

# Lightweight J2EE

轻量级

J2EE

## 企业应用实战

—Struts+Spring+Hibernate整合开发

李刚 著

- 全面介绍J2EE的流行框架
- 详细介绍时下全部架构模式
- 包含多达7个实体关联的实用案例

光盘包含：  
本书所有实例代码



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
[HTTP://WWW.PHEI.COM.CN](http://WWW.PHEI.COM.CN)



Java技术大系

# 轻量级 J2EE 企业应用实战

—Struts+Spring+Hibernate整合开发

李刚 著

电子工业出版社

Publishing House of Electronics Industry  
北京•BEIJING

## 内 容 简 介

本书所介绍的内容是作者多年 J2EE 开发经验的总结，内容涉及 Struts、Hibernate 和 Spring 三个开源框架，还介绍了 Tomcat 和 Jetty 两个开源 Web 服务器的详细用法，以及 J2EE 应用的几种常用架构。

本书不仅是一本 J2EE 入门图书，还详尽而细致地介绍了 JSP 各个方面，包括 JSP 2.0 的规范、Struts 的各种用法、Hibernate 的详细用法，以及 Spring 的基本用法。书中所介绍的轻量级 J2EE 应用，是目前最流行、最规范的 J2EE 架构，分层极为清晰，各层之间以松耦合的方法组织在一起。书的最后配备了两个实例，均采用了贫血模式的架构设计，以便于读者更快地进入 J2EE 应用开发。而第 8 章所介绍的其他架构模式则可作为读者对架构有更好把握后的提高部分。本书配套光盘包括各章内容所用的代码，以及整个应用所需要的开源类库等相关项目文件。

本书适用于有较好的 Java 编程基础，有初步的 J2EE 编程基础的读者。本书既可以作为 J2EE 初学者的入门书籍，也可作为 J2EE 应用开发者的提高指导。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

## 图书在版编目（CIP）数据

轻量级 J2EE 企业应用实战：Struts+Spring+Hibernate 整合开发 / 李刚著. —北京：电子工业出版社，2007.4  
(Java 技术大系)

ISBN 978-7-121-03998-0

I. 轻… II. 李… III. ①JAVA 语言—程序设计 ②软件工具—程序设计 IV. TP312 TP311.56

中国版本图书馆 CIP 数据核字（2007）第 033300 号

责任编辑：高洪霞 裴杰

印 刷：北京天宇星印刷厂

装 订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：34 字数：765 千字

印 次：2007 年 4 月第 1 次印刷

印 数：5000 册 定价：65.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：(010) 68279077；邮购电话：(010) 88254888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：(010) 88258888。

# 前　　言

目前，J2EE 应用确实很流行，从银行、证券系统，到企业信息化平台，甚至一些小公司，都争相使用 J2EE 应用。几年前，J2EE 应用是很“贵族”的产品，那时候使用 EJB 作为 J2EE 的核心，开发成本高，部署成本也高，开发者的学习曲线也陡峭。今天，轻量级 J2EE 应用的流行，让 J2EE 应用开始进入寻常百姓家。当然，轻量级 J2EE 应用是对经典 J2EE 应用的简化，在保留经典 J2EE 应用的架构、良好的可扩展性、可维护性的基础上，简化了 J2EE 应用的开发，降低了 J2EE 应用的部署成本。

轻量级 J2EE 应用让 J2EE 平台以更快的速度占领电子商务、电子政务等各种信息化平台市场。笔者从不否认对经典 J2EE 应用架构的喜爱，那种严谨的架构、全方位考虑的设计、优秀的分布式架构，无疑是一种编程的艺术。但它们太豪华了，以致于限制了它的市场占有率。可以这样说：经典 J2EE 应用是面向开发者的，而轻量级 J2EE 应用则面向用户。优秀的开发者会感慨并喜欢经典 J2EE 应用的设计，但市场则喜欢轻量级 J2EE 应用。轻量级 J2EE 应用模仿了经典 J2EE 应用的架构，保留了经典 J2EE 应用的各种优点，降低了学习难度和开发、部署成本，是一种更实际的信息化平台架构。

## 为什么写作本书

几年前，笔者主要从事实际的开发时，从未想过写一本书，忙是一个原因，更多原因是没有感触。如今，笔者已经在新东方 IT 培训中心担任 J2EE 培训讲师一年多，现已成为广州新东方软件工程师培训讲师的负责人。培训过程中，看到学生们求知若渴的眼睛，以及他们热切的需要：“老师，出一本关于某技术的书吧！”回想起 1999 年底时，笔者刚刚开发 J2EE 学习，完成一个简单的 EJB，居然花了将近一个月时间，其间苦痛只有程序员才懂。如今看到学生们的苦楚，想起更多 J2EE 学习者正备受煎熬，笔者愿意将多年的经验与大家一起分享，这些经验包含笔者多年废寝忘食后的恍然醒悟，也包含笔者跌落后艰难爬出的陷阱。希望这些经验能缩短读者朋友们的学习周期。

需要提醒读者朋友的是，所有的代码必须自己敲过，才是真正属于自己的代码。不要指望光看看本书，就可以成为一个编程高手，一定要踏踏实实地独立完成书中所有应用。学习编程是很辛苦却很有乐趣的事情，记得电影《阿甘正传》中有句话：“偶尔雨停了，可以看见星星。”这种场景很适合编程的境界，大部分时候都在埋头辛苦写代码，调试错误，只在当应用真正运行成功时，获得瞬间的快乐——这种快乐弥足珍贵，也是真正的快乐。

有时候我的学生会拿着他刚买的图书问我，这本书如何？很不幸，有时会发现名为 J2EE 的图书，居然在 JSP 页面中有 Hibernate 的 API。于是我无言以对，这样的图书到底想使读者成为怎样的开发者？阅读这样的图书不仅浪费时间，而且会造成错误的积累。

有感于此，笔者创作了本书，愿与各位读者共享多年的实践经验。即使今天，笔者依然与珠三角很多软件公司联系紧密，很多学生已在华为、立信、中企动力和京华网络等企业就业，有的学生已成长为技术经理，他们依然会就实际开发中的问题与笔者一起探讨，这些经验，都将出现在笔者的 J2EE 系列图书中。

## 本书有什么特点

与市面上已经存在的介绍 J2EE 应用的图书相比，本书有如下几个特色：

### 1. 经验丰富，针对性强

笔者既担任过软件开发的技术经理，也担任过软件公司的培训导师，也从事过职业培训的专职讲师。这些经验影响了笔者写书的目的，不是一本学院派的理论读物，而是一本实际的开发指南。

### 2. 内容实际，实用性强

本书所介绍的 J2EE 应用范例，规模可能并不大，但绝对是目前企业流行的开发架构，绝对严格遵守 J2EE 开发规范。而不是将各种技术杂乱地糅合在一起号称 J2EE。读者参考本书的架构，完全可以身临其境地感受企业实际开发。

### 3. 高屋建瓴，启发性强

本书介绍的几种架构模式，几乎是时下最全面的 J2EE 架构模式。这些架构模式可以直接提升读者对系统架构设计的把握。

## 本书写给谁看

本书适用于有较好的 Java 编程基础，有初步的 J2EE 编程基础的读者。本书既可以作为 J2EE 初学者的入门书籍，也可作为 J2EE 应用开发者的提高指导。

李 刚

2007-1-12

# 目 录

## CONTENTS

### 第1章 J2EE 应用运行及开发

环境的安装与配置 .....	1
1.1 JDK 的下载和安装 .....	2
1.1.1 Windows 下 JDK 的下载和安装 .....	2
1.1.2 Linux 下 JDK 的下载和安装 .....	5
1.2 Tomcat 的下载和安装 .....	6
1.2.1 Tomcat 的下载和安装 .....	7
1.2.2 Tomcat 的基本配置 .....	8
1.2.3 Tomcat 的数据源配置 .....	13
1.3 Jetty 的下载和安装 .....	17
1.3.1 Jetty 的下载和安装 .....	17
1.3.2 Jetty 的基本配置 .....	18
1.4 Eclipse 的安装和使用 .....	25
1.4.1 Eclipse 的下载和安装 .....	25
1.4.2 Eclipse 插件的安装 .....	26
1.4.3 Eclipse 的简单使用 .....	28
本章小结 .....	31

### 第2章 传统表现层 JSP

2.1 JSP 的技术原理 .....	33
2.2 JSP 注释 .....	36
2.3 JSP 声明 .....	37
2.4 JSP 表达式 .....	38
2.5 JSP 脚本 .....	38
2.6 JSP 的三个编译指令 .....	41
2.6.1 page 指令 .....	41
2.6.2 include 指令 .....	44
2.7 JSP 的 7 个动作指令 .....	45
2.7.1 forward 指令 .....	46
2.7.2 include 指令 .....	48
2.7.3 useBean, setProperty, getProperty 指令 .....	49
2.7.4 plugin 指令 .....	52
2.7.5 param 指令 .....	53

2.8 JSP 的 9 个内置对象 .....	54
2.8.1 application 对象 .....	55
2.8.2 config 对象 .....	58
2.8.3 exception 对象 .....	59
2.8.4 out 对象 .....	60
2.8.5 pageContext 对象 .....	61
2.8.6 request 对象 .....	62
2.8.7 response 对象 .....	67
2.8.8 session 对象 .....	70
2.9 Servlet 介绍 .....	72
2.9.1 Servlet 的开发 .....	72
2.9.2 Servlet 的配置 .....	74
2.9.3 Servlet 的生命周期 .....	75
2.9.4 使用 Servlet 作为控制器 .....	76
2.9.5 load-on-startup Servlet .....	80
2.9.6 访问 Servlet 的配置参数 .....	81
2.10 自定义标签库 .....	83
2.10.1 开发自定义标签类 .....	83
2.10.2 建立 TLD 文件 .....	84
2.10.3 在 web.xml 文件中增加 标签库定义 .....	84
2.10.4 使用标签库 .....	85
2.10.5 带属性的标签 .....	86
2.10.6 带标签体的标签 .....	90
2.11 Filter 介绍 .....	94
2.11.1 创建 Filter 类 .....	94
2.11.2 配置 Filter .....	95
2.12 Listener 介绍 .....	96
2.12.1 创建 Listener 类 .....	96
2.12.2 配置 Listener .....	98
2.13 JSP 2.0 的新特性 .....	98
2.13.1 JSP 定义 .....	99
2.13.2 表达式语言 .....	101

2.13.3 简化的自定义标签 .....	108
2.13.4 Tag File 支持 .....	111
本章小结 .....	113
<b>第3章 经典MVC框架Struts</b> .....	114
<b>3.1 MVC简介</b> .....	115
3.1.1 传统的Model 1和Model 2.....	115
3.1.2 MVC及其优势 .....	116
<b>3.2 Struts概述</b> .....	117
3.2.1 Struts的起源.....	117
3.2.2 Struts的体系结构.....	117
<b>3.3 Struts的下载和安装</b> .....	118
<b>3.4 Struts入门</b> .....	119
3.4.1 基本的MVC示例 .....	119
3.4.2 Struts的基本示例.....	126
3.4.3 Struts的流程.....	129
<b>3.5 Struts的配置</b> .....	130
3.5.1 配置ActionServlet.....	130
3.5.2 配置ActionForm.....	132
3.5.3 配置Action .....	133
3.5.4 配置Forward.....	134
<b>3.6 Struts程序的国际化</b> .....	135
3.6.1 Java程序的国际化 .....	136
3.6.2 Struts的国际化 .....	139
<b>3.7 使用动态ActionForm</b> .....	143
3.7.1 配置动态ActionForm .....	143
3.7.2 使用动态ActionForm .....	144
<b>3.8 Struts的标签库</b> .....	145
3.8.1 使用Struts标签的基本配置 .....	145
3.8.2 使用html标签库 .....	146
3.8.3 使用bean标签库 .....	148
3.8.4 使用logic标签库 .....	155
<b>3.9 Struts的数据校验</b> .....	164
3.9.1 ActionForm的代码校验 .....	165
3.9.2 Action的代码校验 .....	169
3.9.3 结合commons-validator.jar 的校验 .....	169
<b>3.10 Struts的异常框架</b> .....	177
<b>3.11 几种常用的Action</b> .....	180
3.11.1 DispatchAction及其子类 .....	181
3.11.2 使用ForwardAction .....	185
3.11.3 使用IncludeAction .....	185
3.11.4 使用SwitchAction .....	186
<b>3.12 Struts的常见扩展方法</b> .....	187
3.12.1 实现PlugIn接口 .....	187
3.12.2 继承RequestProcessor .....	188
3.12.3 继承ActionServlet .....	190
本章小结 .....	191
<b>第4章 使用Hibernate</b>	
<b>完成持久化</b> .....	192
<b>4.1 ORM简介</b> .....	193
4.1.1 什么是ORM .....	193
4.1.2 为什么需要ORM .....	193
4.1.3 流行的ORM框架介绍 .....	193
<b>4.2 Hibernate概述</b> .....	194
4.2.1 Hibernate的起源 .....	194
4.2.2 Hibernate与其他ORM 框架的对比 .....	195
<b>4.3 Hibernate的安装和使用</b> .....	195
4.3.1 Hibernate下载和安装 .....	195
4.3.2 传统JDBC的数据库操作 .....	196
4.3.3 Hibernate的数据库操作 .....	197
<b>4.4 Hibernate的基本映射</b> .....	200
4.4.1 映射文件结构 .....	200
4.4.2 主键生成器 .....	200
4.4.3 映射集合属性 .....	201
4.4.4 映射引用属性 .....	208
<b>4.5 Hibernate的关系映射</b> .....	216
4.5.1 单向N-1的关系映射 .....	217
4.5.2 单向1-1的关系映射 .....	220
4.5.3 单向1-N的关系映射 .....	222
4.5.4 单向N-N的关系映射 .....	225
4.5.5 双向1-N的关系映射 .....	226
4.5.6 双向N-N关联 .....	230
4.5.7 双向1-1关联 .....	232

4.6 Hibernate 查询体系 .....	237	5.7.1 了解 bean 的生命周期 .....	309
4.6.1 HQL 查询 .....	237	5.7.2 定制 bean 的生命周期行为 .....	309
4.6.2 条件查询 .....	247	5.7.3 协调不同步的 bean .....	313
4.6.3 SQL 查询 .....	249	5.8 bean 的继承 .....	315
4.6.4 数据过滤 .....	253	5.8.1 使用 abstract 属性 .....	315
4.7 事件框架 .....	255	5.8.2 定义子 bean .....	317
4.7.1 拦截器 .....	256	5.8.3 Spring bean 的继承与 Java 中继承的区别 .....	318
4.7.2 事件系统 .....	259	5.9 bean 后处理器 .....	319
本章小结 .....	263	5.10 容器后处理器 .....	322
<b>第 5 章 Spring 介绍 .....</b>	<b>264</b>	5.10.1 属性占位符配置器 .....	323
5.1 Spring 的起源和背景 .....	265	5.10.2 另一种属性占位符配置器 (PropertyOverrideConfigurer) .....	324
5.2 Spring 的下载和安装 .....	265	5.11 与容器交互 .....	325
5.3 Spring 实现两种设计模式 .....	266	5.11.1 工厂 bean 简介与配置 .....	325
5.3.1 单态模式的回顾 .....	266	5.11.2 FactoryBean 接口 .....	327
5.3.2 工厂模式的回顾 .....	268	5.11.3 实现 BeanFactoryAware 接口获取 BeanFactory .....	329
5.3.3 Spring 对单态与工厂 模式的实现 .....	270	5.11.4 使用 BeanNameAware 回调本身 .....	330
5.4 Spring 的依赖注入 .....	271	5.12 ApplicationContext 介绍 .....	331
5.4.1 理解依赖注入 .....	272	5.12.1 国际化支持 .....	332
5.4.2 设值注入 .....	273	5.12.2 事件处理 .....	334
5.4.3 构造注入 .....	276	5.12.3 Web 应用中自动加载 ApplicationContext .....	335
5.4.4 两种注入方式的对比 .....	277	5.13 加载多个 XML 配置文件 .....	337
5.5 bean 和 BeanFactory .....	278	5.13.1 ApplicationContext 加载多个配置文件 .....	337
5.5.1 Spring 容器 .....	278	5.13.2 Web 应用启动时加载 多个配置文件 .....	337
5.5.2 bean 的基本定义 .....	280	5.13.3 XML 配置文件中导入 其他配置文件 .....	338
5.5.3 定义 bean 的行为方式 .....	281	本章小结 .....	338
5.5.4 深入理解 bean .....	282	<b>第 6 章 Spring 与 Hibernate 的整合 ....</b>	<b>339</b>
5.5.5 创建 bean 实例 .....	284	6.1 Spring 对 Hibernate 的支持 .....	340
5.6 依赖关系配置 .....	291	6.2 管理 SessionFactory .....	340
5.6.1 配置依赖 .....	292	6.3 Spring 对 Hibernate 的简化 .....	342
5.6.2 注入属性值 .....	297		
5.6.3 注入 field 值 .....	300		
5.6.4 注入方法返回值 .....	301		
5.6.5 强制初始化 bean .....	304		
5.6.6 自动装配 .....	304		
5.6.7 依赖检查 .....	307		
5.7 bean 的生命周期 .....	309		

<b>6.4 使用 HibernateTemplate</b>	343	8.1.3 稳定性、高效性	392
6.4.1 HibernateTemplate 的常规用法	346	8.1.4 花费最小化，利益最大化	393
6.4.2 Hibernate 的复杂用法	347	<b>8.2 如何面对挑战</b>	393
<b>6.5 Hibernate 的 DAO 实现</b>	349	8.2.1 使用建模工具	393
6.5.1 DAO 模式简介	349	8.2.2 利用优秀的框架	394
6.5.2 继承 HibernateDaoSupport 实现 DAO	350	8.2.3 选择性地扩展	396
6.5.3 基于 Hibernate 3.0 实现 DAO	353	8.2.4 使用代码生成器	396
<b>6.6 事务管理</b>	354	<b>8.3 常用的设计模式及应用</b>	397
6.6.1 编程式的事务管理	355	8.3.1 单态模式的使用	397
6.6.2 声明式事务管理	357	8.3.2 代理模式的使用	400
6.6.3 事务策略的思考	366	8.3.3 Spring AOP 介绍	403
本章小结	366	<b>8.4 常见的架构设计策略</b>	408
<b>第 7 章 Spring 与 Struts 的整合</b>	367	8.4.1 贫血模式	408
7.1 Spring 整合第三方 MVC 框架的通用配置	368	8.4.2 Rich Domain Object 模式	413
7.1.1 采用 ContextLoaderListener 创建 ApplicationContext	368	8.4.3 抛弃业务逻辑层	418
7.1.2 采用 load-on-startup Servlet 创建 ApplicationContext	370	本章小结	419
7.2 Spring 与 MVC 框架整合的思考	372	<b>第 9 章 完整实例：消息发布系统</b>	420
7.3 利用 Spring 的 IoC 特性整合	374	9.1 系统架构说明	421
7.3.1 使用 DelegatingRequest Processor	375	9.1.1 系统架构说明	421
7.3.2 使用 DelegatingActionProxy	380	9.1.2 采用架构的优势	421
7.4 使用 ActionSupport 替代 Action	382	9.2 Hibernate 持久层	422
7.5 实用的整合策略	385	9.2.1 编写 PO 类	423
本章小结	388	9.2.2 编写 PO 的映射配置文件	428
<b>第 8 章 企业应用开发的思考与策略</b>	389	9.2.3 连接数据库	431
8.1 企业应用开发面临的挑战	390	9.3 DAO 组件层	434
8.1.1 可扩展性、可伸缩性	390	9.3.1 DAO 组件的结构	434
8.1.2 快捷、可控的开发	392	9.3.2 编写 DAO 接口	435

9.5.3 数据校验的选择 .....	456	10.2.3 映射持久化类 .....	480
9.5.4 访问权限的控制 .....	459	10.3 实现 DAO 层 .....	485
9.5.5 解决中文编码问题 .....	460	10.3.1 DAO 组件的定义 .....	486
9.5.6 JSP 页面输出 .....	462	10.3.2 实现 DAO 组件 .....	492
9.6 系统最后的思考 .....	464	10.3.3 部署 DAO 层 .....	502
9.6.1 传统 EJB 架构的实现 .....	464	10.4 实现 Service 层 .....	505
9.6.2 EJB 架构与轻量级 架构的对比 .....	466	10.4.1 Service 组件设计 .....	505
本章小结 .....	468	10.4.2 Service 组件的实现 .....	506
<b>第 10 章 完整应用：简单</b>		10.5 任务调度的实现 .....	516
<b>工作流系统 .....</b>	<b>469</b>	10.5.1 Quartz 的使用 .....	516
10.1 项目背景及系统结构 .....	470	10.5.2 在 Spring 中使用 Quartz .....	520
10.1.1 应用背景 .....	470	10.6 MVC 层实现 .....	522
10.1.2 系统功能介绍 .....	470	10.6.1 解决中文编码 .....	522
10.1.3 相关技术介绍 .....	471	10.6.2 Struts 与 Spring 的整合 .....	523
10.1.4 系统结构 .....	472	10.6.3 创建 Action .....	524
10.1.5 系统的功能模块 .....	473	10.6.4 异常处理 .....	524
10.2 Hibernate 持久层 .....	473	10.6.5 权限控制 .....	525
10.2.1 设计持久化对象（PO） .....	473	10.6.6 控制器配置 .....	527
10.2.2 创建持久化类 .....	474	本章小结 .....	530



# 第1章

## J2EE 应用运行及开发环境的安装与配置

### 本章要点

- 『 JDK 的下载和安装
- 『 环境变量的设置
- 『 Tomcat 的安装和配置
- 『 在 Tomcat 中部署 Web 应用
- 『 Jetty 的安装和配置
- 『 在 Jetty 中部署 Web 应用
- 『 Eclipse 的安装
- 『 Eclipse 插件的安装

J2EE 应用以其稳定的性能、良好的开放性及严格的安全性，深受企业应用开发者的青睐。实际上，对于信息化要求较高的行业，如银行、电信、证券及电子商务等行业，都会选择使用 J2EE 作为企业的信息平台。

对于一个企业而言，选择 J2EE 构建信息化平台，更体现了一种长远的规划：企业的信息化是不断整合的过程，在未来的日子里，经常会有不同平台、不同系统的异构系统需要整合。J2EE 应用提供的跨平台性、开放性及各种远程访问的技术，为异构系统的良好整合提供了保证。

本书介绍的不是基于 EJB 的 J2EE 应用的开发，因为 EJB 应用的开发周期过长，且必须运行在 J2EE 容器中。而本书介绍的轻量级 J2EE 应用，完全可以运行在 Web 容器中，无需 EJB 容器的支持，但其应用的稳定性及效果都可以得到保证。

## 1.1 JDK 的下载和安装

J2EE 应用的开发及运行都离不开 JDK 的支持。虽然 Java 程序是跨平台的，但 JDK 不是跨平台的。因此，在不同的平台上需要安装不同的 JDK。下面分别介绍在 Windows 和 Linux 下 JDK 的安装。

### 1.1.1 Windows 下 JDK 的下载和安装

目前，JDK 主要有如下三个版本。

**J2SE：**Java 标准版本，包括开发桌面应用的系列类库。

**J2EE：**包含 Java 标准版，还增加了企业应用开发所需的类库。

**J2ME：**Java 2 平台微型版，被使用在资源受限制的小型消费型电子设备上。

本书介绍的是 J2EE 应用的开发，推荐使用 J2EE 的 JDK。下载和安装请按如下步骤进行。

(1) 登录 <http://www.sun.com> 站点，根据所使用的操作系统，选择 J2EE 的最新版本，笔者使用的是 j2eesdk-1\_4\_02\_2005Q2，推荐读者也使用该版本的 J2EE。

下载完成后，得到一个名为 j2eesdk-1\_4\_02\_2005Q2-windows-m1.exe 的可执行文件，该文件就是 J2EE 的安装文件。

(2) 双击下载的可执行性文件，出现如图 1.1 所示的安装向导对话框，表明 JDK 开始安装。

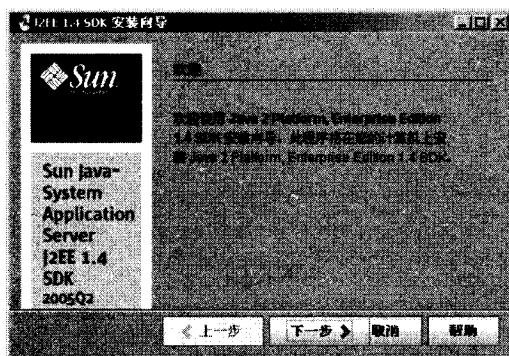


图 1.1 J2EE JDK 安装界面

(3) 安装过程与安装其他 Windows 软件并没有太大的不同，同样是多次单击【下一步】按钮，需要选择安装路径等步骤。笔者建议不要修改 JDK 的安装路径，若要修改也仅仅修改其盘符，程序出现如图 1.2 所示的安装路径选择对话框。

笔者将安装路径选择在 D 盘（因为笔者将所有的工具软件都放在 D 盘），而建议不要修改后面的 Sun\AppSever 路径。直到出现如图 1.3 所示的对话框。

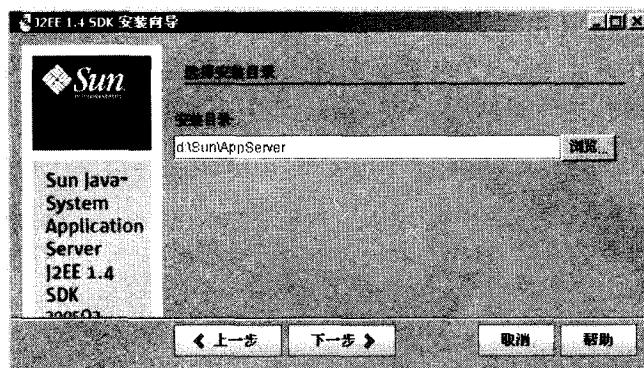


图 1.2 安装路径选择界面

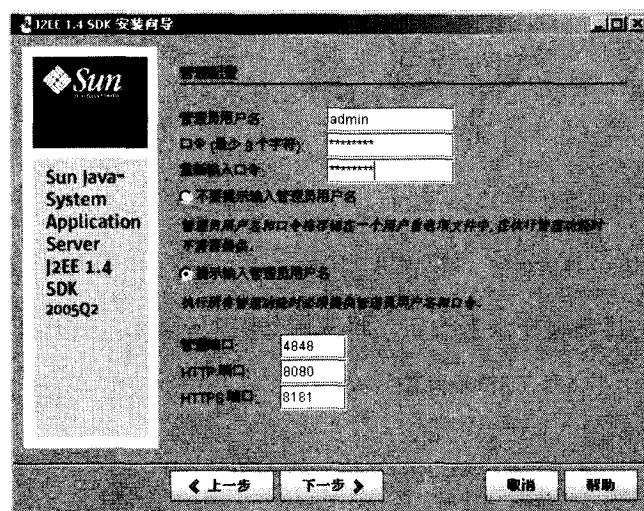


图 1.3 管理员口令窗口

(4) 选择“提示输入管理员用户名”单选框，然后在口令和重新输入口令的密码框中输入两个系统的口令，口令的长度最少必须为 8 个字符。然后单击【下一步】按钮，在下一个对话框中再次单击【下一步】按钮，程序开始安装。

(5) 安装成功后，增加编译和运行必需的环境变量。编译和运行 Java 程序必须增加 CLASSPATH 环境变量。在编译和运行 Java 程序时，需要 JDK 的系统类，如 java.lang.String 等，Java 程序会根据 CLASSPATH 环境变量指定的路径搜索这些类。因此该环境变量就是系列的搜索路径，编译和运行 J2EE 的应用主要需要如下三个 jar 文件：

- Sun\AppServer\jdk\lib\tools.jar
- Sun\AppServer\jdk\lib\dt.jar
- Sun\AppServer\lib\j2ee.jar

这些 jar 文件表面看起来是一个文件，其实是一系列的路径，选择 WinRAR 文件打开其中任意一个文件，看到如图 1.4 所示的窗口。

接着，在系统中增加 CLASSPATH 的环境变量，并将这三个文件的路径添加进去，添加的方法如下：在桌面上“我的电脑”图标上单击右键，出现右键菜单，单击“属性”菜单项，出现“系统特性”对话框，单击该对话框的“高级”选项卡，出现如图 1.5 所示的对话框。



图 1.4 j2ee.jar 文件的内部结构

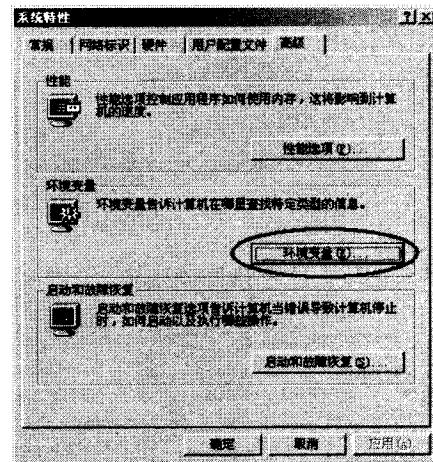


图 1.5 【系统特性】对话框

单击图 1.5 中的【环境变量】按钮，将弹出如图 1.6 所示的【环境变量】对话框。

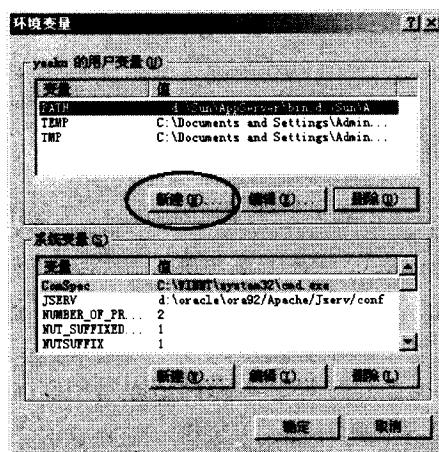


图 1.6 环境变量设置对话框

单击图 1.6 中的【新建】按钮，出现如图 1.7 所示的对话框，读者可以看到增加的 CLASSPATH 环境变量已将 dt.jar, tools.jar, j2ee.jar 三个文件设置到该环境变量中。因为 Windows 多个环境变量中的间隔符号是“;”，所以多个路径之间以英文分号隔开。增加后效果如图 1.7 所示。

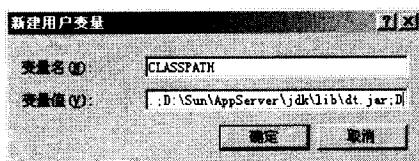


图 1.7 增加 CLASSPATH 环境变量后的效果

用户在编译和运行 Java 程序时，需要用到 java 和 javac 两个命令。由于 Windows 对于外部命令，都按 PATH 环境变量指定的路径搜索可执行性程序，因此为了可以执行 java 和 javac 等命令，应将 java 和 javac 所在的路径添加到 PATH 中。java 和 javac 的路径为 D:\Sun\AppServer\jdk\bin，通常系统已经有了 PATH 环境变量，因此只需将该路径添加到 PATH 变量中即可。

按上面的步骤修改 PATH 环境变量，修改后的效果如图 1.8 所示。

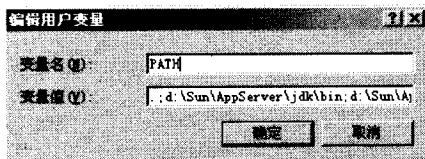


图 1.8 PATH 环境变量的设置

注意：在两个环境变量的设置中，都包含了一个“.”的路径，这个“.”代表系统的当前路径，如果没有增加该路径，可能导致运行 Java 程序时，class 文件已在当前路径，但在系统提供的文件中找不到该文件。

### 1.1.2 Linux 下 JDK 的下载和安装

Linux 下 JDK 的下载和安装与 Windows 下并没有太大的不同，只是对一些环境的设置稍有不同。下载和安装 Linux 下的 JDK 请按如下步骤进行。

(1) 登录 <http://www.sun.com> 站点，同样选择 JDK 的 j2eesdk-1\_4\_02\_2005Q2 的版本，区别只是在下载 Linux 版本成功后，得到的是 j2eesdk-1\_4\_02\_2005Q2-linux-ml.bin 文件。

(2) 进入 j2eesdk-1\_4\_02\_2005Q2-linux-ml所在的路径，执行如下命令：

```
chmod 777 j2eesdk-1_4_02_2005Q2-linux-ml.bin
```

该命令修改 j2eesdk-1\_4\_02\_2005Q2-linux-ml.bin 文件为可执行文件，然后输入如下命令：

```
./ j2eesdk-1_4_02_2005Q2-linux-ml.bin
```

按回车键后，出现与图 1.1 类似的【安装向导】对话框。

(3) 等待安装结束后，设置环境变量。环境变量的设置与在 Windows 下完全相同，

只是设置的方式稍有区别（以 RedHat Linux Fedora Core 4 为例）。

(4) 登录 Linux 系统后，打开 shell 窗口并输入如下命令：

```
cd
```

(5) 该命令将进入当前用户的 home 路径，然后在 home 路径下输入如下命令：

```
ls -a
```

(6) 该命令将列出当前路径下所有的文件（包括隐藏文件），环境变量的设置是通过.bash\_profile 文件设置的。

(7) 使用无格式编辑器编辑该文件来增加 CLASSPATH 环境变量，并修改 PATH 环境变量，修改后的.bash\_profile 文件如下（文件中以#开头的行是注释）：

```
# Get the aliases and functions
# 如果.bashrc 文件存在，执行该文件的内容
# 因此，也可以在.bashrc 文件中设置变量
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
# User specific environment and startup programs
#下面用于设置环境变量
#JAVA_HOME 环境变量是 Tomcat 运行需要的环境变量
JAVA_HOME=/home/yeku/sun/jdk
#其中$JAVA_HOME，表示引用宏变量，第二条路径实际就是
# /home/yeku/sun/jdk/bin
PATH=.:$PATH:$HOME/bin:$JAVA_HOME/bin
# 设置 CLASSPATH 环境变量
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
# 导出环境变量
export ANT_HOME
export JAVA_HOME
export CLASSPATH
export PATH
unset USERNAME
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
```

(8) 在上面中的环境变量中，路径之间以“：“分隔，因为 Linux 以“：“作为路径分隔符。

(9) 重新登录，或者执行如下命令：

```
source .bash_profile
```

两种方式都是为了运行该文件，使文件中关于环境变量的设置起作用。

## 1.2 Tomcat 的下载和安装

Tomcat 是 Java 领域最著名的开源 Web 容器，简单、易用且稳定性极好。既可以作为个人学习之用，也可以作为商业产品发布。

Tomcat 不仅提供了 Web 容器的基本功能，还支持 JAAS 和 JNDI 绑定等。

## 1.2.1 Tomcat 的下载和安装

因为 Tomcat 完全以 Java 编写，因此与平台无关，既可以运行在 Windows 平台上，也可以运行在 Linux 平台上。两个平台上的安装和配置也基本相同，只是环境变量的设置稍有差别而已，关于 Linux 下环境变量的设置请参考 1.1.2 节。本节将以 Windows 平台为例。

下载和安装 Tomcat 按如下步骤进行。

(1) 登录 <http://tomcat.apache.org> 站点，下载 Tomcat 合适的版本，如果使用 JDK1.4，则建议使用 Tomcat 5.0.x 系列，而不是使用 Tomcat 5.5.x 系列。

Tomcat 5.0.x 目前最新的稳定版本是 5.0.28，建议下载该版本。在 Windows 平台下载 zip 包，Linux 下载 tar 包。建议不要下载其安装文件。

(2) 解压缩刚下载到的压缩包，解压缩后应有如下文件结构。

- bin: 存放启动和关闭 Tomcat 的命令的路径。
- common: 存放所有的 Web 应用都需要的类库等。
- conf: 存放 Tomcat 的配置，所有的 Tomcat 的配置都在该路径下设置。
- log: 这是一个空路径，该路径用于保存 Tomcat 每次运行后产生的日志。
- server: 存放 Tomcat 运行所需要的基础类库，该路径是 Tomcat 运行的基础。该路径下还包含一个 webapps 路径，并存放 Tomcat 两个控制台。
- shared: 该路径也是一个空路径，用于系统共享的类库，该路径下包括 classes 和 lib 两个路径，其中 classes 用于存放 class 文件，而 lib 用于存放 Jar 文件。
- temp: 保存 Web 应用运行过程中生成的临时文件。
- webapps: 该路径用于部署 Web 应用，将 Web 应用复制在该路径下，Tomcat 会将该应用自动部署在容器中。
- work: 保存 Web 应用运行过程中编译生成的 class 文件。该文件夹可以删除，但每次应用启动时将自动建立该路径。
- LICENSE 等相关文档。

(3) 将解压缩后的文件夹放在到任意路径下。

(4) Tomcat 的运行需要一个环境变量：JAVA\_HOME。不管是 Windows 还是 Linux，只需要增加该环境变量即可，该环境变量的值指向 JDK 安装路径。

(5) 启动 Tomcat，对于 Windows 平台，只需要双击 Tomcat 安装路径下 bin 路径中的 startup.bat 文件即可。对于 Linux，进入 Tomcat 安装路径的 bin 路径下，然后运行如下命令：

```
chmod 777 startup.sh
```

该命令将 startup.sh 文件变成可执行性文件，接着用如下命令运行 startup.sh 即可：

```
./startup.sh
```

启动 Tomcat 之后，打开浏览器，在地址栏输入 <http://localhost:8080>，然后回车，浏

览器出现如图 1.9 所示界面，即表示 Tomcat 安装成功。

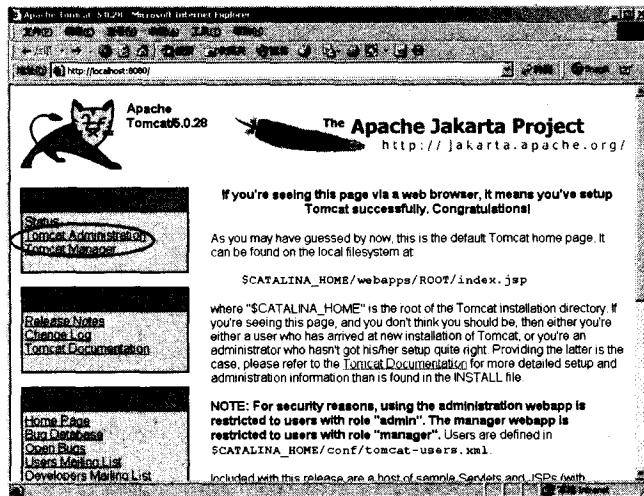


图 1.9 Tomcat 安装成功的界面

## 1.2.2 Tomcat 的基本配置

Tomcat 作为一个 Web 服务器，默认的服务端口是 8080，但该端口完全可以自己控制。虽然 Tomcat 是免费的 Web 服务器，但也提供了两个图形界面的控制台。用户可以使用控制台方便地部署 Web 应用、配置数据源及监控服务器中的 Web 应用等。

下面介绍如何修改 Tomcat 的 Web 服务端口，并进入其控制台来部署 Web 应用。

### 1. 修改端口

Tomcat 的配置文件都放在 conf 路径下，控制端口的配置文件也放在该路径下。打开 conf 下的 server.xml 文件，务必使用记事本或 vi 等无格式的编辑器，不要使用如写字板等有格式的编辑器。在定位 server.xml 文件的 92 行处看到如下代码：

```
<Connector port="8080"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    debug="0" connectionTimeout="20000"
    disableUploadTimeout="true" />
```

其中 port="8080"，就是 Tomcat 提供 Web 服务的端口。将 8080 修改成任意的端口，建议使用 1000 以上的端口，避免与公用端口冲突。笔者将此处修改为 8888，即 Tomcat 的 Web 服务的提供端口为 8888。

修改成功后，重新启动 Tomcat，在浏览器中输入 <http://localhost:8888>，回车将再次看到如图 1.9 所示的界面，即显示 Tomcat 安装成功的界面。

### 2. 进入控制台

在图 1.9 中的红色标识处，显示有两个控制台：一个是 Administration 控制台；另一

个是 Manager 控制台。红色标记的是 Administration 控制台，下面是 Manager 控制台，单击 Administration 控制台将出现如图 1.10 所示的登录界面。

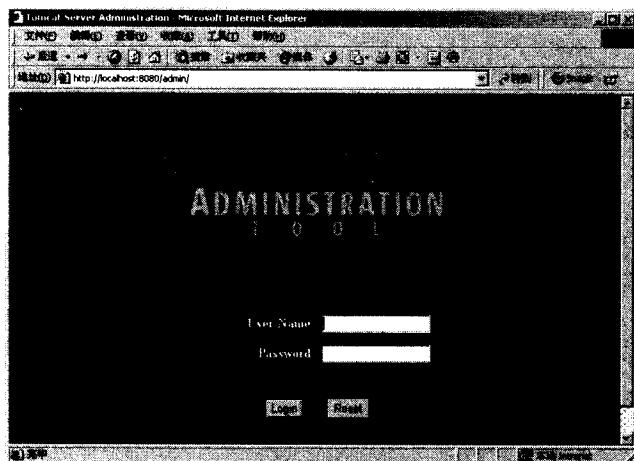


图 1.10 Tomcat Administration 控制台登录界面

单击 Manager 控制台，将出现如图 1.11 所示的登录界面。

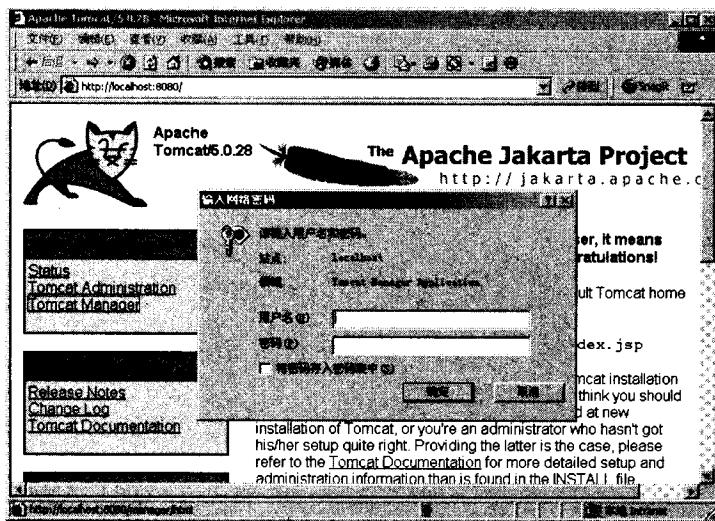


图 1.11 Tomcat Manager 控制台登录界面

两个控制台都需要增加用户名和密码才可以登录，这两个控制台都是通过 Tomcat 的 JAAS 控制的，下面介绍如何增加这两个控制台的用户和密码。

回忆前面关于 Tomcat 文件结构的介绍，server/webapps 下存放这两个控制台的 Web 应用，先进入 server/webapps/manager 路径下，该文件夹对应了 Manager 控制台。再进入该 Web 应用的 WEB-INF 路径下，该路径存放了 Web 应用的配置文件，用无格式编辑

器打开 web.xml 文件。在该文件的最后部分，看到如下配置片段：

```
<!-- Define the Login Configuration for this Application -->
<!-- 确定 JAAS 的登录方式-->
<login-config>
    <!-- BASIC 表明使用弹出式窗口登录-->
    <auth-method>BASIC</auth-method>
    <realm-name>Tomcat Manager Application</realm-name>
</login-config>
<!-- Security roles referenced by this web application -->
<!-- 确定登录该应用所需的安全角色-->
<security-role>
    <description>
        The role that is required to log in to the Manager Application
    </description>
    <!-- 只有 manager 角色才可以登录该应用-->
    <role-name>manager</role-name>
</security-role>
```

通过查看该文件可知，Manager 控制台需要 manager 角色才可以登录。

同样，在 server/webapps/admin/WEB-INF 路径下，打开 web.xml 文件，在该文件的最后部分，发现如下代码：

```
<!-- Login configuration uses form-based authentication -->
<!-- 确定 JAAS 的登录方式-->
<login-config>
    <!-- FORM 表明使用表单提交的方式登录-->
    <auth-method>FORM</auth-method>
    <realm-name>Tomcat Server Configuration Form-Based Authentication
Area</realm-name>
    <form-login-config>
        <!-- 确定登录的 form 页面-->
        <form-login-page>/login.jsp</form-login-page>
        <!-- 确定登录出错的提示页面-->
        <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
</login-config>
<!-- 确定登录该应用所需的安全角色-->
<!-- Security roles referenced by this web application -->
<security-role>
    <description>
        The role that is required to log in to the Administration Application
    </description>
    <!-- 只有 admin 角色才可以登录该应用-->
    <role-name>admin</role-name>
</security-role>
```

通过查看该文件可知，Administration 控制台需要 admin 角色才可以登录。下面将增加 Tomcat 的用户，Tomcat 的用户通过 conf 路径下的 tomcat-users.xml 文件控制，打开该文件，发现有如下代码：

```
<!-- 配置 Tomcat 用户和角色-->
<tomcat-users>
    <!-- 配置第一个用户，名为 tomcat，密码为 tomcat，角色为 tomcat-->
    <user name="tomcat" password="tomcat" roles="tomcat" />
```

```

<!-- 配置第二个用户，名为 role1，密码为 tomcat，角色为 role1-->
<user name="role1" password="tomcat" roles="role1" />
<!-- 配置第三个用户，名为 both，密码为 tomcat，角色为 tomcat 和 role1-->
<user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>

```

系统的配置文件默认提供了三个用户，但这三个用户没有一个是 admin 角色，也没有一个属于 manager 角色。在这里增加 manager 和 admin 两个用户，增加后的配置文件代码如下：

```

<?xml version='1.0' encoding='utf-8'?>
<!-- 配置 Tomcat 用户和角色-->
<tomcat-users>
    <!-- 配置 Tomcat 角色-->
    <role rolename="tomcat"/>
    <role rolename="role1"/>
    <role rolename="manager"/>
    <role rolename="admin"/>
    <!-- 配置 Tomcat 用户-->
    <user username="tomcat" password="tomcat" roles="tomcat"/>
    <user username="both" password="tomcat" roles="tomcat,role1"/>
    <user username="role1" password="tomcat" roles="role1"/>
    <!-- 配置登录 Manager 控制台的用户 manager，密码也是 manager-->
    <user username="manager" password="manager" roles="manager"/>
    <!-- 配置登录 Administration 控制台的用户 admin，密码也是 admin-->
    <user username="admin" password="admin" roles="admin"/>
</tomcat-users>

```

再次重启 Tomcat，进入 Administration 控制台，在用户名和密码的文本框中输入 admin 和 admin（刚才在 tomcat-users.xml 文件中配置的），单击【登录】按钮，将进入 Administration 控制台，如图 1.12 所示界面。

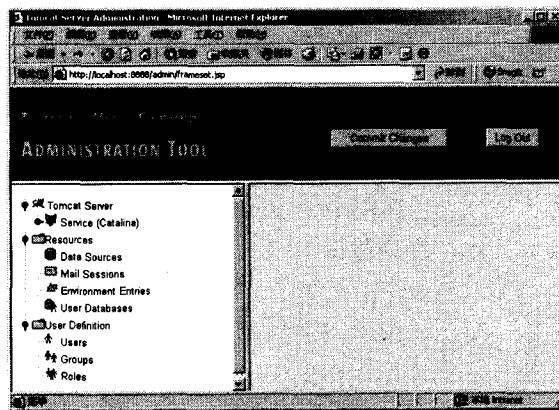


图 1.12 Tomcat 的 Administration 控制台

在该控制台的左边，可看到关于 DataSource、MailSessions、UserDifintion 等项，这表明可通过控制台配置数据源、邮件 Session 及 Tomcat 用户等。该控制台的使用相当简单，此处不再赘述。

现在也可以进入 Manager 控制台了，在弹出的登录对话框的用户名和密码框中分别

输入 manager 和 manager，即可登录 Manager 控制台，看到如图 1.13 所示的界面。

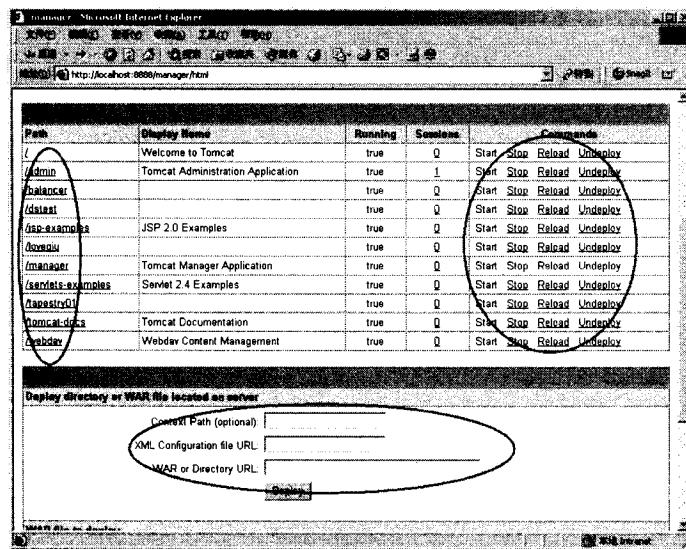


图 1.13 Tomcat 的 Manager 控制台

该控制台可监控部署在该服务器下的所有 Web 应用：左边的红色区标识出了所有部署在该 Web 容器内的全部应用；右边的红色区标识出了对 Web 应用的控制，包括启动、停止及重启等；而下面的红色标识区，则用于部署 Web 应用。

### 3. 部署 Web 应用

在 Tomcat 中部署 Web 应用的方式非常多，主要有如下方式：

- 使用控制台部署；
- 利用 Tomcat 的自动部署；
- 修改 server.xml 文件部署 Web 应用；
- 增加用户的 Web 部署文件。

通过控制台的部署方式实质上和修改 server.xml 文件的部署方式相同。所有在控制台的修改，最终都由服务器转变为修改 server.xml 文件。

笔者不推荐采用修改 server.xml 的配置方式。因为 server.xml 文件是一个系统文件，通常对于系统的文件应尽量避免修改，可通过增加自己的配置文件即可。

下面主要介绍用自动部署来增加用户的 Web 部署文件。

自动部署非常简单，只需将 Web 应用复制到 Tomcat 的 webapps 路径下，Tomcat 就会自动加载该 Web 应用。

增加用户的 Web 部署文件后，为了避免复制 Web 应用，只需简单地增加一个配置文件即可。进入 Tomcat 的 conf\Catalina\localhost 路径下，该路径下默认有三个配置文件，分别对应系统自带的三个 Web 应用（包括两个控制台应用）。

复制三个文件中的任意一个，假设复制了 balancer.xml 文件，并将该文件重命名，

重命名的文件名并不重要，但为了更好的可读性，建议使该文件的文件名与部署的 Web 应用同名。

打开该文件，发现如下内容：

```
<!-- 部署一个 Web 应用，其中 path 是 Web 应用虚拟路径
而 docBase 是 Web 应用的文档路径-->
<Context path="/balancer" docBase="balancer" debug="0" privileged="true">
</Context>
```

其中，每个 Context 元素都对应一个 Web 应用，该元素的 path 属性确定 Web 应用的虚拟路径，而 docBase 则是 Web 应用的文档路径。

假如 E:/webroot 是一个 Web 应用，若想将该应用部署在/test 虚拟路径下，只需将该文件的内容作如下修改：

```
<!-- 部署一个 Web 应用，其中 path 是 Web 应用虚拟路径
而 docBase 是 Web 应用的文档路径-->
<Context path="/test" docBase="e:/webroot" debug="0" privileged="true">
</Context>
```

### 1.2.3 Tomcat 的数据源配置

Tomcat 本身并不具备提供数据源的能力，而是借助于其他一些开源数据源（如 DBCP 和 C3P0 等）来实现。通过 Tomcat 提供的数据源，程序可以通过 JNDI 获得数据源，提供更方便的持久层访问。

下面以 DBCP 为例，介绍数据源的配置。

数据源的配置也有两种方式，这两种方式配置数据源的访问范围不同：一种对所有的 Web 应用都可以访问，另一种只能在某个 Web 应用中访问。前者称为全局数据源，后者称为局部数据源。

不管配置哪种数据源，都必须提供数据源实现的 jar 文件，还有数据库驱动。笔者以 MySql 数据库为例，要配置其数据源则必须提供 MySql 的驱动和 DBCP 数据源的 jar 文件。

登录 <http://jakarta.apache.org/commons/dbcp/> 站点可下载 DBCP。另外，DBCP 还依赖于另外两个项目：commons-pool 和 commons-collections。这三个项目都属于 Apache 的 Jakarta 子项目，在该站点下载这三个项目并解压缩下载到的三个 zip 文件，可以得到如下三个 jar 文件：

- commons-dbcp-1.2.1.jar
- commons-pool-1.2.jar
- commons-collections-3.1.jar

将这三个文件复制到 Tomcat 的 common/lib 路径下，即为 Tomcat 提供了配置数据源所必需的类库。

下面先介绍局部数据源的配置：

局部数据源无须修改系统的配置文件，而是增加用户自己的配置文件。这样不会造

成系统的混乱，而且数据源被封装在一个 Web 应用之内，提供了更好的封装性，防止被其他的 Web 应用访问。

局部数据源只与特定的 Web 应用相关，因此只在特定 Web 应用的配置文件中配置。例如，为上面的 Web 应用增加局部数据源，修改 Tomcat 的 conf\Catalina\localhost 路径下的 Web 配置文件即可。

增加局部数据源后，完整的配置文件如下：

```
<?xml version='1.0' encoding='gb2312'?>
<!-- 配置一个 Web 应用-->
<Context path="/test" docBase="e:/webroot" debug="0" privileged="true">
    <!-- 配置一个资源，资源的名称为 jdbc/dstest，类型为 DataSource 数据源-->
    <Resource name="jdbc/dstest" auth="Container" type="javax.sql.DataSource"/>
    <!-- 定义资源的参数 name 属性指定定义哪个资源的参数-->
    <ResourceParams name="jdbc/dstest">
        <!-- 下面是定义数据源的参数-->
        <parameter>
            <!-- 定义数据源工厂-->
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>
        <!-- 定义数据源的最大活动连接数
        -->
        <parameter>
            <name>maxActive</name>
            <value>100</value>
        </parameter>
        <!-- 定义数据源的超时时长，超过该时间，数据源自动断开连接
        -->
        <parameter>
            <name>removeAbandonedTimeout</name>
            <value>60</value>
        </parameter>
        <!-- 定义数据源中最大空闲连接数，
            一旦容器中空闲连接数超过该数，系统将自动销毁一些连接
        -->
        <parameter>
            <name>maxIdle</name>
            <value>30</value>
        </parameter>
        <!-- 数据源的最大等待数
        -->
        <parameter>
            <name>maxWait</name>
            <value>10000</value>
        </parameter>
        <!-- 连接数据库的用户名 -->
        <parameter>
            <name>username</name>
            <value>root</value>
        </parameter>
        <!-- 连接数据库的密码 -->
        <parameter>
            <name>password</name>
            <value>32147</value>
        </parameter>
        <!-- 指定连接数据库所使用的驱动 -->
```

```

<parameter>
    <name>driverClassName</name>
    <value>com.mysql.jdbc.Driver</value>
</parameter>
<!-- 指定连接数据库的 URL
-->
<parameter>
    <name>url</name>
    <value>jdbc:mysql://localhost:3306/j2ee?autoReconnect=true</value>
</parameter>
</ResourceParams>
</Context>

```

再次启动 Tomcat，该 Web 应用即可通过 JNDI 访问数据源，下面是访问该数据源的代码：

```

//初始化 Context，使用 InitialContext 初始化 Context
Context ctx=new InitialContext();
/*
    通过 JNDI 查找数据源，该 JNDI 为 java:comp/env/jdbc/dstest，分成两个部分
    java:comp/env 是 Tomcat 固定的，Tomcat 提供的 JNDI 绑定都必须加该前缀
    jdbc/dstest 是定义数据源时的数据源名
*/
DataSource ds=(DataSource)ctx.lookup("java:comp/env/jdbc/dstest");
//获取数据库连接
Connection conn=ds.getConnection();
//获取 Statement
Statement stmt=conn.createStatement();
//执行查询，返回 Resultset 对象
ResultSet rs=stmt.executeQuery("select * from newsinf");
while(rs.next())
{
    ...
}

```

上面配置了局部数据源，如还需要配置全局数据源，可通过 Administration 控制台，或者修改 server.xml 配置文件来实现。

通过控制台的配置方式相当简单，因为所作的任何修改，最终依然会变成对 server.xml 文件的修改。因此笔者建议使用直接修改 server.xml 的方式。

**注意：使用全局数据源会破坏 Tomcat 原有的配置文件。**

在 server.xml 文件中找到 GlobalNamingResources 元素，该元素下负责配置所有的全局资源，因此只需在该元素增加数据源的配置即可。增加数据源配置的配置片段如下：

```

<!-- 配置全局命名资源-->
<GlobalNamingResources>
    <!-- 下面省略了系统中其他全局资源的配置-->
    ...
    <!-- 配置一个资源，资源的名称为 jdbc/dstest，类型为 DataSource 数据源-->
    <Resource name="jdbc/dstest" auth="Container" type="javax.sql.DataSource" />
    <!-- 定义资源的参数 name 属性指定定义哪个资源的参数-->
    <ResourceParams name="jdbc/dstest">

```

```
<!-- 下面是定义数据源的参数-->
<parameter>
    <!-- 定义数据源工厂-->
    <name>factory</name>
    <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>
<!-- 定义数据源的最大活动连接数
-->
<parameter>
    <name>maxActive</name>
    <value>100</value>
</parameter>
<!-- 定义数据源的超时时长，超过该时间，数据源自动断开连接
-->
<parameter>
    <name>removeAbandonedTimeout</name>
    <value>60</value>
</parameter>
<!-- 定义数据源中最大空闲连接数，一旦容器中空闲连接数超过该数，系统将自动销毁一些连接
-->
<parameter>
    <name>maxIdle</name>
    <value>30</value>
</parameter>
<!-- 数据源的最大等待数
-->
<parameter>
    <name>maxWait</name>
    <value>10000</value>
</parameter>
<!-- 连接数据库的用户名 -->
<parameter>
    <name>username</name>
    <value>root</value>
</parameter>
<!-- 连接数据库的密码 -->
<parameter>
    <name>password</name>
    <value>32147</value>
</parameter>
<!-- 指定连接数据库所使用的驱动 -->
<parameter>
    <name>driverClassName</name>
    <value>com.mysql.jdbc.Driver</value>
</parameter>
<!-- 指定连接数据库的 URL
-->
<parameter>
    <name>url</name>
    <value>jdbc:mysql://localhost:3306/j2ee?autoReconnect=true</value>
</parameter>
</ResourceParams>
</GlobalNamingResources>
```

通过这种方式配置的数据源可以被所有的 Web 应用访问。

## 1.3 Jetty 的下载和安装

Jetty 是 Java 领域另一个出色的 Web 服务器，这个服务器同样是开源项目。相对于 Tomcat，Jetty 有更大的优点——可作为一个嵌入式服务器，即如果在应用中加入 Jetty 的 Jar 文件，则应用可在代码中对外提供 Web 服务。

下面主要介绍 Jetty 的下载、安装和基本配置。

### 1.3.1 Jetty 的下载和安装

Jetty 也是与平台无关的 Java Web 服务器，既可以在 Windows 平台上运行，也可以在 Linux 平台上运行，下载和安装 Jetty 请按如下步骤进行。

(1) 登录 <http://jetty.mortbay.com/jetty/index.html> 站点，下载 Jetty 的最新版本。笔者成书之时，Jetty 的最新版本是 jetty-6.0.0rc0，下载 jetty-6.0.0rc0.zip 文件，该文件是与平台无关的压缩包，不管是 Windows 还是 Linux 都可使用该压缩包。

(2) 解压缩 jetty-6.0.0rc0.zip 文件，应得到如下的文件结构。

- etc: 该路径用于存放 Jetty 的配置文件。
- examples: 该路径用于存放 Jetty 的示例。
- legal: 该路径用于存放该项目的 Lisence 信息。
- lib: 该路径用于存放运行 Jetty 必需的 Jar 文件。
- modules: 该路径用于存放 Jetty 的模块，包括 API 文档。
- patches: 包含一些补丁说明。
- pom.xml: 是 Jetty 的 build 文件，该文件不是 Ant 的 build 文件，而是 mavaen2 的 build 文件。
- project-site: 包含 Jetty 的网站的必需的样式文件。
- readme.txt: 包含最基本的使用信息。
- start.jar: 启动 Jetty 的启动文件。
- version.txt: Jetty 版本更新日志的简单版本。
- webapps: 该路径用于存放自动部署的 Web 应用，只要将用户的 Web 应用复制到该路径下，Web 应用将自动部署。
- webapps-plus: 存放一些用于演示 Jetty 扩展属性的 Web 应用，该路径下的 Web 应用也可自动部署。

(3) 将解压缩后的文件放在任意路径即可。运行 Jetty 需要使用如下命令：

```
java -jar start.jar
```

建议将上面命令写成脚本，如在 Windows 下写成批处理命令，Linux 下写成一个 shell 脚本。每次直接运行其脚本文件即可。

运行成功后，启动浏览器并在地址栏输入 <http://localhost:8080/>，然后回车，出现如

图 1.14 所示的界面，即表示 Jetty 安装成功。

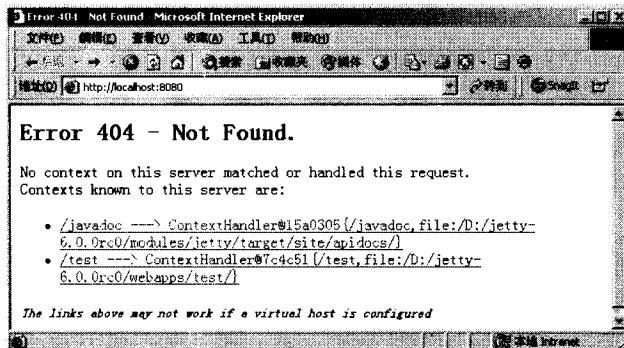


图 1.14 Jetty 安装成功的界面

### 1.3.2 Jetty 的基本配置

Jetty 的基本配置与 Tomcat 类似，在这里主要介绍 Jetty 的端口设置及如何部署 Web 应用。由于 Jetty 是个嵌入式 Web 服务器，因此设置方式比较特殊。

Jetty 的配置文件放在 etc 路径下，该路径下有如下几个配置文件：

- jetty.xml 文件。
- jetty-jmx.xml 文件。
- jetty-plus.xml 文件。
- webdefault.xml 文件。

其中 webdefault.xml 文件是 Web 应用的默认配置文件，与 Jetty 的配置没有太大的关系，该文件通常不需要修改。

另外的三个都是 Tomcat 的配置文件：jetty.xml 文件是默认的配置文件；jetty-jmx.xml 是启动 JMX 控制的配置文件；jetty-plus.xml 文件是在增加 Jetty 扩展功能的配置文件。

在启动 Jetty 时输入如下命令：

```
java -jar startup.jar
```

默认使用 jetty.xml 文件时启动 Jetty，即与如下命令效果相同：

```
java -jar startup.jar etc/jetty.xml
```

启动时也可以指定多个配置文件，可输入如下命令：

```
java -jar startup.jar etc/jetty.xml etc/jetty-plus.xml
```

打开 Jetty 配置文件，该配置文件的根元素是 Configure，另外还会看到有如下的配置元素。

- Set：相当于调用 setXxx 方法。
- Get：相当于调用 getXxx 方法。

- New: 创建某个类的实例。
- Arg: 为方法或构造器传入参数。
- Array: 设置一个数组。
- Item: 设置数组或集合的一项。
- Call: 调用某个方法。

Jetty 是个嵌入式 Web 容器，因此它的服务对应一个 Server 实例，可以看到配置文件中有如下片段：

```
<!-- 配置了一个 Jetty 服务器进程-->
<Configure id="Server" class="org.mortbay.jetty.Server">
```

## 1. 配置 Jetty 服务端口

Configure 元素里的各种子元素，即对该 Server 实例的操作。在 Configure 元素下有如下代码所示的 Set 子元素，Set 子元素的 name 属性为 connectors，效果等同于调用 setConnectors 方法，用于设置 Web 服务的提供端口。该方法需要 Connector 数组，其包含的 Array 子元素则用于设置该方法的参数。Array 元素里的 Item 子元素，则是数组的数据项，每个 Connector 对应一个连接提供者。

```
<!-- 类似于调用 setConnectors 方法-->
<Set name="connectors">
    <!-- 为 setConnectors 方法传入参数-->
    <Array type="org.mortbay.jetty.Connector">
        <!-- 下面的 Connector 提供常见的 Web 服务
        -->
        <Item>
            <New class="org.mortbay.jetty.nio.SelectChannelConnector">
                <Set name="port">8080</Set>
                <Set name="maxIdleTime">30000</Set>
                <Set name="lowResourceMaxIdleTime">3000</Set>
                <Set name="Acceptors">1</Set>
            </New>
        </Item>
        <!-- 如果 Java 的 nio 不可用，则使用如下的 Connector-->
        <!--
        <Item>
            <New class="org.mortbay.jetty.bio.SocketConnector">
                <Set name="port">8081</Set>
                <Set name="maxIdleTime">50000</Set>
            </New>
        </Item>
        -->
        <!-- Use this connector for few very active connections ONLY IF
             SelectChannelConnector cannot handle your load
        <Item>
            <New class="org.mortbay.jetty.nio.BlockingChannelConnector">
                <Set name="port">8083</Set>
                <Set name="maxIdleTime">30000</Set>
                <Set name="lowResourceMaxIdleTime">3000</Set>
            </New>
        </Item>
        -->
        <!-- 下面的 Connector 用于设置 HTTPS 的服务提供端口-->
        <!--
```

```

<Item>
  <New class="org.mortbay.jetty.security.SslSocketConnector">
    <Set name="Port">8443</Set>
    <Set name="maxIdleTime">30000</Set>
    <Set name="Keystore"><SystemProperty name="jetty.home" default=". "/>/etc/keystore</Set>
    <Set name="Password">OBF:1vny1z1o1x8e1vnw1vn61x8g1zl1vn4</Set>
    <Set name="KeyPassword">OBF:lu2u1wm1z7s1z7a1wn1u2g</Set>
  </New>
</Item>
-->
</Array>
</Set>

```

在上面的配置片段中，默认第一个 Connector 是有效的，该 Connector 就是常规 Web 服务的 Connector，其中的 8080 就是 Jetty 的默认端口。

笔者将该片段修改如下：

```

<!-- 下面的 Connector 提供常见的 Web 服务
-->
<Item>
  <!-- 提供基于 nio 的 Connector-->
  <New class="org.mortbay.jetty.nio.SelectChannelConnector">
    <!-- 设置端口号-->
    <Set name="port">8886</Set>
    <Set name="maxIdleTime">30000</Set>
    <Set name="lowResourceMaxIdleTime">3000</Set>
    <Set name="Acceptors">1</Set>
  </New>
</Item>

```

修改成上面所示的样例后，Jetty 的服务端口为 8886。这也是笔者所使用的端口。

## 2. 部署 Web 应用

Jetty 也支持自动部署和配置文件部署。

如果使用默认的配置文件启动，webapps 会自动部署目录。即所有存放在 webapps 路径的 Web 应用将自动部署在 Jetty 容器中。

如果使用带 Jetty 扩展功能来启动，即增加 jetty-plus.xml 文件来启动，则 webapps-plus 也会自动部署目录，将所有放在该路径的 Web 应用自动部署在 Jetty 容器中。

下面看如何使用配置文件来部署 Web 应用。

部署 Web 应用需使用 org.mortbay.jetty.webapp.WebAppContext，该类的实例即对应一个 Web 应用，并且该类还包含多个静态的重载方法：addWebApplications。该方法用于同时部署多个 Web 应用，即用于配置一个自动部署目录。

jetty.xml 配置文件的片段如下：

```

<!-- 调用 WebAppContext 的静态方法 addWebApplications -->
<Call class="org.mortbay.jetty.webapp.WebAppContext" name="addWebApplications">
  <!-- 下面用于为方法传入参数-->
  <Arg><Ref id="contexts"/></Arg>
  <!-- 指定自动部署目录-->
  <Arg>./webapps</Arg>
  <!-- 配置 Web 应用的默认配置文件-->

```

```

<Arg><SystemProperty name="jetty.home" default=". "/>/etc/webdefault.xml</Arg>
<!-- 是否解压缩-->
<Arg type="boolean">True</Arg>
<Arg type="boolean">False</Arg>
</Call>

```

jetty-plus.xml 文件的片段如下：

```

<!-- 调用 WebApplicationContext 的静态方法 addWebApplications -->
<Call class="org.mortbay.jetty.webapp.WebApplicationContext" name=
"addWebApplications">
    <!-- 下面用于为方法传入参数-->
    <Arg><Ref id="Server"/></Arg>
    <!-- 指定自动部署目录-->
    <Arg>./webapps-plus</Arg>
    <!-- 配置 Web 应用的默认配置文件-->
    <Arg>org/mortbay/jetty/webapp/webdefault.xml</Arg>
    <Arg><Ref id="plusConfig"/></Arg>
    <!-- 是否解压缩-->
    <Arg type="boolean">True</Arg>
    <Arg type="boolean">False</Arg>
</Call>

```

通过查看该配置文件不难发现，在每次调用 `addWebApplications` 方法后，即可增加一个 Web 应用的自动部署路径。如有必要，用户完全可以增加自己的自动部署路径，如果增加了自动部署路径，则所有在该路径下的 Web 应用将自动部署。

如果仅需要部署一个 Web 应用，可以有如下两种方法：

- 修改 `jetty.xml` 文件。
- 增加自己的配置文件。

根据前面的介绍，对于 Web 服务器，应尽量避免修改默认的配置文件。如果读者真需要通过修改 `jetty.xml` 文件来部署 Web 应用，则应在 `jetty` 的 `Configure` 元素下增加如下片段：

```

<!-- 创建一个 Web 应用-->
<New class="org.mortbay.jetty.webapp.WebAppContext">
    <!-- 三个构造参数-->
    <Arg><Ref id="contexts"/></Arg>
    <!-- 设置 Web 应用的文档路径-->
    <Arg>G:/StrutsTest/js</Arg>
    <!-- 设置 Web 应用的 url-->
    <Arg>/</Arg>
    <!-- 设置 Web 应用的默认配置描述符-->
    <Set name="defaultsDescriptor">
        <SystemProperty name="jetty.home" default=". "/>/etc/webdefault.xml</Set>
    <!-- 相当于调用 setVirtualHosts 方法，用于设置虚拟主机-->
    <Set name="virtualHosts">
        <!-- Array 表示创建一个数组-->
        <Array type="java.lang.String">
            <Item>localhost</Item>
        </Array>
    </Set>
    <!-- 类似于调用 getSessionHandler 方法-->
    <Get name="SessionHandler">
        <Set name="SessionManager">

```

```

<New class="org.mortbay.jetty.servlet.HashSessionManager">
    <!-- 设置 Session 的超时时长-->
    <Set name="maxInactiveInterval" type="int">600</Set>
</New>
</Set>
</Get>
</New>

```

**注意：**该代码片段在 jetty.xml 文件仅仅被注释，只要取消该代码片段注释即可。但需要注意：jetty.xml 文件默认有个小错误，它的设置超时时长的 Set 元素的 name 属性值为 maxInactiveIntervale。实际上 HashSessionManager 并没有 setMaxInactiveIntervale 方法，通过查看 API 文档发现，它包含一个 setMaxInactiveInterval 方法（最后少一个 e），读者将原有的 e 删除即可。

通常建议增加自己的配置文件，应尽量避免修改系统原有的配置文件。

增加的配置文件如下：

```

<?xml version="1.0" encoding="gb2312"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
"http://jetty.mortbay.org/configure.dtd">
<Configure id="Server" class="org.mortbay.jetty.Server">
    <!-- 创建一个 Web 应用-->
    <New class="org.mortbay.jetty.webapp.WebAppContext">
        <!-- 三个构造参数-->
        <Arg><Ref id="contexts"/></Arg>
        <!-- 设置 Web 应用的文档路径-->
        <Arg>G:/StrutsTest/jsp</Arg>
        <!-- 设置 Web 应用的 url-->
        <Arg>/</Arg>
        <!-- 设置 Web 应用的默认配置描述符-->
        <Set name="defaultsDescriptor">
            <SystemProperty name="jetty.home" default=". "/>/etc/webdefault.xml</Set>
        <!-- 相当于调用 setVirtualHosts 方法，用于设置虚拟主机-->
        <Set name="virtualHosts">
            <!-- Array 表示创建一个数组-->
            <Array type="java.lang.String">
                <Item>localhost</Item>
            </Array>
        </Set>
        <!-- 类似于调用 getSessionHandler 方法-->
        <Get name="SessionHandler">
            <Set name="SessionManager">
                <New class="org.mortbay.jetty.servlet.HashSessionManager">
                    <!-- 设置 Session 的超时时长-->
                    <Set name="maxInactiveInterval" type="int">600</Set>
                </New>
            </Set>
        </Get>
    </New>
</Configure>

```

将该配置文件保存在 etc 路径下，以后每次启动 Jetty 时，可直接加载该配置文件，使用如下启动命令即可（假设该配置文件的文件名为 jetty-yeeku.xml）：

```
java -jar startup.jar etc/jetty.xml etc/jetty-yeeku.xml
```

### 3. 配置 JNDI 绑定

Jetty 同样可以整合 DBCP、C3P0 等数据源来提供容器管理的数据源。提供容器管理的数据源，只是 Jetty JNDI 绑定功能之一。

下面介绍如何在 Jetty 绑定 JNDI，以及 JNDI 的使用。

增加 JNDI 绑定必须使用 Jetty 的 plus 功能。因此，启动时必须增加 jetty-plus.xml 文件。增加 JNDI 的绑定同样有两个方法：

- 修改系统默认的 jetty.xml 文件。
- 增加自己的配置文件。

两种配置方式大同小异，区别是前者需要修改系统默认的配置文件，此处仅介绍增加自己的配置文件方式。

在 Jetty 的 plus 中，有如下包。

`org.mortbay.jetty.plus.naming`: 执行 JNDI 绑定的包。

该包下有如下四个类。

`EnvEntry`: 绑定简单值。

`NamingEntry`: 抽象类，是另外三个类的父类。

`Resource`: 用于绑定数据源等资源。

`Transaction`: 用于绑定事务。

增加数据源绑定请按如下步骤进行。

(1) 此处绑定的数据源依然以 DBCP 为实现，当然也可以绑定 C3P0 数据源，但必须将 DBCP 所需要的 jar 文件复制到 Jetty 可以使用的路径中。根据前面介绍 DBCP 主要需要如下三个文件：

- `commons-dbcp.jar`
- `commons-pool.jar`
- `commons-collections.jar`

将这三个文件复制到 Jetty 的 lib 路径下即可，Jetty 启动时会自动加载该路径的 jar 文件。当然，还需将数据库驱动复制到该路径下。

(2) 增加如下配置文件：

```
<?xml version="1.0"?>
<!-- Jetty 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
"http://jetty.mortbay.org/configure.dtd">
<!-- Jetty 配置文件的根元素-->
<Configure id="Server" class="org.mortbay.jetty.Server">
<!-- 配置第一个环境变量，只是一个普通值-->
<New id="woggle" class="org.mortbay.jetty.plus.naming.EnvEntry">
<Arg>woggle</Arg>
<Arg type="java.lang.Integer">4000</Arg>
</New>
<!-- 配置第二个环境变量，只是一个普通值-->
<New id="wiggle" class="org.mortbay.jetty.plus.naming.EnvEntry">
<Arg>wiggle</Arg>
<Arg type="java.lang.Double">100</Arg>
<Arg type="boolean">true</Arg>
```

```
</New>
<!-- 创建一个数据源-->
<New id="ds" class="org.apache.commons.dbcp.BasicDataSource">
    <!-- 设置数据库驱动-->
    <Set name="driverClassName">com.mysql.jdbc.Driver</Set>
    <!-- 设置数据库 url-->
    <Set name="url">jdbc:mysql://localhost:3306/j2ee</Set>
    <!-- 设置数据库用户名-->
    <Set name="username">root</Set>
    <!-- 设置数据库密码 -->
    <Set name="password">32147</Set>
    <!-- 设置数据库驱动-->
    <Set name="maxActive" type="int">100</Set>
    <!-- 设置数据源最大空闲连接数-->
    <Set name="maxIdle" type="int">30</Set>
    <!-- 设置数据源最大的等待数-->
    <Set name="maxWait" type="int">1000</Set>
    <!-- 设置数据库是否自动提交-->
    <Set name="defaultAutoCommit" type="boolean">true</Set>
    <!-- 设置连接是否自动删除-->
    <Set name="removeAbandoned" type="boolean">true</Set>
    <!-- 设置数据库驱动-->
    <Set name="removeAbandonedTimeout" type="int">60</Set>
    <Set name="logAbandoned" type="boolean">true</Set>
</New>
<!-- 将实际的数据源绑定到 jdbc/mydatasource 这个 JNDI 名-->
<New id="mydatasource" class="org.mortbay.jetty.plus.naming.Resource">
    <Arg>jdbc/mydatasource</Arg>
    <Arg><Ref id="ds"/></Arg>
</New>
</Configure>
```

在上面的配置文件中，绑定了三个 JNDI 值，下面测试该 JNDI 的 Servlet：

```
public class TestServlet extends HttpServlet
{
    InitialContext ic;
    //Servlet 的初始化方法，该方法完成 Context 的初始化
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        try
        {
            ic = new InitialContext();
        }
        catch (Exception e)
        {
            throw new ServletException(e);
        }
    }
    //service 方法是 Servlet 的服务方法
    public void service(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException
    {
        //获取 JSP 页面输出流
        PrintStream out = new PrintStream(response.getOutputStream());
        try
        {
```

```
//在控制台输出 wiggle 的绑定值  
System.out.println(ic.lookup("wiggle"));  
//在控制台输出 woggle 的绑定值  
System.out.println(ic.lookup("woggle"));  
//获取绑定的数据源  
DataSource ds = (DataSource)ic.lookup("jdbc/mydatasource");  
//通过数据源获取数据库连接  
Connection conn = ds.getConnection();  
//通过数据库连接创建 Statement 对象  
Statement stmt = conn.createStatement();  
//通过 Statement 对象执行 SQL 查询，返回 ResultSet 对象  
ResultSet rs = stmt.executeQuery("select * from news");  
//遍历记录集  
while(rs.next())  
{  
    out.println(rs.getString(2));  
}  
}  
catch (Exception e)  
{  
    e.printStackTrace();  
}  
}  
}
```

在 web.xml 文件中增加如下片段：

```
<servlet>  
    <!-- 定义 Servlet-->  
    <servlet-name>aa</servlet-name>  
    <servlet-class>lee.TestServlet</servlet-class>  
  </servlet>  
  <servlet-mapping>  
    <!-- 定义 Servlet 的 url 映射-->  
    <servlet-name>aa</servlet-name>  
    <url-pattern>/aa</url-pattern>  
  </servlet-mapping>
```

启动 Jetty，访问该 Servlet，即看到数据库的访问结果。

## 1.4 Eclipse 的安装和使用

Eclipse 是一个免费的 IDE（集成开发环境）工具，它支持多种开发语言，并不仅仅用于 Java 应用的开发。在免费的 Java 开发工具中，Eclipse 是最受欢迎的。

Eclipse 本身所提供的开发功能非常有限，但它的插件则大大提高了它的功能。Eclipse 的插件非常多，比如 Synchronizer, Lomboz, MyEclipse 等。下面就简单介绍 Eclipse 的安装和使用。

### 1.4.1 Eclipse 的下载和安装

登录 <http://www.eclipse.org> 站点，下载 Eclipse 的最新版本。Eclipse 当前的最新版本

是 3.2，笔者建议下载 Eclipse 3.2。

根据所使用的操作系统，在 Windows 平台上下载 `eclipse-SDK-3.2-win32.zip`；在 Linux 平台上下载 `eclipse-SDK-3.2-linux-gtk.tar.gz` 文件。将下载到的压缩文件解压缩，解压缩后的文件夹可放在任何目录中。

直接双击 `eclipse.exe` 文件，即看到 Eclipse 的启动界面，表明 Eclipse 已经安装成功。

## 1.4.2 Eclipse 插件的安装

Eclipse 本身的开发能力非常有限，但它的插件功能非常强大。Eclipse 的插件的安装方式分为如下两种：

- 在线安装。
- 手动安装。

### 1. 在线安装

在线安装简单方便，适合网络畅通的场景。在线安装是个较好的安装方式，请按如下步骤进行。

(1) 单击“Help”菜单，然后将光标移动到“Software Updates”菜单项上，单击 Software Updates 菜单项的“Find And Install”子菜单项，如图 1.15 所示。

(2) 此时将弹出如图 1.16 所示的对话框，该对话框用于选择安装新插件，或升级已有插件。该对话框中有两个单选按钮：上面一个用于升级已有插件；下面一个用于安装新插件。

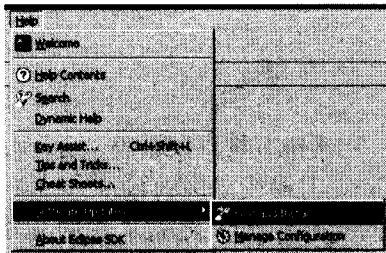


图 1.15 在线安装插件的菜单

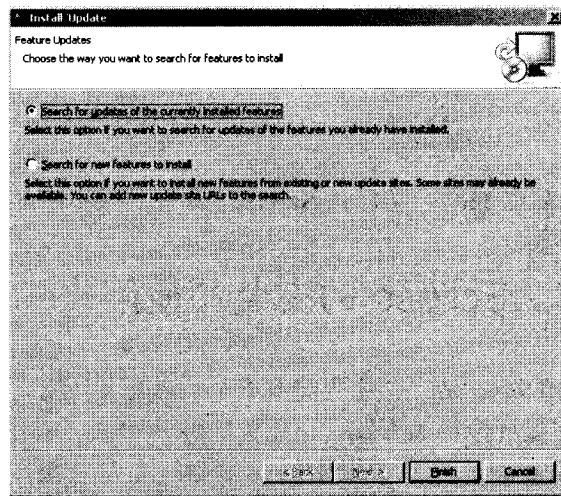


图 1.16 选择升级或安装新插件

(3) 如果需要升级已有插件，则选择第一个单选框，单击【Finish】按钮即可，等待 Eclipse 完成升级。

(4) 如果需要安装新插件，则选择第二个单选按钮，【Next】按钮将变成可用，单击【Next】按钮，弹出如图 1.17 所示的对话框。

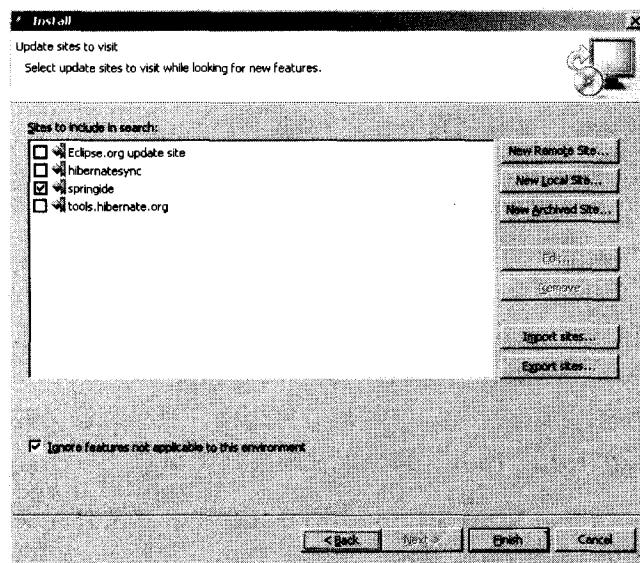


图 1.17 安装新插件

(5) 图 1.17 中空白处显示的是系统已经定义的插件安装地址。如需增加新的插件安装地址，可单击右边的【New Remote Site】按钮，将弹出如图 1.18 所示的对话框。

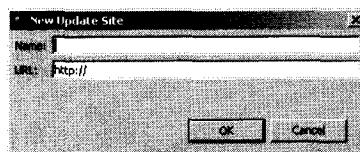


图 1.18 定义新安装地址

(6) 在图 1.18 显示的对话框的【Name】文本框中输入插件名，在【URL】文本框中输入插件地址。单击【OK】按钮，输出成功，返回图 1.17 所显示的对话框。

(7) 在如图 1.17 所示的对话框中选择需要安装的插件，单击【Finish】按钮，进入安装界面。

## 2. 手动安装

手动安装只需要已经下载的插件文件，无须网络支持。手动安装也分为两种安装方式：

- 直接安装。
- 扩展安装。

**直接安装：**将插件中包含的 plugins 和 features 文件夹的内容，复制到 Eclipse 的 plugins 和 features 文件夹内，重新启动 Eclipse 即可。

直接安装简单易用，但效果非常不好，因为容易导致混乱。如果安装的插件非常多，可能导致用户无法精确判断哪些是 Eclipse 默认的插件，哪些是后来扩展的插件。

通常推荐使用扩展安装，扩展安装请按如下步骤进行。

(1) 在 Eclipse 安装路径下新建 links 路径。

(2) 在 links 文件夹内，建立×××.link 文件，该文件的文件名可随意，但后缀必须是 link，通常推荐该文件的文件名与插件名相同。

(3) 编辑×××.link 的内容，该文件内通常只需一行：

path=sync，其中 path=是固定的，而 sync 是文件夹名。

(4) 在×××.link 文件中 path 所指的路径下新建 eclipse 文件夹，再在 eclipse 文件夹内建立 plugins 和 features 文件夹。

(5) 将插件中包含的 plugins 和 features 文件夹的内容，复制到上面建立的 plugins 和 features 文件夹中，重启 Eclipse 即可完成安装。

由此可看出使用扩展安装结构非常清晰，每个插件有单独的文件夹。如果需要卸载某个插件，只需将该插件对应的 link 文件删除即可。

注意：还有一些插件，如 MyEclipse，已经提供了安装文件。安装这种插件更加简单，就像安装其他可执行性程序一样。

### 1.4.3 Eclipse 的简单使用

下面以建立一个简单的 Web 应用为例，简单介绍 Eclipse 的使用（笔者已经安装了 MyEclipse 插件）。

为了建立 Web 应用，首先必须提供 Web 服务器的支持，配置 Web 服务器请按如下步骤进行。

(1) 单击“Window”菜单，选择“Preferences”菜单项，将弹出如图 1.19 所示的对话框。

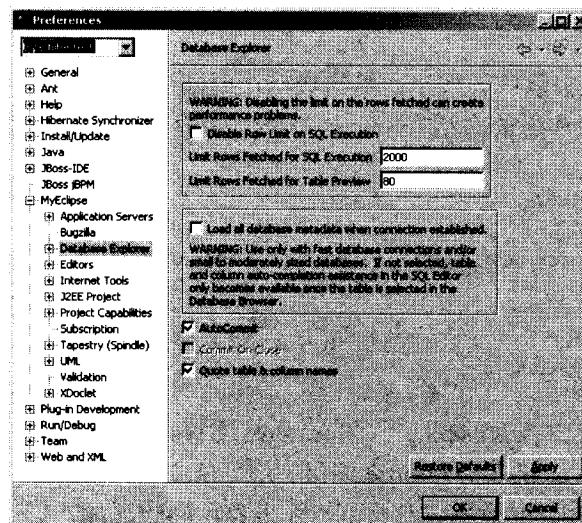


图 1.19 配置参数的对话框

(2) 在图 1.19 所示对话框的左边树形结构中, 单击 “MyEclipse” 节点, 然后再单击 “Application Server” 节点。

(3) 在 “Application Server” 下选择需要配置的服务器, 并单击服务器对应的节点, 以 Tomcat 5.0 为例, 将出现如图 1.20 所示的界面。

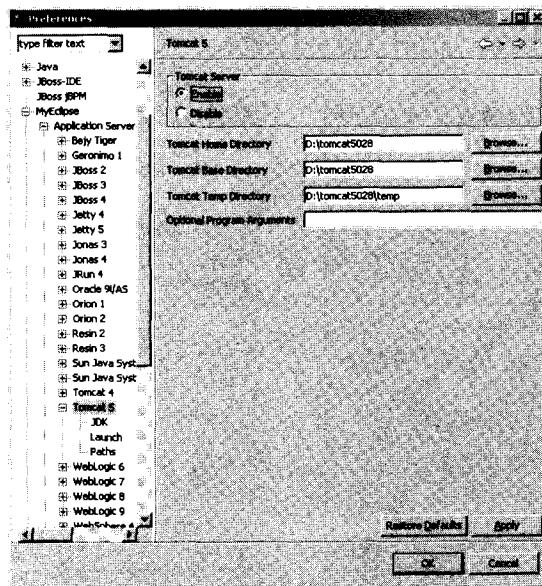


图 1.20 配置 Web 服务器

(4) 选中 “Enable” 单选按钮, 并在 “Tomcat Home Directory” 文本框中输入 Tomcat 的安装路径。单击 【OK】 按钮, 即完成了服务器的安装。

建立一个 Web 应用请按如下步骤进行。

(1) 单击 “File” 菜单, 将光标移到 “New” 菜单项上, 在出现的子菜单中单击 “Other...” 菜单项, 将弹出如图 1.21 所示的对话框。

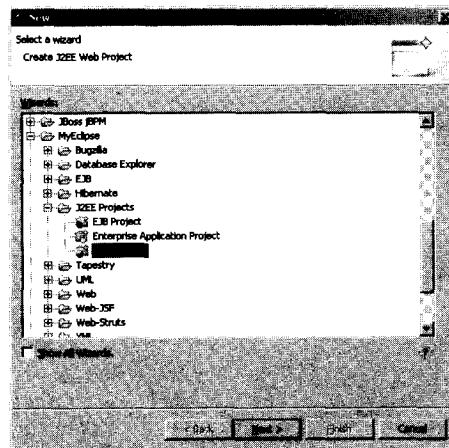


图 1.21 新建 Web 项目

(2) 单击“MyEclipse”节点，并打开“J2EE Projects”节点，然后再单击“Web Project”节点，将弹出如图 1.22 所示的界面。

(3) 在图 1.22 所示的对话框中的“Project Name”文本框中输入项目名，单击【Finish】按钮，即完成 Web 应用的建立。

(4) 单击“File”菜单，将光标移到“New”菜单项上，在出现的子菜单中单击“JSP”菜单项，将弹出如图 1.23 所示的对话框。

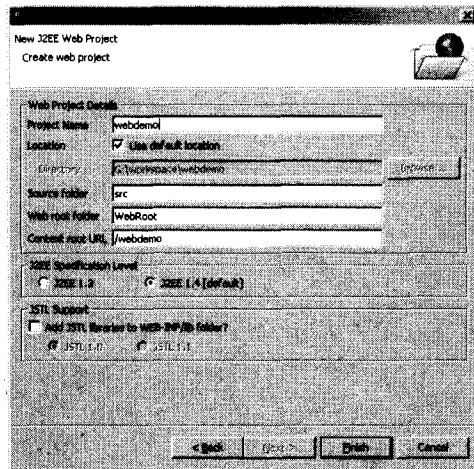


图 1.22 建立 Web 应用

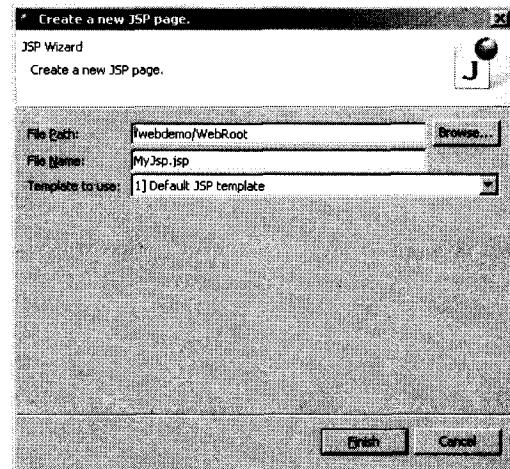


图 1.23 新建 JSP 页面

(5) 在图 1.23 所示的“File Name”文本框中，输入 JSP 的文件名，单击【Finish】按钮，即建立了 JSP 页面。

(6) 编辑 JSP 页面。

(7) 单击工具栏的【Deploy MyEclipse J2EE Project To Server】工具按钮，即弹出如图 1.24 所示的对话框。

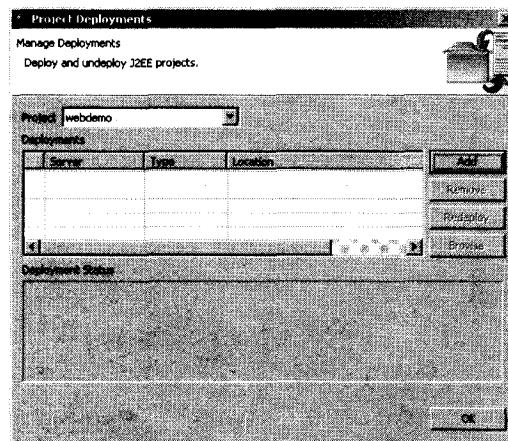


图 1.24 部署 Web 应用

(8) 在图 1.24 所示对话框的 Project 下拉列表中选中需要部署的项目，然后单击【Add】按钮，将弹出如图 1.25 所示的对话框。

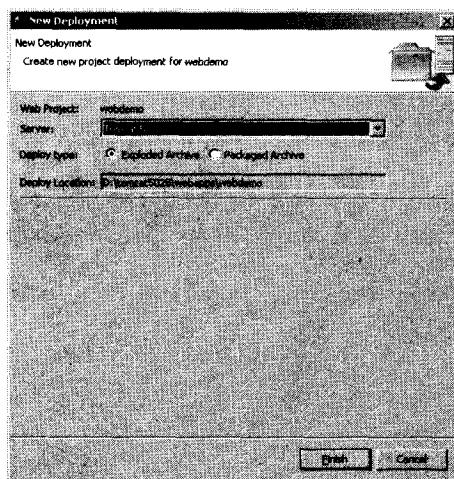


图 1.25 选择服务器

(9) 在图 1.25 所显示对话框的 Server 下拉列表中选中部署的目标服务器，然后单击【Finish】按钮，即完成了 Web 应用的部署。

(10) 在 Eclipse 的工具栏中单击【Run/Stop MyEclipse Application Servers】按钮，启动 Tomcat 5，然后在浏览器中输入刚编辑的 JSP 页面的 URL，即可访问该 JSP 页的内容。

## 本章小结

本章主要介绍了如何搭建轻量级 J2EE 应用的开发环境，同时介绍了在两个平台上开发环境的搭建过程。

另外，详细介绍了经典 Web 容器 Tomcat 的安装和配置，包括 Web 应用的部署和数据源的配置；还介绍了另一个 Web 容器 Jetty，包括 Jetty 的配置原理及 Jetty 配置元素的含义，以及在 Jetty 中如何部署 Web 应用，如何配置数据源等。

最后，详细介绍了 Eclipse 的安装和使用，包括插件的安装，并以一个 Web 应用为示例，简要介绍了 Eclipse 的使用。

# 第 2 章

## 传统表现层 JSP

### 本章要点

- 『 JSP 的技术原理
- 『 JSP 的注释和声明
- 『 JSP 脚本
- 『 JSP 的三个编译指令
- 『 JSP 的七个处理指令
- 『 JSP 的九个内置对象
- 『 Servlet 介绍
- 『 自定义标签开发
- 『 使用 Filter
- 『 使用 Listener
- 『 JSP 2.0 的新特性

JSP 是 Java Server Page 的缩写，是 Servlet 的简化。它是由 Sun 公司提出的，并由许多公司参与制定的一种动态网页标准。其主要特点是在 HTML 页面中加入 Java 代码片段，或者使用各种 JSP 标签，包括使用用户标签，构成 JSP 网页。

早期使用 JSP 页面的用户非常广泛，一个 Web 应用可以全部由 JSP 页面组成，只辅以少量的 JavaBean 即可。自 J2EE 标准出现以后，人们逐渐认识到使用 JSP 充当过多的角色是不合适的。因此，JSP 慢慢发展成单一的表现层技术，不再承担业务逻辑组件及持久层组件的责任。

虽然有各种模板技术，但 JSP 还是最经典、应用最广的表现层技术。

## 2.1 JSP 的技术原理

JSP 是 Servlet 的扩展，在没有 JSP 之前，就已经出现了 Servlet 技术。Servlet 是利用输出流动态生成 HTML 页面，包括每一个 HTML 标签和每个在 HTML 页面中出现的内容。

由于包括大量的 HTML 标签、大量的静态文本及格式等，导致 Servlet 的开发效率极为低下。所有的表现逻辑，包括布局、色彩及图像等，都必须耦合在 Java 代码中，这的确让人不胜其烦。JSP 的出现弥补了这种不足，JSP 通过在标准的 HTML 页面中插入 Java 代码，其静态的部分无须 Java 程序控制，只有那些需要从数据库读取并根据程序动态生成信息时，才使用 Java 脚本控制。

从表面上看，JSP 页面已经不再需要 Java 类，似乎完全脱离了 Java 面向对象的特征。事实上，JSP 是 Servlet 的一种特殊形式，每个 JSP 页面就是一个 Servlet 实例——JSP 页面由系统编译成 Servlet，Servlet 再负责响应用户请求。JSP 其实也是 Servlet 的一种简化，使用 JSP 时，其实还是使用 Servlet，因为 Web 应用中的每个 JSP 页面都会由 Servlet 容器生成对应的 Servlet。对于 Tomcat 而言，JSP 页面生成的 Servlet 放在 work 路径对应的 Web 应用下。

看下面一个简单的 JSP 页面：

```
<!-- 表明此为一个 JSP 页面 -->
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>第一个 JSP 页面</TITLE>
</HEAD>
<BODY>
<!-- 下面是 Java 脚本-->
<%for(int i = 0 ; i < 10; i++) {
    out.println(i);
%>
<br>
<%}>
</BODY>
</HTML>
```

当启动 Tomcat 之后，可以在 Tomcat 的 Catalina\localhost\jsptest\org\apache\jsp 目录下找到如下文件（假如 Web 应用名为 jsptest，上面 JSP 页的名为 test1.jsp）：test1\_jsp.java 和 test1\_jsp.class。这两个文件都是 Tomcat 生成的，Tomcat 根据 JSP 页面生成对应 Servlet 的 Java 文件及 class 文件。

下面是 test1\_jsp.java 文件的源代码，这是一个特殊的 Java 类，是一个 Servlet 类：

```
//JSP 页面经过 Tomcat 编译后默认的包
package org.apache.jsp;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import javax.servlet.jsp.*;
//继承 HttpJspBase 类, 该类其实是个 Servlet 的子类
public final class test1_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent
{
    private static java.util.Vector _jspx_dependants;
    public java.util.List getDependants() {
        return _jspx_dependants;
    }
    //用于响应用户的方法
    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response)
        throws java.io.IOException, ServletException
    {
        //获得页面输出流
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        //获得页面输出流
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;
        //开始生成响应
        try
        {
            _jspxFactory = JspFactory.getDefaultFactory();
            //设置输出的页面格式
            response.setContentType("text/html; charset=gb2312");
            pageContext = _jspxFactory.getPageContext(this, request,
                response, null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            //页面输出流
            out = pageContext.getOut();
            _jspx_out = out;
            //输出流, 开始输出页面文档
            out.write("\r\n");
            //下面输出 HTML 标签
            out.write("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">\r\n");
            out.write("<HTML>\r\n");
            out.write("<HEAD>\r\n");
            out.write("<TITLE>first Jsp</TITLE>\r\n");
            out.write("</HEAD>\r\n");
            out.write("<BODY>\r\n");
            //页面中的循环, 在此处循环输出
            for(int i = 0 ; i < 10; i++)
            {
                out.println(i);
                out.write("\r\n");
                out.write("<br>\r\n");
            }
        }
    }
}
```

```
        out.write("\r\n");
        out.write("</BODY>\r\n");
        out.write("</HTML>\r\n");
        out.write("\r\n");
    }
    catch (Throwable t)
    {
        if (!(t instanceof SkipPageException))
        {
            out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (_jspx_page_context != null) _jspx_page_context.handle
                PageException(t);
        }
    }
    finally
    {
        if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_
page_context);
    }
}
}
```

即使读者不了解上面提供的 Java 代码，也依然不会影响 JSP 页面的编写，因为这都是由 Web 容器负责生成的。图 2.1 显示了 test1.jsp 的执行效果。

根据图 2.1 的执行效果，再次对比 test1.jsp 和 test1\_jsp.java 文件，可得到一个结论：该 JSP 页面中的每个字符都由 test1\_jsp.java 文件的输出流生成。图 2.2 显示了 JSP 页面的工作原理。

根据上面的 JSP 页面工作原理图，可以得到如下四个结论：

- JSP 文件必须在 JSP 服务器内运行。
- JSP 文件必须生成 Servlet 才能执行。
- 每个 JSP 页面的第一个访问者速度很慢，因为必须等待 JSP 编译成 Servlet。

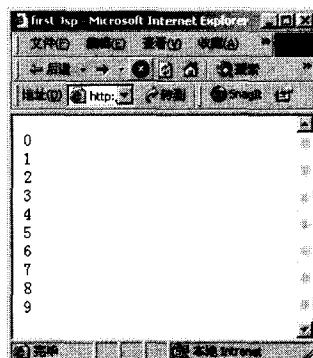


图 2.1 test1.jsp 的执行效果

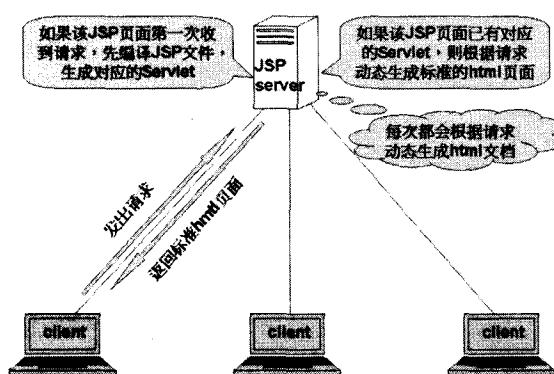


图 2.2 JSP 页面的工作原理

- JSP 页面的访问者无须安装任何客户端，甚至不需要可以运行 Java 的运行环境，因为 JSP 页面输送到客户端的是标准 HTML 页面。

JSP 的出现，大大提高了 Java 动态网站的开发效率，曾一度风靡 Java Web 应用开发者。

## 2.2 JSP 注释

JSP 注释用于表明在程序开发过程中的开发提示，它不会输出到客户端。

JSP 注释的格式如下：

```
<%-- 注释内容 --%>
```

与 JSP 注释形成对比的是 HTML 注释，HTML 注释的格式是

```
<!-- 注释内容 -->
```

看下面的 JSP 页面：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!-- 以下是标准 HTML 部分-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>注释测试</TITLE>
</HEAD>
<BODY>
注释测试
<!-- 增加 JSP 注释-->
<%-- JSP 注释部分 --%>
<!-- 增加 HTML 注释-->
<!-- HTML 注释部分 -->
</BODY>
</HTML>
```

在浏览器中浏览该页面，并查看页面源代码，页面的源代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>注释测试</TITLE>
</HEAD>
<BODY>
注释测试

<!-- HTML 注释部分 -->
</BODY>
</HTML>
```

在上面源代码中可看到，HTML 的注释可以通过源代码查看到，但 JSP 的注释是无法通过源代码查看到的。

## 2.3 JSP 声明

JSP 声明用于声明变量和方法。在 JSP 声明中声明方法看起来很特别，似乎没有类，只有方法定义，而方法又脱离类独立存在。

JSP 声明的格式如下：

```
<%! 声明部分 %>
```

看下面页面的源代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>声明测试</TITLE>
</HEAD>
<!-- 下面是 JSP 声明部分-->
<%!
//声明一个整形变量
public int count;
//声明一个方法
public String info()
{
    return "hello";
}
%>
<BODY>
<%
//将 count 的值加 1 后输出
out.println(count++);
%>
<br>
<%
//输出 info() 方法的返回值
out.println(info());
%>
</BODY>
</HTML>
```

在浏览器中测试该页面时，可以看到正常输出了 count 值，每刷新一次，count 值将加 1，同时也可以看到正常输出了 info 方法的返回值。

打开多个浏览器，甚至可以在不同的机器上打开浏览器刷新该页面，发现每个客户端 count 值是完全连续的，所有的客户端共享了同一个 count 变量。这是因为：JSP 页面会编译成一个 Servlet 类，每个 Servlet 在容器中只有一个实例；而在 JSP 中声明的变量是类的成员变量，成员变量只在创建实例时初始化，该变量的值将一直保存，直到实例销毁。

值得注意的是，info()的值也可正常输出。因为 JSP 声明的方法其实是在 JSP 编译生成的 Servlet 类的方法——Java 里的方法是不能独立存在的，即使在 JSP 页面中也不行。

注意：JSP 声明中独立存在的方法，只是一种假象。

## 2.4 JSP 表达式

JSP 提供了一种输出表达式值的简单方法，输出表达式值的格式如下：

```
<%=表达式%>
```

看下面的 JSP 页面，该页面使用输出表达式的方式输出变量和方法返回值：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>声明测试</TITLE>
</HEAD>
<%!
public int count;

public String info()
{
    return "hello";
}
%>
<BODY>
<!-- 使用表达式输出变量值-->
<%=count++%>
<br>
<!-- 使用表达式输出方法返回值-->
<%=info()%>
</BODY>
</HTML>
```

该页面的执行效果与前一个页面的执行效果没有区别。

## 2.5 JSP 脚本

JSP 脚本的应用非常广泛，可通过 Java 代码镶嵌在 HTML 代码中，即使用 JSP 脚本。因此，所有能在 Java 程序中执行的代码，都可以通过 JSP 脚本执行。

看下面的代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>小脚本测试</TITLE>
</HEAD>
<BODY>
<table bgcolor="9999dd" border="1">
<!-- Java 脚本，这些脚本会对 HTML 的标签产生作用-->
<%
for(int i = 0 ; i < 10 ; i++)
{
%>
```

```
<!-- 上面的循环将使<tr>标签循环-->
<tr>
<td>循环值:</td>
<td><%=i%></td>
</tr>
<%
}
%
<table>
</BODY>
</HTML>
```

对于上面的 JSP 页面，其简单的循环将导致<tr>标签循环 10 次，即生成一个 10 行的表格，并在表格中输出表达式值。

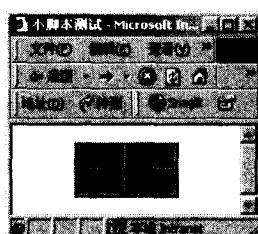
在浏览器中浏览该页面，浏览器中页面的源代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>小脚本测试</TITLE>
</HEAD>
<BODY>
<!-- JSP 页面原有的 table 标签-->
<table bgcolor="9999dd" border="1">
<!-- 下面的 10 行 tr 标签都是由 JSP 脚本控制生成的。-->
<tr>
<td>循环值:</td>
<td>0</td>
</tr>
<tr>
<td>循环值:</td>
<td>1</td>
</tr>
<tr>
<td>循环值:</td>
<td>2</td>
</tr>
<tr>
<td>循环值:</td>
<td>3</td>
</tr>
<tr>
<td>循环值:</td>
<td>4</td>
</tr>
<tr>
<td>循环值:</td>
<td>5</td>
</tr>
<tr>
<td>循环值:</td>
<td>6</td>
</tr>
<tr>
<td>循环值:</td>
<td>7</td>
</tr>
```

```
<tr>
<td>循环值:</td>
<td>8</td>
</tr>
<tr>
<td>循环值:</td>
<td>9</td>
</tr>
</table>
</BODY>
</HTML>
```

在 JSP 脚本中，可以完成 Java 代码中的任何功能，包括连接数据库和执行数据库操作。看下面的 JSP 页面源代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>小脚本测试</TITLE>
</HEAD>
<BODY>
<%
//注册数据库驱动
Class.forName("com.mysql.jdbc.Driver");
//获取数据库连接
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/
j2ee","root","32147");
//创建 Statement
Statement stmt = conn.createStatement();
//执行查询
ResultSet rs = stmt.executeQuery("select * from userinf");
%>
<table bgcolor="9999dd" border="1">
<%
//遍历结果集
while(rs.next())
{
%
<tr>
<!-- 输出结果集-->
<td><%=rs.getString(2)%></td>
<td><%=rs.getString(3)%></td>
</tr>
<%
}
</table>
</BODY>
</HTML>
```



上面的 JSP 页面在 Java 脚本中执行数据库查询，并通过循环将查询结果输出到 JSP 页面。此页面的执行效果如图 2.3 所示。

图 2.3 小脚本连接数据库的效果

## 2.6 JSP 的三个编译指令

JSP 的编译指令是通知 JSP 引擎的消息，它不直接生成输出。编译指令都有默认值，因此开发人员无须为每个指令设置值。

常见的编译指令有三个。

- **page**: 该指令是针对当前页面的指令。
- **include**: 用于指定如何包含另一个页面。
- **taglib**: 用于定义和访问自定义标签。

编译指令的格式如下：

```
<%@ 编译指令名 属性名="属性值" ... %>
```

下面主要介绍 page 和 include 指令，关于 taglib 指令，将在自定义标签库处详细讲解。

### 2.6.1 page 指令

page 指令，通常位于 JSP 页面的顶端，对同一个页面可以有多个 page 指令。page 指令的语法格式如下：

```
<%@page  
[language="Java"]  
[extends="package.class"]  
[import="package.class | package.*", ...]  
[session="true | false"]  
[buffer="none | 8kb | size kb"]  
[autoFlush="true | false"]  
[isThreadSafe="true | false"]  
[info="text"]  
[errorPage="relativeURL"]  
[contentType="mimeType[;charset=characterSet]" | "text/html;charSet=ISO8859-1"]  
[isErrorHandler="true | false"]  
%>
```

下面依次介绍 page 的各个属性。

- **language**: 声明当前 JSP 页面使用的脚本语言的种类，因为页面是 JSP 页面，该属性的值通常都是 java。
- **extends**: 确定 JSP 程序编译时所产生的 Java 类，需要继承的父类，或者需要实现的接口的全限定类名。
- **import**: 用来导入包，下面几个包是默认自动导入的，不需要显式导入。默认导入的包有：java.lang.\*；javax.servlet.\*；javax.servlet.jsp.\*；javax.servlet.http.\*。
- **session**: 设定这个 JSP 页面是否需要 HTTP session。
- **buffer**: 指定输出缓冲区的大小。输出缓冲区的 JSP 内部对象：out 用于缓存 JSP 页面对客户浏览器的输出，默认值为 8kb，可以设置为 none，也可以设置为其他

的值，单位为 kb。

- autoFlush: 当输出缓冲区即将溢出时，是否需要强制输出缓冲区的内容。设置为 true 时为正常输出；如果设置为 false，会在 buffer 溢出时产生一个异常。
- info: 设置该 JSP 程序的信息，也可以看做其说明，可以通过 Servlet. getServletInfo() 方法获取该值。如果在 JSP 页面中，可直接调用 getServletInfo()方法获取该值，因为 JSP 页面的实质就是 Servlet。
- errorPage: 指定错误处理页面。如果本程序产生了异常或者错误，而该 JSP 页面没有对应的处理代码，则会自动调用该指令所指定的 JSP 页面。使用 JSP 页面时，可以不处理异常，即使是 checked 异常。
- isErrorPage: 设置本 JSP 页面是否为错误处理程序。如果该页面本身已是错误处理页面，则无须使用 errorPage 属性。
- contentType: 用于设定生成网页的文件格式和编码方式，即 MIME 类型和页面字符集类型，默认的 MIME 类型是 text/html；默认的字符集为 ISO-8859-1。

从 2.5 节中执行数据库操作的 JSP 页面中可以看出，在 scriptlet2.jsp 页面的头部，使用了两个 page 指令：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page import="java.sql.*"%>
```

其中第二条指令用于导入本页面中使用的类，如果没有通过 page 指令的 import 指令导入这些类，则需在脚本中使用全限定类名——即必须带包名。可见，此处的 import 属性类似于 Java 程序中的 import 关键字的作用。

如果删除第二条 page 指令，则执行效果如图 2.4 所示。

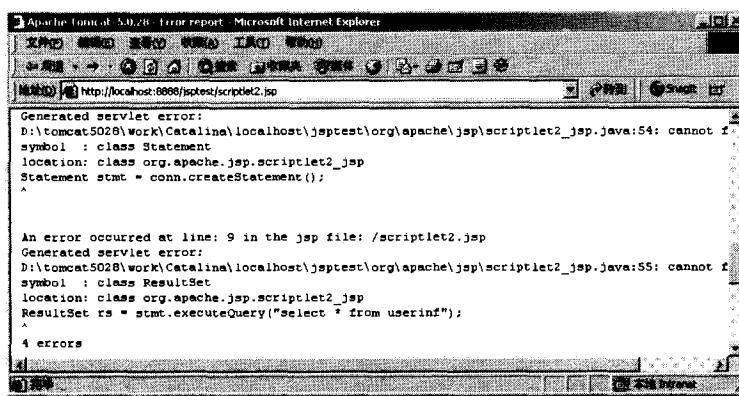


图 2.4 不使用 import 属性的出错效果

看下面的 JSP 页面，该页面使用 page 指令的 info 方法指定了 JSP 页面的描述信息，又使用 getServletInfo()方法输出该描述信息。

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!-- 指定 info 信息-->
<%@ page info="this is a jsp"%>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>测试 page 指令的 info 属性</TITLE>
</HEAD>
<BODY>
<!-- 输出 info 信息-->
<%=getServletInfo()%>
<table>
</BODY>
</HTML>
```

该页面的执行效果如图 2.5 所示。

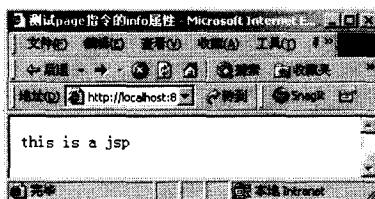


图 2.5 page 指令的 info 属性效果

`errorPage` 属性的实质是 JSP 的一种异常处理机制，JSP 不要求强制处理异常，即使该异常是 `checked` 异常。如果 JSP 页面在运行中抛出未处理的异常，系统将自动跳转到 `errorPage` 属性指定的页面；如果 `errorPage` 没有指定错误页面，系统则将异常信息呈现给客户端浏览器——这是所有的开发者都不愿意见到的场景。

看下面的 JSP 页面，该页面使用了 `page` 指令的 `errorPage` 属性，该属性指定了对页面发生异常时的异常处理页面。

```
<%@ page contentType="text/html; charset=gb2312" language="java" errorPage=
"error.jsp"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>测试 page 指令的 errorPage 属性</TITLE>
</HEAD>
<BODY>
<%
//下面代码将出现运行时异常
int a = 6;
int b = 0;
int c = a / b;
%>
<table>
</BODY>
</HTML>
```

下面是 `error.jsp` 页面，该页面本身是错误处理页面，因此将 `isErrorPage` 设置成 `true`。

```
<%@ page contentType="text/html; charset=gb2312" language="java" isErrorPage=
"true"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
```

```

<HEAD>
<TITLE>出错页面</TITLE>
</HEAD>
<BODY>
<!-- 提醒客户端系统出现异常 -->
系统出现异常<br>
</BODY>
</HTML>

```

在浏览器中浏览前一个页面的效果如图 2.6 所示。

如果将前一个页面中 page 指令的 errorPage 属性删除，再次通过浏览器浏览该页面，执行效果如图 2.7 所示。

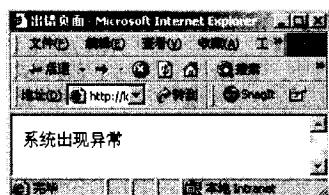


图 2.6 errorPage 属性控制异常处理效果

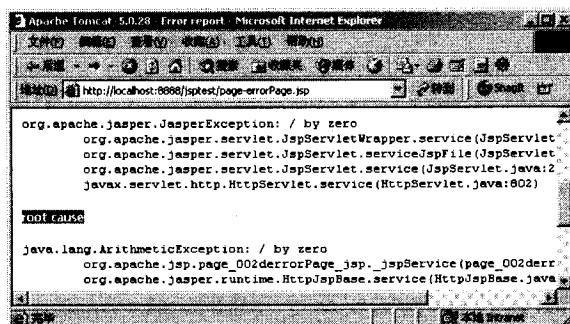


图 2.7 没有 errorPage 属性控制异常处理的效果

可见，使用 errorPage 属性控制异常处理的效果在表现形式上要好得多。

## 2.6.2 include 指令

使用 include 指令，可以将一个外部文件嵌入到当前 JSP 文件中，同时解析这个页面中的 JSP 语句（如果有的话）。这是个静态的 include 语句，不会检查所包含 JSP 页面的变化。

include 既可以包含静态的文本，也可以包含动态的 JSP 页面。静态的编译指令 include，是将被包含的页面加入进来，生成一个完整的页面。

include 编译指令的语法：

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

如果被嵌入的文件经常需要改变，建议使用<jsp:include>操作指令，因为它是动态的 include 语句。

看下面的页面：

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>include 测试</TITLE>
</HEAD>

```

```
<BODY>
<%@include file="scriptlet1.jsp"%>
</BODY>

</HTML>
```

该页面的执行效果与 scriptlet1.jsp 的执行效果相同, 该页面的执行效果如图 2.8 所示。

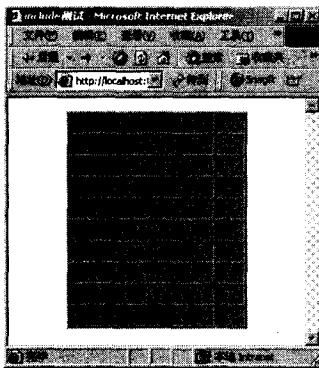


图 2.8 include 执行效果

查看 Tomcat 的 work\localhost\jsptest\org\apache\jsp 路径下的 include\_jsp.java 文件, 从 include.jsp 编译后的源代码可看到, include.jsp 页面已经完全将 scriptlet1.jsp 的代码融入进来, 下面是 include\_jsp.java 文件的片段:

```
//输出流生成表格
out.write("<table bgcolor=\"9999dd\" border=\"1\" align=\"center\" width=\"200\">\r\n");
//循环输出表格的行
for(int i = 0 ; i < 10 ; i++)
{
    out.write("\r\n");
    out.write("<tr>\r\n");
    out.write("<td>循环值</td>\r\n");
    out.write("<td>");
    out.print(i);
    out.write("</td>\r\n");
    out.write("</tr>\r\n");
}
out.write("\r\n");
out.write("<table>\r\n");
```

这就是静态包含意义: 包含页面在编译时已经完全包含了被包含页面的代码。

## 2.7 JSP 的 7 个动作指令

动作指令与编译指令不同, 编译指令是通知 Servlet 引擎的处理消息, 而动作指令只是运行时的脚本动作。编译指令在将 JSP 编译成 Servlet 时起作用; 处理指令通常可替换成 Java 脚本, 是 JSP 脚本的标准化写法。

JSP 动作指令主要有如下 7 个。

- **jsp:forward:** 执行页面转向，将请求的处理转发到下一个页面。
- **jsp:param:** 用于传递参数，必须与其他支持参数的标签一起使用。
- **jsp:include:** 用于动态引入一个 JSP 页面。
- **jsp:plugin:** 用于下载 JavaBean 或 Applet 到客户端执行。
- **jsp:useBean:** 使用 JavaBean。
- **jsp:setProperty:** 修改 JavaBean 实例的属性值。
- **jsp:getProperty:** 获取 JavaBean 实例的属性值。

下面依次讲解这些动作指令。

## 2.7.1 forward 指令

forward 指令用于将页面响应控制转发给另外的页面。既可以转发给静态的 HTML 页面，也可以转发到动态的 JSP 页面，或者转发到容器中的 Servlet。

JSP 的 forward 指令的格式如下：

对于 JSP 1.0，使用如下语法：

```
<jsp:forward page="<%{relativeURL | <%=expression%>}"/>
```

对于 JSP 1.1 以上，可使用如下语法：

```
<jsp:forward page="<%{relativeURL | <%=expression%>}">
  {<jsp:param.../>}
</jsp:forward>
```

第二种语法用于在转发时增加额外的请求参数。增加的请求参数的值可以通过 HttpServletRequest 类的 getParameter 方法获取。

看下面 JSP 页面：

```
<jsp:forward page="forward-result.jsp">
  <jsp:param name="age" value="29"/>
</jsp:forward>
```

这个 JSP 页面非常简单，它不作任何处理，仅将所有的客户端请求转发到 forward-result.jsp 页面，在转发时，增加了一个请求参数：参数名为 age，参数值为 29。在 forward-result.jsp 页面中，使用 request 内置对象（request 内置对象是 HttpServletRequest 的实例，关于 request 的详细信息参看下一节）来获取增加的请求参数值。

forward-result.jsp 的代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>forward 结果页</TITLE>
</HEAD>
<BODY>
<!-- 使用 request 内置对象获取 age 参数的值-->
```

```
<%=request.getParameter("age")%>
</BODY>
</HTML>
```

在浏览器中访问 `jsp-forward.jsp` 页面的执行效果如图 2.9 所示。

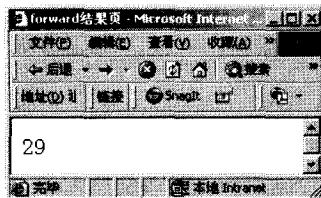


图 2.9 `jsp:forward` 增加的请求参数

从页面的执行效果可看出，`forward` 指令增加的参数可以在转发的新页面中访问到。

`jsp:forward` 指令转发请求时，客户端的请求参数不会丢失，看下面表单提交页面的例子，该页面没有任何动态的内容，是一个静态的 HTML 页面，仅将表单域的值提交到 `jsp-forward` 页。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>提交</TITLE>
</HEAD>
<BODY>
<!-- 表单提交页面 -->
<form id="login" method="post" action="jsp-forward.jsp">
<INPUT TYPE="text" NAME="username">
<INPUT TYPE="submit" value="login">
</FORM>
</BODY>
</HTML>
```

修改 `forward-result.jsp` 页，增加输出表单参数的代码，修改后 `forward-result.jsp` 页面的代码如下：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>forward 结果页</TITLE>
</HEAD>
<BODY>
<!-- 输出 forward 指令增加的参数值-->
<%=request.getParameter("age")%>
<!-- 输出前一个页面接受到的请求参数值-->
<%=request.getParameter("username")%>
</BODY>
</HTML>
```

在静态表单提交页面中的文本框中输入任意字符串，可看到 `forward-result.jsp` 页面中不仅可以输出 `forward` 指令增加的参数，还可以看到表单域的参数值。

`forward-result.jsp` 页面的执行效果如图 2.10 所示。

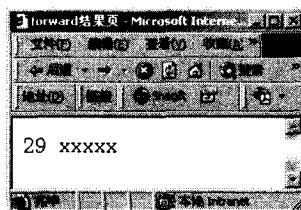


图 2.10 forward-result.jsp 输出两个参数值

通过该页面的执行，可看到表单提交的数据被 `forward` 之后，这些请求的信息依然可以获取到。即 `forward` 指令不会丢失请求数据。

## 2.7.2 include 指令

`include` 指令是一个动态 `include` 指令，也用于导入某个页面，它的导入会每次检查被导入页面的改变。下面是 `include` 指令的使用格式：

```
<jsp:include page="{relativeURL | <%=expression%>}" flush="true"/>
```

或者

```
<jsp:include page="{relativeURL | <%=expression%>}" flush="true">
  <jsp:param name="parameterName" value="patameterValue"/>
</jsp:include>
```

`flush` 属性用于指定输出缓存是否转移到被导入文件中。如果指定为 `true`，则包含在被导入文件中；如果指定为 `false`，则包含在原文件中。对于 JSP 1.1 旧版本，只能设置为 `false`。

对于第二种格式，可用于在导入页面中加入参数值。

看下面的 `jsp0-include.jsp` 页面的代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>jsp-include 测试</TITLE>
</HEAD>
<BODY>
<!-- 使用动态 include 指令导入页面-->
<jsp:include page="scriptlet1.jsp" />
</BODY>
</HTML>
```

此时，页面的执行效果与使用静态 `include` 导入的效果并没有什么不同。通过查看 `jsp0-include.jsp` 页面生成 Servlet 的源代码，可以看到如下片段：

```
//使用页面输出流，生成 HTML 标签内容
out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\
\">\r\n");
out.write("<HTML>\r\n");
out.write("<HEAD>\r\n");
```

```

out.write("<TITLE>jsp-include 测试</TITLE>\r\n");
out.write("</HEAD>\r\n");
out.write("<BODY>\r\n");
//动态导入，直接引入 scriptlet1.jsp 页面
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response,
"scriptlet1.jsp", out);
out.write("\r\n");
out.write("</BODY>\r\n");
out.write("</HTML>\r\n");
out.write("\r\n");

```

对比这两个片段，可看到静态导入和动态导入的区别：静态导入是将被导入页面的代码完全插入，两个页面生成一个整体的 Servlet；而动态导入则在 Servlet 中使用动态导入，从而将页面引入。

再看下面的 JSP 页面：

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>jsp-include 测试</TITLE>
</HEAD>
<BODY>
<jsp:include page="forward-result.jsp" >
    <jsp:param name="age" value="32"/>
</jsp:include>
</BODY>
</HTML>

```

在上面的 JSP 页面中，同样使用 `jsp:include` 指令导入页面。而且在 `jsp:include` 指令中还使用 `param` 指令传入参数，该参数可以在 `forward-result.jsp` 页面中使用 `request` 对象获取。

`forward-result.jsp` 前面已经给出，此处不再赘述。页面执行的效果如图 2.11 所示。

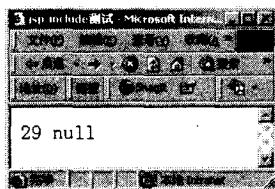


图 2.11 包含参数的 `include` 效果

### 2.7.3 `useBean`, `setProperty`, `getProperty` 指令

这三个都是与 JavaBean 相关的指令，其中第一个指令用于在 JSP 页面中初始化一个 Java 实例；`setProperty` 指令用于修改 JavaBean 实例的属性；`getProperty` 用于获取 JavaBean 实例的属性。

`useBean` 的语法格式如下：

```
<jsp:useBean id="name" class="classname" scope="page | request | session | application" />
```

id 属性是 JavaBean 的实例名, class 属性确定 JavaBean 的实现类。

其中 scope 属性用于确定 JavaBean 实例的生存范围, 该范围有以下四个值。

- page: 该 JavaBean 实例仅在该页面有效。
- request: 该 JavaBean 实例在本次请求有效。
- session: 该 JavaBean 实例在本次 session 内有效。
- application: 该 JavaBean 实例在本应用内一直有效。

setProperty 的语法格式如下:

```
<jsp:setProperty name="BeanName" property="propertyName" value="value" />
```

其中 name 属性确定需要设定 JavaBean 的实例名; property 属性确定需要设置的属性名; value 属性则确定需要设置的属性值。

getProperty 的语法格式如下:

```
<jsp:getProperty name="BeanName" property="propertyName" />
```

其中 name 属性确定需要输出的 JavaBean 的实例名; property 属性确定需要输出的属性名。

看下面的 JSP 页面代码:

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>javabean</TITLE>
</HEAD>
<BODY>
<!-- 实例化 JavaBean 实例, 实现类为 lee.Person, 该实例的实例名为 p1--&gt;
&lt;jsp:useBean id="p1" class="lee.Person" scope="page"/&gt;
<!-- 设置 p1 的 name 属性值--&gt;
&lt;jsp:setProperty name="p1" property="name" value="wawa"/&gt;
<!-- 设置 p1 的 age 属性值--&gt;
&lt;jsp:setProperty name="p1" property="age" value="23"/&gt;
<!-- 输出 p1 的 name 属性值--&gt;
&lt;jsp:getProperty name="p1" property="name"/&gt;&lt;br&gt;
<!-- 输出 p1 的 age 属性值--&gt;
&lt;jsp:getProperty name="p1" property="age"/&gt;
&lt;/BODY&gt;
&lt;/HTML&gt;</pre>
```

对于上面的 JSP 页面中的 setProperty 和 getProperty 标签, 都包含了 name 和 property 属性。property 属性确定需要访问的属性名, 对于这些属性名, 可以无须对应的 JavaBean 中有同名的属性, 但必须有对应的 getter 和 setter 方法。

即如果 property 属性确定了 name 属性, 则要求 JavaBean 中必须有 setName 和 getName 方法。事实上, 当使用 setProperty 和 getProperty 标签时, 系统将自动调用 setName 和 getName 方法来访问 Person 实例的属性。

下面是 Person 类的源代码:

```

public class Person
{
    private String name;
    private int age;
    //name 属性对应的 setter 方法
    public void setName(String name)
    {
        this.name = name;
    }
    //age 属性对应的 setter 方法
    public void setAge(int age)
    {
        this.age = age;
    }
    //name 属性对应的 getter 方法
    public String getName()
    {
        return name;
    }
    //age 属性对应的 getter 方法
    public int getAge()
    {
        return age;
    }
}

```

该页面的执行效果如图 2.12 所示。

对于上面三个标签完全可以不使用，将 javabean.jsp  
修改成如下代码，其内部的执行是完全一样的：

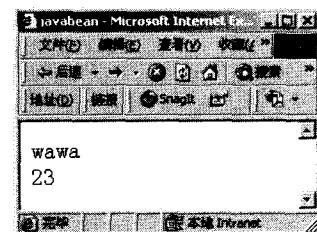


图 2.12 JavaBean 页面的执行效果

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>javabean</TITLE>
</HEAD>
<BODY>
<%
<!-- 实例化 JavaBean 实例，实现类为 lee.Person，该实例的实例名为 p1--&gt;
Person p1 = new Person();
<!-- 设置 p1 的 name 属性值--&gt;
p1.setName("wawa");
<!-- 设置 p1 的 age 属性值--&gt;
p1.setAge(23);
%
&lt;!-- 输出 p1 的 name 属性值--&gt;
&lt;%=p1.getName()%&gt;&lt;br&gt;
&lt;!-- 输出 p1 的 age 属性值--&gt;
&lt;%=p1.getAge()%&gt;
&lt;/BODY&gt;
&lt;/HTML&gt;
</pre>

```

如果 useBean 的 scope 属性值是 page，则完全可以使用 Java 脚本来代替这几个标签。但如果指定的 scope 属性值不是 page，则还需要将该 JavaBean 实例放入特定的生存范围，如下面代码片段所示：

```
//将 p1 放入 request 的生存范围  
request.setAttribute("p1",p1);  
//将 p1 放入 session 的生存范围  
session.setAttribute("p1",p1);  
//将 p1 放入 application 的生存范围  
application.setAttribute("p1",p1);
```

## 2.7.4 plugin 指令

plugin 指令主要用于下载服务器端的 JavaBean 或 Applet 到客户端执行。由于程序在客户端执行，因此客户端必须安装虚拟机。

plugin 的语法格式如下：

```
<jsp:plugin type="bean | applet"  
    code="className"  
    codebase="classFileDirectoryName"  
    [name="instanceName"]  
    [archive="URLtoArchive"]  
    [align="bottom | top | middle | left | right"]  
    [height="displayPixels"]  
    [width="displayPixels"]  
    [hspace="leftRightPixels"]  
    [vspace="topBottomPixels"]  
    [jreversion="JREVersionNumber | 1.2"]  
    [nspluginurl="URLToPlugin"]  
    [iepluginurl="URLToPlugin"]>  
    [<jsp:params>  
        [jsp:param name="parameterName" value="parameterValue"/>]  
    </jsp:params>]  
    [<jsp:fallback>text message for user that can no see the plugin  
    </jsp:fallback> ]  
</jsp:plugin>
```

关于这些属性的说明如下。

- **type:** 指定被执行的 Java 程序的类型。
- **code:** 指定被执行的文件名，该属性值必须以“.class”扩展名结尾。
- **codebase:** 指定被执行文件所在的目录。
- **name:** 给该程序起一个名字用来标识该程序。
- **archive:** 指向一些要预先载入的将要使用到的类的路径。
- **hspace,vspace:** 显示左右，上下的留白。
- **jreversion:** 能正确运行该程序必需的 JRE 版本，默认值是 1.2。
- **nsplugin,ieplugin:** Netscape Navigator, Internet Explorer 下载运行所需 JRE 的地址。
- **<jsp:fallback>** 指令：当不能正确显示该 Applet 时，代替显示的提示信息。

看下面的 JSP 页面代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
<HTML>  
<HEAD>  
<TITLE>jsp:plugin 测试</TITLE>
```

```

</HEAD>
<BODY>
<!-- 使用 Applet-->
<jsp:plugin type="applet" code="Hello.class">
    <!-- 为 Applet 传入参数-->
    <jsp:params>
        <jsp:param name="hello" value="Java 世界"/>
    </jsp:params>
    <!-- 不能显示 Applet 时的提示信息-->
    <jsp:fallback>
        <p>不能下载 jre 插件</p>
    </jsp:fallback>
</jsp:plugin>
</BODY>
</HTML>

```

该 Applet 的显示使用 `jsp:param` 指令传入了一个参数，该参数可以使用 `getParameter` 方法获取。

其中 `Hello.class` 是个基本的 Applet，该 Applet 的代码如下：

```

//Applet 继承 java.applet.Applet 类
public class Hello extends Applet
{
    //重写 paint 方法，该方法将在 Applet 上绘图
    public void paint(Graphics g)
    {
        //先绘出字符串，字符串通过 getParameter 方法获取
        g.drawString(getParameter("hello"), 20, 30);
        //设置颜色
        g.setColor(new Color(255, 200, 200));
        //画出一个矩形
        g.fillRect(50, 60, 200, 150);
    }
}

```

在浏览器中执行该页面，效果如图 2.13 所示。

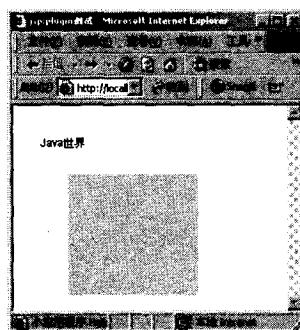


图 2.13 `jsp:plugin` 执行 Applet 的效果

## 2.7.5 param 指令

`param` 指令用于设置参数值，这个指令本身不能单独使用，因为单独的 `Param` 指令

没有实际意义。`param` 指令可以与以下三个指令结合使用：

- `jsp:include`
- `jsp:forward`
- `jsp:plugin`

当与 `include` 指令结合使用时，`param` 指令用于将参数值传入被导入的页面；当与 `forward` 指令结合使用时，`param` 指令用于将参数值传入被转向的页面；当与 `plugin` 指令结合使用时，则用于将参数传入页面中的 JavaBean 实例或 Applet 实例。

`param` 指令的语法格式如下：

```
<jsp:param name="paramName" value="paramValue" />
```

关于 `param` 的具体使用，请参考前面的示例。

## 2.8 JSP 的 9 个内置对象

JSP 页面中包含 9 个内置对象，这 9 个内置对象都是 Servlet API 的类或者接口的实例，只是 JSP 规范将它们完成了默认初始化，即它们已经是对象，可以直接使用。这 9 个内置对象依次如下。

- `application`: `javax.servlet.ServletContext` 的实例，该实例代表 JSP 所属的 Web 应用本身，可用于 JSP 页面，或者 Servlet 之间交换信息。常用的方法有 `getAttribute(String attName)`, `setAttribute(String attName, String attValue)` 和 `getInitParameter(String paramName)` 等。
- `config`: `javax.servlet.ServletConfig` 的实例，该实例代表该 JSP 的配置信息。常用的方法有 `getInitParameter(String paramName)` 及 `getInitParameterNames()` 等方法。事实上，JSP 页面通常无须配置，也就不存在配置信息。因此，该对象更多地在 Servlet 中有效。
- `exception`: `java.lang.Throwable` 的实例，该实例代表其他页面中的异常和错误。只有当页面是错误处理页面，即编译指令 `page` 的 `isErrorPage` 属性为 `true` 时，该对象才可以使用。常用的方法有 `getMessage()` 和 `printStackTrace()` 等。
- `out`: `javax.servlet.jsp.JspWriter` 的实例，该实例代表 JSP 页面的输出流，用于输出内容，形成 HTML 页面。
- `page`: 代表该页面本身，通常没有太大用处。也就是 Servlet 中的 `this`，其类型就是生成的 Servlet。
- `pageContext`: `javax.servlet.jsp.PageContext` 的实例，该对象代表该 JSP 页面上下文，使用该对象可以访问页面中的共享数据。常用的方法有 `getServletContext()` 和 `getServletConfig()` 等。
- `request`: `javax.servlet.http.HttpServletRequest` 的实例，该对象封装了一次请求，客户端的请求参数都被封装在该对象里。这是一个常用的对象，获取客户端请求参数必须使用该对象。常用的方法有 `getParameter(String paramName)`, `getParameter`

`Values(String paramName), setAttribute(String attributeName, Object attributeValue),  
getAttribute(String attributeName)和 setCharacterEncoding(String env)`等。

- `response`: `javax.servlet.http.HttpServletResponse` 的实例，代表服务器对客户端的响应。通常，也很少使用该对象直接响应，输出响应使用 `out` 对象，而 `response` 对象常用于重定向。常用的方法有 `sendRedirect(java.lang.String location)` 等。
- `session`: `javax.servlet.http.HttpSession` 的实例，该对象代表一次会话。从客户端浏览器与站点建立连接起，开始会话，直到关闭浏览器时结束会话。常用的方法有：`getAttribute(String attName), setAttribute(String attName, String attValue)` 等。

## 2.8.1 application 对象

该对象代表 Web 应用本身，整个 Web 应用共享同一个 `application` 对象，该对象主要用于在多个 JSP 页面或 Servlet 之间共享变量。`application` 通过 `setAttribute` 方法将一个值放入某个属性，该属性的值对整个 Web 应用有效，因此 Web 应用的每个 JSP 页面或 Servlet 都可以访问该属性，访问属性的方法为 `getAttribute`。

看下面的页面，该页面仅仅声明了一个整型变量，每次刷新该页面时，该变量值加 1，然后将该变量的值放入 `application` 内。下面是页面的代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>application 测试</TITLE>
</HEAD>
<BODY>
<!-- JSP 声明-->
<%
int i;
%
<!-- 将 i 值自加后放入 application 的变量内-->
<%
application.setAttribute("counter",String.valueOf(++i));
%
<!-- 输出 i 值-->
<%=i%>
</BODY>
</HTML>
```

这个页面的效果很简单，每次刷新该页面时，`i` 值都会自加，并重新修改 `application` 中 `counter` 的值，即每次 `application` 中的 `counter` 都会加 1。

再看下面的 JSP 页面，该页面没有直接访问 `application` 的 `counter` 值：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>application 测试</TITLE>
</HEAD>
<BODY>
```

```
<!-- 直接输出 application 变量值-->
<%=application.getAttribute("counter")%>
</BODY>
</HTML>
```

根据上面的介绍可知，application 中的变量，对于整个 Web 应用中的 JSP 及 Servlet 都是共享的。

在浏览器的地址栏中访问第一个 put-application.jsp 页面，经多次刷新后，看到如图 2.14 所示的页面。

访问 get-application.jsp 页面，可直接看到如图 2.15 所示的效果。

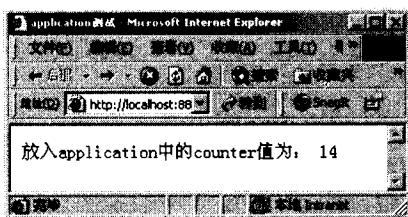


图 2.14 将变量值放入 application 中

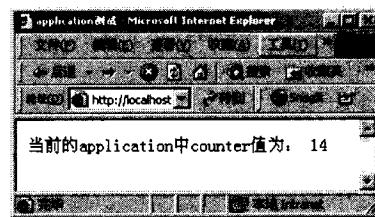


图 2.15 从 application 中取出变量值

由此可看出，当这两个页面在同一个 Web 应用中时，只要 put-application.jsp 向 application 中存入了变量，那么 get-application.jsp 页面就可以从中取出该变量值。

**注意：**application 不仅可用于两个 JSP 页面之间共享数据，还可以用于 Servlet 和 JSP 之间共享数据。

看下面的 Servlet 代码：

```
public class GetApplication extends HttpServlet
{
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException
    {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>");
        out.println("测试 application");
        out.println("</title></head><body>");
        ServletContext sc = getServletConfig().getServletContext();
        out.print("application 中当前的 counter 值为:");
        out.println(sc.getAttribute("counter"));
        out.println("</body></html>");
    }
}
```

该 Servlet 从 application 中取出 counter 的值，并输出到页面，将 Servlet 部署在 Web 应用中。关于 Servlet 的使用，请参考 2.9 节。

在浏览器中访问 Servlet，出现如图 2.16 所示页面。

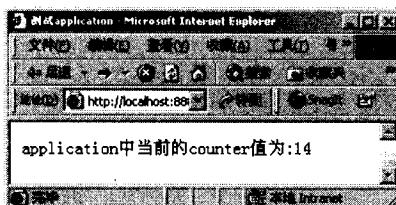


图 2.16 Servlet 访问 application 变量

`application` 还有一个重要作用：可用于加载 Web 应用的配置参数。看如下 JSP 页面，该页面访问数据库，但访问数据库所使用的驱动、url、用户名及密码都在配置文件中给出：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>application 测试</TITLE>
</HEAD>
<BODY>
<%
//从配置参数中获取驱动
String driver = application.getInitParameter("driver");
//从配置参数中获取数据库 url
String url = application.getInitParameter("url");
//从配置参数中获取用户名
String user = application.getInitParameter("user");
//从配置参数中获取密码
String pass = application.getInitParameter("pass");
//注册驱动
Class.forName(driver);
//获取数据库连接
Connection conn = DriverManager.getConnection(url,user,pass);
//创建 Statement 对象
Statement stmt = conn.createStatement();
//执行查询
ResultSet rs = stmt.executeQuery("select * from userinf");
<%
//遍历结果集
while(rs.next())
{
<%
<tr>
<td><%=rs.getString(2)%></td>
<td><%=rs.getString(3)%></td>
</tr>
<%
}
<%
</table>
</BODY>
</HTML>
```

`application` 使用 `context-param` 元素配置，并从 `web.xml` 文件中获取参数。在 `web.xml`

文件中增加如下片段：

```
<!-- 配置第一个参数：driver-->
<context-param>
    <param-name>driver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
</context-param>
<!-- 配置第二个参数：url-->
<context-param>
    <param-name>url</param-name>
    <param-value>jdbc:mysql://localhost:3306/j2ee</param-value>
</context-param>
<!-- 配置第三个参数：user-->
<context-param>
    <param-name>user</param-name>
    <param-value>root</param-value>
</context-param>
<!-- 配置第四个参数：pass-->
<context-param>
    <param-name>pass</param-name>
    <param-value>32147</param-value>
</context-param>
```

在浏览器中浏览该页面时，可看到数据库连接成功，数据查询完全成功。可见，使用 application 可以访问 Web 应用的配置参数。

**注意：**通过这种方式，可以将一些配置信息放在 web.xml 文件中配置，避免使用硬编码方式写在代码中，从而更好地提高程序解耦。

## 2.8.2 config 对象

config 对象代表当前 JSP 配置信息，但 JSP 页面通常无须配置，因此也就不存在配置信息。该对象在 JSP 页面中非常少用，但在 Servlet 则用处相对较大。因为 Servlet 需要配置在 web.xml 文件中，可以指定配置参数。关于 Servlet 的使用将在 2.9 节介绍。

看如下 JSP 页面代码，该 JSP 代码使用了 config 的一个方法 getServletName()：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>小脚本测试</TITLE>
</HEAD>
<BODY>
<!-- 直接输出 config 的 getServletName 的值-->
<%=config.getServletName()
%>
</BODY>
</HTML>
```

该页面的输出是 JSP 文件。所有的 JSP 页面都有相同的名字：jsp。

### 2.8.3 exception 对象

exception 对象是 Throwable 的实例，代表 JSP 页面产生的错误和异常，是 JSP 页面异常框架的一部分。

在 JSP 页面中无须处理异常，即使该异常是 checked 异常。事实上，JSP 页面包含的所有异常都由错误页面处理了。

看下面的异常处理结构：

```
try
{
    //代码处理段
    ...
}
catch (Exception exception)
{
    //异常处理段
    ...
}
```

这是典型的异常捕捉处理块。在 JSP 页面中，普通的 JSP 页面只执行第一个部分——代码处理段。而出错的页面负责第二个部分——异常处理段。在异常处理段中，可以看到有个异常对象，该对象就是内置对象 exception。

**注意：** exception 对象仅在错误处理页面中才有效。结合前面的异常处理结构，读者可以非常清晰地看出这点。

在 JSP 的异常处理体系中，一个出错页面可以处理多个 JSP 页面的异常。指定的异常处理页面通过 page 指令的 errorPage 属性确定。

看下面的页面：

```
<!-- 通过 errorPage 属性指定异常处理页面-->
<%@ page contentType="text/html; charset=gb2312" language="java" errorPage=
"error.jsp"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>测试 page 指令的 errorPage 属性</TITLE>
</HEAD>
<BODY>
<%
int a = 6;
int b = 0;
int c = a / b;
%>
<table>
</BODY>
</HTML>
```

当该页面出现异常时，error.jsp 页面将负责处理该异常。

**注意：** 将异常处理页面中 page 指令的 isErrorPage 属性应设置为 true。

## 2.8.4 out 对象

out 对象代表一个页面输出流，通常用于在页面上输出变量值及常量。一般在使用输出表达式值的地方，都可以使用 out 对象来达到同样效果。

看下面的 JSP 页面代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>out 测试</TITLE>
</HEAD>
<BODY>
<%
//注册数据库驱动
Class.forName("com.mysql.jdbc.Driver");
//获取数据库连接
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/j2ee","root","32147");
//创建 Statement 对象
Statement stmt = conn.createStatement();
//执行查询，获取 ResultSet 对象
ResultSet rs = stmt.executeQuery("select * from userinf");
%>
<table bgcolor="#9999dd" border="1" align="center">
<%
//遍历结果集
while(rs.next())
{
    //输出表格行
    out.println("<tr>");
    //输出表格列
    out.println("<td>");
    //输出结果集的第二列的值
    out.println(rs.getString(2));
    //关闭表格列
    out.println("</td>");
    //开始表格列
    out.println("<td>");
    //输出结果集的第三列的值
    out.println(rs.getString(3));
    //关闭表格列
    out.println("</td>");
    //关闭表格行
    out.println("</tr>");
}
%>
</table>
</BODY>
</HTML>
```

从 Java 的语法上看，上面的程序更容易理解，out 是个页面输出流，在页面中输出表格及其内容。

注意：所有使用 `out` 的地方，都可以使用表达式输出的方式代替，而且使用这种方式更加简洁。通过 `out` 对象的系统介绍，读者可以更好地理解表达式输出的原理。

## 2.8.5 pageContext 对象

这个对象代表页面上下文，该对象主要用于访问页面共享数据。使用 `pageContext` 可以直接访问 `request`, `session`, `application` 范围的属性，看下面的 JSP 页面：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>pageContext 测试</TITLE>
</HEAD>
<BODY>
<%
//使用 pageContext 设置属性，该属性默认在 page 范围内
pageContext.setAttribute("page", "hello");
//使用 request 设置属性，该属性默认在 request 范围内
request.setAttribute("request", "hello");
//使用 pageContext 将属性设置在 request 范围中
pageContext.setAttribute("request2", "hello" ,pageContext.REQUEST_SCOPE);
//使用 session 将属性设置在 session 范围中
session.setAttribute("session", "hello");
//使用 pageContext 将属性设置在 session 范围中
pageContext.setAttribute("session2", "hello" ,pageContext.SESSION_SCOPE);
//使用 application 将属性设置在 application 范围中
application.setAttribute("app", "hello");
//使用 pageContext 将属性设置在 application 范围中
pageContext.setAttribute("app2", "hello" , pageContext.APPLICATION_SCOPE);
//以此获取各属性所在的范围：
out.println("page 变量所在范围: " + pageContext.getAttributesScope("page") +
"<br>");
out.println("request 变量所在范围: " +pageContext.getAttributesScope("request") +
"<br>");
out.println("request2 变量所在范围: "+pageContext.getAttributesScope("request2") +
"<br>");
out.println("session 变量所在范围: " +pageContext.getAttributesScope("session") +
"<br>");
out.println("session2 变量所在范围: " +pageContext.getAttributesScope("session2") +
"<br>");
out.println("app 变量所在范围: " +pageContext.getAttributesScope("app") + "<br>");
out.println("app2 变量所在范围: " +pageContext.getAttributesScope("app2") +
"<br>");
%>
</BODY>
</HTML>
```

上面的 JSP 页面使用 `pageContext` 对象多次设置属性，在设置属性时，如果没有指定属性存在的范围，则属性默认在 `page` 范围内；如果指定了属性所在的范围，则属性可以被存放在 `application`, `session`, `request` 等范围内。

图 2.17 显示了使用 `pageContext` 访问属性的效果。



图 2.17 pageContext 访问属性的示例

图 2.17 中显示了使用 pageContext 获取各属性所在的范围，其中这些范围获取的都是整型变量，这些整型变量分别对应几个生存范围。

- 1: 对应 page 生存范围。
- 2: 对应 request 生存范围。
- 3: 对应 session 生存范围。
- 4: 对应 application 生存范围。

## 2.8.6 request 对象

request 对象是 JSP 中重要的对象，每个 request 对象封装着一次用户请求，并且所有的请求参数都被封装在 request 对象中。因此 request 对象也是获取客户端请求参数的方法。

request 对象不仅封装了表单域值，还可以封装地址栏传递的参数。因此用户也可在 request 对象中增加请求属性。

下面依次介绍这几种情况。

### 1. 封装表单域值

封装表单域值是最常见的情况，几乎每个网站都会大量使用表单。表单用于收集用户信息，一旦用户提交请求，表单的信息将会提交给对应的处理程序。

看下面的表单页面：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>out 测试</TITLE>
</HEAD>
<BODY>
<FORM id="form1" METHOD=POST ACTION="request1.jsp">
用户名: <br>
<INPUT TYPE="text" NAME="username"><hr>
性别: <br>
男: <INPUT TYPE="radio" NAME="gender" value="男">
女: <INPUT TYPE="radio" NAME="gender" value="女"><hr>
喜欢的颜色: <br>
```

```

红: <INPUT TYPE="checkbox" NAME="color" value="红">
绿: <INPUT TYPE="checkbox" NAME="color" value="绿">
蓝: <INPUT TYPE="checkbox" NAME="color" value="蓝"><hr>
来自的国家: <br>
<SELECT NAME="country">
    <option value="中国">中国</option>
    <option value="美国">美国</option>
    <option value="俄罗斯">俄罗斯</option>
</SELECT><hr>
<INPUT TYPE="submit" value="提交">
<INPUT TYPE="reset" value="重置">
</FORM>
</BODY>
</HTML>

```

这个页面没有动态的 JSP 部分，仅仅包含 1 个文本框、2 个单选框、3 个复选框及 1 个下拉列表框，另外包括【提交】和【重置】2 个按钮。页面的执行效果如图 2.18 所示。



图 2.18 表单页

在该页面中输入相应信息后，单击【提交】按钮，表单域的信息被封装成 HttpServletRequest 对象，该对象包含了所有的请求参数，可通过 getParameter 方法获取请求参数的值。

该表单页提交到 request1.jsp，该页面的代码如下：

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>request 测试</TITLE>
</HEAD>
<BODY>
<%
//设置解码方式，对于中文，使用 GBK 解码
request.setCharacterEncoding("GBK");
//下面依次获取表单域的值
String name = request.getParameter("name");
String gender = request.getParameter("gender");
//如果表单域是复选框，将使用该方法获取多个值
String[] color = request.getParameterValues("color");
String national = request.getParameter("country");

```

```
%>
<!-- 下面依次输出表单域的值-->
您的名字: <%=name%><hr>
您的性别: <%=gender%><hr>
//输出复选框获取的数组值
您喜欢的颜色: <%for(String c : color) {out.println(c + " ");}%><hr>
您来自的国家: <%=national%><hr>
</BODY>
</HTML>
```

在页面中可大量使用 request 对象来获取表单域的值，获取表单域的值有如下两个方法。

- String getParamete(String paramName): 获取表单域的值。
- String getParameterValues(String paramName): 获取表单域的数组值。

在获取表单域的值之前，先设置 request 的解码方式，因为获取的参数是简体中文，因此使用 GBK 的解码方式，设置解码方式时使用如下方法。

- setCharacterEncoding("GBK"): 设置解码方式。

在表单提交页的各个输入域内输入对应的值，然后单击【提交】按钮，request1.jsp 就会出现如图 2.19 所示的效果。

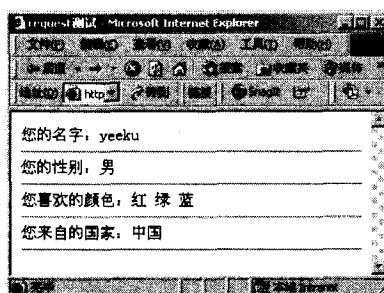


图 2.19 request 获取请求参数

## 2. 封装地址栏参数

如果需要传递的参数是普通字符串，而且在传递少量参数时，可以通过地址栏传递参数。地址栏传递参数的格式是 url?param1=value1&param2=value2&…

请求的 url 和参数之间以“?”分隔，而多个参数之间以“&”分隔。

看下面的 JSP 页面：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>request 测试</TITLE>
</HEAD>
<BODY>
<%
//获取 name 请求参数的值
String name = request.getParameter("name");
//获取 gender 请求参数的值
String gender = request.getParameter("gender");
```

```

%>
<!-- 输出 name 变量值-->
您的名字: <%=name%><hr>
<!-- 输出 gender 变量值-->
您的性别: <%=gender%><hr>
</BODY>
</HTML>

```

该页面使用 request 获取请求参数，此时不再使用表单提交请求参数，而是通过地址栏发送请求参数，直接向该页面请求，请求执行的效果如图 2.20 所示。

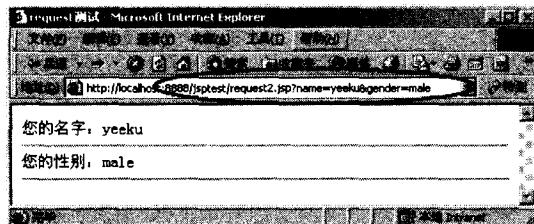


图 2.20 request 获取地址栏参数

### 3. 封装请求属性

HttpServletRequest 还包含用于设置和获取请求属性的两个方法：

- void setAttribute(String attName, Object attValue)。
- Object getAttribute(String attName)。

当 forward 用户请求时，请求的参数和请求属性都不会丢失。看下一个 JSP 页面，这个 JSP 页面是个简单的表单页，该表单页用于提交用户请求：

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>form 页</TITLE>
</HEAD>
<BODY>
<!-- 提交表单-->
<form method="post" action="first.jsp">
    <!-- 取钱的表单-->
    取钱: <input type="text" name="balance">
    <input type="submit" value="提交">
</form>
</BODY>
</HTML>

```

该页面向 first.jsp 页面请求后，balance 参数将被提交到 first.jsp 页面，下面是 first.jsp 页面的实现代码：

```

<%@ page contentType="text/html; charset=gb2312" language="java" import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>

```

```

<TITLE>request 处理 </TITLE>
</HEAD>
<BODY>
<%
//获取请求的钱数
String bal = request.getParameter("balance");
//将钱数的字符串转换成双精度浮点数
double qian = Double.parseDouble(bal);
//对取出的钱进行判断
if (qian < 500)
{
    out.println("给你" + qian + "块");
    out.println("账户减少" + qian);
}
else
{
    //创建了一个 List 对象
    List<String> info = new ArrayList<String>();
    info.add("1111111");
    info.add("2222222");
    info.add("3333333");
    //向 request 中封装了一个请求属性
    request.setAttribute("info" , info);
%>
<!-- 实现转发-->
<jsp:forward page="second.jsp"/>
<%
}
%>
</BODY>
</HTML>

```

first.jsp 页面首先获取请求的取钱数，然后对请求的钱数进行判断。如果请求的钱数小于 500，则允许直接取钱；否则将请求转发到 second.jsp。转发之前，在请求中封装了 info 属性，并在该属性中存入了一个 List 对象。

因此在 second.jsp 页面中，不仅获取了请求的 balance 参数，而且还会获取请求中的 info 属性。second.jsp 页面的代码如下：

```

<%@ page contentType="text/html; charset=gb2312" language="java" import="java.
util.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>request 处理</TITLE>
</HEAD>
<BODY>
<%
String bal = request.getParameter("balance");
double qian = Double.parseDouble(bal);
List<String> info = (List<String>)request.getAttribute("info");
for (String tmp : info)
{
    out.println(tmp + "<br>");
}
out.println("取钱" + qian + "块");
out.println("账户减少" + qian);
%>

```

```
</BODY>
</HTML>
```

如果页面请求的钱数大于 500 时，请求将被转发到 second.jsp 页面处理。当 forward 发出请求时，请求参数和请求属性都不会丢失，即在 second.jsp 页面可以获取请求参数和请求属性的值。

如果请求取钱的钱数为 654，则页面的执行效果如图 2.21 所示。

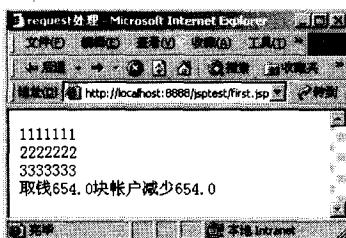


图 2.21 request 访问 attribute

## 2.8.7 response 对象

`response` 代表服务器对客户端的响应。大部分的时候，程序无须使用 `response` 来响应客户端请求，因为有个更简单的响应对象——`out`，它是页面输出流，是 `JspWriter` 的实例。`JspWriter` 是 `Writer` 的子类，`Writer` 是字符流，无法输出非字符内容——即无法输出字节流。

假如需要在 JSP 页面中动态生成一幅位图，使用 `out` 作为响应将无法完成，此时必须使用 `response` 作为响应输出。

除此之外，还可以使用 `response` 来重定向请求，以及用于向客户端增加 `Cookie`。

### 1. response 响应生成图片

对于需要生成非字符响应的情况，就应该使用 `response` 来响应客户端请求。下面的 JSP 页面将在客户端生成一张图片。

```
<%@ page import="java.awt.image.* , javax.imageio.* , java.io.* , java.awt.* "%>
<%
//创建 BufferedImage 对象
BufferedImage image = new BufferedImage(400, 400, BufferedImage.TYPE_INT_RGB);
//以 Image 对象获取 Graphics 对象
Graphics g = image.getGraphics();
//使用 Graphics 画图，所画的图像将会出现在 image 对象中
g.fillRect(0, 0, 400, 400);
//设置颜色：红
g.setColor(new Color(255, 0, 0));
//画出一段弧
g.fillArc(20, 20, 100, 100, 30, 120);
//设置颜色：绿
g.setColor(new Color(0, 255, 0));
//画出一段弧
```

```
g.fillArc(20, 20, 100, 100, 150, 120);
//设置颜色: 蓝
g.setColor(new Color(0, 0, 255));
//画出一段弧
g.fillArc(20, 20, 100, 100, 270, 120);
//设置颜色: 黑
g.setColor(new Color(0, 0, 0));
//画出三个字符串
g.drawString("red:climb", 300, 80);
g.drawString("green:swim", 300, 120);
g.drawString("blue:jump", 300, 160);
g.dispose();
//将图像输出到页面的响应
ImageIO.write(image, "bmp", response.getOutputStream());
%>
```

也可以在其他页面中使用如下标签来显示该图片页面，该 JSP 的代码如下：

```

```

在对 JSP 页面发出请求后，在浏览器中将会看到如图 2.22 所示效果。

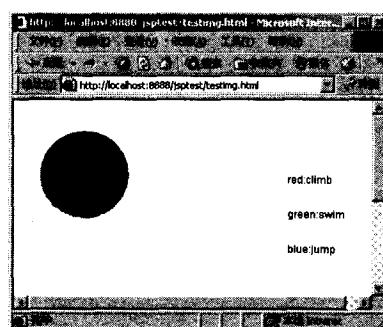


图 2.22 response 生成图片响应

## 2. 重定向

重定向是 response 的另外一个用处，与 forward 不同的是，重定向会丢失所有的请求参数及请求属性。

看下面的 JSP 页面代码：

```
<%@ page language="java" %>
<%
//生成页面响应
out.println("====");
//重定向到 forward-result.jsp 页面
response.sendRedirect("forward-result.jsp");
%>
```

当该页面输出页面响应后，就会发送重定向命令，页面控制将会重定向到 forward-result.jsp 页面。

在地址栏中输入 `http://localhost:8888/jsp/test/redirect.jsp?username="aaa"`，然后回车，看到如图 2.23 所示效果。

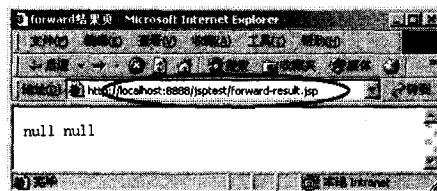


图 2.23 redirect 效果

注意地址栏的改变，使用重定向指令时，地址栏的地址会变成重定向的地址。

**注意：**重定向会丢失所有的请求参数，使用重定向的效果，与在地址栏里重新输入新地址再回车的效果完全一样。

### 3. 增加 Cookie

Cookie 通常用于网站记录客户的某些信息，比如客户的用户名及客户的喜好等。一旦用户下次登录，网站可以获取到客户的相关信息，根据这些客户信息，网站可以对客户提供更友好的服务。Cookie 与 session 的不同之处在于：session 关闭浏览器后就失效，但 Cookie 会一直存放在客户端机器上，除非超出 Cookie 的生命期限。

增加 Cookie 也是使用 response 内置对象完成的，response 对象提供了一个方法。

- void addCookie(Cookie cookie): 增加 Cookie。

正如在上面方法中见到的，在增加 Cookie 之前，必须先创建 Cookie 对象，增加 Cookie 请按如下步骤进行：

- (1) 创建 Cookie 实例；
- (2) 设置 Cookie 的生命期限；
- (3) 向客户端写 Cookie。

看如下 JSP 页面，该页面可以用于向客户端写一个 username 的 Cookie。

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>增加 Cookie</TITLE>
</HEAD>
<BODY>
<%
//获取请求参数
String name = request.getParameter("name");
//以获取到的请求参数为值，创建一个 Cookie 对象
Cookie c = new Cookie("username" , name);
//设置 Cookie 对象的生存期限
c.setMaxAge(24 * 3600);
//向客户端增加 Cookie 对象
response.addCookie(c);
%>
</BODY>
</HTML>
```

如果浏览器没有阻止 Cookie，在地址栏输入 `http://localhost:8888/jsptest/addcookie.jsp?name=waa`，执行该页面后，网站已经将客户端的 `username` 的 Cookie 写入客户端机器。该 Cookie 将在客户端硬盘上一直存在，直到超出该 Cookie 的生存期限（本 Cookie 设置为 24 小时）。

通过 `request` 对象的 `getCookies()` 方法来访问 Cookie，该方法将返回 Cookie 的数组，遍历该数组的每个元素，找出希望访问的 Cookie 即可。

下面是访问 Cookie 的 JSP 页面的代码：

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>增加 Cookie</TITLE>
</HEAD>
<BODY>
<%
//获取本站在客户端上保留的所有 cookie
Cookie[] cookies = request.getCookies();
//遍历客户端上的每个 cookie
for (Cookie c : cookies)
{
    //如果 Cookie 的名为 username，表明该 Cookie 是我们需要访问的 cookie
    if(c.getName().equals("username"))out.println(c.getValue());
}
%>
</BODY>
</HTML>
```

使用该页面将可读出刚才写在客户端的 Cookie。

注意：使用 Cookie 对象必须设置其生存期限，否则 Cookie 将会随浏览器的关闭而自动消失。

## 2.8.8 session 对象

session 对象也是一个非常常用的对象，这个对象代表一次用户会话。一次用户会话的含义是：从客户端浏览器连接服务器开始，到客户端浏览器与服务器断开为止，这个过程就是一次会话。

session 通常用于跟踪用户的会话信息，如判断用户是否登录系统，或者在购物车应用中，系统是否跟踪用户购买的商品等。

session 里的属性可以在多个页面的跳转之间共享。一旦关闭浏览器，即 session 结束，session 里的属性将全部清空。

session 对象的两个常用方法如下。

- `setAttribute(String attName, Object attValue)`: 设置一个 session 属性。
- `getAttribute(String attName)`: 返回一个 session 属性的值。

下面的示例演示了一个购物车应用，以下是商品陈列的 JSP 页面代码：

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>选择物品购买</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="processBuy.jsp">
    书籍: <INPUT TYPE="checkbox" NAME="item" value="book"><br>
    电脑: <INPUT TYPE="checkbox" NAME="item" value="computer"><br>
    汽车: <INPUT TYPE="checkbox" NAME="item" value="car"><br>
    <INPUT TYPE="submit" value="购买">
</FORM>
</BODY>
</HTML>

```

这个页面几乎没有动态的 JSP 部分，仅提供静态的 HTML。表单里包含三个复选按钮，用于提交想购买的物品，表单由 processBuy.jsp 页面处理，其页面的代码如下：

```

<%@ page contentType="text/html; charset=gb2312" language="java" import="java.
util.*"%>
<%
//从 session 对象中取出
Map<String, Integer> itemMap = (Map<String, Integer>)session.getAttribute
("itemMap");
//如果 Map 对象为空，则初始化 Map 对象
if (itemMap == null)
{
    itemMap = new HashMap<String, Integer>();
    itemMap.put("书籍" , 0);
    itemMap.put("电脑" , 0);
    itemMap.put("汽车" , 0);
}
//获取上个页面的请求参数
String[] buys = request.getParameterValues("item");
//遍历数组的各元素
for (String item : buys)
{
    //如果 item 为 book，表示选择购买书籍
    if(item.equals("book"))
    {
        int num1 = itemMap.get("书籍").intValue();
        //将书籍 key 对应的数量加 1
        itemMap.put("书籍" , num1 + 1);
    }
    //如果 item 为 computer，表示选择购买电脑
    else if (item.equals("computer"))
    {
        int num2 = itemMap.get("电脑").intValue();
        //将电脑 key 对应的数量加 1
        itemMap.put("电脑" , num2 + 1);
    }
    //如果 item 为 car，表示选择购买汽车
    else if (item.equals("car"))
    {
        int num3 = itemMap.get("汽车").intValue();
        //将汽车 key 对应的数量加 1
        itemMap.put("汽车" , num3 + 1);
    }
}

```

```

        }
    }
    //将 itemMap 对象放到 session 中
    session.setAttribute("itemMap", itemMap);
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>购买的物品列表</TITLE>
</HEAD>
<BODY>
您所购买的物品：<br>
书籍：<%=itemMap.get("书籍")%>本<br>
电脑：<%=itemMap.get("电脑")%>台<br>
汽车：<%=itemMap.get("汽车")%>辆<p>
<a href="shop.jsp">再次购买</a>
</BODY>
</HTML>

```

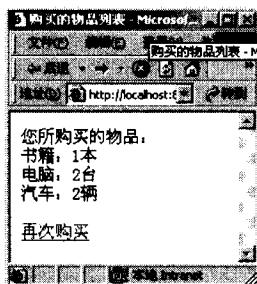


图 2.24 session 购物车效果

利用购物车系统可以反复购买，只要浏览器不关闭，购买的物品信息就不会丢失，图 2.24 显示的是多次购买后的效果。

## 2.9 Servlet 介绍

Servlet 是一种比 JSP 更早的动态网页编程技术。在没有 JSP 之前，Servlet 也是同时充当视图层、业务逻辑层及持久层角色。

Servlet 的开发效率非常低，特别是当使用 Servlet 生成表现层页面时，页面中所有的 HTML 标签，都需采用 Servlet 的输出流来输出，因此极其烦琐。由于 Servlet 是个标准的 Java 类，因此必须由程序员开发，其修改难度大，美工人员根本无法参与 Servlet 页面的开发。这一系列的问题，都阻碍了 Servlet 作为表现层的使用。

自 MVC 规范出现后，Servlet 的责任开始明确下来，仅仅作为控制器使用，不再需要生成页面标签，也不再作为视图层角色使用。

### 2.9.1 Servlet 的开发

Servlet，通常称为服务器端小程序，是运行在服务器端的程序，用于处理及响应客户端的请求。

Servlet 是个特殊的 Java 类，这个 Java 类必须继承 HttpServlet。每个 Servlet 可以响应客户端的请求。Servlet 提供不同的方法用于响应客户端请求。

- doGet：用于响应客户端的 get 请求。
- doPost：用于响应客户端的 post 请求。
- doPut：用于响应客户端的 put 请求。
- doDelete：用于响应客户端的 delete 请求。

事实上，客户端的请求通常只有 get 和 post 两种，Servlet 为了响应这两种请求，必须重写 doGet 和 doPost 两个方法。如果 Servlet 为了响应四个方法，则需要同时重写上面的四个方法。

大部分时候，Servlet 对于所有请求的响应都是完全一样的。此时，可以采用重写一个方法来代替上面的几个方法，Servlet 只需重写 service 方法即可响应客户端的所有请求。

另外，HttpServlet 还包含两个方法。

- init(ServletConfig config)：创建 Servlet 实例时，调用的初始化方法。
- destroy()：销毁 Servlet 实例时，自动调用的资源回收方法。

通常无须重写 init() 和 destroy() 两个方法，除非需要在初始化 Servlet 时，完成某些资源初始化的方法，才考虑重写 init 方法。如果需要在销毁 Servlet 之前，先完成某些资源的回收，比如关闭数据库连接等，才需要重写 destroy 方法。

**注意：**如果重写了 init(ServletConfig config) 方法，则应在重写该方法的第一行调用 super.init(config)。该方法将调用 HttpServlet 的 init 方法。

下面提供一个 Servlet 的示例，该 Servlet 将获取表单请求参数，并将请求参数显示给客户端：

```
//Servlet 必须继承 HttpServlet 类
public class FirstServlet extends HttpServlet
{
    //客户端的响应方法，使用该方法可以响应客户端所有类型的请求
    public void service(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        //设置解码方式
        request.setCharacterEncoding("GBK");
        //获取 name 的请求参数值
        String name = request.getParameter("name");
        //获取 gender 的请求参数值
        String gender = request.getParameter("gender");
        //获取 color 的请求参数值
        String[] color = request.getParameterValues("color");
        //获取 country 的请求参数值
        String national = request.getParameter("country");
        //获取页面输出流
        PrintStream out = new PrintStream(response.getOutputStream());
        //输出 HTML 页面标签
        out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional
        //EN\">");
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Servlet 测试</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        //输出请求参数的值: name
        out.println("您的名字: " + name + "<hr>");
        //输出请求参数的值: gender
        out.println("您的性别: " + gender + "<hr>");
        //输出请求参数的值: color
    }
}
```

```
        out.println("您喜欢的颜色: ");
        for(String c : color)
        {
            out.println(c + " ");
        }
        out.println("<hr>");
        out.println("您喜欢的颜色: ");
        //输出请求参数的值: national
        out.println("您来自的国家: " + national + "<hr>");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

对比该 Servlet 和 2.8.6 节中的 request1.jsp 页面。该 Servlet 和 request1.jsp 页面的效果完全相同，都通过 HttpServletRequest 获取客户端的 form 请求参数，并显示请求参数的值。

Servlet 和 JSP 的区别在于：

- Servlet 中没有内置对象，原来 JSP 中的内置对象都必须通过 HttpServletRequest 对象，或由 HttpServletResponse 对象生成；
- 对于静态的 HTML 标签，Servlet 都必须使用页面输出流逐行输出。

这也正是笔者在前面介绍的：JSP 是 Servlet 的一种简化，使用 JSP 只需要完成程序员需要输出到客户端的内容，至于 JSP 中的 Java 脚本如何镶嵌到一个类中，由 JSP 容器完成。而 Servlet 则是个完整的 Java 类，这个类的 service 方法用于生成对客户端的响应。

## 2.9.2 Servlet 的配置

编辑好的 Servlet 源文件并不能响应用户请求，还必须将其编译成 class 文件。将编译后的 FirstServlet.class 文件放在 WEB-INF/classes 路径下，如果 Servlet 有包，则还应该将 class 文件放在对应的包路径下。

为了让 Servlet 能响应用户请求，还必须将 Servlet 配置在 Web 应用中。配置 Servlet 时，需要修改 web.xml 文件。

配置 Servlet 需要配置两个部分。

- 配置 Servlet 的名字：对应 web.xml 文件中的<servlet>元素。
- 配置 Servlet 的 URL：对应 web.xml 文件中的<servlet-mapping>元素。

因此，配置一个能响应客户请求的 Servlet，至少需要配置两个元素，关于上面 FirstServlet 的配置如下：

```
<!-- 配置 Servlet 的名字-->
<servlet>
    <!-- 指定 Servlet 的名字-->
    <servlet-name>firstServlet</servlet-name>
    <!-- 指定 Servlet 的实现类-->
    <servlet-class>lee.FirstServlet</servlet-class>
</servlet>
<!-- 配置 Servlet 的 URL-->
<servlet-mapping>
```

```

<!-- 指定 Servlet 的名字-->
<servlet-name>firstServlet</servlet-name>
<!-- 指定 Servlet 映射的 URL 地址-->
<url-pattern>/firstServlet</url-pattern>
</servlet-mapping>

```

对 2.8.6 节中的 form 进行简单修改，将 form 表单元素的 action 修改成/firstServlet，在表单域中输入相应数据，然后单击【提交】按钮，效果如图 2.25 所示。

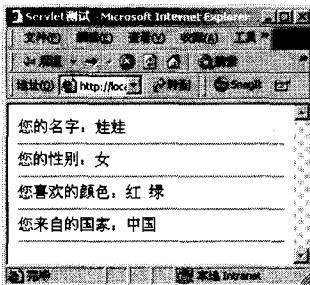


图 2.25 Servlet 处理用户请求

在这种情况下，Servlet 与 JSP 的作用效果完全相同。

### 2.9.3 Servlet 的生命周期

Servlet 在容器中运行，其实例的创建及销毁等都不是由程序员决定的，而是由容器进行控制。

Servlet 的创建有两个选择。

- 客户端请求对应的 Servlet 时，创建 Servlet 实例：大部分的 Servlet 都是这种 Servlet。
- Web 应用启动时，立即创建 Servlet 实例：即 load-on-startup Servlet。

每个 Servlet 的运行都遵循如下生命周期。

(1) 创建 Servlet 实例。

(2) Web 容器调用 Servlet 的 init 方法，对 Servlet 进行初始化。

(3) Servlet 初始化后，将一直存在于容器中，用于响应客户端请求。如果客户端有 get 请求，容器调用 Servlet 的 doGet 方法处理并响应请求。对于不同的请求，有不同的处理方法，或者统一使用 service 方法处理来响应用户请求。

(4) Web 容器角色销毁 Servlet 时，调用 Servlet 的 destroy 方法，通常在关闭 Web 容器之时销毁 Servlet。

Servlet 的生命周期如图 2.26 所示。

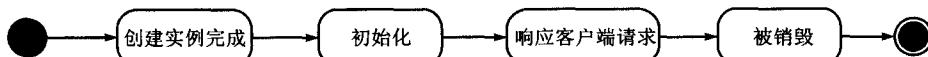


图 2.26 Servlet 的生命周期

## 2.9.4 使用 Servlet 作为控制器

正如前面见到，使用 Servlet 作为表现层的工作量太大，所有的 HTML 标签都需要使用页面输出流生成。因此，使用 Servlet 作为表现层有如下三个劣势。

- 开发效率低，所有的 HTML 标签都需使用页面输出流完成。
- 不利于团队协作开发，美工人员无法参与 Servlet 界面的开发。
- 程序可维护性差，即使修改一个按钮的标题，都必须重新编辑 Java 代码，并重新编译。

在标准的 MVC 模式中，Servlet 仅作为控制器使用。J2EE 的架构也是遵循 MVC 模式的，图 2.27 是 MVC 的示意图。

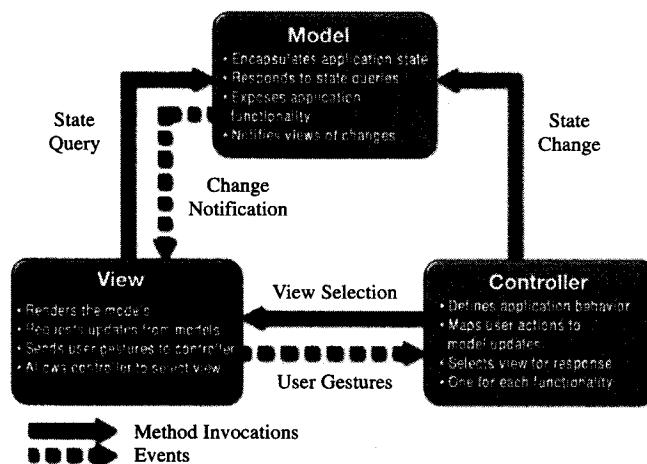


图 2.27 MVC 示意图

下面介绍一个使用 Servlet 作为控制器的 MVC 应用，该应用演示了一个简单的登录验证：

```

<%@ page language="java" contentType="text/html; charset=gb2312" errorPage="error.jsp"%>
<html>
<head>
<title>登录</title>
</head>
<body>
<!-- 输出出错提示-->
<font color="red">
<%
if (request.getAttribute("err") != null)
{
    out.println(request.getAttribute("err"));
}
%>
</font>

```

```

请输入用户名和密码:
<!-- 登录表单, 该表单提交到一个 Servlet-->
<form id="login" method="post" action="login">
    用户名: <input type="text" name="username"/><br>
    密 码: <input type="password" name="pass"/><br>
    <input type="submit" value="登录"/><br>
</form>
</body>
</html>

```

这是个标准的登录页面, 该页面将收集的用户名及密码, 提交到 Servlet。在这里, Servlet 充当控制器角色, 控制器 Servlet 的源代码如下:

```

public class LoginServlet extends HttpServlet
{
    //响应客户端请求
    public void service(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        //Servlet 本身并不输出响应到客户端, 因此必须将请求转发
        RequestDispatcher rd;
        //获取请求参数
        String username = request.getParameter("username");
        String pass = request.getParameter("pass");
        //
        try
        {
            //Servlet 本身, 并不执行任何的业务逻辑处理, 它调用 JavaBean 处理用户请求
            DbDao dd = DbDao.instance("com.mysql.jdbc.Driver",
                                       "jdbc:mysql://localhost:3306/liuyan", "root", "32147");
            //查询结果集
            ResultSet rs = dd.query("select password from user_table where
username = '" + username + "'");
            if (rs.next())
            {
                //用户名和密码匹配
                if (rs.getString("password").equals(pass))
                {
                    //获取 session 对象
                    HttpSession session = request.getSession(true);
                    //设置 session 属性, 跟踪用户会话状态
                    session.setAttribute("name", username);
                    //获取转发对象
                    rd = request.getRequestDispatcher("/welcome.jsp");
                    //转发请求
                    rd.forward(request, response);
                }
                else
                {
                    //用户名和密码不匹配时
                    errMsg += "您的用户名密码不符合, 请重新输入";
                }
            }
            else
            {
                //用户名不存在时
                errMsg += "您的用户名不存在, 请先注册";
            }
        }
    }
}

```

```
        }
    }
    catch (Exception e)
    {
        rd = request.getRequestDispatcher("/error.jsp");
        request.setAttribute("exception" , "业务异常");
        rd.forward(request,response);
    }
    //如果出错，转发到重新登录
    if (errMsg != null && !errMsg.equals(""))
    {
        rd = request.getRequestDispatcher("/login.jsp");
        request.setAttribute("err" , errMsg);
        rd.forward(request,response);
    }
}
}
```

控制器负责接收客户端的请求，它既不直接对客户端输出响应，也不处理用户请求，只将请求转发到 JSP 页，通过调用 JavaBean 来处理用户请求。

下面是 Model JavaBean 的源代码：

```
public class DbDao
{
    //该 JavaBean 做成单态模式
    private static DbDao op;
    private Connection conn;
    private String driver;
    private String url;
    private String username;
    private String pass;
    //构造器私有
    private DbDao()
    {
    }
    //构造器私有
    private DbDao(String driver,String url,String username,String pass)
    throws Exception
    {
        this.driver = driver;
        this.url = url;
        this.username = username;
        this.pass = pass;
        Class.forName(driver);
        conn = DriverManager.getConnection(url,username,pass);
    }
    //下面是各个成员属性的 setter 和 getter 方法
    public void setDriver(String driver) {
        this.driver = driver;
    }
    public void setUrl(String url) {
        this.url = url;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public void setPass(String pass) {
        this.pass = pass;
```

```
}

public String getDriver() {
    return (this.driver);
}

public String getUrl() {
    return (this.url);
}

public String getUsername() {
    return (this.username);
}

public String getPass() {
    return (this.pass);
}

//获取数据库连接
public void getConnection() throws Exception
{
    if (conn == null)
    {
        Class.forName(this.driver);
        conn = DriverManager.getConnection(this.url,this.username,
                                         this. pass);
    }
}

//实例化 JavaBean 的入口
public static DbDao instance()
{
    if (op == null)
    {
        op = new DbDao();
    }
    return op;
}

//实例化 JavaBean 的入口
public static DbDao instance(String driver,String url,String username,
String pass) throws Exception
{
    if (op == null)
    {
        op = new DbDao(driver,url,username,pass);
    }
    return op;
}

//插入记录
public boolean insert(String sql) throws Exception
{
    getConnection();
    Statement stmt = this.conn.createStatement();
    if (stmt.executeUpdate(sql) != 1)
    {
        return false;
    }
    return true;
}

//执行查询
public ResultSet query(String sql) throws Exception
{
    getConnection();
    Statement stmt = this.conn.createStatement();
    return stmt.executeQuery(sql);
}
```

```
    }
    //执行删除
    public void delete(String sql) throws Exception
    {
        getConnection();
        Statement stmt = this.conn.createStatement();
        stmt.executeUpdate(sql);
    }
    //执行更新
    public void update(String sql) throws Exception
    {
        getConnection();
        Statement stmt = this.conn.createStatement();
        stmt.executeUpdate(sql);
    }
}
```

由此可见，其整个结构非常清晰，下面是 MVC 中各个角色的对应组件。

M: Model，即模型，对应 JavaBean。

V: View，即视图，对应 JSP 页面。

C: Controller，即控制器，对应 Servlet。

## 2.9.5 load-on-startup Servlet

在 2.9.3 节中已经介绍过，Servlet 的实例化有两个时机：用户请求之时，或应用启动之时。应用启动时就启动的 Servlet 通常是用于某些后台服务的 Servlet，或者拦截很多请求的 Servlet；这种 Servlet 通常作为应用的基础 Servlet 使用，提供重要的后台服务。

如果需要 Web 应用启动时，可使用 `load-on-startup` 元素完成 Servlet 的初始化。`load-on-startup` 元素只接收一个整型值，这个整型值越小，Servlet 就越优先初始化。

下面是个简单的 Servlet，该 Servlet 不响应用户请求，它仅仅执行计时器功能，每隔一段时间会在控制台打印出当前时间：

```
public class TimerServlet extends HttpServlet
{
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        Timer t = new Timer(1000, new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println(new Date());
            }
        });
        t.start();
    }
}
```

该 Servlet 不会响应用户请求，因此它无须配置 URL 映射，只能在应用启动时初始化。将该 Servlet 配置成 `load-on-startup` Servlet，即可保证应用启动时初始化，创建该 Servlet 实例的配置片段如下：

```

<servlet>
    <!-- Servlet 名-->
    <servlet-name>timerServlet</servlet-name>
    <!-- Servlet 的实现类-->
    <servlet-class>lee.TimerServlet</servlet-class>
    <!-- 配置应用启动时，创建 Servlet 实例-->
    <load-on-startup>1</load-on-startup>
</servlet>

```

该 Servlet 将一直作为后台服务执行。

## 2.9.6 访问 Servlet 的配置参数

配置 Servlet 时，还可以增加附加的配置参数。通过使用配置参数，可以实现更好地解耦，避免将所有的参数以硬编码方式写在程序中。

访问 Servlet 配置参数要通过 `ServletConfig` 类的实例完成，`ServletConfig` 提供如下方法。

- `java.lang.String getInitParameter(java.lang.String name)`: 用于获取初始化参数。

注意：JSP 的内置对象 `config` 就是此处的 `ServletConfig`。

看下面的 Servlet 代码：

```

public class TestServlet extends HttpServlet
{
    //重写 init 方法,
    public void init(ServletConfig config) throws ServletException
    {
        //重写该方法，应该首先调用父类的 init 方法
        super.init(config);
    }
    //响应客户端请求的方法
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        try
        {
            //获取 ServletConfig 对象
            ServletConfig config = getServletConfig();
            //通过 ServletConfig 对象获取配置参数: dirver
            String driver = config.getInitParameter("driver");
            //通过 ServletConfig 对象获取配置参数: url
            String url = config.getInitParameter("url");
            //通过 ServletConfig 对象获取配置参数: user
            String user = config.getInitParameter("user");
            //通过 ServletConfig 对象获取配置参数: pass
            String pass = config.getInitParameter("pass");
            //注册驱动
            Class.forName(driver);
            //获取数据库驱动
            Connection conn = DriverManager.getConnection(url,user,pass);
            //创建 Statement 对象
        }
    }
}

```

```
Statement stmt = conn.createStatement();
//执行查询，获取 ResultSet 对象
ResultSet rs = stmt.executeQuery("select * from userinf");
//获取页面输出流
PrintStream out = new PrintStream(response.getOutputStream());
//输出 HTML 标签
out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
Transitional//EN\"");
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>application 测试</TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("<table bgcolor=\"9999dd\" border=\"1\" align=\"center\">");
//遍历结果集
while(rs.next())
{
    //输出结果集内容
    out.println("<tr>");
    out.println("<td>" + rs.getString(2) + "</td>");
    out.println("<td>" + rs.getString(3) + "</td>");
    out.println("</tr>");
}
out.println("</table>");
out.println("</BODY>");
out.println("</HTML>");
}
catch (Exception e)
{
    e.printStackTrace();
}
}
}
```

Servlet 访问配置参数的方式非常类似于 application 方式。只是 application 的配置参数对整个 Web 应用有效，而 Servlet 配置参数仅对该 Servlet 有效，下面是配置该 Servlet 的配置片段：

```
<servlet>
    <!-- 配置 Servlet 名-->
    <servlet-name>testServlet</servlet-name>
    <!-- 指定 Servlet 的实现类-->
    <servlet-class>lee.TestServlet</servlet-class>
    <!-- 配置 Servlet 的初始化参数： driver-->
    <init-param>
        <param-name>driver</param-name>
        <param-value>com.mysql.jdbc.Driver</param-value>
    </init-param>
    <!-- 配置 Servlet 的初始化参数： url-->
    <init-param>
        <param-name>url</param-name>
        <param-value>jdbc:mysql://localhost:3306/j2ee</param-value>
    </init-param>
    <!-- 配置 Servlet 的初始化参数： user-->
    <init-param>
        <param-name>user</param-name>
        <param-value>root</param-value>
```

```

</init-param>
<!-- 配置 Servlet 的初始化参数: pass-->
<init-param>
    <param-name>pass</param-name>
    <param-value>32147</param-value>
</init-param>
</servlet>

```

另外，还需要为该 Servlet 配置 URL 映射：

```

<servlet-mapping>
    <!-- 确定 Servlet 名-->
    <servlet-name>testServlet</servlet-name>
    <!-- 配置 Servlet 映射的 URL-->
    <url-pattern>/testServlet</url-pattern>
</servlet-mapping>

```

在浏览器中浏览该 Servlet，可看到数据库访问成功的效果图（如果数据的配置正确）。

## 2.10 自定义标签库

在 JSP 规范的 1.1 版中增加了自定义标签库。自定义标签库是一种非常优秀的组件技术。通过使用自定义标签库，可以在简单的标签中封装复杂的功能。

实现自定义标签按如下步骤进行：

- (1) 开发自定义标签处理类；
- (2) 建立一个\*.tld 文件，每个\*.tld 文件对应一个标签库，每个标签库对应多个标签；
- (3) 在 web.xml 文件中增加自定义标签的定义；
- (4) 在 JSP 文件中使用自定义标签。

### 2.10.1 开发自定义标签类

使用标签类，可以使用简单的标签来封装复杂的功能，从而使团队更好地协作开发（能让美工人员更好地参与 JSP 页面的开发）。

自定义标签类都必须继承一个父类：java.Servlet.jsp.tagext.TagSupport。除此之外，自定义标签类还有如下要求。

- 如果标签类包含属性，每个属性都有对应的 getter 和 setter 方法。
- 重写 doStartTag() 或 doEndTag() 方法，这两个方法生成页面内容。
- 如果需要在销毁标签之前完成资源回收，则重写 release() 方法。

下面提供了一个最简单的标签代码：

```

//标签处理类，继承 TagSupport 父类
public class HelloWorldTag extends TagSupport
{
    //重写 doEndTag 方法，该方法在标签结束生成页面内容
    public int doEndTag() throws JspTagException
    {
        try

```

```
{  
    //获取页面输出流，并输出字符串  
    pageContext.getOut().write("Hello World");  
}  
//捕捉异常  
catch (IOException ex)  
{  
    //抛出新异常  
    throw new JspTagException("错误");  
}  
//返回值  
return EVAL_PAGE;  
}  
}
```

这是个非常简单的标签，它只在页面中生成一个“Hello World”的字符串。该标签没有属性，因此无须提供 `setter` 和 `getter` 方法；此外，该标签无须初始化资源，因此无须重写 `init` 方法；在标签结束时无须回收资源，因此无须重写 `destroy` 方法。

## 2.10.2 建立 TLD 文件

TLD 是 Tag Library Definition 的缩写，即标签库定义，文件的后缀是 `tld`，每个 TLD 文件对应一个标签库，一个标签库中可包含多个标签。TLD 文件也称为标签库定义文件。

标签库定义文件的根元素是 `taglib`，它可以有多个 `tag` 子元素，每个 `tag` 子元素都对应一个标签。

下面是 `test.tld` 的标签定义文件，该文件中包含了 `HelloWorldTag` 标签的定义代码：

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"  
      "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">  
<taglib>  
    <!-- 定义标签库版本-->  
    <tlib-version>1.0</tlib-version>  
    <!-- 定义 JSP 版本-->  
    <jsp-version>1.2</jsp-version>  
    <!-- 配置第一个标签-->  
    <tag>  
        <!-- 配置标签名-->  
        <name>helloworld</name>  
        <!-- 确定标签的处理类-->  
        <tag-class>mytag.HelloWorldTag</tag-class>  
        <!-- 确定标签的标签体，标签体为空-->  
        <body-content>empty</body-content>  
    </tag>  
</taglib>
```

这个标签库配置文件非常简单，没有标签体及标签属性等。只是一个空标签。

## 2.10.3 在 web.xml 文件中增加标签库定义

编辑了标签库定义文件还不够，Web 容器还无法加载标签库定义文件。还必须在

web.xml 文件中增加标签库的定义。

在 web.xml 文件中定义标签库时使用 taglib 元素，该元素包含两个子元素：taglib-uri 和 taglib-location，前者确定标签库的 URI；后者确定标签库定义文件的位置。

下面是 web.xml 文件中关于 test.tld 标签库的定义片段：

```
<!-- 定义标签库-->
<taglib>
    <!-- 确定标签库的 URI-->
    <taglib-uri>/tags/tldtest.tld</taglib-uri>
    <!-- 确定标签库定义文件的位置-->
    <taglib-location>/WEB-INF/tldtest.tld</taglib-location>
</taglib>
```

如果需要使用多个标签库，只需要增加多个 taglib 元素即可，因为每个 taglib 元素可对应一个标签库。

## 2.10.4 使用标签库

使用标签库分成以下两步。

- (1) 导入标签库：使用 taglib 编译指令导入标签。
- (2) 使用标签：在 JSP 页面中使用自定义标签。

taglib 的语法格式如下：

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

其中 uri 属性确定标签库定义文件的 URI，这个 URI 就是在 web.xml 文件中为标签库定义的 URI。而 prefix 属性确定的是标签前缀，即在 JSP 页面中使用标签时，该标签库负责处理的标签前缀。

使用标签的语法格式如下：

```
<tagPrefix : tagName tagAttribute="tagValue" ...>
    <tagBody/>
</tagPrefix>
```

如果该标签没有标签体，则可以使用如下语法格式：

```
<tagPrefix : tagName tagAttribute="tagValue" ... />
```

下面的 JSP 页面使用 HelloWorldTag 标签：

```
<%@ page contentType="text/html; charset=GBK" %>
<!-- 导入标签库，指定 mytag 前缀的标签，由/tags/tldtest.tld 的标签库处理-->
<%@ taglib uri="/tags/tldtest.tld" prefix="mytag" %>
<html>
<head>
<title>自定义标签示范</title>
</head>
<body bgcolor="#ffffc0">
<h2>下面显示的是自定义标签中的内容</h2>
<!-- 使用标签，其中 mytag 是标签前缀，根据 taglib 的编译指令，mytag 前缀将
```

由于 uri 为 /tags/tldtest.tld 的标签库处理 -->

```
<mytag:helloworld/><BR>
</body>
</html>
```

该页面的执行效果如图 2.28 所示。

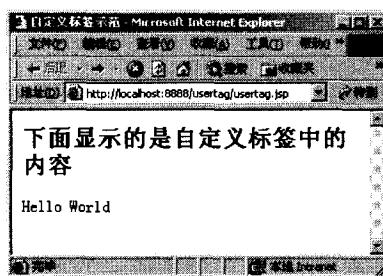


图 2.28 简单标签

## 2.10.5 带属性的标签

除了前面的简单标签外，还有如下两种常用的标签。

- 带属性的标签。
- 带标签体的标签。

正如前面介绍的，带属性的标签必须为每个属性提供对应的 `setter` 和 `getter` 方法。带属性的标签的配置方法与简单标签也略有差别。

下面介绍一个带属性标签的示例：

```
public class QueryTag extends TagSupport
{
    //标签的属性
    private String driver;
    private String url;
    private String user;
    private String pass;
    private String sql;
    //执行数据库访问的对象
    private Connection conn = null;
    private Statement stmt = null;
    private ResultSet rs = null;
    private ResultSetMetaData rsmd = null;
    //标签属性 driver 的 setter 方法
    public void setDriver(String driver) {
        this.driver = driver;
    }
    //标签属性 url 的 setter 方法
    public void setUrl(String url) {
        this.url = url;
    }
    //标签属性 user 的 setter 方法
    public void setUser(String user) {
        this.user = user;
    }
}
```

```
}

//标签属性 pass 的 setter 方法
public void setPass(String pass) {
    this.pass = pass;
}

//标签属性 driver 的 getter 方法
public String getDriver() {
    return (this.driver);
}

//标签属性 url 的 getter 方法
public String getUrl() {
    return (this.url);
}

//标签属性 user 的 getter 方法
public String getUser() {
    return (this.user);
}

//标签属性 pass 的 getter 方法
public String getPass() {
    return (this.pass);
}

//标签属性 sql 的 getter 方法
public String getSql() {
    return (this.sql);
}

//标签属性 sql 的 setter 方法
public void setSql(String sql) {
    this.sql = sql;
}

//标签处理
public int doEndTag() throws JspTagException
{
    try
    {
        //注册驱动
        Class.forName(driver);
        //获取数据库连接
        conn = DriverManager.getConnection(url,user,pass);
        //创建 Statement 对象
        stmt = conn.createStatement();
        //执行查询
        rs = stmt.executeQuery(sql);
        rsmd = rs.getMetaData();
        //获取列数目
        int columnCount = rsmd.getColumnCount();
        //获取页面输出流
        Writer out = pageContext.getOut();
        //在页面输出表格
        out.write("<table border='1' bgColor='9999cc'>");
        //遍历结果集
        while (rs.next())
        {
            out.write("<tr>");
            //逐列输出查询到的数据
            for (int i = 1 ; i <= columnCount ; i++ )
            {
                out.write("<td>");
                out.write(rs.getString(i));
                out.write("</td>");
            }
        }
    }
}
```

```
        }
        out.write("</tr>");
    }
}
catch (Exception ex)
{
    ex.printStackTrace();
    throw new JspTagException("错误");
}
return EVAL_PAGE;
}
//销毁标签前调用的方法
public void destroy()
{
    //关闭结果集
    if (rs != null)
    try
    {
        rs.close();
    }
    catch (SQLException sqle)
    {
        sqle.printStackTrace();
    }
    //关闭 Statement
    if (stmt != null)
    try
    {
        stmt.close();
    }
    catch (SQLException sqle)
    {
        sqle.printStackTrace();
    }
    //关闭数据库连接
    if (conn != null)
    try
    {
        conn.close();
    }
    catch (SQLException sqle)
    {
        sqle.printStackTrace();
    }
}
}
}
```

这个标签有点复杂，它包含 5 个属性：driver, url, user, pass, sql。标签体将根据这 5 个属性执行查询，并将查询结果在页面上显示。由于这个标签带有 5 个属性，因此配置文件也需要指定属性，下面是关于这个标签的配置文件片段：

```
<tag>
<!-- 配置标签名-->
<name>query</name>
<!-- 配置标签的处理类-->
<tag-class>mytag.QueryTag</tag-class>
<!-- 指定标签体为空-->
<body-content>empty</body-content>
```

```

<!-- 配置标签属性: driver-->
<attribute>
    <name>driver</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<!-- 配置标签属性: url -->
<attribute>
    <name>url</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<!-- 配置标签属性: user -->
<attribute>
    <name>user</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<!-- 配置标签属性: pass -->
<attribute>
    <name>pass</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<!-- 配置标签属性: sql -->
<attribute>
    <name>sql</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
</tag>

```

配置完毕后，就可在页面中使用标签，先导入标签库，然后使用标签。使用标签的代码片段如下：

```

<%@ taglib uri="/tags/tldtest.tld" prefix="mytag" %>
...<!-- 其他 HTML 内容-->
<mytag:query driver="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/j2ee"
user="root" pass="32147" sql="select * from products"/>

```

在浏览器中浏览该页面，效果如图 2.29 所示。

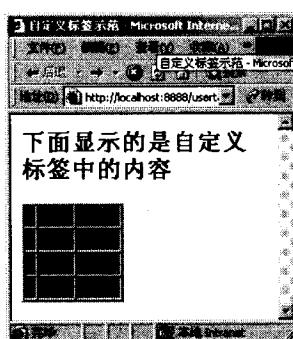


图 2.29 带属性的标签

在 JSP 页面中只需要使用简单的标签，即可完成“复杂”的功能：执行数据库查询，并将查询结果在页面上以表格形式显示。这也正是自定义标签库的目的——以简单的标签，隐藏复杂的逻辑。事实上，自定义标签的功能非常强大，很多项目都提供了相关的标签库，比如 Struts、WebWork 等。

注意：JSTL 是 Sun 提供的一套标签库，这套标签库的功能非常强大。另外，DisplayTag 是 Apache 组织下的一套开源标签库，主要用于生成页面并显示效果。

## 2.10.6 带标签体的标签

带标签体的标签，就是允许在标签内嵌套标签，通常可用于完成一些逻辑运算例如判断和循环等。

带标签体的标签需要继承 BodyTagSupport，该类包含一个 bodyContent 属性，该属性代表标签体。

BodyTagSupport 还包含两个方法。

- doAfterBody(): 每次处理完标签体后调用该方法。
- void doInitBody(): 开始调用标签体时调用该方法。

如果有必要，可以重写这两个方法。

下面以一个迭代器标签为示例，介绍如何开发一个带标签体的标签，该标签体包含两个属性：bean 和 item，bean 属性代表 page 范围内的一个 List；而 item 代表 List 中的每个元素。标签的源代码如下：

```
public class MyIteratorTag extends BodyTagSupport
{
    //标签需要迭代的集合对象名
    private String bean;
    //集合对象的元素
    private String item;
    //集合的当前索引
    private int i = 0;
    private int size;
    private List<String> itemList;
    //bean 属性的 setter 方法
    public void setBean (String s)
    {
        bean = s;
    }
    //bean 属性的 getter 方法
    public String getBean ()
    {
        return bean;
    }
    //item 属性的 setter 方法
    public void setItem (String s)
    {
        item = s;
    }
    //item 属性的 getter 方法
```

```

public String getItem ()
{
    return item;
}
//开始处理标签时，调用该方法。
public int doStartTag() throws JspTagException
{
    //从 page 范围中获取 List 对象
    itemList = (List<String>)pageContext.getAttribute(bean);
    //获取 List 的长度
    size = itemList.size();
    //将集合当前索引的值放在 page 范围的 item 变量中
    pageContext.setAttribute(item , itemList.get(i));
    //返回值为 EVAL_BODY_BUFFERED，表明需要计算标签体
    return EVAL_BODY_BUFFERED;
}
//每次标签体处理完后调用该方法
public int doAfterBody() throws JspException
{
    //移动 List 对象的索引位置
    i++ ;
    //如果索引已经超过了集合的长度
    if (i >= size)
    {
        //将索引回零
        i = 0;
        //不再计算标签体，直接调用 doEndTag 方法
        return SKIP_BODY;
    }
    //将集合的当前元素值放入 page 范围的 item 属性中
    pageContext.setAttribute(item , itemList.get(i));
    //循环计算标签体
    return EVAL_BODY_AGAIN;
}
//标签体结束时调用该方法
public int doEndTag() throws JspTagException
{
    try
    {
        //输出标签体内容
        bodyContent.writeOut(pageContext.getOut());
    }
    catch (IOException ex)
    {
        throw new JspTagException("错误");
    }
    return EVAL_PAGE;
}
}

```

下面是一个嵌套在该标签内的带属性的标签，该标签的功能非常简单，仅仅从 page 范围中获取属性，然后在页面上输出该属性值。其代码如下：

```

public class WritorTag extends TagSupport
{
    //item 属性，该标签从 page 中查找 item 的属性，并输出属性值
    private String item;
    //item 的 setter 方法

```

```
public void setItem (String s)
{
    item = s;
}
//item 的 getter 方法
public String getItem ()
{
    return item;
}
//开始处理标签时的调用该方法
public int doStartTag() throws JspTagException
{
    try
    {
        //从 page 范围内搜索 item 的属性,
        pageContext.getOut().write((String)pageContext.getAttribute(item));
    }
    catch (IOException ex)
    {
        throw new JspTagException("错误");
    }
    //返回 EVAL_PAGE, 继续计算页面输出。
    return EVAL_PAGE;
}
}
```

在处理标签类的各个方法中，不同的返回值对应不同的含义，常用的返回值有以下几个。

- SKIP\_BODY: 不处理标签体，直接调用 doEndTag()方法。
- SKIP\_PAGE: 忽略标签后面的 JSP 页面。
- EVAL\_PAGE: 处理标签结束，直接处理页面内容。
- EVAL\_BODY\_BUFFERED: 处理标签体。
- EVAL\_BODY\_INCLUDE: 处理标签体，但忽略 setBodyContent()和 doInitBody()方法。
- EVAL\_BODY\_AGAIN: 对标签体循环处理。

将上面两个标签配置在标签库中，标签库的配置片段如下：

```
<!-- 配置迭代器标签-->
<tag>
    <!-- 配置标签名-->
    <name>iterator</name>
    <!-- 配置标签的实现类-->
    <tag-class>mytag.MyIteratorTag</tag-class>
    <!-- 配置标签的标签体内容-->
    <body-content>JSP</body-content>
    <!-- 配置 bean 属性-->
    <attribute>
        <name>bean</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
    <!-- 配置 item 属性-->
    <attribute>
        <name>item</name>
        <required>true</required>
```

```

<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
<!-- 配置输出标签-->
<tag>
    <!-- 配置标签名-->
    <name>write</name>
    <!-- 配置标签的实现类-->
    <tag-class>mytag.WritorTag</tag-class>
    <!-- 配置标签的标签体内容：只能是空标签-->
    <body-content>empty</body-content>
    <!-- 配置标签属性 item-->
    <attribute>
        <name>item</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>

```

在 JSP 中嵌套使用两个标签的代码如下：

```

<%
//创建 List 对象
List<String> a = new ArrayList<String>();
a.add("hello");
a.add("world");
a.add("java");
//将 List 放入 page 范围的属性 a
pageContext.setAttribute("a" , a);
%>
<!-- 元素放在表格中-->
<table border="1" bgcolor="ddccff">
<!-- 使用迭代器标签，对 List 对象 a 进行迭代-->
<mytag:iterator bean="a" item = "item">
    <tr>
        <td>
            <!-- 输出 item 属性的值-->
            <mytag:write item="item" />
        </td>
    </tr>
</mytag:iterator>
</table>

```

页面的执行效果如图 2.30 所示。



图 2.30 迭代器标签

注意：本示例的迭代器仅对 page 范围的 List 进行迭代，用法有所局限。读者可以将其扩展，增加一个属性，指定迭代搜索的范围，并可将迭代的目标不局限于 List，而是 Collection，甚至包括数组。大部分的框架如 Struts、WebWork 都包含了自己的迭代器标签。

## 2.11 Filter 介绍

Filter 并不是一个标准的 Servlet，它不能处理用户请求，也不能对客户端生成响应。主要用于对 HttpServletRequest 进行预处理，也可以对 HttpServletResponse 进行后处理，是个典型的处理链。

Filter 有如下几个用处。

- 在 HttpServletRequest 到达 Servlet 之前，拦截客户的 HttpServletRequest。
- 根据需要检查 HttpServletRequest，也可以修改 HttpServletRequest 头和数据。
- 在 HttpServletResponse 到达客户端之前，拦截 HttpServletResponse。
- 根据需要检查 HttpServletResponse，也可以修改 HttpServletResponse 头和数据。

Filter 有如下几个种类。

- 用户授权的 Filter：Filter 负责检查用户请求，根据请求过滤用户非法请求。
- 日志 Filter：详细记录某些特殊的用户请求。
- 负责解码的 Filter：包括对非标准编码的请求解码。
- 能改变 XML 内容的 XSLT Filter 等。

一个 Filter 可负责拦截多个请求或响应；一个请求或响应也可被多个请求拦截。

创建一个 Filter 只需两个步骤：

- (1) 创建 Filter 处理类；
- (2) 在 web.xml 文件中配置 Filter。

### 2.11.1 创建 Filter 类

创建 Filter 必须实现 javax.servlet.Filter 接口，在该接口中定义了三个方法。

- void init(FilterConfig config)：用于完成 Filter 的初始化。
- void destroy()：用于 Filter 销毁前，完成某些资源的回收。
- void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)：实现过滤功能，该方法就是对每个请求及响应增加的额外处理。

下面介绍一个日志 Filter，这个 Filter 负责拦截所有的用户请求，并将请求的信息记录在日志中：

```
public class LogFilter implements Filter
{
    //FilterConfig 可用于访问 Filter 的配置信息
    private FilterConfig config;
    //实现初始化方法
```

```

public void init(FilterConfig config)
{
    this.config = config;
}
//实现销毁方法
public void destroy()
{
    this.config = null;
}
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
{
    //获取 ServletContext 对象，用于记录日志
    ServletContext context = this.config.getServletContext();
    long before = System.currentTimeMillis();
    System.out.println("开始过滤...");
    //将请求转换成 HttpServletRequest 请求
    HttpServletRequest hrequest = (HttpServletRequest)request;
    //记录日志
    context.log("Filter 已经截获到用户的请求的地址: " + hrequest.
    getServletPath());
    try
    {
        //Filter 只是链式处理，请求依然转发到目的地址。
        chain.doFilter(request, response);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    long after = System.currentTimeMillis();
    //记录日志
    context.log("过滤结束");
    //再次记录日志
    context.log("请求被定位到" + ((HttpServletRequest)request).
    getRequestURI() +
        " 所花的时间为: " + (after - before));
}
}
}

```

在上面的请求 Filter 中，仅在日志中记录请求的 URL，对所有的请求都执行 chain.doFilter(request,reponse)方法，当 Filter 对请求过滤后，依然将请求发送到目的地址。如果检查权限，可以在 Filter 中根据用户请求的 HttpSession，判断用户权限是否足够。如果权限不够，则调用重定向即可，无须调用 chain.doFilter(request,reponse)方法。

## 2.11.2 配置 Filter

Filter 的配置和 Servlet 的配置非常相似，都需要配置两个部分：

- 配置 Filter 名。
- 配置 Filter 拦截 URL 模式。

区别在于，Servlet 通常只配置一个 URL，而 Filter 可以同时拦截多个请求的 URL。因此，可以配置多个 Filter 拦截模式。

下面是该 Filter 的配置片段：

```
<!-- 定义 Filter-->
<filter>
    <!-- Filter 的名字-->
    <filter-name>log</filter-name>
    <!-- Filter 的实现类-->
    <filter-class>lee.LogFilter</filter-class>
</filter>
<!-- 定义 Filter 拦截地址-->
<filter-mapping>
    <!-- Filter 的名字-->
    <filter-name>log</filter-name>
    <!-- Filter 负责拦截的 URL-->
    <servlet-name>*</servlet-name>
</filter-mapping>
```

在上面的配置代码中，名为 log 的 Filter 负责拦截所有的客户端请求，该 Filter 并未对客户端请求进行额外的处理，仅仅在日志中简要记录请求的信息。

在该 Web 应用中执行请求时，控制台的效果如图 2.31 所示。

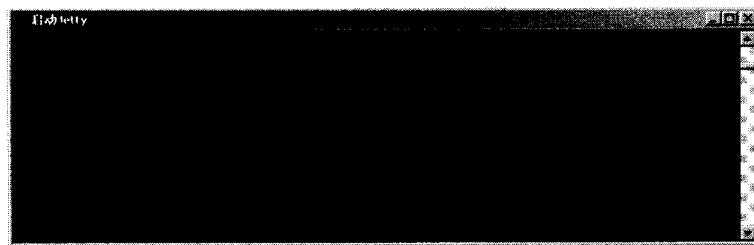


图 2.31 Filter 过滤客户端请求

## 2.12 Listener 介绍

Listener 的作用非常类似于 load-on-startup Servlet。用于在 Web 应用启动时，启动某些后台程序，这些后台程序负责为系统运行提供支持。

Listener 与 load-on-startup Servlet 的区别在于：Listener 的启动时机比 load-on-startup Servlet 早，只是 Listener 是 Servlet 2.3 规范之后才出现的。

使用 Listener 只需要两个步骤：

- (1) 创建 Listener 实现类。
- (2) 在 web.xml 文件中配置 Listener。

### 2.12.1 创建 Listener 类

创建 Listener 类必须实现 ServletContextListener 接口，该接口包含两个方法。

- contextInitialized(ServletContextEvent sce): 启动 Web 应用时，系统调用该 Filter

的方法。

- **contextDestroyed(ServletContextEvent sce):** 关闭 Web 应用时候，系统调用 Filter 的方法。

下面是任务调度的 Listener 类：

```
public class ScheduleListener implements ServletContextListener
{
    //使用 Java Timer 作为任务调度器
    private java.util.Timer timer = null;
    public void contextInitialized(ServletContextEvent sce)
    {
        //启动调度器
        timer = new Timer(true);
        //记录日志
        sce.getServletContext().log(new java.util.Date() + " 计时器已经启动...");
        sce.getServletContext().log (new java.util.Date() + " 计时器已经启动...");
        //调度任务 每 4 分钟执行一次
        timer.schedule(new MyTask() , 0 , 2*60*1000);
        //记录日志
        sce.getServletContext().log(new java.util.Date() + " 计时器执行一次!!!!!");
        sce.getServletContext().log (new java.util.Date() + " 计时器执行一次!!!!!");
    }
    //销毁应用之时，调用该方法
    public void contextDestroyed(ServletContextEvent sce)
    {
        //取消调度器
        timer.cancel();
        //记录日志
        sce.getServletContext().log(new java.util.Date() + " 计时器被销毁!!!");
        sce.getServletContext().log (new java.util.Date() + " 计时器被销毁!!!");
    }
}
```

由于使用了 JDK 的 Timer 作为任务调度器，因此必须实现如下任务类：

```
public class MyTask extends TimerTask
{
    private static boolean isRunning = false;
    //继承 TimerTask 抽象类，必须实现 run 方法，该方法的方法体就是调度的任务
    public void run()
    {
        //如果任务还没有开始运行
        if (!isRunning)
        {
            //任务开始运行
            isRunning = true;
            //在控制台输出当前时间
            System.out.println(new java.util.Date() + "      任务开始");
            //执行任务
            for (int i = 0 ; i < 100 ; i++)
        }
    }
}
```

```
{  
    System.out.println(new java.util.Date() + " 任务完成" + i +  
    "/" + 100);  
    try  
    {  
        Thread.sleep(80);  
    }  
    catch (Exception e)  
    {  
        e.printStackTrace();  
    }  
}  
//任务完成, 将任务运行旗标设为 flase  
isRunning = false;  
System.out.println(new java.util.Date() + "      所有任务完成!");  
}  
//如果任务已经在运行, 即使获得调度, 任务直接退出。  
else  
{  
    System.out.println(new java.util.Date() + "      任务退出!!!!");  
}  
}  
}  
}
```

## 2.12.2 配置 Listener

正如 load-on-startup Servlet 一样, Listener 用于启动 Web 应用的后台服务程序, 但不负责处理及响应用户请求, 因此无须配置 URL。

若将 Listener 配置在 Web 容器中 (如果 Web 容器支持 Listener), 则 Listener 将随 Web 应用的启动而启动。

配置 Listener 时使用<listener>元素, 下面是配置 Listener 的片段:

```
<!-- 配置 Listener-->  
<listener>  
    <!-- 指定 Listener 的实现类-->  
    <listener-class>lee.ScheduleListener</listener-class>  
</listener>
```

在上面的配置中, 既无须配置 Listener 的名字, 也无须配置 Listener 的 URL, 只需配置它的实现类即可。此时容器将自动检测部署在容器中的 Listener, 并在应用启动时, 自动加载所有的 Listener。

## 2.13 JSP 2.0 的新特性

2003 年发布的 JSP 2.0 是 JSP 1.2 的升级版, 新增了一些额外的特性。JSP 2.0 的目标是使动态网页的设计更加容易, 甚至可以无须学习 Java, 即可做出 JSP 页面, 从而更好地支持团队开发。

相比 JSP 1.2, JSP 2.0 主要增加了如下新特性。

- 表达式语言。
- 简化的自定义标签 API。
- Tag 文件语法。
- 除此之外，还可以在 web.xml 文件中配置 JSP 属性。这些属性与以前使用 page 指令的效果相同，可避免每个页面需要重复使用 page 指令定义。

如果需要使用 JSP 2.0 语法，其 web.xml 文件必须使用 Servlet 2.4 以上版本的配置文件，Servlet 2.4 以上版本的配置文件格式如下：

```
<?xml version="1.0" encoding="GBK"?>
<!-- 不再使用 DTD，而是使用 Schema 描述，版本也升级为 2.4-->
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.
  com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <!-- 此处配置 Web 应用的配置-->
</web-app>
```

### 2.13.1 JSP 定义

JSP 属性定义使用<jsp-property-group>元素配置，主要包括如下四个方面。

- 是否允许使用表达式语言：使用<el-ignored/>元素确定，默认值为 false。
- 是否允许使用 Java 脚本：使用<scripting-invalid/>元素确定，默认值为 false。
- 声明 JSP 页面的编码：使用<page-encoding/>元素确定。
- 使用隐式包含：使用<include-prelude/>和<include-coda/>元素确定。

看下面的配置文件：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Servlet2.4 以上版本的 Web 应用配置的根元素-->
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com
  /xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <!-- 关于 JSP 的配置信息-->
  <jsp-config>
    <jsp-property-group>
      <!-- 对哪个文件应用配置-->
      <url-pattern>/test1.jsp</url-pattern>
      <!-- 忽略表达式语言-->
      <el-ignored>true</el-ignored>
      <!-- 页面编码方式-->
      <page-encoding>GBK</page-encoding>
      <!-- 不允许使用 Java 脚本-->
      <scripting-invalid>true</scripting-invalid>
      <!-- 隐式导入页面头-->
      <include-prelude>/inc/top.jspf</include-prelude>
      <!-- 隐式导入页面尾-->
      <include-coda>/inc/bottom.jspf</include-coda>
    </jsp-property-group>
  </jsp-config>
</web-app>
```

```
<jsp-property-group>
    <!-- 对哪个文件应用配置-->
    <url-pattern>/test2.jsp</url-pattern>
    <el-ignored>false</el-ignored>
    <!-- 页面编码方式-->
    <page-encoding>GBK</page-encoding>
    <!-- 允许使用 Java 脚本-->
    <scripting-invalid>false</scripting-invalid>
</jsp-property-group>
</jsp-config>
</web-app>
```

注意：如果在不允许使用 Java 脚本的页面中使用 Java 脚本，则页面将出现错误。即 test1.jsp 页面中不允许出现 Java 脚本。

看下面的 JSP 页面代码，该页面是 test1.jsp：

```
<html>
    <head>
        <title>页面配置 1</title>
    </head>
    <body>
        <h2>页面配置 1</h2>
        下面是表达式语言输出：<br>
        ${1 + 2}
    </body>
</html>
```

在 web.xml 文件中，test1.jsp 页面配置了隐式导入，而且页面会忽略表达式，在浏览器中浏览该页面的效果如图 2.32 所示。

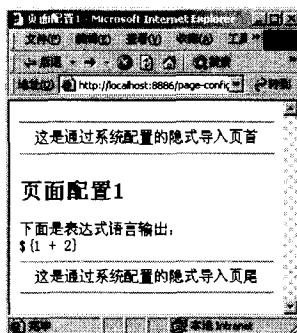


图 2.32 页面配置的运行效果

从图中可以看出，test1.jsp 的表达式语言不能正常输出，因为系统忽略了表达式语言的效果。

因此，在 test2.jsp 页面中使用隐式导入，不但可以使用表达式语言，也可以使用 Java 脚本，页面代码如下：

```
<html>
    <head>
        <title>页面配置 2</title>
    </head>
```

```

<body>
    <h2>页面配置 2</h2>
    下面是表达式语言输出: <br>
    ${1 + 2}<br>
    下面是小脚本输出: <br>
    <%out.println("hello Java");%>
</body>
</html>

```

页面运行效果如图 2.33 所示。

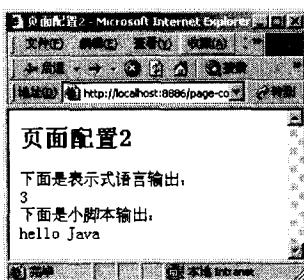


图 2.33 页面配置运行效果

此时，该页面允许使用表达式语言，页面正常输出了表达式语言的值。另外，也可以使用 Java 脚本，但没有隐式导入，所以没有页头和页尾。

## 2.13.2 表达式语言

表达式语言（Expression Language）是一种简化的数据访问方式。使用表达式语言可以以标记格式方便地访问 JSP 的隐含对象和 JavaBeans 组件，在 JSP 2.0 中，建议尽量使用表达式语言使 JSP 文件的格式一致，避免使用 Java 脚本。

表达式语言可用于简化 JSP 页面的开发，允许美工设计人员使用表达式语言的语法获取业务逻辑传过来的变量值。

**注意：**表达式语言是 JSP 的一个重要特性，它并不是一种通用的程序语言，而仅仅是一种数据访问语言，可以方便地访问应用程序数据，避免使用 Java 脚本。

表达式语言的语法格式是：

`${expression}`

### 1. 表达式语言使用算术和逻辑运算符

表达式语言支持的算术运算符非常多，在 Java 语言里支持的算术运算符，表达式语言都可以使用。

看下面的 JSP 页面：

```

<%@ page contentType="text/html; charset=gb2312"%>
<html>
    <head>

```

```
<title>表达式语言 - 算术运算符</title>
</head>
<body>
    <h2>表达式语言 - 算术运算符</h2>
    <hr>
    <table border="1" bgcolor="aaaadd">
        <tr>
            <td><b>表达式语言</b></td>
            <td><b>计算结果</b></td>
        </tr>
        <!-- 直接输出常量-->
        <tr>
            <td>\${1}</td>
            <td>\${1}</td>
        </tr>
        <!-- 计算加法-->
        <tr>
            <td>\${1.2 + 2.3}</td>
            <td>\${1.2 + 2.3}</td>
        </tr>
        <!-- 计算减法-->
        <tr>
            <td>\${1.2E4 + 1.4}</td>
            <td>\${1.2E4 + 1.4}</td>
        </tr>
        <!-- 计算乘法-->
        <tr>
            <td>\${21 * 2}</td>
            <td>\${21 * 2}</td>
        </tr>
        <!-- 计算除法-->
        <tr>
            <td>\${3 / 4}</td>
            <td>\${3 / 4}</td>
        </tr>
        <!-- 计算除法-->
        <tr>
            <td>\${3 % 4}</td>
            <td>\${3 % 4}</td>
        </tr>
        <!-- 计算求余-->
        <tr>
            <td>\${10 mod 4}</td>
            <td>\${10 mod 4}</td>
        </tr>
```

```

</tr>
<!-- 计算三目运算符--&gt;
&lt;tr&gt;
&lt;td&gt;\${(1==2) ? 3 : 4}&lt;/td&gt;
&lt;td&gt;\${(1==2) ? 3 : 4}&lt;/td&gt;
&lt;/tr&gt;
&lt;/table&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

该页面运行了基本表达式运算符号，运行的效果如图 2.34 所示。



图 2.34 算数运算的表达式语言

**注意：**如需要在支持表达式语言的页面中正常输出“\$”符号，则在“\$”符号前加转义字符“\”，否则系统以为“\$”是表达式语言的标记。

下面是逻辑运算符的 JSP 页面代码：

```

<%@ page contentType="text/html; charset=gb2312"%>
<html>
<head>
<title>表达式语言 - 逻辑运算符</title>
</head>
<body>
<h2>表达式语言 - 逻辑运算符</h2>
<hr>
数字之间的比较:
<table border="1" bgcolor="aaaaadd">
<tr>
<td><b>表达式语言</b></td>
<td><b>计算结果</b></td>
</tr>
<!-- 直接比较两个数字--&gt;
&lt;tr&gt;
&lt;td&gt;\${1 &lt; 2}&lt;/td&gt;
&lt;td&gt;\${1 &lt; 2}&lt;/td&gt;
&lt;/tr&gt;
<!-- 使用 lt 比较运算符--&gt;
&lt;tr&gt;
</pre>

```

```
<tr>
    <td>\${1 lt 2}</td>
    <td>\${1 lt 2}</td>
</tr>
<tr>
    <td>\${1 > (4/2)}</td>
    <td>\${1 > (4/2)}</td>
</tr>
<!-- 使用 gt 比较运算符-->
<tr>
    <td>\${1 > (4/2)}</td>
    <td>\${1 > (4/2)}</td>
</tr>
<tr>
    <td>\${4.0 >= 3}</td>
    <td>\${4.0 >= 3}</td>
</tr>
<!-- 使用 ge 比较运算符-->
<tr>
    <td>\${4.0 ge 3}</td>
    <td>\${4.0 ge 3}</td>
</tr>
<tr>
    <td>\${4 <= 3}</td>
    <td>\${4 <= 3}</td>
</tr>
<!-- 使用 le 比较运算符-->
<tr>
    <td>\${4 le 3}</td>
    <td>\${4 le 3}</td>
</tr>
<tr>
    <td>\${100.0 == 100}</td>
    <td>\${100.0 == 100}</td>
</tr>
<!-- 使用 eq 比较运算符-->
<tr>
    <td>\${100.0 eq 100}</td>
    <td>\${100.0 eq 100}</td>
</tr>
<tr>
    <td>\${(10*10) != 100}</td>
    <td>\${(10*10) != 100}</td>
</tr>
<!-- 先执行运算，再进行比较运算，使用 ne 比较运算符-->
<tr>
    <td>\${(10*10) ne 100}</td>
    <td>\${(10*10) ne 100}</td>
</tr>
</table>
字符之间的比较:


|                |                |
|----------------|----------------|
| <b>表达式语言</b>   | <b>计算结果</b>    |
| \\${'a' < 'b'} | \\${'a' < 'b'} |


```

```

<tr>
    <td>\$('hip' &gt; 'hit')</td>
    <td>$('hip' > 'hit')</td>
</tr>
<tr>
    <td>\$('4' &gt; 3)</td>
    <td>$('4' > 3)</td>
</tr>
</table>
</body>
</html>

```

因此，表达式语言不仅可在数字与数字之间比较，还可在字符与字符之间比较，字符串的比较是根据其编码的数字来比较大小的。

## 2. 表达式语言的内置对象

使用表达式语言可以直接获取请求参数，可获取页面中某个 JavaBean 的属性值，获取请求头及获取 session 属性值等，这些都得益于表达式语言的内置对象。

表达式语言包含如下 11 个内置对象。

- **pageContext:** 代表该页面的 pageContext 对象，与 JSP 的 pageContext 内置对象相同。
- **pageScope:** 用于获取 page 范围的属性值。
- **requestScope:** 用于获取 request 范围的属性值。
- **sessionScope:** 用于获取 session 范围的属性值。
- **applicationScope:** 用于获取 application 范围的属性值。
- **param:** 用于获取请求的参数值。
- **paramValues:** 用于获取请求的参数值，与 param 的区别在于，该对象用于获取属性值为数组的属性值。
- **header:** 用于获取请求头的属性值。
- **headerValues:** 用于获取请求头的属性值，与 header 的区别在于，该对象用于获取属性值为数组的属性值。
- **initParam:** 用于获取请求 Web 应用的初始化参数。
- **cookie:** 用于获取应用的 Cookie 值。

看下面的 JSP 页面代码：

```

<%@ page contentType="text/html; charset=gb2312"%>
<html>
    <head>
        <title>表达式语言 - 内置对象</title>
    </head>
    <body>
        <h2>表达式语言 - 内置对象</h2>
        <hr>
        请输入你的名字:
        <!-- 通过表单提交请求参数-->
        <form action="implicit-objects.jsp" method="post">
            <!-- 通过${param['name']} 获取请求参数-->
            你的名字 = <input type="text" name="name" value="${param['name']}'>
            <input type="submit" value="提交">
        </form>
    </body>
</html>

```

```
<% session.setAttribute("user", "abc"); %>
<br>
<table border="1" bgcolor="aaaadd">
    <tr>
        <td><b>表达式语言</b></td>
        <td><b>计算结果</b></td>
    <tr>
        <!-- 使用两种方式获取请求参数值-->
        <td>\${param.name}</td>
        <td>\${param.name}&nbsp;</td>
    </tr>
    <tr>
        <td>\${param["name"]}</td>
        <td>\${param["name"]}&nbsp;</td>
    </tr>
    <!-- 获取请求头信息-->
    <tr>
        <td>\${header.host}</td>
        <td>\${header.host}</td>
    </tr>
    <tr>
        <td>\${header["accept"]}</td>
        <td>\${header["accept"]}</td>
    </tr>
    <!-- 获取初始化参数-->
    <tr>
        <td>\${initParam["author"]}</td>
        <td>\${initParam["author"]}</td>
    </tr>
    <!-- 获取 session 属性-->
    <tr>
        <td>\${sessionScope["user"]}</td>
        <td>\${sessionScope["user"]}</td>
    </tr>
</table>
</body>
</html>
```

该页面使用多个内置对象，来获取请求参数的值，请求头的信息，Web 应用的初始化参数值及 session 属性。页面的执行效果如图 2.35 所示。

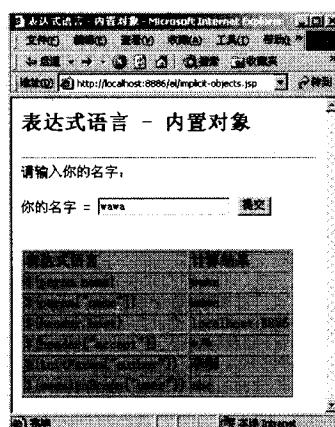


图 2.35 使用内置对象

### 3. 表达式语言的自定义函数

表达式语言除了可以使用基本的运算符外，还可以使用自定义函数。通过使用自定义函数，加强了表达式语言的功能。

自定义函数的用法非常类似于标签的用法，同样需要定义函数处理类和使用标签库。下面介绍定义函数的开发过程。

(1) 开发函数处理类：函数处理类就是普通类，这个普通类中包含若干个静态方法，每个静态方法都可定义成一个函数。

```
public class Functions
{
    //对字符串进行反转
    public static String reverse( String text )
    {
        return new StringBuffer( text ).reverse().toString();
    }
    //统计字符串的个数
    public static int countChar( String text )
    {
        return text.length();
    }
}
```

(2) 使用标签库定义函数，定义函数方法与定义标签库方法相同。下面是定义函数的配置文件：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- 标签库配置文件的根元素-->
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
         version="2.0">
    <jsp-version>2.0</jsp-version>
    <tlib-version>1.0</tlib-version>
    <!-- 定义第一个函数-->
    <function>
        <!-- 定义第一个函数:reverse-->
        <name>reverse</name>
        <!-- 定义函数的处理类-->
        <function-class>lee.Functions</function-class>
        <!-- 定义函数的对应的方法-->
        <function-signature>java.lang.String reverse( java.lang.String )</function-signature>
    </function>
    <function>
        <!-- 定义第二个函数: countChar -->
        <name>countChar</name>
        <!-- 定义函数的处理类-->
        <function-class>lee.Functions</function-class>
        <!-- 定义函数的对应的方法-->
        <function-signature>int countChar( java.lang.String )</function-signature>
    </function>
</taglib>
```

(3) 与自定义标签库相同，在 web.xml 文件中要增加标签库定义。

(4) 在 JSP 页面中使用函数时也需要先导入标签库，然后再使用函数。下面是使用函数的 JSP 页面片段：

```
<%@ page contentType="text/html; charset=gb2312"%>
<!-- 导入标签库-->
<%@ taglib prefix="my" uri="/tags/mytag"%>
<html>
    <head>
        <title>表达式语言 - 自定义函数</title>
    </head>
    <body>
        <h2>表达式语言 - 自定义函数</h2>
        <hr>
        请输入一个字符串：
        <form action="functions.jsp" method="post">
            字符串 = <input type="text" name="name" value="${param['name']}'>
            <input type="submit" value="提交">
        </form>
        <table border="1" bgcolor="aaaadd">
            <tr>
                <td><b>表达式语言</b></td>
                <td><b>计算结果</b></td>
            <tr>
                <td>\${param["name"]}</td>
                <td>\${param["name"]}&nbsp;</td>
            </tr>
            <!-- 使用 reverse 函数-->
            <tr>
                <td>\${my:reverse(param["name"])}</td>
                <td>\${my:reverse(param["name"])}&nbsp;</td>
            </tr>
            <tr>
                <td>\${my:reverse(my:reverse(param["name"]))}</td>
                <td>\${my:reverse(my:reverse(param["name"]))}&nbsp;</td>
            </tr>
            <!-- 使用 countChar 函数-->
            <tr>
                <td>\${my:countChar(param["name"])}</td>
                <td>\${my:countChar(param["name"])}&nbsp;</td>
            </tr>
        </table>
    </body>
</html>
```

**注意：**函数处理类的方法必须是 public static，因为这些方法是直接调用，无须实例化。

自定义函数，也提供了类似于自定义标签库的作用，使用简单的指令就可以完成复杂的功能。与自定义标签库不同的是，可以在表达式语言中直接使用函数。

### 2.13.3 简化的自定义标签

JSP 2.0 的自定义标签更加简单，无须重写烦琐的 doStartTag 和 doEndTag 等方法，

即使是带标签体的标签，也与不带标签体的标签处理方式完全相同，无须重写 doAfterBody 等方法，通常只需重写 doTag 方法。

下面以一个迭代器标签为例，介绍 JSP 2.0 的自定义标签的开发步骤。

### (1) 书写标签处理类。

JSP 2.0 的标签处理类继承 SimpleTagSupport 类，通常只需重写 doTag 方法，即使是带标签体的标签，通常也只需重写 doTag 方法。下面是标签处理类的代码：

```
//简单标签处理类，继承 SimpleTagSupport 类
public class MyIteratorTag extends SimpleTagSupport
{
    //标签属性
    private String bean;
    //标签属性必须的 setter 和 getter 方法
    public void setBean(String bean)
    {
        this.bean = bean;
    }
    public String getBean()
    {
        return bean;
    }
    //标签的处理方法，简单标签处理类只需要重写 doTag 方法
    public void doTag() throws JspException, IOException
    {
        //从 page scope 中获取名为 bean 的集合属性
        Collection<String> itemList = (Collection<String>)getJspContext().
            getAttribute(bean);
        //遍历集合
        for (String s : itemList)
        {
            //将集合的元素设置到 page 范围
            getJspContext().setAttribute("item", s );
            getJspBody().invoke(null);
        }
    }
}
```

### (2) 编写标签库文件。

标签库文件的定义与 JSP 1.2 的版本相似，通常需要定义标签名、标签处理类及标签属性等。区别是新的标签库定义文件采用 JSP 2.0 语法格式，下面是标签库定义的代码：

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" version="2.0">
    <!-- JSP 版本-->
    <jsp-version>2.0</jsp-version>
    <tlib-version>1.0</tlib-version>
    <!-- 定义第一个标签-->
    <tag>
        <!-- 定义标签名-->
        <name>helloWorld</name>
        <!-- 定义标签的处理类-->
        <tag-class>lee.HelloWorldSimpleTag</tag-class>
    
```

```
<!-- 定义标签体为空-->
<body-content>empty</body-content>
</tag>
<!-- 定义第二个标签-->
<tag>
    <!-- 定义标签名-->
    <name>iterator</name>
    <!-- 定义标签处理类-->
    <tag-class>lee.MyIteratorTag</tag-class>
    <!-- 定义标签体, 该标签体是不允许脚本的标签-->
    <body-content>scriptless</body-content>
    <!-- 定义标签属性-->
    <attribute>
        <name>bean</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
</tag>
</taglib>
```

(3) 在 web.xml 文件中, 定义了标签库 URI, 这个步骤与 JSP 1.2 的格式完全相同, 此处不再赘述。

(4) 在 JSP 页面中使用标签时, 同样也需要先导入标签库, 然后使用标签。

下面是使用标签的 JSP 页面代码:

```
<%@ page contentType="text/html; charset=gb2312" import="java.util.*"%>
<%@ taglib prefix="mytag" uri="/tags/mytag"%>
<html>
    <head>
        <title>JSP2.0 - 简单标签</title>
    </head>
    <body>
        <h2>JSP2.0 - 简单标签</h2>
        <hr>
        <%
            //创建一个 List 对象
            List<String> a = new ArrayList<String>();
            a.add("hello");
            a.add("world");
            a.add("java");
            //将 List 对象放入 page 范围内
            pageContext.setAttribute("a" , a);
        %>
        <table border="1" bgcolor="aaaadd">
            <!-- 使用迭代器标签, 对 a 集合进行迭代-->
            <mytag:iterator bean="a">
                <tr>
                    <td>${pageScope.item}</td>
                <tr>
            </mytag:iterator>
            </table>
            <!-- 使用简单标签-->
            <mytag:helloWorld/>
    </body>
</html>
```

页面执行效果如图 2.36 所示。



图 2.36 SimpleTagSupport 的迭代器标签

#### 2.13.4 Tag File 支持

Tag File 是自定义标签的简化用法，使用 Tag File 可以无须定义标签处理类和标签库文件，甚至无须在 web.xml 文件中定义标签库，但仍然可以在 JSP 页面中使用自定义标签。

下面以 Tag File 建立一个迭代器标签，其步骤如下。

(1) 建立 Tag 文件，在 Tag File 的自定义标签中，Tag File 代理了标签处理类，它的格式类似于 JSP 文件。可以这样理解：如使用 JSP 代替 Servlet 作为表现层一样，Tag File 使用了更简单的 Tag File 代替了标签处理类。

Tag File 具有以下 5 个编译指令。

- **taglib:** 作用与 JSP 文件中的 taglib 指令效果相同，用于导入其他标签库。
- **include:** 作用与 JSP 文件中的 include 指令效果相同，用于导入其他 JSP，或静态页面。
- **tag:** 作用类似于 JSP 文件中的 page 指令，有 pageEncoding、body-content 等属性，用于设置页面编码等属性。
- **attribute:** 用于设置自定义标签的属性，类似于自定义标签处理类中的标签属性。
- **variable:** 用于设置自定义标签的变量，这些变量将传回 JSP 页面使用。

下面是迭代器标签的 Tag File，这个 Tag File 的语法与 JSP 语法非常相似：

```
<%@ tag pageEncoding="GBK"%>
<!-- 使用 JSTL 的标签-->
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt"%>
<!-- 定义了四个标签属性-->
<%@ attribute name="color" %>
<%@ attribute name="bgcolor" %>
<%@ attribute name="title" %>
<%@ attribute name="bean" %>
<table border="1" bgcolor="\${color}" bordercolor="000000">
  <tr>
    <td><b>\${title}</b></td>
  </tr>
  <c:forEach items="\${bean}" var="var">
    <tr>
```

```

<td bgcolor="\${bgcolor}">
    \${var}
</td>
</tr>
</c:forEach>
</table>

```

这个 Tag File 的命名必须遵守如下规则： tagName.tag。即 Tag File 的文件名就是标签名，文件名后缀是 tag。将该文件存放在某个路径下，这个路径就是标签库路径。笔者将其放在 /WEB-INF/tags 下，即笔者的标签库路径为 /WEB-INF/tags。

(2) 在页面中使用自定义标签时，需要先导入标签库，再使用标签。使用标签与普通标签用法完全相似，只在导入标签时存在一些差异。由于此时的标签库没有 URI，只有标签库路径。因此导入标签时，使用如下语法格式：

```
<%@ taglib prefix="tagPrefix" tagdir="/WEB-INF/tags" %>
```

其中，prefix 与之前的 taglib 指令的 prefix 属性完全相同，用于确定标签前缀；而 tagdir 标签库路径下存放很多 Tag File，每个 Tag File 对应一个标签。

下面是使用标签的 JSP 页面代码：

```

<!-- 导入标签-->
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<%@ page contentType="text/html; charset=GBK" import="java.util.*" %>
<html>
    <head>
        <title>迭代器 tag file</title>
    </head>
    <body>
        <h2>迭代器 tag file</h2>
        <hr>
        <%
            //创建集合对象，用于迭代
            List<String> a = new ArrayList<String>();
            a.add("hello");
            a.add("world");
            a.add("java");
            //将集合对象放入页面范围
            pageContext.setAttribute("a", a);
        %>
        //使用自定义标签
        <tags:iterator color="#99dd99" bgcolor="#9999cc" title="迭代器" bean="\${pageScope.a}" />
    </body>
</html>

```

在该 JSP 页面中，使用了如下代码导入标签：

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
```

即以 tags 开头的标签，使用 /WEB-INF/tags 路径下的标签文件处理，而在 JSP 页面中使用如下代码标签：

```
<tags:iterator color="#99dd99" bgcolor="#9999cc" title="迭代器" bean="\${pageScope.a}" />
```

tags 表明该标签使用/WEB-INF/tags 路径下的 Tag File 处理标签；而 iterator 是标签名，对应在标签库路径下包含 iterator.tag 文件，由该文件负责处理标签。页面最终的执行效果如图 2.37 所示。



图 2.37 使用 Tag File 的迭代器标签

Tag File 是自定义标签的简化。事实上，就如同 JSP 文件会编译成 Servlet 一样，Tag File 也会编译成 Tag 处理类，自定义标签的后台依然由标签处理类完成，而这个过程由容器完成。使用 Tag File 进一步简化了自定义标签的开发。

## 本章小结

本章系统介绍了 JSP 的相关知识，内容覆盖了 JSP 所有知识点，包括：JSP 的三个编译指令，七个动作指令，九个内置对象，以及 JSP 的 Listener 和 Filter 的使用。

另外，详细介绍了 JSP 的自定义标签库的用法，包括简单标签的开发，带属性标签的开发，迭代器标签的开发等。

最后，系统介绍了 JSP 2.0 的知识，从 JSP 2.0 的配置讲起，详细介绍了表达式语言，简化的自定义标签 API，以及另一种使用 Tag File 开发自定义标签的方式。

# 第3章

## 经典 MVC 框架 Struts

### 本章要点

- 『 传统的 Model 1 和 Model 2
- 『 MVC 的基本知识
- 『 Struts 的基本知识
- 『 Struts 的下载和安装
- 『 Struts 的基本使用
- 『 Struts 的程序国际化
- 『 动态 FormBean
- 『 数据校验
- 『 异常处理框架
- 『 Struts 的常见扩展

从实际应用开发的角度而言，Struts 应该是 MVC 框架的第一选择。因为它具有稳定性，以及成熟的开发群体和丰富的信息资源，保证了企业应用的稳定开发。经过长达六年的发展，Struts 已经成长为稳定、成熟的框架，并且是所有 MVC 框架中应用最广的框架。

近来，WebWork 也加入到 Struts 阵营，更提高了 Struts 的竞争力。

如今，Struts 作为全世界第一个开源 MVC 框架，具有高度的成熟性和广泛的项目应用，保证了其应用的稳定性。

## 3.1 MVC 简介

MVC 架构的核心思想是：将程序分成相对独立，而又能协同工作的三个部分。通过使用 MVC 架构，可以降低模块之间的耦合，提供应用的可扩展性。另外，MVC 的每个组件只关心组件内的逻辑，不应与其他组件的逻辑混合。

MVC 并不是 Java 所独有的概念，而是面向对象程序都应该遵守的设计理念。

### 3.1.1 传统的 Model 1 和 Model 2

在 JSP 技术的发展初期，由于它便于掌握，以及可以快速开发的优点，很快就成了创建 Web 站点的热门技术。在早期的很多 Web 应用里，整个应用主要由 JSP 页面组成，辅以少量 JavaBean 来完成特定的重复操作。在这一时期，JSP 页面同时完成显示业务逻辑和流程控制。因此，开发效率非常高。

这种以 JSP 为主的开发模型就是 Model 1。其应用具体的实现方法如图 3.1 所示。

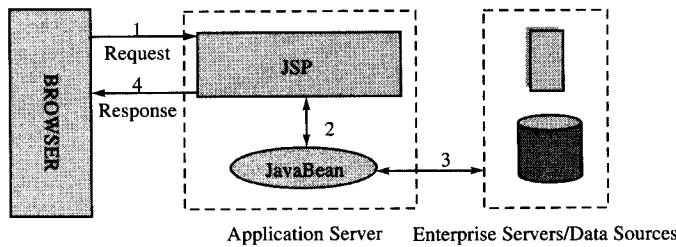


图 3.1 Model 1 模型图

在 Model 1 中，JSP 页面接收处理客户端请求，对请求处理后直接作出响应。其间可以辅以 JavaBean 处理相关业务逻辑。

Model 1 这种模式的实现比较简单，适用于快速开发小规模项目。但从工程化的角度看，它的局限性非常明显：JSP 页面身兼 View 和 Controller 两种角色，将控制逻辑和表现逻辑混杂在一起，从而导致代码的重用性非常低，增加了应用的扩展性和维护的难度。

Model 2 已经是基于 MVC 架构的设计模式。在 Model 2 架构中，Servlet 作为前端控制器，负责接收客户端发送的请求，在 Servlet 中只包含控制逻辑和简单的前端处理；然后，调用后端 JavaBean 来完成实际的逻辑处理；最后，转发到相应的 JSP 页面处理显示逻辑。其具体的实现方式如图 3.2 所示。

由于引入了 MVC 模式，使 Model 2 具有组件化的特点，更适用于大规模应用的开发，但也增加了应用开发的复杂程度。原本需要一个简单的 JSP 页面就能实现的应用，在 Model 2 中被分解成多个协同工作的部分，则需花更多时间才能真正掌握其设计和实现过程。

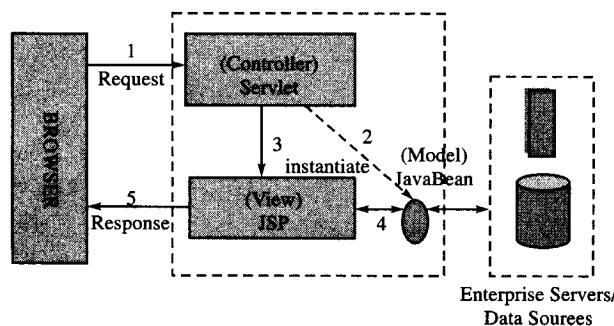
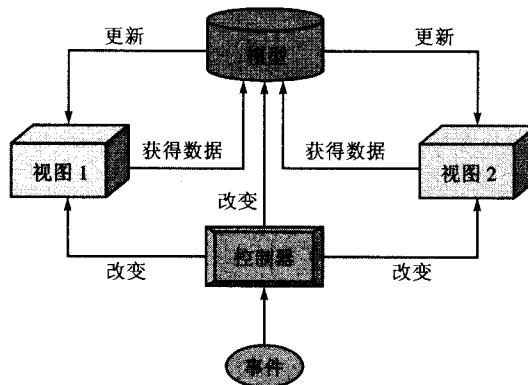


图 3.2 Model 2 模型图

### 3.1.2 MVC 及其优势

MVC 是 Model、View、Controller 三个词的缩写，三个词分别代表应用的三个组成部分：模型、视图与控制器。三个部分以最少的耦合，协同工作，从而提高应用的可扩展性及可维护性。

起初，MVC 模式是针对相同的数据需要不同显示的应用而设计的，其整体的效果如图 3.3 所示。



3.3 MVC 结构

在经典的 MVC 模式中，事件由控制器处理，控制器根据事件的类型改变模型或视图，反之亦然。具体地说，模型维护一个视图列表，这些视图为获得模型变化通知，通常采用观察者模式登记给模型。当模型发生改变时，模型向所有登记过的视图发送通知；接下来，视图从对应的模型中获得信息，然后更新自己。

概括起来，MVC 有如下特点。

- 多个视图可以对应一个模型。按 MVC 设计模式，一个模型对应多个视图，可以减少代码的复制及代码的维护量，一旦模型发生改变，也易于维护。
- 模型返回的数据与显示逻辑分离。模型数据可以应用任何的显示技术，例如使用

JSP 页面、Velocity 模板或者直接产生 Excel 文档等。

- 应用被分隔为三层，降低了各层之间的耦合，提供了应用的可扩展性。
- 控制层的概念也很有效，由于它把不同的模型和不同的视图组合在一起，完成不同的请求。因此，控制层可以说是包含了用户请求权限的概念。
- MVC 更符合软件工程化管理的精神。不同的层各司其职，每一层的组件具有相同的特征，有利于通过工程化和工具化产生管理程序代码。

## 3.2 Struts 概述

随着 MVC 模式的广泛使用，催生了 MVC 框架的产生。在所有的 MVC 框架中，出现最早，应用最广的就是 Struts 框架。

### 3.2.1 Struts 的起源

Struts 是 Apache 软件基金组织 Jakarta 项目的一个子项目，Struts 的前身是 Craig R.McClanahan 编写的 JSP Model 2 架构。

Struts 在英文中是“支架、支撑”的意思，这表明了 Struts 在 Web 应用开发中的巨大作用，采用 Struts 可以更好地遵循 MVC 模式。此外，Struts 提供了一套完备的规范，以及基础类库，可以充分利用 JSP/Servlet 的优点，减轻程序员的工作量，具有很强的可扩展性。

Struts 1.0 版本于 2001 年 6 月发布，目前最新的版本是此 Struts1.2.9。Struts 的作者 Craig R.McClanahan 参与了 JSP 规范制定以及 Tomcat 4 的开发，同时还领导制定了 J2EE 平台的 Web 层架构的规范。受此影响，Struts 框架一经推出，立即引起了 Java 开发者的广泛兴趣，并在全世界推广开来，最终成为世界上应用最广泛的 MVC 框架。

### 3.2.2 Struts 的体系结构

Struts 作为 MVC 模式的典型实现，对 Model、View 和 Controller 都提供了对应的实现组件，其具体的实现如图 3.4 所示。

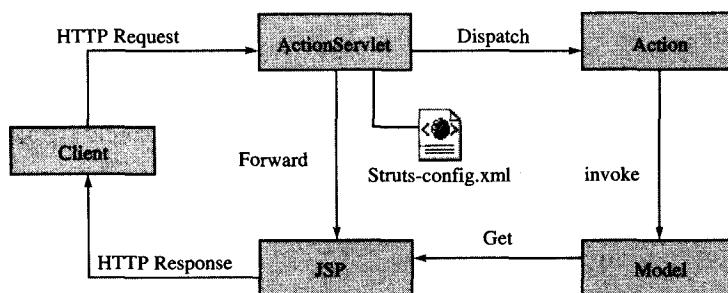


图 3.4 Struts 框架结构图

下面结合该图对 Struts 架构的工作原理简单介绍。

### 1. Model 部分

Struts 的 Model 部分由 ActionForm 和 JavaBean 组成。其中 ActionForm 用于封装用户请求参数，所有的用户请求参数由系统自动封装成 ActionForm 对象；该对象被 ActionServlet 转发给 Action；然后 Action 根据 ActionForm 里的请求参数处理用户请求。

而 JavaBean 则封装了底层的业务逻辑，包括数据库访问等。在更复杂的应用中，JavaBean 所代表的绝非一个简单的 JavaBean，可能是 EJB 组件或者其他业务逻辑组件。该 Model 对应图 3.4 的 Model 部分。

### 2. View 部分

Struts 的 View 部分采用 JSP 实现。Struts 提供了丰富的标签库，通过这些标签库可以最大限度地减少脚本的使用。这些自定义的标签库可以实现与 Model 的有效交互，并增加了显示功能。对应图 3.4 的 JSP 部分。

整个应用由客户端请求驱动，当客户端请求被 ActionServlet 拦截时，ActionServlet 根据请求决定是否需要调用 Model 处理用户请求，当用户请求处理完成后，其处理结果通过 JSP 呈现给用户。

### 3. Controller 部分

Struts 的 Controller 由两个部分组成。

- 系统核心控制器
- 业务逻辑控制器

其中，系统核心控制器对应图 3.4 中的 ActionServlet。该控制器由 Struts 框架提供，继承 HttpServlet 类，因此可以配置成一个标准的 Servlet。该控制器负责拦截所有 HTTP 请求，然后根据用户请求决定是否需要调用业务逻辑控制器，如果需要调用业务逻辑控制器，则将请求转发给 Action 处理，否则直接转向请求的 JSP 页面。

业务逻辑控制器负责处理用户请求，但业务逻辑控制器本身并不具有处理能力，而是调用 Model 来完成处理。业务逻辑控制器对应图 3.4 中的 Action 部分。

## 3.3 Struts 的下载和安装

Struts 目前的最新版本是 1.2.9，笔者所有的代码都基于该版本运行通过，建议读者也下载该版本。下载和安装 Struts 请按如下步骤进行。

(1) 在浏览器的地址栏输入 <http://mirror.vmmatrix.net/apache/struts/binaries/struts-1.2.9-bin.zip>，下载 Struts1.2.9。

(2) 将下载到 zip 文件解压缩，解压缩后有如下文件结构。

- contrib：包含了 Struts 表达式的依赖类库，如 JSTL 等类库。
- lib：包含 Struts 的核心类库，Struts 自定义标签库文件以及数据校验的规则文件等。该文件夹下的文件是 Struts 的核心部分。

- webapps：该文件夹下包含了几 WAR 文件，这些 WAR 文件都是一个 Web 应用，包含了 Struts 的说明文档及范例（struts-documentation 文件夹下包含了 Struts 的 API 文档，用户指南等文档，而 struts-examples 夹下则包含了 Struts 的各种简单范例）等。将这些文件解压缩。

- 其他 license 和 readme 等文档。

(3) 如果需要 Web 应用增加 Struts 的支持，则应该将 lib 文件夹下的 jar 文件全部复制到 Web 应用的 WEB-INF/lib 路径下。

(4) 如果需要使用 Struts 的标签库，应该将 lib 路径下的 TLD 文件复制到 Web 应用的 WEB-INF 路径下，并在 Web 应用的 web.xml 文件中配置对应的标签库。

(5) 如果需要使用 Struts 的数据校验，应将 lib 路径下的 validator-rules.xml 文件复制到 WEB-INF 路径下。

(6) 如果需要使用 Struts 表达式，则应将 contrib\struts-el\lib 路径下的 jar 文件复制到 WEB-INF 路径下，将对应的 TLD 文件也复制到 WEB-INF 路径下，并在 web.xml 文件中配置对应的标签库。

经过上面的步骤，Web 应用已经增加了 Struts 支持。但如果需要编译 Java 文件时能使用 Struts 的类库，则应将 lib 路径下的 struts.jar 文件添加到 CLASSPATH 的环境变量中即可。

## 3.4 Struts 入门

在讲解 Struts 的示例之前，先看一个简单的 MVC 示例。通过两种 MVC 的实现，读者可以看出，借助框架可以减少代码量，并可让程序开发更加规范化。

注意：MVC 只是 Struts 的一种实现方式，不使用 Struts 也可以使用 MVC。因为 MVC 是一种模式，而 Struts 则是一种实现。

### 3.4.1 基本的 MVC 示例

下面的范例也完全遵循 MVC 模式：其中 Model 由 JavaBean 充当；View 由 JSP 页面充当；而 Controller 则由 Servlet 充当。该示例是一个简单的登录流程。

#### 1. View 部分

示例首先进入一个等待用户输入的登录页面，该页面的源代码如下：

```
<%@ page language="java" contentType="text/html; charset=gb2312" errorPage="error.jsp"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>登录</title>
</head>
<script>
```

```
//Javascript 校验完成客户端校验
function check(form)
{
    //如果没有输入用户名
    if (form.username.value==null || form.username.value==" ")
    {
        alert('请输入用户名，然后再登录');
        return false;
    }
    //如果没有输入密码
    else if(form.pass.value==null || form.pass.value==" ")
    {
        alert('请输入密码，然后再登录');
        return false;
    }
    //两者都已经输入
    else
    {
        return true;
    }
}
</script>
<body>
<font color="red">
<%
//用于输出出错提示，出错提示保存在 request 的 err 属性里。
if (request.getAttribute("err") != null)
{
    out.println(request.getAttribute("err"));
}
%>
</font>
<!-- 下面是登录表单-->
请输入用户名和密码：
<form id="login" method="post" action="login" onsubmit="return check(this);">
    用户名: <input type="text" name="username"/><br>
    密 &nbsp;&nbsp;码: <input type="password" name="pass"/><br>
    <input type="submit" value="登录"/><br>
</form>
</body>
</html>
```

该页面是一个简单的 JSP 页面，包含了少量的 JSP 脚本，用于输出 Model 处理用户请求后返回的出错提示。

该页面的运行效果如图 3.5 所示。

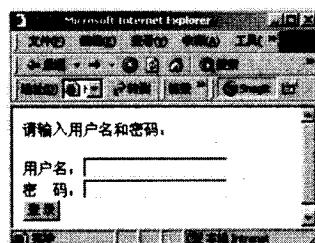


图 3.5 登录界面

该页面也包含了基本的客户端校验，如果没有输入用户名或者密码，就直接单击【登录】按钮，数据不会提交到服务器端，而是在客户端完成校验。客户端校验的效果如图 3.6 所示。

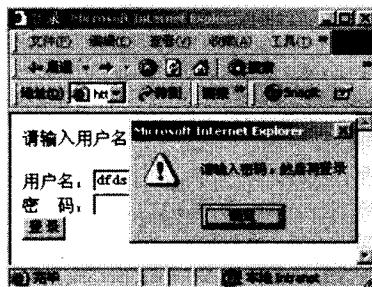


图 3.6 客户端校验的效果

登录成功后的页面非常简单，仅需要输出用户名。成功登录后的页面为 welcome.jsp，该页面的源代码如下：

```
<%@ page language="java" contentType="text/html; charset=gb2312" errorPage="error.jsp"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>登录成功</title>
</head>
<body>
<!-- 输出用户名-->
欢迎您, <%=session.getAttribute("name")%><br>
</body>
</html>
```

两个页面都指定了出错页面，即如果两个页面出现未捕捉异常，则自动跳转到 error.jsp 页面。error.jsp 页面则负责输出异常信息，error.jsp 页面的源代码如下：

```
<%@ page language="java" contentType="text/html; charset=gb2312" isErrorPage="true"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>系统错误</title>
</head>
<body>
<font color="red">
<!-- 输出出错提示-->
<%if(exception!=null)out.println(exception.getMessage() + "<br>");%>
</font>
</body>
</html>
```

从上面的源代码可看出，所有的 JSP 页面仅包含简单的显示逻辑，主要用于收集用户信息和显示系统处理信息，不会包含任何业务处理逻辑。

## 2. Controller 部分

该控制器是一个标准的 Servlet，负责拦截用户请求，解析用户请求参数，并调用合适的 Model，即 JavaBean 处理用户请求。

该 Servlet 的源代码如下：

```
//Servlet, 继承 HttpServlet
public class LoginServlet extends HttpServlet
{
    //Servlet 的服务响应方法
    public void service(ServletRequest request, ServletResponse response)
        throws ServletException, java.io.IOException
    {
        RequestDispatcher rd;
        //解析请求参数, 获取用户名
        String username = request.getParameter("username");
        //解析请求参数, 获取密码
        String pass = request.getParameter("pass");

        String errMsg = "";
        //完成服务器端校验
        //校验用户名
        if (username == null || username.equals(""))
        {
            errMsg += "您的用户名丢失或没有输入, 请重新输入";
        }
        //校验密码
        else if (pass == null || pass.equals(""))
        {
            errMsg += "您的密码丢失或没有输入, 请重新输入";
        }
        //如果用户名、密码都通过校验
        else
        {
            try
            {
                //调用 JavaBean, 创建 JavaBean 实例
                DbDao dd = DbDao.instance("com.mysql.jdbc.Driver",
                    "jdbc:mysql://localhost:3306/liuyan", "root", "32147");
                //调用 JavaBean 的方法
                ResultSet rs = dd.query("select password from user_table
                where username = '" + username + "'");
                //判断用户是否存在
                if (rs.next())
                {
                    //用户名存在, 密码符合, 则正常登录
                    if (rs.getString("password").equals(pass))
                    {
                        //向 Session 中存入用户名
                        HttpServletRequest hrequest = (HttpServletRequest)
                            request;
                        HttpSession session = hrequest.getSession(true);
                        session.setAttribute("name", username);
                        //跳转到成功后的 JSP 页面
                        rd = request.getRequestDispatcher("/welcome.jsp");
                        rd.forward(request, response);
                    }
                }
            }
        }
    }
}
```

```
        else
        {
            //用户名存在，但用户名密码不符合
            errMsg += "您的用户名密码不符合，请重新输入";
        }
    }
else
{
    //用户名不存在
    errMsg += "您的用户名不存在，请先注册";
}
}
catch (Exception e)
{
    //如果出现异常，跳转到 error.jsp 页面
    rd = request.getRequestDispatcher("/error.jsp");
    request.setAttribute("exception", "业务异常");
    rd.forward(request, response);
}
}
//如果出错提示不为空，表明无法正常登录，返回登录页面
if (errMsg != null && !errMsg.equals(""))
{
    rd = request.getRequestDispatcher("/login.jsp");
    request.setAttribute("err", errMsg);
    rd.forward(request, response);
}
}
```

该控制器通过调用 DbDao 来处理用户请求，而 DbDao 则是本系统中的 Model，负责持久层访问。但是该控制器存在少许问题，因为该控制器里出现了 JDBC API。在严格的 J2EE 架构分层里，这是不允许出现的。可将所有传到控制器的值都应封装成 VO(值对象)，而不应该与 JDBC API 耦合。

### 3. Model 部分

本范例的 Model 由于 DbDao 充当，该 DbDao 仅是一个数据库访问的 JavaBean，并不包含任何业务逻辑，对于严格的 J2EE 应用而言，该 JavaBean 更像一个 DAO 对象，而不是业务逻辑对象。

对于分层更清晰的 J2EE 应用，在 Model 下隐藏了更加清晰的分层：业务逻辑层及 DAO 层等。

下面是 DbBean 的源代码：

```
public class DbDao
{
    private static DbDao op;
    //数据库连接
    private Connection conn;
    //数据库驱动
    private String driver;
    //数据库服务的 url
    private String url;
    //数据库用户名
    private String username;
```

```
//数据库密码
private String pass;
//构造器私有，准备做成单态模式
private DbDao()
{
}
//带参数的构造器
private DbDao(String driver, String url, String username, String pass) throws
Exception
{
    this.driver = driver;
    this.url = url;
    this.username = username;
    this.pass = pass;
    Class.forName(driver);
    conn = DriverManager.getConnection(url,username,pass);
}
//各属性的 setter 方法
public void setDriver(String driver) {
    this.driver = driver;
}
public void setUrl(String url) {
    this.url = url;
}
public void setUsername(String username) {
    this.username = username;
}
public void setPass(String pass) {
    this.pass = pass;
}
//各属性的 getter 方法
public String getDriver() {
    return (this.driver);
}
public String getUrl() {
    return (this.url);
}
public String getUsername() {
    return (this.username);
}
public String getPass() {
    return (this.pass);
}
//获取数据库连接
public void getConnection() throws Exception
{
    if (conn == null)
    {
        Class.forName(this.driver);
        conn = DriverManager.getConnection(this.url,this.username,
            this.pass);
    }
}
//静态方法，返回 DbDao 实例
public static DbDao instance()
{
    if (op == null)
    {
```

```
        op = new DbDao();
    }
    return op;
}
//静态方法，返回 DbDao 实例
public static DbDao instance(String driver, String url, String username,
String pass) throws Exception
{
    if (op == null)
    {
        op = new DbDao(driver, url, username, pass);
    }
    return op;
}
//数据库访问操作，执行插入操作
public boolean insert(String sql) throws Exception
{
    //初始化数据库连接
    getConnection();
    //创建 Statement 对象
    Statement stmt = this.conn.createStatement();
    if (stmt.executeUpdate(sql) != 1)
    {
        return false;
    }
    return true;
}
//数据库访问操作，执行查询操作
public ResultSet query(String sql) throws Exception
{
    //初始化数据库连接
    getConnection();
    //创建 Statement 对象
    Statement stmt = this.conn.createStatement();
    return stmt.executeQuery(sql);
}
//数据库访问操作，执行删除操作
public void delete(String sql) throws Exception
{
    //初始化数据库连接
    getConnection();
    //创建 Statement 对象
    Statement stmt = this.conn.createStatement();
    stmt.executeUpdate(sql);
}
//数据库访问操作，执行更新操作
public void update(String sql) throws Exception
{
    //初始化数据库连接
    getConnection();
    //创建 Statement 对象
    Statement stmt = this.conn.createStatement();
    stmt.executeUpdate(sql);
}
```

从上面的示例中可看出 MVC 模式的基本结构及其应用流程。但也可以发现，基于 MVC 模式的开发，比单纯 JSP 的开发要复杂。

因此，使用框架，则可以大大减少代码的重复量，而且可以规范软件开发的行为。下面介绍采用 Struts 来完成相同的功能。

### 3.4.2 Struts 的基本示例

Struts 框架的应用使开发更加规范、统一。所有的控制器都由两部分组成——核心控制器与业务逻辑控制器。核心控制器负责拦截用户请求，而业务逻辑控制器则负责处理用户请求。

为了让核心控制器能拦截到所有的用户请求，应使用模式匹配的 Struts 的核心控制器 Servlet 的 URL。配置 Struts 的核心控制器，需要在 web.xml 文件中增加如下代码：

```
<!-- 将 Struts 的核心控制器配置成标准的 Servlet-->
<servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
<!-- 采用模式匹配来配置核心控制器的 URL-->
<servlet-mapping>
    <servlet-name>actionServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

从上面的配置可出，所有以.do 结尾的请求都会被 actionServlet 拦截，该 Servlet 由 Struts 提供，它将拦截到的请求转入 Struts 体系内。

Struts 的视图依然采用 JSP，该示例与前面 MVC 的示例并无太大区别，只需将 form 提交的 URL 改为 login.do 即可。以.do 结尾可以保证该请求被 Struts 的核心控制器拦截，其他并没有太多区别，此处不再赘述。

#### 1. Controller 部分

核心控制器 ActionServlet 由系统提供，负责拦截用户请求。

业务控制器用于处理用户请求，Struts 要求业务控制器继承 Action，下面是业务控制器 LoginAction 的源代码：

```
//业务控制器必须继承 Action
public class LoginAction extends Action
{
    //必须重写该核心方法，该方法负责处理用户请求
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //解析用户请求参数
        String username = request.getParameter("username");
        String pass = request.getParameter("pass");
        //出错提示
        String errMsg = "";
        //进行服务器端的数据校验
        if (username == null || username.equals(""))
        {
```

```

        errMsg += "您的用户名丢失或没有输入, 请重新输入";
    }
    else if(pass == null || pass.equals(""))
    {
        errMsg += "您的密码丢失或没有输入, 请重新输入";
    }
    else
    {
        //如果可以通过服务器端校验, 则调用 JavaBean 处理用户请求
        try
        {
            DbDao dd = DbDao.instance("com.mysql.jdbc.Driver",
                "jdbc:mysql://localhost:3306/liuyan","root","32147");
            ResultSet rs = dd.query("select password from user_table
                where username = '" +
                + username + "'");

            //判断用户名和密码的情况
            if (rs.next())
            {
                //如果用户名和密码匹配
                if (rs.getString("password").equals(pass))
                {
                    HttpSession session = request.getSession(true);
                    session.setAttribute("name" , username);
                    return mapping.findForward("welcome");
                }
                else
                {
                    //用户名和密码不匹配的情况
                    errMsg += "您的用户名密码不符合, 请重新输入";
                }
            }
            else
            {
                //用户名不存在的情况
                errMsg += "您的用户名不存在, 请先注册";
            }
        }
        catch (Exception e)
        {
            request.setAttribute("exception" , "业务异常");
            return mapping.findForward("error");
        }
    }
    if (errMsg != null && !errMsg.equals(""))
    {
        //如果出错提示不为空, 跳转到 input
        request.setAttribute("err" , errMsg);
        return mapping.findForward("input");
    }
    else
    {
        //否则跳转到 welcome
        return mapping.findForward("welcome");
    }
}
}

```

上面的控制器非常类似于 LoginServlet, 只是将 Servlet 中响应方法的 service 逻辑放

到了 Action 的 execute 方法中完成。

但注意到 execute 方法中除了包含 HttpServletRequest, HttpServletResponse 参数外, 还包括了两个类型的参数: ActionForm, ActionForward。这两个参数分别用于封装用户的请求参数和控制转发。可以注意到: Action 的转发无须使用 RequestDispatcher 类, 而是使用 ActionForward 完成转发。

注意: 业务控制器 Action 类, 应尽量声明成 public, 否则可能出现错误。并注意重写的 execute 方法, 其后面两个参数的类型是 HttpServletRequest 和 HttpServletResponse, 而不是 ServletRequest 和 ServletResponse。

## 2. Struts 的配置文件

在转发时也没有转向一个实际的 JSP 页面, 而是转向逻辑名 error, input, welcome 等。逻辑名并不代表实际的资源, 因此还必须将逻辑名与资源对应起来。

实际上, 此处的控制器没有作为 Servlet 配置在 web.xml 文件中, 因此必须将该 Action 配置在 Struts 中, 让 ActionServlet 了解将客户端请求转发给该 Action 处理。而这一切都是通过 struts-config.xml 文件完成的。

下面是 struts-config.xml 文件的源代码:

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Strut 配置文件的文件头, 包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- Struts 配置文件的根元素-->
<struts-config>
    <action-mappings>
        <!-- 配置 Struts 的 Action, Action 是业务控制器-->
        <action path="/login" type="lee.LoginAction" >
            <!-- 配置该 Action 的转发-->
            <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
            <!-- 配置该 Action 的转发-->
            <forward name="error" path="/WEB-INF/jsp/error.jsp"/>
            <!-- 配置该 Action 的转发-->
            <forward name="input" path="/login.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

从上面的配置文件可看出, Action 必须配置在 struts-config.xml 文件中。注意其中 Action 的 path 属性: /login, 再查看 login.jsp 登录 form 的提交路径 login.do。两个路径的前面部分完全相同, ActionServlet 负责拦截所有以.do 结尾的请求, 然后将.do 前面的部分转发给 struts-config.xml 文件中 Action 处理, 该 Action 的 path 属性与请求的.do 前面部分完全相同。

配置 Action 时, 还配置了三个局部 Forward。

- welcome: 对应/WEB-INF/jsp/welcome.jsp。
- error: 对应/WEB-INF/jsp/error.jsp。
- input: 对应/login.jsp。

后面部分还将讲到：Forward 有局部 Forward 和全局 Forward 两种，前者只对于某个 Action 有效，后者则对于整个 Action 都有效。

Action 使用 ActionMapping 控制转发时，只需转发到 Forward 的逻辑名，而无须转发到具体的资源，这样可避免将转发资源以硬编码的方式写在代码中，从而降低耦合。

**注意：**将 JSP 页面放在 WEB-INF 路径下，可以更好地保证 JSP 页面的安装。因为大多数 Web 容器不允许直接访问 WEB-INF 路径下的资源。因此，这些 JSP 页面不能通过超级链接直接访问，而必须使用 Struts 的转发才可以访问。

### 3.4.3 Struts 的流程

从上一节中读者已经了解了 Struts 的基本流程，下面对 Struts 的程序流程作更详细的讲解。

初学者往往容易被 Struts 的具体运行流程所迷惑，理解运行流程是理解 Struts 的基础，理解了 Struts 的运行流程，开发 Struts 的应用也就更简单了。

图 3.7 显示了从客户端发送请求到获得响应的全过程。

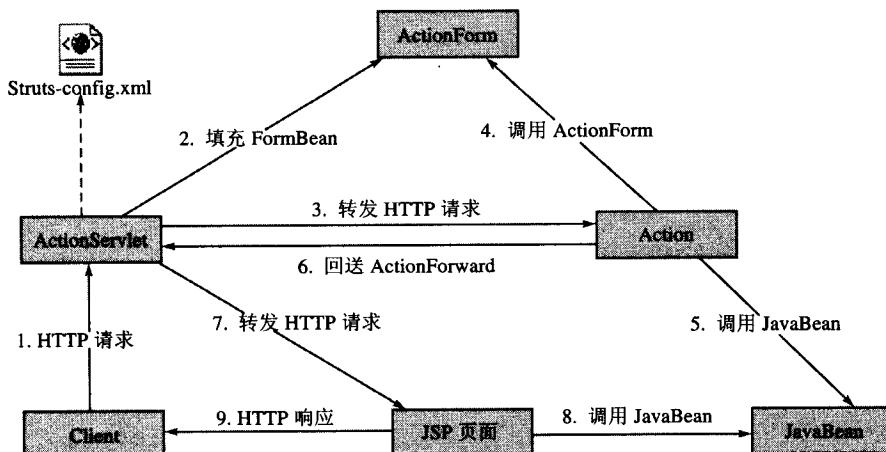


图 3.7 Struts 的程序流程

下面结合图 3.7 对 Struts 的工作流程作详细的讲解。

Web 应用都是请求-响应的程序结构。程序是由客户端 Client 发出 HTTP 请求开始的，客户端请求被 ActionServlet 拦截。在 ActionServlet 处，有两种情况：

- 要求逻辑控制器处理的请求；
- 简单转发的请求。

对于第一种的请求，ActionServlet 需要调用对应的 Action。因此 ActionServlet 将请求转发到 Action，如果请求还配置了对应的 FormBean，则 ActionServlet 还负责用请求参数填充 ActionForm。ActionForm 的实质就是 JavaBean，专门用于封装请求参数。

此时的 Action 将无须从 HTTP Request 中获取请求参数，而是从 ActionForm 中获得

请求参数。Action 获得请求参数后，调用 Model 对象由 JavaBean 处理用户请求。Action 处理完用户请求之后，将处理结果包装成 ActionForward，回送给 ActionServlet。

由于 ActionForward 对象封装了 JSP 资源的映射。因此，ActionServlet 知道调用合适的 JSP 资源表现给客户端。

对于第二种请求，HTTP 请求无须 Action 处理，只是对普通资源的请求，作为超级链接的替代。因为 ActionServlet 直接将该请求转发给 JSP 资源，既不会填充 ActionForm，也无须调用 Action 处理。

JSP 页面在表现之前，还需要调用对应的 JavaBean，此处的 JavaBean 不再是包含业务逻辑的 JavaBean，而是封装了处理结果的普通 VO（值对象）。

JSP 页面根据 VO 的值，可能利用 JSTL 或者 Struts 的标签库来生成 HTTP 响应给客户端。总之 JSP 应尽量避免使用 Java 脚本。

## 3.5 Struts 的配置

前面已经演示了 Struts 的 Action 的配置。但从 Struts 的流程中可以看出：Struts 有一个重要的对象——ActionForm。该对象用于封装用户的请求参数。

另外，还有一个 ActionForward 对象，该对象分全局 Forward 和局部 Forward 两种。

Action, ActionForm, ActionForward，这三个对象构成了 Struts 的核心。

Struts 最核心的控制器是 ActionServlet，该 Servlet 拦截用户请求，并将用户请求转入到 Struts 体系内。

### 3.5.1 配置 ActionServlet

ActionServlet 是一个标准的 Servlet，在 web.xml 文件中配置，该 Servlet 用于拦截所有的 HTTP 请求。因此，应将该 Servlet 配置成自启动 Servlet，即为该 Servlet 配置 load-on-startup 属性。

**注意：**在应用 Struts 的很多场景下，为 Servlet 配置 load-on-startup 属性都是必需的。因此，笔者建议应为 ServletAction 配置 load-on-startup 属性。

在 web.xml 文件中配置 ActionServlet 应增加如下片段：

```
<!-- 将 ActionServlet 配置成自启动的 Servlet-->
<servlet>
    <!-- 指定 Servlet 的名字-->
    <servlet-name>actionServlet</servlet-name>
    <!-- 指定该 Servlet 的实现类-->
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!-- 配置自启动的级别-->
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <!-- 配置 ActionServlet 的 URL 映射-->
```

```

<servlet-name>actionSevlet</servlet-name>
<!-- 所有以.do 结尾的请求由 ActionServlet 拦截-->
<url-pattern>*.do</url-pattern>
</servlet-mapping>

```

该 ActionServlet 作为一个标准 Servlet，配置在 Web 应用中，负责拦截用户请求。该 Servlet 还有加载 Struts 配置文件的责任。但这里并未告诉它如何加载 Struts 的配置文件，以及 Struts 的配置文件放在哪里及文件名是什么。

ActionServlet 默认加载 WEB-INF 下的 struts-config.xml 文件。如果需要 Struts 的配置文件不在 WEB-INF 路径下，或者改变了文件名，则应采用如下方式配置：

```

<servlet>
    <servlet-name>actionSevlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config-user.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

```

在上面的配置中，指定了 ActionServlet 的配置文件：struts-config-user.xml 文件，该文件作为 init-param 参数载入，载入时候指定了参数名： config。 config 是 Struts 固定的参数名，Struts 负责解析该参数，并加载该参数的指定的配置文件。

Struts 支持使用多个配置文件，当有多个配置文件时，应将不同的配置文件配置成不同的模块，并指定不同的 URI。下面的片段配置了两个配置文件：

```

<!-- 配置 ActionServlet-->
<servlet>
    <!-- ActionServlet 的名-->
    <servlet-name>actionSevlet</servlet-name>
    <!-- 配置 Servlet 的实现类-->
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!-- 指定 Struts 的第一个配置文件-->
    <init-param>
        <!-- 指定配置文件的映射-->
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config1.xml</param-value>
    </init-param>
    <!-- 指定 Struts 的第二个配置文件-->
    <init-param>
        <!-- 指定配置文件的映射-->
        <param-name>config/wawa</param-name>
        <param-value>/WEB-INF/struts-config2.xml</param-value>
    </init-param>
    <!-- 将 ActionServlet 配置成自启动 Servlet-->
    <load-on-startup>2</load-on-startup>
</servlet>

```

上面的配置片段中指定了两个配置文件：struts-config1.xml 和 struts-config2.xml 文件。这两个配置文件分别被配置到 config 和 config/wawa 的路径下。表明将 struts-config1.xml 中的 Action 映射到应用的根路径下，而 struts-config2.xml 文件中的 Action 则被映射到应用的 wawa 子路径下。也就是说 wawa 将作为系统的一个模块使用。

### 3.5.2 配置 ActionForm

配置 ActionForm，必须包含 ActionForm 类才行。Struts 要求 ActionForm 必须继承 Struts 的基类：org.apache.struts.action.ActionForm。ActionForm 的实现非常简单，该类只是一个普通的 JavaBean，只要为每个属性提供对应的 setter 和 getter 方法即可。

根据前面的讲解，ActionForm 用于封装用户的请求参数，而请求参数是通过 JSP 页面的表单域传递过来的。因此应保证 ActionForm 的参数与表单域的名字相同。

**注意：**JavaBean 的参数是根据 getter、setter 方法确定的。如果希望有一个 A 的属性，则应该提供 getA 和 setA 的方法。

下面使用 ActionForm 对前面的示例再次改写。

#### 1. ActionForm 的实现

ActionForm 的属性必须与 JSP 页面的表单域相同。本示例的表单包含如下两个表单域：

- username
- password

因此，ActionForm 必须继承 org.apache.struts.action.ActionForm，并为这两个域提供对应的 setter 和 getter 方法，下面是 ActionForm 的源代码：

```
//ActionForm 必须继承 Struts 的基类
public class LoginForm extends ActionForm
{
    private String username;
    private String password;
    //表单域 username 对应的 setter 方法
    public void setUsername(String username)
    {
        this.username = username;
    }
    //表单域 password 对应的 setter 方法
    public void setPassword(String password)
    {
        this.password = password;
    }
    //表单域 username 对应的 getter 方法
    public String getUsername()
    {
        return (this.username);
    }
    //表单域 password 对应的 getter 方法
    public String getPassword()
    {
        return (this.password);
    }
}
```

另外，该 ActionForm 的两个属性名可以不是 username 和 password。只要提供了 username 和 password 的 setter 和 getter 方法即可。

注意：FormBean 类应尽量声明成 public。

## 2. ActionForm 的配置

所有的 ActionForm 都被配置在 struts-config.xml 文件中，该文件包括了一个 form-beans 的元素，该元素内定义了所有的 ActionForm，每个 ActionForm 对应一个 form-bean 元素。

为了定义 LoginForm，必须在 struts-config.xml 文件中增加如下代码：

```
<!-- 用于定义所有的 ActionForm-->
<form-beans>
    <!-- 定义 ActionForm，至少指定两个属性：name , type-->
    <form-bean name="loginForm" type="lee.LoginForm"/>
</form-beans>
```

配置 ActionForm 非常简单，只需指定 ActionForm 的 name 属性即可。该属性定义了 ActionForm 的 id，用于标识该 Form；另外还需要一个 type 属性，该属性定义了 ActionForm 的实现类。

下面将介绍 Action 如何使用该 ActionForm，以及 Action 如何与该 ActionForm 关联。

### 3.5.3 配置 Action

Action 的配置比 ActionForm 相对复杂一点，因为 Action 负责管理与之关联的 ActionForm，它不仅需要配置实现类，还需要配置 Action 的 path 属性，该属性用于被用户请求。

对于只需在本 Action 内有效的 Forward，还应在 Action 元素内配置局部 Forward。

#### 1. Action 的实现

通过 ActionForm，可使 Action 无须从 HTTP 请求中解析参数。因为所有的参数都被封装在 ActionForm 中，下面是 Action 从 ActionForm 取得请求参数的源代码：

```
//Action 必须继承 Action 类
public class LoginAction extends Action
{
    //必须重写该核心方法，该方法 actionForm 将表单的请求参数封装成值对象
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //将 ActionForm 强制类型转换为 LoginForm
        LoginForm loginForm = (LoginForm) form;
        //从 ActionForm 中解析出请求参数：username
        String username = loginForm.getUsername();
        //从 ActionForm 中解析出请求参数：password
        String pass = loginForm.getPassword();
        //后面的处理与前一个示例的 Action 相同。
        ...
    }
}
```

该 Action 从转发过来的 ActionForm 中解析请求参数，对应的 ActionForm 则由 ActionServlet 在接收到用户请求时，负责实例化。

实际的过程是：ActionServlet 拦截到用户请求后，根据用户的请求，在配置文件中查找对应的 Action，Action 的 name 属性指定了用于封装请求参数的 ActionForm；然后 ActionServlet 将创建默认的 ActionForm 实例，并调用对应的 setter 方法完成 ActionForm 的初始化。

ActionServlet 在分发用户请求时，也将对应 ActionForm 的实例一同分发过来。

## 2. Action 的配置

Action 需要配置如下几个方面。

- Action 的 path：ActionServlet 根据该属性来转发用户的请求，即将用户请求转发与之同名的 Action。同名的意思是：将请求的.do 后缀去掉，匹配 Action 的 path 属性值。
- Action 的 name：此处的 name 属性并不是 Action 本身的名字，而是与 Action 关联的 ActionForm。因此该 name 属性必须是前面存在的 ActionForm 名。
- Action 的 type：该属性用于指定 Action 的实现类，也就是负责处理用户请求的业务控制器。
- 局部 Forward：Action 的转发并没有转发到实际的 JSP 资源，而是转发到逻辑名，即 Forward 名。在 Action 内配置的 Forward 都是局部 Forward（该 Forward 只在该 Action 内有效）。

下面是该 Action 的配置代码：

```
<!-- 该元素里配置所有的 Action-->
<action-mappings>
    <!-- 配置 Action，指定了 path, name, type 等属性-->
    <action path="/login" type="lee.LoginAction" name="loginForm">
        <!-- 配置局部 Forward-->
        <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
        <forward name="input" path="/login.jsp"/>
    </action>
</action-mappings>
```

### 3.5.4 配置 Forward

正如前面所讲，Forward 分局部 Forward 和全局 Forward 两种。前者在 Action 里配置，仅对该 Action 有效；后者单独配置，对所有的 Action 都有效。

配置 Forward 非常简单，主要需要指定以下三个属性。

- name：该 Forward 的逻辑名。
- path：该 Forward 映射到的 JSP 资源。
- redirect：是否使用重定向。

局部 Forward 作为 Action 的子元素配置；全局 Forward 配置在 global-forwards 元素里。

下面是配置全局 Forward 的代码：

```
<!-- 配置全局 Forward-->
<global-forwards>
    <!-- 配置 Forward 对象的 name 和 path 属性-->
    <forward name="error" path="/WEB-INF/jsp/error.jsp"/>
</global-forwards>
```

上面的配置代码中，配置了一个全局 Forward，该 Forward 可以被所有的 Action 访问。通常，只将全局资源配置成全局 Forward。

当每个 Action 在转发时，首先在局部 Forward 中查找与之对应的 Forward，如果在局部 Forward 中找不到对应的 Forward 对象，才会到全局 Forward 中查找。因此，局部 Forward 可以覆盖全局 Forward。

下面提供了该应用的 struts-config.xml 文件的全部源代码：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Struts 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- Struts 配置文件的根元素-->
<struts-config>
    <!-- 配置所有的 ActionForm-->
    <form-beans>
        <!-- 配置第一个 ActionForm，指定 ActionForm 的 name 和 type 属性-->
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 配置全局 Forward 对象-->
    <global-forwards>
        <!-- 该 Forward 对象的 name 属性为 error，映射资源为
            /WEB-INF/jsp/error.jsp -->
        <forward name="error" path="/WEB-INF/jsp/error.jsp"/>
    </global-forwards>
    <!-- 此处配置所有的 Action 映射-->
    <action-mappings>
        <!-- 配置 Action 的 path, type 属性
            name 属性配置 Action 对应的 ActionForm-->
        <action path="/login" type="lee.LoginAction" name="loginForm">
            <!-- 还配置了两个局部 Forward，这两个局部 Forward 仅对该 Action
                有效 -->
            <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
            <forward name="input" path="/login.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

## 3.6 Struts 程序的国际化

国际化是指应用程序运行时，可根据客户端请求来自的国家/地区、语言的不同而显示不同的界面。例如，如果请求来自于中文操作系统的客户端，则应用程序中的各种标签、错误提示和帮助等都使用中文；如果客户端使用英文操作系统，则应用程序能自动识别，并作出英文的响应。

引入国际化的目的是为了提供自适应、更友好的用户界面，而并未改变程序的逻辑

功能。国际化的英文单词是 Internationalization，有时也简称 I18N。其中 I 是这个单词的第一个字母，18 表示这个单词的长度，而 N 代表这个单词的最后一个字母。

Struts 的国际化也是基于 Java 的国际化的，下面先介绍 Java 程序的国际化。

### 3.6.1 Java 程序的国际化

Java 程序的国际化思路是将程序中的标签、提示等信息放在资源文件中，每个程序需要所有支持的国家\语言，都必须提供对应的资源文件。其资源文件是 key-value 对，每个资源文件中的 key 是不变的，但 value 则随不同国家\语言而变化。

Java 程序的国际化主要通过如下三个类完成。

- `java.util.ResourceBundle`: 对应用于加载一个资源包。
- `java.util.Locale`: 对应一个特定的国家/区域及语言环境。
- `java.text.MessageFormat`: 用于将消息格式化。

为了实现程序的国际化，必须先提供程序所需要的资源文件。资源文件的内容是和很多 key-value 对。其中 key 是程序使用的部分，而 value 则是程序界面的显示。

资源文件的命名可以有如下三种形式。

- `baseName_language_country.properties`。
- `baseName_language.properties`。
- `baseName.properties`。

其中 `baseName` 是资源文件的基本名，用户可以自由定义。而 `language` 和 `country` 都不可随意变化，必须是 Java 所支持的语言和国家。

#### 1. 国际化支持的语言和国家

事实上，Java 也不可能支持所有国家和语言，如需要获取 Java 所支持的语言和国家，可调用 `Locale` 类的 `getAvailableLocales` 方法来获取。该方法返回一个 `Locale` 数组，该数组里包含了 Java 所支持的语言和国家。

下面的程序简单地示范了如何获取 Java 所支持的国家和语言：

```
public class LocaleList
{
    public static void main(String[] args)
    {
        //返回 Java 所支持的全部国家和语言的数组
        Locale[] localeList = Locale.getAvailableLocales();
        //遍历数组的每个元素，依次获取所支持的国家和语言
        for (int i = 0; i < localeList.length ; i++)
        {
            //打印出所支持的国家和语言
            System.out.println(localeList[i].getDisplayCountry() + "=" +
                localeList[i].getCountry() + " " +
                localeList[i].getDisplayLanguage() + "=" +
                localeList[i].getLanguage());
        }
    }
}
```

程序的运行结果如图 3.8 所示。

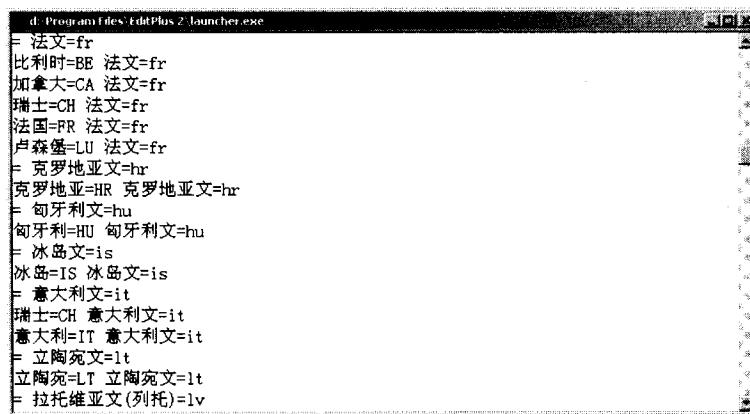


图 3.8 国际化所支持的语言和国家

## 2. 编写国际化所需的资源

国际化所需的资源文件内容是 key-value 对，下面提供了两个资源文件，这两个资源文件很简单，只包含一个 key-value 对。

下面是 MyResource.properties 的文件的内容：

```
//资源文件的内容: key-value 对。
msg=Hello, {0}! Today is {1}.
```

下面是 MyResource\_zh\_CN.properties 文件的内容：

```
//资源文件的内容: key-value 对
msg=你好, {0}! 今天是{1}。
```

所有资源文件的 key 都是相同的，只是 value 会随国家和语言的不同而变化。

对于所有的非西欧文字还必须使用 native2ascii 命令转化，该命令负责将非西欧文字转换成系统可以识别的文字。

因此必须对 MyResource\_zh\_CN.properties 文件进行转化。

命令的格式如下：

```
native2ascii MyResource_zh_CN.properties MyResource_x.properties
```

转换后新生成的文件内容会出现很多乱码，这是正常的。可将其转换成生成的 MyResource\_x.properties 文件重命名为 MyResource\_zh\_CN.properties 即可。

## 3. 完成程序国际化

上面的资源文件实质上有如下两种。

- MyResource\_zh\_CN.properties：指定了确定的国家和语言的资源文件。
- MyResource.properties：没有指定国家和语言的资源文件。

下面的程序可实现程序的国际化：

```
public class Hello
{
```

```
public static void main(String[] args)
{
    Locale currentLocale = null;
    //如果运行程序时指定了国家和语言参数，则以此创建 Locale 对象
    if (args.length == 2)
    {
        currentLocale = new Locale(args[0] , args[1]);
    }
    //否则，直接使用默认的国家和语言
    else
    {
        currentLocale = Locale.getDefault();
    }
    //根据 Locale 加载资源包
    ResourceBundle bundle = ResourceBundle.getBundle("MyResource" ,
    currentLocale);
    //根据 key 获取对应的资源
    String msg = (String)bundle.getObject("msg");
    //创建 MessageFormat 对象，用于获取格式化消息
    MessageFormat mf = new MessageFormat("");
    //创建 Locale 对象
    //设置国际化时所使用的 Locale 实例
    mf.setLocale(currentLocale);
    //设置需要格式化的消息
    mf.applyPattern(msg);
    Date now = new Date();
    //为消息中需要的参数指定值
    Object[] msgParams = {"yeeku",now};
    //输出国际化消息
    System.out.println(mf.format(msgParams));
}
```

如果运行时没有指定表示国家和语言的参数，则程序的运行效果如图 3.9 所示。

当然，笔者所运行的环境是简体中文的操作系统。

运行时，如果指定了两个参数：en Us，则表明用于显示美国英语的环境，图 3.10 显示了程序的运行效果。



图 3.9 没有指定参数时程序国际化的效果



图 3.10 指定参数时程序国际化的效果

但是，程序并没有指定 MyResource\_en\_US.properties 文件，程序从哪里获取资源呢？在 ResourceBundle 加载资源时按如下顺序搜索。

搜索所有国家和语言都匹配的资源文件，例如，对于简体中文的环境，先搜索如下文件：  
MyResource\_zh\_CN.properties

如果没有找到国家和语言都匹配的资源文件，则再搜索语言匹配的文件，即搜索如下文件：

MyResource\_zh.properties

如果上面的文件依然无法搜索到，则搜索 `baseName` 匹配的文件，即搜索如下文件：

`MyResource.properties`

#### 4. 使用类文件代替资源文件

Java 也允许使用类文件代替资源文件，即将所有的 key-value 对存入 class 文件，而不是属性文件。

用来代替资源文件的 Java 文件必须满足如下条件。

- 类的名字必须为 `baseName_language_country`，这与属性文件的命名相似。
- 该类必须继承 `ListResourceBundle`，并重写 `getContents` 方法，该方法返回 `Object` 数组，该数组的每一个项都是 key-value 对。

下面的类文件可以代替上面的属性文件：

```
public class MyResource_zh_CN extends ListResourceBundle
{
    //定义资源
    private final Object myData[][] =
    {
        {"msg", "{0}, 您好！今天是{1}"}
    };
    //重写方法 getContents()
    public Object[][] getContents()
    {
        //该方法返回资源的 key-value 对
        return myData;
    }
}
```

如果系统同时存在资源文件及类文件，则系统将以类文件为主，而不会调用资源文件。对于简体中文的 Locale,  `ResourceBundle` 搜索资源的顺序是：

- (1) `baseName_zh_CN.class`。
- (2) `baseName_zh_CN.properties`。
- (3) `baseName_zh.class`。
- (4) `baseName_zh.properties`。
- (5) `baseName.class`。
- (6) `baseName.properties`。

当系统按上面的顺序搜索资源文件时，如果前面的文件不存在，则会使用下一个；如果一直找不到对应的文件，系统将抛出异常。

### 3.6.2 Struts 的国际化

Struts 的国际化也是通过  `ResourceBundle` 完成的。因此，也必须编写资源文件。下面以前面的应用为例，演示如何实现程序的国际化。

#### 1. 编写资源文件

本示例程序能满足两种语言环境：简体中文和英语。当然，需要满足更多国家的语

言也不是问题，只需提供对应的资源文件即可。

下面是两份资源文件：

```
//英文的资源文件
username=username
pass=password
login=submit
noname=please enter name
nopass=please enter password
```

下面是中文的资源文件：

```
//中文的资源文件
username=用户名
pass=密码
login=登录
noname=请输入用户名，然后再登录
nopass=请输入密码，然后再登录
```

注意：对于非西欧文字的资源文件，必须使用 native2ascii 进行转换。

## 2. 加载资源文件

资源文件的加载通过 struts-config.xml 文件来配置，加载资源文件应从 Web 应用的 WEB-INF/classes 路径开始加载。因此，资源文件必须放在 WEB-INF/classes 路径或该路径的子路径下。如果直接放在 WEB-INF/classes 路径下，在配置资源文件时，直接指定资源文件的 baseName 即可。但如果放在子路径下，则必须以包的形式配置。

下面的示例程序中的资源文件放在 WEB-INF/classes/lee 下。配置资源文件的 struts-config.xml 文件的源代码如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!!-- Struts 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!!-- Struts 配置文件的根元素-->
<struts-config>
    <!-- 配置所有的 ActionForm-->
    <form-beans>
        <!-- 配置第一个 ActionForm，指定 ActionForm 的 name 和 type 属性-->
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 配置全局 Forward 对象-->
    <global-forwards>
        <!-- 该 Forward 对象的 name 属性为 error，映射资源为
            /WEB-INF/jsp/error.jsp -->
        <forward name="error" path="/WEB-INF/jsp/error.jsp"/>
    </global-forwards>
    <!-- 此处配置所有的 Action 映射-->
    <action-mappings>
        <!-- 配置 Action 的 path, type 属性
            name 属性配置 Action 对应的 ActionForm-->
        <action path="/login" type="lee.LoginAction" name="loginForm">
            <!-- 还配置了两个局部 Forward，这两个局部 Forward 仅对该 Action
                有效 -->
        </action>
    </action-mappings>
</struts-config>
```

```

<forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
<forward name="input" path="/login.jsp"/>
</action>
</action-mappings>
<!-- 配置国际化资源, parameter 指定资源文件的位置。-->
<message-resources parameter="lee.messages"/>
</struts-config>

```

Struts 负责加载资源文件, Struts 在应用启动时将加载该资源文件。

**注意:** 如果需要 Struts 实现程序国际化, 必须将 ActionServlet 配置成 load-on-startup 的 Servlet, 只有这样才可以保证在启动应用时加载该资源文件。

### 3. 使用 bean 标签显示国际化信息

根据前面的国际化示例程序我们知道, 程序要实现国际化, 则不能将标签及帮助等提示信息以硬编码方式写在程序中, 而应使用资源文件的 key。

Struts 提供了专门用于国际化的标签 bean, 关于 bean 标签, 将在后面深入讲解。此处仅介绍国际化支持使用的 bean 标签。

为了可以在 Web 应用中使用 bean 标签, 在应该将 struts-bean.tld 复制到 WEB-INF/ 路径下, 并在 web.xml 文件中增加该 Struts 标签库的配置。

下面的 web.xml 文件增加了 Struts struts-bean 标签库, 源代码如下:

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Web 应用的配置文件的文件头-->
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com
  /xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <!-- 配置 ActionServlet, 并配置成 load-on-startup 的 Servlet-->
  <servlet>
    <servlet-name>actionSevlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- 配置 ActionServlet 映射的 URL-->
  <servlet-mapping>
    <servlet-name>actionSevlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!-- 配置 struts-bean 的标签库-->
  <taglib>
    <!-- 配置标签库的 uri-->
    <taglib-uri>/tags/struts-bean</taglib-uri>
    <!-- 配置标签库对应的物理位置-->
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>
</web-app>

```

经过上面的配置, JSP 页面可以使用 bean 标签了, 从而可以通过 bean 标签显示国际化提示。

下面是 login.jsp 文件的源代码, 该文件中不再以硬编码的方式输出提示, 而是输出

的资源文件的 key:

```
<%@ page language="java" contentType="text/html; charset=gb2312" errorPage="" %>
<!-- 增加 struts-bean.xml 的标签库-->
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<!-- 使用国际化资源文件的 key 输出标题-->
<title><bean:message key="login"/></title>
</head>
<script>
function check(form)
{
    if (form.username.value==null || form.username.value==" ")
    {
        <!-- 使用国际化资源文件的 key 输出提示-->
        alert('<bean:message key="noname"/>');
        return false;
    }
    else if(form.pass.value==null || form.pass.value==" ")
    {
        <!-- 使用国际化资源文件的 key 输出提示-->
        alert('<bean:message key="nopass"/>');
        return false;
    }
    else
    {
        return true;
    }
}
</script>
<body>
<font color="red">
<%
if (request.getAttribute("err") != null)
{
    out.println(request.getAttribute("err"));
}
%>
</font>
<form id="login" method="post" action="login.do" onsubmit="return check(this);">
    <!-- 使用国际化资源文件的 key 输出用户名标签-->
    <bean:message key="username"/><input type="text" name="username"/><br>
    <!-- 使用国际化资源文件的 key 输出密码标签-->
    <bean:message key="pass"/><input name="pass"/><br>
    <!-- 使用国际化资源文件的 key 输出登录按钮-->
    <input type="submit" value='<bean:message key="login"/>' /><br>
</form>
</body>
</html>
```

该页面运行效果如图 3.11 所示。

因此，实现国际化只需要编写资源文件，然后使用 bean 标签输出国际化信息。使用 bean 标签，可以大大降低程序国际化的复杂度，这就是使用框架的优势。

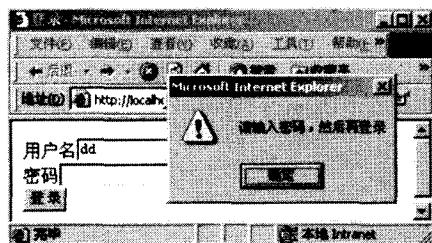


图 3.11 Struts 的程序国际化效果

## 3.7 使用动态 ActionForm

Struts 的 ActionForm 虽然比较简单，但也是异常烦琐的类。说简单，是因为每个类的写法非常简单，只需要为每个表单域提供对应的 setter 和 getter 方法即可。说烦琐，是因为必须大量书写这种简单的类。

好在 Struts 提供了动态 ActionForm，通过使用动态 ActionForm，可以完全不用书写 ActionForm，只需在 struts-config.xml 文件中配置即可。

### 3.7.1 配置动态 ActionForm

所有的动态 ActionForm 的实现类都必须是 org.apache.struts.action.DynaActionForm 类，或者是它的子类。

使用动态 ActionForm 与前面不同的是：因为系统不清楚动态 ActionForm 的属性，所以必须在配置文件中配置对应的属性。可以使用 form-property 元素来配置动态 ActionForm 的属性。

下面是使用动态 ActionForm 的 struts-config.xml 文件的源代码：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Struts 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- Struts 配置文件的根元素-->
<struts-config>
    <form-beans>
        <!-- 配置动态 ActionForm，动态 ActionForm 必须使用 DynaActionForm -->
        <form-bean name="loginForm" type="org.apache.struts.action.
DynaActionForm">
            <!-- 配置 ActionForm 的属性：username-->
            <form-property name="username" type="java.lang.String"/>
            <!-- 配置 ActionForm 的属性：pass-->
            <form-property name="pass" type="java.lang.String"/>
        </form-bean>
    </form-beans>
    <!-- 配置全局 Forward-->
    <global-forwards>
```

```

<forward name="error" path="/WEB-INF/jsp/error.jsp"/>
</global-forwards>
<!-- 配置 action 映射-->
<action-mappings>
    <!-- 配置 Action 的 path,type,name 属性-->
    <action path="/login" type="lee.LoginAction" name="loginForm">
        <!-- 配置两个局部 Forward-->
        <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
        <forward name="input" path="/login.jsp"/>
    </action>
</action-mappings>
<!-- parameter 属性确定资源文件的文件名， 默认在 WEB-INF/classes 下查找文件 -->
<message-resources parameter="messages"/>
</struts-config>

```

从上面的配置文件可看出，动态 `ActionForm` 的配置必须增加 `form-property` 元素，每个属性必须对应一个 `form-property` 元素。

`form-property` 元素包含两个属性。

- `name`: 属性的名字，必须与 JSP 页面的表单域的名字相同。
- `type`: 属性的类型。

### 3.7.2 使用动态 `ActionForm`

动态 `ActionForm` 同样用于封装用户请求参数，但该 `ActionForm` 没有各属性的 `getter` 及 `setter` 方法，因此无法调用对应的 `getter` 方法来解析请求参数。

幸好 `DynaActionForm` 提供了多个重载的 `getter` 方法用于获取请求参数，下面的 `Action` 使用 `DynaActionForm`，该 `Action` 的源代码如下：

```

//Action 必须继承 Action 类
public class LoginAction extends Action
{
    //必须重写该核心方法，该方法 actionForm 将表单的请求参数封装成值对象
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //将 ActionForm 强制类型转换为 DynaActionForm
        DynaActionForm loginForm = (DynaActionForm)form;
        //从 ActionForm 中解析出请求参数: username
        String username = (String)loginForm.get("username");
        //从 ActionForm 中解析出请求参数: password
        String pass = (String)loginForm.get("pass");

        //后面的处理与前一个示例的 Action 相同。
        ...
    }
}

```

使用动态 `ActionForm` 时，请求参数必须使用 `DynaActionForm` 的 `getter` 方法获取。`DynaActionForm` 的 `getter` 方法主要有如下三个。

- `Object get(java.lang.String name)`: 根据属性名返回对应的值。

- `Object get(java.lang.String name, int index)`: 对于有多个重名表单域的情况, Struts 将其当成数组处理, 此处根据表面域名和索引获取对应值。
- `Object get(java.lang.String name, java.lang.String key)`: 对于使用 Map 属性的情况, 根据属性名及对应 key, 获取对应的值。

使用动态 ActionForm 的目的是为了减少代码的书写量。但有些 IDE 工具可以自动生成 ActionForm, 则可以使用普通 ActionForm。

动态 ActionForm 与普通 ActionForm 并没有太大的区别。动态 ActionForm 避免了编写 ActionForm, 但配置变得更复杂了。而普通 ActionForm 使解析请求参数变得更直观。

## 3.8 Struts 的标签库

Struts 提供了大量的标签库, 用于完成表现层的输出。借助于 Struts 的标签库, 可避免在 JSP 中嵌入大量的 Java 脚本, 从而提高代码的可读性。

Struts 主要提供了如下三个标签库。

- `html`: 用于生成 HTML 的基本标签。
- `bean`: 用于完成程序国际化, 输出 Struts 的 ActionForm 的属性值等。
- `logic`: 用于完成循环、选择等流程控制。

### 3.8.1 使用 Struts 标签的基本配置

为了使用 Struts 标签, 必须将 struts-bean.tld 文件复制到 WEB-INF 路径下, 并在 web.xml 文件中增加 html 标签库的配置。

下面是增加了三个标签库配置的 web.xml 文件:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Web 配置文件的文件头-->
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<!-- Web 配置文件的根元素-->
<web-app>
    <!-- 配置 ActionServlet -->
    <servlet>
        <servlet-name>action</servlet-name>
        <!-- 配置 Servlet 的实现类-->
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <!-- 将 ActionServlet 配置 load-on-startup Servlet-->
        <load-on-startup>2</load-on-startup>
    </servlet>
    <!-- 配置 ActionServlet 映射的 URL -->
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!-- 配置 Web 应用的首页 -->
    <welcome-file-list>
```

```
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>
<!-- 配置 html 标签库-->
<taglib>
    <!-- 配置 html 标签库的 URI-->
    <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
    <!-- 指定 html 标签库的物理位置-->
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<!-- 配置 bean 标签库-->
<taglib>
    <!-- 配置 bean 标签库的 URI-->
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <!-- 指定 bean 标签库的物理位置-->
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<!-- 配置 logic 标签库-->
<taglib>
    <!-- 配置 logic 标签库的 URI-->
    <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
    <!-- 指定 logic 标签库的物理位置-->
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
</web-app>
```

因为每个页面都需要使用这三个标签，为避免在每个页面中重复使用 taglib 标签，可将三个标签的导入放在单独的文件中。

下面是 taglib.jsp 文件的源代码：

```
<%@ 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" %>
```

其他的 JSP 页面只需使用 include 包含该页面，即可导入三个标签库，避免了重复使用 taglib 标签。

### 3.8.2 使用 html 标签库

Struts 为 html 的大部分标签提供了对应的 html 标签，html 所支持的标签大致有如下。

- **base**: 表现成一个 HTML 的<base>标签。
- **button**: 表现成一个按钮，该按钮默认没有任何动作。
- **cancel**: 表现成一个取消按钮。
- **checkbox**: 表现成一个 Checkbox 的输入框。
- **error**: 用于输出数据校验的出错提示。
- **file**: 表现成一个文件浏览输入框。
- **form**: 表现成一个 form 域。
- **frame**: 表现成一个 HTML<frame>标签。
- **hidde**: 表现成一个隐藏域。
- **html**: 表现成 HTML 的<html>标签。
- **image**: 表现成表单域的 image 域。

- **img:** 表现成一个 HTML 的 img 标签。
- **javascrip:** 表现成 JavaScript 的校验代码, 这些校验代码根据 ValidatorPlugIn 生成。
- **link:** 表现成 HTML 的超级链接。
- **messages:** 用于输出 Struts 的各种提示信息, 包括校验提示。
- **multibox:** 表现成一个 Checkbox 输入框。
- **option:** 表现成选择框的一个选项。
- **password:** 表现成一个密码输入框。
- **radio:** 表现成一个单选输入框。
- **reset:** 表现成一个重设按钮。
- **rewrite:** 表现成一个 URL。
- **select:** 表现成一个列表选择框。
- **submit:** 表现成一个提交按钮。
- **text:** 表现成一个单行文本输入框。
- **textarea:** 表现成一个多行文本框。

这些标签的使用类似于 html 的标签, 因此比较简单。下面是对 login.jsp 页面的改写, 将原有的 html 标签改写成 Struts 的 html 标签。

当然, 此处还存在没有使用 Struts 的验证框架。因此, 还必须保留 JavaScript 验证。

```
<%@ page language="java" contentType="text/html; charset=gb2312" errorPage="" %>
<!-- 导入 Struts 的三个标签库-->
<%@include file="taglibs.jsp"%><html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<!-- 使用国际化资源文件的 key 输出标题-->
<title><bean:message key="login"/></title>
</head>
<script>
function check(form)
{
    if (form.username.value==null || form.username.value==" ")
    {
        <!-- 使用国际化资源文件的 key 输出提示-->
        alert('<bean:message key="noname"/>');
        return false;
    }
    else if(form.pass.value==null || form.pass.value==" ")
    {
        <!-- 使用国际化资源文件的 key 输出提示-->
        alert('<bean:message key="nopass"/>');
        return false;
    }
    //两者都已经输入
    else
    {
        return true;
    }
}
</script>
<body>
<font color="red">
```

```
<%  
//用于输出出错提示，出错提示保存在 request 的 err 属性里。  
if (request.getAttribute("err") != null)  
{  
    out.println(request.getAttribute("err"));  
}  
%>  
</font>  
<!-- 下面是登录表单 -->  
请输入用户名和密码：  
<html:form action="login.do" onsubmit="return check(this);">  
    <bean:message key="username"/><html:text property="username"/><br>  
    <bean:message key="pass"/><html:password property="pass"/><br>  
    <html:submit><bean:message key="login"/></html:submit><br>  
</html:form>  
</body>  
</html:html>
```

该页面的显示效果与之前的 JSP 显示效果并没有太多不同。可见，html 标签主要用于生成基本的 html 标签。

### 3.8.3 使用 bean 标签库

bean 标签库主要用于输出属性值、提示消息及定义请求参数等。下面是 bean 标签库的常用标签。

- cookie：将请求的 cookie 的值定义成脚本可以访问的 JavaBean 实例。
- define：将某个 bean 的属性值定义成脚本可以访问的变量。
- header：将请求头的值定义成脚本可以访问的变量。
- include：将某个 JSP 资源完整定义成一个 bean 实例。
- message：用于输出国际化信息。
- page：将 page Context 中的特定项定义成一个 bean。
- parameter：将请求参数定义成脚本可以访问的变量。
- resource：加载 Web 应用的资源，并将其变成 JavaBean。
- struts：用于将某个 Struts 的内部配置成一个 bean。
- write：用于输出某个 bean 的属性值。

下面将详细讲解这些标签。

#### 1. cookie 标签

cookie 标签用于将 cookie 值定义成 JavaBean，看下面页面的代码：

```
<%@ page contentType="text/html; charset=gb2312"%>  
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>  
<html>  
    <head>  
        <title>bean:cookie Tag 测试</title>  
    </head>  
    <body>  
        <div align="center">  
            <h1>bean:cookie Tag 测试</h1>
```

```
</div>
<p>作用：获取当前的 Cookie 信息，将其转化成一个 bean 实例<br>
显示我们当前的 session 关联的 cookie 属性 :</p>
<!-- 将 JSESSIONID 的 cookie 定义成 id 为 sess 的 bean-->
<bean:cookie id="sess" name="JSESSIONID" value="JSESSIONID-IS-UNDEFINED" />
<table border="1">
  <tr>
    <th>Property Name</th>
    <th>Correct Value</th>
    <th>Test Result</th>
  </tr>
  <tr>
    <td>comment</td>
    <!-- 下面采用两种方式输出 sess bean 的 comment 属性-->
    <td>
      <jsp:getProperty name="sess" property="comment" />
    </td>
    <td>
      <bean:write name="sess" property="comment" />&nbsp;
    </td>
  </tr>
  <tr>
    <td>domain</td>
    <!-- 下面采用两种方式输出 sess bean 的 domain 属性-->
    <td>
      <jsp:getProperty name="sess" property="domain" />
    </td>
    <td>
      <bean:write name="sess" property="domain"/>&nbsp;
    </td>
  </tr>
  <tr>
    <td>maxAge</td>
    <td>
      <!-- 下面采用两种方式输出 sess bean 的 maxAge 属性-->
      <jsp:getProperty name="sess" property="maxAge" />
    </td>
    <td>
      <bean:write name="sess" property="maxAge" />&nbsp;
    </td>
  </tr>
  <tr>
    <td>path</td>
    <!-- 下面采用两种方式输出 sess bean 的 path 属性-->
    <td>
      <jsp:getProperty name="sess" property="path" />
    </td>
    <td>
      <bean:write name="sess" property="path" />&nbsp;
    </td>
  </tr>
  <tr>
    <td>secure</td>
    <!-- 下面采用两种方式输出 sess bean 的 secure 属性-->
    <td>
      <jsp:getProperty name="sess" property="secure"/>
    </td>
    <td>
      <bean:write name="sess" property="secure" />&nbsp;
    </td>
  </tr>
```

```

</td>
</tr>
<tr>
    <td>value</td>
    <!-- 下面采用两种方式输出 sess bean 的 value 属性-->
    <td>
        <jsp:getProperty name="sess" property="value" />
    </td>
    <td>
        <bean:write name="sess" property="value"/>&nbsp;
    </td>
</tr>
<tr>
    <td>version</td>
    <!-- 下面采用两种方式输出 sess bean 的 version 属性-->
    <td>
        <jsp:getProperty name="sess" property="version" />
    </td>
    <td>
        <bean:write name="sess" property="version"/>&nbsp;
    </td>
</tr>
</table>
</body>
</html>

```

在上面的源文件中，将 JSESSIONID 的 cookie 定义成一个名为 sess 的 bean，然后依次输出该 bean 的属性，页面的运行效果如图 3.12 所示。



图 3.12 cookie 标签的运行效果

## 2. define 标签

define 标签主要用于定义 Java 脚本可以访问的变量。定义该变量时，变量的赋值非常灵活，可以直接为其指定特定值；也可以使用某个 bean 的属性值定义变量；或者将某个范围的 bean 转向另一个范围。例如，将 pageContext 下的 bean 转换到 Session 范围内。

看下面的页面代码：

```

<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
bean:define 标签用来定义一个页面变量<p>
1. 下面是定义的时候直接赋值的方式<br>
如同采用 String test1 = "测试属性"; 一样<br>
<bean:define id="test1" value="测试属性"/>
<%=test1%>
<br>
<hr>
2. 下面是把一个变量的值赋给另一个变量的值<br>
如同 String test2 = test1; <br>
<bean:define id="test2" name="test1"/>
<%=test2%><br>
<br>
<jsp:useBean id="bean1" scope="page" class="lee.LoginForm"/>
<jsp:setProperty name="bean1" property="password" value="tiger"/>
3. 下面将某个 bean 的属性定义成一个页面变量<br>
如同 String test3 = bean1.getPassword();
<bean:define id="test3" name="bean1" property="password"/>
<%=test3%><br>
<br>
<hr>
4. 下面将来自某个范围的 bean 转到另一个范围<br>
如同 session.setAttribute("bean2", bean1);
<bean:define id="bean2" name="bean1" scope="page" toScope="session"/>
<br>
<br>
5. bean:write 的作用: <br>
= ((LoginForm)session.getAttribute("bean2")).getPassword()
<br>
<bean:write name="bean2" property="password" scope="session"/>

```

在上面的页面文件中，出现了以下 4 种情况。

(1) 在使用 **bean** 标签定义 Java 脚本可以访问的变量时，可直接为变量赋值。用法如下：

<bean:define id="variableName" value="variableValue"/> 将 variableValue 的值赋给 variableName 的变量。

(2) 将一个变量的值直接赋给另一个变量，用法如下：

<bean:define id="variable1" name="variable2"/>：将 variable2 的值赋给 variable1。

(3) 将某个 **bean** 的属性定义成页面可以访问的变量，用法如下：

<bean:define id="variableName" name="beanName" property="beanProperty"/>：将 beanName 的 beanProperty 属性的值，赋给 variableName 的变量。

(4) 将一个范围的 **bean** 转换到另一个范围内，用法如下：

<bean:define id="bean1" name="bean2" scope="scope1" toScope="scope2"/>：将 scope1 中的 bean1 转到 scope2 的 bean2。

在最后部分演示了 **bean:write** 的作用——输出 **bean** 的属性值。图 3.13 显示了该页面的运行效果。



图 3.13 bean:define 标签的运行效果

### 3. header 标签

header 标签用于将特定的请求头信息包装成脚本可以访问的变量。header 的用法如下：

<bean:header id="variableName" name="headerName" />, 将 headerName 的请求头定义成 variableName 的变量。

看下面的 JSP 页面源代码：

```

<%@ page contentType="text/html; charset=gb2312"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
  <head>
    <title>bean:header 标签测试</title>
  </head>
  <body>
    <div align="center">
      <h1>bean:header 标签测试</h1>
    </div>显示本次请求的 http 文件头
    <br />
    <br />
    <%
      //获取所有的请求头
      java.util.Enumeration names =
        ((HttpServletRequest) request).getHeaderNames();
    %>
    <table border="1">
      <tr>
        <th>Header Name</th>
        <th>Header Value</th>
      </tr><%
      //遍历请求头, 获取请求头的名字, 根据请求头名依次输出每个头对应的值
      while (names.hasMoreElements()) {
        String name = (String) names.nextElement();
      %>
      <bean:header id="head" name="<% name %>" />
      <tr>
        <td>
    
```

```

<%= name %>
</td>
<td>
<%= head %>
</td>
</tr><%
    }
%>
</table>
</body>
</html>

```

#### 4. parameter 标签

parameter 标签用于将请求参数封装成一个脚本可以访问的变量，用法如下：

`<bean:parameter id="variableName" name="parameter"/>`: 将请求 parameter 的请求参数定义成 variableName 的变量。

看如下代码：

```

<%@ page contentType="text/html;charset=gb2312"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
    <head>
        <title>bean:parameter Tag 测试</title>
    </head>
    <body>
        <div align="center">
            <h1>bean:parameter Tag 测试</h1>
        </div>
        <p>将请求参数封装成 bean</p>
        <!-- 将请求参数 b 的值定义成 a 变量-->
        <bean:parameter id="a" name="b"/>
        <!-- 输出 a 变量的值-->
        <bean:write name="a"/>
    </body>
</html>

```

页面的运行效果如图 3.14 所示。

**注意：**在程序的运行效果图的地址栏中包含了 b 的请求参数，使用 bean:parameter 定义变量时，请求参数中必须包含 b 的请求参数。

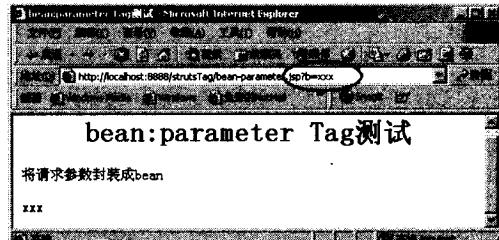


图 3.14 bean:parameter 的运行效果

#### 5. include 标签

include 标签用于将一个完整的 JSP 页面定义成 bean。用法如下：

`<bean:include id="beanName" page="uri" />`: 将 uri 对应的 JSP 资源定义成 beanName 的 bean。

看下面的 JSP 页面：

```

<%@ page contentType="text/html;charset=gb2312" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
    <head>

```

```

<title>测试 bean:include 标签</title>
</head>
bean:include 主要用来引入另外一个页面
<body>
    <div align="center">
        <h1>测试 bean:include 标签</h1>
    </div>
    <!-- 将 bean-header.jsp 页面的定义成 head 的 bean-->
    <bean:include id="head" page="/bean-header.jsp" />
    <hr/>
    <pre>
下面的内容是通过=header 输出<br>
<%= head %>
<br>
<br>
下面的内容是 filter="false"的输出<br>
<bean:write name="head" filter="false"/>
<br>
<br>
下面的内容是 filter="true"的输出<br>
<pre>
<bean:write name="head" filter="true"/>
</pre>
<br/>
    </body>
</html>

```

**注意：**当`<bean:write/>`标签后的 filter 为 true 时，将输出被包含页面的源代码。

JSP 页面使用 bean:include 将 bean-header.jsp 资源定义成标准 bean，然后采用了三种方式输出该 bean。程序的运行效果如图 3.15 所示。

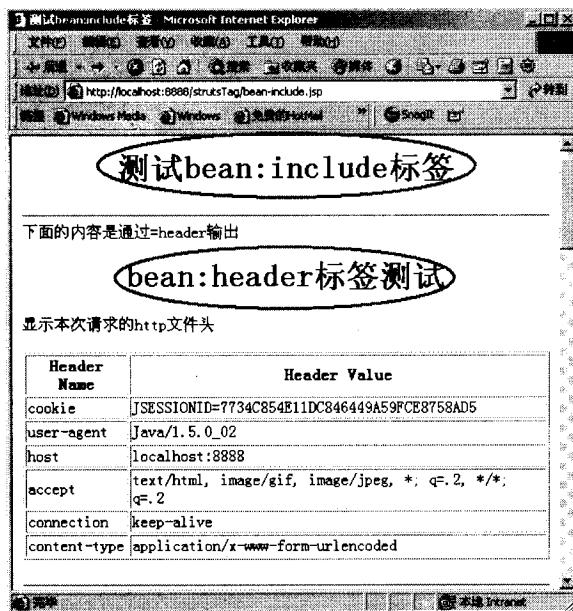


图 3.15 bean:include 的运行效果

图 3.16 中第一个红色标记的内容来自 bean-include.jsp 页面；而第二个红色标记的内容则来自 bean-header.jsp 页面。

### 3.8.4 使用 logic 标签库

logic 标签库是使用最频繁，相对复杂的标签库。logic 标签库主要用于完成基本的流程控制，比如循环及选择等。

logic 标签库主要有如下标签。

- empty：如果给定的变量为空或者为空字符串，则就计算并输出标签体的内容。
- equal：如果给定变量与特定的值相等，则会计算并输出该标签体的内容。
- forward：将某个页面的控制权 forward 确定的 ActionForward 项。
- greaterEqual：如果给定变量大于或等于特定的值，则会计算并输出标签体的内容。
- greaterThan：如果给定变量大于特定的值，则会计算并输出标签体的内容。
- iterate：通过遍历给定集合的元素，对标签体的内容进行循环。
- lessEqual：如果给定变量小于或等于特定的值，则会计算并输出标签体的内容。
- lessThan：如果给定变量小于特定的值，则会计算并输出标签体的内容。
- match：如果特定字符串是给定消息合适的子字符串，则会计算并输出标签体的内容。
- messagesNotPresent：如果请求中不包含特定的消息内容，将计算并输出标签体的内容。
- messagesPresent：如果请求中包含特定的消息内容，则计算并输出标签体的内容。
- notEmpty：如果给定的变量既不为空，也不是空字符串，则计算并输出标签体的内容。
- notEqual：如果给定变量不等于特定的值，则会计算并输出标签体的内容。
- notMatch：如果特定字符串不是给定消息合适的子字符串，则会计算并输出标签体的内容。
- notPresent：如果特定的值没有出现在请求中，则计算并输出标签体的内容。
- present：如果特定的值出现在请求中，则计算并输出标签体的内容。
- redirect：重定向页面。

下面依次讲解这些标签。

#### 1. empty 和 notEmpty 标签

这两个标签都支持标签体。它们用于判断给定变量是否为空，并判断是否计算和输出标签体，具体用法如下：

<logic:empty name="bean" scope="scope"> 标签体 </logic:empty>：如果指定范围 scope 里 bean 为空，则计算并输出标签体的内容。此处的 scope 有 page, request, session 和 application 等四个范围。如果没有指定范围，将在这四个范围中搜索。

<logic:empty name="bean" property="propertyName" scope="scope"> 标签体 </logic:empty>：如果 bean 的 propertyName 的属性为空，则计算并输出标签体的内容。

<logic:notEmpty name="bean" scope="scope"> 标签体 </logic:notEmpty>：如果 bean 不为空，则计算并输出标签体的内容。

<logic:notEmpty name="bean" property="propertyName" scope="scope"> 标签体  
</logic:empty>：如果 bean 的 propertyName 的属性不为空，则计算并输出标签体的内容。

看下面的 JSP 页面代码：

```
<%@ page contentType="text/html; charset=gb2312"%>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
    <head>
        <title>logic:empty 标签测试</title>
    </head>
    <body>
        <div align="center">
            <h1>logic:empty 标签测试</h1>
        </div>
        <!-- 实例化一个 id 为 bean 的 Person 对象 -->
        <jsp:useBean id="bean" scope="page" class="lee.Person" />
        <jsp:setProperty name="bean" property="age" value="5" />
        <br>
        logic:empty 用来判断该 bean 对象，及其属性是否为空<br>
        <hr>
        下面判断 bean 是否为空
        <logic:notEmpty name="bean" >
            不为空
        </logic:notEmpty>
        <logic:empty name="bean" >
            <br>
            空
        </logic:empty>
        <br>
        <hr>
        下面判断属性是否为空
        <logic:empty name="bean" property="age" >
            <br>
            空
        </logic:empty>
        <logic:empty name="bean" property="name" >
            <br>
            空
        </logic:empty>
    </body>
</html>
```

程序的运行效果如图 3.16 所示。

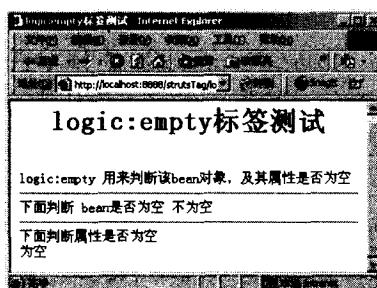


图 3.16 logic:empty 标签的运行效果

## 2. match/notMatch 标签

这两个标签都支持标签体。它们可用于判断给定变量是否包含给定的字符串；并判断是否计算和输出标签体。具体用法如下：

```
<logic:match name="name" property="propertyName" value="subString" scope="scope">
    标签体
</logic:match>
```

判断 scope 范围内名为 name 的 bean 的 propertyName 的属性，是否包含 subString，如果包含该字符串，则计算并输出标签体。

```
<logic:match parameter="paramterName" value="subString" scope="scope">
    标签体
</logic:match>
```

判断请求参数 paramterName 的值中是否包含 subString，如果包含，则计算并输出标签体。

```
<logic:notMatch name="name" property="propertyName" value="subString" scope="scope">
    标签体
</logic:notMatch>
```

判断 scope 范围内名为 name 的 bean 的 propertyName 的属性，是否包含 subString，如果不包含该字符串，则计算并输出标签体。

```
<logic:notMatch parameter="paramterName" value="subString" scope="scope">
    标签体
</logic:notMatch>
```

判断请求参数 paramterName 的值中是否包含 subString，如果不包含，则计算并输出标签体。

看下面的 JSP 页面：

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html>
    <head>
        <title>logic:match 标签测试</title>
    </head>
    <body>
        <div align="center">
            <h1>logic:match 标签测试</h1>
        </div>
        <!-- 定义了一个 Person 的实例，并将其 name 属性设为 "dfdtomsdfs" -->
        <jsp:useBean id="bean" scope="page" class="lee.Person" />
        <jsp:setProperty name="bean" property="name" value="dfdtomsdfs" />
        <!-- 判断 bean 的 name 属性是否包含 tom 子串-->
        <logic:match name="bean" property="name" value="tom">
            bean 的 name 中包含子字符串 "tom"
        </logic:match><br><hr>
        <!-- 判断 ddd 的请求参数是否包含 tom 子串-->
        <logic:notMatch parameter="ddd" value="tom">
            请求参数 ddd 中包含子字符串 "tom"
        </logic:notMatch>
    </body>
</html>
```

```

</logic:notMatch>
</body>
</html>

```

程序的执行效果如图 3.17 所示。

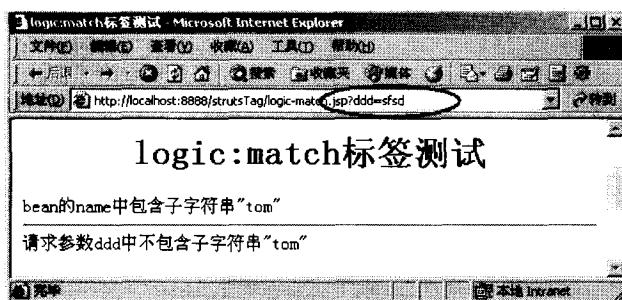


图 3.17 logic:match 的运行效果

注意：红色标记部分的请求参数 ddd 的值为 sfsd，没有包含 tom 子串，因此有图 3.17 的运行效果。

### 3. present 和 notPresent 标签

present 和 notPresent 标签用于判断某个 bean，或判断请求参数是否存在。用法如下：

`<logic:present name="beanName" scope="scope">` 标签体 `</logic:present>`：判断 scope 中 beanName 的 bean 是否存在，如果存在，则计算并输出标签体。此处的 scope 指的是 page, request, session, application 等范围，如果没有指定 scope，则依次搜索这些范围。

`<logic:present name="bean" property="propertyName" scope="scope">` 标签体 `</logic:present>`：判断 bean 的 propertyName 属性是否存在。如果存在，则计算并输出标签体。scope 属性作用与前面的 scope 属性完全相同。

`<logic:present cookie="cookieName">` 标签体 `</logic:present>`：判断名为 cookieName 的 cookie 是否存在，如果存在，则计算并输出标签体。

`<logic:present header="headerName">` 标签体 `</logic:present>`：判断名为 headerName 的请求头是否存在，如果存在，则计算并输出标签体。

`<logic:present parameter="name">` 标签体 `</logic:present>`：判断请求是否包含名为 name 的请求参数，如果包含该参数，则计算并输出标签体。

`<logic:notPresent name="beanName" scope="scope">` 标签体 `</logic:notPresent>`：判断 scope 中 beanName 的 bean 是否存在，如果不存在，则计算并输出标签体。此处的 scope 指的是 page, request, session, application 等范围，如果没有指定 scope，则依次搜索这些范围。

`<logic:notPresent name="bean" property="propertyName" scope="scope">` 标签体 `</logic:notPresent>`：判断 bean 的 propertyName 属性是否存在。如果不存在，则计算并输出标签体。scope 属性的作用与前面的相似。

`<logic:notPresent cookie="cookieName">` 标签体 `</logic:notPresent>`：判断名为

cookieName 的 cookie 是否存在，如果不存在，则计算并输出标签体。

<logic:notPresent header="headerName"> 标签体 </logic:notPresent>：判断名为 headerName 的请求头是否存在，如果不存在，则计算并输出标签体。

<logic:notPresent parameter="name"> 标签体 </logic:notPresent>：判断请求是否包含名为 name 的请求参数，如果没有包含该参数，则计算并输出标签体。

注意：如果某个 bean 为 null，或者 bean 的属性为 null 时，logic:notPresent 标签会输出标签体；当 bean 为 null，或属性值为 null 时，present 标签当其不存在。

看下面的 JSP 页面代码：

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html>
    <head>
        <title>logic:present 标签测试</title>
    </head>
    <body>
        <div align="center">
            <h1>logic:present 标签测试</h1>
        </div>
        <jsp:useBean id="bean" scope="page" class="lee.Person" />
        <table border="1">
            <tr>
                <td>session 范围的 bean 实例</td>
                <td>存在与否</td>
                <td>
                    <logic:present name="bean" scope="session">
                        存在
                    </logic:present>
                    <logic:notPresent name="bean" scope="session">
                        不存在
                    </logic:notPresent>
                </td>
            </tr>
            <tr>
                <td>名为 FOOBAR 的 bean</td>
                <td>存在与否</td>
                <td>
                    <logic:present name="FOOBAR">
                        存在
                    </logic:present>
                    <logic:notPresent name="FOOBAR">
                        不存在
                    </logic:notPresent>
                </td>
            </tr>
            <tr>
                <td>bean 实例</td>
                <td>存在与否</td>
                <td>
                    <logic:present name="bean">
                        存在
                    </logic:present>
                    <logic:notPresent name="bean">
```

```
    不存在
    </logic:notPresent>
</td>
</tr>
<tr>
    <td>名为 JSESSIONID 的 Cookie</td>
    <td>存在与否</td>
    <td>
        <logic:present cookie="JSESSIONID">
            存在
        </logic:present>
        <logic:notPresent cookie="JSESSIONID">
            不存在
        </logic:notPresent>
    </td>
</tr>
<tr>
    <td>名为 FOOBAR 的 Cookie</td>
    <td>存在与否</td>
    <td>
        <logic:present cookie="FOOBAR">存在</logic:present>
        <logic:notPresent cookie="FOOBAR">不存在</logic:notPresent>
    </td>
</tr>
<tr>
    <td>User-Agent 的请求头</td>
    <td>存在与否</td>
    <td>
        <logic:present header="User-Agent">存在</logic:present>
        <logic:notPresent header="User-Agent">不存在</logic:notPresent>
    </td>
</tr>
<tr>
    <td>FOOBAR 的请求头</td>
    <td>存在与否</td>
    <td>
        <logic:present header="FOOBAR">存在</logic:present>
        <logic:notPresent header="FOOBAR">不存在</logic:notPresent>
    </td>
</tr>
<tr>
    <td>param1 的请求参数</td>
    <td>存在与否</td>
    <td>
        <logic:present parameter="param1">存在</logic:present>
        <logic:notPresent parameter="param1">不存在</logic:notPresent>
    </td>
</tr>
<tr>
    <td>FOOBAR 的请求参数</td>
    <td>存在与否</td>
    <td>
        <logic:present parameter="FOOBAR">存在</logic:present>
        <logic:notPresent parameter="FOOBAR">不存在</logic:notPresent>
    </td>
</tr>
<tr>
    <td>bean 的 age 属性</td>
```

```

<td>是否存在</td>
<td>
    <logic:present name="bean" property="age">存在</logic:present>
    <logic:notPresent name="bean" property="age">不存在</logic:notPresent>
</td>
</tr>
<tr>
    <td>bean 的 name 属性</td>
    <td>存在与否</td>
    <td>
        <logic:present name="bean" property="name">存在</logic:present>
        <logic:notPresent name="bean" property="name">不存在</logic:notPresent>
    </td>
</tr>
</table>
</body>
</html>

```

页面依次测试了上面的 5 种情况，页面的运行效果如图 3.18 所示。



图 3.18 logic:present 标签的运行效果

图中标记的部分地址栏里包含了 param1 的请求参数，logic:present 标签也输出了标签体的内容。

logic:messagesPresent 和 logic:messagesNotPresent 标签与 logic:present 的用法非常相似，此处不再赘述。

#### 4. forward 和 redirect 标签

这两个标签都用于转向，但转向的机制不一样。

logic:redirect 标签用于重定向，原有 HTTP 请求中包含的属性及参数全部丢失，用法如下：

<logic:redirect href="page.jsp" />：直接重定向到 page.jsp 资源。

<logic:redirect page="page.jsp" />：利用相对地址来控制转发。在本模块的 URL 中增加 page 属性对应的 URI 生成的重定向地址。

`<logic:redirect forward="forwardName" />`: 利用所有全局 Forward 对象完成重定向, 其中 `forwardName` 必须在全局 Forward 中定义。

`logic:forward` 标签用于转发, 但不会丢失原有的 HTTP 请求中的属性及参数, 用法如下。

`<logic:forward name="page.jsp" />`: 利用所有全局 Forward 对象完成转发, 其中 `forwardName` 必须在全局 Forward 中定义。

下面是分别利用两个标签转发的 JSP 页面:

利用 `logic:redirect` 控制重定向的页面

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<logic:redirect href="other.jsp" />
```

利用 `logic:forward` 控制转发的页面代码

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<logic:forward name="test1" />
```

两次转发的实际资源都在 `other.jsp`, 该页面的源代码如下:

```
<%@ page contentType="text/html; charset=gb2312" errorPage="error.jsp" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html>
<head>
<title>被转向的页面</title>
</head>
<body>
被转向的页面<hr>
请求参数 param1<logic:present parameter="param1">存在</logic:present>
<logic:notPresent parameter="param1">不存在</logic:notPresent>
</body>
</html>
```

该页面使用了 `logic:present` 标签来判断 `param1` 的请求参数是否存在。程序分别对 `logic-redirect.jsp` 和 `logic-forward.jsp` 页面请求, 请求时都在地址栏中增加 `param1` 的参数。

两个页面的执行效果如图 3.19 和图 3.20 所示。

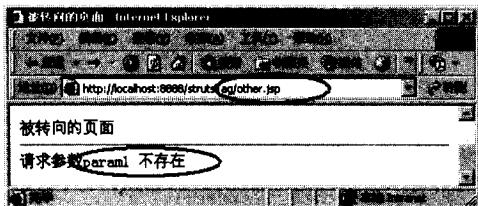


图 3.19 logic:redirect 标签的运行效果

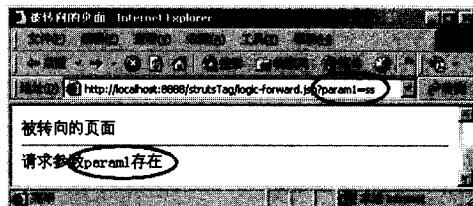


图 3.20 logic:forward 标签的运行效果

注意: 图中标记的部分为地址栏的改变及 `param1` 的参数是否存在。

## 5. iterate 标签

前面的标签都用于条件控制，而 iterate 标签主要用于循环控制。它用于遍历集合里的每个元素，包括访问集合里对象的属性值。用法如下：

```
<logic:iterate id="item" collection="collectionName" indexId="index" offset="1" length="2">
    标签体
</logic:iterate>
```

遍历名为 collectionName 的集合里每个元素被命名为 item，可用于标签体访问，其中 offset 是遍历的起始点，length 用于控制遍历的个数。

```
<logic:iterate id="item" name="bean1" scope="scope" indexId="index" offset="1" length="2">
    标签体
</logic:iterate>
```

遍历名为 bean1 的集合将从 scope 的范围里搜索，如果没有指定范围，将从 page, request, session, application 中依次搜索。offset 和 length 属性的效果与前面的用法相似。

```
<logic:iterate id="user" name="stuff" type="lee.Person" indexId="index" offset="1" length="2">
    标签体
</logic:iterate>
```

遍历名为 bean1 的集合将从 scope 的范围里搜索。如果没有指定范围，将从 page, request, session, application 中依次搜索。offset 和 length 属性的效果与前面的用法相似。此处还指定了集合 bean1 里每个元素的类型，从而可以直接访问每个元素的属性。

看下面的 JSP 页面代码：

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ page import = "lee.Person" %>
<%@ page import = "java.util.ArrayList" %>
<%
    //初始化集合
    ArrayList list1 = new ArrayList();
    list1.add("张三");
    list1.add("李四");
    list1.add("王五");
    list1.add("蔡六");
%>
1.输出的集合的元素，也就是对集合元素做遍历，遍历时指定了集合的起始点，也指定了遍历长度<br>
<logic:iterate id="item" collection="<%list1%>" indexId="index" offset="1" length="2">
    <%=index%>&nbsp;<%=item%><br>
</logic:iterate>
<hr>
2.将集合放入某个生存周期内，然后输出集合元素
<%pageContext.setAttribute("bean1", list1, PageContext.PAGE_SCOPE);%>
<hr>
<logic:iterate id="item" name="bean1" scope="page" indexId="index">
    <%=index%>&nbsp;<%=item%><br>
```

```
</logic:iterate>
<hr>
3. 使用 bean:write 标签来输出集合元素<br>
<logic:iterate id="item" name="bean1" scope="page" indexId="index">
    <bean:write name="index"/>&nbsp;&nbsp;
    <bean:write name="item"/><br>
</logic:iterate>
<hr>
<%>
    //再次初始化集合，集合里每个元素都是一个 Person 实例
    ArrayList users = new ArrayList();
    users.add(new Person("tomcat", 12));
    users.add(new Person("mysql", 24));
    users.add(new Person("oracle", 36));
    pageContext.setAttribute("stuff", users);
<%>
4. 输入集合里元素的属性<br>
<table border= 1 bgcolor="#99bb99">
<logic:iterate id="user" name="stuff" type="lee.Person" indexId="index" >
    <tr>
        <td><%=index%>.</td>
        <td><bean:write name="user" property="name" /></td>
        <td><bean:write name="user" property="age" /></td>
    </tr>
</logic:iterate>
</table>
```

页面的执行效果如图 3.21 所示。

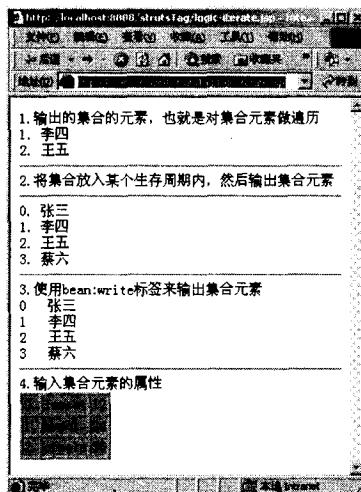


图 3.21 logic:iterator 的运行效果

## 3.9 Struts 的数据校验

数据校验也称为输入校验，指导对用户的输入进行基本过滤，包括必填字段，字段必须为数字及两次输入的密码必须相匹配等。这些是每个 MVC 框架都应该完成的任务，

Struts 提供了基本的数据校验，如果结合 commons-validator.jar，Struts 则拥有强大的校验框架，包括进行客户端的 JavaScript 校验等。

Struts 的数据校验大致有如下几种方式：

- ActionForm 的代码校验。
- Action 里的代码校验。
- 结合 commons-validator.jar 的校验。

### 3.9.1 ActionForm 的代码校验

ActionForm 的代码校验是最基本的校验方式。这种校验方式是重写 ActionForm 的 validate 方法，在该方法内对所有的字段进行基本校验。如果出现不符合要求的输出，则将出错提示封装在 ActionError 对象里，最后将多个 ActionError 组合成 ActionErrors 对象，该对象里封装了全部的出错提示。

下面是重写了 validate 方法的 ActionForm 的代码：

```
public class LoginForm extends ActionForm implements Serializable
{
    private String username = null;
    private String password = null;
    private String vercode = null;
    // username 的 getter 方法
    public String getUsername() {
        return username;
    }
    //username 的 setter 方法
    public void setUsername(String username) {
        this.username = username;
    }
    //password 的 setter 方法
    public String getPassword() {
        return password;
    }
    // password 的 setter 方法
    public void setPassword(String password) {
        this.password = password;
    }
    // vercode 的 getter 方法
    public String getVercode() {
        return vercode;
    }
    // vercode 的 setter 方法
    public void setVercode(String vercode) {
        this.vercode = vercode;
    }
    //重写的 validate 方法，完成数据校验
    public ActionErrors validate(ActionMapping mapping, HttpServletRequest
request)
    {
        //ActionErrors 用于包装所有的出错提示
        ActionErrors errors = new ActionErrors();
        //如果用户名为空
```

```

        if (username == null || username.equals(""))
        {
            //error.username 对于资源文件的 key. 用户名是对应资源文件的第一个参数
            errors.add("username", new ActionError("error.username", "用户名"));
        }
        //如果密码为空
        if (password == null || password.equals(""))
        {
            errors.add("password", new ActionError("error.password", "密码"));
        }
        //如果验证码为空
        if (vercode == null || vercode.equals(""))
        {
            errors.add("vercode", new ActionError("error.vercode", "验证码"));
        }
        //返回包装了所有出错提示的 ActionErrors 对象
        return errors;
    }
}

```

**注意：**重写的是 validate(ActionMapping mapping, HttpServletRequest request)方法，第二个参数的类型是 HttpServletRequest，而不是 ServletRequest。

上面的代码非常简单，为每个属性提供了对应的 setter 和 getter 方法。最后重写了 validate 方法，在 valid 方法里对每个需要校验的域完成校验。如果没有通过校验，则将出错提示包装成 ActionError 对象，最后将多个 ActionError 对象组合成 ActionErrors 后返回。

**注意：**使用 ActionForm 的输入校验时，应为对应的 action 元素增加 input 属性，该属性指定当校验失败后的返回页面。

JSP 页面使用<html:errors/>标签输出出错提示，但出错提示并没有采用硬编码的方式直接定义，而是使用资源文件的 key，这样可实现出错提示的国际化。

下面是登录所用的 JSP 页面的代码：

```

<%@ page contentType="text/html;charset=gb2312" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html>
<head>
<title>注册</title>
</head>
<!-- 下面标签用于输出出错提示-->
<html:errors/>
<form name="form1" method="post" action="login.do">
<table border="0" width="100%">
<tr>
<th align="left">用户名: </th>
<td align="left"> <input type="text" name="username" size=15> </td>
</tr>
<tr>
<th align="left">密 &nbsp;&nbsp;码: </th>
<td align="left"> <input type="text" name="password" size="15"/> </td>
</tr>

```

```

<tr>
    <th align="left"> 验证码 </th>
    <td align="left"> <input type="text" name="vercode" size="15" /> </td>
</tr>
<tr>
    <td>
        <input type="submit" value='<bean:message key="button.submit"/>' />
        &nbsp;
        <input type="reset" value='<bean:message key="button.reset"/>' />
    </td>
</tr>
</table>
</form>
</body>
</html>

```

该 JSP 页面的代码非常简单，除了增加<html:errors/>标签用于输出出错提示外，与前面的页面并没有特别之处，但该页面可以完成数据校验，并可输出出错提示。

本示例中的 Action 更加简单，直接返回一个 Forward，没有调用任何的逻辑处理，此处不再赘述。

在 login.jsp 登录页面中仅输入用户名，然后单击【提交】按钮，将出现如图 3.22 所示的运行效果。

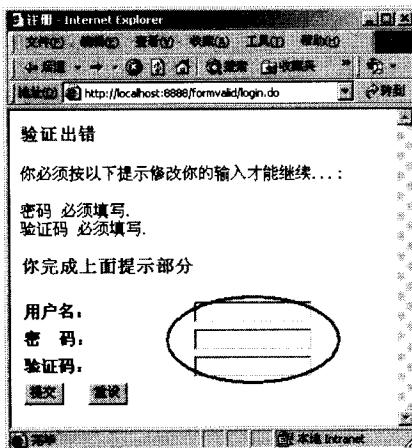


图 3.22 使用 ActionForm 输入校验的运行效果

因为有资源文件的支持，JSP 页面只需要简单的<html:errors/>标签就可生成上面全部的出错提示，看下面的资源文件：

```

#html:errors 将自动加载 errors.header 的作为出错提示的标题
errors.header=<h3><font color="red">验证出错</font></h3>你必须按以下提示修改你的
输入才能继续...:<p>
error.username={0} 必须填写.<br>
error.password={0} 必须填写.<br>
error.vercode={0} 必须填写.<br>
#html:errors 将自动加载 errors. footer 的作为出错提示的最后一行
errors.footer=<h3><font color="green">你完成上面提示部分</font></h3>

```

上面的验证还有一个小小的问题：如果输入没有通过验证，则所有输入的信息将全部丢失，正如图 3.23 显示的，即使用户名文本框中输入了用户名，一旦提交后，该文本框中的数据将完全丢失。为了解决这个问题，建议将 HTML 的<input>标签换成 Struts 的<html:text>标签。

修改后页面的表单片段如下：

```
<html:form action="login.do">
<table border="0" width="100%">
<tr>
    <th align="left">用户名: </th>
    <td align="left"><html:text property="username" size="15"/></td>
</tr>
<tr>
    <th align="left">密 &nbsp;&nbsp;码: </th>
    <td align="left"><html:text property="password" size="15"/></td>
</tr>
<tr>
    <th align="left">验证码: </th>
    <td align="left"><html:text property="vercode" size="15"/></td>
</tr>
<tr>
    <td>
        <input type="submit" value='<bean:message key="button.submit"/>' />
        &nbsp;
        <input type="reset" value='<bean:message key="button.reset"/>' />
    </td>
</tr>
</table>
</html:form>
```

修改后的运行效果如图 3.23 所示。

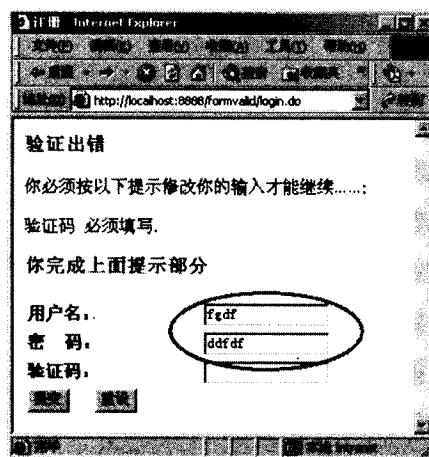


图 3.23 使用 Struts 的 html 标签的运行效果

使用 ActionForm 的数据校验有个显著的问题：ActionErrors 和 ActionError 都是 Struts 不再推荐使用的类。因此，应尽量避免使用这种校验方法。

### 3.9.2 Action 的代码校验

在 Action 里通过代码完成输入校验，是最基本，也最容易使用的方法。与最初的 MVC 设计相似，在调用业务逻辑组件之前，先对数据进行基本校验。这是最传统也是最原始的方法。

在 Action 里完成校验，实际上就是在 execute 方法的前面增加数据校验的部分代码。

下面是增加了数据校验 Action 的代码：

```
public final class LoginAction extends Action
{
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //解析请求参数
        LoginForm loginForm = (LoginForm)form;
        String username = loginForm.getUsername();
        //对请求参数执行基本校验
        if (username == null || username.equals(""))
        {
            request.setAttribute("errors", "nameNotNull");
            //如果未通过校验，返回 input Forward
            return mapping.findForward("input");
        }
        //如果通过校验，则开始调用业务逻辑
        else
        {
            //此处并未调用任何业务逻辑，直接返回
            return mapping.findForward("success");
        }
    }
}
```

这种校验方式非常容易理解，所有的代码都需要程序员自己控制，相当灵活。但有如下几个不方便之处。

- 用户需要书写大量的校验代码，使程序变得烦琐。
- 数据校验应该在填充 ActionForm 里完成，最好能在客户端完成校验，而不是推迟到 Action 里才完成数据校验。

注意：在实际的使用中，这种校验方式不仅程序开发复杂，且性能也不高。

### 3.9.3 结合 commons-validator.jar 的校验

借助于 commons-validator.jar 的支持，Struts 的校验功能非常强大，几乎不需书写任何代码。不仅可以完成服务器端校验，同时还可完成客户端校验，即弹出 Javascript 提示。

使用 commons-validator.jar 校验框架时，有如下几个通用配置。

- 增加校验资源。

- 利用 ValidatorPlugIn 加载校验资源。
- ActionForm 使用 ValidatorForm 的子类。

下面分别通过三个示例讲解这三种校验：基本的校验、对动态 ActionForm 执行校验及弹出 JavaScript 校验提示。

### 1. 继承 ValidatorForm 的校验

如果需要使用 commons-validator 框架，请按如下步骤进行。

(1) Struts 的 ActionForm 必须是 ValidatorForm 的子类，下面是本实例中 ActionForm 的片段：

```
public class LoginForm extends ValidatorForm
{
    //分别对应页面的四个属性
    private String username;
    private String pass;
    private String rpass;
    private String mail;
    //下面提供所有属性的 setter 和 getter 方法
}
```

JSP 页面使用 Struts 的 html 标签提供了基本表单域，包括用户名、密码、重复密码及电子邮件等四个域。

如果两次密码输入不匹配，而且邮件的输入不符合邮件格式，则运行效果如图 3.24 所示。

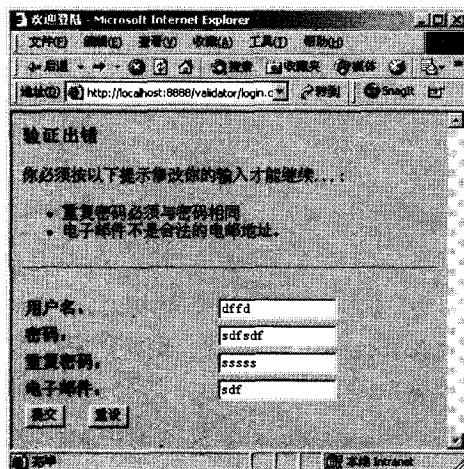


图 3.24 使用 common-validator 校验的效果

(2) 编写表单域时必须满足校验规则。校验规则都由规则文件控制，规则文件有以下两个。

- validator-rules.xml 文件
- validation.xml 文件

第一个文件可在 Struts 的解压缩后的文件夹的 lib 下找到，将该文件复制到 WEB-INF

路径下，该文件是个通用文件。

而 validation.xml 文件则是属于该项目的校验文件，负责定义每个表单域必须满足的规则，以及规则的详细说明。

下面是本示例程序的 validation.xml 文件：

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- 校验规则文件的文件头，包含 DTD 等信息-->
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons Validator Rules
    Configuration 1.1.3//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<!-- 校验规则文件的根元素-->
<form-validation>
<!-- 所有需要校验的 form 都放在 formset 元素里-->
<formset>
    <!-- 定义需要校验的表单，此处的 name 必须与 struts-config.xml 中定义
        的 AcitonForm 的名字一致-->
    <form name="loginForm">
        <!-- 每个 field 元素定义一个表单域，必须满足怎样的规则
            此处定义必须满足两个规则：必填，匹配正则表达式-->
        <field property="username" depends="required,mask">
            <!-- 定义出错字符串的第一个参数的值-->
            <arg key="loginForm.username" position="0"/>
            <!-- 定义正则表达式-->
            <var>
                <var-name>mask</var-name>
                <var-value>^[a-zA-Z]+\$</var-value>
            </var>
        </field>
        <!-- 定义 pass 域，需要满足必填规则-->
        <field property="pass" depends="required">
            <!-- 定义出错字符串的第一个参数的值-->
            <arg key="loginForm.pass" position="0"/>
        </field>
        <!-- 定义 rpass 域，需要满足必填、满足有效条件这两个规则-->
        <field property="rpass" depends="required,validwhen">
            <!-- 定义出错字符串的第一个参数的值-->
            <arg key="loginForm.rpass" position="0"/>
            <!-- 定义出错字符串的第二个参数的值-->
            <arg key="loginForm.pass" position="1"/>
            <!-- 定义不满足有效条件时候额出错提示-->
            <msg name="validwhen" key="loginForm.valid"/>
            <!-- 定义必须满足的有效条件-->
            <var>
                <!-- test 是满足有效条件的固定匹配条件，无须修改-->
                <var-name>test</var-name>
                <!-- 设定有效条件，此项为空，或者此项与 pass 域相等-->
                <var-value>((this==null)or(this==pass))</var-value>
            </var>
        </field>
        <!-- 定义 mail 域，需要满足必填、合法 email 地址这两个规则-->
        <field property="mail" depends="required,email">
            <arg key="loginForm.email" position="0"/>
        </field>
    </form>
</formset>
</form-validation>
```

经过这些规则的定义，Struts 知道了每个表单域都应该满足怎样的规则。如果满足这些规则，将不会提交给 Action 处理。

(3) 指定如果不满足校验规则时，系统将返回到哪个页面，可通过 struts-config.xml 文件指定即可。在 struts-config.xml 文件中配置 action 时，action 有个 input 属性，该属性用于指定不满足规则时返回的页面。另外，还应为 action 元素增加 validate=“true” 属性。

下面是 struts-config.xml 文件的源代码：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- struts 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- struts 配置文件的根元素-->
<struts-config>
    <!-- 定义 ActionForm，注意此处的 name 必须与校验文件中需要校验的
        form-bean 的名字一致 -->
    <form-beans>
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 定义 Action 映射-->
    <action-mappings>
        <!-- 下面的 action 定义增加了 input 属性，该属性定义了不能通过
            输入校验时的返回页面-->
        <action path="/login" type="lee.LoginAction" name="loginForm"
            scope="request" validate="true" input="/login.jsp" >
            <!-- 定义了一个局部 Forward-->
            <forward name="success" path="/welcome.html"/>
        </action>
    </action-mappings>
    <!-- 加载国际化的资源文件-->
    <message-resources parameter="mess"/>
    <!-- 加载验证资源文件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.
            xml,/WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true" />
    </plug-in>
</struts-config>
```

(4) 加载校验规则文件。加载的校验规则文件有两个：validator-rules.xml，该文件由 Struts 提供；validation.xml 文件，该文件由程序员书写，定义了表单域必须满足的规则。

(5) 增加校验所需的国际化资源文件，本示例程序使用的国际化资源文件如下：

```
#下面定义了页面的国际化标题
button.submit=提交
button.reset=重设
title.regist=欢迎登录
username=用户名:
pass=密码:
mail=电子邮件:
rpass=重复密码:
#下面是标准错误提示
errors.header=<h3><font color="red">验证出错</font></h3>你必须按以下提示修改你的
```

```

输入才能继续...:<p>
    errors.required={0} 必须填写。
    errors.minLength={0} 的长度必须大于{1}个字符。
    errors.maxLength={0} 的长度必须小于{1}个字符。
    errors.invalid={0} 格式不符合，无效。
    html.li.open=<li>
    html.li.close=</li>
    errors.byte={0} must be an byte.
    errors.short={0} must be an short.
    errors.integer={0} must be an integer.
    errors.long={0} must be an long.
    errors.float={0} must be an float.
    errors.double={0} must be an double.
    errors.date={0} 不是有效的日期。
    errors.range={0} 必须在{1}到{2}之间。
    errors.creditcard={0} is not a valid credit card number.
    errors.email={0} 不是合法的电邮地址。
    errors.footer=<h3><font color="green">请您先完成上面提示部分</font></h3>
# loginForm 所需要使用的标题
loginForm.username=用户名
loginForm.pass=密码
loginForm.email=电子邮件
loginForm.rpass=重复密码
loginForm.valid={0} 必须与{1}相同

```

(6) 在 JSP 页面中输出出错提示。使用 `html:messages` 标签来完成在 JSP 页面中输出出错提示时，可在登录的 `login.jsp` 页面中增加如下代码：

```

<!-- 如果消息存在，计算并输出标签体-->
<logic:messagesPresent>
    <!-- 直接输出 errors.header 的国际化消息-->
    <bean:message key="errors.header"/>
    <ul>
        <!-- 依次遍历每个错误 -->
        <html:messages id="error">
            <!-- 输出每个出错提示-->
            <li><bean:write name="error"/></li>
        </html:messages>
    </ul><hr />
</logic:messagesPresent>

```

经过上面的 6 个步骤，就可将 Struts 与 common-validator 校验框架成功结合。以后需要增加校验某个表单域时，只需简单修改 `validation.xml` 文件即可，在其中增加需要校验的 form-bean，以及表单域应满足的规则等。

## 2. common-validator 支持的校验规则

common-validator 支持的校验规则非常丰富，特别是 `mask` 和 `validwhen` 两个规则，极大地丰富了该校验框架的功能。

常用的校验规则有如下几种。

- **required:** 必填。
- **validwhen:** 必须满足某个有效条件。
- **minlength:** 输入必须大于最小长度。
- **maxlength:** 输入必须小于最大长度。

- mask: 输入匹配正确的表达式。
- byte: 输入只能是一个 byte 类型变量。
- short: 输入只能是一个 short 类型变量。
- integer: 输入只能是一个 integer 变量。
- long: 输入只能是一个 long 变量。
- float: 输入只能是一个 float 变量。
- double: 输入只能是一个 double 变量。
- date: 输入必须是一个日期。
- intRange: 输入的数字必须在整数范围内。
- floatRange: 输入的数字必须在单精度浮点数范围内。
- doubleRange: 输入的数字必须在双精度浮点数范围内。
- email: 输入必须是有效的 E-mail 地址。
- url: 输入必须是有效的 url 地址。

由此可见，当输入校验无法通过时，系统将出现出错提示。上面的校验规则大多有默认的出错提示 key，因此在国际化资源文件中，下面这些标准的出错提示也必不可少：

```
#违反必填规则时候的出错信息  
errors.required={0} 必须填写。  
#违反最小长度规则时候的出错信息  
errors.minLength={0} 的长度不能少于 {1} 个字符。  
#违反最大长度规则时候的出错信息  
errors.maxLength={0} 的长度不能大于 {1} 个字符。  
#违反 validwhen 规则时候的出错信息  
errors.invalid={0} 无效。  
#违反 byte 规则时候的出错信息  
errors.byte={0} 必须是个 byte 类型的值。  
#违反 short 规则时候的出错信息  
errors.short={0} 必须是个 short 类型的值。  
#违反 integer 规则时候的出错信息  
errors.integer={0} 必须是个 integer 类型的值。  
#违反 long 规则时候的出错信息  
errors.long={0} 必须是个 long 类型的值。  
#违反 float 规则时候的出错信息  
errors.float={0} 必须是个 float 类型的值。  
#违反 double 规则时候的出错信息  
errors.double={0} 必须是个 double 类型的值。  
#违反 date 规则时候的出错信息  
errors.date={0} 必须是有效的日期。  
#违反所有的 range 规则时候的出错信息  
errors.range={0} 必须在 {1} 和 {2} 之间。  
#违反 email 规则时候的出错信息  
errors.email={0} 不是一个有效的 E-mail 地址。
```

这些出错提示都是默认的，如果输入校验无法通过，则将自动出现这些提示。

### 3. 使用 DynaValidatorForm 的校验

即使不书写 ActionForm，也可以利用 common-validator 校验框架。如同在 3.7 节中使用动态 Form 一样，不编写 ActionForm，也可直接在 struts-config.xml 文件中配置

ActionForm。

此时使用的 ActionForm 的实现类，必须既是动态 Form，也是验证 Form，DynaValidatorForm 就是满足这两个条件的 Form。

下面是使用 DynaValidatorForm 的 struts-config.xml 文件的源代码：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- struts 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- struts 配置文件的根元素-->
<struts-config>
    <!-- 定义 ActionForm，注意此处的 name 必须与校验文件中需要校验的
        form-bean 的名字一致 -->
    <form-beans>
        <!-- 定义动态 验证 ActionForm-->
        <form-bean name="loginForm" type="org.apache.struts.validator.
            DynaValidatorForm">
            <!-- 下面依次定义了四个 form 属性-->
            <form-property name="username" type="java.lang.String"/>
            <form-property name="pass" type="java.lang.String"/>
            <form-property name="rpass" type="java.lang.String"/>
            <form-property name="mail" type="java.lang.String"/>
        </form-bean>
    </form-beans>
    <!-- 定义 Action 映射-->
    <action-mappings>
        <!-- 下面的 action 定义增加了 input 属性，该属性定义了不能通过
            输入校验时的返回页面-->
        <action path="/login" type="lee.LoginAction" name="loginForm"
            scope="request" validate="true" input="/login.jsp" >
            <!-- 定义了一个局部 Forward-->
            <forward name="success" path="/welcome.html"/>
        </action>
    </action-mappings>
    <!-- 加载国际化的资源文件-->
    <message-resources parameter="mess"/>
    <!-- 加载验证资源文件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.
            xml,/WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true" />
    </plug-in>
</struts-config>

```

程序的其他地方与第一个示例没有丝毫区别，此处不再赘述。

#### 4. 弹出客户端 JavaScript 提示

如需要弹出客户端 JavaScript 校验非常简单，无须修改其他配置文件，只需修改登录使用的 JSP 页面的两个地方。

(1) 为 form 元素增加 onsubmit="return validateXxxForm(this);”属性，其中的 XxxForm 就是需要校验的 form 名，与 struts-config.xml 中配置的 form-bean 的 name 属性一致，也与 validation.xml 文件中需要校验的 form 的 name 属性一致。

(2) 增加<html:javascript formName="xxxForm"/>, 其中 xxxForm 是需要校验的 form 名。

下面是修改的 login.jsp 页面的代码:

```
<!-- 为 form 元素增加 onsubmit 属性-->
<html:form action="login.do" onsubmit="return validateLoginForm(this); ">
<table border="0" width="100%">
<tr>
    <!-- 输出用户的国际化提示-->
    <th align="left"><bean:message key="username"/></th>
    <td align="left"><html:text property="username" size="15"/></td>
</tr>
<tr>
    <!-- 输出密码的国际化提示-->
    <th align="left"><bean:message key="pass"/></th>
    <td align="left"><html:text property="pass" size="15"/></td>
</tr>
<tr>
    <!-- 输出重复密码的国际化提示-->
    <th align="left"><bean:message key="rpass"/></th>
    <td align="left"><html:text property="rpass" size="15"/></td>
</tr>
<tr>
    <!-- 输出电子邮件的国际化提示-->
    <th align="left"><bean:message key="mail"/></th>
    <td align="left"><html:text property="mail" size="15"/></td>
</tr>
<tr>
    <td> <input type="submit" value='<bean:message key="button.submit"/>' />
        &nbsp;
        <input type="reset" value='<bean:message key="button.reset"/>' />
    </td>
</tr>
</table>
</html:form>
<!-- 增加 html:javascript 标签-->
<html:javascript formName="loginForm"/>
```

如果输入不能通过校验，则运行效果如图 3.25 所示。

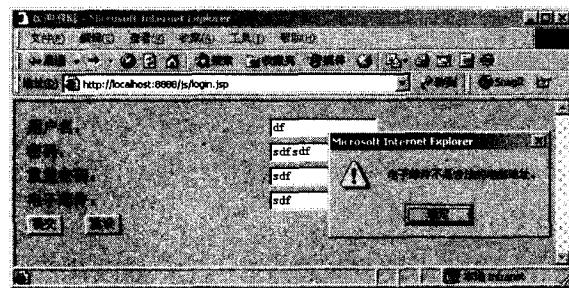


图 3.25 JavaScript 校验的效果

图 3.25 中密码和重复密码两个输入框中的输入不相同，但 JavaScript 校验提示并未弹出。这是因为：并不是所有的校验规则都可“转换”客户端的 JavaScript 校验语法。

**注意：**即使使用了客户端校验规则，也不要删除页面的 `html:messages` 标签。因为该标签会在客户端校验通过，而在服务器端校验并未通过时弹出提示。

在上面的示例中，当输入了合法的电子邮件，然后单击【提交】按钮时，将出现如图 3.26 所示的效果。

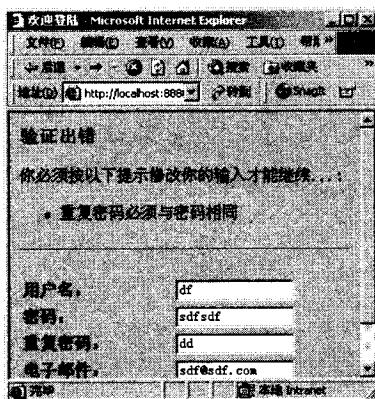


图 3.26 客户端校验失效，服务器端校验自动生效

可见，借助于 `common-validator` 框架，使 Struts 的校验功能更加强大。

## 3.10 Struts 的异常框架

Struts 1.1 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框架，Struts 越来越趋于成熟。

在以前的 Struts 开发过程中，对于异常的处理，主要是采用手动处理的方式：如通过 `try/catch` 等捕获异常；然后将定制个性化的，比较详细的错误信息放进 `ActionMessage` 中；最后在返回页面中把这些错误信息反馈给用户。

对于异常的原始信息，不管是最终用户还是开发员都不希望看到。

借助于 Struts 的异常框架，异常处理只需通过 `struts-config.xml` 文件定义即可。根据异常定义的位置不同，异常可分为局部异常和全局异常两种。

- 局部异常作为 `action` 的子元素中定义。
- 全局异常在 `global-exceptions` 元素中定义。

异常定义的格式如下：

`<exception key="keyName" type="ExceptionName" scope="scope" path="uri"/>`：当 Struts 出现 `ExceptionName` 的异常时，页面自动转向 `uri` 指向的资源，并在该页面输出 `keyName` 对应的出错提示。

下面的示例演示了 Struts 的异常控制。该逻辑组件并未执行真实逻辑，但业务逻辑方法会抛出两个业务逻辑异常：

```

public class BussService
{
    //业务逻辑组件的构造器
    public BussService()
    {
    }

    //业务逻辑方法，该方法会抛出两个业务逻辑异常
    public boolean valid(String username , String password)
        throws ExceptionTestA,ExceptionTestB
    {
        //当用户名为 username 时抛出 ExceptionTestA 异常
        if (username.equals("username"))
        {
            throw new ExceptionTestA("异常 A");
        }
        //当密码为 password 时抛出 ExceptionTestB 异常
        else if (password.equals("password"))
        {
            throw new ExceptionTestB("异常 B");
        }
        else if(username.equals("yeeku") && password.equals("123456"))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

Struts Action 的 execute 方法被设计成可抛出任何异常。因此，重写该方法也可以抛出任何异常，保证了 execute 方法在调用业务逻辑方法时无须使用 try...catch 块。

下面是使用的 Action 的源代码：

```

public class LoginAction extends Action
{
    //重写 execute 方法，抛出 Exception 异常
    public ActionForward execute(ActionMapping mapping , ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //解析请求参数
        DynaValidatorForm registForm = (DynaValidatorForm)form;
        String username = (String)registForm.get("username");
        String password = (String)registForm.get("password");
        //直接调用业务逻辑组件的业务逻辑方法，无须 try...catch 处理异常
        BussService bs = new BussService();
        if (bs.valid(username,password))
        {
            return (mapping.findForward("success"));
        }
        return (mapping.findForward("failure"));
    }
}

```

Action 的 execute 方法抛出了全部异常，如果这些异常得不到合适处理，将会直接

传播到客户端浏览器中，这是绝对禁止的。好在 Struts 的异常处理框架可以负责处理这些异常。

而 Struts 的异常处理只需在 struts-config.xml 文件中配置即可，下面是 struts-config.xml 文件的源代码：

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Struts 配置文件的文件头，包含 DTD 等信息-->
<!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>
    <!-- 定义 ActionForm，此处使用动态、验证 Form-->
    <form-beans>
        <form-bean name="loginForm" type="org.apache.struts.validator.
            DynaValidatorForm">
            <form-property name="username" type="java.lang.String"/>
            <form-property name="password" type="java.lang.String"/>
        </form-bean>
    </form-beans>
    <!-- 全局异常定义-->
    <global-exceptions>
        <!-- 定义 lee.exceptionExceptionTestA 对应的处理-->
        <exception key="test.exceptionA" type="lee.exception.ExceptionTestA"
            scope="request" path="/error.jsp"/>
        <!-- 定义 lee.exceptionExceptionTestA 对应的处理-->
        <exception key="test.unknown" type="java.lang.Exception"
            scope="request" path="/error.jsp"/>
    </global-exceptions>
    <action-mappings>
        <action path="/processLogin" type="lee.LoginAction" name="loginForm"
            scope="request" validate="true" input="/login.jsp">
            <!-- 定义了局部 Exception-->
            <exception key="test.exceptionB" type="lee.exception.
                ExceptionTestB" path="/error.jsp"/>
            <forward name="failure" path="/failure.jsp"/>
            <forward name="success" path="/success.jsp"/>
        </action>
    </action-mappings>
    <!-- 加载国际化资源文件-->
    <message-resources parameter="MyResource"/>
    <!-- 加载验证的资源文件 -->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.xml,
            /WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true" />
    </plug-in>
</struts-config>
```

异常的定义用于确定当某个异常出现时，系统自动转向某个异常显示页面。在上面的定义中，有两个全局异常处理定义：

```
<exception key="test.exceptionA" type="lee.exception.ExceptionTestA"
    scope="request" path="/error.jsp"/>
```

该定义表明，当 Action 的 execute 方法中抛出 lee.exception.ExceptionTestA 时，Struts 将重定向到/error.jsp 页面。该异常定义是全局异常定义，因此当每个 Action 的 execute

方法抛出该异常时，都将可以自动重定向。

而局部异常定义只对本 Action 有效。（作为 action 元素的子元素定义的 exception 是局部异常）

```
<exception key="test.exceptionB" type="lee.exception.ExceptionTestB" path="/error.jsp"/>
```

如果在用户名的文本框中输入任意字符串（只要不是 username），而在密码框中输入 password，然后提交表单。根据业务逻辑方法，将抛出未处理的 lee.exception.ExceptionTestB，此时，Struts 的异常框架将开始工作，于是出现如图 3.27 所示的运行效果。

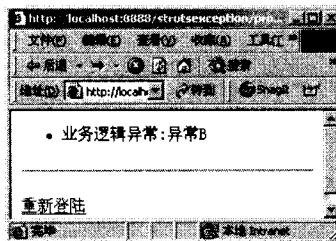


图 3.27 异常的处理效果

图 3.27 页面中出现的异常提示信息则存在资源文件里，资源文件代码如下：

```
#定义 Exception
test.exceptionA=数据库访问异常:{0}
test.exceptionB=业务逻辑异常:{0}
test.unknown=未知异常
```

借助于 Struts 的异常处理框架，将异常处理部分交给 Struts 框架完成，可以避免使用烦琐的 try...catch 块。

## 3.11 几种常用的 Action

除了基本的 Action 之外，Struts 还提供了几个其他类型的 Action，这些 Action 大大丰富了 Struts 的功能。下面介绍如下几个常用的 Action。

- **DispatchAction:** 能同时完成多个 Action 功能的 Action。
- **ForwardActon:** 该类用来整合 Struts 和其他业务逻辑组件，通常只对请求作有效性检查。
- **IncludeAction:** 用于引入其他的资源和页面。
- **LookupDispatchAction:** DispatchAction 的子类，根据按钮的 key，控制转发给 action 的方法。
- **MappingDispatchAction:** DispatchAction 的子类，一个 action 可映射出多个 Action 地址。
- **SwitchAction:** 用于从一个模块转换至另一个模块，如果应用分成多个模块时，

就可以使用 `SwitchAction` 完成模块之间的切换。  
下面对常用的 Action 进行介绍。

### 3.11.1 DispatchAction 及其子类

`DispatchAction` 是仅次于 `Action`，使用最频繁的 `Action`。用于同一个表单中有两个提交按钮时，但提交需要的逻辑处理完全不同的情况。如图 3.28 所示为登录页面。

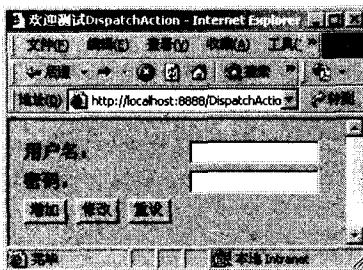


图 3.28 两个提交按钮的表单页

该页面包含了两个提交按钮，但提交按钮需要执行的逻辑却不一样。最容易想到的解决方法是，为每个按钮增加 JavaScript 脚本，提交两个按钮时候分别提交给不同的 `Action` 处理。这是最容易想到，也最麻烦的方式。

Struts 提供了 `DispatchAction`，可支持多个逻辑处理。对于上面的示例，表单需要两个逻辑处理：增加和修改。下面是示例所使用的 `Action` 类的源代码：

```
public class LoginAction extends DispatchAction
{
    //第一个处理逻辑
    public ActionForward add(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        System.out.println("增加");
        request.setAttribute("method", "增加");
        return mapping.findForward("success");
    }
    //第二个处理逻辑
    public ActionForward modify(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        System.out.println("修改");
        request.setAttribute("method", "修改");
        return mapping.findForward("success");
    }
}
```

上面的 `Action` 非常简单，其两个逻辑处理也非常简单。该 `Action` 并没有重写 `execute` 方法，而是书写了两个自定义的方法：`add` 和 `modify`。这两个方法除了方法名与 `execute`

方法不同之外，其他的参数列表及异常的处理完全相同。这两个方法正是 `execute` 方法的替代，用于完成业务逻辑的处理。

问题的关键是：Struts 如何区别不同表单提交与方法之间的对应关系？因为当使用 `DispatchAction` 要求表单提交时，会额外多传递一个参数，该参数用于区分到底调用 Action 中的哪个方法。

这个参数名在 `struts-config.xml` 文件中指定。注意下面 action 的配置代码：

```
<action path="/login" type="lee.LoginAction" name="loginForm"
    scope="request" validate="true" input="/login.jsp" parameter="method">
    <forward name="success" path="/welcome.jsp"/>
</action>
```

在该 action 的配置中，增加了 `parameter` 属性，该属性用于指定参数名，即 Struts 将根据该参数的值调用对应的方法。为了让请求增加 `method` 的参数，对上面的 JSP 页面代码进行简单修改，可在 JSP 页面中增加一个隐藏域，使该隐藏域的名字为 `method`。下面是 JSP 页面的表单代码：

```
<html:form action="login.do">
<table border="0" width="100%">
<tr>
    <th align="left"><bean:message key="username"/></th>
    <td align="left"><html:text property="username" size="15"/></td>
</tr>
<tr>
    <th align="left"><bean:message key="pass"/></th>
    <td align="left"><html:text property="pass" size="15"/></td>
</tr>
<tr>
    <td>
        <input type="hidden" name="method" value="add"/>
        <input type="submit" value='<bean:message key="button.add"/>' onClick="method.value='add'"/>
        <input type="submit" value='<bean:message key="button.modify"/>' onClick="method.value='modify'"/>
        <input type="reset" value='<bean:message key="button.reset"/>' />
    </td>
</tr>
</table>
</html:form>
```

从上面的代码中可以看到，页面中增加了 `method` 的隐藏域，该隐藏域的默认值为 `add`，当单击页面中的【修改】按钮时，该隐藏域的值将变成 `modify`，单击【添加】按钮时，该隐藏域的值变成 `add`。这个隐藏域就是额外传递的参数值，用于告诉 Dispatch 调用哪个方法来处理请求。

如果 `method` 参数的值为 `add`，将调用 `add` 方法；如果 `method` 参数的值为 `modify`，则调用 `modify` 方法。因此在单击不同按钮时，`DispatchAction` 将可自动调用对应的方法来完成处理。

## 1. 使用 `MappingDispatchAction`

`MappingDispatchAction` 可将同一个 Action 的不同方法映射成多个 Action URI，这种

Action 的写法与 DispatchAction 非常相似，同样不需要重写 execute 方法，而是将书写多个自定义的方法。这些方法除了方法名与 execute 方法不同外，其他的参数列表及异常处理完全一样。

下面是本示例所使用的 Action 的源代码：

```
public class LoginAction extends MappingDispatchAction
{
    //第一个处理逻辑
    public ActionForward add(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        System.out.println("增加");
        request.setAttribute("method", "增加");
        return mapping.findForward("success");
    }
    //第二个处理逻辑
    public ActionForward modify(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        System.out.println("修改");
        request.setAttribute("method", "修改");
        return mapping.findForward("success");
    }
}
```

该 Action 与前面的 DispatchAction 没有太大的区别，仅仅改变它的父类：MappingDispatchAction，但变化在于该 Action 的配置，看下面关于该 Action 的配置代码：

```
<!-- 配置第一个 Action，实现类是 lee.LoginAction，parameter 为 add-->
<action path="/add" type="lee.LoginAction" name="loginForm"
    scope="request" validate="true" input="login.jsp" parameter="add">
    <forward name="success" path="/welcome.jsp"/>
</action>
<!-- 配置第二个 Action，实现类是 lee.LoginAction，parameter 为 modify-->
<action path="/modify" type="lee.LoginAction" name="loginForm"
    scope="request" validate="true" input="login.jsp" parameter="modify">
    <forward name="success" path="/welcome.jsp"/>
</action>
```

在这种情况下，两个 action 使用的是同一个 Action 处理类，只是调用的方法不同，同样也可达到上面的效果。当然也需要为页面中的两个按钮增加相应的 JavaScript 脚本，当单击不同按钮时，表单可提交到不同的 action，下面是 JSP 页面三个按钮的源代码：

```
<td>
    <input type="submit" value='<bean:message key="button.add"/>' 
        onClick="document.loginForm.action='add.do'"/>
    <input type="submit" value='<bean:message key="button.modify"/>' 
        onClick="document.loginForm.action='modify.do'"/>
    <input type="reset" value='<bean:message key="button.reset"/>' />
</td>
```

其中，前面两个提交按钮都增加了 onClick 方法，即单击该按钮时，会改变表单的提交地址。

注意：使用 `MappingDispatchAction` 并没有带来太大的优势，系统完全可以书写两个 Action，分别定义两个不同的 action 映射，而其他部分没有区别。

## 2. 使用 `LookupDispatchAction`

`LookupDispatchAction` 也是 `DispatchAction` 的一种，但它的处理更加简单。该 Action 也可包含多个处理方法，它可让处理方法与按钮直接关联，无须使用任何的 JavaScript 脚本。

使用 `LookupDispatchAction` 时，提交按钮必须使用 Struts 的 html 标签，下面是该示例按钮部分的源代码：

```
<td>
    <html:submit property="method">
        <bean:message key="button.add"/>
    </html:submit>
    <html:submit property="method">
        <bean:message key="button.modify"/>
    </html:submit>
    <input type="reset" value='<bean:message key="button.reset"/>' />
</td>
```

代码中两个提交按钮分别增加了 `property` 属性，该属性的值为 `method`。而在 action 的配置中，也使用 `parameter` 作为参数，看下面的 action 配置代码：

```
<action path="/login" type="lee.LoginAction" name="loginForm"
    scope="request" validate="true" input="/login.jsp" parameter="method">
    <forward name="success" path="/welcome.jsp"/>
</action>
```

这段配置代码表明：该 action 也根据 `method` 参数来区分请求分别调用哪个方法，此时无须使用 `method` 的隐藏域，而是将按钮的 `property` 设为 `method`。通过这种方式可以避免书写 JavaScript 脚本。

因此可通过重写 `getKeyMethodMap` 方法完成按钮与 Action 中方法的关联，下面是该 Action 的源代码：

```
public class LoginAction extends LookupDispatchAction
{
    //用于关联按钮和方法
    protected Map getKeyMethodMap()
    {
        Map map = new HashMap();
        //如果按钮标题的 key 为 button.add，则提交该按钮时对应 add 方法
        map.put("button.add", "add");
        //如果按钮标题的 key 为 button.modify，则提交该按钮时对应 modify 方法
        map.put("button.modify", "modify");
        return map;
    }
    //第一个处理逻辑
    public ActionForward add(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
```

```

        System.out.println("增加");
        request.setAttribute("method", "增加");
        return mapping.findForward("success");
    }
    第二个处理逻辑
    public ActionForward modify(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        System.out.println("修改");
        request.setAttribute("method", "修改");
        return mapping.findForward("success");
    }
}

```

LookupDispatchAction 必须重写 getKeyMethodMap 方法，该方法返回一个 Map 对象，并在该对象内保存了按钮标题与方法之间的对应。

### 3.11.2 使用 ForwardAction

如果需要从一个页面或资源转换到另一个资源时，直接使用页面或资源路径的超级链接定位并不是好的做法，这使得控制器没有机会处理相关的请求事宜。

使用 ForwardAction 可以完成请求的转发，当控制器调用 ForwardAction 的 perform() 方法时，它会使用属性 parameter 所设定的路径进行 forward 的动作。下面是一个设定 ForwardAction 的例子：

```

<action-mappings>
    <action path="/welcome"
        type="org.apache.struts.actions.ForwardAction"
        parameter="/welcome.jsp"/>
</action-mappings>

```

该 action 仅仅完成转发，并没有执行其他的额外动作。

页面控制转发的代码如下：

```
<a href="welcome.do">转入</a>
```

当单击转入超级链接时，将可以转向 ForwardAction 中 parameter 指向的资源。

### 3.11.3 使用 IncludeAction

IncludeAction 的用法与 ForwardAction 的用法比较相似，区别在于 ForwardAction 将跳转到 action 定义的资源，而 IncludeAction 用于引入该 action 对应的资源。

下面是 IncludeAction 定义的源代码：

```

<action-mappings>
    <action path="/welcome"
        type="org.apache.struts.actions.IncludeAction"
        parameter="/welcome.jsp"/>
</action-mappings>

```

该 action 用于经 welcome.jsp 作为资源导入。

页面中负责加载该 action 所导入资源的代码如下：

```
<jsp:include page="welcome.do"/><br>
```

上面的代码将会把 welcome action 定义的资源导入该页面。

### 3.11.4 使用 SwitchAction

SwitchAction 主要用于模块之间的切换。当一个应用之中存在多个模块时，使用 SwitchAction 在不同模块之间的 action 之间切换还是相当方便的。

在下面的 web.xml 中，加载了 Struts 的两个配置文件，其中一个作为系统的一个模块加载，该 web.xml 的配置代码如下：

```
<servlet>
    <!-- 定义 Struts 的核心控制器-->
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!-- 指定 Struts 的第一个配置文件-->
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <!-- 指定 Struts 的第二个配置文件，作为 wawa 模块配置-->
    <init-param>
        <param-name>config/wawa</param-name>
        <param-value>/WEB-INF/struts-config1.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

该应用包括了一个 wawa 的模块，并在 struts-config1.xml 文件中配置一个 action，该 action 的配置代码如下：

```
<action-mappings>
    <action path="/welcome" forward="/welcome.jsp"/>
</action-mappings>
```

该 action 的定义非常简单，仅完成页面的转向。如果现在需要从应用的页面请求该 action，可以使用如下 SwitchAction。

定义 SwitchAction 也相當簡單，只需要定义 path、type 属性即可。下面是 SwitchAction 的定义代码：

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

在使用 SwitchAction 时，必须在请求中带两个参数：第一个是 prefix，用来指定模块名称；另一个是 page，用来指定相模块中的资源路径。下面是页面中超级链接对 wawa 模块的 welcome action 请求，页面的超级链接代码如下：

```
<a href="moduleSwitch.do?prefix=/wawa&page=/welcome.do">转入另一个模块</a>
```

上面的超级链接地址中, /wawa 是模块名, 而 page 对应 wawa 模块下的 welcome 的 action。

## 3.12 Struts 的常见扩展方法

Struts 的强大吸引力还来自于它的可扩展性, 其扩展性通常有如下三种方式。

- 实现 PlugIn: 如果需要在应用启动或关闭时完成某些操作, 可以创建自己的 PlugIn 类。
- 继承 RequestProcessor: 如果想在请求被处理中的某个时刻做一些业务逻辑时, 可以考虑实现自己的 RequestProcessor 类。
- 继承 ActionServlet: 如果需要在每次开始处理请求之前, 或者处理请求结束之后完成某些操作, 可以实现自己的 ActionServlet 来完成扩展。

下面分别从三个方面来介绍 Struts 的扩展。

### 3.12.1 实现 PlugIn 接口

Struts 已经演示了 PlugIn 的扩展方法: 与 common-validation 的整合。后面还将介绍 Spring 与 Struts 的整合, 也利用了 PlugIn 的扩展。

在下面的应用中, 系统使用 Hibernate 作为持久层, 在启动时创建 SessionFactory 实例, 并将该 SessionFactory 存入 application, 在应用关闭时销毁 SessionFactory。只需如下两步即可完成此功能。

(1) 实现自己的 PlugIn 类。

实现 PlugIn 接口必须实现如下两个方法。

- void destroy()。
- void init(ActionServlet servlet, ModuleConfig config)。

应用启动时调用 init 方法, 而应用关闭时则调用 destroy 方法。

下面是 SessionFactoryLoaderPlugIn 的实现类:

```
public class SessionFactoryLoaderPlugIn implements PlugIn
{
    //Hibernate 的配置文件
    private String configFile;
    //应用关闭时, 销毁资源
    public void destroy()
    {
        System.out.println("系统销毁 SessionFactory");
    }
    //应用启动时, 完成 SessionFactory 的初始化
    public void init(ActionServlet actionServlet, ModuleConfig config)
        throws ServletException
    {
        System.out.println("系统以 " + getConfigFile() + " 为配置文件初始化
SessionFactory");
    }
}
```

```
        }
        //获取 PlugIn 配置文件的方法
        public String getConfigFile()
        {
            return configFile;
        }
        //负责加载 PlugIn 配置属性的方法
        public void setConfigFile(String configFile)
        {
            this.configFile = configFile;
        }
    }
```

在上面的 PlugIn 中，并没有真正初始化 SessionFactory，仅在控制台打印出字符串来标识创建动作。另外，还提供了 configFile 属性的 setter 和 getter 方法，这两个方法负责访问 plugin 元素的 configFile 属性。

(2) 将 SessionFactoryLoaderPlugIn 配置在 struts-config.xml 文件中。方法与 ValidatorPlugIn 的配置并没有区别，下面是配置 SessionFactoryLoaderPlugIn 的代码：

```
<plug-in className="lee.SessionFactoryLoaderPlugIn">
    <set-property property="configFile" value="/WEB-INF/hibernate.cfg.xml" />
</plug-in>
```

在配置 SessionFactoryLoaderPlugIn 时，配置了 configFile 属性，该属性用于确定 Hibernate 配置文件的文件名。

图 3.29 显示了应用启动时的提示。



图 3.29 使用 PlugIn 后的启动提示

图 3.29 的红色标记部分表明 PlugIn 随应用启动时完成了 SessionFactory 的加载。

注意：本示例并未真正加载 SessionFactory，因为这不是本示例的重点。

## 3.12.2 继承 RequestProcessor

RequestProcessor 是 Struts 的核心类，而 Struts 的核心控制器是 ActionServlet。但 ActionServlet 并未完成真正的处理，只是调用 RequestProcessor，RequestProcessor 才是 Struts 的核心类。

扩展 RequestProcessor 的实例在 Spring 中有个示范，Spring 提供的 Delegating RequestProcessor 是一个很好的示例。下面示例对 RequestProcessor 进行简单的扩展。

RequestProcessor 包含了如下常见的方法。

- ActionForm processActionForm: RequestProcessor 填充 ActionForm 时执行该方法。

- Action processActionCreate: RequestProcessor 调用 Action 时调用该方法。
- boolean processPreprocess: 预处理用户请求时执行该方法。
- boolean processValidate: 处理输入校验时调用该方法。

扩展 RequestProcessor 只需两步即可。

(1) 实现自己的 RequestProcessor。自己的 RequestProcessor 通常都是 RequestProcessor 的子类, 下面是 MyRequestProcessor 的源代码:

```
public class MyRequestProcessor extends RequestProcessor
{
    Date t1;
    //填充 ActionForm 时调用的方法
    protected ActionForm processActionForm(HttpServletRequest request,
        HttpServletResponse response, ActionMapping mapping)
    {
        t1 = new Date();
        System.out.println(t1 + "=====ActionServlet 开始填充 ActionForm");
        //调用父类方法
        return super.processActionForm(request, response, mapping);
    }
    //创建 Action 时调用的方法
    protected Action processActionCreate(HttpServletRequest request,
        HttpServletResponse response, ActionMapping mapping)
        throws java.io.IOException
    {
        t1 = new Date();
        System.out.println(t1 + "=====ActionServlet 开始实例化 Action");
        //调用父类方法
        return super.processActionCreate(request, response, mapping);
    }
    //处理用户请求时执行的方法
    protected boolean processPreprocess(HttpServletRequest request,
        HttpServletResponse response)
    {
        System.out.println("----- 开始显示请求相关信息-----");
        System.out.println("请求获取的 URI = " + request.getRequestURI());
        System.out.println("请求访问的虚拟路 = " + request.getContextPath());
        Cookie cookies[] = request.getCookies();
        if (cookies != null)
        {
            System.out.println("Cookies:");
            for (int i = 0; i < cookies.length; i++)
            {
                System.out.println(cookies[i].getName() + " = " + cookies[i].getValue());
            }
        }
        Enumeration headerNames = request.getHeaderNames();
        System.out.println("请求头:");
        while (headerNames.hasMoreElements())
        {
            String headerName = (String)headerNames.nextElement();
            Enumeration headerValues = request.getHeaders(headerName);
            while (headerValues.hasMoreElements())
            {
                String headerValue = (String)headerValues.nextElement();
            }
        }
    }
}
```

```
        System.out.println("      " + headerName + " = " + headerValue);
    }
}
System.out.println("本地化语言环境 = " + request.getLocale());
System.out.println("请求的方法 = " + request.getMethod());
System.out.println("请求协议 = " + request.getProtocol());
System.out.println("请求的远程地址 = " + request.getRemoteAddr());
System.out.println("请求的远程主机 = " + request.getRemoteHost());
String address = request.getRemoteAddr();
System.out.println("请求的用户名 = " + request.getRemoteUser());
System.out.println("请求的Session Id= " + request.getRequestedSessionId());
System.out.println("请求配置 = " + request.getScheme());
System.out.println("服务器名 = " + request.getServerName());
System.out.println("服务器端口 = " + request.getServerPort());
System.out.println("请求的资源地址 = " + request.getServletPath());
System.out.println("是否加密 = " + request.isSecure());
System.out.println("-----结束显示请求相关信息-----");
int position = address.lastIndexOf(".");
int last = Integer.parseInt(address.substring(position + 1));
if (address.substring(0, 9).equals("192.168.6") && (last < 10 && last >= 1))
{
    return true;
}
if (address.equals("127.0.0.1"))
{
    return true;
}
return false;
}
}
```

本示例继承了 RequestProcessor 类，并重写该类的 processPreprocess 方法。如果该方法返回 false，Struts 将不再继续调用业务逻辑控制器 action。本示例对用户的请求作出分析，然后对局域网内的 IP 地址进行过滤，如果 IP 地址不在 192.168.1.1 和 192.168.1.10 之间，将返回 false。

(2) 在 struts-config.xml 文件中配置 MyRequestProcessor。用户重写了 RequestProcessor，但 Struts 并不知道，必须在 struts-config.xml 中配置才可以。

下面是配置 MyRequestProcessor 的代码：

```
<controller processorClass="lee.MyRequestProcessor" />
```

该属性的配置应该放在 action-mappings 元素之后。

**注意：**重写 RequestProcessor 的方法时，别忘了使用 super 来调用父类的动作。如果没有调用该方法，意味着开发者必须完成 Struts 框架所完成的动作。这是不应该的，因为程序员只是在框架中加入额外的处理，并不是要替代 Struts。

### 3.12.3 继承 ActionServlet

如果需要在开始处理请求，或者处理结束之后加入自己的处理时，可对 ActionServlet 进行扩展。例如解决中文的编码问题。

ActionServlet 接收处理请求参数时，并不是按 GBK 的解码方式处理请求，因此容易形成乱码。为了解决该问题，可以强制指定 ActionServlet 使用 GBK 的解码方式。

实现该功能只需要两个步骤。

(1) 实现自己的 ActionServlet，它是 ActionServlet 的子类：

```
public class MyActionServlet extends ActionServlet
{
    protected void process(HttpServletRequest request,
                           HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException
    {
        //设置解码方式
        request.setCharacterEncoding("GBK");
        //调用父类的方法
        super.process(request, response);
    }
}
```

在本示例中，重写了 process 方法，该方法是 ActionServlet 处理用户请求的方法。当然，该方法会调用 RequestProcessor 处理，首先在重写该方法的第一行设计解码方式，然后调用父类的方法。

**注意：**重写 ActionServlet 的方法时，别忘了调用父类的同名方法。否则 Struts 将完全停止工作。

(2) 在 web.xml 文件中配置 MyActionServlet。由于系统改变了 ActionServlet，因此必须使用 MyActionServlet 来拦截所有的用户请求。

下面是 MyActionServlet 的配置代码：

```
<servlet>
    <!-- 配置核心处理器-->
    <servlet-name>action</servlet-name>
    <!-- 使用自己的核心处理器-->
    <servlet-class>lee.MyActionServlet</servlet-class>
    <!-- 配置自动加载-->
    <load-on-startup>1</load-on-startup>
</servlet>
```

经过上面的配置，Struts 可以正确处理请求中的中文参数。

## 本章小结

本章全面介绍了经典的 MVC 框架：Struts，包括 Struts 的基本使用和基本配置，对 Struts 的流程进行深入剖析，并通过示例讲解了 Struts 的程序国际化，动态表单。

另外重点讲解了 Struts 的验证框架和异常处理框架，使读者真正掌握 Struts 的使用。

最后结合示例详细讲解了 Struts 的常用标签，并对 Struts 的常用 Action 也进行了详细的讲解，最后给出了 Struts 常用的扩展方法。

# 第4章

## 使用 Hibernate 完成持久化

### 本章要点

- 『 ORM 的基本知识
- 『 Hibernate 入门知识
- 『 Hibernate 基本映射
- 『 集合属性，引用属性，复合主键映射
- 『 关联关系映射
- 『 HQL 查询、条件查询
- 『 SQL 查询、过滤等
- 『 Hibernate 的事件机制

Hibernate 是目前最流行的开源对象关系映射（ORM）框架。Hibernate 采用低侵入式的设计，完全采用普通的 Java 对象（POJO），而不必继承 Hibernate 的某个超类或实现 Hibernate 的某个接口。因为 Hibernate 是面向对象的程序设计语言和关系数据库之间的桥梁，所以 Hibernate 允许程序开发者采用面向对象的方式来操作关系数据库。

## 4.1 ORM 简介

ORM 的全称是 Object/Relation Mapping，对象/关系映射。ORM 也可理解是一种规范，具体的 ORM 框架可作为应用程序和数据库的桥梁。目前 ORM 的产品非常多，比如 Apache 组织下的 OJB；Oracle 组织下的 TopLink、JDO 等。

### 4.1.1 什么是 ORM

ORM 并不是一种具体的产品，而是一类框架的总称。它概述了这类框架的基本特征：完成面向对象的程序设计语言与关系数据库的映射。基于 ORM 框架完成映射后，既可利用面向对象程序设计语言的简单易用性，又可利用关系数据库的技术优势。

ORM 框架是面向对象程序设计语言与关系数据库发展不同步时的中间解决方案。笔者认为，随着面向对象数据库的发展，其理论逐步完善，最终会取代关系数据库。只是这个过程不可一蹴而就，ORM 框架在此期间内会蓬勃发展，但随着面向对象数据库的出现，ORM 工具也会退出历史舞台。

### 4.1.2 为什么需要 ORM

在上一节已经基本回答了这个问题，面向对象的程序设计语言代表了目前程序设计语言的主流和趋势，其具备非常多的优势，比如：

- 面向对象的建模与操作。
- 多态及继承。
- 摈弃难以理解的过程。
- 简单易用，易理解性。

但数据库的发展并未与程序设计语言同步，而且关系数据库系统的某些优势也是面向对象的语言目前无法解决的。比如：

- 大量数据操作查找与排序。
- 集合数据连接操作与映射。
- 数据库访问的并发与事务。
- 数据库的约束与隔离。

面对这种面向对象语言与关系数据库系统并存的局面，采用 ORM 就变成一种必然。

### 4.1.3 流行的 ORM 框架介绍

目前 ORM 框架的产品非常多，除了各大著名公司的产品外，甚至其他一些小团队也都有推出自己的 ORM 框架。目前流行的 ORM 框架有如下产品。

- 大名鼎鼎的 Hibernate

Hibernate 出自 Gavin King 的手笔，是目前最流行的开源 ORM 框架，其灵巧的设计，优秀的性能，以及丰富的文档都是其迅速风靡全球的重要因素。

- 传统的 Entity EJB

Entity EJB 实质上也是一种 ORM 技术，是一种备受争议的组件技术，很多人说它非常优秀，也有人说它一钱不值。事实上，EJB 为 J2EE 的蓬勃发展赢得了极高的声誉。就笔者的实际开发经验而言，EJB 作为一种重量级，高花费的 ORM 技术，具有不可比拟的优势。但由于其必须运行在 EJB 容器内，而且学习曲线陡峭，开发周期及成本相对较高，因而限制了 EJB 的广泛使用。

- iBATIS

Apache 软件基金组织的子项目，与其称它是一种 ORM 框架，不如称它是一种“Sql Mapping”框架。相对 Hibernate 的完全对象化封装，iBATIS 更加灵活，但开发过程中开发人员需要完成的代码量更大，而且需要直接编写 SQL 语句。

- Oracle 的 TopLink

作为一个遵循 OTN 协议的商业产品，TopLink 在开发过程中可以自由下载和使用，但作为商业产品使用后，则需要收取费用。可能正是这一点，导致了 TopLink 的市场占有率低下。

- OJB

OJB 是 Apache 软件基金组织的子项目。开源的 ORM 框架，但由于开发文档不多，而且 OJB 的规范并不稳定，因此并未在开发者中赢得广泛的支持。

## 4.2 Hibernate 概述

Hibernate 是目前最流行的 ORM 框架，其采用非常优雅的方式将 SQL 操作完全包装成对象化的操作。其作者 Gavin King 在持久层设计上极富经验，采用非常少的代码实现了整个框架，同时完全开放源代码，即使偶尔遇到无法理解的情况，也可以参照源代码来理解其在持久层上灵巧而智能的设计。

目前 Hibernate 在国内的开发人员相当多，Hibernate 的文档也非常丰富，这些都为学习 Hibernate 铺平了道路，因而 Hibernate 的学习相对简单一些。下面通过对比来了解 Hibernate 和传统 JDBC 操作数据库持久层之间的差异。

### 4.2.1 Hibernate 的起源

当前的软件开发语言已经全面转向面向对象，而数据库系统仍停留在关系数据库阶段。面对复杂的企业环境，同时使用面向对象语言和关系数据库是相当麻烦的，不但中间的过渡难以理解，而且其开发周期也相当长。

Hibernate 是一个面向 Java 环境的对象/关系数据库映射工具。对象/关系数据库映射

(Object/Relational Mapping) 表示一种技术，用来把对象模型表示的对象映射到基于 SQL 的关系模型数据结构中去。

Hibernate 的目标是：释放开发者通常的数据持久化相关的编程任务的 95%。对于以数据为中心的程序而言，往往在数据库中使用存储过程来实现商业逻辑，Hibernate 可能不是最好的解决方案。但对于那些基于 Java 的中间件应用中，设计采用面向对象的业务模型和商业逻辑时，Hibernate 是最有用的。不管怎样，Hibernate 能消除那些针对特定数据库厂商的 SQL 代码，并且把结果集由表格式的形式转换成值对象的形式。

Hibernate 不仅管理 Java 类到数据库表的映射（包括 Java 数据类型到 SQL 数据类型的映射），还提供数据查询和获取数据的方法，可以大幅度地减少在开发时人工使用 SQL 和 JDBC 处理数据的时间。

## 4.2.2 Hibernate 与其他 ORM 框架的对比

Hibernate 能在众多的 ORM 框架中脱颖而出，因为 Hibernate 与其他 ORM 框架对比具有如下优势。

- 开源和免费的 License，方便需要时研究源代码、改写源代码并进行功能定制。
- 轻量级封装，避免引入过多复杂的问题，调试容易，减轻程序员负担。
- 具有可扩展性，API 开放。功能不够用时，可以自己编码进行扩展。
- 开发者活跃，产品有稳定的发展保障。

## 4.3 Hibernate 的安装和使用

Hibernate 的学习难度不大，简单易用。正是这种易用性，征服了大量的开发者。下面简要介绍 Hibernate 的安装和使用。

### 4.3.1 Hibernate 下载和安装

Hibernate 目前的最新版本是 3.1.2，本章所用的代码也是基于该版本测试通过的。安装和使用 Hibernate 请按如下步骤进行：

- 首先登录 <http://www.hibernate.org> 网站，下载 Hibernate 的二进制包（windows 平台下载 zip 包，Linux 平台下载 tar 包）。
- 解压缩下载的压缩包，在 hibernate-3.1 路径下有个 hibernate3.jar 的压缩文件，该文件是 Hibernate 的核心类库文件。该路径下还有 lib 路径，该路径包含 Hibernate 编译和运行的第三方类库。关于这些类库的使用请参看该路径下的 readme.txt 文件。
- 将必需的 Hibernate 类库添加到 CLASSPATH 里，或者使用 ANT 工具。总之，编译和运行时可以找到这些类即可。在 Web 应用中，则应该将这些类库复制到 WEB-INF/lib 下。

### 4.3.2 传统 JDBC 的数据库操作

先看这样一个需求：向数据库里增加一条新闻，该新闻有新闻 Id、新闻标题及新闻内容三个属性。在传统的 JDBC 数据库访问里，实现此功能并不难。

我们可采用如下方法来实现（本程序采用 MySql 数据库）：

```
import java.sql.*;
public class NewsDao
{
    /**
     * @param News 需要保存的新闻实例
     */
    public void saveNews(News news)
    {
        Connection conn = null;
        PreparedStatement pstmt = null;
        int newsId = news.getId();
        String title = news.getTitle();
        String content = news.getContent();
        try
        {
            //注册驱动
            Class.forName("com.mysql.jdbc.Driver");
            /* hibernate: 想连接的数据库
             * user: 连接数据库的用户名
             * pass: 连接数据库的密码
             */
            String url="jdbc:mysql://localhost/hibernate?user=root&password=pass";
            //获取连接
            conn= DriverManager.getConnection(url);
            //创建预编译的 Statement
            pstmt=conn.prepareStatement("insert into news_table values(?, ?, ?)");
            //下面语句为预编译 Statement 传入参数
            pstmt.setInt(1,newsId);
            pstmt.setString(2,title);
            pstmt.setString(3,content);
            //执行更新
            pstmt.executeUpdate();
        }
        catch (ClassNotFoundException cnf)
        {
            cnf.printStackTrace();
        }
        catch (SQLException se)
        {
            se.printStackTrace();
        }
        finally
        {
            try
            {
                //关闭预编译的 Statement
                if (pstmt != null)pstmt.close();
                //关闭连接
                if (conn != null) conn.close();
            }
            catch (SQLException se2)
```

```
        {
            se2.printStackTrace();
        }
    }
}
```

由此可见，这种操作方式丝毫没有面向对象的优雅和易用，而是一种纯粹的过程式操作。在这种简单的数据库访问里，我们没有过多地感觉到这种方式的复杂与缺陷，相比下面采用 Hibernate 的操作，但我们还是可以体会到 Hibernate 的灵巧。

### 4.3.3 Hibernate 的数据库操作

在使用 Hibernate 之前，首先了解一个概念：PO（Persistent Object）持久化对象。持久化对象的作用是完成持久化操作。简单地说，通过该对象可对数据执行增、删和改的操作，以面向对象的方式操作数据库。

Hibernate 里的 PO 是非常简单的，前面已经说过 Hibernate 是低侵入式的设计，完全采用普通 Java 对象来作为持久化对象使用，看下面的 POJO（普通 Java 对象）类：

```
public class News
{
    int id;
    String title;
    String content;
    public void setId(int id)
    {
        this.id = id;
    }
    public int getId()
    {
        return (this.id);
    }
    .....
}
```

此处笔者并未列出该类的 title 和 content 属性的 setter 与 getter 方法，读者可自行增加。这个类与常规的 JavaBean 没有任何区别，是个非常标准的简单 JavaBean。

这个普通的 JavaBean 目前还不具备持久化操作的能力，为了使其具备持久化操作的能力，Hibernate 应采用 XML 映射文件，该映射文件也是非常简单。下面提供该 XML 文件的全部代码：

```
<?xml version="1.0" encoding="gb2312"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!!--上面四行对所有的 hibernate 映射文件都相同 -->
<!-- hibernate-mapping 是映射文件的根元素 -->
<hibernate-mapping>
    <!-- 每个 class 元素对应一个持久化对象 -->
    <class name="News" table="news_table">
        <!-- id 元素定义持久化类的标识属性 -->
        <id name = "id" unsaved-value = "null">
            <generator class="increment"/>
    
```

```

</id>
<!-- property 元素定义常规属性 -->
<property name="title"/>
<property name="content"/>
</class>
</hibernate-mapping>

```

对这个文件作简单地解释：从 1 到 4 行，是该 XML 文件的文件头部分，定义该文件的 xml 版本和 DTD 声明，对于所有 Hibernate 3.x 的映射文件全部相同。因为 hibernate-mapping 元素是所有 hibernate 映射文件的根元素，这个根元素对所有的映射文件都是相同的。

另外，hibernate-mapping 元素下有子元素 class，每个 class 子元素映射一个 PO，更准确地说，应该是持久化类。可以看到：PO = POJO + 映射文件。

现在就可以通过这个持久化类完成数据库的访问：插入一条新闻。

在插入一条新闻之前，还必须完成 Hibernate 管理数据库的配置——连接数据库所需的用户名、密码及数据库名等等基本信息。连接所需的基本信息配置可通过.properties 的属性文件，或 hibernate.cfg.xml 的方式来配置。本例采用 hibernate.cfg.xml 的配置方式，下面是这个配置文件的详细代码：

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://.hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<!--上面四行对所有的 hibernate 连接配置文件都相同 -->
<!-- hibernate- configuration 是连接配置文件的根元素 -->
<hibernate-configuration>
<session-factory>
    <!-- 指定连接数据库所用的驱动 -->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</
    property>
    <!-- 指定连接数据库的 url, hibernate 连接的数据库名 -->
    <property name="connection.url">jdbc:mysql://localhost/hibernate</
    property>
    <!-- 指定连接数据库的用户名 -->
    <property name="connection.username">root</property>
    <!-- 指定连接数据库的密码 -->
    <property name="connection.password">pass</property>
    <!-- 指定连接池的大小-->
    <property name="connection.pool_size">5</property>
    <!-- 指定数据库名-->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</
    property>
    <!-- 根据需要自动创建数据库-->
    <property name="hbm2ddl.auto">create</property>
    <!-- 罗列所有的映射文件-->
    <mapping resource="News.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

该配置文件非常简单，前面四行都是 XML 的基本定义和 DTD 声明，所有的 Hibernate 配置文件前面四行都完全相同。Hibernate 配置文件的根元素是 hibernate-configuration，根元素里有子元素 session-factory，该元素依次有很多 property 元素，property 元素依次

定义连接数据库的驱动、URL、用户名、密码及数据库连接池的大小等。另外，还定义一个名为 dialect 的属性，该属性定义 Hibernate 连接的数据库类型是 MySQL，Hibernate 会针对该数据库的特性在访问时进行优化。最后一行 mapping 定义了持久化类的映射文件，如果有多个持久化映射文件，可在此处罗列多个 mapping 元素。

下面是完成插入新闻的代码：

```
public class NewsDaoHibernate
{
    Configuration configuration;
    SessionFactory sessionFactory;
    Session session;
    public void saveNews(News news)
    {
        //实例化 Configuration
        configuration=new Configuration().configure();
        //实例化 SessionFactory
        sessionFactory = configuration.buildSessionFactory();
        //实例化 Session
        session = sessionFactory.openSession();
        //开始事务
        Transaction tx = session.beginTransaction();
        //增加新闻
        session.save(news);
        //提交事务
        tx.commit();
        //关闭 Session
        session.close();
    }
}
```

此时的代码结构非常清晰，保存新闻仅仅只需要这个语句：session.save(news)，而且是完全对象化的操作方式，可以说是非常简单明了。

在代码显示执行 session.save(News)之前，首先要获取 Session 对象。PO 只有在 Session 的管理下才可完成数据库访问，按 PO 与 Session 的关系，PO 可有如下三个状态：

- 瞬态
- 持久化
- 脱管

对 PO 的操作必须在 Session 管理下才能与数据库同步。Session 由 SessionFactory 厂商提供，SessionFactory 是数据库编译后的内存镜像，通常一个应用对应一个 SessionFactory 对象，该对象由 Configuration 对象生成。Configuration 对象用来加载 Hibernate 配置文件。

最后使用如下方法来完成对新闻的增加：

```
public static void main(String[] args)
{
    News n = new News();
    n.setTitle("新闻标题");
    n.setContent("新闻内容");
    NewsDaoHibernate ndh = new NewsDaoHibernate();
    ndh.saveNews(n);
```

}

这里仅仅提供一个大体的框架，读者可以将其补充完整，以完成新闻的添加。

## 4.4 Hibernate 的基本映射

在上面的例子里，可看到一个简单的 Hibernate 映射文件，每个 Hibernate 映射文件的基本结构都是相同的。

### 4.4.1 映射文件结构

映射文件的根元素为 `hibernate-mapping` 元素，这个元素下可以拥有多个 `class` 子元素，每个 `class` 子元素对应一个持久化类的映射。如下是一个映射文件的基本结构，在 `hibernate-mapping` 元素下可以有多个 `class` 子元素：

```
<hibernate-mapping>
    <class/>
    <class/>
    .....
</hibernate-mapping>
```

接下来看 `class` 元素，每个 `class` 元素对应一个持久化类。首先必须采用 `name` 元素来指定该持久化类映射的类名，此处的类名应该是全限定的类名。如果不使用全限定的类名，则必须在 `hibernate-mapping` 元素里指定 `package` 元素，该元素指定持久化类所在的包名。

如果需要采用继承映射，则 `class` 元素下还会增加 `subclass` 元素、`joined-subclass` 或 `union-subclass` 元素，这些元素分别用于定义子类。

另外，持久化类都需要有一个标识属性，该标识属性用来标识该持久化类的实例，因此标识属性通常被映射成数据表主键。标识属性通过 `id` 元素来指定，`id` 元素的 `name` 属性的值就是持久化类标识属性名。

标识属性通常应该指定主键生成策略，Hibernate 推荐数据表采用逻辑主键，而不采用有物理含义的实体主键。逻辑主键没有实际意义，仅仅用来标识一行记录，通常由 Hibernate 负责生成。负责生成主键的工具称为主键生成器，应尽量为每个持久化类都设置主键生成器。

### 4.4.2 主键生成器

主键生成器是负责生成数据表记录的主键，通常有如下几种常见的主键生成器。

`increment`: 对 `long`, `short` 或 `int` 的数据列生成自动增长主键。

`identity`: 对如 SQL server, MySQL 等支持自动增长列的数据库，如果数据列的类型是 `long`, `short` 或 `int`，可使用主键生成器生成自动增长主键。

**sequence:** 对如 Oracle, DB2 等支持 Sequence 的数据库, 如果数据列的类型是 long, short 或 int, 可使用该主键生成器生成自动增长主键。

**uuid:** 对字符串列的数据采用 128-位 uuid 算法生成唯一的字符串主键。

**property** 元素定义持久化类的普通属性, 该持久化类有多少个普通属性, 就需要有多少个 **property** 元素, **class** 元素的结构如下所示:

```
<class name="News" table="news_table">
    <!-- 定义标识属性-->
    <id name = "id" unsaved-value = "null">
        <!-- 定义主键生成器-->
        <generator class="increment"/>
    </id>
    <!-- 用于定义普通属性 -->
    <property />
    <property />
</class>
```

**<property/>** 元素的定义相对简单, 基本的 **property** 映射只需要 **name** 属性, **name** 属性映射持久类的属性名。如果想指定属性在数据表里存储的列名, 就可以定义 **column** 属性来强制指定列名, 列名默认与属性名相同。

### 4.4.3 映射集合属性

集合属性也是非常常见的情况, 如每个人的考试成绩, 就是典型的 Map 结构, 其中每门功课对应一个成绩; 或者更简单的集合属性, 如某个企业的部门, 一个企业通常对应多个部门等。集合属性是现实中非常普遍的属性关系。

集合属性大致有两种: 第一种是单纯的集合属性, 如像 List、Set 或数组等集合属性; 另一种是 Map 结构的集合属性, 每个属性值都有对应的 key 映射。

集合映射的元素大致有如下几种。

- **list:** 用于映射 List 集合属性。
- **set:** 用于映射 Set 集合属性。
- **map:** 用于映射 Map 集合属性。
- **array:** 用于映射数组集合属性。
- **bag:** 用于映射无序集合。
- **idbag:** 用于映射无序集合, 但为集合增加逻辑次序。

下面分别对 List, Set, Map 的集合属性进行讲解。

#### 1. List 集合属性

List 是有序集合, 因此持久化到数据库时也必须增加一列来表示集合元素的次序。看下面的持久化类, 该 Person 类有个集合属性: schools, 该属性对应多个学校。而集合属性只能以接口声明, 因此在下面的代码中, schools 的类型只能是 List, 不能是 ArrayList, 但该集合属性必须使用实现类完成初始化。

```
public class Person implements Serializable
```

```
{  
    //标识属性  
    private int id;  
    //名字属性  
    private String name;  
    //年龄属性  
    private int age;  
    //集合属性，学校  
    private List schools = new ArrayList();  
    //无参数的构造器  
    Person() {}  
    //标识属性 id 的 setter 方法  
    public void setId(int id)  
    {  
        this.id = id;  
    }  
    //标识属性 id 的 getter 方法  
    public int getId()  
    {  
        return id;  
    }  
    //属性 name 的 getter 方法  
    public String getName()  
    {  
        return name;  
    }  
    //属性 name 的 setter 方法  
    public void setName(String name)  
    {  
        this.name = name;  
    }  
    //属性 age 的 setter 方法  
    public void setAge(int age)  
    {  
        this.age = age;  
    }  
    //属性 age 的 getter 方法  
    public int getAge()  
    {  
        return age;  
    }  
    //集合属性 schools 的 getter 方法  
    public List getSchools()  
    {  
        return schools;  
    }  
    //集合属性 schools 的 setter 方法  
    public void setSchools(List schools)  
    {  
        this.schools = schools;  
    }  
}
```

该持久化类的普通属性的映射与前面相同，不同的是增加了集合属性。对本例的 List 集合属性，应该使用 list 元素完成映射，list 元素要求用 list-index 的子元素来映射有序集合的次序列。集合属性的值会存放在另外的表中，不可能与持久化类存储在同一个表内。因此必须以外键关联，用 key 元素来映射该外键列。

下面是该持久化类的映射文件：

```
<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
<!-- 每个 class 元素映射一个持久化类-->
<class name="Person">
    <!-- 定义持久化类的标识属性-->
    <id name="id" column="personid">
        <!-- 定义主键生成器策略-->
        <generator class="identity"/>
    </id>
    <!-- 映射普通属性 name-->
    <property name="name"/>
    <!-- 映射普通属性 age-->
    <property name="age"/>
    <!-- 映射集合属性 schools，指定存放集合属性的表名-->
    <list name="schools" table="school">
        <!-- 集合属性和持久化类的关联外键-->
        <key column="personid" not-null="true"/>
        <!-- 集合属性的次序列-->
        <list-index column="list_order"/>
        <!-- 集合属性的元素-->
        <element type="string" column="school_name"/>
    </list>
</class>
</hibernate-mapping>
```

有了 POJO 及映射文件，该类就可以完成持久化访问，用于完成持久化的主程序代码如下：

```
private void createAndStorePerson()
{
    //开始 Hibernate Session
    Session session = HibernateUtil.currentSession();
    //开始事务
    Transaction tx = session.beginTransaction();
    //创建 Person 实例
    Person yeeku = new Person();
    //设置 Person 实例的 age 属性
    yeeku.setAge(29);
    //设置 name 属性
    yeeku.setName("aaa");
    //创建集合属性
    List l = new ArrayList();
    l.add("小学");
    l.add("中学");
    //设置集合属性
    yeeku.setSchools(l);
    //持久化 Person 实例，将状态保存到数据库
    session.save(yeeku);
    //提交事务
    tx.commit();
    //关闭 Hibernate Session
```

```
HibernateUtil.closeSession();  
}
```

程序运行结束后，数据库将生成两个表：一个用于保存持久化类 Person 的基本属性；另一个表将用于保存集合属性 schools。

## 2. Set 集合属性

Set 集合属性的映射与 List 非常相似，但因为 Set 是无序的，不可重复的集合。因此 set 元素无须使用 index 元素来指定集合元素的次序。

与 List 相同的是，Set 集合同样需要外键列。用于持久化类和集合属性的关联。Set 集合属性声明时，只能使用 Set 接口，不能使用实现类。可将上面示例的 List 集合属性改为 Set 集合属性，此处不再赘述。

映射文件也与 List 集合属性的映射相似，区别在于使用 set 元素时，无须增加 index 列来保存集合元素的次序。下面是 Set 集合属性的映射文件：

```
<?xml version="1.0"?>  
<!-- Hibernate 的映射文件的文件头，包含 DTD 等信息-->  
<!DOCTYPE hibernate-mapping  
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
<!-- Hibernate 映射文件的根元素-->  
<hibernate-mapping package="lee">  
    <!-- 每个 class 元素映射一个持久化类-->  
    <class name="Person">  
        <!-- 映射标识属性-->  
        <id name="id" column="personid">  
            <!-- 指定主键生成器-->  
            <generator class="identity" />  
        </id>  
        <!-- 映射 name 属性-->  
        <property name="name" />  
        <!-- 映射 age 属性-->  
        <property name="age" />  
        <!-- 映射 Set 集合属性，指定集合属性存入 school 表-->  
        <set name="schools" table="school">  
            <!-- 集合属性的外键列-->  
            <key column="personid" not-null="true" />  
            <!-- 用于映射集合中的元素-->  
            <element type="string" column="school_name" not-null="true" />  
        </set>  
    </class>  
</hibernate-mapping>
```

在上面的映射文件中，element 元素用于映射集合属性里的每个元素。该元素有个 not-null 属性，该属性默认为 false，即该列默认可以为空。

对比 List 和 Set 两种集合属性，可以发现 List 集合里的元素有顺序，而 Set 集合里的元素没有顺序。当集合属性在另外的表中存储时，List 集合属性可以用关联持久化类的外键和集合次序列作为联合主键。但 Set 集合没有次序列，则以关联持久化类的外键和元素列作为联合主键，前提是元素列不能为空。

注意：映射 Set 集合属性时，如果 element 元素包括 not-null=“true” 属性，则集合

属性表以关联持久化类的外键和元素列作为联合主键，否则该表没有主键。但 List 集合属性的表总是以外键列和元素次序列作为联合主键。

观察图 4.1 和图 4.2 的表结构，对比两个图的红色标识处，图 4.2 是当 Set 集合属性的 element 不能为空时，该表才有联合主键。否则该表没有主键。



图 4.1 保存 List 集合属性的表结构



图 4.2 保存 Set 集合属性的表结构

### 3. bag 元素映射

bag 元素既可以为 List 集合属性映射，也可以为 Collection 集合属性映射。不管是哪种集合属性，使用 bag 元素都将被映射成无序集合，而集合属性对应的表没有主键。

bag 元素只需要 key 元素来映射关联的外键列，而使用 element 元素来映射集合属性的每个元素。

下面是持久化类使用 bag 元素的映射文件代码：

```
<?xml version="1.0"?>
<!!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Person">
        <!-- id 元素映射标识属性-- >
        <id name="id" column="personid">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射 name 属性-->
        <property name="name"/>
        <!-- 映射 age 属性-->
        <property name="age"/>
        <!-- 映射集合属性-->
        <bag name="schools" table="school">
```

```
<key column="personid" not-null="true"/>
<element type="string" column="school_name"/>
</bag>
</class>
</hibernate-mapping>
```

#### 4. Map 集合属性

Map 集合属性不仅需要映射属性 value, 还需要映射属性 key。映射 Map 集合属性时, 同样需要指定外键列, 同时还必须指定 Map 的 key 列。显然, 系统将以外键列和 key 列作为联合主键。

与所有集合属性类似的是, 集合属性的声明只能使用接口, 看下面的 POJO 类:

```
public class Person implements Serializable
{
    //标识属性
    private int id;
    //name 属性
    private String name;
    //age 属性
    private int age;
    //Map 集合属性, 成绩
    private Map scores = new HashMap();
    //Person 类的默认构造器
    Person() {}
    //标识属性 id 的 setter 方法
    public void setId(int id)
    {
        this.id = id;
    }
    //标识属性 id 的 getter 方法
    public int getId()
    {
        return id;
    }
    //属性 name 的 setter 方法
    public String getName()
    {
        return name;
    }
    //属性 name 的 getter 方法
    public void setName(String name)
    {
        this.name = name;
    }
    //属性 age 的 setter 方法
    public void setAge(int age)
    {
        this.age = age;
    }
    //属性 age 的 getter 方法
    public int getAge()
    {
        return age;
    }
    //Map 集合属性 id 的 getter 方法
    public Map getScores()
    {
```

```

        return scores;
    }
    //Map 集合属性 scores 的 setter 方法
    public void setScores(Map scores)
    {
        this.scores = scores;
    }
}

```

Map 集合属性应使用 map 元素映射，该 map 元素需要 key 和 map-key 两个子元素。其中 key 子元素用于映射外键列，而 map-key 子元素则用于映射 Map 集合的 Key。该持久化类的映射文件如下：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Person">
        <!-- id 元素用于映射标识属性-->
        <id name="id" column="personid">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射 name 属性-->
        <property name="name"/>
        <!-- 映射 age 属性-->
        <property name="age"/>
        <!-- 映射 Map 集合属性-->
        <map name="scores" table="score">
            <!-- 映射外键列-->
            <key column="personid" not-null="true"/>
            <!-- 映射 Map Key-->
            <map-key column="xueke" type="string"/>
            <!-- 映射 Map Value-->
            <element column="fenshu" type="float"/>
        </map>
    </class>
</hibernate-mapping>

```

程序运行结束后，保存集合属性 scores 的表，并以 personid 和 xueke 作为联合主键。

**注意：** map-key 和 element 元素都必须确定 type 属性。

## 5. 集合性能的对比

当系统从数据库中初始化某个持久化类时，集合属性是否随持久化类一起初始化呢？如果集合属性里包含十万，甚至百万的记录，在初始化持久化类之时，要完成所有集合属性的加载，势必将导致性能急剧下降。系统很有可能只需要使用持久化类的某个属性中的部分记录，这样，没有必要一次加载所有的集合属性。

对于集合属性，通常推荐使用延迟加载策略。所谓延迟加载就是当系统需要使用集

合属性时才从数据库装载关联的数据。

Hibernate 对集合属性默认采用延迟加载，在某些特殊的情况下为 set, list, map 等元素设置 lazy=“false”属性来取消延迟加载。

根据前面的讲解，可将集合分成如下两类。

- 有序集合：集合里的元素可以根据 key 或 index 访问。
- 无序集合：集合里的元素只能遍历。

有序集合都拥有一个由<key>和 <index>组成的联合主键，在这种情况下，集合属性的更新是非常高效的——主键已经被有效地索引。因此当 Hibernate 试图更新或删除某行时，可以迅速找到该行数据。

而对无序集合而言，如果集合中元素是组合元素或者大量文本及二进制字段，数据库可能无法有效地对复杂的主键进行索引。即使可以建立索引，性能也非常差。例如 Set 的主键由<key>和其他元素字段构成，或者根本没有主键。

显然，有序集合的属性在增加、删除及修改中拥有较好的性能表现。

在设计良好的 Hibernate Domain Object 中，集合属性通常都会增加 inverse="true"的属性，此时集合端不再控制关联关系。因此，无须考虑其集合的更新性能。

#### 4.4.4 映射引用属性

引用属性的意思是：持久化类的属性既不是基本数据类型，也不是 String 字符串，而是某个引用变量，该引用属性的类型可以是自定义类。看下面 POJO 的源代码：

```
public class Person implements Serializable
{
    //标识属性
    private int id;
    //普通属性 age
    private int age;
    //引用属性 name
    private Name name;
    //无参数的构造器
    Person() {}
    //标识属性 id 的 setter 方法
    private void setId(int id)
    {
        this.id=id;
    }
    //标识属性 id 的 getter 方法
    public int getId()
    {
        return id;
    }
    //age 属性的 setter 方法
    public void setAge(int age)
    {
        this.age=age;
    }
    //age 属性的 getter 方法
    public int getAge()
```

```

    {
        return age;
    }
    //引用属性 name 的 setter 方法
    public void setName(Name name)
    {
        this.name = name;
    }
    //引用属性 name 的 getter 方法
    public Name getName()
    {
        return name;
    }
}

```

此时 Person 的 name 属性既不是基本数据类型，也不是 String，而是一个自定义类：Name。由于数据库的列无法存储 Name 对象，因此无法直接使用 property 映射 name 属性。

为了映射引用属性，Hibernate 提供了 component 元素。每个 component 元素映射一个引用属性，引用属性必须指定该属性的类型。因此 component 元素要求具有 class 属性，该属性用于确定引用属性的类型。

**注意：**由于 Hibernate 使用 component 映射引用属性，因此很多地方将引用属性翻译成组件属性。笔者认为，此处的组件与一般意义上的组件毫不相干，故不采用此种翻译。

一个自定义类通常还包括其他属性，因此还应该为 component 元素增加 property 的子元素来映射引用属性的子属性。下面是持久化类的映射文件：

```

<?xml version="1.0"?>
<!!-- Hibernate 映射文件的文件头，包含 DTD 信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!!-- Hibernate 映射文件的文件头-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Person" table="person">
        <!-- id 属性映射标识属性-->
        <id name="id" column="person_id">
            <!-- 指定主键生成器策略-->
            <generator class="increment"/>
        </id>
        <!-- 映射普通属性 age-->
        <property name="age"/>
        <!-- 映射引用属性 name，引用属性的类型为 Name-->
        <component name="name" class="Name" unique="true">
            <!-- 映射引用属性的 first 属性-->
            <property name="first"/>
            <!-- 映射引用属性的 last 属性-->
            <property name="last"/>
        </component>
    </class>
</hibernate-mapping>

```

映射文件中的 component 还有 unique=“true” 属性，这并不是必需的，而是与具体

的业务逻辑相关联的。

引用属性还有如下两种特殊的情况：

- 集合属性的元素既不是基本数据类型，也不是 String 字符串，而是引用类型。
- 持久化类的主键是引用类型。

下面对这两种情况具体分析。

### 1. 集合引用属性映射

集合除了可以存放 String 字符串外，还可以存放引用类型。事实上，在更多情况下，集合里存放的都是引用类型，看下面 POJO 的源代码：

```
public class Person implements Serializable
{
    //标识属性 id
    private int id;
    //普通属性 name
    private String name;
    //普通属性 age
    private int age;
    //存放引用类型的集合属性
    private List schools = new ArrayList();
    Person() {}
    //标识属性 id 的 setter 和 getter 方法
    public void setId(int id)
    {
        this.id = id;
    }
    public int getId()
    {
        return id;
    }
    //name 属性的 getter 和 setter 方法
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    //age 属性的 setter 和 getter 方法
    public void setAge(int age)
    {
        this.age = age;
    }
    public int getAge()
    {
        return age;
    }
    //List 集合属性的 setter 和 getter 方法
    public List getSchools()
    {
        return schools;
    }
    public void setSchools(List schools)
    {
```

```

        this.schools = schools;
    }
}

```

表面上看，该持久化类与前面的带集合属性的 POJO 并没有太大的区别。区别仅在主程序部分，前面 Person 实例的 schools 属性里存放系列的字符串，而现在的 schools 属性里存放系列的 School 实例。下面是 School 的源代码：

```

public class School implements Serializable
{
    //School 类的两个属性：name 和 address
    private String name;
    private String address;
    public School(){}
    //带两个参数的构造器
    public School(String s1 , String s2)
    {
        this.name = s1;
        this.address = s2;
    }
    //name 属性的 setter 方法
    public void setName(String name) {
        this.name = name;
    }
    //name 属性的 getter 方法
    public void setAddress(String address) {
        this.address = address;
    }
    //name 属性的 getter 方法
    public String getName() {
        return (this.name);
    }
    //name 属性的 getter 方法
    public String getAddress() {
        return (this.address);
    }
}

```

对于有集合属性的 POJO，都需要使用 set, list, bag 等集合元素来映射集合属性。如果集合里的元素是普通字符串，则使用 element 映射集合元素即可。如果集合元素是自定义类，则须使用 composite-element 子元素来映射集合元素。

由于 composite-element 元素映射一个引用类型，因此需要增加 class 元素来确定集合元素的类型，该元素还支持以 property 的子元素来定义引用类型的子属性。

下面是 Person 类的持久化映射文件：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Person">
        <!-- id 属性映射-->

```

```

<id name="id" column="personid">
    <!-- 指定主键生成器策略-->
    <generator class="identity"/>
</id>
<!-- 映射普通属性 name-->
<property name="name"/>
<!-- 映射普通属性 age-->
<property name="age"/>
<!-- 映射 List 集合属性-->
<list name="schools" table="school">
    <!-- 映射关联外键列-->
    <key column="personid" not-null="true"/>
    <!-- List 有序集合, 需要索引列-->
    <list-index column="list_order"/>
    <!-- composite-element 映射集合里的元素, class 属性确定集合里元素的类
型-->
    <composite-element class="School">
        <!-- 每个 property 属性映射集合元素的基本属性-->
        <property name="name"/>
        <property name="address"/>
    </composite-element>
</list>
</class>
</hibernate-mapping>

```

## 2. 引用类型主键的映射

在数据库中建模时, 尽量不要使用复杂的物理主键, 应考虑为数据库增加一列, 作为逻辑主键。表面上看, 增加逻辑主键增加了数据冗余, 但如果从外键关联的角度看, 使用逻辑主键的主从表关联中, 从表只需增加一个外键列。如果使用多列作为联合主键, 则需要在从表中增加多个外键列。如果有多个从表需要增加外键列, 则数据冗余更大。

使用物理主键还会增加数据库维护的复杂度, 因为主从表之间的约束关系隐晦难懂, 难于维护。

如果数据库采用简单的逻辑主键, 则不会出现引用类型主键。但在一些特殊的情况下, 也会出现引用类型主键, 看下面的持久化类:

```

public class Person
{
    //用作持久化类 Person 的标识属性
    private Name name;
    //普通属性 age
    private int age;
    //默认构造器
    Person() {}
    //name 属性的 setter 方法
    public void setName(Name name)
    {
        this.name = name;
    }
    //name 属性的 getter 方法
    public Name getName()
    {
        return name;
    }
    //age 属性的 setter 方法

```

```

public void setAge(int age)
{
    this.age = age;
}
//age 属性的 getter 方法
public int getAge()
{
    return age;
}
}

```

Person 的标识属性不再是基本数据类型，也不是 String 字符串，而是 Name 类型，该类型是用户自定义的类型。

如果持久化类需要使用引用类型作为表示属性时，则该类应该满足如下两个条件：

- 实现 java.io.Serializable 接口。
- 重写 equals() 和 hashCode() 方法，这两个方法的返回值都应该根据数据表中联合主键的列来判断。

下面是 Name 的源代码：

```

//标识属性类，实现 Serializable 接口
public class Name implements Serializable
{
    private String firstName;
    private String lastName;
    public Name(){}
    public Name(String s1 ,String s2)
    {
        this.firstName = s1;
        this.lastName = s2;
    }
    //firstName 的 setter 和 getter 方法
    public void setFirstName(String fisrtName)
    {
        this.firstName = fisrtName;
    }
    public String getFirstName()
    {
        return this.firstName;
    }
    //lastName 的 setter 和 getter 方法
    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }
    public String getLastNmae()
    {
        return this.lastName;
    }
    //重写 hashCode 方法，该方法根据 firstName 和 last Name 的值计算得到
    public int hashCode()
    {
        return firstName.hashCode() + lastName.hashCode();
    }
    //重写 equals 方法，同样也根据 firstName 和 last Name 两个属性来判断
    public boolean equals(Object o)
    {

```

```

        if (o instanceof Name)
        {
            Name p = (Name)o;
            if (p.getFirstName().equals(firstName) &&
                p.getLastName().equals(lastName))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            return false;
        }
    }
}

```

对于 Person 持久化类，其 Name 是标识属性的类，Name 实例可以唯一标识 Person 实例。根据业务需要，能唯一标识 Person 实例的应该是 firstName 和 lastName 两个属性。因此 hashCode 和 equals 方法都应根据 firstName 和 lastName 两个属性来判断。

引用类型主键的映射时，应使用 composite-id 元素，该元素需要 class 属性来确定主键的引用类型，并使用 key-property 元素来确定引用类型包含的基本属性。

下面是 POJO Person 持久化映射文件：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Person" table="person">
        <!-- composite-id 元素用于映射引用类型的标识属性，其中 class 元素确定属性的
        类型-->
        <composite-id name="name" class="Name">
            <key-property 元素确定标识属性类包含的属性>
            <key-property name="firstName"/>
            <key-property name="lastName"/>
        </composite-id>
        <property name="age"/>
    </class>
</hibernate-mapping>

```

建议尽量不要使用这种复杂的标识属性。关于联合主键的映射还有一种策略，直接将多个属性映射成数据库主键。

### 3. 复合主键的映射

改写上面的持久化类 Person，不使用 name 作为 Person 的标识属性，而是直接使用 firstName 和 lastName 作为标识属性。

映射复合主键的持久化类必须满足如下两个条件。

- 实现 `java.io.Serializable` 接口。
- 重写 `equals()` 和 `hashCode()` 方法，两个方法的返回值应该根据数据表中的联合主键来判断。

看下面改写过的 Person 源代码：

```

public class Person implements Serializable
{
    //直接定义 lastName 和 firstName 两个属性
    private String lastname;
    private String firstname;
    //定义 age 属性
    private int age;
    //默认构造器
    Person() {}
    //lastName 的 setter 和 getter 方法
    public void setLastname(String lastname)
    {
        this.lastname = lastname;
    }
    public String getLastname()
    {
        return lastname;
    }
    //firstName 的 setter 和 getter 方法
    public void setFirstname(String firstname)
    {
        this.firstname = firstname;
    }
    public String getFirstname()
    {
        return firstname;
    }
    //age 属性的 setter 和 getter 方法
    public void setAge(int age)
    {
        this.age = age;
    }
    public int getAge()
    {
        return age;
    }
    //持久化类必须重写 hashCode 与 equals 方法
    //equals 方法根据 firstName 和 lastName 两个属性来判断
    public boolean equals(Object o)
    {
        if (o instanceof Person)
        {
            Person p = (Person)o;
            if( p.getLastname().equals(lastname) &&
                p.getFirstname().equals(firstname) )
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}

```

```

        }
        else
        {
            return false;
        }
    }

    //hashCode 方法根据 firstName 和 lastName 两个属性来取得
    public int hashCode()
    {
        int hashCode = lastname.hashCode() + firstname.hashCode();
        return hashCode;
    }
}

```

对于 Person 实例，firstName 和 lastName 的联合能唯一标识该实例。因此 hashCode 方法和 equals 方法都根据这两个属性来判断。

如需映射 firstName 和 lastName 两个标识属性时，同样可使用 composite-id 元素，此时的 composite-id 元素不需要 name 和 class 属性，因为标识属性既没有实现类，也不是一个真实存在的属性。

看下面的映射文件源代码：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Person" table="personcomid">
        <!-- composite-id 用于映射复合主键-->
        <composite-id>
            <!-- key-property 元素映射复合主键的每个元素-->
            <key-property name="firstname"/>
            <key-property name="lastname"/>
        </composite-id>
        <property name="age"/>
    </class>
</hibernate-mapping>

```

由此可看出，这两种映射方式的效果差不多，其映射出来的表结构也基本一致。

## 4.5 Hibernate 的关系映射

关系是关系型数据库最基本的特征，也是客观世界最基本，最抽象的。关系可分为如下两个类。

- 单向关系：只需单向访问关联端。
  - 双向关系：关联的两端可以互相访问。
- 单向关联可分为：
- 单向 1 – 1

- 单向 1 - N
- 单向 N - 1
- 单向 N - N

双向关联可分为：

- 双向 1 - 1
- 双向 1 - N
- 双向 N - N

可以看出在双向关系里没有 N - 1，因为双向关系 1 - N 和 N - 1 是完全相同的。下面依次讲解每个关联的映射方法。

## 4.5.1 单向 N - 1 的关系映射

N - 1 是非常常见的关联关系（如最常见的父子关系），单向的 N - 1 的关联只需从 N 的一端可以访问 1 的一端。比如多个人对应同一个住址，则只需从人的实体端找到对应的地址实体即可，无须关心某个地址的全部住户。

先看如下两个 POJO：

```
public class Person
{
    private int personid;
    private String name;
    private int age;
    private Address address;

    // 此处省略 personid, name, age 三个属性的 setter 及 getter 方法
    /**
     * @return 返回此人对应的地址
     */
    public Address getAddress()
    {
        return address;
    }
    /**
     * @param address 修改人对应的地址
     */
    public void setAddress(Address address)
    {
        this.address = address;
    }
}
```

这是 Person 对应的 POJO，每个 Person 单向地对应一个 Address。但无法从 Address 端来访问 Person。

下面是 Address 的 POJO：

```
public class Address
{
    private int addressid;
    private String addressdetail;
```

```
/*
 * 设置该地址的标识属性
 * @param addressid Address 标识属性
 */
public void setAddressid(int addressid)
{
    this.addressid = addressid;
}
/**
 * 设置该地址的详细地址位置
 * @param addressdetail 地址的详细地址位置
 */
public void setAddressdetail(String addressdetail)
{
    this.addressdetail = addressdetail;
}
/**
 * 返回该地址的标识属性
 * @return 地址的标识属性
 */
public int getAddressid()
{
    return (this.addressid);
}
/**
 * 返回该地址的详细地址位置
 * @return 地址的详细地址位置
 */
public String getAddressdetail()
{
    return (this.addressdetail);
}
}
```

对于 Address 端而言，并不必关心 Person 持久化类，在 Address 的代码中并没有对 Person 的访问。因此 Address 的映射就是基本映射，无须改变。

### 1. 无连接表的 N – 1

Person 端增加了 Address 属性，该属性不是一个普通的引用属性，而是引用另一个持久化类，使用 many-to-one 元素即可映射 N – 1 的持久化属性。

many-to-one 元素的作用类似于 property 元素，用于映射持久化类的某个属性，区别是该元素映射的是关联持久化类。与 property 元素类似的是，many-to-one 元素也必须拥有 name 属性，用于确定该属性的名字，还可以使用 type 属性确定关联实体的类；column 属性确定外键列的列名。

下面是使用无连接表 N – 1 关联的映射文件代码：

```
<?xml version="1.0"?>
<! -- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://ibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
<! -- 下面 class 元素映射的是持久化类 Person-->
<class name="Person">
```

```

<!-- 映射标识属性 personid -->
<id name="personid" >
    <!-- 定义主键生成器 -->
    <generator class="identity"/>
</id>
<property name="name"/>
<property name="age"/>
<!-- 用来映射关联的 PO column 是 Address 在该表中的外键列名 -->
<many-to-one name="address" column="addressId" />
</class>
<!-- 下面持久化类映射 Address -->
<class name="Address">
    <!-- 映射标识属性 -->
    <id name="addressid">
        <generator class="identity"/>
    </id>
    <property name="addressdetail"/>
</class>
</hibernate-mapping>

```

## 2. 有连接表的 N-1

如果需要使用有连接表的 N-1，则需要使用 `join` 元素（通常，`join` 元素用于强制使用连接表）。`join` 元素的 `table` 属性用于确定连接表的表名；使用 `key` 子元素来确定连接表外键，并为 `join` 元素增加 `many-to-one` 子元素，该子元素用于映射关联属性。

该 `many-to-one` 子元素的定义与不使用连接表的 `many-to-one` 几乎相同：同样使用 `name` 属性确定关联属性的属性名，`type` 指定关联类的类型，`column` 属性指定列名。当然，除了 `name` 属性必须指定外，其他都是可选的。

下面是使用有连接表的 N-1 映射文件代码：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://ibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 映射持久化类 Address-->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="addressdetail"/>
    </class>
    <!-- 映射 Person 持久化类-->
    <class name="Person">
        <!-- 映射标识属性-->
        <id name="personid" >
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <property name="name"/>
        <property name="age"/>

```

```

<!-- 使用 join 元素显式确定连接表-->
<join table="join_table" >
    <!-- 映射关键所用的外键列-->
    <key column="personid" />
    <many-to-one name="address" />
</join>
</class>
</hibernate-mapping>

```

对于单向的 N-1，通常推荐使用无连接表关联即可。因为其映射文件配置简单，易于理解并且效果也不错。

## 4.5.2 单向 1-1 的关系映射

单向 1-1 的 POJO 与 N-1 没有丝毫区别。因为 N 的一端，或者 1 的一端都是直接访问关联实体，即增加对关联类属性的 `setter` 和 `getter` 方法。

事实上，单向 1-1 与 N-1 的映射配置也非常相似。只需要将原有的 `many-to-one` 元素增加 `unique=“true”` 属性，用以表示 N 的一端也必须是唯一的，在 N 的一端增加了唯一的约束，即成为单向 1-1。

### 1. 基于外键的单向 1-1

将与无连接表 N-1 关联的 `many-to-one` 增加 `unique=“true”` 属性即可。下面是映射文件代码：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
    <!-- 下面 class 元素映射的是持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性 personid -->
        <id name="personid" >
            <!-- 定义主键生成器 -->
            <generator class="identity" />
        </id>
        <property name="name" />
        <property name="age" />
        <!-- 用来映射关联的 PO column 是 Address 在该表中的外键列名 增加 unique 变成 1-1 -->
        <many-to-one name="address" column="addressId" unique="true"/>
    </class>
    <!-- 下面持久化类映射 Address -->
    <class name="Address">
        <!-- 映射标识属性 -->
        <id name="addressid" >
            <generator class="identity" />
        </id>
        <property name="addressdetail" />
    </class>
</hibernate-mapping>

```

## 2. 有连接表的单向 1-1

虽然这种情况很少见，但 Hibernate 同样允许采用连接表关联 1-1。有连接表的 1-1 同样只需要将连接表的 N-1 的 many-to-one 元素增加 unique=“true” 属性即可。

下面是有连接表的 1-1 映射文件代码：

```
<?xml version="1.0"?>
<!!-- Hibernate 映射文件的文件头-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!!-- 映射持久化类 Address-->
    <class name="Address">
        <!!-- 映射标识属性-->
        <id name="addressid">
            <!!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!!-- 映射普通属性-->
        <property name="addressdetail"/>
    </class>
    <!!-- 映射 Person 持久化类-->
    <class name="Person">
        <!!-- 映射标识属性-->
        <id name="personid" >
            <!!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <property name="name"/>
        <property name="age"/>
        <!!-- 使用 join 元素显式确定连接表-->
        <join table="join_table" >
            <!!-- 映射主键所用的外键列-->
            <key column="personid"/>
            <!!-- 映射 1-1 关联属性，其中 unique="true" 属性确定为 1-1-->
            <many-to-one name="address" unique="true"/>
        </join>
    </class>
</hibernate-mapping>
```

## 3. 基于主键的单向 1-1

1-1 的关联可以基于主键关联，但基于主键关联的持久化类不能拥有自己的主键生成器策略，它的主键由关联类负责生成。另外，增加 one-to-one 元素来映射关联属性，必须为 one-to-one 元素增加 constrained="true" 属性，表明该类的主键由关联类生成。

下面是基于主键关联的单向 1-1 的映射文件代码：

```
<?xml version="1.0"?>
<!!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping >
    <!!-- 下面 class 元素映射的是持久化类 Person-->
```

```
<class name="Person">
    <!-- 映射标识属性 personid -->
    <id name="personid" >
        <!-- 基于主键关联时, 主键生成策略是 foreign, 表明根据关联类生成主键-->
        <generator class="foreign">
            <!-- 关联持久化类的属性名-->
            <param name="property">address</param>
        </generator>
    </id>
    <property name="name" />
    <property name="age" />
    <!-- 用于映射 1-1 关联-->
    <one-to-one name="address" constrained="true"/></class>
<!-- 下面持久化类映射 Address -->
<class name="Address">
    <!-- 映射标识属性 -->
    <id name="addressid" >
        <generator class="identity"/>
    </id>
    <property name="addressdetail" />
</class>
</hibernate-mapping>
```

### 4.5.3 单向 1 – N 的关系映射

与单向 1 – N 关联的 POJO 不同，需要使用集合属性。因为 1 的一端需要访问 N 的一端，而 N 的一端将以集合的形式表现。

通常，不推荐使用单向的 1 – N 关联。对于 1 – N 的父子关联，使用 1 的一端控制关系的性能比使用 N 的一端控制关系的性能低。性能低的原因是使用 1 的一端控制关联关系时，会额外多出 update 语句，并且一旦使用了 1 的一端来控制关系，因为插入数据时无法同时插入外键列，因此外键列无法增加非空约束。

N 的一端是 Address 端，在单向关联中无须访问关联类，其源代码如下：

```
public class Address implements Serializable
{
    //标识属性
    private int addressid;
    private String addressdetail;
    public Address(){}
}
public Address(String addressdetail){
    this.addressdetail = addressdetail;
}
public void setAddressid(int addressid) {
    this.addressid = addressid;
}
public void setAddressdetail(String addressdetail) {
    this.addressdetail = addressdetail;
}
public int getAddressid() {
    return (this.addressid);
}
public String getAddressdetail() {
    return (this.addressdetail);
```

```

    }
}

```

下面是 Person 端，需要增加集合属性，并且集合属性里每个元素都是持久化类，下面是 Person 的源代码：

```

public class Person implements Serializable
{
    private int personid;
    private String name;
    private int age;
    //集合属性，用于关联持久化类
    private Set addresses = new HashSet();
    //标识属性的 setter 方法
    public void setPersonid(int personid) {
        this.personid = personid;
    }
    //name 属性的 setter 方法
    public void setName(String name) {
        this.name = name;
    }
    //age 属性的 setter 方法
    public void setAge(int age) {
        this.age = age;
    }
    //标识属性 id 的 getter 方法
    public int getPersonid() {
        return (this.personid);
    }
    //name 属性的 getter 方法
    public String getName() {
        return (this.name);
    }
    //age 属性的 getter 方法
    public int getAge() {
        return (this.age);
    }
    //集合属性的 getter 方法
    public Set getAddresses(){
        return addresses;
    }
    //集合属性的 getter 方法
    public void setAddresses(Set addresses){
        this.addresses = addresses;
    }
}

```

对于 1-N 的单向关联而言，需要在 1 的一端增加对应的集合映射元素，例如 set, list, bag 等。与映射集合属性类似，必须为 set, list, bag 等集合元素增加 key 子元素，用以映射关联外键列。

与集合属性不同的是：建立 1-N 关联时，集合中的元素使用 one-to-many 来映射，而不是使用 element 子元素。详细映射看下面部分。

### 1. 无连接表的单向 1-N

使用对应的集合元素映射集合属性时，集合属性必须增加 key 子元素，该子元素用

以确定关联的外键列，使用 one-to-many 映射关联属性。

下面是无连接表的单向 1-N 映射文件的代码：

```
<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid">
            <!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="addressdetail"/>
    </class>
    <!-- 映射持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性-->
        <id name="personid" >
            <!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="name"/>
        <property name="age"/>
        <!-- 映射集合属性，关联到持久化类-->
        <set name="addresses">
            <!-- 确定关联的外键列-->
            <list-index column="displayorder"/>
            <!-- 用以映射到关联类属性-->
            <one-to-many class="Address"/>
        </set>
    </class>
</hibernate-mapping>
```

## 2. 有连接表的单向 1-N

有连接表的单向 1-N 非常类似于 N-N 的映射，但集合元素中不使用 one-to-many 元素来映射关联属性，而是使用 many-to-many 元素。但为了保证是 1 的一端，因此增加 unique="true" 属性。下面是有连接表的单向 1-N 关联的映射文件：

```
<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://ibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid">
```

```

<!-- 确定主键生成器策略-->
<generator class="identity"/>
</id>
<!-- 映射普通属性-->
<property name="addressdetail"/>
</class>
<!-- 映射持久化类 Person-->
<class name="Person">
    <!-- 映射标识属性-->
    <id name="personid" >
        <!-- 确定主键生成器策略-->
        <generator class="identity"/>
    </id>
    <!-- 映射普通属性-->
    <property name="name"/>
    <property name="age"/>
    <!-- 映射集合属性，关联到持久化类-->
    <set name="addresses">
        <!-- 确定关联的外键列-->
        <set-index column="displayorder"/>
        <!-- 用以映射到关联类属性，unique="true"表明 1-N -->
        <many-to-many class="Address" unique="true"/>
    </set >
    </class>
</hibernate-mapping>

```

#### 4.5.4 单向 N-N 的关系映射

单向 N-N 的 POJO 与 1-N 的代码完全相同，其控制关系的一端访问的是集合，被关联的持久化实例以集合的形式存在。

N-N 的关联只能使用连接表，与有连接表 1-N 的关联非常相似，只要去掉 many-to-many 元素的 unique="true" 属性即可。

下面是单向 N-N 关联的映射文件：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid" >
            <!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="addressdetail"/>
    </class>
    <!-- 映射持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性-->

```

```

<id name="personid" >
    <!-- 确定主键生成器策略-->
    <generator class="identity"/>
</id>
<!-- 映射普通属性-->
<property name="name"/>
<property name="age"/>
<!-- 映射集合属性，关联到持久化类-->
<set name="addresses">
    <!-- 确定关联的外键列-->
    <list-index column="displayorder"/>
    <!-- 用以映射到关联类属性，unique="true"表明 1 -N-->
    <many-to-many class="Address"/>
</set>
</class>
</hibernate-mapping>

```

## 4.5.5 双向 1 – N 的关系映射

对于 1-N 的关联，Hibernate 推荐使用双向关联，但不用 1 的一端来控制关联关系，而使用 N 的一端来控制关联关系。

双向的 1 – N 与 N – 1 是完全相同的两种情形，两端都需要增加对关联属性的访问，N 的一端直接访问关联类属性；1 的一端增加集合属性的访问。看下面两个 POJO 的源代码：

在 Person 端，增加对集合属性的访问，源代码如下：

```

public class Person implements Serializable
{
    //标识属性
    private int personid;
    private String name;
    private int age;
    //集合属性，用于映射关联类
    private Set addresses = new HashSet();
    //标识属性的 setter 方法
    public void setPersonid(int personid) {
        this.personid = personid;
    }
    //name 属性的 setter 方法
    public void setName(String name) {
        this.name = name;
    }
    //age 属性的 setter 方法
    public void setAge(int age) {
        this.age = age;
    }
    //标识属性的 getter 方法
    public int getPersonid() {
        return (this.personid);
    }
    //name 属性的 getter 方法
    public String getName() {
        return (this.name);
    }
    //age 属性的 getter 方法

```

```
public int getAge() {
    return (this.age);
}
//关联的集合属性的 getter 方法
public Set getAddresses() {
    return addresses;
}
//关联的集合属性的 setter 方法
public void setAddresses(Set addresses) {
    this.addresses = addresses;
}
}
```

Address 端则增加了对 Person 属性的访问，源代码如下：

```
public class Address implements Serializable
{
    //标识属性
    private int addressid;
    private String addressdetail;
    //关联属性
    private Person person;
    public Address(){
    }
    //有参数的构造器
    public Address(String addressdetail){
        this.addressdetail = addressdetail;
    }
    //标识属性 id 的 setter 方法
    public void setAddressid(int addressid) {
        this.addressid = addressid;
    }
    //addressdetail 属性的 setter 方法
    public void setAddressdetail(String addressdetail) {
        this.addressdetail = addressdetail;
    }
    //标识属性 id 的 getter 方法
    public int getAddressid() {
        return (this.addressid);
    }
    //addressdetail 属性的 getter 方法
    public String getAddressdetail() {
        return (this.addressdetail);
    }
    //关联属性 person 的 getter 方法
    public Person getPerson()
    {
        return person;
    }
    //关联属性 person 的 setter 方法
    public void setPerson(Person person)
    {
        this.person = person;
    }
}
```

## 1. 无连接表的双向 1-N 关联

无连接表的双向 1-N 关联，需要同时修改两个持久化类的配置文件。N 的一端需要

增加 many-to-one 元素来映射关联属性；而 1 的一端需要使用集合元素来映射关联属性。在集合元素里不仅需要增加 key 元素，还需要使用 one-to-many 元素来映射关联属性。

下面是 1-N 关联的映射文件代码：

```
<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid">
            <!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="addressdetail"/>
        <!-- 映射关联属性，column 属性指定外键列名-->
        <many-to-one name="person" column="personId"/>
    </class>
    <!-- 映射持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性-->
        <id name="personid" >
            <!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="name"/>
        <property name="age"/>
        <!-- 映射集合属性，关联到持久化类-->
        <set name="addresses">
            <!-- column 用于指定外键列名-->
            <key column="personId"/>
            <!-- 映射关联类-->
            <one-to-many class="Address"/>
        </set>
    </class>
</hibernate-mapping>
```

**注意：**在上面的配置文件中，两个持久化类的配置文件都需要指定外键列的列名，此时不可以省略。因为不使用连接表的 1-N 关联的外键，所以外键只保存在 N 一端的表中。如果两边指定的外键列名不同，将导致关联映射出错。如果不指定外键列的列名，该列名由系统自动生成，而系统很难保证自动生成的两个列名相同。

## 2. 有连接表的双向 1-N 关联

有连接表的 1-N 双向关联类似于 N-N 关联。1 的一端应使用集合元素映射，然后在集合元素里增加 many-to-many 的子元素，该子元素映射到关联类。另外，应该为 many-to-many 元素增加 unique="true" 属性。N 的一端则使用 join 元素来强制增加连接表。

**注意：**两边确定连接表的 table 属性值应该相同，而且 table 属性不能省略。否则关联映射将出错。

join 元素不仅使用 key 子元素来确定外键列，还需要增加 many-to-one 元素映射到关联属性。

下面是双向 1-N 关联的配置文件：

```

<?xml version="1.0"?>
<!!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!!-- 每个 class 元素映射一个持久化类-->
    <class name="Address">
        <!!-- 映射标识属性-->
        <id name="addressId">
            <!!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!!-- 映射普通属性-->
        <property name="addressDetail"/>
        <!!-- 显式使用 join 元素确定连接表 -->
        <join table="PersonAddress" inverse="true" optional="true">
            <!!-- key 映射外键列-->
            <key column="addressId"/>
            <!!-- many-to-one 元素映射关联属性-->
            <many-to-one name="person"
                column="personId"
                not-null="true"/>
        </join>
    </class>
    <!!-- 映射持久化类 Person-->
    <class name="Person">
        <!!-- 映射标识属性-->
        <id name="personId" >
            <!!-- 确定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!!-- 映射普通属性-->
        <property name="name"/>
        <property name="age"/>
        <!!-- 映射集合属性，关联到持久化类-->
        <set name="addresses" table="PersonAddress">
            <!!-- key 映射外键列-->
            <key column="personId"/>
            <!!-- many-to-one 元素映射关联属性，为保证为 1，增加 unique="true" 属性-->
            <many-to-many column="addressId"
                unique="true"
                class="Address"/>
        </set>
    </class>
</hibernate-mapping>

```

注意：映射的文件的两边都指定了两个外键列，一个是持久化类实例在连接表中的外键列名，另一个是关联属性在连接表中的外键列名。一定要保证两边映射的外键列名对应相同。

## 4.5.6 双向 N-N 关联

双向 N-N 关联需要两端都使用集合属性，两端都增加对集合属性的访问。双向 N-N 关联也必须使用连接表。看下面 POJO 的源代码。

在 Person 端增加了 set 属性，用以存放关联的 address 属性：

```
public class Person
{
    private int personid;
    private String name;
    private int age;
    private Set addresses = new HashSet();
    //标识属性 id 的 setter 方法
    public void setPersonid(int personid) {
        this.personid = personid;
    }
    //name 属性的 setter 方法
    public void setName(String name) {
        this.name = name;
    }
    //age 属性的 setter 方法
    public void setAge(int age) {
        this.age = age;
    }
    //标识属性 id 的 getter 方法
    public int getPersonid() {
        return (this.personid);
    }
    //name 属性的 getter 方法
    public String getName() {
        return (this.name);
    }
    //age 属性的 getter 方法
    public int getAge() {
        return (this.age);
    }
    //关联实体的集合属性的 getter 方法
    public Set getAddresses(){
        return addresses;
    }
    //关联实体的集合属性的 setter 方法
    public void setAddresses(Set addresses){
        this.addresses = addresses;
    }
}
```

在 Address 端增加了 set 属性，用以存放关联的 person 属性。

```
public class Address
```

```

{
    //标识属性
    private int addressid;
    private String addressdetail;
    //用以存放关联实体的集合属性
    private Set persons = new HashSet();
    public Address(){}
    public Address(String addressdetail){
        this.addressdetail = addressdetail;
    }
    //标识属性的 setter 方法
    public void setAddressid(int addressid) {
        this.addressid = addressid;
    }
    //addressdetail 属性的 setter 方法
    public void setAddressdetail(String addressdetail) {
        this.addressdetail = addressdetail;
    }
    //标识属性 id 的 getter 方法
    public int getAddressid() {
        return (this.addressid);
    }
    //属性 addressdetail 的 getter 方法
    public String getAddressdetail() {
        return (this.addressdetail);
    }
    //关联实体集合属性的 getter 方法
    public Set getPersons(){
        return persons;
    }
    //关联实体集合属性的 setter 方法
    public void setPersons(Set persons){
        this.persons = persons;
    }
}

```

双向 N – N 的关联映射需要在两边增加集合元素，用于映射集合属性。集合属性应增加 key 子元素用以映射外键列；集合元素里还应增加 many-to-many 子元素来映射关联实体类。

下面是双向 N – N 关联的映射文件代码：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 每个 class 元素映射一个持久化类-->
    <class name="Address">
        <!-- id 元素映射标识属性-->
        <id name="addressid">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性-->
        <property name="addressdetail"/>

```

```

<!-- 映射集合属性，使用指定连接表-->
<set name="persons" inverse="true" table="jointable">
    <!-- 指定本持久化类在连接表中的外键列名。-->
    <key column="addressId"/>
    <!-- 映射多对多关联类-->
    <many-to-many column="personId" class="Person"/>
</set>
</class>
<!-- 映射持久化类 Person -->
<class name="Person">
    <!-- 映射标识属性-->
    <id name="personid" >
        <!-- 定义主键生成器策略-->
        <generator class="identity"/>
    </id>
    <property name="name"/>
    <property name="age"/>
    <!-- 映射关联实体的集合属性，使用指定连接表-->
    <set name="addresses" table="jointable">
        <!-- 映射本持久化类在连接表中的外键列名-->
        <key column="personId"/>
        <!-- 映射多对多关联类-->
        <many-to-many column="addressId" class="Address"/>
    </set>
</class>
</hibernate-mapping>

```

**注意：**在双向 N - N 关联的两边都需指定连接表的表名及外键列的列名。两个集合元素 set 的 table 元素的值必须指定，而且必须相同。set 元素的两个子元素：key 和 many-to-many 都必须指定 column 属性，其中，key 和 many-to-many 分别指定本持久化类和关联类在连接表中的外键列名，因此两边的 key 与 many-to-many 的 column 属性交叉相同。也就是说，一边的 set 元素的 key 的 column 值为 a，many-to-many 的 column 为 b；则另一边的 set 元素的 key 的 column 值为 b，many-to-many 的 column 值为 a。

#### 4.5.7 双向 1 - 1 关联

前面介绍过，单向的 1-1 关联有三种映射策略：基于主键、基于外键和使用连接表。双向的 1 - 1 关联同样有这三种映射策略。

双向的 1 - 1 关联需要修改 POJO 类，让两边都增加对关联类的访问。看下面 Person 和 Address 的源代码。

在 Person 类中，包含对 Address 属性的 setter 和 getter 方法：

```

public class Person
{
    //基本属性
    private int personid;
    private String name;
    private int age;
    //关联类属性
    private Address address;
    //标识属性 personid 的 setter 方法

```

```

public void setPersonid(int personid) {
    this.personid = personid;
}
//name 属性的 setter 方法
public void setName(String name) {
    this.name = name;
}
//age 属性的 setter 方法
public void setAge(int age) {
    this.age = age;
}
//标识属性 personid 的 getter 方法
public int getPersonid() {
    return (this.personid);
}
//name 属性的 getter 方法
public String getName() {
    return (this.name);
}
//age 属性的 getter 方法
public int getAge() {
    return (this.age);
}
//关联属性 address 的 getter 方法
public Address getAddress(){
    return address;
}
//关联属性 address 的 setter 方法
public void setAddress(Address address){
    this.address = address;
}
}

```

在 Address 类中，同样也包括对关联类 Person 的 setter 和 getter 方法：

```

public class Address implements Serializable
{
    //标识属性 addressid
    private int addressid;
    private String addressdetail;
    private Person person;
    //无参数的构造器
    public Address(){}
    //有参数的构造器
    public Address(String addressdetail){
        this.addressdetail = addressdetail;
    }
    //标识属性 addressid 的 setter 方法
    public void setAddressid(int addressid) {
        this.addressid = addressid;
    }
    //addressdetail 属性的 setter 方法
    public void setAddressdetail(String addressdetail) {
        this.addressdetail = addressdetail;
    }
    //标识属性 addressid 的 getter 方法
    public int getAddressid() {
        return (this.addressid);
    }
}

```

```

//addressdetail 属性的 getter 方法
public String getAddressdetail() {
    return (this.addressdetail);
}
//关联类 Person 的 getter 方法
public Person getPerson() {
    return person;
}
//关联类 Person 的 setter 方法
public void setPerson(Person person) {
    this.person = person;
}
}

```

下面分别介绍这三种映射策略。

### 1. 基于外键的双向 1 – 1 关联

对于基于外键的 1 – 1 关联，其外键可以存放在任意一边，在需要存放外键的一端，增加 many-to-one 元素。正如前面介绍的，为 many-to-one 元素增加 unique="true" 属性来表示为 1 – 1 的关联；并用 name 属性来指定关联属性的属性名。

若另一端需要使用 one-to-one 元素，则该元素需要使用 name 属性指定关联的属性名。为了让系统懂得不再为本表增加一列，因此使用外键关联，用 property-ref 属性来引用关联类的自身属性。

看下面的映射文件：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 映射持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性-->
        <id name="personid" >
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <property name="name"/>
        <property name="age"/>
        <!-- one-to-one 元素映射关联属性，外键列在对方的表内
            person-ref 指向关联类的本身属性。即在 address 属性所属的 Address 类
            内，必须有 person 属性的 setter 和 getter 方法
        -->
        <one-to-one name="address" property-ref="person"/>
    </class>
    <!-- 映射持久化类 Address -->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid" >
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <property name="addressdetail"/>
    
```

```

<!-- many-to-one 元素映射关联属性，unique="true"确定为 1 -1。-->
<many-to-one name="person" unique="true" />
</class>
</hibernate-mapping>

```

**注意：**上面的映射策略可以互换，即让 Person 端存放外键，使用 many-to-one 元素映射关联属性；但 Address 端则必须使用 one-to-one 元素映射，但不可两边都使用相同的元素来映射关联属性。

## 2. 基于主键的双向 1 - 1 关联

基于主键的映射策略，指一端的主键生成器使用 foreign 策略，表明根据对方的主键来生成自己的主键，自己并不能独立生成主键。

当然，任意一边都可以采用 foreign 主键生成器策略，表明根据对方主键生成自己的主键。采用 foreign 主键生成器策略的一端增加 one-to-one 元素映射关联属性，其 one-to-one 属性还应增加 constrained="true" 属性；另一端增加 one-to-one 元素映射关联属性即可。

**注意：**使用 foreign 主键生成器的一端，需额外使用 constrained="true" 属性。

看下面的映射文件：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
    <!-- 映射持久化类 Person-->
    <class name="Person">
        <!-- 映射标识属性-->
        <id name="personid" >
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <property name="name" />
        <property name="age" />
        <!-- one-to-one 元素映射关联属性 -->
        <one-to-one name="address" />
    </class>
    <!-- 映射持久化类 Address -->
    <class name="Address">
        <!-- 映射标识属性-->
        <id name="addressid" >
            <!-- 指定 foreign 主键生成器策略-->
            <generator class="foreign">
                <!-- 指定根据主键生成的关联属性-->
                <param name="property">person</param>
            </generator>
        </id>
        <property name="addressdetail" />
        <!-- 用于映射关联属性，增加 constrained="true" 表明主键
            根据关联属性生成 -->
        <one-to-one name="person" constrained="true"/>
    </class>

```

```
</hibernate-mapping>
```

### 3. 有连接表的双向 1 – 1 关联

采用连接表的双向 1 – 1 关联是相当罕见的情形，其映射相当复杂，数据模型也非常烦琐。通常不推荐使用这种策略。

双向 1 – 1 关联的两边都需要使用 `join` 元素来显式指定连接表，`join` 元素的 `table` 属性用于确定连接表的表名，因此两边的 `join` 元素的 `table` 属性值应该相同。除了在两边都增加 `key` 元素映射连接表中的外键列外，还需增加 `many-to-one` 元素映射关联属性，并为两个 `many-to-one` 元素增加 `unique="true"` 属性表明为 1 – 1 关联。

下面是完整的映射文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="lee">
  <!-- 映射持久化类 Person-->
  <class name="Person">
    <!-- 映射标识属性-->
    <id name="personid" >
      <!-- 指定主键生成器策略-->
      <generator class="identity"/>
    </id>
    <property name="name"/>
    <property name="age"/>
    <!-- 显式使用 join 元素指定连接表关联-->
    <join table="PersonAddress" optional="true">
      <!-- key 元素映射外键列的列名-->
      <key column="personId" unique="true"/>
      <!-- 映射关联属性-->
      <many-to-one name="address" column="addressId"
        not-null="true" unique="true"/>
    </join>
  </class>
  <!-- 映射持久化类 Address -->
  <class name="Address">
    <!-- 映射标识属性-->
    <id name="addressid" >
      <!-- 指定主键生成器策略-->
      <generator class="identity"/>
    </id>
    <property name="addressdetail"/>
    <!-- 显式使用 join 元素指定连接表关联-->
    <join table="PersonAddress" inverse="true" optional="true">
      <!-- key 元素映射外键列的列名-->
      <key column="addressId" unique="true"/>
      <!-- 映射关联属性-->
      <many-to-one name="person" column="personId"
        not-null="true" unique="true"/>
    </join>
  </class>
</hibernate-mapping>
```

注意：带连接表的双向 1-1 关联，两边都须指定连接表的表名及外键列的列名。如 set 的 table 元素的值必须指定，而且必须相同。set 元素的两个子元素：key 和 many-to-many 都必须指定 column 属性。其中，key 和 many-to-many 分别是指定本持久化类和关联类在连接表中的外键列名，因此两边的 key 与 many-to-many 的 column 属性交叉相同。也就是说，一边的 set 元素 key 的 column 值为 a，many-to-many 的 column 值为 b；则另一边的 set 元素 key 的 column 值为 b，many-to-many 的 column 值为 a。

## 4.6 Hibernate 查询体系

Hibernate 提供了异常强大的查询体系，有多种查询方式可以供我们使用。如 Hibernate 的 HQL 查询或者使用条件查询，甚至可以使用原生的 SQL 查询语句等。另外还提供了一种数据过滤功能，这些都用于筛选目标数据。

下面分别介绍 Hibernate 的四种数据筛选方法

### 4.6.1 HQL 查询

HQL 是 Hibernate Query Language 的缩写，HQL 的语法很像 SQL 的语法，但 HQL 是一种面向对象的查询语言。因此，SQL 的操作对象是数据表和列等数据对象；而 HQL 的操作对象是类、实例和属性等。

HQL 是完全面向对象的查询语言，因此可以支持继承和多态等特征。

HQL 查询依赖于 Query 类，每个 Query 实例对应一个查询对象，使用 HQL 查询按如下步骤进行：

- (1) 获取 Hibernate Session 对象；
- (2) 编写 HQL 语句；
- (3) 以 HQL 语句作为参数，调用 Session 的 createQuery 方法创建查询对象；
- (4) 如果 HQL 语句包含参数，调用 Query 的 setXxx 方法为参数赋值；
- (5) 调用 Query 对象的 list 等方法遍历查询结果。

看下面的查询示例：

```
public class HqlQuery
{
    public static void main(String[] args) throws Exception
    {
        HqlQuery mgr = new HqlQuery();
        //调用查询方法
        mgr.findPersons();
        //调用第二个查询方法
        mgr.findPersonsByHappenDate();
        HibernateUtil.sessionFactory.close();
    }
    //第一个查询方法
    private void findPersons()
```

```
{  
    //获得 Hibernate Session  
    Session sess = HibernateUtil.currentSession();  
    //开始事务  
    Transaction tx = sess.beginTransaction();  
    //以 HQL 语句创建 Query 对象  
    //执行 setString 方法为 HQL 语句的参数赋值  
    //Query 调用 list 方法访问查询的全部实例  
    List pl = sess.createQuery("from Person p where p.myEvents.title = :  
        eventTitle")  
        .setString("eventTitle", "很普通事情")  
        .list();  
    //遍历查询的全部结果  
    for (Iterator pit = pl.iterator() ; pit.hasNext(); )  
    {  
        Person p = ( Person )pit.next();  
        System.out.println(p.getName());  
    }  
    //提交事务  
    tx.commit();  
    HibernateUtil.closeSession();  
}  
//第二个查询方法  
private void findPersonsByHappenDate() throws Exception  
{  
    //获得 Hibernate Session 对象  
    Session sess = HibernateUtil.currentSession();  
    Transaction tx = sess.beginTransaction();  
    //解析出 Date 对象  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
    Date start = sdf.parse("2005-01-01");  
    System.out.println("系统开始通过日期查找人" + start);  
    //通过 Session 的 createQuery 方法创建 Query 对象  
    //设置参数  
    //返回结果集  
    List pl = sess.createQuery(  
        "from Person p where p.myEvents.happenDate between :firstDate  
        and :endDate")  
        .setDate("firstDate", start)  
        .setDate("endDate", new Date())  
        .list();  
    //遍历结果集  
    for (Iterator pit = pl.iterator() ; pit.hasNext(); )  
    {  
        Person p = ( Person )pit.next();  
        System.out.println(p.getName());  
    }  
    tx.commit();  
    HibernateUtil.closeSession();  
}
```

通过上面的示例程序，可看出其查询步骤基本相似，但 Query 对象可以连续多次设置参数，这得益于 Hibernate Query 的设计。

通常的 setXxx 方法返回值都是 void，但 Hibernate Query 的 setXxx 方法返回值是 Query 本身。因此，程序通过 Session 创建 Query 后，多次直接调用 setXxx 方法为 HQL

语句的参数赋值，再直接调用 list 方法返回查询到的全部结果。

Query 还包含两个方法。

- setFirstResult(int firstResult): 设置返回的结果集从第几条记录开始。
- setMaxResults(int maxResults): 设置本次查询返回的结果数。

这两个方法用于实现 Hibernate 分页。

下面简单介绍 HQL 语句的语法。

HQL 语句本身不区分大小写，也就是说：HQL 语句的关键字及函数都不区分大小写。但 HQL 语句中所使用的包名、类名、实例名及属性名都区分大小写。

## 1. from 子句

from 是最简单也是最基本的 HQL 语句，from 关键字后紧跟持久化类的类名。例如：

```
from Person
```

表明从 Person 持久化类中选出全部的实例。

大部分时候，推荐为该 Person 的每个实例另起别名。例如：

```
from Person as p
```

在上面的 HQL 语句中，Person 持久化类中的实例的别名为 p，既然 p 是实例名，因此也应该遵守 Java 的命名规则：第一个单词的首字母小写，后面每个单词的首字母大写。

命名别名时，as 关键字是可选的，但为了增加可读性，建议保留。from 后还可同时出现多个持久化类，此时将产生一个笛卡儿积或跨表的连接。

## 2. select 子句

select 字句用于确定选择出的属性，当然 select 选择的属性必须是 from 后持久化类包含的属性，例如：

```
select p.name from Person as p
```

select 可以选择任意属性，不仅可以选择持久化类的直接属性，还可以选择引用属性包含的属性，例如：

```
select p.name.firstName from Person as p
```

select 也支持将选择出的属性存入一个 List 对象中，例如：

```
select new list(p.name, p.address) from Person as p
```

另外，select 甚至可以将选择出的属性直接封装成对象，例如：

```
select new ClassTest(p.name, p.address) from Person as p
```

前提是 ClassTest 支持 p.name 及 p.address 的构造器。假如 p.name 的数据类型是 String，p.address 的数据类型是 String，则 ClassTest 必须有如下的构造器：

```
ClassTest(String s1, String s2)
```

此外，select 还支持给选中的表达式命名别名，例如：

```
select p.name as personName from Person as p
```

这种用法与 new map 结合使用更普遍。例如：

```
select new map(p.name as personName) from Person as p
```

在这种情形下，选择出的是 Map 结构，以 personName 为 key，将实际选出的值作为 value。

### 3. 聚集函数

HQL 也支持在选出的属性上使用聚集函数。HQL 支持的聚集函数与 SQL 完全相同，有如下 5 种。

- avg: 计算属性平均值。
- count: 统计选择对象的数量。
- max: 统计属性值的最大值。
- min: 统计属性值的最小值。
- sum: 计算属性值的总和。

例如：

```
select count(*) from Person  
select max(p.age) from Person as p
```

其中，select 子句还支持字符串连接符、算数运算符及 SQL 函数。例如：

```
select p.name || " " || p.address from Person as p
```

此外，select 子句也支持使用 distinct 和 all 关键字，此时的效果与 SQL 中的效果完全相同。

### 4. 多态查询

HQL 语句被设计成能理解多态查询，其 from 后跟的持久化类名不仅会查询出该持久化类的全部实例，还会查询出该类中子类的全部实例。

如下面的查询语句：

```
from Person as p
```

该查询语句不仅会查询出 Person 的全部实例，还会查询出 Person 的子类。如 Teacher 的全部实例，前提是 Person 和 Teacher 完成了正确继承映射。

HQL 支持在 from 子句中指定任何 Java 类或接口，查询并返回继承了该类的持久化子类的实例或返回实现该接口的持久化类的实例。下面的查询语句将返回所有的被持久化的对象：

```
from java.lang.Object o
```

如果 Named 接口有多个持久化实现类，下面语句将返回这些持久化类的全部实例：

```
from Named as n
```

注意：后面的两个查询将需要多个 SQL SELECT 语句，因此无法使用 order by 子句

对结果集排序，从而不允许对这些查询结果使用 `Query.scroll()` 方法。

## 5. where 子句

`where` 子句用于筛选选中的结果，以缩小选择的范围。如果没有为持久化实例另起别名，可以直接使用属性名引用属性。

如下面的 HQL 语句：

```
from Person where name like "tom%"
```

下面 HQL 语句与上面的语句效果相同：

```
from Person as p where p.name like "tom%"
```

在后面的 HQL 语句中，如果为持久化实例另起了别名，则应该使用完整的属性名。两个 HQL 语句都可返回 `name` 属性以 `tom` 开头的实例。

复合属性表达式加强了 `where` 子句的功能，例如下面的 HQL 语句：

```
from Cat cat where cat.mate.name like "kit%"
```

该查询将被翻译成为一个含有内连接的 SQL 查询，翻译后的 SQL 语句如下：

```
select * from cat_table as table1 cat_table as table2 where table1.mate =  
table2.id and table1.name like "kit%"
```

再看下面的 HQL 查询语句：

```
from Foo foo where foo.bar.baz.customer.address.city like "guangzhou%"
```

翻译成 SQL 查询语句后，将变成一个四表连接的查询。

“=” 运算符不仅可以被用来比较属性的值，也可以用来比较实例：

```
from Cat cat, Cat rival where cat.mate = rival.mate  
select cat, mate  
from Cat cat, Cat mate  
where cat.mate = mate
```

特殊属性（小写）`id` 可以用来表示一个对象的标识符（你也可以使用该对象的属性名）。

```
from Cat as cat where cat.id = 123  
from Cat as cat where cat.mate.id = 69
```

第二个查询是一个内连接查询，但在 HQL 查询语句下，无须体会多表连接，而完全使用面向对象的方式查询。

另外，`id` 也可代表引用标识符。如 `Person` 类有一个引用标识符，它由 `country` 属性与 `medicareNumber` 两个属性组成。下面的 HQL 语句有效：

```
from Person as person  
where person.id.country = 'AU'  
      and person.id.medicareNumber = 123456  
from Account as account  
where account.owner.id.country = 'AU'
```

```
and account.owner.id.medicareNumber = 123456
```

第二个查询是一个多表连接查询，跨越两个表：Person 和 Account。但此处感受不到多表连接查询的效果。

在进行多态持久化的情况下，class 关键字用来存取一个实例的鉴别值（discriminator value）。嵌入 where 子句中的 Java 类名将被作为该类的鉴别值。例如：

```
from Cat cat where cat.class = DomesticCat
```

其中，where 子句中的属性表达式必须以基本类型或者 java.lang.String 结尾，不要使用引用类型属性结尾。如 Account 有 Person 属性；而 Person 有 Name 属性；Name 有 firstName 属性。

看下面的情形：

```
from Account as a where a.person.name.firstName like "dd%" //正确  
from Account as a where a.person.name like "dd%" //错误
```

## 6. 表达式

HQL 的功能非常丰富，其 where 子句后支持的运算符异常丰富，不仅包括 SQL 的运算符，也包括 EJB-QL 的运算符等。

where 子句中允许使用大部分 SQL 支持的表达式。

- 数学运算符 +, -, \*, / 等。
- 二进制比较运算符 =, >=, <=, <>, !=, like 等。
- 逻辑运算符 and, or, not 等。
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of 等。
- 简单的 case, case ... when ... then ... else ... end 和 case, case when ... then ... else ... end 等。
- 字符串连接符 value1 || value2 或者使用字符串连接函数 concat(value1, value2)。
- 时间操作函数：current\_date(), current\_time(), current\_timestamp(), second(), minute(), hour(), day(), month(), year() 等。
- HQL 还支持 EJB-QL3.0 所支持的函数或操作：substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit\_length(), coalesce() 和 nullif() 等。
- 还支持数据库的类型转换函数，如 cast(... as ...)，第二个参数是 Hibernate 的类型名，或者 extract(... from ...)，前提是底层数据库支持 ANSI cast() 和 extract()。
- 如果底层数据库支持单行函数：sign(), trunc(), rtrim(), sin()。则 HQL 语句也完全可以支持。
- HQL 语句支持使用“?”作为参数占位符，这与 JDBC 的参数占位符一致，也可使用命名参数占位符号，方法是在参数名前加冒号“：“，如 :start\_date, :x1 等。
- 也可在 where 子句中使用 SQL 常量，如'foo', 69, '1970-01-01 10:00:01.0' 等。
- 还可以在 HQL 语句中使用 Java public static final 类型的常量，如 eg.Color.TABBY。

除此之外，`where` 子句还支持如下的特殊关键字用法。

- `in` 与 `between...and` 可按如下方法使用：

```
from DomesticCat cat where cat.name between 'A' and 'B'  
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

- 当然，也支持 `not in` 和 `not between...and` 的使用，例如：

```
from DomesticCat cat where cat.name not between 'A' and 'B'  
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

- 子句 `is null` 与 `is not null` 可以被用来测试空值，例如：

```
from DomesticCat cat where cat.name is null;  
from Person as p where p.address is not null;
```

如果在 Hibernate 配置文件中使用，必须进行如下声明：

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

上面的声明表明：将 HQL 转换 SQL 语句时，应使用字符 1 和 0 来取代关键字 `true` 和 `false`。然后才可以在表达式中使用布尔表达式，例如：

```
from Cat cat where cat.alive = true
```

- `size` 关键字用于返回一个集合的大小，例如：

```
from Cat cat where cat.kittens.size > 0  
from Cat cat where size(cat.kittens) > 0
```

- 对于有序集合，还可使用 `minindex` 与 `maxindex` 函数代表最小与最大的索引序数。同理，可以使用 `minelement` 与 `mxelement` 函数代表集合中最小与最大的元素。例如：

```
from Calendar cal where mxelement(cal.holidays) > current date  
from Order order where maxindex(order.items) > 100  
from Order order where minelement(order.items) > 10000
```

- 可以使用 SQL 函数如 `any`, `some`, `all`, `exists`, `in` 等来操作集合里的元素，例如：

```
//操作集合元素  
select mother from Cat as mother, Cat as kit  
where kit in elements(foo.kittens)  
//p 的 name 属性等于集合中某个元素的 name 属性  
select p from NameList list, Person p  
where p.name = some elements(list.names)  
//操作集合元素  
from Cat cat where exists elements(cat.kittens)  
from Player p where 3 > all elements(p.scores)  
from Show show where 'fizard' in indices(show.acts)
```

注意，在这些结构变量中：`size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `mxelement` 等，只能在 `where` 子句中使用。

- 在 `where` 子句中，有序集合的元素(`arrays`, `lists`, `maps`)可以通过 [ ] 运算符访问。

例如：

```
//items 是有序集合属性， items[0] 代表第一个元素  
from Order order where order.items[0].id = 1234  
//holidays 是 map 集合属性， holidays[national day] 是代表其中一个元素  
select person from Person person, Calendar calendar  
where calendar.holidays['national day'] = person.birthDay  
    and person.nationality.calendar = calendar  
//下面同时使用 list 集合和 map 集合属性  
select item from Item item, Order order  
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11  
select item from Item item, Order order  
where order.items[ maxIndex(order.items) ] = item and order.id = 11
```

在[ ]中的表达式甚至可以是一个算数表达式，例如：

```
select item from Item item, Order order  
where order.items[ size(order.items) - 1 ] = item
```

借助于 HQL，可以大大简化选择语句的书写，提高查询语句的可读性，看下面的 HQL 语句：

```
select cust  
from Product prod,  
      Store store  
     inner join store.customers cust  
where prod.name = 'widget'  
    and store.location.name in ( 'Melbourne', 'Sydney' )  
    and prod = all elements(cust.currentOrder.lineItems)
```

如果翻译成 SQL 语句，将变成如下形式：

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order  
FROM customers cust,  
      stores store,  
      locations loc,  
      store_customers sc,  
      product prod  
WHERE prod.name = 'widget'  
    AND store.loc_id = loc.id  
    AND loc.name IN ( 'Melbourne', 'Sydney' )  
    AND sc.store_id = store.id  
    AND sc.cust_id = cust.id  
    AND prod.id = ALL(  
        SELECT item.prod_id  
        FROM line_items item, orders o  
        WHERE item.order_id = o.id  
          AND cust.current_order = o.id  
    )
```

## 7. order by 子句

查询返回的列表（list），可以根据类或引用属性的任何属性来进行排序，例如：

```
from Person as p  
order by p.name, p.age
```

还可使用 asc 或 desc 关键字指定升序或降序的排序规则，例如：

```
from Person as p
order by p.name asc , p.age desc
```

如果没有指定排序规则时，默认采用升序规则。与是否使用 `asc` 关键字没有区别，加 `asc` 是升序排序，不加 `asc` 也是升序排序。

## 8. group by 子句

利用返回聚集值的查询，可以对持久化类或引用属性的属性进行分组，分组时可使用 `group by` 子句。看下面的 HQL 查询语句：

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

类似于 SQL 的规则，出现在 `select` 后的属性要么出现在聚集函数中，要么出现在 `group by` 的属性列表中。看下面示例：

```
//select 后出现的 id 处出现在 group by 之后，而 name 属性则出现在聚集函数中
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

`having` 子句用于对分组进行过滤，例如：

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

注意：`having` 子句用于对分组进行过滤，因此 `having` 子句只能在有 `group by` 子句时才可以使用，没有 `group by` 子句，则不能使用 `having` 子句。

Hibernate 可将 HQL 语句直接翻译成数据库 SQL 语句。因此，如果底层的数据库支持在 `having` 和 `group by` 子句中出现的一般函数或聚集函数，则 HQL 的 `having` 与 `order by` 子句中也可以出现一般函数和聚集函数。例如：

```
select cat
from Cat cat
join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

注意：`group by` 子句与 `order by` 子句中都不能包含算术表达式。

## 9. 子查询

如果底层数据库支持子查询，则可以在 HQL 语句中使用子查询。与 SQL 中子查询相似的是，HQL 中的子查询也需要使用“`()`”括起来。例如：

```
from Cat as fatcat
where fatcat.weight > ( select avg(cat.weight) from DomesticCat cat )
```

如果 select 中包含多个属性，则应该使用元组构造符：

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

## 10. fetch 关键字

对于集合属性，Hibernate 默认采用延迟加载策略。例如，对于持久化类 Person，有集合属性 scores。加载 Person 实例时，默认不加载 scores 属性。如果 Session 被关闭，则 Person 实例将无法访问关联的 scores 属性。

为了解决该问题，可以在 Hibernate 映射文件中取消延迟加载，或者使用 fetch join。例如：

```
from Person as p join p.scores
```

上面的 fetch 语句将会初始化 person 的 scores 集合属性。

如果使用了属性级别的延迟加载，可以用 fetch all properties 来强制 Hibernate 立即抓取那些原本需要延迟加载的属性。例如：

```
from Document fetch all properties order by name
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

## 11. 命名查询

HQL 查询还支持将查询所用的 HQL 语句放入配置文件中，而不是代码中。通过这种方式，可以大大提高程序的解耦。

下面是使用 query 元素定义命名查询的配置文件代码：

```
<!-- 定义命名查询-->
<query name="myNamedQuery">
    <!-- 此处确定命名查询的 HQL 语句-->
    from Person as p where p.age > ?
</query>
```

该命名的 HQL 查询可以直接通过 Session 访问，调用命名查询的示例代码如下：

```
private void findByNamedQuery() throws Exception
{
    //获得 Hibernate Session 对象
    Session sess = HibernateUtil.currentSession();
    //开始事务
    Transaction tx = sess.beginTransaction();
    System.out.println("执行命名查询");
    //调用命名查询
    List pl = sess.getNamedQuery("myNamedQuery")
        //为参数赋值
        .setInteger(0, 20)
        //返回全部结果
        .list();
    //遍历结果集
    for (Iterator pit = pl.iterator(); pit.hasNext(); )
    {
        Person p = (Person) pit.next();
```

```

        System.out.println(p.getName());
    }
    //提交事务
    tx.commit();
    HibernateUtil.closeSession();
}

```

## 4.6.2 条件查询

条件查询是更具面向对象特色的数据查询方式，通过如下三个类完成。

- Criteria：代表一次查询。
- Criterion：代表一个查询条件。
- Restrictions：产生查询条件的工具类。

执行条件查询的步骤如下：

- (1) 获得 Hibernate 的 Session 对象。
- (2) 以 Session 对象创建 Criteria 对象。
- (3) 增加 Criterion 查询条件。
- (4) 执行 Criteria 的 list 等方法返回结果集。

看下面的条件查询示例：

```

private void test()
{
    //获取 Hibernate Session 对象
    Session session = HibernateUtil.currentSession();
    //开始事务
    Transaction tx = session.beginTransaction();
    //创建 Criteria 和添加查询条件同步完成
    //最后调用 list 方法，返回查询到的结果集。
    List l = session.createCriteria(Student.class)
        //此处增加的限制条件必须是 Student 已经存在的属性
        .add( Restrictions.gt("studentNumber", new Long(20050231)) )
        //如果要增加对 Student 关联类的属性限制，则必须重新创建
        //如果此关联属性是集合，则只要集合里任意一个对象的属性满足下面条件即可
        .createCriteria("enrolments")
        .add( Restrictions.gt("semester", new Short("2")) )
        .list();
    Iterator it = l.iterator();
    //遍历查询到的记录
    while (it.hasNext())
    {
        Student s = (Student)it.next();
        System.out.println(s.getName());
        Set enrolments = s.getEnrolments();
        Iterator iter = enrolments.iterator();
        while(iter.hasNext())
        {
            Enrolment e = (Enrolment)iter.next();
            System.out.println(e.getCourse().getName());
        }
    }
    tx.commit();
    HibernateUtil.closeSession();
}

```

```
}
```

在条件查询中, Criteria 接口代表一次查询, 该查询本身不具备任何的数据筛选功能; Session 调用 createCriteria(Class clazz)方法对某个持久化类创建条件查询实例。

Criteria 包含如下两个方法。

- Criteria setFirstResult(int firstResult): 设置查询返回的第一行记录。
- Criteria setMaxResults(int maxResults): 设置查询返回的记录数。

这两个方法与 Query 的用法相似, 都用于完成查询分页。

此外, Criteria 还包含如下常用方法。

- Criteria add(Criterion criterion): 增加查询条件。
- Criteria addOrder(Order order): 增加排序规则。
- List list(): 返回结果集。

Criterion 接口代表一个查询条件, 该查询条件由 Restrictions 负责产生, 而 Restrictions 是专门用于产生查询条件的工具类, 它的方法大部分都是静态方法, 常有的方法有如下几种。

- static Criterion allEq(Map propertyNameValues): 判断指定属性(由 Map 参数的 key 指定)和指定值(由 Map 参数的 value 指定)是否完全相等。
- static Criterion between(String propertyName, Object lo, Object hi): 判断属性值是否在某个值范围之内。
- static Criterion ilike(String propertyName, Object value): 判断属性值是否匹配某个字符串。
- static Criterion ilike(String propertyName, String value, MatchMode matchMode): 判断属性值是否匹配某个字符串, 并确定匹配模式。
- static Criterion in(String propertyName, Collection values): 判断属性值是否在某个集合内。
- static Criterion in(String propertyName, Object[] values): 判断属性值是否是数组元素的其中之一。
- static Criterion isEmpty(String propertyName): 判断属性值是否为空。
- static Criterion isNotEmpty(String propertyName): 判断属性值是否不为空。
- static Criterion isNotNull(String propertyName): 判断属性值是否为空。
- static CriterionisNull(String propertyName): 判断属性值是否不为空。
- static Criterion not(Criterion expression): 对 Criterion 求否。
- static Criterion sizeEq(String propertyName, int size): 判断某个属性的元素个数是否与 size 相等。
- static Criterion sqlRestriction(String sql): 直接使用 SQL 语句作为筛选条件。
- static Criterion sqlRestriction(String sql, Object[] values, Type[] types): 直接使用带参数占位符的 SQL 语句作为条件, 并指定多个参数值。
- static Criterion sqlRestriction(String sql, Object value, Type type): 直接使用带参数占位符的 SQL 语句作为条件, 并指定参数值。

Order 实例代表一个排序标准，并有如下构造器。

Order(String propertyName, boolean ascending): 根据 propertyName 排序，如果后一个参数为 true，则采用升序排序，否则采用降序排序。

如果需要使用关联类的属性来增加查询条件，则应该对属性再次使用 createCriteria 方法。看如下示例：

```
session.createCriteria(Person.class)
    .add(Restrictions.like("name", "dd%"))
    .createCriteria("addresses")
    .add(Restrictions.like("addressdetail", "上海%"))
    .list();
```

上面的代码表示建立 Person 类的条件查询，第一个查询条件是直接过滤 Person 的属性，即选出 name 属性以 dd 开始的 Person 实例，第二个查询条件则过滤 Person 的关联实例的属性，其中 addresses 是 Person 类的关联持久化类 Address，而 addressdetail 则是 Address 类的属性。值得注意的是，查询并不是查询 Address 持久化类，而是查询 Person 持久化类。

注意：使用关联类的条件查询，依然是查询原有持久化类的实例，而不是查询被关联类的实例。

### 4.6.3 SQL 查询

Hibernate 还支持使用 SQL 查询，使用 SQL 查询可以利用某些数据库的特性，或者用于将原有的 JDBC 应用迁移到 Hibernate 应用上。因此，使用命名的 SQL 查询不仅可以将 SQL 语句放在配置文件中配置，还可以用于调用存储过程，从而提高程序的解耦。如果是一个新的应用，通常不要使用 SQL 查询。

SQL 查询是通过 SQLQuery 接口来表示的，由于 SQLQuery 接口是 Query 接口的子接口，因此完全可以调用 Query 接口的方法，例如：

- setFirstResult(): 设置返回结果集的起始点。
- setMaxResults(): 设置查询获取的最大记录数。
- list(): 返回查询到的结果集。

但 SQLQuery 比 Query 多了两个重载的方法。

- addEntity: 将查询到的记录与特定的实体关联。
- addScalar: 将查询的记录关联成标量值。

执行 SQL 查询的步骤如下。

- (1) 获取 Hibernate Session 对象。
- (2) 编写 SQL 语句。
- (3) 以 SQL 语句作为参数，调用 Session 的 createSQLQuery 方法创建查询对象。
- (4) 如果 SQL 语句包含参数，则调用 Query 的 setXxx 方法为参数赋值。
- (5) 调用 SQLQuery 对象的 addEntity 或 addScalar 方法，将选出的结果与实体或标量值关联。

(6) 调用 Query 的 list 方法返回查询的结果集。

看下面的 SQL 查询示例：

```

private void test()
{
    //获取 Hibernate Session 对象
    Session session = HibernateUtil.currentSession();
    //开始事务
    Transaction tx = session.beginTransaction();
    //编写 SQL 语句
    String sqlString = "select {s.*} from student s where s.name like '马军'";
    //以 SQL 语句创建 SQLQuery 对象,
    List l = session.createSQLQuery(sqlString)
        //将查询到的记录与特定实体关联起来
        .addEntity("s", Student.class)
        //返回全部的记录集
        .list();
    //遍历结果集
    Iterator it = l.iterator();
    while (it.hasNext())
    {
        //因为将查询的结果与 Student 类关联, 因此返回是 Student 的集合
        Student s = (Student)it.next();
        Set enrolments = s.getEnrolments();
        Iterator iter = enrolments.iterator();
        while(iter.hasNext())
        {
            Enrolment e = (Enrolment)iter.next();
            System.out.println("=====");
            System.out.println(e.getCourse().getName());
            System.out.println("=====");
        }
    }
    //提交事务
    tx.commit();
    //关闭 Session
    HibernateUtil.closeSession();
}

```

上面的示例显示了将查询记录关联成一个实体。事实上，SQL 查询也支持将查询结果转换成标量值，转换成标量值时使用 addScalar 方法。例如：

```

Double max = (Double) session.createSQLQuery("select max(cat.weight) as
maxWeight from cats cat")
    .addScalar("maxWeight", Hibernate.DOUBLE);
    .uniqueResult();

```

在使用 SQL 查询时，如果需要将查询到的结果转换成特定实体，就要求为选出的字段另起别名。这别名不是随意命名的，而是以“实例名.属性名”的格式命名。例如：

```

//依次将多个选出的字段命名别名，命名别名时都以 ss 作为前缀,ss 是关联实体的别名
String sqlStr = "select stu.studentId as {ss.studentNumber} ,"
    + "stu.name as {ss.name} from "
    + "student as stu where stu.name like '杨海华'";
List l = session.createSQLQuery(sqlStr)

```

```
//将查询出的 ss 实例，关联到 Student 类
.addEntity("ss", Student.class)
.list();
```

在第一个示例中，以 {s.\*} 代表该表的全部字段，且关联实例的别名也被指定为 s。

**注意：**如果不使用 {s.\*} 的形式，就可让实体别名和表别名互不相同，关联实体的类型时，被关联的类必须有对应的 setter 方法。

## 1. 命名 SQL 查询

可以将 SQL 语句不放在程序中，而是放在配置文件中，这种方式以松耦合的方式配置 SQL 语句，可以提高程序解耦。

在 Hibernate 的映射文件中定义查询名，并确定查询所用的 SQL 语句，然后就可以直接调用该命名 SQL 查询。在这种情况下，无须调用 addEntity()方法，因为在配置命名 SQL 查询时，已经完成了查询结果与实体的关联。

下面是命名 SQL 查询的配置片段：

```
<!-- 每个 sql-query 元素定义一个命名 SQL 查询-->
<sql-query name="mySqlQuery">
    <!-- 关联返回的结果与实体类-->
    <return alias="s" class="Student"/>
    <!-- 定义命名 SQL 查询的 SQL 语句-->
    SELECT {s.*}
    from student s WHERE s.name like '杨海华'
</sql-query>
```

sql-query 元素是 hibernate-mapping 元素的子元素。因此以 sql-query 定义的名可以直接通过 Session 访问，即上面定义的 mySqlQuery 查询可以直接访问，下面是使用该命名 SQL 查询的示例代码：

```
private void testNamedSQL()
{
    //获取 Hibernate Session 对象
    Session session = HibernateUtil.currentSession();
    //开始事务
    Transaction tx = session.beginTransaction();
    //调用命名查询，直接返回结果
    List l = session.getNamedQuery("mySqlQuery")
        .list();
    //遍历结果集
    Iterator it = l.iterator();
    while (it.hasNext())
    {
        //在定义 SQL 查询时，已经将结果集与 Student 类关联起来。
        //因此，集合里的每个元素都是 Student 实例。
        Student s = (Student)it.next();
        Set enrolments = s.getEnrolments();
        Iterator iter = enrolments.iterator();
        while(iter.hasNext())
        {
            Enrolment e = (Enrolment)iter.next();
            System.out.println("=====");
            System.out.println(e.getCourse().getName());
        }
    }
}
```

```

        System.out.println("=====");
        System.out.println("=====");
    }
}
tx.commit();
HibernateUtil.closeSession();
}

```

## 2. 调用存储过程

Hibernate3 增加了对存储过程的支持，该存储过程只能返回一个结果集。

下面是 Oracle9i 的存储过程示例：

```

CREATE OR REPLACE FUNCTIONselectAllEmployees
  RETURN SYS_REFCURSOR
AS
  st_cursor SYS_REFCURSOR;
BEGIN
  OPEN st_cursor FOR
  SELECT EMPLOYEE, EMPLOYER,
  STARTDATE, ENDDATE,
  REGIONCODE, EID, VALUE, CURRENCY
  FROM EMPLOYMENT;
  RETURN st_cursor;
END;

```

如果需要调用该存储过程，可以先将其定义成命名 SQL 查询。例如：

```

<!-- 定义命名 SQL 查询，name 属性指定命名 SQL 查询名-->
<sql-query name="selectAllEmployees_SP" callable="true">
  <!-- 定义返回列与关联实体类属性之间的映射-->
  <return alias="emp" class="Employment">
    <!-- 依次定义每列与实体类属性的对应-->
    <return-property name="employee" column="EMPLOYEE"/>
    <return-property name="employer" column="EMPLOYER"/>
    <return-property name="startDate" column="STARTDATE"/>
    <return-property name="endDate" column="ENDDATE"/>
    <return-property name="regionCode" column="REGIONCODE"/>
    <return-property name="id" column="EID"/>
    <!-- 将两列值映射到一个关联类的引用属性-->
    <return-property name="salary">
      <!-- 映射列与引用属性之间的关联-->
      <return-column name="VALUE"/>
      <return-column name="CURRENCY"/>
    </return-property>
  </return>
  { ? = call selectAllEmployees() }
</sql-query>

```

调用存储过程还需要注意以下几个问题。

- 因为存储过程本身完成了查询的全部操作，因此，调用存储过程进行的查询无法使用 `setFirstResult()`/`setMaxResults()` 进行分页。
- 存储过程只能返回一个结果集，如果存储过程返回多个结果集，Hibernate 将仅处理第一个结果集，其他将被丢弃。
- 如果在存储过程里设定 `SET NOCOUNT ON`，将有更好的性能表现，也可以没有该设定。

## 4.6.4 数据过滤

数据过滤并不是一种常规的数据查询方法，而是一种整体的筛选方法。通过过滤数据也可对数据进行筛选，因此，笔者将其放在 Hibernate 的数据查询框架中讲解。

如果一旦启用了数据过滤器，则无论是数据查询，还是数据加载，该过滤器将自动作用于所有数据。只有满足过滤条件的记录才会被选出来。

过滤器与定义在类和集合上映射文件中的“`where`”约束子句非常相似，它们的区别是过滤器可以带参数，应用程序可以在运行时决定是否启用给定的过滤器，以及使用什么样的参数值。而映射文件的“`where`”属性将一直生效，且无法动态传入参数。

过滤器的用法与数据库视图很相似，区别是视图在数据库中已经完成定义，而过滤器则还需在应用程序中确定参数值。

过滤器的使用分成三步：

- (1) 定义过滤器，使用 `filter-def` 元素定义过滤器。
- (2) 使用过滤器，使用 `filter` 元素使用过滤器。
- (3) 在代码中启用过滤器。

前两个步骤都是在 Hibernate 的映射文件中完成的，其中 `filter-def` 是 `hibernate-mapping` 元素的子元素，用于定义一个过滤器；而 `filter` 元素是 `class` 与集合等元素的子元素，将指定的过滤器应用到指定的持久化类中。

一个持久化类或集合可以使用多个过滤器，但一个过滤器也可以作用于多个持久化类或集合。

看下面的映射文件示例：

```
<?xml version="1.0"?>
<!-- Hibernate 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Hibernate 配置文件的根元素-->
<hibernate-mapping>
    <!-- 每个 class 元素定义一个持久化类-->
    <class name="Category" table="category">
        <!-- 定义标识属性-->
        <id name="id" column="category_id" >
            <!-- 指定主键生成器策略-->
            <generator class="native"/>
        </id>
        <!-- 映射 name 属性-->
        <property name="name" type="string"/>
        <!-- 映射 effectiveStartDate 属性-->
        <property name="effectiveStartDate" column="eff_start_date"
type="java.util.Date"/>
        <!-- 映射 effectiveEndDate 属性-->
        <property name="effectiveEndDate" column="eff_end_date" type="java.
util.Date"/>
        <!-- 映射 N-N 关联属性-->
        <set cascade="none" inverse="true" name="products" table="product_>
```

```

category">
    <!-- 定义关联属性的 key, 对应连接表中外键列-->
    <key column="category_id"/>
    <!-- 定义关联属性-->
    <many-to-many column="product_id" class="Product" />
</set>
<!-- 使用过滤器, 并设置过滤器条件-->
<filter name="effectiveDate" condition=":asOfDate BETWEEN eff_start_
date and eff_end_date"/>
</class>
<!-- 定义第二个持久化类-->
<class name="Product" table="product">
    <!-- 定义标识属性-->
    <id name="id" column="product_id" >
        <!-- 指定主键生成器策略-->
        <generator class="native"/>
    </id>
    <!-- 映射 name 属性-->
    <property name="name" type="string"/>
    <!-- 映射 stockNumber 属性-->
    <property name="stockNumber" column="stock_number" type="int"/>
    <!-- 映射 effectiveStartDate 属性-->
    <property name="effectiveStartDate" column="eff_start_date"
type="java.util.Date"/>
    <!-- 映射 effectiveEndDate 属性-->
    <property name="effectiveEndDate" column="eff_end_date" type="java.
util.Date"/>
    <!-- 映射 N - N 关联属性-->
    <set cascade="all" name="categories" fetch="join" table="product_
category" >
        <!-- 定义关联属性的 key, 对应连接表中外键列-->
        <key column="product_id"/>
        <!-- 定义关联属性-->
        <many-to-many column="category_id"
            class="Category" fetch="join">
            <!-- 对关联属性使用第一个过滤器-->
            <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_date and
eff_end_date"/>
            <!-- 对关联属性使用第二个过滤器-->
            <filter name="category" condition="category_id = :catId"/>
        </many-to-many>
    </set>
    <filter name="effectiveDate" condition=":asOfDate BETWEEN eff_start_
date AND eff_end_date"/>
</class>
<!-- 定义第一个过滤器, 该过滤器包含一个 date 类型的参数-->
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>
<!-- 定义第二个过滤器, 该过滤器包含一个 long 类型的参数-->
<filter-def name="category">
    <filter-param name="catId" type="long"/>
</filter-def>
</hibernate-mapping>

```

在上面的配置文件中, 定义了两个过滤器, 过滤器的定义通过 filter-def 元素完成。定义过滤器时, 只需指定过滤器的名字, 以及过滤器的参数即可。例如 Java 里的一个方

法声明，只需方法名和参数列表，没有具体的方法实现。

过滤器的过滤条件是直到使用过滤器时才确定的，即通过 filter 元素来确定过滤条件。其中，filter 的 condition 属性用于确定过滤条件，满足该条件的记录才会被加载。

系统默认不启用过滤器，必须通过 enableFilter(String filterName) 才可以启用过滤器，该方法返回一个 Filter 实例，在 Filter 中包含了 setParameter 方法用于为过滤器参数赋值。

一旦启用了过滤器，过滤器将在整个 Session 内有效，所有的数据加载将自动应用该过滤条件，直到调用 disableFilter 方法为止。

看下面的使用过滤器的示例代码：

```
private void test() throws Exception
{
    //获取 Hibernate Session 对象
    Session session = HibernateUtil.currentSession();
    //开始事务
    Transaction tx = session.beginTransaction();
    //启用第一个过滤器
    session.enableFilter("effectiveDate")
        //为过滤器设置参数
        .setParameter("asOfDate", new Date());
    //启动第二个过滤器
    session.enableFilter("category")
        //为过滤器设置参数
        .setParameter("catId", new Long(2));
    //执行查询，该查询没有任何的查询条件
    Iterator results = session.createQuery("from Product as p")
        .iterate();
    //遍历结果集
    while (results.hasNext())
    {
        Product p = (Product)results.next();
        System.out.println(p.getName());
        //此处获取 Product 关联的种类，过滤器也将自动应用过滤
        Iterator it = p.getCategories().iterator();
        System.out.println(p.getCategories().size());
        while (it.hasNext())
        {
            Category c = (Category)it.next();
            System.out.println(c.getName());
        }
    }
    tx.commit();
    HibernateUtil.closeSession();
}
```

从过滤器定义的常用数据筛选规则中我们可以了解到，如果是临时的数据筛选，最好还是使用常规查询，对于以前使用行列表达式视图的地方，可以考虑使用过滤器。

## 4.7 事件框架

通常，在 Hibernate 执行持久化过程中，应用程序无法参与其中。因为所有的数据持久化操作，对用户都是透明的，所以用户无法加入自己的动作。

通过事件框架，Hibernate 允许应用程序能响应特定的内部事件，从而实现某些通用的功能，或者允许对 Hibernate 功能进行扩展。

Hibernate 的事件框架由以下两个部分组成。

- 拦截器机制：对于特定动作拦截，回调应用中的特定动作。
- 事件系统：重写 Hibernate 的事件监听器。

### 4.7.1 拦截器

通过 Interceptor 接口，可以从 Session 中回调应用程序的特定方法，这种回调机制可让应用程序在持久化对象被保存、更新、删除或加载之前，对其进行检查并修改其属性。

通过 Interceptor 接口，可以在数据进入数据库之前，对数据进行最后的检查，如果数据不符合要求，则可以修改数据，从而避免非法数据进入数据库。当然，通常无须这样做，只有在某些特殊的场合下，才考虑使用拦截器完成检查功能。

使用拦截器时按如下步骤进行：

- (1) 定义实现 Interceptor 接口的拦截器类。
- (2) 通过 Session 起用拦截器，或者通过 Configuration 启用全局拦截器。

下面是一个拦截器的示例代码，该拦截器没有进行任何实际的操作，仅仅打印出标志代码：

```
public class MyInterceptor extends EmptyInterceptor
{
    //更新的次数
    private int updates;
    //插入的次数
    private int creates;
    //当删除数据时，将调用 onDelete 方法
    public void onDelete(Object entity, Serializable id, Object[]
        state, String[] propertyNames, Type[] types)
    {
        // do nothing
    }
    //同步 Session 和数据库中的数据
    public boolean onFlushDirty(Object entity, Serializable id,
        Object[] currentState, Object[] previousState,
        String[] propertyNames, Type[] types)
    {
        //每同步一次，修改的累加器加 1
        updates++;
        for ( int i=0; i < propertyNames.length; i++ )
        {
            if ( "lastUpdateTimestamp".equals( propertyNames[i] ) )
            {
                currentState[i] = new Date();
                return true;
            }
        }
        return false;
    }
    //加载持久化实例时调用该方法
}
```

```

public boolean onLoad(Object entity, Serializable id,
    Object[] state, String[] propertyNames, Type[] types)
{
    System.out.println("=====");
    for ( int i=0; i < propertyNames.length; i++ )
    {
        if ( "name".equals( propertyNames[i] ) )
        {
            System.out.println(state[i]);
            state[i] = "aaa";
            return true;
        }
    }
    return false;
}
//保存持久化实例时，调用该方法
public boolean onSave(Object entity, Serializable id,
    Object[] state, String[] propertyNames, Type[] types)
{
    creates++;
    for ( int i=0; i<propertyNames.length; i++ )
    {
        if ( "createTimestamp".equals( propertyNames[i] ) )
        {
            state[i] = new Date();
            return true;
        }
    }
    return false;
}
//提交刷新
public void postFlush(Iterator entities)
{
    System.out.println("创建的次数: " + creates + ", 更新的次数: "
        + updates);
}
public void preFlush(Iterator entities)
{
    updates=0;
    creates=0;
}
//事务提交之前触发该方法
public void beforeTransactionCompletion(Transaction tx)
{
    System.out.println("事务即将结束");
}
//事务提交之后触发该方法
public void afterTransactionCompletion(Transaction tx)
{
    System.out.println("事务已经结束");
}
}

```

在上面的拦截器实现类中，实现了很多方法，这些方法都是在 Hibernate 执行特定动作时自动调用。

下面是关于拦截器的使用，拦截器的使用有以下两种方法：

- 通过 SessionFactory 的 openSession(Interceptor in)方法打开一个带局部拦截器的

Session。

- 通过 Configuration 的 setInterceptor(Interceptor in)方法设置全局拦截器。

下面是使用局部拦截器的示例代码：

```
public class HibernateUtil
{
    //静态类属性 SessionFactory
    public static final SessionFactory sessionFactory;
    //静态初始化块，完成静态属性的初始化
    static
    {
        try
        {
            //采用默认的 hibernate.cfg.xml 来启动一个 Configuration 的实例
            Configuration configuration=new Configuration().configure();
            //由 Configuration 的实例来创建一个 SessionFactory 实例
            sessionFactory = configuration.buildSessionFactory();
        }
        catch (Throwable ex)
        {
            System.err.println("初始化 sessionFactory 失败." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    //ThreadLocal 是隔离多个线程的数据共享，不存在多个线程之间共享资源，因此不再需要对
    //线程同步
    public static final ThreadLocal session = new ThreadLocal();
    //不加拦截器的打开 Session 方法
    public static Session currentSession() throws HibernateException
    {
        Session s = (Session) session.get();
        //如果该线程还没有 Session，则创建一个新的 Session
        if (s == null)
        {
            s = sessionFactory.openSession();
            //将获得的 Session 变量存储在 ThreadLocal 变量 session 里
            session.set(s);
        }
        return s;
    }
    //加拦截器的打开 Session 方法
    public static Session currentSession(Interceptor it) throws
HibernateException
    {
        Session s = (Session) session.get();
        //如果该线程还没有 Session，则创建一个新的 Session
        if (s == null)
        {
            //以拦截器创建 Session 对象
            s = sessionFactory.openSession(it);
            //将获得的 Session 变量存储在 ThreadLocal 变量 session 里
            session.set(s);
        }
        return s;
    }
    //关闭 Session 对象
    public static void closeSession() throws HibernateException
```

```

    {
        Session s = (Session) session.get();
        if (s != null)
            s.close();
        session.set(null);
    }
}
}

```

在上面的 Hibernate 工具类中，提供两个 currentSession 方法，分别用于不使用拦截器获取 Session 对象和使用拦截器获取 Session 对象。

下面是主程序使用拦截器的代码：

```

private void testUser()
{
    //以拦截器开始 Session
    Session session = HibernateUtil.currentSession(new MyInterceptor());
    //开始事务
    Transaction tx = session.beginTransaction();
    //执行下面代码时，可以看到系统回调 onSave 等方法
    /*
    User u = new User();
    u.setName("Yeeku Lee");
    u.setAge(28);
    u.setNationality("中国");
    session.persist(u);
    u.setAge(29);
    u.setAge(30);
    session.persist(u);
    */
    //执行下面代码，可以看到系统回调 onLoad 等方法
    Object o = session.load(User.class, new Integer(1));
    System.out.println(o);
    User u = (User)o;
    System.out.println(u.getName());
    //提交事务时，可看到系统回调事务相关方法。
    tx.commit();
    HibernateUtil.closeSession();
}

```

## 4.7.2 事件系统

Hibernate3 事件系统是功能更强大的事件框架，该事件系统可以替代拦截器，也可以作为拦截器的补充来使用。

基本上，Session 接口的每个方法都有对应的事件。比如 LoadEvent, FlushEvent, 等。当 Session 调用某个方法时，Hibernate Session 会生成对应的事件，并激活对应事件监听器。在系统默认监听器实现的处理过程中，完成了对所有的数据持久化操作，包括插入和修改等操作。如果用户定义了自己的监听器，则意味着用户必须完成对象的持久化操作。

例如：可以在系统中实现并注册 LoadEventListener 监听器，该监听器负责处理所有调用 Session 的 load()方法的请求。

由于监听器是单态模式对象，即所有同类型的事件处理共享同一个监听器实例，因此监听器不应该保存任何状态，即不应该使用成员变量。

使用事件系统时按如下步骤进行：

- (1) 实现自己的事件监听器类。
- (2) 注册自定义事件监听器，代替系统默认的事件监听器。

实现用户的自定义监听器有如下三个方法。

- 实现对应的监听器接口：这是不可思议的，实现接口必须实现接口内的所有的方法，关键是必须实现 Hibernate 对应的持久化操作，即数据库访问，这意味着程序员完全取代了 Hibernate 的底层操作。
- 继承事件适配器：可以选择性地实现需要关注的方法，但依然需要替代 Hibernate 完成数据库访问。这也不太现实。
- 继承系统默认的事件监听器：扩展特定方法。

实际上，前两种方法很少使用。因为 Hibernate 的持久化操作也是通过这些监听器实现的，如果用户取代了这些监听器，则应该自己实现所有的持久化操作，这意味着用户放弃了 Hibernate 的持久化操作，而改为自己完成 Hibernate 的核心操作。

通常推荐采用第三种方法实现自己的事件监听器。因为 Hibernate 默认的事件监听器都被声明成 non-final，从而方便用户继承。

下面是用户自定义监听器的示例：

```
//自定义 LoadListener，继承默认的 DefaultLoadEventListener 实现类
public class MyLoadListener extends DefaultLoadEventListener
{
    //在 LoadEventListener 接口仅仅定义了这个方法
    public Object onLoad(LoadEvent event,
                         LoadEventListener.LoadType loadType) throws HibernateException
    {
        //先调用父类的 onLoad 方法，从而完成默认的持久化操作
        Object o = super.onLoad(event, loadType);
        //加入用户的自定义处理
        System.out.println("自定义的 load 事件");
        System.out.println(event.getEntityClassName() + "======" +
                           event.getEntityId());
        return o;
    }
}
```

下面还有一个 MySaveListener，用于监听 SaveEvent 事件：

```
//自定义 SaveListener，继承默认的 DefaultSaveEventListener 实现类
public class MySaveListener extends DefaultSaveEventListener
{
    //该方法完成实际的数据插入动作
    protected Serializable performSaveOrUpdate(SaveOrUpdateEvent event)
    {
        //先执行用户自定义的操作
        System.out.println(event.getObject());
        //调用父类的默认持久化操作
        return super.performSaveOrUpdate(event);
    }
}
```

注意：在扩展用户自定义监听器时，别忘了调用父类的对应方法。

注册用户自定义监听器也有以下两种方法。

- 编程式：通过使用 Configuration 对象编程注册。
- 声明式：在 Hibernate 的 XML 格式的配置文件中进行声明，使用 Properties 格式的配置文件将无法配置自定义监听器。

通过编程式方式使用自定义监听器的代码如下：

```
public class HibernateUtil2
{
    //静态类属性 SessionFactory
    public static final SessionFactory sessionFactory;
    //静态初始化块，完成静态属性的初始化
    static
    {
        try
        {
            Configuration cfg = new Configuration();
            //注册 loadEventListener 监听器
            cfg.getSessionEventConfig().setLoadEventListener(
                new MyLoadListener() );
            //注册 saveListener 监听器
            cfg.getSessionEventConfig().setSaveEventListener(
                new MySaveListener() );
            //由 Configuration 的实例来创建一个 SessionFactory 实例
            sessionFactory = cfg.configure().buildSessionFactory();
        }
        catch (Throwable ex)
        {
            System.err.println("初始化 sessionFactory 失败." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    //ThreadLocal 是隔离多个线程的数据共享，不存在多个线程之间共享资源，因此不再需要对
    //线程同步
    public static final ThreadLocal session = new ThreadLocal();
    //不加拦截器的打开 Session 方法
    public static Session currentSession() throws HibernateException
    {
        Session s = (Session) session.get();
        //如果该线程还没有 Session，则创建一个新的 Session
        if (s == null)
        {
            s = sessionFactory.openSession();
            //将获得的 Session 变量存储在 ThreadLocal 变量 session 里
            session.set(s);
        }
        return s;
    }
    //关闭 Session 对象
    public static void closeSession() throws HibernateException
    {
        Session s = (Session) session.get();
        if (s != null)
```

```
        s.close();
        session.set(null);
    }
}
```

如果不想修改代码，也可以在配置文件中使用事件监听器，注册事件监听器的 Hibernate 配置文件代码如下：

```
<?xml version='1.0' encoding="GBK'?>
<!-- Hibernate 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<!-- Hibernate 配置文件的根元素-->
<hibernate-configuration>
    <session-factory>
        <!-设置数据库驱动 -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</
        property>
        <!-- 数据库服务的 url-->
        <property name="connection.url">jdbc:mysql://localhost/hibernate</
        property>
        <!-- 数据库服务的用户名-->
        <property name="connection.username">root</property>
        <!-- 数据库服务的密码-->
        <property name="connection.password">32147</property>
        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">5</property>
        <!-- 设置数据库方言 -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</
        property>
        <!-- 显示 Hibernate 生成的 SQL 语句 -->
        <property name="show_sql">true</property>
        <!-- 配置应用启动时，是否自动建表 -->
        <property name="hbm2ddl.auto">update</property>
        <!-- 列出所有的持久化映射文件-->
        <mapping resource="User.hbm.xml"/>
        <!-- 注册事件监听器-->
        <listener type="load" class="lee.MyLoadListener"/>
        <listener type="save" class="lee.MySaveListener"/>
    </session-factory>
</hibernate-configuration>
```

使用配置文件注册事件监听器虽然方便，但也有不利之处：通过配置文件注册的监听器不能共享实例；如果多个<listener/>元素中使用了相同的类，则每一个引用都将产生一个新拦截器实例；如果需要在多个事件之间共享监听器的实例，则必须使用编程式方式来注册事件监听器。

**注意：**虽然监听器类实现了特定监听器的接口，但在注册时还要明确指出注册的事件。因为一个类可能实现多个监听器的接口，所以在注册时明确指定要监听的事件，可以使得启用或者禁用某个事件的监听配置工作更加简单。

## 本章小结

本章首先全面介绍了持久层框架：Hibernate。包括 Hibernate 配置文件、映射文件，以及映射文件的结构和大部分 XML 元素的含义。然后，详细介绍了 Hibernate 的关联关系映射，包括单向及双向的各种关联映射。还详细介绍了 Hibernate 的查询体系，包括 HQL 查询，条件查询，SQL 查询和数据过滤等。最后，介绍了 Hibernate 的拦截器和事件系统。

# 第5章

## Spring 介绍

### 本章要点

- » Spring 的下载和安装
- » Spring 的核心机制
- » Spring 的 bean 及 BeanFactory
- » 依赖注入的方式
- » bean 的基本行为、生命周期
- » bean 后处理器，容器后处理器
- » ApplicationContext 介绍
- » 加载多个配置文件的方法

Spring 为企业的应用的开发提供一个轻量级的解决方案。该解决方案包括：基于依赖注入的核心机制，基于 AOP 的声明式事务管理与多种持久层技术的整合，以及优秀的 Web MVC 框架等。

Spring 为 J2EE 应用的表现层、业务逻辑层及数据持久层都提供了极好的解决方案，因为 Spring 提供的不仅仅是一种框架，而且提供了一种企业应用的开发规范。Spring 是实际开发的抽象，其提供的“模板设计”大大简化了应用的开发。

Spring 的系列 Template 将通用步骤以优雅的方式完成，留给开发者的仅仅是与特定应用相关的部分，从而大大提高企业应用的开发效率。

Spring 支持对 POJO 的管理，能将 J2EE 应用各层的对象“焊接”在一起，甚至这些对象无须是标准的 JavaBean。

## 5.1 Spring 的起源和背景

2002 年 wrox 出版了《Expert one on one J2EE design and development》一书。该书的作者是 Rod Johnson。在书中，Johnson 对传统的 J2EE 架构提出深层次的思考和质疑，并提出 J2EE 的实用主义思想。

2003 年，J2EE 领域出现一个新的框架：Spring，该框架同样出自 Johnson 之手。事实上，Spring 框架是《Expert one on one J2EE design and development》一书中思想的全面体现和完善，Spring 对实用主义 J2EE 思想进一步改造和扩充，使其发展成更开放、清晰、全面及高效的开发框架。一经推出，就得到众多开发者的拥戴。

传统 J2EE 应用的开发效率低，应用服务器厂商对各种技术的支持并没有真正统一，导致 J2EE 的应用并没有真正实现 Write Once 及 Run Anywhere 的承诺。Spring 作为开源的中间件，独立于各种应用服务器，甚至无须应用服务器的支持，也能提供应用服务器的功能，如声明式事务等。

Spring 致力于 J2EE 应用的各层的解决方案，而不是仅仅专注于某一层的方案。可以说 Spring 是企业应用开发的“一站式”选择，并贯穿表现层、业务层及持久层。然而，Spring 并不想取代那些已有的框架，而与它们无缝地整合。

总结起来，Spring 有如下优点：

- 低侵入式设计，代码污染极低。
- 独立于各种应用服务器，可以真正实现 Write Once, Run Anywhere 的承诺。
- Spring 的 DI 机制降低了业务对象替换的复杂性。
- Spring 并不完全依赖于 Spring，开发者可自由选用 Spring 框架的部分或全部。

## 5.2 Spring 的下载和安装

下载和安装 Spring 请按如下步骤进行。

(1) 登录 <http://www.springframework.org> 站点，下载 Spring 的最新稳定版。笔者建议下载 spring-framework-1.2.8-with-dependencies.zip 包，该压缩包不仅包含 Spring 的开发包，而且包含 Spring 编译和运行所依赖的第三方类库。

解压缩下载到的压缩包，解压缩后的文件夹下应有如下几个文件夹。

- dist：该文件夹下放 Spring 的 jar 包，通常只需要 spring.jar 文件即可。该文件夹下还有一些类似 spring-Xxx.jar 的压缩包，这些压缩包是 spring.jar 压缩包的子模块压缩包。除非确定整个 J2EE 应用只需使用 Spring 的某一方面时，才考虑使用这种分模块压缩包。通常建议使用 spring.jar。
- docs：该文件夹下包含 Spring 的相关文档、开发指南及 API 参考文档。
- lib：该文件夹下包含 Spring 编译和运行所依赖的第三方类库，该路径下的类库并

不是 Spring 必需的，但如果需要使用第三方类库的支持，这里的类库就是必需的。

- samples：该文件夹下包含 Spring 的几个简单示例，可作为 Spring 入门学习的案例。
- src：该文件夹下包含 Spring 的全部源文件，如果在开发过程中有地方无法把握，可以参考该源文件，了解底层的实现。
- test：该文件夹下包含 Spring 的测试示例。
- tiger：该路径下存放关于 JDK1.5 的相关内容。
- 解压缩后的文件夹下，还包含一些关于 Spring 的 license 和项目相关文件。

(2) 将 spring.jar 复制到项目的 CLASSPATH 路径下，对于 Web 应用，将 spring.jar 文件复制到 WEB-INF/lib 路径下，该应用即可以利用 Spring 框架了。

(3) 通常 Spring 的框架还依赖于其他的一些 jar 文件，因此还须将 lib 下对应的包复制到 WEB-INF/lib 路径下，具体要复制哪些 jar 文件，取决于应用所需要使用的项目。通常需要复制 cglib, dom4j, jakarta-commons, log4j 等文件夹下的 jar 文件。

(4) 为了编译 Java 文件，可以找到 Spring 的基础类，将 spring.jar 文件的路径添加到环境变量 CLASSPATH 中。当然，也可使用 ANT 工具，但无须添加环境变量。

## 5.3 Spring 实现两种设计模式

在 Spring 中大量使用的以下两种设计模式：

- 工厂模式
- 单态模式

工厂模式可将 Java 对象的调用者从被调用者的实现逻辑中分离出来，调用者只需关心被调用者必须满足的规则（接口），而不必关心实例的具体实现过程。这是面向接口编程的优势，能提高程序的解耦，避免所有的类以硬编码方式耦合在一起。

如果所有的类直接耦合，极易形成“骨牌效应”，假如 B 类调用了 A 类，一旦 A 类需要修改，则 B 类也需要修改；假如 C 类调用了 B 类，则 C 类也需要修改……依次类推，从而导致整个系统都需要改写。造成“牵一发而动全身”，而系统重构的代价是相当高的。

Spring 倡导“面向接口编程”，可以避免上述的问题，使设计良好的架构可保证系统重构的工作被封闭在重构的层内，绝不会影响其他层，这可以在本书后面的示例中看到。

Spring 容器是实例化和管理全部 bean 的工厂，Spring 默认将所有的 bean 设置成单态模式，无须自己完成单态模式，即对所有相同 id 的 bean 请求都将返回同一个共享实例。因此，单态模式可大大降低 Java 对象在创建和销毁时的系统开销。

### 5.3.1 单态模式的回顾

单态模式限制了类实例的创建，但采用这种模式设计的类，可以保证仅有一个实例，并可提供访问该实例的全局访问点。J2EE 应用的大量组件，都需要保证一个类只有一个

实例。比如数据库引擎访问点只能有一个。

更多的时候，为了提高性能，程序应尽量减少 Java 对象的创建和销毁时的开销。使用单态模式可避免 Java 类被多次实例化，让相同类的全部实例共享同一内存区。

为了防止单态模式的类被多次实例化，应将类的构造器设成私有，这样就保证了只能通过静态方法获得类实例。而该静态方法则保证每次返回的实例都是同一个，这就需将该类的实例设置成类属性，由于该属性需要被静态方法访问，因此该属性应设成静态属性。

下面给出单态模式的示例代码：

```
//单态模式测试类
public class SingletonTest
{
    //该类的一个普通属性。
    int value ;
    //使用静态属性类保存该类的一个实例。
    private static SingletonTest instance;
    //构造器私有化，避免该类被多次实例。
    private SingletonTest()
    {
        System.out.println("正在执行构造器...") ;
    }
    //提供静态方法来返回该类的实例。
    public static SingletonTest getInstance()
    {
        //实例化类实例前，先检查该类的实例是否存在
        if (instance == null)
        {
            //如果不存在，则新建一个实例。
            instance = new SingletonTest();
        }
        //返回该类的成员变量：该类的实例。
        return instance;
    }
    //以下提供对普通属性 value 的 setter 和 getter 方法
    public int getValue()
    {
        return value;
    }
    public void setValue(int values)
    {
        this.value = value;
    }
    public static void main(String[] args)
    {
        SingletonTest t1 = SingletonTest.getInstance();
        SingletonTest t2 = SingletonTest.getInstance();
        t2.setValue(9);
        System.out.println(t1 == t2);
    }
}
```

从程序最后的打印结果可以看出，该类的两个实例完全相同。这证明单态模式类的全部实例是同一共享实例。程序里虽然获得了类的两个实例，但实际上只执行一次构造

器，因为对于单态模式的类，无论有多少次的创建实例请求，都只执行一次构造器。

### 5.3.2 工厂模式的回顾

工厂模式是根据调用数据返回某个类的一个实例，此类可以是多个类的某一个类。通常，这些类满足共同的规则（接口）或父类。调用者只关心工厂生产的实例是否满足某种规范，即实现的某个接口是否可供自己正常调用（调用者仅仅使用）。该模式给对象之间作出了清晰的角色划分，降低程序的耦合。

接口产生的全部实例通常用于实现相同接口，接口里定义了全部实例共同拥有的方法，这些方法在不同的实现类中实现的方式不同。从而使程序调用者无须关心方法的具体实现，降低了系统异构的代价。

下面是工厂模式的示例代码：

```
//Person 接口定义
public interface Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */
    public String sayHello(String name);
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name);
}
```

该接口定义了 Person 的规范，规范要求实现该接口的类必须具有这两个方法：能打招呼，能告别。

```
//American 类实现 Person 接口
public class American implements Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */
    public String sayHello(String name)
    {
        return name + ",Hello";
    }
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name)
    {
        return name + ",Good Bye";
    }
}
```

下面是实现 Person 接口的另一个实现类：Chinese

```
public class Chinese implements Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */
    public String sayHello(String name)
    {
        return name + ", 您好";
    }
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name)
    {
        return name + ", 下次再见";
    }
}
```

然后看 Person 工厂的代码：

```
public class PersonFactory
{
    /**
     * 获得 Person 实例的实例工厂方法
     * @ param ethnic 调用该实例工厂方法传入的参数
     * @ return 返回 Person 实例
     */
    public Person getPerson(String ethnic)
    {
        //根据参数返回 Person 接口的实例。
        if (ethnic.equalsIgnoreCase("chin"))
        {
            return new Chinese();
        }
        else
        {
            return new American();
        }
    }
}
```

以上是最简单的工厂模式框架，其主程序部分如下：

```
public class FactoryTest
{
    public static void main(String[] args)
    {
        //创建 PersonFactory 的实例，获得工厂实例
        PersonFactory pf = new PersonFactory();
        //定义接口 Person 的实例，面向接口编程
        Person p = null;
        //使用工厂获得 Person 的实例
        p = pf.getPerson("chin");
    }
}
```

```
//下面调用 Person 接口的方法
System.out.println(p.sayHello("wawa"));
System.out.println(p.sayGoodBye("wawa"));
//使用工厂获得 Person 的另一个实例
p = pf.getPerson("ame");
//再次调用 Person 接口的方法
System.out.println(p.sayHello("wawa"));
System.out.println(p.sayGoodBye("wawa"));
}
}
```

由此可看出，主程序从 Person 接口的具体类中解耦出来，而且程序调用者无须关心 Person 的实例化过程，主程序仅仅与工厂服务定位结合在一起，可获得所有工厂能产生的实例。具体类的变化，接口无须发生任何改变，调用者程序代码部分也无须发生任何改动。

下面是 Spring 对这两种模式的实现。

### 5.3.3 Spring 对单态与工厂模式的实现

随着 Spring 提供工厂模式的实现，在使用 Spring 时，无须自己提供工厂类。因为 Spring 容器是最大的工厂，而且是个功能超强的工厂。Spring 使用配置文件管理所有的 bean，其配置文件中 bean 由 Spring 工厂负责生成和管理。下面是关于两个实例的配置文件：

```
<!-- 下面是 xml 文件的文件头-->
<?xml version="1.0" encoding="gb2312"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- beans 是 Spring 配置文件的根元素-->
<beans>
    <!-- 定义第一个 bean,该 bean 的 id 为 chinese-->
    <bean id="chinese" class="lee.Chinese"/>
    <!-- 定义第二个 bean,该 bean 的 id 为 american-->
    <bean id="american" class="lee.American"/>
</beans>
```

主程序部分如下：

```
public class SpringTest
{
    public static void main(String[] args)
    {
        //实例化 Spring 容器
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //定义 Person 接口的实例
        Person p = null;
        //通过 Spring 上下文获得 chinese 实例
        p = (Person)ctx.getBean("chinese");
        //执行 chinese 实例的方法
        System.out.println(p.sayHello("wawa"));
        System.out.println(p.sayGoodBye("wawa"));
        //通过 Spring 上下文获得 american 实例
    }
}
```

```

        p = (Person)ctx.getBean("american");
        //执行 american 实例的方法
        System.out.println(p.sayHello("wawa"));
        System.out.println(p.sayGoodBye("wawa"));
    }
}

```

使用 Spring 时：即使没有工厂类 PersonFactory，程序一样可以使用工厂模式，Spring 完全可以提供所有工厂模式的功能。

下面对主程序部分进行简单的修改：

```

public class SpringTest
{
    public static void main(String[] args)
    {
        //实例化 Spring 容器
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //定义 Person 接口的实例 p1
        Person p1 = null;
        //通过 Spring 上下文获得 chinese 实例
        p1 = (Person)ctx.getBean("chinese");
        //定义 Person 接口的实例 p1
        Person p2 = null;
        p2 = (Person)ctx.getBean("chinese");
        System.out.println(p1 == p2);
    }
}

```

程序执行的结果是：

```
true
```

表明 Spring 对接受容器管理的全部 bean，默认采用单态模式管理。笔者建议不要随便更改 bean 的行为方式。因为在性能上，单态的 bean 比非单态的 bean 更优秀。

仔细检查上面的代码，就会发现如下特点：

- 除测试用的主程序部分外，代码并未出现 Spring 特定的类和接口。
- 调用者代码，也就是测试用的主程序部分，仅仅面向 Person 接口编程，而无须知道实现类的具体名称。同时，可以通过修改配置文件来切换底层的具体实现类。
- 由于厂无须多个实例，因此工厂应该采用单态模式设计。其中 Spring 的上下文，也就是 Spring 工厂，已被设计成单态的。

Spring 工厂模式，不仅提供了创建 bean 的功能，还提供对 bean 生命周期的管理。最重要的是还可管理 bean 与 bean 之间的依赖关系。

## 5.4 Spring 的依赖注入

依赖注入（Dependency Injection）是时下的“流行语”，也是目前最优秀的解耦方式。使用依赖注入时，J2EE 应用中的各种组件不需要以硬编码方式耦合在一起，甚至无

须使用工厂模式。当某个 Java 实例需要其他 Java 实例时，系统会自动提供需要的实例，无须程序显式获取。

依赖注入，是 Spring 的核心机制，可以使 Spring 的 bean 以配置文件组织在一起，而不是以硬编码的方式耦合在一起。

## 5.4.1 理解依赖注入

因为某些历史原因，依赖注入还有一种称呼：控制反转（Inversion of Control）。

不管是依赖注入，还是控制反转，其含义完全相同。当某个 Java 实例（调用者）需要另一个 Java 实例（被调用者）时，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。而在依赖注入的模式下，创建被调用者的工作不再由调用者来完成，通常由 Spring 容器来完成，然后注入调用者，因此称为控制反转，也称为依赖注入。

不管是依赖注入，还是控制反转，都说明 Spring 采用动态及灵活的方式来管理各种对象，使对象与对象之间的具体实现互相透明。

为了更好地理解依赖注入，笔者建议参考人类社会的发展，看如下问题在各种社会形态里如何解决：一个人（Java 实例，调用者）需要一把斧子（Java 实例，被调用者）。

在“原始社会”里，几乎没有社会分工。需要斧子的人（调用者）只能自己去磨一把斧子（被调用者）。对应的情形为：Java 程序里的调用者自己创建被调用者。

进入“工业社会”后，随着工厂的出现，斧子不再由普通人完成，而在工厂里被生产出来。此时需要斧子的人（调用者）只需找到工厂，购买斧子，无须关心斧子的制造过程。对应简单工厂设计模式：调用者只需要定位工厂，无须管理被调用者具体的实现。

进入“共产主义”社会后，需要斧子的人甚至无须定位工厂，“坐等”社会提供即可。调用者无须关心被调用者的实现，无须理会工厂，等待 Spring 依赖注入即可。

在第一种情况下，由 Java 实例的调用者创建被调用的 Java 实例，调用者直接使用 new 关键字创建被调用者实例，其程序高度耦合，效率低下。在实际应用中极少使用这种方式。

在第二种情况下，调用者无须关心被调用者的具体实现过程，只需要找到符合某种标准（接口）的实例即可使用。此时调用的代码面向接口编程，可以让调用者和被调用者解耦，这也是工厂模式被大量使用的原因。但调用者需要自己定位工厂，使调用者与工厂耦合在一起。

第三种情况，是最理想的情况，程序完全无须理会被调用者的实现，也无须定位工厂，是最好的解耦方式。实例之间的依赖关系由容器提供。

所谓依赖注入，是指在程序运行过程中，如果需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部的注入。Spring 的依赖注入对调用者和被调用者几乎没有任何要求，完全支持对 POJO 之间依赖关系的管理。

依赖注入通常有两种：

- 设值注入
- 构造注入

## 5.4.2 设值注入

设值注入是指通过 `setter` 方法传入被调用者的实例。这种注入方式简单、直观，因而在 Spring 的依赖注入里大量使用。

`Person` 接口的代码如下：

```
//定义 Person 接口
public interface Person
{
    //Person 接口里定义一个使用斧子的方法
    public void useAxe();
}
```

`Axe` 接口的代码如下：

```
//定义 Axe 接口
public interface Axe
{
    //Axe 接口里有个砍的方法
    public void chop();
}
```

`Person` 的实现类代码如下：

```
//Chinese 实现 Person 接口
public class Chinese implements Person
{
    //面向 Axe 接口编程，而不是具体的实现类
    private Axe axe;
    //默认的构造器
    public Chinese()
    {
    }
    //设值注入所需的 setter 方法
    public void setAxe(Axe axe)
    {
        this.axe = axe;
    }

    //实现 Person 接口的 useAxe 方法
    public void useAxe()
    {
        System.out.println(axe.chop());
    }
}
```

`Axe` 的第一个实现类代码如下：

```
//Axe 的第一个实现类 StoneAxe
public class StoneAxe implements Axe
{
    //默认构造器
    public StoneAxe()
    {
```

```

        }
        //实现 Axe 接口的 chop 方法
        public String chop()
        {
            return "石斧砍柴好慢";
        }
    }
}

```

下面采用 Spring 的配置文件将 Person 实例和 Axe 实例组织在一起。配置文件如下所示：

```

<!-- 下面是标准的 xml 文件头 -->
<?xml version="1.0" encoding="gb2312"?>
<!-- 下面一行定义 Spring 的 xml 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 以上三行对所有的 Spring 配置文件都是相同的 -->
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 定义第一 bean，该 bean 的 id 是 chinese，class 指定该 bean 实例的实现类 -->
    <bean id="chinese" class="lee.Chinese">
        <!-- property 元素用来指定需要容器注入的属性，axe 属性需要容器注入
             此处是设值注入，因此 Chinese 类必须拥有 setAxe 方法 -->
        <property name="axe">
            <!-- 此处将另一个 bean 的引用注入给 chinese bean -->
            <ref local="stoneAxe"/>
        </property>
    </bean>
    <!-- 定义 stoneAxe bean -->
    <bean id="stoneAxe" class="lee.StoneAxe"/>
</beans>

```

从配置文件中可以看到 Spring 管理 bean 的灵巧性。bean 与 bean 之间的依赖关系被放在配置文件里组织，而不是写在代码里。通过配置文件的指定，Spring 能精确地为每个 bean 注入属性。因此，配置文件里 bean 的 class 元素不能是接口，而必须是真正的实现类。

另外，Spring 会自动接管每个 bean 定义里的 property 元素定义。Spring 会在执行无参数的构造器后，创建默认的 bean 实例，并调用对应的 setter 方法为程序注入属性值。在这里，property 定义的属性值将不再由该 bean 来主动创建和管理，而是接收 Spring 的注入。

每个 bean 的 id 属性是该 bean 的唯一标识，程序通过 id 属性来访问 bean，bean 与 bean 的依赖关系也通过 id 属性关联。

下面是主程序部分：

```

public class BeanTest
{
    // 主方法，程序的入口
    public static void main(String[] args) throws Exception
    {
        // 因为是独立的应用程序，显式了实例化 Spring 的上下文。
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        // 通过 Person bean 的 id 来获取 bean 实例，面向接口编程，因此
    }
}

```

```

    //此处强制类型转换为接口类型
    Person p = (Person)ctx.getBean("chinese");
    //直接执行 Person 的 userAxe() 方法。
    p.useAxe();
}
}

```

程序的执行结果如下：

石斧砍柴好慢

当主程序调用 Person 的 useAxe()方法时，该方法的方法体内需要使用 Axe 的实例，但程序里没有任何地方将特定的 Person 实例和 Axe 实例耦合在一起。或者说，程序里没有为 Person 实例传入 Axe 的实例，而 Axe 实例由 Spring 在运行期间动态注入。

Person 实例既不需要了解 Axe 实例的具体实现，也无须了解 Axe 的创建过程。程序在运行到需要 Axe 实例时，由 Spring 创建 Axe 实例，然后注入给需要 Axe 实例的调用者。因此，当 Person 实例运行到需要 Axe 实例的地方时，自然就产生了 Axe 实例，用来供 Person 实例使用。

下面也给出使用 Ant 编译和运行该应用的简单脚本：

```

<?xml version="1.0"?>
<!-- 定义编译该项目的基本信息-->
<project name="spring" basedir=". " default=". ">
    <!-- 定义编译和运行该项目时所需的库文件 -->
    <path id="classpath">
        <!-- 该路径下存放 spring.jar 和其他第三方类库 -->
        <fileset dir="..\..\lib">
            <include name="*.jar"/>
        </fileset>
        <!-- 同时还需要引用已经编译过的 class 文件-->
        <pathelement path=". "/>
    </path>
    <!-- 编译全部的 java 文件-->
    <target name="compile" description="Compile all source code">
        <!-- 指定编译后的 class 文件的存放位置 -->
        <javac destdir=". " debug="true"
            deprecation="false" optimize="false" failonerror="true">
            <!-- 指定需要编译的源文件的存放位置 -->
            <src path=". "/>
            <!-- 指定编译这些 java 文件需要的类库位置-->
            <classpath refid="classpath"/>
        </javac>
    </target>
    <!-- 运行特定的主程序 -->
    <target name="run" description="run the main class" depends="compile">
        <!-- 指定运行的主程序:lee.BeanTest。-->
        <java classname="lee.BeanTest" fork="yes" failonerror="true">
            <!-- 指定运行这些 java 文件需要的类库位置-->
            <classpath refid="classpath"/>
        </java>
    </target>
</project>

```

如果需要改写 Axe 的实现类，或者说提供另一个实现类给 Person 实例使用时。Person

接口、Chinese 类都无须改变，只需提供另一个 Axe 的实现类，然后对配置文件进行简单的修改即可。

Axe 的另一个实现类如下：

```
//Axe 的另一个实现类 SteelAxe
public class SteelAxe implements Axe
{
    //默认构造器
    public SteelAxe()
    {
    }
    //实现 Axe 接口的 chop 方法
    public String chop()
    {
        return "钢斧砍柴真快";
    }
}
```

然后，修改原来的 Spring 配置文件，在其中增加如下代码：

```
<!-- 定义一个 steelAxe bean-->
<bean id="steelAxe" class="lee.SteelAxe"/>
```

该行重新定义了一个 Axe 的实现类：SteelAxe。通过修改 chinese bean 的配置后，将原来传入 stoneAxe 的地方改为传入 steelAxe。也就是将

```
<ref local="stoneAxe"/>
```

改成

```
<ref local="steelAxe"/>
```

此时再次执行程序，将得到如下结果：

钢斧砍柴真快

由此可看出，Person 与 Axe 之间没有任何代码耦合关系，bean 与 bean 之间的依赖关系由 Spring 管理。采用以 setter 方法为目标 bean 注入属性的方式我们称为设值注入。

通过配置文件动态管理，可使对象与对象之间的依赖关系从代码里分离出来，业务对象的更换也变得相当简单。

### 5.4.3 构造注入

所谓构造注入，指通过构造函数来完成依赖关系的设定，而不是通过 setter 方法。

对前面代码 Chinese 类作简单的修改，修改后的代码如下：

```
//Chinese 实现 Person 接口
public class Chinese implements Person
{
    //面向 Axe 接口编程，而不是具体的实现类
    private Axe axe;
    //默认的构造器
```

```

public Chinese()
{
}
//构造注入所需的带参数的构造器
public Chinse(Axe axe)
{
    this.axe = axe;
}
//实现 Person 接口的 useAxe 方法
public void useAxe()
{
    System.out.println(axe.chop());
}
}

```

此时无须 Chinese 类里的 setAxe 方法，在构造 Person 实例时，Spring 为 Person 实例注入所依赖的 Axe 实例。

构造注入的配置文件也需作简单的修改，修改后的配置文件如下：

```

<!-- 下面是标准的 xml 文件头 -->
<?xml version="1.0" encoding="gb2312"?>
<!-- 下面一行定义 Spring 的 xml 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 上述三行对所有的 Spring 配置文件都是相同的 -->
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 定义第一个 bean，该 bean 的 id 是 chinese， class 指定该 bean 实例的实现类 -->
    <bean id="chinese" class="lee.Chinese">
        <constructor-arg><ref bean="steelAxe"/></constructor-arg>
    </bean>
    <!-- 定义 stoneAxe bean -->
    <bean id="steelAxe" class="lee.SteelAxe"/>
</beans>

```

由此可看出，执行效果与使用 steelAxe 设值注入时的执行效果完全相同。区别在于创建 Person 实例中 Axe 属性的时机不同——设值注入是先创建一个默认的 bean 实例，然后调用对应的 setter 方法注入依赖关系；而构造注入则在创建 bean 实例时，已经完成了依赖关系的注入。

#### 5.4.4 两种注入方式的对比

Spring 同时支持两种依赖注入方式：设值注入和构造注入。这两种注入方式各有其优、缺点。

##### 1. 设值注入的优点

- 设值注入与传统的 JavaBean 的写法更相似，程序开发人员更容易了解，接受。通过 setter 方法设定依赖关系显得更加直观、自然。
- 对于复杂的依赖关系，如果采用构造注入，会导致构造器过于臃肿，难以阅读。因为 Spring 在创建 bean 实例时，需要同时实例化其依赖的全部实例，因而导致

性能下降。而使用设值注入，则能避免这些问题。

- 尤其是在某些属性可选的情况下，多参数的构造器更加笨重。

## 2. 构造注入的优点

- 可以在构造器中决定依赖关系的注入顺序。例如，组件中其他依赖关系的注入，常常需要依赖于 Datasource 的注入。采用构造注入时，可以在代码中清晰地决定注入顺序，优先依赖的优先注入。
- 对于依赖关系无须变化的 bean，构造注入更有用处。因为没有 setter 方法，所有的依赖关系全部在构造器内设定。因此，无须担心后续的代码对依赖关系产生破坏。
- 依赖关系只能在构造器中设定，因为只有组件的创建者才能改变组件的依赖关系。对组件的调用者而言，组件内部的依赖关系完全透明，更符合高内聚的原则。

建议采用以设值注入为主，构造注入为辅的注入策略。对于依赖关系无须变化的注入，尽量采用构造注入；而其他的依赖关系的注入，则考虑采用设值注入。

## 5.5 bean 和 BeanFactory

bean 是 Spring 管理的基本单位，在 Spring 的 J2EE 应用中，所有的组件都是 bean，bean 包括数据源、Hibernate 的 SessionFactory 及事务管理器等。Spring 里的 bean 是非常广义的概念，任何的 Java 对象，Java 组件都可被当成 bean 处理。甚至这些组件并不是标准的 JavaBean。

整个应用中各层的对象都处于 Spring 的管理下，这些对象以 bean 的方式存在。Spring 负责创建 bean 实例，并管理其生命周期。bean 在 Spring 容器中运行时，无须感受 Spring 容器的存在，一样可以接受 Spring 的依赖注入，包括 bean 属性的注入，合作者的注入及依赖关系的注入等。

Spring 的容器有两个接口：BeanFactory 和 ApplicationContext，这两个接口的实例也被称为 Spring 上下文，它们都是产生 bean 的工厂，bean 是 Spring 工厂产生的实例。在 Spring 产生 bean 实例时，需要知道每个 bean 的实现类，而 bean 实例的使用者面向接口，无须关心 bean 实例的实现类。因为 Spring 工厂负责维护 bean 实例的实例化，所以使用者无须关心实例化。

bean 定义通常使用 XML 配置文件。正确定义的 bean 由 Spring 提供实例化，以及依赖关系的注入。bean 实例通过 BeanFactory 访问。对于大部分 J2EE 应用，bean 通过 ApplicationContext 提供访问，因为 ApplicationContext 是 BeanFactory 的子接口，提供比 BeanFactory 更多的功能。

### 5.5.1 Spring 容器

Spring 的容器最基本的接口就是：BeanFactory。BeanFactory 负责配置、创建及管理

bean，它有个子接口：ApplicationContext，因此也被称为 Spring 上下文。另外，Spring 容器还负责管理 bean 与 bean 之间的依赖关系。

BeanFactory 接口包含如下的基本方法。

- public boolean containsBean(String name): 判断 Spring 容器是否包含 id 为 name 的 bean 定义。
- public Object getBean(String name): 返回容器 id 为 name 的 bean。
- public Object getBean(String name, Class requiredType): 返回容器中 id 为 name，并且类型为 requiredType 的 bean。
- public Class getType(String name): 返回容器中 id 为 name 的 bean 的类型。

调用者只需使用 getBean 方法即可获得指定 bean 的引用，无须关心 bean 的实例化过程。即 bean 实例的创建过程完全透明。

BeanFactory 有很多实现类，通常使用 org.springframework.beans.factory.xml.XmlBeanFactory 类。但对大部分 J2EE 应用而言，推荐使用 ApplicationContext，因为其是 BeanFactory 的子接口，其常用的实现类是 org.springframework.context.support.FileSystemXmlApplicationContext。

创建 BeanFactory 的实例时，必须提供 Spring 容器管理 bean 的详细配置信息。Spring 的配置信息通常采用 XML 配置文件来设置。因此，在创建 BeanFactory 实例时，应该提供 XML 配置文件作为参数，XML 配置文件通常使用 Resource 对象传入。

Resource 接口：用于访问配置文件资源。

对于大部分 J2EE 应用而言，可在启动 Web 应用时自动加载 ApplicationContext 实例，接受 Spring 管理的 bean 无须知道 ApplicationContext 的存在，也一样可以利用 ApplicationContext 的管理。对于独立的应用程序，也可通过如下方法来实例化 BeanFactory：

```
//以指定路径下 bean.xml 配置文件为参数，创建文件输入流
InputStream is = new FileInputStream("beans.xml");
//以指定的文件输入流 is，创建 Resource 对象
InputStreamResource isr = new InputStreamResource(is);
//以 Resource 对象作为参数，创建 BeanFactory 的实例
XmlBeanFactory factory = new XmlBeanFactory(isr);
```

或者采用如下方法：

```
//搜索 CLASSPATH 路径，以 CLASSPATH 路径下的 beans.xml 文件创建 Resource 对象
ClassPathResource res = new ClassPathResource("beans.xml");
//以 Resource 对象为参数，创建 BeanFactory 实例
XmlBeanFactory factory = new XmlBeanFactory(res);
```

如果应用里有多个属性配置文件，则应该采用 BeanFactory 的子接口 ApplicationContext 来创建 BeanFactory 的实例，ApplicationContext 通常使用如下两个实现类。

- FileSystemXmlApplicationContext：以指定路径的 XML 配置文件创建 ApplicationContext。
- ClassPathXmlApplicationContext：以 CLASSPATH 路径下的 XML 配置文件创建 ApplicationContext。

如果需要同时加载多个 XML 配置文件，可以采用如下方式：

```
//搜索 CLASSPATH 路径，以 CLASSPATH 路径下的 applicationContext.xml  
//service.xml 文件创建 ApplicationContext  
ClassPathXmlApplicationContext appContext =  
    new ClassPathXmlApplicationContext(  
        new String[] {"applicationContext.xml", "service.xml"});  
//事实上，ApplicationContext 是 BeanFactory 的子接口，支持强制类型转换  
BeanFactory factory = (BeanFactory) appContext;
```

当然也可支持从指定路径来搜索特定文件加载：

```
//指定路径下的 applicationContext.xml, service.xml 文件创建 ApplicationContext  
FileSystemXmlApplicationContext appContext =  
    new FileSystemXmlApplicationContext(  
        new String[] {"applicationContext.xml", "service.xml"});  
//事实上，ApplicationContext 是 BeanFactory 的子接口，支持强制类型转换  
BeanFactory factory = (BeanFactory) appContext;
```

下面是 Spring 最简单的配置文件：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->  
<?xml version="1.0" encoding="gb2312"?>  
<!-- 指定 Spring 配置文件的 dtd-->  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<!-- beans 元素是 Spring 配置文件的根元素，所有的 Spring 的配置文件都应该按如下结构书写-->  
<beans>  
  
</beans>
```

在 Spring 的 DTD 部分中，详细规定了 Spring 配置文件里的合法元素，各元素出现的先后顺序和各元素中子元素的合法属性。

## 5.5.2 bean 的基本定义

<beans/>元素是 Spring 配置文件的根元素，<bean>元素是<beans/>元素的子元素，<beans/>元素可以包含多个<bean/>元素，<bean/>子元素定义一个 bean，每个 bean 是接受 Spring 容器里的 Java 实例。

在定义 bean 时，通常必须指定以下两个属性：

- **id:** id 属性是确定该 bean 的唯一标识符，容器对 bean 管理、访问及该 bean 的依赖关系，都通过该属性完成。bean 的 id 属性在 Spring 容器中是唯一的。
- **class:** class 属性指定该 bean 的具体实现类，这里不能是接口。通常情况下，Spring 会直接使用 new 关键字创建该 bean 的实例，因此，这里必须提供 bean 实现类的类名。

下面给出了包含两个 bean 定义的简单配置文件：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->  
<?xml version="1.0" encoding="gb2312"?>
```

```

<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- beans 元素是 Spring 配置文件的根元素，所有的 Spring 的配置文件都应按如下结构书写-->
<beans>
    <!-- 定义第一个 Java 实例 bean1，该 Java 实例对应的实现类为 lee.Test1-->
    <bean id="bean1" class="lee.Test1"/>
    <!-- 定义第二个 Java 实例 bean2，该 Java 实例对应的实现类为 lee.Test2-->
    <bean id="bean2" class="lee.Test2"/>
</beans>

```

在 Spring 容器集中管理 bean 的实例化时，bean 实例可以通过 BeanFactory 的 getBean(String beanid)方法得到。此时，BeanFactory 将变成简单工厂模式里的工厂，程序只需要获取 BeanFactory 引用，即可获得 Spring 容器管理全部实例的引用，从而使程序不需要与具体实例的实现过程耦合。在大部分 J2EE 应用中，当应用启动时，会自动创建 Spring 容器实例，组件之间直接以依赖注入的方式耦合，甚至无须访问 Spring 容器。

### 5.5.3 定义 Bean 的行为方式

在 Spring1.2 版本中，bean 在 Spring 的容器中有两种基本行为。

- singleton：单态。
- non-singleton 或 prototype：原型。

如果一个 bean 被设置成 non-singleton 行为，当程序每次请求该 id 的 bean 时，Spring 都会新建一个 bean 实例，然后返回给程序。在这种情况下，Spring 容器仅仅使用 new 关键字创建 bean 实例，一旦创建成功，容器不再跟踪实例，也不会维护 bean 实例的状态。

通常要求将 Web 应用的控制器 bean 配置成 non-singleton 行为。因为，每次 HttpServletRequest 都需要系统启动一个新 Action 来处理用户请求。

如果一个 bean 被设置成 singleton 时，整个 Spring 容器里只有一个共享实例存在，程序每次请求该 id 的 bean 时，Spring 都会返回该 bean 的共享实例。该容器负责跟踪单态 bean 实例的状态，维护 bean 实例的生命周期。

如果不指定 bean 的基本行为，Spring 默认使用 singleton 行为。在创建 Java 实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此，non-singleton 行为的 bean 创建、销毁时代价比较大。而 singleton 行为的 bean 实例成功后，可以重复使用。因此，应尽量避免将 bean 设置成 non-singleton 行为。

关于自定义 bean 的生命周期行为，请参看 5.7.2 节。设置 bean 的基本行为，是通过 singleton 属性来指定，singleton 属性只接受 true 或 false 值。例如在下面配置文件中配置 singleton 和 non-singleton：

```

<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

```

```

"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- beans 元素是 Spring 配置文件的根元素，所有的 Spring 的配置文件都应该按如下结构书写-->
<beans>
    <!-- 定义第一个 Java 实例 bean1，该 Java 实例对应的实现类为 lee.Person-->
    <bean id="p1" class="lee.Person"/>
    <!-- 定义第二个 Java 实例 bean2，该 Java 实例对应的实现类为 lee.Person，指定该 bean
        为 non-singleton -->
    <bean id="p2" class="lee.Person" singleton="false"/>
</beans>

```

主程序通过如下代码来测试两个 bean 的区别：

```

public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        //搜索 CLASSPATH 路径，以 CLASSPATH 路径下的 beans.xml 文件创建 Resource 对象
        ClassPathResource res = new ClassPathResource("beans.xml");
        //以 Resource 对象为参数，创建 BeanFactory 实例
        XmlBeanFactory beanFactory = new XmlBeanFactory(res);
        //判断两次请求 singleton 行为的 bean 实例是否相等
        System.out.println(beanFactory.getBean("p1") == beanFactory.getBean("p1"));
        //判断两次请求 non-singleton 行为的 bean 实例是否相等
        System.out.println(beanFactory.getBean("p2") == beanFactory.getBean("p2"));
    }
}

```

程序执行结果如下：

```

true
false

```

对于 singleton 行为的 bean，每次请求该 id 的 bean 时，都将返回同一个共享实例，因而两次获取的 bean 实例完全相同；但对 non-singleton 行为的 bean，每次请求该 id 的 bean 时都将产生新的实例，因此两次请求获得 bean 实例不相同。

## 5.5.4 深入理解 bean

Spring 容器对 bean 没有特殊要求，甚至不要求该 bean 像标准的 JavaBean——必须为每个属性提供对应的 getter 和 setter 方法。Spring 中的 bean 是 Java 实例与 Java 组件。而传统 Java 应用中的 bean 通常作为 model，用来封装值对象在各层之间的传递。

Spring 中的 bean 比 JavaBean 的功能要强大，用法也更复杂。当然，传统 JavaBean 也可作为普通的 Spring bean，可接受 Spring 管理。下面的代码演示了 Spring 的 bean 实例，该 bean 实例是数据源，提供数据库连接：

```

<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

```

```

<!-- beans 元素是 Spring 配置文件的根元素，所有的 Spring 的配置文件都应按如下结构书写-->
<beans>
    <!-- 配置 id 为 dataSource 的 bean，该 bean 是个数据源实例-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!-- 确定数据源的驱动 -->
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <!-- 确定连接数据库的 url -->
        <property name="url">
            <!-- j2ee 是需要连接的数据库名 -->
            <value>jdbc:mysql://localhost:3306/j2ee</value>
        </property>
        <!-- 确定连接数据库的用户名 -->
        <property name="username">
            <!-- root 是连接数据库的用户名 -->
            <value>root</value>
        </property>
        <!-- 确定连接数据库的密码 -->
        <property name="password">
            <!-- pass 是连接数据库的密码 -->
            <value>pass</value>
        </property>
    </bean>
</beans>

```

主程序部分由 BeanFactory 来获取该 bean 的实例，获取实例时使用 bean 的唯一标识符：id 属性。该属性是 bean 实例在容器中的访问点。

下面是主程序部分：

```

public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        //实例化 Spring 容器。Spring 容器负责实例化 bean
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //通过 bean id 获取 bean 实例，并强制类型转换为 DataSource
        DataSource ds = (DataSource)ctx.getBean("dataSource");
        //通过 DataSource 来获取数据库连接
        Connection conn = ds.getConnection();
        //通过数据库连接获取 Statement
        java.sql.Statement stmt = conn.createStatement();
        //使用 statement 执行 sql 语句
        stmt.execute("insert into mytable values('wddaa2')");
        //清理资源，回收数据库连接资源
        if (stmt != null)stmt.close();
        if (conn != null)conn.close();
    }
}

```

从该实例可以看出，Spring 的 bean 远远超出值对象的 JavaBean 范畴，此时 bean 可以代表应用中的任何组件及任何资源实例。

虽然 Spring 对 bean 没有特殊要求，但笔者还是建议在 Spring 中的 bean 应满足如下几个原则：

- 每个 bean 实现类都应提供无参数的构造器。
- 接受构造注入的 bean，则应提供对应的构造函数。
- 接受设值注入的 bean，则应提供对应的 setter 方法，并不强制要求提供对应的 getter 方法。

传统 JavaBean 和 Spring 中 bean 存在如下区别。

- 用处不同：传统 JavaBean 更多地作为值对象传递参数，而 Spring 中的 bean 用处几乎无所不在，任何应用组件都可被称为 bean。
- 写法不同：传统 JavaBean 作为值对象，要求每个属性都提供 getter 和 setter 方法；但 Spring 中的 bean 只需为接受设值注入的属性提供 setter 方法。
- 生命周期不同：传统 JavaBean 作为值对象传递，不接收任何容器管理其生命周期；Spring 中的 bean 由 Spring 管理其生命周期行为。

## 5.5.5 创建 bean 实例

大多数情况下，BeanFactory 可直接调用构造函数来创建一个 bean，并以 class 属性确定 bean 实例的实现类。因此，bean 元素的 class 属性通常是必需的，但这并不是创建 bean 的唯一方法。

创建 bean 通常有如下方法：

- 调用构造器创建一个 bean 实例。
- BeanFactory 调用某个类的静态工厂方法创建 bean。
- BeanFactory 调用实例工厂方法创建 bean。

### 1. 调用构造函数 “new” 一个 bean 实例

通过“new”关键字创建 bean 实例是最常见的情形，如果采用设值注入的方式，则要求该类提供无参数的构造器。在这种情况下，class 元素是必需的（除非采用继承），class 属性的值就是 bean 实例的实现类。

然后，BeanFactory 将调用该构造器来创建 bean 实例，该实例是个默认实例。所有的属性执行默认初始化。

接下来，BeanFactory 会根据配置文件来决定依赖关系：首先实例化依赖的 bean；然后为 bean 注入依赖关系；最后将一个完整的 bean 实例返回给程序。此时该 bean 实例的所有属性，已经由 Spring 容器完成了初始化。下面是用调用构造函数“new”一个 bean 的实例。

调用者 bean 的接口和实现类如下：

```
//定义 Person 接口
public interface Person
{
    //Person 接口里定义一个使用斧子的方法
    public void useAxe();
}
```

## Person 的实现类

```
//Chinese 实现 Person 接口
public class Chinese implements Person
{
    //面向 Axe 接口编程，而不是具体的实现类
    private Axe axe;
    //默认的构造器
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例... ");
    }
    //设值注入所需的 setter 方法
    public void setAxe(Axe axe)
    {
        System.out.println("Spring 执行依赖关系注入... ");
        this.axe = axe;
    }
    //实现 Person 接口的 useAxe 方法
    public void useAxe()
    {
        System.out.println(axe.chop());
    }
}
```

下面给出 Person 接口依赖 bean 的接口和实现类：

```
//定义 Axe 接口
public interface Axe
{
    //Axe 接口里有个砍的方法
    public void chop();
}
```

Axe 的实现类 SteelAxe：

```
//SteelAxe 类实现 Axe 接口
public class SteelAxe implements Axe
{
    //默认构造器
    public SteelAxe()
    {
        System.out.println("Spring 实例化依赖 bean: SteelAxe 实例... ");
    }
    //实现 Axe 接口的 chop 方法
    public String chop()
    {
        return "钢斧砍柴真快";
    }
}
```

Spring 的配置文件如下：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
```

```

<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 定义第一个 bean, 该 bean 的 id 是 chinese, class 指定该 bean 实例的实现类 -->
    <bean id="chinese" class="lee.Chinese">
        <!-- property 元素用来指定需要容器注入的属性, axe 属性需要容器注入
        此处是设值注入, 因此 Chinese 类必须拥有 setAxe 方法 -->
        <property name="axe">
            <!-- 此处将另一个 bean 的引用注入 chinese bean -->
            <ref local="steelAxe"/>
        </property>
    </bean>
    <!-- 定义 steelAxe bean -->
    <bean id="steelAxe" class="lee.SteelAxe"/>
</beans>

```

主程序如下：

```

public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        // 在当前路径下搜索 bean.xml 文件, 实例化文件输入流
        InputStream is = new FileInputStream("bean.xml");
        // 以指定的文件输入流 is, 创建 Resource 对象
        InputStreamResource isr = new InputStreamResource(is);
        // 以 Resource 对象作为参数, 创建 BeanFactory 的实例
        XmlBeanFactory factory = new XmlBeanFactory(isr);
        System.out.println("程序已经实例化 BeanFactory...");
        Person p = (Person)factory.getBean("chinese");
        System.out.println("程序中已经完成了 chinese bean 的实例化...");
        p.useAxe();
    }
}

```

执行结果如下：

```

程序已经实例化 BeanFactory
Spring 实例化主调 bean: Chinese 实例...
Spring 实例化依赖 bean: SteelAxe 实例...
Spring 执行依赖关系注入...
程序中已经完成了 chinese bean 的实例化...
钢斧砍柴真快

```

执行结果清楚地反映了执行过程：

- (1) 创建 BeanFactory 实例。
- (2) 调用 Chinese 类的默认构造器创建默认实例。
- (3) 根据配置文件注入依赖关系：先实例化依赖 bean，然后将依赖 bean 注入。
- (4) 返回一个完整的 JavaBean 实例。

## 2. 使用静态工厂方法创建 bean

使用静态工厂方法创建 bean 实例时，class 属性也是必需的，但此时 class 属性并不是该实例的实现类，而是静态工厂类。由于 Spring 需要知道由哪个静态工厂方法来创建 bean 实例，因此使用 factory-method 属性来确定静态工厂方法名，在之后的过程中，Spring 的处理步骤与采用其他方法的创建完全一样。

下面通过 factory-method 指定的方法来创建 bean。注意：这个 bean 定义并没有指定返回对象的类型，只指定静态工厂类。该方法必须是静态的，如果静态工厂方法需要参数，则使用<constructor-arg>元素将其导入。

下面是 Being 接口：

```
public interface Being
{
    //接口定义 testBeing 方法
    public void testBeing();
}
```

下面是接口的两个实现类。

Dog 实现 Being 接口：

```
public class Dog implements Being
{
    private String msg;
    //依赖注入时必需的 setter 方法
    public void setMsg(String msg)
    {
        this.msg = msg;
    }
    //实现接口必须实现的 testBeing 方法
    public void testBeing()
    {
        System.out.println(msg + " 狗爱啃骨头");
    }
}
```

Cat 实现 Being 接口：

```
public class Cat implements Being
{
    private String msg;
    //依赖注入时必需的 setter 方法
    public void setMsg(String msg)
    {
        this.msg = msg;
    }
    //实现接口必须实现的 testBeing 方法
    public void testBeing()
    {
        System.out.println(msg + " 猫喜欢吃老鼠");
    }
}
```

下面的工厂包含静态方法，其静态方法可返回 Being 实例：

```
public class BeingFactory
{
    /**
     * 获取 Being 实例的静态工厂方法
     * param arg 静态工厂方法根据该参数决定返回 Being 的哪个实例
     */
    public static Being getBeing(String arg)
```

```

    {
        // 调用此静态方法的参数为 dog，则返回 Dog 实例
        if (arg.equalsIgnoreCase("dog"))
        {
            return new Dog();
        }
        // 否则返回 Cat 实例
        else
        {
            return new Cat();
        }
    }
}

```

下面是使用静态工厂方法创建 bean 实例的配置文件：

```

<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 此处的 class 元素并非 dog 的实现类，而是产生 dog 的静态工厂类，
        采用静态工厂方法创建 bean 实例，必须使用 factory-method 指定静态工厂方法 -->
    <bean id="dog" class="lee.BeingFactory" factory-method="getBeing">
        <!-- 调用静态工厂方法时，传入的参数使用
            constructor-arg 元素指定 -->
        <constructor-arg>
            <value>dog</value>
        </constructor-arg>
        <!-- property 用于确定普通接受依赖注入的属性 -->
        <property name="msg">
            <value>我是狗</value>
        </property>
    </bean>
    <!-- 此处的 class 元素并非 cat 的实现类，而是产生 dog 的静态工厂类，
        采用静态工厂方法创建 bean 实例，必须使用 factory-method 指定静态工厂方法 -->
    <bean id="cat" class="lee.BeingFactory" factory-method="getBeing">
        <!-- 调用静态工厂方法时，传入的参数使用
            constructor-arg 元素指定 -->
        <constructor-arg>
            <value>cat</value>
        </constructor-arg>
        <!-- property 用于确定普通接受依赖注入的属性 -->
        <property name="msg">
            <value>我是猫</value>
        </property>
    </bean>
</beans>

```

主程序部分如下：

```

public class SpringTest
{
    public static void main(String[] args) throws Exception
    {
        // 在当前路径下搜索 bean.xml 文件，实例化文件输入流
        InputStream is = new FileInputStream("bean.xml");
    }
}

```

```

//以指定的文件输入流 is, 创建 Resource 对象
InputStreamResource isr = new InputStreamResource(is);
//以 Resource 对象作为参数, 创建 BeanFactory 的实例
XmlBeanFactory factory = new XmlBeanFactory(isr);
System.out.println("程序已经实例化 BeanFactory...");
Being b1 = (Being)factory.getBean("dog");
b1.testBeing();
Being b2 = (Being)factory.getBean("cat");
b2.testBeing();
}
}

```

在使用静态工厂方法创建实例时，必须提供工厂类，其工厂类应包含产生实例的静态工厂方法。通过静态工厂方法创建实例时，必须要改变配置文件，主要有如下改变：

- class 属性的值不再是 bean 的实现类，而是静态工厂类。
- 必需使用 factory-method 属性确定产生实例的静态工厂方法。
- 如果静态工厂方法需要参数，则使用<constructor-arg>元素确定静态工厂方法。

### 3. 调用实例工厂方法创建 bean

实例工厂方法必须提供工厂实例，因此必须在配置文件中配置工厂实例，而 bean 元素无需 class 属性。因为 BeanFactory 不再直接实例化该 bean，仅仅是执行工厂的方法，负责生成 bean 实例。

采用实例工厂方法创建 bean 的配置需要如下两个属性。

- factory-bean：该属性的值为工厂 bean 的 id。
- factory-method：该方法负责生成 bean 实例。

与静态工厂方法相似，如果需要在调用工厂方法时传入参数，则使用<constructor-arg>元素来确定参数值。下面是 Person 接口：

```

//Person 接口定义
public interface Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */
    public String sayHello(String name);
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name);
}

```

该接口定义了 Person 的规范，并使该接口拥有两个方法：能打招呼，能告别。其实现的方法如下：

```

//American 类实现 Person 接口
public class American implements Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */

```

```

    */
    public String sayHello(String name)
    {
        return name + ",Hello";
    }
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name)
    {
        return name + ",Good Bye";
    }
}

```

下面是实现 Person 接口的另一个类：Chinese

```

public class Chinese implements Person
{
    /**
     * @param name 对 name 打招呼
     * @return 打招呼的字符串
     */
    public String sayHello(String name)
    {
        return name + ", 您好";
    }
    /**
     * @param name 对 name 告别
     * @return 告别的字符串
     */
    public String sayGoodBye(String name)
    {
        return name + ", 下次再见";
    }
}

```

Person 工厂的代码如下：

```

public class PersonFactory
{
    /**
     * 获得 Person 实例的实例工厂方法
     * @ param ethnic 调用该实例工厂方法传入的参数
     * @ return 返回 Person 实例
     */
    public Person getPerson(String ethnic)
    {
        //根据参数返回 Person 接口的实例
        if (ethnic.equalsIgnoreCase("chin"))
        {
            return new Chinese();
        }
        else
        {
            return new American();
        }
    }
}

```

可以看出，此代码与简单工厂模式的代码非常相似。此时 Spring 不负责创建 bean 实例，而是产生 bean 工厂实例。这是抽象工厂模式：Spring 容器负责生成 bean 实例，该 bean 实例是其他实例的工厂，负责产生程序需要的 bean 实例。配置文件如下：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd"> .
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置工厂 bean，该 bean 负责产生其他 bean 实例 -->
    <bean id="personFactory" class="lee.PersonFactory"/>
    <!-- 采用实例工厂创建 bean 实例，需要指定工厂 bean 的 id 属性
        该 id 属性对应另一个 bean，使用 factory-method 属性确定产生
        bean 的实例工厂方法 -->
    <bean id="chinese" factory-bean="personFactory" factory-method="getPerson">
        <!-- 调用实例工厂方法时，传入的参数通过 constructor-arg 元素指定 -->
        <constructor-arg>
            <value>chin</value>
        </constructor-arg>
    </bean>
    <!-- 采用实例工厂创建 bean 实例，需要指定工厂 bean 的 id 属性
        该 id 属性对应另一个 bean，使用 factory-method 属性确定产生
        bean 的实例工厂方法 -->
    <bean id="american" factory-bean="personFactory" factory-method="getPerson">
        <!-- 调用实例工厂方法时，传入的参数使用 constructor-arg 元素指定 -->
        <constructor-arg>
            <value>ame</value>
        </constructor-arg>
    </bean>
</beans>
```

调用实例工厂方法创建 bean，与调用静态工厂方法创建 bean 的用法基本相似。区别如下：

- 调用实例工厂方法创建 bean 时，必须将实例工厂配置成 bean 实例。而静态工厂方法则无须配置工厂 bean。
- 调用实例工厂方法创建 bean 时，必须使用 factory-bean 属性来确定工厂 bean。而静态工厂方法则使用 class 元素确定静态工厂类。

其相同之处如下：

- 都需使用 factory-method 属性指定产生 bean 实例的工厂方法。
- 工厂方法如果需要参数，都使用 constructor-arg 属性确定参数值。
- 其他依赖注入属性，都使用 property 元素确定参数值。

## 5.6 依赖关系配置

应用中的组件不可能单独存在，总需要调用其他组件。一个系统也不可能只有一个

类实例，如果实例 A 调用实例 B 的方法，可称为 A 依赖于 B。在传统的系统中，如果 A 依赖于 B，则 A 负责创建 B，或者定位 B 工厂，通过 B 工厂获得 B 实例。

而依赖注入却与此相反，如果 A 依赖于 B，则 B 实例不再由 A 负责生成，而由容器负责生成，并注入给 A 实例，因此称为依赖注入，也称为控制反转。

通过使用控制反转，使代码变得非常清晰。

## 5.6.1 配置依赖

根据注入方式的不同，bean 的依赖注入通常表现为如下两种形式。

- 属性：通过 property 属性来指定对应的设值注入。
- 构造器参数：通过 constructor-arg 属性来指定对应的构造注入。

不管是属性，还是构造器参数，都视为 bean 的依赖，接受 Spring 容器管理。依赖关系的值要么是一个确定的值，要么是 Spring 容器中其他 bean 的引用。通常也将被依赖的 bean 称为合作者。在 Spring 在实例化 BeanFactory 时，通常会校验 BeanFactory 中每一个 Bean 的配置。这些校验包括：

- bean 引用的合作者指向一个合法的 bean。
- 对于被设置为 pre-instantiated 的 bean 的 singleton 行为，Spring 会在创建 BeanFactory 时，同时实例化 bean。实例化 bean 时，也会将它所依赖的 bean 一起实例化。

此外，BeanFactory 与 ApplicationContext 实例化容器中 bean 的时机不同：前者在程序需要 bean 实例时才创建 bean；而后者在加载 ApplicationContext 实例时，会自动实例化容器中的全部 bean。

因为 BeanFactory 创建时不再创建 bean 实例，所以有可能正确地创建了 BeanFactory 实例，而在稍后请求 bean 实例时，在创建 bean 或者它的依赖时会出现错误，致使 BeanFactory 还会抛出一个异常。

配置错误的延迟出现，会给系统引入不安全因素。而 ApplicationContext 则默认预实例化 singleton bean。ApplicationContext 实例化过程比 BeanFactory 时间和内存占用率大，但可以在 ApplicationContext 创建时就检验出配置错误。

当然，可以通过设置 singleton bean 的 lazy-load 属性为“true”，来改变 ApplicationContext 的默认行为，使 bean 将不会随 ApplicationContext 启动而实例化。bean 的依赖通常可以接受如下元素指定值：

- value。
- ref。
- bean。
- list, set, map, 以及 props。

## 1. value 元素

value 元素用于确定字符串参数。XML 文档解析器解析以 String 解析出的这些数据，然后将这些参数由 PropertyEditors 完成类型转换（从 java.lang.String 类型转换为所需的参数类型）。如果目标类型是基本数据类型，通常都可以正确转换。

下面代码演示了用 value 元素确定属性值：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="exampleBean" class="lee.ExampleBean">
        <!-- 确定整型属性值 -->
        <property name="integerProperty">
            <value>1</value>
        </property>
        <!-- 确定双精度型属性值 -->
        <property name="doubleProperty">
            <value>2.3</value>
        </property>
    </bean>
</beans>
```

下面是该 bean 的实现类：

```
public class ExampleBean
{
    private int integerProperty;
    private double doubleProperty;
    /**
     * integerProperty 的 setter 方法
     * @ param i integerProperty 属性的设置值
     */
    public void setIntegerProperty(int i)
    {
        this.integerProperty = i;
    }
    /**
     * doubleProperty 的 setter 方法
     * @ param d doubleProperty 属性的设置值
     */
    public void setDoubleProperty(double d)
    {
        this.doubleProperty = d;
    }
}
```

本测试的主程序与之前的主程序相差不大，此处不再赘述。执行的结果是：

```
1
2.3
```

value 元素主要用于传入字符串参数和基本数据类型参数，也可传入合作者 bean，

但不推荐使用 value 元素。

value 元素的值可以通过<null/>元素指定为空。

代码如下：

```
<bean class="ExampleBean">
    <property name="email"><value></value></property>
</bean>
```

与采用如下配置文件的效果相同：

```
<bean class="ExampleBean">
    <property name="email"><null/></property>
</bean>
```

## 2. ref 元素

如果需要为 bean 注入的属性是容器中的某个 bean 实例，推荐使用 ref 元素。因为 ref 元素可防止出现引用错误，用于设置属性值为容器中其他 bean。通常，对属性值为容器中其他 bean 时，推荐采用 ref 元素指定，而不是 value 元素。看下面的配置代码：

```
<bean id="steelAxe" class="lee.SteelAxe"/>
<bean id="chinese" class="lee.Chinese">
    <property name="axe">
        <!-- 引用容器中另一个 bean -->
        <ref local="steelAxe"/>
    </property>
</bean>
```

与下面的配置代码效果完全一样：

```
<bean id="steelAxe" class="lee.SteelAxe"/>
<bean id="chinese" class="lee.Chinese">
    <property name="axe">
        <!-- 直接通过 value 元素指定依赖 bean -->
        <value>steelAxe</value>
    </property>
</bean>
```

第一种形式比第二种形式优越之处在于：使用 ref 标记，可让 Spring 在部署时验证依赖的 bean 是否真正存在；在第二种形式中，steelAxe 属性值仅在创建 bean 实例时做验证，会导致错误的延迟出现。而且，第二种还有额外的类型转换开销。因此，如需要传入合作者 bean 的属性值，推荐采用 ref 元素指定。

ref 元素通常有两个属性：

- bean
- local

bean 用于指定不在同一个 XML 配置文件中的 bean，而 local 用于指定同一个 XML 配置文件中的其他 bean，并且 local 属性值只能是其他 bean 的 id 属性，让 Spring 在解析 XML 时，验证 bean 的名称。

注意：ref 使 value 更严格，并能防止出错；而 local 则使 bean 更严格，也能防止出错。

### 3. bean 元素

如果某个 bean 的依赖 bean 不想被 Spring 容器直接访问，则可以使用嵌套 bean。bean 元素用来定义嵌套 bean，嵌套 bean 只对嵌套它的外部 bean 有效，而 Spring 容器无法直接访问嵌套 bean，因此嵌套 bean 无需 id 属性。修改上面的配置文件，使之变成嵌套 bean 的形式，示例如下：

```
<bean id="chinese" class="lee.Chinese">
    <property name="axe">
        <!-- 属性为嵌套 bean，嵌套 bean 不能由 Spring 容器直接访问。
            因此，嵌套 bean 没有 id 属性。 -->
        <bean class="lee.SteelAxe"/>
    </property>
</bean>
```

嵌套 bean 的配置形式，保证嵌套 bean 不能被容器访问，提高了程序的内聚性。因此不用担心其他程序修改嵌套 bean。外部 bean 的用法与之前的用法完全一样，使用效果也没有区别。

### 4. list, set, map, 以及 props 元素

如果 bean 的属性是集合，则可以使用集合元素。list, set, map 和 props 元素分别用来设置类型为 List, Set, Map 和 Properties 的属性值，用来为 bean 注入集合值。看如下示例代码：

```
public class Chinese implements Person
{
    //下面是系列集合属性，分别表示此人的学校，成绩，健康和斧子
    private List schools = new ArrayList();
    private Map score = new HashMap();
    private Properties health = new Properties();
    private Set axes = new HashSet();
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例... ");
    }
    /**
     * schools 依赖注入必需的 setter 方法
     * @ 1 学校集合
     */
    public void setSchools(List l)
    {
        this.schools = l;
    }
    /**
     * score 依赖注入必需的 setter 方法
     * @ 成绩情况
     */
    public void setScore(Map m)
    {
        this.score = m;
    }
    /**
     * health 依赖注入必需的 setter 方法
     * @ 健康情况
     */
    public void setHealth(Properties p)
    {
        this.health = p;
    }
    /**
     * axes 依赖注入必需的 setter 方法
     * @ 斧子集合
     */
    public void setAxes(Set s)
    {
        this.axes = s;
    }
}
```

```
* health 依赖注入必需的 setter 方法
* @ 健康状况
*/
public void setHealth(Properties p)
{
    this.health = p;
}
/**
 * axes 依赖注入必需的 setter 方法
* @ s 斧子集合
*/
public void setAxes(Set s)
{
    this.axes = s;
}
/**
 * 测试实例属性的方法
*/
public void test()
{
    System.out.println(schools);
    System.out.println(score);
    System.out.println(health);
    System.out.println(axes);
}
}
```

下面是 Spring 的配置文件：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="stoneAxe" class="lee.StoneAxe"/>
    <!-- 定义 chinese bean -->
    <bean id="chinese" class="lee.Chinese">
        <property name="schools">
            <!-- 定义 List 属性，使用 list 元素-->
            <list>
                <!-- list 元素里使用 value 元素确定系列值 -->
                <value>小学</value>
                <value>中学</value>
                <value>大学</value>
            </list>
        </property>
        <property name="score">
            <!-- 定义 Map 属性，使用 map 元素-->
            <map>
                <!-- Map 属性必须是 key -value 对-->
                <entry key="数学">
                    <value>87</value>
                </entry>
                <entry key="英语">
                    <value>89</value>
                </entry>
                <entry key="语文">
```

```

        <value>82</value>
    </entry>
</map>
</property>
<!-- 定义 Properties 属性，使用 props 元素-->
<property name="health">
<props>
<!-- Properties 属性必须是 key -value 对-->
<prop key="血压">正常</prop>
<prop key="身高">175</prop>
</props>
</property>
<!-- 定义 Set 属性，使用 set 元素-->
<property name="axes">
<set>
<!-- set 元素里使用 value,bean,ref 等指定系列值 -->
<value>字符串斧子</value>
<!-- bean 用于指定嵌套 bean 作为属性值-->
<bean class="lee.SteelAxe"/>
<!-- 确定容器中另一个 bean 为属性值 -->
<ref local="stoneAxe"/>
</set>
</property>
</bean>
</beans>

```

如上配置文件所示：set 元素的值可以通过 value, bean, ref 确定。map 元素 entry 的值及 set 元素的值都可以使用如下元素。

- **value**: 确定基本数据类型值或字符串类型值。
- **ref**: 确定另一个 bean 为属性值。
- **bean**: 确定一个嵌套 bean 为属性值。
- **list, set, map, 以及 props**: 确定集合值为属性值。

从上面介绍的依赖关系可知，要么是 JavaBean 式的值，要么直接依赖于其他 bean。在实际的应用中，某个实例的属性可能是某个方法的返回值，类的 field 值，或者属性值。这种非常规的注入方式，Spring 同样提供支持。Spring 甚至支持将 bean 实例的属性值、方法返回值及 field 值直接赋给一个变量。

## 5.6.2 注入属性值

通过 **PropertyPathFactoryBean** 类，可注入某个实例的属性值。**PropertyPathFactoryBean** 用来获得目标 bean 的属性值。获得的值可注入给其他 bean，也可直接定义成新的 bean。

看如下配置文件：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
<!--以下定义了将要被引用的目标 bean-->

```

```

<bean id="person" class="lee.Person" singleton="false">
    <property name="age"><value>30</value></property>
    <property name="son">
        <!-- 嵌套 bean 定义 person bean 的依赖关系 -->
        <bean class="lee.Son">
            <property name="age"><value>11</value></property>
        </bean>
    </property>
</bean>
<!--如下定义 son2 的 bean，该 bean 的 age 属性不是直接注入，而是依赖于其他 bean 的属性值-->
<!--如下将会该 son2 这个 bean 传入 bean person 的 age 属性值-->
<bean id="son2" class="lee.Son">
    <property name="age">
        <!--以下是访问 bean 属性的方式，这样可以将 person 这个 bean 的 son 的 age 属性赋值给 son2 这个 bean 的 age 属性-->
        <bean id="person.son.age"
            class="org.springframework.beans.factory.config.
PropertyPathFactoryBean"/>
    </property>
</bean>
</beans>

```

主程序如下：

```

public class SpringTest
{
    public static void main(String[] args)
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //打印出 son2 的 age 属性值
        System.out.println("son2 的 age 属性值：" + 
            ((Son)ctx.getBean("son2")).getAge());
    }
}

```

执行结果如下：

```
[java] 系统获取 son2 的 age 属性值： 11
```

其中，son2 实例的 age 属性，来自于 person bean 的嵌套 bean 的 age 属性。

一个 bean 实例的属性值，不仅可以注入另一个 bean，还可将 bean 实例的属性值直接定义成 bean 实例，也是通过 PropertyPathFactoryBean 完成的。对上面的配置文件增加如下代码：

```

<!-- 定义 son1 bean，该 bean 直接来自于其他 bean 的属性-->
<bean id="son1" class="org.springframework.beans.factory.config.
PropertyPathFactoryBean">
    <!-- 确定目标 bean，表明 son1 bean 来自哪个 bean 的属性-->
    <property name="targetBeanName">
        <value>person</value>
    </property>
    <!-- 确定属性名，表明 son1 bean 来自哪个目标 bean 的属性-->
    <property name="propertyName">
        <value>son</value>
    </property>

```

```
</bean>
```

主程序部分增加如下的输出：

```
System.out.println("系统获取的 son1: " + ctx.getBean("son1"));
```

主程序部分直接输出 bean1，此输出语句的执行结果如下：

```
[java] 系统获取的 son1: lee.Son@25d2b2
```

使用 PropertyPathFactoryBean 必须指定以下两个属性。

- targetBeanName：用于指定目标 bean，确定获取哪个 bean 的属性值。
- propertyPath：用于指定属性，确定获取目标 bean 的哪个属性值，此处的属性可直接使用属性表达式。例如，如想获取 person bean 的 son 属性的 age 属性，可采用 son.age。

也可将基本数据类型的属性值定义成 bean 实例。在配置文件中再增加如下代码：

```
<!-- 将基本数据类型的属性值定义成 bean 实例-->
<bean id="theAge" class="org.springframework.beans.factory.config.
PropertyPathFactoryBean">
    <!-- 确定目标 bean，表明 theAge bean 来自哪个 bean 的属性-->
    <property name="targetBeanName">
        <value>person</value>
    </property>
    <!-- 确定属性名，表明 theAge bean 来自目标 bean 的哪个属性。
    此处的属性采用属性表达式-->
    <property name="propertyPath">
        <value>son.age</value>
    </property>
</bean>
```

主程序部分增加如下输出：

```
System.out.println("系统获取 theAge 的值: " + ctx.getBean("theAge"));
```

程序执行结果如下：

```
[java] 系统获取 theAge 的值: 11
```

目标 bean 既可以是容器中已有的 bean 实例，也可是嵌套的 bean 实例。因此，下面的定义也是有效的：

```
<!-- 将基本数据类型的属性值定义成 bean 实例-->
<bean id="theAge2" class="org.springframework.beans.factory.config.
PropertyPathFactoryBean">
    <!-- 确定目标 bean，表明 theAge bean 来自哪个 bean 的属性。
    此处采用嵌套 bean 定义目标 bean-->
    <property name="targetObject">
        <!-- 目标 bean 不是容器中的其他 bean，而是如下的嵌套 bean-->
        <bean class="lee.Person">
            <property name="age"><value>12</value></property>
        </bean>
    </property>
    <property name="propertyPath"><value>age</value></property>
</bean>
```

### 5.6.3 注入 field 值

通过 FieldRetrievingFactoryBean 类，可以完成 field 值的注入。FieldRetrievingFactoryBean 用来获得目标 bean 的 field 值。获得的值可注入给其他 bean，也可直接定义成新的 bean。看如下配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!--如下定义 son3 的 bean，该 bean 的 age 属性不是直接注入，而是依赖于某个类的 field
    值-->
    <bean id="son3" class="lee.Son">
        <property name="age">
            <!-- FieldRetrievingFactoryBean 用来获取目标类的 field 值。
                id 属性确定获取哪个类哪个 field 值-->
            <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
                class="org.springframework.beans.factory.config.
                    FieldRetrievingFactoryBean"/>
        </property>
    </bean>
</beans>
```

主程序如下：

```
public class SpringTest
{
    public static void main(String[] args)
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //获取 son3 bean 实例
        Son son3 = (Son)ctx.getBean("son3");
        //输出 son3 的 age 值
        System.out.println("系统获取 son3 的 age 属性值: " + son3.getAge());
    }
}
```

程序的执行结果如下：

```
[java] 系统获取 son3 的 age 属性值: 8
```

其中，son3 bean 的 age 值，与 java.sql.Connection 接口中的 TRANSACTION\_SERIALIZABLE 值相同。field 值也可定义成 bean 实例，只需在配置文件中增加如下代码：

```
<!-- 将 field 值定义成 bean 实例-->
<bean id="theAge3" class="org.springframework.beans.factory.config.
    FieldRetrievingFactoryBean">
    <!-- 确定该 bean 的值来自于 staticField-->
    <property name="staticField">
```

```

<!-- value 指定采用哪个类的哪个静态域值-->
<value>java.sql.Connection.TRANSACTION_SERIALIZABLE</value>
</property>
</bean>

```

主程序部分增加如下输出：

```
System.out.println("系统获取 theAge3 的值: " + ctx.getBean("theAge3"));
```

输出结果如下：

```
[java] 系统获取 theAge3 的值: 8
```

**注意：** 使用 FieldRetrievingFactoryBean 时，必须指定与相应类对应（或接口）的 field 值。

#### 5.6.4 注入方法返回值

通过 MethodInvokingFactoryBean 类，可注入方法返回值。MethodInvokingFactoryBean 用来获得某个方法的返回值，该方法既可以是静态方法，也可以是实例方法。该方法的返回值可以注入 bean 实例属性，也可以直接定义成 bean 实例。看如下配置文件：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!--以下定义目标 bean，后面的 bean 将使用该 bean 的方法返回值注入属性-->
    <bean id="valueGenerator" class="lee.ValueGenerator"/>

    <!--如下定义 son5 的 bean，该 bean 的 age 属性不是直接注入，而是依赖于其他 bean 的返
        回值-->
    <bean id="son5" class="lee.Son">
        <property name="age">
            <!-- 使用嵌套 bean 注入方法返回值-->
            <bean class="org.springframework.beans.factory.config.
MethodInvokingFactoryBean">
                <!-- targetObject 确定目标 bean，确定调用哪个 bean -->
                <property name="targetObject"><ref local="valueGenerator" /></property>
                <!-- targetMethod 确定目标方法，确定调用目标 bean 的哪个方法 -->
                <property name="targetMethod"><value>getValue</value></property>
            </bean>
        </property>
    </bean>
</beans>

```

下面给出 ValueGenerator 的代码部分，其中 ValueGenerator 类包含静态方法和实例方法，这两个方法都返回整型值，代码如下：

```
public class ValueGenerator
{

```

```

//测试用的实例方法
public int getValue()
{
    return 2;
}
//测试用静态方法
public static int getStaticValue()
{
    return 9;
}
}

```

主程序如下：

```

public class SpringTest
{
    public static void main(String[] args)
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //获取 son5 bean 实例
        Son son5 = (Son)ctx.getBean("son5")
        //输出 son5 的 age 值
        System.out.println("系统获取 son5 的 age 属性值: " + son5.getAge());
    }
}

```

执行结果如下：

```
[java] 系统获取 son5 的 age 属性值: 2
```

程序中 bean son5 的 age 属性值，来自于 ValueGenerator 类的实例方法返回值。使用 bean 实例的方法返回值注入，通过 MethodInvokingFactoryBean 完成，但必须指定以下两个属性。

- targetObject：确定目标 bean，该 bean 可以是容器中已有的 bean，也可嵌套 bean。
- targetMethod：确定目标方法，确定通过目标 bean 的哪个方法返回值注入。

如果使用静态方法返回值注入，则无须指定 targetObject，但须指定目标 class，指定目标 class 的属性通过 targetClass 属性。因此，使用静态方法注入时需要指定如下两个属性。

- targetClass：确定目标 class。
- targetMethod：确定目标方法，确定通过目标 bean 的哪个方法返回值注入。

在配置文件中增加如下代码：

```

<!--如下定义 son4 的 bean，该 bean 的 age 属性不是直接注入，而是依赖于其他 bean 的返回值-->
<bean id="son4" class="lee.Son">
    <property name="age">
        <!-- 使用嵌套 bean 注入方法返回值-->
        <bean class="org.springframework.beans.factory.config.
MethodInvokingFactoryBean">
            <!-- targetClass 确定目标类，确定调用哪个类-->
            <property name="targetClass"><value>lee.ValueGenerator</value>
        </property>
        <!-- targetMethod 确定目标方法，确定调用目标 class 的哪个方法。
        该方法必须是静态方法-->
        <property name="targetMethod"><value>getStaticValue</

```

```

    value></property>
</bean>
</property>
</bean>

```

主程序中增加如下输出：

```

//获取 son4 bean 实例
Son son4 = (Son)ctx.getBean("son4")
//输出 son4 的 age 值
System.out.println("系统获取 son4 的 age 属性值: " + son4.getAge());

```

程序执行结果如下：

```
[java] 系统获取 son4 的 age 属性值: 9
```

当然，也可以将方法返回值直接定义成 bean，在配置文件中增加如下代码：

```

<!-- 将静态方法返回值直接定义成 bean -->
<bean id="sysProps" class="org.springframework.beans.factory.config.
MethodInvokingFactoryBean">
    <!-- targetClass 确定目标类，确定调用哪个类-->
    <property name="targetClass"><value>java.lang.System</value></property>
    <!-- targetMethod 确定目标方法，确定调用目标 class 的哪个方法
        该方法必须是静态方法-->
    <property name="targetMethod"><value>getProperties</value></property>
</bean>

```

该配置文件将 `java.lang.System` 类的静态方法 `getProperties` 的返回值，直接定义为 bean 实例。然后再增加如下代码：

```

<!-- 将实例方法返回值直接定义成 bean -->
<bean id="javaVersion" class="org.springframework.beans.factory.config.
MethodInvokingFactoryBean">
    <!-- targetObject 确定目标 bean，确定调用哪个 bean -->
    <property name="targetObject"><ref local="sysProps"/></property>
    <!-- targetMethod 确定目标方法，确定调用目标 bean 的哪个方法 -->
    <property name="targetMethod"><value>getProperty</value></property>
    <!-- 调用目标方法时，传入的参数值 -->
    <property name="arguments">
        <list>
            <value>java.version</value>
        </list>
    </property>
</bean>

```

该配置文件将实例方法返回值直接定义成 bean。该实例是上面配置的 `sysProps` bean，实例方法是 `Property` 类的 `getProperty` 方法。在主程序中增加如下输出：

```
System.out.println("系统获取 Java 版本: " + ctx.getBean("javaVersion"));
```

输出结果如下：

```
[java] 系统获取 Java 版本: 1.4.2_04
```

因此，这种配置方式也可用于定义静态工厂方法来创建 bean 实例，或用实例工厂方

法来创建 bean 实例。配置示例如下：

```
<!-- 定义通过静态工厂方法创建的 bean 实例-->
<bean id="myBean" class="org.springframework.beans.factory.config.
MethodInvokingFactoryBean">
    <!-- staticMethod 属性指定静态工厂类，静态工厂方法-->
    <property name="staticMethod"><value>lee.MyClassFactory.getInstance</
value></property>
</bean>
```

对于这种情形，通常不推荐采用此配置方法，建议采用 factory-method 的方法来配置 bean。请参阅第 5.5.5 节中的创建 bean 实例。

## 5.6.5 强制初始化 bean

Spring 默认有个规则，总是先初始化主调 bean，然后再初始化依赖 bean。

在大多数情况下，bean 之间的依赖非常直接。Spring 容器在返回 bean 实例之前，就完成 bean 依赖关系的注入：假如 bean A 依赖于 bean B，当程序请求 bean A 时，Spring 容器会自动初始化 bean B，再将 bean B 注入 bean A；最后将具备完整依赖的 bean A 返回程序。

在极端的情况下，bean 之间的依赖不够直接。比如在某个类的初始化块中使用其他 bean 时，Spring 总是先初始化主调 bean，而执行初始化块时还没有实例化主调 bean，被依赖 bean 还没实例化。使用 depends-on 可以在初始化主调 bean 之前，强制一个或多个 bean 初始化。配置文件的代码如下：

```
<!-- 配置 beanOne, 该 bean 需要实例化之前，使用 manager bean
     使用 depends-on 强制 manager bean 在初始化 bean One 之前实例化-->
<bean id="beanOne" class="ExampleBean" depends-on="manager">
    <property name="manager"><ref local="manager"/></property>
</bean>
<bean id="manager" class="ManagerBean"/>
```

## 5.6.6 自动装配

Spring 能自动装配 bean 与 bean 之间的依赖关系，即无须使用 ref 显式指定依赖 bean，由 BeanFactory 检查 XML 配置文件内容，根据某种规则，为主调 bean 注入依赖关系。自动装配可作为某个 bean 的属性，因此可以指定单独 bean，使某些 bean 使用自动装配。自动装配可以减少配置文件的工作量，但降低了依赖关系的透明性和清晰性。

使用 bean 元素的 autowire 属性来配置自动装配，其 autowire 属性可以接受如下值。

- no：不使用自动装配。Bean 依赖必须通过 ref 元素定义，这是默认的配置，在较大的部署环境中不建议改变这个配置。
- byName：根据属性名自动装配。BeanFactory 查找容器中全部 bean，并找出其中 id 属性与属性同名的 bean。
- byType：根据属性类型自动装配。BeanFactory 查找容器全部 bean，如果恰好有

一个与依赖属性类型相同的 bean，就自动装配这个属性；如果有多个这样的 bean，就抛出一个致命异常；如果没有匹配的 bean，则什么都不会发生，因而属性不会被设置；如果需要无法自动装配时抛出异常，设置 dependency-check="objects"。

- **constructor:** 同 byType 类似，区别是用于构造注入的参数。如果在 BeanFactory 中，不是恰好有一个 bean 与构造器参数相同类型，则会产生一个致命的错误。
- **autodetect:** BeanFactory 根据 bean 内部结构，决定使用 constructor 或 byType。如果找到一个默认的构造函数，那么就会应用 byType。

### 1. byName 规则

byName 规则，指通过名字注入依赖关系。假如 bean A 的实现类包含 setter B 方法，而 Spring 的配置文件包含 id 为 b 的 bean。当 Spring 容器为 bean A 注入 b 实例时，如果容器中没有与名字匹配的 bean 时，则抛出异常。看如下配置文件：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
  <!-- 指定 chinese bean 使用自动装配，按名字装配 -->
  <bean id="chinese" class="lee.Chinese" autowire = "byName"/>
  <!-- 配置 gundog bean -->
  <bean id="gundog" class="lee.Gundog">
    <property name="name">
      <value>wangwang</value>
    </property>
  </bean>
</beans>
```

上面的配置文件，要求 Chinese 类中有如下方法：

```
/**
 * 依赖关系必需的 setter 方法
 * 因为需要通过名字自动装配，因此 setter 方法名必须是 set + bean 名。bean 名首字母大写
 * @ dog 设置的 dog 值
 */
public void setGunDog(Dog dog)
{
    this.dog = dog;
}
```

### 2. byType 规则

byType 规则，指根据类型匹配来注入依赖关系。假如 A 实例有 setB(B b)方法，而 Spring 配置文件中恰有一个类型 B 的 bean 实例，当容器为 A 注入类型匹配的 bean 实例时，如果容器中没有一个类型为 B 的实例，或有多于一个的 B 实例时，都将抛出异常。看如下配置文件：

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
```

```

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 指定 chinese bean 使用自动装配，按名字装配 -->
    <bean id="chinese" class="lee.Chinese" autowire = "byType"/>
    <!-- 配置 gundog bean -->
    <bean id="gundog" class="lee.Gundog">
        <property name="name">
            <value>wangwang</value>
        </property>
    </bean>
</beans>

```

上面的配置文件要求 Chinese 类中有如下方法：

```

/**
 * 依赖关系必需的 setter 方法
 * 因为使用按类型自动装配，要求 setter 方法的参数类型与容器的 bean 的类型相同。
 * 程序中的 GunDog 实现 Dog 接口
 * @ dog 设置的 dog 值
 */
public void setDog(Dog dog)
{
    this.dog = dog;
}

```

但如果出现如下配置文件：

```

<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 指定 chinese bean 使用自动装配，按名字装配 -->
    <bean id="chinese" class="lee.Chinese" autowire = "byType"/>
    <!-- 配置 gundog bean -->
    <bean id="gundog" class="lee.Gundog">
        <property name="name">
            <value>wangwang</value>
        </property>
    </bean>
    <!-- 配置 petdog bean Pegdog 也实现 Dog 接口 -->
    <bean id="petdog" class="lee.Petdog">
        <property name="name">
            <value>ohoh</value>
        </property>
    </bean>
</beans>

```

此时，Spring 将无法按类型自动装配，因为容器中有两个类型为 Dog 的 bean，Spring 无法确定应为 chinese bean 注入哪个 bean。剩下的两种自动装配大同小异，此处不再赘述。

当一个 bean 既使用自动装配依赖，又使用 ref 显式指定依赖时，则显式指定的依赖将覆盖自动装配。看如下配置文件：

```

<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 指定 chinese bean 使用自动装配，按名字装配 -->
    <bean id="chinese" class="lee.Chinese" autowire = "byName">
        <property name="Gundog">
            <ref local="petdog"/>
        </property>
        <!-- 配置 gundog bean -->
        <bean id="gundog" class="lee.Gundog">
            <property name="name">
                <value>wangwang</value>
            </property>
        </bean>
        <!-- 配置 petdog bean Pegdog 也实现 Dog 接口 -->
        <bean id="petdog" class="lee.Petdog">
            <property name="name">
                <value>ohoh</value>
            </property>
        </bean>
    </beans>

```

可以看出，即使 Chinese 类中有 setGundog(Dog dog)方法，Spring 依然注入 petdog 实例，而不注入 gundog 实例，显式的 ref 指定依赖覆盖自动装配指定的依赖。

**注意：**对于大型的应用，不建议使用自动装配。虽然使用自动装配可减少配置文件的工作量，但大大降低了依赖关系的清晰性和透明性。由于依赖关系的装配依赖于源文件的属性名，导致 bean 与 bean 之间的耦合降低到代码层次，不利于高层次解耦。

## 5.6.7 依赖检查

为了防止配置错误，或者其他出错可能，可使用 Spring 的依赖检查特性来判断 bean 与 bean 之间的依赖关系是否有效。

有效的依赖是：或者是 JavaBean 式的属性，在 bean 的定义中已设置了真实的值；或者通过自动装配提供有效依赖；或者通过 ref 注入了合作者 bean。但依赖检查可保证 bean 的属性得到正确设置。

当某个 bean 的特定属性并不需要设置值时，或者某些属性已有默认值，此时采用依赖检查就会出现错误，则该 bean 就不应该采用依赖检查。

幸好 Spring 可以允许每个 bean 都有自己的依赖检查行为，与自动装配功能一样，依赖检查作为 bean 的属性配置，因此可对每一个 bean 分别应用检查，或取消依赖检查，默认不应用依赖检查。

依赖检查可以以几种不同的模式处理，在配置文件中，通过 bean 元素的 dependency-check 属性来设置依赖检查，该属性有以下值。

- none：不进行依赖检查。没有指定值的 bean 属性（仅仅是没设值）。

- simple: 对基本类型和集合（除了合作者 bean）进行依赖检查。
- objects: 仅对合作者 bean 进行依赖检查。
- all: 对合作者 bean、基本类型和集合全部进行依赖检查。

对如下的 bean 类:

```
public class Chinese implements Person
{
    private Axe axe;
    private int age = 0;
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例..."); 
    }
    // 依赖注入 Axe 必需的 setter 方法
    public void setAxe(Axe axe)
    {
        this.axe = axe;
    }
    // 依赖注入 age 必需的 setter 方法
    public void setAge(int age)
    {
        this.age = age;
    }
    // 测试用方法
    public void useAxe()
    {
        System.out.println(axe.chop() + " 此人的年龄为: " + age);
    }
}
```

如采用下面配置文件:

```
<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="steelAxe" class="lee.SteelAxe"/>
    <!-- 定义 chinese bean，对其全部依赖检查 -->
    <bean id="chinese" class="lee.Chinese" dependency-check="all">
        <!-- 提供斧子的依赖注入 -->
        <property name="axe">
            <ref local="steelAxe"/>
        </property>
    </bean>
</beans>
```

则程序运行过程中出现如下异常:

```
Exception in thread "main" org.springframework.beans.factory.Unsatisfied
DependencyException: Error creating bean with name 'chinese' defined in resource loaded
through InputStream: Unsatisfied dependency expressed through bean property 'age': set
this property value or disable dependency checking for this bean
```

虽然 Chinese 类中的 age 已经有默认值，但配置 dependency-check="all"，要求所有的属性都必须提供正确的值。所以只需将上面的 dependency-check="all" 改成

dependency-check="objects", 此时程序即可正确执行。因为 BeanFactory 仅检查依赖 bean, 不检查简单值。

## 5.7 bean 的生命周期

Spring 不仅负责创建 bean 实例, 对于单态行为的 bean。Spring 还负责跟踪及管理 bean 的生命周期, 并能在 bean 创建之后, 销毁之前回调应用程序。

### 5.7.1 了解 bean 的生命周期

只有 singleton 行为的 bean 接受容器管理生命周期。对于 non-singleton 行为的 bean, Spring 容器仅仅是 “new” 的替代, 容器只负责创建。

non-singleton 行为的 bean 实例化后, 完全交给客户端代码管理, 容器不再跟踪其生命周期。每次客户端请求时, 都产生一个新的实例。Spring 容器不知道 non-singleton bean 什么时候销毁。

而 singleton 行为的 bean, 在每次客户端代码请求时都返回同一个共享实例。客户端代码不能控制 bean 的销毁, 而由容器控制 bean 的产生、销毁。容器可以在创建 bean 之后, 进行某些通用资源申请, 在销毁 bean 之前, 先回收某些资源。比如数据库连接。

### 5.7.2 定制 bean 的生命周期行为

对于 singleton bean, Spring 容器知道 bean 何时实例化结束, 何时销毁。Spring 可以管理实例化结束之后和销毁之前的行为。管理 bean 的生命周期行为主要有如下两个时机:

- 注入依赖关系之后。
- 即将销毁 bean 之前。

#### 1. 依赖关系注入之后的行为

Spring 提供了两种方式, 可在 bean 全部属性设置成功后执行特定行为。

- 使用 init-method 属性。
- 实现 InitializingBean 接口。

第一种方法: 指定 init-method 属性, 确定某个方法应在 bean 全部依赖关系设置结束后自动执行。使用这种方法无须将代码与 Spring 的接口耦合在一起, 其代码污染小。看如下示例代码:

```
public class Chinese implements Person
{
    private Axe axe;
    public Chinese()
    {
```

```

        System.out.println("Spring 实例化主调 bean: Chinese 实例...");  

    }  

    //依赖注入必需的 setter 方法  

    public void setAxe(Axe axe)  

    {  

        System.out.println("Spring 执行依赖关系注入...");  

        this.axe = axe;  

    }  

    //测试用初始化方法  

    public void init()  

    {  

        System.out.println("正在执行初始化方法...");  

    }
}

```

通过 `init-method` 属性，指定 `init` 方法应在 `bean` 全部属性设置结束后自动执行。该代码没有实现任何 Spring 的接口，只是增加一个普通的初始化方法，但依然是一个普通的 Java 文件，没有代码污染。

看下面的配置文件：

```

<!-- XML 文件头部分指定了 XML 文件的编码集-->  

<?xml version="1.0" encoding="gb2312"?>  

<!-- 指定 Spring 配置文件的 DTD-->  

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  

  "http://www.springframework.org/dtd/spring-beans.dtd">  

<!-- Spring 配置文件的根元素 -->  

<beans>  

    <bean id="steelAxe" class="lee.SteelAxe"/>  

    <!-- 配置 chinese bean，使用 init-method 属性指定 init 方法的执行时机 -->  

    <bean id="chinese" class="lee.Chinese" init-method="init">  

        <property name="axe">  

            <ref local="steelAxe"/>  

        </property>  

    </bean>  

</beans>

```

程序执行结果如下：

```

[java] 程序已经实例化 BeanFactory...  

[java] Spring 实例化主调 bean: Chinese 实例...  

[java] Spring 实例化依赖 bean: SteelAxe 实例...  

[java] Spring 执行依赖关系注入...  

[java] 正在执行初始化方法...  

[java] 程序中已经完成了 chinese bean 的实例化...  

[java] 钢斧砍柴真快

```

由此可看出，当 `Chinese` bean 完成依赖 `bean steelAxe` 的注入后，将自动执行 `init` 方法。也可采用第二种方法达到同样的效果，可让 `Chinese` 类实现 `InitializingBean` 接口，该接口提供一个方法：

```
void afterPropertiesSet() throws Exception;
```

实现该接口必须实现该方法，方法体就是依赖注入之后执行的方法。因此可将 `Chinese` 修改成如下形式：

```

public class Chinese implements Person, InitializingBean
{
    private Axe axe;
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例... ");
    }
    //依赖注入必需的 setter 方法
    public void setAxe(Axe axe)
    {
        System.out.println("Spring 执行依赖关系注入... ");
        this.axe = axe;
    }
    //测试用初始化方法
    public void afterPropertiesSet() throws Exception
    {
        System.out.println("正在执行初始化方法... ");
    }
}

```

对于实现 InitializingBean 接口的 bean，无须使用 init-method 属性来指定初始化方法。当注入依赖关系后，Spring 容器会自动调用 afterPropertiesSet 方法。因此，执行效果与采用 init-method 完全一样。但实现 InitializingBean 接口方法，是侵入式设计，因此不推荐采用。

**注意：**如果既采用 init-method 属性指定初始化方法，又采用实现 InitializingBean 接口来指定初始化方法。容器就会先执行 InitializingBean 接口中定义的方法，然后执行 init-method 属性指定的方法。

## 2. bean 销毁之前的行为

与初始化的方法相似，Spring 也提供了两种在 bean 销毁之前执行特定行为的方式。

- 使用 destroy-method。
- 实现 DisposableBean 接口。

第一种方法：destroy-method 属性确定某个方法在 bean 销毁之前自动执行。使用这种方法，无须将代码与 Spring 的接口耦合在一起，其代码污染小。看如下示例代码：

```

public class Chinese implements Person
{
    private Axe axe;
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例... ");
    }
    //依赖注入必需的 setter 方法
    public void setAxe(Axe axe)
    {
        System.out.println("Spring 执行依赖关系注入... ");
        this.axe = axe;
    }
    //测试用销毁方法
    public void close()
    {

```

```

        System.out.println("正在执行销毁前的资源回收方法...");  

    }  

}
}

```

通过 `destroy-method` 属性，可指定 `close` 方法在 `bean` 销毁之前自动执行。该代码没有实现任何 Spring 的接口，只是增加一个普通的销毁方法，依然是一个普通的 Java 文件，没有代码污染。

如下是配置文件：

```

<!-- XML 文件的文件头部分，指定了 XML 文件的编码集-->  

<?xml version="1.0" encoding="gb2312"?>  

<!-- 指定 Spring 配置文件的 dtd-->  

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  

   "http://www.springframework.org/dtd/spring-beans.dtd">  

<!-- Spring 配置文件的根元素 -->  

<beans>  

    <bean id="steelAxe" class="lee.SteelAxe"/>  

    <!-- 配置 chinese bean，使用 init-method 属性指定 init 方法的执行时机 -->  

    <bean id="chinese" class="lee.Chinese" destroy-method="close">  

        <property name="axe">  

            <ref local="steelAxe"/>  

        </property>  

    </bean>  

</beans>

```

也可采用第二种方法达到同样的效果，可让 `Chinese` 类实现 `DisposableBean` 接口，该接口提供一个方法：

```
void destroy() throws Exception;
```

实现该接口必须实现该方法，该方法就是依赖注入之后执行的方法。因此可将 `Chinese` 修改成如下形式：

```

public class Chinese implements Person, DisposableBean  

{  

    private Axe axe;  

    public Chinese()  

    {  

        System.out.println("Spring 实例化主调 bean: Chinese 实例...");  

    }  

    // 依赖注入必需的 setter 方法  

    public void setAxe(Axe axe)  

    {  

        System.out.println("Spring 执行依赖关系注入...");  

        this.axe = axe;  

    }  

    // 测试用销毁方法  

    public void destroy() throws Exception;  

    {  

        System.out.println("正在执行销毁前的资源回收方法...");  

    }
}

```

实现 `DisposableBean` 接口，则无须使用 `destroy-method` 属性来指定销毁之前的方法。由于在销毁 `bean` 实例之前，Spring 容器会自动调用 `destroy` 方法，因此执行效果与采用

`destroy-method` 完全一样。但实现 `DisposableBean` 接口污染了代码，是侵入式设计，因此不推荐采用。

注意：如果既采用 `destroy-method` 属性指定销毁之前的方法，又采用实现 `DisposableBean` 接口来指定销毁之前的方法，则容器就会先执行 `DisposableBean` 接口中定义的方法，然后执行 `destroy-method` 属性指定的方法。

### 5.7.3 协调不同步的 bean

Spring 容器中的 bean 有如下两种基本行为：

- singleton bean。
- non-singleton bean。

当两个 singleton bean 存在某种依赖时，或 non-singleton 依赖 singleton bean 时，仅通过属性定义依赖就足够了。但对 singleton bean 依赖于 non-singleton bean 时，singleton bean 只有一次初始化机会，其依赖关系的设置也在初始化时进行。而其依赖的 non-singleton bean 每次都有新实例，这将导致 singleton bean 的依赖不能得到即时更新，使 singleton bean 的依赖一直是最开始的 bean，即使 non-singleton bean 后来有了更多新的实例。

这样问题就产生了：当 singleton bean 依赖于 non-singleton bean 时，会产生不同步的现象。解决该问题有以下两种思路。

- 部分放弃依赖注入：当 singleton bean 每次需要 non-singleton bean 时，主动向容器请求新的 bean 实例，保证了每次产生的 bean 实例都是新的实例。
- 利用方法注入。

第一种方式显然不是一个好的做法，因为在代码中主动请求新的 bean 实例时，必然导致代码与 Spring API 耦合，造成严重代码污染。通常情形下，我们采用第二种做法——使用方法注入，通常使用 `lookup` 方法注入。

`lookup` 方法注入：指容器能够重写容器中 bean 的抽象或具体方法，并返回查找容器中其他 bean 的结果。被查找的 bean 通常是一个 non-singleton bean（尽管也可以是一个 singleton 的）。Spring 通过使用 CGLIB 库修改客户端的二进制码，从而实现上述的要求。看下面代码：

```
public class SteelAxe implements Axe
{
    //count 是个状态值，每次执行 chop 方法该值增加 1
    private int count = 0;
    public SteelAxe()
    {
        System.out.println("Spring 实例化依赖 bean: SteelAxe 实例...");
    }
    //测试用方法
    public String chop()
    {
        return "钢斧砍柴真快 " + ++count;
    }
}
```

上面的 SteelAxe 将被部署成 non-singleton bean，并被一个 singleton bean 所依赖。singleton bean 的代码如下：

```
public abstract class Chinese implements Person
{
    private Axe axe;
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例... ");
    }
    //方法注入所需的方法，该方法由 Spring 提供实现。
    public abstract Axe createAxe();
    //依赖注入必需的 setter 方法
    public void setAxe(Axe axe)
    {
        System.out.println("Spring 执行依赖关系注入... ");
        this.axe = axe;
    }
    //用于返回此 bean 的依赖 bean
    public Axe getAxe()
    {
        return axe;
    }
}
```

下面给出其配置文件：

```
<!-- XML 文件头部分指定了 XML 文件的编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置 non-singleton bean，该 bean 具有状态变量 count -->
    <bean id="steelAxe" class="lee.SteelAxe" singleton="false"/>
    <bean id="chinese" class="lee.Chinese" >
        <!-- 指定 lookup 方法注入，该方法用来保证每次产生新的 non-singleton bean,
            使用 bean 属性确定 lookup 方法的返回值-->
        <lookup-method name="createAxe" bean="steelAxe"/>
        <property name="axe">
            <ref local="steelAxe"/>
        </property>
    </bean>
</beans>
```

主程序分别使用两种方法来产生依赖 bean：

```
public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        InputStream is = new FileInputStream("bean.xml");
        //以指定的文件输入流 is，创建 Resource 对象
        InputStreamResource isr = new InputStreamResource(is);
        //以 Resource 对象作为参数，创建 BeanFactory 的实例
        XmlBeanFactory factory = new XmlBeanFactory(isr);
        Person p = (Person)factory.getBean("chinese");
```

```

//下面方法直接请求 Person 的依赖 bean, 因为只有一次注入, 因此每次返回的依赖 bean
//都是相同实例
Axe axe1 = p.getAxe();
Axe axe2 = p.getAxe();
System.out.println("没有采用 Lookup 方法注入: ");
System.out.println("Random 的两个实例指向同一个引用: " + (axe1 == axe2));
System.out.println(axe1.chop());
System.out.println(axe2.chop());
//调用 lookup 注入, 则保证每次产生新的 non-singleton bean 实例。
Axe axe3 = p.createAxe();
Axe axe4 = p.createAxe();
System.out.println("采用 Lookup 方法注入之后: ");
System.out.println("Random 的两个实例指向同一个引用: " + (axe3 == axe4));
System.out.println(axe3.chop());
System.out.println(axe4.chop());
}
}

```

程序执行的结果如下：

```

[java] 没有采用 Lookup 方法注入:
[java] Axe 的两个实例指向同一个引用: true
[java] 钢斧砍柴真快 1
[java] 钢斧砍柴真快 2
[java] Spring 实例化依赖 bean: SteelAxe 实例...
[java] Spring 实例化依赖 bean: SteelAxe 实例...
[java] 采用 Lookup 方法注入之后:
[java] Axe 的两个实例指向同一个引用: false
[java] 钢斧砍柴真快 1
[java] 钢斧砍柴真快 1

```

执行结果表明：采用 lookup 方法注入后，产生了两个新的 Axe 实例。可以看到两个 Axe 的状态值都是新值，并且两个 Axe 实例不同。

**注意：**要保证 lookup 方法注入每次产生新的 bean 实例，必须将目标 bean（上例就是 steelAxe）部署成 non-singleton，否则由于容器中只有一个目标 bean 实例，即使采用 lookup 方法注入，每次依然返回同一个 bean 实例；另外，lookup 方法注入不仅能用于设值注入，也可用于构造注入。

## 5.8 bean 的继承

如果两个 bean 之间的配置信息非常相似，可利用继承来减少重复配置工作。

继承是指子 bean 定义可从父 bean 定义继承部分配置信息，也可覆盖特定的配置信息，或者添加一些配置。使用继承配置可以节省很多的配置工作。在实际应用中，通用配置会被配置成模板，可供子 bean 继承。

### 5.8.1 使用 abstract 属性

正如前面所介绍的，通用的配置会被配置成模板，而模板不需要实例化，仅仅作为

子 bean 定义的模板使用。而 ApplicationContext 默认预初始化所有的 singleton bean。

使用 abstract 属性，可以阻止模板 bean 被预初始化。abstract 属性为 true 的 bean 称为抽象 bean，容器会忽略所有的抽象 bean 定义，预初始化时不初始化抽象 bean。如果没有定义 abstract 属性，该属性默认为 false。

如下配置文件定义了一个抽象 bean，该抽象 bean 作为模板使用：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
  <bean id="steelAxe" class="lee.SteelAxe"/>
  <!-- 通过 abstract 属性定义该 bean 是抽象 bean-->
  <bean id="chineseTemplate" class="lee.Chinese" abstract="true">
    <!-- 定义依赖注入的属性-->
    <property name="axe">
      <ref local="steelAxe"/>
    </property>
  </bean>
</beans>
```

从配置文件中可以看出，抽象 bean 的定义与普通 bean 的定义几乎没有区别，仅仅增加 abstract 属性为 true，但主程序执行结果却有显著的差别。下面的主程序采用 ApplicationContext 作为 Spring 容器，ApplicationContext 默认预初始化所有的 singleton bean。其主程序部分如下：

```
public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
    }
}
```

主程序部分仅仅实例化了 ApplicationContext，在实例化 ApplicationContext 时，默认实例化 singleton bean。程序执行结果如下：

```
[java] Spring 实例化依赖 bean: SteelAxe 实例...
```

容器并没有实例化 chineseTemplate bean，而忽略了所有声明为 abstract 的 bean。如果取消 abstract 属性定义，则程序执行结果如下：

```
[java] Spring 实例化依赖 bean: SteelAxe 实例...
[java] Spring 实例化主调 bean: Chinese 实例...
[java] Spring 执行依赖关系注入...
```

可以看出，抽象 bean 是一个 bean 模板，容器会忽略抽象 bean 定义，因而不会实例化抽象 bean。但抽象 bean 无须实例化，因此可以没有 class 属性。如下的配置文件也有效：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd-->
```

```

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="steelAxe" class="lee.SteelAxe"/>
    <!-- 通过 abstract 属性定义该 bean 是抽象 bean。抽象 bean 没有指定 class 属性-->
    <bean id="chineseTemplate" abstract="true">
        <!-- 定义依赖注入的属性-->
        <property name="axe">
            <ref local="steelAxe"/>
        </property>
    </bean>
</beans>

```

注意：抽象 bean 不能实例化，既不能通过 getBean 获得抽象 bean，也不能让其他 bean 的 ref 属性值指向抽象 bean，因而只要企图实例化抽象 bean，都将导致错误。

## 5.8.2 定义子 bean

我们把指定了 parent 属性值的 bean 称为子 bean；parent 指向子 bean 的模板，称为父 bean。

子 bean 可以从父 bean 继承实现类、构造器参数及属性值，也可以增加新的值。如果指定了 init-method, destroy-method 和 factory-method 的属性，则它们会覆盖父 bean 的定义。

子 bean 无法从父 bean 继承如下属性：depends-on, autowire, dependency-check, singleton, lazy-init。这些属性将从子 bean 定义中获得，或采用默认值。

通过设置 parent 属性来定义子 bean，parent 属性值为父 bean id。修改上面的配置文件如下，增加了子 bean 定义：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="steelAxe" class="lee.SteelAxe"/>
    <!-- 通过 abstract 属性定义该 bean 是抽象 bean-->
    <bean id="chineseTemplate" class="lee.Chinese" abstract="true">
        <!-- 定义依赖注入的属性-->
        <property name="axe">
            <ref local="steelAxe"/>
        </property>
    </bean>
    <!-- 通过 parent 属性定义子 bean-->
    <bean id="chineseTemplate" parent="chineseTemplate"/>
</beans>

```

子 bean 与普通 bean 的定义并没有太大区别，仅仅增加了 parent 属性。子 bean 可以没有 class 属性，若父 bean 定义中有 class 属性，则子 bean 定义中可省略其 class 属性，但子 bean 将采用与父 bean 相同的实现类。程序执行结果如下：

```
[java] Spring 实例化依赖 bean: SteelAxe 实例...
[java] Spring 实例化主调 bean: Chinese 实例...
[java] Spring 执行依赖关系注入...
[java] 钢斧砍柴真快
```

另外，子 bean 从父 bean 定义继承了实现类并依赖 bean。但子 bean 也可覆盖父 bean 的定义，看如下的配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="steelAxe" class="lee.SteelAxe"/>
    <bean id="stoneAxe" class="lee.StoneAxe"/>
    <!-- 通过 abstract 属性定义该 bean 是抽象 bean-->
    <bean id="chineseTemplate" class="lee.Chinese" abstract="true">
        <property name="axe">
            <ref local="steelAxe"/>
        </property>
    </bean>
    <!-- 通过 parent 属性定义子 bean-->
    <bean id="chinese" parent="chineseTemplate">
        <!-- 覆盖父 bean 的依赖定义 -->
        <property name="axe">
            <ref local="stoneAxe"/>
        </property>
    </bean>
</beans>
```

程序执行结果如下：

```
[java] Spring 实例化依赖 bean: SteelAxe 实例...
[java] Spring 实例化依赖 bean: StoneAxe 实例...
[java] Spring 实例化主调 bean: Chinese 实例...
[java] Spring 执行依赖关系注入...
[java] 石斧砍柴好慢
```

此时，子 bean 的依赖不再是父 bean 定义的依赖了。

**注意：**上例中的子 bean 定义都没有 class 属性，因为父 bean 定义中已有 class 属性，子 bean 的 class 属性可从父 bean 定义中继承；如果父 bean 定义中也没有指定 class 属性，则子 bean 定义中必须指定 class 属性，否则会出错；如果父 bean 定义指定了 class 属性，子 bean 定义也指定了 class 属性，则子 bean 将定义的 class 属性覆盖父 bean 定义的 class 属性。

### 5.8.3 Spring bean 的继承与 Java 中继承的区别

Spring 中的 bean 继承与 Java 中的继承截然不同。前者是实例与实例之间参数值的延续，后者则是从一般到特殊的细化。前者是对象与对象之间的关系，后者是类与类之间的关系。因此，Spring 中 bean 的继承和 Java 中 bean 的继承有如下区别：

- Spring 中的子 bean 和父 bean 可以是不同类型，但在 Java 中的，子类是对父类的加强，是一种特殊的父类。
- Spring 中 bean 的继承是实例之间的关系，主要表现为参数值的延续；而 Java 中的继承是类与类之间的关系，主要表现为方法及属性的延续。
- Spring 中子 bean 不可作父 bean 使用，不具备多态性；而 Java 中的子类实例完全可当成父类实例使用。

## 5.9 bean 后处理器

Spring 提供了一种 bean，这种 bean 并不对外提供服务，无需 id 属性，但它负责对容器中的其他 bean 执行处理，例如为容器中的目标 bean 生成代理。这种 bean 可称为 bean 后处理器，它在 bean 实例创建成功后，对其进行进一步的加强处理。

bean 后处理器必须实现 BeanPostProcessor 接口，该接口包含以下两个方法。

- Object postProcessBeforeInitialization(Object bean, String name) throws BeansException: 该方法第一个参数是系统即将初始化的 bean 实例；第二个参数是 bean 实例的名字。
- Object postProcessAfterInitialization(Object bean, String name) throws BeansException: 该方法的第一个参数是系统刚完成初始化的 bean 实例；第二个参数是 bean 实例的名字。

实现该接口的 bean 必然会实现这两个方法，这两个方法在容器中的每个 bean 执行初始化方法前后分别调用。

另外，这两个方法也可用于对系统完成的默认初始化进行加强。看如下代码：

```
//自定义 bean 后处理器，负责后处理容器中的所有 bean
public class MyBeanPostProcessor implements BeanPostProcessor
{
    //在初始化 bean 之前调用该方法
    public Object postProcessBeforeInitialization(Object bean,
                                                String beanName) throws BeansException
    {
        //仅仅打印一行字符串
        System.out.println("系统正在准备对" + beanName + "进行初始化...");
        return bean;
    }
    //在初始化 bean 之后调用该方法
    public Object postProcessAfterInitialization(Object bean,
                                                String beanName) throws BeansException
    {
        System.out.println("系统已经完成对" + beanName + "的初始化");
        //如果系统刚完成初始化的 bean 是 Chinese
        if (bean instanceof Chinese)
        {
            //为 Chinese 的实例设置 name 属性。
            Chinese c = (Chinese)bean;
            c.setName("wawa");
        }
    }
}
```

```

        return bean;
    }
}
}

```

下面是 Chinese 的源代码，该类实现了 InitializingBean 接口，另外还提供了一个初始化方法，这都是由 Spring 容器控制的生命周期行为。

```

public class Chinese implements Person, InitializingBean
{
    private Axe axe;
    private String name;
    public Chinese()
    {
        System.out.println("Spring 实例化主调 bean: Chinese 实例... ");
    }
    public void setAxe(Axe axe)
    {
        System.out.println("Spring 执行依赖关系注入... ");
        this.axe = axe;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public void useAxe()
    {
        System.out.println(name + axe.chop());
    }
    public void init()
    {
        System.out.println("正在执行初始化方法 init... ");
    }
    public void afterPropertiesSet() throws Exception
    {
        System.out.println("正在执行初始化方法 afterPropertiesSet... ");
    }
}
}

```

配置文件如下：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 DTD>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置 bean 后处理器，没有 id 属性-->
    <bean class="lee.MyBeanPostProcessor"/>
    <bean id="steelAxe" class="lee.SteelAxe"/>
    <bean id="chinese" class="lee.Chinese" init-method="init">
        <property name="axe">
            <ref local="steelAxe"/>
        </property>
    </bean>
</beans>

```

本程序配置了两个初始化方法：

- init-method 指定初始化方法。

- 实现 InitializingBean 接口。

Chinese 类实现了 BeanPostProcessor 接口，并实现了这两个方法，这两个方法分别在初始化方法调用前、后得到回调。其主程序如下：

```
public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        InputStream is = new FileInputStream("bean.xml");
        //以指定的文件输入流 is, 创建 Resource 对象
        InputStreamResource isr = new InputStreamResource(is);
        //以 Resource 对象作为参数, 创建 BeanFactory 的实例
        XmlBeanFactory factory = new XmlBeanFactory(isr);
        //实例化 BeanPostProcessor
        Chinese c = new Chinese();
        //注册 BeanPostProcessor 实例
        factory.addBeanPostProcessor(c);
        System.out.println("程序已经实例化 BeanFactory...");
        Person p = (Person)factory.getBean("chinese");
        System.out.println("程序中已经完成了 chinese bean 的实例化...");
        p.useAxe();
    }
}
```

使用 BeanFactory 时必须手动注册 BeanPostProcessor 实例。通过 XmlBeanFactory 的 addBeanPostProcessor 可以注册 BeanPostProcessor 实例。程序执行结果如下：

```
[java] 系统正在准备对 steelAxe 进行初始化...
[java] 系统已经完成对 steelAxe 的初始化
[java] Spring 实例化主调 bean: Chinese 实例...
[java] Spring 执行依赖关系注入...
[java] 系统正在准备对 chinese 进行初始化...
[java] 正在执行初始化方法 afterPropertiesSet...
[java] 正在执行初始化方法 init...
[java] 系统已经完成对 chinese 的初始化
[java] wawa 钢斧砍柴真快
```

**注意：**在配置文件中配置 Chinese 实例时，并未确定 name 属性，但程序执行时，其 name 属性有了确定值。这是由 bean 后处理器完成的，在 bean 后处理器中判断 bean 是否为 Chinese 实例，并设置了它的 name 属性。

容器中一旦注册了 bean 后处理器后，bean 后处理器就会自动启动，并在容器中每个 bean 创建时自动工作。

如果实现了 BeanPostProcessor 接口，就可对 bean 做任何操作，包括完全忽略这个回调。BeanPostProcessor 通常用来检查标记接口，或者将 bean 包装成一个 proxy 等。Spring 的很多工具类，就是通过实现 BeanPostProcessor 接口实现的。

从主程序中看到，采用 BeanFactory 作为 Spring 容器时，必须手动注册 BeanPostProcessor。而对于 ApplicationContext，则无须手动注册。因为 ApplicationContext 可自动检测到容器中的 bean 后处理器，并将其注册成 BeanPostProcessor，它们会在 bean 实例创建时自动启动。其主程序采用如下代码可以达到相同的效果：

```
public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        Person p = (Person)factory.getBean("chinese");
        System.out.println("程序中已经完成了 chinese bean 的实例化...");
        p.useAxe();
    }
}
```

使用 ApplicationContext 作为容器时，无须手动注册 BeanPostProcessor。因此如果需要使用 bean 后处理器，建议使用 ApplicationContext，而不使用 BeanFactory。

## 5.10 容器后处理器

与 bean 后处理器对应的是，Spring 还提供了容器后处理器。容器后处理器在容器实例化结束后，对容器进行额外的处理。

容器后处理器必须实现 BeanFactoryPostProcessor 接口。实现该接口必须实现一个方法，该方法的方法体是对 BeanFactory 所做的定制：

```
void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
```

类似于 BeanPostProcessor，ApplicationContext 可自动检测到 BeanFactoryPostProcessor，然后作为容器后处理器注册，但若使用 BeanFactory 作为容器，则必须手动注册。看如下 bean，该 bean 实现了 BeanFactoryPostProcessor 接口：

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor
{
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
        throws BeansException
    {
        //仅打印了一行代码
        System.out.println("程序对 Spring 所做的 BeanFactory 的初始化没有意见...");
    }
}
```

将该 bean 作为普通 bean 部署在容器中，然后使用 ApplicationContext 作为容器，此时，容器会自动调用 BeanFactoryPostProcessor 来处理 BeanFactory。程序执行结果如下：

```
[java] 程序对 Spring 所做的 BeanFactory 的初始化没有意见...
```

Spring 已提供很多容器后处理器，主要包括以下几种。

- PropertyPlaceholderConfigurer：属性占位符配置器。
- PropertyOverrideConfigurer：另一种属性占位符配置器。

## 5.10.1 属性占位符配置器

PropertyPlaceholderConfigurer 是 BeanFactoryPostProcessor 的实现类，它可用于读取 Java Properties 文件中属性，然后插入 BeanFactory 定义中。通过使用 PropertyPlaceholderConfigurer，可以将 Spring 配置文件的某些属性放入属性文件中配置，从而可以修改属性文件。而修改 Spring 配置时，无须修改 BeanFactory 的主 XML 定义文件（比如说数据库的 urls、用户名和密码）。

使用 BeanFactoryPostProcessor 在修改某个部分的属性时，可以无须打开 Spring 配置文件，从而保证不会将新的错误引入 Spring 配置文件。

看下面的配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.Property
PlaceholderConfigurer">
        <!-- locations 属性指定属性文件的位置-->
        <property name="locations">
            <list>
                <value>dbconn.properties</value>
                <!--如果有多个属性文件，依次在下面列出来-->
            </list>
        </property>
    </bean>
    <!-- 配置本地的 DBCP 数据源 -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManager
DataSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>
</beans>
```

如前面所述，ApplicationContext 会自动检测其中的 BeanFactoryPostProcessor，无须额外的注册，容器会自动注册。因此，只需提供如下 Java Properties 文件：

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://wonder:3306/j2ee
jdbc.username=root
jdbc.password=pass
```

通过这种方法，可从主 XML 配置文件中分离出部分配置信息。如果仅需要修改数据库连接属性，则无须修改主 XML 配置文件，只需修改属性文件即可。采用属性占位符的配置方式，可以支持使用多个属性文件。通过这种方式，可将配置文件分割成多个属性文件，从而降低修改配置文件的风险。

## 5.10.2 另一种属性占位符配置器（PropertyOverrideConfigurer）

与 PropertyPlaceholderConfigurer 不同的是：PropertyOverrideConfigurer 利用属性文件的相关信息，覆盖 XML 配置文件中定义。即 PropertyOverrideConfigurer 允许 XML 配置文件中有默认的配置信息。

如果 PropertyOverrideConfigurer 的属性文件有对应配置信息，则 XML 文件中的配置信息被覆盖；否则，直接使用 XML 文件中的配置信息。使用 PropertyOverrideConfigurer 属性文件的格式如下：

```
beanName.property=value
```

beanName 是属性占位符企图覆盖的 bean 名，property 是企图覆盖的属性名。看如下配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
  <!-- 配置一个属性占位符 bean.ApplicationContext 能自动识别 PropertyPlaceholder
Configurer bean -->
  <bean id="propertyOverrider"
    class="org.springframework.beans.factory.config.Property
OverrideConfigurer">
    <property name="locations">
      <list>
        <value>dbconn.properties</value>
        <!--如果有多个属性文件，依次在下面列出来-->
      </list>
    </property>
  </bean>
  <!-- 配置本地的 DBCP 数据源 -->
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManager
DataSource" destroy-method="close">
    <property name="driverClassName" value="dd"/>
    <property name="url" value="xx"/>
    <property name="username" value="dd"/>
    <property name="password" value="dd"/>
  </bean>
</beans>
```

容器自动注册 propertyOverrider bean，读取 dbconn.properties 文件中的属性，并用于覆盖目标 bean 的属性。其配置文件中 dataSource bean 的属性完全是随意输入的，最终被属性文件的配置覆盖，其属性文件如下：

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://wonder:3306/j2ee
dataSource.username=root
dataSource.password=32147
```

注意属性文件的格式必须是：

```
beanName.property=value
```

也就是说，`dataSource` 必须是容器中真实存在的 bean 名，否则程序将出错。

**注意：**仅仅察看 XML 定义文件，程序无法知道 BeanFactory 定义是否被覆盖；当有多个 `PropertyOverrideConfigurer` 对同一个 bean 属性定义了覆盖，则最后一个覆盖有效。

## 5.11 与容器交互

Spring 容器本质上是一个高级工厂，负责产生 bean 实例，并管理 singleton bean 的生命周期。bean 处于容器管理下，通常无须访问容器，一样可以接受容器的注入，依赖关系通常由容器动态注入，而无须 bean 主动请求。容器通常有两种表现形式：

- BeanFactory。
- ApplicationContext。

当然，程序中也可显式地获得容器本身，如需获得容器本身，可采用如下代码。

获得 BeanFactory 容器：

```
InputStream is = new FileInputStream("bean.xml");
//以指定的文件输入流 is, 创建 Resource 对象
InputStreamResource isr = new InputStreamResource(is);
//以 Resource 对象作为参数, 创建 BeanFactory 的实例
BeanFactory factory = new XmlBeanFactory(isr);
```

获得 ApplicationContext 容器：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
```

### 5.11.1 工厂 bean 简介与配置

通常情况下，Spring 容器担任工厂角色，bean 无须自己实现工厂模式，仅需要提供一个工厂方法，用来返回其他 bean 实例。但少数情况下，容器中的 bean 本身就是工厂，其作用是产生其他 bean 实例。

工厂 bean 的配置和其他 bean 的配置区别在于其产品 bean 的配置。在如下的配置文件中配置了一个工厂 bean：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <bean id="personFactory" class="lee.PersonFactory"/>
</beans>
```

该工厂 bean 包含一个实例工厂方法 `Person getPerson(String ethnic)`，该方法用来返回

Person 实例。此时，产品 bean 不再由 Spring 容器产生，因此与普通 bean 的配置不同，不再需要提供 class 元素。因为容器不关心 bean 的实现类，只需要知道产品 bean 的工厂，以及相应的工厂方法。容器执行相应的工厂 bean 的工厂方法，并将工厂方法返回产品 bean。如果工厂方法需要参数，则应提供相应参数。配置产品 bean 通常有两种方法：

- 使用 factory-method 属性确定工厂方法。
- 将 bean 方法返回值定义成 bean 实例。

在配置文件中增加如下代码：

```
<!-- factory-bean 确定工厂 bean 实例，factory-method 确定实例工厂方法
    personFactory 必须对应上面配置中已有的工厂 bean，factory-method 是
    工厂中的实例方法-->
<bean id="chinese" factory-bean="personFactory" factory-method="getPerson">
    <!-- constructor-arg 用于传入工厂方法的参数 -->
    <constructor-arg>
        <value>chin</value>
    </constructor-arg>
</bean>
```

这段配置文件采用 factory-method 来配置产品 bean。配置文件使用 factory-bean 确定工厂 bean；使用 factory-method 确定工厂方法；使用 constructor-arg 确定工厂方法的参数。在配置文件中再增加如下代码：

```
<!-- 定义实例工厂的方法返回值为 bean，通过 MethodInvokingFactoryBean 类完成-->
<bean id="american" class="org.springframework.beans.factory.config.
MethodInvokingFactoryBean">
    <!-- targetObject 属性确定工厂 bean -->
    <property name="targetObject"><ref local="personFactory" /></
    property>
    <!-- targetMethod 属性确定工厂方法 -->
    <property name="targetMethod"><value>getPerson</value></property>
    <!-- arguments 可接受 list 参数，确定工厂方法所需要的参数-->
    <property name="arguments">
        <list>
            <!-- 如需多个参数，此处列出 -->
            <value>ame</value>
        </list>
    </property>
</bean>
```

这段配置文件演示了产品 bean 的又一种配置方法——使用辅助类 MethodInvoking FactoryBean。通过 targetObject 来确定工厂 bean；由 targetMethod 确定工厂方法；arguments 确定工厂方法所需的参数。主程序如下：

```
public class SpringTest
{
    public static void main(String[] args) throws Exception
    {
        //在当前路径下搜索 bean.xml 文件，实例化文件输入流
        InputStream is = new FileInputStream("bean.xml");
        //以指定的文件输入流 is，创建 Resource 对象
        InputStreamResource isr = new InputStreamResource(is);
        //以 Resource 对象作为参数，创建 BeanFactory 的实例
```

```

XmlBeanFactory factory = new XmlBeanFactory(isr);
System.out.println("程序已经实例化 BeanFactory...");
//获取第一个产品 bean
Person p1 = (Person)factory.getBean("chinese");
//测试第一个产品 bean
System.out.println(p1.sayHello("Mary"));
System.out.println(p1.sayGoodBye("Mary"));
//获取第二个产品 bean
Person p2 = (Person)factory.getBean("american");
//测试第二个产品 bean
System.out.println(p2.sayHello("Jack"));
System.out.println(p2.sayGoodBye("Jack"));
}
}

```

主程序获取了两个产品 bean，分别执行两个产品 bean 的测试方法。程序执行结果如下：

```

java] Mary, 您好
java] Mary, 下次再见
java] Jack,Hello
[java] Jack,Good Bye

```

## 5.11.2 FactoryBean 接口

FactoryBean 接口是 Spring 提供的工厂 bean 的标准接口，使用 FactoryBean 接口可简化工厂 bean 的开发和配置。实现该接口的 bean，通常作为工厂 bean 使用，容器通常不会返回 FactoryBean 实例，而是 FactoryBean 的产品。FactoryBean 接口提供如下三个方法：

- Object getObject()
- Class getObjectType()
- boolean isSingleton()

**注意：**实现该接口的 bean 无法作为正常 bean 使用；FactoryBean 的定义与普通 beans 的定义没有区别，但当客户端对该 bean id 请求时，容器将返回该 FactoryBean 的产品。

看如下代码：

```

/**
 * PersonFactory 实现 FactoryBean 接口，变成标准工厂 bean
 */
public class PersonFactory implements FactoryBean
{
    Person p = null;
    /**
     * 实现 FactoryBean 必须实现的方法
     * @return 工厂 beans 产生的实例
     */
    public Object getObject() throws Exception
    {
        if (p == null)
        {

```

```

        p = new Chinese();
    }
    return p;
}
/**
 * 实现 FactoryBean 必须实现的方法
 * @return 工厂 bean 产生的实例的类型
 */
public Class getObjectType()
{
    return Chinese.class;
}
/**
 * 实现 FactoryBean 必须实现的方法
 * @return 工厂 beans 产生的实例是否单态
 */
public boolean isSingleton()
{
    return true;
}
}

```

下面将该 bean 作为普通 bean，并配置在 BeanFactory 中，配置文件如下：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置 BeanFactory 实例-->
    <bean id="personFactory" class="lee.PersonFactory"/>
</beans>

```

主程序部分如下：

```

public class SpringTest
{
    public static void main(String[] args) throws Exception
    {
        //在当前路径下搜索 bean.xml 文件，实例化文件输入流
        InputStream is = new FileInputStream("bean.xml");
        //以指定的文件输入流 is，创建 Resource 对象
        InputStreamResource isr = new InputStreamResource(is);
        //以 Resource 对象作为参数，创建 BeanFactory 的实例
        XmlBeanFactory factory = new XmlBeanFactory(isr);
        //获得 FactoryBean 产生的产品 bean
        Person p1 = (Person)factory.getBean("personFactory ");
        //测试产品 bean
        System.out.println(p1.sayHello("Mary"));
        System.out.println(p1.sayGoodBye("Mary"));
        //获得 FactoryBean 产生的产品 bean
        Person p2 = (Person)factory.getBean("personFactory ");
        System.out.println(p1==p2);
    }
}

```

从代码中可看到：当客户端代码直接请求 FactoryBean id 时，容器并不返回

FactoryBean 实例，而是返回 FactoryBean 的产品 bean。因此执行效果如下：

```
[java] Mary, 您好  
[java] Mary, 下次再见  
[java] true
```

因为 FactoryBean 以单态方法管理产品 bean，所以两次请求的产品是同一个共享实例。通常，容器不会返回 FactoryBean 实例，如果实在需要获取 FactoryBean 本身，可采用如下方式：

```
getBean(&beanId)
```

当请求获得 FactoryBean 时，并不直接请求 bean id，而在 bean id 前增加&符号。此时容器则返回 BeanFactory 本身，而不是其产品 bean。在主程序中增加如下代码：

```
System.out.println(factory.getBean("&personFactory "));
```

则可看到增加如下输出：

```
[java] lee.PersonFactory@c4aad3
```

此时程序返回的是 FactoryBean 本身，而不是其产品。

### 5.11.3 实现 BeanFactoryAware 接口获取 BeanFactory

实现 BeanFactoryAware 接口的 bean 可以直接访问 Spring 容器，被容器创建以后，它会拥有一个指向 Spring 容器的引用。BeanFactoryAware 接口只有一个方法：

- void setBeanFactory(BeanFactory beanFactory)。

该方法有一个参数：beanFactory，该参数指向创建它的 BeanFactory。看如下代码：

```
/**  
 * 定义一个类，实现 BeanFactoryAware 接口，该类将拥有与容器交互的能力  
 */  
public class Chinese implements BeanFactoryAware  
{  
    //将 BeanFactory 容器以成员变量保存  
    private BeanFactory factory;  
    /**  
     * 实现 BeanFactoryAware 接口必须实现的方法  
     * @param beanFactory 创建 bean 实例的 BeanFactory  
     */  
    public void setBeanFactory(BeanFactory beanFactory)  
        throws BeansException  
    {  
        this.factory = beanFactory;  
    }  
    /**  
     * 获得 BeanFactory 的测试方法  
     * @return beanFactory 创建 bean 实例的 BeanFactory  
     */  
    public BeanFactory getFactory()  
    {  
        return factory;
```

```
}
```

将该 bean 部署在 BeanFactory 中，部署该 bean 与部署其他 bean 没有任何区别，其 XML 配置文件如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置 BeanFactoryAware bean-->
    <bean id="chinese" class="lee.Chinese"/>
</beans>
```

对主程序部分进行简单测试，将直接实例化获得 BeanFactory 与通过 Chinese bean 获得 BeanFactory 进行比较，其主程序如下：

```
public class SpringTest
{
    public static void main(String[] args) throws Exception
    {
        //在当前路径下搜索 bean.xml 文件，实例化文件输入流
        InputStream is = new FileInputStream("bean.xml");
        //以指定的文件输入流 is，创建 Resource 对象
        InputStreamResource isr = new InputStreamResource(is);
        //以 Resource 对象作为参数，创建 BeanFactory 的实例
        XmlBeanFactory factory = new XmlBeanFactory(isr);
        Chinese p = (Chinese)factory.getBean("chinese");
        //打印出 Chinese 实例获得的 BeanFactory
        System.out.println(p.getFactory());
        //比较两种方法获得的 BeanFactory
        System.out.println(factory == p.getFactory());
    }
}
```

程序运行结果如下：

```
[java] org.springframework.beans.factory.xml.XmlBeanFactory defining beans
[chinese]; root of BeanFactory hierarchy
[java] true
```

实现 BeanFactoryAware 接口，可让 bean 拥有了访问容器的能力，但污染了代码，导致代码与 Spring 接口耦合在一起。因此，建议不要直接访问容器。

注意：一般情况下，bean 无须访问容器本身，bean 的依赖关系由容器负责注入，在运行过程中无须感受到容器的存在。

#### 5.11.4 使用 BeanNameAware 回调本身

如果某个 bean 需要访问配置文件中本身的 id 属性，则可以使用 BeanNameAware 接口，该接口提供了回调本身的能力。实现该接口的 bean，能访问到本身的 id 属性。该接

口提供一个方法：void setBeanName(String name)。

该方法的 name 参数就是 bean 的 id。该方法在依赖关系设置之后，初始化回调（InitializingBean 的 afterPropertiesSet 方法，或者 init-method 指定的方法）之前被执行。回调 setBeanName 方法可让 bean 获得自己的 id。看如下代码：

```
public class Chinese implements InitializingBean, BeanNameAware
{
    private String beanName;
    // 测试用初始化方法，该方法通过 init-method 属性确定为初始化方法
    public void init()
    {
        System.out.println("正在执行初始化方法 init...");
    }
    // 实现 InitializingBean 接口必须实现的方法，初始化方法的一种
    public void afterPropertiesSet() throws Exception
    {
        System.out.println("正在执行初始化方法 afterPropertiesSet...");
    }
    /**
     * 实现 BeanNameAware 接口必须实现的方法。
     * @Param name bean 的 id。
     */
    public void setBeanName(String name)
    {
        this.beanName = name;
        // 测试，打印出 bean id。
        System.out.println("回调 setBeanName 方法 " + name);
    }
}
```

将该 bean 部署在容器中，与普通 bean 的部署没有任何区别。在主程序中通过如下代码测试：

```
public class SpringTest
{
    public static void main(String[] args) throws Exception
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        Chinese p = (Chinese)ctx.getBean("chinese");
    }
}
```

执行结果如下：

```
[java] 回调 setBeanName 方法 chinese
[java] 正在执行初始化方法 afterPropertiesSet...
[java] 正在执行初始化方法 init...
```

## 5.12 ApplicationContext 介绍

ApplicationContext 是 BeanFactory 接口的子接口，它增强了 BeanFactory 的功能，处于 context 包下。很多时候，ApplicationContext 允许以声明式方式操作容器，无须手动

创建。可利用如 ContextLoader 的支持类，在 Web 应用启动时自动创建 ApplicationContext。当然，也可以采用编程方式创建 ApplicationContext。

context 的基础是 ApplicationContext 接口，它继承 BeanFactory 接口，并提供 BeanFactory 所有的功能。为了以一种更加面向框架的方式工作，context 包使用分层和有继承关系的上下文类，包括以下几种：

- MessageSource，提供国际化支持。
- 资源访问，比如 URL 和文件。
- 事件传递。
- 载入多个配置文件。

ApplicationContext 包括 BeanFactory 的全部功能，因此建议优先使用 ApplicationContext。除非对于某些内存非常关键的应用，才考虑使用 BeanFactory。

### 5.12.1 国际化支持

ApplicationContext 接口继承 MessageSource 接口，因此具备国际化功能。下面是 MessageSource 接口中定义的三个用于国际化的方法：

- String getMessage (String code, Object[] args, Locale loc)。
- String getMessage (String code, Object[] args, String default, Locale loc)。
- String getMessage(MessageSourceResolvable resolvable, Locale locale)。

ApplicationContext 也通过这三个方法，完成消息的国际化。在 ApplicationContext 加载时，自动查找在 context 中 MessageSource bean。该 bean 的名字必须是 MessageSource。一旦找到这个 bean，上述三个方法的调用被委托给 MessageSource。如果没有找到该 bean，ApplicationContext 会查找其父定义中的 MessageSource bean。如果找到，它将被作为 MessageSource 使用。如果无法找到 MessageSource bean，则将会实例化空的 StaticMessageSource bean，该 bean 能接受上述三个方法的调用。

Spring 的国际化通常采用 ResourceBundleMessageSource 类。看下面配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置 messageSource bean，该 bean 的名字必须是 messageSource
        采用 Spring 的实现类 ResourceBundleMessageSource-->
    <bean id="messageSource" class="org.springframework.context.support.
ResourceBundleMessageSource">
        <!-- basenames 确定资源文件的文件名，该属性接受 list 值，用于接受多个资源文件-->
        <property name="basenames">
            <list>
                <!-- 确定一份资源文件，资源文件名为 message -->
                <value>message</value>
                <!-- 如果有多个资源文件，全部列在此处-->
```

```

        </list>
    </property>
</bean>
</beans>

```

然后给出如下两份资源文件：

第一份，英文的资源文件，文件名： message\_en.properties。

```

hello=welcome,{0}
now=now is :{1}

```

第二份，中文的资源文件，文件名： message\_zh.properties。

```

hello=欢迎你,{0}
now=现在时间是:{1}

```

当然，应使用 native2ascii 工具将这份资源文件国际化，命令如下：

```
native2ascii message_zh.properties message_zh_1.properties
```

删除 message\_zh.properties 文件，将 message\_zh\_1.properties 文件重命名为 message\_zh.properties。此时，程序拥有了两份资源文件，可以自适应英语和汉语的环境。主程序部分如下：

```

public class SpringTest
{
    public static void main(String[] args) throws Exception
    {
        //实例化 ApplicationContext
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //创建参数数组
        String[] a = {"读者"};
        //使用 getMessage 方法获取本地化消息。Locale 的 getDefault 方法用来返回计算机
        //环境的 Locale
        String hello = ctx.getMessage("hello", a, Locale.getDefault());
        Object[] b = {new Date()};
        String now = ctx.getMessage("hello", b, Locale.getDefault());
        //打印出两条本地化消息
        System.out.println(hello);
        System.out.println(now);
    }
}

```

程序的执行结果会随环境不同而改变，在简体中文的环境下，执行结果如下：

```

[java] 欢迎你,读者
[java] 欢迎你,06-5-8 下午 3:34

```

英文环境下，执行结果如下：

```

[java] welcome,读者
[java] welcome,5/8/06 3:53 PM

```

当然，即使在英文环境下，“读者”这个词都无法变成英文，因为“读者”是写在程序代码中，而不是从资源文件中获得。

## 5.12.2 事件处理

通过 ApplicationEvent 类和 ApplicationListener 接口，可实现 ApplicationContext 的事件处理。如果容器中有一个 ApplicationListener bean，每当 ApplicationContext 发布 ApplicationEvent 时，ApplicationListener bean 将自动响应，这是标准的观察者模式。

Spring 的事件框架有如下两个重要成员。

- ApplicationEvent：容器事件，必须由 ApplicationContext 发布。
- ApplicationListener：监听器，可由容器中的任何监听器 bean 担任。

以下是个简单的容器事件类，该类继承 ApplicationEvent。代码如下：

```
public class EmailListEvent extends ApplicationEvent
{
    public EmailListEvent(Object source)
    {
        super(source);
    }
    public EmailListEvent(Object source, String address, String text)
    {
        super(source);
        this.address = address;
        this.text = text;
    }
}
```

下面是监听器类，监听器类必须实现 ApplicationListener 接口。实现该接口必须实现一个方法：

- void onApplicationEvent(ApplicationEvent event)

每当容器内发生任何事件时，此方法都会被触发，监听器类的配置文件如下：

```
//监听器类，实现 ApplicationListener 接口
public class EmailNotifier implements ApplicationListener
{
    //实现 ApplicationListener 接口必须实现的方法，该方法会在容器发生事件时自动触发
    public void onApplicationEvent(ApplicationEvent evt)
    {
        //如果事件是程序触发的事件
        if (evt instanceof EmailEvent)
        {
            //发送 E-mail 通知...
            EmailEvent emailEvent = (EmailEvent)evt;
            System.out.println("需要发送邮件的接收地址 " +
                + emailEvent.address);
            System.out.println("需要发送邮件的邮件正文 " + emailEvent.text);
        }
        else
        {
            //容器内置事件不作任何处理
            System.out.println("容器本身的事件 " + evt);
        }
    }
}
```

将监听器配置在容器中，配置文件如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 配置监听器类-->
    <bean id="emailListListener" class="lee.EmailNotifier"/>
</beans>
```

其主程序部分使用 ApplicationContext 的 publishEvent 来发布事件：

```
public class SpringTest
{
    public static void main(String[] args)
    {
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        EmailEvent ele = new EmailEvent("hello", "kongyeeku@gmail.com",
"this is a test");
        ctx.publishEvent(ele);
    }
}
```

程序执行结果如下：

```
[java] 容器本身的事件 org.springframework.context.event.ContextRefreshed
Event[source=org.springframework.context.support.FileSystemXmlApplicationC
ontext: display name
[org.springframework.context.support.FileSystemXmlApplicationContext;
hashCode=7615385]; startup date [Mon May 08 16:54:39 CST 2006]; root of context
hierarchy]
[java] 需要发送邮件的接收地址 kongyeeku@gmail.com
[java] 需要发送邮件的邮件正文 this is a test
```

此时监听器不仅监听到程序发布的事件，同时也监听到容器内置的事件。

**注意：**如果 bean 想发布事件，则 bean 必须获得其容器的引用，应通过实现 BeanFactoryAware 接口达到此目的。

Spring 提供了以下三个内置事件：

- ContextRefreshedEvent: ApplicationContext 容器初始化或刷新触发该事件。
- ContextClosedEvent: ApplicationContext 容器关闭时触发该事件。
- RequestHandledEvent: Web 相关的事件，只能应用于使用 DispatcherServlet 的 Web 应用。

### 5.12.3 Web 应用中自动加载 ApplicationContext

对于 Web 应用，不必在代码中手动实例化 ApplicationContext。可通过 ContextLoader 声明式地创建 ApplicationContext。ContextLoader 有以下两个实现类：

- ContextLoaderListener。
- ContextLoaderServlet。

这两类功能相同，只是 listener 不能在 Servlet 2.2 兼容的容器中使用。根据 Servlet 2.4 规范，listener 会随 Web 应用启动时自动初始化，很多 Servlet 2.3 兼容的容器也提供该功能。

## 1. 使用 ContextLoaderListener

使用 ContextLoaderListener 注册 ApplicationContext 的配置文件如下，注意：下面的配置文件不是在 Spring 的配置文件中增加，而是在 web.xml 文件中增加。

```
<!-- 确定配置文件的位置 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- 此处可以列出多个 Spring 的 XML 配置文件-->
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value></context-param>
    <!-- 应用启动时，自动加载 listener，该 listener 会读取上面确定的 XML 配置文件。
        然后创建 ApplicationContext 实例-->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoader
Listener</listener-class>
    </listener>
```

## 2. 使用 ContextLoaderServlet

使用 ContextLoaderServlet 注册 ApplicationContext 的配置文件如下。同样，下面的配置文件也不是在 Spring 的配置文件中增加，而是在 web.xml 文件中增加。

```
<servlet>
    <!-- 确定 Servlet 的名-->
    <servlet-name>context</servlet-name>
    <!-- 确定 Servlet 对应的类名-->
    <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
    <!-- 确定 Servlet 的启动级别-->
    <load-on-startup>1</load-on-startup>
</servlet>
```

采用这种方式时，应将 context 的启动级别设成最小，即最优先启动。因为 ApplicationContext 是整个应用的核心。

注意：在两种启动方式中，推荐采用第一种。因为根据 Servlet 2.4 规范，listener 比 Servlet 优先启动；关键问题是有些容器并不支持 Servlet2.4 规范，即不支持 listener。

支持 listener 的容器有：

- Apache Tomcat 4.x 及更高版本。
- Jetty 4.x 及更高版本。
- Resin 2.1.8 及更高版本。
- Orion 2.0.2 及更高版本。
- BEA WebLogic 8.1 SP3

不支持 listener 的容器有：

- BEA WebLogic up to 8.1 SP2 及更低版本。
- IBM WebSphere 5.x 及更低版本。
- Oracle OC4J 9.0.3 及更低版本。

## 5.13 加载多个 XML 配置文件

对于大多数的应用，从表现层的 action，到持久层的 DataSource，都被 Spring 作为 bean 管理。如果这些 bean 被配置在同一个文件中，阅读及维护该配置文件将是一件非常有挑战的事情。

因此，Spring 建议：将一个大的配置文件分解成多个小的配置文件，使每个配置文件仅仅管理功能近似于 bean；这样不仅可以分散配置文件，降低修改配置文件的风险，而且更符合“分而治之”的软件工程原理。

多个配置文件最终需要汇总，ApplicationContext 提供如下方式来汇总多个配置文件：

- 使用 ApplicationContext 加载多个配置文件。
- Web 应用启动时加载多个配置文件。
- XML 配置文件中导入其他配置。

### 5.13.1 ApplicationContext 加载多个配置文件

ApplicatonContext 的常用实现类有如下两个：

- ClassPathXmlApplicationContext。
- FileSystemXmlApplicationContext。

这两个类都可以用来加载多个配置文件，它们的构造器都可以接收一个数组，并在该数组中存放多个配置文件。ClassPathXmlApplicationContext 可采用如下代码加载多个配置文件：

```
//创建配置文件数组  
//假设 3 个配置文件：a.xml, b.xml, c.xml  
String[] configLocations = {"a.xml", "b.xml", "c.xml"}  
以配置文件数组为参数，创建 ApplicationContext  
ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
```

与采用 FileSystemXmlApplicationContext 创建 ApplicationContext 的方式相似，区别仅在于二者搜索配置文件的路径不同：ClassPathXmlApplicationContext 通过 CLASSPATH 路径搜索配置文件；而 FileSystemXmlApplicationContext 则在当前路径搜索配置文件。

### 5.13.2 Web 应用启动时加载多个配置文件

参看 5.12.3 节所述，通过 ContextLoaderListener 也可加载多个配置文件，可利用

<context-param>元素来指定多个配置文件位置，其配置如下：

```
<!-- 确定配置文件的位置 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- 此处可以列出多个 Spring 的 XML 配置文件-->
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</
    param-value>
</context-param>
```

### 5.13.3 XML 配置文件中导入其他配置文件

配置文件本身和其子元素 import，可用于导入其他配置文件。具体的配置示例如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素 -->
<beans>
    <!-- 导入第一份配置文件：services.xml-->
    <import resource="services.xml"/>
    <!-- 导入第二份配置文件：resources/messageSource.xml -->
    <import resource="resources/messageSource.xml"/>
    <!-- 导入第三份配置文件：resources/ themeSource.xml -->
    <import resource="/resources/themeSource.xml"/>
    <!-- 下面定义该文件里的其他 bean-->
    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

## 本章小结

本章首先介绍了 Spring 的核心部分：依赖注入和 Spring 管理 bean 的方式，包括 bean 之间依赖关系的管理。

其次，重点介绍了 bean 的配置和使用，包括 bean 的行为方式，bean 的生命周期，以及 bean 的各种依赖关系的管理等。另外，也对 Spring 容器作了详细介绍。

最后，介绍了两个后处理器：bean 后处理器和容器后处理器。

# 第 6 章

## Spring 与 Hibernate 的整合

### 本章要点

- 『 Spring 对 Hibernate 的支持
- 『 管理 SessionFactory
- 『 使用 HibernateTemplate
- 『 Spring 提供的 DAO 支持
- 『 使用依赖注入管理 DAO 组件
- 『 声明式事务管理
- 『 事务代理的配置方法

虽然 Hibernate 大大简化了数据持久层的访问，但使用 Hibernate 进行持久层访问时，还存在一系列的问题，如访问步骤重复多，事务难以控制，以及基于 Hibernate 的 DAO 组件编写复杂等。

Spring 在 Hibernate 基础上，进一步简化了持久层访问，其模板的操作大大降低 Hibernate 的重复操作；Spring 提供的 DAO 支持简化了 DAO 组件的开发；SessionFactory 的依赖注入简化了 Session 的控制等；这些都极大地提高了 J2EE 应用的开发效率。

声明式事务的管理分离了业务逻辑和事务逻辑，将应用从特定的事务逻辑中解耦，使应用可以方便地在不同的事务策略之间切换。

## 6.1 Spring 对 Hibernate 的支持

在所有的 ORM 框架中，Spring 对 Hibernate 的支持最好。Spring 提供很多 IoC 特性的支持，方便地处理大部分典型的 Hibernate 整合问题，如 SessionFactory 的注入、HibernateTemplate 的简化操作及 DAO 支持等。另外，Spring 还提供了统一的异常体系及声明式事务管理等。

一旦 Hibernate 处于 Spring 的管理下，Hibernate 所需要的基础资源，都由 Spring 提供注入。Hibernate 创建 SessionFactory 必需的 DataSource，执行持久化必需的 Session 及持久层访问必需的事务控制等，这些原本必须通过代码控制的逻辑，都将由 Spring 接管；DataSource，SessionFactory，TransactionManager 等，都将作为 Spring 容器中的 bean。将这些 bean 放在配置文件中管理，可以提供很好的解耦。

Spring 提供了 DAO 支持，可以大大简化 DAO 组件的开发。IoC 容器的使用，提供了 DAO 组件与业务逻辑组件之间的松耦合。所有的 DAO 组件，都由容器负责注入到业务逻辑组件中，其业务组件无须关心 DAO 组件的实现。

面向接口编程及 DAO 模式的使用，提高了系统组件之间的解耦，降低了系统重构的成本。

通过 Spring 整合 Hibernate，使持久层的访问更加容易，使用 Spring 管理 Hibernate 持久层有如下优势。

- 通用的资源管理：Spring 的 ApplicationContext 能管理 SessionFactory，使得配置值很容易被管理和修改，无须使用 Hibernate 的配置文件。
- 有效的 Session 管理：Spring 提供了有效、简单和安全的 Hibernate Session 处理。
- IoC 容器提高了 DAO 组件与业务逻辑层之间的解耦。
- DAO 模式的使用，降低了系统重构的代价。
- 方便的事务管理：Hibernate 的事务管理处理会限制 Hibernate 的表现，而 Spring 的声明式事务管理力度是方法级。
- 异常包装：Spring 能够包装 Hibernate 异常，把它们从 checked exception 变为 runtime exception；开发者可选择在恰当的层处理数据中不可恢复的异常，从而避免烦琐的 catch/throw 及异常声明。

## 6.2 管理 SessionFactory

Hibernate 的 SessionFactory，是单个数据库映射关系编译后的内存镜像，是 Hibernate 执行持久化访问的基础部分。

大部分情况下，一个 J2EE 应用对应一个数据库。而 Spring 通过 ApplicationContext 管理 SessionFactory，无须采用单独 Hibernate 应用所必需的 hibernate.cfg.xml 文件。

Spring 配置管理 SessionFactory 与数据库的连接。而在实际的 J2EE 应用中，数据源

会采用依赖注入的方式，传给 Hibernate 的 SessionFactory。具体配置如下所示：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
    <!-- 定义数据源，该 bean 的 ID 为 dataSource-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
        <!-- 指定数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</
      value></property>
        <!-- 指定连接数据库的 URL-->
        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</ value>
      </property>
        <!-- root 为数据库的用户名-->
        <property name="username"><value>root</value></property>
        <!-- pass 为数据库密码-->
        <property name="password"><value>pass</value></property>
    </bean>
    <!-- 定义 Hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
      LocalSessionFactoryBean">
        <!-- 依赖注入数据源，注入正是上文定义的 dataSource-->
        <property name="dataSource"><ref local="dataSource"/></property>
        <!-- mappingResources 属性用来列出全部映射文件-->
        <property name="mappingResources">
            <list>
                <!-- 以下用来列出所有的 PO 映射文件-->
                <value>lee/MyTest.hbm.xml</value>
            </list>
        </property>
        <!-- 定义 Hibernate 的 SessionFactory 的属性 -->
        <property name="hibernateProperties">
            <props>
                <!-- 指定 Hibernate 的连接方法-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
                  MySQLDialect</prop>
                <!-- 不同数据库连接，启动时选择 create, update, create-drop-->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
</beans>

```

SessionFactory 由 ApplicationContext 管理，并随着应用启动时自动加载，可以被处于 ApplicationContext 管理的任意一个 bean 引用，比如 DAO。Hibernate 的数据库访问需要在 Session 管理下，而 SessionFactory 是 Session 的工厂。Spring 采用依赖注入为 DAO 对象注入 SessionFactory 的引用。

Spring 也提供了 Hibernate 的简化访问方式，Spring 采用模板设计模式，提供 Hibernate 访问与其他持久层访问的一致性。如果需要使用容器管理的数据源，则无须提供数据驱动等信息，只需要提供数据源的 JNDI 即可。对上文的 SessionFactory 只需将 dataSource

的配置替换成 JNDI 数据源，并将原有的 dataSource Bean 替换为如下所示：

```
<!-- 此处配置 JNDI 数据源-->
<bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <!-- 指定数据源的 JNDI-->
        <value>java:comp/env/jdbc/myds</value>
    </property>
</bean>
```

## 6.3 Spring 对 Hibernate 的简化

Hibernate 的持久层访问必须按如下步骤进行：

- (1) 创建 Configuration 实例。
- (2) 创建 SessionFactory 实例。
- (3) 创建 Session 实例。
- (4) 打开事务。
- (5) 开始持久化访问。
- (6) 提交事务。
- (7) 如果遇到异常，回滚事务。
- (8) 关闭 Session。

虽然可以采用类似于 HibernateUtils 工具类封装了部分过程，但依然不够简洁，需要通过代码显式地打开 Session，开始事务，然后关闭事务，最后关闭 Session。而 Spring 提供更简单的方式操作持久层，无须显式地打开 Session，也无须在代码中执行任何的事务操作语句。

Spring 提供了 HibernateTemplate，用于持久层访问，该模板类无须显示打开 Session 及关闭 Session。它只要获得 SessionFactory 的引用，将可以智能打开 Session，并在持久化访问结束后关闭 Session，程序开发只需完成持久层逻辑，通用的操作则由 HibernateTemplate 完成。

事务的处理，当然也可以采用编程式事务。Spring 提供了编程式事务的支持。通常，推荐使用声明式事务，使用声明式事务有如下优点：

- 代码中无须实现任何事务逻辑，程序开发者可以更专注于业务逻辑的实现。
- 声明式事务不与任何事务策略耦合，采用声明式事务可以方便地在全局事务和局部事务之间切换。

Spring 的声明式事务以 Spring 的 AOP 为基础，开发者可以不需要对 AOP 深入了解，只需按本章后面部分配置声明式事务代理即可。

Spring 对 Hibernate 的简化，还得益于 Spring 异常处理策略。Spring 认为：底层数据库异常几乎都不可恢复，强制处理底层数据库几乎没有任何意义，但传统 JDBC 数据库访问的异常都是 checked 异常，必须使用 try...、catch 块处理。

另外，Spring 包装了 Hibernate 异常，并转换到 DataAccessException 继承树内，所

有 `DataAccessException` 全部是 `runtime` 异常，但并不强制捕捉。归纳起来，Spring 对 Hibernate 的简化主要有以下几个方面。

- 基于依赖注入的 `SessionFactory` 管理机制，`SessionFactory` 是执行持久化操作的核心组件。传统 Hibernate 应用中，`SessionFactory` 必须手动创建，通过依赖注入，代码无须关心 `SessionFactory`，而它的创建和维护由 `BeanFactory` 负责管理。
- 更优秀的 Session 管理机制。Spring 提供“每事务一次 Session”的机制，该机制能大大提高了系统性能，而且 Spring 对 Session 的管理是透明的，无须在代码中操作 Session。
- 统一的事务管理。无论是编程式事务，还是声明式事务，Spring 都提供一致的编程模型，无须烦琐的开始事务、显式提交及回滚。如果使用声明式事务管理，可将事务管理逻辑与代码分离，使事务可在全局事务和局部事务之间切换。
- 统一的异常处理机制。不再强制开发者在持久层捕捉异常，通常持久层异常被包装成 `DataAccessException` 异常的子类，将底层数据库异常包装成业务异常，开发者可以自己决定在合适的层处理异常。
- `HibernateTemplate` 支持类。`HibernateTempate` 能完成大量 Hibernate 持久层操作，这些操作大多只需要一些简单的代码即可实现。

## 6.4 使用 `HibernateTemplate`

`HibernateTemplate` 可将 Hibernate 的持久层访问模板化，使用 `HibernateTemplate` 非常简单。创建 `HibernateTemplate` 实例后，注入一个 `SessionFactory` 的引用，就可执行持久化操作。`SessionFactoyr` 对象可通过构造参数传入，或通过设值方式传入。例如：

```
//获取 Spring 上下文
ApplicationContext ctx = new FileSystemXmlApplicationContext("bean.xml");
//通过上下文获得 SessionFactory
SessionFactory sessionFactory = (SessionFactory) ctx.getBean("sessionFactory");
```

`HibernateTemplate` 提供如下三个构造函数：

- `HibernateTemplate()`。
- `HibernateTemplate(org.hibernate.SessionFactory sessionFactory)`。
- `HibernateTemplate(org.hibernate.SessionFactory sessionFactory, boolean allowCreate)`。

第一个构造函数：构造一个默认的 `HibernateTemplate` 实例，因此，使用 `HibernateTemplate` 实例之前，还必须使用方法 `setSessionFactory(SessionFactory sessionFactory)` 来为 `HibernateTemplate` 传入 `SessionFactory` 的引用。

第二个构造函数：在构造时已经传入 `SessionFactory` 引用。

第三个构造函数：其 `boolean` 型参数表明，如果当前线程已经存在一个非事务性的 `Session`，是否直接返回此非事务性的 `Session`。

对于在 Web 应用中，通常启动时自动加载 `ApplicationContext`, `SessionFactory` 和 DAO

对象都处在 Spring 上下文管理下。因此无须在代码中显式设置，可采用依赖注入解耦 SessionFactory 和 DAO，其依赖关系可通过配置文件来设置，如下所示：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
    <!-- 定义数据源，该 bean 的 ID 为 dataSource-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!-- 指定数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>
        <!-- 指定连接数据库的 URL-->
        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</value></property>
        <!-- root 为数据库的用户名-->
        <property name="username"><value>root</value></property>
        <!-- pass 为数据库密码-->
        <property name="password"><value>pass</value></property>
    </bean>
    <!-- 定义 Hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- 依赖注入数据源，注入正是上文定义的 dataSource-->
        <property name="dataSource"><ref local="dataSource"/></property>
        <!-- mappingResources 属性用来列出全部映射文件-->
        <property name="mappingResources">
            <list>
                <!-- 以下用来列出所有的 PO 映射文件-->
                <value>lee/Person.hbm.xml</value>
            </list>
        </property>
        <!-- 定义 Hibernate 的 SessionFactory 的属性 -->
        <property name="hibernateProperties">
            <props>
                <!-- 指定 Hibernate 的连接方言-->
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <!-- 不同数据库连接，启动时选择 create, update, create-drop-->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
    <!-- 配置 Person 持久化类的 DAO bean-->
    <bean id="personDao" class="lee.PersonDaoImpl">
        <!-- 采用依赖注入来传入 SessionFactory 的引用-->
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>
</beans>
```

在 DAO 实现类中，可采用更简单的方式来取得 HibernateTemplate 的实例。代码如下：

```
public class PersonDaoImpl implements PersonDao
{
```

```
//执行持久化操作的 HibernateTemplate
HibernateTemplate ht;
private SessionFactory sessionFactory;
//依赖注入 SessionFactory 的必需的 setter 方法
public void setSessionFactory(SessionFactory sessionFactory)
{
    this.sessionFactory = sessionFactory;
}
//该方法用于完成 HibernateTemplate 的初始化
private void setHibernateTemplate()
{
    if (ht == null)
    {
        ht = new HibernateTemplate(sessionFactory);
    }
}
/**
 * 加载 Person 实例
 * @param id 需要加载 Person 实例的主键值
 * @return 返回加载的 Person 实例
 */
public Person get(int id)
{
    setHibernateTemplate();
    return (Person)ht.get(Person.class, new Integer(id));
}
/**
 * 保存 Person 实例
 * @param person 需要保存的 Person 实例
 */
public void save(Person person)
{
    setHibernateTemplate();
    ht.save(person);
}
/**
 * 修改 Person 实例
 * @param person 需要修改的 Person 实例
 */
public void update(Person person)
{
    setHibernateTemplate();
    ht.update(person);
}
/**
 * 删除 Person 实例
 * @param id 需要删除的 Person id
 */
public void delete(int id)
{
    setHibernateTemplate();
    ht.delete(ht.get(Person.class, new Integer(id)));
}
/**
 * 删除 Person 实例
 * @param person 需要删除的 Person 实例
 */
public void delete(Person person)
{
```

```

        setHibernateTemplate();
        ht.delete(person);
    }
    /**
     * 根据用户名查找 Person
     * @param name 用户名
     * @return 用户名对应的全部用户
     */
    public List findByPerson(String name)
    {
        setHibernateTemplate();
        return ht.find("from Person p where p.name like ?" , name);
    }
    /**
     * 返回全部的 Person 实例
     * @return 全部的 Person 实例
     */
    public List findAllPerson()
    {
        setHibernateTemplate();
        return ht.find("from Person ");
    }
}

```

### 6.4.1 HibernateTemplate 的常规用法

HibernateTemplate 提供了非常多的常用方法来完成基本的操作，比如增加、删除、修改及查询等操作，Spring 2.0 更增加对命名 SQL 查询的支持，也增加对分页的支持。大部分情况下，使用 Hibernate 的常规用法，就可完成大多数 DAO 对象的 CRUD 操作。下面是 HibernateTemplate 的常用方法。

- void delete(Object entity): 删除指定持久化实例。
- deleteAll(Collection entities): 删除集合内全部持久化类实例。
- find(String queryString): 根据 HQL 查询字符串来返回实例集合。
- findByNamedQuery(String queryName): 根据命名查询返回实例集合。
- get(Class entityClass, Serializable id): 根据主键加载特定持久化类的实例。
- save(Object entity): 保存新的实例。
- saveOrUpdate(Object entity): 根据实例状态，选择保存或者更新。
- update(Object entity): 更新实例的状态，要求 entity 是持久状态。
- setMaxResults(int maxResults): 设置分页的大小。

下面是一个完整 DAO 类的源代码：

```

public class PersonDAOImpl implements PersonDAO
{
    //采用 log4j 来完成调试时的日志功能
    private static Log log = LogFactory.getLog(NewsDAOHibernate.class);
    //以私有的成员变量来保存 SessionFactory。
    private SessionFactory sessionFactory;
    //以私有变量的方式保存 HibernateTemplate
    private HibernateTemplate hibernateTemplate = null;
    //设值注入 SessionFactory 必需的 setter 方法
}

```

```

public void setSessionFactory(SessionFactory sessionFactory)
{
    this.sessionFactory = sessionFactory;
}
//初始化本 DAO 所需的 HibernateTemplate
public IHibernateTemplate getHibernateTemplate()
{
//首先，检查原来的 hibernateTemplate 实例是否还存在
if ( hibernateTemplate == null)
{
    //如果不存在，新建一个 HibernateTemplate 实例
    hibernateTemplate = new HibernateTemplate(sessionFactory);
}
return hibernateTemplate;
}
//返回全部人的实例
public List getPersons()
{
    //通过 HibernateTemplate 的 find 方法返回 Person 的全部实例
    return getHibernateTemplate().find("from Person");
}
/**
 * 根据主键返回特定实例
 * @ param 特定主键对应的 Person 实例
 * @ param 主键值
public News getNews(int person id)
{
    return (Person)getHibernateTemplate().get(Person.class,
        new Integer(person id));
}
/**
 * @ param 需要保存的 Person 实例
 */
public void savePerson(Person person)
{
    getHibernateTemplate().saveOrUpdate(person);
}
/**
 * @ param personid 需要删除 Person 实例的主键
 */
public void removePerson(int person id)
{
    //先加载特定实例
    Object p = getHibernateTemplate().load(Person.class,
        new Integer(person id));
    //删除特定实例
    getHibernateTemplate().delete(p);
}
}

```

## 6.4.2 Hibernate 的复杂用法 HibernateCallback

HibernateTemplate 还提供了一种更加灵活的方式来操作数据库，通过这种方式可以完全使用 Hibernate 的操作方式。HibernateTemplate 的灵活访问方式是通过如下两个方法完成的：

- Object execute(HibernateCallback action)。
- List execute(HibernateCallback action)。

这两个方法都需要一个 HibernateCallback 的实例，该实例可在任何有效的 Hibernate 数据访问中使用。程序开发者通过 HibernateCallback，可以完全使用 Hibernate 灵活的方式来访问数据库，解决了 Spring 封装 Hibernate 后灵活性不足的缺陷。HibernateCallback 是一个接口，该接口只有一个方法 doInHibernate(org.hibernate.Session session)，该方法只有一个参数 Session。

通常，程序中采用实现 HibernateCallback 的匿名内部类来获取 HibernateCallback 的实例，方法 doInHibernate 的方法体就是 Spring 执行的持久化操作。具体代码如下：

```
public class PersonDaoImpl implements PersonDao
{
    //私有实例变量保存 SessionFactory
    private SessionFactory sessionFactory;
    //依赖注入必需的 setter 方法
    public void setSessionFactory(SessionFactory sessionFactory)
    {
        this.sessionFactory = sessionFactory;
    }
    /**
     * 通过人名查找所有匹配该名的 Person 实例
     * @param name 匹配的人名
     * @return 匹配该任命的全部 Person 集合
     */
    public List findPersonsByName(final String name)
    {
        //创建 HibernateTemplate 实例
        HibernateTemplate hibernateTemplate =
            new HibernateTemplate(this.sessionFactory);
        //返回 HibernateTemplate 的 execute 的结果
        return (List) hibernateTemplate.execute(
            //创建匿名内部类
            new HibernateCallback()
            {
                public Object doInHibernate(Session session)
                    throws Hibernate Exception
                {
                    //使用条件查询的方法返回
                    List result = session.createCriteria(Person.class)
                        .add(Restrictions.like("name", name+"%")).list();
                    return result;
                }
            });
    }
}
```

注意：在方法 doInHibernate 内可以访问 Session，该 Session 对象是绑定到该线程的 Session 实例，该方法内的持久层操作与不使用 Spring 时的持久层操作完全相同，这保证了在对于复杂的持久层访问时，依然可以使用 Hibernate 的访问方式。

## 6.5 Hibernate 的 DAO 实现

DAO 是 J2EE 应用的重要组件，它隐藏了底层的数据库访问细节。DAO 层也是 J2EE 应用分层中的重要分层，该层向上提供通用的数据访问接口。

通过 DAO 组件，可实现业务逻辑和数据库访问的分离，避免业务逻辑与具体的数据库访问实现耦合。对于 J2EE 应用而言，数据库是相对稳定的部分，其 DAO 组件依赖于数据库系统，提供数据库访问的接口，只要数据库没有重构，则 DAO 层通常无须改写。

DAO 层也分隔了数据库与业务逻辑层，使业务逻辑层更加专注于业务逻辑的实现，而无须理会持久层访问实现。

### 6.5.1 DAO 模式简介

DAO 模式是 J2EE 设计模式中的一种核心设计模式。图 6.1 是 DAO 模式的类图，该类图表示了 DAO 模式的各个参与者之间的关系。

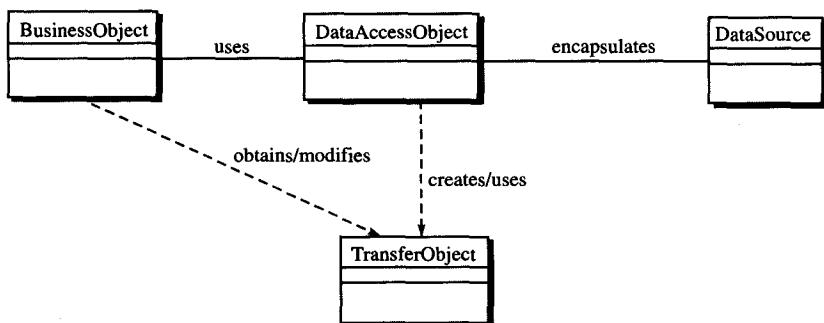


图 6.1 DAO 模式的类图

图 6.2 是 DAO 模式的顺序图，该顺序图则显示模式的各个参与者是如何互动的。

DAO 模式可以提供更好的解耦，将业务逻辑层与持久层访问技术分离，使业务逻辑层无须关注底层数据库访问的实现。使用 DAO 模式主要有如下优势。

- DAO 模式可抽象出数据访问方式，在 BO 访问数据源时，完全感觉不到数据源的存在。软件工程里面有一条很重要的法则，就是一个对象对其他对象的了解越少越好，了解越少就意味着依赖越少，可复用性越高。
- DAO 将数据访问集中在独立的一层。因为所有的数据访问都由 DAO 代理，这层独立的 DAO 就将数据访问的实现与系统的其余部分剥离，将数据访问集中使得系统更具可维护性。
- DAO 还降低了 BO 层的复杂程度。由 DAO 管理复杂的数据访问，从而简化了 BO。所有与数据访问实现有关的代码（如 SQL 语言等）都不用写在 BO 里，从而使

BO 可以集中精力处理业务逻辑，提高了代码的可读性和生产率。

- DAO 还有助于提升系统的可移植性。独立的 DAO 层使得系统能在不同的数据库之间轻易切换，底层的数据库实现对于 BO 来说是不可见的。数据移植时影响的仅仅是 DAO 层，切换不同的数据库并不会影响 BO，因此提高了系统的可复用性。

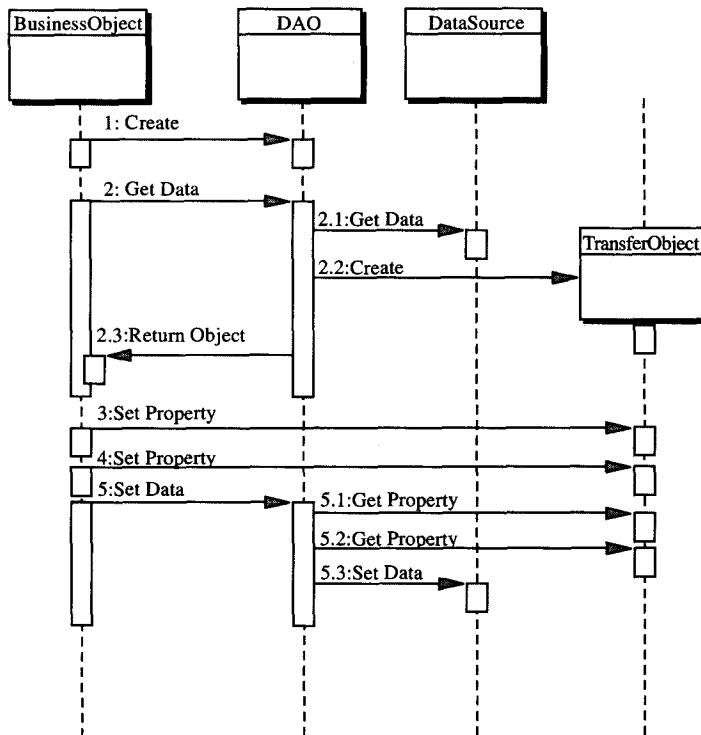


图 6.2 DAO 模式的顺序图

Spring 对 Hibernate 的 DAO 实现提供了良好的支持，主要有如下两种方式：

- 继承 `HibernateDaoSupport` 的实现 DAO。
- 基于 Hibernate 3.0 实现 DAO。

不管采用哪一种实现，这种 DAO 对象都能极好地融合到 Spring 的 `ApplicationContext` 中，遵循依赖注入模式，从而提高解耦。

### 6.5.2 继承 `HibernateDaoSupport` 实现 DAO

Spring 为 Hibernate 的 DAO 提供了工具类——`HibernateDaoSupport`。该类主要提供如下两个方法来方便 DAO 的实现：

- `public final HibernateTemplate getHibernateTemplate()`。
- `public final void setSessionFactory(SessionFactory sessionFactory)`。

其中，`setSessionFactory` 方法用来接收 Spring 的 `ApplicationContext` 依赖注入，可接收配置在 Spring 的 `SessionFactory` 实例；`getHibernateTemplate` 方法则用来根据刚才的

SessionFactory 产生 Session，最后生成 HibernateTemplate 来完成数据库访问。

下面的代码是对利用 HibernateDaoSupport 的 DAO 实现：

```
public class PersonDaoImpl implements PersonDao
{
    /**
     * 加载 Person 实例
     * @param id 需要加载的 Person 实例的主键值
     * @return 返回加载的 Person 实例
     */
    public Person get(int id)
    {
        setHibernateTemplate();
        return (Person)getHibernateTemplate().get(Person.class,
            new Integer(id));
    }
    /**
     * 保存 Person 实例
     * @param person 需要保存的 Person 实例
     */
    public void save(Person person)
    {
        setHibernateTemplate();
        getHibernateTemplate().save(person);
    }
    /**
     * 修改 Person 实例
     * @param person 需要修改的 Person 实例
     */
    public void update(Person person)
    {
        setHibernateTemplate();
        getHibernateTemplate().update(person);
    }
    /**
     * 删除 Person 实例
     * @param id 需要删除的 Person id
     */
    public void delete(int id)
    {
        setHibernateTemplate();
        getHibernateTemplate().delete(getHibernateTemplate().
            get(Person.class, new Integer(id)));
    }
    /**
     * 删除 Person 实例
     * @param person 需要删除的 Person 实例
     */
    public void delete(Person person)
    {
        setHibernateTemplate();
        getHibernateTemplate().delete(person);
    }
    /**
     * 根据用户名查找 Person
     * @param name 用户名
     * @return 用户名对应的全部用户
     */
}
```

```
public List findByPerson(String name)
{
    setHibernateTemplate();
    return getHibernateTemplate().find("from Person p where p.name like ?" ,
                                         name);
}
/***
 * 返回全部的 Person 实例
 * @return 全部的 Person 实例
 */
public List findAllPerson()
{
    setHibernateTemplate();
    return getHibernateTemplate().find("from Person ");
}
}
```

与前面的 PersonDAOImpl 对比，会发现其代码大大减少。事实上，DAO 的实现依然借助于 HibernateTemplate 的模板访问方式。该 DAO 的配置必须依赖于 SessionFactory，具体的配置如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
    <!--定义数据源,该 bean 的 ID 为 dataSource-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
    DriverManagerDataSource">
        <!-- 指定数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver
        </value></property>
        <!-- 指定连接数据库的 URL-->
        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</
        value></property>
        <!-- root 为数据库的用户名-->
        <property name="username"><value>root</value></property>
        <!-- pass 为数据库密码-->
        <property name="password"><value>pass</value></property>
    </bean>
    <!--定义 Hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
    LocalSessionFactoryBean">
        <!-- 依赖注入数据源, 注入上文定义的 dataSource-->
        <property name="dataSource"><ref local="dataSource" /></property>
        <!-- mappingResources 属性用来列出全部映射文件-->
        <property name="mappingResources">
            <list>
                <!--以下用来列出所有的 PO 映射文件-->
                <value>lee/Person.hbm.xml</value>
            </list>
        </property>
        <!--定义 Hibernate 的 SessionFactory 的属性 -->
        <property name="hibernateProperties">
            <props>
                <!-- 指定 Hibernate 的连接方法-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
```

```

        MySQLDialect</prop>
        <!-- 不同数据库连接, 启动时选择 create, update, create-drop-->
        <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
</property>
</bean>
<!-- 配置 Person 持久化类的 DAO bean-->
<bean id="personDAO" class="lee. PersonDAOHibernate">
    <!-- 采用依赖注入来传入 SessionFactory 的引用>
    <property name="sessionFactory"><ref local="sessionFactory"/></
property>
</bean>
</beans>

```

程序中可以通过显式的编码来获得 personDAO bean, 然后执行 CRUD 操作。也可通过依赖注入, 将 personDAO 的实例注入其他 bean 属性, 再执行 CRUD 操作。

在继承 HibrmateDaoSupport 的 DAO 实现里, Hibermate Session 的管理完全不需要 Hibermate 代码打开, 而由 Spring 来管理。Spring 会根据实际的操作, 采用“每次事务打开一次 session”的策略, 自动提高了数据库访问的性能。

### 6.5.3 基于 Hibermate 3.0 实现 DAO

Hibermate 3.0 提供了一种新的技术: contextual Sessions。通过此机制, Hibermate 可以自己管理 Session, 从而保证“每次事务打开一个 Session”。该机制类似于 Spring 的同步策略。

Hibermate 的 contextual Sessions, 是通过 SessionFactory 的 getCurrentSession()方法实现的。该方法会返回由当前 JTA 事务保持的 Session, 如果当前 JTA 事务关联的 Session 不存在, 则系统打开一次新的 Session, 并关联到当前的 JTA 事务; 如果当前 JTA 事务关联的 Session 已经存在, 则直接返回该 Session。执行该操作的前提是 Hibermate 处于事务管理下。通常, Spring 为 Hibermate 提供事务管理。

基于 Hibermate 3.0 的 DAO 的实现, 只需将 Spring 注入 SessionFactory, 然后由 Hibermate 自己管理 Session。即通过 SessionFactory 的 getCurrentSession 方法, 返回当前事务关联的 Session。持久化操作在 Session 管理下如常进行。完整的基于 Hibermate 3.0 实现 DAO 的代码如下:

```

public class PersonDaoImpl implements PersonDao
{
    //私有成员变量保存 SessionFactory
    private SessionFactory sessionFactory;
    /**
     * 依赖注入 SessionFactory 必需的 setter 方法
     * @ sessionFactory
     */
    public void setSessionFactory(SessionFactory sessionFactory)
    {
        this.sessionFactory = sessionFactory;
    }
}

```

```
    /**
     * 根据名字查找 Person 的实例。
     * @param name 需要查找 Person 的名字
     * @return 匹配名字的 Person 实例的集合
     */
    public Collection findPersonsByName(String name)
    {
        return this.sessionFactory.getCurrentSession()
            .createQuery("from lee.Person p where p.name=?")
            .setParameter(0, name)
            .list();
    }
    /**
     * 根据 Person id 加载 Person 实例。
     * @param id 需要 load 的 Person 实例
     * @return 特定 id 的 Person 实例。
     */
    public Person findPersonsById(int id)
    {
        return (Person)this.sessionFactory.getCurrentSession()
            .load(Person.class,new Integer(id));
    }
}
```

该 DAO 的数据库访问方式类似于传统的 Hibernate 的访问，区别在于获取 Session 的方式不同。传统的 Hibernate 的 SessionFactory，采用工具类 HibernateUtils 来保存成静态成员变量，每次采用 HibernateUtils 打开 Session。

传统的 Session 访问方式，很容易造成“每次数据库操作打开一次 Session”，使该方式效率低下，也是 Hibernate 不推荐采用的策略。基于该原因，Hibernate 推荐采用“每次事务打开一次 Session”。Hibernate 3.0 提供 contextual Sessions 的技术，最终达到与继承 HibernateDaoSupport 的 DAO 实现相同的目的。

同样，此 DAO bean 也需要配置在 Spring 的上下文中，需要依赖于 SessionFactory bean。SessionFactory bean 由 Spring 在运行时动态地为 DAO bean 注入。具体的配置文件，读者可参考上文的配置文件。

## 6.6 事务管理

事务是信息化系统的最基本功能，基本的 CRUD 操作没有任何逻辑意义。一次业务逻辑操作往往需要具有原子性：典型的两个账户转账的情形，必然涉及两次数据库操作，这两次操作必须处于事务中，而不能只进行一半。

Hibernate 建议所有的数据库访问都应放在事务内进行，即使只进行只读操作。事务又应该尽可能短，因为长事务会导致长时间无法释放表内行级锁，从而降低系统并发的性能。Spring 同时支持编程式事务和声明式事务，尽量考虑使用声明式事务，因为声明式事务管理可分离业务逻辑和事务管理逻辑，具备良好的适应性。

## 6.6.1 编程式的事务管理

另外，编程式事务提供了 `TransactionTemplate` 模板类，该类可以大大减少事务操作的代码。因此 `TransactionTemplate` 采用 `Callback` 避免让开发者重复书写其打开事务、提交事务及回滚事务等代码，同时 `TransactionTemplate` 无须书写大量的 `try...catch` 块。

`HibernateTemplate` 必须提供 `PlatformTransactionManager` 实例。该实例既可以在代码中手动设置，也可以使用 Spring 的依赖注入。总之，只要获取了 `PlatformTransactionManager` 引用，`TransactionTemplate` 就可以完成事务操作。

使用 `TransactionTemplate` 不需要显式地开始事务，甚至不需要显式地提交事务。这些步骤都由模板完成。但出现异常时，应通过 `TransactionStatus` 的 `setRollbackOnly` 显式回滚事务。

`TransactionTemplate` 的 `execute` 方法接收一个 `TransactionCallback` 实例。`Callback` 也是 Spring 的经典设计，用于简化用户操作，`TransactionCallback` 包含如下方法。

- `Object doInTransaction(TransactionStatus status)`。

该方法的方法体就是事务的执行体。

如果事务的执行体没有返回值，则可以使用 `TransactionCallbackWithoutResult` 类的实例。这是个抽象类，不能直接实例化，只能用于创建匿名内部类。它也是 `TransactionCallback` 接口的子接口，该抽象类包含一个抽象方法：

- `void doInTransactionWithoutResult(TransactionStatus status)`

该方法与 `doInTransaction` 的效果非常相似，区别在于该方法没有返回值，即事务执行体无须返回值。

在下面的示例中，`PlatformTransactionManager` 实例采用适用于 `Hibernate` 的事务管理器来实现类 `HibernateTransactionManager`，该实现类是个局部事务管理器，容器中仅仅部署了该事务管理器 bean，因此应在代码中手动为 `TransactionTemplate` 注入事务管理器 bean。下面是 `Hibernate` 局部事务管理的配置文件的源代码：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的 DTD 定义-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素是 beans-->
<beans>
    <!-- 定义数据源，该 bean 的 ID 为 dataSource-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
        DriverManagerDataSource">
        <!-- 指定数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</
        value></property>
        <!-- 指定连接数据库的 URL-->
        <property name="url"><value>jdbc:mysql://wonder:3306/j2ee</
        value></property>
        <!-- root 为数据库的用户名-->
```

```

<property name="username"><value>root</value></property>
<!-- pass 为数据库密码-->
<property name="password"><value>pass</value></property>
</bean>
<!--定义 Hibernate 的 SessionFactory-->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
    <!-- 依赖注入数据源，注入上文定义的 dataSource-->
    <property name="dataSource"><ref local="dataSource"/></property>
    <!-- mappingResources 属性用来列出全部映射文件-->
    <property name="mappingResources">
        <list>
            <!--以下用来列出所有的 PO 映射文件-->
            <value>lee/MyTest.hbm.xml</value>
        </list>
    </property>
    <!--定义 Hibernate 的 SessionFactory 的属性 -->
    <property name="hibernateProperties">
        <props>
            <!-- 指定 Hibernate 的连接方法-->
            <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDialect</prop>
            <!-- 不同数据库连接，启动时选择 create,update,create-drop-->
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
<!-- 配置 Hibernate 的事务管理器 -->
<!-- 使用 HibernateTransactionManager 类，该类是 PlatformTransactionManager 接口
     针对采用 Hibernate 持久化连接的特定实现。-->
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- HibernateTransactionManager bean 需要依赖注入一个
SessionFactor bean 的引用-->
    <property name="sessionFactory"><ref
local="sessionFactory"/></property>
</bean>
</beans>

```

下面是采用 TransactionTemplate 和 HibernateTemplate 的事务操作代码：

```

public class TransactionTest
{
    public static void main(String[] args)
    {
        //因为并未在 Web 应用中测试，故需要手动创建 Spring 的上下文
        final ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //获得 Spring 上下文的事务管理器
        PlatformTransactionManager transactionManager=
            (PlatformTransactionManager)ctx.getBean("transactionManager");
        final SessionFactory sessionFactory =
            (SessionFactory)ctx.getBean("sessionFactory");
        //以事务管理器实例为参数，创建 TransactionTemplate 对象
        TransactionTemplate tt = new TransactionTemplate(transactionManager);
        //设置 TransactionTemplate 的事务传播属性
        tt.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        //执行 TransactionTemplate 的 execute 方法，该方法需要 TransactionCallback 实例
    }
}

```

```
tt.execute(new TransactionCallbackWithoutResult()
    //采用 TransactionCallbackWithoutResult 匿名内部类的形式执行
    {
        protected void doInTransactionWithoutResult(TransactionStatus ts)
        {
            try
            {
                //以 SessionFactory 实例为参数创建 HibernateTemplate
                HibernateTemplate hibernateTemplate =
                    new HibernateTemplate(sessionFactory);
                MyTestp1 = new MyTest ("Jack");
                //保存第一个实例
                hibernateTemplate.save(p1);
                //让下面的数据库操作抛出异常即可看出事务效果。前面的操作也
                //不会生效
                MyTestp2 = new MyTest ("Jack");
                //保存第二个实例，可将 Person 的 name 属性设为标识属性，并
                //引起主键重复的异常，可看出前一条记录也不会加入数据库中
                hibernateTemplate.save(p2);

            }
            catch (Exception e)
            {
                ts.setRollbackOnly();
            }
        }
    });
}
```

查看数据库的 mytable 表，该表中没有任何记录（如果没有采用事务，第一条记录应该可以进去。而两次保存记录放在 doInTransactionWithoutResult 方法中执行），因为该方法的方法体具有事务性，该方法的数据库操作要么全部生效，要么全部失效。由于第二条记录违反了数据库的主键约束，因此，记录全部失效。

## 6.6.2 声明式事务管理

通常建议采用声明式事务管理。声明式事务管理的优势非常明显，代码中无须关注事务逻辑，由 Spring 声明式事务管理负责事务逻辑；声明式事务管理无须与具体的事务逻辑耦合，可以方便地在不同事务逻辑之间切换。

声明式事务管理的配置方式通常有如下四种。

- 使用 TransactionProxyFactoryBean 为目标 bean 生成事务代理的配置。此方式最传统，但配置文件臃肿，难以阅读。
- 采用 bean 继承的事务代理配置方式比较简洁，但依然是增量式配置。
- 使用 BeanNameAutoProxyCreator，根据 bean name 自动生成事务代理的方式，这是直接利用 Spring 的 AOP 框架配置事务代理的方式，需要对 Spring 的 AOP 框架有所理解，但这种方式避免了增量式配置，效果非常不错。
- DefaultAdvisorAutoProxyCreator：这也是直接利用 Spring 的 AOP 框架配置事务代理的方式，效果也非常不错，只是这种配置方式的可读性不如第三种方式。

## 1. 利用 TransactionProxyFactoryBean 生成事务代理

采用这种方式的配置时，其配置文件的增加非常快，每个 bean 有需要两个 bean 配置一个目标，另外还需要使用 TransactionProxyFactoryBean 配置一个代理 bean。

这是一种最原始的配置方式，下面是使用 TransactionProxyFactoryBean 的配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义数据源-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
        <!-- 定义数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</
value></property>
        <!-- 定义数据库 url-->
        <property name="url"><value>jdbc:mysql://localhost:3306/spring</
value></property>
        <!-- 定义数据库用户名-->
        <property name="username"><value>root</value></property>
        <!-- 定义数据库密码-->
        <property name="password"><value>32147</value></property>
    </bean>
    <!-- 定义一个 hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <!-- 定义 SessionFactory 必须注入 DataSource-->
        <property name="dataSource"><ref local="dataSource"/></property>
        <property name="mappingResources">
            <list>
                <!--以下用来列出所有的 PO 映射文件-->
                <value>Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <!-- 此处用来定义 hibernate 的 SessionFactory 的属性:
                    不同数据库连接启动时选择 create,update,create-drop-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
    <!-- 定义事务管理器，适用于 Hibernete 的事务管理器-->
    <bean id="transactionManager"
 class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <!-- HibernateTransactionManager bean 需要依赖注入一个
SessionFactor bean 的引用-->
        <property name="sessionFactory"><ref local="sessionFactory"/></
property>
    </bean>
    <!-- 定义 DAO Bean，作为事务代理的目标-->
    <bean id="personDaoTarget" class="lee.PersonDaoHibernate">
        <!-- 为 DAO bean 注入 SessionFactor 引用-->
```

```

        <property name="sessionFactory"><ref local="sessionFactory"/></
property>
    </bean>
    <!-- 定义 DAO bean 的事务代理-->
    <bean id="personDao" class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean">
        <!-- 为事务代理 bean 注入事务管理器-->
        <property name="transactionManager"><ref bean="transactionManager"
/></property>
        <!-- 设置事务属性-->
        <property name="transactionAttributes">
            <props>
                <!-- 所有以 find 开头的方法，采用 required 的事务策略，并且只读-->
                <prop key="find*>">PROPAGATION_REQUIRED,readOnly</prop>
                <!-- 其他方法，采用 required 的事务策略 -->
                <prop key="*>">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
        <!-- 为事务代理 bean 设置目标 bean -->
        <property name="target">
            <ref local="personDaoTarget" />
        </property>
    </bean>
</beans>

```

在上面的配置文件中，personDao 需要配置两个部分：一个是 personDao 的目标 bean，该目标 bean 是实际 DAO bean，以实际的 DAO bean 为目标，建立事务代理；另一个是组件，需要一个目标 bean 和一个事务代理来组成。

但这种配置方式还有一个缺点：目标 bean 直接暴露在 Spring 容器中，可以直接引用，如果目标 bean 被误引用，将导致业务操作不具备事务性。

为了避免这种现象，可将目标 bean 配置成嵌套 bean，下面是目标 bean 和事务代理的配置代码：

```

<!-- 定义 DAO bean 的事务代理-->
<bean id="personDao" class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean">
    <!-- 为事务代理 bean 注入事务管理器-->
    <property name="transactionManager"><ref bean="transactionManager"
/></property>
    <!-- 设置事务属性-->
    <property name="transactionAttributes">
        <props>
            <!-- 所有以 find 开头的方法，采用 required 的事务策略，并且只读-->
            <prop key="find*>">PROPAGATION_REQUIRED,readOnly</prop>
            <!-- 其他方法，采用 required 的事务策略 -->
            <prop key="*>">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
    <!-- 为事务代理 bean 设置目标 bean -->
    <property name="target">
        <!-- 采用嵌套 bean 配置目标 bean-->
        <bean class="lee.PersonDaoHibernate" />
        <!-- 为 DAO bean 注入 SessionFactory 引用-->
        <property name="sessionFactory"><ref local="sessionFactory"
/></property>
    </property>
</beans>

```

```
</bean>
</property>
</bean>
```

## 2. 利用继承简化配置

大部分情况下，每个事务代理的事务属性大同小异，事务代理的实现类都是 TransactionProxyFactoryBean，事务代理 bean 都必须注入事务管理器。

对于这种情况，Spring 提供了 bean 与 bean 之间的继承，可以简化配置。将大部分通用的配置，配置成事务模板，而实际的事务代理 bean，则继承事务模板。这种配置方式可以减少部分配置代码。下面是采用继承的配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义数据源-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
        <!-- 定义数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</
value></property>
        <!-- 定义数据库 url-->
        <property name="url"><value>jdbc:mysql://localhost:3306/spring</
value></property>
        <!-- 定义数据库用户名-->
        <property name="username"><value>root</value></property>
        <!-- 定义数据库密码-->
        <property name="password"><value>32147</value></property>
    </bean>
    <!-- 定义一个 hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <!-- 定义 SessionFactory 必须注入 DataSource-->
        <property name="dataSource"><ref local="dataSource"/></property>
        <property name="mappingResources">
            <list>
                <!--以下用来列出所有的 PO 映射文件-->
                <value>Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <!-- 此处用来定义 hibernate 的 SessionFactory 的属性：
不同数据库连接启动时选择 create,update,create-drop-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
    <!-- 定义事务管理器，使用适用于 Hibernate 的事务管理器-->
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <!-- HibernateTransactionManager bean 需要依赖注入一个 Session
Factory bean 的引用-->
```

```

<property name="sessionFactory"><ref local="sessionFactory"/></
property>
</bean>
<!-- 配置事务模板，模板 bean 被设置成 abstract bean，保证不会被初始化-->
<bean id="txBase" class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean"
      lazy-init="true" abstract="true">
    <!-- 为事务模板注入事务管理器-->
    <property name="transactionManager"><ref bean="transactionManager"/>
</property>
    <!-- 设置事务属性-->
    <property name="transactionAttributes">
        <props>
            <prop key="find*>PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="*>PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
<!-- 实际的事务代理 bean-->
<bean id="personDao" parent="txBase">
    <!-- 采用嵌套 bean 配置目标 bean -->
    <property name="target">
        <bean class="lee.PersonDaoHibernate">
            <property name="sessionFactory"><ref local="sessionFactory"/></
property>
        </bean>
    </property>
</bean>
</beans>

```

相比前面直接采用 `TransactionProxyFactoryBean` 的事务代理配置方式，这种配置方式可以大大减少配置文件的代码量。每个事务代理的配置都继承事务模板，无须重复指定事务代理的实现类，也无须重复指定事务传播属性。但如果新的事务代理有额外的事务属性，也可指定自己的事务属性，此时，子 bean 的属性覆盖父 bean 的属性。当然每个事务代理 bean 都必须配置自己的目标 bean，这是不可避免的。

从上面的配置可看出，事务代理的配置依然是增量式的，每个事务代理都需要单独配置。

### 3. 用 `BeanNameAutoProxyCreator` 自动创建事务代理

下面介绍一种优秀的事务代理配置策略，采用这种配置策略，完全可以避免增量式配置，使所有的事务代理由系统自动创建。由于容器中的目标 bean 自动消失，可避免需要使用嵌套 bean 来保证目标 bean 不可被访问。

这种配置方式依赖于 Spring 提供的 bean 后处理器，该后处理器用于为每个 bean 自动创建代理，此处的代理不仅可以是事务代理，也可以是任意的代理，只需要有合适的拦截器即可。这些是 AOP 框架的概念，笔者在此处不对 AOP 进行深入介绍。读者只需了解这种事务代理的配置方式即可。

下面是采用 `BeanNameAutoProxyCreator` 配置事务代理的配置文件：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

```

```
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义数据源-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
        <!-- 定义数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.Driver</
value></property>
        <!-- 定义数据库 url-->
        <property name="url"><value>jdbc:mysql://localhost:3306/spring</
value></property>
        <!-- 定义数据库用户名-->
        <property name="username"><value>root</value></property>
        <!-- 定义数据库密码-->
        <property name="password"><value>32147</value></property>
    </bean>
    <!-- 定义一个 hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <!-- 定义 SessionFactory 必须注入 DataSource-->
        <property name="dataSource"><ref local="dataSource"/></property>
        <property name="mappingResources">
            <list>
                <!--以下用来列出所有的 PO 映射文件-->
                <value>Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <!-- 此处用来定义 hibernate 的 SessionFactory 的属性:
                    不同数据库连接启动时选择 create,update,create-drop-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
    <!-- 定义事务管理器, 使用适用于 Hibernate 的事务管理器-->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <!-- HibernateTransactionManager bean 需要依赖注入一个 Session
Factory bean 的引用-->
        <property name="sessionFactory"><ref local="sessionFactory"/></
property>
    </bean>
    <!-- 配置事务拦截器-->
    <bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <!-- 事务拦截器 bean 需要依赖注入一个事务管理器 -->
        <property name="transactionManager" ref="transactionManager"/>
        <property name="transactionAttributes">
            <!-- 下面定义事务传播属性-->
            <props>
                <prop key="insert">PROPAGATION_REQUIRED</prop>
                <prop key="find">PROPAGATION_REQUIRED,readOnly</prop>
                <prop key="*>PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

```

```

</bean>
<!-- 定义 BeanNameAutoProxyCreator, 该 bean 是个 bean 后处理器, 无须被引用, 因此
没有 id 属性
这个 bean 后处理器, 根据事务拦截器为目标 bean 自动创建事务代理
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAuto
ProxyCreator">
    指定对满足哪些 bean name 的 bean 自动生成业务代理 -->
    <property name="beanNames">
        <!-- 下面是所有需要自动创建事务代理的 bean-->
        <list>
            <value>personDao</value>
        </list>
        <!-- 此处可增加其他需要自动创建事务代理的 bean-->
    </property>
    <!-- 下面定义 BeanNameAutoProxyCreator 所需的事务拦截器-->
    <property name="interceptorNames">
        <list>
            <value>transactionInterceptor</value>
            <!-- 此处可增加其他新的 Interceptor -->
        </list>
    </property>
</bean>
<!-- 定义 DAO Bean , 由于 BeanNameAutoProxyCreator 自动生成事务代理-->
<bean id="personDao" class="lee.PersonDaoHibernate">
    <property name="sessionFactory"><ref local="sessionFactory"/></
property>
</bean>
</beans>

```

TranscationInterceptor 是一个事务拦截器 bean, 需要传入一个 TransactionManager 的引用。配置中使用 Spring 依赖注入该属性, 事务拦截器的事务属性通过 transaction Attributes 来指定, 该属性有 props 子元素, 并在配置文件中定义了三个事务传播规则。

所有以 insert 开始的方法, 都采用 PROPAGATION\_REQUIRED 的事务传播规则。当程序抛出 MyException 异常及其子异常时, 会自动回滚事务; 所有以 find 开头的方法, 都采用 PROPAGATION\_REQUIRED 事务传播规则, 并且具有只读性; 其他方法则采用 PROPAGATION\_REQUIRED 的事务传播规则。

BeanNameAutoProxyCreator 是根据 bean 名生成自动代理的代理创建器, 该 bean 通常需要接受两个参数: 第一个是 beanNames 属性, 该属性用来设置哪些 bean 需要自动生成代理, 另一个属性是 interceptorNames, 该属性则指定事务拦截器, 在自动创建事务代理时, 系统会根据这些事务拦截器的属性来生成对应的事务代理。

为了让读者对这种配置方式有信息, 可对 PersonDaoHibernate 的 save 方法进行简单修改, 修改后的 save 方法如下:

```

/**
 * 保存 person 实例
 * @param person 需要保存的 Person 实例
 */
public void save(Person person)
{
    getHibernateTemplate().save(person);
    //下面两行代码没有实际意义, 仅仅为了引发数据库异常
}

```

```

        DataSource ds = null;
        DataSourceUtils.getConnection(ds);
    }
}

```

在主程序中调用该 save 方法，主程序调用 save 方法的代码如下：

```

for (int i = 0 ; i < 10 ; i++ )
{
    //保存 Person 实例
    pdao.save(new Person(String.valueOf(i) , i + 10));
}

```

执行完主程序的该片段后，数据库不会插入任何记录。如果 BeanNameAutoProxyCreator 的配置修改成如下格式：

```

<!-- 定义 BeanNameAutoProxyCreator, 该 bean 是个 bean 后处理器，无须被引用，因此没有 id 属性
     这个 bean 后处理器，根据事务拦截器为目标 bean 自动创建事务代理
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAuto
ProxyCreator">
    指定对满足哪些 bean name 的 bean 自动生成业务代理 -->
    <property name="beanNames">
        <!-- 下面是所有需要自动创建事务代理的 bean-->
        <list>
            <!-- value>personDao</value-->
        </list>
        <!-- 此处可增加其他需要自动创建事务代理的 bean-->
    </property>
    <!-- 下面定义 BeanNameAutoProxyCreator 所需的事务拦截器-->
    <property name="interceptorNames">
        <list>
            <value>transactionInterceptor</value >
            <!-- 此处可增加其他新的 Interceptor -->
        </list>
    </property>
</bean>

```

注意配置文中 beanNames 属性的变化，将所有 personDao 项加注释，即不再为该 bean 生成事务代理。再次执行主程序，此时程序虽然抛出了数据库异常，但数据记录依然被插入数据库。

这种配置方式相当简洁，每次都增加了新的 bean。如果需要该 bean 的方法具有事务性，只需在 BeanNameAutoProxyCreator 的 beanNames 属性下作相应修改。该行告诉 bean 后处理需要为哪个 bean 生成事务代理。

#### 4. 用 DefaultAdvisorAutoProxyCreator 自动创建事务代理

这种配置方式与 BeanNameAutoProxyCreator 自动创建代理的方式非常相似，都是使用 bean 后处理器为目标 bean 创建实物代理。区别是前者使用事务拦截器创建代理；后者需要使用 Advisor 创建事务代理。

事实上，采用 DefaultAdvisorAutoProxyCreator 的事务代理配置方式更加简洁，因为这个代理生成器自动搜索 Spring 容器中的 Advisor，并为容器中所有的 bean 创建代理。

相对前一种方式，这种方式的可读性不如前一种直观，笔者还是推荐采用第三种配置方式，下面是使用 DefaultAdvisorAutoProxyCreator 的配置方式：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义数据源-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
        <!-- 定义数据库驱动-->
        <property name="driverClassName"><value>com.mysql.jdbc.
Driver</value></property>
        <!-- 定义数据库 url-->
        <property name="url"><value>jdbc:mysql://localhost:3306/spring</
value></property>
        <!-- 定义数据库用户名-->
        <property name="username"><value>root</value></property>
        <!-- 定义数据库密码-->
        <property name="password"><value>32147</value></property>
    </bean>
    <!-- 定义一个 hibernate 的 SessionFactory-->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <!-- 定义 SessionFactory 必须注入 DataSource-->
        <property name="dataSource"><ref local="dataSource"/></property>
        <property name="mappingResources">
            <list>
                <!-- 以下用来列出所有的 PO 映射文件-->
                <value>Person.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <!-- 此处用来定义 hibernate 的 SessionFactory 的属性：
                    不同数据库连接启动时选择 create, update, create-drop-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDialect</prop>
                    <prop key="hibernate.hbm2ddl.auto">update</prop>
                </props>
            </property>
        </bean>
        <!-- 定义事务管理器，使用适用于 Hibernate 的事务管理器-->
        <bean id="transactionManager"
            class="org.springframework.orm.hibernate3.HibernateTransaction
Manager">
            <!-- HibernateTransactionManager bean 需要依赖注入一个
SessionFactor bean 的引用-->
            <property name="sessionFactory"><ref local="sessionFactory"/></
property>
        </bean>
        <!-- 配置事务拦截器-->
        <bean id="transactionInterceptor"
            class="org.springframework.transaction.interceptor.
TransactionInterceptor">
            <!-- 事务拦截器 bean 需要依赖注入一个事务管理器 -->
            <property name="transactionManager" ref="transactionManager"/>
            <property name="transactionAttributes">
                <!-- 下面定义事务传播属性-->
                <props>
                    <prop key="insert">PROPAGATION_REQUIRED</prop>

```

```
<prop key="find*>PROPAGATION_REQUIRED,readonly</prop>
      <prop key="*>PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
<!-- 定义事务 Advisor-->
<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <!-- 定义 advisor 时, 必须传入 Interceptor-->
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>
<!-- DefaultAdvisorAutoProxyCreator 搜索容器中的 advisor, 并为每个 bean 创建代理 -->
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
    <!--定义 DAO Bean ,由于 BeanNameAutoProxyCreator 自动生成事务代理-->
    <bean id="personDao" class="lee.PersonDaoHibernate">
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>
</beans>
```

在这种配置方式下，配置文件变得更加简洁，增加目标 bean 时不需要增加任何额外的代码，容器会自动为目标 bean 生成代理。但这种方式的可读性相对较差。

### 6.6.3 事务策略的思考

Spring 的事务管理都是通过 PlatformTransactionManager 完成。在 Hibernate 应用中，PlatformTransactionManager 可能是 Hibernat SessionFactory，也可能是 JtaTransaction Manager。前者是采用局部事务管理的实现；后者是采用基于 JTA 支持的全局事务管理的实现。

采用声明式事务管理，可以使应用的事务策略方便地在不同的事务策略之间切换。使程序的代码可以更加专注于业务逻辑的实现，而无须理会事务逻辑。

因此，即使应用运行于支持 JTA 事务的应用服务器环境，也应考虑使用 Spring 的声明式事务管理。假如需要改变服务器，而新的服务器无法提供 JTA 事务支持，声明式事务可以方便地切换成局部事务（只需对配置文件简单修改）。

## 本章小结

本章主要介绍 Spring 与 Hibernate 的整合：能简化持久层的访问，大大提高了开发效率。

其次重点介绍了 HibernateTemplate 的使用，包括 Spring 对 Hibernate DAO 支持，以及利用 IoC 特性简化 DAO 开发步骤。

最后详细介绍了 Spring 的四种事务配置策略，同时给出笔者关于事务代理配置的建议。

# 第7章

## Spring 与 Struts 的整合

### 本章要点

- 『 Spring 整合第三方 MVC 框架的通用配置
- 『 使用 DelegatingRequestProcessor 的整合
- 『 使用 DelegatingActionProxy 的整合
- 『 使用 ActionSupport 的整合
- 『 整合方式的分析
- 『 实用的整合策略

虽然 Spring 本身提供了一套极其优秀的 MVC 框架，但这套框架的设计过于追求完美，采用了大量的映射策略，如请求到控制器之间的控制器解析策略，逻辑视图和实际视图之间的视图解析策略等；还有过于细化的角色划分，使得 MVC 层的开发相当繁琐。这对于实际应用的开发往往弊大于利，而且 Spring MVC 的开发群体不够活跃，也存在风险。

而采用 Struts 将不存在这些风险，虽然 Struts 不够完美，但拥有极其稳定的表现，经过长时间的检验，有大量成功的应用可以参考。重要的是，其开发群体相当活跃，相关资源相当丰富。

如果出于学习目的，使用 Spring MVC 可能更有吸引力。但对于企业应用的开发，其稳定的性能及可控的开发周期，才是第一考虑的要素。毕竟，使用一种全新的技术，其风险需要全面评估。

下面介绍 Struts 与 Spring 的整合。

## 7.1 Spring 整合第三方 MVC 框架的通用配置

如果需要使用第三方 MVC 框架，则不能在 web.xml 文件中配置 ApplicationContext 的启动。但是，ApplicationContext 是 Spring 的容器，负责管理所有的组件，从业务逻辑层组件到持久层组件，都必须运行在 Spring 容器中。因此，必须在 Web 应用启动时，创建 Spring 的 ApplicationContext 实例。事实上，Spring ApplicationContext 作为 IoC 容器，总是优先加载。

不管采用怎样的方法，Spring 容器都应该在应用启动时自动加载。为了让 Spring 容器能自动加载，通常有以下两种做法：

- 让 MVC 框架负责创建 ApplicationContext 实例，并在 MVC 框架加载时自动创建 Spring 容器。Struts 就是采用这种机制与 Spring 整合。
- 在 web.xml 文件中加载 Spring 容器，这是最常见的做法。Spring 自己的 MVC 框架就是采用这种策略。

关于让 MVC 框架负责创建 ApplicationContext 实例的情况比较多，因为每个 MVC 框架的启动机制有区别，因此加载 ApplicationContext 的方式也各有不同。

对于在 web.xml 配置文件中配置 ApplicationContext 的自动创建有两种策略：

- 利用 ServletContextListener 实现。
- 采用 load-on-startup Servlet 实现。

根据 Servlet 2.3 标准，所有的 ServletContextListener 都会比 Servlet 优先加载，即使是 load-on-startup Servlet。由于 ApplicationContext 实例是 Spring 容器，负责管理应用中所有的组件，包括业务逻辑层组件和持久层组件。因此，应该尽可能早的创建 Spring 容器。

为此，应该优先采用 listener 创建 ApplicationContext。只是，ServletContextListener 是从 Servlet 2.3 才出现的规范。如果使用了不支持 Servlet 2.3 以上的 Web 服务器，则只能放弃 ServletContextListener，而采用 load-on-startup Servlet 策略。

Spring 管理的组件相当多，如果将所有的组件部署在同一个配置文件里，不仅会降低配置文件的可读性，而且还增大修改配置文件时引入错误的可能性，也不符合软件工程“分而治之”的规则。通常推荐将服务层对象、业务逻辑对象及 DAO 对象都存在于互不相同的 Context 中。而表现层对象如 Spring MVC 控制器，则被配置在表现层 Context 中，甚至将某个特定模块的组件部署在单独的 Context 中。

在实际的应用中，Spring 的配置文件通常不只一个，而是按功能被分成多个。好在，所有负责加载 Spring 容器的工具都可同时加载多个配置文件。

### 7.1.1 采用 ContextLoaderListener 创建 ApplicationContext

使用 ContextLoaderListener 创建 ApplicationContext 时，必须使服务器支持 listener，下面这些服务器都是支持 Listener 的，如果使用这些服务器，则可以使用 Context

LoaderListener 创建 ApplicationContext 实例：

- Apache Tomcat 4.x+。
- Jetty 4.x+。
- Resin 2.1.8+。
- Orion 2.0.2+。
- BEA WebLogic 8.1 SP3。

Spring 提供 ServletContextListener 的一个实现类 ContextLoaderListener，该类可以作为 listener 使用，它会在创建时自动查找 WEB-INF/下的 applicationContext.xml 文件。因此，如果只有一个配置文件，并且文件名为 applicationContext.xml，则只需在 web.xml 文件中增加如下代码即可：

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</
listener-class>
</listener>
```

如果有多个配置文件需要载入，则考虑使用<context-param>元素来确定配置文件的文件名。由于 ContextLoaderListener 加载时，会查找名为 contextConfigLocation 的参数。因此，配置 context-param 时参数名字应该是 contextConfigLocation。

带多个配置文件的 web.xml 文件如下：

```
<!-- XML 文件的文件头-->
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- web.xml 文件的 DTD 等信息-->
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- 确定多个配置文件-->
    <context-param>
        <!-- 参数名为 contextConfigLocation -->
        <param-name>contextConfigLocation</param-name>
        <!-- 多个配置文件之间以, 隔开 -->
        <param-value>/WEB-INF/daoContext.xml,/WEB-INF/application
Context.xml</param-value>
    </context-param>
    <!-- 采用 listener 创建 ApplicationContext 实例-->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoader
Listener</listener-class>
    </listener>
</web-app>
```

如果没有 contextConfigLocation 指定配置文件，则 Spring 自动查找 applicationContext.xml 配置文件。如果有 contextConfigLocation，则利用该参数确定的配置文件。该参数指定的一个字符串，Spring 的 ContextLoaderListener 负责将该字符串分解成多个配置文件，逗号“,”、空格“ ” 及分号“;”都可作为字符串的分割符。

如果既没有 applicationContext.xml 文件，也没有使用 contextConfigLocation 参数确定配置文件，或者 contextConfigLocation 确定的配置文件不存在。都将导致 Spring 无法

加载配置文件或无法正常创建 ApplicationContext 实例。

Spring 根据 bean 定义创建 WebApplicationContext 对象，并将其保存在 Web 应用的 ServletContext 中。大部分情况下，应用中的 bean 无须感受到 ApplicationContext 的存在，只要利用 ApplicationContext 的 IoC 即可。

如果需要在应用中获取 ApplicationContext 实例，则可以通过如下方法获取：

```
WebApplicationContext ctx =  
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

下面是采用 Servlet 获取 ApplicationContext 的完整源代码：

```
public class SpringTestServlet extends HttpServlet  
{  
    //Servlet 的响应方法。  
    public void service(HttpServletRequest request,  
        HttpServletResponse response)  
        throws ServletException, java.io.IOException  
    {  
        //获取 Servlet 的 ServletContext 对象  
        ServletContext sc = getServletContext();  
        //使用 WebApplicationContextUtils 类获得 ApplicationContext  
        WebApplicationContext ctx =  
            WebApplicationContextUtils.getWebApplicationContext(sc);  
        //获取 Servlet 的页面输出流  
        PrintWriter out = response.getWriter();  
        //将 ApplicationContext 对象输出  
        out.println(ctx);  
    }  
}
```

可在程序里手动获取 ApplicationContext 对象，然后直接输出到 Servlet 的响应。将看到 ApplicationContext 加载了 web.xml 文件中指定的两个配置文件。

该 Servlet 执行的效果如图 7.1 所示。

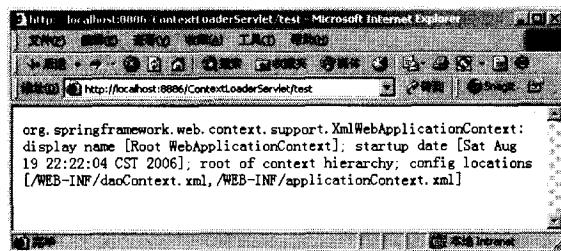


图 7.1 ContextLoaderListener 创建 Spring 容器的测试

## 7.1.2 采用 load-on-startup Servlet 创建 ApplicationContext

如果容器不支持 Listener，则只能使用 load-on-startup Servlet 创建 ApplicationContext 实例，下面的容器都不支持 Listener：

- BEA WebLogic up to 8.1 SP2。

- IBM WebSphere 5.x。
- Oracle OC4J 9.0.3。

Spring 提供了一个特殊的 Servlet 类：ContextLoaderServlet。该 Servlet 在启动时，会自动查找 WEB-INF/下的 applicationContext.xml 文件。

当然，为了让 ContextLoaderServlet 随应用启动而启动，应将此 Servlet 配置成 load-on-startup 的 Servlet。load-on-startup 的值小一点比较合适，因为要保证 ApplicationContext 优先创建。如果只有一个配置文件，并且文件名为 applicationContext.xml，则在 web.xml 文件中增加如下代码即可：

```
<servlet>
    <servlet-name>context</servlet-name>
    <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

该 Servlet 用于提供“后台”服务，作为容器管理应用中的其他 bean，不需要响应客户请求，因此无须配置 servlet-mapping。

如果有多个配置文件，同样可以使用<context-param>元素来确定多个配置文件。事实上，不管是 ContextLoaderServlet，还是 ContextLoaderListener，都依赖于 ContextLoader 创建 ApplicationContext 实例。在 ContextLoader 代码中有如下代码：

```
String configLocation = servletContext.getInitParameter(CONFIG_LOCATION_PARAM);
if (configLocation != null) {
    wac.setConfigLocations(StringUtils.tokenizeToStringArray(configLocation,
        ConfigurableWebApplicationContext.CONFIG_LOCATION_DELIMITERS));
}
```

其中 CONFIG\_LOCATION\_PARAM 是该类的常量，其值为 contextConfigLocation。可看出，ContextLoader 首先检查 servletContext 中是否有 contextConfigLocation 的参数，如果有该参数，则加载该参数指定的配置文件。带多个配置文件的 web.xml 文件如下：

```
<!-- XML 文件的文件头-->
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- web.xml 文件的 DTD 等信息-->
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- 确定多个配置文件-->
    <context-param>
        <!-- 参数名为 contextConfigLocation -->
        <param-name>contextConfigLocation</param-name>
        <!-- 多个配置文件之间以, 隔开 -->
        <param-value>/WEB-INF/daoContext.xml,/WEB-INF/applicationContext.xml</param-value>
    </context-param>
    <!-- 采用 load-on-startup Servlet 创建 ApplicationContext 实例-->
    <servlet>
        <servlet-name>context</servlet-name>
```

```
<servlet-class>org.springframework.web.context.ContextLoader
Servlet</servlet-class>
<!-- 下面值小一点比较合适，会优先加载-->
<load-on-startup>1</load-on-startup>
</servlet>
</web-app>
```

测试所用的 Servlet 与前面所用的没有区别。ContextLoaderServlet 与 ContextLoaderListener 底层都依赖于 ContextLoader。因此，二者的效果几乎没有区别。但它们之间的区别不是它们本身引起的，而是由于 Servlet 2.3 的规范：listener 比 servlet 优先加载。因此，采用 ContextLoaderListener 创建 ApplicationContext 的时机更早。

采用 load-on-startup Servlet 的执行效果如图 7.2 所示：

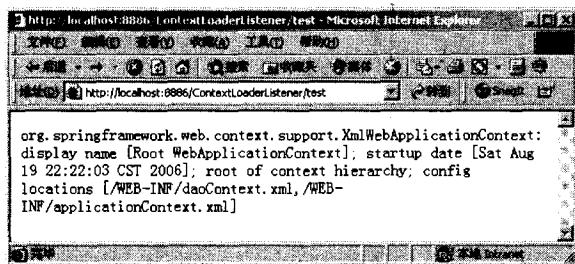


图 7.2 采用 load-on-startup Servlet 的执行效果

当然，也可以通过 ServletContext 的 getAttribute 方法获取 ApplicationContext，使用 WebApplicationContextUtils 类更便捷，因为无须记住 ServletContext 属性名。即使 ServletContext 的 WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE 属性没有对应对象，WebApplicationContextUtils 的 getWebApplicationContext()方法，都将会返回空值，而不会引起异常。

获得了 WebApplicationContext 实例的引用后，可以通过 bean 的名字访问容器中的 bean 实例。但大部分时候，无须通过这种方式访问容器中的 bean。可将表现层的控制器 bean 置入容器的管理中；或将客户端请求直接转发给容器中的 bean；然后由容器管理 bean 之间的依赖。因此，无须手动获取 ApplicationContext 引用。当然，每个框架都会有自己的特定的整合策略。

## 7.2 Spring 与 MVC 框架整合的思考

在介绍 Spring 与 MVC 框架组合之前，如果没有引入 Spring 框架，控制器是如何调用业务逻辑组件的？

在没有引入 Spring 的应用中，控制器显式创建业务逻辑组件，调用业务逻辑组件的方法，并根据业务逻辑方法的返回值确定结果。在实际的应用中，很少见到采用上面的访问策略，因为这是一种非常差的策略。通常会采用服务定位器，或者工厂模式。

实际的业务逻辑组件应该并不是由控制器负责创建，控制器不创建业务逻辑组件至

少有两个原因：

- 每次创建新的业务逻辑组件时，导致性能下降。
- 控制器不应该负责业务逻辑组件的创建，只是业务逻辑组件的使用者，不应该负责其创建实例。

对于采用服务定位器的模式，这种场景是远程访问的场景，其业务逻辑组件已经在某个容器中运行，并对外提供某种服务。控制器无须理会该业务逻辑组件的创建，直接调用即可。但在调用之前，必须先找到该服务——这就是服务定位器的概念。经典 J2EE 应用就是这种结构的应用。

控制器使用服务定位器访问业务逻辑组件的情形如图 7.3 所示。

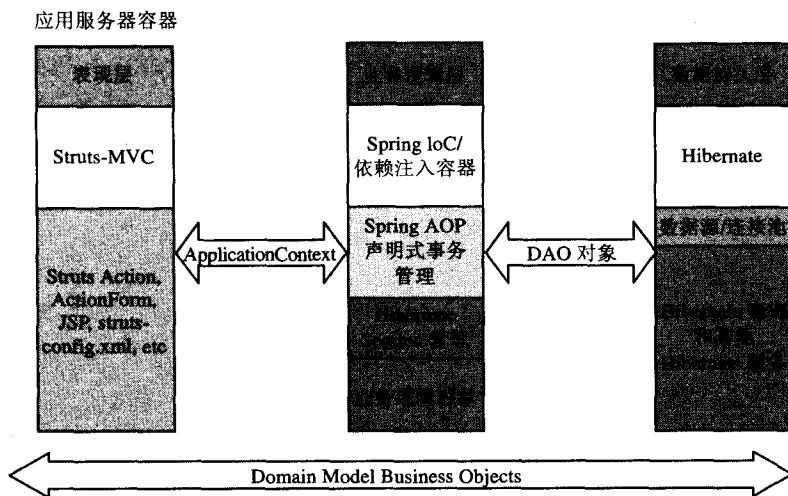


图 7.3 服务定位器策略

对于轻量级的 J2EE 应用，工厂模式则是更实际的策略。因为在轻量级的 J2EE 应用里，业务逻辑组件不是 EJB，通常是一个 POJO，业务逻辑组件的生成通常由工厂负责，而且工厂可以保证该组件的实例只需一个就够了，可以避免重复实例化造成的系统开销。

工厂模式的顺序图如图 7.4 所示。

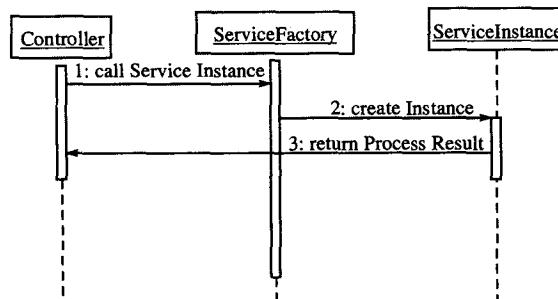


图 7.4 工厂模式顺序图

在采用工厂模式的访问策略中，所有的业务逻辑组件的创建由工厂负责，业务逻辑组件的运行也由工厂负责，而控制器只需定位工厂即可。采用工厂模式，可以提高程序的解耦，将控制器与业务逻辑组件的实现分离。

如果系统采用 Spring 框架，则 Spring 成为最大的工厂。Spring 负责业务逻辑组件的创建和生成，并可管理业务逻辑组件的生命周期。可以这样理解：Spring 是个性能非常优秀的工厂，可以生产出所有的实例，从业务逻辑组件到持久层组件，甚至控制器。

控制器如何访问到 Spring 容器中的业务逻辑组件，是需要解决的问题。为了让 Action 访问 Spring 的业务逻辑组件，有以下两种策略：

- Spring 管理控制器，并利用依赖注入为控制器注入业务逻辑组件。
- 控制器显式定位 Spring 工厂，也就是 Spring 容器的 ApplicationContext 实例，从工厂中获取业务逻辑组件实例的引用。

第一种策略充分利用 Spring 的 IoC 特性，是最优秀的解耦策略。但不可避免带来一些不足之处，归纳起来主要有如下不足之处：

- Spring 管理 Action，必须将所有的 Action 配置在 Spring 容器中，而 struts-config.xml 文件中的配置也不会减少，导致配置文件大量增加。
- Action 的业务逻辑组件接受容器注入，将降低代码的可读性。

总体而言，这种整合策略是利大于弊。

第二种策略与前面介绍的工厂模式并没有太大的不同。区别是：由 Spring 容器充当了业务逻辑组件的工厂。而控制器负责定位 Spring 容器，并通过 Spring 容器访问容器中的业务逻辑组件。这种策略虽然降低了解耦，但提高了程序的可读性。

## 7.3 利用 Spring 的 IoC 特性整合

利用 Spring 的 IoC 特性整合可以更好的解耦，控制器既无须与业务逻辑组件的实现类耦合，也无须与 Spring API 耦合，是一种低侵入式设计。

笔者推荐使用 Spring IoC 容器管理 Struts Action 的方式，因为采用这种方式能充分利用 Spring 依赖注入的优势，无须显式地获取 Spring 的 ApplicationContext 实例。由 Spring IoC 容器管理 Action 也有以下两种方式：

- 使用 DelegatingRequestProcessor。
- 使用 DelegatingActionProxy。

不管采用哪一种方式，都需要随应用启动时创建 ApplicationContext 实例，由 Struts 负责在应用启动时创建 ApplicationContext 实例。根据 3.12.1 节中的内容可知，Struts 中提供了 PlugIn 的扩展点，可在应用启动和关闭时，创建或销毁某些资源。

需要在应用启动时创建 ApplicationContext 实例，这正是 PlugIn 的用武之地。创建 ApplicationContext 实例应采用 Spring 的 ContextLoaderPlugIn 类，该类实现 org.apache.struts.action.PlugIn 接口，用于在启动时加载某个模块。

ContextLoaderPlugin 与 ContextLoaderListener 及 ContextLoaderServlet 的原理相似，

它们都使用 ContextLoader 来创建 ApplicationContext 对象。既可采用 ContextLoaderPlugin 创建 ApplicationContext 实例，也可采用 ContextLoaderListener 创建。但后者创建的 ApplicationContext 实例，将作为前者的父 context。

ContextLoaderPlugin 默认加载的配置文为 servletName-servlet.xml。其中 servletName 是 Struts 的 ActionServlet 对应的 Servlet 名。例如 web.xml 中进行如下定义：

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

ContextLoaderPlugin 默认加载\WEB-INF\actionServlet-servlet.xml，将该文件作为 Spring 的配置文件。因此，如果 Spring 的配置文件只有一个，且文件名为 actionServlet-servlet.xml，则只需在 Struts 配置文件中增加如下代码：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

如果有多个配置文件，或者配置文件的文件名不符合规则，则可以采用“contextConfig Location”属性载入。同样，在多个配置文件之间以“，”隔开。下面是载入多个配置文件配置代码：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation"
        value="/WEB-INF/action-servlet.xml, /WEB-INF/applicationContext.xml"/>
</plug-in>
```

创建了 ApplicationContext 实例后，关键是如何将 ActionServlet 拦截的请求，转发给 Spring 管理的 bean。

由于 Spring IoC 容器不仅管理本身的业务 bean，还负责管理 Struts 的 Action。因此，需要让 ActionServlet 将请求不再转发给 struts-config.xml 配置的 action，而是转发给 ApplicationContext 里配置的 bean。

ActionServlet 将请求转发到 Spring 容器，有以下两个时机。

- 在 ActionServlet 之处将处理转发给 Spring 容器中的 bean。
- 在 Action 之处将处理转发给 Spring 容器中的 bean。

根据这两个时机，完成这个转发也有以下两种策略。

- 采用 DelegatingRequestProcessor，在 ActionServlet 处完成转发。
- 采用 DelegatingActionProxy，在 Action 处完成转发。

### 7.3.1 使用 DelegatingRequestProcessor

查看 Struts 的源代码，我们可以看到由 ActionServlet 调用 RequestProcessor 完成实际的转发。如想在 ActionServlet 处将请求转发给 ApplicationContext 的 bean，可以通过扩展 RequestProcessor 完成，使用扩展的 RequestProcessor 替换 Struts 的 RequestProcessor。

Spring 能完成这种扩展，因为 Spring 提供的 DelegatingRequestProcessor 继承

RequestProcessor。为了让 Struts 使用 DelegatingRequestProcessor，还需要在 struts-config.xml 文件中增加如下代码：

```
//使用 spring 的 RequestProcessor 替换 struts 原有的 RequestProcessor
<controller processorClass="org.springframework.web.struts.Delegating
RequestProcessor"/>
```

完成这个设置后，Struts 会将拦截到的用户请求转发到 Spring context 下的 bean，根据 bean 的 name 属性来匹配。而 Struts 中的 action 配置则无须配置 type 属性，即使配置了 type 属性也没有任何用处，即下面两行配置是完全一样的：

```
//配置 struts action 时候，指定了实现类
<action path="/user" type="lee.UserAction"/>
//配置 struts action 时，没有指定实现类。
<action path="/user"/>
```

对第 3 章的示例程序稍作修改，使其增加客户端验证和程序国际化部分，并调用 Spring 的业务 bean 来验证登录。如下是修改后的 struts-config.xml 文件：

```
<!-- XML 文件版本，编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- struts 配置文件的文件头，包括 dtd 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- struts 配置文件的根元素-->
<struts-config>
    <!-- 配置 formbean，所有的 formbean 都放在 form-beans 元素里定义-->
    <form-beans>
        <!-- 定义了一个 formbean，确定 formbean 名和实现类-->
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 定义 action 部分，所有的 action 都放在 action-mapping 元素里定义-->
    <action-mappings>
        <!-- 这里只定义了一个 action。而且没有指定该 action 的 type 元素-->
        <action path="/login" name="loginForm"
            scope="request" validate="true" input="/login.jsp" >
            <!-- 定义 action 内的两个局部 forward 元素-->
            <forward name="input" path="/login.jsp"/>
            <forward name="welcome" path="/welcome.html"/>
        </action>
    </action-mappings>
    <!-- 使用 DelegatingRequestProcessor 替换 RequestProcessor-->
    <controller processorClass="org.springframework.web.struts.
DelegatingRequestProcessor"/>
    <!-- 加载国际化的资源包-->
    <message-resources parameter="mess"/>
    <!-- 装载验证的资源文件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.
xml,/WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true"/>
    </plug-in>
    <!-- 装载 Spring 配置文件，随应用启动创建 ApplicationContext 实例-->
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
```

```

        value="/WEB-INF/applicationContext.xml,
        /WEB-INF/action-servlet.xml"/>
    </plug-in>
</struts-config>

```

修改后的 struts-config.xml 文件中加载了国际化资源文件，其配置 Struts 的 action 不需要 class 属性，也可完成 ApplicationContext 的创建。

由于程序没有使用 web.xml 文件加载 Spring 容器实例，因此无须增加 ContextLoader 的配置。仅仅增加了 Struts 的标签库配置，主要用于程序国际化等方面。在 Struts 标签库配置中增加如下代码即可：

```

<!-- 配置 bean 标签-->
<taglib>
    <taglib-uri>/tags/struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<!-- 配置 html 标签-->
<taglib>
    <taglib-uri>/tags/struts-html</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<!-- 配置 logic 标签-->
<taglib>
    <taglib-uri>/tags/struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>

```

Struts 的 plug-in 配置部分明确指出，Spring 的配置文件有两个，applicationContext.xml 和 action-servlet.xml。其实完全可以使用一个配置文件，通常习惯将 action bean 单独配置在表现层的 context 内。action-servlet.xml 用于配置表现层 context，其详细配置信息如下：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- spring 配置文件的根元素 -->
<beans>
    <!--每个 request 请求产生新实例，所以将 action 配置成 non-singleton-->
    <bean name="/login" class="lee.LoginAction" singleton="false">
        <!-- 配置依赖注入-->
        <property name="vb">
            <!-- 引用容器中另外的 bean-->
            <ref bean="vb"/>
        </property>
    </bean>
</beans>

```

由于每次请求时，都应该启动新的 action 来处理用户请求，因此应将 action bean 配置成 non-singleton 行为。

**注意：**ActionServlet 转发请求时，是根据 bean 的 name 属性，而不是 id 属性。因此，此处确定的 name 属性应与 Struts 的 action 属性相同。

applicationContext.xml 只有一个 bean 配置（配置了 vb bean）。其详细配置如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- spring 配置文件的根元素 -->
<beans>
    <!-- 配置 ValidBean 实例-->
    <bean id="vb" class="lee.ValidBeanImpl"/>
</beans>
```

ValidBeanImpl 是个业务逻辑 bean，在本示例程序中仅作简单的判断，ValidBeanImpl 的源代码如下：

```
//面向接口编程，实现 ValidBean 接口
public class ValidBeanImpl implements ValidBean
{
    //根据输入的用户名和密码判断是否有效
    public boolean valid(String username, String pass)
    {
        //有效，返回 true
        if (username.equals("scott") && pass.equals("tiger"))
        {
            return true;
        }
        return false;
    }
}
```

对原来的 Action 也作简单的修改，增加依赖注入所需的 setter 方法，使控制器依赖于业务逻辑组件。因为控制器本身没有逻辑能力，所以必须依赖于业务逻辑组件的计算结果决定返回值。

这样，程序的结果更加清晰化。修改后的 Action 源文件如下：

```
//业务控制器继承 Action
public class LoginAction extends Action
{
    //action 控制器将调用的业务逻辑组件
    private ValidBean vb;
    //依赖注入业务逻辑组件的 setter 方法。
    public void setVb(ValidBean vb)
    {
        this.vb = vb;
    }
    //必须重写该核心方法，该方法 actionForm 将表单的请求参数封装成值对象
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception
    {
        //form 由 ActionServlet 转发请求时创建，包装了所有的请求参数
        LoginForm loginForm = (LoginForm) form;
        //获取 username 请求参数
        String username = loginForm.getUsername();
        //获取 pass 请求参数
        String pass = loginForm.getPass();
        //下面作服务器端的数据校验
        String errMsg = "";
        //判断用户名不能为空
    }
}
```

```

        if (username == null || username.equals(""))
        {
            errMsg += "您的用户名丢失或没有输入, 请重新输入";
        }
        //判断密码不能为空
        else if(pass == null || pass.equals(""))
        {
            errMsg += "您的密码丢失或没有输入, 请重新输入";
        }
        //如果用户名和密码不为空, 才调用业务组件
        else
        {
            //vb 是业务逻辑组件, 由容器注入
            if (vb.valid(username, pass))
            {
                return mapping.findForward("welcome");
            }
            else
            {
                errMsg = "您的用户名和密码不匹配";
            }
        }
        //判断是否生成了错误信息,
        if (errMsg != null && !errMsg.equals(""))
        {
            //将错误信息保存在 request 里, 则跳转到 input 对应的 forward 对象
            request.setAttribute("err", errMsg);
            return mapping.findForward("input");
        }
        else
        {
            //如果没有错误信息, 跳转到 welcome 对应的 forward 对象
            return mapping.findForward("welcome");
        }
    }
}
}

```

由于增加了客户端数据校验, 因此对 ActionForm 也应作简单修改, 使 ActionForm 不再继承原有的 ActionForm, 而应继承 ValidatorActionForm。

为了完成数据校验, 还应该编写数据校验规则文件。在 struts-config.xml 文件的尾部, 有一个 plug-in 用来加载校验文件, 其中 validator-rules.xml 文件位于 struts 压缩包的 lib 下, 直接复制过来即可使用, 而 validator.xml 必须自己编写, 其 validator.xml 文件如下:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<!-- 验证规则文件的文件头, 包括 dtd 等信息-->
<!DOCTYPE form-validation PUBLIC
        "-//Apache Software Foundation//DTD Commons Validator Rules
Configuration 1.1.3//EN"
        "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<!-- 验证文件的根元素-->
<form-validation>
    <!-- 所有需要验证的 form 都放在 formset 里-->
    <formset>
        <!-- 需要验证的 form 名, 该名与 struts 里配置的名相同-->
        <form name="loginForm">
            <!-- 指定该 form 的 username 域必须满足的规则: 必填, 模式匹配-->
            <field property="username" depends="required,mask">

```

```

<arg key="loginForm.username" position="0"/>
<var>
    <!-- 确定匹配模式的正则表达式-->
    <var-name>mask</var-name>
    <var-value>^[a-zA-Z]+\$</var-value>
</var>
</field>
<!-- 指定该 form 的 pass 域必须满足的规则：必填-->
<field property="pass" depends="required">
    <msg name="required" key="pass.required"/>
    <arg key="loginForm.pass" position="0"/>
</field>
</form>
</formset>
</form-validation>

```

上面示例程序的结构非常清晰：表现层组件（action）配置在 action-servlet.xml 文件中；而业务逻辑层组件（vb）配置在 applicationContext.xml 文件中；如果应用中有 DAO 组件，可将该组件配置在 dao-context.xml 文件中，并将这三个文件放在 plug-in 元素里一起加载。

由于 DelegatingRequestProcessor 会将请求转发到 action，而该 action 已经处于 IoC 容器管理之下。因此，可以方便地访问容器中的其他 bean。

通过配置文件可看出，action 根本无需 type 属性，即 struts-config.xml 中 action 根本没有实例化，而是通过 DelegatingRequestProcessor 将请求转发给 Spring 容器中的同名 bean。这种转发时机非常早，避免创建 struts-config.xml 配置文件中的 action，因而性能非常好。采用这种整合策略的执行效果如图 7.5 所示。

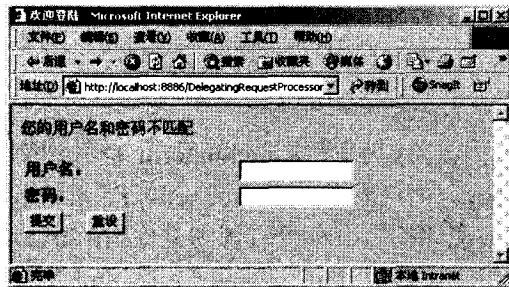


图 7.5 DelegatingRequestProcessor 整合策略登录失败的效果

### 7.3.2 使用 DelegatingActionProxy

使用 DelegatingRequestProcessor 非常简单方便，但有一个缺点：RequestProcessor 是 Struts 的一个扩展点，也许应用程序本身就需要扩展 RequestProcessor，而 DelegatingRequestProcessor 已经使用了这个扩展点。

为了重新利用 Struts 的 RequestProcessor 这个扩展点，有以下两个方法：

- 使应用程序的 RequestProcessor 不再继承 Struts 的 RequestProcessor，改为继承 DelegatingRequestProcessor。

- 使用 DelegatingActionProxy。

前者常常有一些未知的风险，而后者是 Spring 推荐的整合策略。使用 DelegatingActionProxy 与 DelegatingRequestProcessor 的目的只有一个，都是将请求转发给 Spring 管理的 bean。

DelegatingRequestProcessor 可直接替换了原有的 RequestProcessor，并在请求转发给 action 之前，转发给 Spring 管理的 bean；而 DelegatingActionProxy 则被配置成 Struts 的 action，即所有的请求先被 ActionServlet 拦截，然后将请求转发到对应的 action，而 action 的实现类全都是 DelegatingActionProxy；最后由 DelegatingActionProxy 将请求转发给 Spring 容器的 bean。

可以看出：使用 DelegatingActionProxy 比使用 DelegatingRequestProcessor 要晚一步转发到 Spring 的 context。但通过这种方式可以避免占用扩展点。

与使用 DelegatingRequestProcessor 对比，使用 DelegatingActionProxy 仅需要去掉 controller 配置元素，并将所有的 action 实现类改为 DelegatingActionProxy 即可。其详细的配置文件如下：

```
<!-- XML 文件版本，编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- struts 配置文件的文件头，包括 dtd 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- struts 配置文件的根元素-->
<struts-config>
    <!-- 配置 formbean，所有的 formbean 都放在 form-beans 元素里定义-->
    <form-beans>
        <!-- 定义了一个 formbean，确定 formbean 名和实现类-->
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 定义 action 部分，所有的 action 都放在 action-mapping 元素里定义-->
    <action-mappings>
        <!-- 这里只定义了一个 action。必须配置 action 的 type 元素为 Delegating
        ActionProxy -->
        <action path="/login" type="org.springframework.web.struts.
        DelegatingActionProxy"
            name="loginForm" scope="request" validate="true" input="/
            login.jsp" >
            <!-- 定义 action 内的两个局部 forward 元素-->
            <forward name="input" path="/login.jsp"/>
            <forward name="welcome" path="/welcome.html"/>
        </action>
    </action-mappings>
    <!-- 加载国际化的资源包-->
    <message-resources parameter="mess" />
    <!-- 装载验证的资源文件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.
        xml,/WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true" />
    </plug-in>
    <!-- 装载 Spring 配置文件，随应用启动创建 ApplicationContext 实例-->
    <plug-in className="org.springframework.web.struts. ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
```

```
        value="/WEB-INF/applicationContext.xml,  
        /WEB-INF/action-servlet.xml"/>  
</plug-in>  
</struts-config>
```

DelegatingActionProxy 接受 ActionServlet 转发过来的请求，然后转发给 ApplicationContext 管理的 bean，这是典型的链式处理。

通过配置文件可看出，在 struts-config.xml 文件中配置了大量 DelegatingActionProxy 实例，Spring 容器中也配置了同名的 Action。即 Struts 的业务控制器分成了两个部分：第一个部分是 Spring 的 DelegatingActionProxy，这个部分没有实际意义，仅仅完成转发；第二个部分是用户的 Action 实现类，负责实际的处理工作。

这种策略的性能比前一种的策略要差一些，因为需要多创建一个 DelegatingActionProxy 实例。而且在 J2EE 应用中的 Action 非常多，这将导致需要大量创建 DelegatingActionProxy 实例，在使用一次之后，要等待垃圾回收机制回收，从而降低了其性能。

DelegatingActionProxy 的执行效果如图 7.6 所示。

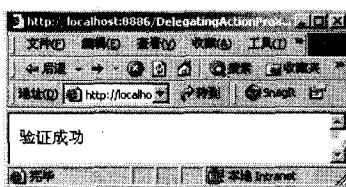


图 7.6 DelegatingActionProxy 整合策略登录成功的效果

## 7.4 使用 ActionSupport 代替 Action

另外，还有一种方法可以用于 Spring 与 Struts 的整合：让 Action 在程序中手动获得 ApplicationContext 实例。在这种整合策略下，Struts 的 Action 不接受 IoC 容器管理，使 Action 的代码与 Spring API 部分耦合，但造成代码污染。

这种策略也有其好处：代码的可读性非常强，在 Action 的代码中显式调用业务逻辑组件时，无须等待容器注入。

在 Action 中访问 ApplicationContext 有以下两种方法：

- 利用 WebApplicationContextUtils 工具类。
- 利用 ActionSupport 支持类。

WebApplicationContextUtils 可以通过 ServletContext 获得 Spring 容器实例。而 ActionSupport 类则提供一个更简单的方法：getWebApplicationContext()，该方法用于获取 ApplicationContext 实例。

Spring 扩展了 Struts 的标准 Action 类，可在其 Struts 的 Action 后加上 Support，Spring 的 Action 有如下几种：

- ActionSupport。
- DispatchActionSupport。

- `LookupDispatchActionSupport`。

- `MappingDispatchActionSupport`。

下面分别给出利用 `ActionSupport` 的示例代码：

新的业务控制器，继承 Spring 的 `ActionSupport` 类

```

public class LoginAction extends ActionSupport
{
    //依然将 ValidBean 作为成员变量
    private ValidBean vb;
    //构造器，注意：不可在构造器中调用 getWebApplicationContext() 方法
    public LoginAction()
    {
    }
    //完成 ValidBean 的初始化
    public ValidBean getVb()
    {
        return (ValidBean)getWebApplicationContext().getBean("vb");
    }
    //必须重写该核心方法，该方法 actionForm 将表单的请求参数封装成值对象
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //form 由 ActionServlet 转发请求时创建，包装了所有的请求参数
        LoginForm loginForm = (LoginForm)form;
        //获取 username 请求参数
        String username = loginForm.getUsername();
        //获取 pass 请求参数
        String pass = loginForm.getPass();
        //下面作服务器端的数据校验
        String errMsg = "";
        //判断用户名不能为空
        if (username == null || username.equals(""))
        {
            errMsg += "您的用户名丢失或没有输入，请重新输入";
        }
        //判断密码不能为空
        else if (pass == null || pass.equals(""))
        {
            errMsg += "您的密码丢失或没有输入，请重新输入";
        }
        //如果用户名和密码不为空，则调用业务组件
        else
        {
            //vb 是业务逻辑组件，通过上面的初始化方法获得
            if (getVb().valid(username, pass))
            {
                return mapping.findForward("welcome");
            }
            else
            {
                errMsg = "您的用户名和密码不匹配";
            }
        }
        //判断是否生成了错误信息
        if (errMsg != null && !errMsg.equals(""))
        {
    }
}

```

```
//将错误信息保存在 request 里，则跳转到 input 对应的 forward 对象
request.setAttribute("err", errMsg);
return mapping.findForward("input");
}
else
{
    //如果没有错误信息，跳转到 welcome 对应的 forward 对象
    return mapping.findForward("welcome");
}
}
}
```

在这种整合策略下，表现层的控制器组件不再接受 IoC 容器管理，因此没有了控制器 context。应将原有的 action-servlet.xml 文件删除，并修改 plug-in 元素，但不要加载该文件。另外，还要修改其 action 配置，将 action 配置的 type 元素修改成实际的处理类。这种整合策略也有个好处，提高了代码的可读性，对传统 Struts 应用开发的改变很小，容易使用。

将该 Action 部署在 struts-config.xml 中，此时 Struts 将负责创建该 Action。struts-config.xml 文件的源代码如下：

```
<!-- XML 文件版本，编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- struts 配置文件的文件头，包括 dtd 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- struts 配置文件的根元素-->
<struts-config>
    <!-- 配置 formbean，所有的 formbean 都放在 form-beans 元素里定义-->
    <form-beans>
        <!-- 定义了一个 formbean，确定 formbean 名和实现类-->
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 定义 action 部分，所有的 action 都放在 action-mapping 元素里定义-->
    <action-mappings>
        <!-- 这里只定义了一个 action。action 的类型为 ActionSupport 的子类 -->
        <action path="/login" type="lee.LoginAction"
            name="loginForm" scope="request" validate="true" input="/login.jsp" >
            <!-- 定义 action 内的两个局部 forward 元素-->
            <forward name="input" path="/login.jsp"/>
            <forward name="welcome" path="/welcome.html"/>
        </action>
    </action-mappings>
    <!-- 加载国际化的资源包-->
    <message-resources parameter="mess" />
    <!-- 装载验证的资源文件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.
            xml,/WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true" />
    </plug-in>
</struts-config>
```

此时，这种配置方式非常简单，Spring 无须使用配置 Action 的配置文件，只需要业务逻辑组件的配置文件，其业务逻辑组件的配置文件如下：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- spring 配置文件的根元素 -->
<beans>
  <!-- 配置 ValidBean 实例-->
  <bean id="vb" class="lee.ValidBeanImpl"/>
</beans>

```

该配置文件中的业务逻辑组件由 Spring 容器负责实现，而 ActionSupport 能够先定位 Spring 容器，然后获得容器的业务逻辑组件。这种整合策略与前面两种整合策略的执行效果完全相同。

从代码中我们可以看出，在这种整合策略下，业务控制器再次退回到 Struts 起初的设计，仅由 struts-config.xml 中 Action 充当，可以只创建实际的 Action 实例，而避免了如 DelegatingActionProxy 整合策略的性能低下，但这种整合策略的代价是污染了代码。

## 7.5 实用的整合策略

下面介绍一种实用的整合策略，采用这种整合策略的 Action 不仅可以避免直接与 Spring API 耦合，也可以避免在配置文件中大量地重复配置，还可以避免创建多余的 DelegatingActionProxy 实例。

这种整合策略其实相当简单，其思路是将获取业务逻辑组件的方式放在父类中实现，而其余的 Action 则从父类中获取。

下面是 BaseAction 的源代码：

```

//BaseAction 继承 ActionSupport 类
public class BaseAction extends ActionSupport
{
    /**
     * 根据 beanName 获取容器中的 bean 实例
     * @param beanName 容器中的 bean 名
     * @return 返回 Spring 容器中的 bean
     */
    public Object getBean(String beanName)
    {
        return getWebApplicationContext().getBean(beanName);
    }
}

```

该父类仅仅将 ActionSupport 的方法再次包装，但产生的优势非常明显，至少可以在实际的业务 Action 中避免调用 getWebApplicationContext 方法。

下面是业务 Action 的源代码：

```

//新的业务控制器，继承 BaseAction 类
public class LoginAction extends BaseAction
{
    //依然将 ValidBean 作为成员变量
    private ValidBean vb;

```

```
//构造器，注意：不可在构造器中调用 getWebApplicationContext()方法
public LoginAction()
{
}
//完成 ValidBean 的初始化
public ValidBean getVb()
{
    return (ValidBean)getBean("vb");
}
//必须重写该核心方法，该方法 actionForm 将表单的请求参数封装成值对象
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception
{
    //form 由 ActionServlet 转发请求时创建，包装了所有的请求参数
    LoginForm loginForm = (LoginForm)form;
    //获取 username 请求参数
    String username = loginForm.getUsername();
    //获取 pass 请求参数
    String pass = loginForm.getPassword();
    //下面作服务器端的数据校验
    String errMsg = "";
    //判断用户名不能为空
    if (username == null || username.equals(""))
    {
        errMsg += "您的用户名丢失或没有输入，请重新输入";
    }
    //判断密码不能为空
    else if(pass == null || pass.equals(""))
    {
        errMsg += "您的密码丢失或没有输入，请重新输入";
    }
    //如果用户名和密码不为空，则调用业务组件
    else
    {
        //vb 是业务逻辑组件，通过上面的初始化方法获得
        if (getVb().valid(username,pass))
        {
            return mapping.findForward("welcome");
        }
        else
        {
            errMsg = "您的用户名和密码不匹配";
        }
    }
    //判断是否生成了错误信息，
    if (errMsg != null && !errMsg.equals(""))
    {
        //将错误信息保存在 request 里，则跳转到 input 对应的 forward 对象
        request.setAttribute("err", errMsg);
        return mapping.findForward("input");
    }
    else
    {
        //如果没有错误信息，跳转到 welcome 对应的 forward 对象
        return mapping.findForward("welcome");
    }
}
```

从代码中可看出，在实际的业务控制器 Action 中，完全从 Spring 的 API 中分离出来，从而可以避免代码污染。

另外，还有一个最大的好处：实际的业务 Action 并没有与任何的整合策略耦合。假如需要在不同的整合策略之间切换，其业务 Action 完全不需要改变。

假设需要将整合策略切换到利用 Spring IoC 特性，则只需将 BaseAction 改成如下形式：

```
//BaseAction 不再继承任何类
public class BaseAction
{
    //业务逻辑组件
    Object serviceObj;
    //依赖注入必需的 setter 方法
    public void setServiceObj(Object obj)
    {
        this.serviceObj = obj
    }
    /**
     * 根据 beanName 获取容器中的 bean 实例
     * @param beanName 容器中的 bean 名
     * @return 返回 Spring 容器中的 bean
     */
    public Object getBean(String beanName)
    {
        return serviceObj;
    }
}
```

从代码中可以看出，BaseAction 等待容器依赖注入。当 BaseAction 的子类调用 getBean 方法时，传入的 beanName 完全没有作用。因为使用了 Spring 的依赖注入，Action 与业务逻辑组件之间的依赖关系可通过配置文件来设定。

下面的示例采用这种实用整合策略，而 Spring 管理 Action 则采用 DelegatingRequest Processor 策略。其 struts-config.xml 和 actionServlet 的配置文件如下：

```
<!-- XML 文件版本，编码集-->
<?xml version="1.0" encoding="gb2312"?>
<!-- struts 配置文件的文件头，包括 dtd 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<!-- struts 配置文件的根元素-->
<struts-config>
    <!-- 配置 formbean，所有的 formbean 都放在 form-beans 元素里定义-->
    <form-beans>
        <!-- 定义了一个 formbean，确定 formbean 名和实现类-->
        <form-bean name="loginForm" type="lee.LoginForm"/>
    </form-beans>
    <!-- 定义 action 部分，所有的 action 都放在 action-mapping 元素里定义-->
    <action-mappings>
        <!-- 这里只定义了一个 action。而且没有指定该 action 的 type 元素-->
        <action path="/login" name="loginForm"
            scope="request" validate="true" input="/login.jsp" >
            <!-- 定义 action 内的两个局部 forward 元素-->
            <forward name="input" path="/login.jsp"/>
            <forward name="welcome" path="/welcome.html"/>
        </action>
    </action-mappings>
</struts-config>
```

```

        </action>
    </action-mappings>
    <!-- 使用 DelegatingRequestProcessor 替换 RequestProcessor-->
    <controller processorClass="org.springframework.web.struts.Delegating
RequestProcessor"/>
    <!-- 加载国际化的资源包-->
    <message-resources parameter="mess" />
    <!-- 装载验证的资源文件-->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames" value="/WEB-INF/validator-rules.
xml,/WEB-INF/validation.xml" />
        <set-property property="stopOnFirstError" value="true" />
    </plug-in>
    <!-- 装载 Spring 配置文件，随应用启动创建 ApplicationContext 实例-->
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
            value="/WEB-INF/applicationContext.xml,
            /WEB-INF/action-servlet.xml" />
    </plug-in>
</struts-config>

```

action-servlet 的配置文件如下：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- 指定 Spring 配置文件的 dtd>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- spring 配置文件的根元素 -->
<beans>
    <!--每个 request 请求产生新实例，所以将 action 配置成 non-singleton-->
    <bean name="/login" class="lee.LoginAction" singleton="false">
        <!-- 配置依赖注入-->
        <property name="vb">
            <!-- 引用容器另外的 bean-->
            <ref bean="vb" />
        </property>
    </bean>
</beans>

```

从上面的策略中看出，采用这种整合策略，有如下优势：

- 可在不同整合策略中自由切换。
- 避免重复创建 DelegatingActionProxy 实例。
- 使业务 Action 避免代码污染。

## 本章小结

本章首先从原理上介绍了 Spring 与 MVC 框架整合，包括对 Spring 与 MVC 框架整合的通用配置，以及如何在 Web 应用启动时自动创建 ApplicationContext 实例。

其次重点介绍了 Spring 与 Struts 整合的几种策略：包括利用 DelegatingRequest Processor，DelegatingActionProxy，以及 ActionSupport 的整合策略，并详细分析了各种整合策略的优缺点。

最后介绍了一种实用的整合策略，并分析了其实用整合策略的优势。

# 第8章

## 企业应用开发的思考与策略

### 本章要点

- » 企业应用开发面临的挑战
- » 如何应对开发的挑战
- » 单态模式的使用
- » 代理模式的使用
- » Spring AOP 介绍
- » 贫血模型介绍
- » Rich Domain Object 模型介绍
- » 几种简化的架构模型

企业级应用的开发平台相当多，如 J2EE, .Net, Ruby On Rails 等。这些平台为企业级应用的开发提供了丰富的支持，都实现了企业应用底层的功能：缓冲池、多线程及持久层访问等。即使有如此多的选择，企业级应用的开发依然困难重重。

所有企业级应用的开发平台都提供了高级、抽象的 API，但仅依靠这些 API 构建企业级的应用远远不够。在这些高级 API 基础上，搭建一个良好的开发体系，也是企业级应用开发必不可少的步骤。本章将具体讨论如何搭建一个良好、可维护、可扩展、高稳定性且能够快速开发的应用架构。

## 8.1 企业应用开发面临的挑战

企业应用的开发是相当复杂的，这种复杂除了表现在技术方面外，还表现在行业本身。

企业级应用的开发往往需要面对更多的问题：大量的并发访问，复杂的环境，网络的不稳定，还有外部的 Crack 行为等。因此企业级应用必须提供更好的多线程支持，具备良好的适应性及良好的安全性等。

由于各行业的应用往往差别非常大，因此企业级应用往往具有很强的行业规则，尤其是优良的企业级应用往往更需要丰富的行业知识。企业应用的开发成功，也需要很多人的共同协作。

下面对企业应用开发面临的挑战作具体分析。

### 8.1.1 可扩展性、可伸缩性

市场是瞬息万变的，企业也是随之而变的。而信息化系统是为企业服务的，随着企业需求的变化，企业应用的变化也是必然的。

笔者在多年开发过程中，经常听到软件开发者对于需求变更的抱怨。当开发进行到中间时，大量的工作需要重新开始，确实给人极大的挫败感，难免软件开发者会抱怨。不过，笔者认为，一个积极的软件开发者应该可以正确对待需求的变更。需求的变更，表明有市场前景，只有有变化的产品才是有市场的产品。

优秀的企业级应用必须具备良好的可扩展性和可伸缩性。因为良好的可扩展性可允许系统动态增加新功能，而不会影响原有的功能。

良好的可扩展性建立在高度的解耦之上。使用 Delphi、PowerBuilder 等工具的软件开发人员对 ini 文件一定不会陌生。使用 ini 文件是一种基本的解耦方式，将运行所需资源、模块的耦合等从代码中分离出来，放入配置文件管理。这是一种优秀的设计思路，最理想的情况是允许使用可插拔式的模块（类似于 Eclipse 的插件方式）。

在 J2EE 应用里，大多采用 XML 文件作为配置文件。使用 XML 配置文件可以避免修改代码，从而能极大地提高程序的解耦。XML 文件常用于配置数据库连接信息，通过使用 XML 文件的配置方式，可以让应用在不同的数据库平台上轻松切换。从而避免在程序中使用硬编码的方式来定义数据库的连接，也避免了在更改数据库时，需要更改程序代码，从而提供更好的适应性。

下面是使用 Spring 的 bean 定义数据源的代码：

```
<!-- 使用 DBCP 数据源-->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <!-- 指定数据库驱动-->
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
```

```
</property>
<!-- 指定数据库服务的 URL--&gt;
&lt;property name="url"&gt;
    &lt;value&gt;jdbc:mysql://localhost:3306/j2ee&lt;/value&gt;
&lt;/property&gt;
<!-- 确定数据库的用户名--&gt;
&lt;property name="username"&gt;
    &lt;value&gt;root&lt;/value&gt;
&lt;/property&gt;
<!-- 确定数据库的密码--&gt;
&lt;property name="password"&gt;
    &lt;value&gt;32147&lt;/value&gt;
&lt;/property&gt;
<!-- 确定数据库的最大活动连接数--&gt;
&lt;property name="maxActive"&gt;
    &lt;value&gt;100&lt;/value&gt;
&lt;/property&gt;
<!-- 确定数据库的最大空闲连接数--&gt;
&lt;property name="maxIdle"&gt;
    &lt;value&gt;30&lt;/value&gt;
&lt;/property&gt;
<!-- 最大等待数--&gt;
&lt;property name="maxWait"&gt;
    &lt;value&gt;1000&lt;/value&gt;
&lt;/property&gt;
<!-- 默认是否自动提交--&gt;
&lt;property name="defaultAutoCommit"&gt;
    &lt;value&gt;true&lt;/value&gt;
&lt;/property&gt;
<!-- 是否允许自动回收连接--&gt;
&lt;property name="removeAbandoned"&gt;
    &lt;value&gt;true&lt;/value&gt;
&lt;/property&gt;
<!-- 连接超时的时长--&gt;
&lt;property name="removeAbandonedTimeout"&gt;
    &lt;value&gt;60&lt;/value&gt;
&lt;/property&gt;
&lt;/bean&gt;</pre>
```

上面的配置文件可用于建立数据库的连接，可等同于如下代码：

```
//创建数据源实例
BasicDataSource ds = new BasicDataSource();
//设置连接数据库的驱动
ds.setDriverClassName("com.mysql.jdbc.Driver");
//设置数据库服务的 URL
ds.setUrl("jdbc:mysql://localhost:3306/j2ee");
//设置数据库用户名
ds.setUsername("root");
//设置数据密码
ds.setPassword("32147");
//设置数据源的最大连接数
ds.setMaxActive(100);
//设置数据源的最大活动数
ds.setMaxIdle(30);
//设置数据源的最大等待数
ds.setMaxWait(1000);
//设置数据源是否自动提交
ds.setDefaultAutoCommit(true);
```

```
// 设置空闲连接是否自动回收  
ds.setRemoveAbandoned(true);  
// 设置回收连接超时的时长  
ds.removeAbandonedTimeout(60);
```

可以看出，第一种方式明显比第二种方式更优秀。因为当系统的数据库发生变化时（这是相当常见的情形），开发用的数据库与实际应用的数据不可能是同一个数据库，当软件系统由客户使用时，其数据库系统也是需要改变的。采用第一种方式则无须修改系统源代码，仅通过修改配置文件就可以让系统适应数据库的改变。

使用 XML 配置文件提高解耦的方式，是目前企业级应用最常用的解耦方式，而依赖注入的方式则提供了更高层次的解耦。使用依赖注入可以将各模块之间的调用从代码中分离出来，并通过配置文件来装配组件。此处的依赖注入并非特指 Spring，事实上，依赖注入容器很多，如 HiveMind 等。

## 8.1.2 快捷、可控的开发

如果没有时间限制，任何一个软件系统在理论上都是可实现的。但这样的条件不存在，软件系统必须要及时投放市场。对于企业级应用，时间的限制则更加严格。正如前文介绍的，企业的信息是瞬息万变的，与之对应的系统必须能与时俱进。因此快捷、可控是企业信息化系统必须面对的挑战。

软件开发人员常常乐于尝试各种新的技术，总希望将各种新的技术带入项目的开发中，因而难免有时会将整个项目陷入危险的境地。

当然，采用更优秀、更新颖的技术，通常可以保证软件系统的性能更加稳定。例如，从早期的 C/S 架构向 B/S 架构的过渡，以及从 Model 1 到 Model 2 的过渡等。这些都提高了软件系统的可扩展性及可伸缩性。

但采用新的技术所带来的风险也是不得不考虑的，开发架构必须重新论证，开发人员必须重新培训，这都需要成本投入。如果整个团队缺乏精通该技术的领导者，项目的开发难免会陷入技术难题，从而导致软件的开发过程变成不可控的——这是非常危险的事情。

成功的企业级应用，往往是保证其良好的可扩展性及可伸缩性，并建立在良好的可控性的基础上。

## 8.1.3 稳定性、高效性

企业级应用还有个显著特点：并发访问量大，访问频繁。因此稳定性、高效性是企业级信息化系统必须达到的要求。

企业级应用必须有优秀的性能，如采用缓冲池的技术。缓冲池专用于保存那些创建开销大的对象，如果对象的创建开销大，花费时间长，该技术可将这些对象缓存，避免了重复创建，从而提高系统性能。典型的应用是数据连接池。

提高企业级应用性能的另一个方法是——数据缓存。但数据缓存有其缺点：数据缓

存虽然在内存中，可极大地提高系统的访问速度；但缓存的数据占用了相当大的内存空间，这将会导致系统的性能下降。因此，数据缓存必须根据实际硬件设施制定，最好使用配置文件来动态管理缓存的大小。

### 8.1.4 花费最小化，利益最大化

这是个永恒的话题，任何一个商业组织都希望尽可能地降低开销。对开发者而言，降低开销主要是如何使在开发上的投资更有保值效果。即开发的软件系统具有很好的复用性，而不是每次面临系统开发任务，总是需要重复开发。

尽可能让软件可以有高层次的复用，这也是软件行业的发展趋势。早期软件多采用结构化的程序设计语言，此时的软件复用多停留在代码复用的层次。面向对象的程序设计语言的出现，使代码复用提高到了类的复用中。

在良好的 J2EE 架构设计中，复用是一个永恒的追求目标。架构设计师希望系统中大部分的组件可以复用，甚至能让系统的整个层可以复用。对于采用 DAO 模式的系统架构，如果数据库不发生大的改变，整个 DAO 层都不需要变化。

## 8.2 如何面对挑战

除了在上文介绍的所面临的各种技术挑战之外，企业级应用还有更多的挑战。每个行业都有各自复杂的规则，软件开发者往往缺乏对行业规则的了解。企业级应用的开发通常需要软件开发者和行业专家齐心协作，但系统开发中沟通成本又相当地高，因为软件开发者与行业专家之间的沟通往往存在不少障碍，这些都会影响系统的开发。

面对这些挑战，笔者有如下建议。

### 8.2.1 使用建模工具

此处的建模工具不一定是 ROSE 等，可以是简单的手画草图。当然，借助于专业的建模工具可以更好地确定系统模型。

任何语言的描述都是很空洞，而且具有很大的歧义性。使用图形则更加直观，而且意义更加明确。推荐使用建模工具主要出于如下两个方面的考虑。

- 用于软件开发者与行业专家之间沟通，正如前文所介绍的，行业专家与软件开发者之间对系统的理解可能存在少许差异。使用图形来帮助交流是不错的主意，通过建模工具绘制的各种图形，可使软件系统的模型更加清晰化。
- 用于软件开发者之间的沟通。即使在软件开发者内部，对于软件的模型往往也不是非常统一。使用建模工具可以减少软件开发者对于系统的理解分歧，从而降低沟通成本。

关于建模工具，推荐采用统一建模语言：UML。但 UML 的使用也需要掌握分寸，

在软件开发人员内部使用时，尽可能使用规范的 UML；但用于与行业专家沟通时，则应该尽量增加文字说明，而不要拘泥于 UML 图形的表现上，切忌仅将一个图形生硬摆出。

## 8.2.2 利用优秀的框架

使用框架可以大大提高系统的开发效率。除非开发一个非常小的系统，而且是开发后无须修改的系统，才可以完全抛弃框架。

优秀的框架本身就是从实际开发中抽取的通用部分，使用框架就可以避免重复开发通用部分。使用优秀的框架不仅可以直接使用框架中的基本组件和类库，还可以提高软件开发人员对系统架构设计的把握。使用框架有如下几个优势。

### 1. 提高生产效率

框架是在实际开发过程中提取出来的通用部分。使用框架可以避免开发重复的代码，看下面的 JDBC 数据库访问代码：

```
//注册数据库驱动
Class.forName("com.mysql.jdbc.Driver");
//数据服务的 URL
String url="jdbc:mysql://localhost/j2ee";
//数据库的用户名
String username="root";
//数据库密码
String password="32147";
//获取数据库连接
Connection conn= DriverManager.getConnection(url,username,password);
//创建 Statement
Statement stmt=conn.createStatement();
String sql="...";
//执行更新
stmt.executeUpdate(sql);
```

上面的代码是连接数据库执行数据更新的代码。而这个过程的大部分都是固定的，包括连接数据库、创建 Statement 及执行更新等，唯一需要变化的是 SQL 语句。

在实际的开发过程中，不可能总是采用这种步骤进行数据库访问，为避免代码重复，可在实际的开发中提取出如下方法：

```
public Class DbBean
{
    //用于执行更新的方法
    update(String sql)
    {
        //创建 Statement 对象
        Statement stmt= getConnection().createStatement();
        //更新所使用的 SQL 语句
        String sql="...";
        //执行更新
        stmt.executeUpdate(sql);
    }
    //获取数据库连接
    private Connection getConnection()
```

```

{
    if (conn == null)
    {
        //注册数据库驱动
        Class.forName("com.mysql.jdbc.Driver");
        //数据服务的URL
        String url="jdbc:mysql://localhost/j2ee";
        //数据库的用户名
        String username="root";
        //数据库密码
        String password="32147";
        //获取数据库连接
        conn= DriverManager.getConnection(url,username,password);
    }
}
...
}

```

上面的代码可以大大减少代码的重复量，但依然需要开发者完成连接数据库，创建 Statement 等步骤。如果使用 Spring 的 JDBC 抽象框架，上面的代码则可以简化为如下：

```

JdbcTemplate jt = new JdbcTemplate();
//为 JdbcTemplate 指定 DataSource
jt.setDataSource(ds);
//更新所使用的 SQL 语句
String sql="...";
//执行更新
jt.update(sql);

```

借助于 Spring 的 JDBC 抽象框架，数据库访问无须手动获取连接，无须创建 Statement 等对象。只需要传入一个 DataSource 对象，由 JdbcTemplate 完成 DataSource 获取数据库连接；创建 Statement 对象；执行数据库更新的通用步骤。而软件开发者只需要提供简单的 SQL 语句即可。

另外，使用框架可以缩短系统的开发时间，特别是对于大型项目的开发，使用框架的优势将更加明显。根据 javaworld 社区的调查，使用框架和不使用框架的时间对比如图 8.1 所示。

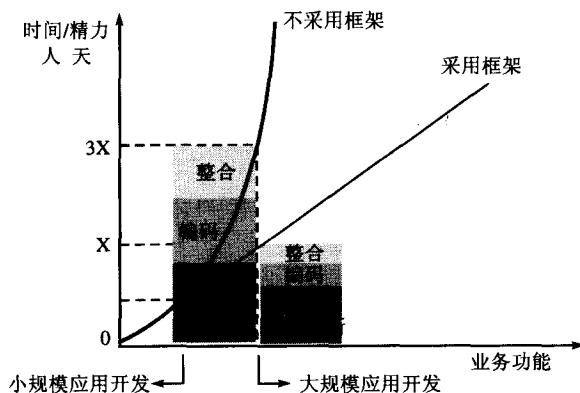


图 8.1 使用框架和不使用框架的对比

## 2. 具有更稳定、更优秀的性能

如果不使用已有的框架，系统开发者将面临着需要自己完成所有的底层部分。除非开发者丝毫不遵守软件复用的原则，总是重复书写相同代码。

系统开发者从系统开发中提取出的共同部分，也可成为框架。不可否认，完全由开发者自己提取框架有自己的优势，开发人员更加熟悉框架的运行，无须投入成本学习新的技术；但借助于已有框架的优势更加明显，已有的框架通常已被非常多的项目验证过，框架的性能等通常更有保障，而开发者自己提取的框架则可能包含许多未知的隐患。

因此为了更好地了解框架底层的运行，建议使用开源框架。

## 3. 更好的保值性

采用框架开发的系统使模块组织更加一致，从而降低了软件开发者之间的沟通成本，使系统具有更好的可读性，从而让软件系统具有更好的保值性。

后期的更新、维护也是企业级应用开发的重要组成部分。而使用框架的系统具有很大的相似性，从而有利于后期的更新及维护。

### 8.2.3 选择性地扩展

软件的需求千变万化，任何框架不可能总是那么完美，难免需要扩展现有的框架。

在许多项目中，开发者往往喜欢实现自己的框架，认为一个固定的框架会限制其发挥，事实上，他们没有意识到如何扩展框架。虽然开发自己的框架可以获得全部的控制权，但是这也意味着需要很多资源来实现它。正如前文讨论过的，实现自己的框架将需要开发者保证框架的稳定性及性能。

而对已有的框架进行扩展，则可最大限度地利用已有的框架，即使是扩展已有的框架，笔者也不建议盲目扩展，因为新增的部分有时会引入新的风险。笔者建议应对已有框架深入研究，尽量利用已有组件，除非无法使用已有框架时，才考虑选择性地扩展。

### 8.2.4 使用代码生成器

使用代码生成器可以自动生成部分程序，不但可以省去许多重复性的劳动，而且在系统开发过程中可以大大节省时间。程序生成器的效率很高，在开发软件的许多环节都有很好的作用，如数据持久化、界面及中间件等。

代码生成器还有个最大的作用：在原型开发期间可以大量重复利用代码生成器。原型系统通常在需求不十分明确时非常有用，此时的需求尚未确定，而软件功能业务无须十分完备，仅提供大致的软件功能，此时的代码生成器就非常有用。

## 8.3 常用的设计模式及应用

关于设计模式的定义可能已经有成千上万，在这里笔者不再重复那些烦琐的设计模式定义。事实上，设计模式既不是深奥的理论，也不是难以理解的概念，仅仅是一种习惯行为，只是这种习惯行为通常比较有效。

设计模式可以理解成一种思维定势，当固定的问题出现时，程序开发者可采用固定的解决方案（这种解决方案通常比较有效）。因此，学习设计模式，并不是学习如何实现设计模式，而是学习一种思维定势——对于固定的问题，通常采用何种方案解决。

### 8.3.1 单态模式的使用

单态模式的实现相当简单，是最常用的设计模式之一，以致于在相当多的地方都可以见到单态模式的痕迹。

下面代码实现了简单的单态模式：

```
public class Singleton
{
    //普通成员属性
    private int value ;
    private static Singleton st;
    //单态模式需要将构造器私有
    private Singleton()
    {
    }
    //静态方法，用于初始化该类的实例
    public static Singleton instance()
    {
        //首先判断该实例是否为空，如果为空
        if (st == null)
        {
            //创建新的实例
            st = new Singleton();
        }
        return st;
    }
    //返回该实例的成员属性的 getter 方法
    public int getValue()
    {
        return value;
    }
    //设置实例的成员属性的 setter 方法
    public void setValue()
    {
        value = 9;
    }
    //主方法，程序的入口
    public static void main(String[] args)
    {
        //创建该类的第一个实例
    }
}
```

```

        Singleton t1 = Singleton.instance();
        //创建该类的第二个实例
        Singleton t2 = Singleton.instance();
        //比较两个实例是否相等
        System.out.println(t1 == t2);
    }
}

```

正如前文所介绍的，学习设计模式绝不是学习这种代码实现。更多地是需要掌握在何时使用这种模式。

在任何不需要重复生成 Java 实例的场景中，都应该考虑使用单态模式。使用单态模式可以保证系统无须生成多个 Java 实例，从而减少内存占用率，也降低 JVM（Java 虚拟机）进行垃圾回收的开销。单态模式通常有如下两个使用场景：

- 工厂模式中的工厂。
- 使用服务定位器模式时的服务定位器。

关于工厂模式，可以参考 5.3.2 节。本节将具体介绍如何使用服务定位器。服务定位器模式在经典的 EJB 应用中非常常用，但这种模式决不局限在这种场景中使用，在任何需要大量定位相似服务的场景中，都可以使用服务定位器模式。

下面介绍 EJB 定位器模式的使用，关于 EJB 的开发，由于篇幅原因，本节不作详细介绍。

为了使用 EJB 定位器模式，可对 EJB 的 Home 接口进行扩展，让所有的 EJB 的 Home 接口都继承下面接口，而不是只继承系统默认的接口。下面是扩展的 Home 接口：

```

//扩展后的 Home 接口，该接口继承 EJBHome 接口
public interface MyEJBHome extends EJBHome
{
    public EJBObject create() throws RemoteException, CreateException;
}

```

这个接口比原来的接口增加了一个 create 方法，该方法用于返回 EJB 的 Remote 接口实例。后面开发的 EJB 其 Home 接口都从该接口派生出来。

下面是服务定位器的代码：

```

public class RemoteFactory
{
    //将该工厂写成单态模式
    private static RemoteFactory instance;
    //用 HashMap 来缓存获得的 EJBRemote
    private Map remoteInterfaces;
    private Context context;
    //单态模式，构造器私有
    private RemoteFactory() throws NamingException
    {
        //执行构造器时，完成缓存池的初始化
        remoteInterfaces = new HashMap();
        //创建 Context 实例
        context = getInitialContext();
    }
    //同步方法，用于获取工厂实例
    public static synchronized RemoteFactory getInstance()
        throws NamingException

```

```

{
    //不是线程安全的,但已经足够 || 加上 synchronized 会变成线程安全,但性能会下降
    if (instance == null)
    {
        instance = new RemoteFactory();
    }
    return instance;
}
//核心方法,通过该 lookup 方法可以返回 EJB 的 Remote 接口
private EJBObject lookup(String jndiName, Class homeInterfaceClass)
    throws NamingException, CreateException, RemoteException
{
    EJBObject remoteInterface = null;
    //从缓存池中取出 EJB 的 Remote 接口实例
    remoteInterface = (EJBObject)remoteInterfaces.get(homeInterfaceClass);
    //如果该接口实例不存在,则执行 JNDI 查找
    if (remoteInterface == null)
    {
        Object obj = context.lookup(jndiName);
        //将查找的对象强制类型转换为 MyEJBHome 类型,该类型是 EJBHome 的子接口
        MyEJBHome homeInterface =
            (MyEJBHome)PortableRemoteObject.narrow(obj, homeInterfaceClass);
        //调用 homeInterface 的 create 方法返回 Remote 接口
        remoteInterface = homeInterface.create();
        //将新创建的对象存入缓存
        remoteInterfaces.put(homeInterfaceClass, remoteInterface);
    }
    //返回 EJB 的 remote 接口实例
    return remoteInterface;
}

//工具方法,用于初始化 Context
private Context getInitialContext()
{
    if (context == null)
    {
        try
        {
            //如果不是在 J2EE 程序里,还需要属性文件才能正确创建 InitialConext()
            context = new InitialContext();
        }
        catch (NamingException ne)
        {
            ne.printStackTrace();
        }
    }
    return context;
}
}

```

从上面的代码可以看出,服务定位器实际上是工厂模式的一种应用。通过使用服务定位器,可以将服务的查找及服务的调用简化成简单的方法调用。而在这个服务定位器中,使用单态模式保证了服务定位器只有一个。

### 8.3.2 代理模式的使用

代理模式也是一种常用的设计模式。传统的代理模式主要用于采用简单对象来代替复杂的对象，如果创建一个对象所需的时间比较长，且计算资源相当昂贵，可以采用一个相对简单的对象来代替它。代理模式可将创建过程推迟到真正需要该对象时完成，一旦整个对象创建成功，对代理的方法调用将变成对实际对象的方法调用。

J2EE 里的代理模式通常是采用功能更强大的对象来代替目标对象。例如，对于普通的业务逻辑组件，其方法都应该有事务性，但这种开始事务和结束事务都是通用步骤。因此原始业务逻辑对象的方法可以无须事务操作，而是由系统生成动态代理来负责事务操作，并调用实际的目标方法。

看如下代理模式的实现，下面是原始的业务逻辑对象，该业务逻辑对象包含两个方法，这两个方法都没有事务操作。Java 的动态代理通常是代理接口，而且，是面向接口编程的。下面给出实现业务逻辑对象接口的源代码：

```
//业务接口
public interface Service
{
    //两个方法声明
    public void method1();
    public void method2();
}
```

下面是该接口的实现类，实现类并没有真正的业务逻辑，仅仅在控制台打印出简单的一行字符：

```
//业务逻辑组件，实现 Service 接口
public class ServiceImpl implements Service
{
    //实现接口实现的两个方法
    //第一个方法，在控制台打印出执行业务方法 1
    public void method1()
    {
        System.out.println("执行业务方法 1");
    }
    //第二个方法，在控制台打印出执行业务方法 2
    public void method2()
    {
        System.out.println("执行业务方法 2");
    }
}
```

业务逻辑组件通常由工厂负责产生，下面是业务逻辑组件的工厂，该工厂被写成单态模式：

```
//负责产生业务逻辑组件的工厂
public class ServiceFactory
{
    //将工厂设计成单态模式
    private static ServiceFactory sf;
```

```

//业务逻辑组件
private Service service;
//构造器私有，保证最多只能产生一个工厂
private ServiceFactory()
{
}
//实例化工厂的方法，用于创建工厂实例
public static ServiceFactory instance()
{
    //如果工厂实例为空，则重新创建工作实例
    if (sf == null)
    {
        sf = new ServiceFactory();
    }
    //返回工厂实例
    return sf;
}
//获取业务逻辑组件的方法
public Service getService(String impl)
{
    //工厂里负责产生业务逻辑组件
    if (impl.equals("one"))
    {
        if (service == null )
        {
            service = new ServiceImpl();
        }
        return service;
    }
    return null;
}
}

```

此时，依然只是一个简单的工厂模式。当获得业务逻辑组件时，其方法不会有任何的事务操作代码。为了让其方法具有事务性，可以增加事务操作所需的代理处理器。

下面是为增加事务操作而提供的代理处理器：

```

//代理处理器，实现 InvocationHandler
public class ProxyHandler implements InvocationHandler
{
    //target 是被代理的目标对象
    private Object target;
    //invoke 方法是实现 InvocationHandler 接口必须实现的方法，该方法会在目标对象方法
    //被调用时，自动被调用
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Exception
    {
        //result 是调用目标方法的返回值
        Object result = null;
        //如果调用的目标方法名为 method1
        if (method.getName().equals("method1"))
        {
            //开始事务
            System.out.println("=====开始事务=====");
            //执行目标方法
            result = method.invoke(target, args);
            //提交事务
            System.out.println("=====提交事务=====");
        }
    }
}

```

```
        }
        //否则
        else
        {
            //直接调用目标方法
            result =method.invoke(target, args);
        }
        //返回调用结果
        return result;
    }
    //该方法用于设置目标对象
    public void setTarget(Object o)
    {
        this.target = o;
    }
}
```

这是个通用的代理处理器，该处理器并不需要与任何特定的目标对象耦合。该处理器对 method1 方法增加了事务处理代码。实际上，此处并未有真正的事务操作代码，程序仅仅以控制台打印的两行语句来代表事务操作代码。

该代理处理器仅加强了 method1 方法，而且加强的方法以硬编码的方式写在代码中。这种方式显然存在一定的局限性：如果需要对其他方法增加事务处理，则需要修改其源代码，重新编译。

为了该代理处理器有更好的适应性，系统可以将需要增加事务操作的方法以 XML 文件配置，而该代理处理器负责解析 XML 文档，根据配置决定对哪个方法增加事务处理。

**注意：**在灵活的代理模式里，不仅仅目标方法可以提供 XML 文件配置，甚至代理处理器都可以通过依赖注入管理，这也是 Spring AOP 的处理方式。

下面是负责生成动态代理的代理工厂：

```
public class MyProxyFactory
{
    /**
     * 实例 Service 对象
     * @param serviceName String
     * @return Object
     */
    public static Object getProxy(Object object)
    {
        //代理处理类实例
        ProxyHandler handler = new ProxyHandler();
        //把该 service 实例托付给代理操作
        handler.setTarget(object);
        //第一个参数是用来创建动态代理的 ClassLoader 对象，只要该对象能访问 Service
        //接口即可
        //第二个参数表明该代理所实现的所有接口，第三个参数是代理的处理类
        return Proxy.newProxyInstance(object.getClass().getClassLoader(),
                                      object.getClass().getInterfaces(),handler);
    }
}
```

在上面的动态代理工厂里，可以动态生成 Java 实例，该实例将实现目标对象所实现

的全部接口。此处的目标对象是 ServiceImpl 实例，该实例实现 Service 接口。

下面是测试动态代理的程序：

```
public class TestService
{
    public static void main(String[] args)
    {
        //Service 对象
        Service service = null;
        //通过 ServiceFactory 生成 Service 实例，该 Service 实例是需要代理的目标对象
        Service targetObject = ServiceFactory.instance().getService("one");
        //根据目标对象，生成代理对象
        Object proxy = MyProxyFactory.getProxy(targetObject);
        //判断代理对象是否实现 Service 接口
        if (proxy instanceof Service)
        {
            //将代理对象转换 Service 实例
            service = (Service)proxy;
        }
        //通过代理对象执行 method1
        service.method1();
        //通过代理对象执行 method2
        service.method2();
    }
}
```

从中可以看出，代理处理器对 method1 方法增加了事务操作代码。由代理工厂生成的代理对象，并具有事务性。下面是执行测试代码的执行结果：

```
=====开始事务=====
执行业务方法 1
=====提交事务=====
执行业务方法 2
```

在执行业务方法 1 的前后，分别增加了开始事务和提交事务的操作代码。虽然这些代码本身没有事务性，但完全可以替换成事务操作代码，也可以替换成权限检查代码，如果调用目标方法的角色没有足够权限，则直接拒绝调用。

### 8.3.3 Spring AOP 介绍

Spring 的 AOP 是上面代理模式的深入。使用 Spring AOP 时，开发者无须实现业务逻辑对象工厂及代理工厂，这两个工厂都由 Spring 容器充当。Spring AOP 允许使用 XML 文件配置目标方法，但 ProxyHandler 允许使用依赖注入管理，为 Spring AOP 提供了更多灵活的选择。

在下面 Spring AOP 的示例中，InvocationHandler 采用动态配置，需要增加的方法也采用动态配置，一个目标对象可以有多个拦截器（类似于代理模式中的代理处理器）。

下面是原始的目标对象：

```
//目标对象的接口
public interface Person
```

```
{  
    //该接口声明了两个方法  
    void info();  
    void run();  
}
```

下面是原始目标对象的实现类：

```
//目标对象的实现类，实现类实现 Person 接口  
public class PersonImpl implements Person  
{  
    //两个成员属性  
    private String name;  
    private int age;  
    //name 属性的 setter 方法  
    public void setName(String name)  
    {  
        this.name = name;  
    }  
    //age 属性的 setter 方法  
    public void setAge(int age)  
    {  
        this.age = age;  
    }  
    //info 方法，该方法仅仅在控制台打印一行字符串  
    public void info()  
    {  
        System.out.println("我的名字是：" + name + "，今年年龄为：" + age);  
    }  
    //run 方法，该方法也在控制台打印一行字符串  
    public void run()  
    {  
        if (age < 45)  
        {  
            System.out.println("我还年轻，奔跑迅速...");  
        }  
        else  
        {  
            System.out.println("我年老体弱，只能慢跑...");  
        }  
    }  
}
```

该 Person 实例将由 Spring 容器负责产生和管理，其 name 属性和 age 属性也采用依赖注入管理。

为了充分展示 Spring AOP 的功能，此处为 Person 对象创建三个拦截器。第一个拦截器是调用方法前的拦截器，代码如下：

```
//调用目标方法前的拦截器，拦截器实现 MethodBeforeAdvice 接口  
public class MyBeforeAdvice implements MethodBeforeAdvice  
{  
    //实现 MethodBeforeAdvice 接口，必须实现 before 方法，该方法将在目标  
    //方法调用之前，自动被调用  
    public void before(Method m, Object[] args, Object target) throws Throwable  
    {  
        System.out.println("方法调用之前...");  
        System.out.println("下面是方法调用的信息: ");  
    }  
}
```

```

        System.out.println("所执行的方法是：" + m);
        System.out.println("调用方法的参数是：" + args);
        System.out.println("目标对象是：" + target);
    }
}

```

第二个拦截器是方法调用后的拦截器，该拦截器将在方法调用结束后自动被调用，代码如下：

```

//调用目标方法后的拦截器，该拦截器实现 AfterReturningAdvice 接口
public class MyAfterAdvice implements AfterReturningAdvice
{
    //实现 AfterReturningAdvice 接口必须实现 afterReturning 方法，该方法将在目标方法
    //调用结束后，自动被调用
    public void afterReturning(Object returnValue, Method m, Object[] args,
Object target) throws Throwable
    {
        System.out.println("方法调用结束...");
        System.out.println("目标方法的返回值是：" + returnValue);
        System.out.println("目标方法是：" + m);
        System.out.println("目标方法的参数是：" + args);
        System.out.println("目标对象是：" + target);
    }
}

```

第三个拦截器是 Around 拦截器，该拦截器既可以在目标方法之前调用，也可以在目标方法调用之后被调用，代码如下：

```

//Around 拦截器实现 MethodInterceptor 接口
public class MyAroundInterceptor implements MethodInterceptor
{
    //实现 MethodInterceptor 接口必须实现 invoke 方法
    public Object invoke(MethodInvocation invocation) throws Throwable
    {
        //调用目标方法之前执行的动作
        System.out.println("调用方法之前：invocation 对象：" + invocation);
        //调用目标方法
        Object rval = invocation.proceed();
        //调用目标方法之后执行的动作
        System.out.println("调用结束...");
        return rval;
    }
}

```

利用 Spring AOP 框架，实现之前的代理模式相当简单，只需要实现对应的拦截器即可，无须创建自己的代理工厂，采用 Spring 容器作为代理工厂即可。下面在 Spring 配置文件中配置目标 bean，以及拦截器。

#### Spring 配置文件的代码：

```

<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的文件头-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素-->
<beans>

```

```

<!-- 配置目标对象-->
<bean id="personTarget" class="lee.PersonImpl">
    <!-- 为目标对象注入 name 属性值-->
    <property name="name">
        <value>Wawa</value>
    </property>
    <!-- 为目标对象注入 age 属性值-->
    <property name="age">
        <value>51</value>
    </property>
</bean>
<!-- 第一个拦截器-->
<bean id="myAdvice" class="lee.MyBeforeAdvice"/>
<!-- 第二个拦截器-->
<bean id="myAroundInterceptor" class="lee.MyAroundInterceptor"/>
<!-- 将拦截器包装成 Advisor, 该对象还确定代理以怎样的方法增加处理-->
<bean id="runAdvisor" class="org.springframework.aop.support.
RegexpMethodPointcutAdvisor">
    <!-- advice 属性确定处理 bean-->
    <property name="advice">
        <!-- 此处的处理 bean 定义采用嵌套 bean, 也可引用容器的另一个 bean-->
        <bean class="lee.MyAfterAdvice"/>
    </property>
    <!-- patterns 确定正则表达式模式-->
    <property name="patterns">
        <list>
            <!-- 确定正则表达式列表-->
            <value>.*run.*</value>
        </list>
    </property>
</bean>
<!-- 使用 ProxyFactoryBean 产生代理对象-->
<bean id="person" class="org.springframework.aop.framework.
ProxyFactoryBean">
    <!-- 代理对象所实现的接口-->
    <property name="proxyInterfaces">
        <value>lee.Person</value>
    </property>
    <!-- 设置目标对象-->
    <property name="target">
        <ref local="personTarget"/>
    </property>
    <!-- 代理对象所使用的拦截器-->
    <property name="interceptorNames">
        <list>
            <value>runAdvisor</value>
            <value>myAdvice</value>
            <value>myAroundInterceptor</value>
        </list>
    </property>
</bean>
</beans>

```

该配置文件使用 `ProxyFactoryBean` 来生成代理对象，在配置 `ProxyFactoryBean` 工厂 bean 时，指定了 `target` 属性，该属性值为 `personTarget`，即指定代理的目标对象为 `personTarget`。通过 `interceptorNames` 属性确定代理所需要的拦截器，拦截器可以是普通的 `Advice`，对目标对象的所有方法起作用；拦截器也可以是 `Advisor`（`Advisor` 是 `Advice`

和切面的组合), 用于确定目标对象的哪些方法需要增加处理, 以及怎样的处理。在上面的配置文件中, 使用了三个拦截器, 其中 myAdvice、myAroundInterceptor 都是普通 Advice, 它们将对目标对象的所有方法起作用。而 runAdvisor 则使用了正则表达式切面, 匹配 run 方法, 即该拦截器只对目标对象的 run 方法起作用。

下面是测试代理的主程序:

```
public class BeanTest
{
    public static void main(String[] args) throws Exception
    {
        //创建 Spring 容器
        ApplicationContext ctx =
            new FileSystemXmlApplicationContext("bean.xml");
        //获取代理对象
        Person p = (Person)ctx.getBean("person");
        //执行 info 方法
        p.info();
        System.out.println("=====");
        //执行 run 方法
        p.run();
    }
}
```

下面是程序的执行结果:

```
方法调用之前...
下面是方法调用的信息:
所执行的方法是:public abstract void lee.Person.info()
调用方法的参数是: null
目标对象是: lee.PersonImpl@b23210
调用方法之前: invocation 对象: [invocation: method 'info', arguments []
]; target is of class [lee.PersonImpl]
我的名字是: Wawa , 今年年龄为: 51
调用结束...
=====
方法调用之前...
下面是方法调用的信息:
所执行的方法是:public abstract void lee.Person.run()
调用方法的参数是: null
目标对象是: lee.PersonImpl@b23210
调用方法之前: invocation 对象: [invocation: method 'run', arguments []
]; target is of class [lee.PersonImpl]
我年老体弱, 只能慢跑...
调用结束...
方法调用结束...
目标方法的返回值是 : null
目标方法是 : public abstract void lee.Person.run()
目标方法的参数是 : null
目标对象是 : lee.PersonImpl@b23210
```

程序的执行结果中用一行 “=” 用于区分两次调用的方法。在调用 info 方法时, 只有 myAdvice 和 myAroundInterceptor 两个拦截器起作用; 调用 run 方法时, 三个拦截器都起了作用。

通过上面的介绍可看出 Spring 的 AOP 框架是对代理模式的简化, 并拓展了代理模式。

Spring AOP 是 Spring 声明式事务的基础。了解 Spring AOP 对深入理解 Spring 的声明式事务管理是非常有好处的。另外，Spring AOP 还可以完成很多功能，如基于 AOP 的权限检查，本书的第 10 章示范了基于 Spring AOP 的权限检查范例，此处不在赘述。

## 8.4 常见的架构设计策略

目前流行的轻量级 J2EE 应用的架构比较一致，采用的技术也比较一致，通常使用 Spring 作为核心，向上整合 MVC 框架，向下整合 ORM 框架。使用 Spring 的 IoC 容器来管理各组件之间的依赖关系时，Spring 的声明事务将负责业务逻辑层对象方法的事务管理。

但在固定的技术组合上，依然可能存在小的变化。下面依次讨论可能存在的架构策略。

### 8.4.1 贫血模式

贫血模式是最常用的设计架构，也是最容易理解的架构。为了让读者通过本书顺利进入轻量级 J2EE 企业应用开发，本书的第 9 章及第 10 章的范例都将采用这种简单的架构模式。

所谓贫血，指 Domain Object 只是单纯的数据类，不包含业务逻辑方法，即每个 Domain Object 类只包含基本的 setter 和 getter 方法。所有的业务逻辑都由业务逻辑组件实现，这种 Domain Object 就是所谓的贫血的 Domain Object，采用这种 Domain Object 的架构即所谓的贫血模式。

下面以第 9 章的消息发布系统的部分代码为例，介绍贫血模式。

在贫血模式里，所有的 Domain Object 只是单纯的数据类，只包含每个属性的 setter 和 getter 方法，如下是两个持久化类。

第一个 Domain Object 是消息，其代码如下：

```
public class News extends BaseObject implements Serializable
{
    //主键
    private Long id;
    //消息标题
    private String title;
    //消息内容
    private String content;
    //消息的发布时间
    private Date postDate;
    //消息的最后修改时间
    private Date lastModifyDate;
    //消息所属分类
    private Category category;
    //消息对应的消息回复
    private Set newsReviews;
    //无参数的构造器
}
```

```
public News() {  
}  
//消息回复对应的 getter 方法  
public Set getNewsReviews() {  
    return newsReviews;  
}  
//消息回复对应的 setter 方法  
public void setNewsReviews(Set newsReviews) {  
    this.newsReviews = newsReviews;  
}  
//消息分类对应的 getter 方法  
public Category getCategory() {  
    return category;  
}  
//消息分类对应的 setter 方法  
public void setCategory(Category category) {  
    this.category = category;  
}  
//消息最后修改时间的 getter 方法  
public Date getLastModifyDate() {  
    return lastModifyDate;  
}  
//消息最后修改时间的 setter 方法  
public void setLastModifyDate(Date lastModifyDate) {  
    this.lastModifyDate = lastModifyDate;  
}  
//消息发布时间的 getter 方法  
public Date getPostDate() {  
    return postDate;  
}  
//消息发布时间的 setter 方法  
public void setPostDate(Date postDate) {  
    this.postDate = postDate;  
}  
//消息内容对应的 getter 方法  
public String getContent() {  
    return content;  
}  
//消息发布者对应的 setter 方法  
public void setContent(String content) {  
    this.content = content;  
}  
//消息主键对应的 getter 方法  
public Long getId() {  
    return id;  
}  
//消息主键对应的 setter 方法  
public void setId(Long id) {  
    this.id = id;  
}  
//消息标题对应的 getter 方法  
public String getTitle() {  
    return title;  
}  
//消息标题对应的 setter 方法  
public void setTitle(String title) {  
    this.title = title;  
}  
//Domain Object 重写 equals 方法
```

```
public boolean equals(Object object) {
    if (!(object instanceof News)) {
        return false;
    }
    News rhs = (News) object;
    return this.poster.equals(rhs.getPoster())
        && this.postDate.equals(rhs.getPostDate());
}
//Domain Object 重写的 hashCode 方法
public int hashCode() {
    return this.poster.hashCode() + this.postDate.hashCode();
}
//Domain Object 重写 toString 方法
public String toString() {
    return new ToStringBuilder(this).append("id", this.id).append("title",
        this.title).append("postDate", this.postDate).append("content",
        this.content).append("lastModifyDate", this.lastModifyDate)
        .append("poster", this.poster)
        .append("category", this.category).append("newsReviews",
        this.newsReviews).toString();
}
}
```

第二个 Domain Object 是消息对应的回复，其代码如下：

```
public class NewsReview extends BaseObject
{
    //消息回复的主键
    private Long id;
    //消息回复的内容
    private String content;
    //消息回复的回复时间
    private Date postDate;
    //回复的最后修改时间
    private Date lastModifyDate;
    //回复的对应的消息
    private News news;
    //消息回复的构造器
    public NewsReview() {
    }
    //回复内容对应的 getter 方法
    public String getContent() {
        return content;
    }
    //回复内容对应的 setter 方法
    public void setContent(String content) {
        this.content = content;
    }
    //回复主键对应的 setter 方法
    public Long getId() {
        return id;
    }
    //回复主键对应的 setter 方法
    public void setId(Long id) {
        this.id = id;
    }
    //回复的最后修改时间对应的 getter 方法
    public Date getLastModifyDate() {
        return lastModifyDate;
    }
}
```

```

    }
    //回复的最后修改时间对应的 setter 方法
    public void setLastModifyDate(Date lastModifyDate) {
        this.lastModifyDate = lastModifyDate;
    }
    //回复对应的消息的 getter 方法
    public News getNews() {
        return news;
    }
    //回复对应的消息的 setter 方法
    public void setNews(News news) {
        this.news = news;
    }
    //回复发布时间的 getter 方法
    public Date getPostDate() {
        return postDate;
    }
    //回复发布时间的 setter 方法
    public void setPostDate(Date postDate) {
        this.postDate = postDate;
    }
    //Domain Object 重写的 equals 方法
    public boolean equals(Object object) {
        if (!(object instanceof NewsReview)) {
            return false;
        }
        NewsReview rhs = (NewsReview) object;
        return this.poster.equals(rhs.getPoster()) &&
            this.postDate.equals(rhs.getPostDate());
        /*return new EqualsBuilder().append(this.news, rhs.news).append(
            this.content, rhs.content).append(this.postDate, rhs.postDate)
            .append(this.lastModifyDate, rhs.lastModifyDate).append(
                this.id, rhs.id).append(this.poster, rhs.poster)
            .isEquals();*/
    }
    //Domain Object 对应的 hashCode 方法
    public int hashCode() {
        return this.poster.hashCode() + this.postDate.hashCode();
        /*return new HashCodeBuilder(-1152635115, 884310249).append(this.news)
            .append(this.content).append(this.postDate).append(
                this.lastModifyDate).append(this.id)
            .append(this.poster).toHashCode();*/
    }
    //Domain Object 对应的 toString 方法
    public String toString() {
        return new ToStringBuilder(this).append("id", this.id).append(
            "postDate", this.postDate).append("lastModifyDate",
            this.lastModifyDate).append("content", this.content).append(
                "poster", this.poster).append("news", this.news).toString();
    }
}

```

从上面贫血模式的 Domain Object 可看出，其类代码中只有 setter 和 getter 方法，这种 Domain Object 只是单纯的数据体，类似于 C 语言的数据结构。虽然它的名字是 Domain Object，却没有包含任何业务对象的相关方法。Martin Fowler 认为，这是一种不健康的建模方式，Domain Model 既然代表了业务对象，就应该包含相关的业务方法。从语言的角度上来说，Domain Model 在这里被映射为 Java 对象（一般都是 ORM），Java 对象应

该是数据与动作的集合，贫血模型相当于抛弃了 Java 面向对象的性质。

Rod Johnson 和 Martin Fowler 一致认为：贫血的 Domain Object 实际上以数据结构代替了对象。他们认为 Domain Object 应该是个完整的 Java 对象，既包含基本的数据，也包含了操作数据相应的业务逻辑方法。

下面是 NewsDAOHibernate 的源代码，该 DAO 对象用于操作 News 对象：

```
//NewsDAO继承 HibernateDaoSupport, 实现 NewsDAO 接口
public class NewsDAOHibernate extends HibernateDaoSupport implements NewsDAO
{
    //根据主键加载消息
    public News getNews(Long id)
    {
        News news = (News) getHibernateTemplate().get(News.class, id);
        if (news == null) {
            throw new ObjectRetrievalFailureException(News.class, id);
        }
        return news;
    }
    //保存新的消息
    public void saveNews(News news) {
        getHibernateTemplate().saveOrUpdate(news);
    }
    //根据主键删除消息
    public void removeNews(Long id)
    {
        getHibernateTemplate().delete(getNews(id));
    }
    //查找全部的消息
    public List findAll()
    {
        getHibernateTemplate().find("from News");
    }
}
```

既然 DAO 对象完成具体的持久化操作，因此基本的 CRUD 操作都应该在 DAO 对象中实现。但 DAO 对象应该包含多少个查询方法，并不是确定的。因此，根据业务逻辑的不同需要，不同的 DAO 对象可能有数量不等的查询方法。

对于现实中 News，应该包含一个业务方法（addNewsReviews 方法）。在贫血模式下，News 类的代码并没有包含该业务方法，只是将该业务方法放到业务逻辑对象中实现，下面是业务逻辑对象实现 addNewsReviews 的代码：

```
public class FacadeManagerImpl implements FacadeManager
{
    //业务逻辑对象依赖的 DAO 对象
    private CategoryDAO categoryDAO;
    private NewsDAO newsDAO;
    private NewsReviewDAO newsReviewDAO;
    private UserDAO userDAO;
    //...此处还应该增加依赖注入 DAO 对象必需的 setter 方法
    //...此处还应该增加其他业务逻辑方法
    //下面是增加新闻回复的业务方法
    public NewsReview addNewsReview(Long newsId, String content)
    {
        //根据新闻 id 加载新闻
```

```

    News news = newsDao.getNews(newsId);
    //以默认构造器创建新闻回复
    NewsReview review = new NewsReview();
    //设置新闻与新闻回复之间的关联
    review.setNews(news);
    //设置新闻回复的内容
    review.setContent(content);
    //设置回复的回复时间
    review.setPostDate(new Date());
    //设置新闻回复的最后修改时间
    review.setLastModifyDate(new Date());
    //保存回复
    newsReviewDAO.saveNewsReview(review);
    return review;
}
}

```

在贫血模式下，业务逻辑对象正面封装了全部的业务逻辑方法，Web 层仅与业务逻辑组件交互即可，无须访问底层的 DAO 对象。Spring 的声明式事务管理将负责业务逻辑对象方法的事务性。

在贫血模式下，其分层非常清晰。Domain Object 并不具备领域对象的业务逻辑功能，仅仅是 ORM 框架持久化所需的 POJO，仅是数据载体。贫血模型容易理解，开发便捷，但严重背离了面向对象的设计思想，所有的 Domain Object 并不是完整的 Java 对象。总结起来，贫血模式存在如下缺点：

- 项目需要书写大量的贫血类，当然也可以借助某些工具自动生成。
- Domain Object 的业务逻辑得不到体现。由于业务逻辑对象的复杂度大大增加，许多不应该由业务逻辑对象实现的业务逻辑方法，完全由业务逻辑对象实现，从而使业务逻辑对象的实现类变得相当臃肿。

贫血模式的优点是：开发简单、分层清晰、架构明晰且不易混淆；所有的依赖都是单向依赖，解耦优秀。适合于初学者及对架构把握不十分清晰的开发团队。

## 8.4.2 Rich Domain Object 模式

在这种模式下，Domain Object 不再是单纯的数据载体，Domain Object 包含了相关的业务逻辑方法。例如，News 类包含了 addNewsView 方法等。

下面是修改后的 News 类的源代码：

```

public class News extends BaseObject
{
    //此处省略了其他的属性
    //此处省略了属性对应的 setter 和 getter 方法
    //增加新闻回复的业务逻辑方法
    public NewsReview addNewsReview(String content)
    {
        //以默认构造器创建新闻回复实例
        NewsReview review = new NewsReview();
        //设置回复内容
        review.setContent(content);
        //设置回复的发布日期
    }
}

```

```

        review.setPostDate(new Date());
        //设置回复的最后修改日期
        review.setLastModifyDate(new Date());
        //设置回复与消息的关联
        review.setNews(this);
        return review;
    }
    //此处省略了重写的 hashCode, equals 等方法
}

```

在上面的 Domain Object 中，包含了相应的业务逻辑方法，这是一种更完备的建模方法。

**注意：**不要在 Domain Object 中对消息回复完成持久化，如需完成持久化，必须调用 DAO 组件；一旦调用 DAO 组件，将造成 DAO 对象和 Domain Object 的双向依赖；另外，Domain Object 中的业务逻辑方法还需要在业务逻辑组件中代理，才能真正实现持久化。

在上面的业务逻辑方法中，并没有进行持久化。如果抛开 DAO 层，这种 Domain Object 也可以独立测试，只是没有进行持久化。

DAO 对象是变化最小的对象，它们都是进行基本的 CRUD 操作，在两种模型下的 DAO 对象没有变化。

另外还需要对业务逻辑对象进行改写，虽然 Domain Object 包含了基本业务逻辑方法，但业务逻辑对象还需代理这些方法，修改后业务逻辑对象的代码如下：

```

public class FacadeManagerImpl implements FacadeManager
{
    //业务逻辑对象依赖的 DAO 对象
    private CategoryDAO categoryDAO;
    private NewsDAO newsDAO;
    private NewsReviewDAO newsReviewDAO;
    private UserDao userDao;
    //...此处还应该增加依赖注入 DAO 对象必需的 setter 方法
    //...此处还应该增加其他业务逻辑方法
    //下面是增加新闻回复的业务方法
    public NewsReview addNewsReview(Long newsId, String content)
    {
        //根据新闻 id 加载新闻
        News news = newsDao.getNews(newsId);
        //通过 News 的业务方法添加回复
        NewsReview review = news.addNewsReview(content);
        //此处必须显示持久化消息回复
        newsReviewDAO.saveNewsReview(review);
        return review;
    }
}

```

在 Rich Domain Object 的模型中，addNewsReview 方法将放在 News 类中实现，而业务逻辑对象仅对该方法进行简单的代理，执行必要的持久化操作。

在这里存在一个问题：业务逻辑方法很多，哪些业务逻辑方法应该放在 Domain Object 对象中实现，而哪些业务逻辑方法完全由业务逻辑对象实现呢？Rod Johnson 认为，可重用度高，与 Domain Object 密切相关的业务方法应放在 Domain Object 对象中实现。

业务逻辑方法是否需要由 Domain Object 实现的标准，从一定程序上说明了采用 Rich

Domain Object 模型的原因。由于某些业务方法只是专一地属于某个 Domain Object，因此将这些方法由 Domain Object 实现，能提供更好的软件复用，能更好地体现面向对象的封装性。

Rich Domain Object 模型的各组件之间关系大致如图 8.2 所示（贫血模式的组件关系图与此类似）。

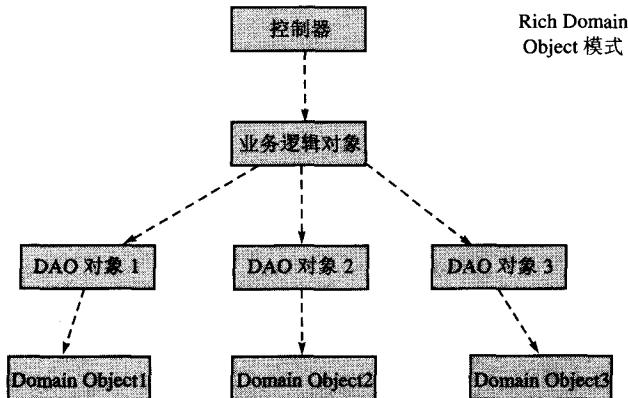


图 8.2 Rich Domain Object 的组件关系图

这种 Rich Domain Object 模型主要的问题是业务逻辑对象比较复杂，由于业务逻辑对象需要正面封装所有的 DAO 对象，因而难免有大量的 DAO 方法（基本的 CRUD）需要业务逻辑对象封装。业务逻辑对象封装 DAO 方法主要基于如下考虑：

- DAO 对象不应该暴露为 Web 层。
  - DAO 对象的 DAO 方法必须增加事务控制代码，而事务控制则放在业务逻辑层完成。
- 为了简化业务逻辑对象的开发，Rich Domain Object 模型可以有如下两个方向的改变：
- 合并业务逻辑对象与 DAO 对象。
  - 合并业务逻辑对象和 Domain Object。

### 1. 合并业务逻辑对象与 DAO 对象

在这种模型下 DAO 对象不仅包含了各种 CRUD 方法，而且还包含各种业务逻辑方法。此时的 DAO 对象，已经完成了业务逻辑对象所有任务，变成了 DAO 对象和业务逻辑对象混合体。此时，业务逻辑对象依赖 Domain Object，既提供基本的 CRUD 方法，也提供相应的业务逻辑方法。

下面是这种模式的代码（Domain Object 的实现与前面的 Rich Domain Object 模式一样，此处不再给出）：

```

// NewsServiceHibernate 继承 HibernateDaoSupport，实现 NewsService 接口
public class NewsServiceHibernate extends HibernateDaoSupport
    implements NewsService
{
    //此处添加 NewsService 对象依赖的 DAO 对象，以及对应的 setter 方法
    //根据主键加载消息
    public News getNews(Long id)
    {
        ...
    }
}

```

```

News news = (News) getHibernateTemplate().get(News.class, id);
if (news == null) {
    throw new ObjectRetrievalFailureException(News.class, id);
}
return news;
}
//保存新的消息
public void saveNews(News news) {
    getHibernateTemplate().saveOrUpdate(news);
}
//根据主键删除消息
public void removeNews(Long id)
{
    getHibernateTemplate().delete(getNews(id));
}
//查找全部的消息
public List findAll()
{
    getHibernateTemplate().find("from News");
}
//下面是增加新闻回复的业务方法
public NewsReview addNewsReview(Long newsId , String content)
{
    //根据新闻 id 加载新闻
    News news = newsDao.getNews(newsId);
    //通过 News 的业务方法添加回复
    NewsReview review = news.addNewsReview(content);
    //此处必须显示持久化消息回复
    newsReviewService.saveNewsReview(review);
    return review;
}
}
}

```

正如上面见到的，DAO 对象和业务逻辑对象之间容易形成交叉依赖（可能某个业务逻辑方法的实现，必须依赖于原来的 DAO 对象）。当 DAO 对象被取消后，业务逻辑对象取代了 DAO 对象，因此变成了一个业务逻辑对象依赖多个业务逻辑对象。而每个业务逻辑对象都可能包含需要多个 DAO 对象协作的业务方法，从而导致业务逻辑对象之间的交叉依赖。

业务逻辑对象和 DAO 对象合并后的组件关系如图 8.3 所示。

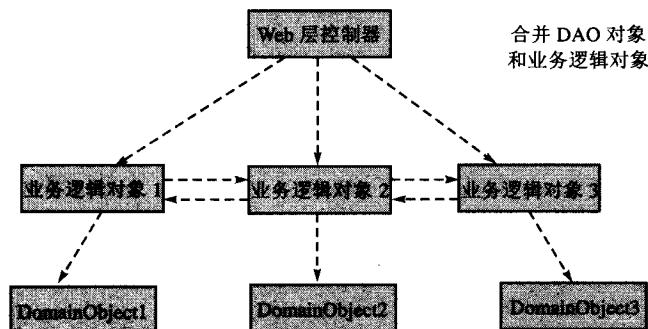


图 8.3 合并 DAO 对象和业务逻辑对象

这种模型也导致了 DAO 方法和业务逻辑方法混合在一起，显得职责不够单一，软

件分层结构不够清晰。此外，使业务逻辑对象之间交叉依赖，容易产生混乱，未能做到彻底的简化。

## 2. 合并业务逻辑对象和 Domain Object

在这种模型下，所有的业务逻辑都应该被放在 Domain Object 里面，而此时的业务逻辑层不再是传统的业务逻辑层，它仅仅封装了事务和少量逻辑，完全无需业务逻辑对象的支持。而 Domain Object 依赖于 DAO 对象执行持久化操作，此处 Domain Object 和 DAO 对象形成双向依赖，这种设计在某些地方也被称为充血模式，但有时会带来相当大的危险。

在这种设计模式下，几乎不再需要业务逻辑层，而 Domain Object 则依赖 DAO 对象完成持久化操作，下面是在这种模式下的 News 类代码：

```
public class News extends BaseObject
{
    //此处省略了其他的属性
    //此处省略了属性对应的 setter 和 getter 方法
    //增加新闻回复的业务逻辑方法
    public NewsReview addNewsReview(String content)
    {
        //以默认构造器创建新闻回复实例
        NewsReview review = new NewsReview();
        //设置回复内容
        review.setContent(content);
        //设置回复的发布日期
        review.setPostDate(new Date());
        //设置回复的最后修改日期
        review.setLastModifyDate(new Date());
        //设置回复与消息的关联
        review.setNews(this);
        //直接调用 newsReviewsDao 完成消息回复的持久化
        newsReviewsDao.save(review);
        return review;
    }
    //此处省略了重写的 hashCode, equals 等方法
}
```

从上面代码中可以看到，由于 Domain Object 必须使用 DAO 对象完成持久化，因此 Domain Object 必须接收 IoC 容器的注入，而 Domain Object 获取容器注入的 DAO 对象，通过 DAO 对象完成持久化操作。

合并业务逻辑对象和 Domain Object 后各组件的关系如图 8.4 所示。

这种模型的优点是：业务逻辑对象非常简单，只提供简单的事务操作，业务逻辑对象无须依赖于 DAO 对象。

但这种模型的缺点也是非常明显的：

- DAO 对象和 Domain Object 形成了双向依赖，其复杂的双向依赖会导致很多潜在的问题。
- 业务逻辑层和 Domain 层的逻辑混淆不清，在实际项目中，极容易导致架构混乱。
- 由于使用业务逻辑对象提供事务封装特性，业务逻辑层必须对所有的 Domain Object 的逻辑提供相应的事务封装，因此业务逻辑对象必须重新定义 Domain

Object 实现的业务逻辑，其工作相当烦琐。

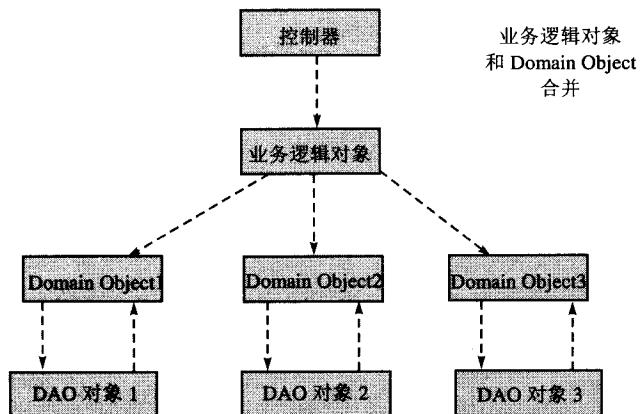


图 8.4 合并业务逻辑组件和 Domain Object

### 8.4.3 抛弃业务逻辑层

在 Rich Domain Object 模型的各种变化中，虽然努力简化业务逻辑对象，但业务逻辑对象依然存在，依然使用业务逻辑对象正面封装所有的业务请求。下面介绍更彻底的简化，即彻底放弃业务逻辑层。

抛弃业务逻辑层也有两种形式：

- Domain Object 彻底取代业务逻辑对象。
- 由控制器直接调用 DAO 对象。

#### 1. Domain Object 完全取代业务逻辑对象

这种设计模式是充血模式更加激进的演化。由于在充血模式中业务逻辑对象的作用仅仅只提供事务封装，业务逻辑对象存在的必要性不是很大，因此考虑对 Domain Object 的业务逻辑方法增加事务管理，而 Web 层的控制器则直接依赖于 Domain Object。

这种模型更加简化，使 Domain Object 与 DAO 对象形成双向依赖，而 Web 层的控制器直接调用 Domain Object 的业务逻辑方法。这种模型在有些地方也被称为胀血模式。

这种模型的优点是：分层少，代码实现简单。

但这种模型的缺点也很明显：

- 业务逻辑对象的所有业务逻辑都将在 Domain Object 中实现，势必引起 Domain Object 的混乱。
- Domain Object 必须向 Web 层直接暴露，可能导致意想不到的问题。

这种模型与充血模式的缺点相同：Domain Object 必须配置在 Spring 容器中，接受 Spring 容器的依赖注入。

在这种架构模型下，Domain Object 相当不稳定。如果业务逻辑需要改变，Domain Object 也需要发生改变，而 DAO 对象与 Domain Object 形成双向依赖，这将导致从底层的 Domain Object 和 DAO 对象的修改，使这种架构模式的分层完全失去意义。各层之间以强

耦合方式组合在一起，各层对象互相依赖，牵一发而动全身，几乎是最差的一种策略。

## 2. 控制器完成业务逻辑

在这种模型里，控制器直接调用 DAO 对象的 CRUD 方法，通过调用基本的 CRUD 方法，完成对应的业务逻辑方法。这种模型下，业务逻辑对象的功能由控制器完成。事务则推迟到控制器中完成，因此对控制器的 execute 方法增加事务控制即可。

对于基本的 CRUD 操作，控制器可直接调用 DAO 对象的方法，省略了业务逻辑对象的封装，这就是这种模型的最大优势。对于业务逻辑简单（当业务逻辑只是大量的 CRUD 操作时）的项目，使用这种模型也未尝不是一种好的选择。

但这种模型将导致控制变得臃肿，因为每个控制器除了包含原有的 execute 方法之外，还必须包含所需要的业务逻辑方法的实现。极大地省略了业务逻辑层的开发，避免了业务逻辑对象不得不大量封装基本的 CRUD 方法的弊端。

这种模型也有其缺点：

- 因为没有业务逻辑层，对于那些需要多个 DAO 参与的复杂业务逻辑，在控制器中必须重复实现，其效率低，也不利于软件重用。
- Web 层的功能不再清晰，人为复杂化。Web 层不仅负责实现控制器逻辑，还需要完成业务逻辑的实现，因此必须精确控制何时调用 DAO 方法控制持久化。
- 扩大了事务的影响范围。大部分情况下，只有业务逻辑方法需要增加事务控制，而 execute 方法无须增加事务控制。但如果 execute 方法直接调用了 DAO 对象的 CRUD 方法，则会导致这些方法不在事务环境下执行。为了让数据库访问都在事务环境下进行，因此不得不将事务范围扩大到整个 execute 方法。这是一种低性能的做法。

## 本章小结

本章首先介绍了笔者在架构设计方面的一些经验，从企业应用开发面临的困难讲起，讲解了面对这些困难时应该采用何种应对策略。

其次介绍了常用的代理模式的使用，并深入介绍了由此衍生出来的 Spring AOP 框架。

最后重点介绍了贫血模型、Rich Domain Object 模型，以及几种简化的模型，并分别分析了几种模型各自的优缺点。

# 第9章

## 完整实例：消息发布系统

### 本章要点

- 『 架构设计的基本知识
- 『 架构设计的原则
- 『 Domain Object 的设计
- 『 设计 DAO 层
- 『 设计业务逻辑层
- 『 设计 Web 层

该系统是一个消息发布系统，作为一个示范性的应用，本身并没有特别复杂的业务逻辑。注册用户可以发布消息，回复消息，是一种优秀的 J2EE 架构。

系统以 Spring 框架为核心，向下整合 Hibernate 进行持久层访问；向上整合 Struts 按清晰的 MVC 模式控制，可以清晰划分应用的层次，提高系统灵活性，提高代码的可扩展、可维护及可复用性等。

本系统示范了一种非常优秀的 J2EE 应用架构，并涉及到如下开源框架：Hibernate、Spring 及 Struts 等。系统的结构清晰、灵活，具有很高的伸缩性，完全能面对复杂多变的业务需求。

## 9.1 系统架构说明

本系统不仅严格按 MVC 模式设计，还按 J2EE 分层设计，将中间层严格分成业务逻辑层、DAO 层及数据持久层等。MVC 层的控制器绝对禁止持久层访问，甚至不参与业务逻辑的实现。

表现层采用传统 JSP 技术，但页面禁止使用 JSP 脚本，从而可以避免将 JSP 页面变得凌乱。JSP 技术结合 Struts 的标签库，让应用的表现层层次清晰，可读性极好。

### 9.1.1 系统架构说明

本系统采用的是典型的 J2EE 三层结构，分为表现层、中间层（业务逻辑层）和数据服务层。三层体系将业务规则、数据访问及合法性校验等工作放在中间层处理。客户端不直接与数据库交互，而是通过组件与中间层建立连接，再由中间层与数据库交互。

该系统的表现层是传统的 JSP 技术，JSP 技术自 1999 年问世以来，经过多年的发展，其广泛的应用和稳定的表现，为其作为表现层技术打下了坚实的基础。

中间层采用的是流行的 Spring+Hibernate，为了将控制层与业务逻辑层分离，又细分为以下几种。

Web 层，就是 MVC 模式里面的“C”（controller），负责控制业务逻辑层与表现层的交互，调用业务逻辑层，并将业务数据返回给表现层作组织表现，该系统的 MVC 框架采用 Struts。

Service 层（就是业务逻辑层），负责实现业务逻辑。业务逻辑层以 DAO 层为基础，通过对 DAO 组件的正面模式包装，完成系统所要求的业务逻辑。

DAO 层，负责与持久化对象交互。该层封装了数据的增、删、查、改的操作。

PO，持久化对象。通过实体关系映射工具将关系型数据库的数据映射成对象，很方便地实现以面向对象方式操作数据库，该系统采用 Hibernate 作为 ORM 框架。

Spring 的作用贯穿了整个中间层，将 Web 层、Service 层、DAO 层及 PO 无缝整合，其数据服务层用来存放数据。

### 9.1.2 采用架构的优势

不可否认，对于简单的应用，采用 ASP 或者 PHP 的开发效率比采用 J2EE 框架的开发效率要高。甚至有人会觉得：这种分层的结构，比一般采用 JSP + Servlet 的系统开发效率还要低。

笔者从以下几个角度来阐述这个问题。

- **开发效率：**软件工程是个特殊的行业，不同于传统的工业，如电器、建筑及汽车行业。这些行业的产品一旦开发出来，交付用户使用后将很少需要后续的维护。

但软件行业不同，软件产品的后期运行维护是个巨大的工程，单纯从前期开发时间上考虑其开发效率是不理智的，也是不公平的。众所周知，对于传统的 ASP 和 PHP 等脚本站点技术，将整个站点的业务逻辑和表现逻辑都混杂在 ASP 或 PHP 页面里，从而导致页面的可读性相当差，可维护性非常低。即使需要简单改变页面的按钮，也不得不打开页面文件，冒着破坏系统的风险。但采用严格分层 J2EE 架构，则可完全避免这个问题。对表现层的修改即使发生错误，也绝对不会将错误扩展到业务逻辑层，更不会影响持久层。因此，采用 J2EE 分层架构，即使前期的开发效率稍微低一点，但也是值得的。

- 需求的变更：以笔者多年的开发经验来看，很少有软件产品的需求从一开始就完全是固定的。客户对软件需求，是随着软件开发过程的深入，不断明晰起来的。因此，常常遇到软件开发到一定程度时，由于客户对软件需求发生了变化，使得软件的实现不得不随之改变。当软件实现需要改变时，是否可以尽可能多地保留软件的部分，尽可能少地改变软件的实现，从而满足客户需求的变更？答案是——采用优秀的解耦架构。这种架构就是 J2EE 的分层架构，在优秀的分层架构里，控制层依赖于业务逻辑层，但绝不与任何具体的业务逻辑组件耦合，只与接口耦合；同样，业务逻辑层依赖于 DAO 层，也不会与任何具体的 DAO 组件耦合，而是面向接口编程。采用这种方式的软件实现，即使软件的部分发生改变，其他部分也尽可能不要改变。

**注意：**即使在传统的硬件行业，也有大量的接口规范。例如，PCI 接口、显卡或者网卡，只要其遵守 PCI 的规范，就可以插入主板，与主板通信。至于这块卡内部的实现，不是主板所关心的，这也正是面向接口编程的好处。假如需要提高电脑的性能，需要更新显卡，只要更换另一块 PCI 接口的显卡，而不是将整台电脑抛弃。如果一台电脑不是采用各种接口组合在一起，而是做成整块，那将意味着即使只需要更新网卡，也要放弃整台电脑。同样，对于软件中的一个个组件，当一个组件需要重构时，尽量不要影响到其他组件。实际上，这是最理想的情况，即使采用目前最优秀的架构，也会有或多或少的影响，这也是软件工程需要努力提高的地方。

- 技术的更新，系统重构：软件行业的技术更新很快，虽然软件行业的发展不快，但小范围的技术更新特别快。一旦由于客观环境的变化，不得不更换技术时，如何保证系统的改变最小呢？答案还是选择优秀的架构。

在传统的 Model 1 的程序结构中，只要有一点小的需求发生改变，将意味着放弃整个页面或者改写。虽然前期的开发速度快，除非可以保证以后永远不会改变应用的结构，否则不要采用 Model 1 的结构。

## 9.2 Hibernate 持久层

采用 Hibernate 作为持久层技术的最大的好处在于：可以完全以面向对象的方式进行系统分析、系统设计。面向对象的分析和面向对象的设计才最接近于程序员的自然思维。

Hibernate 的功能十分强大，对于 Hibernate 的介绍，在这里仅介绍与本节内容相关的技术，如需了解 Hibernate 的详细情况，请参考 Hibernate 的官方文档和相关书籍。

### 9.2.1 编写 PO 类

下面介绍系统的类图，在开发过程中可以根据类图，生成关系型数据库表；或者先把数据库的业务表设计好，再通过工具生成对象。有很多设计工具都可以实现以上功能，如 PowerDesigner 等。但从面向对象分析与设计来讲，推荐使用第一种方法，因为更贴近面向对象的思想。映射配置文件写好以后，我们也可以用 Hibernate 生成数据库的表。

图 9.1 显示了本系统 PO 的类图。

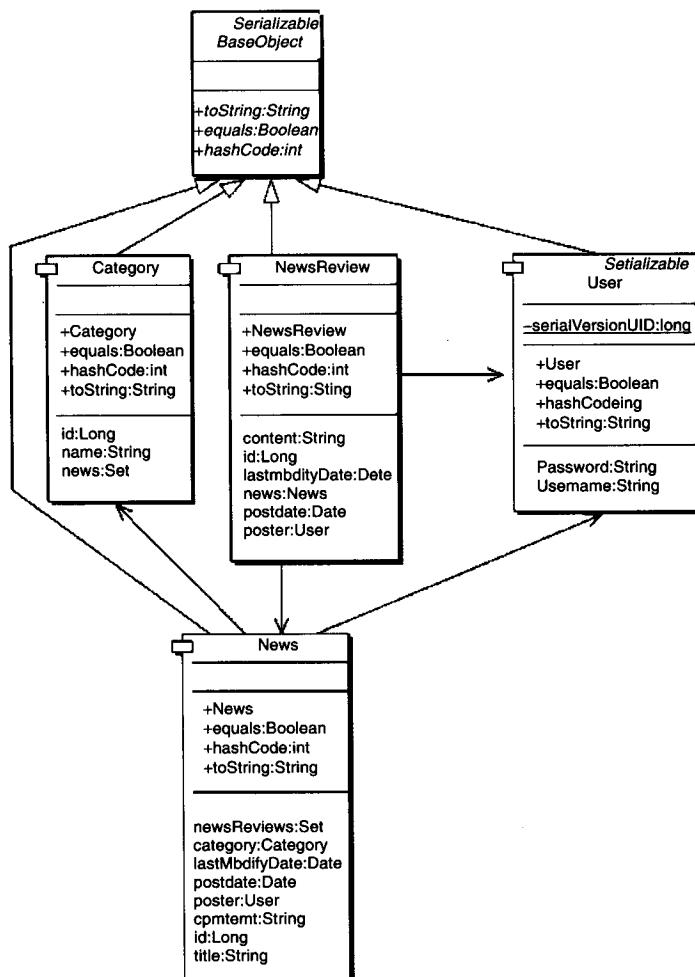


图 9.1 系统 PO 的类图

从类图可以看出，根据要实现的功能，系统的模型 Model 实现类有四个：Category, News, NewsReview 和 User，它们均继承父类 BaseObject，都是普通的 JavaBean，下面是

四个基本的 Persistent Object 类。

- News: 封装了一条消息。包括标题、内容、发布时间及发布人等。
- Category: 封装了一个消息分类。
- User: 封装了一个用户的信息。
- NewsReview: 封装了一条消息评论。

其中，News 有一个 Category 类型的成员变量及 User 类型的成员变量；NewsReview 有一个 News 类型的成员变量和一个 User 类型的成员变量。为了能够使用双向关联（Hibernate 的映射功能，稍后解释），Category 有一个集合型成员变量，用于存放与这个 Category 对象有关联的 News 对象；同样 News 也有一个集合型成员变量，用于存放与这个 News 对象有关联的 NewsReview 对象。

实际上，持久化就是通过成员变量来映射关系数据库里的 1-N 和 N-N 的关系。

下面是 PO 父类 BaseObject 的代码：

```
//将父类声明为 abstract 类
public abstract class BaseObject implements Serializable
{
    //PO 推荐实现的 toString 方法
    public abstract String toString();
    //PO 推荐实现的 equals 方法
    public abstract boolean equals(Object o);
    //PO 推荐实现的 hashCode 方法
    public abstract int hashCode();
}
```

父类 BaseObject 是一个抽象类，定义了三个抽象方法 `toString()`、`equals()` 和 `hashCode()`，这三个方法是 Hibernate 推荐持久化对象时重写的。

关系数据库里的记录可以由主键来唯一标识，但是用什么标准来标记两个对象“相等”呢？两个对象相等与否的判断结果很可能影响到数据的完整性。如果在 Hibernate 无法自行制定两个对象相等与否的标准时，则需要用户自行定义即重写 `equals()` 方法。

如果希望将持久化对象放进 Set 里，或者重新接管已脱管（detached）的持久化对象，则必须重写 `equals()` 和 `hashCode()` 这两个方法，因为 Java 语法要求如果两个对象相等，那么它们的 `hashCode()` 返回值必须相等，因此该方法也需要重写。另外，`toString()` 方法用于给出该对象的描述信息，因此也推荐重写该方法（Hibernate 并没有硬性规定）。此外，BaseObject 还实现了 `Serializable` 接口，该接口是持久化对象推荐实现的。

让其他类继承这个抽象类只是一个可选的写法，至少直接让其他类重写 `equals()` 和 `hashCode()` 方法来实现 `Serializable` 接口也是可以的。

下面具体来看 News 类。

```
/**
 * @hibernate.class table="news"
 * @struts.form include-all="false" extends="BaseForm"
 */
public class News extends BaseObject
{
```

```
//标识属性
private Long id;
//消息标题
private String title;
//消息内容
private String content;
//用于关联发布人，对应另一个持久化类
private User poster;
//发布日期
private Date postDate;
//最后一次回复日期
private Date lastModifyDate;
//消息分类，用于关联另一个持久化类
private Category category;
//用于关联消息回复
private Set newsReviews;
//无参数的构造器
public News()
{
}
/**
 * 用于获取与此消息相关的全部回复
 * @return 返回消息的全部回复
 */
public Set getNewsReviews()
{
    return newsReviews;
}
/**
 * 设置消息关联的回复
 * @param newsReview 消息关联的全部回复
 */
public void setNewsReviews(Set newsReviews)
{
    this.newsReviews = newsReviews;
}
/**
 * 返回消息所属的种类
 * @return 消息所属的种类
 * @hibernate.many-to-one column="id" not-null="true"
 */
public Category getCategory()
{
    return category;
}
/**
 * 设置消息所在的种类
 * @param 消息所属的种类
 */
public void setCategory(Category category)
{
    this.category = category;
}
/**
 * 返回消息的最后评论日期
 * @return 消息的最后评论日期
 * @hibernate.property column="last_modify_date" not-null="true"
 */
public Date getLastModifyDate()
```

```
{  
    return lastModifyDate;  
}  
/**  
 * 设置消息的最后评论时间  
 * @param 消息的最后评论日期  
 */  
public void setLastModifyDate(Date lastModifyDate)  
{  
    this.lastModifyDate = lastModifyDate;  
}  
/**  
 * 返回消息的发布日期  
 * @return 消息的发布日期  
 * @hibernate.property column="post_date" not-null="true"  
 */  
public Date getPostDate()  
{  
    return postDate;  
}  
/**  
 * 设置消息的发布日期  
 * @param 消息的发布日期  
 */  
public void setPostDate(Date postDate)  
{  
    this.postDate = postDate;  
}  
  
/**  
 * 返回消息的发布者  
 * @return 消息的发布者  
 * @hibernate.many-to-one column="username" not-null="true"  
 */  
public User getPoster()  
{  
    return poster;  
}  
/**  
 * 设置消息的发布者  
 * @param 消息的发布者  
 * @hibernate.many-to-one column="username" not-null="true"  
 */  
public void setPoster(User poster)  
{  
    this.poster = poster;  
}  
/**  
 * 返回消息的内容  
 * @return 消息的内容  
 * @hibernate.property column="content" length="3000" not-null="true"  
 */  
public String getContent()  
{  
    return content;  
}  
/**  
 * 设置消息的内容  
 * @param 消息的内容
```

```
/*
public void setContent(String content)
{
    this.content = content;
}
/***
 * 返回消息的 id
 * @return 消息的 id
 * @hibernate.id column="id" generator-class="increment"
 *                 unsaved-value="null"
 */
public Long getId()
{
    return id;
}
/***
 * 设置消息的 id
 * @param 消息的 id
 */
public void setId(Long id)
{
    this.id = id;
}
/***
 * 返回消息的标题
 * @return 消息的标题
 * @hibernate.property column="title" length="50" not-null="true"
 */
public String getTitle()
{
    return title;
}
/***
 * 设置消息的标题
 * @param 消息的标题
 */
public void setTitle(String title)
{
    this.title = title;
}

/**
 * 实现抽象父类 BaseObject 的 equals 方法
 * @see java.lang.Object#equals(Object)
 */
public boolean equals(Object object)
{
    if (!(object instanceof News))
    {
        return false;
    }
    News rhs = (News) object;
    return this.poster.equals(rhs.getPoster())
           && this.postDate.equals(rhs.getPostDate());
}
/***
 * 实现抽象父类 BaseObject 的 hashCode 方法
 * @see java.lang.Object#hashCode()
 */

```

```
public int hashCode()
{
    return this.poster.hashCode() + this.postDate.hashCode();
}
/**
 * 实现抽象父类 BaseObject 的 hashCode 的方法
 * @see java.lang.Object#toString()
 */
public String toString()
{
    return new ToStringBuilder(this).append("id", this.id).append("title",
        this.title).append("postDate", this.postDate).append("content",
        this.content).append("lastModifyDate", this.lastModifyDate)
        .append("poster", this.poster)
        .append("category", this.category).append("newsReviews",
        this.newsReviews).toString();
}
}
```

Hibernate 对 PO 几乎不作任何要求，一般的 POJO (Plain Old Java Object) 就可以充当 PO。只要有一个默认的不带参数的构造器就可以。推荐实现 Serializable 接口时，最好重写 equals() 和 hashCode() 方法。

所谓的 POJO 就是只有属性，以及属性对应 getter 和 setter 方法的 Java 对象。

下面将解释各个属性的含义（后面的映射配置文件会有进一步探讨）：每个 News 对象就相当于数据库表里的一条记录，其中 id 属性映射的是记录的主键；title 与 content 分别是消息的标题和内容；postDate 是发布时间；lastModifyDate 是最后评论时间；category 则是 News 关联（在数据库里通过外键关联）的 Category 对象；newsReviews 则是 News 对象关联的所有 NewsReview 对象。

细心的读者会发现，上面的代码重写 equals() 和 hashCode() 方法时，并不是采用 id 作为判断这两个对象相等的标准。这与 Hibernate 的机制有关，因为对象的标识属性值由 Hibernate 负责生成，Hibernate 仅对已经持久化的对象分配标识属性值，未被持久化的对象是没有标识属性值的。如果在 Set 里面的一个新建对象未被持久化，此时持久化该对象并分配主键。由于 equals() 和 hashCode() 方法都基于主键，因此，hashCode 得出的结果会改变，在 Set 里面就可能起冲突。

因此建议使用“业务键”来作为 equals() 和 hashCode() 方法的基准。“业务键”指的是能在真实世界里区分两个对象的属性。例如，News 就采用了发布人（poster）跟发布时间（postDate）的组合作为业务键，因为同一用户不可能在同一时间发布多条消息。关于如何选择业务键的更多信息请参照 Hibernate 官方文档。

注意：除了由用户自己生成 equals() 和 hashCode() 方法外，也有一些工具可以自动生成这两个方法。例如，Eclipse 插件 commonclipse。有兴趣的读者可以尝试一下。

## 9.2.2 编写 PO 的映射配置文件

仅有一个 POJO 是无法完成数据库的持久化操作的，还必须为 POJO 增加映射文件。

当 POJO 增加映射文件后，可以完成 O/R Mapping，从而在某个特定对象的管理下完成数据库访问。

因此必须为这个 POJO 配上一个映射文件，通常将这个映射文件命名为：类名.hbm.xml，并与这个类放置在同一目录下。News.hbm.xml 的具体内容如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!-- 映射 News 持久化类-->
    <class name="org.yeeku.model.News" table="news">
        <!-- 映射标识属性-->
        <id name="id" column="id" unsaved-value="null">
            <!-- 定义主键生成器策略-->
            <generator class="increment">
            </generator>
        </id>
        <!-- 映射关联类：Category-->
        <many-to-one name="category"
          class="org.yeeku.model.Category" column="category_id" not-null="true">
        </many-to-one>
        <!-- 映射最后评论日期属性-->
        <property name="lastModifyDate" column="last_modify_date" not-null="true">
        </property>
        <!-- 映射发布日期属性-->
        <property name="postDate" column="post_date" not-null="true">
        </property>
        <!-- 映射关联类：User-->
        <many-to-one name="poster" column="username" not-null="true">
        </many-to-one>
        <!-- 映射消息内容属性-->
        <property name="content" column="content" length="3000" not-null="true">
        </property>
        <!-- 映射消息标题属性-->
        <property name="title" column="title" length="50" not-null="true">
        </property>
        <!-- 映射关联类：NewsReview，映射 1-N 关联-->
        <set name="newsReviews" lazy="false" inverse="true" cascade="all-
delete-orphan">
            <meta attribute="field-description">
                @hibernate.list lazy="true" inverse="false" cascade="none"
                @hibernate.collection-key column="id"
                @hibernate.collection-one-to-many class="org.yeeku.model.
NewsReview"
            </meta>
            <key>
                <column name="news_id" />
            </key>
            <one-to-many class="org.yeeku.model.NewsReview" />
        </set>
    </class>
</hibernate-mapping>
```

映射文件根元素 `hibernate-mapping` 下面可以有多个 `class` 子元素，即在一个映射文件里可以配置多个映射对象，但为了清晰起见，建议为每个类单独写一个映射配置文件。

`class` 的 `name` 属性就是要映射的对象类；`table` 是数据库里对应的表名；`class` 下面的子元素就是这个类的属性并与数据库里的字段相对应。

`id` 用于唯一标识该对象，称为标识属性。其中 `name` 属性是类里面的属性名；`column` 是数据库的对应字段。另外，`id` 还有 `generator` 子元素，用于指定主键生成方式，这里用的是自动增长生成（`increment`）策略，其他方式请参照 Hibernate 文档。

这里重点介绍 1-N（`set` 是其中一种方式）和 N-1（many-to-one，通常就是外键关联）的映射。首先来看 1-N 关系，一条消息对应多条评论。`name` 属性是指类里面存放“N”的一方对象的属性。`meta` 一栏这里只是用作为阅读者提供额外信息。`key` 就是“N”的一方存放“1”的一方外键的字段，`one-to-many` 的 `class` 是“N”的一方的映射对象。

**注意：**`lazy` 属性与 `inverse` 属性的设置。`lazy` 是 Hibernate 的一种延迟加载数据策略，比如说一个 `News` 对象与多个 `NewsReview` 对象关联，如果使用 `lazy` 策略，则只有当 `NewsReview` 对象被请求时（如调用 `News` 的 `getNewsReviews` 方法）才会从数据库里读取相应记录，从而达到优化性能的目的。特别是系统如果非常复杂，关联很多时，不使用延迟加载将会对性能有比较大的影响。当然使用延迟加载还有很多问题需要注意，这里为了方便起见设为 `false`，有兴趣的读者可以参考 Hibernate 官方文档。`inverse` 属性标明该元素是否控制关联关系，对于 1-N 的关系，通常推荐由“N”的一端控制关联关系，因此 `set` 元素通常都应包含 `inverse=“true”` 属性。

N-1 的配置则比较简单，与普通属性的配置一样，依此类推将所有映射文件完成，可以参照 `src/org/yeeku/model/NewsReview.hbm.xml`、`Category.hbm.xml` 和 `User.hbm.xml`。完成后可以通过配置 Hibernate 的属性（数据库连接驱动，数据库方言，登录用户名与密码等）自动生成数据库的表，这里不再赘述。

**注意：**除了手工编写以外，开发者还有一些工具可以帮助生成映射配置文件。例如，Xdoclet 就可以用来生成 Hibernate 和 Struts 等配置文件，只需在编写源代码时在适当的地方加上注释，再用 Xdoclet 生成即可。

具体例子参看上面的 `News` 类源码，类定义上包含如下注释：

```
@hibernate.class table="news"
```

标明了该类映射的数据库表是 `news`。所有的 `getter` 方法上面的注释里都有类似的注释，用来标明属性映射的字段。例如：

```
@hibernate.property column="content" length="3000" not-null="true"
```

有兴趣的读者可以对照 `News.hbm.xml` 或者直接参考 Xdoclet 的文档。

## 9.2.3 连接数据库

除了前面介绍的 POJO 和映射文件之外，Hibernate 不知道与哪个数据库连接，也不知道连接数据库时需要哪些属性。因此，Hibernate 控制数据库连接提供了两种方式：

- 采用 `hibernate.properties` 属性文件。
- 采用 `hibernate.cfg.xml` 配置文件。

这两种方式只是形式上的区别，实质的内容没有任何改变。都需要指定连接数据库的 URL、数据库的驱动、用户名及密码等基本信息。如果需要使用连接池，则还应确定连接池配置信息。

### 1. 直接 Hibernate 的配置连接

直接 Hibernate 的配置连接就是使用 `hibernate.cfg.xml` 文件，关于使用 `hibernate.properties` 的方式与此类似，不再赘述。

下面是使用 C3P0 连接池的 `hibernate.cfg.xml` 配置文件的源代码：

```
<?xml version="1.0" encoding="GBK"?>
<!-- Hibernate 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://.hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<!-- Hibernate 配置文件的根元素-->
<hibernate-configuration>
  <!-- 配置 SessionFactory-->
  <session-factory>
    <!-- 配置连接数据库的 URL-->
    <property name="connection.url">jdbc:mysql://localhost/j2ee</
property>
    <!-- 配置连接数据库的用户名-->
    <property name="connection.username">root</property>
    <!-- 配置连接数据库的密码-->
    <property name="connection.password">32147</property>
    <!-- 配置连接数据库的驱动-->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</
property>
    <!-- 配置连接数据库的方言-->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</
property>
    <!-- JDBC 连接池 (use the C3P0) -->
    <!-- 连接池的最大连接数-->
    <property name="hibernate.c3p0.max_size">500</property>
    <!-- 连接池的最小连接数-->
    <property name="hibernate.c3p0.min_size">2</property>
    <!-- 连接池的超时时长-->
    <property name="hibernate.c3p0.timeout">5000</property>
    <!-- 连接池的缓存 statement 的数量-->
    <property name="hibernate.c3p0.max_statements">100</property>
    <property name="hibernate.c3p0.idle_test_period">3000</property>
    <!-- 连接池每次获取连接的数量-->
    <property name="hibernate.c3p0.acquire_increment">2</property>
    <property name="hibernate.c3p0.validate">true</property>
```

```

<!-- 每次执行持久化操作，是否显示对应的 SQL 语句-->
<property name="hibernate.show_sql">true</property>
<!-- 是否在启动时，重新创建数据库 -->
<property name="hbm2ddl.auto">update</property>
<!-- 此处列出所有的映射文件-->
<mapping resource="Person.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

Hibernate 通过上面的配置文件，知道如何控制与数据库的连接，以及采用连接池的方式。

## 2. 采用 Spring 管理 Hibernate

让 Spring 管理 Hibernate 时，必须将 Hibernate 的 SessionFactory 配置在 Spring 容器内，此时有两种做法：

- 在 Spring 容器中配置 SessionFactory。
- 让 hibernate.cfg.xml 文件控制 SessionFactory。

这两种做法的效果相似，都是利用 Spring 的 LocalSessionFactoryBean，负责产生 Hibernate 的 SessionFactory，该 bean 作为其他持久化访问组件的属性注入。

下面是完成在 Spring 容器配置 Hibernate 的 SessionFactory 的形式，该项目采用以下方式：

```

<beans>
    <!-- 配置数据源，使用 DBCP 数据源-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!-- MySQL 数据库的驱动-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <!-- 数据库的 URL-->
        <property name="url" value="jdbc:mysql:///newsboard"/>
        <!-- 指定数据库的用户名-->
        <property name="username" value="root"/>
        <!-- 指定数据库的密码-->
        <property name="password" value="123"/>
        <!-- 指定数据库的最大连接数-->
        <property name="maxActive" value="100"/>
        <!-- 指定数据库的最大空闲连接数-->
        <property name="maxIdle" value="30"/>
        <!-- 指定数据库的最大等待数-->
        <property name="maxWait" value="1000"/>
        <!-- 指定数据库的默认自动提交-->
        <property name="defaultAutoCommit" value="true"/>
        <!-- 指定数据库的连接超时时是否启动删除-->
        <property name="removeAbandoned" value="true"/>
        <!-- 指定数据库的删除数据库连接的超时时长-->
        <property name="removeAbandonedTimeout" value="60"/>
        <property name="logAbandoned" value="true"/>
    </bean>
</beans>

```

本系统使用的数据源是 DBCP 的数据源，DBCP 数据源是来自 Apache 组织的一个优秀的数据源。

DBCP 数据源有四个属性值得注意。

- driverClassName：数据库驱动，通常由第三方提供。
- url：数据库 URL，不同数据库的写法存在差异。
- username：数据库的用户名。
- password：数据库的密码。

配置不同的数据库时，这四个属性通常需要改动。driverClassName 由数据库厂商提供，url 的写法也由厂商决定，请参照数据库的文档。这里的数据源使用的是 MySQL，连接的是 newsboard 数据库。

另外，Spring 提供了对 Hibernate 的 SessionFactory 的管理，只需在 applicationContext.xml 文件中增加如下配置：

```
<!-- 配置 Hibernate 的 SessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
    <!-- 依赖注入 SessionFactory 所需的 DataSource-->
    <property name="dataSource" ref="dataSource"/>
    <!-- 加载所有的映射文件-->
    <property name="mappingResources">
        <!-- 下面列出所有的持久化映射文件-->
        <list>
            <value>org/yeku/model/User.hbm.xml</value>
            <value>org/yeku/model/News.hbm.xml</value>
            <value>org/yeku/model/NewsReview.hbm.xml</value>
            <value>org/yeku/model/Category.hbm.xml</value>
        </list>
    </property>
    <!-- 下面指定 Hibernate 的属性-->
    <property name="hibernateProperties">
        <props>
            <!-- 下面指定 Hibernate 使用的数据库方法-->
            <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDBDialect</prop>
        </props>
    </property>
</bean>
```

其中<property name="dataSource" ref="dataSource"/>就是我们刚才配置的数据源。mappingResources 是我们编写好的映射配置文件。除此之外还需要在 hibernateProperties 里配置 Hibernate 访问数据库使用的方法，到此为止，已经基本完成了 Spring 与 Hibernate 的整合。

如果需要让 hibernate.cfg.xml 文件自己负责 Hibernate SessionFactory 的配置，可以采用如下方式：同样使用 LocalSessionFactoryBean 生成 SessionFactory，但此时无须确定数据库用户名及密码等具体信息，只需在配置文件中确定 hibernate.cfg.xml 文件的位置。

配置代码如下：

```
<!-- 配置 Hibernate 的 SessionFactory-->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
    <!-- 确定配置文件的位置-->
    <property name="configLocation">
```

```
<value>classpath:hibernate.cfg.xml</value>
</property>
</bean>
```

## 9.3 DAO 组件层

业务逻辑层组件依赖于持久层组件，而持久层组件则提供数据表的基本 CRUD 操作。通过持久层组件，使业务逻辑层组件的实现与特定数据库访问分离，从而提高系统的解耦。

### 9.3.1 DAO 组件的结构

通过第 8 章关于系统架构的介绍可知，DAO 模式需要为每个 DAO 组件编写 DAO 接口，同时至少提供一个实现类，根据不同需要，可能有多个实现类。

为了让逻辑组件与具体的 DAO 组件分离，还必须有一个 DAO 工厂。

图 9.2 是本系统的 DAO 组件接口的类图。图 9.3 是本系统 DAO 组件实现类的类图。此处的实现类都是基于 Hibernate 的实现。

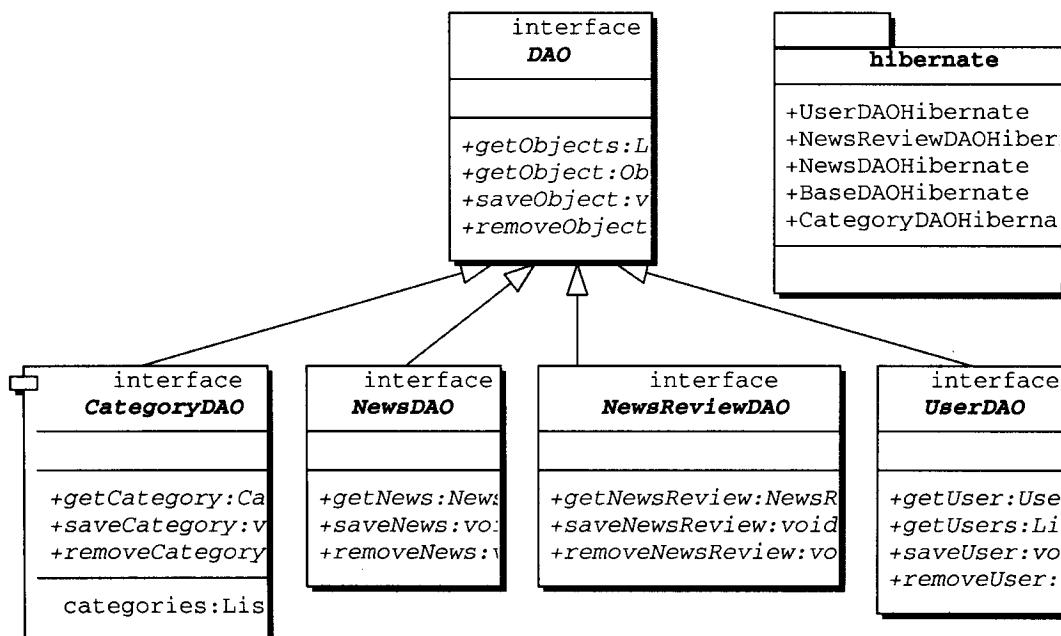


图 9.2 DAO 组件接口的类图

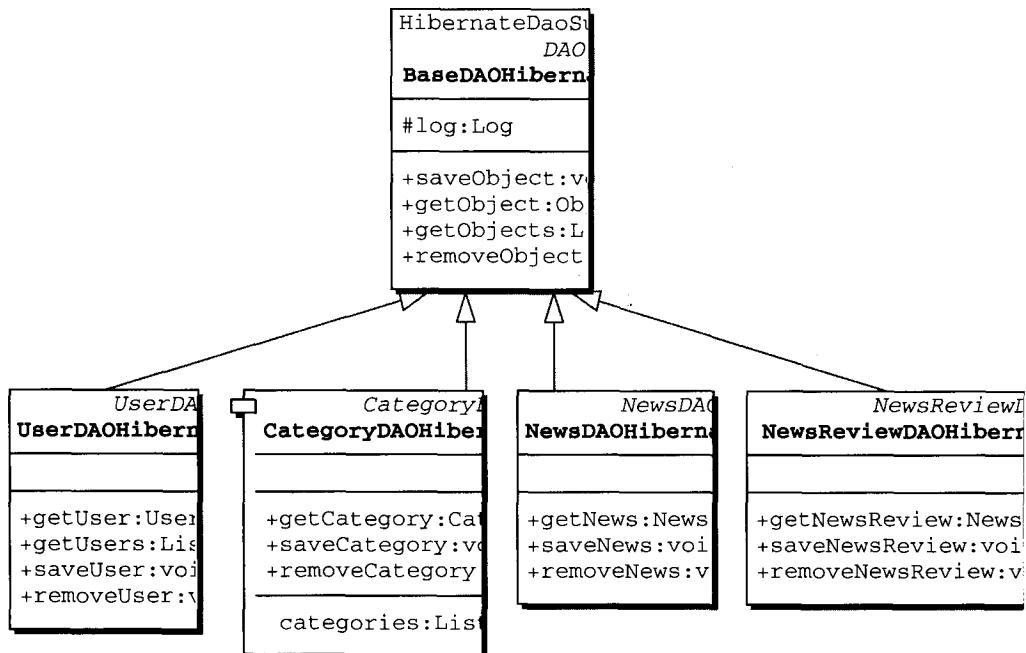


图 9.3 DAO 组件实现类的类图

### 9.3.2 编写 DAO 接口

DAO 接口包类里有 5 个 DAO，其中 DAO 是一个具有一般性的 DAO 接口，其他 DAO 接口都继承这个 DAO。这里选择 DAO 与 NewsDAO 作为示例。

DAO 接口的代码如下：

```

public interface DAO
{
    //返回全部的物品
    public List getObjects(Class class);
    //根据 id 返回某个持久化类
    public Object getObject(Class class, Serializable id);
    //保存持久化类
    public void saveObject(Object o);
    //删除某个持久化类
    public void removeObject(Class class, Serializable id);
}

```

下面对各个方法进行简要说明。

- `getObject` 是获取所有该类型对象的方法。输入参数是某个类的类型，相当于取出一个表所有行的 sql 语句。
- `getObject` 封装的是读取单条记录的操作。
- `saveObject` 封装的是插入或更新单条记录的操作。插入或更新取决于该对象是否已被持久化过。

- `removeObject` 封装的是删除单条记录的操作。

由此可见，DAO 的实质就是对数据表的 CRUD 原子操作。

一般的 DAO 的实现要求用到 Java 的反射机制，因此对于初学者不容易实现。但实现之后可以通过实现类对任何 PO 进行操作。通常，我们用每个类各自的 DAO 对该类进行 CRUD 操作。

NewsDAO 接口的代码如下：

```
public interface NewsDAO extends DAO
{
    //根据 id 返回 News 持久化类
    public News getNews(Long newsId);
    //保存消息
    public void saveNews(News news);
    //删除消息
    public void removeNews(Long newsId);
}
```

可以看到 NewsDAO 里并没有封装取出所有 News 的操作，其实是否封装该操作取决于业务需求。在这个系统里面没有取出所有 News 的需求，因此可以不封装。如果要考虑扩展系统，也可以先封装，在扩展时使用。

**注意：**接口的使用在这里起了很重要的作用。接口就是定义与实现的分离，在一个面向对象的系统中，其功能是由很多对象协作完成的。各个对象是如何实现的，对系统设计人员来讲是透明的；而各个对象之间的协作关系则成为系统设计的关键。小到不同类之间的通信，大到各模块之间的交互，在系统设计之初都要着重考虑，这也是系统设计的主要工作内容。然而通过多态引入接口可以降低对象之间的依赖，解除耦合。很多经典的设计模式都离不开接口，因此，更深层次理解接口是提高技术水平的必不可少的步骤，有兴趣的读者可以参考相关设计模式方面的书籍。

下面依次是 UserDao、CategoryDao 及 NewsReviewDao 的源代码：

UserDao 的源代码：

```
public interface UserDao extends DAO {
    /**
     * 根据用户名获取 User
     * @param username 需要访问的用户名
     * @return 用户名对应的用户
     */
    public User getUser(String username);
    /**
     * 保存用户
     * @param 需要保存的用户
     */
    public void saveUser(User user);
    /**
     * 根据用户名，删除用户，
     * @param username 需要删除用户的用户名
     */
    public void removeUser(String username);
```

CategoryDAO 的源代码：

```
public interface CategoryDAO extends DAO {
    /**
     * 根据 id 获取种类
     * @parameter categoryId 需要加载的种类 id
     * @return 种类 id 对应的种类
     */
    public Category getCategory(Long categoryId);
    /**
     * 保存消息分类
     * @parameter category 需要保存的消息分类
     */
    public void saveCategory(Category category);
    /**
     * 根据消息 id 删除消息分类
     * @categoryid 需要删除的消息分类 id
     */
    public void removeCategory(Long categoryId);
    /**
     * 获取全部消息分类
     * @return 返回数据库中全部消息分类
     */
    public List getCategories();
}
```

NewsReviewDAO 的源代码：

```
public interface NewsReviewDAO extends DAO{
    /**
     * 根据回复 id 获取消息回复
     * @return id 对应的消息回复
     */
    public NewsReview getNewsReview(Long newsReviewId);
    /**
     * 保存特定的消息回复
     * @parameter newsReview 需要保存的消息回复。
     */
    public void saveNewsReview(NewsReview newsReview);
    /**
     * 根据回复 id 删除消息回复
     * @parameter newsReviewId 需删除的消息回复 id
     */
    public void removeNewsReview(Long newsReviewId);
}
```

### 9.3.3 编写 DAO 的具体实现

编写 DAO 的具体实现在这里也很简单，下面是 NewsDAO 接口的 Hibernate 实现的源代码：

```
NewsDAOHibernate 继承 BaseDAOHibernate，实现了 NewsDAO
public class NewsDAOHibernate extends BaseDAOHibernate
    implements NewsDAO
{
```

```
//根据主键查找 News 持久化对象
public News getNews(Long id)
{
    News news = (News) getHibernateTemplate().get(News.class, id);
    //如果不能加载该持久化对象，抛出异常
    if (news == null)
    {
        throw new ObjectRetrievalFailureException(News.class, id);
    }
    return news;
}
//保存消息
public void saveNews(News news)
{
    getHibernateTemplate().saveOrUpdate(news);
}
//根据主键删除消息
public void removeNews(Long id)
{
    getHibernateTemplate().delete(getNews(id));
}
}
```

NewsDAO 的 Hibernate 实现继承了 BaseDAOHibernate，下面是 BaseDAOHibernate 的源代码：

```
public class BaseDAOHibernate extends HibernateDaoSupport
    implements DAO
{
    //提供日志功能
    protected final Log log = LogFactory.getLog(getClass());
    //保存对象
    public void saveObject(Object o)
    {
        getHibernateTemplate().saveOrUpdate(o);
    }
    //根据持久化类的类，主键获取对象
    public Object getObject(Class clazz, Serializable id)
    {
        Object o = getHibernateTemplate().get(clazz, id);
        //如果没有获得该对象
        if (o == null)
        {
            throw new ObjectRetrievalFailureException(clazz, id);
        }
        return o;
    }
    //查找某个持久化类的全部实例，查找到数据库对应表的全部记录
    public List getObjects(Class clazz)
    {
        return getHibernateTemplate().loadAll(clazz);
    }
    //根据主键，删除某个持久化类的特定实例，对应删除数据库的特定记录
    public void removeObject(Class clazz, Serializable id)
    {
        getHibernateTemplate().delete(getObject(clazz, id));
    }
}
```

**BaseDAOHibernate** 就是前面所说的一般 DAO 接口的实现，里面附加了日志（log）功能，增强了系统的可维护性和可管理性。这里的 DAO 实现都很简捷，只需要简单的一行代码就分别实现 CRUD 操作。这得益于 Spring 的强大功能：Spring 提供了 **HibernateDaoSupport** 工具类以及 **HibernateTemplate**，其中 **HibernateTemplate** 允许以模板化方式的操作访问数据库。

注意：前面提到 DAO 需要封装数据源，而在这个 DAO 里面我们感觉不到数据源的存在，这也得益于 Spring 和 Hibernate 的封装，对数据源的访问放在配置文件里管理。

下面依次是 **UserDAO**、**CategoryDAO** 和 **NewsReviewDAO** 实现类的源代码。这些实现类都基于 Spring 的 Hibernate 支持，因此非常简单。

**UserDAOHibernate** 的源代码如下：

```
public class UserDAOHibernate extends BaseDAOHibernate
    implements UserDAO
{
    /**
     * 根据用户名获取 User
     * @param username 需要访问的用户名
     * @return 用户名对应的用户
     */
    public User getUser(String username) {
        User user = (User) getHibernateTemplate().get(User.class, username);
        if(user == null){
            log.warn("user '" + username + "' not found...");
        }
        return user;
    }
    /**
     * 保存用户
     * @param 需要保存的用户
     */
    public void saveUser(final User user) {
        if (log.isDebugEnabled()) {
            log.debug("user's id: " + user.getUsername());
        }
        getHibernateTemplate().saveOrUpdate(user);
        // necessary to throw a DataIntegrityViolation and catch it in UserManager
        getHibernateTemplate().flush();
    }
    /**
     * 根据用户名，删除用户，
     * @param username 需要删除用户的用户名
     */
    public void removeUser(String username) {
        getHibernateTemplate().delete(getUser(username));
    }
}
```

**CategoryDAOHibernate** 的源代码如下：

```
public class CategoryDAOHibernate extends BaseDAOHibernate
    implements CategoryDAO
{
```

```
/*
 * 根据 id 获取种类
 * @parameter categoryId 需要加载的种类 id
 * @return 种类 id 对应的种类
 */
public Category getCategory(Long id) {
    Category category = (Category) getHibernateTemplate().get(
        Category.class, id);
    if (category == null) {
        throw new ObjectRetrievalFailureException(Category.class, id);
    }
    return category;
}
/**
 * 保存消息分类
 * @parameter category 需要保存的消息分类
 */
public void saveCategory(Category category) {
    getHibernateTemplate().saveOrUpdate(category);
}
/**
 * 根据消息 id 删除消息分类
 * @categoryID 需要删除的消息分类 id
 */
public void removeCategory(Long id) {
    getHibernateTemplate().delete(getCategory(id));
}
/**
 * 获取全部消息分类
 * @return 返回数据库中全部消息分类
 */
public List getCategories() {
    return getHibernateTemplate().find("from Category");
}
}
```

NewsReviewDAOHibernate 的源代码如下：

```
public class NewsReviewDAOHibernate extends BaseDAOHibernate
implements NewsReviewDAO
{
    /**
     * 根据回复 id 获取消息回复
     * @return id 对应的消息回复
     */
    public NewsReview getNewsReview(Long id) {
        NewsReview newsReview = (NewsReview) getHibernateTemplate().get(
            NewsReview.class, id);
        if (newsReview == null) {
            throw new ObjectRetrievalFailureException(NewsReview.class, id);
        }
        return newsReview;
    }
    /**
     * 保存特定的消息回复
     * @parameter newsReview 需要保存的消息回复
     */
    public void saveNewsReview(NewsReview newsReview) {
```

```

        getHibernateTemplate().saveOrUpdate(newsReview);
    }
    /**
     * 根据回复 id 删除消息回复
     * @parameter newsReviewId 需删除的消息回复 id
     */
    public void removeNewsReview(Long id) {
        getHibernateTemplate().delete(getNewsReview(id));
    }
}
}

```

### 9.3.4 用 Spring 容器代替 DAO 工厂

通常情况下，引入接口就不可避免需要引入工厂来负责 DAO 组件的生成。但根据第 5 章的介绍可知，Spring 实现了两种基本模式：单态模式和工厂模式。而使用 Spring 可以完全避免使用工厂模式，因为 Spring 就是个功能非常强大的工厂。因此，完全可以让 Spring 充当 DAO 工厂。

由 Spring 充当 DAO 工厂时，无须程序员自己实现工厂模式，只需要将 DAO 组件配置在 Spring 容器中，由 ApplicationContext 负责管理 DAO 组件的创建即可。借助于 Spring 提供的依赖注入，其他组件甚至不用访问工厂，一样可以直接使用 DAO 实例。

正如在工厂模式里必须要提供接口的实现类一样，此处的配置必须提供实现类，而不能仅提供接口，下面的配置代码是在 applicationContext.xml 里面配置了 DAO 组件。

```

<!-- 配置 DAO 组件，必须提供 DAO 的实现类-->
<bean id="dao" class="org.yeeku.dao.hibernate.BaseDAOHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 配置 DAO 组件，必须提供 DAO 的实现类-->
<bean id="newsDAO" class="org.yeeku.dao.hibernate.NewsDAOHibernate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

```

在上面配置的 property 子元素里，引用了 Hibernate 的 SessionFactory。其中，SessionFactory 负责产生 Hibernate Session。Hibernate 的持久化操作必须在 Session 管理下完成。NewsDAOHibernate 实现类并没有提供 setSessionFactory 方法，该方法由其父类 HibernateDaoSupport 提供，用于为 DAO 组件依赖注入 SessionFactory。

依此类推，编写其他几个 PO 的 DAO 接口和 Hibernate 实现，将它们配置在 Spring 的容器中。

通常建议将 DAO 组件以单独的配置文件配置，下面是 daoContext.xml 文件的源代码，该文件负责配置所有的 DAO 组件，并使用了继承简化 DAO 组件的配置：

```

<?xml version="1.0" encoding="GBK"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- DAO 模板，用于被其他的 DAO 组件继承 -->

```

```
<bean id="daoTemplate" abstract="true" lazy-init="true">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- Generic DAO - can be used when doing standard CRUD -->
<bean id="dao" class="org.yeeku.dao.hibernate.BaseDAOHibernate"
      parent="daoTemplate"/>
<!-- 配置普通的 DAO 组件: UserDAO: -->
<bean id="userDAO" class="org.yeeku.dao.hibernate.UserDAOHibernate"
      parent="daoTemplate"/>
<!-- 配置普通的 DAO 组件: NewsDAO -->
<bean id="newsDAO" class="org.yeeku.dao.hibernate.NewsDAOHibernate"
      parent="daoTemplate"/>
<!-- 配置普通的 DAO 组件: NewsReviewDAO-->
<bean id="newsReviewDAO" class="org.yeeku.dao.hibernate.NewsReviewDAOHibernate"
      parent="daoTemplate"/>
<!-- 配置普通的 DAO 组件: CategoryDAO -->
<bean id="categoryDAO" class="org.yeeku.dao.hibernate.CategoryDAOHibernate"
      parent="daoTemplate"/>
</beans>
```

至此，我们已经完成了 Spring 与 Hibernate 的整合。

## 9.4 业务逻辑层

业务逻辑层建立在 DAO 层之上，由业务逻辑组件对 DAO 组件进行 Facade 封装。为了分离业务逻辑层与 DAO 层之间的耦合，业务逻辑层应面向接口编程，即业务逻辑组件只调用 DAO 组件的接口，而不与具体的实现类耦合，同时将业务逻辑放在接口中定义。使 Web 层仅仅与业务逻辑组件的接口耦合，而无须理会业务逻辑组件的实现。

### 9.4.1 业务逻辑组件的结构

业务逻辑组件同样分为接口和实现类两个部分，接口用于定义业务逻辑组件，定义业务逻辑组件必须实现的方法是整个系统运行的核心。

在应用中需要多少个业务逻辑组件，往往取决于系统的大小。通常按模块来设计业务逻辑组件，每个模块设计一个业务逻辑组件，并且每个业务逻辑组件以多个 DAO 组件作为基础，从而实现对外提供系统的业务逻辑服务。

图 9.4 显示了业务逻辑组件的接口图。

### 9.4.2 业务逻辑组件的接口

增加业务逻辑组件的接口，也是为了提供更好的解耦。通过面向接口编程，控制器无须与具体的业务逻辑组件耦合，而是面向接口编程。假如需要改变业务逻辑的实现时，可以只提供新的实现类，而不需要改变其控制器代码。

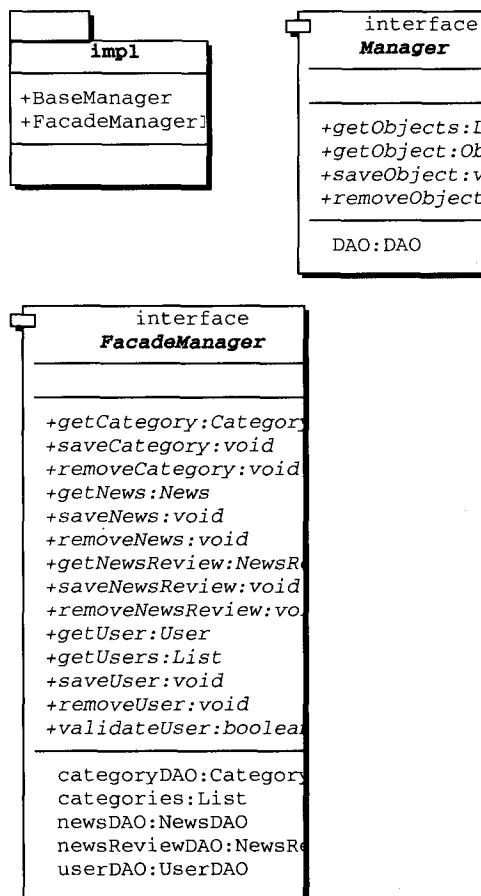


图 9.4 业务逻辑组件的接口类图

下面给出 **FacadeManager** 接口的源代码，该接口是对 DAO 的正面包装：

```

public interface FacadeManager
{
    //增加一个种类
    public void saveCategory(Category category);
    //根据 id 删除种类
    public void removeCategory(String id);
    //获取所有的种类
    public List getCategories();
    //根据主键加载 News 对象
    public News getNews(String id);
    //保存 News 对象
    public void saveNews(News news);
    //删除 News 对象
    public void removeNews(String id);
    //根据 id 加载消息回复对象
    public NewsReview getNewsReview(String id);
    //保存消息回复
    public void saveNewsReview(NewsReview newsReview);
    //删除消息回复
  
```

```
public void removeNewsReview(String id);
//根据用户名加载 User 对象
public User getUser(String username);
//查找用户
public List getUsers(User user);
//保存用户
public void saveUser(User user) throws Exception;
//删除用户
public void removeUser(String username);
//验证用户是否有效
public boolean validateUser(User user);
}
```

### 9.4.3 业务逻辑组件的实现类

业务逻辑组件以 DAO 组件为基础，必须接收 Spring 容器注入的 DAO 组件，因此必须为业务逻辑组件的实现类提供对应的 setter 方法。

业务逻辑组件的实现类将 DAO 组件接口实例作为属性（面向接口编程），下面就是 FacadeManagerImpl 类的源代码：

```
public class FacadeManagerImpl extends BaseManager implements
CategoryManager
{
    //将 CategoryDAO 作为成员属性
    private CategoryDAO categoryDAO;
    //将 NewsDAO 作为成员属性
    private NewsDAO newsDAO;
    //将 NewsReviewDAO 作为成员属性
    private NewsReviewDAO newsReviewDAO;
    //将 UserDao 作为成员属性
    private UserDao userDao;
    //提供依赖注入 CategoryDAO 所需的 setter 方法
    public void setCategoryDAO(CategoryDAO categoryDAO)
    {
        this.categoryDAO = categoryDAO;
    }
    //依赖注入 NewsDAO 所需的 setter 方法
    public void setNewsDAO(NewsDAO newsDAO)
    {
        this.newsDAO = newsDAO;
    }
    //依赖注入 NewsReviewDAO 所需的 setter 方法
    public void setNewsReviewDAO(NewsReviewDAO newsReviewDAO)
    {
        this.newsReviewDAO = newsReviewDAO;
    }
    //依赖注入 UserDao 必需的 setter 方法
    public void setUserDAO(UserDAO userDao)
    {
        this.userDAO = userDao;
    }
    //根据 id 加载消息分类
    public Category getCategory(String id)
    {
        return categoryDAO.getCategory(Long.valueOf(id));
    }
}
```

```
}

//增加消息分类
public void saveCategory(Category category)
{
    categoryDAO.saveCategory(category);
}
//删除消息分类
public void removeCategory(String id)
{
    categoryDAO.removeCategory(Long.valueOf(id));
}
//查询全部的消息分类
public List getCategories()
{
    return categoryDAO.getCategories();
}
//根据主键加载消息
public News getNews(String id)
{
    return newsDAO.getNews(Long.valueOf(id));
}
//增加消息
public void saveNews(News news)
{
    newsDAO.saveNews(news);
}
//删除消息
public void removeNews(String id)
{
    newsDAO.removeNews(Long.valueOf(id));
}
//获取消息回复
public NewsReview getNewsReview(String id)
{
    return newsReviewDAO.getNewsReview(Long.valueOf(id));
}
//增加消息回复
public void saveNewsReview(NewsReview newsReview)
{
    newsReviewDAO.saveNewsReview(newsReview);
}
//删除消息回复
public void removeNewsReview(String id)
{
    newsReviewDAO.removeNewsReview(Long.valueOf(id));
}
//根据用户名查找用户
public User getUser(String username) {
    return userDao.getUser(username);
}
//获取用户列表
public List getUsers(User user)
{
    return userDao.getUsers(user);
}
//增加用户
public void saveUser(User user) throws Exception
{
    try
```

```
{  
    userDAO.saveUser(user);  
}  
catch (DataIntegrityViolationException e)  
{  
    throw new Exception("User '" + user.getUsername()  
        + "' already exists!");  
}  
}  
//删除用户  
public void removeUser(String username)  
{  
    userDAO.removeUser(username);  
}  
//验证用户的方法，用户名存在而且密码正确才会返回 true  
public boolean validateUser(User aUser)  
{  
    User user = getUser(aUser.getUsername());  
    if (user != null && user.getPassword().equals(aUser.getPassword()))  
        return true;  
    else  
        return false;  
}  
}
```

FacadeManagerImpl 继承了 BaseManager，以下是 BaseManager 的源代码：

```
public class BaseManager implements Manager  
{  
    protected DAO dao = null;  
    protected final Log log = LogFactory.getLog(getClass());  
    //依赖注入 DAO 组件必需的 setter 方法  
    public void setDAO(DAO dao)  
    {  
        this.dao = dao;  
    }  
    //根据主键加载持久化对象  
    public Object getObject(Class clazz, Serializable id)  
    {  
        return dao.getObject(clazz, id);  
    }  
    //加载持久化对象的全部实例  
    public List getObjects(Class clazz)  
    {  
        return dao.getObjects(clazz);  
    }  
    //根据主键删除特定对象  
    public void removeObject(Class clazz, Serializable id)  
    {  
        dao removeObject(clazz, id);  
    }  
    //保存对象  
    public void saveObject(Object o)  
    {  
        dao.saveObject(o);  
    }  
}
```

这里的 BaseManager 与 BaseDAO 的作用类似，用于包装工具方法和完成一般的

CRUD 操作。

对于普通的 get 操作如 getNews() 等，Facade 仅仅是调用对应的 DAO 接口：

```
public News getNews(String id)
{
    return newsDAO.getNews(Long.valueOf(id));
}
```

而对于复杂的业务逻辑，可能需要访问多个对象的数据，那么只需在这个方法里调用多个 DAO 接口，将具体实现委派给 DAO 完成。

#### 9.4.4 业务逻辑组件的配置

既然上面业务逻辑组件的 DAO 组件从未被初始化过，那么业务方法如何完成？DAO 组件初始化是由 Spring 的反向控制(Inverse of Control, IoC)，或者称为依赖注入(Dependency Injection, DI)机制完成的。为此我们还需要在 applicationContext.xml 里面配置 FacadeManager 组件。

定义 FacadeManager 组件时必须为其配置所需要的 DAO 组件，配置的示例代码如下：

```
<!-- 配置业务逻辑组件-->
<bean id="facadeManager" class="org.yeeku.service.impl.FacadeManagerImpl">
    <!-- 为其注入多个 DAO 组件-->
    <property name="newsDAO" ref="newsDAO"/>
    <property name="newsReviewDAO" ref="newsReviewDAO"/>
    <property name="categoryDAO" ref="categoryDAO"/>
    <property name="userDAO" ref="userDAO"/>
</bean>
```

请按如下步骤配置业务逻辑层组件。

(1) 在 applicationContext.xml 里加入如下内容：

```
<!-- 定义事务模板类，模板类增加 abstract="true" 属性-->
<bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean">
    <!-- 注入事务管理器-->
    <property name="transactionManager" ref="transactionManager"/>
    <!-- 配置事务属性-->
    <property name="transactionAttributes">
        <props>
            <!-- 所有以 save 开始的方法的事务属性-->
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <!-- 所有以 remove 开始的方法的事务属性-->
            <prop key="remove*">PROPAGATION_REQUIRED</prop>
            <!-- 其他方法的事务属性-->
            <prop key="#">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

在上面加入的内容中定义了一个事务代理模板，其中 “key” 属性用来制定使用该模

板的事务传播属性。例如，key="save\*"，表明该类以 save 开头的方法均采用 PROPAGATION\_REQUIRED 的事务传播属性。

## (2) 在配置文件中增加如下内容：

```
<!-- 配置具体的业务逻辑层组件的事务代理-->
<bean id="manager" parent="txProxyTemplate">
    <!-- 生成业务代理之前，必须使用 target 制定需要生成代理的目标 bean
         目标 bean 采用嵌套 bean 的方式定义-->
    <property name="target">
        <bean class="org.yeeku.service.impl.BaseManager">
            <!-- 定义嵌套 bean 所使用的 DAO 组件-->
            <property name="dao" ref="dao"/>
        </bean>
    </property>
</bean>
```

上面的配置信息表示 BaseManager 继承刚才配置的事务代理模板。并且由容器给 BaseManager 注入“dao”的组件，即 BaseDAOHibernate。而 target 则是 TransactionProxyFactoryBean 需要指定的属性，TransactionProxyFactoryBean 负责为某个 bean 实例生成代理，而代理必须有个目标，target 属性则用于指定目标。

**注意：**此处使用嵌套 bean 配置代理的目标，因为目标 bean 没有事务属性，通过使用嵌套 bean 可避免系统直接访问目标 bean。

当然也可以不使用事务代理模板及嵌套 bean，而是为组件指定单独的事务代理属性，让事务代理的目标引用容器中已经存在的 bean。其源代码如下：

```
<!-- 配置目标 bean 代理对象-->
<bean id="facade"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <!-- 为代理工厂 bean 注入事务管理器-->
    <property name="transactionManager" ref="transactionManager"/>
    <!-- 定义代理的目标 bean
         此处的目标 bean，必须是容器中真实存在的 bean 实例-->
    <property name="target" ref="facadeManager"/>
    <!-- 确定生成事务代理的事务属性-->
    <property name="transactionAttributes">
        <props>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <prop key="remove*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED, readOnly</prop>
        </props>
    </property>
</bean>
```

下面是 applicationContext.xml 文件的源代码，该文件配置了应用的数据源和 SessionFactory 等 bean。而业务逻辑组件也被部署在该文件中：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
```

```

<beans>
    <!-- 配置数据源，使用 DBCP 数据源-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!-- MySQL 数据库的驱动-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <!-- 数据库的 URL-->
        <property name="url" value="jdbc:mysql:///newsboard"/>
        <!-- 指定数据库的用户名-->
        <property name="username" value="root"/>
        <!-- 指定数据库的密码-->
        <property name="password" value="123"/>
        <!-- 指定数据库的最大连接数-->
        <property name="maxActive" value="100"/>
        <!-- 指定数据库的最大空闲连接数-->
        <property name="maxIdle" value="30"/>
        <!-- 指定数据库的最大等待数-->
        <property name="maxWait" value="1000"/>
        <!-- 指定数据库的默认自动提交-->
        <property name="defaultAutoCommit" value="true"/>
        <!-- 指定数据库的连接超时时是否启动删除-->
        <property name="removeAbandoned" value="true"/>
        <!-- 指定数据库的删除数据库连接的超时时间-->
        <property name="removeAbandonedTimeout" value="60"/>
        <property name="logAbandoned" value="true"/>
    </bean>
    <!-- 配置 Hibernate 的 SessionFactory -->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- 依赖注入 SessionFactory 所需的 DataSource-->
        <property name="dataSource" ref="dataSource"/>
        <!-- 加载所有的映射文件-->
        <property name="mappingResources">
            <!-- 下面列出所有的持久化映射文件-->
            <list>
                <value>org/yeeku/model/User.hbm.xml</value>
                <value>org/yeeku/model/News.hbm.xml</value>
                <value>org/yeeku/model/NewsReview.hbm.xml</value>
                <value>org/yeeku/model/Category.hbm.xml</value>
            </list>
        </property>
        <!-- 下面指定 Hibernate 的属性-->
        <property name="hibernateProperties">
            <props>
                <!-- 下面指定 Hibernate 使用的数据库方言-->
                <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDBDialect</prop>
            </props>
        </property>
    </bean>
    <!-- 配置 Hibernate 对应的事务管理器 -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransaction
Manager">
        <!-- 为事务管理器注入 SessionFactory 引用-->
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
    <!-- 配置事务代理模板-->
    <bean id="txProxyTemplate" abstract="true"

```

```
class="org.springframework.transaction.interceptor.  
TransactionProxyFactoryBean">  
!-- 注入事务管理器-->  
<property name="transactionManager" ref="transactionManager"/>  
!-- 确定事务属性-->  
<property name="transactionAttributes">  
    <props>  
        <prop key="save*>PROPAGATION_REQUIRED</prop>  
        <prop key="remove*>PROPAGATION_REQUIRED</prop>  
        <prop key="*>PROPAGATION_REQUIRED,readOnly</prop>  
    </props>  
</property>  
</bean>  
!-- 配置业务逻辑组件的事务代理，使用了 bean 的继承-->  
<bean id="facadeManager" parent="txProxyTemplate">  
    <!-- 配置事务代理时，指定代理的目标，此处的目标是嵌套 bean--&gt;<br/>    <property name="target">  
        <!-- 嵌套 bean 无须使用 id 属性--&gt;<br/>        <bean class="org.yeeku.service.impl.FacadeManagerImpl">  
            <!-- 为业务逻辑组件注入 DAO 组件--&gt;<br/>            <property name="newsDAO" ref="newsDAO"/>  
            <property name="newsReviewDAO" ref="newsReviewDAO"/>  
            <property name="categoryDAO" ref="categoryDAO"/>  
            <property name="userDAO" ref="userDAO"/>  
        </bean>  
    </property>  
</bean>  
</beans>
```

在上面的配置文件中，采用继承业务逻辑组件的事务代理，将原有的业务逻辑组件作为嵌套 bean 配置，避免了直接调用没有事务特性的业务逻辑组件。

至此，我们的系统已经实现了所有的后台业务逻辑，并且向外提供了统一的 Facade 接口，前台 Web 层仅仅依赖这个 Facade 接口。这样 Web 层与后台业务层的耦合已经非常松散，系统可以在不同的 Web 框架中方便切换，即使将整个 Web 层替换掉也非常容易。

## 9.5 Web 层设计

在架构综述里面提到，系统的 Web 层采用的是经典的 J2EE Web MVC 框架 Struts，其表现层也大量使用 Struts 的标签库，关于 Struts 标签库的详细用法，请参考第 3 章的介绍。

### 9.5.1 Action 的实现

Struts 的 Action 实现非常简单，通过继承 Struts 的 Action 基类重写 execute 方法，并在该方法里调用业务逻辑组件的业务方法。在这里，可以发现所有的 Action 有个共同之处——都需要调用业务逻辑组件。

在 Spring 与 Struts 的整合策略里介绍过，业务逻辑组件的实现也不是由 Action 自己控制的，而是接受 Spring 容器的依赖注入。因此必须为 Action 提供对应的 setter 方法，而且每个 Action 都必须为其注入业务逻辑组件，因此可写成一个 Action 基类，让所有的

Action 都从该基类派生。

下面是 Action 基类的源代码：

```
//BaseAction, 作为其他 Action 的父类
public class BaseAction extends Action
{
    // FacadeManager 属性, 面向接口编程
    protected FacadeManager mgr;
    //依赖注入业务逻辑组件必需的 setter 方法
    public void setMgr(FacadeManager mgr)
    {
        this.mgr = mgr;
    }
}
```

注意：在 BaseAction 中，故意将 mgr 属性设置成 protected 的访问权限，目的是为了让其子类可以直接访问该属性，从而提供更简单的访问方式。

而其他的 Action 则简单地从 BaseAction 派生出来，派生出来的 Action 具有一个属性：mgr，该属性就是业务逻辑组件的引用。

下面以几个 Action 作为代表，分别介绍 Action 的实现。

下面的 AddReviewAction 用于拦截用户提交消息回复时的请求，其代码如下：

```
//业务控制器, 以 BaseAction 作为基础
public class AddReviewAction extends BaseAction
{
    /   /必须重写该核心方法, 该方法 actionForm 将表单的请求参数封装成值对象
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //解析 ActionForm, 用于获取请求参数
        DynaValidatorForm addForm = (DynaValidatorForm)form;
        //获取请求参数
        String content = (String)addForm.get("content");
        String newsId = (String)addForm.get("newsId");
        try
        {
            //调用 mgr 的业务方法加载 News 对象
            News news = mgr.getNews(newsId);
            //获取 session 中的 user 值
            String username = (String)request.getSession(true).
                getAttribute (AppConstants.LOGIN_USER);
            //调用 mgr 的业务方法加载 User 对象
            User poster = mgr.getUser(username);
            //创建消息回复实例
            NewsReview newsReview = new NewsReview();
            //设置消息回复的属性
            newsReview.setNews(news);
            newsReview.setPoster(poster);
            newsReview.setContent(content);
            newsReview.setPostDate(new Date());
            newsReview.setLastModifyDate(new Date());
            //持久化消息回复
            mgr.saveNewsReview(newsReview);
        }
    }
}
```

```

        //捕捉异常
        catch (Exception e)
        {
            //出现异常就跳转到 failure
            request.setAttribute("newsId" , newsId);
            return mapping.findForward("failure");
        }
        //执行成功，跳转到 success
        request.setAttribute("newsId" , newsId);
        return mapping.findForward("success");
    }
}

```

下面的 LoadReviewsByNews 用于拦截用户查看消息细节的 Action，该 Action 根据用户请求的 id 加载该消息的回复及消息本身：

```

public class LoadReviewsByNews extends BaseAction
{
    //必须重写该核心方法，该方法 actionForm 将表单的请求参数封装成值对象
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        //希望加载的消息 id
        String newsId = null;
        //考虑到请求有可能从不同地方转发过来，request 的参数和属性
        //分别获取 newsId 属性
        if(request.getAttribute("newsId") == null)
        {
            newsId = request.getParameter("newsId");
        }
        else
        {
            newsId = (String)request.getAttribute("newsId");
        }
        //获取 News 实例
        News news = mgr.getNews(newsId);
        //将 News 实例设置成 Request 属性后转发
        request.setAttribute("news" , news);
        //同时将消息的回复也设置成 Request 属性后转发
        request.setAttribute("reviews", news.getNewsReviews());
        return mapping.findForward("success");
    }
}

```

上面介绍了两种 Action：一种有 ActionForm 的 Action；另一种无需 ActionForm 的 Action。这两种 Action 大同小异，都需要调用 mgr 的业务逻辑方法，区别在于获取请求参数的方式不同。

**注意：**在上面的 Action 中，多次重复访问 PO 对象，而 Action 中通常不允许访问 PO 对象，因为 PO 对象是持久层的组件，应该使用更普通的 JavaBean 作为 VO（值对象）；VO 用于封装业务逻辑组件访问的值，并将这些值传递到 JSP 页面。因为本示例是个较小的示例，所以在 Action 中可直接访问 PO 对象。在第 10 章的示例中，读者将可以看到更加严格的控制。

## 9.5.2 Spring 容器管理 Action

正如前面介绍的，推荐使用 Spring 管理 Struts 的 Action。因为这样可以充分利用 Spring 的 IoC 功能，使 Action 无须关心业务逻辑组件的实现，而由 Spring 负责为 Action 注入业务逻辑组件引用，从而实现更好地解耦。

为了让 Struts 将请求转发到 Spring 容器内的 bean，系统将采用 DelegatingRequestProcessor 的整合策略。因为这种策略无需 Struts 创建 Action 实例，直接由 Spring 容器负责创建 Action 实例，并为其注入依赖关系。使系统更早将请求转发给 Spring 容器控制。

采用这种整合策略，必须在 struts-config.xml 文件中配置 controller 元素，并通过指定 processorClass 属性指定 DelegatingRequestProcessor 处理器。即在配置文件中增加如下代码：

```
<controller inputForward="true"
processorClass="org.springframework.web.struts.DelegatingRequestProcessor"/>
```

经过这个简单的配置，则无须为 struts-config.xml 中的 Action 配置 class 属性，因为 Struts 无须负责创建 Action 实例，由 DelegatingRequestProcessor 直接将请求转发到 Spring 容器内。

下面是 struts-config.xml 文件中 Action 的配置代码：

```
<!-- 添加消息评论 -->
<action path="/addNewsReview"
        name="addNewsReviewForm"
        scope="request"
        validate="true"
        input="input">
    <forward name="failure" path="/loadNewsReviewByNews.do"/>
    <forward name="success" path="/loadNewsReviewByNews.do"/>
</action>
```

在上面的配置代码中，没有为 action 元素确定 class 属性，只有当采用 DelegatingRequestProcessor 替代了默认的 RequestProcessor 后，才允许这样配置。

DelegatingRequestProcessor 转发请求时，请求被转发给 Spring 容器中同名的 bean 处理，因此必须在 Spring 容器中配置同名的 bean。对于上面 Action 的配置，必须在 Spring 容器中配置如下的 bean：

```
<bean name="/addNewsReview" class="org.yeeku.action.AddReviewAction">
    <property name="mgr" ref="facadeManager"/>
</bean>
```

下面是整个应用 struts-config.xml 配置文件的源代码：

```
<?xml version="1.0" encoding="GBK"?>
<!-- Struts 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://struts.apache.org/dtds/struts-config_1_2.dtd">
```

```
<!-- Struts 配置文件的根元素-->
<struts-config>
    <!-- 在 form-beans 元素里配置所有的 ActionForm-->
    <form-beans>
        <!-- 配置登录所使用的 Form-->
        <form-bean name="loginForm" type="org.apache.struts.validator.
DynaValidatorForm">
            <!-- 配置 loginForm 的两个属性-->
            <form-property name="user" type="java.lang.String" />
            <form-property name="pass" type="java.lang.String" />
        </form-bean>
        <!-- 添加消息所用的 Form-->
        <form-bean name="addNewsForm" type="org.apache.struts.validator.
DynaValidatorForm">
            <!-- 配置 addNewsForm 的三属性-->
            <form-property name="title" type="java.lang.String" />
            <form-property name="content" type="java.lang.String" />
            <form-property name="categoryId" type="java.lang.String" />
        </form-bean>
        <!-- 添加消息评论所用的 Form-->
        <form-bean name="addNewsReviewForm" type="org.apache.struts.
validator.DynaValidatorForm">
            <!-- 配置 addNewsReviewForm 的两个属性-->
            <form-property name="content" type="java.lang.String" />
            <form-property name="newsId" type="java.lang.String" />
        </form-bean>
    </form-beans>
    <!-- 配置所有的 Action 映射-->
    <action-mappings>
        <!-- 处理登录 -->
        <action path="/processLogin"
                name="loginForm"
                scope="request"
                validate="true"
                input="input">
            <forward name="input" path="/index.jsp" />
            <forward name="success" path="/listCate.do" />
        </action>
        <!-- 登出系统 -->
        <action path="/logout" scope="request">
            <forward name="success" path="/index.jsp"/>
        </action>
        <!-- 进入主页面 -->
        <action path="/listCate" scope="request">
            <forward name="success" path="/main.jsp"/>
        </action>
        <!-- 根据种类加载所有消息 -->
        <action path="/loadNewsByCategory" scope="request">
            <forward name="failure" path="/listCate.do"/>
            <forward name="success" path="/category_view.jsp"/>
        </action>
        <!-- 添加消息 -->
        <action path="/addNews"
                name="addNewsForm"
                scope="request"
                validate="true"
                input="input">
            <forward name="failure" path="/loadNewsByCategory.do"/>
            <forward name="success" path="/loadNewsByCategory.do"/>
        </action>
    </action-mappings>
</struts-config>
```

```

</action>
<!-- 根据消息 id 加载所有评论 -->
<action path="/loadNewsReviewByNews" scope="request">
    <forward name="success" path="/news_view.jsp"/>
</action>
<!-- 添加消息评论 -->
<action path="/addNewsReview"
        name="addNewsReviewForm"
        scope="request"
        validate="true"
        input="input">
    <forward name="failure" path="/loadNewsReviewByNews.do"/>
    <forward name="success" path="/loadNewsReviewByNews.do"/>
</action>
</action-mappings>
<!-- 配置控制属性-->
<controller inputForward="true"
            processorClass="org.springframework.web.struts.DelegatingRequest
Processor"/>
<!-- 配置国际化的消息资源-->
<message-resources parameter="resource"/>
<!-- 配置数据校验的框架-->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames" value="/WEB-INF/validator-rules.xml,
/WEB-INF/validation.xml"/>
    <set-property property="stopOnFirstError" value="true"/>
</plug-in>
<!-- 配置用于 Spring 整合的插件框架-->
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation" value="/WEB-INF/
daoContext.xml,
/WEB-INF/applicationContext.xml,
/WEB-INF/action-Servlet.xml"/>
</plug-in>
</struts-config>

```

这个配置文件与 Struts 基本的配置文件并没有太多的不同，区别在于该配置文件的 action 元素没有 class 属性，以及使用 DelegatingRequestProcessor 代替了系统默认的 RequestProcessor。

**注意：**虽然 action 元素没有确定 class 属性，但也允许指定 class 属性，只是不会有任何作用。

Spring 对 Action 的配置采用单独的文件配置 action-Servlet.xml，该文件中配置了所有的 Action bean。

因为所有的 Action 都需要为其注入业务逻辑组件，所以此处采用继承简化了 Action bean 的配置。具体的配置文件代码如下：

```

<?xml version="1.0" encoding="GBK"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Spring 配置文件的根元素-->
<beans>

```

```

<!-- 配置 Action 模板，用于被其他 Action 继承-->
<bean id="actionTemplate" abstract="true" singleton="false">
    <property name="mgr" ref="facadeManager"/>
</bean>
<!-- 处理登录-->
<bean name="/processLogin" class="org.yeeku.action.LoginAction"
parent="actionTemplate"/>
<!-- 登出系统-->
<bean name="/logout" class="org.yeeku.action.Logout"/>
<!-- 列出所有的消息分类-->
<bean name="/listCate" class="org.yeeku.action.ListCate" parent=
actionTemplate"/>
<!-- 根据种类列出所有消息-->
<bean name="/loadNewsByCategory" class="org.yeeku.action.LoadNewsBy
Category"
parent="actionTemplate"/>
<!-- 添加消息-->
<bean name="/addNews" class="org.yeeku.action.AddNewsAction" parent=
actionTemplate"/>
<!-- 根据消息查看所有的评论-->
<bean name="/loadNewsReviewByNews" class="org.yeeku.action.LoadReviews
ByNews"
parent="actionTemplate"/>
<!-- 添加消息评论-->
<bean name="/addNewsReview" class="org.yeeku.action.AddReviewAction"
parent="actionTemplate"/>
</beans>

```

至此，已经基本完成了 Struts 与 Spring 的整合。当 ActionServlet 拦截到用户请求时，则调用 DelegatingRequestProcessor，该处理器将请求转发到 Spring 容器中的 bean，由该 bean 负责调用业务逻辑组件处理用户请求，并将处理结果呈现给用户。

### 9.5.3 数据校验的选择

数据校验是表现层必须处理的基本问题。根据第 3 章的介绍，表现层的数据校验分成客户端校验和服务器端校验。不管是客户端校验，还是服务器端校验，Struts 都有很好的支持，完全可以弹出 JavaScript 校验。

此处要提醒读者的是，Struts 的客户端校验有一个弊端。

看如图 9.5 所示的简单登录页面。

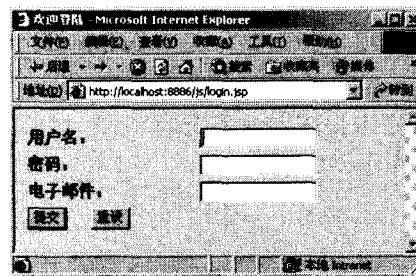


图 9.5 简单的登录页面

在这个简单的登录页面中，假设只需要对页面中三个表单域进行校验，如用户名、密码及电子邮件这三项必填，并且电子邮件为有效的地址，可以采用如下的 JavaScript 代码校验：

```
<script>
function check(form)
{
    var errMsg = "";
    if (form.user.value == null || trim(form.email.value)=="")
    {
        errMsg += "用户名必填" ;
    }
    if (form.pass.value == null || trim(form.pass.value)=="")
    {
        errMsg += "密码必填" ;
    }
    if (form.email.value == null || trim(form. email.value)=="")
    {
        errMsg += "电子邮件必填" ;
    }
    if(trim(form. email.value)!="" && !/^w+([-+.]\w+)*@\w+([-.]?\w+)*\.\w+([-.]?\w+)*$/.test(form.email.value))
    {
        errMsg += "电子邮件格式不对" ;
        return false;
    }
    if(errMsg == null )
    {
        return true;
    }
    return false
}
//用于截取两端的空格
function trim(s)
{
    return s.replace( /\s*/ , " " ).replace( /\s*$/ , " " );
}
</script>
```

这段 JavaScript 代码结构清晰，相当简洁。

通过第 3 章的学习，我们也可以使用 Struts 的验证框架来生成客户端校验，但 Struts 生成的 JavaScript 的校验代码非常多，笔者在此处不能完全列出，但读者可以通过客户端查看网页源代码看到这将近 1000 行的 JavaScript 代码。

仔细检查 Struts 生成的 JavaScript 代码，可以发现这些 JavaScript 代码包含了最小长度校验、最长长度校验及有效范围校验等，而这些与该页面的需求没有丝毫关系。

这正是手写 JavaScript 校验和 Struts 自动生成 JavaScript 校验的区别：Struts 生成的校验会包含更多的代码，即使页面只需要校验一个简单的必填项，Struts 也会生成将近 1000 行的 JavaScript 代码。虽然这些代码不需要程序员手写，但这些代码必须要下载到客户端执行，显然这些无用的代码将加重客户端网络带宽的负担。因此客户端校验依然建议使用手写校验。

**注意：**笔者在这里介绍给读者一个技巧，仔细观察 Struts 生成的 JavaScript 客户端校验代码，就可发现每个页面的 JavaScript 代码只有前面数行不同，其他部分则是完全相同的。没错，这些部分是通用的，我们完全可以将这些通用的部分提取出来，作为通用的 JavaScript 代码，并以单独的 JavaScript 文件保存，当每个页面需要进行校验时，只需导入该通用的 JavaScript 文件即可；而页面则只需加入 Struts 为不同页面生成的不同部分。关于客户端校验，还可以直接采用 ProtoType 校验。

在网络带宽受限的情况下，建议不要采用 Struts 的 JavaScript 校验，并不是意味着可以不使用 Struts 的校验框架。因为服务器端校验依赖于 Struts 的校验要简单得多。

增加校验的详细步骤请参考第 3 章的内容。此处给出本应用的校验规则文件，validation.xml 文件的源代码如下：

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- 校验文件的文件头，包含 DTD 等信息-->
<!DOCTYPE form-validation PUBLIC
        "-//Apache Software Foundation//DTD Commons Validator Rules
Configuration 1.1.3//EN"
        "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<!-- 校验文件的根元素-->
<form-validation>
    <!-- 需要校验的 Form 都放在 formset 元素里-->
    <formset>
        <!-- 需要校验的第一个 form：登录用的 loginForm-->
        <form name="loginForm">
            <!-- 需要校验的 user 域，需满足最小长度，必填两个规则-->
            <field property="user" depends="required,minlength">
                <arg key="loginForm.user" position="0"/>
                <arg name="minlength" key="${var:minlength}" resource="false" position="1"/>
                <var>
                    <var-name>minlength</var-name>
                    <var-value>4</var-value>
                </var>
            </field>
            <!-- 需要校验的 pass 域，需满足最小长度，必填两个规则-->
            <field property="pass" depends="required,minlength">
                <arg key="loginForm.pass" position="0"/>
                <arg name="minlength" key="${var:minlength}" resource="false" position="1"/>
                <var>
                    <var-name>minlength</var-name>
                    <var-value>4</var-value>
                </var>
            </field>
        </form>
        <!-- 需要校验的第二个 form：添加消息用的 addNewsForm -->
        <form name="addNewsForm">
            <!-- 需要校验的 title 域，需满足必填规则-->
            <field property="title" depends="required">
                <arg key="addNewsForm.title" position="0"/>
            </field>
            <!-- 需要校验的 content 域，需满足必填规则-->
            <field property="content" depends="required">
```

```

<arg key="addNewsForm.content" position="0"/>
</field>
<!-- 需要校验的 categoryId 域，需满足必填，整数规则-->
<field property="categoryId" depends="required,integer">
    <arg key="addNewsForm.categoryId" position="0"/>
</field>
</form>
<!-- 需要校验的第三个 form：添加消息评论用的 addNewsReviewForm -->
<form name="addNewsReviewForm">
    <!-- 需要校验的 content 域，需满足必填规则-->
    <field property="content" depends="required">
        <arg key="addNewsReviewForm.content" position="0"/>
    </field>
    <!-- 需要校验的 newsId 域，需满足必填，整数规则-->
    <field property="newsId" depends="required,integer">
        <arg key="addNewsReviewForm.categoryId" position="0"/>
    </field>
</form>
</formset>
</form-validation>

```

推荐的校验策略是：在客户端采用手写的 JavaScript 校验；而服务器端采用 Struts 的校验框架完成。

#### 9.5.4 访问权限的控制

本系统访问权限的控制非常简单，在使用本消息发布系统之前，必须先登录本系统。如果某个未登录的用户企图进入系统使用页面时，则系统将请求转到登录页面。

权限控制的最传统做法是：在 Action 里手动控制，在每次调用业务逻辑方法之前，首先判断用户是否有足够的权限。在本系统中就是判断用户是否登录，但这种方法相当烦琐，而且需要重复判断用户是否登录，这与 DRY（不要书写重复的代码）规则相违背。

但权限控制有更好的选择：利用 Filter 过滤请求，或者使用 AOP 框架拦截请求。笔者在本章采用 Filter 过滤请求完成权限检查，在第 10 章将采用 AOP 框架来完成更复杂的权限检查。

相对而言，利用 AOP 框架的权限检查支持更细的粒度，灵活性更好。

下面是控制权限检查的 Filter 源代码：

```

// 将请求转换成 HttpServletRequest
HttpServletRequest httpServletRequest = (HttpServletRequest) request;
// 将响应转换成 HttpServletResponse
HttpServletResponse httpServletResponse = (HttpServletResponse) response;
// 获取客户端的请求的地址
String requesturi = httpServletRequest.getRequestURI();
// 通过检查 session 中的变量，过滤请求
HttpSession session = httpServletRequest.getSession();
Object currentUser = session.getAttribute(AppConstants.LOGIN_USER);
// 当前会话用户为空而且不是请求登录，则退出登录，欢迎页面和根目录则退回到应用的根目录
if (currentUser == null
    && !requesturi.endsWith("/processLogin.do")
    && !requesturi.endsWith("/logout.do")
    && !requesturi.endsWith("/index.jsp"))

```

```

    && !requestURI.endsWith(HttpServletRequest.getContextPath()
        + "/"))
    {
        httpResponse.sendRedirect(HttpServletRequest.
            getContextPath() + "/");
        return;
    }
    //否则，允许继续处理请求
    chain.doFilter(request, response);
}

```

Filter 本身不能处理用户请求，也不能生成响应，它是个典型的链式处理，拦截用户请求。增加 Filter 的额外处理后，依然将请求转发给 Servlet，由 Servlet 生成客户端响应。

在该用户请求中，如果请求地址不是请求登录、退出登录、欢迎页面或根目录，并且当前用户没有登录时，则退回到应用的根目录。

## 9.5.5 解决中文编码问题

来自我国的请求，基本都以 GBK 方式编码，而 Struts 的 ActionServlet 默认以 ISO 8859-1 方式解码。在这种情况下，不可避免地会产生乱码问题，为避免这种乱码问题，可以有以下两个做法：

- 扩展 ActionServlet。
- 利用 Filter 处理编码方式。

两种方式的处理原理相似，都是通过设置 HttpServletRequest 的解码方式。关于扩展 ActionServlet 在第 3 章已经介绍过了，此处介绍第二种方式——利用 Filter 处理编码。

类似于扩展 ActionServlet，利用 Filter 也需要在 doFilter 的方法中增加设置 HttpServletRequest 的解码方式即可。

下面是本应用所使用的 Filter 的源代码：

```

public class UserLoginFilter implements Filter
{
    //用于本系统所使用的编码方式
    protected String encoding = null;
    //保存 Filter 的配置对象
    protected FilterConfig filterConfig = null;
    //是否忽略配置的编码方式
    protected boolean ignore = false;
    protected String forwardPath = null;
    //销毁 Filter 时调用该方法
    public void destroy()
    {
        this.encoding = null;
        this.filterConfig = null;
    }
    //处理用户请求
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException
    {
        // 设置编码方式，web.xml 里面有 filter 参数的初始化设置
        if (ignore || (request.getCharacterEncoding() == null))

```

```

    {
        String encoding = selectEncoding(request);
        //设置编码方式
        if (encoding != null)
            request.setCharacterEncoding(encoding);
    }
    //将请求转换成 HttpServletRequest
    HttpServletRequest httpServletRequest = (HttpServletRequest) request;
    //将响应转换成 HttpServletResponse
    HttpServletResponse httpServletResponse =
        (HttpServletResponse) response;
    //获取客户端的请求地址
    String requesturi = httpServletRequest.getRequestURI();
    //通过检查 session 中的变量，过滤请求
    HttpSession session = httpServletRequest.getSession();
    Object currentUser = session.getAttribute(AppConstants.LOGIN_USER);
    //当前会话用户为空而且不是请求登录，则退出登录，欢迎页面和根目录则退回到应用的根目录
    if (currentUser == null
        && !requesturi.endsWith("/processLogin.do")
        && !requesturi.endsWith("/logout.do")
        && !requesturi.endsWith("/index.jsp")
        && !requesturi.endsWith(httpServletRequest.getContextPath()
            + "/")) {
        httpServletResponse.sendRedirect(httpServletRequest
            .getContextPath() + "/");
        return;
    }
    chain.doFilter(request, response);
}
//初始化该 Filter 时调用该方法
public void init(FilterConfig filterConfig) throws ServletException
{
    //初始化 FilterConfig 对象
    this.filterConfig = filterConfig;
    //通过 filterConfig 获取请求参数: encoding
    this.encoding = filterConfig.getInitParameter("encoding");
    //通过 filterConfig 获取请求参数: forwardpath
    this.forwardPath = filterConfig.getInitParameter("forwardpath");
    //通过 filterConfig 获取请求参数: ignore
    String value = filterConfig.getInitParameter("ignore");
    //处理 ignore 参数的值
    if (value == null)
        this.ignore = true;
    else if (value.equalsIgnoreCase("true"))
        this.ignore = true;
    else if (value.equalsIgnoreCase("yes"))
        this.ignore = true;
    else
        this.ignore = false;
}
protected String selectEncoding(ServletRequest request) {
    return (this.encoding);
}
}

```

该 Filter 用于拦截整个应用的全部请求，但必须为其配置对应的参数，关于该 Filter 的配置代码如下：

```

<filter>
    <!-- 指定 Filter 的名字-->
    <filter-name>Login Filter</filter-name>
    <!-- 指定 Filter 的实现类-->
    <filter-class>
        org.yeeku.webapp.filter.UserLoginFilter
    </filter-class>
    <!-- 指定 Filter 的第一个初始化参数: encoding -->
    <init-param>
        <param-name>encoding</param-name>
        <param-value>GBK</param-value>
    </init-param>
    <!-- 指定 Filter 的第二个初始化参数: ignore -->
    <init-param>
        <param-name>ignore</param-name>
        <param-value>false</param-value>
    </init-param>
    <!-- 指定 Filter 的第三个初始化参数: forwardpath -->
    <init-param>
        <param-name>forwardpath</param-name>
        <param-value>index.jsp</param-value>
    </init-param>
</filter>
<!-- 指定 Filter 负责拦截的 URL: 负责拦截所有请求 -->
<filter-mapping>
    <filter-name>Login Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

## 9.5.6 JSP 页面输出

JSP 的功能相当简单，除了完成数据的收集，就是完成简单的数据显示。此处的数据显示主要用于从 HttpServletRequest 中获取 Attribute，然后将其显示在 JSP 页面中，也包括对集合对象的显示。

JSP 页面的显示主要依赖于 Struts 的标签库，下面以几个简单示例，来介绍 JSP 如何利用 Struts 输出集合内容。

```

<logic:present name="categories" scope="request">
    <logic:iterate id="item" name="categories" indexId="index" scope="request">
        <tr>
            <td width="10%" align="center">
                <input type="radio" name="categoryId" value='<bean:write
name="item" property="id"/>'>
            </td>
            <td width="30%" align="center">
                <bean:write name="item" property="name"/>
            </td>
        </tr>
    </logic:iterate>
</logic:present>

```

该代码用于输出所有的消息分类，从 Action 转发到该页面时，则在 HttpServletRequest 中封装了 categories 的属性，该属性是个集合对象，集合里每个元素都是 Category 对象。

页面首先通过 logic:present 标签判断 categories 属性是否存在，如果该属性不存在，

则不用输出；如果该属性存在，则使用 logic:iterate 迭代器标签，循环输出 categories 集合中每个元素。该迭代器每次迭代时将集合中的元素放在 page 范围内，并在迭代器标签的标签体内使用 bean:write 标签，输出集合属性中每个元素的属性值。

图 9.6 显示了该代码的效果。

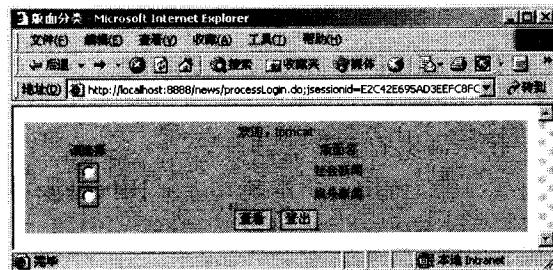


图 9.6 迭代器标签的输出效果

再看下面的页面输出代码：

```
<logic:messagesPresent>
    <div class="title">
        <bean:message key="errors.header"/>
        <ul>
            <html:messages id="error">
                <li><bean:write name="error"/></li>
            </html:messages>
        </ul>
    </div>
</logic:messagesPresent>
```

该代码片段主要用于输出校验信息，页面首先通过 logic:messagesPresent 标签来判断是否包含出错提示信息，如果包含出错提示，则输出该出错提示；如果不存在，则无须输出任何内容。

当出错提示存在时，首先使用<bean:message key="errors.header"/>输出国际化信息，然后使用 html:messages 标签，对系统包含的出错提示逐行输出。

图 9.7 显示了出错提示的显示效果。



图 9.7 服务器端数据校验的提示信息

## 9.6 系统最后的思考

总结前文，采用 Spring 作为 bean 管理容器的最大优势是：使层与层之间的耦合非常松散。

Spring 将各个层整合在一起，但在这种整合方式管理下，在代码里丝毫感觉不到 Spring 的存在。这得益于 Spring 的依赖注入机制，我们可以用普通的 JavaBean 写出符合 J2EE 规范的应用，而无须使用昂贵的 EJB 容器。

为达到同样的划分层次目的，EJB 2.0 和 Spring + Hibernate 分别是怎样做到的？这里以一个例子来进行说明。

### 9.6.1 传统 EJB 架构的实现

从 DAO 层来看，EJB 2.0 的做法就是编写用户的 EntityEJB，需要编写 EJBObject 接口及 EJBHome 接口来实现 EntityBean 的抽象类。因为禁止直接暴露远程接口，我们还需要额外编写一个普通的 JavaBean，用于在 Web 层和业务逻辑层之间传输数据。

本示例使用 CMP 的 Entity EJB，其 EJB 组件的源代码如下：

User 接口的源代码如下：

```
//实体 EJB 的 local 接口
package entityEJB;
import javax.ejb.EJBLocalObject;
//实体 EJB 的 local 接口必须继承 EJBLocalObject 接口
public interface User extends EJBLocalObject
{
    //id 属性的 setter 和 getter 方法
    public Integer getId();
    public void setId(Integer id);
    //name 属性的 setter 和 getter 方法
    public String getName();
    public void setName(String name);
    //password 属性的 setter 和 getter 方法
    public String getPassword();
    public void setPassword(String password);
}
```

UserHome 接口的源代码如下：

```
//实体 EJB 的 local home 接口
//实体 EJB 的 local home 接口必须继承 EJBLocalHome 接口
public interface UserHome extends EJBLocalHome
{
    //创建一条新记录
    public User create (Integer id, String name, String password)
        throws CreateException;
    //根据主键查找记录
    public User findByPrimaryKey (Integer id)
        throws FinderException;
```

```

    //根据用户名查找记录
    public User findByName(String name)
}

```

UserBean 是实现 EntityEJB 的抽象类，其 UserBean 抽象类的源代码如下：

```

public abstract class UserBean implements EntityBean
{
    //实体 EJB 必需的 Context 成员属性
    private EntityContext context;
    //容器管理持久化属性的访问方法
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract String getName();
    public abstract void setName(String name);
    public abstract String getPassword();
    public abstract void setPassword(String password);
    //ejbCreate 方法，对于 home 接口里的 create 方法。
    //该方法是 create 方法的逻辑实现
    public Integer ejbCreate (Integer id, String name, String password)
        throws CreateException
    {
        //create 方法用于初始化属性，并将记录插入数据库
        //由于是容器管理关系字段，因此无需手工插入
        setId(id);
        setName(name);
        setPassword(password);
        return id;
    }
    //下面是 Entity EJB 的生命周期方法
    //用于插入数据库记录时调用，用于完成关联字段的初始化
    public void ejbPostCreate(Integer id, String name, String password)
        throws CreateException
    {
    }
    public void setEntityContext(EntityContext ctx)
    {
        context = ctx;
    }
    public void unsetEntityContext()
    {
        context = null;
    }
    //用于删除 EJB 实例，以及删除对应数据库记录时被调用的方法
    public void ejbRemove()
    {
    }
    //EJB 从数据库加载数据的方法
    public void ejbLoad()
    {
    }
    //EJB 将数据持久化到数据库的方法
    public void ejbStore()
    {
    }
    //EJB 钝化时被调用的方法
    public void ejbPassivate()
    {
    }
}

```

```
    }
    //EJB 激活时被调用的方法
    public void ejbActivate()
    {
    }
}
```

其中，用于传输数据的普通 JavaBean，既是 DTO，也是 VO 对象，该 JavaBean 的源代码如下：

```
//用于传值的 JavaBean
public class User implements Serializable
{
    //id 属性
    private Integer id;
    //用户名属性
    private String username;
    //密码属性
    private String password;
    //无参数的构造器
    public User()
    {
    }
    //password 的 setter 和 getter 方法
    public String getPassword()
    {
        return password;
    }
    public void setPassword(String password)
    {
        this.password = password;
    }
    //username 的 getter 和 setter 方法
    public String getUsername()
    {
        return username;
    }
    public void setUsername(String username)
    {
        this.username = username;
    }
    //id 属性的 getter 和 setter 方法
    public Integer getId()
    {
        return id;
    }
    public void setId(Integer id)
    {
        this.id = id;
    }
}
```

## 9.6.2 EJB 架构与轻量级架构的对比

采用 Spring + Hibernate 的架构仅需要一个普通的 JavaBean，一个简单的 DAO 接口

以及同样简单的 DAO 接口实现，没有硬性规定要实现特定接口。这样不仅编码工作量大幅减少，而且单元测试也非常方便。

反观 EntityEJB，其代码里充斥着 EJB 架构的接口，必须重写多个方法，即使这些方法只是空方法或抽象方法。

在业务逻辑层，EJB 要用 SessionEJB 对 EntityEJB 做正面包装，需要 EJBObject 接口、EJBHome 接口和一个 SessionBean。

### Facade 接口：

```
//业务逻辑接口，用于暴露业务逻辑方法
public interface Facade extends EJBObject
{
    //该接口里暴露的是业务逻辑方法，可以被远程调用
    public boolean validateUser(String name, String password)
        throws RemoteException, NamingException;
}
```

### FacadeHome 接口：

```
//主接口，用于创建 EJB 业务逻辑接口
public interface FacadeHome extends EJBHome
{
    //create 方法用于返回业务逻辑接口
    public Facade create() throws RemoteException, CreateException;
}
```

### FacadeBean 类：

```
public class FacadeBean implements SessionBean
{
    //Session Bean 的私有成员属性
    private SessionContext sctx;
    public void setSessionContext(SessionContext c)
    {
        this.sctx = c;
    }
    //下面是 Session EJB 的系列生命周期方法
    public void ejbCreate() throws CreateException
    {
    }
    public void ejbActivate()
    {
    }
    public void ejbPassivate()
    {
    }
    public void ejbRemove()
    {
    }
    //实现业务逻辑方法，该方法是 remote 接口里 validateUser 方法的逻辑实现
    //因为该方法并不在 Remote 接口或其子接口里定义，因此无须抛出 RemoteException
    public boolean validateUser(String name, String password)
        throws NamingException
    {
        //查找 User EJB 的 JNDI
        UserHome uh = (UserHome)sctx.lookup("UserEJB");
```

```
try
{
    //通过 User EJB 的数据库访问操作完成登录校验
    User user = uh.findByName(name);
    if(user.getPassword().equals("password"))
        return true;
    else
        return false;
}
catch(FinderException fe)
{
    return false;
}
}
```

对比这两种架构可以发现, EntityEJB 就是一种重量级且功能非常强大的 DAO 组件。这种 DAO 组件可以直接访问数据库, 支持事务, 并提供远程服务等。

同 EntityEJB 情况相似, 其代码里也充斥着 EJB 架构的接口。而 Spring + Hibernate 需要的仅是 Facade 接口和 Facade 接口的实现。

将 Spring + Hibernate 与 EJB 相比较, 我们可以看出, 除了编码量减少, 代码侵入性低以外, 还少了一个与 Home 接口对应的寻找资源的接口。因为 Spring 已经将所有组件都集中置于 ApplicationContext 的管理之下, 代码侵入性低的一个直接结果就是, 当系统架构改变时, 能够复用代码。

在采用 EJB 架构的情形下, 如果出于某些原因需要改变系统架构, 不再使用 EJB, 由于原有代码里充斥着 EJB 架构的相关代码, 因此降低了原有代码的重用性, 可能只有用于传输数据的普通 JavaBean 可以复用。当然, EJB 这种重量级的组件技术, 自然有其不可比拟的优势: 比如远程访问、跨资源的事务及性能与稳定性要求比较高的超大型项目等, 那么 EJB 将是很好的选择, 而且这样大型的项目架构也不会轻易改变。如大型移动通信网络点, 有些需要支撑 50 万以上用户, 普通的 Web 容器性能不足以支撑, 通常就要采用 EJB 架构跟 EJB 容器。但这些服务并不是总是必需的, 特别是对于一些中小型的应用。

Spring 的情形就很乐观, 其组件就是普通的 JavaBean, 代码里几乎找不到 Spring 的痕迹, 各层之间耦合松散, 因此改变架构也能重用代码。Spring 实用主义的设计, 确实帮助了广大开发者, 也为中小型的项目提供了更多灵活的选择。

## 本章小结

本章演示了如何开发一个标准的 J2EE 应用, 并从系统架构的设计到具体的实现作出了详细的讲解。

本节的实例具体应用了前面介绍的三个框架: Hibernate, Spring, Struts, 并将这些框架有机地结合在一起, 组成一个完整的应用, 使读者能熟练使用这些框架, 并将它们结合使用, 去开发一些大型的应用。

# 第 10 章

## 完整应用：简单工作流系统

### 本章要点

- 『 J2EE 的系统结构
- 『 面向对象分析与设计
- 『 基于 HibernateDaoSupport 的 DAO 实现
- 『 实现业务逻辑层
- 『 Quartz 任务调度框架
- 『 设计 Web 层

该系统是个简单的工作流系统，通过该系统，可以实现公司日常工作的自动化，从而提高整个公司的办公效率。

系统采用最流行的 J2EE 架构：Struts + Spring + Hibernate。该系统结构成熟，性能良好，运行稳定。Spring 的 IoC 容器负责管理业务逻辑组件、持久层组件及控制层组件，充分利用依赖注入的优势，进一步增强系统的解耦，提高应用的可扩展性，降低系统重构的成本，其工作流后台的作业调度使用 Quartz 框架完成。

## 10.1 项目背景及系统结构

本章将以一个简单的工作流系统为例，为读者示范如何开发完整的 J2EE 应用。该系统包含一个公司日常的事务：日常考勤、工资结算及签核申请等。

### 10.1.1 应用背景

所谓工作流，就是企业或组织日常工作的固定流程。比如签核流程及外贸企业的报关流程等。工作流是 J2EE 应用的主要应用方向之一，在计算机信息系统尚未形成主流时，其工作流都是由人工完成的，但人工完成存在诸多不利，如工作效率低（一个节点的延迟将导致整个工作流的停滞）、信息传递响应速度慢，以及纸张通信资源浪费等。

在 20 世纪 80 年代中，人们终于找到了缓解这些弊病的办法，就是依赖网络的工作流技术。

计算机工作流是完全自动化的流程，避免了使用各种申请文件的人工传送，而申请文件都是以电子文件形式，避免了传送延迟，提高了效率等。

结合了网络技术的工作流功能更加强大，一个全球性的企业信息化平台，借助于 Email、即时通信工具以及自定义工作流，完全可以将各地区的组织有机地组织在一起。各区域的通信，各种流程申请等完全是即时响应，避免等待。

本示例应用的签核系统，完成了考勤改变申请的送签，以及对申请的签核。这是一种简单的工作流，同样可以提高企业的生产效率。另外，本应用额外的打卡系统、自动工资结算等，也可以在一定程度上提高企业的生产效率。

### 10.1.2 系统功能介绍

系统的用户分为两个角色：普通员工和经理。

普通员工的功能包括：系统将自动完成员工每天上下班的考勤记录，包括迟到、早退、旷工等；员工也可以查看本人最近三天的考勤情况，如果发现考勤与实际不符（例如出差，或者病假等），则可提出申请，该申请将由系统自动转发给员工经理，如果经理通过核准，则此申请自动生效，系统将考勤改为实际的情况；此外，员工还可查看自己的工资记录。

经理功能除了包括普通员工的功能外，还有签核员工申请的功能，以及对新增员工的查看和查看员工的上月工资等功能。但经理的考勤不能提出申请，实际的项目中，经理会有更上一级的管理者，因此经理也可以对考勤异动提出申请。实际的项目中采用树形组织结构图来组织部门，每个员工属于部门。

当然，这个系统与实际应用还有一些距离，此示例仅介绍如何开发 J2EE 应用，而不是介绍如何开发工作流系统。

### 10.1.3 相关技术介绍

本系统主要涉及三个开源框架：Struts、Spring 和 Hibernate，同时还使用了 JSP 作为表现层技术。本系统将这四种技术有机地结合在一起，从而构建出一个健壮的 J2EE 应用。

#### 1. 传统表现层技术：JSP

本系统使用 JSP 作为表现层，负责收集用户请求数据，以及业务数据的表示。

JSP 是最传统，也最有效的表现层技术。本系统的 JSP 页面是单纯的表现层，所有的 JSP 页面禁止使用 Java 脚本。结合 Struts 的表现层标签，JSP 可完成全部的表现层功能——数据收集、数据表示和客户端数据校验。

#### 2. MVC 框架

本系统使用 Struts 作为 MVC 框架。Struts 的稳定、简单及易用等，为其赢得了广泛的市场支持。Struts 的 Action 拦截用户所有的请求，包括系统的超链接和表单提交等，都由 Struts 控制请求的处理和转发。

通过 Action 拦截所有请求有个好处：将所有的 JSP 页面放入 WEB-INF/路径下，可以避免用户直接访问 JSP 页面，从而提高系统的安全性。

#### 3. Spring 的应用

Spring 框架是系统的核心部分，Spring 提供的 IoC 容器是业务逻辑组件和 DAO 组件的工厂，它负责生成并管理这些实例。

借助于 Spring 的依赖注入，各组件以松耦合的方式组合在一起，组件与组件之间的依赖正是通过 Spring 的依赖注入管理。其 Service 组件和 DAO 对象都采用面向接口编程的方式，从而降低了系统异构的代价。

其中，DAO 对象的创建使用了 Spring 的 HibernateDaoSupport 作为基类，继承 HibernateDaoSupport 的 DAO 对象实现更加简单，程序开发者无须管理 Hibernate 的 SessionFactory 及 Session 等对象，通过 Spring 提供的 HibernateTemplate 即可完成数据库操作。

Spring 的 AOP 框架为系统的权限提供了支持。因为应用中控制器没有权限检查逻辑，每个控制器都需要重复检查调用者是否有足够的访问权限，而采用 AOP 框架可以避免重复检查，正是 DRY（不要重复你自己）原则的应用。

事务采用 Spring 的声明式事务框架。通过声明式事务，无须将事务策略以硬编码的方式与代码耦合在一起，而是放在配置文件中声明，使业务逻辑组件可以更加专注于业务的实现，从而简化开发。同时，声明式事务降低了不同事务策略的切换代价。

该系统的工资自动结算和自动考勤等都采用 Quartz 框架，该框架使用 Cron 表达式触发来调度作业，从而完成任务自动化。

本系统的测试也基于 Spring 的测试框架完成。

#### 4. Hibernate 的应用

Hibernate 作为 O/R mapping 框架使用，其 O/R mapping 功能简化了数据库的访问，并在 JDBC 层上提供了更好的封装。以面向对象的方式操作数据库，更加符合面向对象程序设计的思路。

Hibernate 以优雅及灵活的方法操作数据库，无需开发者编写烦琐的 SQL 语句，执行冗长的多表查询，而通过对对象与对象之间的关联来操作数据库，为底层的 DAO 对象的实现提供了支持。

### 10.1.4 系统结构

本系统采用严格的 J2EE 应用结构，主要有以下几个分层。

- 表现层：由 JSP 页面组成。
- MVC 层：使用 MVC 框架技术。
- 业务逻辑层：主要由 Spring IoC 容器管理的业务逻辑组件组成。
- DAO 层：由 7 个 DAO 组件组成。
- Hibernate 持久层：由 7 个 PO 组成，并在 Hibernate Session 管理下，完成数据库访问。
- 数据库服务层：使用 MySQL 数据库存储持久化数据。

系统的具体分层如图 10.1 所示。

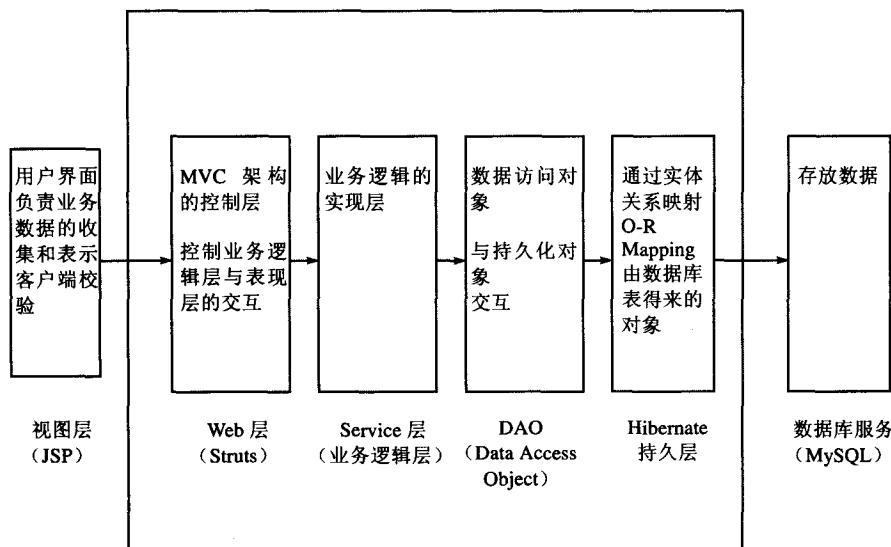


图 10.1 系统结构图

在图 10.1 黑色大方框内的 MVC 控制层、Service 层及 DAO 组件层的组件，都由 Spring IoC 容器负责生成，并管理组件的实例。

## 10.1.5 系统的功能模块

本系统可以大致分为两个模块：经理模块和员工模块，其主要业务逻辑都通过 EmpManager 和 MgrManager 两个业务组件实现，因此可以使用这两个业务逻辑组件来 Facade DAO 组件。

**注意：**通常建议按细粒度的模块来设计 Service 组件，使每个 Service 组件 Facade 多个 DAO 组件；每个 Service 组件包含模块内的业务方法。

本系统主要有如下 7 个 DAO 对象。

- ApplicationDao：提供对 app\_table 表的基本操作。
- AttendDao：提供对 attend\_table 表的基本操作。
- AttendTypeDao：提供对 type\_table 表的基本操作。
- CheckBackDao：提供对 check\_table 表的基本操作。
- EmployeeDao：提供对 emp\_table 表的基本操作。
- ManagerDao：提供对 mgr\_table 表的基本操作。
- PaymentDao：提供对 pay\_table 表的基本操作。

## 10.2 Hibernate 持久层

通过使用 Hibernate 持久层，可以避免使用传统的 JDBC 操作数据库，从而更好地使用面向对象的方式来操作数据库。保证了整个软件开发过程以面向对象的方式进行，即面向对象分析、设计及编程。

### 10.2.1 设计持久化对象（PO）

面向对象分析，是指根据系统需求提取应用中的对象，将这些对象抽象成类，再抽取出需要持久化保存的类，这些需要持久化保存的类就是持久化对象（PO）。该系统并没有预先设计数据库，而是完全从面向对象分析开始，设计了 7 个持久化类。

本系统一共设计了如下 7 个 PO。

- Application：对应普通员工的考勤提出申请，包括申请理由、是否被批复及申请改变的类型等属性。
- Attend：对应每天的考勤，包含考勤时间、考勤员工、是否上班及考勤类别等信息。
- AttendType：对应考勤的类，包含考勤的名称，如迟到、早退等名称。
- CheckBack：对应批复，包含对应的申请，是否通过申请，由哪个经理完成批复等属性。

- **Employee**: 对应系统的员工信息，包含员工的用户名、密码、工资以及对应的经理等属性。
- **Manager**: 对应系统的经理信息，仅包含经理管理的部门名。实际上，**Manager** 继承 **Employee** 类，因此该类同样包含 **Employee** 的所有属性。
- **Payment**: 对应每月所发的薪水信息，包含发薪的月份、领薪的员工及薪资数等信息。

客观世界中的对象不是孤立存在的，以上 7 个 PO 也不是孤立存在的，它们之间存在复杂的关联关系。分析关联关系也是面向对象分析的必要步骤，这 7 个 PO 的关系如下。

**Employee** 是 **Manager** 的父类，同时 **Manager** 和 **Employee** 之间存在 1—N 的关系，即一个 **Manager** 对应多个 **Employee**。

**Employee** 和 **Payment** 之间存在 1—N 的关系，即每个员工可以多次领取薪水。

**Employee** 和 **Attend** 之间存在 1—N 的关系，即每个员工可以参与多次考勤，但每次考勤只对应一个员工。

**Manager** 继承了 **Employee** 类，因此具有 **Employee** 的全部属性，另外，该类还与 **CheckBack** 之间存在 1—N 的关系。

**Application** 与 **Attend** 之间存在 N—1 的关系，即每个 **Attend** 可以被员工多次申请。

**Application** 与 **AttendType** 之间存在 N—1 的关系，即每次申请都有明确的考勤类型，而一个考勤类型可对应多个申请。

**Attend** 与 **AttendType** 之间存在 N—1 的关系，即每个 **Attend** 只属于一个 **AttendType**。这 7 个类之间的类关系如图 10.2 所示。

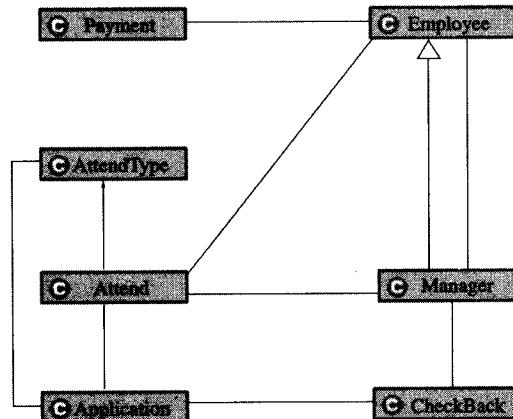


图 10.2 7 个 PO 之间的类关系图

## 10.2.2 创建持久化类

在图 10.2 中可以非常清晰地看到，这些关联关系以类属性的方式表现；持久化类之间的关系通过相应的 setter 和 getter 方法实现。这种关联关系通常对应数据库里的主、外

键约束。

除此之外，持久化对象还有自己的普通属性，这些普通属性通常对应数据库的字段。

Hibernate 对于持久化对象并没有太多额外的要求，只要求持久化对象提供无参数的构造器，实现 Serializable 接口，并重写 hashCode() 和 equals() 两个方法。

下面是 Employee PO 的源代码：

```
public class Employee implements Serializable
{
    private static final long serialVersionUID = 48L;
    //Employee 类的基本属性
    private int id;
    private String name;
    private String pass;
    private double salary;
    private Manager manager;
    //Employee 的关联属性
    private Set<Attend> attends = new HashSet<Attend>();
    private Set<Payment> payments = new HashSet<Payment>();
    //Employee 类的无参数构造器
    public Employee()
    {
    }
    //Employee 的重载的构造器
    public Employee(int id, String name, String pass, double salary,
        Manager manager) {
        this.id = id;
        this.name = name;
        this.pass = pass;
        this.salary = salary;
        this.manager = manager;
    }
    //标识属性 id 的 setter 方法
    public void setId(int id) {
        this.id = id;
    }
    //普通属性 name 的 setter 方法
    public void setName(String name) {
        this.name = name;
    }
    //普通属性 pass 的 setter 方法
    public void setPass(String pass) {
        this.pass = pass;
    }
    //普通属性 salary 的 setter 方法
    public void setSalary(double salary) {
        this.salary = salary;
    }
    //关联持久化类 Manager 的 setter 方法
    public void setManager(Manager manager) {
        this.manager = manager;
    }
    //关联持久化类 Attend 的 setter 方法
    public void setAttends(Set<Attend> attends)
    {
        this.attends = attends;
    }
    //关联持久化类 Attend 的 getter 方法
```

```
public Set<Attend> getAttends()
{
    return attends;
}
//关联持久化类 Payment 的 setter 方法
public void setPayments(Set<Payment> payments)
{
    this.payments = payments;
}
//关联持久化类 Payment 的 getter 方法
public Set<Payment> getPayments()
{
    return payments;
}
//标识属性 id 的 getter 方法
public int getId() {
    return (this.id);
}
//普通属性 name 的 getter 方法
public String getName() {
    return (this.name);
}
//普通属性 pass 的 getter 方法
public String getPass() {
    return (this.pass);
}
//普通属性 salary 的 getter 方法
public double getSalary() {
    return (this.salary);
}
//关联持久化类 Manager 的 getter 方法
public Manager getManager() {
    return (this.manager);
}
//重写的 equals 方法，决定两个对象相等的标准
public boolean equals (Object obj)
{
    //如果比较的对象为空，返回 false
    if (null == obj) return false;
    //如果对象不是 Employee 的实例，则不可能相等
    if (!(obj instanceof Employee))
    {
        return false;
    }
    else
    {
        //将被比较对象转换成 Employee 对象
        Employee employee = (Employee) obj;
        //如果 Employee 的 name 属性或被比较对象的 name 属性为空，则返回 false
        if (null == this.getName() || null == employee.getName() )
        {
            return false;
        }
        else
        {
            //只有当被比较对象的 name 和该对象 name 属性相等时返回 true
            return (this.getName().equals(employee.getName()));
        }
    }
}
```

```

    }
    //根据 name 属性计算 hashCode 值
    public int hashCode ()
    {
        return name.hashCode ();
    }

    public String toString () {
        return super.toString ();
    }
}

```

对 Hibernate 而言，完全支持将普通的 POJO 映射成 PO，但这些 POJO 应尽量遵守如下规则：

- 提供实现一个默认的（无参数的）构造器。
- 提供一个标识属性（identifier property）用于标识该实例。
- 使用非 final 的类。尽量避免将 POJO 声明成 final，这将导致其性能下降。

下面给出另外两个持久化类的源代码，一个是 Employee 的子类，需要使用继承映射；另一个是 Employee 的关联类，需要使用关联映射。

持久化类 Manager 的源代码：

```

//Manager 持久化类继承 Employee，实现 Serializable 的接口
public class Manager extends Employee implements Serializable
{
    private static final long serialVersionUID = 48L;
    //Manager 只有一个特别的属性：部门名
    private String dept;
    //两个 1-N 关联
    private Set<Employee> employees = new HashSet<Employee>();
    private Set<CheckBack> checks = new HashSet<CheckBack>();
    //无参数的构造器
    public Manager()
    {
    }
    //部门名的 getter 方法
    public String getDept()
    {
        return dept;
    }
    //部门名的 setter 方法
    public void setDept(String dept)
    {
        this.dept = dept;
    }
    //经理下属的 getter 方法
    public Set<Employee> getEmployees()
    {
        return this.employees;
    }
    //经理下属的 setter 方法
    public void setEmployees(Set<Employee> employees)
    {
        this.employees = employees;
    }
    //经理签署回执的 getter 方法

```

```
public Set<CheckBack> getChecks()
{
    return this.checks;
}
//经理签署回执的 setter 方法
public void setChecks(Set<CheckBack> checks)
{
    this.checks = checks;
}
}
```

Manager 持久化类是 Employee 的子类，Hibernate 也支持映射。

下面是 Attend 持久化类的源代码，该持久化类和 Employee 之间存在关联关系。

```
public class Attend implements Serializable
{
    private static final long serialVersionUID = 48L;
    //持久化类 Attend 的基本属性
    private int id ;
    private String dutyDay;
    private Date punchTime;
    private boolean isCome;
    //映射 N-1 关联
    private AttendType type;
    //映射 N-1 关联
    private Employee employee;
    //无参数的构造器
    public Attend()
    {
    }
    //有参数的构造器
    public Attend(int id, String dutyDay, Date punchTime, boolean isCome,
        AttendType type, Employee employee) {
        this.id = id;
        this.dutyDay = dutyDay;
        this.punchTime = punchTime;
        this.isCome = isCome;
        this.type = type;
        this.employee = employee;
    }
    //标识属性 id 的 setter 方法
    public void setId(int id) {
        this.id = id;
    }
    //当班日期的 setter 方法
    public void setDutyDay(String dutyDay) {
        this.dutyDay = dutyDay;
    }
    //打卡时间的 setter 方法
    public void setPunchTime(Date punchTime) {
        this.punchTime = punchTime;
    }
    //是否上班打卡的 setter 方法
    public void setIsCome(boolean isCome) {
        this.isCome = isCome;
    }
    //考勤类型的 setter 方法
    public void setType(AttendType type) {
```

```
        this.type = type;
    }
    //对应员工考勤的 setter 方法
    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
    //标识属性 id 的 getter 方法
    public int getId() {
        return (this.id);
    }
    //当班日期的 getter 方法
    public String getDutyDay() {
        return (this.dutyDay);
    }
    //打卡时间的 getter 方法
    public Date getPunchTime() {
        return (this.punchTime);
    }
    //是否上班打卡的 getter 方法
    public boolean getIsCome() {
        return (this.isCome);
    }
    //考勤类型的 getter 方法
    public AttendType getType() {
        return (this.type);
    }
    //关联 Employee 的 getter 方法
    public Employee getEmployee() {
        return (this.employee);
    }
    //重写 equals 方法，当考勤日期、考勤员工、同为上、或者下班考勤时，  
//两个考勤对象相等。
    public boolean equals (Object obj)
    {
        if (null == obj) return false;
        if (!(obj instanceof Attend))
        {
            return false;
        }
        else
        {
            Attend attend = (Attend) obj;
            if (null == this.getDutyDay() || null == attend.getDutyDay()
                || null == this.getEmployee() || null == attend.getEmployee())
            {
                return false;
            }
            else
            {
                return (this.getDutyDay().equals(attend.getDutyDay()))
                    && this.getEmployee().equals(attend.getEmployee())
                    && attend.getIsCome() == getIsCome();
            }
        }
    }
    //重写 hashCode 方法
    public int hashCode ()
    {
        if (getIsCome())
```

```

    {
        return dutyDay.hashCode() + employee.hashCode() + 1;
    }
    return dutyDay.hashCode() + employee.hashCode();
}
//重写 toString 方法
public String toString () {
    return super.toString();
}
}
}

```

### 10.2.3 映射持久化类

在面向对象分析的阶段中，最重要的任务就是提取对象，并分析对象之间的关联关系。本实例中的关联关系多表现为 1—N 关联。对于 1—N 的关联，假设要实现一个简单的从 Parent 到 Child 的 1—N 关联。其映射代码如下：

```

<!-- 关联的持久化类集合-->
<set name="children">
    <!-- 外键列的列名-->
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>

```

如果采用如下代码关联 Parent 与 Child 的关系：

```

Parent p = new Person;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();

```

Hibernate 会产生两条 SQL 语句：

一条 INSERT 语句，为 c 创建一条记录。

一条 UPDATE 语句，创建从 p 到 c 的关联。

采用这种方式不仅效率低，而且违反了列 parent\_id 非空的约束。我们可以通过在集合类映射上指定 not-null="true" 来解决此问题：

```

<set name="children">
    <!-- 增加非空约束-->
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>

```

产生这种现象的根本原因是：p 到 c 的关联不是 Child 对象的一部分，因而在 INSERT 语句中没有创建该外键值。解决的办法是将关联添加到 Child 的映射中。

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

并改为由实体 Child 在管理关联，为了使 Collection 不更新连接，设置 inverse="true"，如下所示：

```
<!-- 1 的一端不再管理关联-->
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

下面的代码是用来添加一个新的 Child:

```
Parent p = new Parent();
Child c = new Child();
c.setParent(p);
session.save(c);
session.flush();
```

现在，只会有一条 INSERT 语句被执行。为了让 Person 端也可以“管理关联”，可以为 Parent 加一个 addChild()方法。

```
//由 Parent 管理关联的方法
public void addChild(Child c)
{
    //实际依然由 Child 一端控制关联
    c.setParent(this);
    children.add(c);
}
```

另外，也可以通过如下方式来增加它们之间的关联。

```
Parent p = new Person
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

通过上面的讨论，对于 1-N 的关联关系，通常推荐映射成双向关联，而且由 N 的一端控制关联关系。

**注意：**对所有 1-N 的关联关系，建议不要使用“1”的一端控制关系，因此建议为 set 元素增加 inverse=“true”属性，让“N”的一端来控制关联关系。

除此之外，分析对象之间的继承层次也是非常重要的任务。在面向对象设计里，继承体现了软件复用的概念。因此，在本应用中，Manager 需要继承 Employee 类。

Hibernate 支持三种继承映射策略。

**subclass** 元素映射子类：用一张表存储整个继承树的全部实例。

**joined-subclass** 元素映射子类：继承树的每层实例对应一张表，且基类的表中保存所有子类的公有列，因此如需创建子类实例，总是需要查询基类的表数据，其子类所在深度越深，查询涉及到的表就越多。

**unioned-subclass** 元素映射子类：继承树的每层实例对应一张表，每层的实例完整地保存在对应的表中，表中的每条记录即可对应一个实例。

关于继承策略的选择，建议使用 unioned-subclass 的继承映射策略，理由如下。

- 使用 subclass 的映射策略时，因为整个继承树的全部实例都保存在一张表内，因

此子类新增的数据列都不可增加非空约束，即使实际需要非空约束也不行，因为父类实例对应的记录根本没有该列，这将导致数据库的约束降低，同时也增加数据冗余，可以说这是最差的映射策略。

- 使用 joined-subclass 映射策略可以避免上面的问题，但使用这种映射策略时，如需查询多层下的子类，可能要访问的表数据过多，影响性能。

而使用 unioned-subclass 的映射策略则可以避免上面的两个问题。

一旦映射了合适的继承策略，Hibernate 完全可以理解多态查询。即查询 Parent 类的实例时，所有 Parent 子类的实例也可被查询到。

下面是本系统所使用的 6 个映射文件。

### Employee 和 Manager 的映射文件

```
<?xml version="1.0"?>
<!!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="org.yeeku.model">
    <!-- 映射持久化类 Employee-->
    <class name="Employee" table="emp_table">
        <!-- 映射标识属性-->
        <id name="id" type="integer" column="emp_id">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 name-->
        <property name="name" column="emp_name" type="string" not-null="true"
            length="50" unique="true"/>
        <!-- 映射普通属性 pass -->
        <property name="pass" column="emp_pass" type="string"
            not-null="true" length="50"/>
        <!-- 映射普通属性 salary -->
        <property name="salary" column="emp_salary" type="double"
            not-null="true" />
        <!-- 映射关联的持久化类：Manager-->
        <many-to-one name="manager" column="mgr_id" lazy="false"/>
        <!-- 映射 1-N 的关联关系：Attend-->
        <set name="attends" inverse="true">
            <key column="emp_id" />
            <one-to-many class="Attend"/>
        </set>
        <!-- 映射 1-N 的关联关系：Payment -->
        <set name="payments" inverse="true">
            <key column="emp_id" />
            <one-to-many class="Payment"/>
        </set>
        <!-- 映射子类：Manager，使用 joined_subclass -->
        <joined-subclass name="Manager" table="mgr_table">
            <!-- 映射标识属性，该属性与父类的标识属性对应，无需主键生成器-->
            <key column="mgr_id"/>
            <!-- 映射普通属性 dept-->
            <property name="dept" column="dept_name" type="string"
                not-null="true" length="50"/>
            <!-- 映射 1-N 关联关系，子类与父类具有关联-->
            <set name="employees" inverse="true">
```

```

<key column="mgr_id" />
<one-to-many class="Employee" />
</set>
<!-- 映射 1-N 关联关系 -->
<set name="checks" inverse="true">
    <key column="mgr_id" />
    <one-to-many class="CheckBack" />
</set>
</joined-subclass>
</class>
</hibernate-mapping>

```

Attend 的映射文件如下：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="org.yeeku.model">
    <!-- 映射持久化类: Attend-->
    <class name="Attend" table="attend_table">
        <!-- 映射标识属性-->
        <id name="id" type="integer" column="attend_id">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性: dutyDay -->
        <property name="dutyDay" column="duty_day" type="string" not-null
            = "true" length="50" />
        <!-- 映射普通属性: punchTime -->
        <property name="punchTime" column="punch_time" type="java.util.Date" />
        <!-- 映射普通属性: isCome -->
        <property name="isCome" column="is_come" type="boolean" not-null="true" />
        <!-- 映射 N-1 关联关系-->
        <many-to-one name="type" column="type_id" not-null="true" lazy="false" />
        <!-- 映射 N-1 关联关系-->
        <many-to-one name="employee" column="emp_id" not-null="true" lazy="false" />
    </class>
</hibernate-mapping>

```

AttendType 的映射文件如下：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://ibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="org.yeeku.model">
    <!-- 映射持久化类 AttendType-->
    <class name="AttendType" table="type_table">
        <!-- 映射标识属性-->
        <id name="id" type="integer" column="type_id">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性: name -->

```

```

<property name="name" column="type_name" type="string" not-null="true" length="50"/>
<!-- 映射普通属性: amerce -->
<property name="ameerce" column="ameerce_amount" type="double" not-null="true" />
</class>
</hibernate-mapping>

```

Application 的映射文件如下：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="org.yeeku.model">
    <!-- 映射持久化类: Application-->
    <class name="Application" table="app_table">
        <!-- 映射标识属性-->
        <id name="id" type="integer" column="app_id">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性: reason -->
        <property name="reason" column="app_reason" type="string" length="50"/>
        <!-- 映射普通属性: result -->
        <property name="result" column="app_result" type="boolean"/>
        <!-- 映射 N-1 关联关系-->
        <many-to-one name="type" column="type_id" not-null="true" lazy="false"/>
        <!-- 映射 N-1 关联关系-->
        <many-to-one name="attend" column="attend_id" not-null="true" lazy="false"/>
        <!-- 映射 1-1 关联关系-->
        <one-to-one name="check" property-ref="app"/>
    </class>
</hibernate-mapping>

```

CheckBack 的映射文件如下：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头，包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://ibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="org.yeeku.model">
    <!-- 映射持久化类 CheckBack-->
    <class name="CheckBack" table="check_table">
        <!-- 映射标识属性-->
        <id name="id" type="integer" column="check_id">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性: reason -->
        <property name="reason" column="check_reason" type="string" length="50"/>
    </class>
</hibernate-mapping>

```

```

<!-- 映射普通属性: result -->
<property name="result" column="check_result" type="boolean"
length="50" not-null="true"/>
<!-- 映射 1-1 的关联, 通过 unique="true", 表明为 1-1 关联 -->
<many-to-one name="app" column="app_id" not-null="true"
unique="true" lazy="false"/>
<!-- 映射 N-1 的关联, 通过 unique="true", 表明为 N-1 关联 -->
<many-to-one name="manager" column="mgr_id" not-null="true"
lazy="false"/>
</class>
</hibernate-mapping>

```

Payment 的映射文件如下：

```

<?xml version="1.0"?>
<!-- Hibernate 映射文件的文件头, 包含 DTD 等信息-->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<!-- Hibernate 映射文件的根元素-->
<hibernate-mapping package="org.yeeku.model">
    <!-- 映射持久化类 Payment-->
    <class name="Payment" table="pay_table">
        <!-- 映射标识属性-->
        <id name="id" type="integer" column="pay_id">
            <!-- 指定主键生成器策略-->
            <generator class="identity"/>
        </id>
        <!-- 映射普通属性 payMonth-->
        <property name="payMonth" column="pay_month" type="string"
not-null="true" length="50"/>
        <!-- 映射普通属性 amount -->
        <property name="amount" column="pay_amount"      type="double"
not-null="true"/>
        <!-- 映射 N-1 关联关系 -->
        <many-to-one name="employee" column="emp_id" not-null="true"
lazy="false"/>
    </class>
</hibernate-mapping>

```

正如前面所提示：所有的 set 元素都被加上 inverse=“true” 声明，让“N”的一端控制关联关系，而不要让“1”的一端控制关联关系。

## 10.3 实现 DAO 层

在 Hibernate 持久层之上，可使用 DAO 组件再次封装数据库操作。通过 DAO 层，可以让业务逻辑层与具体持久层技术分离，一旦需要更换持久层技术时，业务逻辑层组件不需要任何改变。因此，使用 DAO 组件，即意味着引入 DAO 模式，使每个 DAO 组件包含了数据库的访问逻辑；每个 DAO 组件可对一个数据库表完成基本的 CRUD 等操作。

DAO 模式的实现至少需要如下三个部分。

- DAO 工厂类。
- DAO 接口。

- DAO 接口的实现类。

DAO 模式是一种更符合软件工程的开发方式，使用 DAO 模式有如下理由。

- DAO 模式抽象出数据访问方式，业务逻辑组件无须理会底层的数据库访问，而只专注于业务逻辑的实现。
- DAO 将数据访问集中在独立的一层，所有的数据访问都由 DAO 对象完成，这层独立的 DAO 分离了数据访问的实现与其他业务逻辑，使得系统更具可维护性。
- DAO 还有助于提升系统的可移植性。独立的 DAO 层使得系统能在不同的数据库之间轻易切换，底层的数据库实现对于业务逻辑组件是透明的。数据库移植时仅仅影响 DAO 层，不同数据库的切换不会影响业务逻辑组件，因此提高了系统的可复用性。
- 对于不同的持久层技术，Spring 的 DAO 提供一个 DAO 模板，将通用的操作放在模板里完成，而对于特定的操作，则通过回调接口完成。

Spring 为 Hibernate 提供的 DAO 支持类是：HibernateDaoSupport。

### 10.3.1 DAO 组件的定义

DAO 组件提供了各持久化对象的基本的 CRUD 操作。而在 DAO 接口里则对 DAO 组件包含的各种 CRUD 方法提供了声明，但有一些 IDE 工具也可以生成基本的 CRUD 方法。使用 DAO 接口的原因是：避免业务逻辑组件与特定的 DAO 组件耦合。

由于 DAO 组件中的方法不是一开始就设计出来的，其中的很多方法可能会随着业务逻辑的需求而增加，但以下几个方法是通用的。

- **get**: 根据主键加载持久化实例。
- **save**: 保存持久化实例。
- **update**: 更新持久化实例。
- **delete**: 删除持久化实例。

**注意：**在经典的 EJB 中，Entity EJB 是个功能非常强大的 DAO 组件，其功能远远超出这里实现的 DAO 组件。Entity EJB 同样包含了四个类似的方法。

DAO 接口无须给出任何实现，仅仅是 DAO 组件包含的 CRUD 方法的定义，这些方法定义的实现取决于底层的持久化技术，DAO 组件的实现既可以使用传统 JDBC，也可以采用 Hibernate 持久化技术，以及 iBATIS 等技术。下面是关于 DAO 接口的源代码。

ApplicationDao 的接口定义如下：

```
public interface ApplicationDao
{
    /**
     * 根据 id 查找异动申请
     * @param id 需要查找的异动申请 id
     */
    Application get(Integer id);
    /**
     * 增加异动申请
     */

    void add(Application application);
}
```

```

    * @param application 需要增加的异动申请
    */
void save(Application application);
/**
    * 修改异动申请
    * @param application 需要修改的异动申请
    */
void update(Application application);
/**
    * 删除异动申请
    * @param id 需要删除的异动申请 id
    */
void delete(Integer id);
/**
    * 删除异动申请
    * @param application 需要删除的异动申请
    */
void delete(Application application);
/**
    * 查询全部异动申请
    * @return 全部异动申请
    */
List<Application> findAll();
/**
    * 根据员工查询未处理的异动申请
    * @param emp 需要查询的员工
    * @return 该员工对应的未处理的异动申请
    */
List<Application> findByEmp(Employee emp);
}

```

AttendDao 的接口定义如下：

```

public interface AttendDao
{
    /**
     * 根据 id 查找打卡记录
     * @param id 需要查找的打卡记录 id
     */
    Attend get(Integer id);
    /**
     * 增加打卡记录
     * @param attend 需要增加的打卡记录
     */
    void save(Attend attend);
    /**
     * 修改打卡记录
     * @param attend 需要修改的打卡记录
     */
    void update(Attend attend);
    /**
     * 删除打卡记录
     * @param id 需要删除的打卡记录 id
     */
    void delete(Integer id);
    /**
     * 删除打卡记录
     * @param attend 需要删除的打卡记录
     */
}

```

```
/*
void delete(Attend attend);
/**
 * 查询全部打卡记录
 * @return 全部打卡记录
 */
List<Attend> findAll();
/**
 * 根据员工查询该员工的打卡记录
 * @param emp 员工
 * @return 该员工的全部出勤记录
 */
List<Attend> findByEmp(Employee emp);
/**
 * 根据员工、日期查询该员工的打卡记录集合
 * @param emp 员工
 * @param dutyDay 日期
 * @return 该员工某天的打卡记录集合
 */
List<Attend> findByEmpAndDutyDay(Employee emp, String dutyDay);
/**
 * 根据员工、日期、上下班查询该员工的打卡记录集合
 * @param emp 员工
 * @param dutyDay 日期
 * @param isCome 是否上班
 * @return 该员工某天上班或下班的打卡记录
 */
Attend findByEmpAndDutyDayAndCome(Employee emp, String dutyDay, boolean
isCome);
/**
 * 查看员工前三天的非正常打卡
 * @param emp 员工
 * @return 该员工前三天的非正常打卡
 */
List<Attend> findByEmpUnAttend(Employee emp, AttendType type);
}
```

AttendTypeDao 的接口定义如下：

```
public interface AttendTypeDao
{
    /**
     * 根据 id 查找出勤种类
     * @param id 需要查找的出勤种类 id
     */
    AttendType get(Integer id);
    /**
     * 增加出勤种类
     * @param type 需要增加的出勤种类
     */
    void save(AttendType type);
    /**
     * 修改出勤种类
     * @param type 需要修改的出勤种类
     */
    void update(AttendType type);
    /**
     * 删出勤种类
     */
```

```

    * @param id 需要删除的出勤种类 id
    */
void delete(Integer id);
/***
 * 删除出勤种类
 * @param type 需要删除的出勤种类
 */
void delete(AttendType type);
/***
 * 查询全部出勤种类
 * @return 全部出勤种类
 */
List<AttendType> findAll();
}

```

CheckBackDao 的接口定义如下：

```

public interface CheckBackDao
{
    /**
     * 根据 id 查找申请批复
     * @param id 需要查找的申请批复 id
     */
    CheckBack get(Integer id);
    /**
     * 增加申请批复
     * @param check 需要增加的申请批复
     */
    void save(CheckBack check);
    /**
     * 修改申请批复
     * @param check 需要修改的申请批复
     */
    void update(CheckBack check);
    /**
     * 删除申请批复
     * @param id 需要删除的申请批复 id
     */
    void delete(Integer id);
    /**
     * 删除申请批复
     * @param check 需要删除的申请批复
     */
    void delete(CheckBack check);
    /**
     * 查询全部申请批复
     * @return 全部申请批复
     */
    List<CheckBack> findAll();
}

```

EmployeeDao 的接口定义如下：

```

public interface EmployeeDao
{
    /**
     * 根据 id 查找员工
     * @param id 需要查找的员工 id
     */

```

```
/*
Employee get(Integer id);
/**
 * 增加员工
 * @param emp 需要增加的员工
 */
void save(Employee emp);
/**
 * 修改员工
 * @param emp 需要修改的员工
 */
void update(Employee emp);
/**
 * 删除员工
 * @param id 需要删除的员工 id
 */
void delete(Integer id);
/**
 * 删除员工
 * @param emp 需要删除的员工
 */
void delete(Employee emp);
/**
 * 查询全部员工
 * @return 全部员工
 */
List<Employee> findAll();
/**
 * 根据用户名和密码查询员工
 * @param name 员工的用户名
 * @param pass 员工的密码
 * @return 符合用户名和密码的员工集合
 */
List<Employee> findByNameAndPass(String name , String pass);
/**
 * 根据用户名查询员工
 * @param name 员工的用户名
 * @return 符合用户名的员工
 */
Employee findByName(String name);
/**
 * 根据经理查询员工
 * @param mgr 经理
 * @return 符合经理下的所有员工
 */
List<Employee> findByMgr(Manager mgr);
}
```

ManagerDao 的接口定义如下：

```
public interface ManagerDao
{
    /**
     * 根据 id 查找员工
     * @param id 需要查找的经理 id
     */
    Manager get(Integer id);
    /**
```

```

    * 增加经理
    * @param mgr 需要增加的经理
    */
void save(Manager mgr);
/***
    * 修改经理
    * @param mgr 需要修改的经理
    */
void update(Manager mgr);
/***
    * 删除经理
    * @param id 需要删除的经理 id
    */
void delete(Integer id);
/***
    * 删除经理
    * @param mgr 需要删除的经理
    */
void delete(Manager mgr);
/***
    * 查询全部经理
    * @return 全部经理
    */
List<Manager> findAll();
/***
    * 根据用户名和密码查询经理
    * @param name 经理的用户名
    * @param pass 经理的密码
    * @return 符合用户名和密码的经理
    */
List<Manager> findByNameAndPass(String name , String pass);
/***
    * 根据用户名查找经理
    * @param 经理的名字
    * @return 名字对应的经理
    */
Manager findByName(String name);
}

```

PaymentDao 的接口定义如下：

```

public interface PaymentDao
{
    /**
     * 根据 id 查找月结薪水
     * @param id 需要查找的月结薪水 id
     */
    Payment get(Integer id);
    /**
     * 增加月结薪水
     * @param payment 需要增加的月结薪水
     */
    void save(Payment payment);
    /**
     * 修改月结薪水
     * @param payment 需要修改的月结薪水
     */
    void update(Payment payment);
}

```

```

    /**
     * 删除月结薪水
     * @param id 需要删除的月结薪水 id
     */
    void delete(Integer id);
    /**
     * 删除月结薪水
     * @param payment 需要删除的月结薪水
     */
    void delete(Payment payment);
    /**
     * 查询全部月结薪水
     * @return 全部月结薪水
     */
    List<Payment> findAll();
    /**
     * 根据员工查询月结薪水
     * @return 员工对应的月结薪水
     */
    List<Payment> findByEmp(Employee emp);
    /**
     * 根据员工和发薪水月查看月结薪水
     * @param payMonth 发薪月份
     * @param emp 领薪的员工
     * @return 员工对应的月结薪水
     */
    Payment findByMonthAndEmp(String payMonth, Employee emp);
}

```

### 10.3.2 实现 DAO 组件

借助于 Spring 的 DAO 支持，可以很方便地实现 DAO 类。

Spring 为 Hibernate 的整合提供了很好的支持，Spring 的 DAO 支持类是：HiberanteDaoSupport，该类只需要传入一个 SessionFactory 引用，即可得到一个 HibernateTemplate 实例，该实例功能非常强大，数据库的大部分操作也很容易实现。

所有的 DAO 类都继承 HibernateDaoSupport，并实现相应的 DAO 接口。而业务逻辑对象则面向接口编程，无须关心 DAO 的实现细节。通过这种方式，可以让应用在不同的持久化技术之间切换。

如下是 ApplicationDao 接口的实现类 ApplicationDaoHibernate 的源代码：

```

public class ApplicationDaoHibernate extends HibernateDaoSupport
    implements ApplicationDao
{
    /**
     * 根据 id 查找异动申请
     * @param id 需要查找的异动申请 id
     */
    public Application get(Integer id)
    {
        return (Application) getHibernateTemplate().get(Application.class, id);
    }
    /**
     * 增加异动申请
     */
}

```

```
* @param application 需要增加的异动申请
*/
public void save(Application application)
{
    getHibernateTemplate().save(application);
}
/***
 * 修改异动申请
 * @param application 需要修改的异动申请
 */
public void update(Application application)
{
    getHibernateTemplate().saveOrUpdate(application);
}
/***
 * 删除异动申请
 * @param id 需要删除的异动申请 id
 */
public void delete(Integer id)
{
    getHibernateTemplate().delete(getHibernateTemplate().
        get (Application.class , id));
}
/***
 * 删除异动申请
 * @param application 需要删除的异动申请
 */
public void delete(Application application)
{
    getHibernateTemplate().delete(application);
}
/***
 * 查询全部异动申请
 * @return 全部异动申请
 */
public List<Application> findAll()
{
    return (List<Application>)getHibernateTemplate().find(
        "from Application");
}
/***
 * 根据员工查询未处理的异动申请
 * @param emp 需要查询的员工
 * @return 该员工对应的未处理的异动申请
 */
public List<Application> findByEmp(Employee emp)
{
    List<Application> apps = (List<Application>)getHibernateTemplate()
        .find("from Application as a where a.attend.employee=? ", emp);
    if (apps == null || apps.size() < 1 )
    {
        return null;
    }
    List<Application> result = new ArrayList<Application>();
    for (Application app : apps )
    {
        result.add(app);
    }
    return result;
}
```

```
    }  
}
```

如下是 AttendDao 接口的实现类 AttendDaoHibernate 的源代码：

```
public class AttendDaoHibernate extends HibernateDaoSupport  
    implements AttendDao  
{  
    /**  
     * 根据 id 查找打卡记录  
     * @param id 需要查找的打卡记录 id  
     */  

```

```
* @return 该员工的全部出勤记录
*/
public List<Attend> findByEmp(Employee emp)
{
    return (List<Attend>)getHibernateTemplate()
        .find("from Attend as a where a.employee = ?" , emp);
}
/***
 * 根据员工、日期查询该员工的打卡记录集合
 * @param emp 员工
 * @param dutyDay 日期
 * @return 该员工某天的打卡记录集合
 */
public List<Attend> findByEmpAndDutyDay(Employee emp , String dutyDay)
{
    Object[] args = {emp , dutyDay};
    return (List<Attend>)getHibernateTemplate()
        .find("from Attend as a where a.employee = ? and
a.dutyDay = ?" , args);
}
/***
 * 根据员工、日期、上下班查询该员工的打卡记录集合
 * @param emp 员工
 * @param dutyDay 日期
 * @param isCome 是否上班
 * @return 该员工某天上班或下班的打卡记录
 */
public Attend findByEmpAndDutyDayAndCome(Employee emp , String dutyDay ,
boolean isCome)
{
    Object[] args = {emp , dutyDay};
    List<Attend > attends = (List<Attend>)getHibernateTemplate()
        .find("from Attend as a where a.employee = ? and
a.dutyDay = ?"
        , args);
    if (attends == null || attends.size() < 1)
    {
        return null;
    }
    for (Attend attend : attends )
    {
        if (attend.getIsCome() == isCome )
        {
            return attend;
        }
    }
    return null;
}
/***
 * 查看员工前三天的非正常打卡
 * @param emp 员工
 * @return 该员工前三天的非正常打卡
 */
public List<Attend> findByEmpUnAttend(Employee emp , AttendType type)
{
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Calendar c = Calendar.getInstance();
    String end = sdf.format(c.getTime());
    c.add(Calendar.DAY_OF_MONTH, -3);
```

```

        String start = sdf.format(c.getTime());
        Object[] args = {emp , type , start , end};
        return (List<Attend>)getHibernateTemplate()
            .find("from Attend as a where a.employee = ? and a.type != ? and
                  a.dutyDay between ? and ?" , args);
    }
}

```

如下是 `AttendTypeDao` 接口的实现类 `AttendTypeDaoHibernate` 的源代码：

```

public class AttendTypeDaoHibernate extends HibernateDaoSupport
    implements AttendTypeDao
{
    /**
     * 根据 id 查找出勤种类
     * @param id 需要查找的出勤种类 id
     */
    public AttendType get(Integer id)
    {
        return (AttendType)getHibernateTemplate().get(AttendType.class , id);
    }
    /**
     * 增加出勤种类
     * @param type 需要增加的出勤种类
     */
    public void save(AttendType type)
    {
        getHibernateTemplate().save(type);
    }
    /**
     * 修改出勤种类
     * @param type 需要修改的出勤种类
     */
    public void update(AttendType type)
    {
        getHibernateTemplate().saveOrUpdate(type);
    }
    /**
     * 删出勤种类
     * @param id 需要删除的出勤种类 id
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(getHibernateTemplate().
            get(AttendType.class , id));
    }
    /**
     * 删出勤种类
     * @param type 需要删除的出勤种类
     */
    public void delete(AttendType type)
    {
        getHibernateTemplate().delete(type);
    }
    /**
     * 查询全部出勤种类
     * @return 全部出勤种类
     */
    public List<AttendType> findAll()

```

```

    {
        return (List<AttendType>)getHibernateTemplate().find("from AttendType");
    }
}

```

如下是 CheckBackDao 接口的实现类：CheckBackDaoHibernate 的源代码：

```

public class CheckBackDaoHibernate extends HibernateDaoSupport
    implements CheckBackDao
{
    /**
     * 根据 id 查找申请批复
     * @param id 需要查找的申请批复 id
     */
    public CheckBack get(Integer id)
    {
        return (CheckBack)getHibernateTemplate().get(CheckBack.class , id);
    }
    /**
     * 增加申请批复
     * @param check 需要增加的申请批复
     */
    public void save(CheckBack check)
    {
        getHibernateTemplate().save(check);
    }
    /**
     * 修改申请批复
     * @param check 需要修改的申请批复
     */
    public void update(CheckBack check)
    {
        getHibernateTemplate().saveOrUpdate(check);
    }
    /**
     * 删除申请批复
     * @param id 需要删除的申请批复 id
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(getHibernateTemplate().
            get(CheckBack.class , id));
    }
    /**
     * 删除申请批复
     * @param check 需要删除的申请批复
     */
    public void delete(CheckBack check)
    {
        getHibernateTemplate().delete(check);
    }
    /**
     * 查询全部申请批复
     * @return 全部申请批复
     */
    public List<CheckBack> findAll()
    {
        return (List<CheckBack>)getHibernateTemplate().find("from CheckBack");
    }
}

```

}

如下是 EmployeeDao 接口的实现类 EmployeeDaoHibernate 的源代码：

```
public class EmployeeDaoHibernate extends HibernateDaoSupport
    implements EmployeeDao
{
    /**
     * 根据 id 查找员工
     * @param id 需要查找的员工 id
     */
    public Employee get(Integer id)
    {
        return (Employee)getHibernateTemplate().get(Employee.class , id);
    }
    /**
     * 增加员工
     * @param emp 需要增加的员工
     */
    public void save(Employee emp)
    {
        getHibernateTemplate().save(emp);
    }
    /**
     * 修改员工
     * @param emp 需要修改的员工
     */
    public void update(Employee emp)
    {
        getHibernateTemplate().saveOrUpdate(emp);
    }
    /**
     * 删除员工
     * @param id 需要删除的员工 id
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(getHibernateTemplate().
            get(Employee.class , id));
    }
    /**
     * 删除员工
     * @param emp 需要删除的员工
     */
    public void delete(Employee emp)
    {
        getHibernateTemplate().delete(emp);
    }
    /**
     * 查询全部员工
     * @return 全部员工
     */
    public List<Employee> findAll()
    {
        return (List<Employee>)getHibernateTemplate().find("from Employee");
    }
    /**
     * 根据用户名和密码查询员工
     * @param name 员工的用户名
     */
```

```

    * @param pass 员工的密码
    * @return 符合用户名和密码的员工集合
    */
public List<Employee> findByNameAndPass(String name, String pass)
{
    String[] args = {name, pass};
    return (List<Employee>)getHibernateTemplate()
        .find("from Employee where name = ? and pass = ?" , args);
}
/***
    * 根据用户名查询员工
    * @param name 员工的用户名
    * @return 符合用户名的员工
    */
public Employee findByName(String name)
{
    List<Employee> emps = (List<Employee>)getHibernateTemplate()
        .find("from Employee where name = ?" , name);
    if (emps.size() >= 1)
    {
        return emps.get(0);
    }
    return null;
}
/***
    * 根据经理查询员工
    * @param mgr 经理
    * @return 符合经理下的所有员工
    */
public List<Employee> findByMgr(Manager mgr)
{
    return (List<Employee>)getHibernateTemplate().
        find("from Employee as e where e.manager = ?" , mgr);
}
}

```

如下是 ManagerDao 接口的实现类 ManagerDaoHibernate 的源代码：

```

{
    /**
     * 根据 id 查找经理
     * @param id 需要查找的经理 id
     */
    public Manager get(Integer id)
    {
        return (Manager)getHibernateTemplate().get(Manager.class , id);
    }
    /**
     * 增加经理
     * @param mgr 需要增加的经理
     */
    public void save(Manager mgr)
    {
        getHibernateTemplate().save(mgr);
    }
    /**
     * 修改经理
     * @param mgr 需要修改的经理
     */

```

```
public void update(Manager mgr)
{
    getHibernateTemplate().saveOrUpdate(mgr);
}
/**
 * 删除经理
 * @param id 需要删除的经理 id
 */
public void delete(Integer id)
{
    getHibernateTemplate().delete(getHibernateTemplate().
        get(Manager.class , id));
}
/**
 * 删除经理
 * @param mgr 需要删除的经理
 */
public void delete(Manager mgr)
{
    getHibernateTemplate().delete(mgr);
}
/**
 * 查询全部经理
 * @return 全部经理
 */
public List<Manager> findAll()
{
    return (List<Manager>)getHibernateTemplate().find("from Manager");
}
/**
 * 根据用户名和密码查询经理
 * @param name 经理的用户名
 * @param pass 经理的密码
 * @return 符合用户名和密码的经理
 */
public List<Manager> findByNameAndPass(String name , String pass)
{
    String[] args = {name,pass};
    return (List<Manager>)getHibernateTemplate().
        find("from Manager where name = ? and pass = ?" , args);
}
/**
 * 根据用户名查找经理
 * @param 经理的名字
 * @return 名字对应的经理
 */
public Manager findByName(String name)
{
    List<Manager> ml = (List<Manager>)getHibernateTemplate()
        .find("from Manager where name=? " , name);
    if (ml != null && ml.size() > 0)
    {
        return ml.get(0);
    }
    return null;
}
}
```

如下是 PaymentDao 接口的实现类 PaymentDaoHibernate 的源代码：

```
public class PaymentDaoHibernate extends HibernateDaoSupport
    implements PaymentDao
{
    /**
     * 根据 id 查找月结薪水
     * @param id 需要查找的月结薪水 id
     */
    public Payment get(Integer id)
    {
        return (Payment)getHibernateTemplate().get(Payment.class , id);
    }
    /**
     * 增加月结薪水
     * @param payment 需要增加的月结薪水
     */
    public void save(Payment payment)
    {
        getHibernateTemplate().save(payment);
    }
    /**
     * 修改月结薪水
     * @param payment 需要修改的月结薪水
     */
    public void update(Payment payment)
    {
        getHibernateTemplate().saveOrUpdate(payment);
    }
    /**
     * 删除月结薪水
     * @param id 需要删除的月结薪水 id
     */
    public void delete(Integer id)
    {
        getHibernateTemplate().delete(getHibernateTemplate().get(Payment.
            class , id));
    }
    /**
     * 删除月结薪水
     * @param payment 需要删除的月结薪水
     */
    public void delete(Payment payment)
    {
        getHibernateTemplate().delete(payment);
    }
    /**
     * 查询全部月结薪水
     * @return 全部月结薪水
     */
    public List<Payment> findAll()
    {
        return (List<Payment>)getHibernateTemplate().find("from Payment");
    }
    /**
     * 根据员工查询月结薪水
     * @return 员工对应的月结薪水
     */
    public List<Payment> findByEmp(Employee emp)
```

```

    {
        return (List<Payment>)getHibernateTemplate()
            .find("from Payment as p where p.employee = ?" , emp);
    }
    /**
     * 根据员工和发薪水月查看月结薪水
     * @param payMonth 发薪月份
     * @param emp 领薪的员工
     * @return 员工对应的月结薪水
     */
    public Payment findByMonthAndEmp(String payMonth , Employee emp)
    {
        Object[] args = { emp , payMonth};
        List<Payment> pays = (List<Payment>)getHibernateTemplate()
            .find("from Payment as p where p.employee = ? and p.payMonth = ?" ,
                args);
        if (pays == null || pays.size() < 1)
        {
            return null;
        }
        return pays.get(0);
    }
}

```

借助于 Spring 提供的两个工具类： HibernateDaoSupport 和 HibernateTemplate，可以很容易地实现 DAO 组件。

这种简单的实现较之传统的 JDBC 持久化访问，简直不可同日而语。Hiberate 为持久化访问提供了第一层封装，而 Spring 在 Hibernate 的基础上再次简化了持久层的访问。

**注意：**学习框架的过程中也许会有少许的坎坷，但一旦掌握了框架的使用，将大幅度地提高应用的开发效率，而且好的框架所倡导的软件架构还会提高开发者的架构设计知识。

### 10.3.3 部署 DAO 层

通过前面的学习，我们知道， HibernateDaoSupport 类只需要一个 SessionFactory 属性，即可完成数据库访问。而实际的数据库访问由模板类 HibernateTemplate 完成，该模板类提供了大量便捷的方法，简化了数据库的访问。

#### 1. DAO 组件运行的基础

应用的 DAO 组件以 Hibernate 和 Spring 为基础，由 Spring 容器负责生成并管理 DAO 组件。Spring 容器负责为 DAO 组件注入其运行所需要的基础 SessionFactory。

Spring 为整合 Hibernate 提供了大量工具类，通过 LocalSessionFactoryBean 类，可以将 Hibernate 的 SessionFactory 纳入其 IoC 容器内。

使用 LocalSessionFactoryBean 配置 SessionFactory 之前，必须为其提供对应的数据源，配置代码如下：

```

<!-- 配置 dataSource bean-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"

```

```
destroy-method="close">
<!-- 设置数据源使用的驱动类-->
<property name="driverClass">
    <value>com.mysql.jdbc.Driver</value>
</property>
<!-- 设置数据库服务所在的 url-->
<property name="jdbcUrl">
    <value>jdbc:mysql://localhost/auction</value>
</property>
<!-- 设置数据库的用户名-->
<property name="user">
    <value>root</value>
</property>
<!-- 设置数据库的密码-->
<property name="password">
    <value>32147</value>
</property>
<!-- 设置连接池的最大容量-->
<property name="maxPoolSize">
    <value>40</value>
</property>
<!-- 设置连接池的最小容量-->
<property name="minPoolSize">
    <value>1</value>
</property>
<!-- 设置连接池的初始容量-->
<property name="initialPoolSize">
    <value>1</value>
</property>
<!-- 设置最大空闲时间-->
<property name="maxIdleTime">
    <value>20</value>
</property>
</bean>
<!--定义了 Hibernate 的 SessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- 指定 SessionFactory 的 DataSource-->
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
    <!-- 加载全部的映射文件-->
    <property name="mappingResources">
        <list>
            <value>Employee.hbm.xml</value>
            <value>Attend.hbm.xml</value>
            <value>AttendType.hbm.xml</value>
            <value>Application.hbm.xml</value>
            <value>CheckBack.hbm.xml</value>
            <value>Payment.hbm.xml</value>
        </list>
    </property>
    <!-- 配置 Hibernate 的属性-->
    <property name="hibernateProperties">
        <props>
            <!-- 配置 Hibernate 的数据库方法-->
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <!-- 程序运行过程中，显示 Hibernate 转换的 SQL 语句-->
```

```
<prop key="show_sql">true</prop>
<!-- 是否根据映射文件建表-->
<prop key="hibernate.hbm2ddl.auto">update</prop>
<!-- 配置 Hibernate 的批处理量 -->
<prop key="hibernate.jdbc.batch_size">20</prop>
</props>
</property>
</bean>
```

**注意：**Hibernate 属性可以直接在 LocalSessionFactoryBean bean 内配置，也可以在 hibernate.cfg.xml 文件中配置。

## 2. 配置 DAO 组件

所有继承 HibernateDaoSupport 的 DAO 实现类，必须为其提供 SessionFactory 的引用。由于所有 DAO 组件都需要注入 SessionFactory 引用，因此可以使用 bean 继承简化 DAO 组件的配置。本应用同样将所有的 DAO 组件配置在单独的配置文件中，下面是 DAO 对象的配置：

**注意：**配置 DAO 组件只能使用实现类，不能使用接口。

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的根元素，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 配置 DAO 组件模板，用于被其他 DAO 组件继承-->
    <bean id="daoTemplate" abstract="true" lazy-init="true">
        <property name="sessionFactory">
            <!-- 因为 sessionFactory 在另一个配置文件中，
                因此使用 ref bean 来注入，不可使用 ref local 注入。-->
            <ref bean="sessionFactory"/>
        </property>
    </bean>
    <!-- 配置第一个 DAO 组件： employeeDao-->
    <bean id="employeeDao" class="org.yeeku.dao.EmployeeDaoHibernate"
        parent="daoTemplate"/>
    <!-- 配置第二个 DAO 组件： managerDao -->
    <bean id="managerDao" class="org.yeeku.dao.ManagerDaoHibernate" parent=
        "daoTemplate"/>
    <!-- 配置第三个 DAO 组件： attendDao -->
    <bean id="attendDao" class="org.yeeku.dao.AttendDaoHibernate" parent=
        "daoTemplate"/>
    <!-- 配置第四个 DAO 组件： attendTypeDao -->
    <bean id="attendTypeDao" class="org.yeeku.dao.AttendTypeDaoHibernate"
        parent="daoTemplate"/>
    <!-- 配置第五个 DAO 组件： appDao -->
    <bean id="appDao" class="org.yeeku.dao.ApplicationDaoHibernate"
        parent="daoTemplate"/>
    <!-- 配置第六个 DAO 组件： checkDao -->
    <bean id="checkDao" class="org.yeeku.dao.CheckBackDaoHibernate"
        parent="daoTemplate"/>
    <!-- 配置第七个 DAO 组件： payDao -->
    <bean id="payDao" class="org.yeeku.dao.PaymentDaoHibernate"
        parent="daoTemplate"/>
</beans>
```

注意：系统的 DAO 实现类中并未提供 setSessionFactory 方法，该方法由其父类 HibernateDaoSupport 提供，用于依赖注入 SessionFactory 引用。该配置文件中也并未配置 SessionFactory bean，其中 DataSource 和 SessionFactory 的配置在另一个文件中。

## 10.4 实现 Service 层

本系统只使用了两个业务逻辑组件，分别对应系统中包含的两个模块：Manager 和 Employee 模块。这两个模块分别使用不同的 Service 组件，每个组件 Facade 7 个 DAO 组件，系统使用这两个业务逻辑组件将这些 DAO 对象封装在一起。

### 10.4.1 Service 组件设计

Service 组件采用正面模式封装多个 DAO 组件，DAO 对象与 Service 组件之间的关系如图 10.3 所示。

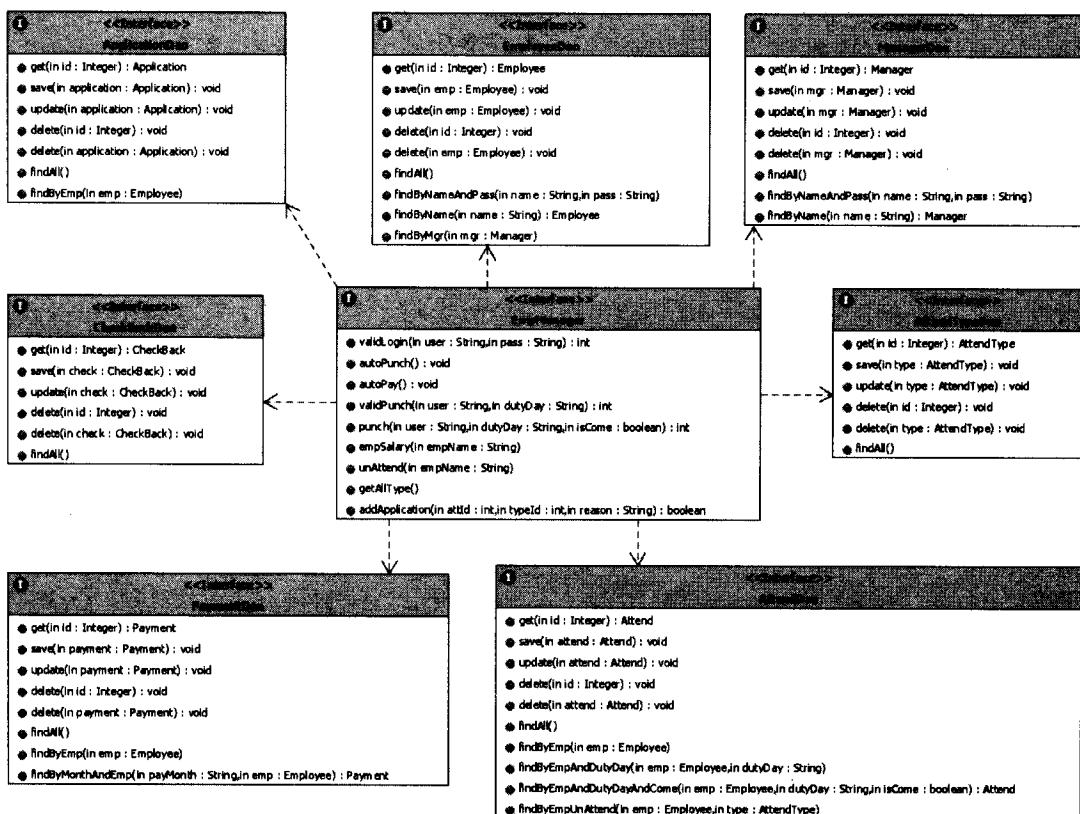


图 10.3 EmpManager 与 DAO 组件接口的类图

在 EmpManager 接口里定义了大量的业务方法，这些方法的实现依赖于 DAO 组件。由于每个业务方法要涉及到多个 DAO 操作，其 DAO 操作是单个的数据记录的操作，而业务逻辑方法的访问，则需要设计多个 DAO 操作，因此每个业务逻辑方法可能需要涉及多条记录的访问。

业务逻辑组件面向 DAO 接口编程，可让业务逻辑组件从 DAO 组件的实现中分离。因此业务逻辑组件只关心业务逻辑的实现，无须关心数据访问逻辑的实现。

## 10.4.2 Service 组件的实现

Service 组件需要实现的业务方法主要取决于业务的需要，通常需要在业务组件中包含对应的方法。

### 1. 业务层组件的实现

业务层组件与具体的数据库访问技术分离，使所有的数据库访问依赖于 DAO 组件，下面是 EmpManagerImpl 的源代码。

```
public class EmpManagerImpl implements EmpManager
{
    //下面是本业务逻辑组件所依赖的 7 个业务逻辑组件
    private ApplicationDao appDao;
    private AttendDao attendDao;
    private AttendTypeDao typeDao;
    private CheckBackDao checkDao;
    private EmployeeDao empDao;
    private ManagerDao mgrDao;
    private PaymentDao payDao;
    //格式化日期所使用的 DateFormat 对象
    private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM");
    //下面是依赖注入 7 个业务逻辑组件的 setter 方法
    public void setAppDao(ApplicationDao appDao)
    {
        this.appDao = appDao;
    }
    public void setAttendDao(AttendDao attendDao)
    {
        this.attendDao = attendDao;
    }
    public void setTypeDao(AttendTypeDao typeDao)
    {
        this.typeDao = typeDao;
    }
    public void setCheckDao(CheckBackDao checkDao)
    {
        this.checkDao = checkDao;
    }
    public void setEmpDao(EmployeeDao empDao)
    {
        this.empDao = empDao;
    }
    public void setMgrDao(ManagerDao mgrDao)
    {
        this.mgrDao = mgrDao;
    }
}
```

```
}

public void setPayDao(PaymentDao payDao)
{
    this.payDao = payDao;
}

/**
 * 验证登录
 * @param user 登录用的用户名
 * @param pass 登录用的密码
 * @return 登录后的身份确认:0 为登录失败, 1 为登录 emp 2 为登录 mgr
 */
public int validLogin(String user , String pass)
{
    if (mgrDao.findByNameAndPass(user,pass).size() >= 1)
    {
        return LOGIN_MGR;
    }
    else if (empDao.findByNameAndPass(user,pass).size() >= 1)
    {
        return LOGIN_EMP;
    }
    else
    {
        return LOGIN_FAIL;
    }
}
/**
 * 自动打卡, 周一到周五, 早上 7: 00 为每个员工打入旷工记录
 */
public void autoPunch()
{
    System.out.println("自动插入旷工记录");
    List<Employee> emps = empDao.findAll();
    String dutyDay = new java.sql.Date(System.currentTimeMillis()).
        toString();
    for (Employee e : emps)
    {
        AttendType atype = typeDao.get(6);
        Attend a = new Attend();
        a.setDutyDay(dutyDay);
        a.setType(atype);
        if (Calendar.getInstance().get(Calendar.HOUR_OF_DAY) < 11)
        {
            a.setIsCome(true);
        }
        a.setIsCome(false);
        a.setEmployee(e);
        attendDao.save(a);
    }
}
/**
 * 自动结算工资, 每月 1 日凌晨 2 点将调用该作业
 */
public void autoPay()
{
    System.out.println("自动插入工资结算");
    List<Employee> emps = empDao.findAll();
    //获取上个月时间
    Calendar c = Calendar.getInstance();
```

```
c.add(Calendar.DAY_OF_MONTH, -15);
String payMonth = sdf.format(c.getTime());
for (Employee e : emps)
{
    Payment pay = new Payment();
    List<Attend> attends = attendDao.findByEmp(e);
    double amount = e.getSalary();
    for (Attend a : attends)
    {
        amount += a.getType().getAmerce();
    }
    pay.setPayMonth(payMonth);
    pay.setEmployee(e);
    pay.setAmount(amount);
    payDao.save(pay);
}
/***
 * 验证某个员工是否可打卡
 * @param user 员工名
 * @param dutyDay 日期
 * @return 可打卡的类别
 */
public int validPunch(String user, String dutyDay)
{
    //不能查找到对应用户，返回无法打卡
    Employee emp = empDao.findByName(user);
    if (emp == null)
    {
        return NO_PUNCH;
    }
    List<Attend> attends = attendDao.findByEmpAndDutyDay(emp, dutyDay);
    //系统没有为用户在当天插入空打卡记录，无法打卡
    if (attends == null)
    {
        return NO_PUNCH;
    }
    //可以上班、下班打卡
    if (attends.size() <= 0)
    {
        return NO_PUNCH;
    }
    else if (attends.size() == 1 && attends.get(0).getIsCome()
        && attends.get(0).getPunchTime() == null)
    {
        return COME_PUNCH;
    }
    else if (attends.size() == 1 && attends.get(0).getPunchTime() == null)
    {
        return LEAVE_PUNCH;
    }
    else if (attends.size() == 2)
    {
        if (attends.get(0).getPunchTime() == null && attends.get(1).
            getPunchTime() == null)
        {
            return BOTH_PUNCH;
        }
        else if (attends.get(1).getPunchTime() == null)
```

```
{  
    return LEAVE_PUNCH;  
}  
else  
{  
    return NO_PUNCH;  
}  
}  
return NO_PUNCH;  
}  
/**  
 * 打卡  
 * @param user 员工名  
 * @param dutyDay 打卡日期  
 * @param isCome 是否是上班打卡  
 * @return 打卡结果  
 */  
public int punch(String user , String dutyDay , boolean isCome)  
{  
    Employee emp = empDao.findByName(user);  
    if (emp == null)  
    {  
        return PUNCH_FAIL;  
    }  
    Attend attend = attendDao.findByEmpAndDutyDayAndCome(emp , dutyDay ,  
    isCome);  
    if (attend == null)  
    {  
        return PUNCH_FAIL;  
    }  
    if (attend.getPunchTime() != null)  
    {  
        return PUNCHED;  
    }  
    int punchHour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);  
    attend.setPunchTime(new Date());  
    //上班打卡  
    if (isCome)  
    {  
        // 9 点之前算正常  
        if (punchHour < 9)  
        {  
            attend.setType(typeDao.get(1));  
        }  
        // 9~11 点之间算迟到  
        else if (punchHour < 11)  
        {  
            attend.setType(typeDao.get(4));  
        }  
        //11 点之后算旷工  
    }  
    //下班打卡  
    else  
    {  
        // 6 点之后算正常  
        if (punchHour > 6)  
        {  
            attend.setType(typeDao.get(1));  
        }  
    }  
}
```

```

        // 4~6 点之间算早退
        else if (punchHour < 4)
        {
            attend.setType(typeDao.get(5));
        }
    }
    attendDao.update(attend);
    return PUNCH_SUCC;
}
/***
 * 员工浏览自己的工资
 * @param empName 员工名
 * @return 该员工的工资列表
 */
public List<PaymentBean> empSalary(String empName)
{
    Employee emp = empDao.findByName(empName);
    List<Payment> pays = payDao.findByEmp(emp);
    List<PaymentBean> result = new ArrayList<PaymentBean>();
    for (Payment p : pays )
    {
        result.add(new PaymentBean(p.getPayMonth(),p.getAmount()));
    }
    return result;
}
/***
 * 员工查看自己的最近三天非正常打卡
 * @param empName 员工名
 * @return 该员工最近三天的非正常打卡
 */
public List<AttendBean> unAttend(String empName)
{
    AttendType type = typeDao.get(1);
    Employee emp = empDao.findByName(empName);
    List<Attend> attends = attendDao.findByEmpUnAttend(emp, type);
    List<AttendBean> result = new ArrayList<AttendBean>();
    for (Attend att : attends )
    {
        result.add(new AttendBean(att.getId() , att.getDutyDay() ,
        att.getType().getName() ,
        att.getPunchTime()));
    }
    return result;
}

/***
 * 返回全部的出勤类别
 * @return 全部的出勤类别
 */
public List<AttendType> getAllType()
{
    return typeDao.findAll();
}
/***
 * 添加申请
 * @param attId 申请的出勤 id
 * @param typeId 申请的类别 id
 * @param reason 申请的理由
 * @return 添加的结果
 */

```

```

    */
public boolean addApplication(int attId , int typeId , String reason)
{
    try
    {
        Application app = new Application();
        Attend attend = attendDao.get(attId);
        AttendType type = typeDao.get(typeId);
        app.setAttend(attend);
        app.setType(type);
        if (reason != null)
        {
            app.setReason(reason);
        }
        appDao.save(app);
        return true;
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return false;
}
}

```

在上面的业务逻辑组件中，有 autoPunch 和 autoPay 两个方法，这两个方法并不由客户端直接调用，而是由任务调度来执行，负责每天为员工完成自动考勤，以及每月 1 日为所有员工完成工资结算。

## 2. 事务管理

事务管理将推迟到 Service 组件而不是 DAO 组件，因为只有对业务逻辑方法添加事务才有实际的意义，对于单个 DAO 方法（基本的 CRUD 方法）增加事务操作是没有太实际意义的。

关于事务属性的配置，建议直接使用 Spring 的 AOP 框架生成事务代理，而不要使用 Spring 提供的 TransactionProxyFactoryBean 配置事务代理。

采用 AOP 框架配置事务代理主要有两个类。

- BeanNameAutoProxyCreator。
- DefaultAdvisorAutoProxyCreator。

这两个配置方式都可避免增量式的配置，不必为每个目标对象配置代理 bean；避免了目标对象被直接调用。

下面是 BeanNameAutoProxyCreator 配置事务代理的代码：

```

<!-- 定义 BeanNameAutoProxyCreator-->
<bean class="org.springframework.aop.framework.autoproxy.
BeanNameAutoProxyCreator">
    <!-- 指定对满足哪些 bean name 的 bean 自动生成业务代理 -->
    <property name="beanNames">
        <!-- 下面是所有需要自动创建事务代理的 bean-->
        <list>
            <value>MgrManager</value>
            <value>EmpManager</value>
        </list>

```

```
<!-- 此处可增加其他需要自动创建事务代理的 bean-->
</property>
<!-- 下面定义 BeanNameAutoProxyCreator 所需的事务拦截器-->
<property name="interceptorNames">
    <list>
        <!-- 此处可增加其他新的 Interceptor -->
        <value>transactionInterceptor</value>
    </list>
</property>
</bean>
```

### 3. 部署业务层组件

单独配置系统的业务逻辑层，可避免因配置文件过大引起配置文件难以阅读。将配置文件按层和模块分开配置，可以提高 Spring 配置文件的可读性和可理解性。

在 applicationContext.xml 配置文件中配置数据源、事务管理器、业务逻辑组件和事务管理器等 bean。具体的配置文件如下：

```
<?xml version="1.0" encoding="gb2312"?>
<!-- Spring 配置文件的文件头，包含 DTD 等信息-->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 配置数据源，采用 C3P0 的数据源-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
 destroy-method="close">
        <!-- 指定驱动-->
        <property name="driverClass">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <!-- 指定数据库服务的 URL-->
        <property name="jdbcUrl">
            <value>jdbc:mysql://localhost/auction</value>
        </property>
        <!-- 配置数据库的用户名-->
        <property name="user">
            <value>root</value>
        </property>
        <!-- 配置数据库的密码-->
        <property name="password">
            <value>32147</value>
        </property>
        <!-- 配置连接池的最大容量-->
        <property name="maxPoolSize">
            <value>40</value>
        </property>
        <!-- 配置连接池的最小容量-->
        <property name="minPoolSize">
            <value>1</value>
        </property>
        <!-- 配置连接池的初始容量-->
        <property name="initialPoolSize">
            <value>1</value>
        </property>
        <property name="maxIdleTime">
            <value>20</value>
        </property>
    </bean>
```

```

<!-- 定义了 Hibernate 的 SessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
    <!-- 由 Spring 依赖注入 DataSource-->
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
    <!-- 列出全部映射文件-->
    <property name="mappingResources">
        <list>
            <value>Employee.hbm.xml</value>
            <value>Attend.hbm.xml</value>
            <value>AttendType.hbm.xml</value>
            <value>Application.hbm.xml</value>
            <value>CheckBack.hbm.xml</value>
            <value>Payment.hbm.xml</value>
        </list>
    </property>
    <!-- 配置 Hibernate 属性-->
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.
MySQLDialect</prop>
            <prop key="show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.jdbc.batch_size">20</prop>
        </props>
    </property>
</bean>
<!-- 配置事务管理器-->
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransaction
Manager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- 配置事务拦截器-->
<bean id="transactionInterceptor"
      class="org.springframework.transaction.interceptor.Transaction
Interceptor">
    <!-- 事务拦截器 bean 需要依赖注入一个事务管理器 -->
    <property name="transactionManager" ref="transactionManager"/>
    <!-- 定义事务传播属性-->
    <property name="transactionAttributes">
        <!-- 下面定义事务传播属性-->
        <props>
            <prop key="get*>">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="*>">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
<!-- 利用继承简化业务逻辑组件的配置，业务组建 bean 的模板-->
<bean id="managerTemplate" abstract="true" lazy-init="true">
    <!-- 依赖注入 ApplicationDao 的实例-->
    <property name="appDao">
        <ref bean="appDao"/>
    </property>
    <!-- 依赖注入 AttendDao 的实例-->
    <property name="attendDao">
        <ref bean="attendDao"/>

```

```
</property>
<!-- 依赖注入 AttendTypeDao 的实例-->
<property name="typeDao">
    <ref bean="attendTypeDao"/>
</property>
<!-- 依赖注入 CheckBackDao 的实例-->
<property name="checkDao">
    <ref bean="checkDao"/>
</property>
<!-- 依赖注入 EmployeeDao 的实例-->
<property name="empDao">
    <ref bean="employeeDao"/>
</property>
<!-- 依赖注入 ManagerDao 的实例-->
<property name="mgrDao">
    <ref bean="managerDao"/>
</property>
<!-- 依赖注入 PaymentDao 的实例-->
<property name="payDao">
    <ref bean="payDao"/>
</property>
</bean>
<!-- 定义 mgrManager bean, 以 managerTemplate 作为父 bean-->
<bean id="mgrManager" class="org.yeeku.service.MgrManagerImpl"
parent="managerTemplate"/>
<!-- 定义 empManager bean, 以 managerTemplate 作为父 bean-->
<bean id="empManager" class="org.yeeku.service.EmpManagerImpl"
parent="managerTemplate"/>
<!-- 定义 BeanNameAutoProxyCreator-->
<bean class="org.springframework.aop.framework.autoproxy.
BeanNameAutoProxyCreator">
    <!-- 指定对满足哪些 bean name 的 bean 自动生成业务代理 -->
    <property name="beanNames">
        <!-- 下面是所有需要自动创建事务代理的 bean-->
        <list>
            <value>MgrManager</value>
            <value>EmpManager</value>
        </list>
        <!-- 此处可增加其他需要自动创建事务代理的 bean-->
    </property>
    <!-- 下面定义 BeanNameAutoProxyCreator 所需的事务拦截器-->
    <property name="interceptorNames">
        <list>
            <!-- 此处可增加其他新的 Interceptor -->
            <value>transactionInterceptor</value>
        </list>
    </property>
</bean>
<!-- 定义任务调度 bean: 每月的工资结算-->
<bean id="cronTriggerPay" class="org.springframework.scheduling.quartz.
CronTriggerBean">
    <!-- 采用嵌套 bean 定义 JobDetailBean -->
    <property name="jobDetail">
        <bean class="org.springframework.scheduling.quartz.
JobDetailBean">
            <!-- 确定作业的实现类-->
            <property name="jobClass">
                <value>org.yeeku.schedule.PayJob</value>
            </property>
```

```

<!-- 为作业实现类注入依赖关系-->
<property name="jobDataAsMap">
    <map>
        <entry key="empMgr">
            <ref local="empManager"/>
        </entry>
    </map>
</property>
</bean>
</property>
<!-- 确定 Cron 表达式，每月的 1 日的凌晨 2 点调度任务-->
<property name="cronExpression">
    <value>0 0 2 1 * ?*</value>
</property>
</bean>
<!-- 定义任务调度 bean：每天的系统自动考勤-->
<bean id="cronTriggerPunch" class="org.springframework.scheduling.
quartz.CronTriggerBean">
    <!-- 采用嵌套 bean 定义 JobDetailBean -->
    <property name="jobDetail">
        <bean class="org.springframework.scheduling.quartz. JobDetailBean">
            <!-- 确定作业的实现类-->
            <property name="jobClass">
                <value>org.yeeku.schedule.PunchJob</value>
            </property>
            <!-- 为作业实现类注入依赖关系-->
            <property name="jobDataAsMap">
                <map>
                    <entry key="empMgr">
                        <ref local="empManager"/>
                    </entry>
                </map>
            </property>
        </bean>
    </property>
    <!-- 确定 Cron 表达式，星期一到星期五的 7 点和 12 点分别调度该任务-->
    <property name="cronExpression">
        <value>0 0 7,12 ? * MON-FRI</value>
    </property>
</bean>
<!-- 调度作业-->
<bean class="org.springframework.scheduling.quartz.Scheduler
FactoryBean">
    <property name="triggers">
        <list>
            <ref local="cronTriggerPunch"/>
            <ref local="cronTriggerPay"/>
        </list>
    </property>
</bean>
</beans>

```

由此可看出，通过配置文件来设置各种组件依赖，并由容器管理其依赖，可提高系统的解耦。

**注意：**采用 BeanNameAutoProxyCreator 配置事务代理时，无须为每个目标额外配置多余的事務代理，可由系统自动为其创建事務代理。事实上，BeanNameAutoProxyCreator

并不是专用于创建事务代理，它可以根据配置的拦截器列表生成代理。在该系统中，只指定了一个 transactionInterceptor，该拦截器是事务拦截器，由 Spring 提供实现。该拦截器也可以由用户提供，在后面的表现层组件中可以看到用户自定义的授权拦截器。

## 10.5 任务调度的实现

系统中常常有些需要自动执行的任务，这些任务可能每隔一段时间就要执行一次，也可能需要在固定的时间自动执行，这些任务的自动执行必须使用任务的自动调度。

JDK 为简单的任务调度提供了 Timer 支持，但对于更复杂的调度，例如，需要在某个特定时刻调度任务时，Timer 就有点力不从心了。好在有另一个开源框架 Quartz，借助于它的支持，既可以实现简单的任务调度，也可以执行复杂的任务调度。

### 10.5.1 Quartz 的使用

Quartz 是一个以 Java 实现的开源任务调度框架，也是个简单、易用的任务调度系统，借助于 Cron 表达式，Quartz 可以支持复杂的任务调度。

#### 1. 安装 Quartz

Quartz 下载和安装请按如下步骤进行。

(1) 登录 <http://www.opensymphony.com/quartz/download.action> 站点，下载 Quartz 的最新版本。目前，Quartz 的最新版为 1.5.2，笔者的示例程序基于该版本完成。将下载到的压缩文件解压缩，发现有如下的文件结构。

- quartz-1.5.2.jar: Quartz 的核心类库。
- quartz-all-1.5.2.jar: Quartz 的完全类库，包括核心类库及其他可选类库。
- quartz-jboss-1.5.2.jar: 可选的与 Jboss 相关的 Quartz 类库。
- quartz-oracle-1.5.2.jar: 可选的与 Oracle 相关的 Quartz 类库。
- quartz-weblogic-1.5.2.jar: 可选的与 Weblogic 相关的 Quartz 类库。
- docs: 存放 Quartz 的相关文档，包括 API 等文档。
- examples: 存放 Quartz 的示例程序。
- lib: 存放 Quartz 编译或运行所依赖的二进制数值包。
- src: 存放 Quartz 的源文件。
- 其他 Quartz 相关说明文档。

(2) 将 quartz-1.5.2.jar 文件增加到 JDK 的 CLASSPATH 里，或者将 quartz-all-1.5.2.jar 增加到 JDK 的 CLASSPATH 里，都可完成 Quartz 的安装。

(3) 如果需要在 Web 应用中使用 Quartz，则应将 quartz-1.5.2.jar 或 quartz-all-1.5.2.jar 文件复制到 WEB-INF/lib 路径下。

#### 2. Quartz 运行的基本属性

Quartz 有一个 quartz.properties 的配置文件，通过该配置文件，可以修改框架运行时

的环境。默认使用 Quartz.jar 里面的 quartz.properties 文件。大部分时候，应该创建一个 quartz.properties 文件，并将它放在 classes 目录中以便 ClassLoader 找到它。下面是 quartz.properties 文件的示例：

```
# 配置主调度器属性
org.quartz.scheduler.instanceName = QuartzScheduler
org.quartz.scheduler.instanceId = AUTO
# 配置线程池
# Quartz 线程池的实现类
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
# 线程池的线程数量
org.quartz.threadPool.threadCount = 5
# 线程池里线程的优先级
org.quartz.threadPool.threadPriority = 5
# 配置作业存储
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

### 3. Quartz 里的作业

作业是一个执行任务的 Java 类，可以是任何 Java 代码。只需实现 org.quartz.Job 接口即可，Job 接口包含一个方法 execute()，execute 方法体是被调度的作业体。一旦实现了 Job 接口和 execute()方法，当 Quartz 确定该作业运行时，它将调用 execute()方法体。下面是一些在作业里面完成某种任务的例子：

Quartz 提供两种基本作业存储方式。

- 第一种类型叫做 RAMJobStore，它利用内存来持久化调度程序信息。这种作业存储类型最容易配置和运行。对许多应用来说，这种存储方式已经足够了。然而，由于调度程序信息保存在 JVM 的内存里面，因此，一旦应用程序中止，则所有的调度信息将被丢失。
- 第二种类型称为 JDBC 作业存储。需要 JDBC 驱动程序和后台数据库保存调度程序的信息，由需要调度程序维护调度信息的用户来设计。

下面是一个简单的作业：

```
//Quartz 里的作业应该实现 Job 接口
public class TestJob implements Job
{
    //判断作业是否执行的旗标
    private boolean isRunning = false;
    //实现 Job 接口必须实现的方法
    public void execute(JobExecutionContext context)
        throws JobExecutionException
    {
        //如果作业没有被调度
        if (!isRunning)
        {
            //设置旗标为 true，标识作业已被调度
            isRunning = true;
            System.out.println(new Date() + " 作业被调度。");
            //作业体就是循环打印出 100 个数字
            for (int i = 0; i < 100; i++)
            {
                System.out.println("作业完成" + (i + 1) + "%");
```

```
        }
        System.out.println(new Date() + "    作业调度结束。");
        //修改旗标为 false, 标识作业没有运行
        isRunning = false;
    }
    //如果作业正在运行, 即使获得调度, 也立即退出
    else
    {
        System.out.println(new Date() + "任务退出");
    }
}
}
```

注意：上面使用了旗标来控制作业，使作业不会被重复执行。

#### 4. Quartz 里的触发器

Quartz 的设计允许作业与作业调度分离。将作业与作业调度的分离通过触发器完成，Quartz 中的触发器用来确定作业的触发时机，其框架提供了一系列触发器类型，但以下两个是最常用的：

- SimpleTrigger。
- CronTrigger。

SimpleTrigger 主要用于简单的调度。例如，如果需要在给定的时间内重复执行作业，或者间隔固定时间执行作业，可以选择 SimpleTrigger。SimpleTrigger 类似与 JDK 的 Timer。

如果需要更复杂的作业调度，则可以考虑使用 CronTrigger。该调度器基于 Calendar-like 调度。当需要在除星期六和星期日以外的每天上午 10: 30 执行作业时，就应使用 CronTrigger。CronTrigger 是基于 Unix Cron 的表达式。

Cron 表达式是一个字符串，字符串以 5 个或 6 个空格隔开，分成 6 个或 7 个域，每个域代表一个时间域，Cron 表达式有如下两种语法格式。

Seconds Minutes Hours DayofMonth Month DayofWeek Year。

上面是包含 7 个域的表达式，还有只包含 6 个域的 Cron 表达式。

Seconds Minutes Hours DayofMonth Month DayofWeek

每个域可出现的字符如下。

- Second: 可出现, - \* / 四个字符, 有效范围为 0~59 的整数。
- Minutes: 可出现, - \* / 四个字符, 有效范围为 0~59 的整数。
- Hours: 可出现, - \* / 四个字符, 有效范围为 0~23 的整数。
- DayofMonth: 可出现, - \* ? / L W C 八个字符, 有效范围为 1~31 的整数。
- Month: 可出现, - \* / 四个字符, 有效范围为 1~12 或 JAN-DEC。
- DayofWeek: 可出现, - \* ? / L C # 八个字符, 有效范围为 1~7 或 SUN-SAT 两个范围。其中 1 表示星期日, 2 表示星期一, 依次类推。
- Year: 可出现, - \* / 四个字符, 有效范围为 1970~2099 年。

每个域通常都使用数字，但还可以出现如下特殊字符，它们的含义如下。

- \* : 表示匹配该域的任意值，假如在 Minutes 域使用\*，即表示在每分钟都会触发事件。

- ? : 只能用在 DayofMonth 和 DayofWeek 两个域。它也匹配该域的任意值，但实际应用中则不会，因为 DayofMonth 和 DayofWeek 会互相影响。例如，想在每月的 20 日触发调度，无论 20 日是星期几，则只能使用如下写法：13 13 15 20 \* ?，其中最后一位只能使用?，而不能使用\*，如果使用\*则表示无论星期几都会触发，但实际并不是这样。
- : 表示范围。例如，在 Minutes 域使用 5-20，表示从 5 分钟到 20 分钟内每分钟触发一次。
- / : 表示从起始时间开始触发，然后每隔固定时间触发一次。例如，在 Minutes 域使用 5/20，则意味着 5 分钟触发一次，而在 25, 45 等分钟时分别触发一次。
- , : 表示列出枚举值。例如，在 Minutes 域使用 5, 20，则意味着在 5 和 20 分钟分别触发一次。
- L : 表示最后。只能出现在 DayofWeek 和 DayofMonth 域，如果在 DayofWeek 域使用 5L，则意味着在最后一个星期四触发。
- W : 表示有效工作日（星期一到星期五）。只能出现在 DayofMonth 域，系统将在离指定日期最近的有效工作日触发事件。例如，在 DayofMonth 使用 5W，如果 5 日是星期六，则将在最近的工作日星期一，即 4 日触发。如果 5 日是星期一到星期五中的一天，则就在 5 日触发。须注意的是，W 不会跨月寻找，例如，1W，1 日恰好是星期六，系统不会在上月的最后一天触发，而是到 3 日触发。
- LW : 这两个字符可以连接使用，表示某个月最后一个工作日，即最后一个星期五。
- # : 用于确定每个月的第几个星期几，只能出现在 DayofMonth 域。如在 4 # 5，表示某月的第五个星期三。

下面的 Quartz Cron 表达式，表示在周一到周五的每天上午 10: 15 分开始执行作业。

```
0 15 10 ? * MON-FRI
```

下面的表达式，表示在 2002—2005 年中的每个月的最后一个星期五上午 10: 15 分开始执行作业。

```
0 15 10 ? * 6L 2002-2005
```

## 5. Quartz 里的调度器

调度器用于将作业与触发器关联，一个作业可关联多个触发器，一个触发器也可用于控制多个作业，Quartz 的调度器由 Scheduler 接口体现。该接口声明了如下方法。

- void addJob(JobDetail jobDetail, boolean replace): 将给定的 JobDetail 实例添加到调度器里。
- Date scheduleJob(JobDetail jobDetail, Trigger trigger): 将添加指定的 JobDetail 实例，添加给定的 trigger 与之关联。
- Date scheduleJob(Trigger trigger): 添加触发器 trigger 来调度作业。

将上面的作业与触发器关联起来，其程序如下：

```
public class MyQuartzServer
{
```

```

public static void main(String[] args)
{
    MyQuartzServer server = new MyQuartzServer();
    try
    {
        //启动调度
        server.startScheduler();
    }
    catch (SchedulerException ex)
    {
        ex.printStackTrace();
    }
}
//调度作业的方法
protected void startScheduler() throws SchedulerException
{
    //使用调度器工厂创建调度器实例
    Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();
    //以作业创建 JobDetail 实例
    JobDetail jobDetail = new JobDetail("dd", Scheduler.DEFAULT_GROUP,
    TestJob.class);
    //创建 trigger
    Trigger trigger = new SimpleTrigger("dd", Scheduler.DEFAULT_GROUP,
    5000, 200);
    //调度器将作业与 trigger 关联起来
    scheduler.scheduleJob(jobDetail, trigger );
    //开始调度
    scheduler.start();
}
}

```

注意：使用 JobDetail 包装一个作业，在包装时，包括给作业命名，以及指定作业所在的组。

## 10.5.2 在 Spring 中使用 Quartz

Spring 的任务调度抽象层简化了任务调度，在 Quartz 基础上提供了更好的调度抽象。本系统使用 Quartz 框架来完成任务调度，创建 Quartz 的作业 bean 有以下两个方法：

- 利用 JobDetailBean 包装 QuartzJobBean 的子类。
- 利用 MethodInvokingJobDetailFactoryBean 工厂 bean 包装普通 JavaBean。

采用两种方法都可创建一个 Quartz 所需的 JobDetailBean，包含需要被调度的作业体。继承 QuartzJobBean 类时，必须实现该类的一个抽象方法。

`executeInternal(JobExecutionContext ctx):` 被调度任务的执行体。

如果采用 MethodInvokingJobDetailFactoryBean 包装，则无须继承任何父类，直接使用配置即可。配置 MethodInvokingJobDetailFactoryBean，需要指定以下两个元素。

- targetObject: 指定任务体的实现类。
- targetMethod: 指定任务体的执行方法。

采用 JobDetailBean 实现作业 bean 的配置格式如下：

```
<!-- 定义 JobDetailBean bean-->
```

```
<bean name="quartzDetail" class="org.springframework.scheduling.quartz.
JobDetailBean">
    <property name="jobClass">
        <value>QuartzJobBean 的子类</value>
    </property>
</bean>
```

如果采用 MethodInvokingJobDetailFactoryBean 包装，格式如下：

```
<!-- 定义目标 bean-->
<bean id="testQuartz" class="lee.TestQuartz"/>
<!-- 定义 JobDetailBean bean-->
<bean id="quartzDetail"
    class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFa
ctoryBean">
    <!-- 指定目标 object-->
    <property name="targetObject">
        <ref bean="testQuartz"/>
    </property>
    <!-- 指定目标方法-->
    <property name="targetMethod">
        <value>test</value>
    </property>
</bean>
```

完成这两步后，在后面只需要以下两个步骤即可完成任务的调度。

(1) 使用 SimpleTriggerBean 创建简单触发器，或使用 CronTriggerBean 创建基于 Cron 表达式的触发器。

(2) 使用 SchedulerFactoryBean 调度作业。

下面是介绍本系统中两个任务调度的作业类。

第一个考勤作业：PunchJob。系统每天为员工自动考勤两次，并加入旷工考勤，然后根据员工的考勤时间转换成实际的考勤类型。

```
//继承 QuartzJobBean 实现作业类
public class PunchJob extends QuartzJobBean
{
    private EmpManager empMgr;
    //依赖注入 EmpManager 的 setter 方法
    public void setEmpMgr(EmpManager empMgr)
    {
        this.empMgr = empMgr;
    }
    private boolean isRunning = false;
    //重写 executeInternal 方法，定义任务执行体
    public void executeInternal(JobExecutionContext ctx)
        throws JobExecutionException
    {
        if (!isRunning)
        {
            System.out.println("开始调度自动打卡");
            isRunning = true;
            empMgr.autoPunch();
            isRunning = false;
        }
    }
}
```

第二个是工资结算作业：PayJob。该作业在每月 1 日自动结算员工上个月的工资。

```
public class PayJob extends QuartzJobBean
{
    private EmpManager empMgr;
    public void setEmpMgr(EmpManager empMgr)
    {
        this.empMgr = empMgr;
    }
    private boolean isRunning;
    public void executeInternal(JobExecutionContext ctx)
        throws JobExecutionException
    {
        if (!isRunning)
        {
            System.out.println("开始调度自动结算工资");
            isRunning = true;
            empMgr.autoPay();
            isRunning = false;
        }
    }
}
```

## 10.6 MVC 层实现

本系统的 MVC 框架使用 Struts。因为，Struts 是最健壮，应用最广的 MVC 框架。系统使用 Struts 框架可以提高系统的可控制性，保证了系统的稳定性及可用性。

### 10.6.1 解决中文编码

本章示例的中文编码并不是采用 Filter 解决，而是使用扩展 ActionServlet 来解决这个问题。

因为系统的所有请求，包括超级链接和表单处理地址，都采用\*.do 的模式。所有的请求都将被 ActionServlet 拦截，所以解决中文编码可考虑在 ActionServlet 解析参数时，设置其解码方式。

为了解决中文编码问题，系统对 ActionServlet 进行简单扩展：

```
public class MyActionServlet extends org.apache.struts.action.ActionServlet
{
    protected void process(HttpServletRequest request,
                          HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException
    {
        // 设置中文编码方式
        request.setCharacterEncoding("GB2312");
        // 调用父类的方法
        super.process(request, response);
    }
}
```

为让该 ActionServlet 拦截所有客户端请求，必须在 web.xml 文件中增加如下配置：

```

<!-- 配置 ActionServlet 的核心 servlet-->
<servlet>
    <servlet-name>action</servlet-name>
    <!-- 使用用户扩展的 ActionServlet-->
    <servlet-class>org.yeeku.struts.MyActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<!-- 映射所有以.do 结尾的请求-->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

完成上面配置后，所有客户端的\*.do 请求都将由 MyActionServlet 负责处理。

## 10.6.2 Struts 与 Spring 的整合

整合 Struts 与 Spring 只有一个要求，让 Struts 将拦截到客户端的请求转发给 Spring 容器中的 bean。

本系统使用 DelegatingRequestProcessor 的整合策略，使用 DelegatingRequestProcessor 的整合策略可避免创建过多的 DelegatingActionProxy 实例，提前将请求转发到 Spring 容器内的 bean。

关于 Struts 与 Spring 整合的详细情况请参阅第 7 章。

为了在应用启动时由 Struts 负责创建 Spring 容器，应在 struts-config.xml 文件中增加如下配置：

```

<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <!-- 下面列出了全部的配置文件-->
    <set-property property="contextConfigLocation"
        value="/WEB-INF/applicationContext.xml,
        /WEB-INF/daoContext.xml,
        /WEB-INF/action-servlet.xml"/>
</plug-in>

```

通常上面的配置，可让 Spring 容器随系统启动时完成初始化。注意在上面的配置中，Spring 的配置文件有一个 action-servlet.xml，该配置文件中全部是 Action。

使用 DelegatingRequestProcessor 的整合策略时，无须确定 action 的实现类。因为 DelegatingRequestProcessor 直接将请求转发到 Spring 容器内。所有的控制器都采用如下格式配置：

```

<action path="/processApp"
    name="appForm"
    scope="request"
    validate="true"
    input="input">
    <forward name="input" path="/processApp.do"/>
    <forward name="result" path="/WEB-INF/jsp/employee/index.jsp"/>

```

```
<forward name="login" path="/WEB-INF/jsp/login.jsp"/>
</action>
```

### 10.6.3 创建 Action

该系统分成两个模块，分别由两个业务逻辑组件：EmpManager 和 MgrManager 实现，对于普通员工的模块，总需要注入 EmpManager 组件，并提供一个 EmpBaseAction 来完成这种通用操作：

```
public class EmpBaseAction extends Action
{
    //依赖注入 EmpManager 业务逻辑组件
    protected EmpManager mgr;
    public void setMgr(EmpManager mgr)
    {
        this.mgr = mgr;
    }
}
```

另外，员工模块的其他 Action 都可以继承 EmpBaseAction，因为该父类将 mgr 属性设置成 `protected`，就是为了让子类可以直接访问。

经理模块同样提供了类似的 MgrBaseAction 类：

```
public class MgrBaseAction extends Action
{
    //依赖注入 MgrManager 业务逻辑组件
    protected MgrManager mgr;
    public void setMgr(MgrManager mgr)
    {
        this.mgr = mgr;
    }
}
```

这两个 Action 是其他业务控制器的父类，通过调用 mgr 的业务方法来处理用户请求，并根据处理结果生成响应。

### 10.6.4 异常处理

异常的处理是采用 Struts 的声明式异常处理，将异常处理部分放在 struts-config.xml 文件中配置。在本系统的 struts-config.xml 文件中增加如下代码：

```
<!-- 系统全局异常-->
<global-exceptions>
    <!-- 定义异常对应的 key 信息-->
    <exception key="auction.AuctionException"
        <!-- 定义异常类-->
        type="org.yeeku.exception.AuctionException"
        scope="request"
        <!-- 跳转到 error.jsp-->
        path="/WEB-INF/jsp/error.jsp"/>
</global-exceptions>
```

上面的定义表明：如果 Struts 的 Action 中抛出 AuctionException 异常，则跳转到 /WEB-INF/jsp/error.jsp 页面，系统将 key 对应的消息显示在 error.jsp 页面中，为了在 error.jsp 显示该异常信息，则应增加如下代码：

```
<!-- logic:messagesPresent 是 Struts 的表现层标签-->
<logic:messagesPresent>
    <!-- 输出异常信息-->
    <p class="error">
        <ul>
            <html:messages id="exception">
                <li><bean:write name="exception"/></li>
            </html:messages>
        </ul><hr />
    </p>
</logic:messagesPresent>
```

## 10.6.5 权限控制

权限控制决定不同的用户可以访问不同的系统页面。对于本系统而言，主要包含 Employee 和 Manager 两个角色，这两个角色分别可以登录不同的模块，调用不同的业务逻辑方法。

与第 9 章示例不同，本系统采用 Spring 的 AOP 支持来完成权限控制。借助于 Spring 的 AOP 框架可以大大简化权限控制，将权限检查部分放入权限检查拦截器中完成，然后由 Action 生成相应的代理，最后在代理中增加权限检查部分。

下面是 Manager 权限检查拦截器的源代码：

```
//拦截器，实现 MethodInterceptor 接口
public class MgrAuthorityInterceptor implements MethodInterceptor
{
    //实现 MethodInterceptor 接口必须实现的方法
    public Object invoke(MethodInvocation invocation) throws Throwable
    {
        HttpServletRequest request = null;
        ActionMapping mapping = null;
        Object[] args = invocation getArguments();
        //解析目标方法的参数
        for (int i = 0 ; i < args.length ; i++)
        {
            if (args[i] instanceof HttpServletRequest) request =
                (HttpServletRequest)args[i];
            if (args[i] instanceof ActionMapping) mapping = (ActionMapping)
                args[i];
        }
        //从 Session 获取 level 属性的值
        String level = (String)request.getSession().getAttribute("level");
        //如果 level 属性值不为空，且为 mgr，则可以调用目标方法
        if ( level != null && level.equals("mgr") )
        {
            return invocation.proceed();
        }
        else
        {
```

```
        return mapping.findForward("login");
    }
}
}
```

系统的拦截器需实现 MethodInterceptor 接口，该接口来自于 AOP 联盟。该拦截对所有的方法进行处理，（事实上，Action 内只有一个方法：execute）如果用户已经登录，而且 Session 中 level 属性值为 mgr，表明以经理身份登录，则正常执行目标对象的方法；否则跳转到登录页面。

普通员工同样包含了权限检查拦截器：

```
//拦截器，实现 MethodInterceptor 接口
public class EmpAuthorityInterceptor implements MethodInterceptor
{
    //实现 MethodInterceptor 接口必须实现的方法
    public Object invoke(MethodInvocation invocation) throws Throwable
    {
        HttpServletRequest request = null;
        ActionMapping mapping = null;
        Object[] args = invocation.getArguments();
        //解析目标方法的参数
        for (int i = 0 ; i < args.length ; i++)
        {
            if (args[i] instanceof HttpServletRequest) request =
                (HttpServletRequest)args[i];
            if (args[i] instanceof ActionMapping) mapping = (ActionMapping)
                args[i];
        }
        //获取 Session 中的用户级别
        String level = (String)request.getSession().getAttribute("level");
        //如果 level 属性不为空，level 属性为 emp 或 mgr 都可以继续调用目标方法
        if (level != null && (level.equals("emp") || level.equals("mgr")))
        {
            return invocation.proceed();
        }
        //否则返回登录页面
        else
        {
            return mapping.findForward("login");
        }
    }
}
```

Spring 的 AOP 框架很好地与 IoC 容器结合，并将两个拦截器配置在 Spring 的容器中，配置代码如下：

```
<!-- 定义经理权限检查拦截器 -->
<bean id="mgrAuthorityInterceptor" class="org.yeeku.struts.authority.
MgrAuthorityInterceptor"/>
<!-- 定义普通员工权限检查拦截器 -->
<bean id="empAuthorityInterceptor" class="org.yeeku.struts.authority.
EmpAuthorityInterceptor"/>
```

使用 BeanNameAutoProxyCreator 可为需要执行权限检查的 Action 生成权限检查代理，使用 BeanNameAutoProxyCreator 的配置代码如下：

```

<!--定以普通员工权限拦截器生成代理-->
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAuto
ProxyCreator">
    <property name="beanNames">
        <list>
            <value>/empPunch</value>
            <value>/processEmpPunch</value>
            <value>/viewEmpSalary</value>
            <value>/viewUnPunch</value>
            <value>/appChange</value>
            <value>/processApp</value>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
            <value>empAuthorityInterceptor</value>
        </list>
    </property>
</bean>

```

使用 BeanNameAutoProxyCreator 为目标 bean 生成代理时，如果目标 bean 被覆盖，则容器中再也没有目标 bean，而只存在代理。

**注意：**在这种方式下，处理用户请求的 Action 不是用户编写的 Action 实例，而是系统生成的代理，如果 Action 没有实现接口，则由系统自动生成代理。

## 10.6.6 控制器配置

为了 Struts 与 Spring 的整合，Struts 的 Action 需要在两个地方配置。

- 在 struts-config.xml 文件中配置对应的 Action。
- 在 Spring 容器中配置实际的 Action。

例如，当用户请求 viewEmpSalary.do 时，ActionServlet 将拦截到该请求，然后将该请求转发到 struts-config.xml 中名为 / viewEmpSalary 的 Action。因此，必须在 struts-config.xml 文件中增加 /viewEmpSalary 的映射。映射代码如下：

```

<!-- 查看员工历史工资 -->
<action path="/viewEmpSalary" scope="request">
    <forward name="viewSalary" path="/WEB-INF/jsp/employee/viewSalary.
    jsp"/>
    <forward name="login" path="/WEB-INF/jsp/login.jsp"/>
</action>

```

DelegatingRequestProcessor 负责将该 Action 转发到 Spring 容器中同名的 bean，则必须在 Spring 的容器中配置同名的 bean，该 bean 才是 Action 的真正实现类。因此在 Spring 的配置文件中必须增加如下代码片段：

```

<bean name="empActionTemplate" abstract="true" singleton="false">
    <property name="mgr">
        <ref bean="empManager"/>
    </property>
</bean>

```

```
<bean name="/viewEmpSalary,/viewMgrSalary"
      class="org.yeeku.struts.ViewSalaryAction" parent="empActionTemplate"/>
```

注意：配置文件使用继承来完成配置，因为所有的 Employee 模块的 Action 都需要注入 EmpManager 组件，这是通用部分，因此将这个通用的注入配置成模板；而 ViewSalaryAction 指定 name 时，确定了两个 name 属性，这意味着这个 bean 可以对应 struts-config.xml 文件中的两个 Action。

struts-config.xml 文件的配置相对简单，与单纯采用 Struts 的应用并没有太大区别，其区别只有两个。

- 使用 DelegatingRequestProcessor 代替 RequestProcessor 处理类。
- 配置 action 元素时，可以不确定 class 属性，因为 action 不再由 Struts 负责创建实例。

下面介绍 Action 在 Spring 中的配置，因为员工 Action 必须接收容器注入员工业务的逻辑组件，而经理 Action 必须接收容器注入经理业务的逻辑组件。因此，可以使用继承来简化 Spring 容器中 Action 的配置。

下面是所有 Action 在 Spring 中配置文件的源代码：

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 登录系统所使用的 Action-->
    <bean name="/logout" class="org.yeeku.struts.LogoutAction"
          singleton="false"/>
    <!-- =====下面是员工的 Action=====-->
    <!-- 所有的 Employee 的 Action 的模板-->
    <bean name="empActionTemplate" abstract="true" singleton="false">
        <property name="mgr">
            <ref bean="empManager"/>
        </property>
    </bean>
    <!-- 处理登录-->
    <bean name="/processLogin" class="org.yeeku.struts.LoginAction"
          parent="empActionTemplate"/>
    <!-- 进入打卡-->
    <bean name="/empPunch,/mgrPunch"
          class="org.yeeku.struts.PunchAction" parent="empActionTemplate"/>
    <!-- 员工打卡，实际上经理打卡、员工打卡用的是同一个处理类-->
    <bean name="/processEmpPunch,/processMgrPunch"
          class="org.yeeku.struts.ProcessPunchAction" parent="empActionTemplate"/>
    <!-- 员工查看工资，实际上经理查看自己工资与员工查看工资用同一个处理类-->
    <bean name="/viewEmpSalary,/viewMgrSalary"
          class="org.yeeku.struts.ViewSalaryAction" parent="empActionTemplate"/>
    <!-- 员工查看出勤异动-->
    <bean name="/viewUnPunch" class="org.yeeku.struts.ViewUnAttendAction"
          parent="empActionTemplate"/>
    <!-- 员工进入出勤异动申请 -->
    <bean name="/appChange" class="org.yeeku.struts.AppChangeAction" parent="empActionTemplate"/>
    <!-- 处理员工的出勤异动申请 -->
```

```
<bean name="/processApp" class="org.yeeku.struts.ProcessAppAction"
parent="empActionTemplate"/>
<!-- =====下面是经理的 Action===== -->
<!-- 经理模块所有的 Action 的模块 bean-->
<bean name="mgrActionTemplate" abstract="true" singleton="false">
    <property name="mgr">
        <ref bean="mgrManager"/>
    </property>
</bean>
<!-- 经理查看部门工资 -->
<bean name="/viewDeptSal" class="org.yeeku.struts.ViewDeptAction" parent =
"mgrActionTemplate"/>
<!-- 经理增加员工 -->
<bean name="/processAdd" class="org.yeeku.struts.AddEmpAction" parent =
"mgrActionTemplate"/>
<!-- 经理查看员工 -->
<bean name="/viewEmp" class="org.yeeku.struts.ViewEmpAction" parent =
"mgrActionTemplate"/>
<!-- 经理查看申请 -->
<bean name="/viewApp" class="org.yeeku.struts.ViewAppAction" parent =
"mgrActionTemplate"/>
<!-- 经理处理申请 -->
<bean name="/check" class="org.yeeku.struts.CheckAppAction" parent =
"mgrActionTemplate"/>
<!-- 以经理权限拦截器生成代理 -->
<bean class="org.springframework.aop.framework.autoproxy. BeanNameAuto
ProxyCreator">
    <property name="beanNames">
        <list>
            <value>/viewDeptSal</value>
            <value>/processAdd</value>
            <value>/viewEmp</value>
            <value>/check</value>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
            <value>mgrAuthorityInterceptor</value>
        </list>
    </property>
</bean>
<!-- 以普通员工权限拦截器生成代理 -->
<bean class="org.springframework.aop.framework.autoproxy. BeanNameAuto
ProxyCreator">
    <property name="beanNames">
        <list>
            <value>/empPunch</value>
            <value>/processEmpPunch</value>
            <value>/viewEmpSalary</value>
            <value>/viewUnPunch</value>
            <value>/appChange</value>
            <value>/processApp</value>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
            <value>empAuthorityInterceptor</value>
        </list>
    </property>
</bean>
```

```
</bean>
<!-- 定义经理权限检查拦截器 -->
<bean id="mgrAuthorityInterceptor" class="org.yeeku.struts.authority.
MgrAuthorityInterceptor"/>
<!-- 定义普通员工权限检查拦截器 -->
<bean id="empAuthorityInterceptor" class="org.yeeku.struts.authority.
EmpAuthorityInterceptor"/>
</beans>
```

在上面配置文件中，使用继承简化了 Action 的配置，同时使用 AOP 框架为每个 Action 生成代理。该系统使用 BeanNameAutoProxyCreator 分别为 Action 生成权限检查代理。因为系统有两个不同模块，因此配置了两个代理创建器 bean，它们负责为所有的 Action 生成对应的权限检查代理。

## 本章小结

本章介绍了一个完整的 J2EE 项目，本项目涉及的表相对较多，业务逻辑也相对复杂。本实例从面向对象分析开始，设计出系统的持久化类，根据系统持久化类生成数据库。

另外，采用流行的 J2EE 架构模型：Struts + Spring + Hibernate，构建了一个完整的 J2EE 应用，包括系统 DAO 层设计，Service 层设计和 MVC 框架，希望读者能参考本章的内容，设计出优秀的轻量级 J2EE 系统。

# 技术凝聚实力 专业创新出版

博文视点 ([www.broadview.com.cn](http://www.broadview.com.cn)) 资讯有限公司是电子工业出版社、CSDN.NET、《程序员》杂志联合打造的专业出版平台，博文视点致力于——IT专业图书出版，为IT专业人士提供真正专业、经典的好书。

请访问 [www.dearbook.com.cn](http://www.dearbook.com.cn) (第二书店) 购买优惠价格的博文视点经典图书。

请访问 [www.broadview.com.cn](http://www.broadview.com.cn) (博文视点的服务平台) 了解更多更全面的出版信息；您的投稿信息在这里将会得到迅速的反馈。

## 典藏外版精品



JOLT 大奖经典之作，关于交互系统设计的真知灼见！

### 软件观念革命 ——交互设计精髓

[美]Alan Cooper, Robert Reimann 著  
詹剑锋、张知非 等译 2005年6月出版  
ISBN 7-121-01180-8 89.00元 650页

这是一本在交互设计前沿有着10年设计咨询经验及25年计算机工业界经验的卓越权威——VB之父ALAN COOPER撰写的“设计数字化产品行为”的启蒙书。

全面阐释软件开发的最佳实践和重大陷阱！

### 程序员修炼之道 ——从小工到专家

[美]Andrew Hunt, David Thomas 著  
马维达 译  
2004年4月出版 ISBN 7-5053-9719-2  
48.00元 362页



本书由一系列独立的部分组成，涵盖的主题从个人责任、职业发展，直至用于使代码保持灵活、并且易于改编和复用的各种架构技术，利用许多富有娱乐性的奇闻轶事、有思想性的例子以及有趣的类比。



设计心理学的经典之作！

中科院院士张跋亲自作序，人机交互专家叶展高度评价！

### 情感化设计

[美]Donald A. Norman 著  
付秋芳、程进三 译  
2005年5月出版 ISBN 7-121-00940-4  
36.00元 206页

设计的最高境界是什么？本书以独特细腻、轻松诙谐的笔法，以本能、行为和反思这三个设计的不同维度为基础，阐述了情感在设计中所处的重要地位与作用。



软件管理方面的“MBA教程”的称号！荣获第15届JOLT大奖！

### JOEL说软件

[美]Joel Spolsky 著  
谭明金、王平 译  
2005年9月出版 ISBN 7-121-01641-9  
39.00元 301页

这是一本关于软件管理的随笔文集。这是一本会让你受益颇多的休闲之作。



被欧美许多重要大学用于“程序设计语言”或者“软件系统”课程！

### 程序设计语言——实践之路

[美]Michael L.Scott 著  
裘宗燕 译  
2005年3月出版 ISBN 7-121-00900-5  
88.00元 884页

这是一本很有特色的教材，其核心是讨论程序设计语言的工作原理和技术。

本书作者Michael Scott是计算机领域的著名学者，译者是北京大学的裘宗燕教授，他熟悉专业，译笔流畅，是一本难得的著、译双馨的佳作。



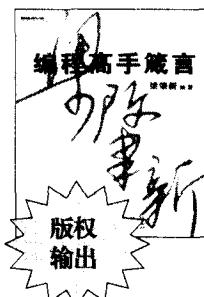
北京印刷学院刘浩学教授翻译，方正色彩管理小组审校推荐！

### 色彩管理

[美]Bruce Fraser, Chris Murphy, Fred Bunting 著  
刘浩学、梁炯、武兵 等译  
2005年7月出版 ISBN 7-121-01470-X  
168.00元 504页

读懂它，不仅可以掌握精确一致的色彩复制技术，在最普及的图形图像软件中如何进行色彩管理，而且还可以知晓建立、评估和编辑ICC PROFILE；不仅可以知道色彩管理是怎么回事，如何做，而且知道为什么要这样做；不仅可以将色彩管理嵌入生产流程中，而且还能帮助改善生产流程，提高工作效率。

# 广 观 本 版 线 上



编程高手箴言

梁肇新

版权  
输出

荣获 2004 年度“中国图书奖”和  
“全国优秀畅销书奖”!

## 编程高手箴言

梁肇新 编著

2003年11月出版 ISBN 7-5053-9141-0  
50.00元(含光盘1张) 416页

中国最具知名度的程序员之一,《超级解霸》作者梁肇新首部专著!

全书通篇没有时髦的IT新名词或新思想,而是踏踏实实地对很多知识进行了深刻的剖析,有助于为编程打下坚实的基础。

用理论指导动手实践

用实践深化理解理论

## 自己动手写操作系统

于渊 编著

2005年8月出版 ISBN 7-121-01577-3  
48.00元(含光盘1张) 374页

本书不同于其他的理论型书籍,而是提供给读者一个动手实践的路图。

在详细分析操作系统原理的基础上,用丰富的实例代码,一步一步地指导读者用C语言和汇编语言编写出一个具备操作系统基本功能的操作系统框架。

加密与解密

荣获 2003 年“全国优秀畅销书奖”,看雪论坛鼎立打造!

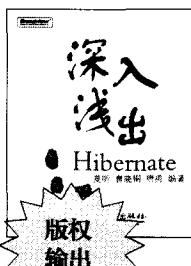
## 加密与解密(第二版)

段钢 编著

2003年6月出版 ISBN 7-5053-8648-4  
49.00元(含光盘1张) 519页

本书全面讲述了Windows平台下的最新软件加密与解密技术及相关解决方案,采用循序渐进的方式,从基本的跟踪调试到深层的拆解脱壳,从浅显的注册码分析到商用软件保护,几乎囊括了Windows下的软件保护的绝大多数内容。

版权  
输出



深入  
浅出

Hibernate

版权  
输出

国内第一本重量级 Hibernate 图书。

## 深入浅出 Hibernate

莫昕、曹晓钢、唐勇 编著

2004年7月出版 ISBN 7-121-00670-7  
59.00元 545页

本书由互联网上影响广泛的开放文档OpenDoc系列自由文献首份文档“Hibernate 开发指南”发展而来。在编写过程中,进行了重新构思与组织,同时对内容的深度与广度进行了重点强化。



版权  
输出

同类书销量第一!

## ERP 原理·设计·实施(第3版)

罗鸿 编著

2005年4月出版 ISBN 7-121-01059-3  
38.00元 384页

本书对ERP相关知识的讨论涵盖了原理、设计与应用的全部过程。前两版出版后均引起了很大的社会反响,作者收到大量读者来信,并与读者进行了良好的交互。第3版再次增加了一些内容,更加贴近读者需要。



版权  
输出

本书通过多种典型实例详细介绍了在Windows系统下数据恢复技术的原理和方法。

## 数据恢复技术(第2版)

戴士剑、涂彦晖 编著

2005年3月出版 ISBN 7-121-00756-8  
69.00元 711页

本书内容包括:硬盘数据组织、文件系统原理、数据恢复技术、文档修复技术、密码丢失处理技术、数据安全技术和数据备份技术。作者戴士剑是国内知名数据恢复专家,有多年的数据恢复工作经验,为客户提供过上千次的数据恢复服务。

# 《轻量级 J2EE 企业应用实战——Struts+ Spring+ Hibernate 整合开发》读者调查表

尊敬的读者：

感谢您对我们的支持与爱护。为了今后为您提供更优秀的图书，请您抽出宝贵的时间将您的意见以下表的方式及时告知我们（可另附页）。我们将从中评选出热心读者若干名，免费赠阅我们以后出版的图书。

姓名:	性别: <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄:	职业:
通信地址:		邮政编码:	
电话:	传真:	E-mail:	

## 1. 影响您购买本书的因素（可多选）：

- 封面封底     价格     内容提要、前言和目录     书评广告     出版物名声  
 作者名声     正文内容     其他 \_\_\_\_\_

## 2. 您对本书的满意度：

从技术角度     很满意     比较满意     一般     较不满意     不满意

改进意见 \_\_\_\_\_

从文字角度     很满意     比较满意     一般     较不满意     不满意

改进意见 \_\_\_\_\_

从版面、封面设计角度     很满意     比较满意     一般     较不满意

不满意     改进意见 \_\_\_\_\_

## 3. 您最喜欢书中的哪篇（或章、节）？请说明理由。

\_\_\_\_\_

## 4. 您最不喜欢书中的哪篇（或章、节）？请说明理由。

\_\_\_\_\_

## 5. 您希望本书在哪些方面进行改进？

\_\_\_\_\_

## 6. 您感兴趣或希望增加的图书选题有：

\_\_\_\_\_

通信地址：北京万寿路 173 信箱 博文视点（100036） 电话：010-51260888  
 如果您对我们出版的图书有任何意见和建议，也可以发邮件给我们，我们将及时回复。  
 E-mail：jsj@phei.com.cn, editor@broadview.com.cn



## 反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传 真：（010）88254397

E-mail： dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036



## 作者简介

作者从事过 6 年的 J2EE 应用开发，担任过 LITEON 公司的 J2EE 技术主管，负责该公司的企业信息平台的架构设计，担任过广东龙泉科技有限公司的 J2EE 技术培训导师，目前在新东方 IT 培训中心担任 J2EE 培训讲师。培训的学生已在华为、从兴电子、瑞达通信、中企动力等公司就职，在珠三角的 J2EE 行业极具影响力。

# 轻量级 J2EE 企业应用实战 ——Struts+Spring+Hibernate 整合开发

## 1. 经验丰富，针对性强

本书凝聚了作者多年 J2EE 开发经验，不是一本学院派的理论读物，而是一本实际的开发指南。

## 2. 内容实际，实用性强

本书所介绍的 J2EE 应用范例，是目前企业流行的开发架构，绝对严格遵守 J2EE 开发规范。读者参考本书的架构，完全可以身临其境地感受企业实际开发。

## 3. 高屋建瓴，启发性强

本书介绍的几种架构模式，几乎是时下最全面的 J2EE 架构模式。这些架构模式可以直接提升读者对系统架构设计的把握。

本书适用于有较好的 Java 编程基础，有初步的 J2EE 编程基础的读者。本书既可以作为 J2EE 初学者的入门书籍，也可作为 J2EE 应用开发者的提高指导。

图书分类: Java > J2EE



网上订购：[www.dearbook.com.cn](http://www.dearbook.com.cn)  
第二书店·第一服务



责任编辑：高洪霞 裴杰  
责任美编：张子建 谢丹丹



ISBN 978-7-121-03998-0



9 787121 039980 >

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

定价：65.00元（含光盘1张）