-2 3 个 Logger 组件实际的日志级别和 Appender 清单

Logger 组件	日志级别	Appender 清单
rootLogger	INFO	console
helloappLogger	WARN	console (继承)
childLogger	WARN (继承)	file, console (继承)

## 17.2 Log4J 的基本使用方法

在应用程序中使用 Log4J，首先在一个配置文件中配置 Log4J 的各个组件，然后就可在程序中通过 Log4J API 来操作日志。

### 17.2.1 定义配置文件

Log4J 由 3 个重要的组件构成：Logger、Appender 和 Layout。Log4J 支持在程序中以

编程方式设置这些组件，还支持通过配置文件来配置组件，后一种方式更为灵活。

Log4J 支持两种配置文件格式：一种是 XML 格式的文件，一种是 Java 属性文件，采用“键=值”的形式。

下面介绍如何以 Java 属性文件的格式来创建 Log4J 的配置文件。

### 1. 配置 Logger 组件

如果配置 root Logger，语法为：

```
log4j.rootLogger = [priority], appenderName, appenderName, ...
```

其中，`priority` 是日志级别，可选值包括 OFF、FATAL、ERROR、WARN、INFO、DEBUG 和 ALL。通过在这里定义级别，可以控制应用程序中相应级别的日志信息的开关。比如在这里定义了 INFO 级别，则应用程序中所有 DEBUG 级别的日志信息将不被打印出来。

`appenderName` 指定 Appender 组件，用户可以同时指定多个 Appender 组件。

如果配置用户自己的 Logger 组件，语法为：

```
log4j.logger.loggerName=[priority], appenderName, appenderName, ...
```

### 2. 配置 Appender 组件

配置日志信息输出目的地 Appender，其语法为：

```
log4j.appender.appenderName = fully.qualified.name.of.appende.class
```

```
log4j.appender.appenderName.option1 = value1
```

```
...
```

```
log4j.appender.appenderName.optionN = valueN
```

Log4J 提供的 Appender 有以下几种：

- `org.apache.log4j.ConsoleAppender`（控制台）
- `org.apache.log4j.FileAppender`（文件）
- `org.apache.log4j.DailyRollingFileAppender`（每天产生一个日志文件）
- `org.apache.log4j.RollingFileAppender`（文件大小到达指定尺寸的时候产生一个新的文件）
- `org.apache.log4j.WriterAppender`（将日志信息以流格式发送到任意指定的地方）

### 3. 配置 Layout 组件

配置 Layout 组件的语法为：

```
log4j.appender.appenderName.layout = fully.qualified.name.of.layout.class
```

```
log4j.appender.appenderName.layout.option1 = value1
```

```
...
```

```
log4j.appender.appenderName.layout.optionN = valueN
```

Log4J 提供的 Layout 有以下几种：

- `org.apache.log4j.HTMLLayout`（以 HTML 表格形式布局）
- `org.apache.log4j.PatternLayout`（可以灵活地指定布局模式）
- `org.apache.log4j.SimpleLayout`（包含日志信息的级别和信息字符串）
- `org.apache.log4j.TTCCLayout`（包含日志产生的时间、线程和类别等信息）

例如，可以根据图 17-2 创建如下的配置文件，文件名为 properties.lcf：

```
## LOGGERS ##
```

```

#configure root logger
log4j.rootLogger=INFO,console
#define a logger named helloAppLogger
log4j.logger.helloappLogger=WARN
#define a second logger that is a child to helloAppLogger
log4j.logger.helloappLogger.childLogger=file

## APPENDERS ##
# define an appender named console, which is set to be a ConsoleAppender
log4j.appenders.console=org.apache.log4j.ConsoleAppender

# define an appender named file, which is set to be a RollingFileAppender
log4j.appenders.file=org.apache.log4j.RollingFileAppender
log4j.appenders.file.File=log.txt

## LAYOUTS ##
# assign a SimpleLayout to console appender
log4j.appenders.console.layout=org.apache.log4j.SimpleLayout

# assign a PatternLayout to file appender
log4j.appenders.file.layout=org.apache.log4j.PatternLayout
log4j.appenders.file.layout.ConversionPattern=%t %p - %m%n

```

### 17.2.2 在程序中使用 Log4J

在程序中访问 Log4J，需要用到 Log4J 的 JAR 文件，可以到 <http://jakarta.apache.org/log4j> 地址下载。下载了 Log4J 的压缩文件后，应该把它解压到本地硬盘，在解压后的目录中，会找到 Log4J 的 JAR 文件。本书光盘的 lib 目录下提供了 log4j-1.2.8.jar 文件，也可以直接使用这个文件。

程序中使用 Log4J 包含以下过程：

- 获得日志记录器
- 读取配置文件，配置 Log4J 环境
- 输出日志信息

#### 1. 获得日志记录器

使用 Log4J，第一步就是获取日志记录器，这个记录器负责控制日志信息的输出。如果要获得 root Logger，可以调用 Logger 类的静态方法 getRootLogger()：

```
Logger rootLogger=Logger.getRootLogger();
```

如果要取得用户自定义的 Logger，可以调用 Logger 的静态方法 getLogger(String name)：

```
Logger helloappLogger = Logger.getLogger ("log4j.logger.helloappLogger");
```

#### 2. 读取配置文件，配置 Log4J 环境

当获得了日志记录器之后，第二步就是读取配置文件，配置 Log4J 环境，有 3 种方法：

- BasicConfigurator.configure(): 自动快速地使用默认 Log4J 环境

- PropertyConfigurator.configure( String configFilename ): 读取使用 Java 属性格式的配置文件并配置 Log4J 环境
- DOMConfigurator.configure( String filename ): 读取 XML 形式的配置文件并配置 Log4J 环境

### 3. 插入日志信息

在以上两个必要步骤执行完毕后，就可以在程序代码中需要生成日志的地方，调用 Logger 的各种输出日志方法来输出不同级别的日志，例如：

```
helloappLogger.warn("This is a log message from the " + helloappLogger.getName());
```

例程 17-1 是一个使用 Log4J 的程序，程序名为 Log4JApp.java。

例程 17-1 Log4JApp.java

```
import org.apache.log4j.Logger;
import org.apache.log4j.Priority;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.PropertyConfigurator;

public class Log4JApp {

    // Get an instance of the helloappLogger
    static Logger helloappLogger =
        Logger.getLogger("helloappLogger");
    // Get an instance of the childLogger
    static Logger childLogger =
        Logger.getLogger("helloappLogger.childLogger");

    public static void main(String[] args) {

        // Load the properties using the PropertyConfigurator
        PropertyConfigurator.configure("properties.lcf");

        // Log Messages using the Parent Logger
        helloappLogger.debug("This is a log message from the " +
            helloappLogger.getName());
        helloappLogger.info("This is a log message from the " +
            helloappLogger.getName());
        helloappLogger.warn("This is a log message from the " +
            helloappLogger.getName());
        helloappLogger.error("This is a log message from the " +
            helloappLogger.getName());
        helloappLogger.fatal("This is a log message from the " +
            helloappLogger.getName());

        // Log Messages using the Child Logger
    }
}
```

```

        childLogger.debug("This is a log message from the " +
            childLogger.getName());
        childLogger.info("This is a log message from the " +
            childLogger.getName());
        childLogger.warn("This is a log message from the " +
            childLogger.getName());
        childLogger.error("This is a log message from the " +
            childLogger.getName());
        childLogger.fatal("This is a log message from the " +
            childLogger.getName());
    }
}

```

---

编译和运行这个程序时，需要将 Log4J 的 JAR 文件设置为 classpath，并且把 Log4J 的配置文件 properties.lcf 拷贝到 Log4JApp.class 所在的目录，然后就可以在 DOS 控制台运行这个程序了，命令为：

```
java -classpath log4j-1.2.8.jar Log4JApp
```

程序运行完毕，会在 DOS 控制台看到以下输出内容：

```

WARN - This is a log message from the helloappLogger
ERROR - This is a log message from the helloappLogger
FATAL - This is a log message from the helloappLogger
WARN - This is a log message from the helloappLogger.childLogger
ERROR - This is a log message from the helloappLogger.childLogger
FATAL - This is a log message from the helloappLogger.childLogger

```

此外，在 Log4JApp.class 所在的目录下会看到一个 log.txt 文件，内容如下：

```

main WARN - This is a log message from the helloappLogger.childLogger
main ERROR - This is a log message from the helloappLogger.childLogger
main FATAL - This is a log message from the helloappLogger.childLogger

```

如果在 properties.lcf 文件中将

```
log4j.logger.helloappLogger.childLogger=file
```

改为：

```
log4j.logger.helloappLogger.childLogger=,console,file
```

然后再运行以上程序，会在 DOS 控制台看到以下输出内容：

```

WARN - This is a log message from the helloappLogger
ERROR - This is a log message from the helloappLogger
FATAL - This is a log message from the helloappLogger
WARN - This is a log message from the helloappLogger.childLogger
WARN - This is a log message from the helloappLogger.childLogger
ERROR - This is a log message from the helloappLogger.childLogger
ERROR - This is a log message from the helloappLogger.childLogger
FATAL - This is a log message from the helloappLogger.childLogger
FATAL - This is a log message from the helloappLogger.childLogger

```

此时，childLogger 的日志在控制台上输出了两次，这是因为 childLogger 继承了父类的 console Appender，同时它本身又定义了一个 console Appender，因而它有两个 console Appender。

## 17.3 在 helloapp 应用中使用 Log4J

如果在 Web 应用中使用 Log4J，则可以创建一个 Servlet，在它的初始化方法中读取 Log4J 的配置文件并且配置 Log4J 环境，这个 Servlet 在 Web 应用启动的时候就被加载和初始化了，然后就可以在其他的 Web 组件中获取 Logger 对象并输出日志。

### 17.3.1 创建用于配置 Log4J 环境的 Servlet

应将配置 Log4J 环境的代码放在 Servlet 的 init()方法中，这样可以保证当 Servlet 被加载并初始化后，Log4J 的环境就已经配置好了。例程 17-2 是这个 Servlet 的源程序。

例程 17-2 Log4JServlet.java

```
package mypack;

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

import org.apache.log4j.PropertyConfigurator;

public class Log4JServlet extends HttpServlet {

    public void init()
        throws ServletException {

        // Get Fully Qualified Path to Properties File
        String path = getServletContext().getRealPath("/");
        String propfile = path + getInitParameter("propfile");

        // Initialize Properties for All Servlets
        PropertyConfigurator.configure(propfile);
    }

}
```

Log4JServlet 通过 getInitParameter("propfile")方法从 web.xml 文件中读取初始化参数 propfile，它代表了 Log4J 配置文件的名字。Log4JServlet 在 web.xml 文件中的配置代码如下：

```
<servlet>
    <servlet-name>log4j</servlet-name>
    <servlet-class>mypack.Log4JServlet</servlet-class>
```

```

<init-param>
    <param-name>propfile</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>

```

### 17.3.2 在 login.jsp 中输出日志

如果要在 login.jsp 中输出日志，首先要将 org.apache.log4j.Logger 类引入：

```
<%@ page import="org.apache.log4j.Logger" %>
```

然后就可以在<% ... %>代码块中取得 Logger 对象并输出日志。例程 17-3 是修改后的 login.jsp 的源代码。

例程 17-3 login.jsp

---

```

<%@ page import="org.apache.log4j.Logger" %>

<html>
    <head>
        <title>helloapp</title>
    </head>
    <body >
        <%
            Logger helloappLogger =Logger.getLogger("helloappLogger");
            // Log Messages using the Parent Logger
            helloappLogger.debug("This is a log message from the " + helloappLogger.getName());
            helloappLogger.info("This is a log message from the " + helloappLogger.getName());
            helloappLogger.warn("This is a log message from the " + helloappLogger.getName());
            helloappLogger.error("This is a log message from the " + helloappLogger.getName());
            helloappLogger.fatal("This is a log message from the " + helloappLogger.getName());
        %>
        <br>
        <form name="loginForm" method="post" action="dispatcher">
            <table>
                <tr><td><div align="right">User Name:</div></td><td><input type="text"
                    name="username"></td></tr>
                <tr><td><div align="right">Password:</div></td><td><input type="password"
                    name="password"></td></tr>
                <tr><td></td><td><input type="Submit" name="Submit" value="Submit"></td></tr>
            </table>
        </form>
    </body>
</html>

```

---

### 17.3.3 发布和运行使用 Log4J 的 helloapp 应用

发布和运行使用 Log4J 的 helloapp 应用的步骤如下。



(1) 将 Log4J 的 JAR 文件拷贝到以下目录:

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/lib

(2) 创建 Log4J 的配置文件 log4j.properties, 它的存放目录为:

<CATALINA\_HOME>/webapps/helloapp/WEB-INF

log4j.properties 文件的内容如下:

```
#define a logger named helloAppLogger
log4j.logger.helloappLogger=WARN,console,file

## APPENDERS ##
# define an appender named console, which is set to be a ConsoleAppender
log4j.appenders.console=org.apache.log4j.ConsoleAppender

# define an appender named file, which is set to be a RollingFileAppender
log4j.appenders.file=org.apache.log4j.RollingFileAppender
log4j.appenders.file.File=C:/jakarta-tomcat/webapps/helloapp/WEB-INF/log.txt

## LAYOUTS ##
# assign a SimpleLayout to console appender
log4j.appenders.console.layout=org.apache.log4j.SimpleLayout

# assign a PatternLayout to file appender
log4j.appenders.file.layout=org.apache.log4j.PatternLayout
log4j.appenders.file.layout.ConversionPattern=%t %p - %m%n
```

**提示**

应将以上 log4j.appenders.file.File 属性的 C:/jakarta-tomcat 替换为 Tomcat 服务器的安装目录。此外，不论在哪种操作系统中，指定 log4j.appenders.file.File 属性都可以使用“/”文件分割符。如果要在 Windows NT/2000 下使用“\”文件分割符，应采用如下形式:

log4j.appenders.file.File=C:\\jakarta-tomcat\\webapps\\helloapp\\WEB-INF\\log.txt

(3) 编译 Log4JServlet，编译生成的类的存放位置为:

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/classes/mypack

(4) 修改 web.xml，为 Log4J servlet 加入<servlet>元素:

```
<servlet>
  <servlet-name>log4j</servlet-name>
  <servlet-class>mypack.Log4JServlet</servlet-class>
  <init-param>
```

```
<param-name>propfile</param-name>
<param-value>/WEB-INF/log4j.properties</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
```

(5) 启动 Tomcat 服务器，访问 <http://localhost:8080/helloapp/login.jsp>，会在 Tomcat 服务器的控制台看到如下日志：

```
WARN - This is a log message from the helloappLogger
ERROR - This is a log message from the helloappLogger
FATAL - This is a log message from the helloappLogger
```

在<CATALINA\_HOME>/webapps/helloapp/WEB-INF 目录下，会看到一个 log.txt 文件，内容如下：

```
http8080-Processor25 WARN - This is a log message from the helloappLogger
http8080-Processor25 ERROR - This is a log message from the helloappLogger
http8080-Processor25 FATAL - This is a log message from the helloappLogger
```

在本书配套光盘的 chapter17/helloapp 目录下是使用 Log4J 的完整 helloapp 应用，只要按以上步骤（2）替换 log4j.properties 文件中 log4j.appender.file.File 属性，然后把整个目录拷贝到<CATALINA\_HOME>/webapps 目录下，就可以运行这个应用了。

## 17.4 小结

Log4J 主要由 3 大组件构成：Logger、Appender 和 Layout。Logger 控制日志信息的输出；Appender 决定日志信息的输出目的地；Layout 决定日志信息的输出格式。Log4J 允许用户在配置文件中灵活地配置这些组件。在程序中使用 Log4J 非常方便，只要先取得日志记录器，然后读取配置文件并配置 Log4J 环境，接下来就可以在程序中任何需要输出日志的地方，调用 Logger 类的适当方法来生成日志。

# 第 18 章 Tomcat 与 Jboss 集成

Jboss 支持 Sun EJB 1.1 和 EJB 2.0 的规范，它是一个免费的 EJB 容器，类似于 SUN 公司的 Java 2 Platform, Enterprise Edition (J2EE) 服务器软件。但是 Jboss 只提供了 EJB 容器的功能，而不能用做 Servlet/JSP 容器。Jboss 支持与 Tomcat 集成，二者协同工作，构成完整的 J2EE 服务器。

Jboss 的一个显著优点是需要比较小的内存和硬盘空间，可以在 64MB 内存以及几兆空间上运行得很好。而 SUN 的 J2EE 软件最少需要内存为 128MB，以及 31MB 硬盘空间。Jboss 启动速度要比 SUN 的 J2EE 快 10 倍。

本章首先介绍 J2EE 的体系结构，然后以 bookstore 应用为例，介绍开发 EJB 组件的过程，最后讲解如何在 Jboss 和 Tomcat 的整合服务器上部署 J2EE 应用。

## 18.1 安装 Jboss 和 Tomcat 整合服务器

Jboss 是一个纯 Java 软件，它的运行需要 JDK，因此在安装 Jboss 前应该先安装好 JDK。建议安装 JDK 1.3 以上的版本，并且在系统环境变量中加入 JAVA\_HOME 变量，它的取值为 JDK 的安装目录。

Jboss 的下载地址为 <http://www.jboss.org/>，可以下载 Jboss 和 Tomcat 集成的最新软件包。在本书配套光盘的 software 目录下提供了 jboss-3.2.1\_tomcat-4.1.24.zip 文件，它是 Jboss 3.2.1 版本和 Tomcat 4.1.24 版本集成的软件包，也可以使用这个文件。

Jboss 和 Tomcat 集成的软件包是个压缩文件，应该把这个压缩文件解压到本地硬盘（例如，把它解压到 C:\jboss-tomcat 目录），假定 Jboss 的根目录为<JBOSS\_HOME>。

接下来运行<JBOSS\_HOME>/bin/run.bat，这个命令同时启动 Jboss 服务器和 Tomcat 服务器。Tomcat 的端口为 8080，Jboss 的端口为 8083。

通过浏览器访问 <http://localhost:8080/>，出现如下错误：

Apache Tomcat/4.1.24 - HTTP Status 500 - No Context configured to process this request  
这是正常的，因为在这个路径下还没有发布 Web 应用。

访问 <http://localhost:8080/jmx-console>，将出现如图 18-1 所示的页面，这是因为在<JBOSS\_HOME>/server/default/deploy 目录下，已经发布了一个 Web 应用，相应的打包文件为 jmx-console.war。

再访问 [http://localhost:8083/](http://localhost:8083)，将出现无错误的空白页。

如果访问上述 3 个 URL，看到的结果和本书描述相同，就说明 Jboss 已经安装成功。

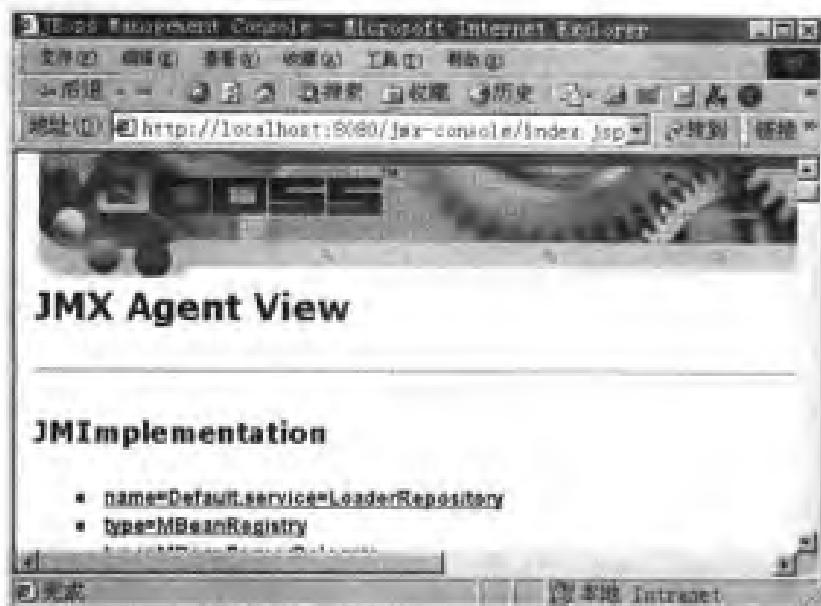


图 18-1 jmx-console 应用的主页

## 18.2 J2EE 体系结构简介

如今，人们对分布式的软件应用系统提出了更高的要求。软件开发人员致力于提高服务器端的运行速度、安全性和可靠性。在电子商务和信息技术领域，设计、开发软件应用应该建立在低成本、高效率和占用资源少的基础上。

Java 2 Platform, Enterprise Edition (J2EE) 技术提供了以组件为基础来设计、开发、组装和发布企业应用的方法，它能够有效降低开发软件的成本，并且提高开发速度。J2EE 平台提供了多层次的分布式的应用模型，应用逻辑根据不同的功能由不同的组件来实现。一个 J2EE 应用由多种组件组合而成，这些组件安装在不同的机器上，组件分布在哪台机器上，是根据组件在 J2EE 体系结构中所处的层次来决定的。

一个多层次的 J2EE 应用结构如图 18-2 所示，它包含如下 4 个层次：

- 客户层组件运行在客户机器上
- Web 层组件运行在 J2EE 服务器上
- 业务层组件运行在 J2EE 服务器上
- Enterprise Information System (EIS) 层运行在数据库服务器上

图 18-2 中的 4 个层如果按照它们在机器上的分布来划分，可以分为 3 层：客户层、J2EE 服务器层及企业信息系统所在的数据库服务器层，这就是通常所说的三层应用结构。三层应用结构扩展了标准的两层应用结构，前者在客户层和数据库服务器层之间，增加了一个多线层的应用服务器。

Enterprise Java Bean (简称 EJB) 组件是应用服务器方的组件，它包含了企业应用的业务逻辑。在运行环境中，企业应用客户通过调用 EJB 组件的方法来执行业务。

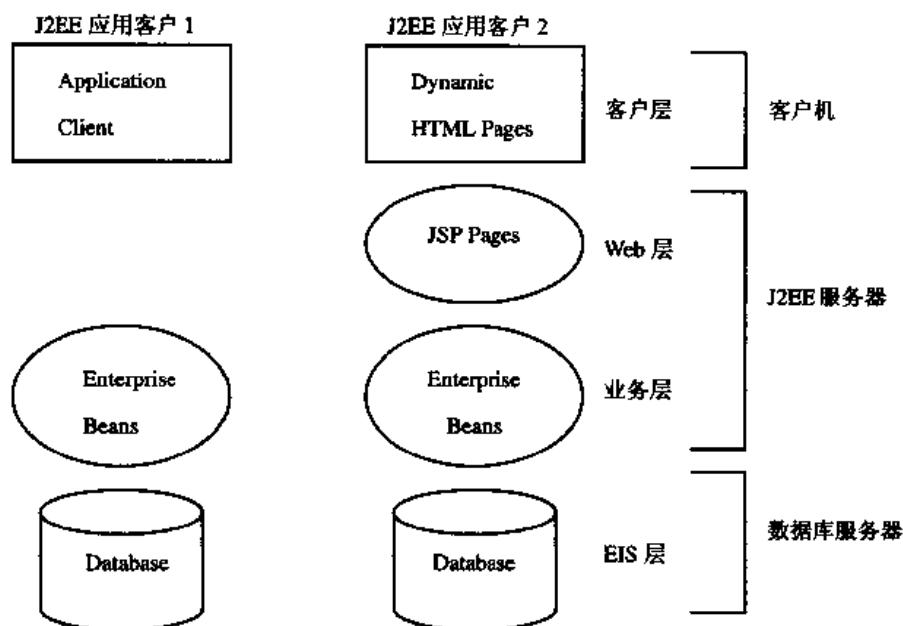


图 18-2 J2EE 多层应用结构

EJB 分两种类型：

- 会话 Bean：实现会话中的业务逻辑
- 实体 Bean：实现一个业务实体

会话 Bean 又有两种类型：

- 有状态会话 Bean：有状态会话 Bean 的实例始终与一个特定的客户关联，它的实例变量可以维护特定客户的状态
- 无状态会话 Bean：无状态会话 Bean 的实例不与特定的客户关联，它的实例变量不能始终代表特定客户的状态

在本章，将创建一个基于 J2EE 的 bookstore 应用，它包含一个无状态会话 Bean，名为 BookDBEJB。新的 bookstore 应用的体系结构如图 18-3 所示。

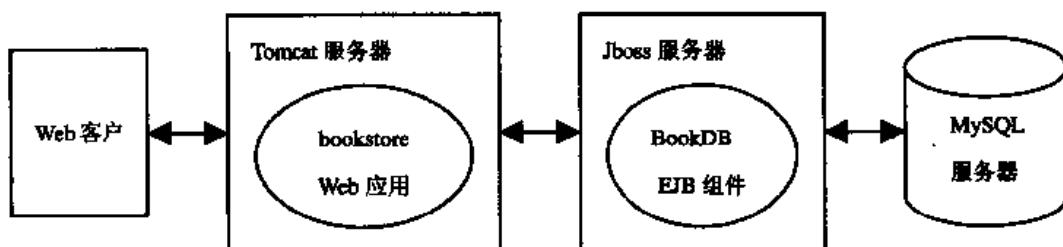


图 18-3 基于 J2EE 的 bookstore 应用的体系结构

在本书第 5 章介绍的 bookstore 应用中，业务逻辑是由 BookDB Java Bean 组件来实现的，这个 Java Bean 组件和 Web 应用都运行在 Tomcat 服务器中。采用 J2EE 结构后，业务逻辑由 BookDB Enterprise Java Bean 来实现，这个 EJB 组件运行在 Jboss 服务器上。

本书配套光盘的 /sourcecode/bookstores/version3/bookstore 目录下提供了本章介绍的 bookstore 应用的源文件。

## 18.3 创建 EJB 组件

在范例中，将创建一个无状态的会话 Bean，名为 BookDBEJB。它将取代原来的 BookDB Java Bean，负责操纵数据库。

一个 EJB 至少需要生成 3 个 Java 文件：Remote 接口、Home 接口和 Enterprise Bean 类。

本例中 BookDBEJB 的 3 个 Java 文件分别为：

- BookDBEJB.java: Remote 接口
- BookDBEJHome.java: Home 接口
- BookDBEJBImpl.java: Enterprise Bean 类

### 18.3.1 编写 Remote 接口

Remote 接口中定义了客户可以调用的业务方法。这些业务方法在 Enterprise Bean 类中实现。以下是 Remote 接口 BookDBEJB.java 的代码：

```
package mypack;
import java.util.*;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface BookDBEJB extends EJBObject {
    public BookDetails getBookDetails(String bookId) throws RemoteException;
    public int getNumberOfBooks() throws RemoteException;
    public Collection getBooks() throws RemoteException;
    public void buyBooks(ShoppingCart cart) throws RemoteException;
}
```

当客户程序访问 EJB 组件的业务方法时，这些方法的参数以及返回值都会在网络上传输，如图 18-4 所示。Sun EJB 规范规定，如果在 Remote 接口中声明的方法的参数类型或返回类型为类，那么这个类必须实现 java.io.Serializable 接口。以上代码中，getBookDetails 方法返回类型为 BookDetails 类，buyBooks 方法的参数类型为 ShoppingCart 类。因此，必须修改 BookDetails 和 ShoppingCart 类的声明，确保它们都实现了 Serializable 接口。此外，在一个 ShoppingCart 对象中会包含多个 ShoppingCartItem 对象，ShoppingCartItem 对象也会作为参数的一部分在网络上传输。因此，ShoppoingCartItem 也必须实现 Serializable 接口。

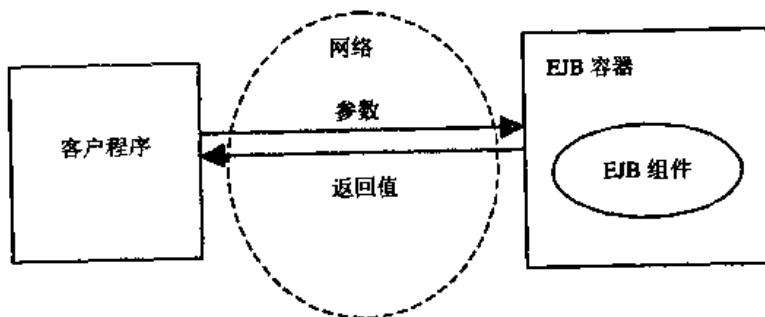


图 18-4 客户程序访问 EJB 的业务方法

### 18.3.2 编写 Home 接口

Home 接口定义了创建、查找和删除 EJB 的方法。本例中的 BookDBEJBHome 接口包含了一个 create 方法，这个方法返回一个 BookDBEJB 对象的远程引用。以下是 BookDBEJBHome 的代码：

```
package mypack;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.EJBHome;

public interface BookDBEJBHome extends EJBHome {
    BookDBEJB create() throws RemoteException, CreateException;
}
```

### 18.3.3 编写 Enterprise Java Bean 类

本例中的 Enterprise Java Bean 名为 BookDBEJBImpl，它实现了远程接口 BookDBEJB 中定义的业务方法。例程 18-1 是 BookDBEJBImpl 类的源代码。

例程 18-1 BookDBEJBImpl.java

---

```
package mypack;

import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;

public class BookDBEJBImpl implements SessionBean {
    private ArrayList books = null;
    private Connection con = null;
    private String dbUrl = "jdbc:mysql://localhost:3306/BookDB";
    private String dbUser="dbuser";
    private String dbPwd="1234";

    // implementation of create and remove remote methods

    public void ejbCreate() throws CreateException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            con = java.sql.DriverManager.getConnection(dbUrl, dbUser, dbPwd);
        }
    }
}
```

```
        } catch (Exception ex) {
            throw new CreateException("Couldn't create bean: " + ex.getMessage());
        }
        books = new ArrayList();
    }

    public void ejbRemove() throws EJBException {
        try {
            con.close();
        } catch (SQLException ex) {
            throw new EJBException("unsetEntityContext: " + ex.getMessage());
        }
        con = null;
        books = null;
    }

    public BookDBEJBImpl() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

    // remote methods

    public int getNumberOfBooks() {
        books = new ArrayList();
        try {
            String selectStatement = "select * " + "from books";
            PreparedStatement prepStmt = con.prepareStatement(selectStatement);
            ResultSet rs = prepStmt.executeQuery();

            while (rs.next()) {
                BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
                    rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
                if (rs.getInt(8) > 0)
                    books.add(bd);
            }
            prepStmt.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
        return books.size();
    }

    public Collection getBooks() {
```

```

books = new ArrayList();
try {
    String selectStatement = "select * " + "from books";
    PreparedStatement prepStmt = con.prepareStatement(selectStatement);
    ResultSet rs = prepStmt.executeQuery();

    while (rs.next()) {

        BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
            rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
        books.add(bd);
    }

    prepStmt.close();
} catch (SQLException ex) {
    ex.printStackTrace();
}

Collections.sort(books);
return books;
}

private String bookId;
public void setBookId(String bookId){
    this.bookId=bookId;
}
public BookDetails getBookDetails()  {
    return getBookDetails(bookId);
}
public BookDetails getBookDetails(String bookId)  {
    try {
        String selectStatement = "select * " + "from books where id = ? ";
        PreparedStatement prepStmt = con.prepareStatement(selectStatement);
        prepStmt.setString(1, bookId);
        ResultSet rs = prepStmt.executeQuery();

        if (rs.next()) {
            BookDetails bd = new BookDetails(rs.getString(1), rs.getString(2), rs.getString(3),
                rs.getFloat(4), rs.getInt(5), rs.getString(6), rs.getInt(7));
            prepStmt.close();
            return bd;
        }
        else {
            prepStmt.close();
            return null;
        }
    }
}

```

```
        } catch (SQLException ex) {
            return null;
        }
    }

public void buyBooks(ShoppingCart cart) {
    Collection items = cart.getItems();
    Iterator i = items.iterator();
    try {
        con.setAutoCommit(false);
        while (i.hasNext()) {
            ShoppingCartItem sci = (ShoppingCartItem)i.next();
            BookDetails bd = (BookDetails)sci.getItem();
            String id = bd.getBookId();
            int quantity = sci.getQuantity();
            buyBook(id, quantity);
        }
        con.commit();
        con.setAutoCommit(true);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public boolean buyBook(String bookId, int quantity) {
    try {
        String selectStatement = "select * " + "from books where id = ? ";
        PreparedStatement prepStmt = con.prepareStatement(selectStatement);
        prepStmt.setString(1, bookId);
        ResultSet rs = prepStmt.executeQuery();
        if (rs.next()) {
            prepStmt.close();
            String updateStatement =
                "update books set saleamount = saleamount + ? where id = ?";
            prepStmt = con.prepareStatement(updateStatement);
            prepStmt.setInt(1, quantity);
            prepStmt.setString(2, bookId);
            prepStmt.executeUpdate();
            prepStmt.close();
        }
    } catch (Exception ex) {ex.printStackTrace();}
    return false;
}
```

}

当 EJB 容器创建一个 EJB 实例时，会调用 Enterprise Java Bean 类的 ejbCreate 方法，在 BookDBEJBImpl 的 ejbCreate 方法中负责创建与 MySQL 数据库的连接。对于所有的数据库操作，都使用同一个数据库连接对象。为了简化起见，这里采用 JDBC 驱动程序访问数据库，没有使用数据源。

```
private Connection con = null;
private String dbUrl = "jdbc:mysql://localhost:3306/BookDB";
private String dbUser="dbuser";
private String dbPwd="1234";

public void ejbCreate() throws CreateException {
    try {
        Class.forName("com.mysql.jdbc.Driver");
        con = java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);

    } catch (Exception ex) {
        throw new CreateException("Couldn't create bean: " + ex.getMessage());
    }
    books = new ArrayList();
}
```

## 18.4 在 Web 应用中访问 EJB 组件

在原来的 bookstore 应用的 common.jsp 中定义了如下 Java Bean:

```
<jsp:useBean id="bookDB" scope="application" class="mypack.BookDB"/>
```

现在，用 BookDBEJB 替换原来的 BookDB Java Bean。BookDBEJB 组件运行在 EJB 容器中，是一种 JNDI 资源。在 Web 应用中，无法创建 BookDBEJB 组件，而应该首先查找名为 ejb/BookDBEJB 的 JNDI 资源，获得该资源的引用：

```
InitialContext ic = new InitialContext();
Object objRef = ic.lookup("java:comp/env/ejb/BookDBEJB");
```

然后把它转换为 BookDBEJBHome 类型：

```
BookDBEJBHome home = (BookDBEJBHome)PortableRemoteObject.narrow(objRef,
    mypack.BookDBEJBHome.class);
```

接下来调用 BookDBEJBHome 的 create 方法。此时，EJB 容器会创建 BookDBEJBImpl 的实例，并调用它的 ejbCreate 方法，然后返回 BookDBEJB 组件的远程引用。

```
bookDB = home.create();
```

得到了 BookDBEJB 组件的远程引用后，就可以把它保存到 ServletContext 中，作为 bookstore Web 应用的共享资源。

```
getServletContext().setAttribute("bookDB", bookDB);
```

例程 18-2 是 common.jsp 的源代码。

## 例程 18-2 common.jsp

```
<%@ page import="mypack.*" %>
<%@ page import="java.util.Properties" %>
<%@ page errorPage="errorpage.jsp" %>
<%@ page import="javax.ejb.* , javax.naming.* , javax.rmi.PortableRemoteObject,
java.rmi.RemoteException" %>
<%!
    private BookDBEJB bookDB;

    public void jsplninit() {

        bookDB =
            (BookDBEJB)getServletContext().getAttribute("bookDB");

        if (bookDB == null) {
            try {
                InitialContext ic = new InitialContext();
                Object objRef = ic.lookup("java:comp/env/ejb/BookDBEJB");
                BookDBEJBHome home = (BookDBEJBHome)PortableRemoteObject.narrow(objRef,
                    mypack.BookDBEJBHome.class);
                bookDB = home.create();
                getServletContext().setAttribute("bookDB", bookDB);
            } catch (RemoteException ex) {
                System.out.println("Couldn't create database bean." + ex.getMessage());
            } catch (CreateException ex) {
                System.out.println("Couldn't create database bean." + ex.getMessage());
            } catch (NamingException ex) {
                System.out.println("Unable to lookup home: " + "java:comp/env/ejb/BookDBEJB." +
                    ex.getMessage());
            }
        }
    }

    public void jspDestroy() {
        bookDB = null;
    }

    public String convert(String s){
        try{
            return new String(s.getBytes("ISO-8859-1"),"GB2312");
        }catch(Exception e){return null;}
    }
%>
```

除了修改 common.jsp，不需要修改任何其他的 JSP 文件。当其他的 JSP 组件调用 BookDBEJB 的业务方法时，使用的 Java 代码和原来一样，例如在 bookdetails.jsp 中调用了

bookDB 的 getBookDetails()方法:

```
<%
    //Get the identifier of the book to display
    String bookId = request.getParameter("bookId");
    if(bookId==null)bookId="201";
    BookDetails book = bookDB.getBookDetails(bookId);
%>
```

对于原来的 bookstore Web 应用, bookDB 代表的是 Java Bean 组件, 它运行在 Tomcat 服务器上, 所以 getBookDetails 方法在 Tomcat 服务器上执行。在本例中, bookDB 代表 BookDBEJB 组件, 它运行在 Jboss 服务器上, 所以 getBookDetails 方法在 Jboss 服务器上执行。

## 18.5 发布 J2EE 应用

在第 2 章中, 讲过在发布一个 Web 应用时, 可以把它打包为 WAR 文件。如果单独发布一个 EJB 组件, 应该把它打包为 JAR 文件。对于一个 J2EE 应用, 在发布时, 应该把它打包为 EAR 文件。

在 Jboss 中, 发布 J2EE 组件的目录为 <JBOSS\_HOME>/server/default/deploy。Jboss 服务器具有热部署功能。所谓热部署, 就是当 Jboss 服务器处于运行状态中, 能监视 <JBOSS\_HOME>/server/default/deploy 目录下文件的更新情况, 一旦监测到有新的 J2EE 组件发布到这个目录下, 或者原有的 J2EE 组件的文件发生了更改, 就会重新部署这些组件。

### 18.5.1 在 Jboss-Tomcat 上部署 EJB 组件

一个 EJB 组件由相关的类文件和 EJB 的发布描述文件构成, 它的目录结构如图 18-5 所示。

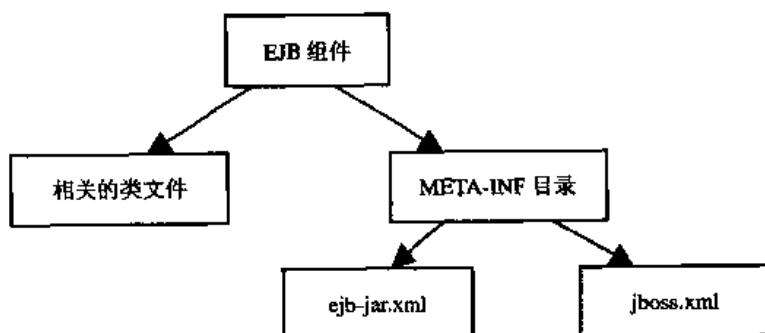


图 18-5 EJB 组件的文件目录结构

假定 BookDBEJB 组件的文件全部放在 <bookdbejb> 目录下, 创建的文件和目录结构如图 18-6 所示。

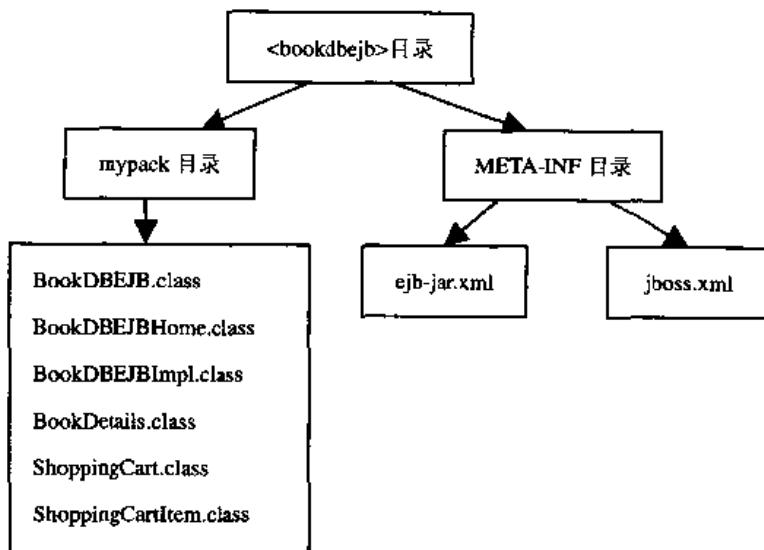


图 18-6 BookDBEJB 组件的文件目录结构

### 1. BookDBEJB 组件的相关类文件

BookDBEJB 组件的 Java 源文件由 BookDBEJB.java、BookDBEJBHome.java 和 BookDBEJBImpl.java 组成。编译这 3 个 Java 文件时，应该把 jboss-j2ee.jar 加入到 classpath 中。jboss-j2ee.jar 文件的位置为<JBOSS\_HOME>/server/default/lib/jboss-j2ee.jar。

BookDBEJB 组件还用到了 BookDetails、ShoppingCart 和 ShoppingCartItem 类，所以应该把这些类也加入进来。

### 2. ejb-jar.xml 文件

ejb-jar.xml 是 EJB 组件的发布描述文件。在这个文件中定义了 EJB 组件的类型，并指定了它的 Remote 接口、Home 接口和 Enterprise Bean 类对应的类文件。以下是 BookDBEJB 组件的 ejb-jar.xml 文件：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0
//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <description>BookStore Application</description>
  <display-name>BookDB EJB</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>BookDBEJB</ejb-name>
      <home>mypack.BookDBEJBHome</home>
      <remote>mypack.BookDBEJB</remote>
      <ejb-class>mypack.BookDBEJBImpl</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
  
```

```
</enterprise-beans>
</ejb-jar>
```

以上配置文件定义了一个无状态的会话 Bean (Stateless Session Bean), <ejb-name>指定了 EJB 组件的名字, <home>指定 Home 接口对应的类名, <remote>指定 Remote 接口对应的类名, <ejb-class>指定 Enterprise Bean 类对应的类名。

### 3. jboss.xml 文件

jboss.xml 是当 EJB 组件发布到 Jboss 服务器中时才必须提供的发布描述文件, 在这个文件中为 EJB 组件指定 JNDI 名字。以下是 BookDBEJB 的 jboss.xml 源文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>BookDBEJB</ejb-name>
      <jndi-name>ejb/BookDBEJB</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

以上代码为 BookDBEJB 组件指定了 JNDI 名字: ejb/BookDBEJB。

### 4. 给 EJB 组件打包

在发布 EJB 组件时, 应该把它打包为 JAR 文件。

在 DOS 窗口中, 转到<bookdbejb>目录, 运行如下命令:

```
jar cvf bookdbejb.jar **
```

在<bookdbejb>目录下将生成 bookdbejb.jar 文件。如果希望单独发布这个 EJB 组件, 只要把这个 JAR 文件拷贝到<JBOSS\_HOME>/server/default/deploy 下即可。在本章的 18.5.3 节, 将把这个 EJB 组件加入到 bookstore J2EE 应用中, 然后再发布整个 J2EE 应用。

## 18.5.2 在 Jboss-Tomcat 上部署 Web 应用

在本书第 2 章中, 已经介绍了 Web 应用的目录结构, 如果要在 Jboss-Tomcat 上发布 Web 应用, 可以完全保持原来的目录结构, 唯一的改动是应该在 WEB-INF 目录下增加一个 jboss-web.xml 文件。

假定 bookstore Web 应用的文件全部位于<bookstorewar>目录下, 它的目录结构如图 18-7 所示。

### 1. web.xml 文件

在 bookstore Web 应用中访问了 BookDBEJB 组件, 所以应该在 web.xml 文件中加入 <ejb-ref> 元素, 声明对这个 EJB 组件的引用。以下是 web.xml 的代码:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
'http://java.sun.com/dtd/web-app_2_3.dtd'>
<web-app>
```

```

<!-- ### EJB References (java:comp/env/ejb) -->
<ejb-ref>
    <ejb-ref-name>ejb/BookDBEJB</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>mypack.BookDBEJBHome</home>
    <remote>mypack.BookDBEJB</remote>
</ejb-ref>

</web-app>

```

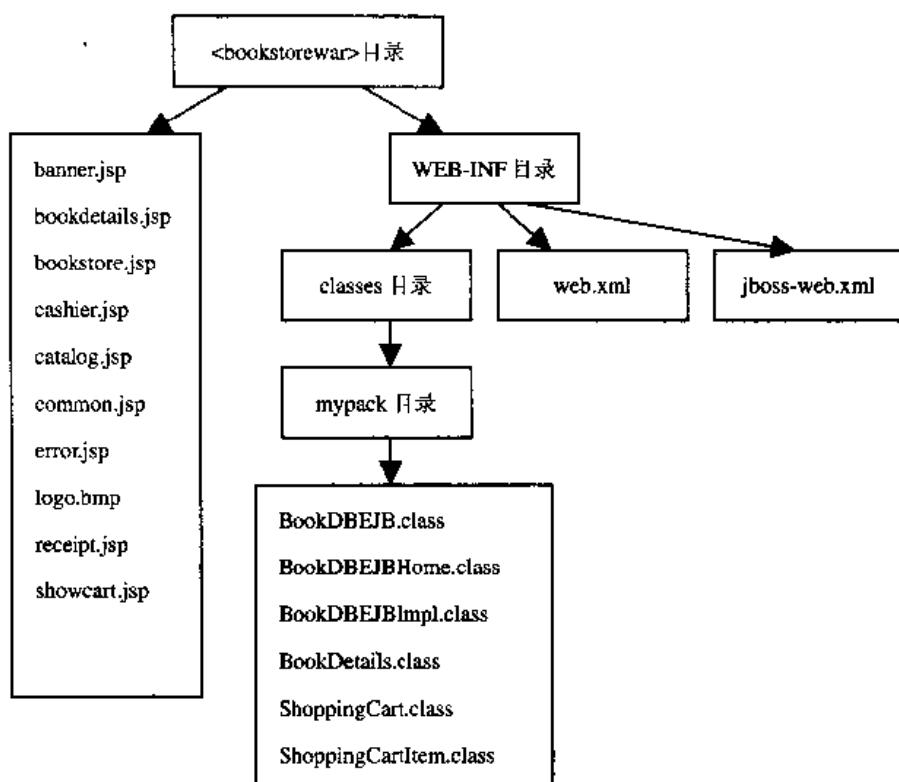


图 18-7 bookstore Web 应用的目录结构

以上代码中声明了对 BookDBEJB 的引用，`<ejb-ref-type>` 声明所引用的 EJB 的类型，`<home>` 声明 EJB 的 Home 接口，`<remote>` 声明 EJB 的 Remote 接口。在 common.jsp 中可以通过`<ejb-ref-name>`来获得 EJB 的引用，代码如下：

```

InitialContext ic = new InitialContext();
Object objRef = ic.lookup("java:comp/env/ejb/BookDBEJB");

```

## 2. jboss-web.xml 文件

`jboss-web.xml` 是当 Web 应用发布到 Jboss 服务器中才必须提供的发布描述文件，在这个文件指定`<ejb-ref-name>`和`<jndi-name>`的映射关系。以下是 `jboss-web.xml` 源文件：

```

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

<jboss-web>

```

```

<ejb-ref>
    <ejb-ref-name>ejb/BookDBEJB</ejb-ref-name>
    <jndi-name>ejb/BookDBEJB</jndi-name>
</ejb-ref>
</jboss-web>

```

在程序中访问 EJB 组件，既可以指定<ejb-ref-name>，也可以指定<jndi-name>。采用前者可以提高程序代码的独立性和灵活性，如图 18-8 所示。例如，如果部署 EJB 组件时，JNDI 名字发生更改，不需要修改程序代码，只需要修改 jboss-web.xml 文件中<ejb-ref-name>和<jndi-name>的映射关系。

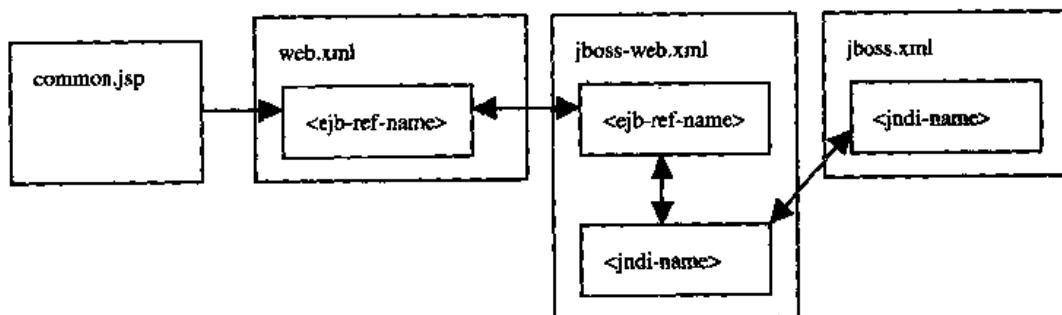


图 18-8 <ejb-ref-name>和<jndi-name>的关系

### 3. 给 Web 应用打包

在发布 Web 应用时，应该把它打包为 WAR 文件。

在 DOS 窗口中，转到<bookstorewar>目录，运行如下命令：

```
jar cvf bookstore.war *.*
```

在<bookstorewar>目录下将生成 bookstore.war 文件。如果希望单独发布这个 Web 应用，只要把这个 WAR 文件拷贝到<JBoss\_HOME>/server/default/deploy 下即可。在下一节，将把这个 Web 应用加入到 bookstore J2EE 应用中，然后再发布整个 J2EE 应用。

#### 18.5.3 在 Jboss-Tomcat 上部署 J2EE 应用

一个 J2EE 应用由 EJB 组件、Web 应用以及发布描述文件构成，它的目录结构如图 18-9 所示。

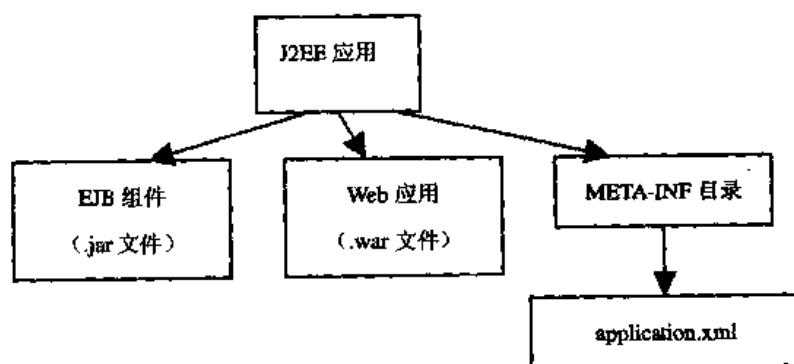


图 18-9 J2EE 应用的目录结构

假定 bookstore J2EE 应用的文件位于<bookstoreear>目录下，它的目录结构如图 18-10 所示。



图 18-10 bookstore J2EE 应用的目录结构

### 1. application.xml 文件

application.xml 是 J2EE 应用的发布描述文件，在这个文件中声明 J2EE 应用所包含的 Web 应用以及 EJB 组件。以下是 application.xml 源文件：

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

<application>
    <display-name>Bookstore J2EE Application</display-name>

    <module>
        <web>
            <web-uri>bookstore.war</web-uri>
            <context-root>/bookstore</context-root>
        </web>
    </module>

    <module>
        <ejb>bookdbejb.jar</ejb>
    </module>

</application>
```

以上代码指明在 bookstore J2EE 应用中包含一个 bookstore Web 应用，WAR 文件为 bookstore.war，它的 URL 路径为/bookstore；此外还声明了一个 EJB 组件，这个组件的 JAR 文件为 bookdbejb.jar。

### 2. 给 J2EE 应用打包

在发布 J2EE 应用时，应该把它打包为 EAR 文件。

在 DOS 窗口中，转到<bookstoreear>目录，运行如下命令：

```
jar cvf bookstore.ear *
```

在<bookstoreear>目录下将生成 bookstore.ear 文件。

### 3. 发布并运行 bookstore J2EE 应用

由于 BookDBEJB 组件通过 JDBC 驱动程序访问 MySQL 数据库，因此应该把 MySQL

的 JDBC 驱动程序 mysqldriver.jar 拷贝到<JBoss\_HOME>/server/default/lib 目录下。然后按以下步骤发布并运行 bookstore J2EE 应用。

### 步骤

- (1) 将 bookstore.ear 文件拷贝到<JBoss\_HOME>/server/default/deploy 目录下。
- (2) 启动 MySQL 服务器。
- (3) 运行<JBoss\_HOME>/bin/run.bat，该命令启动 Jboss 和 Tomcat 服务器。
- (4) 访问 <http://localhost:8080/bookstore/bookstore.jsp>，将会看到 bookstore 应用的主页。

## 18.6 小结

J2EE 是一种多层次的分布式的软件体系结构，业务逻辑由 EJB 组件来实现，它必须运行在 EJB 容器中。JBoss 是免费的 EJB 容器，具有较高的运行效率。通过 Jboss 与 Tomcat 的集成软件，可以发布一个完整的 J2EE 应用。本章以 bookstore 应用为例，介绍了 J2EE 应用的开发和部署过程。

在本书配套光盘的 sourcecode/bookstores/version3/bookstore 目录下提供了所有的源文件，bookstore 应用在 Windows 资源管理器中的展开图如图 18-11 所示。

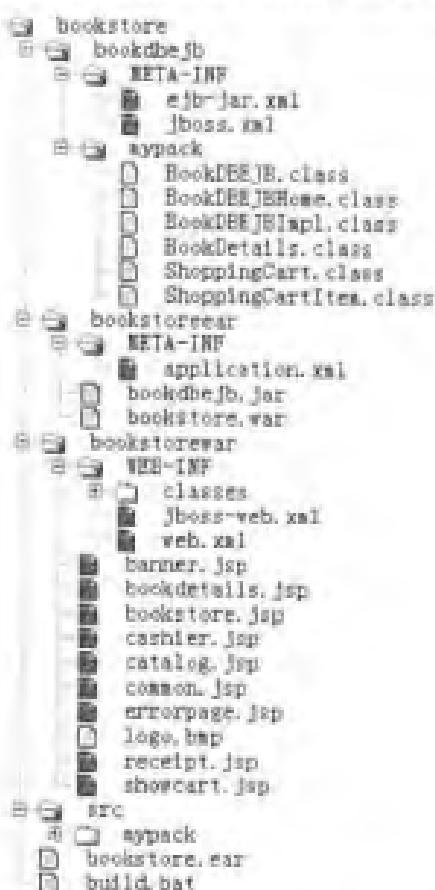


图 18-11 bookstore 应用在 Windows 资源管理器中的展开图

为了便于读者编译和部署本章的程序，在 bookstore 目录下提供了编译和打包的脚本 build.bat，它的内容如下：

```
set jboss_home=C:\jboss-tomcat
set path=%path%;C:\j2sdk1.4.2\bin

set currpath=.
if "%OS%" == "Windows_NT" set currpath=%~dp0%

set src=%currpath%src
set dest=%currpath%bookdbejb
set classpath=%classpath%;%jboss_home%\server\default\lib\jboss-j2ee.jar

javac -classpath %classpath% -sourcepath %src% -d %dest%
%src%\mypack\BookDetails.java

javac -classpath %classpath% -sourcepath %src% -d %dest%
%src%\mypack\ShoppingCartItem.java

javac -classpath %classpath% -sourcepath %src% -d %dest%
%src%\mypack\ShoppingCart.java

javac -classpath %classpath% -sourcepath %src% -d %dest%
%src%\mypack\BookDBEJB.java

javac -classpath %classpath% -sourcepath %src% -d %dest%
%src%\mypack\BookDBEJBHome.java

javac -classpath %classpath% -sourcepath %src% -d %dest%
%src%\mypack\BookDBEJBImpl.java

copy %dest%\mypack %currpath%\bookstorewar\WEB-INF\classes\mypack

cd %currpath%\bookdbejb
jar cvf %currpath%\bookstoreear\bookdbejb.jar *.*
cd ..
cd bookstorewar
jar cvf %currpath%\bookstoreear\bookstore.war *.*
cd ..
cd bookstoreear
jar cvf %currpath%\bookstore.ear *.*
```

运行这个脚本时，只要根据配置修改脚本中 Jboss 目录和 JDK 目录即可：

```
set jboss_home=C:\jboss-tomcat
set path=%path%;C:\j2sdk1.4.2\bin
```

# 第 19 章 开发 Java Mail Web 应用

本章介绍了一个 Java Mail Web 应用，通过它，客户可以访问 Mail 服务器上的邮件账号，收发信件和管理邮件夹。本章首先介绍电子邮件的发送和接收协议，接着介绍 Java Mail API 的常用类，然后讲解通过 Java Mail API 创建应用程序的步骤，最后介绍 Java Mail Web 应用。为了运行 Java Mail 例子，本书光盘提供了一个 Mail 服务器软件，本章还将介绍在 Windows 下安装和配置该 Mail 服务器的方法。

## 19.1 E-mail 协议简介

邮件服务器按照为用户提供 E-mail 发送和接收的服务不同，可以分为发送邮件服务器和接收邮件服务器。发送邮件服务器使用邮件发送协议，现在常用的是 SMTP，所以通常也把发送邮件服务器称为 SMTP 服务器；接收邮件服务器使用接收邮件协议，常用的有 POP3（Post Office Protocol 3）协议和 IMAP（Internet Message Access Protocol），所以通常也把接收邮件服务器称为 POP3 服务器或 IMAP 服务器。各协议的作用如图 19-1 所示。

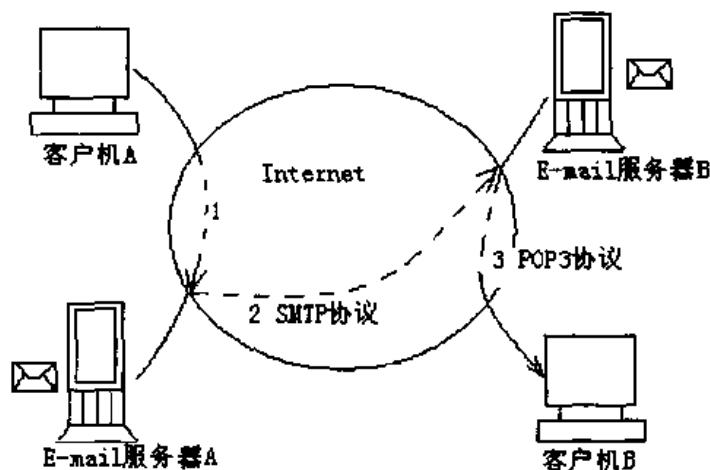


图 19-1 E-mail 系统的工作过程

图 19-1 显示了客户机 A 向客户机 B 发送邮件的过程。E-mail 服务器 A 是 SMTP 服务器，E-mail 服务器 B 是 POP3 服务器。客户机 A 首先把邮件发送到服务器 A，服务器 A 采用 SMTP 协议把邮件发送到服务器 B，服务器 B 采用 POP3 协议把邮件发送到客户机 B。

### 19.1.1 SMTP 简单邮件传送协议

SMTP (Simple Mail Transfer Protocol) 即简单邮件传输协议，是 Internet 传送 E-mail

的基本协议，也是 TCP/IP 协议组的成员。SMTP 解决了 E-mail 系统如何通过一条链路，把邮件从一台机器传递到另一台机器上的问题。

SMTP 的特点是具有良好的可伸缩性，这也是它成功的关键。它既适用于广域网，也适用于局域网。SMTP 由于自身非常简单，使得它的应用非常灵活，在 Internet 上能够接收 E-mail 的服务器都支持 SMTP。

下面介绍 SMTP 发送一封 E-mail 的过程。

客户端邮件首先到达邮件发送服务器，再由发送服务器负责传送到接收方的服务器。发送前发送服务器会与接收方服务器联系，以确认接收方服务器是否已准备好。如果已经准备好，则传送邮件；如果没有准备好，发送服务器便会等待，并在一段时间后继续与接收方服务器进行联系，若在规定的时间内联系不上，发送服务器会发送一个消息到客户的信箱说明这个情况。这种方式在 Internet 中称为“存储 - 转发”方式，以这种方式传送邮件会在沿途各网点上处于等待状态，直至允许其继续前进。虽然该方法降低了邮件的传送速度，但能极大地提高信息发至目的地的成功率。

### 19.1.2 POP3 邮局协议

POP3 即邮局协议第 3 版，这是 Internet 接收 E-mail 的基本协议，也是 TCP/IP 协议组的成员。POP3 既允许接收服务器向 E-mail 用户发出 E-mail，也可以接收来自 SMTP 服务器的 E-mail。对于典型的 Windows 拨号用户，基于 Windows 的 E-mail 客户端软件会与 POP3 服务器交互，收集由 POP3 服务器所接收到的用户 E-mail。基于 POP3 协议的 E-mail 系统，能提供快速、经济和方便的 E-mail 接收服务，深受用户喜爱。

基于 POP3 协议的 E-mail 系统阅读信件的过程如下。

用户通过自己所熟悉的 E-mail 客户端软件，例如 Foxmail、Outlook Express 和 the Bat 等，经过相应的参数设置（主要是设置 POP3 邮件服务器的 IP 地址或者域名、用户账号及其对应密码）后，只要选择接收邮件操作，就能够将所有的邮件从远端的邮件服务器下载到用户的本地硬盘里。邮件下载之后，用户就可以在本地阅读了。当然，如果想节省上网费用，也可以选择在脱机状态下慢慢地阅读邮件。

### 19.1.3 E-mail 接收的新协议 IMAP

IMAP 即直接收取邮件的协议，是与 POP3 不同的一种 E-mail 接收的新协议。IMAP 可以让用户远程接号连接邮件服务器，并且可以在下载邮件之前预览信件主题与信件来源。有了 IMAP，用户阅读邮件服务器上的邮件，就像这些邮件存储在本地机上一样方便。

因为 IMAP 具有远程访问的能力，所以它最有可能被那些认为漫游是重要特性的公司用户所采用。在多数情况下，漫游用户愿意把他们的信件保存在邮件服务器上，这样通过任何一台机器的浏览器都可以收取新的信件或查看旧信。

IMAP 不同于 POP3 协议，POP3 协议将信件存储在一台服务器上，一旦用户和服务器连通，它便将信件发送到客户机上，并从服务器上删除这些邮件。而且，POP3 不支持用户在服务器上创建邮件夹，可见，使用 POP3 时，用户对在邮件服务器上邮件的控制权很小。

而 IMAP 具有所谓的智能邮件存储功能，用户可以在下载邮件前预览相关信息，包括是否下载附件等。用户可以使用邮件服务器上的过滤软件或搜索代理软件，在任何地方、任何机器上通过浏览器获取邮件信息。此外，IMAP 允许用户在邮件服务器上创建自己的邮件夹。但是由于不同的厂商对最新版本的 IMAP 规范的解释有所不同，因此造成了邮件客户机与服务器之间的不一致性。



在通过 Microsoft 的 Outlook Express 软件创建或修改邮件夹时，这些邮件夹都存在于本地硬盘上，而不是在邮件服务器上。

与 POP3 相比，虽然 IMAP 具有很多优点，但是由于 IMAP 受到不同厂商产品之间不兼容的限制，所以还没有被大规模使用。不过，IMAP 在将来不可避免地会得到迅猛发展。当然，POP3 也不会被淘汰，实用的多功能服务器能对 IMAP 和 POP3 这两种协议兼容。

## 19.2 Java Mail API 简介

Java Mail API 是 SUN 为 Java 开发者提供的公用 Mail API 框架，它支持各种电子邮件通信协议，如 IMAP、POP3 和 SMTP，为 Java 应用程序提供了电子邮件处理的公共接口，如图 19-2 所示。

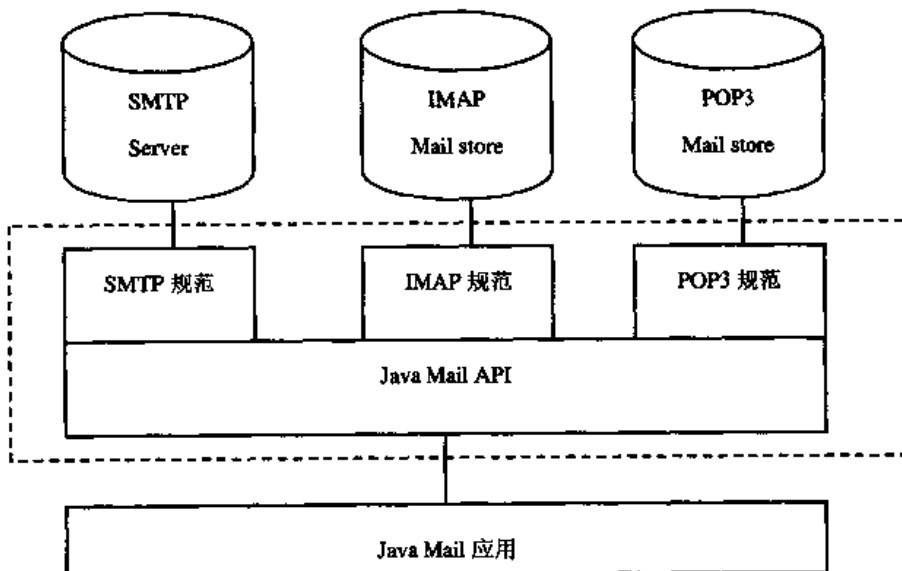


图 19-2 Java Mail 框架

Java Mail API 的最主要的类包括以下 6 种。

### 1. javax.mail.Session

Session 类定义了一个基本邮件会话，是 Java Mail API 最高层入口类。所有其他类都是经由这个 Session 才得以生效。Session 对象从 java.util.Properties 对象中获取信息，如邮件发送服务器、接受邮件协议、发送邮件协议、用户名、密码及整个应用程序中共享的其

他信息。

## 2. javax.mail.Store

Store 类是访问接收邮件服务器上邮件账户的入口，通过 Store 类的 getFolder 方法，可以访问特定的邮件夹。

## 3. javax.mail.Folder

Folder 类代表了邮件夹，用于分级组织邮件，通过 Folder 类可以访问邮件夹中的邮件。

## 4. javax.mail.Message

Message 代表了电子邮件。Message 类封装了邮件信息，提供了访问和设置邮件内容的方法。邮件中包含如下内容：

- 地址信息，包括发件人地址、收件人地址列表、抄送地址列表和广播地址列表
- 邮件标题
- 邮件发送和接收日期
- 邮件具体内容

Message 是个抽象类，常用的子类为 javax.mail.internet.MimeMessage。MimeMessage 能支持 MIME 类型电子邮件消息。

### 提示

MIME (Multipurpose Internet Mail Extensions) 是一种电子邮件编码方式，它可以将发信人的电子邮件中的文本以及各种附件都打包后发送。传送时即时编码，收信人的软件收到邮件后也是即时解码还原，完全自动化，非常方便。当然，先决条件是双方的软件都必须支持 MIME 编码，否则发信人很方便地把信送出去了，但收信人的软件如果没有这种功能，无法把它还原，看到的也就是一大堆乱码了。

## 5. javax.mail.Address

Address 代表了邮件地址，与 Message 一样，Address 也是个抽象类。常用的子类为 javax.mail.internet.InternetAddress 类。

## 6. javax.mail.Transport

Transport 类根据指定的邮件发送协议（通常是 SMTP），通过指定的邮件发送服务器来发送邮件。Transport 类是抽象类，它提供了一个静态方法 send(Message) 来发送邮件。

## 19.3 Java Mail 应用程序开发环境

开发 Java Mail 应用程序需要两个 JAR 文件：mail.jar 和 activation.jar。另外，为了运行本章介绍的程序，应该准备好可以访问的 Mail 服务器。

### 19.3.1 获得 Java Mail JAR 文件

可以到 <http://java.sun.com/products/javamail> 下载最新的 Java Mail API。下载完毕，解

开 javamail\_X.zip 压缩文件，就会获得 mail.jar 文件，它包含了 Java Mail API 中所有的接口和类。

除了 mail.jar 外，还需要到 <http://java.sun.com/beans/glasgow/jaf.html> 下载最新的 JavaBeans Activation Framework。Java Mail API 的所有版本都需要 JavaBeans Activation Framework 的支持。下载完框架后，解开 jaf\_X.zip 文件，就会获得 activation.jar 文件，它包含了 JavaBeans Activation Framework 中所有的接口和类。

此外，在本书配套光盘的 lib 目录下也提供了 mail.jar 和 activation.jar 文件。

### 19.3.2 安装和配置 Mail 服务器

为了运行本章介绍的程序，应该准备好可以访问的 Mail 服务器。本书配套光盘的 software 目录下提供了一个基于 Windows 2000/NT 的 Mail 服务器软件 mailserver.zip，它是 MerakMailServer 公司提供的 Mail 服务器试用版本。把它解压到本地硬盘，然后运行安装程序 Setup.exe。

在安装过程中会出现 Details 窗口，提示用户输入姓名、E-mail、公司和国家信息，如图 19-3 所示。



图 19-3 用户信息输入窗口

在安装的最后阶段，会出现 Domain 配置窗口，如图 19-4 所示，可以输入如下配置信息：

Hostname: mail.mydomain.com  
 Domain: mydomain.com  
 Username: admin  
 Password: 1234

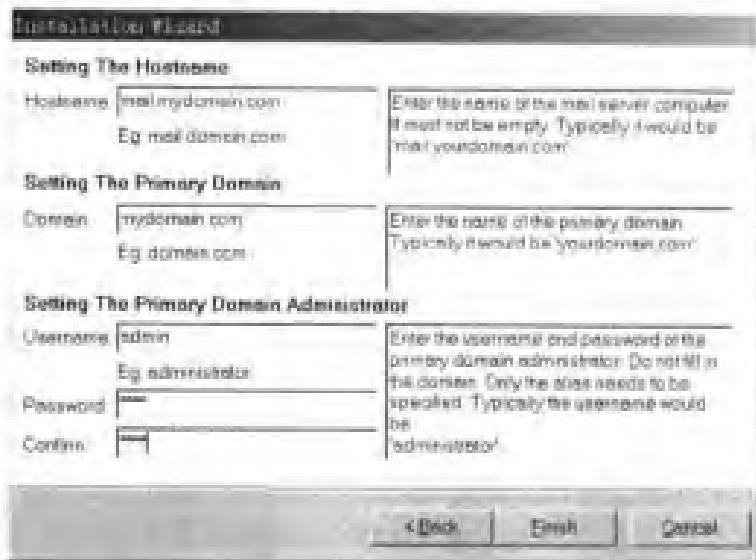


图 19-4 Domain 配置窗口

该 Mail 服务器安装软件会自动在 Windows 操作系统中加入邮件发送和接收服务，发送邮件采用 SMTP，接收邮件支持 POP3 和 IMAP，如图 19-5 所示。每次启动操作系统时，会自动运行这两项服务。



图 19-5 Mail 服务器在 Windows 操作系统中加入邮件发送和接收服务

Mail 服务器安装好以后，选择 Windows 操作系统的【开始】→【程序】→【Merak Mail Server】→【Merak Mail Server Administration】菜单，将运行 Mail 服务器的管理程序，在 Accounts/Users 目录下，会看到已经配置的一个邮件账号：admin@mydomain.com。修改这个账号的接收邮件协议属性，把原来默认的 POP3 协议改为 IMAP，如图 19-6 所示。



把邮件账号的接收邮件协议改为 IMAP，这是因为 IMAP 比 POP3 协议向用户提供更多的对邮件服务器上邮件的控制权限。

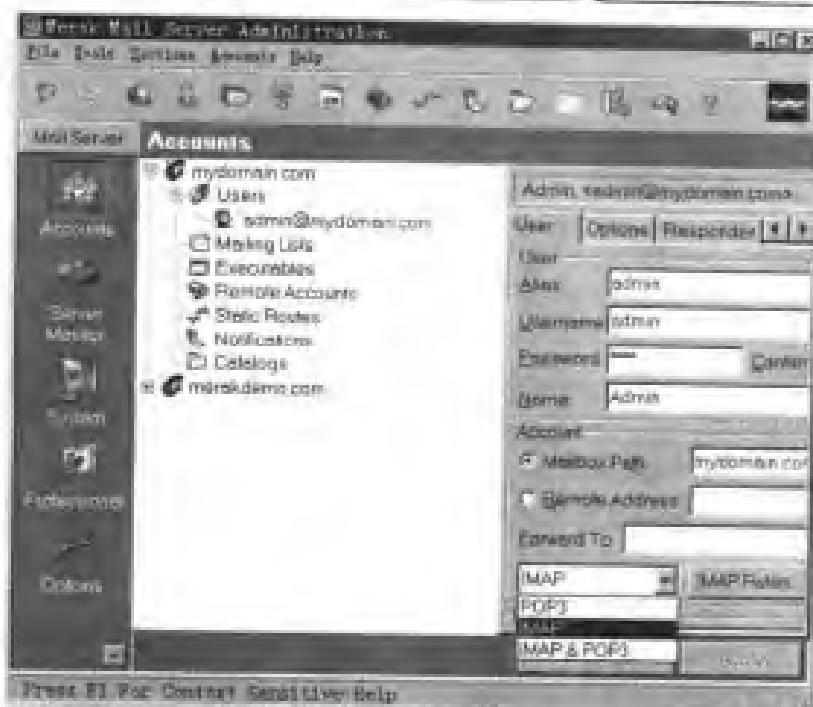


图 19-6 在 Mail 服务器的管理窗口修改 admin 账户的接收邮件协议

这样，Mail 服务器就安装配置成功了，下面将通过 Java 程序来访问在 Mail 服务器上的 admin@mydomain.com 账号。

## 19.4 创建 Java Mail 应用程序

假定 Mail 服务器安装在本地计算机上，访问 Mail 服务器的 admin@mydomain.com 账号需要如下信息：

```
String hostname = "localhost";
String username = "admin";
String password = "1234";
```

创建 Java Mail 应用程序必须包含如下步骤。

### 步骤

#### (1) 设置 Java Mail 属性：

```
//set properties
Properties props = System.getProperties();
props.put("mail.transport.protocol", "smtp");
props.put("mail.store.protocol", "imap");
props.put("mail.smtp.class", "com.sun.mail.smtp.SMTPTransport");
props.put("mail imap.class", "com.sun.mail.imap.IMAPStore");
props.put("mail.smtp.host", hostname);
```

以上 Java Mail 属性的描述参见表 19-1。

表 19-1 Java Mail 属性

Java Mail 属性	描述
mail.transport.protocol	指定邮件发送协议
mail.store.protocol	指定邮件接收协议
mail.smtp.class	指定采用 SMTP 发送邮件的类
mail imap.class	指定采用 IMAP 的 Store 类
mail.smtp.host	指定采用 SMTP 的邮件发送服务器

(2) 调用 javax.mail.Session 类的静态方法 Session.getDefaultInstance 获取 Session 实例，该方法将根据已经配置的 Java Mail 属性来创建 Session 实例：

```
Session mailsession = Session.getDefaultInstance(props, null);
```

(3) 调用 Session 的 getStore 方法来取得 Store 对象，在本例中，将获取采用 IMAP 的 Store 对象：

```
Store store = mailsession.getStore("imap");
```

根据配置的 mail imap.class 属性，可以确定这里 getStore 方法返回 com.sun.mail.imap.IMAPStore 类型的 Store 对象。

(4) 调用 Store 的 connect 方法连接到接收邮件服务器上的特定邮件账号。调用 connect 方法时应该指定接收邮件服务器名称或 IP 地址、邮件账户名和口令。

```
store.connect(hostname,username,password);
```

在获得了 Store 对象后，就可以通过它来访问邮件服务器上的特定邮件账号，可以完成以下功能。

### 1. 创建并发送邮件

```
// create a message
msg = new MimeMessage(mailsession);
InternetAddress[] toAddrs = InternetAddress.parse("admin@mydomain.com", false);
msg.setRecipients(Message.RecipientType.TO, toAddrs);
msg.setSubject("hello");
msg.setFrom(new InternetAddress("admin@mydomain.com"));
msg.setText("How are you");
//send a message
Transport.send(msg);
```

Transport 的静态方法 send(Message) 负责发送邮件服务器，邮件发送协议由 mail.transport.protocol 属性指定，邮件发送服务器由 mail.smtp.host 属性指定。

### 2. 打开 inbox 邮件夹查看邮件信息

```
//check inbox
Folder folder=store.getFolder("inbox");
folder.open(Folder.READ_WRITE);
System.out.println("You have "+folder.getMessageCount()+" messages in inbox.");
System.out.println("You have "+folder.getUnreadMessageCount()+" unread messages in inbox.");
```

### 3. 从邮件夹中读取信件

```
//read first Message in inbox
```

```

Message msg=folder.getMessage(1);
System.out.println("-----the first message in inbox-----");
System.out.println("From:" +msg.getFrom()[0]);
System.out.println("Subject:" +msg.getSubject());
System.out.println("Text:" +msg.getContent());

```

例程 19-1 是 MailTest 的源程序。

例程 19-1 MailTest.java

---

```

import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
import java.util.*;

public class MailTest {

    public MailTest() {
    }

    public static void main(String[] args) throws Exception {
        String hostname = "localhost";
        String username = "admin";
        String password = "1234";

        //set properties
        Properties props = System.getProperties();
        props.put("mail.transport.protocol", "smtp");
        props.put("mail.store.protocol", "imap");
        props.put("mail.smtp.class", "com.sun.mail.smtp.SMTPTransport");
        props.put("mail imap.class", "com.sun.mail.imap.IMAPStore");
        props.put("mail.smtp.host", hostname);

        // Get a Session object
        Session mailsession = Session.getDefaultInstance(props, null);

        // Get a Store object
        Store store = mailsession.getStore("imap");
        // Connect
        store.connect(hostname,username, password);

        //check inbox
        Folder folder=store.getFolder("inbox");
        folder.open(Folder.READ_WRITE);
        System.out.println("You have "+folder.getMessageCount()+" messages in inbox.");
        System.out.println("You have "+folder.getUnreadMessageCount()+" unread messages in inbox.");
    }
}

```

```
//read first Message in inbox  
Message msg=folder.getMessage(1);  
System.out.println("----the first message in inbox-----");  
System.out.println("From:" +msg.getFrom()[0]);  
System.out.println("Subject:" +msg.getSubject());  
System.out.println("Text:" +msg.getContent());  
  
//create a message  
msg = new MimeMessage(mailSession);  
InternetAddress[] toAddrs = InternetAddress.parse("admin@mydomain.com", false);  
msg.setRecipients(Message.RecipientType.TO, toAddrs);  
msg.setSubject("hello");  
msg.setFrom(new InternetAddress("admin@mydomain.com"));  
msg.setText("How are you");  
  
//send a message  
Transport.send(msg);  
}  
}
```

编译和运行该程序时，应该把 mail.jar 和 activation.jar 加入到 classpath 中，运行该程序的输出结果如下：

```
You have 1 messages in inbox.  
You have 1 unread messages in inbox.  
----the first message in inbox-----  
From:admin <admin@mydomain.com>  
Subject:hello  
Text: How are you
```

## 19.5 Java Mail Web 应用简介

在本节将介绍一个 Java Mail Web 应用（简称为 javamail 应用）例子，它向 Web 客户提供了访问 IMAP 服务器上的邮件账号的功能，它允许 Web 客户管理邮件夹、查看邮件。此外，Web 客户也可以通过 SMTP 服务器发送邮件，在 19.1 节已经介绍过。与 POP3 协议相比，IMAP 为客户提供更多的对邮件服务器上邮件的控制权限，如管理邮件和邮件夹等。Web 应用要求用户访问的邮件接收服务器必须支持 IMAP。

图 19-7 显示了该应用的三层结构：Web 客户通过浏览器访问 javamail 应用，该应用可以连接到客户请求的某个 IMAP 服务器上的邮件账号。

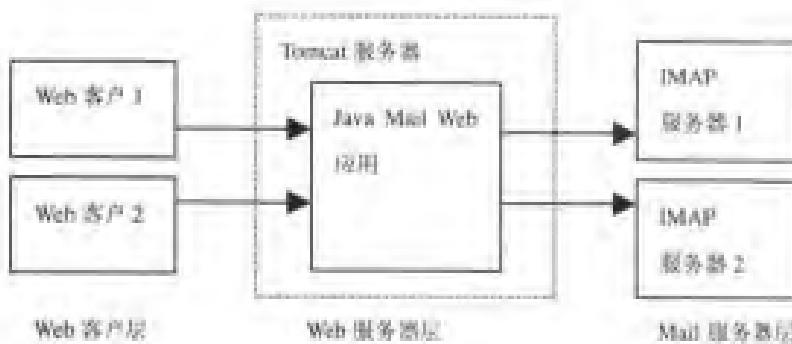


图 19-7 javamail 应用的三层结构

## 19.6 Java Mail Web 应用的程序结构

构成整个应用的所有文件参见表 19-2。

表 19-2 javamail 应用的文件清单

文件名称	描述
PMessage.java	对 Message 数据重新封装，提供更方便地读取邮件信息的方法
MailUserData.java	作为 HTTP Session 范围内的 Java Bean 保存客户的邮件账号信息，还提供了管理邮件和邮件夹的实用方法
common.jsp	包含了各个 JSP 文件的共同内容，如 import 语句和 Java Bean 的声明
link.jsp	包含了各个 JSP 文件的共同连接
login.jsp	提供客户登录页面
connect.jsp	根据客户的登录信息负责连接邮件服务器上的邮件账号
listallfolders.jsp	显示客户邮件账号中所有的邮件夹
listonefolder.jsp	显示客户指定的邮件夹中所有的邮件
showmessage.jsp	显示客户指定的邮件内容
compose.jsp	提供创建、编辑和发送邮件的功能
logout.jsp	退出邮件系统
errorpage.jsp	错误处理网页
web.xml	Web 应用的配置文件

javamail 应用在 Windows 资源管理器中的展开图如图 19-8 所示。

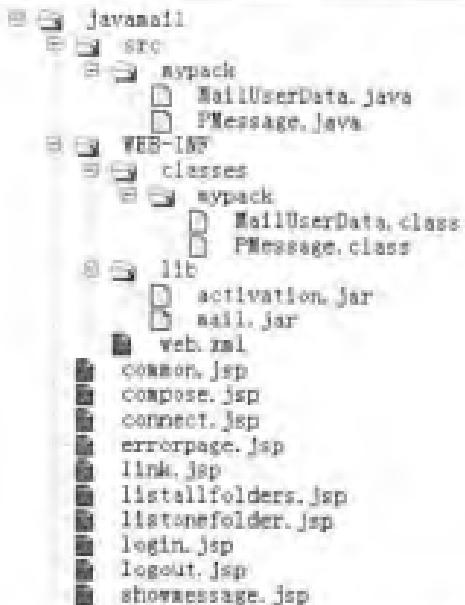


图 19-8 javamail 应用在 Windows 资源管理器中的展开图

javamail 应用的站点导航图如图 19-9 所示，在这个图中显示了各个网页之间的链接关系。

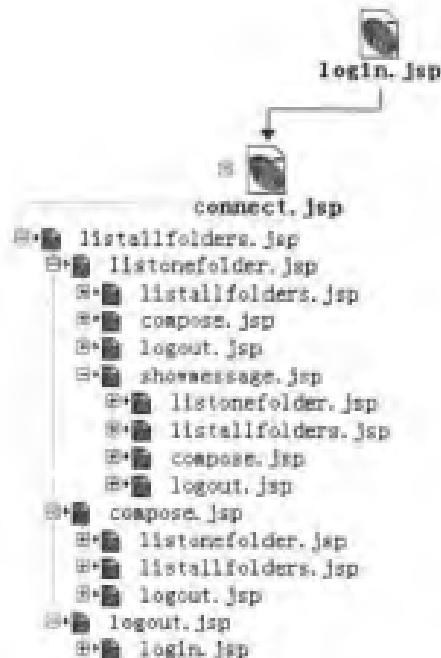


图 19-9 javamail 应用的站点导航图

### 19.6.1 重新封装 Message 数据

PMessage 类对 Message 数据重新封装，提供更方便地读取邮件信息的方法。例如，在 Message 类中读取邮件地址的方法，如 getTo()方法返回 Address[]类型，如果要把它显示到

网页上，必须把 Address 数组转化为对应的字符串。PMessage 的构造方法 PMessage 对 Message 数据重新封装。JSP 网页调用 PMessage 的 getTo 方法可以直接获得字符串类型的地址。

例程 19-2 是 PMessage 的源程序。

例程 19-2 PMessage.java

```
package mypack;
import javax.mail.*;
import javax.mail.internet.*;
import java.util.*;
import java.text.*;

public class PMessage{

    private String subject="";
    private String from="";
    private String to="";
    private String cc="";
    private String bcc="";
    private String date=new Date().toString();
    private int size=0;
    private String text="";
    private boolean readFlag;

    public PMessage(){}
    public PMessage(Message msg) throws Exception{
        if(msg!=null){
            //set Date
            SimpleDateFormat df = new SimpleDateFormat("yy.MM.dd 'at' HH:mm:ss ");
            try{date=df.format((msg.getSentDate()!=null) ? msg.getSentDate() :
msg.getReceivedDate());}
            catch(Exception e){date=new Date().toString();}
            //set Subject
            subject=msg.getSubject();
            //set Size
            size=msg.getSize();
            //set Text
            Object content="";
            try{
                content=msg.getContent();
            }catch(Exception e){}
            if(msg.isMimeType("text/plain") && content!=null)
                text=(String)content;
        }
    }
}
```

```
//set address
from=assembleAddress(msg.getFrom());
to=assembleAddress(msg.getRecipients(Message.RecipientType.TO));
cc=assembleAddress(msg.getRecipients(Message.RecipientType.CC));
bcc=assembleAddress(msg.getRecipients(Message.RecipientType.BCC));

}

}

public PMessage(String to,String cc,String bcc,String subj,String text){
    to.replace(';',',');
    cc.replace(';',',');
    bcc.replace(';',',');
    this.to=to;
    this.cc=cc;
    this.bcc=bcc;
    this.subject=subj;
    this.text=text;
}

private String assembleAddress(Address[] addr){
    if(addr==null) return "";
    String addrString="";
    boolean tf = true;
    for (int i = 0; i < addr.length; i++) {
        addrString=addrString+((tf) ? " " : ", ") + getDisplayAddress(addr[i]);
        tf = false;
    }
    return addrString;
}

// utility method; returns a string suitable for msg header display
private String getDisplayAddress(Address a) {
    String pers = null;
    String addr = null;
    if (a instanceof InternetAddress &&
        ((pers = ((InternetAddress)a).getPersonal()) != null)) {
        addr = pers + " "<" +((InternetAddress)a).getAddress()+">";
    } else
        addr = a.toString();

    return addr;
}
```

```
public String getFrom(){return from;}
public void setFrom(String from){
    this.from=from;
    if(this.from==null)this.from="";
}

public String getTo(){return to;}
public void setTo(String to){
    this.to=to;
    if(this.to==null)this.to="";
}

public String getCC(){return cc;}
public void setCC(String cc){
    this.cc=cc;
    if(this.cc==null)this.cc="";
}

public String getBCC(){return bcc;}
public void setBCC(String bcc){
    this.bcc=bcc;
    if(this.bcc==null)this.bcc="";
}

public int getSize(){return size;}
public void setSize(int size){this.size=size; }

public String getDate(){return date;}
public void setDate(String date){this.date=date; }

public String getSubject(){return subject;}
public void setSubject(String subject){
    this.subject=subject;
    if(this.subject==null)this.subject="";
}

public String getText(){return text;}
public void setText(String text){
    this.text=text;
    if(this.text==null)this.text="";
}

public boolean getReadFlag(){return readFlag;}
public void setReadFlag(boolean readFlag){this.readFlag=readFlag; }

}
```

在 PMessage 类的构造方法 PMessage(String to, String cc, String bcc, String subj, String text) 中，首先将参数传入的地址信息中的分号改为逗号，这是因为 IMAP 要求多个邮件地址之间采用逗号作为间隔。假如用户输入地址信息之间采用分号间隔，PMessage 类的构造方法就先把它们替换为逗号。

```
to.replace(';', ',');
cc.replace(';', ',');
bcc.replace(';', ',');
```

此外，为了简化起见，本 javamail 应用只处理邮件内容为文本的邮件，没有考虑带附件的情况：

```
Object content="";
try{
    content=msg.getContent();
} catch(Exception e){}
if(msg.isMimeType("text/plain") && content!=null)
    text=(String)content;
```

### 19.6.2 用于保存邮件账号信息的 Java Bean

当客户登录到 Mail 服务器后，他的邮件账号信息保存在 MailUserData 中，MailUserData 对象作为 HTTP Session 范围内的 Java Bean 而存在。在 MailUserData 中定义了如下属性，并为这些属性提供了相应的 getXXX 和 setXXX 方法：

```
//客户连接 Mail 服务器邮件账号的 URL
URLName urlName;

//客户当前使用的 Mail Session
Session session;

//客户当前使用的 Store
Store store;

//客户当前访问的 Folder
Folder currFolder;

//客户当前访问的 Message
Message currMsg;
```

在 MailUserData 中还提供了管理邮件和邮件夹的实用方法，这些方法的描述参见表 19-3。

在本书配套光盘的 sourcecode/javamails/version0/javamail/src 目录下提供了 MailUserData.java 源文件。由于篇幅较长，因此不在本书里列出，读者可以参考光盘中的源文件。

关于 MailUserData，有以下几个需要注意的方面：

- 由于本 javamail 应用访问的是 IMAP 服务器，因此所有邮件和邮件夹都存放在 IMAP 邮件服务器上，修改或删除邮件和邮件夹的实际操作自然也在邮件服务器上执行。

- Web 应用指定的系统邮件夹指的是 inbox、Draft、SendBox 和 Trash。其中 inbox 邮件夹是由 IMAP 服务器自动创建的，用于存放接收到的新邮件。IMAP 不允许用户删除该邮件夹。在 javamail 应用中为用户创建了 Draft、SendBox 和 Trash 邮件夹，并为它们指定了特定的用途：Draft 邮件夹存放用户编辑的邮件，SendBox 存放已经发送的邮件，Trash 存放从其他邮件夹中删除的邮件。在 Web 应用层对这些邮件夹进行了限制，不允许用户删除它们或修改它们的名字。
- Java Mail API 的 Message 类没有直接提供删除邮件的方法，如果要删除邮件，首先把 Message 的 DELETED 标志设为 true，然后调用邮件所在邮件夹 Folder 的 expunge 方法，该方法删除邮件夹中所有 DELETED 标志为 true 的邮件。

表 19-3 MailUserData 的管理邮件和邮件夹的实用方法

方 法	描 述
doDeleteFolder	删除用户自己创建的邮件夹，但不允许删除 Web 应用指定的系统邮件夹
doRenameFolder	修改用户自己创建的邮件夹的名字，但不允许修改 Web 应用指定的系统邮件夹的名字
doCreateFolder	创建用户自己的邮件夹
doAppendMessage	把邮件添加到参数指定的邮件夹中
doAssembleMessage	根据参数给定的邮件信息，如标题、收发地址和邮件内容，来构建 Message 对象
doDeleteMessage	如果该邮件在 Trash 邮件夹中，就永久删除该邮件；否则把这封邮件移到 Trash 邮件夹中
doMoveMessage	从用户当前邮件夹中，把用户当前访问的邮件移到参数指定的邮件夹中
doSaveMessage	把用户编辑的邮件保存到 Draft 邮件夹中
doSendMessage	发送邮件，并把邮件保存到 SendBox 邮件夹中

例如，在 doDeleteMessage 方法中，如果邮件不在 Trash 邮件夹中，首先把这个邮件在 Trash 邮件夹中备份，然后把原来邮件的 DELETED 标志设为 true；如果邮件在 Trash 中，就直接把邮件的 DELETED 标志设为 true。两种情况下最后都调用待删除邮件所在邮件夹的 expunge 方法，该方法能够删除邮件夹中所有 DELETED 标志为 true 的邮件：

```
public void doDeleteMessage(int arrayOpt[],Folder f)throws Exception {
    for(int i=0;i<arrayOpt.length;i++){
        if(arrayOpt[i]==0)continue;
        Message msg=f.getMessage(i+1);
        if(f.getName().equals("Trash")){
            Message[] m=new Message[1];
            m[0]=msg;
            Folder Trash=store.getFolder("Trash");
            f.copyMessages(m,Trash);
            msg.setFlag(Flags.Flag.DELETED, true);
        }else{
            msg.setFlag(Flags.Flag.DELETED, true);
        }
    }
    f.expunge();
}
```

以上 doDeleteMessage 方法的 arrayOpt 参数用来指定删除邮件夹中哪些邮件，例如，如果 arrayOpt[5]=1，表示需要删除邮件夹中第 5 封邮件；如果 arrayOpt[5]=0，表示不需要删除这封邮件。

### 19.6.3 定义所有 JSP 文件的相同网页内容

在 common.jsp 中定义了其他 JSP 文件都包含的共同内容，其他的 JSP 文件都通过 include 标记将其包含进来。

在 common.jsp 中声明了所有 JSP 文件中引入的 Java 包，还声明了 HTTP Session 范围内的 MailUserData Java Bean：

```
<jsp:useBean id="mud" scope="session" class="mypack.MailUserData"/>
```

此外，它还能判断用户是否已经登录到邮件服务器或者访问的是 HTTP Session 过期的网页：

- 用户未登录的场合下用户打开浏览器，直接访问：

<http://localhost:8080/javamail/listallfolders.jsp>

- 访问 HTTP Session 过期网页的场合下用户已经通过 login.jsp 登录到邮件服务器，在 listallfolders.jsp 网页上选择“Logout”链接，客户请求转到 logout.jsp，logout.jsp 响应客户请求，调用 session.invalidate() 方法，使当前的 HttpSession 对象和 MailUserData Bean 都失效。此时客户又选择浏览器的【后退】按钮返回到已经访问过的 listallfolders.jsp 网页，然后选择浏览器的刷新按钮。

在以上两种情况下，Servlet 容器都将创建一个新的 HttpSession 和 MailUserData Bean 对象，MailUserData 对象的属性都为空，此时执行如下代码将抛出异常。

```
<%>
if(mud.getStore()==null) throw new Exception("The page you visit expires or you do
not login yet. Please <a href=login.jsp>login again</a>");
```

```
<%>
```

当出现异常后，客户请求由异常处理网页 errorpage.jsp 处理，errorpage.jsp 生成的错误信息如图 19-10 所示。

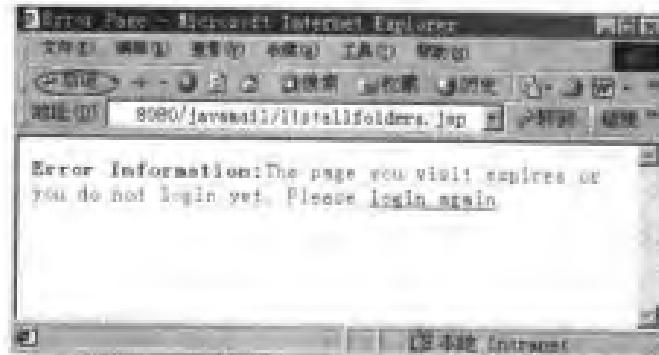


图 19-10 HTTP Session 过期时的出错网页

以下是 common.jsp 的源代码：

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

```

<%@ page import="java.text.*"%>
<%@ page import="mypack.*"%>
<%@ page import="javax.mail.*"%>
<%@ page import="javax.mail.internet.*"%>
<%@ page import="javax.activation.*"%>
<%@ page errorPage="errorpage.jsp"%>

<jsp:useBean id="mud" scope="session" class="mypack.MailUserData"/>

<%
if(mud.getStore()==null) throw new Exception("The page you visit expires or you do
not login yet. Please <a href=login.jsp>login again</a>");

%>

```

所有的 JSP 网页都包含了以下链接：

- CheckMail 查看收件箱中的邮件
- Folders 管理邮件夹
- Compose 创建和编辑邮件
- Logout 退出邮件系统

在 link.jsp 中定义了这些链接，其他的 JSP 网页通过 include 标记将其包含进来。以下是 link.jsp 的代码：

```

<a href="listonefolder.jsp?folder=inbox" >CheckMail</a>
<a href="listallfolders.jsp" >Folders</a>
<a href="compose.jsp" >Compose</a>
<a href="logout.jsp">Logout</a>
<hr>

```

#### 19.6.4 登录 IMAP 服务器上的邮件账号

login.jsp 提供了用户登录 IMAP 服务器的网页。如果已按照 19.3.2 节的内容在本地安装配置了 Mail 服务器，并且拥有一个邮件账号，用户名为：admin，口令为：1234，可以在登录页面中输入如图 19-11 所示的信息。



图 19-11 login.jsp 网页

例程 19-3 是 login.jsp 的源代码。

例程 19-3 login.jsp

```
<html><head><title>JavaMail</title></head>
<body >
<p><center><font face="Arial,Helvetica" font size=+3><b>Welcome to Java Mail
Web</b></font></center></p><hr>

<%
String loginfail=(String)request.getAttribute("loginfail");
if(loginfail!=null && loginfail.equals("true")){
%>
<center><p><font color="red">Login Failed. MailServer,UserName or password are incorrect.
</font></p></center>
<%
}
%>

<form action="connect.jsp" method="post" >
<center>
<table >
<tr>
<td >IMAP Mail Server:</td>
<td ><input TYPE="text" name="hostname" size="25"></td>
</td>
<tr>
<td >Username:</td>
<td ><input type="text" name="username" size="25"></td>
</tr>
<tr>
<td >Password:</td>
<td ><input type="password" name="password" size="25"></td>
</tr>
</table>

</center>
<center><br>
<input type="submit" value="Login">
<input type="reset" name="Reset" value="Reset">
</center>
</form>

</body>
</html>
```

客户递交了登录表单后，该请求由 connect.jsp 来处理。connect.jsp 负责根据客户的登

录信息连接到 IMAP 邮件服务器上的邮件账号，例程 19-4 是 connect.jsp 的代码。

例程 19-4 connect.jsp

```

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%@ page errorPage="errorpage.jsp" %>
<jsp:useBean id="mud" scope="session" class="mypack.MailUserData"/>

<%
    private Properties props=null;
    public void jspInit() {
        props = System.getProperties();
        props.put("mail.transport.protocol", "smtp");
        props.put("mail.store.protocol", "imap");
        props.put("mail.smtp.class", "com.sun.mail.smtp.SMTPTransport");
        props.put("mail.imap.class", "com.sun.mail.pop3.POP3Store");
        props.put("mail.smtp.host", "localhost");
    }
%>

<%
    String hostname = request.getParameter("hostname");
    String username = request.getParameter("username");
    String password = request.getParameter("password");

    // Get a Session object
    Session mailsession = Session.getInstance(props, null);

    // Get a Store object
    Store store = mailsession.getStore("imap");
%>

<%
    try{
        // Connect
        store.connect(hostname,username, password);
    }catch(Exception e){
        request.setAttribute("loginfail","true");
    }
%>
    <jsp:forward page="login.jsp" />
<%
    }//end of catch
%>

```

```
<%>
// save stuff into MUD
mud.setSession(mailsession);
mud.setStore(store);

//create and open default Trash, Draft and sendbox folder

Folder folder=store.getFolder("Trash");
if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

folder=store.getFolder("SendBox");
if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

folder=store.getFolder("Draft");
if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

folder.open(Folder.READ_WRITE);

//save draft into MUD
URLName url = new URLName("imap",hostname,-1,"inbox",username,password);
mud.setURLName(url);

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

如果 Mail 服务器连接失败，connect.jsp 在 request 对象中设置“loginfail”属性，再把客户请求转发给 login.jsp，login.jsp 将显示登录失败信息，如图 19-12 所示。



图 19-12 登录失败时的 login.jsp 网页

如果 Mail 服务器连接成功，首先查看是否存在 Trash、Draft 和 SendBox 邮件夹，如果不存在就创建这些邮件夹。这几个邮件夹连同 inbox 邮件夹将作为 javamail 应用为用户保留的系统邮件夹，不允许用户删除或改名。接下来，connect.jsp 把客户请求再转发给

listallfolders.jsp.

### 19.6.5 管理邮件夹

listallfolders.jsp 用于显示用户邮件夹信息，并且提供了创建、删除邮件夹，以及修改邮件夹名字的功能。如图 19-13 所示。IMAP 协议允许用户创建树形结构的邮件夹系统，即邮件夹下面允许包含子邮件夹。为了简化起见，本例中仅考虑了单层目录结构的情况。



图 19-13 listallfolders.jsp 网页

当用户提交了创建或删除邮件夹的操作后，listallfolders.jsp 将调用 MailUserData Bean 的相应方法来完成实际的操作。

例程 19-5 是 listallfolders.jsp 的代码。

例程 19-5 listallfolders.jsp

```
<%
String operation=request.getParameter("operation");
String folderName=request.getParameter("folder");
String newFolderName=request.getParameter("newFolderName");

if(operation!=null && operation.equals("create"))mud.doCreateFolder(newFolderName);
if(operation!=null && operation.equals("delete"))mud.doDeleteFolder(folderName);
if(operation!=null &&
operation.equals("rename"))mud.doRenameFolder(folderName,newFolderName);
%>

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

```
<html>
<head><title>listallfolders</title></head>
<body>
<center><font face="Arial,Helvetica" font size=+3><b>Folder List</b></font></center><p>
<%@ include file="link.jsp" %>

<%
String operation=request.getParameter("operation");
String folderName=request.getParameter("folder");
String newFolderName=request.getParameter("newFolderName");

if(operation!=null && operation.equals("create"))mud.doCreateFolder(newFolderName);
if(operation!=null && operation.equals("delete"))mud.doDeleteFolder(folderName);
if(operation!=null &&
operation.equals("rename"))mud.doRenameFolder(folderName,newFolderName);

%>

<%
Folder folder;
Store store=mud.getStore();
folder = store.getDefaultFolder();
if (folder == null)
    throw new MessagingException("No folder is available");
Folder[] f = folder.list("%");
%>
<table width="75%" border=1 align=left>
<tr bgcolor="#FFCC66"><td rowspan=1 width="25%"><b>FolderName</b></td>
<td rowspan=1 width="25%"><b>Total Messages</b></td>
<td rowspan=1 width="25%"><b>Unread Messages</b></td>
</tr>

<% for(int i=0; i<f.length;i++){ %>
<tr valign=middle bgcolor="#FFFFCC">
<td><a href="listonefolder.jsp?folder=<%=f[i].getName()%>"><%=f[i].getName()%></a></td>
<td><%=f[i].getMessageCount()%></td>
<td><%= f[i].getUnreadMessageCount()%></td>
</tr>
<% } %>

<tr><td colspan=3>

<table width="50%" border=0 align=center>
<tr><td colspan=2 align=center><b><br>Folder Operation</b></td><tr>
```

```

<form action="listallfolders.jsp" >
<tr>
<td>select operation:</td>
<td>
<select name="operation">
<option value="create" selected> create folder
<option value="delete" >delete folder
<option value="rename" >rename folder
</select>
</td>
</tr>

<tr>
<td>select folder:</td>
<td>
<select name="folder">
<%
for(int i=0;i<f.length;i++){
    if(!f[i].getName().equalsIgnoreCase("inbox") && !f[i].getName().equalsIgnoreCase("Draft")&&
        !f[i].getName().equalsIgnoreCase("Trash")
        && !f[i].getName().equalsIgnoreCase("SendBox")){
        if(i==0)
            out.println("<option value='"+f[i].getName()+"' selected>" +f[i].getName());
        else
            out.println("<option value='"+f[i].getName()+"' >" +f[i].getName());
    }
}
%>
</select>
</td>
</tr>

<tr>
<td>new folder name:</td>
<td>
<input type="text" name="newFolderName">
</td>
</tr>
<tr><td colspan=2 align=center>
<input type="submit" name="submit" value="submit">
</td>
</tr>
</form>
</table>
</td></tr>
</table>

```

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

当客户创建新的邮件夹，或者修改原有的邮件夹的名字时，必须在“new folder name”文本框中输入新的邮件夹的名字，否则 listallfolders.jsp 调用 MailUserData 的 doCreateFolder 或 doRenameFolder 方法会抛出异常，异常由 errorpage.jsp 网页处理。当客户创建新的邮件夹，但没有输入新邮件夹名时，errorpage.jsp 返回的错误信息页面如图 19-14 所示。

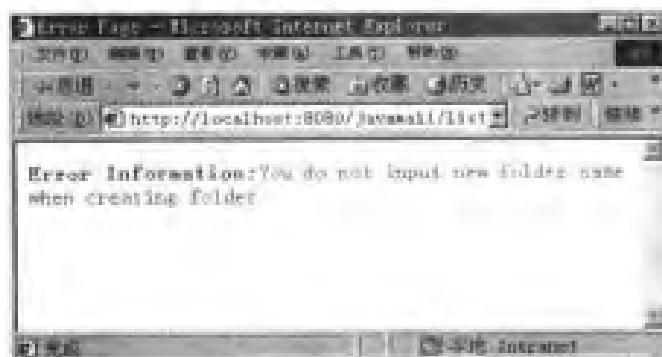


图 19-14 没有输入新邮件夹名时的错误信息

在 listallfolders.jsp 网页中，当用户选中某个邮件夹的链接，客户请求将由 listonefolder.jsp 处理。

#### 19.6.6 查看邮件夹中的邮件信息

listonefolder.jsp 用于显示用户指定的邮件夹中的邮件信息，包括邮件总数和未读邮件数目，而且列出了所有邮件的发送者、日期、标题和大小信息。此外，用户也可以删除邮件，页面如图 19-15 所示。



图 19-15 listonefolder.jsp 网页

通过 Message 类的 isSet(Flags.Flag.SEEN)方法可以判断这封邮件是否被阅读过。如果邮件没有被阅读，就采用粗体字显示。当用户选中某封邮件的链接，如果当前邮件夹为 Draft，那么就把客户请求传给 compose.jsp，否则就由 showmessage.jsp 来处理。以下是显示邮件标题以及设置链接的代码：

```
<%
String link="";
if(f.getName().equals("Draft")){
    link="compose.jsp?edit=true";
    mud.setCurrMsg(m);
}
else link="showmessage.jsp" + "?messageindex=" + i;
if(!m.isSet(Flags.Flag.SEEN))out.println("<b>");
out.println("<a href='"+link+">" +
    ((m.getSubject() != null)&& !m.getSubject().equals("") ?
        m.getSubject() : "<i>No Subject</i></a>"));
if(!m.isSet(Flags.Flag.SEEN))out.println("</b>");
%>
```

例程 19-6 是 listonefolder.jsp 的代码。

例程 19-6 listonefolder.jsp

---

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

<html>
<head><title>listonefolder</title></head>
<body>
<%
String folderName=request.getParameter("folder");
SimpleDateFormat df = new SimpleDateFormat("yy.MM.dd 'at' HH:mm:ss ");
Folder f=null;
if(folderName!=null){
    f=mud.getStore().getFolder(folderName);
    mud.setCurrFolder(f);
} else{
    f=mud.getCurrFolder();
    folderName=f.getName();
}
if(!f.isOpen())f.open(Folder.READ_WRITE);
int msgCount = f.getMessageCount();
int unReadCount = f.getUnreadMessageCount();

//delete message
int arrayOpt[] = new int[msgCount];
for(int i=1;i<=msgCount;i++){
    String optS=request.getParameter("delIndex"+i);
    if(optS!=null) arrayOpt[i-1]=1;
```

```
}

mud.doDeleteMessage(arrayOpt,f);

//refresh msgCount and unReadCount
if(f.isOpen())f.close(true);
f.open(Folder.READ_WRITE);
msgCount = f.getMessageCount();
unReadCount = f.getUnreadMessageCount();
%>

<center><font size="+3"><b>CurrentFolder:<%=folderName%></b></font></center><p>
<%@ include file="link.jsp" %>

<b>Total Messages:<%=msgCount%></b>
<b>Unread Messages:<%=unReadCount%></b>

<form action="listonefolder.jsp">
<table cellpadding=1 cellspacing=1 width="100%" border=1>
<tr bgcolor="#ffffcc">
<td width="5%"></td>
<td width="35%"><b>Sender</b></td>
<td width="20%"><b>Date</b></td>
<td width="30%"><b>Subject</b></td>
<td width="10%"><b>Size</b></td>
</tr>

<%
Message m = null;
// for each message, show its headers
for (int i = 1; i <= msgCount; i++) {
    m = f.getMessage(i);
    // if message has the DELETED flag set, don't display it
    if (m.isSet(Flags.Flag.DELETED))
        continue;
%>

<%--opt --%>
<tr valign="middle">
<td width=5%><input TYPE=CHECKBOX NAME="delIndex<%=i%>"></td>

<%-- from --%>
<td width="35%">
<%  if(!m.isSet(Flags.Flag.SEEN)) out.print("<b>"); %>
<%  out.println((m.getFrom() != null) ? m.getFrom()[0].toString() : " "); %>
<%  if(!m.isSet(Flags.Flag.SEEN))out.print("</b>"); %>
</td>
```

```
<%--date --%>
<td width="20%">
<%if(!m.isSet(Flags.Flag.SEEN))out.println("<b>");%>
<%out.println(df.format((m.getSentDate()!=null)?m.getSentDate(): m.getReceivedDate())); %>
<%if(!m.isSet(Flags.Flag.SEEN))out.println("</b>");%>
</td>

<%--subject & link --%>
<td width="30%">
<%
String link="";
if(f.getName().equals("Draft")){
    link="compose.jsp?edit=true";
    mud.setCurrMsg(m);
}
else link="showmessage.jsp" + "?messageindex=" + i;

if(!m.isSet(Flags.Flag.SEEN))out.println("<b>");
    out.println("<a href="+link+">" +
        ((m.getSubject() != null)&& !m.getSubject().equals("") ?
            m.getSubject() : "<i>No Subject</i></a>"));
if(!m.isSet(Flags.Flag.SEEN))out.println("</b>");
%>
</td>

<%-- size--%>
<td width="10%">
<%
if(!m.isSet(Flags.Flag.SEEN))out.println("<b>"); 
    out.println(m.getSize()+"Bytes");
if(!m.isSet(Flags.Flag.SEEN))out.println("</b>"); 
%>
</td>
</tr>
<%
}
%>
</table>
<p><input type="submit" name="submit" value="delete messages"></form>
</body></html>
```

### 19.6.7 查看邮件内容

showmessage.jsp 用于显示邮件的内容，如图 19-16 所示。



图 19-16 showmessage.jsp 网页

showmessage.jsp 读取用户请求参数 messageindex，它代表了邮件在邮件夹中的序号。然后从当前 Folder 中取出指定邮件，再把它封装到 PMessage 对象中，最后在网页上显示出来。邮件在邮件夹中的序号从 1 开始。例程 19-7 是 showmessage.jsp 的源代码。

例程 19-7 showmessage.jsp

```

<%@ include file="common.jsp" %>
<html>
<head><title>show message</title></head>
<body >

<%
Folder folder=mud.getCurrFolder();
Message currmsg=null;
int msgNum=1;
String messageindex=request.getParameter("messageindex");
if(messageindex!=null){
msgNum=Integer.parseInt(messageindex);
currmsg=folder.getMessage(msgNum);
mud.setCurrMsg(currmsg);
}else
currmsg=mud.getCurrMsg();

PMessage displayMsg=new PMessage(currmsg);
%>

<center><font size="-3"><b>
<%  out.println("Message in "+folder.getName()+" folder "); %>
</b></font></center><p>
<%@ include file="link.jsp" %>

<a href="compose.jsp?reply=true" >Reply</a>

```

```

<a href="listonefolder.jsp?delIndex<%=msgNum%>=on" >Delete</a>

<%-- display message--%>
<table width=90%>
<tr><td>
<b>Date:</b> <%=displayMsg.getDate()%><br>
<b>From:</b> <%=displayMsg.getFrom()%><br>
<b>To:</b> <%=displayMsg.getTo()%><br>
<b>CC:</b> <%=displayMsg.getCC()%><br>
<b>Subject:</b> <%=displayMsg.getSubject()%><br>
<pre><%=displayMsg.getText()%></pre>
</td></tr></table>
</body></html>

```

如果用户选择“Reply”链接，客户请求由 compose.jsp 来处理，如果选择“Delete”链接，客户请求由 listonefolder.jsp 来处理，listonefolder.jsp 将删除请求参数中指定的邮件。

### 19.6.8 创建和发送邮件

compose.jsp 提供了编辑邮件的表单，用户进入 compose.jsp 有 3 个入口：

- 在任意一个网页上选择“Compose”链接，此时 compose.jsp 将创建一封新邮件
- 在 showmessage.jsp 中选择“Reply”链接，此时 compose.jsp 先创建一封回复邮件，再让用户编辑这封邮件
- 当 listonefolder.jsp 显示 Draft 邮件夹时，用户选择某封邮件的链接，此时 compose.jsp 将提供编辑这封邮件的页面

用户创建新邮件的 compose.jsp 网页如图 19-17 所示。



图 19-17 compose.jsp 网页

例程 19-8 是 compose.jsp 的源代码。

例程 19-8 compose.jsp

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

<html>
<head><title>composemessage</title></head>
<body>
<center><font size="+3"><b>Compose Message</b></font></center><p>
<%@ include file="link.jsp" %>

<%
String operation=request.getParameter("operation");
String reply=request.getParameter("reply");
String edit=request.getParameter("edit");

String to = request.getParameter("to");
String cc = request.getParameter("cc");
String bcc = request.getParameter("bcc");
String subj = request.getParameter("subject");
String text = request.getParameter("text");

PMessage displayMsg=new PMessage();
// send message
if (operation != null && operation.equals("send")) {
    displayMsg=new PMessage(to, cc, bcc, subj, text);
    mud.doSendMessage(displayMsg);
    out.println("Message is sent to "+to);
}

// save message
if (operation != null && operation.equals("save")) {
    displayMsg=new PMessage(to, cc, bcc, subj, text);
    mud.doSaveMessage(displayMsg);
    out.println("Message is saved to Draft");
}

//get reply message
if(reply!=null){
    Message currmsg=mud.getCurrMsg();
    displayMsg=new PMessage(currmsg.reply(true));
}

//edit Draft message
if(edit!=null) {
    displayMsg=new PMessage(mud.getCurrMsg());
```

```
}

%>

<form action="compose.jsp" method="post">
<table border="0" width="100%">
<tr><td width="16%" height="22"><p align="right"><b>to:</b></td>
<td width="84%" height="22"><input type="text" name="to" value="<%displayMsg.getTo()%>" size="30" ></td>
</tr>
<tr>
<td width="16%"><p align="right"><b>cc:</b></td>
<td width="84%"><input type="text" name="cc" value="<%displayMsg.getCC()%>" size="30"></td>
</tr>
<tr>
<td width="16%"><p align="right"><b>bcc:</b></td>
<td width="84%"><input type="text" name="bcc" value="<%displayMsg.getBCC()%>" size="30"></td>
</tr>
<tr>
<td width="16%"><p align="right"><b>subject:</b></td>
<td width="84%"><input type="text" name="subject" value="<%displayMsg.getSubject()%>" size="30"></td>
</tr>
<tr>
<td width="16%">&nbsp;</td>
<td width="84%"><textarea name="text" rows="5" cols="40">
<%displayMsg.getText()%</textarea>
</td>
</tr>
</table>
<center>
<b><input type="radio" name="operation" value="save">save draft
<input type="radio" name="operation" value="send" checked>send </b>
<input type="submit" name="submit" value="submit">
<input type="reset" name="reset" value="reset">
</center>
</form>

</body></html>
```

如果用户选择“send”选项，compose.jsp 将发送这封邮件，如果选择“save draft”选项，compose.jsp 将把邮件保存到 Draft 邮件夹中。

### 19.6.9 退出邮件系统

logout.jsp 负责退出邮件系统，结束当前 Http Session，并且提供了再次登录的链接，如图 19-18 所示。



图 19-18 logout.jsp 网页

调用 Store 的 close 方法，将会断开与接收邮件服务器的连接。以下是 logout.jsp 的代码：

```
<%@ include file="common.jsp" %>
<%
    String username=mud.getURLName().getUsername();
    mud.getStore().close();
    session.invalidate();
%>
<html>
<head><title>Logout</title></head>
<body>
<h3>Goodbye,<%=username%>!</h3><br>
<strong><a href="login.jsp">Login again</a></strong>
</body>
</html>
```

## 19.7 在 Tomcat 中配置 Mail Session

在第 6 章已经讲过可以把数据源 DataSource 作为一种 JNDI 资源在 Tomcat 中配置。Tomcat 的 org.apache.commons.dbcp.BasicDataSourceFactory 工厂负责创建 DataSource 对象。在程序中，可以通过 javax.naming.Context 的 lookup 方法来获得 JNDI DataSource 资源的引用。

同样，也可以把 Mail Session 作为一种 JNDI 资源在 Tomcat 中配置。Tomcat 提供了创建 Mail Session 对象的工厂：org.apache.naming.factory.MailSessionFactory。

### 19.7.1 在 server.xml 中配置 Mail Session 资源

可以在 server.xml 的 localhost 对应的<Host>元素下加入如下 javamail 应用的<Context>元素，在<Context>元素中定义了 Mail Session 资源，它的 JNDI 名字为 mail/session，Mail Session 的有关属性，如邮件发送协议、邮件接收协议和邮件发送服务器，在<parameter>子元素中设置如下：

```
<Context path="/javamail" docBase="javamail" debug="0"
reloadable="true" >

    <Resource name="mail/session" auth="Container"
type="javax.mail.Session"/>

    <ResourceParams name="mail/session">
        <parameter>
            <name>factory</name>
            <value>org.apache.naming.factory.MailSessionFactory</value>
        </parameter>

        <parameter>
            <name>mail.transport.protocol</name>
            <value>smtp</value>
        </parameter>

        <parameter>
            <name>mail.store.protocol</name>
            <value>imap</value>
        </parameter>

        <parameter>
            <name>mail.smtp.host</name>
            <value>localhost</value>
        </parameter>

    </ResourceParams>

</Context>
```

### 19.7.2 在 web.xml 中加入对 JNDI Mail Session 资源的引用

由于 javamail 应用将引用 JNDI Mail Session 资源，所以应该在 web.xml 中加入<resource-ref>元素：

```
<resource-ref>
    <description>Java Mail Session</description>
    <res-ref-name>mail/session</res-ref-name>
```

```
<res-type>javax.mail.Session</res-type>
<res-auth>Container</res-auth>
</resource-ref>
```

### 19.7.3 在 javamail 应用中获取 JNDI Mail Session 资源

在 19.6.4 节介绍的 connect.jsp 是通过 javax.mail.Session 的静态方法 getDefaultInstance 来创建 Session 对象的：

```
// Get a Session object
Session mailsession = Session.getDefaultInstance(props, null);
```

现在，将修改 connect.jsp 代码，让它从 Tomcat 的 Mail Session 工厂中取得 Session 对象的引用：

```
Context ctx = new InitialContext();
if(ctx == null )
    throw new Exception("No Context");
Session mailsession =(Session)ctx.lookup("java:comp/env/mail/session");
```

Mail Session 的属性已经在 server.xml 中设置，Mail Session 由 Tomcat 容器来负责创建和管理，所以在 connect.jsp 中不需要再为 Mail Session 设置属性，例程 19-9 是修改后的 connect.jsp。

例程 19-9 connect.jsp

```
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.naming.*" %>

<%@ page errorPage="errorpage.jsp" %>
<jsp:useBean id="mud" scope="session" class="mypack.MailUserData"/>

<%
    String hostname = request.getParameter("hostname");
    String username = request.getParameter("username");
    String password = request.getParameter("password");

    Context ctx = new InitialContext();
    if(ctx == null )
        throw new Exception("No Context");
    Session mailsession =(Session)ctx.lookup("java:comp/env/mail/session");

    // Get a Store object
    Store store = mailsession.getStore("imap");
%>
```

```

<%
try{
    // Connect
    store.connect(hostname,username, password);
}catch(Exception e){
    request.setAttribute("loginfail","true");
%>
<jsp:forward page="login.jsp" />
<%
//end of catch
%>

<%
// save stuff into MUD
mud.setSession(mailsession);
mud.setStore(store);

//create and open default Trash, Draft and sendbox folder

Folder folder=store.getFolder("Trash");
if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

folder=store.getFolder("SendBox");
if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

folder=store.getFolder("Draft");
if(!folder.exists())folder.create(Folder.HOLDS_MESSAGES);

folder.open(Folder.READ_WRITE);

//save draft into MUD
URLName url = new URLName("imap",hostname, -1, "inbox", username, password);
mud.setURLName(url);
%>
<jsp:forward page="listallfolders.jsp" />

```

## 19.8 发布和运行 javamail 应用

在本书配套光盘的 sourcecode/javamails/ 目录下提供了 javamail 应用的两个版本，这两个版本的区别在于：第一个版本的 connect.jsp 直接在程序中创建 Mail Session 对象，第二个版本的 connect.jsp 从 Tomcat 容器中获取 JNDI Mail Session 资源。

运行这两个版本之前，都应该确保 Mail 服务器已经启动。

### 1. 发布 javamail version0 应用

javamail version0 的发布描述文件 web.xml 的<web-app>元素内容为空，不需要做任何修改。在 javamail\WEB-INF\lib 目录下已经存放了 activation.jar 和 mail.jar 文件。此外，也不需要在 Tomcat 的配置文件 server.xml 中加入<Context>元素。

因此，发布 version0 非常简单，只要把 sourcecode/javamails/version0/下的整个 javamail 目录拷贝到<CATALINA\_HOME>/webapps 目录下即可。

### 2. 发布 javamail version1 应用

javamail version1 的发布描述文件 web.xml 中已经配置好了<resource-ref>元素，用于声明对 JNDI Mail Session 资源的引用。不需要对 web.xml 做任何修改。

由于 javamail version1 的 Mail Session 对象由 Tomcat 容器来创建，必须把 activation.jar 和 mail.jar 文件拷贝到<CATALINA\_HOME>/common/lib 目录下。这样，Tomcat 容器才可以访问这些 JAR 文件。

此外，应该在 server.xml 中 localhost 的<Host>元素下加入<Context>元素，在<Context>元素中加入配置 Mail Session 资源的代码。在 sourcecode/javamails/version1/javamail/server\_modify.xml 文件中提供了这段配置代码，可以直接把它拷贝到 server.xml 中。

最后，只要把 sourcecode/javamails/version1/下的整个 javamail 目录拷贝到< CATALINA\_HOME>/webapps 目录下即可。

对于这两个版本，它们提供的客户界面和功能都是一样的。当 Tomcat 服务器启动后，访问 <http://localhost:8080/javamail/login.jsp>，就可以进入 javamail 应用的登录界面。

## 19.9 小 结

邮件服务器按照为用户提供 E-mail 发送和接收的服务不同，可以分为发送邮件服务器和接收邮件服务器。发送邮件服务器常用的是 SMTP，接收邮件服务器使用接收邮件协议，常用的有 POP3 协议和 IMAP。与 POP3 协议相比，IMAP 为客户提供更多的对邮件服务器上邮件的控制权限，如管理邮件和邮件夹等。Java Mail API 是 SUN 为 Java 开发者提供的公用 Mail API 框架，它支持各种电子邮件通信协议，如 IMAP、POP3 和 SMTP，为 Java 应用程序提供了电子邮件处理的公共接口。javax.mail.Session 类定义了一个基本邮件会话，是 Java Mail API 最高层入口类。所有其他类都是经由这个 Session 才得以生效。本章介绍了一个 Java Mail Web 应用，它允许用户访问 IMAP 邮件服务器上的邮件账号。本章提供了 javamail 应用的两个版本，前者在 Web 应用程序中创建 Mail Session 对象，后者从 Tomcat 容器中取得 JNDI Mail Session 资源。

# 第 20 章 Tomcat 与 Apache SOAP 集成

随着计算机技术的不断发展，现代企业面临的环境越来越复杂，其信息系统大多数为多平台、多系统的复杂系统。这就要求今天的企业解决方案具有广泛的兼容能力，可以支持不同的系统平台、数据格式和多种连接方式，要求在 Internet 环境下，实现的系统是松散耦合的、跨平台的、与语言无关的、与特定接口无关的，而且要提供对 Web 应用程序的可靠访问。

Web 服务应支持不同的系统之间用“软件 - 软件对话”的方式相互调用，打破软件应用、网站和各种设备之间的格格不入的状态。SOAP 在 Web 服务堆栈中作为基于 XML 信息交换的一种非常普遍的协议，对于实现“基于 Web 无缝集成”的目标发挥着非常重要的作用。

本章首先介绍 SOAP 的基本概念，然后介绍把 Apache SOAP 集成到 Tomcat 中，创建 SOAP 服务和 SOAP 客户的方法。

## 20.1 SOAP 简介

SOAP (Simple Object Access Protocol) 即简单对象访问协议，是在分散或分布式的环境中交换信息的简单协议，它以 XML 作为数据传送方式。

SOAP 采用的通信协议可以是 HTTP/HTTPS (现在用得最广泛) 协议，也可以是 SMTP/POP3 协议，还可以是为一些应用而专门设计的特殊通信协议。两个系统之间通过 SOAP 通信的过程如图 20-1 所示。

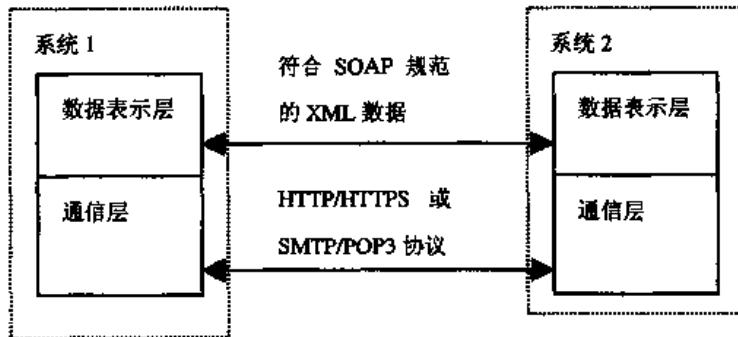


图 20-1 系统间采用 SOAP 通信

SOAP 系统有两种工作模式，一种称为 RPC (Remote Procedure Call)，另一种叫法不统一，在 Microsoft 的文档中称做 Document-Oriented，而在 Apache 的文档中，称为 Message-Oriented，这是一种可以利用 XML 交换更为复杂的结构数据的应用，通常以 SMTP 作为传输协议。下文将集中讨论 RPC。

可以把 SOAP RPC 简单地理解为这样一个开放协议：SOAP=RPC+HTTP+XML。它有

以下特征：

- 采用 HTTP 作为通信协议，采用客户/服务模式
- RPC 作为统一的远程方法调用途径
- XML 作为数据传送的格式，允许服务提供者和客户经过防火墙在 Internet 上进行通信交互

SOAP RPC 的工作流程如图 20-2 所示，从图中可以看到，SOAP RPC 的工作原理非常类似于 Web 的请求/响应方式，两者都以 HTTP 作为通信协议，不同之处在于 Web 客户和 Web 服务器之间传输的是 HTML 数据，而在 SOAP RPC 模式中，SOAP 客户和 SOAP 服务之间传输的是符合 SOAP 规范的 XML 数据。SOAP RPC 采用 HTTP 作为通信协议，它是个无状态协议，无状态协议非常适合松散耦合系统，而且对于负载平衡等都有潜在的优势和贡献。

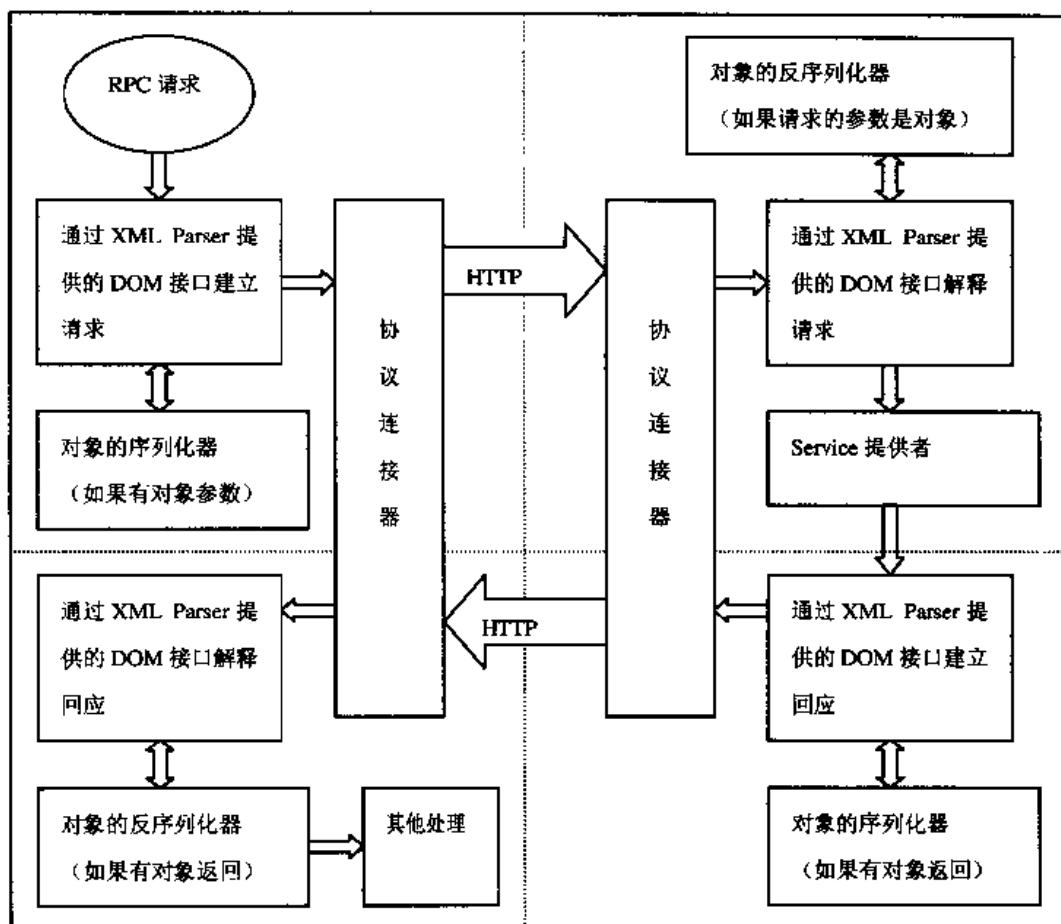


图 20-2 SOAP RPC 的工作流程

SOAP 客户访问 SOAP 服务的流程如下。



- (1) 客户程序创建一个 XML 文档，它包含了提供服务的服务器的 URI、客户请求调用的方法名和参数信息。如果参数是对象，则必须进行序列化操作。
- (2) 目标服务器接收到客户程序发送的 XML 文档，对其进行解析，如果参数是对象，

先对其进行反序列化操作，然后执行客户请求的方法。

(3) 目标服务器执行方法完毕后，如果方法的返回值是对象，则先对其进行序列化操作，然后把返回值以 XML 文档的形式返回给客户。

(4) 客户程序接收到服务器发来的 XML 文档，如果返回值是对象，则先对其进行反序列化操作，最后获得返回值。



**XML Parser** 指的是 XML 解析器，**DOM** (Document Object Model) 接口指的是文档对象模型接口。

SOAP 客户和 SOAP 服务之间采用符合 SOAP 规范的 XML 数据进行通信，它的形式如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <sayHelloResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <sayHelloReturn xsi:type="xsd:string">Hello:weiqin</sayHelloReturn>
        </sayHelloResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

以上是一个 SOAP 服务向 SOAP 客户发回的响应数据。XML 数据的根元素为 **<soapenv:Envelope>**，它可以包含 **<soapenv:Head>** 和 **<soapenv:Body>** 子元素，在 **<soapenv:Body>** 元素下包含了具体的客户请求或服务响应数据。

以上**<soapenv:Envelope>** 元素中定义了一些 **xmlns** 属性，它代表 XML Name Space，即 XML 命名空间，在本书附录 C 对 XML 命名空间的概念进行了解释。

## 20.2 建立 Apache SOAP 环境

Apache SOAP 是 Apache 软件组织对 SOAP 规范的实现，建立在 IBM 的 SOAP4J 的基础上，与所有其他 Apache 工程类似，Apache SOAP 源代码开放。

Apache SOAP 支持 RPC 和 Message-Oriented 两种模式，在本文讨论 RPC 模式。建立 Apache SOAP 环境需要准备 Apache SOAP 软件以及其他相关的 JAR 文件。这些文件的清单、下载地址以及在本书配套光盘上的位置参见表 20-1。

表 20-1 Apache SOAP 所需的文件

文 件	下 载 地 址	在本书配套光盘上的位 置
Apache SOAP 软件	<a href="http://xml.apache.org/soap/index.html">http://xml.apache.org/soap/index.html</a>	software/soap-bin-2.3.1.zip
mail.jar	<a href="http://java.sun.com/products/javamail">http://java.sun.com/products/javamail</a>	lib 目录下
activation.jar	<a href="http://java.sun.com/products/javabeans/glasgow/jaf.html">http://java.sun.com/products/javabeans/glasgow/jaf.html</a>	lib 目录下
axis.jar	<a href="http://xml.apache.org/axis-j/index.html">http://xml.apache.org/axis-j/index.html</a>	lib 目录下



Apache SOAP 要求有 1.1.2 版本或更高的 Apache Xerces，负责解析 Apache SOAP 的 XML 文档。Apache Xerces 是一种 XML 解析器，它支持 DOM Level 2 规范，支持 XML 命名空间。

把 soap-bin-2.3.1.zip 解压到本地硬盘，假定它的根目录为<SOAP\_HOME>，在 lib 子目录下有一个 soap.jar 文件，它包含了 Apache SOAP 所有的类文件，在编译和运行 SOAP 客户程序时需要用到该文件。另外，在 webapps 子目录下有一个 soap.war 文件，下面会用到这个 WAR 文件。

## 20.3 在 Tomcat 上发布 Apache-SOAP Web 应用

运行 Apache SOAP 工程，需要一个 Servlet/JSP 容器。在此，通过 Tomcat 服务器来发布 Apache-SOAP Web 应用，步骤如下。



(1) 把 activation.jar、mail.jar 和 xerces.jar 拷贝到<CATALINA\_HOME>/common/lib 目录下。

(2) 把<SOAP\_HOME>/webapps 目录下的 soap.war 文件拷贝到<CATALINA\_HOME>/webapps 目录下。



在 soap.war 中已经包含了 Apache SOAP 本身的类文件，因此不需要把 soap.jar 文件拷贝到<CATALINA\_HOME>/common/lib 目录下。

这样，Apache-SOAP Web 应用就发布成功了，启动 Tomcat 服务器，访问 <http://localhost:8080/soap>，会看到如图 20-3 所示的网页。



图 20-3 Apache-SOAP 主页

选择“Run”链接将进入 Apache SOAP 的控制页面，如图 20-4 所示。Apache SOAP 控制页面提供了管理 SOAP 服务的功能，包括：

- List：显示已经发布的 SOAP 服务
- Deploy：发布 SOAP 服务
- Un-deploy：删除 SOAP 服务



图 20-4 Apache-SOAP 的控制页面

## 20.4 创建 SOAP 服务

Tomcat 充当 Apache-SOAP Web 应用的容器，而 Apache-SOAP Web 应用充当 SOAP 服务的容器，Apache SOAP 客户程序可以通过 Apache SOAP API 来发出 RPC 请求，访问 SOAP 服务，如图 20-5 所示。

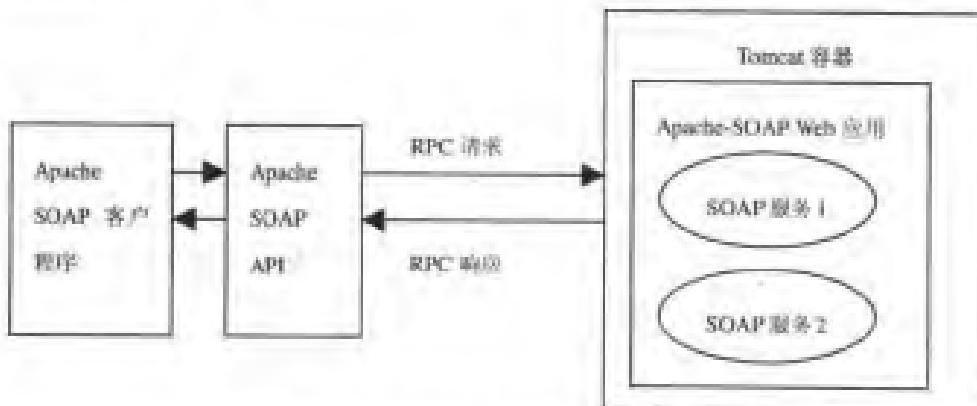


图 20-5 SOAP 客户和 SOAP 服务

创建基于 RPC 的 SOAP 服务包括两个步骤：

- 创建提供 SOAP 服务的 Java 类
- 创建 SOAP 服务的发布描述符文件

#### 20.4.1 创建提供 SOAP 服务的 Java 类

在提供 SOAP 服务的 Java 类中可以定义被 SOAP 客户调用的方法，这些方法必须声明为 public 类型。此外，这些方法的参数或返回类型如果是类，则必须已经实现了 java.io.Serializable 接口。默认情况下，这些参数类型以及返回类型必须在 SOAP 中注册过，以下是已经在 SOAP 中注册的 Java 类型：

- Java 基本类型 (byte,int,short,long,float,double,char,boolean) 以及它们的包装类
- Java 数组
- java.lang.String
- java.util.Date
- java.util.GregorianCalendar
- java.util.Vector
- java.util.Hashtable
- java.util.Map
- java.math.BigDecimal
- javax.mail.internet.MimeBodyPart
- java.io.InputStream
- javax.activation.DataSource
- javax.activation.DataHandler
- org.apache.soap.util.xml.QName
- org.apache.soap.rpc.Parameter
- org.lang.Object

以下是一个简单的 SOAP 服务类，它包含了一个方法 sayHello，这个方法的参数和返回类型都是 java.lang.String，它实现了 Serializable 接口，符合上述 SOAP 规范。

```
package mypack;
public class HelloService {
    public String sayHello(String username) {
        return "Hello:" +username;
    }
}
```

编译这个 Java 类不需要在 classpath 中引入任何与 SOAP 相关的 JAR 文件。编译完毕后，应该把 HelloService.class 文件拷贝到以下目录。

```
<CATALINA_HOME>/webapps/soap/WEB-INF/classes/mypack
```

#### 20.4.2 创建 SOAP 服务的发布描述符文件

SOAP 服务的发布描述符文件用来定义 SOAP 服务，包括提供服务的类名，以及可被

SOAP 客户调用的方法。以下是 HelloService 的发布描述文件，名为 DeploymentDescriptor.xml：

```

<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:helloService">
    <isd:provider type="java"
        scope="Application"
        methods="sayHello">
        <isd:java class="mypack.HelloService"/>

    </isd:provider>
    <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
</isd:service>

```

以上文件配置了<service>、<provider>和<java>元素，下面分别介绍这几种元素。

### 1. <service>元素

<service>元素定义了一项 SOAP 服务，它是其他元素的根元素。它有两个属性：xmlns 表示 XML Name Space，即 XML 命名空间；id 是这项服务的惟一标识符，SOAP 客户将根据 id 属性来访问 SOAP 服务。

### 2. <provider>元素

<provider>定义了 SOAP 服务的实际内容，它的属性描述参见表 20-2 所示。

表 20-2 <provider>元素的属性

属性	描述
type	指定实现 SOAP 服务的语言
scope	指定 SOAP 服务的存在范围
methods	指定可被客户调用的方法名，多个方法之间以空格隔开

以上 scope 属性用来定义服务实例的生存周期，它的概念和本书 8.3 节中介绍的 JavaBean 的 scope 属性很相似。scope 可以是下列值之一：

- page：服务实例一直有效，直至应答发送完毕或把请求传递给了另一个页面
- request：服务实例在请求处理期间，不管是否出现请求传递，一直有效
- session：服务实例对于整个会话都有效
- application：服务实例在 Apache-SOAP Web 应用运行期间一直有效

### 3. <java>元素

<java>元素只有一个属性 class，它用来指定实现服务的 java 类。

## 20.5 管理 SOAP 服务

Apache SOAP 工程提供了两种管理 SOAP 服务的工具，一种是基于 Web 方式的（如图 20-4 所示），还有一种是基于命令行的。这两种工具都能提供显示、发布和删除 SOAP 服务的功能。

下面将介绍基于命令行的 SOAP 服务管理工具，它是由 org.apache.soap.server.ServiceManagerClient 类来实现的，运行 ServiceManagerClient 类的基本语法如下：

```
java org.apache.soap.server.ServiceManagerClient  
http://localhost:8080/soap/servlet/rpcrouter commandname
```



运行以上命令时，应该把 soap.jar、mail.jar 和 activation.jar 加入到 classpath 中。

运行 ServiceManagerClient 类至少需要提供两个参数，其中第一个参数是固定的，它代表了 SOAP 应用中的 rpcrouter Servlet。这个 Servlet 是 Apache SOAP 的核心，它负责管理和调用所有的 SOAP 服务；第二个参数指定具体的命令，可选值包括 list、deploy 和 undeploy。

### 20.5.1 发布 SOAP 服务

发布 SOAP 服务的命令如下：

```
java org.apache.soap.server.ServiceManagerClient  
http://localhost:8080/soap/servlet/rpcrouter deploy DeploymentDescriptor.xml
```

以上命令将把 DeploymentDescriptor.xml 文件中声明的 SOAP 服务发布到 SOAP 服务器中。



发布 helloservice 服务时，应该确保已经把 HelloService.class 拷贝到 <CATALINA\_HOME>/webapps/soap/WEB-INF/classes/mypack 目录下。

### 20.5.2 查看 SOAP 服务

查看已经发布的 SOAP 服务的命令：

```
java org.apache.soap.server.ServiceManagerClient  
http://localhost:8080/soap/servlet/rpcrouter list
```

如果 helloservice 服务已经发布成功，运行 list 命令后将显示这项服务的 id：

urn:helloservice

此外，也可以通过 Apache SOAP 的基于 Web 的管理工具来查看服务，访问 <http://localhost:8080/soap/admin/index.html>，然后选择“List”链接，将会看到如图 20-6 所示的网页。



图 20-6 显示 SOAP 服务的网页

在图 20-6 网页中选择“urn:helloservice”链接，会显示 helloservice 服务的具体内容，如图 20-7 所示。



图 20-7 显示 helloservice 服务的网页

### 20.5.3 删除 SOAP 服务

删除已发布的 SOAP 服务的命令：

```
java org.apache.soap.server.ServiceManagerClient  
http://localhost:8080/soap/servlet/rpcrouter undeploy urn:helloservice
```

运行以上命令将删除 helloservice 服务。如果此时再查看 SOAP 服务，将不会显示这项服务。

在本书配套光盘的 sourcecode/chapter20 目录下的 test.bat 文件中提供了以上发布、查看和删除 helloservice 服务的命令。

## 20.6 创建和运行 SOAP 客户程序

SOAP RPC 客户程序可以通过 SOAP API 发出 RPC 请求，来调用 SOAP 服务的方法。例程 20-1 是访问 helloservice 服务的 sayHello 方法的客户程序。

例程 20-1 HelloClient.java

```
package mypack;  
  
import java.io.*;  
import java.net.*;  
import java.util.*;  
import org.apache.soap.*;  
import org.apache.soap.rpc.*;  
  
public class HelloClient {  
  
    public static void main(String[] args) throws Exception {  
  
        URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");  
  
        String username="Guest";  
        if(args.length!=0) username=args[0];  
  
        // Build the call.  
        Call call = new Call();  
        call.setTargetObjectURI("urn:helloservice");  
        call.setMethodName("sayHello");  
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);  
        Vector params = new Vector();  
        params.addElement(new Parameter("username", String.class, username, null));  
        call.setParams (params);
```

```

// make the call; note that the action URI is empty because the
// XML-SOAP rpc router does not need this. This may change in the
// future.
Response resp = call.invoke(url, "");

// Check the response.
if ( resp.generatedFault() ) {

    Fault fault = resp.getFault();
    System.out.println("The call failed: ");
    System.out.println("  Fault Code   = " + fault.getFaultCode());
    System.out.println("  Fault String = " + fault.getFaultString());
}

else {
    Parameter result = resp.getReturnValue();
    System.out.println(result.getValue());
}
}
}

```

HelloClient 访问 helloservice 服务包含如下步骤。



(1) 定义访问 Apache-SOAP Web 应用的 rpcrouter Servlet 的 URL:

```
URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");
```

(2) 读取命令行参数，设置 username 的值:

```
String username="Guest";
if(args.length!=0) username=args[0];
```

(3) 创建 org.apache.soap.rpc.RPCMessage.Call 对象，Call 对象提供了发出 RPC 请求的方法。在发出 RPC 请求之前，首先应该为 Call 对象设置相关的 RPC 请求信息:

```

// Build the call.
Call call = new Call();
call.setTargetObjectURI("urn:helloservice");
call.setMethodName("sayHello");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
Vector params = new Vector();
params.addElement(new Parameter("username", String.class, username, null));
call.setParams(params);

```

以上代码为 Call 对象设置了如下信息:

- 被调用服务的 id，它通过 Call 对象的 setTargetObjectURI()方法设置
- 待调用方法的名字，它通过 Call 对象的 setMethodName()方法设置
- 待调用方法的参数的编码方式，它通过 Call 对象的 setEncodingStyleURI()方法设置
- 待调用方法的参数值通过 Call 对象的 setParams()方法设置

`setParams()`方法的参数是一个 Java Vector（向量）。这个向量包含所有的参数，向量中索引为 0 的参数是被调用方法的第一个参数，索引为 1 的参数是被调用方法的第二个参数，依次类推。向量中的每一个元素都是一个 `org.apache.soap.rpc.Parameter` 的实例。`Parameter` 构造方法要求指定参数的名字、Java 类型和值，还有一个可选的编码方式。如果指定了 null 编码方式（正如本例所做的那样），则使用 `Call` 对象的默认编码方式。

#### (4) 调用 Call 实例的 invoke 方法：

```
Response resp = call.invoke(url, "");
```

`invoke` 方法负责向 SOAP 服务器发出客户指定的 RPC 请求，服务器接收到 RPC 请求后，会在服务器端执行相应服务方法，然后把返回值传给客户。`invoke` 方法返回 `Response` 对象，它封装了 SOAP 的响应结果，如果服务器端执行相应服务方法发生错误，将把错误信息封装在 `Response` 对象中，否则就把服务方法的正常返回值封装在 `Response` 对象中。通过 `Response` 的 `generatedFault` 方法可以获得错误对象，通过 `Response` 的 `getReturnValue` 方法可以获得服务方法的正常返回值。

```
// Check the response.  
if ( resp.generatedFault() ) {  
    Fault fault = resp.getFault();  
    System.out.println("The call failed: ");  
    System.out.println("  Fault Code  = " + fault.getFaultCode());  
    System.out.println("  Fault String = " + fault.getFaultString());  
}  
  
else {  
    Parameter result = resp.getReturnValue();  
    System.out.println(result.getValue());  
}
```

编译 `HelloClient` 时，应该把 `soap.jar` 加入到 classpath 中。

运行 `HelloClient` 时，应该确保 `helloservice` 服务已经发布，并且 Apache-SOAP Web 应用处于运行状态。此外，运行 `HelloClient` 时需要把 `soap.jar`、`mail.jar` 和 `activation.jar` 加入到 classpath 中，然后在命令行运行如下命令：

```
java mypack.HelloClient weiqin
```

将会看到下面的输出内容：

```
Hello:weiqin
```

为了便于读者运行，本书配套光盘的 `sourcecode/chapter20` 目录下包含了本章例子的源文件，并且提供了编译和运行程序的脚本：

- `compile.bat` 提供了编译 `HelloService.java` 和 `HelloClient.java` 的命令
- `test.bat` 文件提供了发布、查看和删除 `helloservice` 服务以及运行 `HelloClient` 的命令
- 在 `lib` 子目录下包含了编译和运行客户程序所需的所有 JAR 文件

可以把整个 `chapter20` 目录拷贝到本地硬盘，再按以下步骤运行本章例子。

步骤

- (1) 把 activation.jar、mail.jar 和 xerces.jar 拷贝到<CATALINA\_HOME>/common/lib 目录下。
- (2) 把 soap.war 拷贝到<CATALINA\_HOME>/webapps 目录下，启动 Tomcat 服务器。
- (3) 运行 compile.bat，从而编译 HelloService.java 和 HelloClient.java。
- (4) 把编译生成的 HelloService.class 文件拷贝到以下目录：  
<CATALINA\_HOME>/webapps/soap/WEB-INF/classes/mypack
- (5) 运行 test.bat，该批处理脚本依次执行以下任务：
  - 发布、查看 helloservice 服务
  - 运行 HelloClient
  - 删除 helloservice 服务

## 20.7 小 结

SOAP (Simple Object Access Protocol) 即简单对象访问协议，是在分散或分布式的环境中交换信息的简单协议，它以 XML 作为数据传送方式。SOAP 系统有两种工作模式，一种称为 RPC (Remote Procedure Call)，另一种称为 Message-Oriented，本文讨论了 RPC。Apache SOAP 是 Apache 软件组织对 SOAP 规范的实现。Tomcat 充当 Apache-SOAP Web 应用的容器，而 Apache-SOAP Web 应用充当 SOAP 服务的容器，Apache SOAP 客户程序可以通过 Apache SOAP API 来发出 RPC 请求，访问 SOAP 服务。建立 SOAP 应用的环境，有几个值得注意的地方：

- 在服务器方，把 activation.jar、mail.jar 和 xerces.jar 拷贝到<CATALINA\_HOME>/common/lib 目录下
- 发布某个 SOAP 服务时，应该把实现 SOAP 服务的 class 文件拷贝到<CATALINA\_HOME>/webapps/soap/WEB-INF/classes 目录下
- 在 SOAP 客户方，需要把 soap.jar、mail.jar 和 activation.jar 加入到 classpath 中

# 第 21 章 Tomcat 与 Apache AXIS 集成

Apache AXIS 是 Apache SOAP 项目的第三代产品。Apache AXIS 在运行速度、灵活性和稳定性方面都超过了 Apache SOAP。此外，Apache AXIS 还支持 WSDL (Web Service Description Language)。WSDL 是由 IBM 和 Microsoft 制订的 Web 服务描述语言规范，它得到了许多软件组织的认可和支持。

本章介绍如何把 Apache AXIS 集成到 Tomcat 中，以及在 AXIS 中创建 SOAP 服务和 SOAP 客户的方法。

## 21.1 建立 Apache AXIS 环境

建立 Apache AXIS 环境需要的文件的清单、下载地址以及在本书配套光盘上的位置参见表 21-1。

表 21-1 Apache AXIS 所需的文件

文件	下载地址	在本书配套光盘上的位置
Apache AXIS 软件	<a href="http://xml.apache.org/axis/index.html">http://xml.apache.org/axis/index.html</a>	software/axis-1_1.zip
mail.jar	<a href="http://java.sun.com/products/javamail">http://java.sun.com/products/javamail</a>	lib 目录下
activation.jar	<a href="http://java.sun.com/products/javabeans/glasgow/jaf.html">http://java.sun.com/products/javabeans/glasgow/jaf.html</a>	lib 目录下
xerces.jar	<a href="http://xml.apache.org/xerces-j/index.html">http://xml.apache.org/xerces-j/index.html</a>	lib 目录下

把 axis-1\_1.zip 解压到本地硬盘，假定它的根目录为<AXIS\_HOME>，在其 lib 子目录下包含了 AXIS 的所有 JAR 文件，在编译和运行 SOAP 客户程序时需要用到这些 JAR 文件。在<AXIS\_HOME>/webapps 子目录下有一个 axis 子目录，它是一个 Web 应用。

## 21.2 在 Tomcat 上发布 Apache-AXIS Web 应用

运行 Apache AXIS 工程，需要一个 Servlet/JSP 容器，在此，通过 Tomcat 服务器来发布 Apache-AXIS Web 应用，步骤如下。



(1) 把 activation.jar、mail.jar 和 xerces.jar 拷贝到<CATALINA\_HOME>/common/lib 目录下。

(2) 把<AXIS\_HOME>/webapps 目录下的整个 axis 子目录拷贝到<CATALINA\_HOME>/webapps 目录下。



在<CATALINA\_HOME>/webapps/axis 子目录下已经包含了 Apache AXIS 本身的类文件，因此不需要把<AXIS\_HOME>/lib 目录下的 JAR 文件拷贝到<CATALINA\_HOME>/common/lib 目录下。

这样，Apache-AXIS Web 应用就发布成功了。启动 Tomcat 服务器，访问 <http://localhost:8080/axis>，将会看到如图 21-1 所示的网页。

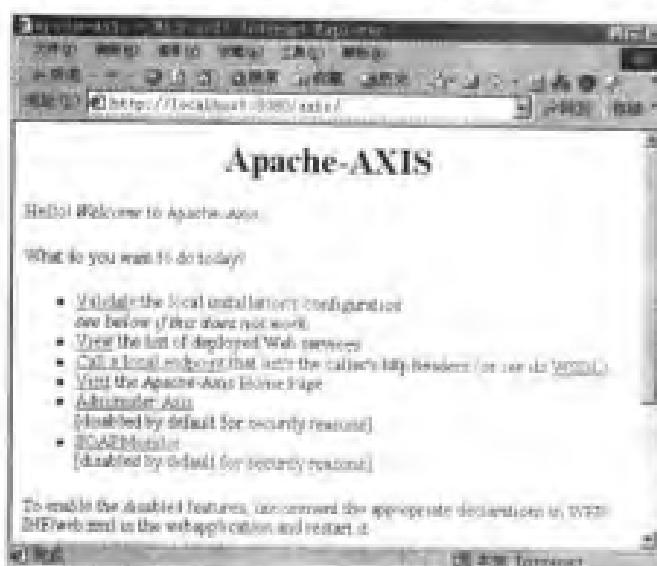


图 21-1 Apache-AXIS 主页

选择“Validate”链接，将运行 happyaxis.jsp。它能够检查 Apache AXIS 的配置是否正确，例如询问是否准备好了必要的 JAR 文件。如果在 happyaxis.jsp 的返回网页上没有汇报错误，那说明配置已经成功，可以忽略警告信息。

### 21.3 创建 SOAP 服务

Tomcat 充当 Apache-AXIS Web 应用的容器，而 Apache-AXIS Web 应用又充当 SOAP 服务的容器，SOAP 客户程序可以通过 Apache AXIS API 来发出 RPC 请求，访问 SOAP 服务，如图 21-2 所示。

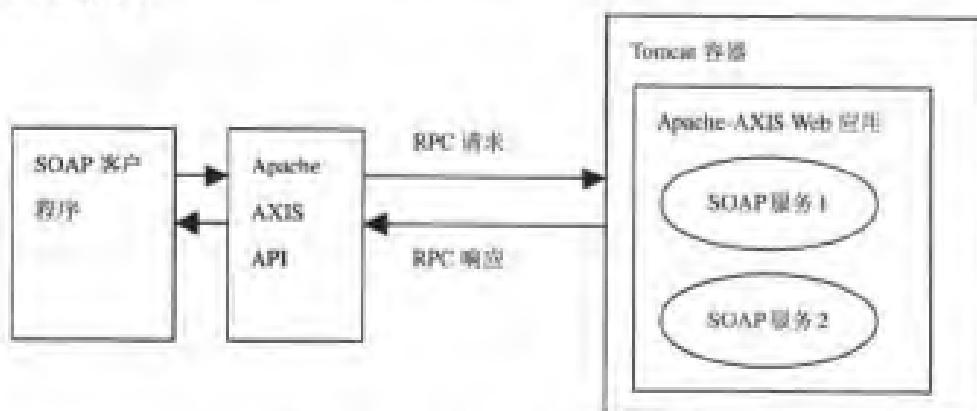


图 21-2 SOAP 客户和 SOAP 服务

创建基于 RPC 的 SOAP 服务包括两个步骤：

- 创建提供 SOAP 服务的 Java 类
- 创建 SOAP 服务的发布描述符文件

### 21.3.1 创建提供 SOAP 服务的 Java 类

以下是一个简单的 SOAP 服务类，它包含了一个方法 sayHello：

```
package mypack;

public class HelloService {
    public String sayHello(String username) {
        return "Hello:" + username;
    }
}
```

编译这个 Java 类不需要在 classpath 中引入任何与 axis 相关的 JAR 文件。编译完毕，应该把 HelloService.class 文件拷贝到以下目录：

```
<TOMCAT_HOME>/webapps/axis/WEB-INF/classes/mypack
```

### 21.3.2 创建 SOAP 服务的发布描述符文件

Apache AXIS 使用 Web 服务发布描述符文件 WSDD (Web Service Deployment Descriptor) 来发布 SOAP 服务。以下是 HelloService 的发布描述文件，名为 deploy.wsdd：

```
<deployment name="test" xmlns="http://xml.apache.org/axis/wsdd/"
           xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <service name="urn:helloservice" provider="java:RPC">
        <parameter name="className" value="mypack.HelloService" />
        <parameter name="allowedMethods" value="sayHello" />
    </service>

</deployment>
```

以上文件配置了<deployment>、<service>和<parameter>元素，下面分别讲述这几种元素。

#### 1. <deployment>元素

<deployment>元素指定了 WSDD 所用的 XML 名字空间。<deployment>元素是其他元素的根元素，在<deployment>元素中可以包含多个<service>元素。

#### 2. <service>元素

<service>元素定义了一项SOAP服务，它有两个属性：name属性代表这项服务的唯一标志符，SOAP客户将根据name属性来访问SOAP服务；Provider属性指定实现这项服务的语言以及服务方式。

### 3. <parameter>元素

<parameter>元素包含 name 和 value 属性，如果 name 属性取值为 className，则指定实现这项服务的类名；如果 name 属性取值为 allowedMethods，则指定这项服务包含的方法。

如果要删除已经发布的 SOAP 服务，则可以使用<undeployment>元素。例如，如果删除 helloservice 服务，则可以创建如下的 undeploy.wsdd 文件：

```
<undeployment name="test" xmlns="http://xml.apache.org/axis/wsdd/">
    <service name="urn:helloservice"/>
</undeployment>
```

## 21.4 管理 SOAP 服务

Apache AXIS 工程提供了两种管理 SOAP 服务的工具，一种是基于 Web 的，还有一种是基于命令行的。这两种工具都能提供发布和删除 SOAP 服务的功能。

下面将介绍基于命令行的 SOAP 服务管理工具，它是由 org.apache.axis.client.AdminClient 类来实现的。



可以在本书配套光盘的 sourcecode/chapter21 目录下的 test.bat 文件中找到以下两小节介绍的发布和删除 helloservice 服务的命令。运行这些命令时，应该把<AXIS\_HOME>/lib 目录下的所有 JAR 文件以及 xerces.jar 文件加入到 classpath 中。

### 21.4.1 发布 SOAP 服务

SOAP 服务时，只要指定 wsdd 文件即可，使用方法如下：

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

以上命令将把 deploy.wsdd 文件中声明的 SOAP 服务发布到 SOAP 服务器中。



发布 HelloService 服务时，应该确保已经把 HelloService.class 拷贝到 <TOMCAT\_HOME>/webapps/axis/WEB-INF/classes/mypack 目录下。

可以通过 Apache AXIS 的基于 Web 的管理工具来查看服务。在如图 21-1 所示的网页上选择“View”链接，将会看到如图 21-3 所示的网页。

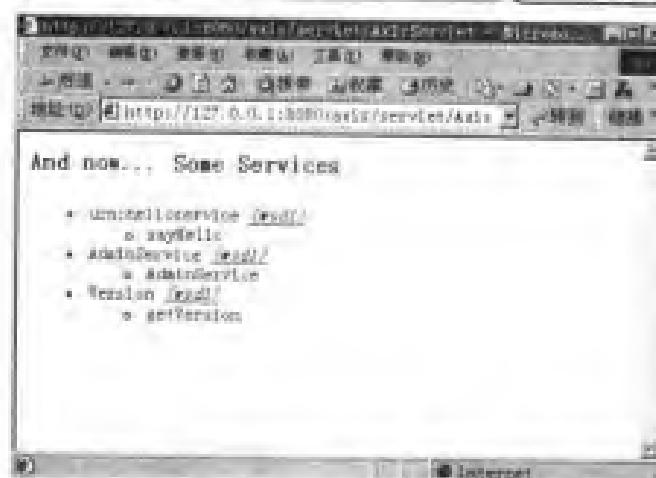


图 21-3 显示 SOAP 服务的网页

选择“urn:helloservice”链接，将会显示描述 helloservice 服务的 WSDL（Web Service Description Language），如图 21-4 所示。

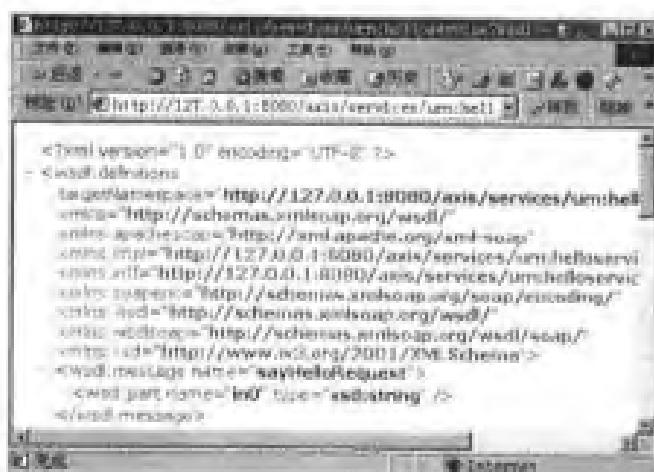


图 21-4 helloservice 的 WSDL

#### 21.4.2 删除 SOAP 服务

删除 SOAP 服务时，只要指定包含<undeployment>元素的 wsdd 文件即可。使用方法如下：

```
java org.apache.axis.client.AdminClient undeploy.wsdd
```

运行以上命令，将删除 helloservice 服务。

### 21.5 创建和运行 SOAP 客户程序

SOAP RPC 客户程序可以通过 Apache AXIS API 发出 RPC 请求，调用 SOAP 服务的方法，例程 21-1 是访问 helloservice 服务的 sayHello 方法的客户程序。

## 例程 21-1 HelloClient.java

```
package mypack;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;

public class HelloClient
{
    public static void main(String[] args) {
        try {
            String username="Guest";
            if(args.length!=0) username=args[0];

            String endpoint = "http://localhost:8080/axis/services/helloservice";

            Service service = new Service();
            Call call = (Call) service.createCall();

            call.setTargetEndpointAddress( new java.net.URL(endpoint) );
            call.setOperationName(new QName("urn:helloservice", "sayHello") );
            String ret = (String) call.invoke( new Object[] { username } );

            System.out.println(ret);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

HelloClient 访问 helloservice 服务包含如下步骤。

 步骤

(1) 读取命令行参数，设置 username 的值：

```
String username="Guest";
if(args.length!=0) username=args[0];
```

(2) 定义 helloservice 服务的位置：

```
String endpoint ="http://localhost:8080/axis/services/helloservice";
```

(3) 创建 Service 实例，然后通过 Service 实例创建 Call 实例。它提供了发出 RPC 请求的功能。在发出 RPC 请求之前，首先应该为 Call 对象设置相关的 RPC 请求信息：

```
Service service = new Service();
Call call = (Call) service.createCall();
call.setTargetEndpointAddress( new java.net.URL(endpoint) );
call.setOperationName(new QName("urn:helloservice", "sayHello") );
```

## (4) 调用 Call 实例的 invoke 方法。

```
String ret = (String) call.invoke( new Object[] { username } );
```

invoke 方法负责向 SOAP 服务器发出客户指定的 RPC 请求，服务器接收到 RPC 请求后，会在服务器端执行相应的服务方法，然后把返回值传给客户。

编译 HelloClient 时，应该把<AXIS\_HOME>/lib 目录下的 JAR 文件以及 xerces.jar 加入到 classpath 中。

运行 HelloClient 时，应该确保 helloservice 服务已经发布，Apache-AXIS Web 应用处于运行状态。此外，运行 HelloClient 时需要把<AXIS\_HOME>/lib 目录下的所有 JAR 文件以及 xerces.jar 加入到 classpath 中，然后在命令行运行如下命令：

```
java mypack.HelloClient weiqin
```

将会看到如下的输出内容：

```
Hello:weiqin
```

为了便于读者运行本章的例子，本书配套光盘的 sourcecode/chapter21 目录下包含了本章例子的源文件，并且提供了编译和运行程序的脚本。

- compile.bat 提供了编译 HelloService.java 和 HelloClient.java 的命令
- test.bat 文件提供了发布、查看和删除 helloservice 服务以及运行 HelloClient 的命令
- 在 lib 子目录下包含了编译和运行客户程序所需的所有 JAR 文件

可以把整个 chapter21 目录拷贝到本地硬盘，再按以下步骤运行本章例子。

## 步骤

(1) 把 activation.jar、mail.jar 和 xerces.jar 拷贝到<CATALINA\_HOME>/common/lib 目录下。

(2) 把整个 axis 子目录拷贝到<CATALINA\_HOME>/webapps 目录下，启动 Tomcat 服务器。

(3) 运行 compile.bat，从而编译 HelloService.java 和 HelloClient.java。

(4) 把编译生成的 HelloService.class 文件拷贝到以下目录：

<CATALINA\_HOME>/webapps/axis/WEB-INF/classes/mypack

(5) 运行 test.bat，该批处理脚本依次执行以下任务：

- 发布、查看 helloservice 服务
- 运行 HelloClient
- 删除 helloservice 服务

AXIS 还允许用户通过浏览器，以 GET 方式访问 SOAP 服务。例如，可以通过以下 URL 直接访问 helloservice 服务：

<http://localhost:8080/axis/services/um:helloservice?method=sayHello&parameter=weiqin>

访问以上 URL，将会看到如图 21-5 所示的网页。

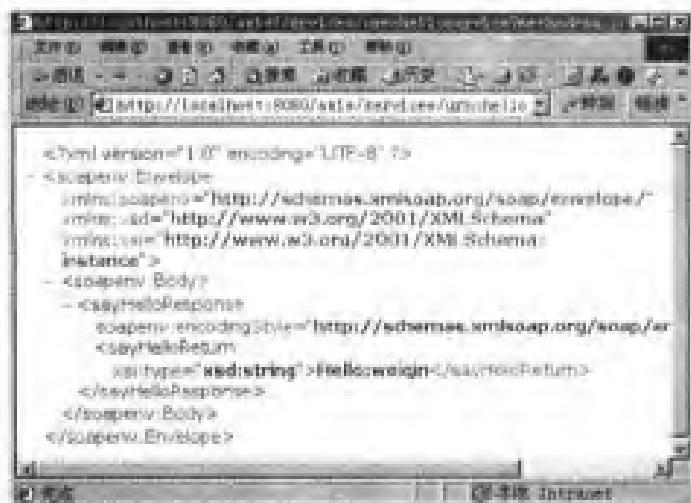


图 21-5 helloservice 服务的响应结果

在上一章已经讲过，SOAP 客户和 SOAP 服务之间传送的是符合 SOAP 规范的 XML 数据。图 21-5 显示了 helloservice 服务向 SOAP 客户返回的 XML 数据。

## 21.6 发布 JWS 服务

所谓 JWS (Java Web Service) 服务，就是 Java Web 服务。AXIS 允许把 Java 源文件的扩展名改为 jws，然后把它拷贝到<CATALINA\_HOME>/webapps/axis 目录下，这样 AXIS 会自动编译 JWS 文件，并把它加入到 JWS 服务中。

如果要把 HelloService 作为 JWS 服务来发布，只要将 HelloService.java 改名为 HelloService.jws，再把 HelloService.jws 拷贝到以下位置：

<CATALINA\_HOME>/webapps/axis>HelloService.jws

然后访问以下 URL：

<http://localhost:8080/axis>HelloService.jws?method=sayHello&parameter=weiqin>

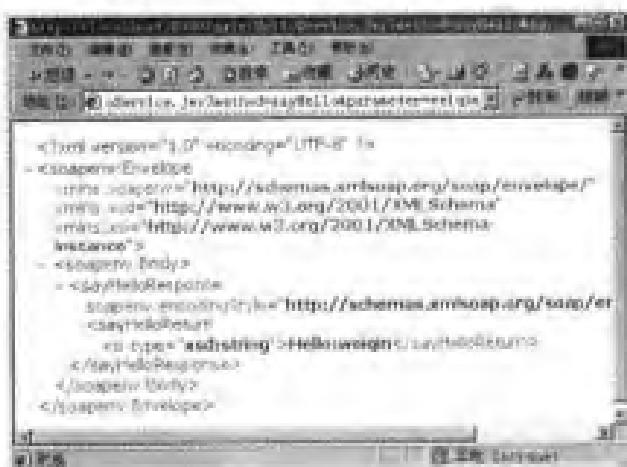


图 21-6 helloservice 服务的响应结果

## 21.7 小结

Apache AXIS 是 Apache SOAP 项目的第三代产品。Tomcat 充当 Apache-AXIS Web 应用的容器，而 Apache-AXIS Web 应用充当 SOAP 服务的容器，Apache AXIS 客户程序可以通过 Apache AXIS API 来发出 RPC 请求，访问 SOAP 服务。建立 Apache AXIS 应用的环境，有下面几个值得注意的地方：

- 在服务器方，把 activation.jar, mail.jar 和 xerces.jar 拷贝到<CATALINA\_HOME>/common/lib 目录下
- 发布某个 SOAP 服务时，应该把实现 SOAP 服务的 CLASS 文件拷贝到 <CATALINA\_HOME>/webapps/axis/WEB-INF/classes 目录下
- 在 SOAP 客户方，需要把<AXIS\_HOME>/lib 目录下的所有 JAR 文件以及 xerces.jar 加入到 classpath

# 第 22 章 Tomcat 与其他 HTTP 服务器集成

Tomcat 最主要的功能是提供 Servlet/JSP 容器，尽管它也可以作为独立的 Java Web 服务器，它在对静态资源（如 HTML 文件或图像文件）的处理速度，以及提供的 Web 服务器管理功能方面都不如其他专业的 HTTP 服务器，如 IIS 和 Apache 服务器。

因此在实际应用中，常常把 Tomcat 与其他 HTTP 服务器集成。对于不支持 Servlet/JSP 的 HTTP 服务器，可以通过 Tomcat 服务器来运行 Servlet/JSP 组件。

当 Tomcat 与其他 HTTP 服务器集成时，Tomcat 服务器的工作模式通常为进程外的 Servlet 容器，Tomcat 服务器与其他 HTTP 服务器之间通过专门的插件来通信。关于 Tomcat 服务器的工作模式的概念可以参考本书 1.4 节。

本章首先讨论 Tomcat 与 HTTP 服务器集成的一般原理，然后介绍 Tomcat 与 Apache 以及 IIS 集成的详细步骤。

## 22.1 Tomcat 与 HTTP 服务器集成的原理

Tomcat 服务器通过 Connector 连接器组件与客户程序建立连接，Connector 组件负责接收客户的请求，以及把 Tomcat 服务器的响应结果发送给客户。默认情况下，Tomcat 在 server.xml 中配置了两种连接器：

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector port="8080"
           maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
           enableLookups="false" redirectPort="8443" acceptCount="100"
           debug="0" connectionTimeout="20000"
           disableUploadTimeout="true" />

<!-- Define a Coyote/JK2 AJP 1.3 Connector on port 8009 -->
<Connector port="8009"
           enableLookups="false" redirectPort="8443" debug="0"
           protocol="AJP/1.3" />
```

第一个连接器监听 8080 端口，负责建立 HTTP 连接。在通过浏览器访问 Tomcat 服务器的 Web 应用时，使用的就是这个连接器。

第二个连接器监听 8009 端口，负责和其他的 HTTP 服务器建立连接。在把 Tomcat 与其他 HTTP 服务器集成时，就需要用到这个连接器。

Web 客户访问 Tomcat 服务器上 JSP 组件的两种方式如图 22-1 所示。

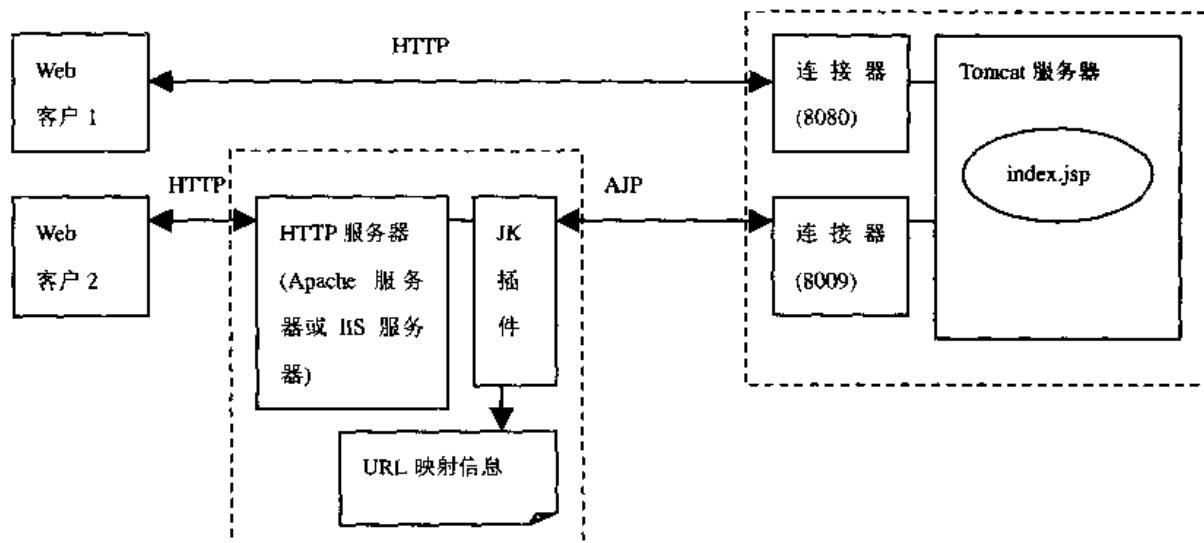


图 22-1 Web 客户访问 Tomcat 服务器上的 JSP 组件的两种方式

在图 22-1 中, Web 客户 1 直接访问 Tomcat 服务器上的 JSP 组件, 他访问的 URL 为 `http://localhost:8080/index.jsp`。Web 客户 2 通过 HTTP 服务器访问 Tomcat 服务器上的 JSP 组件。假定 HTTP 服务器使用的 HTTP 端口为默认的 80 端口, 那么 Web 客户 2 访问的 URL 为 `http://localhost:80/index.jsp` 或者 `http://localhost/index.jsp`。

下面, 介绍 Tomcat 与 HTTP 服务器之间是如何通信的。

### 22.1.1 JK 插件

Tomcat 提供了专门的 JK 插件来负责 Tomcat 和 HTTP 服务器的通信。应该把 JK 插件安置在对方的 HTTP 服务器上。当 HTTP 服务器接收到客户请求时, 它会通过 JK 插件来过滤 URL, JK 插件根据预先配置好的 URL 映射信息, 决定是否要把客户请求转发给 Tomcat 服务器处理。

假定在预先配置好的 URL 映射信息中, 所有 “`/*.jsp`” 形式的 URL 都由 Tomcat 服务器来处理, 那么在图 22-1 的例子中, JK 插件将把客户请求转发给 Tomcat 服务器, Tomcat 服务器于是运行 `index.jsp`, 然后把响应结果传给 HTTP 服务器, HTTP 服务器再把响应结果传给 Web 客户 2。

对于不同的 HTTP 服务器, Tomcat 提供了不同的 JK 插件的实现模块。本章将用到以下 JK 插件:

- 与 Windows 下的 Apache HTTP 服务器集成: `mod_jk_2.0.46.dll`
- 与 Linux(RedHat)下的 Apache HTTP 服务器集成: `mod_jk.so-ap2.0.46-rh72..46-rh72`
- 与 IIS 服务器集成: `isapi_redirect.dll`

### 22.1.2 AJP 协议

AJP 是为 Tomcat 与 HTTP 服务器之间通信而定制的协议, 能提供较高的通信速度和效

率。在配置 Tomcat 与 HTTP 服务器集成中，读者可以不必关心 AJP 协议的细节。关于 AJP 的知识也可以参考网址：

<http://jakarta.apache.org/builds/jakarta-tomcat-connectors/jk2/doc/common/AJPv13.html>

## 22.2 在 Windows 下 Tomcat 与 Apache 服务器集成

Apache HTTP 服务器是 Apache 软件组织提供的开放源代码软件，它是一个非常优秀的专业的 Web 服务器，为网络管理员提供了丰富多彩的 Web 管理功能，包括目录索引、目录别名、内容协商、可配置的 HTTP 错误报告、CGI 程序的 SetUID 执行、子进程资源管理、服务器端图像映射、重写 URL、URL 拼写检查以及联机手册等。

Apache HTTP 服务器本身没有提供 Servlet/JSP 容器。因此，在实际应用中，把 Tomcat 与 Apache 集成，可以建立具有实用价值的商业化的 Web 平台。

在 Windows NT/2000 下 Tomcat 与 Apache 服务器集成需要准备的软件参见表 22-1。

表 22-1 在 Windows NT/2000 下 Tomcat 与 Apache 服务器集成需要准备的软件

软 件	下 载 位 置	本 书 配 套 光 盘 上 的 位 置
基于 Windows NT/2000 的 Apache HTTP 服务器软件	<a href="http://httpd.apache.org/download.cgi">http://httpd.apache.org/download.cgi</a>	software/ apache_2.0.47-win32-x86-no_ssl.msi
JK 插件	<a href="http://jakarta.apache.org/builds/jakarta-tomcat-connectors/jk">http://jakarta.apache.org/builds/jakarta-tomcat-connectors/jk</a>	lib/mod_jk_2.0.46.dll

### 1. 安装 Apache HTTP 服务器

运行 apache\_2.0.47-win32-x86-no\_ssl.msi，就启动了 Apache HTTP 服务器的安装程序，只要按默认设置进行安装即可。如果安装成功，会自动在 Windows 中加入 Apache HTTP 服务，如图 22-2 所示。



图 22-2 加入到 Windows 服务中的 Apache 服务

假定 Apache 的根目录为<APACHE\_HOME>, 在其 conf 子目录下有一个配置文件 httpd.conf, 如果 Apache 安装在本机, 并且采用默认的 80 端口作为 HTTP 端口, 在 httpd.conf 文件中会看到如下属性:

```
Listen 80  
ServerName localhost:80
```

在操作系统的【开始】→【程序】→【Apache HTTP Server 2.0.47】→【Control Apache Server】菜单中, 提供了重启(Restart)、启动(Start)和关闭(Stop) Apache 服务器的子菜单。



应该确保 80 端口没有被占用, 否则 Apache 服务器无法启动。

Apache 服务器启动后, 就可以通过访问 Apache 的测试页来确定是否安装成功。访问 http://localhost, 如果出现如图 22-3 所示的网页, 就说明 Apache 已经安装成功了。

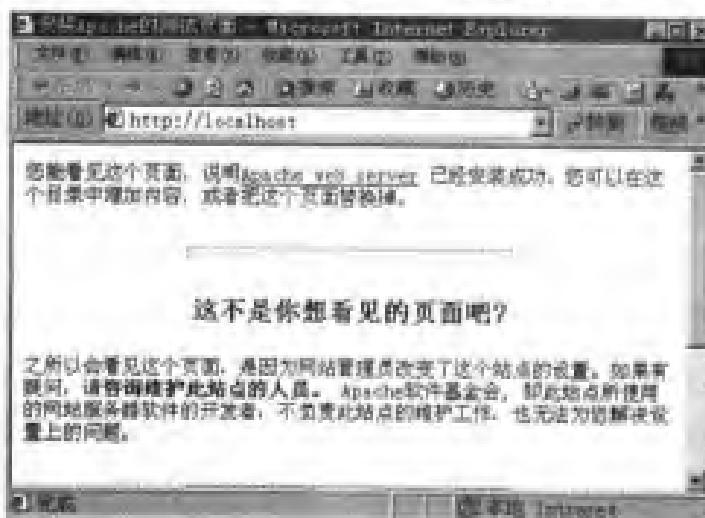


图 22-3 Apache 服务器的测试网页

## 2. 在 Apache 中加入 JK 插件

在 Apache 中加入 JK 插件, 只要把 mod\_jk\_2.0.46.dll 拷贝到<APACHE\_HOME>/modules 目录下即可。

## 3. 创建 workers.properties 文件

workers.properties 文件用于配置 Tomcat 的信息, 它的存放位置为<APACHE\_HOME>/conf/workers.properties。在本书配套光盘的 sourcecode/chapter22/windows\_apache 目录下提供了 workers.properties 文件, 它的内容如下 (“#” 后面为注释信息):

```
workers.tomcat_home=C:\jakarta-tomcat #让 mod_jk 模块知道 Tomcat  
workers.java_home=C:\j2sdk1.4.2 #让 mod_jk 模块知道 j2sdk  
ps=\ #指定文件路径分割符  
worker.list=worker1  
worker.worker1.port=8009 #工作端口, 若没占用则不用修改  
worker.worker1.host=localhost #Tomcat 服务器的地址
```

```
worker.worker1.type=ajp13 #类型
worker.worker1.lbfactor=1 #负载平衡因数
```

以上文件中的属性描述参见表 22-2。

表 22-2 workers.properties 文件的属性

属性	描述
workers.tomcat_home	指定 Tomcat 服务器的根目录
workers.java_home	指定 JDK 的根目录
worker.list	指定 Tomcat 服务器工作名单
worker.worker1.port	指定 Tomcat 服务器使用的 JK 端口
worker.worker1.host	指定 Tomcat 服务器的 IP 地址
worker.worker1.type	指定 Tomcat 服务器与 Apache 之间的通信协议
worker.worker1.lbfactor	指定负载平衡因数 (Load Balance Factor)。只有在使用了负载平衡器 (Load Balancer) 的情况下，这个属性才有意义

#### 4. 修改 Apache 的配置文件 httpd.conf

打开<APACHE\_HOME>/conf/httpd.conf 文件，在其末尾加入以下内容：

```
# Using mod_jk2.dll to redirect dynamic calls to Tomcat
LoadModule jk_module modules/mod_jk_2.0.46.dll
JkWorkersFile "conf/workers.properties"
JkLogFile "logs/mod_jk2.log"
JkLogLevel debug
JkMount /*.jsp worker1
JkMount /helloapp/* worker1
```

在本书配套光盘的 sourcecode/chapter22/windows\_apache/httpd\_modify.conf 文件中提供了以上内容，它指示 Apache 服务器加载 JK 插件，并且为 JK 插件设置相关属性，这些属性的描述参见表 22-3。

表 22-3 JK 插件的相关属性

属性	描述
LoadModule	指定加载的 JK 插件
JkWorkersFile	指定 JK 插件的工作文件
JkLogFile	指定 JK 插件使用的日志文件，在实际配置中，可以通过查看这个日志文件，来跟踪 JK 插件的运行过程，这对排错很有用
JkLogLevel	指定 JK 插件的日志级别，可选值包括 debug、info 和 error 等
JkMount	指定 JK 插件处理的 URL 映射信息

JkMount 用来指定 URL 映射信息，“JkMount /\*.jsp worker1”表示“\*.jsp”形式的 URL 都由 worker1 代表的 Tomcat 服务器来处理；“JkMount /helloapp/\* worker1”表示访问 helloapp 应用的 URL 都由 worker1 来处理。

#### 5. 测试配置

重启 Tomcat 服务器和 Apache 服务器，通过浏览器访问 <http://localhost/index.jsp>，如果

出现 Tomcat 的默认主页，说明配置已经成功。此外，如果在 Tomcat 服务器上已经发布了 helloapp 应用，可以访问 <http://localhost/helloapp/index.htm>，如果正常返回 helloapp 应用的 index.htm 网页，说明配置已经成功。如果配置有误，可以查看 JK 插件生成的日志信息，它有助于查找错误原因。在 Apache 的配置文件 httpd.conf 中设定该日志文件的存放位置为 <APACHE\_HOME>/logs/mod\_jk2.log

## 6. Apache 与多个 Tomcat 服务器集成时的负载平衡

在实际应用中，如果网站的访问量非常大，为了提高访问速度，可以将多个 Tomcat 服务器与 Apache 集成，让它们共同分担运行 Servlet/JSP 组件的任务。JK 插件的 loadbalancer（负载平衡器）负责根据在 workers.properties 文件中预先配置的 lbfactor（负载平衡因数）为这些 Tomcat 服务器分配工作负荷，实现负载平衡。

假定 Apache 和两个 Tomcat 服务器集成，一个 Tomcat 服务器和 Apache 运行在同一台机器上，使用的 JK 端口为 8009，还有一个 Tomcat 服务器运行在另一台机器上，主机名为 anotherhost，使用的 JK 端口为 8009。

以下是把 Apache 和这两个 Tomcat 服务器集成的步骤。

### 步骤

(1) 把 mod\_jk\_2.0.46.dll 拷贝到<APACHE\_HOME>/lib 目录下。

(2) 在<APACHE\_HOME>/conf 目录下创建如下的 workers.properties 文件（注意粗体部分的内容）：

```
ps=\ #指定文件路径分割符  
worker.list=worker1,worker2,loadbalancer  
  
worker.worker1.port=8009 #工作端口，若没占用则不用修改  
worker.worker1.host=localhost #Tomcat 服务器的地址  
worker.worker1.type=ajp13 #类型  
worker.worker1.lbfactor=100 #负载平衡因数  
  
worker.worker2.port=8009 #工作端口，若没占用则不用修改  
worker.worker2.host=anotherhost #Tomcat 服务器的地址  
worker.worker2.type=ajp13 #类型  
worker.worker2.lbfactor=100 #负载平衡因数  
  
worker.loadbalancer.type=lb  
worker.loadbalancer.balanced_workers=worker1, worker2
```

以上文件创建了两个 worker： worker1 和 worker2 分别代表两个 Tomcat 服务器，它们由 worker.loadbalancer 来分配工作负荷。

(3) 修改<APACHE\_HOME>/conf/httpd.conf 文件，在文件末尾加入如下内容：

```
# Using mod_jk2.dll to redirect dynamic calls to Tomcat  
LoadModule jk_module modules/mod_jk_2.0.46.dll  
JkWorkersFile "conf/workers.properties"  
JkLogFile "logs/mod_jk2.log"  
JkLogLevel debug
```

```
JkMount /*.jsp loadbalancer
JkMount /helloapp/* loadbalancer
```

当客户请求“\*.jsp”或“/helloapp/\*”形式的 URL，该请求都由 loadbalancer 来负责转发，它根据在 workers.properties 文件中为 worker1 和 worker2 分配的 lbfactor 属性，来决定如何调度它们。

**提示**

只有在使用了 loadbalancer 的情况下，workers.properties 文件中 worker 的 lbfactor 属性才有意义，lbfactor 取值越大，表示分配给 Tomcat 服务器的工作负荷越大。

(4) 修改两个 Tomcat 服务器的 JK 端口，确保它们和 workers.properties 文件中的配置对应。此外，在使用了 loadbalancer 后，要求 worker 的名字和 Tomcat 的 server.xml 文件中的<Engine>元素的 jvmRoute 属性一致。

所以应该分别修改两个 Tomcat 的 sever.xml 文件，把它们的<Engine>元素的 jvmRoute 属性分别设为 worker1 和 worker2。以下是修改后的两个 Tomcat 服务器的<Engine>元素：

Tomcat 服务器 1：

```
<Engine name="Catalina" defaultHost="localhost" debug="0" jvmRoute="worker1">
```

Tomcat 服务器 2：

```
<Engine name="Catalina" defaultHost="localhost" debug="0" jvmRoute="worker2">
```

(5) 在完成以上步骤后，分别启动两个 Tomcat 服务器和 Apache 服务器，然后访问 <http://localhost/index.jsp>，会出现 Tomcat 服务器的默认主页，由于此时由 loadbalancer 来调度 Tomcat 服务器，因此不能断定到底访问的是哪个 Tomcat 服务器的 index.jsp，这对于 Web 客户来说是透明的。

如果在进行以上实验时，两个 Tomcat 服务器都在同一台机器上运行，应该确保它们没有使用相同的端口。在 Tomcat 的默认的 server.xml 中，一共配置了以下 3 个端口：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector port="8080" />
<!-- Define a Coyote/JK2 AJP 1.3 Connector on port 8009 -->
<Connector port="8009" />
```

如果两个 Tomcat 服务器都在同一台机器上运行，则至少应该对其中一个 Tomcat 服务器的以上 3 个端口号都进行修改。

此外，如果把 Tomcat 和其他 HTTP 服务器集成，Tomcat 主要负责处理 HTTP 服务器转发过来的客户请求，通常不会直接接受 HTTP 请求。因此为了提高 Tomcat 的运行性能，可以关闭 Tomcat 的 HTTP 连接器，方法为在 server.xml 中把 Tomcat 的 HTTP Connector 的配置注释掉。

## 22.3 在 Linux 下 Tomcat 与 Apache 服务器集成

在 Linux 下 Tomcat 与 Apache 服务器集成的步骤与在 Windows NT/2000 下非常相似。

在 Linux 下 Tomcat 与 Apache 服务器集成需要准备的软件参见表 22-4。

表 22-4 在 Linux 下 Tomcat 与 Apache 服务器集成需要准备的软件

软件	下载位置	本书配套光盘上的位置
基于 Linux 的 Apache HTTP 服务器软件	<a href="http://httpd.apache.org/download.cgi">http://httpd.apache.org/download.cgi</a>	software/httpd-2.0.47.tar.gz
JK 插件	<a href="http://jakarta.apache.org/builds/jakarta-tomcat-connectors/jk">http://jakarta.apache.org/builds/jakarta-tomcat-connectors/jk</a>	lib/mod_jk.so-ap2.0.46-rh72..46-rh72

本书选用的 Linux 是 RedHat，如果安装的是其他类型的 Linux，可以到表 22-4 列出的地址下载相应的 JK 插件。

### 1. 安装 Apache HTTP 服务器

以下是在 Linux 下安装 Apache 服务器的步骤。

#### 步骤

(1) 建立 httpd 用户，把 httpd-2.0.47.tar.gz 文件拷贝到/tmp 目录下。

(2) 将 httpd-2.0.47.tar.gz 文件解压，命令为：

```
gzip -d httpd-2.0.47.tar.gz  
tar xvf httpd-2.0.47.tar
```

(3) 用超级用户账号登录 Linux，命令为：su

(4) 转到/tmp/httpd-2.0.47 目录，配置 Apache，命令为：

```
./configure --prefix=/home/httpd
```

“--prefix”选项用来设定 Apache 的安装目录。根据以上设置，Apache 将被安装到 /home/httpd 目录。

(5) 编译 Apache，命令为：make。

(6) 安装 Apache，命令为：make install。

(7) 安装好以后，假定 Apache 的根目录为<APACHE\_HOME>，打开<APACHE\_HOME>/conf/httpd.conf 文件，配置“Listen”和“ServerName”属性：

```
Listen 80  
ServerName localhost
```

(8) 转到<APACHE\_HOME>/bin 目录，运行 apachectl configtest 命令，来测试安装是否成功。如果显示 Syntax ok，则表示安装成功。

启动 Apache 服务器的命令为：<APACHE\_HOME>/bin/apachectl start。

终止 Apache 服务器的命令为：<APACHE\_HOME>/bin/apachectl stop。

#### 提示

应该确保 80 端口没有被占用，否则 Apache 服务器无法启动。

也可以通过访问 Apache 的测试页来确定是否安装成功。访问 <http://localhost>，如果出现如图 22-3 所示的网页，就说明 Apache 已经安装成功了。

### 2. 在 Apache 中加入 JK 插件

在 Apache 中加入 JK 插件，只要把 mod\_jk.so-ap2.0.46-rh72..46-rh72 拷贝到

<APACHE\_HOME>/libexec 目录下即可。

### 3. 创建 workers.properties 文件

在<APACHE\_HOME>/conf 目录下创建以下 workers.properties 文件。此外，在本书配套光盘的 sourcecode/chapter22/linux\_apache 目录下也提供了该文件：

```
ps=/ #指定文件路径分割符
worker.list=worker1
worker.worker1.port=8009 #工作端口,若没占用则不用修改
worker.worker1.host=localhost #Tomcat 服务器的地址
worker.worker1.type=ajp13 #类型
worker.worker1.lbfactor=1 #负载平衡因数
```

### 4. 修改 Apache 的配置文件 httpd.conf

打开<APACHE\_HOME>/conf/httpd.conf 文件，在其末尾加入以下内容：

```
LoadModule jk_module libexec/mod_jk.so-ap2.0.46-rh72..46-rh72
JkWorkersFile "conf/workers.properties"
JkLogFile "logs/mod_jk2.log"
JkLogLevel debug
JkMount /*.jsp worker1
JkMount /helloapp/* worker1
```

在本书配套光盘的 sourcecode/chapter22/linux\_apache 目录下的 httpd\_modify.conf 文件中提供了以上内容。

### 5. 测试配置

重启 Tomcat 服务器和 Apache 服务器。通过浏览器访问 <http://localhost/index.jsp>，如果出现 Tomcat 的默认主页，说明配置已经成功。此外，如果在 Tomcat 服务器上已经发布了 helloapp 应用，则可以访问 <http://localhost/helloapp/index.htm>，如果正常返回 helloapp 应用的 index.htm 网页，说明配置已经成功。如果配置有误，可以查看 JK 插件生成的日志信息，它有助于查找错误原因。在 Apache 的配置文件 httpd.conf 中设定该日志文件的存放位置为：<APACHE\_HOME>/logs/mod\_jk2.log。

## 22.4 Tomcat 与 IIS 服务器集成

IIS (Internet Information Service) 服务器是微软开发的功能强大的 Web 服务器，IIS 为创建和开发电子商务的提供了安全的 Web 平台。把 Tomcat 与 IIS 集成，可以扩展 IIS 的功能，使它支持 Java Web 应用。

### 22.4.1 准备相关文件

在开始本节的操作之前，假定在机器上安装了 IIS 服务器，应该准备好以下 3 个文件。

#### 1. JK 插件

在本书配套光盘的 lib 目录下提供了用于 IIS 的 JK 插件：isapi\_redirect.dll，此外，也

可以到以下地址下载最新的 JK 插件：

<http://jakarta.apache.org/builds/jakarta-tomcat-connectors/jk>

可以把 JK 插件 isapi\_redirect.dll 拷贝到<CATALINA\_HOME>/bin 目录下。

### 2. workers.properties 文件

在<CATALINA\_HOME>/conf 目录下创建如下的 workers.properties 文件。在本书配套光盘的 sourcecode/chapter22/iis 目录下也提供了该文件：

```
workers.tomcat_home=C:\jakarta-tomcat #让 mod_jk 模块知道 Tomcat  
workers.java_home=C:\j2sdk1.4.2 #让 mod_jk 模块知道 j2sdk  
ps=\ #指定文件路径分割符  
worker.list=worker1  
worker.worker1.port=8009 #工作端口,若没占用则不用修改  
worker.worker1.host=localhost #Tomcat 服务器的地址  
worker.worker1.type=ajp13 #类型  
worker.worker1.lbfactor=1 #负载平衡因数
```

### 3. uriworkermap.properties 文件

在<CATALINA\_HOME>/conf 目录下创建如下的 uriworkermap.properties 文件，它为 JK 插件指定 URL 映射。在本书配套光盘的 sourcecode/chapter22/iis 目录下也提供了该文件：

```
/*.jsp=worker1  
/helloapp/*=worker1
```



尽管把以上 3 个文件都放在 Tomcat 目录下，其实 Tomcat 服务器并不会访问这些文件。以上给出的是按照惯例的一种配置，事实上，也可以把这些文件放在文件系统的其他地方。

#### 22.4.2 编辑注册表

在配置 Apache 和 Tomcat 集成时，JK 插件的属性是在 Apache 的配置文件 httpd.conf 中设置的。配置 IIS 和 Tomcat 集成时，应该在操作系统的注册表中设置 JK 插件的属性，以下是操作步骤。



(1) 在 Windows NT/2000 中通过 regedit 命令编辑注册表，创建一个新的键：HKEY\_LOCAL\_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi Redirector\1.0，如图 22-4 所示。

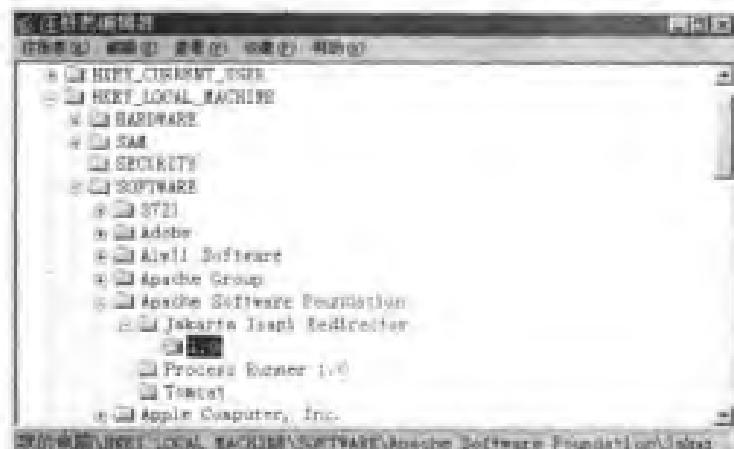


图 22-4 在注册表中创建 Jakarta Isapi Redirector\1.0 键

(2) 在 Jakarta Isapi Redirector\1.0 键下面创建新的字符串，参见表 22-5。创建好之后的注册表如图 22-5 所示。

表 22-5 在 Jakarta Isapi Redirector\1.0 键下面创建的字符串

字符串	字符串值	描述
extension_url	/jakarta/isapi_redirect.dll	指定访问 isapi_redirect.dll 文件的 uri。在 IIS 中将创建名为 jakarta 的虚拟目录，在该目录下包含 isapi_redirect.dll 文件，参见 22.4.3 节
log_file	C:\jakarta-tomcat\logs\isapi.log	指定 JK 模件使用的日志文件，在实际配置中，可以通过查看这个日志文件，来跟踪 JK 模件的运行过程，这对排错很有用
log_level	debug	指定 JK 模件的日志级别，可选值包括 debug、info 和 error 等
worker_file	C:\jakarta-tomcat\conf\workers.properties	指定 JK 模件的工作文件
worker_mount_file	C:\jakarta-tomcat\conf\map\workermap.properties	指定 JK 模件的 URL 映射文件



图 22-5 在 Jakarta Isapi Redirector\1.0 键下面创建新的字符串

在本书配套光盘的 sourcecode/chapter22/iis 目录下提供了注册表编辑文件 jk.reg，如果不按照以上方式手工修改注册表，也可以直接运行 jk.reg 文件（选中这个文件再双击鼠标即可），它会把以上配置内容自动添加到注册表中。jk.reg 的内容如下：

Windows Registry Editor Version 5.00

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi Redirector
V.0]
"extension_uri"="/jakarta/isapi_redirect.dll"
"log_file"="C:\jakarta-tomcat\logs\isapi.log"
"log_level"="debug"
"worker_file"="C:\jakarta-tomcat\conf\workers.properties"
"worker_mount_file"="C:\jakarta-tomcat\conf\uriworkermap.properties"
```

在运行 jk.reg 文件之前，应该把文件中的“C:\jakarta-tomcat”目录替换为 Tomcat 安装目录。

### 22.4.3 在 IIS 中加入“jakarta”虚拟目录

注册表修改以后，应该在 IIS 中加入名为“jakarta”的虚拟目录，它是 JK 插件所在的目录，以下是操作步骤。



(1) 选择操作系统的【控制面板】→【管理工具】→【Internet 服务管理器】选项，打开 Internet 信息服务管理器，如图 22-6 所示。

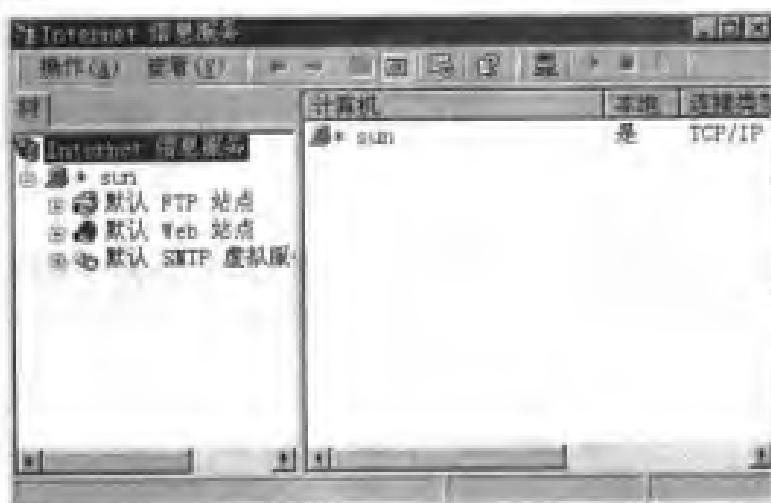


图 22-6 Internet 信息服务管理器窗口

(2) 选中【默认 Web 站点】，单击鼠标右键，在下拉菜单中选择【新建】→【虚拟目录】选项，如图 22-7 所示。创建一个虚拟目录，名为“jakarta”，对应的实际文件资源路径应该是 isapi\_redirect.dll 文件所在的目录<CATALINA\_HOME>/bin。



图 22-7 创建虚拟目录

(3) 修改刚刚创建的 jakarta 虚拟目录的属性，将其执行许可权限设为“脚本和可执行程序”，如图 22-8 所示。这步操作很重要，它保证在注册表中设置的 extension\_uri 对应的 /jakarta/isapi\_redirect.dll 可以被执行。如果漏掉这步操作，会导致无法访问 Tomcat 中的 Servlet/JSP 组件。



图 22-8 修改 jakarta 虚拟目录的执行许可权限

#### 22.4.4 把 JK 插件作为筛选器加入到 IIS

在 IIS 中加入名为“jakarta”的虚拟目录后，还应该把 JK 插件作为 ISAPI 筛选器加入到 IIS 中，以下是操作步骤。

## 步骤

(1) 在 Internet 信息服务主窗口的目录树中选择 IIS 主机节点，单击鼠标右键，在下拉菜单中选择【属性】选项，如图 22-9 所示。在出现的窗口中单击【编辑】按钮，打开 IIS 主机的属性窗口，如图 22-10 所示。

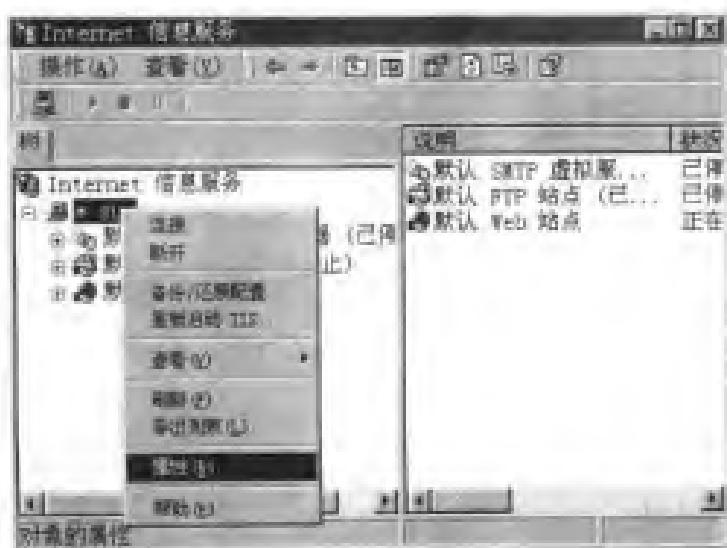


图 22-9 配置 IIS 主机的属性

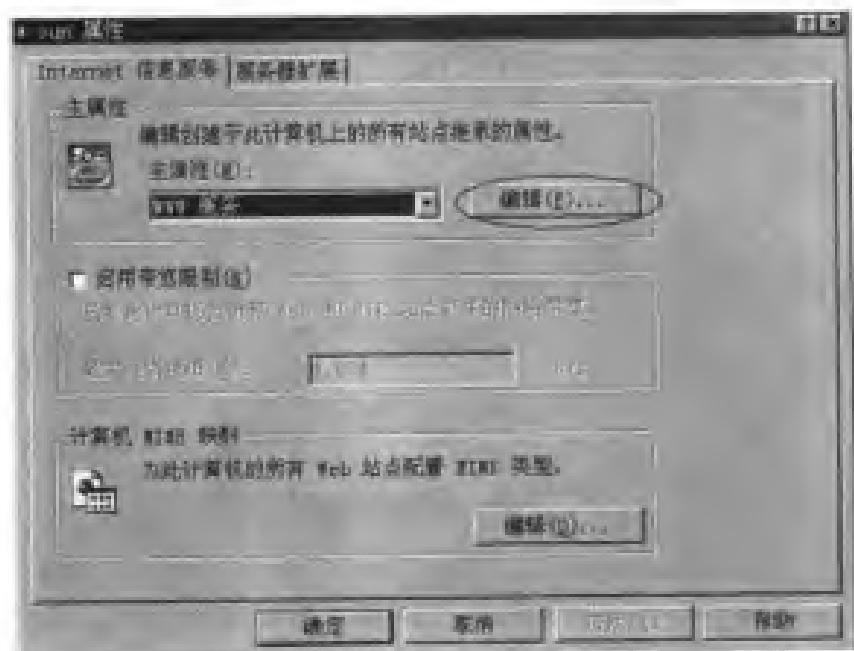


图 22-10 IIS 主机的属性窗口

(2) 在主属性的 WWW 服务区域单击【编辑】按钮，打开 WWW 服务主属性窗口，增加新的 ISAPI 筛选器，筛选器名称为“jakarta”，可执行文件为<CATALINA\_HOME>/bin/isapi\_redirect.dll，如图 22-11 所示。



图 22-11 增加新的 ISAPI 筛选器

(3) 重新启动 IIS 服务器，如果配置正常，在 WWW 服务主属性的 ISAPI 筛选器子窗口中，新加的 jakarta 筛选器的状态应该变为绿色向上的箭头，如图 22-12 所示。



图 22-12 jakarta 筛选器被装载

#### 22.4.5 测试配置

重启 Tomcat 服务器和 IIS 服务器，通过浏览器访问 `http://localhost/index.jsp`。如果出现 Tomcat 的默认主页，说明配置已经成功。此外，如果在 Tomcat 服务器上已经发布了 helloapp 应用，可以访问 `http://localhost/helloapp/index.htm`；如果正常返回 helloapp 应用的 `index.htm` 网页，说明配置已经成功；如果配置有误，可以查看 JK 插件生成的日志信息，它有助于查找错误原因。在注册表中设定该日志文件的存放位置为 `<CATALINA_HOME>/logs/isapi.log`。

## 22.5 小结

本章介绍了通过 JK 插件来实现 Tomcat 与 Apache 以及 IIS 服务器集成的步骤。Tomcat 提供了专门的 JK 插件来负责 Tomcat 和 HTTP 服务器的通信。JK 插件安置在对方 HTTP 服务器上。当 HTTP 服务器接收到客户请求时，它会通过 JK 插件来过滤 URL，JK 插件根据预先配置好的 URL 映射信息，来决定是否要把客户请求转发给 Tomcat 服务器处理。Tomcat 与 Apache 以及 IIS 服务器集成的异同之处参见表 22-6。

表 22-6 Tomcat 与 Apache 以及 IIS 服务器集成的异同之处

	Tomcat 与 Apache 集成	Tomcat 与 IIS 集成
JK 插件的工作文件	<code>workers.properties</code> 文件	<code>workers.properties</code> 文件
设置 JK 插件属性	在 Apache 的配置文件 <code>httpd.conf</code> 中设置	在注册表中设置
设置 URL 映射信息	在 Apache 的配置文件 <code>httpd.conf</code> 中设置	在 <code>uriworkermap.properties</code> 文件中设置
加载 JK 插件	把 JK 插件拷贝到 <code>&lt;APACHE_HOME&gt;/lib</code> 目录下，在 Apache 的配置文件 <code>httpd.conf</code> 中设置 <code>LoadModule</code> 属性	把 JK 插件所在的目录作为 IIS 的虚拟目录，把 JK 插件作为 ISAPI 筛选器加入到 IIS 中

# 第 23 章 创建嵌入式 Tomcat 服务器

本章介绍如何把 Tomcat 嵌入到 Java 应用程序中，在程序中配置 Tomcat 的组件，并控制 Tomcat 服务器的启动和关闭。在这种情况下，Tomcat 服务器将与 Java 应用程序运行在同一个进程中，Tomcat 服务器的工作模式为进程内的 Servlet 容器，关于 Tomcat 服务器的工作模式的概念可以参考本书 1.4 节（Tomcat 的工作模式）。把 Tomcat 服务器作为进程内的 Servlet 容器来运行，可以使 Java 应用程序更灵活地控制 Servlet 容器，而且 Servlet 容器和 Java 应用程序可以共享内存数据。

## 23.1 将 Tomcat 嵌入 Java 应用

在本书第 1 章中，曾讲过 Tomcat 是由一系列嵌套的组件组成，每种组件都有特定的用途。默认情况下，这些组件在 server.xml 文件中进行配置。如果把 Tomcat 嵌入到 Java 应用中，这个文件就无用了，所以必须通过程序来配置这些组件的实例。Tomcat 最主要的组件包括：

- 顶层类组件：Server 和 Service
- 容器类组件：Engine、Host 和 Context
- 连接器：Connector

通过以下 XML 代码可以理解 server.xml 文件中主要组件的嵌套层次：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">

    <Service name="Tomcat-Standalone">

        <Connector className="org.apache.catalina.connector.http.HttpConnector"
            port="8080" minProcessors="5" maxProcessors="75"
            enableLookups="true" redirectPort="8443"
            acceptCount="10" debug="0" connectionTimeout="60000"/>

        <Engine name="Standalone" defaultHost="localhost" debug="0">

            <Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">

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

                <Context path="/jsp-examples" docBase="jsp-examples" debug="0"
                    reloadable="true">
                </Context>
            </Host>
        </Engine>
    </Service>
</Server>
```

```

<Context path="/servlets-examples" docBase="servlets-examples" debug="0"
    reloadable="true">
</Context>

</Host>

</Engine>

</Service>

</Server>

```

在 Java 应用中，也必须创建以上的组件结构。由于`<server>`和`<service>`元素是自动创建的，所以不必在程序中创建这两种对象。除此以外的组件对象都必须在程序中创建，步骤如下。

### 步骤

- (1) 创建 `org.apache.Catalina.Engine` 实例，这个对象代表了 Engine 组件，充当`<Host>`元素的容器。
- (2) 创建 `org.apache.Catalina.Host` 实例，这个对象代表了虚拟主机，应该把这个实例加入到 Engine 对象中。
- (3) 创建若干个 `org.apache.Catalina.Context` 实例，每个实例代表了在虚拟主机中的一个 Web 应用。应该把这些 Context 实例加入到先前创建的 Host 对象中。
- (4) 创建 `org.apache.Catalina.Connector` 实例，把它和先前创建的 Engine 对象关联起来。

为了将 Tomcat 服务器嵌入到 Java 应用中，需要用到 Tomcat 中已经存在的一些类，这些类提供了把 Tomcat 和其他 Java 应用集成的接口，最主要的一个类是 `org.apache.catalina.startup.Embedded`。Embedded 类的主要方法的描述参见表 23-1，也可以参考 Tomcat 的 API 文档，地址为：

`<CATALINA_HOME>/webapps/tomcat-docs/catalina/docs/api/index.html`

表 23-1 Embedded 类的主要方法

方 法	描 述
<code>createEngine</code>	创建 Engine 实例
<code>createHost</code>	创建 Host 实例
<code>createContext</code>	创建 Context 实例
<code>createConnector</code>	创建 Connector 实例
<code>addConnector</code>	把 Connector 实例加入到 Connector 集合中，并且把当前的 Connector 和先前创建的 Engine 实例关联起来
<code>start</code>	启动 Tomcat 服务器
<code>stop</code>	停止 Tomcat 服务器

## 23.2 创建嵌入了Tomcat的Java示范程序

在上一节，介绍了把Tomcat嵌入Java应用必须包含的步骤。例程23-1是一个示范程序，在这段程序中创建并配置了相关的各种Tomcat容器。

例程 23-1 EmbeddedTomcat.java

```
import java.net.URL;
import java.net.InetAddress ;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Deployer;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.logger.SystemOutLogger;
import org.apache.catalina.startup.Embedded;
import org.apache.catalina.Container;

public class EmbeddedTomcat {

    private String path = null;
    private Embedded embedded = null;
    private Host host = null;

    /**
     * Default Constructor
     */
    public EmbeddedTomcat() {
    }

    /**
     * Basic Accessor setting the value of the context path
     * @param      path - the path
     */
    public void setPath(String path) {
        this.path = path;
    }

    /**
     * Basic Accessor returning the value of the context path
     * @return - the context path
     */
    public String getPath() {
        return path;
    }
}
```

```
/**  
 * This method Starts the Tomcat server.  
 */  
public void startTomcat() throws Exception {  
  
    Engine engine = null;  
  
    // Set the home directory  
    System.setProperty("catalina.home", getPath());  
  
    // Create an embedded server  
    embedded = new Embedded();  
    embedded.setDebug(5);  
    embedded.setLogger(new SystemOutLogger());  
  
    // Create an engine  
    engine = embedded.createEngine();  
    engine.setDefaultHost("localhost");  
  
    // Create a default virtual host  
    host = embedded.createHost("localhost", getPath() + "/webapps");  
    engine.addChild(host);  
  
    // Create the ROOT context  
    Context context = embedded.createContext("", getPath() + "webapps/ROOT");  
    host.addChild(context);  
  
    // Create the jsp examples context  
    Context jspExamplesContext = embedded.createContext("/jsp-examples",  
        getPath() + "webapps/jsp-examples");  
    host.addChild(jspExamplesContext);  
  
    // Create the servlet examples context  
    Context servletExamplesContext = embedded.createContext("/servlets-examples",  
        getPath() + "webapps/servlets-examples");  
    host.addChild(servletExamplesContext);  
  
    // Install the assembled container hierarchy  
    embedded.addEngine(engine);  
  
    // Assemble and install a default HTTP connector  
    InetAddress addr=null;  
    Connector connector = embedded.createConnector(addr, 8080, false);  
    embedded.addConnector(connector);
```

```
// Start the embedded server
embedded.start();
}

/**
 * This method Stops the Tomcat server.
 */
public void stopTomcat() throws Exception {
    // Stop the embedded server
    embedded.stop();
}

/**
 * Registers a WAR
 * @param contextPath - the context path under which the
 *                      application will be registered
 * @param url - the URL of the WAR file to be registered.
 */
public void registerWAR(String contextPath, URL url) throws Exception {

    if ( contextPath == null ) {

        throw new Exception("Invalid Path : " + contextPath);
    }
    String displayPath = contextPath;
    if( contextPath.equals("/") ) {
        contextPath = "";
    }
    if ( url == null ) {
        throw new Exception("Invalid WAR : " + url);
    }

    Deployer deployer = (Deployer)host;
    Context context = deployer.findDeployedApp(contextPath);

    if (context != null) {
        throw new Exception("Context " + contextPath + " already Exists!");
    }
    deployer.install(contextPath, url);
}

/**
 * removes a WAR
 * @param contextPath - the context path to be removed
 */
public void unregisterWAR(String contextPath) throws Exception {
```

```
Context context = host.map(contextPath);
if ( context != null ) {
    embedded.removeContext(context);
}
else {
    throw new Exception("Context does not exist for named path : "
        + contextPath);
}
}

public static void main(String args[]) {

    try {

        EmbeddedTomcat tomcat = new EmbeddedTomcat();
        String rootpath=null;
        if(args.length>0)
            rootpath=args[0];
        else
            throw new Exception("Tomcat's root path is unknown.");
        tomcat.setPath(rootpath);
        tomcat.startTomcat();
        Thread.sleep(120000);
        tomcat.stopTomcat();
        System.exit(0);
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
}
}
```

在作为程序入口的 main 方法中，先创建了一个 EmbeddedTomcat 类的实例，然后从 main 方法的参数中读取后面程序将引用的 Tomcat 的安装路径，它相当于 <CATALINA\_HOME> 环境变量。接着调用 startTomcat 方法，在 startTomcat 方法中执行了以下步骤。

### 步骤

(1) 将 Tomcat 的安装路径设置为系统属性：

```
// Set the home directory
System.setProperty("catalina.home", getPath());
```

(2) 创建 Embedded 对象，然后设置跟踪级别和日志：

```
// Create an embedded server
embedded = new Embedded();
```

```
embedded.setDebug(5);
embedded.setLogger(new SystemOutLogger());
```



在发布 Web 应用产品时，应该将跟踪级别设为 0，这样可以提高运行效率。

(3) 创建 Engine 实例，再设置默认的虚拟主机名：

```
// Create an engine
engine = embedded.createEngine();
engine.setDefaultHost("localhost");
```

(4) 创建 Host 对象，名字为“localhost”，路径为<CATALINA\_HOME>/webapps/，再把 Host 对象加入到 Engine 对象中：

```
// Create a default virtual host
host = embedded.createHost("localhost", getPath() + "/webapps");
engine.addChild(host);
```

(5) 创建一些 Context 对象，把这些对象加入到 Host 对象中。在这个例子中，创建了 3 个 Context 实例：ROOT、jsp-examples 和 servlets-examples。这 3 个 Web 应用是 Tomcat 自带的 Web 应用，都位于<CATALINA\_HOME>/webapps 目录下：

```
// Create the ROOT context
Context context = embedded.createContext("", getPath() + "webapps/ROOT");
host.addChild(context);
```

```
// Create the jsp examples context
Context jspExamplesContext = embedded.createContext("/jsp-examples",
getPath() + "webapps/jsp-examples");
host.addChild(jspExamplesContext);
```

```
// Create the servlet examples context
Context servletExamplesContext = embedded.createContext("/servlets-examples",
getPath() + "webapps/servlets-examples");
host.addChild(servletExamplesContext);
```

(6) 把包含了 Host 和 Context 的 Engine 对象加入到 Embedded 对象中：

```
// Install the assembled container hierarchy
embedded.addEngine(engine);
```

(7) 创建 Connector 对象，并且把它和 Engine 对象关联：

```
// Assemble and install a default HTTP connector
Connector connector = embedded.createConnector(null, 8080, false);
embedded.addConnector(connector);
```

(8) 通过 Embedded.start 方法启动 Tomcat 服务器：

```
// Start the embedded server
embedded.start();
```

当 startTomcat 方法执行结束，主程序开始睡眠 2min(120000ms)，在这段时间内，Tomcat 服务器可以响应各种 HTTP 请求。当主程序醒来后，Tomcat 服务器被终止，整个程序运行

完毕：

```
Thread.sleep(120000); //sleep two minutes  
tomcat.stopTomcat();  
System.exit(0);
```

以上程序是一个显示如何将 Tomcat 嵌入到 Java 应用中的简单例子。在这个例子中，我们通过简单的计时方法来终止 Tomcat 服务器。如果想使程序更深入一步，可以通过事件触发机制来终止 Tomcat 服务器。

EmbeddedTomcat 类中还有另外两个方法：registerWar 和 unregisterWar 方法。

registerWar 方法用来发布一个新的 Web 应用。它带有两个参数：一个代表 Web 应用的 context 路径，另一个代表 Web 应用的文件路径。例如，发布 helloapp 应用的代码如下：

```
registerWar("/helloapp", "jar:file:C:/jakarta-tomcat/webapps/helloapp.war");
```

unregisterWar 方法用来根据给定的 ContextPath 删 除已发布的 Web 应用。如果想删除 helloapp 应用，代码如下：

```
unregisterWar("/helloapp");
```

### 23.3 运行嵌入式 Tomcat 服务器

编译和运行以上示范程序时，应该把以下 JAR 文件加入到 CLASSPATH 中：

- <CATALINA\_HOME>\server\lib\catalina.jar
- <CATALINA\_HOME>\common\lib\jmx.jar
- <CATALINA\_HOME>\bin\commons-logging-api.jar
- <CATALINA\_HOME>\server\lib\commons-modeler.jar
- <CATALINA\_HOME>\common\lib\servlet-api.jar
- <CATALINA\_HOME>\common\lib\jasper-compiler.jar
- <CATALINA\_HOME>\common\lib\jasper-runtime.jar
- <CATALINA\_HOME>\common\lib\jsp-api.jar
- <CATALINA\_HOME>\common\lib\naming-common.jar
- <CATALINA\_HOME>\common\lib\naming-factory.jar
- <CATALINA\_HOME>\common\lib\naming-java.jar
- <CATALINA\_HOME>\common\lib\naming-resources.jar
- <CATALINA\_HOME>\server\lib\tomcat-util.jar
- <CATALINA\_HOME>\server\lib\commons-digester.jar
- <CATALINA\_HOME>\server\lib\tomcat-coyote.jar
- <CATALINA\_HOME>\common\lib\commons-collections.jar
- <CATALINA\_HOME>\server\lib\commons-beanutils.jar
- <CATALINA\_HOME>\server\lib\catalina-optional.jar
- <CATALINA\_HOME>\server\lib\servlets-default.jar
- <CATALINA\_HOME>\server\lib\tomcat-http11.jar
- <CATALINA\_HOME>\common\lib\commons-el.jar

在设定 classpath 时，应按照以上列出的先后顺序向 classpath 中添加 JAR 文件，否则可能会导致程序运行出错。此外，上面的 JAR 文件都来自于 Tomcat 5.0.12 版本，如果使用其他版本的 JAR 文件，有些 JAR 文件在两个版本中的名字可能不一样，而且存放位置也可能不一样，因此必须进行相应的改动。

为了便于读者运行本章例子，在本书配套光盘的 sourcecode/chapter23 目录下提供了 EmbeddedTomcat.java 源文件，以及编译和运行 EmbeddedTomcat 的脚本：

- setclasspath.bat：设置 classpath
- compile.bat：编译 EmbeddedTomcat.java
- startup.bat：运行 EmbeddedTomcat，启动嵌入式 Tomcat 服务器

可以按以下步骤来运行本章程序。

## 步骤

(1) 将本书配套光盘的 sourcecode 目录下的整个 chapter23 目录拷贝到本地硬盘。

(2) 修改 setclasspath.bat 中的 catalina\_home 环境变量，把它的值设置为 Tomcat 服务器的安装目录。setclasspath.bat 中的内容如下：

```
set catalina_home=C:\jakarta-tomcat

set
classpath=%catalina_home%\server\lib\catalina.jar;%catalina_home%\common\lib\jmx.jar;%catalina_
_home%\bin\commons-logging-api.jar;%catalina_home%\server\lib\commons-modeler.jar;%catalina_
home%\common\lib\servlet-api.jar;%catalina_home%\common\lib\jasper-compiler.jar;%catalina_ho
me%\common\lib\jasper-runtime.jar;%catalina_home%\common\lib\jsp-api.jar;%catalina_home%\co
mmon\lib\naming-common.jar;%catalina_home%\common\lib\naming-factory.jar;%catalina_home%\c
ommon\lib\naming-java.jar;%catalina_home%\common\lib\naming-resources.jar;%catalina_home%\s
erver\lib\tomcat-util.jar;%catalina_home%\server\lib\commons-digester.jar;%catalina_home%\se
rver\lib\tomcat-coyote.jar;%catalina_home%\common\lib\commons-collections.jar;%catalina_home%\s
erver\lib\commons-beanutils.jar;%catalina_home%\server\lib\catalina-optional.jar;%catalina_home%\s
erver\lib\servlets-default.jar;%catalina_home%\server\lib\tomcat-http11.jar;%catalina_home%\co
mmon\lib\commons-el.jar
```

(3) 运行 compile.bat，该脚本会先调用 setclasspath.bat，然后编译 EmbeddedTomcat.java，编译生成的 EmbeddedTomcat.class 位于当前目录。compile.bat 的内容如下：

```
set currpath=.
if "%OS%" == "Windows_NT" set currpath=%~dp0%

call "%currpath%setclasspath.bat"

javac -d %currpath% -classpath %classpath% %currpath%EmbeddedTomcat.java
```

(4) 运行 startup.bat，该脚本会先调用 setclasspath.bat，然后运行 EmbeddedTomcat 类。startup.bat 中内容如下：

```
set currpath=.
if "%OS%" == "Windows_NT" set currpath=%~dp0%

call "%currpath%setclasspath.bat"
```

```
java -classpath %classpath%;%classpath% EmbeddedTomcat %catalina_home%\
```

本程序运行时，会在 Windows 控制台窗口中看到一些输出日志。如果出现以下的日志内容，就表示 Tomcat 服务器已经启动了：

```
2003-9-27 22:19:24 org.apache.coyote.http11.Http11Protocol init
```

```
信息: Initializing Coyote HTTP/1.1 on port 8080
```

```
2003-9-27 22:19:24 org.apache.coyote.http11.Http11Protocol start
```

```
信息: Starting Coyote HTTP/1.1 on port 8080
```

如果程序正常运行，可以通过浏览器分别访问 ROOT、jsp-examples 和 servlets-examples Web 应用：

<http://localhost:8080>

<http://localhost:8080/jsp-examples/>

<http://localhost:8080/servlets-examples/>

## 23.4 小结

在本章中，我们讨论了将 Tomcat 嵌入到 Java 应用中的步骤，还给出了一个具体的 Java 程序，来说明如何创建并配置各种 Tomcat 组件，以及启动和终止 Tomcat 服务器的方法。

# 第 24 章 在 Tomcat 中配置 SSL

在网络上，信息在源与宿的传递过程中会经过其他的计算机。一般情况下，中间的计算机不会监听路过的信息，但在使用网上银行或者进行信用卡交易的时候有可能被监视，从而导致个人隐私的泄露。由于 Internet 和 Intranet 体系结构的原因，总有某些人能够读取并替换用户发出的信息。随着电子商务的不断发展，人们对信息安全的要求越来越高。因此 Netscape 公司提出了 SSL (Server Socket Layer) 协议，旨在达到在开放网络 (Internet) 上安全保密地传输信息的目的，这种协议在 Web 上获得了广泛的应用。

## 24.1 SSL 简介

SSL 是一种保证在网络上的两个节点之间进行安全通信的机制。SSL 可以用来建立安全的网络连接，网络通信协议如 HTTP 和 IMAP (Internet Messaging Application Protocol) 都可以采用 SSL 机制。把采用了 SSL 机制的 HTTP 称为 HTTPS 协议。HTTP 使用的默认端口为 80，而 HTTPS 使用的默认端口为 443。

IETF (Internet Engineering Task Force) 对 SSL 进行了标准化，制定了 RFC2246 规范，并将其称为 TLS (Transport Layer Security)。从技术上讲，TLS 1.0 与 SSL 3.0 的差别非常微小。

客户在网上商店购物，当他输入信用卡信息进行网上支付交易时，存在两个不安全因素：

- 客户的信用卡信息在网络上传输时有可能被他人截获
- 客户正在访问的 Web 站点是个非法站点，专门从事网上欺诈活动

SSL 使用加密技术实现会话双方信息的安全传递，可以实现信息传递的保密性和完整性，并且会话双方能鉴别对方身份。

### 24.1.1 加密通信

当 Web 客户与 Web 服务器进行通信时，通信数据有可能被网络上其他计算机监视。SSL 使用加密技术实现会话双方之间信息的安全传递，这意味着数据从一端发送到另一端时，发送者先对数据加密，然后再把它发送给接收者。这样，在网络上传输的是经过加密的数据。如果有人在网络上非法截获了这种数据，由于没有解密的密钥，因此无法获得真正的原始数据。接收者接收到加密的数据后，先对数据解密，然后再处理。采用 SSL 的通信过程如图 24-1 所示。客户和服务器的加密通信需要在两端进行设置。多数 Web 浏览器可以支持 40 位或者 128 位的加密，或者两者都支持，而服务器只有在安装服务器安全证书后才可以加密通信。

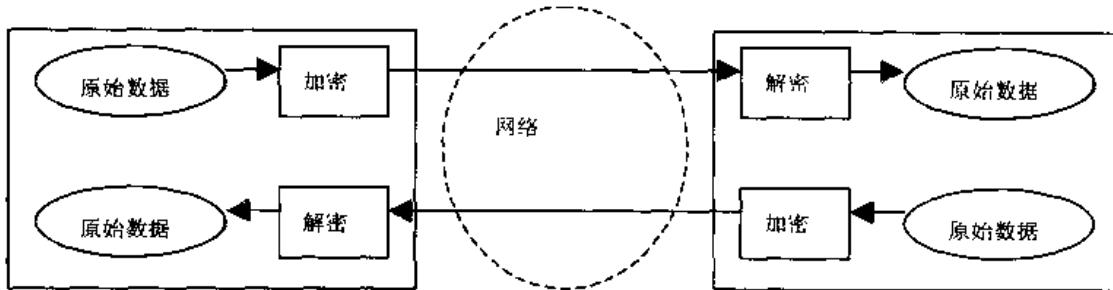


图 24-1 基于 SSL 的加密通信

### 24.1.2 安全证书

除了对数据加密通信，SSL 还采用了身份认证机制，确保通信双方都可以验证对方的真实身份。它和现实生活中人们使用身份证来证明自己的身份很相似。在现实生活中，每人都拥有唯一的身份证件，这个身份证件上记录了个人的真实信息，身份证件由国家权威机构颁发，不允许伪造。在身份证件不能被别人假冒复制的情况下，只要出示身份证件，就可以证明自己的身份。

个人可以通过身份证件来证明自己的身份，对于一个单位，可以通过营业执照来表明身份，营业执照也由国家权威机构颁发，不允许伪造，它保证了营业执照的可信性。

SSL 通过安全证书来表明 Web 客户或 Web 服务器身份。当 Web 客户通过安全的连接与 Web 服务器通信时，Web 服务器会先向客户出示它的安全证书，这个证书声明该 Web 站点是安全的，而且的确是这个站点。每一个证书在全世界范围内都是唯一的，没有其他 Web 站点能假冒原始安全站点的身份，可以把安全证书理解为电子身份证件。在某些情况下，Web 服务器也会要求 Web 客户出示安全证书，以便核实该 Web 客户的身份，这主要用于 B2B (Business to Business) 事务中。在多数情况下，Web 服务器不要求验证 Web 客户的身份。

获取安全证书有两种途径，一种是从权威机构购买证书，还有一种是创建自我签名的证书。

#### 1. 从权威机构获得证书

安全证书在电子商务领域可以有效地保证 Web 站点的可信性。安全证书采用加密技术制作而成，他人几乎无法伪造。安全证书由国际权威的证书机构 (CA, Certificate Authority) 如 VeriSign ([www.verisign.com](http://www.verisign.com)) 和 Thawte ([www.thawte.com](http://www.thawte.com)) 颁发，它们保证了证书的可信性。申请安全证书时，必须支付一定的费用。一个安全证书只对一个 IP 地址有效，如果系统环境中多个 IP 地址，则必须为每个 IP 地址购买安全证书。

#### 2. 创建自我签名证书

在某些场合，通信双方只关心数据在网络上可以安全传输，并不需要对方进行身份验证。在这种情况下，可以创建自我签名 (self-sign) 的证书，比如通过 SUN 公司提供的 keytool 工具就可以创建这样的证书。这样的证书就像是自己制作的名片，缺乏权威性，达不到身份认证的目的。

既然自我签名证书不能有效地证明自己的身份，那么有何意义呢？在技术上，无论是从权威机构获得的证书，还是自己制作的证书，采用的技术都是一样的。使用这些证书，都可以实现安全的加密通信。具体的实现机制将在下面一节介绍。

### 24.1.3 SSL 工作原理

SSL 的安全证书采用了公钥加密技术。公钥加密是使用一对非对称的密钥加密或解密的方法。每一对密钥都由公钥和私钥组成。公钥被广泛发布，私钥是隐密的，不公开。用公钥加密的数据只能够被私钥解密。反过来，使用私钥加密的数据只能用公钥解密。这个非对称的特性使得公钥加密很有用。

在安全证书中包含了这一对非对称的密钥。只有安全证书的所有者才知道私钥。当你将自己的证书发送给其他人时，实际上发给他们的是你的公开密钥，这样他们就可以向你发送用你的公钥加密的数据，只有你才能使用私钥对数据解密并读取加密信息。

安全证书中的数字签名部分是用户的电子身份证。数字签名告诉收件人该信息确实由他发出，不是伪造的，也没有被篡改。

当 Web 客户采用 HTTPS 协议访问安全的 Web 站点时（采用 https://ip:port 形式），Web 站点将自动向客户发送它的安全证书。在 Web 客户与 Web 服务器进行 SSL 握手的阶段，采用非对称加密方法传递数据，由此来建立一个安全的会话，接下来将采用对称加密方法传递实际的通信数据，以下是它们的通信过程。

#### 步骤

- (1) 用户浏览器将自己的 SSL 版本号、加密设置参数、与 Session 有关的数据以及其他一些必要信息发送到服务器。
- (2) 服务器将自己的 SSL 版本号、加密设置参数、与 Session 有关的数据以及其他一些必要信息发送给浏览器，同时发给浏览器的还有服务器的证书。如果服务器的 SSL 需要验证用户身份，还会发出请求要求浏览器提供用户证书。
- (3) 客户端检查服务器证书，如果检查失败，就提示不能建立 SSL 连接。如果成功，那么继续下一步骤。
- (4) 客户端浏览器为本次会话生成预备主密码 (pre-master secret)，并将其用服务器公钥加密后发送给服务器。
- (5) 如果服务器要求鉴别客户身份，客户端还要再对另外一些数据签名后并将其与客户端证书一起发送给服务器。
- (6) 如果服务器要求鉴别客户身份，则检查签署客户证书的 CA 是否可信。如果不在信任列表中，结束本次会话。如果检查通过，服务器用自己的私钥解密收到的预备主密码 (pre-master secret)，并用它通过某些算法生成本次会话的主密码 (master secret)。
- (7) 客户端与服务器均使用此主密码 (master secret) 生成本次会话的会话密钥 (对称密钥)。在双方 SSL 握手结束后传递任何消息均使用此会话密钥。这样做的主要原因是对称加密比非对称加密的运算量低一个数量级以上，能够显著提高双方会话时的运算速度。
- (8) 客户端通知服务器此后发送的消息都使用这个会话密钥进行加密，并通知服务器

客户端已经完成本次 SSL 握手。

(9) 本次握手过程结束，会话已经建立。在接下来的会话过程中，双方使用同一个会话密钥分别对发送以及接受的信息进行加密、解密。

## 24.2 在 Tomcat 中使用 SSL

Tomcat 既可以作为独立的 Servlet 容器，也可以作为其他 HTTP 服务器附加的 Servlet 容器。如果 Tomcat 在非独立模式下工作，通常不必配置 SSL，而由它从属的 HTTP 服务器来实现和客户的 SSL 通信。Tomcat 和 HTTP 服务器之间的通信无需采用加密机制，HTTP 服务器将解密后的数据传给 Tomcat，并把 Tomcat 发来的数据加密后传给客户。

如果 Tomcat 作为独立的 Java Web 服务器，则可以根据安全需要，为 Tomcat 配置 SSL，它包含两个步骤：

- (1) 准备安全证书；
- (2) 配置 Tomcat 的 SSL 连接器（Connector）。

### 24.2.1 准备安全证书

在 24.1.2 节中讲过，获得安全证书有两种方式：一种方式是到权威机构购买，还有一种方式是创建自我签名的证书。本节将介绍后一种方式。

SUN 提供了制作证书的工具 keytool。在 JDK 1.4 以上版本包含了这一工具，它的位置为<JAVA\_HOME>\bin\keytool.exe。此外，也可以到以下站点单独下载 keytool：

<http://java.sun.com/products/jse/>

通过 keytool 工具创建证书的命令为：

keytool -genkey -alias tomcat -keyalg RSA

以上命令将生成一对非对称密钥和自我签名的证书，这个命令中的参数的意思为：

-genkey：生成密钥对（如果没有的话）。

-alias：指定密钥对的别名，该别名是公开的。

-keyalg：指定加密算法，本例中采用通用的 RSA 算法。

该命令的运行过程如图 24-2 所示。首先会提示输入 keystore 的密码，可以输入 Tomcat 默认的密码 changeit，然后提示输入个人信息，如姓名、组织单位和所在城市等，只要输入真实信息即可。接着会提示输入信息是否正确，输入“y”表示信息正确。最后要求输入 <tomcat> 的主密码，这里设置为与 keystore 相同的密码，因此只要根据提示按回车键即可。

以上命令将在操作系统的用户目录下生成名为“.keystore”的文件，假定登录操作系统的用户名为 user1，在 Windows NT/2000 下，该文件的位置为：

C:\Documents and Settings\user1\keystore

在 Linux 下，该文件的位置为：

home\user1\keystore

此外，如果希望生成的 keystore 文件放在其他目录中，可以在 keytool 命令中加入 -keystore 参数，这个参数用来指定 keystore 文件的存放位置，例如以下命令将在 C:\mypath

目录下生成 keystore 文件:

```
keytool -genkey -alias tomcat -keyalg RSA -keystore C:/mypath/keystore
```



图 24-2 用 keytool 生成证书

### 24.2.2 配置 SSL 连接器

在 Tomcat 的 server.xml 中，已经提供了现成的配置 SSL 连接器的代码，只要把 <Connector> 元素的注释去掉即可：

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="8443" minProcessors="5" maxProcessors="75"
    enableLookups="true" disableUploadTimeout="true"
    acceptCount="100" debug="0" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS"/>
```

-->

在 24.1 节中讲过基于 SSL 的 HTTP（即 HTTPS），使用的默认端口为 443。在这里，Tomcat 将 HTTPS 端口设为 8443。Connector 的一些属性的描述参见表 24-1。

表 24-1 SSL Connector 的属性

属性	描述
clientAuth	如果设为 true，表示 Tomcat 要求所有的 SSL 客户出示安全证书，对 SSL 客户进行身份验证
keystoreFile	指定 keystore 文件的存放位置，可以指定绝对路径，也可以指定相对于 CATALINA_BASE 环境变量的相对路径。如果此项没有设定，默认情况下，Tomcat 将从当前操作系统用户的用户目录下读取“keystore”文件
keystorePass	指定 keystore 的密码，如果此项没有设定，默认情况下，Tomcat 将使用“changeit”密码
sslProtocol	指定套接口(Socket)使用的加密/解密协议。默认值为 TLS，用户不便修改这个默认值
ciphers	指定套接口可用的用于加密的密码清单，多个密码间以逗号分隔。如果此项没有设定，默认情况下，套接口可以使用任意一个可用的密码

### 24.2.3 访问支持 SSL 的 Web 站点

由于 SSL 技术已建立到大多数浏览器和 Web 服务器程序中，因此，仅需在 Web 服务器端安装服务器证书就可以激活 SSL 功能了。

如果已经按以上步骤在 Tomcat 中配置好 SSL，就可以启动 Tomcat 服务器，然后从 IE 浏览器中以 HTTPS 方式访问在 Tomcat 服务器上的任何一个 Web 应用。例如，可以访问如下地址：

<https://localhost:8443>

当 Tomcat 接收到这一 HTTPS 请求后，会向客户的浏览器发送服务器的安全证书，IE 浏览器接收到证书后，将向客户显示安全警报窗口，如图 24-3 所示。

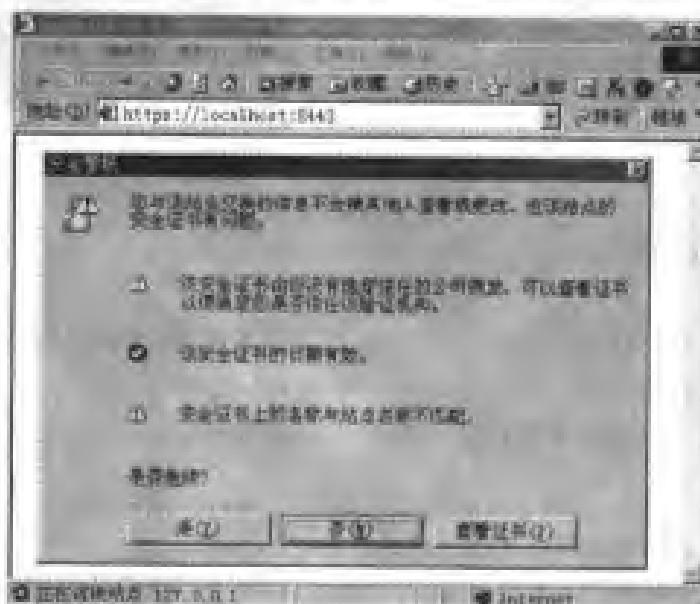


图 24-3 IE 的安全警报窗口

在安全警报窗口中的第一行提示信息为：“您与该站点交换的信息不会被其他人查看或更改。但该站点的安全证书有问题。”

以上信息的意思是，一方面，该安全证书非权威机构颁发，不能作为有效的验证对方身份的凭据。另一方面，假如与对方通信，通信数据会经过加密后在网络上传输，因此不会被其他人监视或修改。

在安全警报窗口中，如果单击【否】按钮，就表示不信任 Tomcat 服务器出示的证书，因此浏览器会结束与 Tomcat 服务器的通信。

如果单击【是】按钮，表示信任 Tomcat 服务器出示的证书，浏览器将建立与 Tomcat 服务器的 SSL 握手，Tomcat 服务器接着把客户请求的数据发送过来。对于以上 URL，将在浏览器端显示 Tomcat 的默认 index.jsp 主页。

如果单击【查看证书】按钮，将出现证书窗口，如图 24-4 所示。从图中可以看到证书的“颁发者”和“颁发给”都是同一个人，说明这是自我签名的证书，非权威机构颁发。



图 24-4 证书窗口

在图 24-4 中选择“详细信息”标签，将显示证书的详细信息，如图 22-5 所示。



图 24-5 证书的详细信息

可以看出，在证书中公布了证书发送者的身份信息和公钥。而私钥只有证书发送者拥有，不会向证书接收者公开。

## 24.3 小结

SSL (Server Socket Layer) 是一种保证在网络上的两个节点之间进行安全通信的机制。SSL 使用加密技术实现会话双方信息的安全传递，可以实现信息传递的保密性、完整性，并且会话双方能鉴别对方身份。SSL 通过安全证书来表明 Web 客户或 Web 服务器身份。当 Web 客户通过安全的连接与 Web 服务器通信时，Web 服务器会先向客户出示它的安全证书，这个证书声明该 Web 站点是安全的，而且的确是这个站点。获取安全证书有两种途径，一种是从权威机构购买证书，还有一种办法是创建自我签名的证书。我们把采用了 SSL 机制的 HTTP 称为 HTTPS 协议。HTTP 使用的默认端口为 80，而 HTTPS 使用的默认端口为 443。Web 客户可以通过 HTTPS 协议访问安全的 Web 站点，形式为 `https://ip:port/`。

# 第 25 章 JSP 2.0 的新特征

SUN 公司新发布的 JSP 2.0 版是对 JSP 1.2 的升级，增加了一些有趣的新特性。JSP 2.0 的目标是使动态网页的设计、开发和维护更加容易，网页编写者不必懂得 Java 编程语言，也可以编写 JSP 网页。JSP 2.0 增加了一种称为 SimpleTag 的扩展机制来简化标签 API (Tag API)。JSP 2.0 引入的最主要的新特性包括：

- 引入简单表达式语言 (EL, Expression Language)，它用于 JSP 页面中的数据访问。这种表达式语言简化了 JSP 中数据访问的代码，不需要使用 Java Scriptlet 或者 Java 表达式
- 引入创建自定义标签的新语法，该语法使用.tag 和.tagx 文件，这类文件可由开发人员或者网页作者编写
- 对 XML 语法做了实质性的改进，增加了新的标准文件扩展名 (.tagx 用于标签文件，.jspx 用于 JSP 文件)

当新的 JSP 版本产生后，最初打算把版本号定为 1.3，但由于这些新特性对 JSP 应用程序的开发模型产生了非常深刻的影响，专家组感到有必要把主版本号升级到 2.0，这样才能充分反映这种影响。此外，新的版本号也有助于把开发人员的注意力吸引到这些有趣的新特性上来。令人欣慰的是，所有合法的 JSP 1.2 页面同时也是合法的 JSP 2.0 页面。

本章主要讨论表达式语言、简化的标签 API 和标签文件。Jakarta Tomcat 5 支持新的 JSP 2.0 和 Servlet 2.4 规范，因此下文将以 Tomcat 5 作为运行 JSP 2.0 的容器。

## 25.1 JSP 表达式语言

在 JSP 2.0 中引进的表达式语言 (EL)，是一种简洁的数据访问语言。通过它可以在 JSP 网页中方便地访问应用程序数据，无需使用 Scriptlet (<% 和 %>) 或者请求时 (request-time) 表达式 (<%= 和 %>)。



尽管这种表达式语言是 JSP 的一个重要特性，但它并不是一种通用的编程语言，它仅仅是一种数据访问语言。

在 JSP 2.0 之前，网页作者只能使用表达式 <%= 和 %> 访问应用程序数据，比如下面的例子：

Your name is : <%=name%>

表达式语言允许网页作者使用简单的语法访问数据。比如要访问一个变量，可以采用如下形式：

Your name is: \${name}

### 25.1.1 访问数据

表达式语言可以使用点号运算符“.”访问对象的属性，比如表达式\${customer.name}表示对象 customer 的 name 属性。

表达式语言也可使用方括号运算符“[]”访问对象的属性，比如表达式\${customer["name"]}和\${customer.name}是等价的。

方括号运算符“[]”也可以用来访问数组中的元素，比如\${customers[0]}表示访问数组 customers 中的第一个元素。

### 25.1.2 运算符

表达式语言支持算术运算符、关系运算符和逻辑运算符，以完成大多数的数据处理操作。它还提供了一个用于测试对象是否为空的特殊运算符“empty”。empty 运算符判断某个集合是否为空或字符串是否为 null。比方说，对于表达式\${empty customer.name}，如果 customer.name 为 null，就返回 true。empty 运算符可以与“!”运算符一起使用，比如对于\${!empty customer.name}，如果 customer.name 不为 null，就返回 true。

所有的运算符参见表 25-1。

表 25-1 表达式语言运算符

运 算 符	说 明
+	加
-	减
*	乘
/ 或 div	除
% 或 mod	模（求余）
== 或 =	等于
!=	不等于
< 或 lt	小于
> 或 gt	大于
<= 或 le	小于等于
>= 或 ge	大于等于
&& 或 and	逻辑与
or or	逻辑或
! 或 not	逻辑非
empty	检查是否为空值
a ? b : c	条件运算符

### 25.1.3 隐含对象

除了运算符外，表达式语言还定义了一些隐含对象以支持网页作者访问特定的应用程

序数据。表达式语言定义的隐含对象参见表 25-2 所示。

表 25-2 表达式语言中的隐含对象

隐含对象	说 明
applicationScope	Web 应用范围内的所有对象的集合
cookie	所有 cookie 组成的集合
header	HTTP 请求头部，采用字符串形式
headerValues	HTTP 请求头部，采用字符串集合形式
initParam	Web 应用的所有初始化参数名组成的集合
pageContext	当前页面的 javax.servlet.jsp.PageContext 对象
pageScope	页面范围内所有对象的集合
param	所有请求参数名组成的集合
paramValues	所有请求参数值组成的集合
requestScope	所有请求范围内的对象的集合
sessionScope	所有会话范围内的对象的集合

#### 25.1.4 表达式语言的例子

网页作者无需学习 Java 也能够使用这种表达式语言。下面是使用表达式语言的 JSP 例子。本章所有例子的源文件都位于本书配套光盘的 sourcecode/chapter25 目录下。

##### 1. 关于基本语法的例子

例程 25-1 显示了表达式语言的基本语法和隐含对象的使用。

例程 25-1 syntax.jsp

```
<HTML>
<HEAD>
<TITLE>Expression Language Examples</TITLE>
</HEAD>

<BODY>

<H3>JSP Expression Language Examples</H3>
<P>
The following table illustrates some EL expressions and implicit objects:

<TABLE BORDER="1">
<THEAD>
<TD><B>Expression</B></TD>
<TD><B>Value</B></TD>
</THEAD>
<TR>
<TD>${2 + 5}</TD>
<TD>$[2 + 5]</TD>
```

```
<TR>
    <TD>\${4/5}</TD>
    <TD>\${4/5}</TD>
</TR>
<TR>
    <TD>\${5 div 6}</TD>
    <TD>\${5 div 6}</TD>
</TR>
<TR>
    <TD>\${5 mod 7}</TD>
    <TD>\${5 mod 7}</TD>
</TR>
<TR>
    <TD>\${2 < 3}</TD>
    <TD>\${2 < 3}</TD>
</TR>
<TR>
    <TD>\${2 gt 3}</TD>
    <TD>\${2 gt 3}</TD>
</TR>
<TR>
    <TD>\${3.1 le 3.2}</TD>
    <TD>\${3.1 le 3.2}</TD>
</TR>
<TR>
    <TD>\${(5 > 3) ? 5 : 3}</TD>
    <TD>\${(5 > 3) ? 5 : 3}</TD>
</TR>
<TR>
    <TD>\${header["host"]}</TD>
    <TD>\${header["host"]}</TD>
</TR>
<TR>
    <TD>\${header["user-agent"]}</TD>
    <TD>\${header["user-agent"]}</TD>
</TR>

</TABLE>

</BODY>
</HTML>
```

假定把 syntax.jsp 发布到 helloapp 应用中，可以把 syntax.jsp 拷贝到 **<CATALINA\_HOME>/webapps/helloapp** 目录下。此外，对于采用 JSP 2.0 新语法的 Web 应用，则应该对它的配置文件 web.xml 进行如下修改（以下代码中粗体部分为增加的内容）：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">
  .....
</web-app>
```

接下来启动 Tomcat 服务器，访问 <http://localhost:8080/helloapp/syntax.jsp>，生成的网页如图 25-1 所示。

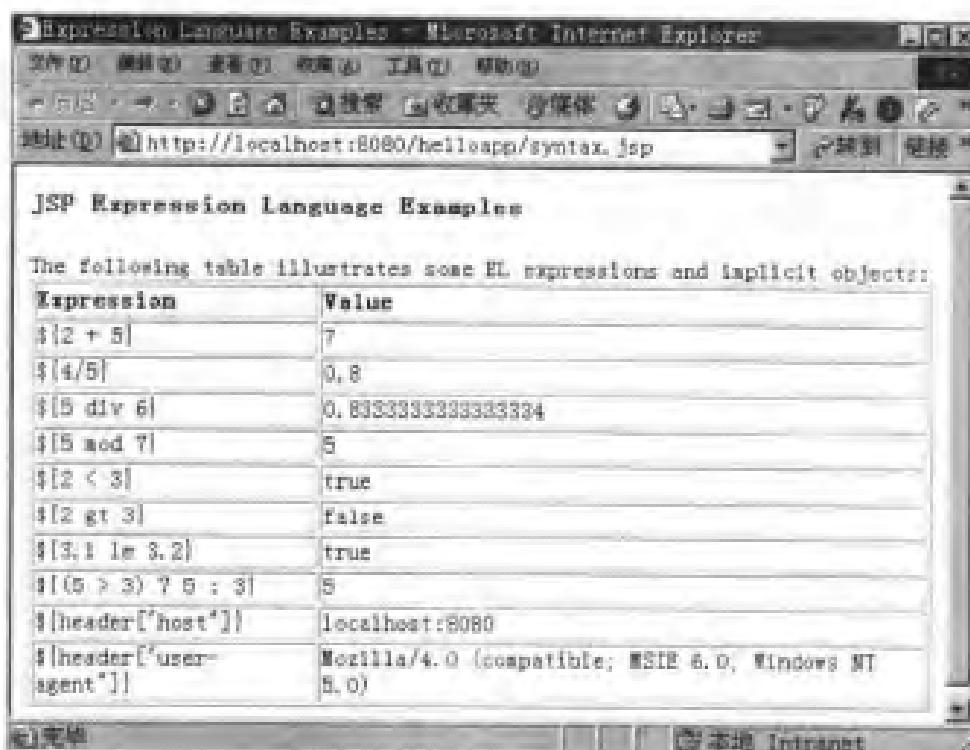


图 25-1 syntax.jsp 网页

## 2. 读取表单数据的例子

隐含对象 \$param[var] 可用于读取表单的数据。例程 25-2 给出了一个简单的表单，它包含一个“name”参数。

例程 25-2 form.jsp

```
<HTML>
<HEAD>
<TITLE>Form Content</TITLE>
</HEAD>
```

```
<BODY>

<H3>Fill-out-form</H3>
<P>
<FORM action="form.jsp" method="GET">
    Name = <input type="text" name="name" value="${param['name']}>
    <input type="submit" value="Submit Name">
</FORM>
<P>
The Name is: ${param.name}
</BODY>
</HTML>
```

同样，运行这个例子只需要把 form.jsp 拷贝到<CATALINA\_HOME>/webapps/helloapp 目录下，再通过浏览器访问 <http://localhost:8080/helloapp/form.jsp>。

在网页表单中输入名字并单击【提交】按钮，输入的名字就会出现在同一页面中的“**The Name is:**”字样后面，如图 25-2 所示。

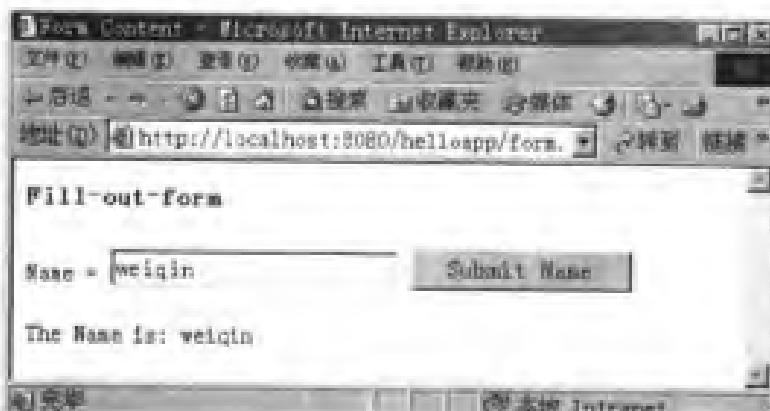


图 25-2 form.jsp 网页

### 3. 定义和使用函数的例子

表达式语言可以访问函数。函数必须定义在 public 类型的类中，并且函数本身必须声明为 public static 类型。函数定义好以后，应该把函数签名（signature）映射到标签库描述符（TLD）文件中。

为了说明函数的使用，这里举一个简单的函数的例子，该函数用于对两个数求和。首先要编写求两数之和的 Java 方法代码，在 Compute.java 中定义了一个静态方法，它接收两个字符串参数，把它们解析成整数并返回它们的和。

以下是 Compute.java 的代码：

```
/** Compute.java */
package mypack;
import java.util.*;
public class Compute {
    public static int add(String x, String y) {
```

```

int a = 0;
int b = 0;
try {
    a = Integer.parseInt(x);
    b = Integer.parseInt(y);
} catch(Exception e) {}
return a + b;
}
}

```

下面编写一个 JSP 页面来使用这个函数。sum.jsp 中提供了一个包含两个字段的表单，用户输入两个数字并单击【求和】按钮，就会调用上面的函数把两个数相加，结果显示在同一个页面上。

以下是 sum.jsp 的源代码：

```

<%@ taglib prefix="my" uri="/jsp2-example-taglib %>
<HTML>
<HEAD>
<TITLE>Functions</TITLE>
</HEAD>

<BODY>

<H3>Add Numbers</H3>
<P>
<FORM action="sum.jsp" method="GET">
    X = <input type="text" name="x" value="${param['x']}">
    <BR>
    Y = <input type="text" name="y" value="${param['y']}"/>
    <input type="submit" value="Add Numbers">
</FORM>

<P>
The sum is: ${my:add(param['x'],param['y'])}

</BODY>
</HTML>

```

运行这个例子的步骤如下。

### 步骤

(1) 编译 Compute.java，把编译生成的类文件放到以下位置：

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/classes/mypack/Compute.class

(2) 把 add 函数的签名映射到标签库中。方法为在<CATALINA\_HOME>/webapps/helloapp/WEB-INF 目录下创建 jsp2-example-taglib.tld 文件，代码如下：

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

```

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsp-taglibrary_2_0.xsd"
    version="2.0">
    <tlib-version>1.0</tlib-version>
    <short-name>jsp2-example-taglib</short-name>
    <uri>/jsp2-example-taglib</uri>

    <function>
        <description>add x and y</description>
        <name>add</name>
        <function-class>mypack.Compute</function-class>
        <function-signature>
            int add(java.lang.String,java.lang.String)
        </function-signature>
    </function>

</taglib>

```

(3) 在<CATALINA\_HOME>/webapps/helloapp/WEB-INF/web.xml 文件中加入<taglib>元素：

```

<taglib>
    <taglib-uri>
        /jsp2-example-taglib
    </taglib-uri>
    <taglib-location>
        /WEB-INF/jsp2-example-taglib.tld
    </taglib-location>
</taglib>

```

(4) 把 sum.jsp 拷贝到<CATALINA\_HOME>/webapps/helloapp 目录下。

(5) 启动 Tomcat 服务器，通过浏览器访问 <http://localhost:8080/helloapp/sum.jsp>。如果运行正常，会看到如图 25-3 所示的窗口。



图 25-3 sum.jsp 网页

## 25.2 简单标签扩展

JSP 1.2 中的标签处理 API 由于允许标签体中包含 Scriptlet 而变得复杂，现在利用表达式语言可以编写不含 Scriptlet 的 JSP 网页。JSP 2.0 引入了一种新的标签扩展机制，称为“简单标签扩展”，这种机制有两种使用方式：

- Java 开发人员可以定义实现接口 javax.servlet.jsp.tagext.SimpleTag 的类
- 不懂 Java 的网页编写人员则可以使用标签文件

### 25.2.1 实现 SimpleTag 接口

以下程序 HelloTag.java 是一个简单的标签处理类，它的作用是打印 “This is my first tag!”：

```
package mypack;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;
/**
 * SimpleTag handler that prints "This is my first tag!"
 */
public class HelloTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().write("This is my first tag!");
    }
}
```

以下是使用上述标签的 JSP 例子 helloworld.jsp：

```
<%@ taglib prefix="mytag" uri="/jsp2-example-taglib" %>
<HTML>
<HEAD>
<TITLE>Simple Tag Handler</TITLE>
</HEAD>
<BODY>
<H2>Simple Tag Handler</H2>
<P><B>My first tag prints</B>: <mytag:hello/></BODY></HTML>
```

运行上述例子的步骤如下。

#### 步骤

(1) 编译 HelloTag.java，编译时应该把<CATALINA\_HOME>/common/lib/jsp-api.jar 文件加入到 classpath 中。把编译生成的类文件放到以下位置：

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/classes/mypack>HelloTag.class

(2) 在<CATALINA\_HOME>/webapps/helloapp/WEB-INF/jsp2-eample-taglib.tld 文件中

添加如下标签描述符：

```
<tag>
<description>Prints this is my first tag</description>
<name>hello</name>
<tag-class>mypack.HelloTag</tag-class>
<body-content>empty</body-content>
</tag>
```

(3) 把 helloworld.jsp 拷贝到<CATALINA\_HOME>/webapps/helloapp 目录中。

(4) 启动 Tomcat 服务器, 通过浏览器访问 <http://localhost:8080/helloworld/helloworld.jsp>。会看到如图 25-4 所示的网页。

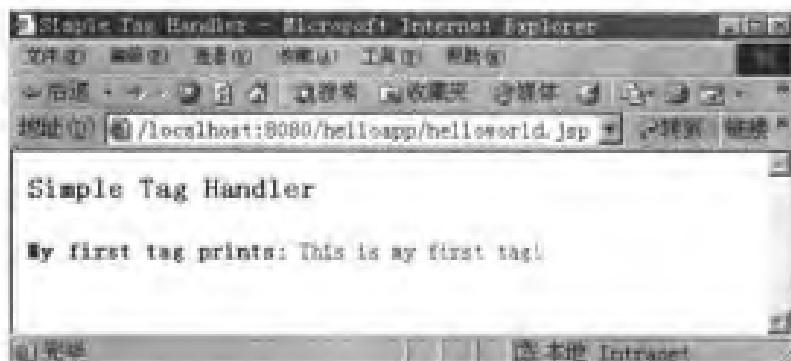


图 25-4 helloworld.jsp 网页

### 25.2.2 使用标签文件

使用简单标签扩展机制的另一种方法是采用标签文件。标签文件是一种资源文件，网页作者可以利用它抽取一段 JSP 代码，通过定制功能来实现代码的复用。换句话说，标签文件允许 JSP 网页作者使用 JSP 语法创建可复用的标签库。标签文件的扩展名必须是“.tag”。

以下是一个非常简单的标签文件 greentings.tag，它仅包含一串字符：

```
Hello there. How are you doing?
```

一旦定义了标签文件，就可以在 JSP 网页中使用这种定制的功能。参看以下 chat.jsp 网页：

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<HTML>
<HEAD>
<TITLE>JSP 2.0 Examples - Hello World Using a Tag File</TITLE>
</HEAD>
<BODY>
<H2>Tag File Example</H2>
<P><B>The output of my first tag file is</B>: <tags:greetings/>
</BODY>
</HTML>
```

运行这个例子的步骤如下。

## 步骤

- (1) 把标签文件 greetings.tags 拷贝到<CATALINA\_HOME>/webapps/helloapp/WEB-INF/tags 目录下。
- (2) 把 chat.jsp 文件拷贝到<CATALINA\_HOME>/webapps/helloapp 目录下。
- (3) 启动 Tomcat 服务器，通过浏览器访问 <http://localhost:8080/helloapp/chat.jsp>。如果运行正常，会看到如图 25-5 所示的网页。

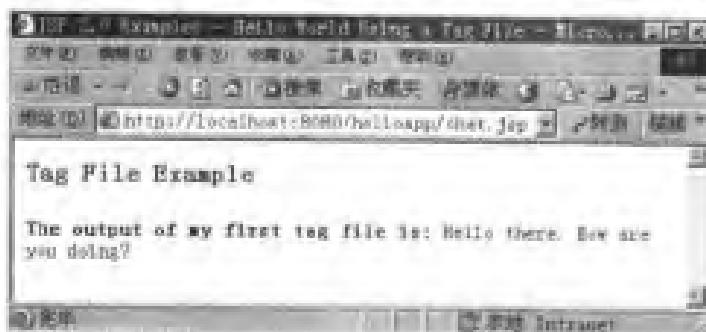


图 25-5 chat.jsp 网页

**技巧** 通过标签文件来创建标签时，不必在 TLD 文件中添加标签描述符，而可以把标签文件并放在特殊的目录中，然后在 JSP 网页中通过 taglib 指令导入并直接使用它。

### 25.2.3 用做模板的标签文件

标签文件可以作为模板使用。例如，以下标签文件 display.tag 定义了一个表格模板，指令 attribute 类似于 TLD 文件中的<attribute>元素，允许声明自定义的属性。

以下是 display.tag 的代码：

```
<%@ attribute name="color" %>
<%@ attribute name="bgcolor" %>
<%@ attribute name="title" %>
<TABLE border="0" bgcolor="\${color}">
<TR>
    <TD><B>\${title}</B></TD>
</TR>
<TR>
    <TD bgcolor="\${bgcolor}">
        <jsp:doBody/>
    </TD>
</TR>
</TABLE>
```

例程 25-3 (newsportal.jsp) 是使用上述标签的例子。

友

Tomcat  

(3) 启动 Tomcat 服务器，通过浏览器访问 <http://localhost:8080/helloapp/newsportal.jsp>，会看到如图 25-6 所示的网页。

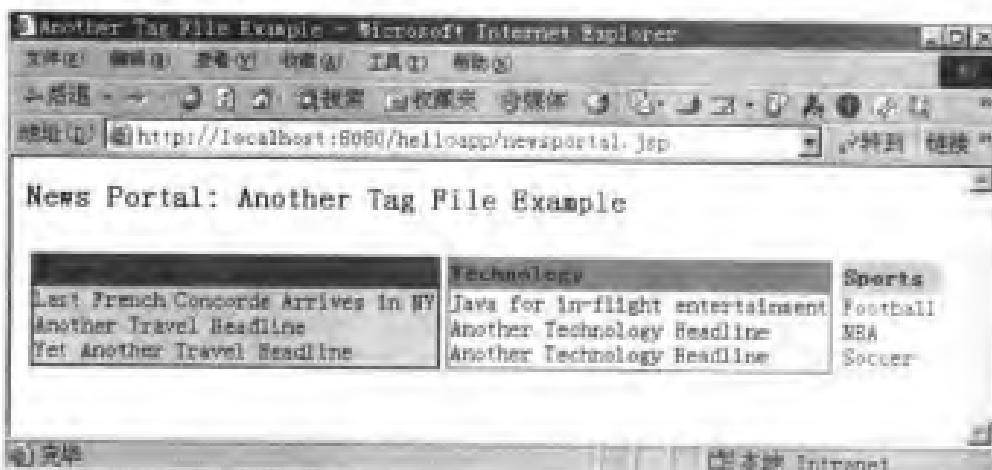


图 25-6 newsportal.jsp 网页

### 25.3 小结

JSP 2.0 版是对 JSP 1.2 的升级，所有合法的 JSP 1.2 页面同时也是合法的 JSP 2.0 页面。JSP 2.0 的目标是使动态网页的设计、开发和维护更加容易，网页编写者不懂得 Java 编程语言，也可以编写 JSP 网页。JSP 2.0 引入了简单表达式语言（EL，Expression Language），它用于 JSP 页面中的数据访问。这种表达式语言简化了 JSP 中数据访问的代码，不需要使用 Java Scriptlet 或者 Java 表达式；JSP 2.0 还引入创建自定义标签的新语法，该语法使用.tag 和.tagx 文件，这类文件可由开发人员或者网页作者编写。Jakarta Tomcat5 实现了 JSP 2.0 和 Servlet 2.4 规范。

在本书配套光盘的 sourcecode/chapter25/helloapp 目录下包含了本章所有的源文件。只要把整个 helloapp 目录拷贝到<CATALINA\_HOME>/webapps 目录下，就可以运行 helloapp 应用。

# 第 26 章 Velocity 模板语言

Velocity 是 Apache 软件组织提供的一项开放源码项目，它是一个基于 Java 的模板引擎。网页作者可以通过 Velocity 模板语言定义模板（template），在模板中不包含任何 Java 程序代码。Java 开发人员编写程序代码来设置上下文（context），它包含了用于填充模板的数据。Velocity 引擎能够把模板和上下文合并起来，生成动态网页。

Velocity 模板语言（VTL）旨在为网页作者提供简捷的方法，来生成动态网页内容。即使没有编程经验的网页作者也可以很快掌握 VTL 语言。VTL 模板和 JSP 网页的区别在于：VTL 模板中不包含任何 Java 代码，并且 VTL 模板不用经过 JSP 编译器的编译，VTL 模板的解析是由 Velocity 引擎来完成的。

尽管 Velocity 也可用于其他独立应用程序的开发，其主要用途是简化 Web 应用开发。Velocity 将 Java 代码从 Web 页面中分离出来，使 Web 站点在长时间运行后仍然具有很好的可维护性。

本文首先通过一个简单的 Velocity 例子来讲解创建基于 Velocity 的 Web 应用的步骤，然后详细介绍 Velocity 模板语言的各个要素。

## 26.1 安装 Velocity

访问 <http://jakarta.apache.org/builds/>，下载 Velocity 软件，也可以从本书配套光盘的 software 目录下获得 Velocity 软件 velocity-1.3.1.zip。把 velocity-1.3.1.zip 文件解压到本地硬盘，在其根目录下有两个 JAR 文件：velocity-1.3.1.jar 和 velocity-dep-1.3.1.jar。在开发 Velocity 应用程序时将用到这两个 JAR 文件。

## 26.2 Velocity 的简单例子

创建基于 Velocity 的 Web 应用包括两个步骤：

- 创建 Velocity 模板
- 创建扩展 VelocityServlet 的 Servlet 类

### 26.2.1 创建 Velocity 模板

下面使用 Velocity 模板语言，定义一个简单的模板，它把两个数相加，并显示其结果。在 Velocity 模板语言中，“\$”符号表示跟随其后的字符串为变量。如果要把“\$”符号作为普通的字符串处理，应该采用“\\$”的形式。

以下是 add.vm 的代码:

```
<html>
<head> <title>Velocity Example</title></head>
<body>
<h1>Velocity Example</h1>
<p>$a+$b=$c</p>
</body>
</html>
```

## 26.2.2 创建扩展 VelocityServlet 的 Servlet 类

在 Velocity API 中提供了 VelocityServlet 类, 它是 HttpServlet 类的子类。在 VelocityServlet 类中包含两个重要方法: loadConfiguration 方法和 handleRequest 方法。

### 1. loadConfiguration 方法

loadConfiguration 方法类似于 HttpServlet 类的 init 方法, 区别在于 loadConfiguration 方法返回 java.util.Properties 对象。loadConfiguration 方法的定义如下:

```
protected Properties loadConfiguration(ServletConfig config)
throws IOException,FileNotFoundException
```

### 2. handleRequest 方法

handleRequest 方法类似于 HttpServlet 类的 doGet 和 doPost 方法, 区别在于 handleRequest 方法中增加了一个 org.apache.velocity.context.Context 类型的参数。Context 类用来存放所有用于显示到 HTML 页面上的数据。handleRequest 方法的定义如下:

```
public Template handleRequest(HttpServletRequest request, HttpServletResponse response, Context
context);
```

下面, 创建扩展了 VelocityServlet 的 AddServlet 类。在它的 loadConfiguration 方法中把 add.vm 所在的文件路径以及 Velocity 日志文件路径作为属性存放在 Properties 对象中, 然后返回 Properties 对象:

```
Properties p=new Properties();
.....
p.setProperty(Velocity.FILE_RESOURCE_LOADER_PATH,path);
p.setProperty("runtime.log",path+"velocity.log");
return p;
```

在 handleRequest 方法中把变量 a、变量 b 和变量 c 对应的数据存放在 context 对象中:

```
int a=11;
int b=22;
int c=a+b;
context.put("a",new Integer(a));
context.put("b",new Integer(b));
context.put("c",new Integer(c));
```

handleRequest 方法接下来获得 add.vm 模板, 然后返回代表 add.vm 模板的 Template 对象:

```
Template outty=null;
```

```
.....
outty=getTemplate("add.vm");
.....
return outty;
```

例程 26-1 是 AddServlet 的源程序。

例程 26-1 AddServlet.java

```
package mypack;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.velocity.*;
import org.apache.velocity.context.*;
import org.apache.velocity.servlet.*;
import org.apache.velocity.app.*;
import org.apache.velocity.exception.*;

public class AddServlet extends VelocityServlet
{
    protected Properties loadConfiguration(ServletConfig config)
        throws IOException,FileNotFoundException{

        Properties p=new Properties();
        String path=config.getServletContext().getRealPath("/");
        if(path==null){
            System.out.println("VelocityAdd.loadConfiguration():"+
                "unable to get the current webapp root. Using '/'.");
            path="/";
        }
        p.setProperty(Velocity.FILE_RESOURCE_LOADER_PATH,path);
        p.setProperty("runtime.log",path+"velocity.log");

        return p;
    }

    public Template handleRequest(HttpServletRequest request, HttpServletResponse
        response,Context
        context){

        Template outty=null;

        try{
            int a=11;
```

```

int b=22;
int c=a+b;
context.put("a",new Integer(a));
context.put("b",new Integer(b));
context.put("c",new Integer(c));

outty=getTemplate("add.vm");
}catch(ParseErrorException ex1){
    System.out.println("VelocityAdd: parse error for template "+ex1);
}catch(ResourceNotFoundException ex2){
    System.out.println("VelocityAdd: template not found "+ex2);
}catch(Exception ex3){
    System.out.println("VelocityAdd: error "+ex3);
}
return outty;
}

)

```

### 26.2.3 发布和运行基于 Velocity 的 Web 应用

假定把以上例子发布到 helloapp 应用中，将创建如图 26-1 所示的目录结构。

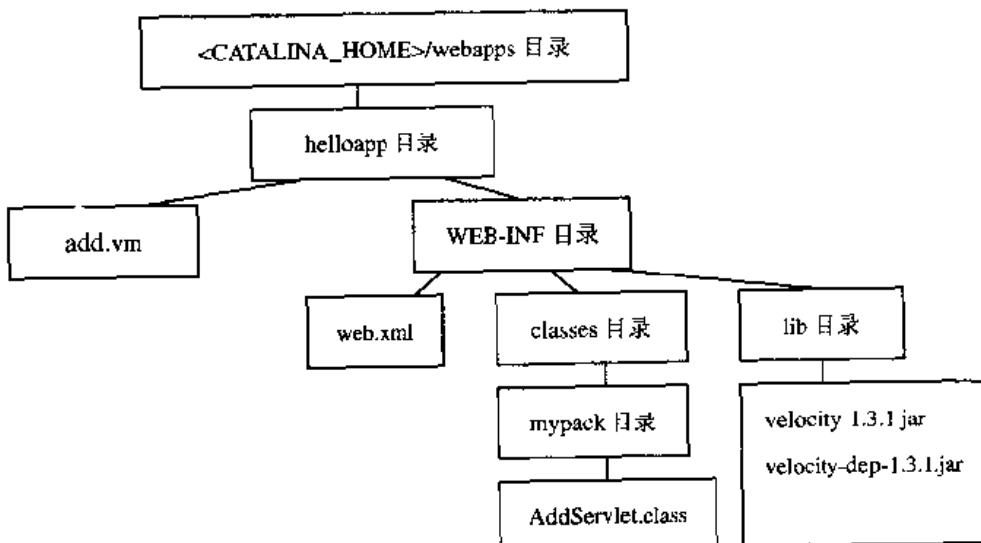


图 26-1 helloapp 应用的目录结构

以下是操作步骤。



- (1) 编译 AddServlet.java，编译 AddServlet 时需要把 servlet-api.jar 和 velocity-1.3.1.jar 加入到 classpath 中。

(2) 把编译生成的 AddServlet.class 拷贝到以下位置:

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/classes/mypack/AddServlet.class

(3) 把 add.vm 拷贝到以下位置:

<CATALINA\_HOME>/webapps/helloapp/add.vm

(4) 把 velocity-1.3.1.jar 和 velocity-dep-1.3.1.jar 拷贝到以下位置:

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/lib/velocity-1.3.1.jar

<CATALINA\_HOME>/webapps/helloapp/WEB-INF/lib/velocity-dep-1.3.1.jar

(5) 在<CATALINA\_HOME>/webapps/helloapp/WEB-INF/web.xml 中配置 AddServlet。代码如下:

```
<servlet>
    <servlet-name>add</servlet-name>
    <servlet-class>mypack.AddServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>add</servlet-name>
    <url-pattern>/add</url-pattern>
</servlet-mapping>
```

(6) 启动 Tomcat 服务器, 访问 <http://localhost:8080/helloapp/add>, 会出现如图 26-2 所示的网页。



图 26-2 add.vm 模板生成的网页

在本书配套光盘的 sourcecode/chaper26 目录下提供了 compile.bat 文件以及 helloapp 应用的所有源文件。compile.bat 是编译 AddServlet 类的脚本文件。只要把整个 helloapp 目录拷贝到<CATALINA\_HOME>/webapps 目录下, 即可以运行本章介绍的 helloapp 应用。

## 26.3 注释

在 VTL 中, 单行注释的前导符为 “##”, 对于多行注释, 采用 “##” 和 “##” 符号。例如:

```
<p>This text is visible.</p> ## This text is not.
<p>This text is visible. </p>
```

```
<p>This text is visible. </p> /* This text, as part of a multi-line comment,  
is not visible. This text is not visible; it is also part of the  
multi-line comment. This text still not visible. */ <p>This text is outside  
the comment, so it is visible. </p>  
## This text is not visible.
```

注释部分的内容不会被输出到网页上，因此以上代码的输出为：

```
This text is visible.  
This text is visible.  
This text is visible.  
This text is outside the comment, so it is visible.
```

## 26.4 引用

VTL 中有 3 种类型的引用：变量、属性和方法。下面分别讲述这 3 种引用。

### 26.4.1 变量引用

变量引用的简略标记是由一个前导“\$”字符后跟一个 VTL 标识符 (Identifier) 组成。一个 VTL 标识符必须以一个字母开始 (a..z 或 A..Z)，剩下的字符将由以下类型的字符组成：

- 字母 (a..z, A..Z)
- 数字 (0..9)
- 连字符 (“-”)
- 下划线 (“\_”)

下面是一些有效的变量引用：

```
$foo  
$mudSlinger  
$mud-slinger  
$mud_slinger  
$mudSlinger1
```

给引用变量赋值，有两种办法。一种办法是在 Java 代码中给变量赋值（例如 26.2 节介绍的例子就采用这种方法），此外，也可以在模板中通过#set 指令给变量赋值，例如：

```
#set( $foo = "bar" )  
The output is $foo.
```

紧跟#set 指令后的所有\$foo 变量的值都为"bar"。以上代码的输出为：

```
The output is bar.
```

### 26.4.2 属性引用

VTL 引用的第二种元素是属性。属性引用的简略标记是前导符\$后跟一个 VTL 标识符，再后跟一个点号 (“.”），最后又是一个 VTL 标识符。以下是一些有效的示例：

```
$client.phone  
$client.firstname
```

下面举一个演示属性引用的完整的模板例子 properties.vm。代码如下：

```
<html>  
<head> <title>Velocity Example</title></head>  
<body>  
<h1>Velocity Properties Example</h1>  
<p>Clients First Name:$client.firstname</p>  
<p>Clients Last Name:$client.lastname</p>  
<p>Clients Phone Number:$client.phone</p>  
</body>  
</html>
```

给引用属性赋值，有两种办法。一种办法是在 Java 代码中创建一个 Hashtable 对象，把所有的属性保存在 Hashtable 对象中，再把 Hashtable 对象保存在 Context 对象中。例如，可以创建一个扩展了 VelocityServlet 类的 PropertiesServlet 类，以下是它的 handleRequest 方法：

```
public Template handleRequest(HttpServletRequest request, HttpServletResponse  
response,Context context){  
  
    Template outty=null;  
    try{  
        Hashtable client=new Hashtable();  
        client.put("firstname","Weiqin");  
        client.put("lastname","Sun");  
        client.put("phone","56781234");  
        context.put("client",client);  
        outty=getTemplate("properties.vm");  
    }catch(ParseErrorException ex1){  
        System.out.println("VelocityAdd: parse error for template "+ex1);  
    }catch(ResourceNotFoundException ex2){  
        System.out.println("VelocityAdd: template not found "+ex2);  
    }catch(Exception ex3){  
        System.out.println("VelocityAdd: error "+ex3);  
    }  
    return outty;  
}
```

把 properties.vm 和 PropertiesServlet 例子发布到 helloapp 应用中（参考本书配套光盘的 sourcecode/chaper26/helloapp 目录），访问 <http://localhost:8080/helloapp/properties>，将会看到如图 26-3 所示的网页。

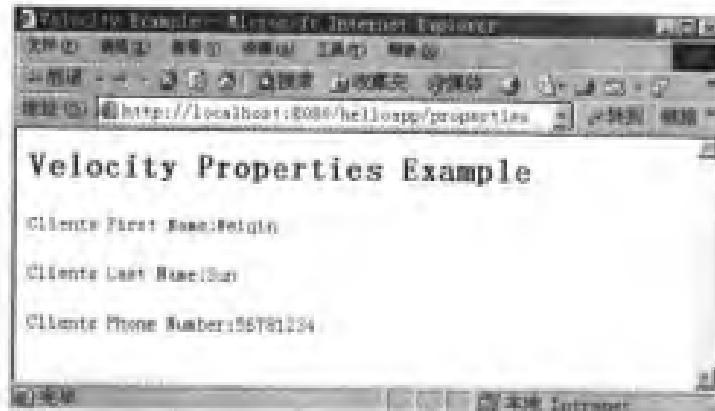


图 26-3 properties.vm 模板生成的网页

给引用属性赋值的第二种办法是定义一个 JavaBean 类，在本例中为 Client.java。在这个类中定义 Client 的各种属性，以及相应的 getXXX 和 setXXX 方法。然后在 Java 代码中创建一个 Client 对象，调用其 setXXX 方法设置各个属性，再把 Client 对象保存在 Context 对象中。下面将创建一个扩展 VelocityServlet 类的 PropertiesServlet\_1 类，以下是它的 handleRequest 方法：

```
public Template handleRequest(HttpServletRequest request, HttpServletResponse response,Context context){

    Template outty=null;
    try{
        Client client=new Client();
        client.setFirstname("WeiQin");
        client.setLastname("Sun");
        client.setPhone("56781234");
        context.put("client",client);
        outty=getTemplate("properties.vm");
    }catch(ParseErrorException ex1){
        System.out.println("VelocityAdd: parse error for template "+ex1);
    }catch(ResourceNotFoundException ex2){
        System.out.println("VelocityAdd: template not found "+ex2);
    }catch(Exception ex3){
        System.out.println("VelocityAdd: error "+ex3);
    }
    return outty;
}
```

Client.java 的代码如下：

```
package mypack;

public class Client
{
    private String firstname;
    private String lastname;
```

```

private String phone;
public String getFirstname(){
    return firstname;
}
public String getLastname(){
    return lastname;
}
public String getPhone(){
    return phone;
}
public void setFirstname(String firstname){
    this.firstname=firstname;
}
public void setLastname(String lastname){
    this.lastname=lastname;
}
public void setPhone(String phone){
    this.phone=phone;
}
}

```

在 Client 类中，firstname、lastname 和 phone 属性都被定义为 private 类型，因此 Velocity 引擎解析模板中的 \$client.firstname 时，不可能直接访问 Client 对象的 firstname 属性，而是调用 Client 对象的 getFirstname 方法来获取 firstname 属性。

把 Client 类和 PropertiesServlet\_1 类发布到 helloapp 应用中（参考本书配套光盘的 sourcecode/chaper26/helloapp 目录），访问 [http://localhost:8080/helloapp/properties\\_1](http://localhost:8080/helloapp/properties_1)，显示结果如上图 26-3 所示。

### 26.4.3 方法引用

方法在 Java 程序代码中定义，VTL 中方法引用的简略标记为前导符“\$”后跟一个 VTL 标识符，再跟一个 VTL 方法体（Method Body）。VTL 方法体由一个 VTL 标识符后跟一个左括号，再跟可选的参数列表，最后是右括号。下面是一些有效的方法示例：

```

$customer.getAddress()
$purchase.getTotal()
$page.setTitle( "My Home Page" )
$person.setAttributes( [ "Strange", "Weird", "Excited" ] )

```

对于上一节介绍的 properties.vm 例子，假定 \$client 代表一个 Client JavaBean 对象，可以在模板中访问 \$client 的各种方法：

```

<html>
<head> <title>Velocity Example</title></head>
<body>
<h1>Velocity Properties Example</h1>
<p>Clients First Name:$client.getFirstname()</p>
<p>Clients Last Name:$client.getLastname()</p>

```

```
<p>Clients Phone Number:$client.getPhone()</p>
</body>
</btml>
```

#### 26.4.4 正式引用符

引用的简略符号如上所述，另外还有一种正式引用符（Formal Reference Notation），示例如下：

```
 ${mudSlinger}
 ${customer.phone}
 ${purchase.getTotal()}
```

在大多数情况下，将使用引用的简略符号，但在一些特殊情况下，需要采用正式引用符来区分引用和普通的字符串。例如：

Jack is a \$vicemaniac.

以上代码存在不确定性：Velocity 把\$vicemaniac（而不是 \$vice）作为整个变量。如果实际引用的变量为\$vice，可使用正式引用符来解决这个问题：

Jack is a \${vice}maniac

这样 Velocity 知道 \$vice（而不是 \$vicemaniac）是一个引用。正式引用符常用在引用变量和文本直接邻近的地方。

#### 26.4.5 安静引用符

当 Velocity 遇到一个未赋值的引用时，会输出这个引用的映像。例如下面是一个表单中的文本框：

```
<input type="text" name="email" value="$email"/>
```

当表单初次装入时，引用变量\$email 无值，Velocity 将在文本框中直接显示“\$email”字符串。在实际应用中，希望在文本框中显示一个空白域而不是“\$email”字符串。使用安静引用符（Quiet Reference Notation）可以绕过 Velocity 的常规行为，达到希望的效果。

安静引用符的前导字符为“\$!”，例如：

```
<input type="text" name="email" value="$!email"/>
```

这样当表单初次装入时，尽管引用变量\$email 仍然没有值，但是在文本框中将显示空字符串。

正式引用符和安静引用符可以一起使用，如下所示：

```
<input type="text" name="email" value="$!{email}">
```

#### 26.4.6 转义符

VTL 中的“\$”具有特殊的含义，如果希望把“\$”符号作为普通的字符来处理，应该采用“\\$”形式，其中“\”为转义符。

例如：

```
#set( $email = "foo" )
```

```
$email
\$email
\\$email
\\\$email
```

以上代码的输出结果为：

```
foo
$email
\foo
\$email
```

对于 “\\\$email”，Velocity 把 “\\” 解析为 “\” 字符。对于 “\\\\$email”，Velocity 先把开头的 “\\” 解析为 “\” 字符，再把 “\\$” 解析为 “\$” 字符。

#### 26.4.7 大小写替换

Velocity 借鉴了 Java Bean 的特征，来解决引用名在上下文中既可作为对象也可作为对象方法的问题。例如以下代码：

```
$client.getFirstname()
## is the same as
$client.firstname

$data.getRequest().getServerName()
## is the same as
$data.Request.ServerName
## is the same as
${data.Request.ServerName}
```

Velocity 可以捕捉和纠正代码中可能出现的大小写错误。例如假定用户实际上想调用的方法为 `getFirstName()`，他在模板中给出的引用为 `$client.firstname`，Velocity 首先尝试 `Client` 实例的 `getfirstname()` 方法，如果失败，再尝试 `getFirstname()` 方法。类似地，当一个模板引用 `$client.Firstname`，Velocity 将先尝试 `getFirstname()` 方法，如果失败，再尝试 `getfirstname()` 方法。



只有当 `Client` 类中提供了 `getfirstname()` 或 `getFirstname()` 方法，`$client.firstname` 才能被解析为 `getfirstname()` 或 `getFirstname()` 方法。否则，即使在类 `Client` 中定义了 `public` 类型的 `firstname` 实例变量，也不能通过 `$client.firstname` 来访问。

## 26.5 指令

模板设计员可以通过引用来输出动态网页内容，此外，还可以采用指令（一种方便的脚本元素）来灵活地控制站点的外观和内容。

### 26.5.1 #set 指令

#set 指令用来为引用变量或引用属性赋值。例如：

```
#set( $primate = "monkey" )  
#set( $customer.Behavior = $primate )
```

赋值表达式的左边必须是一个变量引用或者属性引用。右边可以是下面的类型之一：

- 变量引用
- 字符串
- 属性引用
- 方法引用
- 数字
- 数组列表

以下代码演示了上述的每种赋值类型：

```
#set( $monkey = $bill ) ## variable reference  
#set( $monkey.Friend = "monica" ) ## string literal  
#set( $monkey.Blame = $whitehouse.Leak ) ## property reference  
#set( $monkey.Plan = $spindoctor.weave($web) ) ## method reference  
#set( $monkey.Number = 123 ) ## number literal  
#set( $monkey.Say = ["Not", $my, "fault"] ) ## ArrayList
```



在最后一个例子中，\$monkey.Say 为 ArrayList 类型，通过 \$monkey.Say.get(0)方法可以访问数组的第一个元素。

赋值表达式的右边也可以是一个简单的算术表达式，例如：

```
#set( $value = $foo + 1 )  
#set( $value = $bar - 1 )  
#set( $value = $foo * $bar )  
#set( $value = $foo / $bar )
```

如果赋值表达式的右边是一个属性或方法引用，并且取值是 null，Velocity 将不会把它赋值给左边的引用变量。在这种机制下，给一个已经赋值的引用变量重新赋值可能会失败。这是使用 Velocity 的新手常犯的错误。例如：

```
#set( $result = $query.criteria("name") )  
Name is $result  
#set( $result = $query.criteria("address") )  
Address is $result
```

如果 \$query.criteria("name") 返回字符串“Linda”，\$query.criteria("address") 返回“Shanghai”，上述代码将输出正常结果：

Name is Linda

Address is Shanghai

如果 \$query.criteria("name") 返回字符串“Linda”，而 \$query.criteria("address") 返回 null，

上述代码的输出结果将不符合逻辑：

```
Name is Linda
```

```
Address is Linda
```

对此的解决方法是预设 \$result 为 false。这样如果 \$query.criteria() 调用失败，就可以对其进行检查：

```
#set( $criteria = ["name", "address"] )
#foreach( $criterion in $criteria )

    #set( $result = false )
    #set( $result = $query.criteria($criterion) )

    #if( $result )
        Query was successful.
        $criterion is $result
    #else
        Query was unsuccessful.
        $criterion is unknown.
    #end

#end
```

上述代码中使用了 #if 指令和 #foreach 循环指令，它们的用法可以分别参考本章的 26.5.3 和 26.5.5 节。

## 26.5.2 字面字符串

当使用 #set 指令时，在双引号中的字面字符串（String Literal）将被解析，例如：

```
#set( $directoryRoot = "www" )
#set( $templateName = "index.vm" )
#set( $template = "$directoryRoot/$templateName" )
$template
```

以上代码输出结果为： www/index.vm。

当字面字符串括在单引号中时，将不被解析，例如：

```
#set( $foo = "bar" )
$foo
#set( $blargh = '$foo' )
$blargh
```

以上代码输出结果为：

```
Bar
$foo
```

## 26.5.3 #if 指令

当 #if 指令中的 IF 条件为真时，Velocity 将输出 #if 代码块包含的文本。例如：

```
#if( $foo )
    <strong>Velocity!</strong>
#end
```

Velocity 首先对变量 \$foo 求值，以决定 if 条件是否为真。在以下两种情况下 if 条件为真：

- \$foo 是一个逻辑类型变量，并且值为 true
- \$foo 的值非空

如果 IF 条件为真，Velocity 将输出 #if 和 #end 语句之间的内容。在这种情况下，以上代码的输出将是“Velocity!”。

在以下两种情况下 if 条件为假：

- \$foo 是一个逻辑类型变量，并且值为 false
- \$foo 的值为 null

如果 if 条件为假，Velocity 将不输出 #if 和 #end 语句之间的内容。在这种情况下，以上代码没有输出结果。

在 #if 语句中还可以包含 #elseif 和 #else 项。Velocity 引擎将在遇到第一个为真的表达式时停止逻辑判断。在下面的例子中，假设 \$foo 具有值 15，\$bar 具有值 6：

```
#if( $foo < 10 )
    <strong>Go North</strong>
#elseif( $foo == 10 )
    <strong>Go East</strong>
#elseif( $bar == 6 )
    <strong>Go South</strong>
#else
    <strong>Go West</strong>
#end
```

在以上代码中，\$foo 大于 10，所以前面两个比较失败。接下来判断 “\$bar == 6” 逻辑表达式，结果为真，所以输出结果为 Go South。

#### 26.5.4 比较运算

在 if 条件表达式中，Velocity 支持 3 种变量类型的比较运算：字符串比较、对象比较和整数比较。

##### 1. 字符串比较

字符串比较使用等于操作符 “==” 来决定两个字符串的内容是否相同，例如：

```
#set ($country = "China")
#if($country == "China")
    Chinese people
#else
    foreign people
#end
```

以上代码的输出结果为：Chinese people。

## 2. 对象比较

对象比较使用等于操作符“==”来比较对象，例如：

```
#if($client1==$client2)
```

值得注意的是，只有当等号两边的引用变量引用同一个对象时，才为 true。对于以上表达式，要求 \$client1 和 \$client2 都指向同一个 Client 对象。否则即使两个对象的属性相同，但不是同一个对象，比较结果仍为 false。例如：如果 \$client1 和 \$client2 分别代表两个不同的 Client 对象，这两个 Client 对象具有相同的 name 属性，此时比较结果仍为 false。

## 3. 整数比较

当进行数值比较时，Velocity 目前只支持整数类型数据的比较。以下是整数比较的例子：

```
#if($a==10)
#if($a>10)
#if($a<10)
```

### 26.5.5 #foreach 循环指令

#foreach 指令用来构成循环代码。以下是一个演示#foreach 指令的模板文件，名为 loop.vm：

```
<html>
<head> <title>Velocity Example</title></head>
<body>
<h1>Velocity Loop Example</h1>

<table border=1>
<tr>
<td>First Name</td>
<td>Last Name</td>
<td>Phone</td>
</tr>

#foreach($client in $clientlist)

<tr>
<td>$client.firstname</td>
<td>$client.lastname</td>
<td>$client.phone</td>
</tr>

#end

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

以上#foreach 循环将遍历\$clientlist 列表中的所有 Client 对象。每经过一次循环，将从 \$clientlist 列表中取得一个对象，把它赋值给\$client 变量。

\$clientlist 变量的类型可以是 Vector、Hashtable 或者数组。假定\$clientlist 变量为 Hashtable 类型，我们将在扩展 VelocityServlet 的 LoopServlet 类中为\$clientlist 赋值。以下是 LoopServlet 类的 handleRequest 方法：

```
public Template handleRequest(HttpServletRequest request, HttpServletResponse response, Context context){  
  
    Template outty=null;  
  
    try{  
        Hashtable clientlist=new Hashtable();  
  
        Client client=new Client();  
        client.setFirstname("Xiaowen");  
        client.setLastname("Li");  
        client.setPhone("56781234");  
        clientlist.put(client.getFirstname(),client);  
  
        client=new Client();  
        client.setFirstname("Xiaowei");  
        client.setLastname("Cao");  
        client.setPhone("56782345");  
        clientlist.put(client.getFirstname(),client);  
  
        client=new Client();  
        client.setFirstname("Xiaojie");  
        client.setLastname("Sun");  
        client.setPhone("56783456");  
        clientlist.put(client.getFirstname(),client);  
  
        context.put("clientlist",clientlist);  
  
        outty=getTemplate("loop.vm");  
    }catch(ParseErrorException ex1){  
        System.out.println("Velocity Add: parse error for template "+ex1);  
    }catch(ResourceNotFoundException ex2){  
        System.out.println("Velocity Add: template not found "+ex2);  
    }catch(Exception ex3){  
        System.out.println("Velocity Add: error "+ex3);  
    }  
    return outty;  
}
```

把 loop.vm 和 LoopServlet 类发布到 helloapp 应用中（参考本书配套光盘的 sourcecode/chaper26/helloapp 目录），访问 <http://localhost:8080/helloapp/loop>，显示结果如

图 26-4 所示。

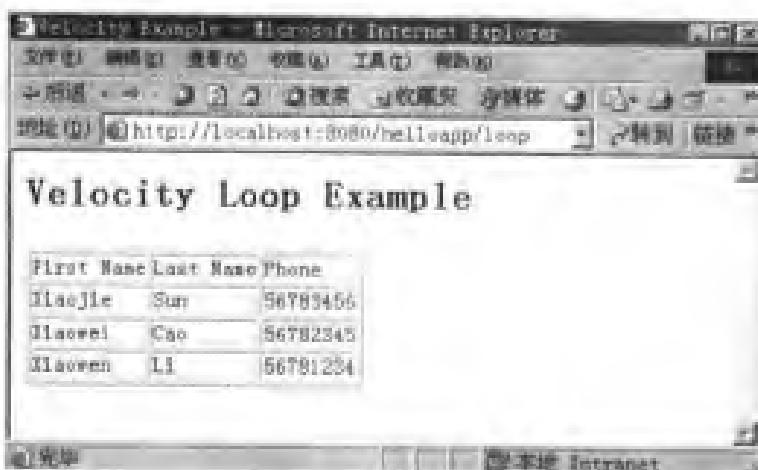


图 26-4 loop.vm 生成的网页

### 26.5.6 #include 指令

#include 指令用来导入本地文件，这些文件将插入到模板中#include 指令被定义的地方。例如：

```
#include( "one.txt" )
```

#include 指令引用的文件名放在双引号内，如果超过一个文件，其间用逗号隔开，例如：

```
#include( "one.gif", "two.txt", "three.htm" )
```

被包含的文件并不一定要直接给出文件名，事实上，最好的办法是使用变量而不是文件名。这在需要根据特定逻辑来决定导入相应文件的情况下很有用，例如：

```
#if($client.balance>1000)
#set($page="wealthy.htm")
#else
#set($page="value.htm")
#end
#include( "head.htm", $page, "footer.htm" )
```

### 26.5.7 #parse 指令

#parse 指令和#include 指令很相似，两者都可以把其他文件导入到当前模板中。区别在于，#parse 指令能够解析被导入的文件。此外，单个#parse 指令只允许导入一个文件。与#include 指令一样，#parse 指令也允许文件名用变量表示。以下是#parse 指令的例子：

```
#parse( "head.vm" ) ##load and parse head.vm file
#parse($mypage) ##load and parse the file specified by $mypage
```

### 26.5.8 #macro 指令

#macro 指令允许模板设计者在 VTL 模板中定义重复的段，称之为 Velocity 宏。Velocity

宏不管是在复杂还是简单的场合都非常有用。把模板中重复的代码定义在一个 Velocity 宏中，在模板中所有出现重复代码的地方都可以用宏来代替，这样可以使模板更加简洁，易于维护。例如，以下是一个宏的定义：

```
#macro( macroname )
<tr><td></td></tr>
#end
```

在以上例子中，Velocity 宏被命名为“macroname”，在 Velocity 模板的其他地方调用宏的形式为：#macroname()。

当模板被调用时，Velocity 将把#macroname()替换为：<tr><td></td></tr>。

Velocity 宏可以不带参数（如上例所示），也可以带一些参数。当宏被调用时，所带的参数必须与其定义时的参数一样。下面这个宏带有两个参数：

```
#macro( tablerows $color $somelist )
#foreach( $something in $somelist )
<tr><td bgcolor=$color>$something</td></tr>
#end
#end
```

以上例子定义的宏名为 tablerows，它有两个参数：第一个参数为 \$color，第二个为 \$somelist。

所有合法的 VTL 模板的内容都可以作为 Velocity 宏的主体部分。#tablerows 宏包含了一个 foreach 语句。在#tablerows 宏的定义中有两个#end 语句，第一个属于#foreach，第二个结束宏定义。

下面是调用宏的代码：

```
#set( $greatlakes = ["Superior","Michigan","Huron","Erie","Ontario"] )
#set( $color = "blue" )





```

请注意\$greatlakes 替换了\$somelist 参数。当#tablerows 宏被调用时，将产生以下输出：

```
<table>
<tr><td bgcolor="blue">Superior</td></tr>
<tr><td bgcolor="blue">Michigan</td></tr>
<tr><td bgcolor="blue">Huron</td></tr>
<tr><td bgcolor="blue">Erie</td></tr>
<tr><td bgcolor="blue">Ontario</td></tr>

```

Velocity 宏的参数可以是以下的 VTL 元素：

- 引用 (Reference): 以“\$”打头的元素
- 字面字符串 (String literal): 比如 “\$foo” 或 “hello”
- 字面数字: 1, 2 .....
- 整数范围: [ 1..2] 或 [\$foo .. \$bar]
- 对象数组: [ "a", "b", "c" ]
- 布尔真: true

- 布尔假: false

把引用作为参数传递给 Velocity 宏时, 请注意引用是按“名字”传递的。这意味着它们的值在每次使用时才产生。例如:

```
#macro( callme $a )
$a $a $a
#end
#callme( $foo.bar() )
```

以上代码把\$foo.bar()方法引用作为参数传给宏, bar()方法将被调用 3 次。

### 26.5.9 转义 VTL 指令

VTL 采用反斜杠 (“\”) 来进行符号转义, 例如 “\#” 将被 Velocity 解析为普通的字符 “#”。首先看一个没有进行转义的例子:

```
\#if( $jazz )
Vyacheslav Ganelin
#end
```

对于以上代码, 如果 \$jazz 为 true, 输出为: Vyacheslav Ganelin ; 如果 \$jazz 为 false, 则没有输出。

如果对#if 指令进行转义, 将改变输出结果, 例如:

```
\#if( $jazz )
Vyacheslav Ganelin
\#end
```

对于以上代码, “\#if” 将被 Velocity 解析为普通的字符串 “#if”。因此不管 \$jazz 是真或假, 输出结果都为:

```
\#if($ jazz )
Vyacheslav Ganelin
#end
```

事实上, 因为所有指令都被转义了, \$jazz 永远不会被求值。

### 26.5.10 VTL 的格式

当 Velocity 解析 VTL 代码时, 其行为不受代码中的换行和空格的影响。例如:

```
Send me #set($foo = ["$10 and ","a cake"])\#foreach($a in $foo)$a #end please.
```

上述代码也可以写成:

```
Send me
#set( $foo =["$10 and ","a cake"] )
#foreach( $a in $foo )
$a
#end
please.
```

或者

```
Send me
```

```
#set($foo      = ["$10 and ","a cake"])
#foreach ($a in $foo )$a
#end please.
```

上面 3 种写法的输出结果都一样。

## 26.6 其他特征

### 26.6.1 数学运算

Velocity 有一些内置的数学功能，下面的代码分别演示了加减乘除运算：

```
#set( $foo = $bar + 3 )
#set( $foo = $bar - 4 )
#set( $foo = $bar * 6 )
#set( $foo = $bar / 2 )
```

当进行除法运算时，得到的结果是整数。余数可以通过模运算符“%”获得，例如：

```
#set( $foo = $bar % 5 )
```

在 Velocity 中，只有整数可以进行数学运算；如果执行非整数的数学运算，该操作将被记录到日志中，并返回 null。

### 26.6.2 范围操作符

范围操作符 (Range Operator) 可以定义包含 Integer 对象的数组，它常和#set 或#foreach 语句一起使用。范围操作符的形式为：[n .. m]。

n 和 m 都必须是整数。m 大于或者小于 n 都没关系；在 m 小于 n 的情况下，数组下标从大到小计数。下面是使用范围操作符的例子。

第一个例子：

```
#foreach( $foo in [1..5] )
$foo
#end
```

第二个例子：

```
#foreach( $bar in [2..-2] )
$bar
#end
```

第三个例子：

```
#set( $arr = [0..1] )
#foreach( $i in $arr )
$i
#end
```

第四个例子：

```
[1..3]
```

以上例子的输出结果分别为：

第一个例子: 1 2 3 4 5

第二个例子: 2 1 0 -1 -2

第三个例子: 0 1

第四个例子: [1..3]

根据第四个例子的输出结果可以看出, 范围操作符只有和#set 或#foreach 指令一起使用时, 才代表 Integer 对象数组, 否则它将被解析为普通的字符串。

### 26.6.3 字符串的连接

Velocity 开发者常问的一个问题是“我如何把多个字符串连起来?” “是否有类似于 Java 中的“+”操作符?”, 答案是否定的。

在 VTL 中, 如果要串联字符串, 只需要把这些字符串“放在一起”, 例如:

```
#set($size = "Big")
#set($name = "Ben")
The clock is $size$name.
```

以上代码的输出为: The clock is BigBen.

如果想把几个字符串串联后再传递给一个方法, 或者赋值给一个引用变量, 可以采用以下方法:

```
#set($size = "Big")
#set($name = "Ben")
#set($clock = "{$size}{$name}")
The clock is $clock.
```

以上代码的输出结果与上一个例子是一样的。再看最后一个例子, 若想把“静态”字符串和引用混合在一起, 可能需要使用正式引用符号, 例如:

```
#set($size = "Big")
#set($name = "Ben")
#set($clock = "{$size}Tall{$name}")
The clock is $clock.
```

以上代码的输出为: The clock is BigTallBen.

## 26.7 小结

Velocity 是一个基于 Java 的模板引擎。Velocity 引擎能够把模板和上下文合并起来, 生成动态网页。创建基于 Velocity 的 Web 应用包括两个步骤: 第一步, 创建 Velocity 模板; 第二步, 创建扩展 VelocityServlet 的 Servlet 类。模板设计员可以在模板中通过引用来输出动态网页内容, 还可以采用指令(一种方便的脚本元素)来灵活地控制站点的外观和内容。VTL 中有 3 种类型的引用: 变量、属性和方法。Velocity 常用的指令包括: 赋值指令#set、条件指令#if、循环指令#foreach、文件包含指令#include、文件解析指令#parse 和宏指令#macro。

# 附录 A server.xml 文件

Tomcat 服务器是由一系列可配置的组件构成，Tomcat 的组件可以在 <CATALINA\_HOME>\conf\server.xml 文件中进行配置，每个 Tomcat 组件和 server.xml 文件中的一种配置元素对应。

本附录对一些常用的元素做了介绍。关于 server.xml 的更多信息，可以参考 Tomcat 的文档：

<CATALINA\_HOME>/webapps/tomcat-docs/config/index.html

下面首先看一个 server.xml 文件的样例：

```
<!-- Example Server Configuration File -->
```

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">

    <Service name="Catalina">

        <Connector port="8080"
                   maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
                   enableLookups="false" redirectPort="8443" acceptCount="100"
                   debug="0" connectionTimeout="20000"
                   disableUploadTimeout="true" />

        <Engine name="Catalina" defaultHost="localhost" debug="0" >

            <Logger className="org.apache.catalina.logger.FileLogger"
                   prefix="catalina_log." suffix=".txt"
                   timestamp="true"/>

            <Realm className="org.apache.catalina.realm.MemoryRealm" />

            <Host name="localhost" debug="0" appBase="webapps"
                  unpackWARs="true" autoDeploy="true">

                <Valve className="org.apache.catalina.valves.AccessLogValve"
                      directory="logs"   prefix="localhost_access_log." suffix=".txt"
                      pattern="common" resolveHosts="false"/>

                <Logger className="org.apache.catalina.logger.FileLogger"
                      directory="logs"   prefix="localhost_log." suffix=".txt"
                      timestamp="true"/>

            <Context path="/sample" docBase="sample" debug="0"
```

```
reloadable="true" >

<Resource name="jdbc/BookDB"
auth="Container"
type="javax.sql.DataSource" />

<ResourceParams name="jdbc/BookDB">
<parameter>
<name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>

<parameter>
<name>maxActive</name>
<value>100</value>
</parameter>

<parameter>
<name>maxIdle</name>
<value>30</value>
</parameter>

<parameter>
<name>maxWait</name>
<value>10000</value>
</parameter>

<parameter>
<name>username</name>
<value>dbuser</value>
</parameter>
<parameter>
<name>password</name>
<value>1234</value>
</parameter>

<parameter>
<name>driverClassName</name>
<value>com.mysql.jdbc.Driver</value>
</parameter>

<parameter>
<name>url</name>
<value>jdbc:mysql://localhost:3306/BookDB?autoReconnect=true</value>
</parameter>
```

```
</ResourceParams>

</Context>

</Host>

</Engine>

</Service>

<Service name="Apache">

    <Connector port="8009"
               enableLookups="false" redirectPort="8443" debug="0"
               protocol="AJP/1.3" />

    <Engine name="Apache" defaultHost="localhost" debug="0" >

        <Logger className="org.apache.catalina.logger.FileLogger"
               prefix="apache_log." suffix=".txt"
               timestamp="true"/>

        <Realm className="org.apache.catalina.realm.MemoryRealm" />

    </Engine>

</Service>

</Server>
```

以上 XML 代码中，每个元素都代表一种 Tomcat 组件。这些元素可分为 4 类。

#### 1. 顶层类元素

顶层类元素包括<Server>元素和<Service>元素，它们位于整个配置文件的顶层。

#### 2. 连接器类元素

连接器类元素代表了介于客户与服务之间的通信接口，负责将客户的请求发送给服务器，并将服务器的响应结果传递给客户。

#### 3. 容器类元素

容器类元素代表处理客户请求并生成响应结果的组件，有 3 种容器类元素，它们是 Engine、Host 和 Context。Engine 组件为特定的 Service 组件处理所有客户请求，Host 组件为特定的虚拟主机处理所有客户请求，Context 组件为特定的 Web 应用处理所有客户请求。

#### 4. 嵌套类元素

嵌套类元素代表了可以加入到容器中的组件，如<Logger>元素、<Valve>元素和<Realm>元素。

下面，对基本的 Tomcat 元素逐一介绍。

## A.1 配置 Server 元素

<Server>元素代表整个 Catalina Servlet 容器，它是 Tomcat 实例的顶层元素，由 org.apache.catalina.Server 接口来定义。<Server>元素中可包含一个或多个<Service>元素，但<Server>元素不能作为任何其他元素的子元素。在本章样例中定义了以下<Server>元素：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
<Server>元素的属性描述参见表 A-1。
```

表 A-1 <Server>元素的属性

描 述	属 性
className	指定实现 org.apache.catalina.Server 接口的类，默认值为 org.apache.catalina.core.StandardServer
port	指定 Tomcat 服务器监听 shutdown 命令的端口。终止 Tomcat 服务器运行时，必须在 Tomcat 服务器所在的机器上发出 shutdown 命令。该属性是必须设定的
shutdown	指定终止 Tomcat 服务器运行时，发给 Tomcat 服务器的 shutdown 监听端口的字符串。该属性是必须设定的

## A.2 配置 Service 元素

<Service>元素由 org.apache.catalina.Service 接口定义，它包含一个<Engine>元素，以及一个或多个<Connector>元素，这些<Connector>元素共享同一个<Engine>元素。例如，在样例 server.xml 文件中配置了两个<Service>元素：

```
<Service name="Catalina">
<Service name="Apache">
```

第一个<Service>处理所有直接由 Tomcat 服务器接收的 Web 客户请求，第二个<Service>处理由 Apache 服务器转发过来的 Web 客户请求。

<Service>元素的属性描述参见表 A-2。

表 A-2 <Service>元素的属性

描 述	属 性
className	指定实现 org.apache.catalina.Service 接口的类，默认值为 org.apache.catalina.core.StandardService
name	定义 Service 的名字

## A.3 配置 Engine 元素

<Engine>元素由 org.apache.catalina.Engine 接口定义。每个<Service>元素只能包含一个<Engine>元素。<Engine>元素处理在同一个<Service>中所有<Connector>元素接收到的客户

请求。例如，在样例 server.xml 文件中配置了以下<Engine>元素：

```
<Engine name="Catalina" defaultHost="localhost" debug="0" >
```

<Engine>元素的属性描述参见表 A-3。

表 A-3 <Engine>元素的属性

描 述	属 性
className	指定实现 org.apache.catalina.Engine 接口的类，默认值为 org.apache.catalina.core.StandardEngine
defaultHost	指定处理客户请求的默认主机名，在<Engine>的<Host>子元素中必须定义这 一主机
name	定义 Engine 的名字

在<Engine>元素中可以包含如下子元素：

- <Logger>
- <Realm>
- <Valve>
- <Host>

## A.4 配置 Host 元素

<Host>元素由 org.apache.catalina.Host 接口定义。一个<Engine>元素中可以包含多个<Host>元素。每个<Host>元素定义了一个虚拟主机，它可以包含一个或多个 Web 应用。

例如，在样例 server.xml 文件中配置了以下<Host>元素：

```
<Host name="localhost" debug="0" appBase="webapps"
      unpackWARs="true" autoDeploy="true">
```

以上代码定义了一个名为 localhost 的虚拟主机，Web 客户访问它的 URL 为：

http://localhost:8080/

<Host>元素的属性描述参见表 A-4。

表 A-4 <Host>元素的属性

描 述	属 性
className	指定实现 org.apache.catalina.Host 接口的类，默认值为 org.apache.catalina.core.StandardHost
appBase	指定虚拟主机的目录，可以指定绝对目录，也可以指定相对于<CATALINA_HOME>的相对目录。如果此项没有设定，默认值为<CATALINA_HOME>/webapps
unpackWARs	如果此项设为 true，表示将把 Web 应用的 WAR 文件先展开为开放目录结构后再运行。如果设为 false，将直接运行 WAR 文件
autoDeploy	如果此项设为 true，表示当 Tomcat 服务器处于运行状态时，能够监测 appBase 下的文件，如果有新的 Web 应用加入进来，会自动发布这个 Web 应用
alias	指定虚拟主机的别名，可以指定多个别名
deployOnStartup	如果此项设为 true，表示 Tomcat 服务器启动时会自动发布 appBase 目录下所有的 Web 应用，如果 Web 应用在 server.xml 中没有相应的<Context>元素，将采用 Tomcat 默认的 Context。deployOnStartup 的默认值为 true
name	定义虚拟主机的名字

在<Host>元素中可以包含如下子元素：

- <Logger>
- <Realm>
- <Valve>
- <Context>

## A.5 配置 Context 元素

<Context>元素由 org.apache.catalina.Context 接口定义。<Context>元素是使用最频繁的元素。每个<Context>元素代表了运行在虚拟主机上的单个 Web 应用。一个<Host>元素中可以包含多个<Context>元素。

例如，在样例 server.xml 文件中配置了以下<Context>元素：

```
<Context path="/sample" docBase="sample" debug="0"
        reloadable="true" >
```

<Context>元素的属性描述参见表 A-5。

表 A-5 <Context>元素的属性

描 述	属 性
className	指定实现 org.apache.catalina.Context 接口的类，默认值为 org.apache.catalina.core.StandardContext
path	指定访问该 Web 应用的 URL 入口
docBase	指定 Web 应用的文件路径。可以给定绝对路径，也可以给定相对于 Host 的 appBase 属性的相对路径。如果 Web 应用采用开放目录结构，那就指定 Web 应用的根目录；如果 Web 应用是个 WAR 文件，那就指定 WAR 文件的路径
reloadable	如果这个属性设为 true，Tomcat 服务器在运行状态下会监视在 WEB-INF/classes 和 WEB-INF/lib 目录下 CLASS 文件的改动。如果监测到有 class 文件被更新，服务器会自动重新加载 Web 应用
cookies	指定是否通过 Cookie 来支持 Session，默认值为 true
useNaming	指定是否支持 JNDI，默认值为 true

在<Context>元素中可以包含如下子元素：

- <Logger>
- <Realm>
- <Valve>
- <Resource>
- <ResourceParams>

## A.6 配置 Connector 元素

<Connector>元素由 org.apache.catalina.Connector 接口定义。<Connector>元素代表与客户程序实际交互的组件，它负责接收客户请求，以及向客户返回响应结果。

例如，在样例 server.xml 文件中配置了两个<Connector>元素：

```
<Connector port="8080"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    debug="0" connectionTimeout="20000"
    disableUploadTimeout="true" />

<Connector port="8009"
    enableLookups="false" redirectPort="8443" debug="0"
    protocol="AJP/1.3" />
```

第一个<Connector>元素定义了一个 HTTP Connector，它通过 8080 端口接收 HTTP 请求；第二个<Connector>元素定义了一个 JK Connector，它通过 8009 端口接收由其他 HTTP 服务器（如 Apache 服务器）转发过来的客户请求。

所有的<Connector>元素都具有一些共同的属性，这些属性描述参见表 A-6。

表 A-6 <Connector>元素的共同属性

描 述	属 性
className	指定实现 org.apache.catalina.Connector 接口的类
enableLookups	如果设为 true，表示支持域名解析，可以把 IP 地址解析为主机名。Web 应用中调用 request.getRemoteHost 方法将返回客户的主机名。该属性的默认值为 true
redirectPort	指定转发端口。如果当前端口只支持 non-SSL 请求，在需要安全通信的场合，将把客户请求转发到基于 SSL 的 redirectPort 端口

HttpConnector 的属性描述参见表 A-7。

表 A-7 HttpConnector 元素的属性

描 述	属 性
className	指定实现 org.apache.catalina.Connector 接口的类，默认值为 org.apache.coyote.tomcat5 .CoyoteConnector
enableLookups	参见表 A-6
redirectPort	参见表 A-6
port	设定 TCP/IP 端口号，默认值为 8080
address	如果服务器有两个以上 IP 地址，该属性可以设定端口监听的 IP 地址，默认情况下，端口会监听服务器上所有 IP 地址
bufferSize	设定由端口创建的输入流的缓存大小，默认值为 2048byte
protocol	设定 HTTP 协议，默认值为 HTTP/1.1
maxThreads	设定处理客户请求的线程的最大数目，这个值也决定了服务器可以同时响应客户请求的最大数目，默认值为 200
acceptCount	设定在监听端口队列中的最大客户请求数，默认值为 10。如果队列已满，客户请求将被拒绝
connectionTimeout	定义建立客户连接超时的时间，以毫秒为单位。如果设置为 -1，表示不限制建立客户连接的时间

JK Connector 的属性描述参见表 A-8。

表 A-8 JK Connector 的属性

描述	属性
className	指定实现 org.apache.catalina.Connector 接口的类，默认值为 org.apache.coyote.tomcat5.CoyoteConnector
enableLookups	参见表 A-6
redirectPort	参见表 A-6
port	设定 AJP 端口号
protocol	必须设定为 AJP/1.3 协议

## 附录 B web.xml 文件

Web 应用发布描述符文件（即 web.xml 文件）是在 Servlet 规范中定义的。它是 Web 应用的配置文件。web.xml 中的元素和 Tomcat 容器完全独立。例程 B-1 是一个 web.xml 的样例，在后面的讲解中都会用到这个样例。

例程 B-1 web.xml 的样例

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <display-name>Sample Application</display-name>

    <description>
        This is a sample application
    </description>

    <filter>
        <filter-name>SampleFilter</filter-name>
        <filter-class>mypack.SampleFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>SampleFilter</filter-name>
        <url-pattern>*.jsp</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name> SampleServlet </servlet-name>
        <servlet-class>mypack.SampleServlet</servlet-class>
        <init-param>
            <param-name>initParam1</param-name>
            <param-value>2</param-value>
        </init-param>
        <load-on-startup> 1 </load-on-startup>
    </servlet>

```

```
<!-- Define the SampleServlet Mapping -->
<servlet-mapping>
    <servlet-name>SampleServlet</servlet-name>
    <url-pattern>/sample</url-pattern>
</servlet-mapping>

<session-config>
    <session-timeout>30</session-timeout>
</session-config>

<welcome-file-list>
    <welcome-file>login.jsp </welcome-file>
    <welcome-file>index.htm </welcome-file>
</welcome-file-list>

<taglib>
    <taglib-uri>/mytaglib</taglib-uri>
    <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
</taglib>

<resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/sampleDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>

<!-- Define a Security Constraint on this Application -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>sample application</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>guest</role-name>
    </auth-constraint>
</security-constraint>

<!-- Define the Login Configuration for this Application -->
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Tomcat Server Configuration Form-Based Authentication Area</realm-name>
    <form-login-config>
        <form-login-page>/login.jsp</form-login-page>
        <form-error-page>/error.jsp</form-error-page>
```

```

</form-login-config>
</login-config>

<!-- Security roles referenced by this web application -->
<security-role>
    <description>
        The role that is required to log into the sample Application
    </description>
    <role-name>guest</role-name>
</security-role>

</web-app>

```

以上 web.xml 依次定义了如下元素：

- <web-app>
- <display-name>
- <description>
- <filter>
- <filter-mapping>
- <servlet>
- <servlet-mapping>
- <session-config>
- <welcome-file-list>
- <taglib>
- <resource-ref>
- <security-constraint>
- <login-config>
- <security-role>



在 web.xml 中元素定义的先后顺序不能颠倒，否则 Tomcat 服务器可能会抛出 SAXParseException。

web.xml 中的开头几行往往是固定的，它定义了该文件的字符编码、XML 的版本以及引用的 DTD 文件。在 web.xml 中顶层元素为<web-app>，其他所有的子元素都必须定义在<web-app>内。<display-name>元素定义这个 Web 应用的名字，Java Web 服务器的 Web 管理工具将用这个名字来标志 Web 应用。<description>元素用来声明 Web 应用的描述信息。

下面将介绍几种最常用的元素的配置方法。

## B.1 配置 Servlet 过滤器

对于 Servlet 容器收到的客户请求，以及发出的响应结果，Servlet 过滤器都能检查和修

改其中的信息。在 Web 应用中加入 Servlet 过滤器，需要在 web.xml 中配置两个元素<filter>和<filter-mapping>。以下是<filter>元素的示范代码：

```
<filter>
    <filter-name>SampleFilter</filter-name>
    <filter-class>mypack.SampleFilter</filter-class>
</filter>
```

以上代码定义了一个过滤器，名为 SampleFilter，实现这个过滤器的类是 mypack.SampleFilter 类。<filter>元素的属性描述参见表 B-1。

表 B-1 <filter>元素的属性

属性	描述
<filter-name>	定义过滤器的名字。当 Web 应用中有多个过滤器时，不允许过滤器重名
<filter-class>	指定实现这个过滤器的类，这个类负责具体的过滤事务

<filter-mapping>元素用来设定过滤器负责过滤的 URL。以下是<filter-mapping>元素的示范代码：

```
<filter-mapping>
    <filter-name>SampleFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

以上代码指明当客户请求访问 Web 应用中的所有 JSP 文件时，将触发 SampleFilter 过滤器工作。具体的过滤事务由在<filter>元素中指定的 mypack.SampleFilter 类完成。

<filter-mapping>元素的属性描述参见表 B-2。

表 B-2 <filter-mapping>元素的属性

属性	描述
<filter-name>	指定过滤器名，这里的过滤器名必需和<filter>元素中定义的过滤器名匹配
<url-pattern>	指定过滤器负责过滤的 URL

## B.2 配置 Servlet

<servlet>元素用来定义 Servlet，以下代码定义了一个名为 SampleServlet 的 Servlet，实现这个 Servlet 的类是 mypack.SampleServlet：

```
<servlet>
    <servlet-name> SampleServlet </servlet-name>
    <servlet-class>mypack.SampleServlet</servlet-class>
    <init-param>
        <param-name>initParam1</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup> 1 </load-on-startup>
</servlet>
```

<servlet>元素的属性描述参见表 B-3。

表 B-3 &lt;servlet&gt;元素的属性

属性	描述
<servlet-name>	定义 Servlet 的名字
<servlet-class>	指定实现这个 Servlet 的类
<init-param>	定义 Servlet 的初始化参数（包括参数名和参数值）。一个<servlet>元素中可以有多个<init-param>。在 Servlet 类中通过 getInitParameter(String name)方法访问初始化参数
<load-on-startup>	指定当 Web 应用启动时，装载 Servlet 的次序。当这个值为正数或零时，Servlet 容器先加载数值小的 Servlet，再依次加载其他数值大的 Servlet。如果这个值为负数或者没有设定，那么 Servlet 容器将在 Web 客户首次访问这个 Servlet 时加载它

## B.3 配置 Servlet 映射

<servlet-mapping>元素用来设定客户访问某个 Servlet 的 URL。以下代码为 SampleServlet 指定的相对 URL 为“/sample”：

```
<!-- Define the Manager Servlet Mapping -->
<servlet-mapping>
    <servlet-name>SampleServlet</servlet-name>
    <url-pattern>/sample</url-pattern>
</servlet-mapping>
```

<servlet-mapping>使得程序中定义的 Servlet 类名和客户访问的 URL 彼此独立。当 Servlet 类名发生改变，只要修改<servlet>元素中的<servlet-class>属性，而客户端访问的 URL 无须做响应的改动。

<servlet-mapping>元素的属性描述参见表 B-4。

表 B-4 &lt;servlet-mapping&gt;元素的属性

属性	描述
<servlet-name>	指定 Servlet 的名字。这里的 Servlet 名字应该和<servlet>元素中定义的名字匹配
<url-pattern>	指定访问这个 Servlet 的 URL。这里只需给出对于整个 Web 应用的相对 URL 路径

## B.4 配置 Session

<session-config>元素用来设定 HttpSession 的生命周期。例如，以下代码指明，Session 可以保持不活动状态的最长时间为 30 秒，超过这一时间，Servlet 容器将把它作为无效 Session 处理。

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

<session-config>元素只包括一个属性<session-timeout>，它用来设定 Session 可以保持

不活动状态的最长时问，这里采用的时间单位为“秒”。

## B.5 配置 Welcome 文件清单

当客户访问 Web 应用时，如果仅仅给出 Web 应用的 Root URL，没有指定具体的文件名，Servlet 容器会自动调用 Web 应用的 Welcome 文件。`<welcome-file-list>`元素用来设定 Welcome 文件清单。以下代码中声明了两个 Welcome 文件：login.jsp 和 index.htm：

```
<welcome-file-list>
    <welcome-file>login.jsp </welcome-file>
    <welcome-file>index.htm </welcome-file>
</welcome-file-list>
```

`<welcome-file-list>`元素中可以包含多个`<welcome-file>`，当 Servlet 容器调用 Web 应用的 Welcome 文件时，首先寻找第一个`<welcome-file>`指定的文件。如果这个文件存在，那么把这一文件返回给客户；如果这个文件不存在，Servlet 容器将依次寻找下一个 Welcome 文件，直到找到为止；如果`<welcome-file-list>`元素中指定的所有文件都不存在，服务器将向客户端返回“HTTP 404 Not Found”的出错信息。

## B.6 配置 Tag Library

`<taglib>`元素用来设置 Web 应用所引用的 Tag Library。以下代码定义了一个“/mytaglib”标签库，它对应的 TLD 文件为/WEB-INF/mytaglib.tld。

```
<taglib>
    <taglib-uri>/mytaglib</taglib-uri>
    <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
</taglib>
```

`<taglib>`元素中的属性描述参见表 B-5。

表 B-5 `<taglib>`元素的属性

属性	描述
<code>&lt;taglib-uri&gt;</code>	设定 Tag Library 的唯一标识符，在 Web 应用中将根据这一标识符来引用 Tag Library
<code>&lt;taglib-location&gt;</code>	指定和 Tag Library 对应的 TLD 文件的位置

## B.7 配置资源引用

如果 Web 应用访问了由 Servlet 容器管理的某个 JNDI Resource，必须在 web.xml 文件中声明对这个 JNDI Resource 的引用。表示资源引用的元素为`<resource-ref>`，以下是声明引用 jdbc/SampleDB 数据源的代码。

```
<resource-ref>
```

```

<description>DB Connection</description>
<res-ref-name>jdbc/sampleDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>

```

<resource-ref>的属性描述参见表 B-6。

表 B-6 <resource-ref>的属性

属性	描述
description	对所引用的资源的说明
res-ref-name	指定所引用资源的 JNDI 名字
res-type	指定所引用资源的类名字
res-auth	指定管理所引用资源的 Manager, 它有两个可选值: Container 和 Application Container 表示由容器来创建和管理 Resource, Application 表示由 Web 应用来创建和管理 Resource

## B.8 配置安全约束

<security-constraint>用来为 Web 应用定义安全约束。以下代码指明当用户访问该 Web 应用下所有的资源，必须具备 guest 角色。

```

<!-- Define a Security Constraint on this Application --&gt;
&lt;security-constraint&gt;
  &lt;web-resource-collection&gt;
    &lt;web-resource-name&gt;sample application&lt;/web-resource-name&gt;
    &lt;url-pattern&gt;/*&lt;/url-pattern&gt;
  &lt;/web-resource-collection&gt;
  &lt;auth-constraint&gt;
    &lt;role-name&gt;guest&lt;/role-name&gt;
  &lt;/auth-constraint&gt;
&lt;/security-constraint&gt;
</pre>

```

<security-constraint>元素和<web-resource-collection>元素中的属性描述参见表 B-7 和表 B-8。

表 B-7 <security-constraint>元素的属性

属性	描述
<web-resource-collection>	声明受保护的 Web 资源
<auth-constraint>	声明可以访问受保护资源的角色，可以包含多个<role-name>子元素

表 B-8 <web-resource-collection>元素的属性

属性	描述
<web-resource-name>	标识受保护的 Web 资源
<url-pattern>	指定受保护的 URL 路径

## B.9 配置安全验证登录界面

<login-config>元素指定当 Web 客户访问受保护的 Web 资源时，系统弹出的登录对话框的类型。以下代码配置了基于表单验证的登录界面：

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Tomcat Server Configuration Form-Based Authentication Area</realm-name>
    <form-login-config>
        <form-login-page>/login.jsp</form-login-page>
        <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
</login-config>
```

<login-config>元素的各个属性说明参见表 B-9。

表 B-9 <login-config>元素的属性

属性	描述
<auth-method>	指定验证方法。它有 3 个可选值：BASIC（基本验证）、DIGEST（摘要验证）和 FORM（基于表单的验证）
<realm-name>	设定安全域的名称
<form-login-config>	当验证方法为 FORM 时，配置验证网页和出错网页
<form-login-page>	当验证方法为 FORM 时，设定验证网页
<form-error-page>	当验证方法为 FORM 时，设定出错网页

## B.10 配置对安全验证角色的引用

<security-role>元素指明这个 Web 应用引用的所有角色名字。例如，以下代码声明引用了 guest 角色：

```
<!-- Security roles referenced by this web application -->
<security-role>
    <description>
        The role that is required to log in to the sample Application
    </description>
    <role-name>guest</role-name>
</security-role>
```

# 附录 C XML 简介

XML，即可扩展标记语言（Extensible Markup Language），是一种可以用来创建自己标记的标记语言。它是 Internet 环境中跨平台的、依赖于内容的技术，它可以简化 Internet 上的文档信息传输，并且在目前流行的分布式结构中作为系统之间通信的公共语言。XML 是年轻的 meta 语言。早在 1998 年，W3C 就发布了 XML 1.0 规范。内容建设者们已经开始开发各种各样的 XML 应用程序，比如说数学标记语言 MathML、化学标记语言 CML 等。

XML 不仅满足了 Web 开发者的需要，而且适用于任何对出版业感兴趣的人。Oracle、IBM 以及 Microsoft 公司都积极地投入人力与财力研发 XML 相关软件，这无疑确定了 XML 在 IT 产业的美好前景。

## C.1 SGML、HTML 与 XML 的比较

HTML 和 XML 都基于 SGML，即标准通用标记语言（Standard Generalized Markup Language）。SGML 太复杂，HTML 虽然简单但缺乏可扩展性，XML 克服了两者的缺点，所以被广泛地应用于 Web 开发领域。

### 1. 标准通用标记语言 SGML

SGML 是描述电子文档的国际化标准，它是用于书写其他语言的元语言，以逻辑化和机构化的方式描述文本文档，主要用于文档的创建、存储以及分发。

SGML 文档已经被美国军方及美国航空业使用多年，但是对于 Web 工作者来说却显得非常复杂，难以理解，使许多本打算使用它的人望而却步，难怪有人把它翻译为“听起来很棒，但或许以后会用（Sounds great, maybe later）”。SGML 的复杂导致了 HTML 语言（SGML 的一个子集）的成长。

### 2. 超文本标记语言 HTML

HTML 使得 Web 开发变得非常得简单。开发者无需了解 HTML 语法，就可以使用 HTML 编辑器进 Web 创作。

但是 HTML 存在很大的局限性。由于标准的 HTML 标记已经由 W3C 预先确定，所以当描述复杂文档时 HTML 就显得力不从心。HTML 是面向描述的，而非面向对象的。因此，HTML 标记不会给出内容的含义。为什么 W3C 不再引进些标记来描述内容呢？因为这么做将导致另一个难题：浏览器生产公司会引进新的但却是非通用的标记来吸引用户使用他们的产品。

使用当前的 HTML，开发者必须对文档进行许多的调整才能与流行的浏览器兼容。由于浏览器不会去检查错误的 HTML 代码，因此导致 Internet 上大量的文档包含了错误的 HTML 语法。这个问题越来越严重，W3C 开始寻找解决办法。而这就是 XML。

### 3. 可扩展标记语言 XML

XML 可以看做是 SGML 的简版。XML 是可扩展的，我们可以创建自定义元素以满足创作需要。有了这个强大特征，不用等待 W3C 委员会发布包含需要的标记的下一个 HTML 版本了。

XML 是结构化的，每个 XML 文档都基于特定的结构。如果一个文档没有适当的结构，那么就不能认为它是 XML。

XML 比 SGML 更容易存取。因为它具有良好的结构，因此程序员可以容易地编写软件来描述 XML 文档。XML 可以方便地区分文档内容和 XML 标记元素。

## C.2 DTD 文档类型定义

DTD (Document Type Definition) 可以看做是标记语言的语法文件，它是一套定义 XML 标记如何使用的规则。DTD 定义了元素、元素的属性和取值，以及哪个元素可以被包含在另一个元素中的说明。DTD 还可以用于定义实体。

下面是一个关于 E-mail 的 DTD 文件：

```
<!ELEMENT Mail (From, To, Cc?, Date?, Subject, Body)>
<!ELEMENT From (#PCDATA) >
<!ELEMENT To (#PCDATA) >
<!ELEMENT Cc (#PCDATA) >
<!ELEMENT Date (#PCDATA) >
<!ELEMENT Subject (#PCDATA) >
<!ELEMENT Body (#PCDATA | P | Br)* >
<!ELEMENT P (#PCDATA | Br)* >
<!ATTLIST P align (left | right | justify) "left" >
<!ELEMENT Br EMPTY >
```

根据上面 DTD 的内容，与之符合的 XML 文档具备如下特征：

- 有一个 From，一个 To，一个可选择的 Cc，一个可选择的 Date，一个 Subject 和一个 Body
- From、To、Cc、Date 和 Subject 元素只包含文本信息
- Body 元素可以含有文本和零个或者多个 P 和 Br 元素
- P 元素可以包含文本和零个或者多个 Br 元素
- P 元素有一个 align 属性，它的可取值范围是 left、justify 或者 right，默认值是 left
- Br 元素的内容为空

XML 解析器将使用这个 DTD 来解析 XML 文档。DTD 使人们能够发布文档以与其他人共享。XML 文档应该包含告诉 XML 执行程序寻找 DTD 的指令，XML 文件开头的 <!DOCTYPE> 元素提供了这一功能。请看下面的例子：

```
<!DOCTYPE Mail system "http://mymailsystem.com/DTDS/mail.dtd">
<Mail>
...
...
```

...  
</Mail>

## C.3 有效 XML 文档以及简化格式的 XML 文档

XML 文档分为两类：

- 简化格式的 XML 文档
- 有效的 XML 文档

### 1. 简化格式的 XML 文档

简化格式的 XML 文档必须遵从下面几个原则：

- 至少有一个元素
- 遵守 XML 规范
- 根元素（比如上面例子中的<Mail>元素）应该不被其他元素所包含
- 适当的元素嵌套
- 除了保留实体外，所有的实体都要声明

即使没有声明 DTD 文件，XML 解析器也可以解析简化格式的 XML 文档。这个特征对于 Web 应用程序非常有利，因为应用程序不需要了解用于创建 XML 文档的 DTD 结构。

例如，以下是一个简化格式的 XML 文档的例子：

```
<?xml version="1.0" standalone="no"?>
<Mail>
  <From>Author</From>
  <To>Receiver</To>
  <Date> Thu, 7 Oct 1999 11:15:16 -0600 </Date>
  <Subject>XML Introduction</Subject>
  <Body>
    <P>Thanks for reading <Br/> this article</P>
    <Br/>
    <P>Hope you enjoyed this article</P>
  </Body>
</Mail>
```

第一行是 XML 声明，其中 version 属性指明了 XML 的版本，standalone 属性取值为“no”表示标记声明在文档内部并不是独立的。XML 声明可以看做是“运行指令”。尽管这个声明不是必需的，但最好包含它，以提高文档的灵活性。

### 2. 有效 XML 文档

有效 XML 文档指的是那些拥有一个 DTD 参考文件的 XML 文档。一个有效 XML 文档必须首先是简化格式的 XML 文档。这个文档的 DTD 文件的有效性保证了 XML 执行程序的正常运行以及文档在支持 XML 的浏览器中的正确显示。

例如，以下是一个遵守 mail.dtd 文件的有效 XML 文档。Date 元素被省略，因为在 mail.dtd 中它是可选的。元素 P 的 align 属性取值为 justify：

```
<?xml version="1.0" standalone="no"?>
```

```
<!DOCTYPE Mail system "http://mymailsystem.com/DTDs/mail.dtd">
<Mail>
    <From>Author</From>
    <To>Receiver</To>
    <Cc>Receiver2</Cc>
    <Subject>XML Introduction</Subject>
    <Body>
        Comments:<P align="justify">Thanks for reading<br/>this article</P>
        <br/>
        <P>Hope you enjoyed this article</P>
    </Body>
</Mail>
```

XML 文档可以含有注释信息，注释的语法与 HTML 相似。除了“`--`”字符串外，任何其他文本信息都可以放置在标记`<!--` 和`-->`之间。例如以下代码是 Tomcat 的 server.xml 文件中的第一行注释信息：

```
<!-- Example Server Configuration File -->
```

## C.4 XML 中的常用术语

通过本书的学习，读者已经了解到 Tomcat 的配置文件、Web 应用的发布描述文件、J2EE 应用的发布描述文件以及 SOAP 服务的发布描述文件，都是基于 XML 的。下面将解释本书中涉及的几个 XML 术语。

### C.4.1 URL、URN 和 URI

URL 是统一资源定位符（Uniform Resource Locator）的缩写，URN 是统一资源名称（Uniform Resource Name）的缩写，URI 是统一资源标识符（Uniform Resource Identifier）的缩写。

URL 是通过“通信协议+网络地址”字符串来唯一标识信息位置及资源访问路径的一种方法。

URN 主要用于唯一标识全球范围内由专门机构负责的稳定的信息资源，URN 通常给出资源名称而不提供资源位置。

URI 是一种用字符串来唯一标识信息资源的工业标准（RFC2396），它使用的范围及方式都较为广泛，在 XML 中用 URI 来标识元素的命名空间（Namespace），URI 包括了 URL 和 URN。

### C.4.2 XML 命名空间

XML 命名空间提供了一种避免元素名冲突的方法。

### 1. 名字冲突

由于 XML 中的元素名不是固定的，因此当两个不同的文档使用同样的名字描述两个不同类型的元素时就会发生名字冲突。

例如有两份 XML 文档，它们都包含了一个

```
<table>
<tr>
<td>Apples</td>
<td>Bananas</td>
</tr>
</table>
```

第二个文档的

```
<table>
<name>African Coffee Table</name>
<width>80</width>
<length>120</length>
</table>
```

如果这两个 XML 文档被放到一起，就会发生元素名冲突，因为这两个文档都包含了一个

### 2. 用一个前缀解决名字冲突

为了解决名字冲突，可以为这两个 XML 文档中的元素分别加上不同的前缀，这样就可以区分它们。

XML 文档 1：

```
<h:table>
<h:tr>
<h:td>Apples</h:td>
<h:td>Bananas</h:td>
</h:tr>
</h:table>
```

XML 文档 2：

```
<f:table>
<f:name>African Coffee Table</f:name>
<f:width>80</f:width>
<f:length>120</f:length>
</f:table>
```

现在就没有元素名冲突的问题了，因为两个文档的

### 3. 使用命名空间

下面再分别为这两个文档加上命名空间的信息。

XML 文档 1：

```
<h:table xmlns:h="http://www.mynameSpace.com/fruit">
<h:tr>
```

```
< h:td>Apples</h:td>
< h:td>Bananas</h:td>
</h:tr>
</h:table>
```

#### XML 文档 2:

```
< f:table xmlns:f="http://www.mynamespace.com/furniture">
< f:name>African Coffee Table</f:name>
< f:width>80</f:width>
< f:length>120</f:length>
</f:table>
```

以上代码在<table>元素中增加了一个 `xmlns` 属性，它代表 XML Name Space，即 XML 命名空间，它用于把元素前缀与一个命名空间 URI 相关联。

命名空间属性 `xmlns` 放在一个元素的起始标记中，它的语法如下：

```
xmlns:namespace_prefix="namespace"
```

在上面的例子中，命名空间本身是用一个 Internet 地址定义的：

```
xmlns:f="http://www.mynamespace.com/furniture"
```

W3C 命名空间规范规定命名空间本身应该是一个统一资源标识符 (URI)。当一个命名空间在一个元素的起始标记中定义时，所有有相同前缀的子元素都与这同一个命名空间相关。