飞思科技
www.feait.com.cn

开发专家之Sun ONE

精通Struts：基于MVC的Java Web设计与开发

随书附赠光盘为本书所有范例源程序以及涉及的所有软件的最新版本的安装程序

J2SE+J2EE+J2ME

Sun ONE

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

孙卫琴
飞思科技产品研发中心
编著
监制

飞思出品
www.fecit.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精粹
Tomcat与Java Web开发技术详解
精通Struts：基于MVC的Java Web设计与开发

开发专家之Sun ONE

Sun ONE

随书附赠光盘为本书所有范例源程序以及涉及的所有软件的最新版本的安装程序

当红开源软件使用指南
Java Web开发黄金拍档

上架指导：Java编程类
和《Tomcat与Java Web开发技术详解》摆放在一起

飞思在线：http://www.fecit.com.cn

总 策 划：郭 晶
执行策划：何郑燕
王 蒙
责任编辑：陆舒敏
特约编辑：王芳明
封面设计：张 跃

ISBN 7-121-00052-0
9 787121 000522 >

ISBN 7-121-00052-0 定价：49.00元
(附赠光盘1张)

开发专家之 Sun ONE

精通 Struts：基于 MVC 的 Java Web 设计与开发

孙卫琴　编著

飞思科技产品研发中心　监制

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

Struts 是目前非常流行的基于 MVC 的 Java Web 框架。本书详细介绍了在最新的 Struts 1.1 以及不久将推出的 Struts 1.2 版本上设计和开发 Java Web 应用的各种技术。

书中内容注重理论与实践相结合，按照由浅入深、前后照应的顺序来安排内容：对复杂的 Struts 框架讲解犹如庖丁解牛，先提供整体概貌，再深入局部细节；在剖析局部时，注重和框架的其他部分相联系。

书中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。随书附赠光盘内容为本书所有范例源程序，以及本书涉及的所有软件的最新版本的安装程序。

本书内容循序渐进，语言深入浅出。无论对于 Java Web 开发的新手还是行家来说，本书都是精通 Struts 技术和开发 Java Web 应用所必备的实用手册。

图书在版编目（CIP）数据

精通 Struts：基于 MVC 的 Java Web 设计与开发/孙卫琴编著. —北京：电子工业出版社，2004.8
（开发专家之 Sun ONE）
ISBN 7-121-00052-0

Ⅰ. 精... Ⅱ. 孙... Ⅲ. JAVA 语言—程序设计 Ⅳ. TP 312

中国版本图书馆 CIP 数据核字（2004）第 061637 号

责任编辑：陆舒敏　　特约编辑：王芳明
印　　刷：北京智力达印刷有限公司
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：32　字数：819.2 千字
印　　次：2004 年 8 月第 1 次印刷
印　　数：6000 册　　定价：49.00 元　（附赠光盘 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 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

前　言

如今，Java 2 Enterprise Edition（J2EE）平台已经日趋成熟，并得到广泛应用。在 J2EE 技术中发展最迅猛的当属 JavaServer Page（JSP）技术。几乎所有的基于 Java 的 Web 应用都使用 JSP。一些免费、开放源代码的 JSP/Servlet 容器，如 Apache 软件组织提供的 Tomcat，更进一步推动了 JSP 技术的发展。

随着 JSP 技术的成熟，越来越多的开发人员开始使用 Web 应用框架。框架为 Web 应用提供了预备的软件架构和相关的软件包，它大大提高了开发 Web 应用的速度和效率。现在，当开发人员开始一个新的项目时，他们首先考虑的问题不是“我们是否需要框架”，而是“我们应该使用什么样的框架”。

Jakarta-Struts 是 Apache 软件组织提供的一项开放源代码项目，它为 Java Web 应用提供了模型-视图-控制器（Model-View-Controller，MVC）框架，尤其适用于开发大型可扩展的 Web 应用。Struts 这个名字来源于在建筑和旧式飞机中使用的支撑金属架。Struts 为 Web 应用提供了一个通用的框架，使得开发人员可以把精力集中在如何解决实际业务问题上。此外，Struts 框架提供了许多可供扩展和定制的地方，使得应用程序可以方便地扩展框架，来更好地适应用户的实际需求。

当然，开发人员需要花一定的时间来学习和运用该框架。不过，一旦掌握了 Struts，就可以大大简化 Web 应用的开发过程，提高开发效率，缩短开发周期。如果想综合利用 Servlet 和 JSP 的优点来建立可扩展的 Web 应用，Struts 是不错的选择。

本书结合大量典型实用的 Java Web 实例，详细阐明了在最新的 Struts 1.1 及不久将推出的 Struts 1.2 版本上设计与开发 Java Web 应用的技术。

本书的组织结构和主要内容

本书按照由浅入深、前后照应的顺序来安排内容。对复杂的 Struts 框架讲解犹如庖丁解牛，先提供整体概貌，再深入局部细节。在剖析局部时，注重和框架的其他部分相联系。相信读者通过从整体到局部，再从局部到整体的反复学习，最终会对 Struts 框架了如指掌，游刃有余，并能把 Struts 框架灵活运用到实际的 Java Web 开发中。本书主要包含以下内容。

1．Struts 框架入门

第 1 章到第 3 章为入门篇，第 1 章简单介绍了开发 Java Web 涉及的各种技术，第 2 章和第 3 章通过两个 Struts 应用实例：helloapp 应用和 addressbook 应用，引导读者把握设计、开发和部署 Struts 应用的整体流程，充分体会 Struts 框架在开放大型、可扩展的 Web 应用方面发挥的优势。

2．配置 Struts 应用，开发模型、视图和控制器

第 4 章到第 7 章以一个电子商务网站 netstore 应用为例，深入探讨了 Struts 框架的核心组件 ActionServlet 和 RequestProcessor 的实现原理，详细介绍了开发 Struts 应用的模型、视图和控制器的各种技术，细致地描述了 Struts 配置文件的每个元素的使用方法。

3．开发 Struts 应用的一些实用技术

第 8 章到第 11 章介绍了开发 Struts 应用的一些实用技术，如 Struts 框架的扩展点、Struts 应用的国际化、Validator 验证框架和异常处理机制等。

4．Struts 标签库的使用方法

第 12 章到第 16 章结合具体的 Struts 应用实例，详细介绍了 Struts 标签库：HTML、Bean、Logic、Nested 和 Tiles 标签库的使用方法。

5．Struts 框架和 EJB 以及 Web 服务的集成

第 17 章和第 18 章介绍如何采用 EJB 和 Web 服务来实现 Struts 框架的模型，并且介绍了如何运用业务代理模式来提高模型层和控制层之间的相互独立性，使得当模型的实现发生变化时，不会对控制层造成任何影响。

6．Struts 应用的日志、项目管理以及测试

第 19 章到第 21 章介绍了如何采用第三方软件，如 Apache Common Logging API、Log4J、ANT 和 StrutsTestCase，来控制 Struts 应用的输出日志、管理以及测试 Struts 应用项目。

本书的范例程序

Java Web 开发是实践性很强的技术，为了使读者不但能掌握 Struts 框架的理论，并且能迅速获得开发 Struts 应用的实际经验，彻底掌握并会灵活运用 Struts 技术，本书提供了大量典型实用的 Java Web 实例。所有的范例选用最新的 Struts1.1 版本（2004/5/10 发布），在 Tomcat 5.0.24 上运行通过；对于用 EJB 来实现模型的 netstore 应用，在 JBoss 与 Tomcat 的集成软件（Jboss-3.2.1_tomcat-4.1.24）上运行通过。在本书附赠光盘中包含了所有范例源文件。本书以 9 个 Struts 应用的例子贯穿全书。

1．netstore 应用

netstore 应用是一个充分运用了 Struts 各种技术的综合例子，它实现了一个购物网站，更加贴近于实际应用。本书以 netstore 应用为例，详细介绍了模型的设计与开发策略，涉及的技术包括：业务对象、模型与关系型数据库之间的持久化框架、控制层访问模型的业务代理模式，以及采用 EJB 和 SOAP Web 服务来实现模型。

2．helloapp 应用

helloapp 应用是本书最简单的 Struts 应用例子，本书一共提供了 5 个版本，每个版本各有侧重点：

- version1：引导读者入门，把握设计、开发和部署 Struts 应用的整体流程。
- version2：介绍 Struts 应用的国际化。
- version3：介绍如何在 Struts 应用中使用 Apache Common Logging API 和 Log4J。
- version4：介绍如何用 ANT 工具来管理 Struts 应用项目。
- version5：介绍如何用 StrutsTestCase 工具来测试 Struts 应用。

3．addressbook 应用

addressbook 应用实现了一个电子通讯簿，本书提供了 3 个版本，每个版本各有侧重点：

- version1：引导读者入门，进一步掌握为视图、控制器和模型组件分配功能的技巧，以及如何实现这些组件之间的通信和数据共享。

- version2：介绍 Struts 插件的用法，以及使用 Token 机制来避免表单数据的重复提交。
- version3：介绍如何为 HTML 表单分页。

4．exsample 应用

exsample 应用用于演示如何在 Struts 应用中处理异常。

5．htmltaglibs 应用

htmltaglibs 应用用于演示如何使用 Struts HTML 标签库。

6．beantaglibs 应用

beantaglibs 应用用于演示如何使用 Struts Bean 标签库。

7．logictaglibs 应用

logictaglibs 应用用于演示如何使用 Struts Logic 标签库。

8．nestedtaglibs 应用

nestedtaglibs 应用用于演示如何使用 Struts Nested 标签库。

9．tilestaglibs 应用

tilestaglibs 应用用于演示如何使用 Struts Tiles 框架和 Tiles 标签库。

这本书是否适合您

本书侧重于介绍基于 Struts 的 Java Web 应用的框架结构，适用于所有从事开发 Java Web 应用的读者。阅读本书，要求读者具备 Servlet 和 JSP 编程的基础知识。如果您对这些还不了解，可以参考作者的另一本书：《Tomcat 与 Java Web 开发技术详解》，该书已由电子工业出版社于 2004 年 4 月出版。

如果您是开发 Struts 应用的新手，建议按照本书的先后顺序来学习。您可以先从简单的 Struts 应用实例下手，把握 Struts 框架的大致流程，然后逐步深入了解各个组件的细节。Struts 框架本身是一个环环相扣的有机整体，本书在内容安排上注重前后照应，帮助读者把 Struts 框架的各个环节联系起来，最终达到对 Struts 框架了如指掌。

如果您已经在开发 Struts 应用方面有着丰富经验，则可以把本书作为实用的 Struts 技术参考资料。本书深入探讨了 Struts 框架的内置核心组件的实现原理，详细介绍了开发 Struts 应用的各种实用技术。灵活运用本书介绍的 Struts 最新技术，将使 Struts 应用开发更加得心应手。

实践是掌握 Java Web 开发技术最迅速有效的办法。以 Struts 标签库为例，已经在使用 Struts 的读者一定有这样的体会，即使熟读了 Apache 网站提供的关于 Struts 标签库的文档，还是对如何使用 Struts 标签库无从下手。为了让读者彻底掌握并学会灵活运用 Struts 技术，本书提供了大量典型的例子，在本书附赠光盘上提供了完整的源代码，以及软件安装程序。建议读者在学习 Struts 技术的过程中，善于将理论与实践相结合，达到事半功倍的效果。

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内容涉及的所有软件的最新版本的安装程序，包括：

（1）Tomcat 的安装软件（Tomcat 5.0.24）。
（2）MySQL 服务器的安装软件（MySQL 4.0.14）。
（3）Apache AXIS 软件（Apache AXIS1.1）。
（4）Log4J 软件（Log4J 1.2.8）。
（5）Struts 软件（Struts 1.1，2004/05/10 发布）。
（6）Struts 源代码（Struts 1.1，2004/05/11 发布）。
（7）Jakarta Log 标签库（Jakarta Log 1.2）。
（8）JBoss 与 Tomcat 的集成软件（Jboss-3.2.1_tomcat-4.1.24）。
（9）Ant 的安装软件（Ant 1.5.4）。
（10）StrutsTestCase 软件（StrutsTestCase 2.1.0，2004/01/15 发布）。
（11）OJB 软件（OJB 1.0）。

2．lib 目录

在该目录下提供了编译本书提供的 Java Web 应用所需的 JAR 文件。

3．sourcecode 目录

在该目录下提供了本书所有的源程序。

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，此外曹汉玉、孙金定和曹文伟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　　linda_j2ee@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

第 1 章　Struts 与 Java Web 应用简介 1
1.1　Java Web 应用概述 1
1.1.1　Servlet 组件 2
1.1.2　JSP 组件 3
1.1.3　共享数据在 Web 应用中的范围 3
1.1.4　JavaBean 组件及其在 Web 应用中的范围 5
1.1.5　客户化 JSP 标签 5
1.1.6　EJB 组件 6
1.1.7　XML 语言 6
1.1.8　Web 服务器和应用服务器 7
1.2　Web 组件的三种关联关系 7
1.2.1　请求转发 7
1.2.2　请求重定向 8
1.2.3　包含 9
1.3　MVC 概述 9
1.3.1　MVC 设计模式 10
1.3.2　JSP Model1 和 JSP Model2 11
1.4　Struts 概述 13
1.4.1　Struts 实现 MVC 的机制 13
1.4.2　Struts 的工作流程 15
1.5　小结 17
第 2 章　Struts 应用：helloapp 应用 19
2.1　分析 helloapp 应用的需求 19
2.2　运用 Struts 框架 19
2.3　创建视图组件 20
2.3.1　创建 JSP 文件 20
2.3.2　创建消息资源文件 23
2.3.3　创建 ActionForm Bean 23
2.3.4　数据验证 25
2.4　创建控制器组件 26
2.4.1　Action 类的工作机制 28
2.4.2　访问封装在 MessageResources 中的本地化文本 28
2.4.3　业务逻辑验证 28
2.4.4　访问模型组件 29
2.4.5　向视图组件传递数据 30
2.4.6　把 HTTP 请求转发给合适的视图组件 30

2.5 创建模型组件 30
2.6 创建存放常量的 Java 文件 31
2.7 创建配置文件 32
2.7.1 创建 Web 应用的配置文件 32
2.7.2 创建 Struts 框架的配置文件 33
2.8 发布和运行 helloapp 应用 35
2.8.1 服务器端装载 hello.jsp 的流程 37
2.8.2 表单验证的流程 37
2.8.3 逻辑验证失败的流程 39
2.8.4 逻辑验证成功的流程 40
2.9 小结 41
第 3 章 Struts 应用的需求分析与设计 43
3.1 收集和分析应用需求 43
3.2 设计数据库 44
3.3 设计应用的业务逻辑 45
3.3.1 访问 XML 格式的用户信息 45
3.3.2 访问数据库 47
3.4 设计用户界面 49
3.4.1 界面风格 50
3.4.2 使用客户化标签 52
3.5 设计 ActionForm 53
3.6 设计 Action 和 Action 映射 55
3.6.1 设计 LogonAction 58
3.6.2 设计 LogoffAction 60
3.6.3 设计 InsertAction 61
3.6.4 设计 SearchAction 62
3.6.5 设计 DisplayAllAction 63
3.7 设计客户化标签 64
3.7.1 设计 ValidateSessionTag 标签 64
3.7.2 设计 DisplayTag 标签 66
3.7.3 创建客户化 app 标签库的 TLD 文件 67
3.8 小结 68
第 4 章 配置 Struts 应用 71
4.1 Web 应用的发布描述文件 71
4.1.1 Web 应用发布描述文件的文档类型定义（DTD） 71
4.2 为 Struts 应用配置 web.xml 文件 72
4.2.1 配置 Struts 的 ActionServlet 72
4.2.2 声明 ActionServlet 的初始化参数 73
4.2.3 配置欢迎文件清单 74
4.2.4 配置错误处理 75

4.2.5 配置 Struts 标签库 76
4.3 Struts 配置文件 77
4.3.1 org.apache.struts.config 包 77
4.3.2 <struts-config>元素 79
4.3.3 <data-sources>元素 80
4.3.4 <form-beans>元素 82
4.3.5 <global-exceptions>元素 83
4.3.6 <global-forwards>元素 84
4.3.7 <action-mappings>元素 85
4.3.8 <controller>元素 87
4.3.9 <message-resources>元素 87
4.3.10 <plug-in>元素 88
4.3.11 配置多应用模块 89
4.4 Digester 组件 91
4.5 Struts 控制面板工具 91
4.6 重新载入配置文件 93
4.7 小结 94
第 5 章 Struts 控制器组件 95
5.1 控制器组件的控制机制 95
5.1.1 ActionServlet 类 96
5.1.2 RequestProcessor 类 99
5.1.3 Action 类 105
5.2 使用内置的 Struts Action 类 112
5.2.1 org.apache.struts.actions.ForwardAction 类 112
5.2.2 org.apache.struts.actions.IncludeAction 类 114
5.2.3 org.apache.struts.actions.DispatchAction 类 115
5.2.4 org.apache.struts.actions.LookupDispatchAction 类 118
5.2.5 org.apache.struts.actions.SwitchAction 类 121
5.3 利用 Token 解决重复提交 121
5.4 实用类 125
5.4.1 RequestUtils 类 125
5.4.2 TagUtils 类 125
5.4.3 ModuleUtils 类 126
5.4.4 Globals 类 126
5.5 小结 127
第 6 章 Struts 模型组件 129
6.1 模型在 MVC 中的地位 129
6.2 模型的概念和类型 130
6.2.1 概念模型 130
6.2.2 设计模型 131

6.3 业务对象（BO） 133
6.3.1 业务对象的特征和类型 133
6.3.2 业务对象的重要性 134
6.4 业务对象的持久化 134
6.4.1 对业务对象进行持久化的作用 135
6.4.2 数据访问对象（DAO）设计模式 135
6.4.3 常用的 ORM 软件 136
6.5 创建 netstore 应用的模型 137
6.5.1 为 netstore 应用创建模型的步骤 137
6.5.2 创建 netstore 应用的业务对象 138
6.5.3 创建 netstore 应用的数据库 142
6.5.4 netstore 应用的 ORM 框架 145
6.5.5 联合使用业务代理和 DAO 模式 155
6.6 小结 167
第 7 章 Struts 视图组件 169
7.1 视图概述 169
7.2 在视图中使用 JavaBean 170
7.2.1 DTO 数据传输对象 170
7.2.2 Struts 框架提供的 DTO: ActionForm Bean 171
7.3 使用 ActionForm 172
7.3.1 ActionForm 的生命周期 172
7.3.2 创建 ActionForm 174
7.3.3 配置 ActionForm 176
7.3.4 访问 ActionForm 177
7.3.5 处理表单跨页 178
7.4 使用动态 ActionForm 184
7.4.1 配置动态 ActionForm 185
7.4.2 动态 ActionForm 的 reset()方法 186
7.4.3 访问动态 ActionForm 187
7.4.4 动态 ActionForm 的表单验证 187
7.4.5 在 netstore 应用中使用动态 ActionForm 187
7.5 小结 189
第 8 章 扩展 Struts 框架 191
8.1 Struts 插件（PlugIn） 191
8.2 扩展 Struts 的配置类 194
8.3 控制器扩展点 195
8.3.1 扩展 ActionServlet 类 195
8.3.2 扩展 RequestProcessor 类 196
8.3.3 扩展 Action 类 198
8.4 扩展视图组件 200

8.5 扩展模型组件 200
8.6 小结 204
第 9 章 Struts 应用的国际化 205
9.1 本地化与国际化的概念 205
9.2 Web 应用的中文本地化 206
9.2.1 处理 HTTP 请求数据编码 207
9.2.2 处理数据库数据编码 207
9.2.3 处理 XML 配置文件编码 207
9.2.4 处理响应结果的编码 208
9.3 Java 对 I18N 的支持 208
9.3.1 Locale 类 208
9.3.2 ResourceBundle 类 214
9.3.3 MessageFormat 类和复合消息 214
9.4 Struts 框架对国际化的支持 216
9.4.1 创建 Struts 的 Resource Bundle 216
9.4.2 访问 Resource Bundle 217
9.5 对 helloapp 应用实现国际化 220
9.5.1 对 JSP 文件的文本、图片和按钮进行国际化 220
9.5.2 创建临时中文资源文件 221
9.5.3 对临时资源文件进行编码转换 222
9.5.4 创建英文资源文件 223
9.5.5 采用 Servlet 过滤器设置请求数据的字符编码 223
9.5.6 运行国际化的 helloapp 应用 225
9.6 异常处理的国际化 226
9.7 小结 226
第 10 章 Validator 验证框架 229
10.1 安装和配置 Validator 框架 229
10.1.1 validator-rules.xml 文件 229
10.1.2 validation.xml 文件 233
10.1.3 Validator 插件 237
10.2 Validator 框架和 ActionForm 238
10.3 Validator 框架和 Struts 客户化标签 240
10.4 在 netstore 应用中使用 Validator 框架 241
10.5 创建自定义的验证规则 245
10.6 在 Validator 框架中使用 JavaScript 247
10.7 Validator 框架的国际化 250
10.8 小结 251
第 11 章 异常处理 253
11.1 Java 异常处理 253

11.1.1 Java 异常 253
11.1.2 JVM 的方法调用堆栈 254
11.1.3 异常处理对性能的影响 255
11.1.4 系统异常和应用异常 256
11.1.5 使用异常链 256
11.1.6 处理多样化异常 258
11.2 Struts 框架异常处理机制概述 260
11.3 Struts 框架异常处理机制的细节 261
11.3.1 Java Web 容器处理异常的机制 261
11.3.2 ActionServlet 类处理异常的机制 261
11.3.3 RequestProcessor 类处理异常的机制 262
11.3.4 ExceptionHandler 类处理异常的机制 263
11.4 在 Struts 应用中处理异常的各种方式 266
11.4.1 创建异常类 266
11.4.2 由 Java Web 容器捕获异常 269
11.4.3 以配置方式处理异常 271
11.4.4 以编程方式处理异常 277
11.5 小结 282
第 12 章 Struts HTML 标签库 283
12.1 用于生成基本的 HTML 元素的标签 283
12.1.1 <html:html>标签 288
12.1.2 <html:base>标签 288
12.1.3 <html:link>和<html:rewrite>标签 289
12.1.4 <html:img>标签 292
12.2 基本的表单标签 293
12.2.1 <html:form>标签 296
12.2.2 <html:text>标签 296
12.2.3 <html:cancel>标签 297
12.2.4 <html:reset>标签 298
12.2.5 <html:submit>标签 298
12.2.6 <html:hidden>标签 299
12.3 检查框和单选按钮标签 299
12.3.1 <html:checkbox>标签 304
12.3.2 <html:multibox>标签 305
12.3.3 <html:radio>标签 306
12.4 下拉列表和多选列表标签 307
12.4.1 <html:select>标签 311
12.4.2 <html:option>标签 312
12.4.3 <html:options>标签 313
12.4.4 <html:optionsCollection>标签 314

12.5 在表单中上传文件标签 315
12.5.1 <html:file>标签 319
12.5.2 在 ActionForm Bean 中设定 FormFile 属性 319
12.5.3 在 Action 类中处理文件上传 319
12.6 <html:errors>标签 320
12.6.1 错误消息的来源 324
12.6.2 格式化地显示错误消息 326
12.6.3 <html:errors>标签的用法 326
12.7 <html:messages>标签 328
12.8 小结 330
第 13 章 Struts Bean 标签库 331
13.1 访问 HTTP 请求信息或 JSP 隐含对象 331
13.1.1 <bean:header>标签 334
13.1.2 <bean:parameter>标签 334
13.1.3 <bean:cookie>标签 335
13.1.4 <bean:page>标签 336
13.2 访问 Web 应用资源 337
13.2.1 <bean:message>标签 339
13.2.2 <bean:resource>标签 341
13.2.3 <bean:struts>标签 341
13.2.4 <bean:include>标签 342
13.3 定义或输出 JavaBean 342
13.3.1 <bean:define>标签 344
13.3.2 <bean:size>标签 345
13.3.3 <bean:write>标签 346
13.4 小结 347
第 14 章 Struts Logic 标签库 349
14.1 进行比较运算的 Logic 标签 349
14.2 进行字符串匹配的 Logic 标签 354
14.3 判断指定内容是否存在的 Logic 标签 356
14.3.1 <logic:empty>和<logic:notEmpty>标签 359
14.3.2 <logic:present>和<logic:notPresent>标签 360
14.3.3 <logic:messagesPresent>和<logic:messagesNotPresent>标签 361
14.4 进行循环遍历的 Logic 标签 363
14.4.1 遍历集合 366
14.4.2 遍历 Map 367
14.4.3 设定被遍历的变量 369
14.5 进行请求转发或重定向的 Logic 标签 369
14.5.1 <logic:forward>标签 369
14.5.2 <logic:redirect>标签 370

14.6 小结 370
第 15 章 Struts Nested 标签库 373
15.1 <nested:nest>和<nested:writeNesting>标签 373
15.2 <nested:root>标签 379
15.3 和其他标签库中的标签功能相同的 Nested 标签 381
15.4 小结 382
第 16 章 Tiles 框架 383
16.1 采用基本的 JSP 语句创建复合式网页 383
16.2 采用 JSP 的 include 指令创建复合式网页 387
16.3 采用<tiles:insert>标签创建复合式网页 391
16.4 采用 Tiles 模板创建复合式网页 394
16.5 采用 Tiles 模板和 Tiles 组件创建复合式网页 396
16.5.1 Tiles 组件的基本使用方法 397
16.5.2 通过 Struts Action 来调用 Tiles 组件 399
16.5.3 Tiles 组件的组合 399
16.5.4 Tiles 组件的扩展 402
16.6 小结 403
第 17 章 Struts 与 EJB 组件 405
17.1 J2EE 体系结构简介 405
17.2 创建 EJB 组件 406
17.2.1 编写 Remote 接口 407
17.2.2 编写 Home 接口 408
17.2.3 编写 Enterprise Java Bean 类 408
17.3 在 Struts 应用中访问 EJB 组件 413
17.3.1 创建业务代理实现类 NetstoreEJBDelegate 413
17.3.2 运用 EJBHomeFactory 模式 416
17.4 发布 J2EE 应用 419
17.4.1 在 Jboss-Tomcat 上部署 EJB 组件 419
17.4.2 在 Jboss-Tomcat 上部署 Web 应用 421
17.4.3 在 Jboss-Tomcat 上部署 J2EE 应用 422
17.5 小结 424
第 18 章 Struts 与 SOAP Web 服务 425
18.1 SOAP 简介 425
18.2 建立 Apache AXIS 环境 427
18.3 创建和发布 SOAP 服务 428
18.3.1 创建实现 SOAP 服务的 Java 类 428
18.3.2 创建 Web 服务发布描述文件 429
18.3.3 发布 SOAP 服务 432
18.4 在 Struts 应用中访问 SOAP 服务 432

18.5 小结 438
第 19 章 Struts 与 Apache 通用日志包 439
19.1 Apache 通用日志包概述 439
19.1.1 Log 接口 439
19.1.2 LogFactory 接口 441
19.2 常用的日志实现 441
19.2.1 NoOpLog 日志器 441
19.2.2 SimpleLog 日志器 442
19.2.3 Log4J 日志器 442
19.3 配置通用日志接口 443
19.3.1 准备 JAR 文件 443
19.3.2 指定日志器 443
19.3.3 设置日志器的属性 444
19.4 配置 Log4J 444
19.4.1 配置 Log4J 的一般步骤 444
19.4.2 Log4J 的配置样例 446
19.4.3 Log4J 对应用性能的影响 447
19.5 在 Struts 应用中访问通用日志接口 447
19.5.1 在 Action 类中访问通用日志接口 448
19.5.2 在 JSP 中访问通用日志接口 450
19.6 小结 454
第 20 章 用 ANT 工具管理 Struts 应用 455
20.1 Web 应用常用的开发目录结构 455
20.2 安装配置 ANT 455
20.3 创建 build.xml 文件 456
20.3.1 设置公共属性 462
20.3.2 设置 classpath 463
20.3.3 定义 help target 463
20.3.4 定义 clean-all target 463
20.3.5 定义 prepare target 464
20.3.6 定义 compile target 464
20.3.7 定义 build target 465
20.3.8 定义 deploy target 466
20.3.9 定义 javadoc target 466
20.4 运行 ANT 466
20.4.1 运行 help target 466
20.4.2 运行 deploy target 467
20.4.3 运行 javadoc target 468
20.5 小结 468

第 21 章　用 StrutsTestCase 测试 Struts 应用 469
21.1　StrutsTestCase 简介 469
21.2　制订单元测试用例 469
21.3　创建 StrutsTestCase 测试类 470
21.4　用 ANT 工具运行测试程序 471
21.4.1　准备必要的 JAR 文件 471
21.4.2　在 build.xml 文件中定义 test target 471
21.4.3　运行测试程序 472
21.5　创建包含多个单元测试用例的测试类 473
21.6　小结 475
附录 A　Struts1.1 的 UML 类框图 476
附录 B　Struts 资源 477
B.1　Struts 邮件列表 477
B.2　Struts 资源 Web 站点 477
B.3　Tiles 标签库站点 477
B.4　Nested 标签库站点 478
B.5　Struts GUI 工具 478
B.6　Easy Struts 工程 478
附录 C　发布和运行 addressbook 应用 479
C.1　发布 addressbook 应用 479
C.2　运行 addressbook 应用 479
附录 D　发布和运行 netstore 应用 482
D.1　运行 netstore 所需的软件 482
D.2　netstore 应用的目录结构 483
D.3　安装 netstore 数据库 484
D.4　发布 netstore 应用 484
D.4.1　在工作模式 1 下发布 netstore 应用 484
D.4.2　在工作模式 2 下发布 netstore 应用 485
D.4.3　在工作模式 3 下发布 netstore 应用 485
D.4.4　在工作模式 4 下发布 netstore 应用 486
D.4.5　在工作模式 5 下发布 netstore 应用 486
D.5　运行 netstore 应用 487
附录 E　编译本书的 Java Web 样例 491
附录 F　Struts 1.2 API 的新特征 492

第 1 章　Struts 与 Java Web 应用简介

Struts 为 Java Web 应用提供了现成的通用的框架。Struts 可以大大提高 Web 应用的开发速度。如果没有 Struts，开发人员将不得不首先花大量的时间和精力来设计、开发自己的框架。如果在 Web 应用中恰到好处地使用 Struts，将把从头开始设计框架的时间节省下来，使得开发人员可以把精力集中在如何解决实际业务问题上。

而且 Struts 本身是一群经验丰富的 Web 开发专家的集体智慧结晶，在全世界范围内得到广泛运用并得到一致认可。因此对于开发大型复杂的 Web 应用，Struts 是不错的框架选择。

本章先回顾了开发 Java Web 应用涉及的各种技术，由于本书的重点是 Struts，因此没有对这些技术进行深入探讨，如果需要深入了解这些技术，可以参考作者的另一本书《Tomcat 与 Java Web 开发技术详解》。

本章接着介绍了 MVC 设计模式的结构和优点，然后介绍了 Sun 公司提出的在 Java Web 开发领域的两种规范：JSP Model1 和 JSP Model2，最后介绍了 Struts 实现 MVC 的机制。

1.1　Java Web 应用概述

Java Web 应用的核心技术是 Java Server Page 和 Servlet。此外，开发一个完整的 Java Web 应用还涉及以下概念及技术：

- JavaBean 组件
- EJB 组件
- 自定义 JSP 标签
- XML
- Web 服务器和应用服务器

图 1-1 显示了 Java Web 应用的结构。

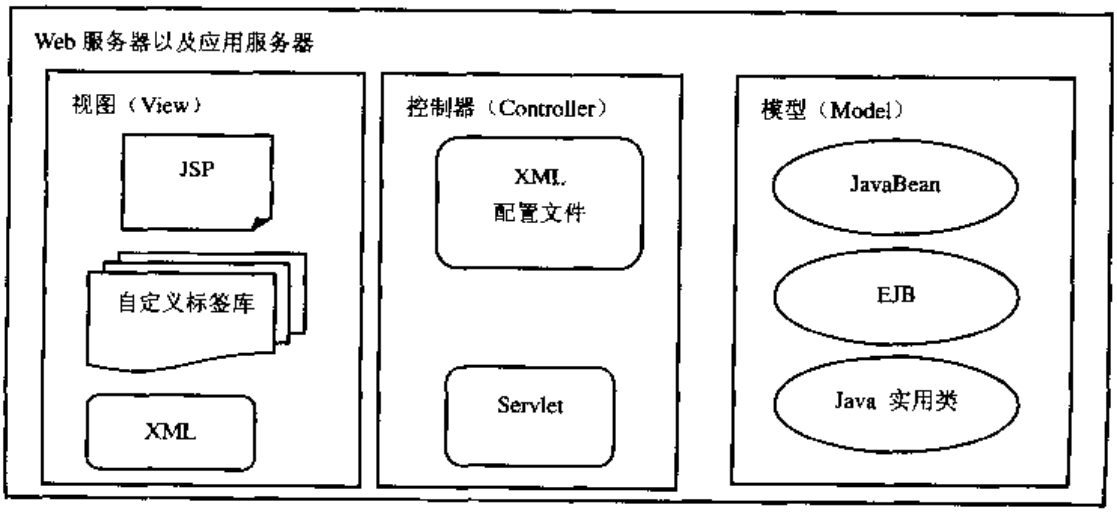

图 1-1　Java Web 应用的结构

1.1.1 Servlet 组件

Servlet 在 Web 应用中担任重要角色。Servlet 运行于 Servlet 容器中，可以被 Servlet 容器动态加载，来扩展服务器的功能，并提供特定的服务。Servlet 按照请求/响应的方式工作。在 Struts 框架中，控制器组件就是由 Servlet 来构成的。

当用户请求访问某个 Servlet 时，Servlet 容器将创建一个 ServletRequest 对象和 ServletResponse 对象。在 ServletRequest 对象中封装了用户请求信息，然后 Servlet 容器把 ServletRequest 对象和 ServletResponse 对象传给用户所请求的 Servlet。Servlet 把响应结果写到 ServletResponse 中，然后由 Servlet 容器把响应结果传给用户。图 1-2 显示了 Servlet 容器响应用户请求的过程。

图 1-2 Servlet 容器响应用户请求的过程

在 Java Servlet API 中有以下几个比较重要的类，它们决定了 Web 应用的请求/响应方式及各种共享数据的存放地点：

- HttpServletRequest：Servlet 容器把 HTTP 请求信息包存在 HttpServletRequest 对象中，Servlet 组件从 request 对象中读取用户的请求数据。此外，HttpServletRequest 可以存放 request 范围内的共享数据。
- HttpServletResponse：用于生成 HTTP 响应结果。
- HttpSession：Servlet 容器为每个 HTTP 会话创建一个 HttpSession 实例，HttpSession 可以存放 session 范围的共享数据
- ServletContext：Servlet 容器为每个 Web 应用创建一个 ServletContext 实例，ServletContext 可以存放 application 范围的共享数据。

HttpServletRequest、HttpSession 和 ServletContext 分别提供了在 request、session 和 application 范围内保存和读取共享数据的方法：

```
//save shared data
setAttribute(String key, Object value);
//get shared data
getAttribute(String key);
```

在保存共享数据时，应该指定属性 key。在读取共享数据时，将根据这个属性 key 来检索共享数据。例如，以下代码把 ShoppingCart 对象（购物车）存放在 session 范围内，存放时指定属性 key 为“cart”，然后再通过这个属性 key 把 ShoppingCart 对象检索出来：

```
ShoppingCart shoppingCart=new ShoppingCart();
ShoppingCart myCart=null;
//save cart
session.setAttribute("cart",shoppingCart);
//get cart
```

```
myCart=(ShoppingCart)session.getAttribute("cart");
```

如果要深入了解 Servlet 的最新技术，可以访问 http://java.sun.com/products/servlet/index.html 或者 http://www.servlets.com。

1.1.2　JSP 组件

在传统的 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，网络上传送给用户端的仅为输出结果。JSP 技术大大降低了对用户浏览器的要求，即使用户浏览器端不支持 Java，也可以访问 JSP 网页。

当 JSP 容器接收到 Web 用户的一个 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，JSP 容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。图 1-3 显示了 JSP 容器初次执行 JSP 的过程：

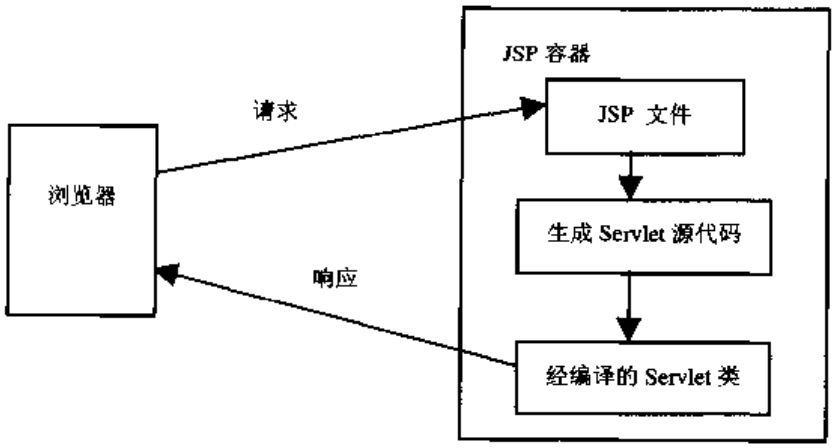

图 1-3　JSP 容器初次执行 JSP 的过程

尽管 JSP 在本质上就是 Servlet，但这两者的创建方式不一样。Servlet 完全由 Java 程序代码构成，擅长于流程控制和事务处理，而通过 Servlet 来生成动态网页很不直观；JSP 由 HTML 代码和 JSP 标签构成，可以方便地编写动态网页。因此在实际应用中，采用 Servlet 来控制业务流程，而采用 JSP 来生成动态网页。在 Struts 框架中，JSP 位于 MVC 设计模式的视图层，而 Servlet 位于控制层。

如果要深入学习最新的 JSP 开发技术，可以访问 http://java.sun.com/products/jsp 或者 http://java.sun.com/products/jsp/docs.html。

1.1.3　共享数据在 Web 应用中的范围

在 Web 应用中，如果某种数据需要被多个 Web 组件共享，可以把这些共享数据存放在特定的范围内。共享数据有 4 种存在范围，参见图 1-4，这 4 种范围分别是：

- page：共享数据的有效范围是用户请求访问的当前 JSP 网页。

- request：共享数据的有效范围为“用户请求访问的当前 Web 组件，以及和当前 Web 组件共享同一个用户请求的其他 Web 组件”。如果用户请求访问的是 JSP 网页，那么该 JSP 网页的<%@ include>指令以及<forward>标记包含的其他 JSP 文件也能访问共享数据。request 范围内的共享数据实际上存放在 HttpServletRequest 对象中。
- session：共享数据存在于整个 HTTP 会话的生存周期内，同一个 HTTP 会话中的 Web 组件共享它。session 范围内的共享数据实际上是存放在 HttpSession 对象中的。
- application：共享数据存在于整个 Web 应用的生命周期内，Web 应用中的所有 Web 组件都能共享它。共享数据实际上存放在 ServletContext 对象中。

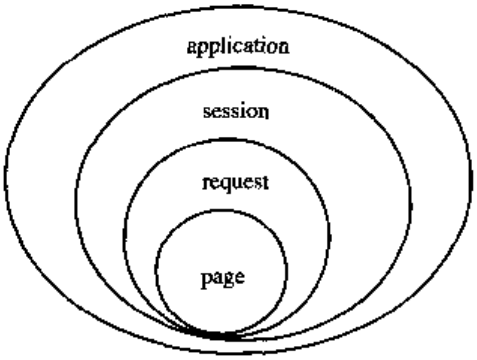

图 1-4 共享数据在 Web 应用中的范围

图 1-4 根据 4 种范围的生命周期的长短，直观地比较了它们的大小。但并不意味着这几种范围之间存在包含关系。

当客户第一次访问 Web 应用中支持会话的某个网页时，就会开始一个新的 HTTP 会话，Servlet 容器为这个会话创建一个 HttpSession 对象。接下来，当客户浏览这个 Web 应用的不同网页时，始终处于同一个会话中。会话拥有特定的生命周期。在以下情况中，会话将结束生命周期，Servlet 容器会将 HTTP 会话所占用的资源释放掉：

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

把共享数据保存在 session 范围内，有助于服务器在同一个 HTTP 会话中跟踪用户的状态，例如在购物网站中，可以跟踪用户购物车的状态。但是在 session 范围内保存大量的共享数据，会消耗大量的内存资源。假设一个网站同时被 1000 个用户访问，每个用户的 session 占用 0.5MB 的内存，那么所有的 session 共占用 500MB 空间。解决 session 消耗大量内存有两个办法：

- 运用 Java Web 容器的 Session 管理工具，对 Session 进行持久化管理。如 Tomcat 就提供了管理 Session 的功能。
- 如果把共享数据保存在 request 范围内也能完成和存放在 session 范围内同样的功能，则优先考虑保存在 request 范围内。因为 HttpServletRequest 对象的生命周期比 HttpSession 对象短得多，当服务器响应完用户请求时，相应的 request 对象就

结束生命周期，Java 虚拟机会负责回收 request 对象占用的内存。

1.1.4 JavaBean 组件及其在 Web 应用中的范围

JavaBean 是一种符合特定规范的 Java 对象，在 JavaBean 中定义了一系列的属性，并提供了访问和设置这些属性的公共方法。JavaBean 可以作为共享数据，存放在 page、request、session 或 application 范围内。在 JSP 文件中，可以通过专门的标签来定义或访问 JavaBean。假定有一个 JavaBean 的类名为 CounterBean，它有一个 count 属性，以下代码显示了在 JSP 文件中分别定义 4 种范围内的 JavaBean 对象的语法：

```
//in page scope
<jsp:useBean id="myBean1" scope="page" class="CounterBean" />
//in request scope
<jsp:useBean id="myBean2" scope="request" class="CounterBean" />
//in session scope
<jsp:useBean id="myBean3" scope="session" class="CounterBean" />
//in application scope
<jsp:useBean id="myBean4" scope="application" class="CounterBean" />
```

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="myBean1" 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="myBean1" property="count" value="0" />
```

当 JSP 与 JavaBean 搭配使用时，JSP 可侧重于生成动态网页，数据或逻辑由 JavaBean 来提供，这样能充分利用 JavaBean 组件的可重用性特点，提高开发网站的效率。

在 Struts 框架中，ActionForm Bean 就是一种典型的 JavaBean，它能够在视图层和控制层之间传递用户输入的表单数据。它有两种存在范围：request 和 session。

如果要深入了解 JavaBean，可以访问 http://java.sun.com/products/javabeans。

1.1.5 客户化 JSP 标签

JSP 标签库技术是在 JSP1.1 版本中才出现的，它支持用户在 JSP 文件中自定义客户化标签，这些可重用的标签能够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。自定义 JSP 标签可以使 JSP 代码更加简洁，有助于将 JSP 文件中的 Java 程序代码分离出去，使 JSP 文件侧重于提供 HTML 表示层数据。客户化标签有以下优点：

- 标签具有可重用性，因此可以提高开发效率。
- 可以在 JSP 页面以静态或动态的方式客户化设置自定义标签的属性。
- 标签可以访问 JSP 网页中的所有对象，如 HttpServletRequest 和 HttpServletResponse 等。

- 标签可以相互嵌套，来完成复杂的逻辑。
- 标签可以使 JSP 页面变得更加简洁，提高可读性。

Struts 提供了五种功能强大的标签库，包括：Bean 标签库、HTML 标签库、Logic 标签库、Nested 标签库和 Tiles 标签库。熟练使用这些标签库，可以简化开发交互式的、基于表单的 Web 应用的过程。

如果要深入学习客户化 JSP 标签技术，可以访问 http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html 或者 http://jsptags.com。

1.1.6 EJB 组件

Enterprise Java Bean（简称 EJB）组件是基于标准分布式对象技术、CORBA 和 RMI 的服务器端 Java 组件。EJB 组件和 JavaBean 组件一样，都用于实现企业应用的业务逻辑，它们的根本区别在于：EJB 组件总是分布式的，Sun 公司制定的 EJB 组件模型要求 EJB 组件运行于 EJB 服务器（通常称为应用服务器）中，而 JavaBean 组件可以和 Servlet 或 JSP 运行在由 Servlet/JSP 容器提供的同一个 Java 虚拟机中。

EJB 组件提供了应用的业务逻辑，但不涉及表示层，不提供用户界面，在 Struts 框架中，它位于 MVC 设计模式的模型层。

EJB 组件运行在 EJB 容器中，EJB 容器由专门的 EJB 容器厂商提供。实现了 EJB 规范的 EJB 容器提供安全、资源共享、持续运行、并行处理、事务完整性等服务，从而简化了商业应用系统的开发。

如果要深入学习 EJB 组件开发技术，可以访问 http://java.sun.com/products/ejb/index.html。

1.1.7 XML 语言

XML，即可扩展标记语言（Extensible Markup Language），是一种用来创建自定义标记的标记语言。XML 在 Web 应用以及 Web 服务开发中得到了广泛的运用，可用来描述结构化的数据。XML 的标记通常都包含一对起始和结束标签，在标签之间插入相应的数据，例如：

```
<friend>
   <name> Linda</name>
   <phone>68834567</phone>
   <address>Shanghai,China</address>
</friend>
```

以上代码由四个标签构成，这一组数据代表了通讯簿中一个朋友的通信信息。在 XML 文件中通常会声明文档类型定义（DTD，Document Type Definition）。DTD 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，以及元素之间的嵌套关系。

XML 文件常用做各种软件应用的配置文件。在基于 Struts 框架的 Web 应用中，有两个重要的配置文件：web.xml 和 struts-config.xml。web.xml 文件用于配置 Web 应用，如 Servlet

组件；struts-config.xml 用于配置 Struts 框架，如各种 Action 组件。

如果要深入了解 XML，可以访问 http://www.w3.org/TR/2000/REC-xml-20001006 或者 http://www.w3schools.com/xml/default.asp。

1.1.8　Web 服务器和应用服务器

任何一个 Web 应用都离不开 Web 服务器以及应用服务器，Web 服务器用于处理 HTTP 请求，应用服务器可以提供和 Web 应用相关的服务，如 EJB 容器就是一种处理业务逻辑和事务的应用服务器。

本书使用的 Web 服务器为 Apache 软件组织开发的 Tomcat 服务器，它是一个开放源代码的软件。Tomcat 不仅可以作为独立的 Web 服务器，还可以同时作为优秀的 Servlet/JSP 容器。本书使用的 EJB 容器为 Jboss，它是由 Jboss 软件组织开发的开放源代码软件，性能可靠，运行速度快。

1.2　Web 组件的三种关联关系

Web 应用程序如此强大的原因之一是它们能彼此链接和聚合信息资源。Web 组件之间存在三种关联关系：

- 请求转发
- URL 重定向
- 包含

存在以上关联关系的 Web 组件可以是 JSP 或 Servlet，对于 Struts 应用，则还包括 Action。这些 Web 组件都可以访问 HttpServletRequest 和 HttpServletResponse 对象，具有处理请求、生成响应结果的功能。本节讨论这三种关联关系的概念，以及如何使用 Java Servlet API 来实现它们。在本书后文介绍配置 Struts 的 Action 组件时将涉及这些概念。

1.2.1　请求转发

请求转发允许把请求转发给同一应用程序中的其他 Web 组件。这种技术通常用于 Web 应用控制层的 Servlet 流程控制器，它检查 HTTP 请求数据，并将请求转发到合适的目标组件，目标组件执行具体的请求处理操作，并生成响应结果。图 1-5 显示了一个 Servlet 把请求转发给另一个 JSP 组件的过程。

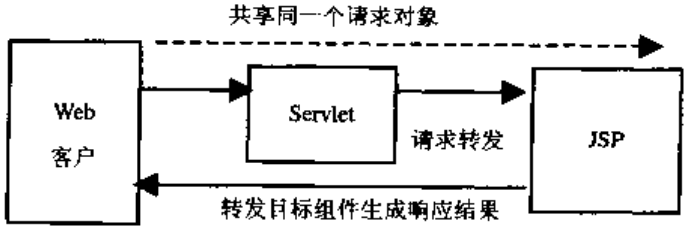

图 1-5　请求转发

Servlet类使用javax.servlet.RequestDispatcher.forward()方法来转发它所收到的HTTP 请求。转发目标组件将处理该请求并生成响应结果，或者将请求继续转发到另一个组件。最初请求的 ServletRequest 和 ServletResponse 对象被传递给转发目标组件，这使得目标组件可以访问整个请求上下文。值得注意的是，只能把请求转发给同一 Web 应用中的组件，而不能转发给其他 Web 应用的组件。

如果当前的 Servlet 组件要把请求转发给一个 JSP 组件，如 hello.jsp，可以在 Servlet 的 service()方法中执行以下代码：

```
RequestDispatcher rd =request.getRequestDispatcher("hello.jsp");
// Forward to requested URL
rd.forward(request, response);
```

在 JSP 页面中，可以使用<jsp:forward>标签来转发请求，例如：

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

对于请求转发，转发的源组件和目标组件共享 request 范围内的共享数据。

1.2.2 请求重定向

请求重定向类似于请求转发，但也有一些重要区别：

- Web 组件可以将请求重定向到任一 URL，而不仅仅是同一应用中的 URL。
- 重定向的源组件和目标组件之间不共用同一个 HttpServletRequest 对象，因此不能共享 request 范围内的共享数据。

图 1-6 显示了一个 Servlet 把请求重定向给另一个 JSP 组件的过程。

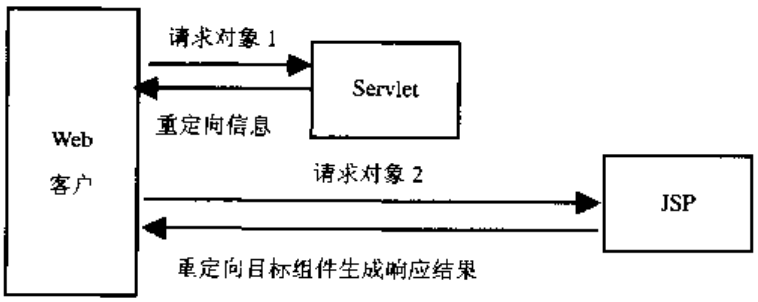

图 1-6 请求重定向

如果当前应用的 Servlet 组件要把请求转发给 URL “http://jakarta.apache.org/struts”，可以在 Servlet 的 service()方法中执行以下代码：

```
response.sendRedirect ("http://jakarta.apache.org/struts/ ");
```

从图 1-6 中可以看出，HttpServletResponse 的 sendRedirect()方法向浏览器返回包含重定向的信息，浏览器根据这一信息迅速发出一个新 HTTP 请求，请求访问重定向目标组件。

1.2.3　包含

包含关系允许一个 Web 组件聚集来自同一个应用中其他 Web 组件的输出数据，并使用被聚集的数据来创建响应结果。这种技术通常用于模板处理器，它可以控制网页的布局。模板中每个页面区域的内容来自不同的 URL，从而组成单个页面。这种技术能够为应用程序提供一致的外观和感觉。包含关系的源组件和目标组件共用同一个 HttpServletRequest 对象，因此它们共享 request 范围内的共享数据。图 1-7 显示了一个 Servlet 包含另一个 JSP 组件的过程。

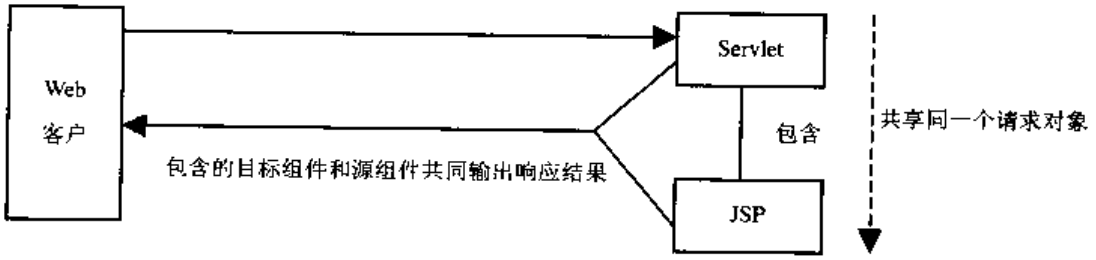

图 1-7　Web 组件的包含关系

Servlet 类使用 javax.servlet.RequestDispatcher.include()方法包含其他的 Web 组件。例如，如果当前的 Servlet 组件包含了三个 JSP 文件：header.jsp、main.jsp 和 footer.jsp，则可以在 Servlet 的 service()方法中执行以下代码：

```
RequestDispatcher rd;
rd = req.getRequestDispatcher("/header.jsp"))
rd.include(req, res);
rd = req.getRequestDispatcher("/main.jsp"))
rd.include(req, res);
rd = req.getRequestDispatcher("/footer.jsp"))
rd.include(req, res);
```

在 JSP 文件中，可以通过<include>指令来包含其他的 Web 资源，例如：

```
<%@ include file="header.jsp" %>
<%@ include file="main.jsp" %>
<%@ include file="footer.jsp" %>
```

1.3　MVC 概述

MVC 是 Model-View-Controller 的简称，即模型－视图－控制器。MVC 是 Xerox PARC 在 20 世纪 80 年代为编程语言 Smalltalk－80 发明的一种软件设计模式，至今已被广泛使用，最近几年被推荐为 Sun 公司 J2EE 平台的设计模式，受到越来越多的 Web 开发者的欢迎。

1.3.1 MVC 设计模式

MVC 是一种设计模式，它强制性地把应用程序的输入、处理和输出分开。MVC 把应用程序分成三个核心模块：模型、视图和控制器，它们分别担负不同的任务。图 1-8 显示了这几个模块各自的功能以及它们的相互关系。

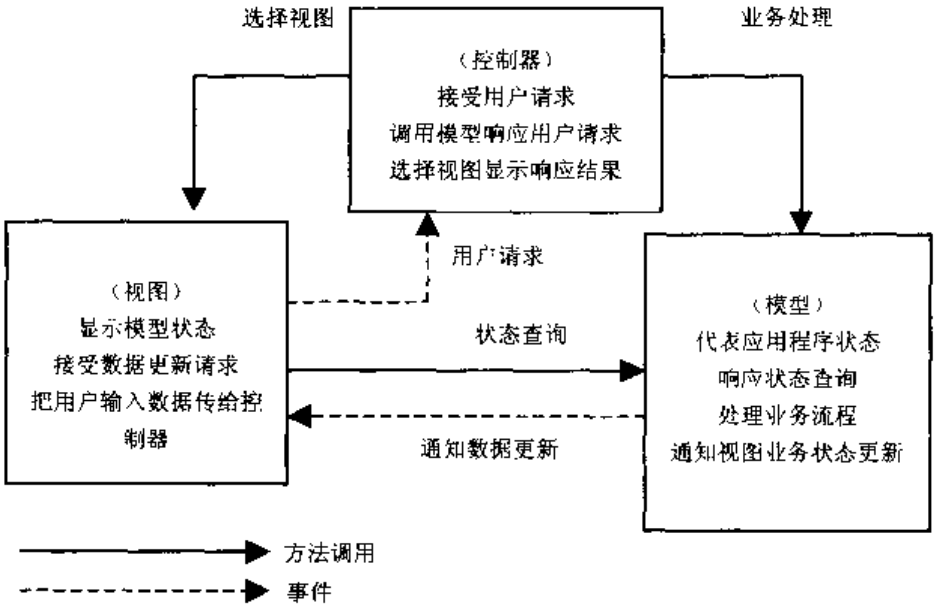

图 1-8 MVC 设计模式

1．视图

视图是用户看到并与之交互的界面。视图向用户显示相关的数据，并能接收用户的输入数据，但是它并不进行任何实际的业务处理。视图可以向模型查询业务状态，但不能改变模型。视图还能接受模型发出的数据更新事件，从而对用户界面进行同步更新。

对于基于请求/响应方式的 Web 应用，模型位于 Web 服务器端，视图位于用户浏览器端，目前无法做到模型向视图主动发出数据更新事件，使用户界面能自动刷新。

2．模型

模型是应用程序的主体部分。模型表示业务数据和业务逻辑。一个模型能为多个视图提供数据。由于同一个模型可以被多个视图重用，所以提高了应用的可重用性。

3．控制器

控制器接受用户的输入并调用模型和视图去完成用户的需求。当 Web 用户单击 Web 页面中的提交按钮来发送 HTML 表单时，控制器接收请求并调用相应的模型组件去处理请求，然后调用相应的视图来显示模型返回的数据。

4．MVC 处理过程

现在我们来总结 MVC 处理过程。首先控制器接收用户的请求，并决定应该调用哪个

模型来进行处理；然后模型根据用户请求进行相应的业务逻辑处理，并返回数据；最后控制器调用相应的视图来格式化模型返回的数据，并通过视图呈现给用户。

5．MVC 的优点

在最初的 JSP 网页中，像数据库查询语句这样的数据层代码和像 HTML 这样的表示层代码混在一起。经验比较丰富的开发者会将数据从表示层分离开来，但这通常不是很容易做到的，它需要精心地计划和不断地尝试。MVC 从根本上强制性地将它们分开。尽管构造 MVC 应用程序需要一些额外的工作，但是它给我们带来的好处是毋庸置疑的。

首先，多个视图能共享一个模型。如今，同一个 Web 应用程序会提供多种用户界面，例如用户希望既能通过浏览器来收发电子邮件，还希望通过手机来访问电子邮箱，这就要求 Web 网站同时提供 Internet 界面和 WAP 界面。在 MVC 设计模式中，模型响应用户请求并返回响应数据，视图负责格式化数据并把它们呈现给用户，业务逻辑和表示层分离，同一个模型可以被不同的视图重用，所以大大提高了代码的可重用性。

其次，模型是自包含的，与控制器和视图保持相对独立，所以可以方便地改变应用程序的数据层和业务规则。如果把数据库从 MySQL 移植到 Oracle，或者把 RDBMS 数据源改变成 LDAP 数据源，只需改变模型即可。一旦正确地实现了模型，不管数据来自数据库还是 LDAP 服务器，视图都会正确地显示它们。由于 MVC 的三个模块相互独立，改变其中一个不会影响其他两个，所以依据这种设计思想能构造良好的松耦合的构件。

此外，控制器提高了应用程序的灵活性和可配置性。控制器可以用来连接不同的模型和视图去完成用户的需求，也可以为构造应用程序提供强有力的手段。给定一些可重用的模型和视图，控制器可以根据用户的需求选择适当的模型进行处理，然后选择适当的视图将处理结果显示给用户。

6．MVC 的适用范围

使用 MVC 需要精心的计划，由于它的内部原理比较复杂，所以需要花费一些时间去理解它。将 MVC 运用到应用程序中，会带来额外的工作量，增加应用的复杂性，所以 MVC 不适合小型应用程序。

但对于开发存在大量用户界面，并且业务逻辑复杂的大型应用程序，MVC 将会使软件在健壮性、代码重用和结构方面上一个新的台阶。尽管在最初构建 MVC 框架时会花费一定的工作量，但从长远的角度来看，它会大大提高后期软件开发的效率。

1.3.2　JSP Model1 和 JSP Model2

尽管 MVC 设计模式很早就出现了，但在 Web 应用的开发中引入 MVC 却是步履维艰。主要原因是在早期的 Web 应用的开发中，程序语言和 HTML 的分离一直难以实现。例如在 JSP 网页中执行业务逻辑的程序代码和 HTML 表示层数据混杂在一起，因而很难分离出单独的业务模型。这使得维护 JSP 网页非常困难，很难满足用户的变化性需求。

在早期的 Java Web 应用中，JSP 文件负责业务逻辑、控制网页流程并创建 HTML，参见图 1-9。JSP 文件是一个独立的、自主完成所有任务的模块，这给 Web 开发带来一系列问题：

- HTML 代码和 Java 程序强耦合在一起：JSP 文件的编写者必须既是网页设计者，

又是 Java 开发者。但实际情况是，多数 Web 开发人员要么只精通网页设计，能够设计出漂亮的网页外观，但是编写的 Java 代码很糟糕；要么仅熟悉 Java 编程，能够编写健壮的 Java 代码，但是设计的网页外观很难看。皆备两种才能的开发人员很少见。

- 内嵌的流程逻辑：要理解应用程序的整个流程，必须浏览所有网页，试想一下拥有 100 个网页的网站的错综复杂的逻辑。
- 调试困难：除了很糟的外观之外，HTML 标记、Java 代码和 JavaScript 代码都集中在一个网页中，使调试变得相当困难。
- 强耦合：更改业务逻辑或数据可能牵涉相关的多个网页。
- 美学：设想有 1000 行代码的网页，其编码样式看起来杂乱无章。即使有彩色语法显示，阅读和理解这些代码仍然比较困难。

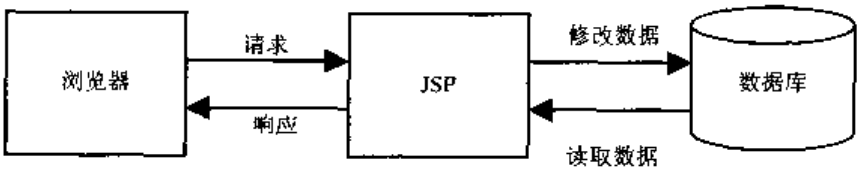

图 1-9 JSP 作为自主独立的模块

为了解决以上问题，Sun 公司先后制定了两种规范，称为 JSP Model1 和 JSP Model2。虽然 Model1 在一定程度上实现了 MVC，但是它的运用并不理想；直到基于 J2EE 的 JSP Model2 问世才得以改观。JSP Model 2 用 JSP 技术实现视图的功能，用 Servlet 技术实现控制器的功能，用 JavaBean 技术实现模型的功能。

JSP Model 1 和 JSP Model 2 的本质区别在于处理用户请求的位置不同。在 Model 1 体系中，如图 1-10 所示，JSP 页面负责响应用户请求并将处理结果返回用户。JSP 既要负责业务流程控制，又要负责提供表示层数据，同时充当视图和控制器，未能实现这两个模块之间的独立和分离。尽管 Model 1 体系十分适合简单应用的需要，它却不适合开发复杂的大型应用程序。不加选择地随意运用 Model 1，会导致 JSP 页内嵌入大量的 Java 代码。尽管这对于 Java 程序员来说可能不是什么大问题，但如果 JSP 页面是由网页设计人员开发并维护的（通常这是开发大型项目的规范），这就确实是个问题了。从根本上讲，将导致角色定义不清和职责分配不明，给项目管理带来很多麻烦。

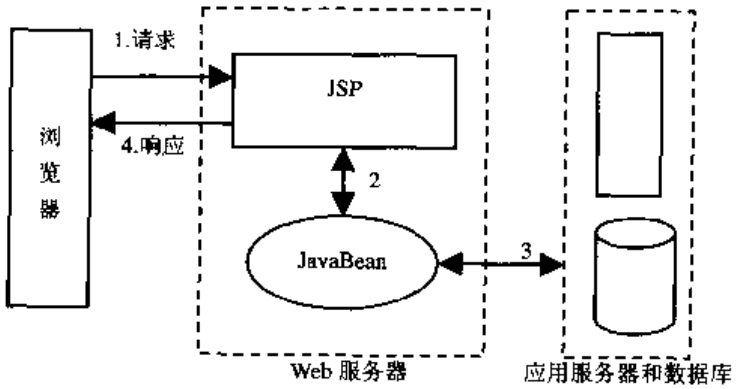

图 1-10 JSP Model1

JSP Model 2 体系结构（如图 1-11 所示）是一种联合使用 JSP 与 Servlet 来提供动态内容服务的方法。它吸取了 JSP 和 Servlet 两种技术各自的突出优点，用 JSP 生成表示层的内容，让 Servlet 完成深层次的处理任务。在这里，Servlet 充当控制器的角色，负责处理用户请求，创建 JSP 页需要使用的 JavaBean 对象，根据用户请求选择合适的 JSP 页返回给用户。在 JSP 页内没有处理逻辑，它仅负责检索原先由 Servlet 创建的 JavaBean 对象，从 Servlet 中提取动态内容插入到静态模板。这是一种有突破性的软件设计方法，它清晰地分离了表达和内容，明确了角色定义以及开发者与网页设计者的分工。事实上，项目越复杂，使用 Model 2 设计模式的好处就越大。

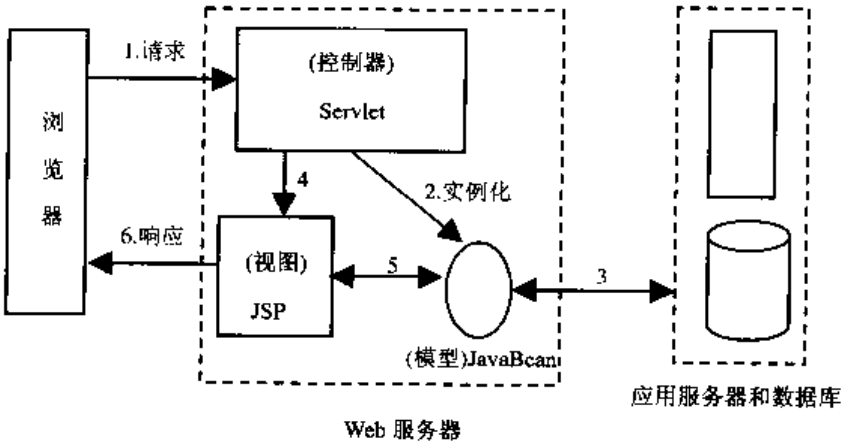

图 1-11　JSP Model2

1.4　Struts 概述

当建筑师开始一个建筑项目时，首先要设计该建筑的框架结构，有了这份蓝图，接下来的实际建筑过程才会有条不紊，井然有序。同样，软件开发者开始一个软件项目时，首先也应该构思该软件应用的框架，规划软件模块，并定义这些模块之间的接口和关系。框架可以提高软件开发的速度和效率，并且使软件更便于维护。

对于开发 Web 应用，要从头设计并开发出一个可靠、稳定的框架并不是一件容易的事。幸运的是，随着 Web 开发技术的日趋成熟，在 Web 开发领域出现了一些现成的优秀的框架，开发者可以直接使用它们，Struts 就是一种不错的选择，它是基于 MVC 的 Web 应用框架。

1.4.1　Struts 实现 MVC 的机制

Struts 实质上就是在 JSP Model2 的基础上实现的一个 MVC 框架。在 Struts 框架中，模型由实现业务逻辑的 JavaBean 或 EJB 组件构成，控制器由 ActionServlet 和 Action 来实现，视图由一组 JSP 文件构成。图 1-12 显示了 Struts 实现的 MVC 框架。

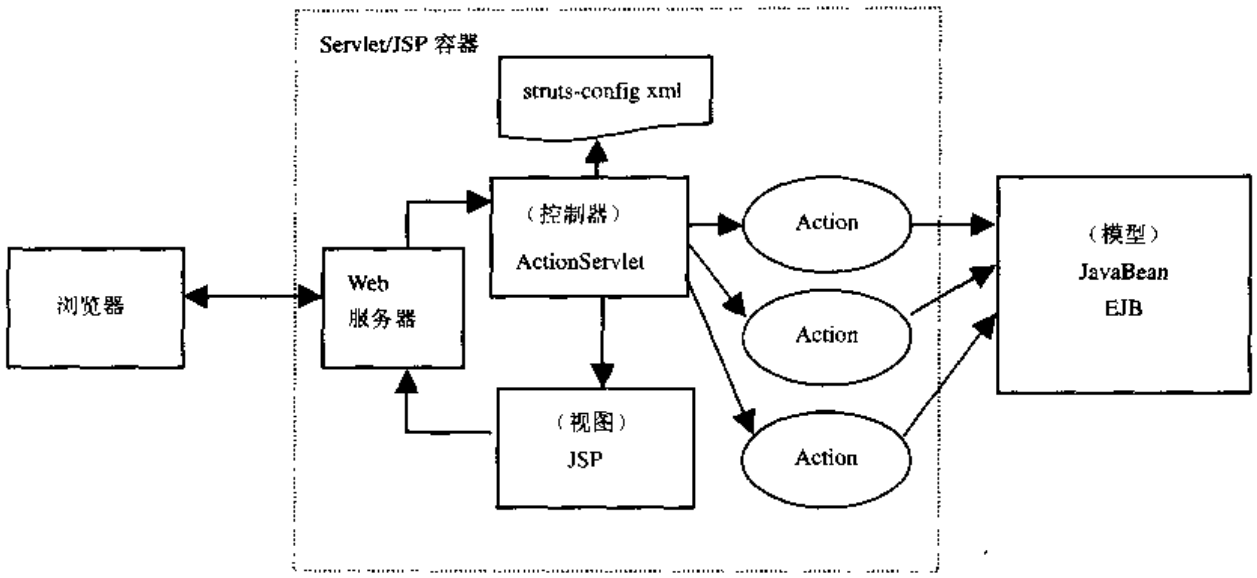

图 1-12　Struts 实现的 MVC 框架

1．视图

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 标签或客户化标签，如 Struts 标签库中的标签。

此外，通常把 Struts 框架中的 ActionForm Bean 也划分到视图模块中。ActionForm Bean 也是一种 JavaBean，除了具有一些 JavaBean 的常规方法，还包含一些特殊的方法，用于验证 HTML 表单数据以及将其属性重新设置为默认值。Struts 框架利用 ActionForm Bean 来进行视图和控制器之间表单数据的传递，参见图 1-13。Struts 框架把用户输入的表单数据保存在 ActionForm Bean 中，把它传递给控制器，控制器可以对 ActionForm Bean 中的数据进行修改，JSP 文件使用 Struts 标签读取修改后的 ActionForm Bean 的信息，重新设置 HTML 表单。

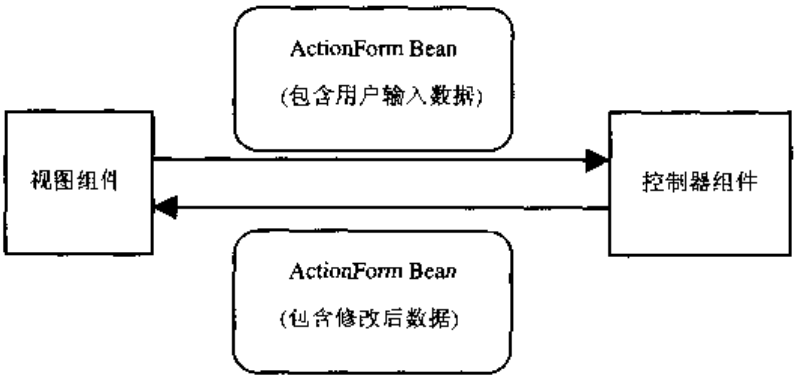

图 1-13　ActionForm Bean 的作用

2．模型

模型表示应用程序的状态和业务逻辑。对于大型应用，业务逻辑通常由 JavaBean 或 EJB 组件来实现。

3．控制器

控制器由 ActionServlet 类和 Action 类来实现。ActionServlet 类是 Struts 框架中的核心组件。ActionServlet 继承了 javax.servlet.http.HttpServlet 类，它在 MVC 模型中扮演中央控制器的角色。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 对象不存在，ActionServlet 会先创建这个 Action 对象。

Action 类负责调用模型的方法，更新模型的状态，并帮助控制应用程序的流程。对于小型简单的应用，Action 类本身也可以完成一些实际的业务逻辑。

对于大型应用，Action 充当用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 根据用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 类侧重于控制应用程序的流程，而不是实现应用程序的逻辑。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 控制器收到用户请求后，把请求转发到一个 Action 实例。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。Action 的 execute()方法返回 ActionForward 对象，它封装了把用户请求再转发给其他 Web 组件的信息。用户定义自己的 Action 类，即 Action 基类的子类时，必须覆盖 execute()方法，在 Action 基类中该方法返回 null。

4．Struts 的配置文件 struts-config.xml

上面讲到一个用户请求是通过 ActionServlet 来处理和转发的。那么，ActionServlet 如何决定把用户请求转发给哪个 Action 对象呢？这就需要一些描述用户请求路径和 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 struts-config.xml 中。在该配置文件中，每一个 Action 的映射信息都通过一个<action>元素来配置。

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。

1.4.2　Struts 的工作流程

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 ActionServlet，ActionServlet 从 struts-config.xml 文件中读取配置信息，把它们存放到各种配置对象中，例如 Action 的映射信息存放在 ActionMapping 对象中。

当 ActionServlet 接收到一个客户请求时，将执行如下流程。

（1）检索和用户请求匹配的 ActionMapping 实例，如果不存在，就返回用户请求路径无效的信息。

（2）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。

（3）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。

（4）如果 ActionForm 的 validate()方法返回 null 或返回一个不包含 ActionMessage 的 ActionErrors 对象，就表示表单验证成功。

（5）ActionServlet 根据 ActionMapping 实例包含的映射信息决定将请求转发给哪个 Action。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。

（6）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。

（7）ActionForward 对象指向的 JSP 组件生成动态网页，返回给客户。

对于以上流程的流程（4），如果 ActionForm 的 validate()方法返回一个包含一个或多个 ActionMessage 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用户提交表单的 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。

图 1-14 显示了 Struts 响应用户请求的工作流程。

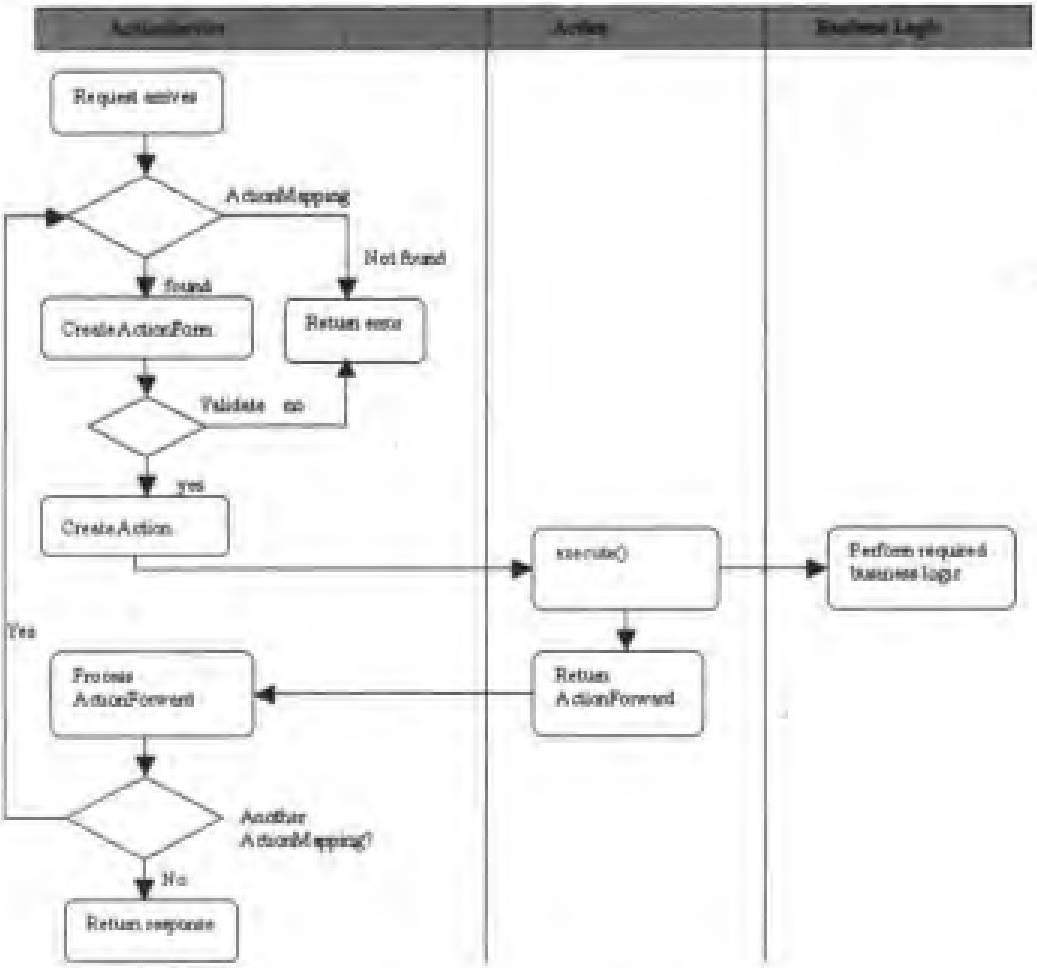

图 1-14 Struts 响应用户请求的工作流程

1.5 小 结

本章首先回顾了开发 Java Web 应用涉及的各种技术，侧重介绍了开发 Struts 应用所需的知识，如共享数据的 4 种范围：page、request、session 和 application，以及 Web 组件的三种关联关系：请求转发、重定向和包含。

本章接着介绍了 MVC 设计模式的结构和优点，MVC 把应用程序分成三个核心模块：模型、视图、控制器，能够提高应用的可重用性和可扩展性，从而提高开发大型复杂软件系统的效率。然后本章介绍了 Sun 公司提出的在 Java Web 开发领域的两种规范：JSP Model 1 和 JSP Model 2，最后介绍了 Struts 实现 MVC 的机制。Struts 实质上就是在 JSP Model 2 的基础上实现的一个 MVC 框架。在 Struts 框架中，模型由实现业务逻辑的 JavaBean 或 EJB 组件构成，控制器由 ActionServlet 和 Action 来实现，视图由一组 JSP 文件构成。

下面再解释一下本书涉及的几个术语：

- Struts 应用：在本书中，把基于 Struts 框架的 Java Web 应用简称为 Struts 应用。
- Servlet 容器、JSP 容器和 Java Web 容器：三者的字面含义不一样。Servlet 容器，顾名思义，是指运行 Servlet 的容器，而 Java Web 容器指的是运行 Java Web 的容器。但对于实际的 Java Web 容器产品，如 Tomcat，肯定同时也是 Servlet/JSP 容器。因此读者可以把它们理解为同一种容器。

第 2 章　Struts 应用：helloapp 应用

本章讲解了一个简单的 Struts 应用例子——helloapp 应用，这个例子可以帮助读者迅速入门，获得开发 Struts 应用的基本经验。该应用的功能非常简单：接受用户输入的姓名<name>，然后输出“Hello <name>”。开发 helloapp 应用涉及以下内容：

- 分析应用需求。
- 把基于 MVC 设计模式的 Struts 框架运用到应用中。
- 创建视图组件，包括 HTML 表单（hello.jsp）和 ActionForm Bean（HelloForm.java）。
- 创建 application.properties 资源文件。
- 数据验证，包括表单验证和业务逻辑验证。
- 创建控制器组件：HelloAction.java。
- 创建模型组件：PersonBean.java。
- 创建包含被各个模块共享的常量数据的 Java 文件：Constants.java。
- 创建配置文件：web.xml 和 struts-config.xml。
- 编译、发布和运行 helloapp 应用。

2.1　分析 helloapp 应用的需求

在开发应用时，首先从分析需求入手，列举该应用的各种功能，以及限制条件。helloapp 应用的需求非常简单，其包括如下需求：

- 接受用户输入的姓名<name>，然后返回字符串“Hello <name> !”。
- 如果用户没有输入姓名就提交表单，将返回出错信息，提示用户首先输入姓名。
- 如果用户输入姓名为“Monster”，将返回出错信息，拒绝向“Monster”打招呼。
- 为了演示模型组件的功能，本应用使用模型组件来保存用户输入的姓名。

2.2　运用 Struts 框架

下面把 Struts 框架运用到 helloapp 应用中。Struts 框架可以方便迅速地把一个复杂的应用划分成模型、视图和控制器组件，而 Struts 的配置文件 struts-config.xml 则可以灵活地组装这些组件，简化开发过程。

以下是 helloapp 应用的各个模块的构成：

- 模型包括一个 JavaBean 组件 PersonBean，它有一个 userName 属性，代表用户输入的名字。它提供了 get/set 方法，分别用于读取和设置 userName 属性，它还提供一个 save()方法，负责把 userName 属性保存到持久化存储系统中，如数据库或

文件系统。对于更为复杂的 Web 应用，JavaBean 组件可以作为 EJB 或 Web 服务的前端组件。

- 视图包括一个 JSP 文件 hello.jsp，它提供用户界面，接受用户输入的姓名。视图还包括一个 ActionForm Bean，它用来存放表单数据，并进行表单验证，如果用户没有输入姓名就提交表单，将返回出错信息。
- 控制器包括一个 Action 类 HelloAction，它完成三项任务：一是进行业务逻辑验证，如果用户输入的姓名为“Monster”，将返回错误消息；二是调用模型组件 PersonBean 的 save()方法，保存用户输入的名字；三是决定将合适的视图组件返回给用户。

除了创建模型、视图和控制器组件，还需要创建 Struts 的配置文件 struts-config.xml，它可以把这些组件组装起来，使它们协调工作。此外，还需要创建整个 Web 应用的配置文件 web.xml。

2.3 创建视图组件

在本例中，视图包括两个组件：

- 一个 JSP 文件：hello.jsp。
- 一个 ActionForm Bean：HelloForm Bean。

下面分别讲述如何创建这两个组件。

2.3.1 创建 JSP 文件

hello.jsp 提供用户界面，能够接受用户输入的姓名。此外，本 Web 应用的所有输出结果也都由 hello.jsp 显示给用户。图 2-1 显示了 hello.jsp 提供的网页。

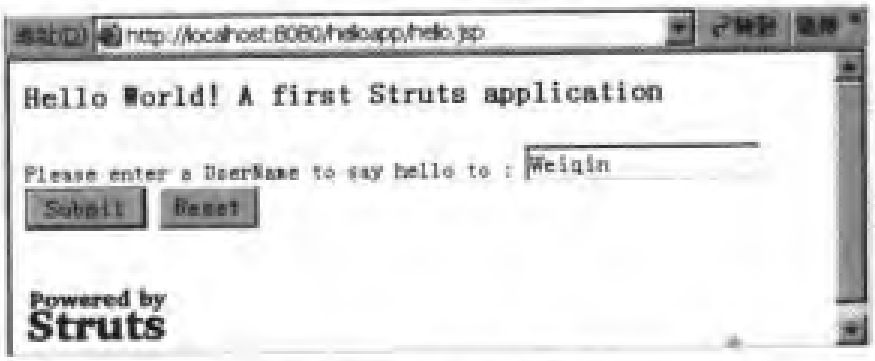

图 2-1 hello.jsp 的网页

在图 2-1 中，当用户输入姓名“Weiqin”后，单击【Submit】按钮提交表单，本应用将返回“Hello Weiqin!”，参见图 2-2。

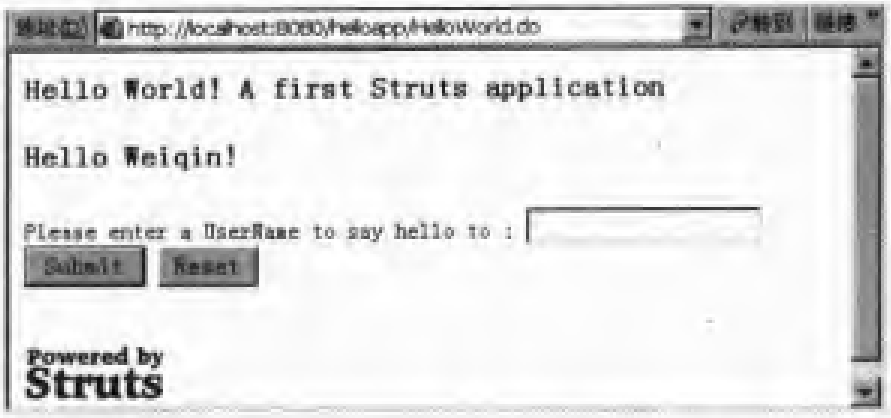

图 2-2　hello.jsp 接受用户输入后正常返回的网页

例程 2-1 为 hello.jsp 文件的源代码。

例程 2-1　hello.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html:html locale="true">
  <head>
    <title><bean:message key="hello.jsp.title"/></title>
    <html:base/>
  </head>

  <body bgcolor="white"><p>
    <h2><bean:message key="hello.jsp.page.heading"/></h2><p>
   <html:errors/><p>

    <logic:present name="personbean" scope="request">
      <h2>
        <bean:message key="hello.jsp.page.hello"/>
        <bean:write name="personbean" property="userName" />!<p>
      </h2>
    </logic:present>

    <html:form action="/HelloWorld.do" focus="userName" >
      <bean:message key="hello.jsp.prompt.person"/>
      <html:text property="userName" size="16" maxlength="16"/><br>
      <html:submit property="submit" value="Submit"/>
      <html:reset/>

    </html:form><br>
```

```
        <html:img page="/struts-power.gif" alt="Powered by Struts"/>
    </body>
</html:html>
```

以上基于 Struts 框架的 JSP 文件有以下特点：

- 没有任何 Java 程序代码。
- 使用了许多 Struts 的客户化标签，例如<html:form>和<logic:present>标签。
- 没有直接提供文本内容，取而代之的是<bean:message>标签，输出到网页上的文本内容都是由<bean:message>标签来生成的。例如：

```
<bean:message key="hello.jsp.prompt.person"/>
```

Struts 客户化标签是联系视图组件和 Struts 框架中其他组件的纽带。这些标签可以访问或显示来自于控制器和模型组件的数据。在本书第 12 章至第 16 章将专门介绍 Struts 标签的用法，本节先简单介绍几种重要的 Struts 标签。

hello.jsp 开头几行用于声明和加载 Struts 标签库：

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
```

以上代码表明该 JSP 文件使用了 Struts Bean、Html 和 Logic 标签库，这是加载客户化标签库的标准 JSP 语法。

hello.jsp 中使用了来自 Struts HTML 标签库中的标签，包括<html:errors>, <html:form>和<html:text>：

- <html:errors>：用于显示 Struts 框架中其他组件产生的错误消息。
- <html:form>：用于创建 HTML 表单，它能够把 HTML 表单的字段和 ActionForm Bean 的属性关联起来。
- <html:text>：该标签是<html:form>的子标签，用于创建 HTML 表单的文本框。它和 ActionForm Bean 的属性相关联。

hello.jsp 中使用了来自 Struts Bean 标签库的两个标签<bean:message>和<bean:write>：

- <bean:message>：用于输出本地化的文本内容，它的 key 属性指定消息 key，与消息 key 匹配的文本内容来自于专门的 Resource Bundle，关于 Resource Bundle 的概念参见本书第 9 章（Struts 应用的国际化）。
- <bean:write>：用于输出 JavaBean 的属性值。本例中，它用于输出 personbean 对象的 userName 属性值：

```
<bean:write name="personbean" property="userName" />
```

hello.jsp 使用了来自 Struts Logic 标签库的<logic:present>标签。<logic:present>标签用来判断 JavaBean 在特定的范围内是否存在，只有当 JavaBean 存在时，才会执行标签主体中的内容：

```
<logic:present name="personbean" scope="request">
        <h2>
```

```
        Hello <bean:write name="personbean" property="userName" />!<p>
      </h2>
    </logic:present>
```

在本例中，<logic:present>标签用来判断在 request 范围内是否存在 personbean 对象，如果存在，就输出 personbean 的 userName 属性值。与<logic:present>标签相对的是<logic:notPresent>标签，它表示只有当 JavaBean 在特定的范围内不存在时，才会执行标签主体中的内容。

2.3.2 创建消息资源文件

hello.jsp 使用<bean:message>标签来输出文本内容。这些文本来自于 Resource Bundle，每个 Resource Bundle 都对应一个或多个本地化的消息资源文件，本例中的资源文件为 application.properties，例程 2-2 是该消息资源文件的内容。

例程 2-2　application.properties 文件

```
#Application Resources for the "Hello" sample application
hello.jsp.title=Hello - A first Struts program
hello.jsp.page.heading=Hello World! A first Struts application
hello.jsp.prompt.person=Please enter a UserName to say hello to :
hello.jsp.page.hello=Hello

#Validation and error messages for HelloForm.java and HelloAction.java
hello.dont.talk.to.monster=We don't want to say hello to Monster!!!
hello.no.username.error=Please enter a <i>UserName</i> to say hello to!
```

以上文件以“消息 key/消息文本”的格式存放数据，文件中“#”的后面为注释行。对于以下 JSP 代码：

```
<bean:message key="hello.jsp.title"/>
```

<bean:message>标签的 key 属性为“hello.jsp.tilte”，在 Resource Bundle 中与之匹配的内容为：

```
hello.jsp.title=Hello - A first Struts program
```

因此，以上<bean:message>标签将把“Hello - A first Struts program”输出到网页上。

2.3.3 创建 ActionForm Bean

当用户提交了 HTML 表单后，Struts 框架将自动把表单数据组装到 ActionForm Bean 中。ActionForm Bean 中的属性和 HTML 表单中的字段一一对应。ActionForm Bean 还提供数据验证方法，以及把属性重新设置为默认值的方法。Struts 框架中定义的 ActionForm 类是抽象的，必须在应用中创建它的子类，来存放具体的 HTML 表单数据。例程 2-3 为 HelloForm.java 的源程序，它用于处理 hello.jsp 中的表单数据。

例程 2-3 HelloForm.java

```
package hello;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public final class HelloForm extends ActionForm {

    private String userName = null;

    public String getUserName() {
        return (this.userName);
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    /**
     * Reset all properties to their default values.
     */
    public void reset(ActionMapping mapping, HttpServletRequest request) {
        this.userName = null;
    }

    /**
     * Validate the properties posted in this request. If validation errors are
     * found, return an <code>ActionErrors</code> object containing the errors.
     * If no validation errors occur, return <code>null</code> or an empty
     * <code>ActionErrors</code> object.
     */
    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request) {

        ActionErrors errors = new ActionErrors();

        if ((userName == null) || (userName.length() < 1))
            errors.add("username", new ActionMessage("hello.no.username.error"));

        return errors;
    }
}
```

从以上代码中可以看出，ActionForm Bean 实质上是一种 JavaBean，不过它除了具有 JavaBean 的常规方法，还有两种特殊方法：

- validate()：用于表单验证。
- reset()：把属性重新设置为默认值。

2.3.4 数据验证

几乎所有和用户交互的应用都需要数据验证，而从头设计并开发完善的数据验证机制往往很费时。幸运的是，Struts 框架提供了现成的、易于使用的数据验证功能。Struts 框架的数据验证可分为两种类型：表单验证和业务逻辑验证，在本例中，它们分别运用于以下场合：

- 表单验证：如果用户没有在表单中输入姓名就提交表单，将生成表单验证错误。
- 业务逻辑验证：如果用户在表单中输入的姓名为“Monster”，按照本应用的业务规则，即不允许向“Monster”打招呼，因此将生成业务逻辑错误。

第一种类型的验证，即表单验证由 ActionForm Bean 来负责处理。在本例中，HelloForm.java 的 validate()方法负责完成这一任务：

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();

    if ((userName == null) || (userName.length() < 1))
        errors.add("username", new ActionMessage("hello.no.username.error"));

    return errors;
  }
}
```

当用户提交了 HTML 表单后，Struts 框架将自动把表单数据组装到 ActionForm Bean 中。接下来 Struts 框架会自动调用 ActionForm Bean 的 validate()方法进行表单验证。如果 validate()方法返回的 ActionErrors 对象为 null，或者不包含任何 ActionMessage 对象，就表示没有错误，数据验证通过。如果 ActionErrors 中包含 ActionMessage 对象，就表示发生了验证错误，Struts 框架会把 ActionErrors 对象保存到 request 范围内，然后把请求转发到恰当的视图组件，视图组件通过<html:errors>标签把 request 范围内的 ActionErrors 对象中包含的错误消息显示出来，提示用户修改错误。

在 Struts 早期的版本中使用 ActionError 类来表示错误消息，ActionError 类是 ActionMessage 的子类。Struts 1.2 将废弃 ActionError，统一采用 ActionMessage 类来表示正常或错误消息。

第二种类型的验证，即业务逻辑验证由 Action 来负责处理，参见本章的 2.4.3 小节。

2.4 创建控制器组件

控制器组件包括 ActionServlet 类和 Action 类。ActionServlet 类是 Struts 框架自带的，它是整个 Struts 框架的控制枢纽，通常不需要扩展。Struts 框架提供了可供扩展的 Action 类，它用来处理特定的 HTTP 请求，例程 2-4 为 HelloAction 类的源程序。

例程 2-4 HelloAction.java

```
package hello;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.util.MessageResources;

public final class HelloAction extends Action {

    /**
     * Process the specified HTTP request, and create the corresponding HTTP
     * response (or forward to another web component that will create it).
     * Return an <code>ActionForward</code> instance describing where and how
     * control should be forwarded, or <code>null</code> if the response has
     * already been completed.
     */
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
    throws Exception {

        // These "messages" come from the ApplicationResources.properties file
        MessageResources messages = getResources(request);

        /*
         * Validate the request parameters specified by the user
```

```
        * Note: Basic field validation done in HelloForm.java
        *       Business logic validation done in HelloAction.java
        */
        ActionMessages errors = new ActionMessages();
        String userName = (String)((HelloForm) form).getUserName();

        String badUserName = "Monster";

        if (userName.equalsIgnoreCase(badUserName)) {
            errors.add("username", new ActionMessage("hello.dont.talk.to.monster",
             badUserName ));
            saveErrors(request, errors);
            return (new ActionForward(mapping.getInput()));
        }

        /*
         * Having received and validated the data submitted
         * from the View, we now update the model
         */
        PersonBean pb = new PersonBean();
        pb.setUserName(userName);
        pb.saveToPersistentStore();

        /*
         * If there was a choice of View components that depended on the model
         * (or some other) status, we'd make the decision here as to which
         * to display. In this case, there is only one View component.
         *
         * We pass data to the View components by setting them as attributes
         * in the page, request, session or servlet context. In this case, the
         * most appropriate scoping is the "request" context since the data
         * will not be neaded after the View is generated.
         *
         * Constants.PERSON_KEY provides a key accessible by both the
         * Controller component (i.e. this class) and the View component
         * (i.e. the jsp file we forward to).
         */

        request.setAttribute( Constants.PERSON_KEY, pb);

        // Remove the Form Bean - don't need to carry values forward
        request.removeAttribute(mapping.getAttribute());

        // Forward control to the specified success URI
        return (mapping.findForward("SayHello"));
    }
```

HelloAction.java 是本应用中最复杂的程序，下面分步讲解它的工作机制和流程。

2.4.1 Action 类的工作机制

所有的 Action 类都是 org.apache.struts.action.Action 的子类。Action 子类应该覆盖父类的 execute()方法。当 ActionForm Bean 被创建，并且表单验证顺利通过后， Struts 框架就会调用 Action 类的 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()方法包含以下参数：

- ActionMapping：包含了这个 Action 的配置信息，和 struts-config.xml 文件中的<action>元素对应。
- ActionForm：包含了用户的表单数据，当 Struts 框架调用 execute()方法时，ActionForm 中的数据已经通过了表单验证。
- HttpServletRequest：当前的 HTTP 请求对象。
- HttpServletResponse：当前的 HTTP 响应对象。

Action 类的 execute()方法返回 ActionForward 对象，它包含了请求转发路径信息。

2.4.2 访问封装在 MessageResources 中的本地化文本

在本例中，Action 类的 execute()方法首先获得 MessageResources 对象：

```
MessageResources messages = getResources(request);
```

在 Action 类中定义了 getResources(HttpServletRequest request)方法，该方法返回当前默认的 MessageResources 对象，它封装了 Resource Bundle 中的文本内容。接下来 Action 类就可以通过 MessageResources 对象来访问文本内容。例如，如果要读取消息 key 为“hello.jsp.title”对应的文本内容，可以调用 MessageResources 类的 getMessage(String key)方法：

```
String title=messages.getMessage("hello.jsp.title");
```

2.4.3 业务逻辑验证

接下来，Action 类的 execute()方法执行业务逻辑验证：

```
ActionMessages errors = new ActionMessages();
String userName = (String)((HelloForm) form).getUserName();
```

```
String badUserName = "Monster";

if (userName.equalsIgnoreCase(badUserName)) {
    errors.add("username", new ActionMessage("hello.dont.talk.to.monster", badUserName ));
    saveErrors(request, errors);
    return (new ActionForward(mapping.getInput()));
}
```

如果用户输入的姓名为“Monster”，将创建包含错误信息的 ActionMessage 对象，ActionMessage 对象被保存到 ActionMessages 对象中。接下来调用在 Action 基类中定义的 saveErrors()方法，它负责把 ActionMessages 对象保存到 request 范围内。最后返回 ActionForward 对象，Struts 框架会根据 ActionForward 对象包含的转发信息把请求转发到恰当的视图组件，视图组件通过<html:errors>标签把 request 范围内的 ActionMessages 对象中包含的错误消息显示出来，提示用户修改错误。

在 2.3.4 小节中还提到了 ActionErrors 对象。图 2-3 显示了 ActionMessages、ActionErrors、ActionMessage 和 ActionError 类的类框图。ActionErrors 继承 ActionMessages，ActionError 继承 ActionMessage，ActionMessages 和 ActionMessage 之间为聚集关系，即一个 ActionMessages 对象中可以包含多个 ActionMessage 对象。

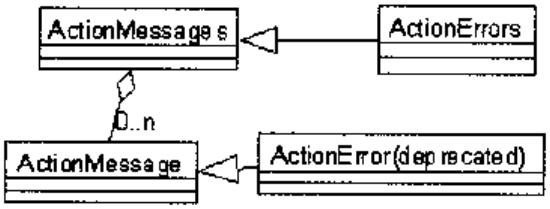

图 2-3　ActionMessages、ActionErrors、ActionMessage 和 ActionError 类的类框图

表单验证通常只对用户输入的数据进行简单的语法和格式检查，而业务逻辑验证会对数据进行更为复杂的验证。在很多情况下，需要模型组件的介入，才能完成业务逻辑验证。

2.4.4　访问模型组件

接下来，HelloAction 类创建了一个模型组件 PersonBean 对象，并调用它的 saveToPersistentStore()方法来保存 userName 属性：

```
PersonBean pb = new PersonBean();
pb.setUserName(userName);
pb.saveToPersistentStore();
```

本例仅提供了 Action 类访问模型组件简单的例子。在实际应用中，Action 类会访问模型组件，完成更加复杂的功能，例如：

- 从模型组件中读取数据，用于被视图组件显示。
- 和多个模型组件交互。
- 依据从模型组件中获得的信息，来决定返回哪个视图组件。

2.4.5 向视图组件传递数据

Action 类把数据存放在 request 或 session 范围内，以便向视图组件传递信息。以下是 HelloAction.java 向视图组件传递数据的代码：

```
request.setAttribute( Constants.PERSON_KEY, pb);

// Remove the Form Bean - don't need to carry values forward
request.removeAttribute(mapping.getAttribute());
```

以上代码完成两件事：

- 把 PersonBean 对象保存在 request 范围内。
- 从 request 范围内删除 ActionForm Bean。由于后续的请求转发目标组件不再需要 HelloForm Bean，所以可将它删除。

2.4.6 把 HTTP 请求转发给合适的视图组件

最后，Action 类把流程转发给合适的视图组件。

```
// Forward control to the specified success URI
return (mapping.findForward("SayHello"));
```

2.5 创建模型组件

在 2.4 节中已经讲过，Action 类会访问模型组件。本例中模型组件为 JavaBean: PersonBean。例程 2-5 是 PersonBean 的源代码。

例程 2-5 PersonBean.java

```
package hello;
public class PersonBean {

    private String userName = null;

    public String getUserName() {
        return this.userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }

    /**
     * This is a stub method that would be used for the Model to save
     * the information submitted to a persistent store. In this sample
```

```
     * application it is not used.
     */
    public void saveToPersistentStore() {

        /*
         * This is a stub method that might be used to save the person's
         * name to a persistent store(i.e. database) if this were a real application.
         *
         * The actual business operations that would exist within a Model
         * component would depend upon the requirements of the application.
         */
    }
}
```

PersonBean 是一个非常简单的 JavaBean，它包括一个 userName 属性，以及相关的 get/set 方法。此外，它还有一个业务方法 saveToPersistentStore()。本例中并没有真正实现这一方法。在实际应用中，这个方法可以用来把 JavaBean 的属性保存在持久化存储系统中，如数据库或文件系统。

通过这个简单的例子，读者可以进一步理解 Struts 框架中使用模型组件的一大优点，它把业务逻辑的实现和应用的其他部分分离开来，可以提高整个应用的灵活性、可重用性和可扩展性。如果模型组件的实现发生改变，例如本来把 JavaBean 的属性保存在 MySQL 数据库中，后来改为保存在 Oracle 数据库中，此时 Action 类不需要做任何变动。不仅如此，即使模型组件由 JavaBean 改为 EJB，运行在远程应用服务器上，也不会对 Action 类造成任何影响。

2.6　创建存放常量的 Java 文件

根据 2.4.5 小节，HelloAction 类和视图组件之间通过 HttpServletRequest 的 setAttribute() 和 getAttribute()方法来共享 request 范围内的数据。下面再看一下 HelloAction 类调用 HttpServletRequest 的 setAttribute()方法的细节。

当 HelloAction 类调用 HttpServletRequest 的 setAttribute()方法，向 hello.jsp 传递 PersonBean 对象时，需要提供一个名为“personbean”的属性 key：

```
request.setAttribute("personbean",pb);
```

hello.jsp 通过这个名为“personbean”的属性 key 来读取 PersonBean 对象：

```
<logic:present name="personbean" scope="request">
    <h2>
      Hello <bean:write name="personbean" property="userName" />!<p>
    </h2>
</logic:present>
```

对于 Struts 应用，提倡将这些属性 key 常量定义在一个 Java 文件 Constants.java 中，例

程 2-6 显示了它的源程序。

例程 2-6 Constants.java

```
package hello;
public final class Constants {
    /**
     * The application scope attribute under which our user database
     * is stored.
     */
    public static final String PERSON_KEY = "personbean";
}
```

这样，HelloAction 类可以按以下方式来调用 HttpServletRequest 的 setAttribute()方法：

```
request.setAttribute( Constants.PERSON_KEY, pb);
```

把一些常量定义在 Constants.java 中可以提高 Action 类的独立性。当属性 key 常量值发生改变时，只需要修改 Constants.java 文件，而不需要修改 Action 类。

此外，本例把 PersonBean 对象保存在 HttpServletRequest 对象中。对于其他实际的 Web 应用，也可以根据需要把 JavaBean 对象保存在 HttpSession 对象中。

2.7 创建配置文件

2.7.1 创建 Web 应用的配置文件

对于 Struts 应用，它的配置文件 web.xml 应该对 ActionServlet 类进行配置。此外，还应该声明 Web 应用所使用的 Struts 标签库，本例中声明使用了三个标签库：Struts Bean、Struts HTML 和 Struts Logic 标签库。例程 2-7 为 web.xml 的源代码。

例程 2-7 web.xml

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <display-name>HelloApp Struts Application</display-name>

 <!-- Standard Action Servlet Configuration  -->
  <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>
     <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>

  <!-- The Usual Welcome File List -->
  <welcome-file-list>
    <welcome-file>hello.jsp</welcome-file>
  </welcome-file-list>

  <!-- Struts Tag Library Descriptors -->
  <taglib>
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
  </taglib>

</web-app>
```

2.7.2　创建 Struts 框架的配置文件

正如前面提及的，Struts 框架允许把应用划分成多个组件，提高开发速度。而 Struts 框架的配置文件 struts-config.xml 可以把这些组件组装起来，决定如何使用它们。例程 2-8 是 helloapp 应用的 struts-config.xml 文件的源代码。

例程 2-8　struts-config.xml

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

```
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<!--
     This is the Struts configuration file for the "Hello!" sample application
-->

<struts-config>

    <!-- ======== Form Bean Definitions ================================== -->
    <form-beans>
        <form-bean name="HelloForm" type="hello.HelloForm"/>
    </form-beans>

  <!-- ========== Action Mapping Definitions ============================== -->
  <action-mappings>
    <!-- Say Hello! -->
    <action      path        = "/HelloWorld"
                 type        = "hello.HelloAction"
                 name        = "HelloForm"
                 scope       = "request"
                 validate    = "true"
                 input       = "/hello.jsp"
     >
        <forward name="SayHello" path="/hello.jsp" />
    </action>
  </action-mappings>

  <!-- ========== Message Resources Definitions =========================== -->
  <message-resources parameter="hello.application"/>

</struts-config>
```

以上代码对 helloapp 应用的 HelloForm、HelloAction 和消息资源文件进行了配置，首先通过<form-bean>元素配置了一个 ActionForm Bean，名叫 HelloForm，它对应的类为 hello.HelloForm：

```
<form-bean name="HelloForm" type="hello.HelloForm"/>
```

接着通过<action>元素配置了一个 Action 组件：

```
<action      path        = "/HelloWorld"
             type        = "hello.HelloAction"
             name        = "HelloForm"
             scope       = "request"
             validate    = "true"
             input       = "/hello.jsp"
```

```
>
        <forward name="SayHello" path="/hello.jsp" />
</action>
```

<action>元素的 path 属性指定请求访问 Action 的路径，type 属性指定 Action 的完整类名，name 属性指定需要传递给 Action 的 ActionForm Bean，scope 属性指定 ActionForm Bean 的存放范围，validate 属性指定是否执行表单验证，input 属性指定当表单验证失败时的转发路径。<action>元素还包含一个<forward>子元素，它定义了一个请求转发路径。

本例中的<action>元素配置了 HelloAction 组件，对应的类为 hello.HelloAction，请求访问路径为“HelloWorld”，当 Action 类被调用时，Struts 框架应该把已经包含表单数据的 HelloForm Bean 传给它。HelloForm Bean 存放在 request 范围内，并且在调用 Action 类之前，应该进行表单验证。如果表单验证失败，请求将被转发到接收用户输入的网页 hello.jsp，让用户纠正错误。

struts-config.xml 文件最后通过<message-resources>元素定义了一个 Resource Bundle：

```
<message-resources parameter="hello.application"/>
```

<message-resources>元素的 parameter 属性指定 Resource Bundle 使用的消息资源文件。本例中 parameter 属性为“hello.application”，表明消息资源文件名为“application.properties”，它的存放路径为 WEB-INF/classes/hello/application.properties。

2.8　发布和运行 helloapp 应用

helloapp 应用作为 Java Web 应用，它的目录结构应该符合 Sun 公司制定的 Java Web 应用的规范，此外，由于 helloapp 应用使用了 Struts 框架，因此应该把 Struts 框架所需的 JAR 文件和标签库描述文件 TLD 文件包含进来。访问 http://jakarta.apache.org/builds，可以下载最新的 Struts 软件包，把 struts 压缩文件解压后，在其 lib 子目录下提供了 Struts 框架所需的 JAR 文件：

- commons-beanutils.jar
- commons-collections.jar
- commons-digester.jar
- commons-fileupload.jar
- commons-logging.jar
- commons-validator.jar
- jakarta-oro.jar
- struts.jar

在 Struts 软件包的 lib 子目录下还提供了所有的 Struts 标签库描述 TLD 文件：

- struts-bean.tld
- struts-html.tld
- struts-logic.tld
- struts-nested.tld

- struts-tiles.tld

图 2-4 显示了 helloapp 应用的目录结构。

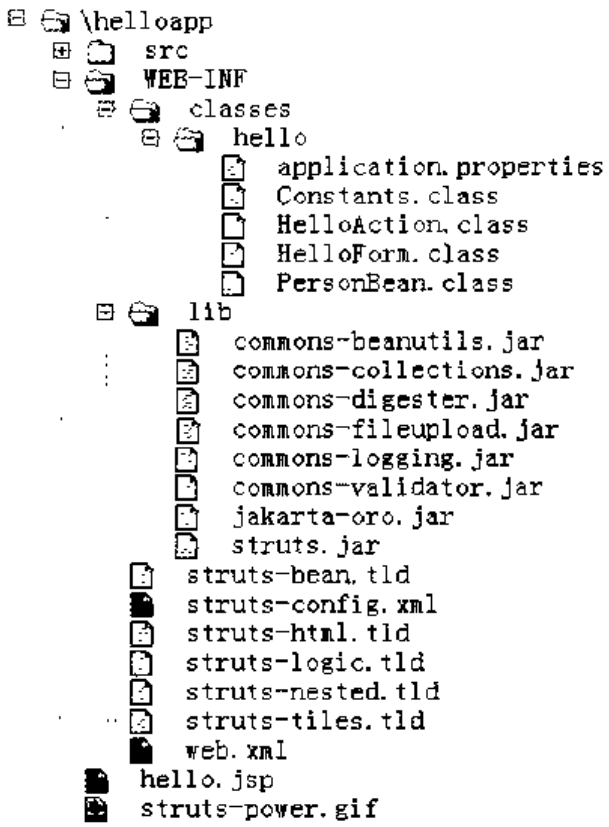

图 2-4 helloapp 应用的目录结构

helloapp 应用的 Java 源文件位于 helloapp/src 目录下，编译这些 Java 源文件时，应该把 Servlet API 的 JAR 文件以及 Struts 的 struts.jar 文件加到 classpath 中。如果在本地安装了 Tomcat 服务器，假定 Tomcat 的根目录为<CATALINA_HOME>，在<CATALINA_HOME>\common\lib 目录下提供了 servlet-api.jar 文件。

在本书配套光盘的 sourcecode/helloapp/version1/helloapp 目录下提供了该应用的所有源文件，只要把整个 helloapp 子目录拷贝到<CATALINA_HOME>/webapps 下，就可以按开放式目录结构发布这个应用。

如果 helloapp 应用开发完毕，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，也可以按如下步骤在 Tomcat 服务器上发布 helloapp 应用。

步骤

（1）在 DOS 下转到 helloapp 应用的根目录。

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

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

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

（4）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。所以在服务器启动后，会发现服务器把 helloapp.war 展开到<CATALINA_HOME> /webapps/helloapp 目录中。

（5）通过浏览器访问 http://localhost:8080/helloapp/hello.jsp。

2.8.1　服务器端装载 hello.jsp 的流程

在 Tomcat 服务器上成功发布了 helloapp 应用后，访问 http://localhost:8080/helloapp/hello.jsp 会看到如图 2-5 所示的网页。服务器端装载 hello.jsp 网页的流程如下。

（1）<bean:message>标签从 Resource Bundle 中读取文本，把它输出到网页上。

（2）<html:form>标签在 request 范围中查找 HelloForm Bean。如果存在这样的实例，就把 HelloForm 对象中的 userName 属性赋值给 HTML 表单的 userName 文本框。由于此时还不存在 HelloForm 对象，所以忽略这项操作。

（3）把 hello.jsp 的视图呈现给客户。

图 2-5　直接访问 hello.jsp 的输出网页

2.8.2　表单验证的流程

在 hello.jsp 网页上，不输入姓名，直接单击【Submit】按钮，会看到如图 2-6 所示的网页。

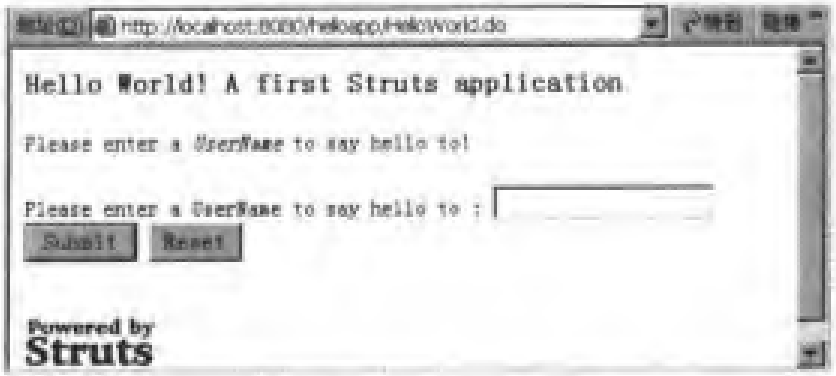

图 2-6　表单验证失败的 hello.jsp 网页

当客户提交 HelloForm 表单时，请求路径为“/HelloWorld.do”：

```
<html:form action="/HelloWorld.do" focus="userName" >
```

服务器端执行表单验证流程如下。

（1）Servlet 容器在 web.xml 文件中寻找<url-pattern>属性为“*.do”的<servlet-mapping>元素：

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

（2）Servlet 容器依据以上<servlet-mapping>元素的<servlet-name>属性“action”，在 web.xml 文件中寻找匹配的<servlet>元素：

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

（3）Servlet 容器把请求转发给以上<servlet>元素指定的 ActionServlet，ActionServlet 依据用户请求路径“/HelloWorld.do”，在 Struts 配置文件中检索 path 属性为“/HelloWorld”的<action>元素：

```
<action     path      = "/HelloWorld"
            type      = "hello.HelloAction"
            name      = "HelloForm"
            scope     = "request"
            validate  = "true"
            input     = "/hello.jsp"
>
    <forward name="SayHello" path="/hello.jsp" />
</action>
```

更确切地说，ActionServlet 此时检索的是 ActionMapping 对象，而不是直接访问 Struts 配置文件中的<action>元素。因为在 ActionServlet 初始化的时候，会加载 Struts 配置文件，把各种配置信息保存在相应的配置类的实例中，例如<action>元素的配置信息存放在 ActionMapping 对象中。

（4）ActionServlet 根据<action>元素的 name 属性，创建一个 HelloForm 对象，把客户提交的表单数据传给 HelloForm 对象，再把 HelloForm 对象保存在<action>元素的 scope 属性指定的 request 范围内。

（5）由于<action>元素的 validate 属性为 true，ActionServlet 调用 HelloForm 对象的

validate()方法执行表单验证：

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();

    if ((userName == null) || (userName.length() < 1))
        errors.add("username", new ActionMessage("hello.no.username.error"));

    return errors;
}
```

（6）HelloForm 对象的 validate()方法返回一个 ActionErrors 对象，里面包含一个 ActionMessage 对象，这个 ActionMessage 对象中封装了错误消息，消息 key 为“hello.no.username.error”，在 Resource Bundle 中与值匹配的消息文本为：

```
hello.no.username.error=Please enter a <i>UserName</i> to say hello to!
```

（7）ActionServlet 把 HelloForm 的 validate()方法返回的 ActionErrors 对象保存在 request 范围内，然后根据<action>元素的 input 属性，把客户请求转发给 hello.jsp。

（8）hello.jsp 的<html:errors>标签从 request 范围内读取 ActionErrors 对象，再从 ActionErrors 对象中读取 ActionMessage 对象，把它包含的错误消息显示在网页上。

2.8.3　逻辑验证失败的流程

接下来在 hello.jsp 的 HTML 表单中输入姓名“Monster”，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）重复 2.8.2 小节的流程（1）～（4）。

（2）ActionServlet 调用 HelloForm 对象的 validate()方法，这次 validate()方法返回的 ActionErrors 对象中不包含任何 ActionMessage 对象，表示表单验证成功。

（3）ActionServlet 查找 HelloAction 实例是否存在，如果不存在就创建一个实例。然后调用 HelloAction 的 execute()方法。

（4）HelloAction 的 execute()方法先进行逻辑验证，由于没有通过逻辑验证，就创建一个 ActionMessage 对象，这个 ActionMessage 对象封装了错误消息，消息 key 为“hello.dont.talk.to.monster”，在 Resource Bundle 中与值匹配的消息文本为：

```
hello.dont.talk.to.monster=We don't want to say hello to Monster!!!
```

execute()方法把 ActionMessage 对象保存在 ActionMessages 对象中，再把 ActionMessages 对象存放在 request 范围内。最后返回一个 ActionForward 对象，该对象包含的请求转发路径为<action>元素的 input 属性指定的 hello.jsp。

以下是 execute()方法中进行逻辑验证的代码：

```
ActionMessages errors = new ActionMessages();
String userName = (String)((HelloForm) form).getUserName();
String badUserName = "Monster";

if (userName.equalsIgnoreCase(badUserName)) {
    errors.add("username", new ActionMessage("hello.dont.talk.to.monster", badUserName ));
    saveErrors(request, errors);
    return (new ActionForward(mapping.getInput()));
}
```

（5）ActionServlet 依据 HelloAction 返回的 ActionForward 对象，再把请求转发给 hello.jsp。

（6）hello.jsp 的<html:errors>标签从 request 范围内读取 ActionMessages 对象，再从 ActionMessages 对象中读取 ActionMessage 对象，把它包含的错误消息显示在网页上，如图 2-7 所示。

图 2-7 逻辑验证失败时的 hello.jsp 网页

2.8.4 逻辑验证成功的流程

接下来，在 hello.jsp 的 HTML 表单中输入姓名“Weiqin”，然后单击【Submit】按钮。当服务器端响应客户请求时，流程如下。

（1）重复 2.8.3 小节的流程（1）～（3）。

（2）HelloAction 的 execute()方法先执行逻辑验证，这次通过了验证，然后执行相关的业务逻辑，最后调用 ActionMapping.findForward()方法，参数为“SayHello”：

```
// Forward control to the specified success URI
return (mapping.findForward("SayHello"));
```

（3）ActionMapping.findForward()方法从<action>元素中寻找 name 属性为“SayHello”的<forward>子元素，然后返回与之对应的 ActionForward 对象，它代表的请求转发路径为

"/hello.jsp"。

更确切地说，ActionMapping 从本身包含的 HashMap 中查找 name 属性为"SayHello"的 ActionForward 对象。在 ActionServlet 初始化时会加载 Struts 配置文件，把<action>元素的配置信息存放在 ActionMapping 对象中。<action>元素中可以包含多个<forward>子元素，每个<forward>子元素的配置信息存放在一个 ActionForward 对象中，这些 ActionForward 对象存放在 ActionMapping 对象的 HashMap 中。

（4）HelloAction 的 execute()方法然后把 ActionForward 对象返回给 ActionServlet，ActionServlet 再把客户请求转发给 hello.jsp。

（5）hello.jsp 的<bean:message>标签从 Resource Bundle 中读取文本，把它们输出到网页上，最后生成动态网页，如图 2-8 所示。

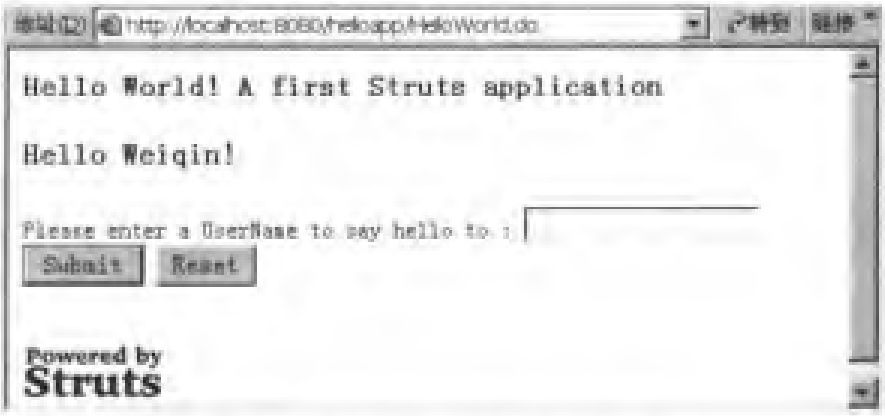

图 2-8　通过数据验证的 hello.jsp 网页

2.9　小　　结

本章通过简单但完整的 helloapp 应用的例子，演示了如何把 Struts 框架运用到 Web 应用的开发中。通过这个例子，读者可以掌握以下内容：

- 分析应用需求，把应用分解为模型、视图和控制器来实现这些需求。
- 利用 Struts 的标签库来创建视图组件，视图组件中的文本内容保存在专门的消息资源文件中，在 JSP 文件中通过 Struts 的<bean:message>标签来访问它，这样可以很方便地实现 Struts 应用的国际化，支持多国语言。
- Struts 框架采用 ActionForm Bean 把视图中的表单数据传给控制器组件，ActionForm Bean 被存放在 request 或 session 范围内，它能够被 JSP 组件、Struts 标签以及 Action 类共享。
- 数据验证分为两种类型：HTML 表单验证和业务逻辑验证。表单验证由 ActionForm Bean 的 validate()方法来实现。业务逻辑验证由 Action 类或模型组件来实现。
- ActionMessage 可以表示数据验证错误，它被保存在 ActionMessages（或其子类

ActionErrors）集合对象中。ActionMessages 对象被保存在 request 范围内，Struts 的视图组件可以通过<html:errors>标签来访问它。

- Action 类的 execute()方法调用模型组件来完成业务逻辑，它还能决定把客户请求转发给哪个视图组件。
- 模型组件具有封装业务实现细节的功能，开发者可以方便地把模型组件移植到远程应用服务器上，这不会对 MVC 的其他模块造成影响。
- 通过调用 HttpServletRequest 或 HttpSession 的 setAttribute()以及 getAttribute()方法，可以保存或访问在 request 或 session 范围内的 Java 对象，从而实现视图组件和控制器组件之间信息的交互与共享。
- 利用 struts-config.xml 文件来配置 Struts 应用。

第 3 章　Struts 应用的需求分析与设计

软件开发过程通常包括五个阶段：分析、设计、编码、测试和发布。如果在 Web 应用开发中套用现成的 Struts 框架，则可以简化每个开发阶段的工作，开发人员可以更加有针对性地去分析应用需求，不必重新设计框架，而只需在 Struts 框架的基础上，设计 MVC 各个模块包含的具体组件。在编码过程中，可以充分利用 Struts 提供的各种实用类和标签库，简化编码工作。

本章以电子通讯簿 addressbook 应用为例，讲述了对 Struts 应用进行需求分析和设计的方法，它包括以下步骤：

- 收集和分析应用需求
- 设计数据库
- 设计客户界面
- 设计 ActionForm
- 设计 Action
- 设计应用的业务逻辑组件

尽管在 Web 应用的设计阶段不涉及编码，但本章为便于讲解设计思路，提供了某些组件的代码实现。

本章样例位于配套光盘的 sourcecode/addressbook/version1 目录下，发布和运行步骤参见本书附录 C。

3.1　收集和分析应用需求

软件开发过程的起点是获得对客户业务过程的理解，分析员在和具有业务知识的客户交谈中，可以发现需求，了解系统功能，然后用 UML 用例（UseCase）来描述这些需求。

下面分析电子通讯簿 addressbook 应用的需求。当用户登入到这个应用中后，将执行添加或查询好朋友联系地址等操作。这个应用包含以下用例：

- 安全登入
- 添加好朋友联系地址
- 根据特定条件查询好朋友联系地址
- 列出所有好朋友联系地址清单
- 安全登出

接下来展开每个用例的细节和逻辑流程，以文档的形式来描述用例。用例文档中应包含以下内容：

- 前置条件：开始使用这个用例之前必须满足的条件。

- 主事件流：用例的正常流程。
- 其他事件流：用例的非正常流程，如错误流。
- 后置条件：用例的执行结果必须为真的条件，并不是每个用例都有后置条件。

下面分析 addressbook 应用中每个用例的细节。

- 用例 1：安全登入
  - 前置条件：无。
  - 主事件流：用户输入正确的用户名和密码，安全登入到应用中，向用户返回主操作菜单。
  - 其他事件流 A1：如果用户未输入用户名或密码，则显示错误提示信息：用户名和密码不允许为空。
  - 其他事件流 A2：如果用户输入非法的用户名或密码，则显示错误提示信息：用户名或密码不正确。
- 用例 2：添加好朋友联系地址
  - 前置条件：用户已经安全登入到应用中。
  - 主事件流：接收用户输入的好朋友联系地址信息（姓名、电话和地址），把信息保存到数据库中，返回提示信息：记录已经被成功保存到数据库。
  - 其他事件流 A1：如果用户未输入姓名、电话或地址，则显示错误提示信息：姓名、电话或地址不允许为空。
- 用例 3：根据特定条件查询好朋友联系地址
  - 前置条件：用户已经安全登入到应用中。
  - 主事件流：用户输入模糊查询条件（如姓名、电话和地址信息），返回符合查询条件的所有记录。
  - 其他事件流 A1：如果用户没有输入任何查询条件，则显示错误提示信息：必须至少提供一个查询条件。
- 用例 4：列出所有好朋友联系地址清单
  - 前置条件：用户已经安全登入到应用中。
  - 主事件流：返回所有的通信地址信息。
- 用例 5：安全登出
  - 前置条件：用户已经安全登入到应用中。
  - 主事件流：结束当前 HTTP 会话，重新返回到 Welcome 页面。

3.2 设计数据库

接下来分析应用中的数据流——数据从何而来，保存在什么地方。对于 addressbook 应用，用户输入好朋友联系地址信息，包括姓名、电话号码和通信地址，服务器端把这些信息保存在数据库中。因此，可以在数据库中创建一张表：ADDRESSBOOK_TABLE，来存放以上信息。表 ADDRESSBOOK_TABLE 的结构如表 3-1 所示。

表 3-1　ADDRESSBOOK_TABLE 表的结构

字　段	类　型	说　明
ID	int（4）	记录 ID，能自动增长，为主键
NAME	char（25）	朋友姓名
PHONE	char（10）	电话号码
ADDRESS	char（50）	地址

addressbook 应用的用例 1 提供了验证用户身份的功能。可以把所有合法用户信息（用户名和密码）也保存在数据库中。不过，在本例中为了演示解析 XML 数据的方法，把用户信息保存在 XML 文件 userdatabase.xml 中。假定本应用的初始合法用户为“guest”，密码为“guest”，在 userdatabase.xml 文件中包含如下初始数据：

```
<database>
    <user userName="guest" password="guest" >
    </user>
</database>
```

3.3　设计应用的业务逻辑

在 Struts 框架中，模型组件负责完成业务逻辑，模型组件可以是 JavaBean、EJB 和实用类。在 addressbook 应用中包括如下业务逻辑：

- 访问 XML 格式的用户信息。
- 访问和操纵数据库，包括添加和查询通信地址信息。

3.3.1　访问 XML 格式的用户信息

UserDatabaseServlet 负责访问 userdatabase.xml 文件，它的 load()方法加载 userdatabase.xml 的数据，把这些数据存放在 Hashtable 中。例程 3-1 是 UserDatabaseServlet 的源程序。

例程 3-1　UserDatabaseServlet.java

```
public final class UserDatabaseServlet
    extends HttpServlet {

        private Hashtable database = null;
        private int debug = 0;
        private String pathname = "/WEB-INF/userdatabase.xml";

        public void destroy() {
            getServletContext().removeAttribute(Constants.DATABASE_KEY);
        }
        public int getDebug() {
```

```
        return (this.debug);
    }
    public void addUser(UserBean user){
        database.put(user.getUserName(),user);
    }
    public void init() throws ServletException {
        String value;
        value = getServletConfig().getInitParameter("debug");
        try {
            debug = Integer.parseInt(value);
        } catch (Throwable t) {
            debug = 0;
        }
        if (debug >= 1)
            log("Initializing database servlet");
            value = getServletConfig().getInitParameter("pathname");
            if (value != null)
                pathname = value;
        try {
        /* Try and load the database.xml file. If we have loaded,
         * store the contents into our context so that we can use it
         * to compare against for user logons.
         */
            load();
            getServletContext().setAttribute(Constants.DATABASE_KEY,
                                             database);
        } catch (Exception e) {
            log("Database load exception", e);
            throw new UnavailableException
             ("Cannot load database from '" + pathname + "'"+e.getMessage());
        }
    }
    /**
     * Load the current databasc.xml file. Using the <code>Digester<code>
     * @see org.apache.commons.digester.Digester
     */
    private synchronized void load() throws Exception {
        database = new Hashtable();
        if (debug >= 1) log("Loading database from '" + pathname + "'");
        InputStream is = getServletContext().getResourceAsStream(pathname);
        if (is == null) {
            log("No such resource available - loading empty database");
            return;
        }
        BufferedInputStream bis = new BufferedInputStream(is);
        Digester digester = new Digester();
```

```
                digester.push(this);
                digester.setDebug(debug);
                digester.setValidating(false);
                digester.addObjectCreate("database/user", "addressbook.model.UserBean");
                digester.addSetProperties("database/user");
                digester.addSetNext("database/user", "addUser");
                digester.parse(bis);
                bis.close();
            }
        }
```

org.apache.commons.digester.Digester 类提供了解析 xml 数据的功能，可以参阅它的 API 文档，来深入了解它的使用方法。

UserDatabaseServlet 应该在 Web 容器加载 Web 应用时就被加载并初始化，因此在 web.xml 文件中应进行如下配置：

```
<servlet>
    <servlet-name>userdatabase</servlet-name>
    <servlet-class>addressbook.UserDatabaseServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>pathname</param-name>
      <param-value>/WEB-INF/userdatabase.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

当 UserDatabaseServlet 被初始化时，其 init()方法被调用，该方法调用 load()方法加载 userdatabase.xml 文件中的数据，把它保存在一个 Hashtable 对象中，然后 init()方法把这个 Hashtable 对象保存在 ServletContext 中，即 application 范围内。通过这种方式，Web 应用的所有组件都可以共享 application 范围内的这个 Hashtable 对象。

如果严格按照 MVC 模式来设计 Java Web 应用，Servlet 组件应该位于控制层，不负责实现具体的业务逻辑。本例作为简单的 Web 应用，为了简化起见，直接采用 UserDatabaseServlet 来实现了部分业务逻辑。

3.3.2　访问数据库

通信地址信息保存在数据库中，本例通过一个实用类和一个 JavaBean 来操纵数据库，这两个类为 DbUtil 和 AddressBookBean。

DbUtil 类负责建立和数据库的连接，它提供了 connectToDb()方法，该方法建立和数据库的连接并返回这个连接。代码如下：

```
public static java.sql.Connection connectToDb(String hostName, String databaseName)throws
Exception
{
        Connection connection=null;
        String connName = dbUrl + hostName+":3306"+"/"+databaseName;
          Class.forName(driverName).newInstance();
        connection = DriverManager.getConnection(connName);
        return connection;
}
```

AddressBookBean 代表通信地址信息，它包括 name、phone 和 address 属性，并且提供相关的 get/set 方法。此外，它还负责查询以及向 ADDRESSBOOK_TABLE 表添加新的记录。它的 insert()方法向 ADDRESSBOOK_TABLE 表中增加一条新的通信地址信息，代码如下：

```
public void insert()throws Exception{
    Connection con= DbUtil.connectToDb();
    PreparedStatement pStmt=null;
    try{
            pStmt=con.prepareStatement("INSERT INTO " + Constants.TABLENAME +
                                       " (name,phone,address)"+
                                       " values(?,?,?)");
            con.setAutoCommit(false);

            pStmt.setString(1,name);
            pStmt.setString(2,phone);
            pStmt.setString(3,address);
            int j=pStmt.executeUpdate();
            con.commit();
     }catch(Exception ex)
     {
        try{
            con.rollback();
        }catch(SQLException sqlex){
            sqlex.printStackTrace(System.out);
        }
        throw ex;
     }finally{
       try{
        pStmt.close();
        con.close();
       }catch(Exception e){e.printStackTrace();}
     }
  }
```

AddressBookBean 的 search()方法根据特定的查询条件，从 ADDRESSBOOK_TABLE 表中读取所有满足条件的数据，代码如下：

```
public static Vector search(String strSql)throws Exception{
    Vector addressbookBeans=new Vector();
    Connection con= DbUtil.connectToDb();
    PreparedStatement pStmt=null;
    ResultSet rs=null;
    try{
      pStmt=con.prepareStatement(strSql);
      rs=pStmt.executeQuery();
      while(rs.next())
        addressbookBeans.add(new AddressBookBean(
        rs.getString("NAME"),rs.getString("PHONE"),rs.getString("ADDRESS")));
      return addressbookBeans;
    }finally{
        try{
          rs.close();
          pStmt.close();
          con.close();
        }catch(Exception e){e.printStackTrace();}
    }
}
```

在本应用中，为了简化起见，对于每一个涉及访问数据库的用户请求，都会创建一个和数据库的连接，数据库访问完毕就关闭这个连接。由于每建立一个数据库连接都会耗费很多时间和资源，因此这种做法会影响 Web 应用的运行效率。在实际应用中，可以采用数据源来提高访问数据库的效率。数据源的配置和使用可以参见本书的 4.3.3 小节（<data-sources>元素）。

3.4　设计用户界面

用户界面是 Web 应用和用户交互的窗口。用户界面可以向用户输出信息，也可以接收用户的输入信息。根据用例，可以制定出用户界面，包括用户界面的功能、与用户交互的信息，以及用户界面之间的相互切换关系。

对于 addressbook 应用，粗看起来应该包含 4 个界面：登入、添加数据、查询数据和显示所有数据。不需要提供专门的系统登出界面，因为这只是一个事件，而不涉及输入或输出额外信息。本应用还应该提供一个主菜单界面，它提供访问系统其他功能的链接。当用户添加数据操作成功后，应该提供一个确认界面，向用户通报数据添加成功的信息。此外，为了使系统界面变得更友好，本应用还包括一个 Welcome 界面。表 3-2 列出了每个用户界面的功能。

表 3-2 addressbook 应用的用户界面及其功能

界 面	字 段	字段类型	说 明
Welcome 界面（index.jsp）	无	无	显示欢迎光临的信息，提供到登入界面的链接
登入界面（logon.jsp）	username，password	字符串	可编辑
添加数据界面（insert.jsp）	name，phone，address	字符串	可编辑
查询数据界面（search.jsp）	name，phone，address	字符串	可编辑
显示所有数据界面（display.jsp）	name，phone，address	字符串	只读
主菜单界面（mainMenu.jsp）	无	无	提供所有操作菜单
添加数据确认界面(confirmation.jsp)	无	无	向用户通报数据添加成功的信息

根据用例的事件流，可以确定各个界面的访问入口，以及界面之间的切换关系。UML 状态图可以直观地描述界面的入口和切换关系。

addressbook 应用的 search 界面接收到用户输入的查询条件后，把用户请求转到 display 界面，显示满足查询条件的记录。另外，如果用户未经过安全验证就直接访问 insert、search、display 或 mainMenu 界面，将先转到 logon 界面，让用户先登入系统。图 3-1 显示了 addressbook 应用界面的 UML 状态图。

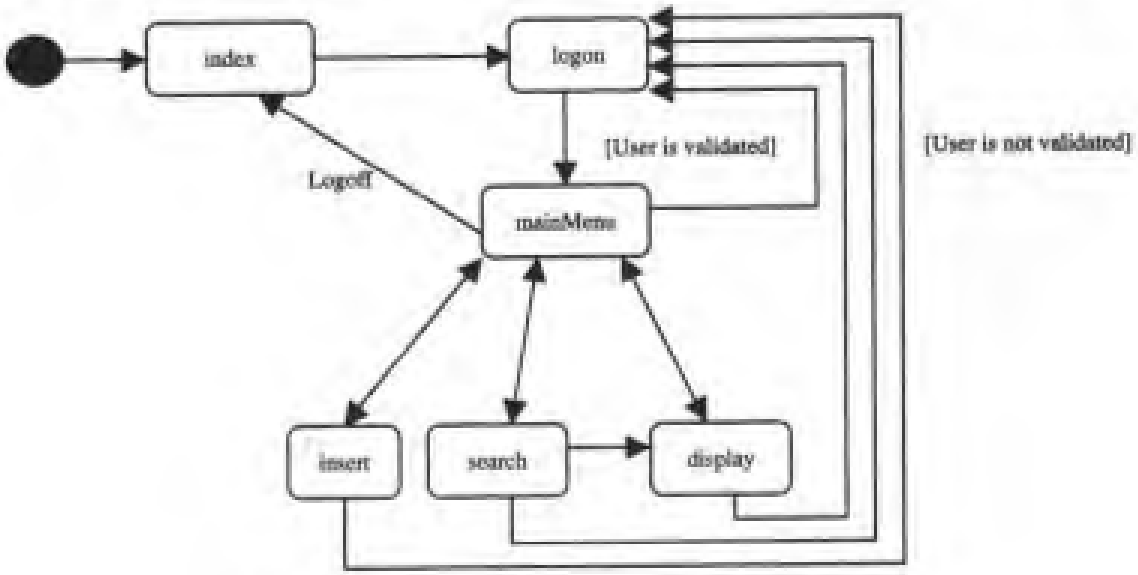

图 3-1 addressbook 应用界面的 UML 状态图

3.4.1 界面风格

所有的界面最好应保持同样的风格。在 addressbook 应用中，每个界面都包括相同的页头和页角，参见图 3-2。为了重用页头和页角的代码，分别用 header.jsp 和 footer.jsp 来定义页头和页角，其他 JSP 文件通过<include>语句把它们包含进来。

除了使用标准的 JSP <include>语句，还可以使用 Struts 的 Tiles 框架来创建复合式的 JSP 页面。参见本书第 16 章（Tiles 框架）。

图 3-2　logon.jsp 网页

例如，可以把 logon.jsp 分解为 header.jsp、logonContent.jsp 和 footer.jsp。logon.jsp 的代码如下：

```
<%@ include file="taglibs.jsp" %>
<%@ include file="header.jsp" %>
<%@ include file="logonContent.jsp" %>
<%@ include file="footer.jsp" %>
```

addressbook 应用中的 logon.jsp、mainMenu.jsp、insert.jsp、display.jsp 和 search.jsp 都分别由 hearder.jsp、xxxContent.jsp 和 footer.jsp 组合而成，参见图 3-3。

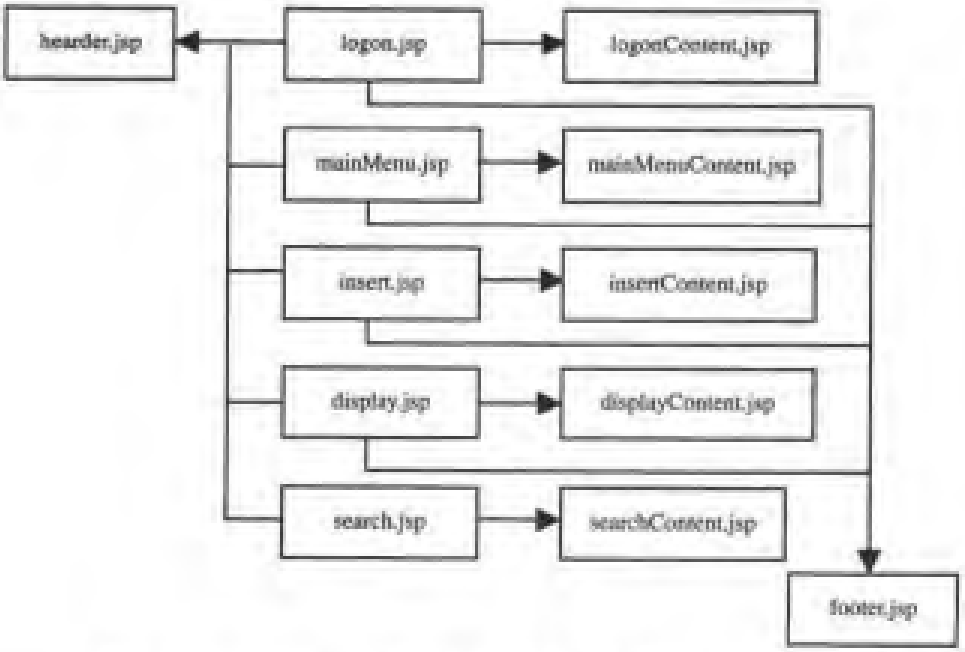

图 3-3　addressbook 应用中 JSP 文件之间的组合关系

3.4.2 使用客户化标签

在 JSP 页面中使用客户化标签，可以把 Java 程序代码和 JSP 页面分离，使 JSP 页面侧重于生成动态网页，而不涉及应用逻辑。addressbook 应用使用了 Struts HTML、Bean 和 Logic 标签库中的标签。此外，该应用需要自定义两个标签 ValidateSessionTag 和 DisplayTag，它们位于自定义的标签库 app.tld 中。ValidateSessionTag 标签用于会话验证，DisplayTag 用于输出通信地址信息，它们的实现参见本章 3.7 节。

为了提高 JSP 代码的可重要性，可以把声明客户化标签库的语句放在单独的 JSP 文件 taglibs.jsp 中，其他的 JSP 文件通过<include>语句把 taglibs.jsp 包含进来。例程 3-2 为 taglibs.jsp 的代码。

例程 3-2 taglibs.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<%-- Listing of all of the taglibs that we reference in this application --%>
<%-- Struts provided Taglibs --%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<%-- Application specific Taglibs --%>
<%@ taglib uri="/WEB-INF/app.tld" prefix="app" %>

<%-- Common error page for all JSPs --%>
<%@ page errorPage="/errorpage.jsp" %>
```

例如，在 insert.jsp 的主文件 insertContent.jsp 中，先把 taglibs.jsp 包含进来，接下来的代码中除了 JSP 标签和客户化标签，没有任何程序代码。例程 3-3 是 insertContent.jsp 的代码。

例程 3-3 insertContent.jsp

```
<%@ include file="taglibs.jsp" %>
<app:validateSession/>

<html:errors/>
<html:form action="/insert.do" focus="title">
  <center>
    <table border="0" cellspacing="2" cellpadding="2" width="100%">
      <tr>
        <td align="right"> <bean:message key="prompt.name"/></td>
        <td><html:text property="name" size="25" maxlength="25"/></td>
      </tr>
      <tr>
```

```
        <td align="right"> <bean:message key="prompt.phone"/></td>
        <td><html:text property="phone" size="25" maxlength="10"/></td>
      </tr>
      <tr>
        <td align="right"> <bean:message key="prompt.address"/></td>
        <td><html:text property="address" size="25" maxlength="50"/></td>
      </tr>
      <tr>
        <td align="right">
          <html:submit property="submit" >
          <bean:message key="button.insert"/>
          </html:submit>
        </td>
        <td align="left">
          <html:reset  >
          <bean:message key="button.reset"/>
          </html:reset>
        </td>
      </tr>
    </table>
  </center>
</html:form>
<html:link forward="mainMenu"><bean:message key="goto.mainMenu"/></html:link>
```

3.5　设计 ActionForm

ActionForm Bean 用于在视图组件和控制器组件之间传递 HTML 表单数据，通常每个 HTML 表单对应一个 ActionForm Bean，HTML 表单中的字段和 ActionForm Bean 中的属性一一对应。此外，ActionForm 的 validate()方法用于对用户输入的数据进行合法性验证。由于 ActionForm 工作于视图组件和控制器组件之间，不会访问模型组件，因此，validate()方法通常不涉及对数据的业务逻辑验证，只是完成简单的数据格式和语法检查。

在 addressbook 应用中包含三个 ActionForm Bean：LogonForm、InsertForm 和 SearchForm。表 3-3 列出了这个 ActionForm 的属性以及 validate()方法的功能。

表 3-3　addressbook 应用的 ActionForm Bean

ActionForm 名	属　性	validate()方法
LogonForm	username，password	username 和 password 都不允许为空
InsertForm	name,phone,address	name,phone,address 都不允许为空
SearchForm	name,phone,address	name,phone,address 不允许都为空

例程 3-4 为 InsertForm.java 的源代码。

例程 3-4 InsertForm.java

```
package addressbook.forms;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

/**
 * <strong>InsertForm</strong> handles the form
 * that the user will use to insert a new Address into
 * the database.
 */

public final class InsertForm extends ActionForm {

    private String name = null;
    private String phone = null;
    private String address = null;

    public String getName() {
        return name;
    }
    public String getPhone() {
        return phone;
    }
    public String getAddress() {
        return address;
    }
    public void reset(ActionMapping mapping, HttpServletRequest request) {

        name = null;
        phone = null;
        address = null;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setPhone(String phone) {
        this.phone= phone;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public ActionErrors validate(ActionMapping mapping,
```

```
                                    HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if (((name == null) || (name.length() < 1)))
            errors.add("name", new ActionMessage("error.name.required"));
        if(((phone == null) || (phone.length() < 1)))
            errors.add("phone", new ActionMessage("error.phone.required"));
        if(((address == null) || (address.length() < 1)))
            errors.add("address", new ActionMessage("error.address.required"));

        return errors;

    }
}
```

3.6　设计 Action 和 Action 映射

Action 负责单个事件的流程控制。addressbook 应用包括登入事件、登出事件、添加新记录事件、查询事件和显示所有记录事件。因此可以分别创建 LogonAction、LogoffAction、InsertAction、SearchAction 和 DisplayAllAction。

Action 映射决定了 Action 与其他 Web 组件之间的关联关系。表 3-4 列出了 addressbook 应用中每个 Action 的入口（即调用 Action 的组件）、传递给 Action 的 ActionForm，以及出口（即 Action 把请求转发到的目标组件）。

表 3-4　addressboook 应用的 Action 映射

Action	入　口	ActionForm	出　口
LogonAction	logon.jsp	LogonForm	mainMenu.jsp
LogoffAction	mainMenu.jsp	无	index.jsp
InsertAction	insert.jsp	InsertForm	confirmation.jsp
SearchAction	search.jsp	SearchForm	display.jsp
DisplayAllAction	mainMenu.jsp	无	display.jsp

JSP 文件使用<html:link>标签来链接到 Action 组件，例如，在 mainMenu.jsp 中把请求转发到 LogOffAction 的代码为：

```
<tr>
  <td>
    <li><html:link forward="logoff"><bean:message key="mainMenu.logoff"/></html:link></li>
  </td>
</tr>
```

<html:link>标签的 forward 属性和 struts-config.xml 文件中的<global-forwards>元素的<forward>子元素匹配：

```
<global-forwards>
```

```
    <forward  name="logoff"  path="/logoff.do"/>
    ......
</global-forwards>
```

这样，当用户在 mainMenu.jsp 中选择 Logoff 菜单时，Struts 框架就会把用户请求转发给 LogoffAction 组件。

此外，JSP 文件采用<html:form>标签来定义 HTML 表单，其 action 属性直接指向 Action 组件。例如，在 logon.jsp 中，当用户提交登入表单时，Struts 框架就会把用户请求转发给 LogOnAction 组件，相关的代码如下：

```
<html:form action="/logon.do" focus="userName">
```

图 3-4 为 addressbook 应用中 Action 组件的映射图。

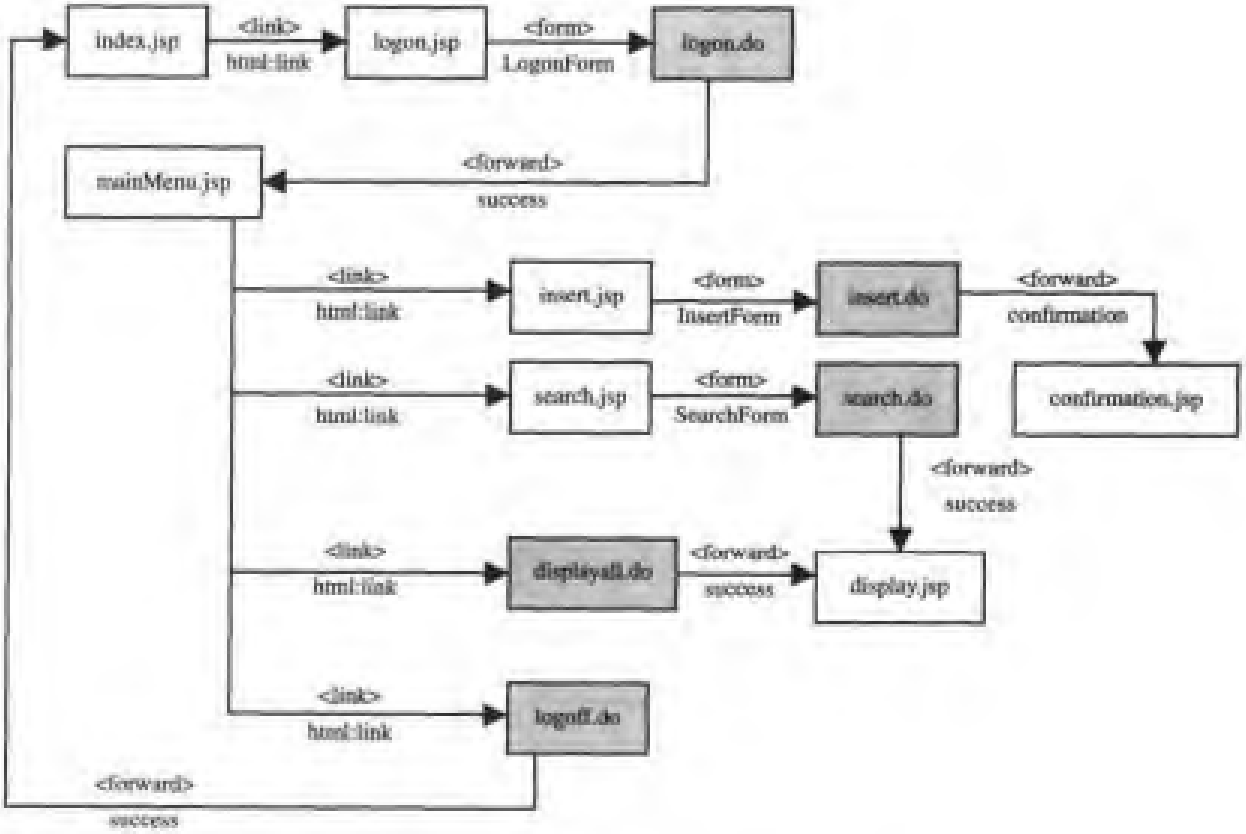

图 3-4　addressbook 应用中 Action 组件的映射图。

以上 Action 的映射关系在 Struts 配置文件中配置，例程 3-5 为 struts-config.xml 文件的源代码。

例程 3-5　struts-config.xml

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

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
```

```
<struts-config>

  <!-- ========== Form Bean Definitions ================================== -->
  <form-beans>
    <form-bean        name="logonForm"
                      type="addressbook.forms.LogonForm"/>
    <form-bean        name="searchForm"
                      type="addressbook.forms.SearchForm"/>
    <form-bean        name="insertForm"
                      type="addressbook.forms.InsertForm"/>

  </form-beans>

  <!-- ========== Global Forward Definitions ============================== -->
  <global-forwards>
    <forward    name="logoff"                path="/logoff.do"/>
    <forward    name="logon"                 path="/logon.jsp"/>
    <forward    name="success"               path="/mainMenu.jsp"/>
    <forward    name="search"                path="/search.jsp"/>
    <forward    name="displayall"            path="/displayall.do"/>
    <forward    name="insert"                path="/insert.jsp"/>
    <forward    name="mainMenu"                path="/mainMenu.jsp"/>
    <forward    name="confirmation"     path="/confirmation.jsp"/>
  </global-forwards>

<!-- ========== Action Mapping Definitions ============================== -->
    <action-mappings>
    <action       path="/search"
                  type="addressbook.actions.SearchAction"
                  name="searchForm"
                  attribute="myForm"
                  scope="request"
                  input="/search.jsp">
      <forward name="success" path="/display.jsp"/>
    </action>

    <action       path="/displayall"
                  type="addressbook.actions.DisplayAllAction"
                  name="nestedForm"
                  scope="request"
                  input="/mainMenu.jsp">
      <forward name="success" path="/display.jsp"/>
    </action>

    <action     path="/insert"
```

```
                type="addressbook.actions.InsertAction"
                name="insertForm"
                scope="request"
                input="/insert.jsp"
                validate="true">
        </action>

        <!-- Process a user logoff -->
        <action     path="/logoff"
                    type="addressbook.actions.LogoffAction">
          <forward name="success" path="/index.jsp"/>
        </action>

        <!-- Process a user logon -->
        <action     path="/logon"
                    type="addressbook.actions.LogonAction"
                    name="logonForm"
                   scope="request"
                   input="/logon.jsp">
        </action>
    </action-mappings>

    <message-resources parameter="addressbook.ApplicationResources"/>

</struts-config>
```

3.6.1 设计 LogonAction

当用户在 logon.jsp 网页上提交登入表单后，Struts 框架就会把用户请求转发给 LogonAction 组件。LogonAction 执行安全验证任务，如果验证成功，就把请求转发给 mainMenu.jsp，否则把请求转发给 logon.jsp，并且显示验证失败信息。

LogonAction 在进行安全验证时，先从当前 ServletContext 中取出存放用户信息的 Hashtable 对象，然后检查用户输入的用户名和密码是否在 Hashtable 对象中存在，其代码如下：

```
Hashtable database = (Hashtable)
  servlet.getServletContext().getAttribute(Constants.DATABASE_KEY);
if (database == null)
        errors.add(ActionMessages.GLOBAL_MESSAGE,
                new ActionMessage("error.database.missing"));
else {
    user = (UserBean) database.get(userName);
    if ((user != null) && !user.getPassword().equals(password)){
        user = null;
    }
```

```
        if (user == null)
                errors.add(ActionMessages.GLOBAL_MESSAGE,
                        new ActionMessage("error.password.mismatch"));
    }
```

如果用户输入的用户名和密码在 Hashtable 对象中存在，就表示验证成功，LogonAction 把用户信息保存在当前 HTTPSession 中，其代码如下：

```
// Save our logged-in user in the session
HttpSession session = request.getSession();
session.setAttribute(Constants.USER_KEY, user);
```

例程 3-6 是 LogonAction 的 execute()方法的完整代码。

例程 3-6　LogonAction 的 execute()方法

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)throws Exception {
    String userName=null;
    String password=null;

    UserBean user = null;
    // Demonstrate a trace call to the logger
    if (log.isTraceEnabled()) {
        log.trace("LogonAction: entering method");
    }
    // Validate the request parameters specified by the user
    ActionMessages errors = new ActionMessages();
    if (form != null){
        userName = ((LogonForm) form).getUserName();
        password = ((LogonForm) form).getPassword();
    }
    Hashtable database = (Hashtable)
      servlet.getServletContext().getAttribute(Constants.DATABASE_KEY);
    if (database == null)
            errors.add(ActionMessages.GLOBAL_MESSAGE,
                    new ActionMessage("error.database.missing"));
    else {
        user = (UserBean) database.get(userName);
        if ((user != null) && !user.getPassword().equals(password)){
            user = null;
        }
        if (user == null)
                errors.add(ActionMessages.GLOBAL_MESSAGE,
                        new ActionMessage("error.password.mismatch"));
    }
```

```
        // Report any errors we have discovered back to the original form
        if (!errors.isEmpty()) {
            saveErrors(request, errors);
            return (new ActionForward(mapping.getInput()));
        }

        // Save our logged-in user in the session
        HttpSession session = request.getSession();
        session.setAttribute(Constants.USER_KEY, user);
        // Demonstrate a debug call to the logger
        if (log.isDebugEnabled()) {
            log.debug("LogonAction: User '" + user.getUserName() +
              "' logged on in session " + session.getId());
        }
        // Remove the obsolete form bean
        if (mapping.getAttribute() != null) {
            if ("request".equals(mapping.getScope()))
                request.removeAttribute(mapping.getAttribute());
            else
                session.removeAttribute(mapping.getAttribute());
        }

        // Forward control to the specified success URI
        return (mapping.findForward(Constants.FORWARD_SUCCESS));
    }
```

3.6.2 设计 LogoffAction

当用户在 mainMunu.jsp 网页上选择 Logoff 菜单时，Struts 框架就会把用户请求转发给 LogoffAction 组件。LogoffAction 从当前 HTTPSession 中删除用户信息，然后使 HTTPSession 无效，再把请求转发给 index.jsp。例程 3-7 是 LogoffAction 的 execute()方法的代码。

例程 3-7　LogoffAction 的 execute()方法

```
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)throws Exception {

        // Extract attributes we will need
        HttpSession session = request.getSession();
        UserBean user = (UserBean) session.getAttribute(Constants.USER_KEY);

        // Process this user logoff and log something to the servlet
        // log so we can track users
```

```
        if (user != null) {
            if (log.isDebugEnabled()) {
            log.debug("LogoffAction: User [" + user.getUserName() +
                              "] logged off in session " + session.getId());
            }
        }
        // Clean up the session for safe keeping, and then get rid of it
        session.removeAttribute(Constants.USER_KEY);
        session.invalidate();

        // Forward control to the Success URI
        return (mapping.findForward(Constants.FORWARD_SUCCESS));
    }
```

3.6.3　设计 InsertAction

当用户在 insert.jsp 网页上提交了 InsertForm 表单后，Struts 框架就会把用户请求转发给 InsertAction 组件。InsertAction 调用 AddressBookBean 的 insert()方法，把记录保存到数据库中，然后 InsertAction 把请求转发给 confirmation.jsp。例程 3-8 是 InsertAction 的 execute()方法的代码。

例程 3-8　InsertAction 的 execute()方法

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {

    String name=null;
    String phone=null;
    String address=null;

    ActionMessages errors = new ActionMessages();

    name = ((InsertForm) form).getName();
    phone=((InsertForm)form).getPhone();
    address=((InsertForm)form).getAddress();

    try
    {
        AddressBookBean bean=new AddressBookBean(name,phone,address);
        bean.insert();
    }catch(Exception ex)
    {
```

```
            ex.printStackTrace(System.out);
            errors.add(ActionMessages.GLOBAL_MESSAGE,
                    new ActionMessage("error.insert.failed"));
        }

        if (!errors.isEmpty()) {
            saveErrors(request, errors);
            return (new ActionForward(mapping.getInput()));
        }
        // If we had no errors, then add a confirmation message
        ActionMessages actionMessages = new ActionMessages();
        actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
                            new ActionMessage("record.inserted"));
        saveMessages(request,actionMessages);

        return (mapping.findForward(Constants.FORWARD_CONFIRMATION));
    }
```

3.6.4 设计 SearchAction

当用户在 search.jsp 网页上提交了 SearchForm 表单后，Struts 框架就会把用户请求转发给 SearchAction 组件。SearchAction 调用根据用户输入的查询条件，生成模糊查询 SQL 语句，再把 SQL 语句保存到 HTTPSession 中，然后把请求转发给 display.jsp。例程 3-9 是 SearchAction 的 execute()方法的代码。

例程 3-9　SearchAction 的 execute()方法

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)throws Exception {

    ActionMessages errors = new ActionMessages();
    String name = ((SearchForm) form).getName();
    String phone = ((SearchForm) form).getPhone();
    String address=((SearchForm)form).getAddress();

    if (!errors.isEmpty()) {
        saveErrors(request, errors);
        return (new ActionForward(mapping.getInput()));
    }

    String strSql = new String("SELECT * FROM " + Constants.TABLENAME
                    + " WHERE");
```

```
        if (!name.equals(""))
            strSql = strSql + "name LIKE '"+ name +"%' AND";
        if (!phone.equals(""))
            strSql = strSql + " phone LIKE '"+ phone +"%' AND";
        if (!address.equals(""))
            strSql = strSql + " address LIKE '"+ address +"%'";
        else
            strSql = strSql.substring(0,strSql.length()-3);

        strSql = strSql + "ORDER by ID";
        HttpSession session = request.getSession();
        if (log.isDebugEnabled()) {
            log.debug("SearchAction session = " + session);
            log.debug("SearchAction strSql = " + strSql);
        }
        session.setAttribute(Constants.SQLSTMT_KEY, strSql);
        return (mapping.findForward(Constants.FORWARD_SUCCESS));
    }
```

3.6.5　设计 DisplayAllAction

当用户在 mainMenu.jsp 网页上选择了 display 表单时，Struts 框架就会把用户请求转发给 DisplayAllAction 组件。DisplayAllAction 生成查询 SQL 语句，该语句能够查询 ADDRESSBOOK_TABLE 中所有的记录。DisplayAllAction 再把 SQL 语句保存到 HTTPSession 中，然后把请求转发给 display.jsp。例程 3-10 是 DisplayAllAction 的 execute() 方法的代码。

例程 3-10　DisplayAllAction 的 execute()方法

```
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)throws Exception {

        String strSql = new String("SELECT * FROM " +
        Constants.TABLENAME + " ORDER BY ID");
        HttpSession session = request.getSession();
        session.setAttribute(Constants.SQLSTMT_KEY, strSql);

        return (mapping.findForward(Constants.FORWARD_SUCCESS));
    }
```

3.7 设计客户化标签

在 addressbook 应用中定义了两个客户化标签：ValidateSessionTag 和 DisplayTag。ValidateSessionTag 标签用于会话验证，DisplayTag 用于输出通信地址信息。

3.7.1 设计 ValidateSessionTag 标签

根据 addressbook 应用的需求分析，当用户访问应用中的主菜单，以及执行添加、查询和显示记录的操作时，要求用户已经通过安全验证，处于有效的 HTTP Session 中，否则把用户请求转到 logon.jsp，提示用户重新登入。

根据 3.6.1 小节的内容，在 LogonAction 中，如果用户输入合法的用户名和口令，就会把用户名保存在当前的 HttpSession 中，其代码如下：

```
// Save our logged-in user in the session
HttpSession session = request.getSession();
session.setAttribute(Constants.USER_KEY, user);
```

这样，如果用户访问其他的 JSP 网页，如 insert.jsp，只要检查当前 HttpSession 中是否保存了用户名，就可以判断用户是否处于经过安全验证的有效 Session 中。如果用户不处于有效的 Session 中，就把请求转发给 logon.jsp。这段验证逻辑是由 ValidateSessionTag 来实现的。例程 3-11 为 ValidateSessionTag 的源程序。

例程 3-11 ValidateSessionTag.java

```
package addressbook.tags;

import java.io.IOException;
import javax.servlet.http.HttpSession;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;
import addressbook.Constants;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * <strong>ValidateSessionTag</strong> is a custom tag used
 * with this application to determine is a user has a current
 * validate session.
 */
```

```
public final class ValidateSessionTag extends TagSupport {

    private String name = Constants.USER_KEY;
    private String page = Constants.LOGON_JSP;
    private Log log =LogFactory.getLog(this.getClass().getName());

    public int doEndTag() throws JspException {
     boolean valid = false;
     HttpSession session = pageContext.getSession();
     if ((session != null) && (session.getAttribute(name) != null))
          valid = true;

     if (valid)
          return (EVAL_PAGE);
     else {
          try {
           pageContext.forward(page);
          } catch (Exception e) {
           throw new JspException(e.toString());
          }
          return (SKIP_PAGE);
     }
    }
    public int doStartTag() throws JspException {
     return (SKIP_BODY);
    }
    public String getName() {
     return (this.name);
    }
    public String getPage() {
     return (this.page);
    }
    public void release() {
         super.release();
         this.name = Constants.USER_KEY;
         this.page = Constants.LOGON_JSP;
    }
    public void setName(String name) {
     this.name = name;
    }
    public void setPage(String page) {
     this.page = page;
    }
}
```

所有必须先通过安全验证才能访问的 JSP 文件，如 insert.jsp，无需在 JSP 文件中编写专门的程序，只要使用 validateSession 标签就可以完成 Session 验证任务。在 insertContent.jsp 的开头使用了 validateSession 标签：

```
<app:validateSession/>
```

3.7.2 设计 DisplayTag 标签

display.jsp 用于输出所有符合条件的通信地址信息，同样，我们可以把这段流程由 DisplayTag 来完成。根据 3.6.4 和 3.6.5 小节的内容，无论是 SearchAction 还是 DispalyAllAction 组件，都会把 SQL 查询语句保存在当前的 HTTPSession 中。DisplayTag 先从 HTTPSession 中取出 SQL 查询语句，再调用 AddressBookBean 的 search()方法，从数据库中获得符合条件的通信地址记录，然后把它输出到 JSP 网页上。例程 3-12 是 DisplayTag 类的 doEndTag()方法。

例程 3-12 DisplayTag 类的 doEndTag()方法

```
public int doEndTag() throws JspException
{
    JspWriter out = pageContext.getOut();
    HttpSession session = pageContext.getSession();
    try
    {
        String strSql=(String)session.getAttribute(Constants.SQLSTMT_KEY);
        Vector addressBookBeans=AddressBookBean.search(strSql);
        out.println("<table border=\"2\" cellspacing=\"0\" cellpadding=\"0\">");
        out.println("<tr>");
        out.println("<th BGCOLOR=\"#00FF00\"><b>Name</b></th>");
        out.println("<th BGCOLOR=\"#00FF00\"><b>Phone<b></th>");
        out.println("<th BGCOLOR=\"#00FF00\"><b>Address</b></th>");
        out.println("</tr>");

        for(int i=0;i<addressBookBeans.size();i++)
        {
            AddressBookBean bean=(AddressBookBean)addressBookBeans.elementAt(i);
            out.println("<tr>");
            out.println("<td>"+ bean.getName() + "</td>");
            out.println("<td>"   +bean.getPhone() + "</td>");
            out.println("<td>"+bean.getAddress()+"</td>");
        }
        out.println("</table>");
    }catch (Exception ex)
    {
      throw new JspTagException("IOException:" + ex.toString());
    }
```

```
        return super.doEndTag();
    }
```

这样，display.jsp 无需在 JSP 文件中编写专门的程序，只要使用 display 标签就可以显示相关的通信地址数据。在 displayContent.jsp 中使用了 display 标签：

```
<app:display />
```

3.7.3　创建客户化 app 标签库的 TLD 文件

在创建了 DisplayTag 和 ValidateSessionTag 标签处理类后，假定把这两个标签都放在自定义的 app 标签库中，接下来应该创建 app 标签库的描述文件 app.tld，参见例程 3-13。这个文件和其他 Struts 标签库的 tld 文件都位于 Web 应用的 WEB-INF 目录下。

例程 3-13　app.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">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>Application Tag Library</shortname>
  <uri>http://jakarta.apache.org/taglibs/struts-example-1.0</uri>
  <info>
    This tag library contains functionality for the Addressbook Struts
    Sample Application. With small modifications, they can be used
    as generic tags.
  </info>

  <tag>
    <name>validateSession</name>
    <tagclass>addressbook.tags.ValidateSessionTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>
      Validate that there is a currently logged on user, by checking for
      the existence of a session-scope bean under the specified name.
      If there is no such bean, forward control to the specified page,
      which will typically be a logon form.

      Attributes:
      name - Name of the session-scope bean to check for
      page - Context-relative path to the logon page
    </info>
```

```
        <attribute>
          <name>name</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
        </attribute>

        <attribute>
          <name>page</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
        </attribute>
      </tag>

      <tag>
        <name>display</name>
        <tagclass>addressbook.tags.DisplayTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>
            Displays all the records from the database that match a given criteria
        </info>
      </tag>
    </taglib>
```

3.8 小　　结

本章通过一个更为复杂的 addressbook 应用，来讲述如何分析和设计 Struts 应用。读者通过这个例子，可以进一步掌握为视图、控制器和模型组件分配功能的技巧，以及实现这些组件之间的通信和数据共享的方法。表 3-5 列出了 addressbook 应用的 MVC 结构。以下是一些设计 Struts 应用的策略：

- JSP 文件侧重于生成动态网页，应该将程序代码从 JSP 文件中分离，JSP 文件借助客户化标签来完成视图层的程序逻辑。
- 客户化标签负责视图层的程序逻辑，如数据显示（DisplayTag），或者进行会话的有效性验证（ValidateSessionTag）。客户化标签既可以访问模型组件，也可以访问存放在 Web 应用的 request、session 和 application 范围内的共享数据。例如，DisplayTag 从当前 HTTPSession 对象中取得 SQL 查询语句；DisplayTag 还访问模型组件 AddressBookBean 来进行数据库查询操作。
- ActionForm 在对表单数据进行验证时，主要是进行语法和格式的检查，不涉及业务逻辑验证。
- Action 组件侧重于业务逻辑验证和流程控制，决定用户请求的转发。Action 组件既可以访问模型组件，也可以访问存放在 Web 应用的 request、session 和 application 范围内的共享数据。例如，SearchAction 把 SQL 查询语句保存在 HTTPSession 对象中；InsertAction 访问模型组件 AddressBookBean 来进行数据库插入操作。

- 模型组件负责实际业务逻辑，如对数据库中数据的查询、添加和删除操作。模型组件应该与视图及控制器保持独立，模型组件不应该访问 Servlet API 或 Struts API 中的类，例如 HttpServletRequest、HttpServletResponse、Action 或 ActionForm 等对象。

表 3-5　addressbook 应用的 MVC 结构

视　图			控 制 器	模　型
JSP 组件	ActionForm Bean	客户化标签	ActionServlet	AddressBookBean 类
index.jsp	LogonForm Bean	Struts HTML 库的标签	UserDatabaseServlet	DBUtil 类
logon.jsp	InsertForm Bean	Struts Bean 库的标签	LogonAction	UserBean 类
mainMenu.jsp	SearchForm Bean	Struts Logic 库的标签	LogoffAction	
search.jsp		用户自定义 app 标签库的标签	InsertAction	
display.jsp			SearchAction	
confirmation.jsp			DisplayAllAction	

第 4 章　配置 Struts 应用

Struts 应用采用两个基于 XML 的配置文件来配置应用。这两个配置文件为 web.xml 和 struts-config.xml。web.xml 适用于所有的 Java Web 应用，它是 Web 应用的发布描述文件，在 Java Servlet 规范中对它做了定义。对于 Struts 应用，在 web.xml 文件中除了配置 Java Web 应用的常规信息，还应该配置和 Struts 相关的特殊信息。

struts-config.xml 文件是 Struts 应用专有的配置文件，事实上，也可以根据需要给这个配置文件起其他的文件名。

4.1　Web 应用的发布描述文件

Web 应用的发布描述文件可以在应用开发者、发布者和组装者之间传递配置信息。Web 容器在启动时从该文件中读取配置信息，根据它来装载和配置 Web 应用。

所有和 Servlet2.3 规范兼容的 Servlet 容器支持以下的发布信息：

- 初始化参数
- Session 配置
- Servlet 声明
- Servlet 映射
- 应用生命周期的监听类
- 过滤器定义和映射
- MIME 类型映射
- 欢迎文件列表
- 出错处理页面

当 Web 应用的 JSP 文件使用了客户化标签，或者 Web 容器作为 J2EE 应用服务器的一部分时，还需要配置以下两个元素：

- 标签库映射
- JNDI 引用

4.1.1　Web 应用发布描述文件的文档类型定义（DTD）

文档类型定义（DTD，Document Type Definition）对 XML 文档的格式做了定义。DTD 把 XML 文档（包括 Web 应用的发布描述文件和 Struts 配置文件）都划分为以下组件：

- 元素
- 属性
- 实体

每一种 XML 文档都有独自的 DTD 文件。2.3 版本的 Web 应用发布描述文件的 DTD 可以从 http://java.sun.com/dtd/index.html 上下载。

下面的 DTD 代码定义了 Web 应用发布描述文件的顶层元素<web-app>：

```
<!ELEMENT web-app (icon?, display-name?, description?,
    distributable?, context-param*, filter*, filter-mapping*,
    listener*, servlet*, servlet-mapping*, session-config?, mime-
    mapping*, welcome-file-list?, error-page*, taglib*, resource-
    env-ref*, resource-ref*, security-constraint*, login-config?,
    security-role*, env-entry*, ejb-ref*, ejb-local-ref*)
>
```

<web-app>元素是 web.xml 的根元素，其他元素（即以上 DTD 代码中括号以内的元素）必须嵌入在<web-app>元素以内。在上面的 DTD 代码中，还使用了一系列的特殊符号来修饰元素，表 4-1 对这些符号的作用做了说明。

子元素之间的顺序由它们在父元素中出现的先后顺序决定。例如，在<web-app>父元素中，<servlet>元素必须出现在<servlet-mapping> 元素的前面，<servlet-mapping>元素必须出现在<taglib>元素的前面。

表 4-1 DTD 中特殊符号的作用

符 号	含 义
无符号	该子元素在父元素内必须存在且只能存在一次
+	该子元素在父元素内必须存在，可以存在一次或者多次
*	该子元素在父元素内可以不存在，或者存在一次或者多次。它是比较常用的符号
?	该子元素在父元素内可以不存在，或者只存在一次。它是比较常用的符号

4.2 为 Struts 应用配置 web.xml 文件

web.xml 文件对于配置任何 Java Web 应用都是必需的。当配置 Struts 应用时，还应该在 web.xml 文件中配置和 Struts 相关的配置选项。本节讲述配置 Struts 应用的必要步骤。

4.2.1 配置 Struts 的 ActionServlet

首先同时也是最重要的一步是配置 ActionServlet。在 web.xml 中配置 ActionServlet 有两个步骤。第一步是用<servlet>元素来声明 ActionServlet。

<servlet>元素的 DTD 定义如下：

```
<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
    (servlet-class|jsp-file), init-param*, load-on-startup?, run-
    as?, security-role-ref*)
>
```

在以上<servlet>的子元素中，经常用到的有<servlet-name>、<servlet-class>和

<init-param>。其中，<servlet-name>元素用来定义 Servlet 的名称，<servlet-class>元素用来指定 Servlet 的完整类名。以下是声明 ActionServlet 的代码：

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

配置 ActionServlet 的下一步为配置<servlet-mapping>元素，它用来指定 ActionServlet 可以处理哪些 URL。以下代码对 ActionServlet 做了完整的配置：

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
 </servlet-mapping>
```

这里应该注意的是，<servlet>元素中的<servlet-name>子元素必须和<servlet-mapping>元素中的<servlet-name>子元素匹配。以上代码的<url-pattern>属性为“*.do”，表明 ActionServlet 负责处理所有以“.do”扩展名结尾的 URL。例如，如果用户请求的 URL 为 http://localhost:8080/helloapp/HelloWorld.do，Web 容器将把该请求转发给 ActionServlet。

此外，还可以按以下方式设置<url-pattern>属性：

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>/do/*</url-pattern>
 </servlet-mapping>
```

以上代码的<url-pattern>属性为“/do/*”，表明 ActionServlet 负责处理所有以“/do”为前缀的 URL。例如，如果用户请求的 URL 为 http://localhost:8080/helloapp/do/HelloWorld，Web 容器将把该请求转发给 ActionServlet。在本书后文所介绍的 netstore 应用中，就是按前缀匹配的方式来设置 ActionServlet 的<url-pattern>属性的。

不管应用中包含多少子应用，都只需要配置一个 ActionServlet。有些开发者希望设置多个 ActionServlet 类来处理应用中不同的功能，其实这是不必要的，因为 Servlet 本身支持多线程。而且，目前的 Struts 框架只允许在应用中配置一个 ActionServlet。

4.2.2　声明 ActionServlet 的初始化参数

初始化参数用来对 Servlet 的运行时环境进行初始配置。<servlet>的<init-param>子元素用于配置 Servlet 初始化参数，例如：

```
<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>
</servlet>

<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

<init-param>子元素用于声明 Servlet 初始化参数，其格式以“参数名/参数值”的形式成对出现。例如，以上代码为 ActionServlet 配置了初始化参数 config，config 参数是配置 Struts 的 ActionServlet 专有的，它用来设置 Struts 配置文件的相对路径。

在 Struts 的早期版本中，ActionServlet 还包括 content、locale 和 bufferSize 等初始化参数，Struts 1.2 将不再支持这些初始化参数，这些初始化参数被 Struts 配置文件中的<controller>元素的 content、locale 和 bufferSize 属性代替，详细的内容请参见 4.3.8 小节（<controller>元素）。

4.2.3 配置欢迎文件清单

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，Web 容器会自动调用 Web 应用的欢迎文件。<welcome-file-list>元素用来设置欢迎文件清单。以下代码声明了两个欢迎文件：welcome.jsp 和 index.jsp。

```
<welcome-file-list>
    <welcome-file>welcome.jsp </welcome-file>
    <welcome-file>index.jsp </welcome-file>
</welcome-file-list>
```

<welcome-file-list>元素中可以包含多个<welcome-file>子元素，当 Web 容器调用 Web 应用的欢迎文件时，首先寻找第一个<welcome-file>指定的文件。如果这个文件存在，将把这一个文件返回给客户；如果这个文件不存在，Web 容器将依次寻找下一个欢迎文件，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。

由于在<welcome-file-list>元素中不能配置 Servlet 映射，因此不能直接把 Struts 的 Action 作为欢迎文件。可以采用一种变通的方法来实现在欢迎文件中调用 Struts Action。首先，在 Struts 配置文件中为被调用的 Action 创建一个全局的（global）转发项，例如：

```
<global-forwards>
 <forward name="welcome" path="HelloWorld.do"/>
</global-forwards>
```

然后创建一个名叫 welcome.jsp 的 JSP 文件（也可以采用其他文件名称），当该页面被加载时，它把请求转发给以上<forward>元素指定的 Action。welcome.jsp 的代码如下：

```
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html>
 <body>
  <logic:forward name="welcome"/>
 </body>
</html>
```

最后在 web.xml 文件中把 welcome.jsp 文件配置为欢迎文件，代码如下：

```
<welcome-file-list>
 <welcome-file>welcome.jsp</welcome-file>
</welcome-file-list>
```

4.2.4　配置错误处理

尽管 Struts 框架提供了功能强大的通用错误处理机制，但不能保证处理所有的错误或异常。当错误发生时，如果 Struts 框架不能处理这种错误，就把错误抛给 Web 容器。在默认情况下，Web 容器会向用户浏览器直接返回原始的错误信息。如果想避免直接让用户看到原始的错误信息，可以在 Web 应用的发布描述文件中配置<error-page>元素。以下代码演示了如何使用<error-page>元素来避免让用户直接看到 HTTP 404 或 HTTP 500 错误。

```
<error-page>
 <error-code>404</error-code>
 <location>/common/404.jsp</location>
</error-page>

<error-page>
 <error-code>500</error-code>
 <location>/common/500.jsp</location>
</error-page>
```

如果在 web.xml 文件中做了以上配置，当 Web 容器捕获到 HTTP 404 或 HTTP 500 错误时，将根据错误代码检索<error-page>元素，如果有匹配项，就返回<location>子元素指定的文件。例如，如果发生 HTTP 404 错误，将返回 404.jsp 网页。

也可以为 Web 容器捕获的 Java 异常配置<error-page>元素，这时需要设置<exception-type>子元素，它用于指定 Java 异常类。Web 容器可能捕获如下异常：

- RuntimeException 或 Error
- ServletException 或它的子类
- IOException 或它的子类

在<exception-type>元素中声明的 Java 异常类必须是以上所列举的情况之一。以下代码演示了如何配置 ServletException 异常和 IOException：

```
<error-page>
  <exception-type>javax.servlet.ServletException</exception-type>
  <location>/common/system_error.jsp</location>
</error-page>

<error-page>
  <exception-type>java.io.IOException</exception-type>
  <location>/common/system_ioerror.jsp</location>
</error-page>
```

如果在 web.xml 文件中做了以上配置，当 Web 容器捕获到 ServletException 或 IOException 异常时，将根据异常类型检索<error-page>元素，如果有匹配项，就返回<location>子元素指定的文件。例如，如果发生 ServletException 异常，将返回 system_error.jsp 网页。

4.2.5 配置 Struts 标签库

Struts 框架提供了一些实用的客户化标签库。如果在应用中使用了这些标签库，那么必须在 Web 应用发布描述文件中配置它们，配置元素为<taglib>。以下代码演示了如何使用<taglib>元素来配置客户化标签库：

```
<taglib>
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

<taglib>元素有两个子元素：<taglib-uri>和<taglib-location>。<taglib-uri>元素指定标签库的相对或者绝对 URI 地址，Web 应用将根据这一 URI 来访问标签库；<taglib-location>元素指定标签库描述文件在文件资源系统中的物理位置。

如果 Web 应用中没有使用 Struts 标签库，就没有必要在 web.xml 文件中配置它。此外，也可以按以上方式在 web.xml 文件中配置用户自定义的客户化标签库，例如：

```
<taglib>
```

```
    <taglib-uri>/WEB-INF/mytaglibs.tld</taglib-uri>
    <taglib-location>/WEB-INF/mytaglibs.tld</taglib-location>
</taglib>
```

4.3　Struts 配置文件

Struts 框架在启动时会读入其配置文件，根据它来创建和配置各种 Struts 组件。Struts 配置文件使得开发者可以灵活地组装和配置各个组件，提高了应用软件的可扩展性和灵活性，可以避免硬编码。Struts 配置文件是基于 XML 的，相应的 DTD 文件为 struts-config_1_2.dtd。

4.3.1　org.apache.struts.config 包

在 Struts 1.1 中加入了 org.apache.struts.config 包。在 Struts 应用启动时，会把 Struts 配置文件中的配置信息读入到内存中，并把它们存放在 config 包中相关 JavaBean 类的实例中。图 4-1 为 org.apache.struts.config 包中主要类的类框图。

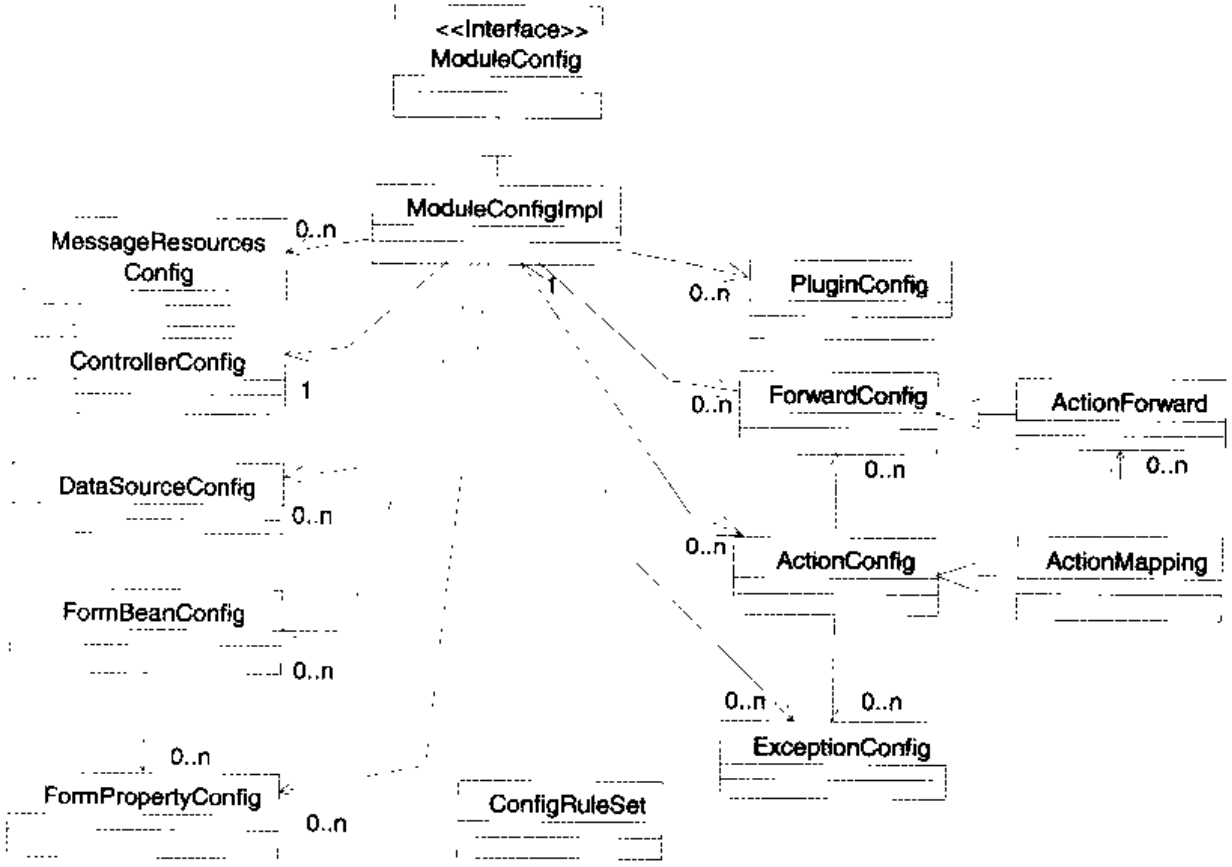

图 4-1　包 org.apache.struts.config 的类框图

org.apache.struts.config 包中的每一个类都和 Struts 配置文件中特定的配置元素对应，例如，<action>元素和 ActionMapping 类对应，<forward>元素和 ActionForward 类对应。由

于一个<action>元素可以包含多个<forward>子元素，因此，ActionMapping 类和 ActionForward 类之间存在一对多的关联关系。

当 Struts 框架完成了对配置文件的验证和解析后，就把配置文件中的信息存放在这些类的实例中。这些类的实例可以充当配置信息的运行时容器，Struts 组件可以方便地通过它们来获取配置信息。

org.apache.struts.config.ModuleConfig 在 Struts 框架中扮演了十分重要的角色。如图 4-1 所示，它是整个 org.apache.struts.config 包的核心，在 Struts 应用运行时用来存放整个 Struts 应用的配置信息。如果有多个子应用，每个子应用都会有一个 ModuleConfig 对象。ModuleConfig 和 Struts 配置文件的根元素<struts-config>对应。<struts-config>根元素中包含<form-bean>、<action>和<forward>等一系列子元素，因此 ModuleConfig 中包含了和每个子元素对应的配置类实例。在 ModuleConfig 的实现类 ModuleConfigImpl 中定义了如下成员变量：

```
/**
 * The set of action configurations for this module, if any,
 * keyed by the <code>path</code> property.
 */
protected HashMap actionConfigs = null;
/**
 * The set of JDBC data source configurations for this
 * module, if any, keyed by the <code>key</code> property.
 */
protected HashMap dataSources = null;
/**
 * The set of exception handling configurations for this
 * module, if any, keyed by the <code>type</code> property.
 */
protected HashMap exceptions = null;
/**
 * The set of form bean configurations for this module, if any,
 * keyed by the <code>name</code> property.
 */
protected HashMap formBeans = null;
/**
 * The set of global forward configurations for this module, if any,
 * keyed by the <code>name</code> property.
 */
protected HashMap forwards = null;
/**
 * The set of message resources configurations for this
 * module, if any, keyed by the <code>key</code> property.
 */
protected HashMap messageResources = null;
/**
 * The set of configured plug-in Actions for this module,
```

```
     * if any, in the order they were declared and configured.
     */
    protected ArrayList plugIns = null;

    /**
     * The controller configuration object for this module.
     */
    protected ControllerConfig controllerConfig = null;
```

图 4-1 中的 org.apache.struts.config.ConfigRuleSet 类的功能不同于其他类，它包含了解析 Struts 配置文件所需要的一组规则。在应用启动时，该类负责构造 org.apache.struts.config 包中其他用于保存配置信息的 JavaBean 类的实例。

下面分别介绍 Struts 配置文件中每个元素的用法。

4.3.2　<struts-config>元素

<struts-config>元素是 Struts 配置文件的根元素，和它对应的配置类为 org.apache.struts.config.ModuleConfig 类。<struts-config>元素有 8 个子元素，它的 DTD 定义如下：

```
<!ELEMENT struts-config (data-sources?, form-beans?, global-exceptions?, global-forwards?,
action-mappings?, controller?, message-resources*, plug-in*)
>
```

在 Struts 配置文件中，必须按照以上 DTD 指定的先后顺序来配置<struts-config>元素的各个子元素，如果颠倒了这些子元素在配置文件中的顺序，在 Struts 应用启动时会生成如图 4-2 所示的 XML 解析错误。

图 4-2　解析 Struts 配置文件时生成的错误

4.3.3 <data-sources>元素

<data-sources>元素用来配置应用所需要的数据源。数据源负责建立和特定数据库的连接，许多数据源采用连接池机制实现，以便提高数据库访问性能。Java 语言提供了 javax.sql.DataSource 接口，所有的数据源必须实现该接口。许多应用服务器和 Web 容器提供了内在的数据源组件，很多数据库厂商也提供了数据源的实现。图 4-3 表现了 Web 应用通过数据源访问数据库的过程。

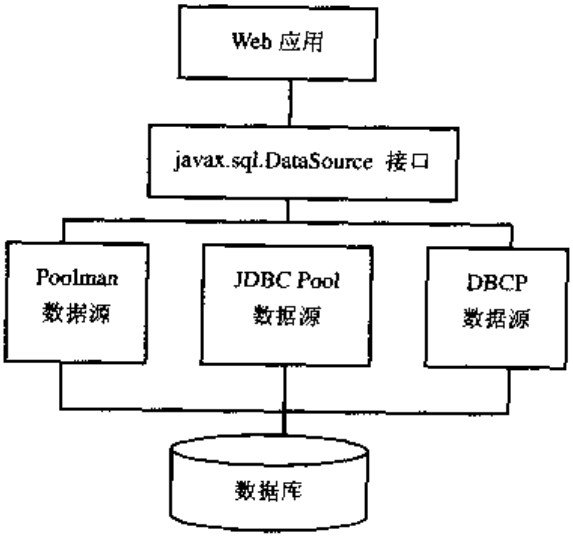

图 4-3 Web 应用通过数据源访问数据库

<data-sources>元素包含零个、一个或多个<data-source>子元素。<data-source>元素用于配置特定的数据源，它可以包含多个<set-property>子元素。<set-property>元素用于设置数据源的各种属性。下面的代码演示了如何在 Struts 配置文件中配置数据源：

```
<data-sources>
 <data-source type="org.apache.commons.dbcp.BasicDataSource" >
  <set-property property="autoCommit" value="true"/>
  <set-property property="description" value="MySQL Data Source"/>
  <set-property property="driverClass" value="com.mysql.jdbc.Driver"/>
  <set-property property="maxCount" value="10"/>
  <set-property property="minCount" value="2"/>
  <set-property property="user" value="root"/>
  <set-property property="password" value=""/>
  <set-property property="url" value="jdbc:mysql://localhost:3306/addressbooksample"/>
 </data-source>
</data-sources>
```

以上代码使用<data-source>元素配置了和 MySQL 数据库的连接。<data-source>元素的 type 属性用来指定数据源的实现类。以上代码使用的是 Apache 软件组织提供的 DBCP 数据

源。开发者应该根据实际应用的需要来选用合适的数据源实现。表 4-2 列出了几种比较流行的数据源实现。

表 4-2　可选的数据源实现

名　称	供 应 商	URL
Poolman	开放源代码软件	http://sourceforge.net/projects/poolman/
Expresso	Jcorporate	http://www.jcorporate.com
JDBC Pool	开放源代码软件	http://www.bitmechanic.com/projects/jdbcpool/
DBCP	Jakarta	http://jakarta.apache.org/commons/index.html

配置了数据源后，就可以在 Action 类中访问数据源。在 org.apache.struts.action.Action 类中定义了 getDataSource(HttpRequest)方法，它用于获取数据源对象的引用。以下程序代码演示了如何在 Action 类中访问数据源：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response) throws Exception
{
 javax.sql.DataSource dataSource;
 java.sql.Connection myConnection;
 try {
  dataSource = getDataSource(request);
  myConnection = dataSource.getConnection();
  // do what you wish with myConnection
 } catch (SQLException sqle) {
    getServlet().log("Connection.process", sqle);
 } finally {
    //enclose this in a finally block to make
    //sure the connection is closed
    try {
       myConnection.close();
    } catch (SQLException e) {
       getServlet().log("Connection.close", e);
    }
  }
}
```

也可以在配置文件中声明多个数据源，此时需要为每一个数据源分配惟一的 key 值，通过该值来标识特定的数据源。例如：

```
<data-sources>
   <data-source key="A" type="org.apache.commons.dbcp.BasicDataSource">
      ... properties as before ...
   </data-source>
   <data-source key="B" type="org.apache.commons.dbcp.BasicDataSource">
```

```
        ... properties as before ...
      </data-source>
      ...
    </data-sources>
```

在 Action 类中通过以下方式访问特定的数据源：

```
dataSourceA = getDataSource(request, "A");
dataSourceB = getDataSource(request, "B");
```

4.3.4 <form-beans>元素

<form-beans>元素用来配置多个 ActionForm Bean。<form-beans>元素包含零个或多个<form-bean>子元素。每个<form-bean>元素又包含多个属性。表 4-3 对<form-bean>元素的主要属性做了说明。

表 4-3 <form-bean>元素的属性

属 性	描 述
className	指定和<form-bean>元素对应的配置类，默认值为 org.apache.struts.config.FormBeanConfig。如果在这里设置自定义的类，该类必须扩展 FormBeanConfig 类
name	指定该 ActionForm Bean 的惟一标识符，整个 Struts 框架用该标识符来引用这个 bean。该属性是必需的
type	指定 ActionForm 类的完整类名，该属性是必需的

在配置<form-bean>元素的 type 属性时，必须给出 ActionForm 类的完整类名，即应该把类的包名也包含在内。

以下是<form-beans>元素的配置代码示例：

```
<form-beans>
  <form-bean   name="logonForm"
               type="addressbook.forms.LogonForm"/>
</form-beans>
```

如果配置动态 ActionForm Bean，还必须配置<form-bean>元素的<form-property>子元素。在本书 7.4 节（使用动态 ActionForm）详细讲述了动态 ActionForm Bean 的用法。<form-property>元素用来指定表单字段，它有四个属性，如表 4-4 所示。

表 4-4 <form-property>元素的属性

属 性	描 述
className	指定和<form-property>元素对应的配置类，默认值为 org.apache.struts.config.FormPropertyConfig
initial	以字符串的形式设置表单字段的初始值，如果没有设置该属性，则基本类型的表单字段的默认值为 0，对象类型的表单字段的默认值为 null
name	指定表单字段的名字，该属性是必需的
type	指定表单字段的类型，如果表单字段为 Java 类，必须给出完整的类名，该属性是必需的

下面的代码演示了如何使用<form-property>元素来配置动态 ActionForm Bean：

```
<form-bean    name="userForm"
        type="org.apache.struts.action.DynaActionForm">
        <form-property name="firstName" type="java.lang.String"/>
        <form-property name="lastName" type="java.lang.String"/>
        <form-property name="age" type="java.lang.Integer" initial="18"/>
</form-bean>
```

在 Struts 的早期版本中，<form-bean>元素还有一个 dynamic 属性，如果该属性为 true，则表示动态 ActionForm。Strut 1.2 将废弃该属性，因为 Struts 框架能够根据 type 属性来自动判别是否为动态 ActionForm。

4.3.5　<global-exceptions>元素

<global-exceptions>元素用于配置异常处理。<global-exceptions>元素可以包含零个或者多个<exception>元素。

<exception>元素用来设置 Java 异常和异常处理类 org.apache.struts.action.ExceptionHandler 之间的映射。表 4-5 对<exception>元素的属性做了说明。

表 4-5　<exception>元素的属性

属　性	描　述
className	指定和<exception>元素对应的配置类，默认值为 org.apache.struts.config.ExceptionConfig
handler	指定异常处理类，默认值为 org.apache.struts.action.ExceptionHandler
key	指定在 Resource Bundle 中描述该异常的消息 key
path	指定当异常发生时的转发路径
scope	指定 ActionMessages 实例的存放范围，可选值包括 request 和 session，此项的默认值为 request
type	指定所需处理的异常类的名字，此项是必需的
bundle	指定 Resource Bundle

以下是配置<global-exception>元素的示例：

```
<global-exceptions>
    <exception
      key="global.error.invalidlogin"
      path="/security/signin.jsp"
      scope="request"
      type="netstore.framework.exceptions.InvalidLoginException"/>
</global-exceptions>
```

关于 Struts 框架采用配置方式来处理异常的步骤，请参见本书的 11.4.3 小节（以配置方式处理异常）。

4.3.6 <global-forwards>元素

<global-forwards>元素用来声明全局的转发关系。<global-forwards>元素由零个或者多个<forward>元素组成。<forward> 元素用于把一个逻辑名映射到特定的 URL。通过这种方式，Action 类或者 JSP 文件无需指定实际的 URL，只要指定逻辑名就能实现请求转发或者重定向，这可以减弱控制组件和视图组件之间的耦合，并且有助于维护 JSP 文件。图 4-4 演示了如何通过<forward> 元素来实现 Web 组件之间的相互转发。

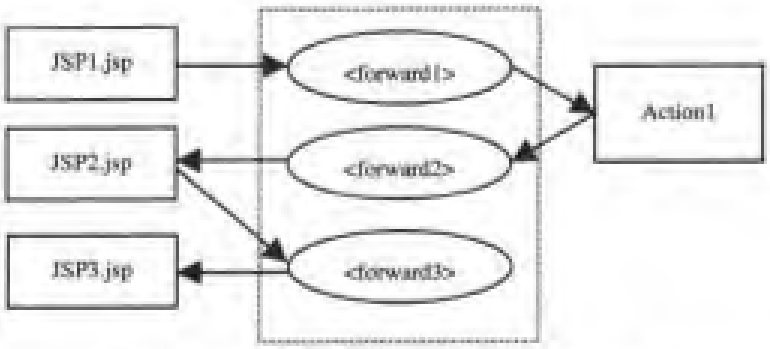

图 4-4 通过<forward>元素来实现 Web 组件之间的相互转发

表 4-6 对<forward>元素的属性做了描述。

表 4-6 <forward>元素的属性

属 性	描 述
className	和<forward>元素对应的配置类，默认值为 org.apache.struts.action.ActionForward
contextRelative	如果此项为 true，表示当 path 属性以 "/" 开头时，给出的是相对于当前上下文的 URL。此项的默认值为 false
name	转发路径的逻辑名，此项是必需的
path	指定转发或重定向的 URI，此项是必需的，必须以"/" 开头。当 contextRelative 属性为 false 时，表示 URI 路径相对于当前应用(application-relative)；当 contextRelative 属性为 true 时，表示 URI 路径相对于当前上下文(context-relative)
redirect	当此项为 true 时，表示执行重定向操作；当此项为 false 时，表示执行请求转发操作。重定向与请求转发的区别参见本书 1.2 节（Web 组件的三种关联关系），此项默认值为 false

对于图 4-4，可以配置如下<global-forwards>元素：

```
<global-forwards>
    <forward    name="forward1"     path="/Action1.do"/>
    <forward    name="forward2"     path="/JSP2.jsp"/>
    <forward    name="forward3"     path="/JSP3.jsp"/>
</global-forwards>
```

如果 JSP1.jsp 把请求转发给 Action1，可以使用以下代码：

```
<html:link forward="forward1">
```

或者：

```
<logic:forward name="forward1"/>
```

如果 Action1 的 execute()方法把请求转发给 JSP2.jsp，可以使用以下代码：

```
return (mapping.findForward("forward2"));
```

4.3.7　<action-mappings>元素

<action-mappings>元素包含零个或者多个<action>元素。<action>元素描述了从特定的请求路径到相应的 Action 类的映射。

在<action>元素中可以包含多个<exception>和<forward>子元素，它们分别配置局部的异常处理及请求转发仅被当前的 Action 所访问。

在<global-exceptions>元素中定义的<exception>子元素代表全局的异常配置。在<global-forwards>元素中定义的<forward>子元素代表全局的请求转发。在不同位置配置<exception>和<forward>元素的语法和属性是一样的。

表 4-7 对<action>元素的属性做了描述。

表 4-7　<action>元素的属性

属　性	描　述
attribute	设置和 Action 关联的 ActionForm Bean 在 request 或 session 范围内的属性 key。例如，假定 Form Bean 存在于 request 范围内，并且此项设为 "myBean"，那么 request.getAttribute("myBean")就可以返回该 Bean 的实例。此项为可选项
className	和<action>元素对应的配置元素，默认值为 org.apache.struts.action.ActionMapping
forward	指定转发的 URL 路径
include	指定包含的 URL 路径
input	指定包含输入表单的 URL 路径。当表单验证失败时，将把请求转发到该 URL
name	指定和该 Action 关联的 ActionForm Bean 的名字，该名字必须在<form-bean>元素中定义过，此项是可选项
path	指定访问 Action 的路径，它以"/"开头，没有扩展名
parameter	指定 Action 的配置参数，在 Action 类的 execute()方法中，可以调用 ActionMapping 对象的 getParameter()方法来读取该配置参数
roles	指定允许调用该 Action 的安全角色，多个角色之间以逗号隔开。在处理请求时，RequestProcessor 会根据该配置项来决定用户是否有调用 Action 的权限
scope	指定 ActionForm Bean 的存在范围，可选值为 request 和 session，默认值为 session
type	指定 Action 类的完整类名
unknown	如果此项为 true，表示可以处理用户发出的所有无效的 Action URL，默认值为 false
validate	指定是否要先调用 ActionForm Bean 的 validate( )方法，默认值为 true

<action>元素的 forward、include 和 type 属性相互排斥，也就是说只能设置其中的一项。forward 属性的作用和 org.apache.struts.actions.ForwardAction 类相同，参见本书的 5.2.1 小节。include 属性的作用和 org.apache.struts.actions.IncludeAction 类相同，参见本书的 5.2.2 小节。

以下是<action>元素的配置代码示例：

```
<action      path="/search"
                type="addressbook.actions.SearchAction"
                name="searchForm"
                scope="request"
                validate="true"
                input="/search.jsp">
        <forward name="success" path="/display.jsp"/>
</action>
```

这段配置代码表明，如果用户请求的 URI 为“search.do”，Struts 框架将把请求转发给 SearchAction；与 SearchAction 关联的表单为“searchForm”，它位于 request 范围内。Struts 框架会在 Struts 配置文件中检索匹配的<form-bean>元素：

```
<form-bean        name="searchForm"
                  type="addressbook.forms.SearchForm"/>
```

Struts 框架在把请求转发给 SearchAction 之前，先调用 SearchForm 的 validate()方法，如果表单验证失败，将把请求转发给<action>元素的 input 属性指定的 search.jsp 文件。

在以上<action>元素中还有一个<forward>子元素，如果在 SearchAction 的 execute()方法中执行以下代码，表明把请求转发给 display.jsp：

```
return (mapping.findForward("success"));
```

1．局部的和全局的<forward>元素

在<action>元素中定义的<forward>元素表示局部的请求转发项，在<globals-forward>元素中定义的<forward>元素表示全局的请求转发项。在两个不同位置定义的<forward>元素的语法是一样的。例如，以下代码分别定义了一个全局的<forward>和一个局部的<forward>，它们的 name 属性都是“success”：

```
<global-forwards>
    <forward name="success" path="/success.jsp"/>
</global-forwards>

<action    path="/Action1"   ……>
    <forward name="success" path="/login.jsp"/>
</action>

<action    path="/Action2" ……>
</action>
```

假定在 Action1 和 Action2 的 execute()方法中都执行以下代码：

```
return (mapping.findForward("success"));
```

Action1 将把请求转发给局部<forward>元素指定的 login.jsp，而 Action2 将把请求转发给全局的<forward>元素指定的 success.jsp。由此可见，如果在<action>元素中定义了局部的

<forward>元素，它的优先级别高于全局的<forward>元素。

2．<action>的 forward 属性

<Action>的 forward 属性和<forward>子元素是两个不同的概念。forward 属性指定和 path 属性匹配的请求转发路径，例如：

```
<action    path="/hello"
           forward="/hello.jsp">
</action>
```

对于以上代码，当用户请求的 URI 为“/hello.do”，Struts 框架将把请求转发给 hello.jsp 文件。

4.3.8　<controller>元素

<controller>元素用于配置 ActionServlet。表 4-8 对<controller>元素的属性做了描述。

表 4-8　<set-property>元素的属性

属　性	描　述
bufferSize	指定上载文件的输入缓冲的大小，该属性为可选项，默认值为 4096
className	指定和<controller>元素对应的配置类，默认值为 org.apache.struts.config.ControllerConfig
contentType	指定响应结果的内容类型和字符编码，该属性为可选项，默认值为 text/html，如果在 Action 和 JSP 网页中也设置了内容类型和字符编码，将会覆盖该设置
locale	指定是否把 Locale 对象保存到当前用户的 Session 中，默认值为 false
processorClass	指定负责处理请求的 Java 类的完整类名，默认值为 org.apache.struts.action.RequestProcessor，如果把此项设置为自定义的类，那么应该保证该类扩展了 org.apache.struts.action.RequestProcessor 类
tempDir	指定处理文件上载的临时工作目录，如果此项没有设置，将采用 Servlet 容器为 Web 应用分配的临时工作目录
nocache	如果为 true，在响应结果中将加入特定的头参数：Pragma，Cache-Control 和 Expires，防止页面被存储在客户浏览器的缓存中，默认值为 false

如果应用包含多个子应用，可以在每个子应用的 Struts 配置文件中配置<controller>元素。这样，尽管这些子应用共享同一个 ActionServlet 对象，但是它们可以使用不同的 RequestProcessor 类。

以下是<controller>元素的配置代码示例：

```
<controller
  contentType="text/html;charset=UTF-8"
  locale="true"
  processorClass="CustomRequestProcessor"/>
```

4.3.9　<message-resources>元素

<message-resources>元素用来配置 Resource Bundle，Resource Bundle 用于存放本地化

消息文本。关于 Resource Bundle 的详细用法参见本书的第 9 章（Struts 应用的国际化）。表 4-9 对<message-resources>元素的属性做了描述。

表 4-9 <message-resources>元素的属性

属 性	描 述
className	和<message-resources>元素对应的配置类，默认值为 org.apache.struts.config.MessageResourcesConfig
factory	指定消息资源的工厂类，默认值为 org.apache.struts.util.PropertyMessageResourcesFactory 类
key	指定 Resource Bundle 存放在 ServletContext 对象中时采用的属性 key，默认值为由 Globals.MESSAGES_KEY 定义的字符串常量。只允许有一个 Resource Bundle 采用默认的属性 key
null	指定 MessageResources 类如何处理未知的消息 key。如果此项为 true，将返回空字符串。如果此项为 false，将返回类似 "???global.label.missing???" 的字符串。该属性为可选项，默认值为 true
parameter	指定 Resource Bundle 的消息资源文件名。例如，如果此项设为 "pack1.pack2. ApplicationResources"，那么对应的实际资源文件的存放路径为：WEB-INF/classes/pack1/pack2/ApplicationResources.properties

以下是<message-resources>元素的配置代码示例，第一个元素采用默认的 key 属性，表示默认的 Resource Bundle。同一个 Struts 配置文件只允许有一个默认的 Resource Bundle，所以第二个元素必须显式地设置 key 属性：

```
<message-resources
  null="false"
  parameter="defaultResources"/>
<message-resources
  key="images"
  null="false"
  parameter="ImageResources"/>
```

许多 Struts 客户化标签都通过 bundle 属性来指定 Resource Bundle，标签的 bundle 属性和<message-resources>元素的 key 属性匹配，例如：

```
<bean:message  key="msgKey"/>
<bean:message  key="msgKey" bundle="images" />
```

在以上代码中，第一个<bean:message>标签没有指定 bundle 属性，表示从默认的 Resource Bundle 中读取消息文本，第二个<bean:message>标签的 bundle 属性为 "images"，它和 key 属性为 "images" 的<message-resources>元素匹配。

4.3.10 <plug-in>元素

<plug-in>元素用于配置 Struts 插件。关于 Struts 插件的运行机制和使用方法参见本书的 8.1 节（Struts 插件）。表 4-10 对<plug-in>元素的属性做了描述。

表 4-10 <plug-in>元素的属性

属 性	描 述
className	指定 Struts 插件类，插件类必须实现 org.apache.struts.action.PlugIn 接口

<plug-in>元素可以包含零个或多个<set-property>子元素。以下是<plug-in>元素的配置代码示例：

```
<plug-in
  className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

以上代码配置了 ValidatorPlugIn 插件，它用于初始化 Validator 验证框架。本书的第 10 章（Validator 验证框架）详细讲述了 Validator 验证框架的用法。

4.3.11　配置多应用模块

Struts 1.1 支持多应用模块，即同一个应用包含多个子应用，每个子应用可以处理相关的一组功能。例如，对于网上购物应用，可以由一个子应用来处理商品和商品目录信息，再由另一个子应用处理购物车和订单信息。把应用划分成多个模块，可以简化应用的并行开发过程，缩短开发周期。

所有的子应用都共享同一个 ActionServlet 实例，但每个子应用都有单独的配置文件。把应用划分成多个子应用模块包括以下步骤：

（1）为每个子应用创建单独的 Struts 配置文件。

（2）在 web.xml 的 ActionServlet 的配置代码中添加每个子应用信息。

（3）采用<forward>元素或 SwitchAction 类来实现子应用之间的切换。

例如，把某个应用划分为两个子应用模块，配置文件分别为：struts-config.xml 和 struts-moduleB.xml，参见图 4-5。

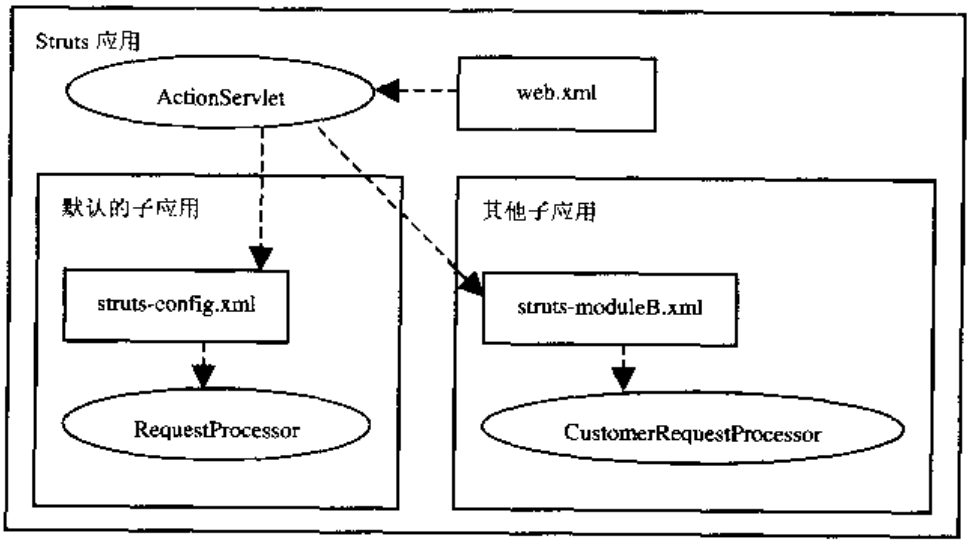

图 4-5　包含两个子应用的 Struts 应用

由图 4-5 可知，struts-config.xml 文件中的<controller>元素的 processorClass 属性为默认

值“org.apache.struts.action.RequestProcessor”，struts-moduleB.xml 文件中的<controller>元素的 processorClass 属性为“CustomerRequestProcessor”。

接下来，应该在 web.xml 的 ActionServlet 的配置代码中添加每个子应用信息，其代码如下：

```
<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>config/moduleB</param-name>
     <param-value>/WEB-INF/struts-moduleB.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

第一个<init-param>元素的 param-name 属性为 config，表示配置的是默认子应用模块。第二个<init-param>元素的 param-name 属性以“config / ”开头，表示非默认子应用模块。

有两种办法进行子应用模块之间的切换。一种办法是使用全局或局部的<forward>元素，此时将<forward>元素的 contextRelative 属性设为 true；还有一种办法是使用内置的 org.apache.struts.actions.SwitchAction 类。

以下是使用全局<forward>元素的例子：

```
<global-forwards>
   <forward name="toModuleB"
      contextRelative="true"
      path="/moduleB/index.do"
      redirect="true"/>
   ...
</global-forwards>
```

也可以使用<action>元素中的局部<forward>元素，例如：

```
<action-mappings>
   ...
    <action ... >
      <forward name="success"
        contextRelative="true"
        path="/moduleB/index.do"
        redirect="true"/>
    </action>
   ...
</action-mappings>
```

此外，还可以使用 org.apache.struts.actions.SwitchAction 类，例如：

```
<action-mappings>
    <action path="/toModule"
    type="org.apache.struts.actions.SwitchAction"/>
    ...
</action-mappings>
```

如果要从默认模块切换到 ModuleB，可以采用与以下类似的 URL：

http://localhost:8080/toModule.do**?prefix=/moduleB&page=/index.do**

如果要从 ModuleB 切换到默认模块，可以采用与以下类似的 URL：

http://localhost:8080/toModule.do**?prefix=&page=/index.do**

4.4　Digester 组件

Digester 组件是 Apache 的另一个开放源代码项目。它由一系列类组成，负责读取并解析 XML 文件，然后创建并初始化存放 XML 文件信息的 Java 对象。

当 Struts 应用被初始化时，首先会读取并解析配置文件。Struts 框架采用 Digester 组件来解析配置文件，然后创建一系列 org.apache.struts.config 包中的对象，这些对象用于存放配置信息。org.apache.struts.config.ConfigRuleSet 类决定了 Digester 组件解析配置文件并创建配置对象的规则。一般说来，没有必要修改 ConfigRuleSet 类，除非您想扩展配置文件。如果要深入学习 Digester 组件，可以访问 http://jakarta.apache.org/commons/digester/。

4.5　Struts 控制面板工具

如果 Struts 应用的规模比较小，Struts 的配置文件比较简单，则配置文件维护起来也比较简单——可以采用 XML 编辑工具或者普通的文本编辑工具来编辑配置文件。但是对于大规模的 Struts 应用来说，其配置文件往往也相当复杂，可以采用专门的 Struts 控制面板工具（Struts Console）来维护配置文件。Struts 控制面板工具由 James Holmes 创建，它是基于 Swing 的程序，可以方便地编辑 Struts 配置文件中的各个元素。该软件的下载地址为：http://www.jamesholmes.com/struts/console/。从该网址下载了 Struts Console 软件的压缩文件后，把它解压缩到本地，运行 bin 目录下的 console.bat，就可以启动 Struts Console。如图 4-6 所示为在 Struts Console 中编辑<action>元素的界面。

可以把 Struts 控制面板工具作为插件，把它集成到多种流行的 Java IDE（integrated development environment）中。Struts 控制面板工具目前支持的 IDE 包括：

- Borland JBuilder 4.0 或以上版本
- Eclipse 1.0 或以上版本
- IBM WebSphere Appl. Dev. 4.0.3 或以上版本
- IntelliJ IDEA 3.0 (build 668) 或以上版本

- NetBeans 3.2 或以上版本
- Oracle JDeveloper 9*i* 或以上版本
- Sun One Studio (Forte) 3.0 或以上版本

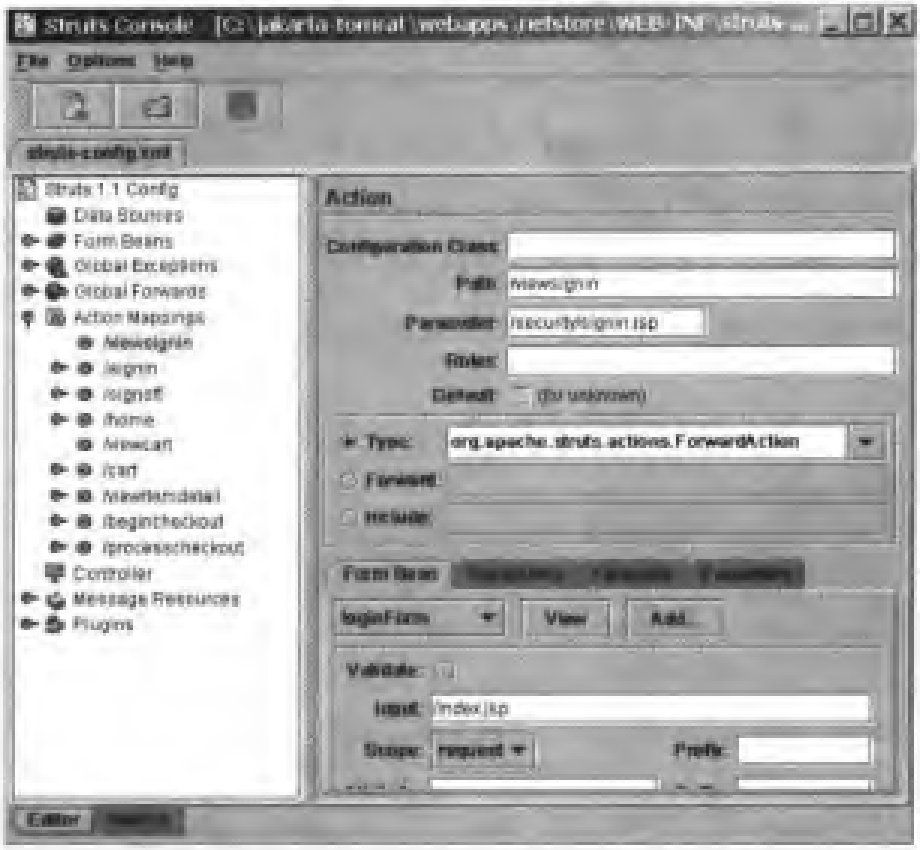

图 4-6 在 Struts Console 中编辑<action>元素的界面

假定 Struts Console 的根目录为<struts-console>，JBuilder 的根目录为<JBuilder>，把 Struts Console 作为插件加入到 Jbuilder 中的步骤如下。

步骤

（1）如果 JBuilder 处于运行状态，则先关闭 JBuilder。

（2）如果在<JBuilder>\lib 目录下不存在 xerces.jar 文件，则把<struts-console>\com.jamesholmes.console.struts\lib 目录下的 xerces.jar 文件拷贝到<JBuilder>\lib 目录下。

（3）把<struts-console>\com.jamesholmes.console.struts\lib 目录下的 struts-console.jar 文件拷贝到<JBuilder>\lib\ext 目录下。

（4）重启 Jbuilder。

如果要在 Jbuilder 中编辑 Struts 配置文件，只需选择【File】→【Open File】命令，打开需要编辑的 Struts 配置文件，Jbuilder 会自动启动 Struts Console 插件。如图 4-7 所示为 Struts Console 在 JBuilder IDE 中运行的界面。

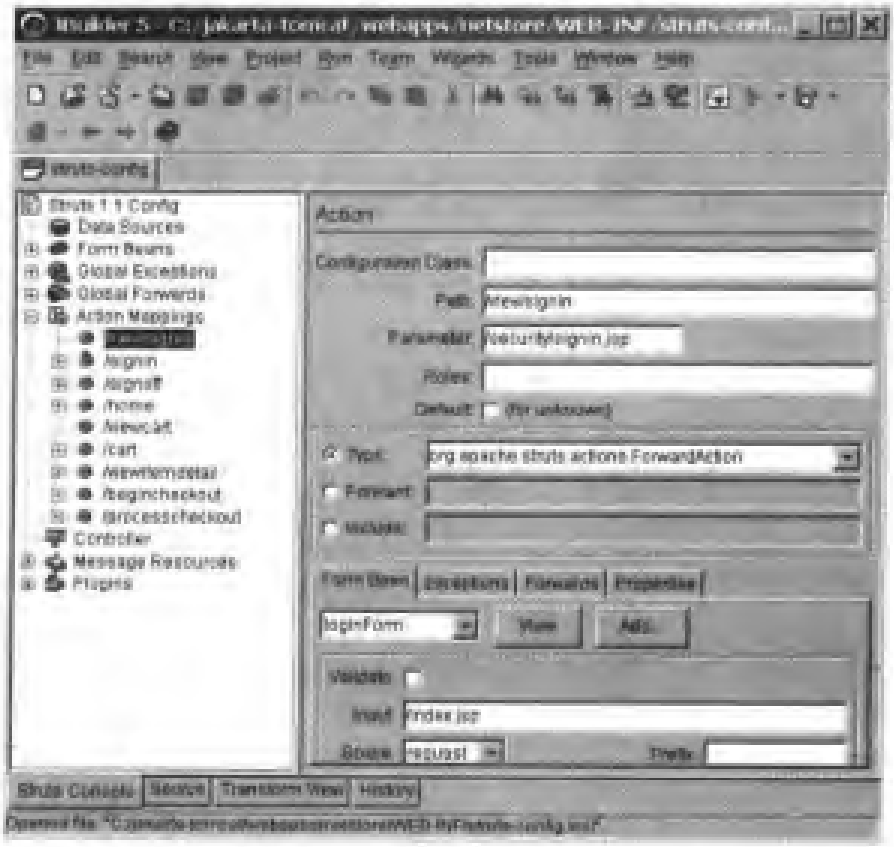

图 4-7　Struts Console 在 JBuilder IDE 中运行的界面

4.6　重新载入配置文件

当 Web 容器首次启动时，会加载并解析 web.xml 文件。默认情况下，在 Web 容器运行时不会监测 web.xml 文件的更新并重新加载它。事实上，有许多 Web 容器由于安全的原因，不支持动态加载 web.xml 文件的功能。

Struts 配置文件也在 Web 容器首次启动时被加载并解析。由于安全的原因，Web 容器在运行时也不会监测 Struts 配置文件的更新并重新加载它。

某些 Struts 应用可能需要在不重启 Web 容器的情况下，提供重新加载 Struts 配置文件的功能。有两种办法可以做到这点。

一种办法是创建一个 Struts Action 类，它能够重新初始化 ActionServlet （为了提高安全性，最好对调用此 Action 类的权限进行限制）。在 ActionServlet 重新初始化时，能够把更新后的 Struts 配置文件的内容重新读入到内存。

第二种办法是创建一个线程，它负责监视配置文件的 lastModifiedTime 属性。这个线程周期性地睡眠，每次睡眠若干秒，醒来后就比较配置文件的当前 lastModifiedTime 属性和保存在内存中的上一次的属性。如果这两个值不一样，说明文件被改动了，于是重新加载应用。与第一种办法相比，这种办法可以避免用户随意地重新加载应用。不过，第二种办法完全由线程来决定何时重新加载应用。

4.7 小　　结

本章详细介绍了 Struts 应用的两个配置文件 web.xml 和 struts-config.xml 的用法。web.xml 适用于所有的 Java Web 应用，它是 Web 应用的发布描述文件。struts-config.xml 文件是 Struts 应用专有的配置文件。在 Struts 应用启动时，会把 Struts 配置文件中的配置信息读入到内存，并把它们存放在 org.apache.struts.config 包中相关 JavaBean 类的实例中。

对于 Struts 框架的初学者，在初次阅读本章内容时，要完全掌握各种元素的配置细节会存在一定难度，因此只需对各种元素的用法有一般性的了解即可。在本书后文介绍 Struts 的各种实用技术时，都会涉及到 Struts 的配置知识，本书后面的内容会帮助读者进一步理解和掌握 Struts 配置元素的用法。

第 5 章　Struts 控制器组件

正如前面所提及的，Struts 控制器组件负责接收用户请求、更新模型，以及选择合适的视图组件返回给用户。控制器组件有助于将模型层和视图层分离，有了这种分离，就可以在同一个模型的基础上得心应手地开发多种类型的视图。Struts 控制器组件主要包括：

- ActionServlet 组件：充当 Struts 框架的中央控制器。
- RequestProcessor 组件：充当每个子应用模块的请求处理器。
- Action 组件：负责处理一项具体的业务。

Struts 框架采用控制器组件来预处理所有的客户请求，这种集中控制方式可以满足 MVC 设计模式的两大需求：

- 首先，控制器在用户输入数据和模型之间充当媒介/翻译者的角色，提供一些通用功能，如安全、登入和其他针对具体用户请求的重要服务，当系统的这些通用功能出现需求变更时，不需要修改整个应用，只需要修改局部的控制器组件即可。
- 其次，由于所有的请求都经过控制器过滤，因此可以降低视图组件之间，以及视图组件和模型组件之间的相互依赖关系，提高每个组件的相对独立性。由控制器组件来决定把合适的视图组件返回给用户，这可以减少视图组件之间直接的、错综复杂的链接关系，使应用更加灵活，便于维护。

Struts 框架采用 ActionServlet 和 RequestProcessor 组件进行集中控制，并采用 Action 组件来处理单项业务。本章主要以 netstore 应用为例，对这些控制器组件进行了更深入的探讨。netstore 应用是一个充分运用了 Struts 各种技术的综合例子，它实现了一个购物网站，更加贴近于实际应用。它的发布和运行步骤请参见本书的附录 D（发布和运行 netstore 应用）。

5.1　控制器组件的控制机制

Struts 的控制器组件主要完成以下任务：

- 接收用户请求。
- 根据用户请求，调用合适的模型组件来执行相应的业务逻辑。
- 获取业务逻辑执行结果。
- 根据当前状态以及业务逻辑执行结果，选择合适的视图组件返回给用户。

在 Struts 框架中，由一系列组件来共同完成控制任务。图 5-1 显示了控制器组件的类框图。

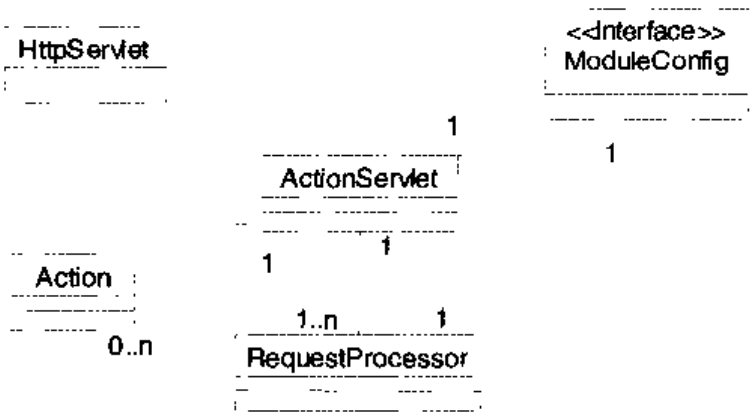

图 5-1 控制器组件的类框图

5.1.1 ActionServlet 类

org.apache.struts.action.ActionServlet 类是 Struts 框架的核心控制器组件，所有的用户请求都先由 ActionServlet 来处理，然后再由 ActionServlet 把请求转发给其他组件。Struts 框架只允许在一个应用中配置一个 ActionServlet 类，在应用的生命周期中，仅创建 ActionServlet 类的一个实例，这个 ActionServlet 实例可以同时响应多个用户请求。

1. Struts 框架初始化过程

Servlet 容器在启动时，或者用户首次请求 ActionServlet 时加载 ActionServlet 类。在这两种情况下，Servlet 容器都会在 ActionServlet 被加载后立即执行它的 init()方法，这可以保证当 ActionServlet 处理用户请求时已经被初始化。以下是 ActionServlet 的 init()方法完成的初始化流程。

（1）调用 initInternal()方法，初始化 Struts 框架内在的消息资源，如与系统日志相关的通知、警告和错误消息。

（2）调用 initOther()方法，从 web.xml 文件中加载 ActionServlet 的初始化参数，如 config 参数。

（3）调用 initServlet()方法，从 web.xml 文件中加载 ActionServlet 的 URL 映射信息。此外还会注册 web.xml 和 Struts 配置文件所使用的 DTD 文件，这些 DTD 文件用来验证 web.xml 和 Struts 配置文件的语法。

（4）调用 initModuleConfig()方法，加载并解析默认子应用模块的 Struts 配置文件；创建 ModuleConfig 对象，把它存储在 ServletContext 中。

（5）调用 initModuleMessageResources()方法，加载并初始化默认子应用模块的消息资源；创建 MessageResources 对象，把它存储在 ServletContext 中。

（6）调用 initModuleDataSources()方法，加载并初始化默认子应用模块的数据源。如

果在 Struts 配置文件中没有定义<data-sources>元素，就忽略这一流程。

（7）调用 initModulePlugins()方法，加载并初始化默认子应用模块的所有插件。

（8）当默认子应用模块被成功地初始化后，如果还包括其他子应用模块，将重复流程（4）～流程（7），分别对其他子应用模块进行初始化。

如图 5-2 所示为 ActionServlet 类的 init()方法时序图，它直观显示了以上所描述的初始化流程。

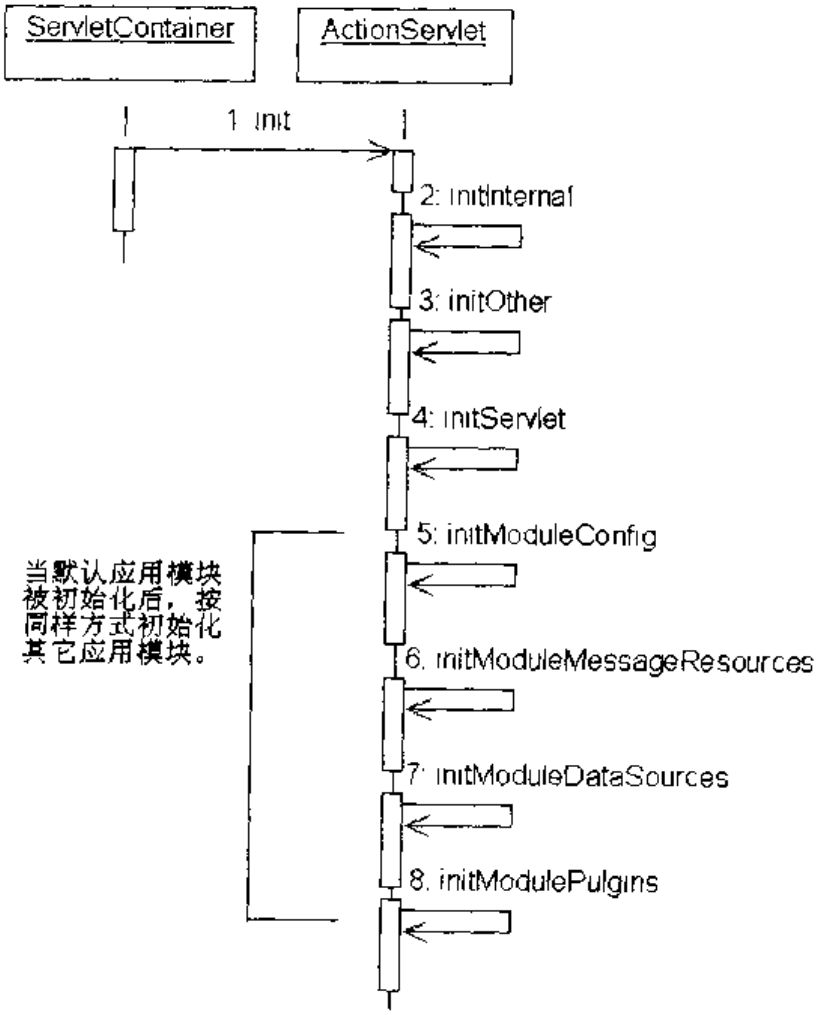

图 5-2　ActionServlet 的 init()方法的时序图

2．ActionServlet 的 process()方法

当 ActionServlet 实例接收到 HTTP 请求后，在 doGet()或 doPost()方法中都会调用 process()方法来处理请求。以下是 ActionServlet 的 process()方法的源代码：

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

    ModuleUtils.getInstance().selectModule(request, getServletContext());
    getRequestProcessor(getModuleConfig(request)).process(request, response);
}
```

process()方法本身看上去并不复杂，但是它所调用的其他方法的实现比较复杂。在process()方法中，首先调用 org.apache.struts.util.ModuleUtils 类的 selectModule()方法，这个方法选择负责处理当前请求的子应用模块，然后把与子应用模块相关的 ModuleConfig 和 MessageResources 对象存储到 request 范围中，这使得框架的其余组件可以方便地从 request 范围中读取这些对象，从而获取应用配置信息和消息资源。

process()方法的第二步操作为获得 RequestProcessor 类的实例，然后调用 RequestProcessor 类的 process()方法，来完成实际的预处理请求操作。

3．扩展 ActionServlet 类

在 Struts 1.1 之前的版本中，ActionServlet 类本身包含了很多处理请求的代码。从 Struts 1.1 开始，多数功能被移到 org.apache.struts.action.RequestProcessor 类中，以便减轻 ActionServlet 类的控制负担。在 5.1.2 小节中将讨论 RequestProcessor 类。

尽管新版本的 Struts 框架允许在应用中创建扩展 ActionServlet 类的子类，但是这在多数情况下没有必要，因为控制器的多数控制功能位于 RequestProcessor 类中。

如果实际应用确实需要创建自己的 ActionServlet 类，则可以创建一个 ActionServlet 类的子类，然后在 web.xml 文件中配置这个客户化 ActionServlet 类。例程 5-1 是一个扩展 ActionServlet 的例子，它覆盖了 init()方法。

例程 5-1　扩展 ActionServlet 的 ExtendedActionServlet 类

```
package netstore.framework;

import javax.servlet.ServletException;
import javax.servlet.UnavailableException;
import org.apache.struts.action.ActionServlet;
import netstore.service.INetstoreService;
import netstore.service.NetstoreServiceImpl;
import netstore.framework.util.IConstants;
import netstore.framework.exceptions.DatastoreException;
/**
 * Extend the Struts ActionServlet to perform your own special
 * initialization.
 */
public class ExtendedActionServlet extends ActionServlet {

  public void init()throws ServletException {

    // Make sure to always call the super's init()first
    super.init();

    // do some custom operation
    ......
  }
}
```

以上自定义的 ActionServlet 类覆盖了父类的 init()方法，在 init()方法中执行了一些客户化操作。如果覆盖了 init()方法，应该确保首先调用 super.init()，它保证 ActionServlet 的默认初始化操作被执行。除了覆盖 init()方法外，事实上，还可以根据实际需要覆盖 ActionServlet 的任何其他方法。

如果 Struts 应用使用自定义的 ActionServlet 类，则应该在 web.xml 文件中对其进行配置：

```
<servlet>
    <servlet-name>netstore</servlet-name>
    <servlet-class>
        netstore.framework.ExtendedActionServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>netstore</servlet-name>
    <url-pattern>/action/*</url-pattern>
</servlet-mapping>
```

以上代码的<url-pattern>属性为“/action/*”，表明 ActionServlet 负责处理所有以“/action”为前缀的 URL。例如，如果用户请求的 URL 为 http://localhost:8080/netstore/action/cart，Web 容器将把该请求转发给 ActionServlet。

5.1.2　RequestProcessor 类

对于多应用模块的 Struts 应用，每个子应用模块都有各自的 RequestProcessor 实例。在 ActionServlet 的 process()方法中，一旦选择了正确的子应用模块，就会调用子应用模块的 RequestProcessor 实例的 process()方法来处理请求。在 ActionServlet 调用这个方法时，会把当前的 request 和 response 对象传给它。

Struts 框架只允许应用中存在一个 ActionServlet 类，但是可以存在多个客户化的 RequestProcessor 类，每个子应用模块都可以拥有单独的 RequestProcessor 类。如果想修改 RequestProcessor 类的一些默认功能，可以覆盖 RequestProcessor 基类中的相关方法。

1．RequestProcessor 类的 process()方法

RequestProcessor 类的 process()方法负责实际的预处理请求操作，例程 5-2 为 process()方法的源代码。

例程 5-2　RequestProcessor 类的 process()方法

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

    // Wrap multipart requests with a special wrapper
```

```
request = processMultipart(request);

// Identify the path component we will use to select a mapping
String path = processPath(request, response);
if (path == null) {
    return;
}
if (log.isDebugEnabled()) {
    log.debug("Processing a '" + request.getMethod() +
              "' for path '" + path + "'");
}

// Select a Locale for the current user if requested
processLocale(request, response);

// Set the content type and no-caching headers if requested
processContent(request, response);
processNoCache(request, response);

// General purpose preprocessing hook
if (!processPreprocess(request, response)) {
    return;
}

// Identify the mapping for this request
ActionMapping mapping = processMapping(request, response, path);
if (mapping == null) {
    return;
}

// Check for any role required to perform this action
if (!processRoles(request, response, mapping)) {
    return;
}

// Process any ActionForm bean related to this request
ActionForm form = processActionForm(request, response, mapping);
processPopulate(request, response, form, mapping);
if (!processValidate(request, response, form, mapping)) {
    return;
}

// Process a forward or include specified by this mapping
if (!processForward(request, response, mapping)) {
    return;
```

```
        }
        if (!processInclude(request, response, mapping)) {
            return;
        }

        // Create or acquire the Action instance to process this request
        Action action = processActionCreate(request, response, mapping);
        if (action == null) {
            return;
        }

        // Call the Action instance itself
        ActionForward forward =
            processActionPerform(request, response,
                                 action, form, mapping);

        // Process the returned ActionForward instance
        processForwardConfig(request, response, forward);
    }
```

RequestProcessor 类的 process()方法依次执行以下流程。

（1）调用 processMultipart()方法。如果 HTTP 请求方式为 POST，并且请求的 contentType 属性以“multipart/form-data”开头，标准的 HttpServletRequest 对象将被重新包装，以方便处理“multipart”类型的 HTTP 请求。如果请求方式为 GET，或者 contentType 属性不是“multipart”，就直接返回原始的 HttpServletRequest 对象。

（2）调用 processPath()方法，获得请求 URI 的路径，这一信息可用于选择合适的 Struts Action 组件。

（3）调用 processLocale()方法，当 ControllerConfig 对象的 locale 属性为 true，将读取用户请求中包含的 Locale 信息，然后把 Locale 实例保存在 session 范围内。

（4）调用 processContent()方法，读取 ControllerConfig 对象的 contentType 属性，然后调用 response.setContentType(contentType)方法，设置响应结果的文档类型和字符编码。processContent()方法的代码如下：

```
    protected void processContent(HttpServletRequest request,
                                  HttpServletResponse response) {
        String contentType = moduleConfig.getControllerConfig().getContentType();
        if (contentType != null) {
            response.setContentType(contentType);
        }
    }
```

（5）调用 processNoCache() 方法，读取 ControllerConfig 对象的 nocache 属性，如果 nocache 属性为 true，在响应结果中将加入特定的头参数：Pragma、Cache-Control 和 Expires，

防止页面被存储在客户浏览器的缓存中。processNoCache()方法的代码如下：

```
protected void processNoCache(HttpServletRequest request,
                              HttpServletResponse response) {
    if (moduleConfig.getControllerConfig().getNocache()) {
        response.setHeader("Pragma", "No-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", 1);
    }
}
```

（6）调用 processPreprocess()方法。该方法不执行任何操作，直接返回 true。子类可以覆盖这个方法，执行客户化的预处理请求操作。本书的 8.3.2 小节（扩展 RequestProcessor 类）详细介绍了子类覆盖 processPreprocess()方法的流程。

（7）调用 processMapping()方法，寻找和用户请求的 URI 匹配的 ActionMapping。如果不存在这样的 ActionMapping，则向用户返回恰当的错误消息。

（8）调用 processRoles()方法，先判断是否为 Action 配置了安全角色，如果配置了安全角色，就调用 isUserInRole()方法判断当前用户是否具备必需的角色；如果不具备，就结束请求处理流程，向用户返回恰当的错误消息。

（9）调用 processActionForm()方法，先判断是否为 ActionMapping 配置了 ActionForm，如果配置了 ActionForm，就先从 ActionForm 的存在范围内寻找该 ActionForm 实例；如果不存在，就创建一个实例。接下来把它保存在合适的范围中，保存时使用的属性 key 为 ActionMapping 的 name 属性。

（10）调用 processPopulate()方法。如果为 ActionMapping 配置了 ActionForm，就先调用 ActionForm 的 reset()方法，再把请求中的表单数据组装到 ActionForm 中。

（11）调用 processValidate()方法，如果为 ActionMapping 配置了 ActionForm，并且 ActionMapping 的 validate 属性为 true，就调用 ActionForm 的 validate()方法。如果 validate()方法返回的 ActionErrors 对象中包含 ActionMessage 对象，说明表单验证失败，就把 ActionErrors 对象存储在 request 范围内，再把请求转发到 ActionMapping 的 input 属性指定的 Web 组件。如果 ActionForm 的 validate()方法执行表单验证成功，就继续执行下一步请求处理流程。

（12）调用 processForward()方法，判断是否在 ActionMapping 中配置了 forward 属性。如果配置了这个属性，就调用 RequestDispatcher 的 forward()方法，请求处理流程结束，否则继续下一步。

（13）调用 processInclude()方法，判断是否在 ActionMapping 中配置了 include 属性。如果配置了这个属性，就调用 RequestDispatcher 的 include()方法，请求处理流程结束，否则继续下一步。

在本书的 4.3.7 小节（<action-mappings>元素）中提到<action>元素的 forward、include 和 type 属性相互排斥，也就是说只能设置其中的一项。RequestProcessor 类的 processForward()方法和 processInclude()方法分别处理<action>元素的 forward 和 include 属性。

（14）调用 processActionCreate()方法，先判断是否在 Action 缓存中存在这个 Action 实例，如果不存在，就创建一个 Action 实例，把它保存在 Action 缓存中。

（15）调用 processActionPerform()方法，该方法再调用 Action 实例的 execute()方法。execute()方法位于 try/catch 代码中，以便捕获异常。processActionPerform()方法的代码如下：

```
protected ActionForward
    processActionPerform(HttpServletRequest request,
                         HttpServletResponse response,
                         Action action,
                         ActionForm form,
                         ActionMapping mapping)
    throws IOException, ServletException {
      try {
          return (action.execute(mapping, form, request, response));
      } catch (Exception e) {
          return (processException(request, response. e,form,mapping));
      }
  }
```

（16）调用 processActionForward()方法，把 Action 的 execute()方法返回的 ActionForward 对象作为参数传给它。processActionForward()根据 ActionForward 对象包含的请求转发信息来执行请求转发或重定向。

在 RequestProcessor 类的 process()方法中，会访问 ControllerConfig、ActionMapping 和 ActionForward 实例的属性。ControllerConfig 类和 Struts 配置文件的<controller>元素对应，ActionMapping 类和<action>元素对应，ActionForward 类和<forward>元素对应。process()方法通过访问这三个类的实例的属性来获得相关的配置信息。

2．扩展 RequestProcessor 类

开发人员可以很方便地创建客户化的 RequestProcessor 类。下面举例说明如何扩展 RequestProcessor 类。假定应用允许用户在同一个会话的任何时候都可以改变它的 Locale 设置，在 RequestProcessor 类中定义的 processLocale()方法的默认行为是：仅仅当 session 范围内不存在 Locale 实例的时候，才会把 Locale 实例保存在 session 范围内，这通常发生在处理同一个会话中的第一个请求时。

例程 5-3 为一个客户化的 RequestProcessor 类的例子，它检查每次用户发出的 HTTP 请求中包含的 Locale 信息，每当 Locale 发生变化时，就会把新的 Locale 实例保存在 session 范围内。这样，用户在同一个会话的任何时候修改 Locale 设置都会生效。

例程 5-3　CustomRequestProcessor.java

```
package netstore.framework;

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

```
import org.apache.struts.action.Action;
import org.apache.struts.action.RequestProcessor;
import org.apache.struts.Globals;
/**
 * A customized RequestProcessor that checks the user's preferred locale
 * from the request each time. If a Locale is not in the session or
 * the one in the session doesn't match the request, the Locale in the
 * request is set in the session.
 */
public class CustomRequestProcessor extends RequestProcessor {

  protected void processLocale(HttpServletRequest request,
                               HttpServletResponse response) {

    // Are we configured to select the Locale automatically?
    if (!moduleConfig.getControllerConfig().getLocale()) {
      return;
    }

    // Get the Locale (if any) that is stored in the user's session
    HttpSession session = request.getSession();
    Locale sessionLocale = (Locale)session.getAttribute(Globals.LOCALE_KEY);

    // Get the user's preferred Locale from the request
    Locale requestLocale = request.getLocale();

    // If was never a Locale in the session or it has changed, set it
    if (sessionLocale == null || (sessionLocale != requestLocale) ){
      if (log.isDebugEnabled()) {
        log.debug(" Setting user locale '" + requestLocale + "'");
      }
      // Set the new Locale into the user's session
      session.setAttribute( Globals.LOCALE_KEY, requestLocale );
    }
  }
}
```

在 Struts 配置文件中，<controller>元素的 processorClass 属性用于配置 RequestProcessor 类：

```
<controller
  contentType="text/html;charset=GB2312"
  locale="true"
  nocache="true"
  processorClass="netstore.framework.CustomRequestProcessor"/>
```

本书的 8.3.2 小节（扩展 RequestProcessor 类）还会介绍通过覆盖 processPreprocess()方法来扩展 RequestProcessor 类。

5.1.3　Action 类

Action 类是用户请求和业务逻辑之间的桥梁。每个 Action 充当客户的一项业务代理。在 RequestProcessor 类预处理请求时，在创建了 Action 的实例后，就调用自身的 processActionPerform()方法，该方法再调用 Action 类的 execute()方法。

Action 的 execute()方法调用模型的业务方法，完成用户请求的业务逻辑，然后根据执行结果把请求转发给其他合适的 Web 组件。例程 5-4 为 netstore 应用的 LoginAction 的源程序。

例程 5-4　LoginAction.java

```
package netstore.security;

import java.util.Locale;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import netstore.customer.view.UserView;
import netstore.framework.exceptions.BaseException;
import netstore.framework.SessionContainer;
import netstore.framework.NetstoreBaseAction;
import netstore.framework.util.IConstants;
import netstore.service.INetstoreService;

/**
 * Implements the logic to authenticate a user for the Netstore application.
 */
public class LoginAction extends NetstoreBaseAction {
  /**
   * Called by the controller when the user attempts to log in to the
   * Netstore application.
   */
  public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response )
  throws Exception{

    // The email and password should have already been validated by the ActionForm
    String email = ((LoginForm)form).getEmail();
    String password = ((LoginForm)form).getPassword();
```

```
        // Log in through the security service
        INetstoreService serviceImpl = getNetstoreService();
        UserView userView = serviceImpl.authenticate(email, password);

        // Create a single container object to store user data
        SessionContainer existingContainer = null;
        HttpSession session = request.getSession(false);
        if ( session != null ){
          existingContainer = getSessionContainer(request);
          session.invalidate();
        }else{
          existingContainer = new SessionContainer();
        }

        // Create a new session for the user
        session = request.getSession(true);

        // Store the UserView in the container and store the container in the session
        existingContainer.setUserView(userView);
        session.setAttribute(IConstants.SESSION_CONTAINER_KEY, existingContainer);

        // Return a Success forward
        return mapping.findForward(IConstants.SUCCESS_KEY);
      }
    }
```

LoginAction 的 execute()方法首先从 LoginForm 中读取 email 和 password 属性，把它们传给 INetstoreService 的 authenticate()方法，INetstoreService 为控制器组件提供了访问模型的业务代理接口。在本书的 6.5.5 小节（联合使用业务代理和 DAO 模式）中介绍了 INetstoreService 接口的实现。

如果 authenticate()方法没有抛出任何异常，就会创建一个新的 HttpSession 对象，并且把一个包含用户信息的 UserView 对象保存在 session 范围内。UserView 类包含了供视图访问的用户信息，如 firstName 和 lastName 属性等。UserView 用于在模型层和视图层之间传递用户信息，这种类型的 JavaBean 被称为 DTO，用于不同层之间的数据传输。关于 DTO 的更多知识，可以参考本书的 7.2.1 小节（DTO 数据传输对象）。例程 5-5 是 UserView 的源程序。

例程 5-5　UserView.java

```
package netstore.customer.view;

import netstore.framework.view.BaseView;
/**
 * Mutable data representing a user of the system.
 */
```

```
public class UserView extends BaseView {
   private String lastName;
   private String firstName;
   private String emailAddress;
   private String creditStatus;

   public UserView(){
      super();
   }

   public String getFirstName(){
      return firstName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }

   public void setLastName(String lastName) {
      this.lastName = lastName;
   }

   public String getLastName(){
      return lastName;
   }

   public String getEmailAddress(){
      return emailAddress;
   }

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

   public void setCreditStatus(String creditStatus) {
      this.creditStatus = creditStatus;
   }
   public String getCreditStatus(){
      return creditStatus;
   }
}
```

LoginAction 还使用了 SessionContainer 类。这个类用来存放 session 范围内的共享数据。通常，开发人员可以直接把共享数据存放在 HttpSession 中，然后调用它的 getAttribute()和 setAttribute()方法来读取或存放数据，本应用特地采用 SessionContainer 类来简化这种读取或存放 session 范围内共享数据的行为。HttpSession、SessionContainer 和 UserView 的关系

为：UserView 存放在 SessionContainer 中，SessionContainer 存放在 HttpSession 中，如图 5-3 所示。关于 SessionContainer 类的实现参见本书的 8.5 节（扩展模型组件）。

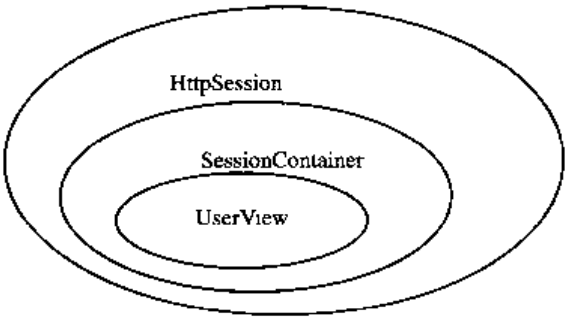

图 5-3 HttpSession、SessionContainer 和 UserView 的关系

1．Action 类缓存

为了确保线程安全（thread-safe），在一个应用的生命周期中，Struts 框架只会为每个 Action 类创建一个 Action 实例。所有的客户请求共享同一个 Action 实例，并且所有请求线程可以同时执行它的 execute()方法。

RequestProcessor 类包含一个 HashMap，作为存放所有 Action 实例的缓存，每个 Action 实例在缓存中存放的属性 key 为 Action 类名。在 RequestProcessor 类的 processActionCreate() 方法中，首先检查在 HashMap 中是否存在 Action 实例，如果存在，就返回这个实例；否则，就创建一个新的 Action 实例。创建 Action 实例的代码位于同步代码块中，以保证只有一个线程创建 Action 实例。一旦线程创建了 Action 实例并把它存放到 HashMap 中，以后所有的线程都会直接使用这个缓存中的实例。以下是 processActionCreate()方法的代码：

```
protected HashMap actions = new HashMap();
protected Action processActionCreate(HttpServletRequest request,
                                     HttpServletResponse response,
                                     ActionMapping mapping)
    throws IOException {

    // Acquire the Action instance we will be using (if there is one)
    String className = mapping.getType();
    if (log.isDebugEnabled()) {
        log.debug(" Looking for Action instance for class " + className);
    }

    Action instance = null;
    synchronized (actions) {
        // Return any existing Action instance of this class
        instance = (Action) actions.get(className);
        if (instance != null) {
            if (log.isTraceEnabled()) {
                log.trace("    Returning existing Action instance");
            }
            return (instance);
```

```
            }

            // Create and return a new Action instance
            if (log.isTraceEnabled()) {
                log.trace("    Creating new Action instance");
            }
            try {
                instance = (Action) RequestUtils.applicationInstance(className);
                // :TODO: Maybe we should propagate this exception
                // instead of returning null.
            } catch (Exception e) {
                log.error(
                    getInternal().getMessage("actionCreate", mapping.getPath()),
                    e);
                response.sendError(
                    HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
                    getInternal().getMessage("actionCreate", mapping.getPath()));
                return (null);
            }

            instance.setServlet(this.servlet);
            actions.put(className, instance);
        }
        return (instance);
    }
```

在 Java API 中，java.util.Vector 和 java.util.ArrayList 都是集合类，java.util.Hashtable 和 java.util.HashMap 都是 Map 类，其中 Vector 和 Hashtable 的存取元素的方法采用了 Java 的同步机制，而 ArrayList 和 HashMap 没有使用同步机制。同步机制可以有效地避免共享资源的竞争，但是滥用同步机制会对应用的性能造成影响。因此，应该根据实际需要，谨慎地选择使用 Vector 或 ArrayList，以及 Hashtable 或 HashMap。

2．ActionForward 类

Action 类的 execute()方法返回一个 ActionForward 对象。ActionForward 对象代表了 Web 资源的逻辑抽象，这里的 Web 资源可以是 JSP 页、Java Servlet 或 Action。从 execute()方法中返回 ActionForward 对象有两种方法：

- 在 execute()方法中动态创建一个 ActionForward 实例：

```
return   new ActionForward("Failure ", "/security/signin.jsp",true);
```

以上 ActionForward 构造方法的第一个参数代表 ActionForward 实例的逻辑名，第二个参数指定转发路径，第三个参数指定是否进行重定向（如果为 true，则代表重定向；如果为 false，则代表请求转发）。

- 在 Struts 配置文件中配置<forward>元素：

```
<action
    path="/signin"
    type="netstore.security.LoginAction"
    scope="request"
    name="loginForm"
    validate="true"
    input="/security/signin.jsp">
    <forward name="Success" path="/action/home"/>
    <forward name="Failure" path="/security/signin.jsp" redirect="true"/>
</action>
```

配置了<forward>元素后，在 Struts 框架初始化时就会创建存放<forward>元素配置信息的 ActionForward 对象。

在 execute()方法中只需调用 ActionMapping 实例的 findForward()方法，来获得特定的 ActionForward 实例：

```
return mapping.findForward( "Failure" );
```

以上 ActionMapping 的 findForward()方法的参数值为“Failure”，它和 name 属性为“Failure”的 ActionForward 对象匹配。findForward()方法先调用 findForwardConfig()方法，在 Action 级别（即<action>元素内的<forward>子元素）寻找匹配的 ActionForward 对象。如果没有，再在全局级别（即<global-forwards>元素内的<forward>子元素）中寻找匹配的 ActionForward 对象。如果找到，就返回该 ActionForward 对象。以下是 ActionMapping 类的 findForward()方法的源程序：

```
public ActionForward findForward(String name) {
        ForwardConfig config = findForwardConfig(name);
        if (config == null) {
            config = getModuleConfig().findForwardConfig(name);
        }
        return ((ActionForward) config);
    }
```

如果 findForward()方法没有找到匹配的 ActionForward 对象，它不会抛出异常，而是返回 null。在浏览器端，用户将收到一个空白页。

采用第二种方式，无需在程序中硬编码来指定转发资源的物理路径，而是在配置文件中配置转发资源，程序中只需引用转发资源的逻辑名即可，这提高了应用的灵活性和可维护性。

3. 创建支持多线程的 Action 类

在 Struts 应用的生命周期中，只会为每个 Action 类创建一个实例，所有的客户请求共享这个实例。因此，必须保证在多线程环境中，Action 也能正常工作。保证线程安全的重要原则是在 Action 类中仅仅使用局部变量，谨慎地使用实例变量。

如果在 Action 的 execute()方法中定义了局部变量，对于每个调用 execute()方法的线程，

Java 虚拟机会在每个线程的堆栈中创建局部变量，因此每个线程拥有独立的局部变量，不会被其他线程共享。当线程执行完 execute()方法时，它的局部变量就会被销毁。

如果在 Action 类中定义了实例变量，那么在 Action 实例的整个生命周期中，这个实例变量被所有的请求线程共享。因此不能在 Action 类中定义代表特定客户状态的实例变量，例如不能定义购物车实例变量，因为每个用户拥有各自不同的购物车。

在 Action 类中定义的实例变量代表了可以被所有请求线程访问的共享资源。为了避免共享资源的竞争，在必要的情况下，需要采用 Java 同步机制对访问共享资源的代码块进行同步。

4. Action 类的安全

在某些情况下，如果 Action 类执行的功能非常重要，则只允许具有特定权限的用户才能访问该 Action。为了防止未授权的用户来访问 Action，可以在配置 Action 时指定安全角色：

```
<action     path="/sample"
            type="SampleAction"
            roles="manager,admin" >
</action>
```

<action>元素的 roles 属性指定访问这个 Action 的用户必须具备的安全角色，多个角色之间以逗号隔开。以上代码表明，访问 SampleAction 必须具备 manager 和 admin 角色。

RequestProcessor 类在预处理请求时会调用自身的 processRoles()方法，该方法先检查在配置文件中是否为 Action 配置了安全角色，如果配置了安全角色，就调用 HttpServletRequest 的 isUserInRole()方法，来判断用户是否具备了必要的安全角色。如果不具备，就直接向客户端返回错误。以下是 processRoles()方法的源程序：

```
protected boolean processRoles(HttpServletRequest request,
                               HttpServletResponse response,
                               ActionMapping mapping)
    throws IOException, ServletException {

    // Is this action protected by role requirements?
    String roles[] = mapping.getRoleNames();
    if ((roles == null) || (roles.length < 1)) {
        return (true);
    }

    // Check the current user against the list of required roles
    for (int i = 0; i < roles.length; i++) {
        if (request.isUserInRole(roles[i])) {
            if (log.isDebugEnabled()) {
                log.debug(" User '" + request.getRemoteUser() +
                    "' has role '" + roles[i] + "', granting access");
            }
            return (true);
```

```
            }
        }

        // The current user is not authorized for this action
        if (log.isDebugEnabled()) {
            log.debug(" User '" + request.getRemoteUser() +
                      "' does not have any required role, denying access");
        }
        response.sendError(HttpServletResponse.SC_BAD_REQUEST,
                           getInternal().getMessage("notAuthorized",
                                                    mapping.getPath()));
        return (false);
    }
```

关于如何在 Tomcat 中为 Web 应用配置安全域，即配置用户和安全角色的映射关系，可以参考 Tomcat 的相关文档：

<CATALINA_HOME>\webapps\tomcat-docs\realm-howto.html

以上解决方案的局限在于必须事先配置 Action 的安全角色。对于有些应用，需要在应用运行时动态地添加或删除角色，在这种情况下，光靠配置解决不了这一问题，需要在应用中以编程的方式来保证 Action 类的安全访问。

5.2 使用内置的 Struts Action 类

Struts 提供了一些现成的 Action 类，在 Struts 应用中直接使用这些 Action 类可以大大节省开发时间。其中最常用的几个 Action 类为：

- org.apache.struts.actions.ForwardAction
- org.apache.struts.actions.IncludeAction
- org.apache.struts.actions.DispatchAction
- org.apache.struts.actions.LookupDispatchAction
- org.apache.struts.actions.SwitchAction

下面依次讨论这几个 Action 的作用和使用方法。

5.2.1 org.apache.struts.actions.ForwardAction 类

在 JSP 网页中，尽管可以直接通过<jsp:forward>标签把请求转发给其他 Web 组件，如图 5-4 所示，但是 Struts 框架提倡先把请求转发给控制器，再由控制器来负责请求转发，如图 5-5 所示。

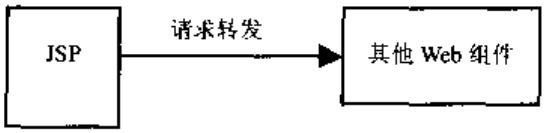

图 5-4　JSP 网页直接把请求转发给其他 Web 组件

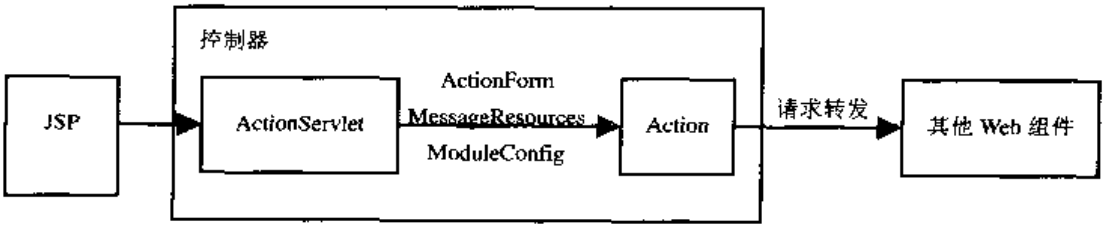

图 5-5　JSP 网页经过控制器把请求转发给其他 Web 组件

由控制器来负责请求转发有以下一些优点：

- 控制器具有预处理请求功能，它能够选择正确的子应用模块来处理请求，并且把子应用模块的 ModuleConfig 和 MessageResources 对象存放在 request 范围内。这样，请求转发的目标 Web 组件就可以正常地访问 ModuleConfig 和 MessageResources 对象。
- 如果 JSP 页面中包含 HTML 表单，那么控制器能够创建和这个表单对应的 ActionForm 对象，把用户输入表单数据组装到 ActionForm 中。如果<action>元素的 validate 属性为 true，那么还会调用 ActionForm 的表单验证方法。控制器把 ActionForm 对象存放在 request 或 session 范围内，这样请求转发的目标 Web 组件也可以访问 ActionForm。
- JSP 网页之间直接相互转发违背了 MVC 的分层原则。按照 MVC 设计思想，控制器负责处理所有请求，然后选择恰当的视图组件返回给用户。如果直接让 JSP 相互调用，控制器就失去了流程控制作用。

对于用户自定义的 Action 类，既可以负责请求转发，还可以充当客户端的业务代理。如果仅仅需要 Action 类提供请求转发功能，则可以使用 org.apache.struts.actions.ForwardAction 类。ForwardAction 类专门用于转发请求，不执行任何其他业务操作。

例如在 netstore 应用中，如果用户在 index.jsp 网页上选择“登入”链接，将把请求转发给“/action/viewsignin”：

```
<html:link page="/action/viewsignin">
```

“/action/viewsignin”以“/action”作为前缀，根据本章 5.1.1 小节的对 netstore 应用的 ActionServlet 的配置，该请求先由 ActionServlet 接收，ActionServlet 再寻找 path 属性为“/viewsignin”的<action>元素：

```
<action
    path="/viewsignin"
    parameter="/security/signin.jsp"
    type="org.apache.struts.actions.ForwardAction"
    scope="request"
```

```
    name="loginForm"
    validate="false"
    input="/index.jsp">
</action>
```

ActionServlet 把请求转发给 ForwardAction，ForwardAction 再把请求转发给<action>元素中 parameter 属性指定的 Web 组件。总之，在 Web 组件之间通过 ForwardAction 类来进行请求转发，可以充分利用 Struts 控制器的预处理请求功能。

此外，也可以通过<action>元素的 forward 属性来实现请求转发，以下代码能完成同样的功能：

```
<action
    path="/viewsignin"
    forward="/security/signin.jsp"
    scope="request"
    name="loginForm"
    validate="false"
    input="/index.jsp">
</action>
```

5.2.2 org.apache.struts.actions.IncludeAction 类

在 JSP 网页中，尽管可以直接通过<include>指令包含另一个 Web 组件，但是 Struts 框架提倡先把请求转发给控制器，再由控制器来负责包含其他 Web 组件。IncludeAction 类提供了包含其他 Web 组件的功能。与 ForwardAction 一样，Web 组件通过 IncludeAction 类来包含另一个 Web 组件，可以充分利用 Struts 控制器的预处理功能。

在 Struts 配置文件中，按以下方式配置 IncludeAction 类：

```
<action
    path="/sample"
    parameter="sampleInclude.jsp"
    type="org.apache.struts.actions.IncludeAction"
    name="sampleForm"
    scope="request"
    input="/sample.jsp" />
```

<action>的 paramter 属性指定需要包含的 Web 组件。此外，也可以通过<action>元素的 include 属性来包含 Web 组件，以下代码能完成同样的功能：

```
<action
    path="/sample"
    include="sampleInclude.jsp"
    name="sampleForm"
    scope="request"
    input="/sample.jsp" />
```

5.2.3　org.apache.struts.actions.DispatchAction 类

通常，在一个 Action 类中只能完成一种业务操作，如果希望在同一个 Action 类中完成一组相关的业务操作，可以使用 DispatchAction 类。例如，与购物车相关的业务操作包括：查看购物车、添加商品、修改商品及数量等。一种设计方案是为每种业务操作创建独立的 Action 类，如 AddItemAction、ViewShoppingCartAction 和 UpdateShoppingCartAction。尽管这种设计方案是可行的，但是这三个 Action 在执行各自的任务中，可能会执行一些相同的操作，比如 AddItemAction 和 UpdateShoppingCartAction 都要进行相同的数据验证。

为了减少重复编程，使应用更加便于维护，可以由同一个 Action 类来完成一组相关的业务操作，DispatchAction 类就提供了这种功能。

创建一个扩展 DispatchAction 类的子类，不必覆盖 execute()方法，而是创建一些实现实际业务操作的方法，这些业务方法都应该和 execute()方法具有同样的方法签名，即它们的参数和返回类型都应该相同，此外也应该声明抛出 Exception。例程 5-6 是一个扩展 DispatchAction 的例子。

例程 5-6　ShoppingCartActions.java

```
package netstore.order;

//import all the necessary packages here
.....
/**
 * Implements all of the functionality for the shopping cart.
 */
public class ShoppingCartActions extends NetstoreDispatchAction {
  /**
   * This method just forwards to the success state, which should represent
   * the shoppingcart.jsp page.
   */
  public ActionForward view(ActionMapping mapping,
                            ActionForm form,
                            HttpServletRequest request,
                            HttpServletResponse response)
    throws Exception {

    // Call to ensure that the user container has been created
    SessionContainer sessionContainer = getSessionContainer(request);
    return mapping.findForward(IConstants.SUCCESS_KEY);
  }

/**
 * This method updates the items and quantities for the shopping cart from the
 * request.
```

```
    */
    public ActionForward update(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
      throws Exception {

      updateItems(request);
      updateQuantities(request);
      return mapping.findForward(IConstants.SUCCESS_KEY);
    }

  /**
    * This method adds an item to the shopping cart based on the id and qty
    * parameters from the request.
    */
    public ActionForward addItem(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
      throws Exception {

      SessionContainer sessionContainer = getSessionContainer(request);

      // Get the id for the product to be added
      String itemId = request.getParameter( IConstants.ID_KEY );
      String qtyParameter = request.getParameter( IConstants.QTY_KEY );

      int quantity;
      if(qtyParameter != null) {
        Locale userLocale = sessionContainer.getLocale();
        Format nbrFormat = NumberFormat.getNumberInstance(userLocale);
        try {
          Object obj = nbrFormat.parseObject(qtyParameter);
          quantity = ((Number)obj).intValue();
        }
        catch(Exception ex) {
          // Set the default quantity
          quantity = 1;
        }
      }

      // Call the Netstore service and ask it for an ItemView for the item
      INetstoreService serviceImpl = getNetstoreService();
      ItemDetailView itemDetailView = serviceImpl.getItemDetailView( itemId );
```

```
        // Add the item to the cart and return
        sessionContainer.getCart().addItem(
            new ShoppingCartItem(itemDetailView, quantity));

        return mapping.findForward(IConstants.SUCCESS_KEY);
    }

    /**
     * Update the items in the shopping cart. Currently, only deletes occur
     * during this operation.
     */
    private void updateItems(HttpServletRequest request) {
        // Multiple checkboxes with the name "deleteCartItem" are on the
        // form. The ones that were checked are passed in the request.
        String[] deleteIds = request.getParameterValues("deleteCartItem");

        // Build a list of item ids to delete
        if(deleteIds != null && deleteIds.length > 0) {
            int size = deleteIds.length;
            List itemIds = new ArrayList();
            for(int i = 0;i < size;i++) {
                itemIds.add(deleteIds[i]);
            }
            // Get the ShoppingCart from the SessionContainer and delete the items
            SessionContainer sessionContainer = getSessionContainer(request);
            sessionContainer.getCart().removeItems(itemIds);
        }
    }

    /**
     * Update the quantities for the items in the shopping cart.
     */
    private void updateQuantities(HttpServletRequest request) {
        Enumeration enum = request.getParameterNames();
        // Iterate through the parameters and look for ones that begin with
        // "qty_". The qty fields in the page were all named "qty_" + itemId.
        // Strip off the id of each item and the corresponding qty value.
        while(enum.hasMoreElements()) {
            String paramName = (String)enum.nextElement();
            if(paramName.startsWith("qty_")) {
                String id = paramName.substring(4, paramName.length());
                String qtyStr = request.getParameter(paramName);
                if(id != null && qtyStr != null) {
                    ShoppingCart cart = getSessionContainer(request).getCart();
                    cart.updateQuantity(id, Integer.parseInt(qtyStr));
                }
```

```
        }
      }
    }
  }
```

ShoppingCartActions 类扩展了 NetstoreDispatchAction，而 NetstoreDispatchAction 又扩展了 DispatchAction 类。ShoppingCartActions 类提供了三种业务方法：addItem()、update()和 view()。

ShoppingCartActions 类中还提供了两种实用方法：updateItems()和 updateQuantities()，这两种方法仅仅在 ShoppingCartActions 类内部调用。

在配置 DispatchAction 类时，需要把 parameter 属性设置为“method”，以下是 ShoppingCartActions 类的配置代码：

```
<action path="/cart"
    type="netstore.order.ShoppingCartActions"
    scope="request"
    input="/order/shoppingcart.jsp"
    validate="false"
    parameter="method">
    <forward name="Success" path="/order/shoppingcart.jsp" redirect="true"/>
</action>
```

把 parameter 的属性值设置为“method”后，当用户请求访问 DispatchAction 时，应该提供 method 请求参数，例如：

```
http://localhost:8080/netstore/action/cart?method=addItem&id=2
```

以上 method 请求参数值为“addItem”，它指定了需要调用的业务方法，因此 DispatchAction 将调用相应的 addItem()方法。

5.2.4 org.apache.struts.actions.LookupDispatchAction 类

LookupDispatchAction 类是 DispatchAction 的子类，在 LookupDispatchAction 类中也可以定义多个业务方法。通常 LookupDispatchAction 主要应用于在一个表单中有多个提交按钮，而这些按钮又有一个共同的名字的场合，这些按钮的名字和具体的 ActionMapping 的 parameter 属性值相对应。下面举例说明如何使用 LookupDispatchAction 类。

首先，在同一个 HTML 表单中定义多个提交按钮。以 shipping.jsp 为例，以下是在同一个 HTML 表单中定义两个按钮的 JSP 代码片断：

```
<html:form action="/processcheckout">
    ......
    <html:submit property="action">
     <bean:message key="button.saveorder"/>
    </html:submit>
```

```
<html:submit property="action">
 <bean:message key="button.checkout"/>
</html:submit>

</html:form>
```

以上两个提交按钮的 property 属性值都为“action”。在<html:submit>标签中嵌套了一个<bean:message>标签，它决定了按钮的 Label 内容，以及当用户提交按钮时调用的 Action 的业务方法。

其次，在 Resource Bundle 中加入和以上<bean:message>标签的 key 属性匹配的文本信息：

```
button.checkout=Check Out
button.saveorder=Save Order
```

这样，在输出网页上，表单的两个按钮的 Label 分别为“Check Out”和“Save Order”。以上 JSP 代码的输出内容为：

```
<form name="checkoutForm"
  method="post"
  action="/netstore/action/processcheckout" >
    ......
  <input type="submit" name="action" value="Save Order">
  <input type="submit" name="action" value="Check Out">
</form>
```

如图 5-6 所示为 shipping.jsp 的输出网页。

图 5-6　shipping.jsp 的输出网页

接着创建一个扩展 LookupDispatchAction 类的子类：

```
public class ProcessCheckoutAction extends LookupDispatchAction
```

在 ProcessCheckoutAction 类中实现 getKeyMethodMap()方法，这个方法返回包含 key/value 数据的 java.util.Map 对象。

Map 对象中的 key 和<bean:message>标签的 key 属性匹配，value 和 LookupDispatch Action 子类的业务方法匹配。

以下为 ProcessCheckoutAction 类的 getKeyMethodMap()方法：

```
protected Map getKeyMethodMap(){
  Map map = new HashMap();
  map.put("button.checkout", "checkout" );
  map.put("button.saveorder", "saveorder" );
  return map;
}
```

根据以上程序，应该在 ProcessCheckoutAction 类中提供两种业务方法：checkout()和 saveorder()方法。当用户单击提交【Check Out】按钮，Struts 框架就会调用 checkout()方法，当用户单击提交【Save Order】按钮时，Struts 框架就会调用 saveorder()方法。

这些业务方法都应该和 execute()方法具有同样的方法签名，即它们的参数和返回类型都应该相同。业务方法声明抛出的异常为 IOException 和 ServletException。checkout()和 saveorder()方法的声明如下：

```
public ActionForward checkout(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)throws
IOException,ServletException

public ActionForward saveorder(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response) throws
IOException,ServletException
```

本书提供的 ProcessCheckoutAction 类并没有真正实现付款或保存订单业务，在 checkout()和 saveorder()方法中都只是返回 payment.jsp。读者可以根据实际应用需求来扩展这两种方法。

最后，在 Struts 配置文件中配置 LookUpDispatchAction：

```
<action path="/processcheckout"
  type="netstore.order.ProcessCheckoutAction"
  scope="request"
  input="/order/shipping.jsp"
  name="checkoutForm"
  validate="true"
  parameter="action">
  <forward name="Success" path="/order/payment.jsp"/>
</action>
```

在以上的<action>元素中，应该把 parameter 属性设置为“action”，使它和<html:submit>标签的 property 属性保持一致。

5.2.5　org.apache.struts.actions.SwitchAction 类

SwitchAction 类用于子应用模块之间的切换。关于多应用模块的配置方法，请参见本书的 4.3.11 小节（配置多应用模块）。如果使用 SwitchAction 类来进行子应用模块之间的切换，只需在 Struts 配置文件中做如下配置：

```
<action-mappings>
    <action path="/toModule"
    type="org.apache.struts.actions.SwitchAction"/>
    ...
</action-mappings>
```

对于请求访问 SwitchAction 的 URL，需要提供两个请求参数：

- prefix：指定子应用模块的前缀，以“/”开头，默认子应用模块的前缀为空字符串“”。
- page：指定被请求 Web 组件的 URI，只需指定相对于被切换后的子应用模块的相对路径。

例如，如果要从默认模块切换到 ModuleB，并把请求转发给 ModuleB 的“/index.do”，可以采用与以下类似的 URL：

```
http://localhost:8080/action/toModule?prefix=/moduleB&page=/index.do
```

如果要从 ModuleB 切换到默认模块，并把请求转发给默认模块的“/index.do”，可以采用与以下类似的 URL：

```
http://localhost:8080/action/toModule?prefix=&page=/index.do
```

对于 netstore 应用中的 ActionServlet，由于它处理以“action”为前缀的 URL，所以此处访问 SwitchAction 的 URI 为“/action/toModule”。

5.3　利用 Token 解决重复提交

在某些情况下，如果用户对同一个 HTML 表单多次提交，Web 应用必须能够判断用户的重复提交行为，以做出相应的处理。例如，对于注册表单，如果用户已经提交表单并且服务器端成功地注册了用户信息，此时用户又通过浏览器的后退功能，退回到原来的页面，重复提交表单，服务器端应该能够识别用户的误操作行为，避免为用户重复注册。

可以利用同步令牌（Token）机制来解决 Web 应用中重复提交的问题，Struts 也给出了一个参考实现。org.apache.struts.action.Action 类中提供了一系列和 Token 相关的方法：

1．protected boolean isTokenValid(javax.servlet.http.HttpServletRequest request)

判断存储在当前用户会话中的令牌值和请求参数中的令牌值是否匹配。如果匹配，就返回 true，否则返回 false。只要符合以下情况之一，就会返回 false：

- 不存在 HttpSession 对象
- 在 session 范围内没有保存令牌值
- 在请求参数中没有令牌值
- 存储在当前用户 session 范围内的令牌值和请求参数中的令牌值不匹配

2．protected void resetToken(javax.servlet.http.HttpServletRequest request)

从当前 session 范围内删除令牌属性。

3．protected void saveToken(javax.servlet.http.HttpServletRequest request)

创建一个新的令牌，并把它保存在当前 session 范围内。如果 HttpSession 对象不存在，就首先创建一个 HttpSession 对象。

具体的 Token 处理逻辑由 org.apache.struts.util.TokenProcessor 类来完成，它的 generateToken(request)方法根据用户会话 ID 和当前系统时间来生成一个惟一的令牌。

下面以 addressbook 应用的 insert 表单为例，介绍利用 Struts 的 Token 机制来避免重复提交表单的流程。

（1）在用户请求 insert.jsp 之前，首先把请求转发给 PrepareInsertAction，它调用 saveToken(request)方法，创建一个新的令牌，并把它保存在当前的 session 范围内。PrepareInsertAction 接着再把请求转发给 insert.jsp。

（2）insert.jsp 中的<html:form>标签的处理类判断在 session 范围内是否存在 Token，如果存在，就在表单中生成一个包含 Token 信息的隐藏字段，这段逻辑由 org.apache.struts.taglib.html.FormTag 类的 renderToken()方法完成：

```
/**
 * Generates a hidden input field with token information, if any.
 * @return A hidden input field containing the token.
 * @since Struts 1.1
 */
protected String renderToken() {
    StringBuffer results = new StringBuffer();
    HttpSession session = pageContext.getSession();

    if (session != null) {
        String token =
            (String) session.getAttribute(Globals.TRANSACTION_TOKEN_KEY);
```

```
                if (token != null) {
                    results.append("<input type=\"hidden\" name=\"");
                    results.append(Constants.TOKEN_KEY);
                    results.append("\" value=\"");
                    results.append(token);
                    if (this.isXhtml()) {
                        results.append("\" />");
                    } else {
                        results.append("\">");
                    }
                }
            }
            return results.toString();
        }
```

当用户收到 insert.jsp 的网页后，在源文件中会看到表单中定义了一个包含 Token 信息的隐藏字段：

```
<form name="insertForm" method="post" action="/addressbook/insert.do">
  <input type="hidden" name="org.apache.struts.taglib.html.TOKEN"
  value="1398bc4b43b6fef12e6bdf3ff4627fcb">
  ......
</form>
```

Action 类的 isTokenValid(request)方法在判断 Token 是否有效时，实际上就是把这个隐藏字段的值和当前用户会话中的令牌值做比较。

（3）在用户提交了表单后，由 InsertAction 处理请求。在 InsertAction 中，首先调用 isTokenValid(request)方法，判断当前用户会话中的令牌值和请求参数中的令牌值是否匹配。如果不匹配，就生成错误信息，并调用 saveToken(request)方法，创建一个新的令牌，然后返回。如果匹配，就调用 resetToken(request)方法，从当前会话中删除 Token，然后执行插入数据的操作。

这样，在用户提交了表单后，如果又通过浏览器的后退功能，退回到刚才的 insert.jsp 网页，如再次提交表单，其请求将由 InsertAction 来处理。InsertAction 先调用 isTokenValid(request)方法判断当前用户会话中的令牌值和请求参数中的令牌值是否匹配。此时，由于当前会话中已经不存在 Token，所以 isTokenValid(request)方法返回 false。

下面对原来的 addressbook 应用进行一些修改，使它的 insert 表单不允许重复提交，修改后的源程序位于本书配套光盘的 sourcecode/addressbook/version2 目录下。以下是修改步骤。

（1）创建 PrepareInsertAction，它的 execute()方法的代码如下：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
        throws Exception {
                saveToken(request);
                return mapping.findForward(Constants.FORWARD_SUCCESS);
}
```

（2）修改 InsertAction，在 execute()方法的开头添加判断 Token 是否有效的逻辑：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)throws Exception {

        ActionMessages errors = new ActionMessages();
        if(!isTokenValid(request)){
           errors.add(ActionMessages.GLOBAL_MESSAGE,
                              new ActionMessage("error.invalid.token"));
           saveErrors(request, errors);
           saveToken(request);
           return (new ActionForward(mapping.getInput()));
        }
        else{
              resetToken(request);
        }
        // insert the record
        ……
}
```

根据以上代码，当用户首次提交表单时，isTokenValid()方法返回 true。如果又通过浏览器的后退功能，退回到刚才的 insert.jsp 网页，再次提交表单，isTokenValid()方法返回 false，此时会生成错误消息，并调用 saveToken()方法创建一个新的 Token，然后返回到原来的 insert.jsp 网页。如果此时用户再次提交表单，则 isTokenValid()方法又会返回 true。

（3）修改 Struts 配置文件，配置如下转发关系：

```
PrepareInsertAction → insert.jsp → InsertAction
```

首先修改<global-forwards>元素中的一项<forward>子元素。

修改前：

```
<forward    name="insert"   path="/insert.do"/>
```

修改后：

```
<forward    name="insert"   path="/prepareinsert.do"/>
```

接着配置 PrepareInsertAction：

```
<action     path="/prepareinsert"
                type="addressbook.actions.PrepareInsertAction" >
        <forward name="success" path="/insert.jsp"/>
</action>
```

（4）修改资源文件，在 ApplicationResources.properties 文件中加入验证 Token 无效时的错误信息：

```
error.invalid.token=<li>You are not allowed to submit the save form twice</li>
```

5.4 实　用　类

在创建 Web 应用时，有许多检索和处理 HTTP 请求的操作是重复的。为了提高应用代码的可重用性，减少冗余，Struts 框架提供了一组提供这些通用功能的实用类，它们可以被所有的 Struts 应用共享。

5.4.1 RequestUtils 类

org.apache.struts.util.RequestUtils 为 Struts 控制框架提供了一些处理请求的通用方法。例如，ActionServlet 类就调用了它的 selectModule()方法，RequestUtils 类中的所有方法都是线程安全的，在这个类中没有定义任何实例变量，所有的方法都被声明为 static 类型。因此，不必创建 RequestUtils 类的实例，可以直接通过类名来访问这些方法。表 5-1 列出了 RequestUtils 类的一些常用方法。

表 5-1　RequestUtils 类的常用方法

方　法	描　述
absoluteURL(HttpServletRequest request,String url)	创建并返回绝对 URL 路径，参数 path 指定相对于上下文（context-relative）的相对路径
createActionForm(HttpServletRequest request, ActionMapping mapping, ModuleConfig moduleConfig, ActionServlet servlet)	先从 request 或 session 范围内查找该 ActionForm，如果存在，就直接将它返回，否则先创建它的实例，把它保存在 request 或 session 范围内，再把它返回，mapping 参数包含了<action>元素的配置信息，例如它的 scope 属性指定 ActionForm 的范围
populate(Object bean,HttpServletRequest request)	把 HTTP 请求中的参数值组装到指定的 JavaBean 中，请求的参数名和 JavaBean 的属性名匹配。当 ActionServlet 把用户输入的表单数据组装到 ActionForm 中时，就调用此方法

5.4.2 TagUtils 类

org.apache.struts.taglib.TagUtil 类为 JSP 标签处理类提供了许多实用方法，如果要使用 TagUtils 类，首先应该调用 TagUtils.getInstance()方法，获得 TagUtils 类的实例，getInstance()方法为静态方法。表 5-2 列出了 TagUtils 类的一些常用方法。

表 5-2 TagUtils 类的常用方法

方 法	描 述
getInstance()	返回一个 TagUtils 的实例，该方法为静态的，如果要在程序中获得 TagUtils 的实例，可以调用 TagUtils.getInstance()方法
getActionMessages(PageContext pageContext,String paramName)	调用 pageContext.findAttribute(paramName)方法，从 page、request、session 和 application 范围内检索并返回 ActionMessages 对象，参数 paramName 指定检索 ActionMessages 对象的属性 key
getModuleConfig(PageContext pageContext)	返回 ModuleConfig 对象，如果不存在，就返回 null
lookup(PageContext pageContext, String name,String scope)	返回特定范围内的 JavaBean，参数 scope 指定 JavaBean 的所在范围，name 参数指定 JavaBean 在特定范围内的名字
message(PageContext pageContext, String bundle,String locale,String key)	从指定的 Resource Bundle 中返回一条消息文本，参数 locale 指定 Locale，参数 key 指定消息 key
write(PageContext pageContext,String text)	向网页上输出特定的文本，参数 text 用于指定文本内容

在 Struts 的早期版本中，org.apache.struts.util 包还提供了 ResponseUtil 类，Strut 1.2 将废弃这个类，它的功能被移到了 TagUtils 类中。

5.4.3 ModuleUtils 类

org.apache.struts.taglib.ModuleUtils 类提供了处理子应用模块的实用方法，如果要使用 ModuleUtils 类，首先应该调用 ModuleUtils.getInstance()方法，获得 ModuleUtils 类的实例，getInstance()方法为静态方法。表 5-3 列出了 ModuleUtils 类的一些常用方法。

表 5-3 ModuleUtils 类的常用方法

方 法	描 述
getInstance()	返回一个 ModuleUtils 的实例，该方法为静态的，如果要在程序中获得 ModuleUtils 的实例，可以调用 ModuleUtils.getInstance()方法
getModuleConfig(javax.servlet.http.HttpServletRequest request)	从 request 范围内检索并返回 ModuleConfig 对象
getModuleConfig(java.lang.String prefix, javax.servlet.ServletContext context)	从 application 范围内检索并返回 ModuleConfig 对象，参数 prefix 指定子应用模块名的前缀
getModuleName(javax.servlet.http.HttpServletRequest request, javax.servlet.ServletContext context)	返回请求访问的子应用模块的名字
selectModule(javax.servlet.http.HttpServletRequest request, javax.servlet.ServletContext context)	选择请求访问的子应用模块，把和子应用模块相关的 ModuleConfig 和 MessageResources 对象存储到 request 范围中

5.4.4　Globals 类

org.apache.struts.Globals 类提供一组公共类型的静态常量，被用做在特定范围内存放 JavaBean 的属性 key。例如：

```
public static final String LOCALE_KEY = "org.apache.struts.action.LOCALE";
```

当 Struts 框架在 session 范围内存放 Locale 实例时，用 Globals.LOCALE_KEY 作为属性 key：

```
session.setAttribute(Globals.LOCALE_KEY,locale);
```

表 5-4 列出了 Globals 类中常用的常量。

表 5-4　Globals 类中定义的常量

方　法	描　述
ACTION_SERVLET_KEY	代表在 application 范围内存放 ActionServlet 实例的属性 key
DATA_SOURCE_KEY	代表在 application 范围内存放默认的 DataSource 实例的属性 key
ERROR_KEY	代表在 request 范围内存放 ActionErrors 实例的属性 key
LOCALE_KEY	代表在 session 范围内存放 Locale 实例的属性 key
MAPPING_KEY	代表在 request 范围内存放 ActionMapping 实例的属性 key
MESSAGE_KEY	代表在 request 范围内存放 ActionMessages 实例的属性 key
MESSAGES_KEY	代表在 application 范围内存放各个子应用模块的 MessageResources 实例的属性 key 的前缀
MODULE_KEY	代表在 application 范围内存放各个子应用模块的 ModuleConfig 实例的属性 key 的前缀
REQUEST_PROCESSOR_KEY	代表在 application 范围内存放各个子应用模块的 RequestProcessor 实例的属性 key 的前缀

对于默认的子应用模块，在 application 范围内存放它的 ModuleConfig 实例的属性 key 为 Globals.MODULE_KEY，即 "org.apache.struts.action.MODULE"；对于其他子应用模块，例如 "/moduleB"，在 application 范围内存放它的 ModuleConfig 实例的属性 key 为 "org.apache.struts.action.MODULE/moduleB"。

5.5　小　　结

本章深入介绍了构成 Struts 框架的中央控制器 ActionServlet 和 RequestProcessor 的控制流程。一个 Struts 应用只能有一个 ActionServlet 实例，对于每个子应用模块都有各自的 RequestProcessor 实例。ActionServlet 在初始化阶段负责加载每个子应用模块的 Struts 配置文件，并把所有的配置信息保存到内存中相关配置类的实例中。

在处理 HTTP 请求时，ActionServlet 负责选择合适的 RequestProcessor 实例，然后调用 RequestProcessor 实例的 process()方法来处理请求，ActionServlet 把包含所有配置信息的 ModuleConfig 对象传给 RequestProcessor 实例的 process()方法，process()方法将依据 ModuleConfig 对象包含的配置信息来处理具体的请求。

如果想深入了解 Struts 框架的控制流程的各个细节，最有效的办法是阅读 ActionServlet 和 RequestProcessor 类的源程序，本书配套光盘的 software/jakarta-struts-src-20040511.zip 文件包含了最新版本的 Struts 软件的所有源代码。如果彻底读懂了这两个类的源程序，看似复杂的 Struts 框架的结构就会变得一目了然。不过，对于初学者，一开始就从阅读这两个类的源程序下手来了解 Struts 框架，难度比较大，更有效的学习方法是从本书第 2 章的简单的 Struts 应用例子下手，循序渐进地学习 Struts 框架。

第 6 章　Struts 模型组件

本章详细介绍了构成 Struts 应用的模型组件。模型代表应用的业务数据和逻辑。坦率地说，Struts 框架并没有为设计和创建模型组件提供现成的框架。不过，Struts 允许使用其他模型框架来处理应用的业务领域，如 EJB（Enterprise JavaBean）和 JDO（Java Data Object），以及常规的 JavaBean 和 ORM（Object-Relation Mapping）。本章提供了模型的一种实现方案。在本书的第 17 章（Struts 与 EJB 组件）和第 18 章（Struts 与 SOAP Web 服务），将分别提供用 EJB 和 Web 服务来实现模型的方案，并且探讨模型实现方式的改变对 Struts 框架有何影响。

本章以实现 netstore 应用的模型为例，来讲解模型组件在 Struts 框架中承担的角色和功能。本章还介绍了如何把持久化框架集成到 Struts 应用中。

6.1　模型在 MVC 中的地位

模型是应用中最重要的一部分，它包含了业务实体和业务规则，负责访问和更新持久化数据。应该把所有的模型组件放在系统中的同一个位置，这有利于维护数据的完整性，减少数据冗余，提高可重用性。

模型应该和视图以及控制器之间保持独立。在分层的框架结构中，位于上层的视图和控制器依赖于下层模型的实现，而下层模型不应该依赖于上层的视图和控制器的实现。Struts 应用的各个层次之间的依赖关系，如图 6-1 所示。

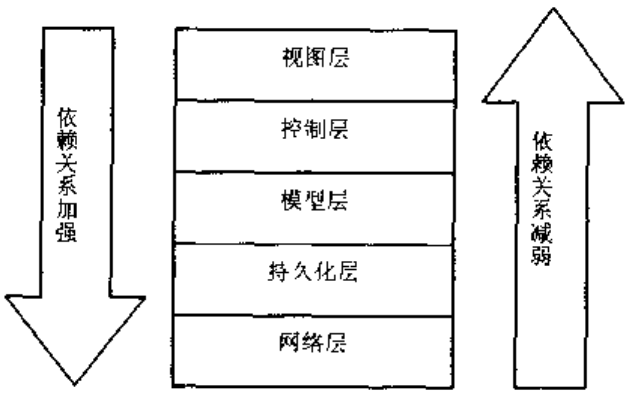

图 6-1　Struts 应用的各个层次之间的依赖关系

如果在模型组件中通过 Java 的 import 语句引入了视图和控制器组件，这就违反了以上原则。下层组件访问上层组件会使应用的维护、重用和扩展变得困难。

6.2 模型的概念和类型

在科学和工程技术领域，模型是一个很有用途的概念，它可以用来模拟一个真实的系统。例如，在建筑领域，在设计建筑物时，会先创建按一定比例缩小的建筑模型；在天文领域，可以用计算机仿真程序来为天体的运行建立模型；在数学领域，可以用一组数学方程式来为某个经济系统建立模型。建立模型最主要的目的是帮助理解、描述或模拟真实世界中目标系统的运转机制。

在软件开发领域，模型用来表示真实世界的实体。在软件开发的不同阶段，需要为目标系统创建不同类型的模型。在分析阶段，需要创建概念模型。在设计阶段，需要创建设计模型。可以采用面向对象建模语言 UML 来描述模型。

6.2.1 概念模型

在建立模型之前，首先要对问题域进行详细的分析，确定用例，接下来就可以根据用例来创建概念模型。概念模型用来模拟问题域中的真实实体。概念模型描述了每个实体的概念和属性，以及实体之间的关系。但在这个阶段并不描述实体的行为。

创建概念模型的目的是帮助更好地理解问题域，识别系统中的实体，这些实体在设计阶段很有可能变为类。如图 6-2 所示，它描述了 netstore 应用的概念模型。注意在这个图中只定义了实体的属性以及实体的关系，而没有定义实体的方法。

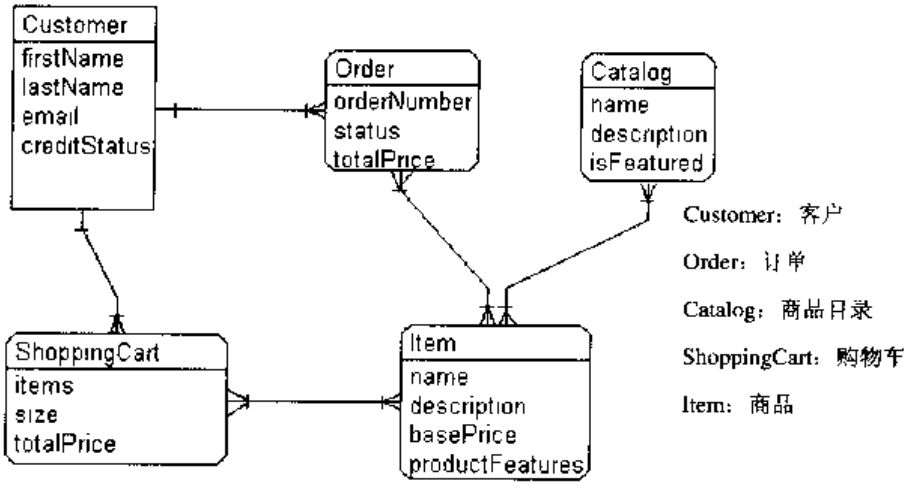

图 6-2 netstore 应用的概念模型

概念模型清楚地显示了问题域中的实体。不管是技术人员还是非技术人员都能看得懂概念模型，他们可以很容易地提出模型中存在的问题，帮助系统分析人员及早对模型进行修改。在软件设计与开发周期中，模型的变更需求提出得越晚，所耗费的开发成本就越大。

从图 6-2 中可以看出 netstore 应用中的实体之间存在以下关系：

- Customer 和 Order 实体：一对多。

- Customer 和 ShoppingCart 实体：一对多。
- ShoppingCart 和 Item 实体：多对多。
- Catalog 和 Item 实体：多对多。
- Order 和 Item 实体：多对多。

商品（Item）与商品目录（Catalog）之间存在多对多的关系，这是因为一个商品目录包含多个商品，而一个商品也可以属于多个商品目录。例如，电动剃须刀既可以被划分为电器类商品，也可以被划分为男士用品。

6.2.2　设计模型

概念模型是在软件分析阶段创建的，它帮助开发人员对应用的需求获得清晰精确的理解。在软件设计阶段，需要在概念模型的基础上创建设计模型。可以用 UML 类框图、活动图以及状态图来描述设计模型。图 6-3 采用类框图来描述了 netstore 应用的设计模型。

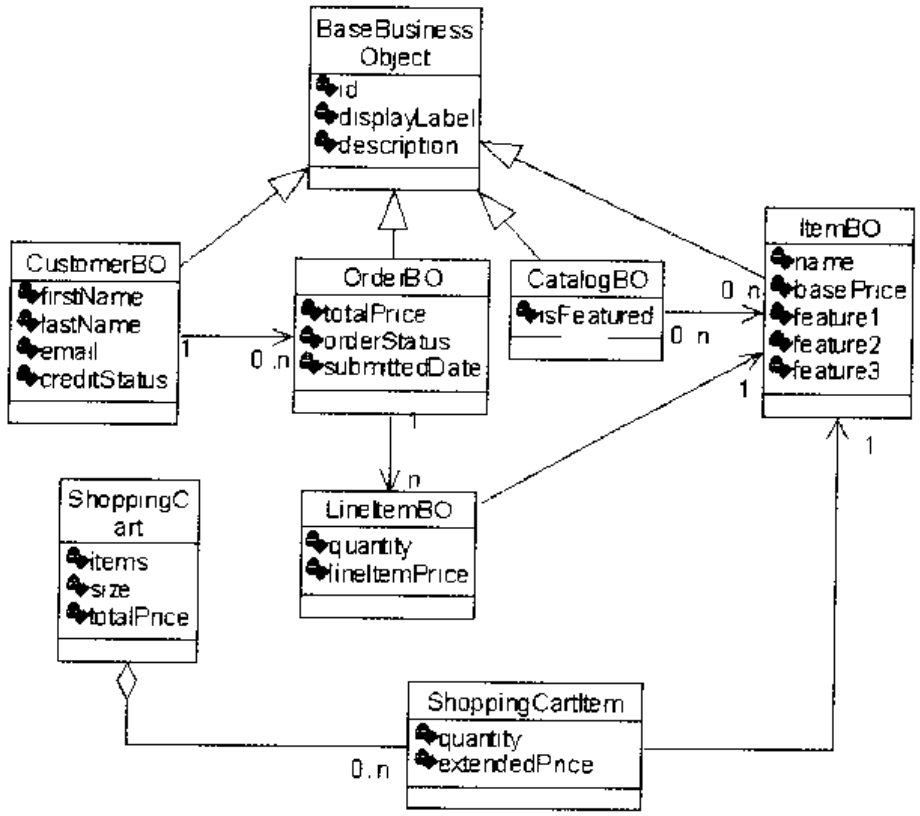

图 6-3　netstore 应用的业务对象类框图

图 6-3 显示了 netstore 应用中所有的业务对象，包括对象的属性和对象之间的关系。在设计模型中，还应该定义对象的方法。为了简化起见，本图省略了显示对象的方法。

根据 UML 语言，类之间存在四种关系。

1．**关联**（Association）

关联指的是类之间的引用关系。例如订单对象 OrderBO 需要引用 CustomerBO 对象，来表明这个订单是由哪个客户发出的。如果类 A 引用类 B，那么被应用的类 B 将被定义为

类 A 的属性。例如，在 OrderBO 中定义了 CustomerBO 属性：

```
public class OrderBO extends BaseBusinessObject {
  // A list of line items for the order
  private List lineItems;
  // The customer who placed the order
  private CustomerBO customer;
  // The current price of the order
  private double totalPrice;
  // The id of the customer
  private Integer customerId;
  // Whether the order is inprocess, shipped, canceled, etc...
  private String orderStatus;
  // The date and time that the order was received
  private Timestamp submittedDate;
   .....
}
```

2. **依赖**（Dependency）

依赖指的是类之间的访问关系。如果类 A 访问类 B 的属性或方法，那么可以说类 A 依赖类 B。与关联关系不同的是，无需把类 B 定义为类 A 的属性。

3. **累积**（Aggregation）

累积指的是整体与个体之间的关系，可以把累积看做一种强关联关系。例如购物车 ShoppingCart 和购物条目 ShoppingCartItem 之间就是累积关系。一个购物车包含多个购物条目。如果类 A 和类 B 之间存在累积关系，那么在类 A 中会定义一个集合属性，来存放类 B 对象。例如，在 ShoppingCart 中定义了 List 类型的属性，来存放所有的 ShoppingCartItem 对象。ShoppingCart 中还定义了 addItem()和 removeItem()等方法，用来向 List 中加入或删除 ShoppingCartItem 对象。以下是 ShoppingCart 类的部分代码：

```
public class ShoppingCart {
  private List items = null;

  public void addItem(ShoppingCartItem newItem) {

    // Check to see if this item is already present, if so, inc the qty
    int size = getSize();
    ShoppingCartItem cartItem = findItem(newItem.getId());
    if(cartItem != null) {
      cartItem.setQuantity(cartItem.getQuantity() + newItem.getQuantity());
    }
    else {

      // Must have been a different item, so add it to the cart
      items.add(newItem);
```

```
    }
  }

  public void removeItem(String itemId) {
    int size = getSize();
    ShoppingCartItem item = findItem(itemId);
    if(item != null) {
      items.remove(item);
    }
  }
    .....
}
```

4．一般化（Generalization）

一般化指的是类之间的继承关系。例如 CustomerBO、OrderBO、CatalogBO、ItemBO 和 LineItemBO 都继承 BaseBusinessObject 类。

如何为应用创建合理的面向对象的设计模型不在本书的讨论范围之内。读者可以浏览 http://www.uml.org/，来深入学习 UML 语言，并掌握利用 UML 来建模的技术。

业务对象的类型和结构，以及业务规则显然取决于问题域。每个应用都有自身的特点，即使在同一个应用中，需求也会经常发生变更。在分析和设计阶段，仔细分析问题域，构建出富有弹性、可扩展的模型，可以使应用更好地适应各种不可预料的需求变化。

6.3　业务对象（BO）

业务对象，即 Business Object（BO），是对真实世界的实体的软件抽象。它可以代表业务领域中的人、地点、事物或概念。例如，netstore 应用中的订单、客户、购物车都是业务对象。

业务对象包括状态和行为。例如订单 OrderBO，表示客户的订单信息，包括价格、购买商品、数量等。此外，OrderBO 中还应该包含客户信息，表明是由哪个客户发出的订单。判断一个类是否可以成为业务对象的一个重要标准，是看这个类是否同时拥有状态和行为。

6.3.1　业务对象的特征和类型

如果一个类可以作为业务对象，它应具有以下特征：

- 包含状态和行为
- 代表业务领域的人、地点、事物或概念
- 可以重用

业务对象可分为三种类型：

- 实体业务对象

- 过程业务对象
- 事件业务对象

实体业务对象要算是最为人们所熟悉的。实体对象可以代表人、地点、事物或概念。通常，可以把业务领域中的名词，例如客户、订单、商品等作为实体业务对象。在 J2EE 应用中，这些名词可以作为实体 Bean。对于更普通的 Web 应用，这些名词可以作为包含状态和行为的 JavaBean。

过程业务对象代表应用中的业务过程或流程，它们通常依赖于实体业务对象。可以把业务领域中的动词，例如客户发出订单、登入应用等作为过程业务对象。在 J2EE 应用中，它们通常作为会话 Bean 或者消息驱动 Bean。在非 J2EE 应用中，它们可作为常规的 JavaBean，具有管理和控制应用的行为。过程业务对象也可以拥有状态，例如在 J2EE 应用中，会话 Bean 可分为有状态和无状态两种类型。

事件业务对象代表应用中的一些事件（如异常、警告或超时）。这些事件通常由系统中的某种行为触发。例如，在 Java Swing 应用中，当客户按下一个按钮，就会有一个事件业务对象产生，以便通知框架调用相关的事件处理器来处理事件。

6.3.2 业务对象的重要性

在应用中使用业务对象有许多好处，最重要的一点就是业务对象提供了通用的术语和概念，不管是技术人员还是非技术人员都可以共享并理解它们。它们可以直观地代表真实世界中的概念，开发小组的所有成员都能理解它们。如果针对同一个业务领域需要开发出多个应用，那么这些应用可以共享这些业务对象。业务对象的可重用特性可以提高应用开发速度，减少冗余。

此外，业务对象可以隐藏实现细节，对外只暴露接口。例如，如果业务对象的某个方法需要传入 java.util.ArrayList 类型的参数，那么应该把参数定义为 java.util.List 接口类型。这样，假定这个方法的实现发生改变，用 LinkedList 取代 ArrayList 来实现原来的功能，这种改变不会对方法调用者造成任何影响。

在充分了解到业务对象在应用中的重要性后，接下来需要关心的问题是，这些业务对象的状态从何而来，当应用终止运行时，这些状态被存放到什么地方。这就涉及到了对象的持久化问题，我们将在 6.4 节中来讨论。

6.4 业务对象的持久化

通常，持久化意味着通过手工或其他方式输入到应用中的数据，能够在应用结束运行后依然存在。即使应用运行结束或者计算机关闭后，这些信息依然存在。不管是大、中或小型的应用，都需要数据的持久化。

6.4.1　对业务对象进行持久化的作用

当应用中的业务对象在内存中创建后，它们不可能永远存在。最后，它们要么从内存中清除，要么被持久化到数据存储库中。内存无法永久保存数据，因此必须对业务对象进行持久化。否则，如果对象没有被持久化，用户在应用运行时发出的订单信息将在应用结束运行后随之消失。netstore 应用中的订单、客户和商品信息都应该被持久化。一旦数据被持久化，它们可以在应用运行时被重新读入到内存中，并重新构造出业务对象。业务对象的持久化过程如图 6-4 所示。

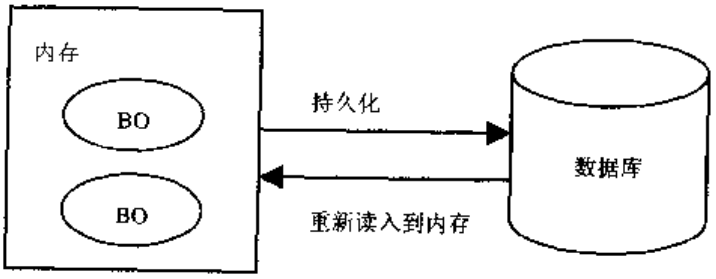

图 6-4　业务对象的持久化

关系型数据库被广泛用来存储数据。关系型数据库中存放的是关系型数据，它是非面向对象的。把业务对象映射到非面向对象的数据库中，存在着阻抗不匹配（impedance mismatch），因为对象由状态和行为组成，而关系型数据库则由表组成，对象之间的各种关系和关系型数据库中表之间的关系并不一一对应。例如对象之间的继承关系就不能直接映射到关系型数据库中。

6.4.2　数据访问对象（DAO）设计模式

面向对象的开发方法是当今的主流，但是同时不得不使用关系型数据库，在企业级应用开发的环境中，对象-关系的映射（Object-Relation Mapping，简称 ORM）是一种耗时的工作。围绕对象-关系的映射和持久化数据的访问，在软件领域中发展起来了一种数据访问对象（Data Access Object，简称 DAO）设计模式。

DAO 模式提供了访问关系型数据库系统所需的所有操作的接口，其中包括创建数据库、定义表、字段和索引，建立表间的关系，更新和查询数据库等。DAO 模式将底层数据访问操作与高层业务逻辑分离开，对上层提供面向对象的数据访问接口。在 DAO 的实现中，可以采用 XML 语言来配置对象和关系型数据之间的映射。图 6-5 显示了 DAO 模式在系统中所处的位置。

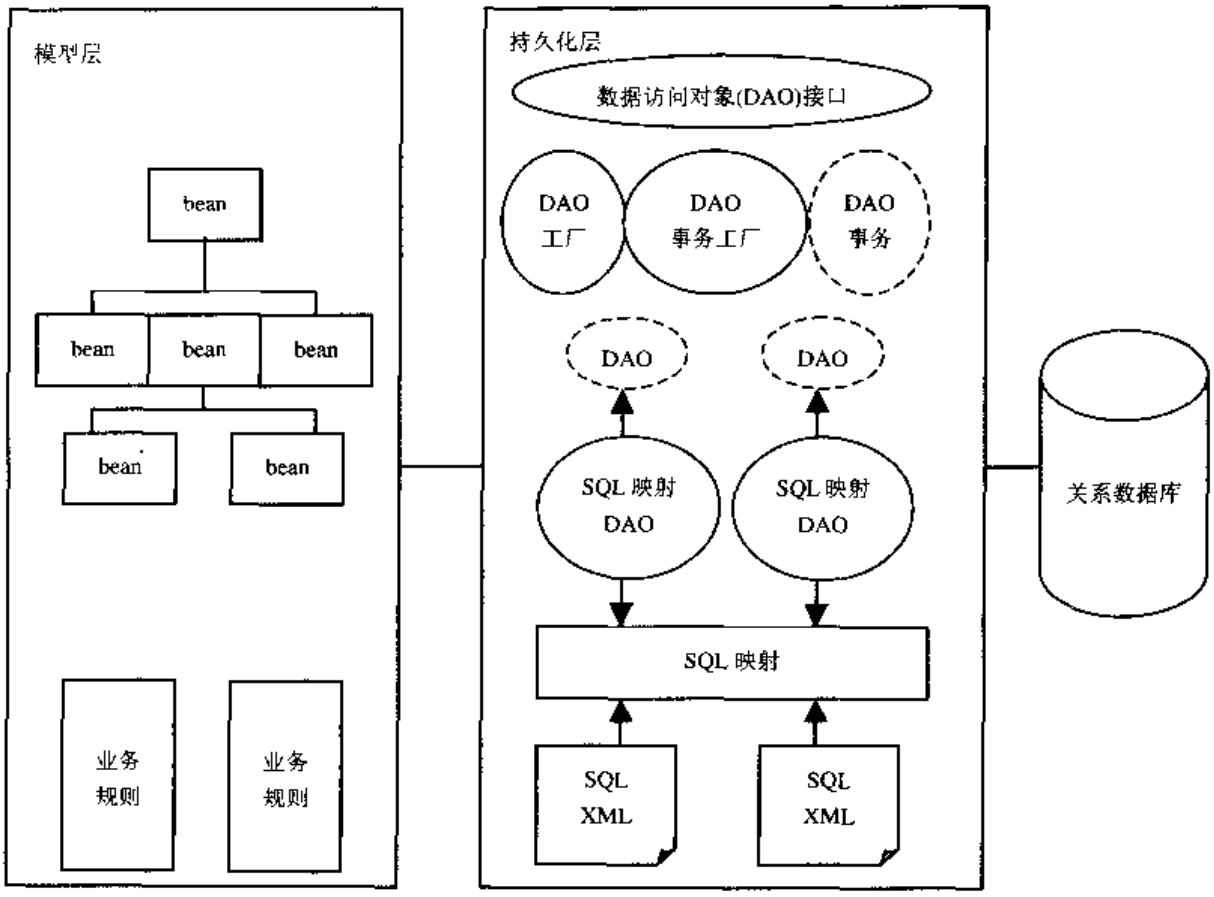

图 6-5 DAO 模式在系统中所处的位置

对于 Java 应用，可以直接通过 JDBC 编程来访问数据库。JDBC 可以说是访问持久数据层最原始、最直接的方法，本书第 3 章介绍的 addressbook 应用就是直接通过 JDBC 编程来访问数据库的。在企业级应用开发中，可以通过 JDBC 编程，来开发自己的 DAO API，把数据库访问操作封装起来，供业务层统一调用。

如果数据模型非常复杂，那么直接通过 JDBC 编程来实现持久化框架需要有专业的知识。对于企业应用的开发人员，花费大量时间从头开发自己的持久化框架不是很可行。通常，可以直接采用第三方提供的持久化框架，如 ORM 软件产品。许多 ORM 框架都采用 DAO 设计模式来实现，为模型层提供了访问关系型数据库的 API。6.4.3 节将介绍几种流行的 ORM 软件产品。

本章提及了好几个概念：持久化框架、ORM 框架和 DAO 设计模式。它们的关系是：ORM 框架是一种持久化框架，DAO 是用于实现持久化框架的一种设计模式。

6.4.3 常用的 ORM 软件

有许多 ORM 软件可供选择。有些是商业化的，有些是免费的。表 6-1 列出了一些 ORM 软件及它们的 URL 地址。

表 6-1　ORM 软件

ORM 软件	URL
TopLink	http://otn.oracle.com/products/ias/toplink/content.html
Torque	http://jakarta.apache.org/turbine/torque/index.html
ObjectRelationalBridge	http://db.apache.org/ojb/
FrontierSuite	http://www.objectfrontier.com
Castor	http://castor.exolab.org
FreeFORM	http://www.chimu.com/projects/form/
Expresso	http://www.jcorporate.com
JRelationalFramework	http://jrf.sourceforge.net
VBSF	http://www.objectmatter.com
Jgrinder	http://sourceforge.net/projects/jgrinder/
Hibernate	http://www.hibernate.org/

不管是使用商业化产品，还是非商业化产品，都应该确保选用的 ORM 框架没有“渗透”到应用中，应用的上层组件应该和 ORM 框架保持独立。有些 ORM 框架要求在业务对象中引入它们的类和接口，这会带来一个问题，如果日后想改用其他的 ORM 框架，就必须修改业务对象。在本章的 6.5.5 节中，将介绍联合使用业务代理模式和 DAO 模式来避免 ORM 框架渗透到应用的上层组件中。

6.5　创建 netstore 应用的模型

在讨论了构建 Struts 模型的基本原理后，现在我们将把理论运用到开发 netstore 应用的实践当中。netstore 应用是一个简化的电子商务应用例子，尽管它还不能代表具有实际商业利用价值的完整的电子商务应用，但它所提供的对象模型囊括了创建模型的各种知识。

6.5.1　为 netstore 应用创建模型的步骤

netstore 应用的持久化状态保存在关系型数据库中。人们常常把 ERP（Enterprise Resource Planning，企业资源计划）系统和关系型数据库联合使用。但是多数电子商务应用为了提高性能和简化开发，往往直接把数据库用于前端。

本书选用 MySQL 数据库来存放数据。只要熟悉 SQL DDL（Data Definition Language），就可以方便地把本书中举的例子移植到其他类型的数据库中。

为 netstore 应用创建模型包括以下步骤：

（1）创建业务对象。

（2）创建数据库。

（3）把业务对象映射到数据库。

（4）测试业务对象是否可以被持久化到数据库。

值得注意的是，以上所有步骤都没有在 Struts 框架中提及。在完成这些步骤时，不必

考虑使用模型的客户程序。netstore Web 应用是业务对象的一潜在客户程序。如果设计和编码合理，业务对象可以支持不同类型的客户程序。业务对象用于查询和持久化数据，应该避免在客户层重复实现业务对象中的逻辑。

为了帮助把业务对象和客户层分离，在 netstore 应用中还使用了业务代理（Business Delegate）设计模式。业务代理充当客户端业务逻辑的抽象，它隐藏了业务服务的实现，可以避免业务对象与客户层对业务功能的重复实现。业务代理模式的结构如图 6-6 所示。

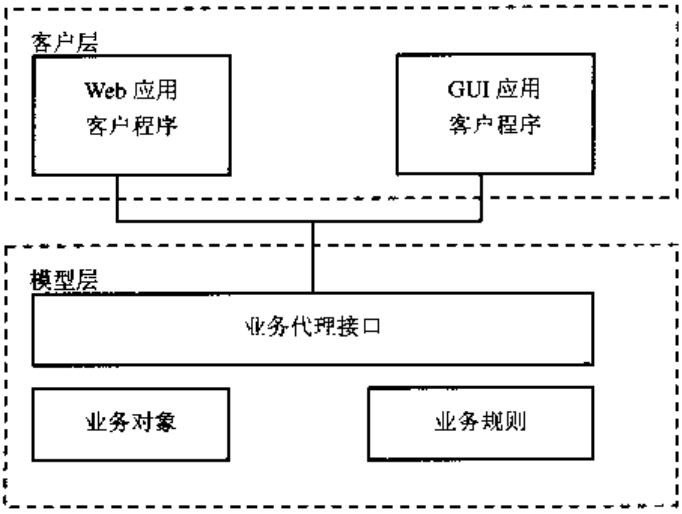

图 6-6 业务代理模式的结构

6.5.2 创建 netstore 应用的业务对象

业务对象包含数据和行为，它们对应着数据库中一条或多条记录。本章使用 JavaBean 来实现它们。由于所有的业务对象都共享一些公共的属性，因此可以创建一个抽象父类 BaseBusinessObject 类来存放这些公共属性。例程 6-1 为 BaseBusinessObject 类的源程序。

例程 6-1 BaseBusinessObject 类

```
package netstore.businessobjects;

/**
 * An abstract super class that many business objects will extend.
 */
abstract public class BaseBusinessObject implements java.io.Serializable {
  private Integer id;
  private String displayLabel;
  private String description;

  public Integer getId() {
    return id;
  }

  public void setId(Integer id) {
```

```
    this.id = id;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public String getDescription() {
    return description;
  }

  public void setDisplayLabel(String label) {
    this.displayLabel = label;
  }

  public String getDisplayLabel() {
    return displayLabel;
  }
}
```

BaseBusinessObject 类可以避免每个业务对象都重复定义这些公共属性。如果需要，还可以在这个类中定义公共的业务逻辑。

例程 6-2 为订单类 OrderBO 的源程序。OrderBO 是一个普通的 JavaBean 类，它包含一些属性和 get/set 方法，此外，它还包括一个业务方法 recalculatePrice()。

例程 6-2　OrderBO 类

```
package netstore.businessobjects;

import java.util.Iterator;
import java.util.List;
import java.util.LinkedList;
import java.math.BigDecimal;
import java.sql.Timestamp;
/**
 * The Order Business Object, which represents a purchase order that a customer
 * has or is about to place.
 */
public class OrderBO extends BaseBusinessObject {
  // A list of line items for the order
  private List lineItems;
  // The customer who placed the order
  private CustomerBO customer;
  // The current price of the order
  private double totalPrice;
  // The id of the customer
  private Integer customerId;
```

```
// Whether the order is inprocess, shipped, cancéled, etc...
private String orderStatus;
// The date and time that the order was received
private Timestamp submittedDate;

/**
 * Default NoArg Constructor
 */
public OrderBO() {
   super();
   // Initialize the line items as a linked list to keep them in order
   lineItems = new LinkedList();
}
/**
 * Additional constructor that takes the neccessary arguments to initialize
 */
public OrderBO( Integer id, Integer custId, String orderStatus,
                    Timestamp submittedDate, double price ){
   this.setId(id);
   this.setCustomerId(custId);
   this.setOrderStatus(orderStatus);
   this.setSubmittedDate(submittedDate);
   this.setTotalPrice(price);
}

public void setCustomer( CustomerBO owner ){
   customer = owner;
}

public CustomerBO getCustomer(){
   return customer;
}

public double getTotalPrice(){
   return this.totalPrice;
}

private void setTotalPrice( double price ){
   this.totalPrice = price;
}

public void setLineItems( List lineItems ){
   this.lineItems = lineItems;
}

public List getLineItems(){
```

```
    return lineItems;
  }

  public void addLineItem( LineItemBO lineItem ){
    lineItems.add( lineItem );
  }

  public void removeLineItem( LineItemBO lineItem ){
    lineItems.remove( lineItem );
  }

  public void setCustomerId(Integer customerId) {
    this.customerId = customerId;
  }

  public Integer getCustomerId() {
    return customerId;
  }

  public void setOrderStatus(String orderStatus) {
    this.orderStatus = orderStatus;
  }

  public String getOrderStatus() {
    return orderStatus;
  }

  public void setSubmittedDate(Timestamp submittedDate) {
    this.submittedDate = submittedDate;
  }

  public Timestamp getSubmittedDate() {
    return submittedDate;
  }

  private void recalculatePrice(){
    double totalPrice = 0.0;

    if ( getLineItems() != null ){
      Iterator iter = getLineItems().iterator();
      while( iter.hasNext() ){
        // Get the price for the next line item and make sure it's not null
        Double lineItemPrice = ((LineItemBO)iter.next()).getUnitPrice();
        // Check for an invalid lineItem. If found, return null right here
        if (lineItemPrice != null){
          totalPrice += lineItemPrice.doubleValue();
```

```
                }
            }
            // Set the price for the order from the calcualted value
            setTotalPrice( totalPrice );
        }
    }
}
```

由于篇幅关系，不再列举其他业务对象的源程序，它们的实现和 OrderBO 类很相似。

在设计业务对象时，不必考虑如何把它们映射到数据库，也不必考虑如何把面向对象的概念，如继承、多态映射到数据库中，这是下一步再做的事。

6.5.3 创建 netstore 应用的数据库

一旦创建了业务对象，就可以创建数据库模型和数据库 Schema。如何创建数据库 Schema 不在本书的讨论范围之内。如果应用的规模比较小，几乎任何人都能够创建数据库 Schema。如果应用的规模比较大，包含很多表，表之间的关系复杂，最好是由数据库专家来创建数据库 Schema。netstore 应用的 schema 不是很复杂。netstore 应用的数据模型如图 6-7 所示。

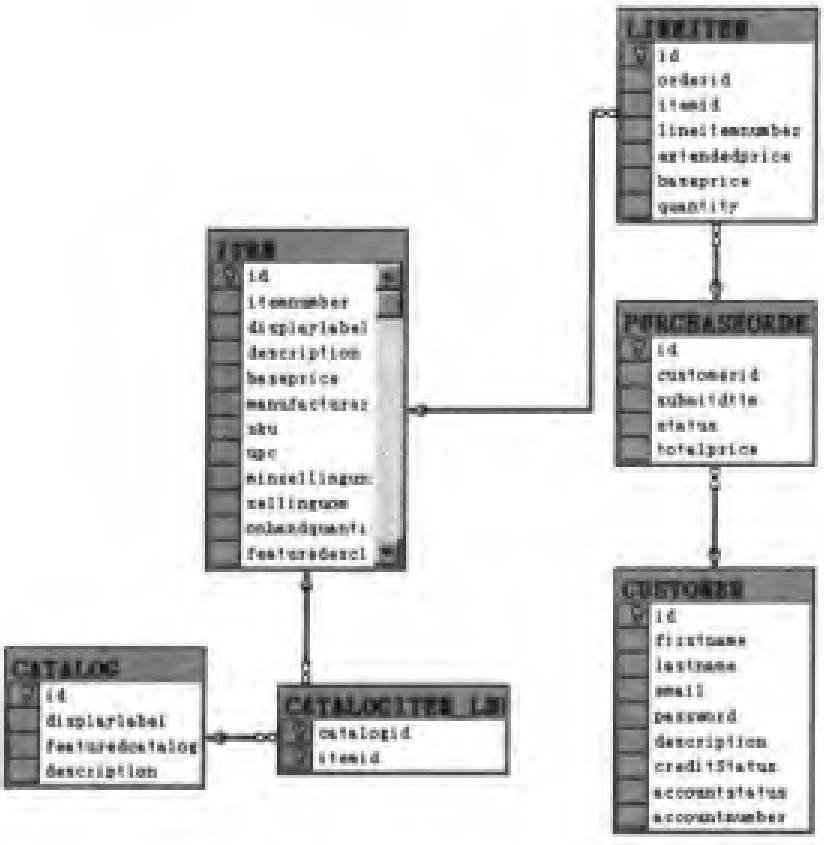

图 6-7 netstore 应用的数据模型

图 6-7 中的每张表（CATALOGITEM_LNK 表除外）都有一个 ID 字段，它作为表的主键。为了在系统中能够找到所需的对象，需要为每个对象分配惟一的标识符。在关系数据库中，这种标识符被称为关键字，而在面向对象术语中，则叫做 OID（Object Identifier），即对象标识符。通常 OID 都使用整数表示。

OID 应当不具有任何业务含义，因为任何有业务含义的列都有改变的可能性，关系数据库学的最重要的一个理论就是：不要给关键字赋予任何业务意义。假如关键字具有了业务意义，如果用户决定改变业务含义，也许他们想要增加几个数字或把数字变为字母，那么就必须修改相关的关键字。一个表中的主关键字有可能被其他表作为外键。就算是一个简单的改变，譬如在客户号码上增加一个数字，也可能会造成极大的维护上的开销。在关系数据库中，这种 OID 策略被称为代理关键字（Surrogate Key）。

可以采用 SQL DDL 语言来创建数据库的 Schema。SQL DDL 用于生成数据库中的物理实体。例程 6-3 为 netstore 应用的数据库 Schema。

例程 6-3　netstore 应用的数据库 Schema

```
# The SQL DDL for the Netstore Application

# Execute the next line if you need to clear the netstore database
DROP DATABASE netstore;

# Creates the initial database
CREATE DATABASE netstorc;

# Make sure you are creating the tables in the netstore tablespace
use netstore;

CREATE TABLE CATALOG(
   id int NOT NULL,
   displaylabel varchar(50) NOT NULL,
   featuredcatalog char(1) NULL,
   description varchar(255) NULL
);

ALTER TABLE CATALOG ADD
   CONSTRAINT PK_CATALOG PRIMARY KEY(id);

CREATE TABLE CUSTOMER (
   id int NOT NULL,
   firstname varchar(50) NOT NULL,
   lastname varchar(50) NOT NULL,
   email varchar(50) NOT NULL,
   password varchar(15) NOT NULL,
   description varchar(255) NULL,
   creditStatus char(1) NULL,
   accountstatus char(1) NULL,
```

```
    accountnumber varchar(15) NOT NULL
);

ALTER TABLE CUSTOMER ADD
    CONSTRAINT PK_CUSTOMER PRIMARY KEY(id);

CREATE TABLE ITEM (
    id int NOT NULL,
    itemnumber varchar (255) NOT NULL,
    displaylabel varchar(50) NOT NULL,
    description varchar (255) NULL,
    baseprice decimal(9,2) NOT NULL,
    manufacturer varchar (255) NOT NULL,
    sku varchar (255) NOT NULL,
    upc varchar (255) NOT NULL,
    minsellingunits int NOT NULL,
    sellinguom varchar (255) NOT NULL,
    onhandquantity int NOT NULL,
    featuredesc1 varchar (255) NULL,
    featuredesc2 varchar (255) NULL,
    featuredesc3 varchar (255) NULL,
    smallimageurl varchar (255) NULL,
    largeimageurl varchar (255) NULL
);

ALTER TABLE ITEM ADD
    CONSTRAINT PK_ITEM PRIMARY KEY(id);


CREATE TABLE CATALOGITEM_LNK(
    catalogid int NOT NULL,
    itemid int NOT NULL
);

ALTER TABLE CATALOGITEM_LNK ADD
    CONSTRAINT PK_CATALOGITEM_LNK PRIMARY KEY(catalogid, itemid);

ALTER TABLE CATALOGITEM_LNK ADD
    CONSTRAINT FK_CATALOGITEM_LNK_CATALOG FOREIGN KEY
    (catalogid) REFERENCES CATALOG(id);

ALTER TABLE CATALOGITEM_LNK ADD
    CONSTRAINT FK_CATALOGITEM_LNK_ITEM FOREIGN KEY
    (itemid) REFERENCES ITEM(id);

CREATE TABLE PURCHASEORDER (
```

```
    id int NOT NULL,
    customerid int NOT NULL,
    submitdttm timestamp NOT NULL,
    status varchar (15) NOT NULL,
    totalprice decimal(9,2) NOT NULL,
);

ALTER TABLE PURCHASEORDER ADD
    CONSTRAINT PK_PURCHASEORDER PRIMARY KEY(id);

ALTER TABLE PURCHASEORDER ADD
    CONSTRAINT FK_PURCHASEORDER_CUSTOMER FOREIGN KEY
    (customerid) REFERENCES CUSTOMER(id);

CREATE TABLE LINEITEM (
    id int NOT NULL,
    orderid int NOT NULL,
    itemid int NOT NULL,
    lineitemnumber int NULL,
    extendedprice decimal(9, 2) NOT NULL,
    baseprice decimal(9, 2) NOT NULL,
    quantity int NOT NULL
);

ALTER TABLE LINEITEM ADD
    CONSTRAINT PK_LINEITEM PRIMARY KEY(id);

ALTER TABLE LINEITEM ADD
    CONSTRAINT FK_LINEITEM_PURCHASEORDER FOREIGN KEY
    (orderid) REFERENCES PURCHASEORDER(id);

ALTER TABLE LINEITEM ADD
    CONSTRAINT FK_LINEITEM_ITEM FOREIGN KEY
    (itemid) REFERENCES ITEM(id);
```

例程 6-3 中的数据库 Schema 在 MySQL 服务器上运行通过。如果要在其他数据库服务器上运行，可能需要对 SQL 语句做适当的调整。

一旦执行了以上的 DDL，就可以向表中插入数据。

6.5.4　netstore 应用的 ORM 框架

在决定选用某一种 ORM 软件时，有以下几个因素值得考虑：

- 使用某种 ORM 软件的成本

- 对上层应用的渗透程度
- 是否提供详细的使用说明文档

ORM 软件的价格通常是个重要因素。对于 netstore 应用，我们决定采用开放源代码的 ObjectRelationalBridge（OJB），它由 Apache 开放源代码软件组织提供。OJB 软件的下载网址为 http://db.apache.org/ojb/。

本应用选用 OJB，这并不意味着它是适合所有应用的最佳方案。用户应该根据实际情况来选用合适的 ORM 软件。

OJB 提供了非常详细的使用文档。在 Web 应用的持久化层中使用 OJB 框架包括以下步骤。

（1）在 repository_user.xml 文件中配置业务对象和数据库中表的映射关系，repository_user.xml 文件的存放位置为 WEB-INF/classes 目录。

（2）在 repository.xml 文件中配置和数据库的连接，repository.xml 文件的存放位置为 WEB-INF/classes 目录。

（3）在数据库中创建 OJB 专用的表。

（4）把和 OJB 相关的 JAR 文件以及数据库的 JDBC 驱动程序 JAR 文件拷贝到 WEB-INF/lib 目录中。

（5）在应用程序中通过 OJB API 来访问数据库。

1. 配置 OJB 的 repository_user.xml 文件

OJB 采用单个的 XML 文件 repository_user.xml 把所有的业务对象映射到数据库中。这个文件在运行时由映射框架来解析它。映射框架根据这个文件来执行相应的 SQL 语句。例程 6-4 为 netstore 应用的映射文件的部分代码，它用于把 CustomerBO 类映射到数据库的 CUSTOMER 表。

例程 6-4　CustomerBO 类的 XML 映射

```
<class-descriptor
  class="netstore.businessobjects.CustomerBO"
  table="CUSTOMER">

  <field-descriptor
    name="id"
    column="id"
    jdbc-type="INTEGER"
    primarykey="true"
    autoincrement="true" />
```

```
<field-descriptor
   name="firstName"
   column="firstname"
   jdbc-type="VARCHAR"
/>

<field-descriptor
   name="lastName"
   column="lastname"
   jdbc-type="VARCHAR"
/>

<field-descriptor
   name="email"
   column="email"
   jdbc-type="VARCHAR"
/>

<field-descriptor
   name="password"
   column="password"
   jdbc-type="VARCHAR"
 />

<field-descriptor
   name="accountStatus"
   column="accountstatus"
   jdbc-type="CHAR"
/>

<field-descriptor
   name="creditStatus"
   column="creditstatus"
   jdbc-type="CHAR"
/>

<collection-descriptor
 name="submittedOrders"
 element-class-ref="netstore.businessobjects.OrderBO">
 <inverse-foreignkey field-ref="customerId"/>
</collection-descriptor>

</class-descriptor>
```

在例程 6-4 中，<class-descriptor>元素指定类和表的映射，它的 class 属性指定类名，table 属性指定表名。<class-descriptor>元素包含多个<field-descriptor>元素，它指定类的属性和表

的字段的映射。<class-descriptor>元素还包含一个<collection-descriptor>元素，它指定 CustomerBO 和 OrderBO 实体之间一对多的关系。<collection-descriptor>元素的 name 属性为“submittedOrders”，它和 CustomerBO 类中定义的 Vector 类型的 submittedOrders 属性匹配：

```
private Vector submittedOrders = null;
public void setSubmittedOrders( Vector aList ){
    this.submittedOrders = aList;
}

public Vector getSubmittedOrders(){
    return submittedOrders;
}
```

<collection-descriptor>元素、CustomerBO 和 OrderBO 之间的映射关系如图 6-8 所示。

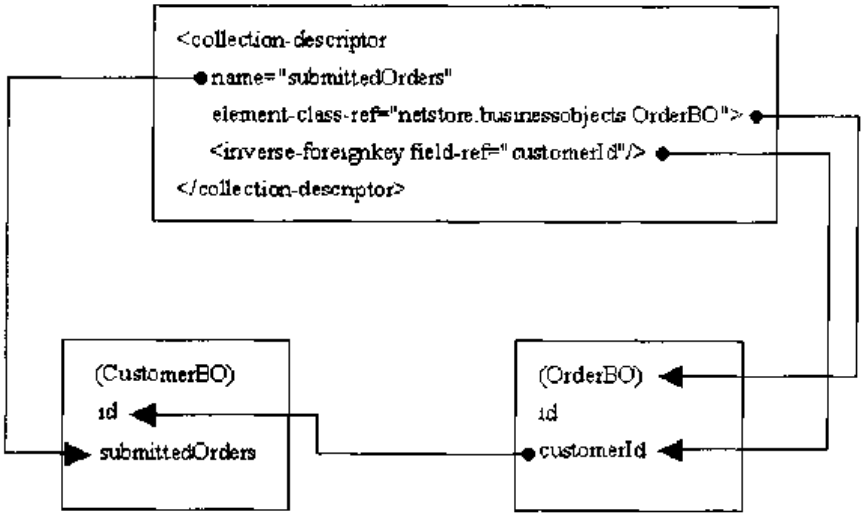

图 6-8 <collection-descriptor>元素、CustomerBO 和 OrderBO 之间的映射关系

例程 6-5 为 OrderBO 类和 PURCHASEORDER 表之间的 XML 映射代码。

例程 6-5 OrderBO 类的 XML 映射

```
<class-descriptor
  class="netstore.businessobjects.OrderBO"
  table="PURCHASEORDER">

  <field-descriptor
    name="id"
    column="id"
    jdbc-type="INTEGER"
    primarykey="true"
    autoincrement="true"
  />

  <field-descriptor
    name="customerId"
```

```
    column="customerid"
    jdbc-type="INTEGER"
  />

  <field-descriptor
    name="orderStatus"
    column="status"
    jdbc-type="VARCHAR"
  />

  <field-descriptor
    name="submittedDate"
    column="submitdttm"
    jdbc-type="TIMESTAMP"
  />

  <field-descriptor
    name="totalPrice"
    column="totalprice"
    jdbc-type="DOUBLE"
  />

  <reference-descriptor
   name="customer"
   class-ref="netstore.businessobjects.CustomerBO">
    <foreignkey field-ref="customerId"/>
  </reference-descriptor>

  <collection-descriptor
   name="lineItems"
   element-class-ref="netstore.businessobjects.LineItemBO">
   <inverse-foreignkey field-ref="orderId"/>
  </collection-descriptor>

</class-descriptor>
```

在例程 6-5 中，<reference-descriptor>元素指定 OrderBO 和 CustomerBO 实体之间的关联关系。<reference-descriptor>元素的 name 属性为“customer”，它和 OrderBO 类中定义的 CustomerBO 类型的 customer 属性匹配：

```
private CustomerBO customer;
public void setCustomer( CustomerBO owner ){
    customer = owner;
}
public CustomerBO getCustomer(){
    return customer;
}
```

<reference-descriptor>元素、OrderBO 和 CustomerBO 之间的映射关系如图 6-9 所示。

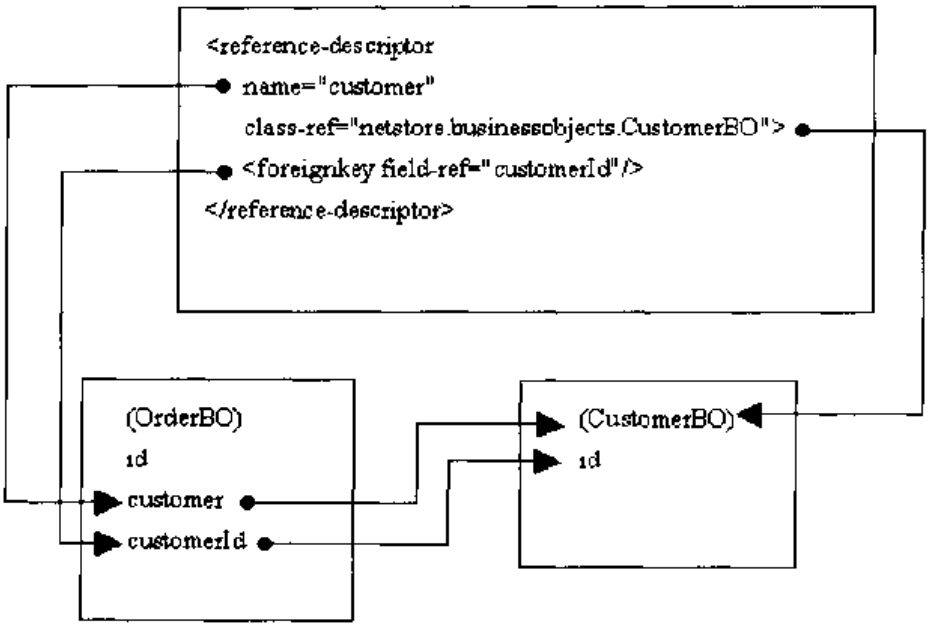

图 6-9 <reference-descriptor>元素、OrderBO 和 CustomerBO 之间的映射关系

业务对象的父类 BaseBusinessObject 并不会映射到数据库的某张表。不过在映射其子类时，会把从父类中继承而来的属性（如 id 属性）和对应表的相关字段映射。尽管多数持久化框架支持继承映射，但本应用并没有采用这种做法，而是把父类属性映射到每个子类的对应表中，这可以减少 SQL 数据库中表的连接，提高数据库性能。

接下来采用同样方式对其余的业务对象进行映射。一旦所有的映射都在 XML 中定义，还需要配置数据库连接信息，详细内容参见下一小节。

2．配置 OJB 的 repository.xml 文件

OJB 框架在 repository.xml 文件中配置数据库连接，配置代码参见例程 6-6。

例程 6-6 包含数据库连接信息的 respository.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a sample metadata repository for the
        Apache ObJectRelationalBridge (OJB) System.
        Use this file as a template for building your own mappings.
-->

<!-- doctype definition
        By default we are using a local DTD that is expected to reside
        in the same directory as this repository.xml file.
        If you intend to validate your repository.xml against
        the public dtd at the Apache site, please replace the string
        "repository.dtd"
        by the public adress
        "http://db.apache.org/ojb/dtds/1.0/repository.dtd".
```

```
        In this case validation will only work if the machine you
        run your application on can connect to the internet!
-->

<!DOCTYPE descriptor-repository PUBLIC
        "-//Apache Software Foundation//DTD OJB Repository//EN"
        "repository.dtd"
[
<!ENTITY internal SYSTEM "repository_internal.xml">
<!-- here the junit include files begin    -->
<!ENTITY user SYSTEM "repository_user.xml">
]>

<descriptor-repository version="1.0" isolation-level="read-uncommitted"
        proxy-prefetching-limit="50">

    <!-- this connection was used as the default one within OJB -->
    <jdbc-connection-descriptor
        jcd-alias="default"
        default-connection="true"
        platform="MYSQL"
        jdbc-level="2.0"
        driver="com.mysql.jdbc.Driver"
        protocol="jdbc"
        subprotocol="mysql"
        dbalias="//localhost:3306/netstore"
        username="root"
        password=""
        eager-release="false"
        batch-mode="false"
        useAutoCommit="1"
        ignoreAutoCommitExceptions="false" >

        <connection-pool
                maxActive="21"
                validationQuery="" />

        <sequence-manager
                className="org.apache.ojb.broker.util.sequence.SequenceManagerHighLowIm
        pl">
                <attribute attribute-name="grabSize" attribute-value="20"/>
                <attribute attribute-name="autoNaming" attribute-value="true"/>
                <attribute attribute-name="globalSequenceId" attribute-value="false"/>
                <attribute attribute-name="globalSequenceStart" attribute-value="10000"/>
        </sequence-manager>
    </jdbc-connection-descriptor>
```

```
    <!-- include ojb internal mappings here -->
    &internal;
    <!-- include user defined mappings here -->
    &user;
</descriptor-repository>
```

应该根据具体的环境来配置以上文件。对于 Web 应用，OJB 的配置文件 repository.xml 和 repository_user.xml 文件都存放在 WEB-INF/classes 目录下。

3. 在数据库中创建 OJB 专用的表

为了保证 OJB 框架能正常运行，还应该在 netstore 数据库中创建 OJB 专用的表，例程 6-7 是创建这些表的 SQL 语句。

例程 6-7　创建 OJB 专用表的 DDL

```
use netstore;

# ------------------------------------------------------------
# OJB_HL_SEQ
# ------------------------------------------------------------
drop table if exists OJB_HL_SEQ;

CREATE TABLE OJB_HL_SEQ
(
        TABLENAME VARCHAR (175) NOT NULL,
        FIELDNAME VARCHAR (70) NOT NULL,
        MAX_KEY INTEGER,
        GRAB_SIZE INTEGER,
        VERSION INTEGER,
    PRIMARY KEY(TABLENAME,FIELDNAME)
);

# ------------------------------------------------------------
# OJB_LOCKENTRY
# ------------------------------------------------------------
drop table if exists OJB_LOCKENTRY;

CREATE TABLE OJB_LOCKENTRY
(
        OID_ VARCHAR (250) NOT NULL,
        TX_ID VARCHAR (50) NOT NULL,
        TIMESTAMP_ TIMESTAMP,
        ISOLATIONLEVEL INTEGER,
        LOCKTYPE INTEGER,
    PRIMARY KEY(OID_,TX_ID)
);
```

```
# ----------------------------------------------------------
# OJB_NRM
# ----------------------------------------------------------
drop table if exists OJB_NRM;

CREATE TABLE OJB_NRM
(
        NAME VARCHAR (250) NOT NULL,
        OID_ LONGBLOB,
    PRIMARY KEY(NAME)
);

# ----------------------------------------------------------
# OJB_DLIST
# ----------------------------------------------------------
drop table if exists OJB_DLIST;

CREATE TABLE OJB_DLIST
(
        ID INTEGER NOT NULL,
        SIZE_ INTEGER,
    PRIMARY KEY(ID)
);

# ----------------------------------------------------------
# OJB_DLIST_ENTRIES
# ----------------------------------------------------------
drop table if exists OJB_DLIST_ENTRIES;

CREATE TABLE OJB_DLIST_ENTRIES
(
        ID INTEGER NOT NULL,
        DLIST_ID INTEGER NOT NULL,
        POSITION_ INTEGER,
        OID_ LONGBLOB,
    PRIMARY KEY(ID)
);

# ----------------------------------------------------------
# OJB_DSET
# ----------------------------------------------------------
drop table if exists OJB_DSET;

CREATE TABLE OJB_DSET
(
```

```
        ID INTEGER NOT NULL,
        SIZE_ INTEGER,
    PRIMARY KEY(ID)
);

# ----------------------------------------------------------------------
# OJB_DSET_ENTRIES
# ----------------------------------------------------------------------
drop table if exists OJB_DSET_ENTRIES;

CREATE TABLE OJB_DSET_ENTRIES
(
        ID INTEGER NOT NULL,
        DLIST_ID INTEGER NOT NULL,
        POSITION_ INTEGER,
        OID_ LONGBLOB,
    PRIMARY KEY(ID)
);

# ----------------------------------------------------------------------
# OJB_DMAP
# ----------------------------------------------------------------------
drop table if exists OJB_DMAP;

CREATE TABLE OJB_DMAP
(
        ID INTEGER NOT NULL,
        SIZE_ INTEGER,
    PRIMARY KEY(ID)
);

# ----------------------------------------------------------------------
# OJB_DMAP_ENTRIES
# ----------------------------------------------------------------------
drop table if exists OJB_DMAP_ENTRIES;

CREATE TABLE OJB_DMAP_ENTRIES
(
        ID INTEGER NOT NULL,
        DMAP_ID INTEGER NOT NULL,
        KEY_OID MEDIUMBLOB,
        VALUE_OID MEDIUMBLOB,
    PRIMARY KEY(ID)
);
```

4．通过 OJB API 访问数据库

OJB 提供了两种不同的 API：PersistenceBroker 和 ODMG。可以参考 OJB 的文档来了解这两种 API 的区别。相比起来 ODMG 的功能更强大些，它支持对象查询语言（Object Query Language，OQL），但使用起来也更为复杂。netstore 应用使用 ODMG API，因为它的功能更多。ODMG 框架在独立模式或客户/服务器模式下都可以运行。在有多个服务器的环境下，客户/服务器模式非常有用。但是在本例中使用的是独立模式，因为在 netstore 应用中不存在多个服务器。在独立模式下，持久化框架和 netstore 应用运行在同一个 Java 虚拟机中。

如何使用 ODMG API 来访问数据库不是本书的重点，读者可以浏览 http://db.apache.org/ojb/，在该网站上提供了详细的帮助文档。以下代码用于示范如何使用 ODMG API 来执行数据库查询。这段代码查询数据库中的 ITEM 表，把查到的记录映射为 ItemBO 对象，然后把这些 ItemBO 对象保存到一个 List 类型的 results 变量里：

```
org.odmg. Implementation odmg = null;
org.odmg. Database db = null;
odmg = OJB.getInstance();
db = odmg.newDatabase();
//open database
db.open("default", Database.OPEN_READ_WRITE);

Transaction tx = odmg.newTransaction();
tx.begin();
List results = null;
try{
        org.odmg. OQLQuery query = odmg.newOQLQuery();
        // Set the OQL select statement
        query.create( "select featuredItems from " + ItemBO.class.getName() );
        results = (List)query.execute();
        tx.commit();
}catch( Exception ex ){
        // Rollback the transaction
        tx.abort();
        ex.printStackTrace();
        throw DatastoreException.datastoreError(ex);
}
```

对于 Web 应用，如果在程序中使用了 OJB API，应该把和 OJB 相关的 JAR 文件以及数据库的 JDBC 驱动程序 JAR 文件拷贝到 WEB-INF/lib 目录中。

6.5.5　联合使用业务代理和 DAO 模式

最后一个问题是创建业务代理接口。业务代理接口直接访问持久化框架，处理实际的业务逻辑。netstore 应用的 Action 类可以使用这个业务代理接口，而不必直接和持久化框架交互。这种做法有助于削弱上层 Web 应用和持久化框架之间的关系，提高持久化框架的相

对独立性，这种设计模式被称为业务代理模式。

此外，还需要采用 DAO 模式来削弱应用的业务逻辑和数据库访问逻辑的关系。当使用持久化框架的时候，DAO 模式可以把业务对象和持久化框架分离，当持久化实现机制发生改变时，这种改变不会对业务对象产生影响。

联合运用 DAO 模式和业务代理模式，可以提高持久化层、模型层和 Web 应用层的相互独立性。业务代理和 DAO 的联合模式如图 6-10 所示。

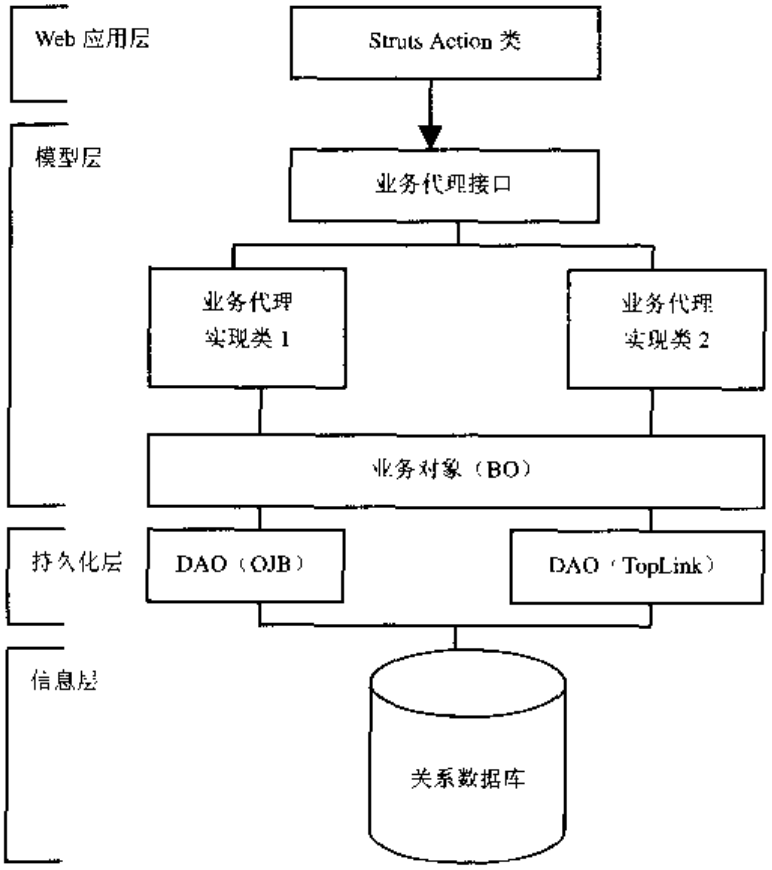

图 6-10 业务代理和 DAO 的联合模式

图 6-10 中的 Struts Action 类访问业务代理接口。在持久化框架层，既可以选用 OJB 框架，也可以选用 Toplink 框架，这不会影响业务对象的实现。对于 netstore 应用，业务代理接口为 INetstoreService，在本书中一共提供了五种实现方式：

- 方式一：持久化数据保存在基于 XML 的文件 database.xml 中，业务代理实现类 NetstoreDebugServiceImpl 从 database.xml 文件中读取业务数据。
- 方式二：业务代理实现类 NetstoreServiceImpl 通过 OJB 框架访问数据库
- 方式三：采用 EJB 组件实现模型。EJB 组件通过 OJB 框架访问数据库，业务代理实现类 NetstoreEJBDelegate 调用 EJB 组件方法来提供服务，参见第 17 章（Struts 与 EJB 组件）。
- 方式四：采用 EJB 组件实现模型，采用 EJBHomeFactory 模式来访问 EJB Home 接口。EJB 组件通过 OJB 框架访问数据库，业务代理实现类 NetstoreEJBFromFactoryDelegate 调用 EJB 组件的方法来提供服务，详细内容参见第 17 章（Struts

与 EJB 组件）。

- 方式五：采用 Web 服务实现模型。Web 服务通过 OJB 框架访问数据库，业务代理实现类 NetstoreWebServiceDelegate 调用 Web 服务方法来提供服务，参见第 18 章（Struts 与 SOAP Web 服务）。

对于方式一和方式二，其业务代理和 DAO 的联合模式的结构图如图 6-11 所示。

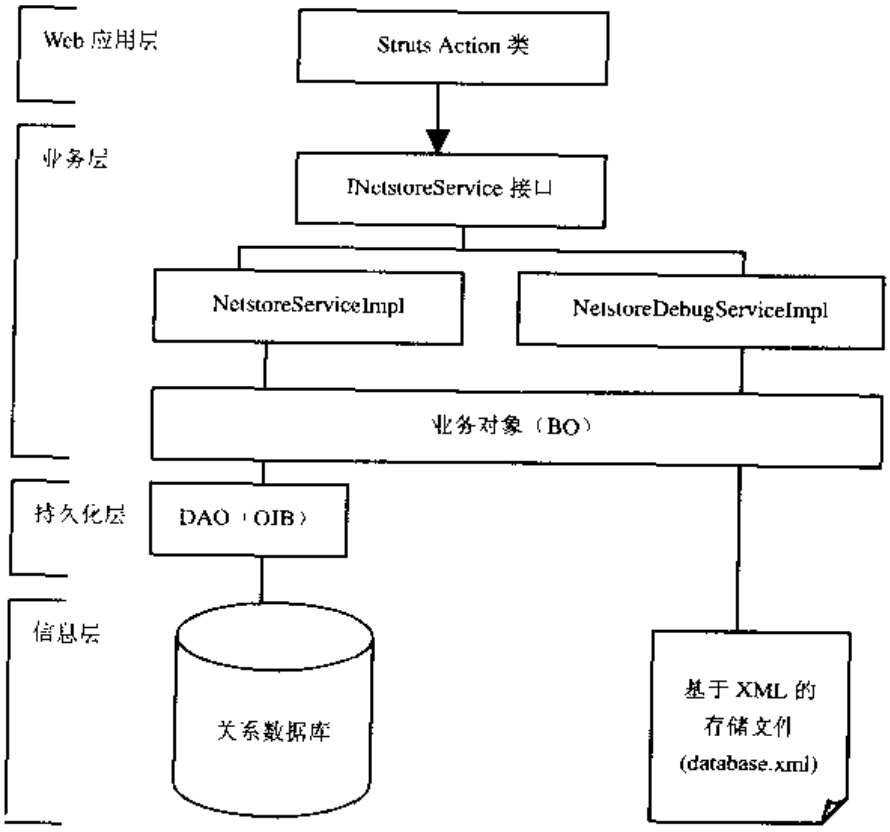

图 6-11　netstore 应用中的业务代理和 DAO 的联合模式

例程 6-8 为 Netstore 应用的业务代理接口 INetstoreService 的源程序。

例程 6-8　业务代理接口 INetstoreService.java

```
package netstore.service;

import javax.servlet.ServletContext;
import java.util.List;
import netstore.catalog.view.ItemDetailView;
import netstore.catalog.view.ItemSummaryView;
import netstore.framework.exceptions.DatastoreException;
import netstore.framework.security.IAuthentication;
/**
 * The business interface for the Netstore Application. It defines all
 * of the methods that a client may call on the Netstore application.
```

```
 *
 * This interface extends the IAuthentication interface to create a
 * cohesive interface for the Netstore application.
 */
public interface INetstoreService extends IAuthentication {

  public List getFeaturedItems() throws DatastoreException;

  public ItemDetailView getItemDetailView( String itemId )
    throws DatastoreException;

  public void setServletContext( ServletContext ctx );

  public void destroy();
}
```

例程 6-8 中的 INetstoreService 接口定义了所有被 Web 应用层调用的方法。INetstoreService 用来削弱服务和客户程序的关系，即使是其他类型的非 Web 客户程序也可以使用同样的服务。

INetstoreService 扩展了 IAuthentication 接口。IAuthentication 接口中声明了安全验证方法。例程 6-9 为 IAuthentication 接口的源程序，它仅包含两种方法。

例程 6-9　IAuthentication 接口

```
package netstore.framework.security;

import netstore.customer.view.UserView;
import netstore.framework.exceptions.InvalidLoginException;
import netstore.framework.exceptions.ExpiredPasswordException;
import netstore.framework.exceptions.AccountLockedException;
import netstore.framework.exceptions.DatastoreException;
/**
 * Defines the security methods for the system.
 */
public interface IAuthentication {
  /**
   * Log the user out of the system.
   */
  public void logout(String email);

  /**
   * Authenticate the user's credentials and either return a UserView for the
   * user or throw one of the security exceptions.
   */
  public UserView authenticate(String email, String password) throws
  InvalidLoginException,ExpiredPasswordException,AccountLockedException,DatastoreException;
```

```
}
```

例程 6-10（NetstoreServiceImpl.java）提供了 INetstoreService 接口的一种实现。也可以采用其他方式来实现这一接口，这不会影响客户程序，因为客户程序调用的是接口，而不是实现。

例程 6-10　NetstoreServiceImpl.java

```
package netstore.service;
//import all the necessary packages here
.....
public class NetstoreServiceImpl implements INetstoreService{
  ServletContext servletContext = null;

  // Implementation specific references
  Implementation odmg = null;
  Database db = null;

  /**
   * Create the service, which includes initializing the persistence
   * framework.
   */
  public NetstoreServiceImpl() throws DatastoreException {
    super();
    init();
  }

  public void setServletContext( ServletContext ctx ){
    this.servletContext = ctx;
  }

  public ServletContext getServletContext(){
    return servletContext;
  }

  /**
   * Return a list of items that are featured.
   */
  public List getFeaturedItems() throws DatastoreException {
    // Start a transaction
    Transaction tx = odmg.newTransaction();
    tx.begin();

    List results = null;
    try{
      OQLQuery query = odmg.newOQLQuery();
      // Set the OQL select statement
```

```
            query.create( "select featuredItems from " + ItemBO.class.getName() );
            results = (List)query.execute();
            tx.commit();
        }catch( Exception ex ){
            // Rollback the transaction
            tx.abort();
            ex.printStackTrace();
            throw DatastoreException.datastoreError(ex);
        }
        int size = results.size();
        List items = new ArrayList();
        for( int i = 0; i < size; i++ ){
            ItemBO itemBO = (ItemBO)results.get(i);
            ItemSummaryView newView = new ItemSummaryView();
            newView.setId( itemBO.getId().toString() );
            newView.setName(getGBString( itemBO.getDisplayLabel() ));
            newView.setUnitPrice( itemBO.getBasePrice() );
            newView.setSmallImageURL( itemBO.getSmallImageURL() );
            newView.setDescription(getGBString( itemBO.getDescription() ));
            items.add( newView );
        }
        return items;
    }
    /**
     * Return an detailed view of an item based on the itemId argument.
     */
    public ItemDetailView getItemDetailView( String itemId )
        throws DatastoreException{
            // Start a transaction
            Transaction tx = odmg.newTransaction();
            tx.begin();

            List results = null;
            try{
                OQLQuery query = odmg.newOQLQuery();

                // Set the OQL select statement
                String queryStr = "select item from " + ItemBO.class.getName();
                queryStr += " where id = $1";
                query.create(queryStr);
                query.bind(itemId);

                // Execute the transaction
                results = (List)query.execute();
                tx.commit();
            }catch( Exception ex ){
```

```
            // Rollback the transaction
            tx.abort();
            ex.printStackTrace();
            throw DatastoreException.datastoreError(ex);
        }

        if (results.isEmpty() ){
            throw DatastoreException.objectNotFound();
        }

        ItemBO itemBO = (ItemBO)results.get(0);

        // Build a ValueObject for the Item
        ItemDetailView view = new ItemDetailView();
        view.setId( itemBO.getId().toString() );
        view.setDescription(getGBString( itemBO.getDescription()));
        view.setLargeImageURL( itemBO.getLargeImageURL() );
        view.setName(getGBString( itemBO.getDisplayLabel()) );
        view.setProductFeature(getGBString( itemBO.getFeature1() ));
        view.setUnitPrice( itemBO.getBasePrice() );
        view.setTimeCreated( new Timestamp(System.currentTimeMillis() ));
        view.setModelNumber( itemBO.getModelNumber() );
        return view;
    }

    /**
     * Authenticate the user's credentials and either return a UserView for the
     * user or throw one of the security exceptions.
     */
    public UserView authenticate(String email, String password) throws
        InvalidLoginException,ExpiredPasswordException,AccountLockedException,
        DatastoreException {

        // Start a transaction
        Transaction tx = odmg.newTransaction();
        tx.begin();

        // Query the database for a user that matches the credentials
        List results = null;
        try{
            OQLQuery query = odmg.newOQLQuery();
            // Set the OQL select statement
            String queryStr = "select customer from " + CustomerBO.class.getName();
            queryStr += " where email = $1 and password = $2";
            query.create(queryStr);
```

```
            // Bind the input parameters
            query.bind( email );
            query.bind( password );

            // Retrieve the results and commit the transaction
            results = (List)query.execute();
            tx.commit();
          }catch( Exception ex ){
            // Rollback the transaction
            tx.abort();
            ex.printStackTrace();
            throw DatastoreException.datastoreError(ex);
          }

          // If no results were found, must be an invalid login attempt
          if ( results.isEmpty() ){
            throw new InvalidLoginException();
          }

          // Should only be a single customer that matches the parameters
          CustomerBO customer   = (CustomerBO)results.get(0);

          // Make sure the account is not locked
          String accountStatusCode = customer.getAccountStatus();
          if ( accountStatusCode != null && accountStatusCode.equals( "L" ) ){
            throw new AccountLockedException();
          }

          // Populate the Value Object from the Customer business object
          UserView userView = new UserView();
          userView.setId( customer.getId().toString() );
          userView.setFirstName( getGBString(customer.getFirstName() ));
          userView.setLastName(getGBString( customer.getLastName() ));
          userView.setEmailAddress( customer.getEmail() );
          userView.setCreditStatus( customer.getCreditStatus() );

          return userView;
        }

    /**
     * Log the user out of the system.
     */
    public void logout(String email){
      // Do nothing with right now, but might want to log it for auditing reasons
    }
```

```
    public void destroy(){
      // Do nothing for this example
    }

    /**
     * Opens the database and prepares it for transactions
     */
    private void init() throws DatastoreException {
      // get odmg facade instance
      odmg = OJB.getInstance();
      db = odmg.newDatabase();
      //open database
      try{
        db.open("default", Database.OPEN_READ_WRITE);
      }catch( Exception ex ){
        System.out.println( ex );
        throw DatastoreException.datastoreError(ex);
      }
    }
     private static String getGBString(String s){
       try{
         return new String(s.getBytes("ISO-8859-1"),"GB2312");
       }catch(Exception e){return null;}
    }
}
```

以上类实现了 INetstoreService 接口中的所有方法。因为 INetstoreService 接口扩展了 IAuthentication 接口，NetstoreServiceImpl 类也必须实现安全验证方法。业务代理接口的实现和 Struts 框架以及 Web 容器完全独立，这使得它可以被各种类型的应用重用，这是在本章一开始就提出要达到的目标。

原先提及应该调用 OJB 框架中的一些方法来解析 XML 映射信息，并且建立和数据库的连接，这些步骤在 NetstoreServiceImpl 类的初始化方法 init()方法中完成。当这个实现类的构造方法被调用时，将会加载并解析 XML 文件。当构造方法成功执行完毕时，持久化框架就可以被调用了。

业务代理实现类的构造方法应该由客户程序来调用。在本应用中使用一个工厂类，它同时也是个 Struts 插件（PlugIn），它决定初始化哪个业务代理实现类。例程 6-11 为工厂类 NetstoreServiceFactory 的源程序。

例程 6-11　NetstoreServiceFactory 类

```
package netstore.service;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import org.apache.struts.action.PlugIn;
```

```
import org.apache.struts.action.ActionServlet;
import org.apache.struts.config.ModuleConfig;
import netstore.framework.util.IConstants;

/**
 * A factory for creating Netstore Service Implementations. The specific
 * service to instantiate is determined from the initialization parameter
 * of the ServiceContext. Otherwise, a default implementation is used.
 * @see netstore.service.NetstoreDebugServiceImpl
 */
public class NetstoreServiceFactory implements INetstoreServiceFactory, PlugIn{
  // Hold onto the servlet for the destroy method
  private ActionServlet servlet = null;
  // The default is to use the debug implementation
   String serviceClassname =
   "netstore.service.NetstoreDebugServiceImpl";

   public INetstoreService createService() throws
    ClassNotFoundException, IllegalAccessException, InstantiationException {
    String className = servlet.getInitParameter( IConstants.SERVICE_CLASS_KEY );

    if (className != null ){
       serviceClassname = className;
    }
    INetstoreService instance =
     (INetstoreService)Class.forName(serviceClassname).newInstance();

    instance.setServletContext( servlet.getServletContext() );

    return instance;
  }

  public void init(ActionServlet servlet, ModuleConfig config)
    throws ServletException{
    // Store the servlet for later
    this.servlet = servlet;

    /* Store the factory for the application. Any Netstore service factory
     * must either store itself in the ServletContext at this key, or extend
     * this class and don't override this method. The Netstore application
     * assumes that a factory class that implements the IStorefrtonServiceFactory
     * is stored at the proper key in the ServletContext.
     */
    servlet.getServletContext().setAttribute( IConstants.SERVICE_FACTORY_KEY, this );
  }
  public void destroy(){
    // Do nothing for now
  }
```

```
}
```

NetstoreServiceFactory 类从 web.xml 文件中读取初始化参数，该参数指明需要实例化的 INetstoreService 实现类的类名。如果不存在这个初始化参数，就使用默认的实现类（本例中为 NetstoreDebugServiceImpl）。如果希望 NetstoreServiceFactory 类实例化 NetstoreServiceImpl，可以在 web.xml 文件中对 ActionServlet 类配置如下初始化参数：

```
<servlet>
  <servlet-name>netstore</servlet-name>
  <servlet-class>netstore.framework.ExtendedActionServlet</servlet-class>
   ......
  <init-param>
    <param-name>netstore-service-class</param-name>
    <param-value>netstore.service.NetstoreServiceImpl</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

由于工厂类实现了 Struts 插件接口，因此 Struts 应用启动时将加载该插件类，创建它的实例，并且调用它的 init()方法进行初始化。该 init()方法把服务工厂类本身的实例保存到 application 范围中，在需要的时候可以再把它取出来：

```
servlet.getServletContext().setAttribute( IConstants.SERVICE_FACTORY_KEY, this );
```

为了创建服务实现类的实例，客户程序（如 Action 类）需要从 ServletContext 中取出工厂类实例，然后调用它的 createService()方法。createService()方法将调用服务实现类的不带参数的构造方法。Action 类调用 netstore 业务代理接口的时序图如图 6-12 所示。

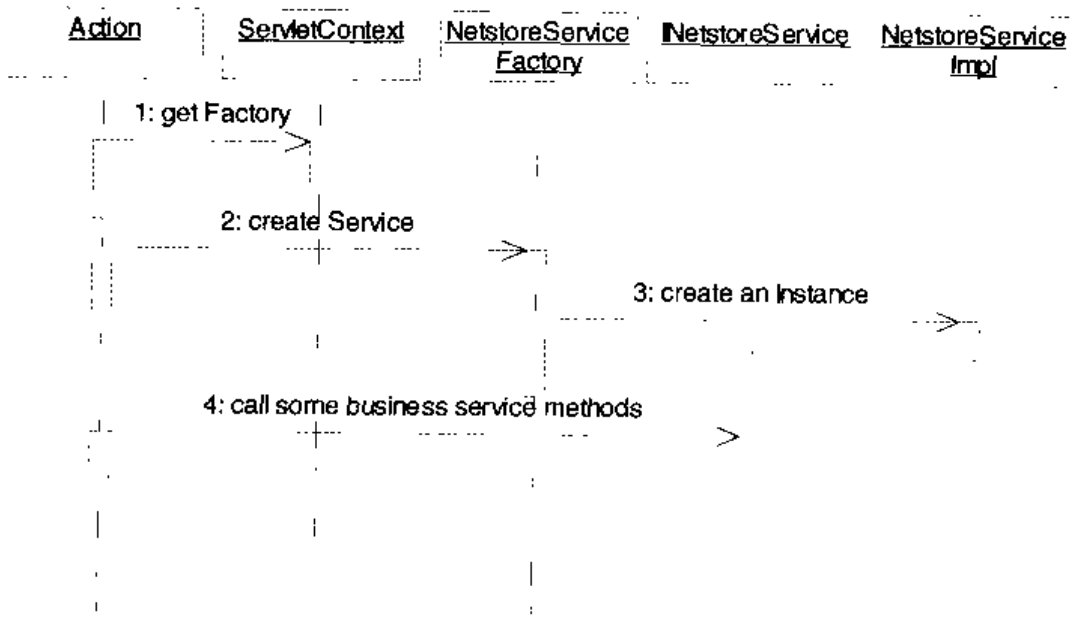

图 6-12　Action 类调用 netstore 业务代理接口的时序图

例程 6-12 为 LoginAction 类的源程序，调用 netstore 业务代理接口的相关代码用粗体字来表示。

例程 6-12 LoginAction 类

```
package netstore.security;

//import all the necessary packages here
.....
/**
 * Implements the logic to authenticate a user for the netstore application.
 */
public class LoginAction extends NetstoreBaseAction {
  protected static Log log = LogFactory.getLog( NetstoreBaseAction.class );
  /**
   * Called by the controller when the a user attempts to login to the
   * netstore application.
   */
  public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response )
  throws Exception{

    // Get the user's login name and password. They should have already
    // validated by the ActionForm.
    String email = ((LoginForm)form).getEmail();
    String password = ((LoginForm)form).getPassword();

    // Obtain the ServletContext
    ServletContext context = getServlet().getServletContext();

    // Login through the security service
    INetstoreService serviceImpl = getNetstoreService();

    // Attempt to authenticate the user
    UserView userView = serviceImpl.authenticate(email, password);

    UserContainer existingContainer = getUserContainer(request);
    existingContainer.setUserView(userView);
    System.out.println( existingContainer.getUserView() );
    return mapping.findForward(IConstants.SUCCESS_KEY);
  }
}
```

第一行粗体字表明 LoginAction 类继承 NetstoreBaseAction 类，第二行粗体字调用

getNetstoreService()方法，这一方法在 NetstoreBaseAction 父类中定义，以便让所有的 Action 子类都能调用这个方法。getNetstoreService()方法先取出工厂实例，再调用工厂的 createService()方法。例程 6-13 为 NetstoreBaseAction 类的 getNetstoreService()方法的源代码。

例程 6-13　NetstoreBaseAction 类的 getNetstoreService()方法

```
protected INetstoreService getNetstoreService(){
    INetstoreServiceFactory factory =
    (INetstoreServiceFactory)getApplicationObject( IConstants.SERVICE_FACTORY_KEY );
    INetstoreService service = null;

    try{
      service = factory.createService();
    }catch( Exception ex ){
      log.error( "Problem creating the Netstore Service", ex );
    }
    return service;
}
```

LoginAction 类中第三行粗体字调用服务方法 authenticate()。authenticate()方法返回一个 UserView 对象：

```
UserView userView = serviceImpl.authenticate(email, password);
```

如果安全验证失败，authenticate()方法将抛出 InvalidLoginException 异常。

值得注意的是，Action 类引用的是 INetstoreService 接口，而不是它的实现类。正如前面所说的，这可以保证当 INetstoreService 接口的实现发生改变时，不对 Action 类的程序代码构成任何影响。

6.6　小　　结

本章介绍了模型的实现方法。Struts 框架并没有在模型层提供现成可用的组件。模型的实现应该和 Struts 应用的控制层以及视图层保持独立。

模型采用业务对象来描述状态和行为，为了使业务对象持久化，需要把业务对象映射到关系型数据库。本章采用 OJB 作为持久化框架，它负责连接和访问数据库，OJB 是 DAO 设计模式的一种实现。

模型向客户程序提供了业务代理接口，业务代理接口直接访问持久化框架，处理实际的业务逻辑。Struts 应用的 Action 类可以使用这个业务代理接口，而不必直接和持久化框架交互。这种做法有助于削弱上层 Web 应用和持久化框架之间的关系，提高持久化框架的相对独立性。

第 7 章　Struts 视图组件

Struts 框架的视图负责为客户提供动态网页内容。Struts 视图主要由 JSP 网页构成，此外，Struts 框架还提供了 Struts 客户化标签和 ActionForm Bean，这些组件提供对国际化、接收用户输入的表单数据、表单验证和错误处理等的支持，使开发者可以把更多的精力放在实现业务需求上。

本章重点介绍了 ActionForm 的运行机制和使用方法，此外还介绍了动态 ActionForm 的配置和使用方法。Struts 视图离不开 Struts 客户化标签的支持，关于 Struts 客户化标签的用法，将在本书的第 12～16 章做详细介绍。

7.1　视 图 概 述

视图是模型的外在表现形式，用户通过视图来了解模型的状态。同一个模型可以有多种视图，例如，netstore 应用的主页向用户显示了所有商品的概要信息，如图 7-1 所示，而在商品明细页面上则显示了单个商品的详细信息，如图 7-2 所示。用户可以根据自己的需要，来访问不同的视图。

图 7-1　netstore 应用的主页

图 7-2 netstore 应用的商品明细网页

在 Stuts 框架中，视图主要由 JSP 组件构成，此外，视图还可以包含以下组件：

- HTML 文档
- JSP 客户化标签
- JavaScript 和 stylesheet
- 多媒体文件
- 消息资源（Resource Bundle）
- ActionForm Bean

7.2 在视图中使用 JavaBean

JavaBean 是可重用的、平台独立的 Java 组件，JavaBean 支持属性、事件、方法和持久化。Struts 框架仅利用了 JavaBean 的一小部分特性。在 Struts 应用中的 JavaBean 和普通的 Java 类很相似，不过，它应该遵守以下规范：

- 必须提供不带参数的构造方法。
- 为 Bean 的所有属性提供公共类型的 get/set 方法。
- 对于 boolean 类型的属性，如果存在 isXXX()方法，那么该方法返回 boolean 类型的属性值。
- 对于数组类型的属性，应该提供 getXXX(int index)和 setXXX(int index, Property Element value)方法，用来读取或设置数组中的元素。

7.2.1 DTO 数据传输对象

在本书的第 6 章（Struts 模型组件）介绍过可以利用 JavaBean 来创建业务对象，实体业务对象包含了模型的状态信息。此外，Struts 框架还利用 JavaBean 来创建数据传输对象

（Data Transfer Object，简称 DTO）。DTO 用于在不同的层之间传递数据。例如，在 netstore 应用中，模型层和视图层之间就通过 DTO 来传递数据，如图 7-3 所示。模型层的数据实际上经过控制层再到达视图层，为了简化起见，图 7-3 中省略了控制层。

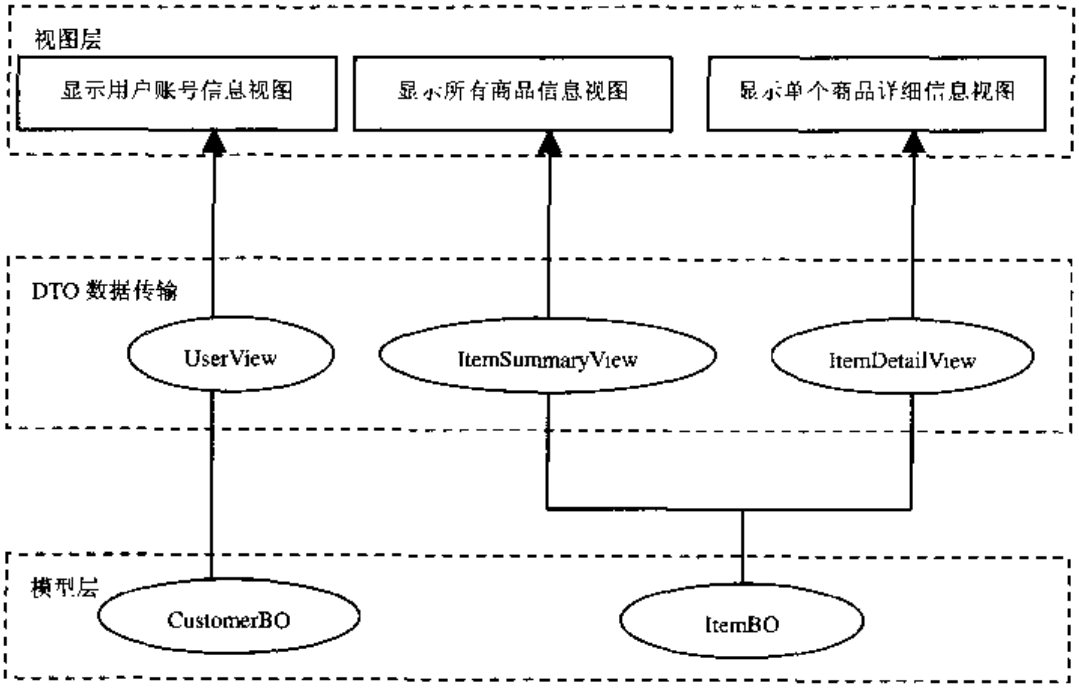

图 7-3　netstore 应用采用 DTO 在不同的层之间传递数据

在图 7-3 中，并没有把模型层的业务对象直接传递到视图层（从技术角度来说是可以实现的），而是采用 DTO 来传输数据，这样做有两个好处：

- 减少传输数据的冗余，提高传输效率。例如，对于显示所有商品信息的视图，并不需要显示每个商品的详细信息，因此可以创建针对这个视图的 ItemSummaryView Bean，它仅包含了商品的简要信息。
- 有助于实现各个层之间的独立，使每个层分工明确。模型层负责业务逻辑，视图层负责向用户展示模型状态。采用 DTO，模型层对视图层屏蔽了业务逻辑细节，向视图层提供可以直接显示给用户的数据。

7.2.2　Struts 框架提供的 DTO：ActionForm Bean

ActionForm Bean 是 Struts 框架提供的 DTO，用于在视图层和控制层之间传递 HTML 表单数据。控制层可以从 ActionForm Bean 中读取用户输入的表单数据，也可以把来自模型层的数据存放到 ActionForm Bean 中，然后把它返回给视图。ActionForm Bean 还具有表单验证功能，可以为模型层过滤不合法的数据。

在本书第 6 章（Struts 模型组件），曾经强调过模型层应该和 Web 应用层保持独立。由于 ActionForm 类中使用了 Servlet API，因此不提倡直接把 ActionForm Bean 传给模型层，而应该在控制层把 ActionForm Bean 的数据重新组装到自定义的 DTO 中，再把它传递给模型层。在各个层之间传输数据的 DTO 如图 7-4 所示。

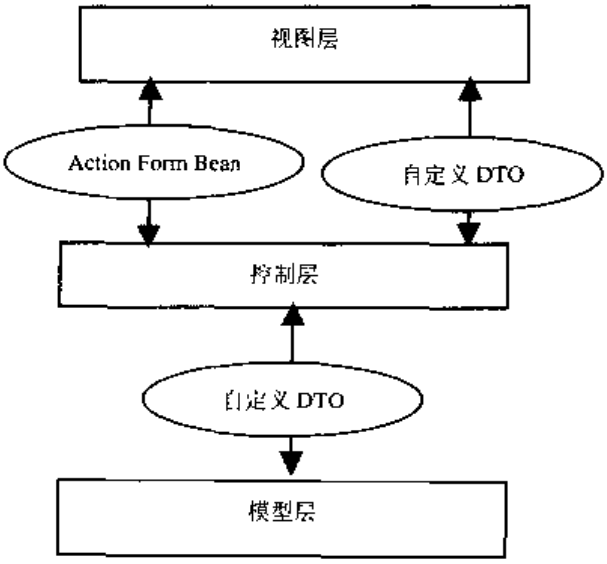

图 7-4　在各个层之间传输数据的 DTO

7.3　使用 ActionForm

7.3.1　ActionForm 的生命周期

ActionForm Bean 有两种存在范围：request 和 session。如果 ActionForm 存在于 request 范围，它仅在当前的请求/响应生命周期中有效。在请求从一个 Web 组件转发到另一个 Web 组件的过程中，ActionForm 实例一直有效。当服务器把响应结果返回给客户，ActionForm 实例及其包含的数据就会被销毁。如果 ActionForm 存在于 session 范围，同一个 ActionForm 实例在整个 HTTP 会话中有效。ActionForm 的范围如图 7-5 所示。

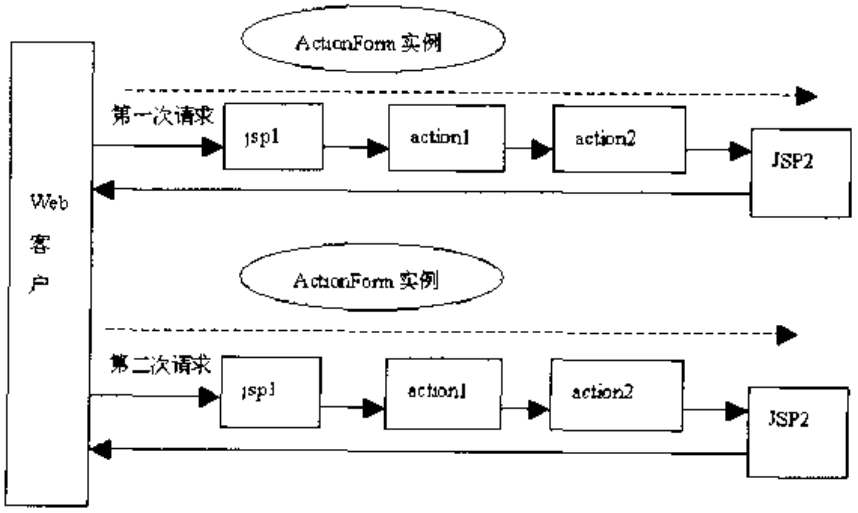

图 7-5　ActionForm 的范围

如图 7-5 所示，假定发给 jsp1 组件的请求依次转发给 action1、action2 和 jsp2 组件，jsp2 组件最后生成响应结果，返回给客户。如果 ActionForm 在 request 范围内，那么在第一个请求/响应周期内，以上四个组件由于共享同一个 HttpServletRequest 对象，所以也共用同一个 ActionForm 实例。当用户发出第二个请求时，Struts 框架将创建一个新的 ActionForm 实例，以上四个组件共用这个新的 ActionForm 实例。

如果 ActionForm 在 session 范围内，那么无论是第一次请求，还是第二次请求，只要这两个请求处于同一个 HTTP 会话中，这些 Web 组件就始终共用同一个 ActionForm 实例。

在 Struts 配置文件中，<action>元素的 scope 属性用来设置 ActionForm 的范围，默认值为 session。

当控制器接收到请求时，如果请求访问的 Web 组件为 Action，并且为这个 Action 配置了和 ActionForm 的映射，控制器将从 request 或 session 范围中取出 ActionForm 实例，如果该实例不存在，就会自动创建一个新的实例。当控制器接收到一个新的请求时，ActionForm 的生命周期如图 7-6 所示。

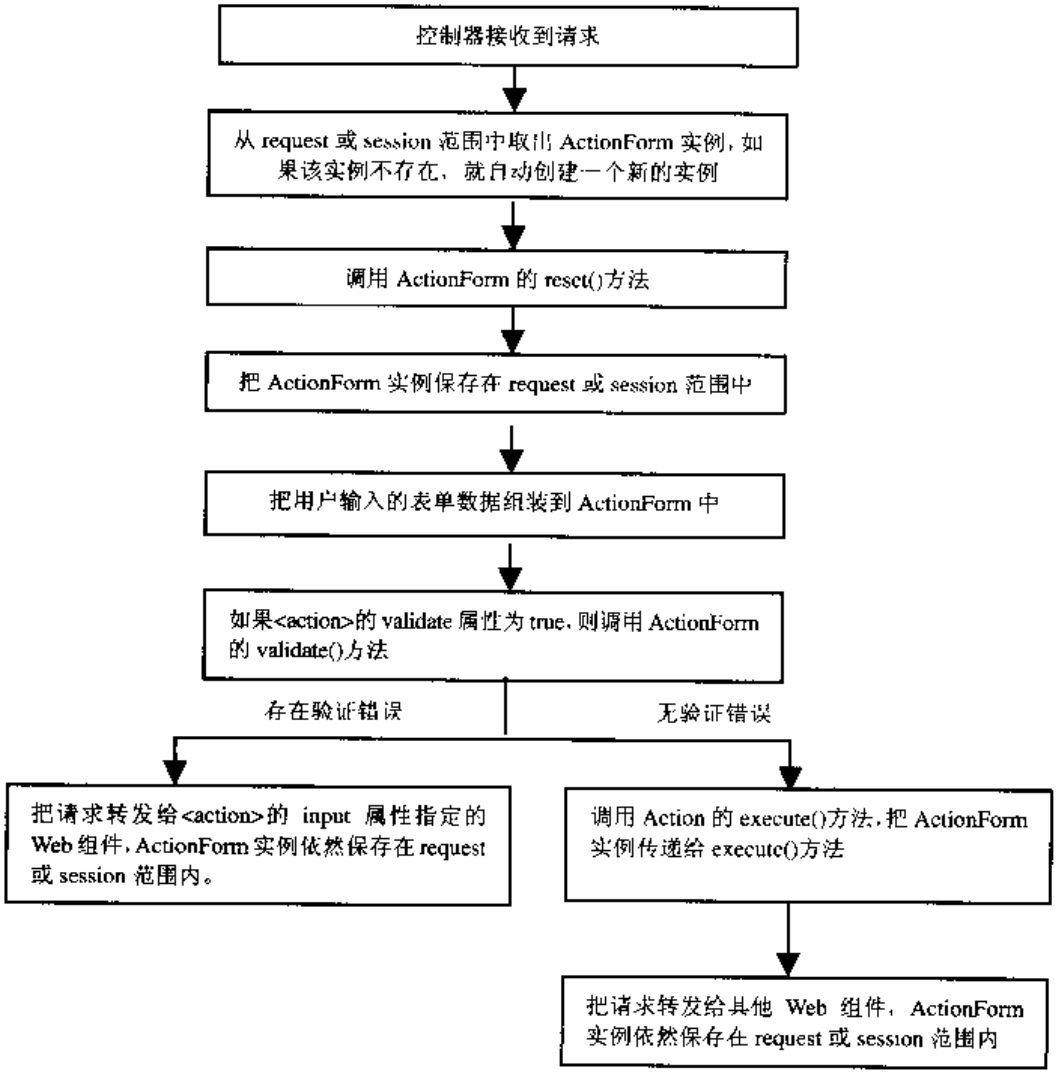

图 7-6　ActionForm 的生命周期

图 7-6 显示了 Struts 框架在处理 HTTP 请求的过程中和 ActionForm 相关的步骤。当 ActionForm 检查到了表单验证错误时，就会把请求转发给<action>元素的 input 属性指定的 Web 组件。此时存放了用户表单数据的 ActionForm 实例依然存在，input 属性指定的 Web 组件中的<html:form>标签可以从 ActionForm 实例中取出数据，把它们输出到网页的表单上。

7.3.2 创建 ActionForm

Struts 框架中定义的 ActionForm 类是抽象的，必须在应用中创建它的子类，来捕获具体的 HTML 表单数据，ActionForm Bean 中的属性和 HTML 表单中的字段一一对应。例如，对于 netstore 应用的登入表单，创建了与之对应的 LoginForm Bean，参见例程 7-1。

例程 7-1 LoginForm.java

```
package netstore.security;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;
/**
 * Form bean for the user signon page.
 */
public class LoginForm extends ActionForm {
  private String password = null;
  private String email = null;

  /**
   * Public accessors and mutators
   */
  public void setEmail(String email) {
    this.email = email;
  }

  public String getEmail() {
    return (this.email);
  }

  public String getPassword() {
    return (this.password);
  }

  public void setPassword(String password) {
    this.password = password;
  }

  public ActionErrors validate(ActionMapping mapping,
```

```
                                        HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();

    if(getEmail() == null || getEmail().length() < 1) {
      errors.add( ActionErrors.GLOBAL_MESSAGE,
                  new ActionMessage("global.required", "email" ));
    }
    if(getPassword() == null  | getPassword().length() < 1) {
      errors.add( ActionErrors.GLOBAL_MESSAGE,
                  new ActionMessage("global.required", "password" ));
    }
    return errors;
  }

  /**
   * Reset all properties to their default values.
   *
   * @param mapping The mapping used to select this instance
   * @param request The servlet request we are processing
   */
  public void reset(ActionMapping mapping,
                    HttpServletRequest request) {
    this.password = null;
    this.email = null;
  }
}
```

1．validate()方法

如果 Struts 的配置文件满足以下两个条件，Struts 控制器就会调用 ActionForm 的 validate()方法：

- 为 ActionForm 配置了 Action 映射，即<form-bean>元素的 name 属性和<action>元素的 name 属性匹配。
- <action>元素的 validate 属性为 true。

在 ActionForm 基类中定义的 validate()方法直接返回 null，如果创建了扩展 ActionForm 基类的子类，那么应该在子类中覆盖 validate()方法。

validate()方法返回 ActionErrors 对象，如果返回的 ActionErrors 对象为 null，或者不包含任何 ActionMessage 对象，就表示没有错误，数据验证通过。如果 ActionErrors 中包含 ActionMessage 对象，就表示发生了验证错误

validate()方法主要负责检查数据的格式和语法，而不负责检查数据是否符合业务逻辑。例如，HTML 表单的文本域只能处理字符串类型的数据，即使是年龄、数量等数字类型的数据，也被当做字符串处理。所以在 ActionForm 中可以把年龄、数量等定义为字符串类型属性，然后在 validate()方法中进行验证，判断是否可以把用户输入的年龄、数量转换为有效的数字。以下是验证用户输入的订单数量 orderQtyStr 是否为有效数字的代码：

```
private String orderQtyStr=null;
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
  // Validate that the qty entered was in fact a number
  try{
    // Integer.parse was not used because it's not really I18N-safe
    java.text.Format format = java.text.NumberFormat.getNumberInstance();
    Number orderQty = (Number)format.parseObject( orderQtyStr );
  }catch( Exception cx ){
    // The quantity entered by the user was not a valid qty
    errors.add( ActionErrors.GLOBAL_MESSAGE,
                new ActionMessage( "order.quantity.invalid" ));
  }
  ......
}
```

2．reset()方法

如图 7-6 所示，不管 ActionForm 存在于哪个范围内，对于每一个请求，控制器都会先调用 ActionForm 的 reset()方法，然后再把用户输入的表单数据组装到 ActionForm 中。reset()方法用于恢复 ActionForm 的属性的默认值，例如把 boolean 类型属性设为 true 或 false，把字符串属性设为 null 或某个初始值。

如果 ActionForm 在 request 范围内，那么对于每个新的请求都会创建新的 ActionForm 实例。当新的实例创建后，如果它的属性已经被初始化为默认值，那么接着再在 reset()方法中把属性设为默认值不是很有必要。因此在这种情况下，可以让 reset()方法为空。

对于 session 范围内的 ActionForm，同一个 ActionForm 实例会被多个请求共享，reset()方法在这种情况下极为有用。

7.3.3 配置 ActionForm

Struts 配置文件的<form-beans>元素用来配置所有的 ActionForm Bean。<form-beans>元素可以包含多个<form-bean>子元素，它代表单个的 ActionForm Bean，例如：

```
<form-bean
    name="loginForm"
    type="netstore.security.LoginForm"/>
</form-bean>
```

接下来为 ActionForm 配置 Action 映射。同一个 ActionForm 可以和多个 Action 映射。在<action>元素中，name 和 scope 属性分别指定 ActionForm 的名字和范围，validate 属性指定是否执行表单验证：

```
<action
    path="/signin"
    type="netstore.security.LoginAction"
```

```
    scope="request"
    name="loginForm"
    validate="true"
    input="/security/signin.jsp">
    <forward name="Success" path="/action/home"/>
    <forward name="Failure" path="/security/signin.jsp" redirect="true"/>
</action>
```

7.3.4　访问 ActionForm

ActionForm 可以被 JSP、Struts 标签、Action 和其他 Web 组件访问。访问 ActionForm 大致有以下一些方法：

1．使用 Struts HTML 标签库

Struts HTML 标签库提供了一组和 ActionForm 密切关联的标签，<html:form>标签生成 HTML 表单，它包括<html:text>、<html:select>、<html:option>、<html:radio>和<html:submit>等子标签，这些子标签构成 HTML 表单的字段或按钮。<html:form>标签能和 ActionForm 交互，读取 ActionForm 的属性值，把它们赋值给表单中对应的字段。关于 HTML 标签库中标签的使用方法，请参见第 12 章（Struts HTML 标签库）的内容。

2．从 request 或 session 范围内取出 ActionForm 实例

Struts 框架把 ActionForm 实例保存在 HttpServletRequest 或 HttpSession 中，保存时采用的属性 key 为<form-bean>元素的 name 属性。因此，如果 ActionForm 在 request 范围内，则可以调用 HttpServletRequest 的 getAttribute()方法读取 ActionForm 实例，例如：

```
LoginForm loginForm=(LoginForm)request.getAttribute("loginForm");
```

如果 ActionForm 在 session 范围内，则可以调用 HttpSession 的 getAttribute()方法读取 ActionForm 实例，例如：

```
LoginForm loginForm=(LoginForm)session.getAttribute("loginForm");
```

3．在 Action 类的 execute()方法中直接访问 ActionForm

如果配置了 ActionForm 和 Action 的映射，Struts 框架就会把 ActionForm 作为参数传递给 Action 的 execute()方法，因此在 Action 类的 execute()方法中可以读取或设置 ActionForm 属性，参见例程 7-2。

例程 7-2　在 Action 类的 execute()方法中访问 ActionForm

```
public ActionForward execute( ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response )
  throws Exception{
```

```
        // Get the user's login name and password. They should already have
        // been validated by the ActionForm.
        String email = ((LoginForm)form).getEmail();
        String password = ((LoginForm)form).getPassword();

        // Log in through the security service.
        INetstoreService serviceImpl = getNetstoreService();
        UserView userView = serviceImpl.authenticate(email, password);

        SessionContainer existingContainer = null;
        HttpSession session = request.getSession(false);
        if ( session != null ){
          existingContainer = getSessionContainer(request);
          session.invalidate();
        }else{
          existingContainer = new SessionContainer();
        }

        // Create a new session for the user.
        session = request.getSession(true);
        existingContainer.setUserView(userView);
        session.setAttribute(IConstants.SESSION_CONTAINER_KEY, existingContainer);

        return mapping.findForward(IConstants.SUCCESS_KEY);
    }
```

7.3.5 处理表单跨页

有的时候，由于表单数据太多，无法在同一个页面显示（如用于用户注册的表单），可以把它拆分成多个表单，分多个页面显示。在这种情况下，既可以为每个表单创建单独的 ActionForm，也可以只创建一个 ActionForm，它和多个表单对应。

下面以 addressbook 应用的 insertForm 表单为例，讨论如何实现一个 ActionForm 对应多个表单。在本书的第 3 章中（Struts 应用的分析与设计）对 addressbook 应用做了详细介绍，本章介绍的 addressbook 应用的源代码位于配套光盘的 sourcecode/addressbook/version3 目录下。

尽管 insertForm 表单只包含三个字段：name、phone 和 address，但为了演示如何把表单分页，我们假定需要把 insertForm 表单拆分成两个表单：第一个表单在 insertContent.jsp 中定义，包括 name 和 phone 字段，如图 7-7 所示；第二个表单在 insertContent_next.jsp 中定义，包括 address 字段，如图 7-8 所示。

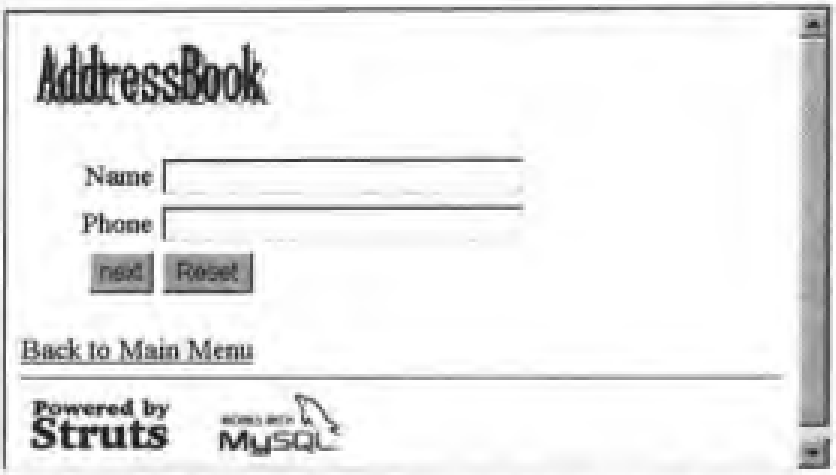

图 7-7　在 insertContent.jsp 中定义的第一个表单

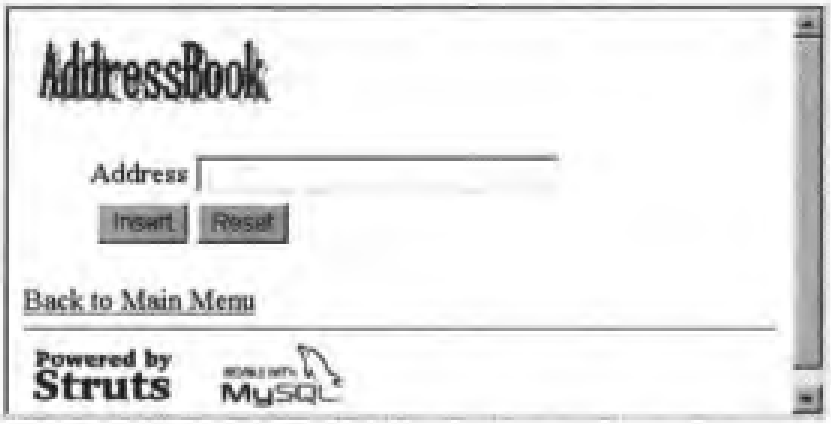

图 7-8　在 insertContent_next.jsp 中定义的第二个表单

以上两个表单分别对应不同的 Action：“/insert1”和“/insert2”，但是这两个 Action 与同一个 ActionForm 映射。如图 7-9 所示为这些 Web 组件之间的转发关系。

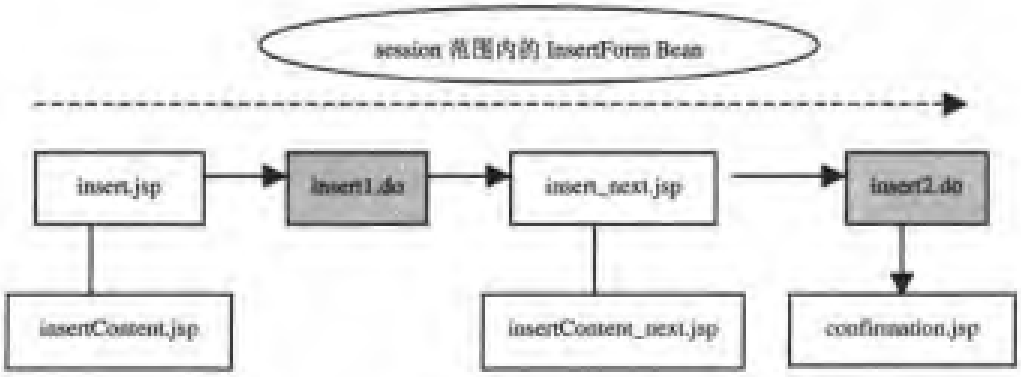

图 7-9　和 Insert 操作有关的 Web 组件之间的转发关系

1．把 HTML 表单拆分到多个 JSP 页面中

在 insertContent.jsp 和 insertContent_next.jsp 中均定义了 HTML 表单，由于这两个表单

都对应于同一个 ActionForm，因此可以在每个表单中定义一个隐含字段<html:hidden property="page"/>，它代表当前页面编号，ActionForm 将通过这个字段来识别当前正在处理的是哪个表单。例程 7-3 和 7-4 为定义两个 HTML 表单的代码。

例程 7-3　insertContent.jsp 中定义 HTML 表单的代码

```
<html:form action="/insert1.do" focus="title">
  <html:hidden property="page" value="1"/>
  <center>
    <table border="0" cellspacing="2" cellpadding="2" width="100%">
      <tr>
        <td align="right"> <bean:message key="prompt.name"/></td>
        <td><html:text property="name" size="25" maxlength="25"/></td>
      </tr>
      <tr>
        <td align="right"> <bean:message key="prompt.phone"/></td>
        <td><html:text property="phone" size="25" maxlength="10"/></td>
      </tr>
       <tr>
        <td align="right">
          <html:submit property="submit" value="next">
          <bean:message key="button.insert"/>
          </html:submit>
        </td>
        <td align="left">
          <html:reset   >
          <bean:message key="button.reset"/>
          </html:reset>
        </td>
      </tr>
    </table>
  </center>
</html:form>
```

例程 7-4　insertContent_next.jsp 中定义 HTML 表单的代码

```
<html:form action="/insert2.do" focus="title">
  <html:hidden property="page" value="2"/>
  <center>
    <table border="0" cellspacing="2" cellpadding="2" width="100%">
      <tr>
        <td align="right"> <bean:message key="prompt.address"/></td>
        <td><html:text property="address" size="25" maxlength="50"/></td>
      </tr>
      <tr>
        <td align="right">
          <html:submit property="submit" >
```

```
            <bean:message key="button.insert"/>
            </html:submit>
          </td>
          <td align="left">
            <html:reset  >
            <bean:message key="button.reset"/>
            </html:reset>
          </td>
        </tr>
      </table>
    </center>
  </html:form>
```

2. 创建和多个 HTML 表单对应的 ActionForm

以上两个 HTML 表单都对应 InsertForm Bean。在创建 InsertForm 时有以下几点需要注意：

- 提供和 HTML 表单的隐藏字段 page 对应的 page 属性：

```
private String page=null;
public String getPage(){
    return page;
}
public void setPage(String page){
    this.page=page;
}
```

- 在 reset()方法中，只能把和当前正在处理的表单相关的属性恢复为默认值，否则，如果每次都把 ActionForm 的所有属性恢复为默认值，将使用户输入的上一页表单数据丢失。由于 Struts 框架先调用 reset()方法，然后再把用户输入的表单数据组装到 ActionForm 中，因此在 reset()方法中，不能根据 page 属性来判断处理的是哪个页面，而应该直接从 HttpServletRequest 对象中读取当前表单的 page 字段值：

```
int numPage=new Integer(request.getParameter("page")).intValue();
```

- 在 validate()方法中，仅对和当前表单相关的属性进行验证。由于 Struts 框架在调用 validate()方法之前，已经把用户输入的表单数据组装到 ActionForm 中，因此在 validate()方法中可以根据 page 属性决定正在处理哪个表单。

例程 7-5 为 InsertForm 的代码。

例程 7-5　和两个 HTML 表单对应的 InsertForm.java

```
package addressbook.forms;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionMessage;
```

```
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

/**
 * <strong>InsertForm</strong> handles the form
 * that the user will use to insert a new Address into
 * the database.
 */
public final class InsertForm extends ActionForm {
    private String name = null;
    private String phone = null;
    private String address = null;
    private String page=null;

      public String getName() {
        return name;
      }
      public String getPhone() {
        return phone;
      }
      public String getAddress() {
        return address;
      }

      public String getPage(){
        return page;
      }
      public void reset(ActionMapping mapping, HttpServletRequest request) {
        int numPage=0;
        try{
              numPage=new Integer(request.getParameter("page")).intValue();
         }catch(Exception e){}

        if(numPage==1){
              name=null;
              phone=null;
        }
        if(numPage==2){
              address=null;
        }
        page=null;
      }

      public void setName(String name) {
        this.name = name;
```

```
    }
    public void setPhone(String phone) {
      this.phone= phone;
    }
    public void setAddress(String address) {
      this.address = address;
    }

    public void setPage(String page){
      this.page=page;
    }
    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request) {

      ActionErrors errors = new ActionErrors();
      int numPage=0;
      try{
            numPage=new Integer(page).intValue();
       }catch(Exception e){}

      if(numPage==1){
        if (((name == null) || (name.length() < 1)))
          errors.add("name", new ActionMessage("error.name.required"));
        if(((phone == null) || (phone.length() < 1)))
          errors.add("phone", new ActionMessage("error.phone.required"));
      }
      if(numPage==2){
          if(((address == null) || (address.length() < 1)))
          errors.add("address", new ActionMessage("error.address.required"));
      }
     return errors;
    }
}
```

3. 配置 ActionForm 和多个 Action 映射

当 ActionForm 与多个表单对应时，应该把 ActionForm 存放在 session 范围内。在本例中，当用户提交第一个表单时，请求由 org.apache.struts.actions.ForwardAction 来处理，ForwardAction 类是 Struts 框架内置的 Action 类，它的 execute()方法负责把请求再转发给<action>元素的 parameter 属性指定的 Web 组件。当用户提交第二个表单时，请求被转发给 InsertAction。无需对原来的 InsertAction 类做任何改动，它的源程序保持不变。以下是<action>元素的配置代码：

```
<action path="/insert1"
        parameter="/insert_next.jsp"
        type="org.apache.struts.actions.ForwardAction"
```

```
            name="insertForm"
            scope="session"
            input="/insert.jsp"
            validate="true">
    </action>
    <action    path="/insert2"
               type="addressbook.actions.InsertAction"
               name="insertForm"
               scope="session"
               input="/insert_next.jsp"
               validate="true">
    </action>
```

如图 7-10 所示为当用户分别提交第一、二个表单后 InsertForm Bean 的状态图。

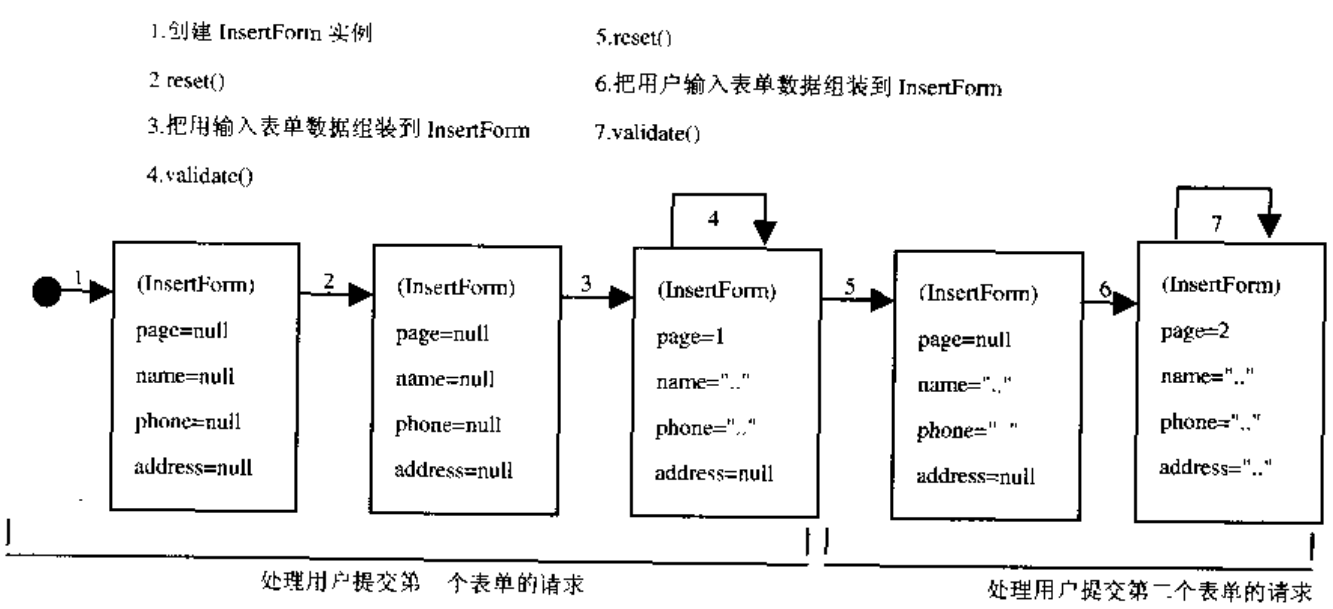

图 7-10 InsertForm 的状态图

7.4 使用动态 ActionForm

在 Struts 框架中，ActionForm 对象用来包装 HTML 表单数据，并能动态返回用于显示给用户的数据。自定义的 ActionForm 必须符合 JavaBean 规范，并继承 Struts 的 ActionForm 类，同时，用户可以有选择地覆盖两个方法：reset()和 validate()。

ActionForm 的以上特性可以简化 Web 应用的开发，因为它可以协助自动进行表示层的数据验证。ActionForm 的惟一缺点是对于大型的 Struts 应用，必须以编程的方式创建大量的 ActionForm 类，如果 HTML 表单的字段发生变化，就必须修改并重编译相关的 ActionForm 类。

Struts 1.1 对此做出了改进，引入了动态 ActionForm 类的概念。Struts 框架的 DynaActionForm 类及其子类实现了动态 ActionForm，DynaActionForm 类是 ActionForm 类的子类。

7.4.1　配置动态 ActionForm

动态 ActionForm 支持在 Struts 配置文件中完成 ActionForm 的全部配置，没有必要编写额外的程序来创建具体的 ActionForm 类。配置动态 ActionForm 的方法为：在 Struts 配置文件中配置一个<form-bean>元素，将 type 属性设置成 DynaActionForm 或它的某个子类的全名。以下代码配置了一个名为 loginForm 的动态 ActionForm，它包含三个属性：username，password 和 rememberMe。

```
<form-beans>
  <form-bean
    name="loginForm"
    type="org.apache.struts.action.DynaActionForm">

    <!-- Specify the dynamic properties of the form -->
    <form-property
      name="email"
      type="java.lang.String "/>
    <form-property
      name="password"
      type="java.lang.String "/>

    <!-- You can also set the initial value of a property -->
    <form-property
      initial="false"
      name="rememberMe"
      type="java.lang.Boolean "/>
  </form-bean>
<form-beans>
```

<form-bean> 的 <form-property> 子元素用来设置动态 ActionForm 的属性。<form-property>元素的 name 属性指定属性名，type 指定属性类型。可以把动态 ActionForm 的属性设为以下 Java 类型：

- java.lang.BigDecimal
- java.lang.BigInteger
- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Class
- java.lang.Double
- java.lang.Float
- java.lang.Integer
- java.lang.Long

- java.lang.Short
- java.lang.String
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

如果表单的字段值为 Java 基本类型，在配置时应该用相应的包装类型来代替，例如 int 类型的包装类型为 Integer：

```
<form-property
        initial="0"
        name="age"
        type="java.lang.Integer"/>
```

7.4.2 动态 ActionForm 的 reset()方法

DynaActionForm 基类提供了 initialize()方法，它把表单的所有属性恢复为默认值。表单属性默认值由<form-bean>的<form-property>子元素的 initial 属性来决定。如果没有设置 initial 属性，则表单属性的默认值由其 Java 类型来自动决定，例如对象类型的默认值为 null，整数类型的默认值为 0，boolean 类型的默认值为 false。

DynaActionForm 基类的 initialize()方法的代码如下：

```
public void initialize(ActionMapping mapping) {
        String name = mapping.getName();
        if (name == null) {
            return;
        }
        FormBeanConfig config =
            mapping.getModuleConfig().findFormBeanConfig(name);
        if (config == null) {
            return;
        }
        FormPropertyConfig props[] = config.findFormPropertyConfigs();
        for (int i = 0; i < props.length; i++) {
            set(props[i].getName(), props[i].initial());
        }
    }
```

DynaActionForm 基类的 reset()方法不执行任何操作，其代码如下：

```
public void reset(ActionMapping mapping, HttpServletRequest request) {
        ;           // Default implementation does nothing
}
```

如果希望 Struts 框架在每次把表单数据组装到动态 ActionForm 中之前，先把所有的属性恢复为默认值，可以定义一个扩展 DynaActionForm 类的子类，然后覆盖其 reset()方法，在 reset()方法中只要调用 initialize()方法即可，代码如下：

```
public class MyDynaActionForm extends DynaActionForm{
    ......
    public void reset(ActionMapping mapping, HttpServletRequest request) {
            initialize(mapping);
    }
}
```

7.4.3　访问动态 ActionForm

Action 类和 JSP 都可以访问动态 ActionForm，使用方法与标准 ActionForm 大致相同，只有一点小差别。如果使用标准 ActionForm 对象，在标准 ActionForm 中针对每个属性都提供了 get/set 方法，来读取或设置属性。例如对于 email 属性，应该提供了 getEmail()和 setEmail()方法。

而 DynaActionForm 把所有的属性保存在一个 Map 类对象中，并提供了下面的用于访问所有属性的通用方法：

```
public Object get(String name)
public void set(String name, Object value)
```

get(String name)方法根据指定的属性名返回属性值；set(String name, Object value)方法用于为给定的属性赋值。例如，如果要访问 DynaActionForm 类中的 email 属性，可以采用以下代码：

```
//get email
String email = (String)form.get("email");
//set email
form.set("email","linda@yahoo.com");
```

7.4.4　动态 ActionForm 的表单验证

DynaActionForm 基类的 validate()方法没有提供任何默认的验证行为。可以定义扩展 DynaActionForm 的子类，然后覆盖 validate()方法，但是以编程的方式来验证动态 ActionForm 违背了 Struts 框架提供动态 ActionForm 的初衷，即以配置来替代编程。幸运的是，可以采用另一种验证机制，即 Validator 框架来完成验证。Validator 框架允许采用特定的配置文件来为动态 ActionForm 配置验证规则。关于 Validator 框架的使用方法可以参考第 10 章的内容（Validator 验证框架）。

7.4.5　在 netstore 应用中使用动态 ActionForm

在 netstore 应用中定义了名为“itemDetailForm”的动态 ActionForm，它包含一个 ItemDetailView 类型的属性 view，以下是它的配置代码：

```
<form-bean
        name="itemDetailForm"
```

```
        dynamic="true"
        type="org.apache.struts.action.DynaActionForm">
            <form-property name="view" type="netstore.catalog.view.ItemDetailView"/>
    </form-bean>
```

在本章 7.1 节的如图 7-1 所示的 netstore 应用主页上，如果选择了某个商品，该请求将转发给 GetItemDetailAction 来处理。GetItemDetailAction 的配置代码如下：

```
<action
    path="/viewitemdetail"
    name="itemDetailForm"
    input="/index.jsp"
    type="netstore.catalog.GetItemDetailAction"
    scope="request"
    validate="false">
    <exception
        key="global.error.invalidlogin"
        path="/index.jsp"
        scope="request"
        type="netstore.framework.exceptions.DatastoreException"/>

    <forward name="Success" path="/catalog/itemdetail.jsp"/>
</action>
```

GetItemDetailAction 根据用户选择的商品 ID 调用模型的业务方法，并检索出该商品的详细信息，把它存放在 ItemDetailView 对象中，再把 ItemDetailView 对象赋值给动态 itemDetailForm 的 view 属性。GetItemDetailAction 的 execute()方法的代码如下：

```
public ActionForward execute( ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response )
    throws Exception {
    // Get the primary key of the item from the request
    String itemId = request.getParameter( IConstants.ID_KEY );

    // Call the netstore service and ask it for an ItemView for the item
    INetstoreService serviceImpl = getNetstoreService();

    ItemDetailView itemDetailView = serviceImpl.getItemDetailView( itemId );

    // Set the returned ItemView into the Dynamic Action Form
    // The parameter name 'view' is what is defined in the struts-config
    ((DynaActionForm)form).set("view", itemDetailView);

    // Return the ActionForward that is defined for the success condition
```

```
    return mapping.findForward( IConstants.SUCCESS_KEY );
}
```

GetItemDetailAction 最后把请求转发给 itemdetail.jsp，itemdetail.jsp 的网页参见本章 7.1 节的图 7-2，它通过<bean:write>标签把动态 itemDetailForm 的 view 属性包含的数据输出到网页上，相关代码如下：

```
<!-- PRODUCT NAME -->
<bean:write name="itemDetailForm" property="view.name"/>
<!-- PRODUCT MODEL -->
<bean:write name="itemDetailForm" property="view.modelNumber"/>
<!-- PRODUCT DESCRIPTION -->
<bean:write name="itemDetailForm" property="view.description"/>
<!-- PRODUCT PRICE -->
$<bean:write name="itemDetailForm" format="#,##0.00" property="view.unitPrice"/>
<!-- PRODUCT Feature -->
<bean:write name="itemDetailForm" property="view.productFeature"/>
```

关于<bean:write>标签的用法请参见本书的 13.3.3 小节（<bean:write>标签）。

7.5　小　　结

本章侧重介绍了构成 Struts 视图的组件之一：ActionForm。ActionForm 用于在视图层和控制层之间传递表单数据，ActionForm 可以存放在 request 和 session 范围内。ActionForm 是一种 Web 组件，不应该在模型层直接访问 ActionForm Bean。同一个 ActionForm 可以对应多个 HTML 表单，在这种情况下，有以下开发技巧：

- 在 HTML 表单中定义<html:hidden property="page"/>隐藏字段，来标识当前页面。
- 在 ActionForm 中定义 page 属性，它和表单中的隐藏字段 page 对应。
- 在 ActionForm 的 reset()方法中，只能把和当前表单相关的属性恢复为默认值。可以调用 request.getParameter("page")方法来读取当前的页面编号。
- 在 ActionForm 的 validate()方法中，只能对和当前表单相关的属性进行验证。此时 page 属性代表当前的页面编号。
- 在配置 ActionForm 和 Action 的映射时，应该把 ActionForm 的范围设为 session。

Struts 框架还引入了 DynaActionForm 类，它允许以配置的方式来创建动态 ActionForm，使用 DynaActionForm 有以下几点需要注意：

- <form> 的 <form-property> 子元素用于配置动态 ActionForm 的属性。<form-property>元素的 type 属性指定 ActionForm 的属性的类型。如果属性为 Java 基本类型，应该把属性设置为相应的 Java 包装类型。
- 提倡使用 Validator 框架来验证动态 ActionForm，这样可以避免以编程的方式来实现 validate()方法。

- 在一般情况下，可以直接使用 DynaActionForm 基类，没有必要创建扩展 DynaActionForm 的子类。但是，DynaActionForm 基类的 reset()方法没有执行任何操作，如果需要提供属性的复位功能，可以创建扩展 DynaActionForm 类的子类，然后覆盖 reset()方法，在 reset()方法中只要调用 DynaActionForm 的 initialize()方法即可。
- 如果要访问 DynaActionForm 的属性，应该调用以下方法：

```
public Object get(String name)
public void set(String name, Object value)
```

以上方法的 name 参数代表属性名。

第 8 章　扩展 Struts 框架

Struts 框架的一大优势在于它允许开发人员根据实际需要来扩展框架，定制客户化的功能。可以把框架比做房屋的基本结构，用户可以在基本结构的基础上对房屋进行个性化的装饰，比如选择所喜欢的墙纸、涂料颜色等。当然，用户也可以选择默认的装饰方案。

一个好的软件框架也应该具备可扩展特性。在 Struts 框架中提供了许多可扩展之处，不妨将其称为扩展点（Extension Point）。以下是 Struts 的扩展点：

- 一般性扩展点：Struts 插件（PlugIn）、扩展 Struts 配置类。
- 控制器的扩展点：扩展 ActionServlet 类、RequestProcessor 类和 Action 类。
- 视图的扩展点：扩展 Struts 客户化标签。
- 模型的扩展点：扩展 SessionContainer 类和 ApplicationContainer 类。

本章提到的有些知识，如扩展 ActionServlet 类和 Action 类，在本书的其他地方也曾提及。本章侧重于归纳 Struts 框架的扩展点，讲解了每一种扩展点的扩展机制以及扩展方法。

8.1　Struts 插件（PlugIn）

Struts1.1 框架提供了动态插入和加载组件的功能，这种组件被称为 Struts 插件。Struts 插件实际上就是一个 Java 类，它在 Struts 应用启动时被初始化，在应用关闭时被销毁。任何作为插件的 Java 类都应该实现 org.apache.struts.action.PlugIn 接口。PlugIn 接口包括两个方法，参见例程 8-1。

例程 8-1　org.apache.struts.action.PlugIn 接口

```
public interface PlugIn {
 /**
  * Notification that the specified application module is being started.
  */
  public void init(ActionServlet servlet, ApplicationConfig config)
    throws ServletException;

 /**
  * Notification that the application module is being shut down.
  */
  public void destroy();
}
```

一个 Struts 应用可以包含一个或多个插件。在 Struts 应用启动时，Struts 框架调用每个插件类的 init()方法进行初始化。在插件的初始化阶段，可以完成一些初始化操作，如建立

数据库连接，或者和远程系统的连接等。

当应用被关闭时，Struts 框架就会调用每个插件类的 destroy()方法。destroy()方法可以用来完成释放资源的任务，如关闭数据库连接或远程系统连接等。

以下是一个使用 Struts 插件的例子。在本书第 3 章（Struts 应用的需求分析与设计）介绍的 addressbook 应用中，Web 容器在启动时会加载并初始化 UserDatabaseServlet 类，UserDatabaseServlet 类在初始化时会加载 WEB-INF/userdatabase.xml 文件中的数据。现在采用 UserDatabasePlugIn 来实现同样的功能。修改后的 addressbook 应用的所有源文件位于本书配套光盘的 sourcecode/addressbook/version2/addressbook 目录下。

例程 8-2 是 UserDatabasePlugIn 的源程序。

例程 8-2 UserDatabasePlugIn.java

```
package addressbook;
//import all the necessary packages here
.....
public final class UserDatabasePlugIn implements PlugIn{

    private Hashtable database = null;
    private Log log = LogFactory.getLog(this.getClass().getName());

    private String pathname;
    public String getPathname() {
        return pathname;
    }
    public void setPathname(String pathname) {
        this.pathname = pathname;
    }

    public void destroy() {
         database = null;
    }

  public void addUser(UserBean user){
        database.put(user.getUserName(),user);
  }
  public void init(ActionServlet servlet, ModuleConfig config)
   throws ServletException{
         try {
               /* Try and load the database.xml file. If we have loaded,
                * store the contents into our context so that we can use it
                * to compare against for user logons.
                */
               load(servlet);
               servlet.getServletContext().setAttribute(Constants.DATABASE_KEY,
                                        database);
```

```
            } catch (Exception e) {
                log.debug("Database load exception", e);
                throw new UnavailableException
                 ("Cannot load database from '" + pathname + "':"+e.getMessage());
            }
        }
        /**
         * Load the current database.xml file. Using the <code>Digester<code>
         * @see org.apache.commons.digester.Digester
         */
        private synchronized void load(ActionServlet servlet) throws Exception {
            database = new Hashtable();
            log.debug("Loading database from '" + pathname + "'");
            InputStream is = servlet.getServletContext().getResourceAsStream(pathname);
            if (is == null) {
               log.debug("No such resource available - loading empty database");
               return;
            }
            BufferedInputStream bis = new BufferedInputStream(is);
            Digester digester = new Digester();
            digester.push(this);
            digester.setDebug(2);
            digester.setValidating(false);
            digester.addObjectCreate("database/user", "addressbook.model.UserBean");
            digester.addSetProperties("database/user");
            digester.addSetNext("database/user", "addUser");

            digester.parse(bis);
            bis.close();
        }
    }
```

当 Struts 框架调用 UserDatabasePlugin 类的 init()方法时，会把 ActionServlet 作为参数传给 init()方法。因此在 init()方法中可以调用 ActionServlet 的 getServletContext()方法来获得 ServletContext 对象的引用。

UserDatabasePlugin 类的 init()方法负责创建一个 database 实例，并把它保存在 ServletContext 对象中，这样，database 实例就可以被整个应用共享：

```
servlet.getServletContext().setAttribute(Constants.DATABASE_KEY, database);
```

除了创建以上插件类外，还需要在 Struts 配置文件中配置插件，Struts 框架在启动时将根据相关的配置信息来初始化插件。与插件对应的配置元素为<plug-in>，在本书的 4.3.10 小节（<plug-in>元素）中详细介绍了这个元素的一系列属性。配置 UserDatabasePlugin 的代码如下：

```
<plug-in   className="addressbook.UserDatabasePlugIn">
    <set-property
```

```
        property="pathname"
        value="/WEB-INF/userdatabase.xml"/>
</plug-in>
```

以上<plug-in>元素包含一个<set-property>子元素，它定义了插件的 pathname 属性，与之对应，在 UserDatabasePlugIn 类中定义了 pathname 成员变量以及 get/set 方法：

```
private String pathname;
public String getPathname() {
    return pathname;
}
public void setPathname(String pathname) {
    this.pathname = pathname;
}
```

Struts 框架在加载插件时，会调用插件类的 setPathname()方法，把<set-property>子元素设置的 pathname 属性值传给 UserDatabasePlugIn 实例的 pathname 成员变量。

根据 Struts 配置文件的 DTD 定义，在 Struts 配置文件中，<plug-in>元素必须位于其他配置元素的后面。此外，如果在配置文件中配置了多个插件，Struts 框架将按照它们在配置文件中的先后顺序来依次初始化它们。

8.2 扩展 Struts 的配置类

在 Struts 应用启动时，Struts 配置文件中的所有信息都会被读到内存中，这些信息存放在 org.apache.struts.config 包的相应配置类的实例中。

根据第 4 章（配置 Struts 应用）对各种配置元素的介绍，会发现多数配置元素都有一个 className 属性，这个属性用来设置和配置元素对应的配置类。每个配置元素都有默认配置类，Struts 框架允许对这些默认配置类进行扩展。

例如，<action>元素默认的配置类为 ActionMapping。下面定义一个客户化<action>元素，它有一个客户化属性 sslRequired，用来指定是否采用 HTTPS 协议，可选值包括 true 和 false。这个客户化<action>元素的用法如下：

```
<action     className="CustomActionMapping"
            sslRequired="true"
            path="/hello"
            type="HelloAction"
            ......
>
</action>
```

与以上客户化<action>元素对应的配置类为 CustomActionMapping，这个类应该扩展 ActionMapping 类，并且提供 sslRequired 成员变量以及 get/set 方法：

```
public class CustomActionMapping extends ActionMapping {
```

```
    /**
     * Should use HTTPS or HTTP
     */
    protected boolean sslRequired = false;

    public boolean getSslRequired() {
        return (this.sslRequired);
    }

    public void setSslRequired (boolean sslRequired) {
        if (configured) {
            throw new IllegalStateException("Configuration is frozen");
        }
        this. sslRequired = sslRequired;
    }
    ......
}
```

8.3　控制器扩展点

Struts 框架在控制器中提供了许多可扩展之处，允许扩展 ActionServlet、RequestProcessor 和 Action 类，来实现各种客户化功能。

8.3.1　扩展 ActionServlet 类

在 Struts 1.1 以前的版本中，Struts 应用通常都需要扩展 ActionServlet 类，来实现各种定制的控制功能。在 Struts 1.1 中，扩展 ActionServlet 类不再是必需的。不过在某些情况下，根据实际情况需要，可不妨扩展 ActionServlet 类。

在本书的第 5 章（Struts 控制器组件）中讲过，当 Struts 应用启动时会加载 ActionServlet 类并调用它的 init()方法，来对 Struts 框架进行初始化。如果需要修改 Struts 框架的初始化行为，可以创建一个 org.apache.struts.action.ActionServlet 类的子类，然后覆盖它的 init()方法。

接下来，应该在 web.xml 文件中对自定义的 ActionServlet 进行配置。例如，在 netstore 应用中创建了 ExtendedActionServlet 类，以下是它的配置代码：

```
<servlet>
  <servlet-name>netstore</servlet-name>
  <servlet-class>
    netstore.framework.ExtendedActionServlet
  </servlet-class>
</servlet>
```

在 Struts 1.1 中，预处理 HTTP 请求的具体操作由 RequestProcessor 类来完成。因此，

如果需要对预处理 HTTP 请求的方式进行客户化，可以扩展 RequestProcessor 类，这将在接下来的一节中讨论。

8.3.2 扩展 RequestProcessor 类

如果扩展了 RequestProcessor 类，应该在 Struts 配置文件中通过<controller>元素对自定义的 RequestProcessor 类进行配置。例如：

```
<controller
  contentType="text/html;charset=GB2312"
  locale="true"
  nocache="true"
  processorClass="netstore.framework.CustomRequestProcessor"/>
```

<controller>元素的 processorClass 属性用来指定使用的 RequestProcessor 类。Struts 框架在启动时会创建这个 RequestProcessor 类的实例，并利用它来处理所有的 HTTP 请求。由于每个子应用模块都有各自的配置文件，因此可以为每个子应用模块配置不同的 RequestProcessor 类。

RequestProcessor 类的一个扩展点为 processPreprocess()方法。在 RequestProcessor 基类中，该方法不执行任何操作，直接返回 true。它的代码如下：

```
protected boolean processPreprocess( HttpServletRequest request,
                                     HttpServletResponse response ){
  return (true);
}
```

RequestProcessor 类在 process()方法中处理请求，以下是 process()方法调用 processPreprocess()方法的相关程序代码：

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

        .....
        // General purpose preprocessing hook
        if (!processPreprocess(request, response)) {
            return;
        }
        .....
        ActionForm form = processActionForm(request, response, mapping);
        processPopulate(request, response, form, mapping);
        if (!processValidate(request, response, form, mapping)) {
            return;
        }
         .....
        // Create or acquire the Action instance to process this request
```

```
        Action action = processActionCreate(request, response, mapping);
        if (action == null) {
            return;
        }
        // Call the Action instance itself
        ActionForward forward =
            processActionPerform(request, response,
                                 action, form, mapping);
        ......
    }
```

从以上的程序代码中可以看出，process()方法在调用 Action 的 execute()方法之前，就会调用 processPreprocess()方法。如果 processPreprocess()方法返回 true，则表示继续按正常的流程处理请求。

在自定义的 RequestProcessor 类中，可以覆盖 processPreprocess()方法来执行特定的逻辑。如果在某些条件下希望终止处理请求，只需让 processPreprocess()返回 false 即可。在这种情况下，仍需要以编程的方式来决定如何转发或重定向请求。

例如，如果希望 processPreprocess()方法实现这样一段逻辑：如果 HTTP 请求来自本机，就按正常的流程来处理请求，否则，将把请求重定向到错误页面，显示无权访问的错误消息。以下是实现这段逻辑的 processPreprocess()方法的代码：

```
protected boolean processPreprocess( HttpServletRequest request,
                                     HttpServletResponse response){

    boolean continueProcessing = true;

    // Get the name of the remote host and log it
    String remoteHost = request.getRemoteHost();
    log.info( "Request from host: " + remoteHost );

    // Make sure the host is from one that you expect
    if ( remoteHost == null || !remoteHost.startsWith( "127.") ){
        // Not the localhost, so don't allow the host to access the site
        continueProcessing = false;
        ForwardConfig config = moduleConfig.findForwardConfig("Unauthorized");
        try{
          response.sendRedirect( config.getPath() );
        }catch( Exception ex ){
          log.error( "Problem sending redirect from processPreprocess()" );
        }
    }
    return continueProcessing;
}
```

以上代码中值得注意的地方是，如果该方法返回 false，那么应该在方法内部决定如何转发请求。

Servlet 2.3 API 提供了 Servlet 过滤器，它可以实现和 processPreprocess()方法相同的功能。Serlvet 过滤器也可以用来执行客户化的预处理请求操作，Web 容器先调用 Servlet 过滤器，再把请求转发给 Struts 控制器。

与扩展 RequestProcessor 类相比，通过 Serlvet 过滤器来预处理请求有两大弱点：

- 首先，Servlet 过滤器是在 Servlet 2.3 API 中才出现的，如果使用的是 Servlet 2.2，就无法使用 Servlet 过滤器。
- 其次，由于 Servlet 过滤器在请求到达 Struts 控制器之前就被调用，因此 Servlet 过滤器访问 Struts API 是不可取的。例如，在 processPreprocess()方法中可以访问 ActionForward 对象，而在 Servlet 过滤器中查找 ActionForward 对象会导致错误，因为此时 ActionForward 对象有可能还不存在。

8.3.3 扩展 Action 类

Struts 框架的 Action 类是最频繁的扩展点。对于具体的 Struts 应用，可以先为应用创建一个扩展 Struts Action 类的 Action 基类，在这个 Action 基类中定义应用中所有 Action 的一些公共逻辑，它可以作为其他 Action 的父类。这种处理方法可以提高代码的可重用性，减少代码的重复。例程 8-3 为 netstore 应用的 Action 基类。

例程 8-3 NetstoreBaseAction.java

```
package netstore.framework;

//import all the necessary packages here
.....

/**
 * An abstract Action class that all Netstore action classes should
 * extend.
 */
abstract public class NetstoreBaseAction extends Action {
  Log log = LogFactory.getLog( this.getClass() );

  protected INetstoreService getNetstoreService(){
    INetstoreServiceFactory factory =
    (INetstoreServiceFactory)getApplicationObject( IConstants.SERVICE_FACTORY_KEY );
    INetstoreService service = null;

    try{
      service = factory.createService();
    }catch( Exception ex ){
      log.error( "Problem creating the Netstore Service", ex );
    }
    return service;
  }
```

```
/**
 * Retrieve a session object based on the request and the attribute name.
 */
protected Object getSessionObject(HttpServletRequest req,
                                  String attrName) {
  Object sessionObj = null;
  HttpSession session = req.getSession(false);
  if ( session != null ){
      sessionObj = session.getAttribute(attrName);
  }
  return sessionObj;
}

/**
 * Return the instance of the ApplicationContainer object.
 */
protected ApplicationContainer getApplicationContainer() {
  return
  (ApplicationContainer)getApplicationObject(IConstants.APPLICATION_CONTAINER_KEY);
}

/**
 * Retrieve the SessionContainer for the user tier to the request.
 */
protected SessionContainer getSessionContainer(HttpServletRequest request) {

  SessionContainer sessionContainer = (SessionContainer)getSessionObject(request,
  IConstants.SESSION_CONTAINER_KEY);

  // Create a SessionContainer for the user if it doesn't exist already
  if(sessionContainer == null) {
     sessionContainer = new SessionContainer();
     sessionContainer.setLocale(request.getLocale());
     HttpSession session = request.getSession(true);
     session.setAttribute(IConstants.SESSION_CONTAINER_KEY, sessionContainer);
  }

  return sessionContainer;
}

/**
 * Retrieve an object from the application scope by its name. This is
 * a convience method.
 */
protected Object getApplicationObject(String attrName) {
```

```
        return servlet.getServletContext().getAttribute(attrName);
    }

    public boolean isLoggedIn( HttpServletRequest request ){
        SessionContainer container = getSessionContainer(request);
        if ( container.getUserView() != null ){
            return true;
        }else{
            return false;
        }
    }
}
```

在 NetstoreBaseAction 类中定义了应用中所有 Action 的公共行为，例如判断用户是否登入，以及获取当前用户的 SessionContainer 对象，在应用的其他 Action 子类中不必重复实现这些逻辑。

8.4 扩展视图组件

一般说来，没有必要扩展视图组件，因为不同的应用有不同的外观和界面，一个应用的 JSP 页面不大可能适用于另一个不同的应用。不过，可以扩展 Struts 客户化标签，因为标签处理器为常规的 Java 类，可以通过定义子类的方式来扩展它们，这些扩展后的客户化标签能够被不同的应用重用。

Struts HTML 标签库中的标签对视图内容的影响最大。因此可以扩展这些标签来创建客户化的应用外观。当扩展了标签后，应该定义存放这些标签的标签库。尽管可以把自定义的标签加入到标准的 Struts 标签库中，但是这会使将应用升级到新的 Struts 版本变得更加麻烦。所以建议定义单独的标签库，来存放和特定应用相关的客户化标签。

一旦为客户化标签库创建了 TLD 文件，并且在 web.xml 中注册了标签库，就可以在 JSP 文件中方便地使用这些标签了。

8.5 扩展模型组件

Struts 框架本身没有在模型层提供现成的模型组件，因此扩展模型组件不属于 Struts 技术。本节以 netstore 应用为例，讲述了自定义的两个用于存放模型状态信息的模型组件。在实际应用中，可以根据需要对这两个模型组件进行扩展。

这两个组件为 SessionContainer 和 ApplicationContainer 类。SessionContainer 类可以取代 HttpSession，用于存取 HTTP 会话范围内的用户信息；ApplicationContainer 类可以取代 ServletContext，用于存取应用范围内的信息。

我们知道，在 HttpSession 中以属性的方式保存或读取数据时，必须提供属性 key。例如：

```
public void setAttribute( "userViewKey", userView );
public Object getAttribute( "userViewKey" );
```

所以在使用 HttpSession 来存放或读取某种属性时，必须先知道属性 key。有的程序员喜欢使用更加客户化的接口，例如：

```
sessionContainer.setUserView( userView);
sessionContainer.getUserView();
```

采用以上方式，程序员不必关心属性 key 是什么，也不必知道属性到底存放在什么地方。

SessionContainer 的实例应该存放在 session 范围内。与 HttpSession 相比，SessionContainer 提供了更加客户化的存取 session 范围内共享数据的方法。图 8-1 和图 8-2 比较了这两种方式的区别。

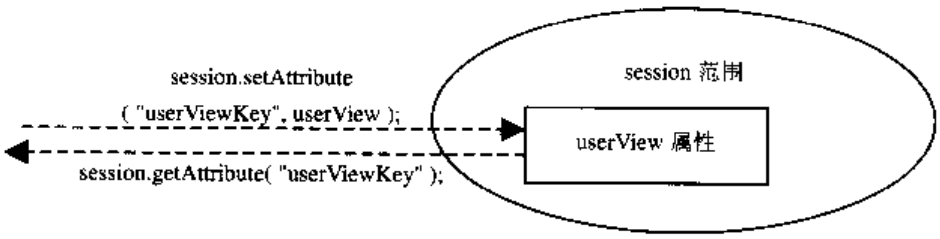

图 8-1　通过 HttpSession 来存取 session 范围内的共享数据

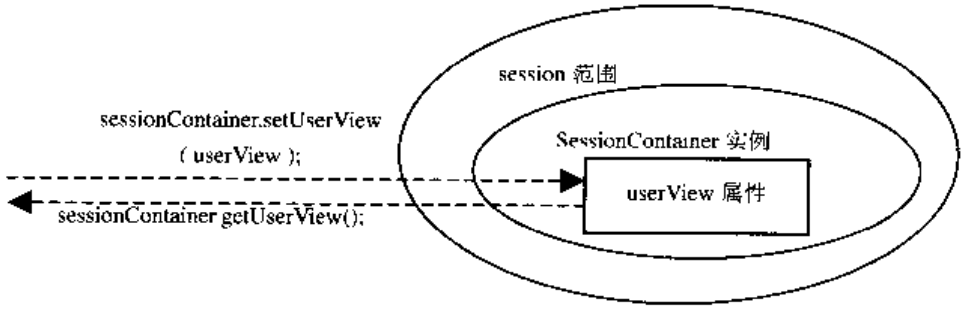

图 8-2　通过 SessionContainer 类来存取 session 范围内的共享数据

SessionContainer 类的实现并不复杂，它是一个普通的 JavaBean，包含一些属性以及相应的 get/set 方法。例程 8-4 为 SessionContainer 类的源程序。

例程 8-4　SessionContainer 类

```
package netstore.framework;
import java.util.Locale;
import javax.servlet.http.HttpSessionBindingListener;
import javax.servlet.http.HttpSessionBindingEvent;
import netstore.customer.view.UserView;

/**
 * Used to store information about a specific user. This class is used
 * so that the information is not scattered throughout the HttpSession.
 * Only this object is stored in the session for the user. This class
 * implements the HttpSessionBindingListener interface so that it can
```

```
 * be notified of session timeout and perform the proper cleanup.
 */
public class SessionContainer implements HttpSessionBindingListener {

  // The user's shopping cart
  private ShoppingCart cart = null;
  //   Data about the user that is cached
  private UserView userView = null;

  /**
   * The Locale object for the user. Although Struts stores a Locale for
   * each user in the session, the locale is also maintained here.
   */
  private Locale locale;

  public SessionContainer() {
     super();
     initialize();
  }

  public ShoppingCart getCart() {
     return cart;
  }

  public void setCart(ShoppingCart newCart) {
     cart = newCart;
  }

  public void setLocale(Locale aLocale) {
     locale = aLocale;
  }

  public Locale getLocale() {
     return locale;
  }

  /**
   * The container calls this method when it is being unbound from the
   * session.
   */
  public void valueUnbound(HttpSessionBindingEvent event) {
     // Perform resource cleanup
     cleanUp();
  }

  /**
```

```
     * The container calls this method when it is being bound to the
     * session.
     */
    public void valueBound(HttpSessionBindingEvent event) {
        // Don't need to do anything, but still have to implement the
        // interface method.
    }

    public UserView getUserView() {
        return userView;
    }

    public void setUserView(UserView newView) {
        userView = newView;
    }

    /**
     * Initialize all of the required resources
     */
    private void initialize() {
        // Create a new shopping cart for this user
        cart = new ShoppingCart();
    }

    /**
     * Clean up any open resources. The shopping cart is left intact
     * intentionally.
     */
    public void cleanUp() {
        setUserView( null );
    }
}
```

每当有新的用户登入应用时，netstore 应用就会为这个用户创建一个 SessionContainer 实例。SessionContainer 实例被保存在 HttpSession 对象中。SessionContainer 类实现了 HttpSessionBindingListener 接口（简称为监听器），在这个接口中声明了两个方法：

- valueBound()：当 SessionContainer 和 HttpSession 对象绑定的时候（即调用 HttpSession 的 setAttribute()方法把 SessionContainer 对象保存在 session 范围内的时候），Web 容器会自动调用 SessionContainer 的 valueBound()方法。
- valueUnbound()：当 SessionContainer 和 HttpSession 对象解除绑定的时候（即调用 HttpSession 对象的 removeAttribute()方法把 SessionContainer 对象从 session 范围内删除，或 Session 过期的时候），Web 容器会自动调用 SessionContainer 的 valueUnbound()方法。

在 netstoreBaseAction 类中定义了 getSessionContainer()方法，它用于返回当前 session

范围内的 SessionContainer 对象，如果该对象不存在，就先创建它，把它保存在 session 范围内：

```
/**
 * Retrieve the SessionContainer for the user tier to the request.
 */
protected SessionContainer getSessionContainer(HttpServletRequest request) {
  SessionContainer sessionContainer = (SessionContainer)getSessionObject(request,
  IConstants.SESSION_CONTAINER_KEY);

  // Create a SessionContainer for the user if it doesn't exist already
  if(sessionContainer == null) {
    sessionContainer = new SessionContainer();
    sessionContainer.setLocale(request.getLocale());
    HttpSession session = request.getSession(true);
    session.setAttribute(IConstants.SESSION_CONTAINER_KEY, sessionContainer);
  }
  return sessionContainer;
}
```

ApplicationContainer 用来存放 application 范围内的数据。在应用启动的时候创建这个类的实例，并在应用终止的时候销毁它。在实际应用中，可以进一步扩展 ApplicationContainer 类，用它来存放与特定应用相关的、被应用中所有 Web 组件共享的信息。

8.6 小　　结

本章归纳了 Struts 框架的所有可扩展点。Struts 框架的可扩展性使开发者可以方便地定制客户化功能，提高应用的灵活性和对各种需求的可适应性。然而，实现更多的功能是要花费更大代价的，应该避免滥用 Struts 的可扩展特性。Struts 由核心包加上很多工具包构成，它们已经提供了很多现成的功能。因此不要盲目地扩展 Struts 框架，在决定编写扩展代码前，务必先确认 Struts 没有提供现成的您需要的功能。否则，重复的功能会导致应用结构混乱，将来还得花费额外的精力来清除重复功能。

此外，必须考虑扩展后的框架是否会和将来的新的 Struts 版本兼容。从 Struts 1.0 到 Struts 1.1 就发生了很大的改动。例如，在 Struts 1.0 中，Action 的执行业务流程控制的方法为 perform()，在 Struts 1.1 中，该方法被废弃（deprecated），而为 execute()方法所取代。在现有的 Struts 版本的 API 中，如果有的类的方法已经声明将要被废弃，应该尽量不要覆盖这些方法，否则，当采用新的 Struts 版本时，就不得不对应用做相应的升级。

第 9 章　Struts 应用的国际化

随着全球经济的一体化成为一种主流趋势，如今的企业已经不再仅仅着眼于本国市场，而是开始致力于在全球范围内为它们的产品寻找客户。万维网（World Wide Web）的迅猛发展推动了跨国业务的发展，它成为一种在全世界范围内发布产品信息、吸引客户的有效手段。为了使企业 Web 应用能支持全球客户，软件开发者应该开发出支持多国语言、国际化的 Web 应用。

本章首先介绍了软件的本地化与国际化的概念和区别，然后讨论了 Struts 框架提供的实现 Web 应用国际化的机制，最后以 helloapp 应用为例，介绍了对 Struts 应用实现国际化的步骤。本章提供的 helloapp 应用的源文件位于配套光盘的 sourcecode/helloapp/version2 目录下。

9.1　本地化与国际化的概念

在过去，软件开发者在开发应用程序时，将注意力集中于实现具体的业务逻辑。软件面向的用户群是固定的，软件只需要支持一种语言。如今，随着跨国业务的迅猛发展，需要同一个软件能同时支持多种语言和国家。

对于已经开发好的软件，比如某种英文软件，如果要使其支持新的语言，比如中文，必须对该软件进行中文本地化。对软件的本地化涉及修改原有程序的代码、数据和配置文件等，需要在原有软件的基础上开发出新的软件版本。

国际化（简称为 I18N）指的是在软件设计阶段，就应该使软件具有支持多种语言和地区的功能。这样，当需要在应用中添加对一种新的语言和国家的支持时，不需要对已有的软件返工，无需修改应用的程序代码。

软件的本地化与国际化的区别如图 9-1 所示。简单地说，本地化意味着针对不同语言的客户，开发出不同的软件版本；国际化意味着同一个软件可以面向使用各种不同语言的客户。

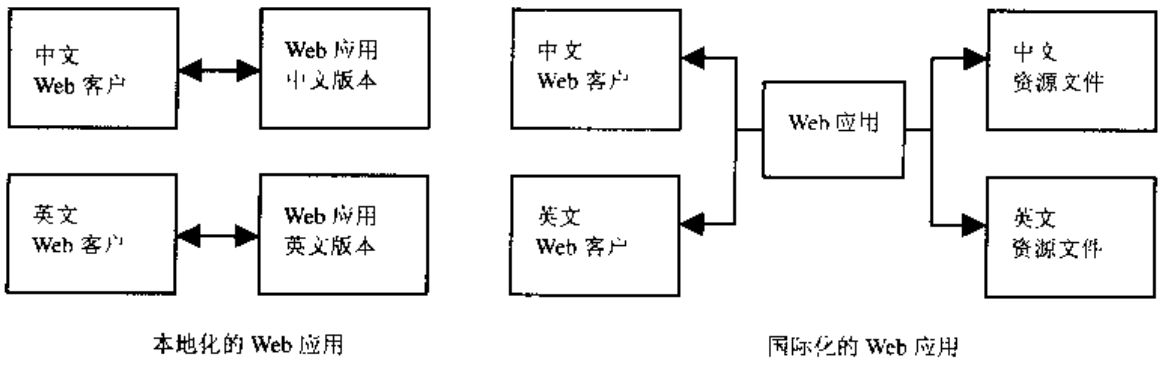

图 9-1　软件的本地化与国际化的区别

如果一个应用支持国际化，它应该具备以下特征：

- 当应用需要支持一种新的语言时，无需修改应用程序代码。
- 文本、消息和图片从源程序代码中抽取出来，存储在外部。
- 应该根据用户的语言和地理位置，对与特定文化相关的数据，如日期、时间和货币，进行正确的格式化。
- 支持非标准的字符集。
- 可以方便快捷地对应用做出调整，使它适应新的语言和地区。

I18N 是 Internationalization 的简称，因为该单词的首字母 I 与尾字母 N 中间隔着 18 个字符，由此得名。

在对一个 Web 应用进行国际化时，除了应该对网站上的文本、图片和按钮进行国际化外，还应该对数字和货币等根据不同国家的标准进行相应的格式化，这样才能保证各个国家的用户都能顺利地读懂这些数据。

Locale（本地）指的是一个具有相同风俗、文化和语言的区域。如果一个应用没有事先把 I18N 作为内嵌的功能，那么当这个应用需要支持新的 Locale 时，开发人员必须对嵌入在源代码中的文本、图片和消息进行修改，然后重新编译源代码。每当这个应用需要支持新的 Locale 时，就必须重复这些繁琐的步骤，这种做法显然大大降低了软件开发效率。

9.2 Web 应用的中文本地化

无论是对 Web 应用的本地化还是国际化，都会涉及字符编码转换问题。Web 应用的各种可能的输入和输出流如图 9-2 所示，当数据流的源与目的地使用不同的字符编码时，就需要对字符编码进行正确的转换。

本节以对 Web 应用进行中文本地化为例，来讲解如何处理 Web 应用各种输入和输出流的字符编码转换，简体中文的字符编码为“GB2312”。

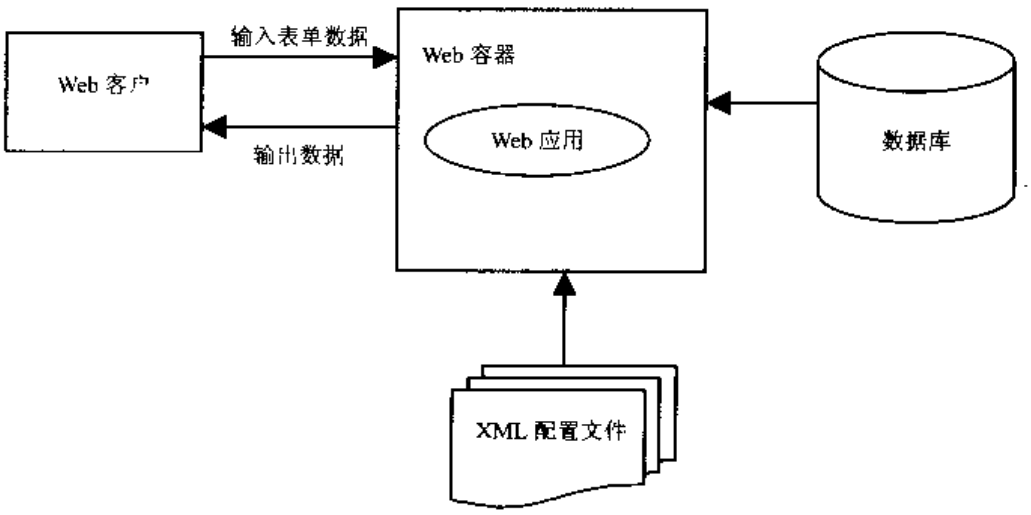

图 9-2 Web 应用的输入流和输出流

9.2.1　处理 HTTP 请求数据编码

默认情况下，IE 浏览器发送请求时采用“ISO-8859-1”字符编码，如果 Web 应用程序要正确地读取用户发送的中文数据，则需要进行编码转换。

一种方法是在处理请求前，先设置 HttpServletRequest 对象的字符编码：

```
request.setCharacterEncoding("gb2312");
```

还有一种办法是对用户输入的请求数据进行编码转换：

```
String clientData=request.getParameter("clientData");
if(clientData!=null)
    clientData=new String(clientData.getBytes("ISO-8859-1"),"GB2312");
```

9.2.2　处理数据库数据编码

如果数据库系统的字符编码为“GB2312”，那么可以直接读取数据库中的中文数据，而无需进行编码转换。如果数据库字符编码为“ISO-8859-1”，那么必须先对来自数据库的数据进行编码转换，然后才能使用。例如：

```
Connection con= DbUtil.connectToDb();
PreparedStatement pStmt=null;
ResultSet rs=null;
pStmt=con.prepareStatement("select FIELD1 from MYTABLE");
rs=pStmt.executeQuery();
while(rs.next()){
    String field1=rs.getString(" FIELD1");
    String field1_ch= new String(field1.getBytes("ISO-8859-1"),"GB2312");
    //process data
}
```

9.2.3　处理 XML 配置文件编码

如果在 XML 文件中包含中文，可以将 XML 文件的字符编码设为“GB2312”。例如 netstore 应用的 database.xml 文件就采用了中文编码。这样，当 Java 程序加载和解析 XML 文件时无需再进行编码转换。以下是 database.xml 文件的部分内容：

```
<?xml version='1.0' encoding="GB2312"?>
<database>
  <user oid="12" email="weiqin@yahoo.com" password="weiqin"
      firstname="卫琴" lastname="孙"/>
  <product oid="110" name="CD 随身听" image="multimedia/cdsuishentin_small.gif"
      price="1290.00" description="可播放多种格式"/>
  <product oid="111" name="电热水壶" image="multimedia/dianrenshuihu_small.gif"
      price="269.00" description="方便、快捷、卫生、省电。"/>
```

```
    .....
</database>
```

9.2.4 处理响应结果的编码

可以通过以下方式来设置响应结果的编码：

- 在 Servlet 中

```
response.setContentType("text/html;charset=GB2312");
```

- 在 JSP 中

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

- 在 HTML 中

```
<head>
  <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset= GB2312">
</head>
```

9.3 Java 对 I18N 的支持

Java 在其核心库中提供了支持 I18N 的类和接口。本节将简短地介绍 Java 中和 I18N 有关的核心组件。Struts 框架依赖于这些 Java I18N 组件来实现对 I18N 的支持，因此，掌握 Java I18N 组件的使用方法有助于理解 Struts 应用的国际化机制。

9.3.1 Locale 类

java.util.Locale 类是最重要的 Java I18N 类，在 Java 语言中，几乎所有对国际化和本地化的支持都依赖于这个类。

关于 Locale 的概念已经在本章 9.1 节中做了解释。Locale 类的实例代表一种特定的语言和地区。如果 Java 类库中的某个类在运行时需要根据 Locale 对象来调整其功能，那么就称这个类是本地敏感的（Locale-Sensitive）。例如，java.text.DateFormat 类就是本地敏感的，因为它需要依照特定的 Locale 对象来对日期进行相应的格式化。

Local 对象本身并不执行和 I18N 相关的格式化或解析工作。Local 对象仅仅负责向本地敏感的类提供本地化信息。例如，DateFormat 类依据 Local 对象来确定日期的格式，然后对日期进行语法分析和格式化。

创建 Local 对象时，需要明确地指定其语言和国家代码。以下代码片段创建了两个 Local 对象，一个是美国的，另外一个是中国的：

```
Locale usLocale = new Locale("en", "US");
Locale chLocale = new Locale("ch", "CH");
```

构造方法的第一个参数是语言代码。语言代码由两个小写字母组成，遵从 ISO-639 规范。可以从 http://www.unicode.org/unicode/onlinedat/languages.html 中获得完整的语言代码列表。

构造方法的第二个参数是国家代码，它由两个大写字母组成，遵从 ISO-3166 规范。可以从 http://www.unicode.org/unicode/onlinedat/countries.html 中获得完整的国家代码列表。

Locale 类提供了几个静态常量，它们代表一些常用的 Locale 实例。例如，如果要获得 Japanese Locale 实例，可以使用如下两种方法之一：

```
Locale locale1 = Locale.JAPAN;
Locale locale2 = new Locale("ja", "JP");
```

1．Web 容器中 Locale 对象的来源

Java 虚拟机在启动时会查询操作系统，为运行环境设置默认的 Locale。Java 程序可以调用 java.util.Locale 类的静态方法 getLocale()来获得默认的 Locale：

```
Locale defaultLocale = Locale.getDefault();
```

Web 容器在其本地环境中通常会使用以上默认的 Locale；而对于特定的终端用户，Web 容器会从 HTTP 请求中获取 Locale 信息。Web 容器中 Locale 对象的来源，如图 9-3 所示。

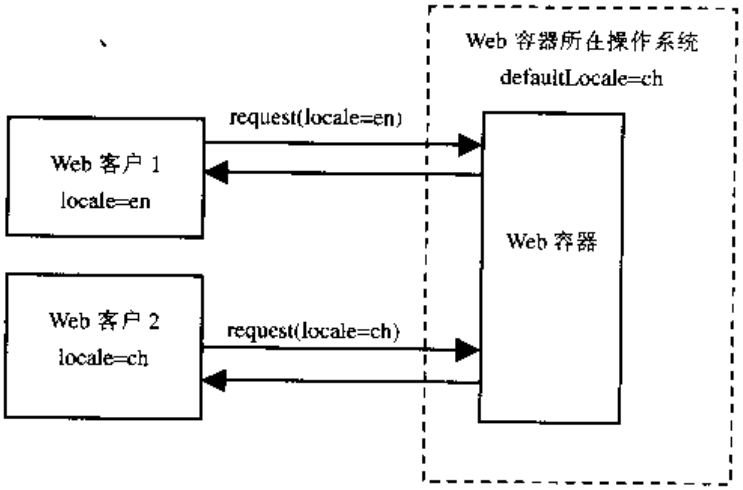

图 9-3　Web 容器中 Locale 对象的来源

2．在 Web 应用中访问 Locale 对象

在上文中讲过可以通过 Locale 的构造方法来创建 Locale 对象，在创建 Locale 对象时应该把语言和国家代码两个参数传递给构造方法。对于 Web 应用程序，通常不必创建自己的 Locale 实例，因为 Web 容器会负责创建所需的 Locale 实例。在应用程序中，可以调用 HttpServletRequest 对象的以下两个方法，来取得包含 Web 客户的 Locale 信息的 Locale 实例：

```
public java.util.Locale getLocale();
public java.util.Enumeration getLocales();
```

这两个方法都会访问 HTTP 请求中的 Accept-Language 头信息。getLocale()方法返回客户优先使用的 Locale，而 getLocales()方法返回一个 Enumeration 集合对象，它包含了按优先级降序排列的所有 Locale 对象。如果客户没有配置任何 Locale，getLocale()方法将会返回默认的 Locale。

大多数 Web 浏览器允许用户配置 Locale。例如，对于微软的 IE 浏览器来说，可以选

择【工具】→【Internet 选项】命令，单击【语言】按钮，然后在弹出的语言设置对话框中选择语言，如图 9-4 所示。

图 9-4 在 IE 浏览器中设置用户的 Locale

例程 9-1（LocaleServlet.java）演示了如何在 Servlet 中访问用户的 Locale 信息。

例程 9-1 使用 Servlet 访问用户的 Locale 信息

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Locale;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Prints out information about a user's preferred locales
 */
public class LocaleServlet extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html;charset=GB2312";

  /**
   * Initialize the servlet
   */
  public void init(ServletConfig config) throws ServletException {
```

```
    super.init(config);
  }

  /**
   * Process the HTTP Get request
   */
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();

    out.println("<html>");
    out.println("<head><title>The Example Locale Servlet</title></head>");
    out.println("<body>");

    // Retrieve and print out the user's preferred locale
    Locale preferredLocale = request.getLocale();
    out.println("<p>The user's preffered Locale is " + preferredLocale + "</p>");

    // Retrieve all of the supported locales of the user
    out.println("<p>A list of preferred Locales in descreasing order</p>");
    Enumeration allUserSupportedLocales = request.getLocales();
    out.println("<ul>");
    while( allUserSupportedLocales.hasMoreElements() ){
      Locale supportedLocale = (Locale)allUserSupportedLocales.nextElement();
      StringBuffer buf = new StringBuffer();
      buf.append("<li>");
      buf.append("Locale: ");
      buf.append( supportedLocale );
      buf.append( " - " );
      buf.append( supportedLocale.getDisplayName() );
      buf.append("</li>");
      // Print out the line for a single Locale
      out.println( buf.toString() );
    }
    out.println("</ul>");

    // Get the container's default locale
    Locale servletContainerLocale = Locale.getDefault();
    out.println("<p>The container's Locale " + servletContainerLocale + "</p>");
    out.println("</body></html>");
  }
}
```

假定把 LocaleServlet 发布到 helloapp 应用中，在 web.xml 中的配置代码如下：

```
<servlet>
```

```
    <servlet-name>locale</servlet-name>
    <servlet-class>LocaleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>locale</servlet-name>
    <url-pattern>locale</url-pattern>
  </servlet-mapping>
```

如果 Web 容器在中文操作系统中运行，IE 浏览器的语言选项采用如图 9-4 所示的设置，则运行上面的 Servlet，将会看到类似如图 9-5 所示的输出结果。

图 9-5 LocaleServlet 的输出结果

如果对如图 9-4 所示的 IE 浏览器的语言选项进行一些改动，颠倒一下中文和英文两种语言的优先级，则再次运行 LocaleServlet，将得到如图 9-6 所示的输出结果。

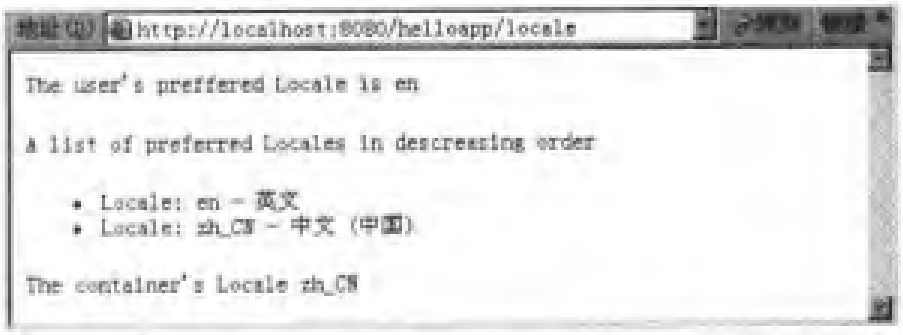

图 9-6 修改 IE 浏览器的语言优先级后 LocaleServlet 的输出结果

从图 9-6 中可以看出，Web 客户选用的 Locale 由原来的中文变成了英文，但 Web 容器默认的 Locale 保持不变，因为后者是由操作系统决定的。

3. 在 Struts 应用中访问 Locale 对象

由于 Web 服务器并不和客户浏览器保持长期的连接，因此每个发送到 Web 容器的 HTTP 请求中都包含了 Locale 信息。Struts 配置文件的<controller>元素的 locale 属性指定是否把 Locale 对象保存在 session 范围中，默认值为 true，表示会把 Locale 对象保存在 session 范围中。在处理每一个用户请求时，RequestProcessor 类都会调用它的 processLocale()方法，该方法代码如下：

```
protected void processLocale(HttpServletRequest request,
                             HttpServletResponse response) {
```

```
        // Are we configured to select the Locale automatically?
        if (!moduleConfig.getControllerConfig().getLocale()) {
            return;
        }
        // Has a Locale already been selected?
        HttpSession session = request.getSession();
        if (session.getAttribute(Globals.LOCALE_KEY) != null) {
            return;
        }
        // Use the Locale returned by the servlet container (if any)
        Locale locale = request.getLocale();
        if (locale != null) {
            if (log.isDebugEnabled()) {
                log.debug(" Setting user locale '" + locale + "'");
            }
            session.setAttribute(Globals.LOCALE_KEY, locale);
        }
    }
```

从以上代码中可以看出，尽管每次发送的HTTP请求都包含Locale信息，processLocale()方法把Locale对象存储在session范围中必须满足以下条件：

- Struts配置文件的<controller>元素的locale属性为true。
- 在session范围内Locale对象还不存在。

processLocale()方法把Locale对象存储在session范围中时，属性key为Globals.LOCALE_KEY，这个常量的字符串值为“org.apache.struts.action.LOCALE”。

如果应用程序允许用户在同一个会话中改变Locale，那么应该对每一个新的HttpServletRequest调用其getLocale()方法，来判断用户是否改变了Locale，如果Locale发生改变，就把新的Locale对象保存在session范围内。本书5.1.2小节的例程5-3所示的CustomRequestProcessor类实现了这一功能。

在Struts应用程序中可以很方便地获取Locale信息。例如，如果在Action类中访问Locale信息，可以调用在Struts Action基类中定义的getLocale()方法，该方法的源代码如下：

```
    protected Locale getLocale(HttpServletRequest request) {
        return RequestUtils.getUserLocale(request, null);
    }
```

Action类的getLocale()方法调用RequestUtils.getUserLocale()方法，该方法的源代码如下：

```
    public static Locale getUserLocale(HttpServletRequest request, String locale) {
            Locale userLocale = null;
            HttpSession session = request.getSession(false);

            if (locale == null) {
```

```
            locale = Globals.LOCALE_KEY;
        }

        // Only check session if sessions are enabled
        if (session != null) {
            userLocale = (Locale) session.getAttribute(locale);
        }
        if (userLocale == null) {
            // Returns Locale based on Accept-Language header or the server default
            userLocale = request.getLocale();
        }
        return userLocale;
    }
```

getUserLocale()方法先通过 HttpServletRequest 参数获得 HttpSession 对象，然后再通过 HttpSession 对象来读取 Locale 对象。如果存在 HttpSession 对象并且 HttpSession 中存储了 Locale 对象，就返回该 Locale 对象，否则就直接调用 HttpServletRequest 的 getLocale()方法取得 Locale 对象，并将它返回。

在 Web 应用程序的其他地方也可以直接调用 RequestUtils 类的 getUserLocale()方法来获得 Locale 对象。

9.3.2 ResourceBundle 类

java.util.ResourceBundle 类提供存放和管理与 Locale 相关的资源的功能。这些资源包括文本域或按钮的 Label、状态信息、图片名、错误信息和网页标题等。

Struts 框架并没有直接使用 Java 语言提供的 ResourceBundle 类。在 Struts 框架中提供了两个类：

- org.apache.struts.util.MessageResources
- org.apache.struts.util.PropertyMessageResources

这两个类具有和 ResourceBundle 相似的功能，其中 PropertyMessageResources 是 MessageResources 类的子类。

9.3.3 MessageFormat 类和复合消息

Java 的 ResourceBundle 和 Struts 的 MessageResources 类都允许使用静态和动态的文本。静态文本指定的是事先就已经具有明确内容的文本，例如文本框和按钮的 Label。动态文本指的是只有在运行时才能确定内容的文本。下面通过一个例子来说明这两者的区别。

假定应用需要向用户显示提示信息，提示用户表单中的名字和电话文本域不允许为空。一种方法是在 Resource Bundle 中添加如下信息：

```
error.requiredfield.name=The Name field is required to save.
error.requiredfield.phone=The Phone field is required to save.
```

尽管以上方法是可行的，但是如果有上百个域不允许为空时怎么办？按照以上方法，需要给每一个域添加一个消息，这会导致 Resource Bundle 变得很庞大，难以维护。

事实上，以上两则消息间惟一的不同是具体的域名不一样。一个更加容易且易维护的方法是使用 java.text.MessageFormat 类的功能，此时只需要提供以下的消息文本：

```
error.requiredfield=The {0} field is required to save.
label.phone=Phone
label.name=Name
```

以上消息中包含了一个参数{0}，如果还有多个参数，可以用{1}、{2}来表示，依次类推。在运行时，MessageFormat 类的 format()方法可以把参数{0}替换为真正的动态文本内容。例程 9-2（FormatExample.java）为使用 MessageFormat 类来格式化不同文本消息的例子。

例程 9-2　FormatExample.java

```
import java.util.ResourceBundle;
import java.util.Locale;
import java.text.MessageFormat;

public class FormatExample {

  public static void main(String[] args) {
    // Load the resource bundle
    ResourceBundle bundle = ResourceBundle.getBundle( "ApplicationResources" );

    // Get the message template
    String requiredFieldMessage = bundle.getString( "error.requiredfield" );

    // Create a String array of size one to hold the arguments
    String[] messageArgs = new String[1];

    // Get the "Name" field from the bundle and load it in as an argument
    messageArgs[0] = bundle.getString( "label.name" );

    // Format the message using the message and the arguments
    String formattedNameMessage =
      MessageFormat.format( requiredFieldMessage, messageArgs );

    System.out.println( formattedNameMessage );

    // Get the "Phone" field from the bundle and load it in as an argument
    messageArgs[0] = bundle.getString( "label.phone" );

    // Format the message using the message and the arguments
    String formattedPhoneMessage =
      MessageFormat.format( requiredFieldMessage, messageArgs );
```

```
            System.out.println( formattedPhoneMessage );
        }
    }
```

通常把包含可变数据的消息称为复合消息。复合消息允许在程序运行时把动态数据加入到消息文本中。这能够减少 Resource Bundle 中的静态消息数量，从而减少把静态消息文本翻译成其他 Locale 版本所花费的时间。

当然，在 Resource Bundle 中使用复合信息会使文本的翻译变得更加困难。因为文本包含了直到运行时才知道的替代值，而把包含替代值的消息文本翻译成不同的语言时，往往要对语序做适当调整。

Struts 框架封装了 MessageFormat 类的功能，支持复合消息文本，该功能的实现对于 Struts 的其他组件是透明的。

9.4 Struts 框架对国际化的支持

Struts 框架对国际化的支持体现在能够输出和用户 Locale 相符合的文本和图片上。当 Struts 配置文件的<controller>元素的 locale 属性为 true 时，Struts 框架把用户的 Locale 实例保存在 session 范围内，这样，Struts 框架能自动根据这一 Locale 实例来从 Resource Bundle 中选择合适的资源文件。如图 9-7 所示，当用户的 Locale 为英文时，Struts 框架就会向用户返回来自于 application_en.properties 文件的文本内容；当用户的 Locale 为中文时，Struts 框架就会向用户返回来自于 application_ch.properties 文件的文本内容。

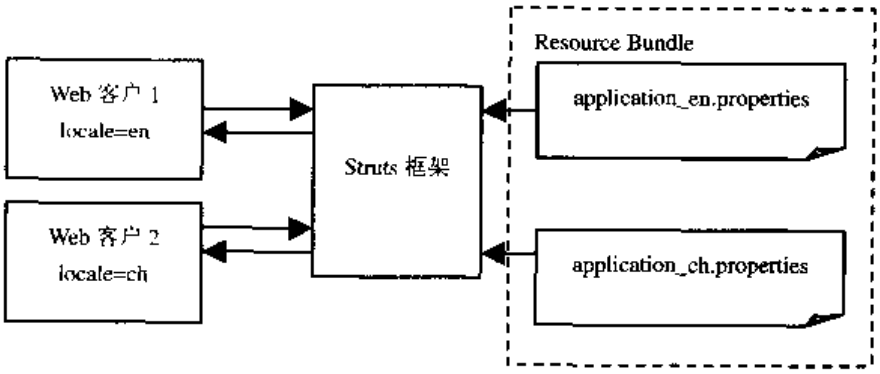

图 9-7 Struts 框架根据用户的 Locale 来选择资源文件

9.4.1 创建 Struts 的 Resource Bundle

对于多应用模块的 Struts 应用，可以为每个子应用配置一个或多个 Resource Bundle，应用模块中的 Action、ActionForm Bean、JSP 页和客户化标签都可以访问这些 Bundle。Struts 配置文件中的每个<message-resources>元素定义了一个 Resource Bundle。当应用中包含多个 Resource Bundle 时，它们通过<message-resources>元素的 key 属性来区别，例如：

```
<message-resources
```

```
    parameter="application"/>
  <message-resources
    key="IMAGE_RESOURCE_KEY"
    parameter="imageresources"/>
```

以上代码配置了两个 Resource Bundle。其中第一个<message-resources>元素没有设置 key 属性，表示默认的 Resource Bundle。

Resource Bundle 的持久化消息文本存储在资源文件中，其扩展名为“.properties”，这一文件中消息的格式为：key=value。

在创建 Resource Bundle 的资源文件时，可以先提供一个默认的资源文件，例如对于以下 Resource Bundle 的配置代码：

```
<message-resources    parameter="application"/>
```

默认资源文件应该取名为 application.properties。如果应用程序需要支持中文用户，可以再创建一个包含中文消息的资源文件，文件名为：application_ch_CN.properties 或 application_ch.properties。

当 Struts 框架处理 Locale 为中文的用户请求时，它会依次搜索如下资源文件：

- application_ch_CN.properties
- application_ch.properties
- application.properties

Struts 框架首先在 WEB-INF/classes/目录下寻找 application_ch_CN.properties 文件，如果存在该文件，就从该文件中获取文本消息，否则再依次寻找 application_ch.properties 和 application.properties 文件。

应该总是为 Resource Bundle 提供默认的资源文件，这样，当不存在和某个 Locale 对应的资源文件时，就可以使用默认的资源文件。

应该把 Resource Bundle 的资源文件放在能被定位并加载的位置。对于 Web 应用程序，资源文件的存放目录为 WEB-INF/classes 目录。如果在配置 Resource Bundle 时还给定了包名，那么包名应该和资源文件所在的子目录对应。例如，如果在 Struts 配置文件中对 Resource Bundle 做如下配置：

```
<message-resources parameter=" hello.application"/>
```

那么默认资源文件名为 application.properties，它的存放位置为：WEB-INF/classes/hello/application.properties

9.4.2　访问 Resource Bundle

Struts 应用的每个 Resource Bundle 和 org.apache.struts.util.MessageResources 类（实际上是其子类 PropertyMessageResources）的一个实例对应。MessageResources 对象中存放了来自资源文件的文本。当应用程序初始化时，这些 MessageResources 实例被存储在

ServletContext 中（即 application 范围内），因此任何一个 Web 组件都可以访问它们。一个 MessageResources 对象可以包含多种本地化版本的资源文件的数据，如图 9-8 所示为 MessageResources 和资源文件的关系。

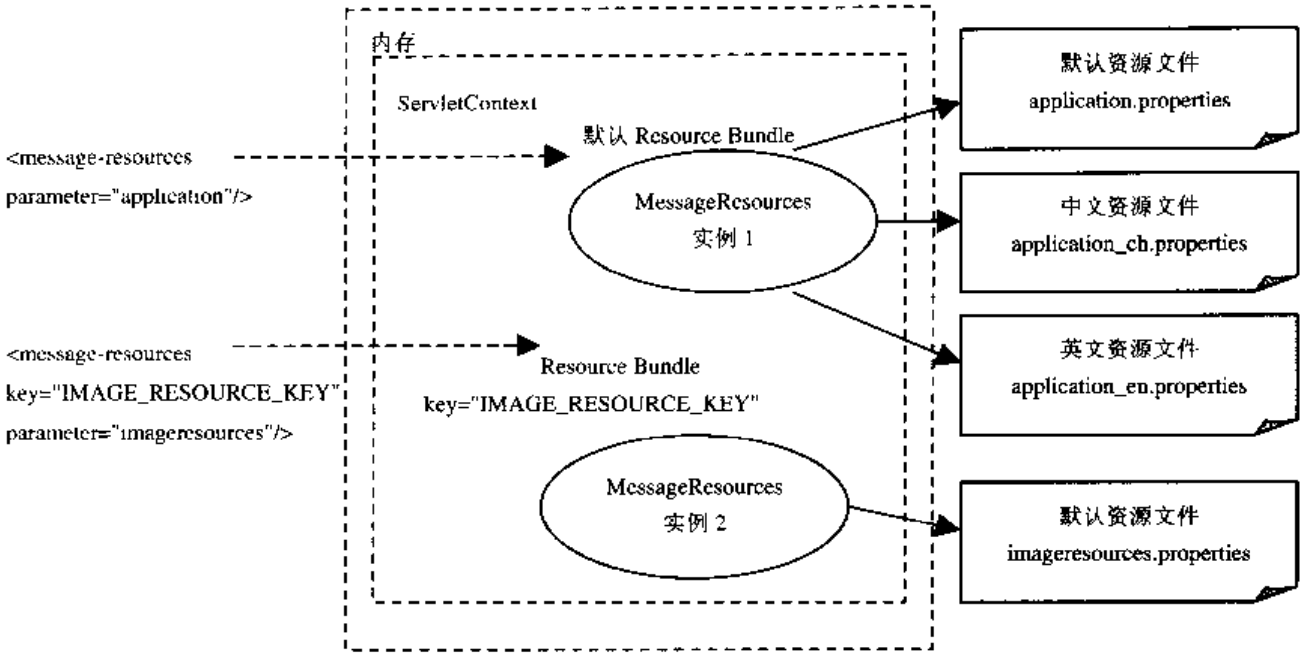

图 9-8 MessageResources 和资源文件的关系

Struts 应用、子应用模块、Resource Bundle 和资源文件之间存在以下关系：

- 一个 Struts 应用可以有多个子应用模块，必须有且只有一个默认子应用模块。
- 一个子应用模块可以有多个 Resource Bundle，必须有且只有一个默认 Resource Bundle。
- 一个 Resource Bundle 可以有多个资源文件，必须有且只有一个默认资源文件。

下面介绍在 Struts 应用中访问 Resource Bundle 的途径。

1．通过编程来访问 Resource Bundle

在 Action 基类中定义了 getResources(request)方法，它可以返回默认的 MessageResources 对象，代表当前应用模块使用的默认 Resource Bundle。如果要获得特定的 MessageResources 对象，可以调用 Action 基类的 getResources(request,key)方法，其中参数 key 和 Struts 配置文件的<message-resources>元素的 key 属性对应。得到了 MessageResources 对象后，就可以通过它的方法来访问消息文本。例如：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {

    Locale locale = getLocale(request);
    MessageResources messages = getResources(request);
    String msg=messages.getMessage(locale, "hello.no.username.error ");
```

```
        ......
    }
```

org.apache.struts.util.MessageResources 的 getMessage()方法有好几种重载形式，下面列出常用的几种：

- 根据参数指定的 Locale 检索对应的资源文件，然后返回和参数 key 对应的消息文本：

  getMessage(java.util.Locale locale, java.lang.String key)

- 根据参数指定的 Locale 检索对应的资源文件，然后返回和参数 key 对应的消息文本，args 参数用于替换复合消息文本中的参数：

  getMessage(java.util.Locale locale, java.lang.String key, java.lang.Object[] args)

- 根据默认的 Locale 检索对应的资源文件，然后返回和参数 key 对应的消息文本：

  getMessage(java.lang.String key)

2. 使用和 Resource Bundle 绑定的 Struts 组件

Struts 框架中的许多内在组件和 Resource Bundle 是绑定在一起的，如：

- ActionMessage 类和<html:errors>标签。
- Struts Bean 标签库的<bean:message>标签。
- 在 Validator 验证框架中访问 Resource Bundle，参见第 10 章（Validator 验证框架）。
- 在声明型异常处理中访问 Resource Bundle，参见第 11 章（Struts 异常处理）。

1）联合使用 ActionMessage 类和<html:errors>标签

每个 ActionMessage 实例代表 Resource Bundle 中的一条消息。在调用 ActionMessage 的构造方法时，需要传递消息 key。例如，以下代码在 ActionForm 的 validate()方法中创建了 ActionMessage 对象。

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();

    if ((userName == null) || (userName.length() < 1))
        errors.add("username", new ActionMessage("hello.no.username.error"));
    return errors;
}
```

对于复合消息，在创建 ActionMessage 对象时，可以调用带两个参数的构造方法：ActionMessage(java.lang.String key, java.lang.Object[] values)，values 参数用于替换复合消息中的参数。

在 JSP 页面中，使用<html:errors>标签，就能读取并显示 ActionErrors 集合中所有 ActionMessage 对象包含的消息文本。

2）使用 Bean 标签库的<bean:message>标签

Struts 框架包含了一些可以访问应用的消息资源的客户化标签，其中使用最频繁的一个标签为<bean:message>标签。<bean:message>标签从应用的 Resource Bundle 中获取消息字

符串。例如，以下 JSP 代码使用<bean:message>标签输出网页的标题：

```
<head>
  <title><bean:message key="hello.jsp.title"/></title>
</head>
```

以上<bean:message>标签的 key 属性为“hello.jsp.title”，<bean:message>标签根据存储在 session 范围内的 Locale 实例，从默认的 Resource Bundle 中检索和 Locale 对应的资源文件，再从资源文件中读取和“hello.jsp.title”对应的消息字符串：

```
hello.jsp.title=Hello - A first Struts program
```

最后，<bean:message>标签向网页上输出“Hello - A first Struts program”。

<bean:message>有一个 bundle 属性，用于指定被访问的 Resource Bundle，它和<message-resources>元素的key属性匹配。如果没有设置bundle属性，将访问默认的Resource Bundle。

9.5 对 helloapp 应用实现国际化

在本书第 2 章介绍了 helloapp 应用，该应用只支持英文，如果在输入表单中输入中文，在返回网页上将会出现乱码。现在对它进行一些修改，使它能同时支持中文和英文两种客户。

9.5.1 对 JSP 文件的文本、图片和按钮进行国际化

下面以 helloapp 应用的 hello.jsp 为例，介绍对 JSP 文件进行国际化的步骤。

步骤

（1）设置字符编码。

为了保证同一个 JSP 页面能支持多种语言，可以将所有 JSP 页面的字符编码统一设为“UTF-8”：

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
```

（2）对文本国际化。

在 JSP 文件中不应该直接包含本地化的消息文本，而应该通过<bean:message>标签从 Resource Bundle 中获得文本。本书第 2 章中提供的 hello.jsp 已经做到了这点，它包含以下<bean:message>标签：

```
<bean:message key="hello.jsp.title"/>
<bean:message key="hello.jsp.page.heading"/>
<bean:message key="hello.jsp.page.hello"/>
<bean:message key="hello.jsp.prompt.person"/>
```

（3）对按钮国际化。

在 hello.jsp 的 HTML 表单中定义了如下按钮：

```
<html:submit property="submit" value="Submit"/>
<html:reset/>
```

为了使按钮的 Label 支持多种语言，对上面的代码做如下修改：

```
<html:submit property="submit" >
    <bean:message key="hello.jsp.page.submit"/>
</html:submit>
<html:reset property="reset" >
    <bean:message key="hello.jsp.page.reset"/>
</html:reset>
```

这样，按钮的 Label 内容由<bean:message>标签从 Resource Bundle 中获得。

（4）对图片国际化。

在 hello.jsp 中包含如下图片：

```
<html:img page="/struts-power.gif" alt="Powered by Struts"/>
```

为了使 hello.jsp 能根据不同的 Locale 动态加载本地化的图片，对上面的代码做如下修改：

```
<html:img pageKey="hello.jsp.page.stutsimage" altKey="hello.jsp.page.struts"/>
```

以上的<html:img>标签将从 Resource Bundle 中读取需要加载的图片名，以及当用户将鼠标指针移到图片上时显示的替代文本内容。<html:img>标签的 pageKey 和 altKey 属性均和 Resource Bundle 中相应的消息 key 匹配，其中 pageKey 属性指定图片名，altKey 属性指定当用户将鼠标移到图片上时显示的替代文本内容。

如果用户的 Locale 为中文，那么<html:img>标签的 pageKey 和 altKey 属性与 Resource Bundle 中的以下内容匹配：

```
hello.jsp.page.stutsimage=/struts-power-ch.gif
hello.jsp.page.stuts=基于 Struts 技术
```

如果用户的 Locale 为英文，<html:img>标签的 pageKey 和 altKey 属性与 Resource Bundle 中的以下内容匹配：

```
hello.jsp.page.stutsimage=/struts-power.gif
hello.jsp.page.stuts=Powered By Struts
```

9.5.2　创建临时中文资源文件

根据 9.5.1 小节的内容，由于在 hello.jsp 中增加了一些消息 key，应该在默认的 application.properties 文件中添加相应的消息文本，修改后的文件内容如下（其中粗体字为增加的内容）：

```
#Application Resources for the "Hello" sample application
```

```
#Application Resources that are specific to the hello.jsp file
hello.jsp.title=Hello - A first Struts program
hello.jsp.page.heading=Hello World! A first Struts application
hello.jsp.prompt.person=Please enter a UserName to say hello to :
hello.jsp.page.hello=Hello
hello.jsp.page.submit=Submit
hello.jsp.page.reset=Reset
hello.jsp.page.stutsimage=/struts-power.gif
hello.jsp.page.stuts=Powered By Struts

#Validation and error messages for HelloForm.java and HelloAction.java
hello.dont.talk.to.monster=We don't want to say hello to Monster!!!
hello.no.username.error=Please enter a <i>UserName</i> to say hello to!
```

接下来根据默认的 application.properties 文件，创建一个临时中文资源文件，假定名为 application_temp.properties，内容如下：

```
#Application Resources for the "Hello" sample application

#Application Resources that are specific to the hello.jsp file
hello.jsp.title=Hello - 第一个 Struts 应用
hello.jsp.page.heading=Hello World! 第一个 Struts 应用
hello.jsp.prompt.person=请输入您想打招呼的用户名:
hello.jsp.page.hello=您好
hello.jsp.page.submit=提交
hello.jsp.page.reset=复位
hello.jsp.page.stutsimage=/struts-power-ch.gif
hello.jsp.page.stuts=基于 Struts 技术

#Validation and error messages for HelloForm.java and HelloAction.java
hello.dont.talk.to.monster=我们不想对 Monster 打招呼!!!
hello.no.username.error=请输入您想打招呼的<i>用户名</i>!
```

9.5.3 对临时资源文件进行编码转换

在 JDK 中提供了 native2ascii 命令，它能够实现字符编码转换。在 DOS 下执行以下命令，将生成按 GB2312 编码的中文资源文件 application_zh_CN.properties：

```
native2ascii -encoding gb2312  application_temp.properties  application_zh_CN.properties
```

执行以上命令后，将生成具有如下内容的 application_zh_CN.properties 文件：

```
#Application Resources for the "Hello" sample application

#Application Resources that are specific to the hello.jsp file
hello.jsp.title=Hello - \u7b2c\u4e00\u4e2aStruts\u5e94\u7528
hello.jsp.page.heading=Hello World! \u7b2c\u4e00\u4e2aStruts\u5e94\u7528
```

```
hello.jsp.prompt.person=\u8bf7\u8f93\u5165\u4f60\u60f3\u6253\u62db\u547c\u7684\u7528\u6237\u5
40d:
hello.jsp.page.hello=\u4f60\u597d
hello.jsp.page.submit=\u63d0\u4ea4
hello.jsp.page.reset=\u590d\u4f4d
hello.jsp.page.stutsimage=/struts-power-ch.gif
hello.jsp.page.stuts=\u57fa\u4e8eStruts\u6280\u672f

#Validation and error messages for HelloForm.java and HelloAction.java
hello.dont.talk.to.monster=\u6211\u4eec\u4e0d\u60f3\u5bf9Monster\u6253\u62db\u547c!!!
hello.no.username.error=\u8bf7\u8f93\u5165\u4f60\u60f3\u6253\u62db\u547c\u7684<i>\u7528\u6237
\u540
d</i>!
```

当 Web 客户的 Locale 为中文时，Struts 框架将自动选择来自 application_zh_CN.properties 文件的消息文本。

9.5.4　创建英文资源文件

在 helloapp 应用中已经存在 application.properties 文件，只需要复制该文件，然后把新的文件改名为 application_en.properties。

完成了以上三个小节的操作内容后，在 helloapp 应用的 WEB-INF/classes/hello 目录下应该有三个资源文件：

- 默认资源文件：application.properties。
- 英文资源文件：application_en.properties。
- 中文资源文件：application_zh_CN.properties。

9.5.5　采用 Servlet 过滤器设置请求数据的字符编码

调用 HttpServletRequest 的 setCharacterEncoding("UTF-8")方法，能够把用户的请求数据的字符编码也设为 UTF-8。这样，Web 应用的输入和输出都采用同一种编码，就无需在程序中进行编码转换。这一功能可以由 Servlet 过滤器来完成，因为它可以预处理所有的 HTTP 请求，这样能避免在每个 Web 组件中设置请求数据编码，如图 9-9 所示。

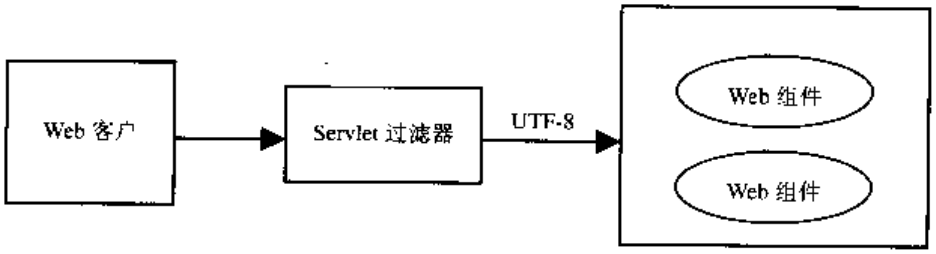

图 9-9　Servlet 过滤器预先设置请求数据的编码

您也许会问，为什么不在 Servlet 过滤器中预先设置 HttpServletResponse 的字符编码？这主要是因为在 Servlet 过滤器中预先设置了 HttpServletResponse 的字符编码后，只能决定 Servlet 使用的编码，而对 JSP 不起作用。所以如果要设置 JSP 的输出编码，还是应该按照本章 9.5.1 小节的第 1 点说明直接在 JSP 文件中声明。

例程 9-3 为 SetCharacterEncodingFilter 的源程序。

例程 9-3 SetCharacterEncodingFilter.java

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.UnavailableException;

/**
 * Example filter that sets the character encoding to be used in parsing the
 * incoming request
 */
public class SetCharacterEncodingFilter implements Filter {

    public void destroy() {}
    /**
     * Select and set (if specified) the character encoding to be used to
     * interpret request parameters for this request.
     */
    public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain)throws IOException, ServletException {

        request.setCharacterEncoding("UTF-8");
        // 传递控制到下一个过滤器
        chain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {}
}
```

接下来在 web.xml 文件中配置 SetCharacterEncodingFilter 过滤器，这个过滤器预处理所有的 URL：

```
<filter>
    <filter-name>Set Character Encoding</filter-name>
    <filter-class>SetCharacterEncodingFilter</filter-class>
```

```
</filter>
<filter-mapping>
    <filter-name>Set Character Encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

9.5.6　运行国际化的 helloapp 应用

按照以上步骤对原来的 helloapp 应用做了改动后，新的 Web 应用可以支持 I18N，如果还需要 Web 应用支持日本客户，将不需要修改 Web 应用的代码或配置，只要仿照 9.5.2 小节和 9.5.3 小节再创建一个 application_ja_JP.properties 资源文件即可。把修改后的 helloapp 应用发布到 Tomcat 服务器中，然后按以下方式访问 helloapp 应用。

- 参照 9.3.1 小节第 2 点的图 9-4，把 IE 浏览器的语言选项设置为中文，然后访问 helloapp 应用，将会看到如图 9-10 所示的中文网页，网页中的文本、按钮的 Label 以及图片均为中文。如果在网页表单中输入中文名字，会得到正确的返回结果。

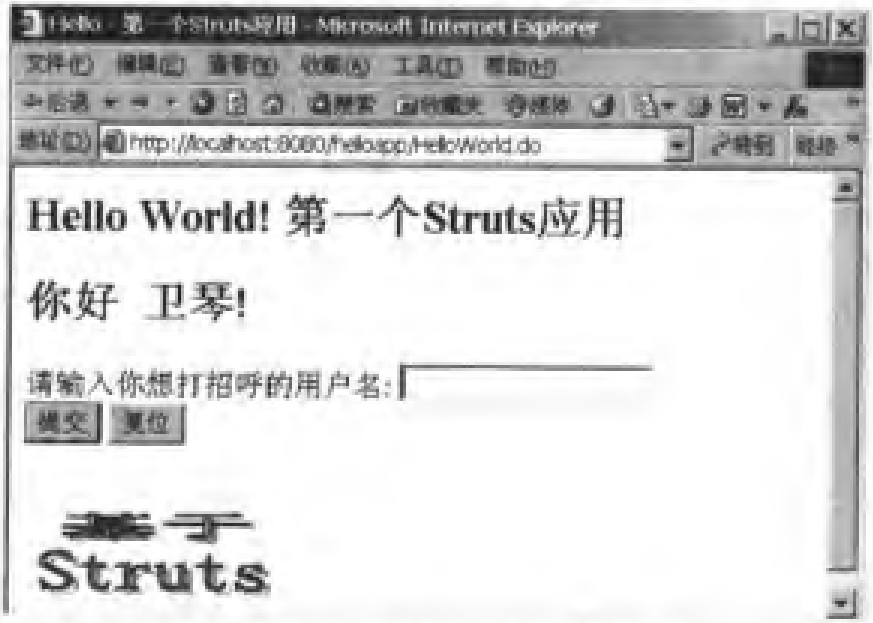

图 9-10　helloapp 应用面向中文客户的输出网页

- 参照 9.3.1 小节第 2 点图 9-4，把 IE 浏览器的语言选项设置为英文，然后访问 helloapp 应用，将会看到如图 9-11 所示的英文网页。

在同一个会话中，Struts 框架仅读取 HTTP 请求的 Locale 信息一次，然后就把 Locale 对象保存在 session 范围内，因此在同一个会话中，Locale 对象保持不变，选用的资源文件也不会变化，因此在上面的实验当中，把 IE 浏览器的语言选项设为英文后，应该重新打开一个 IE 浏览器，保证在一个新的会话中访问 helloapp 应用，这样才会看到英文网页。

图 9-11 helloapp 应用面向英文客户的输出网页

9.6 异常处理的国际化

在处理异常时，也应该考虑对 I18N 的支持。除非已经对应用抛出的异常消息做本地化，否则不应该直接向终端用户展示原始的异常消息。不懂 Java 语言的终端用户很难理解从 Java 虚拟机堆栈中抛出的 Java 异常消息，如果这些异常消息使用的不是用户的本地语言，那就更加会让用户困惑不解。

应该先把异常捕获，对异常消息进行本地化后，再把它展示给用户。Struts 的 Resource Bundle 和 ActionMessage 类可以完成这一功能。直接向终端用户显示 Java 异常绝对是不可取的。即使发生了无法恢复的系统错误，也应该向用户显示本地化的系统错误页。

关于异常处理的更多知识请参考本书的第 11 章（Struts 异常处理）。

9.7 小 结

本章介绍了软件的本地化与国际化的概念，然后详细介绍了对 Struts 应用实现国际化的原理和方法。与国际化密切相关的两个组件是 Locale 和 Resource Bundle：

- Locale：包含了用户的本地化信息，如语言和国家。
- Resource Bundle：包含了多个消息资源文件，每个消息资源文件存放和一种 Locale 相对应的本地化消息文本。

Struts 框架在初始化时，把 Resource Bundle（即 MessageResources 对象）存储在 application 范围内；在响应用户请求时，把包含用户 Locale 信息的 Locale 实例存储在 session 范围内。Struts 框架能自动根据这一 Locale 实例，从 Resource Bundle 中检索相应的资源文件，再从资源文件中读取本地化的消息文本。

对 Struts 应用实现国际化应该遵循以下原则：

- 尽量不在 Servlet 中使用含非英文字符的常量字符串。
- 对于 JSP 文件，应该对 page 指令中的 charset 属性进行相应的设置。
- 不要在 JSP 文件中直接包含本地化的消息资源，而应该把消息资源存放在 Resource Bundle 的资源文件中。
- 不必在每个 JSP 或 Servlet 中设置 HTTP 请求的字符编码，可以在 Servlet 过滤器中设置编码：

```
HttpServletRequest.setCharaterEncoding(String encoding);
```

- 尽量使用“UTF-8”作为 HTTP 请求和响应的字符编码，而不是“GBK”或“GB2312”。
- 充分考虑底层数据库所使用的编码，它可能会给应用程序的移植带来麻烦。

第 10 章　Validator 验证框架

Struts 框架能够在 ActionForm Bean 的 validate()方法中对用户输入的表单数据进行验证。这种验证方式有两个局限：

- 必须通过程序代码来实现验证逻辑，如果验证逻辑发生变化，必须重新编写和编译程序代码。
- 当系统中有多个 ActionForm Bean，并且它们包含一些相同的验证逻辑时，开发人员必须对每个 ActionForm Bean 进行重复编程，完成相同的验证逻辑，这会降低代码的可重用性。

本章介绍由 David Winterfeldt 创建的 Validator 框架，它能被集成到 Struts 框架中，负责数据验证。Validator 框架能够克服以上在 ActionForm Bean 中以编程方式进行数据验证的局限，它允许为 Struts 应用灵活的配置验证规则，无需编程。

如今，Validator 框架在 Struts 应用中得到了广泛的运用，Apache 组织已经把它添加到 Jarkarta 工程中，在 Struts 软件中携带了 Validator 框架。

10.1　安装和配置 Validator 框架

Validator 框架如今成为 Jakarta 的公共项目的一部分，可以从 http://jakarta.apache.org/commons/下载单独的 Validator 框架软件。此外，在 Struts 软件中也携带了 Validator 框架。

Validator 框架主要依赖于两个 JAR 文件：

- jakarta-oro.jar：提供了一组处理文本的类，具有文本替换、过滤和分割等功能。
- commons-validator.jar：提供了一个简单、可扩展的验证框架，包含了通用的验证方法和验证规则。

在 Struts 软件中携带了这两个 JAR 文件。如果在 Struts 应用中使用了 Validator 框架，那么把这两个 JAR 文件添加到应用的 WEB-INF/lib 目录中。

Validator 框架采用两个基于 XML 的配置文件来配置验证规则。这两个文件为 validator-rules.xml 和 validation.xml。在 Struts 应用中，它们均存放在 WEB-INF 目录下。

10.1.1　validator-rules.xml 文件

validator-rules.xml 文件包含了一组通用的验证规则，对所有的 Struts 应用都适用。Struts 软件中携带了这个文件。一般情况下，没有必要修改该文件，除非需要修改或扩展默认的规则。

如果想扩展默认的验证规则，最好把自定义的客户化规则放在另一个 XML 文件中，而不是直接添加到 validator-rules.xml 文件中。这样，当升级 Validator 框架的版本时，无需修改 validator-rules.xml 文件。

validator-rules_1_1.dtd 定义了 validator-rules.xml 文件的语法。validator-rules.xml 文件的根元素为<form-validation>，它可以包含一个或多个<global>元素。<global>元素可以包含一个或多个<validator>元素。这些元素的 DTD 定义如下：

```
<!element form-validation (global+)>
<!element global (validator+)>
```

每个<validator>元素定义了一个惟一的验证规则。例如，以下代码定义了一个名为“required”的验证规则：

```
<validator     name="required"
               classname="org.apache.struts.validator.FieldChecks"
               method="validateRequired"
               methodParams="java.lang.Object,
                             org.apache.commons.validator.ValidatorAction,
                             org.apache.commons.validator.Field,
                             org.apache.struts.action.ActionMessages,
                             javax.servlet.http.HttpServletRequest"
               msg="errors.required">
</validator>
```

<validator>元素中还可以包含<javascript>子元素。关于<javascript>子元素的用法，将在本章 10.6 节中介绍。

<validator>元素有 7 个属性，它的 DTD 定义如下：

```
<!ATTLIST validator name CDATA #REQUIRED>
<!ATTLIST validator classname CDATA #REQUIRED>
<!ATTLIST validator method CDATA #REQUIRED>
<!ATTLIST validator methodParams CDATA #REQUIRED>
<!ATTLIST validator msg CDATA #REQUIRED>
<!ATTLIST validator depends CDATA #IMPLIED>
<!ATTLIST validator jsFunctionName CDATA #IMPLIED>
```

1．name 属性

name 属性指定验证规则的逻辑名，这个名字必须是惟一的。

2．classname 和 method 属性

classname 和 method 属性分别指定实现验证规则逻辑的类和方法。例如，对于以上“required”验证规则，由 FieldChecks 类的 validateRequired()方法来实现。methodParams 属性用来指定验证方法包含的参数，多个参数之间以逗号隔开。

3．msg 属性

msg 属性指定来自于 Resource Bundle 中的消息 key。当验证失败时，Validator 框架将根据这个消息 key 到 Resource Bundle 中查找匹配的消息文本。默认情况下，Validator 框架使用以下的消息文本:

```
errors.required={0} is required.
errors.minlength={0} cannot be less than {1} characters.
errors.maxlength={0} cannot be greater than {1} characters.
errors.invalid={0} is invalid.
errors.byte={0} must be a byte.
errors.short={0} must be a short.
errors.integer={0} must be an integer.
errors.long={0} must be a long.
errors.float={0} must be a float.
errors.double={0} must be a double.
errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid email address
```

应该把以上内容添加到应用的 Resource Bundle 中。如果不打算使用以上默认的错误消息文本，可以修改 validator-rules.xml 文件中<validator>元素的 msg 属性，使它引用其他自定义的消息 key；或者也可以修改以上的错误消息文本。

4．depends 属性

depends 属性指定在调用当前验证规则之前必须先调用的其他验证规则。例如，以下“minLength”验证规则的配置代码中就使用了 depends 属性：

```
<validator
  name="minLength"
  classname="org.apache.struts.validator.FieldChecks"
  method="validateMinLength"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionMessages,
    javax.servlet.http.HttpServletRequest"
  depends="required"
  msg="errors.minlength">
</validator>
```

以上代码表明，在调用“minLength”验证规则之前，应该先调用“required”规则。如果 depends 属性包含多个验证规则，则用逗号隔开，例如：

```
depends="required,integer"
```

如果调用 depends 属性指定的验证规则时验证失败，就不会再调用下一个规则。例如，

如果被验证的数据未通过“required”验证规则，表明该数据根本不存在，就没有必要再调用“minLength”验证规则，来检查数据的长度是否大于或等于最小长度。

Validator 框架提供了基本的、通用的验证规则，对 Struts 应用以及非 Struts 应用都适用。org.apache.commons.validator.GenericValidator 类提供了一组实现这些规则的静态方法。表 10-1 对这些方法做了说明。

表 10-1 GenericValidator 类的验证方法

方 法	描 述
isBlankOrNull	验证字段是否为 null，或者长度是否为 0
isByte	验证字段是否可以被转换为有效的 byte 类型的数据
isCreditCard	验证字段是否为有效的信用卡号
isDate	验证字段是否为有效的日期
isDouble	验证字段是否可以被转换为有效的 double 类型的数据
isEmail	验证字段是否为有效的 E-Mail 地址
isFloat	验证字段是否可以被转换为有效的 float 类型的数据
isInRange	验证字段是否介于最小值和最大值之间
isInt	验证字段是否可以被转换为有效的 int 类型的数据
isLong	验证字段是否可以被转换为有效的 long 类型的数据
isShort	验证字段是否可以被转换为有效的 short 类型的数据
matchRegexp	验证字段是否和正规表达式匹配
maxLength	验证字段是否小于或等于最大值
minLength	验证字段是否大于或等于最小值

在 Struts 框架中定义了专门用于验证表单字段的 org.apache.struts.validator.FieldChecks 类，它提供了和 GenericValidator 类似的方法。以下是 FieldChecks 类包含的验证方法：

- validateByte
- validateCreditCard
- validateDate
- validateDouble
- validateEmail
- validateFloat
- validateInteger
- validateLong
- validateMask
- validateMinLength
- validateMaxLength
- validateRange
- validateRequired
- validateShort

FieldChecks 类的 validateMask()方法和 GenericValidator 类的 matchRegexp()方法很相

似。validateMask()方法能够验证字段值是否和一个给定的 Jakarta 正规表达式（Regular Expression，简称 RegExp）匹配。Jakarta RegExp 表达式用于定义字符模式，例如："^[a-zA-Z]*$" 表示字符串必须只包含字母；"^\d{5}\d*$" 表示字符串必须是五位数字。要了解更多关于 RegExp 的知识，可以参阅 http://jakarta.apache.org/regexp/index.html。

在 validator_rules.xml 文件中，"mask" 验证规则就调用了 validateMask()方法，配置代码如下：

```
<validator name="mask"
           classname="org.apache.struts.validator.FieldChecks"
           method="validateMask"
           methodParams="java.lang.Object,
                         org.apache.commons.validator.ValidatorAction,
                         org.apache.commons.validator.Field,
                         org.apache.struts.action.ActionMessages,
                         javax.servlet.http.HttpServletRequest"
           depends=""
           msg="errors.invalid">
</validator>
```

FieldChecks 类实现了具体的验证逻辑。在对表单字段进行验证时，会调用 FieldChecks 类的相关方法，如果验证失败，就会创建包含错误消息的 ActionMessage 对象，并把该对象添加到 ActionMessages 集合对象中。

10.1.2　validation.xml 文件

Validator 框架的第二个配置文件为 validation.xml 文件。这个文件是针对于某个具体 Struts 应用的，需要开发人员来创建，它可以为应用中的 ActionForm 配置所需的验证规则，取代在 ActionForm 类中以编程的方式来实现验证逻辑。

例程 10-1 显示了一个简单的 validation.xml 文件样例。

例程 10-1　validation.xml 文件样例

```
<form-validation>
 <global>
  <constant>
   <constant-name>phone</constant-name>
   <constant-value>^\d{8}\d*$</constant-value>
  </constant>
 </global>

 <formset>
  <form name="checkoutForm">
          <field
           property="phone"
           depends="required,mask,minlength">
```

```
            <arg0 key="label.phone"/>
            <arg1 name="minlength" key="${var:minlength}" resource="false"/>
            <var>
              <var-name>mask</var-name>
              <var-value>${phone}</var-value>
            </var>
            <var>
                <var-name>minlength</var-name>
                <var-value>7</var-value>
            </var>
          </field>
      </form>
    </formset>
</form-validation>
```

1．<form-validation>元素

validation_1_1.dtd 描述了 validation.xml 文件的语法。validation.xml 文件的根元素为<form-validation>元素，它包含两个子元素：<global>和<formset>。<global>元素可以出现零次或多次，而<formset>元素可以出现一次或多次。它们的 DTD 定义如下：

```
<!element form-validation (global*, formset+)>
```

2．<global>元素

在<global>元素中可以定义<constant>子元素，它用来定义常量表达式，在文件的其余地方可以引用这些常量表达式。以下代码配置了两个<constant>元素：

```
<global>
    <constant>
        <constant-name>phone</constant-name>
        <constant-value>^\d{8}\d*$</constant-value>
    </constant>
    <constant>
        <constant-name>zip</constant-name>
        <constant-value>^\d{6}\d*$</constant-value>
    </constant>
</global>
```

以上代码定义了两个常量 phone 和 zip，在<formset>元素中可以通过${constant-name}的形式来引用它们，例如访问 phone 常量的形式为${phone}。

3．<formset>元素

<formset>元素包含两个子元素：<constant>和<form>。<constant>可以出现零次或多次，<form>元素可以出现一次或多次。

在<global>中定义的<constant>元素代表全局常量，而在<formset>元素中定义的<constant>元素代表局部常量，后者只能在当前<formset>元素中使用。

<formset>元素有两个属性：language 和 country。language 和 country 属性用于支持 I18N，本章的 10.7 节将对此做介绍。

4．<form>元素

<formset>元素的<form>子元素用于为表单配置验证规则，它的 name 属性指定了表单的名字。<form>元素可以包含一个或多个<field>子元素。

5．<field>元素

<form>元素的<field>子元素用于配置表单中字段的验证规则。例如在例程 10-1 中，对 checkoutForm 表单的 phone 字段配置了验证规则。

表 10-2 对<field>元素的属性做了描述。

表 10-2　<field>元素的属性

属　性	描　述
property	指定 ActionForm Bean 中需要进行验证的字段的名字。
depends	指定字段的验证规则，多个规则之间以逗号隔开。

<field>元素也包含一些子元素，它的 DTD 定义如下：

```
<!element field (msg?, arg0?, arg1?, arg2?, arg3?, var*)>
```

6．<msg>元素

<field>元素的<msg>子元素指定验证规则对应的消息文本。该消息文本将替代在 validator-rules.xml 文件中为验证规则配置的默认的消息文本。例如：

```
<field property="phone" depends="required,mask,minlength">
  <msg name="mask" key="phone.invalidformat"/>
  ......
</field>
```

以上代码的<msg>元素表明，当“mask”验证规则验证失败时，错误消息文本来自于 Resource Bundle，消息 key 为“phone.invalidformat”。

<msg>元素有三个属性，DTD 定义如下：

```
<!ATTLIST msg name      CDATA #IMPLIED
              key       CDATA #IMPLIED
              resource  CDATA #IMPLIED >
```

表 10-3 对<msg>元素的属性做说明。

表 10-3 <msg>元素的属性

属 性	描 述
name	指定验证规则的名字
key	当 resource 属性为 true 时，key 属性指定消息 key，该消息 key 应该在 Resource Bundle 中存在，当 resource 属性为 true 时，key 属性直接指定消息文本
resource	当此项为 true 时，表示使用来自 Resource Bundle 的消息；如果为 false，表示直接在 key 属性中设置消息文本，默认值为 true

7. <arg>元素

<field>元素可以包含四个附加的子元素：<arg0>、<arg1>、<arg2>和<arg3>，用于替换复合消息中的参数。<arg0>元素指定第一个替换值，<arg1>指定第二个值，依次类推。每个 arg 元素包含三个属性：name、key 和 resource，这些属性的用法和<msg>元素很相似。例程 10-2 为包含<arg0>和<arg1>元素的例子：

例程 10-2 包含<arg0>和<arg1>元素的<field>元素

```
<field
  property="phone"
  depends="required,mask,minlength">
  <arg0 key="label.phone"/>
  <arg1 name="minlength" key="${var:minlength}" resource="false"/>
  <var>
    <var-name>mask</var-name>
    <var-value>${phone}</var-value>
  </var>
  <var>
    <var-name>minlength</var-name>
    <var-value>7</var-value>
  </var>
</field>
```

以上代码没有配置<msg>元素，因此当某个验证规则验证失败时，将使用在 validator-rules.xml 中配置的默认消息文本。例如，如果“minlength”验证规则失败，则 validator-rules.xml 中设置的消息 key 为“errors.minlength”：

```
<validator
  name="minLength"
  ......
  msg="errors.minlength">
</validator>
```

在 Resource Bundle 中与“errors.minglength”匹配的中文消息文本为：

```
errors.minlength={0} 不能少于 {1} 字符.
```

<arg0>元素没有设置 name 属性，因此适用于所有验证规则，它将取代以上复合消息中

的第一个参数{0}，<arg0>元素的 key 属性为“label.phone”，在 Resource Bundle 中与之匹配的中文消息文本为：

```
label.phone=电话
```

<arg1>元素的 name 属性为“minlength”，表示仅适用于“minlength”验证规则。它将取代以上复合消息中的第二个参数{1}。<arg1>元素的 resource 属性为 false，表明此时 key 属性直接指定文本。key 属性值由变量${var:minlength}决定，此处值为 7。

当“minlength”验证规则验证失败时，<arg0>和<arg1>元素将分别取代以上复合消息中的两个参数，如图 10-1 所示。

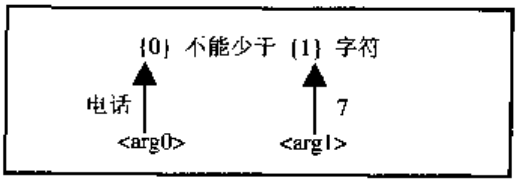

图 10-1　<arg0>和<arg1>元素分别取代复合消息中的两个参数

根据图 10-1 的内容，最后返回的错误消息为：“电话不能少于 7 字符”。

8．<var>元素

<field>元素还可以包含零个或多个<var>元素。<var>元素用来向验证规则传递参数。在 10.1.2 小节的例程 10-2 中，phone 常量被传递给 mask 验证规则：

```
<var>
    <var-name>mask</var-name>
    <var-value>${phone}</var-value>
</var>
```

phone 常量的值为“^\d{8}\d*$”，它定义了一种字符模式，表示字符串长度为 8 位，并且只能包含数字字符。mask 验证规则能够比较用户输入的 phone 字段是否和指定的字符模式匹配。

<arg>元素也可以访问<var>元素，语法为${var:var-name}，例如：

```
<arg1 name="minlength" key="${var:minlength}" resource="false"/>
<var>
    <var-name>minlength</var-name>
    <var-value>7</var-value>
</var>
```

10.1.3　Validator 插件

为了在 Struts 框架中使用 Validator，可以采用插件机制把 Validator 加入到框架中。这需要在 Struts 配置文件中配置 ValidatorPlugIn 插件，代码如下：

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
```

```
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
    </plug-in>
```

当应用启动时，Struts 框架会加载 ValidatorPlugIn 插件并调用它的 init()方法。init()方法根据 pathnames 属性，加载相应的 validator-rules.xml 和 validation.xml 文件，把验证信息读入到内存中。关于 Struts 插件的运行机制和使用方法，可以参考本书 8.1 节（Struts 插件）。

10.2 Validator 框架和 ActionForm

Validator 框架不能用于验证标准的 org.apache.struts.action.ActionForm 类。如果要使用 Validator 框架，应该采用 ActionForm 类的两个子类：org.apache.struts.validator.DynaValidatorForm 和 org.apache.struts.validator.ValidatorForm。ActionForm 类及其子类的类框图如图 10-2 所示。

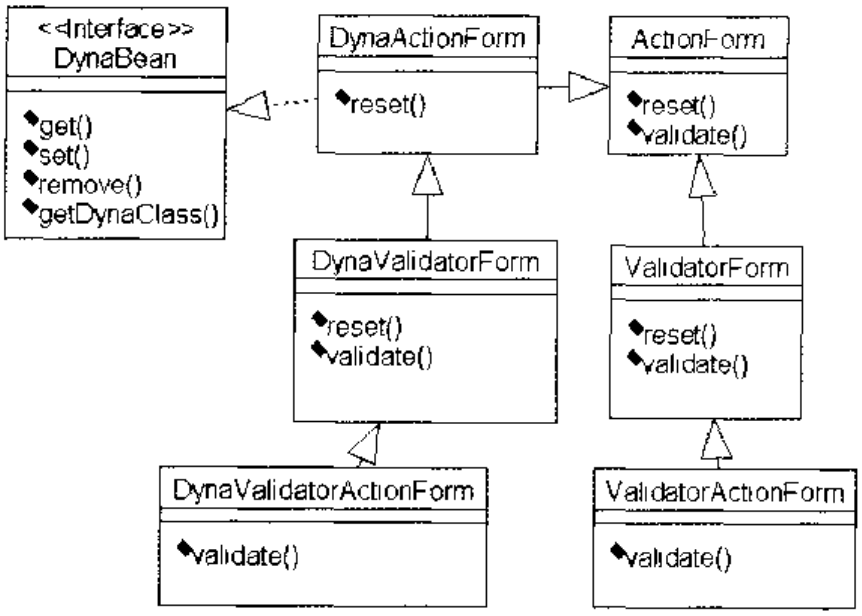

图 10-2 ActionForm 类及其子类的类框图

DynaValidatorForm 支持在动态 ActionForm 中使用 Validator 框架，ValidatorForm 支持在标准 ActionForm 中使用 Validator 框架。无论是对于动态 ActionForm 还是标准 ActionForm，配置 Validator 框架的方式都是一样的。

DynaValidatorForm 和 ValidatorForm 类都实现了 validate()方法，所以当创建它们的子类时，不必再覆盖 validate()方法。ValidatorForm 类的 validate()方法的代码如下：

```
/**
 * Validate the properties that have been set from this HTTP request,
 * and return an <code>ActionErrors</code> object that encapsulates any
 * validation errors that have been found.  If no errors are found, return
 * <code>null</code> or an <code>ActionErrors</code> object with no
 * recorded error messages.
```

```
     *
     * @param mapping The mapping used to select this instance
     * @param request The servlet request we are processing
     * @return   <code>ActionErrors</code> object that encapsulates any   validation errors
     */
    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request) {

        ServletContext application = getServlet().getServletContext();
        ActionErrors errors = new ActionErrors();

        String validationKey = getValidationKey(mapping, request);

        Validator validator = Resources.initValidator(validationKey,
                                this,
                                application, request,
                                errors, page);
        try {
            validatorResults = validator.validate();
        } catch (ValidatorException e) {
            log.error(e.getMessage(), e);
        }
        return errors;
    }
```

以上 validate()方法调用验证框架的验证方法进行验证，如果验证失败，就会创建包含错误消息的 ActionMessage 对象，并把该对象添加到 ActionErrors 集合对象中。validate()方法最后返回 ActionErrors 对象。

ValidatorForm 有一个子类 ValidatorActionForm。如图 10-3 所示为这两种类的区别。

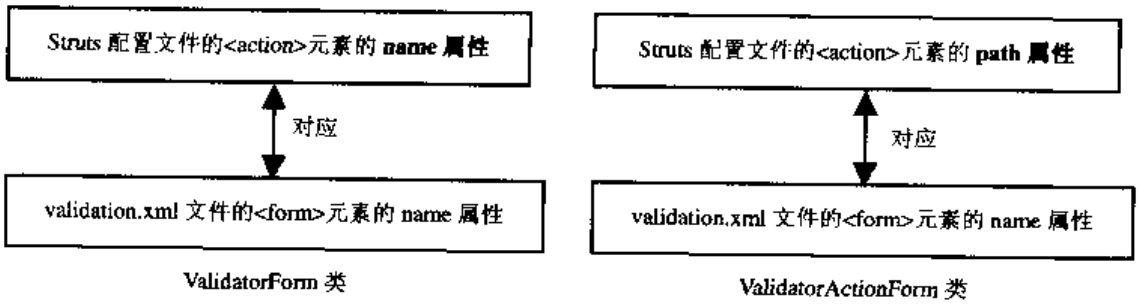

图 10-3　ValidatorForm 和 ValidatorActionForm 类的区别

Validator 框架提供这两种类的目的在于可以更加精确地控制执行验证的条件。例如，假定有个名为“editForm”的表单对应两个 Action：saveAction 和 cancelAction，如图 10-4 所示。

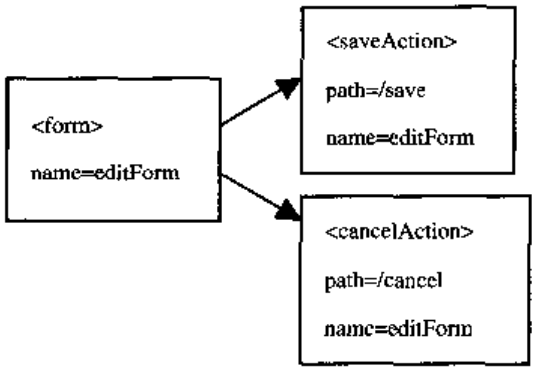

图 10-4 editForm 表单和它的 Action

假定 editForm 表单包含两个验证规则：验证规则 A 和验证规则 B。如果对于用户的保存或取消动作，都要执行两个验证规则，则可以创建一个扩展 ValidatorForm 类的 EditForm 类，然后在 validation.xml 中做如下配置：

```
<formset>
   <form name="editForm">
     验证规则 A
     验证规则 B
   </form>
   ......
</formset>
```

如果对于用户的保存动作，仅执行验证规则 A，对于取消动作，仅执行验证规则 B，则可以创建一个扩展 ValidatorActionForm 类的 EditForm 类，然后在 validation.xml 中做如下配置：

```
<formset>
   <form name="/save">
      验证规则 A
   </form>
   <form name="/cancel">
      验证规则 B
   </form>
   ......
</formset>
```

对于动态 ActionForm，也有 DynaValidatorForm 和 DynaValidatorActionForm 之分，它们的区别与 ValidatorForm 和 ValidatorActionForm 的区别一样。

10.3 Validator 框架和 Struts 客户化标签

根据 10.2 节的内容来看，DynaValidatorForm 和 ValidatorForm 类都实现了 validate()方法，如果验证失败，就会返回包含错误消息的 ActionErrors 对象。

Struts 框架处理 DynaValidatorForm 或 ValidatorForm 类的流程与标准的 ActionForm 一

样，RequestProcessor 调用它们的 validate()方法进行表单验证，如果验证失败，RequestProcessor 就把 validate()方法返回的 ActionErrors 对象保存在 request 范围内。

Struts 的一些客户化标签能够和 Validator 框架协同工作，其中<html:errors>和<html:messages>标签都能够访问 request 范围内的 ActionMessages（或其子类 ActionErrors）对象，向用户显示 Validator 框架生成的验证错误消息，关于这两个标签的详细用法可分别参见本书的 12.6 节（<html:errors>标签）和 12.7 节（<html:messages>标签）。

10.4　在 netstore 应用中使用 Validator 框架

下面看一个使用 Validator 框架的具体的例子。netstore 应用的 checkoutForm 表单用于输入用户的送货地址信息，如图 10-5 所示。

图 10-5　netstore 应用的 checkoutForm 表单

以上表单的所有字段都不允许为空，并且邮编和电话号码必须为数字类型，邮编长度为 6，电话号码长度为 8。本例中采用 DynaValidatorForm 类型的动态 ActionForm，因此无需编写 ActionForm 类的程序代码，只需在 Struts 配置文件中进行如下配置：

```
<form-bean
  name="checkoutForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="name" type="java.lang.String"/>
  <form-property name="address" type="java.lang.String"/>
  <form-property name="city" type="java.lang.String"/>
  <form-property name="state" type="java.lang.String"/>
  <form-property name="postalCode" type="java.lang.String"/>
```

```
    <form-property name="country" type="java.lang.String"/>
    <form-property name="phone" type="java.lang.String"/>
  </form-bean>
```

与上面的 checkoutForm 映射的 Action 有两个：CheckoutAction 和 ProcessCheckoutAction，它们的配置代码如下：

```
<action
      path="/begincheckout"
      name="checkoutForm"
      input="/order/shoppingcart.jsp"
      type="netstore.order.CheckoutAction"
      scope="session"
      validate="false">
      <forward name="Success" path="/order/shipping.jsp"/>
 </action>

<action path="/processcheckout"
   type="netstore.order.ProcessCheckoutAction"
   scope="session"
   input="/order/shipping.jsp"
   name="checkoutForm"
   validate="true"
   parameter="action">
   <forward name="Success" path="/order/payment.jsp"/>
</action>
```

当用户在 shoppingcart.jsp 网页上选择“确认购买”图标时，请求将转发给 CheckoutAction，由于<action>元素的 validate 属性为 false，Struts 框架不会对 checkoutForm 进行表单验证，CheckoutAction 先判断用户是否登入，如果登入，就把请求转发给 shipping.jsp，shipping.jsp 将生成 checkoutForm 表单，如图 10-5 所示。

当用户在 shipping.jsp 网页上单击了【Save Order】或【Check Out】按钮时，请求将转发给 ProcessCheckoutAction，由于此时<action>元素的 validate 属性为 true，Struts 框架会先调用 checkoutForm 类的 validate()方法进行表单验证。

如图 10-6 所示为和 checkoutForm 表单相关的请求处理流程。

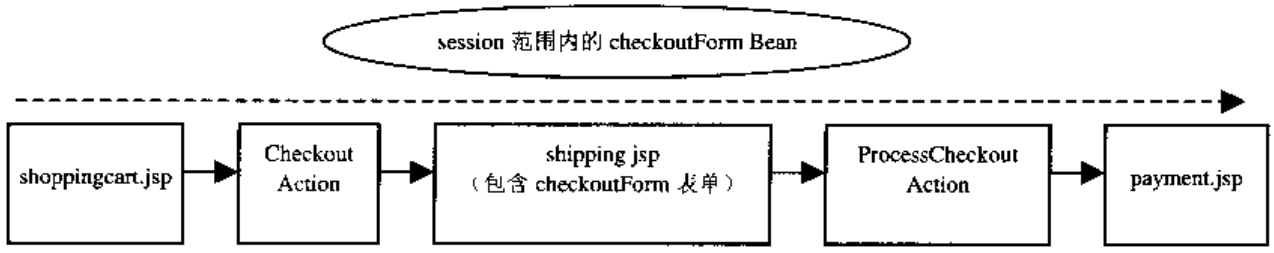

图 10-6 与 checkoutForm 表单相关的请求处理流程

以上两个<action>元素的 scope 属性都为 session，表示把 checkoutForm Bean 存放在 session 范围内。在同一个会话内，一旦用户在 shipping.jsp 网页的 checkoutForm 表单上输入了数据，则再次访问该表单时，表单的<html:form>标签会把 session 范围内 checkoutForm Bean 的数据填充到表单上，表单将显示用户上次输入的数据，这样可以避免用户重复输入表单数据。

下一步是配置 validation.xml 文件。应该为需要进行验证的每个表单字段配置验证规则。在某些情况下，同一个字段可能需要指定多个规则。例程 10-3 是配置 checkoutForm 的验证规则的部分代码。

例程 10-3　validation.xml 文件中配置 checkoutForm 的验证规则的部分代码

```
<form-validation>
    <global>
        <constant>
            <constant-name>phone</constant-name>
            <constant-value>^\d{8}\d*$</constant-value>
        </constant>
        <constant>
            <constant-name>zip</constant-name>
            <constant-value>^\d{6}\d*$</constant-value>
        </constant>
    </global>

    <formset>
        <form name="checkoutForm">
            <field
                property="name"
                depends="required">
                <arg0 key="label.name"/>
            </field>

            <field
                property="phone"
                depends="required,mask,minlength">
                <arg0 key="label.phone"/>
                <arg1 name="minlength" key="${var:minlength}" resource="false"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>${phone}</var-value>
                </var>
                <var>
                    <var-name>minlength</var-name>
                    <var-value>7</var-value>
                </var>
```

```
        </field>
        ......
    </form>
</formset>
</form-validation>
```

此外，应该按照本章 10.1.3 小节的介绍在 Struts 配置文件中配置 ValidatorPlugIn 插件，然后把 Validator 框架使用的默认消息文本添加到应用的 Resource Bundle 中。对于中文用户，应该在资源文件中添加如下中文消息：

```
errors.required={0} 是必须的.
errors.minlength={0} 不能少于 {1} 字符.
errors.maxlength={0} 不能多于 {1} 字符.
errors.invalid={0} 无效.

errors.byte={0} 必须为 byte 类型.
errors.short={0} 必须为 byte 类型.
errors.integer={0} 必须为 byte 类型.
errors.long={0} 必须为 byte 类型.
errors.float={0} 必须为 byte 类型.
errors.double={0} 必须为 byte 类型.

errors.date={0} 不是有效的日期类型.
errors.range={0} 不在范围 {1} 和 {2}之间.
errors.creditcard={0} 不是有效的信用卡号.
errors.email={0} 不是有效的 E-Mail 地址.
```

为 netstore 应用配置好 Validator 框架后，就可以运行 netstore 应用。在如图 10-5 所示的网页上直接提交 checkoutForm 表单，Validator 框架会自动对表单字段进行验证，显示如图 10-7 所示的验证结果。

图 10-7 使用 Validator 框架验证 checkoutForm 表单的结果

10.5　创建自定义的验证规则

Validator 框架预先配置好了一些常用的验证规则，它可以满足多数 struts 应用的需要。如果 Validator 框架提供的默认的验证规则不能满足应用的需要，也可以按以下步骤创建自己的验证规则。

（1）创建包含验证方法的 Java 类。

（2）编辑 validator-rules.xml 文件或者另外创建验证规则的配置文件。如果创建了新的配置文件，在 Struts 配置文件中配置 ValidatorPlugin 插件时，应该在<plug-in>元素的 pathnames 属性中添加这一文件信息。

（3）在 validation.xml 文件中使用新创建的验证规则。

所有的验证方法都应该使用以下的方法签名：

```
public static boolean validateXXX( java.lang.Object,
                                   org.apache.commons.validator.ValidatorAction,
                                   org.apache.commons.validator.Field,
                                   org.apache.struts.action.ActionMessages,
                                   javax.servlet.http.HttpServletRequest,
                                   javax.servlet.ServletContext );
```

验证方法的方法名应该以“validate”开头，表 10-4 对 validateXXX()方法的参数做了解释。

表 10-4　validateXXX()方法的参数

参　数	描　述
Object	指定需要进行数据验证的 JavaBean
ValidatorAction	指定 ValidatorAction 类
Field	指定需要验证的字段
ActionMessages	指定当验证失败时存放 ActionMessage 对象的 ActionMessages 集合对象。
HttpServletRequest	指定当前的 request 对象
ServletContext	指定应用的 ServletContext 对象

在多数情况下，验证方法应该是静态的。当然，也可以定义非静态的验证方法。不管验证方法是否为静态，都应该保证它们可以在多线程环境下安全运行。可以参照 org.apache.struts.validator.FieldChecks 类的源程序来编写自己的验证规则类，例程 10-4 创建了一个新的验证规则类，它可以验证字符串是否可以被转换为有效的 boolean 类型的值。

例程 10-4　NewValidator.java

```
package validate;
```

```
import java.io.Serializable;
import javax.servlet.http.HttpServletRequest;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.util.ValidatorUtils;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.util.RequestUtils;
import org.apache.struts.validator.Resources;

public class NewValidator implements Serializable {
  /**
   * A validate routine that ensures the value is either true or false.
   */
  public static boolean validateBoolean(Object bean,
                                        ValidatorAction va, Field field,
                                        ActionMessages errors,
                                        HttpServletRequest request) {
    String value = null;
    // The boolean value is stored as a String
    if (field.getProperty() != null && field.getProperty().length() > 0){
        value = ValidatorUtils.getValueAsString(bean, field.getProperty() );
    }

    if ( value==null | !(value.equalsIgnoreCase("true") || value.equalsIgnoreCase("false"))){
        errors.add( field.getKey(),
                    Resources.getActionMessage(request, va, field));
    }

     // Return true if the value was successfully converted, false otherwise
     return (errors.isEmpty());
    }
}
```

NewValidator.java 的源程序位于本书配套光盘的 sourcecode/chapter10/src 目录下。接下来把这个规则加入到 validator-rules.xml 文件中，或者把它放在单独的专门存放客户化规则的配置文件中。以下是配置“boolean”规则的代码：

```
<validator name="boolean"
  classname="validate.NewValidator"
  method="validateBoolean"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
```

```
    org.apache.commons.validator.field,
    org.apache.struts.action.ActionMessages,
    javax.servlet.http.HttpServletRequest"
  msg="errors.boolean">
```

<validator>元素的 msg 属性为“errors.boolean”，应该在 Resource Bundle 中提供与之匹配的消息文本：

```
errors.boolean={0} must be a boolean.
```

最后，可以在 validation.xml 文件使用“boolean”规则。例如以下代码为某个表单的 sendEmailConfirmation 字段配置了“boolean”规则：

```
<field property="sendEmailConfirmation" depends="boolean">
    <arg0 key="label.emailconfirmation"/>
</field>
```

10.6　在 Validator 框架中使用 JavaScript

在默认情况下，Validator 框架在 Web 服务器端执行表单验证，事实上，Validator 框架也可以进行浏览器客户端验证。如果要进行客户端验证，需要用到 Struts 的<html:javascript>标签，它能够在 JSP 网页中生成用于客户端验证的 JavaScript 脚本。以下是实现 Validator 框架客户端验证的步骤。

（1）在 validator-rules.xml 文件的<validator>元素中配置<javascript>元素：

```
<validator
  name="required"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateRequired"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.field,
    org.apache.struts.action.ActionMessages,
    javax.servlet.http.HttpServletRequest"
  msg="errors.required">
  <javascript><![CDATA[
    function validateRequired(form) {
      var bvalid = true;
      var focusField = null;
      var i = 0;
      var fields = new Array();
      oRequired = new required();
```

```
        for (x in oRequired) {
        if ((form[oRequired[x][0]].type == 'text' ||
              form[oRequired[x][0]].type ==       'textarea' ||
              form[oRequired[x][0]].type == 'select-one' ||
              form[oRequired[x][0]].type == 'radio' ||
              form[oRequired[x][0]].type == 'password') &&
              form[oRequired[x][0]].value == '') {
               if (i == 0)
                 focusField = form[oRequired[x][0]];
                 fileds[i++] = oRequired[x][1];
                 bvalid = false;
              }
        }

        if (fields.length > 0) {
           focusField.focus();
           alert(fields.join('\n'));
        }
        return bvalid;
     }}}>
     </javascript>
  </validator>
```

在 Struts 框架默认的 validator-rules.xml 文件中已经包含了以上<javascript>子元素，通常可以使用这个默认配置。

（2）在 JSP 页面中包含<html:javascript>标签：

```
<html:javascript formName="checkoutForm"/>
```

<html:javascript>标签的 formName 属性指定需要验证的表单的名字。该标签能够访问为表单配置的验证规则包含的<javascript>元素，把它包含的脚本写到 JSP 网页中，生成 validateXXX()的函数，XXX 代表表单的名字。例如，如果表单名为 checkoutForm，<html:javascript>标签将创建名为 validateCheckoutForm()的函数，该函数负责执行验证逻辑。

以下是<html:javascript>标签在 JSP 网页中生成的 JavaScript 脚本的部分代码：

```
<script type="text/javascript" language="Javascript1.1">

<!-- Begin
      var bCancel = false;
      function validateCheckoutForm(form) {
         if (bCancel)
            return true;
         else
         var formValidationResult;
         formValidationResult = validateRequired(form)
                                    && validateMask(form)
                                    && validateMinLength(form);
```

```
    return (formValidationResult == 1);
}
.....
function validateMinLength(form) {
        var bValid = true;
        var focusField = null;
        var i = 0;
        var fields = new Array();
        oMinLength = new minlength();
        for (x in oMinLength) {
                if (form[oMinLength[x][0]].type == 'text' ||
                    form[oMinLength[x][0]].type == 'textarea') {
                    var iMin = parseInt(oMinLength[x][2]("minlength"));
                    if (form[oMinLength[x][0]].value.length < iMin) {
                        if (i == 0) {
                            focusField = form[oMinLength[x][0]];
                        }
                        fields[i++] = oMinLength[x][1];
                        bValid = false;
                    }
                }
        }
        if (fields.length > 0) {
                focusField.focus();
                alert(fields.join('\n'));
        }
        return bValid;
    }
//End -->
</script>
```

（3）对需要验证的表单，应该定义 onsubmit 事件：

```
<html:form action="processcheckout" onsubmit="return validateCheckoutForm(this);">
```

当用户在表单上按下提交按钮后，就会调用<html:javascript>标签生成的 JavaScript 脚本的 validateCheckoutForm()函数，执行客户端验证。如果验证失败，就会在 IE 浏览器弹出的对话框中向用户显示错误消息，表单不会被提交到服务器端。

本书配套光盘中提供的 netstore 应用的 checkoutForm 表单默认情况下采用服务器端验证，如果需要改为客户端验证，应该按照以上方法修改 shipping.jsp 文件，在文件中添加<html:javascript>标签：

```
<html:javascript formName="checkoutForm"/>
```

10.7 Validator 框架的国际化

Validator 框架提供了对 I18N 的支持。无论是客户端验证，还是服务器端验证，Validator 框架都从应用的 Resource Bundle 中获得错误消息。validation.xml 的<formset>元素包含了 language 和 country 属性，这两个属性可以设置 Locale。如果没有设置这两个属性，<formset>元素使用默认的 Locale。

如果对于不同的 Locale 需要采用不同的验证规则，那么应该在 validation.xml 文件中针对每个 Locale 创建独立的<formset>元素。例如，假定某应用的 registrationForm 表单的 userName 字段通常必须满足三个验证规则："required"、"mask" 和 "minLength"，而对于中文 Locale，userName 字段仅需满足 "required" 验证规则，那么应该对这个表单配置两个<formset>元素：

```
<formset>
  <form name="registrationForm">
    <field
      property="userName"
      depends="required,mask,minLength">
      <arg0 key="registrationForm.firstname.displayname"/>
      <var>
        <var-name>mask</var-name>
        <var-value>^\w+$</var-value>
      </var>
      <var>
        <var-name>minLength</var-name>
        <var-value>5</var-value>
      </var>
    </field>
  </form>
</formset>
<formset language="zh">
  <form name="registrationForm">
    <field
      property="userName"
      depends="required">
      <arg0 key="registrationForm.firstname.displayname"/>
    </field>
  </form>
</formset>
```

10.8　小　　结

本章介绍了 Validator 框架的原理和使用方法，Validator 框架采用两个配置文件来为表单配置验证规则。下面再来总结一下在 Struts 应用中使用 Validator 框架的步骤。

（1）创建扩展 ValidatorForm 或 ValidatorActionForm 类的 ActionForm 类，如果使用动态 ActionForm，则无需创建扩展 DynaValidatorForm 或 DynaValidatorActionForm 类的子类，可以直接进行第二步。

（2）在 Struts 配置文件中配置<form>和<action>元素。

（3）把 Validator 框架使用的消息文本添加到应用的 Resource Bundle 中。

（4）在 validation.xml 文件中为表单配置验证规则。

（5）在 Struts 配置文件中配置 ValidatorPlugIn 插件。

第11章 异常处理

当Java虚拟机（JVM）在执行应用程序的某个方法时遇到了非正常现象，JVM就会生成一个异常对象，把它抛给客户，以便向客户通报程序在运行中出了问题，这里的客户可以是上层调用方法，或者是终端用户。合理地处理异常，可以使应用更加健壮。

Struts框架处理异常的机制建立在Java处理异常的基础之上。本章先介绍了JVM处理异常的原理，然后介绍了Struts框架处理异常的机制。最后通过一个Struts应用的例子exsample应用，来讲解在Struts应用中处理异常的各种方式。exsample应用的主页index.jsp如图11-1所示，它提供了访问其他样例网页的链接，这些样例网页用于演示处理异常的各种方法。

图11-1 exsample应用的主页index.jsp

11.1 Java异常处理

在介绍Struts异常处理机制之前，先回顾一下Java异常处理原理，理解Java虚拟机（JVM）的异常处理过程有助于为应用设计正确的异常处理方案。处理异常需要JVM付出额外的负载，因此用户必须谨慎地处理异常。

11.1.1 Java异常

Java异常是在Java程序运行时遇到非正常情况而创建的对象，它封装了异常信息。Java异常的根类为java.lang.Throwable。Throwable类有两个直接子类：java.lang.Error和java.lang.Exception。Error类表示程序本身无法恢复的严重错误，如Java虚拟机错误。Exception类表示可以被程序捕获并处理的错误。在Struts应用中自定义的异常必须直接或间接继承Exception类。如图11-2所示为是主要的Java异常类的类框图。

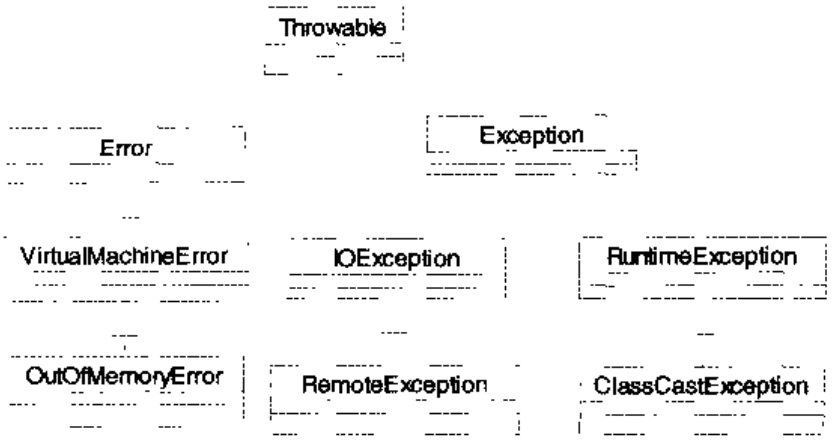

图 11-2 主要 Java 异常类的类框图

11.1.2 JVM 的方法调用堆栈

JVM 用方法调用栈（method invocation stack）来跟踪每个线程中一系列的方法调用过程。该栈保存了每个调用方法的本地信息。对于独立 Java 应用程序，可以一直回溯到该程序的入口方法 main()。当一个新方法被调用时，JVM 把描述该方法的栈结构置入栈顶，位于栈顶的方法为正在执行的方法。如图 11-3 所示描述了方法调用栈的结构。在图 11-3 中，方法的调用顺序为：main()方法调用 methodB()方法，methodB()方法调用 methodA()方法。

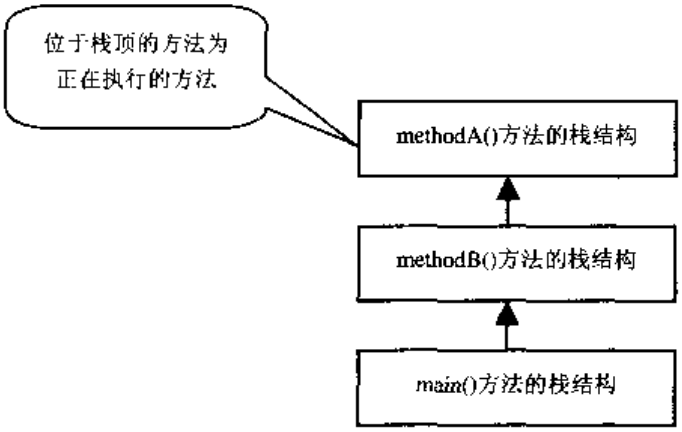

图 11-3 JVM 的方法调用栈

如果方法中的代码块可能抛出异常，那么有两种处理办法：

- 在当前方法中通过 try/catch 语句捕获并处理异常，例如：

```
public void methodA(int a){
  try{
    //some codes which might throw Exception
    if(a<0) throw new SpecialException();
  }catch(SpecialException e){
   //deal with Exception
```

```
    }
}
```

- 在方法的声明处通过 throws 语句声明抛出异常，例如：

```
public void methodA(int a) throws SpecialException{
  //some codes which might throw Exception
  if(a<0) throw new SpecialException();
}
```

当一个 Java 方法正常执行完毕，JVM 会从调用栈中弹出该方法的栈结构，然后继续处理前一个方法。如果 Java 方法在执行代码过程中抛出异常，JVM 必须找到能捕获该异常的 catch 代码块。它首先查看当前方法是否存在这样的 catch 代码块，如果存在，就执行该 catch 代码块；否则，JVM 会从调用栈中弹出该方法的栈结构，继续到前一个方法中查找合适的 catch 代码块。

例如，当 methodA()方法抛出 SpecialException 时，如果在该方法中提供了捕获 SpecialException 的 catch 代码块，就执行这个异常处理代码块。如果 methodA()方法未捕获异常，而是采用第二种方式声明继续抛出 SpecialException，JVM 的处理流程将退回到上层调用方法 methodB()，再查看 methodB()方法有没有捕获 SpecialException。如果在 methodB()方法中存在捕获 SpecialException 的 catch 代码块，就执行该 catch 代码块。此时 methodB()方法的定义如下：

```
public void methodB(int a) {
  try{
      methodA(a);
  }catch(SpecialException e){
    //deal with Exception
  }
}
```

如果 methodB()方法也没有捕获 SpecialException，而是声明抛出 SpecialException，JVM 的处理流程将退回到 main()方法。此时 methodB()方法的定义如下：

```
public void methodB(int a) throws SpecialException{
    methodA(a);
}
```

最后，如果 JVM 向上追溯到 main()方法，仍然没有找到处理该异常的代码块，该线程就会异常终止。如果该线程是主线程，应用程序也将随之终止，此时 JVM 将把异常直接抛给用户，在用户终端上会看到原始的异常信息。

在上述回溯过程中，如果 JVM 在某个方法中找到了处理该异常的代码块，则该方法的栈结构将成为栈顶元素，程序流程将转到该方法的异常处理代码部分继续执行。

11.1.3 异常处理对性能的影响

一般说来，在 Java 程序中使用 try/catch 语句不会对应用的性能造成很大的影响。仅仅

当异常发生时，JVM 虚拟机需要执行额外的操作，来定位处理异常的代码块，这时会对性能产生负面影响。如果抛出异常的代码块和捕获异常的代码块位于同一个方法中，这种影响就会小一些；如果 JVM 必须搜索方法调用堆栈来寻找异常处理代码块，其对性能的影响就比较大了。尤其是在异常处理代码块位于调用栈的最底部时，JVM 定位异常处理代码块需要做大量的工作。

因此不应该使用异常处理机制来控制程序的正常流程，而应该确保仅仅在程序中可能出现异常的地方使用 try/catch 语句。此外，应该使异常处理代码块位于适当的层次，如果当前方法具备处理可能发生的某种异常的能力，就尽量自行处理，不要把自己可以处理的异常推让给上层调用方法去处理。

11.1.4 系统异常和应用异常

从开发应用的角度来看，可以把异常分为系统异常和应用异常。系统异常在性质上比应用异常更加严重，前者通常和应用逻辑无关，而是底层系统出现了问题，如数据库服务器异常终止，网络连接中断或者应用软件自身存在缺陷。终端用户不能修复这种错误，而需要通知系统管理员或者软件开发人员来处理。

应用异常是由于违反了商业规则或者业务逻辑而导致的错误。例如，一个被锁定的用户试图登入应用。这种错误不是致命的错误，可以把错误信息报告给用户，让用户进行相应的处理。

在 Struts 应用中，针对不同的异常，可以采取不同的措施。如果是用户能够恢复的应用异常，那么可以把控制流程转回到用户输入页面，向用户友好地汇报所出现的问题以及应该采取的恢复措施。如果是系统异常，可以显示一个系统出错页面，建议用户让系统管理员来解决这个问题。

在处理异常时，要注意不能让终端用户看到原始的 Java 异常信息。如果不懂 Java 技术的终端用户看到一堆 Java 异常的堆栈跟踪信息，只会感到困惑不解。所以，应该先对原始的 Java 异常进行包装，然后向用户显示容易理解的错误信息。另外，可以把原始的 Java 异常信息记入日志文件，以帮助系统管理员或软件开发者找到出错的根本原因。

11.1.5 使用异常链

当不需要用户来处理和关心原始的异常时，常见的做法是捕获原始的异常，把它包装成为一个新的不同类型的异常，再抛出新的异常。例如，假设一个 Struts 应用需要把一个图像文件上传到数据库中，Action 类调用如下的 uploadImageFile()方法：

```
public void uploadImageFile( String imagePath ) throws UploadException{
    try{
        //upload the image file
        ......
    }catch(IOException e){
        //log the original exception
        ......
```

```
            throw new UploadException();
        }catch(SQLException){
            //log the original exception
            ......
            throw new UploadException();
        }
    }
```

uploadImageFile()方法执行上传图像文件操作时，可能会捕获 IOException 或者 SQLException。但是用户没有必要关心错误的底层细节，他们只需要知道上传图像失败，具体的调试和排错由系统管理员或者软件开发人员来处理。因此，uploadImageFile()方法捕获到原始的异常后，在 catch 代码块中先把原始的异常记入日志，然后向用户抛出 UploadException 异常。

JDK 1.4 版本中的 Throwable 类支持异常链机制。所谓异常链就是在把原始异常包装为新的异常类时，在新的异常类中封装了原始异常类，这有助于查找产生异常的根本原因。此外，也可以由开发者自行设计支持异常链的异常类，例程 11-1 提供了一种实现方案。

例程 11-1　支持异常链的异常类 BaseException.java

```
import java.io.PrintStream;
import java.io.PrintWriter;
/**
 *  This is the common superclass for all application exceptions. This
 *  class and its subclasses support the chained exception facility that allows
 *  a root cause Throwable to be wrapped by this class or one of its
 *  descendants.
 */
public class BaseException extends Exception {
  protected Throwable rootCause = null;

  protected BaseException( Throwable rootCause ) {
    this.rootCause = rootCause;
  }

  public void setRootCause(Throwable anException) {
    rootCause = anException;
  }

  public Throwable getRootCause() {
    return rootCause;
  }

  public void printStackTrace() {
    printStackTrace(System.err);
  }
```

```
    public void printStackTrace(PrintStream outStream) {
      printStackTrace(new PrintWriter(outStream));
    }

    public void printStackTrace(PrintWriter writer) {
      super.printStackTrace(writer);

      if ( getRootCause() != null ) {
        getRootCause().printStackTrace(writer);
      }
      writer.flush();
    }
  }
```

在 BaseException 中定义了 Throwable 类型的 rootCause 变量，它用于保存原始的 Java 异常。假定 UploadException 类扩展了 BaseException 类，以下是把 IOException 包装为 UploadException 的代码：

```
try{
      //upload the image file
      ......
}catch(IOException e){
     //log the original exception
     ......
     UploadException   ue= new UploadException();
     ue.setRootCause(e);
     throw ue;
}
```

11.1.6 处理多样化异常

与异常链相似的一个概念是多样化异常。在实际应用中，有时需要一个方法同时抛出多个异常。例如，假定用户提交的 HTML 表单上有多个字段域，业务规则要求每个字段域的值都符合特定规则，并且假定对这些字段域的验证是在 Web 服务器端执行的。

如果应用不支持在一个方法中同时抛出多个异常，用户每次将只能看到针对一个字段域的验证错误。当改正了一个错误后，重新提交表单，又会收到针对另一个字段域的验证错误，这会令用户很烦恼。

有效的做法是每次当用户提交表单后，都验证所有的字段域，然后向用户显示所有的验证错误信息。不幸的是，在 Java 方法中一次只能抛出一个 Throwable 实例。因此需要开发者自行设计支持多样化异常的异常类。例程 11-2 提供了一种实现方案。

例程 11-2　支持多样化异常的异常类 BaseException.java

```
import java.util.List;
import java.util.ArrayList;
```

```
import java.io.PrintStream;
import java.io.PrintWriter;
/**
 *  This is the common superclass for all application exceptions. This
 *  class and its subclasses support the chained exception facility that allows
 *  a root cause Throwable to be wrapped by this class or one of its
 *  descendants. This class also supports multiple exceptions via the
 *  exceptionList field.
 */
public class BaseException extends Exception{

  protected Throwable rootCause = null;
  private List exceptions = new ArrayList();

  public BaseException(){
    super();
  }

  public BaseException( Throwable rootCause ) {
    this.rootCause = rootCause;
  }

  public List getExceptions() {
    return exceptions;
  }

  public void addException( BaseException ex ){
    exceptions.add( ex );
  }

  public void setRootCause(Throwable anException) {
    rootCause = anException;
  }

  public Throwable getRootCause() {
    return rootCause;
  }

  public void printStackTrace() {
    printStackTrace(System.err);
  }

  public void printStackTrace(PrintStream outStream) {
    printStackTrace(new PrintWriter(outStream));
  }
```

```
    public void printStackTrace(PrintWriter writer) {
      super.printStackTrace(writer);

      if ( getRootCause() != null ) {
        getRootCause().printStackTrace(writer);
      }
      writer.flush();
    }
  }
```

BaseException 类包含一个 List 类型的 exceptions 变量，用来存放其他的 Exception。以下代码显示了 BaseException 的用法：

```
public void check() throws BaseException{
  BaseException be=new BaseException();
  try{
    checkField1();
  }catch(Field1Exception e){be.addException(e);}
  try{
    checkField2();
  }catch(Field2Exception e){be.addException(e);}
  throw be;
}
```

11.2 Struts 框架异常处理机制概述

Struts 框架在视图层和控制层提供了对异常处理的支持。如图 11-4 所示为 Struts 框架处理异常的主要流程。

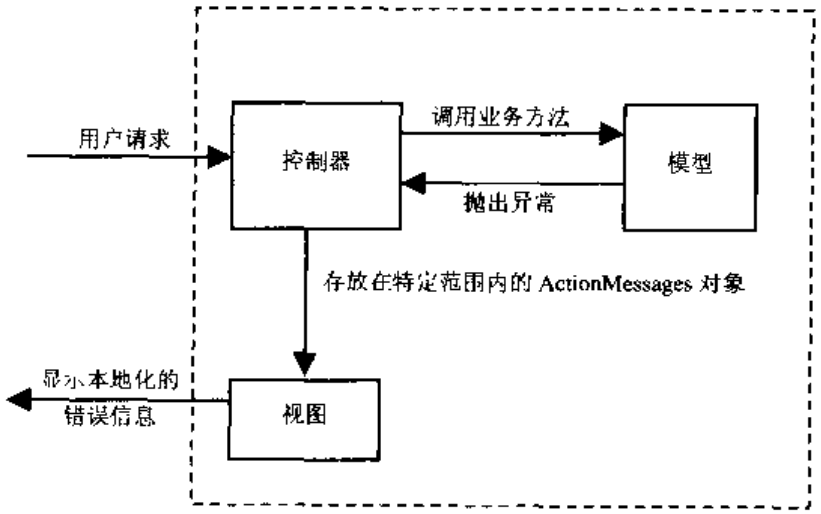

图 11-4 Struts 框架处理异常的流程

Struts 的控制器负责捕获各种异常，包括控制器运行中本身抛出的异常，以及调用模型的业务方法时抛出的异常。当 Struts 的控制器捕获到异常后，在异常处理代码块中，创建

描述异常信息的 ActionMessage 对象把它保存在 ActionMessages（或其子类 ActionErrors）对象中，然后把 ActionMessages 保存在特定范围（request 范围或 session 范围）内。接下来，视图层的<html:errors>标签检索特定范围内的 ActionMessages 对象，把本地化的错误消息输出到网页上。

Struts 框架的这种异常处理机制可以避免直接向用户显示原始的 Java 异常信息，而是在控制层对 Java 异常进行重新包装，在视图层提供能够让用户理解的错误信息。此外，Struts 框架向用户显示的错误消息来自于 Resource Bundle，这有助于实现异常处理的国际化。

11.3　Struts 框架异常处理机制的细节

Struts 框架处理异常是以 JVM 的异常处理机制为基础的。要深入理解 Struts 框架异常处理的细节，首先看一下 Struts 应用在响应用户请求时的方法调用过程，参见图 11-5。

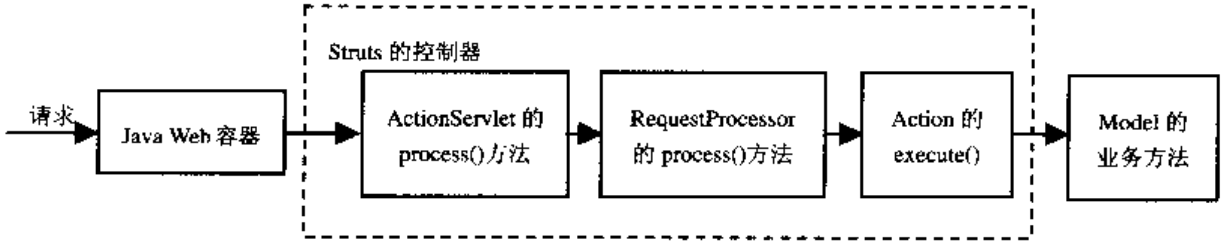

图 11-5　Struts 应用响应用户请求时的方法调用过程

11.3.1　Java Web 容器处理异常的机制

尽管 Struts 框架提供了功能强大的通用错误处理机制，但不能保证捕获所有的错误或异常。当错误发生时，如果 Struts 框架不能处理这种错误，就把错误抛给 Java Web 容器。Java Web 容器先查看是否在 Web 应用发布描述文件中配置了相应的<error-page>元素，如果存在该元素，就返回<error-page>元素的<location>子元素指定的错误页面；否则就会把错误直接抛给用户。关于<error-page>元素的用法参见本书的 4.2.4 小节（配置错误处理）。

11.3.2　ActionServlet 类处理异常的机制

ActionServlet 类的 process()方法不捕获任何异常，仅仅声明向上层调用方法抛出 IOException 或 ServletException。以下是 process()方法的代码：

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

    ModuleUtils.getInstance().selectModule(request, getServletContext());
    getRequestProcessor(getModuleConfig(request)).process(request, response);
}
```

11.3.3 RequestProcessor 类处理异常的机制

RequestProcessor 类是 Struts 框架处理异常的核心组件。其 process()方法不捕获任何异常，仅仅声明向上层调用方法抛出 IOException 或 ServletException。以下是 process()方法的部分代码：

```
public void process(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    ......
    // Call the Action instance itself
    ActionForward forward =processActionPerform(request, response,
                    action, form, mapping);
    ......
}
```

RequestProcessor 类的 process()方法调用自身的 processActionPerform()方法，该方法再调用 Action 类的 execute()方法。以下是 processActionPerform()方法的代码：

```
protected ActionForward  processActionPerform(HttpServletRequest request,
                    HttpServletResponse response,
                    Action action,
                    ActionForm form,
                    ActionMapping mapping)
    throws IOException, ServletException {
    try {
        return (action.execute(mapping, form, request, response));
    } catch (Exception e) {
        return (processException(request, response, e,form,mapping));
    }
}
```

从以上代码中可以看出，processActionPerform()方法能够捕获 Action 类的 execute()方法抛出的所有异常。在它的 catch 代码块中，调用自身的 processException()方法来处理异常。以下是 processException()方法的代码：

```
protected ActionForward processException(HttpServletRequest request,
                    HttpServletResponse response,
                    Exception exception,
                    ActionForm form,
                    ActionMapping mapping)
    throws IOException, ServletException {
    // Is there a defined handler for this exception?
    ExceptionConfig config = mapping.findException(exception.getClass());
    if (config == null) {
```

```
                log.warn(getInternal().getMessage("unhandledException",
                                                  exception.getClass()));
                if (exception instanceof IOException) {
                    throw (IOException) exception;
                } else if (exception instanceof ServletException) {
                    throw (ServletException) exception;
                } else {
                    throw new ServletException(exception);
                }
            }
            // Use the configured exception handling
            try {
                ExceptionHandler handler = (ExceptionHandler)
                RequestUtils.applicationInstance(config.getHandler());
                return (handler.execute(exception, config, mapping, form,
                                        request, response));
            } catch (Exception e) {
                throw new ServletException(e);
            }
        }
```

在 processException()方法中，先查看是否存在 ExceptionConfig 对象：

```
ExceptionConfig config = mapping.findException(exception.getClass());
```

ExceptionConfig 和 Struts 配置文件中的异常配置元素<exception>对应，ExceptionConfig 对象中封装了<exception>元素的配置信息。<exception>元素的配置方法参见本章的 11.4.3 小节。

如果不存在 ExceptionConfig 对象，就继续向上层调用方法抛出异常。如果找到了 ExceptionConfig 对象，就调用它的 getHandler()方法获得异常处理类 ExceptionHandler，并创建异常处理类的实例：

```
ExceptionHandler handler = (ExceptionHandler)
        RequestUtils.applicationInstance(config.getHandler());
```

最后调用 ExceptionHandler 类的 execute()方法，来处理异常：

```
handler.execute(exception, config, mapping, form,request,response));
```

11.3.4 ExceptionHandler 类处理异常的机制

Struts 框架提供了默认的异常处理类 org.apache.struts.action.ExceptionHandler，它的 execute()方法负责处理异常，以下是 execute()方法的代码：

```
public ActionForward execute(
    Exception ex,
```

```
                ExceptionConfig ae,
                ActionMapping mapping,
                ActionForm formInstance,
                HttpServletRequest request,
                HttpServletResponse response)
                throws ServletException {

                ActionForward forward = null;
                ActionMessage error = null;
                String property = null;

                // Build the forward from the exception mapping if it exists
                // or from the form input
                if (ae.getPath() != null) {
                    forward = new ActionForward(ae.getPath());
                } else {
                    forward = mapping.getInputForward();
                }

                // Figure out the error
                if (ex instanceof ModuleException) {
                    error = ((ModuleException) ex).getActionMessage();
                    property = ((ModuleException) ex).getProperty();
                } else {
                    error = new ActionMessage(ae.getKey(), ex.getMessage());
                    property = error.getKey();
                }

                this.logException(ex);

                // Store the exception
                request.setAttribute(Globals.EXCEPTION_KEY, ex);
                this.storeException(request, property, error, forward, ae.getScope());

                return forward;
            }
```

ExceptionHandler 类的 execute()方法首先决定转发路径：

```
if (ae.getPath() != null) {
    forward = new ActionForward(ae.getPath());
} else {
    forward = mapping.getInputForward();
}
```

接着把异常信息包装到 ActionMessage 对象中：

```
// Figure out the error
if (ex instanceof ModuleException) {
    error = ((ModuleException) ex).getActionMessage();
    property = ((ModuleException) ex).getProperty();
} else {
```

```
            error = new ActionMessage(ae.getKey(), ex.getMessage());
            property = error.getKey();
        }
```

然后调用自身的 storeException()方法。storeException()方法负责把 ActionMessage 对象保存在 ActionMessages 对象中，再把 ActionMessages 对象存放在适当的范围内。以下是 storeException()方法的代码：

```
protected void storeException(
        HttpServletRequest request,
        String property,
        ActionMessage error,
        ActionForward forward,
        String scope) {

        ActionMessages errors = new ActionMessages();
        errors.add(property, error);

        if ("request".equals(scope)) {
            request.setAttribute(Globals.ERROR_KEY, errors);
        } else {
            request.getSession().setAttribute(Globals.ERROR_KEY, errors);
        }
    }
```

如图 11-6 所示为当 Action 的 execute()方法抛出异常时，Struts 框架处理异常的时序图。

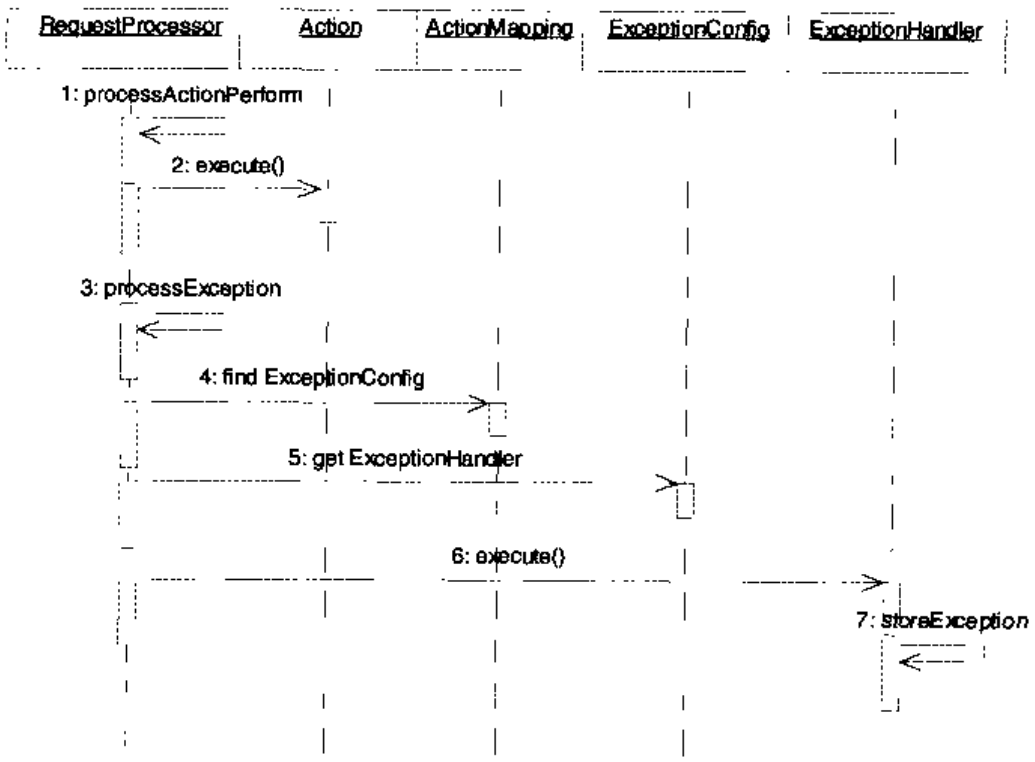

图 11-6　Struts 框架处理异常的时序图

11.4 在 Struts 应用中处理异常的各种方式

本节通过 exsample 应用，来介绍在 Struts 应用中处理异常的各种方式。在 exsample 应用中提供了 5 个 Action 类，它们都调用 Model 类的 businessMethod()方法。Model 类代表模型组件，它的 businessMethod()方法的代码如下：

```
public void businessMethod() throws BaseException{
        BaseException be=new BaseException();
        be.setMessageKey("error.business.error");

        RuleDisobeyException ex1=new RuleDisobeyException();
        ex1.setMessageKey("error.rule.disobey");
        ex1.setMessageArgs(new String[]{"1.","One"} );
        be.addException(ex1);

        RuleDisobeyException ex2=new RuleDisobeyException();
        ex2.setMessageKey("error.rule.disobey");
        ex2.setMessageArgs(new String[]{"2.","Two"} );
        be.addException(ex2);

        throw be;
    }
```

businessMethod()方法代表实际的业务方法。为了演示如何处理异常，在该方法中仅仅抛出一个 BaseException 异常。BaseException 对象中还包含了两个 RuleDisobeyException 对象。RuleDisobeyException 类是 BaseException 类的子类。关于 BaseException 类的实现请参见 11.4.1 小节。

11.4.1 创建异常类

在 Struts 的控制层，将把 Java 异常类包装为 ActionMessage，视图从 ActionMessage 对象中读取错误信息。如果异常类包含的异常消息事先就和 Struts 的 Resurce Bundle 绑定在一起，可以简化 Struts 控制层组件把 Java 异常类包装为 ActionMessage 的步骤。在 Struts 应用中定义异常有两种方式：

- 扩展 ModuleException 类
- 创建自定义的异常类体系

1. 扩展 ModuleException 类

在 Struts API 中提供了专门的异常类 org.apache.struts.util.ModuleException，它是 Exception 类的子类。它的优点在于可以很好地和 Struts 的 Resource Bundle 绑定。ModuleException 有以下构造方法：

```
ModuleException(java.lang.String key)
```

ModuleException(java.lang.String key, java.lang.Object[] values)

其中参数 key 指定错误消息 key，与 Resource Bundle 中的消息 key 匹配，对于复合式消息，可以设置对象数组类型的 values 参数，它用来替换复合式消息中的参数。

对于用户定义的异常类，只要扩展了 ModuleException 类，就可以继承 ModuleException 类的优点。

ModuleException 的缺点在于它是 Struts 的控制层组件，不能被模型层使用。根据模型层应该不依赖于控制层的原则，在模型层中不能引入 Struts API 中的类。所以，对于同一种类型的异常，必须在模型层和控制层中创建两个独立的异常类，这样势必导致重复代码。而且，在 Action 类的 execute()方法中，如果捕获了来自模型层的异常，要把它先包装成为 ModuleException，才能继续把它抛出，例如：

```
try{
  //call Model's business method
.....
}catch(SpecialModelException e){
    throw new SpecialModuleException();
}
```

以上代码中 SpecialModelException 代表模型层的异常，SpecialModuleException 代表扩展了 ModuleException 的供控制层使用的异常。尽管这两个异常类都描述同一种错误，但是为了不违背 MVC 的分层原则，必须为它们分别编写程序代码。

2. 创建自定义的异常类体系

为了克服以上使用 ModuleException 的弊端，可以创建自己的异常类体系，模型层和控制层能够共用这些异常类。例程 11-3 提供了自定义的异常类的根类的一种实现方式。

例程 11-3 BaseException.java

```
package exsample;

import java.util.List;
import java.util.ArrayList;
import java.io.PrintStream;
import java.io.PrintWriter;
/**
 *  This is the common superclass for all application exceptions. This
 *  class and its subclasses support the chained exception facility that allows
 *  a root cause Throwable to be wrapped by this class or one of its
 *  descendants. This class also supports multiple exceptions via the
 *  exceptionList field.
 */
public class BaseException extends Exception{

  protected Throwable rootCause = null;
  private List exceptions = new ArrayList();
```

```
private String messageKey = null;
private Object[] messageArgs = null;

public BaseException(){
   super();
}

public BaseException( Throwable rootCause ) {
   this.rootCause = rootCause;
}

public List getExceptions() {
   return exceptions;
}

public void addException( BaseException ex ){
   exceptions.add( ex );
}

public void setMessageKey( String key ){
   this.messageKey = key;
}

public String getMessageKey(){
   return messageKey;
}

public void setMessageArgs( Object[] args ){
   this.messageArgs = args;
}

public Object[] getMessageArgs(){
   return messageArgs;
}

public void setRootCause(Throwable anException) {
   rootCause = anException;
}

public Throwable getRootCause() {
   return rootCause;
}

public void printStackTrace() {
   printStackTrace(System.err);
}
```

```
    public void printStackTrace(PrintStream outStream) {
      printStackTrace(new PrintWriter(outStream));
    }

    public void printStackTrace(PrintWriter writer) {
      super.printStackTrace(writer);

      if ( getRootCause() != null ) {
        getRootCause().printStackTrace(writer);
      }
      writer.flush();
    }
  }
```

以上 BaseException 类定义了如下属性：

```
protected Throwable rootCause = null;
private List exceptions = new ArrayList();
private String messageKey = null;
private Object[] messageArgs = null;
```

这些属性使得 BaseException 类具有以下功能：

- 支持异常的级联，rootCause 属性指定原始异常。
- 支持多样化异常，exceptions 属性存放所有嵌套的异常。
- 支持和 Struts 的 Resouce Bundle 绑定，messageKey 属性指定消息 key。
- 支持复合式消息，messageArgs 属性指定复合式消息中的参数。

11.4.2 由 Java Web 容器捕获异常

根据 Java Servlet 规范，当 Java Web 容器捕获到 Web 应用抛出的异常时，将查看是否在 web.xml 中配置了相应的<error-page>元素，如果存在<error-page>元素，就会返回其<location>子元素指定的错误网页。例如，在 exsample 应用的 web.xml 文件中配置了如下<error-page>元素：

```
<error-page>
  <error-code>500</error-code>
  <location>/system_error.jsp</location>
 </error-page>

 <error-page>
  <exception-type>javax.servlet.ServletException</exception-type>
  <location>/system_error.jsp</location>
 </error-page>
```

从 exsample 应用的主页 index.jsp 上选择“action1.do”链接，该请求转发给 Action1，

它的 execute()方法调用 Model 类的 businessMethod()方法，但不捕获任何异常。execute()方法的代码如下：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {
        new Model().businessMethod();
        return (mapping.findForward("result"));
}
```

此外，在 Struts 配置文件中也没有为 Action1 抛出的异常配置异常处理类，因此，由 Model 类的 businessMethod()方法抛出的 BaseException 异常，先被 RequestProcessor 类的 processException()方法重新包装为 ServletException 异常，然后抛给 Java Web 容器。processException()方法的这段异常处理代码如下：

```
if (config == null) {
    log.warn(getInternal().getMessage("unhandledException",
                                      exception.getClass()));
    if (exception instanceof IOException) {
        throw (IOException) exception;
    } else if (exception instanceof ServletException) {
        throw (ServletException) exception;
    } else {
        throw new ServletException(exception);
    }
}
```

Java Web 容器捕获到 ServletException 异常后，找到和处理这个异常匹配的<error-page>元素，然后向用户返回<location>子元素指定的 system_error.jsp 网页，如图 11-7 所示。

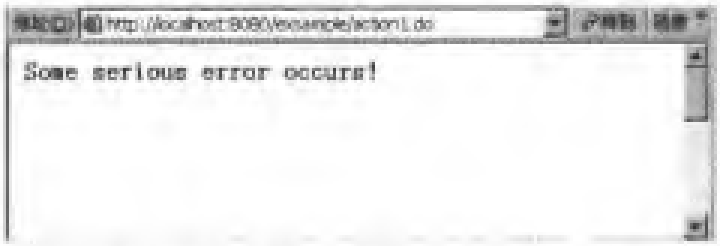

图 11-7 Java Web 容器返回错误网页 system_error.jsp

尽管 Java Servlet 规范要求 Java Web 容器提供对<error-page>元素的支持，但是有些实际的 Java Web 容器并没有实现这一功能。例如，在 Tomcat 的某些版本上运行 exsample 应用时，您会发现，即使在 web.xml 中正确配置了相应的<error-page>元素，Tomcat 服务器并没有返回<error-page>元素指定的错误网页，而是直接向用户显示 HTTP 500 错误和原始的异常信息。

如果把 web.xml 中的<error-page>元素的配置代码删除，在这种情况下，Web 容器就把原始的异常信息直接抛给用户，如图 11-8 所示。

图 11-8　Java Web 容器直接抛出原始异常信息

在开发实际应用时，让 Web 容器来捕获和业务逻辑相关的应用异常是不可取的，这种异常应该由 Struts 的控制器来处理。Web 容器通常负责捕获严重的系统异常。

11.4.3　以配置方式处理异常

Struts 框架允许以配置方式处理异常。配置方式可以避免在 Action 类中通过硬编码来处理异常，从而提高应用的灵活性、可重用性和可维护性。

根据本章 11.3.3 小节的内容，对于 Action 类的 execute()方法抛出的异常，RequestProcessor 类会先查找相应的异常处理元素<exception>。如果<exception>元素被嵌套在<global-exceptions>元素中，就表示全局的异常处理元素，对所有 Action 都适用。如果<exception>元素被嵌套在<action>元素中，就表示局部的异常处理元素，仅对当前的 Action 适用。

例如，以下是为 Action3 配置的局部异常处理元素：

```
<action      path="/action3"
             type="exsample.Action3" >
 <exception
  key="error.business.error"
```

```
        path="/app_error.jsp"
        handler="exsample.BaseExceptionHandler"
        type="exsample.BaseException"/>
    </action>
```

<exception>元素包括以下属性：

- type：指定待处理的异常类。
- handler：指定异常处理类，默认值为 org.apache.struts.action.ExceptionHandler。如果设置为用户自定义的异常处理类，则该类必须继承 ExceptionHandler。
- path：指定转发路径。
- key：指定错误消息 key，Struts 框架将根据这个 key 到 Resource Bundle 中寻找匹配的消息文本。
- bundle：指定 Resource Bundle，如果没有设置此项，将使用默认的 Resource Bundle。
- scope：指定 ActionMessages 的存放范围，可选值包括 request 和 session。默认值为 request。

1．使用 Struts 默认的 ExceptionHandler

从 exsample 应用的主页 index.jsp 上选择“action2.do”链接，该请求转发给 Action2，它的 execute()方法调用 Model 类的 businessMethod()方法，捕获到 BaseException 后，把它包装为 ModuleException，再继续抛出 ModuleException。execute()方法的代码如下：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {
        try{
            new Model().businessMethod();
        }catch(BaseException be){
          throw new ModuleException("error.business.error");
        }
        // Forward control to the specified success URI
        return (mapping.findForward("result"));
    }
```

在 Struts 配置文件中为 Action2 配置了如下<exception>元素：

```
<action      path="/action2"
             type="exsample.Action2" >
 <exception
  key="error.business.error"
  path="/app_error.jsp"
  type="org.apache.struts.util.ModuleException"/>
</action>
```

以上<exception>元素没有指定 handler 属性，因此将使用默认的 org.apache.struts.action.ExceptionHandler。ExceptionHandler 对异常信息重新包装后，把请求

转发给<exception>元素的 path 属性指定的 app_error.jsp 网页。app_error.jsp 的代码如下：

```
<html:html locale="true">
  <head>
    <title>Showing Error</title>
    <html:base/>
  </head>
  <body bgcolor="white"><p>
    <h2>Showing Error</h2><p>
     <html:errors/><p>
     <logic:messagesNotPresent>
         There are no errors.
     </logic:messagesNotPresent><P>
  </body>
</html:html>
```

app_error.jsp 网页的<html:errors>标签从 ActionMessages 对象中读取 ActionMessage 对象，把错误消息输出到网页上，如图 11-9 所示。

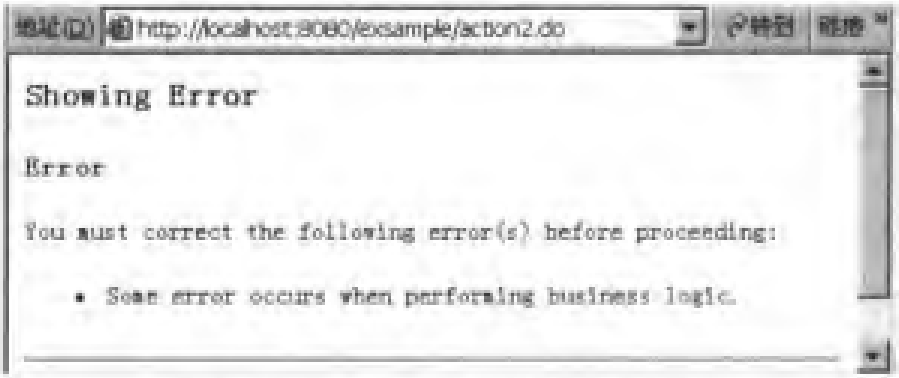

图 11-9　app_error.jsp 网页输出的错误信息

2. 创建自定义的 ExceptionHandler

对于默认的异常处理类 org.apache.struts.action.ExceptionHandler，仅仅处理当前异常类的信息，忽略异常类中嵌套的其他异常信息。如果希望异常处理类能够处理多样化异常，可以定义自己的异常处理类，例程 11-4 提供了一种实现方案。

例程 11-4　能够处理多样化异常的 BaseExceptionHandler.java

```
package exsample;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ExceptionHandler;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
```

```
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.config.ExceptionConfig;
import org.apache.struts.Globals;
import java.util.List;
import java.util.Iterator;


public class BaseExceptionHandler extends ExceptionHandler {

    public ActionForward execute(Exception ex,
                                 ExceptionConfig config,
                                 ActionMapping mapping,
                                 ActionForm formInstance,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws ServletException {

      ActionMessages errors = new ActionMessages();
      ActionForward forward = null;
      ActionMessage error = null;
      String property = null;

      /* Get the path for the forward either from the exception element
       * or from the input attribute.
       */
      String path = null;
      if (config.getPath() != null) {
       path = config.getPath();
      }else{
        path = mapping.getInput();
      }
      // Construct the forward object
      forward = new ActionForward(path);

      /* Figure out what type of exception has been thrown. The Struts
       * AppException is not being used in this example.
       */
      if( ex instanceof BaseException) {
        /*// This is the specialized behavior
        BaseException baseException = (BaseException)ex;
        String messageKey = baseException.getMessageKey();
        Object[] exArgs = baseException.getMessageArgs();
        if ( exArgs != null && exArgs.length > 0 ){
          // If there were args provided, use them in the ActionMessage
          error = new ActionMessage( messageKey, exArgs );
```

```
        }else{
          // Create an ActionMessage without any arguments
          error = new ActionMessage( messageKey );
        }*/

        processBaseException( (BaseException)ex, config,request,forward, property,errors);
       // See if this exception contains a list of subexceptions
        List exceptions = ((BaseException)ex).getExceptions();
        if (exceptions != null && !exceptions.isEmpty() ){
          int size = exceptions.size();
          Iterator iter = exceptions.iterator();

          while( iter.hasNext() ){
           // All subexceptions must be BaseExceptions
                BaseException subException = (BaseException)iter.next();

                processBaseException(subException, config,request,forward,
                property,errors);
          }
        }
      }else{
        error = new ActionMessage(config.getKey());
        property = error.getKey();
        storeException(request, property, error, forward, config.getScope(),errors);
      }

      return forward;
    }

    protected void processBaseException( BaseException ex,
                                         ExceptionConfig config,
                                         HttpServletRequest request,
                                         ActionForward forward,
                                         String property,
                                         ActionMessages errors){

        String messageKey = ex.getMessageKey();
        ActionMessage error=null;
        Object[] exArgs = ex.getMessageArgs();
        if ( exArgs != null && exArgs.length > 0 ){
          // If there were args provided, use them in the ActionMessage
          error = new ActionMessage( messageKey, exArgs );
        }else{
          // Create an ActionMessage without any arguments
          error = new ActionMessage( messageKey );
        }
```

```
            storeException(request, property, error, forward, config.getScope(),errors);
        }

        protected void storeException(HttpServletRequest request,
                                      String property,
                                      ActionMessage error,
                                      ActionForward forward,
                                      String scope,ActionMessages errors) {
            errors.add(property, error);
            if ("request".equals(scope)){
                request.setAttribute(Globals.ERROR_KEY, errors);
            } else {
                request.getSession().setAttribute(Globals.ERROR_KEY, errors);
            }
        }
    }
```

BaseExceptionHandler 扩展了 ExceptionHandler，在它的 execute()方法中，能够把 BaseException 以及逐级嵌套的其他 BaseException 全部重新包装成 ActionMessage 对象，然后把这些 ActionMessage 对象保存在 ActionMessages 对象中。

从 exsample 应用的主页 index.jsp 上选择“action3.do”链接，该请求转发给 Action3，它的 execute()方法调用 Model 类的 businessMethod()方法，不捕获任何异常。execute()方法的代码如下：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {
        new Model().businessMethod();
        return (mapping.findForward("result"));
}
```

在 Struts 配置文件中为 Action3 配置了如下<exception>元素：

```
<action      path="/action3"
             type="exsample.Action3" >
  <exception
    key="error.business.error"
    path="/app_error.jsp"
    handler="exsample.BaseExceptionHandler"
    type="exsample.BaseException"/>
</action>
```

根据以上的<exception>元素配置，在 Action3 的 execute()方法抛出 BaseException 异常后，由 BaseExceptionHandler 类来处理异常，它把 BaseException 以及嵌套的异常都重新包装成 ActionMessage 对象，再把它们存放在 request 范围内的 ActionMessages 对象中，如图

11-10 所示。

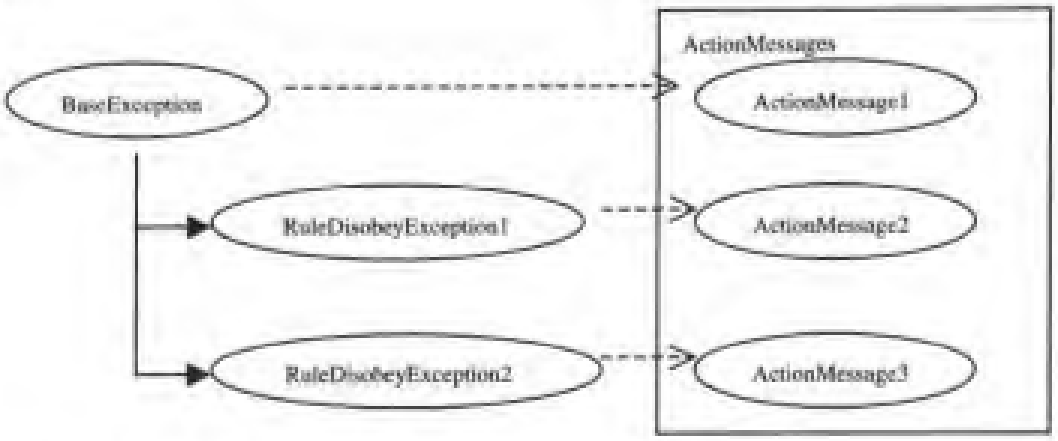

图 11-10　BaseExceptionHandler 把 BaseException 包装成 ActionMessage 对象

BaseExceptionHandler 类最后把请求转发给 app_error.jsp 网页。app_error.jsp 网页的<html:errors>标签从 ActionMessages 对象中读取所有的 ActionMessage 对象，把错误消息输出到网页上，如图 11-11 所示。

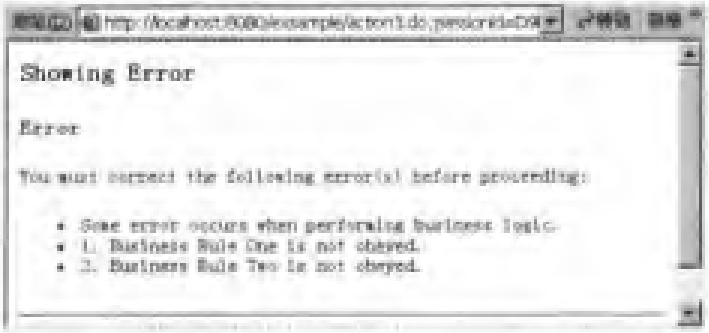

图 11-11　app_error.jsp 网页输出的错误信息

11.4.4　以编程方式处理异常

11.4.3 小节介绍了如何以配置方式处理异常，此外，也可以在 Action 类中以编程方式来处理异常。有时可以把这两种方式结合使用，在 Action 类中先捕获并处理相关的异常，没有被 Action 捕获的异常再由 RequestProcessor 类来捕获，然后由 RequestProcessor 根据异常配置信息调用相应的 ExceptionHandler 处理该异常。

1．在 Action 类中捕获异常

从 exsample 应用的主页 index.jsp 上选择“action4.do”链接，该请求转发给 Action4，它的 execute()方法调用 Model 类的 businessMethod()方法，捕获到 BaseException 后，把 BaseException 重新包装成 ActionMessage 对象，然后把 ActionMessage 对象保存在 ActionMessages 对象中。接着把请求转发给错误处理页面 app_error.jsp。以下是 execute()方法的代码：

```
public ActionForward execute(ActionMapping mapping,
```

```
                                        ActionForm form,
                                        HttpServletRequest request,
                                        HttpServletResponse response)
        throws Exception {

            try{
                new Model().businessMethod();
            }catch(BaseException ex){
              // Log the exception

              // Create and store the action error
              ActionMessages errors = new ActionMessages();
              ActionMessage newMessage = new ActionMessage( ex.getMessageKey(),
              ex.getMessageArgs() );
              errors.add( ActionMessages.GLOBAL_MESSAGE, newMessage );
              saveErrors( request, errors );

              // Return an ActionForward for the Failure resource
              return mapping.findForward( Constants.APP_ERROR_PAGE );

            }catch( Throwable ex ){
              // Log the exception

              // Create and store the action error
              ActionMessage newError = new ActionMessage( "error.system.error" );
              ActionMessages errors = new ActionMessages();
              errors.add( ActionMessages.GLOBAL_MESSAGE, newError );
              saveErrors( request, errors );

              // Return an ActionForward for the system error resource
              return mapping.findForward( Constants.APP_ERROR_PAGE );
            }
            // Forward control to the specified success URI
             return (mapping.findForward("result"));
        }
```

在以上 catch 代码块中，最后把请求转发给 app_error.jsp 网页。如图 11-12 所示为 app_error.jsp 输出的错误消息。

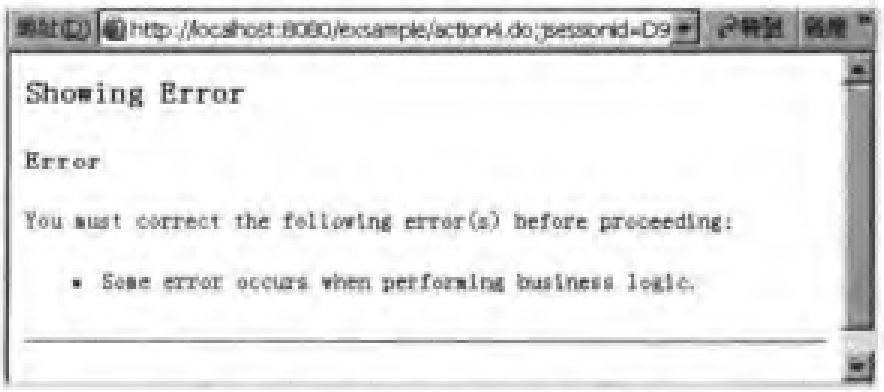

图 11-12 app_error.jsp 网页输出错误信息

2. 在 Action 类中处理多样化异常

以上 11.4.4 小节的第 1 点的 Action4 处理异常的行为与默认的 ExceptionHandler 很相似，两者都只处理当前的异常，忽略嵌套的其他异常。另外，如果在应用的每个 Action 类中都按照 11.4.4 小节第 1 点的方式来处理异常，会导致大量的重复代码。因此，可以定义一个负责处理异常的 Action 父类，其他的 Action 只需继承这个 Action 类即可，这样可以避免重复处理异常。例程 11-5 为 BaseAction 类的代码。

例程 11-5　负责处理异常的 BaseAction.java

```
public abstract class BaseAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
  throws Exception {
    ActionForward forwardPage = null;
    try{
      // Inform the specific action instance to do its thing
      forwardPage = executeAction(mapping, form, request, response);
    }catch (BaseException ex){
      // Log the application exception using your logging framework
      // Call the generic exception handler routine
      forwardPage = processExceptions( request, mapping, ex );
    }catch (Throwable ex){
      // Log the system exception using your logging framework
      // Treat all other exceptions as system errors
      forwardPage = mapping.findForward( Constants.SYSTEM_ERROR_PAGE );
    }
    return forwardPage;
  }

abstract public ActionForward executeAction( ActionMapping mapping,
                                             ActionForm form,
                                             HttpServletRequest request,
                                             HttpServletResponse response
                                             )throws BaseException;


protected ActionForward processExceptions( HttpServletRequest request,
                                           ActionMapping mapping,
                                           BaseException ex ){
 ActionMessages errors = new ActionMessages();
 ActionForward forward = null;

 Locale locale = getLocale(request);
```

```
        processBaseException(errors, ex, locale);

        // Either return to the input resource or a configured failure forward
        String inputStr = mapping.getInput();
        ActionForward failureForward = mapping.findForward(Constants.APP_ERROR_PAGE);

        if ( inputStr != null) {
           forward = new ActionForward( inputStr );
        }else if (failureForward != null){
           forward = failureForward;
        }

        // See if this exception contains a list of subexceptions
        List exceptions = ex.getExceptions();
        if (exceptions != null && !exceptions.isEmpty() ){
           int size = exceptions.size();
           Iterator iter = exceptions.iterator();

           while( iter.hasNext() ){
            // All subexceptions must be BaseExceptions
              BaseException subException = (BaseException)iter.next();

                  processBaseException(errors, subException, locale);
           }
        }

        // Tell the Struts framework to save the errors into the request
        saveErrors( request, errors );

        // Return the ActionForward
        return forward;
    }
    protected void processBaseException( ActionMessages errors,
                                                    BaseException ex,
                                                    Locale locale) {

       // Holds the reference to the ActionMessage to be added
       ActionMessage newActionMessage = null;

       // The errorCode is the key to the resource bundle
       String errorCode = ex.getMessageKey();
       /**
        * If there are extra arguments to be used by the MessageFormat object,
        * insert them into the argList. The arguments are context sensitive
        * arguments for the exception; there may be 0 or more.
```

```
        */
        Object[] args = ex.getMessageArgs();

        /**
         * In an application that had to support I18N, you might want to
         * format each value in the argument array based on its type and the
         * user locale. For example, if there is a Date object in the array, it
         * would need to be formatted for each locale.
         */

        // Now construct an instance of the ActionMessage class
        if ( args != null && args.length > 0 ){
          // Use the arguments that were provided in the exception
          newActionMessage = new ActionMessage( errorCode, args );
        }else{
          newActionMessage = new ActionMessage( errorCode );
        }
        errors.add( ActionMessages.GLOBAL_MESSAGE, newActionMessage );
    }
```

BaseAction 类处理异常的方式与 BaseExceptionHandler 很相似，它能够把 BaseException 中逐级嵌套的所有异常都包装为 ActionMessage 对象。BaseAction 类中定义了一个抽象的 executeAction()方法，继承 BaseAction 类的子类不必覆盖父类的 execute()方法，而是应该实现抽象的 executeAction()方法。

exsample 应用的 Action5 就是继承 BaseAction 的例子，参见例程 11-6。

例程 11-6　Action5.java

```
public final class Action5 extends BaseAction {

    public ActionForward executeAction( ActionMapping mapping,
                                        ActionForm form,
                                        HttpServletRequest request,
                                        HttpServletResponse response)
    throws BaseException{
        new Model().businessMethod();
        return (mapping.findForward("result"));
    }
}
```

从 exsample 应用的主页 index.jsp 上选择“action5.do”链接，该请求转发给 Action5，它的 execute()方法调用 Model 类的 businessMethod()方法，捕获到 BaseException 后，能够把 BaseException 中逐级嵌套的所有异常都包装为 ActionMessage 对象。错误处理页面 app_error.jsp 输出的错误信息如图 11-13 所示。

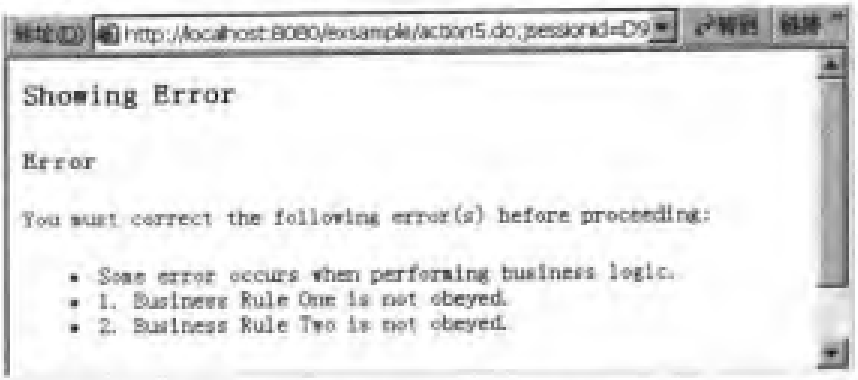

图 11-13 app_error.jsp 网页输出的错误信息

11.5 小 结

Struts 框架提供了强大的异常处理机制。Struts 的控制器负责捕获异常，并把异常包装为与 Resource Bundle 绑定的 ActionMessage 对象。在视图层中，<html:errors>标签能够读取 ActionMessages 集合中的所有 ActionMessage 对象，向用户显示本地化的错误消息。

Struts 框架支持两种异常处理方式：编程方式和配置方式。编程方式是指在 Action 类的 execute()方法中直接捕获并处理异常。配置方式是指在 Struts 配置文件中配置<exception>元素，该元素指定处理特定异常的异常处理类，以及请求转发路径。配置方式可以避免在 Action 类中通过硬编码来处理异常，从而提高应用的灵活性、可重用性和可维护性。

第 12 章　Struts HTML 标签库

Struts HTML 标签可以和标准的 HTML 元素完成相同的功能。在 Struts 应用中提倡使用 Struts HTML 标签，这是因为这些标签可以和 Struts 框架的其他组件紧密地联系在一起。例如，<html:form>标签用来定义 HTML 表单，Struts 框架能够把这个表单中的数据自动映射到相应的 ActionForm Bean 中。Struts HTML 标签大致分为以下几类：

- 用于生成基本的 HTML 元素的标签
- 用于生成 HTML 表单的标签
- 显示错误或正常消息的标签

本章通过一个 Struts 应用例子：htmltaglibs 应用，来讲解如何使用 Struts HTML 标签。htmltaglibs 应用的主页 index.jsp 如图 12-1 所示，它提供了访问其他样例网页的链接，这些样例网页用于演示各种 HTML 标签的使用方法。

图 12-1　htmltaglibs 应用的主页 index.jsp

本章介绍的 htmltaglibs 应用的源代码位于配套光盘的 sourcecode/htmltaglibs 目录下，如果要在 Tomcat 上发布这个应用，只要把整个 htmltaglibs 目录拷贝到<CATALINA_HOME>/webapps 目录下即可。

12.1　用于生成基本的 HTML 元素的标签

Struts HTML 标签库中的许多标签都和基本的 HTML 元素对应，这些标签包括：

- <html:html>：生成 HTML <html> 元素。
- <html:base>：生成 HTML <base> 元素。

- <html:link>：生成 HTML Anchor <a>元素。
- <html:rewrite>：生成用户请求的 URI。
- <html:img>：生成 HTML <img> 元素。

从 htmltaglibs 应用的主页 index.jsp 上选择“HtmlBasic”链接，就可以访问本节的样例网页 HtmlBasic.jsp。如图 12-2 所示。

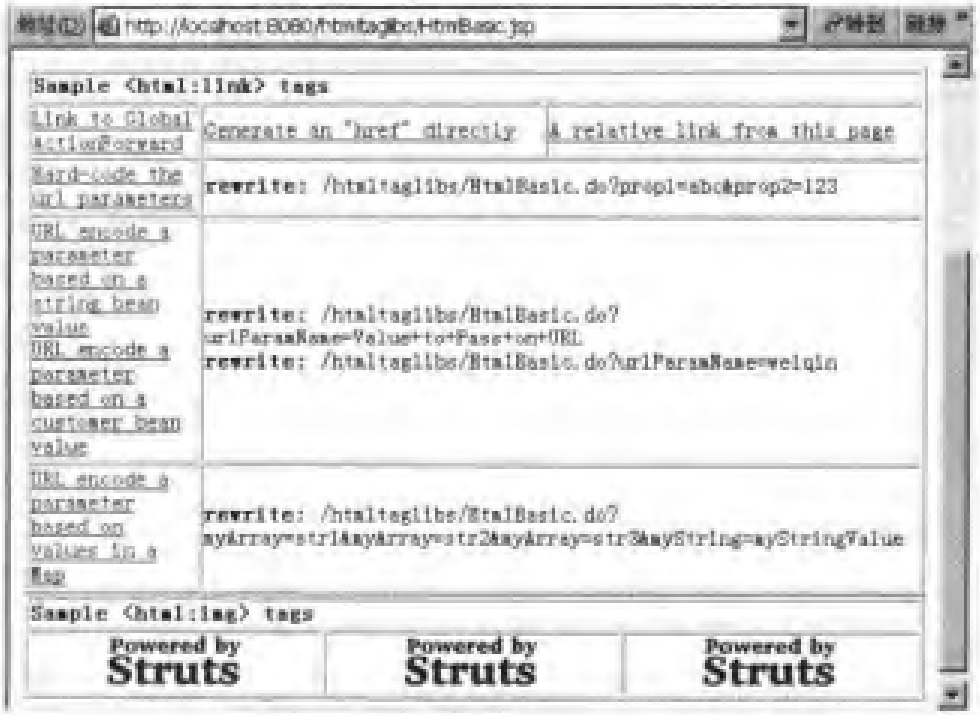

图 12-2 HtmlBasic.jsp 网页

从图 12-2 中可以看出，HtmlBasic.jsp 网页上没有表单，也没有提交按钮。网页上包含了一些 URL 链接。例程 12-1 为 HtmlBasic.jsp 的代码。

例程 12-1 HtmlBasic.jsp

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ page import="htmltaglibs.beans.CustomerBean"%>
<html:html lang=true>
<head>
<title>Base HTML Tags</title>
<html:base/>
</head>
<body bgcolor="white">

<h3>Sample code for basic Struts html tags</h3>

<p>This page provides examples of the following Struts HTML tags:<br>
<ul>
<li>&lt;html:html&gt;</li>
```

```
<li>&lt;html:base&gt;</li>
<li>&lt;html:link&gt;</li>
<li>&lt;html:rewrite&gt;</li>
<li>&lt;html:img&gt;</li>
</ul>
<table border="1" width="100%">
  <tr>
    <th colspan="3" align="left">
      Sample &lt;html:link&gt; tags
    </th>
  </tr>

  <%--
  The following section contains three <html:link> tags. Each
  demonstrates a different way of creating an anchor tag (<a href=...>).
  --%>
  <tr>
    <td align="left">
      <%-- Create link from a Global Forward in the struts-config.xml --%>
      <html:link forward="index">
        Link to Global ActionForward
      </html:link>
    </td>
    <td align="left">
      <%-- Create link by specifying a full URL --%>
      <html:link href="http://jakarta.apache.org/struts/index.html">
        Generate an "href" directly
      </html:link>
    </td>
    <td align="left">
      <%-- Create the link as a relative link from this page --%>
      <html:link page="/HtmlBasic.do">
        A relative link from this page
      </html:link>
    </td>
  </tr>

  <%--
  The <html:link> and <html:rewrite> tags a very similar. The only difference
  is that <html:rewrite> creates the URI without prepending the
  "http://hostname:port/" part.
  --%>
  <tr>
    <%-- Create link and hard-code request parameters --%>
    <td colspan="1" align="left">
      <html:link page="/HtmlBasic.do?prop1=abc&amp;prop2=123">
```

```
            Hard-code the url parameters
          </html:link>
        </td>
        <%-- Create the same rewrite string for the above link. --%>
        <td colspan="2" align="left">
          <b>rewrite: </b>
          <html:rewrite page="/HtmlBasic.do?prop1=abc&amp;prop2=123" />
        </td>
      </tr>

<%
  /*
   * Create a String object to store as a bean in
   * the page context and embed in this link
   */
  String stringBean = "Value to Pass on URL";
  pageContext.setAttribute("stringBean", stringBean);
%>
<jsp:useBean id="customerBean" scope="page" class="htmltaglibs.beans.CustomerBean"
/>
<jsp:setProperty name="customerBean" property="name" value="weiqin" />

      <tr>
        <%-- Create link with request parameters from a bean --%>
        <td colspan="1" align="left">
          <%-- For this version of the <html:link> tag:                    --%>
          <%--     paramID = the name of the url parameter                 --%>
          <%--     paraName = the "attribute" for the bean holding the value --%>
          <html:link page="/HtmlBasic.do"
                     paramId="urlParamName" paramName="stringBean">
          URL encode a parameter based on a string bean value
          </html:link>
          <br>
          <html:link page="/HtmlBasic.do"
                     paramId="urlParamName" paramName="customerBean"
              paramProperty="name">
          URL encode a parameter based on a customer bean value
          </html:link>
        </td>
        <%-- Create the same rewrite string for the above link. --%>
        <td colspan="2" align="left">
          <b>rewrite: </b>
          <html:rewrite page="/HtmlBasic.do"
                     paramId="urlParamName" paramName="stringBean" />
          <br>
          <b>rewrite: </b>
```

```
            <html:rewrite page="/HtmlBasic.do"
                    paramId="urlParamName" paramName="customerBean"
                paramProperty="name"/>
          </td>
        </tr>
      <%
        /*
         * Store values in a Map (HashMap in this case)
         * and construct the URL based on the Map
         */
        java.util.HashMap myMap = new java.util.HashMap();
        myMap.put("myString", new String("myStringValue") );
        myMap.put("myArray", new String[] { "str1", "str2", "str3" });
        pageContext.setAttribute("map", myMap);
      %>
       <tr>
          <%-- Create a link with request parameters from a Map --%>
          <td colspan="1" align="left">
            <%-- For this version of the <html:link> tag: --%>
            <%--     map = a map with name/value pairs to pass on the url --%>
            <html:link page="/HtmlBasic.do" name="map">
            URL encode a parameter based on values in a Map
            </html:link>
          </td>
          <%-- Create the same rewrite string for the above link. --%>
          <td colspan="2" align="left">
            <b>rewrite: </b>
            <html:rewrite page="/HtmlBasic.do" name="map"/>
          </td>
       </tr>

    </table>

    <table border="1" width="100%">
      <tr>
        <th colspan="3" align="left">
          Sample &lt;html:img&gt; tags
        </th>
      </tr>

      <tr>
        <%-- Create a default <img> tag --%>
        <td align="center">
          <html:img page="/struts-power.gif" />
        </td>
```

```
    <%-- Create an <img> tag with request parameters from a bean --%>
    <td align="center">
      <%-- Note "src" requires using full relative path --%>
      <html:img src="/htmltaglibs/struts-power.gif"
                paramId="urlParamName" paramName="stringBean" />
    </td>

    <%-- Create an <img> tag with request parameters from a map --%>
    <td align="center">
      <html:img page="/struts-power.gif" name="map" />
    </td>
  </tr>
</table>
</html:html>
```

12.1.1 <html:html>标签

<html:html>标签用于在网页的开头生成 HTML 的<html>元素。<html:html>标签有一个 lang 属性，用于显示用户使用的语言：

```
<html:html lang="true">
```

如果客户浏览器使用中文语言，那么以上代码在运行时将被解析为普通的HTML代码：

```
<html lang="zh-CN">
```

当 lang 属性为“true”时，<html:html>标签将先根据存储在当前 HttpSession 中的 Locale 对象来输出网页使用的语言，如果不存在 HttpSession，或者 HttpSession 中没有 Locale 对象，就根据客户浏览器提交的 HTTP 请求头中的 Accept-Language 属性来输出语言，如果 HTTP 请求头中没有 Accept-Language 属性，就根据默认的 Locale 来输出语言。

在 Struts 的早期版本中，<html:html>标签采用 locale 属性来完成和 lang 属性相同的功能。区别在于如果 locale 属性为“true”，并且不存在 HttpSession，Struts 框架会创建 HttpSession，然后把从 HTTP 请求中获得的 Locale 保存在 HttpSession 中。这种做法强制性地要求应用支持 Session，不是很可取，所以 locale 属性在 Struts 1.2 版本中被 lang 属性所替代。

12.1.2 <html:base>标签

<html:base>标签在网页的<head>部分生成 HTML <base> 元素。HTML <base>元素用于生成当前网页的绝对 URL 路径。在本例中<html:base/>将输出如下内容：

```
<base href="http://localhost:8080/htmltaglibs/HtmlBasic.jsp">
```

如果在网页中使用了<html:base>标签，当该网页引用同一个应用的其他网页时，只需给出相对于当前网页的相对 URL 路径即可。

12.1.3　<html:link>和<html:rewrite>标签

<html:link>标签用于生成 HTML <a>元素。<html:link>在创建超链接时，有两个优点：

- 允许在 URL 中以多种方式包含请求参数。
- 当用户浏览器关闭 Cookie 时，会自动重写 URL，把 SessionID 作为请求参数包含在 URL 中，用于跟踪用户的 Session 状态。

<html:link>标签有以下重要属性：

- forward：指定全局转发链接。
- href：指定完整的 URL 链接。
- page：指定相对于当前网页的 URL。

<html:rewrite>用于输出超链接中的 URI 部分，但它并不生成 HTML <a> 元素。URI 指的是 URL 中协议、主机和端口以后的内容。URI 用于指定具体的请求资源。例如，对于 URL：http://localhost:8080/htmltaglibs/HtmlBasic.do，它的 URI 为/ htmltaglibs/HtmlBasic.do。

以下的一些例子用于演示如何使用<html:link>和<html:rewrite>标签。

1．创建全局转发链接

首先，在 Struts 配置文件的<global-forwards>元素中定义一个<forward>元素：

```
<global-forwards>
    <forward  name="index"  path="/index.jsp"/>
</global-forwards>
```

接着，在 JSP 文件中创建<html:link> 标签：

```
<html:link forward="index">
  Link to Global ActionForward
</html:link>
```

<html:link>标签的 forward 属性和<global-forwards>元素中的<forward>子元素匹配。以上代码生成如下 HTML 内容：

```
<a href="/htmltaglibs/index.jsp">Link to Global ActionForward</a>
```

值得注意的是，<html:link>的 forward 属性只能引用 Struts 配置文件中<global-forwards>内的<forward>子元素，如果引用<action>内的<forward>子元素，在运行时将会抛出以下异常：

```
Cannot create rewrite URL: Java.net.MalformedURLException: Cannot retrieve ActionForward
```

2．创建具有完整 URL 的链接

如果 Web 应用需要链接到其他站点，应该给出其他站点的完整 URL，例如：

```
<html:link href="http://jakarta.apache.org/struts/index.html">
  Generate an "href" directly
</html:link>
```

<html:link>标签的 href 属性用于指定一个完整的 URL 路径，以上代码生成如下 HTML 内容：

```
<a href="http://jakarta.apache.org/struts/index.html"> Generate an "href" directly</a>
```

值得注意的是，如果指定了<html:link>标签的 href 属性，即使用户浏览器的 Cookie 关闭，<html:link>标签也不会把用户 SessionID 作为请求参数加入到 URL 中。

3．从当前网页中创建相对 URL

如果从一个网页链接到同一个应用中的另一网页，可以采用以下方式：

```
<html:link page="/HtmlBasic.do">
  A relative link from this page
</html:link>
```

<html:link>标签的 page 属性用于指定相对于当前应用的 URI。以上代码生成如下 HTML 内容：

```
<a href="/htmltaglibs/HtmlBasic.do">A relative link from this page</a>
```

4．在 URL 或 URI 中包含请求参数

如果要在 URL 或 URI 中包含请求参数，只要把请求参数加在 URL 或 URI 的末尾就可以了。例如：

```
<html:link page="/HtmlBasic.do?prop1=abc&amp;prop2=123">
  Hard-code the url parameters
</html:link>
<%-- or --%>
rewrite: <html:rewrite page="/HtmlBasic.do?prop1=abc&amp;prop2=123" />
```

以上代码生成如下 HTML 内容：

```
<a href="/htmltaglibs/HtmlBasic.do?prop1=abc&amp;prop2=123"> Hard-code the url
parameters</a>
rewrite: /htmltaglibs/HtmlBasic.do?prop1=abc&amp;prop2=123
```

在 HTML 编码中，“&”代表特殊字符“&”。

5．在 URL 或 URI 中包含单个请求变量

如果要在 URL 中包含一个请求参数，而这个参数的值存在于当前网页可访问的一个变量中，可以按以下方法来实现。

为了演示这一功能，首先创建一个当前网页可访问的变量。例如，本例中创建了两个变量，一个是字符串类型变量，一个是 CustomerBean，它们都存放于 page 范围内：

```
<%
  /*
```

```
    * Create a String object to store as a bean in
    * the page context and embed in this link
    */
   String stringBean = "Value to Pass on URL";
   pageContext.setAttribute("stringBean", stringBean);
%>
<jsp:useBean id="customerBean" scope="page" class="htmltaglibs.beans.CustomerBean"
/>
<jsp:setProperty name="customerBean" property="name" value="weiqin" />
```

接着，把这两个变量作为请求参数，加入到 URL 或 URI 中：

```
<html:link page="/HtmlBasic.do"
                    paramId="urlParamName" paramName="stringBean">
            URL encode a parameter based on a string bean value
</html:link>
<html:link page="/HtmlBasic.do"
                    paramId="urlParamName" paramName="customerBean"
                    paramProperty="name">
            URL encode a parameter based on a customer bean value
</html:link>

rewrite: <html:rewrite page="/HtmlBasic.do"
                    paramId="urlParamName" paramName="stringBean" />
rewrite: <html:rewrite page="/HtmlBasic.do"
                    paramId="urlParamName" paramName="customerBean"
                    paramProperty="name"/>
```

<html:link>标签的 paramId 属性指定请求参数名，paramName 属性指定变量的名字。如果变量为 JavaBean，用 paramProperty 属性指定 JavaBean 的属性。对于本例的 stringBean，请求参数值为 stringBean 的字符串值。对于 customerBean，指定了 paramProperty 属性，请求参数值为 customerBean 的 name 属性值。以上代码生成如下 HTML 内容：

```
<a href="/htmltaglibs/HtmlBasic.do?urlParamName=Value+to+Pass+on+URL">URL encode a
 parameter based on a string bean value</a>
<a href="/htmltaglibs/HtmlBasic.do?urlParamName=weiqin">URL encode a parameter based on a
 customer bean value</a>

rewrite: /htmltaglibs/HtmlBasic.do?urlParamName=Value+to+Pass+on+URL
rewrite: /htmltaglibs/HtmlBasic.do?urlParamName=weiqin
```

6. 在 URL 或 URI 中包含多个请求变量

如果在 URL 或 URI 中包含多个请求参数，而这些参数的值来自于多个变量，需要先定义一个 Map 类型的 Java 类，如 java.util.HashMap，用它来存放请求变量。例如：

```
<%
   /*
    * Store values in a Map (HashMap in this case)
    * and construct the URL based on the Map
```

```
    */
    java.util.HashMap myMap = new java.util.HashMap();
    myMap.put("myString", new String("myStringValue") );
    myMap.put("myArray", new String[] { "str1", "str2", "str3" });
    pageContext.setAttribute("map", myMap);
%>
```

在以上代码的 HashMap 中存放了两个对象，其中第二个对象是个字符串数组。HashMap 被存放在 PageContext 中。接下来就可以把这个 HashMap 作为请求参数，加入到 URL 或 URI 中：

```
<%-- For this version of the <html:link> tag: --%>
<%-- map = a map with name/value pairs to pass on the url --%>
<html:link page="/HtmlBasic.do" name="map">
  URL encode a parameter based on values in a Map
</html:link>
<%-- Create the same rewrite string for the above link. --%>
rewrite: <html:rewrite page="/HtmlBasic.do" name="map"/>
```

<html:link>标签的 name 属性指定包含请求变量的 HashMap 对象。HashMap 对象中的每一对“key/value”代表一对或多对“请求参数名/请求参数值”。以上代码生成如下的 HTML 内容：

```
<a href="/htmltaglibs/HtmlBasic.do?myString=myStringValue&amp;myArray=str1&amp;
myArray=str2&amp;myArray=str3">URL encode a parameter based on values in a Map</a>

rewrite: /htmltaglibs/HtmlBasic.do?myString=myStringValue&amp;myArray=str1&amp;
myArray=str2&amp;myArray=str3
```

12.1.4 <html:img>标签

<html:img>标签用于在 HTML 页中嵌入图片。此外，它还允许包含请求变量，以便动态控制图片的输出。

1．生成基本的 HTML <img>元素

<html:img >标签的基本使用方法如下：

```
<html:img page="/struts-power.gif" />
```

<html:img>标签的 page 属性指定相对于当前页面的 URI，以上代码生成如下的 HTML 内容：

```
<img src="/htmltaglibs/struts-power.gif">
```

2．生成包含单个请求变量的 HTML <img>元素

在<html:img>标签中包含单个请求变量的用法和<html:link>标签很相似。例如：

```
<%-- Note "src" requires using full relative path --%>
<html:img src="/htmltaglibs/struts-power.gif"
```

paramId="urlParamName" paramName="stringBean" />

<html:img>标签的 src 属性指定完整的 URI。paramId 指定参数名，paramName 属性指定提供参数值的请求变量。以上代码生成如下的 HTML 内容：

```
<img src="/htmltaglibs/struts-power.gif?urlParamName=Value+to+Pass+on+URL">
```

3．生成包含多个请求变量的 HTML <img>元素

在<html:img>标签中包含多个请求变量的用法和<html:link>标签很相似。例如：

```
<html:img page="/struts-power.gif" name="map" />
```

以上代码生成如下 HTML 内容：

```
<img src="/htmltaglibs/struts-power.
gif?myString=myStringValue&myArray=str1&myArray=str2&myArray=str3">
```

12.2　基本的表单标签

Struts HTML 标签库提供了一组生成 HTML 表单的标签：

- <html:form>：生成 HTML <form> 元素。
- <html:text>：生成 HTML <INPUT type=text> 元素。
- <html:hidden>：生成 HTML <INPUT type=hidden >元素。
- <html:submit>：生成 HTML <INPUT type=submit >元素。
- <html:cancel>：在表单上生成取消按钮。
- <html:reset>：生成 HTML <INPUT type=reset >元素。

从 htmltaglibs 应用的主页 index.jsp 上选择“FormBasic”链接，就可以访问本节的样例网页 FormBasic.jsp，如图 12-3 所示。

图 12-3　FormBasic.jsp 网页

从图 12-3 中可以看出，FormBasic.jsp 网页上包含一个 HTML 表单，表单中有两个文本框、两个隐藏字段、一个提交按钮、一个取消按钮和一个复位按钮。如果在第一个文本框中输入“propValue”，再提交表单，第二个隐藏字段就会回显“propValue”。如果单击【Cancel】（取消）按钮，在表单下方的文本框中就会显示“Cancel was pressed!”。

例程 12-2 为 FormBasic.jsp 的源代码。

例程 12-2　FormBasic.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:html>
<head>
<title>Form Basics sample code</title>
</head>
<body bgcolor="white">

<h3>Form Basics sample code</h3>

<p>This page provides examples of the following Struts HTML tags:<br>
<ul>
<li>&lt;html:form&gt;</li>
<li>&lt;html:text&gt;</li>
<li>&lt;html:hidden&gt;</li>
<li>&lt;html:submit&gt;</li>
<li>&lt;html:cancel&gt;</li>
<li>&lt;html:reset&gt;</li>
</ul>

<html:form action="FormBasic.do">

<table border="1" width="100%">

  <tr>
    <th colspan="3" align="left">
      Sample &lt;html:hidden&gt; tags
    </th>
  </tr>

  <tr>
    <td align="left" >
      Enter value to become hidden: <html:text property="hiddenValue"/>
    </td>
    <td align="left" >
      &lt;html:hidden&gt; tags are rendered below
    </td>
  </tr>
```

```
    <tr>
      <td align="left" >
          &lt;html:hidden property="hiddenValue" /&gt;
      </td>
      <td align="left" >
          <html:hidden property="hiddenValue" />
      </td>
    </tr>

    <tr>
      <td align="left" >
          &lt;html:hidden property="hiddenValue" write="true" /&gt;
      </td>
      <td align="left" >
          <html:hidden property="hiddenValue" write="true" />
      </td>
    </tr>
  </table>
  <br>

  <table border="1" width="100%">

    <tr>
      <th colspan="3" align="left">
        Page Cancel status
      </th>
    </tr>

    <tr>
      <td align="right">
        Which was pressed?:
      </td>
      <td align="left">
        <html:text property="status" disabled="true" />
      </td>
    </tr>
  </table>

  <table border="0" width="100%">

    <tr>
      <td align="right">
        Press Submit, Cancel or Reset:
      </td>
      <td align="left">
```

```
            <html:submit>Submit</html:submit>
            <html:cancel>Cancel</html:cancel>
            <html:reset>Reset</html:reset>
          </td>
        </tr>

      </table>

      </html:form>

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

12.2.1 <html:form>标签

处理 HTML 表单是 Web 应用的主要工作之一。在 Struts 应用中，使用<html:form>标签来创建表单。例如：

```
<html:form action="FormBasic.do">
```

以上代码生成如下的 HTML 内容：

```
<form name="FormBasicForm" method="POST"
            action="/htmltaglibs/FormBasic.do">
```

<html:form>标签的 action 属性用来指定当用户提交表单后，处理用户请求的组件。Struts 框架将参照 Struts 配置文件来查找相应的 Action 组件。在 struts-config.xml 文件中，与“FormBasic.do”对应的代码为：

```
<action     path="/FormBasic"
            type="htmltaglibs.actions.FormBasicAction"
            name="FormBasicForm"
            scope="session"
            input="/FormBasic.jsp"
            validate="false">
            <forward name="success" path="/FormBasic.jsp"/>
</action>
```

根据以上配置代码，当用户提交表单后，由 FormBasicAction 来处理表单。此外，与 HTML 表单关联的 ActionForm 为 FormBasicForm Bean。

12.2.2 <html:text>标签

该标签在表单上创建 HTML 文本框字段。在本例中，定义了一个属性名为“hiddenValue”的文本框：

```
<html:text property="hiddenValue"/>
```

<html:text>标签的 property 属性指定字段的名字，它和 ActionForm Bean 中的一个属性匹配。以上代码定义了 hiddenValue 字段，因此在 FormBasicForm Bean 中应该定义相应的 hiddenValue 属性：

```
private String hiddenValue;
public String getHiddenValue() { return this.hiddenValue; }
public void setHiddenValue(String hiddenValue) { this.hiddenValue = hiddenValue; }
```

这样，当用户提交了表单时，Struts 框架会把 hiddenValue 字段的内容赋值给 FormBasicForm Bean 的 hiddenValue 属性。

12.2.3　<html:cancel>标签

<html:cancel>标签在表单中生成取消按钮。当用户按下取消按钮时，将产生一个取消事件，这个事件由 Action 类来捕获，至于如何处理这个事件，可以在 Action 类的 execute() 方法中编程来实现。

用户选择取消按钮的情形为：用户已经按下了提交按钮，在等待服务器响应结果的过程中忽然改变主意，想取消操作，就会按下取消按钮。因此，在 Action 类中处理取消事件时，可以执行一些清除操作，取消已经执行的响应用户请求的部分操作。

以下代码在网页的表单中创建取消按钮：

```
<html:cancel>Cancel</html:cancel>
```

以上代码生成如下 HTML 内容：

```
<input type="submit" name="org.apache.struts.taglib.html.CANCEL"
                     value="Cancel">
```

在 Action 类中，应该以编程方式来处理取消事件。例程 12-3 为 FormBasicAction 类的源程序。

例程 12-3　FormBasicAction.java

```
package htmltaglibs.actions;

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;
import htmltaglibs.forms.FormBasicForm;

public class FormBasicAction extends Action {
```

```
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        FormBasicForm fbf = (FormBasicForm) form;

        if (isCancelled(request)) {
            /*
             * If request was cancelled, we would clean up any processing
             * that was unfinished and release any resources we may
             * have locked.
             */
            // Set status to reflect that cancel WAS pressed!
            fbf.setStatus("Cancel was pressed!");

            return (mapping.findForward("success"));
        } else {
            // Set status to reflect that cancel WAS NOT pressed!
            fbf.setStatus("Submit was pressed!");

            return (mapping.findForward("success"));
        }
    }
}
```

Action 类的 isCancelled(request)方法用来判断取消事件有没有发生。如果这个方法返回 true，就表示取消事件发生了，可以在程序中进行相关的操作。

12.2.4 <html:reset>标签

<html:reset>标签生成表单的复位按钮，使用方法如下：

```
<html:reset>Reset</html:reset>
```

以上代码生成如下的 HTML 内容：

```
<input type="reset" name="reset" value="Reset">
```

这个标签的用法很简单，它和 HTML 元素<input type="reset">在功能上相同。

12.2.5 <html:submit>标签

<html:submit>标签生成表单的提交按钮，使用方法如下：

```
<html:submit>Submit</html:submit>
```

以上代码生成如下的 HTML 内容：

```
<input type="submit" name="submit" value="Submit">
```

这个标签的用法很简单，它和 HTML 元素<input type="submit">在功能上相同。

12.2.6 <html:hidden>标签

这个标签在表单上生成隐藏字段。隐藏字段用于在表单上存放不希望让用户看到或不允许修改的信息。例如，隐藏字段可以取代 Cookie，来存放用户状态信息。本例提供了使用该标签的两种方式，这两种方式都会生成 HTML <input type="hidden" >元素，其中第二种方式能把隐藏字段的值显示在网页上。

第一种方式为：

```
<html:hidden property="hiddenValue" />
```

以上代码生成如下的 HTML 内容：

```
<input type="hidden" name="hiddenValue" value="propValue">
```

第二种方式增加了 write="true" 属性：

```
<html:hidden property="hiddenValue" write="true" />
```

以上代码生成如下的 HTML 内容：

```
<input type="hidden" name="hiddenValue" value="propValue">propValue
```

当初次访问 FormBasic.jsp 时，以上两个隐藏字段的 value 值都为空，只有当用户在属性为“hiddenValue”的文本框中输入“propValue”并提交表单时，这两个隐藏字段的 value 值才会变为“propValue”。

以上定义的两个隐藏字段，以及在 12.2.2 小节中介绍的文本框字段，它们的 property 属性都为“hiddenValue”，因此它们都和 FormBasicForm Bean 的 hiddenValue 属性对应。第二个隐藏字段还把 FormBasicForm Bean 的 hiddenValue 属性的值回显在网页上。如果需要在表单上显示某个值，但不希望用户改变这个值，就可以使用这种方法。

12.3 检查框和单选按钮标签

本节介绍用于在 HTML 表单上生成检查框和单选按钮的标签。这些标签必须嵌套在<html:form>标签中，包括：

- <html:checkbox>：生成 HTML <INPUT type=checkbox> 元素。
- <html:multibox>：在表单上生成复选框元素。
- <html:radio>：生成 HTML <INPUT type=radio>元素。

从 htmltaglibs 应用的主页 index.jsp 上选择“CheckBox”链接，就可以访问使用检查框

和单选按钮标签的样例网页 CheckBox.jsp，如图 12-4 所示。

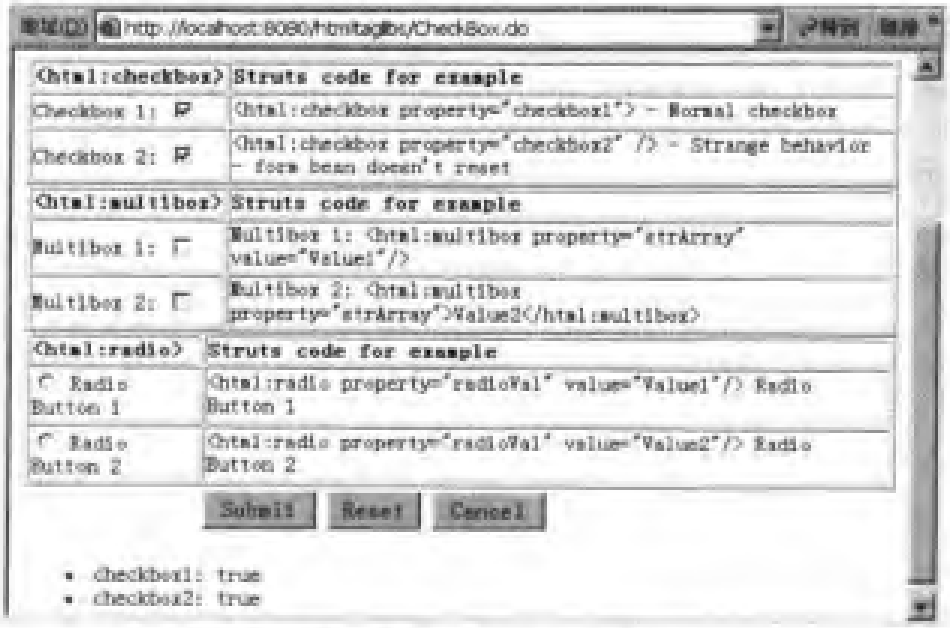

图 12-4 CheckBox.jsp 网页

如图 12-4 所示的网页包含两个<html:checkbox>标签、两个<html:multibox>标签，还有两个 <html:radio>标签。例程 12-4 为 CheckBox.jsp 的源代码。

例程 12-4 CheckBox.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html:html>
<head>
<title>Checkboxes and Radio Buttons</title>
</head>
<body bgcolor="white">

<h3>Checkboxes and Radio Buttons</h3>

<p>This page provides examples of the following Struts HTML tags:<br>
<ul>
<li>&lt;html:checkbox&gt;</li>
<li>&lt;html:multibox&gt;</li>
<li>&lt;html:radio&gt;</li>
</ul>

<html:form action="CheckBox.do">

<table border="1" width="100%">
  <tr>
    <th align="left" width="20%">
```

```
      &lt;html:checkbox&gt;
      </th>
      <th align="left" width="80%">
      Struts code for example
      </th>
    </tr>
    <tr>
      <td align="left">
        Checkbox 1:
        <html:checkbox property="checkbox1"/>
      </td>
      <td align="left">
          &lt;html:checkbox property="checkbox1"&gt;
          - Normal checkbox
      </td>
    </tr>
    <tr>
      <td align="left">
        Checkbox 2:
        <html:checkbox property="checkbox2" />
      </td>
      <td align="left">
          &lt;html:checkbox property="checkbox2" /&gt;
          - Strange behavior - form bean doesn't reset
      </td>
    </tr>

</table>

<table border="1" width="100%">
  <tr>
    <th align="left" width="20%">
      &lt;html:multibox&gt;
    </th>
    <th align="left" width="80%">
      Struts code for example
    </th>
  </tr>
  <tr>
    <td align="left" width="20%">
      Multibox 1:
      <html:multibox property="strArray" value="Value1"/>
    </td>
    <td align="left" width="80%">
      Multibox 1:
      &lt;html:multibox property="strArray" value="Value1"/&gt;
```

```
        </td>
      </tr>
      <tr>
        <td align="left" width="20%">
          Multibox 2:
          <html:multibox property="strArray">Value2</html:multibox>
        </td>
        <td align="left" width="80%">
          Multibox 2:
          &lt;html:multibox property="strArray"&gt;Value2&lt;/html:multibox&gt;
        </td>
      </tr>
    </table>

    <table border="1" width="100%">
      <tr>
        <th align="left" width="20%">
        &lt;html:radio&gt;
        </th>
        <th align="left" width="80%">
        Struts code for example
        </th>
      </tr>
      <tr>
        <td align="left" width="20%">
          <html:radio property="radioVal" value="Value1"/>
          Radio Button 1
        </td>
        <td align="left" width="80%">
          &lt;html:radio property="radioVal" value="Value1"/&gt;
          Radio Button 1
        </td>
      </tr>
      <tr>
        <td align="left" width="20%">
          <html:radio property="radioVal" value="Value2"/>
          Radio Button 2
        </td>
        <td align="left" width="80%">
          &lt;html:radio property="radioVal" value="Value2"/&gt;
          Radio Button 2
        </td>
      </tr>
    </table>

    <table border="0" width="100%">
```

```
    <tr>
      <td align="left" width="20%"> </td>
      <td align="left">
        <html:submit>Submit</html:submit>
        <html:reset>Reset</html:reset>
        <html:cancel>Cancel</html:cancel>
      </td>
    </tr>
  </table>
  <logic:present name="CheckBoxForm" scope="session">
          <ul>
           <li>checkbox1: <bean:write name="CheckBoxForm" property="checkbox1" />
           <li>checkbox2: <bean:write name="CheckBoxForm" property="checkbox2" />
  </logic:present >
  </html:form>

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

与 CheckBox.jsp 网页的表单对应的 ActionForm Bean 为 CheckBoxForm.java，它的属性与表单中的各个元素对应。例程 12-5 为 CheckBoxForm.java 的源程序。

例程 12-5　CheckBoxForm.java

```
package htmltaglibs.forms;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public class CheckBoxForm extends ActionForm {

  // Default bean constructor
  public CheckBoxForm() { }

  // For <html:checkbox> sample code
  private boolean checkbox1;
  public boolean getCheckbox1() { return this.checkbox1; }
  public void setCheckbox1(boolean checkbox1) { this.checkbox1 = checkbox1;}

  // For <html:checkbox> sample code
  private boolean checkbox2;
  public boolean getCheckbox2() { return this.checkbox2; }
  public void setCheckbox2(boolean checkbox2) { this.checkbox2 = checkbox2;}

  // For <html:multibox> sample code
  private String strArray[] = new String[0];
```

```
    public String[] getStrArray() { return (this.strArray); }
    public void setStrArray(String strArray[]) { this.strArray = strArray;}

    // For <html:radio> sample code
    private String radioVal = "";
    public String getRadioVal() { return (this.radioVal); }
    public void setRadioVal(String radioVal) { this.radioVal = radioVal;}

    public void reset(ActionMapping mapping, HttpServletRequest request) {

      this.setCheckbox1(false);
//    Note: With checkbox2 never reset here, it won't ever appear "unset"
//    this.setCheckbox2(false);

      this.strArray = new String[0];
      this.radioVal = "";
    }
}
```

12.3.1 <html:checkbox>标签

<html:checkbox>标签在表单上生成标准的 HTML 检查框。假如 ActionForm Bean 中的某个属性只有两种可选值（如 true 和 false），就可以在表单中用<html:checkbox>标签来表示。<html:checkbox>的使用方法为：

```
<html:checkbox property="checkbox1"/>
```

以上代码生成的 HTML 内容如下：

```
<input type="checkbox" name="checkbox1" value="true">
```

CheckBox.jsp 包含两个<html:checkbox>标签，它们分别和 CheckBoxForm Bean 中的 checkbox1 和 checkbox2 属性关联。在 CheckBoxForm Bean 中，checkbox1 和 checkbox2 属性必须定义为 boolean 类型：

```
private boolean checkbox1;
public boolean getCheckbox1() { return this.checkbox1; }
public void setCheckbox1(boolean checkbox1) { this.checkbox1 = checkbox1;}

private boolean checkbox2;
public boolean getCheckbox2() { return this.checkbox2; }
public void setCheckbox2(boolean checkbox2) { this.checkbox2 = checkbox2;}
```

<html:checkbox>有一个 value 属性，用来设置用户选中检查框时的值，value 的默认值为 true。可以采用以下方式来显式指定 value 属性：

```
<html:checkbox property="checkbox1" value="true"/>
```

以上代码表示如果用户选择了这个检查框，就把 CheckBoxForm Bean 中的 checkbox1 属性设置为 true。

如果把 value 属性设置为 false，例如：

```
<html:checkbox property="checkbox1" value="false"/>
```

则该代码表示，如果用户选择了这个检查框，就把 CheckBoxForm Bean 中的 checkbox1 属性设置为 false。

以上规则很容易让人混淆。例如，当 value="false" 时，如果用户没有选择这个检查框，就把 ActionForm 中对应的属性设置为 true。

为了使检查框能正常工作，必须在 ActionForm Bean 的 reset()方法中对其复位。当<html:checkbox>的 value 属性为 true 时，必须在 reset()方法中把对应的属性设置为 false。当<html:checkbox>的 value 属性为 false 时，必须在 reset()方法中把对应的属性设置为 true。

在本例中，为了演示检查框没有复位的情形，只把 checkbox1 属性复位：

```
this.setCheckbox1(false);
// Note: With checkbox2 never reset here, it won't ever appear "unset"
// this.setCheckbox2(false);
```

以上 reset()方法没有复位 checkbox2 的值，在这种情况下，会导致 checkbox2 无法正常工作。一旦用户选择了 checkbox2，以后 checkbox2 将永远为选中状态。即使用户取消了选择，checkbox2 的值仍为 true，如图 12-5 所示。

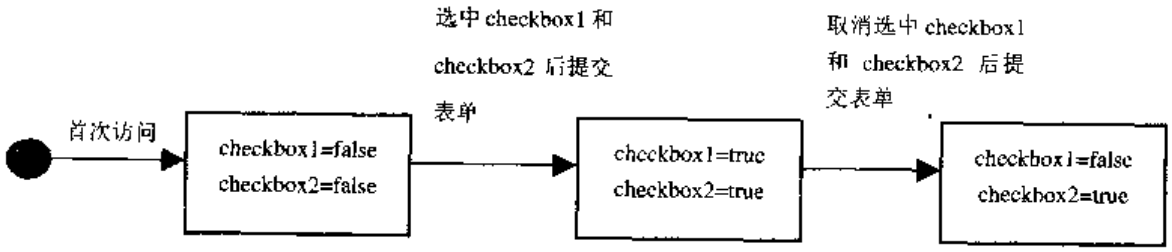

图 12-5　checkbox1 和 checkbox2 的状态图

12.3.2　<html:multibox>标签

<html:multibox>标签和<html:checkbox>一样，可以提供 HTML <input type="checkbox" > 元素。区别在于<html:multibox>标签可以生成复选框，它和 ActionForm Bean 的关联方式不一样。

如果应用中有多个 CheckBox，并且希望在 ActionForm Bean 中用单个数组来表示它们，就可以采用<html:multibox>。<html:multibox>的使用方法如下：

步骤

（1）在 ActionForm Bean 中定义一个数组，来存放所有 CheckBox 的值：

```
// For <html:multibox> sample code
  private String strArray[] = new String[0];
  public String[] getStrArray() { return (this.strArray); }
  public void setStrArray(String strArray[]) { this.strArray = strArray;}
```

（2）其次，在表单中加入<html:multibox>元素，通过设置 property="strArray" 来把它和 ActionForm Bean 中的数组关联。

（3）对于每个<html:multibox>元素，设置它的初始值，有以下两种方式：

```
<html:multibox property="strArray" value="Value1"/>
```

或者

```
<html:multibox property="strArray">Value2</html:multibox>
```

当用户提交表单时，所有被选中的复选框的值都会被存放到 ActionForm Bean 中的相应数组中。如果某个复选框没有被选中，那么数组就不会包含它的值。例如，如果用户选中了本例的两个复选框，那么 CheckBoxForm Bean 的 strArray 数组的内容为{"Value1","Value2"}。

与<html:checkbox>相比，<html:multibox>具有更高的灵活性，<html:multibox>可以动态决定被选中的复选框的数目。只要这些复选框的 property 属性相同，而 value 属性不一样，它们就和 ActionForm Bean 的同一个数组对应。此外，<html:checkbox>只能和 ActionForm Bean 中 boolean 类型的属性对应，而<html:multibox>没有这样的限制。

12.3.3 <html:radio>标签

<html:radio>标签提供 HTML<input type="radio">元素，表示单选按钮。多个<html:radio> 标签可以成组使用，例如，本应用中包含两个<html:radio>标签：

```
<html:radio property="radioVal" value="Value1"/>
<html:radio property="radioVal" value="Value2"/>
```

以上标签的 property 属性相同，而仅仅是 value 不同，它们都和 CheckBoxForm Bean 中的 radioVal 属性对应。它们生成的 HTML 内容如下：

```
<input type="radio" name="radioVal" value="Value1">
<input type="radio" name="radioVal" value="Value2">
```

由于以上两个<input>元素使用同一个 name 属性，因此在同一时刻，只允许用户选择一个按钮。假定用户选择了第一个 value 值为“Value1”的按钮，然后提交表单，CheckBoxForm Bean 中的 radioVal 属性将被设为“Value1”。如果两个按钮都没选中，CheckBoxForm Bean 中的 radioVal 属性将被设为空字符串“”。

12.4　下拉列表和多选列表标签

本节介绍用于在 HTML 表单上生成下拉列表（只支持单项选择）或多选列表（支持多项选择）的标签。这些标签必须嵌套在<html:form>标签中，包括：

- <html:select>：生成 HTML <select>元素。
- <html:option>：生成 HTML <option>元素。
- <html:options>：生成一组 HTML <option>元素。
- <html:optionsCollection>：生成一组 HTML <option>元素。

从 htmltaglibs 应用的主页 index.jsp 上选择“HtmlSelect”链接，就可以访问本节的样例网页 HtmlSelect.jsp，如图 12-6 所示。

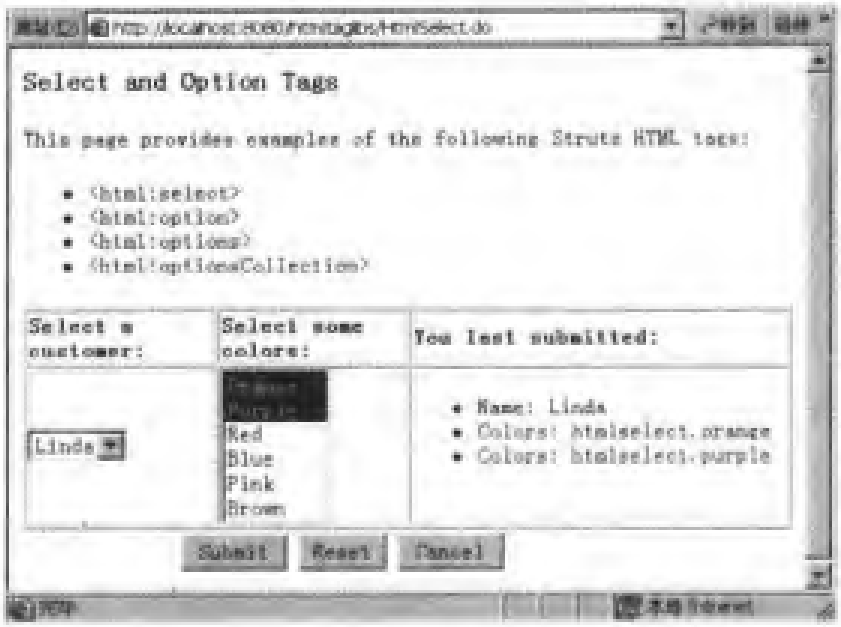

图 12-6　HtmlSelect.jsp 网页

图 12-6 的网页定义了两个列表：客户列表和颜色列表，其中客户列表为下拉列表，只支持单项选择；颜色列表为多选列表，支持多项选择。例程 12-6 为 HtmlSelect.jsp 的源代码。

例程 12-6　HtmlSelect.jsp

```
<%@ page language="java" import="htmltaglibs.beans.*, java.util.*" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html:html>
<head>
<title>Select and Option Tags</title>
</head>
<body bgcolor="white">
```

```
<h3>Select and Option Tags</h3>
<p>This page provides examples of the following Struts HTML tags:<br>
<ul>
<li>&lt;html:select&gt;</li>
<li>&lt;html:option&gt;</li>
<li>&lt;html:options&gt;</li>
<li>&lt;html:optionsCollection&gt;</li>
</ul>
<html:form action="HtmlSelect.do">

<table border="1" width="100%">

  <tr>
    <%
      Vector colorCollection = new Vector();
      colorCollection.add(
          new org.apache.struts.util.LabelValueBean("Pink", "htmlselect.pink"));
      colorCollection.add(
          new org.apache.struts.util.LabelValueBean("Brown", "htmlselect.brown"));
      pageContext.setAttribute("colorCollection", colorCollection);
    %>
    <th align="left" width="25%" >Select a customer: </th>
    <th align="left" width="25%" >Select some colors: </th>
    <th align="left" width="50%" >You last submitted: </th>
  </tr>

  <tr>
    <td align="left" width="25%" >
      <html:select property="custId">
        <html:optionsCollection property="customers"
                                label="name"
                                value="custId" />
      </html:select>
    </td>
    <td align="left" width="25%" >
      <html:select property="colors" size="6" multiple="true" >

        <%-- Specify some options using the basic version of the tag --%>
        <html:option value="htmlselect.orange">Orange</html:option>
        <html:option value="htmlselect.purple">Purple</html:option>

        <%-- Specify some by referring to a properties file --%>
        <html:option value="htmlselect.red" bundle="htmlselect.Colors" key="htmlselect.red"/>
        <html:option value="htmlselect.blue" bundle="htmlselect.Colors" key="htmlselect.blue"/>
```

```
            <%-- Specify some from our collection of LabelValueBean's --%>
            <html:options collection="colorCollection"
                          property="value"
                          labelProperty="label" />

          </html:select>
        </td>

        <td align="left" width="50%" >
          <logic:present name="HtmlSelectForm" scope="session">
            <ul>
              <li>Name: <bean:write name="HtmlSelectForm" property="cust.name" />
              <logic:iterate id="element" name="HtmlSelectForm" property="cust.favColors">
              <li>Colors: <bean:write name="element" />
              </logic:iterate>
            </ul>
          </logic:present>
        </td>
      </tr>
    </table>

    <table border="0" width="100%">
      <tr>
        <td align="left" width="20%"> </td>
        <td align="left">
          <html:submit>Submit</html:submit>
          <html:reset>Reset</html:reset>
          <html:cancel>Cancel</html:cancel>
        </td>
      </tr>
    </table>

    </html:form>
    </html:html>
```

与 HtmlSelect.jsp 网页的表单对应的 ActionForm 为 HtmlSelectForm Bean，它的属性和表单中的各个元素对应。例程 12-7 为 HtmlSelectForm.java 的源程序。

例程 12-7　HtmlSelectForm.java

```
package htmltaglibs.forms;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionErrors;
```

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import htmltaglibs.beans.CustomerBean;

public class HtmlSelectForm extends ActionForm {

  private CustomerBean customers[];
  public CustomerBean [] getCustomers() { return this.customers; }
  public void setCustomers(CustomerBean [] customers) {
    this.customers = customers;

  }

  private CustomerBean cust = new CustomerBean();
  public CustomerBean getCust() { return cust; }
  public void setCust(CustomerBean cust) { this.cust = cust; }

  private int custId;
  public int getCustId() { return this.custId; }
  public void setCustId(int custId) { this.custId = custId; }

  private String colors[];
  public String [] getColors() { return this.colors; }
  public void setColors(String [] colors) { this.colors = colors; }

  // Default bean constructor
  public HtmlSelectForm() {

    customers = new CustomerBean[3];

    for (int i=0; i<3 ; i++ ) {
      customers[i] = new CustomerBean();
      customers[i].setCustId(i);
    }

    customers[0].setName("Tom");
    customers[1].setName("Linda");
    customers[2].setName("Jane");
  }

  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.cust = new CustomerBean();
    this.colors = new String[0];
    this.custId = -1;
  }
```

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {

    ActionErrors errors = new ActionErrors();

    // if custId is -1, then no customer was chosen yet.
    if (custId == -1) {
      return errors;
    }
    /*
     * No real validation done. Just set values based on input.
     */
    this.customers[this.custId].setFavColors(this.colors);
    this.cust = this.customers[this.custId];
    return errors;
  }
}
```

12.4.1　<html:select>标签

<html:select>标签生成 HTML <select>元素。它可以在表单上创建下拉列表或多选列表。在<html:select>标签中可以包含多个<html:option>，<html:options>和<html:optionCollections>标签。<html:select>标签的基本形式为：

```
<html:select property="propertyname" multiple="true" size="6">
  [1 or more <html:option>, <html:options>, <html:optionCollections> tags]
</html:select>
```

<html:select>标签有以下重要属性：

- size 属性：指定每次在网页上显示的可选项的数目。
- multiple 属性：指定是否支持多项选择，如果设置为 true，就表示多选列表，支持多项选择；否则表示下拉列表，只支持单项选择。默认值为 false。
- property 属性：与 ActionForm Bean 中的某个属性对应，这个属性用来存放用户在列表上选中选项的值。在单项选择的情况下，ActionForm Bean 中的对应属性应该定义为简单类型（不能为数组）。在多项选择的情况下，ActionForm Bean 中的对应属性应该定义为数组类型，以便存放用户选择的多个选项。

例如，htmlselect.jsp 中的客户列表为下拉列表，颜色列表为多选列表：

```
<html:select property="custId">
<html:select property="colors" size="6" multiple="true" >
```

客户列表的 property 属性为 custId，与 HtmlSelectForm 的 custId 属性对应；颜色列表的 property 属性为 colors，与 HtmlSelectForm 的 colors 属性对应。在 HtmlSelectForm 中定义了 custId 和 colors 属性，custId 为简单类型，而 colors 属性为数组类型：

```
private int custId;
public int getCustId() { return this.custId; }
public void setCustId(int custId) { this.custId = custId; }

private String colors[];
public String [] getColors() { return this.colors; }
public void setColors(String [] colors) { this.colors = colors; }
```

12.4.2 <html:option>标签

<html:option>标签生成 HTML <option>元素。这个标签被嵌套在<html:select>标签中，代表列表的一个可选项。可选项的 Label（即在网页上的显示值，如图 12-7 所示）有两个来源：

- 在<html:option>和</html:option>之间的文本内容。
- 由<html:option>标签的 key、locale 和 bundle 属性指定的 Resource Bundle 中的内容。

图 12-7 颜色列表

例如，在如图 12-7 所示的颜色列表由提供了一组颜色选项：

```
<html:select property="colors" size="6" multiple="true" >
    <%-- Specify some options using the basic version of the tag --%>
    <html:option value="htmlselect.orange">Orange</html:option>
    <html:option value="htmlselect.purple">Purple</html:option>

    <%-- Specify some by referring to a properties file --%>
    <html:option value="htmlselect.red" bundle="htmlselect.Colors" key="htmlselect.red"/>
    <html:option value="htmlselect.blue" bundle="htmlselect.Colors" key="htmlselect.blue"/>
    ......
</html:select>
```

在以上代码中，Orange 和 Purple 选项的显示值直接在标签中指定。Red 和 Blue 选项的显示值来源于 Resource Bundle。<html:option>的 bundle 属性指定 Resource Bundle，它和 Struts 配置文件中<message-resources>元素的 key 属性匹配。在 Struts 配置文件中定义了如下<message-resources> 元素：

```
<message-resources parameter="HtmlSelectColors" key="htmlselect.Colors" />
```

以上<message-resources>元素配置的 Resource Bundle 的资源文件为 HtmlSelectColors.properties。该文件内容如下：

```
htmlselect.red=Red
htmlselect.blue=Blue
```

<html:option>元素的 key 属性和以上资源文件中的消息 key 匹配。

把列表的可选项的显示文本存放在 Resource Bundle 中，而不是直接在 JSP 文件中指定，有利于实现 Struts 应用的国际化。

<html:option>元素的 value 属性指定可选项的实际值。例如，Orange 选项的实际值为“htmlselect.orange”，Purple 选项的实际值为“htmlselect.purple”。当用户选中这两项后提交表单时，HtmlSelectForm 的 colors 属性的内容为{“htmlselect.orange”, “htmlselect. purple”}。

12.4.3　<html:options>标签

<html:options>标签提供一组 HTML<option>元素。在<html:select>元素中可以包含多个<html:options>元素。例如，在本例的颜色列表中包含了一个<html:options>标签：

```
<html:select property="colors" size="6" multiple="true" >
    ……
    <%-- Specify some from our collection of LabelValueBean's --%>
    <html:options    collection="colorCollection"
                     property="value"
                     labelProperty="label" />
</html:select>
```

<html:options>标签的 collection 属性指定存放可选项的集合，这个集合应该存在于 page 范围内。在 htmlselect.jsp 的开头定义了 colorCollection 集合，为 Vector 类型，它包含两个 org.apache.struts.util.LabelValueBean 类型的 JavaBean 对象，colorCollection 集合被存放在 page 范围中：

```
<%
    Vector colorCollection = new Vector();
    colorCollection.add(
        new org.apache.struts.util.LabelValueBean("Pink", "htmlselect.pink"));
    colorCollection.add(
        new org.apache.struts.util.LabelValueBean("Brown", "htmlselect.brown"));
    pageContext.setAttribute("colorCollection", colorCollection);
%>
```

每个 LabelValueBean 实例代表了一个可选项，它有两个属性：label 和 value，分别代表可选项的显示值和实际值。<html:options>元素的 property 属性指定集合中可选项的实际值，在本例中为 LabelValueBean 的 value 属性；labelProperty 属性指定集合中可选项的显示值，在本例中为 LabelValueBean 的 label 属性。<html:options>标签和 colorCollection 集合的

对应关系如图 12-8 所示。

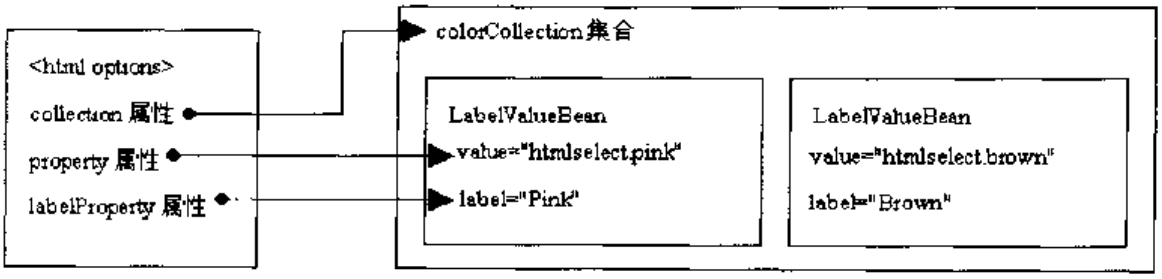

图 12-8 <html:options>标签和 colorCollection 集合的对应关系

12.4.4 <html:optionsCollection>标签

<html:optionsCollection>标签提供一组 HTML <option>元素。在<html:select>元素中可以包含多个<html:optionsCollection>元素。例如，在本例的客户列表中包含了一个<html:optionsCollection>标签：

```
<html:select property="custId">
    <html:optionsCollection property="customers"
                            label="name"
                            value="custId" />
</html:select>
```

<html:optionsCollection>的 name 属性指定包含可选项集合的 JavaBean 的名字。在以上代码中没有指定 name 属性，那么将使用和表单关联的 HtmlSelectForm Bean。property 属性指定可选项集合，在本例中为 HtmlSelectForm 的 customers 属性。以下为 HtmlSelectForm 类中定义 customers 属性的代码：

```
private CustomerBean customers[];
public CustomerBean [] getCustomers() { return this.customers; }
public void setCustomers(CustomerBean [] customers) {
    this.customers = customers;
}
```

customers 集合中包含一组 CustomerBean。每个 CustomerBean 实例代表了列表的一个可选项，它有两个属性：name 和 custId，分别代表可选项的显示值和实际值。以下是 CustomerBean 中定义 name 和 custId 属性的代码：

```
private int custId;
private String name;
private String[] favColors=new String[0];
public CustomerBean() {
}
public int getCustId(){
  return this.custId;
```

```
}
public String getName(){
   return this.name;
}
public String[] getFavColors(){
   return this.favColors;
}

public void setCustId(int custId){
   this.custId=custId;
}
public void setName(String name){
   this.name=name;
}
```

<html:optionsCollection>标签的 value 属性指定集合中可选项的实际值，在本例中为 CustomerBean 的 custId 属性；label 属性指定集合中可选项的显示值，在本例中为 CustomerBean 的 name 属性。图 12-9 显示了<html:optionsCollection>标签和 HtmlSelectForm 的对应关系。

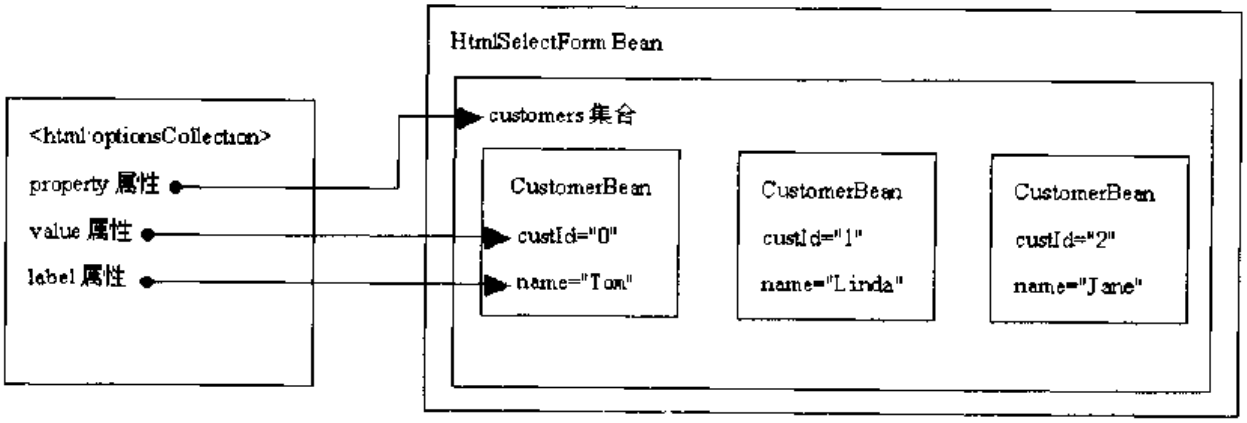

图 12-9　<html:optionsCollection>标签和 HtmlSelectForm 的对应关系

12.5　在表单中上传文件标签

文件上传指的是用户通过浏览器把用户本地的文件上传到 Web 服务器端。<html:file>标签生成 HTML <input type=file>元素，提供从 HTML 表单中上传文件的功能。

从 htmltaglibs 应用的主页 index.jsp 上选择“HtmlFile”链接，就可以访问使用<html:file>标签的样例网页 HtmlFile.jsp，如图 12-10 所示。

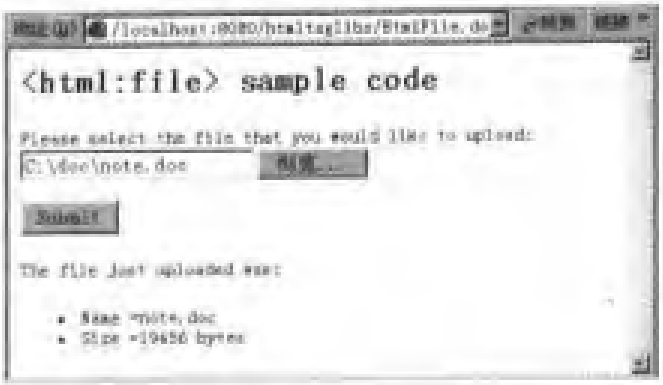

图 12-10 HtmlFile.jsp 网页

例程 12-8 为 HtmlFile.jsp 的源代码。

例程 12-8 HtmlFile.jsp

```
<%@ page language="java" %>
<%@ page import="org.apache.struts.action.*" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html:html>
<head>
<title>&lt;html:file&gt; sample code</title>
</head>
<body bgcolor="white">

<h1>&lt;html:file&gt; sample code</h1>
<!--
        The most important part is to declare your form's enctype
        to be "multipart/form-data", and to have an html:file
        element that maps to your ActionForm's FormFile property
-->
<html:form action="HtmlFile.do" enctype="multipart/form-data">
        Please select the file that you would like to upload:<br />
        <html:file property="file" /><br /><br />
        <html:submit />
</html:form>

<p>
<logic:notEmpty name="HtmlFileForm" property="fname" >
    The file just uploaded was:<p>
    <ul>
      <li>Name =<bean:write name="HtmlFileForm" property="fname" />
      <li>Size =<bean:write name="HtmlFileForm" property="size" />
    </ul>
</logic:notEmpty>

</body>
```

```
</html:html>
```

与 Htmlfile.jsp 网页上的表单对应的 ActionForm 为 HtmlFileForm Bean，例程 12-9 为 HtmlFileForm.java 的源程序。

例程 12-9　HtmlFileForm.java

```
package htmltaglibs.forms;

import org.apache.struts.action.ActionForm;
import org.apache.struts.upload.FormFile;
import org.apache.struts.upload.MultipartRequestHandler;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionMapping;

public class HtmlFileForm extends ActionForm
{

    // Default bean constructor
    public HtmlFileForm() { }

    /**
     * The file that the user has uploaded
     */
    private FormFile file;
    public FormFile getFile() { return this.file; }
    public void setFile(FormFile file) { this.file = file; }

    /**
     * The name of the file - only for displaying results
     */

    private String fname;
    public String getFname() { return this.fname; }
    public void setFname(String fname) { this.fname = fname; }
    /**
     * The size of the file - only for displaying results
     */
    private String size;
    public String getSize() { return this.size; }
    public void setSize(String size) { this.size = size; }
}
```

具体的文件上传操作由 HtmlFileAction 来完成，例程 12-10 为 HtmlFileAction.java 的源程序。

例程 12-10　HtmlFileAction.java

```
package htmltaglibs.actions;
```

```
//import all the necessary packages here
.....

public class HtmlFileAction extends Action
{

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
    throws Exception {

        String dir=servlet.getServletContext().getRealPath("/upload");

        HtmlFileForm hff = (HtmlFileForm) form;

        // org.apache.struts.upload.FormFile contains the uploaded file
        FormFile file = hff.getFile();

        // If no file was uploaded (e.g. first form load), then display View
        if (file == null ) {
                return mapping.findForward("success");
        }

        // Get the name and file size
        String fname = file.getFileName();
        String size = Integer.toString(file.getFileSize()) + " bytes";

        InputStream streamIn = file.getInputStream();
        OutputStream streamOut = new FileOutputStream(dir + "/"+fname);

        int bytesRead = 0;
        byte[] buffer = new byte[8192];
        while ((bytesRead = streamIn.read(buffer, 0, 8192)) != -1) {
            streamOut.write(buffer, 0, bytesRead);
        }

        streamOut.close();
        streamIn.close();
        // Populate the form bean with the results for display in the View
        hff.setFname(fname);
        hff.setSize(size);

        // Clean up our toys when done playing
        file.destroy();

        // Forward to default display
        return mapping.findForward("success");
    }
}
```

12.5.1　<html:file>标签

<html:file>标签可以方便地实现文件上传的功能。样例代码如下：

```
<html:form action="HtmlFile.do" method="POST" enctype="multipart/form-data">
        Please select the file that you would like to upload:<br />
        <html:file property="file" /><br /><br />
        <html:submit />
</html:form>
```

使用<html:file>标签需要注意以下几点：

- <html:file>必须嵌套在<html:form>标签中。
- <html:form>标签的 method 属性必须设为“POST”。
- <html:form>标签的编码类型 enctype 属性必须设为“multipart/form-data”。
- <html:file>标签必须设置 property 属性，这个属性和 ActionForm Bean 中 FormFile 类型的属性对应。

12.5.2　在 ActionForm Bean 中设置 FormFile 属性

与 JSP 文件中的<html:file property="file" />对应，在 ActionForm Bean 中必须定义一个名为“file”的属性。这个属性必须为 org.apache.struts.upload.FormFile 类型。代码如下：

```
/**
 * The file that the user has uploaded
 */
private FormFile file;
public FormFile getFile() { return this.file; }
public void setFile(FormFile file) { this.file = file; }
```

12.5.3　在 Action 类中处理文件上传

文件上传比传送 HTTP 表单的其他数据要复杂。幸运的是，Struts 框架提供了这方面的功能。如果要处理实际的文件上传和保存操作，可以采用以下代码：

```
String dir=servlet.getServletContext().getRealPath("/upload");
HtmlFileForm hff = (HtmlFileForm) form;

// org.apache.struts.upload.FormFile contains the uploaded file
FormFile file = hff.getFile();

// If no file was uploaded (e.g. first form load), then display View
if (file == null ) {
        return mapping.findForward("success");
}
```

```
// Get the name and file size
String fname = file.getFileName();
String size = Integer.toString(file.getFileSize()) + " bytes";

InputStream streamIn = file.getInputStream();
OutputStream streamOut = new FileOutputStream(dir + "/"+fname);

int bytesRead = 0;
byte[] buffer = new byte[8192];
while ((bytesRead = streamIn.read(buffer, 0, 8192)) != -1) {
    streamOut.write(buffer, 0, bytesRead);
}

streamOut.close();
streamIn.close();
```

以上代码创建了一个读取用户上传文件的 InputStream 对象，然后创建了一个把上传数据写到目标文件中的 OutputStream 对象。接下来通过一个循环把数据从源文件写到目标文件中。在本例中，目标文件存放于<CATALINA_HOME>/webapps/htmltaglibs/upload 目录下。

12.6 <html:errors>标签

<html:errors>标签用于输出错误消息。从 htmltaglibs 应用的主页 index.jsp 上选择“HtmlError”链接，就可以访问本节的样例网页 HtmlError.jsp，如图 12-11 所示。

图 12-11 HtmlError.jsp 网页

选中图 12-11 中的检查框，然后提交表单，将会显示两种类型的错误：一种是全局错误，还有一种是字段级错误。如图 12-12 所示为显示错误消息的 HtmlError.jsp 网页。

图 12-12　显示错误消息的 HtmlError.jsp 网页

例程 12-11 为 HtmlError.jsp 的代码。

例程 12-11　HtmlError.jsp

```
<%@ page language="java" %>
<%@ page import="org.apache.struts.action.*" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:html>
<head>
<title>&lt;html:errors&gt; sample code</title>
</head>
<body bgcolor="white">

<h1>&lt;html:errors&gt; sample code</h1>
<b>1. &lt;html:errors property="org.apache.struts.action.GLOBAL_MESSAGE"/&gt;</b>
<br>This will display page-level errors, if any, right here:<p>
<html:errors property="org.apache.struts.action.GLOBAL_MESSAGE"/>
<b>2. &lt;html:errors bundle="HtmlErrors" /&gt;</b>
<br>This will display ALL errors, if any, right here with no formatting:<p>
<html:errors bundle="HtmlErrors" />
<br>
```

```
<html:form action="HtmlError.do">

<table border="1" width="100%">

  <tr>
    <th align="center" width="35%">
      Checking box will generate error
    </th>
    <th align="left" width="65%">
     &lt;html:errors property="checkbox1" bundle="HtmlErrors" /&gt;
    </th>
  </tr>

  <tr>
    <td align="center">
      <html:checkbox property="checkbox1"/>
    </td>
    <td align="left">
       <html:errors property="checkbox1" bundle="HtmlErrors" />
    </td>
  </tr>

  <tr>
    <td align="right">
      <html:submit>submit</html:submit>
    </td>
    <td align="left">
      <html:reset>Reset</html:reset>
      <html:cancel>Cancel</html:cancel>
    </td>
  </tr>

</table>

</html:form>

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

在 HtmlError.jsp 中定义了一个表单，和这个表单对应的 ActionForm 为 HtmlErrorForm Bean，它在 validate()方法中执行表单验证，生成一系列错误消息。例程 12-12 为 HtmlErrorForm.java 的源程序。

例程 12-12　HtmlErrorForm.java

```
package htmltaglibs.forms;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;


public class HtmlErrorForm extends ActionForm {

  // Default bean constructor
  public HtmlErrorForm() { }

  private boolean checkbox1;
  public boolean getCheckbox1() { return this.checkbox1; }
  public void setCheckbox1(boolean checkbox1) { this.checkbox1 = checkbox1; }

  // ---------------------------------------------------- Public Methods
    /**
     * Reset all properties to their default values.
     *
     * @param mapping The mapping used to select this instance
     * @param request The servlet request we are processing
     */
    public void reset(ActionMapping mapping, HttpServletRequest request) {
          this.setCheckbox1(false);
    }

    /**
     * Validate the properties posted in this request. If validation errors are
     * found, return an <code>ActionErrors</code> object containing the errors.
     * If no validation errors occur, return <code>null</code> or an empty
     * <code>ActionErrors</code> object.
     *
     * @param mapping The current mapping (from struts-config.xml)
     * @param request The servlet request object
     * @return ActionErrors The ActionErrors object containing the errors.
     */
    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request) {

        ActionErrors errors = new ActionErrors();
        /*
         * If the checkbox is checked, display error messages
```

```
         */
        if ( this.getCheckbox1() ) {

            // First, a GLOBAL_ERROR message for the entire page.
            errors.add(ActionMessages.GLOBAL_MESSAGE,
                    new ActionMessage("error.global.fromform") );
            // Also, display a specific error for this parameter
            errors.add("checkbox1", new ActionMessage("error.checkbox"));
        }

        return errors;
    }
}
```

12.6.1 错误消息的来源

<html:errors>标签在 request 和 session 范围内寻找 ActionMessages（或其子类 ActionErrors）集合对象，再从 ActionMessages 集合对象中读取 ActionMessage 对象，把 ActionMessage 对象包含的消息文本显示到网页上。<html:errors>标签处理类获取 ActionMessages 对象的代码如下：

```
// Were any error messages specified?
ActionMessages errors = null;
try {
    errors = TagUtils.getInstance().getActionMessages(pageContext, name);
} catch (JspException e) {
    TagUtils.getInstance().saveException(pageContext, e);
    throw e;
}
```

以上代码调用 TagUtils 类的 getActionMessages()方法来获取 ActionMessages，name 参数指定 ActionMessages 对象存放在 request 或 session 范围内的属性 key，默认值为 Globals.ERROR_KEY。getActionMessages()方法将依次搜索 request 和 session 范围，根据 name 参数检索出匹配的 ActionMessages 对象。

在 ActionForm Bean 和 Action 类中都可以生成 ActionMessages 对象。ActionForm Bean 的 validate()方法执行表单验证，返回 ActionErrors 对象，Struts 的控制器组件 RequestProcessor 然后把 ActionErrors 对象存放在 request 范围内，存放时的属性 key 为 Globals.ERROR_KEY。以下是 HtmlErrorForm 的 validate()方法：

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    /*
     * If the checkbox is checked, display error messages
     */
```

```
        if ( this.getCheckbox1() ) {
            // First, a GLOBAL_ERROR message for the entire page.
            errors.add(ActionMessages.GLOBAL_MESSAGE,
                    new ActionMessage("error.global.fromform") );
            // Also, display a specific error for this parameter
            errors.add("checkbox1", new ActionMessage("error.checkbox"));
        }
        return errors;
    }
```

在 Action 类的 execute()方法中可以进行数据逻辑验证，如果验证失败，将生成 ActionMessages 对象。以下是 helloapp 应用的 LogonAction 的 execute()方法：

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception {
        /*
         * Validate the request parameters specified by the user
         * Note: Basic field validation done in HelloForm.java
         *       Business logic validation done in HelloAction.java
         */
        ActionMessages errors = new ActionMessages();
        String userName = (String)((HelloForm) form).getUserName();

        String badUserName = "Monster";

        if (userName.equalsIgnoreCase(badUserName)) {
            errors.add("username", new ActionMessage("hello.dont.talk.to.monster",
            badUserName ));
            saveErrors(request, errors);
            return (new ActionForward(mapping.getInput()));
        }
        ......
}
```

以上代码先创建了 ActionMessages 对象，它用来存放 ActionMesssage 对象，最后调用 Action 基类的 saveErrors()方法。saveErrors()方法把 ActionMessages 对象保存在 request 范围内，以下是在 Action 基类中定义的 saveErrors()方法：

```
protected void saveErrors(HttpServletRequest request, ActionMessages errors) {
        // Remove any error messages attribute if none are required
        if ((errors == null) || errors.isEmpty()) {
            request.removeAttribute(Globals.ERROR_KEY);
            return;
        }
        // Save the error messages we need
```

```
        request.setAttribute(Globals.ERROR_KEY, errors);
    }
```

此外，Validator 验证框架也能生成 ActionMessages 对象，详细内容请参见本书的第 10 章（Validator 验证框架）。

12.6.2 格式化地显示错误消息

<html:errors>标签能够格式化地显示 ActionMessage 对象包含的消息文本，而这些消息文本预先存放在 Resource Bundle 中。例如，在 HtmlErrors.properties 文件中包含如下错误消息文本：

```
error.global.fromform=<li>This is a global error </li>
error.checkbox=<li>This is an error that is related to CheckBox1</li>
errors.footer=<hr>
errors.header=<h3><font color="red">Validation Error</font></h3>
```

<html:errors>标签能够识别错误消息文本中的 HTML 元素，如<li>元素、<hr>元素、<h3>元素和<font>元素。在 HtmlErrors.properties 文件中定义了 errors.header 和 errrors.footer，它们分别生成消息主体的页头和页尾，如图 12-13 所示。

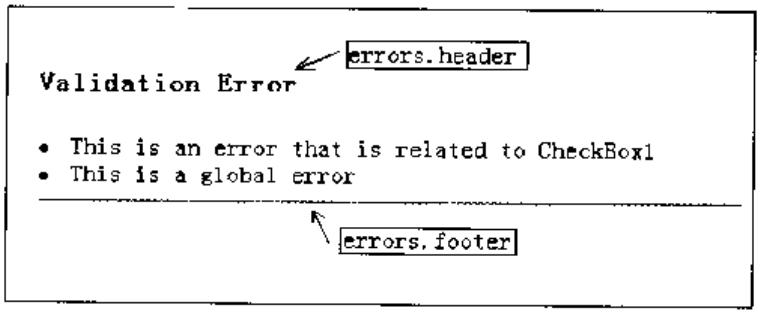

图 12-13 格式化地显示错误消息

12.6.3 <html:errors>标签的用法

<html:errors>标签可以放在网页的任何地方，既可以位于 HTML 表单内，也可以位于 HTML 表单外。<html:errors>标签输出的错误消息和它在网页中的位置无关。<html:errors>标签具有以下重要属性：

- name：指定 ActionMessages 对象存放在 request 或 session 范围内的属性 key。标签处理类将根据这一属性 key 来检索 request 或 session 范围的 ActionMessages 对象。默认值为 Globals.ERROR_KEY。
- propertity：指定消息属性。如果此项没有设置，将显示 ActionMessages 对象中所有的 ActionMessage。
- bundle：指定 Resource Bundle。如果此项没有设置，将从应用默认的 Resource Bundle 中获取消息文本。

ActionMessages 的 add(java.lang.String property, ActionMessage message)方法的 property 参数用于设置消息属性。同一消息属性可以对应多条消息，例如：

```
errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage("error.global.error1") );
errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage("error.global.error2") );
```

<html:errors>标签的 propertity 属性和以上 ActionMessages 的 add()方法的 property 属性匹配，<html:errors>标签能够输出 ActionMessages 集合中和指定消息属性对应的所有消息。

本应用的 Struts 配置文件定义了两个 Resource Bundle：

```
<message-resources parameter="HtmlErrors" />
<message-resources parameter="HtmlErrors" key="HtmlErrors" />
```

第一个 Resource Bundle 没有设置 key 属性，表示应用默认的 Resource Bundle。为了简化起见，这两个 Resource Bundle 的 parameter 属性相同，它们都使用同一个资源文件：HtmlErrors.properties 文件。

<html:errors>标签的属性和 ActionMessages、ActionMessage 以及 Resource Bundle 的对应关系，如图 12-14 所示。

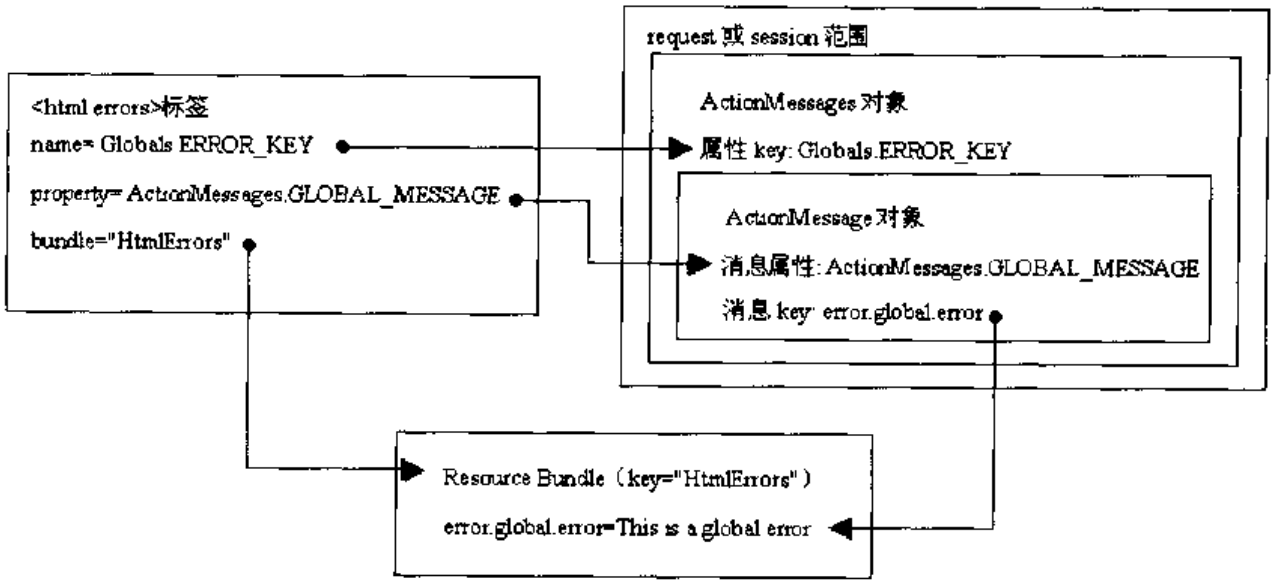

图 12-14　<html:errors>标签的属性和 ActionMessages、ActionMessage 以及 Resource Bundle 的对应关系

本例中，提供了使用<html:errors>的三种方法，下面分别进行讨论。

1．显示全局消息

全局消息指的是不和特定表单字段关联的消息，消息属性为 ActionMessages.GLOBAL_MESSAGE。在 HtmlErrorForm 的 validate()方法中生成如下全局消息：

```
errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage("error.global.fromform") );
```

ActionMessages.GLOBAL_MESSAGE 是一个常量，它的值为“org.apache.struts.action.GLOBAL_MESSAGE”，它代表全局消息。

在样例网页中指定显示全局消息的代码为：

```
<html:errors property="org.apache.struts.action.GLOBAL_MESSAGE"/>
```

由于以上<html:errors>标签没有指定 bundle 属性，所以<html:errors>标签将从应用默认的 Resource Bundle 中读取消息。

2. 显示所有的消息

如果在<html:errors>标签中没有设置 property 属性，将显示 ActionMessages 对象中所有的消息。本例中显示所有消息的代码为：

```
<html:errors bundle="HtmlErrors" />
```

以上代码指定了 Resource Bundle，在 struts-config.xml 文件中对这个 Resource Bundle 的配置为：

```
<message-resources parameter="HtmlErrors" key="HtmlErrors" />
```

<html:errors>标签的 bundle 属性和以上<message-resources>元素的 key 属性匹配。

3. 显示和特定表单字段关联的消息

在 HtmlErrorForm 的 validate()方法中，如果检查框被选中，将生成如下和检查框相关的错误消息：

```
errors.add("checkbox1", new ActionMessage("error.checkbox"));
```

本例中指定显示和表单中检查框相关错误消息的代码为：

```
<html:errors property="checkbox1" bundle="HtmlErrors" />
```

12.7 <html:messages>标签

<html:messages>标签和<html:errors>标签有些相似，也能够在网页上输出消息，不过两者的使用方法有些差别。

例如，在本书第 3 章介绍的 addressbook 应用中，当用户插入记录成功后，就会转到 confirm.jsp，它显示一条“The new Address has been inserted into the database”消息，如图 12-15 所示。

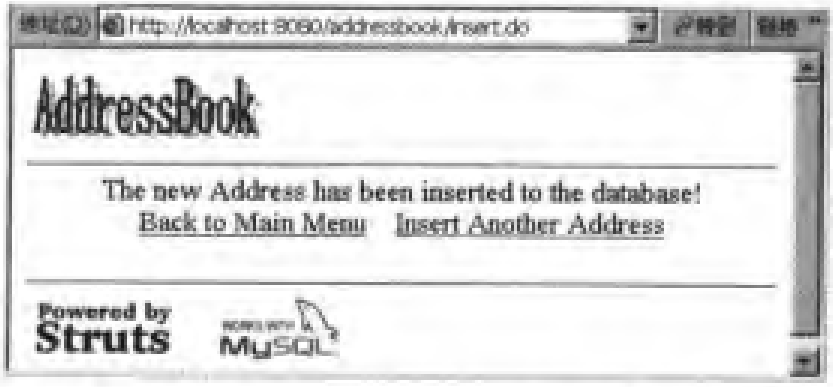

图 12-15 confirm.jsp 网页显示正常消息

在 confirm.jsp 中，用于显示正常消息的代码如下：

```
<html:messages id="message" message="true">
    <td><bean:write name="message"/></td>
</html:messages>
```

<html:messages>标签有以下一些重要属性：

- name：指定 ActionMessages 对象存放在 request 或 session 范围内的属性 key。标签处理类将根据这一属性 key 来检索 request 或 session 范围的 ActionMessages 对象。
- message 属性：指定消息的来源。如果为 true，则从 request 或 session 范围内检索出属性 key 为 Globals.MESSAGE_KEY 的 ActionMessages 对象，此时 name 属性无效；如果为 false，则根据 name 属性来检索 ActionMessages 对象，如果此时没有设置 name 属性，将采用默认值 Globals.ERROR_KEY。message 属性的默认值为 false。
- id 属性：用来命名从消息集合中检索出的每个 ActionMessage 对象，它和<bean:write>标签的 name 属性匹配。在上例中，<html:messages>标签的处理类每次从消息集合中取出一个 ActionMessage 对象，就把它命名为“message”，<bean:write>标签接着把这个名为“message”的 ActionMessage 对象的消息输出到网页上。

在 addresbook 应用的 InsertAction.java 的 execute()方法中，如果成功插入了一条记录，就创建一条正常消息，把它保存在 ActionMessages 集合中，接着再调用 saveMessages()方法，把 ActionMessages 集合存放到 request 范围内。其代码如下：

```
// If we had no errors, then add a confirmation message
ActionMessages actionMessages = new ActionMessages();
actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
                                new ActionMessage("record.inserted"));
saveMessages(request,actionMessages);
```

在 Action 基类中定义了 saveMessages(request,message)方法，它把 ActionMessages 集合存放到 request 范围内。其代码如下：

```
protected void saveMessages(HttpServletRequest request,
                            ActionMessages messages) {
    // Remove any messages attribute if none are required
    if ((messages == null) || messages.isEmpty()) {
        request.removeAttribute(Globals.MESSAGE_KEY);
        return;
    }
    // Save the messages we need
    request.setAttribute(Globals.MESSAGE_KEY, messages);
}
```

12.8 小　结

本章详细介绍了 Struts HTML 标签库中各种标签的作用和使用方法。HTML 标签库中的大部分标签用于生成 HTML 表单中的元素，这些标签包括：

- <html:text>：生成 HTML <INPUT type=text> 元素。
- <html:hidden>：生成 HTML <INPUT type=hidden >元素。
- <html:submit>：生成 HTML <INPUT type=submit >元素。
- <html:cancel>：在表单上生成取消按钮。
- <html:reset>：生成 HTML <INPUT type=reset >元素。
- <html:checkbox>：生成 HTML <INPUT type=checkbox> 元素。
- <html:multibox>：在表单上生成复选框元素。
- <html:radio>：生成 HTML <INPUT type=radio>元素。
- <html:file>：实现文件上传功能。

以上多数标签的 property 属性都和 ActionForm Bean 中的某个属性匹配。有些标签，如<html:checkbox>和<html:file>标签，对 ActionForm Bean 中的对应属性的类型进行了限制。<html:checkbox>标签要求 ActionForm Bean 中的对应属性为 boolean 类型，<html:file>标签要求 ActionForm Bean 中的对应属性为 org.apache.struts.upload.FormFile 类型。

HTML 标签库中的有些标签，如<html:option>，<html:errors>和<html:messages>，可以和 Resource Bundle 紧密关联，便于实现 Struts 应用的国际化。这些标签都可以设置 bundle 属性，它和 Struts 配置文件的<message-resources>元素的 key 属性匹配，用来指定 Resource Bundle。

第 13 章　Struts Bean 标签库

Struts Bean 标签库中的标签可以访问已经存在的 JavaBean 以及它们的属性，还可以定义新的 Bean，把它存放在 page 范围内或者用户指定的范围内，供网页内其他元素访问。

有一些 Bean 标签可以方便地访问 HTTP 请求的 Header 信息、请求参数或 Cookie，把这些信息存放在一个新定义的 JavaBean 中。Bean 标签库中的标签大致分为以下三类：

- 用于访问 HTTP 请求信息或 JSP 隐含对象的 Bean 标签。
- 用于访问 Web 应用资源的 Bean 标签。
- 用于定义或输出 JavaBean 的 Bean 标签。

本章通过一个 Struts 应用例子：beantaglibs 应用，来讲解如何使用 Struts Bean 标签。beantaglibs 应用的主页 index.jsp 如图 13-1 所示，它提供了访问其他样例网页的链接，这些样例网页用于演示各种 Bean 标签的使用方法。

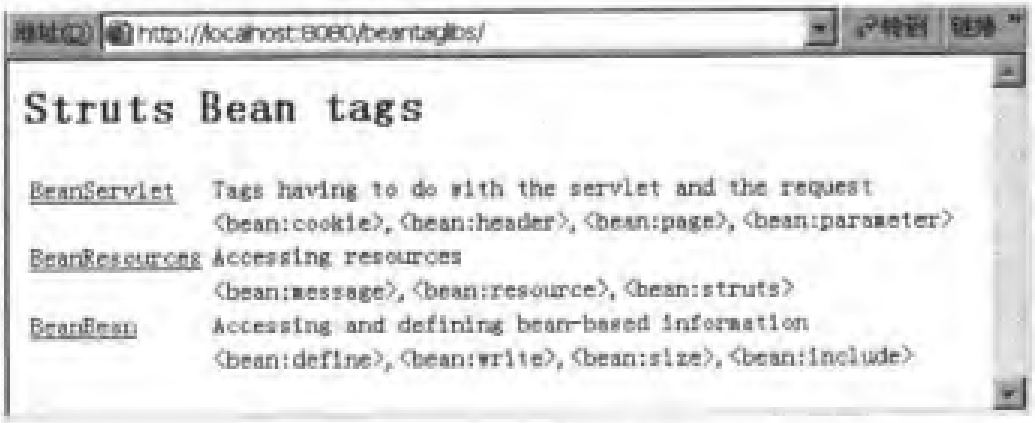

图 13-1　beantaglibs 应用的主页 index.jsp

本章介绍的 beantaglibs 应用的源代码位于配套光盘的 sourcecode/beantaglibs 目录下，如果要在 Tomcat 上发布这个应用，只要把整个 beantaglibs 目录拷贝到<CATALINA_HOME>/webapps 目录下即可。

13.1　访问 HTTP 请求信息或 JSP 隐含对象

本节介绍以下标签：

- <bean:cookie>：访问 Cookie 信息。
- <bean:header>：访问 HTTP 请求中的 Header 信息。
- <bean:parameter>：访问请求参数。
- <bean:page>：访问 JSP 隐含对象。

从 beantaglibs 应用的主页 index.jsp 上选择“BeanServlet”链接，就可以访问本节的样

例网页 BeanServlet.jsp，如图 13-2 所示。

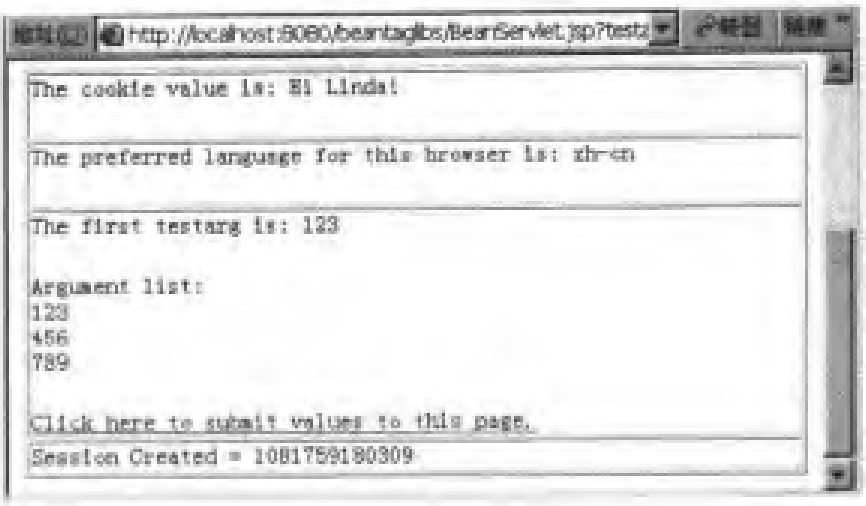

图 13-2 BeanServlet.jsp

首次访问 BeanServlet.jsp，网页上的 cookie value 为“firsttime”，testarg 及 Argument list 的内容均为“noarg”。当选择网页上的“Click here to submit values to this page”时，将会看到如图 13-2 所示的网页。例程 13-1 为 BeanServlet.jsp 的源代码。

例程 13-1 BeanServlet.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ page import="javax.servlet.http.Cookie" %>
<html:html>
<head>
<title>Bean Servlet sample code</title>
</head>
<body bgcolor="white">

<h3>Bean Servlet sample code</h3>

<p>This page provides examples of the following Struts BEAN tags:<br>
<ul>
<li>&lt;bean:header&gt;</li>
<li>&lt;bean:parameter&gt;</li>
<li>&lt;bean:cookie&gt;</li>
<li>&lt;bean:page&gt;</li>
</ul>

<table border="1" width="100%">
  <%--
  The following section contains creating and displaying a cookie.
  --%>
```

```
<tr>
  <td align="left">
   <bean:cookie id="cookie" name="/tags/cookiedemo" value="firsttime"/>
   <%
     if (cookie.getValue().equals("firsttime")) {
       Cookie c = new Cookie("/tags/cookiedemo", "Hi Linda!");
       c.setComment("A test cookie");
       c.setMaxAge(3600); //60 seconds times 60 minutes
       response.addCookie(c);
     }
     %>
    The cookie value is: <bean:write name="cookie" property="value"/><P>
  </td>
</tr>
<%--
The following section gets a value from the request header.
--%>
<tr>
   <td align="left">
      <bean:header id="lang" name="Accept-Language"/>
      The preferred language for this browser is: <bean:write name="lang"/><P>
   </td>
</tr>
<%--
The following section looks at some parameters passed in.
--%>
<tr>
   <td align="left">
    <bean:parameter id="arg1" name="testarg" value="noarg"/>
    The first testarg is: <bean:write name="arg1"/>
   <P>
   <bean:parameter id="arg2" multiple="yes" name="testarg" value="noarg"/>

   Argument list:<BR>
   <%  for (int i = 0; i < arg2.length; i++) {
       out.write(arg2[i] + "<BR>");
   }
   %>
   <P>

   <html:link page="/BeanServlet.jsp?testarg=123&testarg=456&testarg=789">
   Click here to submit values to this page.</html:link>
  </td>
</tr>
<%--
The following section finds a value from the session.
```

```
--%>
<tr>
  <td align="left">
    <bean:page id="this_session" property="session"/>
    Session Created = <bean:write name="this_session"
                                  property="creationTime"/>
  </td>
</tr>
</table>
</body>
</html:html>
```

13.1.1 <bean:header>标签

<bean:header>标签用于检索 HTTP 请求中的 Header 信息，它具有以下属性：

- id 属性：定义一个 java.lang.String 类型的变量，这个变量存放在 page 范围内。
- name 属性：指定需要检索的 Header 信息。

<bean:header>标签的示范代码如下：

```
<bean:header id="lang" name="Accept-Language"/>
The preferred language for this browser is: <bean:write name="lang"/><P>
```

在以上代码中，<bean:header>标签检索 HTTP 请求 Header 中的“Accept-Language”信息，然后把它赋值给 lang 变量，lang 变量为字符串类型，存放在 page 范围内。接下来再通过<bean:write>标签输出 lang 变量的值。

以上<bean:header>标签执行的操作等价于以下的 JSP 程序：

```
<%
    String id="lang";
    String name="Accept-Language";
    String value =((HttpServletRequest) pageContext.getRequest()).getHeader(name);
    pageContext.setAttribute(id, value);
%>
```

13.1.2 <bean:parameter>标签

<bean:parameter>标签用于检索 HTTP 请求参数，它具有以下属性：

- id 属性：定义一个 java.lang.String 类型的变量，这个变量存放在 page 范围内。
- name 属性：指定请求参数名。
- value 属性：指定请求参数的默认值。

<bean:parameter>标签的示范代码如下：

```
<bean:parameter id="arg1" name="testarg" value="noarg"/>
The first testarg is: <bean:write name="arg1"/>
```

以上代码的<bean:parameter>标签定义了一个名为“arg1”的字符串类型的变量，默认

值为“noarg”。如果访问 BeanServlet.jsp 的 URL 不包含“testarg”请求参数，如：

```
http://localhost:8080/beantaglibs/BeanServlet.jsp
```

那么 arg1 变量的值为“noarg”，以上代码的输出内容为：

```
The first testarg is: noarg
```

如果访问 BeanServlet.jsp 的 URL 中包含“testarg”请求参数，如：

```
http://localhost:8080/beantaglibs/BeanServlet.jsp?testarg=123&testarg=456&testarg=789
```

那么 arg1 变量的值为第一个名为“testarg”请求参数的值，以上代码的输出内容为：

```
The first testarg is: 123
```

如果希望检索出所有和参数名匹配的请求参数，应该设置<bean:parameter>标签的 multiple 属性（可以设置为任意一个字符串），此时 id 属性定义的变量不再是 java.lang.String 类型，而是字符串数组类型，用于存放所有和 name 属性匹配的请求参数值，例如：

```
<bean:parameter id="arg2" multiple="yes" name="testarg" value="noarg"/>
Argument list:<BR>
<%  for (int i = 0; i < arg2.length; i++) {
        out.write(arg2[i] + "<BR>");
    }
%>
```

以上<bean:parameter>标签定义的 arg2 变量为字符串数组类型，对于 URL：

```
http://localhost:8080/beantaglibs/BeanServlet.jsp?testarg=123&testarg=456&testarg=789
```

arg2 变量的内容为{“123”，“456”，“789”}，以上代码的输出内容为：

```
Argument list:
123
456
789
```

13.1.3　<bean:cookie>标签

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。每个 Cookie 包含一对 name/value 信息，分别表示 Cookie 的名字和值。javax.servlet.http.Cookie 类用来定义一个 Cookie。

<bean:cookie>标签可以检索保存在浏览器中的 Cookie，它有以下属性：

- id 属性：定义一个 javax.servlet.http.Cookie 类型的变量，这个变量被存放在 page 范围内。
- name 属性：指定 Cookie 的名字。
- value 属性：指定 Cookie 的默认值。如果由 name 属性指定的 Cookie 不存在，就

使用 value 属性指定的默认值。

<bean:cookie>标签的示范代码如下：

```
<bean:cookie id="cookie" name="/tags/cookiedemo" value="firsttime"/>
<%
        if (cookie.getValue().equals("firsttime")) {
          Cookie c = new Cookie("/tags/cookiedemo", "Hi Linda!");
          c.setComment("A test cookie");
          c.setMaxAge(3600); //60 seconds times 60 minutes
          response.addCookie(c);
        }
%>
The cookie value is: <bean:write name="cookie" property="value"/><P>
```

以上代码的<bean:cookie>标签定义了一个 javax.servlet.http.Cookie 类型的变量，名为 cookie，<bean:cookie>标签先检索名为“/tags/cookiedemo”的 Cookie 是否存在，如果存在，就把它的值赋给 cookie 变量的 value 属性；如果不存在，cookie 变量的 value 属性就采用默认值“firsttime”。

接下来在 JSP 程序代码中创建一个名为“/tags/cookiedemo”的 Cookie，Cookie 值为“Hi Linda!”，再调用 response.addCookie()方法把它写到客户浏览器端。最后再通过<bean:write>标签输出 cookie 变量的 value 属性。

当第一次访问 BeanServlet.jsp 时，由于在客户浏览器端还不存在名为“/tags/cookiedemo”的 Cookie，所以 cookie 变量的 value 属性采用默认值“firsttime”，与此同时，BeanServlet.jsp 把名为“/tags/cookiedemo”的 Cookie 写到客户浏览器端。当第二次访问 BeanServlet.jsp 时，<bean:cookie>标签检索到客户浏览器端的 Cookie，因此 cookie 变量的 value 属性为该 Cookie 的值“Hi Linda!”。

<bean:cookie>标签还有一个 multiple 属性，如果设置了 multiple 属性（可以设置为任意一个字符串），可以检索出所有和 Cookie 名字匹配的 Cookie。此时，id 属性定义了一个 Cookie 数组类型的变量，而不是单个 Cookie 类型的变量。<bean:cookie>标签的 multiple 属性和<bean:parameter>标签的 multiple 属性很相似。

13.1.4 <bean:page>标签

<bean:page>标签用于检索 JSP 隐含对象，如 session、request 和 response 等。<bean:page>标签具有以下属性：

- id 属性：定义一个引用隐含对象的变量，这个变量存放在 page 范围内。
- property 属性：指定隐含对象的名字，可选值包括 application、config、request、response 和 session。

<bean:page>标签的示范代码如下：

```
<bean:page id="this_session" property="session"/>
Session Created = <bean:write name="this_session"
                              property="creationTime"/>
```

以上代码的<bean:message>标签定义了一个名为“this_session”的变量，它引用 JSP 网页的 session 隐含对象。接下来的<bean:write>标签输出 this_session 变量的 creationTime 属性。

13.2　访问 Web 应用资源

本节介绍以下 Bean 标签：

- <bean:message>：显示 Resource Bundle 中的消息。
- <bean:resource>：把 Web 资源装载到一个 JavaBean 中。
- <bean:struts>：访问 Struts 的内在配置对象。
- <bean:include>：包含一个 Web 资源。

从 beantaglibs 应用的主页 index.jsp 上选择“BeanResources”链接，就可以访问本节的样例网页 BeanResources.jsp，如图 13-3 所示。

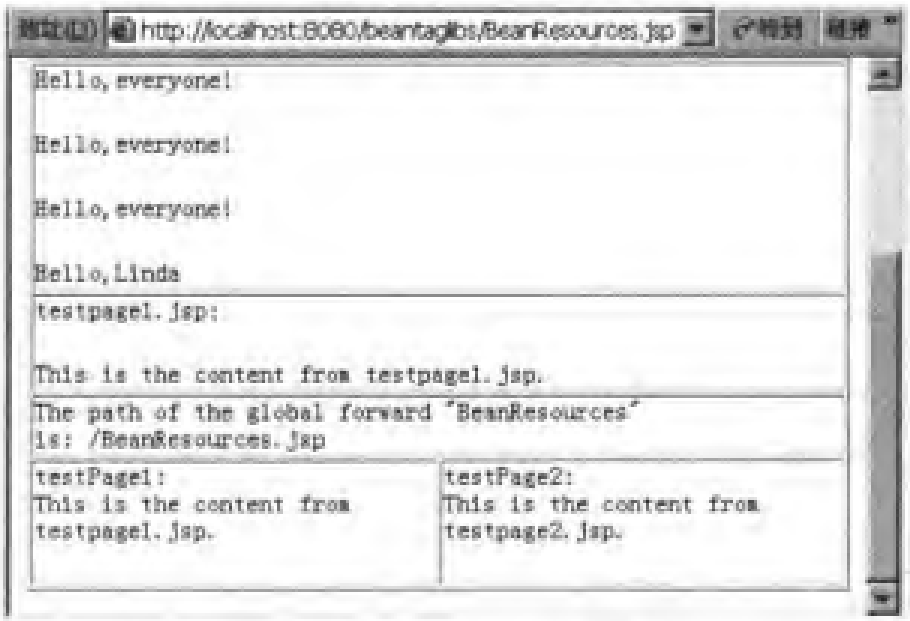

图 13-3　BeanResources.jsp 网页

例程 13-2 为 BeanResources.jsp 的源代码。

例程 13-2　BeanResources.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ page import="beantaglibs.SomeBean" %>
<html:html>
<head>
<title>Bean Resources sample code</title>
</head>
<body bgcolor="white">
```

```
<h3>Bean Resources sample code</h3>

<p>This page provides examples of the following Struts BEAN tags:<br>
<ul>
<li>&lt;bean:message&gt;</li>
<li>&lt;bean:resource&gt;</li>
<li>&lt;bean:struts&gt;</li>
</ul>

<table border="1" width="100%">
  <%--
  The following section contains code getting messages.
  --%>
  <tr>
    <td align="left" colspan="2">
     <%
         request.setAttribute("stringBean","hello");
         SomeBean bean=new SomeBean();
         bean.setName("hello");
         request.setAttribute("someBean",bean);
     %>
     <bean:message bundle="special" key="hello"/><P>
     <bean:message bundle="special" name="stringBean"/><p>
     <bean:message bundle="special" name="someBean"
                   property="name"/><P>
     <bean:message key="hello" arg0="Linda" />
    </td>
  </tr>
  <%--
  The following section shows how to use web resources.
  --%>
  <tr>
    <td align="left" colspan="2">
     testpage1.jsp:<P>
     <bean:resource id="resource" name="/testpage1.jsp"/>
     <bean:write name="resource"/>
     </td>
 </tr>

  <%--
  The following section shows how to get a Struts configuration object.
  --%>
  <tr>
    <td align="left" colspan="2">
     The path of the global forward "BeanResources" is:
```

```
    <bean:struts id="forward" forward="BeanResources"/>
    <bean:write name="forward" property="path"/>
    </td>
  </tr>
  <%--
  The following section shows how to get an application web response.
  --%>
  <tr>
    <td align="left">
    testPage1:<BR>
    <bean:include id="tp1" page="/testpage1.jsp"/>
    <bean:write name="tp1" filter="false"/><P>
    </td>
    <td align="left">
    testPage2:<BR>
    <bean:include id="tp2" forward="testpage2"/>
    <bean:write name="tp2" filter="false"/><P>
    </td>
  </tr>
</table>
</body>
</html:html>
```

13.2.1 <bean:message>标签

<bean:message>标签用于输出 Resource Bundle 中的一条消息。<bean:message>标签的 bundle 属性指定 Resource Bundle，它和 Struts 配置文件的<message-resources>元素的 key 属性匹配。如果没有设置 bundle 属性，就采用默认的 Resource Bundle。beantaglibs 应用的 Struts 配置文件中配置了两个 Resource Bundle：

```
<message-resources parameter="ApplicationResources"   />
<message-resources parameter="SpecialResources" key="special" />
```

第一个 Resource Bundle 没有指定 key 属性，因此是默认的 Resource Bundle，它的资源文件为 ApplicationResources.properties，在这个文件中定义了一条消息：

```
hello=Hello,{0}
```

第二个 Resource Bundle 指定 key 属性为“special”，它的资源文件为 SpecialResources.properties，在这个文件中定义了一条消息：

```
hello=Hello,everyone!
```

在<bean:message>标签中指定消息 key 有三种方式：

- <bean:message>标签的 key 属性直接指定消息 key，例如：

```
<bean:message bundle="special" key="hello"/>
```

- <bean:message>标签的 name 属性指定一个可以转化为字符串的变量，这个变量的字符串值为消息 key，例如：

```
<%
    request.setAttribute("stringBean","hello");
%>
<bean:message bundle="special" name="stringBean"/>
```

以上代码定义了一个字符串类型的 stringBean 变量，值为“hello”，因此消息 key 为“hello”。

- 同时指定<bean:message>标签的 name 属性和 property 属性。name 属性指定一个 JavaBean，property 属性指定 JavaBean 的一个属性，这个 JavaBean 的属性的值就是消息 key，例如：

```
<%
        SomeBean bean=new SomeBean();
        bean.setName("hello");
        request.setAttribute("someBean",bean);
%>
<bean:message bundle="special" name="someBean" property="name"/>
```

以上代码定义了一个 SomeBean 类型的 someBean 变量，它的 name 属性为“hello”，因此消息 key 为“hello”。如图 13-4 所示为<bean:message>标签的属性和 JavaBean 及 Resource Bundle 的对应关系。

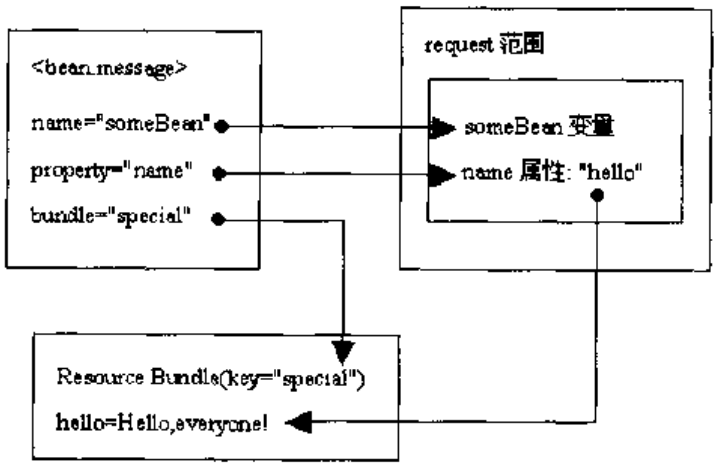

图 13-4 <bean:message>标签的属性和 JavaBean 及 Resource Bundle 的对应关系

对于带参数的复合消息，可以使用<bean:message>标签的 arg0、arg1、arg2、arg3 和 arg4 属性来设置参数值，arg0 代表第一个参数，依次类推。例如：

```
<bean:message key="hello" arg0="Linda" />
```

以上代码的<bean:message>标签没有设置 bundle 属性，因此从默认的 Resource Bundle 中获取消息，与消息 key 匹配的消息为：

```
hello=Hello,{0}
```

<bean:message>标签的 arg0 属性替代第一个参数{0}，因此以上<bean:message>标签的输出内容为：Hello,Linda。

13.2.2 <bean:resource>标签

<bean:resource>标签用于检索 Web 资源的内容，它具有以下属性：

- id 属性：定义一个代表 Web 资源的变量。
- name 属性：指定 Web 资源的路径。
- input 属性：如果没有设置 input 属性，则 id 属性定义的变量为字符串类型；如果给 input 属性设置了值(可以是任意字符串)，则 id 属性定义的变量为 java.io.InputStream 类型。

<bean:resource>标签的示范代码如下：

```
<bean:resource id="resource" name="/testpage1.jsp"/>
<bean:write name="resource"/>
```

以上代码定义的 resource 变量代表“/testpage1.jsp”资源，由于没有设置 input 属性，因此 resource 变量为字符串类型，它的值为 testpage1.jsp 的源文件内容。

13.2.3 <bean:struts>标签

<bean:struts>标签用于检索 Struts 框架内在的对象，如 ActionFormBean、ActionForward 或 ActionMapping。<bean:struts>标签的 id 属性定义一个 page 范围的变量，用来引用 Struts 框架的内在对象。必须设置 formbean、forward 或 mapping 属性中的一个属性，来指定被引用的 Struts 内在对象：

- formbean 属性：指定 ActionFormBean 对象，和 Struts 配置文件的<form-bean>元素匹配。
- forward 属性：指定 ActionForward 对象，和 Struts 配置文件的<global-forwards>元素的<forward>子元素匹配。
- mapping 属性：指定 ActionMapping 对象，和 Struts 配置文件的<action>元素匹配。

<bean:struts>标签的示范代码如下：

```
<bean:struts id="forward" forward="BeanResources"/>
<bean:write name="forward" property="path"/>
```

以上代码的<bean:struts>标签定义了一个名为“forward”的变量，它引用一个名为“BeanResources”的 ActionForward 对象，在 Struts 配置文件中，与之匹配的<forward>元素为：

```
<global-forwards>
    <forward    name="BeanResources"    path="/BeanResources.jsp"/>
    ......
</global-forwards>
```

接下来<bean:write>标签输出 ActionForward 对象的 path 属性值。

13.2.4 <bean:include>标签

<bean:include>标签和标准的 JSP 标签<jsp:include> 很相似，都可以用来包含其他 Web 资源的内容，区别在于<bean:include>标签把其他 Web 资源的内容存放在一个变量中，而不是直接显示到网页上。<bean:include>标签的 id 属性定义一个代表其他 Web 资源的变量。

可以通过<bean:include>标签的 forward、page 或 href 属性来指定其他 Web 资源：

- forward：指定全局转发路径，和 Struts 配置文件的<global-forwards>元素中的<forward>子元素匹配。
- page：指定相对于当前应用的 URI，以“/”开头。
- href：指定完整的 URL。

<bean:include>标签的示范代码如下：

```
testPage1:<BR>
<bean:include id="tp1" page="/testpage1.jsp"/>
<bean:write name="tp1" filter="false"/><P>

testPage2<BR>
<bean:include id="tp2" forward="testpage2"/>
<bean:write name="tp2" filter="false"/><P>
```

13.3 定义或输出 JavaBean

本节介绍以下 Bean 标签：

- <bean:define>：定义一个变量。
- <bean:write>：显示 JavaBean 或其属性的内容。
- <bean:size>：获得 Map 或 Collection 集合的长度。

从 beantaglibs 应用的主页 index.jsp 上选择“BeanBean”链接，就可以访问本节的样例网页 BeanBean.jsp，如图 13-5 所示。

图 13-5 BeanBean.jsp 网页

例程 13-3 为 BeanBean.jsp 的代码。

例程 13-3　BeanBean.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ page import="java.util.HashMap" %>
<%@ page import="java.util.GregorianCalendar" %>
<%@ page import="java.util.Calendar" %>
<html:html>
<head>
<title>Bean Bean sample code</title>
</head>
<body bgcolor="white">

<h3>Bean Bean sample code</h3>

<p>This page provides examples of the following Struts BEAN tags:<br>
<ul>
<li>&lt;bean:define&gt;</li>
<li>&lt;bean:write&gt;</li>
<li>&lt;bean:size&gt;</li>
</ul>

<table border="1" width="100%">

  <%--
  The following section contains code defining variables.
  --%>

<tr>
    <td align="left" colspan="2">
    <bean:define id="stringBean" value="BeanTaglibs"/>
    Application Name: <bean:write name="stringBean"/><BR>
     <% request.setAttribute("sessionBean", session); %>
     <bean:define id="contextBean" name="sessionBean" property="servletContext"/>
     Servlet Context Name: <bean:write name="contextBean"
      property="servletContextName"/><BR>
     <bean:define id="contextBean_copy" name="contextBean"
                  type="javax.servlet.ServletContext"/>
     Servlet API Major Version:
                <bean:write name="contextBean_copy" property="majorVersion"/>
  </td>
</tr>
<%--
The following section contains code getting the size of a collection.
```

```
--%>
<tr>
<%  HashMap lines = new HashMap();
      lines.put("1", "Line 1");
      lines.put("2", "Line 2");
      lines.put("3", "Line 3");
      request.setAttribute("lines", lines);
       %>
      <td align="left" colspan="2">
      <bean:size id="length" name="lines"/>
      Line Count: <bean:write name="length"/>
     </td>
</tr>
<%--
The following section contains code showing how to write bean values.
--%>
<tr>
 <%
  request.setAttribute("floatval", Float.valueOf("3.14159"));
  Calendar gc = GregorianCalendar.getInstance();
  gc.setTime(new java.util.Date(session.getCreationTime()));
  request.setAttribute("now", gc);
  String boldStart = "<B>";
  String boldEnd = "</B>";
  request.setAttribute("bs", boldStart);
  request.setAttribute("be", boldEnd);
  %>
  <td align="left" colspan="2">
  Pi is: <bean:write format="#.####" name="floatval"/><BR>
  Session Started at: <bean:write format="MM-dd-yyyy hh:mm:ss"
                                  name="now" property="time"/><BR>
  This should not be in <bean:write name="bs"/>bold
                        <bean:write name="be"/><BR>
  This should be in <bean:write name="bs" filter="false"/>bold
                    <bean:write name="be" filter="false"/><BR>
 </td>
</tr>
</table>
</body>
</html:html>
```

13.3.1 <bean:define>标签

<bean:define>标签用于定义一个变量。id 属性指定变量的名字，toScope 属性指定这个变量的存放范围，如果没有设置 toScope 属性，则这个变量存放在 page 范围内。给 id 属性

定义的变量赋值有三种方式：

- 设置 value 属性，此时 id 属性定义的变量为字符串类型，value 属性代表这个变量的字符串值。例如，以下代码定义了一个字符串类型的变量 stringBean，它的值为“BeanTaglibs”：

```
<bean:define id="stringBean" value="BeanTaglibs"/>
Application Name: <bean:write name="stringBean"/><BR>
```

- 同时设置 name 和 property 属性。name 属性指定一个已经存在的 Bean，property 属性指定已经存在的 Bean 的某个属性。id 属性定义的变量的值由 property 属性决定。例如：

```
<% request.setAttribute("sessionBean" , session); %>
<bean:define id="contextBean" name="sessionBean" property="servletContext"/>
Servlet Context Name: <bean:write name="contextBean" property="servletContextName"/><BR>
```

以上代码先把 JSP 隐含对象 session 保存在 request 范围内，取名为 sessionBean。接下来<bean:define>标签定义了一个名为“contextBean”的变量，这个变量引用 sessionBean 的 servletContext 属性，因此 contextBean 变量为 javax.servlet.ServletContext 类型，实际上引用该 Web 应用的 ServletContext 实例。

- 同时设置 name 属性和 type 属性。name 属性指定已经存在的 JavaBean，type 属性指定这个 JavaBean 的完整的类名。id 属性定义的变量引用这个已经存在的 JavaBean。例如，以下代码定义了一个名为“contextBean_copy”的变量，这个变量引用已经存在的 contextBean：

```
<bean:define id="contextBean_copy" name="contextBean"
                    type="javax.servlet.ServletContext"/>
Servlet API Major Version: <bean:write name="contextBean_copy" property="majorVersion"/>
```

对于由 name 属性指定的已经存在的 JavaBean，在默认情况下，<bean:define>标签会依次在 page、request、session 和 application 范围内寻找这个 JavaBean。此外，也可以设置<bean:define>标签的 scope 属性，明确指定这个 JavaBean 的范围，这样，<bean:define>标签只会在指定的范围内寻找 JavaBean。

13.3.2　<bean:size>标签

<bean:size>标签用于获得 Map、Collection 或数组的长度。<bean:size>标签的 id 属性定义一个 Integer 类型的变量，name 属性指定已经存在的 Map、Collection 或数组变量。id 属性定义的变量的值为 Map、Collection 或数组的长度。例如：

```
<% HashMap lines = new HashMap();
    lines.put("1", "Line 1");
    lines.put("2", "Line 2");
    lines.put("3", "Line 3");
    request.setAttribute("lines", lines);
%>
```

```
<bean:size id="length" name="lines"/>
Line Count: <bean:write name="length"/>
```

以上代码先定义了一个 HashMap 类型的 lines 变量，长度为 3，lines 变量存放在 request 范围内。接下来<bean:size>标签定义了一个 Interger 类型的变量，名为“length”。length 变量的值为 HashMap 对象的长度。

13.3.3 <bean:write>标签

<bean:write>标签用于在网页上输出某个 Bean 或它的属性的内容。<bean:write>标签的 name 属性指定已经存在的变量，例如：

```
<bean:size id="length" name="lines"/>
Line Count: <bean:write name="length"/>
```

以上的<bean:write>标签仅仅设置了 name 属性，<bean:write>标签处理类会调用 name 属性指定的变量的 toString()方法获得字符串，然后输出字符串内容。此处 name 属性指定的 length 变量为 Interger 类型，值为 3，因此输出的字符串为“3”。

如果希望输出 Bean 的某个属性值，应该同时设置<bean:write>标签的 name 属性和 property 属性。property 属性指定 Bean 的属性，例如：

```
Server Info: <bean:write name="contextBean" property="serverInfo"/>
```

以上代码将输出 contextBean 变量的 serverInfo 属性的值。

<bean:write>标签的 format 属性用于设置输出数据的格式，例如：

```
<%
  request.setAttribute("floatval", Float.valueOf("3.14159"));
  Calendar gc = GregorianCalendar.getInstance();
  gc.setTime(new java.util.Date(session.getCreationTime()));
  request.setAttribute("now", gc);
 %>
  Pi is: <bean:write format="#.####" name="floatval"/><BR>
  Session Started at: <bean:write format="MM-dd-yyyy hh:mm:ss"
                                  name="now" property="time"/><BR>
```

以上代码先定义了两个 request 范围内的变量 floatval 和 now。floatval 为 Float 类型，now 变量为 Calendar 类型。<bean:write>标签在输出 floatval 变量的内容时，把 format 属性设置为“#.####”，因此输出的内容为“3.1416”。<bean:write>标签在输出 now 变量的 time 属性值时，把 format 属性设置为“MM-dd-yyyy hh:mm:ss”，因此输出的内容为“05-25-2004 09:59:38”。

<bean:write>标签还有一个 filter 属性，默认值为 true。如果 filter 属性为 true，将把输出内容中的特殊 HTML 符号作为普通字符串来显示；如果 filter 属性为 false，则不会把输出内容中的特殊 HTML 符号转化为普通字符串。例如：

```
<%
  String boldStart = "<B>";
  String boldEnd = "</B>";
  request.setAttribute("bs", boldStart);
  request.setAttribute("be", boldEnd);
  %>
  This should not be in <bean:write name="bs"/>bold
                        <bean:write name="be"/><BR>
  This should be in <bean:write name="bs" filter="false"/>bold
                    <bean:write name="be" filter="false"/><BR>
```

以上代码先定义了两个 request 范围内的变量 bs 和 be，它们的值分别为“<B>”和“</B>”。在 HTML 语言中，“<B>”和“</B>”为粗体字的标记。第一组<bean:write>标签没有设置 filter 属性，将采用它的默认值 true，此时“<B>”和“</B>”作为普通文本，因此输出内容为：

This should not be in <B>bold </B>

第二组<bean:write>标签的 filter 属性为 false，此时“<B>”和“</B>”作为 HTML 的粗体字标记来处理，因此输出内容为：

This should be in **bold**

13.4　小　　结

本章讨论了 Bean 标签库中的标签的作用和使用方法，大多数 Bean 标签具有以下共同属性：

- id 属性：定义一个变量。默认情况下，这个变量被存放在 page 范围内。对于<bean:define>标签，它的 toScope 属性可以指定这个变量的存放范围，可选值包括：page、request、session 和 application。
- name 属性：指定已经存在的 JavaBean 的名字。property 属性指定这个 JavaBean 的某个属性。scope 属性指定这个 JavaBean 的范围，可选值包括：page、request、session 和 application。如果没有设置 scope 属性，将依次从 page、request、session 和 application 范围内寻找这个 JavaBean。

在 Bean 标签库中，使用最频繁的标签为<bean:message>和<bean:write>。<bean:message>标签用于输出 Resource Bundle 中的一条消息，<bean:write>标签用于输出某个 Bean 或其属性的内容。

第 14 章　Struts Logic 标签库

Struts Logic 标签库中的标签可以根据特定的逻辑条件来控制输出网页内容，或者循环遍历集合中的所有元素。Logic 标签库中的标签大致分为以下几类：

- 进行比较运算的 Logic 标签。
- 进行字符串匹配的 Logic 标签。
- 判断指定内容是否存在的 Logic 标签。
- 进行循环遍历的 Logic 标签。
- 进行请求转发或重定向的 Logic 标签。

本章通过一个 Struts 应用例子：logictaglibs 应用，来讲解如何使用 Struts Logic 标签。logictaglibs 应用的主页 index.jsp 如图 14-1 所示，它提供了访问其他样例网页的链接，这些样例网页用于演示各种 Logic 标签的使用方法。

图 14-1　logictaglibs 应用的主页 index.jsp

本章介绍的 logictaglibs 应用的源代码位于配套光盘的 sourcecode/logictaglibs 目录下，如果要在 Tomcat 上发布这个应用，只要把整个 logictaglibs 目录复制到<CATALINA_HOME>/webapps 目录下即可。

14.1　进行比较运算的 Logic 标签

本节将介绍以下进行比较运算的 Logic 标签：

- <logic:equal>：比较变量是否等于指定的常量。
- <logic:notEqual>：比较变量是否不等于指定的常量。
- <logic:greaterEqual>：比较变量是否大于或等于指定的常量。

- <logic:greaterThan>：比较变量是否大于指定的常量。
- <logic:lessEqual>：比较变量是否小于或等于指定的常量。
- <logic:lessThan>：比较变量是否小于指定的常量。

从 logictaglibs 应用的主页 index.jsp 上选择“LogicCompare”链接，就可以访问本节的样例网页 LogicCompare.jsp，如图 14-2 所示。

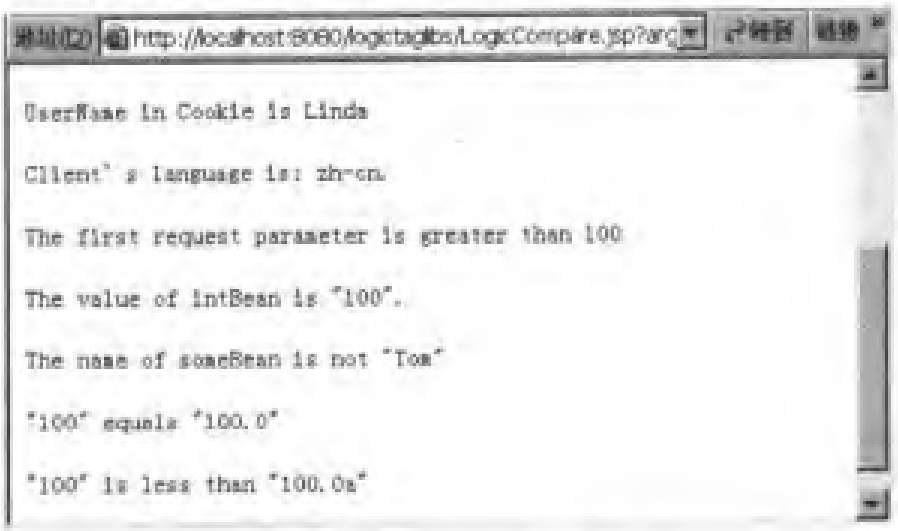

图 14-2 LogicCompare.jsp

例程 14-1 为 LogicCompare.jsp 的源代码。

例程 14-1 LogicCompare.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ page import="logictaglibs.SomeBean" %>
<html:html>
<head>
<title>Logic Compare sample code</title>
</head>
<body bgcolor="white">
<h3>Logic Compare sample code</h3>
<p>This page provides examples of the following Struts LOGIC tags:<br>
<ul>
  <li>&lt;logic:equal&gt;</li>
  <li>&lt;logic:lessEqual&gt;</li>
  <li>&lt;logic:lessThan&gt;</li>
  <li>&lt;logic:greaterEqual&gt;</li>
  <li>&lt;logic:greaterThan&gt;</li>
  <li>&lt;logic:notEqual&gt;</li>
</ul>
```

```
<%
        Cookie c = new Cookie("username", "Linda");
        c.setComment("A test cookie");
        c.setMaxAge(3600); //60 seconds times 60 minutes
        response.addCookie(c);
%>

<logic:equal cookie="username" value="Linda" >
        UserName in Cookie is Linda <p>
</logic:equal>

<logic:equal header="Accept-Language" value="zh-cn" >
        Client's language is: zh-cn. <p>
</logic:equal>

<logic:greaterThan parameter="arg1" value="100" >
        The first request parameter is greater than 100 <p>
</logic:greaterThan >

<%
        request.setAttribute("intBean",new Integer(100));
%>
<logic:equal name="intBean" value="100" >
        The value of intBean is "100".<p>
</logic:equal >

<%
        SomeBean bean=new SomeBean();
        bean.setName("Linda");
        request.setAttribute("someBean",bean);
%>
<logic:notEqual name="someBean" property="name" value="Tom" >
        The name of someBean is not "Tom" <p>
</logic:notEqual >

<% request.setAttribute("number","100"); %>
<logic:equal name="number" value="100.0" >
        "100" equals "100.0" <p>
</logic:equal >

<logic:lessThan name="number" value="100.0a" >
        "100" is less than "100.0a" <p>
</logic:lessThan >

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

所有的比较运算标签都比较一个变量和指定常量的大小。表 14-1 列出了每个标签比较结果为 true 的条件。

表 14-1 比较运算标签比较结果为 true 的条件

比较运算标签	比较结果为 true 的条件
<logic:equal>	变量等于指定常量
<logic:notEqual>	变量不等于指定常量
<logic:lessThan>	变量小于指定常量
<logic:lessEqual>	变量小于或等于指定常量
<logic:greaterThan>	变量大于指定常量
<logic:greaterEqual>	变量大于或等于指定常量

比较运算标签的 value 属性指定常量值，可以通过以下方式来设置变量：

- 设置 cookie 属性，此时变量为 cookie 属性指定的 Cookie 的值，例如：

```
<%
        Cookie c = new Cookie("username", "Linda");
        c.setComment("A test cookie");
        c.setMaxAge(3600); //60 seconds times 60 minutes
        response.addCookie(c);
%>

<logic:equal cookie="username" value="Linda" >
        UserName in Cookie is Linda <p>
</logic:equal>
```

以上代码比较名为“username”的 Cookie 的值是否为“Linda”，此处比较结果为 true，因此执行标签主体的内容。

当第一次访问 LogicCompare.jsp 时，在客户浏览器中还不存在名为“username”的 Cookie，因此以上<logic:equal>标签的比较结果为 false。在第二次访问 LogicCompare.jsp 时，<logic:equal>标签的比较结果才为 true。

- 设置 header 属性，此时变量为 header 属性指定的 HTTP 请求中的 Header 信息，例如：

```
<logic:equal header="Accept-Language" value="zh-cn" >
        Client's language is: zh-cn. <p>
</logic:equal>
```

以上代码比较 HTTP 请求 Header 中的“Accept-Language”的值是否为“zh-cn”，如果比较结果为 true，就执行标签主体的内容。

- 设置 parameter 属性，此时变量为 parameter 属性指定的请求参数值，例如：

```
<logic:greaterThan parameter="arg1" value="100" >
        The first request parameter is greater than 100 <p>
</logic:greaterThan >
```

以上代码比较名为“arg1”的请求参数的值是否大于“100”，如果比较结果为 true，就执行标签主体的内容。例如，如果请求访问 LogicCompare.jsp 的 URL 为：http://localhost:8080/logictaglibs/LogicCompare.jsp?arg1=200，就会输出标签主体的文本。

- 设置 name 属性，此时 name 属性指定被比较的变量。比较运算标签调用变量的 toString()方法，获得被比较的字符串值，例如：

```
<%
        request.setAttribute("intBean",new Integer(100));
%>
<logic:equal name="intBean" value="100" >
        The value of intBean is "100".<p>
</logic:equal >
```

以上代码先定义了一个名为“intBean”的 Integer 类型的变量。接下来<logic:equal>标签比较 intBean 变量的 toString()方法返回的字符串值是否为“100”。此处的比较结果为 true，因此将执行标签主体的内容。

在默认情况下，将依次在 page、request、session 和 application 范围内寻找 name 属性指定的变量。此外，也可以通过 scope 属性来指定变量的存在范围。

- 同时设置 name 和 property 属性，此时 name 属性指定已经存在的 JavaBean，property 属性指定 JavaBean 的属性，被比较的变量为这个属性的值，例如：

```
<%
        SomeBean bean=new SomeBean();
        bean.setName("Linda");
        request.setAttribute("someBean",bean);
%>
<logic:notEqual name="someBean" property="name" value="Tom" >
        The name of someBean is not "Tom" <p>
</logic:notEqual >
```

以上代码先定义了一个名为“someBean”的 JavaBean。接下来<logic:notEqual>标签比较 someBean 的 name 属性的值是否不等于“Tom”，此处的比较结果为 true，因此执行标签主体的内容。

如果两个字符串都可以成功地转化为数字，就比较数字的大小，否则就进行字符串比较。例如：

```
<% request.setAttribute("number","100"); %>
<logic:equal name="number" value="100.0" >
        "100" equals "100.0" <p>
</logic:equal >

<logic:lessThan name="number" value="100.0a" >
        "100" is less than "100.0a" <p>
</logic:lessThan >
```

在以上代码中，第一个<logic:equal>标签把“100”和“100.0”进行比较，由于两者都

是数字，因此比较数值是否相等，其比较结果为 true。第二个<logic:lessThan>标签把“100”和“100.0a”进行比较，由于后者是字符串，因此比较字符串“100”是否小于字符串“100.0a”，其比较结果为 true。

14.2 进行字符串匹配的 Logic 标签

本节将介绍以下进行字符串匹配的 Logic 标签：

- <logic:match>：判断变量中是否包含指定的常量字符串。
- <logic:notMatch>：判断变量中是否不包含指定的常量字符串。

从 logictaglibs 应用的主页 index.jsp 上选择“LogicMatch”链接，就可以访问本节的样例网页 LogicMatch.jsp，如图 14-3 所示。

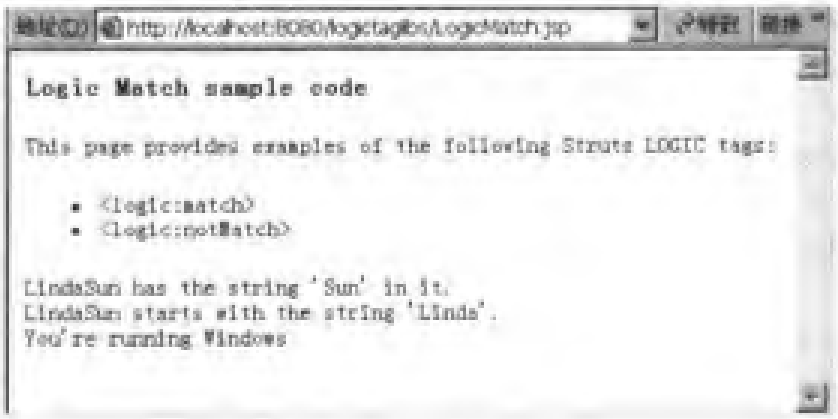

图 14-3 LogicMatch.jsp

例程 14-2 为 LogicMatch.jsp 的源代码。

例程 14-2 LogicMatch.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ page import="java.util.Enumeration" %>
<html:html>
<head>
<title>Logic Match sample code</title>
</head>
<body bgcolor="white">

<h3>Logic Match sample code</h3>

<p>This page provides examples of the following Struts LOGIC tags:<br>
<ul>
<li>&lt;logic:match&gt;</li>
```

```
<li>&lt;logic:notMatch&gt;</li>
</ul>

<%--
Variables used on this page
--%>
<%
  request.setAttribute("authorName", "LindaSun");
%>
<%--
The following section shows match and notMatch.
--%>

<logic:match name="authorName" scope="request" value="Linda">
    <bean:write name="authorName"/> has the string 'Sun' in it.
</logic:match>
<logic:notMatch name="authorName" scope="request" value="Linda">
    <bean:write name="authorName"/> doesn't have the string 'Linda' in it.
</logic:notMatch>
<BR>
<logic:match name="authorName" scope="request" value="Linda" location="start">
    <bean:write name="authorName"/> starts with the string 'Linda'.
</logic:match>
<logic:notMatch name="authorName" scope="request" value="Linda" location="start">
    <bean:write name="authorName"/> doesn't start with the string 'Linda'.
</logic:notMatch>
<BR>
<logic:match header="user-agent" value="Windows">
    You're running Windows
</logic:match>
<logic:notMatch header="user-agent" value="Windows">
    You're not running Windows
</logic:notMatch>
<BR>
</body>
</html:html>
```

所有的字符串匹配标签都判断一个变量中是否包含指定的常量字符串。表 14-2 列出了每个标签判断结果为 true 的条件。

表 14-2 字符串匹配标签判断结果为 true 的条件

字符串匹配标签	判断结果为 true 的条件
<logic:match>	变量中包含指定的常量字符串
<logic:notMatch>	变量中不包含指定的常量字符串

字符串匹配标签的 value 属性指定常量值，可以通过 cookie、header、parameter、name 和 property 属性来设置变量，它们的用法和 14.1 节介绍的比较运算标签的相应属性的用法

很相似。例如：

```
<%
  request.setAttribute("authorName", "LindaSun");
%>
<logic:match name="authorName" scope="request" value="Linda">
   <bean:write name="authorName"/> has the string 'Sun' in it.
</logic:match>
```

以上代码先定义了一个名为“authorName”的字符串变量，它的字符串值为“LindaSun”。接下来<logic:match>标签判断在 authorName 变量中是否包含“Linda”子字符串，此处的判断结果为 true，因此执行标签主体的内容。

字符串匹配标签的 location 属性指定子字符串的位置，可选值包括：

- start：子字符串位于母字符串的起始位置。
- end：子字符串位于母字符串的结尾。

例如，以下<logic:notMatch>标签判断在 authorName 变量的起始位置是否不包含“Linda”子字符串，如果比较结果为 true，就执行标签主体的内容，此处的比较结果为 false：

```
<logic:notMatch name="authorName" scope="request" value="Linda" location="start">
   <bean:write name="authorName"/> doesn't start with the string 'Linda'.
</logic:notMatch>
```

如果没有指定 location 属性，子字符串可以位于母字符串的任何位置。

14.3 判断指定内容是否存在的 Logic 标签

本节将介绍以下标签：

- <logic:empty>：判断指定的变量是否为 null，或者为空字符串“”。
- <logic:notEmpty>：判断指定的变量是否不为 null，并且不是空字符串“”。
- <logic:present>：判断指定的安全角色、用户、Cookie、HTTP 请求 Header 或 JavaBean 是否存在。
- <logic:notPresent>：判断指定的安全角色、用户、Cookie、HTTP 请求 Header 或 JavaBean 是否不存在。
- <logic:messagesPresent>：判断指定的消息是否存在。
- <logic:messagesNotPresent>：判断指定的消息是否不存在。

从 logictaglibs 应用的主页 index.jsp 上选择“LogicPresence”链接，就可以访问本节的样例网页 LogicPresence.jsp，如图 14-4 所示。

图 14-4　LogicPresence.jsp

例程 14-3 为 LogicPresence.jsp 的源代码。

例程 14-3　LogicPresence.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ page import="java.util.HashMap" %>
<%@ page import="org.apache.struts.Globals" %>
<%@ page import="org.apache.struts.action.ActionMessage" %>
<%@ page import="org.apache.struts.action.ActionErrors" %>
<html:html>
<head>
<title>Logic Presence sample code</title>
</head>
<body bgcolor="white">

<h3>Logic Presence sample code</h3>

<p>This page provides examples of the following Struts LOGIC tags:<br>
<ul>
<li>&lt;logic:empty&gt;</li>
<li>&lt;logic:messagesPresent&gt;</li>
<li>&lt;logic:messagesNotPresent&gt;</li>
<li>&lt;logic:notEmpty&gt;</li>
<li>&lt;logic:notPresent&gt;</li>
```

```
<li>&lt;logic:present&gt;</li>
</ul>

<%--
Variables used on this page
--%>
<%
ActionErrors errors = new ActionErrors();
errors.add("totallylost", new ActionMessage("application.totally.lost"));
request.setAttribute(Globals.ERROR_KEY, errors);
request.setAttribute("myerrors", errors);
request.setAttribute("emptyString", "");
%>
<%--
The following section shows empty and notEmpty.
--%>

<logic:empty name="emptyString">
    The variable named emptyString is empty!<P>
</logic:empty>
<logic:notEmpty name="emptyString">
     The variable named emptyString is not empty!<P>
</logic:notEmpty>
<P>
<%--
The following section shows present and notPresent.
--%>
<logic:present name="noSuchBean" property="noSuchProperty">
    Both noSuchBean and noSuchProperty exist!
</logic:present>
<logic:notPresent name="noSuchBean" property="noSuchProperty">
    Either noSuchBean or noSuchProperty does not exist!
</logic:notPresent>
<P>

<logic:present name="emptyString" >
    There is a JavaBean named "emptyString". <p>
</logic:present>
<logic:notPresent name="emptyString" property="noSuchProperty">
    EmptyString doesn't have such a property named "noSuchProperty".
</logic:notPresent>
<P>
<logic:present header="user-agent">
    Yep, we got a user-agent header.
</logic:present>
<logic:notPresent header="user-agent">
```

```
    No, user-agent header does not exist.
</logic:notPresent>
<P>
<%--
The following section shows messagesPresent and messagesNotPresent.
--%>
<logic:messagesPresent>
    Yes, there are errors.
</logic:messagesPresent><P>
<logic:messagesPresent name="myerrors">
    Yes, there are errors in myerrors collection.
</logic:messagesPresent><P>
<logic:messagesNotPresent message="true" >
   There are no normal messages.
</logic:messagesNotPresent><P>

<logic:messagesNotPresent property="noSuchError">
    There is no error named "SuchError".
</logic:messagesNotPresent><P>
<logic:messagesPresent property="totallylost">
    There is an error named "totallylost".
</logic:messagesPresent>

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

14.3.1　<logic:empty>和<logic:notEmpty>标签

<logic:empty>和<logic:notEmpty>标签判断指定的变量是否为空字符串“”。表 14-3 列出了每个标签判断结果为 true 的条件。

表 14-3　<logic:empty>和<logic:notEmpty>标签判断结果为 true 的条件

标 签 名	判断结果为 true 的条件
<logic: empty >	指定的变量为 null 或者为空字符串“”
<logic: notEmpty >	指定的变量不为 null，并且不为空字符串“”

可以设置<logic:empty>和<logic:notEmpty>标签的 name 属性，或者同时设置 name 属性和 property 属性，来指定变量。例如：

```
<%
request.setAttribute("emptyString", "");
%>
<logic:empty name="emptyString">
    The variable named emptyString is empty!<P>
</logic:empty>
```

以上代码先定义了一个字符串变量 emptyString，它是空字符串。接下来<logic:empty>标签判断 emptyString 变量是否为空字符串，其判断结果为 true，因此将执行标签主体的内容。

14.3.2 <logic:present>和<logic:notPresent>标签

<logic:present>和<logic:notPresent >标签判断指定的对象是否存在，表 14-4 列出了每个标签判断结果为 true 的条件。

表 14-4 <logic:present>和<logic:notPresent >标签判断结果为 true 的条件

标 签 名	判断结果为 true 的条件
<logic: present >	存在指定的对象
<logic: notPresent >	不存在指定的对象

<logic:present>和<logic:notPresent >标签具有以下属性，分别用于判断某种类型的对象是否存在：

- cookie 属性：判断指定的 cookie 是否存在。
- header 属性：判断指定的 HTTP 请求 Header 是否存在。
- role 属性：判断当前通过权限验证的用户是否具有指定的安全角色。多个安全角色之间以逗号隔开，例如：
  <logic:present role="role1,role2,role3"> code..... </logic:present>
- user 属性：判断当前通过权限验证的用户是否拥有指定的用户名。
- parameter 属性：判断指定的请求参数是否存在。
- name 属性：判断指定的 JavaBean 是否存在。
- 同时设置 name 和 property 属性：name 属性指定 JavaBean，property 属性指定 JavaBean 的某个属性，判断这个属性是否存在并且是否为 null。

下面举例说明<logic:present>和<logic:notPresent >标签的用法。

- 设置 name 属性，例如：

```
<%
request.setAttribute("emptyString", "");
%>
<logic:present name="emptyString" >
    There is a JavaBean named "emptyString".
</logic:present>
```

<logic:present>标签判断名为“emptyString”的变量是否存在，其判断结果为 true，因此执行标签主体的内容。

- 同时设置 name 属性和 property 属性，例如：

```
<logic:notPresent name="emptyString" property="noSuchProperty">
    EmptyString doesn't have such a property named "noSuchProperty".
</logic:notPresent>
<logic:notPresent name="noSuchBean" property="noSuchProperty">
```

```
    Either noSuchBean or noSuchProperty does not exist!
</logic:notPresent>
```

第一个<logic:notPresent>标签判断名为“emptyString”的 JavaBean 是否不存在，或者它的“noSuchProperty”属性是否不存在，或者“noSuchProperty”属性是否为 null。由于 emptyString 变量不存在“noSuchProperty”属性，其判断结果为 true，因此执行标签主体的内容。

第二个<logic:notPresent>标签判断名为“noSuchBean”的 JavaBean 是否不存在，或者它的“noSuchProperty”属性是否不存在，或者“noSuchProperty”属性是否为 null。由于不存在名为“noSuchBean”的 JavaBean，其判断结果为 true，因此执行标签主体的内容。

- 设置 header 属性，例如：

```
<logic:present header="user-agent">
    Yep, we got a user-agent header.
</logic:present>
```

<logic:present>标签判断名为“user-agent”的 HTTP 请求 Header 是否存在，其判断结果为 true，因此执行标签主体的内容。

14.3.3　<logic:messagesPresent>和<logic:messagesNotPresent>标签

<logic:messagesPresent>和<logic:messagesNotPresent>标签用来判断是否在 request 范围内存在指定的 ActionMessages（或其子类 ActionErrors）对象，以及在 ActionMessages 对象中是否存在特定的消息。表 14-5 列出了每个标签判断结果为 true 的条件。

表 14-5　<logic:messagesPresent>和<logic:messagesNotPresent>标签判断结果为 true 的条件

标 签 名	判断结果为 true 的条件
<logic: messagesPresent >	存在指定的消息
<logic: messagesNotPresent >	不存在指定的消息

表 14-6 列出了每个标签的属性。

表 14-6　<logic:messagesPresent>和<logic:messagesNotPresent>的属性

属　性	说　明
name	指定在 request 范围内检索 ActionMessages 对象的属性 key
message	如果为 true，则从 request 范围内检索属性 key 为 Globals.MESSAGE_KEY 的 ActionMessages 对象，此时 name 属性指定的值无效 如果为 false，则根据 name 属性指定的属性 key，从 request 范围内检索 ActionMessages 对象，如果此时没有设置 name 属性，则默认的属性 key 为 Globals.ERROR_KEY
property	指定从 ActionMessages 对象中检索某条消息的消息 key

<logic:messagesPresent>标签的处理类为 MessagesPresentTag.java，它的 condition()方法用来判断指定的消息是否存在：

```
protected boolean condition(boolean desired) throws JspException {
```

```
        ActionMessages am = null;

        String key = name;
        if (message != null && "true".equalsIgnoreCase(message)){
            key = Globals.MESSAGE_KEY;
        }
        try {
            am = TagUtils.getInstance().getActionMessages(pageContext, key);
        } catch (JspException e) {
            TagUtils.getInstance().saveException(pageContext, e);
            throw e;
        }
        Iterator iterator = (property == null) ? am.get() : am.get(property);

        return (iterator.hasNext() == desired);
    }
```

在以上的 condition()方法中，name，message 和 property 变量的初始值分别来自于<logic:messagesPresent>标签的 name，message 和 property 属性。condition()方法的判断流程如下。

（1）把 name 变量赋值给局部变量 key，这个变量将用于在 request 范围内检索 ActionMessages 对象。

（2）如果 message 变量为 true，就把局部变量 key 设为 Globals.MESSAGE_KEY。因此当<logic:messagesPresent>标签的 message 属性为 true 时，设置 name 属性是无效的。

（3）调用 TagUtils.getInstance().getActionMessages(pageContext, key)方法，获得 request 范围内的 ActionMessages 对象。

（4）如果 property 变量不为 null，将再判断在 ActionMessages 对象中是否包含由 property 变量指定的消息 key。

下面举例说明<logic:messagesPresent>和<logic:messagesNotPresent>标签的用法。首先在 JSP 程序代码中定义一个 ActionErrors 对象，它包含一条消息。然后把它分别以 Globals.ERROR_KEY 和“myerrors”作为属性 key，保存在 request 范围内：

```
<%
ActionErrors errors = new ActionErrors();
errors.add("totallylost", new ActionMessage("application.totally.lost"));
request.setAttribute(Globals.ERROR_KEY, errors);
request.setAttribute("myerrors", errors);
%>
```

以下分几种情况介绍<logic:messagesPresent>和<logic:messagesNotPresent>标签的用法：

- 未设置 name、message 和 property 属性，例如：

```
<logic:messagesPresent>
    Yes, there are errors.
</logic:messagesPresent><P>
```

默认情况下，<logic:messagesPresent>标签从 request 范围内检索属性 key 为 Globals.ERROR_KEY 的 ActionMessages 对象，其判断结果为 true，将执行标签主体的内容。

- 设置 name 属性，例如：

```
<logic:messagesPresent name="myerrors">
    Yes, there are errors in myerrors collection.
</logic:messagesPresent><P>
```

<logic:messagesPresent>标签从 request 范围内检索 key 为“myerrors”的 ActionMessages 对象，其判断结果为 true，将执行标签主体的内容。

- 设置 message 属性，例如：

```
<logic:messagesNotPresent message="true" >
  There are no normal messages.
</logic:messagesNotPresent>
```

<logic:messagesNotPresent>标签从 request 范围内检索属性 key 为 Globals.MESSAGE_KEY 的 ActionMessages 对象，由于不存在这样的 ActionMessages 对象，其判断结果为 true，将执行标签主体的内容。

- 设置 property 属性，但它指定的消息 key 不存在，例如：

```
<logic:messagesNotPresent property="noSuchError">
    There is no error named "SuchError".
</logic:messagesNotPresent>
```

<logic:messagesNotPresent>标签从 request 范围内检索属性 key 为 Globals.ERROR_KEY 的 ActionMessages 对象，然后再从 ActionMessages 对象中检索消息 key 为“noSuchError”的消息，由于不存在这样的消息，其判断结果为 true，将执行标签主体的内容。

- 设置 property 属性，并且它指定的消息 key 存在，例如：

```
<logic:messagesPresent property="totallylost">
    There is an error named "totallylost".
</logic:messagesPresent>
```

<logic:messagesPresent>标签从 request 范围内检索属性 key 为 Globals.ERROR_KEY 的 ActionMessages 对象，然后再从 ActionMessages 对象中检索消息 key 为“totallylost”的消息，判断结果为 true，将执行标签主体的内容。

14.4　进行循环遍历的 Logic 标签

<logic:iterate>是 Logic 标签库中最复杂的标签，也是用途最广的一个标签，它能够在

一个循环中遍历数组、Collection、Enumeration、Iterator 或 Map 中的所有元素。

从 logictaglibs 应用的主页 index.jsp 上选择“LogicIterate”链接，就可以访问本节的样例网页 LogicIterate.jsp，如图 14-5 所示。

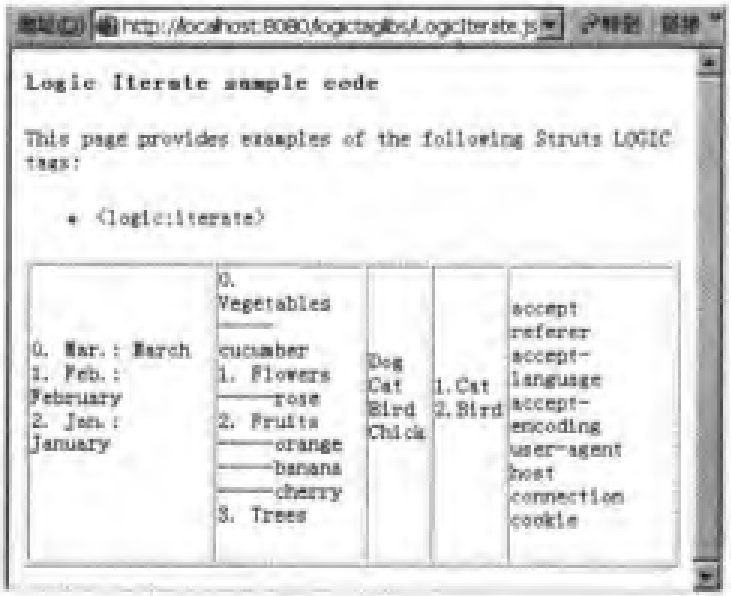

图 14-5 LogicIterate.jsp

例程 14-4 为 LogicIterate.jsp 的源代码。

例程 14-4 LogicIterate.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ page import="java.util.HashMap" %>
<%@ page import="java.util.Vector" %>
<html:html>
<head>
<title>Logic Iterate sample code</title>
</head>
<body bgcolor="white">

<h3>Logic Iterate sample code</h3>

<p>This page provides examples of the following Struts LOGIC tags:<br>

<ul>
<li>&lt;logic:iterate&gt;</li>
</ul>

<table border="1">
```

```
<tr>
<td>
<%--
Variables used on this page
--%>

<%
HashMap months = new HashMap();
months.put("Jan.", "January");
months.put("Feb.", "February");
months.put("Mar.", "March");
request.setAttribute("months", months);
%>
<%--
The following section shows iterate.
--%>
<logic:iterate id="element" indexId="ind" name="months">
   <bean:write name="ind"/>.
   <bean:write name="element" property="key"/>:
   <bean:write name="element" property="value"/><BR>
</logic:iterate><P>

</td>

<td>
<%
HashMap h = new HashMap();
String vegetables[] = {"pepper", "cucumber"};
String fruits[] = {"apple","orange","banana","cherry","watermelon"};
String flowers[] = {"chrysanthemum","rose"};
String trees[]={"willow"};
h.put("Vegetables", vegetables);
h.put("Fruits", fruits);
h.put("Flowers", flowers);
h.put("Trees",trees);
request.setAttribute("catalog", h);
%>
<%--
The following section shows iterate.
--%>

<logic:iterate id="element" indexId="ind" name="catalog">
   <bean:write name="ind"/>. <bean:write name="element" property="key"/><BR>
   <logic:iterate id="elementValue" name="element" property="value" length="3" offset="1">
        -----<bean:write name="elementValue"/><BR>
   </logic:iterate>
```

```
</logic:iterate><P>

</td>
<td>
<%
 Vector animals=new Vector();
 animals.addElement("Dog");
 animals.addElement("Cat");
 animals.addElement("Bird");
 animals.addElement("Chick");
 request.setAttribute("Animals", animals);
%>
<logic:iterate id="element" name="Animals">
    <bean:write name="element"/><BR>
</logic:iterate><p>

</td>
<td>
<logic:iterate id="element" indexId="index" name="Animals" offset="1" length="2">
    <bean:write name="index"/>.<bean:write name="element"/><BR>
</logic:iterate><p>
</td>
<td>
<logic:iterate id="header" collection="<%= request.getHeaderNames()%>">
    <bean:write name="header"/><BR>
</logic:iterate>
</td>
</tr>
<table>
</body>
</html:html>
```

14.4.1 遍历集合

<logic:iterate>的 name 属性指定需要进行遍历的集合对象，它每次从集合中检索出一个元素，然后把它存放在 page 范围内，并以 id 属性指定的字符串来命名这个元素，例如：

```
<%
 Vector animals=new Vector();
 animals.addElement("Dog");
 animals.addElement("Cat");
 animals.addElement("Bird");
 animals.addElement("Chick");
 request.setAttribute("Animals", animals);
%>
<logic:iterate id="element" name="Animals">
```

```
    <bean:write name="element"/><BR>
</logic:iterate><p>
```

以上代码先定义了一个 Vector 类型的集合变量 Animals，它存放在 request 范围内。接下来<logic:iterate>标签在一个循环中遍历 Animals 集合中的所有元素，每次检索到一个元素，就把它命名为“element”，并存放在 page 范围内。在<logic:iterate>标签主体中，还嵌套了一个<bean:write>标签，它用于输出每个元素的内容。以上代码的输出内容如下：

```
Dog
Cat
Bird
Chick
```

length 属性指定需要遍历的元素的数目，如果没有设置 length 属性，就遍历集合中的所有元素。offset 属性指定开始遍历的起始位置，默认值为“0”，表示从集合的第一个元素开始遍历。indexId 属性定义一个代表当前被遍历元素序号的变量，这个变量被存放在 page 范围内，可以被标签主体的<bean:write>标签访问。例如：

```
<logic:iterate id="element" indexId="index" name="Animals" offset="1" length="2">
    <bean:write name="index"/>.<bean:write name="element"/><BR>
</logic:iterate><p>
```

以上<logic:iterate>标签的 length 属性为 2，表示只需遍历 Animals 集合中的两个元素。offset 为“1”，表示将从 Animals 集合中的第二个元素开始遍历。在<logic:iterate>标签主体中，嵌套了两个<bean:write>标签，分别输出每个元素的序号和内容。以上代码的输出内容如下：

```
1.Cat
2.Bird
```

14.4.2　遍历 Map

<logic:iterate>标签还可以遍历 HashMap 中的元素，例如：

```
<%
HashMap months = new HashMap();
months.put("Jan.", "January");
months.put("Feb.", "February");
months.put("Mar.", "March");
request.setAttribute("months", months);
%>
<logic:iterate id="element" indexId="ind" name="months">
  <bean:write name="ind"/>.
  <bean:write name="element" property="key"/>:
  <bean:write name="element" property="value"/><BR>
</logic:iterate><P>
```

以上代码先定义了一个名为“months”的 HashMap，存放在 request 范围内。接下来在

<logic:iterate>标签遍历 months 对象的每一个元素，每一个元素都包含一对 key/value。在<logic:iterate>标签主体中包含三个<bean:write>标签，分别输出每个元素的序号、key 和 value。以上代码的输出内容如下：

```
0. Mar.: March
1. Feb.: February
2. Jan.: January
```

如果 HashMap 中的每个元素的 value 是集合对象，则可以采用嵌套的<logic:iterate>标签遍历集合中的所有对象，例如：

```
<%
HashMap h = new HashMap();
String vegetables[] = {"pepper", "cucumber"};
String fruits[] = {"apple","orange","banana","cherry","watermelon"};
String flowers[] = {"chrysanthemum","rose"};
String trees[]={"willow"};
h.put("Vegetables", vegetables);
h.put("Fruits", fruits);
h.put("Flowers", flowers);
h.put("Trees",trees);
request.setAttribute("catalog", h);
%>
<logic:iterate id="element" indexId="ind" name="catalog">
  <bean:write name="ind"/>. <bean:write name="element" property="key"/><BR>
  <logic:iterate id="elementValue" name="element" property="value" length="3" offset="1">
      -----<bean:write name="elementValue"/><BR>
  </logic:iterate>
</logic:iterate><P>
```

以上代码先定义了一个名为“catalog”的 HashMap，存放在 request 范围内。它的每个元素的 value 为字符串数组。接下来外层的<logic:iterate>标签遍历 HashMap 中的所有元素，内层的<logic:iterate>标签访问每个元素的 value 属性，遍历 value 属性引用的字符串数组中的所有元素。以上代码的输出内容如下：

```
0. Vegetables
-----cucumber
1. Flowers
-----rose
2. Fruits
-----orange
-----banana
-----cherry
3. Trees
```

14.4.3　设置被遍历的变量

可以通过以下方式来设置需要遍历的变量。

- 设置 name 属性，name 属性指定需要遍历的集合或 Map，例如：

```
<logic:iterate id="element" name="Animals">
    <bean:write name="element"/><BR>
</logic:iterate><p>
```

- 设置 name 属性和 property 属性，name 属性指定一个 JavaBean，property 属性指定 JavaBean 的一个属性，这个属性为需要遍历的集合或 Map，例如：

```
<logic:iterate id="element" indexId="ind" name="catalog">
  <bean:write name="ind"/>. <bean:write name="element" property="key"/><BR>
  <logic:iterate id="elementValue" name="element" property="value" length="3" offset="1">
      -----<bean:write name="elementValue"/><BR>
  </logic:iterate>
</logic:iterate><P>
```

- 设置 collection 属性，collection 属性指定一个运行时表达式，表达式的运算结果为需要遍历的集合或 Map，例如：

```
<logic:iterate id="header" collection="<%= request.getHeaderNames()%>">
    <bean:write name="header"/><BR>
</logic:iterate>
```

14.5　进行请求转发或重定向的 Logic 标签

本节将介绍用于请求转发或重定向的标签：

- <logic:forward>：进行请求转发。
- <logic:redirect>：进行请求重定向。

14.5.1　<logic:forward>标签

<logic:forward >标签用于请求转发，它的 name 属性指定转发目标，与 Struts 配置文件中的<global-forwards>元素的<forward>子元素匹配。例如，在 Struts 配置文件中配置了如下<forward>元素：

```
<global-forwards>
    <forward    name="index"   path="/index.jsp"/>
    ......
</global-forwards>
```

在样例网页 LogicForward.jsp 中通过<logic:forward>标签把请求转发给 name 属性为“index”的全局转发目标：

```
<logic:forward name="index"/>
```

在通过浏览器访问 http://localhost:8080/logictaglibs/LogicForward.jsp 时，<logic:forward>标签把请求转发给 index.jsp，因此在浏览器端看到的是 index.jsp 的输出网页。

14.5.2 <logic:redirect>标签

<logic:redirect>标签用于请求重定向，它的 forward、href 和 page 属性指定重定向目标，这几个属性的用法和<html:link>标签的 forward、href 和 page 属性的用法很相似，可以参见本书的 12.1.3 小节（<html:link>和<html:rewrite>标签）的内容。

在样例网页 LogicRedirect.jsp 中通过<logic:redirect>标签把请求重定向到“http://www.apache.org”：

```
<logic:redirect href="http://www.apache.org"/>
```

在通过浏览器访问 http://localhost:8080/logictaglibs/LogicRedirect.jsp 时，<logic:redirect>标签把请求重定向到“http://www.apache.org”，因此在浏览器端看到的是 Apache 的主页。

14.6 小 结

本章讨论了 Logic 标签库中的标签的作用和使用方法，大多数 Logic 标签能够根据特定的逻辑条件来控制流程，这些标签包括：

- <logic:equal>
- <logic:greaterEqual>
- <logic:greaterThan>
- <logic:lessEqual>
- <logic:lessThan>
- <logic:notEqual>
- <logic:empty>
- <logic:notEmpty>
- <logic:match>
- <logic:notMatch>
- <logic:present>
- <logic:notPresent>
- <logic:messagesPresent>
- <logic:messagesNotPresent>

以上标签都具有以下形式：

```
<logic:conditionalTag  ……>
 if the result of the comparation is true, perform the tag body
</logic: conditionalTag >
```

这些 Logic 标签判断是否满足特定的条件，如果其判断结果为 true，就执行标签主体的内容。它们的处理类的 doStartTag()方法的代码相同：

```
public int doStartTag() throws JspException {
        if (condition())
            return (EVAL_BODY_INCLUDE);
        else
            return (SKIP_BODY);
}
```

以上 doStartTag()方法调用 condition()方法来判断是否满足特定条件，如果满足，就执行标签主体内容。每个标签处理类的 condition()方法的实现不一样。

除了条件判断标签，在 Logic 标签库中还提供了一个非常实用的标签<logic:iterate>，它能够循环遍历集合或 Map 中的所有元素。此外，在 Logic 标签库中还提供了<logic:forward>和<logic:redirect>，分别用于请求转发和重定向。

Logic 标签库中的多数标签都可以设置 name 属性，指定已经存在的 JavaBean 的名字。property 属性指定这个 JavaBean 的某个属性。scope 属性指定这个 JavaBean 的范围，可选值包括：page、request、session 和 application。如果没有设置 scope 属性，将依次在 page、request、session 和 application 范围内寻找这个 JavaBean。

第 15 章　Struts Nested 标签库

Struts Nested 标签库的一部分标签用于表达 JavaBean 之间的嵌套关系，还有一部分标签能够在特定的嵌套级别提供和其他 Struts 标签库的标签相同的功能。

本章将通过一个 Struts 应用例子：nestedtaglibs 应用，来讲解如何使用 Struts Nested 标签。nestedtaglibs 应用的主页 index.jsp 如图 15-1 所示，它提供了访问其他样例网页的链接，这些样例网页用于演示各种 Nested 标签的使用方法。

图 15-1　nestedtaglibs 应用的主页 index.jsp

本章介绍的 nestedtaglibs 应用的源代码位于配套光盘的 sourcecode/nestedtaglibs 目录下，如果要在 Tomcat 上发布这个应用，只要把整个 nestedtaglibs 目录拷贝到<CATALINA_HOME>/webapps 目录下即可。

15.1　<nested:nest>和<nested:writeNesting>标签

本节将介绍以下标签：

- <nested:nest>：定义一个新的嵌套级别。
- <nested:writeNesting>：输出当前嵌套级别信息。

<nested:nest>标签可以表达 JavaBean 之间的嵌套关系。本章以三个 JavaBean 为例，来解释<nested:nest>标签的用法。这三个 JavaBean 分别是：PersonForm Bean、Person Bean 和 Address Bean。在 PersonForm Bean 中包含一个 Person Bean 类型的属性 person，在 Person Bean 中又包含一个 Address Bean 类型的属性 address。这三个 Bean 之间存在嵌套关系，如图 15-2 所示。

图 15-2　PersonForm Bean、Person Bean 和 Address Bean 之间的嵌套关系

如果采用 UML 术语，这三个 Bean 之间应该是关联关系。本章为了便于讲解 Nested 标签库的用法，把三个 Bean 之间的关系描述为嵌套关系。

从 nestedtaglibs 应用的主页 index.jsp 上选择“Nested”链接，就可以访问本节的样例网页 Nested.jsp。在 Nested.jsp 中定义了一个 PersonForm 表单，它使用<nested:nest>标签来表达以上三个 Bean 之间的嵌套关系。如图 15-3 所示为 Nested.jsp 的网页。

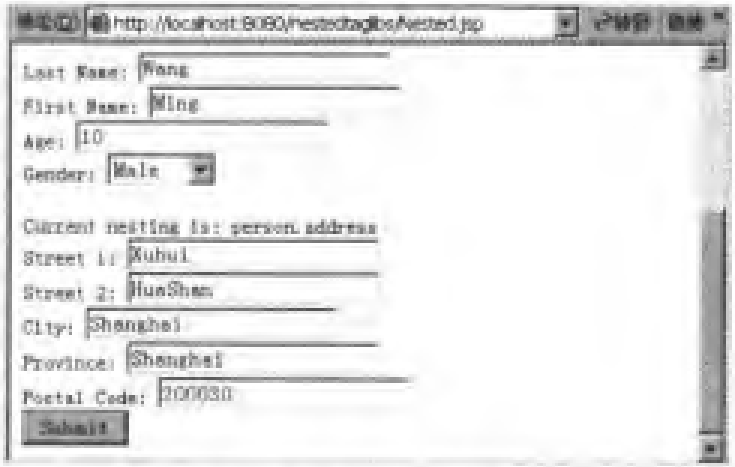

图 15-3 Nested.jsp 网页

例程 15-1 为 Nested.jsp 的源代码。

例程 15-1 Nested.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>
<%@ page import="java.util.Vector" %>
<%@ page import="nestedtaglibs.Person" %>
<%@ page import="nestedtaglibs.Address" %>
<html:html>
    <head>
        <title>Nested sample code</title>
    </head>
    <body bgcolor="white">
        <h3>Nested sample code</h3>
        <p>This page provides examples of the following Struts NESTED tags:<br>
        <ul>
        <li>&lt;nested:nest&gt;</li>
        <li>&lt;nested:select&gt;</li>
        <li>&lt;nested:text&gt;</li>
        <li>&lt;nested:writeNesting&gt;</li>
```

```
</ul>

<%--
The following section shows nest.
--%>
<html:form action="/showPerson">
    <nested:nest property="person">
        Last Name: <nested:text property="lastName"/><BR>
        First Name: <nested:text property="firstName"/><BR>
        Age: <nested:text property="age"/><BR>
        Gender:
        <nested:select property="gender">
        <html:option value="MALE">Male</html:option>
        <html:option value="FEMALE">Female</html:option>
        </nested:select><P>
        <nested:nest property="address">
            Current nesting is: <nested:writeNesting/><BR>
            Street 1: <nested:text property="street1"/><BR>
            Street 2: <nested:text property="street2"/><BR>
            City: <nested:text property="city"/><BR>
            Province: <nested:text property="province"/><BR>
            Postal Code: <nested:text property="postalCode"/><BR>
        </nested:nest>
    </nested:nest>
    <html:submit/>
</html:form>
</body>
</html:html>
```

在 Nested.jsp 中定义了一个表单，与之对应的 ActionForm Bean 名为 PersonForm。Struts 配置文件对 PersonForm Bean 的配置如下：

```
<form-bean name="PersonForm" type="nestedtaglibs.PersonForm"/>
```

PersonForm.java 的源程序参见例程 15-2。

例程 15-2　PersonForm.java

```
package nestedtaglibs;

import org.apache.struts.action.ActionForm;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionMapping;

public class PersonForm extends ActionForm {
  Person person = new Person();
  public void reset (ActionMapping mapping, HttpServletRequest request) {
    person = new Person();
```

```
        person.setAddress(new Address());
      }

      public Person getPerson() {
        return this.person;
      }
      public void setPerson(Person person) {
        this.person = person;
      }
    }
```

在 PersonForm Bean 中包含了一个 Person Bean 类型的 person 属性，Person.java 的源程序参见例程 15-3。

例程 15-3 Person.java

```
package nestedtaglibs;

public class Person {
  private String lastName;
  private String firstName;
  private String age;
  private String gender;
  private Address address;
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setAge(String age) {
    this.age = age;
  }
  public String getAge() {
    return age;
  }
  public void setGender(String gender) {
    this.gender = gender;
  }
  public String getGender() {
    return gender;
```

```
    }
    public void setAddress(Address address) {
      this.address = address;
    }
    public Address getAddress() {
      return address;
    }

}
```

在 Person Bean 中包含了一个 Address Bean 类型的 address 属性，Address.java 的源程序参见例程 15-4。

例程 15-4　Address.java

```
package nestedtaglibs;

public class Address {
  private String street1;
  private String street2;
  private String city;
  private String province;
  private String postalCode;
  public String getStreet1() {
    return street1;
  }
  public void setStreet1(String street1) {
    this.street1 = street1;
  }
  public void setStreet2(String street2) {
    this.street2 = street2;
  }
  public String getStreet2() {
    return street2;
  }
  public void setCity(String city) {
    this.city = city;
  }
  public String getCity() {
    return city;
  }
  public void setProvince(String province) {
    this.province = province;
  }
  public String getProvince() {
    return province;
  }
```

```
    public void setPostalCode(String postalCode) {
      this.postalCode = postalCode;
    }
    public String getPostalCode() {
      return postalCode;
    }
  }
```

在 Nested.jsp 中定义了两个<nested:nest>标签。第一个<nested:nest>标签嵌套在<html:form>表单标签中，代码如下：

```
<html:form action="/showPerson">
  <nested:nest property="person">
    Last Name: <nested:text property="lastName"/><BR>
    ......
  </nested:nest>
  ......
</html:form>
```

以上<nested:nest>标签的上层 JavaBean 为与<html:form>表单标签对应的 PersonForm Bean。<nested:nest>标签的 property 属性为“person”，代表 PersonForm Bean 的 person 属性。这个 person 属性代表 Person Bean，因此嵌套在<nested:nest>标签内部的 Nested 标签都相对于这个 Person Bean。例如，<nested:nest>标签内嵌套的第一个<nested:text>标签的 property 属性为“lastName”，代表 Person Bean 的 lastName 属性。

第二个<nested:nest>标签嵌套在第一个<nested:nest>标签内部，其代码如下：

```
<html:form action="/showPerson">
  <nested:nest property="person">
    ......
    <nested:nest property="address">
      Current nesting is: <nested:writeNesting/><BR>
      Street 1: <nested:text property="street1"/><BR>
      ......
    </nested:nest>
  </nested:nest>
  ......
</html:form>
```

在以上代码中，第二个<nested:nest>标签的 property 属性为“address”，代表 PersonBean 的 address 属性。这个 address 属性代表 Address Bean，因此嵌套在第二个<nested:nest>标签内部的 Nested 标签都相对于这个 Address Bean。

第二个<nested:nest>标签内还嵌套了一个<nested:writeNesting>标签，它显示当前的嵌套级别，输出结果为“person.address”。

如图 15-4 所示为本例中 Nested 标签和相关 JavaBean 的对应关系。

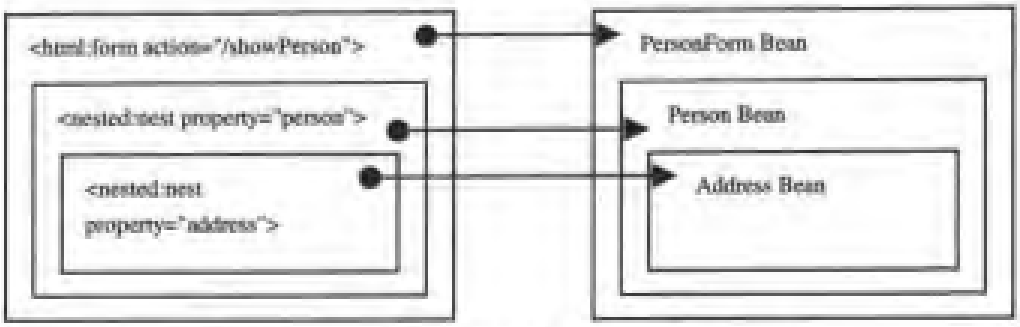

图 15-4　Nested 标签和相关 JavaBean 的对应关系

15.2　<nested:root>标签

在默认情况下，<nested:nest>标签的 property 属性为当前 ActionForm Bean 的某个属性，或者为与上层<nested:nest>标签对应的 JavaBean 的某个属性。可以使用<nested:root>标签来显式指定顶层级别的 JavaBean。<nested:root>标签的 name 属性指定 JavaBean 的名字。嵌套在<nested:root>标签中的<nested:nest>标签的 property 属性为这个 JavaBean 的某个属性。

在 15.1 节中的图 15-3 所示的 Nested.jsp 网页上提交 PersonForm 表单，请求将转发给 org.apache.struts.actions.ForwardAction，ForwardAction 再把请求转发给 ShowPerson.jsp。在 Struts 配置文件中配置了如下 Action 映射：

```
<action     path="/showPerson"
            name="PersonForm"
            type="org.apache.struts.actions.ForwardAction"
            validate="false"
            scope="request"
            parameter="/ShowPerson.jsp">
</action>
```

如图 15-5 所示为 ShowPerson.jsp 的网页。

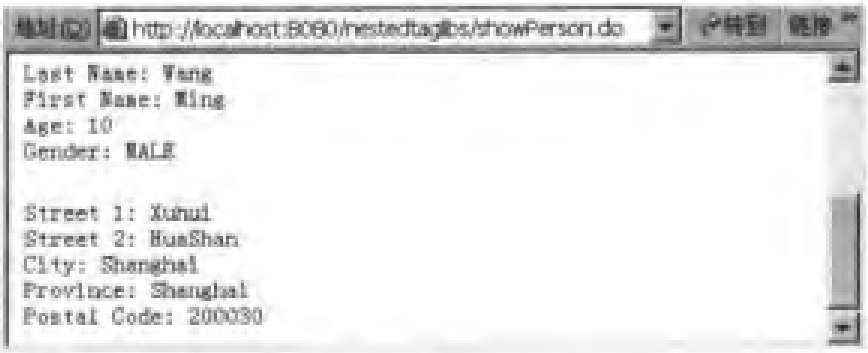

图 15-5　ShowPerson.jsp 的网页

例程 15-5 为 ShowPerson.jsp 的源代码。

例程 15-5 ShowPerson.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>
<%@ page import="java.util.Vector" %>
<%@ page import="nestedtaglibs.Person" %>
<%@ page import="nestedtaglibs.Address" %>

<html:html>
<head>
<title>NESTED sample code</title>
</head>
<body bgcolor="white">
<h3>NESTED sample code</h3>

<p>This page provides examples of the following Struts NESTED tags:<br>
<ul>
<li>&lt;nested:nest&gt;</li>
<li>&lt;nested:root&gt;</li>
<li>&lt;nested:write&gt;</li>
</ul>

<%--
The following section shows nest.
--%>
<jsp:useBean id="PersonForm" type="nestedtaglibs.PersonForm" scope="request"/>
<nested:root name="PersonForm">
    <nested:nest property="person">
        Last Name: <nested:write property="lastName"/><BR>
        First Name: <nested:write property="firstName"/><BR>
        Age: <nested:write property="age"/><BR>
        Gender: <nested:write property="gender"/><P>
        <nested:nest property="address">
            Street 1: <nested:write property="street1"/><BR>
            Street 2: <nested:write property="street2"/><BR>
            City: <nested:write property="city"/><BR>
            Province: <nested:write property="province"/><BR>
            Postal Code: <nested:write property="postalCode"/><BR>
        </nested:nest>
    </nested:nest>
</nested:root>
</body>
</html:html>
```

在以上代码中，<nested:root>标签的 name 属性为"PersonForm"，代表当前的 PersonForm Bean。嵌套其中的<nested:nest>标签的 property 属性为"person"，代表 PersonForm Bean 的 person 属性。

15.3 和其他标签库中的标签功能相同的 Nested 标签

许多 Nested 标签库中的标签具有和其他标签库中的标签相同的功能，区别在于 Nested 标签库中的标签的属性相对于当前的嵌套级别。例如：

```
<nested:nest property="person">
  Last Name: <nested:text property="lastName"/><BR>
  ......
</nested:nest>
```

上面<nested:text>标签和<html:text>标签具有相同的功能，都可以生成文本框。两者的区别在于<nested:text>标签的 property 属性为与当前嵌套级别对应的 JavaBean 的某个属性，而<html:text>标签的 property 属性为与当前表单对应的 ActionForm Bean 的某个属性。

表 15-1 列出了 Nested 标签和其他标签库中的标签的对应关系。

表 15-1 Nested 标签和其他标签库中的标签的对应关系

Nested 标签	其他标签库中的标签
<nested:checkbox>	<html:checkbox>
<nested:errors>	<html:errors>
<nested:file>	<html:file>
<nested:form>	<html:form>
<nested:hidden>	<html:hidden>
<nested:image>	<html:image>
<nested:img>	<html:img>
<nested:link>	<html:link>
<nested:messages>	<html:messages>
<nested:multibox>	<html:multibox>
<nested:options>	<html:options>
<nested:optionsCollection>	<html:optionsCollection>
<nested:password>	<html:password>
<nested:radio>	<html:radio>
<nested:select>	<html:select>
<nested:submit>	<html:submit>
<nested:text>	<html:text>
<nested:textarea>	<html:textarea>
<nested:define>	<bean:define>
<nested:message>	<bean:message>
<nested:size>	<bean:size>

（续表）

Nested 标签	其他标签库中的标签
<nested:write>	<bean:write>
<nested:empty>	<logic:empty>
<nested:equal>	<logic:equal>
<nested:greaterEqual>	<logic:greaterEqual>
<nested:greaterThan>	<logic:greaterThan>
<nested:iterate>	<logic:iterate>
<nested:lessEqual>	<logic:lessEqual>
<nested:lessThan>	<logic:lessThan>
<nested:match>	<logic:match>
<nested:messagesNotPresent>	<logic:messagesNotPresent>
<nested:messagesPresent>	<logic:messagesPresent>
<nested:notEmpty>	<logic:notEmpty>
<nested:notEqual>	<logic:notEqual>
<nested:notMatch>	<logic:notMatch>
<nested:notPresent>	<logic:notPresent>
<nested:present>	<logic:present>

15.4 小　　结

Nested 标签库的一部分标签用于表达 JavaBean 之间的嵌套关系，这些标签包括：

- <nested:nest>：定义一个新的嵌套级别。
- <nested:writeNesting>：输出当前嵌套级别信息。
- <nested:root>：指定顶层级别的 JavaBean。

还有一部分 Nested 标签库的标签能够在特定的嵌套级别提供和其他 Struts 标签相同的功能，如<nested:text>、<nested:define>和<nested:equal>，它们的功能分别与<html:text>、<bean:define>和<logic:equal>相似；区别在于 Nested 标签库中的标签是相对于当前的嵌套级别。

第 16 章　Tiles 框架

在开发 Web 站点时，常常要求同一站点的所有 Web 页面保持一致的外观，比如有相同的布局、页头、页尾和菜单。如图 16-1 所示为一种典型的网页布局。

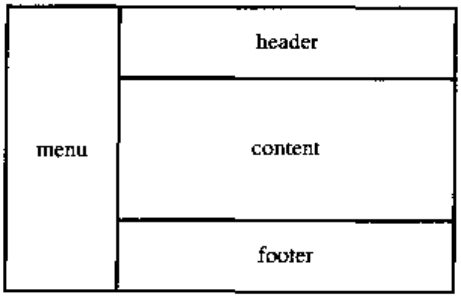

图 16-1　一种典型的网页布局

如图 16-1 所示，网页被划分为四个部分：Header、Menu、Footer 和 Content。对于同一站点的所有 Web 页面，Header、Menu 和 Footer 部分的内容相同，而只有 Content 部分的内容不相同。如果采用基本的 JSP 语句来编写所有的 Web 页面，显然会导致大量的重复编码，增加开发和维护成本。

Tiles 框架为创建 Web 页面提供了一种模板机制，它能将网页的布局和内容分离。它允许先创建模板，然后在运行时动态地将内容插入到模板中。Tiles 框架是建立在 JSP 的 include 指令基础上的，但它提供了比 JSP 的 include 指令更强大的功能。Tiles 框架具有如下特性：

- 创建可重用的模板
- 动态构建和装载页面
- 定义可重用的 Tiles 组件
- 支持国际化

Tiles 框架包含以下内容：

- Tiles 标签库
- Tiles 组件的配置文件
- TilesPlugIn 插件

本章循序渐进地介绍了构建如图 16-1 所示的复合式 Web 页面的若干方案，每个方案都建立在上一个方案的基础之上。本章的样例程序为 tilestaglibs 应用，针对每一种方案，都提供了独立的版本。

16.1　采用基本的 JSP 语句创建复合式网页

创建动态 Web 页面的最基本的办法是为每个页面创建独立的 JSP 文件。如图 16-2 和图 16-3 所示分别为 tilestaglibs 应用的主页 index.jsp 和产品页面 product.jsp。

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version1/tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version1 目录下的整个 tilestaglibs 子目录复制到<CATALINA_HOME>/webapps 目录下即可。

图 16-2 tilestaglibs 应用的主页 index.jsp

图 16-3 tilestaglibs 应用的产品页面 product.jsp

例程 16-1 和例程 16-2 分别为 index.jsp 和 product.jsp 的源代码。

例程 16-1 index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
  <head>
    <title>TilesTaglibs Sample</title>
  </head>
  <body >
    <%-- One table lays out all of the content for this page --%>
    <table width="100%" height="100%" >
      <tr>
        <%-- Sidebar--%>
        <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
          <table>
            <tr>
              <%-- Sidebar top --%>
```

```
                <td width="150" height="65" valign="top" align="left">
                    <a href="">
                        <img src="chinese.gif" border="0" /></a>
                    <a href="">
                        <img src="usa.gif" border="0"/></a>
                </td>
            </tr>
            <tr>
                <%-- Sidebar bottom --%>
                <td>
                    <font size="5">Links</font><p>
                    <a href="index.jsp">Home</a><br>
                    <a href="product.jsp">Products</a><br>
                    <a href="">Hot Link1</a><br>
                    <a href="">Hot Link2</a><br>
                    <a href="">Hot Link3</a><br>

                </td>
            </tr>
        </table>
    </td>
    <%-- Main content--%>
    <td valign="top" height="100%" width="*">
        <table width="100%" height="100%">
            <tr>
                <%-- Header--%>
                <td valign="top" height="15%">
                    <font size="6">Welcome to ABC Inc.</font>
                    <hr>
                </td>
            </tr>
            <tr>
                <%-- Content--%>
                <td valign="top" height="*">
                    <font size="4">Page-specific content goes here</font>
                </td>
            </tr>
            <tr>
                <%-- Footer--%>
                <td valign="bottom" height="15%">
                    <hr>
                    Thanks for stopping by!
                </td>
            </tr>
        </table>
    </td>
```

```
            </tr>
        </table>
    </body>
</html>
```

例程 16-2 product.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
    <head>
        <title>TilesTaglibs Sample</title>
    </head>
    <body >
        <%-- One table lays out all of the content for this page --%>
        <table width="100%" height="100%" >
            <tr>
                <%-- Sidebar--%>
                <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
                    <table>
                        <tr>
                            <%-- Sidebar top --%>
                            <td width="150" height="65" valign="top" align="left">
                                <a href="">
                                    <img src="chinese.gif" border="0" /></a>
                                <a href="">
                                    <img src="usa.gif" border="0"/></a>
                            </td>
                        </tr>
                        <tr>
                            <%-- Sidebar bottom --%>
                            <td>
                                <font size="5">Links</font><p>
                                <a href="index.jsp">Home</a><br>
                                <a href="product.jsp">Products</a><br>
                                <a href="">Hot Link1</a><br>
                                <a href="">Hot Link2</a><br>
                                <a href="">Hot Link3</a><br>
                            </td>
                        </tr>
                    </table>
                </td>
                <%-- Main content--%>
                <td valign="top" height="100%" width="*">
                    <table width="100%" height="100%">
                        <tr>
                            <%-- Header--%>
                            <td valign="top" height="15%">
```

```
                <font size="6">Welcome to ABC Inc.</font>
                <hr>
            </td>
        </tr>
        <tr>
            <%-- Content--%>
            <td valign="top" height="*">
                <font size="4">Products</font> <p>
                <li>product1</li> <br>
                <li>product2</li> <br>
                <li>product3</li> <br>
            </td>
        </tr>
        <tr>
            <%-- Footer--%>
            <td valign="bottom" height="15%">
                <hr>
                Thanks for stopping by!
            </td>
        </tr>
    </table>
</td>
</tr>
</table>
</body>
</html>
```

由例程 16-1 和例程 16-2 可以看出，在 index.jsp 和 product.jsp 文件中，只有粗体字标识的代码块不是重复代码，其余部分均为重复代码。如果网页的相同部分发生需求变更，必须手工修改所有的 JSP 文件。可见，采用基本的 JSP 语句来编写上述网页，会导致 JSP 代码的大量冗余，增加开发与维护成本。

16.2 采用 JSP 的 include 指令创建复合式网页

为了减少代码的冗余，可以把 index.jsp 和 product.jsp 中的相同部分放在单独的 JSP 文件中，然后在 index.jsp 和 product.jsp 文件中通过 JSP include 指令把其他 JSP 文件包含进来。图 16-4 和图 16-5 分别显示了 index.jsp 和 product.jsp 文件包含的其他 JSP 文件。

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version2/ tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version2 目录下的整个 tilestaglibs 子目录复制到<CATALINA_HOME>/webapps 目录下即可。

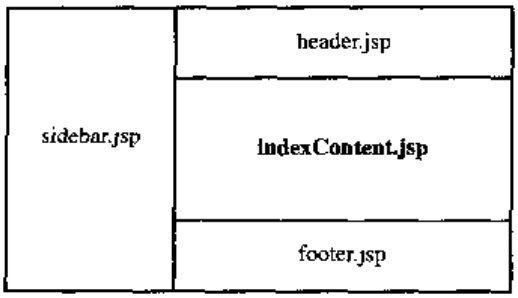

图 16-4 index.jsp 包含的其他 JSP 文件

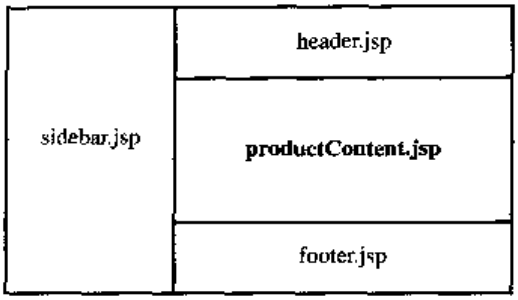

图 16-5 product.jsp 包含的其他 JSP 文件

如图 16-4 和图 16-5 所示，在 index.jsp 和 product.jsp 中均包含 header.jsp、sidebar.jsp 和 footer.jsp，只有网页主体部分包含的 JSP 文件不同。例程 16-3、16-4、16-5、16-6、16-7、16-8 和 16-9 分别为 header.jsp、footer.jsp、sidebar.jsp、indexContent.jsp、productContent.jsp、index.jsp 和 product.jsp 的源代码。

例程 16-3 header.jsp

```
<font size="6">Welcome to ABC Inc.</font>
<hr>
```

例程 16-4 footer.jsp

```
<hr>
Thanks for stopping by!
```

例程 16-5 sidebar.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<table >
    <tr>
        <%-- Sidebar top component --%>
        <td width="150" height="65" valign="top" align="left">
          <a href=""><img src="chinese.gif" border="0"/></a>
          <a href=""><img src="usa.gif" border="0"/></a>
        </td>
    </tr>
    <tr>
        <%-- Sidebar bottom component --%>
        <td>
            <table>
                <tr>
                    <td>
                        <font size="5">Links</font><p>
                        <a href="index.jsp">Home</a><br>
                        <a href="product.jsp">Products</a><br>
                        <a href="">Hot Link1</a><br>
                        <a href="">Hot Link2</a><br>
```

```
                <a href="">Hot Link3</a><br>
              </td>
            </tr>
          </table>
        </td>
      </tr>
    </table>
```

例程 16-6　indexContent.jsp

```
<font size="4">Page-specific content goes here</font>
```

例程 16-7　productContent.jsp

```
<font size="4">Products</font> <p>
<li>product1</li> <br>
<li>product2</li> <br>
<li>product3</li> <br>
```

例程 16-8　index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
  <head>
    <title>TilesTaglibs Sample</title>
  </head>
  <body >
    <%-- One table lays out all of the content for this page --%>
    <table width="100%" height="100%">
      <tr>
        <%-- Sidebar section --%>
        <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
          <jsp:include page="sidebar.jsp"/>
        </td>
        <%-- Main content section --%>
        <td height="100%" width="*">
          <table width="100%" height="100%">
            <tr>
              <%-- Header section --%>
              <td valign="top" height="15%">
                <jsp:include page="header.jsp"/>
              </td>
            <tr>
            <tr>
              <%-- Content section --%>
              <td valign="top" height="*">
                <jsp:include page="indexContent.jsp"/>
```

```
                    </td>
                </tr>
                <tr>
                    <%-- Footer section --%>
                    <td valign="bottom" height="15%">
                        <jsp:include page="footer.jsp"/>
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
</body>
</html>
```

例程 16-9　product.jsp

```
<%@ page contentType="text/html; charset=UTF-8"  %>
<html>
    <head>
        <title>TilesTaglibs Sample</title>
    </head>
    <body >
        <%-- One table lays out all of the content for this page --%>
        <table width="100%" height="100%">
            <tr>
                <%-- Sidebar section --%>
                <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
                    <jsp:include page="sidebar.jsp"/>
                </td>
                <%-- Main content section --%>
                <td height="100%" width="*">
                    <table width="100%" height="100%">
                        <tr>
                            <%-- Header section --%>
                            <td valign="top" height="15%">
                                <jsp:include page="header.jsp"/>
                            </td>
                        </tr>
                        <tr>
                            <%-- Content section --%>
                            <td valign="top" height="*">
                                <jsp:include page="productContent.jsp"/>
                            </td>
                        </tr>
                        <tr>
                            <%-- Footer section --%>
```

```
                <td valign="bottom" height="15%">
                    <jsp:include page="footer.jsp"/>
                </td>
            </tr>
        </table>
    </td>
</tr>
</table>
</body>
</html>
```

采用 JSP include 指令来创建复合式页面，已经在提高代码可重用性方面迈出了正确的一步。在 index.jsp 和 product.jsp 中包含的相同内容，被放在单独的 JSP 页面中。index.jsp 和 product.jsp 只需通过 JSP include 指令把这些相同的内容包含进来，这样提高了代码的可重用性。但是 JSP include 指令不能完全避免代码冗余，例如，从例程 16-8 和例程 16-9 中可以看出，index.jsp 和 product.jsp 中仍然存在许多重复代码，只有粗体字标识的代码块不是重复代码。

此外，和 16.1 节介绍的方案相比，尽管第二种方案减少了重复代码，但 JSP 文件的数量增加了，由原来的两个文件增加到 7 个文件，所以软件的复杂度也增加了。

16.3　采用<tiles:insert>标签创建复合式网页

Tiles 标签库的<tiles:insert>标签和 JSP include 指令具有相同的功能，也能把其他的 JSP 页面插入到当前页面中。例如，以下两条语句的作用是相同的：

```
<jsp:include page="indexContent.jsp"/>
<tiles:insert page="indexContent.jsp" flush="true"/>
```

<tiles:insert>标签的 page 属性指定被插入的 JSP 文件；flush 属性的可选值包括 true 和 false。当 flush 的属性值为 true 时，表示在执行插入操作之前，先调用当前页面的输出流的 flush()方法。

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version3/tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version3 目录下的整个 tilestaglibs 子目录拷贝到<CATALINA_HOME>/webapps 目录下即可。

以下是在 tilestaglibs 应用中使用<tiles:insert>标签的步骤。

（1）安装 Tiles 标签库所需的文件。

在 Struts 的下载软件中包含了运行 Tiles 标签库所需的文件。如果在 Web 应用中使用了 Tiles 标签库，以下文件必须位于 WEB-INF/lib 目录中：

- struts.jar
- commons-digester.jar
- commons-beanutils.jar
- commons-collections.jar
- commons-logging.jar

此外，应该把 Tiles 标签库的定义文件 struts-tiles.tld 拷贝到 WEB-INF 目录下。

（2）在 web.xml 文件中配置如下<taglib>元素：

```
<taglib>
    <taglib-uri>/WEB-INF/struts-tiles.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>
```

（3）创建 index.jsp 和 product.jsp 文件。

修改 16.2 节的例程 16-8（index.jsp）和例程 16-9（product.jsp），在 index.jsp 和 product.jsp 文件的开头，通过<%@ taglib>指令引入 Tiles 标签库，然后把源代码中的 JSP include 指令改为<tiles:insert>标签。例程 16-10 和例程 16-11 分别为修改后的 index.jsp 和 product.jsp 文件。

例程 16-10　index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<html>
    <head>
        <title>TilesTaglibs Sample</title>
    </head>
    <body >
        <%-- One table lays out all of the content for this page --%>
        <table width="100%" height="100%">
            <tr>
                <%-- Sidebar section --%>
                <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
                    <tiles:insert page="sidebar.jsp" flush="true"/>
                </td>
                <%-- Main content section --%>
                <td height="100%" width="*">
                    <table width="100%" height="100%">
                        <tr>
                            <%-- Header section --%>
                            <td valign="top" height="15%">
                                <tiles:insert page="header.jsp" flush="true"/>
                            </td>
                        </tr>
                        <tr>
                            <%-- Content section --%>
```

```
                    <td valign="top" height="*">
                        <tiles:insert page="indexContent.jsp" flush="true"/>
                    </td>
                </tr>
                <tr>
                    <%-- Footer section --%>
                    <td valign="bottom" height="15%">
                        <tiles:insert page="footer.jsp" flush="true"/>
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
</body>
</html>
```

例程 16-11　product.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<html>
    <head>
        <title>TilesTaglibs Sample</title>
    </head>
    <body >
        <%-- One table lays out all of the content for this page --%>
        <table width="100%" height="100%">
            <tr>
                <%-- Sidebar section --%>
                <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
                    <tiles:insert page="sidebar.jsp" flush="true"/>
                </td>
                <%-- Main content section --%>
                <td height="100%" width="*">
                    <table width="100%" height="100%">
                        <tr>
                            <%-- Header section --%>
                            <td valign="top" height="15%">
                                <tiles:insert page="header.jsp" flush="true"/>
                            </td>
                        </tr>
                        <tr>
                            <%-- Content section --%>
                            <td valign="top" height="*">
                                <tiles:insert page="productContent.jsp" flush="true"/>
                            </td>
```

```
                </tr>
                <tr>
                    <%-- Footer section --%>
                    <td valign="bottom" height="15%">
                        <tiles:insert page="footer.jsp" flush="true"/>
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
</body>
</html>
```

从例程 16-10 和例程 16-11 中可以看出，用<tiles:insert>标签取代 JSP include 指令来创建复合式页面，其代码只有细微的差别，两者的利弊也很相似。单纯使用<tiles:insert>标签来创建复合式页面，还没有充分发挥 Tiles 框架的优势。

16.4 采用 Tiles 模板创建复合式网页

在 16.3 节中，尽管使用了<tiles:insert>标签，在 index.jsp 和 product.jsp 文件中还是存在很多的重复代码。为了提高 Web 页面的可重用性和可维护性，可以引入 Tiles 的模板机制。

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version4/tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version4 目录下的整个 tilestaglibs 子目录复制到<CATALINA_HOME>/webapps 目录下即可。

通俗地讲，Tiles 模板是一种描述页面布局的 JSP 页面。Tiles 模板只定义了 Web 页面的样式，而不指定内容。在 Web 应用运行时，才把特定内容插入到模板页面中。同一模板可以被多个 Web 页面共用。

使用模板，可以轻松地实现 Web 应用的所有页面保持相同的外观和布局，而无需为每个页面硬编码。在一个应用中，大多数页面使用同一个模板，某些页面可能需要不同的外观，而使用其他的模板，因此一个应用可能有一个以上的模板。

以下是在 tilestaglibs 应用中使用 Tiles 模板的步骤。

（1）安装 Tiles 标签库所需的文件，同 16.3 节中提到的步骤（1）。

（2）在 web.xml 文件中配置<taglib>元素，同 16.3 节中提到的步骤（2）。

（3）定义模板文件，参见例程 16-12。

例程 16-12　layout.jsp

```
<%@ page contentType="text/html; charset=UTF-8"  %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>
<html>
    <head>
        <title>TilesTaglibs Sample</title>
    </head>
    <body >
          <%-- One table lays out all of the content --%>
        <table width="100%" height="100%">
            <%-- Sidebar section --%>
            <tr>
                <td width="150" valign="top" align="left" bgcolor="#CCFFCC">
                    <tiles:insert attribute="sidebar"/>
                </td>
                <%-- Main content section --%>
                <td valign="top" height="100%" width="*">
                    <table width="100%" height="100%">
                        <tr>
                            <%-- Header section --%>
                            <td height="15%">
                                <tiles:insert attribute="header"/>
                            </td>
                        <tr>
                        <tr>
                            <%-- Content section --%>
                            <td valign="top" height="*">
                                <tiles:insert attribute="content"/>
                            </td>
                        </tr>
                        <tr>
                            <%-- Footer section --%>
                            <td valign="bottom" height="15%">
                                <tiles:insert attribute="footer"/>
                            </td>
                        </tr>
                    </table>
                </td>
            </tr>
        </table>
    </body>
</html>
```

在模板文件 layout.jsp 中定义了网页的布局，但没有指定各部分的具体内容。在 layout.jsp 中包含了多个<tiles:insert>标签，它的 attribute 属性仅仅指定了待插入内容的逻辑

名，而没有指定真正被插入的文件。

（4）在 index.jsp 和 product.jsp 中运用 Tiles 模板，请参见例程 16-13 和例程 16-14。

例程 16-13 index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert page="layout.jsp" flush="true">
        <tiles:put name="sidebar" value="sidebar.jsp"/>
        <tiles:put name="header"  value="header.jsp"/>
        <tiles:put name="content" value="indexContent.jsp"/>
        <tiles:put name="footer"  value="footer.jsp"/>
</tiles:insert>
```

例程 16-14 product.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert page="layout.jsp" flush="true">
        <tiles:put name="sidebar" value="sidebar.jsp"/>
        <tiles:put name="header"  value="header.jsp"/>
        <tiles:put name="content" value="productContent.jsp"/>
        <tiles:put name="footer"  value="footer.jsp"/>
</tiles:insert>
```

在 index.jsp 和 product.jsp 中，<tiles:insert>标签指定插入的模板文件，index.jsp 和 product.jsp 均使用相同的模板文件 layout.jsp。<tiles:insert>标签中包含了若干<tiles:put>子标签，它指定插入到模板中的具体内容。<tiles:put>标签的 name 属性和模板文件中的<tiles:insert>标签的 attribute 属性匹配，<tiles:put>标签的 value 属性指定插入到模板中的具体 JSP 文件。

采用 Tiles 模板机制，大大提高了代码的可重用性和可维护性，模板中包含了网页共同的布局。如果布局发生变化，就只需要修改模板文件，而无需修改具体的网页文件。不过，从例程 16-13 和 16-14 中可以看出，尽管 index.jsp 和 product.jsp 文件的长度都缩短了，但是两者还是存在重复代码。

16.5 采用 Tiles 模板和 Tiles 组件创建复合式网页

为了最大程度地提高代码的可重用性和灵活性，Tiles 框架引入了 Tiles 组件的概念。Tiles 组件可以代表一个完整的网页，也可以代表网页的一部分。简单的 Tiles 组件可以组合成复杂的 Tiles 组件，或被扩展为复杂的 Tiles 组件。

16.5.1　Tiles 组件的基本使用方法

Tiles 框架允许在专门的 XML 文件中配置 Tiles 组件。例如，以下代码定义了一个名为“index-definition”的 Tiles 组件，它描述整个 index.jsp 网页：

```
<tiles-definitions>
  <definition  name="index-definition"  path="/layout.jsp">
    <put name="sidebar" value="sidebar.jsp"/>
    <put name="header" value="header.jsp"/>
    <put name="content" value="indexContent.jsp"/>
    <put name="footer"  value="footer.jsp"/>
  </definition>
</tiles-definitions>
```

<definition>元素的 name 属性指定 Tiles 组件的名字，path 属性指定 Tiles 组件使用的模板，<definition>元素的<put>子元素用于向模板中插入具体的网页内容。

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version5/tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version5 目录下的整个 tilestaglibs 子目录复制到<CATALINA_HOME>/webapps 目录下即可。

以下是在 tilestaglibs 应用中使用 Tiles 组件的步骤。

（1）安装 Tiles 标签库所需的文件，同 16.3 节中提到的步骤（1）。

（2）在 web.xml 文件中配置<taglib>元素，同 16.3 节中提到的步骤（2）。

（3）在专门的 XML 文件中配置 Tiles 组件，在本例中把这个配置文件命名为 tiles-defs.xml，这个文件位于 WEB-INF 目录下。例程 16-15 为 tiles-defs.xml 文件的代码。

例程 16-15　tiles-defs.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
 <!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
       "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">

<tiles-definitions>
  <definition  name="index-definition"  path="/layout.jsp">
    <put name="sidebar" value="sidebar.jsp"/>
    <put name="header"  value="header.jsp"/>
    <put name="content" value="indexContent.jsp"/>
    <put name="footer"  value="footer.jsp"/>
  </definition>
```

```
    <definition  name="product-definition"  path="/layout.jsp">
      <put name="sidebar" value="sidebar.jsp"/>
      <put name="header"   value="header.jsp"/>
      <put name="content" value="productContent.jsp"/>
      <put name="footer"   value="footer.jsp"/>
    </definition>

</tiles-definitions>
```

以上代码定义了两个 Tiles 组件，它们分别代表完整的 index.jsp 和 product.jsp 页面。

（4）在 Strut 配置文件中配置 TilesPlugin 插件，其代码如下：

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
  <set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" />
  <set-property property="definitions-parser-validate" value="true" />
</plug-in>
```

TilesPlugin 插件用于加载 Tiles 组件的配置文件。在<plug-in>元素中包含几个<set-property>子元素，用于向 TilesPlugin 插件传入附加的参数：

- definitions-config 参数：指定 Tiles 组件的配置文件，如果有多个配置文件，则它们之间用逗号分隔。
- definitions-parser-validate 参数：指定 XML 解析器是否验证 Tiles 配置文件，可选值包括 true 和 false，默认值为 true。

（5）在 web.xml 文件中配置 ActionServlet。

为了保证在 Web 应用启动时加载 TilesPlugin 插件，应该加入 ActionServlet 控制器，ActionServlet 控制器在初始化时能加载所有的插件。以下是在 web.xml 文件中配置 ActionServlet 的代码：

```
<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>
    <load-on-startup>3</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

（6）在 index.jsp 和 product.jsp 中插入 Tiles 组件，参见例程 16-16 和例程 16-17：

例程 16-16　index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert definition="index-definition"/>
```

例程 16-17　product.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert definition="product-definition"/>
```

16.5.2　通过 Struts Action 来调用 Tiles 组件

如果 Tiles 组件代表完整的网页，那么可以直接通过 Struts Action 来调用 Tiles 组件。例如，如果希望通过 Struts Action 来调用 16.5.1 小节定义的名为“index-definition”的 Tiles 组件，那么可以在 Struts 配置文件中配置如下 Action 映射：

```
<action-mappings>
    <action    path="/index"
               type="org.apache.struts.actions.ForwardAction"
               parameter="index-definition">
    </action>
</action-mappings>
```

接下来通过浏览器访问 http://localhost:8080/tilestaglibs/index.do，该请求先被转发到 ForwardAction，ForwardAction 再把请求转发给名为“index-definition”的 Tiles 组件，最后在浏览器端，用户将看到和 index.jsp 相同的页面。

通过 Struts Action 来调用 Tiles 组件，可以充分发挥 Struts 框架负责流程控制的功能。此外，还可以减少 JSP 文件的数目。例如，如果直接通过 Struts Action 来调用名为“index-definition”的 Tiles 组件，就不必再创建 index.jsp 文件。

16.5.3　Tiles 组件的组合

Tiles 组件是一种可重用的组件，可以像搭积木一样，把简单的 Tiles 组件组装成复杂的 Tiles 组件。例如，可以把名为“index-definition”的 Tiles 组件的左边部分拆分为独立的 Tiles 组件，名为“sidebar-definition”，如图 16-6 所示。

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version6/tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version6 目录下的整个 tilestaglibs 子目录复制到<CATALINA_HOME>/webapps 目录下即可。

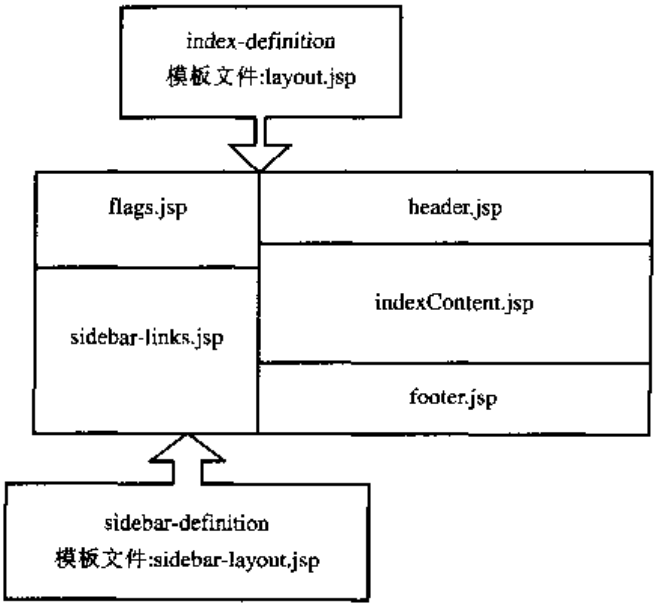

图 16-6 把名为“index-definition”的 Tiles 组件的左边部分拆分为独立的 Tiles 组件

以下是在 tilestaglibs 应用中使用组合式 Tiles 组件的步骤。

步骤

（1）在 tiles-def.xml 文件中重新定义“sidebar-definition”、“index-definition”和“product-definition”这三个 Tiles 组件。在一个 Tiles 组件中包含另一个 Tiles 组件的语法为：

```
<definition   name="index-definition" path="/layout.jsp">
    <put name="sidebar" value="sidebar-definition" type="definition"/>
    ……
</definition>
```

以上<put>子元素的 value 属性指定被包含的 Tiles 组件的名字，type 属性设置为“definition”，表示value属性指定的是Tiles组件，而不是JSP文件。例程16-18是tiles-def.xml文件的代码。

例程 16-18 tiles-def.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
  "-//Apache Software Foundation//DTD Tiles Configuration//EN"
  "http://jakarta.apache.org/struts/dtds/tiles-config.dtd">
<tiles-definitions>
  <definition   name="sidebar-definition"    path="/sidebar-layout.jsp">
    <put name="top"   value="flags.jsp"/>
    <put name="bottom" value="sidebar-links.jsp"/>
  </definition>
```

```
    <definition  name="index-definition"    path="/layout.jsp">
        <put name="sidebar" value="sidebar-definition" type="definition"/>
        <put name="header"   value="header.jsp"/>
        <put name="content" value="indexContent.jsp"/>
        <put name="footer"   value="footer.jsp"/>
    </definition>

    <definition  name="product-definition"   path="/layout.jsp">
        <put name="sidebar" value="sidebar-definition" type="definition"/>
        <put name="header"   value="header.jsp"/>
        <put name="content" value="productContent.jsp"/>
        <put name="footer"   value="footer.jsp"/>
    </definition>

</tiles-definitions>
```

（2）创建名为“sidebar-definition”的 Tiles 组件的相关 JSP 文件。

名为“sidebar-definition”的 Tiles 组件的模板文件为 sidebar-layout.jsp，被插入到这个模板中的两个 JSP 文件分别为 flags.jsp 和 sidebar-links.jsp。例程 16-19、16-20 和 16-21 分别为这几个 JSP 文件的源代码。

例程 16-19　sidebar-layout.jsp

```
<%@ page contentType="text/html; charset=UTF-8"  %>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>
<table >
    <tr>
        <%-- Sidebar top component --%>
        <tiles:insert attribute="top"/>
    </tr>
    <tr>
        <%-- Sidebar bottom component --%>
        <tiles:insert attribute="bottom"/>
    </tr>
</table>
```

例程 16-20　sidebar-links.jsp

```
<%-- Sidebar bottom component --%>
    <td>
        <table>
            <tr>
                <td>
                    <font size="5">Links</font><p>
                    <a href="index.jsp">Home</a><br>
                    <a href="product.jsp">Products</a><br>
                    <a href="">Hot Link1</a><br>
```

```
                <a href="">Hot Link2</a><br>
                <a href="">Hot Link3</a><br>
            </td>
        </tr>
    </table>
</td>
```

例程 16-21 flags.jsp

```
<%-- Sidebar top component --%>
<td width="150" height="65" valign="top" align="left">
    <a href=""><img src="chinese.gif" border="0"/></a>
    <a href=""><img src="usa.gif" border="0"/></a>
</td>
```

16.5.4 Tiles 组件的扩展

在 16.5.3 小节的 tiles-def.xml 文件中，“index-definition”和“product-definition”两个 Tiles 组件的定义中仍然存在重复代码。可以利用 Tiles 组件的可扩展特性来进一步消除冗余代码。解决方法为先定义一个包含这两个 Tiles 组件的共同内容的父类 Tiles 组件，命名为“base-definition”，然后再让“index-definition”和“product-definition”这两个 Tiles 组件继承这个父类组件。如图 16-7 所示为改进后的 Tiles 组件的关系。

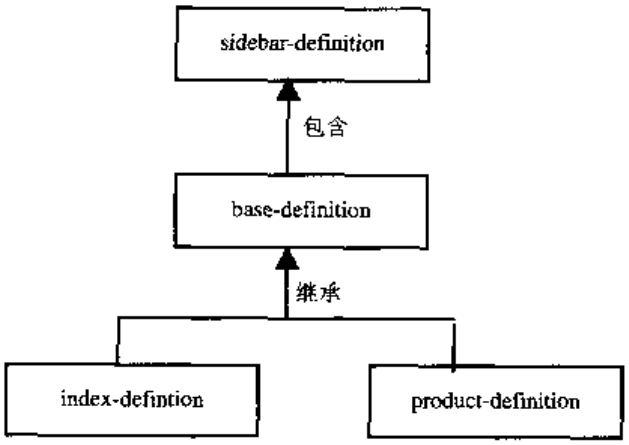

图 16-7 改进后的 Tiles 组件的关系

本节介绍的 tilestaglibs 应用的源程序位于配套光盘的 sourcecode/tilestaglibs/version7/tilestaglibs 目录下。如果要在 Tomcat 上发布这个应用，只要把 version7 目录下的整个 tilestaglibs 子目录复制到<CATALINA_HOME>/webapps 目录下即可。

一个 Tiles 组件继承另一个 Tiles 组件的语法如下，其中<definition>元素的 extends 属性指定被扩展的父类 Tiles 组件：

```
<definition  name="index-definition" extends="base-definition">
```

例程 16-22 为改进后的 tiles-def.xml 的代码。

例程 16-22　tiles-def.xml

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

<!DOCTYPE tiles-definitions PUBLIC
        "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
        "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">
<tiles-definitions>
   <definition   name="sidebar-definition"    path="/sidebar-layout.jsp">
      <put name="top"   value="flags.jsp"/>
      <put name="bottom" value="sidebar-links.jsp"/>
   </definition>
   <definition   name="base-definition"     path="/layout.jsp">
      <put name="sidebar" value="sidebar-definition" type="definition"/>
      <put name="header"   value="header.jsp"/>
      <put name="content" value=""/>
      <put name="footer"   value="footer.jsp"/>
   </definition>
   <definition   name="index-definition"   extends="base-definition">
      <put name="content" value="indexContent.jsp"/>
   </definition>
   <definition   name="product-definition"   extends="base-definition">
      <put name="content" value="productContent.jsp"/>
   </definition>
</tiles-definitions>
```

16.6　小　　结

传统的 GUI 工具包，如 Java AWT 和 Java Swing，都提供了一些功能强大的布局管理器，它们指定各个视图组件在窗口中的分布位置。布局管理器有助于创建复合式的复杂界面，一个复合式界面由一些简单的基本界面组成。利用布局管理器来创建 GUI 界面有以下优点：

- 可重用性：基本界面可以被重用，组合成各种不同的复合式界面。
- 可扩展性：可以方便地扩展基本界面，从而创建更复杂的界面。
- 可维护性：每个基本界面之间相互独立，当复合式界面中的局部区域发生变化时，不会影响到其他区域。

不幸的是，JSP 技术本身并没有直接提供布局或布局管理器。为了简化 Web 页面的开发，提高可重用性和可扩展性，Struts Tiles 框架提供了一种模板机制，模板定义了网页的布局，同一模板可以被多个 Web 页面共用。此外，Tiles 框架还允许定义可重用的 Tiles 组件，它可以描述一个完整的网页，也可以描述网页的局部内容。简单的 Tiles 组件可以被组合或扩展成为更复杂的 Tiles 组件。

本章由浅入深地介绍了创建复合式 Web 页面的几种方案。与采用基本的 JSP 语言来创建 Web 页面相比，Tiles 框架大大提高了视图层程序代码的可重用性、可扩展性和可维护性。不过，使用 Tiles 框架也增加了开发视图的难度和复杂度。如果 Web 应用规模很小，界面非常简单，不妨直接采用基本的 JSP 语言来编写网页。对于大型复杂的 Web 应用，可以充分运用 Tiles 框架的优势，从整体上提高网页开发的效率。

第 17 章　Struts 与 EJB 组件

本章首先简单介绍了 J2EE 的体系结构，然后以 netstore 应用为例，介绍开发 EJB 组件的过程，接着介绍如何在 Struts 应用中通过业务代理接口来访问 EJB 组件，最后介绍如何在 JBoss 和 Tomcat 的整合服务器上部署 J2EE 应用。

17.1　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 17-1 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

如图 17-1 所示中的 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程的应用服务器。

Enterprise Java Bean（简称 EJB）组件是运行在应用服务器端的组件，它包含了企业应用的业务逻辑。在运行环境中，企业应用客户程序通过调用 EJB 组件的方法来执行业务。

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。

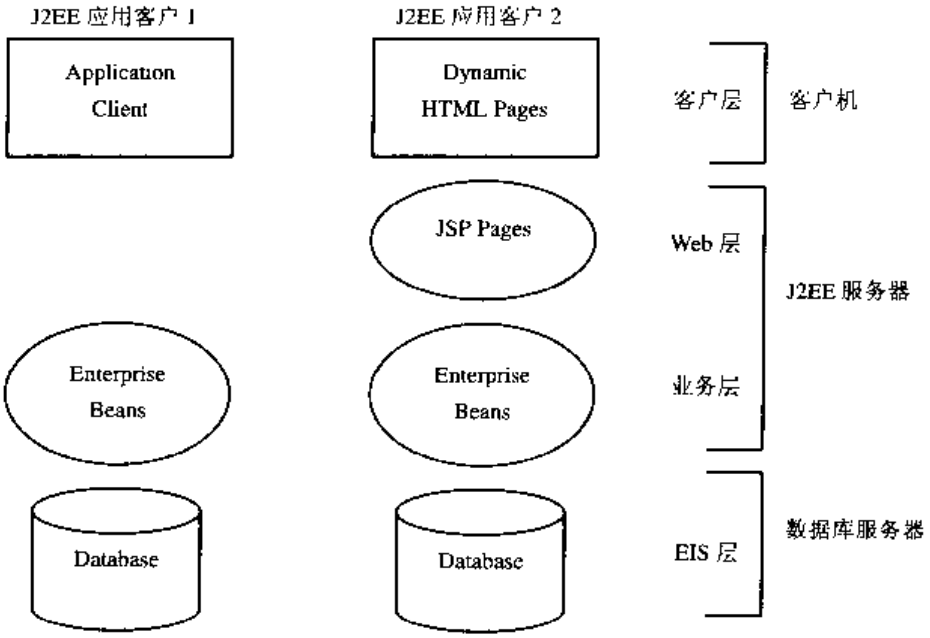

图 17-1 J2EE 多层应用结构

在本章中将创建一个基于 J2EE 的 netstore 应用，它包含一个无状态会话 Bean，名为 NetstoreEJB。新的 netstore 应用的体系结构如图 17-2 所示。

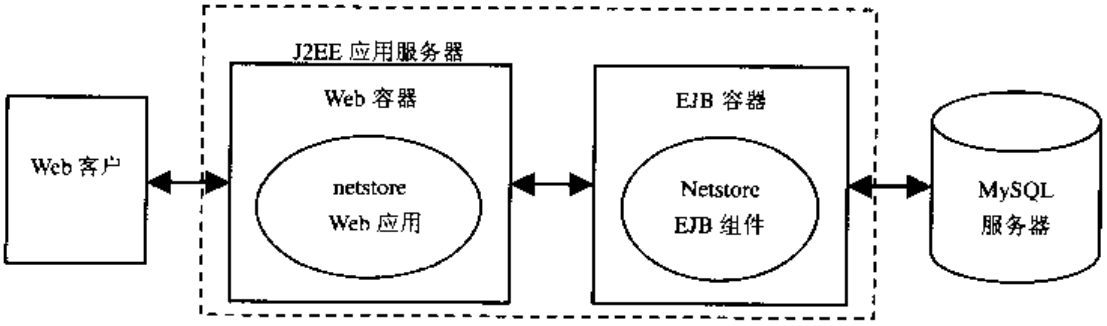

图 17-2 基于 J2EE 的 netstore 应用的体系结构

在本书的第 6 章（Struts 模型组件）介绍的 netstore 应用中，业务逻辑的实现类为 netstore.service.NetstoreServiceImpl，这个类的实例和 Web 应用都运行在 Web 容器中。采用 J2EE 结构后，业务逻辑由 NetstoreEJB 组件来实现，这个 EJB 组件运行在 EJB 容器中。本书选用 JBoss 与 Tomcat 的整合服务器来发布 J2EE 应用，它能够同时充当 Web 容器和 EJB 容器。

17.2 创建 EJB 组件

在范例中，将创建一个无状态的会话 Bean，名为 NetstoreEJB。它通过持久化中间件 OJB 来操纵数据库。

一个 EJB 至少包括 3 个 Java 文件：Remote 接口、Home 接口和 Enterprise Bean 类。本例中 NetstoreEJB 组件的 3 个 Java 文件分别为：

- NetstoreEJB.java：Remote 接口
- NetstoreEJBHome.java：Home 接口
- NetstoreEJBImpl.java：Enterprise Bean 类

17.2.1　编写 Remote 接口

NetstoreEJB 组件的 Remote 接口为 NetstoreEJB.java。在 Remote 接口中声明了客户程序可以调用的业务方法。本例中为了削弱客户程序与模型的关系，先定义了一个 INetstore 接口，它没有引入任何 J2EE API 中的类，接下来让 Remote 接口继承 INetstore 接口。客户程序将通过 INetstore 接口来访问 EJB 组件的业务方法，如图 17-3 所示。

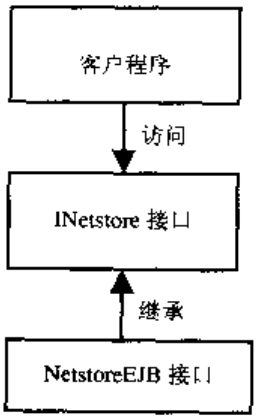

图 17-3　客户程序通过 INetstore 接口来访问 EJB 组件的业务方法

例程 17-1 和 17-2 分别为 INetstore 接口和 NetstoreEJB 接口的源程序。

例程 17-1　INetstore.java

```
package netstore.service.ejb;

import java.rmi.RemoteException;
import java.util.List;
import netstore.catalog.view.ItemDetailView;
import netstore.customer.view.UserView;
import netstore.framework.exceptions.*;

/**
 * The business interface for the Netstore session bean
 */
public interface INetstore {

  public UserView authenticate( String email, String password )
    throws InvalidLoginException, ExpiredPasswordException,
      AccountLockedException, DatastoreException, RemoteException;
```

```
    public List getFeaturedItems() throws DatastoreException, RemoteException;

    public ItemDetailView getItemDetailView( String itemId )
      throws DatastoreException, RemoteException;
  }
```

例程 17-2　Remote 接口 NetstoreEJB.java

```
package netstore.service.ejb;

import javax.ejb.EJBObject;

public interface NetstoreEJB extends EJBObject, INetstore {
  /**
   * The remote interface for the Netstore session bean. All methods are
   * declared in the INetstore business interface.
   */
}
```

17.2.2　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 NetstoreEJBHome 接口包含了一个 create()方法，这个方法返回一个 NetstoreEJB 对象的远程引用。例程 17-3 是 NetstoreEJBHome 的源程序。

例程 17-3　NetstoreEJBHome.java

```
package netstore.service.ejb;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

/**
 * The home interface for the Netstore session bean.
 */
public interface NetstoreEJBHome extends EJBHome {
  public NetstoreEJB create() throws CreateException, RemoteException;
}
```

17.2.3　编写 Enterprise Java Bean 类

本例中的 Enterprise Java Bean 名为 NetstoreEJBImpl，它实现了远程接口 NetstoreEJB 中定义的业务方法。例程 17-4 是 NetstoreEJBImpl 类的源代码。

例程 17-4　NetstoreEJBImpl.java

```
package netstore.service.ejb;

//import all the necessary packages here
.....

/**
 * This is a simple Session Bean implementation of the Netstore service
 */
public class NetstoreEJBImpl implements SessionBean, INetstore {
   private SessionContext ctx;
   private Implementation odmg = null;
   private Database db = null;

   public UserView authenticate( String email, String password )
    throws InvalidLoginException, ExpiredPasswordException,
      AccountLockedException, DatastoreException {

       // Query the database for a user that matches the credentials
       List results = null;
       try{
          OQLQuery query = odmg.newOQLQuery();
          // Set the OQL select statement
          String queryStr = "select customer from " + CustomerBO.class.getName();
          queryStr += " where email = $1 and password = $2";
          query.create(queryStr);

          // Bind the input parameters
          query.bind( email );
          query.bind( password );

          // Retrieve the results
          results = (List)query.execute();
       }catch( Exception ex ){
          ex.printStackTrace();
          throw DatastoreException.datastoreError(ex);
       }

       // If no results were found, must be an invalid login attempt
       if ( results.isEmpty() ){
          throw new InvalidLoginException();
       }

       // Should only be a single customer that matches the parameters
       CustomerBO customer   = (CustomerBO)results.get(0);
```

```
        // Make sure the account is not locked
        String accountStatusCode = customer.getAccountStatus();
        if ( accountStatusCode != null && accountStatusCode.equals( "L" ) ){
          throw new AccountLockedException();
        }

        // Populate the value object from the Customer business object
        UserView userView = new UserView();
        userView.setId( customer.getId().toString() );
        userView.setFirstName( getGBString(customer.getFirstName() ));
        userView.setLastName(getGBString( customer.getLastName() ));
        userView.setEmailAddress( customer.getEmail() );
        userView.setCreditStatus( customer.getCreditStatus() );

        return userView;
    }

    public List getFeaturedItems() throws DatastoreException {
      List results = null;
      try{
        OQLQuery query = odmg.newOQLQuery();
        // Set the OQL select statement
        query.create( "select featuredItems from " + ItemBO.class.getName() );
        results = (List)query.execute();
      }catch( Exception ex ){
        ex.printStackTrace();
        throw DatastoreException.datastoreError(ex);
      }
      List items = new ArrayList();
      Iterator iter = results.iterator();
      while (iter.hasNext()){
        ItemBO itemBO = (ItemBO)iter.next();
        ItemSummaryView newView = new ItemSummaryView();
        newView.setId( itemBO.getId().toString() );
        newView.setName(getGBString( itemBO.getDisplayLabel() ));
        newView.setUnitPrice( itemBO.getBasePrice() );
        newView.setSmallImageURL( itemBO.getSmallImageURL() );
        newView.setDescription(getGBString( itemBO.getDescription() ));
        items.add( newView );
      }
      return items;
    }

    public ItemDetailView getItemDetailView( String itemId )
     throws DatastoreException {
```

```
        List results = null;
        try{
          OQLQuery query = odmg.newOQLQuery();

          // Set the OQL select statement
          String queryStr = "select item from " + ItemBO.class.getName();
          queryStr += " where id = $1";
          query.create(queryStr);
          query.bind(itemId);

          // Execute the query
          results = (List)query.execute();
        }catch( Exception ex ){
          ex.printStackTrace();
          throw DatastoreException.datastoreError(ex);
        }

        //
        if (results.isEmpty() ){
          throw DatastoreException.objectNotFound();
        }

        ItemBO itemBO = (ItemBO)results.get(0);

        // Build a ValueObject for the Item
        ItemDetailView view = new ItemDetailView();
        view.setId( itemBO.getId().toString() );
        view.setDescription(getGBString( itemBO.getDescription()));
        view.setLargeImageURL( itemBO.getLargeImageURL() );
        view.setName(getGBString( itemBO.getDisplayLabel()) );
        view.setProductFeature(getGBString( itemBO.getFeature1() ));
        view.setUnitPrice( itemBO.getBasePrice() );
        view.setTimeCreated( new Timestamp(System.currentTimeMillis() ));
        view.setModelNumber( itemBO.getModelNumber() );
        return view;
    }

    /**
     * Opens the database and prepares it for transactions.
     */
    private void init() throws DatastoreException {
      try{
        // get odmg facade instance
        odmg = OJB.getInstance();
        db = odmg.newDatabase();
        //open database
```

```
            db.open("default", Database.OPEN_READ_WRITE);
        }catch( Exception ex ){
            System.out.println( ex );
            throw DatastoreException.datastoreError(ex);
        }
    }

    public void ejbCreate() throws CreateException {
        try {
            init();
        }catch ( DatastoreException e ) {
            throw new CreateException(e.getMessage());
        }
    }

    public void ejbRemove() {
        try {
            if (db != null) {
                db.close();
            }
        }catch ( ODMGException e ) { }
    }

    public void setSessionContext( SessionContext assignedContext ) {
        ctx = assignedContext;
    }

    public void ejbActivate() {
        // Nothing to do for a stateless bean
    }

    public void ejbPassivate() {
        // Nothing to do for a stateless bean
    }

    private static String getGBString(String s){
        try{
            return new String(s.getBytes("ISO-8859-1"),"GB2312");
        }catch(Exception e){return null;}
    }
}
```

当 EJB 容器创建一个 EJB 实例时，会调用 Enterprise Java Bean 类的 ejbCreate()方法，NetstoreEJBImpl 的 ejbCreate()方法调用自身的 init()方法。init()方法负责获得 OJB 中间件的实例，然后再通过 OJB 中间件创建与 MySQL 数据库的连接。

17.3　在 Struts 应用中访问 EJB 组件

在第 6 章中介绍了如何通过业务代理模式来削弱 Web 应用和模型之间的关系。本章依然使用这种业务代理模式，当模型的实现方式发生变化时，无需修改 Struts 应用的控制层组件，只需为控制层提供一个新的业务代理实现类 NetstoreEJBDelegate 即可，如图 17-4 所示。

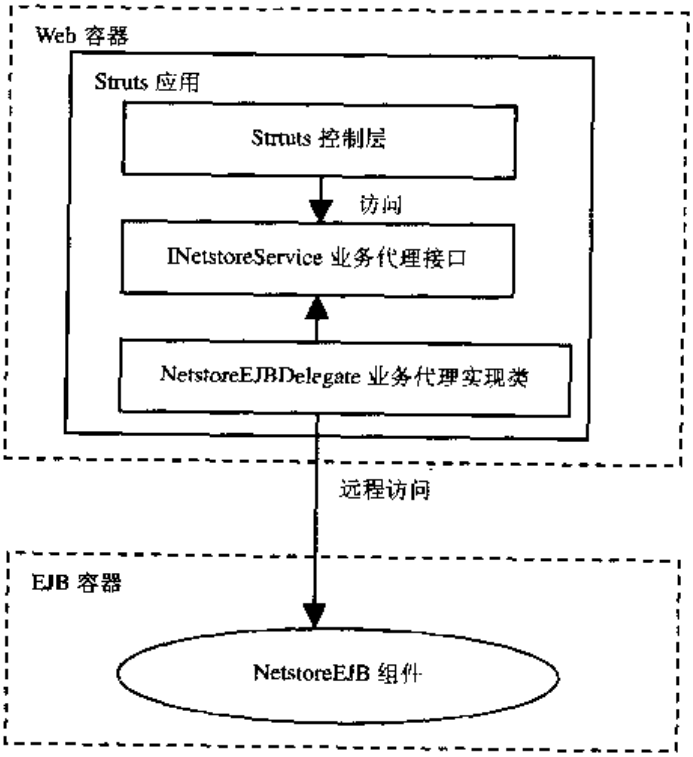

图 17-4　Struts 应用通过业务代理访问 EJB 组件

17.3.1　创建业务代理实现类 NetstoreEJBDelegate

业务代理实现类 NetstoreEJBDelegate 扩展了 INetstoreService 接口。它的构造方法调用自身的 init()方法，init()方法负责获得 NetstoreEJB 组件的远程引用。NetstoreEJB 组件运行在 EJB 容器中，是一种 JNDI 资源。在 Web 应用中，无法创建 NetstoreEJB 组件，而应该首先查找名为“ejb/NetstoreEJB”的 JNDI 资源，获得该资源的引用：

```
InitialContext ic = new InitialContext();
Object objRef = ic.lookup("java:comp/env/ejb/NetstoreEJB");
```

然后把它转换为 NetstoreEJBHome 类型：

```
NetstoreEJBHome home = (NetstoreEJBHome)PortableRemoteObject.narrow(objRef,
        netstore.service.ejb.NetstoreEJBHome.class);
```

接下来调用 NetstoreEJBHome 的 create()方法。此时，EJB 容器会创建 NetstoreEJBImpl 的实例，并调用它的 ejbCreate()方法，然后返回 NetstoreEJB 组件的远程引用。

```
netstore = home.create();
```

在 NetstoreEJBDelegate 的业务方法中，只需调用 NetstoreEJB 组件的相应业务方法即可。例如：

```
public List getFeaturedItems() throws DatastoreException {
    try {
      return netstore.getFeaturedItems();
    }
    catch (RemoteException e) {
      throw DatastoreException.datastoreError(e);
    }
}
```

例程 17-5 为 NetstoreEJBDelegate 的源程序。

例程 17-5　NetstoreEJBDelegate.java

```
package netstore.service.ejb;

//import all the necessary packages here
.....
/**
 * This class is a business delegate that supports the implementation of the
 * INetstoreService interface using the Netstore session bean.
 */
public class NetstoreEJBDelegate implements INetstoreService {

  private INetstore netstore;
  ServletContext servletContext = null;
  public NetstoreEJBDelegate() {
    init();
  }
  private void init() {
    try {
      InitialContext ic = new InitialContext();
      Object objRef = ic.lookup("java:comp/env/ejb/NetstoreEJB");
      NetstoreEJBHome home = (NetstoreEJBHome)PortableRemoteObject.narrow(objRef,
            netstore.service.ejb.NetstoreEJBHome.class);
      netstore = home.create();

    }
    catch (NamingException e) {
      throw new RuntimeException(e.getMessage());
```

```
        }
        catch (CreateException e) {
            throw new RuntimeException(e.getMessage());
        }
        catch (RemoteException e) {
            throw new RuntimeException(e.getMessage());
        }
    }

    public UserView authenticate( String email, String password )
     throws InvalidLoginException, ExpiredPasswordException,
       AccountLockedException, DatastoreException {
        try {
            return netstore.authenticate(email, password);
        }
        catch (RemoteException e) {
            throw DatastoreException.datastoreError(e);
        }
    }

    public List getFeaturedItems() throws DatastoreException {
        try {
            return netstore.getFeaturedItems();
        }
        catch (RemoteException e) {
            throw DatastoreException.datastoreError(e);
        }
    }

    public ItemDetailView getItemDetailView( String itemId )
     throws DatastoreException {
        try {
            return netstore.getItemDetailView(itemId);
        }
        catch (RemoteException e) {
            throw DatastoreException.datastoreError(e);
        }
    }

    public void logout( String email ) {
        // Do nothing for this example
    }

    public void destroy() {
        // Do nothing for this example
```

```
    }
     public void setServletContext( ServletContext ctx ){
       this.servletContext = ctx;
    }

    public ServletContext getServletContext(){
       return servletContext;
    }
}
```

17.3.2 运用 EJBHomeFactory 模式

从 17.3.1 小节中可以看出，如果客户程序访问 EJB 组件的业务方法，那么必须先获得 EJB 组件的 HOME 接口的引用，然后再调用 HOME 接口的 create()方法，获得 EJB 组件的远程引用。通过 JNDI 查找 HOME 接口是一项非常耗时的工作。如果对于每个需要访问 EJB 组件的客户请求，都先执行查找 HOME 接口的步骤，那么显然会降低 Web 应用的运行效率。事实上，HOME 接口是无状态的，不和特定的客户关联，因此同一个 HOME 接口引用可以被多个客户请求或客户线层共享。

为了提高访问 EJB 组件的效率，可以把已经获得的 HOME 接口引用保存在 Web 应用的缓存中，避免以后重复查找相同的 HOME 接口。保存 HOME 接口引用有两种方式：一种方式是把 HOME 接口引用存放在 ServletContext 中；还有一种方式是运用 EJBHomeFactory 模式，把 HOME 接口引用存放在专门的 EJB HOME 工厂类中。第二种方式不依赖于 ServletContext 类，对于不是基于 Web 的应用程序也同样适用。

例程 17-6 给出了 EJBHomeFactory 的一种实现方式。

例程 17-6　EJBHomeFactory.java

```
package netstore.service.ejb;

import java.io.InputStream;
import java.io.IOException;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

/**
 * This class implements the EJBHomeFactory pattern. It performs JNDI
 * lookups to locate EJB homes and caches the results for subsequent calls.
 */
public class EJBHomeFactory {
  private Map homes;
  private static EJBHomeFactory singleton;
```

```
    private InitialContext ctx;

     private EJBHomeFactory() throws NamingException {
       homes = Collections.synchronizedMap(new HashMap());
       ctx = new InitialContext();
     }

    /**
     * Get the Singleton instance of the class.
     */
    public static EJBHomeFactory getInstance() throws NamingException {
       if (singleton == null) {
          singleton = new EJBHomeFactory();
       }
       return singleton;
    }

    /**
     * Specify the JNDI name and class for the desired home interface.
     */
    public EJBHome lookupHome(String jndiName, Class homeClass)
     throws NamingException {
      EJBHome home = (EJBHome)homes.get(homeClass);
      if (home == null) {
         home = (EJBHome)PortableRemoteObject.narrow(ctx.lookup(
           jndiName), homeClass);
         // Cache the home for repeated use
         homes.put(homeClass, home);
      }
      return home;
    }
}
```

EJBHomeFactory 类包含一个 Map 类型的 homes 成员变量，它充当 HOME 接口引用的缓存。EJBHomeFactory 类的 lookupHome()方法先查看是否在 homes 缓存中存放了所要查找的 HOME 接口引用，如果存在，就直接返回这个接口引用；如果不存在，就通过 JNDI 查找 HOME 接口，把找到的接口引用保存在 homes 缓存中，并返回这个接口引用。

运用 EJBHomeFactory 模式后，netstore 应用的业务代理实现类只需从 EJB HOME 工厂中获得 HOME 接口引用。例程 17-7 为业务代理实现类 NetstoreEJBFromFactoryDelegate 的源程序。

例程 17-7　NetstoreEJBFromFactoryDelegate.java

```
package netstore.service.ejb;

//import all the necessary packages here
```

```
.....
/**
 * This class is a business delegate that supports the implementation of the
 * INetstoreService interface using the Netstore session bean.
 */
public class NetstoreEJBFromFactoryDelegate implements INetstoreService {

  private INetstore netstore;
  ServletContext servletContext = null;
  public NetstoreEJBFromFactoryDelegate() {
    init();
  }

 private void init() {
  try {
    NetstoreEJBHome home = (NetstoreEJBHome)EJBHomeFactory.getInstance().
      lookupHome("java:comp/env/ejb/NetstoreEJB",
      NetstoreEJBHome.class);
    netstore = home.create();
  }
  catch (NamingException e) {
    throw new RuntimeException(e.getMessage());
  }
  catch (CreateException e) {
    throw new RuntimeException(e.getMessage());
  }
  catch (RemoteException e) {
    throw new RuntimeException(e.getMessage());
  }
}

  public UserView authenticate( String email, String password )
   throws InvalidLoginException, ExpiredPasswordException,
     AccountLockedException, DatastoreException {
    try {
      return netstore.authenticate(email, password);
    }
    catch (RemoteException e) {
      throw DatastoreException.datastoreError(e);
    }
  }

  public List getFeaturedItems() throws DatastoreException {
    try {
      return netstore.getFeaturedItems();
    }
```

```
        catch (RemoteException e) {
            throw DatastoreException.datastoreError(e);
        }
    }

    public ItemDetailView getItemDetailView( String itemId )
     throws DatastoreException {
        try {
            return netstore.getItemDetailView(itemId);
        }
        catch (RemoteException e) {
            throw DatastoreException.datastoreError(e);
        }
    }

    public void logout( String email ) {
        // Do nothing for this example
    }

    public void destroy() {
        // Do nothing for this example
    }
     public void setServletContext( ServletContext ctx ){
        this.servletContext = ctx;
    }

    public ServletContext getServletContext(){
        return servletContext;
    }
}
```

17.4　发布 J2EE 应用

在发布 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 J2EE 应用，在发布时，应该把它打包为 EAR 文件。本节介绍如何在 JBoss 与 Tomcat 的整合服务器上发布 netstore J2EE 应用。在 Jboss-Tomcat 服务器中，发布 J2EE 组件的目录为<JBOSS_HOME>/server/default/deploy。

17.4.1　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 17-5 所示。

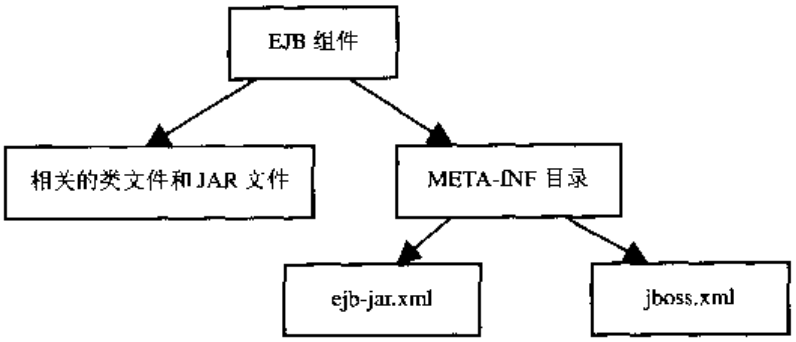

图 17-5 EJB 组件的文件目录结构

1．创建 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 NetstoreEJB 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>Netstore Application</description>
    <display-name>Netstore EJB</display-name>
    <enterprise-beans>
      <session>
        <ejb-name>NetstoreEJB</ejb-name>
        <home>netstore.service.ejb.NetstoreEJBHome</home>
        <remote>netstore.service.ejb.NetstoreEJB</remote>
        <ejb-class>netstore.service.ejb.NetstoreEJBImpl</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
      </session>
    </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>元素指定 EJB 组件的名字，<home>元素指定 Home 接口对应的类名，<remote>元素指定 Remote 接口对应的类名，<ejb-class>元素指定 Enterprise Bean 类对应的类名。

2．创建 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 NetstoreEJB 的 jboss.xml 源文件：

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

```
    <session>
      <ejb-name>NetstoreEJB</ejb-name>
      <jndi-name>ejb/NetstoreEJB</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

以上代码为 NetstoreEJB 组件指定了 JNDI 名字：ejb/NetstoreEJB。

3. 给 EJB 组件打包

在发布 EJB 组件时，应该把它打包为 JAR 文件。假定 NetstoreEJB 组件的源文件都位于<netstoreejb>目录下。在 DOS 窗口中，转到<netstoreejb>目录，运行如下命令：

```
jar  cvf  netstoreejb.jar  *.*
```

在<netstoreejb>目录下将生成 netstoreejb.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 17.4.3 小节，将把这个 EJB 组件加入到 netstore J2EE 应用中，然后再发布整个 J2EE 应用。

17.4.2　在 Jboss-Tomcat 上部署 Web 应用

如果要在 Jboss-Tomcat 上发布 Web 应用，应该在 WEB-INF 目录下增加一个 jboss-web.xml 文件，此外还应该对原来的 web.xml 文件做适当修改。

1. 修改 web.xml 文件

在 netstore Web 应用中访问了 NetstoreEJB 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，其代码如下：

```
<!-- ### EJB References (java:comp/env/ejb) -->
<ejb-ref>
    <ejb-ref-name>ejb/NetstoreEJB</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>netstore.service.ejb.NetstoreEJBHome</home>
    <remote> netstore.service.ejb.NetstoreEJB</remote>
</ejb-ref>
```

在以上代码中声明了对 NetstoreEJB 的引用，<ejb-ref-type>元素声明所引用的 EJB 的类型，<home>元素声明 EJB 的 Home 接口，<remote>元素声明 EJB 的 Remote 接口。

此外，还应该在 web.xml 文件中配置业务代理实现类，ActionServlet 类的初始化参数“netstore-service-class”用来指定业务代理实现类：

```
<servlet>
    <servlet-name>netstore</servlet-name>
    <servlet-class>netstore.framework.ExtendedActionServlet</servlet-class>
    ……
    <init-param>
        <param-name>netstore-service-class</param-name>
```

```
            <param-value>netstore.service.NetstoreEJBDelegate</param-value>
        </init-param>
        ......
    </servlet>
```

除了把业务代理实现类指定为“netstore.service.NetstoreEJBDelegate”外，也可以指定为“netstore.service.NetstoreEJBFromFactoryDelegate”，这个类在本章的 17.3.2 小节中做了介绍。

2．创建 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，在这个文件中指定<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/NetstoreEJB</ejb-ref-name>
        <jndi-name>ejb/NetstoreEJB</jndi-name>
    </ejb-ref>
</jboss-web>
```

3．给 Web 应用打包

在发布 Web 应用时，应该把它打包为 WAR 文件。假定 netstore 应用的所有源文件位于<netstore>目录下。在 DOS 窗口中，转到<netstore>目录，运行如下命令：

```
jar  cvf  netstore.war  *.*
```

在<netstore>目录下将生成 netstore.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。在下一节中，将把这个 Web 应用加入到 netstore J2EE 应用中，然后再发布整个 J2EE 应用。

17.4.3 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 17-6 所示。

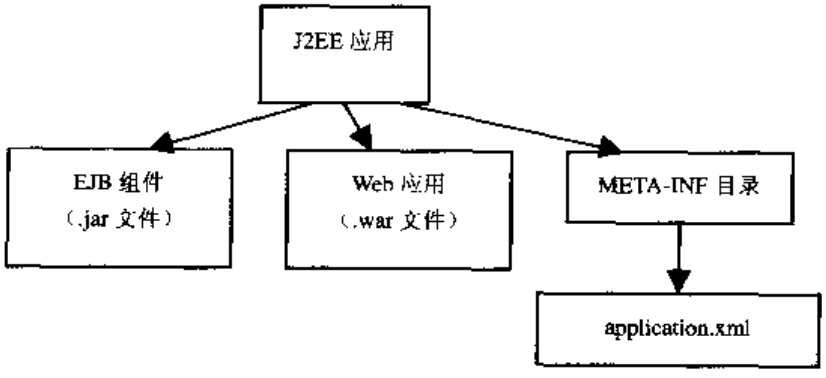

图 17-6　J2EE 应用的目录结构

假定 netstore J2EE 应用的文件位于<netstoreear>目录下，它的目录结构如图 17-7 所示。

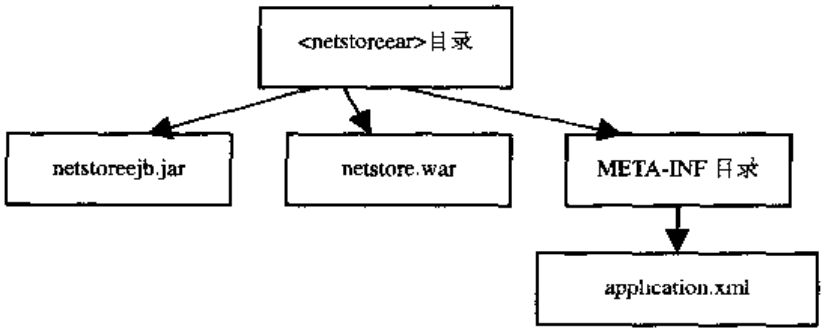

图 17-7　netstore J2EE 应用的目录结构

1．**创建 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>Netstore J2EE Application</display-name>

    <module>
    <web>
        <web-uri>netstore.war</web-uri>
        <context-root>/netstore</context-root>
    </web>
    </module>

    <module>
        <ejb>netstoreejb.jar</ejb>
    </module>

</application>
```

以上代码指明在 netstore J2EE 应用中包含一个 Web 应用，WAR 文件为 netstore.war，URL 路径为“/netstore”；此外还包含一个 EJB 组件，这个组件的 JAR 文件为 netstoreejb.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<netstoreear>目录，运行如下命令：

```
jar  cvf  netstore.ear  *.*
```

在<netstoreear>目录下将生成 netstore.ear 文件。

3．**发布并运行 netstore J2EE 应用**

在 JBoss 与 Tomcat 的整合服务器上发布并运行 netstore J2EE 应用的步骤如下。

步骤

（1）将 netstore.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/netstore/，将会进入 netstore 应用的主页。

17.5 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，EJB 组件必须运行在 EJB 容器中。本章采用 EJB 组件实现了 netstore 应用的模型，由于合理运用了业务代理模式，因此当模型的实现方式发生变化时，对 Struts 应用的控制层组件没有任何影响。

在 JBoss 与 Tomcat 的整合服务器上发布本章样例的详细步骤请参见附录 D.4.3（在工作模式 3 下发布 netstore 应用）和 D.4.4（在工作模式 4 下发布 netstore 应用）。

第 18 章　Struts 与 SOAP Web 服务

SOAP 在 Web 服务中作为基于 XML 信息交换的一种非常普遍的协议，对于实现基于 Web 的无缝集成系统发挥着非常重要的作用。SOAP 有助于实现松散耦合的、跨平台的、与语言无关的、与特定接口无关的分布式系统。

本章首先简单介绍了 SOAP 的概念，然后介绍如何通过 SOAP 服务来实现 netstore 应用的模型，以及如何在 struts 应用中访问 SOAP 服务。

18.1　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 作为数据传送的方式。

SOAP 采用的通信协议可以是 HTTP/HTTPS 协议（现在应用得最广泛），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。如图 18-1 所示为两个系统之间通过 SOAP 通信的过程。

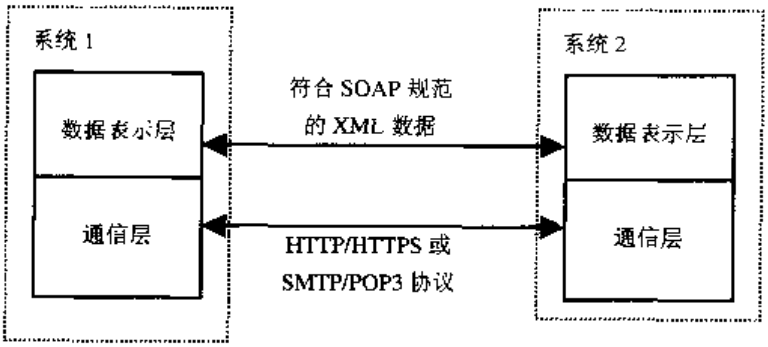

图 18-1　系统间采用 SOAP 通信

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。

如图 18-2 所示为 SOAP RPC 的工作流程，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，它是一个无状态协议，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。

如图 18-2 所示，SOAP 客户访问 SOAP 服务的流程如下。

流程

（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 XML 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，则先对其进行 XML 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 XML 序列化操作，然后把返回值以 XML 文档返回给客户。

（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 XML 反序列化操作，最后获得返回值。

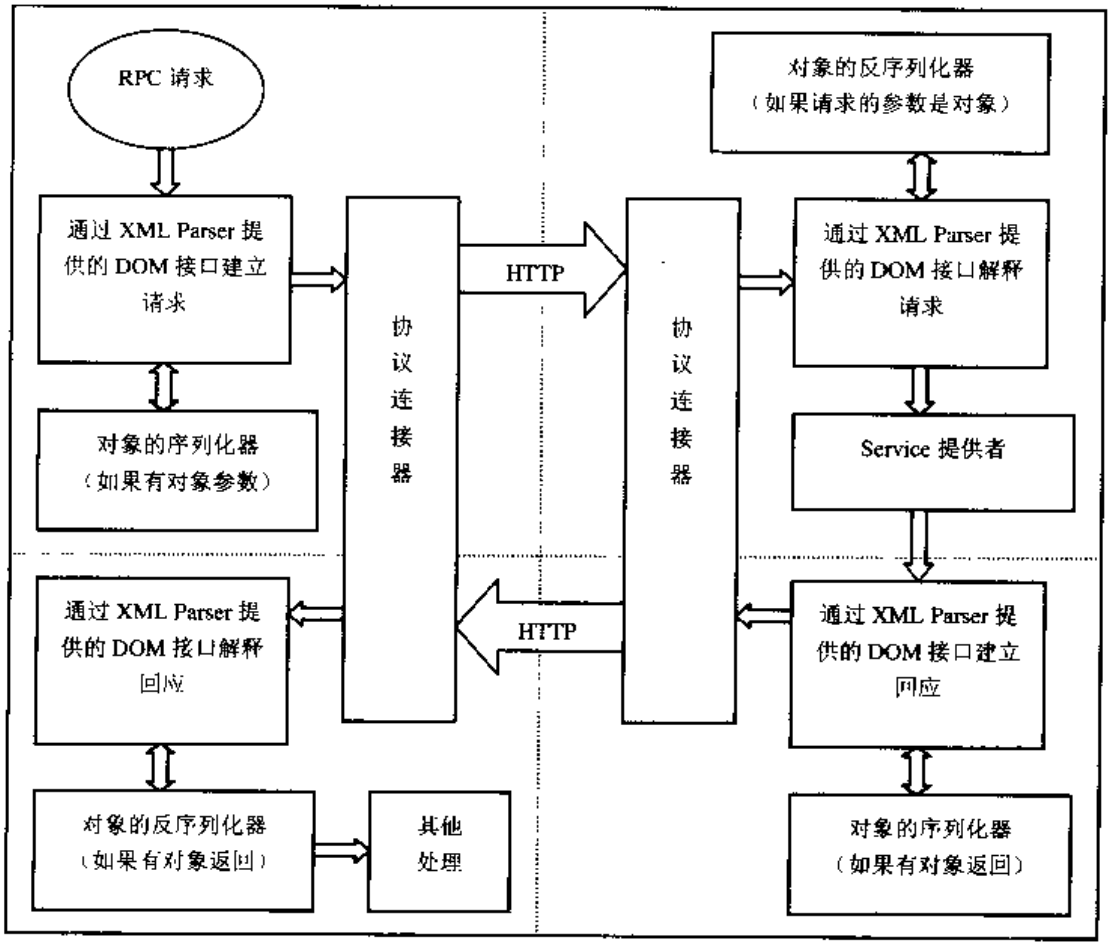

图 18-2 SOAP RPC 的工作流程

图 18-2 中的 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。

18.2　建立 Apache AXIS 环境

Apache AXIS 是 Apache 软件组织对 SOAP 规范的实现，是 Apache SOAP 项目的第三代产品。Apache AXIS 的下载地址为：http://xml.apache.org/axis/index.html，在本书配套光盘的 software 目录下也提供了这个软件。

在 Apache AXIS 软件中提供了一个 Apache-AXIS Web 应用，可以把它发布到 Tomcat 服务器中。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 18-3 所示。

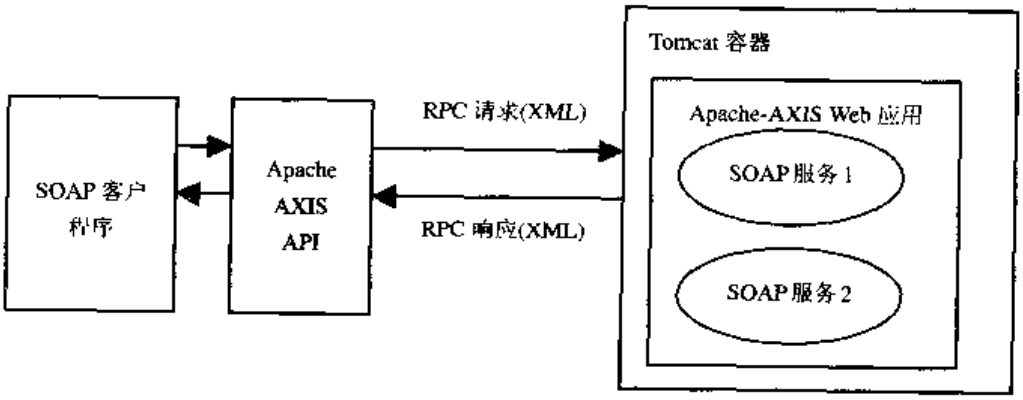

图 18-3　SOAP 客户和 SOAP 服务

在本书配套光盘的 sourcecode/netstore/axis 目录下也提供了这个 Apache-AXIS Web 应用，只要把整个 axis 子目录复制到<CATALINA_HOME>\webapps 目录下，就可以发布这个 Web 应用。

启动 Tomcat 服务器，访问 http://localhost:8080/axis，将会看到如图 18-4 所示的 axis 应用的主页。

图 18-4 Apche-AXIS 主页

18.3 创建和发布 SOAP 服务

本节介绍如何通过 SOAP 服务来实现 netstore 应用的模型。在 Apache AXIS 上创建和发布基于 RPC 的 SOAP 服务包括以下步骤：

（1）创建实现 SOAP 服务的 Java 类。
（2）创建 SOAP 服务的发布描述文件。
（3）通过 AXIS 的 AdminClient 客户程序发布 SOAP 服务。

18.3.1 创建实现 SOAP 服务的 Java 类

在 netstore 应用中，SOAP 服务负责通过持久化中间件 OJB 来访问 MySQL 数据库，实

现各种业务逻辑。可以把 netstore.service. NetstoreServiceImpl 类作为实现 SOAP 服务的 Java 类，它提供了各种业务方法，这个类的源程序参见本书的 6.5.5 小节（联合使用业务代理和 DAO 模式）的例程 6-10。

18.3.2　创建 Web 服务发布描述文件

Apache AXIS 使用 Web 服务发布描述文件 WSDD（Web Service Deployment Descriptor）来发布 SOAP 服务。例程 18-1 是一个 SOAP 服务发布描述文件，文件名为 deploy.wsdd。这个文件定义了一个名为“urn:netstoreservice”的 SOAP 服务。

例程 18-1　SOAP 服务发布描述文件 deploy.wsdd

```
<deployment name="netstore" xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <service name="urn:netstoreservice" provider="java:RPC">
      <parameter name="className" value="netstore.service.NetstoreServiceImpl" />
      <parameter name="allowedMethods"
      value="authenticate,getFeaturedItems,getItemDetailView" />
    </service>

    <beanMapping qname="myNS:ItemDetailView" xmlns:myNS="urn:BeanService"
     languageSpecificType="java:netstore.catalog.view.ItemDetailView"/>

    <beanMapping  qname="myNS:List"  xmlns:myNS="urn:BeanService"
    languageSpecificType="java:java.util.List"/>

    <beanMapping  qname="myNS:ItemSummaryView"  xmlns:myNS="urn:BeanService"
    languageSpecificType="java:netstore.catalog.view.ItemSummaryView"/>

    <beanMapping  qname="myNS:Date"  xmlns:myNS="urn:BeanService"
    languageSpecificType="java:java.util.Date"/>

    <beanMapping  qname="myNS:UserView"  xmlns:myNS="urn:BeanService"
    languageSpecificType="java:netstore.customer.view.UserView"/>

</deployment>
```

以上文件配置了<deployment>、<service>、<parameter>和<beanMapping>元素。下面分别讲述这几种元素：

- <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。

- <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惟一

标识符，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。

- <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，多个方法之间以逗号隔开。

- <beanMapping>元素

<beanMapping>元素为在网络上传输的 JavaBean 对象指定专门的 XML 序列化器和反序列化器。AXIS 提供的 XML 序列化器为 org.apache.axis.encoding.ser.BeanSerializerFactory，XML 反序列化器为 org.apache.axis.encoding.ser.BeanDeserializerFactory。在本章的 18.3.2 小节的第 1 点将详细介绍 Java 数据与 XML 数据的映射。

如果要删除已经发布的 SOAP 服务，可以使用<undeployment>元素。例如，如果删除 netstoreservice 服务，可以创建如下的 undeploy.wsdd 文件：

```
<undeployment name="netstore" xmlns="http://xml.apache.org/axis/wsdd/">
  <service name="urn:netstoreservice"/>
</undeployment>
```

1. Java 数据与 XML 数据的映射

SOAP 客户和 SOAP 服务之间采用符合 SOAP 规范的 XML 数据进行通信。当 SOAP 客户或 SOAP 服务发送 Java 数据时，XML 序列化器会把 Java 数据映射为 XML 数据；当 SOAP 客户或 SOAP 服务接收到 XML 数据时，XML 反序列化器会把 XML 数据映射为 Java 数据。图 18-5 显示了 Java 数据与 XML 数据的映射过程。

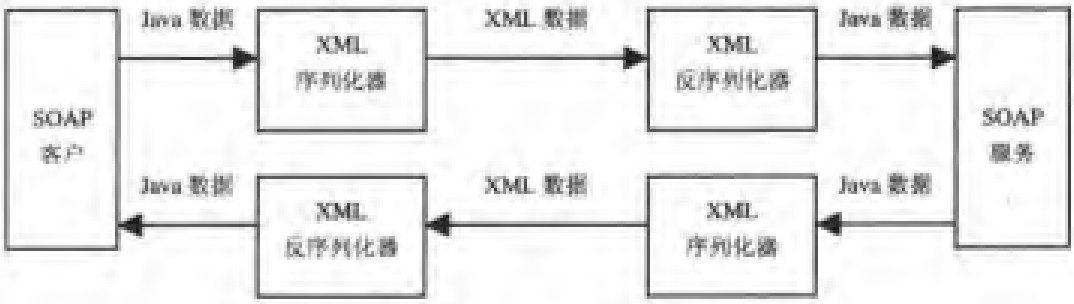

图 18-5 Java 数据与 XML 数据的映射

Apache Axis 为一些常用 Java 数据类型提供了标准的 XML 映射，参见表 18-1。

表 18-1 常用 Java 数据类型的标准 XML 映射

Java 类型	XML 映射
byte[]	xsd:base64Binary
boolean	xsd:boolean
byte	xsd:byte
java.util.Calendar	xsd:dateTime
java.math.BigDecimal	xsd:decimal

（续表）

Java 类型	XML 映射
double	xsd:double
float	xsd:float
byte[]	xsd:hexBinary
int	xsd:int
java.math.BigInteger	xsd:integer
long	xsd:long
javax.xml.namespace.QName	xsd:QName
short	xsd:short
java.lang.String	xsd:string

对于用户自定义的 Java 类，如果这个类的实例将在网络上传输，那么必须创建专门的 XML 序列化器和反序列化器，并在 Web 服务发布描述文件中通过<typeMapping>元素指定映射关系，例如：

```
<typeMapping  qname="ns:local"  xmlns:ns="someNamespace"
      languageSpecificType="java:my.java.MyClass"
      serializer="my.java.Serializer"
      deserializer="my.java.DeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

以上代码为用户自定义的 Java 类 my.java.MyClass 指定 XML 序列化器为 my.java.Serializer 类，XML 反序列化器为 my.java.DeserializerFactory 类。

对于符合 JavaBean 规范的 Java 类，AXIS 提供了现成的序列化器：

- XML 序列化器：org.apache.axis.encoding.ser.BeanSerializerFactory
- XML 反序列化器：org.apache.axis.encoding.ser.BeanDeserializerFactory

由于 netstore.catalog.view.ItemDetailView 类符合 JavaBean 规范，因此可以直接使用 AXIS 提供的序列化器：

```
<typeMapping  qname=" myNS:ItemDetailView"  xmlns:ns=" urn:BeanService "
     languageSpecificType=" java:netstore.catalog.view.ItemDetailView "
     serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
     deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

为了简化起见，对于 JavaBean 类，也可以用<beanMapping>元素来指定使用以上默认的序列化器，以上<typeMapping>代码和以下<beanMapping>代码是等价的：

```
<beanMapping  qname="myNS:ItemDetailView"  xmlns:myNS="urn:BeanService"
languageSpecificType="java:netstore.catalog.view.ItemDetailView"/>
```

netstoreservice 服务为 SOAP 客户提供了三个服务方法，这些方法的参数以及返回值都将通过网络传输，因此，必须为它们指定 XML 序列器。authenticate()方法的返回类型为 UserView 类；getFeaturedItems()方法的返回类型为 List 类，这个 List 集合用来存放 ItemSummaryView 对象；getItemDetailView()方法的返回类型为 ItemDetailView。此外，

UserView、ItemSummaryView、ItemDetailView 都是 BaseView 的子类，在 BaseView 中还包含一个 Date 类型的成员变量。所以必须为以下类设置 XML 映射：

- netstore.customer.view.UserView
- netstore.catalog.view.ItemSummaryView
- netstore.catalog.view.ItemDetailView
- java.util.Date
- java.util.List

由于以上类都符合 JavaBean 规范，所以可以通过<beanMapping>元素为它们指定使用 AXIS 提供的现成的 XML 序列化器。

18.3.3 发布 SOAP 服务

在 Apache AXIS 上发布 SOAP 服务，首先应该把实现 SOAP 服务的类复制到 axis 应用的 WEB-INF/classes 目录下。本例中，应该把 netstore.service.NetstoreServiceImpl 类以及相关的类都拷贝到 axis/WEB-INF/classes 目录下。NetstoreServiceImpl.class 的存放位置为：axis/WEB-INF/classes/netstore/service/NetstoreServiceImpl.class。

接下来，在 axis 应用已经运行的状态下，通过 AXIS 的专门的客户程序 AdminClient 来发布 SOAP 服务。在 DOS 命令行中，运行以下命令，就会发布 SOAP 服务：

```
java  org.apache.axis.client.AdminClient  deploy.wsdd
```

在以上命令中，参数“deploy.wsdd”代表 18.3.1 小节介绍的 Web 服务发布描述文件。如果 SOAP 服务发布成功，那么访问 http://localhost:8080/axis/servlet/AxisServlet 时，应该看到输出网页上包含了名为“netstoreservice”的 Web 服务，如图 18-6 所示。

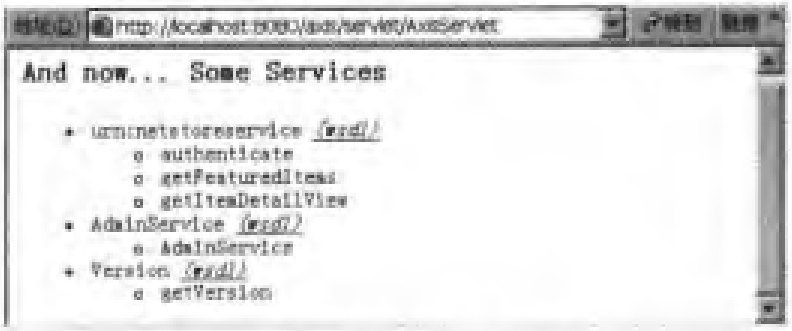

图 18-6 axis 应用提供的 Web 服务

如果要删除 SOAP 服务，可以在 DOS 命令行中，运行以下命令：

```
java  org.apache.axis.client.AdminClient  undeploy.wsdd
```

18.4 在 Struts 应用中访问 SOAP 服务

在第 6 章中介绍了如何通过业务代理模式来削弱 Web 应用和模型之间的关系，本章依

然使用这种业务代理模式，当模型的实现方式发生变化，无需修改 Struts 应用的控制层组件，而只需为控制层提供一个新的业务代理实现类 NetstoreWebServiceDelegate 即可，如图 18-7 所示。

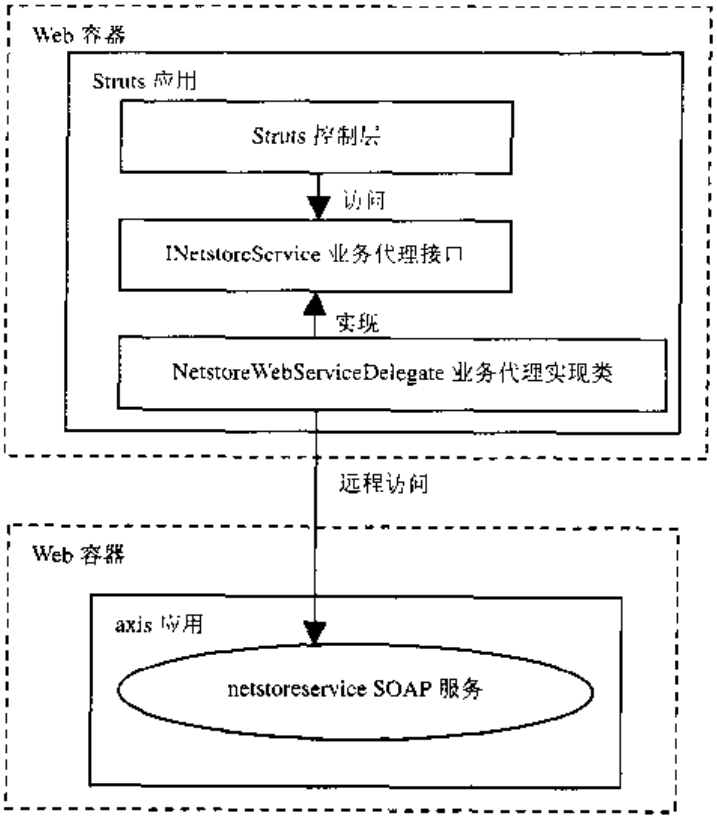

图 18-7　Struts 应用通过业务代理访问 Web 服务

例程 18-2 是 NetstoreWebServiceDelegate 类的源程序。

例程 18-2　NetstoreWebServiceDelegate.java

```
package netstore.service;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.ser.BeanDeserializerFactory;
import org.apache.axis.encoding.ser.BeanSerializerFactory;
import javax.xml.namespace.QName;
import javax.xml.rpc.ParameterMode;
import java.rmi.RemoteException;
import java.util.Hashtable;
import java.util.List;
import java.util.Date;
import javax.servlet.ServletContext;
import netstore.catalog.view.ItemDetailView;
import netstore.catalog.view.ItemSummaryView;
import netstore.customer.view.UserView;
```

```
import netstore.framework.exceptions.*;
import netstore.service.INetstoreService;

/**
* This class is a business delegate that supports the implementation of the
* INetstoreService interface using the Netstore session bean.
*/
public class NetstoreWebServiceDelegate implements INetstoreService {

  private Service netstoreService;
  ServletContext servletContext = null;
  String endpoint=null;
  public NetstoreWebServiceDelegate() {
    init();
  }

  private void init() {
    try {
      endpoint = "http://localhost:8080/axis/services/netstoreservice";
      netstoreService = new Service();
    }
    catch (Exception e) {
      throw new RuntimeException(e.getMessage());
    }
  }

  public UserView authenticate( String email, String password )
  throws InvalidLoginException, ExpiredPasswordException,
  AccountLockedException, DatastoreException {
    try {
        Call        call      = (Call) netstoreService.createCall();
        QName qn = new QName("urn:BeanService","UserView");
        Class cls = UserView.class;
        call.registerTypeMapping(cls, qn,
        new org.apache.axis.encoding.ser.BeanSerializerFactory(cls, qn),
        new org.apache.axis.encoding.ser.BeanDeserializerFactory(cls, qn));

        QName dateqn = new QName("urn:BeanService","Date");
        Class datecls = Date.class;
        call.registerTypeMapping(datecls, dateqn,
        new org.apache.axis.encoding.ser.BeanSerializerFactory(datecls, dateqn),
        new org.apache.axis.encoding.ser.BeanDeserializerFactory(datecls, dateqn));

        call.setTargetEndpointAddress( new java.net.URL(endpoint) );
        call.setOperationName(new QName("urn:netstoreservice", "authenticate") );
        call.addParameter( "arg1",    org.apache.axis.encoding.XMLType.XSD_STRING ,
```

```
            ParameterMode.IN );
            call.addParameter( "arg2",    org.apache.axis.encoding.XMLType.XSD_STRING ,
              ParameterMode.IN );
            call.setReturnType(qn,UserView.class);

            return (UserView) call.invoke( new Object[] { email,password } );

        }
        catch (javax.xml.rpc.ServiceException e) {
            throw DatastoreException.datastoreError(e);
        }catch (java.net.MalformedURLException e) {
            throw DatastoreException.datastoreError(e);
        }catch(java.rmi.RemoteException e){
            throw DatastoreException.datastoreError(e);
        }
    }

    public List getFeaturedItems() throws DatastoreException {
        try {
            Call        call        = (Call) netstoreService.createCall();

            QName sumqn = new QName("urn:BeanService","ItemSummaryView");
            Class sumcls = ItemSummaryView.class;
            call.registerTypeMapping(sumcls, sumqn,
            new org.apache.axis.encoding.ser.BeanSerializerFactory(sumcls, sumqn),
            new org.apache.axis.encoding.ser.BeanDeserializerFactory(sumcls, sumqn));

            QName dateqn = new QName("urn:BeanService","Date");
            Class datecls = Date.class;
            call.registerTypeMapping(datecls, dateqn,
            new org.apache.axis.encoding.ser.BeanSerializerFactory(datecls, dateqn),
            new org.apache.axis.encoding.ser.BeanDeserializerFactory(datecls, dateqn));

            QName qn = new QName("urn:BeanService","List");
            Class cls = List.class;
            call.registerTypeMapping(cls, qn,
            new org.apache.axis.encoding.ser.BeanSerializerFactory(cls, qn),
            new org.apache.axis.encoding.ser.BeanDeserializerFactory(cls, qn));

            call.setTargetEndpointAddress( new java.net.URL(endpoint) );
            call.setOperationName(new QName("urn:netstoreservice", "getFeaturedItems") );
            call.setReturnType(qn,List.class);
            return (List) call.invoke(new Object[] { } );

        }
        catch (javax.xml.rpc.ServiceException e) {
```

```
            throw DatastoreException.datastoreError(e);
        }catch (java.net.MalformedURLException e) {
            throw DatastoreException.datastoreError(e);
        }catch(java.rmi.RemoteException e){
            e.printStackTrace();
            throw DatastoreException.datastoreError(e);
        }
    }

    public ItemDetailView getItemDetailView( String itemId )
    throws DatastoreException {
        try {
            Call        call      = (Call) netstoreService.createCall();
            call.setTargetEndpointAddress( new java.net.URL(endpoint) );

            QName dateqn = new QName("urn:BeanService","Date");
            Class datecls = Date.class;
            call.registerTypeMapping(datecls, dateqn,
            new org.apache.axis.encoding.ser.BeanSerializerFactory(datecls, dateqn),
            new org.apache.axis.encoding.ser.BeanDeserializerFactory(datecls, dateqn));

            QName qn = new QName("urn:BeanService","ItemDetailView");
            Class cls = ItemDetailView.class;
            call.registerTypeMapping(cls, qn,
            new org.apache.axis.encoding.ser.BeanSerializerFactory(cls, qn),
            new org.apache.axis.encoding.ser.BeanDeserializerFactory(cls, qn));
            call.setOperationName(new QName("urn:netstoreservice", "getItemDetailView") );
            call.addParameter( "arg1",    org.apache.axis.encoding.XMLType.XSD_STRING ,
              ParameterMode.IN );
            call.setReturnType(qn,ItemDetailView.class);

            return (ItemDetailView) call.invoke( new Object[]{ itemId } );

        }
        catch (javax.xml.rpc.ServiceException e) {
            throw DatastoreException.datastoreError(e);
        }catch (java.net.MalformedURLException e) {
            throw DatastoreException.datastoreError(e);
        }catch(java.rmi.RemoteException e){
            throw DatastoreException.datastoreError(e);
        }
    }

    public void logout( String email ) {
    // Do nothing for this example
    }
```

```
    public void destroy() {
    // Do nothing for this example
    }
    public void setServletContext( ServletContext ctx ){
      this.servletContext = ctx;
    }

    public ServletContext getServletContext(){
      return servletContext;
    }
  }
```

NetstoreWebServiceDelegate 类的构造方法调用自身的 init()方法。在 init()方法中先定义 netstoreservice 服务的位置，然后创建一个 Service 实例：

```
endpoint = "http://localhost:8080/axis/services/netstoreservice";
netstoreService = new Service();
```

每个业务方法中都调用 netstoreservice 服务的相应方法来完成业务。以 authenticate()方法为例，先通过 Service 实例创建 Call 实例，它提供了发出 RPC 请求的功能：

```
Call   call = (Call) netstoreService.createCall();
```

接下来为需要在网络上传输的方法参数和返回值指定 XML 序列器和反序列化器。当 SOAP 客户访问 SOAP 服务的 authenticate ()方法时，将在网络上传输的 Java 数据类型包括：UserView 和 Date 类。以下代码为这些类设置 XML 序列器为 org.apache.axis.encoding.ser.BeanSerializerFactory，反序列化器为 org.apache.axis.encoding.ser.BeanDeserializerFactory：

```
QName qn = new QName("urn:BeanService","UserView");
Class cls = UserView.class;
call.registerTypeMapping(cls, qn,
new org.apache.axis.encoding.ser.BeanSerializerFactory(cls, qn),
new org.apache.axis.encoding.ser.BeanDeserializerFactory(cls, qn));

QName dateqn = new QName("urn:BeanService","Date");
Class datecls = Date.class;
call.registerTypeMapping(datecls, dateqn,
new org.apache.axis.encoding.ser.BeanSerializerFactory(datecls, dateqn),
new org.apache.axis.encoding.ser.BeanDeserializerFactory(datecls, dateqn));
```

接下来为 Call 实例设置 netstoreservice 服务的位置、调用的方法名、方法的参数和方法的返回类型：

```
call.setTargetEndpointAddress( new java.net.URL(endpoint) );
call.setOperationName(new QName("urn:netstoreservice", "authenticate") );
call.addParameter( "arg1",
     org.apache.axis.encoding.XMLType.XSD_STRING ,ParameterMode.IN );
call.addParameter( "arg2",   org.apache.axis.encoding.XMLType.XSD_STRING ,
```

```
        ParameterMode.IN );
    call.setReturnType(qn,UserView.class);
```

接下来调用 Call 实例的 invoke()方法：

```
return (UserView) call.invoke( new Object[] { email,password } );
```

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。

可以在 netstore 应用的 web.xml 文件中配置业务代理实现类，ActionServlet 类的初始化参数“netstore-service-class”用来指定业务代理实现类：

```
<servlet>
    <servlet-name>netstore</servlet-name>
    <servlet-class>netstore.framework.ExtendedActionServlet</servlet-class>
    ……
    <init-param>
        <param-name>netstore-service-class</param-name>
        <param-value>netstore.service.NetstoreWebServiceDelegate</param-value>
    </init-param>
    ……
</servlet>
```

18.5 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的协议，它以 XML 作为数据传送的方式。本章采用 SOAP 服务实现了 netstore 应用的模型，由于合理运用了业务代理模式，因此当模型的实现方式发生变化时，对 Struts 应用的控制层组件没有任何影响。

在 Tomcat 上发布本章样例的详细步骤请参见附录 D.4.5（在工作模式 5 下发布 netstore 应用）。

第 19 章　Struts 与 Apache 通用日志包

在应用程序中输出日志有三个目的：

- 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，向文件或控制台打印代码的调试信息。

本章介绍如何在 Struts 应用中使用 Apache 通用日志包，来输出各种级别的日志，并且控制日志的输出地点和输出格式。

19.1　Apache 通用日志包概述

Apache 通用日志包（Commons Logging Package）是 Apache 的一个开放源代码项目，它提供了一组通用的日志接口，用户可以自由地选择实现日志接口的第三方软件。通用日志包目前支持以下日志实现：

- Log4j 日志器（下载网址：http://jakarta.apache.org/log4j）
- JDK 1.4 Logging 日志器（在 JDK 1.4 中自带了该日志器）
- SimpleLog 日志器（把日志消息输出到标准的系统错误流 System.err）
- NoOpLog 日志器（不输出任何日志消息）

通用日志包中的两个常用接口为 LogFactory 和 Log，下面分别介绍它们的用法。

19.1.1　Log 接口

通用日志包把日志消息分为 6 种级别：FATAL、ERROR、WARN、INFO、DEBUG 和 TRACE。其中 FATAL 级别最高，TRACE 级别最低。通用日志包采用日志级别机制，可以灵活地控制输出的日志内容。

org.apache.commons.logging.Log 接口代表日志器，它提供了一组输出日志的方法：

- fatal（Object message）：输出 FATAL 级别的日志消息。
- error（Object message）：输出 ERROR 级别的日志消息。
- warn（Object message）：输出 WARN 级别的日志消息。
- info（Object message）：输出 INFO 级别的日志消息。
- debug（Object message）：输出 DEBUG 级别的日志消息。
- trace（Object message）：输出 TRACE 级别的日志消息。

对于以上输出日志的方法，只有当它输出日志的级别大于或等于为日志器配置的日志级别时，这个方法才会被真正执行。例如，如果日志器的日志级别为 WARN，那么在程序

中，它的 fatal()、error()和 warn()方法会被执行，而 info()、debug()和 trace()方法不会被执行。

如何指定日志器的日志级别，这依赖于具体的日志器实现。在本章的 19.3 和 19.4 节将分别介绍为 SimpleLog 和 Log4J 设置日志级别的方法。

Log 接口还提供了一组判断是否允许输出特定级别的日志消息的方法：

- isFatalEnabled( )
- isErrorEnabled( )
- isWarnEnalbed( )
- isInfoEnabled( )
- isDebugEnabled( )
- isTraceEnabled( )

在程序中输出某种级别的日志消息之前，提倡先调用以上方法来判断该级别的日志是否允许输出，这有助于提高应用的性能。例如以下代码先把日志消息添加到 StringBuffer 中，最后再调用日志器的 debug()方法输出日志：

```
StringBuffer buf = new StringBuffer();
buf.append( "Login Successful - " );
buf.append( "Name: " );
buf.append( userView.getFirstName() );
buf.append( " " );
buf.append( userView.getLastName() );
buf.append( " - " );
buf.append( "Email: " );
buf.append( userView.getEmailAddress() );

// Log the information for auditing purposes
log.debug( buf.toString() );
```

对于以上代码，如果日志器实际上不允许输出 DEBUG 级别的日志，那么执行日志器的 debug()方法不会输出任何消息，此时向 StringBuffer 中添加消息的一大串操作都将是多余的。为了提高性能，可以合理地使用 isDebugEnabled( )方法，避免应用执行多余的操作：

```
if ( log.isDebugEnabled() ){
  StringBuffer buf = new StringBuffer();
  buf.append( "Login Successful - " );
  buf.append( "Name: " );
  buf.append( userView.getFirstName() );
  buf.append( " " );
  buf.append( userView.getLastName() );
  buf.append( " - " );
  buf.append( "Email: " );
  buf.append( userView.getEmailAddress() );
```

```
        // Log the UserView for auditing purposes
        log.debug( buf.toString() );
    }
```

19.1.2　LogFactory 接口

org.apache.commons.logging.LogFactory 接口提供了获得日志器实例的两个静态方法：

```
public static Log getLog(String name)throws LogConfigurationException;
public static Log getLog(Class class)throws LogConfigurationException;
```

第一个 getLog()方法以 name 参数作为日志器的名字；第二个 getLog()方法以 class 参数指定的类的名字作为日志器的名字，以下是第二个 getLog()方法的一种实现方式：

```
public static Log getLog(Class class)throws LogConfigurationException{
    //call getLog(String name)
    getLog(class.getName());
}
```

19.2　常用的日志实现

本节将介绍几种常用的日志实现：

- NoOpLog 日志器
- SimpleLog 日志器
- Log4J 日志器

19.2.1　NoOpLog 日志器

在通用日志包中自带了 org.apache.commons.logging.impl.NoOpLog 日志实现类，它实现了 Log 接口，但是它的输出日志方法不执行任何操作：

```
/** Do nothing */
public void trace(Object message) { }
/** Do nothing */
public void debug(Object message) { }
/** Do nothing */
public void info(Object message) { }
/** Do nothing */
public void warn(Object message) { }
/** Do nothing */
public void error(Object message) { }
/** Do nothing */
public void fatal(Object message) { }
```

19.2.2 SimpleLog 日志器

在通用日志包中自带了 org.apache.commons.logging.impl.SimpleLog 日志实现类，它实现了 Log 接口，它把日志消息输出到标准的系统错误流 System.err，在 SimpleLog 初始化的过程中，从名为“simplelog.properties”的属性文件中读取以下属性：

- org.apache.commons.logging.simplelog.defaultlog：为 SimpleLog 的所有实例设置默认的日志级别，可选值包括：fatal、error、warn、info、debug 和 trace。如果没有设置这个属性，其默认值为 info。
- org.apache.commons.logging.simplelog.showShortLogname：如果为 true，表示在输出的日志消息中应包含当前日志器实例的简写名字。默认值为 true。
- org.apache.commons.logging.simplelog.showdatetime：如果为 true，表示在输出的日志消息中应包含当前时间信息。默认值为 false。

SimpleLog 将根据以上属性来控制日志输出级别和输出格式。在它的输出日志方法中，先判断是否允许输出该级别的日志，当判断结果为 true 时，才输出日志。以下是它的 debug() 方法的代码：

```
public final void debug(Object message) {
    if (isLevelEnabled(SimpleLog.LOG_LEVEL_DEBUG)) {
        log(SimpleLog.LOG_LEVEL_DEBUG, message, null);
    }
}
```

以上 debug()方法先调用 isLevelEnabled()方法来判断是否允许输出该级别的日志。以下是 isLevelEnabled()方法的代码，其中 currentLogLevel 变量代表当前日志器的日志级别：

```
protected boolean isLevelEnabled(int logLevel) {
    // log level are numerically ordered so can use simple numeric
    // comparison
    return (logLevel >= currentLogLevel);
}
```

19.2.3 Log4J 日志器

Log4J 是 Apache 的一个开放源代码项目，它是一个日志操作包。Log4J 允许灵活地指定日志消息输出的目的地（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，通过定义日志消息的级别，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。

Log4J 主要由三大组件构成：

- Logger：负责生成日志，并能根据配置的日志级别来决定什么日志消息应该被输出，什么日志消息应该被忽略。
- Appender：定义日志消息输出的目的地，指定日志消息应该被输出到什么地方，

这些地方可以是控制台、文件和网络设备等。

- Layout：指定日志消息的输出格式。

这三个组件协同工作，使得开发者能够依据日志消息类别来输出日志，并能够在程序运行期间，控制日志消息的输出格式以及日志存放地点。

19.3　配置通用日志接口

在 Web 应用中使用通用日志包，首先需要进行以下配置：

- 准备有关的 JAR 文件。
- 在 commons-logging.properties 属性文件中指定日志器（即日志实现类）。
- 为特定的日志器设置日志级别、输出格式和输出地点等属性。

19.3.1　准备 JAR 文件

通用日志包的单独下载网址为：http://jakarta.apache.org/commons/logging/。在 Struts 1.1 的下载软件中也包含了通用日志包，包文件名为 commons-logging.jar。如果 Web 应用中使用了通用日志接口，应该把 commons-logging.jar 文件复制到 WEB-INF/lib 目录下。

如果选用 Tomcat 5.0.24 作为 Web 容器，则在<CATALINA_HOME>\bin 目录下已经包含了 commons-logging-api.jar 文件，当 Tomcat 启动时，会加载这个文件中的类。因此可以不必把 commons-logging.jar 文件复制到 Web 应用的 WEB-INF/lib 目录下。

commons-logging.jar 文件只包含了 SimpleLog 和 NoOpLog 两种日志实现。如果 Web 应用采用了其他第三方的日志实现，如 Log4J，则应该到 Apache 网站上单独下载它的包文件，把 Log4j 的 JAR 文件复制到 WEB-INF/lib 目录下。

19.3.2　指定日志器

通用日志接口从名为“commons-logging.properties”的属性文件中获取实现日志接口的日志器信息，这个文件位于 WEB-INF/classes 目录下。

在该文件中的“org.apache.commons.logging.Log”属性指定日志实现类。例如，如果日志实现类为 SimpleLog，则配置代码如下：

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

在运行时，通用日志接口会检索 commons-logging.properties 文件，并且实例化由“org.apache.commons.logging.Log”属性指定的日志实现类。

如果采用 Log4J 日志器，只需重新设置 org.apache.commons.logging.Log 属性即可：

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JCategoryLog
```

19.3.3 设置日志器的属性

多数日志器都允许以配置的方式，指定日志的输出格式、地点或级别等。下面讲述如何为 SimpleLog 设置属性。此外，由于配置 Log4J 的属性比较复杂，因此在本章 19.4 节单独介绍 Log4J 的配置。

1. 配置 SimpleLog 日志器

如果采用 SimpleLog 作为日志器，那么其日志输出地点是固定的，为 System.err，无需配置，可以在 simplelog.properties 属性文件中设置其日志级别。例如以下代码指定 SimpleLog 的日志级别为 info：

```
org.apache.commons.logging.simplelog.defaultlog = info
```

应该把 simplelog.properties 文件放在 WEB-INF/classes 目录下。

19.4 配置 Log4J

Log4J 由三个重要的组件构成：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，而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。

19.4.1 配置 Log4J 的一般步骤

配置 Log4J，需要分别设置它的 Logger、Appender 和 Layout 的属性。

1. 配置 Logger 组件

Logger 组件支持继承关系，所有的 Logger 组件都直接或间接继承 rootLogger。配置 rootLooger 的语法为：

```
log4j.rootLogger = [priority] , appenderName, appenderName, ...
```

其中，priority 是日志级别，可选值包括 OFF、FATAL、ERROR、WARN、INFO、DEBUG、TRACE 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消息的开关。比如在这里定义了 INFO 级别，则应用程序中所有 DEBUG 和 TRACE 级别的日志消息将不被打印出来。

appenderName 指定 Appender 组件，用户可以同时指定多个 Appender 组件。例如，以下代码指定 rootLogger 的日志级别为 INFO，它有两个 Appender，名为“console”和“file”：

```
log4j.rootLogger=INFO,console,file
```

2. 配置 Appender 组件

配置日志消息输出目的地 Appender，其语法为：

```
log4j.appender.appenderName = fully.qualified.name.of.appender.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（将日志消息以流格式发送到任意指定的地方）

例如，以下代码定义了一个名为“file”的 Appender，它把日志消息输出到 log.txt 文件：

```
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=log.txt
```

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（包含日志消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）

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 19-1。

表 19-1　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	日志消息所在的类名
%m%n	表示日志消息的内容

例如，如果为名为“file”的 Appender 配置 PatternLayout 布局，则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：

```
http8080-Processor24 INFO - This is an info message
```

在以上的日志内容中，“%t”对应“http8080-Processor24”，“%p”对应“INFO”，“%m%n”对应后面具体的日志消息。

19.4.2 Log4J 的配置样例

假定根据实际需要，要求程序中的日志消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，并且当日志消息输出到控制台时采用 SimpleLayout 布局，当日志消息输出到文件时采用 PatternLayout 布局，此时 Logger、Appender 和 Layout 这三个组件的关系如图 19-1 所示。

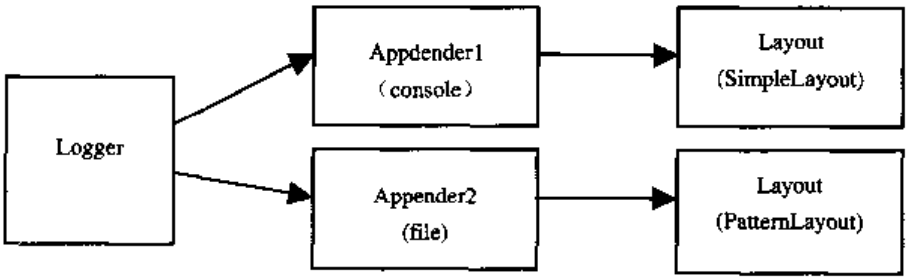

图 19-1 Logger、Appender 和 Layout 三个组件的关系

根据如图 19-1 所示创建如下名为 log4j.properties 的属性文件，该文件位于 WEB-INF/classes 目录下：

```
## LOGGERS ##
#define a logger named helloAppLogger
log4j.rootLogger=INFO,console,file

## APPENDERS ##
# define an appender named console, which is set to be a ConsoleAppender
log4j.appender.console=org.apache.log4j.ConsoleAppender

# define an appender named file, which is set to be a RollingFileAppender
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=log.txt

## LAYOUTS ##
# assign a SimpleLayout to console appender
log4j.appender.console.layout=org.apache.log4j.SimpleLayout

# assign a PatternLayout to file appender
log4j.appender.file.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.file.layout.ConversionPattern=%t %p - %m%n
```

以上配置代码指定 rootLogger 的日志级别为“INFO”，日志输出地点为控制台和 log.txt 文件。当日志输出到控制台时，采用 SimpleLayout 布局；当日志输出到文件时，采用 PatternLayout 布局。

19.4.3　Log4J 对应用性能的影响

如果在程序运行中输出大量日志，显然会对应用的性能造成一定的影响。Log4J 对性能的影响程度取决于以下因素：

- 日志输出目的地：例如把日志输出到控制台的速度和输出到文件系统的速度是不一样的。
- 日志输出格式：例如采用 SimpleLayout 输出日志消息比采用 PatternLayout 简单，因此速度更快。
- 日志级别：日志级别设置得越低，输出的日志内容越多，对性能的影响也越大。在产品测试阶段，可以把日志级别设置得低一些，便于跟踪和调试程序，而在产品发布阶段，应该把日志级别设置得高一些。

在运用 Log4J 时，应该充分考虑它对应用性能的影响，合理地配置 Log4J 的各种属性。

19.5　在 Struts 应用中访问通用日志接口

在 Struts 应用中访问通用日志接口，首先应该按照 19.3 节的介绍配置通用日志接口，接下来就可以在 Java 类或 JSP 文件中访问它。通用日志接口的优点在于它允许灵活地指定日志实现。当日志实现发生改变时，例如由 SimpleLog 改为 Log4J，对 Struts 应用程序代码没有任何影响，如图 19-2 所示。

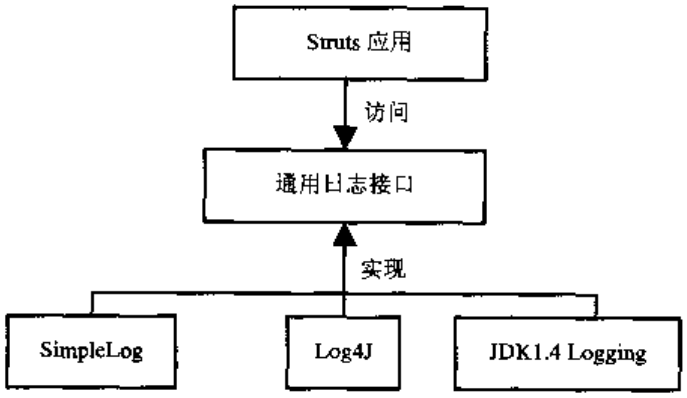

图 19-2　在 Struts 应用中访问通用日志接口

本节以 helloapp 应用为例，介绍如何在 Action 类和 JSP 中访问通用日志接口。所有的源文件位于本书配套光盘的 sourcecode/helloapp/version3 目录下。

19.5.1 在 Action 类中访问通用日志接口

在 Action 类或其他 Java 类中访问通用日志接口的步骤如下。

步骤

（1）通过 import 语句引入 logging API：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

（2）调用 LogFactory 类的静态方法 getLog()，获得 Log 类的一个实例：

```
Log log=LogFactory.getLog("helloapplog");
```

（3）调用 Log 类的输出日志方法：

```
log.trace("This is a trace message");
log.debug("This is a debug message");
log.info("This is an info message");
log.warn("This is a warn message");
log.error("This is an error message");
log.fatal("This is a fatal message");
```

例程 19-1 为使用了通用日志接口的 HelloAction 的源程序。

例程 19-1　HelloAction.java

```
package hello;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.util.MessageResources;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;


public final class HelloAction extends Action {
```

```
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
    throws Exception {

        Log log=LogFactory.getLog("helloapplog");
        log.trace("This is a trace message");
        log.debug("This is a debug message");
        log.info("This is an info message");
        log.warn("This is a warn message");
        log.error("This is an error message");
        log.fatal("This is a fatal message");

        // These "messages" come from the ApplicationResources.properties file
        MessageResources messages = getResources(request);

        ActionMessages errors = new ActionMessages();
        String userName = (String)((HelloForm) form).getUserName();

        String badUserName = "Monster";

        if (userName.equalsIgnoreCase(badUserName)) {
            errors.add("username", new ActionMessage("hello.dont.talk.to.monster",
             badUserName ));
            saveErrors(request, errors);
            return (new ActionForward(mapping.getInput()));
        }

        PersonBean pb = new PersonBean();
        pb.setUserName(userName);
        pb.saveToPersistentStore();

        request.setAttribute( Constants.PERSON_KEY, pb);

        // Remove the Form Bean - don't need to carry values forward
        request.removeAttribute(mapping.getAttribute());

        // Forward control to the specified success URI
        return (mapping.findForward("SayHello"));
    }
}
```

通过浏览器访问 HelloAction，以下是采用不同的日志器的运行结果。

- 按照 19.3 节介绍的配置步骤，指定日志器为 SimpleLog，日志级别为 INFO。此时日志内容输出到控制台，如图 19-3 所示。从图 19-3 中可以看到，由于日志级别

为 INFO，所以没有输出 TRACE 和 DEBUG 级别的日志。

图 19-3 当日志器指定为 SimpleLog 时，HelloAction 在控制台上输出的日志

如果直接访问 http://localhost:8080/helloapp/HelloWorld.do，会发现在控制台上没有输出任何日志。这是因为 Struts 框架首先对 HelloForm 表单进行验证，由于没有提供任何表单数据，验证失败，此时 Struts 框架根本不会调用 HelloAction 的 execute()方法，而是直接返回包含错误信息的 hello.jsp 网页。

- 按照 19.3 和 19.4 节的介绍配置步骤，指定日志器为 Log4J，日志级别为 INFO，日志内容输出到控制台和文件。此时控制台的输出内容如图 19-4 所示，此外，在<CATALINA_HOME>/bin 目录下生成 log.txt 文件，其日志内容如下：

```
http8080-Processor24 INFO - This is an info message
http8080-Processor24 WARN - This is a warn message
http8080-Processor24 ERROR - This is an error message
http8080-Processor24 FATAL - This is a fatal message
```

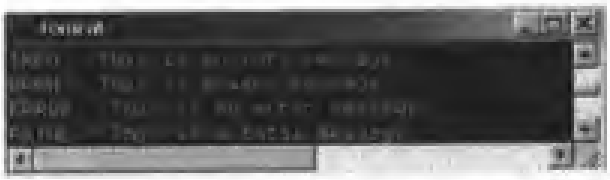

图 19-4 当指定日志器为 Log4J 时，HelloAction 在控制台上输出的日志

19.5.2 在 JSP 中访问通用日志接口

在 JSP 中访问通用日志接口有两种方式：编程方式和采用 Log 标签库方式，下面对它们分别进行介绍。

1. 以编程方式访问通用日志接口

可以按照和 Java 类访问通用日志接口相同的方式，在 JSP 的程序代码中访问通用日志接口，参见例程 19-2（logtest1.jsp）。

例程 19-2 logtest1.jsp

```
<%@ page import="org.apache.commons.logging.Log" %>
<%@ page import="org.apache.commons.logging.LogFactory" %>

<%-- Get a reference to the logger for this class --%>
<% Log logger = LogFactory.getLog( this.getClass() ); %>
```

```
<html>
<head>
  <title>Using Commons Logging in a JSP page</title>
</head>

<body>
  <% logger.warn( "This is a warn message from a jsp" ); %>
  <% logger.error( "This is an error message from a jsp" ); %>
  There should be two log messages in the log file.
</body>
</html>
```

从浏览器中访问 logtest1.jsp，以下是采用不同的日志器的运行结果。

- 按照 19.3 节介绍的配置步骤，指定日志器为 SimpleLog，日志级别为 INFO。此时日志内容输出到控制台，如图 19-5 所示。

图 19-5　当指定日志器为 SimpleLog 时，logtest1.jsp 在控制台上输出的日志

- 按照 19.3 和 19.4 节介绍的配置步骤，指定日志器为 Log4J，日志级别为 INFO，日志内容输出到控制台和文件。此时控制台的输出内容如图 19-6 所示。此外，在 <CATALINA_HOME>/bin 目录下生成 log.txt 文件，其日志内容如下：

```
http8080-Processor24 WARN - This is a warn message from a jsp
http8080-Processor24 ERROR - This is an error message from a jsp
```

图 19-6　当指定日志器为 Log4J 时，logtest1.jsp 在控制台上输出的日志

2. 使用 Log 标签库

Apache 提供了一个客户化 Log 标签库，通过它可以在 JSP 文件中方便地访问通用日志接口，无需编写程序代码。使用 Log 标签库的前提条件是通用日志接口必须采用 Log4J 作为日志实现。

以下是使用 Log 标签库的步骤。

（1）从 Apache 网站上下载 Log 标签库，在本书配套光盘的 software 目录下也提供了

jakarta-taglibs-log-current.zip 文件。解压该文件，把 taglibs-log.tld 文件复制到 WEB-INF 目录下，并把 taglibs-log.jar 文件复制到 WEB-INF/lib 目录下。

（2）在 web.xml 文件中声明对 Log 标签库的引用：

```
<taglib>
    <taglib-uri>/WEB-INF/taglibs-log.tld</taglib-uri>
    <taglib-location>/WEB-INF/taglibs-log.tld</taglib-location>
</taglib>
```

（3）在 JSP 文件中访问 Log 标签库，参见例程 19-3（logtest2.jsp）。

例程 19-3　logtest2.jsp

```
<%@ taglib uri="/WEB-INF/taglibs-log.tld" prefix="log" %>
<log:debug message="This is a debug message using the log tag" />

<html>
<head>
  <title>Using the Log Tag in a JSP page</title>
</head>

<body>
  <log:info message="This is an info message using the log tag" />
  <log:warn message="This is a warn message using the log tag" />

  There should be two log messages in the log4j log file.
</body>
</html>
```

在 logtest2.jsp 中，<log:debug>、<log:info>和<log:warn>标签分别输出 DEBUG、INFO 和 WARN 级别的日志消息。

按照 19.3 和 19.4 节介绍的配置步骤，指定日志器为 Log4J，日志级别为 INFO，日志内容输出到控制台和文件。通过浏览器访问 logtest2.jsp，此时控制台的输出内容如图 19-7 所示，此外，在<CATALINA_HOME>/bin 目录下生成 log.txt 文件，日志内容如下：

```
http8080-Processor24 INFO - This is an info message using the log tag
http8080-Processor24 WARN - This is a warn message using the log tag
```

由于 Log4J 的日志级别为 INFO，因此<log:debug>标签并没有输出 DEBUG 级别的日志。

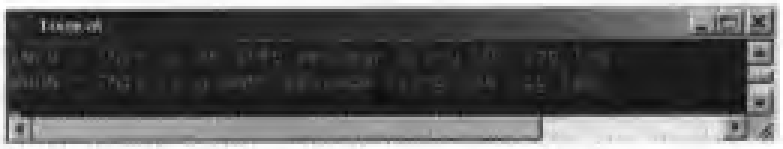

图 19-7　logtest2.jsp 在控制台上输出的日志

Log 标签库还包含一个<log:dump>标签，它能够输出特定范围内所有对象的信息，它

的 scope 属性指定对象的范围，可选值包括：page、request、session 和 application，参见例程 19-4（logtest3.jsp）。

例程 19-4　logtest3.jsp

```
<%@ taglib uri="/WEB-INF/taglibs-log.tld" prefix="log" %>
<html>
<head>
  <title>Using the Log Tag in a JSP page</title>
</head>
<body>
<%
  request.setAttribute("stringInRequestScope","This is a string in request scope");
  session.setAttribute("stringInSessionScope","This is a string in session scope");
%>
  --------dump objects in page scope----------
  <log:dump scope="page"/>
  --------dump objects in request scope----------
  <log:dump scope="request"/>
  --------dump objects in session scope----------
  <log:dump scope="session"/>
  --------dump objects in application scope----------
  <log:dump scope="application"/>
</body>
</html>
```

按照 19.3 和 19.4 节的配置步骤，指定日志器为 Log4J，日志级别为 INFO，日志内容输出到控制台和文件。通过浏览器访问 logtest3.jsp，其输出网页，如图 19-8 所示。

图 19-8　logtest3.jsp 的输出网页

19.6 小　　结

本章介绍了在 Struts 应用中访问 Apache 通用日志接口的步骤。通用日志接口的优点在于允许用户灵活地指定日志实现，当用户改变了日志实现时，不会对 Struts 应用的程序代码造成任何影响。

本章详细介绍了配置通用日志接口的步骤。通用日志接口的属性文件为 commons-logging.properties，SimpleLog 的属性文件为 simplelog.properties，Log4J 的属性文件为 log4j.properties。这三个文件均位于 Web 应用的 WEB-INF/classes 目录下。

在 Java 程序中，先通过 LogFactory 获得 Log 实例，然后再调用 Log 实例的各种输出日志方法，如 info()或 warn()方法等，来输出特定级别的日志消息。

在 JSP 文件中还可以通过 Log 标签库来访问通用日志接口。使用 Log 标签库的前提条件是通用日志接口必须采用 Log4J 作为日志实现。

第 20 章　用 ANT 工具管理 Struts 应用

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。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 Web 应用的一种常用开发目录结构，然后以 helloapp 应用为例，介绍如何用 ANT 工具来创建和发布 Struts 应用。

20.1　Web 应用常用的开发目录结构

在 Web 应用的开发阶段，需要创建或准备各种类型的文件，如 Java 源文件、Java 类文件、JAR 文件、JSP 文件、HTML 文件、XML 文件、TLD 文件和属性文件。到了 Web 应用的发布阶段，需要把相关的文件打包为 WAR 文件再发布。合理地组织这些文件的目录结构，可以简化项目的统一管理与开发工作。如图 20-1 所示为 Web 应用的一种常用开发目录结构。

```
app.home ----+              - Web 应用项目的顶层目录
             +-- build      - 存放由 ANT 工具生成的 Web 应用的开放目录结构
             +-- deploy     - 存放由 ANT 工具生成的 Web 应用的 WAR 文件
             +-- doc        - 存放由 ANT 工具生成的 JavaDoc 文件
             +-- lib        - 存放所有的 JAR 文件
             +-- classes    - 存放由 ANT 工具编译生成的 Java 类文件
             +-- src        - 存放 Java 源文件和属性文件
             +-- web        - 存放 Web 应用的 JSP、HTML 和图片文件

                +--WEB-INF  - 存放 web.xml、struts-config.xml 和 TLD 文件等
```

图 20-1　Web 应用常用的开发目录结构

20.2　安装配置 ANT

ANT 的下载地址为 http://ant.apache.org/，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 DOS 命令行下直接运行 ANT。

在上述设置完成后，就可以使用 ANT 了。

20.3 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。表 20-1 列出了最常用的构建工程的 ANT 内置任务。

表 20-1 ANT 内置任务

ANT 任务	描 述
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
javadoc	为 Java 源文件创建 JavaDoc 文档

下面为 helloapp 应用创建一个 build.xml 文件，用于编译 helloapp 应用的 Java 源代码，并且将这个应用的相关文件打包为 WAR 文件，然后把它发布到 Web 服务器中。

首先创建 helloapp 应用的初始目录结构，参见本书配套光盘的 sourcecode/helloapp/version4/helloappBeforeBuild，在这个目录下已经包含了如图 20-2 所示的文件目录结构。

从图 20-2 中可以看出，在 helloapp 应用项目的根目录下包括 build.xml 和 build.properties 文件，此外还包含 lib、src 和 web 三个子目录：

- lib 目录：包含所有的 JAR 文件。
- src 目录：包含所有的 Java 源文件，以及供 Struts 框架访问的资源文件 application.properties。
- web 目录：包含 Web 应用的 JSP 和 GIF 文件。在 WEB-INF 子目录下包含 web.xml、struts-config.xml 和 TLD 文件。

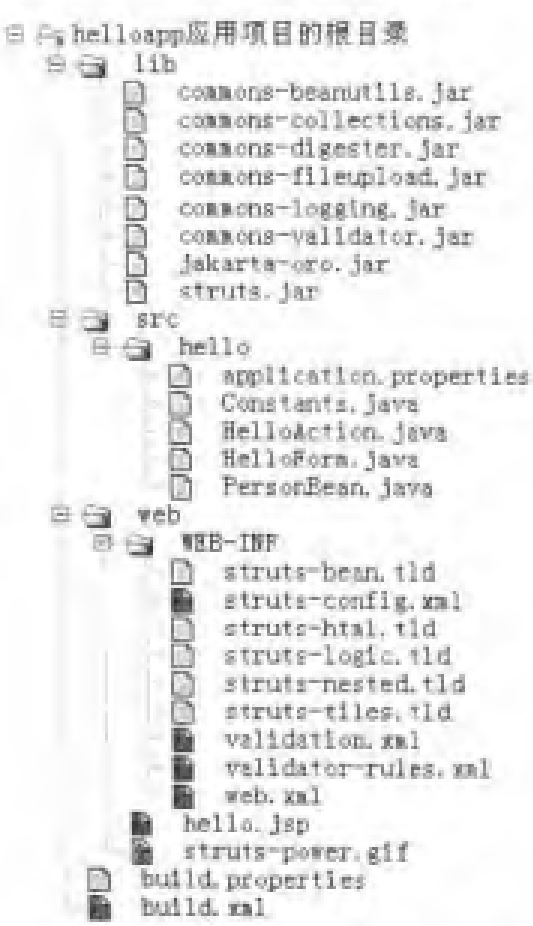

图 20-2　helloapp 应用原有的文件目录展开图

在 helloapp 应用的根目录下提供了 build.xml 文件。例程 20-1 是 build.xml 的代码。

例程 20-1　build.xml 文件

```
<project name="helloapp" default="help" basedir=".">

<!-- ========== Property Definitions ========== -->

  <!--
      All properties should be defined in this section.
      Any host-specific properties should be defined
      in the build.properties file.

      In this app, the following properties are defined in build.properties:

    o  tomcat.home      - the home directory of your Tomcat installation
    o  webapps.home     - the place to copy the war file to deploy it
  -->

  <property file="build.properties" />
```

```
    <property name="app.home"          value="." />
    <property name="app.name"          value="helloapp" />
    <property name="javadoc.pkg.top"   value="hello" />

    <property name="src.home"          value="${app.home}/src"/>
    <property name="lib.home"          value="${app.home}/lib"/>
    <property name="classes.home"      value="${app.home}/classes"/>
    <property name="deploy.home"       value="${app.home}/deploy"/>
    <property name="doc.home"          value="${app.home}/doc"/>
    <property name="web.home"          value="${app.home}/web"/>

    <property name="build.home"        value="${app.home}/build"/>
    <property name="build.classes"     value="${build.home}/WEB-INF/classes"/>
    <property name="build.lib"         value="${build.home}/WEB-INF/lib"/>

<!-- ================== Compilation Classpath =========================== -->
    <!--
        This section creates the classpath for compilation.
    -->

    <path id="compile.classpath">

      <!-- The object files for this application -->
      <pathelement location="${classes.home}"/>

      <!-- The lib files for this application -->
      <fileset dir="${lib.home}">
        <include name="*.jar"/>
        <include name="*.zip"/>
      </fileset>

      <!-- All files/jars that Tomcat makes available -->
      <fileset dir="${tomcat.home}/common/lib">
        <include name="*.jar"/>
      </fileset>
      <pathelement location="${tomcat.home}/common/classes"/>

    </path>

<!-- ================== Build Targets below here========================== -->

<!-- ================== "help" Target ==================================== -->
    <!--
        This is the default ant target executed if no target is specified.
        This helps avoid users just typing 'ant' and running a
        default target that may not do what they are anticipating...
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```

```
    -->

  <target name="help" >
    <echo message="Please specify a target! [usage: ant &lt;targetname&gt;]" />
    <echo message="Here is a list of possible targets: "/>
    <echo message="   clean-all.....Delete build dir, all .class and war files"/>
    <echo message="   prepare.....Creates directories if required" />
    <echo message="   compile.....Compiles source files" />
    <echo message="   build.....Build war file from .class and other files"/>
    <echo message="   deploy.....Copy war file to the webapps directory" />
    <echo message="   javadoc.....Generates javadoc for this application" />
  </target>

<!-- ==================== "clean-all" Target ============================== -->
    <!--
            This target should clean up any traces of the application
            so that if you run a new build directly after cleaning, all
            files will be replaced with what's current in source control
    -->

  <target name="clean-all" >
      <delete dir="${build.home}"/>
      <delete dir="${classes.home}"/>
      <delete dir="${deploy.home}"/>

      <!-- can't delete directory if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}" failonerror="false"/>

      <!-- deleting the deployed .war file is fine even if Tomcat is running -->
      <delete dir="${webapps.home}/${app.name}.war" />

      <!-- delete the javadoc -->
      <delete dir="${doc.home}"/>

  </target>

<!-- ==================== "prepare" Target ================================ -->
      <!--
            This target is executed prior to any of the later targets
            to make sure the directories exist. It only creates them
            if they need to be created....
            Other, similar, preparation steps can be placed here.
      -->

  <target name="prepare">
```