

人才储备计划

**J2EE Web 应用开发技术教材**

创智和宇信息系统有限公司  
业务拓展事业部

2006 年 6 月



## 关于本课程 ..... 1

课程描述 .....	1
听众 .....	1
学员应具备的条件 .....	1
课程目标 .....	1
历史版本 .....	1

## 第 1 章 J2EE 的基本概念和规范 ..... 2

1.1. J2EE 定义 .....	2
1.2. J2EE 规范构成 .....	3
1.3. J2EE 体系结构 .....	3
1.3.1. Java 语言系统 .....	3
1.3.2. 客户端程序设计模型 .....	4
1.3.3. 中间层基础结构 .....	4
1.3.4. 程序员企业级 API .....	4
1.4. J2EE N-TIERS 结构 .....	4
1.5. C/S 与 B/S 结构区别 .....	6
1.6. 小结 .....	7

## 第 2 章 WEB 服务器和 WEB 应用服务器 ..... 8

2.1. WEB 服务器(WEB SERVER) .....	8
2.2. 应用程序服务器(THE APPLICATION SERVER) .....	8
2.3. 一个例子 .....	9
2.3.1. 情景 1：不带应用程序服务器的 Web 服务器 .....	9
2.3.2. 情景 2：带应用程序服务器的 Web 服务器 .....	9
2.3.3. 附加说明 .....	9
2.4. 应用程序服务器(THE APPLICATION SERVER) .....	10
2.4.1. BEA Weblogic .....	10
2.4.2. IBM Websphere .....	10

2.4.3.     JBoss.....	10
2.4.4.     Tomcat .....	10
2.5.     小结 .....	11
<b>第3章 利用TOMCAT创建和发布WEB应用.....</b>	<b>12</b>
3.1.    TOMCAT与SERVLET容器.....	13
3.2.    TOMCAT结构 .....	13
3.2.1.    server.xml文件属性.....	15
3.3.    JAVA WEB应用简介 .....	16
3.4.    TOMCAT的工作模式.....	17
3.4.1.    独立的Servlet 容器.....	17
3.4.2.    进程内的Servlet 容器.....	17
3.4.3.    进程外的Servlet 容器.....	18
3.5.    TOMCAT的版本.....	18
3.6.    TOMCAT安装配置.....	19
3.6.1.    j2sdk的安装.....	19
3.6.2.    使用可执行文件安装Tomcat.....	20
3.6.3.    解压安装Tomcat .....	20
3.7.    基本的安装测试 .....	20
3.8.    TOMCAT ADMIN的安装.....	22
3.9.    TOMCAT启动分析.....	23
3.10.   TOMCAT目录结构.....	24
3.11.   创建和发布WEB应用.....	24
3.11.1.   Web应用的目录结构.....	24
3.11.2.   Eclipse +MyEclipse 开发helloapp的Web应用.....	25
3.12.   手工部署开发式目录结构 .....	38
3.12.1.   在server.xml文件中加入<Context>元素.....	38
3.13.   利用TOMCAT配置虚拟主机.....	38
3.14.   小结 .....	39
<b>第4章 JAVA SERVLET .....</b>	<b>40</b>
4.1.    第一个SERVLET .....	40
4.1.1.   基本结构.....	40
4.1.2.   输出纯文本的Servlet.....	44
4.1.3.   Servlet的编译和安装.....	45
4.2.    处理表单数据 .....	49
4.2.1.   表单数据概述.....	49
4.2.2.   实例: 读取表单变量.....	50
4.2.3.   实例: 输出所有表单数据.....	52
4.2.4.   实例: 测试表单.....	54
4.3.    读取HTTP请求 .....	56
4.3.1.   http请求头概述.....	56
4.3.2.   实例: 在Servlet中读取请求头.....	57
4.3.3.   实例: 输出所有的请求头.....	57
4.4.    访问CGI变量 .....	59

4.4.1.	<i>CGI</i> 概述.....	59
4.4.2.	标准 <i>CGI</i> 变量的 <i>Servlet</i> 等价表示.....	60
4.4.3.	实例：读取 <i>CGI</i> 变量.....	61
4.5.	HTTP应答状态 .....	63
4.5.1.	状态代码概述.....	63
4.5.2.	设置状态代码.....	64
4.5.3.	<i>HTTP</i> 状态代码及其含义.....	64
4.5.4.	实例：访问多个搜索引擎.....	67
4.6.	设置HTTP应答头 .....	70
4.6.1.	<i>HTTP</i> 应答头概述.....	70
4.6.2.	常见应答头及其含义.....	71
4.6.3.	实例：定时刷新页面的例子.....	72
4.7.	处理COOKIE.....	74
4.7.1.	<i>Cookie</i> 概述.....	74
4.7.2.	<i>Servlet</i> 的 <i>Cookie API</i> .....	74
4.7.3.	几个 <i>Cookie</i> 工具函数.....	76
4.8.	会话状态 .....	79
4.8.1.	会话状态概述.....	79
4.8.2.	会话状态跟踪API.....	80
4.8.3.	查看当前请求的会话对象.....	80
4.8.4.	查看和会话有关的信息.....	80
4.8.5.	在会话对象中保存数据.....	81
4.8.6.	实例：显示会话信息.....	82
4.9.	小结 .....	84
<b>第5章 JAVA SERVER PAGE.....</b>		<b>85</b>
5.1.	概述 .....	85
5.2.	JSP和其他类似技术比较 .....	86
5.3.	JSP语法概要表 .....	86
5.4.	脚本元素、指令和预定义变量.....	88
5.4.1.	<i>JSP</i> 脚本元素.....	88
5.4.2.	<i>JSP</i> 指令.....	89
5.4.3.	实例：脚本元素和指令的应用.....	91
5.5.	JSP预定义变量 .....	92
5.5.1.	<i>HttpServletRequest</i> 类的 <i>Request</i> 对象.....	92
5.5.2.	<i>HttpServletResponse</i> 类的 <i>Response</i> 对象.....	92
5.5.3.	<i>JspWriter</i> 类的 <i>out</i> 对象.....	92
5.5.4.	<i>HttpSession</i> 类的 <i>session</i> 对象.....	93
5.5.5.	<i>ServletContext</i> 类的 <i>application</i> 对象.....	93
5.5.6.	<i>ServletConfig</i> 类的 <i>Config</i> 对象.....	93
5.5.7.	<i>PageContext</i> 类的 <i>PageContext</i> 对象.....	93
5.5.8.	<i>Object</i> 类的 <i>Page</i> （相当于 <i>this</i> ）对象 .....	93
5.5.9.	<i>exception</i> .....	93
5.6.	JSP动作 .....	95
5.6.1.	<i>jsp:include</i> 动作.....	96

5.6.2.	<i>jsp:useBean</i> 动作 .....	97
5.6.3.	关于 <i>jsp:useBean</i> 的进一步说明.....	99
5.6.4.	<i>jsp:setProperty</i> 动作.....	100
5.6.5.	<i>jsp:getProperty</i> 动作.....	103
5.6.6.	<i>jsp:forward</i> 动作.....	104
5.6.7.	<i>jsp:plugin</i> 动作.....	104
5.6.8.	附录: JSP注释和字符引用约定.....	104
5.7.	JSP/SERVLET应用程序优化 .....	104
5.7.1.	在 <i>HttpServlet init()</i> 方法中缓存数据.....	104
5.7.2.	禁用 <i>servlet</i> 和 <i>Jsp</i> 的自动装载功能.....	105
5.7.3.	控制 <i>HttpSession</i> .....	105
5.7.4.	使用 <i>gzip</i> 压缩.....	106
5.7.5.	不要使用 <i>SingleThreadModel</i> .....	106
5.7.6.	使用线程池.....	107
5.7.7.	选择正确的包括机制.....	107
5.7.8.	在 <i>useBean</i> 动作中使用合适的范围.....	107
5.7.9.	其他技术.....	107
5.8.	实例: BOOKSTORE应用 .....	108
5.8.1.	Web服务器层.....	108
5.8.2.	数据库层.....	108
5.8.3.	JavaBean 和实用类.....	122
5.8.4.	发布bookstore应用 .....	132
5.9.	小结 .....	132
<b>第 6 章 JDBC技术.....</b>		<b>133</b>
6.1.	概述 .....	133
6.2.	JDBC的设计目的.....	133
6.3.	JDBC的主要功能.....	133
6.4.	与ODBC相比JDBC特点.....	135
6.5.	JDBC结构.....	135
6.6.	JDBC工作原理.....	136
6.7.	数据库应用的模型 .....	138
6.8.	通过JDBC 实现对数据库的访问 .....	139
6.8.1.	编写访问数据库程序的步骤.....	139
6.8.2.	实现对数据库的一般查询Statement.....	141
6.8.3.	预编译方式执行SQL语句PreparedStatement.....	143
6.8.4.	执行存储过程CallableStatement.....	143
6.8.5.	ResultSet对象 .....	143
6.8.6.	数据转换.....	144
6.8.7.	对应JAVA 与SQL类型 .....	144
6.8.8.	NULL结果值.....	147
6.8.9.	获得结果集中的结构信息.....	147
6.9.	更新数据库 .....	148
6.9.1.	对表中的记录进行操作.....	148
6.9.2.	创建和删除表.....	150

6.9.3. 增加和删除表中的列.....	150
6.9.4. 利用PreparedStatement对象实现数据更新.....	151
6.10. 参数的输入与输出 .....	151
6.11. 事务处理 .....	152
6.12. 批量处理JDBC语句提高处理速度 .....	154
6.13. 处理中文编码问题 .....	155
6.14. 通过JDBC访问数据库的JSP范例程序 .....	157
6.15. 在BOOKSTORE应用中通过JDBC访问数据库.....	159
6.16. 数据源DATASOURCE .....	164
6.16.1. 数据源和连接池.....	165
6.16.2. 数据源和JNDI资源.....	165
6.17. 在TOMCAT中配置数据源.....	166
6.17.1. server.xml 中加入<Resource>元素.....	166
6.17.2. 在应用程序的web.xml 中加入<resource-ref>元素.....	168
6.18. 程序中访问数据源 .....	168
6.18.1. 通过数据源访问数据库的JSP范例程序.....	168
6.18.2. 在bookstore应用中通过数据源访问数据库.....	170
6.19. 小结 .....	175

## 第7章 STRUTS与MVC设计模式 ..... 177

7.1. STRUTS与JAVA WEB应用 .....	177
7.1.1. Web组件的三种关联.....	177
7.1.2. MVC概述 .....	179
7.1.3. JSP Model1 和JSP Model2 .....	181
7.1.4. Struts 概述.....	182
7.2. 从HELLOAPP开始STRUTS应用 .....	187
7.2.1. helloapp的需求.....	187
7.2.2. 组建Struts框架.....	187
7.2.3. 创建工程.....	187
7.2.4. 创建视图组件.....	189
7.2.5. 创建控制器组件.....	196
7.2.6. 创建模型组件.....	199
7.2.7. 配置文件.....	199
7.3. 配置STRUTS组件 .....	201
7.3.1. Web 应用部署描述符.....	202
7.3.2. ActionServlet 的参数.....	204
7.3.3. Struts配置.....	207
7.3.4. Struts 配置元素.....	209
7.3.5. Struts config骨架.....	215
7.3.6. 应用资源文件.....	216
7.3.7. 配置Struts 核心.....	217
7.3.8. 配置模块化应用.....	219
7.4. STRUTS控制器组件 .....	220
7.4.1. Action类.....	221
7.4.2. RequestProcessor类.....	222

7.4.3. <i>action</i> 类.....	223
7.4.4.    使用内置的struts <i>action</i> 类.....	225
7.4.5.    利用 <i>token</i> 解决重复提交 .....	226
7.4.6.    实用类.....	227
5. STRUTS模型组件 .....	229
7.5.1.    模型在MVC中的地位.....	229
7.5.2.    模型的概念和类型.....	230
7.5.3.    业务对象(BO).....	230
7.5.4.    业务对象的持久化.....	231
7.5.5.    小结.....	233
6. STRUTS视图组件 .....	233
7.6.1.    视图概述.....	233
7.6.2.    在视图中使用JavaBean .....	234
7.6.3.    使用ActionForm.....	235
7. 扩展STRUTS框架 .....	240
7.7.1.    PlugIn.....	240
7.7.2.    请求是如何被处理的.....	242
7.7.3.    创建你自己的RequestProcessor .....	246
7.7.4.    ActionServlet .....	247
7.7.5.    结论.....	248
8. STRUTS应用国际化 .....	248
7.8.1.    本地化与国际化的概念.....	248
7.8.2.    Web应用的中文本地化 .....	248
7.8.3.    Java对I18N的支持.....	249
7.8.4.    Web容器中Locale 对象的来源.....	250
7.8.5.    在Web应用中访问Locale 对象.....	250
7.8.6.    在Struts应用中访问Locale 对象.....	250
7.8.7.    ResourceBundle 类.....	251
7.8.8.    MessageFormat类和符合消息.....	251
7.8.9.    Struts框架对国际化的支持.....	251
7.8.10.   异常处理的国际化.....	253
7.8.11.   小结.....	253
9. VALIDATOR验证框架 .....	254
10. 异常处理 .....	255
11. STRUTS标签 .....	258
7.11.1.   在Struts应用中使用标签库.....	259
7.11.2.   Struts HTML标签库.....	259
7.11.3.   Struts Bean标签库.....	261
7.11.4.   Struts Logic标签库.....	261
7.11.5.   Struts Template 标签库.....	263
7.11.6.   Struts Nested 标签库.....	263
12. 使用TILES框架 .....	263
7.12.1.   Tiles配置和基本配置文件介绍.....	264
7.12.2.   使用Tiles .....	267

7.12.3. 小结.....	267
7.13. 用STRUTSTESTCASE测试STRUTS应用 .....	268
7.13.1. 准备StrutsTestCase.....	268
7.13.2. 初试StrutsTestCase.....	271
7.13.3. 深入StrutsTestCase.....	272
7.13.4. 关于Web.xml和Struts-Config.xml.....	273
7.13.5. 小结.....	273
<b>第8章 HIBERNATE与JAVA对象持久化技术.....</b>	<b>274</b>
8.1. QUICK START.....	274
8.1.1. 准备工作.....	274
8.1.2. 构建Hibernate 基础代码.....	274
8.1.3. 由数据库产生基础代码.....	275
8.2. HIBERNATE 配置 .....	283
8.2.1. 准备.....	283
8.2.2. 第一段代码.....	284
8.2.3. Hibernate基础语义.....	287
8.3. HIBERNATE高级特性 .....	288
8.3.1. XDoclet 与Hibernate 映射.....	288
8.3.2. 数据检索.....	296
8.3.3. Hibernate Query Language (HQL) .....	299
8.4. 数据关联 .....	299
8.4.1. 一对关联.....	299
8.4.2. 一对多关联.....	302
8.4.3. 多对多关联.....	310
8.5. 数据访问 .....	314
8.5.1. PO和VO .....	314
8.5.2. 关于unsaved-value.....	317
8.5.3. Inverse 和Cascade .....	318
8.5.4. 延迟加载 (Lazy Loading) .....	318
8.5.5. 事务管理.....	321
8.5.6. 基于JDBC的事务管理.....	322
8.5.7. 基于JTA的事务管理.....	322
8.5.8. 锁 (locking) .....	325
8.5.9. 悲观锁 (Pessimistic Locking) .....	325
8.5.10. 乐观锁 (Optimistic Locking) .....	326
8.5.11. Hibernate分页.....	329
8.5.12. Cache 管理.....	331
8.5.13. Session管理.....	334
<b>第9章 轻量级框架SPRING .....</b>	<b>338</b>
9.1. SPRING概述 .....	338
9.1.1. Spring是什么? .....	338
9.1.2. Spring项目 .....	339
9.1.3. 入门指引.....	339

9.1.4. Putting a Spring into Hello World: Spring之Hello World .....	344
9.2. SPRING基础 .....	351
9.2.1. Introducing Inversion of Control: 控制反转介绍.....	351
9.2.2. Types of Inversion of Control: 控制反转类型.....	351
9.2.3. Inversion of Control in Spring: Spring中的控制反转.....	353
9.2.4. Dependency Injection with Spring: 使用Spring依赖注入 .....	353
9.2.5. Configuring the BeanFactory: 配置Bean工厂 (BeanFactory) .....	356
9.2.6. Injection Parameters: 注入参数.....	360
9.2.7. Understanding Bean Naming: 理解Bean命名.....	370
9.2.8. Bean Instantiation Modes: Bean实例化模式.....	372
9.2.9. Resolving Dependency: 依赖解析.....	374
9.2.10. Auto-Wiring Your Beans: 自动装配你的Bean .....	375
9.2.11. Checking Dependency: 依赖检查 .....	378
9.2.12. Bean Inheritance: Bean继承 .....	380
9.2.13. 小结.....	381
9.3. ASPECT ORIENTED PROGRAMMING WITH SPRING: 基于SPRING的AOP编程 .....	382
9.3.1. AOP Concepts: AOP概念.....	382
9.3.2. Types of AOP: AOP的种类.....	383
9.3.3. AOP in Spring: Spring 中的AOP .....	383
9.3.4. Advisors and Pointcuts in Spring: Spring 里的通知者和切入点.....	401
9.3.5. All About Proxies: 关于代理.....	413
9.3.6. Advanced Use of Pointcuts: 切入点的高级使用.....	418
9.3.7. Getting Started with Introductions: 引入初步.....	419
9.3.8. Framework Services for AOP: AOP框架服务.....	419
9.3.9. 小结.....	419
9.4. 基于SPRING的数据访问 .....	420
9.4.1. Exploring the JDBC Infrastructure: 探索JDBC的底层结构.....	420
9.4.2. Spring JDBC Infrastructure: Spring JDBC底层机制 .....	420
9.4.3. Database Connection and DataSources: 数据库连接和数据源 .....	420
9.4.4. Using DataSources in DAO Classes: 在DAO类中使用数据源.....	420
9.4.5. Exception Handling: 异常处理.....	420
9.4.6. The JDBC Template Class: JDBC模板类 .....	420
9.4.7. Selecting The Data as Java Objects: 选出作为Java对象的数据.....	420
9.4.8. Updating Data: 更新数据.....	420
9.4.9. Inserting Data: 插入数据.....	420
9.4.10. Transactions: 事务 .....	421
9.4.11. Why JDBC: 为什么用JDBC.....	421
9.4.12. Using JDBC Data Access in the Sample Application: 在示例程序中使用JDBC数据访问	
421	
9.4.13. 小结.....	421
9.5. 在SPRING应用中使用HIBERNATE .....	421
9.5.1. What Is Hibernate: 什么是Hibernate.....	421
9.5.2. The Hibernate Query Language: Hibernate查询语言.....	421
9.5.3. Selecting Data: 选择数据 .....	421

9.5.4.	<i>Updating and Inserting Data: 更新并插入数据</i> .....	421
9.5.5.	<i>Using Hibernate in the Sample Application: 在示例应用中使用Hibernate</i> .....	421
9.5.6.	<i>小结</i> .....	421
9.6.	<i>基于SPRING的应用程序设计及实现</i> .....	422
9.6.1.	<i>Designing to Interfaces: 面向接口的设计</i> .....	422
9.6.2.	<i>Impact of Spring on Interface-Based Design: Spring 对于基于接口设计的冲击</i> .....	422
9.6.3.	<i>Building a Domain Object Model: 域对象模型的构建</i> .....	422
9.6.4.	<i>Designing and Building the Data Access Tier: 持久层设计与构建</i> .....	422
9.6.5.	<i>Designing the Business Tier: 业务层设计</i> .....	423
9.6.6.	<i>小结</i> .....	423
9.7.	<i>基于SPRING的WEB应用开发</i> .....	423
9.7.1.	<i>Introducing Spring MVC: 介绍Spring MVC</i> .....	423
9.7.2.	<i>Using Handler Mappings: 使用Handler Mappings</i> .....	423
9.7.3.	<i>Using Handler Interceptors: 使用处理拦截器</i> .....	423
9.7.4.	<i>Woking with Controllers: 让控制器工作起来</i> .....	423
9.7.5.	<i>Views, Locales, and Themes: 视图, Locales, 主题</i> .....	424
9.7.6.	<i>Using Command Controllers: 使用命令控制器</i> .....	424
9.7.7.	<i>Using Spring MVC in the Sample Application: 在范例应用种使用Spring MVC</i> .....	424
9.7.8.	<i>使用Spring 中JSP标记</i> .....	424
9.7.9.	<i>Using Velocity: 使用Velocity</i> .....	425
9.7.10.	<i>Using XSLT Views: 使用XSLT 视图</i> .....	425
9.7.11.	<i>Using PDF Views: 使用PDF 视图</i> .....	425
9.7.12.	<i>Using Excel Views: 使用Excel 视图</i> .....	425
9.7.13.	<i>Using Tiles: 使用Tiles</i> .....	425
9.8.	<i>SPRING和STRUTS</i> .....	425
9.8.1.	<i>Exploring the Struts Architecture: 考察Struts 体系结构</i> .....	425
9.8.2.	<i>A Struts Application: 一个Sturts 程序</i> .....	425
9.8.3.	<i>Accessing Spring Beans: 访问Spring Bean</i> .....	425
9.8.4.	<i>Using Other Views: 使用第三方Views</i> .....	425
9.8.5.	<i>Using Struts Actions as Spring Beans: 将Struts Action 定义为Spring Bean</i> .....	426
9.8.6.	<i>Combining Struts and Spring MVC: 整合Struts 和Spring MVC</i> .....	426
9.8.7.	<i>小结</i> .....	426

# 关于本课程

这部分给出了一个简要的课程描述、听众对象，建议的前提条件以及课程的目标。

## 课程描述

本课程给出 J2EE 平台技术概念上的一格较为清晰而完整的思路，帮助学生掌握各技术间的相互关系和重要的思想。通过一系列的实战训练，使学生掌握基于 J2EE 平台的 Java 语言高级编程技术。本课程偏重于 J2EE WEB 应用开发，旨在培养学员的开发技能，使学员能成为一个合格的 J2EE 软件开发工程师。

## 听众

本课程是为以下学员设置的：有一定的 JAVA 语言基础，了解关系型数据库，希望提高 JAVA 开发技能，使用 J2EE 开发应用系统。

## 学员应具备的条件

本课程要求学员具备以下条件：

- ✓ 成功完成 JAVA 基础培训课程，或者具备同等的技能和知识；
- ✓ 了解关系型数据库，并且具备数据库基础知识，能熟练使用 SQL 语言；
- ✓ 有软件工程概念，并希望能通过实践来验证软件工程方法；
- ✓ 了解 HTML 标记语言，希望提高自身开发技能，使用 J2EE 开发 WEB 应用系统；

## 课程目标

本课程结束时，学员能够：

- 了解当前主流 J2EE 开发系统总体架构；
- 掌握常用系统的安装、配置和维护，最终能达真正掌握到三层（N 层）结构软件开发；
- 掌握 J2EE 领域相关知识，学会利用 Struts、Hibernate、Spring 等现今主流开源框架开发 J2EE 应用系统；

## 历史版本

修订时间	修订者	修订内容
2006 年 6 月 24 日	谢平	创建文档
2006 年 8 月 4 日	谢平	校对并调整格式
2006 年 10 月 14 日	谢平	发布 v1.0
2007 年 2 月 5 日	谢平	开始修订 v1.1
2007 年 3 月 8 日	谢平	修订一些代码错误

# 第1章 J2EE 的基本概念和规范

## 1.1. J2EE 定义

J2EE 是 Sun 公司提出的多层(multi-tiered), 分布式(distributed), 基于组件(component-base)的企业级应用模型(enterprise application model)。在这样的一个应用系统中, 可按照功能划分为不同的组件, 这些组件又可在不同计算机上, 并且处于相应的层次(tier)中。所属层次包括客户层(client tier)组件, web 层和组件, Business 层和组件, 企业信息系统(EIS)层。

J2EE 平台由一整套服务 (Services)、应用程序接口 (APIs) 和协议构成, 它对开发基于 Web 的多层应用提供了功能支持。J2EE 主要由以下部分组成:

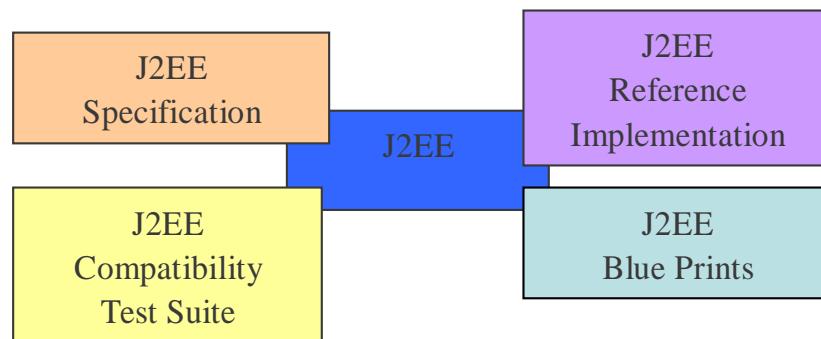


图 1-1: J2EE 组成

**J2EE 规范:** 定义 J2EE 的规范, 开发商按照这个规范实现自己的 J2EE 系统。

**J2EE 参考实现:** J2EE 平台的软件开发包 (J2EE SDK); J2EE 的完全实现; 一组工具和运行环境; 完全免费得到 ([java.sun.com/j2ee](http://java.sun.com/j2ee))。

**J2EE 兼容测试包:** 用来测试一种应用是否符合 J2EE 的规范

**J2EE 蓝图:** 提供了用于实施基于 J2EE 的多层应用的文档和实例套件的编程模型, 简化了开发基于 J2EE 的多层应用程序的复杂性。它应被用作开发人员设计和优化组件的原则, 以便从策略上对开发工作进行分工, 分配技术资源。

## 1.2. J2EE 规范构成

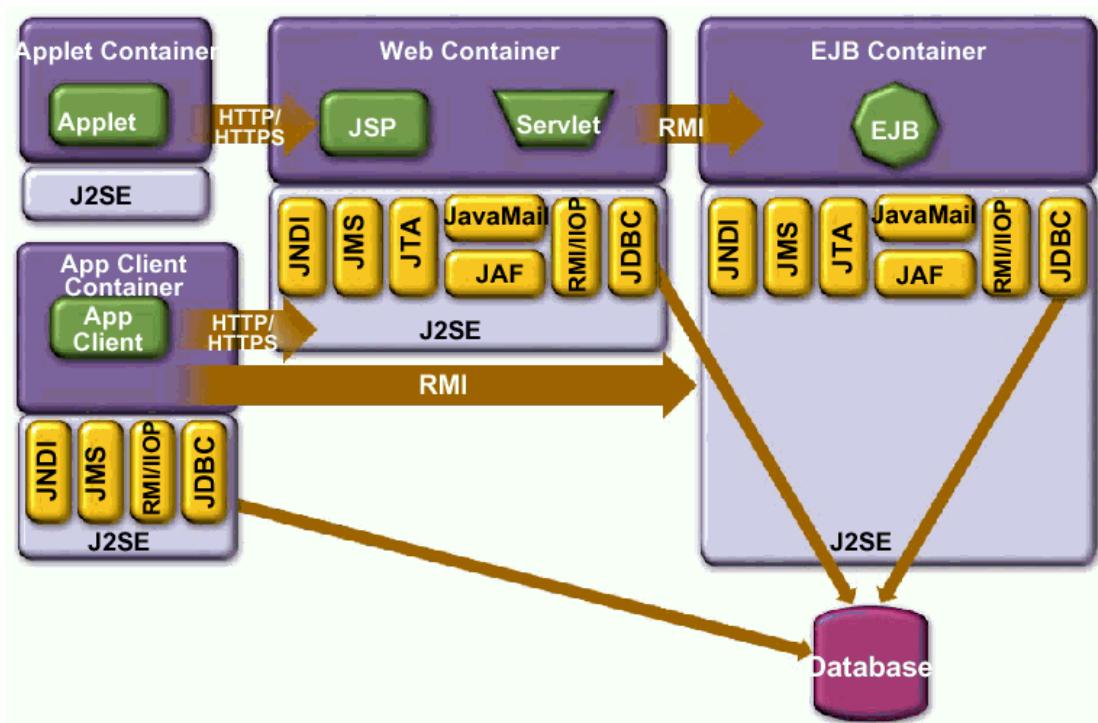


图 1-2: J2EE 规范构成

构成 J2EE 规范的主要核心技术有： JDBC,JNDI,EJBs,RMI,JSP,Java Servlets,XML,JMS,Java IDL,JTS,JTA,JavaMail,JAF 等。

## 1.3. J2EE 体系结构

J2EE 体系结构可以被分为 5 部分：

- Java 语言系统
- 客户端程序设计模型
- 中间层基础结构
- 程序员企业级 API
- 非程序员可见 API

最后一部分，非程序员可见 API，包括定义了如何将其他产品插入到 J2EE 中的 API，如连接器 API，以及 J2EE 模型中被最近的改进有效替代的 API，如 JTA（Java Transaction API）。

### 1.3.1. Java 语言系统

在高层次上，Java 语言系统看起来与.NET Framework 类似。在这两种情况中，源代码都是被翻译成一种中间语言。但是，在.NET 平台中，这种中间语言是 MSIL，而在 Java 系统中，是 Java Byte Code。在这两种情况中，中间语言被带入到运行环境中。在 Framework 中，运行环境是 Common Language Runtime。对于 Java，运行环境是 Java 虚拟机（Java Virtual Machine）。总体而言，Common Language Runtime 和 Java 虚拟机有类似的功能，并且在技术进步方面，都无可置疑地在发展和彼此交互跃进。

这两种系统之间最重要的区别与源代码到中间语言的翻译有关。在.NET 平台中，中间语言设计用来适应各种语言的需求。在 Java 中，中间语言设计用来满足 Java 的需求。虽然

从理论上，从除 Java 外的语言生成 Java Byte Code 是可能的，但是实际上这还没有在任何一种商业产品中证明。

### 1.3.2. 客户端程序设计模型

J2EE 客户端程序设计模型重点集中在与浏览器的交互上。客户端程序设计模型有 3 部分：Java Applets，Java Servlets 和 Java Server Pages。

Java Applets 用来对在浏览器内运行的 Java 代码进行打包。

处理 HTTP 请求和 HTML 响应的重要技术是 Java Servlets 和 Java Server Pages 。

### 1.3.3. 中间层基础结构

对于 J2EE，中间层基础结构是 Enterprise Java Beans (EJB)。该规范的当前版本是 3.0，可以从网上获得。重要想法包括：

- 通过组件示例的共享所实现的高可伸缩性
- 以中间层为中心的安全性
- 自动事务处理边界管理

EJB 加入了一种新的体系结构想法，一项自动管理组件状态的技术。这项技术被称为 entity beans (实体 BEAN)。虽然这种想法具有吸引力，但是当前的实施却依赖于独立于数据库缓存的中间层数据缓存。很不幸的是，在这两种缓存之间没有保持一致性的机制。这意味着对实体 BEAN 的任何使用都会带来数据库损坏的高风险。在缓存一致性问题解决之前，在最佳试验技术方面，必须得不断考虑实体 BEAN 技术。

### 1.3.4. 程序员企业级 API

我们调用 Java Enterprise API 时的最重要部分如下：

- Java Database Connection (JDBC, Java 数据库连接) 2.0 - 是用于从 Java 中访问关系型数据库的 API 。这与.NET 平台空间中的 ADO.NET 相当。
- Java Naming and Directory Interface (JNDI, Java 命名与目录接口) - 是用于从 Java 中访问企业名称与目录服务的信息的 API 。这与.NET 平台空间中的 Active Directory Services Interface (ADSI, 活动目录服务接口)有点类似。
- Java Message Service (JMS, Java 消息服务) 1.0 - 是用于异步工作流的 Java API 。这在功能上与 Microsoft Message Queue API 相当，这个 API 已经被排队组件所替代。

## 1.4. J2EE n-tiers 结构

早期的网络系统设计常常采用三层结构。最常见的结构，就是表示 ( presentation ) 层，领域 ( domain ) 层，以及基础架构 ( infrastructure ) 层。

n 层结构的提出是为了适应当前 B/S 模式开发 WEB Application 的需要而提出的。传统的 Brown 模型是指：表示层 ( Presentation )，控制 / 中介层 ( Controller/Mediator )，领域层 ( Domain )，数据映射层 ( Data Mapping )，和数据源层 ( Data Source )。它其实就是在三层架构中增加了两个中间层。控制 / 中介层位于表示层和领域层之间，数据映射层位于领域层和基础架构层之间。

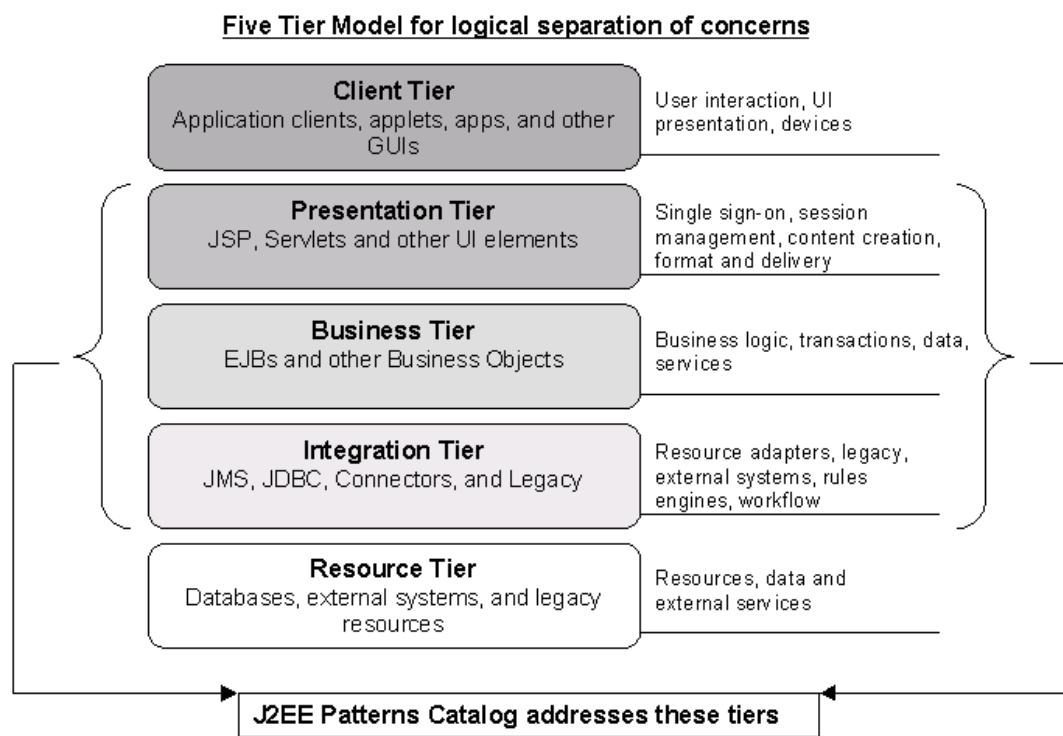


图 1-3: J2EE n-tiers

下表是几种模型的对照:

ISA	Brown	J2EE 层	对应程序部分
表示层	表示层	客户层	浏览器 HTML 页面, XSL, 手机客户端等
		服务器表示层	JSP 及 ActionForm, XML
	控制/中介层		Controller 控制器及 Action
领域层	领域层	业务层	Javabeans/SessionBean/Session Facade
	数据映射层	整合层	EntityBean/JDO/Hibernate/JDBC
数据层	数据源层	资源层	RDBMS 数据库

J2EE 的基本原则之一，是使得各个层的实现解除耦合或耦合最小化。最终实现可以任意的切换某个层的实现。

例如，在数据映射层，可以采用 EJB 的 BMP, CMP，也可以采用 Hibernate 等 O/RMapping，或者采用 JDO。这由部署的环境来决定。

N 层架构的核心是提供可规模化特性，一方面是从服务负载上可规模化，能同时为极大的规模的用户同时提供服务；另一方面是服务功能上的可规模化，可形成极大的规模的软件群系统，各分系统可以共享信息、服务，形成企业级的信息高速公路。N 层可以分别放在各自不同的硬件系统上的，所以灵活性很高，能够适应客户机数目的增加和处理负荷的变动。例如，在追加新业务处理时，可以相应增加装载功能层的服务器。因此，系统规模越大这种形态的优点就越显著。

另外，N 层结构从逻辑上相互独立，某一层的变动通常不影响其它层，具有很高的可重用性。除此以外，n 层结构还有以下优点：

- 利用单一的访问点，可以在任何地方访问站点的数据库；
- 对于各种信息源，不论是文本还是图形都采用相同的界面；
- 所有的信息，不论其基于的平台，都可以用相同的界面访问；
- 可跨平台操作；

5. 减少整个系统的成本;
6. 维护升级十分方便;
7. 具有良好的开放性;
8. 进行严密的安全管理;
9. 系统管理简单, 可支持异种数据库, 有很高的可用性

典型的三层结构:

界面层	收集信息 将信息发送到逻辑层做处理 逻辑层接收处理结果 将结果显示 .....
逻辑层	接收界面输入 与数据层交互执行已设计的业务 操作(业务逻辑, 系统服务等) 将处理结果发送到界面层 .....
数据层	数据存储 数据获取 数据维护 数据完整性 .....

## 1.5. C/S 与 B/S 结构区别

内容	C/S 结构	B/S 结构
<u>硬件环境</u>	一般建立在专用的网络上, 小范围里的网络环境, , 局域网之间再通过专门服务器提供连接和数据交换服务。	建立在广域网之上的, 不必是专门的网络硬件环境, 例与电话上网, 租用设备。信息自己管理。有比 C/S 更强的适应范围。一般只要有操作系统和浏览器就行。
<u>安全要求</u>	一般面向相对固定的用户群, 对信息安全的控制能力很强。一般高度机密的信息系统采用 C/S 结构适宜。可以通过 B/S 发布部分可公开信息。	建立在广域网之上。对安全的控制能力相对弱, 可能面向不可知的用户。
<u>程序架构</u>	程序可以更加注重流程, 可以对权限多层次校验, 对系统运行速度可以较少考虑。	对安全以及访问速度的多重的考虑, 建立在需要更加优化的基础之上, 比 C/S 有更高的要求 B/S 结构的程序架构是发展的趋势, 目前更加趋向成熟。
<u>软件重用</u>	程序可以不可避免的整体性考虑, 构件的重用性不如在 B/S 要求下的构件的重用性好。	对的多重结构, 要求构件相对独立的功能。能够相对较好的重用。
<u>系统维护</u>	程序由于整体性, 必须整体考察, 处理出现的问题以及系统升级, 升级难, 可能是再做一个全新的系统。	构件组成, 方便构件个别的更换, 实现系统的无缝升级。系统维护开销减到最小。用户从网上自己下载安装就可以实现升级。

<u>处理问题</u>	程序可以处理用户面固定，并且在相同区域，安全要求高需求，与操作系统相关，应该都是相同的系统。	建立在广域网上，面向不同的用户群，分散地域，这是 C/S 无法做到的。与操作系统平台关系最小。
<u>用户接口</u>	多是建立的 Window 平台上，表现方法有限，对程序员普遍要求较高。	建立在浏览器上，有更加丰富和生动的表现方式与用户交流。并且大部分难度减低，减低开发成本。
<u>信息流</u>	程序一般是典型的中央集权的机械式处理，交互性相对低。	信息流向可变化，B-B B-C 等信息、流向的变化，更像交易中心。

**说明：**

B-B：贸易二者间的商务活动；是指在企业对企业或商业机构对商业机构之间，通过 Internet 或专用网方式进行电子商务活动。

B-C：贸易与个人之间的商务活动；是指企业通过 Internet 为消费者提供一个新型的购物环境—网上商店；消费者通过互联网在网上购物，在网上支付。

## 1.6. 小结

在本章里面，我们介绍了 J2EE 的定义，J2EE 规范的构成，J2EE 体系结构，n 层架构的内容，C/S 与 B/S 结构的区别，为后面的学习打下铺垫。

# 第2章 Web 服务器和 Web 应用服务器

通俗的讲，Web 服务器传送(serves)页面使浏览器可以浏览，然而应用程序服务器提供的是客户端应用程序可以调用(call)的方法(methods)。确切一点，你可以说：Web 服务器专门处理 HTTP 请求(request)，但是应用程序服务器是通过很多协议来为应用程序提供(serves)商业逻辑(business logic)。

## 2.1. Web 服务器(Web Server)

Web 服务器可以解析(handles)HTTP 协议。当 Web 服务器接收到一个 HTTP 请求(request)，会返回一个 HTTP 响应(response)，例如送回一个 HTML 页面。为了处理一个请求(request)，Web 服务器可以响应(response)一个静态页面或图片，进行页面跳转(redirect)，或者把动态响应(dynamic response)的产生委托(delegate)给一些其它的程序例如 CGI 脚本，JSP(JavaServer Pages)脚本，servlets，ASP(Active Server Pages)脚本，服务器端(server-side)JavaScript，或者一些其它的服务器端(server-side)技术。无论它们(译者注：脚本)的目的如何，这些服务器端(server-side)的程序通常产生一个 HTML 的响应(response)来让浏览器可以浏览。

要知道，Web 服务器的代理模型(delegation model)非常简单。当一个请求(request)被送到 Web 服务器里来时，它只单纯的把请求(request)传递给可以很好的处理请求(request)的程序(译者注：服务器端脚本)。Web 服务器仅仅提供一个可以执行服务器端(server-side)程序和返回(程序所产生的)响应(response)的环境，而不会超出职能范围。服务器端(server-side)程序通常具有事务处理(transaction processing)，数据库连接(database connectivity)和消息(messaging)等功能。

虽然 Web 服务器不支持事务处理或数据库连接池，但它可以配置(employ)各种策略(strategies)来实现容错性(fault tolerance)和可扩展性=scalability)，例如负载平衡(load balancing)，缓冲(caching)。集群特征(clustering—features)经常被误认为仅仅是应用程序服务器专有的特征。

## 2.2. 应用程序服务器(The Application Server)

根据我们的定义，作为应用程序服务器，它通过各种协议，可以包括 HTTP，把商业逻辑暴露给(expose)客户端应用程序。Web 服务器主要是处理向浏览器发送 HTML 以供浏览，而应用程序服务器提供访问商业逻辑的途径以供客户端应用程序使用。应用程序使用此商业逻辑就象你调用对象的一个方法(或过程语言中的一个函数)一样。

应用程序服务器的客户端(包含有图形用户界面(GUI)的)可能会运行在一台 PC、一个 Web 服务器或者甚至是其它的应用程序服务器上。在应用程序服务器与其客户端之间来回穿梭(traveling)的信息不仅仅局限于简单的显示标记。相反，这种信息就是程序逻辑(program logic)。正是由于这种逻辑取得了(takes)数据和方法调用(calls)的形式而不是静态 HTML，所以客户端才可以随心所欲的使用这种被暴露的商业逻辑。

在大多数情形下，应用程序服务器是通过组件(component)的应用程序接口(API)把商业逻辑暴露(expose)(给客户端应用程序)的，例如基于 J2EE(Java 2 Platform, Enterprise Edition)应用程序服务器的 EJB(Enterprise JavaBean)组件模型。此外，应用程序服务器可以管理自己的资源，例如看大门的工作(gate-keeping duties)包括安全(security)，事务处理(transaction processing)，资源池(resource pooling)，和消息(messaging)。就象 Web 服务器一样，应用程序服务器配置了多种可扩展=scalability)和容错(fault tolerance)技术。

## 2.3.一个例子

例如，设想一个在线商店(网站)提供实时定价(real-time pricing)和有效性(availability)信息。这个站点(site)很可能提供一个表单(form)让你来选择产品。当你提交查询(query)后，网站会进行查找(lookup)并把结果内嵌在 HTML 页面中返回。网站可以有很多种方式来实现这种功能。我要介绍一个不使用应用程序服务器的情景和一个使用应用程序服务器的情景。观察以下这两中情景的不同会有助于你了解应用程序服务器的功能。

### 2.3.1. 情景 1：不带应用程序服务器的 Web 服务器

在此种情景下，一个 Web 服务器独立提供在线商店的功能。Web 服务器获得你的请求(request)，然后发送给服务器端(server-side)可以处理请求(request)的程序。此程序从数据库或文本文件(flat file，译者注：flat file 是指没有特殊格式的非二进制的文件，如 properties 和 XML 文件等)中查找定价信息。一旦找到，服务器端(server-side)程序把结果信息表示成(formulate)HTML 形式，最后 Web 服务器把它发送到你的 Web 浏览器。

简而言之，Web 服务器只是简单的通过响应(response)HTML 页面来处理 HTTP 请求(request)。

### 2.3.2. 情景 2：带应用程序服务器的 Web 服务器

情景 2 和情景 1 相同的是 Web 服务器还是把响应(response)的产生委托(delegates)给脚本(译者注：服务器端(server-side)程序)。然而，你可以把查找定价的商业逻辑(business logic)放到应用程序服务器上。由于这种变化，此脚本只是简单的调用应用程序服务器的查找服务(lookup service)，而不是已经知道如何查找数据然后表示为(formulate)一个响应(response)。这时当该脚本程序产生 HTML 响应(response)时就可以使用该服务的返回结果了。

在此情景中，应用程序服务器提供(serves)了用于查询产品的定价信息的商业逻辑。(服务器的)这种功能(functionality)没有指出有关显示和客户端如何使用此信息的细节，相反客户端和应用程序服务器只是来回传送数据。当有客户端调用应用程序服务器的查找服务(lookup service)时，此服务只是简单的查找并返回结果给客户端。

通过从响应产生(response-generating)HTML 的代码中分离出来，在应用程序之中该定价(查找)逻辑的可重用性更强了。其他的客户端，例如收款机，也可以调用同样的服务(service)来作为一个店员给客户结帐。相反，在情景 1 中的定价查找服务是不可重用的因为信息内嵌在 HTML 页中了。

总而言之，在情景 2 的模型中，在 Web 服务器通过回应 HTML 页面来处理 HTTP 请求(request)，而应用程序服务器则是通过处理定价和有效性(availability)请求(request)来提供应用程序逻辑的。

### 2.3.3. 附加说明

现在，XML Web Services 已经使应用程序服务器和 Web 服务器的界线混淆了。通过传送一个 XML 有效载荷(payload)给服务器，Web 服务器现在可以处理数据和响应(response)的能力与以前的应用程序服务器同样多了。

另外，现在大多数应用程序服务器也包含了 Web 服务器，这就意味着可以把 Web 服务器当作是应用程序服务器的一个子集(subset)。虽然应用程序服务器包含了 Web 服务器的功能，但是开发者很少把应用程序服务器部署(deploy)成这种功能(capacity)(译者注：这种功能是指既有应用程序服务器的功能又有 Web 服务器的功能)。相反，如果需要，他们通常会把 Web 服务器独立配置，和应用程序服务器一前一后。这种功能的分离有助于提高性能(简单

的 Web 请求(request)就不会影响应用程序服务器了), 分开配置(专门的 Web 服务器, 集群 (clustering)等等), 而且给最佳产品的选取留有余地。

## 2.4. 应用程序服务器(The Application Server)

### 2.4.1. BEA Weblogic

Weblogic 可以到 BEA 的网站上免费注册之后下载到最新 Weblogic 9.0 企业版, License 可以免费使用 1 年时间, 其实这已经完全足够了。Weblogic 的下载连接:

http://commerce.bea.com , Weblogic 的 在 线 文 档 :  
http://dev2dev.bea.com.cn/products/wlser/index.html 。

### 2.4.2. IBM Websphere

Websphere 同样可以下载到免费的试用版本, 到IBM 的developerWorks 网站可以看到 Websphere 试用产品的下载和相关的Websphere的资料, developerWorks中文网站的连接是: http://www-128.ibm.com/developerworks/cn/websphere/ , Websphere 的 下 载 连 接 : http://www-128.ibm.com/developerworks/websphere/downloads/index.html?S\_TACT=105AGX10&S\_CMP=HP 。

### 2.4.3. JBoss

JBoss 是免费开源的App Server, 同时它也是一个EJB容器, 可以免费的从JBoss 网站下载: http://www.jboss.com/downloads/index, 然而Jboss的文档是不免费, 需要花钱购买, 所以为我们学习JBoss 设置了一定的障碍。在Jdon上有几篇不错的JBoss 配置文档, 可以用来参考: http://www.jdon.com/idea.html

### 2.4.4. Tomcat

Tomcat 严格意义上并不是一个真正的 App Server, 它只是一个可以支持运行 Serlvet/JSP 的 Web 容器, 不过 Tomcat 也扩展了一些 App Server 的功能, 如 JNDI, 数据库连接池, 用户事务处理等等。Tomcat 被非常广泛的应用在中小规模的 Java Web 应用中, 因此本文做一点下载、安装和配置 Tomcat 的介绍: Tomcat 是 Apache 组织下 Jakarta 项目下的一个子项目, 它的主网站是: http://jakarta.apache.org/tomcat/ , Tomcat 最新版本是 Tomcat 5.0.28, 软件下载的连接是: http://tomcat.apache.org/download-55.cgi#5.0.28。

下载 Tomcat 既可以下载 zip 包, 也可以下载 exe 安装包(个人建议 zip 更干净些), 不管哪种情况, 下载完毕安装好以后 (zip 直接解压缩就可以了)。需要设置两个环境变量:

JAVA\_HOME=C:\j2sdk1.4.2

CATALINA\_HOME=D:\tomcat5 (你的 Tomcat 安装目录)

这样就安装好了, 启动 Tomcat 运行 CATALINA\_HOME\bin\startup.bat, 关闭 Tomcat 运行 shutdown.bat 脚本。Tomcat 启动以后, 默认使用 8080 端口, 因此可以用浏览器访问 http://localhost:8080 来测试 Tomcat 是否正常启动。

Tomcat 提供了两个 Web 界面的管理工具, URL 分别是:

http://localhost:8080/admin/index.jsp

http://localhost:8080/manager/html

在启用这两个管理工具之前, 先需要手工配置管理员用户和口令。用一个文本工具打开 CATALINA\_HOME\conf\tomcat-users.xml 这个文件, 加入如下几行:

```
<role rolename="manager"/>
```

```
<role rolename="admin"/>
<user username="robbin" password="12345678"
roles="admin,manager,tomcat"/>
```

这样用户“robbin”就具备了超级管理员权限。重新启动 Tomcat 以后，你就可以使用该用户来登陆如上的两个管理工具，通过 Web 方式进行 Tomcat 的配置和管理了。在“利用 Tomcat 创建和发布 Web 应用”章节中我们将详细介绍 Tomcat 的使用。

## 2.5. 小结

本章主要介绍了两个重要概念 **Web 服务器** 和 **Web 应用服务器**，通过两个例子说明 Web 服务器和 Web 应用服务器应用的场景。之后介绍了几种市面上常见的应用程序服务器，其中 Tomcat 将作为我们 Java Web 应用开发的服务器。

## 第3章 利用 Tomcat 创建和发布 Web 应用

自从 JSP 发布之后，推出了各式各样的 JSP 引擎。Apache Group 在完成 GNUJSP1.0 的开发以后，开始考虑在 SUN 的 JSWDK 基础上开发一个可以直接提供 Web 服务的 JSP 服务器，当然同时也支持 Servlet，这样 Tomcat 就诞生了。Tomcat 是 jakarta 项目中的一个重要的子项目，其被 JavaWorld 杂志的编辑选为 2001 年度最具创新的 java 产品(Most Innovative Java Product)，同时它又是 sun 公司官方推荐的 servlet 和 jsp 容器，因此其越来越多的受到软件公司和开发人员的喜爱。servlet 和 jsp 的最新规范都可以在 tomcat 的新版本中得到实现。其次，Tomcat 是完全免费的软件，任何人都可以从互联网上自由地下载。Tomcat 与 Apache 的组合相当完美。

与传统的桌面应用程序不同，Tomcat 中的应用程序是一个 WAR (Web Archive) 文件。WAR 是 Sun 提出的一种 Web 应用程序格式，与 JAR 类似，也是许多文件的一个压缩包。这个包中的文件按一定目录结构来组织：通常其根目录下包含有 Html 和 Jsp 文件或者包含这两种文件的目录，另外还会有一个 WEB-INF 目录，这个目录很重要。通常在 WEB-INF 目录下有一个 web.xml 文件和一个 classes 目录，web.xml 是这个应用的配置文件，而 classes 目录下则包含编译好的 Servlet 类和 Jsp 或 Servlet 所依赖的其它类(如 JavaBean)。通常这些所依赖的类也可以打包成 JAR 放到 WEB-INF 下的 lib 目录下，当然也可以放到系统的 CLASSPATH 中，但那样移植和管理起来不方便。

在 Tomcat 中，应用程序的部署很简单，你只需将你的 WAR 放到 Tomcat 的 webapp 目录下，Tomcat 会自动检测到这个文件，并将其解压。你在浏览器中访问这个应用的 Jsp 时，通常第一次会很慢，因为 Tomcat 要将 Jsp 转化为 Servlet 文件，然后编译。编译以后，访问将会很快。另外 Tomcat 也提供了一个应用：manager，访问这个应用需要用户名和密码，用户名和密码存储在一个 xml 文件中 (<CATALINA\_HOME>/conf/tomcat-users.xml)。通过这个应用，辅助于 Ftp，你可以在远程通过 Web 部署和撤销应用(当然本地也可以)。

Tomcat 不仅仅是一个 Servlet 容器，它也具有传统的 Web 服务器的功能：处理 Html 页面。但是与 Apache 相比，它的处理静态 Html 的能力就不如 Apache。我们可以将 Tomcat 和 Apache 集成到一块，让 Apache 处理静态 Html，而 Tomcat 处理 Jsp 和 Servlet。这种集成只需要修改 Apache 和 Tomcat 的配置文件即可。

在 Tomcat 中，你还可以利用 Servlet2.3/2.4 提供的事件监听器功能，来对你的应用或者 Session 实行监听。Tomcat 也提供其它的一些特征，如与 SSL 集成到一块，实现安全传输。还有 Tomcat 也提供 JNDI 支持，这与那些 J2EE 应用服务器提供的是一致的。说到这里我们要介绍通常所说的应用服务器(如 WebLogic)与 Tomcat 有何区别。应用服务器提供更多的 J2EE 特征，如 EJB，JMS，JAAS 等，同时也支持 Jsp 和 Servlet。而 Tomcat 则功能没有那么强大，它不提供 EJB 等支持。但如果与 JBoss (一个开源的应用服务器)集成到一块，则可以实现 J2EE 的全部功能。既然应用服务器具有 Tomcat 的功能，那么 Tomcat 有没有存在的必要呢？事实上，我们的很多中小应用不需要采用 EJB 等技术，Jsp 和 Servlet 已经足够，这时如果用应用服务器就有些浪费了。而 Tomcat 短小精悍，配置方便，能满足我们的需求，这种情况下我们自然会选择 Tomcat。

基于 Tomcat 的开发其实主要是 Jsp 和 Servlet 的开发，开发 Jsp 和 Servlet 非常简单，你可以用普通的文本编辑器或者 IDE，然后将其打包成 WAR 即可。我们这里要提到另外一个工具 Ant，Ant 也是 Jakarta 中的一个子项目，它所实现的功能类似于 Unix 中的 make。你需要写一个 build.xml 文件，然后运行 Ant 就可以完成 xml 文件中定义的工作，这个工具对于一个大的应用来说非常好，我们只需在 xml 中写很少的东西就可以将其编译并打包成 WAR。事实上，在很多应用服务器的发布中都包含了 Ant。另外，在 Jsp1.2 中，可以利用标签库实

现 Java 代码与 Html 文件的分离，使 Jsp 的维护更方便。

Tomcat 也可以与其它一些软件集成起来实现更多的功能。如与上面提到的 JBoss 集成起来开发 EJB，与 Cocoon (Apache 的另外个项目) 集成起来开发基于 Xml 的应用，与 OpenJMS 集成起来开发 JMS 应用，除了我们提到的这几种，可以与 Tomcat 集成的软件还有很多。

### 3.1. Tomcat 与 Servlet 容器

Jakarta Tomcat 服务器是一种 Servlet/Jsp 容器。Servlet 是一种运行在支持 Java 语言的服务器上的组件。

Servlet 最常见的用途是扩展 Java Web 服务器功能，提供非常安全的、可移植的、易于使用的 CGI (见注释) 替代品。它是一种动态加载的模块，为来自 Web 客户的请求提供服务。它完全运行在 Java 虚拟机上。由于它在服务器端运行，因此它的运行不依赖于浏览器。

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

当客户请求范围某个 Servlet 时，Servlet 容器将创建一个 ServletRequest 对象和 ServletResponse 对象。在 ServletRequest 对象中封装了客户请求信息，然后 Servlet 容器把 ServletRequest 对象和 ServletResponse 对象传递给客户请求的 Servlet。Servlet 把响应结果写到 ServletResponse 中，然后由 Servlet 容器把响应结构传给客户。Servlet 容器响应客户请求的过程如下图所示：

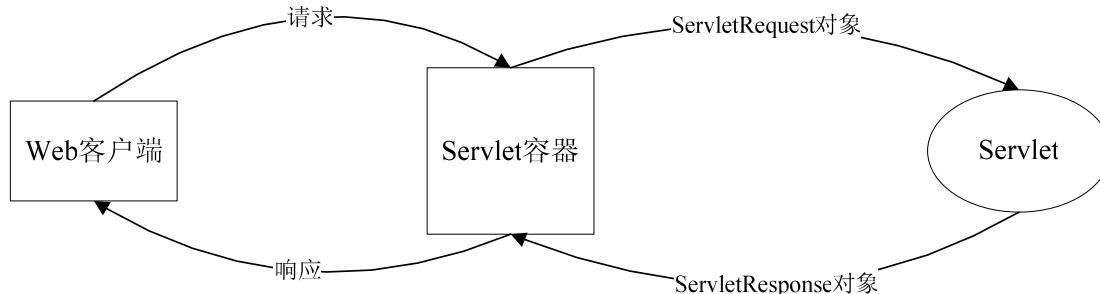


图 3-1: Servlet 容器响应客户请求的过程

**注意:** CGI: CGI(Common Gateway Interface)是 HTTP 服务器与你的或其它机器上的程序进行“交谈”的一种工具，其程序须运行在网络服务器上。绝大多数的 CGI 程序被用来解释处理来自表单的输入信息，并在服务器产生相应的处理，或将相应的信息反馈给浏览器。CGI 程序使网页具有交互功能。CGI 处理步骤(1)通过 Internet 把用户请求送到服务器。(2)服务器接收用户请求并交给 CGI 程序处理。(3)CGI 程序把处理结果传送给服务器。(4)服务器把结果送回到用户。

### 3.2. Tomcat 结构

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

```
<!--服务器-->
<Server>
    <!--服务-->
    <Service>
        <!--连接器-->
        <Connector/>
        <!--应用容器-->
        <Engine>
            <!--虚拟主机-->
            <Host>
                <!--WEB 应用程序-->
                <Context>
                </Context>
            </Host>
        </Engine>
    </Service>
</Server>
```

在以上 XML 代码中,每个元素都代表一种 Tomcat 组件.这些元素分四类:

**1.顶层类元素:** 顶层类元素包括<Server>元素和<Service>元素,他们位于整个配置文件的顶层.

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

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

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

下面, 再对一些基本的 Tomcat 元素进行介绍。如果要了解这些元素, 可以参考 (Server.xml 文件)。

- **<Server>元素**

代表了整个 Catalina Servlet 容器,它是 Tomcat 实例的顶层元素.可包含一个或多个<Service>元素.

- **<Service>元素**

包含一个<Engine>元素,以及一个或多个<Connector>元素,这些<Connector>元素共享同一个<Engine>元素.

- **<Connector>元素**

代表和客户程序实际交互的组件,他负责接收客户请求,以及向客户返回响应结果.

- **<Engine>元素**

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

- **<Host>元素**

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

- **<Context>元素**

每个<Context>元素代表了运行虚拟主机上的那个 Web 应用.一个<Host>元素中可以包含多个<Context>元素.

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

### 3.2.1. server.xml 文件属性

元素名	属性	解释
server	port	指定一个端口，这个端口负责监听关闭 tomcat 的请求
	shutdown	指定向端口发送的命令字符串
service	name	指定 service 的名字
Connector( 表示客户端和服务之间的连接)	port	指定服务器端要创建的端口号，并在这个端口监听来自客户端的请求
	minProcessors	服务器启动时创建的处理请求的线程数
	maxProcessors	最大可以创建的处理请求的线程数
	enableLookups	如果为 true，则可以通过调用 request.getRemoteHost() 进行 DNS 查询来得到远程客户端的实际主机名，若为 false 则不进行 DNS 查询，而是返回其 ip 地址
	redirectPort	指定服务器正在处理 http 请求时收到了一个 SSL 传输请求后重定向的端口号
	acceptCount	指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理
	connectionTimeout	指定超时的时间数(以毫秒为单位)
Engine(表示指定 service 中的请求处理器，接收和处理来自 Connector 的请求)	defaultHost	指定缺省的处理请求的主机名，它至少与其中的一个 host 元素的 name 属性值是一样的
Context( 表示一个 web 应用程序，通常为 WAR 文件，关于 WAR 的具体信息见 servlet 规范)	docBase	应用程序的路径或者是 WAR 文件存放的路径
	path	表示此 web 应用程序的 url 的前缀，这样请求的 url 为 http://localhost:8080/path/****
	reloadable	这个属性非常重要，如果为 true，则 tomcat 会自动检测应用程序的 /WEB-INF/lib 和 /WEB-INF/classes 目录的变化，自动装载新的应用程序，我们可以在不重启 tomcat 的情况下改变应用程序
Host(表示一个虚拟主机)	name	指定主机名
	appBase	应用程序基本目录，即存放应用程序的目录
	unpackWARs	如果为 true，则 tomcat 会自动将 WAR 文件解压，否则不解压，直接从 WAR 文件中运行应用程序
Logger(表示日志，调试和错误)	className	指定 logger 使用的类名，此类必须实现 org.apache.catalina.Logger 接口

信息)	prefix	指定 log 文件的前缀
	suffix	指定 log 文件的后缀
	timestamp	如果为 true, 则 log 文件名中要加入时间, 如下例:localhost_log.2001-10-04.txt
Realm(表示存放用户名, 密码及 role 的数据库)	className	指定 Realm 使用的类名, 此类必须实现 org.apache.catalina.Realm 接口
Valve(功能与 Logger 差不多, 其 prefix 和 suffix 属性解释和 Logger 中的一样)	className	指定 Valve 使用的类名, 如用 org.apache.catalina.valves.AccessLogValve 类可以记录应用程序的访问信息
	directory	指定 log 文件存放的位置
	pattern	有两个值, common 方式记录远程主机名或 ip 地址, 用户名, 日期, 第一行请求的字符串, HTTP 响应代码, 发送的字节数。combined 方式比 common 方式记录的值更多

Tomcat 各个组件之间的嵌套关系:

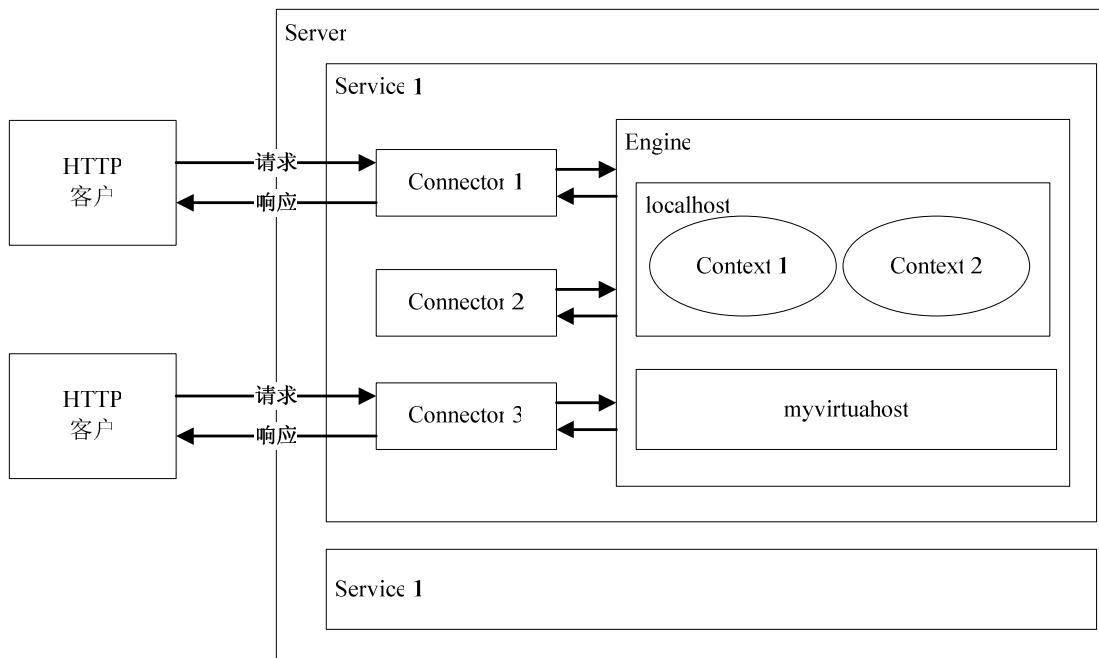


图 3-2

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

### 3.3. Java Web 应用简介

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

器中运行。”在 Java Web 应用中可以包含如下内容：

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

**注意：**可以说 Tomcat 服务器是 Servlet/JSP 容器，也可以说它是 Java Web 应用容器。

这两种说法并不矛盾，因为构成 Java Web 应用的最主要的组件就是 Servlet 和 JSP。

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

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

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

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

Tomcat 服务器的各个组件响应客户请求的过程如下图所示。在下图中，每个 Context 容器只对应一个 Java Web 应用。

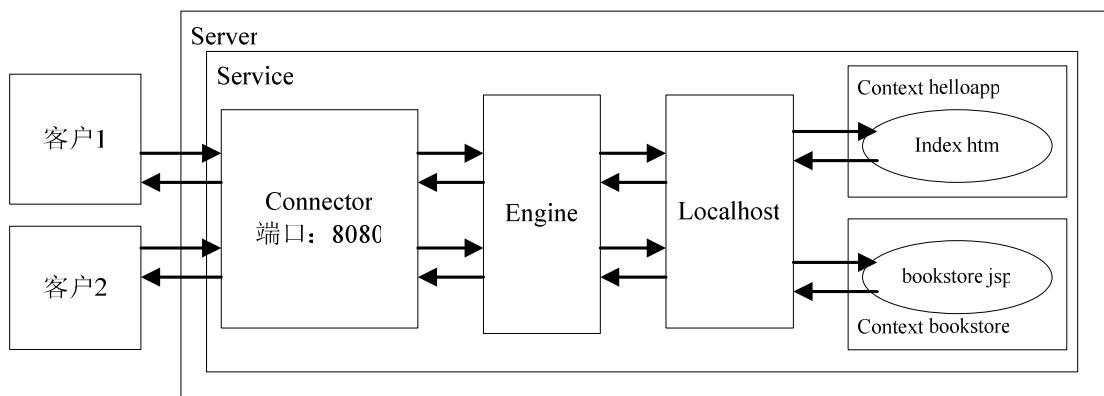


图 3-3

## 3.4. Tomcat 的工作模式

Tomcat 作为 Servlet 容器，有 3 种工作模式：独立的 Servlet 容器，进程内的 Servlet 容器和进程外的 Servlet 容器。

### 3.4.1. 独立的 Servlet 容器

Tomcat 作为独立的 Servlet 容器时，它是内置在 Web 服务器中的一部分，是指使用基于 Java 的 Web 服务器的情形，例如 Servlet 容器是 Java Web Server 的一部分。独立的 Servlet 容器是 Tomcat 的默认模式。然而，大多数的 Web 服务器并非基于 Java，所以 Tomcat 又发展了其他两种工作模式以与非基于 Java 的 Web 服务器结合。

### 3.4.2. 进程内的 Servlet 容器

Tomcat 作为进程内的 Servlet 容器时，Servlet 容器是作为 Web 服务器的插件和 Java 容器的实现。

Web 服务器插件在内部地址空间打开一个 JVM(Java Virtual Machine)使 Java 容器得以在内部运行。如有某个需要调用 Servlet 的请求，插件将取得对此请求的控制并将它传递(使用 JNI)给 Java 容器。进程内的容器对于多线程，单进程的服务器非常适合，并且提供了很好的运行速度，只是伸缩性有所不足。

**注意：**JNI 是 Java Native Interface 的简写，它是 Java 本地调用接口。通过这个接口，Java 程序可以和其他语言编写的本地程序进行通信。

### 3.4.3. 进程外的 Servlet 容器

Tomcat 作为进程外的 Servlet 容器时，Servlet 容器运行于 Web 服务器之外的地址空间，并且作为 Web 服务器的插件和 Java 容器的实现的结合。

Web 服务器插件和 Java 容器 JVM 使用 IPC 机制(通常是 TCP/IP)进行通信。当一个调用 Servlet 的请求到达时，插件将取得对此请求的控制并将其传递(使用 IPC 等)给 Java 容器，进程外容器的反应时间或进程外容器引擎不如进程内容器，但进程外容器引擎在许多其他可比的方面更好(如伸缩性，稳定性等)。

**注意：**IPC 是 Interprocess Communication(进程间通信)的简写，它是实现进程间通信的一种技术。

Tomcat 既可作为独立的容器(主要用于开发与调试)，又可作为对现有服务器的附加(当前支持 Apache,IIS 和 Netscape 服务器)。所以在配置 Tomcat 时，必须决定如何应用它，如果选择第 2 或第 3 种模式，还需要安装一个 Web 服务器接口。

## 3.5. Tomcat 的版本

Tomcat 是 Servlet 规范的标准实现，随着 SUN 不断推出新版本的 Servlet 和 JSP 的规范，Tomcat 的版本也随之不断地升级，现在最新的版本是 6.x(本文主要介绍 5.x)，但是还有大量的早期版本的 Tomcat 在被使用。在实际应用中，需要根据所使用的 Servlet 和 JSP 的版本来选择 Tomcat，Tomcat 版本和 Servlet 规范的对应关系如下表所示。

Servlet/JSP 规范版本	Tomcat 版本
2.4/2.0	5.x
2.3/1.2	4.1.31
2.2/1.1	3.3.2

Tomcat 的版本已经发展到 5.x，它的发展受到了很多开发者的关注，但是如果在 Web 应用中并没有使用很多 Servlet 规范新版本里的技术，总是使用最新版本的 Tomcat 并不是一个明智的选择。其中 Tomcat 4.1.x 是比较稳定和成熟的版本(对于使用 Servlet 2.3 版以下的应用程序)，被广泛应用到实际的 Web 应用的开发中。下面介绍各个版本的特点，希望通过这些版本的简单介绍，可以使读者对 Tomcat 的过去、现在和发展有所了解。

**Apache Tomcat 5.5.x:** 虽然它和 Apache Tomcat 5.0.x 支持同样版本的 Servlet 和 JSP 标准，但是它在很多方面也有比较大的改变，使得它在性能、稳定性、总花费时间等方面都有了很大的改进。

**Apache Tomcat 5.0.x:** 它相对于 Apache Tomcat 4.1 在很多方面作了改进。包括如下方面：

- 对性能作了优化，减少了垃圾回收的时间。
- 重新设计了应用程序部署器，可以使用一个可选的应用程序部署器完成验证和编译。
- 使用 JMX 技术对服务器进行监控。

- 加强了服务器的可扩展性和可靠性。
- 增强了 Taglibs 的处理能力。
- 使用 Windows 和 UNIX 本地的包装器改进了平台集成性。
- 增强的 Security Manager 支持。
- 完善了会话集群。

Apache Tomcat 4.x: 在 Apache Tomcat 4.x 里实现了一个新的名为 Catalina 的 Servlet 容器，它是基于一个完全新的体系结构的、完全放弃了 Tomcat 3.x 的构架。

Apache Tomcat 4.1.x: Apache Tomcat 4.1.x 是 Apache Tomcat 4.0.x 的一个升级，有很多新的关键更新，包括以下内容。

- 基于 JMX 的管理控制。
  - 实现了新的 Coyote connector (支持 HTTP/1.1、AJP 1.3 和 JNI)。
  - 重写了 Jasper JSP 编译器。
  - 提高了 Web 管理应用与开发工具的集成。
  - 提供了客户化的 Ant 任务，使 Ant 程序根据 build.xml 和 Web 管理应用交互。
- Apache Tomcat 3.x 是比较原始的版本，这里就不介绍了。

## 3.6. Tomcat 安装配置

Tomcat 是基于 Java 的一个 Servlet 容器，它的运行离不开 JDK 的支持。所以，要首先安装 JDK，然后才能正确安装 Tomcat。下面就一步步介绍如何从零开始安装 Tomcat 服务器。

### 3.6.1. j2sdk 的安装

j2sdk 是 Java 语言的编译环境，可以从 SUN 公司的网站上免费下载。下载网址是：

<http://java.sun.com/j2se/downloads/index.html>

把 JDK 下载后执行安装程序，假定安装目录是 C:\j2sdk1.4.2，把这个目录设定为 JAVA\_HOME。

安装完成后，需要做些配置工作，JDK 才能开始正常工作，可以按照下面介绍的步骤配置 JDK。

(1) 在桌面上右击【我的电脑】，选择【属性】命令，在出现的对话框中选择【高级】选项卡，然后单击【环境变量】按钮，出现如下图所示的对话框。

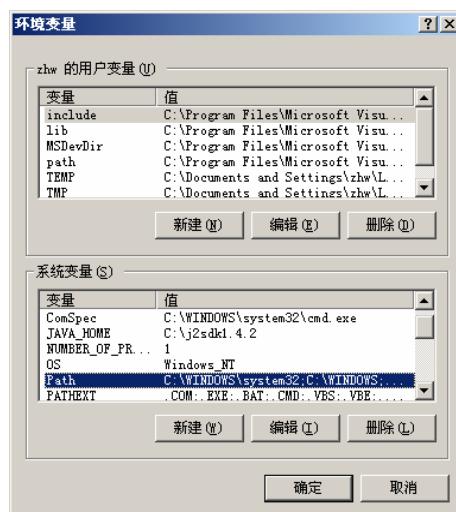


图 3-4

(2) 检查在【系统变量】部分是否有 Path 变量, 如果没有新建一个名为 Path 的变量, 则添加路径 C:\j2sdk1.4.2\bin;; 如果有, 则在原有路径的末尾添加 C:\j2sdk1.4.2\bin;;。效果如下图所示。



图 3-5

- (3) 单击【确定】按钮, 保存所做的修改。  
(4) 新建一个系统变量, 名为 JAVA\_HOME, 值为 C:\j2sdk1.4.2。  
(5) 新建一个系统变量, 名为 CLASSPATH, 值为.;C:\j2sdk1.4.2\lib\tools.jar;C:\j2sdk1.4.2\lib\dt.jar。

 注意: CLASSPATH 变量的值必须以“.”开头。

### 3.6.2. 使用可执行文件安装 Tomcat

只有在确保 JDK 安装正确的情况下才可以安装 Tomcat, Tomcat 提供了可执行文件的安装程序, 可以从其官方网站免费下载, 网址为 <http://jakarta.apache.org/tomcat/index.html>。

我们使用的版本是 Tomcat 5.0.28, 执行 Tomcat 5.0.28 安装程序, 使用默认设置就可以了。

假定安装的主目录是 C:\Tomcat 5.0, 把它设定为 TOMCAT\_HOME, 添加一个新的系统变量 TOMCAT\_HOME, 将其值设置为 C:\Tomcat 5.0 (Tomcat 安装的主目录), 然后单击【确定】按钮, 保存所做的更改。

### 3.6.3. 解压安装 Tomcat

解压安装 Tomcat 与使用可执行安装程序的配置过程是基本一样的。

- (1) 把压缩包解压到硬盘的某个目录, 并指定这个目录为 TOMCAT\_HOME。  
(2) 像使用可执行文件安装 Tomcat 那样设置 TOMCAT\_HOME 环境变量。

 注意: 解压安装的 Tomcat 需要直接运行 TOMCAT\_HOME\bin 目录下的启动脚本 startup.bat 来启动。

另外, 在解压缩时没有设置 Tomcat 管理程序的用户名和密码, 所以, 如果要使用 Tomcat 的 Admin 和 Manager 应用来管理 Tomcat 的各种资源必须手动添加用户角色、用户名和密码, 具体添加的方法请查阅关于 Tomcat 用户角色和 Web 应用安全限制的介绍。

## 3.7. 基本的安装测试

按照上面介绍的步骤安装好 Tomcat 后, 可以启动 Tomcat 并测试其安装是否正确。

如果使用可执行文件安装 Tomcat, 可以在 Windows 系统中选择【开始】/【所有程序】/Apache Tomcat 5.0/Configure Tomcat, 在弹出的对话框中单击 Start 按钮, 就可以启动 Tomcat 了。

注意：关于 Tomcat 的启动和关闭方法还有几种，在后面将会介绍。

Tomcat 启动完成后，在浏览器地址栏中输入 `http://localhost:8080/`，可以看到如下图所示的 Tomcat 的欢迎页面。

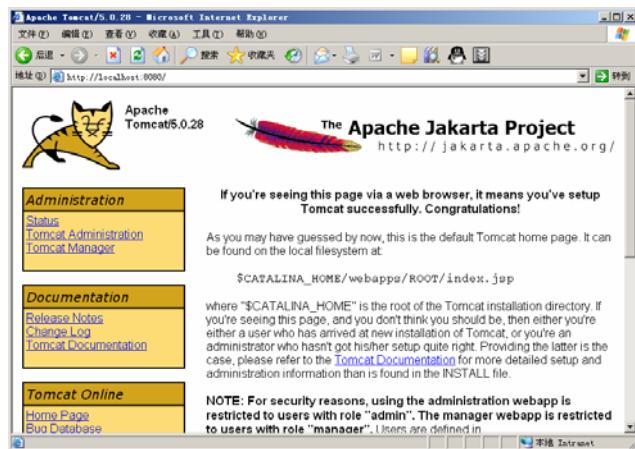


图 3-6

单击该页面左下角的 `Servlet Examples` 链接，执行名为 `Request Info` 的例子，此时页面显示如下图所示。

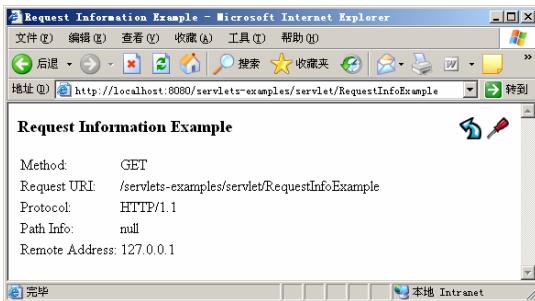


图 3-7

### 3.8. Tomcat Admin 的安装

Tomcat 的版本从 5.5.x 开始不再在 Tomcat 的安装程序中提供 Admin 和 Manager 两个用于管理 Tomcat 资源的 Web 应用，如果开发者需要使用这两个应用程序，必须重新下载安装。要手动安装 Admin 应用程序，可以按照下面的步骤进行：

- (1) 从 Tomcat 的官方网站免费下载 Admin 应用程序。
- (2) 把下载后的压缩包解压缩到本地硬盘。
- (3) 把解压目录下的 conf 和 server 文件夹复制到<TOMCAT\_HOME>目录下，然后启动 Tomcat，在浏览器地址栏中输入网址。

`http://localhost:8080/admin/`

此时可以看到页面显示如下图所示。



图 3-8

**注意：**如果使用后安装的 Admin 或者 Manager 应用，需要在 Tomcat 设置相应的用户名和用户角色，才可以登录这两个应用程序对 Tomcat 系统进行管理。

### 3.9. Tomcat 启动分析

Tomcat 有多种启动方式，下面根据不同的安装方式对 Tomcat 的启动方式进行介绍，然后简单分析 Tomcat 的启动脚本。

如果使用 Tomcat 的 Windows 安装程序安装 Tomcat，可以在 Windows 的所有程序列表中找到启动 Tomcat 的程序，具体方法在介绍 Tomcat 的安装测试时有过介绍，这里不再详述。

如果在安装时，把 Tomcat 安装为 Windows 服务，也可以从 Windows 服务窗口中启动 Tomcat。具体步骤如下：

- (1) 在【控制面板】窗口中双击【管理工具】。
- (2) 在管理工具页面中双击【服务】快捷方式。
- (3) 在【服务】控制台中右部的列表中找到【Tomcat 服务】一项，并双击，出现如下图所示对话框。



图 3-9

- (4) 使用【启动】和【停止】按钮启动和关闭 Tomcat。

**注意：**这种方式只在 Tomcat 被设置为 Windows 服务时可用。

还有一种方式是直接运行 Tomcat 的启动批处理文件 startup.bat，这种方式无论对于使用安装程序安装的还是解压缩安装的都可用。启动批处理文件 startup.bat 位于 <TOMCAT\_HOME>\bin 目录下，双击这个批处理文件就可以启动 Tomcat。另外还有一个批处理文件 shutdown.bat 用于关闭 Tomcat。

通过分析 startup.bat 文件和 shutdown.bat 文件可以发现，它们都调用了同一目录下的 catalina.bat 脚本文件，只是使用了不同的初始化参数，例如 startup.bat 文件使用 start 参数调用 catalina.bat 脚本文件，而 shutdown.bat 文件使用 stop 参数调用，catalina.bat 脚本文件还允许一些其他的传入参数，如下表所示。

参 数	描 述
-----	-----

start	启动 Tomcat 服务器（新窗口）
stop	关闭 Tomcat 服务器
run	启动 Tomcat 服务器
debug	以 debug 模式启动 Tomcat 服务器
embedded	以 embedded 模式启动 Tomcat 服务器

### 3.10. Tomcat 目录结构

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

可以在 Java Web 应用下的 WEB-INF 目录中建立 lib 子目录，存放各种 Jar 文件，这些 Jar 文件只能被当前 Web 应用访问。

在运行过程中，Tomcat 类装载器先装载 classes 目录下的类，再装载 lib 目录下的类。如果两个目录下存在同名的类，classes 目录下的类具有优先权。

### 3.11. 创建和发布 Web 应用

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

#### 3.11.1. Web 应用的目录结构

Web 应用具有固定的目录结构，这里假定开发一个名为 helloapp 的 Web 应用，目录结构如下：

目录	描述
/helloapp	根目录，所有的 JSP 和 HTML 文件都存放于此目录下
/helloapp/WEB-INF	存放 Web 应用的发布描述文件 web.xml
/helloapp/WEB-INF/classes	存放各种 class 文件，Servlet 类文件也放于此目录下
/helloapp/WEB-INF/lib	存放 Web 应用所需的各种 Jar 文件。例如： JDBC 驱动 Jar

在 helloapp 应用中创建了如下组件：

HTML 组件： index.htm

JSP 组件： login.jsp 和 hello.jsp

Servlet 组件： DispatcherServlet

这些组件之间的连接关系为：

index.htm --> login.jsp --> DispatcherServlet --> hello.jsp

### 3.11.2. Eclipse+MyEclipse 开发 helloapp 的 Web 应用

Eclipse+MyEclipse 安装配置使用见《Eclipse 整合开发工具》与《Eclipse 与 MyEclipse》。

步骤一：建立一个工程，选择 File --> New --> Project，如下图所示：

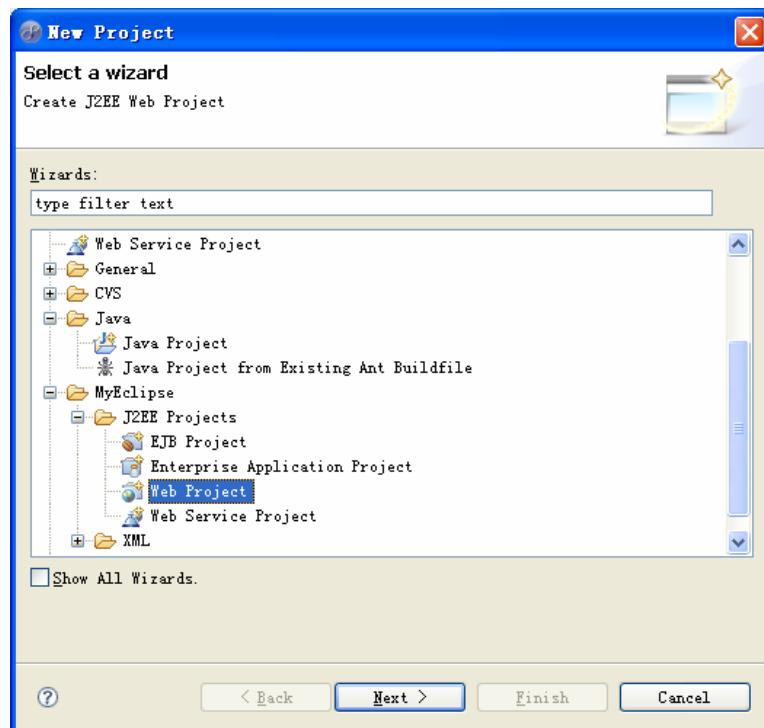


图 3-10

步骤二：输入项目名称 helloapp，并点击完成。

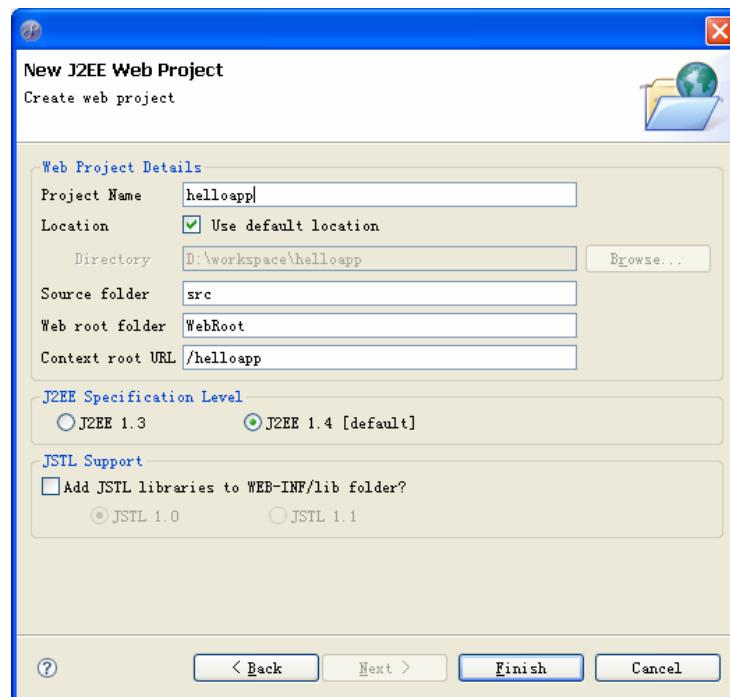


图 3-11

Eclipse 自动生成了一个 WebProject 的目录结构，如下图所示：



图 3-12

步骤三：部署 HTML 文件，index.htm，如下图所示，输入文件名。

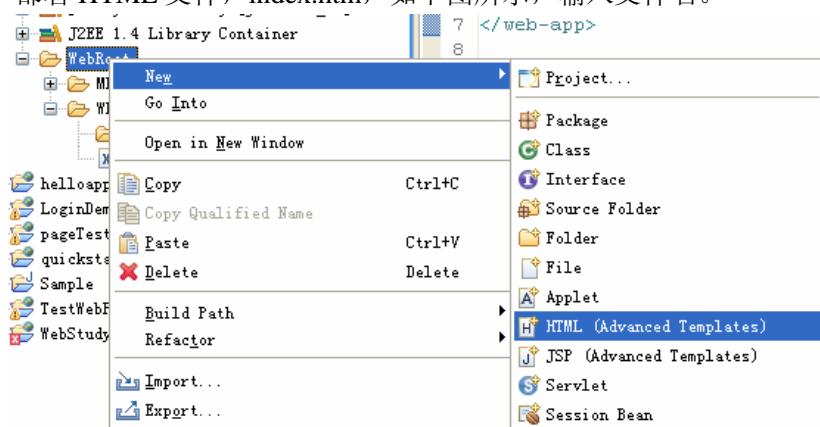


图 3-13

代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
```

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

步骤四：在 Eclipse 中配置 Tomcat 运行环境：

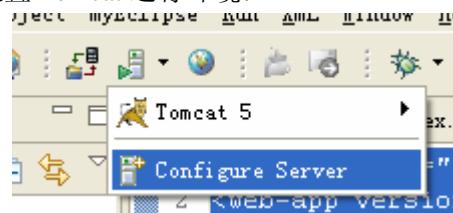


图 3-14

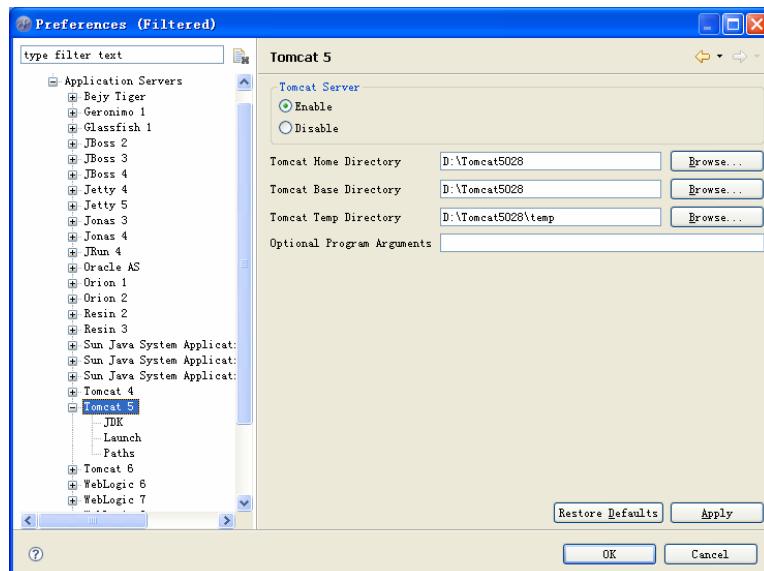


图 3-15

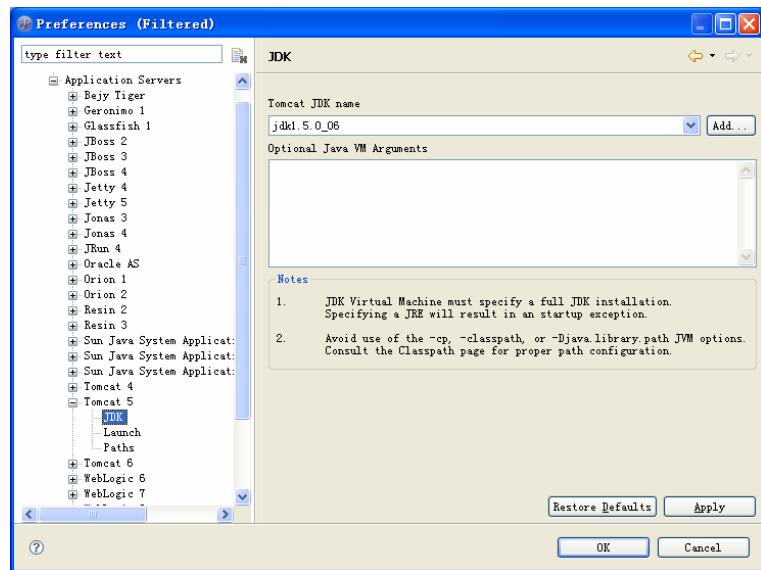


图 3-16

步骤五：配置完成后点击部署，完成后 OK，将应用部署到 Tomcat 中去：



图 3-17

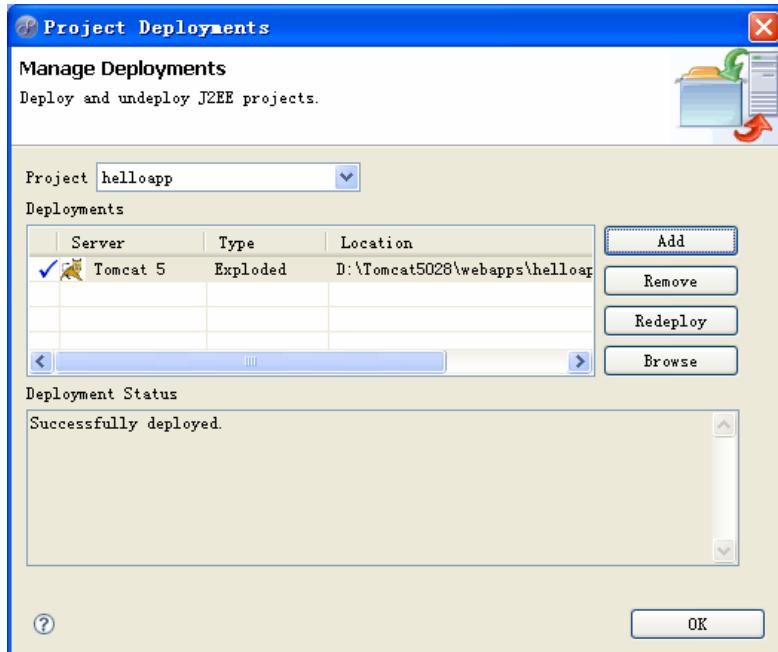


图 3-18

步骤六：运行 Tomcat

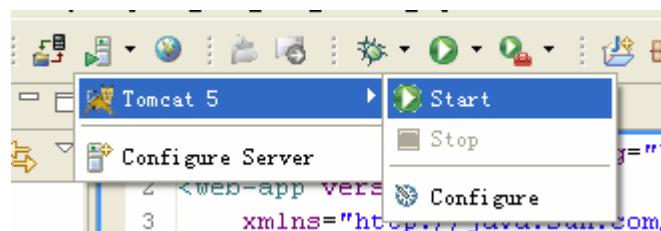


图 3-19

步骤七：在浏览器中输入 `http://localhost:8080/helloapp/index.htm` 该页显示结果如下图所示：

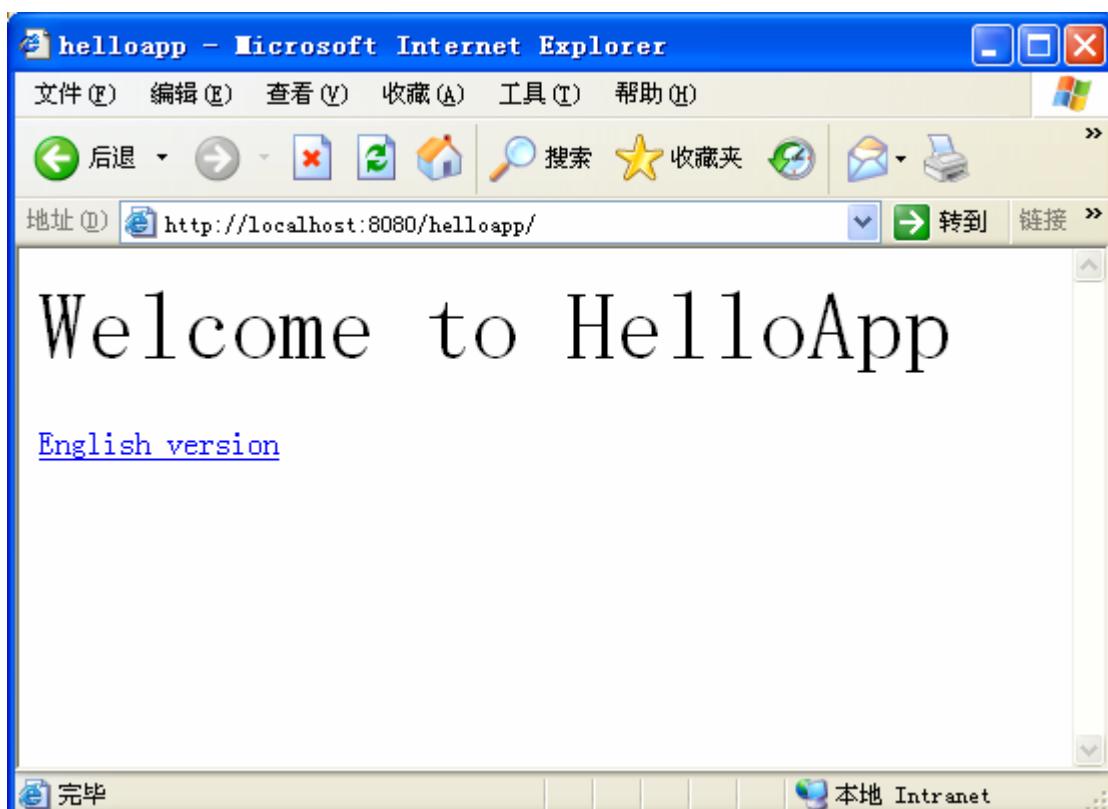


图 3-20

#### 步骤八：部署 JSP

创建 2 个 JSP 文件，其中一个是 `login.jsp`，它显示登陆页面，要求输入用户名和口令，这个页面链接到一个名为 `DispatcherServlet` 的 Servlet。还有一个 JSP 文件是 `hello.jsp`。这个 JSP 被 `DispatcherServlet` 调用，实现 Hello 页面。（Servlet/JSP 语法将在后续的课程中介绍，这里着重介绍发布的过程。）这两个 JSP 文件都应放在 `helloapp` 的 `WebRoot` 目录下。

`login.jsp` 代码如下：

---

```

<html>
  <head>
    <title>helloapp</title>
  </head>
  <body>

    <br>
    <form name="loginForm" method="post" action="dispatcher">

```

```
<table>
  <tr>
    <td>
      <div align="right">
        User Name:
      </div>
    </td>
    <td>
      <input type="text" name="username">
    </td>
  </tr>
  <tr>
    <td>
      <div align="right">
        Password:
      </div>
    </td>
    <td>
      <input type="password" name="password">
    </td>
  </tr>
  <tr>
    <td></td>
    <td>
      <input type="Submit" name="Submit" value="Submit">
    </td>
  </tr>
</table>
</form>
</body>
</html>
```

---

hello.jsp 代码如下：

---

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

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

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

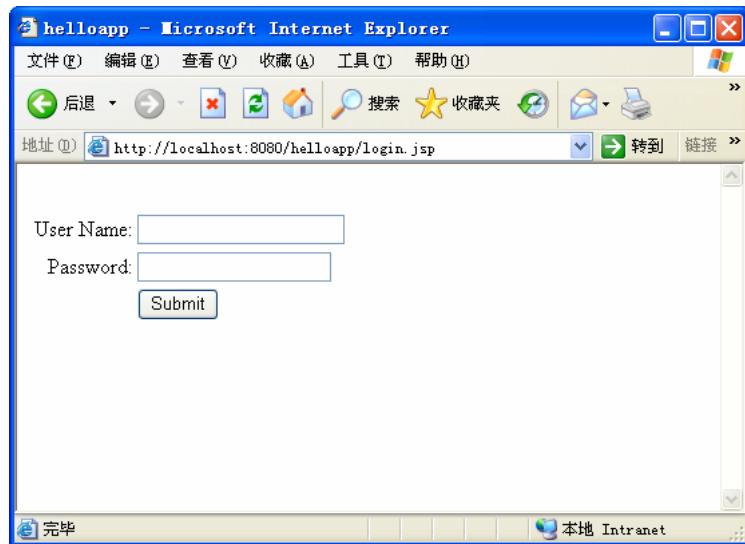


图 3-21

#### 步骤九：部署 Servlet

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

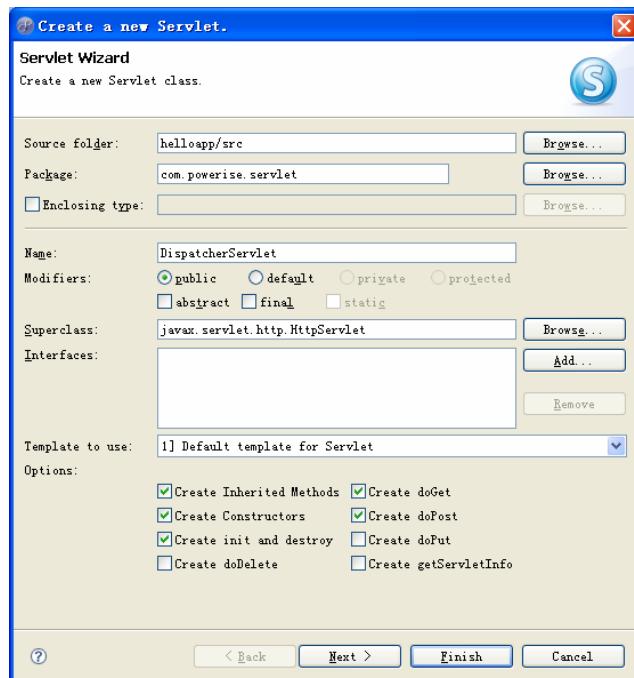


图 3-22

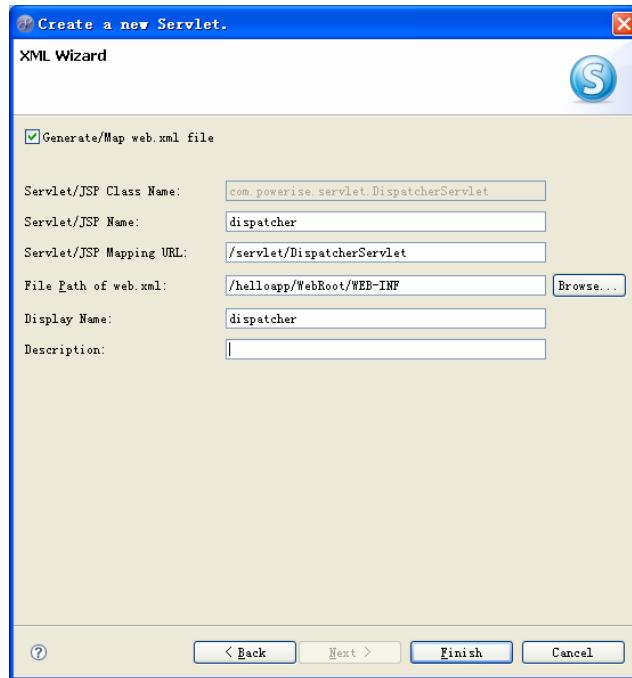


图 3-23

DispatcherServlet 代码如下：

```
package com.powerise.servlet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class DispatcherServlet extends HttpServlet {

    private String target = "/hello.jsp";

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

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

        // If it is a get request forward to doPost()
    }
}
```

```
    doPost(request, response);  
}  
  
public void doPost(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
  
    // Get the username from the request  
    String username = request.getParameter("username");  
    // Get the password from the request  
    String password = request.getParameter("password");  
  
    // Add the user to the request  
    request.setAttribute("USER", username);  
    request.setAttribute("PASSWORD", password);  
  
    // Forward the request to the target named  
    ServletContext context = getServletContext();  
  
    System.out.println("Redirecting to " + target);  
    RequestDispatcher dispatcher = context.getRequestDispatcher(target);  
    dispatcher.forward(request, response);  
}  
  
public void destroy() {  
}  
}
```

---

点击完成，Eclipse 自动帮助完成 web.xml 文件，查看一下。重复步骤五，Redeploy 这个应用。

#### 步骤十：部署 JSP Tag Library

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

1) 编写用户处理 hello 标签的类 HelloTag.java，以下是 HelloTag.java 的代码：

```
package com.powerise.tag;  
  
import javax.servlet.jsp.JspException;  
import javax.servlet.jsp.JspTagException;  
import javax.servlet.jsp.tagext.TagSupport;  
  
public class HelloTag extends TagSupport {  
    public HelloTag() {  
    }  
}
```

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

    try {

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

        throw new JspTagException(e.getMessage());
    }

    // We want to return SKIP_BODY because this Tag does not support
    // a Tag Body
    return SKIP_BODY;
}

public void release() {

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

- 2) 创建 Tag Library 的描述文件 mytaglib.tld 文件，在这个文件中定义了 mytaglib 标签库和 hello 标签。这个文件存放在/WEB-INF/mytaglib.tld。代码如下：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

```
<!-- a tag library descriptor -->
```

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

    <tag>
        <name>hello</name>
        <tagclass>com.powerise.tag.HelloTag</tagclass>
```

```
<bodycontent>empty</bodycontent>
<info>Just Says Hello</info>
</taglib>

3) 在 web.xml 中加入<taglib>元素，代码如下：
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <display-name>dispatcher</display-name>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            com.powerise.servlet.DispatcherServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/dispatcher</url-pattern>
    </servlet-mapping>
    <!—在Servlet2.4版本后不需要声明-->
    <taglib>
        <taglib-uri>/mytaglib</taglib-uri>
        <taglib-location>/WEB-INF/mytaglib.tld</taglib-location>
    </taglib>
</web-app>
```

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

**注意：**当使用 web-app\_2\_4 版本 dtd ( Document Type Definition 文档类型定义 ) 定义的时候，是不需要在 web.xml 文件里面显示声明标记库插件的，使用低于 2\_4 版本的时候才需要使用，上例中我们使用了显示声明的方式其实是不必要的。

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

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

以上 taglib 指令中，prefix 用来指定引用 mytaglib 标签库时的前缀，修改后的 hello.jsp 文件如下：

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

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

<body>
    <b><mm:hello /> : <%=request.getAttribute("USER")%>
    </b>
</body>
</html>
```

hello.jsp 修改后，再依次访问 index.htm --> login.jsp --> DispatcherServlet --> hello.jsp，最后生成的网页如下图所示：

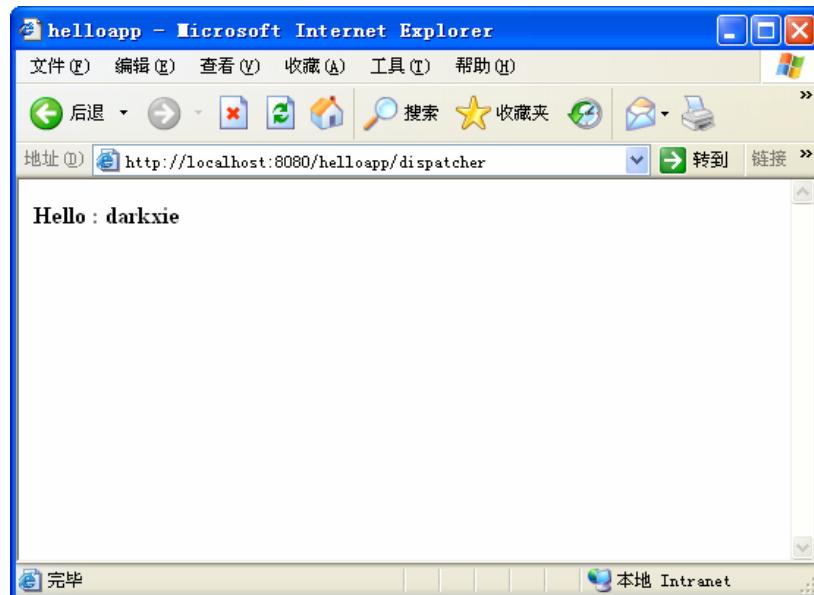


图 3-24

#### 步骤十一： 创建并发布 WAR 文件

Tomcat 既可以运行采用开发式目录结构，也可以运行 WAR (web achieve) 文件，只要将 Eclipse 工作目录下该项目的 WebRoot 文件夹拷贝到<CATALINA\_HOME>/webapps 目录下，即可运行开发式目录结构的 helloapp 应用。

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

#### 利用 Eclipse 进行打包：

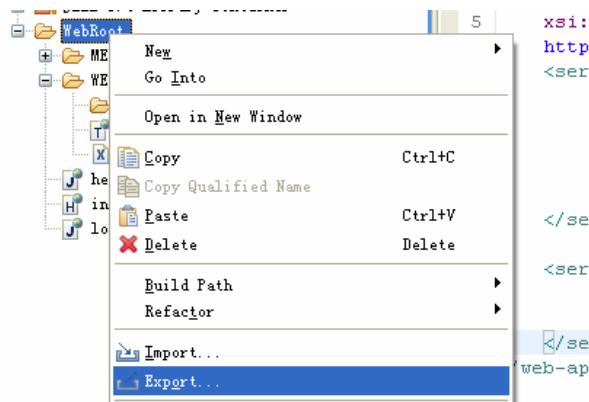


图 3-25

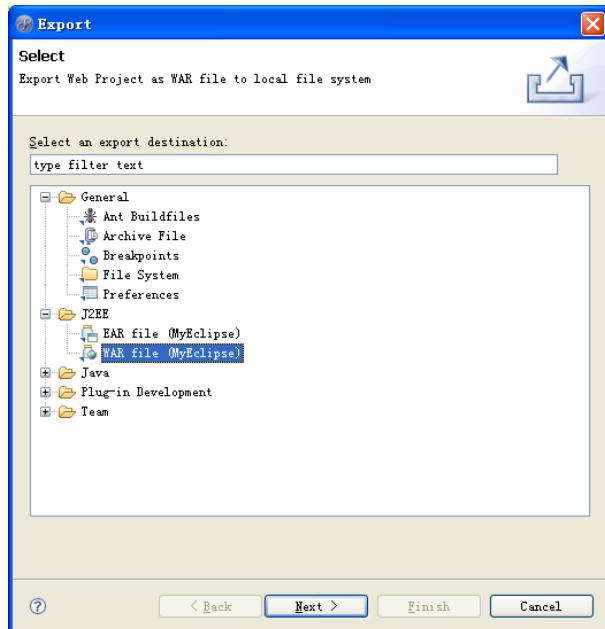


图 3-26

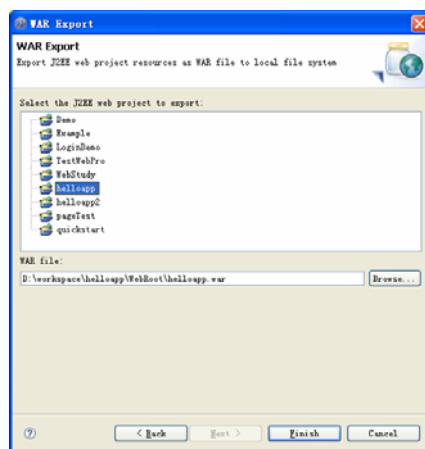


图 3-27

点击 Finish 即完成打包步骤，生成好 WAR 文件。

**利用 java 命令进行打包：**

进入你要打包的目录如：D:\workspace\helloapp\WebRoot

在命令行模式下输入：jar cvf helloapp.war \*.\*即生成好 WAR 文件。

在 Tomcat 下发布 WAR 文件:

- 1) 把 helloapp.war 文件拷贝到<CATALINA\_HOME>/webapps 目录下;
- 2) 删除原先的 helloapp 目录;
- 3) 重启 Tomcat 服务器;

## 3.12. 手工部署开发式目录结构

### 3.12.1. 在 server.xml 文件中加入<Context>元素

<Context>元素是<CATLINA\_HOME>/conf/server.xml 中使用最频繁的元素，它代表了运行在<Host>上单个 Web 应用。一个<Host>中可以有多个<Context>元素。每个 Web 应用必须有唯一的 URL 路径，这个 URL 路径在<Context>元素的 path 属性中设定。

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

```
<Host name="localhost" debug="0" appBase="webapps"
      unpackWARs="true" autoDeploy="true"
      xmlValidation="false" xmlNamespaceAware="false">
  .....
  <Context    path="/helloapp"    docBase="D:\workspace\helloapp\WebRoot"    debug="5"
  reloadable="true" crossContext="true"/>
  .....
</Host>
```

server.xml 具体属性配置说明见上节。

**注意：**在开发阶段将 reloadable 属性设为 true，有助调试程序。但是该功能会加重服务器的运行负荷，建议在 Web 应用产品发布阶段，将这个属性设为 false。

## 3.13. 利用 Tomcat 配置虚拟主机

在 Tomcat 的配置文件 server.xml 中，Host 元素代表虚拟主机，在同一个 Engine 元素下可以配置多个虚拟主机。例如，有两个公司的 Web 应用都发布在同一个 Tomcat 服务器上，可以分别为这两个公司创建一个虚拟主机，分别为：

```
www.company1.com
www.company2.com
```

这样当Web客户访问以上两个Web应用时分别拥有各自的主机。此外，还可以为虚拟主机建立别名，例如，如果希望Web客户访问 www.company1.com或者company1.com连接到同一个Web，那么可以把compan1.com作为虚拟主机的别名来处理。

步骤一： 打开<CATLINA\_HOME>/conf/server.xml 文件，发现<Engine>元素中已经有一个名为 localhost 的<Host>元素，可以在它后面(即</Host>后面)加入如下<Host>元素：

```
<Host name="www. company1.com" debug="0" appBase="C:\ company1"
      unpackWARs="true" autoDeploy="true">
  <alias>company1.com</alias>
  <alias>company1</alias>

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

</Host>

在<Host>的 deployOnStartup 属性为 true 的情况下，如果你没有在 server.xml 中为 helloapp 应用加入<Context>元素，Tomcat 服务器也可以自动发布和运行 helloapp 应用。在这种情况下，Tomcat 使用默认的 DefaultContext。关于 DefaultContext 的内容可以参考 Tomcat 文档：  
<CATLINA\_HOME>/webapps/tomcat-docs/config/defaultcontext.html

步骤二： 把 helloapp 应用（WAR 文件或者整个 helloapp 项目的 WebRoot 目录）拷贝到 appBase 属性指定的目录 C:\company1 下。

步骤三： 为了使配置的虚拟机生效，必须在 DNS（域名解析，就是域名对应到 IP 地址的一种服务）服务器中注册以上虚拟机名和别名，使他们的 IP 地址都指向 Tomcat 服务器所在的机器。必须注册以下名字：

www.company1.com

company1.com

company

步骤四： 重启 Tomcat 服务器，然后通过浏览器访问：

<http://www.company1.com:8080/helloapp/index.htm>

如果返回正常的页面就说明配置成功。

**△ 注意：** 在开发阶段我们无法配置 DNS 服务器，可以通过修改本机 C:\WINDOWS\system32\drivers\etc\hosts 文件来解决这个问题，在这个文件的最后一行加入：  
<你的 IP 地址> www.company1.com  
<你的 IP 地址> company1.com  
<你的 IP 地址> company

## 3.14. 小结

本章介绍了 Jakarta Tomcat 服务器的概念和主要功能，简单讨论了 Java Web 应用，这是 Tomcat 服务器工作的核心内容，还讲解了如何安装 Tomcat 服务器步骤，通过 helloapp Web 应用的例子，介绍了 Tomcat 上创建和发布 Web 应用的步骤。在这里我们可以学到：

- 创建 Web 应用的目录文件；
- 创建 web.xml 文件；
- 把 HTML、Servlet、JSP 和 Tag Library 部署到 Web 应用中；
- 把整个 Web 应用打包发布部署；
- 配置虚拟主机。

# 第4章 Java Servlet

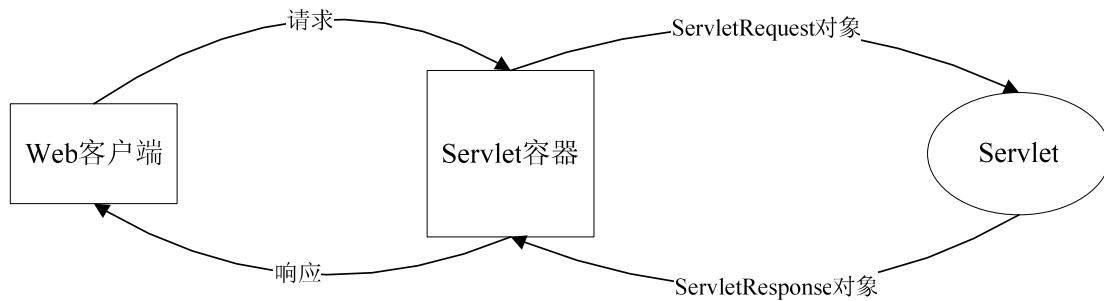


图 4-1: Servlet 容器响应用户请求的过程

Servlet 是 Java 技术对 CGI 编程的回答。Servlet 程序在服务器端运行，动态地生成 Web 页面。与传统的 CGI 和许多其他类似 CGI 的技术相比，Java Servlet 具有更高的效率，更容易使用，功能更强大，具有更好的可移植性，更节省投资：

## 高效

在传统的 CGI 中，每个请求都要启动一个新的进程，如果 CGI 程序本身的执行时间较短，启动进程所需要的开销很可能反而超过实际执行时间。而在 Servlet 中，每个请求由一个轻量级的 Java 线程处理（而不是重量级的操作系统进程）。

在传统 CGI 中，如果有 N 个并发的对同一 CGI 程序的请求，则该 CGI 程序的代码在内存中重复装载了 N 次；而对于 Servlet，处理请求的是 N 个线程，只需要一份 Servlet 类代码。在性能优化方面，Servlet 也比 CGI 有着更多的选择，比如缓冲以前的计算结果，保持数据库连接的活动，等等。

## 方便

Servlet 提供了大量的实用工具例程，例如自动地解析和解码 HTML 表单数据、读取和设置 HTTP 头、处理 Cookie、跟踪会话状态等。

## 功能强大

在 Servlet 中，许多使用传统 CGI 程序很难完成的任务都可以轻松地完成。例如，Servlet 能够直接和 Web 服务器交互，而普通的 CGI 程序不能。Servlet 还能够在各个程序之间共享数据，使得数据库连接池之类的功能很容易实现。

## 可移植性好

Servlet 用 Java 编写，Servlet API 具有完善的标准。因此，为 I-Planet Enterprise Server 写的 Servlet 无需任何实质上的改动即可移植到 Apache、Microsoft IIS 或者 WebStar。几乎所有的主流服务器都直接或通过插件支持 Servlet。

## 节省投资

不仅有许多廉价甚至免费的 Web 服务器可供个人或小规模网站使用，而且对于现有的服务器，如果它不支持 Servlet 的话，要加上这部分功能也往往是免费的（或只需要极少的投资）。

## 4.1. 第一个 Servlet

### 4.1.1. 基本结构

下面的代码显示了一个简单 Servlet 的基本结构。该 Servlet 处理的是 GET 请求，所谓的 GET 请求，如果你不熟悉 HTTP，可以把它看成是当用户在浏览器地址栏输入 URL、点击 Web 页面中的链接、提交没有指定 METHOD 的表单时浏览器所发出的请求。Servlet 也

可以很方便地处理 POST 请求。POST 请求是提交那些指定了 METHOD=“POST”的表单时所发出的请求，具体请参见稍后几节的讨论。

```
package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class myFirstServlet extends HttpServlet {

    /**
     * Constructor of the object.
     */
    public myFirstServlet() {
        super();
    }

    /**
     * Destruction of the servlet. <br>
     */
    public void destroy() {
        super.destroy(); // Just puts "destroy" string in log
        // Put your code here
    }

    /**
     * The doDelete method of the servlet. <br>
     *
     * This method is called when a HTTP delete request is received.
     *
     * @param request the request send by the client to the server
     * @param response the response send by the server to the client
     * @throws ServletException if an error occurred
     * @throws IOException if an error occurred
     */
    public void doDelete(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {

        // Put your code here
    }
}
```

```
/**  
 * The doGet method of the servlet. <br>  
 *  
 * This method is called when a form has its tag value method equals to get.  
 *  
 * @param request the request send by the client to the server  
 * @param response the response send by the server to the client  
 * @throws ServletException if an error occurred  
 * @throws IOException if an error occurred  
 */  
  
public void doGet(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
    out  
        .println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01  
Transitional//EN\">");  
    out.println("<HTML>");  
    out.println(" <HEAD><TITLE>A Servlet</TITLE></HEAD>");  
    out.println(" <BODY>");  
    out.print("    This is ");  
    out.print(this.getClass());  
    out.println(", using the GET method");  
    out.println(" </BODY>");  
    out.println("</HTML>");  
    out.flush();  
    out.close();  
}  
  
/**  
 * The doPost method of the servlet. <br>  
 *  
 * This method is called when a form has its tag value method equals to post.  
 *  
 * @param request the request send by the client to the server  
 * @param response the response send by the server to the client  
 * @throws ServletException if an error occurred  
 * @throws IOException if an error occurred  
 */  
  
public void doPost(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out
.println("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">");
out.println("<HTML>");
out.println(" <HEAD><TITLE>A Servlet</TITLE></HEAD>");
out.println(" <BODY>");
out.print("    This is ");
out.print(this.getClass());
out.println(", using the POST method");
out.println(" </BODY>");
out.println("</HTML>");
out.flush();
out.close();
}

/**
 * The doPut method of the servlet. <br>
 *
 * This method is called when a HTTP put request is received.
 *
 * @param request the request send by the client to the server
 * @param response the response send by the server to the client
 * @throws ServletException if an error occurred
 * @throws IOException if an error occurred
 */
public void doPut(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

    // Put your code here
}

/**
 * Returns information about the servlet, such as
 * author, version, and copyright.
 *
 * @return String information about this servlet
 */
public String getServletInfo() {
    return "This is my default servlet created by Eclipse";
}

/**
```

```
* Initialization of the servlet. <br>
*
* @throws ServletException if an error occurs
*/
public void init() throws ServletException {
    // Put your code here
}

}
```

---

如果某个类要成为 Servlet，则它应该从 HttpServlet 继承，根据数据是通过 GET 还是 POST 发送，覆盖 doGet、doPost 方法之一或全部。doGet 和 doPost 方法都有两个参数，分别为 HttpServletRequest 类型和 HttpServletResponse 类型。HttpServletRequest 提供访问有关请求的信息的方法，例如表单数据、HTTP 请求头等等。HttpServletResponse 除了提供用于指定 HTTP 应答状态（200, 404 等）、应答头（Content-Type, Set-Cookie 等）的方法之外，最重要的是它提供了一个用于向客户端发送数据的 PrintWriter。对于简单的 Servlet 来说，它的大部分工作是通过 println 语句生成向客户端发送的页面。

注意 doGet 和 doPost 抛出两个异常，因此你必须在声明中包含它们。另外，你还必须导入 java.io 包（要用到 PrintWriter 等类）、javax.servlet 包（要用到 HttpServlet 等类）以及 javax.servlet.http 包（要用到 HttpServletRequest 类和 HttpServletResponse 类）。

最后，doGet 和 doPost 这两个方法是由 service 方法调用的，有时你可能需要直接覆盖 service 方法，比如 Servlet 要处理 GET 和 POST 两种请求时。

#### 4.1.2. 输出纯文本的 Servlet

下面是一个 Servlet 的 HelloWorld 例子：

---

```
package com.powerise.edu.servlet;

package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class helloWorldServlet extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request the request send by the client to the server
}
```

```
* @param response the response send by the server to the client
* @throws ServletException if an error occurred
* @throws IOException if an error occurred
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out
        .println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\">");
    out.println("<HTML>");
    out.println(" <HEAD><TITLE>A Servlet</TITLE></HEAD>");
    out.println(" <BODY>");
    out.println(" Hello World!");
    out.println(" </BODY>");
    out.println("</HTML>");
    out.flush();
    out.close();
}

}
```

---

### 4.1.3. Servlet 的编译和安装

不同的 Web 应用服务器上安装 Servlet 的具体细节可能不同, 请参考 Web 服务器文档了解更权威的说明。在本文中, 为了避免同一服务器上不同用户的 Servlet 命名冲突, 我们把所有 Servlet 都放入一个独立的包 com.powerise.edu.servlet 中; 如果你和其他人共用一个服务器, 而且该服务器没有“虚拟服务器”机制来避免这种命名冲突, 那么最好也使用包。把 Servlet 放入了包 com.powerise.edu.servlet 之后, helloWorldServlet.java 实际上是放在 com.powerise.edu.servlet 目录下。

本文以 Tomcat5.0.28 作为 WEB 应用服务器, Eclipse3.2 加 MyEclipse5.0 作为开发工具。  
步骤一: 安装 Eclipse 及对应的 MyEclipse 版本  
步骤二: 配置 Eclipse 插件

选择 Window—>Customize Perspective 选择你要的插件:

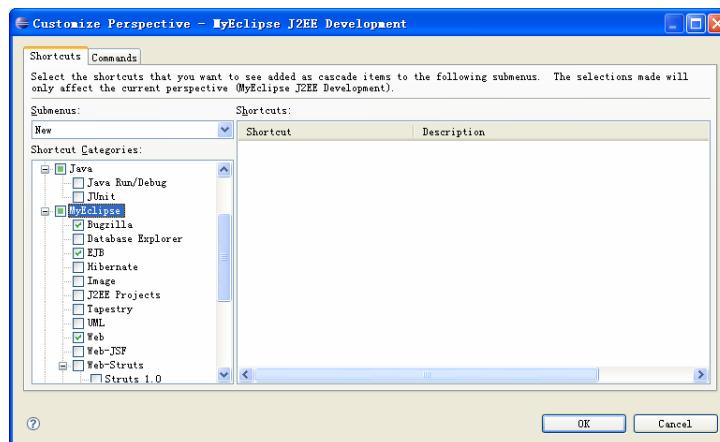


图 4-2: Eclipse 中配置插件

### 步骤三：建立一个 Web Project

选择 File—>Project

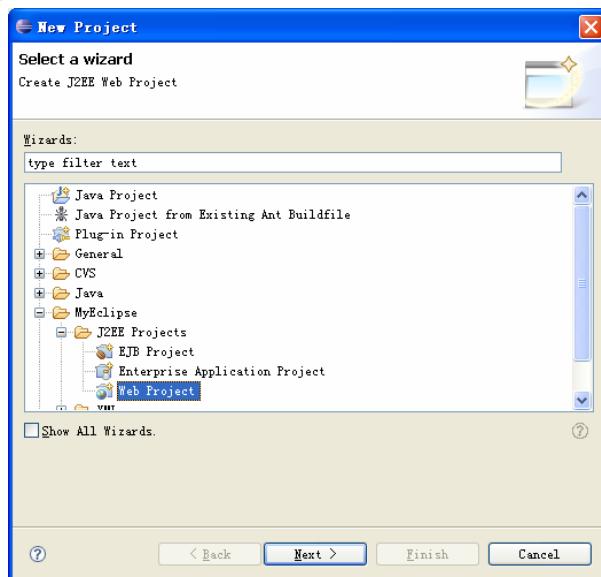


图 4-3: 创建 WEB 工程

根据工程向导完成 Web Project。

### 步骤四：在 SRC 上点右键选择 New—>Servlet

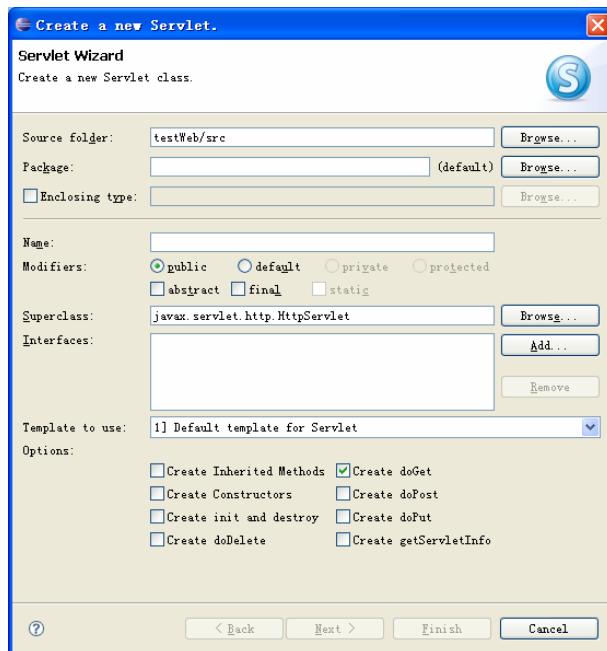


图 4-4 创建一个 Servlet

输入 Servlet 的名字，按照导航，点击完成。Eclipse 已经自动生成好了代码，保存。

#### 步骤五：查看该项目工程中 web.xml 文件

.....

```
<servlet>
    <description>helloWorldServlet</description>
    <display-name>helloWorldServlet</display-name>
    <servlet-name>helloWorldServlet</servlet-name>
    <servlet-class>com.powerise.edu.servlet.helloWorldServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>helloWorldServlet</servlet-name>
    <url-pattern>/servlet/helloWorldServlet</url-pattern>
</servlet-mapping>
.....
```

Eclipse 已经帮你自动配置了以上代码。

#### 步骤六：配置 Tomcat 的 Server.xml 文件（Tomcat 在实验 A 中已经安装好了）

找到 Server.xml 中

```
<Logger className="org.apache.catalina.logger.FileLogger"
        directory="logs" prefix="localhost_log." suffix=".txt"
        timestamp="true"/>
```

在该段下面加入

```
<Context path="" docBase="D:\workspace\testWeb\WebRoot" debug="5" reloadable="true"
crossContext="true"/>
```

#### 步骤七：启动 Tomcat

步骤八：访问 <http://localhost:8080/servlet/helloWorldServlet> 运行效果如下：

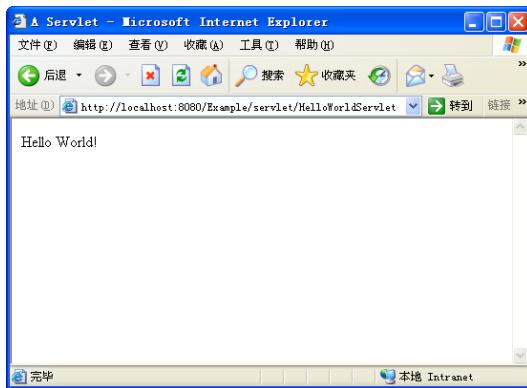


图 4-5

**注意：**如果 TOMCAT 以前部署过别的应用，请删除 D:\Tomcat5028\conf\Catalina 及 D:\Tomcat5028\work\Catalina 目录。

### 补充说明：

通过 `println` 语句输出 HTML 并不方便，根本的解决方法是使用 JavaServer Pages(JSP)。对于标准的 Servlet 来说，由于 Web 页面中有两个部分(DOCTYPE 和 HEAD)一般不会改变，因此可以用工具函数来封装生成这些内容的代码。

虽然大多数主流浏览器都会忽略 DOCTYPE 行，但严格地说 HTML 规范是要求有 DOCTYPE 行的，它有助于 HTML 语法检查器根据所声明的 HTML 版本检查 HTML 文档合法性。在许多 Web 页面中，HEAD 部分只包含<TITLE>。虽然许多有经验的编写者都会在 HEAD 中包含许多 META 标记和样式声明，但这里只考虑最简单的情况。

下面的 Java 方法只接受页面标题为参数，然后输出页面的 DOCTYPE、HEAD、TITLE 部分。程序清单如下：

---

```
package com.powerise.edu.servlet;

public class ServletUtilities {
    public static final String DOCTYPE = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.01 Transitional//EN">";

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

---

下面是应用了 `ServletUtilities` 之后重写 `helloWorldServlet` 类得到的 `helloWorldServlet2`：

---

```
package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class helloWorldServlet2 extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request the request send by the client to the server
     * @param response the response send by the server to the client
     * @throws ServletException if an error occurred
     * @throws IOException if an error occurred
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        System.out.println(ServletUtilities.headWithTitle("Hello World 2"));
        out.println(ServletUtilities.headWithTitle("Hello World 2"));
        out.println("<BODY>");
        out.println(" Hello World 2!");
        out.println(" </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }
}
```

---

## 4.2. 处理表单数据

### 4.2.1. 表单数据概述

如果你曾经使用过 Web 搜索引擎，或者浏览过在线书店、股票价格、机票信息，或许会留意到一些古怪的 URL，比如 “<http://host/path?user=Marty+Hall&origin=bwi&dest=lax>”。这个 URL 中位于问号后面的部分，即 “user=Marty+Hall&origin=bwi&dest=lax”，就是表单数据，这是将 Web 页面数据发送给服务器程序的最常用方法。对于 GET 请求，表单数据附加到 URL 的问号后面（如上例所示）；对于 POST 请求，表单数据用一个单独的行发送给服务器。

以前，从这种形式的数据提取出所需要的表单变量是 CGI 编程中最麻烦的事情之一。首先，GET 请求和 POST 请求的数据提取方法不同：对于 GET 请求，通常要通过

QUERY\_STRING 环境变量提取数据；对于 POST 请求，则一般通过标准输入提取数据。第二，程序员必须负责在“&”符号处截断变量名字-变量值对，再分离出变量名字（等号左边）和变量值（等号右边）。第三，必须对变量值进行 URL 反编码操作。因为发送数据的时候，字母和数字以原来的形式发送，但空格被转换成加号，其他字符被转换成“%XX”形式，其中 XX 是十六进制表示的字符 ASCII（或者 ISO Latin-1）编码值。例如，如果 HTML 表单中名为“users”的域值为“~hall, ~gates, and ~mcnealy”，则实际向服务器发送的数据为“users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy”。最后，即第四个导致解析表单数据非常困难的原因在于，变量值既可能被省略（如“param1=val1 & param2=& param3=val3”），也有可能一个变量拥有一个以上的值，即同一个变量可能出现一次以上（如“param1=val1 & param2=val2 & param1=val3”）。

Java Servlet 的好处之一就在于所有上述解析操作都能够自动完成。只需要简单地调用 HttpServletRequest 的 getParameter 方法、在调用参数中提供表单变量的名字（大小写敏感）即可，而且 GET 请求和 POST 请求的处理方法完全相同。

getParameter 方法的返回值是一个字符串，它是参数中指定的变量名字第一次出现所对应的值经反编码得到得字符串（可以直接使用）。如果指定的表单变量存在，但没有值，getParameter 返回空字符串；如果指定的表单变量不存在，则返回 null。如果表单变量可能对应多个值，可以用 getParameterValues 来取代 getParameter。getParameterValues 能够返回一个字符串数组。

最后，虽然在实际应用中 Servlet 很可能只会用到那些已知名字的表单变量，但在调试环境中，获得完整的表单变量名字列表往往是有用的，利用 getParameterNames 方法可以方便地实现这一点。getParameterNames 返回的是一个 Enumeration，其中的每一项都可以转换为调用 getParameter 的字符串。

### 4.2.2. 实例：读取表单变量

下面是一个简单的例子，它读取三个表单变量 param1、param2 和 param3，并以 HTML 列表的形式列出它们的值。请注意，虽然在发送应答内容之前必须指定应答类型（包括内容类型、状态以及其他 HTTP 头信息），但 Servlet 对何时读取请求内容却没有什么要求。

另外，我们也可以很容易地把 Servlet 做成既能处理 GET 请求，也能够处理 POST 请求，这只需要在 doPost 方法中调用 doGet 方法，或者覆盖 service 方法（service 方法调用 doGet、doPost、doHead 等方法）。在实际编程中这是一种标准的方法，因为它只需要很少的额外工作，却能够增加客户端编码的灵活性。

如果你习惯用传统的 CGI 方法，通过标准输入读取 POST 数据，那么在 Servlet 中也有类似的方法，即在 HttpServletRequest 上调用 getReader 或者 getInputStream，但这种方法对普通的表单变量来说太麻烦。然而，如果是要上载文件，或者 POST 数据是通过专门的客户程序而不是 HTML 表单发送，那么就要用到这种方法。

注意用第二种方法读取 POST 数据时，不能再用 getParameter 来读取这些数据。

ThreeParamsServlet.java:

---

```
package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ThreeParamsServlet extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request
     *          the request send by the client to the server
     * @param response
     *          the response send by the server to the client
     * @throws ServletException
     *          if an error occurred
     * @throws IOException
     *          if an error occurred
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        out.println(ServletUtilities.headWithTitle("读取三个请求参数"));
        out.println("<BODY>");
        out.println("<h1>读取三个请求参数</h1>" + "\n");
        out.println("<li>param1:" + request.getParameter("param1") + "\n");
        out.println("<li>param2:" + request.getParameter("param2") + "\n");
        out.println("<li>param3:" + request.getParameter("param3") + "\n");
        out.println("  </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }

    /**
     * The doPost method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to
     * post.
     *
     * @param request
     *          the request send by the client to the server
     * @param response
     */
}
```

```

    *
        the response send by the server to the client
    * @throws ServletException
    *
        if an error occurred
    * @throws IOException
    *
        if an error occurred
    */

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

}

```

注意：response.setContentType("text/html; charset=gb2312"); 这段代码，当Servlet中有中文字符时，需要设置以下字符编码方式。

在浏览器里面输入：

<http://localhost:8080/servlet/ThreeParamsServlet?param1=p1&param2=p2&param3=p3> 看看输出是什么？？



图 4-6

### 4.2.3. 实例：输出所有表单数据

下面这个例子寻找表单所发送的所有变量名字，并把它们放入表格中，没有值或者有多个值的变量都突出显示。

首先，程序通过 HttpServletRequest 的 getParameterNames 方法得到所有的变量名字，getParameterNames 返回的是一个 Enumeration。接下来，程序用循环遍历这个 Enumeration，通过.hasMoreElements 确定何时结束循环，利用 nextElement 得到 Enumeration 中的各个项。由于 nextElement 返回的是一个 Object，程序把它转换成字符串后再用这个字符串来调用 getParameterValues。

getParameterValues 返回一个字符串数组，如果这个数组只有一个元素且等于空字符串，说明这个表单变量没有值，Servlet 以斜体形式输出 “No Value”；如果数组元素个数大于 1，说明这个表单变量有多个值，Servlet 以 HTML 列表形式输出这些值；其他情况下 Servlet 直接把变量值放入表格。

ShowParametersServlet.java

---

注意，ShowParametersServlet.java 用到了前面介绍过的 ServletUtilities.java。

```
package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Enumeration;

public class ShowParametersServlet extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request the request send by the client to the server
     * @param response the response send by the server to the client
     * @throws ServletException if an error occurred
     * @throws IOException if an error occurred
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String title = "读取所有请求参数";
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        out.println(ServletUtilities.headWithTitle(title)
            + "<BODY BGCOLOR=#FDF5E6>\n" + "<H1 ALIGN=CENTER>" +
        title
            + "</H1>\n" + "<TABLE BORDER=1 ALIGN=CENTER>\n"
            + "<TR BGCOLOR=#FFAD00>\n" + "<TH>参数名字<TH>参数值");
        Enumeration paramNames = request.getParameterNames();
        while (paramNames.hasMoreElements()) {
            String paramName = (String) paramNames.nextElement();
            out.println("<TR><TD>" + paramName + "\n<TD>");
            String[] paramValues = request.getParameterValues(paramName);
            if (paramValues.length == 1) {
                String paramValue = paramValues[0];
                if (paramValue.length() == 0)
                    out.print("<I>No Value</I>");
                else
                    out.print(paramValue);
            }
        }
    }
}
```

```

        } else {
            out.println("<UL>");
            for (int i = 0; i < paramValues.length; i++) {
                out.println("<LI>" + paramValues[i]);
            }
            out.println("</UL>");
        }
    }

out.println("</TABLE>\n</BODY></HTML>");

}

/***
 * The doPost method of the servlet. <br>
 *
 * This method is called when a form has its tag value method equals to post.
 *
 * @param request the request send by the client to the server
 * @param response the response send by the server to the client
 * @throws ServletException if an error occurred
 * @throws IOException if an error occurred
 */
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    this doGet(request, response);
}
}

```

在浏览器输入：

http://localhost:8080/Example/servlet>ShowParametersServlet?test=test&test1=test1  
效果如下：

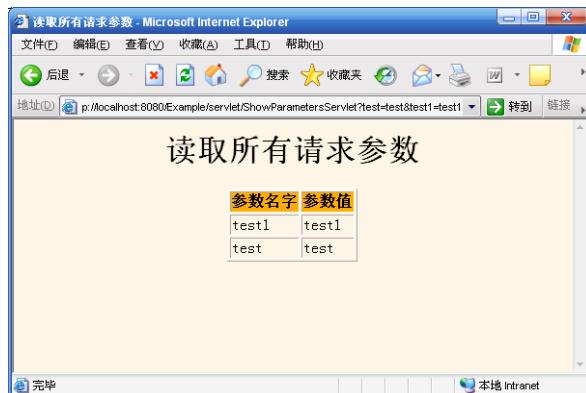


图 4-7

#### 4.2.4. 实例： 测试表单

下面是向上述 Servlet 发送数据的表单 PostForm.html。就像所有包含密码输入域的表单

一样，该表单用 POST 方法发送数据。我们可以看到，在 Servlet 中同时实现 doGet 和 doPost 这两种方法为表单制作带来了方便。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>示例表单</title>
<meta http-equiv="Content-type" content="text/html; charset=gb2312">
</head>
<body>
<h1>用 POST 方法发送数据的表单</h1>
<form action="/servlet>ShowParametersServlet" method="post">
Item Number:<input name="itemNum" type="text"><br>
Quantity:<input name="quantity" type="text"><br>
Price Each:<input name="Price" type="text"><br>
First name:<input name="firstname" type="text"><br>
Middle Initial:<input name="lastname" type="initial"><br>
<textarea name="address" rows=3 cols=40></textarea><br>
Credit Card:<br>
<input type="radio" name="cardtype"
value="Visa">Visa<br>
<input type="radio" name="cardtype"
value="Master Card">Master Card<br>
<input type="radio" name="cardtype"
value="Amex">American Express<br>
<input type="radio" name="cardtype"
value="Discover">Discover<br>
<input type="radio" name="cardtype"
value="Java SmartCard">Java SmartCard<br>
Credit Card Number:
<input type="PASSWORD" name="cardNum"><br>
Repeat Credit Card Number:
<input type="PASSWORD" name="cardNum"><br><br>
<CENTER>
<input type="SUBMIT" value="Submit Order">
</CENTER>
</form>
</body>
</html>
```

---

在浏览器中输入： <http://localhost:8080/Example/PostForm.html>， 显示如下：

图 4-8

输入数据选择提交：

参数名字	参数值
cardnum	sdf dssdf
Price	sdfdsf
quantity	dfsdf
firstname	dfsdfsd
lastname	fdfsdf
initial	fds
cardtype	Master Card
itemnum	dfsdf
address	sdfdsfdf

图 4-9

## 4.3. 读取 http 请求

### 4.3.1. http 请求头概述

HTTP 客户程序（例如浏览器），向服务器发送请求的时候必须指明请求类型（一般是 GET 或者 POST）。如有必要，客户程序还可以选择发送其他的请求头。大多数请求头并不是必需的，但 Content-Length 除外。对于 POST 请求来说 Content-Length 必须出现。

下面是一些最常见的请求头：

**Accept:** 浏览器可接受的 MIME (Multipurpose Internet Mail Extensions 多用途网际邮件扩充协议) 类型。

**Accept-Charset:** 浏览器可接受的字符集。

**Accept-Encoding:** 浏览器能够进行解码的数据编码方式，比如 gzip。Servlet 能够向支持 gzip 的浏览器返回经 gzip 编码的 HTML 页面。许多情形下这可以减少 5 到 10 倍的下载时间。

**Accept-Language:** 浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到。

**Authorization:** 授权信息，通常出现在对服务器发送的 WWW-Authenticate 头的应答中。

**Connection:** 表示是否需要持久连接。如果 Servlet 看到这里的值为“Keep-Alive”，或者看到请求使用的是 HTTP 1.1 (HTTP 1.1 默认进行持久连接)，它就可以利用持久连接的优点，当页面包含多个元素时（例如 Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet 需要在应答中发送一个 Content-Length 头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。

**Content-Length:** 表示请求消息正文的长度。

**Cookie:** 这是最重要的请求头信息之一，参见后面《Cookie 处理》一章中的讨论。

**From:** 请求发送者的 email 地址，由一些特殊的 Web 客户程序使用，浏览器不会用到它。

**Host:** 初始 URL 中的主机和端口。

**If-Modified-Since:** 只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回 304 “Not Modified” 应答。

**Pragma:** 指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝。

**Referer:** 包含一个 URL，用户从该 URL 代表的页面出发访问当前请求的页面。

**User-Agent:** 浏览器类型，如果 Servlet 返回的内容与浏览器类型有关则该值非常有用。

**UA-Pixels, UA-Color, UA-OS, UA-CPU:** 由某些版本的 IE 浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和 CPU 类型。

有关 HTTP 头完整、详细的说明，请参见 <http://www.w3.org/Protocols/> 的 HTTP 规范。

### 4.3.2. 实例：在 Servlet 中读取请求头

在 Servlet 中读取 HTTP 头是非常方便的，只需要调用 HttpServletRequest 的 getHeader 方法即可。如果客户请求中提供了指定的头信息，getHeader 返回对应的字符串；否则，返回 null。部分头信息经常要用到，它们有专用的访问方法：getCookies 方法返回 Cookie 头的内容，经解析后存放在 Cookie 对象的数组中，请参见后面有关 Cookie 章节的讨论；getAuthType 和 getRemoteUser 方法分别读取 Authorization 头中的一部分内容；getDateHeader 和 getIntHeader 方法读取指定的头，然后返回日期值或整数值。

除了读取指定的头之外，利用 getHeaderNames 还可以得到请求中所有头名字的一个 Enumeration 对象。

最后，除了查看请求头信息之外，我们还可以从请求主命令行获得一些信息。getMethod 方法返回请求方法，请求方法通常是 GET 或者 POST，但也有可能是 HEAD、PUT 或者 DELETE。getRequestURI 方法返回 URI (URI 是 URL 的从主机和端口之后到表单数据之前的那一部分)。getRequestProtocol 返回请求命令的第三部分，一般是“HTTP/1.0”或者“HTTP/1.1”。

### 4.3.3. 实例：输出所有的请求头

下面的 Servlet 实例把所有接收到的请求头和它的值以表格的形式输出。另外，该 Servlet 还会输出主请求命令的三个部分：请求方法，URI，协议/版本。

ShowRequestHeadersServlet.java:

---

```
package com.powerise.edu.servlet;
```

```
import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Enumeration;

public class ShowRequestHeadersServlet extends HttpServlet {
    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request
     *          the request send by the client to the server
     * @param response
     *          the response send by the server to the client
     * @throws ServletException
     *          if an error occurred
     * @throws IOException
     *          if an error occurred
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        String title = "显示所有请求头";
        out.println(ServletUtilities.headWithTitle(title)
            + "<BODY BGCOLOR=#FDF5E6>\n" + "<H1 ALIGN=CENTER>" +
            title
            + "</H1>\n" + "<B>Request Method: </B>" + request.getMethod()
            + "<BR>\n" + "<B>Request URI: </B>" + request.getRequestURI()
            + "<BR>\n" + "<B>Request Protocol: </B>" +
            request.getProtocol() + "<BR><BR>\n"
            + "<TABLE BORDER=1 ALIGN=CENTER>\n"
            + "<TR BGCOLOR=#FFAD00>\n"
            + "<TH>Header Name<TH>Header Value");
        Enumeration headerNames = request.getHeaderNames();
        while (headerNames.hasMoreElements()) {
            String headerName = (String) headerNames.nextElement();
            out.println("<TR><TD>" + headerName);
            out.println(" <TD>" + request.getHeader(headerName));
        }
        out.println("</TABLE>\n</BODY></HTML>");
    }
}
```

```
/**  
 * The doPost method of the servlet. <br>  
 *  
 * This method is called when a form has its tag value method equals to  
 * post.  
 *  
 * @param request  
 *          the request send by the client to the server  
 * @param response  
 *          the response send by the server to the client  
 * @throws ServletException  
 *          if an error occurred  
 * @throws IOException  
 *          if an error occurred  
 */  
  
public void doPost(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    this doGet(request, response);  
}  
}
```



冬 4-10

## 4.4. 访问 CGI 变量

#### 4.4.1. CGI 概述

**定义：**CGI(Common Gateway Interface)是HTTP服务器与你的或其它机器上的程序进行“交谈”的一种工具，其程序须运行在网络服务器上。

**功能:** 绝大多数的 CGI 程序被用来解释处理来自表单的输入信息，并在服务器产生相

应的处理，或将相应的信息反馈给浏览器。CGI 程序使网页具有交互功能。

**运行环境：**CGI 程序在 UNIX 操作系统上 CERN 或 NCSA 格式的服务器上运行。在其它操作系统（如：windows）的服务器上也广泛地使用 CGI 程序，同时它也适用于各种类型机器。

**CGI 处理步骤：**(1)通过 Internet 把用户请求送到服务器。(2)服务器接收用户请求并交给 CGI 程序处理。(3)CGI 程序把处理结果传送给服务器。(4)服务器把结果送回到用户。

CGI 变量汇集了各种有关请求的信息：

- 部分来自 HTTP 请求命令和请求头，例如 Content-Length 头；
- 部分来自 Socket 本身，例如主机的名字和 IP 地址；
- 也有部分与服务器安装配置有关，例如 URL 到实际路径的映射。

#### 4.4.2. 标准 CGI 变量的 Servlet 等价表示

表 4-1 标准 CGI 变量与 Servlet

CGI 变量	含义	从 doGet 或 doPost 访问
AUTH_TYPE	如果提供了 Authorization 头，这里指定了具体的模式（basic 或者 digest）。	request.getAuthType()
CONTENT_LENGTH	只用于 POST 请求，表示所发送数据的字节数。严格地讲，等价的表达方式应该是 String.valueOf(request.getContentLength())（返回一个字符串）。但更常见的是用 request.getContentLength() 返回含义相同的整数。	request.getContentLength()
CONTENT_TYPE	如果指定的话，表示后面所跟数据的类型。	request.getContentType()
DOCUMENT_ROOT	与 http://host/对应的路径。注意低版本 Servlet 规范中的等价表达方式是 request.getRealPath("/")。	getServletContext().getRealPath("/")
HTTP_XXX_YYY	访问任意 HTTP 头。	request.getHeader("Xxx-Yyy")
PATH_INFO	URL 中的附加路径信息，即 URL 中 Servlet 路径之后、查询字符串之前的那部分。	request.getPathInfo()
PATH_TRANSLATED	映射到服务器实际路径之后的路径信息。	request.getPathTranslated()
QUERY_STRING	这是字符串形式的附加到 URL 后面的查询字符串，数据仍旧是 URL 编码的。在 Servlet 中很少需要用到未经解码的数据，一般使用 getParameter 访问各个参数。	request.getQueryString()
REMOTE_ADDR	发出请求的客户机的 IP 地址。	request.getRemoteAddr()
REMOTE_HOST	发出请求的客户机的完整的域名，	request.getRemoteHost()

	如 java.sun.com。如果不能确定该域名，则返回 IP 地址。	
REMOTE_USER	如果提供了 Authorization 头，则代表其用户部分。它代表发出请求的用户名。	request.getRemoteUser()
REQUEST_METHOD	请求类型。	通常是 GET 或者 POST。但偶尔也会出现 HEAD, PUT, DELETE, OPTIONS, 或者 TRACE. request.getMethod()
SCRIPT_NAME URL	中调用 Servlet 的那一部分，不包含附加路径信息和查询字符串。 request.getServletPath() SERVER_NAME Web 服务器名字。	request.getServerName()
SERVER_PORT	服务器监听的端口。	严格地说，等价表达应该是返回字符串的 String.valueOf(request.getServerPort())。但经常使用返回整数值的 request.getServerPort()。
SERVER_PROTOCOL	请求命令中的协议名字和版本（即 HTTP/1.0 或 HTTP/1.1）。	request.getProtocol()
SERVER_SOFTWARE	Servlet 引擎的名字和版本。	getServletContext().getServerInfo()

#### 4.4.3. 实例：读取 CGI 变量

下面这个 Servlet 创建一个表格，显示除了 HTTP\_XXX\_YYY 之外的所有 CGI 变量。  
HTTP\_XXX\_YYY 是 HTTP 请求头信息，请参见上一节介绍。

---

```

package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ShowCGIVariables extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     */

```

```
* @param request
*           the request send by the client to the server
* @param response
*           the response send by the server to the client
* @throws ServletException
*           if an error occurred
* @throws IOException
*           if an error occurred
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();
    String[][] variables = {
        { "AUTH_TYPE", request.getAuthType() },
        { "CONTENT_LENGTH", String.valueOf(request.getContentLength()) },
        { "CONTENT_TYPE", request.getContentType() },
        { "DOCUMENT_ROOT", getServletContext().getRealPath("/") },
        { "PATH_INFO", request.getPathInfo() },
        { "PATH_TRANSLATED", request.getPathTranslated() },
        { "QUERY_STRING", request.getQueryString() },
        { "REMOTE_ADDR", request.getRemoteAddr() },
        { "REMOTE_HOST", request.getRemoteHost() },
        { "REMOTE_USER", request.getRemoteUser() },
        { "REQUEST_METHOD", request.getMethod() },
        { "SCRIPT_NAME", request.getServletPath() },
        { "SERVER_NAME", request.getServerName() },
        { "SERVER_PORT", String.valueOf(request.getServerPort()) },
        { "SERVER_PROTOCOL", request.getProtocol() },
        { "SERVER_SOFTWARE", getServletContext().getServerInfo() } };
    String title = "显示 CGI 变量";
    out.println(ServletUtilities.headWithTitle(title)
            + "<BODY BGCOLOR=#FDF5E6>\n" + "<H1 ALIGN=CENTER>" +
            title
            + "</H1>\n" + "<TABLE BORDER=1 ALIGN=CENTER>\n"
            + "<TR BGCOLOR=#FFAD00>\n"
            + "<TH>CGI Variable Name<TH>Value");
    for (int i = 0; i < variables.length; i++) {
        String varName = variables[i][0];
        String varValue = variables[i][1];
        if (varValue == null)
            varValue = "<I>Not specified</I>";
        out.println("<TR><TD>" + varName + "<TD>" + varValue);
    }
}
```

```

        out.println("</TABLE></BODY></HTML>");

    }

    /**
     * The doPost method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to
     * post.
     *
     * @param request
     *          the request send by the client to the server
     * @param response
     *          the response send by the server to the client
     * @throws ServletException
     *          if an error occurred
     * @throws IOException
     *          if an error occurred
     */
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        this doGet(request, response);
    }
}

```

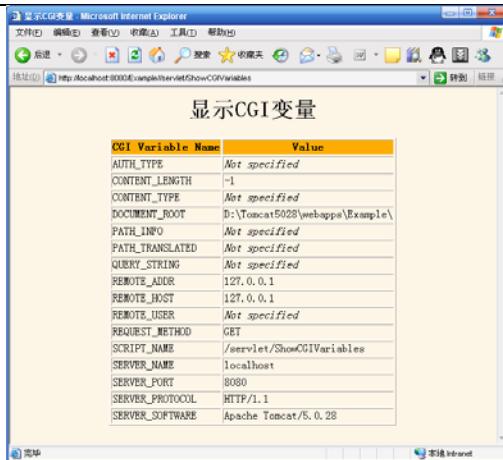


图 4-11

## 4.5. http 应答状态

### 4.5.1. 状态代码概述

Web 服务器响应浏览器或其他客户程序的请求时，其应答一般由以下几个部分组成：一个状态行，几个应答头，一个空行，内容文档。下面是一个最简单的应答：

HTTP/1.1 200 OK

Content-Type: text/plain

### Hello World

状态行包含 HTTP 版本、状态代码、与状态代码对应的简短说明信息。在大多数情况下，除了 Content-Type 之外的所有应答头都是可选的。但 Content-Type 是必需的，它描述的是后面文档的 MIME 类型。虽然大多数应答都包含一个文档，但也有一些不包含，例如对 HEAD 请求的应答永远不会附带文档。有许多状态代码实际上用来标识一次失败的请求，这些应答也不包含文档（或只包含一个简短的错误信息说明）。

Servlet 可以利用状态代码来实现许多功能。例如，可以把用户重定向到另一个网站；可以指示出后面的文档是图片、PDF 文件或 HTML 文件；可以告诉用户必须提供密码才能访问文档；等等。这一部分我们将具体讨论各种状态代码的含义以及利用这些代码可以做些什么。

### 4.5.2. 设置状态代码

如前所述，HTTP 应答状态行包含 HTTP 版本、状态代码和对应的状态信息。由于状态信息直接和状态代码相关，而 HTTP 版本又由服务器确定，因此需要 Servlet 设置的只有一个状态代码。

Servlet 设置状态代码一般使用 HttpServletResponse 的 setStatus 方法。setStatus 方法的参数是一个整数（即状态代码），不过为了使得代码具有更好的可读性，可以用 HttpServletResponse 中定义的常量来避免直接使用整数。这些常量根据 HTTP 1.1 中的标准状态信息命名，所有的名字都加上了 SC 前缀（Status Code 的缩写）并大写，同时把空格转换成了下划线。也就是说，与状态代码 404 对应的状态信息是“Not Found”，则 HttpServletResponse 中的对应常量名字为 SC\_NOT\_FOUND。但有两个例外：和状态代码 302 对应的常量根据 HTTP 1.0 命名，而 307 没有对应的常量。

设置状态代码并非总是意味着不要再返回文档。例如，虽然大多数服务器返回 404 应答时会输出简单的“File Not Found”信息，但 Servlet 也可以定制这个应答。不过，定制应答时应当在通过 PrintWriter 发送任何内容之前先调用 response.setStatus。

虽然设置状态代码一般使用的是 response.setStatus(int) 方法，但为了简单起见，HttpServletResponse 为两种常见的情形提供了专用方法：sendError 方法生成一个 404 应答，同时生成一个简短的 HTML 错误信息文档；sendRedirect 方法生成一个 302 应答，同时在 Location 头中指示新文档的 URL。

### 4.5.3. HTTP 状态代码及其含义

应当谨慎地使用那些只有 HTTP 1.1 支持的状态代码，因为许多浏览器还只能够支持 HTTP 1.0。如果你使用了 HTTP 1.1 特有的状态代码，最好能够检查请求的 HTTP 版本号（通过 HttpServletRequest 的 getProtocol 方法）。

下表显示了常见的 HTTP 1.1 状态代码以及它们对应的状态信息和含义。

表 4-2HTTP1.1 状态代码含义

状态代码	状态信息	含义
100	Continue	初始的请求已经接受，客户应当继续发送请求的其余部分。（HTTP 1.1 新）
101	Switching Protocols	服务器将遵从客户的请求转换到另外一种协议（HTTP 1.1 新）
200	OK	一切正常，对 GET 和 POST 请求的应答文档跟在后面。如果不用 setStatus 设置状态代码，Servlet 默认使用 202 状态代码。
201	Created	服务器已经创建了文档，Location 头给出了它的 URL。

202	Accepted	已经接受请求，但处理尚未完成。
203	Non-Authoritative Information	文档已经正常地返回，但一些应答头可能不正确，因为使用的是文档的拷贝（HTTP 1.1 新）。
204	No Content	没有新文档，浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而 Servlet 可以确定用户文档足够新，这个状态代码是很有用的。
205	Reset Content	没有新的内容，但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容（HTTP 1.1 新）。
206	Partial Content	客户发送了一个带有 Range 头的 GET 请求，服务器完成了它（HTTP 1.1 新）。
300	Multiple Choices	客户请求的文档可以在多个位置找到，这些位置已经在返回的文档内列出。如果服务器要提出优先选择，则应该在 Location 应答头指明。
301	Moved Permanently	客户请求的文档在其他地方，新的 URL 在 Location 头中给出，浏览器应该自动地访问新的 URL。
302	Found	<p>类似于 301，但新的 URL 应该被视为临时性的替代，而不是永久性的。注意，在 HTTP1.0 中对应的状态信息是“Moved Temporarily”，而 <code>HttpServletResponse</code> 中相应的常量是 <code>SC_MOVED_TEMPORARILY</code>，而不是 <code>SC_FOUND</code>。出现该状态代码时，浏览器能够自动访问新的 URL，因此它是一个很有用的状态代码。为此，Servlet 提供了一个专用的方法，即 <code>sendRedirect</code>。使用 <code>response.sendRedirect(url)</code> 比使用 <code>response.setStatus(response.SC_MOVED_TEMPORARILY)</code> 和 <code>response.setHeader("Location",url)</code> 更好。这是因为：</p> <p>首先，代码更加简洁。</p> <p>第二，使用 <code>sendRedirect</code>，Servlet 会自动构造一个包含新链接的页面（用于那些不能自动重定向的老式浏览器）。</p> <p>最后，<code>sendRedirect</code> 能够处理相对 URL，自动把它们转换成绝对 URL。</p> <p>注意这个状态代码有时候可以和 301 替换使用。例如，如果浏览器错误地请求 <code>http://host/~user</code>（缺少了后面的斜杠），有的服务器返回 301，有的则返回 302。</p> <p>严格地说，我们只能假定只有当原来的请求是 GET 时浏览器才会自动重定向。请参见 307。</p>
303	See Other	类似于 301/302，不同之处在于，如果原来的请求是 POST，Location 头指定的重定向目标文档应该通过 GET 提取（HTTP 1.1 新）。
304	Not Modified	客户端有缓冲的文档并发出一个条件性的请求（一般是提供 If-Modified-Since 头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。
305	Use Proxy	客户请求的文档应该通过 Location 头所指明的代理服务器提取（HTTP 1.1 新）。
307	Temporary Redirect	和 302（Found）相同。许多浏览器会错误地响应 302 应答进行重定向，即使原来的请求是 POST，即使它实际上只能在

		POST 请求的应答是 303 时才能重定向。由于这个原因，HTTP 1.1 新增了 307，以便更加清除地区分几个状态代码：当出现 303 应答时，浏览器可以跟随重定向的 GET 和 POST 请求；如果是 307 应答，则浏览器只能跟随对 GET 请求的重定向。注意， <code>HttpServletResponse</code> 中没有为该状态代码提供相应的常量。（HTTP 1.1 新）
<b>400</b>	<b>Bad Request</b>	请求出现语法错误。
<b>401</b>	<b>Unauthorized</b>	客户试图未经授权访问受密码保护的页面。应答中会包含一个 <b>WWW-Authenticate</b> 头，浏览器据此显示用户名/密码对话框，然后在填写合适的 <b>Authorization</b> 头后再次发出请求。
<b>403</b>	<b>Forbidden</b>	资源不可用。服务器理解客户的请求，但拒绝处理它。通常由于服务器上文件或目录的权限设置导致。
<b>404</b>	<b>Not Found</b>	无法找到指定位置的资源。这也是一个常用的应答， <code>HttpServletResponse</code> 专门提供了相应的方法： <code>sendError(message)</code> 。
405	Method Not Allowed	请求方法（GET、POST、HEAD、DELETE、PUT、TRACE 等）对指定的资源不适用。（HTTP 1.1 新）
406	Not Acceptable	指定的资源已经找到，但它的 MIME 类型和客户在 <code>Accpet</code> 头中所指定的不兼容（HTTP 1.1 新）。
407	Proxy Authentication Required	类似于 401，表示客户必须先经过代理服务器的授权。（HTTP 1.1 新）
408	Request Timeout	在服务器许可的等待时间内，客户一直没有发出任何请求。客户可以在以后重复同一请求。（HTTP 1.1 新）
409	Conflict	通常和 PUT 请求有关。由于请求和资源的当前状态相冲突，因此请求不能成功。（HTTP 1.1 新）
410	Gone	所请求的文档已经不再可用，而且服务器不知道应该重定向到哪一个地址。它和 404 的不同在于，返回 407 表示文档永久地离开了指定的位置，而 404 表示由于未知的原因文档不可用。（HTTP 1.1 新）
411	Length Required	服务器不能处理请求，除非客户发送一个 <code>Content-Length</code> 头。（HTTP 1.1 新）
412	Precondition Failed	请求头中指定的一些前提条件失败（HTTP 1.1 新）。
413	Request Entity Too Large	目标文档的大小超过服务器当前愿意处理的大小。如果服务器认为自己能够稍后再处理该请求，则应该提供一个 <code>Retry-After</code> 头（HTTP 1.1 新）。
414	Request URI Too Long	URI 太长（HTTP 1.1 新）。
<b>500</b>	<b>Internal Server Error</b>	服务器遇到了意料不到的情况，不能完成客户的请求。
501	Not Implemented	服务器不支持实现请求所需要的功能。例如，客户发出了一个服务器不支持的 PUT 请求。
502	Bad Gateway	服务器作为网关或者代理时，为了完成请求访问下一个服务

		器，但该服务器返回了非法的应答。
503	Service Unavailable	服务器由于维护或者负载过重未能应答。例如，Servlet 可能在数据库连接池已满的情况下返回 503。服务器返回 503 时可以提供一个 Retry-After 头。
504	Gateway Timeout	由作为代理或网关的服务器使用，表示不能及时地从远程服务器获得应答。（HTTP 1.1 新）
505	HTTP Version Not Supported	服务器不支持请求中所指明的 HTTP 版本。（HTTP 1.1 新）

注意：上表中粗体标识的部分比较常用。

#### 4.5.4. 实例：访问多个搜索引擎

下面这个例子用到了除 200 之外的另外两个常见状态代码：302 和 404。302 通过 sendRedirect 方法设置，404 通过 sendError 方法设置。

在这个例子中，首先出现的 HTML 表单用来选择搜索引擎、搜索字符串、每页显示的搜索结果数量。表单提交后，Servlet 提取这三个变量，按照所选择的搜索引擎的要求构造出包含这些变量的 URL，然后把用户重定向到这个 URL。如果用户不能正确地选择搜索引擎，或者利用其他表单发送了一个不认识的搜索引擎名字，则返回一个提示搜索引擎找不到的 404 页面。

注意：这个 Servlet 要用到后面给出的 SearchSpec 类，SearchSpec 的功能是构造适合不同搜索引擎的 URL。

SearchEngines.java:

---

```

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.net.URLEncoder;

public class SearchEngines extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request
     *          the request send by the client to the server
     * @param response
     *          the response send by the server to the client
     * @throws ServletException
     *          if an error occurred
     * @throws IOException
     */
    
```

```
*           if an error occurred
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // getParameter 自动解码 URL 编码的查询字符串。由于我们
    // 要把查询字符串发送给另一个服务器，因此再次使用
    // URLEncoder 进行 URL 编码
    String searchString = URLEncoder.encode(request
        .getParameter("searchString"));
    String numResults = request.getParameter("numResults");
    String searchEngine = request.getParameter("searchEngine");
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    for (int i = 0; i < commonSpecs.length; i++) {
        SearchSpec searchSpec = commonSpecs[i];
        if (searchSpec.getName().equals(searchEngine)) {
            String url = response.encodeURL(searchSpec.makeURL(
                searchString, numResults));
            response.sendRedirect(url);
            return;
        }
    }
    response.sendError(response.SC_NOT_FOUND,
        "No recognized search engine specified.");
}

/**
 * The doPost method of the servlet. <br>
 *
 * This method is called when a form has its tag value method equals to
 * post.
 *
 * @param request
 *           the request send by the client to the server
 * @param response
 *           the response send by the server to the client
 * @throws ServletException
 *           if an error occurred
 * @throws IOException
 *           if an error occurred
 */
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    this.doGet(request, response);
}
```

```
}

SearchSpec.java:
package com.powerise.edu.servlet;

public class SearchSpec {
    private String name, baseURL, numResultsSuffix;

    private static SearchSpec[] commonSpecs = {
        new SearchSpec("google", "http://www.google.com/search?q=", "&num="),
        new SearchSpec("baidu", "http://www.baidu.com/s?wd=", "&cl="),
        new SearchSpec("sohu", "http://www.sogou.com/sohu?query=", "&pid="),
        new SearchSpec("sina", "http://iask.com/s?k=", "&cl=&fsh_s=") };

    public SearchSpec(String name, String baseURL, String numResultsSuffix) {
        this.name = name;
        this.baseURL = baseURL;
        this.numResultsSuffix = numResultsSuffix;
    }

    public String makeURL(String searchString, String numResults) {
        return (baseURL + searchString + numResultsSuffix + numResults);
    }

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

    public static SearchSpec[] getCommonSpecs() {
        return (commonSpecs);
    }
}
```

下面是调用上述 Servlet 的 HTML 表单。SearchEngines.htm:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<HTML>
<HEAD>
<TITLE>访问多个搜索引擎</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<FORM ACTION="/servlet/SearchEngines">
<CENTER>
    搜索关键字:
    <INPUT TYPE="TEXT" NAME="searchString">
```

```

<BR>
每页显示几个查询结果:
<INPUT TYPE="TEXT" NAME="numResults"
VALUE=10 SIZE=3>
<BR>
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="google">
Google |
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="baidu">
百度 |
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="sohu">
搜狐 |
<INPUT TYPE="RADIO" NAME="searchEngine"
VALUE="sina">
新浪<BR>
<INPUT TYPE="SUBMIT" VALUE="Search">
</CENTER>
</FORM>
</BODY>
</HTML>

```



图 4-12



图 4-13

## 4.6. 设置 HTTP 应答头

### 4.6.1. HTTP 应答头概述

Web 服务器的 HTTP 应答一般由以下几项构成：一个状态行，一个或多个应答头，一

个空行，内容文档。设置 HTTP 应答头往往和设置状态行中的状态代码结合起来。例如，有好几个表示“文档位置已经改变”的状态代码都伴随着一个 Location 头，而 401(Unauthorized) 状态代码则必须伴随一个 WWW-Authenticate 头。

然而，即使在没有设置特殊含义的状态代码时，指定应答头也是很有用的。应答头可以用来完成：设置 Cookie，指定修改日期，指示浏览器按照指定的间隔刷新页面，声明文档的长度以便利用持久 HTTP 连接，……等等许多其他任务。

设置应答头最常用的方法是 `HttpServletResponse` 的 `setHeader`，该方法有两个参数，分别表示应答头的名字和值。和设置状态代码相似，设置应答头应该在发送任何文档内容之前进行。

`setDateHeader` 方法和 `setIntHeader` 方法专门用来设置包含日期和整数值的应答头，前者避免了把 Java 时间转换为 GMT 时间字符串的麻烦，后者则避免了把整数转换为字符串的麻烦。

`HttpServletResponse` 还提供了许多设置常见应答头的简便方法，如下所示：

`setContentType`: 设置 Content-Type 头。大多数 Servlet 都要用到这个方法。

`setContentLength`: 设置 Content-Length 头。对于支持持久 HTTP 连接的浏览器来说，这个函数是很有用的。

`addCookie`: 设置一个 Cookie (Servlet API 中没有 `setCookie` 方法，因为应答往往包含多个 Set-Cookie 头)。

另外，如上节介绍，`sendRedirect` 方法设置状态代码 302 时也会设置 Location 头。

## 4.6.2. 常见应答头及其含义

有关 HTTP 头详细和完整的说明，请参见 <http://www.w3.org/Protocols/> 规范。

表 4-3 常见应答头

应答头	说明
Allow	服务器支持哪些请求方法 (如 GET、POST 等)。
Content-Encoding	文档的编码 (Encode) 方法。只有在解码之后才可以得到 Content-Type 头指定的内容类型。利用 gzip 压缩文档能够显著地减少 HTML 文档的下载时间。Java 的 GZIPOutputStream 可以很方便地进行 gzip 压缩，但只有 Unix 上的 Netscape 和 Windows 上的 IE 4、IE 5 才支持它。因此，Servlet 应该通过查看 Accept-Encoding 头 (即 <code>request.getHeader("Accept-Encoding")</code> ) 检查浏览器是否支持 gzip，为支持 gzip 的浏览器返回经 gzip 压缩的 HTML 页面，为其他浏览器返回普通页面。
Content-Length	表示内容长度。只有当浏览器使用持久 HTTP 连接时才需要这个数据。如果你想要利用持久连接的优势，可以把输出文档写入 <code>ByteArrayOutputStream</code> ，完成后查看其大小，然后把该值放入 Content-Length 头，最后通过 <code>byteArrayStream.writeTo(response.getOutputStream())</code> 发送内容。
Content-Type	表示后面的文档属于什么 MIME 类型。Servlet 默认为 <code>text/plain</code> ，但通常需要显式地指定为 <code>text/html</code> 。由于经常要设置 Content-Type，因此 <code>HttpServletResponse</code> 提供了一个专用的方法 <code>setContentTyep</code> 。 Date 当前的 GMT 时间。你可以用 <code>setDateHeader</code> 来设置这个头以避免转换时间格式的麻烦。
Expires	应该在什么时候认为文档已经过期，从而不再缓存它？

Last-Modified	文档的最后改动时间。客户可以通过 If-Modified-Since 请求头提供一个日期，该请求将被视为一个条件 GET，只有改动时间迟于指定时间的文档才会返回，否则返回一个 304（Not Modified）状态。Last-Modified 也可用 setDateHeader 方法来设置。
Location	表示客户应当到哪里去提取文档。Location 通常不是直接设置的，而是通过 HttpServletResponse 的 sendRedirect 方法，该方法同时设置状态代码为 302。
Refresh	<p>表示浏览器应该在多少时间之后刷新文档，以秒计。除了刷新当前文档之外，你还可以通过 setHeader("Refresh", "5; URL=http://host/path") 让浏览器读取指定的页面。</p> <p>这种功能通常是通过设置 HTML 页面 HEAD 区的 &lt;META HTTP-EQUIV="Refresh" CONTENT="5;URL=http://host/path" &gt; 实现，这是因为，自动刷新或重定向对于那些不能使用 CGI 或 Servlet 的 HTML 编写者十分重要。但是，对于 Servlet 来说，直接设置 Refresh 头更加方便。</p> <p>Refresh 的意思是“N 秒之后刷新本页面或访问指定页面”，而不是“每隔 N 秒刷新本页面或访问指定页面”。因此，连续刷新要求每次都发送一个 Refresh 头，而发送 204 状态代码则可以阻止浏览器继续刷新，不管是使用 Refresh 头还是&lt;META HTTP-EQUIV="Refresh" ...&gt;。</p> <p>Refresh 头不属于 HTTP 1.1 正式规范的一部分，而是一个扩展，但 Netscape 和 IE 都支持它。</p>
Server	服务器名字。Servlet 一般不设置这个值，而是由 Web 服务器自己设置。
Set-Cookie	设置和页面关联的 Cookie。Servlet 不应使用 response.setHeader("Set-Cookie", ...)，而是应使用 HttpServletResponse 提供的专用方法 addCookie。参见下文有关 Cookie 设置的讨论。
WWW-Authenticate	<p>客户应该在 Authorization 头中提供什么类型的授权信息？在包含 401（Unauthorized）状态行的应答中这个头是必需的。例如，response.setHeader("WWW-Authenticate", "BASIC realm= \"executives\"")。</p> <p>Servlet 一般不进行这方面的处理，而是让 Web 服务器的专门机制来控制受密码保护页面的访问（例如.htaccess）。</p>

### 4.6.3. 实例：定时刷新页面的例子

```

package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

```

```
public class AutoRefresh extends HttpServlet {  
    static int RefNum = 1;  
  
    /**  
     * The doGet method of the servlet. <br>  
     *  
     * This method is called when a form has its tag value method equals to get.  
     *  
     * @param request  
     *          the request send by the client to the server  
     * @param response  
     *          the response send by the server to the client  
     * @throws ServletException  
     *          if an error occurred  
     * @throws IOException  
     *          if an error occurred  
     */  
  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        String title = "定时刷新";  
        response.setHeader("Refresh", "5");  
        response.setContentType("text/html;charset=gb2312");  
        PrintWriter out = response.getWriter();  
        out.println(ServletUtilities.headWithTitle(title));  
        out.println("  <BODY>");  
        out.println("    <H1>" + title + "</H1>");  
        out.println("    <H1>第" + RefNum++ + "次: " + new java.util.Date()  
                  + "</H1>");  
        out.println("  </BODY>");  
        out.println("</HTML>");  
        out.flush();  
        out.close();  
    }  
  
    /**  
     * The doPost method of the servlet. <br>  
     *  
     * This method is called when a form has its tag value method equals to  
     * post.  
     *  
     * @param request  
     *          the request send by the client to the server  
     * @param response  
     *          the response send by the server to the client
```

```
* @throws ServletException
*           if an error occurred
* @throws IOException
*           if an error occurred
*/
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    this doGet(request, response);
}

}
```

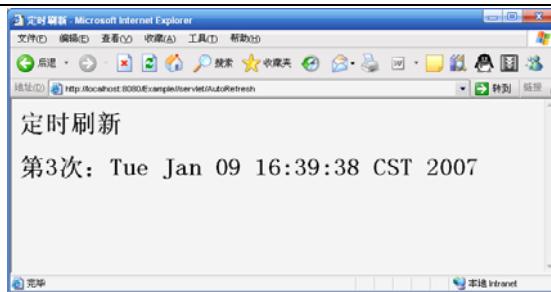


图 4-14

## 4.7. 处理 Cookie

### 4.7.1. Cookie 概述

Cookie 是服务器发送给浏览器的体积很小的纯文本信息，用户以后访问同一个 Web 服务器时浏览器会把它们原样发送给服务器。通过让服务器读取它原先保存到客户端的信息，网站能够为浏览器提供一系列的方便，例如在线交易过程中标识用户身份、安全要求不高的场合避免用户重复输入名字和密码、门户网站的主页定制、有针对性地投放广告，等等。

Cookie 的目的就是为用户带来方便，为网站带来增值。虽然有着许多误传，事实上 Cookie 并不会造成严重的安全威胁。Cookie 永远不会以任何方式执行，因此也不会带来病毒或攻击你的系统。另外，由于浏览器一般只允许存放 300 个 Cookie，每个站点最多存放 20 个 Cookie，每个 Cookie 的大小限制为 4 KB，因此 Cookie 不会塞满你的硬盘，更不会被用作“拒绝服务”攻击手段。

### 4.7.2. Servlet 的 Cookie API

要把 Cookie 发送到客户端，Servlet 先要调用 `new Cookie(name,value)` 用合适的名字和值创建一个或多个 Cookie，通过 `cookie.setXXX` 设置各种属性，通过 `response.addCookie(cookie)` 把 cookie 加入应答头。

要从客户端读入 Cookie，Servlet 应该调用 `request.getCookies()`，`getCookies()` 方法返回一个 Cookie 对象的数组。在大多数情况下，你只需要用循环访问该数组的各个元素寻找指定名字的 Cookie，然后对该 Cookie 调用 `getValue` 方法取得与指定名字关联的值。

#### 4.7.2.1. 创建 Cookie

调用 Cookie 对象的构造函数可以创建 Cookie。Cookie 对象的构造函数有两个字符串参数：Cookie 名字和 Cookie 值。名字和值都不能包含空白字符以及下列字符： [ ] ( ) = , " / ?

@ : ;

### 4.7.2.2. 读取和设置 Cookie 属性

把 Cookie 加入待发送的应答头之前，你可以查看或设置 Cookie 的各种属性。下面摘要介绍这些方法：

表 4-4 操作 Cookie 方法

方法	含义
getComment/setComment	获取/设置 Cookie 的注释。
getDomain/setDomain	获取/设置 Cookie 适用的域。一般地，Cookie 只返回给与发送它的服务器名字完全相同的服务器。使用这里的方法可以指示浏览器把 Cookie 返回给同一域内的其他服务器。注意域必须以点开始(例如.sitename.com)，非国家类的域(如.com, .edu, .gov)必须包含两个点，国家类的域(如.com.cn, .edu.uk)必须包含三个点。
getMaxAge/setMaxAge	获取/设置 Cookie 过期之前的时间，以秒计。如果不设置该值，则 Cookie 只在当前会话内有效，即在用户关闭浏览器之前有效，而且这些 Cookie 不会保存到磁盘上。参见下面有关 LongLivedCookie 的说明。
getName/setName	获取/设置 Cookie 的名字。本质上，名字和值是我们始终关心的两个部分。由于 HttpServletRequest 的 getCookies 方法返回的是一个 Cookie 对象的数组，因此通常要用循环来访问这个数组查找特定名字，然后用 getValue 检查它的值。
getPath/setPath	获取/设置 Cookie 适用的路径。如果不指定路径，Cookie 将返回给当前页面所在目录及其子目录下的所有页面。这里的方法可以用来设定一些更一般的条件。例如，someCookie.setPath("/"), 此时服务器上的所有页面都可以接收到该 Cookie。
getSecure/setSecure	获取/设置一个 boolean 值，该值表示是否 Cookie 只能通过加密的连接(即 SSL)发送。
getValue/setValue	获取/设置 Cookie 的值。如前所述，名字和值实际上是我们始终关心的两个方面。不过也有一些例外情况，比如把名字作为逻辑标记(也就是说，如果名字存在，则表示 true)。
getVersion/setVersion	获取/设置 Cookie 所遵从的协议版本。默认版本 0(遵从原先的 Netscape 规范)；版本 1 遵从 RFC 2109，但尚未得到广泛的支持。

### 4.7.2.3. 在应答头中设置 Cookie

Cookie 可以通过 HttpServletResponse 的 addCookie 方法加入到 Set-Cookie 应答头。下面是一个例子：

```
Cookie userCookie = new Cookie("user", "uid1234");
response.addCookie(userCookie);
```

### 4.7.2.4. 读取保存到客户端的 Cookie

要把 Cookie 发送到客户端，先要创建 Cookie，然后用 addCookie 发送一个 Set-Cookie

HTTP 应答头。这些内容已经在上面介绍。从客户端读取 Cookie 时调用的是 HttpServletRequest 的 getCookies 方法。该方法返回一个与 HTTP 请求头中的内容对应的 Cookie 对象数组。得到这个数组之后，一般是用循环访问其中的各个元素，调用 getName 检查各个 Cookie 的名字，直至找到目标 Cookie。然后对这个目标 Cookie 调用 getValue，根据获得的结果进行其他处理。

上述处理过程经常会遇到，为方便计下面我们提供一个 getCookieValue 方法。只要给出 Cookie 对象数组、Cookie 名字和默认值，getCookieValue 方法就会返回匹配指定名字的 Cookie 值，如果找不到指定 Cookie，则返回默认值。

### 4.7.3. 几个 Cookie 工具函数

下面是几个工具函数。这些函数虽然简单，但是，在和 Cookie 打交道的时候很有用。

#### 4.7.3.1. 获取指定名字的 Cookie 值

该函数是 ServletUtilities.java 的一部分。getCookieValue 通过循环依次访问 Cookie 对象数组的各个元素，寻找是否有指定名字的 Cookie，如找到，则返回该 Cookie 的值；否则，返回参数中给出的默认值。getCookieValue 能够在一定程度上简化 Cookie 值的提取。

```
public static String getCookieValue(Cookie[] cookies, String cookieName, String defaultValue) {
    for(int i=0; i<cookies.length; i++) {
        Cookie cookie = cookies[i];
        if (cookieName.equals(cookie.getName()))
            return(cookie.getValue());
    }
    return(defaultValue);
}
```

#### 4.7.3.2. 自动保存的 Cookie

下面是 LongLivedCookie 类的代码。如果你希望 Cookie 能够在浏览器退出的时候自动保存下来，则可以用这个 LongLivedCookie 类来取代标准的 Cookie 类。

```
package com.powerise.edu.servlet;

import javax.servlet.http.Cookie;

public class LongLivedCookie extends Cookie {
    public static final int SECONDS_PER_YEAR = 60 * 60 * 24 * 365;

    public LongLivedCookie(String name, String value) {
        super(name, value);
        setMaxAge(SECONDS_PER_YEAR);
    }
}
```

#### 4.7.3.3. 实例：定制的搜索引擎界面

下面也是一个搜索引擎界面的例子，通过修改前面 HTTP 状态代码的例子得到。在这个

Servlet 中，用户界面是动态生成而不是由静态 HTML 文件提供的。Servlet 除了负责读取表单数据并把它们发送给搜索引擎之外，还要把包含表单数据的 Cookie 发送给客户端。以后客户再次访问同一表单时，这些 Cookie 的值将用来预先填充表单，使表单自动显示最近使用过的数据。

### SearchEnginesFrontEnd.java

Servlet 构造一个主要由表单构成的用户界面。第一次显示的时候，它和前面用静态 HTML 页面提供的界面差不多。然而，用户选择的值将被保存到 Cookie（本页面将数据发送到 CustomizedSearchEngines Servlet，由后者设置 Cookie）。用户以后再访问同一页面时，即使浏览器是退出之后再启动，表单中也会自动填好上一次搜索所填写的内容。

注意该 Servlet 用到了 ServletUtilities.java，其中 getCookieValue 前面已经介绍过，headWithTitle 用于生成 HTML 页面的一部分。另外，这里也用到了前面已经说明的 LongLiveCookie 类，我们用它来创建作废期限很长的 Cookie。

---

```
package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Cookie;

@SuppressWarnings("serial")
public class SearchEnginesFrontEnd extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        Cookie[] cookies = request.getCookies();
        String searchString = ServletUtilities.getCookieValue(cookies,
            "searchString", "Java Programming");
        String numResults = ServletUtilities.getCookieValue(cookies,
            "numResults", "10");
        String searchEngine = ServletUtilities.getCookieValue(cookies,
            "searchEngine", "google");
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        String title = "Searching the Web";
        out.println(ServletUtilities.headWithTitle(title)
            + "<BODY BGCOLOR=\"#FDF5E6\">\n"
            + "<H1 ALIGN=\"CENTER\">Searching the Web</H1>\n" + "\n"
            + "<FORM ACTION=\"/servlet/CustomizedSearchEngines\">\n"
            + "<CENTER>\n" + "Search String:\n"
            + "<INPUT TYPE=\"TEXT\" NAME=\"searchString\"\n" + " VALUE=\"\""
            + searchString + "\"><BR>\n" + "Results to Show Per Page:\n"
```

```

+ "<INPUT TYPE=\"TEXT\" NAME=\"numResults\"\n" + " VALUE="
+ numResults + " SIZE=3><BR>\n"
+ "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n"
+ " VALUE=\"google\"" + checked("google", searchEngine) + ">\n"
+ "Google \n"
+ "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n"
+ " VALUE=\"baidu\"" + checked("baidu", searchEngine)
+ ">\n" + "百度 \n"
+ "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n"
+ " VALUE=\"sohu\"" + checked("sohu", searchEngine) + ">\n"
+ "搜狐 \n" + "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n"
+ " VALUE=\"sina\"" + checked("hotbot", searchEngine) + ">\n"
+ "新浪\n" + "<BR>\n"
+ "<INPUT TYPE=\"SUBMIT\" VALUE=\"Search\">\n" + "</CENTER>\n"
+ "</FORM>\n" + "\n" + "</BODY>\n" + "</HTML>\n");
}

private String checked(String name1, String name2) {
    if (name1.equals(name2))
        return (" CHECKED");
    else
        return ("");
}

```

---

#### CustomizedSearchEngines.java

前面的 SearchEnginesFrontEnd Servlet 把数据发送到 CustomizedSearchEngines Servlet。本例在许多方面与前面介绍 HTTP 状态代码时的例子相似，区别在于，本例除了要构造一个针对搜索引擎的 URL 并向用户发送一个重定向应答之外，还要发送保存用户数据的 Cookies。

```

package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Cookie;
import java.net.URLEncoder;

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

```

```
String searchString = request.getParameter("searchString");
Cookie searchStringCookie = new LongLivedCookie("searchString",
    searchString);
response.addCookie(searchStringCookie);
searchString = URLEncoder.encode(searchString);
String numResults = request.getParameter("numResults");
Cookie numResultsCookie = new LongLivedCookie("numResults", numResults);
response.addCookie(numResultsCookie);
String searchEngine = request.getParameter("searchEngine");
Cookie searchEngineCookie = new LongLivedCookie("searchEngine",
    searchEngine);
response.addCookie(searchEngineCookie);
SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
for (int i = 0; i < commonSpecs.length; i++) {
    SearchSpec searchSpec = commonSpecs[i];
    if (searchSpec.getName().equals(searchEngine)) {
        String url = searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
    }
}
response.sendError(response.SC_NOT_FOUND,
    "No recognized search engine specified.");
}

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

---

## 4.8. 会话状态

### 4.8.1. 会话状态概述

HTTP 协议的“无状态”(Stateless)特点带来了一系列的问题。特别是通过在线商店购物时，服务器不能顺利地记住以前的事务就成了严重的问题。它使得“购物篮”之类的应用很难实现：当我们把商品加入购物篮时，服务器如何才能知道篮子里原先有些什么？即使服务器保存了上下文信息，我们仍旧会在电子商务应用中遇到问题。例如，当用户从选择商品的页面（由普通的服务器提供）转到输入信用卡号和送达地址的页面（由支持 SSL 的安全服务器提供），服务器如何才能记住用户买了些什么？

这个问题一般有三种解决方法：

#### 1. Cookie:

利用 HTTP Cookie 来存储有关购物会话的信息，后继的各个连接可以查看当前会话，

然后从服务器的某些地方提取有关该会话的完整信息。这是一种优秀的，也是应用最广泛的方法。然而，即使 Servlet 提供了一个高级的、使用方便的 Cookie 接口，仍旧有一些繁琐的细节问题需要处理：

从其他 Cookie 中分别出保存会话标识的 Cookie。

为 Cookie 设置合适的作废时间（例如，中断时间超过 24 小时的会话一般应重置）。

把会话标识和服务器端相应的信息关联起来。（实际保存的信息可能要远远超过保存到 Cookie 的信息，而且象信用卡号等敏感信息永远不应该用 Cookie 来保存。）

### 2. 改写 URL:

你可以把一些标识会话的数据附加到每个 URL 的后面，服务器能够把该会话标识和它所保存的会话数据关联起来。这也是一个很好的方法，而且还有当浏览器不支持 Cookie 或用户已经禁用 Cookie 的情况下也有效这一优点。然而，大部分使用 Cookie 时所面临的问题同样存在，即服务器端的程序要进行许多简单但单调冗长的处理。另外，还必须十分小心地保证每个 URL 后面都附加了必要的信息（包括非直接的，如通过 Location 给出的重定向 URL）。如果用户结束会话之后又通过书签返回，则会话信息会丢失。

### 3. 隐藏表单域:

HTML 表单中可以包含下面这样的输入域：`<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">`。这意味着，当表单被提交时，隐藏域的名字和数据也被包含到 GET 或 POST 数据里，我们可以利用这一机制来维持会话信息。然而，这种方法有一个很大的缺点，它要求所有页面都是动态生成的，因为整个问题的核心就是每个会话都要有一个唯一标识符。

Servlet 为我们提供了一种与众不同的方案：HttpSession API。HttpSession API 是一个基于 Cookie 或者 URL 改写机制的高级会话状态跟踪接口：如果浏览器支持 Cookie，则使用 Cookie；如果浏览器不支持 Cookie 或者 Cookie 功能被关闭，则自动使用 URL 改写方法。Servlet 开发者无需关心细节问题，也无需直接处理 Cookie 或附加到 URL 后面的信息，API 自动为 Servlet 开发者提供一个可以方便地存储会话信息的地方。

## 4.8.2. 会话状态跟踪 API

在 Servlet 中使用会话信息是相当简单的，主要的操作包括：查看和当前请求关联的会话对象，必要的时候创建新的会话对象，查看与某个会话相关的信息，在会话对象中保存信息，以及会话完成或中止时释放会话对象。

## 4.8.3. 查看当前请求的会话对象

查看当前请求的会话对象通过调用 HttpServletRequest 的 `getSession` 方法实现。如果 `getSession` 方法返回 `null`，你可以创建一个新的会话对象。但更经常地，我们通过指定参数使得不存在现成的会话时自动创建一个会话对象，即指定 `getSession` 的参数为 `true`。因此，访问当前请求会话对象的第一个步骤通常如下所示：

```
HttpSession session = request.getSession(true);
```

## 4.8.4. 查看和会话有关的信息

HttpSession 对象生存在服务器上，通过 Cookie 或者 URL 这类后台机制自动关联到请求的发送者。会话对象提供一个内建的数据结构，在这个结构中可以保存任意数量的键-值对。在 2.1 或者更早版本的 Servlet API 中，查看以前保存的数据使用的是 `getValue("key")` 方法。`getValue` 返回 `Object`，因此你必须把它转换成更加具体的数据类型。如果参数中指定的键不存在，`getValue` 返回 `null`。

API 2.2 版推荐用 `getAttribute` 来代替 `getValue`, 这不仅是因为 `getAttribute` 和 `setAttribute` 的名字更加匹配 (和 `getValue` 匹配的是 `putValue`, 而不是 `setValue`), 同时也因为 `setAttribute` 允许使用一个附属的 `HttpSessionBindingListener` 来监视数值, 而 `putValue` 则不能。

但是, 由于目前还只有很少的商业 Servlet 引擎支持 2.2, 下面的例子中我们仍旧使用 `getValue`。这是一个很典型的例子, 假定 `ShoppingCart` 是一个保存已购买商品信息的类:

```
 HttpSession session = request.getSession(true);
 ShoppingCart previousItems =
 (ShoppingCart)session.getValue("previousItems");
 if (previousItems != null) {
 doSomethingWith(previousItems);
 } else {
 previousItems = new ShoppingCart(...);
 doSomethingElseWith(previousItems);
 }
```

大多数时候我们都是根据特定的名字寻找与它关联的值, 但也可以调用 `getValuesNames` 得到所有属性的名字。`getValuesNames` 返回的是一个 `String` 数组。API 2.2 版推荐使用 `getAttributeNames`, 这不仅是因为其名字更好, 而且因为它返回的是一个 `Enumeration`, 和其他方法 (比如 `HttpServletRequest` 的 `getHeaders` 和 `getParameterNames`) 更加一致。

虽然开发者最为关心的往往是保存到会话对象的数据, 但还有其他一些信息有时也很有用。

`getID`: 该方法返回会话的唯一标识。有时该标识被作为键-值对中的键使用, 比如会话中只保存一个值时, 或保存上一次会话信息时。

`isNew`: 如果客户 (浏览器) 还没有绑定到会话则返回 `true`, 通常意味着该会话刚刚创建, 而不是引用自客户端的请求。对于早就存在的会话, 返回值为 `false`。

`getCreationTime`: 该方法返回建立会话的以毫秒计的时间, 从 1970.01.01 (GMT) 算起。要得到用于打印输出的时间值, 可以把该值传递给 `Date` 构造函数, 或者 `GregorianCalendar` 的 `setTimeInMillis` 方法。

`getLastAccessedTime`: 该方法返回客户最后一次发送请求的以毫秒计的时间, 从 1970.01.01 (GMT) 算起。

`getMaxInactiveInterval`: 返回以秒计的最大时间间隔, 如果客户请求之间的间隔不超过该值, Servlet 引擎将保持会话有效。负数表示会话永远不会超时。

## 4.8.5. 在会话对象中保存数据

如上节所述, 读取保存在会话中的信息使用的是 `getValue` 方法(或, 对于 2.2 版的 Servlet 规范, 使用 `getAttribute`)。保存数据使用 `putValue` (或 `setAttribute`) 方法, 并指定键和相应的值。注意 `putValue` 将替换任何已有的值。有时候这正是我们所需要的 (如下例中的 `referringPage`), 但有时我们却需要提取原来的值并扩充它 (如下例 `previousItems`)。示例代码如下:

```
 HttpSession session = request.getSession(true);
 session.putValue("referringPage", request.getHeader("Referer"));
 ShoppingCart previousItems = (ShoppingCart)session.getValue("previousItems");
 if (previousItems == null) {
 previousItems = new ShoppingCart(...);
 }
```

```
String itemID = request.getParameter("itemID");
previousItems.addEntry(Catalog.getEntry(itemID));

session.putValue("previousItems", previousItems);
```

## 4.8.6. 实例：显示会话信息

下面这个例子生成一个 Web 页面，并在该页面中显示有关当前会话的信息。

```
package com.powerise.edu.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

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

@SuppressWarnings("serial")
public class ShowSession extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request
     *          the request send by the client to the server
     * @param response
     *          the response send by the server to the client
     * @throws ServletException
     *          if an error occurred
     * @throws IOException
     *          if an error occurred
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Searching the Web";
        String heading;
        Integer accessCount = new Integer(0);
```

```
;

if (session.isNew()) {
    heading = "Welcome, Newcomer";
} else {
    heading = "Welcome Back";
    Integer oldAccessCount =
        // 在 Servlet API 2.2 中使用 getAttribute 而不是 getValue
        (Integer) session.getValue("accessCount");
    if (oldAccessCount != null) {
        accessCount = new Integer(oldAccessCount.intValue() + 1);
    }
}

// 在 Servlet API 2.2 中使用 putAttribute
session.putValue("accessCount", accessCount);

out.println(ServletUtilities.headWithTitle(title)
    + "<BODY BGCOLOR=\"#FDF5E6\">\n" + "<H1 ALIGN=\"CENTER\">"
    + heading + "</H1>\n"
    + "<H2>Information on Your Session:</H2>\n"
    + "<TABLE BORDER=1 ALIGN= CENTER>\n"
    + "<TR BGCOLOR=\"#FFAD00\">\n" + " <TH>Info Type<TH>Value\n"
    + "<TR>\n" + " <TD>ID\n" + " <TD>" + session.getId() + "\n"
    + "<TR>\n" + " <TD>Creation Time\n" + " <TD>"
    + new Date(session.getCreationTime()) + "\n" + "<TR>\n"
    + " <TD>Time of Last Access\n" + " <TD>"
    + new Date(session.getLastAccessedTime()) + "\n" + "<TR>\n"
    + " <TD>Number of Previous Accesses\n" + " <TD>" + accessCount
    + "\n" + "</TABLE>\n" + "</BODY></HTML>");

}

/***
 * The doPost method of the servlet. <br>
 *
 * This method is called when a form has its tag value method equals to
 * post.
 *
 * @param request
 *          the request send by the client to the server
 * @param response
 *          the response send by the server to the client
 * @throws ServletException
 *          if an error occurred
 * @throws IOException
 */
```

```
*           if an error occurred  
*/  
public void doPost(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    this doGet(request, response);  
}  
}
```

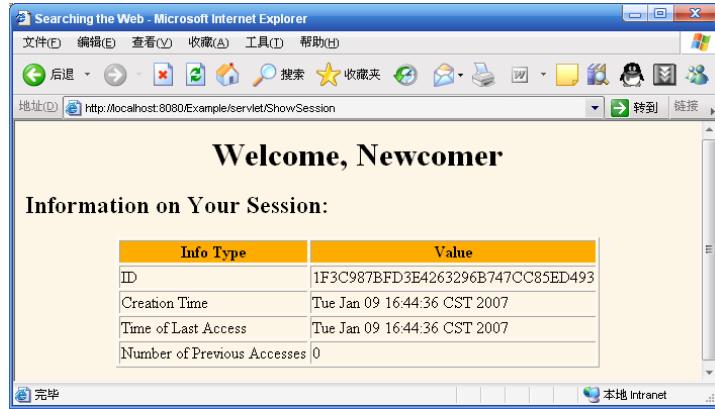


图 4-15

## 4.9. 小结

Servlet 是 J2EE 的核心技术之一，同时它还是 J2EE WEB 应用开发中最重要的技术，本章一开始就通过一个实例介绍了 Servlet 基本概念，代表 Servlet 生命周期的 3 个方法：init()，service() 和 destory()。然后通过一系列的实例介绍了 Servlet API 中常用到的一些接口和类，我们可以学会利用 Servlet：

- 处理表单数据；
- 读取 HTTP 请求；
- 访问 CGI 变量；
- 处理 HTTP 应答状态；
- 设置 HTTP 应答头；
- 处理 COOKIE；
- 处理用户会话；

# 第5章 Java Server Page

## 5.1. 概述

Java 是一种简单易用、完全面向对象、具有平台无关性且安全可靠的主要面向 Internet 的开发工具。自从 1995 年正式问世以来，Java 的快速发展已经让整个 Web 世界发生了翻天覆地的变化。随着 Java Servlet 的推出，Java 在电子商务方面开始崭露头角，Java Server Page 技术的推出，更是让 Java 成为基于 Web 的应用程序的首选开发工具。

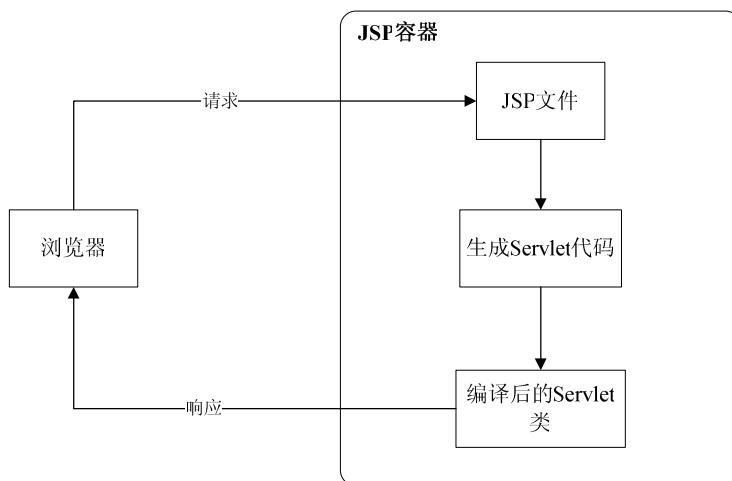


图 5-1: JSP 容器初次执行 JSP 的过程

JavaServer Pages (JSP) 是一种实现普通静态 HTML 和动态 HTML 混合编码的技术，许多由 CGI 程序生成的页面大部分仍旧是静态 HTML，动态内容只在页面中有限的几个部分出现。但是包括 Servlet 在内的大多数 CGI 技术及其变种，总是通过程序生成整个页面。JSP 使得我们可以分别创建这两个部分。例如，下面就是一个简单的 JSP 页面：

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>欢迎访问网上商店</TITLE></HEAD>
<BODY>
<H1>欢迎</H1>
<SMALL>欢迎,
<!-- 首次访问的用户名为"New User" -->
<% out.println(Utils.getUserNameFromCookie(request)); %>
要设置帐号信息, 请点击
<A HREF="Account-Settings.html">这里</A></SMALL>
<P>
页面的其余内容。
</BODY></HTML>
  
```

JavaServer Pages (JSP) 使得我们能够分离页面的静态 HTML 和动态部分。HTML 可以用任何通常使用的 Web 制作工具编写，编写方式也和原来的一样；动态部分的代码放入特殊标记之内，大部分以“`<%`”开始，以“`%>`”结束。例如，下面是一个 JSP 页面的片断，如果我们用 `http://host/OrderConfirmation.jsp?title=Core+Web+Programming` 这个 URL 打开该页面，则结果显示“Thanks for ordering Core Web Programming”。

Thanks for ordering

```
<I><%= request.getParameter("title") %></I>
```

JSP 页面文件通常以.jsp 为扩展名,而且可以安装到任何能够存放普通 Web 页面的地方。虽然从代码编写来看, JSP 页面更象普通 Web 页面而不象 Servlet, 但实际上, JSP 最终会被转换成正规的 Servlet, 静态 HTML 直接输出到和 Servlet service 方法关联的输出流。

JSP 到 Servlet 的转换过程一般在出现第一次页面请求时进行。因此, 如果你希望第一个用户不会由于 JSP 页面转换成 Servlet 而等待太长的时间, 希望确保 Servlet 已经正确地编译并装载, 你可以在安装 JSP 页面之后自己请求这个页面。

另外也请注意, 许多 Web 服务器允许定义别名, 所以一个看起来指向 HTML 文件的 URL 实际上可能指向 Servlet 或 JSP 页面。

除了普通 HTML 代码之外, 嵌入 JSP 页面的其他成分主要有如下三种: 脚本元素 (Scripting Element), 指令 (Directive), 动作 (Action)。脚本元素用来嵌入 Java 代码, 这些 Java 代码将成为转换得到的 Servlet 的一部分; JSP 指令用来从整体上控制 Servlet 的结构; 动作用来引入现有的组件或者控制 JSP 引擎的行为。为了简化脚本元素, JSP 定义了一组可以直接使用的变量 (预定义变量), 比如前面代码片断中的 request 就是其中一例。

## 5.2. JSP 和其他类似技术比较

### JSP 和 Active Server Pages (ASP) 相比

Microsoft 的 ASP 是一种和 JSP 类似的技术。JSP 和 ASP 相比具有两方面的优点。首先, 动态部分用 Java 编写, 而不是 VB Script 或其他 Microsoft 语言, 不仅功能更强大而且更容易使用。第二, JSP 应用可以移植到其他操作系统和非 Microsoft 的 Web 服务器上。

### JSP 和纯 Servlet 相比

JSP 并没有增加任何本质上不能用 Servlet 实现的功能。但是, 在 JSP 中编写静态 HTML 更加方便, 不必再用 println 语句来输出每一行 HTML 代码。更重要的是, 借助内容和外观的分离, 页面制作中不同性质的任务可以方便地分开: 比如, 由页面设计专家进行 HTML 设计, 同时留出供 Servlet 程序员插入动态内容的空间。

### JSP 和服务器端包含 (Server-Side Include, SSI) 相比

SSI 是一种受到广泛支持的在静态 HTML 中引入外部代码的技术。JSP 在这方面的支持更为完善, 因为它可以用 Servlet 而不是独立的程序来生成动态内容。另外, SSI 实际上只用于简单的包含, 而不是面向那些能够处理表单数据、访问数据库的“真正的”程序。

### JSP 和 JavaScript 相比

JavaScript 能够在客户端动态地生成 HTML。虽然 JavaScript 很有用, 但它只能处理以客户端环境为基础的动态信息。除了 Cookie 之外, HTTP 状态和表单提交数据对 JavaScript 来说都是不可用的。另外, 由于是在客户端运行, JavaScript 不能访问服务器端资源, 比如数据库、目录信息等等。

## 5.3. JSP 语法概要表

表 5-1 JSP 语法概要

JSP 元素	语法	说明	备注
JSP 表达式	<%= expression %>	计算表达式并输出结果。等价的 XML 表达是: <jsp:expression> expression </jsp:expression>	可以使用的预定义变量包括: request, response, out, session, application, config, pageContext。这些预定义变量也可以在 JSP Scriptlet 中使用。
JSP Scriptlet	<% code %>	插入到 service 方法的代码。等	

		价的 XML 表达是: <jsp:scriptlet> code </jsp:scriptlet>	
JSP 声明	<%! code %>	代码被插入到 Servlet 类（在 service 方法之外）。 等价的 XML 表达是: <jsp:declaration> code </jsp:declaration>	
page 指令	<%@ page att="val" %>	作用于 Servlet 引擎的全局性指令。 等价的 XML 表达是 <jsp:directive.page att="val" %>。	合法的属性如下表，其中粗体表示默认值： <b>import="package.class"</b> <b>contentType="MIME-Type"</b> <b>isThreadSafe="true false"</b> <b>session="true false"</b> <b>buffer="size kb none"</b> <b>autoFlush="true false"</b> <b>extends="package.class"</b> <b>info="message"</b> <b>errorPage="url"</b> <b>isErrorPage="true false"</b> <b>language="java"</b>
include 指令	<%@ include file="url" %>	当 JSP 转换成 Servlet 时，应当包含本地系统上的指定文件。 等价的 XML 表达是： <jsp:directive.include file="url" %> 其中 URL 必须是相对 URL。	利用 jsp:include 动作可以在请求的时候（而不是 JSP 转换成 Servlet 时）引入文件。
JSP 注释	<%-- comment --%>	注释： JSP 转换成 Servlet 时被忽略。如果要把注释嵌入结果 HTML 文档，使用普通的 HTML 注释标记<-- comment -->。	
jsp:include 动作	<jsp:include page="relative URL" flush="true"/>	当 Servlet 被请求时，引入指定的文件。如果你希望在页面转换的时候包含某个文件，使用 JSP include 指令。 注意：在某些服务器上，被包含文件必须是 HTML 文件或 JSP 文件，具体由服务器决定（通常根据文件扩展名判断）。	
jsp:useBean 动作	<jsp:useBean att="val" %> 或者	寻找或实例化一个 Java Bean。 可能的属性包括：	

	<jsp:useBean att=val*> ... </jsp:useBean>	id="name"	
--	---	-----------	--

## 5.4. 脚本元素、指令和预定义变量

### 5.4.1. JSP 脚本元素

JSP 脚本元素用来插入 Java 代码, 这些 Java 代码将出现在由当前 JSP 页面生成的 Servlet 中。脚本元素有三种格式:

表达式格式<%= expression %>: 计算表达式并输出其结果。

Scriptlet 格式<% code %>: 把代码插入到 Servlet 的 service 方法。

声明格式<%! code %>: 把声明加入到 Servlet 类 (在任何方法之外)。下面我们详细说明它们的用法。

#### 5.4.1.1. JSP 表达式

JSP 表达式用来把 Java 数据直接插入到输出。其语法如下:

<%= Java Expression %>

计算 Java 表达式得到的结果被转换成字符串, 然后插入到页面。计算在运行时进行(页面被请求时), 因此可以访问和请求有关的全部信息。例如, 下面的代码显示页面被请求的日期/时间:

Current time: <%= new java.util.Date() %>

为简化这些表达式, JSP 预定义了一组可以直接使用的对象变量。后面我们将详细介绍这些隐含声明的对象, 但对于 JSP 表达式来说, 最重要的几个对象及其类型如下:

request: HttpServletRequest;

response: HttpServletResponse;

session: 和 request 关联的 HttpSession

out: PrintWriter (带缓冲的版本, JspWriter), 用来把输出发送到客户端, 下面是一个例子:

Your hostname: <%= request.getRemoteHost() %>

最后, 如果使用 XML 的话, JSP 表达式也可以写成下面这种形式:

<jsp:expression>

Java Expression

</jsp:expression>

请记住 XML 元素和 HTML 不一样。XML 是大小写敏感的, 因此务必使用小写。

#### 5.4.1.2. JSP Scriptlet

如果你要完成的任务比插入简单的表达式更加复杂, 可以使用 JSP Scriptlet。JSP Scriptlet 允许你把任意的 Java 代码插入 Servlet。JSP Scriptlet 语法如下:

<% Java Code %>

和 JSP 表达式一样, Scriptlet 也可以访问所有预定义的变量。例如, 如果你要向结果页面输出内容, 可以使用 out 变量:

<%

String queryData = request.getQueryString();

out.println("Attached GET data: " + queryData);

```
%>
```

注意 Scriptlet 中的代码将被照搬到 Servlet 内, 而 Scriptlet 前面和后面的静态 HTML(模板文本) 将被转换成 `println` 语句。这就意味着, Scriptlet 内的 Java 语句并非一定要是完整的, 没有关闭的块将影响 Scriptlet 外的静态 HTML。例如, 下面的 JSP 片断混合了模板文本和 Scriptlet:

```
<% if (Math.random() < 0.5) { %>
    Have a <B>nice</B> day!
<% } else { %>
    Have a <B>lousy</B> day!
<% } %>
```

上述 JSP 代码将被转换成如下 Servlet 代码:

```
if (Math.random() < 0.5) {
    out.println("Have a <B>nice</B> day!");
} else {
    out.println("Have a <B>lousy</B> day!");
}
```

如果要在 Scriptlet 内部使用字符 “%>”, 必须写成 “%>”。另外, 请注意`<% code %>`的 XML 等价表达是:

```
<jsp:scriptlet>
Code
</jsp:scriptlet>
```

### 5.4.1.3. JSP 声明

JSP 声明用来定义插入 Servlet 类的方法和成员变量, 其语法如下:

```
<%! Java Code %>
```

由于声明不会有任何输出, 因此它们往往和 JSP 表达式或 Scriptlet 结合在一起使用。例如, 下面的 JSP 代码片断输出自从服务器启动 (或 Servlet 类被改动并重新装载以来) 当前页面被请求的次数:

```
<%! private int accessCount = 0; %>
自从服务器启动以来页面访问次数为:
<%= ++accessCount %>
```

和 Scriptlet 一样, 如果要使用字符串 “%>”, 必须使用 “%>” 代替。最后, `<%! code %>` 的 XML 等价表达方式为:

```
<jsp:declaration>
Code
</jsp:declaration>
```

### 5.4.2. JSP 指令

JSP 指令影响 Servlet 类的整体结构, 它的语法一般如下:

```
<%@ directive attribute="value" %>
```

另外, 也可以把同一指令的多个属性结合起来, 例如:

```
<%@ directive attribute1="value1"
attribute2="value2"
...>
```

```
attributeN="valueN" %>
```

JSP 指令分为两种类型：第一是 page 指令，用来完成下面这类任务：导入指定的类，自定义 Servlet 的超类，等等；第二是 include 指令，用来在 JSP 文件转换成 Servlet 时引入其他文件。JSP 规范也提到了 taglib 指令，其目的是让 JSP 开发者能够自己定义标记。

### 5.4.2.1. page 指令

page 指令的作用是定义下面一个或多个属性，这些属性大小写敏感。

import="package.class"，或者 import="package.class1,...,package.classN":

用于指定导入哪些包，例如：<%@ page import="java.util.\*" %>。import 是唯一允许出现一次以上的属性。

contentType="MIME-Type" 或 contentType="MIME-Type; charset=Character-Set":

该属性指定输出的 MIME 类型。默认是 text/html。例如，下面这个指令：

```
<%@ page contentType="text/plain" %>
```

和下面的 Scriptlet 效果相同：

```
<% response.setContentType("text/plain"); %>
```

isThreadSafe="true|false"

默认值 true 表明 Servlet 按照标准的方式处理，即假定开发者已经同步对实例变量的访问，由单个 Servlet 实例同时地处理多个请求。如果取值 false，表明 Servlet 应该实现 SingleThreadModel，请求或者是逐个进入，或者多个并行的请求分别由不同的 Servlet 实例处理。

session="true|false"

默认值 true 表明预定义变量 session（类型为 HttpSession）应该绑定到已有的会话，如果不存在已有的会话，则新建一个并绑定 session 变量。如果取值 false，表明不会用到会话，试图访问变量 session 将导致 JSP 转换成 Servlet 时出错。

buffer="size kb|none"

该属性指定 JspWrite out 的缓存大小。默认值和服务器有关，但至少应该是 8 KB。

autoflush="true|false"

默认值 true 表明如果缓存已满则刷新它。autoflush 很少取 false 值，false 值表示如果缓存已满则抛出异常。如果 buffer="none"，autoflush 不能取 false 值。

extends="package.class"

该属性指出将要生成的 Servlet 使用哪个超类。使用该属性应当十分小心，因为服务器可能已经在用自定义的超类。

info="message"

该属性定义一个可以通过 getServletInfo 方法提取的字符串。

errorPage="url"

该属性指定一个 JSP 页面，所有未被当前页面捕获的异？筛靡趁娲？怒？

isErrorPage="true|false"

该属性指示当前页面是否可以作为另一 JSP 页面的错误处理页面。默认值 false。

language="java"

该属性用来指示所使用的语言。目前没有必要关注这个属性，因为默认的 Java 是当前唯一可用的语言。

定义指令的 XML 语法为：

```
<jsp:directive.directiveType attribute=value />
```

例如，下面这个指令：

```
<%@ page import="java.util.*" %>
它的 XML 等价表达是:
<jsp:directive.page import="java.util.*" />
```

### 5.4.2.2. include 指令

include 指令用于 JSP 页面转换成 Servlet 时引入其他文件。该指令语法如下：

```
<%@ include file="relative url" %>
```

这里所指定的 URL 是和发出引用指令的 JSP 页面相对的 URL，然而，与通常意义上的相对 URL 一样，你可以利用以“/”开始的 URL 告诉系统把 URL 视为从 Web 服务器根目录开始。包含文件的内容也是 JSP 代码，即包含文件可以包含静态 HTML、脚本元素、JSP 指令和动作。

例如，许多网站的每个页面都有一个小小的导航条。由于 HTML 框架存在不少问题，导航条往往用页面顶端或左边的一个表格制作，同一份 HTML 代码重复出现在整个网站的每个页面上。include 指令是实现该功能的非常理想的方法。使用 include 指令，开发者不必再把导航 HTML 代码拷贝到每个文件中，从而可以更轻松地完成维护工作。

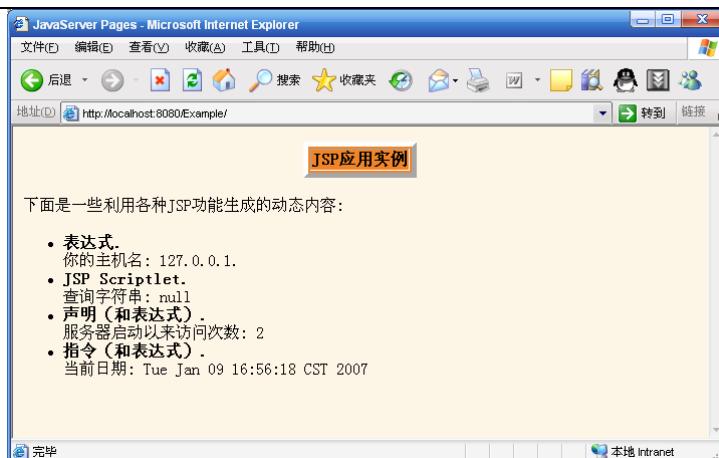
由于 include 指令是在 JSP 转换成 Servlet 的时候引入文件，因此如果导航条改变了，所有使用该导航条的 JSP 页面都必须重新转换成 Servlet。如果导航条改动不频繁，而且你希望包含操作具有尽可能好的效率，使用 include 指令是最好的选择。然而，如果导航条改动非常频繁，你可以使用 jsp:include 动作。jsp:include 动作在出现对 JSP 页面请求的时候才会引用指定的文件，请参见本文后面的具体说明。

### 5.4.3. 实例：脚本元素和指令的应用

下面是一个使用 JSP 表达式、Scriptlet、声明、指令的简单例子。

```
<%@ page contentType="text/html; charset=gb2312" language="java" errorPage="" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>JavaServer Pages</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
VLINK="#551A8B" ALINK="#FF0000">
<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
<TR>
<TH CLASS="TITLE"> JSP 应用实例
</TABLE>
</CENTER>
<P> 下面是一些利用各种 JSP 功能生成的动态内容:
<UL>
<LI><B>表达式.</B><BR>
你的主机名: <%= request.getRemoteHost() %>.
<LI><B>JSP Scriptlet.</B><BR>
<% out.println("查询字符串: " +
```

```
request.getQueryString()); %>
<LI><B>声明（和表达式）.</B><BR>
<%! private int accessCount = 0; %>
    服务器启动以来访问次数:<%= ++accessCount %>
<LI><B>指令（和表达式）.</B><BR>
<%@ page import = "java.util.*" %>
    当前日期:<%= new Date() %>
</UL>
</BODY>
</HTML>
```



## 5.5. JSP 预定义变量

为了简化 JSP 表达式和 Scriptlet 的代码，JSP 提供了 9 个预先定义的变量（或称为隐含对象）。这些变量是 request、response、out、session、application、config、pageContext、page 和 exception。

### 5.5.1. HttpServletRequest 类的 Request 对象

这是和请求关联的 HttpServletRequest，通过它可以查看请求参数（调用 getParameter），请求类型（GET, POST, HEAD, 等），以及请求的 HTTP 头（Cookie, Referer, 等）。严格说来，如果请求所用的是 HTTP 之外的其他协议，request 可以是 ServletRequest 的子类（而不是 HttpServletRequest），但在实践中几乎不会用到。

### 5.5.2. HttpServletResponse 类的 Response 对象

这是和应答关联的 HttpServletResponse。注意，由于输出流（参见下面的 out）是带缓冲的，因此，如果已经向客户端发送了输出内容，普通 Servlet 不允许再设置 HTTP 状态代码，但在 JSP 中却是合法的。

### 5.5.3. JspWriter 类的 out 对象

这是用来向客户端发送内容的 PrintWriter。然而，为了让 response 对象更为实用，out 是带缓存功能的 PrintWriter，即 JspWriter。JSP 允许通过 page 指令的 buffer 属性调整缓存的大小，甚至允许关闭缓存。

out 一般只在 Scriptlet 内使用，这是因为 JSP 表达式是自动发送到输出流的，很少需要显式地引用 out。

## 5.5.4. HttpSession 类的 session 对象

这是和请求关联的 HttpSession 对象。前面我们已经介绍过会话的自动创建，我们知道，即使不存在 session 引用，这个对象也是自动绑定的。但有一个例外，这就是如果你用 page 指令的 session 属性关闭了会话，此时对 session 变量的引用将导致 JSP 页面转换成 Servlet 时出错。

## 5.5.5. ServletContext 类的 application 对象

这是一个 ServletContext，也可以通过 getServletConfig().getServletContext()获得。

## 5.5.6. ServletConfig 类的 Config 对象

代码片段配置对象，表示 Servlet 的配置。

## 5.5.7. PageContext 类的 PageContext 对象

管理网页属性,为 JSP 页面包装页面的上下文，管理对属于 JSP 中特殊可见部分中已命名对象的访问，它的创建和初始化都是由容器来完成的。

## 5.5.8. Object 类的 Page (相当于 this) 对象

它是 this 的同义词，当前用处不大。它是为了 Java 不再是唯一的 JSP 编程语言而准备的占位符。

JSP 动作利用 XML 语法格式的标记来控制 Servlet 引擎的行为。利用 JSP 动作可以动态地插入文件、重用 JavaBean 组件、把用户重定向到另外的页面、为 Java 插件生成 HTML 代码。

## 5.5.9. exception

处理 JSP 文件执行时发生的错误和异常，只有在错误页面里才可以使用，前提是：在页面指令里要有 isErrorPage=true 的页面中。

如： a.jsp:

```
<%@ page contentType="text/html;charset=gb2312" errorPage="error.jsp"%>
error.jsp:
<%@ page contentType="text/html;charset=gb2312" isErrorPage="true"%>
```

则：在 error.jsp 页面中便可使用 exception 内置对象，用法如下：

```
<%=exception.getMessage()%>
```

下面创建一个可能会抛出异常的 JSP 页面 jspSum.jsp，在这个页面中读取客户请求中的两个参数 num1 和 num2，把他们转化为整数类型，再对其求和，最后把结果输出到网页上。

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

jspSum.jsp 的代码如下：

---

```
<%@ page contentType="text/html; charset=GB2312"%>
<%@ page errorPage="errorpage.jsp"%>

<html>
<head>
<title>jspSum.jsp</title>
```

```
</head>
<body>
<%!
    private intToInt(String num){
        return Integer.valueOf(num).intValue();
    }
%>
<%
    int num1 =ToInt(request.getParameter("num1"));
    int num2 =ToInt(request.getParameter("num2"));
%>

<p>
    运算结果为：
    <%=num1%>
    +
    <%=num2%>
    =
    <%=(num1 + num2)%>
</p>
</body>
</html>
```

---

errorpage.jsp 代码如下：

---

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

<html>
    <head>
        <title>Error Page</title>
    </head>
    <body>

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

```
%>
</p>
</body>
</html>
```

通过如下 URL 访问 jspSum.jsp:

http://localhost:8080/Example/jspSum.jsp?num1=22&num2=33

这时页面正常执行，生成网页如下所示：



图 5-2

如果将 URL 修改成如下样子：

http://localhost:8080/Example/jspSum.jsp?num1=22&num2=xx

这时 num2 无法转化为整数，就会抛出 NumberFormatException，这时客户请求就会转到 errorpage.jsp，生成网页如下所示：

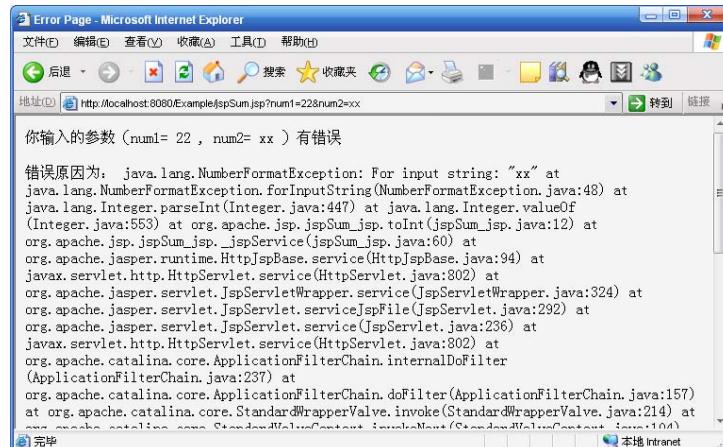


图 5-3

**注意：**pageContext 中的属性默认在当前页面是共享； session 中的属性是在当前 session 中是共享； ServletContext 对象中的属性则是对所有页面都是共享；

## 5.6. JSP 动作

- **jsp:include:** 在页面被请求的时候引入一个文件。
- **jsp:useBean:** 寻找或者实例化一个 JavaBean。
- **jsp:setProperty:** 设置 JavaBean 的属性。
- **jsp:getProperty:** 输出某个 JavaBean 的属性。
- **jsp:forward:** 把请求转到一个新的页面。
- **jsp:plugin:** 根据浏览器类型为 Java 插件生成 OBJECT 或 EMBED 标记。

### 5.6.1. jsp:include 动作

该动作把指定文件插入正在生成的页面。其语法如下：

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

前面已经介绍过 include 指令，它是在 JSP 文件被转换成 Servlet 的时候引入文件，而这里的 jsp:include 动作不同，插入文件的时间是在页面被请求的时候。jsp:include 动作的文件引入时间决定了它的效率要稍微差一点，而且被引用文件不能包含某些 JSP 代码（例如不能设置 HTTP 头），但它的灵活性却要好得多。

例如，下面的 JSP 页面把 4 则新闻摘要插入一个“What's New？”页面。改变新闻摘要时只需改变这四个文件，而主 JSP 页面却可以不作修改：

---

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage="" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>What's New</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
VLINK="#551A8B" ALINK="#FF0000">
<CENTER>
  <TABLE BORDER=5 BGCOLOR="#EF8429">
    <TR>
      <TH CLASS="TITLE"> What's New at JspNews.com
    </TABLE>
  </CENTER>
<P> Here is a summary of our four most recent news stories:
<OL>
  <LI>
    <jsp:include page="news/Item1.html" flush="true"/>
  <LI>
    <jsp:include page="news/Item2.html" flush="true"/>
  <LI>
    <jsp:include page="news/Item3.html" flush="true"/>
  <LI>
    <jsp:include page="news/Item4.html" flush="true"/>
</OL>
</BODY>
</HTML>
```

---

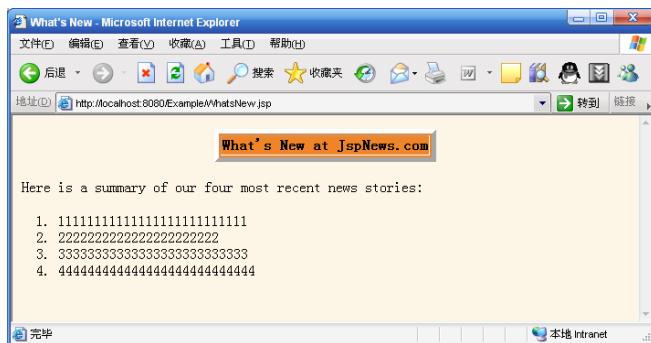


图 5-4

## 5.6.2. jsp:useBean 动作

`jsp:useBean` 动作用来装载一个将在 JSP 页面中使用的 JavaBean。这个功能非常有用，因为它使得我们既可以发挥 Java 组件重用的优势，同时也避免了损失 JSP 区别于 Servlet 的方便性。`jsp:useBean` 动作最简单的语法为：

```
<jsp:useBean id="name" class="package.class" />
```

这行代码的含义是：“创建一个由 `class` 属性指定的类的实例，然后把它绑定到其名字由 `id` 属性给出的变量上”。不过，就象我们接下来会看到的，定义一个 `scope` 属性可以让 Bean 关联到更多的页面。此时，`jsp:useBean` 动作只有在不存在同样 `id` 和 `scope` 的 Bean 时才创建新的对象实例，同时，获得现有 Bean 的引用就变得很有必要。

获得 Bean 实例之后，要修改 Bean 的属性既可以通过 `jsp:setProperty` 动作进行，也可以在 Scriptlet 中利用 `id` 属性所命名的对象变量，通过调用该对象的方法显式地修改其属性。这使我们想起，当我们说“某个 Bean 有一个类型为 X 的属性 `foo`”时，就意味着“这个类有一个返回值类型为 X 的 `getFoo` 方法，还有一个 `setFoo` 方法以 X 类型的值为参数”。

有关 `jsp:setProperty` 动作的详细情况在后面讨论。但现在必须了解的是，我们既可以通过 `jsp:setProperty` 动作的 `value` 属性直接提供一个值，也可以通过 `param` 属性声明 Bean 的属性值来自指定的请求参数，还可以列出 Bean 属性表明它的值应该来自请求参数中的同名变量。

在 JSP 表达式或 Scriptlet 中读取 Bean 属性通过调用相应的 `getXXX` 方法实现，或者更一般地，使用 `jsp:getProperty` 动作。

注意包含 Bean 的类文件应该放到服务器正式存放 Java 类的目录下，而不是保留给修改后能够自动装载的类的目录。例如，对于 Java Web Server 来说，Bean 和所有 Bean 用到的类都应该放入 `classes` 目录，或者封装进 jar 文件后放入 `lib` 目录，但不应该放到 `servlets` 下。

下面是一个很简单的例子，它的功能是装载一个 Bean，然后设置/读取它的 `message` 属性。

BeanTest.jsp:

---

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage="" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
</HEAD>
```

```
<BODY>
<CENTER>
  <TABLE BORDER=5>
    <TR>
      <TH CLASS="TITLE"> Reusing JavaBeans in JSP
    </TABLE>
  </CENTER>
<P>
  <jsp:useBean id="test" class="com.powerise.edu.javabean.SimpleBean" />
  <jsp:setProperty name="test" property="message" value="Hello WWW" />
<H1>Message: <I>
  <jsp:getProperty name="test" property="message" />
</I></H1>
</BODY>
</HTML>
```

BeanTest 页面用到了一个 SimpleBean。SimpleBean 的代码如下：

```
package com.powerise.edu.javabean;

public class SimpleBean {
  private String message = "No message specified";

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }

  @Override
  public String toString() {
    // TODO Auto-generated method stub
    return super.toString();
  }

  /**
   * @param args
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
  }
}
```

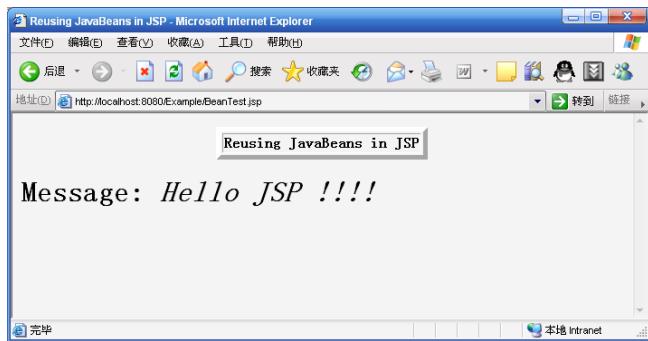


图 5-5

### 5.6.3. 关于 `jsp:useBean` 的进一步说明

使用 Bean 最简单的方法是先用下面的代码装载 Bean:

```
<jsp:useBean id="name" class="package.class" />
```

然后通过 `jsp:setProperty` 和 `jsp:getProperty` 修改和提取 Bean 的属性。不过有两点必须注意。第一，我们还可以用下面这种格式实例化 Bean:

```
<jsp:useBean ...>
  Body
</jsp:useBean>
```

它的意思是，只有当第一次实例化 Bean 时才执行 Body 部分，如果是利用现有的 Bean 实例则不执行 Body 部分。正如下面将要介绍的，`jsp:useBean` 并非总是意味着创建一个新的 Bean 实例。

第二，除了 `id` 和 `class` 外，`jsp:useBean` 还有其他三个属性，即：`scope`, `type`, `beanName`。下表简要说明这些属性的用法。

表 5-2 `jsp:useBean` 属性用法

属性	用法
<code>id</code>	命名引用该 Bean 的变量。如果能够找到 <code>id</code> 和 <code>scope</code> 相同的 Bean 实例， <code>jsp:useBean</code> 动作将使用已有的 Bean 实例而不是创建新的实例。
<code>class</code>	指定 Bean 的完整包名及类名。
<code>scope</code>	<p>指定 Bean 在哪种上下文内可用，可以取下面的四个值之一： <code>page</code>, <code>request</code>, <code>session</code> 和 <code>application</code>。</p> <p>默认值是 <code>page</code>，表示该 Bean 只在当前页面内可用（保存在当前页面的 <code>PageContext</code> 内）。</p> <p><code>request</code> 表示该 Bean 在当前的客户请求内有效（保存在 <code>ServletRequest</code> 对象内）。</p> <p><code>session</code> 表示该 Bean 对当前 <code>HttpSession</code> 内的所有页面都有效。</p> <p>最后，如果取值 <code>application</code>，则表示该 Bean 对所有具有相同 <code>ServletContext</code> 的页面都有效。</p> <p><code>scope</code> 之所以很重要，是因为 <code>jsp:useBean</code> 只有在不存在具有相同 <code>id</code> 和 <code>scope</code> 的对象时才会实例化新的对象；如果已有 <code>id</code> 和 <code>scope</code> 都相同的对象则直接使用已有的对象，此时 <code>jsp:useBean</code> 开始标记和结束标记之间的任何内容都将被忽略。</p>
<code>request</code>	指定引用该对象的变量的类型，它必须是 Bean 类的名字、超类名字、该类所实现的接口名字之一。请记住变量的名字是由 <code>id</code> 属性指定的。

session	指定 Bean 的名字。如果提供了 type 属性和 beanName 属性，允许省略 class 属性。
---------	---

## 5.6.4. jsp:setProperty 动作

jsp:setProperty 用来设置已经实例化的 Bean 对象的属性，有两种用法。首先，你可以在 jsp:useBean 元素的外面（后面）使用 jsp:setProperty，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
property="someProperty" ... />
```

此时，不管 jsp:useBean 是找到了一个现有的 Bean，还是新创建了一个 Bean 实例， jsp:setProperty 都会执行。第二种用法是把 jsp:setProperty 放入 jsp:useBean 元素的内部，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
property="someProperty" ... />
</jsp:useBean>
```

此时， jsp:setProperty 只有在新建 Bean 实例时才会执行，如果是使用现有实例则不执行 jsp:setProperty。

jsp:setProperty 动作有下面四个属性：

表 5-3 jsp:setProperty 属性用法

属性	说明
name	name 属性是必需的。它表示要设置属性的是哪个 Bean。
property	property 属性是必需的。它表示要设置哪个属性。有一个特殊用法：如果 property 的值是“*”，表示所有名字和 Bean 属性名字匹配的请求参数都将被传递给相应的属性 set 方法。
value	value 属性是可选的。该属性用来指定 Bean 属性的值。字符串数据会在目标类中通过标准的 valueOf 方法自动转换成数字、 boolean、 Boolean、 byte、 Byte、 char、 Character。例如， boolean 和 Boolean 类型的属性值（比如“true”）通过 Boolean.valueOf 转换， int 和 Integer 类型的属性值（比如“42”）通过 Integer.valueOf 转换。 value 和 param 不能同时使用，但可以使用其中任意一个。
param	param 是可选的。它指定用哪个请求参数作为 Bean 属性的值。如果当前请求没有参数，则什么事情也不做，系统不会把 null 传递给 Bean 属性的 set 方法。因此，你可以让 Bean 自己提供默认属性值，只有当请求参数明确指定了新值时才修改默认属性值。

例如，下面的代码片断表示：如果存在 numItems 请求参数的话，把 numberOfItems 属性的值设置为请求参数 numItems 的值；否则什么也不做。

```
<jsp:setProperty name="orderBean"
property="numberOfItems"
param="numItems" />
```

如果同时省略 value 和 param，其效果相当于提供一个 param 且其值等于 property 的值。

进一步利用这种借助请求参数和属性名字相同进行自动赋值的思想，你还可以在 `property`（Bean 属性的名字）中指定“\*”，然后省略 `value` 和 `param`。此时，服务器会查看所有的 Bean 属性和请求参数，如果两者名字相同则自动赋值。

下面是一个利用 JavaBean 计算素数的例子。如果请求中有一个 `numDigits` 参数，则该值被传递给 Bean 的 `numDigits` 属性；`numPrimes` 也类似。

### JspPrimes.jsp

---

```
<%@ page contentType="text/html; charset=gb2312" language="java" import="java.sql.*"
errorPage="" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>在 JSP 中使用 JavaBean</TITLE>
</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR>
<TH CLASS="TITLE"> 在 JSP 中使用 JavaBean
</TABLE>
</CENTER>
<P>
<jsp:useBean id="primeTable" class="com.powerise.edu.javabean.NumberedPrimes" />
<jsp:setProperty name="primeTable" property="numDigits" />
<jsp:setProperty name="primeTable" property="numPrimes" />
Some
<jsp:getProperty name="primeTable" property="numDigits" />
digit primes:
<jsp:getProperty name="primeTable" property="numberedList" />
</BODY>
</HTML>
```

---

### NumberedPrimes.java

---

```
package com.powerise.edu.javabean;

import java.util.List;

public class NumberedPrimes {
    private static int numDigits = 100;
    private int numPrimes = 0;

    @SuppressWarnings("unused")
    private List numberedList = new java.util.ArrayList();
```

```
private void init(int[] num) {
    int i = 0;
    for (; i < numDigits; i++) {
        num[i] = i + 1;
    }
}

private List getPrime() {
    int i = 2, j = 1;
    int[] numbers = new int[numDigits];
    init(numbers);
    while (j < numDigits) {
        if (numbers[j] != 0)
            while (i < numDigits) {
                if (numbers[i] != 0)
                    if (numbers[i] % numbers[j] == 0)
                        numbers[i] = 0;// 如果不是素数,置零
                i++;
            }
        j++;
        i = j + 1;
    }
    i = 0;
    int n = 0;
    for (; i < numDigits && n < numPrimes; i++)
        if (numbers[i] != 0) {
            numberedList.add(numbers[i]);
            n++;
        }
    return numberedList;
}

public static void main(String[] args) {
    new NumberedPrimes().getNumberedList();
}

}

public int getNumDigits() {
    return numDigits;
}

@SuppressWarnings("static-access")
public void setNumDigits(int numDigits) {
```

```
    this.numDigits = numDigits;  
}  
  
public int getNumPrimes() {  
    return numPrimes;  
}  
  
public void setNumPrimes(int numPrimes) {  
    this.numPrimes = numPrimes;  
}  
  
public List getNumberedList() {  
    return getPrime();  
}  
  
}
```

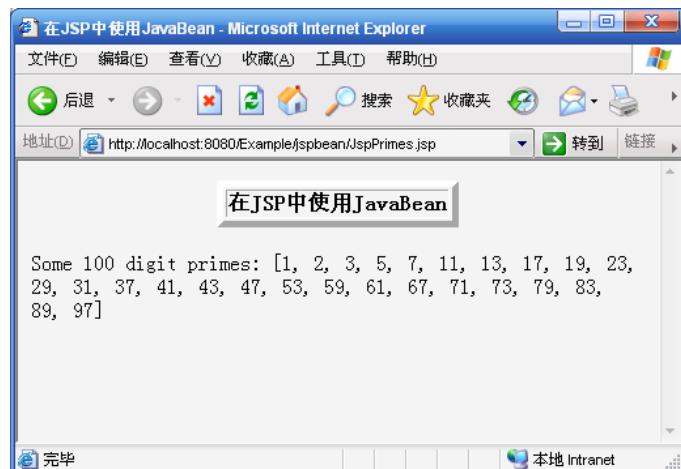


图 5-6

### 5.6.5. jsp:getProperty 动作

jsp:getProperty 动作提取指定 Bean 属性的值，转换成字符串，然后输出。jsp:getProperty 有两个必需的属性，即：name，表示 Bean 的名字；property，表示要提取哪个属性的值。下面是一个例子，更多的例子可以在前文找到。

```
<jsp:useBean id="itemBean" ... />  
...  
<UL>  
<LI>Number of items:  
<jsp:getProperty name="itemBean" property="numItems" />  
<LI>Cost of each:  
<jsp:getProperty name="itemBean" property="unitCost" />  
</UL>
```

## 5.6.6. jsp:forward 动作

jsp:forward 动作把请求转到另外的页面。jsp:forward 标记只有一个属性 page。page 属性包含的是一个相对 URL。page 的值既可以直接给出，也可以在请求的时候动态计算，如下面的例子所示：

```
<jsp:forward page="/utils/errorReporter.jsp" />  
<jsp:forward page="<% someJavaExpression %>" />
```

## 5.6.7. jsp:plugin 动作

jsp:plugin 动作用来根据浏览器的类型，插入通过 Java 插件 运行 Java Applet 所必需的 OBJECT 或 EMBED 元素。

## 5.6.8. 附录：JSP 注释和字符引用约定

下面是一些特殊的标记或字符，你可以利用它们插入注释或可能被视为具有特殊含义的字符。

表 5-4 JSP 注释和字符

语法	用途
<%-- comment --%>	JSP 注释，也称为“隐藏注释”。JSP 引擎将忽略它。标记内的所有 JSP 脚本元素、指令和动作都将不起作用。
<!-- comment -->	HTML 注释，也称为“输出的注释”，直接出现在结果 HTML 文档中。标记内的所有 JSP 脚本元素、指令和动作正常执行。
<%	在模板文本（静态 HTML）中实际上希望出现“%”的地方使用。
%>	在脚本元素内实际上希望出现“%>”的地方使用。
\'	使用单引号的属性内的单引号。不过，你既可以使用单引号也可以使用双引号，而另外一种引号将具有普通含义。
\"	使用双引号的属性内的双引号。参见“\” 的说明。

## 5.7. JSP/Servlet 应用程序优化

### 5.7.1. 在 HttpServlet init()方法中缓存数据

服务器会在创建 servlet 实例之后和 servlet 处理任何请求之前调用 servlet 的 init()方法。该方法在 servlet 的生命周期中仅调用一次。为了提高性能，在 init()中缓存静态数据或完成要在初始化期间完成的代价昂贵的操作。例如，一个最佳实践是使用实现了 javax.sql.DataSource 接口的 JDBC 连接池。

DataSource 从 JNDI 树中获得。每调用一次 SQL 就要使用 JNDI 查找 DataSource 是非常昂贵的工作，而且严重影响了应用的性能。Servlet 的 init()方法可以用于获取 DataSource 并缓存它以便之后的重用：

```
public class ControllerServlet extends HttpServlet  
{  
    private javax.sql.DataSource testDS=null;  
    public void init(ServletConfig config) throws ServletException  
    {  
        super.init(config);  
        Context ctx=null;
```

```
try
{
ctx=newInitialContext();
testDS=(javax.sql.DataSource)ctx.lookup("jdbc/testDS");
}
catch(NamingExceptionne)
{
ne.printStackTrace();
}
catch(Exceptionne)
{
e.printStackTrace();
}
}

public javax.sql.DataSource getTestDS()
{
return testDS;
}

...
...
```

## 5.7.2. 禁用 servlet 和 Jsp 的自动装载功能

当每次修改了 Servlet/JSP 之后，你将不得不重新启动服务器。由于自动装载功能减少开发时间，该功能被认为在开发阶段是非常有用的。但是，它在运行阶段是非常昂贵的；servlet/JSP 由于不必要的装载，增加类装载器的负担而造成很差的性能。同样，这会使你的应用由于已被某种类装载器装载的类不能和当前类装载器装载的类不能相互协作而出现奇怪的冲突现象。因此，在运行环境中为了得到更好的性能，关闭 servlet/JSP 的自动装载功能。

## 5.7.3. 控制 HttpSession

许多应用需要一系列客户端的请求，因此他们能互相相关联。由于 HTTP 协议是无状态的，所以基于 Web 的应用需要负责维护这样一个叫做 session 的状态。为了支持必须维护状态的应用，Javaservlet 技术提供了管理 session 和允许多种机制实现 session 的 API。HttpSession 对象扮演了 session，但是使用它需要成本。无论何时 HttpSession 被使用和重写，它都由 servlet 读取。你可以通过使用下面的技术来提高性能：

在 JSP 页面中不要创建默认的 HttpSession：默认情况下，JSP 页面创建 HttpSession。如果你在 JSP 页面中不用 HttpSession，为了节省性能开销，使用下边的页面指令可以避免自动创建 HttpSession 对象：

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

1) 不要将大的对象图存储在 HttpSession 中：如果你将数据当作一个大的对象图存储在 HttpSession 中，应用服务器每次将不得不处理整个 HttpSession 对象。这将迫使 Java 序列化和增加计算开销。由于序列化的开销，随着存储在 HttpSession 对象中数据对象的增大，系统的吞吐量将会下降。

2) 用完后释放 HttpSession：当不在使用 HttpSession 时，使用 HttpSession.invalidate()

方法使 session 失效。

3) 设置超时值: 一个 servlet 引擎有一个默认的超时值。如果你不删除 session 或者一直把 session 用到它超时的时候, servlet 引擎将把 session 从内存中删除。由于在内存和垃圾收集上的开销, session 的超时值越大, 它对系统弹性和性能的影响也越大。试着将 session 的超时值设置的尽可能低。

#### 5.7.4. 使用 gzip 压缩

压缩是删除冗余信息的作法, 用尽可能小的空间描述你的信息。使用 gzip (GNUzip) 压缩文档能有效地减少下载 HTML 文件的时间。你的信息量越小, 它们被送出的速度越快。因此, 如果你压缩了由你 web 应用产生的内容, 它到达用户并显示在用户屏幕上的速度就越快。不是任何浏览器都支持 gzip 压缩的, 但检查一个浏览器是否支持它并发送 gzip 压缩内容到浏览器是很容易的事情。下边的代码段说明了如何发送压缩的内容。

```
publicvoiddoGet(HttpServletRequestrequest,HttpServletResponseresponse)
throwsIOException,ServletException
{
OutputStreamout=null
//ChecktheAccepting-EncodingheaderfromtheHTTPRequest.
//Iftheheaderincludesgzip,chooseGZIP.
//Iftheheaderincludescompress,chooseZIP.
//Otherwisechoosenocompression.
Stringencoding=request.getHeader("Accept-Encoding");
if(encoding!=null&&encoding.indexOf("gzip")!=-1)
{
response.setHeader("Content-Encoding","gzip");
out=newGZIPOutputStream(response.getOutputStream());
}
elseif(encoding!=null&&encoding.indexOf("compress")!=-1)
{
response.setHeader("Content-Encoding","compress");
out=newZIPOutputStream(response.getOutputStream());
}
else
{
out=response.getOutputStream();
}
...
...
}
```

#### 5.7.5. 不要使用 SingleThreadModel

SingleThreadModel 保证 servlet 一次仅处理一个请求。如果一个 servlet 实现了这个接口, servlet 引擎将为每个新的请求创建一个单独的 servlet 实例, 这将引起大量的系统开销。如果你需要解决线程安全问题, 请使用其他的方法替代这个接口。SingleThreadModel 在 Servlet2.4 中是不再提倡使用。

## 5.7.6. 使用线程池

servlet 引擎为每个请求创建一个单独的线程，将该线程指派给 service()方法，然后在 service()方法执行完后删除该线程。默认情况下，servlet 引擎可能为每个请求创建一个新的线程。由于创建和删除线程的开销是很昂贵的，于是这种默认行为降低了系统的性能。我们可以使用线程池来提高性能。根据预期的并发用户数量，配置一个线程池，设置好线程池里的线程数量的最小和最大值以及增长的最小和最大值。起初，servlet 引擎创建一个线程数与配置中的最小线程数量相等的线程池。然后 servelt 引擎把池中的一个线程指派给一个请求而不是每次都创建新的线程，完成操作之后，servelt 引擎把线程放回到线程池中。使用线程池，性能可以显著地提高。如果需要，根据线程的最大数和增长数，可以创建更多的线程。

## 5.7.7. 选择正确的包括机制

在 JSP 页面中，有两中方式可以包括文件：包括指令(<% @include file="test.jsp"%>) 和包括动作(<jsp:include page="test.jsp" flush="true"/>)。包括指令在编译阶段包括一个指定文件的内容；例如，当一个页面编译成一个 servlet 时。包括动作是指在请求阶段包括文件内容；例如，当一个用户请求一个页面时。包括指令要比包括动作快些。因此除非被包括的文件经常变动，否则使用包括指令将会获得更好的性能。

## 5.7.8. 在 useBean 动作中使用合适的范围

使用 JSP 页面最强大方式之一是和 JavaBean 组件协同工作。JavaBean 使用<jsp:useBean>标签可以嵌入到 JSP 页面中。语法如下：

```
<jsp:useBean id="name" scope="page|request|session|application" class="package.className"
type="typeName">
</jsp:useBean>
```

scope 属性说明了 bean 的可见范围。scope 属性的默认值是 page。你应该根据你应用的需求选择正确的范围，否则它将影响应用的性能。

例如，如果你需要一个专用于某些请求的对象，但是你把范围设置成了 session，那么那个对象将在请求结束之后还保留在内存中。它将一直保留在内存中除非你明确地把它从内存中删除、使 session 无效或 session 超时。如果你没有选择正确的范围属性，由于内存和垃圾收集的开销将会影响性能。因此为对象设置合适的范围并在用完它们之后立即删除。

## 5.7.9. 其他技术

- 避免字符串连接：由于 String 对象是不可变对象，使用“+”操作符将会导致创建大量的临时对象。你使用的“+”越多，产生的临时对象就越多，这将影响性能。当你需要连接字符串时，使用 StringBuffer 替代“+”操作。
- 避免使用 System.out.println：System.out.println 同步处理磁盘输入/输出，这大大地降低了系统吞吐量。尽可能地避免使用 System.out.println。尽管有很多成熟的调试工具可以用，但有时 System.out.println 为了跟踪、或调试的情况下依然很有用。你应该配置 System.out.println 仅在错误和调试阶段打开它。使用 final Boolean 型的变量，当配置成 false 时，在编译阶段完成优化检查和执行跟踪输出。
- ServletOutputStream 与 PrintWriter 比较：由于字符输出流和把数据编码成字节，使用 PrintWriter 引入了小的性能开销。因此，PrintWriter 应该用在所有的字符集都正确地转换做完之后。另一方面，当你知道你的 servlet 仅返回二进制数据，使用 ServletOutputStream，因为 servlet 容器不编码二进制数据，这样你就能消除字符集

转换开销。

## 5.8. 实例：bookstore 应用

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

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

bookstore 应用的软件结构如下图所示：

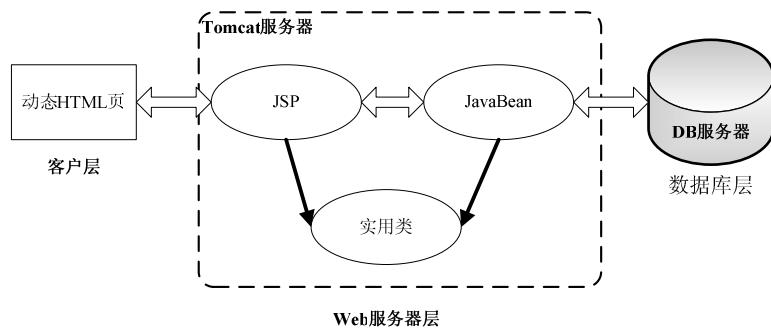


图 5-7：bookstore 应用的软件结构

### 5.8.1. Web 服务器层

开发 bookstore 应用最主要的工作就是开发 Web 服务器层组件。构成 Web 服务器层组件的具体文件名如下表所示：

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

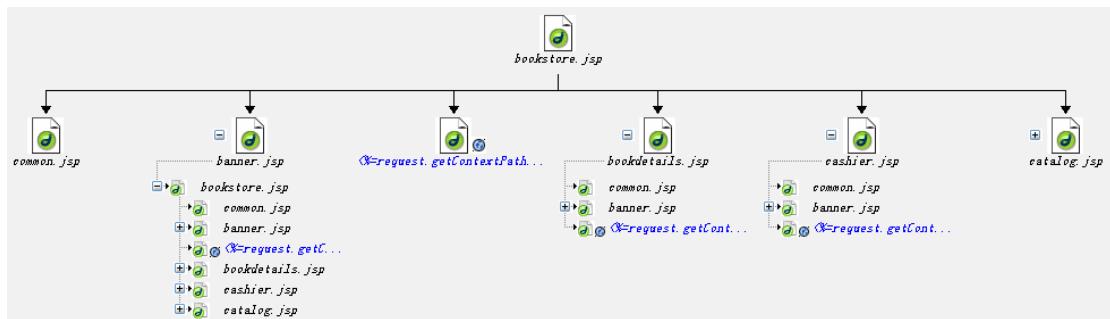
### 5.8.2. 数据库层

bookstore 应用采用 Ms Sql Server2000 作为数据库服务器。网上书店中所有书的信息存放在数据库的 books 表中。books 表的字段参见下表：

字段	类型	描述
id	Varchar(8)	书的 ID 号, primary key

name	Varchar(24)	作者姓名
title	Varchar(96)	书的名字
price	money	价格
yr	int	出版年份
description	Varchar(30)	书的描述信息
saleAmount	int	销售数量

**注意：** Ms Sql Server2000 需要打上 SP3 才能利用 JDBC 正常访问。



冬 5-8

### 5.8.2.1. banner.jsp

`banner.jsp` 用于显示书店的 logo，它是所有网页的公共部分。在其他 JSP 网页中，可以通过 `<%@include>` 标记将 `banner.jsp` 包含进去。代码如下：

```
<a href="bookstore.jsp">  </a><hr />
```

### **5.8.2.2. common.jsp**

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

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

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

代码如下：

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

### 5.8.2.3. bookstore.jsp

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

通过 <http://localhost:8080/bookstore0/bookstore.jsp>, 进入网上书店主页, 如下图所示:



图 5-9

#### 5.8.2.4. bookdetails.jsp

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

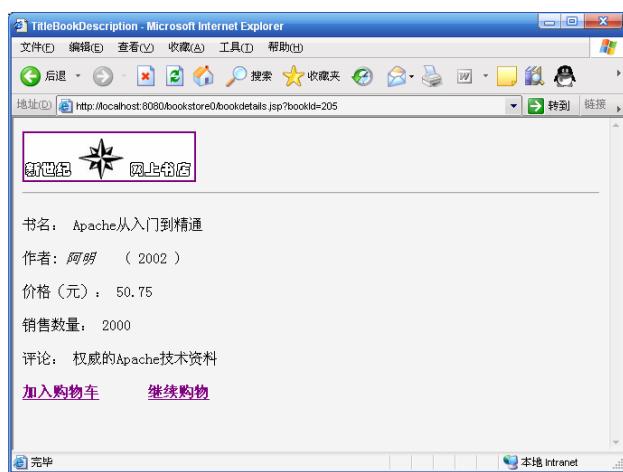


图 5-10

如果客户查询的书的编号在数据库中不存在，bookdetails.jsp 将显示提示信息，如下图所示：

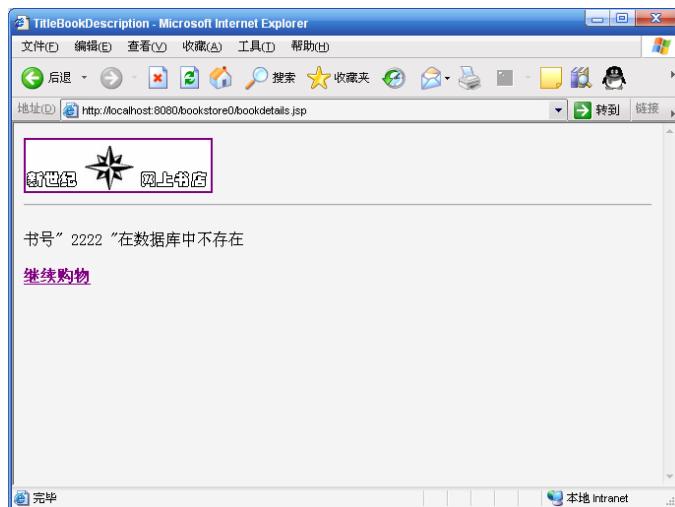


图 5-11

```

<%@ page contentType="text/html; charset=GB2312"%>
<%@ include file="common.jsp"%>
<%@ page import="java.util.*"%>

<html>
    <head>
        <title>TitleBookDescription</title>
    </head>
    <body>
        <%@ include file="banner.jsp"%>
        <br>
        <%
            //Get the identifier of the book to display
            String bookId = request.getParameter("bookId");
            if (bookId == null)
                bookId = "201";
            BookDetails book = bookDB.getBookDetails(bookId);
        %>

        <%
            if (book == null) {
        %>
        <p>
            书号"
            <%=bookId%>
            "在数据库中不存在
        <p>
            <strong><a href="<%=request.getContextPath()%>/catalog.jsp">继续购物
        </a>
        </strong>
    
```

```
<%
    return;
}

%>

<p>
    书名:
    <%=book.getTitle()%>
</p>
作者:
<em><%=book.getName()%> </em>&nbsp;&nbsp; (
<%=book.getYear()%>
)
<br>
<p>
    价格(元):
    <%=book.getPrice()%>
</p>
<p>
    销售数量:
    <%=book.getSaleAmount()%>
</p>
<p>
    评论:
    <%=book.getDescription()%>
</p>
<p>
    <strong><a
        href="" href=""/catalog.jsp?Add=<%=bookId%>">加入购物车</a>&nbsp;
        &nbsp; &nbsp; <a href=""/catalog.jsp">">继续购物</a>
    </p>
    </strong>
</body>
</html>
```

### 5.8.2.5. catalog.jsp

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

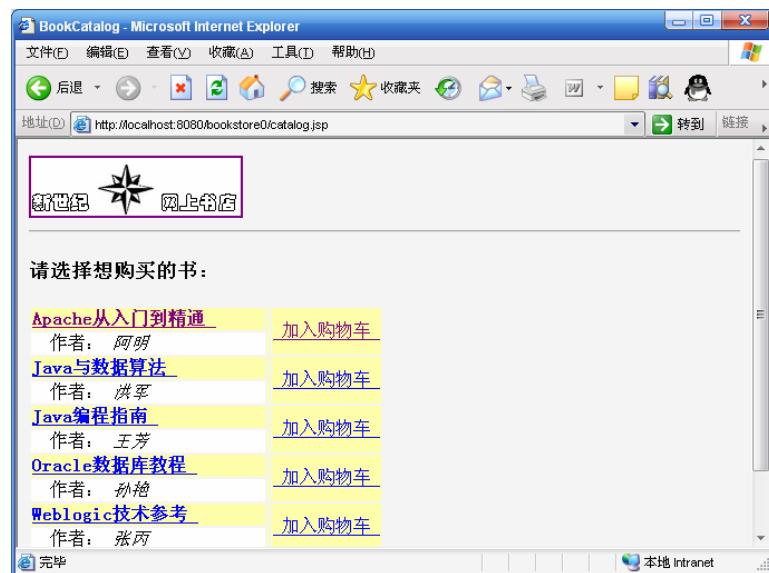


图 5-12

```

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

<%@ include file="common.jsp"%>
<%@ page import="java.util.*"%>

<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart" />

<html>
    <head>
        <title>BookCatalog</title>
    </head>
    <body>
        <%@ include file="banner.jsp"%>

        <%
            // Additions to the shopping cart
            String bookId = request.getParameter("Add");
            if (bookId != null) {
                BookDetails book = bookDB.getBookDetails(bookId);
                cart.add(bookId, book);
            }
        %>
        <p>
            <h3>
                <font color="red"> 您已将 <i><%=book.getTitle()%>
                </i> 加入购物车</font>
            </h3>
        <%
            if (cart.getNumberofItems() > 0) {

```

```
%>

<p>
    <strong><a href="<%=request.getContextPath()%>/showcart.jsp">察看购物
车</a>&nbsp;&nbsp;&nbsp;
        <a href="<%=request.getContextPath()%>/cashier.jsp">付账</a>
    </p>
</strong>

<%
}
%>

<h3>
    请选择想购买的书:
</h3>

<table>
<%
    Collection c = bookDB.getBooks();
    Iterator i = c.iterator();
    while (i.hasNext()) {
        BookDetails book = (BookDetails) i.next();
        bookId = book.getBookId();
    }
%>

<tr>
    <td bgcolor="#ffffaa">
        <a
            href="<%=request.getContextPath()%>/bookdetails.jsp?bookId=<%=bookId%>"><stron
g>
                <%=book.getTitle()%>&nbsp;</strong>
            </a>
        </td>

        <td bgcolor="#ffffaa" rowspan=2>
        <td bgcolor="#ffffaa" rowspan=2>
            <a
                href="<%=request.getContextPath()%>/catalog.jsp?Add=<%=bookId%>">&nbsp;加入购物车
            &nbsp;</a>
        </td>
    </td>
</tr>
```

```

<tr>
    <td bgcolor="#ffffff">
        &nbsp;&nbsp;作者:
        <em><%=book.getName()%>&nbsp; </em>
    </td>
</tr>

<%
}
%>

</table>

</body>
</html>

```

### 5.8.2.6. showcart.jsp

showcart.jsp 用于显示客户购物车中的信息，如下图所示，客户可以通过选择“删除”链接从购物车中删除一本书，也可以通过选择“清空购物车”链接删除所有购买的书目。showcart.jsp 中的“继续购物”链接转到 catalog.jsp。“付帐”链接转到 cashier.jsp。



图 5-13

```

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

<%@ include file="common.jsp"%>
<%@ page import="java.util.*"%>

<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart" />

<html>
    <head>
        <title>TitleShoppingCart</title>

```

```
</head>
<body>
<%@ include file="banner.jsp"%>
<%
    String bookId = request.getParameter("Remove");
    if (bookId != null) {
        cart.remove(bookId);
        BookDetails book = bookDB.getBookDetails(bookId);
    }
%>

<font color="red" size="+2">您删除了一本书: <em><%=book.getTitle()%>
</em> <br>&nbsp;<br> </font>

<%
    }
    if (request.getParameter("Clear") != null) {
        cart.clear();
    }
%>

<font color="red" size="+2"><strong> 清空购物车 </strong>
<br>&nbsp;<br>
</font>

<%
    }
    // Print a summary of the shopping cart
    int num = cart.getNumberofItems();
    if (num > 0) {
%>

<font size="+2">您的购物车內有<%=num%>本书 </font>
<br>
&nbsp;

<table>
    <tr>
        <th align=left>
            数量
        </th>
        <th align=left>
            书名
        </th>
        <th align=left>
```

价格

```
</th>
</tr>

<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item = (ShoppingCartItem) i.next();
        BookDetails book = (BookDetails) item.getItem();
    }
%>

<tr>
    <td align="right" bgcolor="#ffffff">
        <%=item.getQuantity()%>
    </td>

    <td bgcolor="#ffffaa">
        <strong><a href="<%=request.getContextPath()%>/bookdetails.jsp?bookId=<%=book.getId()%>">
            <%=book.getTitle()%>
        </a>
        </strong>
    </td>

    <td bgcolor="#ffffaa" align="right">
        <%=book.getPrice()%>
    </td>

    <td bgcolor="#ffffaa">
        <strong> <a href="<%=request.getContextPath()%>/showcart.jsp?Remove=<%=book.getId()%>">
            >删除</a>
        </strong>
    </td>
</tr>

<%
    // End of while
%>
```

```
<tr>
    <td colspan="5" bgcolor="#ffffff">
        <br>
    </td>
</tr>

<tr>
    <td colspan="2" align="right" bgcolor="#ffffff">
        总额(元)
    </td>
    <td bgcolor="#ffffaa" align="right">
        <%=cart.getTotal()%>
    </td>
    <td>
        <br>
    </td>
</tr>

</table>

<p>
    &nbsp;
<p>
    <strong><a href="<%=request.getContextPath()%>/catalog.jsp">继续购物
</a>&nbsp;&nbsp;&nbsp;
        <a href="<%=request.getContextPath()%>/cashier.jsp">付账
</a>&nbsp;&nbsp;&nbsp;
        <a href="<%=request.getContextPath()%>/showcart.jsp?Clear=clear">清空
购物车</a>
    </strong>
    <%
    } else {
    %>

    <font size="+2">您的购物车目前为空</font>
    <br>
    &nbsp;
    <br>
    <a href="<%=request.getContextPath()%>/catalog.jsp">继续购物</a>

    <%
    // End of if
    %
%>
```

```
</body>  
</html>
```

### 5.8.2.7. cashier.jsp

cashier.jsp 提供了客户输入信用卡信息的表单，如下图所示。客户点击【提交】按钮，就会转到 receipt.jsp 网页。

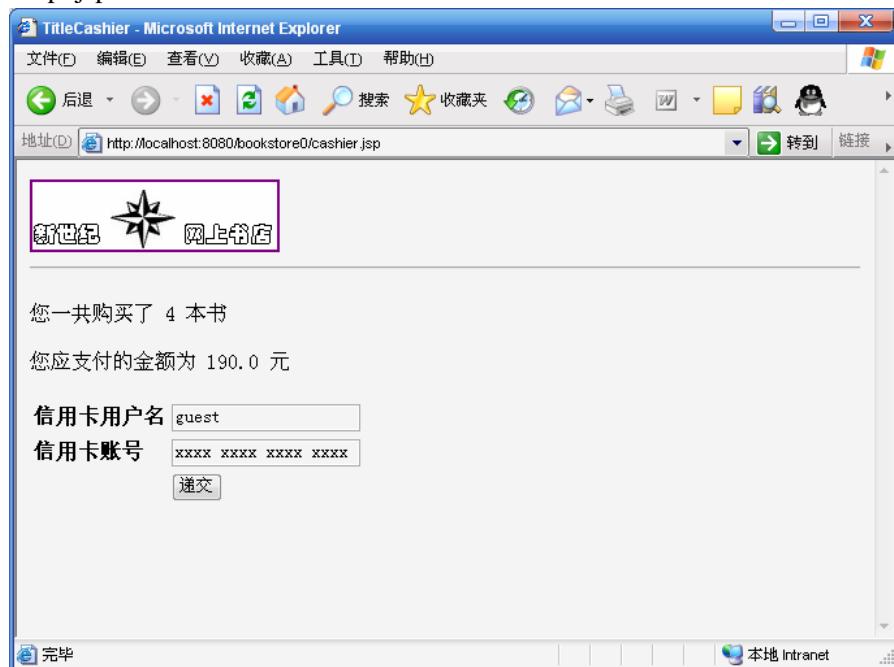


图 5-14

```
<%@ page contentType="text/html; charset=GB2312"%>  
  
<%@ include file="common.jsp"%>  
<%@ page import="java.util.*"%>  
  
<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart" />  
  
<html>  
<head>  
<title>TitleCashier</title>  
</head>  
<body>  
<%@ include file="banner.jsp"%>  
<p>  
    您一共购买了  
<%=cart.getNumberOfItems()%>  
    本书  
</P>  
<p>
```

您应支付的金额为  
<%=cart.getTotal()%>  
元  
</p>

```
<form action="<%="request.getContextPath()%>/receipt.jsp" method="post">
<table>
<tr>
<td>
<strong>信用卡用户名</strong>
</td>
<td>
<input type="text" name="cardname" value="guest" size="19">
</td>
</tr>
<tr>
<td>
<strong>信用卡账号</strong>
</td>
<td>
<input type="text" name="cardnum" value="xxxx xxxx xxxx
xxxx"
size="19">
</td>
</tr>
<tr>
<td></td>
<td>
<input type="submit" value="递交">
</td>
</tr>
</table>
</form>
</body>
</html>
```

### 5.8.2.8. receipt.jsp

receipt.jsp 结束当前的 Session，和客户告别，并且提供了继续购物的链接，如下图所示。客户选择“继续购物”，又会转到 bookstore 的首页。

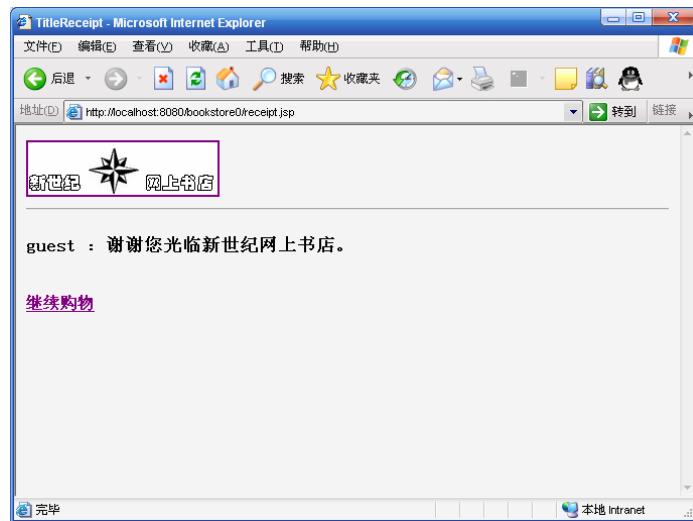


图 5-15

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

<%@ include file="common.jsp"%>
<%@ page import="java.util.*"%>

<jsp:useBean id="cart" scope="session" class="mypack.ShoppingCart" />

<%
    bookDB.buyBooks(cart);
    // Payment received -- invalidate the session
    session.invalidate();
%>
<html>
    <head>
        <title>TitleReceipt</title>
        <%@ include file="banner.jsp"%>
    </head>
<body>

    <h3>
        <%=request.getParameter("cardname")%>
        : 谢谢您光临新世纪网上书店。
    </h3>
    <br>
    <strong><a href="<%=request.getContextPath()%>/bookstore.jsp">继续购物</a>&nbsp;&nbsp;&nbsp;</strong>
</body>
</html>
```

### 5.8.2.9. errorpage.jsp

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

```
<%@ page import="java.io.*" %>
<!--设置中文输出-->
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page isErrorPage="true" %>
<html><head><title>Error Page</title></head>
<body>
<p>
    服务器端发生错误:<%= exception.getMessage() %>
</p>
<p>
    错误原因为: <% exception.printStackTrace(new PrintWriter(out));%>
</p>
</body>
</html>
```

当浏览器正在链接 bookdetails.jsp 网页时，如果数据库服务器关闭，将会导致数据库连接异常，异常发生后，客户请求由 errorpage.jsp 处理，如下图所示：

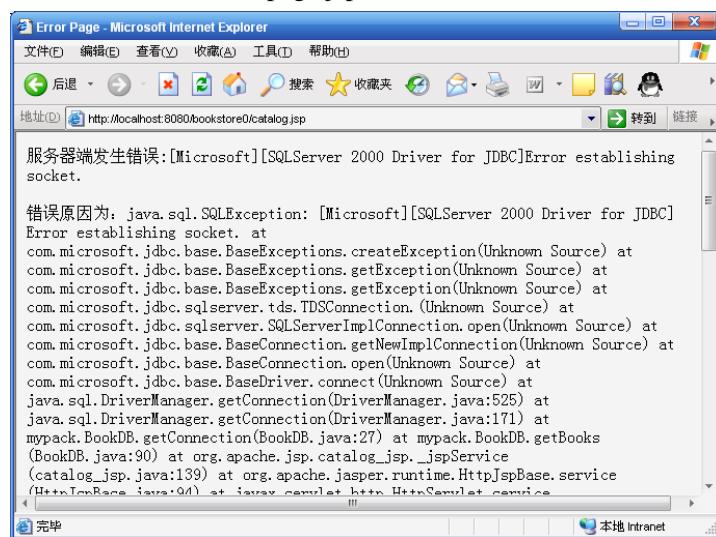


图 5-16

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

### 5.8.3. JavaBean 和实用类

在 bookstore 应用中，我们创建了以下类：

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

- ShoppingCartItem.java

### 5.8.3.1. 实体类

BookDB.java 实现:

```
package mypack;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;

public class BookDB {

    private ArrayList books;

    private String dbUrl =
"jdbc:microsoft:sqlserver://127.0.0.1:1433;databaseName=BookDB";

    private String dbUser = "sa";

    private String dbPwd = "sa";

    public BookDB() throws Exception {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
    }

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

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

    public void closePrepStmt(PreparedStatement prepStmt) {
        try {
```

```
        if (prepStmt != null)
            prepStmt.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

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

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

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

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

public Collection getBooks() throws Exception {
```

```
Connection con = null;
PreparedStatement prepStmt = null;
ResultSet rs = null;
books = new ArrayList();
try {
    con = getConnection();
    String selectStatement = "select * from books";
    prepStmt = con.prepareStatement(selectStatement);
    rs = prepStmt.executeQuery();

    while (rs.next()) {

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

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

Collections.sort(books);
return books;
}

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

        if (rs.next()) {
            BookDetails bd = new BookDetails(rs.getString(1), rs
                .getString(2), rs.getString(3), rs.getFloat(4), rs
                .getInt(5), rs.getString(6), rs.getInt(7));
            prepStmt.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        closeResultSet(rs);
        closePrepStmt(prepStmt);
        closeConnection(con);
    }
}
```

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

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

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

```
prepStmt.setString(1, bookId);
rs = prepStmt.executeQuery();

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

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

BookDetails 代表了具体的一本书，它的属性和 books 表字段对应。以下是源代码：

```
package mypack;

public class BookDetails implements Comparable {
    private String bookId = null;

    private String title = null;

    private String name = null;

    private float price = 0.0F;

    private int year = 0;

    private String description = null;

    private int saleAmount;

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

```
    this.year = year;
    this.description = description;
    this.saleAmount = saleAmount;

}

public String getTitle() {
    return title;
}

public float getPrice() {
    return price;
}

public int getYear() {
    return year;
}

public String getDescription() {
    return description;
}

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

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

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

public int compareTo(0bject o) {
    BookDetails n = (BookDetails) o;
    int lastCmp = title.compareTo(n.title);
    return (lastCmp);
}
}
```

### 5.8.3.2. 购物车的实现

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

例如，某个客户的购物车内包含如下内容：

- 《Java 编程指南》2 本；
- 《Apache 从入门到精通》3 本

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

```
Object item;
```

```
Int quantity;
```

item 变量代表客户购买的书，它引用 BookDetails 对象，quantity 表示书的数量。在这个购物模型中 ShoppingCart、ShoppingCartItem 和 BookDetails 对象之间的关系如下图所示：

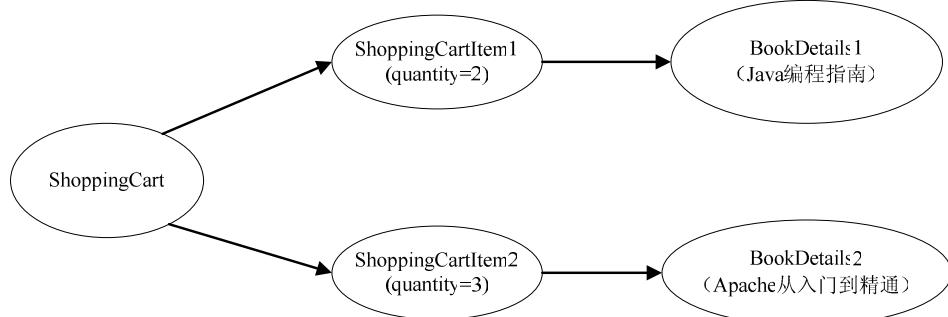


图 5-17

以下是 ShoppingCartItem 的源码：

```

package mypack;

public class ShoppingCartItem {
    Object item;
    int quantity;

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

    public void incrementQuantity() {
        quantity++;
    }

    public void decrementQuantity() {
        quantity--;
    }

    public Object getItem() {
        return item;
    }

    public int getQuantity() {
    }
}
  
```

```
        return quantity;
    }
}
```

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

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

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

- public synchronized void remove(String bookId)

从购物车中删除一本书，即将这本书的 ShoppingCartItem 的 quantity 减 1。如果 ShoppingCartItem 的 quantity 小于或者等于 0，就把 ShoppingCartItem 的 ShoppingCart 中删除。

- public synchronized Collection getItems()

从购物车中返回所有的 ShoppingCartItem 对象的集合。

- public synchronized int getNumberOfItems()

返回购物车中所有书的数量。

- public synchronized double getTotal()

返回购物车中所有书的总金额。

ShoppingCart 的源代码如下：

```
package mypack;
```

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;

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

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

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

```
public synchronized void remove(String bookId) {  
    if(items.containsKey(bookId)) {  
        ShoppingCartItem scitem = (ShoppingCartItem) items.get(bookId);  
        scitem.decrementQuantity();  
  
        if(scitem.getQuantity() <= 0)  
            items.remove(bookId);  
  
        numberofItems--;  
    }  
}  
  
public synchronized Collection getItems() {  
    return items.values();  
}  
  
protected void finalize() throws Throwable {  
    items.clear();  
}  
  
public synchronized int getNumberofItems() {  
    return numberofItems;  
}  
public synchronized double getTotal() {  
    double amount = 0.0;  
  
    for(Iterator i = getItems().iterator(); i.hasNext(); ) {  
        ShoppingCartItem item = (ShoppingCartItem) i.next();  
        BookDetails bookDetails = (BookDetails) item.getItem();  
  
        amount += item.getQuantity() * bookDetails.getPrice();  
    }  
    return roundOff(amount);  
}  
  
private double roundOff(double x) {  
    long val = Math.round(x*100); // cents  
    return val/100.0;  
}  
  
public synchronized void clear() {  
    items.clear();  
    numberofItems = 0;
```

```
}
```

## 5.8.4. 发布 bookstore 应用

bookstore 应用在 Eclipse 工程中的展开图如下图所示：

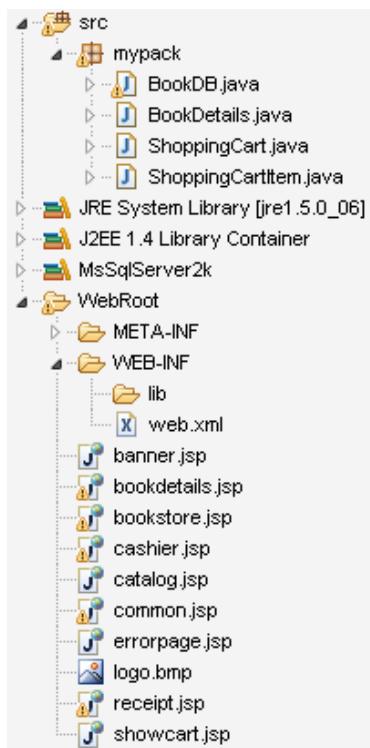


图 5-18

发布 bookstore 应用的方式可以参考“创建和发布 Web 应用”一节中的步骤。

## 5.9. 小结

JSP 是一种基于 Java 的脚本技术，它主要任务是将 HTML 编码从 Web 页面的业务逻辑中有效的分离出来。本章通过一系列的例子介绍了 JSP 的使用，JSP 文件可以包含：JSP 指令（或称为指示语句）、JSP 声明、直接插入 Java 代码（Scriptlet）、变量数据的 Java 表达式和 JSP 预定义变量。本章的重点内容为 JSP 预定义变量和 JSP 的动作。本章还结合前面的内容介绍了一些 JSP/Servlet 应用程序优化的技巧。

现在我们已经掌握了 JSP/Servlet 的基本使用方法，我们使用 JSP 结合 JavaBean 和实用类的方式开发了一个 bookstore 应用，这个应用里面覆盖了一部分前面的知识点，同时引入了 JDBC 的知识内容。在后续的章节里面我们将陆续介绍 bookstore 应用涉及的 Web 技术。本章 bookstore 应用的项目为 bookstore0，后面我们将继续完善 bookstore 应用，完善后的版本完成的功能和提供的用户界面都是一致的，但是实现方式上有所差别。

# 第6章 JDBC 技术

## 6.1. 概述

JDBC 是一种用于执行 SQL 语句的 Java API（有意思的是，JDBC 本身是个商标名而不是一个缩写字；然而，JDBC 常被认为代表“Java 数据库连接（Java Database Connectivity）”。它由一组用 Java 编程语言编写的类和接口组成。JDBC 为工具/数据库开发人员提供了一个标准的 API，使他们能够用纯 Java API 来编写数据库应用程序。

有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。换言之，有了 JDBC API，就不必为访问 Sybase 数据库专门写一个程序，为访问 Oracle 数据库又专门写一个程序，为访问 Informix 数据库又写另一个程序，等等。您只需用 JDBC API 写一个程序就够了，它可向相应数据库发送 SQL 语句。而且，使用 Java 编程语言编写的应用程序，就无须去忧虑要为不同的平台编写不同的应用程序。将 Java 和 JDBC 结合起来将使程序员只须写一遍程序就可让它在任何平台上运行。

Java 具有坚固、安全、易于使用、易于理解和可从网络上自动下载等特性，是编写数据库应用程序的杰出语言。所需要的只是 Java 应用程序与各种不同数据库之间进行对话的方法。而 JDBC 正是作为此种用途的机制。

JDBC 扩展了 Java 的功能。例如，用 Java 和 JDBC API 可以发布含有 applet 的网页，而该 applet 使用的信息可能来自远程数据库。企业也可以用 JDBC 通过 Intranet 将所有职员连到一个或多个内部数据库中（即使这些职员所用的计算机有 Windows、Macintosh 和 UNIX 等各种不同的操作系统）。随着越来越多的程序员开始使用 Java 编程语言，对从 Java 中便捷地访问数据库的要求也在日益增加。

MIS 管理员们都喜欢 Java 和 JDBC 的结合，因为它使信息传播变得容易和经济。企业可继续使用它们安装好的数据库，并能便捷地存取信息，即使这些信息是储存在不同数据库管理系统上。新程序的开发期很短。安装和版本控制将大为简化。程序员可只编写一遍应用程序或只更新一次，然后将它放到服务器上，随后任何人就都可得到最新版本的应用程序。对于商务上的销售信息服务，Java 和 JDBC 可为外部客户提供获取信息更新的更好方法。

## 6.2. JDBC 的设计目的

**ODBC:** 微软的 ODBC 是用 C 编写的，而且只适用于 Windows 平台，无法实现跨平台地操作数据库。

**SQL 语言:** SQL 尽管包含有数据定义、数据操作、数据管理等功能，但它并不是一个完整的编程语言，而且不支持流控制，需要与其它编程语言相配合使用。

**JDBC 的设计:** 由于 Java 语言具有健壮性、安全、易使用并自动下载到网络等方面的优点，因此如果采用 Java 语言来连接数据库，将能克服 ODBC 局限于某一系统平台的缺陷；将 SQL 语言与 Java 语言相互结合起来，可以实现连接不同数据库系统，即使用 JDBC 可以很容易地把 SQL 语句传送到任何关系型数据库中。

**JDBC 设计的目的:** 它是一种规范，设计出它的最主要的目的是让各个数据库开发商为 Java 程序员提供标准的数据库访问类和接口，使得独立于 DBMS 的 Java 应用程序的开发成为可能（数据库改变，驱动程序跟着改变，但应用程序不变）。

## 6.3. JDBC 的主要功能

- 创建与数据库的连接；
- 发送 SQL 语句到任何关系型数据库中；

- 处理数据并查询结果。

编程实例：

---

```
.....
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // (1) 创建与数据库的连接

    Connection
    con=DriverManager.getConnection("jdbc:odbc:DatabaseDSN","Login","Password");

    Statement stmt=con.createStatement();

    ResultSet rs=stmt.executeQuery("select * from DBTableName");// (2) 发送 SQL 语句到数据
    库中

    while(rs.next())

    { String name=rs.getString("Name") ;           // (3) 处理数据并查询结果。

        int age=rs.getInt("age");

        float wage=rs.getFloat("wage");

    }

    rs.close();                                     // (4)
    关闭

    stmt.close();

    con.close();

}

catch(SQLException e)

{     System.out.println("SQLState:"+ e.getSQLState());

    System.out.println("Message:" + e.getMessage());

    System.out.println("Vendor:" + e.getErrorCode());

}

.....
```

---

## 6.4. 与 ODBC 相比 JDBC 特点

- ODBC 是用 C 语言编写的，不是面向对象的；而 JDBC 是用 Java 编写的，是面向对象的。
- ODBC 难以学习，因为它把简单的功能与高级功能组合在一起，即便是简单的查询也会带有复杂的任选项；而 JDBC 的设计使得简单的事情用简单的方法来完成。
- ODBC 是局限于某一系统平台的，而 JDBC 提供 Java 与平台无关的解决方案。
- 但也可以通过 Java 来操作 ODBC，这可以采用 JDBC-ODBC 桥接方式来实现（因为 Java 不能直接使用 ODBC，即在 Java 中使用本地 C 的代码将带来安全缺陷）。

## 6.5. JDBC 结构

JDBC 的设计师为 JDBC 建立了一个分层的结构，在开发人员如何配置 JDBC，而不需要改变应用程序代码方面提供了很大的灵活性。下图说明了 JDBC API 的分层结构。

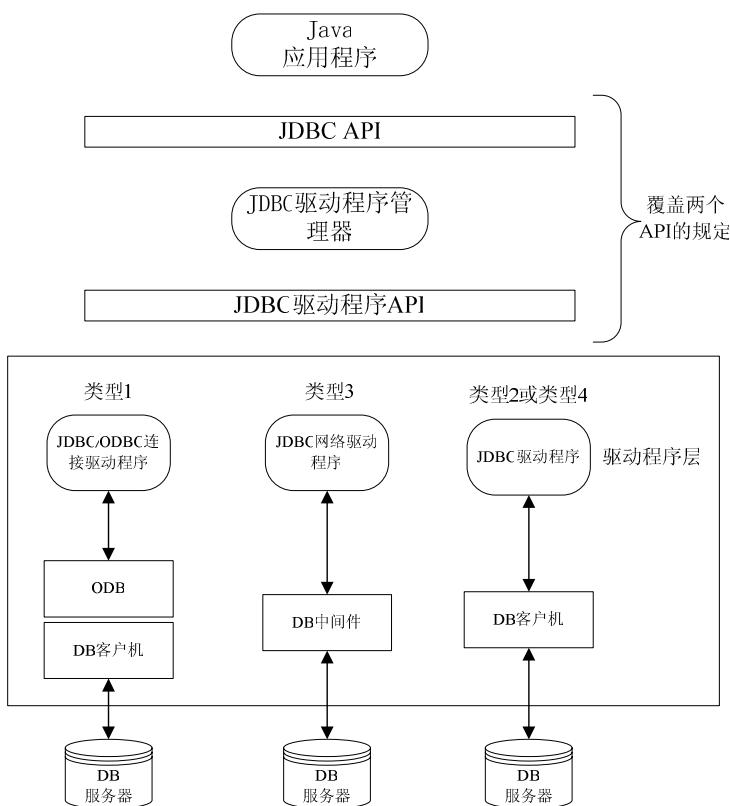


图 6-1: JDBC 结构

如图中所示，多种策略被用于实现 Driver 层；这四种策略对应 JDBC 规范中描述的四种驱动器类型。驱动器类型定义如下：

### (1) JDBC-ODBC 桥加 ODBC 驱动程序

JavaSoft 桥产品利用 ODBC 驱动程序提供 JDBC 访问。注意，必须将 ODBC 二进制代码（许多情况下还包括数据库客户机代码）加载到使用该驱动程序的每个客户机上。因此，这种类型的驱动程序最适合于企业网（这种网络上客户机的安装不是主要问题），或者是用 Java 编写的应用程序服务器代码。

JDBC-ODBC 桥接方式利用微软的开放数据库互连接口(ODBC API)同数据库服务器通讯，客户端计算机首先应该安装并配置 ODBC driver 和 JDBC-ODBC bridge 两种驱动程序。

### (2) 本地 API

这种类型的驱动程序把客户机 API 上的 JDBC 调用转换为 Oracle、Sybase、Informix、

DB2 或其它 DBMS 的调用。注意，象桥驱动程序一样，这种类型的驱动程序要求将某些二进制代码加载到每台客户机上。

这种驱动方式将数据库厂商的特殊协议转换成 Java 代码及二进制类码，使 Java 数据库客户方与数据库服务器方通信。例如：Oracle 用 SQLNet 协议, DB2 用 IBM 的数据库协议。数据库厂商的特殊协议也应该被安装在客户机上。

### (3) JDBC 网络纯 Java 驱动程序

这种驱动程序将 JDBC 转换为与 DBMS 无关的网络协议，之后这种协议又被某个服务器转换为一种 DBMS 协议。这种网络服务器中间件能够将它的纯 Java 客户机连接到多种不同的数据库上。所用的具体协议取决于提供者。通常，这是最为灵活的 JDBC 驱动程序。有可能所有这种解决方案的提供者都提供适合于 Intranet 用的产品。为了使这些产品也支持 Internet 访问，它们必须处理 Web 所提出的安全性、通过防火墙的访问等方面的额外要求。几家提供者正将 JDBC 驱动程序加到他们现有的数据库中间件产品中。

这种方式是纯 Java driver。数据库客户以标准网络协议(如 HTTP、SHTTP)同数据库访问服务器通信，数据库访问服务器然后翻译标准网络协议成为数据库厂商的专有特殊数据库访问协议(也可能用到 ODBC driver)与数据库通信。对 Internet 和 Intranet 用户而言这是一个理想的解决方案。Java driver 被自动的，以透明的方式随 Applets 自 Web 服务器而下载并安装在用户的计算机上。

### (4) 本地协议纯 Java 驱动程序

这种类型的驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议。这将允许从客户机机器上直接调用 DBMS 服务器，是 Intranet 访问的一个很实用的解决方法。

这种方式也是纯 Java driver。数据库厂商提供了特殊的 JDBC 协议使 Java 数据库客户与数据库服务器通信。然而，将把代理协议同数据库服务器通信改用数据库厂商的特殊 JDBC driver。这对 Intranet 应用是高效的，可是数据库厂商的协议可能不被防火墙支持，缺乏防火墙支持在 Internet 应用中会存在潜在的安全隐患。

## 6.6. JDBC 工作原理

JDBC 主要包含两部分：面向 Java 程序员的 JDBC API 及面向数据库厂商的 JDBC Driver API。

(1) 面向 Java 程序员的 JDBC API：Java 程序员通过调用此 API 从而实现连接数据库、执行 SQL 语句并返回结果集等编程数据库的能力，它主要是由一系列的接口定义所构成，如下图所示：

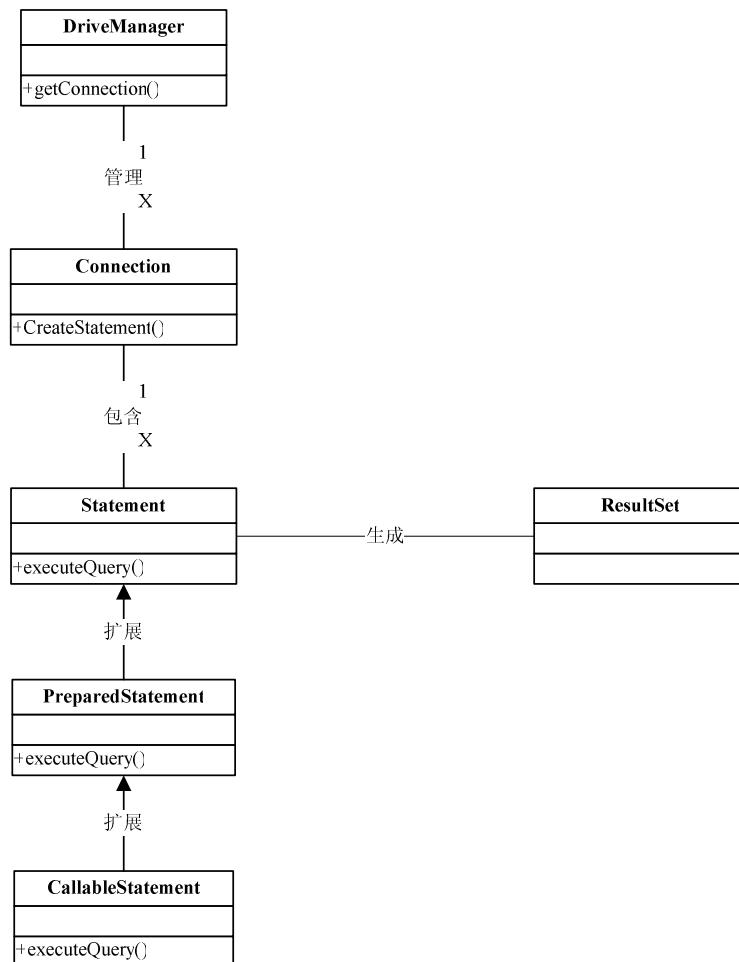


图 6-2: JDBC 接口定义

`java.sql.DriverManager`: 该接口主要定义了用来处理装载驱动程序并且为创建新的数据库连接提供支持。

`java.sql.Connection`: 该接口主要定义了实现对某一种指定数据库连接的功能。

`java.sql.Statement`: 该接口主要定义了在一个给定的连接中作为 SQL 语句执行声明的容器以实现对数据库的操作。它主要包含有如下的两种子类型。

`java.sql.PreparedStatement`: 该接口主要定义了用于执行带或不带 IN 参数的预编译 SQL 语句。

`java.sql.CallableStatement`: 该接口主要定义了用于执行数据库的存储过程的雕用。

`java.sql.ResultSet`: 该接口主要定义了用于执行对数据库的操作所返回的结果集。

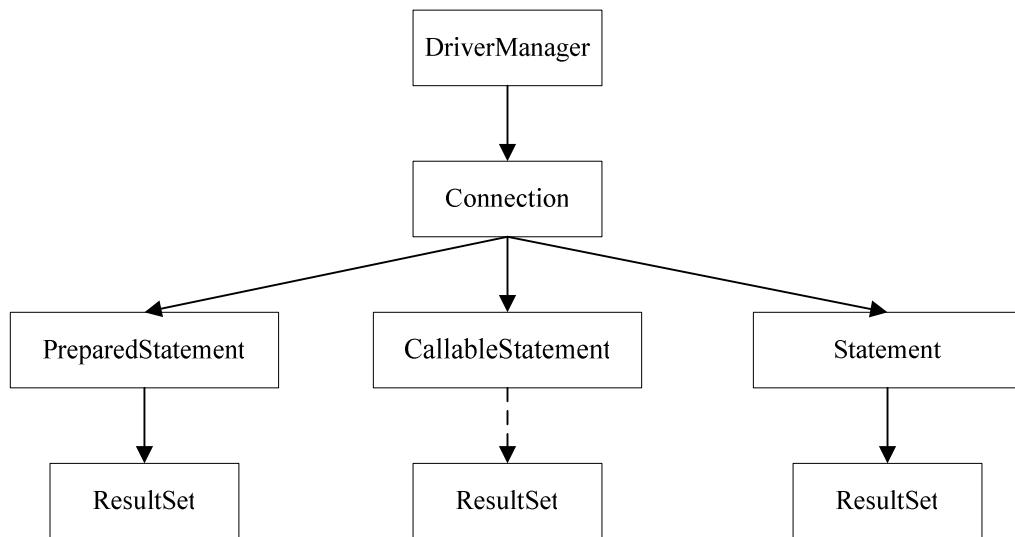


图 6-3: java.sql 包中主要类的关系

(2) 面向数据库厂商的 JDBC Drive API: 数据库厂商必须提供相应的驱动程序并实现 JDBC API 所要求的基本接口 (每个数据库系统厂商必须提供对 DriverManager、Connection、Statement、ResultSet 等接口的具体实现), 从而最终保证 Java 程序员通过 JDBC 实现对不同的数据库操作。

JDBC 的设计基于 X/Open SQL CLI (调用级接口) 这一模型。它通过定义出一组 API 对象和方法以用于同数据库进行交互。

在 Java 程序中要操作数据库, 一般应该通过如下几步 (利用 JDBC 访问数据库的编程步骤):

(1) 加载连接数据库的驱动程序

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

(2) 创建与数据源的连接

```
String url="jdbc:odbc:DatabaseDSN";
```

```
Connection con=DriverManager.getConnection(url,"Login","Password");
```

(3) 查询数据库: 创建 Statement 对象并执行 SQL 语句以返回一个 ResultSet 对象。

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

```
ResultSet rs=stmt.executeQuery("select * from DBTableName");
```

(4) 获得当前记录集中的某一记录的各个字段的值

```
String name=rs.getString("Name");
```

```
int age=rs.getInt("age");
```

```
float wage=rs.getFloat("wage");
```

(5) 关闭查询语句及与数据库的连接 (注意关闭的顺序先 rs 再 stmt 最后为 con)

```
rs.close();
```

```
stmt.close();
```

```
con.close();
```

## 6.7. 数据库应用的模型

(1) 两层结构 (C/S): 在此模型下, 客户端的程序直接与数据库服务器相连接并发送 SQL 语句 (但这时就需要在客户端安装被访问的数据库的 JDBC 驱动程序), DBMS 服务器向客户返回相应地结果, 客户程序负责对数据的格式化。



图 6-4: C/S 结构模型

主要的缺点：受数据库厂商的限制，用户更换数据库时需要改写客户程序；受数据库版本的限制，数据库厂商一旦升级数据库，使用该数据库的客户程序需要重新编译和发布；对数据库的操作与处理都是在客户程序中实现，使客户程序在编程与设计时较为复杂。

**(2) 三（或多）层结构 (B/S):** 在此模型下，主要在客户端的程序与数据库服务器之间增加了一个中间服务器（可以采用 C++ 或 Java 语言来编程实现），隔离客户端的程序与数据库服务器。客户端的程序（可以简单为通用的浏览器）与中间服务器进行通信，然后由中间服务器处理客户端程序的请求并管理与数据库服务器的连接。

## 6.8. 通过 JDBC 实现对数据库的访问

### 6.8.1. 编写访问数据库程序的步骤

#### (1) 引用必要的包

```
import java.sql.*; //它包含有操作数据库的各个类与接口
```

#### (2) 加载连接数据库的驱动程序类

为实现与特定的数据库相连接，JDBC 必须加载相应的驱动程序类。这通常可以采用 Class.forName() 方法显式地加载一个驱动程序类，由驱动程序负责向 DriverManager 登记注册并在与数据库相连接时，DriverManager 将使用此驱动程序。

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

注意：这条语句直接加载了 sun 公司提供的 JDBC-ODBC Bridge 驱动程序类。

#### (3) 创建与数据源的连接

```
String url="jdbc:odbc:DatabaseDSN";
```

```
Connection con=DriverManager.getConnection(url,"Login","Password");
```

注意：采用 DriverManager 类中的 getConnection() 方法实现与 url 所指定的数据源建立连接并返回一个 Connection 类的对象，以后对这个数据源的操作都是基于该 Connection 类对象；但对于 Access 等小型数据库，可以不用给出用户名与密码。

```
String url="jdbc:odbc:DatabaseDSN";
```

```
Connection con=DriverManager.getConnection(url);
```

```
System.out.println(con.getCatalog()); //取得数据库的完整路径及文件名
```

JDBC 借用了 url 语法来确定全球的数据库（数据库 URL 类似于通用的 URL），对由 url 所指定的数据源的表示格式为

jdbc:[ database locator]

jdbc---指出要使用 JDBC

subprotocol---定义驱动程序类型

database locator---提供网络数据库的位置和端口号(包括主机名、端口和数据库系统名等)

jdbc:odbc://host.domain.com:port/databasefile

主协议 jdbc 驱动程序类型为 odbc，它指明 JDBC 管理器如何访问数据库，该例指名为采用 JDBC-ODBC 桥接方式；其它为数据库的位置表示。

---

**例如：装载 mySQL JDBC 驱动程序**

---

```
.....  
Class.forName("org.gjt.mm.mysql.Driver ");  
String url  
="jdbc:mysql://localhost/softforum?user=soft&password=soft1234&useUnicode=true&chara  
cterEncoding=8859_1"  
//testDB 为你的数据库名  
Connection conn= DriverManager.getConnection(url);
```

---

**例如：装载 Oracle JDBC OCI 驱动程序（用 thin 模式）**

---

```
.....  
Class.forName("oracle.jdbc.driver.OracleDriver ");  
String url="jdbc:oracle:thin:@localhost:1521:orcl";  
//orcl 为你的数据库的 SID  
String user="scott";  
String password="tiger";  
Connection conn= DriverManager.getConnection(url,user,password);  
.....
```

---

注意：也可以通过 con.setCatalog("MyDatabase") 来加载数据库。

---

**例如：装载 DB2 驱动程序**

---

```
.....  
Class.forName("com.ibm.db2.jdbc.app.DB2Driver ")  
String url="jdbc:db2://localhost:5000/sample";  
//sample 为你的数据库名  
String user="admin";  
String password="";  
Connection conn= DriverManager.getConnection(url,user,password);  
.....
```

---

**例如：装载 MicroSoft SQLServer 驱动程序**

---

```
.....  
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver ");  
String url="jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=pubs";  
//pubs 为你的数据库的  
String user="sa";  
String password="";  
Connection conn= DriverManager.getConnection(url,user,password);  
.....
```

---

**(4) 查询数据库的一些结构信息**

这主要是获得数据库中的各个表，各个列及数据类型和存储过程等各方面的信息。根据这些信息，从而可以访问一个未知结构的数据库。这主要是通过 DatabaseMetaData 类的对象来实现并调用其中的方法来获得数据库的详细信息（即数据库的基本信息，数据库中的各

个表的情况，表中的各个列的信息及索引方面的信息)。

```
DatabaseMetaData dbms=con.getMetaData();
System.out.println("数据库的驱动程序为 "+dbms.getDriverName());
```

#### (5) 查询数据库中的数据:

在 JDBC 中查询数据库中的数据的执行方法可以分为三种类型，分别对应 Statement（用于执行不带参数的简单 SQL 语句字符串），PreparedStatement（预编译 SQL 语句）和 CallableStatement（主要用于执行存储过程）三个接口。

### 6.8.2. 实现对数据库的一般查询 Statement

**1、创建 Statement 对象**（要想执行一个 SQL 查询语句，必须首先创建出 Statement 对象，它封装代表要执行的 SQL 语句）并执行 SQL 语句以返回一个 ResultSet 对象，这可以通过 Connection 类中的 createStatement()方法来实现。

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

**2、执行一个 SQL 查询语句，以查询数据库中的数据。** Statement 接口提供了三种执行 SQL 语句的方法：executeQuery()、executeUpdate() 和 execute()。具体使用哪一个方法由 SQL 语句本身来决定。

方法 executeQuery 用于产生单个结果集的语句，例如 SELECT 语句等。

方法 executeUpdate 用于执行 INSERT、UPDATE 或 DELETE 语句以及 SQL DDL（数据定义语言）语句，例如 CREATE TABLE 和 DROP TABLE。INSERT、UPDATE 或 DELETE 语句的效果是修改表中零行或多行中的一列或多列。executeUpdate 的返回值是一个整数，指示受影响的行数（即更新计数）。对于 CREATE TABLE 或 DROP TABLE 等不操作行的语句，executeUpdate 的返回值总为零。

方法 execute 用于执行返回多个结果集、多个更新计数或二者组合的语句。一般不会需要该高级功能。

下面给出通过 Statement 类中的 executeQuery()方法来实现的代码段。executeQuery()方法的输入参数是一个标准的 SQL 查询语句，其返回值是一个 ResultSet 类的对象。

```
ResultSet rs=stmt.executeQuery ("select * from DBTableName");
```

#### 要点：

①JDBC 在编译时并不对将要执行的 SQL 查询语句作任何检查，只是将其作为一个 String 类对象，直到驱动程序执行 SQL 查询语句时才知道其是否正确。对于错误的 SQL 查询语句，在执行时将会产生 SQLException。

②一个 Statement 对象在同一时间只能打开一个结果集，对第二个结果集的打开隐含着对第一个结果集的关闭。

③如果想对多个结果集同时操作，必须创建出多个 Statement 对象，在每个 Statement 对象上执行 SQL 查询语句以获得相应的结果集。

④如果不需要同时处理多个结果集，则可以在一个 Statement 对象上顺序执行多个 SQL 查询语句，对获得的结果集进行顺序操作。

---

```
import java.sql.*;
```

```
public class ResultSetTest
{
    public static void main(String args[])
}
```

```
{      try

{

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

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

    Statement stmt=con.createStatement();

    ResultSet rs1=stmt.executeQuery("select name from student");

    ResultSet rs2=stmt.executeQuery("select age from student");

    //此时 rs1 已经被关闭

    while(rs2.next())

    {

        System.out.println(rs2.getObject(1));

    }

    rs2.close();

    stmt.close();

    con.close();

}

catch(Exception e)

{

    System.out.println(e);

}

}
```

---

注意：

此时显示出的将是姓名还是年龄？（将显示的是 rs2 的结果集的内容，即学生的年龄，因为采用 JDBC-ODBC 方式的驱动程序时，并且是采用同一个 Statement 对象，它只会保留最新的结果集，rs1 中的内容将会被新的结果集所取代）。

**3、关闭 Statement 对象：每一个 Statement 对象在使用完毕后，都应该关闭。**

```
stmt.close();
```

### 6.8.3. 预编译方式执行 SQL 语句 PreparedStatement

由于 Statement 对象在每次执行 SQL 语句时都将该语句传给数据库，如果需要多次执行同一条 SQL 语句时，这样将导致执行效率特别低，此时可以采用 PreparedStatement 对象来封装 SQL 语句。如果数据库支持预编译，它可以将 SQL 语句传给数据库作预编译，以后每次执行该 SQL 语句时，可以提高访问速度；但如果数据库不支持预编译，将在语句执行时才传给数据库，其效果类同于 Statement 对象。

另外 PreparedStatement 对象的 SQL 语句还可以接收参数，可以用不同的输入参数来多次执行编译过的语句，较 Statement 灵活方便（详见后文介绍）。

**1、创建 PreparedStatement 对象：**从一个 Connection 对象上可以创建一个 PreparedStatement 对象，在创建时可以给出预编译的 SQL 语句。

```
PreparedStatement pstmt=con.prepareStatement("select * from DBTableName");
```

**2、执行 SQL 语句：**可以调用 executeQuery() 来实现，但与 Statement 方式不同的是，它没有参数，因为在创建 PreparedStatement 对象时已经给出了要执行的 SQL 语句，系统并进行了预编译。

```
ResultSet rs=pstmt.executeQuery(); // 该条语句可以被多次执行
```

**3、关闭 PreparedStatement**

```
pstmt.close(); //其实是调用了父类 Statement 类中的 close()方法
```

### 6.8.4. 执行存储过程 CallableStatement

CallableStatement 类是 PreparedStatement 类的子类，因此可以使用在 PreparedStatement 类及 Statement 类中的方法，主要用于执行存储过程。

**1、创建 CallableStatement 对象：**使用 Connection 类中的 prepareCall 方法可以创建一个 CallableStatement 对象，其参数是一个 String 对象，一般格式为：

不带输入参数的存储过程 “{call 存储过程名()}”。

带输入参数的存储过程 “{call 存储过程名(?, ?)}”

带输入参数并有返回结果参数的存储过程 “{?=call 存储过程名(?, ?, ...)}”

```
CallableStatement cstmt=con.prepareCall("{call Query1()}");
```

**2、执行存储过程：**可以调用 executeQuery() 方法来实现。

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

**3、关闭 CallableStatement**

```
cstmt.close(); //其实是调用了父类 Statement 类中的 close()方法
```

检索记录集以获得当前记录集中的某一记录的各个字段的值

### 6.8.5. ResultSet 对象

① 执行完毕 SQL 语句后，将返回一个 ResultSet 类的对象，它包含所有的查询结果。但对 ResultSet 类的对象方式依赖于光标（Cursor）的类型，而对每一行中的各个列，可以

按任何顺序进行处理（当然，如果按从左到右的顺序对各列进行处理可以获得较高的执行效率）；

ResultSet 类中的 Course 方式主要有：

ResultSet.TYPE\_FORWARD\_ONLY（为缺省设置）：光标只能前进不能后退，也就是只能从第一个一直移动到最后一个。

ResultSet.TYPE\_SCROLL\_SENSITIVE：允许光标前进或后退并感应到其它 ResultSet 的光标的移动情形。

ResultSet.TYPE\_SCROLL\_INSENSITIVE：允许光标前进或后退并不能感应到其它 ResultSet 的光标的移动情形。

ResultSet 类中的数据是否允许修改主要有：

ResultSet.CONCUR\_READ\_ONLY（为缺省设置）：表示数据只能只读，不能更改。

ResultSet.CONCUR\_UPDATABLE：表示数据允许被修改。

可以在创建 Statement 或 PreparedStatement 对象时指定 ResultSet 的这两个特性。

Statement

```
stmt=con.createStatement(ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
```

或

```
PreparedStatement pstmt=con.PrepareStatement("insert into bookTable values (?,?,?,?)",ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_UPDATABLE);
```

② ResultSet 类的对象维持一个指向当前行的指针，利用 ResultSet 类的 next()方法可以移动到下一行（在 JDBC 中，Java 程序一次只能看到一行数据），如果 next()的返回值为 false，则说明已到记录集的尾部。另外 JDBC 也没有类似 ODBC 的书签功能的方法。

③ 利用 ResultSet 类的 getXXX()方法可以获得某一列的结果，其中 XXX 代表 JDBC 中的 Java 数据类型，如 getInt()、getString()、getDate()等。访问时需要指定要检索的列（可以采用 int 值作为列号（从 1 开始计数）或指定列（字段）名方式，但字段名不区别字母的大小写）。

```
while(rs.next())
{
    String name=rs.getString("Name"); //采用“列名”的方式访问数据

    int age=rs.getInt("age");

    float wage=rs.getFloat("wage");

    String homeAddress=rs.getString(4); //采用“列号”的方式访问数据
}
```

## 6.8.6. 数据转换

利用 ResultSet 类的 getXXX()方法可以实现将 ResultSet 中的 SQL 数据类型转换为它所返回的 Java 数据类型。

## 6.8.7. 对应 JAVA 与 SQL 类型

因为 Java 与 SQL 服务的目标不同，被开发在不同的环境中，所以二者之间不存在一一对应的类型也是可以理解的。这意味着，需要在 Java 与 SQL 类型之间建立起对映来。JDBC

定义了 Java 与 SQL 类型之间的对映。

表 6-1 SQL 类型到 Java 类型所定义的对映

JDBC 类型	Java 类型
CHAR	String
VARCHAR	String
LONGVSRCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
ARRAY	Array
BINARY	byte[]
BIGINT	long,Long
BIT	boolean,Boolean
DISTINCT	基础类型的对象
DOUBLE	double,Double
FLOAT	double,Double
INTEGER	int,Integer
JAVA_OBJECT	基础 Java 类
LONGVARBINARY	byte[]
REAL	float,Float
REF	Ref
SMALLINT	short,Integer
BLOB	Blob
BOOLEAN	boolean,Boolean
CLOB	Clob
DATALINK	java.net.URL
DATE	java.sql.Date
STRUCT	Struct,SQLData
TIME	java.sql.Time
TIMESTAMP	java.sql.timestamp
TINYINT	byte,Integer
VARBINARY	byte[]

我们可以看到，SQL 类型 CHAR、VARCHAR 与 LONGVARCHAR 都对映着 String 类型。所以，这些类型在 Java 中可以被看作是相同的。因为 LONGVARCHAR 类型可以构成很大的字符串，JDBC 也允许这些字符串作为 Java InputStreams 来检索。

我们看到，SQL NUMERIC 与 DECIMAL 类型都对映着 java.math.BigDecimal 类型。而且，SQL REAL 类型对映着 Java float 类型，SQL FLOAT 与 DOUBLE 类型都对映着 Java double 类型。

SQL BINARY、VARBINARY 与 LONGVARBINARY 类型都对映着 Java 中的 byte 数组。所以，与 character 类型的情况一样，它们在 Java 中都可以看作是相同的类型。与 LONGVARBINARY 一样，LONGVARBINARY 也可以作为 Java InputStreams 来检索。

SQL DATE、TIME 与 TIMESTAMP 类型分别对映着 java.sql.Date、java.sql.Time 与 java.sql.Timestamp。这些类都是 java.util.Date 类的子类，用于对映 SQL 类型。

其他类型的对映基本上与我们期望的一样，如下表所示：

表 6-2 Java 类型和 JDBC 类型对比

Java 类型	JDBC 类型
Array	ARRAY
Blob	BLOB
boolean ,Boolean	BIT, BOOLEAN
byte	TINYINT
byte[]	BINARY,VARBINARY,LONGVARBINAR
Clob	CLOB
double, Double	DOUBLE
float, Float	REAL
int, Integer	INTEGER
java.math.BigDecimal	NUMERIC
java.net.URL	DATALINK
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
long, Long	BIGINT
Ref	REF
short	SMALLINT
String	CHAR,VARCHAR,LONGVARCHAR
Struct	STRUCT
Java class	JAVA_OBJECT

上表显示了从 Java 到 SQL 的对映基本上是第一张表中对映的反转。下表显示了 ResultSet 对象的 get<type>与 set<type>对象方法之间所允许的对映/变换。用于一个给定过程的建议类型用黑体字表示。

表 6-3ResultSet 对象 get/set 方法变换

	TINYINT	SIMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINAR	DATE	TIME	TIMESTAMP
getAsciiStream()											✓	✓	✓	✓	✓	✓	✓	✓	
getBigDecimal()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getBinaryStream()														✓	✓	✓			
getBoolean()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getByte()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getBytes()														✓	✓	✓			
getDouble()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getDate()												✓	✓	✓			✓		✓
getFloat()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getInt()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getLong()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
getObject()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

getShort()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
getString()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
getTime()												✓	✓	✓						✓	✓
getTimestamp()												✓	✓	✓					✓		✓
getUnicodeStream()												✓	✓	✓	✓	✓	✓	✓			

## 6.8.8. NULL 结果值

要确定给定结果值是否是 JDBC NULL，必须先读取该列，然后使用 `ResultSet.wasNull` 方法检查该次读取是否返回 JDBC NULL。

当使用 `ResultSet.getXXX` 方法读取 JDBC NULL 时，方法 `wasNull` 将返回下列值之一：

(1) Java null 值

对于返回 Java 对象的 `getXXX` 方法（例如 `getString`、`getBigDecimal`、`getBytes`、`getDate`、`getTime`、`getTimestamp`、`getAsciiStream`、`getUnicodeStream`、`getBinaryStream`、`getObject` 等）。

(2) 零值：对于 `getByte`、`getShort`、`getInt`、`getLong`、`getFloat` 和 `getDouble`。

(3) false 值：对于 `getBoolean`

## 6.8.9. 获得结果集中的结构信息

利用 `ResultSet` 类的 `getMetaData()` 方法来获得结果集中的一些结构信息（主要提供用来描述列的数量、列的名称、列的数据类型。利用 `ResultSetMetaData` 类中的方法）。

.....

```
ResultsetMetaData rsmd=rs.getMetaData();
rsmd.getColumnCount(); //返回结果集中的列数
rsmd.getColumnLabel(1); //返回第一列的列名（字段名）
```

例如：

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

```
ResultSet rs=stmt.executeQuery("select * from TableName");
```

```
for(int i=1; i<=rs.getMetaData().getColumnCount(); i++) //跟踪显示各个列的名称
```

```
{ System.out.print(rs.getColumnName(i)+"\t");
}
```

```
while(rs.next())
```

```
{ //跟踪显示各个列的值
```

```
for(int j=1; j<=rs.getMetaData().getColumnCount(); j++)
```

```
{      System.out.print(rs.getObject(j)+"\t");  
  
}  
  
}  
.....
```

## 6.9. 更新数据库

前面主要介绍如何实现对数据库的查询操作，但在许多应用中需要实现对数据库的更新，这主要涉及修改、插入和删除等（即 SQL 语句中的 Insert、Update、Delete、Create、Drop 等）。仍然通过创建 Statement 对象来实现，但不再调用 executeQuery()方法，而是使用 executeUpdate()方法。

要点：正确区分 Statement 类中的 executeQuery()、execute()和 executeUpdate()方法的用法：

- (1) executeQuery() 执行一般的 SQL 查询语句（即 SELECT 语句）并返回 Resultset 对象；
- (2) execute()可以执行各种 SQL 查询语句，并可能返回多个结果集（这一般主要发生在执行了返回多个结果集的存储过程时），此时可以采用 Resultset 类的 getResultSet()来获得当前的结果集；
- (3) executeUpdate()执行对数据库的更新的 SQL 语句或 DDL 语句。

### 6.9.1. 对表中的记录进行操作

对一个表中的记录可以进行修改、插入和删除等操作，分别对应 SQL 的 Update、Insert、Delete 操作； executeUpdate()方法的输入参数仍然为一个 String 对象（即所要执行的 SQL 语句），但输出参数不是 ResultSet 对象，而是一个整数（它代表操作所影响的记录行数）。

```
.....  
  
Statement stmt=con.createStatement();  
  
stmt.executeUpdate("Update bookTable set Title='Java2' where Author='zhang'");  
  
stmt.executeUpdate("Delete from bookTable where Author='zhang'");  
  
stmt.executeUpdate("Insert      into      bookTable(BookID,Author,Title)      values(1,'Li  
Ming','Java2')"); //未给出的列，其值为 NULL
```

程序实例：对数据库中的表进行更新操作并显示操作前后的结果

```
import java.sql.*;
```

```
public class DBUpdateSetTest  
  
{    public static void main(String args[])  
  
{        try
```

```
{  
  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
  
    Connection  
    con=DriverManager.getConnection("jdbc:odbc:studlist");  
  
    Statement stmt=con.createStatement();  
  
    ResultSet rs=stmt.executeQuery("select * from student");  
  
    System.out.println("Result before executeUpdate");  
  
    while(rs.next())  
  
    {  
  
        System.out.println(rs.getString("name"));  
  
        System.out.println(rs.getString("age"));  
  
    }  
  
    stmt.executeUpdate("Update student set name='Yang' where id=0");  
  
    stmt.executeUpdate("Delete from student where id=2");  
  
    stmt.executeUpdate("Insert      into      student(id,name,age,sex)  
values(2,'zhang',30,true)");  
  
    rs=stmt.executeQuery("select * from student");  
  
    System.out.println("Result After executeUpdate");  
  
    while(rs.next())  
  
    {  
  
        System.out.println(rs.getString("name"));  
  
        System.out.println(rs.getString("age"));  
  
    }
```

```
        rs.close();

        stmt.close();

        con.close();

    }

    catch(Exception e)

    {

        System.out.println(e);

    }

}

}

....
```

## 6.9.2. 创建和删除表

创建和删除一个表主要对应于 SQL 的 Create Table 和 Drop Table 语句。这可以通过 Statement 对象的 executeUpdate()方法来完成。

① 创建表

```
Statement stmt=con.createStatement();
stmt.executeUpdate("create table TableName(ID integer, Name VARCHAR(20), Age
integer");
stmt.executeUpdate("Insert into TableName(ID, Name, Age) values(1,'Yang
Ming',30);
```

② 删除表

```
Statement stmt=con.createStatement();
stmt.executeUpdate("Drop Table TableName");
```

## 6.9.3. 增加和删除表中的列

对一个表的列进行更新操作主要是使用 SQL 的 ALTER Table 语句。对列所进行的更新操作会影响到表中的所有的行。

① 增加表中的一列

```
Statement stmt=con.createStatement();
stmt.executeUpdate("Alter Table TableName add Column Address VarChar(50)");
stmt.executeUpdate("Update TableName set Address='Beijing,China' where ID=1");
```

② 删除表中的一列

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

```
stmt.executeUpdate("Alter Table TableName Drop Column Address");
stmt.executeQuery("Select * from TableName");
```

## 6.9.4. 利用 PreparedStatement 对象实现数据更新

同 SQL 查询语句一样，对数据更新语句时也可以在 PreparedStatement 对象上执行。使用 PreparedStatement 对象，只需传递一次 SQL 语句，可以多次执行它，并且可以利用数据库的预编译技术，提高执行效率。另外也可以接受参数。

```
PreparedStatement pstmt=con.prepareStatement("Update TableName set
Address='Beijing,China' where ID >1");
pstmt.executeUpdate();
```

## 6.10. 参数的输入与输出

要实现使用 SQL 语句的输入与输出参数，必须在 PreparedStatement 类的对象上进行操作；同时由于 CallableStatement 类是 PreparedStatement 类的子类，所以在 CallableStatement 对象上的操作也可以使用输入与输出参数；其主要的编程原理是在生成 CallableStatement 或 PreparedStatement 类的对象时，可以在 SQL 语句中指定输入或输出参数，在执行这个 SQL 语句之前，要对输入参数进行赋值。

### (1) 使用 PreparedStatement 类的对象

通过 prepareStatement 类的对象可以实现在查询语句与数据更新语句方面都可以设置输入参数。

具体的方法是在 SQL 语句中用“？”标明参数，在执行 SQL 语句之前，使用 setXXX 方法给参数赋值，然后使用 executeQuery() 或 executeUpdate() 来执行这个 SQL 语句。每次执行 SQL 语句之前，可以给参数重新赋值。

setXXX 方法用于给相应的输入参数进行赋值，其中 XXX 是 JDBC 的数据类型，如：Int、String 等。setXXX 方法有两个参数，第一个是要赋值的参数在 SQL 语句中的位置，SQL 语句中的第一个参数的位置为 1，第二个参数的位置为 2；setXXX 方法的第二个参数是要传递的值，如 100、“Peking”等，随 XXX 的不同而为不同的类型。

```
PreparedStatement pstmt=con.prepareStatement("Update TableName set Name=? where
ID=?");
```

```
pstmt.setString(1,"zhang Hua"); //设置第一个参数（Name）为 “zhang Hua”

for(int i=1;i<3;i++)

{ pstmt.setInt(2,i); //设置第二个参数（ID）为 1,2

pstmt.executeUpdate();

}
```

要点：最终实现 Update TableName set Name=zhang Hua where ID=1 与 Update TableName set Name=zhang Hua where ID=2 的效果。

### (2) 使用 CallableStatement 对象

如果要求调用数据库的存储过程，要使用 CallableStatement 对象。另外还有些存储过程

要求用户输入参数，这可以在生成 CallableStatement 对象的存储过程调用语句中设置输入参数。在执行这个存储过程之前使用 setXXX 方法给参数赋值，然后再执行这个存储过程。

```
CallableStatement cstmt=con.prepareCall("{call Query(?)}"); //Query 为存储过程名
```

```
cstmt.setString(1,"输入参数"); //为存储过程提供输入参数
```

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

### (3) 接收输出参数

某些存储过程可能会返回输出参数，这时在执行这个存储过程之前，必须使用 CallableStatement 的 registerOutParameter 方法首先登记输出参数，在 registerOutParameter 方法中要给出输出参数的相应位置以及输出参数的 SQL 数据类型。在执行完存储过程以后，必须使用 getXXX 方法来获得输出参数的值。并在 getXXX 方法中要指出获得哪一个输出参数（通过序号来指定）的值。

实例：存储过程 getTestData 有三个输入参数并返回一个输出参数，类型分别为 VARCHAR。在执行完毕后，分别使用 getString()方法来获得相应的值。

```
CallableStatement cstmt = con.prepareCall("{? = call getTestData (?, ?, ?)}");
```

```
cstmt.setString(1,Value); //设置输入参数
```

```
cstmt.setInt(2,Value);
```

```
cstmt.setFloat(3,Value);
```

```
cstmt.registerOutParameter(1,java.sql.Types.VARCHAR); //登记输出参数
```

```
ResultSet rs = cstmt.executeQuery(); //执行存储过程
```

```
rs.getString(1); //获得第一个字
```

段的值

```
String returnResult=cstmt.getString(1); //获得返回的输出参数的值
```

要点：由于 getXXX 方法不对数据类型作任何转换，在 registerOutParameter 方法中指明数据库将返回的 SQL 数据类型，在执行完存储过程以后必须采用相应匹配的 getXXX 方法来获得输出参数的值。

## 6.11. 事务处理

在数据库操作中，一项事务是指由一条或多条对数据库更新的 SQL 语句所组成的一个不可分割的工作单元。只有当事务中的所有操作都正常完成，整个事务才能被提交到数据库。如果一项操作没有完成，就必须撤销整个事务。

例如在银行转帐事务中，假定张三从自己帐号上把 1000 元钱转到李四的帐号上，相关的 SQL 语句如下：

```
Update account set money = money - 1000 where name = 'zhangsan';
```

```
Update account set money = money +1000 where name = 'lisi';
```

这两条 SQL 语句必须作为一个完整的事务来处理，也就是说，只有当两条 SQL 语句都成功执行，才能提交整个事务。只要有一条语句执行失败，整个事务必须撤销，否则会导致数据库中的数据不一致。例如，假定第一条语句执行成功了，第二条语句却执行失败（例如执行第二条语句的时候数据库服务器刚好关机），张三户头上的钱就少了 1000 元，李四户头上的钱却没有多出来，那么银行转帐的事务就混乱了。

我们可以利用 Connection 类中提供的 3 个控制事务的方法：

```
setAutoCommit(Boolean autoCommit); //设置是否提交事务  
commit(); //提交事务  
rollback(); //撤销事务
```

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

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

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

## 6.12. 批量处理 JDBC 语句提高处理速度

有时候 JDBC 运行得不够快，这可以使用数据库相关的存储过程。当然，作为存储过程的一个替代方案，可以试试使用 Statement 的批量处理特性以提高速度。

存储过程的最简单的形式就是包含一系列 SQL 语句的过程，将这些语句放在一起便于在同一个地方管理也可以提高速度。Statement 类可以包含一系列 SQL 语句，因此允许在同一个数据库事务执行所有的那些语句而不是执行对数据库的一系列调用。

使用批量处理功能涉及下面的两个方法：

addBatch(String) 方法

executeBatch 方法

果你正在使用 Statement 那么 addBatch 方法可以接受一个通常的 SQL 语句，或者如果你在使用 PreparedStatement，那么也可以什么都不向它增加。

executeBatch 方法执行那些 SQL 语句并返回一个 int 值的数组，这个数组包含每个语句影响的数据的行数。

注意：如果将一个 SELECT 语句或者其他返回一个 ResultSet 的 SQL 语句放入批量处理中就会导致一个 SQLException 异常。

关于 java.sql.Statement 的简单范例可以是：

.....

```
con = DriverManager.getConnection(url,"myLogin", "myPassword");

con.setAutoCommit(false);

stmt = con.createStatement();

stmt.addBatch("INSERT INTO student " + "VALUES(4,'Yang',20,True)");

stmt.addBatch("INSERT INTO student " + "VALUES(5,'li',20,True)");

stmt.addBatch("INSERT INTO student " + "VALUES(6,'zhang',20,True)");

stmt.addBatch("INSERT INTO student " + "VALUES(7,'wang',20,True)");

stmt.addBatch("INSERT INTO student " + "VALUES(8,'liu',20,True)");

stmt.addBatch("INSERT INTO student " + "VALUES(9,'chen',20,True)");

stmt.addBatch("INSERT INTO student " + "VALUES(10,'xiao',20,True)");

int [] updateCounts = stmt.executeBatch();

con.commit();

con.setAutoCommit(true);

.....
```

---

PreparedStatement 有些不同，它只能处理一部分 SQL 语法，但是可以有很多参数，因

此重写上面的范例的一部分就可以得到下面的结果：

// 注意这里没有删除语句

```
PreparedStatement stmt = conn.prepareStatement(  
    "INSERT INTO student VALUES(?, ?, ?, ?)"  
)  
  
User[ ] users = ...;  
  
for(int i=0; i  
  
stmt.setInt(1, users[i].getID());  
  
stmt.setString(2, users[i].getName());  
  
stmt.setInt(3, users[i].getAge());  
  
stmt.setBoolean(4, users[i].getSex());  
  
stmt.addBatch();  
  
}  
  
int[ ] counts = stmt.executeBatch();
```

如果你不知道你的语句要运行多少次，那么这是一个很好的处理 SQL 代码的方法。在不使用批量处理的情况下，如果添加 50 个用户，那么性能就有影响，如果某个人写了一个脚本添加一万个用户，程序可能变得很糟糕。添加批处理功能就可以帮助提高性能，而且在后面的那种情况下代码的可读性也会更好。

## 6.13. 处理中文编码问题

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

```
<%@ page contentType="text/html; charset=gb2312" language="java"%>
```

不同的数据库的 JDBC 驱动程序采用的默认编码有所不同，例如 MySql 数据库采用的是 ISO-8859-1 编码，Oracle 在安装的时候你可以选择它的编码，Ms Sql Server 数据库采用的是操作系统默认的字符编码。为了从数据库中读取数据并正确的显示在网页上，我们可以采取以下几种办法：

**方法一：在设定连接数据库的 URL 时，指定字符编码。**

例如针对 MySql 的连接 URL 我们可以写成：

```
jdbc:mysql://localhost:3306/yourDB?useUnicode=true&characterEnconding=GB2312
```

采用这种方式从数据库取出的数据，使用的字符编码为指定的 GB2312 编码方式，这样

它就可以直接显示在网页上。注意：对于不同的数据库，数据库的 URL 中字符编码的设置形式可能不一致。

**方法二：如果在设定数据库的 URL 时，没有设定字符编码，则首先应该知道 JDBC 驱动程序使用的默认字符编码，然后对从数据库中取出的数据进行字符编码转换。**

例如：

```
String name = rs.getString(1);
String password = rs.getString(2);
int age = rs.getInt(3);
name = new String(name.getBytes("ISO-8859-1"),"GB2312");
password = new String(password.getBytes("ISO-8859-1"),"GB2312");
```

**方法三：使用过滤器优美的解决中文字符编码的问题。**

前提条件，每个页面使用

```
<%@ page contentType="text/html; charset= GB2312" language="java"
import="java.sql.*" errorPage="" %>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
```

步骤 1：添加过滤器

在 TOMCAT 中找到这 2 个文件 RequestDumperFilter.java , SetCharacterEncodingFilter.java , 他们位于 D:\Tomcat5.0.28\webapps\jsp-examples\WEB-INF\classes\filtters, 加到你的工程文件里去，编译他们。

步骤 2：配置 WEB.XML

在 web.xml 里加入这一段

.....

```
<filter>
    <filter-name>Set Character Encoding</filter-name>
    <filter-class>filters.SetCharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>GB2312</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Set Character Encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
.....
```

步骤 3：修改 server.xml

在 server.xml 修改 2 个地方

```
<Connector port="8080"
           maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
           enableLookups="false" redirectPort="8443" acceptCount="100"
           debug="0" connectionTimeout="20000"
           disableUploadTimeout="true" URIEncoding='GBK' />
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
```

```
port="8009" minProcessors="5" maxProcessors="75"
enableLookups="true" redirectPort="8443"
acceptCount="10" debug="0" connectionTimeout="0"
useURIValidationHack="false" protocol="AJP/1.3"

protocolHandlerClassName="org.apache.jk.server.JkCoyoteHandler"
URIEncoding='GB2312' />
```

## 6.14. 通过 JDBC 访问数据库的 JSP 范例程序

以下是一个 JSP 访问数据的范例程序，名为 DbJsp.jsp。在这个程序中建立了和数据库的连接，向 books 表中加入了一条记录，然后将 books 中所有记录以表格的方式显示出来，最后删除追加的新记录。

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

<html>
<head>
    <title>DbJsp. jsp</title>
</head>
<body>
<%
    //以try开始
    try {
        Connection con;
        Statement stmt;
        ResultSet rs;
        //加载驱动程序，下面的代码为加载MS Sql Server驱动程序
        Driver driver = (Driver) Class.forName(
            "com.microsoft.jdbc.sqlserver.SQLServerDriver")
            .newInstance();
        DriverManager.registerDriver(driver);
        //用适当的驱动程序连接到数据库
        String dbUrl =
"jdbc:microsoft:sqlserver://127.0.0.1:1433;databaseName=BookDB";
        String dbUser = "sa";
        String dbPwd = "sa";
        //建立数据库连接
    }
}
```

```
con = java.sql.DriverManager
    .getConnection(dbUrl, dbUser, dbPwd);
//创建一个JDBC声明
stmt = con.createStatement();
//增加新记录
stmt
    .executeUpdate("INSERT INTO books (id, name, title, price) VALUES
('999', 'Tom', 'Tomcat Bible', 44.5)");
//查询记录
rs = stmt.executeQuery("SELECT id, name, title, price from books");
//输出查询结果
out.println("<table border=1 width=400>");
while (rs.next()) {
    String col1 = rs.getString(1);
    String col2 = rs.getString(2);
    String col3 = rs.getString(3);
    float col4 = rs.getFloat(4);
    //打印所显示的数据
    out.println("<tr><td>" + col1 + "</td><td>" + col2
+ "</td><td>" + col3 + "</td><td>" + col4
+ "</td></tr>");
}
out.println("</table>");

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

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

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

访问 <http://localhost:8080/bookstore0/DbJsp.jsp>, 输出结果如下图所示:



图 6-5

## 6.15. 在 bookstore 应用中通过 JDBC 访问数据库

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

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

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

```
public BookDB() throws Exception {  
    Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");  
}
```

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

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

当数据库访问结束后，应该依次关闭 ResultSet、PreparedStatement（或 Statement）和 Connection 对象，从而释放数据库连接占用的资源。在 BookDB.java 中定义了三个方法分别关闭这 3 种对象：

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

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

```
Connection con = null;
PreparedStatement prepStmt = null;
ResultSet rs = null;
try {
    con = getConnection();
    //访问数据库
} finally {
    closeResultSet(rs);
    closePrepStmt(prepStmt);
    closeConnection(con);
}
```

下面给出 BookDB.java 的源代码:

```
package mypack;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;

public class BookDB {

    private ArrayList books;

    private String dbUrl =
"jdbc:microsoft:sqlserver://127.0.0.1:1433;databaseName=BookDB";

    private String dbUser = "sa";

    private String dbPwd = "sa";

    public BookDB() throws Exception {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
    }

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

    public void closeConnection(Connection con) {
        try {
            if (con != null)
```

```
        con.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

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

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

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

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

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

```
        closeResultSet(rs);
        closePrepStmt(prepStmt);
        closeConnection(con);
    }
    return books.size();
}

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

        while (rs.next()) {

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

    Collections.sort(books);
    return books;
}

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

```
prepStmt.setString(1, bookId);
rs = prepStmt.executeQuery();

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

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

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

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

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

public static void main(String[] args) {
    BookDB b;
    try {
        b = new BookDB();
        System.out.println(b.getNumberOfBooks());
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

## 6.16. 数据源 DataSource

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

## 6.16.1. 数据源和连接池

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

## 6.16.2. 数据源和 JNDI 资源

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

可以简单的把 JNDI 理解成一种将对象和名字绑定的技术，对象工厂负责生产出对象，这些对象都和唯一的名字绑定。外部程序可以通过名字来获得某个对象的引用。

在 `javax.naming` 包中提供了 `Context` 接口，该接口提供了将对象和名字绑定，以及通过名字检查对象的方法，`Context` 中主要方法有两个：

`bind(Name name, Object obj)` 将名称绑定到对象。

`lookup(Name name)` 检索指定的对象。

外部应用程序范围对象工厂中的对象的过程如下图所示：

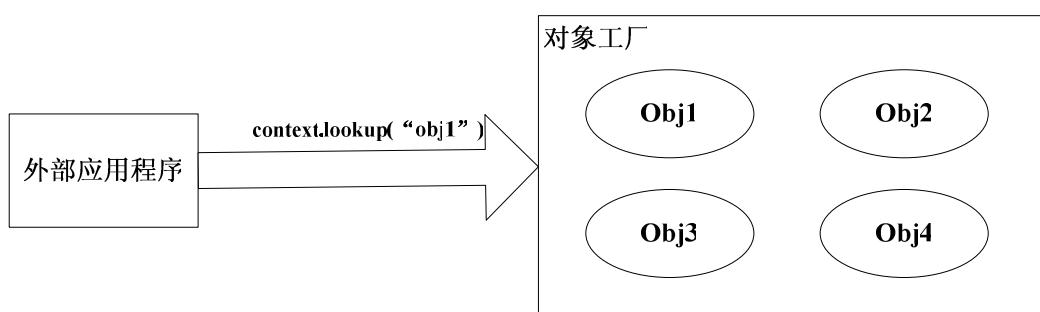


图 6-6：外部应用程序范围对象工厂中的对象

应用程序服务器把 `DataSource` 作为一种可配置的 JNDI 资源来处理。利用对象生成工厂来生成 `DataSource` 对象。各个应用程序服务器的工厂有所不同，Tomcat 工厂为 `org.apache.commons.dbcp.BasicDataSourceFactory`。假定我们需要配置两个 `DataSource`，一个名为 `jdbc/BookDB`，还有一个名为 `jdbc/BankDB`，`bookstore` 应用访问名为 `jdbc/BookDB` 的 `DataSource` 的过程如下图所示：

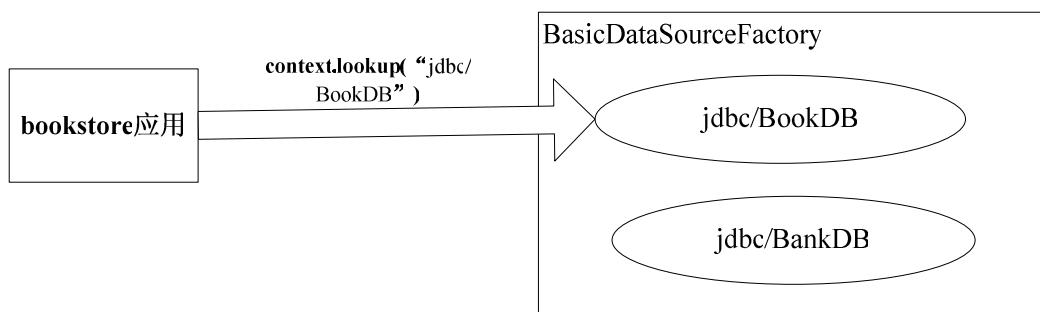


图 6-7：bookstore 应用程序访问 jdbc/BookDB 数据源

## 6.17. 在 Tomcat 中配置数据源

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

### 6.17.1. server.xml 中加入<Resource>元素

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

```
.....
<Context path="/bookstore1" docBase="D:\workspace\bookstore1\WebRoot" debug="0"
reloadable="true" >
    <Resource name="jdbc/BookDB" auth="Container" type="javax.sql.DataSource" />
    <ResourceParams name="jdbc/BookDB">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>
        <!-- Maximum number of dB connections in pool. Make sure you
            configure your mysqld max_connections large enough to handle
            all of your db connections. Set to 0 for no limit.
        -->
        <parameter>
            <name>maxActive</name>
            <value>100</value>
        </parameter>
        <!-- Maximum number of idle dB connections to retain in pool.
            Set to 0 for no limit.
        -->
        <parameter>
            <name>maxIdle</name>
            <value>30</value>
        </parameter>
        <!-- Maximum time to wait for a dB connection to become available
            in ms, in this example 10 seconds. An Exception is thrown if
            this timeout is exceeded. Set to -1 to wait indefinitely.
            Maximum time to wait for a dB connection to become available
            in ms, in this example 10 seconds. An Exception is thrown if
            this timeout is exceeded. Set to -1 to wait indefinitely.
        -->
        <parameter>
            <name>maxWait</name>
            <value>10000</value>
```

```

</parameter>

<!-- DB username and password for dB connections -->
<parameter>
    <name>username</name>
    <value>sa</value>
</parameter>
<parameter>
    <name>password</name>
    <value>sa</value>
</parameter>

<!-- Class name for JDBC driver -->
<parameter>
    <name>driverClassName</name>
    <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
</parameter>

<!-- The JDBC connection url for connecting to your DB.-->
<parameter>
    <name>url</name>
    <value>jdbc:microsoft:sqlserver://127.0.0.1:1433;databaseName=BookDB</value>
</parameter>
</ResourceParams>
</Context>
.....

```

在以上代码中，定义了<Resource>和<ResourceParams>元素，<Resource>的属性描述参见下表：

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

在<ResourceParam>元素中指定了配置 BookDB 数据源的参数，<ResourceParam>元素的参数见下表：

属性	描述
factory	指定生成 DataSource 的 factory 的类名
maxActive	指定数据库连接池中出于活动状态的数据库连接的最大数目，取值为 0 表示不受限制
maxIdle	指定数据库连接池中出于空闲状态的数据库连接的最大数目，取值为 0 表示不受限制
maxWait	指定数据库连接池中的数据库连接出于空闲状态的最长时间(以毫秒为单位)，超过这一时间将会抛出异常，取值为 -1，表示可以无限期

	等待
username	指定连接数据库的用户名
password	指定连接数据库的口令
driverClassName	指定连接数据库的 JDBC 驱动程序
url	指定连接数据库的 URL

## 6.17.2. 在应用程序的 web.xml 中加入<resource-ref>元素

如果 Web 应用访问了由 Servlet 容器管理的某个 JNDI Resource，必须在 web.xml（注意：Tomcat 的 conf 目录下也有个 web.xml 文件，这里要用到的是你的 WebRoot 下的 web.xml 文件）文件中声明对这个 JNDI Resource 的引用。表示资源引用的元素为<resource-ref>，以下是声明引用 jdbc/BookDB 数据源的代码：

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

<resource-ref>的属性描述见下表：

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

## 6.18. 程序中访问数据源

Javax.naming.Context 提供了查找 JDNI Resource 的接口，例如可以通过以下代码得到 jdbc/BookDB 数据源的引用：

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

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

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

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

## 6.18.1. 通过数据源访问数据库的 JSP 范例程序

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

```
<!--首先导入一些必要的 packages-->
<%@ page import="java.io.*"%>
<%@ page import="java.util.*"%>
<!--告诉编译器使用 SQL 包-->
<%@ page import="java.sql.*"%>
```

```
<%@ page import="javax.sql.*"%>
<%@ page import="javax.naming.*"%>

<!--设置中文输出--&gt;
&lt;%@ page contentType="text/html; charset=GB2312"%&gt;

&lt;html&gt;
    &lt;head&gt;
        &lt;title&gt;DbJsp.jsp&lt;/title&gt;
    &lt;/head&gt;
    &lt;body&gt;
        &lt;%
            //以 try 开始
            try {
                Connection con;
                Statement stmt;
                ResultSet rs;
                //建立数据库连接
                Context ctx = new InitialContext();
                DataSource ds = (DataSource) ctx
                    .lookup("java:comp/env/jdbc/BookDB");
                con = ds.getConnection();
                //创建一个 JDBC 声明
                stmt = con.createStatement();
                //增加新记录
                stmt
                    .executeUpdate("INSERT INTO books (id,name,title,price) VALUES
('999','Tom','Tomcat Bible',44.5)");
                //查询记录
                rs = stmt.executeQuery("SELECT id,name,title,price from books");
                //输出查询结果
                out.println("&lt;table border=1 width=400&gt;");
                while (rs.next()) {
                    String col1 = rs.getString(1);
                    String col2 = rs.getString(2);
                    String col3 = rs.getString(3);
                    float col4 = rs.getFloat(4);
                    //打印所显示的数据
                    out.println("&lt;tr&gt;&lt;td&gt;" + col1 + "&lt;/td&gt;&lt;td&gt;" + col2
+ "&lt;/td&gt;&lt;td&gt;" + col3 + "&lt;/td&gt;&lt;td&gt;" + col4
+ "&lt;/td&gt;&lt;/tr&gt;");
                }
                out.println("&lt;/table&gt;");
            }
        %&gt;</pre>
```

```

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

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

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

```

## 6.18.2. 在 bookstore 应用中通过数据源访问数据库

在 bookstore 应用中，上一个版本的对于每个需要访问数据库的客户请求，BookDB.java 都会首先建立与数据库的连接，然后再完成相关的事务。频繁的连接数据库，会影响 Web 服务器的工作效率，降低服务器相应客户请求的速度。

为了提高访问数据库的效率，在这里我们改进了 BookDB.java，通过 JNDI DataSource 来访问数据库。对于每一个需要访问数据库的客户请求，BookDB.java 不必直接建立和数据库的连接，只需要从 Servlet 容器提供的数据源的数据库连接池中取出一个空闲状态的连接。BookDB 通过数据源范围数据库的过程如下图所示：

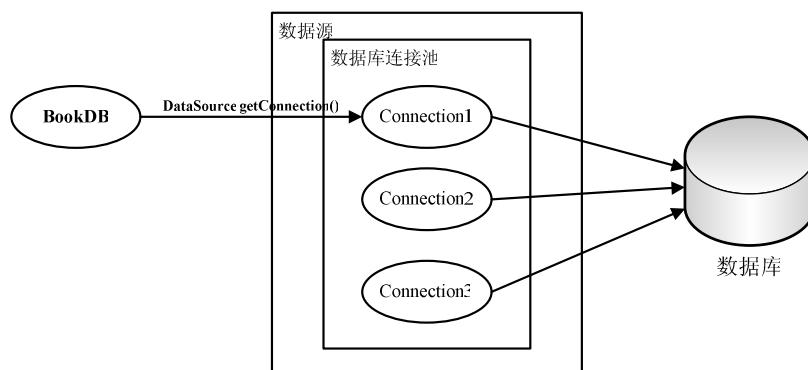


图 6-8: BookDB 通过数据源范围数据库

代码如下：

```

package mypack;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

```

```
import java.util.Iterator;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

public class BookDB {

    private ArrayList books;

    private DataSource ds = null;

    public BookDB() throws Exception {

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

    }

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

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

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

    public void closeResultSet(ResultSet rs) {
```

```
try {
    if (rs != null)
        rs.close();
} catch (Exception e) {
    e.printStackTrace();
}

}

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

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

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

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

```
rs = prepStmt.executeQuery();

while (rs.next()) {

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

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

Collections.sort(books);
return books;
}

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

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

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

```
        closeConnection(con);
    }
}

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

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

        if (rs.next()) {
            prepStmt.close();
            String updateStatement = "update books set saleamount = saleamount + ?
where id = ?";
            prepStmt = con.prepareStatement(updateStatement);
        }
    } finally {
        closeConnection(con);
    }
}
```

```
        prepStmt.setInt(1, quantity);
        prepStmt.setString(2, bookId);
        prepStmt.executeUpdate();
        prepStmt.close();
    }

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

}
```

## 6.19. 小结

本章介绍了利用 JDBC 访问数据库的一些 API 及具体操作过程。介绍了 Web 应用访问数据库的两种方法：一种通过 JDBC API 访问数据库，还有一种通过数据源访问数据库。这两种方法不同之处在于获得数据库连接的方式不一样。采用数据源，可以避免每次访问数据库都建立数据库连接，这样可以提高访问数据库的效率。在 java.sql 包中提供了访问数据库的 API，常用类包括：Connection（代表数据库连接）、Statement（执行静态 SQL 语句）、PreparedStatement（执行动态 SQL 语句）和 ResultSet（代表 select 查询语句得到的记录集合）。

本章介绍了两种访问数据库的方法（JDBC 和数据源），代码和结构非常相似，但是他们是有区别的：

1、构造方法的实现不一样；

```
public BookDB() throws Exception {
    Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
}
```

```
public BookDB() throws Exception {
```

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

```
}
```

2、获得连接 getConnection 方法实现不一样；

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

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

}

3、closeConnection 方法代码相同但是作用不一样；

利用 JDBC 方式访问数据库直接关闭数据库连接，而用数据源方式仅仅是把数据库连接对象返回到数据库，使连接对象又恢复到空闲状态。

# 第7章 Struts 与 MVC 设计模式

## 7.1. Struts 与 Java Web 应用

Java Web 应用的核心技术是 Java Server Page 和 Servlet。此外，开发一个完成的 Java Web 应用还设计以下概念和技术：

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

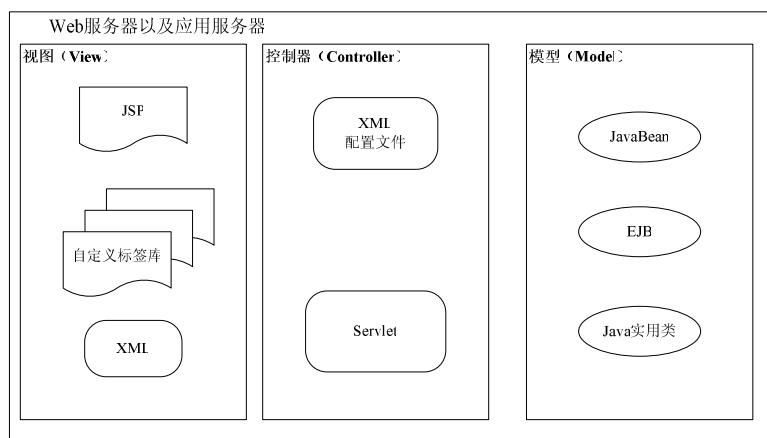


图 7-1: Java Web 应用的结构

### 7.1.1. Web 组件的三种关联

Web 应用程序如此强大的原因之一是他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，则还包含 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。

#### 7.1.1.1. 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，目标组件执行具体请求处理操作，并生成响应结果。下图显示了一个 Servlet 把请求转发给另一个 JSP 组件的过程。

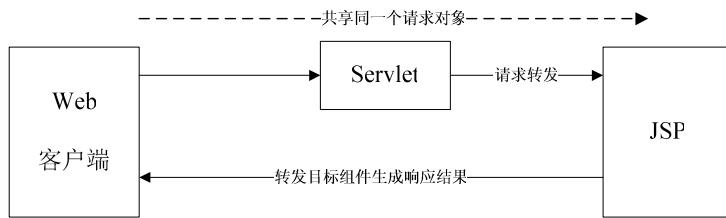


图 7-2: 请求转发

Sevlet 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>`转发请求，例如：

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

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

### 7.1.1.2. 请求重定向

请求重定向类似请求转发，但也有一些重要区别：

- Web 组件可以将请求重定向到任一 URL，而不仅仅是统一应用中的 URL。
- 重定向的资源组件和目标组件之间不共用一个 `HttpServletRequest` 对象，因此不能共享 `request` 范围内的共享数据。

下图显示了一个 Serlet 把请求重定向给另一个 JSP 组件的过程。

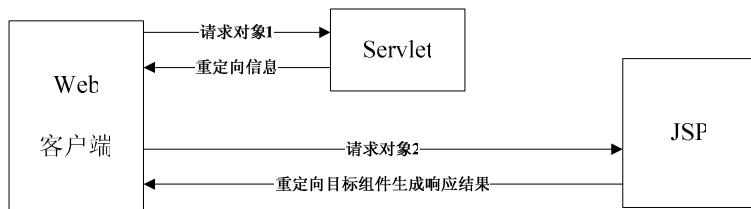


图 7-3: 请求重定向

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");
```

从上图中可以看出，`HttpServletResponse` 的 `sendRedirect()`方法向浏览器返回包含重定向信息，浏览器根据这一信息发送出一个新的 HTTP 请求，请求访问重定向目标组件。

### 7.1.1.3. 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效果。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，conger 组成单个页面。这种技术能够为应用程序

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，因此他们共享 request 范围内的共享数据。下图显示了一个 Servlet 包含另一个 JSP 组件的过程。

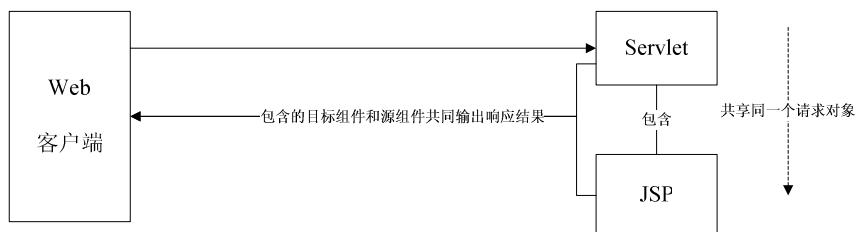


图 7-4: Web 组件的包含关系

Servlet 类使用 javax.servlet.RequestDispatcher.include()方法包含其他 Web 组件。例如，如果当前的 Servlet 组件包含了 3 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 Web 资源，例如：

```

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

```

### 7.1.2. MVC 概述

模型-视图-控制器（MVC）是 80 年代 Smalltalk-80 出现的一种软件设计模式，现在已经被广泛的使用。

开发模式上采用 Model-View-Controller（MVC）模式，MVC 模式将程序代码整理切割为三部份，Model 部分是业务与应用领域（Business domain）相关逻辑、管理状态之对象，Controller 部分接收来自 View 所输入的资料并与 Model 部分互动，是业务流程控制（Flow Control）之处，View 部分则负责展现资料、接收使用者输入资料。

- 模型（Model）模型是应用程序的主体部分。模型表示业务数据，或者业务逻辑。
- 视图（View）视图是应用程序中用户界面相关的部分，是用户看到并与之交互的界面。
- 控制器（controller）控制器工作就是根据用户的输入，控制用户界面数据显示和更新 model 对象状态。

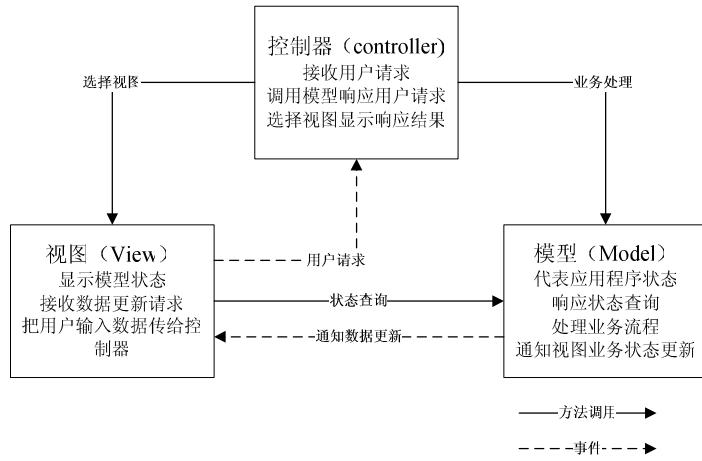


图 7-5: MVC 设计模式

### 7.1.2.1. MVC 的优点

可以为一个模型在运行时同时建立和使用多个视图。变化-传播机制可以确保所有相关的视图及时得到模型数据变化，从而使所有关联的视图和控制器做到行为同步。

视图与控制器的可接插性，允许更换视图和控制器对象，而且可以根据需求动态的打开或关闭、甚至在运行期间进行对象替换。

模型的可移植性。因为模型是独立于视图的，所以可以把一个模型独立地移植到新的平台工作。需要做的只是在新平台上对视图和控制器进行新的修改。

潜在的框架结构。可以基于此模型建立应用程序框架，不仅仅是用在设计界面的设计中。

### 7.1.2.2. MVC 的不足之处

增加了系统结构和实现的复杂性。对于简单的界面，严格遵循 MVC，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率。

视图与控制器间的过于紧密的连接。视图与控制器是相互分离，但确实联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。

视图对模型数据的低效率访问。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

目前，一般高级的界面工具或构造器不支持 MVC 模式。改造这些工具以适应 MVC 需要和建立分离的部件的代价是很高的，从而造成使用 MVC 的困难。

### 7.1.2.3. MVC 的应用范围

使用 MVC 需要精心的计划，由于它内部原理比较复杂，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。

但是对于开发存在大量用户界面上，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台阶。尽管在最初构建 MVC 框架时会花费一定工作量，但从长远的角度来看，它会大大提供后期软件开发效率。

### 7.1.3. JSP Model1 和 JSP Model2

SUN 在 JSP 出现早期制定了两种规范，称为 Model1 和 Model2。

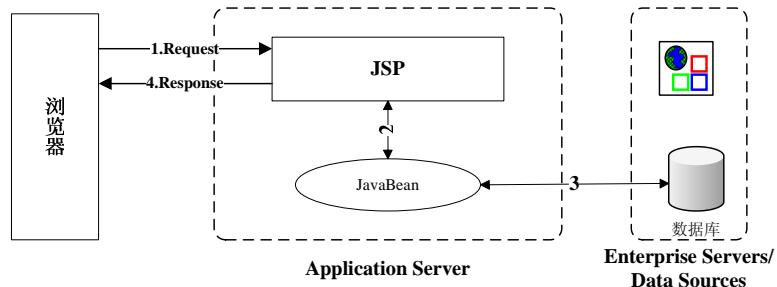


图 7-6: JSP Model 1

Model 1 它是以页面为中心的。适用于完成简单的应用程序。实现这个模式的应用程序有一系列的 Jsp 页面,在这些页面里用户程序运行从一个页面到另一个页面。因为它的简单容易。Model 1 应用的主要问题是难以维护，并且毫无灵活性可言。另外，由于开发人员已经同时被卷入到了页面开发和商业逻辑的编码中，这个架构模式在页面设计人员和 web 开发人员之间很难实现劳动分工。

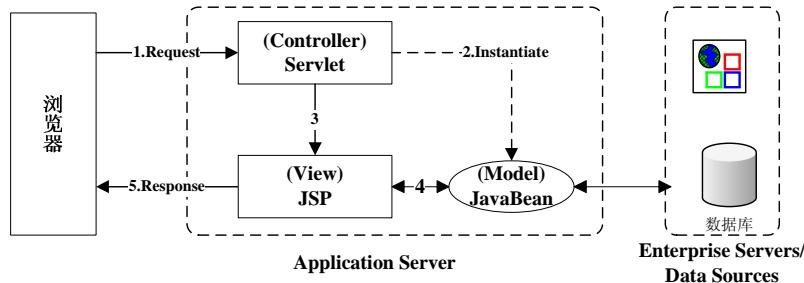


图 7-7: JSP Model 2

**JSP Model1 和 JSP Model2 的本质区别在于处理用户请求的位置不同。**

在 Model1 体系中，JSP 页面负责响应用户请求并将处理结果返回用户。JSP 既要负责具体业务流程控制，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 Model1 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。不加选择的使用 Model1，会导致 JSP 页内嵌入大量 Java 代码。尽管这对 Java 程序员不是多大的问题，但如果 JSP 页面由网页设计人员开发并进行维护的（大量的实际项目中情况确实是这样的），这个问题就会变得十分突出。从根本上讲，将导致角色定义不清和职责分配不明，会给项目管理带来很大的麻烦。

JSP Model2 体系结构是一种联合使用 JSP 和 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，根据用户请求选择使用的 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 Model2 的好处越大，越容易体现。

## 7.1.4. Struts 概述

Struts 实质上就是在 JSP Model2 的基础上实现一个 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。

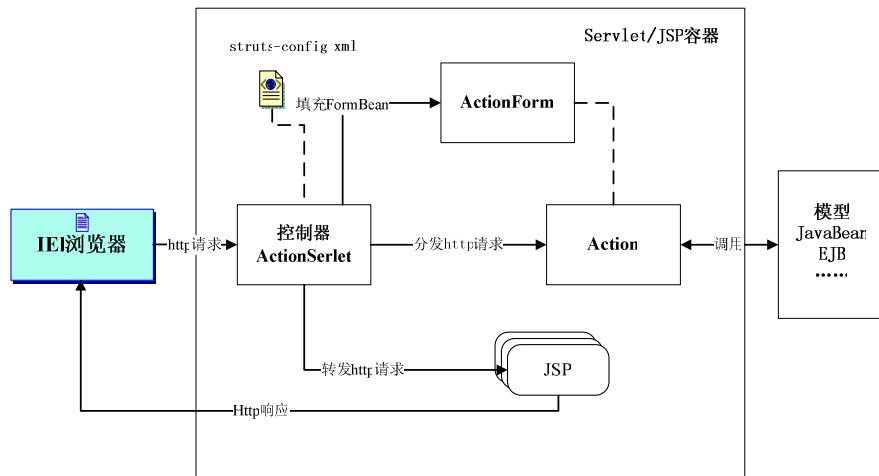


图 7-8: Struts 实现的 MVC 框架

### 7.1.4.1. Model

Struts 没有定义具体的 Model 层的实现，Model 层通常是和业务逻辑紧密相关的，还通常有持续化的要求，Struts 目前没有考虑到这一层，但是，不管在开源世界还是商业领域，都有一些都别优秀的工具可以为 Model 层次的开发提供便利，例如优秀的 O/R Mapping 开源框架 Hibernate。

### 7.1.4.2. View

通常，Web 应用的 UI 由以下文件组成：

- HTML
- JSP

而 JSP 中通常包含以下组件：

- 自定义标签
- DTO(Data Transfer Object 数据传输对象)

在 Struts 中，还包含了以下两种常用的组件：

- Struts ActionForms
- 资源绑定(java resource bundles)，例如将标签的显示内容，错误提示的内容通过配置文件来配置，这样可以为实现国际化提供基础。

由此可见，Struts 对于传统的 Web UI 所作的扩充就是 Struts ActionForms 和资源绑定，接下来对其进行进一步描述。

### 7.1.4.3. Controller

J2EE 的前端控制器(Front Controller)设计模式中利用一个前端控制器来接受所有客户请求，为应用提供一个中心控制点，在该控制点上，可以很方便地添加一些全局性的，如加密、国际化、日志等通用操作。Controller 的实现机制正是建立在前端控制器的设计模式基础上。

前面我们介绍过，Struts 的控制器拥有一些职责，其中最主要的是以下几个：

1. 接收客户请求。
2. 映射请求到指定的业务操作。
3. 获取业务操作的结果并以有效的方式提供给客户。
4. 根据业务操作的结果和当前的状态把不同的 UI 推向给客户。

在 Struts 框架中，控制器中不同的组件负责不同的控制职责，下图是 Struts 框架中关于控制器部分的一个组件图：

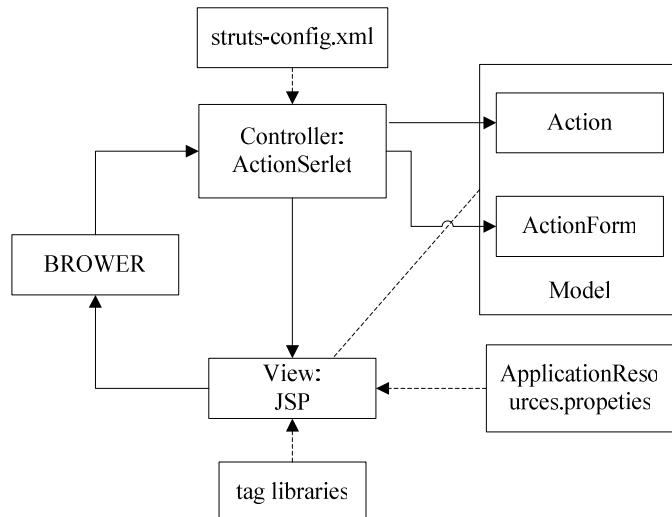


图 7-9：控制器组件

在上图中，很明显地可以看出，`ActionServlet` 处于核心位置，那么，我们就先来了解一下 `ActionServlet`。

`org.apache.struts.action.ActionServlet` 在 Struts 应用程序中扮演接收器的角色，所有客户端的请求在被其它类处理之前都得通过 `ActionServlet` 的控制。

当 `ActionServlet` 的实例接收到一个 HTTP 请求，不管是通过 `get` 方法或 `post` 方法，`ActionServlet` 的 `process()` 方法被调用并用以处理客户请求。`process()` 方法实现显示如下：

```

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

    RequestUtils.selectApplication( request, getServletContext( ) );
    getApplicationConfig( request ).getProcessor( ).process( request, response );
}
    
```

该方法的实现很简单，`RequestUtils.selectApplication( request, getServletContext( ) );`语句是用来根据用户访问的上下文路径来选择处理的应用，如果你只有一个 Struts 配置文件，就表示你只有一个 Struts 应用。`getApplicationConfig( request ).getProcessor( ).process( request, response );`语句用来获取一个处理器，并将客户请求提交给处理器处理。

#### 7.1.4.4. Struts 初始化处理流程

根据在 `web.xml` 中配置的初始化参数，Servlet 容器将决定在容器的第一次启动，或第一次客户请求 `ActionServlet` 的时机加载 `ActionServlet`，不管哪种方式加载，和其它 Servlet 一样，`ActionServlet` 的 `init()` 方法将被调用，开始初始化过程。让我们来看看

在初始化过程中将发生些什么，理解了这些，对于我们 debug 和扩展自己的应用更加得心应手。

1. 初始化框架的内部消息绑定，这些消息用来输出提示，警告，和错误信息到日志文件中。`org.apache.struts.action.ActionResources` 用来获取内部消息；
2. 加载 `web.xml` 中定义的不同参数，用以控制 `ActionServlet` 的不同行为，这些参数包括 `config, debug, detail, and convertNull`；
3. 加载并初始化 `web.xml` 中定义的 `servlet` 名称和 `servlet` 映射信息。通过初始化，框架的各种 DTD 被注册，DTD 用来在下一步校验配置文件的有效性；
4. 为默认应用加载并初始化 Struts 配置文件，配置文件即初始化参数 `config` 指定的文件。默认配置文件被解析，产生一个 `ApplicationConfig` 对象存于 `ServletContext` 中。可以通过关键字 `org.apache.struts.action.APPLICATION` 从 `ServletContext` 中获取 `ApplicationConfig`；
5. Struts 配置文件中指定的每一个消息资源都被加载，初始化，并存在 `ServletContext` 的合适区域(基于每个 `message-resources` 元素的 `key` 属性)，如果 `key` 属性没有设置，则为 `org.apache.struts.action.MESSAGE`；
6. Struts 配置文件中声明的每一个数据源被加载并且初始化，如果没有配置数据源，这一步跳过；
7. 加载并初始化 Struts 配置文件中指定的插件。每一个插件的 `init()` 方法被调用；当默认应用加载完成，`init()` 方法判断是否有应用模块需要加载，如果有，重复步骤 4—7 步成应用模块的加载。

#### 7.1.4.5. Struts 工作流程

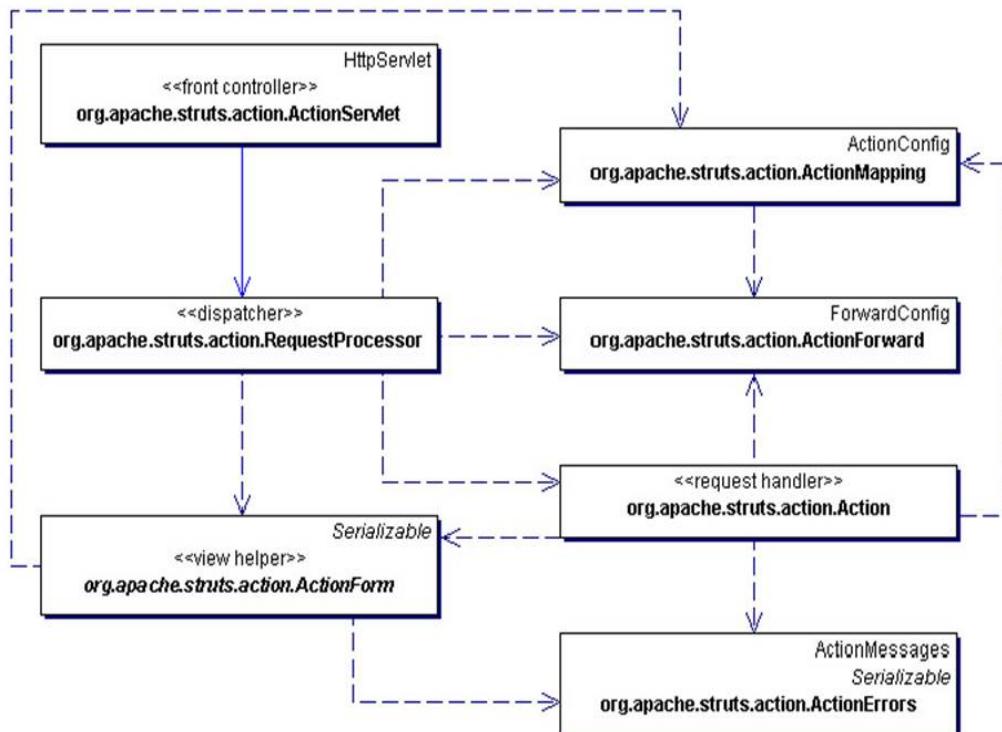


图 7-10: Struts 工作流程

上图是 Struts 的工作流程，前边我们提到，所有的请求都提交给 `ActionServlet` 来处理。

**ActionServlet** 是一个 FrontController，它是一个标准的 Servlet，它将 request 转发给 RequestProcessor 来处理。

**ActionMapping** 是 ActionConfig 的子类，实质上是对 struts-config.xml 的一个映射，从中可以取得所有的配置信息。

**RequestProcessor** 根据提交过来的 url，如\*.do，从 ActionMapping 中得到相应的 ActionForm 和 Action。然后将 request 的参数对应到 ActionForm 中，进行 form 验证。如果验证通过则调用 Action 的 execute() 方法来执行 Action，最终返回 ActionForward。

**ActionForward** 是对 mapping 中一个 forward 的包装，对应于一个 url。

**ActionForm** 使用了 ViewHelper 模式，是对 HTML 中 form 的一个封装。其中包含有 validate 方法，用于验证 form 数据的有效性。ActionForm 是一个符合 JavaBean 规范的类，所有的属性都应满足 get 和 set 对应。对于一些复杂的系统，还可以采用 DynaActionForm (Struts1.1) 来构造动态的 Form，即通过预制参数来生成 Form。这样可以更灵活的扩展程序。

**ActionErrors** 是对错误信息的包装，一旦在执行 action 或者 form.validate 中出现异常，即可产生一个 ActionError 并最终加入到 ActionErrors。在 Form 验证的过程中，如果有 Error 发生，则会将页面重新导向至输入页，并提示错误。

**Action** 是用于执行业务逻辑的 RequestHandler。每个 Action 都只建立一个 instance。Action 不是线程安全的，所以不应该在 Action 中访问特定资源。一般来说，应改使用 Business Delegate 模式来对 Business tier 进行访问以解除耦合。

Struts 提供了多种 Action 供选择使用。普通的 Action 只能通过调用 execute 执行一项任务，而 DispatchAction 可以根据配置参数执行，而不是仅进入 execute() 函数，这样可以执行多种任务。如 insert, update 等。LookupDispatchAction 可以根据提交表单按钮的名称来执行函数。

当 ActionServlet 接收到一个客户请求时，将执行如下流程：

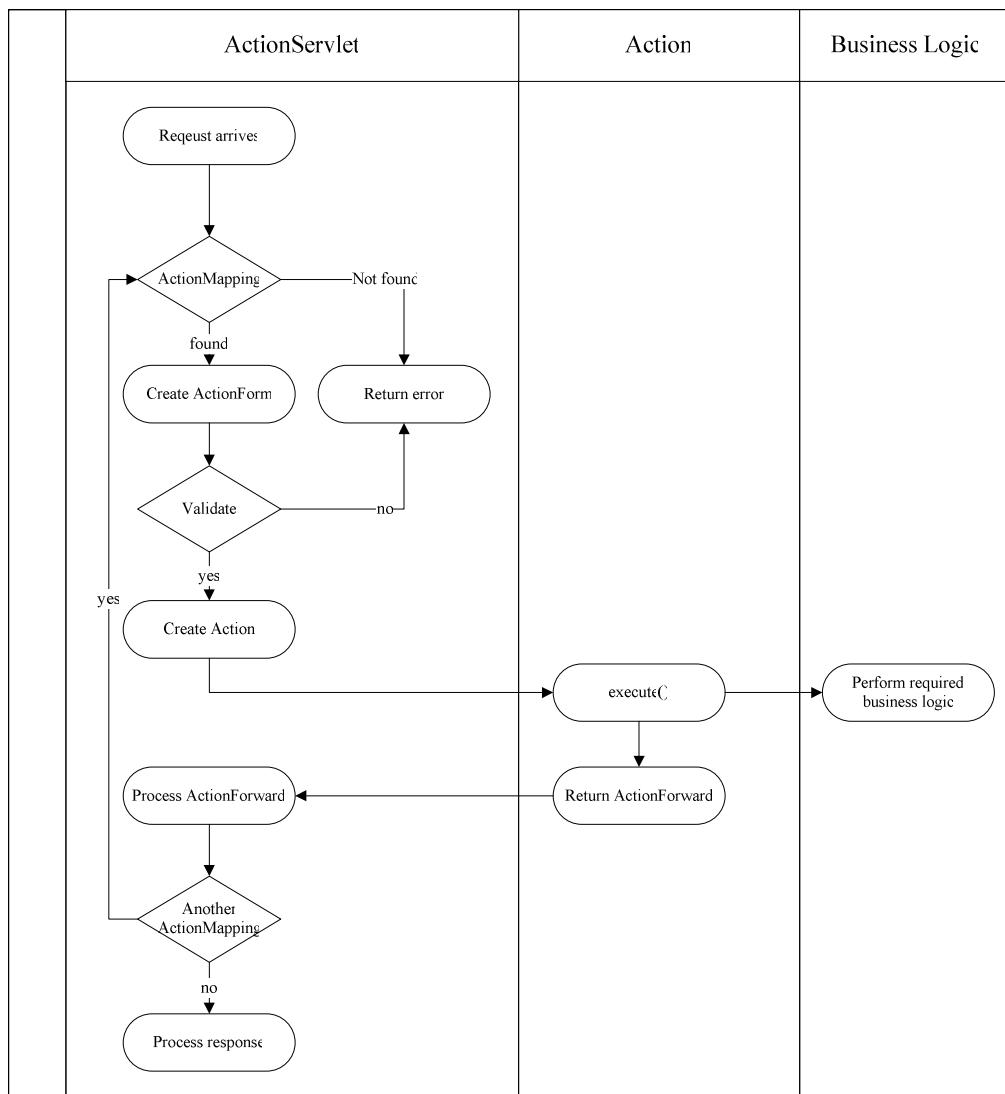


图 7-11: Struts 响应用户请求的工作流程

- 1) 检索和用户请求匹配的 ActionMapping 实例，如果不存在，就放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 或者一个不包含 ActionMessage 的 ActionErrors 对象，就表示表单验证成功。验证不成功则发送给用户提交表单的 JSP 组件，或者转向其他组件。这样 Action 对象不会被创建。
- 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 对象只想 JSP 组件。
- 7) ActionForward 对象指向 JSP 组件生成动态网页，返回给客户。

## 7.2. 从 helloapp 开始 Struts 应用

### 7.2.1. helloapp 的需求

helloapp 的需要非常简单，主要是体验 Struts 开发过程，其内容如下：

- 接收用户输入姓名<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，将返回错误信息，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” say hello。
- 为了演示模型组件的功能，本应用使用模型组件来保护用户输入的姓名。

### 7.2.2. 组建 Struts 框架

Helloapp 应用的各个模块组成：

**JavaBean 组件：**PersonBean，有一个 `userName` 属性，代表用户输入的名字。提供了 `get/set` 方法。这里可以预留 `save()` 方法，将数据保存到数据库中（JavaBean 组件可以作为 EJB 或者 Web 服务前端组件）。

**视图：**hello.jsp，提供用户界面，接收用户输入的姓名。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。

**控制器：**Action 类 HelloAction，主要任务有，一、进行业务逻辑验证，如果用户输入姓名为“Monster”，将返回错误消息；二、调用模型组件 PersonBean 的 `save()` 方法，保存用户输入的名字；三、决定将合适的视图组件返回给用户。

除了以上的模块，我们还要创建 Struts 的配置文件 struts-config.xml，利用 struts-config.xml 将这些组件组装起来。另外还要建立所有 Web 应用中的配置文件 web.xml。

### 7.2.3. 创建工程

我们在本教程中使用 Eclipse+MyEclipse 开发，采用 Struts1.1 规范，首先我们需要配置 MyEclipse 的插件，选择 Window—>Customize Perspective：

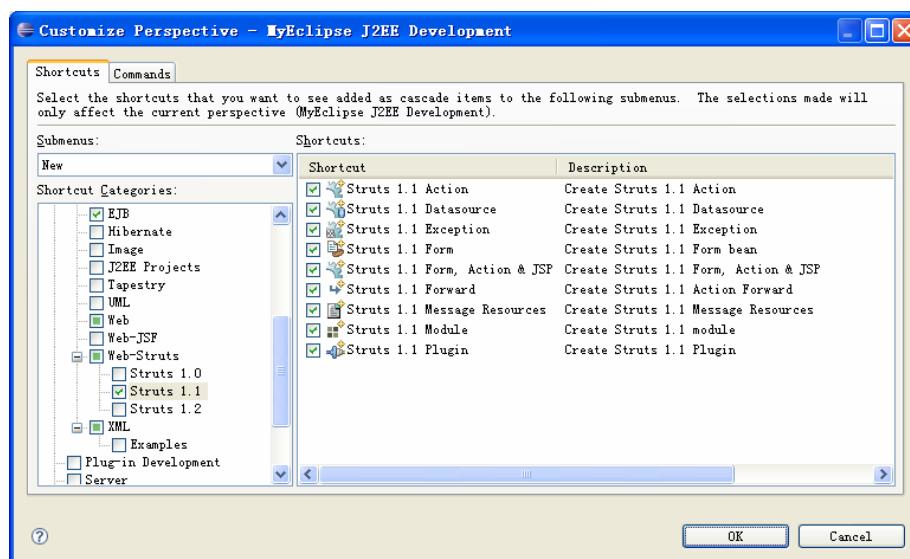


图 7-12：配置 MyEclipse 的 Struts 插件

利用 Eclipse 导航建立一个 Web Project 工程：

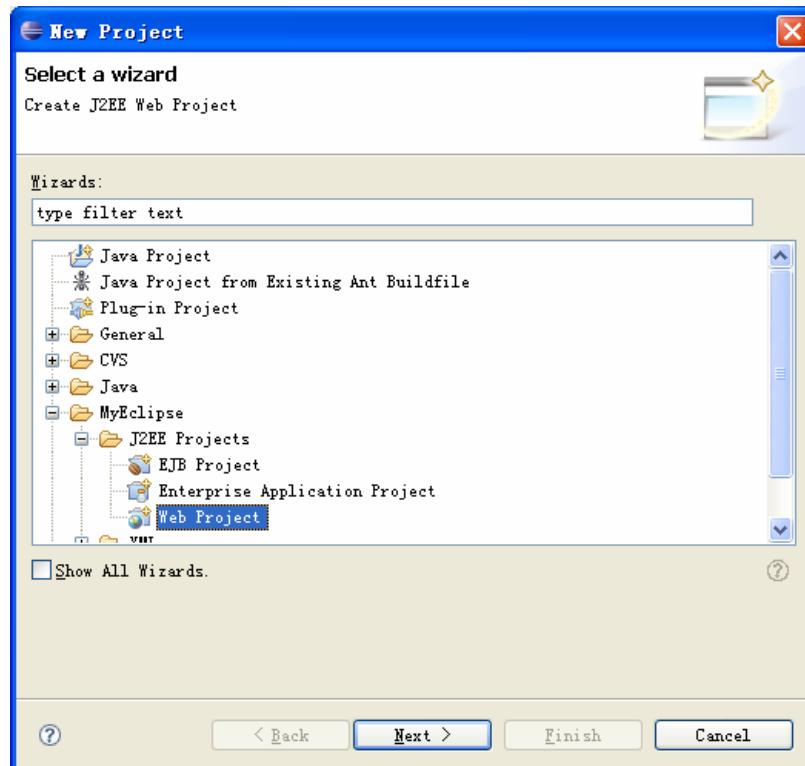


图 7-13: Project 导航

完成后，导入 Struts 所需要的组件：

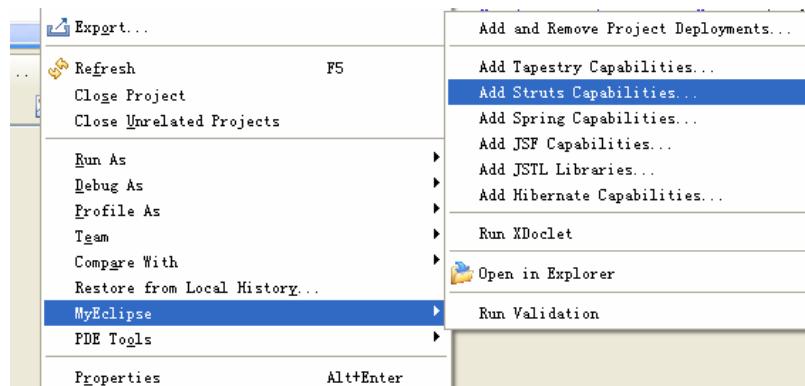


图 7-14: 选择加入 Struts 支持组件

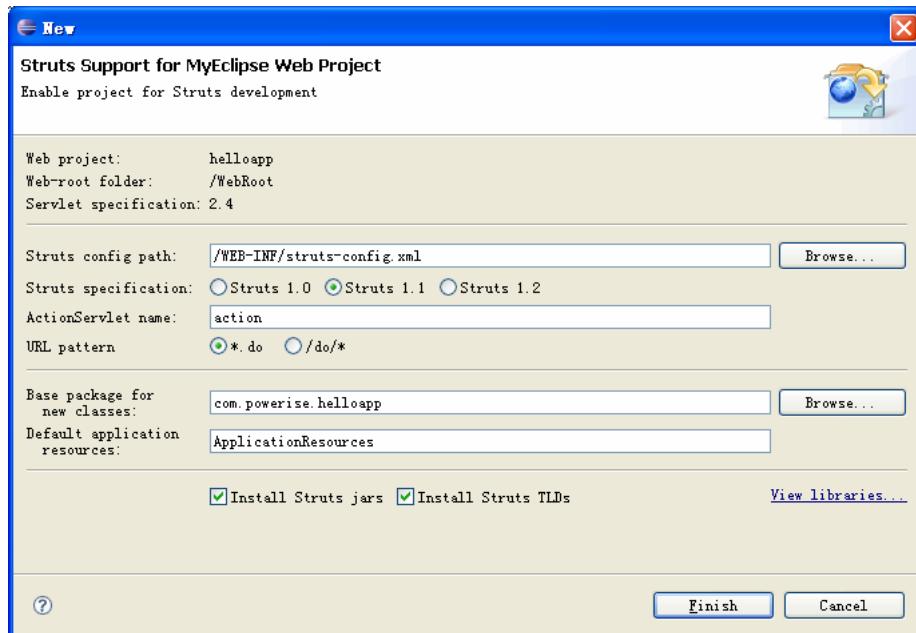


图 7-15: 定义配置信息

说明:

Struts config path: Struts 配置文件路径;

Struts specification: Struts 规范 (现在有 1.0, 1.1, 1.2 三种规范, 这里演示 1.1 规范);

ActionServlet name: 总控制器名字, Struts1.1 规范后用户可以自定义控制器, 只需要集成 ActionServlet。这里不修改这个参数。

URL pattern : URL 的样式, 例如 http://localhost:8080/helloapp.do, http://localhost:8080/do/helloapp, 用户也可以自己定义这个后缀。

Base package for new classes: 基本的包路径;

Default application resources: 资源文件;

Intall Struta jars: 是否需要 Struts 的 jar 包, 默认需要;

Install Struts TLDs: 是否需要 Struts 的标记库, 默认需要;

## 7.2.4. 创建视图组件

### 7.2.4.1. 创建 JSP 文件

helloapp.jsp 提供用户界面, 能够接收用户输入姓名。此外, 本 Web 应用的所有输出结果也都由 helloapp.jsp 显示。

在 WebRoot 上单击右键, 选择 New->JSP, 显示以下界面, 注意使用 Struts 模板:

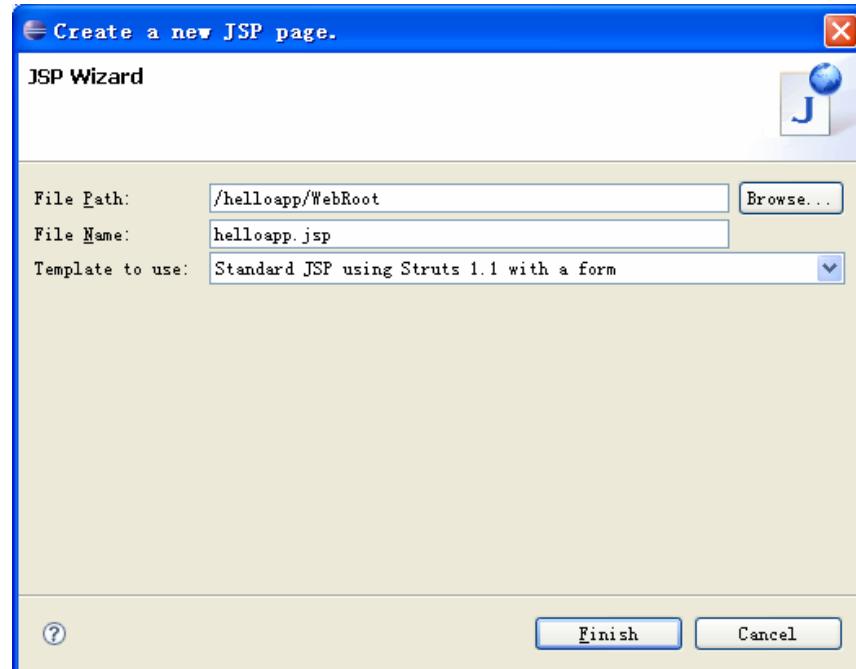


图 7-16: 建立 JSP 视图

代码如下所示：

```
<%@ page language="java" pageEncoding="UTF-8"%>

<%@ 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"%>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
<head>
    <html:base />

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

</head>

<body bgcolor="white">
    <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 />
</body>
</html:html>
```

以上基于 Struts 框架的 JSP 文件有以下特点：

- 没有任何 JSP 代码；
- 使用了许多 Struts 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由<bean:message>标签来生成。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。

hello.jsp 文件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"%>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested"%>
```

上面代码表明了该 JSP 文件使用了 Struts Bean、Html、logic、titles、nested 标签库，这是加载标签库的标准 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>、<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专门的 Resource Bundle，关于 Resource Bundle 在后面的章节将作详细讲解。

<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 中使用了来自 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，才会执行标签主体体中的内容：

```
<logic:present name="personbean" scope="request">
<h2>
    <bean:message key="hello.jsp.page.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 在特定的范田内不存在时，才会执行标签主体中的内容。

#### 7.2.4.2. 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 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 ApplicationResources.properties。

选择 New->File:

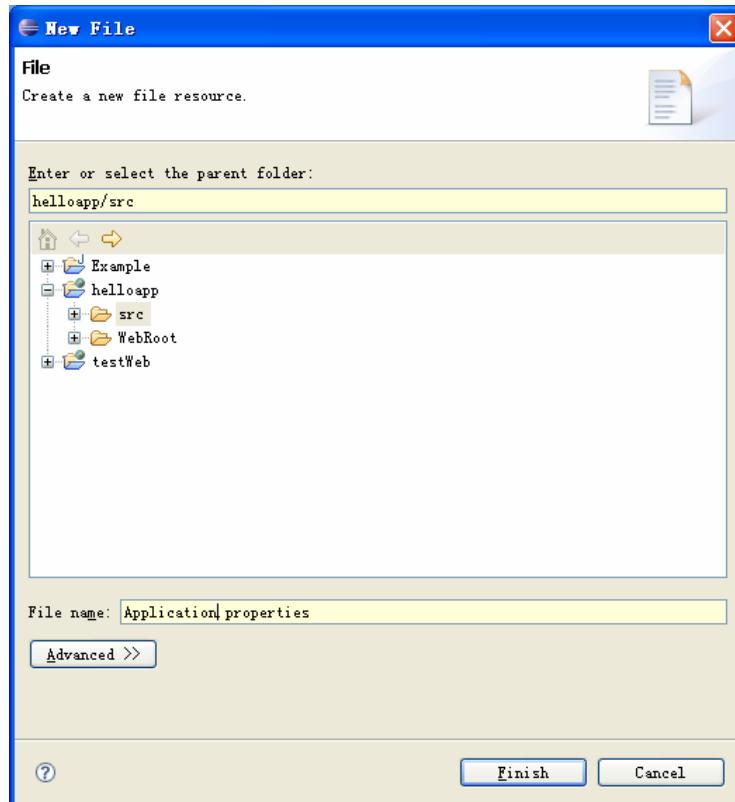


图 7-17：建立资源文件

ApplicationResources.properties 代码如下：

---

```
# Resources for parameter 'ApplicationResources'
```

---

```
# Project P/helloapp
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.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。对于以下代码：

```
<bean:message key="hello.jsp.title" />  
<bean:message>标签的 key 属性为"hello.jsp.title"，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>标签把“Hello - A first Struts program”输出至网页上。

#### 7.2.4.3. 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。HelloWorldForm.java 代码如下：

```
package com.powerise.struts.form;

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

/**
 * MyEclipse Struts Creation date: 06-19-2006
 *
 * XDoclet definition:
 *
 * @struts.form name="HelloForm"
 */
public class HelloWorldForm extends ActionForm

    /*
     * Generated fields
     */

    /**
     *

```

```
/*
private static final long serialVersionUID = 1L;
/** name property */
private String userName = null;

/*
 * Generated Methods
 */

/***
 * Method validate
 *
 * @param mapping
 * @param request
 * @return ActionErrors
 */
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;
}

/***
 * Method reset
 *
 * @param mapping
 * @param request
 */
public void reset(ActionMapping mapping, HttpServletRequest request) {
    // TODO Auto-generated method stub
    this.userName = null;
}

public String getUserName() {
    return userName;
}

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

---

}

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。

#### 7.2.4.4. 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。HelloWorldForm.java 中 validate() 方法负责完成这一任务：

```
/*
 * Method validate
 *
 * @param mapping
 * @param request
 * @return ActionErrors
 */

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 对象保存在 request 范围内，然后把请求转发到恰当的视图组件，视图组件<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。

**注意：在 Struts1.2 中将废弃 ActionErrors，统一采用 ActionMessage 类来表示正常或错误消息。**

对于业务逻辑验证由 Action 来负责处理。

## 7.2.5. 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，通常不需要扩张。Struts 框架提供了可以扩展的 Action 类，它用来处理特定的 Http 请求，下面为 HelloAction.java 代码：

```
package com.powerise.struts.action;

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 org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

import com.powerise.bean.PersonBean;
import com.powerise.struts.form.HelloWorldForm;

/**
 * MyEclipse Struts Creation date: 06-19-2006
 *
 * XDoclet definition:
 *
 * @struts.action-forward name="SayHello" path="/helloApp.jsp"
 */

public class HelloAction extends Action {
    /*
     * Generated Methods
     */

/**
 * Method execute
 *
 * @param mapping
 * @param form
 * @param request
 * @param response
 * @return ActionForward
 */

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        MessageResources messages = getResources(request);
        ActionMessages errors = new ActionMessages();
    }
}
```

```
HelloWorldForm HelloForm = (HelloWorldForm) form;
String userName = HelloForm.getUserName();
String badUserName = "Monster";
if (userName.equalsIgnoreCase(badUserName)) {
    errors.add("username", new ActionMessage(
        "hello.dont.talk.to.monster"));
    saveErrors(request, errors);
    return (new ActionForward(mapping.getInput()));
}
PersonBean pb = new PersonBean();
pb.setUserName(userName);
pb.saveToPersistentStore();

request.setAttribute("personbean", pb);
request.removeAttribute(mapping.getAttribute());
return (mapping.findForward("SayHello"));
}
```

### 7.2.5.1. Action 类的工作机制

所有的 Action 类都是继承自 `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){}
```

`execute()` 方法包含以下参数：

- `ActionMapping`: 包含了这个 Action 的的位置信息，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。

### 7.2.5.2. 访问封装在 MessageResources 中的本地化文本

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

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

在 Action 类中定义了 `getResources(HttpServletRequest request)` 方法，该方法放回当前默认的 `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 `MessageResource` 对象来访问文本内容。例如：在本例中要访问“hello.jsp.title”内容，可以调用 `MessageResources` 类的 `getMessage(String key)` 的方法：

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

### 7.2.5.3. 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();
HelloWorldForm HelloForm = (HelloWorldForm) form;
String userName = HelloForm.getUserName();
String badUserName = "Monster";
if (userName.equalsIgnoreCase(badUserName)) {
    errors.add("username", new ActionMessage(
        "hello.dont.talk.to.monster"));
    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 Action 基类中定义的 saveErrors() 方法，它负责把 ActionMessage 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给恰当的视图组件，视图组件通过<html:error>标签把 request 范围内的 ActionMessage 对象中包含的错误信息显示出来，提示用户修改。

### 7.2.5.4. 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();
```

说明：在本例中仅提供了 Action 类访问模型组件的简单例子。在实际应用中，Action 类会访问模型组件，完成更加复杂的功能：

- 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。

### 7.2.5.5. 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("personbean", pb);
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，所以可以把它删除。（如果不删除，将被虚拟机自动回收）

### 7.2.5.6. 把 HTTP 请求转发给合适的视图组件

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

### 7.2.6. 创建模型组件

本例中模型组件为 JavaBean: PersonBean, 以下是示例代码:

---

```
package com.powerise.bean;
```

```
public class PersonBean {  
    private String userName = null;  
  
    public String getUserName() {  
        return userName;  
    }  
  
    public void setUserName(String userName) {  
        this.userName = userName;  
    }  
    public void saveToPersistentStore(){  
    }  
}
```

---

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, 如数据库或者文件系统。

### 7.2.7. 配置文件

在创建 WEB 工程并引入 Struts 框架的时候, web 应用的配置文件及 Struts 框架的配置文件就已经被创建。

#### 7.2.7.1. web 应用的配置文件

web.xml:

---

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4"  
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee  
                           http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">  
    <servlet>  
        <servlet-name>action</servlet-name>  
        <servlet-class>  
            org.apache.struts.action.ActionServlet  
        </servlet-class>  
        <init-param>
```

```
<param-name>config</param-name>
<param-value>/WEB-INF.struts-config.xml</param-value>
</init-param>
<init-param>
    <param-name>debug</param-name>
    <param-value>3</param-value>
</init-param>
<init-param>
    <param-name>detail</param-name>
    <param-value>3</param-value>
</init-param>
<load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
<taglib>
    <taglib-uri>/WEB-INF/spring-beans.dtd</taglib-uri>
    <taglib-location>/WEB-INF/spring-beans.dtd</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-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>
<taglib>
    <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF/struts-tiles.tld</taglib-uri>
```

```
<taglib-location>/WEB-INF.struts-tiles.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF.struts-nested.tld</taglib-uri>
    <taglib-location>/WEB-INF.struts-nested.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF/rssutils.tld</taglib-uri>
    <taglib-location>/WEB-INF/rssutils.tld</taglib-location>
</taglib>
</web-app>
```

---

### 7.2.7.2. Struts 框架的配置文件

struts-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="HelloForm"
            type="com.powerise.struts.form.HelloWorldForm" />
    </form-beans>

    <action-mappings>
        <action path="/HelloWorld"
            type="com.powerise.struts.action.HelloAction" name="HelloForm"
            scope="request" validate="true" input="/hello.jsp">
            <forward name="SayHello" path="/hello.jsp" />
        </action>
    </action-mappings>

    <message-resources parameter="ApplicationResources" />
</struts-config>
```

---

## 7.3. 配置 Struts 组件

除了 Java 类和 JavaServer 页面之外，开发人员必须创建，或者修改，几个配置文件以使 Struts 应用能运转起来：

- web.xml

这是 Java Servlet 要求的 web 应用部署描述符。Servlet/JSP 容器使用这个文件来载入和配置你的应用。

- struts-config.xml

Struts 框架的部署描述符。它用来载入和配置 Struts 框架使用的各种组件。

- Application.properties

该文件为你的 Struts 应用提供资源。像 build.xml 文件一样，它不是严格要求的，但是大多数 Struts 应用都要用到。

尽管处理这些文件看起来也许不象是在进行“Java 开发”，但是正确的使用它们却是使你的 web 应用能拿得出手的基本要求。在这一章，我们会仔细讨论这些文件是如何工作的，以及它们能对你的应用的开发和部署做些什么。

除了每个 Struts 应用都需要的配置文件外，还有其它一些 Struts 应用也可能要用的东西。如果使用可选的组件，可能还需要另外的 XML 配置文件，比如 Tiles 框架和 Struts Validator。如果你想要把你的应用分成多个模块，每个模块也要有其自己的 Struts 配置和资源文件。

### 7.3.1. Web 应用部署描述符

框架的核心是 ActionServlet，Struts 把它当作是一个控制器。

虽然它也可以被子类化，但大多数开发人员都将它看成是一个黑盒。他们总是在 web 应用部署描述符（web.xml）中配置它，然后让它自己工作。

ActionServlet 可以接受许多初始化参数。大多数都有合理的缺省值，不需要重新设定。但有一些却必须设置，以使你的应用能正常工作。

在这一节，我们将考察一个典型的 Struts web 部署描述符，并且详细讨论 ActionServlet 的初始化参数。

Web 应用部署描述符的目的和格式在 Sun Servlet 规范 [Sun, JST] 中定义。基本上，它应该告诉 servlet 容器如何配置 servlet 和其它你的应用需要的高层次对象。

Struts 框架有两个组件需要从应用部署描述符中配置：ActionServlet 和标签库（可选）。虽然大多数 Struts 应用的确需要使用标签库，但它也不是严格要求的。使用 XSLT 或者 Velocity 的应用根本不需要配置标签库。

以下是一个示例的配置清单：

---

```
<!--A-->
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<!--B-->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
```

```
<param-value>3</param-value>
</init-param>
<init-param>
    <param-name>detail</param-name>
    <param-value>3</param-value>
</init-param>
<load-on-startup>0</load-on-startup>
</servlet>
<!--C-->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
<!--D-->
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
<!--E-->
<taglib>
    <taglib-uri>/WEB-INF/spring-beans.dtd</taglib-uri>
    <taglib-location>/WEB-INF/spring-beans.dtd</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-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>
<taglib>
    <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF/struts-tiles.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>
```

```
<taglib>
    <taglib-uri>/WEB-INF/struts-nested.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/WEB-INF/rssutils.tld</taglib-uri>
    <taglib-location>/WEB-INF/rssutils.tld</taglib-location>
</taglib>
</web-app>
```

代码中编号的地方对应下面的注释。

- A. 标识为 web 应用部署描述符—前两行将文件标识为一个 web 应用部署描述符。
- B. 配置 ActionServlet—这一段告诉容器装入 action 名称下的 ActionServlet。有四个参数传递给 ActionServlet: application, config, debug, 和 detail。(ActionServlet 也可以接受其它参数; 我们将再下一节涉及)。这一段的最后一个设定, <load-on-startup>, 给容器一个 actionservlet 的权重。如果设置为 2, 则允许其它 servlet 在需要时首先装入。这将在你子类化了 ActionServlet 时显得很重要, 以便你可以使用其它 servlet 先期装入的资源。对一个应用来说, 仅可以装入一个 ActionServlet 或者 ActionServlet 的子类。ActionServlet 设计为可以和应用中的其它组件共享资源。装入多个 ActionServlet 会造成冲突; 因为一个 ActionServlet 可能会改写另一个 ActionServlet 提交的资源。Struts 1.1 支持模块化应用, 但仍然仅允许装入一个 ActionServlet。
- C. 标识 Struts 请求—这一段告诉容器将匹配\*.do 格式的文件请求转发到 action servlet。这就是我们在 A 段中配置的 ActionServlet。不匹配这种格式的文件请求将不被 Struts 处理。比如对\*.html 或 \*.jsp 文件的请求通常由容器内建的服务来处理。
- D. 创建 welcome 文件—不幸的是, 在这里设置一个 index.do 文件将不会工作。容器希望 welcome 文件也是一个物理文件。
- E. 配置标签库— 这里我们配置应用中使用的标签库。3 个核心的 Struts 标签库—bean, html, 和 logic—将在大多数应用中使用。如果你的应用中使用了其它的标签库, 他们也在此进行配置。第一个元素, <taglib-uri>, 给出标签库的逻辑名称。这通常看起来像是一个文件路径, 但其实不是。JSP 在导入标签库时将引用这个 URI。第 2 个元素, <taglib-location>, 提供提供了一个上下文-相关的标签库描述符 (\*.tld) 路径。TLD 标识了库的实际类型(Java 类)。当需要这个库时, 容器会搜索标签库类文件的 classpath 。对 Struts 标签库来说, 容器将在 struts.jar 文件中找到这些包。

关于 web 应用部署描述符得更多细节, 请参见 Java Servlet 规范 [Sun, JST] 以及书籍 WebDevelopment with JavaServer 页面 [Fields]。

### 7.3.2. ActionServlet 的参数

Struts ActionServlet 可接受多个参数, 这些参数都总结在下表中:

参数	缺省值	说明	备注
config	/WEB-INF/strutsconfig.xml	包含配置信息的 XML 文件的上下文-相关路径	
config/\${pr}		使用标明的前缀的应用模块的	1. 1 以后

efix}		XML 配置文件的上下文-相关路径。在多模块应用中可以根据需要重复多次	
convertNull	false	一个参数，在组装表单时强制模拟 Struts 1.0 行为。如果设置为 true，数字的 Java 包装类类型（如 java.lang.Integer）将缺省为 null(而不是 0)	1. 1 以后
debug	0	调试的详细级别，控制针对这个 servlet 将记录多少信息。接受的值为 0 (off) 和 1(最不严格)直到 6 (最严格)。大多数 Struts 组件设置为级别 0 或者 2	
detail	0	用来处理应用配置文件的 Digester 的调试详细级别。.接受值为 0 (off) 和 1(最不严格)到 6 (最严格)	
validating	true	标识是否使用一个检验 XML 的解析器来处理配置 文件(强烈推荐)。	
application	无	应用资源束的名称，风格像是一个类名称。引用到位于名为 resources 的包 中 的 一 个 名 为 application.properties 的文件，这里使用 resources.application。这种情况下，资源可以是 classes 下的子目录 (或者 JAR 文件中的一个包)。	不推荐；使用 <messageresources> 元素的 parameter 属性进行配置
bufferSize	4096	处理文件上传时输入文件缓冲区的大小	不推荐；使用 <controller> 元素的 buffer-Size 属性配置
content	text/html	每个响应的缺省内容类型和字符编码； 可以被转发到的 servlet 或者 JSP 重写。	不推荐；使用 <controller> 元素的 contentType 属性配置
factory	org.apache.struts.util.propertyMessageResourcesFactory	MessageResourcesFactory 用来创建应用消息资源对象的类名	不推荐；使用 <messageresources> 元素的 factory 属性配置
formBean	org.apache.struts.action.ActionFormBean	ActionFormBean 实现使用的 Java 类名称	不推荐； 使用每一个 <form-bean> 元素的

			class-Name 属性配置
forward	org.apache.struts.action. ActionForward	ActionForward 实现使用的 Java 类名。	不推荐；使用每个<forward>元素的 className 属性配置
locale	true	如果设置为 true，并且存在一个用户会话，在用户会话中存储一个合适的 java.util.Locale 对象（在 Action.LOCALE_KEY 标识的标准关键字下）（如果还没有 Locale 对象存在的情况下）。	不推荐；使用<controller>元素的 locale 属性配置
mapping	org.apache.struts.action. ActionMapping	ActionMapping 实现使用的 Java 类名	不推荐；使用每个<action>元素的 className 属性配置，或者使用模块应用的<action-mappings>元素的 type 属性配置
max 文件 Size	250M	文件上传时可以接收的最大文件尺寸（Byte）。可以表示为 "K", "M", "G"。分别解释为 kilobytes, megabytes, 或者 gigabytes,	不推荐，使用<controller>元素的 maxFileSize 属性配置
multipartCl ass	org.apache.struts.uploa dDiskMultipartRequest Handler	MultiPartRequestHandler 实现类的全限定名称，用来处理文件上传。如果没有设置，禁止 Struts 多部分请求处理	
nocache	false	如果设置为 true，将在每个响应前加上 HTTP 头。这样可以使浏览器对我们产生的和转发的响应的缓存失效	不推荐；使用<controller>元素的 nocache 属性设置
null	True	如果设置为 true，那么如果使用了未知的消息关键字，应用资源将返回 null。否则，将返回一个包含不愉快信息的错误信息	不推荐；使用<message-resou rces> 元素的 null 属性配置
tempDir	提供给 web 应用作为 servlet 上下文 属性的 工作目录	处理文件上传时的工作目录	不推荐；使用<controller>元素的 tempDir 属性配置

### 7.3.3. Struts 配置

Struts 配置文件 (struts-config.xml) 用来装入多个关键的框架组件。这些对象一起构成了 Struts 配置。

Struts 配置和 Struts ActionServlet 一起工作，来创建应用的控制层。在这一节，我们将探索为什么需要 Struts 配置。下一节，我们讲看看 Struts 开发人员要如何来创建和维护配置。

**注：从 Struts 1.1 开始，一个应用可以分成多个模块模块。每个模块都有其自己的 Struts 配置。每个 Struts 应用至少有一个缺省，或成为“根”模块。如果你没有使用多模块，或者正使用 Struts 1.0，那么当我们说模块时，你可以将它想象成应用。**

我们在本章的末尾讨论 Struts 1.1 应用于多模块的配置。

Struts 配置是你的应用的真实蓝图。它知道表单中有什么字段。它知道哪里可以找到 JSP 文件。它也知道应用执行的每一个 Action，以及每个 action 需要的实际资源。

这看起来好像是把许多信息集中在了一个地方。实际上就是。但是通过将这些实现细节放在一起，许多开发人员会发现他们的应用更加易于创建和维护。

Struts 配置中的每个组件都是 Java 对象。ActionForm 对象知道字段和表单。

ActionForward 对象知道何处可以找到 JSP。ActionMapping 对象知道那个表单和转发用于每个应用能理解的命令。

一个非常简单的应用可以在一个实例化方法内创建所有这些信息对象，然后设置需要的缺省值。例如：

```
ActionForwards globals = new ActionForwards();
ActionForward logoff = new ActionForward();
logoff.setName("logoff");
logoff.setPath("/Logoff.do");
globals.addForward(logoff);
ActionForward logon = new ActionForward();
logoff.setName("logon");
logoff.setPath("/Logon.do");
globals.addForward(logon);
```

但是，实践中，初始化方法很快就会成为维护负担，并造成许多问题。

具有讽刺意味的是，像这样的类并不涉及到多少编程问题。它只是从存在的类中实例化对象。它几乎不需要位于 Java 代码中。

事实上，它也不。Java 语言可以通过名称创建一些给定的类。Java 也支持一些如可以决定一个类在运行时支持那些方法的反射特征。

**定义：反射 告诉我们 Java 类提供什么方法。自省 (Introspection) 帮助我们推论出这些方法哪些是在运行时用来配置 JavaBean 的属性。Java 工具和框架（如 Struts）使用反射和自省来自动化装入和配置 JavaBean 对象。这样就消除了哪些因为粗心易导致错误的编写和装入仅仅为了装入其他对象的简单对象时的任务。**

将这些特征结合在一起，其实并不需要一个 Java 类。你需要的是一个文档来描述如何实例化一个 Java 类使之成为一个全功能的对象。

当然，象 Struts 这样的框架并不是唯一具有这个问题的东西。Servlet 容器基于同一原因也需要同样的东西。开发人员不得不告诉容器应用中需要什么 servlet 以及其他一些对象。

不是编写一个 Java 类并插入到容器之中，Sun 的工程师却是选择了使用一个 XML 文档。

容器读入这个文档并使用它来实例化和配置应用需要的 servlet。

Struts 配置文件对 Struts 来说就像部署描述符对容器一样。Struts 控制器读入配置文件并使用它来创建和配置框架需要的那些对象。

每个编写过部署描述符（web.xml）文件的 web 开发人员都应该使用过 XML 元素来创建一个 Java 对象。例如，我们可以在 web.xml 中部署一个 servlet：

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
        org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>3</param-value>
    </init-param>
    <init-param>
        <param-name>detail</param-name>
        <param-value>3</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
</servlet>
```

而下面则是 Struts 配置 文件中部署一个 forward 对象的代码片段：

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

事实上， struts 配置并不是配置 应用而是部署 它。但大多数开发人员，还是很自然的将它这些对象视为是“配置，” 所以我们还是使用这个词汇。

### 7.3.3.1. 变更管理

按这种方式部署预配置的 Java 对象是个强大的特征。当然，强大的功能也带来强大的职责。因为 Struts 配置文件装入框架对象，所以它也对框架对象负责。

通过描述框架组件间如何交互，Struts 配置文件成为了一个管理应用变更的非常有效率的工具。实践中，文件要胜过一个简单的对象载入器，并被用作为动态的设计文档。

不难想象，可以用工具来读入 Struts 配置文件并用它来产生和创建一个 UML 图。有好多 Struts GUI 现在都可以帮助你维护这个 XML (参见下节：Struts 配置元素)。不久就会出现可视化的工具可以帮助你维护由 Struts 配置文件表达的架构设计。

### 7.3.3.2. 受保护的变更原则

Struts 配置文件帮助你可以以最小的努力对应用变更作快速的反应。如果一个对象需要初始化为另一个值， 你并不需要编辑，编译和部署一个 Java 类。许多配置细节都涉及到

表现层。团队中工作于该层的人员可能不都是 Java 工程师。使用 XML 文档可以使配置被页面设计员和项目管理者都能访问到。需要 Java 工程师来创建和修改应用的基对象，但配置这些对象却可以委派给其它人。实践中，我们常常将不经常的变更—基础 Java 类—从经常变更的事物—Java 对象在运行时如何部署中分离出来。这就是受保护的变更原则 (principle of Protected Variation [Larman])。

**定义:** *Protected Variation* 受保护的变更是一个设计原则，它鼓励使用一个稳定的接口来封装变更的可以预知点。数据-驱动设计，服务查询，解释器驱动设计，反射设计都是这种机制的不同实现。

受保护的变更可以让我们记住一个单一的变更点可以产生一个单一的维护点。通常从基对象（不常改变）中分离出实现细节（经常改变），我们就可以减少维护应用要做的努力。

### 7.3.4. Struts 配置元素

我们在前面讨论过，Struts 配置文件是一个用来部署 Java 对象的 XML 文档。配置中的每个元素对应一个 Java 对象。当你在 Struts 配置文件中插入一个元素，你就是告诉 Struts 控制器在应用初始化时要创建一个 Java 对象。如果从一个有一些示例性注释的空白的配置文件开始，可以很容易得将它修饰为你的应用所用。但是如果你只是遵循一些通用示例的话，也可能容易丢掉一些重要的特征。

大多数 Java 开发人员都知道，如果他们需要关于 Java 类的更详细的信息时，他们会查找 JavaDoc。但是，你如何查找关于一个 XML 文档的更多信息呢？

每一个良构的 XML 文档，包括 Struts 配置文件，都包括一个描述该文档可用元素的指针。这就是文档类型定义 (DTD)。如果你看看 struts-config.xml 文件的顶部，你就会发现这个元素：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation/DTD Struts Configuration
1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">
```

这是告诉我们这个文档的 DTD 的官方参考版本可以从所标示的 URL 找到。在内部，Struts 使用来自于 Jakarta Commons [ASF, Commons] 项目的 Digester 来解析 Struts 配置文件。Digester 使用 struts-config DTD 来校验文档的格式，并且创建文档所描述的 Java 对象。如果 XML 文件包含了非正式文档化的元素，或者以非正式文档化的方式使用了元素，Digester 将不会处理这个文件。

如果基于某些原因，XML 校验产生了问题，你可以使用 4.3 节所描述的 validating servlet 参数将校验特性关闭。但我们不推荐这样做。

在内部，Struts 使用其自己的 DTD 拷贝来处理配置。并不是每次在你的应用程序载入的时候都要从 Internet 取回 DTD 的参考版本。（基于某些神秘的原因，一小部分开发人员说，将 validating 设置为 false 好像可以在没有 Internet 连接的时候有助于应用的载入。通常，不管有没有 Internet 连接你都应该设置 validating=true。）

下表总结了可以在 Struts 1.1 中使用的配置元素。对 Struts 1.1 新加的元素我们会简要说明。

元素	说明
data-sources	包含一个 DataSource 对象 (JDBC 2.0 Standard Extension) 的集合
data-source	标识一个 DataSource 对象，可以被实例化，和进行配置，并在 servlet 上下文中作为一个属性 (或者在 application-scope 的 bean 中)
set-property	标识一个附加的 JavaBean 配置属性的方法名称和初始化值。
global-exceptions	描述一个可以被 Action 对象抛出的例外的集合

	从 Struts 1.1
exceptions	为一个例外类型注册 ExceptionHandler 从 Struts 1.1
form-beans	描述这个应用模块中的 form bean 描述符集合
form-bean	描述一个可以被<action>元素引用的 ActionForm 子类
form-properties	描述一个 JavaBean 属性, 可用来配置一个 DynaActionForm 实例或者其子类 从 Struts 1.1
global-forwards	描述对所有 Action 对象都可以作为返回值的 ActionForward 对象集合
forward	描述一个可以被 Action 作为返回值的 ActionForward 对象
action-mappings	描述一个可以用来处理匹配 ActionServlet 注册到容器的 url-pattern 格式的请求的 ActionMappings 对象集合
action	描述一个 ActionMapping 对象, 可以用来处理一个对特定的模块相关的 URI 的请求
controller	描述一个封装了应用模块运行时配置的控制器配置 bean 从 Struts 1.1
message-resources	描述该模块的消息模板一起的消息资源 MessageResources 对象 从 Struts 1.1
plug-in	标识一个通用应用的 plug-in 模块的全限定类名, 它接受应用的启动和退出事件的通知 从 Struts 1.1

如上所述, 现今许多编成组件都以 XML 文件的方式进行配置, 包括 Java servlet 容器。关于 Java 和 XML 的更多知识, 我们推荐 J2EE and XML Development [Gabrick] 这本书。

### 7.3.4.1. <global-exceptions>

在一个 Struts 应用中, ActionServlet 位于调用树的最顶层, 但工作会委托给 Action 对象。这种分而治之的策略在许多环境下都工作的很好, 例外会被进行例外处理。许多应用喜欢用一致的方式来处理例外, 但这会在多个 Action 复制例外处理代码。

为了在全部 Action 对象中进行一致的例外处理, 你可以在一个 Struts 配置文件中注册一个 ExceptionHandler 。 框架 提供 了一个 缺省 的 ExceptionHandler (org.apache.struts.action.ExceptionHandler) , 它可以在一个请求范围的属性下存储例外, 为里外信息创建 ActionError 对象, 并转发控制到 JSP 或 其他你选择的 URI。

Struts <html:errors> 标签会自动输出你的例外信息的本地化版本。所以, 你可以使用同一个页面来显示你想用来显示的校验错误的例外错误信息。

如果你还需要做些其他的事情, ExceptionHandler 可以被子类化而加入新的行为。如果需要的话, 每个里外都可以标识其自己的句柄类。

你可以为一个例外注册一个全局句柄以及针对某个 ActionMapping 的局部句柄。要注册一个例外, 你需要提供 Exception 类型, 消息关键字, 以及响应路径, 如下所示:

```
<exception type="org.apache.struts.webapp.example.ExpiredPasswordException"
key="expired.password" path="/changePassword.jsp"/>
```

### 7.3.4.2. <form-beans>

Struts ActionForm (org.apache.struts.action.ActionForm) 提供了一个方便的存储通过 HTTP 请求提交的输入属性的方法。但是为了存储输入属性，控制器必须首先创建一个 ActionForm 并将其存储在请求或者会话的上下文中，在这里，可以被其他框架组件—比如 JSP 标签—所找到。

如果在输入页面上有多个表单，那么每个表单都需要其 ActionForm 对象有不同的属性名。因此，你不能仅使用标准的命名。因为 bean 特性名称是其公共 API 的一部分，我们应该能提供开发者友好的 ActionForm 命名，比如 logonForm 之类。

某些特别的 ActionForm，比如如 DynaActionForms (org.apache.struts.action.DynaActionForm)，需要在创建时传递额外的属性。所以，我们也需要有地方来放置这些元素。ActionFormBean (org.apache.struts.action.ActionFormBean) 通过将之存储为一个 ActionForm 对象描述符来解决了所有的这些问题。每个 ActionFormBean 都有描述一个 ActionForm 特性名称和类型的属性。ActionFormBean 也包含 property 特性，可以在 DynaActionForm 中使用。

Struts 配置文件提供了一个 <form-bean> 元素来归类一个模块所使用的 ActionFormBean。每个 ActionFormBean 都由一个相应的 <form-bean> 元素创建。在运行时，控制器调用适当的 ActionFormBean 来找出哪一个 ActionForm 对象被创建，在哪里存储它，以及什么特性需要使用。

下面是一个针对常规 ActionForm 和 DynaActionForm 的 <form-bean> 元素配置：

```
<form-bean name="menuForm" type="org.apache.struts.scaffold.MenuForm"/>
<form-bean name="logonForm" type="org.apache.struts.action.DynaActionForm">
  <form-property name="username" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String"/>
</form-bean>
```

menuForm <form-bean>表达了一个常规的 ActionForm 子类；它要求一个对应的 Java 类支持。logonForm <formbean>并不要求使用一个特别的子类，但可以使用 DynaActionForm。(DynamicActionForm 从 Struts 1.1 中引入)

### 7.3.4.3. <global-forwards>

通过集中细节，Struts 配置将改变最小化。当环境改变时，多数实现细节可以通过培植进行改变，而不用改动 Java 或者 JSP 源代码。

Web 应用中一个令人痛苦的细节处理就是 URI [W3C, URI]。多数 URI 都直接映射到应用目录书中的物理文件。这对常规的站点来说还比较容易。要将“一个页面放到 web 上，”你只需要讲页面存储到站点的某个目录。而目录已经映射到站点的公共 URI，没有其他需要配置的了。

发布 web 页面其实就是传输文件。这其实很简单，直到你想删除么某些页面或者使用不同的页面。当这些事情发生了，(实际上它们经常发生)，你不得不更新应用中所有对该页面的引用之处。如果漏掉了一些，某些地方仍然引用到旧的页面，你就会遇到我们的数据库人员所称的“异常更新”(update anomaly)。两个假定为相同的情况现在都不同了。数据库对这个问题的解决方案是规范化(normalization.)我们将情况存放进一个表，而每个人都从这个表中进行查找。如果情况改变，我们仅仅需要更新情况表，每个人都会被带到相同的页面。

Struts 处理 URI 表的方法是 ActionForward。一个 ActionForward 对应一个 URI 的逻辑名称。其他组件可以引用这个名称，而不需要知道任何有关 URI 的情况。如果 URI 改变，我们只需要改变它的 ActionForward。

其他组件通过请求 ActionForward 的路径来得到更新过的 URI。这样，通过封装实现细节到逻辑名称之后，我们最小化了变更并减少了潜在错误。

ActionForward 的主要使用者是 Action 对象。当一个 Action 完成时，它返回一个 ActionForward 或者 null。如果 Action 没有返回 null，ActionServlet 就将控制转发到返回的 ActionForward 的路径。典型地，Action 将通过名称查找 ActionForward，而不需要知道关于 URI 的任何事情。你可以部署一个全局转发 Global ActionForward 在 <global-forwards>元素中，像这样：

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

这些 forward 对应用中的每个 Action 都有效。你也可以部署一个局部 ActionForward 到<action> 元素中。局部转发仅针对该 ActionMapping 有效。

#### 7.3.4.4. <action-mappings>

ActionForm 将应用需要的数据存储到 collect 中。ActionForward 归类那些应用要用的 URI。

ActionMapping 描述应用要采取的操作、命令。

Action 对象处理操作的实际工作。但一个操作有大量的管理细节。ActionMapping 就是用来包装这些细节。

一个重要的细节就是用来调用 Action 对象的 URI [W3C, URI]。Action 的 URI 被用作一个 ActionMapping 的逻辑标识符，或者路径。当 web 浏览器请求一个 Action 的 URI，ActionServlet 首先查找相应的 ActionMapping。ActionMapping 则告诉 ActionServlet 哪个 Action 对象要用于这个 URI。

除了 URI 路径和 Action 类型以外，ActionMapping 还包含了几个可以用来在 Action 被调用时发生的行为的属性。改变这些属性会改变 Action 对象的行为。这可以帮助开发人员是同一个 Action 对象有更多的用途。如果没有 ActionMapping 对象，开发人员可能需要创建比现在多得多的 Action 类。

你也可以使用 ActionMapping 来简化到另一个路径的转发或者重定向。但绝大多数情况下，它只是用来连接 Action 对象。

<action-mappings> 元素描述了我们的应用要用来处理请求的 ActionMapping 对象 (org.apache.struts.action.ActionMapping) 的集合。请求要到达应用然后到达 ActionServlet，它必须匹配上下文和我们在容器中注册的 url-pattern 格式。因为所有的请求都匹配这个格式，我们就不需要使用上下文或者 url-pattern 来标识一个 ActionMapping。所以如果如果 URL 是针对 http://localhost/myApp/myAction.do 我们只需要引用 /myAction 作为 ActionMapping 的路径。每个 ActionMapping 都由对应的嵌入 <action-mappings> 元素中的<action> 元素创建，如下所示：

```
<action-mappings>
    <action path="/logoff" type="app.LogoffAction"/>
    <action path="/logonSubmit" type="app.LogonAction" name="logonForm" scope="request">
```

```
validate="true" input="/pagess/Logon.jsp"/>
<action path="/logon" type="app.ContinueAction">
    <forward name="continue" path="/pagess/Logon.jsp"/>
</action>
<action path="/welcome" type="app.ContinueAction">
    <forward name="continue" path="/pagess/Welcome.jsp"/>
</action>
</action-mappings>
```

一个 ActionMapping 可以引用很多属性。紧随 Action 对象之后，ActionMapping 可能是 Struts 应用另一个最重要的对象。

#### 7.3.4.5. <Controller>

Struts 允许多个应用模块共享一个单一的控制器 servlet。每个模块有其自己的 Struts 配置并且可以相对于其他模块独立开发。<controller> 元素允许每个模块为 ActionServlet 标识一套不同的配置参数。它们大多数是部署描述符中的原始 <init-params> 设置。

<controller> 元素设置的属性值存储在一个控制器配置 bean (org.apache.struts.config.ControllerConfig) 之中。每个应用模块要创建一个控制器配置，包括缺省的根模块。如果一个模块的 struts-config 提供了一个 <controller> 元素，就是用来设置模块的控制器配置 bean 的属性。

因为各个模块共享 ActionServlet，你也可以为每个模块插入不同的请求处理器。这可以使每个模块按自己的方式处理请求，而不用子类化共享的 servlet。下面是一个 <controller> 元素的例子，它设置 nocache 和 null 配置属性为 true 并且装入一个定制的请求处理器：

```
<controller nocache="true" null="true"
processorClass="com.myCompany.struts.RequestProcessor"/>
```

请求处理器是 ActionServlet 处理循环的核心。大多数情况下，你可以编写和装入一个请求处理器，来代替创建你自己的 ActionServlet 子类。

#### 7.3.4.6. <message-resources>

每个模块都应该有其自己的缺省消息资源束。这是一个 Struts 组件，如 JSP 标签，在没有其他特别标明的情况下要使用的资源束。你也可以装入其它额外的资源束，连同特定的消息模板。例如，许多开发人员喜欢将图像相关的信息放在一个单独的资源中。

<message-resources> 元素用来部署应用需要使用的资源束。下面是一个 <message-resources> 元素的例子，它为模块部署了缺省的资源束，另一个则部署了一个图像消息的资源：

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

在需要时，框架会在名为 resources 的包中的名称为 application.properties 的文件中寻找缺省的消息资源束。包，或者文件文件夹，可以在容器的 classpath 路径的任何地方。典型的，资源束也以 JAR 文件的方式，或者放在 WEB-INF/classes 文件夹中。

如果 JSP 标签，或者其他组件，标识了 resources.image 资源束，框架会在 resources 包中查找名为 image.properties 的文件。

### 7.3.4.7. <plug-in>

对 Action 来说，需要特别的资源来完成其工作的情况并不常见。不过，它可能需要使用一个非数据源兼容的连接池。或者也许需要创建一个应用 bean 来为表单使用。也许需要读入自己的配置文件来创建一系列的对象，就像 struts-config 所做的一样。在一个常规 web 应用中，这些任务通常委托给一个特殊的 servlet。在 Struts 应用中，我们倾向于将这些任务委托给 Action。当一个 Action 需要初始化和销毁它自己的资源时，它可以实现 PlugIn 接口 (org.apache.struts.action.PlugIn)。这个接口宣称了 init 和 destroy 方法，控制器可以在适当的时候进行调用。

PlugIn Action 可以在 Struts 配置中通过<plug-in>元素进行注册。下面是一个标准的 plug-in，用来初始化 Struts Validator:

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

### 7.3.4.8. <data-sources>

虽然 Struts 框架是模型中立的，它仍然需要和业务层甚至数据-访问层进行交互。一个组件期望调用这给它传递一个活动的 SQL 连接 (java.sql.Connection) 的情况并不是不常见。这也使调用者 (比如 Struts) 有责任管理连接的生命周期。为了提供给应用在连接到数据访问组件时更多的灵活性，JDBC 2.0 标准扩展包提供了一个基于工厂的方法来获取数据。对一个应用连接到数据库，或者其他数据服务的首选方法是使用一个实现数据源接口 (javax.sql.DataSource) 的对象。

在一个 web 应用中，一个数据源对象通常代表一个应用中所有用户都可以共享使用的连接池。获取一个数据库连接可能在实践和资源上都有很昂贵的代价。典型地，web 应用以一个单独的账户登陆到数据库中，然后自己管理这个单独账户的安全性。

为了帮助开发人员使用连接，Struts 提供了一个数据源管理组件。你可以使用这个组件来实例化和配置一些实现数据源的对象，并且可以从 JavaBean 的属性进行整体配置。

如果你的数据库管理系统没有提供满足这两个要求的组件，你也可以使用 JakartaCommons 数据库连接池基本数据源类 (org.apache.commons.dbcp.BasicDataSource)。在 Struts 1.1 中，Struts 通用数据源 (org.apache.struts.util.GenericDataSource) 是一个 BasicDataSource 类的包装类。(这个 Struts 类已经不推荐了，仅提供向后兼容之用)。

如果你的数据库管理系统提供了它自己的数据源可供 Struts 使用，你就应该考虑使用这个实现。在 Struts 中使用 BasicDataSource 或者 GenericDataSource 并不比使用其它类更有效率。要根据你的环境选择最好的实现。

你也可以配置不止一个数据源，然后根据名称进行选择。这个特征可以提供更好的安全型和扩展型，或者用一个数据源实现和另一个进行比较。下面是一个数据源配置，使用 MySQL 数据库的 Struts 缺省配置：

```
<data-sources>
    <data-source>
        <set-property property="maxCount" value="4"/>
        <set-property property="minCount" value="2"/>
        <set-property property="description" value="Artimus:MySQL Data Source Config"/>
    </data-source>
</data-sources>
```

```
<set-property property="driverClass" value="org.gjt.mm.mysql.Driver"/>
<set-property property="url" value="jdbc:mysql://localhost:3306/artimus"/>
<set-property property="autoCommit" value="true"/>
<set-property property="user" value="root"/>
<set-property property="password" value="" />
</data-source>
</data-sources>
```

不像其他 struts 配置元素，<data-source> 元素非常依赖于 <set-property> 元素。因为开发人员经常需要配置他们自己的 DataSource 子类，所以只有较少的属性内建进了 <datasource> 元素之中。数据源对象是 Struts 和数据访问层之间的桥梁。而配置中的其他组件组成了 Struts 的控制 层。

### 7.3.4.9. <set-property>另类用法

如果你子类化了一些 Struts 配置对象，你可以使用<set-property>元素来自己的属性到子类之中。这就使你可以扩展框架类，而不用改变配置文件如何解析。下面是一个例子，它传递一个 XLS 样式表引用到一个（假定的）ActionForward 对象的定制实现：

```
<global-forwards type="app.struts.XlsForward">
  <forward name="logon">
    <set-property property="styleName" value="default"/>
    <set-property property="stylePath" value="/logon.xls"/>
  </forward>
</global-forwards>
```

当 logon 元素的 XlsForward 对象被实例化时，Digester 相当于调用  
logon.setStyleName("default");  
logon.setStylePath("/logon.xls");

你也可以在很多 Struts 配置元素中使用这种方法，使所有的对象都成为完全可插入的。

### 7.3.5. Struts config 骨架

以下代码是一个 Struts 配置文件的骨架，展示了最常用的元素和属性。这个文件也和 Struts 空白应用中的配置文件非常相似：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Config 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <data-sources>
    <data-source>
      <set-property name="${}" value="${}" />
    </data-source>
  </data-sources>
  <form-beans>
    <form-bean name="${}" type="${}">
      <form-property name="${}" type="${}" />
    </form-bean>
  </form-beans>
</struts-config>
```

```
</form-bean>
</form-beans>
<global-exceptions>
    <exception type="${}" key="${}" path="${}" />
</global-exceptions>
<global-forwards>
    <forward name="${}" path="${}" />
</global-forwards>
<action-mappings>
    <action path="${}" type="${}" name="${}" scope="${}"
        validate="${}" input="${}">
        <forward name="${}" path="${}" />
        <exception type="${}" key="${}" path="${}" />
    </action>
</action-mappings>
<controller processorClass="${}" />
<message-resources parameter="${}" />
<plug-in className="${}">
    <set-property property="${}" value="${}" />
</plug-in>
</struts-config>
```

完整的 Struts 配置元素和属性清单, 请查阅 struts-config API。

### 7.3.6. 应用资源文件

Struts 框架提供了好用和灵活的消息系统。在 Java 和 JSP 代码中, 要给定一个消息的关键字; 消息文本在运行时从属性文件中检索。框架文档将消息属性文件引用为 application resources 或者 message resource bundle。

如果你想要本地化你的应用, 你可以为你想要支持的场所创建一个额外的应用资源文件。这实际上是创建一个资源束 (java.util.ResourceBundle)。框架会为每个用户维护一个标准的 Locale 对象 (java.util.Locale)。针对用户场所的合适的消息会自动从资源束中进行选取。

#### 定义:

*Locale (场所)* 对象是一个特定的语言和地区的识别符。

*ResourceBundle* 对象包含场所特定的对象。当需要一个场所特定的对象时, 可以从资源束中取得, 它返回匹配用户场所的对象。Struts 框架为消息文本使用基于字符串的资源束。

属性文件自身是一个平面的文本文件, 每一行是一个关键字-值对。你可以使用任何文本编辑器进行编辑, 包括 Windows Notepad。

应用资源文件的缺省名称是通过在 web.xml 向 Struts ActionServlet 传递一个初始化参数决定的。下面这个片断, 参数的名称是 application:

```
<init-param>
    <param-name>application</param-name>
    <param-value>application</param-value>
</init-param>
```

这个参数没有缺省值。在应用中使用 Struts 应用资源束之前必须首先进行配置。应用资源文件位于应用的 CLASSPATH 之中，这样 Struts 可以找到它。最好是放在你的应用的 class 文件夹中。这可能是在 WEB-INF/classes 文件夹中，或者，如果你以二进制部署你的应用时在 WEB-INF/lib 下的一个 JAR 文件中。

param-value 应该使你的文件按包命名格式的全限定名称。这意味着如果将资源文件直接放在 classes 下，你可以直接使用文件名，如前面的代码片断所示。

如果你将文件放在一个子目录中，那么该子目录就相当于一个 Java 包。如果应用资源束在一个名为 resources 的子目录下，你就应该这样来标识：

```
<init-param>
    <param-name>application/param-name>
    <param-value>resources.application</param-value>
</init-param>
```

物理文件的系统路径应该是：

WEB-INF/classes/resources/application.properties

如果你将类移到了 JAR 中，不需要进行什么改变。Struts 可以在 JAR 文件中找到资源束，就想找到其他类一样。为了本地化你的应用，为每个支持的场所添加资源文件，并修改基本名称：

WEB-INF/classes/resources/  
application.properties  
application\_es.properties  
application\_fr\_CA.properties

Application 名称只是一个习惯。对框架来说没有缺省设定。你可以将名称改为你认为适合的任何名字。另一个通用的缺省是使用应 applicationResources 作为名称，因为它一些早期的 Struts 例子中经常使用。

### 7.3.7. 配置 Struts 核心

当此为止，我们已经全部涉及了要使你的 Struts 应用运行需要构建和定制的三个文件。

- 部署描述符 (web.xml)
- Struts 配置文件 (struts-config.xml)
- 应用资源束 (application.properties)

#### 7.3.7.1. 安装 Java 和 Java servlet 容器

第一步是安装一个 servlet 容器，比如 Tomcat。

#### 7.3.7.2. 安装开发环境

Eclipse+MyEclipse。

#### 7.3.7.3. 安装 Struts 核心文件

运行 Struts 需要的常备文件都在 Struts 库文件分发包 (jakarta-struts-1.1-lib.zip) 中提供了。这些包括几个 JAR，标签库描述符，DTDs，和标准的 XML 配置文件。这些常备文件和你要提供的 4 个配置文件一起，创建了一个核心的 Struts 配置。

#### 7.3.7.4. 配置 Tiles 框架

Tiles 是一个 Struts 框架的可选组件，是一个强大的页面组装工具，在其意义上是一

个真正的框架（我们将在后面的章节详细讨论 Tiles 框架的使用）。使用 Struts 框架的其他部分时并不需要使用和配置 Tiles。但是如果你喜欢 Tiles，这就是一个练习。

你需要用来使用 Tiles 的所有文件都在 Struts library distribution 中提供了。如果你让你的应用基于 Struts 空白应用或者你已经安装了所有必需的 Struts 库文件夹到应用的 /WEB-INF 或者 /WEB-INF/lib 文件夹中，那么基本的所需文件已经有了。

下面是检查表：

- 从 Struts lib 文件夹拷贝 struts-tiles.tld 和 tiles-config.dtd 文件（如果没有）到 /WEB-INF 文件夹。
- 插入下面的语句快到（如果没有）到 /WEB-INF/web.xml 文件中，并且紧跟其他 <taglib> 元素：

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

- 创建一个空白的 tiles-defs.xml（如果没有）在 /WEB-INF 文件夹中，像这样：

```
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles Conig//EN"
"http://jakarta.apache.org/struts/dtds/tiles-config.dtd">
<tiles-definitions>
    <!-- skeleton definition
        <definition name="${name}" path="${path}">
            <put name="${name}" value="${value}" />
        </definition>
        end blank definition -->
</tiles-definitions>
```

- 插入这个 <plug-in> 元素到 struts-config.xml，位置在关闭的 </struts-config> 元素之前：

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
    <set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" />
</plug-in>
```

### 7.3.7.5. 配置 Struts Validator

像 Tiles 一样，Struts Validator 也是框架的一个可选组件。你需要用来使用 Validator 的所有文件都在 Struts library distribution 中提供了。如果你让你的应用基于 Struts 空白应用（4.10）或者你已经安装了所有必需的 Struts 库文件夹到应用的 /WEB-INF 或者 /WEB-INF/lib 文件夹中，那么基本的所需文件已经有了。

下面是检查表：

- 从 Struts lib 文件夹拷贝 struts-validator.tld 和 validator-rules.xml 文件（如果没有）到 /WEB-INF 文件夹。
- 插入下面的语句快到（如果没有）到 /WEB-INF/web.xml 文件中，并且紧跟其他 <taglib> 元素：

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

- 创建一个空白的 validations.xml (如果没有)在 /WEB-INF 文件夹中, 像这样:

```
<form-validation>
<formset>
<!-- skeleton form
<form name="${ }">
<field
property="${ }"
depends="${ }">
<arg0 key="${ }"/>
</field>
</form>
end skeleton form -->
</formset>
</form-validation>
```

- 插入这个 <plug-in> 元素到 struts-config.xml , 位置在关闭的 </struts-config> 元素之前:

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

### 7.3.8. 配置模块化应用

Struts 架构的一个关键优点是所有的请求都通过一个单一的控制点。开发人员可以集中那些应用于每一个请求的功能，并且避免整个应用中的代码重复。因为 Java 是一个多线程平台，Struts 使用一个单独的控制器 servlet 也提供了最好的性能可能性。开发人员编写更少的代码，机器在更少的资源下更快的运行。一切都很完美。

在 Struts 1.0 中，引用以单独模式运行。Struts 配置文件对用户来说是一个。如果多个开发人员工作于一个应用，他们需要有一个方法来管理 Struts 配置的更新。

在实践中，开发人员倾向于将他们的工作分割为逻辑单元，非常想他们将 Java 代码分派到包中。团队成员在应用中有他们自己的名字空间并不是不常见。

在一个在线拍卖应用中，一个团队可能工作于 registration 模块；另一个团对可能工作于 bidding 模块。第 1 个团队可能具有一个/reg 文件夹来存放他们的 JSP 页面以及一个 app.reg 包来存放他们的 Java 代码。同时，团队 2 可能有一个/bid 文件夹存放 JSP 页面以及一个 app.bid 包来存放 Java 代码。在资源文件中，每个团对可能都有他们以 reg. 或 bid. 为前缀的关键字。

当然，你不需要按这种方式以大型团队来组织大型项目。许多单独的开发者也做同样的事情。限制一个文件夹中的文件数量和包中的类数量，被认为是很好的项目管理。

许多开发人员也按逻辑模块来组织项目，而不管是否需要共享文件。

在 Struts 1.1 中，将应用分为模块的理念成为了一种习惯—现在已经集成到框架中了。让我们回头看看，Struts 是如何将应用组织为模块的。

web 应用容器是我们可以为每个应用创建上下文来共享服务。这个上下文对应于服务器的 webapp 目录的一个子目录。按同样的道理，Struts 1.1 通过为每个应用创建一个前缀来共享应用的使用。多个模块可以运行于相同的应用空间，每一个都在起自己的前缀之下，与多个应用可以运行于同一个服务器空间非常相似—每一个都有其自己的上下文。

我们在编写 web 应用时，我们经常引用到上下文-相关的 URI。这是一个不包含应用名字或上下文的路径。同时，我们在编写 Struts 应用时，我们也经常应用一个模块-相关的 URI。

不用惊讶，这也是一个路径，不包含模块名称，或者模块前缀。表 4.4 所示的是绝对的，上下文-相关的，和模块相关的同一个 URI。就像你可以编写一个 web 应用，并将它配置在某个上下文中一样，你可以编写一个 Struts 应用模块并配置它在某个前缀下运行。

编写一个模块和编写一个独立运行的应用并没有多大的不同。所有世纪的配置都在部署描述符中。如果你将某个模块从某个前缀移动到根，或者从根移动到某个前缀下，或者从一个前缀移动到另一个前缀，模块中没有 JSP, Java 代码，或者 XML 代码需要改变。要设置一个应用来使用分离的 reg 和 bid 模块，我们可以这样配置 servlet 描述符：

```
<servlet>
<servlet-name>action</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
<!-- The default (or "root") module -->
<init-param>
<param-name>config</param-name>
<param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
<!-- The register module -->
<init-param>
<param-name>config/reg</param-name> <-- Includes prefix! -->
<param-value>/WEB-INF/struts-config-reg.xml</param-value>
</init-param>
<!-- The bidding module -->
<init-param>
<param-name>config/bid</param-name> <-- Includes prefix! -->
<param-value>/WEB-INF/struts-config-bid.xml</param-value>
</init-param>
<!-- ... other servlet elements ... -->
</servlet>
```

在这个例子中，reg 团队可以工作在 struts-config-reg.xml 而 bid 团队可以工作在 struts-config-bid.xml 配置。每个团队都工作于他们的模块，就像她们工作于一个单模块的应用。框架作了关于模块前缀的所有调整，就像容器为应用上下文所作的调整和配置。

## 7.4. Struts 控制器组件

Struts 控制器组件负责接受用户请求、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 控制器组建主要包括：

- 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，部需要修改整个应用，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，错综复杂的连接关系，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。

Struts 的控制器组件主要完成以下任务：

- 接受用户请求
- 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。

### 7.4.1. Action 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有 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。

Struts 框架初始化过程

- (1) 调用 initInternal() 方法，初始化 Struts 框架内的消息资源，如与系统日志相关的同志、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)～(7)，分别对其他子应用模块进行初始化。

#### 7.4.1.1. ActionServlet 的 process()方法

当 ActionServlet 实例接受到 HTTP 请求之后，在 doGet() 或 doPost() 方法都会调用 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);
}
```

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()` 方法的第二步操作作为获得 `RequestProcessor` 类的实例，然后调用 `RequestProcessor` 类的 `process()` 方法，来完成十几的预处理请求操作。

### 7.4.1.2. 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。

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` 的子类，然后在 `web.xml` 文件中配置这个客户化 `ActionServlet` 类。

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，还可以根据十几需要覆盖 `ActionServlet` 的任何其他方法。

### 7.4.2. 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。

#### 7.4.2.1. RequestProcessor 类的 process() 方法

`RequestProcessor` 类的 `process()` 方法负责实际的预处理请求操作。

`RequestProcessor` 类的 `process()` 方法一次执行以下流程：

(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。

(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。

(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。

(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 isuserrole() 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 processactionform() 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。

(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。

(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。

### 7.4.2.2. 扩展 requestprocessor 类

开发人员可以很方便地创建客户化的 requestprocessor 类。

在 struts 配置文件中，<controller>元素的 processorclass 属性 requestprocessor 类。

### 7.4.3. 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。

### 7.4.3.1. 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，以后所有的线程就会直接使用这个缓存中的实例。

### 7.4.3.2. 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
- 在 struts 配置文件中配置<forward>元素。

配置了<forward>元素后，在 struts 框架初始化时就会创建存放<forward>元素；

在 execute() 方法中只需调用 actionmapping 实例的 findforward() 方法，来获取特定的 actionforward 实例。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。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无力路径，而是在配置文件中配置转发资源，程序中秩序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。

### 7.4.3.3. 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，必需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元素是在 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。

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。

#### 7.4.3.4. 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>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。

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未 action 配置了安全角色，如果配置了安全角色，就调用 httpsservletrequest 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。

#### 7.4.4. 使用内置的 struts action 类

##### 7.4.4.1. 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，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。

有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 messengeresources 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 messengeresources 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，然后选择适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。

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。

##### 7.4.4.2. 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。

<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。

### 7.4.4.3. 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。

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，即他们的参数和返回类型都应该相同，此外也应该声明抛出 exception。

在配置 dispatchaction 类时，需要把 parameter 属性设置为“method”。设置之后，当用户请求访问 dispatchaction 时，应该提供 method 请求参数。

### 7.4.4.4. 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。

在 struts 配置文件中配置 lookupdispatchaction：

在<action>元素中，设置 parameter 属性时，需要使它和<html:submit>标签的 property 属性保持一致。

### 7.4.4.5. org.apache.struts.actions.switchaction 类

switchaction 类用于子应用模块之间的切换。对于请求访问 switchaction 的 url，需要提供两个参数：

- 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。

### 7.4.5. 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 应用必需能够判断用户的重复提交行为，以作出相应的处理。

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：

- Protected boolean istokenvalid(javax.servlet.http.HttpServletRequest)
- 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，就返回 false：
  - 不能存在 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。
- protected void resettoken(javax.servlet.http.HttpServletRequest request)

从当前 session 范围内删除令牌属性。

- 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唯一的令牌。

## 7.4.6. 实用类

在创建 web 应用时，有许多检索和处理 http 请求的操作时重复的。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。

### 7.4.6.1. 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。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。

**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

### 7.4.6.2. tagutils 类

org.apache.struts.taglib.TagUtil 类为 jsp 标签处理类提供了许多实例方法，如果要使用 TagUtil 类，首先应调用 TagUtil.getInstance() 方法，获得 TagUtil 类的实例，getInstance() 方法为静态方法。

**tagutils 类的常用方法**

方法	描述
getInstance()	返回一个 TagUtil 的实例。该方法为静态的，如果要在程序中获得 TagUtil 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 范围内减缩并返回 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)	向网页上输入特定的文本, 参数 text 用于指定文本内容

### 7.4.6.3. 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。

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
getmodulenname(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 messengeresources 对象存储到 request 范围中

#### 7.4.6.4. **globals** 类

org.apache.struts.globals 类提供一组公共类型的静态常量，被用作在特定范围内存放 javabean 的属性 key。

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

### 7.5. Struts 模型组件

模型代表应用的业务数据和逻辑。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)。

#### 7.5.1. 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，而上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：



7-18 Struts 应用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。

## 7.5.2. 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。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。

### 7.5.2.1. 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成类。

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概念模型中存在的问题,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。

### 7.5.2.2. 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。

根据 UML 语言,类直接存在四种关系。

- 1 关联(Association): 关联指的是类之间的引用关系。
- 2 依赖(Dependency): 依赖指的是类之间的访问关系。
- 3 累积(Aggregation): 累积指的是整体与个体之间的关系,可以把累积看作一种强关联关系。
- 4 一般化(Generalization): 一般化指的是类之间的继承元素。

### 7.5.3. 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。

业务对象包括状态和行为。

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。

### 7.5.3.1. 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种的业务过程或流程，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，他们可作为常规的 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。

事件业务对象代表应用中的一些时间（如异常、警告或超时）。这些时间通常由系统中的某种行为处罚。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时间处理器来处理事件。

### 7.5.3.2. 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他们。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他们。如果正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，对外只保露接口。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 来实现原有的功能，这种概念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，当应用中指运行时，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。

### 7.5.4. 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。

### 7.5.4.1. 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，他们要么从内存中清楚，要么被持久化到数据存储库中。内存无法永久保存数据，因此必需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。

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），因此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。

### 7.5.4.2. 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。

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。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，供业务层同一调用。

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性框架不是很可行。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。

### 7.5.4.3. 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。

TopLink

<http://otn.oracle.com/products/ias/toplink/content.html>

Torque

<http://jakarta.apache.org/turbine/torque/index.html>

ObjectRelationalBridge

<http://db.apache.org/obj>

FronierSuite

<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://sourceforget.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印入它们的类和接口，这会带来一个问题，如果日后想改用其他的 ORM 框架，就必需修改业务对象。

### 7.5.5. 小结

Struts 框架并没有在模型层提供线程可用的组件。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。

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。

## 7.6. 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经历放在实现业务离求上。

本节重点介绍 ActionForm 的运行机制和使用方法。此外还介绍了动态 ActionForm 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，将在后面的章节详细介绍。

### 7.6.1. 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。用户可以根据自己的需要，来访问不同的视图。

在 Struts 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

## 7.6.2. 在视图中使用 JavaBean

JavaBean 是可以重用的、平台独立性的 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，它应该遵循以下规范：

- 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 set XXX(int index, PropertyElement value) 方法，用来读取或设置数组中的元素。

### 7.6.2.1. DTO 数据传输对象

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，在模型层和视图层之间就通过 DTO 来传递数据，模型层的数据实际上经过控制层再达到视图层。采用 DTO 来传输数据有两个好处：

- 减少传输数据的冗余，提高传输效率。
- 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 DTO。

### 7.6.2.2. Struts 框架提供的 DTO：ActionForm Bean

ActionForm Bean 是 Struts 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。

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 的数据重新组件到自定义的 DTO 中，再把它传递给模型层。在各个层之间传输数据的 DTO 如下图所示：

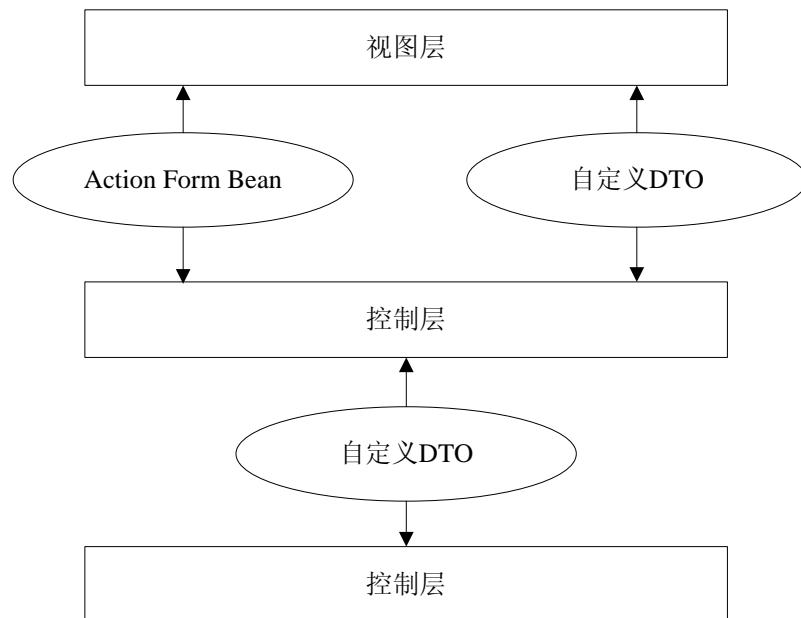


图 7-19 在各个层之间传输数据的 DTO

### 7.6.3. 使用 ActionForm

#### 7.6.3.1. ActionForm 的生命周期

ActionForm Bean 有两种存在的范围: 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 的范围如下图所示:

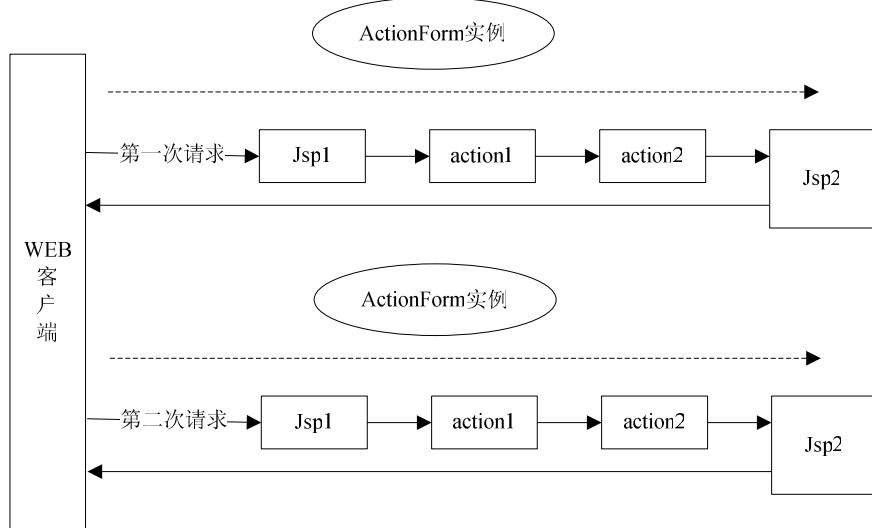


图 7-20: ActionForm 的范围

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 组件始终共用一个 ActionForm 实例。

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 范围中提取 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

的生命周期如下图所示：

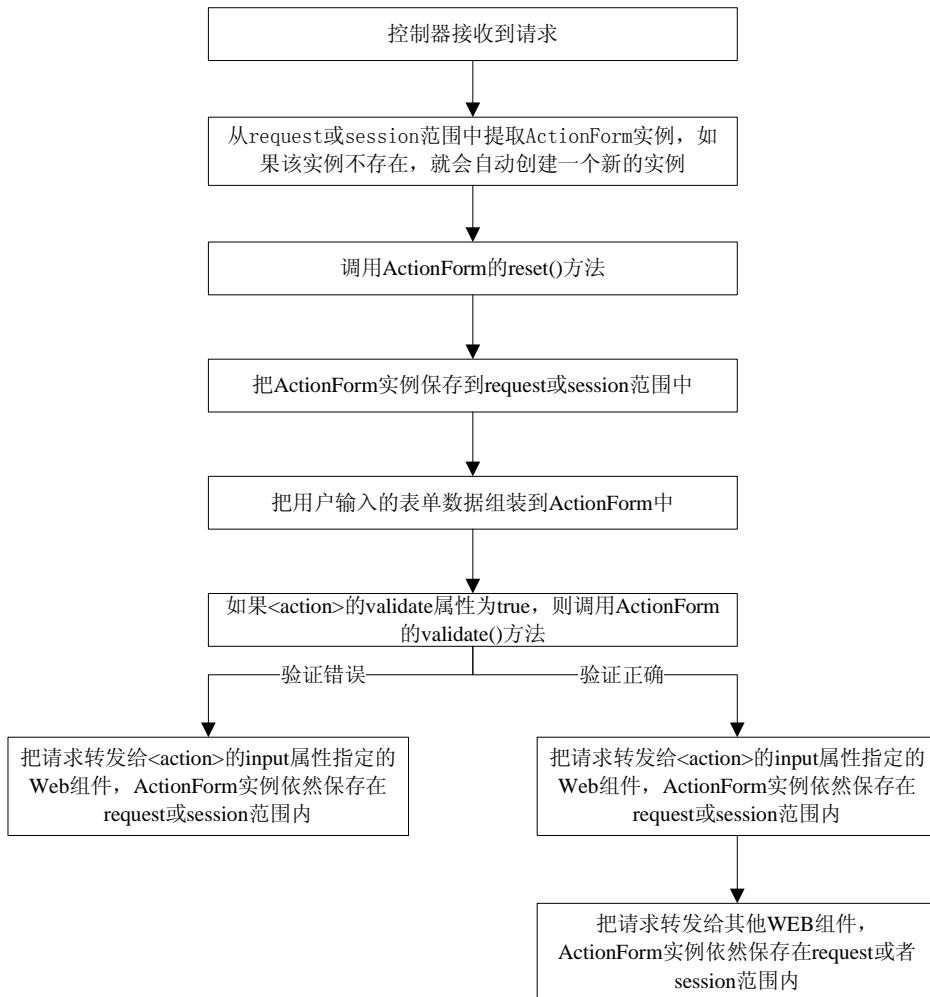


图 7-21： ActionForm 的生命周期

### 7.6.3.2. 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 的属性和 HTML 表单中字段一一对应。如下面的程序：

---

package com.cssp.cms.web.form;

```

import org.apache.struts.action.*;
import javax.servlet.http.*;
import com.cssp.comm.util.Util;

public class contentTypeActionForm
    extends ActionForm {
    private String action;
    private String ispublic;
    private String orders;
    private String parentid;
    private String typeid;
  
```

```
private String typename;
private String typetext;
public String getAction() {
    return action;
}

public void setAction(String action) {
    this.action = action;
}

public String getIspublic() {
    return ispublic;
}

public void setIspublic(String ispublic) {
    this.ispublic = ispublic;
}

public String getOrders() {
    return orders;
}

public void setOrders(String orders) {
    this.orders = orders;
}

public String getParentid() {
    return parentid;
}

public void setParentid(String parentid) {
    this.parentid = parentid;
}

public String getTypeid() {
    return typeid;
}

public void setTypeid(String typeid) {
    this.typeid = typeid;
}

public String getTypename() {
    return typename;
```

```
}

public void setTypename(String typename) {
    this.typename = typename;
}

public ActionErrors validate(ActionMapping actionMapping,
                            HttpServletRequest httpServletRequest) {
    ActionErrors errors = new ActionErrors();
    if (action == null || action.length() == 0) {
        action = "list";
    }
    if (Util.nullOrBlank(parentid))this.parentid = "0";
    if (action.equals("editdo") || action.equals("add")) {
        if (Util.nullOrBlank(parentid) || Util.nullOrBlank(typename)) {
            errors.add("error.nullerror", new ActionError("error.nullerror"));
        }
    }
    return errors;
}

public void reset(ActionMapping actionMapping,
                  HttpServletRequest httpServletRequest) {
}

public String getTypetext() {
    return typetext;
}

public void setTypetext(String typetext) {
    this.typetext = typetext;
}
```

---

### 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。

### 2、reset()方法

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 () 方法用户恢复 ActionForm 的属性默认值。

## 7.6.3.3. 配置 ActionForm

Struts 配置文件的<form-bean>元素用来配置所有的 ActionForm Bean。<form-beans>

元素可以包含多个<form-bean>元素，它代表单个 ActionForm Bean，例如：

```
<form-beans>
    <form-bean name="adminLoginForm" type="com.cssp.cms.web.form.AdminLoginForm" />
        <form-bean name="adminChangePwdActionForm" type="com.cssp.cms.web.form.adminChangePwdActionForm" />
            <form-bean name="contentTypeActionForm" type="com.cssp.cms.web.form.contentTypeActionForm" />
                <form-bean name="userInfoActionForm" type="com.cssp.cms.web.form.userInfoActionForm" />
                    <form-bean name="filesInfoActionForm" type="com.cssp.cms.web.form.filesInfoActionForm" />
                        <form-bean name="contentInfoActionForm" type="com.cssp.cms.web.form.contentInfoActionForm" />
                    </form-bean>
    </form-beans>
```

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（默认为 true）指定是否执行表单验证：

```
<action input="/error/error.jsp" name="adminLoginForm" path="/adminLoginAction" scope="request" type="com.cssp.cms.web.action.config.AdminLoginAction" validate=true>
    <forward name="adminMain" path="/config/adminMain.jsp" />
</action>
```

#### 7.6.3.4. 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，这些标签构成 HTML 表单字段或者按钮。<html:form>标签能和 ActionForm 交互，读取 ActionForm 的属性值，把他们赋值给表单中对应的字段。

##### 2. 从 request 或 session 范围内取出 ActionForm 实例

Struts 框架把 ActionForm 实例保存再 HttpServletRequest 或 HttpSession 中，保存时采用属性的 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 的实例，例如：

```
HelloWorldForm HelloForm2 = (HelloWorldForm)request.getAttribute("HelloWorldForm");
```

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：

```
HelloWorldForm HelloForm3 = (HelloWorldForm)session.getAttribute("HelloWorldForm");
```

##### 3. 在 Action 类的 execute() 方法中直接访问 ActionForm

如果配置了 ActionForm 和 Action 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() 方法中可以读取或者设置 ActionForm 的属性，如下面代码所示：

```
public ActionForward execute(ActionMapping actionMapping,
```

```
ActionForm actionForm,
HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse) {
    AdminLoginForm adminLoginForm = (AdminLoginForm) actionForm;
    ActionErrors errors = new ActionErrors();
    CheckAdminUser myCheckAdminUser = new CheckAdminUser();
    MacMD5 md5 = new MacMD5();
    if
        (myCheckAdminUser.checkUserOK(md5.CalcMD5(adminLoginForm.getUsername()),
            md5.CalcMD5(adminLoginForm.getPassword())))
    {
        HttpSession session = httpServletRequest.getSession();
        session.setAttribute("adminUserName",
            md5.CalcMD5(adminLoginForm.getUsername()));
        session.setAttribute("adminPasswd",
            md5.CalcMD5(adminLoginForm.getPassword()));
        myCheckAdminUser = null;
        return actionMapping.findForward("adminMain");
    }
    else {
        errors.add("error", new ActionError("error.pwderror"));
        saveErrors(httpServletRequest, errors);
        myCheckAdminUser = null;
        httpServletRequest.setAttribute("adminLoginForm", adminLoginForm);
        return actionMapping.findForward("adminLogin");
    }
}
```

---

## 7.7. 扩展 Struts 框架

一个好的软件框架应该具备可扩展特性。在 Struts 框架中提供了许多可扩展之处，不妨将其称为可扩展点（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。

### 7.7.1. PlugIn

根据 Struts 文档，“PlugIn 是一个须在应用启动和关闭时需被通知的模块定制资源或服务配置包”。这就是说，你可以创建一个类，它实现 PlugIn 的接口以便在应用启动和关闭时做你想要的事。

假如创建了一个 web 应用，其中使用 Hibernate 做为持久化机制；当应用一启动，就需要初始化 Hinernate，这样在 web 应用接收到第一个请求时，Hibernate 已被配置完毕并待命。同时在应用关闭时要关闭 Hibernate。跟着以下两步可以实现 Hibernate PlugIn 的需求。

1. 创建一个实现 PlugIn 接口的类，如下：

```
public class HibernatePlugIn implements PlugIn{
    private String configFile;
    // This method will be called at application shutdown time
    public void destroy() {
        System.out.println("Entering HibernatePlugIn.destroy()");
        //Put hibernate cleanup code here
        System.out.println("Exiting HibernatePlugIn.destroy()");
    }
    //This method will be called at application startup time
    public void init(ActionServlet actionServlet, ModuleConfig config)
        throws ServletException {
        System.out.println("Entering HibernatePlugIn.init()");
        System.out.println("Value of init parameter " +
            configFile());
        System.out.println("Exiting HibernatePlugIn.init()");
    }
    public String getConfigFile() {
        return name;
    }
    public void setConfigFile(String string) {
        configFile = string;
    }
}
```

实现 PlugIn 接口的类必须是实现以下两个方法：

init() 和 destroy()。在应用启动时 init() 被调用，关闭 destroy() 被调用。Struts 允许你传入初始参数给你的 PlugIn 类；为了传入参数你必须在 PlugIn 类里为每个参数创建一个类似 JavaBean 形式的 setter 方法。在 HibernatePlugIn 类里，欲传入 configFile 的名字而不是在应用里将它硬编码进去

2. 在 struts-config.xml 里面加入以下几行告知 Struts 这个新的 PlugIn

```
<struts-config>
    ...
    <!-- Message Resources -->
    <message-resources parameter="sample1.resources.ApplicationResources"/>
    <!-- Declare your plugins -->
    <plug-in className="com.sample.util.HibernatePlugIn">
        <set-property property="configFile" value="/hibernate.cfg.xml"/>
    </plug-in>
</struts-config>
```

ClassName 属性是实现 PlugIn 接口类的全名。为每一个初始化传入 PlugIn 类的初始化参数增加一个<set-property>元素。在这个例子里，传入 config 文档的名称，所以增加了一个 config 文档路径的<set-property>元素。

Tiles 和 Validator 框架都是利用 PlugIn 给初始化读入配置文件。另外两个你还可以在 PlugIn 类里做的事情是：

假如应用依赖于某配置文件，那么可以在 PlugIn 类里检查其可用性，假如配置文件不

可用则抛出 `ServletException`。这将导致 `ActionServlet` 不可用。

`PlugIn` 接口的 `init()` 方法是你改变 `ModuleConfig` 方法的最后机会, `ModuleConfig` 方法是描述基于 Struts 模型静态配置信息的集合。一旦 `PlugIn` 被处理完毕, Struts 就会将 `ModuleConfig` 冻结起来。

## 7.7.2. 请求是如何被处理的

`ActionServlet` 是 Struts 框架里唯一一个 `Servlet`, 它负责处理所有请求。它无论何时收到一个请求, 都会首先试着为现有请求找到一个子应用。一旦子应用被找到, 它会为其生成一个 `RequestProcessor` 对象, 并调用传入 `HttpServletRequest` 和 `HttpServletResponse` 为参数的 `process()` 方法。

大部分请处理都是在 `RequestProcessor.process()` 发生的。`Process()` 方法是以模板方法 (Template Method) 的设计模式来实现的, 其中有完成 `request` 处理的每个步骤的方法; 所有这些方法都从 `process()` 方法顺序调用。例如, 寻找当前请求的 `ActionForm` 类和检查当前用户是否有权限执行 `action mapping` 都有几个单独的方法。这给我们提供了极大的弹性空间。Struts 的 `RequestProcessor` 对每个请求处理步骤都提供了默认的实现方法。这意味着, 你可以重写你感兴趣的方法, 而其余剩下的保留默认实现。例如, Struts 默认调用 `request.isUserInRole()` 检查用户是否有权限执行当前的 `ActionMapping`, 但如果你需要从数据库中查找, 那么你要做的就是重写 `processRoles()` 方法, 并根据用户角色返回 `true` 或 `false`。

首先我们看 `process()` 方法的默认实现方式, 然后我将解释 `RequestProcessor` 类里的每个默认的方法, 以便你决定要修改请求处理的哪一部分。

---

```
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);
}
```

---

1、processMultipart()：在这个方法中，Struts 读取 request 以找出 contentType 是否为 multipart/form-data。假如是，则解析并将其打包成一个实现 HttpServletRequest 的包。当你生成一个放置数据的 HTML FORM 时，request 的 contentType 默认是 application/x-www-form-urlencoded。但是如果你的 form 的 input 类型是 FILE-type 允许

用户上载文件，那么你必须把 form 的 contentType 改为 multipart/form-data。如这样做，你永远不能通过 HttpServletRequest 的 getParameter() 来读取用户提交的 form 值；你必须以 InputStream 的形式读取 request，然后解析它得到值。

2、processPath(): 在这个方法中，Struts 将读取 request 的 URI 以判断用来得到 ActionMapping 元素的路径。

3、processLocale(): 在这个方法中，Struts 将得到当前 request 的 Locale; Locale 假如被配置，将作为 org.apache.struts.action.LOCALE 属性的值被存入 HttpSession。这个方法的附作用是 HttpSession 会被创建。假如你不想此事发生，可将在 struts-config.xml 文件里 ControllerConfig 的 local 属性设置为 false，如下：

```
<controller>          <set-property property="locale">
    value="false"/></controller>
```

4、processContent(): 通过调用 response.setContentType() 设置 response 的 contentType。这个方法首先会试着的得到配置在 struts-config.xml 里的 contentType。默认为 text/html，重写方法如下：

```
<controller>          <set-property property="contentType">
    value="text/plain"/></controller>
```

5、processNoCache(): Struts 将为每个 response 的设置以下三个 header，假如已在 struts 的 config.xml 将配置为 no-cache。

```
response.setHeader("Pragma", "No-cache"); response.setHeader("Cache-Control", "no-cache"); response.setDateHeader("Expires", 1);
```

假如你想设置为 no-cache header，在 struts-config.xml 中加如下几行

```
<controller>          <set-property property="noCache">
    value="true"/></controller>
```

6、processPreprocess(): 这是一个一般意义的预处理 hook，其可被子类重写。在 RequestProcessor 里的实现什么都没有做，总是返回 true。如此方法返回 false 会中断请求处理。

7、processMapping(): 这个方法会利用 path 信息找到 ActionMapping 对象。ActionMapping 对象在 struts-config.xml file 文件里表示为<action>

```
<action path="/newcontact" type="com.sample.NewContactAction">
    name="newContactForm" scope="request">          <forward name="sucess">
    path="/sucessPage.do"/>                      <forward name="failure">
    path="/failurePage.do"/></action>
```

ActionMapping 元素包含了如 Action 类的名称及在请求中用到的 ActionForm 的信息，另外还有配置在当前 ActionMapping 的里的 ActionForwards 信息。

8、processRoles(): Struts 的 web 应用安全提供了一个认证机制。这就是说，一旦用

户登录到容器，Struts 的 processRoles() 方法通过调用 request.isUserInRole() 可以检查他是否有权限执行给定的 ActionMapping。

```
<action path="/addUser" roles="administrator"/>
```

假如你有一个 AddUserAction，限制只有 administrator 权限的用户才能新添加用户。你所要做的就是在 AddUserAction 的 action 元素里添加一个值为 administrator 的 role 属性。

9、processActionForm(): 每个 ActionMapping 都有一个与它关联的 ActionForm 类。struts 在处理 ActionMapping 时，他会从<action>里 name 属性找到相关的 ActionForm 类的值。

```
<form-bean name="newContactForm"
type="org.apache.struts.action.DynaActionForm">
  <form-property type="java.lang.String" name="lastName"
name="firstName" type="java.lang.String"/>
</form-bean>
```

在这个例子里，首先会检查 org.apache.struts.action.DynaActionForm 类的对象是否在 request 范围内。如是，则使用它，否则创建一个新的对象并在 request 范围内设置它。

10、processPopulate(): 在这个方法里，Struts 将匹配的 request parameters 值填入 ActionForm 类的实例变量中。

11、processValidate(): Struts 将调用 ActionForm 的 validate() 方法。假如 validate() 返回 ActionErrors，Struts 将用户转到由<action>里的 input 属性标示的页面。

12、processForward() and processInclude(): 在这两个方法里，Struts 检查<action>元素的 forward 和 include 属性的值，假如有配置，则把 forward 和 include 请求放在配置的页面内。

```
<action forward="/Login.jsp" path="/loginInput"/>           <action
include="/Login.jsp" path="/loginInput"/>
```

你可以从他们的名字看出其不同之处。processForward() 调用 RequestDispatcher.forward()，processInclude() 调用 RequestDispatcher.include()。假如你同时配置了 forward 和 include 属性，Struts 总会调用 forward，因为 forward 是首先被处理的。

13、processActionCreate(): 这个方法从<action>的 type 属性得到 Action 类名，并创建返回它的实例。在这里例子中 struts 将创建一个 com.sample.NewContactAction 类的实例。

14、processActionPerform(): 这个方法调用 Action 类的 execute() 方法，其中有你写入的业务逻辑。

15、processForwardConfig(): Action 类的 execute() 将会返回一个 ActionForward 类型的对象，指出哪一页面将展示给用户。因此 Struts 将为这个页面创建 RequestDispatcher，然后再调用 RequestDispatcher. forward() 方法。

以上列出的方法解释了 RequestProcessor 在请求处理的每步默认实现及各个步骤执行的顺序。正如你所见，RequestProcessor 很有弹性，它允许你通过设置<controller>里的属性来配置它。例如，假如你的应用将生成 XML 内容而不是 HTML，你可以通过设置 controller 的某个属性来通知 Struts。

### 7.7.3. 创建你自己的 RequestProcessor

从以上内容我们已经明白了 RequestProcessor 的默认实现是怎样工作的，现在我将通过创建你自己的 RequestProcessor 展示一个怎样自定义 RequestProcessor 的例子。为了演示创建一个自定义 RequestProcessor，我将修改例子实现以下连个业务需求：

我们要创建一个 ContactImageAction 类，它将生成 images 而不是一般的 HTML 页面。

在处理这个请求之前，将通过检查 session 里的 userName 属性来确认用户是否登录。假如此属性没有被找到，则将用户转到登录页面。

分两步来实现以上连个业务需求。

创建你自己的 CustomRequestProcessor 类，它将继承 RequestProcessor 类，如下：

```
public class CustomRequestProcessor  
    extends RequestProcessor {  
  
    protected boolean processPreprocess (  
        HttpServletRequest request,  
        HttpServletResponse response) {  
        HttpSession session = request.getSession(false);  
        //If user is trying to access login page  
        // then don't check  
        if( request.getServletPath().equals("/loginInput.do")  
            || request.getServletPath().equals("/login.do") )  
            return true;  
        //Check if userName attribute is there is session.  
        //If so, it means user has allready logged in  
        if( session != null &&  
            session.getAttribute("userName") != null)  
            return true;  
        else{  
            try{  
                //If no redirect user to login Page  
                request.getRequestDispatcher  
                    ("/Login.jsp").forward(request, response);  
            } catch(Exception ex){  
            }  
        }  
        return false;  
    }  
}
```

```
}

protected void processContent(HttpServletRequest request,
    HttpServletResponse response) {
    //Check if user is requesting ContactImageAction
    // if yes then set image/gif as content type
    if( request.getServletPath().equals("/contactimage.do")){
        response.setContentType("image/gif");
        return;
    }
    super.processContent(request, response);
}
}
```

在 CustomRequestProcessor 类的 processPreprocess 方法里，检查 session 的 userName 属性，假如没有找到，将用户转到登录页面。

对于产生 images 作为 ContactImageAction 类的输出，必须要重写 processContent 方法。首先检查其 request 是否请求/contactimage 路径，如是则设置 contentType 为 image/gif；否则为 text/html。

加入以下几行代码到 struts-config.xml 文件里的<action-mapping>后面，告知 Struts，CustomRequestProcessor 应该被用作 RequestProcessor 类

```
<controller>
    <set-property property="processorClass"
        value="com.sample.util.CustomRequestProcessor"/>
</controller>
```

请注意，假如你只是很少生成 contentType 不是 text/html 输出的 Action 类，重写 processContent() 就没有问题。如不是这种情况，你必须创建一个 Struts 子系统来处理生成 image Action 的请求并设置 contentType 为 image/gif

Title 框架使用自己的 RequestProcessor 来装饰 Struts 生成的输出。

## 7.7.4. ActionServlet

假如你仔细研究 Struts web 应用的 web.xml 文件，它看上去像这样：

```
<web-app>
    <servlet>
        <servlet-name>action=</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <!-- All your init-params go here-->
    </servlet>
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

这就是说，ActionServlet 负责处理所有发向 Struts 的请求。你可以创建 ActionServlet

的一个子类，假如你想在应用启动和关闭时或每次请求时做某些事情。但是你必须在继承 ActionServlet 类前创建 PlugIn 或 RequestProcessor。在 Servlet 1.1 前，Title 框架是基于继承 ActionServlet 类来装饰一个生成的 response。但从 1.1 开始，就使用 TilesRequestProcessor 类。

### 7.7.5. 结论

开发你自己的 MVC 模型是一个很大的决心——你必须考虑开发和维护代码的时间和资源。Struts 是一个功能强大且稳定的框架，你可以修改它以使其满足你大部分的业务需求。

另一方面，也不要轻易地决定扩展 Struts。假如你在 RequestProcessor 里放入一些低效率的代码，这些代码将在每次请求时执行并大大地降低整个应用的效率。当然总有创建你自己的 MVC 框架比扩展 Struts 更好的情况。

## 7.8. Struts 应用国际化

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。

### 7.8.1. 本地化与国际化的概念

国际化（简称为 I18N）指的是软件设计阶段，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。

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。

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作出调整，使它适应新的语言和地区。

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关的格式化，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 作为那前的功能，那么当这个应用需要支持新的 Locale 时，开发人员必需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，就必需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。

### 7.8.2. Web 应用的中文本地化

无论时对 Web 应用的本地化还是国际化，都会涉及字符编码转换问题。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。

### 7.8.2.1. 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");
```

### 7.8.2.2. 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”，那么必需先对来自数据库的数据进行编码转换，然后才能使用。

### 7.8.2.3. 处理 XML 配置文件编码

如果在 XML 文件中包含中文，可以将 XML 文件的字符编码为“GB2312”。这样，当 Java 程序加载和解析 XML 文件时无需再进行编码转换。

```
<?xml version='1.0' encoding="GB2312"?>
```

### 7.8.2.4. 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>
```

### 7.8.3. Java 对 I18N 的支持

Java 在其核心库中提供了支持 I18N 的类和接口。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。

#### 7.8.3.1. Locale 类

java.util.Locale 类时最重要的 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。

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

行相关的格式化。

Locale 对象本身不执行和 I18N 相关的格式化或解析工作。Locale 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 Locale 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 Locale 对象时，需要明确地指定其语言和国家代码。如：

```
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，他们代表一些常用的 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");
```

#### 7.8.4. 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；而对于特定的中断用户，Web 容器会从 HTTP 请求中获取 Locale 信息。

#### 7.8.5. 在 Web 应用中访问 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。

#### 7.8.6. 在 Struts 应用中访问 Locale 对象

有序 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。

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 范围中必需满足以下条件：

- 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。

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。

Action 类的 getLocale() 方法调用 RequestUtils.getUserLocale() 方法。

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() 方法来获取 Locale 对象。

### 7.8.7. 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。

### 7.8.8. MessageFormat 类和符合消息

Java 的 ResourceBundle 和 Struts 的 MessageResources 类都允许使用静态和动态的文本。静态文本指定的是实现就已经具有明确内容的文本。动态文本指的是只有在运行时才能确定内容的文本。

通常把包含可变数据的消息成为符合消息。符合消息允许在程序运行时把动态数据加入到消息文本中。这样能够减少 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 中使用符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，往往要对语言做适当调整。

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。

### 7.8.9. 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与用户 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。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。

### 7.8.9.1. 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。

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，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\_CH.properties 或 application\_ch.properties。

当 Struts 框架处理 Locale 为中文的用户请求时，Struts 框架首先在 WEB-INF/classes/ 目录下寻找 application\_ch\_CH.properties 文件，如果存在该文件，就从该文件中获取文本消息，否则再一次寻找 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。

### 7.8.9.2. 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。

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的 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。

### 7.8.9.3. 通过编程来访问 Resource Bundle

在 Action 积累中定义了 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 配置文件中的<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。

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：

- 根据参数置顶的 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：

```
getMessaeg(java.lang.String key)
```

#### 7.8.9.4. 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。

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。

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 参数用户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。

- Struts 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。

<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。

- 在 Validator 验证框架中访问 Resource Bundle。
- 在声明型异常处理中访问 Resource Bundle。

#### 7.8.10. 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。

#### 7.8.11. 小结

本小节介绍了软件的本地化与国际化的概念，然后消息介绍了对 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 框架的初始化时, 把 Resource Bundle (即 MessageResources 对象) 存储在 application 范围内; 在响应用户请求时, 把包装用户 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.setCharacterEncoding(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。

## 7.9. Validator 验证框架

Struts 提供了一个名为 Validator 的优秀组件。Validator 可以插入 Struts 应用程序, 甚至直接和最新的 Struts 发行版绑定在一起。只要几个 JAR 文件就可以了。但是 Validator 强在什么地方呢? 为什么要用它代替 JavaScript 呢?

而且, 您应当认识到, Validator 的大部分执行都使用 JavaScript。所以实际上并没有离开 JavaScript, 而且得到的客户端验证也是 JavaScript 擅长的内容。但是, Validator 消除了 JavaScript 的许多问题。首先, 它是由成千上万的 Struts 开发人员和用户编码、测试和调试过的, 因此降低了您需要进行的测试数量。(Validator 只是降低了测试负担, 但并没有完全消除它。) 另外, Validator 提供了大量常用验证函数, 所以您不必为电子邮件地址、电话号码、邮编以及其他常用数据编写验证器。

而最重要的, 可能是 Struts Validator 主要通过配置文件工作, 而不用内联的 HTML 代码。通过简单的 XML 文件, 可以指示要验证哪个字段, 要执行哪类验证。Struts 和 Validator 负责把配置变成工作的 JavaScript 代码, 您这一边不需要做任何额外工作! 虽然偶尔也要为特定于应用程序的数据添加新的验证函数, 但是在 HTML 中使用这些函数的工作由 Struts 处理——不需要手工过程。这就是 Validator 真正胜出而珍贵的地方。

Validator 框架如今成为 Jakarta 的公共项目的一部分, Struts 携带了 Validator 框架, 主要是两个 jar 文件, jakarta-oto.jar 和 commons-validator.jar 文件。前者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, 后者提供了一个简单的, 可扩展的验证框架, 包含了通用的验证方法和验证规则。如果 Struts 使用 Validator 需要添加那两个 jar 文件到 lib 目录。Validator 框架采用了两个基于 XML 的配置文件来配置验证规则, 这两个文件是 validator-rules.xml 和 validation。在 Struts 应用中, 它们放在 WEB-INF 目录。

Validator-rules.xml 文件包含了一组通用的验证规则, 对所有的 String 都适用。一般的情况下没必要修改, 除非需要修改或者扩展默认的规则。建议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, 这样当升级 Validator 框架版本的时候, 也无需修改 validator-rules.xml 文件。因为这个 XML 是不推荐修改的, 所以我只是简单的介绍一下具体的<validator>元素的几个属性, 至于具体需要开发人员来开发的是 validation 的 XML 文件。<validator>元素的 name 属性指定验证规则的逻辑名, 这个名字必须唯一。还有

classname 和 method 属性分别指实现验证规则逻辑的类和方法. msg 属性指定来自 Resource 中的消息 key, 验证失败, 根据这个消息 Key 去找匹配的消息文本. depends 属性指定在调用当前验证规则之前必须调用其他验证规则。

Validation.xml 文件是针对某个具体的 Struts 应用, 需要开发人员来创建的, 它可以为应用中的 ActionForm 配置所需要的验证规则, 取代 ActionForm 类中编程的方式来实现验证. <form-validation>元素是 validation.xml 文件的根元素, 它包含两个子元素<global>和<formset>, <global>可以出现零次或者多次, 而<formset>元素可以出现一次或者多次他们的 DTD 定义是:<!element form-validation(global\*, formset+)>. <global>元素可以定义<constant>子元素, 它用来定义常量表达式, 在文件中其余地方可以应用这些常量表达式. <formset>元素包含两个子元素<constant>和<form>, <form>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>包含一个或者多<field>子元素, 用于配置表单中字段的验证规则. <field>的<msg>子元素指定验证规则对应的消息文本, 该消息文本替代 validator-rules.xml 文件中为验证规则配置的默认的消息文本. <msg>元素的主要属性是:name 指定验证规则的名字, key 当 resource 属性为 true 的时候, key 属性指定的消息 key 该消息 key 应该在 Resource Bundle 中存在. key 属性直接指定消息文本. resource 当此项为 true 的时候, 表示使用来自 ResourceBundle 的消息. <field>元素还可以包括附加的子元素<arg0><arg1><arg2>用于替换符合消息中的参数. <field>元素元素还可以包含零个或者多个<var>元素, 用来向验证规则传递参数。

Validator 插件需要在 Struts 配置文件中配置 ValidatorPlugIn 插件如下:

```
<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, 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。

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 的两个子类, validator 包 DynaValidatorForm 和 ValidatorForm 类. DynaValidatorForm 支持动态的 ActionForm 中使用 Validator 框架, ValidatorForm 支持在标准的 ActionForm 中使用 validator 框架. 无论是动态的还是标准的, 它们的配置方法都一样. 而且 Struts 的一些标签也可以和 Validator 框架协同工作, 例如<html:errors>和<html:messages>Validator 框架提供了对 I18N 的支持, 无论是客户端验证, 还是服务器验证. Validator 框架都从应用 ResourceBundle 中获取错误信息. 所以说它很好的支持的国际化. 总结 Struts 应用中使用 validator 框架的步骤:

创建扩展 ValidatorForm 或者 ValidatorActionForm 类的 ActionForm 类, 如果使用动态的 ActionForm, 则无需创建扩展 DynaValidatorForm 或者 DynaValidatorActionForm 类的子类, 可以直接进行。

- 在 Struts 配置文件中配置<form>和<action>元素.
- 把 Validator 框架使用的消息文本添加到应用的 ResourceBundle 中.
- 在 Validation.xml 文件中为表单配置验证规则.
- 在 Struts 配置文件中配置 ValidatorPlugIn 插件.

## 7.10. 异常处理

总的来说, 在 struts 新的版本中加入了对异常的处理, 称之为: Exception Handling,

标志着作为一个整体的解决框架，struts 越来越趋于成熟。

通常来说，以前在用 struts 开发的过程中，对于异常的处理，主要是采用手动处理的方式：如通过 try/catch 等等捕获异常，然后定制个性化的比较详细的错误信息放进 ActionError 中，然后在具体的返回页面中把这些错误信息反馈给用户（包括开发员）。异常原始的信息不管是最终用户还是开发员都是不希望看到的。

下面着重讲在 struts 中是如何通过配置文件来解决异常。

Struts 中的 Exception Handleing 不难，简单高效是业内给其的一个比较好的评价。

通过配置文件（主要是 struts-config.xml）来定制异常处理，就象定义 formbean 一样，定制异常也有两种方法，姑且把它分为：“全局异常”和“局部异常”。

全局异常，定义方法如下：

```
....  
....  
<global-exceptions>  
    <exception key="expired.InvalidItemsCatalogName"  
type="com.iplateau.jshop.common.waf.exceptions.InvalidItemsCatalogNameException"  
        scope="request"  
        path="error.jsp"/>  
</global-exceptions>  
....  
....
```

上述代码在 struts-config.xml 中定义了一个全局异常，它的作用是抛出 InvalidItemsCatalogNameException（本处的意思是当在添加商品分类的时候发现该类别已经存在）异常的时候返回到 error.jsp 中，并且携带自定的比较规范的异常信息 expired.InvalidItemsCatalogName， expired.InvalidItemsCatalogName 可以在应用程序的资源配置文件中找到，如：

expired.InvalidItemsCatalogName=你要添加的商品类别已经存在，请添加新的类别！

局部异常，定义方法如下：

```
....  
....  
<action-mappings>  
    <action path="/addItemsCatalogAction"  
        type="com.iplateau.jshop.action.ItemsCatalogAction"  
        name="itemsCatalogForm">  
        <exception key="expired.InvalidItemsCatalogName"  
type="com.iplateau.jshop.common.waf.exceptions.InvalidItemsCatalogNameException"  
            path="/error.jsp"/>
```

```
<forward name="success" path="***Layout(此处采用 Tiles 进行辅助开发)" />
</action>
</action-mappings>
.....
.....
(关于 Tiles 的内容参看我的另一篇文章“Struts 使用 Tiles 辅助开发”)
```

下面我们把关注的目光放在具体 action 里，看看 struts 是如何进行异常处理的

ItemsCatalogAction.java

```
package com.iplateau.jshop.action.ItemsCatalogAction;

import com.iplateau.jshop.business.ItemsCatalogMap;
import com.iplateau.jshop.action.ItemsCatalogForm;
import ***;

public class ItemsCatalogAction extends BaseAction {
    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        String flg="error";
        String act=request.getParameter("act");
        ItemsCatalogMap map=new ItemsCatalogMap();
        ItemsCatalogForm thisform=(ItemsCatalogForm)form;
        // ItemsCatalogMap 为具体的处理商品类别的类
        if(act.equals("create")){
            map.validateNameOfCatalog(thisform);
            map.create(thisform);
            flg="success";
        }
        return mapping.findForward(flg);
    }
}
```

上便是一个简单的 action, 其中用 act 的具体内容来判断此时的 action 要处理的操作, 如 act 为 create 的时候, 处理的是商品类别的添加等等。

看了上边的代码, 可以了解, 在调用具体的业务类进行添加商品类别以前先要判断此时要添加的商品类别的合法性, 如:

调用 map.validateNameOfCatalog(thisform);  
其中在 ItemsCatalogMap 定义的 validateNameOfCatalog 方法抛出  
InvalidItemsCatalogNameException。

下面让我们运行以下添加商品类别的例子, 如果此时数据库中已经存在“牛奶制品”这个商品类别, 而我们又要添加此类别的时候, 程序如我们预想的一样转到 error.jsp 并且显示了我们要先是的错误信息: 你要添加的商品类别已经存在, 请添加新的类别!

其实就这么简单, 我们的定制的异常在程序中并不需要用 try/catch 来捕获, 一旦出现了我们已经定义的异常那么就会转到相应得页面, 并且携带定制的信息。

还记得在 struts 先前的版本中我们的请求都是通过 action 的 perform 来处理, 可是现在都要通过 execute 来处理, 就其原因一个很重要的就是“成全” Exception Handling。为什么呢?

因为 perform 在声明的时候仅仅抛出 IOException 和 ServletException, 这远远不能满足 Exception Handling 的要求, 那么让我们看看 execute 是怎样的: 它笼统的抛出 Exception (所有异常的父类)

上面的阐述只是一个引子, 而且默认 struts 的异常是通过 org.apache.struts.action.ExceptionHandler 来处理的, 你可以定义自己的处理方式, 只要继承它并实现其中的 execute 方法, 这个方法在 ExceptionHandler 的定义如下:

```
public ActionForward execute(Exception ex,  
                           ExceptionConfig ae,  
                           ActionMapping mapping,  
                           ActionForm formInstance,  
                           HttpServletRequest request,  
                           HttpServletResponse response)  
throws ServletException
```

## 7.11. Struts 标签

Struts 框架提供了一系列的框架组件, 同时, 他也提供了一系列的标签(Tag)用于和框架进行交互。Struts 提供的标签包含在以下四个标签库(Tag libraries)中:

- HTML
- Bean
- Logic
- Template

这四个标签库所包含的标签功能各自截然不同, 从标签库的名字我们可以看出其功能, 如, HTML 标签库是用来包装 HTML 控件的。

### 7.11.1. 在 Struts 应用中使用标签库

和使用其它标签库一样，使用 Struts 提供的标签库只需要简单的两步：

1、在 web.xml 中声明标签库：

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

2、在 JSP 页面中引入标签库：

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

### 7.11.2. Struts HTML 标签库

HTML 标签库中的标签列表	
标签名	描述
base	包装 HTML 的 base 元素
button	包装 HTML 的 button 类型的 input 元素
cancel	包装 HTML cancel 按钮
checkbox	包装 HTML checkbox 类型的输入域
errors	有条件地显示一些 error 消息，显示 ActionErrors 信息
file	包装 HTML 文件上传输入域
form	定义 HTML form 元素
frame	包装 HTML frame 元素
hidden	包装 HTML hidden 输入域
html	包装 HTML 中的 html 元素
image	包装 "image"类型的输入域
img	包装 HTML 的 img 元素
javascript	包装根据 ValidatorPlugIn 提供的校验规则所提供的 javascript 校验脚本
link	包装超链接
messages	有条件地显示一些提示信息，显示 ActionMessages 信息
multibox	包装多选输入框
option	包装一个选择输入框
options	包装一批选择输入框
optionsCollection	包装一批选择输入框集

password	包装密文输入框
radio	包装单选输入框
reset	包装“重置”功能的按钮
rewrite	包装一个 URL
select	包装一个选择输入框
submit	包装一个提交按钮
text	包装一个文本输入框
textarea	包装一个备注输入框

在这里，不打算对每一个标签的使用进行详细说明，要想了解每一个标签的使用，请查看 Struts 官方文档。

接下来，我们着重学习以下几个非常重要的标签的使用，举一反三，通过这几个标签的使用，我想，即使不去看官方文档，也能够对其它标签的使用有个基本的了解。

### 7.11.2.1. form 标签

Struts 的 form 标签是最重要的标签之一，他包装了 HTML 的标准 form 标签，提供了将 HTML form 和 ActionForm 连接起来的功能。

HTML form 中的每一个域对应 ActionForm 的一个属性，当提交 HTML from 后，Struts 根据匹配关系，将 HTML from 域的值赋给 ActionForm 的同名属性。下表列举了 form 标签的属性，并且针对每一个属性加以详细说明：

Struts form 标签属性列表	
Name	Description
action	form 提交的目标地址，action 用来选择一个 Struts Action 对提交后的客户请求进行处理。通过和 action 的 path 属性进行匹配来选择 Struts Action。 如果在 web.xml 中设置的 servlet 映射是扩展名映射，则可以用以下方式进行 Action 匹配(扩展名可以不作为匹配内容): <html:form action="login.do" focus="accessNumber"> 上面的语句表示 form 提交后，交给 path 属性值为 login 的 Action 进行处理 如果在 web.xml 中设置的 servlet 映射是路径映射，则 action 的值必须完全匹配 Struts Action 的 path 属性值，例如： <html:form action="login" focus="accessNumber">
enctype	提交 form 使用的编码方式
focus	页面初始化时光标定位的输入控件名
method	提交请求的 HTTP 方式('POST' or 'GET')
name	对应的 ActionForm 的名字，如果不指定，则使用 Struts Action 在配置文件中 name 属性所指定的 ActionForm。
onreset	当 form 重置时执行的 Javascript
onsubmit	当 form 提交时执行的 javascript
scope	与 form 对应的 ActionForm 的有效区域，可以为 request 或 session
style	CSS 样式表 The CSS styles to be applied to this HTML element.
styleClass	CSS 样式类别
styleId	HTML 元素的 ID
target	form 提交的目标 frame
type	form 对应的 ActionForm 的类名

### 7.11.3. Struts Bean 标签库

Struts Bean 标签库中的标签列表	
Tag name	Description
cookie	Define a scripting variable based on the value(s) of the specified request cookie.
define	Define a scripting variable based on the value(s) of the specified bean property.
header	Define a scripting variable based on the value(s) of the specified request header.
include	Load the response from a dynamic application request and make it available as a bean.
message	Render an internationalized message string to the response.
page	Expose a specified item from the page context as a bean.
parameter	Define a scripting variable based on the value(s) of the specified request parameter.
resource	Load a web application resource and make it available as a bean.
size	Define a bean containing the number of elements in a Collection or Map.
struts	Expose a named Struts internal configuration object as a bean.
write	Render the value of the specified bean property.

### 7.11.4. Struts Logic 标签库

Struts Logic 标签库中包含的标签列表	
Tag name	Description
empty	如果标签 parameter, propertie 等属性所指定的变量值为 null 或空字符串，则处理标签包含的内容
equal	如果标签 parameter, propertie 等属性所指定的变量的值等于标签 value 属性所指定的值，则处理标签所包含的内容，如： <pre>&lt;logic:equal      value="modify"      property="action" name="projectForm"&gt; &lt;bean:message key="project.project_modify"/&gt; &lt;/logic:equal&gt;</pre> 上面的示例表示，如果 projectForm 的 action 属性等于 modify，则处理<bean:message key="project.project_modify"/>语句。
forward	Forward control to the page specified by the ActionForward entry.
greaterEqual	Evaluate the nested body content of this tag if the requested variable is greater than or equal to the specified value.
greaterThan	Evaluate the nested body content of this tag if the requested variable is greater than the specified value.
iterate	Repeat the nested body content of this tag over a specified collection.

lessEqual	Evaluate the nested body content of this tag if the requested variable is less than or equal to the specified value.
lessThan	Evaluate the nested body content of this tag if the requested variable is less than the specified value.
match	Evaluate the nested body content of this tag if the specified value is an appropriate substring of the requested variable.
messagesNotPresent	Generate the nested body content of this tag if the specified message is not present in this request.
messagesPresent	Generate the nested body content of this tag if the specified message is present in this request.
notEmpty	Evaluate the nested body content of this tag if the requested variable is neither null nor an empty string.
notEqual	Evaluate the nested body content of this tag if the requested variable is not equal to the specified value.
notMatch	Evaluate the nested body content of this tag if the specified value is not an appropriate substring of the requested variable.
notPresent	Generate the nested body content of this tag if the specified value is not present in this request.
present	Generate the nested body content of this tag if the specified value is present in this request.
redirect	Render an HTTP redirect.

执行比较功能的标签通用属性表	
Name	Description
name	The name of a bean to use to compare against the value attribute. If the property attribute is used, the value is compared against the property of the bean, instead of the bean itself.
parameter	The name of a request parameter to compare the value attribute against.
property	The variable to be compared is the property (of the bean specified by the name attribute) specified by this attribute. The property reference can be simple, nested, and/or indexed.
scope	The scope within which to search for the bean named by the name attribute. All scopes will be searched if not specified.
value	The constant value to which the variable, specified by another attribute(s) of this tag, will be compared.

示例：

To check whether a particular request parameter is present, you can use the Logic present tag:

```
<logic:present parameter="id">
    <!-- Print out the request parameter id value -->
</logic:present>
```

To check whether a collection is empty before iterating over it, you can use the notEmpty tag:

```
<logic:notEmpty name="userSummary" property="addresses">
    <!-- Iterate and print out the user's addresses -->
```

```
</logic:notEmpty>
```

Finally, here's how to compare a number value against a property within an ActionForm:

```
<logic:lessThan property="age" value="21">
  <!-- Display a message about the user's age -->
</logic:lessThan>
```

### 7.11.5. Struts Template 标签库

Struts Template 标签库中的标签列表	
Tag name	Description
insert	Retrieve (or include) the specified template file, and then insert the specified content into the template's layout. By changing the layout defined in the template file, any other file that inserts the template will automatically use the new layout.
put	Create a request-scope bean that specifies the content to be used by the get tag. Content can be printed directly or included from a JSP or HTML file.
get	Retrieve content from a request-scope bean, for use in the template layout.

### 7.11.6. 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，还有一部分标签在特定的级别提供和其他 Struts 标签库的标签相同的功能。其中最主要的两个标签为：“<nested:root>”和“<nested:nest>”。

1. <nested:nest>：定义一个新的嵌套级别。<nested:nest> 标签可以表达 JavaBean 之间的嵌套关系。<html:form action="/showPerson">< nested:nest property="person"> LastName: < nested:text property="lastName"> </nested:nest> </html:form >以上的<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 。

2. <nested:root>：用来显示的指定顶级级别的 JavaBean。<nest: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。<nested:root name="PersonForm"><nested:nest property="person"><nested:write property="lastName"/></nested:nest></nested:root>以上代码中，<nested:root>标签的 name 属性“PersonFrom”，代表当前的 PersonFormBean。 嵌套其中的<nested:nest>标签的 property 属性为“person”，代表 PersonForm Bean 的 person 属性。

## 7.12. 使用 Tiles 框架

Struts1.1 以后增加了 Tiles 包使得 struts 在页面的处理方面多了一种选择。并且更容易实现代码的重用。

Tiles 中对页面的划分有点象 jakarta 的另外个项目 Turbine 中的 TDK。增加了 layout

的概念。其实就是把一个页面划分为几块。通常的来说一个页面大概可以划分为如下几块：

- head 页面头部：存放一个运用的公共信息：logo 等，如果是网站可能是最上面的一块。
- menu 页面菜单：放置一个运用中需要使用的菜单，或者在每一个页面都使用的连接。
- footer 页面尾部：如版权信息等。
- body 页面主题内容：每个页面相对独立的内容。

如果按上面的划分那对每一个页面我们只要写 body 里面的内容，其他的就可以共享重用。

如果大多数页面的布局基本相同我们甚至可以使用一个 jsp 文件根据不同的参数调用不同的 body。

### 7.12.1. Tiles 配置和基本配置文件介绍

Tiles 有一个配置文件：tiles-defs.xml

tiles-defs.xml 定义了每一个页面的组成元素和形式。

下面我将说明如下所示的一个 tiles-defs.xml 文件

.tiles-defs.xml

```
<!-- 定义 site.mainLayout -->
<definition name=site.mainLayout path=/layouts/classicLayout.jsp>
    <put name=title value=Tiles Blank Site />
    <put name=header value=/tiles/common/header.jsp />
    <put name=menu value=site.menu.bar />
    <!-- menu 的组成为 site.menu.bar 对应的页面 -->
    <put name=footer value=/tiles/common/footer.jsp />
    <put name=body value=/tiles/body.jsp />
</definition>

<!-- 定义 site.index.page，继承 site.mainLayout -->
<definition name=site.index.page extends=site.mainLayout>
    <put name=title value=Tiles Blank Site Index />
    <put name=body value=/tiles/body.jsp />
    <!-- 以上两个元素将替换 site.mainLayout 中的元素 -->
</definition>

<definition name=site.menu.bar path=/layouts/vboxLayout.jsp>
    <putList name=list>
        <add value=site.menu.links />
        <add value=site.menu.documentation />
    </putList>
</definition>
```

</tiles-definitions>

附：/layouts/classicLayout.jsp

```
<html>
<head>
<title><tiles:getAsString name=title/>
</title>
</head>
<body bgcolor=#ffffff text=#000000 link=#023264 alink=#023264 vlink=#023264>
<table border=0 width=100% cellspacing=5>
<tr>
<td colspan=2><tiles:insert attribute=header /></td>
</tr>
<tr>
<td width=140 valign=top>
<tiles:insert attribute='menu' />
</td>
<td valign=top align=left>
<tiles:insert attribute='body' />
</td>
</tr>
<tr>
<td colspan=2>
<tiles:insert attribute=footer />
</td>
</tr>
</table>
</body>
</html>
```

在 web.xml 里面配置 tiles, 配置完后对应 struts action servlet 的配置如下:

web.xml

```
-----
<!-- Action Servlet Configuration -->
<servlet>
<servlet-name>action</servlet-name>
<!-- Specify servlet class to use:
- Struts1.0.x: ActionComponentServlet
- Struts1.1: ActionServlet
- no Struts: TilesServlet
-->
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
<!-- Tiles Servlet parameter
Specify configuration file names. There can be several comma
separated file names
-->
<init-param>
<param-name>definitions-config</param-name>
```

```
<param-value>/WEB-INF/tiles-defs.xml</param-value>
</init-param>
<!-- Tiles Servlet parameter
Specify Tiles debug level.
O : no debug information
1 : debug information
2 : more debug information
--&gt;
&lt;init-param&gt;
&lt;param-name&gt;definitions-debug&lt;/param-name&gt;
&lt;param-value&gt;1&lt;/param-value&gt;
&lt;/init-param&gt;
<!-- Tiles Servlet parameter
Specify Digester debug level. This value is passed to Digester
O : no debug information
1 : debug information
2 : more debug information
--&gt;
&lt;init-param&gt;
&lt;param-name&gt;definitions-parser-details&lt;/param-name&gt;
&lt;param-value&gt;0&lt;/param-value&gt;
&lt;/init-param&gt;
<!-- Tiles Servlet parameter
Specify if xml parser should validate the Tiles configuration file.
true : validate. DTD should be specified in file header.
false : no validation
--&gt;
&lt;init-param&gt;
&lt;param-name&gt;definitions-parser-validate&lt;/param-name&gt;
&lt;param-value&gt;true&lt;/param-value&gt;
&lt;/init-param&gt;
<!-- Struts configuration, if Struts is used --&gt;
&lt;init-param&gt;
&lt;param-name&gt;config&lt;/param-name&gt;
&lt;param-value&gt;/WEB-INF/struts-config.xml&lt;/param-value&gt;
&lt;/init-param&gt;
&lt;init-param&gt;
&lt;param-name&gt;validate&lt;/param-name&gt;
&lt;param-value&gt;true&lt;/param-value&gt;
&lt;/init-param&gt;
&lt;init-param&gt;
&lt;param-name&gt;debug&lt;/param-name&gt;
&lt;param-value&gt;2&lt;/param-value&gt;
&lt;/init-param&gt;</pre>
```

```
<init-param>
<param-name>detail</param-name>
<param-value>2</param-value>
</init-param>
<load-on-startup>2</load-on-startup>
</servlet>
```

## 7.12.2. 使用 Tiles

如果已经配置好 tiles-defs.xml，接下来就可以在 jsp 文件中使用这些定义了。

有如下的方式使用 tiles

```
<tiles:insert definition=site.mainLayout flush=true />
    插入 site.mainLayout 标记的一页
<tiles:insert template=/tutorial/basic/myFramesetLayout.jsp >
<tiles:put name=title content=My first frameset page direct=true />
<tiles:put name=header content=/tutorial/common/header.jsp direct=true/>
<tiles:put name=footer content=/tutorial/common/footer.jsp direct=true/>
<tiles:put name=menu content=/tutorial/basic/menu.jsp direct=true/>
<tiles:put name=body content=/tutorial/basic/helloBody.jsp direct=true/>
</tiles:insert>
/tutorial/basic/myFramesetLayout.jsp
-----
<html>
<head>
<title><tiles:get name=title/></title>
</head>
<frameset rows=73, *, 73>
<frame src=<%=request.getContextPath()%><tiles:get name=header /> name=header >
<frame src=<%=request.getContextPath()%><tiles:get name=body /> name=body >
<frame src=<%=request.getContextPath()%><tiles:get name=footer /> name=footer >
</frameset>

</html>
    插入/tutorial/basic/myFramesetLayout.jsp
    并把 title 的值设定为:My first frameset page
    header 设定为/tutorial/common/header.jsp
```

## 7.12.3. 小结

Tiles 的使用在他的文档里面写的比较详细。以上是一些简单和基本的使用。具体的文档可以看 Struts 里面的一个 tiles-documentation.war 的包。但即使是这个包也不是很全。可以通过上面的连接到作者的主页上去找。

个人觉得使用 Tiles 在做企业运用的时候可能不如在做网站那样更能体现优越性。但在系统开始设计的时候考虑并规划好整个 UI，那在修改和维护的时候将节省不少的工作量，因为通常 UI 的确定在代码编写结束和完成，所有尽可能的使用多个子页面构成一个页面，后面的美化和维护就比直接维护一个很大的页面容易。

## 7.13. 用 StrutsTestCase 测试 Struts 应用

一个具有良好系统架构的 J2EE 应用程序至少有三层组成, 即表现层, 商业层和系统集成层(包括数据存取以及和其他系统集成), 目前, Struts 是应用比较广泛, 实现 MVC2 模式应用于表现层的一种技术. 在这里面, Struts Action 主要用来完成一些简单的数据校验, 转换, 以及流程转发控制(注意:这里流程不是业务规则). 因此在对整个应用程序进行测试时, 我们同时也要测试 Struts Action.

但是, 测试 Struts Action 相对测试简单的 JavaBean 是比较困难, 因为 Struts 是运行在 Web 服务器中, 因此要测试 Struts Action 就必须发布应用程序然后才能测试. 我们想象一下, 对于一个拥有上千个 JSP page 和数百甚至数千 Java Classes 的大规模应用程序, 要把他们发布到诸如 Weblogic 之类的应用服务器再测试, 需要多少的时间和硬件资源? 所以这种模式的测试是非常费时费力的。

所以, 如果有一种办法能够不用发布应用程序, 不需要 Web 服务器就能象测试普通 Java Class 一样测试 Struts Action, 那就能极大地加强 Struts 的可测试性能, 使应用程序测试更为容易, 简单快速. 现在这个工具来了, 这就是 StrutsTestCase.

StrutsTestCase 是一个功能强大且容易使用的 Struts Action 开源测试工具, 它本身就是在大名鼎鼎的 JUnit 基础上发展起来的。因此通过和 JUnit 结合使用能极大加强应用程序的测试并加快应用程序的开发。

StrutsTestCase 提供了两者测试方式, 模仿方式和容器测试方式. 所谓模仿方式就是有 StrutsTestCase 本身来模拟 Web 服务器. 而容器测试方式则需要 Web 服务器. 本文要讨论的是前者, 原因很简单, 不需要 Web 服务器就能象测试普通的 Java Class 一样测试 Struts Action.

### 7.13.1. 准备 StrutsTestCase

StrutsTestCase 是一个开源工具, 可以到 <http://strutstestcase.sourceforge.net> 下载. 如果你使用 Servlet2.3 就下载 StrutsTestCase213-2.3.jar, 使用 Servlet2.4 的就下载 StrutsTestCase213-2.4.jar. 另外 StrutsTestCase 本身就是从 JUnit 继承的, 所以你还需要下载 JUnit3.8.1。

在本文中, 我们用一个简单的例子来做测试. 假设我们有一张表 Hotline(country varchar2(50), pno varchar2(50)), 我们要做的是根据输入条件从这张表检索相应的记录. 检索条件是 country.

Value Object:

```
package sample;

public class HotlineDTO implements Serializable{
    private String country = "";
    private String pno = "";
    /**
     * Method HotlineActionForm
     *
     */
    public HotlineDTO () {
        super();
    }
}
```

```
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public void setPno(String pno) {
        this.pno = pno;
    }
    public String getCountry() {
        return (this.country);
    }
    public String getPno() {
        return (this.pno);
    }
}
```

ActionForm:

```
package sample;
import org.apache.struts.action.ActionForm;
public class HotlineActionForm extends ActionForm{
    private String country = "";
    private String pno = "";
    /**
     * Method HotlineActionForm
     *
     *
     */
    public HotlineActionForm() {
        super();
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public void setPno(String pno) {
        this.pno = pno;
    }
    public String getCountry() {
        return (this.country);
    }
    public String getPno() {
        return (this.pno);
    }
}
```

Action Class:

```
public class SearchHotlineAction extends Action {  
    public ActionForward execute(ActionMapping mapping, ActionForm form,  
HttpServletRequest request,  
    HttpServletResponse response) throws Exception {  
        String target = "";  
        try{  
            //调用 HotlineDAO 检索 hotline  
            String country=((HotlineActionForm)form).getCountry();  
            List hotlineList = HotlineDAO.getHotlineList(country);  
            if(hotlineList!=null && hotlineList.size()>0){  
                request.setAttribute("hotlineList",hotlineList);  
                target = "hotlineList";  
            }else{  
                target = "notfound";  
            }  
        }catch(Exception ex){  
            ....  
        }  
    }  
}
```

Struts Config:

```
<struts-config>  
  
<form-beans>  
    <form-bean name="hotlineActionForm" type="sample.HotlineActionForm" />  
    .....  
</form-beans>  
<action-mappings>  
    <action path="/SearchHotline"  
        name="hotlineActionForm"  
        type="sample.SearchHotlineAction "  
        scope="request"  
        validate="false">  
        <forward name="hotlineList" path="/hotlineList.jsp"/>  
        <forward name="notfound" path="/searchHotline.jsp"/>  
    </action>  
    .....  
<action-mappings>  
    .....  
<struts-config>
```

### 7.13.2. 初试 StrutsTestCase

当采用模拟方式时,所有的 StrutsTestCase 测试 Class 都是从 MockStrutsTestCase 继承下来的.

下面我们就创建一个最简单的测试 Class.

```
public class SearchHotlineAction extends MockStrutsTestCase {  
  
    public void setUp() throws Exception{  
    }  
  
    public void tearDown() throws Exception{  
    }  
  
    public void testSearchHotline() throws Exception{  
        setRequestPathInfo("/SearchHotline.do");  
        addRequestParameter("country", "CN");  
        actionPerform();  
    }  
}
```

上面的 Class 相信用过 JUnit 的朋友都很熟悉.

好了,一个简单的测试例子就完成了,如果你用的是 Eclipse 就选择 Run->Run...->JUnit->New 就可以直接运行.不需要发布你的程序,不需要任何的 Web 服务器支持,就可以测试 Struts Action,这就是 StrutsTestCase 带来的好处.下面简单地介绍它是怎么工作的。

在上面的例子中,我们调用 setRequestPathInfo()告诉StrutsTestCase我们要模拟 JSP 调用 SearchHotline.do 这个 Action,并且调用 addRequestParameter()增加了一个参数 country.最后调用 actionPerform()运行.

看到这里,大家发现一个问题没有?在上面 Action 的源代码里我们是通过

```
String country=((HotlineActionForm) form).getCountry();
```

也就是 ActionForm 来取得输入的参数值,可我们在 testSearchHotline() 方法里并没有设置 ActionForm

那么它是怎么出来的呢?其实大家如果熟悉 Struts 的运行流程的话就知道, JSP 接受用户的输入并发送请求时都是类似这样的 http://hostname/servletName?param1=value1&m2=value2. 只是 Struts 接受到这些参数后再根据 Struts Config 里的 Action 和 ActionForm 的映射把他们转为 ActionForm 后传给

Action 的.

在上面的例子, 我们只是简单地运行了 Action, 那么 Action 是否正确执行以及返回的结果是不是我们想要的呢?

我们继续完善 testSearchHotline() 这个 Method.

```
public void testSearchHotline() throws Exception{
    setRequestPathInfo("/SearchHotline.do");
    addRequestParameter("country", "CN");
    actionPerform();
    verifyNoActionErrors();
    verifyForward("hotlineList");
    assertNotNull(request.getAttribute("hotlineList"));
    List hotlineList = (List) request.getAttribute("hotlineList");
    for (Iterator it = hotlineList.iterator(); it.hasNext();) {
        ....
    }
}
```

我们在 actionPerform() 后增加了几行语句来断定 Struts Action 是否正确执行.

verifyNoActionErrors() -- 判断 Action 里没有任何的 Action;

verifyForward("hotlineList") -- 判断 Action 确实转发到 hotlineList;

assertNotNull(request.getAttribute("hotlineList")) -- 判断 Action 确实返回了 hotlineList 并且不为空

到这里, 我们已经基本上讨论完了 StrutsTestCase 的核心部分. 从头到尾, 我们没有发布应用程序, 也不需要 Web 服务器, 对我们来讲, Struts Action 就象普通的 Java Class 一样容易调试测试. 这就是 StrutsTestCase 给我们带来的方便.

### 7.13.3. 深入 StrutsTestCase

除了以上我们用到的几个断定和校验方法外, StrutsTestCase 还提供了其他几个方法便于我们测试 Struts Action. 下面我一一讲述, 具体的大家可以参考文档.

verifyActionErrors/Messages -- 校验 ActionActionServlet controller 是否发送了 ActionError 或 ActionMessage. 参数为 ActionError/Message Key

verifyNoActionErrors/Messages -- 校验 ActionActionServlet controller 没有发送 ActionError 或 ActionMessage

VerifyForward -- 校验 Action 是否正确转发到指定的 ActionForward.

VerifyForwardPath -- 校验 Action 是否正确转发到指定的 URL

verifyInputForward -- 校验 Action 是否转发到 Action Mapping 里的 input 属性

verifyTilesForward/verifyInputTilesForward-- 和以上类似, 应用程序使用到 tiles

时用的

#### 7.13.4. 关于 Web.xml 和 Struts-Config.xml

缺省情况下, StrutsTestCase 认为你的 Web.xml 和 struts-config.xml 的路径分别是:

/WEB-INF/web.xml 和 /WEB-INF/struts-config.xml

1. 假如你的 web.xml/struts-config.xml 的路径是

d:/app/web/WEB-INF/web.xml (struts-config.xml) 的话, 就需要把 d:/app/web 加到 classpath.

2. 假如你的 struts config 是 struts-config-module.xml,  
那么必须调用 setConfigFile() 设置你的 struts config 文件

#### 7.13.5. 小结

J2EE 应用程序的测试在开发过程中占有相当重要的位置, 利用 StrutsTestCase 能极大方便你测试基于 Struts 的应用程序。

# 第8章 Hibernate 与 Java 对象持久化技术

## 8.1. Quick Start

### 8.1.1. 准备工作

- 1、下载 Ant 软件包，解压缩（如 C:\ant\）。并将其 bin 目录（如 c:\ant\bin）添加到系统 PATH 中。
- 2、下载 Hibernate、Hibernate-Extension 和 Middlegen-Hibernate 软件包的最新版本。

<http://prdownloads.sourceforge.net/hibernate/>

### 8.1.2. 构建 Hibernate 基础代码

Hibernate 基础代码包括：

#### 1、POJO

POJO 在 Hibernate 语义中理解为数据库表所对应的 Domain Object。这里的 POJO 就是所谓的“Plain Ordinary Java Object”，字面上来讲就是无格式普通 Java 对象，简单的可以理解为一个不包含逻辑代码的值对象（Value Object 简称 VO）。

一个典型的 POJO：

```
public class TUser implements Serializable {  
    private String name;  
    public User(String name) {  
        this.name = name;  
    }  
    /** default constructor */  
    public User() {}  
    public String getName() {  
        return this.name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

#### 2、Hibernate 映射文件

Hibernate 从本质上讲是一种“对象—关系型数据映射”（Object Relational Mapping，简称 ORM）。前面的 POJO 在这里体现的就是 ORM 中 Object 层的语义，而映射（Mapping）文件则是将对象（Object）与关系型数据（Relational）相关联的纽带，在 Hibernate 中，映射文件通常以“. hbm.xml”作为后缀。

构建 Hibernate 基础代码通常有以下途径：

- 1、手工编写
- 2、直接从数据库中导出表结构，并生成对应的 ORM 文件和 Java 代码。

这是实际开发中最常用的方式，也是这里所推荐的方式。

通过直接从目标数据库中导出数据结构，最小化了手工编码和调整的可能性，从而最大

程度上保证了 ORM 文件和 Java 代码与实际数据库结构相一致。

3、根据现有的 Java 代码生成对应的映射文件，将 Java 代码与数据库表相绑定。

通过预先编写好的 POJO 生成映射文件，这种方式在实际开发中也经常使用，特别是结合了 xdoclet 之后显得尤为灵活，其潜在问题就是与实际数据库结构之间可能出现的同步上的障碍，由于需要手工调整代码，往往调整的过程中由于手工操作的疏漏，导致最后生成的配置文件错误，这点需要在开发中特别注意。

结合 xdoclet，由 POJO 生成映射文件的技术我们将在“高级特性”章节中进行探讨。

### 8.1.3. 由数据库产生基础代码

通过 Hibernate 官方提供的 MiddleGen for Hibernate 和 Hibernate\_Extension 工具包，我们可以很方便的根据现有数据库，导出数据库表结构，生成 ORM 和 POJO。

1) 首先，将 MiddleGen-Hibernate 软件包解压缩（如解压缩到 C:\MiddleGen\）。

2) 配置目标数据库参数

进入 MiddleGen 目录下的\config\database 子目录，根据我们实际采用的数据库打开对应的配置文件。如这里我们用的是 mysql 数据库，对应的就是 mysql.xml 文件。

```
<property name="database.script.file"
value="${src.dir}/sql/${name}-mysql.sql"/>
<property name="database.driver.file"
value="${lib.dir}/mysql.jar"/>
<property name="database.driver.classpath"
value="${database.driver.file}"/>
<property name="database.driver"
value="org.gjt.mm.mysql.Driver"/>
<property name="database.url"
value="jdbc:mysql://localhost/sample"/>
<property name="database.userid"
value="user"/>
<
<property name="database.password"
value="mypass"/>
<property name="database.schema"
value="" />
<property name="database.catalog"
value="" />
<property name="jboss.datasource.mapping"
value="mySQL" />
```

其中下划线标准的部分是我们进行配置的内容，分别是数据 url 以及数据库用户名和密码。

3) 修改 Build.xml

修改 MiddleGen 根目录下的 build.xml 文件，此文件是 MiddleGen-Hibernate 的 Ant 构建配置。MiddleGen-Hibernate 将根据 build.xml 文件中的具体参数生成数据库表映射文件。可配置的项目包括：

a) 目标数据库配置文件地址查找关键字 ” !ENTITY”，得到：

```
<!DOCTYPE project [
```

```
<!ENTITY database SYSTEM  
"file:./config/database/hsqldb.xml">  
]>
```

默认情况下，MiddleGen 采用的是 hsqldb.xml，将其修改为我们所用的数据库配置文件 (mysql.xml)：

```
<!DOCTYPE project [  
<!ENTITY database SYSTEM  
"file:./config/database/mysql.xml">  
]>
```

b) Application name

查找：

```
<property name="name" value="aireline" />
```

“aireline”是 MiddleGen 原始配置中默认的 Application Name，将其修改为我们所希望的名称，如“HibernateSample”：

```
<property name="name" value="HibernateSample"/>
```

c) 输出目录

查找关键字“name=“build.gen-src.dir””，得到：

```
<property name="build.gen-src.dir"  
value="${build.dir}/gen-src"/>
```

修改 value="\${build.dir}/gen-src"使其指向我们所期望的输出目录，这里我们修改为：

```
<property name="build.gen-src.dir"  
value="C:\sample"/>
```

d) 对应代码的 Package name

查找关键字“destination”，得到：

```
<hibernate  
destination="${build.gen-src.dir}"  
package="${name}.hibernate"  
genXDocletTags="false"  
genIntergratedCompositeKeys="false"  
javaTypeMapper=  
"middlegen.plugins.hibernate.HibernateJavaTypeMapper"  
/>
```

可以看到，hibernate 节点 package 属性的默认设置实际上是由前面的 Application Name（\${name}）和“.hibernate”组合而成，根据我们的需要，将其改为：

```
<hibernate  
destination="${build.gen-src.dir}"  
package="org.hibernate.sample"  
genXDocletTags="true"  
genIntergratedCompositeKeys="false"  
javaTypeMapper=  
"middlegen.plugins.hibernate.HibernateJavaTypeMapper"  
/>
```

这里还有一个属性 genXDocletTags，如果设置为 true，则生成的代码将包含 xdoclet

tag, 这为以后在开发过程中借助 xdoclet 进行映射调整提供了帮助。关于 Hibernate 的 xdoclet 使用, 请参见“高级特性”中的相关内容。

注意, 如果使用的数据库为 SQLServer, 需要将 build.xml 中如下部分(下划线部分)删除, 否则 MiddleGen 会报出找不到表的错误。

```
<middlegen
appname="${name}"
prefsdir="${src.dir}"
gui="${gui}"
databaseurl="${database.url}"
initialContextFactory="${java.naming.factory.initial}"
providerURL="${java.naming.provider.url}"
datasourceJNDIName="${datasource.jndi.name}"
driver="${database.driver}"
username="${database.userid}"
password="${database.password}"
schema="${database.schema}"
catalog="${database.catalog}"
>
```

至此为止, MiddleGen 已经配置完毕, 在 MiddleGen 根目录下运行 ant, 就将出现 MiddleGen 的界面:

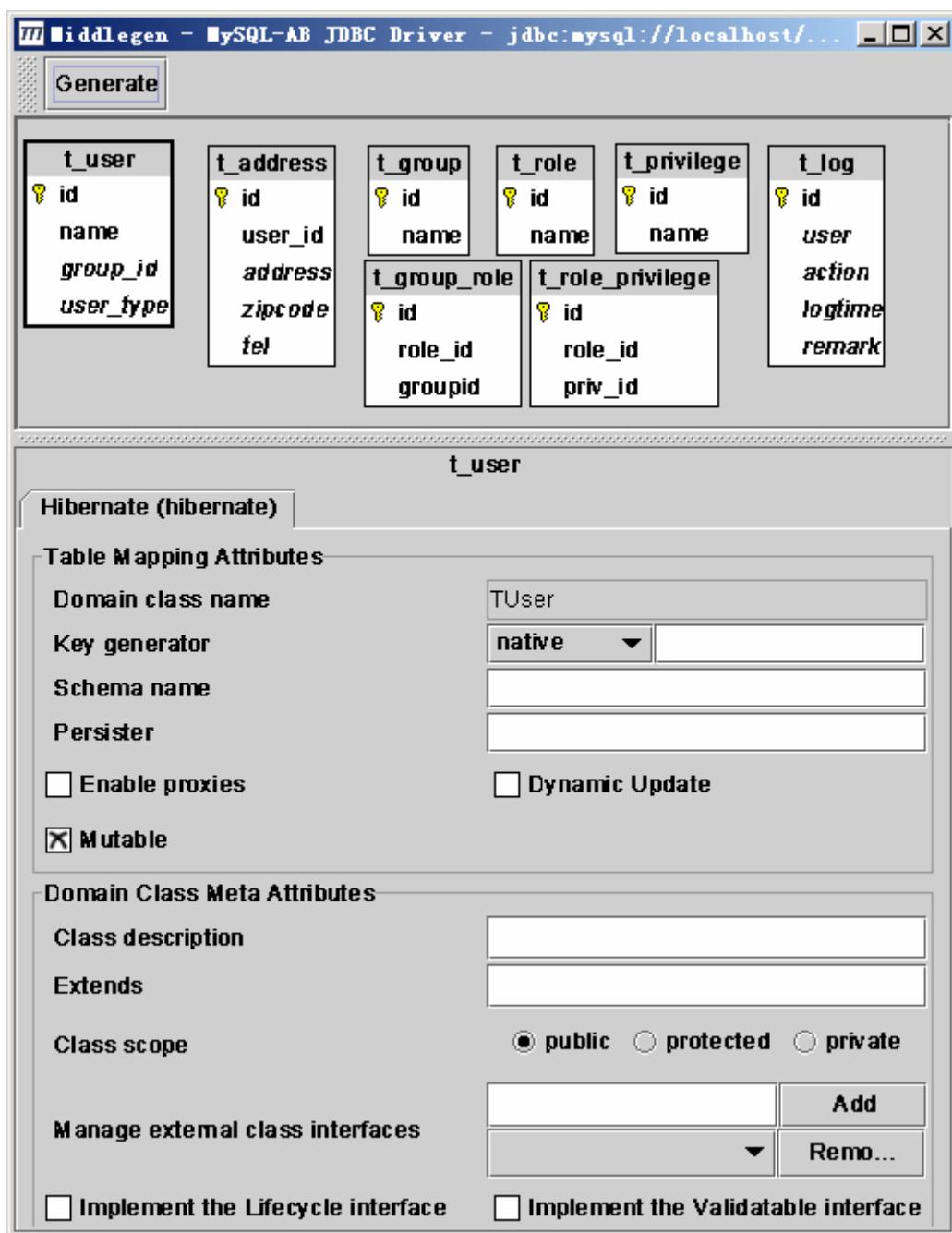


图 8-1

可以看到，数据库中的表结构已经导入到 MiddleGen 的操作界面中，选定数据库表视图中的表元素，我们即可调整各个数据库表的属性。

### 1、Domain Class Name

对应 POJO 的类名

### 2、Key Generator 主键产生器可选项说明：

#### 1) Assigned

主键由外部程序负责生成，无需 Hibernate 参与。

#### 2) hilo

通过 hi/lo 算法实现的主键生成机制，需要额外的数据库表保存主键生成历史状态。

#### 3) seqhilo

与 hilo 类似，通过 hi/lo 算法实现的主键生成机制，只是主键历史状态保存在

Sequence 中，适用于支持 Sequence 的数据库，如 Oracle。

#### 4) increment

主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存着当前的最大值，之后每次需要生成主键的时候将此值加 1 作为主键。

这种方式可能产生的问题是：如果当前有多个实例访问同一个数据库，那么由于各个实例各自维护主键状态，不同实例可能生成同样的主键，从而造成主键重复异常。因此，如果同一数据库有多个实例访问，此方式必须避免使用。

#### 5) identity

采用数据库提供的主键生成机制。如 DB2、SQL Server、MySQL 中的主键生成机制。

#### 6) sequence

采用数据库提供的 sequence 机制生成主键。如 Oracle 中的 Sequence。

#### 7) native

由 Hibernate 根据底层数据库自行判断采用 identity、hilo、sequence 其中一种作为主键生成方式。

#### 8) uuid.hex

由 Hibernate 基于 128 位唯一值产生算法生成 16 进制数值（编码后以长度 32 的字符串表示）作为主键。

#### 9) uuid.string

与 uuid.hex 类似，只是生成的主键未进行编码（长度 16）。在某些数据库中可能出现问题（如 PostgreSQL）。

#### 10) foreign

使用外部表的字段作为主键。

一般而言，利用 uuid.hex 方式生成主键将提供最好的性能和数据库平台适应性。

另外由于常用的数据库，如 Oracle、DB2、SQLServer、MySQL 等，都提供了易用的主键生成机制（Auto-Increase 字段或者 Sequence）。我们可以在数据库提供的主键生成机制上，采用 generator-class=native 的主键生成方式。

不过值得注意的是，一些数据库提供的主键生成机制在效率上未必最佳，大量并发 insert 数据时可能会引起表之间的互锁。

数据库提供的主键生成机制，往往是通过在一个内部表中保存当前主键状态（如对于自增型主键而言，此内部表中就维护着当前的最大值和递增量），之后每次插入数据会读取这个最大值，然后加上递增量作为新记录的主键，之后再把这个新的最大值更新回内部表中，这样，一次 Insert 操作可能导致数据库内部多次表读写操作，同时伴随的还有数据的加锁解锁操作，这对性能产生了较大影响。因此，对于并发 Insert 要求较高的系统，推荐采用 uuid.hex 作为主键生成机制。

**3、如果需要采用定制的主键产生算法，则在此处配置主键生成器，主键生成器必须实现 net.sf.hibernate.id.IdentifierGenerator 接口。**

#### 4、Schema Name

数据库 Schema Name。

#### 5、Persister

自定义持久类实现类类名。如果系统中还需要 Hibernate 之外的持久层实现机制，如通过存储过程得到目标数据集，甚至从 LDAP 中获取数据来填充我们的 POJO。

#### 6、Enable proxies

是否使用代理（用于延迟加载[Lazy Loading]）。

#### 7、Dynamic Update

如果选定，则生成 Update SQL 时不包含未发生变动的字段属性，这样可以在一定程度上提升 SQL 执行效能。

#### 8、**Mutable**

类是否可变，默认为选定状态（可变）。如果不希望应用程序对此类对应的数据记录进行修改（如对于数据库视图），则可将取消其选定状态，之后对此类的 Delete 和 Update 操作都将失效。

#### 9、**Implement the Lifecycle interface**

是否实现 Lifecycle 接口。Lifecycle 接口提供了数据固化过程中的控制机制，通过实现 Lifecycle 接口，我们可以在数据库操作中加入回调（Call Back）机制，如在数据库操作之前，之后触发指定操作。

#### 10、**Implement the Validatable interface**

是否实现 Validatable 接口。通过实现 Validatable 接口，我们可以在数据被固化到数据库表之前对其合法性进行验证。

值得注意的是，通过实现 Lifecycle 接口，我们同样可以在数据操作之前验证数据合法性，不同的是，Validatable 接口中定义的 validate 方法可能会被调用多次，因此设计中应避免在 Validatable 接口的 validate 方法实现中加入业务逻辑的验证。

以上是针对 Class 的设置，同样，在 MiddleGen 中，我们也可以设定字段属性。在 MiddleGen 中选定某个字段，界面下方即出现字段设置栏：

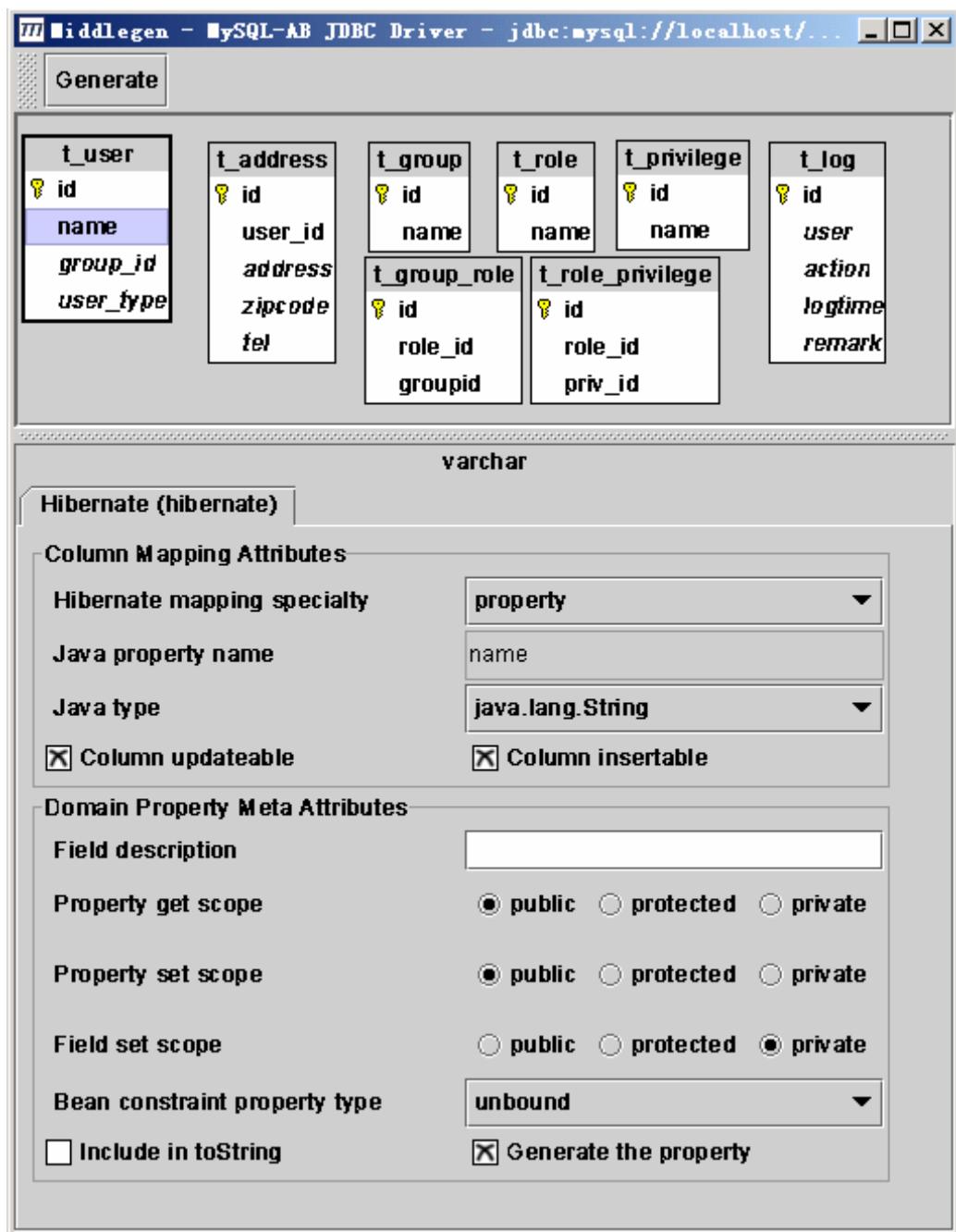


图 8-2

在这里我们可以设置字段的属性，其中：

### 1、Hibernate mapping specialty

映射类型：

Key : 主键

Property : 属性

Version : 用于实现 optimistic locking，参见“高级特性”章节中关于 optimistic locking 的描述

### 2、Java property name

字段对应的 Java 属性名

### 3、Java Type

字段对应的 Java 数据类型

#### 4、Column updateable

生成 Update SQL 时是否包含本字段。

#### 5、Column insertable

生成 Insert SQL 时是否包含本字段。

单击窗口顶部的 Generate 按钮，MiddleGen 即为我们生成这些数据库表所对应的 Hibernate 映射文件。在 MiddleGen 根目录下的\build\gen-src\net\hibernate\sample 目录中，我们可以看到对应的以.hbm.xml 作为后缀的多个映射文件，每个映射文件都对应了数据库的一个表。仅有映射文件还不够，我们还需要根据这些文件生成对应的 POJO。

POJO 的生成工作可以通过 Hibernate Extension 来完成，Hibernate Extension 的 tools\bin 目录下包含三个工具：

##### 1、hbm2java.bat

根据映射文件生成对应的 POJO。通过 MiddleGen 我们已经得到了映射文件，下一步就是通过 hbm2java.bat 工具生成对应的 POJO。

##### 2、class2hbm.bat

根据 POJO class 生成映射文件，这个工具很少用到，这里也就不再详细介绍。

##### 3、ddl2hbm.bat

由数据库导出库表结构，并生成映射文件以及 POJO。这个功能与 MiddleGen 的功能重叠，但由于目前还不够成熟（实际上已经被废弃，不再维护），提供的功能也有限，所以我们还是采用 MiddleGen 生成映射文件，之后由 hbm2java 根据映射文件生成 POJO 的方式。

为了使用以上工具，首先我们需要配置一些参数，打开 tools\bin\setenv.bat 文件，修改其中的 JDBC\_DRIVER 和 HIBERNATE\_HOME 环境变量，使其指向我们的实际 JDBC Driver 文件和 Hibernate 所在目录，如

```
set JDBC_DRIVER=c:\mysql\mysql.jar  
set HIBERNATE_HOME=c:\hibernate
```

同时检查环境变量 CP 中的各个项目中是否实际存在，特别是%CORELIB%下的 jar 文件，某些版本的发行包中，默认配置中的文件名与实际的文件名有所出入（如%CORELIB%\commons-logging.jar，在 Hibernate 发行包中，可能实际的文件名是 commons-logging-1.0.3.jar，诸如此类）。

使用 hbm2java，根据 MiddleGen 生成的映射文件生成 Java 代码：

打开 Command Window，在 tools\bin 目录下执行：

```
hbm2java c:\sample\org\hibernate\sample\*.xml --output=c:\sample\
```

即可生成对应的 POJO。生成的 POJO 保存在我们指定的输出目录下 (c:\sample)。

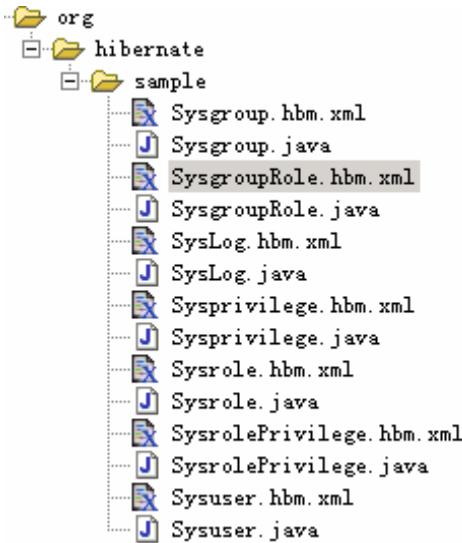


图 8-3

目前为止，我们已经完成了通过 MiddleGen 产生 Hibernate 基础代码的工作。配置 MiddleGen 也许并不是一件轻松的事情，对于 Eclipse 的用户而言，目前已经出现了好几个 Hibernate 的 Plugin，通过这些 Plugin 我们可以更加轻松的完成上述工作。

## 8.2. Hibernate 配置

前面已经得到了映射文件和 POJO，为了使 Hibernate 能真正运作起来，我们还需要一个配置文件。

Hibernate 同时支持 xml 格式的配置文件，以及传统的 properties 文件配置方式，不过这里建议采用 xml 型配置文件。xml 配置文件提供了更易读的结构和更强的配置能力，可以直接对映射文件加以配置，而在 properties 文件中则无法配置，必须通过代码中的 Hard Coding 加载相应的映射文件。下面如果不作特别说明，都指的是基于 xml 格式文件的配置方式。

配置文件名默认为“`hibernate.cfg.xml`”(或者 `hibernate.properties`)，Hibernate 初始化期间会自动在 CLASSPATH 中寻找这个文件，并读取其中的配置信息，为后期数据库操作做好

### 8.2.1. 准备

配置文件应部署在 CLASSPATH 中，对于 Web 应用而言，配置文件应放置在在 `\WEB-INF\classes` 目录下。

一个典型的 `hibernate.cfg.xml` 配置文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-2.0.
dtd">
<hibernate-configuration>
<!-- SessionFactory 配置 -->
<session-factory>
<!-- 数据库URL -->
<property name="hibernate.connection.url">
```

```
jdbc:mysql://localhost/sample
</property>
<!-- 数据库JDBC驱动 -->
<property name="hibernate.connection.driver_class">
org.gjt.mm.mysql.Driver
</property>
<!-- 数据库用户名 -->
<property name="hibernate.connection.username">
User
</property>
<!-- 数据库用户密码 -->
<property name="hibernate.connection.password">
dialect.MySQLDialect
</property>
<!-- 是否将运行期生成的SQL输出到日志以供调试 -->
<property name="hibernate.show_sql">
True
</property>
<!-- 是否使用数据库外连接 -->
<property name="hibernate.use_outer_join">
True
</property>
<!-- 事务管理类型，这里我们使用JDBC Transaction -->
<property name="hibernate.transaction.factory_class">
net.sf.hibernate.transaction.JDBCTransactionFactory
</property>
<!--映射文件配置，注意配置文件名必须包含其相对于根的全路径 -->
<mapping resource="net/xiaxin/xdoclet/TUser.hbm.xml"/>
<mapping resource="net/xiaxin/xdoclet/TGroup.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

一个典型的 hibernate.properties 配置文件如下：

```
hibernate.dialect net.sf.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class org.gjt.mm.mysql.Driver
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql:///sample
hibernate.connection.username user
hibernate.connection.password mypass
```

## 8.2.2. 第一段代码

上面我们已经完成了 Hiberante 的基础代码，现在先从一段最简单的代码入手，感受一下 Hibernate 所提供的强大功能。

下面这段代码是一个 JUnit TestCase，演示了 TUser 对象的保存和读取。考虑到读者可能没有 JUnit 的使用经验，代码中加入了一些 JUnit 相关注释。

```
public class HibernateTest extends TestCase {
    Session session = null;
    /**
     * JUnit中setUp方法在TestCase初始化的时候会自动调用
     * 一般用于初始化公用资源
     * 此例中，用于初始化Hibernate Session
     */
    protected void setUp(){
        try {
            /**
             * 采用hibernate.properties配置文件的初始化代码:
             * Configuration config = new Configuration();
             * config.addClass(TUser.class);
             */
            //采用hibernate.cfg.xml配置文件
            //请注意初始化Configuration时的差异:
            // 1.Configuration的初始化方式
            // 2.xml文件中已经定义了Mapping文件，因此无需再Hard Coding导入
            // POJO文件的定义
            Configuration config = new Configuration().configure();
            SessionFactory sessionFactory =
            config.buildSessionFactory();
            session = sessionFactory.openSession();
        } catch (HibernateException e) {
            e.printStackTrace();
        }
    }
    /**
     * 与setUp方法相对应，JUnit TestCase执行完毕时，会自动调用tearDown方法
     * 一般用于资源释放
     * 此例中，用于关闭在setUp方法中打开的Hibernate Session
     */
    protected void tearDown(){
        try {
            session.close();
        } catch (HibernateException e) {
            e.printStackTrace();
        }
    }
    /**
     * 对象持久化（Insert）测试方法
     *
     * JUnit中，以”test”作为前缀的方法为测试方法，将被JUnit自动添加
     * 到测试计划中运行
     */
```

```
*/  
public void testInsert(){  
try {  
TUser user = new TUser();  
user.setName("Emma");  
session.save(user);  
session.flush();  
} catch (HibernateException e) {  
e.printStackTrace();  
Assert.fail(e.getMessage());  
}  
}  
/**  
* 对象读取(Select) 测试  
* 请保证运行之前数据库中已经存在name='Erica' 的记录  
*/  
public void testSelect(){  
String hql=  
" from TUser where name='Erica'";  
try {  
List userList = session.find(hql);  
TUser user =(TUser)userList.get(0);  
Assert.assertEquals(user.getName(), "Erica");  
} catch (HibernateException e) {  
e.printStackTrace();  
Assert.fail(e.getMessage());  
}  
}  
}
```

主流 IDE，如 Eclipse、IntelliJ IDEA 和 JBuilder 中都内置了 JUnit 支持。下面是 Eclipse 中运行该代码的结果（在 Run 菜单中选择 Run as -> JUnit Test 即可）：

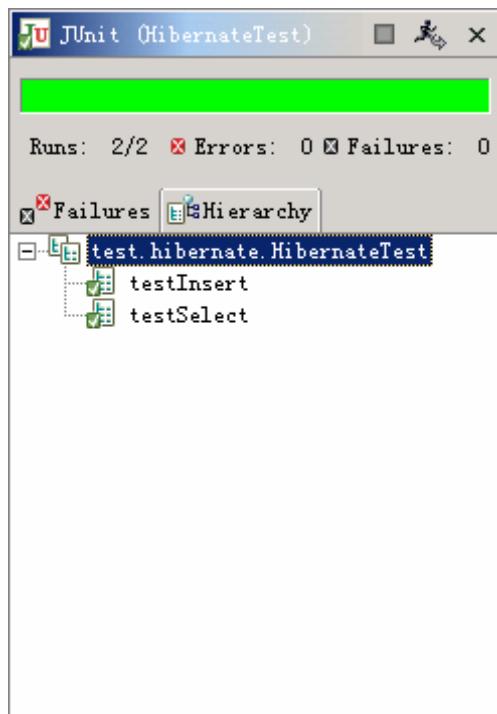


图 8-4

现在我们已经成功实现了一个简单的 TUser 实例的保存和读取。可以看到，程序中通过少量代码实现了 Java 对象和数据库数据的同步，同时借助 Hibernate 的有力支持，轻松实现了对象到关系型数据库的映射。

相对传统的 JDBC 数据访问模式，这样的实现无疑更符合面向对象的思想，同时也大大提高了开发效率。

上面的代码中引入了几个 Hibernate 基础语义：

1. Configuration
2. SessionFactory
3. Session

下面我们就这几个关键概念进行探讨。

### 8.2.3. Hibernate 基础语义

#### 8.2.3.1. Configuration

正如其名，Configuration 类负责管理 Hibernate 的配置信息。Hibernate 运行时需要获取一些底层实现的基本信息，其中几个关键属性包括：

1. 数据库 URL
2. 数据库用户
3. 数据库用户密码
4. 数据库 JDBC 驱动类
5. 数据库 dialect，用于对特定数据库提供支持，其中包含了针对特定数据库特性的实现，如 Hibernate 数据类型到特定数据库数据类型的映射等。

使用 Hibernate 必须首先提供这些基础信息以完成初始化工作，为后继操作做好准备。这些属性在 hibernate 配置文件 (hibernate.cfg.xml 或 hibernate.properties) 中加以设定（参见前面“Hibernate 配置”中的示例配置文件内容）。

当我们调用：Configuration config = new Configuration().configure(); 时，Hibernate会自动在当前的CLASSPATH 中搜寻 hibernate.cfg.xml 文件并将其读取到内存中作为后继操作的基础配置。Configuration 类一般只有在获取 SessionFactory 时需要涉及，当获取 SessionFactory 之后，由于配置信息已经由 Hibernate 维护并绑定在返回的 SessionFactory 之上，因此一般情况下无需再对其进行操作。

我们也可以指定配置文件名，如果不希望使用默认的 hibernate.cfg.xml 文件作为配置文件的话：

```
File file = new File("c:\\sample\\\\myhibernate.xml");
Configuration config = new Configuration().configure(file);
```

### 8.2.3.2. SessionFactory

SessionFactory 负责创建 Session 实例。我们可以通过 Configuration 实例构建 SessionFactory：

```
Configuration config = new Configuration().configure();
SessionFactory sessionFactory = config.buildSessionFactory();
```

Configuration 实例 config 会根据当前的配置信息，构造 SessionFactory 实例并返回。SessionFactory 一旦构造完毕，即被赋予特定的配置信息。也就是说，之后 config 的任何变更将不会影响到已经创建的 SessionFactory 实例 (sessionFactory)。如果需要使用基于改动后的 config 实例的 SessionFactory，需要从 config 重新构建一个 SessionFactory 实例。

### 8.2.3.3. Session

Session 是持久层操作的基础，相当于 JDBC 中的 Connection。Session 实例通过 SessionFactory 实例构建：

```
Configuration config = new Configuration().configure();
SessionFactory sessionFactory = config.buildSessionFactory();
Session session = sessionFactory.openSession();
```

之后我们就可以调用 Session 所提供的 save、find、flush 等方法完成持久层操作：

Find:

```
String hql= " from TUser where name='Erica' ";
List userList = session.find(hql);
```

Save:

```
TUser user = new TUser();
user.setName("Emma");
session.save(user);
session.flush();
```

最后调用 Session.flush 方法强制数据库同步，这里即强制 Hibernate 将 user 实例立即同步到数据库中。如果在事务中则不需要 flush 方法，在事务提交的时候，hibernate 自动会执行 flush 方法，另外当 Session 关闭时，也会自动执行 flush 方法。

## 8.3. Hibernate 高级特性

### 8.3.1. XDoclet 与 Hibernate 映射

在 POJO 中融合 XDoclet 的映射文件自动生成机制，提供了除手动编码和由数据库导出

基础代码的第三种选择。本章将结合 XDoclet 对 Hibernate 中的数据映射进行介绍。

实际开发中,往往首先使用 MiddleGen 和 hbm2java 工具生成带有 XDoclet tag 的 POJO (MiddleGen build.xml 中的 genXDocletTags 选项决定了是否在映射文件中生成 XDoclet Tag, 详见 Hibernate Quick Start 章节中关于 MiddleGen 的说明)。之后通过修改 POJO 中的 XDocleltag 进行映射关系调整。

XDoclet 已经广泛运用在 EJB 开发中,在其最新版本里,包含了一个为 Hibernate 提供支持的子类库 Hibernate Doclet, 其中包含了生成 Hibernate 映射文件所需的 ant 构建支持以及 java doc tag 支持。

XDoclet 实现基本原理是,通过在 Java 代码加入特定的 JavaDoc tag, 从而为其添加特定的附加语义, 之后通过 XDoclet 工具对代码中 JavaDoc Tag 进行分析, 自动生成与代码对应的配置文件, XDoclet。

在 Hibernate-Doclet 中, 通过引入 Hibernate 相关的 JavaDoc tag, 我们就可以由代码生成对应的 Hibernate 映射文件。

下面是一个代码片断, 演示了 Hibernate-Doclet 的使用方式:

```
/**  
 * @hibernate.class  
 * table="TUser"  
 */  
  
public class TUser implements Serializable {  
....  
/**  
 * @hibernate.property  
 * column="name"  
 * length="50"  
 * not-null="true"  
 *  
 * @return String  
 */  
  
public String getName() {  
    return this.name;  
}  
....  
}
```

以上是使用 Hibernate-Doclet 描述 POJO (TUser) 及其对应表 (TUser) 之间映射关系的一个例子。

其中用到了两个 hibernate doclet tag, @hibernate.class 和@hibernate.property。这两个 tag 分别描述了 POJO 所对应的数据库表信息, 以及其字段对应的库表字段信息。之后 Hibernate Doclet 就会根据这些信息生成映射文件:

```
<  
hibernate-mapping>  
<class  
name="net.xiaxin.xdoclet.TUser"  
table="TUser"  
>
```

```

<property
name="name"
type="java.lang.String"
column="name"
not-null="true"
length="50"
>
</class>
</hibernate-mapping>

```

这样我们只需要维护 Java 代码，而无需再手动编写具体的映射文件即可完成 Hibernate 基础代码。

熟记 Hibernate-Doclet 众多的 Tag，显然不是件轻松的事情，好在目前的主流 IDE 都提供了 Live Template 支持。我们只需进行一些配置工作，就可以实现 Hibernate-Doclet Tag 的自动补全功能，从而避免了手工编写过程中可能出现的问题。

下面我们就 Hibernate Doclet 中常用的 Tag 进行探讨，关于 Tag 的详细参考，请参见 XDoclet 的官方指南（<http://xdoclet.sourceforge.net/xdoclet/tags/hibernate-tags.html>）以及 Hibernate Reference (<http://www.hibernate.org>)。

常用 Hibernate-Doclet Tag 介绍：

1. Class 层面：

1) @hibernate.class

描述 POJO 与数据库表之间的映射关系，并指定相关的运行参数。

参数	描述	类型	必须
table	类对应的表名默认值：当前类名	Text	N
dynamic-update	生成 Update SQL 时，仅包含发生变动的字段 默认值: false	Bool	N
dynamic-insert	生成 Insert SQL 时，仅包含非空(null)字段 默认值: false	Bool	N
Proxy	代理类 默认值: 空	Text	N
discriminator-value	子类辨别标识，用于多态支持。	Text	N
where	数据甄选条件，如果只需要处理库表中某些特定数据的时候，可通过此选项设定结果集限定条件。如用户表中保存了全国所有用户的 data，而我们的系统只是面向上海用户，则可指定 where="location='Shanghai'"	Text	N

典型场景：

```
/**  
 * @hibernate.class  
 * table="TUser" (1)  
 * dynamic-update="true" (2)  
 * dynamic-insert="true" (3)  
 * proxy="" (4)  
 * discriminator-value="1" (5)  
 */  
public class TUser implements Serializable {  
....  
}
```

本例中：

1 table 参数指定了当前类 (TUser) 对应数据库表 “TUser”。

2 dynamic-update 参数设定为生成 Update SQL 时候，只包括当前发生变化的字段（提高 DB Update 性能）。

3 Dynamic-insert 参数设定为生成 Insert SQL 时候，只包括当前非空字段。（提高 DB Insert 性能）

4 Proxy 参数为空，表明当前类不使用代理 (Proxy)。代理类的作用是为 LazyLoading 提供支持，请参见下面关于 Lazy Loading 的有关内容。

5 discriminator-value 参数设为”1”。

discriminator-value 参数的目的是对多态提供支持。请参见下面关于 @hibernate.discriminator 的说明。

2) @hibernate.discriminator

@hibernate.discriminator (识别器) 用于提供多态支持。

参数	描述	类型	必须
column	用于区分各子类的字段名称。 默认值：当前类名	text	Y
type	对应的 Hibernate 类型	Bool	N
length	字段长度	Bool	N

如：

TUser 类对应数据库表 TUser，并且 User 类有两个派生类 SysAdmin、SysOperator。

在 TUser 表中，根据 user\_type 字段区分用户类型。

为了让 Hibernate 根据 user\_type 能自动识别对应的 Class 类型 (如 user\_type==1 则自动映射到 SysAdmin 类, user\_type==2 则自动映射到 SysOperator 类)，我们需要在映射文件中进行配置，而在 Hibernate-Doclet 中，对应的就是@hibernate.discriminator 标识和 @hibernate.class 以及 @hibernate.subclass 的 discriminator-value 属性。

典型场景：

```

/**
 *
 * @hibernate.class
 * table="TUser"
 * dynamic-update="true"
 * dynamic-insert="true"
 *
 * @hibernate.discriminator column="user_type" type="integer"
 */
public class TUser implements Serializable {
.....
}

```

根类 TUser 中，通过@hibernate.discriminator 指定了以"user\_type"字段作为识别字段。

```

/**
 * @hibernate.subclass
 * discriminator-value="1"
 */
public class SysAdmin extends TUser {
.....
}

/**
 * @hibernate.subclass
 * discriminator-value="2"
 */
public class SysOperator extends TUser {
.....
}

```

SysAdmin 和 SysOperator 均继承自 TUser，其 discriminator-value 分别设置为"1"和"2"，运行期 Hibernate 在读取 t\_user 表数据时，会根据其 user\_type 字段进行判断，如果是 1 的话则映射到 SysAdmin 类，如果是 2 映射到 SysOperator 类。上例中，描述 SysAdmin 和 SysOperator 时，我们引入了一个 Tag:

@hibernate.subclass，顾名思义，@hibernate.subclass 与@hibernate.class 不同之处就在于，@hibernate.subclass 描述的是一个子类，实际上，这两个 Tag 除去名称不同外，并没有什么区别。

## 2. Method 层面：

### 1) @hibernate.id

描述 POJO 中关键字段与数据库表主键之间的映射关系。

参数	描述	类型	必须
column	用于区分各子类的字段名称。 默认值：当前类名	text	Y

type	字段类型。 Hibernate总是使用对象型数据类型作为字段类型，如int对应Integer，因此这里将id设为基本类型[如int]以避免对象创建的开销的思路是没有实际意义的，即使这里设置为基本类型，Hibernate内部还是会使用对象型数据对其进行处理，只是返回数据的时候再转换为基本类型而已。	text	N
length	字段长度	text	N
unsaved-value	用于对象是否已经保存的判定值。 详见“数据访问”章节的相关讨论。	Text	N
generator-class	主键产生方式（详见Hibernate Quick Start中关于MiddleGen的相关说明） 取值可为下列值中的任意一个： assigned hilo seqhilo increment identity sequence native uuid.hex uuid.string foreign	Text	Y

## 2) @hibernate.property

描述 POJO 中属性与数据库表字段之间的映射关系。

参数	描述	类型	必须
column	用于区分各子类的字段名称。 默认值：当前类名	text	Y
type	字段类型	Text	N
length	字段长度	Text	N
not-null	字段是否允许为空	Bool	N
unique	字段是否唯一(是否允许重复值)	Bool	N
insert	Insert操作时是否包含本字段数据 默认：true	Bool	N
update	Update操作时是否包含本字段数据 默认：true	Bool	N

典型场景：

```

/**
 * @hibernate.property
 * column="name"
 * length="50"
 * not-null="true"
 *
 * @return String
 */
public String getName() {
    return this.name;
}

```

注意：在编写代码的时候请，对将 POJO 的 getter/setter 方法设定为 public，如果设定为 private，Hibernate 将无法对属性的存取进行优化，只能转而采用传统的反射机制进行操作，这将导致大量的性能开销（特别是在 1.4 之前的 Sun JDK 版本以及 IBM JDK 中，反射所带来的系统开销相当可观）。

包含 XDoclet Tag 的代码必须由 xdoclet 程序进行处理以生成对应的映射文件，xdoclet 的处理模块可通过 ant 进行加载，下面是一个简单的 hibernate xdoclet 的 ant 构建脚本(注意实际使用时需要根据实际情况对路径和 CLASSPATH 设定进行调整)：

```
<?xml version="1.0"?>
```

```
<project name="Hibernate" default="hibernate" basedir=".">>
<property name="xdoclet.lib.home"
value="C:\xdoclet-1.2.1\lib"/>
<target name="hibernate" depends=""
description="Generates Hibernate class descriptor files.">
<taskdef name="hibernatedoclet"
classname="xdoclet.modules.hibernate.HibernateDocletTask">
<classpath>
<fileset dir="${xdoclet.lib.home}">
<include name="*.jar"/>
</fileset>
</classpath>
</taskdef>
<hibernatedoclet
destdir="./src/"
excludedtags="@version,@author,@todo"
force="true"
verbose="true"
mergedir=".">
<fileset dir="./src/">
<include name="**/hibernate/sample/*.java"/>
</fileset>
<hibernate version="2.0"/>
</hibernatedoclet>
</target>
</project>
```

除了上面我们介绍的 Hibernate Doclet Tag，其他还有：

Class 层面：

@hibernate.cache  
@hibernate.jcs-cache  
@hibernate.joined-subclass  
@hibernate.joined-subclass-key  
@hibernate.query

Method 层面

@hibernate.array  
@hibernate.bag  
@hibernate.collection-cache  
@hibernate.collection-composite-element  
@hibernate.collection-element  
@hibernate.collection-index  
@hibernate.collection-jcs-cache  
@hibernate.collection-key  
@hibernate.collection-key-column  
@hibernate.collection-many-to-many

```
@hibernate.collection-one-to-many  
@hibernate.column  
@hibernate.component  
@hibernate.generator-param  
@hibernate.index-many-to-many  
@hibernate.list  
@hibernate.many-to-one  
@hibernate.map  
@hibernate.one-to-one  
@hibernate.primitive-array  
@hibernate.set  
@hibernate.timestamp  
@hibernate.version
```

具体的 Tag 描述请参见 XDoclet 官方网站提供的 Tag 说明 1。下面的 Hibernate 高级特性介绍中，我们也将涉及到这些 Tag 的实际使用。

### 8.3.2. 数据检索

数据查询与检索是 Hibernate 中的一个亮点。相对其他 ORM 实现而言，Hibernate 提供了灵活多样的查询机制。其中包括：

1. Criteria Query
2. Hibernate Query Language (HQL)
3. SQL

#### 8.3.2.1. Criteria Query

Criteria Query 通过面向对象化的设计，将数据查询条件封装为一个对象。简单来讲，Criteria Query 可以看作是传统 SQL 的对象化表示，如：

```
Criteria criteria = session.createCriteria(TUser.class);  
criteria.add(Expression.eq("name", "Erica"));  
criteria.add(Expression.eq("sex", new Integer(1)));
```

这里的 criteria 实例实际上是 SQL “Select \* from t\_user wherename='Erica' and sex=1”的封装（我们可以打开 Hibernate 的 show\_sql 选项，以观察 Hibernate 在运行期生成的 SQL 语句）。

Hibernate 在运行期会根据 Criteria 中指定的查询条件（也就是上面代码中通过 criteria.add 方法添加的查询表达式）生成相应的 SQL 语句。

这种方式的特点是比较符合 Java 程序员的编码习惯，并且具备清晰的可读性。正因为此，不少 ORM 实现中都提供了类似的实现机制（如 Apache OJB）。

对于 Hibernate 的初学者，特别是对 SQL 了解有限的程序员而言，Criteria Query 无疑是上手的极佳途径，相对 HQL，Criteria Query 提供了更易于理解的查询手段，借助 IDE 的 Coding Assist 机制，Criteria 的使用几乎不用太多的学习。

#### 8.3.2.2. Criteria 查询表达式

Criteria 本身只是一个查询容器，具体的查询条件需要通过 Criteria.add 方法添加到 Criteria 实例中。如前例所示，Expression 对象具体描述了查询条件。针对 SQL 语法，Expression 提供了对应的查询限定机制，包括：

方法	描述
Expression.eq	对应 SQL “field = value” 表达式。 如 Expression.eq("name", "Erica")
Expression.allEq	参数为一个 Map 对象，其中包含了多个属性一值对应关系。相当于多个 Expression.eq 关系的叠加。
Expression.gt	对应 SQL 中的 “field > value ” 表达式
Expression.ge	对应 SQL 中的 “field >= value” 表达式
Expression.lt	对应 SQL 中的 “field < value” 表达式
Expression.le	对应 SQL 中的 “field <= value” 表达式
Expression.between	对应 SQL 中的 “between” 表达式 如下面的表达式表示年龄 (age) 位于 13 到 50 区间内。 Expression.between("age", new Integer(13), new Integer(50));
Expression.like	对应 SQL 中的 “field like value” 表达式
Expression.in	对应 SQL 中的 ”field in ...” 表达式
Expression.eqProperty	用于比较两个属性之间的值，对应 SQL 中的 “field = field”。 如： Expression.eqProperty( "TUser.groupID", "TGroup.id" )；
Expression.gtProperty	用于比较两个属性之间的值，对应 SQL 中的 “field > field”。
Expression.geProperty	用于比较两个属性之间的值，对应 SQL 中的 “field >= field”。
Expression.ltProperty	用于比较两个属性之间的值，对应 SQL 中的 “field < field”。
Expression.leProperty	用于比较两个属性之间的值，对应 SQL 中的 “field <= field”。

	<= field”。
Expression.and	<p>and 关系组合。 如:</p> <pre>Expression.and(     Expression.eq("name", "Erica"),     Expression.eq(         "sex",         new Integer(1)     ) );</pre>
Expression.or	<p>or 关系组合。 如:</p> <pre>Expression.or(     Expression.eq("name", "Erica"),     Expression.eq("name", "Emma") );</pre>
Expression.sql	<p>作为补充, 本方法提供了原生 SQL 语法的支持。我们可以通过这个方法直接通过 SQL 语句限定查询条件。</p> <p>下面的代码返回所有名称以 “Erica” 起始的记录:</p> <pre>Expression.sql(     "lower({alias}.name) like lower(?)",     "Erica%",     Hibernate.STRING );</pre> <p>其中的 “{alias}” 将由 Hibernate 在运行期使用当前关联的 POJO 别名替换。</p>

注意 Expression 各方法中的属性名参数 (如 Express. eq 中的第一个参数), 这里所谓属性名是 POJO 中对应实际库表字段的属性名 (大小写敏感), 而非库表中的实际字段名称。

### 8.3.2.3. Criteria 高级特性

#### 限定返回的记录范围

通过 criteria. setFirstResult/setMaxResults 方法可以限制一次查询返回的记录范围:

```
Criteria criteria = session.createCriteria(TUser.class);
//限定查询返回检索结果中, 从第一百条结果开始的 20 条记录
criteria.setFirstResult(100);
criteria.setMaxResults(20);
```

#### 对查询结果进行排序

```
//查询所有 groupId=2 的记录
```

```
//并分别按照姓名(顺序)和groupId(逆序)排序
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("groupId", new Integer(2)));
criteria.addOrder(Order.asc("name"));
criteria.addOrder(Order.desc("groupId"));
```

Criteria 作为一种对象化的查询封装模式，不过由于 Hibernate 在实现过程中将精力更加集中在 HQL 查询语言上，因此 Criteria 的功能实现还没做到尽善尽美（这点上，OJB 的 Criteria 实现倒是值得借鉴），因此，在实际开发中，建议还是采用 Hibernate 官方推荐的查询封装模式：HQL。

### 8.3.3. Hibernate Query Language (HQL)

Criteria 提供了更加符合面向对象编程模式的查询封装模式。不过，HQL (HibernateQuery Language) 提供了更加强大的功能，在官方开发手册中，也将 HQL 作为推荐的查询模式。相对 Criteria，HQL 提供了更接近传统 SQL 语句的查询语法，也提供了更全面的特性。

最简单的一个例子：

```
String hql = "from org.hibernate.sample.TUser";
Query query = session.createQuery(hql);
List userList = query.list();
```

上面的代码将取出 TUser 的所有对应记录。

如果我们需要取出名为“Erica”的用户的记录，类似 SQL，我们可以通过 SQL 语句加以限定：

```
String hql =
"from org.hibernate.sample.TUser as user where user.name='Erica'";
Query query = session.createQuery(hql);
List userList = query.list();
```

其中我们新引入了两个子句“as”和“where”，as 子句为类名创建了一个别名，而 where 子句指定了限定条件。

HQL 子句本身大小写无关，但是其中出现的类名和属性名必须注意大小写区分。

关于 HQL，Hibernate 官方开发手册中已经提供了极其详尽的说明和示例，详见 Hibernate 官方开发手册 (Chapter 11)。

## 8.4. 数据关联

### 8.4.1. 一对一关联

Hibernate 中的一对一关联由“one-to-one”节点定义。在我们的权限管理系统示例中，每个用户都从属于一个用户组。如用户“Erica”从属于“System Admin”组，从用户的角度出发，这就是一个典型的（单向）一对一关系。

每个用户对应一个组，这在我们的系统中反映为 TUser 到 TGroup 的 one-to-one 关系。其中 TUser 是主控方，TGroup 是被控方。

one-to-one 关系定义比较简单，只需在主控方加以定义。这里，我们的目标是由 TUser 对象获取其对应的 TGroup 对象。因此 TUser 对象是主控方，为了实现一对一关系，我们在 TUser 对象的映射文件 TUser.hbm.xml 中加入 one-to-one 节点，对 TGroup 对象进行一对一关联：

```

<hibernate-mapping>
<class
  name="org.hibernate.sample.TUser"
  table="t_user"
  dynamic-update="true"
  dynamic-insert="true"
>
.....
<one-to-one
  name="group"
  class="org.hibernate.sample.TGroup"
  cascade="none"
  outer-join="auto"
  constrained="false"
/>
.....
</class>
</hibernate-mapping>

```

如果采用 XDoclet，则对应的 Tag 如下：

```

/**
 * @hibernate.class
 * table="t_user"
 * dynamic-update="true"
 * dynamic-insert="true"
 *
 */
public class TUser implements Serializable {
.....
private TGroup group;
/**
 * @hibernate.one-to-one
 * name="group"
 * cascade="none"
 * class="org.hibernate.sample.TGroup"
 * outer-join="auto"
 * @return
 */
public TGroup getGroup() {
    return group;
}
.....
}

```

one-to-one 节点有以下属性：

属性	描述	类型	必须
----	----	----	----

name	映射属性	Text	N
class	目标映射类。 注意要设为包含Package name的全路径名称。	Text	N
cascade	操作级联 (cascade) 关系。 可选值： <b>all</b> ：所有情况下均进行级联操作。 <b>none</b> ：所有情况下均不进行级联操作。 <b>save-update</b> : 在执行 <b>save-update</b> 时进行级联操作。 <b>delete</b> : 在执行 <b>delete</b> 时进行级联操作。 级联 (cascade) 在Hibernate映射关系中是个非常重要的概念。它指的是当主控方执行操作时，关联对象（被动方）是否同步执行同一操作。如对主控对象调用 <b>save-update</b> 或 <b>delete</b> 方法时，是否同时对关联对象（被动方）进行 <b>save-update</b> 或 <b>delete</b> 。 这里，当用户 (TUser) 被更新或者删除时，其所关联的组 (TGroup) 不应被修改或者删除，因此，这里的级联关系设置为 <b>none</b> 。	Text	N
constrained	约束 表明主控表的主键上是否存在一个外键 (foreign key) 对其进行约束。这个选项关系到 <b>save</b> 、 <b>delete</b> 等方法的级联操作顺序。	Bool	N
outer-join	是否使用外联接。 <b>true</b> : 总是使用 <b>outer-join</b> <b>false</b> : 不使用 <b>outer-join</b> <b>auto</b> (默认) : 如果关联对象没有采用Proxy机制，则使用 <b>outer-join</b> .	Text	N
property-ref	关联类中用于与主控类相关联的属性名称。 默认为关联类的主键属性名。 这里我们通过主键达成一对一的关联，所以采用默认值即可。如果一对一的关联并非建立在主键之间，则可通过此参数指定	Text	N

	关联属性。		
access	属性值的读取方式。 可选项： field property (默认) ClassName	Text	N

## 8.4.2. 一对多关联

一对多关系在系统实现中也很常见。典型的例子就是父亲与孩子的关系。而在我们现在的这个示例中，每个用户（TUser）都关联到多个地址（TAddress），如一个用户可能拥有办公室地址、家庭地址等多个地址属性。这样，在系统中，就反应为一个“一对多”关联。

一对多关系分为单向一对多关系和双向一对多关系。

单向一对多关系只需在“一”方进行配置，双向一对多关系需要在关联双方均加以配置。

### 单向一对多关系

#### 配置：

对于主控方（TUser）：

```
TUser.hbm.xml:
<hibernate-mapping>
<class
  name="org.hibernate.sample.TUser"
  table="t_user"
  dynamic-update="true"
  dynamic-insert="true"
>
.....
<set
  name="addresses"
  table="t_address"
  lazy="false"
  inverse="false"
  cascade="all"
  sort="unsorted"
  order-by="zipcode asc"
>
<key
  column="user_id"
>
</key>
<one-to-many
  class="org.hibernate.sample.TAddress"
/>
</set>
```

```

.....
</class>
</hibernate-mapping>

    对应的 XDoclet Tag 如下：

/**
 * @hibernate.collection-one-to-many
 * class="org.hibernate.sample.TAddress"
 *
 * @hibernate.collection-key column="user_id"
 *
 * @hibernate.set
 * name="addresses"
 * table="t_address"
 * inverse="false"
 * cascade="all"
 * lazy="false"
 * sort="unsorted"
 * order-by="zipcode asc"
 *
 */
public Set getAddresses() {
    return addresses;
}

```

被动方 (Taddress) 的记录由 Hibernate 负责读取，之后存放在主控方指定的 Collection 类型属性中。

对于 one-to-many 关联关系，我们可以采用 java.util.Set ( 或者 net.sf.hibernate.collection.Bag ) 类型的 Collection，表现在 XML 映射文件中也就是 <set>...</set> ( 或<bag>...</bag> ) 节点。关于 Hibernate 的 Collection 实现，请参见 Hibernate Reference。

one-to-many 节点有以下属性：

属性	描述	类型	必须
name	映射属性	Text	N
table	目标关联数据库表。	Text	N
lazy	是否采用延迟加载。 <i>关于延迟加载，请参见后面相关章节。</i>	Text	N
inverse	用于标识双向关联中的被动态一端。 <b>inverse=false的一方（主控方）负责维护关联关系。</b> 默认值： false	Bool	N

cascade	操作级联 (cascade) 关系。 可选值： <b>all</b> ：所有情况下均进行级联操作。 <b>none</b> ：所有情况下均不进行级联操作。 <b>save-update</b> : 在执行 <b>save-update</b> 时 进行级联操作。 <b>delete</b> : 在执行 <b>delete</b> 时进行级联操作。	Text	N
sort	排序类型。 可选值： <b>unsorted</b> : 不排序（默认） <b>natural</b> : 自然顺序（避免与 <b>order-by</b> 搭配使用） <b>comparatorClass</b> : 指以某个实现了 <b>java.util.Comparator</b> 接口的类作为排 序算法。	Text	N
order-by	指定排序字段及其排序方式。 （JDK1.4以上版本有效）。 对应SQL中的 <b>order by</b> 子句。 避免与 <b>sort</b> 的“ <b>natural</b> ”模式同时使 用。	Text	N
where	数据甄选条件，如果只需要处理库表中某 些特定数据的时候，可通过此选项设定结 果集限定条件。	Text	N
outer-join	是否使用外联接。 <b>true</b> : 总是使用 <b>outer-join</b> <b>false</b> : 不使用 <b>outer-join</b> <b>auto</b> (默认) : 如果关联对象没有采用 <b>Proxy</b> 机制，则使用 <b>outer-join</b> .	Text	N
batch-size	采用延迟加载特性时 (Lazy Loading) 一次读入的数据数量。 此处未采用延迟加载机制，因此此属性忽 略。	Int	N
access	属性值的读取方式。 可选项： <b>field</b> <b>property</b> (默认) <b>ClassName</b>	Text	N

通过单向一对多关系进行关联相对简单，但是存在一个问题。由于是单向关联，为了保持关联关系，我们只能通过主控方对被控方进行级联更新。且如果被关联方的关联字段为“NOT NULL”，当 Hibernate 创建或者更新关联关系时，还可能出现约束违例。

例如我们想为一个已有的用户“Erica”添加一个地址对象：

```
Transaction tx = session.beginTransaction();
TAddress addr = new TAddress();
addr.setTel("1123");
addr.setZipcode("233123");
addr.setAddress("Hongkong");
user.getAddresses().add(addr);
session.save(user); //通过主控对象级联更新
tx.commit();
```

为了完成这个操作，Hibernate 会分两步（两条 SQL）来完成新增 t\_address 记录的操作：

1. save(user) 时：

```
insert into t_address (user_id, address, zipcode, tel)
values (null, "Hongkong", "233123", "1123")
```

2. tx.commit() 时

```
update t_address set user_id="1" , address="Hongkong",
zipcode="233123", tel="1123" where id=2
```

第一条 SQL 用于插入新的地址记录。

第二条 SQL 用于更新 t\_address，将 user\_id 设置为其关联的 user 对象的 id 值。问题就出在这里，数据库中，我们的 t\_address.user\_id 字段为“NOT NULL”型，当 Hibernate 执行第一条语句创建 t\_address 记录时，试图将 user\_id 字段的值设为 null，于是引发了一个约束违例异常：

```
net.sf.hibernate.PropertyValueException: not-null property
references a null or transient value:
org.hibernate.sample.TAddress.userId
```

因为关联方向是单向，关联关系由 TUser 对象维持，而被关联的 addr 对象本身并不知道自己与哪个 TUser 对象相关联，也就是说，addr 对象本身并不知道 user\_id 应该设为什么数值。因此，在保存 addr 时，只能先在关联字段插入一个空值。之后，再由 TUser 对象将自身的 id 值赋予关联字段 addr.user\_id，这个赋值操作导致 addr 对象属性发生变动，在事务提交时，hibernate 会发现这一改变，并通过 update sql 将变动后的数据保存到数据库。

第一个步骤中，企图向数据库的非空字段插入空值，因此导致了约束违例。既然 TUser 对象是主控方，为什么就不能自动先设置好下面的 TAddress 对象的关联字段值再一次做 Insert 操作呢？莫名其妙？

我们可以在设计的时候通过一些手段进行调整，以避免这样的约束违例，如将关联字段设为允许 NULL 值、直接采用数值型字段作为关联（有的时候这样的调整并不可行，很多情况下我们必须针对现有数据库结构进行开发），或者手动为关联字段属性赋一个任意非空值（即使在这里通过手工设置了正确的 user\_id 也没有意义，hibernate 还是会自动再调用一条 Update 语句进行更新）。甚至我们可以将被控方的关联字段从其映射文件中剔除（如将 user\_id 字段的映射从 TAddress.hbm.xml 中剔除）。这样 Hibernate 在生成第一条 insert 语句的时候就不会包含这个字段（数据库会使用字段默认值填充），如：之后 update 语句会

根据主控方的 one-to-many 映射配置中的关联字段去更新被动态方关联字段的内容。在我们这里的例子中，如果将 user\_id 字段从 TAddress.hbm.xml 文件中剔除，Hibernate 在保存数据时会生成下面几条 SQL：

```
1. insert into t_address (address, zipcode, tel) values  
( 'Hongkong', '233123', '1123' )  
2. update t_address set user_id=1 where id=7
```

生成第一条 insert 语句时，没有包含 user\_id 字段，数据库会使用该字段的默认值（如果有的话）进行填充。因此不会引发约束违例。之后，根据第一条语句返回的记录 id，再通过 update 语句对 user\_id 字段进行更新。但是，纵使采用这些权益之计，由于 Hibernate 实现机制中，采用了两条 SQL 进行一次数据插入操作，相对单条 insert，几乎是两倍的性能开销，效率较低，因此，对于性能敏感的系统而言，这样的解决方案所带来的开销可能难以承受。

针对上面的情况，我们想到，如果 addr 对象知道如何获取 user\_id 字段的内容，那么执行 insert 语句的时候直接将数据植入即可。这样不但绕开了约束违例的可能，而且还节省了一条 Update 语句的开销，大幅度提高了性能。双向一对多关系的出现则解决了这个问题。它除了避免约束违例和提高性能的好处之外，还带来另外一个优点，由于建立了双向关联，我们可以在关联双方中任意一方，访问关联的另一方（如可以通过 TAddress 对象直接访问其关联的 TUser 对象），这提供了更丰富灵活的控制手段。

#### ➤ 双向一对多关系

双向一对多关系，实际上是“单向一对多关系”与“多对一关系”的组合。也就是说我们必须在主控方配置单向一对多关系的基础上，在被控方配置多对一关系与其对应。

#### 配置：

上面我们已经大致完成了单向方一对多关系的配置，我们只需在此基础上稍做修改，并对 (t\_address) 的相关属性进行配置即可：

TUser.hbm.xml：

```
<hibernate-mapping>  
  <class  
    name="org.hibernate.sample.TUser"  
    table="t_user"  
    dynamic-update="true"  
    dynamic-insert="true"  
  >  
  ....  
  <set  
    name="addresses"  
    table="t_address"  
    lazy="false"  
    inverse="true" ①  
    cascade="all"  
    sort="unsorted"  
    order-by="zipcode asc"  
  >  
  <key  
    column="user_id"
```

```
>
</key>
<one-to-many
class="org.hibernate.sample.TAddress"
/>
</set>
</class>
</hibernate-mapping>
```

这里与前面不同，inverse 被设为“true”，这意味着 TUser 不再作为主控方，而是将关联关系的维护工作交给关联对象 org.hibernate.sample.TAddress 来完成。这样 TAddress 对象在持久化过程中，就可以主动获取其关联的 TUser 对象的 id，并将其作为自己的 user\_id，之后执行一次 insert 操作即可完成全部工作。

在 one-to-many 关系中，将 many 一方设为主动方（inverse=false）将有助性能的改善。

对应的 xdoclet tag 如下：

```
public class TUser implements Serializable {
.....
private Set addresses = new HashSet();
.....
/**
 * @hibernate.collection-one-to-many
 * class="org.hibernate.sample.TAddress"
 *
 * @hibernate.collection-key column="user_id"
 *
 * @hibernate.set
 * name="addresses"
 * table="t_address"
 * inverse="true"
 * lazy="false"
 * cascade="all"
 * sort="unsorted"
 * order-by="zipcode asc"
 */
public Set getAddresses() {
    return addresses;
}
.....
}

TAddress.hbm.xml:
<hibernate-mapping>
<class
name="org.hibernate.sample.TAddress"
table="t_address"
```

```
dynamic-update="false"
dynamic-insert="false"
>
.....
<many-to-one
name="user" ①
class="org.hibernate.sample.TUser"
cascade="none"
outer-join="auto"
update="true"
insert="true"
access="property"
column="user_id"
not-null="true"
/>
</class>
</hibernate-mapping>
```

① 在 TAddress 对象中新增一个 TUser field “user”，并为其添加对应的 getter/setter 方法。同时删除原有的 user\_id 属性及其映射配置，否则运行期会报字段重复映射错误：“Repeated column in mapping”。

对应 Xdoclet tag:

```
public class TAddress implements Serializable {
.....
private TUser user;
.....
/**
 * @hibernate.many-to-one
 * name="user"
 * column="user_id"
 * not-null="true"
 *
 */
public TUser getUser() {
    return this.user;
}
.....
}
```

再看上面那段代码片断：

```
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("name", "Erica"));
List userList = criteria.list();
TUser user =(TUser)userList.get(0);
Transaction tx = session.beginTransaction();
TAddress addr = new TAddress();
```

```

addr.setTel("1123");
addr.setZipcode("233123");
addr.setAddress("Hongkong");
user.getAddresses().add(addr);
session.save(user); //通过主控对象级联更新
tx.commit();
    
```

尝试运行这段代码，结果凄凉的很，还是约束违例。

为什么会这样，我们已经配置了 TAddress 的 many-to-one 关系，这么看来似乎没什么效果……不过，别忘了上面提到的 inverse 属性，这里我们把 TUser 的 inverse 设为“true”，即指定由对方维护关联关系，在这里也就是由 TAddress 维护关联关系。TUser 既然不再维护关联关系，那么 TAddress 的 user\_id 属性它也自然不会关心，必须由 TAddress 自己去维护 user\_id：

```

.....
TAddress addr = new TAddress();
addr.setTel("1123");
addr.setZipcode("233123");
addr.setAddress("Hongkong");
addr.setUser(user); //设置关联的TUser对象
user.getAddresses().add(addr);
session.save(user); //级联更新
.....
    
```

观察 Hibernate 执行过程中调用的 SQL 语句：

```

insert into t_address (user_id, address, zipcode, tel) values
(1, 'Hongkong', '233123', '1123')
    
```

正如我们所期望的，保存工作通过单条 Insert 语句的执行来完成。

many-to-one 节点有以下属性：

属性	描述	类型	必须
name	映射属性	Text	Y
column	关联字段。	Text	N
class	类名 默认为映射属性所属类型	Text	N
cascade	操作级联 (cascade) 关系。 可选值： all：所有情况下均进行级联操作。 none：所有情况下均不进行级联操作。 save-update：在执行 save-update 时进行级联操作。 delete：在执行 delete 时进行级联操作。	Text	N

update	是否对关联字段进行Update操作 默认: true	Bool	N
insert	是否对关联字段进行Insert 操作 默认: true	Bool	N
outer-join	是否使用外联接。 true: 总是使用outer-join false: 不使用outer-join auto(默认) : 如果关联对象没有采用Proxy机制, 则使用outer-join.	Text	N
property-ref	用于与主控类相关联的属性的名称。 默认为关联类的主键属性名。 这里我们通过主键进行关联, 所以采用默认值即可。如果关联并非建立在主键之间, 则可通过此参数指定关联属性。	Text	N
access	属性值的读取方式。 可选项: field property (默认) ClassName	Text	N

级联与关联关系的差别?

### 8.4.3. 多对多关联

Hibernate关联关系中相对比较特殊的就是多对多关联, 多对多关联与一对关联和一对多关联不同, 多对多关联需要另外一张映射表用于保存多对多映射信息。由于多对多关联的性能不佳(由于引入了中间表, 一次读取操作需要反复数次查询), 因此在设计中应该避免大量使用。同时, 在对多对关系中, 应根据情况, 采取延迟加载(Lazy Loading 参见后续章节)机制来避免无谓的性能开销。

在一个权限管理系统中, 一个常见的多对多的映射关系就是Group 与Role, 以及Role与Privilege之间的映射。

- Group代表“组”(如“业务主管”);
  - Role代表“角色”(如“出纳”、“财务”);
  - Privilege 代表某个特定资源的访问权限(如“修改财务报表”, “查询财务报表”)。
- 这里我们以Group和Role之间的映射为例:
- 一个Group中包含了多个Role, 如某个“业务主管”拥有“出纳”和“财务”的双重角色。
  - 而一个Role也可以属于不同的Group。

#### 配置:

在我们的实例中, TRole 和TPrivilege 对应数据库中的t\_role、t\_privilege表。TGroup.hbm.xml中关于多对多关联的配置片断:

```
<hibernate-mapping>
```

```
<class
name="org.hibernate.sample.TGroup"
table="t_group"
dynamic-update="false"
dynamic-insert="false"
>
.....
<set
name="roles"
table="t_group_role" ①
lazy="false"
inverse="false"
cascade="save-update" ②
>
<key
column="group_id" ③
>
</key>
<many-to-many
class="org.hibernate.sample.TRole"
column="role_id" ④
/>
</set>
</class>
</hibernate-mapping>
```

① 这里为t\_group 和t\_role之间的映射表。

② 一般情况下，cascade应该设置为“`save-update`”，对于多对多逻辑而言，很少出现删除一方需要级联删除所有关联数据的情况，如删除一个Group，一般不会删除其中包含的Role（这些Role 可能还被其他的Group所引用）。反之删除Role一般也不会删除其所关联的所有Group。

③ 映射表中对子t\_group表记录的标识字段。

④ 映射表中对子t\_role表记录的标识字段。

对应的xdoclet tag如下：

```
public class TGroup implements Serializable {
.....
private Set roles = new HashSet();
/**
 * @hibernate.set
 * name="roles"
 * table="t_group_role"
 * lazy="false"
 * inverse="false"
 * cascade="save-update"
 * sort="unsorted"
```

```
*  
* @hibernate.collection-key  
* column="group_id"  
*  
* @hibernate.collection-many-to-many  
* class="org.hibernate.sample.TRole"  
* column="role_id"  
*  
*/  
public Set getRoles() {  
    return roles;  
}  
.....  
}
```

TRole.hbm.xml中关于多对多关联的配置片断:

```
<hibernate-mapping>  
<class  
name="org.hibernate.sample.TRole"  
table="t_role"  
dynamic-update="false"  
dynamic-insert="false"  
>  
.....  
<set  
name="groups"  
table="t_group_role"  
lazy="false"  
inverse="true"  
cascade="save-update"  
sort="unsorted"  
>  
<key  
column="role_id"  
>  
</key>  
<many-to-many  
class="org.hibernate.sample.TGroup"  
column="group_id"  
outer-join="auto"  
>  
</set>  
</class>  
</hibernate-mapping>
```

对应的xdoclet如下:

```

public class TRole implements Serializable {
private Set groups = new HashSet();
.....
/**
*
* @hibernate.set
* name="groups"
* table="t_group_role"
* cascade="save-update"
* inverse="true"
* lazy="false"
*
* @hibernate.collection-key
* column="role_id"
*
* @hibernate.collection-many-to-many
* class="org.hibernate.sample.TGroup"
* column="group_id"
*
*/
public Set getGroups() {
    return groups;
}
}

```

many-to-many节点中各个属性描述:

属性	描述	类型	必须
class	类名 关联目标类。	Text	N
outer-join	是否使用外联接。 true: 总是使用outer-join false: 不使用outer-join auto(默认) : 如果关联对象没有采用Proxy机制, 则使用outer-join.	Text	N

#### 作用:

多对多关系中, 由于关联关系是两张表相互引用, 因此在保存关联状态时必须对双方同时保存。

```

public void testPersist(){
    TRole role1 = new TRole();
    role1.setName("Role1");
    TRole role2 = new TRole();

```

```
role2.setName("Role2");
TRole role3 = new TRole();
role3.setName("Role3");
TGroup group1 = new TGroup();
group1.setName("group1");
TGroup group2 = new TGroup();
group2.setName("group2");
TGroup group3 = new TGroup();
group3.setName("group3");
group1.getRoles().add(role1);
group1.getRoles().add(role2);
group2.getRoles().add(role2);
group2.getRoles().add(role3);
group3.getRoles().add(role1);
group3.getRoles().add(role3);
role1.getGroups().add(group1);
role1.getGroups().add(group3);
role2.getGroups().add(group1);
role2.getGroups().add(group2);
role3.getGroups().add(group2);
role3.getGroups().add(group3);

try {
    Transaction tx = session.beginTransaction();
    //多对多关系必须同时对关联双方进行保存
    session.save(role1);
    session.save(role2);
    session.save(role3);
    session.save(group1);
    session.save(group2);
    session.save(group3);
    tx.commit();
} catch (Exception e) {
    e.printStackTrace();
    Assert.fail(e.getMessage());
}
}
```

上面的代码创建3个TGroup对象和3个TRole对象，并形成了多对多关系。

## 8.5. 数据访问

### 8.5.1. PO 和 VO

PO即 Persistence Object

VO即 Value Object

PO和VO是Hibernate中两个比较关键的概念。

首先，何谓VO，很简单，VO就是一个简单的值对象。

如：

```
TUser user = new TUser();
user.setName("Emma");
```

这里的user就是一个VO。VO只是简单携带了对象的一些属性信息。何谓PO？即纳入Hibernate管理框架中的VO。看下面两个例子：

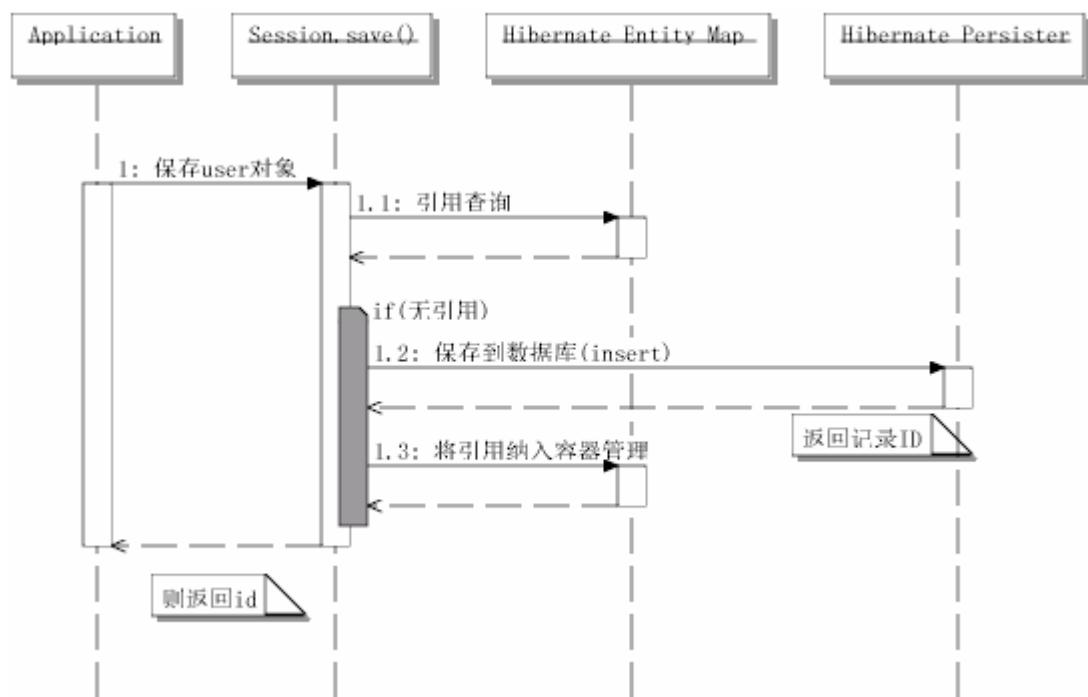
```
TUser user = new TUser();
TUser anotherUser = new TUser();
user.setName("Emma");
anotherUser.setName("Kevin");
//此时user和anotherUser都是vo
Transaction tx = session.beginTransaction();
session.save(user);
//此时的user已经经过Hibernate的处理，成为一个po
//而anotherUser仍然是个vo
tx.commit();
//事务提交之后，库表中已经插入一条用户"Emma"的记录
//对于anotherUser则无任何操作
```

```
Transaction tx = session.beginTransaction();
user.setName("Emma_1"); //PO
anotherUser.setName("Kevin_1"); //VO
tx.commit();
//事务提交之后，PO的状态被固化到数据库中
//也就是说数据库中“Emma”的用户记录已经被更新为“Emma_1”
//此时anotherUser仍然是个普通Java对象，它的属性更改不会
//对数据库产生任何影响
```

另外，通过Hibernate返回的对象也是PO：

```
//由Hibernate返回的PO
TUser user = (TUser)session.load(TUser.class,new Integer(1));
VO经过Hibernate进行处理，就变成了PO。
```

上面的示例代码session.save(user)中，我们把一个vo “user” 传递给Hibernate的Session.save方法进行保存。在save方法中，Hibernate对其进行如下处理：



1. 在当前session所对应的实体容器（Entity Map）中查询是否存在user对象的引用。
2. 如果引用存在，则直接返回user对象id，save过程结束。Hibernate中，针对每个Session有一个实体容器（实际上是一个Map对象），如果此容器中已经保存了目标对象的引用，那么hibernate会认为此对象已经与Session相关联。对于save操作而言，如果对象已经与Session相关联（即已经被加入Session的实体容器中），则无需进行具体的操作。因为之后的Session.flush过程中，Hibernate会对此实体容器中的对象进行遍历，查找出发生变化的实体，生成并执行相应的update语句。
3. 如果引用不存在，则根据映射关系，执行insert操作。
  - a) 在我们这里的示例中，采用了native的id生成机制，因此hibernate会从数据库取得insert操作生成的id并赋予user对象的id属性。
  - b) 将user对象的引用纳入Hibernate的实体容器。
  - c) save过程结束，返回对象id。而Session.load方法中，再返回对象之前，Hibernate就已经将此对象纳入其实体容器中。

VO和PO的主要区别在于：

- VO是独立的Java Object。
- PO是由Hibernate纳入其实体容器（Entity Map）的对象，它代表了与数据库中某条记录对应的Hibernate实体，PO的变化在事务提交时将反应到实际数据库中。

如果一个PO与Session对应的实体容器中分离（如Session关闭后的PO），那么此时，它又会变成一个VO。

由PO、VO的概念，又引申出一些系统层次设计方面的问题。如在传统的MVC架构中，位于Model层的PO，是否允许被传递到其他层面。由于PO的更新最终将被映射到实际数据库中，如果PO在其他层面（如View层）发生了变动，那么可能会对Model层造成意想不到的破坏。

因此，一般而言，应该避免直接PO传递到系统中的其他层面，一种解决办法是，通过一个VO，通过属性复制使其具备与PO相同属性值，并以其为传输媒质（实际上，这个VO被用作Data Transfer Object，即所谓的DTO），将此VO传递给其他层面以实现必须的数据传送。属性复制可以通过Apache Jakarta Commons Beanutils

(<http://jakarta.apache.org/commons/beanutils/>)组件提供的属性批量复制功能，

避免繁复的get/set操作。

下面的例子中，我们把user对象的所有属性复制到anotherUser对象中：

```
TUser user = new TUser();
TUser anotherUser = new TUser();
user.setName("Emma");
user.setUserType(1);
try {
    BeanUtils.copyProperties(anotherUser, user);
    System.out.println("UserName => "
+ anotherUser.getName()
);
    System.out.println("UserType => "
+ anotherUser.getUserType()
);
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
}
```

## 8.5.2. 关于 unsaved-value

在非显示数据保存时，Hibernate将根据这个值来判断对象是否需要保存。所谓显式保存，是指代码中明确调用session 的save、update、saveOrUpdate方法对对象进行持久化。如：

```
session.save(user);
```

而在某些情况下，如映射关系中，Hibernate 根据级联（Cascade）关系对联接类进行保存。此时代码中没有针对级联对象的显示保存语句，需要Hibernate 根据对象当前状态判断是否需要保存到数据库。此时，Hibernate即将根据unsaved-value进行判定。

首先Hibernate会取出目标对象的id。之后，将此值与unsaved-value进行比对，如果相等，则认为目标对象尚未保存，否则，认为对象已经保存，无需再进行保存操作。如：user对象是之前由hibernate从数据库中获取，同时，此user对象的若干个关联对象address 也被加载，此时我们向user 对象新增一个address 对象，此时调用session.save(user)，hibernate会根据unsaved-value判断user对象的数个address关联对象中，哪些需要执行save操作，而哪些不需要。

对于我们新加入的address 对象而言，由于其id（Integer 型）尚未赋值，因此为null，与我们设定的unsaved-value（null）相同，因此hibernate将其视为一个未保存对象，将为其生成insert语句并执行。

这里可能会产生一个疑问，如果“原有”关联对象发生变动（如user的某个“原有”的address 对象的属性发生了变化，所谓“原有”即此address对象已经与user相关联，而不是我们在此过程中为之新增的），此时id值是从数据库中读出，并没有发生改变，自然与unsaved-value（null）也不一样，那么Hibernate是不是就不保存了？

上面关于PO、VO 的讨论中曾经涉及到数据保存的问题，实际上，这里的“保存”，实际上是“insert”的概念，只是针对新关联对象的加入，而非数据库中原有关联对象的“update”。所谓新关联对象，一般情况下可以理解为未与Session 发生关联的VO。而“原有”关联对象，则是PO。如上面关于PO、VO的讨论中所述：

对于`save`操作而言，如果对象已经与Session相关联（即已经被加入Session的实体容器中），则无需进行具体的操作。因为之后的`Session.flush`过程中，Hibernate会对此实体容器中的对象进行遍历，查找出发生变化的实体，生成并执行相应的`update`语句。

### 8.5.3. Inverse 和 Cascade

`Inverse`，直译为“反转”。在Hibernate语义中，`Inverse`指定了关联关系中的方向。关联关系中，`inverse="false"`的为主动方，由主动方负责维护关联关系。具体可参见一对多关系中的描述。

而`Cascade`，译为“级联”，表明对象的级联关系，如`TUser`的`Cascade`设为`all`，就表明如果发生对`user`对象的操作，需要对`user`所关联的对象也进行同样的操作。如对`user`对象执行`save`操作，则必须对`user`对象相关联的`address`也执行`save`操作。

初学者常常混淆`inverse`和`cascade`，实际上，这是两个互不相关的概念。`Inverse`指的是关联关系的控制方向，而`cascade`指的是层级之间的连锁操作。

### 8.5.4. 延迟加载（Lazy Loading）

为了避免一些情况下，关联关系所带来的无谓的性能开销。Hibernate引入了延迟加载的概念。如，示例中`user`对象在加载的时候，会同时读取其所关联的多个地址（`address`）对象，对于需要对`address`进行操作的应用逻辑而言，关联数据的自动加载机制的确非常有效。但是，如果我们只是想要获得`user`的性别（`sex`）属性，而不关心`user`的地址（`address`）信息，那么自动加载`address`的特性就显得多余，并且造成了极大的性能浪费。为了获得`user`的性别属性，我们可能还要同时从数据库中读取数条无用的地址数据，这导致了大量无谓的系统开销。延迟加载特性的出现，正是为了解决这个问题。

所谓延迟加载，就是在需要数据的时候，才真正执行数据加载操作。对于我们这里的`user`对象的加载过程，也就意味着，加载`user`对象时只针对其本身的属性，而当我们需要获取`user`对象所关联的`address`信息时（如执行`user.getAddresses`时），才真正从数据库中加载`address`数据并返回。

我们将前面一对多关系中的`lazy`属性修改为`true`，即指定了关联对象采用延迟加载：

```
<hibernate-mapping>
<class
  name="org.hibernate.sample.TUser"
  table="t_user"
  dynamic-update="true"
  dynamic-insert="true"
>
.....
<set
  name="addresses"
  table="t_address"
  lazy="true"
  inverse="false"
  cascade="all"
  sort="unsorted"
  order-by="zipcode asc"
>
```

```
<key  
column="user_id"  
>  
</key>  
<one-to-many  
class="org.hibernate.sample.TAddress"  
>  
</set>  
.....  
</class>  
</hibernate-mapping>  
尝试执行以下代码：  
Criteria criteria = session.createCriteria(TUser.class);  
criteria.add(Expression.eq("name", "Erica"));  
List userList = criteria.list();  
TUser user = (TUser)userList.get(0);  
System.out.println("User name => " + user.getName());  
Set hset = user.getAddresses();  
session.close() //关闭Session  
TAddress addr = (TAddress)hset.toArray()[0];  
System.out.println(addr.getAddress());  
运行时抛出异常：  
LazyInitializationException - Failed to lazily initialize a collection -  
no session or session was closed
```

如果我们稍做调整，将`session.close`放在代码末尾，则不会发生这样的问题。这意味着，只有我们实际加载user关联的address时，Hibernate才试图通过`session`从数据库中加载实际的数据集，而由于我们读取address之前已经关闭了`session`，所以报出`session`已关闭的错误。

这里有个问题，如果我们采用了延迟加载机制，但希望在一些情况下，实现非延迟加载时的功能，也就是说，我们希望在Session关闭后，依然允许操作user的addresses属性。如，为了向View层提供数据，我们必须提供一个完整的User对象，包含其所关联的address信息，而这个User对象必须在Session关闭之后仍然可以使用。

`Hibernate.initialize`方法可以通过强制加载关联对象实现这一功能：

```
Hibernate.initialize(user.getAddresses());  
session.close();  
//通过Hibernate.initialize方法强制读取数据  
//addresses对象即可脱离session进行操作  
Set hset = user.getAddresses();  
TAddress addr = (TAddress)hset.toArray()[0];  
System.out.println(addr.getAddress());
```

为了实现透明化的延迟加载机制，hibernate进行了大量努力。其中包括JDKCollection接口的独立实现。

如果我们尝试用`HashSet`强行转化Hibernate返回的`Set`型对象：

```
Set hset = (HashSet)user.getAddresses();
```

就会在运行期得到一个`java.lang.ClassCastException`, 实际上, 此时返回的是一个Hibernate的特定Set实现“`net.sf.hibernate.collection.Set`”对象, 而非传统意义上的JDK Set实现。

这也正是我们为什么在编写POJO时, 必须用JDK Collection接口(如`Set`, `Map`) , 而非特定的JDK Collection实现类(如`HashSet`、`HashMap`) 申明Collection属性的原因。

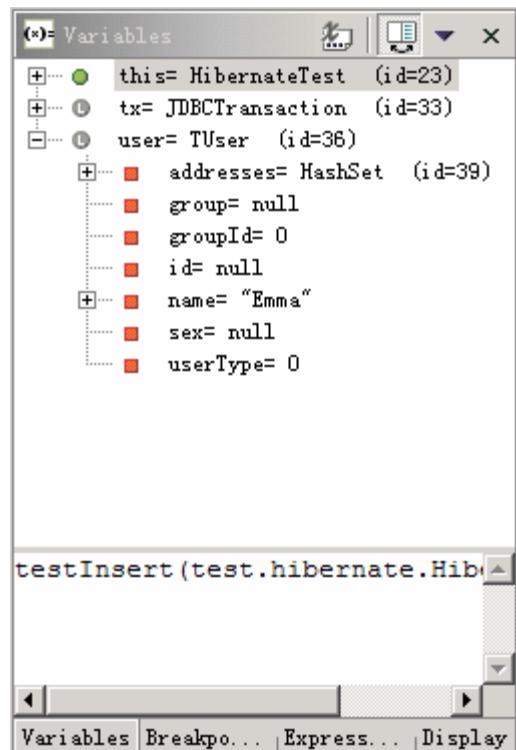
回到前面TUser类的定义:

```
public class TUser implements Serializable {
    .....
    private Set addresses = new HashSet();
}
```

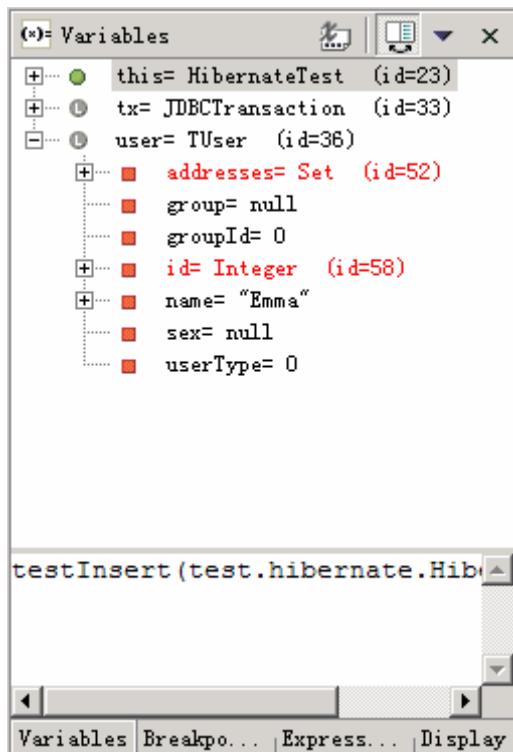
我们通过`Set`接口, 申明了一个`addresses`属性, 并创建了一个`HashSet`作为`addresses`的初始实例, 以便我们创建TUser实例后, 就可以为其添加关联的`address`对象:

```
TUser user = new TUser();
TAddress addr = new TAddress();
addr.setAddress("Hongkong");
user.getAddresses().add(addr);
session.save(user);
```

此时, 这里的`addresses`属性还是一个`HashSet`对象, 其中包含了一个`address`对象的引用。那么, 当调用`session.save(user)`时, Hibernate是如何处理这个`HashSet`型属性的呢? 通过Eclipse的Debug窗口, 我们可以看到`session.save`方法执行前后`user`对象发生的变化:



`session.save`方法之前的`user`对象



session.save方法之后的user对象

可以看到，user对象在通过Hibernate处理之后已经发生了变化。

首先，由于insert操作，Hibernate获得数据库产生的id值（在我们的例子中，采用native方式的主键生成机制），并填充到user对象的id属性。这个变化比较容易理解。

另一方面，Hibernate使用了自己的Collection实现

“net.sf.hibernate.collection.Set”对user中的HashSet型addresses属性进行了替换，并用数据对其进行填充，保证新的addresses与原有的addresses包含同样的实体元素。由于拥有自身的Collection实现，Hibernate就可以在Collection层从容的实现延迟加载特性。只有程序真正读取这个Collection时，才激发底层实际的数据库操作。

### 8.5.5. 事务管理

Hibernate 是 JDBC 的轻量级封装，本身并不具备事务管理能力。在事务管理层，Hibernate 将其委托给底层的 JDBC 或者 JTA，以实现事务管理和调度功能。

Hibernate的默认事务处理机制基于JDBC Transaction。我们也可以通过配置文件设定采用JTA作为事务管理实现：

```
<hibernate-configuration>
<session-factory>
.....
<property name="hibernate.transaction.factory_class">
net.sf.hibernate.transaction.JTATransactionFactory
<!--net.sf.hibernate.transaction.JDBCTransactionFactory-->
</property>
.....
</session-factory>
</hibernate-configuration>
```

## 8.5.6. 基于 JDBC 的事务管理

将事务管理委托给JDBC 进行处理无疑是最简单的实现方式，Hibernate 对于JDBC事务的封装也极为简单。我们来看下面这段代码：

```
session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
.....
tx.commit();
```

从JDBC层面而言，上面的代码实际上对应着：

```
Connection dbconn = getConnection();
dbconn.setAutoCommit(false);
.....
dbconn.commit();
```

就这么简单，Hibernate并没有做更多的事情（实际上也没法做更多的事情），只是将这样的JDBC代码进行了封装而已。这里要注意的是，在`sessionFactory.openSession()`中，hibernate会初始化数据库连接，与此同时，将其`AutoCommit` 设为关闭状态 (`false`)。而其后，在`Session.beginTransaction` 方法中，Hibernate 会再次确认`Connection` 的`AutoCommit` 属性被设为关闭状态（为了防止用户代码对`session` 的`Connection.AutoCommit`属性进行修改）。

这也就是说，我们一开始从`SessionFactory`获得的`session`，其自动提交属性就已经被关闭 (`AutoCommit=false`)，下面的代码将不会对数据库产生任何效果：

```
session = sessionFactory.openSession();
session.save(user);
session.close();
```

这实际上相当于 JDBC Connection的`AutoCommit`属性被设为`false`，执行了若干JDBC操作之后，没有调用`commit`操作即将`Connection`关闭。如果要使代码真正作用到数据库，我们必须显式的调用`Transaction`指令：

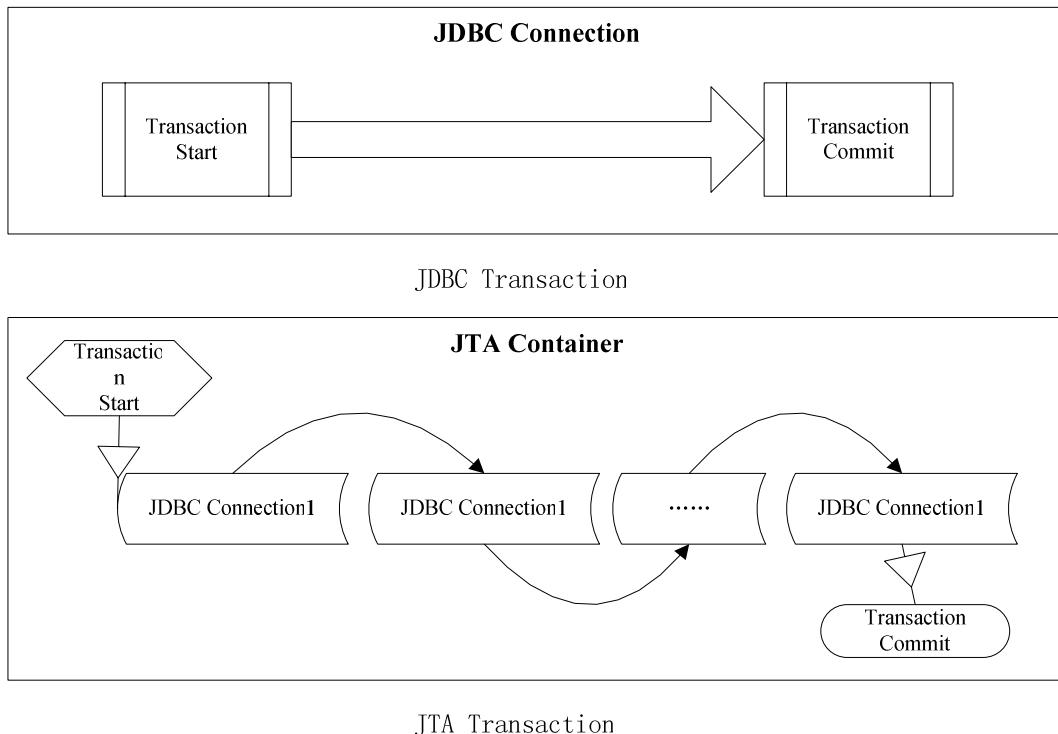
```
session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(user);
tx.commit();
session.close();
```

## 8.5.7. 基于 JTA 的事务管理

JTA 提供了跨Session 的事务管理能力。这一点是与JDBC Transaction 最大的差异。JDBC 事务由Connnection管理，也就是说，事务管理实际上是在JDBC Connection中实现。事务周期限于Connection的生命周期之类。同样，对于基于JDBC Transaction的Hibernate 事务管理机制而言，事务管理在Session 所依托的JDBC Connection中实现，事务周期限于Session的生命周期。

JTA 事务管理则由 JTA 容器实现，JTA 容器对当前加入事务的众多Connection 进行调度，实现其事务性要求。JTA的事务周期可横跨多个JDBC Connection生命周期。同样对于基于JTA 事务的Hibernate而言，JTA事务横跨可横跨多个Session。

下面这幅图形象的说明了这个问题：



图中描述的是 JDBC Connection 与事务之间的关系，而 Hibernate Session 在这里与 JDBC Connection 具备同等的逻辑含义。从上图中我们可以看出，JTA 事务是由 JTA Container 维护，而参与事务的 Connection 无需对事务管理进行干涉。这也就是说，如果采用 JTA Transaction，我们不应该再调用 Hibernate 的 Transaction 功能。上面基于 JDBC Transaction 的正确代码，这里就会产生问题：

```

public class ClassA{
    public void saveUser(User user){
        session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        session.save(user);
        tx.commit();
        session.close();
    }
}

public class ClassB{
    public void saveOrder(Order order){
        session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        session.save(order);
        tx.commit();
        session.close();
    }
}

public class ClassC{
    public void save(){
}

```

```
.....  
UserTransaction tx = new InitialContext().lookup(".....");  
ClassA.save(user);  
ClassB.save(order);  
tx.commit();  
.....  
}  
}
```

这里有两个类ClassA和ClassB，分别提供了两个方法：saveUser和saveOrder，用于保存用户信息和订单信息。在ClassC中，我们接连调用了ClassA.saveUser方法和ClassB.saveOrder方法，同时引入了JTA中的UserTransaction以实现ClassC.save方法中的事务性。

问题出现了，ClassA 和ClassB 中分别都调用了Hibernate 的Transaction 功能。在Hibernate 的JTA 封装中，Session.beginTransaction 同样也执行了InitialContext.lookup方法获取UserTransaction实例，Transaction.commit方法同样也调用了UserTransaction.commit方法。实际上，这就形成了两个嵌套的JTA Transaction：ClassC 申明了一个事务，而在ClassC 事务周期内，ClassA 和ClassB也企图申明自己的事务，这将导致运行期错误。

因此，如果决定采用JTA Transaction，应避免再重复调用Hibernate 的Transaction功能，上面的代码修改如下：

```
public class ClassA{  
    public void save(TUser user){  
        session = sessionFactory.openSession();  
        session.save(user);  
        session.close();  
    }  
    .....  
}  
  
public class ClassB{  
    public void save (Order order){  
        session = sessionFactory.openSession();  
        session.save(order);  
        session.close();  
    }  
    .....  
}  
  
public class ClassC{  
    public void save(){  
        .....  
        UserTransaction tx = new InitialContext().lookup(".....");  
        classA.save(user);  
        classB.save(order);  
        tx.commit();  
        .....  
}
```

```
}
```

```
}
```

上面代码中的ClassC.save方法，也可以改成这样：

```
public class ClassC{
public void save(){
.....
session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
classA.save(user);
classB.save(order);
tx.commit();
.....
}
}
```

实际上，这是利用Hibernate来完成启动和提交UserTransaction的功能，但这样的做法比原本直接通过InitialContext获取UserTransaction 的做法消耗了更多的资源，得不偿失。

在EJB 中使用JTA Transaction 无疑最为简便，我们只需要将save 方法配置为JTA事务支持即可，无需显式申明任何事务，下面是一个Session Bean的save方法，它的事务属性被申明为“Required”，EJB容器将自动维护此方法执行过程中的事务：

```
/**
* @ejb.interface-method
* view-type="remote"
*
* @ejb.transaction type = "Required"
*/
public void save(){
//EJB环境中，通过部署配置即可实现事务申明，而无需显式调用事务
classA.save(user);
classB.save(log);
}//方法结束时，如果没有异常发生，则事务由EJB容器自动提交。
```

## 8.5.8. 锁 (locking)

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，我们希望针对某个cut-off时间点的数据进行处理，而不希望在结算进行过程中（可能是几秒钟，也可能是几个小时），数据再发生变化。此时，我们就需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，在这里，也就是所谓的“锁”，即给我们选定的目标数据上锁，使其无法被其他程序修改。

Hibernate支持两种锁机制：即通常所说的“悲观锁（Pessimistic Locking）”和“乐观锁（Optimistic Locking）”。

## 8.5.9. 悲观锁（Pessimistic Locking）

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实

现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

一个典型的倚赖数据库的悲观锁调用：

```
select * from account where name='Erica' for update
```

这条sql语句锁定了account表中所有符合检索条件(name='Erica')的记录。本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。Hibernate的悲观锁，也是基于数据库的锁机制实现。

下面的代码实现了对查询记录的加锁：

```
String hqlStr =  
"from TUser as user where user.name='Erica'";  
Query query = session.createQuery(hqlStr);  
query.setLockMode("user", LockMode.UPGRADE); //加锁  
List userList = query.list(); //执行查询，获取数据  
query.setLockMode对查询语句中，特定别名所对应的记录进行加锁（我们为TUser类指定了一个别名“user”），这里也就是对返回的所有user记录进行加锁。观察运行期Hibernate生成的SQL语句：
```

```
select tuser0_.id as id, tuser0_.name as name, tuser0_.group_id  
as group_id, tuser0_.user_type as user_type, tuser0_.sex as sex  
from t_user tuser0_ where (tuser0_.name='Erica') for update
```

这里Hibernate通过使用数据库的for update子句实现了悲观锁机制。

Hibernate的加锁模式有：

- LockMode.NONE：无锁机制。
- LockMode.WRITE：Hibernate在Insert和Update记录的时候会自动获取。
- LockMode.READ：Hibernate在读取记录的时候会自动获取。

以上这三种锁机制一般由Hibernate内部使用，如Hibernate为了保证Update过程中对象不会被外界修改，会在save方法实现中自动为目标对象加上WRITE锁。

- LockMode.UPGRADE：利用数据库的for update子句加锁。
- LockMode.UPGRADE\_NOWAIT：Oracle的特定实现，利用Oracle的for update nowait子句实现加锁。

上面这两种锁机制是我们在应用层较为常用的，加锁一般通过以下方法实现：

```
Criteria.setLockMode  
Query.setLockMode  
Session.lock
```

注意，只有在查询开始之前（也就是Hibernate生成SQL之前）设定加锁，才会真正通过数据库的锁机制进行加锁处理，否则，数据已经通过不包含for update子句的Select SQL加载进来，所谓数据库加锁也就无从谈起。

## 8.5.10. 乐观锁 (Optimistic Locking)

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

如一个金融系统，当某个操作员读取用户的数据，并在读出的用户数据的基础上进行修改时（如更改用户帐户余额），如果采用悲观锁机制，也就意味着整个操作过程中（从操作员读出数据、开始修改直至提交修改结果的全过程，甚至还包括操作员中途去煮咖啡的时间），数据库记录始

终处于加锁状态，可以想见，如果面对几百上千个并发，这样的情况将导致怎样的后果。乐观锁机制在一定程度上解决了这个问题。

乐观锁，大多是基于数据版本（version）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。

读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

对于上面修改用户帐户信息的例子而言，假设数据库中帐户信息表中有一个version字段，当前值为1；而当前帐户余额字段（balance）为\$100。

- 1 操作员A 此时将其读出（version=1），并从其帐户余额中扣除\$50（\$100-\$50）。
- 2 在操作员A操作的过程中，操作员B也读入此用户信息（version=1），并从其帐户余额中扣除\$20（\$100-\$20）。
- 3 操作员A完成了修改工作，将数据版本号加一（version=2），连同帐户扣除后余额（balance=\$50），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录version更新为2。
- 4 操作员B完成了操作，也将版本号加一（version=2）试图向数据库提交数据（balance=\$80），但此时比对数据库记录版本时发现，操作员B提交的数据版本号为2，数据库记录当前版本也为2，不满足“提交版本必须大于记录当前版本才能执行更新”的乐观锁策略，因此，操作员B 的提交被驳回。

这样，就避免了操作员B 用基于version=1 的旧数据修改的结果覆盖操作员A的操作结果的可能。

从上面的例子可以看出，乐观锁机制避免了长事务中的数据库加锁开销（操作员A和操作员B操作过程中，都没有对数据库数据加锁），大大提升了大并发量下的系统整体性能表现。

需要注意的是，乐观锁机制往往基于系统中的数据存储逻辑，因此也具备一定的局限性，如在上例中，由于乐观锁机制是在我们的系统中实现，来自外部系统的用户余额更新操作不受我们系统的控制，因此可能会造成脏数据被更新到数据库中。在系统设计阶段，我们应该充分考虑到这些情况出现的可能性，并进行相应调整（如将乐观锁策略在数据库存储过程中实现，对外只开放基于此存储过程的数据更新途径，而不是将数据库表直接对外公开）。

Hibernate 在其数据访问引擎中内置了乐观锁实现。如果不考虑外部系统对数据库的更新操作，利用Hibernate提供的透明化乐观锁实现，将大大提升我们的生产力。

Hibernate中可以通过class描述符的optimistic-lock属性结合version描述符指定。现在，我们为之前示例中的TUser加上乐观锁机制。

1. 首先为TUser的class描述符添加optimistic-lock属性：

```
<hibernate-mapping>
<class
  name="org.hibernate.sample.TUser"
  table="t_user"
  dynamic-update="true"
  dynamic-insert="true"
  optimistic-lock="version">
<!--
.....
</class>
```

```
</hibernate-mapping>  
optimistic-lock属性有如下可选取值:
```

➤ none

无乐观锁

➤ version

通过版本机制实现乐观锁

➤ dirty

通过检查发生变动过的属性实现乐观锁

➤ all

通过检查所有属性实现乐观锁其中通过version实现的乐观锁机制是Hibernate官方推荐的乐观锁实现，同时也是Hibernate中，目前唯一在数据对象脱离Session发生修改的情况下依然有效的锁机制。因此，一般情况下，我们都选择version方式作为Hibernate乐观锁实现机制。

## 2. 添加一个Version属性描述符

```
<hibernate-mapping>  
<class  
name="org.hibernate.sample.TUser"  
table="t_user"  
dynamic-update="true"  
dynamic-insert="true"  
optimistic-lock="version"  
>  
<id  
name="id"  
column="id"  
type="java.lang.Integer"  
>  
<generator class="native">  
</generator>  
</id>  
<version  
column="version"  
name="version"  
type="java.lang.Integer"  
>  
.....  
</class>  
</hibernate-mapping>
```

*注意version 策点必须出现在ID 策点之后。*

这里我们声明了一个version属性，用于存放用户的版本信息，保存在TUser表的version字段中。

此时如果我们尝试编写一段代码，更新TUser表中记录数据，如：

```
Criteria criteria = session.createCriteria(TUser.class);  
criteria.add(Expression.eq("name", "Erica"));
```

```
List userList = criteria.list();
TUser user =(TUser)userList.get(0);
Transaction tx = session.beginTransaction();
user.setUserType(1); //更新UserType字段
tx.commit();
```

每次对TUser进行更新的时候，我们可以发现，数据库中的version都在递增。而如果我们尝试在tx.commit之前，启动另外一个Session，对名为Erica的用户进行操作，以模拟并发更新时的情形：

```
Session session= getSession();
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("name", "Erica"));
Session session2 = getSession();
Criteria criteria2 = session2.createCriteria(TUser.class);
criteria2.add(Expression.eq("name", "Erica"));
List userList = criteria.list();
List userList2 = criteria2.list();
TUser user =(TUser)userList.get(0);
TUser user2 =(TUser)userList2.get(0);
Transaction tx = session.beginTransaction();
Transaction tx2 = session2.beginTransaction();
user2.setUserType(99);
tx2.commit();
user.setUserType(1);
tx.commit();
```

执行以上代码，代码将在tx.commit()处抛出StaleObjectStateException异常，并指出版本检查失败，当前事务正在试图提交一个过期数据。通过捕捉这个异常，我们就可以在乐观锁校验失败时进行相应处理。

### 8.5.11. Hibernate 分页

数据分页显示，在系统实现中往往带来了较大的工作量，对于基于JDBC的程序而言，不同数据库提供的分页(部分读取)模式往往各不相同，也带来了数据库间可移植性上的问题。Hibernate中，通过对不同数据库的统一接口设计，实现了透明化、通用化的分页实现机制。

我们可以通过Criteria.setFirstResult和Criteria.setFetchSize方法设定分页范围，如：

```
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("age", "20"));
//从检索结果中获取第100条记录开始的20条记录
criteria.setFirstResult(100);
criteria.setFetchSize(20);
```

同样，Query接口也提供了与其一致的方法。

Hibernate中，抽象类net.sf.hibernate.dialect指定了所有底层数据库的对外统一接口。通过针对不同数据库提供相应的dialect实现，数据库之间的差异性得以消除，从而为上层机制提供了透明的、数据库无关的存储层基础。

对于分页机制而言，dialect中定义了一个方法如下：

```
Public String getLimitString(  
String querySelect,  
boolean hasOffset  
)
```

此方法用于在现有Select语句基础上，根据各数据库自身特性，构造对应的记录返回限定子句。如MySQL中对应的记录限定子句为Limit，而Oracle中，可通过rownum子句实现。我们来看MySQLDialect中的getLimitString实现：

```
public String getLimitString(String sql, boolean hasOffset) {  
return new StringBuffer( sql.length()+20 )  
.append(sql)  
.append( hasOffset ? " limit ?, ?" : " limit ?" )  
.toString();  
}
```

从上面可以看到，MySQLDialect.getLimitString方法的实现实际上是在给定的Select语句后追加MySQL所提供的专有SQL子句limit来实现。下面是Oracle9Dialect 中的getLimitString 实现，其中通过Oracle特有的rownum子句实现了数据的部分读取。

```
public String getLimitString(String sql, boolean hasOffset)  
{  
StringBuffer pagingSelect =  
new StringBuffer( sql.length()+100 );  
if (hasOffset) {  
pagingSelect.append(  
"select * from ( select row_.*, rownum rownum_ from ( "  
);  
}  
else {  
pagingSelect.append("select * from ( ");  
}  
pagingSelect.append(sql);  
if (hasOffset) {  
pagingSelect.append(  
" ) row_ where rownum <= ? ) where rownum_ > ?"  
);  
}  
else {  
pagingSelect.append(" ) where rownum <= ?");  
}  
return pagingSelect.toString();  
}
```

大多数主流数据库都提供了数据部分读取机制，而对于某些没有提供相应机制的数据库而言，Hibernate也通过其他途径实现了分页，如通过Scrollable ResultSet，如果JDBC 不支持Scrollable ResultSet，Hibernate 也会自动通过ResultSet 的next 方法进行记录定位。这样，Hibernate 通过底层对分页机制的良好封装，使得开发人员无需关心数据分页的细节实现，将数据逻辑和存储逻辑分离开来，在提高生产效率的同时，也大大加强了系统在不同数据库平台之间的可移植性。

### 8.5.12. Cache 管理

Cache往往是提高系统性能的最重要的手段。在笔者记忆中，DOS时代SmartDrv2所带来的磁盘读写性能提升还历历在目（记得95年时安装Windows 3.0，在没有SmartDrv常驻内存的情况下，大概需要15分钟左右，而加载了SmartDrv，只需要2分钟即可完成整个安装过程）。

Cache对于大量依赖数据读取操作的系统而言（典型的，如114查号系统）尤为重要，在大并发量的情况下，如果每次程序都需要向数据库直接做查询操作，所带来的性能开销显而易见，频繁的网络传输、数据库磁盘的读写操作（大多数数据库本身也有Cache，但即使如此，访问数据库本身的开销也极为可观），这些都大大降低了系统的整体性能。

此时，如果能把数据在本地内存中保留一个镜像，下次访问时只需从内存中直接获取，那么显然可以带来显著的性能提升（可能是几倍，甚至几十倍的整体读取性能提升）。引入Cache机制的难点是如何保证内存中数据的有效性，否则脏数据的出现将给系统带来难以预知的严重后果。

Hibernate 中实现了良好的Cache 机制，我们可以借助Hibernate 内部的Cache迅速提高系统数据读取性能。

**需要注意的是：**Hibernate做为一个应用级的数据访问层封装，只能在其作用范围内保持Cache 中数据的有效性，也就是说，在我们的系统与第三方系统共享数据库的情况下，Hibernate 的Cache机制可能失效。

Hibernate 在本地JVM 中维护了一个缓冲池，并将从数据库获得的数据保存到池中以供下次重复使用（如果在Hibernate中数据发生了变动，Hibernate同样也会更新池中的数据版本）。此时，如果有第三方系统对数据库进行了更改，那么，Hibernate并不知道数据库中的数据已经发生了变化，也就是说，池中的数据还是修改之前的版本，下次读取时，Hibernate会将此数据返回给上层代码，从而导致潜在的问题。

外部系统的定义，并非限于本系统之外的第三方系统，即使在本系统中，如果出现了绕过 Hibernate数据存储机制的其他数据存取手段，那么Cache的有效性也必须细加考量。如，在同一套系统中，基于Hibernate和基于JDBC的两种数据访问方式并存，那么通过JDBC更新数据库的时候，Hibernate同样无法获知数据更新的情况，从而导致脏数据的出现。

基于Java 的Cache 实现，最简单的莫过于HashTable，hibernate 提供了基于Hashtable 的Cache 实现机制，不过，由于其性能和功能上的局限，仅供开发调试中使用。同时，Hibernate 还提供了面向第三方Cache 实现的接口，如JCS、EHCache、OSCache、JBoss Cache、SwarmCache等。

Hibernate中的Cache大致分为两层，第一层Cache在Session实现，属于事务级数据缓冲，一旦事务结束，这个Cache 也就失效。此层Cache 为内置实现，无需我们进行干涉。第二层Cache，是Hibernate 中对其实例范围内的数据进行缓存的管理容器。也是这里我们讨论的主题。

Hibernate早期版本中采用了JCS (Java Caching System —Apache Turbine项目中的一个子项目) 作为默认的第二层Cache实现。由于JCS的发展停顿，以及其内在的一些问题（在某些情况下，可能导致内存泄漏以及死锁），新版本的Hibernate已经将JCS去除，并用EHCache 作为其默认的第二级Cache实现。相对JCS， EHCache更加稳定，并具备更好的缓存调度性能，缺陷是目前还无法做到分布式缓存，如果我们的系统需要在多台设备上部署，并共享同一个数据库，必须使用支持分布式缓存的Cache实现（如JCS、JBossCache）以避免出现不同系统实例之间缓存不一致而导致脏数据的情况。

Hibernate对Cache进行了良好封装，透明化的Cache机制使得我们在上层结构的实现中无需面对繁琐的Cache维护细节。

目前Hibernate支持的Cache实现有：

名称	类	集 群 支持	查询缓冲
HashTable	net.sf.hibernate.cache.HashtableCacheProvider	N	Y
EHCache	net.sf.ehcache.hibernate.Provider	N	Y
OSCache	net.sf.hibernate.cache.OSCacheProvider	N	Y
SwarmCache	net.sf.hibernate.cache.SwarmCacheProvider	Y	
JBossCache	net.sf.hibernate.cache.TreeCacheProvider	Y	

其中SwarmCache和JBossCache均提供了分布式缓存实现（Cache集群）。

(注：最新版本的OSCache也提供了分布式实现。)

其中SwarmCache 提供的是invalidation 方式的分布式缓存，即当集群中的某个节点更新了缓存中的数据，即通知集群中的其他节点将此数据废除，之后各个节点需要用到这个数据的时候，会重新从数据库中读入并填充到缓存中。

而JBossCache提供的是Reapplication式的缓存，即如果集群中某个节点的数据发生改变，此节点会将发生改变的数据的最新版本复制到集群中的每个节点中以保持所有节点状态一致。

使用第二层Cache，需要在hibernate.cfg.xml配置以下参数（以EHCache为例）：

```

<hibernate-configuration>
<session-factory>
    ...
    <property name="hibernate.cache.provider_class">
        net.sf.ehcache.hibernate.Provider
    </property>
    ...
</session-factory>
</hibernate-configuration>

```

另外还需要针对Cache实现本身进行配置，如EHCache的配置文件：

```

<ehcache>
    <diskStore path="java.io.tmpdir"/>
    <defaultCache
        maxElementsInMemory="10000" //Cache中最大允许保存的数据数量
        eternal="false" //Cache中数据是否为常量
        timeToIdleSeconds="120" //缓存数据钝化时间
        timeToLiveSeconds="120" //缓存数据的生存时间
        overflowToDisk="true" //内存不足时，是否启用磁盘缓存
    />
</ehcache>

```

其中“//”开始的注释是笔者追加，实际配置文件中不应出现。

之后，需要在我们的映射文件中指定各个映射实体的Cache策略：

```
<class name=" org.hibernate.sample.TUser" .... >
<cache usage="read-write" />
...
<set name="addresses" .... >
<cache usage="read-only" />
...
</set>
</class>
```

缓冲描述符cache可用于描述映射类和集合属性。

上例中，class节点下的cache描述符指定了针对类TUser的缓存策略为“read-write”，即缓冲中的TUser实例为可读可写，而集合属性addresses的缓存策略为只读。

cache usage可选值有以下几种：

1. read-only

只读。

2. read-write

可读可写。

3. nonstrict-read-write

如果程序对并发数据修改要求不是非常严格，只是偶尔需要更新数据，可以采用本选项，以减少无谓的检查，获得较好的性能。

4. transactional

事务性cache。在事务性Cache中，Cache的相关操作也被添加到事务之中，如果由于某种原因导致事务失败，我们可以连同缓冲池中的数据一同回滚到事务开始之前的状态。目前Hibernate内置的Cache中，只有JBossCache支持事务性的Cache实现。

不同的Cache实现，支持的usage也各不相同：

名称	read-only	read-write	nonstrict-read-write	transactional
HashTable	Y	Y	Y	
EHCache	Y	Y	Y	
OSCache	Y	Y	Y	
SwarmCache	Y	Y		
JBossCache	Y			Y

配置好Cache之后，Hibernate在运行期会自动应用Cache机制，也就是说，我们对PO的更新，会自动同步到Cache中去，而数据的读取，也会自动化的优先从Cache中获取，对于上层逻辑代码而言，有没有应用Cache机制，并没有什么影响。

需要注意的是Hibernate的数据库查询机制。我们从查询结果中取出数据的时候，用的最多的是两个方法：

```
Query.list();
```

```
Query.iterate();
```

对于list方法而言，实际上Hibernate是通过一条Select SQL获取所有的记录。并将其读出，填入到POJO中返回。而iterate方法，则是首先通过一条Select SQL 获取所有符合查询条件的记录的id，再对这个id集合进行循环操作，通过单独的Select SQL 取出每个id 所对应的记录，之后填入POJO中返回。

也就是说，对于list 操作，需要一条SQL 完成。而对于iterate 操作，需要n+1条SQL。

看上去iterate方法似乎有些多余，但在不同的情况下确依然有其独特的功效，如对海量数据的查询，如果用list方法将结果集一次取出，内存的开销可能无法承受。

另一方面，对于我们现在的Cache机制而言，list方法将不会从Cache中读取数据，它总是一次性从数据库中直接读出所有符合条件的记录。而iterate 方法因为每次根据id获取数据，这样的实现机制也就为从Cache读取数据提供了可能，hibernate首先会根据这个id 在本地Cache 内寻找对应的数据，如果没找到，再去数据库中检索。如果系统设计中对Cache比较倚重，则请注意编码中这两种不同方法的应用组合，有针对性的改善代码，最大程度提升系统的整体性能表现。

通观以上内容，Hibernate通过对Cache的封装，对上层逻辑层而言，实现了Cache的透明化实现，程序员编码时无需关心数据在Cache中的状态和调度，从而最大化协调了性能和开发效率之间的平衡。

### 8.5.13. Session 管理

无疑，Session是Hibernate运作的灵魂，作为贯穿Hibernate应用的关键，Session中包含了数据库操作相关状态信息。如对JDBC Connection 的维护，数据实体的状态维持等。对Session 进行有效管理的意义，类似JDBC 程序设计中对于JDBC Connection 的调度管理。有效的Session管理机制，是Hibernate应用设计的关键。

大多数情况下，Session 管理的目标聚焦于通过合理的设计，避免Session 的频繁创建和销毁，从而避免大量的内存开销和频繁的JVM垃圾回收，保证系统高效平滑运行。

在各种session 管理方案中， ThreadLocal 模式得到了大量使用。ThreadLocal 是Java 中一种较为特殊的线程绑定机制。通过ThreadLocal存取的数据，总是与当前线程相关，也就是说，JVM 为每个运行的线程，绑定了私有的本地实例存取空间，从而为多线程环境常出现的并发访问问题提供了一种隔离机制。

首先，我们需要知道，SessionFactory负责创建Session，SessionFactory是线程安全的，多个并发线程可以同时访问一个SessionFactory 并从中获取Session 实例。而Session并非线程安全，也就是说，如果多个线程同时使用一个Session实例进行数据存取，则将会导致Session 数据存取逻辑混乱。下面是一个典型的Servlet，我们试图通过一个类变量session 实现Session的重用，以避免每次操作都要重新创建：

```
public class TestServlet extends HttpServlet {  
    private Session session;  
  
    public void doGet( HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
        session = getSession();  
        doSomething();  
        session.flush();  
    }  
  
    public void doSomething(){
```

```
.....//基于session的存取操作  
}  
}
```

代码看上去正确无误，甚至在我们单机测试的时候可能也不会发生什么问题，但这样的代码一旦编译部署到实际运行环境中，接踵而来的莫名其妙的错误很可能使得我们摸不找头脑。

问题出在哪里？

首先，Servlet 运行是多线程的，而应用服务器并不会为每个线程都创建一个Servlet实例，也就是说，TestServlet在应用服务器中只有一个实例（在Tomcat中是这样，其他的应用服务器可能有不同的实现），而这个实例会被许多个线程并发调用，doGet 方法也将被不同的线程反复调用，可想而知，每次调用doGet 方法，这个唯一的TestServlet 实例的session 变量都会被重置，线程A 的运行过程中，其他的线程如果也被执行，那么session的引用将发生改变，之后线程A 再调用session，可能此时的session 与其之前所用的session就不再一致，显然，错误也就如期而至。

ThreadLocal的出现，使得这个问题迎刃而解。

我们对上面的例子进行一些小小的修改：

```
public class TestServlet extends HttpServlet {  
    private ThreadLocal localSession = new ThreadLocal();  
    public void doGet( HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
        localSession.set(getSession());  
        doSomething();  
        session.flush();  
    }  
    public void doSomething(){  
        Session session = (Session)localSession.get();  
        .....//基于session的存取操作  
    }  
}
```

可以看到，localSession 是一个ThreadLocal 类型的对象，在doGet 方法中，我们通过其set 方法将获取的session 实例保存，而在doSomething 方法中，通过get 方法取出session实例。

这也就是ThreadLocal的独特之处，它会为每个线程维护一个私有的变量空间。实际上，其实现原理是在JVM 中维护一个Map，这个Map的key 就是当前的线程对象，而value则是线程通过ThreadLocal.set方法保存的对象实例。当线程调用ThreadLocal.get方法时，ThreadLocal会根据当前线程对象的引用，取出Map中对应的对象返回。

这样，ThreadLocal通过以各个线程对象的引用作为区分，从而将不同线程的变量隔离开来。回到上面的例子，通过应用ThreadLocal 机制，线程A 的session 实例只能为线程A所用，同样，其他线程的session实例也各自从属于自己的线程。这样，我们就实现了线程安全的Session共享机制。

Hibernate官方开发手册的示例中，提供了一个通过ThreadLocal维护Session的好榜样：

```
public class HibernateUtil {  
    private static SessionFactory sessionFactory;  
    static {
```

```
try {
// Create the SessionFactory
sessionFactory = new
Configuration().configure().buildSessionFactory();
} catch (HibernateException ex) {
throw new RuntimeException(
"Configuration problem: " + ex.getMessage(),
ex
);
}
}

public static final ThreadLocal session = new ThreadLocal();
public static Session currentSession() throws HibernateException
{
Session s = (Session) session.get();
// Open a new Session, if this Thread has none yet
if (s == null) {
s = sessionFactory.openSession();
session.set(s);
}
return s;
}
public static void closeSession() throws HibernateException {
Session s = (Session) session.get();
session.set(null);
if (s != null)
s.close();
}
}
```

在代码中，只要借助上面这个工具类获取Session 实例，我们就可以实现线程范围内的Session 共享，从而避免了在线程中频繁的创建和销毁Session 实例。不过注意在线程结束时关闭Session。

同时值得一提的是，新版本的Hibernate在处理Session的时候已经内置了延迟加载机制，只有在真正发生数据库操作的时候，才会从数据库连接池获取数据库连接，我们不必过于担心Session的共享会导致整个线程生命周期内数据库连接被持续占用。

上面的HibernateUtil类可以应用在任何类型的Java程序中。特别的，对于Web程序而言，我们可以借助Servlet2.3规范中新引入的Filter机制，轻松实现线程生命周期内的Session管理（关于Filter的具体描述，请参考Servlet2.3规范）。

Filter的生命周期贯穿了其所覆盖的Servlet（JSP也可以看作是一种特殊的Servlet）及其底层对象。Filter在Servlet被调用之前执行，在Servlet调用结束之后结束。因此，在Filter 中管理Session 对于Web 程序而言就显得水到渠成。下面是一个通过Filter 进行Session管理的典型案例：

```
public class PersistenceFilter implements Filter
{
```

```
protected static ThreadLocal hibernateHolder = new ThreadLocal();
public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
throws IOException, ServletException
{
hibernateHolder.set(getSession());
try
{
.....
chain.doFilter(request, response);
.....
}
finally
{
Session sess = (Session)hibernateHolder.get();
if (sess != null)
{
hibernateHolder.set(null);
try
{
sess.close();
}
catch (HibernateException ex) {
throw new ServletException(ex);
}
}
}
}
}
.....
}
```

通过在doFilter中获取和关闭Session，并在周期内运行的所有对象（Filter链中其余的Filter，及其覆盖的Servlet和其他对象）对此Session实例进行重用，保证了一个HttpRequest处理过程中只占用一个Session，提高了整体性能表现。

在实际设计中，Session的重用做到线程级别一般已经足够，企图通过 HttpSession 实现用户级的 Session 重用反而可能导致其他的问题。凡事不能过火，Session 重用也一样。

# 第9章 轻量级框架 Spring

## 9.1. Spring 概述

### 9.1.1. Spring 是什么？

通常 Spring 是指一个用于构造 JAVA 应用程序的轻量级框架。这句话包含两个方面的意思：第一，你可以采用 Spring 来构造任何程序，这与 Struts 这样的框架有所不同，你不限定于只编写 web 应用。第二，以上解释种的“轻量级”并不意味着类的数量很少，或者发行包的大小，实际商，它指的是 Spring 的核心思想—那就是最少的侵入。

Spring 是轻量级的，意味着你只需要对你的程序代码作很少的改动（假如有的话），而获得 Spring 核心带来的好处，你也可以在任何时候选择抛弃 Spring。注意，上面这句话只针对 Spring 核心本身—很多附加的 Spring 组件，比如数据访问组件，需要与 Spring 框架有相对紧密得多的耦合关系。但是这样的耦合带来的好处显而易见，在后面的章节中我们展示的技术都是为了尽量不侵入你的程序代码。

#### 9.1.1.1. Inverting Control or Injecting Dependencies: 控制反转还是依赖注入

Spring 框架的核心基于“控制反转(Inversion of Control,IoC)”原理。IoC 是一种将组件依赖关系的创建和管理置于程序外部的技术。

假设，类 Foo 依赖于类 Bar 的一个实例来进行某些操作。传统的方式，Foo 使用 new 操作来创建一个 Bar 的实例，或者通过某种工厂类来获得。使用 IoC 方法，Bar 的一个实例（或者其子类的实例）通过某些外部处理过程在运行时动态的传递给 Foo 的。这种在运行时注入依赖的行为方式，使得 IoC 后来被改称为另一个含义更明确的名字：“依赖注入（Dependency Injection,DI）”。在 Spring 的概念范围内，你可以不加区分的使用这两个专用名词，就是说当使用 IoC 的时候换用 DI 总是正确的。

Spring 的 DI 实现是基于两个 Java 核心概念：JavaBean 和 Interface。当你使用 DI 的时候，你可以使得依赖配置与你的代码隔离。JavaBean 提供了一种创建 Java 资源的标准方法，并且这些资源是可以通过标准方式配置的。实际上，任何 Spring 管理的资源都会被应用为 bean。

接口与 DI 是互相受益的技术。针对接口设计与编程有助于应用程序的灵活性，但要把采用接口设计的应用程序的各部分连接起来其复杂度非常高，并且给开发者带来了额外的编码负担。通过采用 DI，为基于接口的设计而编写的辅助代码大大减少。反过来，通过采用接口，你可以获得 DI 的最大好处，因为你的 bean 可以采用任何满足其依赖的接口实现。

在 DI 的语言环境中，Spring 运行更像一个容器，而非框架—为你应用程序的类提供所有所需依赖的实例，它不像 EJB 中创建持久化 Entity Bean 那样需要侵入你的程序。采用 Spring 进行 DI，你所需要作的仅是你的类遵循 JavaBeans 的命名规范—不需要从某个特别的超类继承，或者遵循某种私有的特定的命名规范。使用 DI 时，对你的代码所需要进行的改动至多时让你的 JavaBean 暴露更多的属性，以便动态注入更多的依赖关系。

使用 DI 而非传统的方法的好处：

- 减少“粘合”代码
- 依赖外置化
- 在统一的地方管理依赖

- 提高可测试性
- 鼓励良好的设计

### 9.1.1.2. Beyond Dependency Injection: 依赖注入之外

出类先进的 DI 之外，Spring 内核本身也是杰出的工具。Spring 具有很多精良的附加功能，这些都是基于 DI 的基本原理优雅地设计构建的。Spring 提供一个应用所需的所有层面的功能，从用于数据访问工具的 API，直到先进的 MVC 功能。Spring 具有这些功能最大的好处在于，虽然 Spring 提供了它自己的方案，你可以非常容易的把他们和其他工具或者框架整合在一起。你可以在 Spring 中使用以下的功能：

- Spring 中面向方向编程
- 在 Spring 中访问数据
- 进行事务管理
- 简化与整合 J2EE
- Web 层的 MVC
- 远程访问（Remoting）支持
- Mail 支持
- 计划任务支持
- 简化的异常处理
- 源代码级的 Metadata（元数据）

## 9.1.2. Spring 项目

### 9.1.2.1. Origins of Spring: Spring 起源

Spring 的起源可以追溯到 Rod Johnson 编写的“Expert One-to-One J2EE Design and Development”一书 (Wrox, 2002)。在这本书中，Rod 展示了他的 interface21 框架，他为自己的应用编写了这一框架。这一框架发布到开源世界后，组成了我们现在所知的 Spring 框架基础。

### 9.1.2.2. The Spring Community: Spring 社群

### 9.1.2.3. Spring for Microsoft .Net: 基于微软.NET 的 Spring

### 9.1.2.4. The Spring Rich Client Platform: Spring 富客户端平台

### 9.1.2.5. The Spring IDE

### 9.1.2.6. The Acegi Security System for Spring: 用于 Spring 的 Acegi 安全系统

## 9.1.3. 入门指引

Spring 借助 SourceForge 平台管理开发过程，如果我们需要获得 Spring 的最新版本可以通过以下连接 <http://www.sourceforge.net/project/springframework> 来获得。Spring 的每个版本都以两种形式提供：一种包含有全部依赖关系，另一种不包含。如果只打算使用 Spring 诸多组件

其中一个组件，那么你应该下载不带依赖关系的版本，手工挑选你自己需要的依赖包。不过一般而言我们都使用包含全部依赖关系。另外值得注意的是，和Spring打包在一起的依赖包当初就用来编译改发布包，因此可以确定这些依赖关系包的版本完全匹配。

### 9.1.3.1. The Full Distribution: 完整发布包

所有发布版本都提供 spring.jar 文件，它差不多包含 Spring Framework 类的完整发布包。之所以说“差不多”是因为它实际上并未包含任何 mock 类，这些类随着 Spring 一起发布，作为辅助测试用。因为你发布应用时几乎从不需要同时分发 mock 类；你只是在开发环境下用他们来辅助测试过程。除此之外，spring.jar 文件包含 Spring main 源码树里所有其他类。

### 9.1.3.2. The Component Distributions: 组件发布包

除了 spring.jar 文件，Spring 还包含了前面提及另外 8 个 JAR 文件；其中一个包含 mock 类，另外 7 个包含 Spring framework 的独立组件。下表中列出了这些 jar 文件：

JAR 文件	说明
spring-aop.jar	这个 JAR 文件包含在应用中使用 Spring 的 AOP 特性时所需的所有类。如果打算使用其他基于 AOP 的 Spring 特性，比如声明型事务管理(declarative transaction management)，也需要在应用中包含这个 JAR 文件。
spring-context.jar	这个包里的类为 Spring 核心提供了大量扩展。你可以找到使用 Spring ApplicationContext 特性时所需的全部类，以及支持 EJB、JNDI 和邮件所需的类。这个包还囊括 Spring 远程调用(remoting)类，用来与模板(templateing)引擎如 Velocity 和 FreeMarker 集成的类，以及验证基类(base validation class)。值得注意的是，实际上许多类被包含在这个包里并不太恰当，比如远程调用和 EJB 支持之类的特性，打包成单独的 JAR 文件更为恰当，值得庆幸的是在 Spring2.0 版本中这点得到了改善。
spring-core.jar	所用应用都要用到这个 JAR 文件，它包含访问配置文件、创建和管理 bean 以及进行 DI 操作相关的所有类。如果你的应用只需要基本的 DI 支持，该 JAR 文件足以满足要求。此外该 JAR 文件还包含一组极为有用的工具类，Spring 代码库大量使用了这些类，也可以用自己的应用中。
spring-dao.jar	这个 JAR 囊括 Spring DAO 支持相关的所有基类，还包含 JDBC 和 Spring 的事务抽象层(transaction abstraction layer)进行数据访问的所有类。为了使用声明型事务支持，还需要在自己的应用里包含 spring-aop.jar。
spring-mock.jar	Spring 提供了一整套 mock 类来辅助应用的测试。Spring 测试套件使用了其中大量 mock 类，可令你的应用测试更为简单。至于模拟(mock)HttpServletRequest 和 HttpServletResponse 类在 web 应用单元测试中的巨大用处是勿庸置疑的。
spring-orm.jar	这个 JAR 文件对 Spring 的标准 DAO 特性进行了扩展，使其支持 Hibernate, iBATIS 和 JDO。这个 JAR 文件里大量的类都依赖 spring-dao.jar 里的类，毫无疑问你需要同时包含后者。将来每个 ORM 工具或许会单独打包，这样一来，使用 iBATIS 时你的应用就无需再包含 Hibernate 和 JDO 相关的类。
spring-web.jar	这个 JAR 文件包含 web 应用使用 Spring 时所需的核心类，包括自动载入 ApplicationContext 特性的类、Struts 集成类、文件上传的支持类和大量辅助类，用来执行重复性任务如解析查询(query)字符串里的 int 值。
spring-webmvc.jar	这个 JAR 文件囊括 Spring MVC 框架相关的所有类。如果你的应用使用了独立的 MVC 框架，则无需这个 JAR 文件里的任何类。

Spring 框架包含许多特性，并被很好地组织在下图所示的七个模块中。

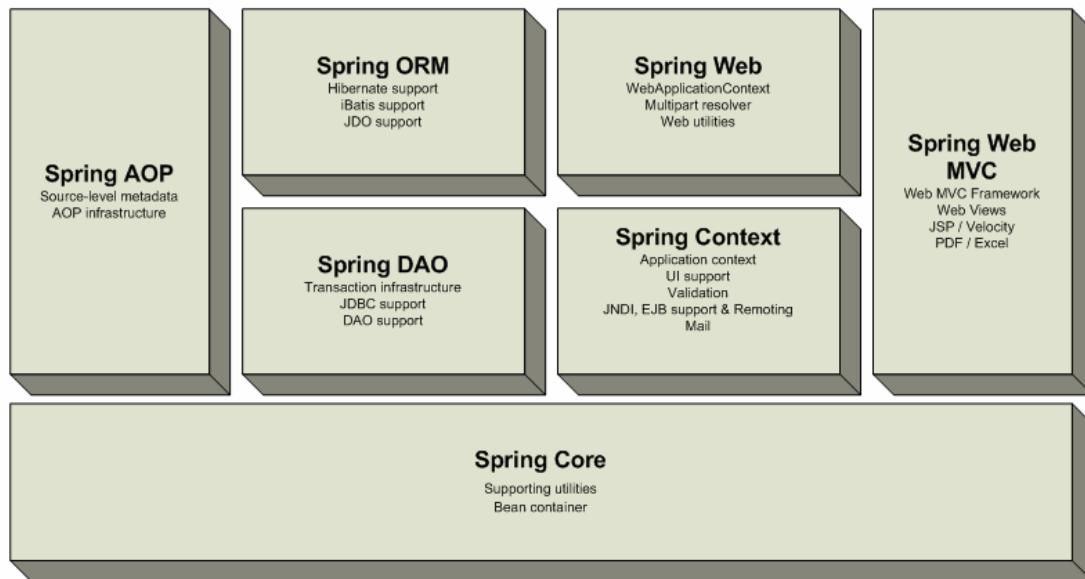


图 9-1: Spring 框架特性

Core 封装包是框架的最基础部分，提供 IoC 和依赖注入特性。这里的基础概念是 BeanFactory，它提供对 Factory 模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

构建于 Core 封装包基础上的 Context 封装包，提供了一种框架式的对象访问方法，有些象 JNDI 注册器。Context 封装包的特性得自于 Beans 封装包，并添加了对国际化 (I18N) 的支持（例如资源绑定），事件传播，资源装载的方式和 Context 的透明创建，比如说通过 Servlet 容器。

DAO 提供了 JDBC 的抽象层，它可消除冗长的 JDBC 编码和解析数据库厂商特有的错误代码。并且，JDBC 封装包还提供了一种比编程性更好的声明性事务管理方法，不仅仅是实现了特定接口，而且对所有的 POJOs (plain old Java objects) 都适用。

ORM 封装包提供了常用的“对象/关系”映射 APIs 的集成层。其中包括 JPA、JDO、Hibernate 和 iBatis。利用 ORM 封装包，可以混合使用所有 Spring 提供的特性进行“对象/关系”映射，如前边提到的简单声明性事务管理。

Spring 的 AOP 封装包提供了符合 AOP Alliance 规范的面向方面的编程 (aspect-oriented programming) 实现，让你可以定义，例如方法拦截器 (method-interceptors) 和切点 (pointcuts)，从逻辑上讲，从而减弱代码的功能耦合，清晰的被分离开。而且，利用 source-level 的元数据功能，还可以将各种行为信息合并到你的代码中，这有点象 .Net 的 attribute 的概念。

Spring 中的 Web 包提供了基础的针对 Web 开发的集成特性，例如多方文件上传，利用 Servlet listeners 进行 IoC 容器初始化和针对 Web 的 application context。当与 WebWork 或 Struts 一起使用 Spring 时，这个包使 Spring 可与其他框架结合。

Spring 中的 MVC 封装包提供了 Web 应用的 Model-View-Controller (MVC) 实现。Spring 的 MVC 框架并不是仅仅提供一种传统的实现，它提供了一种清晰的 分离模型，在领域模型代码和 web form 之间。并且，还可以借助 Spring 框架的其他特性。

### 9.1.3.3. Choosing a Distribution Option: 如何选择发布包

决定选用哪些发布包其实很简单。如果你正在构建 web 应用并将全程使用 Spring，那么最好使用 spring.jar 文件，以免陷入忙于维护不同文件的泥沼。同样，如果你的应用仅仅

用到简单的 DI 容器，那么只需 spring-core.jar 即可搞定。如果你对发布的大小要求很高，那么理应精挑细选，只取用包含自己所需特性的 JAR 文件。合者纳之，余者弃之。

### 9.1.3.4. Spring 使用场景

借助搭积木方式来解释一下各种情景下使用 Spring 的情况，从简单的 Applet 一直到完整的使用 Spring 的事务管理功能和 Web 框架的企业应用。

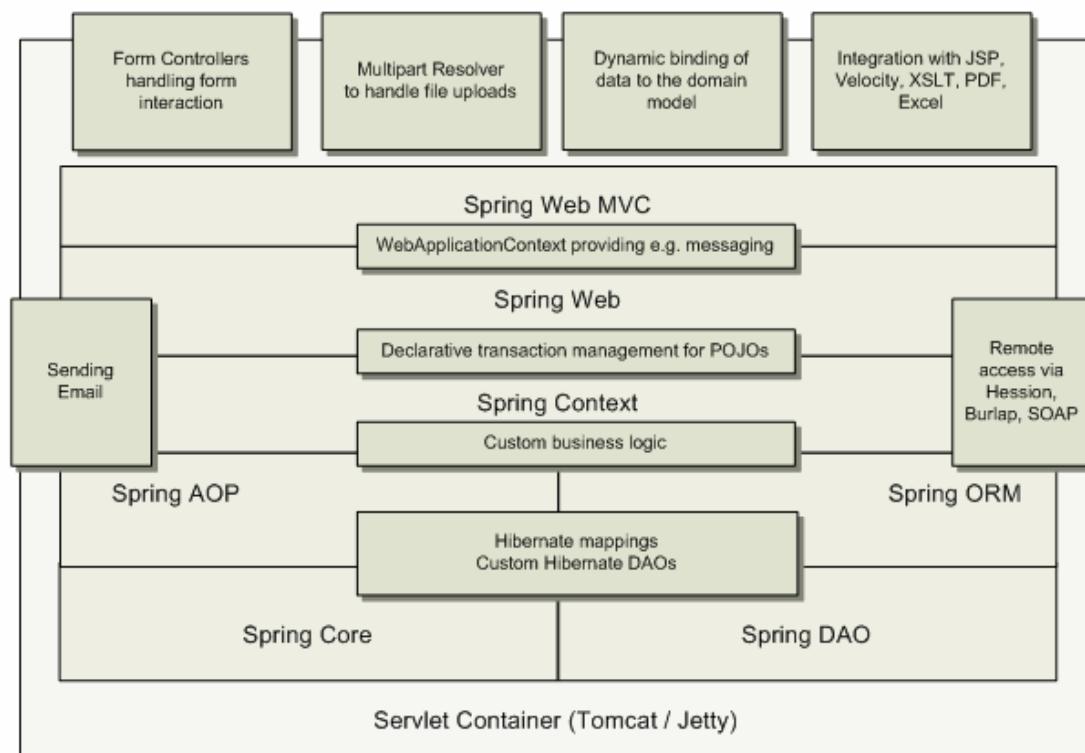


图 9-2: 典型的完整 Spring 的 Web 应用

通过用 Spring 的 声明事务管理特性，Web 应用可以做到完全事务性，就像使用 EJB 提供的那种容器管理的事务一样。所有自定义的业务逻辑可以通过简单的 POJO 来实现，并利用 Spring 的 IoC 容器进行管理。对于其他的服务，比如发送 email 和不依赖 web 层的校验信息，还可以让你自己决定在哪里执行校验规则。Spring 本身的 ORM 支持可以和 JPA、Hibernate、JDO 以及 iBatis 集成起来，例如使用 Hibernate，你可复用已经存在的映射文件与标准的 Hibernate SessionFactory 配置。用控制器去无缝整合 web 层和领域模型，消除对 ActionForms 的依赖，或者避免了其他 class 为领域模型转换 HTTP 参数的需要。

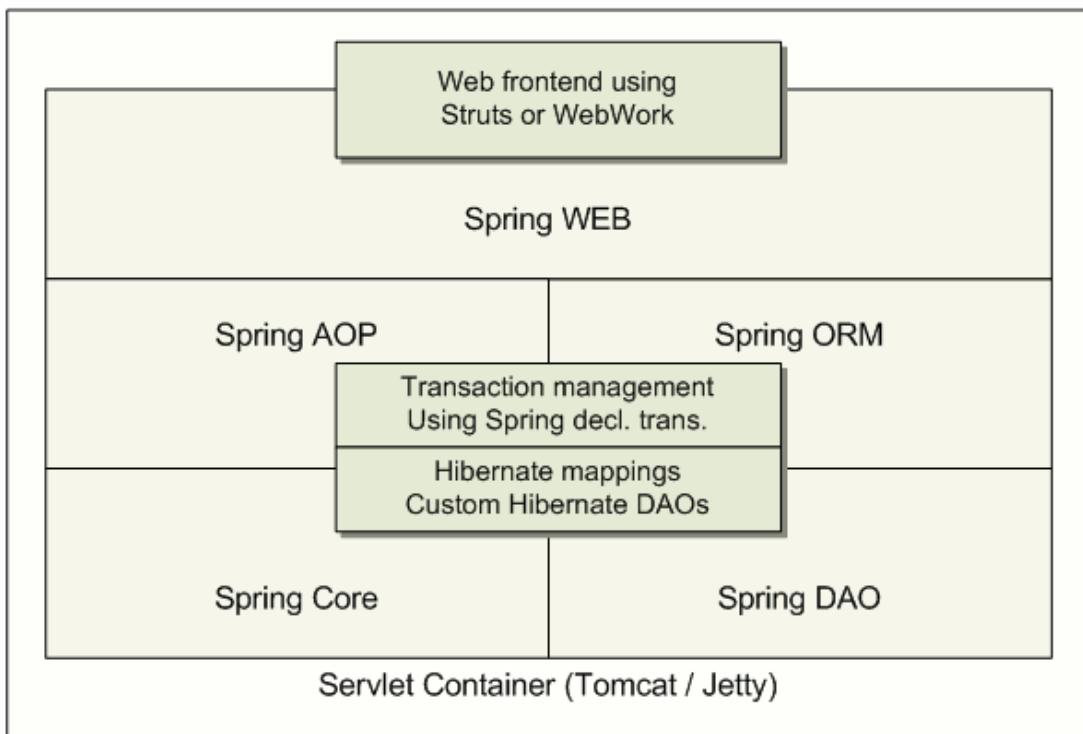


图 9-3: 使用了第三方框架的 Spring 中间层

有的时候，现有情况不允许你彻底地从一种框架切换到另一种框架。然而，Spring 却不需要 强制你使用它的全部，Spring 不是一种 全有全无 的解决方案。如果，现有的应用使用了 WebWork、Struts、Tapestry 或其他的 UI 框架作为前端程序，完全可以只与 Spring 的事务特性进行集成。只需要使用 ApplicationContext 来挂接你的业务逻辑和通过 WebApplicationContext 来集成你的 web 层前端程序。

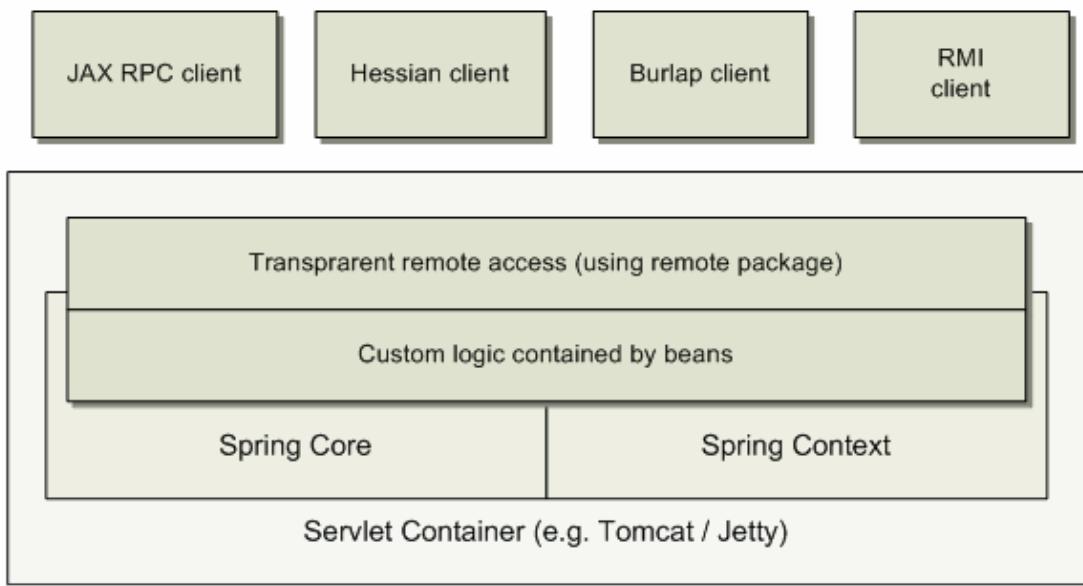


图 9-4: 远程使用场景

当你需要通过 WebService 来访问你的现有代码时，你可使用 Spring 提供的 Hessian-、Burlap-、Rmi- 为前缀的接口或者 JaxRpcProxyFactory 这个代理类。你会发现，远程访问现有应用程序不再那么困难了。

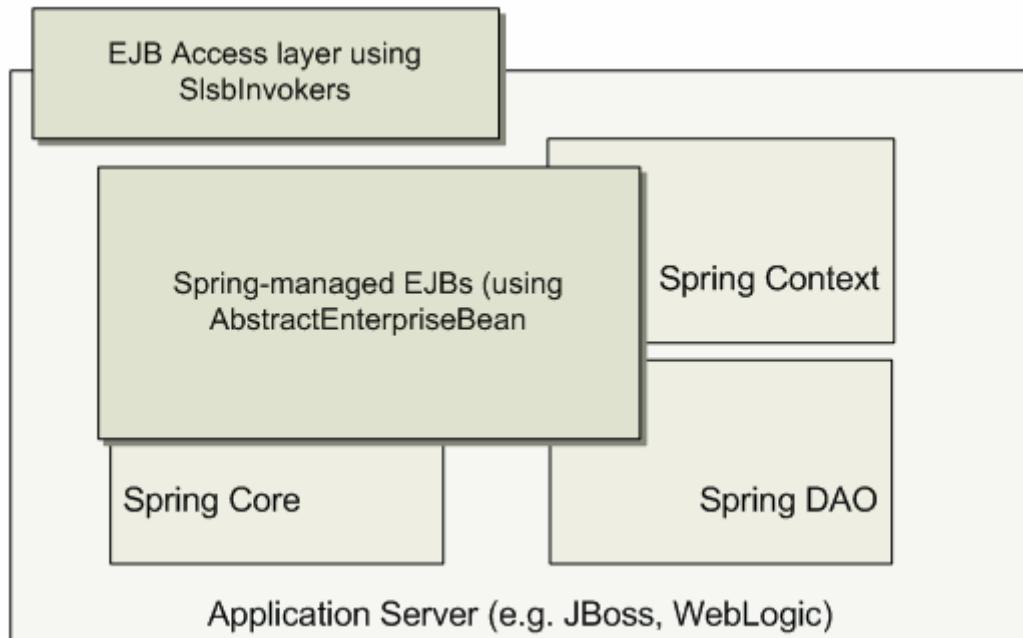


图 9-5: EJBs-包装现有的 POJOs

Spring 还为 EJB 提供了 数据访问和抽象层，让你可以复用已存在的 POJO 并将他们包装在无状态 SessionBean 中，以便在可能需要声明式安全（EJB 中的安全管理，译者注）的非安全的 Web 应用中使用。

### 9.1.4. Putting a Spring into Hello World: Spring 之 Hello World

我们作了很多个 Hello World 实例，相信你已经完全熟悉这个传统的例子。

```
package com.powerise.spring.c1;
```

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

就实例而言，这个算是相当简单的一它确能工作，但是不太具有可扩展性。如果我们想要改变消息的内容呢？如果我们想以不同的方式输出这个消息呢？或许要输出到 stderr 而非 stdout？或许要附上 HTML 标签而非是普通文本？

我们打算重新定义实例应用程序的需求，比方说，它必须支持一个简单灵活的机制以方便改变消息的内容，它还必须能够简便的改变显示（rendering）状态。在简单的 Hello World 例子中，你可以轻易的做到以上两条，只需要对代码作相应的变动即可。然而，在一个更大的应用中，重新编译需要耗费时间，还要求再次对应用进行完整性的测试。显然这不可行，更好的解决方法是将消息的内容移到代码外部，并在运行时读入消息内容，比如从命令行参数，如下所示：

```
package com.powerise.spring.c1;
```

```
public class HelloWorldCommandLine {  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            System.out.println(args[0]);  
        } else {  
            System.out.println("Hello World!");  
        }  
    }  
}
```

这个例子实现了我们的设想之一—现在我们无需改变代码就可以更改消息的内容。不过该应用程序仍有不足：负责消息显示(render)的组件还是得负责消息得获取。要改变消息获取的方式也就等于要改变显示器(renderer)里的代码。姑且我们不论我们还无法轻易改变输出器这一事实；即使能做到这一点，那就意味着要改变这个应用程序的那个类本身。

如果我们进一步修改这个应用程序，使之脱离简单的 Hello World，那么更好的方式是重构显示（rendering）和消息获取的逻辑，使之分别成为独立的组件。另外，如果真想让我们的应用程序变得灵活，我们还得让这些实现自接口（interface），并用这些接口确定组件和启动程序（launcher）之间的相关性。

为了重构消息获取逻辑，我们可以定义一个只有单个方法(method)getMessage()的简单接口 MessageProvider，如下代码：

```
package com.powerise.spring.c1;
```

```
public interface MessageProvider {  
    public String getMessage();  
}
```

而下面代码 MessageRenderer 接口会由可以显示（render）消息的所有组件实现。

```
package com.powerise.spring.c1;
```

```
public interface MessageProviderer {  
    public void render();  
  
    public void setMessageProvider(MessageProvider provider);  
  
    public MessageProvider getMessageProvider();  
}
```

如上所示，MessageRenderer 接口只有一个方法 render()，还有一个 JavaBean 风格的属性（property）MessageProvider。任何 MessageRenderer 的实现都和消息获取部分相分离，而是依赖于 MessageRenderer。创建这个接口的简单实现：

```
package com.powerise.spring.c1;
```

```
public class HelloWorldMessageProvider implements MessageProvider {  
    public String getMessage() {  
        return "Hello World";  
    }  
}
```

```
}
```

在以上代码中，我们创建了一个简单的 MessageProvider，返回的消息一成不变，总是“Hello World!”。StandardOutMessageRenderer 类同样很简单。

```
package com.powerise.spring.c1;

public class StandarOutMessageRenderer implements MessageRenderer {
    private MessageProvider messageProvider = null;

    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }

    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
                "You must set the property messageProvider of class:"
                + StandarOutMessageRenderer.class.getName());
        }
        System.out.println(messageProvider.getMessage());
    }

    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }
}
```

至此，剩下要做的就是重写入口类(entry class)的 main 方法，代码如下所示：

```
package com.powerise.spring.c1;

public class HelloWorldDecoupled {
    public static void main(String[] args) {
        MessageRenderer mr = new StandardOutMessageRenderer();
        MessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

上述代码相当简单，我们先是实例化(instantiate)了 HelloWorldMessageProvider 和 StandardOutMessageRenderer 的实例，不过这两个实例声明时的类别分别是 MessageProvider 和 MessageRenderer。然后把 MessageProvider 传入 MessageRenderer，然后调用 MessageRenderer.render()。

目前这个实例差不多是我们苦苦寻找的，不过还是有个小问题。要想改变 MessageRender 或 MessageProvider 接口的实现就得改变代码。为了解决这个问题，我们可以创建一个简单

的工厂类，负责从 properties 文件里读取实现类的名称，并为应用程序实例化实现类，如下代码所示：

```
package com.powerise.spring.c1;

import java.io.InputStream;
import java.util.Properties;

public class MessageSupportFactory {
    private static MessageSupportFactory instance = null;

    private Properties props = null;

    private MessageRenderer renderer = null;

    private MessageProvider provider = null;

    private MessageSupportFactory() {
        props = new Properties();
        try {
            InputStream is =
                getClass().getClassLoader().getResourceAsStream("msf.properties");
            props.load(is);
            is.close();
            // 获取实现类
            String rendererClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");
            renderer = (MessageRenderer) Class.forName(rendererClass)
                .newInstance();
            provider = (MessageProvider) Class.forName(providerClass)
                .newInstance();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    static {
        instance = new MessageSupportFactory();
    }

    public static MessageSupportFactory getInstance() {
        return instance;
    }
}
```

```
public MessageRenderer getMessageRenderer() {
    return renderer;
}

public MessageProvider getMessageProvider() {
    return provider;
}

}
```

上述实现过于繁琐而幼稚，错误处理也过于简单，此外配置文件的名称也是硬编码于代码中，但是不管怎么样，我们已经写了一定量的代码。更完整适合于产品系统的实现显然需要更多的代码，就这里讲述的内容而言，示例代码已经绰绰有余。这个类所用的配置文件相当简单，如下所示：

```
renderer.class=com.powerise.spring.c1.StandardOutMessageRenderer
provider.class=com.powerise.spring.c1.HelloWorldMessageProvider
```

对 main()方法稍作修改，我们即可达到目标：

```
package com.powerise.spring.c1;
```

```
public class HelloWorldDecoupledWithFactory {
    public static void main(String[] args) {
        MessageRenderer mr = MessageSupportFactory.getInstance().getMessageRenderer();
        MessageProvider mp = MessageSupportFactory.getInstance().getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

#### 9.1.4.1. Refactoring with Spring：用 Spring 进行重构

上一节的例子达到了我们示例应用程序确定的目标，不过仍存在两个问题。其一，为了整个应用程序拼合在一起，我们必须编写大量粘合代码，尽管这样能保证各组件之间的松散耦合。其二，我们还是必须人工为 MessageRenderer 实现提供一个 MessageProvider 实例 (instance)。借助 Spring，我们可以一并解决这两个问题。

为了解决粘合代码过多这个问题，我们完全可以移除应用程序里的 MessageSupportFactory 类，并用 Spring 类 DefaultListableBeanFactory 取而代之。先不必深究这个类，眼下只要知道—这个类相当于 MessageSupportFactory 更通用的版本，只不过玩了点小花样罢了，如以下代码：

```
package com.powerise.spring.c1;

import java.io.FileInputStream;
import java.util.Properties;

import org.springframework.beans.factory.BeanFactory;
```

```
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;

public class HelloWorldSpring {
    public static void main(String[] args) throws Exception {
        // 获取 bean factory
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        MessageProvider mp = (MessageProvider) factory.getBean("provider");
        mr.setMessageProvider(mp);
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // 获取 the bean factory
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        // 创建 definition reader
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
            factory);
        // 加载配置项
        Properties props = new Properties();
        props.load(new
        FileInputStream("D:\\workspace\\SpringExample\\src\\msf.properties"));
        rdr.registerBeanDefinitions(props);
        return factory;
    }
}
```

上述代码中，`main()`方法先是取得一个 `DefaultListableBeanFactory` 实例，其声明类型为 `BeanFactory`，并由这个实例通过 `BeanFactory.getBean()` 方法取得 `MessageRenderer` 和 `MessageProvider` 实例。我们不必深究 `getBeanFactory()` 方法，只需要知道这个方法负责从 `properties` 文件读取 `BeanFactory` 的配置信息，然后返回已配置的实例。这个 `properties` 文件和前面的文件完全一致。

`BeanFactory` 接口和其实现类构成了 Spring DI 的核心。Spring 在其 DI 容器中广泛使用了 JavaBeans 规范；鉴于此，Spring 总是将受其管理的组件作为 Bean 来引用，因此可以使用 `getBean()` 方法。

现在尽管我们在启动类里编写了更多代码，不过已不需要创建 `factory` 代码，同时我们也获得了更为健壮的 `factory` 实现—错误处理更合理，还采用了完全分离的配置机制。不过，这段代码还有瑕疵。启动类必须知道 `MessageRenderer` 的依赖关系，并取得这些依赖关系，将其传给 `MessageRenderer`。在上述代码中 Spring 不过是扮演了一个复杂工厂类的角色，负责创建和提供所需类的实例。最终，借助 Spring 的 DI 功能，我们就可以使用外部 `BeanFactory` 配置把整个应用程序粘合（glue）在一起。配置文件需要稍作修改。

```
#The MessageRenderer
renderer.class=com.powerise.spring.c1.StandardOutMessageRenderer
```

```
render.messageProvider(ref)=provider  
#The MessageProvider
```

```
provider.class=com.powerise.spring.c1.HelloWorldMessageProvider
```

注意，实际上我们只添加了一行，这行负责把 provider bean 赋给 renderer bean 的 MessageProvider 属性。关键字(ref)表示这个属性的值作为对另一个 bean 的引用(reference)而非文字值(literal value)。使用这个配置，我们可以移除启动类对 MessageProvider 接口的所有引用，如下代码所示：

```
package com.powerise.spring.c1;  
  
import java.io.FileInputStream;  
import java.util.Properties;  
  
import org.springframework.beans.factory.BeanFactory;  
import org.springframework.beans.factory.support.DefaultListableBeanFactory;  
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;  
  
public class HelloWorldSpringDI {  
    public static void main(String[] args) throws Exception {  
        // 获取 bean factory  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");  
        mr.render();  
    }  
  
    private static BeanFactory getBeanFactory() throws Exception {  
        // 获取 the bean factory  
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
        // 创建 definition reader  
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(  
            factory);  
        // 加载配置项  
        Properties props = new Properties();  
        props.load(new FileInputStream(  
            "D:\\workspace\\SpringExample\\src\\msf.properties"));  
        rdr.registerBeanDefinitions(props);  
        return factory;  
    }  
}
```

如上述代码所示，main()方法现在只需要获取 MessageRenderer bean，然后调用 render()；Spring 已经创建了 MessageProvider 的实现并将其注入(inject)到 MessageRenderer 实例中。注意对借助 Spring 组装在一起的类，我们无需作任何改动。事实上，这些类和 Spring 任何部分都无关联，对 Spring 的存在也毫无觉察。不过你也可以实现 Spring 专有接口，以便以更多的方式与 DI 容器进行交互。例如，我们可以移除 MessageRenderer.render()里的检查代码

一查看是否已设置 MessageProvider 实现，并只进行一次检查。不过这些需要 Spring 专有接口，这些内容我们将在后面的章节介绍。

## 9.2. Spring 基础

### 9.2.1. Introducing Inversion of Control: 控制反转介绍

#### 9.2.1.1. Inversion of Control and Dependency Injection: 控制反转和依赖注入

IoC 或者 DI 的核心思想在于提供一个更加简单机制来规定组件之间的依赖关系（一般涉及到对象间的合作），并且在它们生命周期中对依赖关系进行管理。一个需要特定的依赖的组件一般会涉及一个依赖对象，在 IoC 的概念中叫做目标（target）。换句话说，IoC 提供了这样的服务，使一个组件能够在它的整个生命周期中访问它的依赖和服务，用这种方法与它的依赖进行交互。总的来说，IoC 能够被分解为两种子类型：依赖注入和依赖查找。这两种子类型在具体实现 IoC 服务的时候被进一步分解。从这个定义上，我们能清晰的看到当谈及 DI 的时候总要说 IoC，但是当说到 IoC 的时候我们却不一定说 DI。

### 9.2.2. Types of Inversion of Control: 控制反转类型

你可能想知道为什么会有两种不同的 IoC 类型，为什么这些类型又会进一步分解成不同的实现。这好像没有一个清晰的答案。的确，这些不同的类型提供灵活的弹性，但是对于我们，IoC 看来更像旧概念的一个混合体；这两种不同类型的 IoC 说明了这个问题。

依赖查找是一种更加传统的方法，第一眼看上去，Java 程序员对它很熟悉。依赖注入相对新一些，方法还没有完全定型，虽然第一眼看上去有些反常，事实上它比依赖查找更加富有弹性且有用。

在依赖查找风格的 IoC 中，一个组件必须获得一个依赖的参考，反观在依赖注入风格中，依赖关系由 IoC 容器从字面意义上注入到组件中。依赖查找有两种类型：依赖拖拽和上下文配置依赖查找。依赖注入同样也有两种常见风格：构造器依赖注入和 Setter 依赖注入。

#### 9.2.2.1. Dependency Pull: 依赖拖拽

在依赖拖拽中，依赖关系是根据需要从一个登记处获取（拖拽）下来。任何一个写过 EJB 代码的程序员都使用过依赖拖拽。Spring 提供了依赖拖拽作为从框架管理的组件中重新获取组件的一种机制。以下代码展示了一个基于 Spring 的应用程序中典型的依赖拖拽。

```
public static void main(String[] args) throws Exception {
    // 获取 bean factory
    BeanFactory factory = getBeanFactory();
    MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
    mr.render();
}
```

这种类型的 IoC 不仅流行于 J2EE 应用中，在从注册处获得依赖关系的 JNDI 查找（JNDI lookups）中也得到了广泛的应用，它在使用 Spring 的许多环境中也起着关键性的作用。

### 9.2.2.2. Contextualised Dependency Lookup: 上下文配置依赖

#### 查找

上下文配置依赖查找 (CDL) 与之类似，在一些地方与依赖拖拽相仿。但是在 CDL 中，查找是在容器管理的资源中进行的，而不是从一个集中的注册处，同时它一般在规定点执行。CDL 通过以下代码的形式来工作，组件实现一个特定的接口。

```
package com.powerise.spring.c2;

public interface ManagedComponent {
    public void performLookup(Container container);
}
```

通过实现这个接口，组件通知它需要获取依赖关系。当容器准备好传递依赖到组件中去，它会依次调用各个组件的 `performLookup()` 方法。组件这个时候就能够通过容器的接口来查找依赖关系，如下代码。

```
package com.powerise.spring.c2;

public class ContextualizedDependencyLookup implements ManagedComponent {

    private Dependency dep;

    public void performLookup(Container container) {
        this.dep = (Dependency) container.getDependency("myDependency");
    }
}
```

### 9.2.2.3. Constructor Dependency Injection: 构造器依赖注入

构造器依赖注入就是在组件的构造器处提供依赖关系的注入。这种组件声明一个构造器或者一组构造器从构造参数中获取依赖关系，IoC 容器会在实例化它时将依赖关系传送给它，例如以下代码。

```
package com.powerise.spring.c2;

public class ConstructorInjection {
    private Dependency dep;

    public ConstructorInjection(Dependency dep) {
        this.dep = dep;
    }
}
```

### 9.2.2.4. Setter Dependency Injection: Setter 依赖注入

在 Setter 依赖注入中，IoC 容器通过 JavaBean 形式的方法将组件的依赖关系注入到组件中。组件的 Setter 方法将一组依赖关系暴露给 IoC 容器，并受之控制。如以下代码。

```
package com.powerise.spring.c2;
```

```
public class SetterInjection {  
    private Dependency dep;  
  
    public void setMyDependency(Dependency dep) {  
        this.dep = dep;  
    }  
}
```

在容器中，依赖需要通过 `setMyDependency()` 方法暴露出来，它是 `myDependency` 遵循 JavaBean 风格命名的方法。实际上，setter 注入是最广泛使用的注入机制，它也是最容易实现的 IoC 机制之一。

### 9.2.3. Inversion of Control in Spring: Spring 中的控制反转

我们先前提到过，控制反转是 Spring 提供的非常重要的一个功能，并且 Spring 的实现中的核心部分就是基于依赖注入的，同时还提供了依赖查找的功能。Spring 提供自动使独立的对象合作的功能，当然这使用了依赖注入实现。在基于 Spring 的应用程序中，总是偏向于通过依赖注入将合作关系传递给独立的对象，而不是让独立的对象通过查找来获取合作关系。尽管依赖注入是连接独立对象使之合作的首选方案，你还是需要依赖查找来访问独立的对象。在很多环境中，Spring 不能自动的将你的应用程序组件通过依赖注入连接起来，这是你必须通过依赖查找来访问刚刚初始化的一组对象。当你使用 Spring MVC 支持来构建一个 Web 应用程序时，Spring 可以避免这些问题，将你的整个程序自动的粘合起来。在 Spring 中只要可以使用依赖注入，那你就应该尽量使用它；否则你只能求助于依赖查找的能力了。

Spring 的 IoC 容器的一个有趣的功能是，它能够作为其自己的依赖注入容器和外部的依赖查找容器之间的一个适配器。

Spring 支持构造器依赖注入和 setter 依赖注入，支持标准的 IoC 功能和很多的有用的附加功能，这些能够使你的生活更加便利。

### 9.2.4. Dependency Injection with Spring: 使用 Spring 依赖注入

Spring 的依赖注入支持非常全面，已经远远超越了一般意义上的标准 IoC 功能，在后面将会看到更加详细的讨论。这里介绍 Spring 依赖注入容器的基本知识，分析 setter 和构造器依赖注入，同时会深入的分析 Spring 中的依赖注入是如何配置的。

#### 9.2.4.1. Beans and BeanFactories: Beans 和 Bean 工厂 (BeanFactories)

Spring 的依赖注入容器的核心是 Bean 工厂。Bean 工厂负责管理组件和它们之间的依赖关系。Spring 中，这种 bean 用来查阅所有受容器管理的组件。典型的情况你的 beans 会在某种程度上依附于 JavaBeans 规范，但是这不是必须的，尤其如果你计划使用构造器依赖注入来连接你的 beans 的时候。

你的应用程序需要通过 BeanFactory 接口来使用 Spring 的 DI 容器。也就是说，你的程序必须创建实现了 BeanFactory 接口的类来配置它的 Bean 和依赖的消息。完成以后，你的程序能够通过 BeanFactory 来访问这些 beans，继续处理其它工作。在某些情况下，所有的

这些内容的设置都会被自动处理，但是在另一些情况下，你需要自己来完成编码。本章中的所有例子都需要手动设置 BeanFactory 的实现。

虽然 BeanFactory 都可以通过编程来进行配置，但是更常见的是通过外部的一些配置文件来完成配置。Bean 配置在内部是通过实现了 BeanDefinition 接口的类的实例来表现的。Bean 的配置不仅存储着关于 bean 自己的信息，同时还有其依赖的 beans 的信息。对于任何同时实现了 beanDefinitionRegistry 接口的 BeanFactory 类，你可以从配置文件中读取 Bean 定义（BeanDefinition）数据，既可以通过 PropertiesBeanDefinitionReader（基于.properties 文件）也可以通过 XMLBeanDefinitionReader（基于 XML 文件）。这两种主要的 BeanFactory 实现都与 Spring 实现的 BeanDefinitionRegistry 共存。

如此，你能够在 BeanFactory 中标记你的 beans，每个 bean 都被命名。每一个 bean 至少要有一个命名，但是可以拥有多个。第一个命名后的任何命名都被认为是同一个 bean 的别名。你可以使用 bean 的命名来从 BeanFactory 获取它，同时也可以建立依赖关系——例如 bean X 依赖于 bean Y。

### 9.2.4.2. BeanFactory Implementations: BeanFactory 的实现

对于 BeanFactory 的描述也许是看起过渡复杂了，但是应用中，它并不复杂。实际上，我们已经在前面的部分中讨论了所有的概念。

```
package com.powerise.spring.c1;

import java.io.FileInputStream;
import java.util.Properties;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;

public class HelloWorldSpringDI {
    public static void main(String[] args) throws Exception {
        // 获取 bean factory
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // 获取 the bean factory
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        // 创建 definition reader
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
                factory);
        // 加载配置项
        Properties props = new Properties();
        props.load(new FileInputStream(
                "D:\\workspace\\SpringExample\\src\\msf.properties"));
    }
}
```

```
rdr.registerBeanDefinitions(props);
return factory;
}

}
```

在这个例子中，你会发现我们在使用 DefaultListableBeanFactory——Spring 提供的两个主要的 BeanFactory 实现中的一个，我们会使用 PropertiesBeanDefinitionReader 从一个属性 (properties)文件中读取 Bean 定义信息。只要 BeanFactory 的实现被创建和配置好，我们就可以通过 MessageRenderer 的命名来获取它，Renderer 是通过属性文件来进行配置的。

除了 PropertiesBeanDefinitionReader 以外，Spring 还提供了 XmlBeanDefinitionReader，它允许你使用 XML 文件来代替属性文件配置和管理你的 bean。虽然属性文件对于小而简单的程序很理想，但是当你处理大量的 beans 的时候你会发现这是一个麻烦。因为这个原因，除了对于特别细碎的程序我们应该全部选择 XML 配置格式。所以这里要详细的讨论一下两个主要 BeanFactory 中的第二个：XmlBeanFactory。

XmlBeanFactory 派生于 DefaultListableBeanFactory 并且简单的扩展了它，使它能够通过 XmlBeanDefinitionReader 自动获取配置信息。这种方式优于创建如下代码：

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.core.io.FileSystemResource;

public class XmlConfig {
    public static void main(String[] args) {
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        XmlBeanDefinitionReader rdr = new XmlBeanDefinitionReader(factory);
        rdr.loadBeanDefinitions(new FileSystemResource("ch4/src/conf/beans.xml"));
        Oracle oracle = (Oracle)factory.getBean("oracle");
    }
}
```

你可以替换成以下代码：

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class XmlConfigWithBeanFactory {
    public static void main(String[] args) {
        XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));
    }
}
```

余下部分，包括示例程序，我们将只使用格式 XML 配置。你可以自己去研究一下

properties 格式——你会发现在 Spring 的基础代码中充斥着这样的例子。

当然，你也可以定义自己的 BeanFactory 实现，然而这样做是很麻烦的；你需要实现很多的接口来达到与 BeanFactory 相同级别的功能，而不象所提供的 BeanFactory 那样简单。如果你想定义一个新的配置机制，那么你需要创建一个定义读取器（definition reader），然后把它封装到一个派生自 DefaultListableBeanFactory 的简单的 BeanFactory 实现里。这就是实现 XmlBeanFactory 所使用的方法；可以查看 Spring 的代码来了解具体的细节。

## 9.2.5. Configuring the BeanFactory : 配置 Bean 工厂 (BeanFactory)

开始建立一个基于 Spring 的应用程序的关键在于为你的程序创建 BeanFactory 配置。一个没有任何 bean 定义的基本配置文件看起来是这个样子的：

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
</beans>
```

任何 bean 的定义都是通过在根节点的<beans>标记下通过使用<bean>来声明的。<bean>标记下面有两个属性是必需的：id 和 class。Id 属性用来给这个 bean 一个默认的命名，class 属性用来指定这个 bean 的类型。前面章节中 Hello World 示例中的两个 bean（renderer 和 provider）是如何通过配置文件定义的。使用 XML 配置的 Hello World 示例：

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="renderer" class="com.powerise.spring.c1.StandardOutMessageRenderer">
    </bean>
    <bean id="provider" class="com.powerise.spring.c1.HelloWorldMessageProvider">
    </bean>
</beans>
```

我们可以通过 XmlBeanFactory 读取这些配置。

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

import com.powerise.spring.c1.MessageProvider;
import com.powerise.spring.c1.MessageRenderer;

public class HelloWorldXml {
    public static void main(String[] args) throws Exception {
        // get the bean factory
        BeanFactory factory = getBeanFactory();
```

```
MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
MessageProvider mp = (MessageProvider) factory.getBean("provider");

mr.setMessageProvider(mp);
mr.render();
}

private static BeanFactory getBeanFactory() throws Exception {
    // get the bean factory
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
        "D:\\workspace\\SpringExample\\src\\beans.xml"));

    return factory;
}
}
```

此处非常有意思的地方就在于 main()方法的代码与前面的例子没有任何改变。这是因为它通过 BeanFactory 接口来协同，而不是其它子接口或者类。虽然你也许需要在配置的时候指定它与特定的 BeanFactory 合作，但是你不需要在你的应用程序中的其它地方通过 getBean()方法来查找 beans。这是一种应该遵循的很好的模式，你应该避免使你的应用程序与一个特定的 BeanFactory 实现间过渡耦合。

以上代码中有一个问题，这个问题我们在前面遇到并已经解决了——应用程序依然必须将 provider bean 传递给参考 bean 来满足它们的依赖。Spring 通过 setter 依赖注入来解决这个问题。我们当然也可以通过 XML 配置支持来完成。

### 9.2.5.1. Using Setter Injection: 使用 Setter 依赖注入

通过 XML 支持来配置 setter 依赖注入，你需要在<bean>标记下指定<property>标记，将<property>标记放置到你需要注入依赖关系的地方。例如，给 provider bean 的 messageProvider 属性指派 renderer bean，我们可以简单的修改 renderer 的<bean>标记下的内容，就像下面这样：

```
<bean id="renderer"
      class="com.powerise.spring.c1.StandardOutMessageRenderer">
    <property name="messageProvider">
        <ref local="provider" />
    </property>
</bean>
```

从这段代码里，我们将 provider bean 指派给 messageProvider 属性。我们使用<ref>标记将一个 bean 的引用分配给一个属性（稍后详细讨论）。现在我们可以移除 Hellow World 示例中的无用的属性分配，就像以下代码：

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
```

```
import com.powerise.spring.c1.MessageRenderer;

public class HelloWorldXmlWithDI {
    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));

        return factory;
    }
}
```

这个例子充分利用了 Spring 的依赖注入能力，完全使用 XML 格式进行配置。

### 9.2.5.2. Using Constructor Injection: 使用构造器依赖注入

前面的例子中，MessageProvider 的实现即 HelloWorldMessageProvider 为每个 getMessage() 方法返回相同的手工编码的信息。在 Spring 配置文件中，你可以轻松的创建一个允许在外部定义消息的可配置的 messageProvider，就像以下代码：

```
package com.powerise.spring.c2;

import com.powerise.spring.c1.MessageProvider;

public class ConfigurableMessageProvider implements MessageProvider {

    private String message;

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

如你所见，不可能在不提供某个消息的值（除非支持 null）的情况下创建一个 ConfigurableMessageProvider 的实例。这正是我们所需要的，这个类非常适合使用构造器依赖注入。

```
<bean id="provider" class="com.powerise.spring.c2.ConfigurableMessageProvider">
    <constructor-arg>
        <value>This is a configurable message</value>
    </constructor-arg>
</bean>
```

这段代码中，代替`<property>`标记，我们使用了`<constructor-arg>`标签。因为我们这次没有传递另一个`bean`进去，只是一个`String`字面值，我们使用了`<value>`标记代替`<ref>`指定了构造器依赖注入的值。

当你拥有超过一个的构造器参数或者你的类拥有多个构造器，你需要给每个`<constructor-arg>`标记一个索引属性在构造器署名指定参数的索引，它起始于0。在你处理一个具有多个参数的构造器的时候你最好使用索引属性来避免在参数之间发生混淆并且确认Spring选用了正确的构造器。

### 避免构造器混淆

早某些情况，Spring会难以确定你需要使用哪个构造器进行依赖注入。这一般发生在你拥有两个具有相同数量的参数且参数的类型也完全相同的构造器的时候。

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class ConstructorConfusion {
    private String someValue;

    public ConstructorConfusion(String someValue) {
        System.out.println("ConstructorConfusion(String) called");
        this.someValue = someValue;
    }

    public ConstructorConfusion(int someValue) {
        System.out.println("ConstructorConfusion(int) called");
        this.someValue = "Number: " + Integer.toString(someValue);
    }

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));

        ConstructorConfusion cc = (ConstructorConfusion) factory
            .getBean("constructorConfusion");
        System.out.println(cc);
    }

    public String toString() {
```

```
    return someValue;  
}  
}
```

这里，你可以清楚地看到这些代码的功能——它简单的从 BeanFactory 获取一个 ConstructorConfusion 类型的 bean，然后将值写入到标准输出。

```
<bean id="constructorConfusion"  
      class="com.powerise.spring.c2.ConstructorConfusion">  
    <constructor-arg>  
      <value>90</value>  
    </constructor-arg>  
</bean>
```

这里会调哪—个构造器？运行一下次出的代码得到如下的输出：

```
ConstructorConfusion(String) called
```

```
90
```

这表明具有 String 参数的构造器被调用了。这不是所想要的效果，因为我们希望任何具有整数前缀的值都被传递到通过数字注入的构造器中，即例子中的 int 构造器。改变这个问题，我们需要对配置文件进行一个修改：

```
<bean id="constructorConfusion"  
      class="com.powerise.spring.c2.ConstructorConfusion">  
    <constructor-arg type="int">  
      <value>90</value>  
    </constructor-arg>  
</bean>
```

注意，现在<constructor-arg>标记拥有一个附加属性 type，它指定了 Spring 所查找的参数的类型。重新修改配置文件，运行例子中的代码可以得到正确的输出：

```
ConstructorConfusion(int) called
```

```
Number: 90
```

## 9.2.6. Injection Parameters: 注入参数

在前面的两个例子里，你看到了如何使用 setter 依赖注入和构造器依赖注入将其他组件和值注射到 bean 里面。Spring 支持非常多的依赖注入参数选项，不仅允许你注入组件和简单的值，还支持包括 Java 集合类、外部定义的属性文件，甚至其它工厂中的 beans。你可以在 setter 依赖注入和构造器依赖注入中使用所有这些参数类型，通过在<property>和<constructor-args>标记下分别设置的相应标记。

### 9.2.6.1. 简值注入

将简单的值注入到 beans 里面是很简单的。如果需要，只需要简单的在配置标记中指定值，将其封装在一个<value>标记中。默认情况，<value>标记只能读取 String 值，但是也可以将这些值转换到任何原生数据类型或者原生封装类。下面代码展示了一个暴露了多种属性的简单的 bean。

```
package com.powerise.spring.c2;  
  
import org.springframework.beans.factory.BeanFactory;  
import org.springframework.beans.factory.xml.XmlBeanFactory;
```

```
import org.springframework.core.io.FileSystemResource;

public class InjectSimple {
    private String name;

    private int age;

    private float height;

    private boolean isProgrammer;

    private Long ageInSeconds;

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));
        InjectSimple simple = (InjectSimple) factory.getBean("injectSimple");
        System.out.println(simple);
    }

    public void setAgeInSeconds(Long ageInSeconds) {
        this.ageInSeconds = ageInSeconds;
    }

    public void setIsProgrammer(boolean isProgrammer) {
        this.isProgrammer = isProgrammer;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setHeight(float height) {
        this.height = height;
    }

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

    public String toString() {
        return "Name :" + name + "\n" + "Age:" + age + "\n"
            + "Age in Seconds: " + ageInSeconds + "\n" + "Height: "
            + height + "\n" + "Is Programmer?: " + isProgrammer;
    }
}
```

```
}
```

除了这些属性外，InjectSimple 类还可以定义 main()方法来创建一个 XmlBeanFactory 然后从 Spring 获取一个 InjectSimple 的 bean。这个 bean 的属性值被写入到标准输出。

```
<bean id="injectSimple" class="com.powerise.spring.c2.InjectSimple">
    <property name="name">
        <value>John Smith</value>
    </property>
    <property name="age">
        <value>35</value>
    </property>
    <property name="height">
        <value>1.78</value>
    </property>
    <property name="isProgrammer">
        <value>true</value>
    </property>
    <property name="ageInSeconds">
        <value>1103760000</value>
    </property>
</bean>
```

你可以看到，可以在 bean 里面定义接受 String 值的属性，可以是原生数据类型值或者原生封装类值，使用<value>标记把这些值注入到这些属性。下面是运行这个例子所生成的输出，如我们所料：

```
Name: John Smith
Age: 35
Age in Seconds: 1103760000
Height: 1.78
Is Programmer?: true
```

### 9.2.6.2. 在同一个工厂注入 Beans

就像你所看到的，允许使用<ref>标记将一个 bean 注入到另一个 bean 中。

```
package com.powerise.spring.c2;
```

```
public class InjectRef {
    private Oracle oracle;

    public void setOracle(Oracle oracle) {
        this.oracle = oracle;
        System.out.println(oracle.defineMeaningOfLife());
    }
}
```

要配置 Spring 将一个 bean 注入到另一个中，你首先需要配置两个 beans：一个是要注入的，另一个是注入的目标。这样做了以后，你可以使用<ref>标记在目标 bean 里面配置注

入。记住，`<ref>`必须在一个`<property>`或者`<constructor-arg>`标记下，这决定于你在使用 setter 依赖注入还是构造器依赖注入。

```
<bean id="oracle" name="wiseworm"
      class="com.powerise.spring.c2.BookwormOracle"/>
<bean id="injectRef" class="com.powerise.spring.c2.InjectRef">
    <property name="oracle">
        <ref local="oracle"/>
    </property>
</bean>
```

要说明的非常重要的一点是，要注入的类型不一定就是目标中定义的类型；这些类型只需要相容就可以了。相容意味着如果目标类中定义的类型是个接口，那么注入的类型必须实现了这个接口。如果定义了的类型诗歌类，那么注入的类型必须是相同的类或者一个子类。在例子中，`InjectRef` 类型定义了一个 `setOracle()`方法接受一个 `Oracle` 的实例，那是一个接口，注入的类型是一个实现了 `Oracle` 的类 `BookwormOracle`。这点可能会造成一些开发者的混淆，但是这真的非常简单。注入受控于任何 Java 代码所需要遵循的类型控制关系，所以当你理解了 Java 的类型是如何工作的，那么理解注入中的类型控制就非常简单了。

在前面的例子里，要注入的 `bean` 的 `id` 使用了本地的`<ref>`标记属性来指定。你后面将会看到，在“理解 Bean 命名”那一节中，你可以给一个 `bean` 多个命名，这样你可以通过很多的别名来引用它。当你使用本地属性，那么意味着`<ref>`标记仅仅通过 `bean` 的 `id` 属性查找，而不通过任何的别名查找。通过任何命名来注入 `bean`，使用 `bean` 的`<ref>`标记的属性代替本地属性。下面展示了前面例子的另一种配置方式，通过别名来注入 `bean`。

```
<bean id="injectRef" class="com.powerise.spring.c2.InjectRef">
    <property name="oracle">
        <ref bean="wiseworm"/>
    </property>
</bean>
<bean id="oracle" name="wiseworm" class="com.powerise.spring.c2.BookwormOracle"/>
```

在这个例子中，`oracle` `bean` 通过 `name` 属性的一个别名给出，然后它被通过别名和 `bean` 属性的`<ref>`标记之间的连接注入到 `injectRef` `bean`。

### 9.2.6.3. 注入和 BeanFactory 嵌套

到目前为止我们注入的 `bean` 与需要被注入的 `bean` 都放在同一个 `bean` 工厂里。可是，Spring 支持 Bean 工厂的分层结构，如此一个工厂可以被当作另一个工厂的父节点。通过允许 Bean 工厂的嵌套，Spring 允许你将你的配置文件分割为多个不同的文件——对于拥有大量 `beans` 的工程是个福音。

当嵌套 Bean 工厂时，Spring 允许其中的 `beans` 将子工厂作为父工厂的参考 `beans`。唯一的缺点是这只能在配置中使用。调用子 Bean 工厂的 `getBean()` 来访问父工厂中的一个 `bean` 是不可能的。

使用 `XmlBeanFactory` 的 Bean 工厂嵌套非常容易被约束。将一个 `XmlBeanFactory` 嵌套到另一个里，只需要简单的将父 `XmlBeanFactory` 作为子 `XmlBeanFactory` 的构造器参数传递过去就可以了。嵌套 `XmlBeanFactory` 的代码：

```
BeanFactory parent = new XmlBeanFactory(new FileSystemResource(
    "D:\\workspace\\SpringExample\\src\\parent.xml"));
BeanFactory child = new XmlBeanFactory(new FileSystemResource(
```

```
"D:\\workspace\\SpringExample\\src\\beans.xml");
```

在子 Bean 工厂的配置文件中，在父 Bean 工厂中引用一个 bean 与子 Bean 工厂中引用一个 bean 的工作方式是相同的，除非你在子 Bean 工厂中有一个与之同名的 bean。如果那样，你只要用父工厂替代<ref>标记的 bean 属性，就可以了。

父 Bean 工厂的配置

```
<beans>
    <bean id="injectBean" class="java.lang.String">
        <constructor-arg>
            <value>Bean In Parent</value>
        </constructor-arg>
    </bean>
    <bean id="injectBeanParent" class="java.lang.String">
        <constructor-arg>
            <value>Bean In Parent</value>
        </constructor-arg>
    </bean>
</beans>
```

如你所见，配置简单定义了两个 beans： injectBean 和 injectBeanParent。两个都是父工厂中数值 Bean 的 String 对象。

```
<bean id="target1" class="com.powerise.spring.c2.SimpleTarget">
    <property name="val">
        <ref bean="injectBeanParent"/>
    </property>
</bean>

<bean id="target2" class="com.powerise.spring.c2.SimpleTarget">
    <property name="val">
        <ref local="injectBean"/>
    </property>
</bean>

<bean id="target3" class="com.powerise.spring.c2.SimpleTarget">
    <property name="val">
        <ref parent="injectBean"/>
    </property>
</bean>

<bean id="injectBean" class="java.lang.String">
    <constructor-arg>
        <value>Bean In Child</value>
    </constructor-arg>
</bean>
```

注意我们在此处定义了四个 beans。这个 listing 中的 injectBean 与父工厂的 injectBean 类似，除了那个 String 的值不同，这表明他是在从子 BeanFactory 获取的。

Target1 这个 bean 使用了<ref>标记的 bean 属性来引用另外一个命名为 injectBeanParent 的 bean。因为这个 bean 只在父 Bean 工厂中存在，target1 获得那个 bean 的引用。这里有两点很有意义。第一点，你既可以在子 Bean 工厂也可以在父 Bean 工厂使用 bean 属性来引用 bean。这使透明的引用 bean 易于实现，当你的程序膨胀时允许你在配置文件中移动 beans。第二点，你不能使用本地属性来引用父工厂中的 beans。XML 解析器会在同一个文件中检查本地的属性是否存在一个可用的元素（element）对应，阻止引用父工厂中的 beans。

Target2 这个 bean 使用了<ref>标记的 bean 属性来引用 injectBean。因为这个 bean 在两个 Bean 工厂中都有定义，target2 bean 所得到的 injectBean 的引用来自己那个 BeanFactory。

Target3 这个 bean 使用了<ref>标记的 parent 属性来直接引用父 Bean 工厂中的 injectBean。因为 target3 使用了<ref>标记的 parent 属性，在子 Bean 工厂中所声明的 injectBean 被完全忽略了。

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class HierarchicalBeanFactoryUsage {
    public static void main(String[] args) {
        BeanFactory parent = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\parent.xml"));
        BeanFactory child = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"), parent);

        SimpleTarget target1 = (SimpleTarget) child.getBean("target1");
        SimpleTarget target2 = (SimpleTarget) child.getBean("target2");
        SimpleTarget target3 = (SimpleTarget) child.getBean("target3");

        System.out.println(target1.getVal());
        System.out.println(target2.getVal());
        System.out.println(target3.getVal());
    }
}
```

下面是这个例子的输出：

Bean In Parent

Bean In Child

Bean In Parent

和我们料想的一样，target1 和 target3 的 bean 都得到了父 Bean 工厂中的 bean 的引用，而 target2 bean 从子 BeanFactory 中得到了 bean 的引用。

#### 9.2.6.4. 使用集合进行注入

一般，你的 bean 需要访问对象的集合，而不是访问一个单一的 bean 或者值。因此，理所当然的，Spring 允许你在你的一个 bean 中注入一个对象的集合。使用集合很简单：你可以选择<list>、<map>、<set>或者<props>来描述 List、Map、Set 或者 Properties 的实例，然

后就像你在其它注入中所用到的方式一传递这些对象。这个<props>标记只允许传递 String 值，因为 Properties 类只允许 String 作为属性的值。当使用<list>、<map>或者<set>时，你可以使用注入到属性时可以使用的任何标记，甚至可以是其它的集合标记。这样，允许你传递存储 Map 的 List，存储 Set 的 Map，甚至是 List 存储 Map、Map 中存储 Set、Set 中存储 List 这样的嵌套！

```
package com.powerise.spring.c2;

import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class CollectionInjection {
    private Map map;

    private Properties props;

    private Set set;

    private List list;

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));

        CollectionInjection instance = (CollectionInjection) factory
            .getBean("injectCollection");
        instance.displayInfo();
    }

    public void setList(List list) {
        this.list = list;
    }

    public void setSet(Set set) {
        this.set = set;
    }

    public void setMap(Map map) {
```

```
        this.map = map;
    }

    public void setProps(Properties props) {
        this.props = props;
    }

    public void displayInfo() {

        // display the Map
        Iterator i = map.keySet().iterator();

        System.out.println("Map contents:\n");
        while (i.hasNext()) {
            Object key = i.next();
            System.out.println("Key: " + key + " - Value: " + map.get(key));
        }

        // display the properties
        i = props.keySet().iterator();
        System.out.println("\nProperties contents:\n");
        while (i.hasNext()) {
            String key = i.next().toString();
            System.out.println("Key: " + key + " - Value: "
                + props.getProperty(key));
        }

        // display the set
        i = set.iterator();
        System.out.println("\nSet contents:\n");
        while (i.hasNext()) {
            System.out.println("Value: " + i.next());
        }

        // display the list
        i = list.iterator();
        System.out.println("\nList contents:\n");
        while (i.hasNext()) {
            System.out.println("Value: " + i.next());
        }
    }
}
```

这是很长的一段代码，但是它所作的工作去很少。Main()方法从 Spring 获得了一个 CollectionInjection 的 bean，然后调用 displayInfo()方法。这个方法只是输出从 Spring 注入的

List、Map、Properties 和 Set 的实例的内容。在下面代码中你可以看到对于 CollectionInjection 类中注入的每一个属性所对应的注入值的配置。

```
<!-- collection injection samples -->
<bean id="injectCollection"
      class="com.powerise.spring.c2.CollectionInjection">
    <property name="map">
      <map>
        <entry key="someValue">
          <value>Hello World!</value>
        </entry>
        <entry key="someBean">
          <ref local="oracle" />
        </entry>
      </map>
    </property>
    <property name="props">
      <props>
        <prop key="firstName">Rob</prop>
        <prop key="secondName">Harrop</prop>
      </props>
    </property>
    <property name="set">
      <set>
        <value>Hello World!</value>
        <ref local="oracle" />
      </set>
    </property>
    <property name="list">
      <list>
        <value>Hello World!</value>
        <ref local="oracle" />
      </list>
    </property>
  </bean>
```

在这段代码中，你可以看到，我们给 ConstructorInjection 类暴露的四个 setter 注入了值。对于 map 属性，我们使用<map>标记注入了一个 Map 的实例。注意，每一个条目都通过<entry>标签来指定，并且每一个都有一个 String 的键值和一个条目值。条目的值可以是你给单独的属性能够注入的任何类型；这个例子展示了<value>和<ref>标记的使用方法，添加了一个 String 值和一个 bean 的引用到 Map 中。对于 props 属性，我们使用了<props>标记来建立一个 java.util.Properties 的实例，通过<prop>标记来扩展它。注意，虽然<prop>标记使用了类似<entry>标记的方式来键入，但是你只能给 Properties 实例的每个属性指定一个 String 的值。

<list>和<set>标记工作的方式完全相同：你给每个元素指定值，可以使用在给一个属性注入单独的值的时候所能使用的任何标记，如<value>和<ref>。你可以看到我们给 List 和 Set

各添加了一个 String 值和一个 bean 引用。输出结果如下：

Map contents:

Key: someValue - Value: Hello World!

Key: someBean - Value: com.powerise.spring.c2.BookwormOracle@45a877

Properties contents:

Key: secondName - Value: Harrop

Key: firstName - Value: Rob

Set contents:

Value: Hello World!

Value: com.powerise.spring.c2.BookwormOracle@45a877

List contents:

Value: Hello World!

Value: com.powerise.spring.c2.BookwormOracle@45a877

要记住，对于<list>、<map>和<set>元素，你可以使用任何用来设置非集合属性的标记来指定集合的条目的值。这个理念非常的强大，因此你在注入集合的时候后就不会被限制于使用原生值了，你也可以将一个集合的 bean 注入到另一个集合中。

使用这项功能，很容易将你的程序模块化，提供不同的、用户可选择的程序关键逻辑块的实现。考虑一个这样的系统，在线的提供定制的商业逻辑，能够允许集体事务的创建、验证、下订单。在这个系统中，当每个订单已经准备要生产时，它的完成稿需要被发送到合适的打印机去。唯一比较复杂的事，这些打印机收取完成稿的方式不一样，有的通过 e-mail，有的通过 FTP，还有的通过安全拷贝协议 SCP (SecureCopyProtocol)。使用 Spring 的集合注入，你可以为这个功能创建一个标准接口：

```
package com.powerise.spring.c2;
```

```
public interface ArtworkSender {  
    public void sendArtwork(String artworkPath, Recipient recipient);  
  
    public String getFriendlyName();  
  
    public String getShortName();  
}
```

通过这个接口，你可以创建多种实现，它们都具有自我描述的能力：

```
package com.powerise.spring.c2;
```

```
public class FtpArtworkSender implements ArtworkSender {
```

```
    public void sendArtwork(String artworkPath, Recipient recipient) {
```

```
// ftp logic here...
}

public String getFriendlyName() {
    return "File Transfer Protocol";
}

public String getShortName() {
    return "ftp";
}

}
```

通过上面这样的实现，你只需简单的将一个 List 传入到 ArtworkManager 类就可以完成工作了。通过 getFriendlyName()方法，你可以显示一个传送方式的列表，在你配置每个公文稿的模版时提供给管理员来选择。还有，如果你面向 ArtworkSender 接口编写代码，你的程序可以与每个独立的实现保持完全的解耦。

### 9.2.7. Understanding Bean Naming: 理解 Bean 命名

Spring 提供了非常复杂的 bean 命名结构，它允许你灵活的处理多种情况。每个 bean 在包含它的 Bean 工厂中至少要有一个唯一的命名。Spring 遵循一个简单的决策过程来决定为每个 bean 使用哪个命名。如果没有指定 id 属性，Spring 会寻找 name 属性，如果具有定义，使用在 name 属性中定义的第一个命名。(我们说第一个命名，是应为在 name 属性中可以定义多个命名；我们在后面会详细介绍。) 如果 name 和 id 属性都没有被指定，Spring 使用 bean 的类名作为命名，当然是假如没有其它的 bean 使用了相同的命名的情况下。下面展示了一个配置的例子，里面使用了全部的三种命名方案。

```
<!-- naming examples -->
<bean id="string1" class="java.lang.String" />
<bean name="string2" class="java.lang.String" />
<bean class="java.lang.String" />
```

这三种方法从技术角度来说是等价的，但是哪种方案才是你的应用程序的最佳选择呢？首先，要避免使用基于类名的自动命名机制。这样就不允许你自由的定义相同类型的多个 bean，所以最好自己给他们命名。这样，如果 Spring 以后修改了默认的命名方式，你的程序依旧可以运行。当选择使用 id 还是 name 的时候，一定使用 id 来指定 bean 的默认命名。XML 的 Id 属性在 Spring 配置文件的 DTD 中被声明为 XML 唯一标记 (XML identity)。这意味着不仅 XML 解析器可以验证你的文件，任何好的 XML 编辑器也可以同样进行验证，因此可以减少错误输入 bean 命名所造成的问题的数量。很重要的，这允许你的 XML 编辑器检验你在本地属性的<ref>标记中引用的 bean 是否存在。

这样做的唯一缺点是，使用 id 属性会限制你只能使用 XML 元素的 ID 标记中允许的字符。如果你发现你不能够在你的命名中使用某个字符，你可以通过 name 属性来指定命名，那样就可以不用受限制于 XML 命名规则了。如此说，你还是应该考虑给你的 bean 通过 id 命名，然后你可以给你的 bean 通过别名来定义一个描述性的命名，在下一节我们将讨论。

#### 9.2.7.1. Bean 别名

Spring 允许一个 bean 拥有多个命名。你可以通过在 bean 的<bean>标记的 name 属性中，

指定逗号分隔或者分号分隔的名称列表来实现这个功能。你可以通过这样代替使用 id 属性或者与 id 属性共同使用。Listing 4-32 展示了一个简单的<bean>配置，对单一的 bean 定义了多个命名。

```
<bean id="name1" name="name2,name3,name4" class="java.lang.String"/>
```

如你所见，我们定义了四个命名：一个使用了 id 属性，其余的三个使用了逗号分隔的名字属性的列表。下面展示了一个简单的 Java 程序，它从 Bean 工厂中四次获取取了同一个 bean，每次使用了不同的命名，并且验证了获取的是同一个 bean。

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class BeanNameAliasing {
    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));

        String s1 = (String) factory.getBean("name1");
        String s2 = (String) factory.getBean("name2");
        String s3 = (String) factory.getBean("name3");
        String s4 = (String) factory.getBean("name4");

        System.out.println((s1 == s2));
        System.out.println((s2 == s3));
        System.out.println((s3 == s4));
    }
}
```

使用上面的配置，这段代码在标准输出中输出了 3 次 true，验证了使用不同命名访问的 bean 实际上是同一个 bean。你可以通过将 bean 的任意一个命名传送给 Bean 工厂的 getAliases(String)方法获得 bean 的别名列表。返回的命名的列表中命名的数量重视少于 bean 所有命名数量 1 个，因为 Spring 认为其中的一个命名是默认的。哪个命名是默认的决定与你如何配置这个 bean。如果你通过 id 属性指定了一个命名，那么它就是默认的。如果你没有使用 id 属性，那么传送命名属性的列表中的第一个命名被作为默认的。Bean 别名是一个奇怪的玩意，因为在你建立一个新的应用程序时一般不会使用。因为如果你将其它的很多 bean 注入到另一个 bean 中，这时他们也许凑巧使用了相同的命名来访问。然而，如果你的应用程序已经应用且开始维护，对其进行修改等等，那么 bean 别名将会变得有用。

考虑如下场景：你拥有一个应用程序，其中有 50 个不同的 bean 通过 Spring 进行配置，它们都需要一个 Foo 接口的具体实现。其中的二十五个 bean 使用了 StandardFoo 实现（bean 命名为 standardFoo），其余的 25 个使用了 SuperFoo 的实现（bean 命名为 superFoo）。六个月后，你的程序已经付诸应用，你决定将前 25 个 bean 转换到使用 SuperFoo 的实现。这样，你有三种选择。

第一种，将 standardFoo 的 bean 的实现类改变为 SuperFoo。这样做的缺点是，你拥有两

个 SuperFoo 类的实例存在，而事实上你只需要一个。进一步说，当配置需要改变的时候你需要修改两个 bean 的配置。

第二种选择是更新需要修改的 25 个 bean 的注入配置，将 bean 的名字从 standardFoo 修改为 superFoo。这样做并不优雅——你要进行查找和替换，但是当需要回滚变动的时候，管理程序并不舒服，需要从你们的版本控制系统重新获取一个配置文件的早前版本。

第三种，也是最理想的方式，删除（或者注释掉）standardFoo bean 的定义，然后给 superFoo 添加一个 standardFoo 的别名。这样的变动工作量很小，将系统恢复到以前的配置也非常简单。

### 9.2.8. Bean Instantiation Modes: Bean 实例化模式

默认情况下，Spring 中的所有 bean 都是单例（singletons）。这意味着 Spring 维护 bean 的唯一实例，所有的以来对象引用同一个实例，对 Bean 工厂的 getBean()方法的每一次调用都返回同一个实例。我们在前面的例子 Listing 4-33 中演示了这个情况，那里我们使用了标识对比（==）而不是 equals()方法来检查那些 bean 是否是同一个。

单例这个术语在 Java 中可替代的表示两种不同的概念：一个对象在应用程序中只有单一的实例，或者单例的设计模式。我们在这里称第一个概念为单例（singleton），而对于模式的概念我们成为单例模式（Singleton）。当人们混淆需要使用单一实例和需要应用单例模式这两个概念的时候，问题显现出来。以下是一个单例模式的经典实现：

```
package com.powerise.spring.c2;

public class Singleton {
    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

这种模式的目的是允许你在你的应用程序中能够维护和访问一个单一的实例，但是使用它会带来耦合度的上升的危害。你的应用程序代码必须知道单例类的确切信息才可以得到它的实例——完全的将这种能力转移到接口的代码中。实际上，单例模式是两种模式的合体。首先，必须的，模式使其可以维护一个对象的单一实例。其次，必要性少一些，模式将对象查找能力完全移出到接口中。使用单例模式还会使任意替换具体实现的工作变得难以进行，因为大部分对象需要单例的实例来直接访问单例对象。当你试图对你的程序进行单元测试的时候，这会使你头疼不堪，因为你不能将这些单例通过测试中的假冒(mock)对象代替。

幸运的是，通过 Spring 你可以享受单例实例化样式，而不用遵从单例设计模式的规范。Spring 中的所有 bean 默认都被创建为单例实例，Spring 使用同一个实例来完成所有对该 bean 的请求。当然，Spring 并不限制于使用单例实例；它也可以为满足每个依赖和每次调用 getBean()而创建该 bean 的新的实例。实现这些对你的程序的代码没有任何的侵入，因为这个原因，我们喜欢称 Spring 为创建模式无关的。这是一个非常强有力的概念。如果你一开始将一个对象设计为单例的，但是后来发现对于多线程的访问这样并不合适，那么你可以将

它修改为一个非单例的对象却对你的代码没有任何的影响。

注意：改变你的 bean 的实例化模式对你的应用程序代码没有影响，但是如果你依赖于 Spring 的生命周期接口，它会造成一些问题。

改变实例化模式，将单例模式转换为非单例模式很简单，如下代码所示：

```
<!-- non-singleton examples -->
<bean id="nonSingleton" class="java.lang.String"
      singleton="false">
    <constructor-arg>
        <value>Rob Harrop</value>
    </constructor-arg>
</bean>
```

如你所见，这个 bean 的声明与前面你看到的其它声明的唯一区别是将单例属性设置为 false。

```
package com.powerise.spring.c2;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class NonSingleton {
    public static void main(String[] args) {

        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\beans.xml"));

        String s1 = (String) factory.getBean("nonSingleton");
        String s2 = (String) factory.getBean("nonSingleton");

        System.out.println("Identity Equal?: " + (s1 == s2));
        System.out.println("Value Equal?: " + s1.equals(s2));
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

运行这个例子会得到如下输出：

Identity Equal?: false

Value Equal?: true

Rob Harrop

Rob Harrop

你可以从中看到这两个 String 对象的值明显是相同的，但是标识却不同，尽管是实际上这两个实例都是使用相同的 bean 命名获取的。

### 9.2.8.1. 选择一个实例化模式

在大部分场景，很容易看出哪种实例化模式更适合。有代表性的是我们发现单例是我们

的 bean 默认的实例化模式。一般而言，单例方式应该在如下的场景下使用：

- **不分状态的共享对象：**当你拥有一个没有状态之分并且有多个依赖对象的对象。因为如果没有状态之分，你就不需要同步，当每次某个依赖对象在工作中需要使用这个 bean 的时候你就不需要创建一个新的实例。
- **只读状态的共享对象：**这与前一点类似，但是你有一些只读状态。这种情况，你依然不需要同步，所以为了满足每个请求而创建新的实例对于这个 bean 来说只不过是增加无用的开销。
- **使用共享状态的共享对象：**如果你的 bean 的状态必须共享，那么单例是最理想的选择。则何种情况，要保证你的状态回写的同步操作的粒度越小越好。
- **具有可记录状态的高吞吐量对象：**如果你有个 bean 在你的程序中被大量使用，那么你也许会发现维护一个单例和其所有 bean 状态回写的同步比持续创建上百个该 bean 的实例能够提供更好的性能。当使用这种方法，一定要努力保持同步的细粒度而不要牺牲一致性。你会发现这种方法在你的程序运行了很长的时间而创建了大量的实例后会显得特别有用，此时你的共享对象只有少量的可回写状态，而这时实例化新的实例将会耗费巨大。

在以下场景你需要考虑使用非单例创建模式：

- **具有可回写状态的对象：**如果你的 bean 拥有一些可回写状态，这时你会发现对它们进行同步操作要比为每个请求创建一个新的实例耗费的资源大。
- **具有私有状态的对象：**在一些情况，你的某个独立对象需要拥有它私有的状态，这样它们能够与其所依赖的 bean 单独处理自己的操作。这种情况下，单例显然不合适，你需要使用非单例模式。

从 Spring 的实例化管理中你可以得到的主要益处是，你的应用程序可以马上从与单例相关的低内存占用率上得到好处，而你并不需要付出太多的工作。这样，如果你发现单例不能够满足你的程序的要求，也可以轻松的修改你的配置，让它们使用非单例模式。

### 9.2.9. Resolving Dependency：依赖解析

在一般的操作中，Spring 能够通过简单查找配置文件解析依赖关系。这样说，Spring 能够确保每个 bean 配置的顺序正确，这样每个 bean 的依赖关系都可以被正确配置。如果 Spring 不这样做，只是无序的创建和配置它们，一个 bean 可能在它依赖的对象初始化前被创建。这显然不是你所要的，它可能在你的应用程序中引发各种各样的问题。

不幸的是，Spring 并不知道你的代码中 bean 间存在的所有依赖关系。例如，有一个 bean，叫 bean A，它里面得到另一个 bean 的实例，叫 bean B，通过构造器中的 getBean() 调用。在这种情况下，Spring 不知道 bean A 依赖于 bean B，这可能会引起 bean A 在 bean B 之前被实例化。你可以通过附加信息通知 Spring 你的 bean 依赖其它 bean，使用<bean>标记的 depends-on 属性。

```
<bean id="A" class="com.powerise.spring.c2.BeanA" depends-on="b"/>
<bean id="B" class="com.powerise.spring.c2.BeanB"/>
```

在这个配置里，我们声明 bean A 依赖于 bean B。Spring 在实例化 bean 的时候会考虑这些内容，保证 bean B 在 bean A 前被实例化。

当开发应用程序时，应避免在应用中采用此类设计；而应通过值注入或者构造器注入约定。不过，如果我们需要将 Spring 代码与遗留系统相整合，那么我们就可能发现代码中定义的依赖关系需要我们为 Spring Framework 提供与其相关的进一步的扩展信息。

## 9.2.10. Auto-Wiring Your Beans: 自动装配你的 Bean

上面所有这些例子中，我们都必须通过配置文件清晰的定义每个 bean 是如何进行装配的。如果你不喜欢自己将你的程序装配起来，你可以尝试让 Spring 自动装配。默认情况，自动装配是被关闭的。开启它，需要指定你想要使用哪种自动装配方法，给你想要自动装配的 bean 配置指定装配属性。

Spring 支持四种自动装配模式：通过命名、通过类型、构造器或自动监测。当使用通过命名的自动装配，Spring 尝试将每个属性连接到一个同名的 bean 上。如此，如果目标 bean 拥有一个叫做 foo 的属性而 Bean 工厂中定义了一个命名为 foo 的 bean，那么 foo 这个 bean 会被分配给目标的 foo 属性。

当使用通过类型的自动装配，Spring 试图将目标 bean 的每个属性与 Bean 工厂中对应的同类型的 bean 连接起来。如果在目标 bean 中有一个 String 类型的属性，而在 Bean 工厂中有 String 类型的 bean，那么 Spring 会将 String 的 bean 和目标类型的 String 属性连接起来。如果你在同一个 Bean 工厂中有多个相同类型的 bean，比如 String，那么 Spring 不能确定使用哪个进行自动装配，而后会抛出异常。

构造器自动装配模式运作方式与通过类型的自动装配类似，只是它通过构造器代替 setter 进行注入。Spring 试图最大数量的匹配构造器中包含的参数。比如，如果你的 bean 拥有两个构造器，一个接受一个 String 另一个接受一个 String 和一个整数，而你的 Bean 工厂中同时有 String 和整数 bean，Spring 会使用那个接受两个参数的构造器。

最后一种模式，自动检测，通知 Spring 在构造器自动装配和通过类型的自动装配间进行选择。如果你的 bean 拥有一个默认构造器（没有参数），那么 Spring 会使用通过类型的方式；否则，会使用构造器方式。配置自动装配 XML 代码：

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
    <bean id="foo" class="com.powerise.spring.c2.autowiring.Foo" />
    <bean id="bar" class="com.powerise.spring.c2.autowiring.Bar" />

    <bean id="targetByName" autowire="byName"
          class="com.powerise.spring.c2.autowiring.Target" />
    <bean id="targetByType" autowire="byType"
          class="com.powerise.spring.c2.autowiring.Target" />
    <bean id="targetConstructor" autowire="constructor"
          class="com.powerise.spring.c2.autowiring.Target" />
    <bean id="targetAutodetect" autowire="autodetect"
          class="com.powerise.spring.c2.autowiring.Target" />
</beans>
```

这种配置你可能看其来感觉非常眼熟。注意每个目标 bean 都拥有一个不同值的自动装配属性。

```
package com.powerise.spring.c2.autowiring;
```

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
```

```
import org.springframework.core.io.FileSystemResource;

public class Target {
    private Foo foo;

    private Foo foo2;

    private Bar bar;

    public Target() {

    }

    public Target(Foo foo) {
        System.out.println("Target(Foo) called");
    }

    public Target(Foo foo, Bar bar) {
        System.out.println("Target(Foo, Bar) called");
    }

    public void setFoo(Foo foo) {
        this.foo = foo;
        System.out.println("Property foo set");
    }

    public void setFoo2(Foo foo) {
        this.foo2 = foo;
        System.out.println("Property foo2 set");
    }

    public void setMyBarProperty(Bar bar) {
        this.bar = bar;
        System.out.println("Property myBarProperty set");
    }

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
            "D:\\workspace\\SpringExample\\src\\autowiring.xml"));
        Target t = null;

        System.out.println("Using byName:\n");
        t = (Target) factory.getBean("targetByName");
    }
}
```

```
System.out.println("\nUsing byType:\n");
t = (Target) factory.getBean("targetByType");

System.out.println("\nUsing constructor:\n");
t = (Target) factory.getBean("targetConstructor");

System.out.println("\nUsing autodetect:\n");
t = (Target) factory.getBean("targetAutodetect");

}
```

在这段代码中，你可以看到 Target 这个类有三个构造器：一个没有参数的构造器，一个接受 Foo 实例的构造器，一个接受 Foo 和 Bar 实例的构造器。对应这三个构造器，Target Bean 有三个属性：两个是 Foo 类型一个是 Bar 类型。每个属性和构造器当被调用时都回相标准输出一段信息。Main 方法会从 Bean 工厂的声明中获取每个目标 bean，触发自动装配过程。下面是运行这个例子的输出：

Using byName:

Property foo set

Using byType:

Property foo set

Property foo2 set

Property myBarProperty set

Using constructor:

Target(Foo, Bar) called

Using autodetect:

Property foo set

Property foo2 set

Property myBarProperty set

从输出中，你可以看到 Spring 什么时候使用通过命名的自动装配，只有一个 foo 属性使用这种方式，因为只有这个属性与配置文件中的 Bean 相对应。当使用通过类型的自动装配，Spring 对三个属性都进行了装配。foo 和 foo2 属性连接到 foo bean，而 myBarProperty 连接到 bar bean。当使用构造器自动装配，Spring 使用了两个参数的构造器，因为 Spring 能够满

足两个参数的 bean，这样就不用回退去使用另一个构造器了。这种情况下，自动检测功能与通过类型的方式相同，因为我们定义了一个默认的构造器。如果我们没这么做，自动检测所做的会与构造器自动装配一样。

### 9.2.10.1. 什么时候使用自动装配

在大部分情况，关于是否应该使用自动装配这个问题的答案当然应该是“不！”，虽然自动装配可以在小规模的程序中节省你的时间，但是它会养成你的坏习惯，在大规模的程序中弹性也不够。使用通过命名的自动装配看起来是个好主意，但是它要求你给属性进行人工命名，才能享受自动装配功能带来的好处。Spring 背后的思想是你可以自由的创建你的类，然后由 Spring 为你工作，而不用你为它们做其它工作。你也许试图使用通过类型的自动装配，直到你意识到这样你只能在 Bean 工厂给每种类型创建一个 bean——当你需要维护同一个类型但具有不同配置的多个 bean 的时候这个限制是个大问题。当使用构造器自动装配的时候也会发生同样的问题。这些模式遵从相同的语义学，如通过类型的和自动检测的自动装配，后者只不过是通过类型的和通过构造器的自动装配绑定在一起而已。

在一些情况下，自动装配可以节省你的时间，但是对它们进行精确的定义实际上并不会消耗你太多的工作，那样你能够从精确的语义、完全自由的属性命名还有规定要管理同一个类型的多少个实例这些地方受益。除了非常小的应用程序，无论如何也要绕开使用自动装配。

### 9.2.11. Checking Dependency: 依赖检查

在创建 bean 实例和装配依赖关系时，Spring 默认情况不会检查 bean 的每个属性是否都有对应的值。很多情况下，你不需要 Spring 进行检查，但是如果你有一个每个属性都必须有值对应的 bean，那么你可以让 Spring 帮你检查。

如本文指出的，这并不总是有效，因为你可能给某些属性提供了默认值，也许只是声明某些特殊的属性一定要有对应的值；Spring 的依赖检查能力并不考虑这种情况。这就是说，在特定的情况下让 Spring 进行这种检查对你是很有用的。大多数情况下，它允许你将检查从代码中移除，只让 Spring 在启动时进行一次检查。

除了默认的不检查以外，Spring 有三种依赖检查的模式：

简单模式、对象模式、全模式。简单模式检查是否所有的集合类和内建类属性都有值对应。这种模式下，Spring 不检查其它类型的属性是否有值对应。这种模式对检查 bean 的所有配置参数是否有值对应非常有意义，因为那些参数一般都是内建类值或者内建类集合值。

对象模式检查那些简单模式所不检查的类型的属性，但是它也不检查简单模式检查的那些属性。所以如果你有一个具有两个属性的 bean，一个是 int 类型另一个是 Foo 类型，那么对象模式检查会检查 Foo 属性是否被指定了值，但是不会对 int 属性进行检查。

全模式检查所有的属性，对简单模式和对象模式所检查的属性都进行检查。下面展示了一个具有两个属性的简单的类：有一个 int 属性，还有一个自己同类型的属性。

```
package com.powerise.spring.c2.depcheck;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class SimpleBean {
    private int someInt;
```

```
private SimpleBean nestedSimpleBean;

public void setSomeInt(int someInt) {
    this.someInt = someInt;
}

public void setNestedSimpleBean(SimpleBean nestedSimpleBean) {
    this.nestedSimpleBean = nestedSimpleBean;
}

public static void main(String[] args) {
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
        "D:\\workspace\\SpringExample\\src\\depcheck.xml"));

    SimpleBean simpleBean1 = (SimpleBean) factory.getBean("simpleBean1");
    SimpleBean simpleBean2 = (SimpleBean) factory.getBean("simpleBean2");
    SimpleBean simpleBean3 = (SimpleBean) factory.getBean("simpleBean3");
}
}

代码中的 main()方法从 Bean 工厂获取三个 bean，都是 SimpleBean 类型。
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
    <bean id="foo" class="com.powerise.spring.c2.autowiring.Foo" />
    <bean id="bar" class="com.powerise.spring.c2.autowiring.Bar" />

    <bean id="targetByName" autowire="byName"
          class="com.powerise.spring.c2.autowiring.Target" />
    <bean id="targetByType" autowire="byType"
          class="com.powerise.spring.c2.autowiring.Target" />
    <bean id="targetConstructor" autowire="constructor"
          class="com.powerise.spring.c2.autowiring.Target" />
    <bean id="targetAutodetect" autowire="autodetect"
          class="com.powerise.spring.c2.autowiring.Target" />
</beans>
```

如你在配置中看到的，这个 Java 程序里，从 Bean 工厂中获取的每个 bean 都具有一个不同的 dependency-check 属性。这里的配置确保通过 dependency-check 属性配置的需要创建的那些属性都有值对应，这样的结果是这个 Java 程序可以无误的运行。你可以尝试注释掉一些 `<property>` 标记，看看会发生什么——Spring 会抛出一个 `org.springframework.beans.factory.UnsatisfiedDependencyException` 的异常，其中会指明是哪个属性应该有值对应但却没有。

## 9.2.12. Bean Inheritance: Bean 继承

某些情况下，你也许需要定义多个相同类型或是实现了共用接口的 bean。这时，如果你希望这些 bean 共享一些配置但又有一些不同的设置，这会是个问题。保持共享配置同步的过程经常出错，在大项目中这样做还非常的耗时。为了解决这个问题，Spring 允许你定义一个<bean>从 Bean 工厂的其它 bean 处继承属性设置。如果需要，你可以重写需要的子 bean 中的任何一个属性的值，这使你可以进行完全的控制，父 bean 可以给你的每个 bean 提供一个基础的配置。下面展示了两个 bean 的简单配置，其中一个继承自另一个。

```
<!-- inheritance examples -->
<bean id="inheritParent"
      class="com.powerise.spring.c2.inheritance.SimpleBean">
    <property name="name">
      <value>Rob Harrop</value>
    </property>
    <property name="age">
      <value>22</value>
    </property>
  </bean>

<bean id="inheritChild"
      class="com.powerise.spring.c2.inheritance.SimpleBean"
      parent="inheritParent">
    <property name="age">
      <value>35</value>
    </property>
  </bean>
```

在这段代码中，你可以看到 inheritChild bean 的<bean>标记中有个附加的属性 parent，它表明 Spring 应该将 inheritParent bean 视为这个 bean 的父 bean。因为 inheritChild bean 的 age 属性有自己的值定义，Spring 将这个值传送给 bean。然而，inheritChild 没有 name 属性的值，所以 Spring 使用了来自 inheritParent bean 的值传递给它。下面展示了前面配置中出现的 SimpleBean 类的代码。

```
package com.powerise.spring.c2.inheritance;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class SimpleBean {
    public String name;

    public int age;

    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(""));

        // Create a bean
        SimpleBean simpleBean = (SimpleBean) factory.getBean("inheritChild");

        System.out.println("Name: " + simpleBean.getName());
        System.out.println("Age: " + simpleBean.getAge());
    }
}
```

```
"D:\\workspace\\SpringExample\\src\\beans.xml");

SimpleBean parent = (SimpleBean) factory.getBean("inheritParent");
SimpleBean child = (SimpleBean) factory.getBean("inheritChild");

System.out.println("Parent:\\n" + parent);
System.out.println("Child:\\n" + child);
}

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

public void setAge(int age) {
    this.age = age;
}

public String toString() {
    return "Name: " + name + "\\n" + "Age:" + age;
}
}
```

如你所见，SimpleBean 的 main()方法从 Bean 工厂获取了 inheritChild 和 inheritParent 的 bean，然后将它们属性的内容写入到标准输出。下面是例子运行的输出：

```
Parent:
Name: Rob Harrop
Age: 22
Child:
Name: Rob Harrop
Age: 35
```

如我们期望的，inheritChild bean 从 inheritParent bean 的 name 属性获得了值，但是在 age 属性还可以提供自己的值。

### 9.2.12.1. 使用 Bean 继承的思考

子 bean 可以从父 bean 继承构造器参数和属性值，所以你可以通过继承使用两种风格的注入。这个层面的灵活性使 bean 继承成为一个构建应用程序时强有力的工具，可以代替大批的 bean 定义。如果你要声明一批具有共享的相同属性值的 bean 时，可以避免通过重复的拷贝和粘贴操作来共享值；取而代之，可以在你的配置中添加继承层次关系。

当你使用继承时，记住，bean 继承不需要符合 Java 的继承层次关系。在五个相同类型的 bean 间建立继承关系是完全可行的。应该把 bean 继承看作模版功能，而不是一个实际的继承功能。但是要注意，如果你修改了子 bean 的类型，那么新的类型必须与父 bean 兼容。

### 9.2.13. 小结

这一章我们总体上讲述了 Spring 内核和 IoC。展示了不同类型 IoC 的很多例子，对在你的应用程序中使用每种机制的优缺点进行了讨论。我们考察了 Spring 提供哪些 IoC 机制，

还有每种机制何时该在你的程序中使用，何时不该。在探索 IoC 时，我们介绍了 Spring 的 Bean 工厂，它是使 Spring 具有 IoC 功能的核心组件，更明确的说，我们专注于介绍 XmlBeanFactory，它允许通过 XML 对 Spring 进行外部配置。

我们还介绍了 Spring 的 IoC 的基础功能，包括 setter 依赖注入、构造器依赖注入、自动装配和 bean 继承。在有关配置的讨论中，我们展示了如何通过 XmlBeanFactory 使用多种不同值配置你的 bean 属性（包括其它的 bean）。

这里只是浅显得介绍了 Spring 和 Spring 的 IoC 容器。后面我们将详细分析 Spring 中的一些与 IoC 相关的功能，而后我们会尽一步细致的分析 Spring 内核提供的其它功能。

## 9.3. Aspect Oriented Programming with Spring: 基于 Spring 的 AOP 编程

AOP (Aspect Oriented Programming) 是一种新兴的编程技术。它可以解决 OOP 和过程化方法不能够很好解决的横切 (crosscut) 问题，如：事务、安全、日志等横切关注。当未来系统变得越来越复杂，横切关注点就成为一个大问题的时候，AOP 就可以很轻松的解决横切关注点这个问题，使得 AOP 编程成为。Spring 是基于 J2EE 的轻量级开源开发框架，其中 Spring AOP 组件实现了面向方面编程。

AOP 是 OOP 的延续，是 Aspect Oriented Programming 的缩写，意思是面向方面编程。AOP 实际是 GoF 设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP 可以说也是这种目标的一种实现。

### 9.3.1. AOP Concepts: AOP 概念

和很多别的技术一样，AOP 有自己独特的概念和术语。解释如何在程序中使用 AOP 之前，我们首先必须正确理解这些术语。下面是 AOP 核心 概念的解释。

**联结点 (Joinpoint):** 一个联结点是程序执行过程中的一个特定点。典型的联结点有：调用一个方法；方法执行这个过程本身；类初始化；对象初始化等。联结点是 AOP 的核心概念之一，它用来定义你在程序的哪里加入新的逻辑。

**通知 (Advice):** 在某一个特定的联结点处运行的代码称为“通知”。通知有很多种，比如在联结点之前执行的前置通知 (before advice) 和在联结点之后执行的后置通知 (after advice)。各种类型的通知包括“around”、“before”和“throws”通知。许多 AOP 框架包括 Spring 都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。

**切入点 (Pointcut):** 一个切入点是用来定义某一个通知该何时执行的一组联结点。通过定义切入点，我们可以精确的控制程序种什么组件接到什么通知。之前我们提过，一个典型的联结点调用就是方法调用，而一个典型的切入点调用就是对某一个类的所有方法调用的集合。通常我们会通过组件复杂的切入点来控制什么时候被执行。AOP 框架必须允许开发者指定切入点：例如，使用正则表达式。

**方面 (Aspect):** 通知和切入点的组合叫做方面，所以，方面定义了一段程序种应该包括的逻辑，以及何时应该执行该逻辑。事务管理是 J2EE 应用中一个很好的横切关注点例子。方面用 Spring 的 Advisor 或拦截器实现。

**织入 (Weaving):** 织入是将方面真正加入程序代码的过程。对于静态 AOP 方案而言，织入是在编译时完成的，通常时在编译过程中增加一个步骤。类似的，动态 AOP 方案则时在程序运行时织入的。Spring 和其他纯 Java AOP 框架一样，在运行时完成织入。

**目标 (Target):** 如果一个对象的执行过程受到某个 AOP 的修改，那么它就叫一个目标对象。目标对象也常被称为被通知对象。

**引入 (Introduction):** 通过引入，我们可以在一个对象中加入新的方法或者属性，以改变它的结构，这样即使该对象的类没有实现某一个接口，我们也可以修改它，使之实现该接口。Spring 允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 `IsModified` 接口，来简化缓存。

上面这些概念有些难以理解，不过我们在看过一些例子就可以慢慢理解。另外，有很多概念无关紧要，因为 Spring AOP 帮你处理了，还有别的一些概念因为 Spring AOP 实现里的抉择而更本不会出现。

### 9.3.2. Types of AOP: AOP 的种类

我们之前提到过有两种不同的 AOP：静态 AOP 和动态 AOP。两者之间的区别就在于什么时候织入，以及如何织入。

#### 9.3.2.1. Static AOP: 静态 AOP

最早 AOP 实现大多数都是静态的。在静态 AOP 中，织入是编译的一个步骤。用 Java 的术语说，静态 AOP 通过直接对字节码进行操作，包括修改代码和扩展类，来完成织入的过程。显然，这种办法生成的程序性能很好，因为最好的结构就是普通的 Java 字节码，在运行时不再需要特别的技巧来确定什么时候应该执行通知。

这种方法的缺点是，如果你想对方面作什么修改，即使只加入一个新的联结点，你都必须重新编译整个程序。AspectJ 是静态 AOP 的一个很好的例子。

#### 9.3.2.2. Dynamic AOP: 动态 AOP

Spring AOP 属于动态 AOP。与静态 AOP 不同，动态 AOP 中织入是在运行时动态完成的。织入具体是如何完成的，各个实现有所不同。Spring AOP 采取的方法是建立代理，然后代理在适当的时候执行通知。动态 AOP 一个弱点在于其性能一般不如静态 AOP，不过动态 AOP 的性能现在也有提高。动态 AOP 的主要优点在于你可以随意修改程序所有方面而不需要重新编译。

#### 9.3.2.3. Choosing an AOP Type: 选择哪种 AOP

选择该使用静态 AOP 还是动态 AOP 事实上是个很难决定的事。不过你不需要只使用其中一种，因为两种各有各的优势。为此，Spring 特别提供了与 AspectJ 衔接的功能，这样你就可以同时方便的使用两种 AOP。总而言之，静态 AOP 出现的时间比较久，功能也往往比较齐全，有很多联结点可以选择。事实上，Spring 只支持 AspectJ 所支持的一部分功能。所以，如果性能对你很重要，或者你要用到 Spring 中没有提供的 AOP 功能，那么你应该使用 AspectJ。除此之外，大部分情况下 Spring AOP 是你最佳的选择。Spring 本身已经提供了很多基于 AOP 的功能，如声明式事务管理 (declarative transaction management)。用 AspectJ 再把这些重写一遍难免浪费时间精力，更重要的是，Spring 提供的版本是经过实战检验的。

最重要得一点是，让你得程序具体需要来决定使用哪种 AOP。另外，如果同时使用两种 AOP 比较适合你的程序，那么就不要特意使用一种。总的来说，我们觉得 Spring AOP 比 AspectJ 用起来简单，所以我们一般首选 Spring AOP。如果我们发现 Spring AOP 功能不够强大，或者程序性能调优时发现有需要，那么我们改用 AspectJ。

### 9.3.3. AOP in Spring: Spring 中的 AOP

Spring 中的 AOP 实现逻辑上可以认为分成两个部分。一部分是与 Spring 其他部分独立的 AOP 核心，提供完全程序控制的 AOP 功能；另一部分则是一组框架服务，它们让程序中

使用 AOP 变得更简单。Spring 的其他一些功能则建立在这两者之上，如事务管理和 EJB 帮助类，他们通过提供基于 AOP 的服务来简化程序设计。

Spring AOP 只有完整的 AOP 功能中的一部分，它只提供了诸如 AspectJ 等其他 AOP 实现中提供的构造的一小部分。事实上，正是因为 Spring AOP 中没有大量的鲜少用到的功能，它才使用起来如此简单，而简单正是 Spring AOP 最强大的地方之一。Spring 设计目标之一就是不提供全部的 AOP 功能，这样 Spring 就能集中力量将 AOP 中最常用的功能设计得很容易使用。为了保证你在需要的时候可以使用 AOP 的全部功能，Spring 的设计师们将 Spring 设计得可以与 AspectJ 完全衔接。

### 9.3.3.1. The AOP Alliance: AOP 联盟

AOP 联盟 (<http://aopalliance.sourceforge.net>) 是很多开源 AOP 项目的代表们所组成的一个联合组织，Spring 创始人 Rod Johnson 也名列其中。AOP 联盟的目的是为各个 AOP 实现定义一个标准的接口。AOP 联盟非常保守，因为他们不希望在 AOP 还在发展时就对它进行太多的限制。所以，到现在 AOP 联盟们只对部分 AOP 功能定义了标准接口。只要在有 AOP 联盟标准接口的地方，Spring 都使用其接口，这样，某些通知可以在不同支持 AOP 联盟接口的 AOP 实现之间重用。

### 9.3.3.2. Hello World in AOP: AOP 的 Hello World

在我们讨论 Spring AOP 实现之前，我们先给出一个简单得例子，为以后的讨论提供一个语境。在这个例子中，我们先写一个简单的输出“World”字样的类，然后我们运行时用 AOP 让这个类的一个对象输入“Hello World!”。

```
package com.powerise.spring.c3;

public class MessageWriter {
    public void writeMessage() {
        System.out.print("World");
    }
}
```

这个 MessageWriter 类再简单不过了，它只有一个方法，该方法输出“World”到 stdout。我们现在要通知它—也就是加一个通知—writeMessage() 应输出“Hello World!”而不是“World”。

为了达到这个效果，在这个方法之前我们需要执行一段代码输出“Hello”，在这个方法之后我们需要执行一段代码输出“！”。用 AOP 术语说，我们需要一个包围通知（Around Advice），就是一个包围联结点的通知。在这个例子中，联结点是对 writeMessage() 的调用。下面是这个包围通知的实现，MessageDecorator 类：

```
package com.powerise.spring.c3;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MessageDecorator implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.print("Hello ");
        Object retVal = invocation.proceed();
    }
}
```

```
        System.out.println("!");
        return retVal;
    }
}
```

MethodInterceptor（方法拦截器）是 AOP 联盟定义的标准接口，用来实现方法调用连接点的包围通知。MethodInvocation 类代表当前被通知的方法调用，我们使用这个类来控制具体什么时候进行方法调用。因为我们用的是包围通知，所以我们在方法执行前进行一些操作，还可以在方法执行完后返回前进行另外一些操作。在下面代码中，我们简单的输出“Hello”到 stdout，然后用 MethodInvocation.proceed() 调用目标方法，最后输出“！”到 stdout。

这个例子最后一步就是将 MessageDecorator 通知织入代码中。为了做到这一步，我们奖励一个 MessageWriter 对象，即目标对象，然后对他创立一个代理，并让代理工厂（proxy factory）织入 MessageDecorator 通知。

```
package com.powerise.spring.c3;

import org.springframework.aop.framework.ProxyFactory;

public class HelloWorldAOPExample {
    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();
        // create the proxy
        ProxyFactory pf = new ProxyFactory();
        pf.addAdvice(new MessageDecorator());
        pf.setTarget(target);
        MessageWriter proxy = (MessageWriter) pf.getProxy();
        // write the messages
        target.writeMessage();
        System.out.println("");
        proxy.writeMessage();
    }
}
```

上面最重要的部分是我们用 ProxyFactory 类来创建目标对象的代理，同时织入通知。我们通过调用 addAdvice()，把 MessageDecorator 通知传给 ProxyFactory，然后通过调用 setTarget() 设定织入的目标对象。我们设定了目标对象，也加入了通知，我们就可以调用 ProxyFactory.getProxy() 来获得一个代理。最后，我们在原来的目标对象和生成代理对象上分别调用 writeMessage() 结果如下：

```
World
Hello World!
```

我们可以看到，没有经过处理的对象上调用 writeMessage() 就是简单的方法调用，stdout 上没有写入新的东西。不过调用代理会执行 MessageDecorator 里的代码，从而生成我们所需要输出的：Hello World!。从这个例子我们可以看到，被通知类不会与 Spring 或者 AOP 联盟接口之间产生依赖关系。Spring AOP 的优美之处，事实上所有 AOP 的优美之处，就在于我们几乎可以通知任何类，不论写该类时有没有考虑到 AOP。唯一的限制，至少在 Spring AOP 里的唯一限制，是我们不能通知 final 类，因为 final 类不能被扩展，所以我们无法生成它的代理。

### 9.3.3.3. Spring AOP Architecture: Spring AOP 架构

Spring AOP 架构的核心是建立在代理上的。当我们建立被通知类的对象时，我们必须使用 ProxyFactory 类加入我们需要织入该类的所有通知，然后为该类的一个对象创建代理。使用 ProxyFactory 创建 AOP 代理时一个完全程序化做法。通常情况下，我们不需要直接这么做，我们可以改用 ProxyFactoryBean 类来声明式的生成代理。不过，理解代理是如何生成的十分重要，所以我们将继续使用程序化生成代理的方法。后面的内容中我们再讨论如何使用 ProxyFactoryBean 来生成代理。

Spring 内部有两种实现代理的方法，JDK 动态代理和 CGLIB 代理。但是 CGLIB 的性能通常都好于 JDK 动态代理，这点主要是 JDK 反射机制性能引起的。理解代理的意义和 Spring 内部怎么使用它们的是提高你程序性能的关键。

- **Spring 里的联结点**

Spring AOP 里最明显的简化之一就是它只支持一种联结点：方法调用。如果你熟悉别的 AOP 实现，如 AspectJ，乍看起来这似乎是个很大的限制，因为 AspectJ 里支持很多种别的联结点。不过实际上，这让 Spring 更易用。

所有联结点中，方法调用无疑是最有用的一种，实际的编程中很多用到 AOP 的任务都是用它来完成的。记住，如果你需要通知除了方法调用之外别的联结点，你总可以同时使用 Spring 和 AspectJ。

- **Spring 里的方面**

在 Spring AOP 中，一个方面是由一个实现 Advisor（通知者）接口类表示的。Spring 中提供了一些使用方便的实现 Advisor 接口的类，这样你就不用在自己的程序中创建各种不一样的 Advisor 的实现了。Advisor 接口由两个子接口，IntroductionAdvisor（引入通知者）和 PointcutAdvisor（切入点通知者）。所有用切入点控制该在哪些联结点运行通知的 Advisor 都应该实现 PointcutAdvisor 接口。

在 Spring 中，引入被认为是一种特殊的通知。我们可以通过使用 IntroductionAdvisor 接口来控制应该对哪些类实施引入。

### 9.3.3.4. About the ProxyFactory Class: 关于 ProxyFactory 类

ProxyFactory 类控制着 Spring AOP 中织入和创建代理过程。在真正创建代理之前，我们首先要设定目标对象。之前我们看到，我们可以用 setTarget() 来完成这个步骤。ProxyFactory 内部将生成代理的过程转交给一个 DefaultAopProxyFactory 对象来完成，后者又根据程序中的设置将其转交给一个 Cglib2AopProxy 或者 JdkDynamicProxy 来完成。

通过使用 ProxyFactory 类，我们可以控制哪些方面需要织入到代理中去。前面我们提到过，我们只能在被通知代码中织入方面，也就是一个通知和一个切入点的结合。不过，有时候我们希望在目标类所有方法调用时执行通知，而不只一部分方法调用。这种情况下，ProxyFactory 提供 addAdvice()（加入通知）方法。该方法内部将我们提供的通知包在一个 DefaultPointcutAdvisor 对象内，然后将其切入点定为“所有方法调用”。DefaultPointcutAdvisor 是 PointcutAdvisor 的标准实现。

我们可以用同一个 ProxyFactory 类来生成多个不同的代理，各个代理的方面可以不同。为了方便使用，ProxyFactory 类有 removeAdvice()（清除通知）和 removeAdvisor()（清除通知者）方法，通过这些方法我们可以删除之前加入代理工厂的任何通知和通知者。要查询一个 ProxyFactory 对象是否包含某一个通知，我们可以调用 adviceIncluded() 方法，其参数是我们想要查询的通知。

注意，ProxyFactory 类的好一个方法已经被淘汰，现在我们应该使用如 addAdvice() 之类

的方法。在 JavaDoc 中我们可以找到这些方法的详细资料，尽量不要使用已经淘汰的方法，因为以后版本的 Spring 里很有可能不再提供这些方法。另外，每一个淘汰的方法都有一个替换的方法。

### 9.3.3.5. Creating Advice Spring: 在 Spring 中创建通知

Spring 支持五种不同的通知，如下表所示：

通知名称	接口	说明
前置通知	org.springframework.aop.MethodBeforeAdvice	使用前置通知可以在联结点执行前进行自定义的操作。不过 Spring 里面只有一种联结点，即方法调用，所以前置通知事实上就是让你能在方法调用前进行一些操作。前置通知可以访问调用的目标方法，也可以对该方法的参数进行操作，不过它不能影响方法调用的本身。
后置通知 (活返回后通知)	org.springframework.aop.AfterReturningAdvice;	后置通知在联结点处的方法调用已经完成，并且已经返回一个值时运行。后置通知可以访问调用的目标方法，以及该方法的参数返回值。因为等到通知执行时该方法已经调用，后置通知完全不能影响方法调用本身。
包围通知	org.aopalliance.intercept.MethodInterceptor	Spring 中的包围通知模仿 AOP 联盟的“方法拦截器”标准。包围通知可以在目标方法之前和之后运行，我们也可以定义在什么时候调用目标方法。如果需要，我们也可以另写自己的逻辑而完全不调用目标方法。
抛出通知	org.springframework.aop.ThrowsAdvice	抛出通知仅当方法调用抛出一个异常时才被调用，它在目标方法调用返回时执行。抛出通知可以只接受特定的异常，如果这样的话，那么我们可以访问抛出异常的方法、目标方法及参数。
引入	org.springframework.aop.IntroductionAdvisor	Spring 将引入看成一个特殊的拦截器。使用引入拦截器，我们可以定义通知引入的方法的实现。

经验表明，以上五种通知和方法调用联结点可以完成 90% 的 AOP 工作。对于剩下的不常用的 10%，我们可以使用 AspectJ。

- 通知的接口

之前我们对 ProxyFactory 类的讨论中讲到，我们有两种方法将一个通知加到代理中去。一是直接使用 addAdvice() 方法，二是使用 addAdvicor() 方法和 Advisor (通知者) 类。起初每一种通知的接口都时相对独立的，不过后来所有的通知接口被纳入到一个继承体系中来。这个体系是建立在 AOP 联盟接口上的。如下图所示：

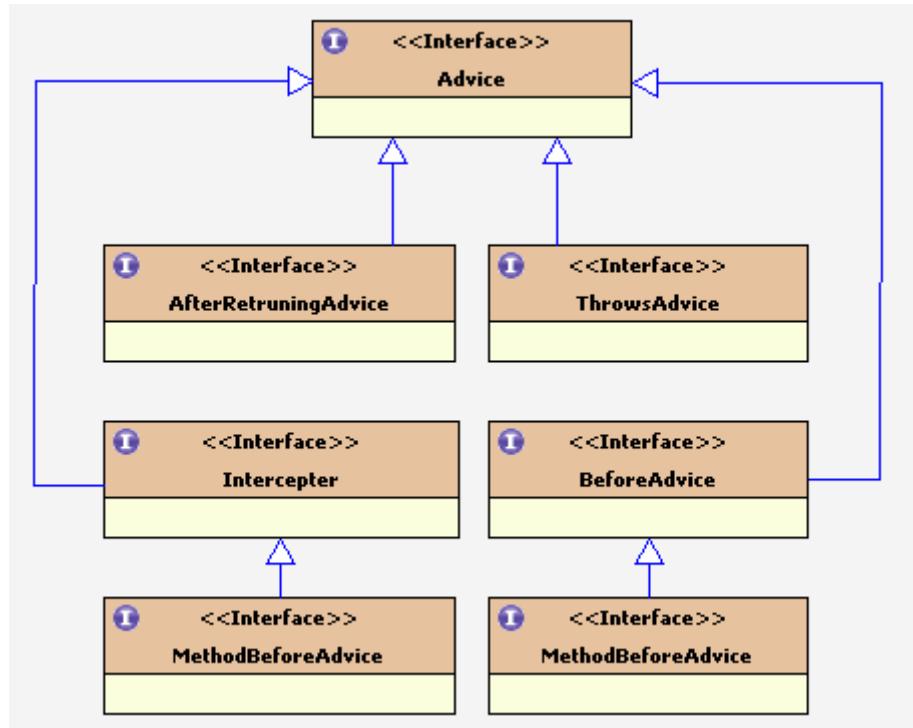


图 9-6: Spring 不同的通知接口

### ● 创建前置通知

前置通知是 Spring 中最有用的通知之一，它可以修改传给目标方法的参数，也可以抛出一个异常来阻止方法执行。我在这里将展示两个前置通知的例子，一是在目标方法执行前想 stdout 写出该方法名字的简单例子，二是限制调用一个对象的方法的简单安全性通知。

```

package com.powerise.spring.c3;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleBeforeAdvice implements MethodBeforeAdvice {
    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();
        // create the proxy
        ProxyFactory pf = new ProxyFactory();
        pf.addAdvice(new SimpleBeforeAdvice());
        pf.setTarget(target);
        MessageWriter proxy = (MessageWriter) pf.getProxy();
        // write the messages
        proxy.writeMessage();
    }

    public void before(Method method, Object[] args, Object target)
            throws Throwable {
    }
}
  
```

```
        System.out.println("Before method: " + method.getName());
    }
}
```

这段代码中我们可以看到，我们先创建了一个 `MessageWriter` 类，然后用一个 `SimpleBeforeAdvice` 类的实例通知它。`SimpleBeforeAdvice` 类实现 `MethodBeforeAdvice`（前置通知）接口，后者仅仅定义了一个方法，`before()`。AOP 框架负责在联结点处的方法调用之前调用该 `before()` 方法。记住，我们现在使用的是 `addAdvice()` 提供的缺省切入点，其定义为“该类中所有方法”。方法 `before()` 接受三个参数，被调用的方法、将被传给该方法的参数和目标对象（为一个 `Object`）。`SimpleBeforeAdvice` 类使用 `before()` 方法的 `Method` 参数获得目标方法的名字，然后在 `stdout` 上输出该信息。运行上述程序我们得到以下结果：

```
Before method: writeMessage
```

```
World
```

我们可以看到，`writeMessage()` 的输出出现在屏幕上，不过在此之前，我们可以看到 `SampleBeforeAdvice` 生成的输出。

### ■ 用前置通知管理方法调用的安全性

上一个例子太过简单，不能真正展示 AOP 的力量。现在我们使用一个前置通知来检查用户身份，进而决定是否可以执行该方法。如果用户身份无效，我们的前置通知会抛出一个异常，于是目标方法就不会执行。这里的例子为了简单起见，允许用户任何密码登入，但是只允许一个写在程序里的用户名访问目标方法。无论如何，我们可以通过这个例子看到使用 AOP 实现横切关注点是多么容易。

```
package com.powerise.spring.c3.security;

public class SecureBean {
    public void writeSecureMessage() {
        System.out.println("Every time I learn something new, "
            + "it pushes some old stuff out my brain");
    }
}
```

我们要求用户验证身份，所以我们需要某种办法将他们的信息记录下来。下面我们展示用来记录用户身份的 `UserInfo` 类。

```
package com.powerise.spring.c3.security;

public class UserInfo {
    private String userName;

    private String password;

    public UserInfo(String userName, String password) {
        this.userName = userName;
        this.password = password;
    }

    public String getPassword() {
        return password;
    }
}
```

```
    }

    public String getUserName() {
        return userName;
    }

}
```

这个类没有什么特别的，它简单的保存用户信息以便以后访问。下面显示 SecurityManager 类，它负责验证用户身份，并将该身份保存起来。

```
package com.powerise.spring.c3.security;

public class SecurityManager {
    private static ThreadLocal threadLocal = new ThreadLocal();

    public void login(String userName, String password) {
        // assumes that all credentials
        // are valid for a login
        threadLocal.set(new UserInfo(userName, password));
    }

    public void logout() {
        threadLocal.set(null);
        int x = 0;
    }

    public UserInfo getLoggedOnUser() {
        return (UserInfo) threadLocal.get();
    }
}
```

我们的程序用 SecurityManager 来认证用户身份，之后它来获取当前认证的用户相关信息。验证身份用 login()方法。在真正的程序中，login()方法通常会用数据库或 LDAP 目录来检查验证申请，不过我们现在假设所有的用户都能通过验证。login()方法为用户创建一个 UserInfo 对象并通过一个 ThreadLocal 对象将之保存在本进程中。相反的，logout()方法将 ThreadLocal 中所保存的不论什么值都设为 null。最后，getLoggedOnUser()类返回当前认证的用户 UserInfo 对象。如果没有认证的用户，该方法返回 null。

我们需要检查当前没有通过验证的用户，如果有，该用户能否访问 SecureBean 的方法。为此我们创建一个前置通知来检查 SecurityManager.getLoggedOnUser()返回的 UserInfo 对象是否是我们允许的用户。该通知（SecurityAdvice）代码如下：

```
package com.powerise.spring.c3.security;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class SecurityAdvice implements MethodBeforeAdvice {
```

```
private SecurityManager securityManager;

public SecurityAdvice() {
    this.securityManager = new SecurityManager();
}

public void before(Method method, Object[] args, Object target)
        throws Throwable {
    UserInfo user = securityManager.getLoggedOnUser();

    if (user == null) {
        System.out.println("No user authenticated");
        throw new SecurityException(
            "You must login before attempting to invoke the method: "
            + method.getName());
    } else if ("robh".equals(user.getUserName())) {
        System.out.println("Logged in user is robh - OKAY!");
    } else {
        System.out.println("Logged in user is " + user.getUserName()
            + " NOT GOOD :(");
        throw new SecurityException("User " + user.getUserName()
            + " is not allowed access to method " + method.getName());
    }
}
```

SecurityAdvice 类在构造函数中创建一个 SecurityManager 的实例，并将其保存在一个属性中。注意，我们写的程序和 SecurityAdvice 不需要共用一个 SecurityManager 对象，因为其数据都是保存在 ThreadLocal 中的。在 before()方法中，我们简单的核查验证的用户名是不是 robh，如果是，我们就允许访问，不然我们就抛出一个异常。注意，我们需要检查 UserInfo 是否是 null，因为有可能没有用户通过验证。如下代码：

```
package com.powerise.spring.c3.security;

import org.springframework.aop.framework.ProxyFactory;

public class SecurityExample {
    public static void main(String[] args) {
        // get the security manager
        SecurityManager mgr = new SecurityManager();
        // get the bean
        SecureBean bean = getSecureBean();
        // try as robh
        mgr.login("robh", "pwd");
        bean.writeSecureMessage();
```

```
mgr.logout();
// try as janm
try {
    mgr.login("janm", "pwd");
    bean.writeSecureMessage();
} catch (SecurityException ex) {
    System.out.println("Exception Caught: " + ex.getMessage());
} finally {
    mgr.logout();
}
// try with no credentials
try {
    bean.writeSecureMessage();
} catch (SecurityException ex) {
    System.out.println("Exception Caught: " + ex.getMessage());
}
}

private static SecureBean getSecureBean() {
    // create the target
    SecureBean target = new SecureBean();
    // create the advice
    SecurityAdvice advice = new SecurityAdvice();
    // get the proxy
    ProxyFactory factory = new ProxyFactory();
    factory.setTarget(target);
    factory.addAdvice(advice);
    SecureBean proxy = (SecureBean) factory.getProxy();
    return proxy;
}
}
```

在getSecureBean()方法中，我们创建了SecureBean类得一个代理，它有一个SecurityAdvice通知。我们返回这个代理。当调用方调用该代理上得任何方法时，SecurityAdvice 得一个实例会先收到该调用并进行安全检查。在main()中我们测试了三种不同的情况，我们分别在两个不同的用户身份和没有用户身份的情况下调用 SecureBean.writeSecureMessage()方法。因为SecurityAdvice 只在当前用户时 robh 时才允许方法调用。程序运行输出结果如下：

```
Logged in user is robh - OKAY!
Every time I learn something new, it pushes some old stuff out my brain
Logged in user is janm NOT GOOD :(
Exception Caught: User janm is not allowed access to method writeSecureMessage
No user authenticated
Exception Caught: You must login before attempting to invoke the method:
writeSecureMessage
```

我们可以看到，只有第一次 SecureBean.writeSecureMessage()方法被执行。另外两次调

用被 SecurityAdvice 抛出的 SecurityException 阻止了。

这个例子很简单，不过它突出了前置通知的用处。实现安全性能时前置通知的一个典型例子。当我们需要修改目标方法的参数时前置通知也很有用。

### ● 创建后置通知

如其名称所示，后置通知在方法调用联结点返回后调用。既然该方法已经调用了，我们已经没有办法修改它的参数。我们只能读取，而不能修改目标方法的执行路径或者让目标方法不执行。这些限制并不出人意料；真正出人意料的是我们也不能在后置通知中修改目标方法的返回值。我们只能进行一些新的操作。虽然我们不能修改返回值，但是我们还可以抛出一个异常，这样调用方就只会看到这个异常而不是返回值。

在这里我们将使用两个后置通知的例子。第一个例子在目标方法执行后简单的向 stdout 输出一条信息。第二个例子演示如何通过后置通知向一个方法加入新的出错检查。我们展示一个为加密算法生成密钥的 KeyGenerator（密钥生成）类。很多加密算法有一个问题，就是一小部分可能的密钥被认为很弱。一个弱的密钥导致在不知道密钥的情况下要获得原来的信息变得很容易许多。对于 DES 算法来说，有  $2^{56}$  次方个可能的密钥，其中，4 个值很弱，另有 12 个值比较弱。虽然随机生成的密钥是这 16 个值中的一个可能性非常小 ( $2^{52}$  次方中一次)，但是进行这个检查是非常容易的，以至于不检查几乎是松散松懈的表现。我们在第二个例子中，我们用后置通知检查 KeyGenerator 生成的密钥是否是弱密钥，如果是，我们抛出一个异常。

SimpleAfterReturningAdvice（简单的后置通知）类展示了如何使用后置通知方法返回后向 stdout 输出一条信息。

```
package com.powerise.spring.c3;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleAfterReturningAdvice implements AfterReturningAdvice {

    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();

        // create the proxy
        ProxyFactory pf = new ProxyFactory();

        pf.addAdvice(new SimpleAfterReturningAdvice());
        pf.setTarget(target);

        MessageWriter proxy = (MessageWriter) pf.getProxy();

        // write the messages
        proxy.writeMessage();
    }
}
```

```
public void afterReturning(Object returnValue, Method method,
    Object[] args, Object target) throws Throwable {
    System.out.println("");
    System.out.println("After method: " + method.getName());
}
```

这个例子和我们之前看到的 SimpleBeforeAdvice（简单的前置通知）类并没有太大的不同。注意，AfterReturningAdvice（后置通知）接口仅仅定义了一个方法，afterReturning()，其参数为目标方法调用的返回值、目标方法的引用、目标方法的参数以及目标对象。运行这个例子，我们得到下面的输出：

```
World
After method: writeMessage
```

这个输出和前置通知的非常像，只是如我们所希望的，该通知输出的信息出现在 writeMessage()本身输出的信息之后。

后置方法的一个好处是对可能返回无效的方法的返回值进行进一步的出错检查。在我们之前所讲述的情况下，一个密钥生成器生成的密钥可能对某个加密算法而言是弱密钥。在理想情况下，密钥生成器应该对这些密钥进行测试，不过因为一般生成弱密钥的几率非常低，很多密钥生成器没有进行检查。我们可以使用后置通知来通知生成密钥的方法进行进一步的检查。下面是一个非常原始的密钥生成类 KeyGenerator：

```
package com.powerise.spring.c3.crypto;

import java.util.Random;

public class KeyGenerator {
    public static final long WEAK_KEY = 0xFFFFFFFF0000000L;
    public static final long STRONG_KEY = 0xACDF03F590AE56L;

    private Random rand = new Random();

    public long getKey() {
        int x = rand.nextInt(3);

        if(x == 1) {
            return WEAK_KEY;
        } else {
            return STRONG_KEY;
        }
    }
}
```

这个密钥生成器显然极不安全，不过真正的密钥生成器要很多年才会生成一个弱密钥。我们为了演示将用这个三分之一几率生成弱密钥的密钥生成器。如下代码，我们看到 WeakKeyCheckAdvice（弱密钥检查通知）检查 getKey() 的结果是不是一个弱密钥。

```
package com.powerise.spring.c3.crypto;
```

```
import java.lang.reflect.Method;  
  
import org.springframework.aop.AfterReturningAdvice;  
  
public class WeakKeyCheckAdvice implements AfterReturningAdvice {  
  
    public void afterReturning(Object returnValue, Method method,  
        Object[] args, Object target) throws Throwable {  
  
        if ((target instanceof KeyGenerator)  
            && ("getKey".equals(method.getName()))) {  
            long key = ((Long) returnValue).longValue();  
  
            if (key == KeyGenerator.WEAK_KEY) {  
                throw new SecurityException(  
                    "Key Generator generated a weak key. Try again");  
            }  
        }  
    }  
}
```

在 afterReturning()方法中，我们首先检查在联结点处执行的方法是不是 getKey()。如果是我们就检查其结果是不是弱密钥。如果是弱密钥，那么我们就抛出一个 SecurityException（安全异常）来通知调用方。如下代码：

```
package com.powerise.spring.c3.crypto;  
  
import org.springframework.aop.framework.ProxyFactory;  
  
public class AfterAdviceExample {  
    public static void main(String[] args) {  
        KeyGenerator keyGen = getKeyGenerator();  
  
        for (int x = 0; x < 10; x++) {  
            try {  
                long key = keyGen.getKey();  
                System.out.println("Key: " + key);  
            } catch (SecurityException ex) {  
                System.out.println("Weak Key Generated!");  
            }  
        }  
    }  
}  
  
private static KeyGenerator getKeyGenerator() {
```

```
KeyGenerator target = new KeyGenerator();

ProxyFactory factory = new ProxyFactory();
factory.setTarget(target);
factory.addAdvice(new WeakKeyCheckAdvice());

return (KeyGenerator) factory.getProxy();
}
```

创建完以 KeyGenerator 为目标的通知代理后，AfterAdviceExample 类尝试生成 10 个密钥。如果在某此生成时抛出了 SecurityException 异常，那么我们向 stdout 写一条信息告诉用户生成了一个弱密钥。否则我们显示生成的密钥本身。在我们机器运行一次的结果如下：

```
Key: 48658904092028502
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
Weak Key Generated!
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
```

我们看到 KeyGenerator 有时会生成弱密钥，而 WeakKeyCheckAdvice 保证生成弱密钥时会抛出 SecurityException 异常。

### ● 创建包围通知

包围通知功能上和前置通知加后置通知类似，除了一个重要区别：我们可以修改返回值。不仅如此，我们还可以阻止目标方法执行。这意味着包围通知我们可以将目标方法完全换成新的代码。Spring 里的包围通知模仿 MethodInterceptor 接口的拦截器。包围通知有很多作用，你会发现 Spring 的很多功能，如远程代理支持（remote proxy support）和事务管理（transaction support），都是由拦截器完成的。拦截器也是剖析程序运行的好办法。这里我们的例子都是基于拦截器的。

我们不再为方法拦截器建立一个简单的例子（前面已经介绍过）。该例子展示了如何用简单的方法拦截在方法执行前后输出信息。注意前面的这个例子说明 MethodInterceptor（方法拦截器）类的 invoke() 方法所接受的参数与 MethodBeforeAdvice（前置通知）和 AfterReturningAdvice（后置通知）不同。也就是说，invoke() 不接受目标对象、目标方法及其参数为参数。不过，通过提供给 invoke() 的 MethodInvocation（方法调用）对象，我们还是可以访问上述信息。我们会在下面的例子看到。

在这个例子中，我希望通过某种通知来获得目标对象方法的运行性能。具体一点说，我们要知道每个方法执行了多久。为了达到这个目的，我们使用 Spring 中提供的 Stopwatch（停表）类，另外，我们显然需要 MethodInterceptor，因为我们要在方法执行前开始计时，并且在方法开始后停止计时。下面代码是我们要用 Stopwatch 类和包围通知进行性能剖析的 WorkerBean 类。

```
package com.powerise.spring.c3.profiling;
```

```
public class WorkerBean {  
    public void doSomeWork(int noOfTimes) {  
        for (int x = 0; x < noOfTimes; x++) {  
            work();  
        }  
    }  
  
    private void work() {  
        System.out.print("");  
    }  
}
```

这个类非常简单。doSomeWork()方法接受一个参数 noOfTimes，然后执行 work()方法 noOfTimes 遍。work()方法简单调用 System.out.print()，传入参数为空字符串。这样我们可以防止编译器优化 work()方法，进而影响对 work()调用。

下面代码是 ProfilingInterceptor（剖析拦截器）类，它使用 StopWatch()类来对方法运行时间进行剖析。如下代码：

```
package com.powerise.spring.c3.profiling;  
  
import java.lang.reflect.Method;  
  
import org.aopalliance.intercept.MethodInterceptor;  
import org.aopalliance.intercept.MethodInvocation;  
import org.springframework.util.StopWatch;  
  
public class ProfilingInterceptor implements MethodInterceptor {  
  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
        // 启动 StopWatch  
        StopWatch sw = new StopWatch();  
        sw.start(invocation.getMethod().getName());  
        Object returnValue = invocation.proceed();  
        sw.stop();  
        dumpInfo(invocation, sw.getTotalTimeMillis());  
        return returnValue;  
    }  
  
    private void dumpInfo(MethodInvocation invocation, long ms) {  
        Method m = invocation.getMethod();  
        Object target = invocation.getThis();  
        Object[] args = invocation getArguments();  
        System.out.println("Executed method: " + m.getName());  
        System.out.println("On object of type: " + target.getClass().getName());  
        System.out.println("With arguments:");  
    }  
}
```

```
        for (int x = 0; x < args.length; x++) {
            System.out.print("    > " + args[x]);
        }
        System.out.print("\n");
        System.out.println("Took: " + ms + " ms");
    }
}
```

在 MethodInterceptor 接口定义的唯一方法 invoke() 中，我们创建一个 StopWatch 实例，然后立刻启动计时，并调用 MethodInvocation.proceed() 来执行目标方法。一旦方法执行完毕，我们就记录下返回值，停止计时，并将用毫秒数以及 MethodInvocation 对象传给 dumpInfo() 方法。最后，我们返回 MethodInvocation.proceed() 所返回的值，以保证调用方法会得到正确的返回值。在这种情况下，我们不想影响程序的调用堆，我们只想“偷看”方法调用。如果需要的话，我们可以完全修改调用堆，比如说把调用目标方法改为调用另一个对象，调用一个远程服务，或者干脆在拦截器中重写整个方法的逻辑并返回一个值。

方法 dumpInfo() 简单的将关于方法调用的信息，以及方法执行所用的时间写到 stdout。在 dumpInfo() 方法的头三行我们可以看到如何通过 MethodInvocation 对象来确定调用的是什么方法、原来的方法调用目标和所用的参数。

下面代码 ProfilingExample 类中首先用一个 ProfilingInterceptor 通知一个 WorkerBean 实例，然后剖析 doSomeWork() 方法。

```
package com.powerise.spring.c3.profiling;

import org.springframework.aop.framework.ProxyFactory;

public class ProfilingExample {
    public static void main(String[] args) {
        WorkerBean bean = getWorkerBean();
        bean.doSomeWork(10000000);
    }

    private static WorkerBean getWorkerBean() {
        WorkerBean target = new WorkerBean();

        ProxyFactory factory = new ProxyFactory();
        factory.setTarget(target);
        factory.addAdvice(new ProfilingInterceptor());

        return (WorkerBean) factory.getProxy();
    }
}
```

现在这段代码已经很熟悉了。在我们的机器上运行得到以下输出：

```
Executed method: doSomeWork
On object of type: com.powerise.spring.c3.profiling.WorkerBean
With arguments:
    > 10000000
```

Took: 3078 ms

在这个输出中我们可以看到哪个方法被调用、目标对象是什么类的、参数是什么，以及方法运行了多长时间。

### ● 创建抛出通知

抛出通知在联结点之后运行（记住 Spring 中的联结点永远是方法调用），这和后置通知一样。不过抛出通知只在方法抛出一个异常时才会执行。另外，抛出通知对程序运行本身不作什么改变，这也和后置通知一样。使用抛出通知时我们不能选择忽已经抛出的异常而替换目标方法返回一个值，我们能作的对程序运行造成影响的修改只有改变抛出异常的类型。事实上是一个很强大的思想，它可以大大简化程序开发。比如说我们需要用一个 API，它所抛出的种种异常设计得非常糟糕。我们可以抛出通知去通知该 API 中所有类，这样我们可以重新设计异常继承结构，使之更易于管理和有描述性。当然，我们也可以用抛出通知提供一个集中得错误日志，这样就可以减少散布在程序各处得记录错误日志代码的数量。

我们使用 ThowsAdvice(抛出通知)接口来实现抛出通知。与之前的其他接口不同，ThrowsAdvice 接口没有定义任何方法，它只是 Spring 使用的一个标志接口 (marker interface，该类接口不能有实现)，其原因是 Spring 允许类型抛出通知，也就是可以定义我们的通知具体接受哪几类异常。Spring 通过反射机制寻找固定的方法签名的方式来实现类型抛出通知。Spring 寻找两种不同的方法签名。展示这个问题的最好方法是看下面简单的例子。

```
package com.powerise.spring.c3;

public class ErrorBean {
    public void errorProneMethod() throws Exception {
        throw new Exception("Foo");
    }

    public void otherErrorProneMethod() throws IllegalArgumentException {
        throw new IllegalArgumentException("Bar");
    }
}
```

以上代码有两个方法的简单 Bean，其两种方法抛出两个不同类的异常。下面 SimpleThrowsAdvice 类展示了 Spring 在抛出通知上寻找两个不同方法签名。

```
package com.powerise.spring.c3;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleThrowsAdvice implements ThrowsAdvice {
    public static void main(String[] args) throws Exception {
        ErrorBean errorBean = new ErrorBean();

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(errorBean);
        pf.addAdvice(new SimpleThrowsAdvice());
    }
}
```

```
ErrorBean proxy = (ErrorBean) pf.getProxy();

try {
    proxy.errorProneMethod();
} catch (Exception ignored) {

}

try {
    proxy.otherErrorProneMethod();
} catch (Exception ignored) {

}

}

public void afterThrowing(Exception ex) throws Throwable {
    System.out.println("/**/");
    System.out.println("Generic Exception Capture");
    System.out.println("Caught: " + ex.getClass().getName());
    System.out.println("/**/\n");
}

public void afterThrowing(Method method, Object[] args, Object target,
    IllegalArgumentException ex) throws Throwable {
    System.out.println("/**/");
    System.out.println("IllegalArgumentException Capture");
    System.out.println("Caught: " + ex.getClass().getName());
    System.out.println("Method: " + method.getName());
    System.out.println("/**/\n");
}
}
```

main()方法中的代码非常容易看懂，我们主要讲解两个 afterThrowing()方法。Spring 首先会查看抛出通知中的一个或者多个叫做 afterThrowing()的 public 方法。该方法的返回类型并不重要，不过我们觉得一般设为 void 为好，因为该方法不能放回任何有意义的值。SimpleThrowsAdvice 类的第一个 afterThrowing()方法接受一个名为 Exception 的参数。我们可以将参数的类型定义为任何一个 Exception 类。如果我们的程序不需要知道是哪个方法抛出了异常，以及它的参数，那么这个方法非常好用。注意我们的这个例子会接受所有的 Exception 及子类，除非这个子类有自己的 afterThrowing()方法。

第二个 afterThrowing()方法有四个参数：抛出异常的方法、该方法的参数、方法调用的目标对象以及抛出的异常。这些参数的顺序很重要，而且我们必须标出所有的四个参数。注意，第二个 afterThrowing()会捕获所有类型为 IllegalArgumentException(或其子类)的异常。运行上述例子得到以下结果：

\*\*\*

Generic Exception Capture  
Caught: java.lang.Exception

\*\*\*

\*\*\*  
IllegalArgumentException Capture  
Caught: java.lang.IllegalArgumentException  
Method: otherErrorProneMethod  
\*\*\*

我们看到，简单的抛出一个 `Exception` 会运行第一个 `afterThrowing()`方法，而抛出的是 `IllegalArgumentException` 时会运行第二个 `afterThrowing()`方法。对于抛出的每一个异常，Spring 只会运行一个 `afterThrowing()`方法，如上代码所示，Spring 会选择签名与抛出的异常最匹配的一个方法。如果抛出通知定义了两个 `afterThrowing()`方法，他们的异常类型一样，但是一个只有一个参数而另一个有四个参数，那么 Spring 会选择运行那个有四个参数的 `afterThrowing()`方法。

我们提过抛出通知在很多情况下都很有用；我们可以用它重新定义整个异常继承结构，或者构建集中的异常日志。我们现在调试运行中的程序时异常通知格外有用，因为我们无需修改程序代码就可以添加新的日志代码。

#### ● 选择通知种类

一般而言，使用那种通知是由应用程序的具体要求决定的，不过我们应当选择适合要求的最精确的通知种类。这就是说，如果前置通知够用时就不要使用包围通知。在多数情况中，其他三种通知能做的事包围通知都能做，不过用包围通知做我们要做的事可能太小题大做了。使用最精确的种类可以使代码的意图更清晰，同时也减少出错的可能性。比如说一个计算方法调用次数的通知，如果用前置通知写计数器就行了，不过如果是包围通知，你还需要记得调用该方法，以及返回其返回值。这些小事情容易产生更多似是而非的错误。使用最精确的通知种类可以减少这类错误的发生范围。

### 9.3.4. Advisors and Pointcuts in Spring: Spring 里的通知者

#### 和切入点

到目前为止我们全部例子都用 `ProxyFactory.addAdvice()`方法为代理设定通知。我们之前讲过，该方法会在后台委派给 `addAdvisor()`方法，而后者会创建一个 `DefaultPointcutAdvisor` 实例并将切入点设为“所有方法调用”。这样，目标的所有方法就都被通知了。在某些情况下，如果 AOP 做日志时，这样做可能正是我们需要的，可是在别的的情况下，我们希望只通知相关的方法而不是所有的方法。

当然，我们可以简单的在通知内检查被通知的方法是不是正确的，不过这种方法有几个缺点。第一，将接受的方法名称列表直接写进代码中会降低通知的复用性。使用切入点可以控制哪些方法被通知而无需将该列表写入通知里，显然这样做复用性比较好。在通知中直接写入方法名称列表的后两个缺点与性能相关。为了确定被调用的方法应不应该被通知，每次目标上的任何一个方法被调用时我们都需要做一个检查。这显然会影响程序的性能。当使用切入点时，每个方法会被检查一遍，其结果会被缓冲起来日后使用。另外一个不使用切入点来限制被通知方法的性能问题是，Spring 生成代理时可以优化没有被通知的方法，这样没有被通知的方法执行起来比较快。

强烈建议你不要在通知中直接检查方法，只要有空能，你就使用切入点，控制应该通知目标的那些方法。不过，有些时候我们还是不得不在通知中写入一些检查。比如在之前例子中，我们用后置通知来截获 KeyGenerator 类来生成弱密钥。这种通知和目标类关系非常密切，所以在通知中检查目标正确与否是明智的选择。我们称通知和目标之间的这种关系为“目标亲和”。总的来说，如果通知没有目标亲和度或者目标亲和度很低一就是说如果通知适用于任何类型或者适用于很多种不同的类型一那么我们应该使用切入点。当通知有很强的目标亲和度时，我们应该尝试在通知中进行检测，以确保通知的用法是正确的。因为这种通知被不正确的使用时，我们会遇到奇怪的错误，通知中检查则可以减少这种错误。我们同时也建议避免通知不需要通知的方法。因为使用通知会明显降低执行的速度，这对整个应用程序的性能会造成影响。

### 9.3.4.1. The Pointcut Interface: 切入点接口

在 Spring 中创建切入点就要实现 Pointcut (切入点) 接口。

```
public interface Pointcut{  
    ClassFilter getClassFilter();  
    MethodMatcher getMethodMatcher();  
}
```

我们从上面代码中可以看到，Pointcut 接口定义了两个方法，getClassFilter() (获得类过滤器) 和 getMethodMatcher() (获得方法匹配器)，两者分别返回一个 ClassFilter (类过滤器) 实例和一个 MethodMatcher (方法匹配器) 实例。如果我们要重新写一个自己的切入点，那么我们就要实现一个 ClassFilter 接口和一个 MethodMatcher 接口。我们一般不需要这样，因为 Spring 已经提供了一些 Pointcut 的实现，一般情况下这些实现足够满足我们的需求。

Spring 需要确定一个 Pointcut 是否适合用于某个方法时，它会先用 Pointcut.getClassFilter() 返回的 ClassFilter 测试该方法的类。下面是 ClassFilter 的接口代码：

```
public interface ClassFilter{  
    boolean matches(Class clazz);  
}
```

我们看到，ClassFilter 接口只定义了一个方法，matches() (与……匹配)，其参数为一个代表被检测类的 Class 实例。如果切入点适用于该类，那么 matches() 返回 true，否则返回 false。

相比之下 MethodMatcher 接口比较复杂。

```
public interface MethodMatcher{  
    boolean matches(Method m,Class targetClass);  
    boolean isRuntime();  
    boolean matches(Method m,Class targetClass,Object[] args);  
}
```

Spring 支持两种不同的 MethodMatcher，静态的和动态的。一个 MethodMatcher 是哪一种取决于 isRuntime() 方法的返回值。在使用一个 MethodMatcher 之前，Spring 会调用 isRuntime()，如果返回值是 false，那么该 MethodMatcher 就是静态的，如果返回值是 true 就是动态的。

如果切入点是静态的，那么 Spring 会针对目标上的每一个方法调用一次 MethodMatcher 的 matches(Method, Class) 方法，其返回值被缓冲起来以方便日后调用该方法时使用。这样，对于每一个方法的适用性测试只会进行一次，之后调用该方法时不会再调用 matches()。

如果切入点是动态的。Spring 仍然会在目标方法第一次调用时用 matches(Method, Class) 进行一个静态的测试来检查其总的适用性。不过，如果该测试返回 true，那么此基础上，每

次该方法调用时 Spring 会再次调用 matches(Method , Class, Object[])方法。这样，一个动态的 MethodMatcher 可以根据一次具体的方法调用，而不仅仅是方法本身，来决定切入点是否适用。

显然，静态切入点一也就是 MethodMatcher 为静态的切入点一的性能比动态切入点要好的多，因为它们不需要在每次调用时重新检查。不过，使用动态切入点来决定是否执行通知要比使用静态切入点更加灵活。总而言之，我们建议尽量使用静态切入点。不过，如果执行通知的开销非常大的话，我们可以使用动态切入点来避免在不需要的时候执行通知。

Spring 提供了静态切入点和动态切入点的抽象基类，我们可以很少重写 Pointcut 实现。

### ● 已有的切入点实现

Spring 提供了 Pointcut 接口的七个实现，其中两个抽象类分别用来简化创建静态和动态切入点，另外 5 个实体类中，一个用来组合不同的切入点，一个用来处理流程切入点，一个做简单的名字匹配，最后两个用正则表达式定义切入点。下面是这些实现的概要。

实现类	说明
org.springframework.aop.support.ComposablePointcut (可组合切入点)	ComposablePointcut 可以通过 union() (并集) 和 intersection() (交集) 等操作组合两个或者两个以上的切入点。
org.springframework.aop.support.ControlFlowPointcut (流程切入点)	ControlFlowPointcut 是一种特殊的切入点，它匹配另一个方法的流程中包含的所有方法，也就是另一个方法执行时直接或间接调用的所有方法。
org.springframework.aop.support.DynamicMethodMatcherPointcut (动态方法匹配器切入点)	DynamicMethodMatcherPointcut 的意图是提供动态切入点的基类。
org.springframework.aop.support.JdkRegexpMethodPointcut (JDK 正则表达式切入点)	通过 JdkRegexpMethodPointcut 我们可以用 JDK 支持的正则表达式定义切入点。此类需要 JDK1.4 版本或更新的版本。
org.springframework.aop.support.NameMatchMethodPointcut (名称匹配方法切入点)	使用 NameMatchMethodPointcut 我们可以给出一个方法名列表，然后切入点简单的测试目标方法名是否匹配。
org.springframework.aop.support.Perl5RegexpMethodPointcut (Perl5 正则表达式切入点)	通过 Perl5RegexpMethodPointcut 我们可以用 Perl5 的正则表达式语法定义切入点。该类需要 Jakarta ORO 项目。
org.springframework.aop.StaticMethodMatcherPointcut (静态方法匹配切入点)	StaticMethodMatcherPointcut 的意图是提供写静态切入点的基类。

我们这里将讲解五种基本的实现。至于比较高级的 ComposablePointcut 和 ControlFlowPointcut 我们将在后面的内容讨论。

### ● 使用 DefaultPointcutAdvisor (缺省切入点通知者)

在使用任何 Pointcut 实现之前，我们必须先生成一个 Advisor (通知者)，更准确的说是一个 PointcutAdvisor (切入点通知者)。我们之前讲过，Spring 用 Advisor 来表示方面，即通知和切入点的结合，其中切入点定义那些方法应该被通知，以及如何通知。Spring 提供了四个 PointcutAdvisor 实现，现在我们只讨论其中一个 DefaultPointcutAdvisor 。 DefaultPointcutAdvisor 是用来结合一个 Pointcut 和一个 Advice 的简单 PointcutAdvisor。

### ● 使用 StaticMethodMatcherPointcut (静态方法匹配器切入点) 创建静态切入点

这里我们以 StaticMethodMatcherPointer 为基类创建一个简单的静态切入点。

StaticMethodMatcherPointer 只需要我们实现一个方法，matches(Method,Class)，Pointcut 的其他实现是自动完成的。虽然我们只需要实现这一个方法，但是我们有时候也会像本例一样覆盖 getClassFilter()方法，保证只有正确的方法才被通知。

这个例子中我们有两个类 BeanOne 和 BeanTwo，两者的方法名都一样。

```
package com.powerise.spring.c3.staticpc;
```

```
public class BeanOne {  
    public void foo() {  
        System.out.println("foo");  
    }  
  
    public void bar() {  
        System.out.println("bar");  
    }  
}
```

BeanTwo 类和 BeanOne 类方法完全一样。这个例子中，我们想要用同一个 DefaultPointAdvisor 来创建两个代理，不过其通知只适用于 BeanOne 类的 foo()方法。要达到这个目的，我们需要创建一个 SimpleStaticPointcut（简单的静态切入点）类。

```
package com.powerise.spring.c3.staticpc;
```

```
import java.lang.reflect.Method;  
  
import org.springframework.aop.ClassFilter;  
import org.springframework.aop.support.StaticMethodMatcherPointcut;  
  
public class SimpleStaticPointcut extends StaticMethodMatcherPointcut {  
  
    public boolean matches(Method method, Class cls) {  
        return ("foo".equals(method.getName()));  
    }  
  
    public ClassFilter getClassFilter() {  
        return new ClassFilter() {  
            public boolean matches(Class cls) {  
                return (cls == BeanOne.class);  
            }  
        };  
    }  
}
```

我们可以看到已经实现了 StaticMethodMatcher 基类的要求 matches(Method,Class)方法。我们的实现在方法名称是 foo 时返回 true，否则返回 false。注意我们同时也覆盖了 getClassFilter()方法，其返回的 ClassFilter 实例的 matches()方法当且仅当类是 BeanOne 时才会返回 true。通过这个静态切入点，我们定义了只有 BeanOne 类的方法匹配，同时只有该类

的 foo()方法匹配。

下面代码显示了 SimpleAdvice 类，该类简单的在方法调用的前后各写出一条信息。

```
package com.powerise.spring.c3.staticpc;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class SimpleAdvice implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println(">> Invoking " + invocation.getMethod().getName());
        Object retVal = invocation.proceed();
        System.out.println(">> Done");
        return retVal;
    }
}
```

下面代码中我们可以看到一个简单的驱动程序，它用 SimpleAdvice 和 SimpleStaticPointcut 类来创建一个 DefaultPointcutAdvisor 类实例。

```
package com.powerise.spring.c3.staticpc;

import org.aopalliance.aop.Advice;
import org.springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class StaticPointcutExample {
    public static void main(String[] args) {
        BeanOne one = new BeanOne();
        BeanTwo two = new BeanTwo();

        BeanOne proxyOne;
        BeanTwo proxyTwo;

        // create pointcut, advice and advisor
        Pointcut pc = new SimpleStaticPointcut();
        Advice advice = new SimpleAdvice();
        Advisor advisor = new DefaultPointcutAdvisor(pc, advice);

        // create BeanOne proxy
        ProxyFactory pf = new ProxyFactory();
        pf.addAdvisor(advisor);
        pf.setTarget(one);
        proxyOne = (BeanOne) pf.getProxy();
```

```
// create BeanTwo proxy
pf = new ProxyFactory();
pf.addAdvisor(advisor);
pf.setTarget(two);
proxyTwo = (BeanTwo) pf.getProxy();

proxyOne.foo();
proxyTwo.foo();

proxyOne.bar();
proxyTwo.bar();

}

}
```

注意，两个代理是用同一个 DefaultPointcutAdvisor 实例生成的，一个是 BeanOne 实例的代理，另一个是 BeanTwo 实例的代理。最后，两个代理的 foo() 和 bar() 方法都被调用了一遍。运行上面例子得到以下结果：

```
>> Invoking foo
foo
>> Done
foo
bar
bar
```

我们看到只有在执行 BeanOne 类的 foo() 方法时，SimpleAdvice 才会真正被执行。限制那些方法会执行通知非常简单，我们下面将讨论不同的代理选择时会看到，这也是提高程序性能的关键。

#### ● 使用 DynamicMethodMatcherPointcut (动态方法匹配器切入点) 创建动态切入点

这里我们将看到创建动态切入点和创建静态切入点没有太大区别。比如说，我们可以像以下代码为一个类创建动态切入点。

```
package com.powerise.spring.c3.dynamicpc;

public class SampleBean {
    public void foo(int x) {
        System.out.println("Invoked foo() with: " + x);
    }

    public void bar() {
        System.out.println("Invoked bar()");
    }
}
```

在本例中我们只要通知 foo() 方法，不过与前一个例子不同，我们只在其 int 参数不为 null 时才通知它。

与静态切入点一样，Spring 也为动态切入点提供了一个基类，DynamicMethodMatcherPointcut (动态方法匹配器切入点)。DynamicMethodMatcherPointcut

类只有一个抽象方法, matches(Method,Class)以控制静态检查。下面是 SimpleDynamicPointcut 类。

```
package com.powerise.spring.c3.dynamicpc;

import java.lang.reflect.Method;

import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class SimpleDynamicPointcut extends DynamicMethodMatcherPointcut {
    public boolean matches(Method method, Class cls) {
        System.out.println("Static check for " + method.getName());
        return ("foo".equals(method.getName()));
    }

    public boolean matches(Method method, Class cls, Object[] args) {
        System.out.println("Dynamic check for " + method.getName());

        int x = ((Integer) args[0]).intValue();

        return (x != 100);
    }

    public ClassFilter getClassFilter() {
        return new ClassFilter() {
            public boolean matches(Class cls) {
                return (cls == SampleBean.class);
            }
        };
    }
}
```

我们覆盖 getClassFilter()方法, 这样做的结果是, 我们不需要在方法匹配器中检查类了。这对动态检查十分重要。虽然我们只需要实现动态检查, 但是我们也实现了静态检查, 因为我们知道 bar()方法永远不需要接受通知。在静态检查中表明这一点, Spring 就永远不会针对该方法作动态检查。如果我们没有写静态检查, 那么每次 bar()方法调用时 Spring 都要做一个动态检查, 尽管检查的结果永远是 false。在 matches(Method,Class,Object[])方法中, 我们看到, 当 foo()的 int 参数为 100 时我们返回 false, 否则返回 true。另外注意, 在动态检查中, 我们知道检查的方法一定是 foo(), 因为别的方法都不能通过静态检查。下面代码显示了该切入点是如何工作的。

```
package com.powerise.spring.c3.dynamicpc;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
```

```
import com.powerise.spring.c3.staticpc.SimpleAdvice;

public class DynamicPointcutExample {
    public static void main(String[] args) {
        SampleBean target = new SampleBean();

        // create advisor
        Advisor advisor = new DefaultPointcutAdvisor(
            new SimpleDynamicPointcut(), new SimpleAdvice());

        // create proxy
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        SampleBean proxy = (SampleBean)pf.getProxy();

        proxy.foo(1);
        proxy.foo(10);
        proxy.foo(100);

        proxy.bar();
        proxy.bar();
        proxy.bar();
    }
}
```

注意，我们重复使用了静态切入点例子中的通知类。不过现在，只有 `foo()` 的前两次调用应该接到通知。动态检查会防止 `foo()` 的第三次接到调用通知，而静态检查会防止 `bar()` 的调用接到通知。运行上面例子我们得到如下输出：

```
Static check for foo
Static check for bar
Static check for hashCode
Static check for clone
Static check for toString
Static check for foo
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 1
>> Done
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 10
>> Done
Dynamic check for foo
```

```
Invoked foo() with: 100
```

```
Static check for bar
```

```
Invoked bar()
```

```
Invoked bar()
```

```
Invoked bar()
```

只有 `foo()` 方法的前两次调用接到通知。注意，`bar()` 的几次调用一次都没有经过动态检查，因为我们对其进行了静态检查。有意思的是，`foo()` 方法事实上经过了两次静态检查，一次在初始化阶段，一次在它第一次被调用时。

我们可以看到动态切入点比静态切入点更灵活，不过使用它们开支比较大，所以我们要慎重选择动态切入点。

### ● 使用简单的名字匹配

创建切入点时，我们往往只需要方法名字匹配，而无需理会方法的签名和返回类型。在这种情况下，我们可以用 `NameMatchMethodPointcutAdvisor`（名称匹配方法切入点）来匹配一组方法名称，而不需要创建 `StaticMethodMatcherPointcut` 的子类。使用 `NameMatchMethodPointcut` 时，我们不理会方法的签名，所以名称 `foo` 可以匹配 `foo()`，也可以匹配 `foo(int)`。我们再来看一个有 4 个方法的简单类。

```
package com.powerise.spring.c3.namepc;
```

```
public class NameBean {  
    public void foo() {  
        System.out.println("foo");  
    }  
  
    public void foo(int x) {  
        System.out.println("foo " + x);  
    }  
  
    public void bar() {  
        System.out.println("bar");  
    }  
  
    public void yup() {  
        System.out.println("yup");  
    }  
}
```

这个例子中我们想用 `NameMatchMethodPointcut` 匹配 `foo()`、`foo(int)` 和 `bar()` 方法，也就是说，我们想匹配 `foo` 和 `bar` 两个名字。

```
package com.powerise.spring.c3.namepc;
```

```
import org.springframework.aop.Advisor;  
import org.springframework.aop.framework.ProxyFactory;  
import org.springframework.aop.support.DefaultPointcutAdvisor;  
import org.springframework.aop.support.NameMatchMethodPointcut;
```

```
import com.powerise.spring.c3.staticpc.SimpleAdvice;

public class NamePointcutExample {
    public static void main(String[] args) {
        NameBean target = new NameBean();

        // create advisor
        NameMatchMethodPointcut pc = new NameMatchMethodPointcut();
        pc.addMethodName("foo");
        pc.addMethodName("bar");
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

        // create the proxy
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        NameBean proxy = (NameBean)pf.getProxy();

        proxy.foo();
        proxy.foo(999);
        proxy.bar();
        proxy.yup();
    }
}
```

我们不需要为切入点创建一个类，我们简单的创建一个 NameMatchMethodPointcut 实例就可以了。注意我们用 addMethodName()方法向切入点加了两个名称： foo 和 bar。运行上面的例子，我们得到如下输出：

```
>> Invoking foo
foo
>> Done
>> Invoking foo
foo 999
>> Done
>> Invoking bar
bar
>> Done
yup
```

foo()、foo(int)和 bar()三个方法因为我们切入点而接到通知，不过 yup()方法不会接到通知。

### ● 用正则表达式创建切入点

上面我们讨论了如何匹配一组方法名称。不过如果我们事先不知道所有方法名称，而只知道他们的模式呢？比如说，我们想匹配所有名称以 get 开头的方法。在这种情况下，我们可以 使用 JdkRegexpMethodPointcut ( JDK 正则表达式方法切入点 ) 或者 Perl5RegexpMethodPointcut ( Perl5 正则表达式方法切入点 ) 都行，来用正则表达式匹配方法

名。

```
package com.powerise.spring.c3.regexppc;

public class RegexpBean {
    public void foo1() {
        System.out.println("foo1");
    }

    public void foo2() {
        System.out.println("foo2");
    }

    public void bar() {
        System.out.println("bar");
    }
}
```

用基于正则表达式的切入点，我们可以匹配类中所有名字以 foo 开头的方法。如下所示：

```
package com.powerise.spring.c3.regexppc;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.JdkRegexpMethodPointcut;

import com.powerise.spring.c3.staticpc.SimpleAdvice;

public class RegexpPointcutExample {
    public static void main(String[] args) {
        RegexpBean target = new RegexpBean();

        // create the advisor
        JdkRegexpMethodPointcut pc = new JdkRegexpMethodPointcut();
        pc.setPattern(".*foo.*");
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

        // create the proxy
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        RegexpBean proxy = (RegexpBean)pf.getProxy();

        proxy.foo1();
        proxy.foo2();
        proxy.bar();
    }
}
```

```
    }  
}
```

注意，我们不需要为切入点创建一个类，我们只需要创建 `JdkRegexpMethodPointcut` 的一个实例（用 `Perl5RegexpMethodPointcut` 也一样方便），设定匹配的模式就好了。有意思的是正则表达式的模式。在匹配方法名称时，Spring 会用该方法完全限定名称，所以对于 `foo1()`，Spring 会检查 `com.powerise.spring.c3.regexppc.RegexpBean.foo1` 是否匹配，因此我们的模式要以.\*开头。这个方法非常强大，因为它允许我们匹配一个包里所有方法而无需知道包里有哪里类，这些类又有哪些方法。运行上面例子输出如下：

```
>> Invoking foo1  
foo1  
>> Done  
>> Invoking foo2  
foo2  
>> Done  
bar
```

只有 `foo1()` 和 `foo2()` 被通知了，因为 `bar()` 方法不匹配正则表达式模型。

### ● 通知者的便利实

对于很多 Pointcut 实现，Spring 也提供相对 Advisor 的便利实现，这些实现也可以当做 Pointcut 使用。比如说，我们在之前的例子中同时使用了 `NameMatchMethodPointcut` 和 `DefaultPointcutAdvisor`。我们可以直接使用 `NameMatchMethodPointcutAdvisor`（名称匹配方法切入点通知者），这样更加简洁。

```
package com.powerise.spring.c3.namepc;  
  
import org.springframework.aop.framework.ProxyFactory;  
import org.springframework.aop.support.NameMatchMethodPointcutAdvisor;  
  
import com.powerise.spring.c3.staticpc.SimpleAdvice;  
  
public class NamePointcutUsingAdvisor {  
    public static void main(String[] args) {  
        NameBean target = new NameBean();  
  
        // create advisor  
        NameMatchMethodPointcutAdvisor           advisor      = new  
NameMatchMethodPointcutAdvisor(  
            new SimpleAdvice());  
        advisor.addMethodName("foo");  
        advisor.addMethodName("bar");  
  
        // create the proxy  
        ProxyFactory pf = new ProxyFactory();  
        pf.setTarget(target);  
        pf.addAdvisor(advisor);  
        NameBean proxy = (NameBean) pf.getProxy();
```

```
    proxy.foo();
    proxy.foo(999);
    proxy.bar();
    proxy.yup();

}

}
```

上面代码中我们没有创建 NameMatchMethodPointcut 实例，而是直接在一个 NameMatchMethodPointcutAdvisor 实例上设定切入点得细节。这样 NameMatchMethodPointcutAdvisor 既是一个 Advisor，又是一个 Pointcut。

你可以在 org.springframework.aop.support 包的 JavaDoc 中找到关于 Advisor 各个便利实现的完整细节。使用两种方法性能上没有什么区别，两种方法编程途径也差不多，除了第二种方法可以少写一点点代码。我们比较喜欢第一种方法，因为我们觉得其意图在代码中体现得要稍微明显一点。

### 9.3.5. All About Proxies: 关于代理

到目前为止，我们只粗略地讨论了 ProxyFactory 生成的代理。Spring 中有两种代理 JDK Proxy 类生成的 JDK 代理和 CGLIB Enhancer 类生成的基于 CGLIB 的代理。理解两种不同是程序中 AOP 代码性能调到最优的关键。我们在这里讨论两种代理的不同之处，以及这些不同之处会如何影响程序的性能。

实际上这两种代理的实现很相似，他们的性能也差不多。之所以会有两种代理是因为 JDK1.3 下 Proxy 类的性能很差。为了解决这个问题，Spring 提供了 CGLIB 代理，它在 1.3 版的 JVM 上的性能和 JDK1.4 代理比较接近。

CGLIB 代理的本已是解决 JDK1.3 下 Proxy 类的性能问题，当时的实现尽可能的和 JDK 代理保持一致。这样唯一的坏处是 Spring 不能完全使用 CGLIB 提供的其他性能。在 Spring 的新版本中这种情况得到了改善。现在 CGLIB 代理经过了大量的优化，并且在很多情况下性能远远超过 JDK 代理。在我们看两种代理实现的不同之处前，我们先要理解生成代理到底需要干什么。

#### 9.3.5.1. 理解代理

代理的核心任务就是拦截方法调用，并在需要的时候执行匹配某方法的通知链。通知的管理和调用基本与代理无关，所以这是由 Spring AOP 架构负责的。不过，代理还是需要拦截所有的方法调用，并在必要时将该调用传给 AOP 框架，再由后者调用通知。

除了这个核心任务之外，代理还需要支持些别的功能。我们可以通过设定，让一个代理用 AopContext 类暴露它自己，这样我们就可以在目标对象内读取代理并执行上面的通知方法。代理需要保证用 ProxyFactory.setExposeProxy() 设定了这个选项后，它被正确的暴露出来。不仅如此，所有代理缺省状态下实现 Advised (被通知) 接口，这样在代理生成后我们还能修改它的通知链，当然还有其他用处。代理还需要保证任何会放回 this (也就是代理的目标) 的方法都会返回代理而不是目标。

我们可以看到，一个典型的代理要完成很多事。JDK 代理和 CGLIB 代理都实现上面讲的这些逻辑。但是两种代理实现这些逻辑相差很远。

#### 9.3.5.2. 使用 JDK 动态代理

JDK 代理是 Spring 提供的最基本的代理。和 CGLIB 不同，JDK 代理只能代理接口，不

能代理类。这样，你想代理的每一个对象都必须实现至少一个接口。总的来说，用接口而不是类是很好的设计理念，但是我们不是总能这么做，特别是使用第三方代码或者遗留代码时。在这种情况下，我们只能使用 CGLIB 代理。

使用 JDK 代理时，JVM 会拦截所有的方法调用，并将其传至代理的 invoke()方法。然后 invoke()会决定该方法有没有被通知，如果有，那么代理会调用通知链，然后利用反射调用目标方法本身。另外，invoke()方法会执行上一节中所讨论的所有逻辑。

JDK 代理在调用 invoke()之前不能判断一个方法是否接受通知。这就意味着，即使对于代理上没有被通知的方法，invoke()也会被调用，所有的检查也会进行一遍，最后该方法还是会用反射来调用。显然，每次方法调用时都会造成额外的开销，即使代理只是用反射执行了一个没有通知的方法，别的什么也没有干。

如果要让 ProxyFactory 使用 JDK 代理，那么我们需要用 setProxyInterfaces()方法设定一组需要代理的接口。

### 9.3.5.3. 使用 CGLIB 代理

使用 JDK 代理时，如何处理一个特定的方法调用的决定是在程序运行时作出的，也就是每次方法被调用时。使用 CGLIB 代理可以避开这种处理方法，转而使用一种性能上有很大提高的办法。对 CGLIB 内部操作的详细讨论超过了我们学习的范围，不过总的来讲，CGLIB 会在运行中随时为代理创建新类的字节码，并尽可能的重用已经生成类的字节码。这样我们就可以对其进行大量调优。

在首次创建 CGLIB 代理时，CGLIB 会询问 Spring 每个方法应该如何处理。这意味着很多决定在 JDK 代理中每次 invoke()调用时都要进行，而在 CGLIB 代理中只需要一次。因为 CGLIB 会生成真正的字节码，所以我们在处理方法时可以非常灵活。比如说，CGLIB 生成的字节码会很恰当的直接调用未被通知的方法，这样代理所造成的额外开支就大大减少了。另外，CGLIB 生成的字节码会很恰当的直接调用未被通知的方法，这样代理所造成的额外开支就大大减少了。另外，CGLIB 代理还会检查该方法会不会返回 this，如果不会，那么它会允许该方法直接被调用，这再次大大减少了运行开支。

CGLIB 代理处理固定通知链的方法与 JDK 代理也不一样。如果你能保证一个代理生成之后就不会改变了，那么它就是一个固定通知链。缺省状态下，我们在代理生成后还可以改变其通知者和通知，虽然实际上这很少用到。CGLIB 有处理固定通知链的方法用来降低执行通知链的运行开销。

除了以上的优化外，CGLIB 代理执行被通知方法时也使用了生成字节码的能力，这对提高性能有一点帮助。因此 CGLIB 代理执行被通知方法时的效率也略微高过 JDK 代理。

### 9.3.5.4. 比较两者性能

到目前为止，我们只是粗略地讨论了不同代理地实现有什么不同。这里我们运行一个简单的测试程序来比较 CGLIB 代理和 JDK 代理性能差异。

```
package com.powerise.spring.c3.proxies;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.Advised;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class ProxyPerfTest {
```

```
public static void main(String[] args) {
    ISimpleBean target = new SimpleBean();

    Advisor advisor = new DefaultPointcutAdvisor(new TestPointcut(),
        new NoOpBeforeAdvice());

    runCglibTests(advisor, target);
    runCglibFrozenTests(advisor, target);
    runJdkTests(advisor, target);
}

private static void runCglibTests(Advisor advisor, ISimpleBean target) {
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);

    ISimpleBean proxy = (ISimpleBean) pf.getProxy();
    System.out.println("Running CGLIB (Standard) Tests");
    test(proxy);
}

private static void runCglibFrozenTests(Advisor advisor, ISimpleBean target) {
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    pf.setFrozen(true);

    ISimpleBean proxy = (ISimpleBean) pf.getProxy();
    System.out.println("Running CGLIB (Frozen) Tests");
    test(proxy);
}

private static void runJdkTests(Advisor advisor, ISimpleBean target) {
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(target);
    pf.addAdvisor(advisor);
    pf.setInterfaces(new Class[] { ISimpleBean.class });

    ISimpleBean proxy = (ISimpleBean) pf.getProxy();
    System.out.println("Running JDK Tests");
    test(proxy);
}

private static void test(ISimpleBean bean) {
```

```
long before = 0;
long after = 0;

// 测试 advised 方法
System.out.println("Testing Advised Method");
before = System.currentTimeMillis();
for (int x = 0; x < 500000; x++) {
    bean.advised();
}
after = System.currentTimeMillis();
;

System.out.println("Took " + (after - before) + " ms");

// 测试 unadvised 方法
System.out.println("Testing Unadvised Method");
before = System.currentTimeMillis();
for (int x = 0; x < 500000; x++) {
    bean.unadvised();
}
after = System.currentTimeMillis();
;

System.out.println("Took " + (after - before) + " ms");

// 测试 equals() 方法
System.out.println("Testing equals() Method");
before = System.currentTimeMillis();
for (int x = 0; x < 500000; x++) {
    bean.equals(bean);
}
after = System.currentTimeMillis();
;

System.out.println("Took " + (after - before) + " ms");

// 测试 hashCode() 方法
System.out.println("Testing hashCode() Method");
before = System.currentTimeMillis();
for (int x = 0; x < 500000; x++) {
    bean.hashCode();
}
after = System.currentTimeMillis();
;
```

```

System.out.println("Took " + (after - before) + " ms");

// 对 Advised 方法进行测试
Advised advised = (Advised) bean;

System.out.println("Testing Advised.getProxiedInterfaces() Method");
before = System.currentTimeMillis();
for (int x = 0; x < 500000; x++) {
    advised.getProxiedInterfaces();
}
after = System.currentTimeMillis();

System.out.println("Took " + (after - before) + " ms");

System.out.println(">>>\n");
}
}

```

我们看到上面代码测试了三种不同的代理：标准 CGLIB 代理，固定通知链的 CGLIB 代理，以及 JDK 代理。对于每种代理我们进行以下五种测试：

**被通知方法：**接受通知的方法。我们在这个测试中使用的是一个空的前置通知以减小通知本身对性能测试的影响。

**未被通知方法：**代理上一个不接受通知的方法。通常代理上会有很多未被通知的方法。这个测试反映不同的代理处理未被通知方法的性能。

**equals()方法：**这个测试反映执行 equals()方法的额外开销。这当你的代理在 HashMap 或类似的集合中作为键值时尤为重要。

**hashCode()方法：**和 equals()方法一样， hashCode()方法在代理作为 HashMap 或者类似的集合的键值时非常重要。

**执行 Advised 接口上的方法：**如我们之前所说，缺省状态下代理会实现 Advised（被通知）接口，这样我们可以在代理生成后修改它，也可以查询有关该代理的信息。这个测试检查各种代理类型处理 Advised 接口上方法的速度。

我们测试的配置如下图所示：



图 9-7

运行测试时，我们将 JVM 内存堆的初始大小设定为 256M，以减小内存堆的大小对测试的影响，使用 JDK 版本为 1.6.0-beta。我们得到的结果如下表：

测试内容	CGLIB(标准)	CGLIB(固定)	JDK
被通知方法	266	156	281
未被通知方法	156	47	187

equals()	31	31	391
hashCode()	141	31	156
advised.getProxiedInterfaces()	109	94	203

从上表中的结果可以看到，CGLIB 代理的性能要高于 JDK 代理。执行被通知方法时，标准 CGLIB 的性能和 JDK 代理性能差距有限，不过当我们使用固定通知链 CGLIB 代理时，性能差异就非常可观了。对于未被执行通知，CGLIB 代理代理性能远高于 JDK 代理。CGLIB 代理的 equals() 和 hashCode() 方法也一样要明显快过 JDK 代理。注意 equals() 比 hashCode() 要慢。这是因为 equals() 有特别的处理来保证 equals() 含义不会因为代理而改变。至于 Advised 接口上的方法，我们注意到 CGLIB 也会比较快，但是不会快太多。这是因为 Advised 方法在 intercept() 的较前部分中执行，所以很多别的方法需要的逻辑被跳过了。

### 9.3.5.5. 该使用哪个代理

一般来说，CGLIB 代理可以代理接口，也可以代理类，而 JDK 代理只能代理接口。同时，CGLIB 代理的性能要优于 JDK 代理。显然从性能的角度上来说 CGLIB 代理是正确的选择。使用 CGLIB 代理唯一要注意的是，它会为每一个不同的代理生成一个新类，在新版本 Spring（1.1 开始）中，Spring 能正确的重用生成的类，从而降低频繁的类生成带来的运行开销和 CGLIB 代理类所用的内存。当我们代理类时，CGLIB 代理时缺省选项，因为只有它能生成类的代理。如果要在代理接口时使用 CGLIB，我们就必须使用 setOptimize()（设定优化）方法将 ProxyFactory 类的优化标志设为 true。

### 9.3.6. Advanced Use of Pointcuts: 切入点的高级使用

### 9.3.6.1. Using Control Flow Pointcuts: 使用流程切入点

### 9.3.6.2. Using ComposablePointcut: 使用可组合切入点

### 9.3.6.3. Pointcutting Summary: 切入点总结

## 9.3.7. Getting Started with Introductions: 引入初步

### 9.3.7.1. Introduction Basics: 引入基础知识

### 9.3.7.2. Object Modification Detection with Introductions: 用 引入解决对象篡改检测问题

### 9.3.7.3. Introductions Summary: 引入小结

## 9.3.8. Framework Services for AOP: AOP 框架服务

### 9.3.8.1. Configuring AOP Declaratively: 声明性的设置 AOP

### 9.3.8.2. Using Automatic Proxying: 使用自动代理

## 9.3.9. 小结

我们通过这个章节介绍了 AOP 的核心概念，并学习了这些概念如何转变为 Spring AOP 实现。我们学习了 Spring AOP 中已实现及未实现的功能，同时指出如果要使用 Spring 中没有实现的功能应该考虑 AspectJ AOP 方案。我们用了很大的篇幅解释 Spring 提供的不同通知类型的细节，然后我们展示了其中四种的例子。我们也查看了如何用切入点来限制哪些方法会接到通知。特别的，我们讨论了 Spring 中五种切入点实现。然后我们详细学习了 AOP 代理是如何构造的，有哪几种不同的选择，以及他们都有什么区别。最后我们比较了三种不同类型代理的性能，并得出了 CGLIB 代理性能是最好的结论。所以，在大部分情况下，甚至所有情况下，CGLIB 代理都是正确的选择。

我们还看到了联结点的高级选项，以及如何引入扩展某个对象实现的接口集合。为了避免将创建 AOP 的逻辑直接写进代码中我们讲解了如何用 Spring 框架服务来声明性的配置 AOP。

## 9.4. 基于 Spring 的数据访问

**9.4.1. Exploring the JDBC Infrastructure:** 探索 JDBC 的底层结构

**9.4.2. Spring JDBC Infrastructure:** Spring JDBC 底层机制

**9.4.3. Database Connection and DataSources:** 数据库连接和数据源

**9.4.4. Using DataSources in DAO Classes:** 在 DAO 类中使用数据源

**9.4.5. Exception Handling:** 异常处理

**9.4.6. The JDBC Template Class:** JDBC 模板类

**9.4.7. Selecting The Data as Java Objects:** 选出作为 Java 对象的数据

**9.4.8. Updating Data:** 更新数据

**9.4.9. Inserting Data:** 插入数据

Getting the Primary Key Value: 获得主键

#### 9.4.10. Transactions: 事务

#### 9.4.11. Why JDBC: 为什么用 JDBC

#### 9.4.12. Using JDBC Data Access in the Sample

**Application:** 在示例程序中使用 JDBC 数据访问

#### 9.4.13. 小结

### 9.5. 在 Spring 应用中使用 Hibernate

#### 9.5.1. What Is Hibernate: 什么是 Hibernate

#### 9.5.1.1. Configuring Hibernate in a Spring Application: 在 Spring 应用中配置 Hibernate

#### 9.5.1.2. Mapping Files: 映射文件

简单映射

一对多映射

多对多映射

#### 9.5.2. The Hibernate Query Language: Hibernate 查询语言

#### 9.5.3. Selecting Data: 选择数据

##### 9.5.3.1. Simple Mappings: 简单映射

##### 9.5.3.2. 一对多和多对多选择

##### 9.5.3.3. 高级选择

#### 9.5.4. Updating and Inserting Data: 更新并插入数据

##### 9.5.4.1. Why Hibernate: 为什么选择 Hibernate

#### 9.5.5. Using Hibernate in the Sample Application: 在示例应用 中使用 Hibernate

#### 9.5.6. 小结

## 9.6. 基于 Spring 的应用程序设计及实现

### 9.6.1. Designing to Interfaces: 面向接口的设计

#### 9.6.1.1. Why Design to Interfaces: 为什么要面向接口

#### 9.6.1.2. The Factory Pattern: 工厂模式

### 9.6.2. Impact of Spring on Interface-Based Design: Spring

对于基于接口设计的冲击

### 9.6.3. Building a Domain Object Model: 域对象模型的构建

#### 9.6.3.1. Spring and the Domain Object Model: Spring 与域对象模型

#### 9.6.3.2. DOM!=Value Object: 域对象模型!=值对象

#### 9.6.3.3. Why Create a Domain Object Model: 为何要创建域对象模型

#### 9.6.3.4. Modeling Domain Objects: 域对象建模

#### 9.6.3.5. To Encapsulate Behavior or Not: 域对象中是否要封装行为

#### 9.6.3.6. The SpringBlog Domain Object Model: SpringBlog 域对象建模

#### 9.6.3.7. Canonicalization and Memory Considerations for a DOM: DOM 的标准化和内存的考量

#### 9.6.3.8. Domain Object Model Summary: 域对象模型综述

### 9.6.4. Designing and Building the Data Access Tier: 持久层设计与构建

#### 9.6.4.1. Why Have a Data Access Tier: 为何需要数据访问层

#### 9.6.4.2. Practical Design Considerations: 设计中需要注意的几点

#### 9.6.4.3. Canonicalization and Data Access: 标注化与数据访问

#### 9.6.4.4. Data Access Tier Summary: 数据访问层综述

### 9.6.5. Designing the Business Tier: 业务层设计

#### 9.6.5.1. Why Have a Business Tier: 为何需要业务层

#### 9.6.5.2. Designing Business Interfaces: 业务接口设计

#### 9.6.5.3. Service Object Granularity: 服务对象颗粒度

#### 9.6.5.4. Assembling Domain Object Relationships in The Business Tier: 在业务层组装域对象关联

#### 9.6.5.5. Business Method Granularity: 业务方法颗粒度

#### 9.6.5.6. Implementing Validation Logic: 实现数据验证逻辑

### 9.6.6. 小结

## 9.7. 基于 Spring 的 Web 应用开发

### 9.7.1. Introducing Spring MVC: 介绍 Spring MVC

#### 9.7.1.1. Introduction and Overview: 介绍和概览

#### 9.7.1.2. Implementation: 实现

### 9.7.2. Using Handler Mappings: 使用 Handler Mappings

### 9.7.3. Using Handler Interceptors: 使用处理拦截器

### 9.7.4. Working with Controllers: 让控制器工作起来

#### 9.7.4.1. AbstractController

#### 9.7.4.2. ParameterizableViewController

#### 9.7.4.3. MultiActionController

### 9.7.5. Views, Locales, and Themes: 视图, Locales, 主题

#### 9.7.5.1. Using Views Programmatically

#### 9.7.5.2. Using ViewResolver Implementations : 使用 ViewResolver 实现

#### 9.7.5.3. Using Localized Messages: 使用本地化消息

#### 9.7.5.4. Using Locales: 使用 Locales

#### 9.7.5.5. Using Themes: 使用主题

### 9.7.6. Using Command Controllers: 使用命令控制器

#### 9.7.6.1. Using Form Controllers: 使用表单控制器

#### 9.7.6.2. Exploring the AbstractWizardFormController: 探索 AbstractWizardFormController

#### 9.7.6.3. File Upload

### 9.7.7. Using Spring MVC in the Sample Application: 在范例应用中使用 Spring MVC

#### 9.7.8. 使用 Spring 中 JSP 标记

##### 9.7.8.1. Using the Message Tag: 使用 Message 标记

##### 9.7.8.2. Using the Theme Tag: 使用 Theme 标记

##### 9.7.8.3. Using the hasBindErrors Tag: 使用 hasBindErrors 标记

##### 9.7.8.4. Using the nestedPath Tag: 使用 nestedPath 标记

### 9.7.8.5. Using the bind Tag: 使用 bind 标记

### 9.7.8.6. Using the transform Tag: 使用 transform 标记

### 9.7.8.7. JSP Best Practices: JSP 最佳实践

## 9.7.9. Using Velocity: 使用 Velocity

### 9.7.9.1. Integrating Velocity and Spring: 集成 Velocity 和 Spring

### 9.7.9.2. Advanced Velocity Concepts: 高级 Velocity 主题

### 9.7.10. Using XSLT Views: 使用 XSLT 视图

### 9.7.11. Using PDF Views: 使用 PDF 视图

#### 9.7.11.1. Implementing a PDF View: 实现一个 PDF 视图

### 9.7.12. Using Excel Views: 使用 Excel 视图

### 9.7.13. Using Tiles: 使用 Tiles

#### 9.7.13.1. Integrating Tiles and Spring: 集成 Tiles 与 Spring

#### 9.7.13.2. Advanced Tiles Concepts: 高级 Tiles 主题

#### 9.7.13.3. Tiles Best Practices: Tiles 最佳实践

## 9.8. Spring 和 Struts

### 9.8.1. Exploring the Struts Architecture: 考察 Struts 体系结构

### 9.8.2. A Struts Application: 一个 Sturts 程序

### 9.8.3. Accessing Spring Beans: 访问 Spring Bean

### 9.8.4. Using Other Views: 使用第三方 Views

### 9.8.5. Using Struts Actions as Spring Beans: 将 Struts Action 定义为 Spring Bean

### 9.8.6. Combining Struts and Spring MVC: 整合 Struts 和 Spring MVC

### 9.8.7. 小结