

Java开发利器

珍藏版



附：视频讲解
本书源代码

强锋科技

郭 锋 编著

Spring

从入门到精通

- ✓ 以最新版本Spring 2.0进行讲解
- ✓ 语言流畅，实例丰富，非常容易学习
- ✓ 书中代码全部来自实际项目，有很强的实践指导意义
- ✓ 最后配有综合案例，让您领会项目开发的思路
- ✓ 作者是系统架构师，有良好的大局观，讲解Spring高屋建瓴
- ✓ 配多媒体演示光盘，提高学习效率



清华大学出版社

Spring 从入门到精通



丛书特色:

- ✓ 作者全都是富有丰富编程经验的一线开发人员
- ✓ 全面攻克 Java 开发领域的前沿技术
- ✓ 极大地提升您的 Java 应用开发水准
- ✓ 确保技术的实用性和深入性
- ✓ 贯穿丰富的应用实例, 真正做到学以致用
- ✓ 配视频演示光盘讲述界面操作, 既节省篇幅, 又容易上手

光盘内容:

- ✓ 全程录制本书涉及软件的基本操作
- ✓ 书中所涉及的程序源代码

ISBN 7-302-13811-7



9 787302 138112 >

定价: 55.00 元(附光盘 1 张)

Java 开发利器

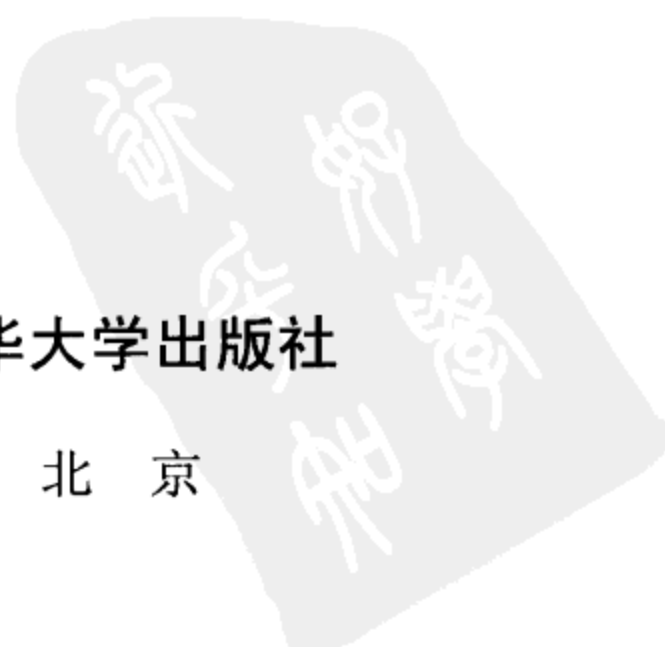
Spring 从入门到精通

强锋科技

郭 锋 编著

清华大学出版社

北 京



内 容 简 介

本书由浅入深,循序渐进地介绍了 Spring 的体系结构和相关知识点,目的是帮助初学者快速掌握 Spring,并能使用 Spring 进行应用程序的开发。

全书共分 14 章,内容涵盖了 Spring 的基础概念、核心容器、Spring AOP、事务处理、持久层封装、Web 框架、定时器、Spring 和 Struts、Spring 和 Hibernate、Spring 和 Ant、Spring 和 Junit。本书最大的特色在于每章都是由浅入深,从一个简单的示例入手,让读者快速了解本章内容,然后再详细讲解本章涉及的基本原理和知识点,最后再通过一个详细的示例来巩固所学内容。本书每一章的例子都是经过精挑细选,具有很强的针对性,力求让读者通过亲自动手做而掌握其原理和方法,从而学习尽可能多的知识。

本书适用于初、中级软件开发人员,同时也可用作高校相关专业师生和社会培训班的教材。

版权所有,翻印必究。举报电话:010-62782989 13501256678 13801310933

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

本书防伪标签采用特殊防伪技术,用户可通过在图案表面涂抹清水,图案消失,水干后图案复现;或将表面膜揭下,放在白纸上用彩笔涂抹,图案在白纸上再现的方法识别真伪。

图书在版编目(CIP)数据

Spring 从入门到精通/郭锋编著. —北京:清华大学出版社, 2006.10

(Java 开发利器)

ISBN 7-302-13811-7

I. S… II. 郭… III. Java 语言-程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2006)第 109604 号

出 版 者:清华大学出版社 地 址:北京清华大学学研大厦

<http://www.tup.com.cn> 邮 编:100084

社 总 机:010-62770175 客户服务:010-62776969

组稿编辑:欧振旭

文稿编辑:马子杰

封面设计:范华明

版式设计:侯哲芬

印 刷 者:清华大学印刷厂

装 订 者:三河市李旗庄少明装订厂

发 行 者:新华书店总店北京发行所

开 本:185×260 印张:31.5 字数:697 千字

版 次:2006 年 10 月第 1 版 2006 年 10 月第 1 次印刷

书 号:ISBN 7-302-13811-7/TP·8307

印 数:1~5000

定 价:55.00 元



企业应用软件越来越复杂，导致软件开发人员常常把精力放在了技术的研发和应用，而忽略了对业务本身复杂性的分析。如果您目前正疲于应付自己研发的应用框架所出现的各种各样的问题，或者说您正打算从事J2EE开发，却对J2EE庞大的体系无所适从，那么请选择Spring吧！它为您提供了在企业应用软件开发中所需要的一站式服务，帮助您减少花费在软件技术上的时间，从而把精力放在对业务本身的理解上。

您在学习Spring的过程中，应当首先从环境配置开始，然后通过一些简单的实例，快速掌握Spring开发的基本流程，接着再由浅入深地学习一下Spring的各个知识点，最后要多做几个实例以巩固所学的知识。



清华大学出版社长期以来致力于 Java 开发技术专业书籍的编撰和出版，所出版的 Java 技术书籍以其系统、专业、实用而深受广大读者的喜爱。本套“Java 开发利器”丛书也秉承了清华程序设计图书的传统优势，集技术与应用于一体，将全面提升您的 Java 应用开发水准，并将 Java 开发的前沿技术一网打尽。

第一批（已出版）：

《Eclipse 从入门到精通》
《Hibernate 开发及整合应用大全》
《Struts Web 设计与开发大全》
《Tomcat Web 开发及整合应用》
《J2ME 手机游戏开发技术详解》

第二批：

《Eclipse SWT/JFace 核心应用》
《Eclipse Web 开发从入门到精通（实例版）》
《Spring 从入门到精通》
《Ant 开发及整合应用详解》
《JSP 网络开发技术与整合应用》

前 言

企业应用软件的开发变得越来越庞大，软件技术也越来越复杂。为此，软件开发人员常常忙于对技术的研发和应用，而忽略了对业务本身复杂性的分析。为了减少软件开发人员花费在软件技术上的时间，而把精力放在对业务本身的理解，市场上出现了很多解决这个问题应用框架，而 Spring 无疑是其中最优秀的一个。

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。该框架的主要优势是其良好的设计和分层架构，正因如此，软件开发人员可以只选择 Spring 提供的某项技术，例如 AOP，而不需要使用它提供的其他技术。Spring 还提供了和其他开源软件的无缝结合，为 J2EE 应用程序开发提供集成的框架。

为了帮助众多初学者快速掌握 Spring，笔者精心编著了本书。本书依照读者的学习规律，首先介绍基本概念和基本操作。在读者掌握了这些基本概念和基本操作的基础上，再对内容进行深入的讲解，严格遵循由浅入深、循序渐进的原则。本书按照对 Spring 基础知识进行掌握的先后顺序进行编排。

本书在编排上力争让读者能够快速掌握 Spring 的使用方法。例如，在第 2 章中首先以一个实例告诉读者如何使用 Spring，让读者快速入门，掌握 Spring 的基本流程。接下来按照 Spring 基础知识的结构来进行讲解，从一个简单的示例入手，由浅入深，让读者快速了解本章内容，然后再详细讲解本章涉及的基本原理和知识点，最后再通过一个详细的示例来巩固所学内容。

本书特色

1. 快速入门

经常听到软件开发人员讲希望学习 Spring，但因其环境配置太复杂，导致开发人员降低了学习的兴趣。本书对环境的配置进行了详细的讲解，尽量减少读者浪费在环境配置上的时间，把更多的精力放在对 Spring 原理的理解上。

2. 通俗易懂

本书可以让读者很快地了解使用 Spring 进行应用程序开发的基本流程，从而迅速动手进行 Spring 的应用开发。书中的每章都是从一个简单的示例入手，让读者快速了解本章内容，然后再详细讲解本章涉及到的基本原理和知识点，最后再通过一个详细的示例来巩固所学内容。这样由浅入深的讲解符合读者的学习过程。对于比较晦涩的知识，尽量通过大量的示例来进行讲解。

3. 主次分明，避虚就实

本书定位于初学者，所以除了内容通俗易懂，能快速上手之外，还尽量避免对大量参

数的介绍。而这些内容在 Spring 的参考手册里都有详细的介绍，因而本书只讲解常用的一些知识点。

4. 配有视频讲解光盘

对于一些基础性的操作，本书配备了多媒体光盘，并配有语音讲解，以方便读者学习。

本书内容

第 1 章是 Spring 概述。介绍 Spring 的历史、特点和工作原理，并详细讲解了学习 Spring 应该注意的问题，给出了网络上一些比较好的学习资源。

第 2 章开始 Spring 之旅。首先一步一步地帮助读者建立起 Spring 的开发环境，然后从一个实例入手进行讲解，让读者快速了解 Spring 的开发流程。

第 3 章是 Spring 基础概念。首先从 IoC 的基本思想开始讲解，然后通过实例的方式使读者对其概念和工作原理有较为深入的了解，最后把第 2 章中的第一个实例进行改编，使其通过构造方式来实现同样的功能。

第 4 章是 Spring 的核心容器。首先讲解 Spring 的基础 Bean 的相关知识，然后介绍 Spring 是如何对 Bean 进行管理，最后讲解 Spring 提供的一些更强大的功能。BeanFactory 和 ApplicationContext 是了解 Spring 核心的关键。

第 5 章是 Spring 的 AOP。首先让读者了解 AOP 的基本思想，然后通过对 Java 代理机制进行分析，引出 AOP 的 3 个关键概念，接着对 Spring 的 AOP 进行介绍，最后通过一个完整的实例让读者掌握 Spring AOP 的使用方法。

第 6 章是 Spring 的事务处理，主要讲述 Spring 事务处理的基本概念和使用方法。首先对传统的事务处理进行介绍，然后介绍怎样使用 Spring 来进行事务处理。其中介绍了 Spring 所提供的声明式和编程式事务处理。

第 7 章是 Spring 的持久层封装，主要结合 Spring 的持久层来讲解如何使用 Spring 的事务处理。首先对传统的数据库连接进行介绍，然后介绍怎样使用 Spring 来进行持久层封装。

第 8 章是 Spring 的 Web 框架。首先从 MVC 的实现原理入手，然后通过一个实例让读者快速了解 Spring MVC 的开发流程，接着讲解 Spring 的模型、视图、控制器及数据验证和对国际化的支持，最后通过一个实例详细讲解 Spring MVC 的开发过程。

第 9 章是 Spring 的定时器。首先对定时器进行介绍，接着讲解定时器的两种实现方式，最后讲解在 Spring 中如何实现定时器。

第 10 章是 Spring 与 Struts 的整合。首先对 Struts 进行介绍，然后通过一个示例使读者快速掌握 Struts，接着讲解 Struts 提供的几个比较重要的类及 Struts 对国际化的支持和自定义标签功能，然后给出 Spring 与 Struts 整合的几种实现方式，最后通过实例展示 Spring 与 Struts 的整合方式。

第 11 章是 Spring 与 Hibernate 的整合。首先对 Hibernate 进行介绍，并给出一个示例使读者快速了解 Hibernate，接着介绍 Hibernate 的配置方式和映射方法及几个辅助工具，然后给出 Spring 与 Hibernate 的整合方式，并通过实例展示这种整合方式。

第 12 章是在 Spring 中使用 Ant。首先对 Ant 进行介绍，然后着重介绍 Ant 的下载和安

装，最后结合前面的 Spring 示例介绍 Ant 的使用方法。

第 13 章是在 Spring 中使用 Junit。首先对 Junit 进行介绍，然后着重介绍 Junit 的下载和安装，最后结合前面的 Spring 示例介绍 Junit 的使用方法。

第 14 章是用 Spring 实现新闻发布系统实例。主要讲解使用 Spring 和 Hibernate 来实现新闻发布系统，从而使读者对 Spring 和 Hibernate 有一个全面的掌握。

本书读者对象

本书具有通俗易懂、主次分明、指导性强、快速入门的特点，力求以通俗的语言及丰富的实例来指导读者透彻学习 Spring 各方面的技术。本书可以作为初学 Spring 的入门教材，也可以帮助中级读者提高技能，对高级读者也有一定的启发意义。

本书作者

本书由郭锋统筹编写，其他参与编写、资料整理和程序调试及视频录制的人员有王俊标、陈晨、李卓龙、高守传、郭瑞、周宇炜、蔡雪焄、陈杰、荣飞、郑林、张路平、项宇峰、罗皓菡、赵正坤、公芳亮、程明雷、梁文建、马斗、邱哲、宋昕、陈刚、张传毓、赖梅艳、杨晓强、梁计锋、强致懿、郭腊梅、肖萍和程鹏辉等，在此一并表示感谢。若您在阅读过程中有疑难问题，可发 E-mail 到 gf3008@126.com，作者将尽最大努力为您答疑解惑。

由于写作时间仓促，加之作者水平所限，本书恐有不妥之处，望各位读者和专家指正。

作 者

2006 年 9 月



读者意见反馈卡

您购买的书名: _____ 您的姓名: _____ 性别: ☐男 ☐女
年龄: _____ 文化程度: _____ 职业: _____
邮编: _____ 通信地址: _____ E-mail: _____
您常用的软件: 1 _____ 2 _____ 3 _____ 4 _____

您购买本书的原因 (可多选):

☐封面与装帧 ☐引言目录 ☐正文内容 ☐丛书风格 ☐价格 ☐光盘 ☐专业性强 ☐别人介绍
☐出版社或作者名声 ☐售后服务

本书最令您满意的是 (可多选):

☐专业性强、覆盖面广 ☐内容翔实、定位准确 ☐精益求精、售后服务

您可以承受的图书价格:

☐20 元以下 ☐30 元以下 ☐40 元以下 ☐50 元以下 ☐只要内容好, 不论价格

您对本书的评价:

封面装帧:	<input type="checkbox"/> 很好	<input type="checkbox"/> 较好	<input type="checkbox"/> 一般	<input type="checkbox"/> 不满意	建议_____
印刷质量:	<input type="checkbox"/> 很好	<input type="checkbox"/> 较好	<input type="checkbox"/> 一般	<input type="checkbox"/> 不满意	建议_____
正文质量:	<input type="checkbox"/> 很好	<input type="checkbox"/> 较好	<input type="checkbox"/> 一般	<input type="checkbox"/> 不满意	建议_____
写作风格:	<input type="checkbox"/> 很好	<input type="checkbox"/> 较好	<input type="checkbox"/> 一般	<input type="checkbox"/> 不满意	建议_____
专业水平:	<input type="checkbox"/> 很好	<input type="checkbox"/> 较好	<input type="checkbox"/> 一般	<input type="checkbox"/> 不满意	建议_____

您希望增加哪些图书选题: 1 _____ 2 _____ 3 _____

您认为本书有哪些错误:

章_____节_____	页码_____	行_____列_____	图号_____	错误_____	应改为_____
章_____节_____	页码_____	行_____列_____	图号_____	错误_____	应改为_____
章_____节_____	页码_____	行_____列_____	图号_____	错误_____	应改为_____
章_____节_____	页码_____	行_____列_____	图号_____	错误_____	应改为_____

您的其他建议:

1 _____
2 _____
3 _____

请填写好本卡后寄给:

清华大学校内金地公司

邮编: 100084

电话: (010) 62791976-220

《Spring 从入门到精通》编辑部收

传真: (010) 62788903

公司网址: www.thjd.com.cn

E-mail: oyzx_sp@263.net

如需本书可与本编辑部联系邮购, 汇款请按以上地址填写, 另加邮费 15% (挂号)

目 录

第 1 篇 Spring 入门

第 1 章	Spring 概述	2
1.1	Spring 的历史	2
1.2	Spring 简介	2
1.2.1	Spring 框架介绍	2
1.2.2	Spring 的特点	4
1.3	如何学习 Spring	4
1.4	网络上的资源	5
1.5	小结	5
第 2 章	开始 Spring 之旅	6
2.1	建立 Spring 开发环境	6
2.1.1	下载 JDK	6
2.1.2	安装 JDK	7
2.1.3	设定 Path 与 CLASSPATH	7
2.1.4	下载 Eclipse	8
2.1.5	配置 Eclipse	8
2.1.6	下载 Tomcat	8
2.1.7	设定 TOMCAT_HOME	9
2.1.8	下载 Eclipse 的 Tomcat 插件	9
2.1.9	为 Eclipse 配置 Tomcat 插件	10
2.1.10	下载 Spring	12
2.1.11	简单介绍 Spring 包	12
2.1.12	在 Eclipse 中配置 Spring	13
2.2	第一个使用 Spring 实现 HelloWorld 的例子	15
2.2.1	编写存放 HelloWorld 的 Java 文件 HelloWorld.java	16
2.2.2	配置 Spring 的 config.xml	16
2.2.3	编写测试程序 TestHelloWorld.java	17
2.2.4	运行测试程序并查看输出结果	18
2.3	改写第一个 Spring 例子实现中英文输出	19
2.3.1	编写接口文件 Hello.java	19
2.3.2	编写实现接口的两个类 (ChHello、EnHello)	19

2.3.3	修改 Spring 的 config.xml.....	20
2.3.4	修改测试程序 TestHelloWorld.java	21
2.3.5	运行测试程序并查看输出结果.....	21
2.4	小结.....	22

第 2 篇 Spring 技术详解

第 3 章	Spring 基础概念	24
3.1	反向控制/依赖注入.....	24
3.1.1	反向控制 (IoC)	24
3.1.2	依赖注入 (DI)	29
3.2	依赖注入的 3 种实现方式.....	29
3.2.1	接口注入 (interface injection)	29
3.2.2	Set 注入 (setter injection)	30
3.2.3	构造注入 (constructor injection)	31
3.3	将 HelloWorld 实例改为构造注入方式实现.....	31
3.3.1	修改 HelloWorld.java.....	31
3.3.2	修改 config.xml	32
3.3.3	编写测试程序 TestHelloWorld.java	32
3.3.4	运行测试程序并查看输出结果.....	33
3.4	使用哪种注入方式.....	33
3.5	小结.....	33
第 4 章	Spring 的核心容器	34
4.1	什么是 Bean	34
4.2	Bean 的基础知识.....	34
4.2.1	Bean 的标识 (id 和 name)	34
4.2.2	Bean 的类 (class)	35
4.2.3	Singleton 的使用	36
4.2.4	Bean 的属性	37
4.2.5	对于 null 值的处理.....	40
4.2.6	使用依赖 depends-on	41
4.2.7	一个较为完整的 Bean 配置文档.....	41
4.3	Bean 的生命周期.....	42
4.3.1	Bean 的定义	42
4.3.2	Bean 的初始化.....	43
4.3.3	Bean 的使用	46
4.3.4	Bean 的销毁	47

4.4	用 ref 的属性指定依赖的 3 种模式	50
4.4.1	用 local 属性指定	50
4.4.2	用 Bean 属性指定	51
4.4.3	用 parent 属性指定	51
4.4.4	用 local 属性与 Bean 属性指定依赖的比较	52
4.5	Bean 自动装配的 5 种模式	52
4.5.1	使用 byName 模式	52
4.5.2	使用 byType 模式	53
4.5.3	使用 constructor 模式	54
4.5.4	使用 autodetect 模式	55
4.5.5	使用 no 模式	56
4.5.6	对 5 种模式进行总结	57
4.6	Bean 依赖检查的 4 种模式	57
4.6.1	为什么要使用依赖检查	57
4.6.2	使用 simple 模式	57
4.6.3	使用 object 模式	58
4.6.4	使用 all 模式	58
4.6.5	使用 none 模式	58
4.6.6	对 4 种模式进行总结	59
4.7	集合的注入方式	59
4.7.1	List	59
4.7.2	Set	60
4.7.3	Map	61
4.7.4	Properties	61
4.7.5	对集合的注入方式进行总结	62
4.8	管理 Bean	62
4.8.1	使用 BeanWrapper 管理 Bean	63
4.8.2	使用 BeanFactory 管理 Bean	64
4.8.3	使用 ApplicationContext 管理 Bean	65
4.8.4	3 种管理 Bean 的方式的比较	66
4.9	ApplicationContext 更强的功能	66
4.9.1	国际化支持	66
4.9.2	资源访问	69
4.9.3	事件传递	70
4.10	小结	73
第 5 章	Spring 的 AOP	74
5.1	AOP 基本思想	74

5.1.1	认识 AOP.....	74
5.1.2	AOP 与 OOP 对比分析.....	74
5.1.3	AOP 与 Java 的代理机制.....	75
5.2	从一个输出日志的实例分析 Java 的代理机制.....	75
5.2.1	通用的日志输出方法.....	75
5.2.2	通过面向接口编程实现日志输出.....	77
5.2.3	使用 Java 的代理机制进行日志输出.....	78
5.2.4	对这 3 种实现方式进行总结.....	80
5.3	AOP 的 3 个关键概念.....	81
5.3.1	切入点 (Pointcut)	81
5.3.2	通知 (Advice)	82
5.3.3	Advisor.....	82
5.4	Spring 的 3 种切入点 (Pointcut) 实现.....	82
5.4.1	静态切入点.....	82
5.4.2	动态切入点.....	83
5.4.3	自定义切入点.....	83
5.5	Spring 的通知 (Advice)	83
5.5.1	Interception Around 通知.....	83
5.5.2	Before 通知.....	84
5.5.3	After Returning 通知.....	84
5.5.4	Throw 通知.....	84
5.5.5	Introduction 通知.....	84
5.6	Spring 的 Advisor	84
5.7	用 ProxyFactoryBean 创建 AOP 代理.....	85
5.7.1	使用 ProxyFactoryBean 代理目标类的所有方法.....	85
5.7.2	使用 ProxyFactoryBean 代理目标类的指定方法.....	86
5.7.3	正则表达式简介.....	87
5.8	把输出日志的实例改成用 Spring 的 AOP 来实现.....	89
5.8.1	采用 Interception Around 通知的形式实现.....	89
5.8.2	采用 Before 通知的形式实现.....	94
5.8.3	采用 After Returning 通知的形式实现.....	97
5.8.4	采用 Throw 通知的形式实现.....	101
5.9	Spring 中 AOP 的两种代理方式.....	105
5.9.1	Java 动态代理.....	105
5.9.2	CGLIB 代理.....	106
5.10	Spring 中的自动代理.....	108
5.11	一个用 Spring AOP 实现异常处理和记录程序执行时间的实例.....	114

5.11.1	异常处理和记录程序执行时间的实例简介	114
5.11.2	定义负责异常处理的 Advice 为 ExceptionHandler.java	114
5.11.3	定义记录程序执行时间的 Advice 为 TimeHandler.java	115
5.11.4	定义业务逻辑接口 LogicInterface.java	116
5.11.5	编写实现业务逻辑接口的类 Logic1.java	116
5.11.6	编写一个不实现业务逻辑接口的类 Logic2.java	117
5.11.7	使用自动代理定义配置文件 config.xml	118
5.11.8	编写测试类 Logic1 的程序 TestAop.java	119
5.11.9	输出自动代理时类 Logic1 异常处理和记录程序执行时间的信息	119
5.11.10	编写测试类 Logic2 的程序 TestAop.java	120
5.11.11	输出自动代理时类 Logic2 异常处理和记录程序执行时间的信息	120
5.11.12	使用 ProxyFactoryBean 代理定义配置文件 config.xml	121
5.11.13	编写测试类 Logic1 的程序 TestAop.java	122
5.11.14	输出 ProxyFactoryBean 代理时类 Logic1 记录程序执行时间的信息	123
5.11.15	编写测试类 Logic2 的程序 TestAop.java	123
5.11.16	输出 ProxyFactoryBean 代理时类 Logic2 异常处理的信息	124
5.12	小结	124
第 6 章	Spring 的事务处理	125
6.1	简述事务处理	125
6.1.1	事务处理的基本概念	125
6.1.2	事务处理的特性	125
6.1.3	对事务处理特性的总结	126
6.2	事务处理的 3 种方式	127
6.2.1	关系型数据库的事务处理	127
6.2.2	传统的 JDBC 事务处理	128
6.2.3	分布式事务处理	129
6.3	Spring 的事务处理	129
6.3.1	Spring 事务处理概述	129
6.3.2	编程式事务处理	131
6.3.3	声明式事务处理	136
6.4	使用编程式还是声明式事务处理	140
6.5	小结	140
第 7 章	Spring 的持久层封装	141
7.1	传统的 JDBC 数据访问技术	141
7.2	通过 XML 实现 DataSource（数据源）注入	142
7.2.1	使用 Spring 自带的 DriverManagerDataSource	142
7.2.2	使用 DBCP 连接池	144

7.2.3	使用 Tomcat 提供的 JNDI.....	145
7.3	使用 JdbcTemplate 访问数据	146
7.3.1	Template 模式简介.....	146
7.3.2	回顾事务处理中 TransactionTemplate 的实现方式	148
7.3.3	JdbcTemplate 的实现方式	150
7.3.4	使用 JdbcTemplate 查询数据库	158
7.3.5	使用 JdbcTemplate 更改数据库	158
7.4	使用 ORM 工具访问数据.....	159
7.4.1	ORM 简述.....	159
7.4.2	使用 Hibernate	160
7.4.3	使用 iBatis	164
7.5	小结.....	167
第 8 章	Spring 的 Web 框架	168
8.1	Web 框架介绍	168
8.1.1	MVC 模式简介.....	168
8.1.2	MVC 模式的结构图.....	169
8.1.3	MVC 模式的功能示意图.....	169
8.1.4	使用 MVC 模式的好处.....	169
8.1.5	Model1 规范	170
8.1.6	Model2 规范	171
8.1.7	Spring MVC 的特点	171
8.2	一个在 JSP 页面输出 “HelloWorld” 的 Spring MVC 实例	172
8.2.1	配置 web.xml.....	173
8.2.2	编写实现输出的 JSP 页面 index.jsp	174
8.2.3	编写控制器 HelloWorldAction.java	174
8.2.4	配置 Spring 文档 dispatcherServlet-servlet.xml	175
8.2.5	启动 Tomcat	176
8.2.6	运行程序.....	177
8.2.7	把 index.jsp 改为使用 Jstl.....	177
8.2.8	修改配置文档 dispatcherServlet-servlet.xml	178
8.2.9	运行修改后的程序.....	179
8.2.10	使用 Log4j 时应该注意的问题	179
8.3	Spring MVC 的模型和视图 (ModelAndView)	180
8.3.1	模型和视图.....	180
8.3.2	Jstl 简介	182
8.3.3	视图解析.....	184
8.4	Spring MVC 的控制器 (Controller)	184

8.4.1	Controller 架构	185
8.4.2	表单控制器 (SimpleFormController)	185
8.4.3	多动作控制器 (MultiActionController)	189
8.5	Spring MVC 的分发器 (DispatcherServlet)	196
8.5.1	分发器的定义方式	196
8.5.2	分发器的初始化参数	197
8.5.3	分发器的工作流程	198
8.5.4	分发器与视图解析器的结合	199
8.5.5	在一个 Web 应用中使用不同的视图层技术	200
8.5.6	在 DispatcherServlet 中指定处理异常的页面	201
8.6	处理器映射	202
8.6.1	映射示例	202
8.6.2	映射原理	204
8.6.3	添加拦截器	205
8.7	数据绑定	209
8.7.1	绑定原理	209
8.7.2	使用自定义标签	210
8.7.3	Validator 应用	214
8.8	本地化支持	218
8.8.1	在头信息中包含客户端的本地化信息	219
8.8.2	根据 cookie 获取本地化信息	219
8.8.3	从会话中获取本地化信息	221
8.8.4	根据参数修改本地化信息	222
8.9	一个用 Spring MVC 实现用户登录的完整示例	224
8.9.1	在 Eclipse 中建立 Tomcat 工程项目 myMVC	224
8.9.2	编写日志文件放在 myMVC/WEB-INF/src 下	225
8.9.3	配置 web.xml	226
8.9.4	编写登录页面 login.jsp	227
8.9.5	编写显示成功登录的页面 success.jsp	227
8.9.6	编写存放用户登录信息的 Bean	228
8.9.7	编写用户输入信息验证 UserValidator.java	228
8.9.8	编写用户登录逻辑 Login.java	229
8.9.9	编写配置文件 dispatcherServlet-servlet.xml	230
8.9.10	运行程序	231
8.9.11	国际化支持	231
8.9.12	运行国际化后的程序	236
8.10	小结	237

第 9 章 Spring 的定时器	238
9.1 定时器简述	238
9.2 定时器的两种实现方式	238
9.2.1 Timer	239
9.2.2 Quartz	240
9.2.3 两种方式的比较	243
9.3 利用 Spring 简化定时任务的开发	243
9.3.1 在 Spring 中使用 Timer 实现定时器	243
9.3.2 在 Spring 中使用 Quartz 实现定时器	245
9.4 小结	246

第 3 篇 Spring 与其他工具整合应用

第 10 章 Spring 与 Struts 的整合	248
10.1 Struts 介绍	248
10.1.1 Struts 的历史	248
10.1.2 Struts 的体系结构	248
10.2 Struts 的下载和配置	249
10.2.1 下载 Struts	249
10.2.2 配置 Struts	249
10.3 一个在 JSP 页面输出“HelloWorld”的 Struts 实例	252
10.3.1 配置 web.xml	252
10.3.2 编写实现输出的 JSP 页面 index.jsp	253
10.3.3 编写控制器 HelloWorldAction.java	254
10.3.4 配置 Struts 文档 struts-config.xml	255
10.3.5 启动 Tomcat	255
10.3.6 运行程序	256
10.4 Struts 的几个主要类简介	256
10.4.1 ActionServlet (控制器)	256
10.4.2 Action (适配器)	259
10.4.3 ActionMapping (映射)	260
10.4.4 ActionForm (数据存储)	263
10.4.5 DispatchAction (多动作控制器)	266
10.5 国际化支持	269
10.6 Struts 的自定义标签	273
10.6.1 Bean 标签	274
10.6.2 Logic 标签	275
10.6.3 Html 标签	277

10.7 Spring 与 Struts 整合的 3 种方式.....	278
10.7.1 通过 Spring 的 ActionSupport 类.....	279
10.7.2 通过 Spring 的 DelegatingRequestProcessor 类	282
10.7.3 通过 Spring 的 DelegatingActionProxy 类	286
10.7.4 比较 3 种整合方式.....	289
10.8 采用第 3 种整合方式编写一个用户注册的例子	290
10.8.1 Spring 与 Struts 整合环境的配置.....	290
10.8.2 编写 web.xml.....	293
10.8.3 编写用户注册页面 regedit.jsp	293
10.8.4 编写用户注册成功页面 success.jsp	294
10.8.5 编写用户类 User.java.....	294
10.8.6 编写 Struts 的配置文件 struts-config.xml	295
10.8.7 编写 Spring 的配置文件 config.xml.....	296
10.8.8 编写控制器 RegeditAction.java.....	296
10.8.9 编写业务逻辑接口 Regedit.java.....	297
10.8.10 编写具体的业务逻辑类 RegeditImpl.java	297
10.8.11 运行用户注册的例子.....	298
10.9 小结.....	299
第 11 章 Spring 与 Hibernate 的整合	300
11.1 Hibernate 介绍.....	300
11.2 Hibernate 的下载和配置	300
11.2.1 下载 Hibernate.....	300
11.2.2 配置 Hibernate.....	301
11.3 一个实现数据新增的 Hibernate 示例	302
11.3.1 建立数据库表.....	303
11.3.2 编写表对应的 POJO	304
11.3.3 编写 POJO 对应的 XML	305
11.3.4 编写 Hibernate 的配置文件	306
11.3.5 编写测试案例.....	306
11.3.6 运行测试程序.....	307
11.4 Hibernate 的配置.....	308
11.5 Hibernate 的映射	310
11.5.1 集合映射.....	311
11.5.2 组件映射.....	321
11.5.3 关联映射.....	323
11.5.4 继承映射.....	346
11.6 Hibernate 的工具.....	350

11.6.1	从数据库到映射	350
11.6.2	从映射到 POJO	358
11.7	Hibernate 的几个主要类简介	360
11.7.1	Configuration (管理 Hibernate)	360
11.7.2	SessionFactory (创建 Session)	361
11.7.3	Session (提供 Connection)	361
11.8	通过 XML 来整合 Spring 和 Hibernate	363
11.9	整合 Struts、Spring 和 Hibernate 实现用户注册的示例	364
11.9.1	Spring、Struts 和 Hibernate 整合环境的配置	365
11.9.2	编写 web.xml	368
11.9.3	编写用户注册页面 regedit.jsp	369
11.9.4	编写用户注册成功页面 success.jsp	369
11.9.5	建立数据库表结构	369
11.9.6	生成映射文件 User.hbm.xml 和 POJO	370
11.9.7	编写接口 UserDAO.java 和实现类 UserDAOImpl.java	372
11.9.8	编写 Struts 的配置文件 struts-config.xml	373
11.9.9	编写 Spring 的配置文件 config.xml	374
11.9.10	编写控制器 RegeditAction.java	376
11.9.11	编写业务逻辑接口 Regedit.java	376
11.9.12	编写具体的业务逻辑类 RegeditImpl.java	377
11.9.13	运行用户注册的例子	377
11.10	小结	379
第 12 章	在 Spring 中使用 Ant	380
12.1	Ant 介绍	380
12.2	Ant 的下载和安装	380
12.2.1	下载 Ant	380
12.2.2	安装 Ant	381
12.3	在 Spring 中使用 Ant	382
12.3.1	在 Eclipse 中配置 Ant	382
12.3.2	建立 build.xml	382
12.3.3	运行 Ant	384
12.4	小结	384
第 13 章	在 Spring 中使用 Junit	385
13.1	Junit 介绍	385
13.2	Junit 的下载和安装	386
13.2.1	下载 Junit	386
13.2.2	安装 Junit	386

13.3 在 Spring 中使用 Junit	386
13.3.1 在 Eclipse 中配置 Junit	387
13.3.2 扩展 TestCase 类	387
13.3.3 运行 Junit	388
13.4 使用 Junit 时常用的一些判定方法	389
13.5 利用 Ant 和 Junit 进行自动化测试	389
13.5.1 修改 build.xml	389
13.5.2 运行 Ant	391
13.5.3 自动生成测试报告	392
13.6 小结	393

第 4 篇 Spring 实例

第 14 章 用 Spring 实现新闻发布系统实例	396
14.1 新闻发布系统的介绍	396
14.2 检查环境配置	396
14.2.1 检查 JDK 配置	396
14.2.2 检查 Tomcat 配置	396
14.2.3 检查 Ant 配置	397
14.3 在 Eclipse 下建立项目 myNews	398
14.3.1 在 Eclipse 下建立项目	398
14.3.2 编写 Ant build 文件	401
14.3.3 配置 web.xml 文件	402
14.4 设计新闻发布系统	403
14.4.1 设计页面	403
14.4.2 设计持久化类	410
14.4.3 设计数据库	415
14.4.4 新闻发布系统在持久层的整体 UML 图	416
14.5 编写新闻发布系统的 JSP 页面	416
14.5.1 新闻发布的展示页面 show.jsp	416
14.5.2 发布新闻页面 release.jsp	418
14.5.3 用户注册页面 regedit.jsp	419
14.5.4 管理员登录页面 login.jsp	420
14.5.5 错误处理页面 error.jsp	421
14.6 建立数据库表并生成 XML 和 POJO	422
14.6.1 存放用户信息的数据库表	423
14.6.2 存放新闻的数据库表	424
14.6.3 存放新闻类别的数据库表	426

14.6.4	存放用户权限的数据库表.....	427
14.6.5	建立表之间的关系.....	429
14.6.6	生成对应的 XML.....	432
14.6.7	生成 POJO.....	443
14.7	编写新闻发布系统的 VO 和 DAO.....	450
14.7.1	用户类 User.java.....	450
14.7.2	用户权限类 UsersAuthor.java.....	452
14.7.3	新闻类 News.java.....	452
14.7.4	新闻类别类 NewsType.java.....	454
14.7.5	用户 DAO 接口 UserDAO.java.....	454
14.7.6	新闻 DAO 接口 NewsDAO.java.....	455
14.7.7	新闻类别 DAO 接口 NewsTypeDAO.java.....	455
14.7.8	用户 DAO 实现类 UserDAOImpl.java.....	456
14.7.9	新闻 DAO 实现类 NewsDAOImpl.java.....	457
14.7.10	新闻类别 DAO 实现类 NewsTypeDAOImpl.java.....	458
14.8	编写新闻发布系统的业务逻辑类.....	459
14.8.1	登录接口 Login.java.....	459
14.8.2	注册接口 Regedit.java.....	460
14.8.3	发布接口 Release.java.....	460
14.8.4	登录实现类 LoginImpl.java.....	460
14.8.5	注册实现类 RegeditImpl.java.....	461
14.8.6	发布实现类 ReleaseImpl.java.....	462
14.9	编写新闻发布系统的控制器类.....	464
14.9.1	登录控制器类 LoginController.java.....	464
14.9.2	注册控制器类 RegeditController.java.....	466
14.9.3	发布控制器类 ReleaseController.java.....	467
14.9.4	显示控制器类 ShowController.java.....	469
14.10	编写辅助类 NewsUtil.java.....	471
14.11	编写配置文件 dispatcherServlet-servlet.xml.....	475
14.12	运行验证程序.....	480
14.13	小结.....	484

第 1 篇



Spring 入门

第 1 章 Spring 概述

第 2 章 开始 Spring 之旅



第 1 章 Spring 概述

可以说，Spring 是目前最引人注目的一个开源框架。它是一个轻量级的 J2EE 应用程序框架，实现了 IoC 模式。本章主要对 Spring 进行简单介绍，在后面的章节里将会对其进行详细全面的介绍。

1.1 Spring 的历史

企业应用软件的开发变得越来越庞大，软件技术也越来越复杂。为此，软件开发人员常常忙于对技术的研发和应用，而忽略了对业务本身复杂性的分析。为了减少软件开发人员花费在软件技术上的时间，而把精力放在对业务本身的理解，市场上出现了很多解决这个问题应用框架，而 Spring 无疑是其中最优秀的一个。

从 2003 年 1 月开始，Spring 落户于 SourceForge，它最初起源于 2002 年 Rod Johnson 出版的《Expert One-on-One J2EE 设计与开发》书中的代码。这本书阐述了 Spring 的架构思想，读者可以仔细钻研此书，以了解 Spring 框架背后所展示的思想。

Rod Johnson 后来又写了一本 *Expert One on one J2EE Development Without EJB* 的书，便是告诉读者使用 Spring 就可以代替 EJB 了。

1.2 Spring 简介

从 Spring 诞生之日起，就引起了广大 Java 开发人员的强烈关注。

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。该框架的主要优势是其良好的设计和分层架构，软件开发人员可以只选择 Spring 提供的某项技术，例如 AOP，而不需要使用它提供的其他技术。同时，Spring 还提供了和其他开源软件的无缝结合，为 J2EE 应用程序开发提供了集成的框架。

1.2.1 Spring 框架介绍

Spring 是一个提供了解决 J2EE 问题的一站式框架。

Spring 的核心是控制反转，通过配置文件来完成业务对象之间的依赖注入，它鼓励一个良好的习惯，那就是注入对接口编程而不是对类编程。

Spring 还提供了事务处理的功能，它能够在各种底层事务处理技术上提供一个统一的编

程模型。

Spring 提供了一个简单而有效的 JDBC 应用。不但如此，它还能和其他一些开源框架进行无缝结合。

Spring 还提供了一个强大而灵活的 Web 框架，它同样是基于控制反转的。

Spring 框架由 7 个模块组成，如图 1.1 所示。

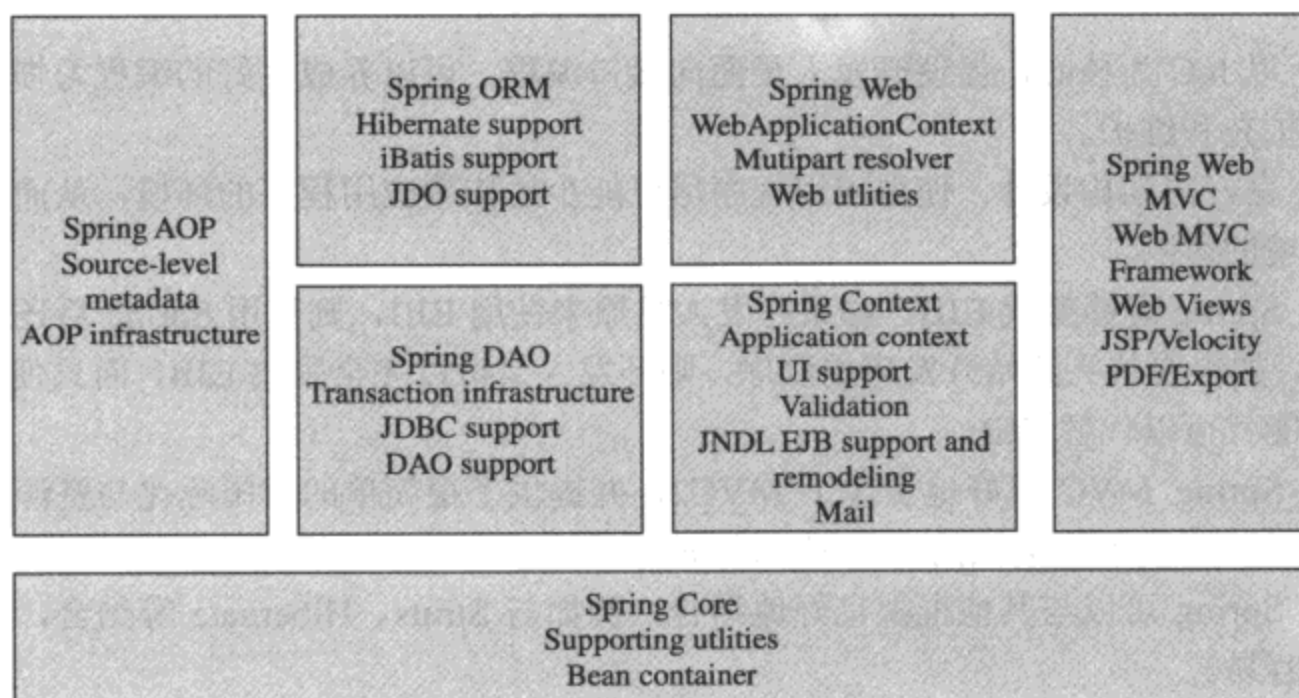


图 1.1 Spring 框架图

下面介绍 Spring 框架每个模块的功能。

(1) 核心容器：提供了 Spring 框架的核心功能。BeanFactory 是 Spring 核心容器的主要组件。它通过控制反转将应用程序的配置和依赖性规范与实际的应用程序代码分开，这是整个 Spring 的基础。

(2) Spring Context：通过配置文件，向 Spring 框架提供上下文信息。它构建在 BeanFactory 之上，另外增加了国际化、资源访问等功能。

(3) Spring AOP：Spring 提供了面向方面编程的功能，因为 Spring 的核心是基于控制反转的，所以可以很容易地使 Spring 的依赖注入为 AOP 提供支持。

(4) Spring DAO：提供了一个简单而又有效的 JDBC 应用，使用它的 DAO 就足以应付开发人员的日常应用了。

(5) Spring ORM：Spring 除了有自己的 JDBC 应用之外，还提供了对其他一些 ORM 框架的支持，例如 JDO、Hibernate 和 iBatis 等。基于 Spring 的良好设计，这些开源框架都可以和 Spring 进行良好的结合。

(6) Spring Web：提供了简化的处理多部分请求以及将请求参数绑定到域对象的任务。

(7) Spring MVC：Spring 提供了 MVC2 模式的实现，使用起来非常方便，但它不强迫开发人员使用。如果开发人员对其他的 MVC 框架比较熟悉，仍然可以使用它们。Spring 对此提供了很好的支持，例如 Spring 可以和 Struts 集成在一起。

1.2.2 Spring 的特点

Spring 之所以能迅速在 Java 开发人员中流行，这是因为 Spring 具有以下特点：

(1) 设计良好的分层结构，使得开发人员可以很简单地进行扩充，并引入先进的设计理念。

(2) 以 IoC 为核心，促使开发人员面向接口编程，可以养成良好的编程习惯，从而便于程序的扩充和维护。

(3) 良好的架构设计，使得应用程序尽可能少地依赖应用程序的环境，从而使得应用脱离了环境的影响。

(4) Spring 能够替代 EJB。如果开发人员原来使用 EJB，则使用 Spring 后还可以继续使用 EJB，如果要从头开始开发应用程序，则开发人员可以完全脱离 EJB，而只使用 Spring 提供的功能就可以代替 EJB。

(5) Spring MVC 很好地实现了 MVC2，并提供了很简单的对国际化与资源访问的支持，而且可以和 Spring 提供的 IoC 和 AOP 联系起来。

(6) Spring 可以与其他框架良好地结合，例如与 Struts、Hibernate 等结合，这使应用开发更为容易。

1.3 如何学习 Spring

要学习 Spring，需要把握以下几点：

(1) 首先根据本书或相关文章的介绍编写一个使用 Spring 实现的应用程序，以增强对 Spring 的感性认识。例如 Spring 网站提供了一个关于 Spring MVC 的示例，网址为 <http://www.springframework.org/docs/MVC-step-by-step/Spring-MVC-step-by-step.html>，可以根据上面的介绍一步一步地实现这个示例。

(2) 注重理解 IoC 的基本概念和思想，明白 IoC 的内涵，因为 Spring 的核心就是建立在 IoC 基础之上的。推荐看看 Martin Fowler 写的文章——控制反转和依赖注入模式。

(3) 从 Spring 对 Bean 的管理开始，理解 Bean 在 Spring 中的处理方式和 Bean 在 Spring 文件中的配置方式，进而学习 Spring 对国际化的支持和资源访问等基础知识。

(4) 学习 Spring AOP 的实现方式，更重要的是学习 AOP 的理念。

(5) 学习 Spring 的持久层管理和对事务处理的方式。Spring 提供了自己的持久层管理框架，但也与其他的持久层框架进行了很好的整合。

(6) Spring MVC 框架和 Struts 框架一样实现了 MVC2 模式，如果对 Struts 有一定了解，学习 Spring MVC 会比较简单。

(7) 学习 Spring 最重要的是从实例入手，每学习一个新的概念或思想，都要通过实例来实现它，从而使得开发人员对该项概念或思想有更深刻的认识。

1.4 网络上的资源

(1) 学习 Spring 最重要的网站:

<http://www.springframework.org>, 有关 Spring 的最新信息都会在该网站列出。

(2) 笔者学习 Spring 时看到的第一个中文指南:

http://www.xiaxin.net/Spring_Dev_Guide.rar

(3) 笔者学习 Spring MVC 时看到的第一个培训教材, 指导开发人员一步一步地实现 Spring 的 MVC:

<http://www.springframework.org/docs/MVC-step-by-step/Spring-MVC-step-by-step.html>

(4) Spring 开发手册中文版:

<http://www.jactiongroup.net/reference/html/index.html>

(5) Spring 中文论坛:

<http://spring.jactiongroup.net/index.php>

(6) martinowler 写的“控制反转和依赖注入模式”:

<http://www.martinfowler.com/articles/injection.html>

(7) Java 视线论坛:

<http://www.hibernate.org.cn/>

(8) 一个我国台湾省高手写的 Spring 教程:

<http://www.javaworld.com.tw/confluence/display/opensource/Spring>

(9) 通过 AppFuse 源代码来学习 Spring:

<http://raibledesigns.com/wiki/Wiki.jsp?page=AppFuse>

(10) 一个学习 Java 的好地方, 里面也有很多 Spring 的教程:

<http://www.javafan.net>

(11) 里面的文章对学习任何技术都很有帮助:

<http://www.csdn.net>

(12) IBM developerworks 中国网站:

<http://www-128.ibm.com/developerworks/cn/java/>

(13) Spring 框架完全进阶专题中国 IT 实验室:

<http://www.chinaitlab.com/www/techspecial/spring/>

1.5 小 结

本章简单介绍了 Spring 的历史、特点和学习方法, 并提供了网上的一些关于 Spring 的学习资源。在后面的章节里, 将会对其进行详细而全面的介绍。

第 2 章 开始 Spring 之旅

学习一门新的技术，最好的方式就是从简单的实例开始，然后逐渐地涉及与其相关的概念和技术。在本章中，笔者首先会一步一步地帮助读者建立起 Spring 的开发环境，然后从一个实例入手，让读者快速地了解 Spring 的开发流程，感受 Spring 的与众不同。

2.1 建立 Spring 开发环境

对于初学者来说，可能最烦恼的事情就是环境配置的问题了，他们往往会在这上面浪费不少的时间。为了让读者快速了解 Spring，笔者将从零开始，一步一步地建立 Spring 的开发环境。

2.1.1 下载 JDK

任何人都可以从 Java 的官方网站 <http://java.sun.com/javase/downloads/> 自由下载 JDK。Java 5.0 版本的下载画面如图 2.1 所示。

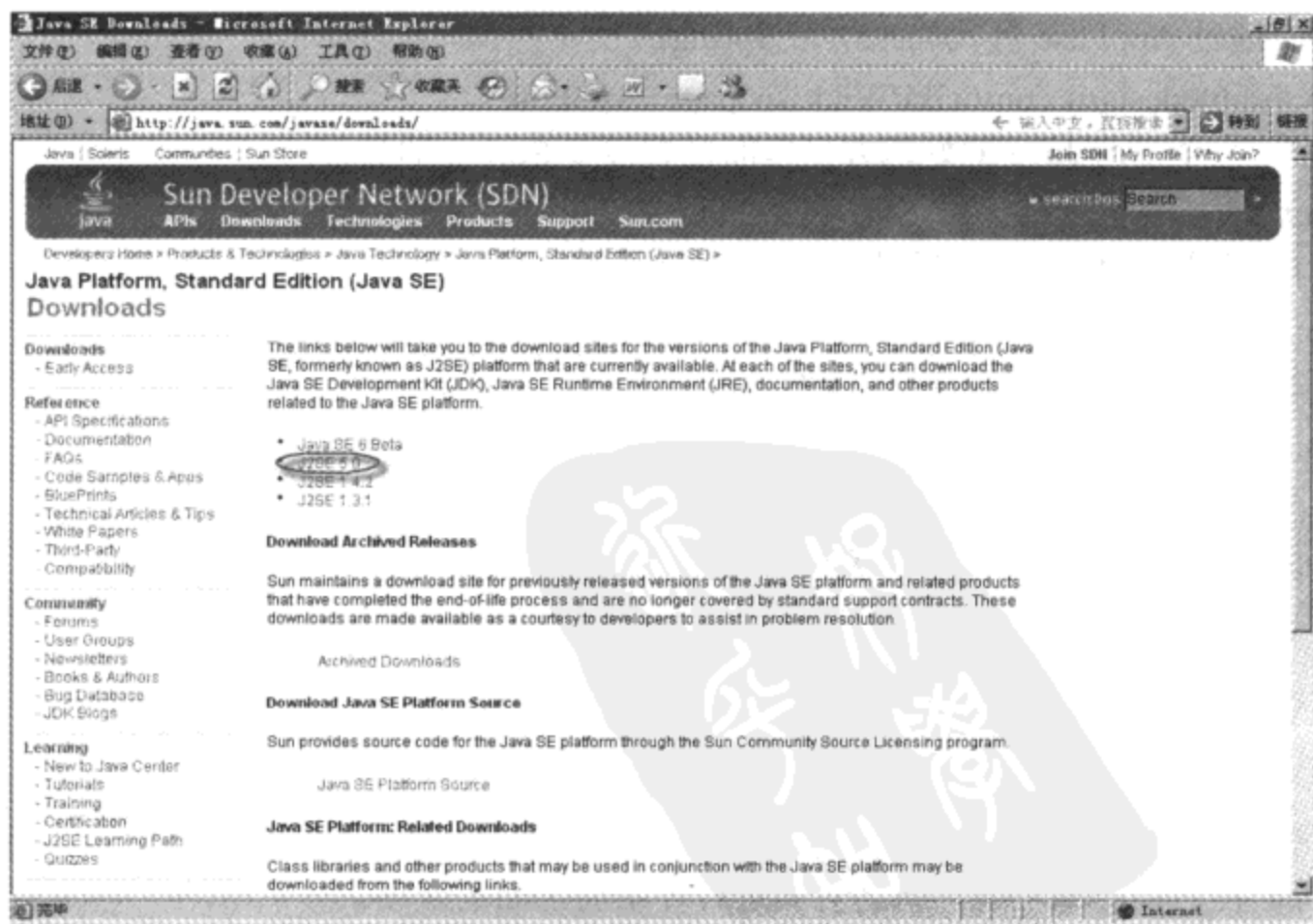


图 2.1 Java 5.0 版本的下载画面

下载结束后，硬盘中会生成名为 jdk-1_5_0_06-windows-i586-p 的可执行 JDK 安装程序。该程序的大小约为 59.86MB。

2.1.2 安装 JDK

双击执行 jdk-1_5_0_06-windows-i586-p.exe，便可自动解压缩进行安装。安装过程很简单，只要遵循画面指示即可。

说明：默认的安装路径是 C:\jdk1.5.0，当然读者可以修改这个路径。

2.1.3 设定 Path 与 CLASSPATH

JDK 安装完毕后，还要设定系统的环境变量 Path 与 CLASSPATH。步骤如下：

- (1) 在 Windows 桌面中，右击“我的电脑”。
- (2) 在弹出的快捷菜单中选择“属性”命令，弹出“系统属性”对话框。
- (3) 在“系统属性”对话框中，单击“高级”选项卡中的“环境变量”按钮，弹出“环境变量”对话框，如图 2.2 所示。
- (4) 单击“系统变量”选项区域中的“新建”按钮，在弹出的“编辑系统变量”对话框中设定系统变量 Path，Path=C:\jdk1.5.0\bin，如图 2.3 所示。



图 2.2 “环境变量”对话框

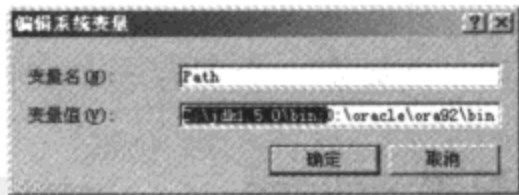


图 2.3 设定系统变量 Path

- (5) 单击“系统变量”选项区域中的“新建”按钮，在弹出的“编辑系统变量”对话框中设定系统变量 CLASSPATH，CLASSPATH=.;C:\jdk1.5.0\lib\dt.jar;C:\jdk1.5.0\lib\tools.jar，如图 2.4 所示。



图 2.4 设定系统变量 CLASSPATH

⚠注意: Path 与 Classpath 这两个变量不用区分大小写; 如果这两个变量在系统中已存在, 则在“环境变量”对话框中选中要设定的环境变量, 然后单击“编辑”按钮进行修改; 如果在系统中不存在, 则要单击“新建”按钮, 新建环境变量。

2.1.4 下载 Eclipse

在 <http://www.eclipse.org/downloads> 即可下载 Eclipse。Eclipse 3.1.2 版本的下载画面如图 2.5 所示。

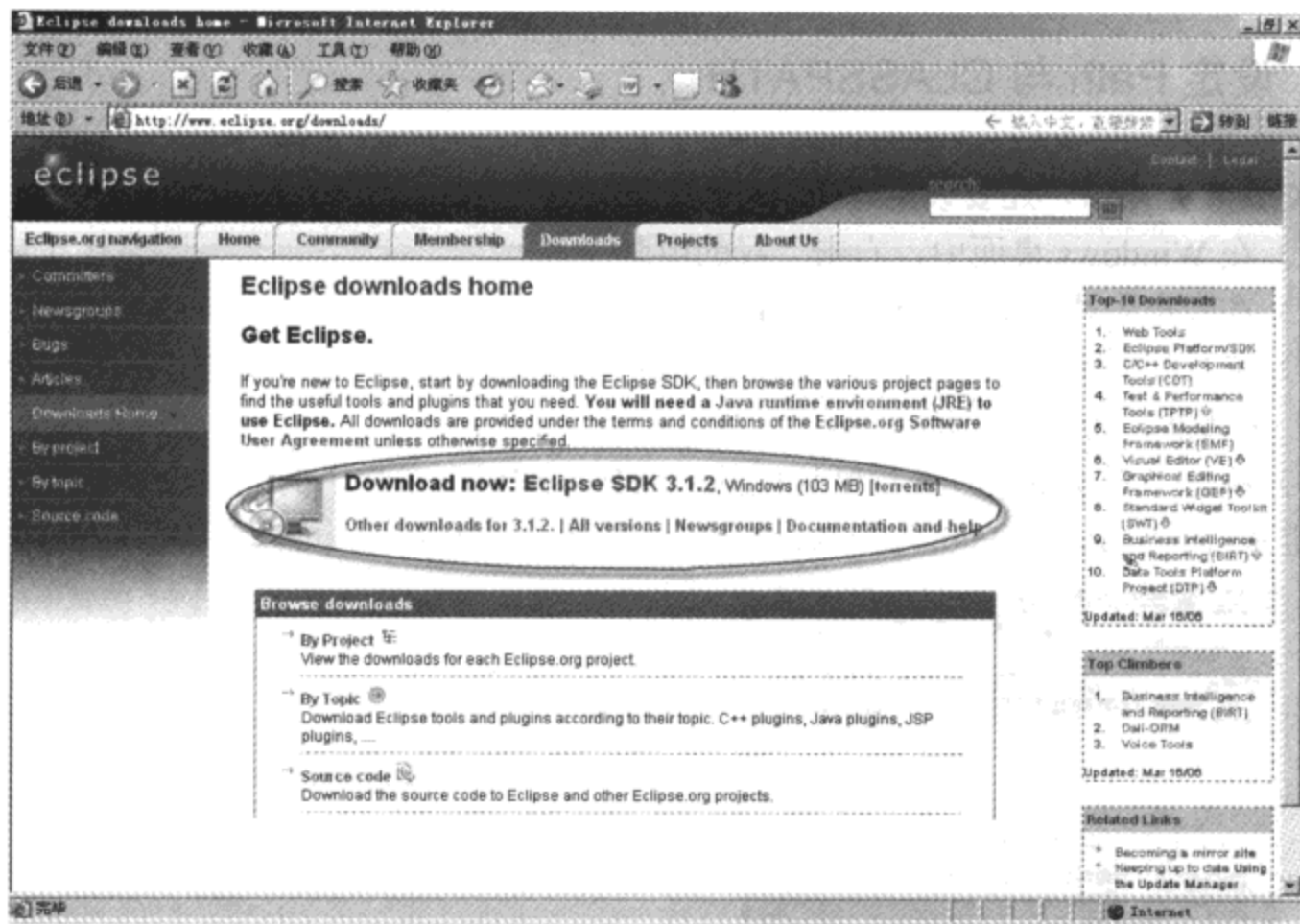


图 2.5 Eclipse 3.1.2 版本的下载画面

下载后, 在硬盘上会保存一个文件名为 eclipse-SDK-3.1.2-win32 的压缩文件。

2.1.5 配置 Eclipse

把上面下载的 eclipse-SDK-3.1.2-win32.zip 解压缩到 D 盘根目录下, 并修改文件夹名为 eclipse。

2.1.6 下载 Tomcat

在 <http://tomcat.apache.org> 即可下载到 Tomcat 的各种版本。Tomcat 5.5.17 版本的下载画面如图 2.6 所示。下载后, 在硬盘上会保存一个文件名为 apache-tomcat-5.5.17 的压缩文件。

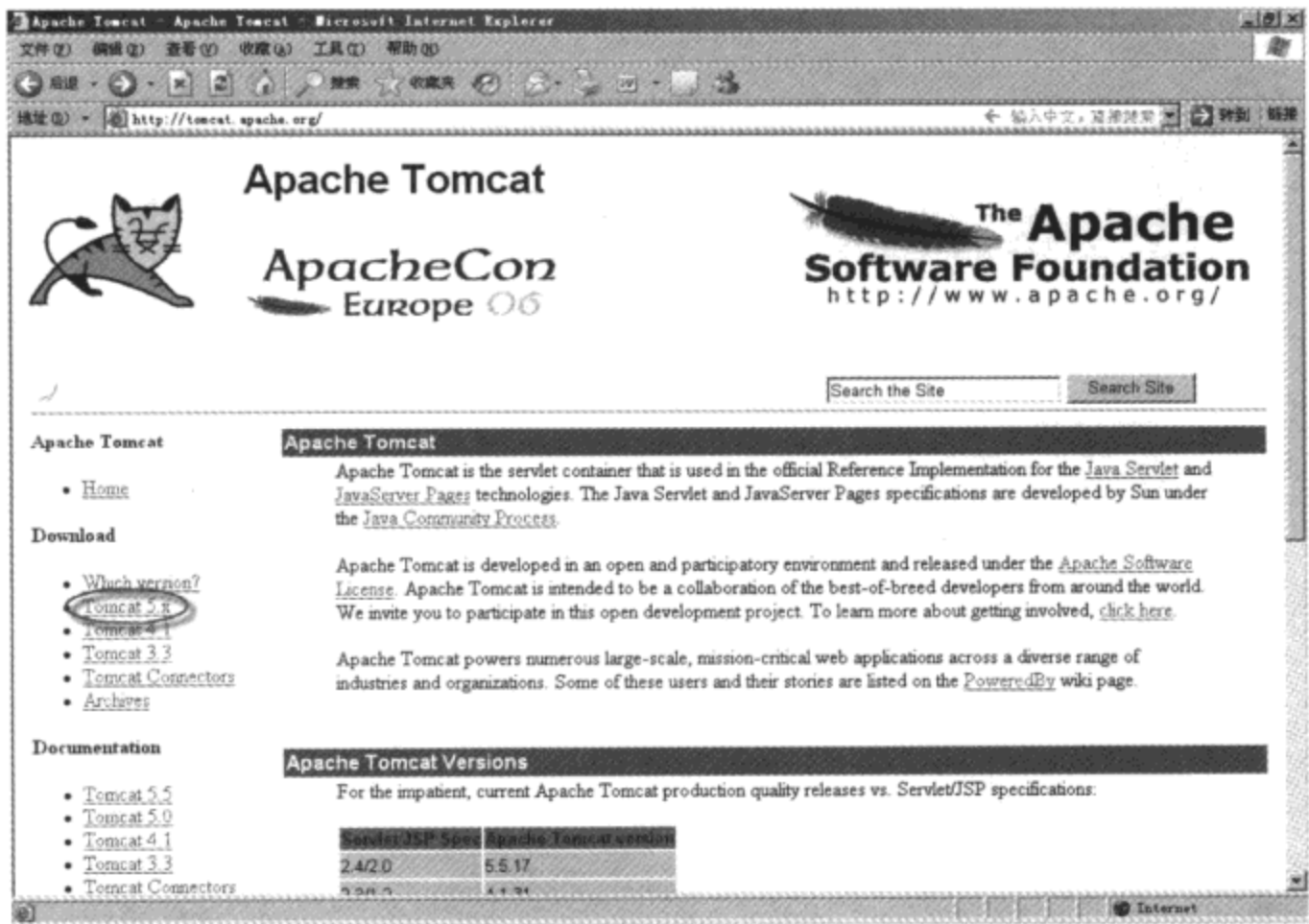


图 2.6 Tomcat 5.5.17 版本的下载画面

2.1.7 设定 TOMCAT_HOME

下载完 Tomcat 后，还要对其进行配置。其步骤如下：

(1) 把前面下载的 apache-tomcat-5.5.17.zip 解压缩到 D 盘根目录下，将会产生一个 apache-tomcat-5.5.17 文件夹。

(2) 用前面设定 JDK 的 Path 和 CLASSPATH 的方法，在系统环境变量中新建一个环境变量 TOMCAT_HOME，设定 TOMCAT_HOME= D:\jakarta-tomcat-5.5.17，如图 2.7 所示。

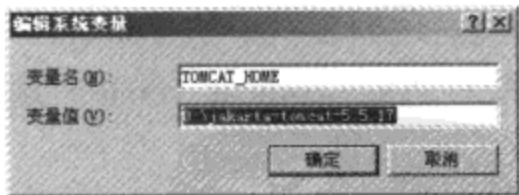


图 2.7 设定系统变量 TOMCAT_HOME

2.1.8 下载 Eclipse 的 Tomcat 插件

在 <http://www.sysdeo.com/sysdeo/eclipse/tomcatplugin> 可以下载 Tomcat 为 Eclipse 配置的插件。插件 tomcatPluginV31 的下载画面如图 2.8 所示。

下载后，在硬盘上会保存一个文件名为 tomcatPluginV31 的压缩文件。

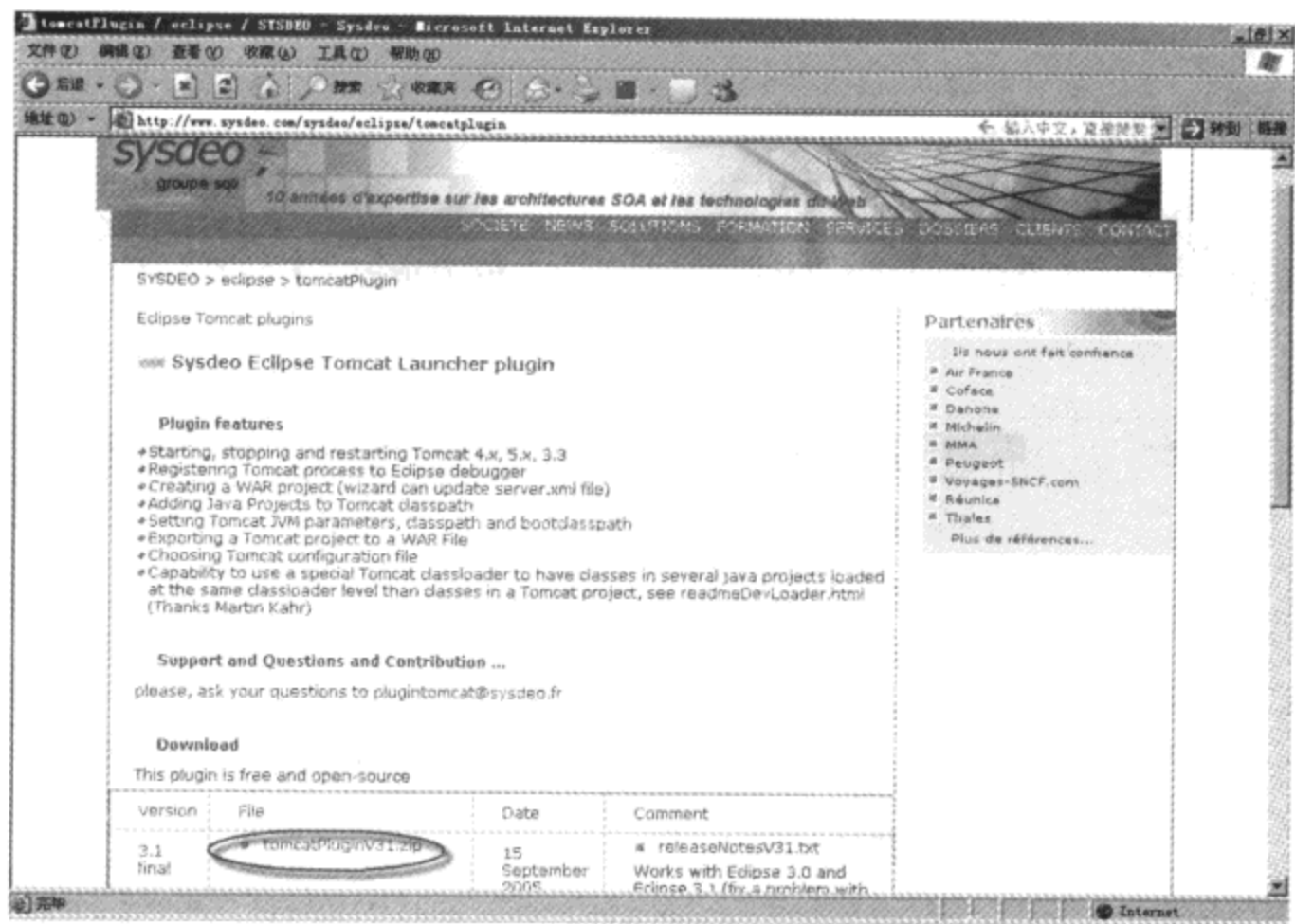


图 2.8 插件 tomcatPluginV31 的下载画面

2.1.9 为 Eclipse 配置 Tomcat 插件

将 Eclipse 和 Tomcat 联系起来。其步骤如下：

- (1) 将下载完毕的 tomcatPluginV31.zip 解压缩，会产生一个 tomcatPluginV31 文件夹。
- (2) 把 tomcatPluginV31 文件夹下的 com.sysdeo.eclipse.tomcat_3.1.0 文件夹复制到 D:\eclipse\plugins 下。
- (3) 双击 D 盘 Eclipse 文件夹下的 eclipse.exe 图标，运行 Eclipse。
- (4) 在 Eclipse 菜单栏中选择 Windows→Preferences 命令，弹出 Preferences 对话框。
- (5) 单击 Preferences 对话框左边列表框中的 Tomcat。
- (6) 在 Preferences 对话框右边的 Tomcat version 选项区域中选择 Tomcat 版本为 Version 5.x。
- (7) 在 Preferences 对话框右边的 Tomcat home 文本框中输入 D:\jakarta-tomcat-5.5.17。
- (8) 在 Preferences 对话框右边的 Context declaration mode 选项区域中，采用系统默认的 Server.xml，然后单击 OK 按钮即可。为 Eclipse 配置 Tomcat 插件的画面如图 2.9 所示。
- (9) 单击 Eclipse 的 Start Tomcat 图标，启动 Tomcat，如图 2.10 所示。
- (10) Tomcat 启动后，在 IE 地址栏中输入 <http://localhost:8080>，即可测试是否配置成功。Tomcat 启动成功的画面如图 2.11 所示。

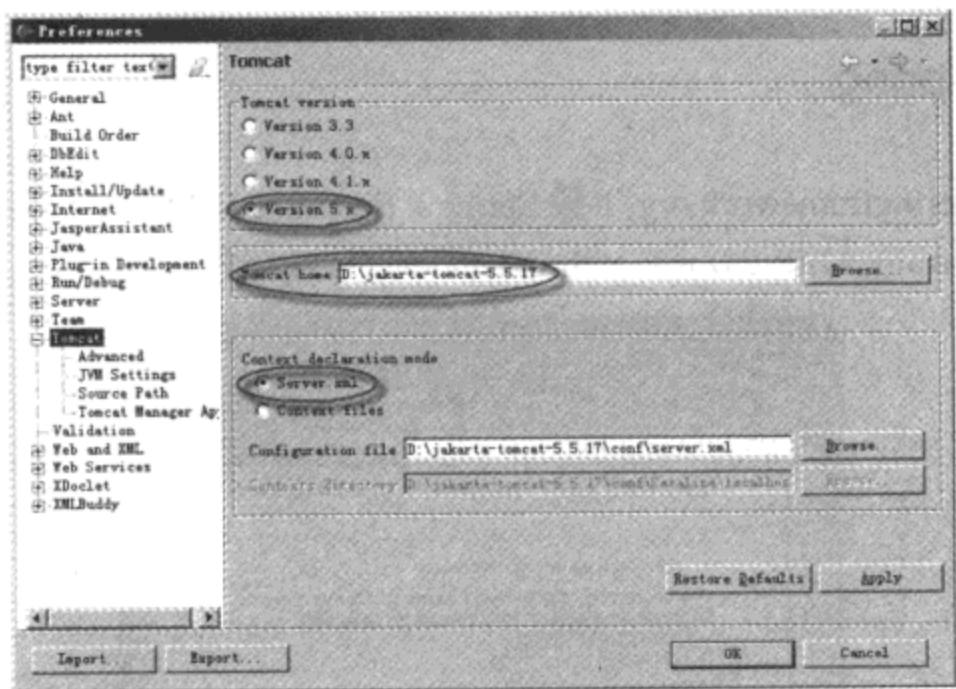


图 2.9 为 Eclipse 配置 Tomcat 插件

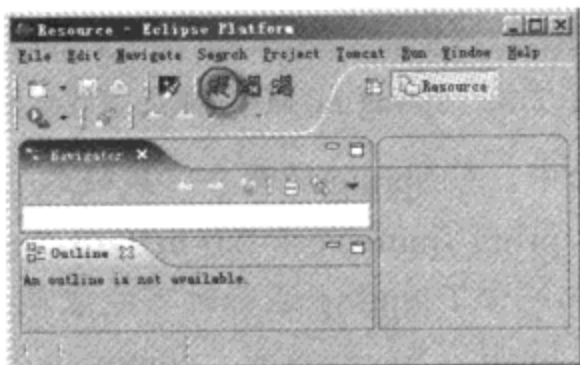


图 2.10 启动 Tomcat

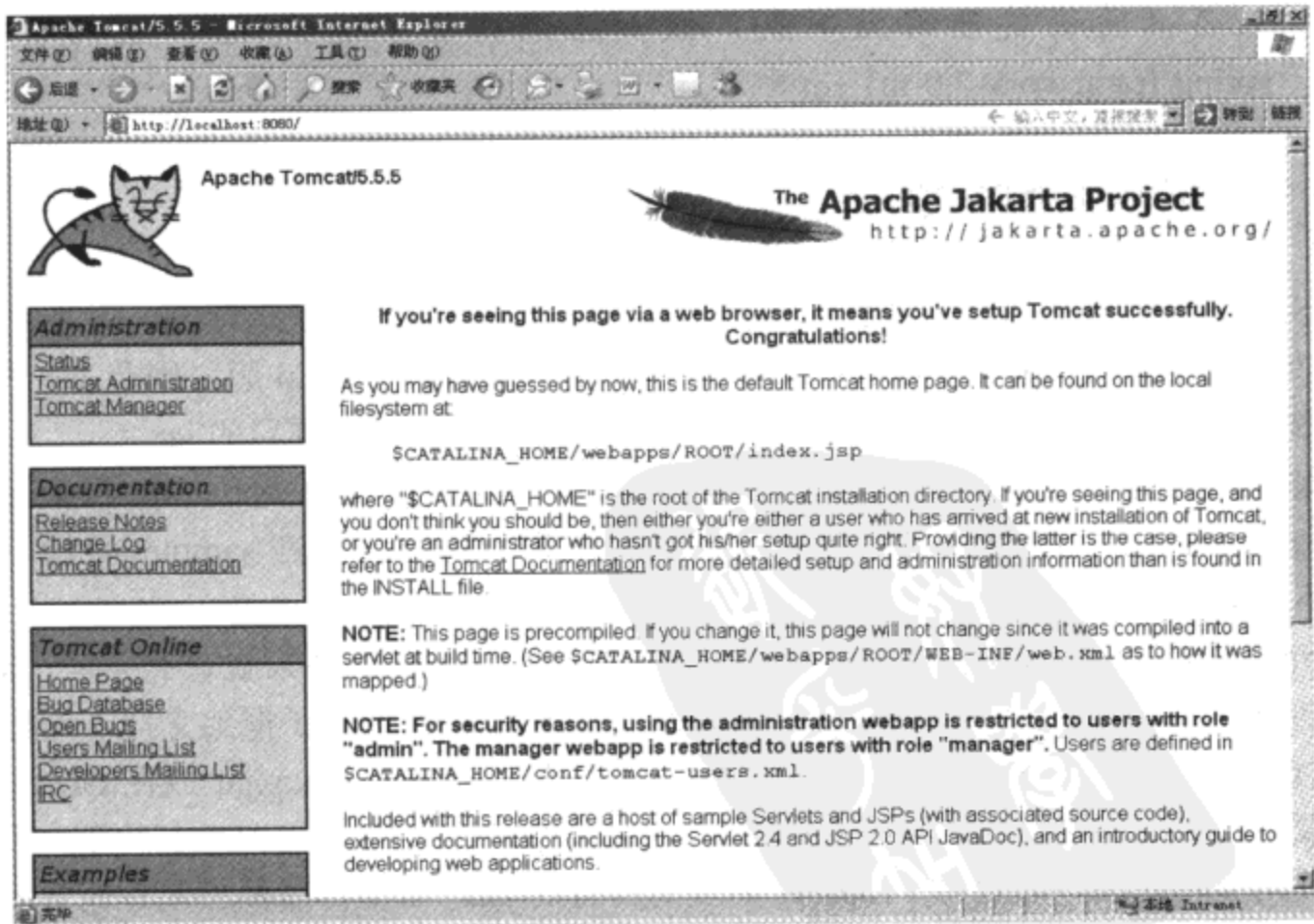


图 2.11 Tomcat 启动成功的画面

2.1.10 下载 Spring

从 <http://www.springframework.org> 下载 Spring。Spring 2.0 M1 版本的下载画面如图 2.12 所示。下载 `spring-framework-2.0-m1-with-dependencies.zip` 到本地硬盘。

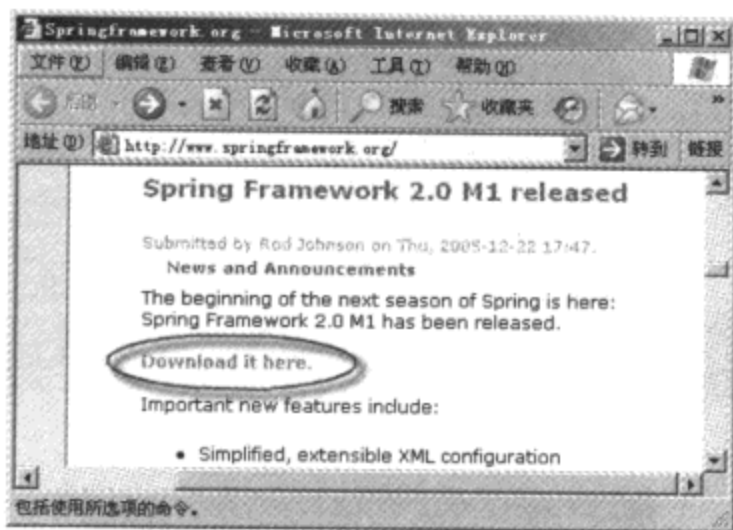



图 2.12 Spring 2.0 M1 版本的下载画面

 说明：在 Spring 的下载画面中，有 Spring 的两个下载版本：`spring-framework-2.0-m1-with-dependencies.zip` 和 `spring-framework-2.0-m1.zip`。`spring-framework-2.0-m1-with-dependencies.zip` 版本包含了 Spring 运行时可能依赖的一些其他开源项目的 jar，例如 `ant`、`hibernate`、`log4j`、`freemarker`、`commons-dbcp`、`dom4j`、`aopalliance` 和 `commons-logging` 等。这样，如果以后程序中需要这些 jar，直接使用就可以了，而不必再去其他网站下载。假如已经拥有了这些 jar，则只需要下载 `spring-framework-2.0-m1.zip` 即可。

2.1.11 简单介绍 Spring 包

当 Spring 下载完毕后，将其解压缩。在 `dist` 目录下存放的就是运行 Spring 所需要的相关 jar。如果下载的是 `spring-framework-2.0-m1-with-dependencies.zip` 版，则会多一个 `lib` 目录。在该目录下存放的是运行 Spring 时可能依赖的一些其他开源项目的 jar。

在解压缩后的 `spring-framework-2.0-m1-with-dependencies` 文件夹下，`src` 目录下存放的是 Spring 的源代码。有兴趣的读者可以钻研一下大师们写的代码。在 `samples` 目录下存放的是使用 Spring 的一些例子，例如 `jpetstore`、`petclinic` 等都在里面。

Spring 的核心是 `spring-core.jar`，对于编写简单的单机程序来说，把这个 jar 放在 `CLASSPATH` 下即可。如果以后程序中需要使用到 Spring 其他的子框架支持，再将其他的 jar 放在 `CLASSPATH` 下。例如，程序中需要 AOP 时，把 `spring-aop.jar` 放在 `CLASSPATH` 下；需要使用 Spring 的 MVC 时，把 `spring-webmvc.jar` 放在 `CLASSPATH` 下等。当然也可以直接使用 `spring.jar`，它包括了 Spring 需要的所有的 jar，而不再需要加入其他的 jar。

2.1.12 在 Eclipse 中配置 Spring

当 Spring 解压缩完毕后,就可以在 Eclipse 中配置 Spring 了。这里主要从 3 个方面讲解。首先在 Eclipse 中建立一个项目 myApp,然后把 Spring 相关的 jar 配置到该项目中,最后在项目中建立 3 个包。这 3 个包的作用分别为:com.gc.action 用来存放实体类,com.gc.impl 用来存放接口类,com.gc.test 用来存放测试类。具体步骤如下:

(1) 运行 Eclipse,选择 File→New→Project 命令,弹出 New Project 对话框,如图 2.13 所示。

(2) 选择列表框中 Java 下的 Tomcat Project,然后单击 Next 按钮,弹出 New Tomcat Project 对话框,如图 2.14 所示。

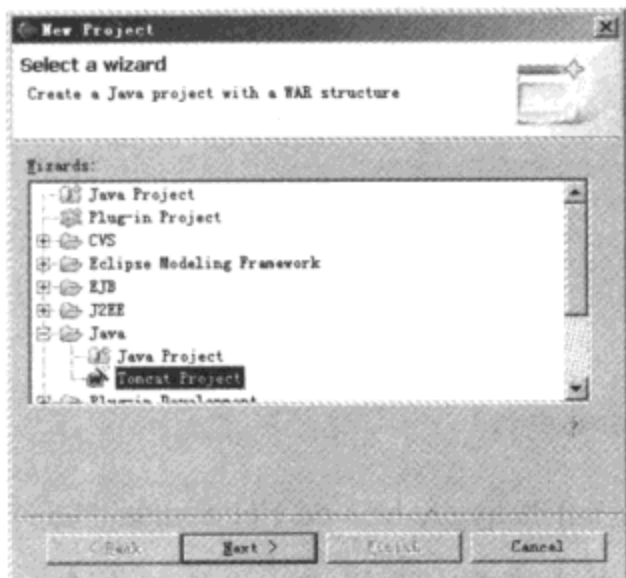


图 2.13 New Project 对话框

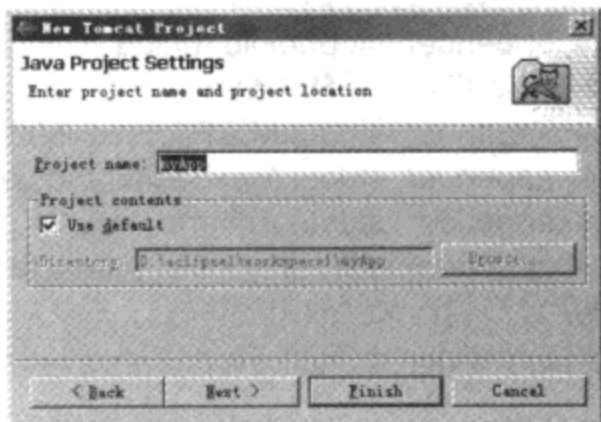


图 2.14 New Tomcat Project 对话框

(3) 在 New Tomcat Project 对话框中的 Project name 文本框中输入“myApp”,然后单击 Finish 按钮,项目即可建立成功。myApp 的目录结构如图 2.15 所示。

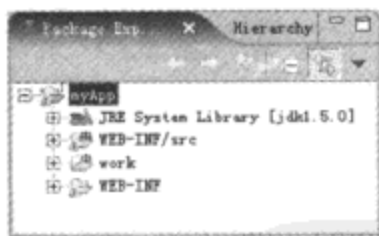


图 2.15 myApp 的目录结构

(4) 把 spring.jar 复制到 myApp/WEB-INF/lib 目录下,即 CLASSPATH 中。

(5) 在程序中需要 jakarta 的日志元件,所以把 spring-framework-2.0-m1/lib/jakarta-commons 下的 commons-logging.jar 复制到 myApp/WEB-INF/lib 目录下。

(6) 把 spring-framework-2.0-m1/lib/log4j 下的 log4j-1.2.9.jar 复制到 myApp/WEB-INF/lib 目录下。

(7) 用 Windows 自带的文本编辑器建立一个文件 log4j.properties,把 log4j.properties 放到 myApp/WEB-INF/src 目录下。

(8) 编辑 log4j.properties 文件，内容如下：

```
log4j.rootLogger=DEBUG,stdout

log4j.logger.org=ERROR, A1
#定义 log4j 的显示方式
log4j.appender.A1=org.apache.log4j.RollingFileAppender
#指定日志输入文件的名称
log4j.appender.A1.File=org.log
#指定日志输入文件的大小
log4j.appender.A1.MaxFileSize=500KB
log4j.appender.A1.MaxBackupIndex=50
log4j.appender.A1.Append=true
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
#指定日志输入文件的内容格式
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
#定义 log4j 的显示方式
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
#指定日志输入文件的名称
log4j.appender.stdout.File=gf.log
#指定日志输入文件的大小
log4j.appender.stdout.MaxFileSize=500KB
log4j.appender.stdout.MaxBackupIndex=50
log4j.appender.stdout.Append=true
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
#指定日志输入文件的内容格式
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
```

(9) 在 myApp 上单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，将弹出 Properties for myApp 对话框，如图 2.16 所示。

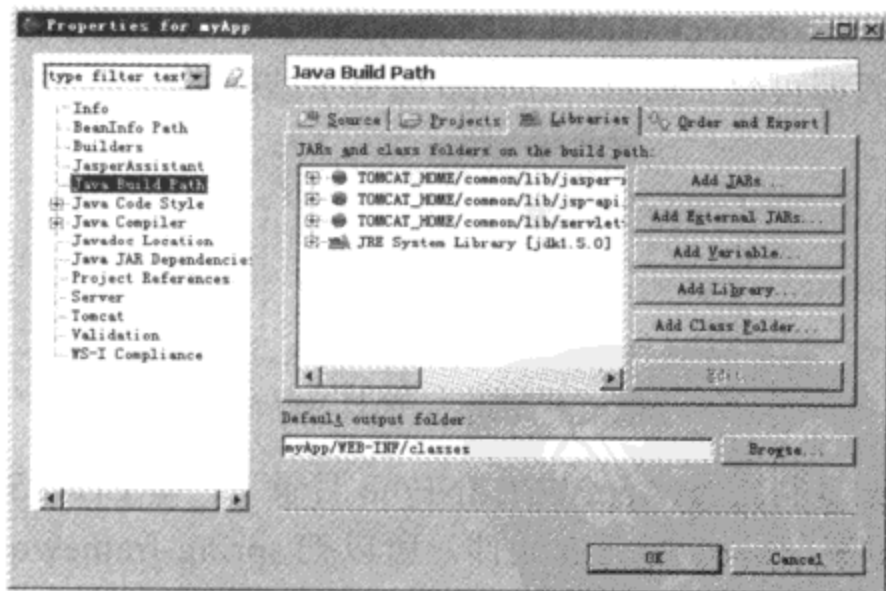


图 2.16 Properties for myApp 对话框

(10) 在 Properties for myApp 对话框中选择左边列表框中的 Java Build Path。

(11) 选择 Properties for myApp 对话框右边的 Libraries 选项卡。

(12) 在 Libraries 选项卡中，单击 Add JARs...按钮，弹出 JAR Selection 对话框，如图 2.17 所示。

(13) 在 JAR Selection 对话框中, 打开列表框 myApp 一直到 lib 目录下出现 3 个 jar, 即 spring.jar、commons-logging.jar、log4j-1.2.9.jar。

(14) 按住 Ctrl 键, 选中这 3 个 jar, 然后单击 OK 按钮, 返回到 Properties for myApp 对话框。

(15) 在 Properties for myApp 对话框中单击 OK 按钮, 即可完成对 Spring 的配置。

(16) 再次在 myApp 上单击鼠标右键, 在弹出的快捷菜单中选择 New→Package 命令, 将弹出 New Java Package 对话框, 如图 2.18 所示。

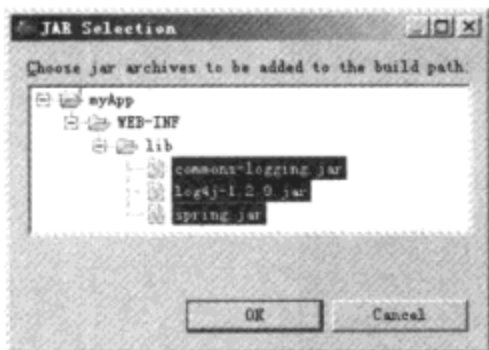


图 2.17 JAR Selection 对话框

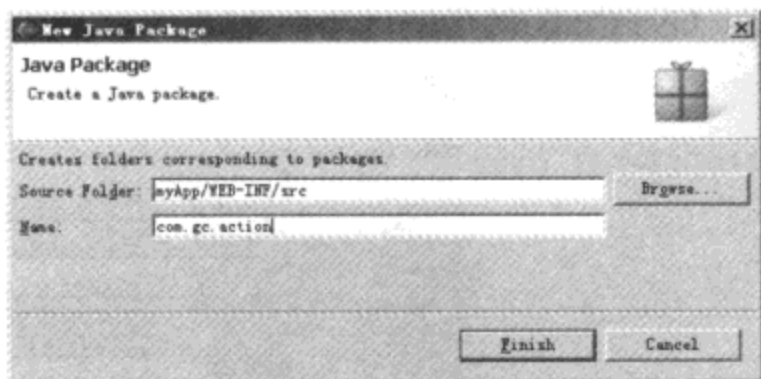


图 2.18 New Java Package 对话框

(17) 在 New Java Package 对话框中的 Name 文本框中输入 com.gc.action, 然后单击 Finish 按钮, 即可建立 com.gc.action 包。

(18) 用同样的方法建立 com.gc.impl 包和 com.gc.test 包。

(19) 最终配置好 Spring 的 myApp 项目的目录结构如图 2.19 所示。

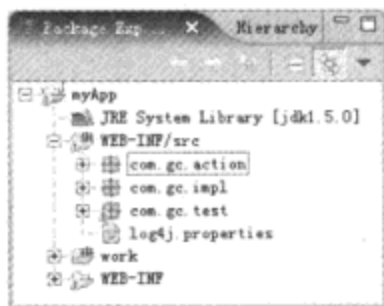


图 2.19 配置好 Spring 的 myApp 项目的目录结构

至此, 一个简单的 Spring 运行环境配置完毕。假如在后面的项目中还需要用到 Spring 所依赖的其他开源项目的 jar, 可以用上面的方法随时添加到 myApp/WEB-INF/lib 下。

说明: 在 Eclipse 中, WEB-INF/src、WEB-INF/lib、WEB-INF/classes 都是 CLASSPATH, 程序编译后 myApp/WEB-INF/src 目录下的文件会复制到 myApp/WEB-INF/classes 目录下。

2.2 第一个使用 Spring 实现 HelloWorld 的例子

既然是编写程序, 那就从 HelloWorld 开始吧。接下来笔者将使用 Spring 来编写一个 HelloWorld 程序, 通过这个程序来展示 Spring 与众不同的一个功能。编写一个简单的 Spring

程序分 3 步：首先编写 JavaBean，在本例中即为 HelloWorld.java。接下来配置 XML，在本例中即为 config.xml。最后编写测试程序，在本例中即为 TestHelloWorld.java。

2.2.1 编写存放 HelloWorld 的 Java 文件 HelloWorld.java

这个 JavaBean 用来存储将要展示的 HelloWorld。编写步骤如下：

(1) 在 com.gc.action 包上单击鼠标右键，在弹出的快捷菜单中选择 New→Class 命令，弹出 New Java Class 对话框，如图 2.20 所示。

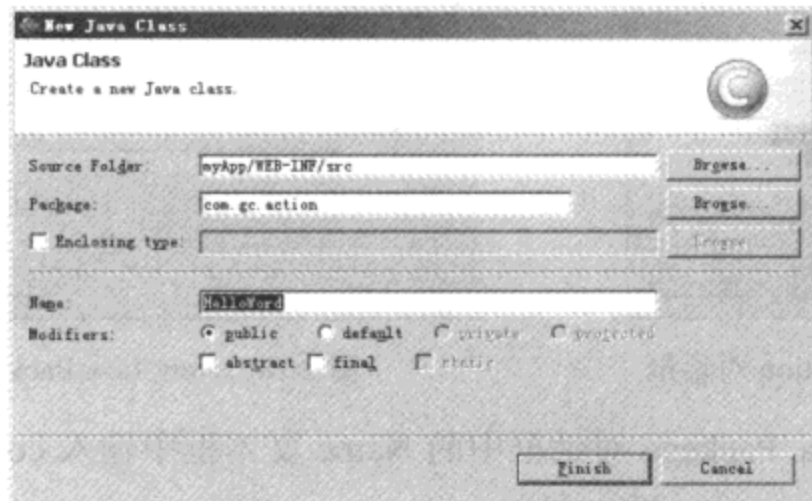


图 2.20 New Java Class 对话框

(2) 在 New Java Class 对话框中的 Name 文本框中输入“HelloWorld”，其他都采用默认值，然后单击 Finish 按钮，Eclipse 即在 com.gc.action 包下建立了一个 HelloWorld.java 文件。

(3) 在 HelloWorld.java 中输入如下代码：

```
//*****HelloWorld.java*****
package com.gc.action;
public class HelloWorld {
    //该变量用来存储字符串
    public String msg = null;
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
}
```

2.2.2 配置 Spring 的 config.xml

在上面编写的 HelloWorld.java 中，有 set 和 get 两个方法，用来负责对字符串的存取。

通常存取的代码如下：

```
//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.HelloWorld;
public class TestHelloWorld {
    public static void main(String[] args) {
        HelloWorld HelloWorld = new HelloWorld();
        //利用 set 方法将 HelloWorld 注入程序中
        HelloWorld.setMsg("HelloWorld");
        //利用 get 方法获取刚才注入的 HelloWorld
        System.out.println(HelloWorld.getMsg());
    }
}
```

可是有了 Spring 后，就不需要再通过程序来存取字符串了。Spring 提供了通过 XML 来定义字符串的功能。编写步骤如下：

- (1) 使用 Windows 自带的文本编辑器来建立 config.xml 文件。
- (2) 配置 config.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义一个 Bean-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">
        <!--将其变量 msg 通过依赖注入-->
        <property name="msg">
            <value>HelloWorld</value>
        </property>
    </bean>
</beans>
```

代码说明：

- ❑ id="HelloWorld"，用来唯一表示该 Bean。
- ❑ class="com.gc.action.HelloWorld"，用来表示该 Bean 的来源。
- ❑ name="msg"，和 JavaBean 中定义的变量对应“<value>HelloWorld</value>”，设定了希望向 JavaBean 中注入的字符串“HelloWorld”。

(3) config.xml 配置完毕后，将其放在当前的工作路径下，即 myApp 下，和 WEB-INF 在同一个目录。

2.2.3 编写测试程序 TestHelloWorld.java

这个 Java 文件主要用来对刚才编写的程序进行测试。编写步骤如下：

- (1) 用上面的方法，在 com.gc.test 包下建立 TestHelloWorld.java 文件。
- (2) 在 TestHelloWorld.java 中输入如下代码：

```
//***** TestHelloWorld.java*****  
package com.gc.test;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.FileSystemXmlApplicationContext;  
import com.gc.action.HelloWorld;  
public class TestHelloWorld {  
    public static void main(String[] args) {  
        //通过 ApplicationContext 来获取 Spring 的配置文件  
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");  
        //通过 Bean 的 id 来获取 Bean  
        HelloWorld HelloWorld = (HelloWorld) actx.getBean("HelloWorld");  
        System.out.println(HelloWorld.getMsg());  
    }  
}
```

上述测试代码中，程序利用 ApplicationContext 来获取 config.xml，从而完成 JavaBean 与 XML 之间关系的建立，接着就可以使用 getBean()方法来获取在 XML 中定义的内容了。

2.2.4 运行测试程序并查看输出结果

在 Eclipse 中运行 Java 程序的步骤如下：

- (1) 确保当前在 Eclipse 中编辑的是 TestHelloWorld.java 文件。
- (2) 选择 Run→Run As→Java Application 命令，Eclipse 即可运行 TestHelloWorld.java。
- (3) 输出结果为“HelloWorld”，如图 2.21 所示：

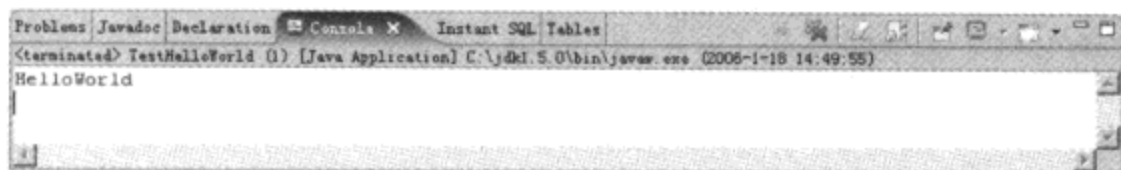


图 2.21 输出结果为“HelloWorld”

- (4) 把 config.xml 中<value>HelloWorld</value>的字符串“HelloWorld”改为“Hello gf”，再次运行 TestHelloWorld.java。
- (5) 输出结果为“Hello gf”，如图 2.22 所示：

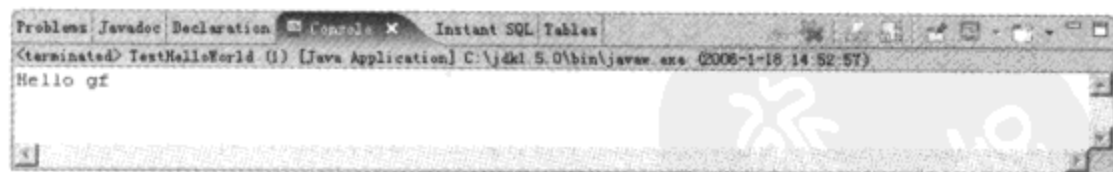


图 2.22 输出结果为“Hello gf”

至此，第一个 Spring 程序基本完成。可以看到，不用改变一行代码，只是改变了一下 XML 的配置，就可以使程序输出不同的结果。这就意味着，如果要改变一些对象间的依赖关系，只需要配置一下 XML 文件，而不用更改代码了，这就是 Spring 的一个与众不同之处。这里其实是简单利用了 Spring 的 IoC 功能，Spring 不仅能通过 XML 任意定义字符串，还可以改变类的来源。接下来笔者将展示 Spring 的这个功能。

2.3 改写第一个 Spring 例子实现中英文输出

通过第一个展示 HelloWorld 的 Spring 程序，读者初步感受了 Spring 的特色。下面将对这个实例进行扩充，利用 XML 配置来实现不同类之间的切换，从而使程序根据配置文件来输出中文问候语或英文问候语。编写这个程序分 3 步：首先编写接口 Hello.java，接下来编写实现类 ChHello.java 和 EnHello.java，接着修改 config.xml，最后改写测试程序 TestHelloWorld.java。

2.3.1 编写接口文件 Hello.java

这个接口定义了一个方法 doSalutation()，接下来的实现类都将实现这个接口。编写步骤如下：

- (1) 用上面的方法，在 com.gc.impl 包下建立 Hello.java 文件。
- (2) 在 Hello.java 中输入如下代码：

```
/****** Hello.java*****  
package com.gc.impl;  
public interface Hello {  
    //该方法用来输出问候语  
    public String doSalutation();  
}
```

2.3.2 编写实现接口的两个类（ChHello、EnHello）

这两个类主要用来实现中英文的输出。其中，ChHello 类用来输出中文问候语，EnHello 类用来输出英文问候语。编写步骤如下：

- (1) 在 com.gc.action 包下建立 ChHello.java 文件，并且这个类实现 Hello 接口中定义的 doSalutation() 方法。
- (2) 在 ChHello.java 中输入如下代码：

```
/****** ChHello.java*****  
package com.gc.action;  
import com.gc.impl.Hello;  
public class ChHello implements Hello{  
    //该变量用来存储字符串  
    public String msg = null;  
    //设定变量 msg 的 set 方法  
    public void setMsg(String msg) {  
        this.msg = msg;  
    }  
    //获取变量 msg 的 get 方法
```



```
public String getMsg() {  
    return this.msg;  
}  
//该方法用来输出中文问候语  
public String doSalutation() {  
    return "你好 " + this.msg;  
}  
}
```

(3) 在 com.gc.action 包下建立 EnHello.java 文件，并且这个类实现 Hello 接口中定义的 doSalutation() 方法。

(4) 在 EnHello.java 中输入如下代码：

```
/****** EnHello.java *****  
package com.gc.action;  
import com.gc.impl.Hello;  
public class EnHello implements Hello{  
    //该变量用来存储字符串  
    public String msg = null;  
    //设定变量 msg 的 set 方法  
    public void setMsg(String msg) {  
        this.msg = msg;  
    }  
    //获取变量 msg 的 get 方法  
    public String getMsg() {  
        return this.msg;  
    }  
    //该方法用来输出英文问候语  
    public String doSalutation() {  
        return "Hello " + this.msg;  
    }  
}
```

2.3.3 修改 Spring 的 config.xml

config.xml 主要用于根据不同的类来源来实现中英文输出。改写步骤如下：

(1) 在 Eclipse 中打开 myApp 下的 config.xml 文件。

(2) 把第一个 Spring 实例 config.xml 中的 class="com.gc.action.HelloWorld" 改为 class="com.gc.action.EnHello"，把注入的值 "HelloWorld" 改为 "gf"。

(3) 改写后的 config.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <!-- id 用来唯一表示该 Bean，class 用来表示该 Bean 的来源-->  
    <bean id="HelloWorld" class="com.gc.action.EnHello">
```



```
<!--将其变量 msg 通过依赖注入-->
<property name="msg">
    <value>gf</value>
</property>
</bean>
</beans>
```

2.3.4 修改测试程序 TestHelloWorld.java

编写步骤如下：

- (1) 打开 com.gc.test 包下的 TestHelloWorld.java 文件。
- (2) 把测试程序改写为如下所示：

```
//***** TestHelloWorld.java*****
package com.gc.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import com.gc.impl.Hello;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 来获取 Spring 的配置文件
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        Hello hello = (Hello) actx.getBean("HelloWorld");
        System.out.println(hello.doSalutation());
    }
}
```

2.3.5 运行测试程序并查看输出结果

运行步骤如下：

- (1) 用上面介绍过的在 Eclipse 中运行程序的方法运行测试程序 TestHelloWorld.java，输出结果为“Hello gf”，如图 2.23 所示。

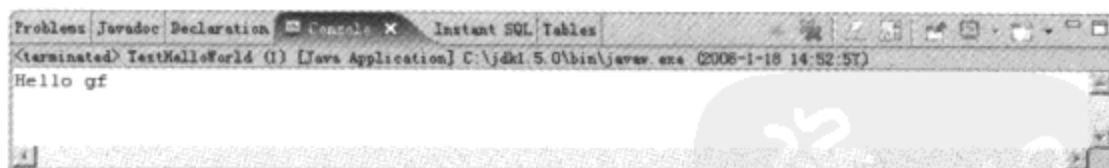


图 2.23 输出结果为“Hello gf”

- (2) 把 config.xml 中的 class 改为 class="com.gc.action.ChHello"。
- (3) 运行测试程序，输出结果为“你好 gf”，如图 2.24 所示。

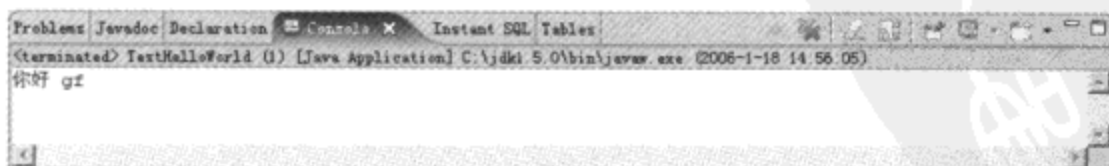


图 2.24 输出结果为“你好 gf”

可以看到，只是更改了 config.xml 中的 class 来源，测试程序输出的结果就改变了。

2.4 小 结

在本章中，笔者首先带领读者从零开始，一步一步地建立起 Spring 的开发环境，然后又编写了两个简单的 Spring 单机程序。在第一个实例中，读者可以任意改变向 JavaBean 中注入的字符串；在第二个实例中，除了可以任意改变向 JavaBean 中注入的字符串外，还可以把 ChHello 和 EnHello 这两类通过 XML 来配置，且不用改变测试代码，即可实现不同的输出。如果再写一个实现 Hello 接口中 doSalutation() 方法的类，仍然不用更改测试代码，只改变一下 XML 的配置，即可实现其他的输出。

笔者之所以不遗余力地向读者介绍上面的程序，是因为它们虽然简单，但却展示了 Spring 一个与众不同的功能，即 Spring 的 IoC。在本书的第 3 章笔者将详细讲解 Spring 的 IoC 和 DI。



第 2 篇



Spring 技术详解

- 第 3 章 Spring 基础概念
- 第 4 章 Spring 的核心容器
- 第 5 章 Spring 的 AOP
- 第 6 章 Spring 的事务处理
- 第 7 章 Spring 的持久层封装
- 第 8 章 Spring 的 Web 框架
- 第 9 章 Spring 的定时器



第3章 Spring 基础概念

在第2章中，笔者通过两个简单的实例展示了 Spring 的 IoC 功能，接下来将对 Spring 的 IoC 进行详细的讲解，因为 Spring 的核心就是 IoC。在本章中，首先从 IoC 的基本思想开始，然后通过实例的方式使读者对其概念和工作原理有一个深入的了解，最后会把第2章中的第一个实例进行改编，使其通过构造方式来实现同样的功能。

3.1 反向控制/依赖注入

近年来，在 Java 社区中掀起了一股轻量级容器的热潮，几乎每隔一段时间，就会有新的轻量级容器出现，这些轻量级的容器能够较好地帮助开发者快速地将不同的组件组装成一个应用程序。在这些轻量级的容器的背后，有一个共同的模式决定着容器装配组件的方式，就是“反向控制”，即 IoC，英文全称是 Inversion of Control。Martin Fowler 深入地探索了“反向控制”的工作原理，并为其起了一个新的名字叫做“依赖注入”，即 DI，英文全称是 Dependency Injection。关于 Martin Fowler 的这篇文章，读者可以在其网站上看到，网址是 <http://www.martinfowler.com/articles/injection.html>。

3.1.1 反向控制 (IoC)

单从字面上，其实很难理解“反向控制”所要表达的含义。其实在编程时，开发人员常说的“实现必须依赖抽象，而不是抽象依赖实现”就是“反向控制”的一种表现方式。下面，笔者主要通过举例来说明这个抽象的概念。这个实例主要说明的是如何通过 IoC 来实现业务逻辑从哪种数据库中取数据的问题。可能的取数据方式有3种，分别是：

- ❑ 从 SQL Server 数据库中取数据。
- ❑ 从 DB2 数据库中取数据。
- ❑ 从 Oracle 数据库中取数据。

介绍这个实例的思路是：首先介绍编写这类程序通常的做法，然后指出这种做法的不足，接着给出一种比较好的做法，即通过 IoC 来实现这类功能，最后对这种做法进行总结，使读者一步一步地了解 IoC。编写这类程序通常做法的具体步骤如下：

(1) 通常编写这类程序都是先编写一个从数据库取数据的类 `SqlServerDataBase.java`，这里以从 SQL Server 数据库中取数据为例。`SqlServerDataBase.java` 的示例代码如下。其中 `getDataFromSqlServer()` 是 `SqlServerDataBase` 类中的一个方法，具体负责从 SQL Server 数据库中取数据。

```
//***** SqlServerDataBase.java*****  
public class SqlServerDataBase {  
    .....  
    //从 SQLServer 数据库中获取数据  
    public List getData() {  
        .....  
    }  
}
```

(2) 业务逻辑类 Business.java 通过 SqlServerDataBase.java 中的方法来从 SQL Server 数据库中取数据。Business.java 的示例代码如下。其中 SqlServerDataBase 是用来从 SQL Server 数据库中取数据的类。

```
//***** Business.java*****  
public class Business {  
    private SqlServerDataBase db = new SqlServerDataBase();  
    .....  
    //从 SQL Server 数据库中获取数据  
    public void getData() {  
        .....  
        List list = db.getDataFromSqlServer();  
        .....  
    }  
}
```

可以看出以上程序编写的不足之处：Business 类依赖于 SqlServerDataBase 类，如果业务改变，用户现在要求从 DB2 或 Oracle 数据库中取数据，则这个程序就不适用了，必须要修改 Business 类。

(3) 改为从 DB2 数据库中取数据，DB2DataBase.java 的示例代码如下。其中 getDataFromDB2() 是 DB2DataBase 类中的一个方法，具体负责从 DB2 数据库中取数据。

```
//***** DB2DataBase.java*****  
public class DB2DataBase {  
    .....  
    //从 DB2 数据库中获取数据  
    public List getData() {  
        .....  
    }  
}
```

(4) 必须修改业务逻辑类 Business.java，改为从 DB2 数据库中取数据。Business.java 的示例代码如下，其中 DB2DataBase 是用来从 DB2 数据库中取数据的类。

```
//***** Business.java*****  
public class Business {  
    private DB2DataBase db = new DB2DataBase();  
    .....  
    //从 DB2 数据库中获取数据  
    public void getData() {
```



```

.....
List list = db.getDataFromDB2();
.....
}
}

```

(5) 同样，用户现在要求从 Oracle 数据库中取数据，则这个程序就不适用了。改为从 Oracle 数据库中取数据，OracleDataBase.java 的示例代码如下。其中 getDataFromOracle () 是 OracleDataBase 类中的一个方法，具体负责从 Oracle 数据库中取数据。

```

//***** OracleDataBase.java*****
public class OracleDataBase {
    .....
    //从 Oracle 数据库中获取数据
    public List getData() {
        .....
    }
}

```

(6) 还要修改业务逻辑类 Business.java，改为从 Oracle 数据库中取数据。Business.java 的示例代码如下。其中 OracleDataBase 是用来从 Oracle 数据库中取数据的类。

```

//***** Business.java*****
public class Business {
    private OracleDataBase db = new OracleDataBase();
    .....
    //从 Oracle 数据库中获取数据
    public void getData() {
        .....
        List list = db.getDataFromOracle ();
        .....
    }
}

```

至此，读者应该可以发现了，这可不是一个好的设计，因为每次业务需求的变动都要导致程序的大量修改，怎样才能改变这种情形的发生呢？怎样才能实现 Business 类的重用呢？IoC 就可以解决这个问题，它可以通过面向抽象编程来改变这种情况。

下面就利用 IoC 来实现 Business 类的重用，编写思路是：首先编写一个获取数据的接口，然后每个具体负责从各种数据库中获取数据的类都实现这个接口，而在业务逻辑类中，则根据接口编程，并不与具体获取数据的类打交道。通过 IoC 来实现这个功能的具体步骤如下。

(1) 编写用来获取数据的接口 DataBase。DataBase.java 的示例代码如下：

```

//***** DataBase.java*****
public interface DataBase {
    //该方法用来获取数据
    public void getData();
}

```

(2) 编写具体负责从 SQL Server 数据库中取数据的类 `SqlServerDataBase`，该类实现了接口 `DataBase`。`SqlServerDataBase.java` 的示例代码如下：

```
//***** SqlServerDataBase.java*****  
public class SqlServerDataBase implement DataBase {  
    //该方法用来获取数据  
    public void getData() {  
        //以下是具体从 SQL Server 数据库中取数据的代码  
        .....  
    }  
}
```

(3) 编写业务逻辑类 `Business`，该类只针对接口 `DataBase` 编码，而不针对实体类。`Business.java` 的示例代码如下：

```
//***** Business.java*****  
public class Business {  
    //针对接口 DataBase 定义变量  
    private DataBase db;  
    public void setDataBase(DataBase db) {  
        this.db = db;  
    }  
    .....  
    //根据注入的数据库类，从×××数据库中获取数据  
    public void getData() {  
        .....  
        db.getData();  
        .....  
    }  
}
```

(4) 编写测试类 `TestBusiness`。`TestBusiness.java` 的示例代码如下：

```
//***** TestBusiness.java*****  
public class TestBusiness {  
    private Business business = new Business();  
    .....  
    //根据注入的数据库类，从 SQL Server 数据库中获取数据  
    public void getData() {  
        .....  
        business.setDataBase(new SqlServerDataBase());  
        business.getData();  
        .....  
    }  
}
```

通过这种方式 `Business` 类就可以重用了，不管从哪种数据库中获取数据，`Business` 类都不用改动，只需要实现具体的 `DataBase` 接口就可以了。例如，用户要求改为从 DB2 数据库中获取数据，只要实现一个具体负责从 DB2 数据库中取数据的类就可以了。

(5) 编写具体负责从 DB2 数据库中取数据的类 `DB2DataBase`，该类实现了接口

DataBase。DB2DataBase.java 的示例代码如下：

```
/****** DB2DataBase.java*****  
public class DB2DataBase implement DataBase {  
    public void getData() {  
        //以下是具体从 DB2 数据库中取数据的代码  
        .....  
    }  
}
```

(6) 业务逻辑类 Business 不用作任何改动，修改测试类 TestBusiness。TestBusiness.java 的示例代码如下：

```
/****** TestBusiness.java*****  
public class TestBusiness {  
    private Business business = new Business();  
    .....  
    //根据注入的数据库类，从 DB2 数据库中获取数据  
    public void getData() {  
        .....  
        business.setDataBase(new DB2DataBase());  
        business.getData();  
        .....  
    }  
}
```

(7) 如果用户又要求改为从 Oracle 数据库中获取数据，只要实现一个具体负责从 Oracle 数据库中取数据的类就可以了。编写具体负责从 Oracle 数据库中取数据的类 OracleDataBase，该类实现了接口 DataBase。OracleDataBase.java 的示例代码如下：

```
/****** OracleDataBase.java*****  
public class OracleDataBase implement DataBase {  
    public void getData() {  
        //以下是具体从 Oracle 数据库中取数据的代码  
        .....  
    }  
}
```

(8) 业务逻辑类 Business 不用作任何改动，修改测试类 TestBusiness。TestBusiness.java 的示例代码如下：

```
/****** TestBusiness.java*****  
public class TestBusiness {  
    private Business business = new Business();  
    .....  
    //根据注入的数据库类，从 Oracle 数据库中获取数据  
    public void getData() {  
        .....  
        business.setDataBase(new OracleDataBase());  
        business.getData();  
        .....  
    }  
}
```

```
}  
}
```

从上面的例子可以看到，在第一个例子中，使用通常的做法，**Business** 类依赖于具体获取数据的类；而在第二个例子中，通过接口来编程，即控制关系的反向转移，实现了 IoC 功能，并使代码获得了重用。这也就实现了上面所说的“实现必须依赖抽象，而不是抽象依赖实现”。

3.1.2 依赖注入 (DI)

Martin Fowler 在其文章中提出了“它们反转了哪方面的控制”的问题后，就为 IoC 起了一个更能说明这种模式特点的新名字，叫做“依赖注入”，即 **Dependency Injection**，Spring 就是使用 **Dependency Injection** 来实现 IoC 功能，接着 Martin Fowler 介绍了 **Dependency Injection** 的 3 种实现方式，接下来笔者将结合上面的例子对这 3 种实现方式进行详细的讲解。

3.2 依赖注入的 3 种实现方式

在讲解依赖注入的 3 种实现方式之前，这里先澄清一下依赖注入的意义：让组件依赖于抽象，当组件要与其他实际对象发生依赖关系时，通过抽象来注入依赖的实际对象。

依赖注入的 3 种实现方式分别是：接口注入(**interface injection**)、Set 注入(**setter injection**)和构造注入(**constructor injection**)。接下来笔者还将主要通过举例的方式，把依赖注入的 3 种实现方式介绍给读者。

3.2.1 接口注入 (interface injection)

接口注入指的就是在接口中定义要注入的信息，并通过接口完成注入。结合前面的示例，其具体步骤如下。

(1) 编写一个接口 **IBusiness**，各种数据库的注入将通过这个接口进行。**IBusiness.java** 的示例代码如下：

```
//***** IBusiness.java*****  
public interface IBusiness {  
    public void createDI(DataBase db);  
}
```

(2) 任何想要使用数据库实例的类都必须实现这个接口，业务逻辑类 **Business** 实现这个接口 **IBusiness**。**Business.java** 的示例代码如下：

```
//***** Business.java*****  
public class Business implement IBusiness {  
    private DataBase db;  
    public void createDI (DataBase db) {
```



```
        this.db = db;
    }
    .....
    //根据注入的数据库类，从×××数据库中获取数据
    public void getData() {
        .....
        db.getData();
        .....
    }
}
```

(3) 编写测试类 TestBusiness。TestBusiness.java 的示例代码如下：

```
/****** TestBusiness.java*****
public class TestBusiness {
    private Business business = new Business();
    .....
    //根据注入的数据库类，从 Oracle 数据库中获取数据
    public void getData() {
        .....
        business.createDI (new OracleDataBase());
        business.getData();
        .....
    }
}
```

如果要完成依赖关系注入的对象，必须实现 IBusiness 接口。

3.2.2 Set 注入 (setter injection)

Set 注入指的就是在接受注入的类中定义一个 Set 方法，并在参数中定义需要注入的元素。为了让类 Business 接受 DataBase 的注入，需要为它定义一个 Set 方法来接受 DataBase 的注入。Business.java 的示例代码如下：

```
/****** Business.java*****
public class Business {
    private DataBase db;
    public void setDataBase(DataBase db) {
        this.db = db;
    }
    .....
    //根据注入的数据库类，从×××数据库中获取数据
    public void getData() {
        .....
        db.getData();
        .....
    }
}
```

更详细的代码，可以参看 3.1 节的第二个例子，采用的就是 Set 注入的方式。

3.2.3 构造注入 (constructor injection)

构造注入指的就是在接受注入的类中定义一个构造方法，并在参数中定义需要注入的元素。为了让类 Business 接受 DataBase 的注入，需要为它定义一个构造方法，来接受 DataBase 的注入。Business.java 的示例代码如下：

```
/****** Business.java *****  
public class Business {  
    private DataBase db;  
    public Business (DataBase db) {  
        this.db = db;  
    }  
    .....  
    //根据注入的数据库类，从×××数据库中获取数据  
    public void getData() {  
        .....  
        db.getData();  
        .....  
    }  
}
```

3.3 将 HelloWorld 实例改为构造注入方式实现

Spring 支持 Set 注入 (setter injection) 和构造注入 (constructor injection)，但更推荐使用 Set 注入。上面讲过，第 2 章的第一个实现 HelloWorld 的实例就是采用 Set 注入方式实现的，读者可以参看第 2 章的实例。下面笔者把这个实例改为采用构造注入方式实现。改写思路是：首先修改类 HelloWorld，在该类中增加一个构造方法，然后修改 Spring 的配置文档 config.xml，最后编写测试程序 TestHelloWorld.java。

3.3.1 修改 HelloWorld.java

修改 com.gc.action 包下的 HelloWorld.java，增加一个构造方法，并把要注入的字符串 msg 作为参数，代码如下，在 HelloWorld 类中增加了一个构造方法 public HelloWorld (String msg)。

```
/****** HelloWorld.java *****  
package com.gc.action;  
public class HelloWorld {  
    //该变量用来存储字符串  
    public String msg = null;  
    //增加了一个构造方法  
    public HelloWorld (String msg) {  
        this.msg = msg;  
    }  
}
```

```

//设定变量 msg 的 set 方法
public void setMsg(String msg) {
    this.msg = msg;
}
//获取变量 msg 的 get 方法
public String getMsg() {
    return this.msg;
}
}

```

3.3.2 修改 config.xml

在 Spring 中, 利用 Set 注入和构造注入时, 在 XML 配置文档中使用的语法是不一样的。修改配置文件 config.xml 内容如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义一个 Bean, 通过构造函数进行注入-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">
        <!--通过构造函数进行注入-->
        <constructor-arg index="0">
            <value>HelloWorld</value>
        </constructor-arg>
    </bean>
</beans>

```

代码说明:

- ☐ constructor-arg, 用来表示是通过构造方式来注入参数的。
- ☐ index="0", 表示是构造方法中的第一个参数, 如果只有一个参数, 则可以不用设置这个属性值。

3.3.3 编写测试程序 TestHelloWorld.java

修改 TestHelloWorld.java, 代码如下:

```

//***** TestHelloWorld.java *****
package com.gc.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import com.gc.action.HelloWorld;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 来获取 Spring 的配置文件
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        HelloWorld HelloWorld = (HelloWorld) actx.getBean("HelloWorld");
        System.out.println(HelloWorld.getMsg());
    }
}

```

```
}  
}
```

3.3.4 运行测试程序并查看输出结果

在 Eclipse 中运行 Java 程序的步骤如下：

- (1) 确保当前在 Eclipse 中编辑的是 TestHelloWorld.java 文件。
- (2) 选择 Run→Run As→Java Application 命令，Eclipse 即可运行 TestHelloWorld.java。
- (3) 输出结果为 “HelloWorld”，如图 3.1 所示。

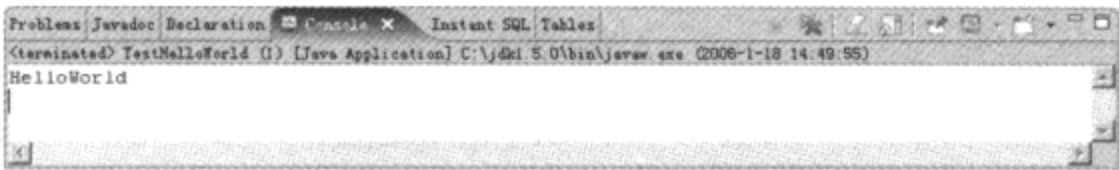


图 3.1 输出结果为 “HelloWorld”

3.4 使用哪种注入方式

至于要使用构造注入或 Set 注入来完成依赖注入这个问题，其实就是在讨论：要在对象建立时就准备好所有的资源，或是在对象建立好后，使用 Set 注入来进行设定。

因为网络上关于这个的讨论太多了，各有各的道理，所以在这里不想再进行讨论，只是将笔者在实际工作中的一些经验分享给读者。

使用构造注入可以在建构对象的同时一并完成依赖关系的建立，对象一建立则所有的一切也就准备好了，但如果要建立的对象关系很多，使用构造注入会在建构函式上留下一长串的参数，且不易记忆，这时使用 Set 注入会是个不错的选择。

使用 Set 注入可以有明确的名称，可以了解注入的对象会是什么，像 setXXX()这样的名称会比记忆 Constructor 上某个参数位置代表某个对象更好。

然而使用 Set 注入由于提供了 setXXX()方法，所以不能保证相关的数据成员或资源在执行时不会被更改设定，所以如果开发人员想要让一些数据成员或资源变为只读或是私有，使用构造注入会是个简单的选择。

3.5 小结

Spring 的核心是个 IoC 容器，用户可以用 Setter 或 Constructor 的方式来实现自己的业务对象。至于对象与对象之间的关系建立，则通过组态设定，让 Spring 在执行时根据组态的设定来建立对象之间的依赖关系，开发人员就不必特地撰写一些 Helper 来自行建立这些对象之间的依赖关系，这不仅减少了大量的程序撰写，也降低了对对象之间的耦合程度。当读者了解了 IoC 的工作原理和一些基本使用方法后，也就对 Spring 的核心有了一定的认识，接下来从第 4 章开始笔者将详细讲解 Spring 的语法。

第 4 章 Spring 的核心容器

Spring 的核心容器实现了 IoC，其目的是提供一种无侵入式的框架。在本章中，首先讲解 Spring 的基础 Bean 的相关知识，然后介绍 Spring 是如何对 Bean 进行管理，最后讲解 Spring 提供的一些更强大的功能。BeanFactory 和 ApplicationContext 是了解 Spring 核心的关键。

4.1 什么是 Bean

Bean 是描述 Java 的软件组件模型，有点类似于 Microsoft 的 COM 组件的概念。在 Java 模型中，通过 Bean 可以无限扩充 Java 程序的功能，通过 Bean 的组合可以快速生成新的应用程序。对于开发人员来说，最重要的是 Bean 可以实现代码的重复利用。

Bean 被设计成可以在不同的环境里重复使用，其功能没有限制。一个 Bean 可以完成一个简单的功能，例如存储一个问候语；也可以完成一个复杂的功能，例如实现自动取款机的取钱功能。一个 Bean 可以被设计成在单独服务器上独立工作，也可以与其他一组分布式组件协同工作。

4.2 Bean 的基础知识

在 Spring 中，有两个最基本、最重要的包，即 org.springframework.beans 和 org.springframework.context。在这两个包中，为了实现一种无侵入式的框架，代码中大量引用了 Java 中的反射机制，通过动态调用的方式避免了硬编码，为 Spring 的反向控制特性提供了基础。

在这两个包中，其中最重要的类是 BeanFactory 和 ApplicationContext。BeanFactory 提供了一种先进的配置机制来管理任何种类的 Bean。ApplicationContext 建立在 BeanFactory 之上，并增加了其他的功能，例如对于国际化的支持、获取资源、事件传递等。

要想掌握 BeanFactory 和 ApplicationContext，就必须先了解 Bean 的一些基础知识。

4.2.1 Bean 的标识（id 和 name）

先来看第 2 章中实现 HelloWorld 的实例中 Spring 的一个配置文档。示例代码如下：

```
<!--Bean 的配置文档-->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
```



```
<beans>
  <!--定义一个 Bean, id 是这个 Bean 的唯一标识, class 指出这个 Bean 的来源 -->
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" >
    <!--配置 Bean 的开始 -->
    <property name="msg">
      <value>HelloWorld</value>
    </property>
  <!--定义上面 Bean 的结束 -->
  </bean>
</beans>
```

读者从上面的示例代码中可以看到, 在 Spring 的配置文档中, 一个 Bean 有一个 id。这个 id 在管理 Bean 的 BeanFactory 或 ApplicationContext 中必须是唯一标识的, 因为在代码中通过 BeanFactory 或 ApplicationContext 来获取 Bean 的实例时, 都要用它来作为唯一的索引。

当然也可以使用 name 属性来指定 Bean 的 id。示例代码如下:

```
<!--Bean 的配置文档-->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!--定义一个 Bean, name 是这个 Bean 的唯一标识, class 指出这个 Bean 的来源 -->
  <bean name="HelloWorld" class="com.gc.action.HelloWorld" >
    <!--配置 Bean 的开始 -->
    <property name="msg">
      <value>HelloWorld</value>
    </property>
  <!--定义上面 Bean 的结束 -->
  </bean>
</beans>
```

在 Spring 中可以用 id 或者 name 属性来指定 Bean 的 id, 并且在这两个或其中一个属性中至少指定一个 id。id 和 name 的区别是:

id 属性允许指定一个 Bean 的 id, 并且它在 XML DTD 中作为一个真正的 XML 元素的 ID 属性被标记, 所以 XML 解析器能够在其他元素指向它的时候做一些额外的校验。

但是, XML 规范严格限定了在 XML ID 中合法的字符。如果在开发中有必要使用一些非法的字符, 即不符合 ID 规定的字符, 或者要给 Bean 增加其他的别名, 则可以通过 name 属性指定一个或多个 id。当指定多个 id 时要用逗号 (,) 或者分号 (;) 分隔。

综上所述, 用 id 属性指定 Bean 的 id 是一个比较好的方式。

📌说明: 一个 Bean 一般只有一个 id; 如果一个 Bean 有超过一个的 id, 那么另外的那些 id 可以认为是这个 Bean 的别名。

4.2.2 Bean 的类 (class)

在上面的配置文档中, 在 Bean 的 id 属性后面, 还有 Bean 的另外一个属性 class。示例

代码如下：

```
<bean id="HelloWorld" class="com.gc.action.HelloWorld" >
```

在 Spring 的配置文档中，class 属性指明了 Bean 的来源，即 Bean 的实际路径。

🔔 注意：class 属性路径的完整性。例如，要写 class="com.gc.action.HelloWorld"，而不能写 class=" HelloWorld"。

4.2.3 Singleton 的使用

在 Spring 中，Bean 可以被定义为两种部署模式中的一种：singleton 或 non-singleton (prototype)。Spring 默认为 singleton 模式。

- ❑ 如果一个 Bean 被定义为 singleton 模式，那么就只有一个共享的实例存在，所有对这个 Bean 的请求都会返回这个唯一的实例。
- ❑ 如果一个 Bean 被定义为 non-singleton (prototype) 模式，那么对这个 Bean 的每次请求都会创建一个新的 Bean 实例，读者可以把它当成类似 new 的操作。

在前面所有的实例中，Spring 配置文档中的 Bean 都被默认定义为 singleton 模式，也就是说下面两段代码的效果是一样的。

(1) 配置文档中 id 为 HelloWorld 的 Bean 被默认定义为 singleton 模式的代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- id 用来唯一表示该 Bean，class 用来表示该 Bean 的来源-->
    <bean id="HelloWorld" class="com.gc.action. HelloWorld ">
        <!--通过依赖注入来完成-->
        <property name="msg">
            <value> HelloWorld </value>
        </property>
    </bean>
</beans>
```

(2) 配置文档中 id 为 HelloWorld 的 Bean 被显式定义为 singleton 模式的代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- id 用来唯一表示该 Bean，class 用来表示该 Bean 的来源-->
    <bean id="HelloWorld" class="com.gc.action. HelloWorld " singleton="true">
        <!--通过依赖注入来完成-->
        <property name="msg">
            <value> HelloWorld </value>
        </property>
    </bean>
</beans>
```

上面这两段示例代码，客户端每次向 BeanFactory 请求时，只返回一个实例。如果要客户端每次向 BeanFactory 请求时都创建新的实例，则要把 `singleton="true"` 改为 `singleton="false"`。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!-- id 用来唯一表示该 Bean，class 用来表示该 Bean 的来源-->
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" singleton="false">
    <!--通过依赖注入来完成-->
    <property name="msg">
      <value> HelloWorld </value>
    </property>
  </bean>
</beans>
```

4.2.4 Bean 的属性

在 Spring 中，Bean 的属性值有两种注入方式：基于 setter 的依赖注入和基于构造函数的依赖注入。基于 setter 的依赖注入，是在调用无参的构造函数或无参的静态工厂方法实例化配置文档中定义的 Bean 之后，通过调用 Bean 上的 setter 方法实现的。基于构造函数的依赖注入，是通过调用带有许多参数的构造方法实现的，每个参数表示一个对象或者属性。

在前面的实例中，已经对这两种实现方式都作了介绍和举例，但所举示例中主要是直接指定属性值，这里就不再进行详细介绍了。需要指出的是，在定义 Bean 时，除了直接指定属性值外，还可以参考配置文档中定义的其他 Bean。

这里把前面的 HelloWorld 示例改为输出 HelloWorld 时加上当前时间。改写这个示例的思路是：首先在名称为 HelloWorld 的 Bean 中增加一个 Date 类型的私有变量，然后改写 Spring 的配置文档 config.xml，添加一个 id 为 date 的 Bean，并在原来的 id 为 HelloWorld 的 Bean 中参考配置文档中定义的 id 为 date 的 Bean，最后修改测试程序 TestHelloWorld.java，查看输出结果。具体步骤如下：

(1) 在 Eclipse 中打开 com.gc.action 包下的 HelloWorld.java。

(2) 在名称为 HelloWorld 的 Bean 中，增加一个 Date 类型的私有变量 date。HelloWorld.java 的示例代码如下：

```
/** ***** HelloWorld.java ***** */
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    //构造函数
```

```
public HelloWorld (String msg) {  
    this.msg = msg;  
}  
//设定变量 msg 的 set 方法  
public void setMsg(String msg) {  
    this.msg = msg;  
}  
//获取变量 msg 的 get 方法  
public String getMsg() {  
    return this.msg;  
}  
//设定变量 msg 的 set 方法  
public void setDate(Date date) {  
    this.date = date;  
}  
//获取变量 date 的 get 方法  
public Date getDate() {  
    return this.date;  
}  
}
```

(3) 打开 myApp 目录下的 config.xml。

(4) 在 config.xml 中添加对 Date 的定义，定义 Bean 的 id 为 date，同时在 id 为 HelloWorld 的 Bean 中增加一个 name 为 date 的属性设定，并指定为参考配置文档中定义的 id 为 date 的 Bean。config.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <!--该 Bean 的两个变量都通过依赖注入来完成之间的联系-->  
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">  
        <!--msg 变量通过依赖注入来完成-->  
        <property name="msg">  
            <value>HelloWorld</value>  
        </property>  
        <!--date 变量通过依赖注入来完成-->  
        <property name="date">  
            <ref bean="date"/>  
        </property>  
    </bean>  
    <bean id="date" class="java.util.Date"/>  
</beans>
```

在上述代码中，id 为 HelloWorld 的 Bean 中的 date 属性通过<ref bean="date"/>参考了 id 为 date 的 Bean。

(5) 改写测试程序 TestHelloWorld，增加对当前时间的输出。TestHelloWorld.java 的示例代码如下：


```
//***** TestHelloWorld.java*****  
package com.gc.test;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.FileSystemXmlApplicationContext;  
  
import com.gc.action.HelloWorld;  
import com.gc.impl.Hello;  
//该程序主要用来测试增加对当前时间的输出的功能  
public class TestHelloWorld {  
    public static void main(String[] args) {  
        //通过 ApplicationContext 来获取 Spring 的配置文件  
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");  
        HelloWorld helloWorld = (HelloWorld) actx.getBean("HelloWorld");  
        System.out.println(helloWorld.getDate() + " " + helloWorld.getMsg());  
    }  
}
```

运行 TestHelloWorld.java, 输出结果如图 4.1 所示。

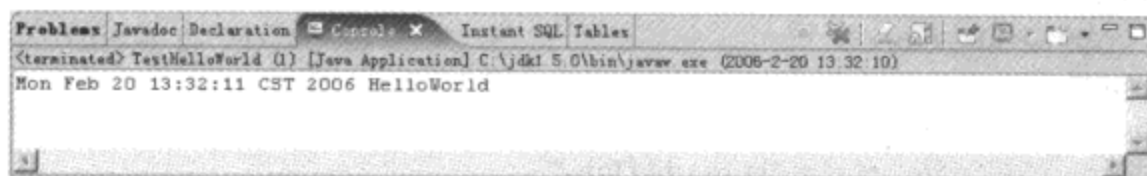


图 4.1 输出结果为“当前时间+HelloWorld”

(6) 除了在配置文档 config.xml 中单独定义一个 id 为 date 的 Bean, 然后让 id 为 HelloWorld 的 Bean 参考它外, 还可以在 id 为 HelloWorld 的 Bean 的 date 属性中直接定义参考的 Bean。改写 config.xml, 示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">  
        <property name="msg">  
            <value>HelloWorld</value>  
        </property>  
        <!--注意参考方式的变化-->  
        <property name="date">  
            <bean id="date" class="java.util.Date"/>  
        </property>  
    </bean>  
</beans>
```

(7) 测试程序 TestHelloWorld 不用改变, 运行测试程序, 输出结果与如图 4.1 所示的输出结果一样, 如图 4.2 所示。

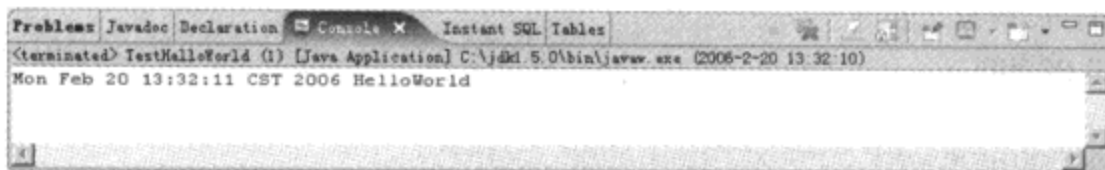


图 4.2 输出结果同样为“当前时间+HelloWorld”

4.2.5 对于 null 值的处理

在 Bean 的属性配置中，往往会遇到 null 值的问题。在 Spring 中，null 值是这样处理的：在下面的代码中，把 msg 设定为 null，共有两种方法。示例代码 1 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld">
    <!--注意对 null 值的处理-->
    <property name="msg">
      <value>null</value>
    </property>
    <property name="date">
      <bean id="date" class="java.util.Date"/>
    </property>
  </bean>
</beans>
```

示例代码 2 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld">
    <!--注意对 null 值处理的另一种方式-->
    <property name="msg">
      <null/>
    </property>
    <property name="date">
      <bean id="date" class="java.util.Date"/>
    </property>
  </bean>
</beans>
```

这两个示例代码的主要区别是：第一个示例代码用的是<value>null</value>，第二个示例代码用的是<null/>。这两个示例代码的功能都与在 Bean 中设定 this.msg = null 等价。

4.2.6 使用依赖 depends-on

Bean 的 depends-on 属性可以用来在初始化使用这个 Bean 之前, 强制执行一个或多个 Bean 的初始化。把前面示例代码中的配置文档 config.xml 中 id 为 HelloWorld 的 Bean 的 depends-on 属性设定为 depends-on="date", 则在初始化 HelloWorld 之前, 会强制初始化 date。示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--使用 depends-on 时, 依赖的 Bean 必须先进行初始化-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <!--参考 date Bean-->
        <property name="date">
            <ref bean="date"/>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>
</beans>
```

通过前面的示例可以知道, 即使不加 depends-on="date", 一样可以得到相同的输出结果。这里只是作为示例代码来讲解。当然, 有时开发人员如果想在 Bean 初始化之前强制初始化另外一个或多个 Bean, 就可以使用依赖 depends-on。

4.2.7 一个较为完整的 Bean 配置文档

下面的代码是一个较为完整的 Bean 配置文档, 并添加了详细的注释。主要目的是让读者经过前面对 Bean 的基础知识学习后, 对 Spring 中的配置文档有一个全面的了解。Bean 配置文档的示例代码如下:

```
<!--Bean 的配置文档-->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义一个 Bean, id 是这个 Bean 的唯一标识, class 指出这个 Bean 的来源, singleton 指定这个 Bean 是否是单例模式, depends-on 指定这个 Bean 初始化前, 强制初始化 date-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" singleton="true" depends-on="date">
        <!--配置 Bean 的属性 -->
        <property name="msg">
            <value>HelloWorld</value>
```

```
</property>
<!--指定 Bean 的一个依赖 -->
<property name="date">
    <ref bean="date"/>
</property>
<!--定义上面 Bean 的结束 -->
</bean>
<bean id="date" class="java.util.Date"/>
</beans>
```

4.3 Bean 的生命周期

前面的章节，主要是让读者了解一下 Spring 中 Bean 的一些相关属性等基础知识。一个 Bean 从定义到销毁都有一个生命周期。在 Spring 中，Bean 的生命周期包括 Bean 的定义、Bean 的初始化、Bean 的使用和 Bean 的销毁 4 个阶段。下面分别进行介绍。

4.3.1 Bean 的定义

在 Spring 中，通常是通过配置文档的方式来定义 Bean 的。例如，在第 2 章中的实例即是如此。来看一下第 2 章实例中的配置文档 config.xml，为了便于阅读，在这里加上了对每一行的注释。config.xml 的示例代码如下：

```
<!--Bean 的配置文档-->
<!--首先定义为 XML 的方式来存储 Bean 的配置-->
<?xml version="1.0" encoding="UTF-8"?>
<!--声明使用的是 http://www.springframework.org/dtd/spring-beans.dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--配置 Bean 的开始，根节点 Beans 中包含一个或多个 Bean 元素-->
<beans>
    <!--定义一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">
        <!--配置 Bean 的开始 -->
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <!--定义上面 Bean 的结束 -->
    </bean>
<!--配置 Bean 的结束 -->
</beans>
```

上面的示例代码只定义了一个 Bean。在一个配置文档中，可以定义多个 Bean。定义多个 Bean 的 config.xml 示例代码如下：

```
<!--Bean 的配置文档-->
<!--首先定义为 XML 的方式来存储 Bean 的配置-->
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!--声明使用的是 http://www.springframework.org/dtd/spring-beans.dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--配置 Bean 的开始，根节点 Beans 中包含一个或多个 Bean 元素-->
<beans>
    <!--定义一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">
        <!--定义这个 Bean 中的一个变量 msg，并给它赋值为 HelloWorld -->
        <property name="msg">
            <value>HelloWorld</value>
        </property>
    <!--定义上面 Bean 的结束 -->
    </bean>
    <!--定义另一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id=" EnHello " class=" com.gc.action.EnHello ">
        <!--定义这个 Bean 中的一个变量 msg，并给它赋值为 gf -->
        <property name="msg">
            <value>gf</value>
        </property>
    <!--定义上面 Bean 的结束 -->
    </bean>
    <!--再定义一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id=" ChHello " class=" com.gc.action.ChHello ">
        <!--定义这个 Bean 中的一个变量 msg，并给它赋值为 gd-->
        <property name="msg">
            <value>gd</value>
        </property>
    <!--定义上面 Bean 的结束 -->
    </bean>
    <!--定义更多的 Bean-->
    .....
<!--配置 Bean 的结束 -->
</beans>
```

📌 注意：在一个大的应用中，会有很多的 Bean 需要在配置文档中定义，这样配置文档就会很大，变得不好维护。这时可以把相关的 Bean 放在一个配置文档中，出现多个配置文档。

4.3.2 Bean 的初始化

在 Spring 中，Bean 的初始化有两种方式：

- ❑ 在配置文档中通过指定 `init-method` 属性来完成。
- ❑ 实现 `org.springframework.beans.factory.InitializingBean` 接口。

如果一个 Bean 实现了 `org.springframework.beans.factory.InitializingBean` 接口，则它的所有必需的属性被 `BeanFactory` 设置后，会自动执行它的 `afterPropertiesSet()` 方法。下面通过示

例代码来分别说明这两种方式。

第一种方式，通过在配置文档中指定 `init-method` 属性来完成。实现思路是：首先在 `HelloWorld.java` 中增加一个方法 `init()`，用来完成初始化工作，然后修改配置文档 `config.xml`，指定 Bean 中要初始化的方法，最后编写测试程序 `TestHelloWorld.java`。具体步骤如下：

(1) 在 `HelloWorld.java` 中增加一个方法 `init()`，用来完成初始化工作，并去掉构造函数 `HelloWorld()`。`HelloWorld.java` 的示例代码如下：

```
/****** HelloWorld.java*****  
package com.gc.action;  
import java.util.Date;  
public class HelloWorld {  
    //该变量用来存储字符串  
    private String msg = null;  
    //该变量用来存储日期  
    private Date date = null;  
    //初始化  
    public void init () {  
        this.msg = "HelloWorld";  
        this.date = new Date();  
    }  
    //设定变量 msg 的 set 方法  
    public void setMsg(String msg) {  
        this.msg = msg;  
    }  
    //获取变量 msg 的 get 方法  
    public String getMsg() {  
        return this.msg;  
    }  
    //设定变量 msg 的 set 方法  
    public void setDate(Date date) {  
        this.date = date;  
    }  
    //获取变量 date 的 get 方法  
    public Date getDate() {  
        return this.date;  
    }  
}
```

(2) 修改配置文档 `config.xml`，指定 Bean 中要初始化的方法为 `init()`，并且去掉通过 setter 注入值的方式。`config.xml` 的示例代码如下：

```
<!--Bean 的配置文档-->  
<!--首先定义为 XML 的方式来存储 Bean 的配置-->  
<?xml version="1.0" encoding="UTF-8"?>  
<!--声明使用的是 http://www.springframework.org/dtd/spring-beans.dtd -->  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<!--配置 Bean 的开始，根节点 Beans 中包含一个或多个 Bean 元素-->
```

```
<beans>
  <!--定义一个 Bean, id 是这个 Bean-的唯一标识, class 指出这个 Bean 的来源 -->
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" init-method="init" >
  <!--定义上面 Bean 的结束 -->
  </bean>
<!--配置 Bean 的结束 -->
</beans>
```

(3) 测试程序不变，运行 TestHelloWorld.java，输出结果如图 4.3 所示。

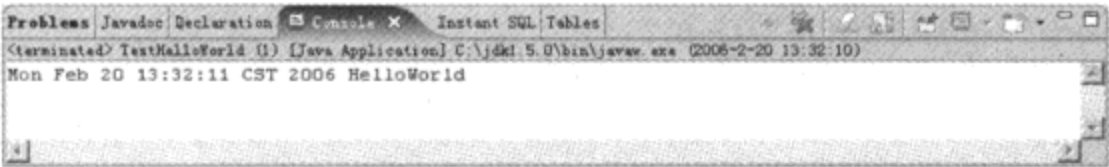


图 4.3 输出结果为“当前时间+HelloWorld”

第二种方式，实现 org.springframework.beans.factory.InitializingBean 接口。实现思路是：首先让 HelloWorld.java 实现 InitializingBean 接口，增加一个方法 afterPropertiesSet ()用来完成初始化工作，然后修改配置文件 config.xml，最后编写测试程序 TestHelloWorld.java。具体步骤如下：

(1) 让 HelloWorld.java 实现 InitializingBean 接口，增加一个方法 afterPropertiesSet ()，用来完成初始化工作。HelloWorld.java 的示例代码如下：

```
//***** HelloWorld.java*****
package com.gc.action;

import org.springframework.beans.factory.InitializingBean;
import java.util.Date;

public class HelloWorld implements InitializingBean {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    //初始化
    public void afterPropertiesSet () {
        this.msg = "HelloWorld";
        this.date = new Date();
    }
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
    //设定变量 msg 的 set 方法
    public void setDate(Date date) {
        this.date = date;
    }
}
```



```
//获取变量 date 的 get 方法
public Date getDate() {
    return this.date;
}
}
```

(2) 修改配置文档 config.xml。config.xml 的示例代码如下：

```
<!--Bean 的配置文档-->
<!--首先定义为 XML 的方式来存储 Bean 的配置-->
<?xml version="1.0" encoding="UTF-8"?>
<!--声明使用的是 http://www.springframework.org/dtd/spring-beans.dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--配置 Bean 的开始，根节点 Beans 中包含一个或多个 bean 元素-->
<beans>
    <!--定义一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld">
    <!--定义上面 Bean 的结束 -->
    </bean>
<!--配置 Bean 的结束 -->
</beans>
```

(3) 测试程序不变，运行 TestHelloWorld.java，输出结果与图 4.3 所示的输出结果一样，如图 4.4 所示。

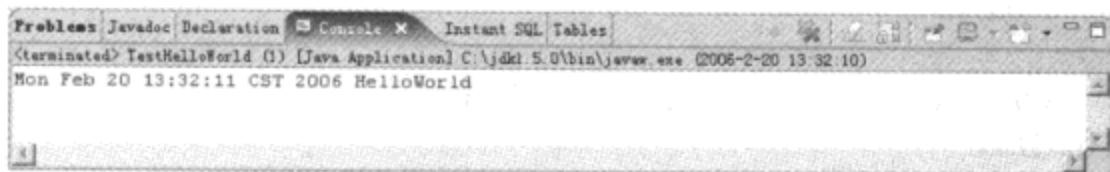


图 4.4 输出结果为“当前时间+HelloWorld”

以上两种方式实现的功能是一样的，但是第一种方式却没有把代码耦合于 Spring。所以在这里推荐使用第一种方式。

4.3.3 Bean 的使用

在 Spring 中，Bean 的使用有 3 种方式。

第一种，使用 BeanWrapper。示例代码如下：

```
HelloWorld helloWorld = new HelloWorld();
BeanWrapper bw = new BeanWrapperImpl(helloWorld);
bw.setPropertyValue("msg", "HelloWorld");
System.out.println(bw.getPropertyValue("msg "));
```

第二种，使用 BeanFactory。示例代码如下：

```
InputStream is = new FileInputStream("config.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
HelloWorld helloWorld = (HelloWorld) factory.getBean("HelloWorld");
```

```
System.out.println(helloWorld.getMsg());
```

第三种，使用 `ApplicationContext`。示例代码如下：

```
ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
HelloWorld HelloWorld = (HelloWorld) actx.getBean("HelloWorld");
System.out.println(HelloWorld.getMsg());
```

4.3.4 Bean 的销毁

在 Spring 中，Bean 的销毁有以下两种方式：

第一种，在配置文档中通过指定 `destroy-method` 属性来完成。

第二种，实现 `org.springframework.beans.factory.DisposableBean` 接口。

如果一个 Bean 实现了 `org.springframework.beans.factory.DisposableBean` 接口，则会自动执行它的 `destroy()` 方法。下面通过示例代码来分别说明这两种方式。

第一种方式，通过在配置文档中指定 `destroy-method` 属性来完成。其实现思路是：首先在 `HelloWorld.java` 中增加一个方法 `cleanup()`，用来完成销毁工作，并打印出销毁的内容，然后修改配置文档 `config.xml`，指定 Bean 中要销毁的方法，最后编写测试程序 `TestHelloWorld.java`。具体步骤如下：

(1) 在 `HelloWorld.java` 中增加一个方法 `cleanup()`，用来完成销毁工作。`HelloWorld.java` 的示例代码如下：

```
//***** HelloWorld.java*****
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    //初始化
    public void init () {
        this.msg = "HelloWorld";
        this.date = new Date();
    }
    //销毁
    public void cleanup () {
        this.msg = "";
        this.date = null;
        System.out.println("您销毁了 msg " + this.msg + "和 date " + this.date);
    }
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
```

```

public String getMsg() {
    return this.msg;
}
//设定变量 msg 的 set 方法
public void setDate(Date date) {
    this.date = date;
}
//获取变量 date 的 get 方法
public Date getDate() {
    return this.date;
}
}

```

(2) 修改配置文件 config.xml，指定 Bean 中要初始化的方法为 cleanup ()。config.xml 的示例代码如下：

```

<!--Bean 的配置文件-->
<!--首先定义为 Xml 的方式来存储 Bean 的配置-->
<?xml version="1.0" encoding="UTF-8"?>
<!--声明使用的是 http://www.springframework.org/dtd/spring-beans.dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--配置 Bean 的开始，根节点 Beans 中包含一个或多个 Bean 元素-->
<beans>
    <!--定义一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" init-method="init" destroy-method="cleanup">
    <!--定义上面 Bean 的结束 -->
    </bean>
<!--配置 Bean 的结束 -->
</beans>

```

(3) 测试程序不变，运行 TestHelloWorld.java，输出结果如图 4.5 所示。

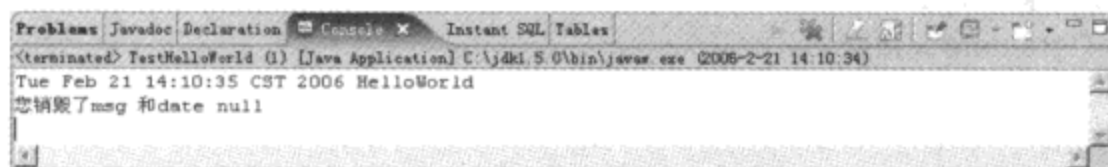


图 4.5 输出结果为“当前时间+HelloWorld”和销毁的提示信息

第二种方式，实现 org.springframework.beans.factory.DisposableBean 接口。其实现思路是：首先让 HelloWorld.java 实现 DisposableBean 接口，增加一个方法 destroy ()用来完成销毁工作，然后修改配置文件 config.xml，最后编写测试程序 TestHelloWorld.java。具体步骤如下：

(1) 让 HelloWorld.java 实现 DisposableBean 接口，增加一个方法 destroy ()，用来完成销毁工作。HelloWorld.java 的示例代码如下：

```

//***** HelloWorld.java*****
package com.gc.action;
import org.springframework.beans.factory. InitializingBean;
import org.springframework.beans.factory. DisposableBean;

```

```

import java.util.Date;
public class HelloWorld implements InitializingBean ,DisposableBean {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    //初始化
    public void afterPropertiesSet () {
        this.msg = "HelloWorld";
        this.date = new Date();
    }
    //销毁
    public void destroy () {
        this.msg = "";
        this.date = null;
        System.out.println("您销毁了 msg " + this.msg + "和 date " + this.date);
    }
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
    //设定变量 date 的 set 方法
    public void setDate(Date date) {
        this.date = date;
    }
    //获取变量 date 的 get 方法
    public Date getDate() {
        return this.date;
    }
}

```

(2) 修改配置文档 config.xml。config.xml 的示例代码如下：

```

<!--Bean 的配置文档-->
<!--首先定义为 XML 的方式来存储 Bean 的配置-->
<?xml version="1.0" encoding="UTF-8"?>
<!--声明使用的是 http://www.springframework.org/dtd/spring-beans.dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<!--配置 Bean 的开始，根节点 Beans 中包含一个或多个 Bean 元素-->
<beans>
    <!--定义一个 Bean，id 是这个 Bean 的唯一标识，class 指出这个 Bean 的来源 -->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" init-method="init">
    <!--定义上面 Bean 的结束 -->
    </bean>
<!--配置 Bean 的结束 -->
</beans>

```


(3) 测试程序不变, 运行 TestHelloWorld.java, 输出结果与图 4.5 所示的输出结果一样, 如图 4.6 所示。

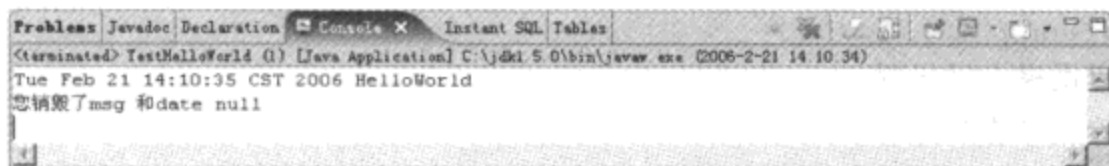


图 4.6 输出结果同样为“当前时间+HelloWorld”和销毁的提示信息

以上两种方式实现的功能是一样的, 同 Bean 的初始化工作一样, 但是第一种方式却没有把代码耦合于 Spring, 所以在这里推荐使用第一种方式。

4.4 用 ref 的属性指定依赖的 3 种模式

在 Spring 中, 用 ref 的属性指定依赖。有 3 种模式: local、Bean 和 parent。下面分别对这 3 种模式进行讲解并进行比较。

4.4.1 用 local 属性指定

如果一个 Bean 与被参考引用的 Bean 在同一个 XML 文件中而且被参考引用的 Bean 是用 id 来命名的, 那么就可以使用 ref 的 local 属性。这样会让 XML 解析器更早地在 XML 文档解析时, 验证 Bean 的 id。local 属性的值必须与被参考引用的 Bean 的 id 属性一致。如果在同一个 XML 文件中没有匹配的元素, XML 解析器将会产生一个错误。

用 XML 来配置文档的一个明显缺点是, 开发人员不能及时地发现 XML 中的错误。为了能够尽可能早地发现 XML 中的错误配置信息, 如果一个 Bean 与被参考引用的 Bean 在同一个 XML 文件中, 那么使用 local 形式将是最好的选择。用 ref 的 local 属性来指定依赖的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
    <property name="msg">
      <value>HelloWorld</value>
    </property>
    <!-- local 属性的值必须与被参考引用的 Bean 的 id 属性一致-->
    <property name="date">
      <ref local="date"/>
    </property>
  </bean>
  <bean id="date" class="java.util.Date"/>
</beans>
```

4.4.2 用 Bean 属性指定

用 ref 元素的 Bean 属性指定被参考引用的 Bean 是 Spring 中最常见的形式,它允许指向的 Bean 可以在同一个 XML 中,也可以不在同一个 XML 中。Bean 属性的值可以与被参考引用的 Bean 的 id 属性相同,也可以与被参考引用的 Bean 的 name 属性相同。用 ref 的 Bean 属性来指定依赖的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
    <property name="msg">
      <value>HelloWorld</value>
    </property>
    <!--允许指向的 Bean 可以在同一个 XML 中,也可以不在同一个 XML 中-->
    <property name="date">
      <ref bean="date"/>
    </property>
  </bean>
  <bean id="date" class="java.util.Date"/>
</beans>
```

4.4.3 用 parent 属性指定

用 parent 属性指定被参考引用的 Bean 时,允许引用当前 BeanFactory 或 ApplicationContext 的父 BeanFactory 或 ApplicationContext 中的 Bean。parent 属性的值可以与被参考引用的 Bean 的 id 属性相同,也可以与被参考引用 Bean 的 name 属性相同。用 ref 的 parent 属性来指定依赖的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
    <property name="msg">
      <value>HelloWorld</value>
    </property>
    <!--允许引用当前 BeanFactory 或 ApplicationContext 的父 BeanFactory 或 ApplicationContext
中的 Bean -->
    <property name="date">
      <ref parent="date"/>
    </property>
  </bean>
</beans>
```

4.4.4 用 local 属性与 Bean 属性指定依赖的比较

在这 3 种模式中，用 local 属性指定依赖和用 Bean 属性指定依赖最为常用。下面主要对这两种模式进行比较。

- ❑ 相同之处：两种模式都可以用 Bean 的 id 来进行参考引用，都可以对同一 XML 中的 Bean 进行参考引用。
- ❑ 不同之处：用 Bean 属性指定依赖可以用 Bean 的 name 来进行参考引用，还可以对不同 XML 中的 Bean 进行参考引用。

4.5 Bean 自动装配的 5 种模式

在 Spring 中，自动装配可以指定给每一个单独的 Bean，因此可以给一些 Bean 使用自动装配而其他的 Bean 不使用自动装配。通过使用自动装配，开发人员可以减少指定属性的需要，从而节省一些属性设定的工作。使用 Bean 元素的 autowire 属性来指定 Bean 定义的自动装配，共有 5 种模式，即 byName、byType、constructor、autodetect 和 no。

4.5.1 使用 byName 模式

byName 模式指的就是通过 Bean 的属性名字进行自动装配。在 Spring 的配置文档 XML 中，查找一个与将要装配的属性同样名字的 Bean。

在配置文档中，有一个 id 为 HelloWorld 的 Bean 被设置为通过 byName 自动装配，HelloWorld.java 包含一个 date 变量，Spring 就会查找一个叫做 date 的 Bean 定义，然后用它来设置 date 属性。使用 byName 模式的 HelloWorld.java 的示例代码如下：

```
//***** HelloWorld.java*****
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
}
```

```
//设定变量 msg 的 set 方法
public void setDate(Date date) {
    this.date = date;
}
//获取变量 date 的 get 方法
public Date getDate() {
    return this.date;
}
}
```

配置文件 config.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 通过 Bean 的属性名字进行自动装配-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="byName">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>
</beans>
```

4.5.2 使用 byType 模式

byType 模式指的就是如果 XML 中正好有一个与属性类型一样的 Bean，就自动装配这个属性。如果有多于一个这样的 Bean，就抛出一个异常，指出可能不能对那个 Bean 使用 byType 的自动装配。

在配置文档中，有一个 id 为 HelloWorld 的 Bean 被设置为通过 byType 自动装配，HelloWorld.java 包含一个 date 变量，Spring 就会查找一个类型为 date 的 Bean 定义，然后用它来设置 date 属性。使用 byType 模式的 HelloWorld.java 的示例代码如下：

```
//***** HelloWorld.java*****
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
```



```

        return this.msg;
    }
    //设定变量 msg 的 set 方法
    public void setDate(Date date) {
        this.date = date;
    }
    //获取变量 date 的 get 方法
    public Date getDate() {
        return this.date;
    }
}

```

配置文档 config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--如果 XML 中正好有一个同属性类型一样的 Bean，就自动装配这个属性-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="byType">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>
</beans>

```

如果没有匹配的 Bean，则什么都不会发生，属性不会被设置。如果这是开发人员不想要的情况，通过设置 `dependency-check="objects"` 属性值来指定在这种情况下应该抛出错误。

4.5.3 使用 constructor 模式

constructor 模式指的就是根据构造函数的参数进行自动装配。在配置文档中，有一个 id 为 HelloWorld 的 Bean 被设置为通过 constructor 自动装配，HelloWorld.java 包含一个构造函数方法，Spring 就会根据构造函数的参数查找合适类型的 Bean 定义，然后用它来设置构造函数的参数的值。使用 constructor 模式的 HelloWorld.java 的示例代码如下：

```

//***** HelloWorld.java*****
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    public HelloWorld (Date date) {
        this.date = date;
    }
}

```

```

//设定变量 msg 的 set 方法
public void setMsg(String msg) {
    this.msg = msg;
}
//获取变量 msg 的 get 方法
public String getMsg() {
    return this.msg;
}
//设定变量 msg 的 set 方法
public void setDate(Date date) {
    this.date = date;
}
//获取变量 date 的 get 方法
public Date getDate() {
    return this.date;
}
}

```

配置文档 config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--根据构造函数的参数进行自动装配-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="constructor">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>
</beans>

```

4.5.4 使用 autodetect 模式

autodetect 模式指的就是通过对 Bean 检查类的内部来选择 constructor 或 byType。如果先找到 constructor 就用 constructor；如果没有 constructor，而找到 byType，就用 byType。使用 autodetect 模式的 HelloWorld.java 的示例代码如下：

```

//***** HelloWorld.java *****
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    public HelloWorld (Date date) {
        this.date = date;
    }
}

```

```

    }
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
    //设定变量 date 的 set 方法
    public void setDate(Date date) {
        this.date = date;
    }
    //获取变量 date 的 get 方法
    public Date getDate() {
        return this.date;
    }
}

```

配置文件 config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--通过对 Bean 检查类的内部来选择 constructor 或 byType-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="autodetect">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>
</beans>

```

在上面这个示例中，Bean 的属性设定为 autowire="autodetect"。在 HelloWorld.java 中找到了构造函数，则 Spring 会使用 constructor 模式；如果没有构造函数，则使用 byType 模式。

4.5.5 使用 no 模式

no 模式指的就是不使用自动装配。通过上面几节可以了解到，Bean 的引用必须通过 ref 元素定义。这是默认的配置，在很多企业级的应用环境中不鼓励使用自动装配模式，因为它对于 Bean 之间的参考依赖关系不清晰。使用 no 模式的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--不使用自动装配-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="no">

```

```
<property name="msg">
    <value>HelloWorld</value>
</property>
<property name="date">
    <ref bean="date"/>
</property>
</bean>
<bean id="date" class="java.util.Date"/>
</beans>
```

4.5.6 对 5 种模式进行总结

显式的指定依赖，例如 `property` 和 `constructor-arg` 元素，总会覆盖自动装配。正如前面已经提到过的，对于大型的应用，不鼓励使用自动装配，因为它去除了参考依赖的透明性和清晰性。

有了自动装配后，可以减少开发人员的输入工作，但是却使开发人员很难看出 Bean 的每个属性是否都设定完成。应该怎么解决这个问题呢？Spring 提出了它的一些解决方案，4.6 节将进行详细讲解。

4.6 Bean 依赖检查的 4 种模式

就像自动装配功能一样，依赖检查能够分别对每一个 Bean 应用或取消应用。默认的是不检查依赖关系。使用 Bean 元素的 `dependency-check` 属性来指定 Bean 定义的依赖检查共有 4 种模式：`simple`、`object`、`all`、`none`。

4.6.1 为什么要使用依赖检查

前面提到，在自动装配中，因为是隐式的，不像前面通过 `ref` 的属性指定依赖那么直接，所以开发人员很难看出 Bean 的每个属性是否都设定完成。这时就要借助于依赖检查来实现查看 Bean 的每个属性是否都设定完成的功能。

4.6.2 使用 simple 模式

`simple` 模式指的是对基本类型、字符串和集合进行依赖检查。使用 `simple` 模式的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--只会对字符串 msg 进行依赖检查-->
```



```
<bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="autodetect" dependency-check="simple">
</bean>
<bean id="date" class="java.util.Date"/>
</beans>
```

上述代码中，由于设定为 `dependency-check="simple"`，则只会对字符串 `msg` 进行依赖检查。

4.6.3 使用 object 模式

object 模式指的是对依赖的对象进行依赖检查。使用 object 模式的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!--对依赖的对象进行依赖检查-->
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="autodetect" dependency-check="object">
  </bean>
  <bean id="date" class="java.util.Date"/>
</beans>
```

上述代码中，由于设定为 `dependency-check="object"`，则只会对 `date` 对象进行依赖检查。

4.6.4 使用 all 模式

all 模式指的是对全部属性进行依赖检查。使用 all 模式的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!--对全部属性进行依赖检查-->
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="autodetect" dependency-check="all">
  </bean>
  <bean id="date" class="java.util.Date"/>
</beans>
```

上述代码中，由于设定为 `dependency-check="all"`，则会对 `msg` 字符串和 `date` 对象都进行依赖检查。

4.6.5 使用 none 模式

none 模式指的是不进行依赖检查。使用 none 模式的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!--不进行依赖检查-->
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" autowire="autodetect" dependency-
check="none">
  </bean>
  <bean id="date" class="java.util.Date"/>
</beans>
```

上述代码中，由于设定为 `dependency-check="none"`，则不会对 `msg` 字符串和 `date` 对象进行依赖检查。

4.6.6 对 4 种模式进行总结

一般情况下，依赖检查和自动装配结合使用。当开发人员想查看 Bean 的每个属性是否都设定完成的时候，依赖检查的作用会显得更大。当依赖检查和自动装配结合使用时，依赖检查会在自动装配完成后发生。当 Bean 的属性都有默认的值，或者不需要对 Bean 的属性是否都被设置到 Bean 上检查时，依赖检查的作用就不是很大了。

4.7 集合的注入方式

前面讲的 Bean 的配置都是针对简单类型和实体的属性配置，对于集合 `list`、`set`、`map` 以及 `props` 元素则有不同的配置方式，在 Spring 中相应地可以用来定义和设置类型为 Java 的 `Lis`、`Set`、`Map` 和 `Properties`。下面分别来进行讲解。

4.7.1 List

假如把前面的 `HelloWorld` 类中的 `msg` 类型改为 `List`，则 `HelloWorld.java` 的示例代码如下：

```
/** ***** HelloWorld.java ***** */
package com.gc.action;
//该类用于演示集合注入
public class HelloWorld {
    //定义一个 List 变量 msg
    private List msg = null;
    //变量 msg 的 set 方法
    public void setMsg(List msg) {
        this.msg = msg;
    }
    //变量 msg 的 get 方法
```

```
public List getMsg() {  
    return this.msg;  
}  
}
```

则配置文档示例代码如下：

```
<beans>  
  <bean id=" HelloWorld " class=" com.gc.action. HelloWorld ">  
    <property name="msg">  
      <!--集合 list 元素的配置方式-->  
      <list>  
        <value> gf </value>  
        <value> gd </value>  
        <value> HelloWorld </value>  
      </list>  
    </property>  
  </bean>  
</beans>
```

4.7.2 Set

假如把前面的 HelloWorld 类中的 msg 类型改为 Set，则 HelloWorld.java 的示例代码如下：

```
/** ***** HelloWorld.java ***** */  
package com.gc.action;  
  
public class HelloWorld {  
    //定义一个 Set 变量 msg  
    private Set msg = null;  
    //变量 msg 的 set 方法  
    public void setMsg(Set msg) {  
        this.msg = msg;  
    }  
    //变量 msg 的 get 方法  
    public Set getMsg() {  
        return this.msg;  
    }  
}
```

则配置文档示例代码如下：

```
<beans>  
  <bean id=" HelloWorld " class=" com.gc.action. HelloWorld ">  
    <property name="msg">  
      <!--集合 set 元素的配置方式-->  
      <set>  
        <value> gf </value>  
        <value> gd </value>
```

```
        <value> HelloWorld </value>
      </set>
    </property>
  </bean>
</beans>
```

4.7.3 Map

假如把前面的 HelloWorld 类中的 msg 类型改为 Map，则 HelloWorld.java 的示例代码如下：

```
//***** HelloWorld.java*****
package com.gc.action;

public class HelloWorld {
    //定义一个 Map 变量 msg
    private Map msg = null;
    //变量 msg 的 set 方法
    public void setMsg(Map msg) {
        this.msg = msg;
    }
    //变量 msg 的 get 方法
    public Map getMsg() {
        return this.msg;
    }
}
```

则配置文档示例代码如下：

```
<beans>
  <bean id=" HelloWorld " class=" com.gc.action. HelloWorld ">
    <property name="msg">
      <!--集合 map 元素的配置方式-->
      <map>
        <entry key="gf">
          <value> HelloWorld </value>
        </entry>
        <entry key="gd">
          <value> HelloWorld </value>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

4.7.4 Properties

假如把前面的 HelloWorld 类中的 msg 类型改为 Properties，则 HelloWorld.java 的示例

代码如下：

```
//***** HelloWorld.java *****
package com.gc.action;

public class HelloWorld {
    //定义一个 Properties 变量 msg
    private Properties msg = null;
    //变量 msg 的 set 方法
    public void setMsg(Properties msg) {
        this.msg = msg;
    }
    //变量 msg 的 get 方法
    public Properties getMsg() {
        return this.msg;
    }
}
```

则配置文档示例代码如下：

```
<beans>
  <bean id=" HelloWorld " class=" com.gc.action. HelloWorld ">
    <property name="msg">
      <!--集合 Properties 元素的配置方式-->
      <props>
        <prop key="gf"> HelloWorld </prop>
        <prop key="gd"> HelloWorld </prop>
      </props>
    </property>
  </bean>
</beans>
```

4.7.5 对集合的注入方式进行总结

对于 List、Set、Map 和 Properties 来说都是类似的，就是都要先把要注入的信息注入到集合中去，然后再把集合注入到相关的 Bean 中。

4.8 管理 Bean

在前面的几节中，主要是对 Spring 中 Bean 的一些相关知识进行了讲解。接下来主要讲解在 Spring 中是如何来管理这些 Bean 的。在 Spring 中，对 Bean 的管理主要有 3 种方式，分别是使用 BeanWrapper 管理 Bean、使用 BeanFactory 管理 Bean 和使用 ApplicationContext 管理 Bean。

4.8.1 使用 BeanWrapper 管理 Bean

在 org.springframework.beans 包中，还有两个非常重要的类：BeanWrapper 接口及它的实现 BeanWrapperImpl。BeanWrapper 封装了一个 Bean 的行为，提供了设置和获得属性值的功能。通过 BeanWrapper 可以获得 Bean 的属性描述、查询只读或者可写属性。

下面通过 BeanWrapper 来实现输出 HelloWorld。其实现思路是：首先修改 HelloWorld.java，增加一个无参数的构造函数，配置文档 Config.xml 不用改变，修改测试代码，然后查看输出结果。具体步骤如下：

(1) 修改 HelloWorld.java，增加一个无参数的构造函数。HelloWorld.java 的示例代码如下：

```
//***** HelloWorld.java *****
package com.gc.action;
import java.util.Date;
public class HelloWorld {
    //该变量用来存储字符串
    private String msg = null;
    //该变量用来存储日期
    private Date date = null;
    public HelloWorld () {
    }
    //设定变量 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取变量 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
    //设定变量 msg 的 set 方法
    public void setDate(Date date) {
        this.date = date;
    }
    //获取变量 date 的 get 方法
    public Date getDate() {
        return this.date;
    }
}
```

(2) 修改测试程序 TestHelloWorld。TestHelloWorld.java 的示例代码如下：

```
//***** TestHelloWorld.java *****
package com.gc.test;
import java.util.Date;
import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;
```

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.action.HelloWorld;
public class TestHelloWorld {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 Class.forName()方法获取类 HelloWorld 的一个实例
        Object obj = Class.forName("com.gc.action.HelloWorld").newInstance();
        //通过 BeanWrapper 来设定类 HelloWorld 的属性
        BeanWrapper bw = new BeanWrapperImpl(obj);
        //根据类变量设定变量的值
        bw.setPropertyValue("msg", "HelloWorld");
        bw.setPropertyValue("date", new Date());
        //把刚才设定的值根据变量名进行输出
        System.out.println(bw.getPropertyValue("date") + " " + bw.getPropertyValue("msg"));
    }
}

```

(3) 运行测试程序，输出结果如图 4.7 所示。

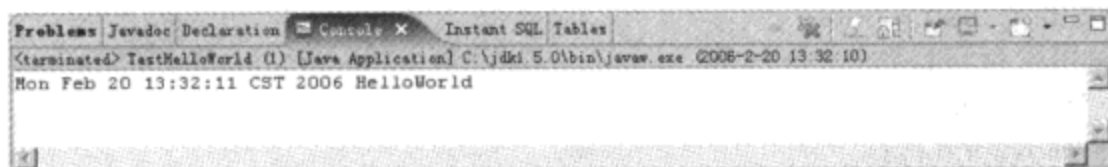


图 4.7 输出结果为“当前时间+HelloWorld”

4.8.2 使用 BeanFactory 管理 Bean

BeanFactory 实际上是实例化，配置和管理众多 Bean 的容器。这些 Bean 通常会彼此合作，因而它们之间会产生依赖。

一个 BeanFactory 可以用接口 `org.springframework.beans.factory.BeanFactory` 表示，这个接口有多个实现。最常用的简单的 BeanFactory 实现是 `org.springframework.beans.factory.xml.XmlBeanFactory`。

下面通过 BeanFactory 来实现输出 HelloWorld。其实现思路是：HelloWorld.java 和配置文档 Config.xml 都不用改变，修改测试代码，然后查看输出结果。具体步骤如下：

(1) 修改测试程序 TestHelloWorld。TestHelloWorld.java 的示例代码如下：

```

//***** TestHelloWorld.java *****
package com.gc.test;
import java.util.Date;
import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.action.HelloWorld;

```

```
public class TestHelloWorld {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 ClassPathResource 获取配置文档
        ClassPathResource res = new ClassPathResource("config.xml");
        //通过 XmlBeanFactory 来解析配置文档
        XmlBeanFactory factory = new XmlBeanFactory(res);
        //根据 id 获取 Bean
        HelloWorld helloWorld = (HelloWorld) factory.getBean("HelloWorld");
        //输出 Bean 在配置文档中设定的内容
        System.out.println(helloWorld.getDate() + " " + helloWorld.getMsg())
    }
}
```

(2) 运行测试程序，输出结果如图 4.8 所示。

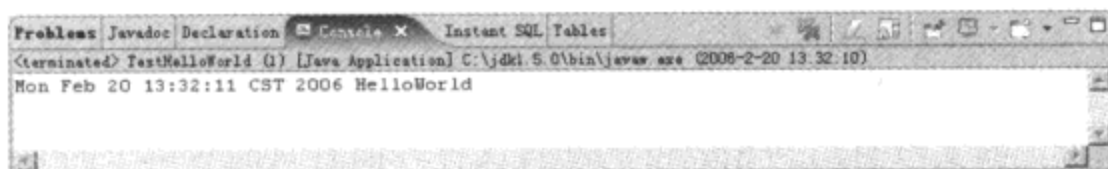


图 4.8 输出结果为“当前时间+HelloWorld”

4.8.3 使用 ApplicationContext 管理 Bean

ApplicationContext 建立在 BeanFactory 之上，并增加了其他功能，例如对于国际化的支持、获取资源、事件传递等。BeanFactory 提供了配置框架和基本功能，而 ApplicationContext 为它增加了更强的功能。一般来说，ApplicationContext 是 BeanFactory 的完全超集，任何 BeanFactory 功能同样也适用于 ApplicationContext。

下面通过 ApplicationContext 来实现输出 HelloWorld。其实现思路是：HelloWorld.java 和配置文档 Config.xml 都不用改变，修改测试代码，然后查看输出结果。具体步骤如下：

(1) 修改测试程序 TestHelloWorld。TestHelloWorld.java 的示例代码如下：

```
//***** TestHelloWorld.java *****
package com.gc.test;
import java.util.Date;
import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.action.HelloWorld;
public class TestHelloWorld {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        //根据配置文档中定义的 id 获得 Bean
```



```
    HelloWorld HelloWorld = (HelloWorld) actx.getBean("HelloWorld");  
    //输出 Bean 在配置文档中设定的内容  
    System.out.println(HelloWorld.getDate() + " " + HelloWorld.getMsg())  
}  
}
```

(2) 运行测试程序，输出结果如图 4.9 所示。

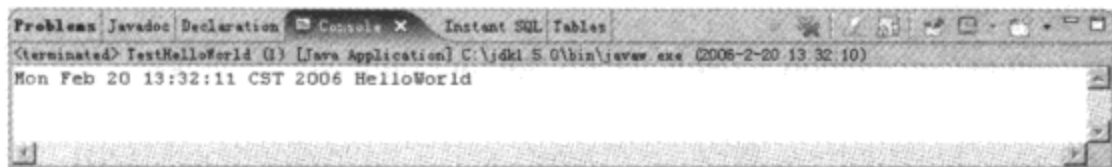


图 4.9 输出结果为“当前时间+HelloWorld”

4.8.4 3 种管理 Bean 的方式的比较

从上面的实例可以看出，3 种管理 Bean 的方式实现了同样的功能。只是对于 BeanWrapper 来说，其实没有用到 Spring 的配置文件，而且只能对单个的 Bean 进行设定，所以一般来说，在实际编程中，并不常用。

常用的是 BeanFactory 和 ApplicationContext。因为目前使用 Spring 主要是进行 B/S 结构的编程，而且可以把 ApplicationContext 看作是 BeanFactory 的超集，所以 ApplicationContext 更常用，而且提供的功能更多。接下来主要讲解 ApplicationContext 提供的一些其他功能。

4.9 ApplicationContext 更强的功能

相对于 BeanFactory 来说，ApplicationContext 除了提供 BeanFactory 的所有功能外，还有一些其他的功能，主要包括国际化支持、资源访问和事件传递。下面将讲解 ApplicationContext 在 BeanFactory 的基本功能之外的其他功能。

4.9.1 国际化支持

在笔者参加的项目中，对于信息的处理，通常有两种方法：

- ☐ 将信息存放在数据库，用的时候从数据库里取。
- ☐ 将信息存放在 Java 类里，用的时候从 Java 类里取。

这两种方式对于实现国际化来说，都是比较困难的。而 Spring 在国际化方面提供了良好的支持。ApplicationContext 继承了 org.springframework.context.MessageResource 接口，使用 getMessage() 的各个方法来取得信息资源，从而实现国际化信息的目的。getMessage() 有如下 3 个方法：

(1) String getMessage (String code, Object[] args, String default, Locale loc)，这个方法是从 MessageSource 取得信息的基本方法。如果找不到指定的信息，则会使用默认信息。

(2) `String getMessage (String code, Object[] args, Locale loc)`, 和上一个方法差不多, 只是没有默认值可以指定。如果找不到信息, 就会抛出一个 `NoSuchMessageException`。

(3) `String getMessage (MessageSourceResolvable resolvable, Locale locale)`, 通过 `MessageSourceResolvable` 来传入需要获取信息的代号。

当 `ApplicationContext` 被加载的时候, 它会自动查找在 XML 中定义的 `messageSource`。Spring 约定这个 Bean 必须被定义为 `messageSource`。开发人员可以通过 `org.springframework.context.support.ResourceBundleMessageSource` 来取得国际化信息。

下面举一个通过 Spring 实现国际化的例子。实现思路是: 首先定义 Spring 的配置文档 `config.xml`, 然后定义存放信息资源的文档, 最后编写测试程序, 查看信息的输出。具体步骤如下:

(1) 继续在以前的实例上进行修改。打开 Spring 的配置文档 `config.xml`, 进行修改。`config.xml` 的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!--负责国际化支持-->
  <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename">
      <!--国际化支持的定义在文件名为 messages 的文件中-->
      <value>messages</value>
    </property>
  </bean>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" >
  </bean>
  <bean id="date" class="java.util.Date"/>
</beans>
```

在上面的配置文档中, 定义了一个 id 为 `messageSource` 的 Bean, 这个 Bean 的来源是 `org.springframework.context.support.ResourceBundleMessageSource`。设定 Bean 的 `basename` 为 `messages`, 意味着存放信息资源的文档的名称为 `messages.properties` 或 `messages.calss`。

(2) 用记事本编写存放信息资源的文档 `messages.properties`。`messages.properties` 的内容如下:

```
HelloWorld=问候语: {0} 问候时间: {1}
```

代码说明: {0}和{1}用来标识当从外部传入参数时, 传入值存放的位置。

(3) 把 `messages.properties` 存放在 `ClassPath` 下, 即 `myApp\WEB-INF\src` 下。

(4) 编写测试程序 `TestHelloWorld`。`TestHelloWorld.java` 的示例代码如下:

```
//***** TestHelloWorld.java*****
package com.gc.test;
import java.util.Calendar;
```

```
import java.util.Locale;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.action.HelloWorld;
public class TestHelloWorld {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        //设定当前时间
        Object[] objs = new Object[] {"HelloWorld", Calendar.getInstance().getTime()};
        //国际化支持
        String msg = actx.getMessage("HelloWorld", objs, Locale.CHINA);
        System.out.println(msg);
    }
}
```

代码说明:

- ❑ objs 是一个数组, 用来存放要传入的内容。数组中的内容分别对应于 messages.properties 中的{0}和{1}。
- ❑ 测试程序使用的是 getMessage()方法的第二种。

(5) 运行测试程序, 查看输出结果如图 4.10 所示。

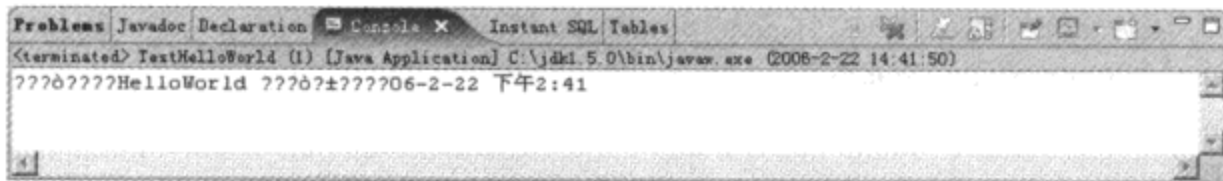


图 4.10 输出结果为乱码

(6) 输出的结果并不是想要的结果。虽然在测试程序中也指定了 Locale.CHINA, 但并没有输出中文。这是因为 Java 本身在转码过程中出现了问题。下面通过转码来解决它。

(7) 把 messages.properties 放在 C 盘的根目录下。

(8) 单击 Windows 的“开始”菜单, 选择“运行”命令, 弹出“运行”对话框, 如图 4.11 所示。

(9) 在“运行”对话框中输入“cmd”, 然后单击“确定”按钮, 弹出“cmd 命令”对话框, 如图 4.12 所示。

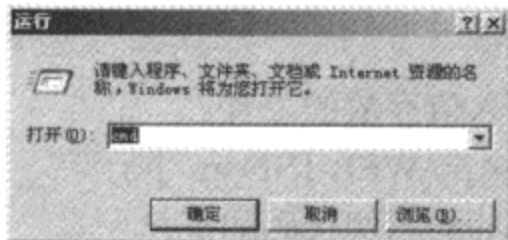


图 4.11 “运行”对话框

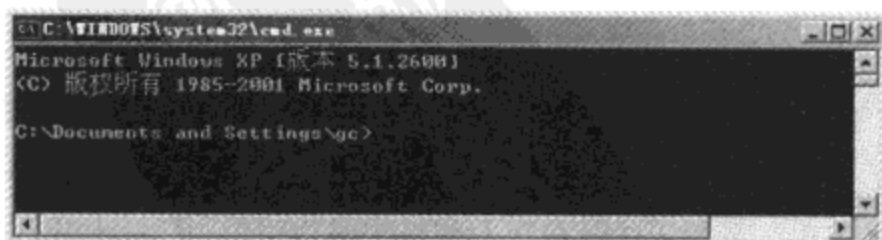


图 4.12 “cmd 命令”对话框

(10) 在“cmd 命令”对话框中, 输入“cd\”回到 C 盘根目录下。

(11) 再输入 “native2ascii messages.properties messages.txt”，然后按 Enter 键，如图 4.13 所示。

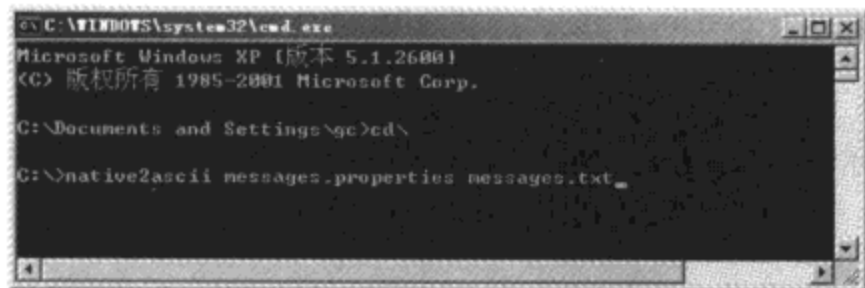


图 4.13 输入 “native2ascii messages.properties messages.txt” 后的 “cmd 命令” 对话框

(12) 这时在 C 盘的根目录下就会产生一个 messages.txt。messages.txt 的内容如下：

```
HelloWorld=\u95ee\u5019\u8bed\u5339 {0} \u95ee\u5019\u65f6\u95f4\u5339 {1}
```

(13) 将 messages.txt 中的内容复制到 myApp/WEB-INF/src/messages.properties 中，覆盖原来的内容。

(14) 再次运行测试程序，查看输出结果，如图 4.14 所示。

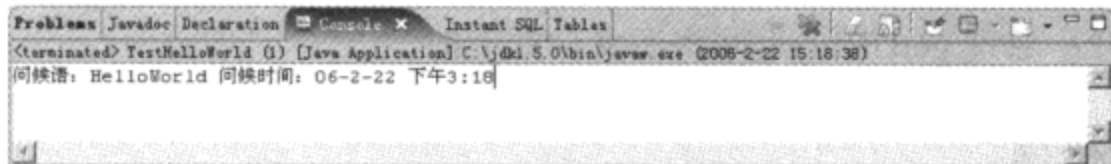


图 4.14 输出问候语和问候时间

(15) 在 myApp/WEB-INF/src 下用同样的方法新建一个 messages_en_US.properties 文件，输入内容如下：

```
HelloWorld=Language {0} Time {1}
```

(16) 修改测试程序 TestHelloWorld，将 Locale.CHINA 改为 Locale.US。

(17) 再次运行测试程序，查看英文输出结果如图 4.15 所示。

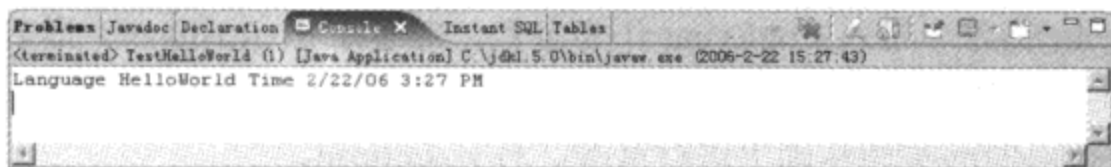


图 4.15 输出英文的问候语和问候时间

通过上面的示例可以看到，只需要修改 Locale，就可以很容易地实现对国际化的支持。

4.9.2 资源访问

很多时候应用程序都需要存取资源。Spring 提供了对资源文件的存取。ApplicationContext 继承了 ResourceLoader 接口，开发人员可以使用 getResource() 方法并指定资源文件的 URL 来存取。

ApplicationContext 对资源文件的存取在设定资源文件的路径上有如下 3 种方式：

第一种，通过虚拟路径来存取。如果资源文件位于 CLASSPATH 下，可以通过这种方

法来获取。示例代码如下：

```
ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
Resource resource = actx.getResource("classpath: messages.properties");
```

这里要说明的是“classpath:”是 Spring 约定的 URL 虚拟路径。

第二种，通过实际路径来存取。指定标准的 URL，例如“file:”或“http:”。示例代码如下：

```
ApplicationContext actx =new FileSystemXmlApplicationContext("config.xml");
Resource resource = actx.getResource("file:d:/eclipse/workspace/myApp/WEB-INF/src /messages.
properties");
```

第三种，通过相对路径来存取。示例代码如下：

```
ApplicationContext actx =new FileSystemXmlApplicationContext("config.xml");
Resource resource = actx.getResource("WEB-INF/ src /messages.properties");
```

当通过 ApplicationContext 取得一个 Resource 后，开发人员就可以使用 getFile()来存取资源文件的内容了。还可以通过 exists()来检查资源文件是否存在；通过 isOpen()检查资源文件是否被打开；通过 getURL()返回资源文件的 URL。

4.9.3 事件传递

ApplicationContext 中的事件处理是通过 ApplicationEvent 类和 ApplicationListener 接口来提供的。通过 ApplicationContext 的 publishEvent()方法来通知 ApplicationListener。

接下来实现一个程序用来输出日志信息。其实现思路是：首先定义一个继承 ApplicationEvent 的类 LogEvent。LogEvent 类就是通过 ApplicationContext 被发布出去的。然后定义一个实现 ApplicationListener 接口的类 LogListener，则 ApplicationContext 会在发布 LogEvent 事件时通知 LogListener。接着实现一个 ApplicationContextAware 接口的类 Log，通过 publishEvent()方法，带入 LogEvent 作为参数，来通知 LogListener。最后编写一个测试程序来输出规定格式的信息。具体步骤如下：

(1) 继续在前面的实例基础上进行修改。打开 Eclipse。

(2) 在 com.gc.action 包中定义一个继承 ApplicationEvent 的类 LogEvent。LogEvent 类就是通过 ApplicationContext 被发布出去的。LogEvent.java 的示例代码如下：

```
/** ***** LogEvent.java ***** */
package com.gc.action;

import org.springframework.context.ApplicationEvent;
// LogEvent 类就是通过 ApplicationContext 被发布出去的
public class LogEvent extends ApplicationEvent {
    public LogEvent(Object msg) {
        super(msg);
    }
}
```

(3) 在 com.gc.action 包中定义一个实现 ApplicationListener 接口的类 LogListener, 则 ApplicationContext 会在发布 LogEvent 事件时通知 LogListener, 并输出相应的消息。LogListener.java 的示例代码如下:

```
//*****LogListener.java*****
package com.gc.action;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class LogListener implements ApplicationListener {
    //ApplicationContext 会在发布 LogEvent 事件时通知 LogListener
    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof LogEvent) {
            //设定时间
            SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            format.setLenient(false);
            String currentDate = format.format(new Date());
            System.out.println("输出时间: " + currentDate + " 输出内容: " + event.toString());
        }
    }
}
```

(4) 在 com.gc.action 包中实现一个 ApplicationContextAware 接口的类 Log, 通过 publishEvent()方法, 带入 LogEvent 作为参数, 来通知 LogListener。Log.java 的示例代码如下:

```
//*****Log.java*****
package com.gc.action;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class Log implements ApplicationContextAware {
    //设定变量 applicationContext
    private ApplicationContext applicationContext;
    //变量 applicationContext 的 set 方法
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
    //通过 publishEvent 发布事件
    public int log(String log) {
        LogEvent event = new LogEvent(log);
        this.applicationContext.publishEvent(event);
        return 0;
    }
}
```

```
}  
}
```

(5) 定义配置文档 config.xml。config.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <!--负责国际化支持-->  
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundle-  
MessageSource">  
        <property name="basename">  
            <value>messages</value>  
        </property>  
    </bean>  
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" >  
    </bean>  
    <bean id="date" class="java.util.Date"/>  
    <!--负责事件传递-->  
    <bean id="log" class="com.gc.action.Log"/>  
    <bean id="listener" class="com.gc.action.LogListener"/>  
</beans>
```

(6) 在 com.gc.test 包中编写一个测试程序 TestHelloWorld 来输出规定的内容。TestHelloWorld.java 的示例代码如下：

```
//***** TestHelloWorld.java *****  
package com.gc.test;  
import java.util.Calendar;  
import com.gc.action.Log;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.FileSystemXmlApplicationContext;  
  
import com.gc.action.HelloWorld;  
public class TestHelloWorld {  
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,  
ClassNotFoundException {  
        //通过 ApplicationContext 获取配置文档  
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");  
        Log log = (Log)actx.getBean("log");  
        log.log("gf");  
    }  
}
```

(7) 运行测试程序，输出结果如图 4.16 所示。

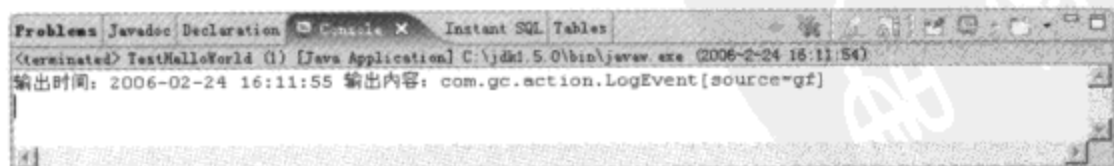


图 4.16 输出日志信息

其实对于日志输出来说，上面的实例只是为了说明事件传递的功能。如果用 Spring 的 AOP 来实现日志的输出将是更好的选择。

4.10 小 结

本章主要讲述了 Spring 的核心容器。Spring 的核心容器实现了 IoC，而 Bean 是了解核心容器的基础。所以在本章中首先讲解了 Spring 的基础 Bean 的相关知识，然后介绍了 Spring 管理 Bean 的 3 种方式，最后讲解了 Spring 提供的一些国际化、资源获取、事件传递等功能。了解了 Spring 的核心和基础知识后，接下来的第 5 章将讲述 Spring 更激动人心的功能——AOP。



第 5 章 Spring 的 AOP

在第 4 章中，笔者通过事件传递的机制实现了日志的输出，其实 Spring 的 AOP 在这方面提供了更强大的功能。本章将首先让读者了解 AOP 的基本思想，然后通过对 Java 代理机制进行分析，引出 AOP 的 3 个关键概念，接着对 Spring 的 AOP 进行介绍，最后通过一个完整的实例让读者掌握 Spring AOP 的使用方法。

5.1 AOP 基本思想

AOP 的意思是面向方面编程，英文全称是 Aspect Oriented Programming，它使开发人员可以更好地将本不该彼此粘合在一起的功能分离开。

5.1.1 认识 AOP

开发人员在编写应用程序时，通常包含两种代码：一种是和业务系统有关的代码，一是和业务系统关系不大的代码，例如日志、权限、异常处理、事务处理等。以前编写代码时，这两种代码基本是写在一起的，这样在程序中，到处充满着相同或类似的代码，例如日志信息的输出，每个方法都要写日志的输出，不利于程序的维护。而 AOP 就是使这两种代码分离的思想。使用 AOP，就不用业务逻辑中实现与业务功能关系不大的代码，从而降低了两种代码的耦合性，达到易于维护和重用的目的。

AOP，从字面的意思来理解就是面向方面编程，但却容易让初学者一头雾水，很多资料也都对 AOP 进行过解释，但过多地从理论的角度进行讲解其实用性，这里不再多讲，仅用一个例子来说明：基本上每个方法都要用日志进行记录，那么如果按照面向对象的思路来说，就是每个对象都有记录日志这样一个行为。要在每个方法里添加日志的信息，必然会产生大量的重复代码，但可以将记录日志看作是一个横切面，所有对这些方法的调用都要经过这个横切面，然后在这个横切面进行记录日志的操作，这样就达到代码重用和易于维护的目的了，这就是 AOP 的思想。

5.1.2 AOP 与 OOP 对比分析

OOP（面向对象编程）对现代编程产生了深远的影响，经过多年的发展，目前已日趋成熟，它能很好地解决软件系统中角色划分的问题，借助于面向对象的分析、设计和实现，开发人员可以将现实领域的实体转换成软件系统中的对象，从而很自然地完成从现实到软

件的转换。但 OOP 并不是一种完美的思想，在某些方面，OOP 也有其不足之处。比如在日志、事务处理、权限管理等方面，当应用 OOP 将这些内容封装为对象的行为时，会产生大量的重复代码，虽然通过一些方法可以减少这种重复，但却不能彻底地解决该问题，于是 AOP 出现了。前面讲过 AOP 能够降低代码的耦合性，使得代码易于维护和重用。一个应用程序分为核心关注点和横切关注点。核心关注点和具体应用的功能相关，而横切关注点存在于整个系统的范围内。

在 AOP 里，每个关注点的实现并不知道是否有其他关注点关注它，这是 AOP 和 OOP 的主要区别。在 AOP 里，组合的流向是从横切关注点到主关注点，而 OOP 中组合的流向则是从主关注点到横切关注点。从这点可以看出 AOP 和 OOP 它们所关注的对象是不同的，所以 AOP 可以和 OOP 很好地共存，AOP 是 OOP 的有益补充，而不是其对立面。

5.1.3 AOP 与 Java 的代理机制

AOP 是一种思想，它和具体的实现技术无关。任何一种符合 AOP 思想的技术实现，都可以看作是 AOP 的实现。

JDK 1.3 以后，Java 提供了动态代理的机制。通过 Java 的动态代理机制，就可以很容易地实现 AOP 的思想。实际上 Spring 的 AOP 也是建立在 Java 的代理机制之上的。要理解 Spring 的 AOP，先来了解 Java 的代理机制。下面主要通过一个输出日志的实例来使读者先了解 Java 的代理机制，从而引出 AOP 的几个关键点。

5.2 从一个输出日志的实例分析 Java 的代理机制

上面讲到，要了解 Spring 的 AOP，先来了解 Java 的代理机制。本节主要通过一个输出日志的实例来分析 Java 的代理机制。首先介绍以前写日志的时候是怎么实现的，然后讲解使用 Java 的代理机制怎么实现日志的输出，接着讲解怎样通过 Java 的动态代理机制把这个日志输出改成通用的，最后引出 AOP 的几个关键点。

5.2.1 通用的日志输出方法

在笔者使用 Spring 以前开发的程序中，不管是使用 Java 自动的日志工具，还是使用 Log4j，或是自己编写的日志工具，都要在每一个业务逻辑方法里编写记录日志的代码。使用 AOP 就可以使业务逻辑和记录日志这两件事情分离开。这个输出日志实例的实现思路是：首先给出原来在程序中编写日志的方法，然后编写测试程序，查看输出结果，最后对这种方法进行总结，指出这种方法的缺点。具体编写步骤如下：

(1) 打开 Eclipse，在 `com.gc.action` 包中建立一个 Java 文件 `TimeBook.java`，用来模拟实际业务中考勤审核的业务逻辑。

(2) 原来在程序中编写日志时，都要在每一个业务逻辑方法里编写记录日志的代码。

TimeBook.java 的示例代码如下：

```
/****** TimeBook.java*****  
import org.apache.log4j.Level;  
import org.apache.log4j.Logger;  
public class TimeBook {  
    private Logger logger = Logger.getLogger(this.getClass().getName());  
    //审核数据的相关程序  
    public void doAuditing(String name) {  
        logger.log(Level.INFO, name + " 开始审核数据....");  
        //审核数据的相关程序  
        .....  
        logger.log(Level.INFO, name + " 审核数据结束....");  
    }  
}
```

代码说明：

- ☐ 在业务逻辑中使用 log4j 作为日志输出的工具。
- ☐ doAuditing()方法用来处理实际业务中的考勤审核。
- ☐ 参数 name，用来传入是谁执行了类 TimeBook 中的 doAuditing()方法。
- ☐ 在审核代码的前后添加了用 logger.log()方法实现日志输出的功能。

(3) 编写测试程序，继续在以前的测试程序 TestHelloWorld 的基础上进行修改，TestHelloWorld.java 的示例代码如下：

```
/****** TestHelloWorld.java*****  
package com.gc.test;  
import com.gc.action.TimeBook;  
public class TestHelloWorld {  
    public static void main(String[] args) {  
        TimeBook timeBook = new TimeBook();  
        timeBook.doAuditing("张三");  
    }  
}
```

代码说明：timeBook.doAuditing("张三")表示该程序的执行人是“张三”。

(4) 运行测试程序，查看通过 TimeBook 类输出的日志信息，如图 5.1 所示。

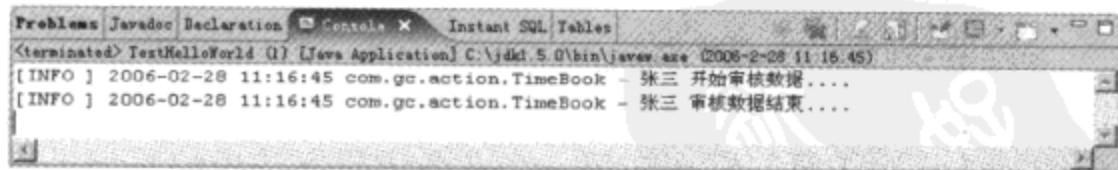


图 5.1 通过 TimeBook 类输出日志信息

在上面的示例中，笔者把日志信息添加在了具体的业务逻辑中，假如程序中其他的代码都需要日志输出的功能，那么每个程序就都要添加和上面类似的代码。这样，在程序中，就会存在很多类似的日志输出代码，造成了很大的耦合，通过什么方法可以使业务逻辑和输出日志的代码分离呢？通过面向接口编程可以改进这个问题。

5.2.2 通过面向接口编程实现日志输出

通过前面的示例程序，读者可以了解到以前添加日志信息方法的缺点。下面主要通过面向接口编程来改进这个缺点。其实现思路是：首先把执行考勤审核的 `doAuditing()` 方法提取出来成为接口，然后通过一个实体类来实现这个方法，在这个方法里编写具体的考勤审核的业务逻辑，接着通过一个代理类来进行日志输出，最后编写测试程序，查看输出结果。具体步骤如下：

(1) 在 `com.gc.impl` 包中，建立一个接口 `TimeBookInterface`。`TimeBookInterface.java` 的示例代码如下：

```
//***** TimeBookInterface.java*****
package com.gc.impl;

import org.apache.log4j.Level;
//通过面向接口编程实现日志输出
public interface TimeBookInterface {
    public void doAuditing(String name);
}
```

(2) 在 `com.gc.action` 包中，使前面已经建立好的类 `TimeBook` 实现接口 `TimeBookInterface`，在 `doAuditing()` 方法中编写具体的考勤审核代码。`TimeBook.java` 的示例代码如下：

```
//***** TimeBook.java*****
package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
    public void doAuditing(String name) {
        //审核数据的相关程序
        .....
    }
}
```

(3) 编写一个代理类，用来实现日志的输出，在该类中针对前面的接口 `TimeBookInterface` 编程，而不针对具体的类，从而实现具体业务逻辑与日志输出代码。`TimeBookProxy.java` 的示例代码如下：

```
//***** TimeBookProxy.java*****
package com.gc.action;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;

import com.gc.impl.TimeBookInterface;

public class TimeBookProxy {
    private Logger logger = Logger.getLogger(this.getClass().getName());
```



```
private TimeBookInterface timeBookInterface;
//在该类中针对前面的接口 TimeBookInterface 编程，而不针对具体的类
public TimeBookProxy(TimeBookInterface timeBookInterface) {
    this.timeBookInterface = timeBookInterface;
}
//实际业务处理
public void doAuditing(String name) {
    logger.log(Level.INFO, name + " 开始审核数据....");
    timeBookInterface.doAuditing(name);
    logger.log(Level.INFO, name + " 审核数据结束....");
}
}
```

(4) 修改测试程序 TestHelloWorld，把类 TimeBook 当作参数传入代理类 TimeBookProxy 中，从而实现对具体负责考勤审核类 TimeBook 的调用。TestHelloWorld.java 的示例代码如下：

```
//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
public class TestHelloWorld {
    public static void main(String[] args) {
        //这里针对接口进行编程
        TimeBookProxy timeBookProxy = new TimeBookProxy(new TimeBook());
        timeBookProxy.doAuditing("张三");
    }
}
```

(5) 运行测试程序，可以得到通过 TimeBookProxy 类输出日志信息，如图 5.2 所示。

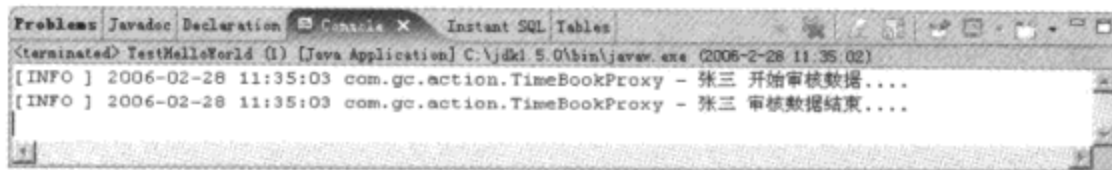


图 5.2 通过 TimeBookProxy 类输出日志信息

和前面一个日志输出做对比，可以看到，在这个示例中，具体负责考勤审核的业务逻辑代码和日志信息的代码分离开了，并且以后只要实现了接口 TimeBookInterface 的类，都可以通过代理类 TimeBookProxy 实现日志信息的输出，而不用再每个类里面都写日志信息输出的代码，从而实现了日志信息的代码重用。

5.2.3 使用 Java 的代理机制进行日志输出

前面的代码虽然有了一些改进，但是仍然有一定局限性，因为要使用代理类，就必须实现固定的接口，有没有一种通用的机制，不管是不是实现这个接口，都可以实现日志信息的输出呢？

Java 提供的 InvocationHandler 接口可以实现这种功能，首先编写一个日志信息的代理

类，这个代理类实现了接口 `InvocationHandler`，然后和前面一个实例类似，编写一个接口，并实现这个接口，在实现类中编写具体的考勤审核代码，最后针对接口编写测试类，查看测试结果。具体步骤如下：

(1) 编写一个日志信息的代理类 `LogProxy`，这个代理类实现了接口 `InvocationHandler`，可以对任何接口实现日志信息的输出。`LogProxy.java` 的示例代码如下：

```
/** ***** LogProxy.java ***** */
package com.gc.action;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
//代理类实现了接口 InvocationHandler
public class LogProxy implements InvocationHandler {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private Object delegate;
    //绑定代理对象
    public Object bind(Object delegate) {
        this.delegate = delegate;
        return Proxy.newProxyInstance(delegate.getClass().getClassLoader(), delegate.getClass().getInterfaces(), this);
    }
    //针对接口编程
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result = null;
        try {
            //在方法调用前后进行日志输出
            logger.log(Level.INFO, args[0] + " 开始审核数据....");
            result = method.invoke(delegate, args);
            logger.log(Level.INFO, args[0] + " 审核数据结束....");
        } catch (Exception e) {
            logger.log(Level.INFO, e.toString());
        }
        return result;
    }
}
```

(2) 使用 `com.gc.impl` 包中的接口 `TimeBookInterface`。`TimeBookInterface.java` 的示例代码如下：

```
/** ***** TimeBookInterface.java ***** */
package com.gc.impl;

import org.apache.log4j.Level;
//针对接口编程
public interface TimeBookInterface {
```

```
public void doAuditing(String name);  
}
```

(3) 使用 `com.gc.action` 包中的类 `TimeBook`, `doAuditing()` 方法中编写具体的考勤审核代码。`TimeBook.java` 的示例代码如下:

```
/****** TimeBook.java *****  
package com.gc.action;  
import com.gc.impl.TimeBookInterface;  
public class TimeBook implements TimeBookInterface {  
    public void doAuditing(String name) {  
        //审核数据的相关程序  
        .....  
    }  
}
```

(4) 修改测试代码 `TestHelloWorld`, 使用日志代理类 `LogProxy` 实现日志的输出。`TestHelloWorld.java` 的示例代码如下:

```
/****** TestHelloWorld.java *****  
package com.gc.test;  
import com.gc.action.TimeBook;  
import com.gc.action.TimeBookProxy;  
import com.gc.impl.TimeBookInterface;  
import com.gc.action.LogProxy;  
public class TestHelloWorld {  
    public static void main(String[] args) {  
        //实现了对日志类的重用  
        LogProxy logProxy = new LogProxy();  
        TimeBookInterface timeBookProxy = (TimeBookInterface)logProxy.bind(new TimeBook());  
        timeBookProxy.doAuditing("张三");  
    }  
}
```

(5) 运行测试程序, 可以得到通过 `LogProxy` 类输出日志信息, 如图 5.3 所示。

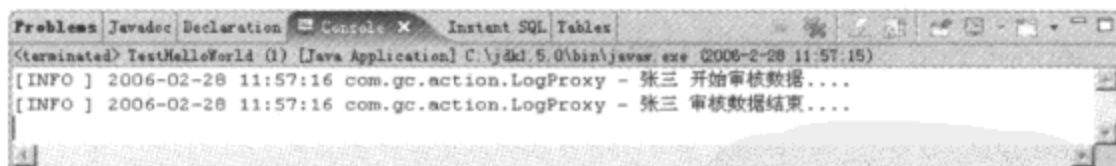


图 5.3 通过 `LogProxy` 类输出日志信息

这种方式, 对于其他的类也同样适用, 这样就真正地实现了业务逻辑和输出日志信息代码的分离。

5.2.4 对这 3 种实现方式进行总结

第一种方式, 需要在每个类里都增加对输出日志信息的代码; 第二种方式, 虽然实现了业务逻辑与输出日志信息代码的分离, 但还是必须依赖于固定的接口; 第三种方式, 真

正实现了对输出日志信息代码的重用，并且不依赖于固定的接口实现。

从第三种方式中，也可以看出 Java 动态代理机制的强大，而 Spring 的 AOP 正是建立在 Java 动态代理的基础上的，当读者通过上面的示例一步一步地对 Java 的动态代理机制有了一定的了解后，接下来就可以逐渐地进入 AOP 了。

5.3 AOP 的 3 个关键概念

因为 AOP 的概念难于理解，所以在前面首先对 Java 动态代理机制进行了一下讲解，从而使读者能够循序渐进地来理解 AOP 的思想。

学习 AOP，关键在于理解 AOP 的思想，能够使用 AOP。对于 AOP 众多的概念，读者只要理解 3 个重要的概念即可。这 3 个概念是 Pointcut、Advice 和 Advisor。

5.3.1 切入点（Pointcut）

在介绍 Pointcut 前，有必要先介绍一下 Join Point（连接点）的概念。Join Point 指的是程序运行中的某个阶段点，如某个方法调用、异常抛出等。前面示例中的 doAuditing() 方法就是一个 Join Point，表示程序是要在这个地方加入 Advice。

Pointcut 是 Join Point 的集合，它是程序中需要注入 Advice 的位置的集合，指明 Advice 要在什么样的条件下才能被触发。

org.springframework.aop.Pointcut 接口用来指定通知到特定的类和方法。查看 Spring 下载包里的源文件 Pointcut.java，路径是 spring-framework-2.0-m1\src\org\springframework\aoop，可以看到 Pointcut.java 源代码如下：

```
//***** Pointcut.java*****
package org.springframework.aop;

public interface Pointcut {
    //用来将切入点限定在给定的目标类中
    ClassFilter getClassFilter();
    //用来判断切入点是否匹配目标类给定的方法
    MethodMatcher getMethodMatcher();

    Pointcut TRUE = TruePointcut.INSTANCE;
}
```

代码说明：

- ❑ 接口 ClassFilter，用来将切入点限定在给定的目标类中。
- ❑ 接口 MethodMatcher，用来判断切入点是否匹配目标类给定的方法。

从上面可以看出，在接口 Pointcut 中，主要包含两个接口：ClassFilter 和 MethodMatcher，有利于代码的重用。

5.3.2 通知 (Advice)

Advice 是某个连接点所采用的处理逻辑，也就是向连接点注入的代码。前面示例中提取出来输出日志信息的代码就是一个 Advice，表示要在 Join Point 加入这段代码。

5.3.3 Advisor

Advisor 是 Pointcut 和 Advice 的配置器，它包括 Pointcut 和 Advice，是将 Advice 注入程序中 Pointcut 位置的代码。

上面只是粗略地对 AOP 的 3 个概念进行一下说明，目的是让读者能够较快地进入到 AOP 中，接下来将会分别对这 3 个概念进行更加详细的讲解。

5.4 Spring 的 3 种切入点 (Pointcut) 实现

上节讲过，Pointcut 是 Join Point 的集合，它是程序中需要注入 Advice 的位置的集合。Spring 主要提供了 3 种切入点 (Pointcut) 的实现：静态切入点、动态切入点和自定义切入点，下面分别进行讲解。

5.4.1 静态切入点

静态切入点只限于给定的方法和目标类，而不考虑方法的参数。Spring 在调用静态切入点时只在第一次的时候计算静态切入点的位置，然后把它缓存起来，以后就不需要再进行计算。使用 `org.springframework.aop.support.RegexpMethodPointcut` 可以实现静态切入点，`RegexpMethodPointcut` 是一个通用的正则表达式切入点，它是通过 Jakarta ORO 来实现的，需要把 `jakarta-oro-2.0.8.jar` 加入到 ClassPath 中，它的正则表达式语法和 Jakarta ORO 的正则表达式语法是一样的。使用 `RegexpMethodPointcut` 的一个示例代码如下：

```
<bean id="settersAndAbsquatulatePointcut"
      class="org.springframework.aop.support.RegexpMethodPointcut">
  <property name="patterns">
    <!--设定切入点-->
    <list>
      <value>.*save.*</value>
      <value>.*do.*</value>
    </list>
  </property>
</bean>
```

代码说明：

- `.*save.*`，表示所有以 `save` 开头的方法都是切入点。

□ `* do.*`, 表示所有以 `do` 开头的方法都是切入点。

5.4.2 动态切入点

动态切入点与静态切入点的区别是, 它不仅限于给点的方法和类, 动态切入点还可以指定方法的参数。因为参数的变化性, 所以动态切入点不能缓存, 需要每次调用的时候都进行计算, 因此使用动态切入点有很大的性能损耗。

当切入点需要在执行时根据参数值来调用通知时, 就需要使用动态切入点。Spring 提供了一个内建的动态切入点: 控制流切入点。此切入点匹配基于当前线程的调用堆栈。开发人员只有在当前线程执行时找到特定的类和特定的方法才返回 `true`。

其实大多数的切入点可以使用静态切入点, 所以很少有机会创建动态切入点。

5.4.3 自定义切入点

因为 Spring 中的切入点是 Java 类, 而不是语言特性 (如 AspectJ), 因此可以定义自定义切入点。因为 AOP 还没有完全成熟, Spring 提供的文档在这方面也没有提供更详细的解释, 所以这里将不再对动态切入点和自定义切入点进行更加详细的介绍。

5.5 Spring 的通知 (Advice)

Spring 提供了 5 种 Advice 类型: Interception Around、Before、After Returning、Throw 和 Introduction。它们分别在以下情况下被调用: 在 `JointPoint` 前后、`JointPoint` 前、`JointPoint` 后、`JointPoint` 抛出异常时、`JointPoint` 调用完毕后。下面来进行更详细的讲解。

5.5.1 Interception Around 通知

Interception Around 通知会在 `JointPoint` 的前后执行, 前面示例中的 `LogProxy` 就是一个 Interception Around 通知, 它在考勤审核程序的前后都执行了。Spring 中最基本的通知类型便是 Interception Around 通知。实现 Interception Around 通知的类需要实现接口 `MethodInterceptor`, 示例代码如下:

```
public class LogInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println(" 开始审核数据...");
        Object rval = invocation.proceed();
        System.out.println(" 审核数据结束...");
        return rval;
    }
}
```

5.5.2 Before 通知

Before 通知只在 JointPoint 前执行, 实现 Before 通知的类需要实现接口 MethodBeforeAdvice, 示例代码如下:

```
public class LogBeforeAdvice implements MethodBeforeAdvice {  
    public void before(Method m, Object[] args, Object target) throws Throwable {  
        System.out.println(" 开始审核数据...");  
    }  
}
```

5.5.3 After Returning 通知

After Returning 通知只在 JointPoint 后执行, 实现 After Returning 通知的类需要实现接口 AfterReturningAdvice, 示例代码如下:

```
public class LogAfterAdvice implements AfterReturningAdvice {  
    public void afterReturning (Method m, Object[] args, Object target) throws Throwable {  
        System.out.println(" 审核数据结束...");  
    }  
}
```

5.5.4 Throw 通知

Throw 通知只在 JointPoint 抛出异常时执行, 实现 Throw 通知的类需要实现接口 ThrowsAdvice, 示例代码如下:

```
public class LogThrowAdvice implements ThrowsAdvice {  
    public void afterThrowing (RemoteException ex) throws Throwable {  
        System.out.println(" 审核数据抛出异常, 请检查..." + ex);  
    }  
}
```

5.5.5 Introduction 通知

Introduction 通知只在 JointPoint 调用完毕后执行, 实现 Introduction 通知的类需要实现接口 IntroductionAdvisor 和接口 IntroductionInterceptor。

前面所讲的知识点, 更多的是理论, 下面的章节将会讲述更多的实例, 来帮助读者更好地理解上面的理论知识。

5.6 Spring 的 Advisor

前面讲过, Advisor 是 Pointcut 和 Advice 的配置器, 它是将 Advice 注入程序中 Pointcut

位置的代码。org.springframework.aop.support.DefaultPointcutAdvisor 是最通用的 Advisor 类。在 Spring 中，主要通过 XML 的方式来配置 Pointcut 和 Advice。

5.7 用 ProxyFactoryBean 创建 AOP 代理

使用 Spring 提供的类 org.springframework.aop.framework.ProxyFactoryBean 是创建 AOP 的最基本的方式。

5.7.1 使用 ProxyFactoryBean 代理目标类的所有方法

在 Spring 中，ProxyFactoryBean 是在 XML 中进行配置的，它的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="log" class="com.gc.action.LogAround"/>
    <bean id="timeBook" class="com.gc.action.TimeBook"/>
    <!--设定代理类-->
    <bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <!--这里代理的是接口-->
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <!--程序中的 Advice-->
        <property name="target">
            <ref bean="timeBook"/>
        </property>
        <!--是 ProxyFactoryBean 要代理的目标类-->
        <property name="interceptorNames">
            <list>
                <value>log</value>
            </list>
        </property>
    </bean>
</beans>
```

代码说明：

- ❑ id 为 log 的 Bean，是程序中的 Advice。
- ❑ id 为 timeBook 的 Bean，是 ProxyFactoryBean 要代理的目标类。
- ❑ id 为 logProxy 的 Bean，就是 ProxyFactoryBean。
- ❑ ProxyFactoryBean 的 proxyInterfaces 属性，指明要代理的接口。
- ❑ ProxyFactoryBean 的 target 属性，指明要代理的目标类，这个目标类实现了上面 proxyInterfaces 属性指定的接口。

- ❑ ProxyFactoryBean 的 interceptorNames 属性，指明要在代理的目标类中插入的 Advice。
- ❑ ProxyFactoryBean 还有一个 proxyTargetClass 属性，如果这个属性被设定为“true”，说明 ProxyFactoryBean 要代理的不是接口类，而是要使用 CGLIB 方式来进行代理，后面会详细讲解使用 CGLIB 方式来进行代理。

🔔注意：ProxyFactoryBean 的 proxyInterfaces 属性只支持使用字符串的方式进行注入，不支持使用 Bean 的依赖方式进行注入。

5.7.2 使用 ProxyFactoryBean 代理目标类的指定方法

在上面的示例中，Advice 会代理目标类的所有方法。如果要代理目标类的指定方法，则需要使用 Spring 提供的 org.springframework.aop.support.RegexpMethodPointcutAdvisor 类。代理目标类的指定方法的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="log" class="com.gc.action.LogAround"/>
    <bean id="timeBook" class="com.gc.action.TimeBook"/>
    <!--代理目标类的指定方法-->
    <bean id="logAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="log"/>
        </property>
        <!--指定要代理的方法-->
        <property name="patterns">
            <value>.*doAuditing.* </value>
        </property>
    </bean>
    <!--设定代理类-->
    <bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <property name="target">
            <ref bean="timeBook"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>logAdvisor</value>
            </list>
        </property>
    </bean>
</beans>
```

代码说明:

- ❑ 在 id 为 logAdvisor 的 Bean 中设定 Advice 和要指定的方法。
- ❑ 把 id 为 logProxy 的 Bean 的 interceptorNames 属性值改为 logAdvisor。
- ❑ logAdvisor 的 advice 属性指定 Advice。
- ❑ logAdvisor 的 patterns 属性指定要代理的方法。“doAuditing”表示只有 doAuditing() 方法才使用指定的 Advice。

patterns 属性值使用的是正则表达式, 关于正则表达式的使用, 下节将会进行讲解。

⚠注意: 因为要使用正则表达式, 所以要把 spring-framework-2.0-m1\lib\oro 目录下的 jakarta-oro-2.0.8.jar 加入到 ClassPath 下, 加入方法可参考第 2 章。

5.7.3 正则表达式简介

正则表达式最早是由数学家 Stephen Kleene 于 1956 年在对自然语言的递增研究成果的基础上提出来的。正则表达式并非一门专用语言, 但它可用于在一个文件中查找字符。下面把几个常用的进行一下讲解:

- (1) “.”, 可以用来匹配任何一个字符。比如: 正则表达式为 “g.f”, 它就会匹配 “gaf”、“glf”、“g*f” 和 “g#f” 等。
- (2) “[]”, 只有 [] 里指定的字符才能匹配。比如: 正则表达式为 “g[abc]f”, 它就只能匹配 “gaf”、“gbf” 和 “gcf”, 而不会匹配 “glf”、“g*f” 和 “g#f” 等。
- (3) “*”, 表示匹配次数, 可以任意次, 用来确定紧靠该符号左边的符号出现的次数。比如: 正则表达式为 “g.*f”, 它能匹配 “gaf”、“gaaf”、“gf” 和 “g*f” 等。
- (4) “?”, 可以匹配 0 或 1 次, 用来确定紧靠该符号左边的符号出现的次数。比如: 正则表达式为 “g.?f”, 它能匹配 “gaf” “g*f” 等。
- (5) “\”, 是正则表达式的连接符。比如: 正则表达式为 “g.\-f”, 它能匹配 “g-f”、“ga-f” 和 “g*-f” 等。

上面的示例中, 只对 TimeBook 类的 doAuditing() 方法有效, 如果要对 TimeBook 类中所有以 do 开头的方法有效, 可以这样设定 “.*do.*”, 示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="log" class="com.gc.action.LogAround"/>
  <bean id="timeBook" class="com.gc.action.TimeBook"/>
  <!--代理目标类的指定方法-->
  <bean id="logAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref bean="log"/>
    </property>
    /*对所有以 do 开头的方法有效*/
    <property name="patterns">
```

```

        <value>.*do.* </value>
    </property>
</bean>
<!--设定代理类-->
<bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
<!--指定要代理的类-->
    <property name="interceptorNames">
        <list>
            <value>logAdvisor</value>
        </list>
    </property>
</bean>
</beans>

```

上述代码对 TimeBook 类中所有以 do 开头的方法都有效。如果要对 TimeBook 类中所有方法有效，可以这样设定 “.*com\.gc\.impl\.TimeBookInterface.*”，示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="log" class="com.gc.action.LogAround"/>
    <bean id="timeBook" class="com.gc.action.TimeBook"/>
    <!--代理目标类的指定方法-->
    <bean id="logAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="log"/>
        </property>
        /*注意包路径的定义*/
        <property name="patterns">
            <value>.*com\.gc\.impl\.TimeBookInterface.*</value>
        </property>
    </bean>
    <!--设定代理类-->
    <bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <property name="target">
            <ref bean="timeBook"/>
        </property>
        <!--指定要代理的类-->
        <property name="interceptorNames">
            <list>

```

```
        <value>logAdvisor</value>
      </list>
    </property>
  </bean>
</beans>
```

上述代码对 TimeBook 类中所有的方法都有效。

🔔注意：包的路径要用 “\” 连接符来表示 “.” 不是正则表达式的 “.”。

5.8 把输出日志的实例改成用 Spring 的 AOP 来实现

在前面输出日志信息的实例中，没有用到 Spring 的任何组件，只是使用了 Java 的动态代理机制，但是却实实在在地体现了 AOP 的思想，如果使用 Spring 提供的 AOP 功能，应该怎样来实现前面那个实例呢？下面笔者就主要讲解如何把前面的那个实例改成通过 Spring 提供的 AOP 来实现。

5.8.1 采用 Interception Around 通知的形式实现

Interception Around 通知会在 JointPoint 的前后执行，实现 Interception Around 通知的类需要实现接口 MethodInterceptor。其实现思路是：首先实现接口 MethodInterceptor，在 invoke() 方法里编写负责输出日志信息的代码，具体的业务逻辑还使用前面的接口 TimeBookInterface 和它的实现类 TimeBook，然后在 Spring 的配置文档中定义 Pointcut，最后编写测试程序，执行测试程序，查看输出结果。具体步骤如下：

(1) 编写负责输出日志信息的类 LogAround，该类实现了接口 MethodInterceptor，重写了 invoke() 方法。LogAround.java 的示例代码如下：

```
/** ***** LogAround.java ***** */
package com.gc.action;

import org.aopalliance.intercept.MethodInvocation;
import org.aopalliance.intercept.MethodInterceptor;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
// Interception Around 通知会在 JointPoint 的前后执行
public class LogAround implements MethodInterceptor {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //负责输出日志信息的代码
    public Object invoke(MethodInvocation mi) throws Throwable {
        logger.log(Level.INFO, mi.getArguments()[0] + " 开始审核数据....");
        try {
            Object result = mi.proceed();
            //返回值即是被调用的方法的返回值
            return result;
        }
    }
}
```



```

    }
    finally {
        logger.log(Level.INFO, mi.getArguments()[0] + " 审核数据结束....");
    }
}
}

```

代码说明：

- ❑ 参数 MethodInvocation，通过它可以获得方法的名称、程序传入的参数 Object[] 等。
- ❑ proceed() 方法，通过它即可执行被调用的方法。
- ❑ return result，返回值即是被调用的方法的返回值。

(2) 使用 com.gc.impl 包中的接口 TimeBookInterface。TimeBookInterface.java 的示例代码如下：

```

//***** TimeBookInterface.java *****
package com.gc.impl;

import org.apache.log4j.Level;

public interface TimeBookInterface {
    //负责具体的业务逻辑
    public void doAuditing(String name);
}

```

(3) 使用 com.gc.action 包中的类 TimeBook，doAuditing() 方法中编写具体的考勤审核代码。TimeBook.java 的示例代码如下：

```

//***** TimeBook.java *****
package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
    public void doAuditing(String name) {
        //审核数据的相关程序
        .....
    }
}

```

(4) 定义 Spring 的配置文档 config.xml。config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--使用依赖注入完成变量值设定-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <!--使用 Bean 进行参考-->
        <property name="date">

```

```

        <ref bean="date"/>
    </property>
</bean>
<bean id="date" class="java.util.Date"/>

<!--以下是使用 Spring AOP 实现日志输出的 Bean-->
<bean id="log" class="com.gc.action.LogAround"/>
<bean id="timeBook" class="com.gc.action.TimeBook"/>
<!--使用 Spring 提供的 ProxyFactoryBean 来实现代理-->
<bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
    <!--指定要代理的类-->
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
</beans>

```

代码说明：

- ❑ id 为 log 的 Bean，负责输出日志信息。
- ❑ id 为 timeBook 的 Bean，负责具体的业务逻辑考勤审核。
- ❑ id 为 logProxy 的 Bean，使用 Spring 提供的 ProxyFactoryBean 来实现代理，在该 Bean 里要定义相关的属性，包括要代理的接口、目标类以及要使用的 Interceptor。

(5) 修改测试代码 TestHelloWorld，使用 Spring 提供的代理类 ProxyFactoryBean 实现日志的输出。TestHelloWorld.java 的示例代码如下：

```

//***** TestHelloWorld.java*****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取 XML
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("logProxy");
        timeBookProxy.doAuditing("张三");
    }
}

```

(6) 运行测试程序，可以得到通过 LogAround 类输出日志信息，如图 5.4 所示。

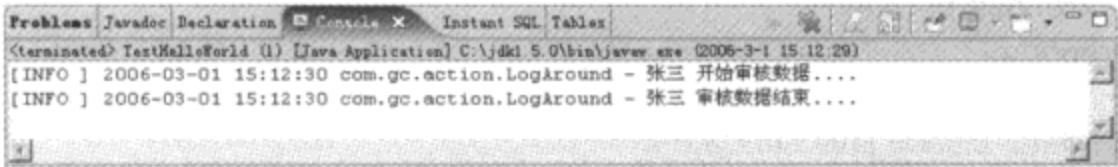


图 5.4 通过 LogAround 类输出日志信息

上面这个例子即实现了 Interception Around 通知,最终的效果和前面使用 Java 代理的效果一样,但是功能却更加强大。假如有一个新的关账程序要实现日志输出,则可以建立新的接口和其实现类,然后只要在配置文件中注册一下就可以使新的实现类也输出日志信息了。

(7) 在 com.gc.impl 包中建立新的接口 FinanceInterface。FinanceInterface.java 的示例代码如下:

```
//***** FinanceInterface.java*****  
package com.gc.impl;  
  
import org.apache.log4j.Level;  
//定义为接口主要是为了实现代理  
public interface FinanceInterface {  
    public void doCheck(String name);  
}
```

(8) 在 com.gc.action 包中新建类 Finance, doCheck()方法中编写具体的财务关账代码。Finance.java 的示例代码如下:

```
//***** Finance.java*****  
package com.gc.action;  
import com.gc.impl. FinanceInterface;  
public class Financeimplements FinanceInterface {  
    public void doCheck (String name) {  
        //关账的相关程序  
        .....  
    }  
}
```

(9) 定义 Spring 的配置文档 config.xml。config.xml 的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <!--使用依赖注入完成变量值设定-->  
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">  
        <property name="msg">  
            <value>HelloWorld</value>  
        </property>  
        <property name="date">  
            <ref bean="date"/>  
        </property>  
    </bean>
```



```

<bean id="date" class="java.util.Date"/>

<!--以下是使用 Spring AOP 实现日志输出的 Bean-->
<bean id="log" class="com.gc.action.LogAround"/>
<bean id="timeBook" class="com.gc.action.TimeBook"/>
<!--以下是考勤审核-->
<bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
<!--指定要代理的类-->
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
<!--以下是财务关账-->
<bean id="finance" class="com.gc.action.Finance"/>
<bean id="logProxy1" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.FinanceInterface</value>
    </property>
    <property name="target">
        <ref bean="finance"/>
    </property>
<!--指定要代理的类-->
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
</beans>

```

代码说明:

- ❑ id 为 finance 的 Bean, 负责具体的业务逻辑财务关账。
- ❑ id 为 logProxy1 的 Bean, 使用 Spring 提供的 ProxyFactoryBean 来实现代理, 在该 Bean 里要定义相关的属性, 包括要代理的接口、目标类以及要使用的 Interceptor。

(10) 修改测试代码 TestHelloWorld, 使用 Spring 提供的代理类 ProxyFactoryBean 实现日志的输出。TestHelloWorld.java 的示例代码如下:

```

//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;

```



```
import com.gc.impl.TimeBookInterface;
import com.gc.impl.FinanceInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取 XML
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        FinanceInterface financeProxy = (FinanceInterface)actx.getBean("logProxy1");
        financeProxy.doCheck("李四");
    }
}
```

(11) 运行测试程序，可以得到通过 LogAround 类输出日志信息，如图 5.5 所示。

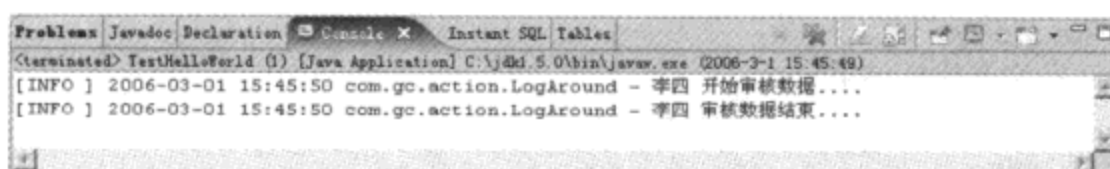


图 5.5 通过 LogAround 类输出日志信息

5.8.2 采用 Before 通知的形式实现

Before 通知只在 JoinPoint 的前面执行，实现 Before 通知的类需要实现接口 MethodBeforeAdvice。实现思路是：首先实现接口 MethodBeforeAdvice，在 before()方法里编写负责输出日志信息的代码，具体的业务逻辑还使用前面的接口 TimeBookInterface 和它的实现类 TimeBook，然后在 Spring 的配置文档中定义 Pointcut，最后测试程序不用改变，执行测试程序，查看输出结果。具体步骤如下：

(1) 编写负责输出日志信息的类 LogBefore，该类实现了接口 MethodBeforeAdvice，重写了 before()方法。LogBefore.java 的示例代码如下：

```
//***** LogBefore.java *****
package com.gc.action;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;
//实现 Before 通知的类需要实现接口 MethodBeforeAdvice
public class LogBefore implements MethodBeforeAdvice {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    // Before 通知只在 JoinPoint 的前面执行
    public void before(Method method, Object[] args, Object target) throws Throwable {
        logger.log(Level.INFO, args[0] + " 开始审核数据....");
    }
}
```

(2) 使用 com.gc.impl 包中的接口 TimeBookInterface。TimeBookInterface.java 的示例

代码如下：

```

//***** TimeBookInterface.java*****
package com.gc.impl;

import org.apache.log4j.Level;

public interface TimeBookInterface {
    public void doAuditing(String name);
}

```

(3) 使用 com.gc.action 包中的类 TimeBook，doAuditing()方法中编写具体的考勤审核代码。TimeBook.java 的示例代码如下：

```

//***** TimeBook.java*****
package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
    public void doAuditing(String name) {
        //审核数据的相关程序
        .....
    }
}

```

(4) 定义 Spring 的配置文档 config.xml。config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--使用依赖注入完成变量值设定-->
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <property name="date">
            <ref bean="date"/>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>

    <!--以下是使用 Spring AOP 实现日志输出的 Bean-->
    <bean id="log" class="com.gc.action.LogAop"/>
    <!--负责具体的业务逻辑考勤审核-->
    <bean id="timeBook" class="com.gc.action.TimeBook"/>

    <bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <property name="target">

```

```

        <ref bean="timeBook"/>
    </property>
    <!--指定要代理的类-->
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
<!--以下是实现 Before 通知-->
<bean id="logBefore" class="com.gc.action.LogBefore"/>
<bean
id="logBeforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="logBefore"/>
    </property>
    <!--代理指定类中的 doAuditing 方法-->
    <property name="patterns">
        <value>.*doAuditing.* </value>
    </property>
</bean>
<bean id="logProxy2" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
    <!--指定要代理的类-->
    <property name="interceptorNames">
        <list>
            <value>logBeforeAdvisor</value>
        </list>
    </property>
</bean>
</beans>

```

代码说明：

- ❑ id 为 timeBook 的 Bean，负责具体的业务逻辑考勤审核。
- ❑ id 为 logProxy2 的 Bean，使用 Spring 提供的 ProxyFactoryBean 来实现代理，在该 Bean 里要定义相关的属性，包括要代理的接口、目标类以及要使用的 Interceptor。
- ❑ id 为 logBeforeAdvisor 的 Bean，使用类 RegexpMethodPointcutAdvisor 来实现对切入点的配置
- ❑ 属性名为 patterns 的值，指明只对 doAuditing() 方法有效。

(5) 修改测试代码 TestHelloWorld，使用 Spring 提供的代理类 ProxyFactoryBean 实现日志的输出。TestHelloWorld.java 的示例代码如下：

```
//***** TestHelloWorld.java *****
```



```
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 来获取 XML
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("logProxy2");
        timeBookProxy.doAuditing("张三");
    }
}
```

(6) 运行测试程序，可以得到通过 LogBefore 类输出日志信息，如图 5.6 所示。

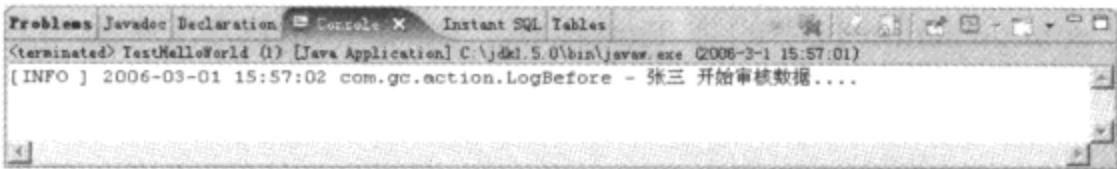


图 5.6 通过 LogBefore 类输出日志信息

上面这个例子即实现了 Before 通知。

5.8.3 采用 After Returning 通知的形式实现

After Returning 通知只在 JointPoint 的后面执行，实现 After Returning 通知的类需要实现接口 AfterReturningAdvice。其实现思路是：首先实现接口 AfterReturningAdvice，在 afterReturning()方法里编写负责输出日志信息的代码，具体的业务逻辑还使用前面的接口 TimeBookInterface 和它的实现类 TimeBook，然后在 Spring 的配置文档中定义 Pointcut，最后测试程序不用改变，执行测试程序，查看输出结果。具体步骤如下：

(1) 编写负责输出日志信息的类 LogAfter，该类实现了接口 AfterReturningAdvice，重写了 afterReturning()方法。LogAfter.java 的示例代码如下：

```
//***** LogAfter.java*****
package com.gc.action;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.aop.AfterReturningAdvice ;
// After Returning 通知只在 JointPoint 的后面执行
public class LogAfter implements AfterReturningAdvice {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //实现 After Returning 通知的类需要实现接口 AfterReturningAdvice
    public void afterReturning(Object object, Method method, Object[] args, Object target) throws
    Throwable {
        logger.log(Level.INFO, args[0] + " 审核数据完成...");
    }
}
```



```

    }
}

```

(2) 使用 com.gc.impl 包中的接口 TimeBookInterface。TimeBookInterface.java 的示例代码如下：

```

//***** TimeBookInterface.java*****
package com.gc.impl;

import org.apache.log4j.Level;
//审核数据的相关程序
public interface TimeBookInterface {
    public void doAuditing(String name);
}

```

(3) 使用 com.gc.action 包中的类 TimeBook，doAuditing()方法中编写具体的考勤审核代码。TimeBook.java 的示例代码如下：

```

//***** TimeBook.java*****
package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
    public void doAuditing(String name) {
        //审核数据的相关程序
        .....
    }
}

```

(4) 定义 Spring 的配置文档 config.xml。config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <property name="date">
            <ref bean="date"/>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>

    <!-- 以下是使用 Spring AOP 实现日志输出的 Bean-->
    <bean id="log" class="com.gc.action.LogAop"/>
    <bean id="timeBook" class="com.gc.action.TimeBook"/>

    <bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>

```

```
</property>
<property name="target">
    <ref bean="timeBook"/>
</property>
<!--指定代理类-->
<property name="interceptorNames">
    <list>
        <value>log</value>
    </list>
</property>
</bean>
<!--以下是实现 Before 通知-->
<bean id="logBefore" class="com.gc.action.LogBefore"/>
<bean
id="logBeforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="logBefore"/>
    </property>
    <!--只对 doAuditing()方法有效-->
    <property name="patterns">
        <value>.*doAuditing.* </value>
    </property>
</bean>
<bean id="logProxy2" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
    <!--指定代理类-->
    <property name="interceptorNames">
        <list>
            <value>logBeforeAdvisor</value>
        </list>
    </property>
</bean>
<!--以下是实现 After 通知-->
<bean id="logAfter" class="com.gc.action.LogAfter"/>
<bean
id="logAfterAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="logAfter"/>
    </property>
    <!--只对 doAuditing()方法有效-->
    <property name="patterns">
        <value>.*doAuditing.* </value>
    </property>
</bean>
```

```
<bean id="logProxy3" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
    <!--指定代理类-->
    <property name="interceptorNames">
        <list>
            <value>logAfterAdvisor</value>
        </list>
    </property>
</bean>
</beans>
```

代码说明：

- ❑ id 为 timeBook 的 Bean，负责具体的业务逻辑考勤审核。
- ❑ id 为 logProxy3 的 Bean，使用 Spring 提供的 ProxyFactoryBean 来实现代理，在该 Bean 里要定义相关的属性，包括要代理的接口、目标类以及要使用的 Interceptor。
- ❑ id 为 logAfterAdvisor 的 Bean，使用类 RegexpMethodPointcutAdvisor 来实现对切入点的配置。
- ❑ 属性名为 patterns 的值，指明只对 doAuditing()方法有效。

(5) 修改测试代码 TestHelloWorld，使用 Spring 提供的代理类 ProxyFactoryBean 实现日志的输出。TestHelloWorld.java 的示例代码如下：

```
//***** TestHelloWorld.java*****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("logProxy3");
        timeBookProxy.doAuditing("张三");
    }
}
```

(6) 运行测试程序，可以得到通过 LogAfter 类输出日志信息，如图 5.7 所示。

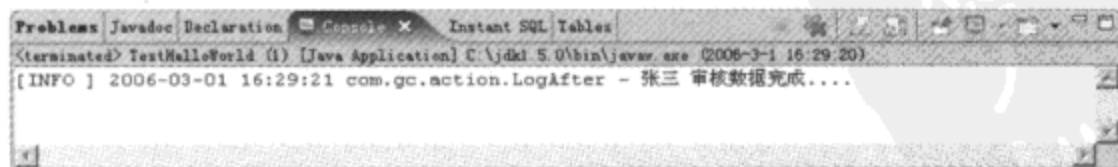


图 5.7 通过 LogAfter 类输出日志信息

5.8.4 采用 Throw 通知的形式实现

Throw 通知只在 JointPoint 抛出异常时执行，实现 Throw 通知的类需要实现接口 ThrowsAdvice。实现思路是：首先实现接口 ThrowsAdvice，在 afterThrowing()方法里编写负责输出日志信息的代码，具体的业务逻辑还使用前面的接口 TimeBookInterface 和它的实现类 TimeBook，然后在 Spring 的配置文档中定义 Pointcut，最后测试程序不用改变，执行测试程序，查看输出结果。具体步骤如下：

(1) 编写负责输出日志信息的类 LogThrow，该类实现了接口 ThrowsAdvice，重写了 afterThrowing()方法。LogThrow.java 的示例代码如下：

```
//***** LogThrow.java*****
package com.gc.action;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.aop.ThrowsAdvice;
//实现 Throw 通知的类需要实现接口 ThrowsAdvice
public class LogThrow implements ThrowsAdvice {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //在 afterThrowing()方法里编写负责输出日志信息的代码
    public void afterThrowing(Method method, Object[] args, Object target, Throwable subclass)
throws Throwable {
        logger.log(Level.INFO, args[0] + " 审核数据有异常抛出...");
    }
}
```

(2) 使用 com.gc.impl 包中的接口 TimeBookInterface。TimeBookInterface.java 的示例代码如下：

```
//***** TimeBookInterface.java*****
package com.gc.impl;

import org.apache.log4j.Level;
//针对接口编程
public interface TimeBookInterface {
    public void doAuditing(String name);
}
```

(3) 使用 com.gc.action 包中的类 TimeBook，doAuditing()方法中编写具体的考勤审核代码。TimeBook.java 的示例代码如下：

```
//***** TimeBook.java*****
package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
```



```

public void doAuditing(String name) {
    //审核数据的相关程序
    .....
}
}

```

(4) 定义 Spring 的配置文档 config.xml。config.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <property name="date">
            <ref bean="date"/>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>

    <!--以下是使用 Spring AOP 实现日志输出的 Bean-->
    <bean id="log" class="com.gc.action.LogAop"/>
    <bean id="timeBook" class="com.gc.action.TimeBook"/>

    <bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <property name="target">
            <ref bean="timeBook"/>
        </property>
        <!--指定代理类-->
        <property name="interceptorNames">
            <list>
                <value>log</value>
            </list>
        </property>
    </bean>
    <!--以下是实现 Before 通知-->
    <bean id="logBefore" class="com.gc.action.LogBefore"/>
    <bean
id="logBeforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="logBefore"/>
        </property>
        <!--只对 doAuditing 方法有效-->
        <property name="patterns">
            <value>.*doAuditing.*</value>

```

```
        </property>
    </bean>
    <bean id="logProxy2" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <property name="target">
            <ref bean="timeBook"/>
        </property>
        <!--指定代理类-->
        <property name="interceptorNames">
            <list>
                <value>logBeforeAdvisor</value>
            </list>
        </property>
    </bean>
    <!--以下是实现 After 通知-->
    <bean id="logAfter" class="com.gc.action.LogAfter"/>
    <bean
id="logAfterAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="logAfter"/>
        </property>
        <!--只对 doAuditing 方法有效-->
        <property name="patterns">
            <value>.*doAuditing.* </value>
        </property>
    </bean>
    <bean id="logProxy3" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.gc.impl.TimeBookInterface</value>
        </property>
        <property name="target">
            <ref bean="timeBook"/>
        </property>
        <!--指定代理类-->
        <property name="interceptorNames">
            <list>
                <value>logAfterAdvisor</value>
            </list>
        </property>
    </bean>
    <!--以下是实现 Throw 通知-->
    <bean id="logThrow" class="com.gc.action.LogThrow"/>
    <bean
id="logThrowAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="logThrow"/>
        </property>
```

```
<!--只对 doAuditing 方法有效-->
<property name="patterns">
    <value>.*doAuditing.* </value>
</property>
</bean>
<bean id="logProxy4" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.gc.impl.TimeBookInterface</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
    <!--指定代理类-->
    <property name="interceptorNames">
        <list>
            <value>logThrowAdvisor</value>
        </list>
    </property>
</bean>
</beans>
```

代码说明：

- ❑ id 为 timeBook 的 Bean，负责具体的业务逻辑考勤审核。
- ❑ id 为 logProxy4 的 Bean，使用 Spring 提供的 ProxyFactoryBean 来实现代理，在该 Bean 里要定义相关的属性，包括要代理的接口、目标类以及要使用的 Interceptor。
- ❑ id 为 logThrowAdvisor 的 Bean，使用类 RegexpMethodPointcutAdvisor 来实现对切入点的配置。
- ❑ 属性名为 patterns 的值，指明只对 doAuditing()方法有效。

(5) 修改测试代码 TestHelloWorld，使用 Spring 提供的代理类 ProxyFactoryBean 实现日志的输出。TestHelloWorld.java 的示例代码如下：

```
/** ***** TestHelloWorld.java ***** */
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取配置文件
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("logProxy4");
        timeBookProxy.doAuditing("张三");
    }
}
```

(6) 运行测试程序，可以没有任何日志信息输出，如图 5.8 所示。从输出结果可以看到，没有任何日志信息输出。这是因为在 doAuditing()方法里没有异常抛出，现在在 doAuditing()方法里增加一个异常抛出。

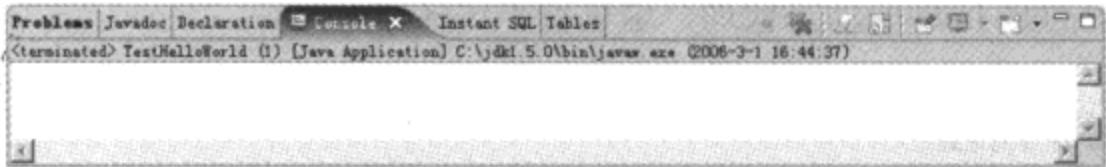


图 5.8 没有任何日志信息输出

(7) 在 TimeBook 类的 doAuditing()方法里增加一个异常。TimeBook.java 的示例代码如下：

```

//***** TimeBook.java *****
package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
    public void doAuditing(String name) {
        //审核数据的相关程序
        .....
        int m = 1/0;
    }
}

```

(8) 运行测试程序，可以看到通过 LogThrow 类输出日志信息，如图 5.9 所示。



图 5.9 通过 LogThrow 类输出日志信息

上面分别通过 Spring 提供的不同通知类型，实现了日志输出的实例，分别展示它们的编写和配置方式，目的还是让读者能很快地理解 Spring 的通知类型。

5.9 Spring 中 AOP 的两种代理方式

在本章刚开始时，首先从 Java 的动态代理入手，引出 Spring 的 AOP 代理，而且在前面的示例中，使用的就是 Spring 支持的 Java 动态代理，其实 Spring 也支持 CGLIB 代理。

5.9.1 Java 动态代理

在前面的示例中，使用的就是 Spring 支持的 Java 动态代理，也就是说代理的是接口，

Spring 默认使用的是 Java 的动态代理，这里就不再多作介绍。

5.9.2 CGLIB 代理

正如前面所说，Spring 也提供了对 CGLIB 代理的支持，主要改变就是设定 ProxyFactoryBean 的 proxyTargetClass 属性，将该属性值设定为 true 即可。因为要使用 CGLIB 代理，所以要将 cglib-nodep-2.1_3.jar 加入到 CLASSPATH 中，下面的示例中将使用前面的 Around 通知来实现。具体编写步骤如下：

(1) 使用前面负责输出日志信息的类 LogAround，该类实现了接口 MethodInterceptor，重写了 invoke() 方法。LogAround.java 的示例代码如下：

```
//***** LogAround.java*****
package com.gc.action;

import org.aopalliance.intercept.MethodInvocation;
import org.aopalliance.intercept.MethodInterceptor;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
//该类实现了接口 MethodInterceptor
public class LogAround implements MethodInterceptor {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //负责输出日志信息的代码
    public Object invoke(MethodInvocation mi) throws Throwable {
        logger.log(Level.INFO, mi.getArguments()[0] + " 开始审核数据...");
        try {
            Object result = mi.proceed();
            return result;
        }
        finally {
            logger.log(Level.INFO, mi.getArguments()[0] + " 审核数据结束...");
        }
    }
}
```

(2) 假如把 Spring 的配置文件做一下修改，把 id 为 logProxy 的 Bean 增加一个属性 proxyTargetClass，并设定该属性值为 true。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <property name="date">
            <ref bean="date"/>
        </property>
    </bean>

```

```

</bean>
<bean id="date" class="java.util.Date"/>

<!--以下是使用 Spring AOP 实现日志输出的 Bean-->
<bean id="log" class="com.gc.action.LogAop"/>
<bean id="timeBook" class="com.gc.action.TimeBook"/>

<bean id="logProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyTargetClass">
        <value>true</value>
    </property>
    <property name="target">
        <ref bean="timeBook"/>
    </property>
    <!--指定代理类-->
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
</beans>

```

(3) 修改测试代码 TestHelloWorld, 使用 Spring 提供的代理类 ProxyFactoryBean 实现日志的输出。TestHelloWorld.java 的示例代码如下:

```

//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("logProxy");
        timeBookProxy.doAuditing("张三");
    }
}

```

(4) 运行测试程序, 可以看到有异常信息输出, 如图 5.10 所示。

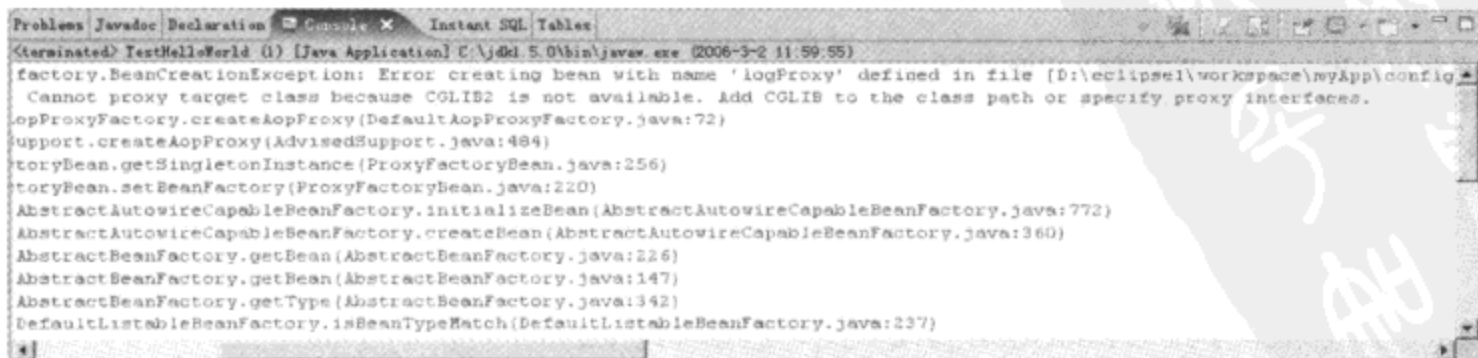


图 5.10 异常信息输出

这是因为在 Spring 的配置文档中指定了要用 CGLIB 方式实现代理，但却没有把 cglib-nodep-2.1_3.jar 加入到 CLASSPATH 中去。按照第 2 章中介绍的方法，把 spring-framework-2.0-m1/lib/cglib 目录下的 cglib-nodep-2.1_3.jar 加入到 CLASSPATH 中。

(5) 再次运行测试程序，可以看到使用 CGLIB 代理输出的日志信息，如图 5.11 所示。

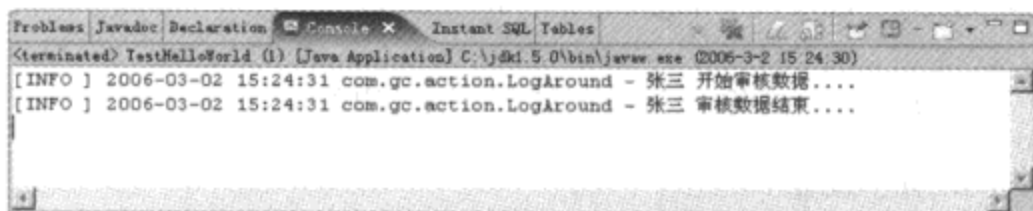


图 5.11 使用 CGLIB 代理输出的日志信息

使用 CGLIB 代理的好处就是不用再像使用 Java 动态代理那样去实现特定的接口，一个普通的 Java 类就可以了。

5.10 Spring 中的自动代理

不管是使用 Java 的动态代理还是使用 CGLIB 代理，虽然功能很强大，但是对于每一个类，都要在 Spring 的配置文档中建立相应的代理，如果只是一个很小的应用系统，还看不出来工作量有多大，但对于一个大型的企业应用来说，工作量就太大了，而且重复性的工作很多，幸好 Spring 提供了一种自动代理的方式，可以减轻这部分工作。要使用 Spring 中的动态代理，org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator 包是必需的。下面就讲解一下怎么来实现自动代理。

这个例子是所有类中以 do 开头的方法，在被调用时，都要进行日志输出。实现思路是：首先在接口 TimeBookInterface 中增加一个方法 doCheck()，在其实现类 TimeBook 中实现这个方法，接着使用前面示例中的 Before、After 通知，然后修改配置文档，定义自动代理，最后编写测试程序，查看输出结果。具体步骤如下所示：

(1) 在 com.gc.impl 包的接口 TimeBookInterface 中增加一个方法 doCheck()。TimeBookInterface.java 的示例代码如下：

```
//***** TimeBookInterface.java*****  
package com.gc.impl;  
  
import org.apache.log4j.Level;  
  
public interface TimeBookInterface {  
    public void doAuditing(String name);  
    public void doCheck(String name);  
}
```

(2) 在 com.gc.action 包的类 TimeBook 中增加一个方法 doCheck()，doCheck()方法中编写财务关账的代码。TimeBook.java 的示例代码如下：

```
//***** TimeBook.java*****
```

```

package com.gc.action;
import com.gc.impl.TimeBookInterface;
public class TimeBook implements TimeBookInterface {
    public void doAuditing(String name) {
        //审核数据的相关程序
        System.out.println ("审核数据的相关程序正在执行...");
    }
    public void doCheck(String name) {
        //关账的相关程序
        System.out.println ("财务关账的相关程序正在执行...");
    }
}

```

(3) 使用前面示例中负责输出日志信息的类 LogBefore, 该类实现了接口 MethodBeforeAdvice, 重写了 before()方法。LogBefore.java 的示例代码如下:

```

//***** LogBefore.java*****
package com.gc.action;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class LogBefore implements MethodBeforeAdvice {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //用于在执行审核程序前调用该方法
    public void before(Method method, Object[] args, Object target) throws Throwable {
        logger.log(Level.INFO, args[0] + " 开始审核数据...");
    }
}

```

(4) 使用前面示例中负责输出日志信息的类 LogAfter, 该类实现了接口 AfterReturningAdvice, 重写了 afterReturning()方法。LogAfter.java 的示例代码如下:

```

//***** LogAfter.java*****
package com.gc.action;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.aop.AfterReturningAdvice;

public class LogAfter implements AfterReturningAdvice {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //用于在执行审核程序后调用该方法
    public void afterReturning(Object object, Method method, Object[] args, Object target) throws
    Throwable {

```



```
        logger.log(Level.INFO, args[0] + " 审核数据完成...");
    }
}
```

(5) 修改配置文件 config.xml，增加一个负责自动代理的 Bean。config.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
        <property name="msg">
            <value>HelloWorld</value>
        </property>
        <property name="date">
            <ref bean="date"/>
        </property>
    </bean>
    <bean id="date" class="java.util.Date"/>

    <!--以下是使用 Spring AOP 实现日志输出的 Bean-->
    <bean id="log" class="com.gc.action.LogAop"/>
    <bean id="timeBook" class="com.gc.action.TimeBook"/>

    <bean id="autoProxyCreator"
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

    <!--以下是实现 Before 通知-->
    <bean id="logBefore" class="com.gc.action.LogBefore"/>
    <bean id="logBeforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="logBefore"/>
        </property>
        <!--指定以 do 开头的方法-->
        <property name="patterns">
            <value>.*do.* </value>
        </property>
    </bean>

    <!--以下是实现 After 通知-->
    <bean id="logAfter" class="com.gc.action.LogAfter"/>
    <bean id="logAfterAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="logAfter"/>
        </property>
        <!--指定以 do 开头的方法-->
        <property name="patterns">
```

```
        <value>.*do.* </value>
    </property>
</bean>
</beans>
```

(6) 修改测试代码 TestHelloWorld, 调用 doCheck()方法。TestHelloWorld.java 的示例代码如下:

```
//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("timeBook");
        timeBookProxy.doCheck("张三");
    }
}
```

(7) 运行测试程序, 可以看到调用 doCheck()方法和 LogBefore、LogAfter 通知的日志信息输出, 如图 5.12 所示。

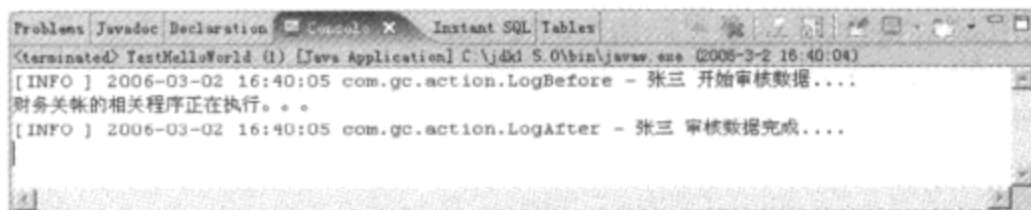


图 5.12 调用 doCheck()方法和 LogBefore、LogAfter 通知的日志信息输出

(8) 修改测试代码 TestHelloWorld, 调用 doAuditing()方法。TestHelloWorld.java 的示例代码如下:

```
//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[ ] args) {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("timeBook");
        timeBookProxy.doAuditing("张三");
    }
}
```

(9) 运行测试程序, 可以看到调用 doAuditing()方法和 LogBefore、LogAfter 通知的日志信息输出, 如图 5.13 所示。

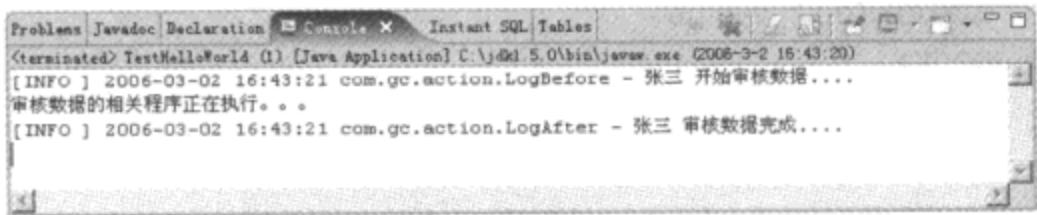


图 5.13 调用 doAuditing()方法和 LogBefore、LogAfter 通知的日志信息输出

(10) 修改配置文档 config.xml，把 LogBefore 通知定义为指定 doCheck()方法，把 LogAfter 通知定义为指定 doAuditing()方法。config.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="HelloWorld" class="com.gc.action.HelloWorld" depends-on="date">
    <property name="msg">
      <value>HelloWorld</value>
    </property>
    <property name="date">
      <ref bean="date"/>
    </property>
  </bean>
  <bean id="date" class="java.util.Date"/>

  <!--以下是使用 Spring AOP 实现日志输出的 Bean-->
  <bean id="log" class="com.gc.action.LogAop"/>
  <bean id="timeBook" class="com.gc.action.TimeBook"/>

  <bean
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
    id="autoProxyCreator"/>

  <!--以下是实现 Before 通知-->
  <bean id="logBefore" class="com.gc.action.LogBefore"/>
  <bean
    id="logBeforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref bean="logBefore"/>
    </property>
    <!--指定对 doCheck 方法有效-->
    <property name="patterns">
      <value>.*doCheck.*</value>
    </property>
  </bean>

  <!--以下是实现 After 通知-->
  <bean id="logAfter" class="com.gc.action.LogAfter"/>
  <bean
    id="logAfterAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref bean="logAfter"/>
    </property>
  </bean>
</beans>
```



```
</property>
<!--指定对 doAuditing 方法有效-->
<property name="patterns">
    <value>.*doAuditing.* </value>
</property>
</bean>
</beans>
```

(11) 修改测试代码 TestHelloWorld, 调用 doCheck()方法。TestHelloWorld.java 的示例代码如下:

```
//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("timeBook");
        timeBookProxy.doCheck("张三");
    }
}
```

(12) 运行测试程序, 可以看到调用 doCheck()方法和 LogBefore 通知的日志信息输出, 如图 5.14 所示。

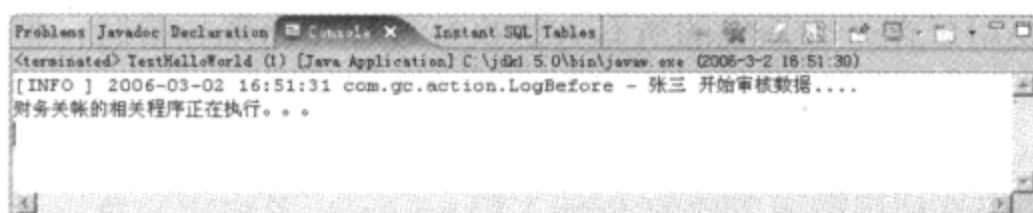


图 5.14 调用 doCheck()方法和 LogBefore 通知的日志信息输出

(13) 修改测试代码 TestHelloWorld, 调用 doAuditing()方法。TestHelloWorld.java 的示例代码如下:

```
//***** TestHelloWorld.java *****
package com.gc.test;
import com.gc.action.TimeBook;
import com.gc.action.TimeBookProxy;
import com.gc.impl.TimeBookInterface;
public class TestHelloWorld {
    public static void main(String[] args) {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("config.xml");
        TimeBookInterface timeBookProxy = (TimeBookInterface)actx.getBean("timeBook");
        timeBookProxy.doAuditing("张三");
    }
}
```


(14) 运行测试程序, 可以看到调用 `doAuditing()` 方法和 `LogAfter` 通知的日志信息输出, 如图 5.15 所示。

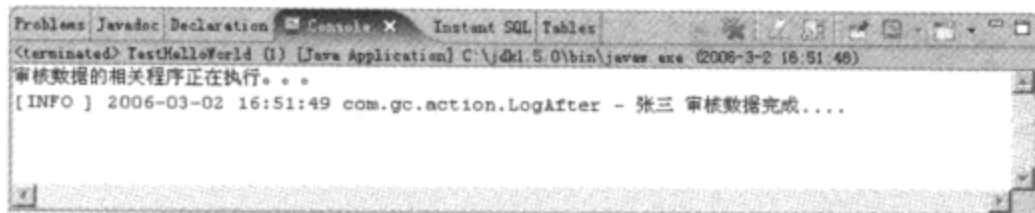


图 5.15 调用 `doAuditing()` 方法和 `LogAfter` 通知的日志信息输出

上面的示例展示了 Spring 的自动代理, 并接着展示了利用正则表达式进行方法的匹配, 由此可以看出使用自动代理的简便。

5.11 一个用 Spring AOP 实现异常处理和记录程序执行时间的实例

虽然前面也给出了 Spring AOP 的一些实例, 但因为主要目的是为了介绍 Spring 的知识点, 不一定会很完整, 下面笔者就通过一个完整的用 Spring AOP 实现异常处理和记录程序执行时间的实例来展示使用 Spring AOP 的整个过程。

5.11.1 异常处理和记录程序执行时间的实例简介

这个实例主要用于在一个系统的所有方法执行过程中出现异常时, 把异常信息都记录下来, 另外记录每个方法的执行时间。用两个业务逻辑来说明上述功能, 这两个业务逻辑首先使用 Spring AOP 的自动代理功能, 然后一个用 Java 的动态代理, 一个用 CGLIB 代理。

实现思路是: 仍然使用前面所建的 Java 工程 `myApp`, 首先定义负责异常处理的 Advice 为 `ExceptionHandler.java`, 定义记录程序执行时间的 Advice 为 `TimeHandler.java`, 接着定义业务逻辑接口 `LogicInterface.java`, 编写实现业务逻辑接口的类 `Logic1.java`, 该业务逻辑在 Spring AOP 中使用 Java 的动态代理, 编写另一个业务逻辑 `Logic2.java` 不实现业务逻辑接口, 该业务逻辑在 Spring AOP 中使用 CGLIB 代理, 然后使用自动代理定义配置文件 `config.xml`, 最后编写测试程序 `TestAop.java`, 执行它并查看输出结果。下面就一步一步来实现这个实例。

5.11.2 定义负责异常处理的 Advice 为 `ExceptionHandler.java`

在 `myApp` 工程的 `com.gc.action` 包中新建 `ExceptionHandler.java`, 该类主要负责当程序执行过程中出现异常时, 把异常信息都记录下来。而 Spring 提供的通知类型中 `Throw` 通知可以实现这个功能, 因此这里使用 `Throw` 通知类型来实现 Advice, 类 `ExceptionHandler` 必须实现 `ThrowsAdvice` 接口, 重写 `afterThrowing()` 方法。`ThrowsAdvice.java` 的示例代码如下:

```

//***** ExceptionHandler.java*****
package com.gc.action;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.aop.ThrowsAdvice;
//使用 Throw 通知类型来实现 Advice
public class ExceptionHandler implements ThrowsAdvice{
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //重写 afterThrowing()方法
    public void afterThrowing(Method method, Object[] args, Object target, Throwable subclass)
throws Throwable {
        logger.log(Level.INFO, args[0] + " 执行 " + method.getName() + " 时有异常抛出...." +
subclass);
    }
}

```

代码说明：

- ☐ 必须重写 afterThrowing()方法。
- ☐ 当程序有异常发生时，就会输出“×××执行×××方法时有异常抛出”和具体异常的记录信息。

5.11.3 定义记录程序执行时间的 Advice 为 TimeHandler.java

在 myApp 工程的 com.gc.action 包中新建 TimeHandler.java，该类主要负责记录每个方法的执行时间，而 Spring 提供的通知类型中 Around 通知可以实现这个功能，因此这里使用 Around 通知类型来实现 Advice，类 TimeHandler 必须实现 MethodInterceptor 接口，重写 invoke()方法。TimeHandler.java 的示例代码如下：

```

//***** TimeHandler.java*****
package com.gc.action;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
//类 TimeHandler 必须实现 MethodInterceptor 接口
public class TimeHandler implements MethodInterceptor{
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //重写 invoke()方法
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        long procTime = System.currentTimeMillis();
        logger.log(Level.INFO, methodInvocation.getArguments()[0] + " 开始执行 " +
methodInvocation.getMethod() + " 方法");
        try {

```

```
        Object result = methodInvocation.proceed();
        return result;
    }
    finally {
        //计算执行时间
        procTime = System.currentTimeMillis() - procTime;
        logger.log(Level.INFO, methodInvocation.getArguments()[0] + " 执行 " +
methodInvocation.getMethod() + " 方法结束");
        logger.log(Level.INFO, "执行 " + methodInvocation.getMethod().getName() + " 方
法共用了 " + procTime + "毫秒");
    }
}
```

代码说明：

- ❑ 必须重写 `invoke()` 方法。
- ❑ 当有方法执行时，就会输出“×××在什么时间开始执行×××方法”、“×××在什么时间执行×××方法结束”和“执行×××方法共用了×××毫秒”的记录信息。

5.11.4 定义业务逻辑接口 `LogicInterface.java`

在 `myApp` 工程的 `com.gc.impl` 包中新建接口 `LogicInterface.java`，该接口主要用来实现使用 Spring AOP 的动态代理机制，在这个接口里共定义了 3 个方法——新增、修改和删除。`LogicInterface.java` 的示例代码如下：

```
/** ***** LogicInterface.java ***** */
package com.gc.impl;
//该接口主要用来实现使用 Spring AOP 的动态代理机制
public interface LogicInterface {
    public void doInsert(String name);
    public void doUpdate(String name);
    public void doDelete(String name);
}
```

5.11.5 编写实现业务逻辑接口的类 `Logic1.java`

在 `myApp` 工程的 `com.gc.action` 包中新建业务逻辑类 `Logic1.java`，该类主要负责具体的业务逻辑，这个类实现了前面定义的接口 `LogicInterface`，并重写了接口 `LogicInterface` 定义的 3 个方法——新增、修改和删除，用来实现使用 Spring AOP 的动态代理机制，并在删除方法里增加一个“`i = i / 0`”，用来模拟异常的发生。`Logic1.java` 的示例代码如下：

```
/** ***** Logic1.java ***** */
package com.gc.action;

import com.gc.impl.LogicInterface;
```

```
//实现这个接口
public class Logic1 implements LogicInterface{
    //负责新增
    public void doInsert(String name){
        System.out.println("执行具体负责新增的业务逻辑...");
        for (int i = 0; i < 1000000000; i++) {
            //模拟执行时间
        }
    }
    //负责修改
    public void doUpdate(String name){
        System.out.println("执行具体负责修改的业务逻辑...");
        for (int i = 0; i < 2000000000; i++) {
            //模拟执行时间
        }
    }
    //负责删除
    public void doDelete(String name){
        System.out.println("执行具体负责删除的业务逻辑...");
        for (int i = 0; i < 3000000000; i++) {
            i = i / 0; //模拟异常发生
        }
    }
}
```

5.11.6 编写一个不实现业务逻辑接口的类 Logic2.java

在 myApp 工程的 com.gc.action 包中新建业务逻辑类 Logic2.java，该类主要负责具体的业务逻辑——新增、修改和删除，这个类不实现前面定义的接口 LogicInterface，用来实现使用 Spring AOP 的 CGLIB 代理机制，并在删除方法里增加一个“ $i = i / 0$ ”，用来模拟异常的发生。Logic2.java 的示例代码如下：

```
/****** Logic2.java *****
package com.gc.action;
//该类采用 CGLIB 代理机制
public class Logic2 {
    //负责新增
    public void doInsert(String name){
        System.out.println("执行具体负责新增的业务逻辑...");
        for (int i = 0; i < 1000000000; i++) {
            //模拟执行时间
        }
    }
    //负责修改
    public void doUpdate(String name){
        System.out.println("执行具体负责修改的业务逻辑...");
        for (int i = 0; i < 2000000000; i++) {
            //模拟执行时间
        }
    }
}
```



```

    }
}
//负责删除
public void doDelete(String name){
    System.out.println("执行具体负责删除的业务逻辑...");
    for (int i = 0; i < 300000000; i++) {
        i = i / 0; //模拟异常发生
    }
}
}
}

```

5.11.7 使用自动代理定义配置文件 config.xml

在 myApp 根目录下，新建一个 Spring 的配置文件 exception_config.xml，主要用来使用 Spring 的自动代理功能，代理本系统所有的程序。exception_config.xml 的示例代码如下：

```

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

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="logic1" class="com.gc.action.Logic1"/>
    <bean id="logic2" class="com.gc.action.Logic2"/>
    <!-- 设定为自动代理 -->
    <bean id="autoProxyCreator"
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
    <!-- 负责记录有异常发生时的信息 -->
    <bean id="exceptionHandler" class="com.gc.action.ExceptionHandler"/>

    <bean id="exceptionHandlerAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="exceptionHandler"/>
        </property>
        <!-- 对指定类的任何方法有效 -->
        <property name="patterns">
            <value>.*</value>
        </property>
    </bean>

    <!-- 负责记录方法的记录时间 -->
    <bean id="timeHandler" class="com.gc.action.TimeHandler"/>
    <bean id="timeHandlerAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="timeHandler"/>
        </property>
        <!-- 对指定类的任何方法有效 -->
        <property name="patterns">
            <value>.*</value>
        </property>
    </bean>

```

```
</property>
</bean>
</beans>
```

5.11.8 编写测试类 Logic1 的程序 TestAop.java

在 myApp 工程的 com.gc.test 包中新建测试类 TestAop.java, 该类主要负责对前面程序的测试和演示, 这里先编写对业务逻辑 Logic1.java 的演示。TestAop.java 的示例代码如下:

```
//***** TestAop.java *****
package com.gc.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.impl.LogicInterface;

public class TestAop {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("exception_config.xml");
        LogicInterface logic = (LogicInterface)actx.getBean("logic1");
        //模拟执行新增、修改、删除方法
        try {
            logic.doInsert("张三");
            logic.doUpdate("李四");
            logic.doDelete("王五");
        } catch (Exception ex) {
        }
    }
}
```

5.11.9 输出自动代理时类 Logic1 异常处理和记录程序执行时间的信息

运行测试程序, 即可看到记录 Logic1.java 中每个方法执行时间和在删除方法里捕获的异常的信息, 如图 5.16 所示。

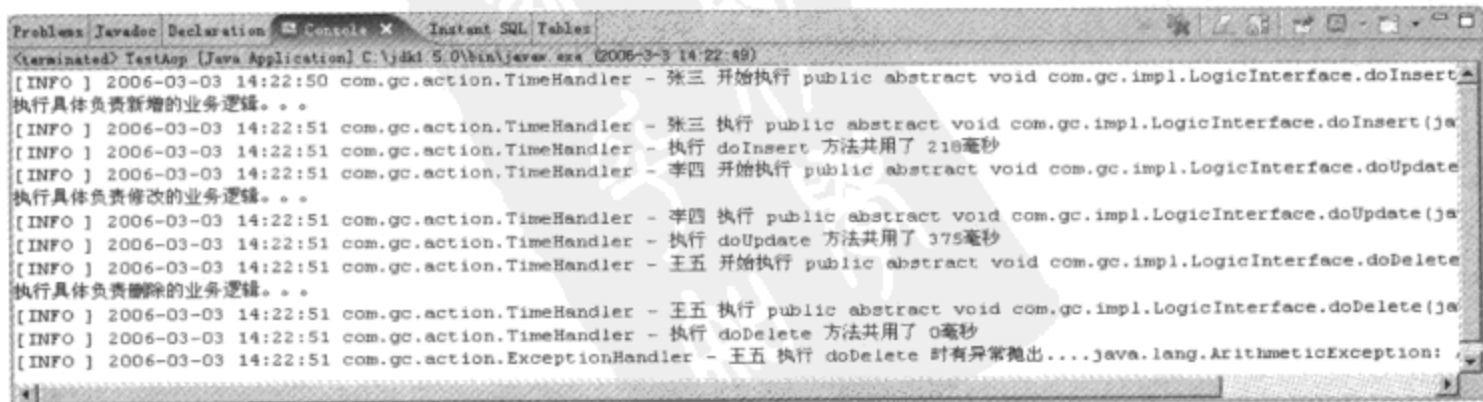


图 5.16 记录 Logic1.java 中每个方法执行时间和在删除方法里捕获的异常信息

5.11.10 编写测试类 Logic2 的程序 TestAop.java

上面主要是针对业务逻辑 Logic1.java 的测试，所以在记录的信息中，类名显示的是 Logic1 实现的接口 LogicInterface 的名称，对于业务逻辑 Logic2.java 的测试，需要修改测试代码 TestAop.java。TestAop.java 的示例代码如下：

```
//***** TestAop.java *****
package com.gc.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.action.Logic2;

public class TestAop {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("exception_config.xml");
        Logic2 logic2 = (Logic2)actx.getBean("logic2");
        //模拟执行新增、修改和删除方法
        try {
            logic2.doInsert("张三");
            logic2.doUpdate("李四");
            logic2.doDelete("王五");
        } catch (Exception ex) {

        }
    }
}
```

5.11.11 输出自动代理时类 Logic2 异常处理和记录程序执行时间的信息

运行测试程序，即可看到记录 Logic2.java 中每个方法执行时间和在删除方法里捕获的异常的信息，如图 5.17 所示。

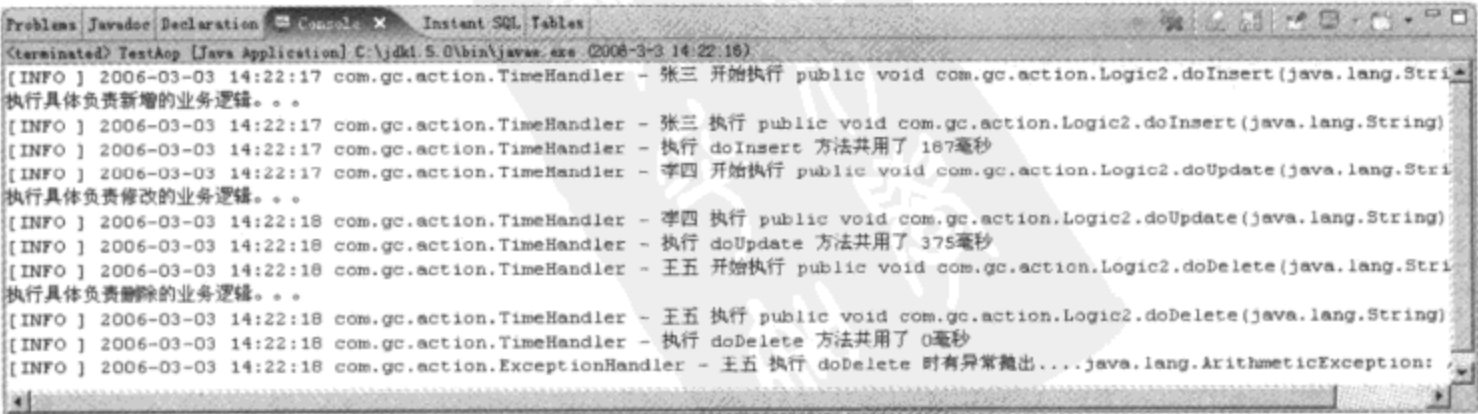


图 5.17 记录 Logic2.java 中每个方法执行时间和在删除方法里捕获的异常信息

上面主要是针对业务逻辑 Logic2.java 的测试，所以在记录的信息中，类名显示的是 Logic2 的名称。这是使用接口和不使用接口时显示信息的主要区别。

以上示例使用的是 Spring AOP 的自动代理，那对于使用 Spring AOP 的动态代理和 CGLIB 代理来说，应该怎么实现呢？下面就来讲解使用 ProxyFactoryBean 实现动态代理和 CGLIB 代理。

5.11.12 使用 ProxyFactoryBean 代理定义配置文件 config.xml

改写 Spring 的配置文件 exception_config.xml，主要用来使用 Spring 的 ProxyFactoryBean 代理功能，对 Logic1.java 只记录程序执行时间，不捕获发生的异常，对 Logic2.java 只捕获发生的异常，不记录程序执行时间，去掉前面的自动代理。exception_config.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="logic1" class="com.gc.action.Logic1"/>
    <bean id="logic2" class="com.gc.action.Logic2"/>
    <!--设定为自动代理-->
    <!--负责记录有异常发生时的信息-->
    <bean id="exceptionHandler" class="com.gc.action.ExceptionHandler"/>

    <bean id="exceptionHandlerAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="exceptionHandler"/>
        </property>
        <!--对指定类的任何方法有效-->
        <property name="patterns">
            <value>.*</value>
        </property>
    </bean>
    <!--负责记录方法的记录时间-->
    <bean id="timeHandler" class="com.gc.action.TimeHandler"/>
    <bean id="timeHandlerAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref bean="timeHandler"/>
        </property>
        <!--对指定类的任何方法有效-->
        <property name="patterns">
            <value>.*</value>
        </property>
    </bean>
    <bean id="logic1Proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
```



```

        <property name="proxyInterfaces">
            <value>com.gc.impl.LogicInterface</value>
        </property>
        <property name="target">
            <ref bean="logic1"/>
        </property>
        <!--指定代理类-->
        <property name="interceptorNames">
            <list>
                <value>timeHandlerAdvisor</value>
            </list>
        </property>
    </bean>
    <bean id="logic2Proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyTargetClass">
            <value>true</value>
        </property>
        <property name="target">
            <ref bean="logic2"/>
        </property>
        <!--指定代理类-->
        <property name="interceptorNames">
            <list>
                <value>exceptionHandler</value>
            </list>
        </property>
    </bean>
</beans>

```

5.11.13 编写测试类 Logic1 的程序 TestAop.java

这里先改写对业务逻辑 Logic1.java 的演示。TestAop.java 的示例代码如下：

```

//***** TestAop.java *****
package com.gc.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.impl.LogicInterface;

public class TestAop {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
        ClassNotFoundException {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("exception_config.xml");
        LogicInterface logic = (LogicInterface)actx.getBean("logic1Proxy");
        //模拟执行新增、修改和删除方法
        try {

```

```
        logic.doInsert("张三");
        logic.doUpdate("李四");
        logic.doDelete("王五");
    } catch (Exception ex) {

    }

}

}
```

5.11.14 输出 ProxyFactoryBean 代理时类 Logic1 记录程序执行时间的信息

运行测试程序，即可看到记录 Logic1.java 中每个方法执行时间的信息，如图 5.18 所示。

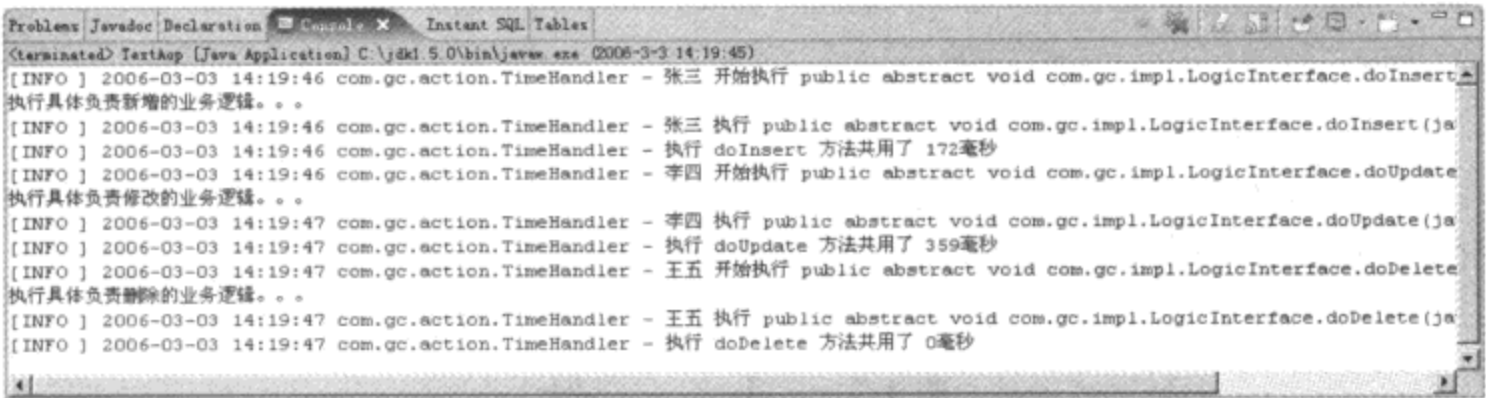


图 5.18 记录 Logic1.java 中每个方法执行时间的信息

5.11.15 编写测试类 Logic2 的程序 TestAop.java

上面主要是针对业务逻辑 Logic1.java 的测试，所以在记录的信息中，只显示了每个方法的执行时间，而没有捕获删除方法抛出的异常信息，对于业务逻辑 Logic2.java 的测试，使其只捕获删除方法抛出的信息，而不记录每个方法执行的时间，需要修改测试代码 TestAop.java。TestAop.java 的示例代码如下：

```
//***** TestAop.java *****
package com.gc.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

import com.gc.action.Logic2;

public class TestAop {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
    ClassNotFoundException {
        //通过 ApplicationContext 获取配置文档
        ApplicationContext actx=new FileSystemXmlApplicationContext("exception_config.xml");
        Logic2 logic2 = (Logic2)actx.getBean("logic2Proxy ");
        //模拟执行新增、修改和删除方法
    }
}
```

```
        logic2.doInsert("张三");
        logic2.doUpdate("李四");
        logic2.doDelete("王五");
    }
}
```

5.11.16 输出 ProxyFactoryBean 代理时类 Logic2 异常处理的信息

运行测试程序,即可看到记录 Logic2.java 中在删除方法里捕获的异常的信息,如图 5.19 所示。

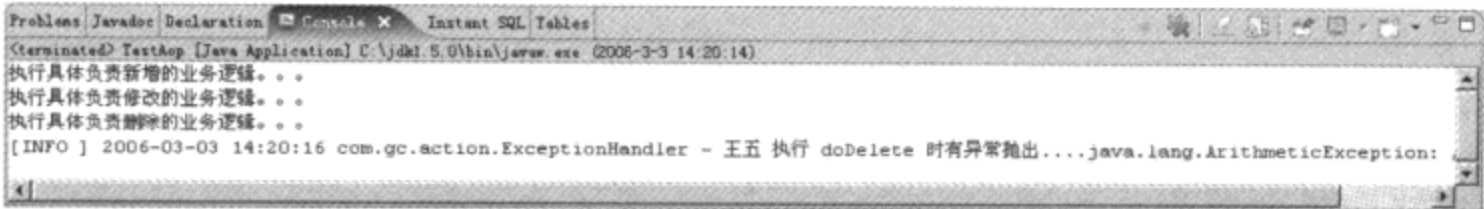


图 5.19 记录 Logic2.java 中在删除方法里捕获的异常信息

上面主要是针对业务逻辑 Logic2.java 的测试,所以在记录的信息中,只捕获删除方法抛出的异常信息,而不记录每个方法执行的时间。

5.12 小 结

本章首先从 AOP 的基本思想讲起,接着对 Java 的动态代理机制进行了讲解,然后讲解了 AOP 的 3 个重要概念,又使用不同的通知分别实现了日志输出的示例,最后通过一个完整的实例使读者更全面地了解了 Spring 中的 AOP。

因为 AOP 在技术和思想上都还没有完全成熟,如果大量地应用 AOP 可能会有一定的负面作用,可能会达不到开发者的初衷,使得代码更难以理解。因此,要适度地使用 AOP。

总的来说,AOP 编程的实质就是:业务逻辑不再实现横切关注点,而是由单独的类来封装。当业务逻辑需要用到封装的横切关注点时,AOP 会自动把封装的横切关注点插入到具体的业务逻辑中。



第 6 章 Spring 的事务处理

从第 5 章中读者可以了解到，使用 Spring 提供的 AOP，就可以实现事务处理，本章主要讲述 Spring 事务处理的基本概念和使用方法。首先对传统的事务处理作一介绍，然后再介绍使用 Spring 怎样来进行事务处理。Spring 提供了声明式和编程式事务处理。

6.1 简述事务处理

事务处理在应用程序开发中起着至关重要的作用。首先来看一下事务处理的基本概念，假如读者已经了解了事务的基本概念，则可以跳过这一章。

6.1.1 事务处理的基本概念

事务处理由若干个步骤组成，这些步骤之间具有一定的逻辑关系，作为一个整体的操作过程，所有步骤必须同时操作成功或者失败。当所有的步骤都操作成功时，事务就算操作成功了，而当其中某一个步骤操作失败的时候，则该步骤之前的操作就必须撤销。简单来说，所谓事务，就是一系列必须都成功的操作，只要有一步操作失败，所有其他的步骤将都要撤销。这里有两个概念要说明一下。

(1) 提交 (Commit)：当所有的操作步骤都被完整执行后，称该事务被提交。

(2) 回滚 (RollBack)：由于某一操作步骤执行失败，导致所有步骤都没有被提交，则事务必须回滚，即回到事务执行前的状态。

举例来说明：在一个银行系统中，假如一个数据库表用来存放客户的存款金额，一个数据库表用来存放客户的存取款历史。客户进行取钱操作的过程就是一个典型的事务。当客户进行取钱操作时，一方面要改变客户的存款金额，一方面要增加客户的取款历史记录，如果改变客户存款金额的操作成功之后，因为种种原因，比如网络故障导致增加客户取款历史记录的操作不成功，这时就必须将改变客户存款金额的操作撤销掉，使其恢复至更改前的状态，否则，如果改变客户存款金额的操作成功，而没有增加客户的取款历史记录，就会造成数据库中记录的不一致。很明显，这种情况是不允许发生的。在这种情况下，事务就必须回滚到这一事务执行前的状态。

6.1.2 事务处理的特性

每个事务都有一些它们所共有的特性，叫做 ACID 特性，分别是原子性 (Atomicity)、

一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。下面分别对这4种特性进行讲解。

1. 原子性

事务的原子性表示事务执行过程中，把事务作为一个工作单元处理，一个工作单元可能包括若干个操作步骤，每个操作步骤都必须完成才算完成，若因任何原因导致其中的一个步骤操作失败，则所有步骤都操作失败，前面已经完成的步骤也必须回滚，系统将返回到事务开始前的状态。

在前面银行取钱的例子中，如果改变客户的存款金额和增加客户的取款历史记录两个步骤中有任何一个不成功，则整个操作步骤都不成功，不可能出现客户的存款金额改变了，但客户的取款历史记录却没有增加，如果出现这种情况，后果也是可以想象的。

2. 一致性

事务的一致性保证数据处于一致状态。如果事务开始时系统处于一致状态，则事务结束时系统也应处于一致状态，不管事务成功还是失败。当然一致性不只是事务本身的问题，它也是由同一组业务规则或完整性限制确定的，因为如果业务规则不保证数据处于一致状态，只是事务本身也是无能为力的，所以为了保证一致性，需要事务本身和开发人员的共同合作。

在前面银行取钱的例子中，如果改变客户的存款金额和增加客户的取款历史记录两个步骤中有任何一个不成功，则系统必须返回到客户取钱之前的那个状态，也就是客户的存款金额和客户的取款历史记录必须是同步的。

3. 隔离性

事务的隔离性保证事务访问的任何数据不会受其他事务所做的任何改变的影响，直到该事务完成。可以想象，假如两个事务同时访问同一个数据，而且都执行修改数据的操作，那是多么的可怕。

在前面银行取钱的例子中，当一个取钱的事务正在进行时，不允许其他的事务再对它进行操作。这是因为，假如一个事务正在进行取钱的动作，另外一个事务也在进行取钱的动作，则造成的结果是可想而知的。

4. 持久性

事务的持久性保证假如事务执行成功，则它在系统中产生的结果应该是持久的。事务提交是数据源和应用程序之间的一个协议，而事务日志就是这个协议的书面记录。更为重要的是，改变本身不是持久的，另一事务可能在此后改变数据。事务日志能提供可追查的依据。

在前面银行取钱的例子中，取钱后扣除金额的动作和增加客户取款历史记录的动作是持久的，一直保持在系统中。假如是因为业务逻辑的错误导致事务需要恢复到以前的状态，则可以使用数据库的事务日志。

6.1.3 对事务处理特性的总结

所有这些事务特性，不管其内部如何关联，仅仅是保证从事务开始到事务完成，不管

事务成功与否，都能正确地管理事务涉及的数据。

事务处理应该保证事务的原子性、隔离性和持久性，开发人员应该保证事务的一致性，保证业务规则、指定主键、设定关联和其他一些规则。当提交事务的时候，数据库根据这些设定验证其数据的一致性。如果发现事务结果与系统确定的规则一致，则事务提交；如果结果不符合要求，则事务撤销。

事务的原子性表示事务是否完全完成。一致性表示当事务执行失败时，所有被该事务影响的数据都应该恢复到事务执行前的状态。隔离性表示在事务执行过程中对数据的修改，在事务提交之前对其他事务是不可见的。持久性表示对数据的操作应该是持久的，但也是可以恢复的。

6.2 事务处理的 3 种方式

在了解了事务处理的基本概念之后，下面主要讲述实际开发过程中使用事务处理的方式，主要有 3 种，分别是关系型数据库的事务处理、传统的 JDBC 事务处理和分布式事务处理。

6.2.1 关系型数据库的事务处理

关系型数据库提供了事务处理的能力。在数据库中，事务处理一般分为 3 个步骤：首先告诉数据库开始一个事务，通知数据库把接下来的所有操作当作一个工作单元来对待；接着，执行一些相关的命令；最后通知数据库是否要提交或者回滚事务。当提交事务时，所作的更改就变为持久性的；当回滚事务时，所有的更改都被撤销。在数据库中，调用事务处理的示例代码如下：

```
Begin Transaction （启动事务处理）  
//事务处理步骤  
Commit 或 RollBack （提交或回滚）  
End Transaction （提交事务处理）
```

在上面两条声明之间的所有语句都成为事务处理的一部分。命令 Begin Transaction 位于整个事务处理的起始位置，因此其后的所有命令只有在执行到命令 End Transaction 时才会被一并执行。

🔗说明：事务之间的隔离程度使用 Isolation Level 来表示。其定义如下：read uncommitted：允许读取任何提交或未提交的记录，即不管记录是否提交都允许读取；read committed：只允许读取已经提交的记录。repeatable read：只能看到事务开始时刻的数据库的一个快照，事务开始后更改的所有记录不管提交未提交都无法看到。大多数数据库默认的隔离级别是 read committed。

6.2.2 传统的 JDBC 事务处理

常用的数据库接口，如 JDBC、ADO，都提供了基于数据连接进行事务处理的功能。一般的流程是：首先获取数据源，然后根据数据源获取数据连接，接着设定事务开始，执行相应的操作，最后执行成功则提交，执行失败则回滚。下面，通过示例来看 JDBC 中是怎么使用事务处理的，示例代码如下：

```
//***** TimeBook.java*****
Public Class HelloWorld {
    private DataSource dataSource;
    //获取数据源
    public void setDataSource (DataSource dataSource) {
        this.dataSource = dataSource;
    }
    Connection conn = null;
    Statement stmt = null;
    try {
        //获取数据连接
        conn = dataSource.getConnection();
        //开始启动事务
        conn.setAutoCommit(false);
        stmt = conn.createStatement();
        //执行相应操作
        stmt.executeUpdate("insert into hello values(1,'gf', 'HelloWorld')");
        //执行成功则提交事务
        conn.commit();
    } catch (SQLException e) {
        if (conn != null) {
            try {
                //执行不成功，则回滚
                conn.rollback();
            } catch (SQLException ex) {
                System.out.println("数据连接有异常" + ex);
            }
        }
    }
    } finally {
        //假如 stmt 不为空，则关闭 stmt
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException ex) {
                System.out.println("执行操作有异常" + ex);
            }
        }
    }
    if (conn != null) {
        try {
            conn.close();
        }
    }
}
```

```
        } catch (SQLException ex) {  
            System.out.println("数据连接有异常" + ex);  
        }  
    }  
}
```

上述代码基本上是一个完整的使用 JDBC 进行事务处理的示例，从上述代码可以看出，使用传统的 JDBC 事务处理，代码很繁琐，但这是必须的。

6.2.3 分布式事务处理

前面介绍的两种事务处理都是只针对一个数据库的事务操作，如果需要对多个数据库事务进行处理时，比如银行间的转账，前面的方法就无能为力了。

分布式事务就是事务分布在多个资源上、由多个组件共享的事务。分布式事务具有如下的特征：

- ❑ 组件要在同一原子操作中与多个资源通信。在前面的银行示例中，可能账户是农业银行的，要往民生银行进行转账，这就涉及到两家银行系统之间近期操作，当然也需要进行事务处理。
- ❑ 多个组件要在同一原子操作中操作。当从农业银行往民生银行进行转账时，要么两家银行的系统都操作成功，要么都操作失败，如果只是农业银行的账户扣钱了，而民生银行的账户没有增加钱，用户当然不会满意。
- ❑ 分布式事务需要多个不同的事务管理器的合作。

如果要对分布式事务处理进行讲解，可能得需要单独的一本书才可以，因为本书主要是对 Spring 进行讲解，所以这里只是把最基础的概念介绍给读者。前面介绍了一些事务处理的基础概念，那 Spring 给开发人员提供了什么样的事务处理功能呢？

6.3 Spring 的事务处理

Spring 提供了编程式事务处理（programmatic transaction management）与声明式事务处理（declarative transaction management）。下面首先对 Spring 提供的事务处理进行大概的讲解，然后再分别对这两种事务处理方式进行讲解。

6.3.1 Spring 事务处理概述

Spring 中的事务处理实际上是基于动态 AOP 机制实现。为了实现动态 AOP，前面讲过，Spring 在默认情况下会使用 Java 的动态代理机制，因为 Java 的动态代理机制要求其代理的对象必须实现一个接口，并且在该接口中定义准备进行代理的方法，而对于没有实现任何接口的 Java Bean，Spring 是通过 CGLIB 来实现的。

Spring 事务的中心接口是 `org.springframework.transaction.PlatformTransactionManager`, `PlatformTransactionManager` 的代码如下:

```
//***** PlatformTransactionManager.java*****  
public interface PlatformTransactionManager {  
    //目前的事务  
    TransactionStatus getTransaction(TransactionDefinition definition)  
        throws TransactionException;  
    //提交事务  
    void commit(TransactionStatus status) throws TransactionException;  
    //事务回滚  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

在 Spring 中有很多 `PlatformTransactionManager` 的实现类, 它们通常在 Spring 的配置文件中声明, 比如 `TransactionManager`。

`TransactionDefinition` 代表着事务处理时一些属性定义, 比如事务的名称、隔离层次 (Isolation Level)、传播行为 (Propagation Behavior)、只读 (Read-only) 和超时 (Timeout) 等。`TransactionDefinition` 的示例代码如下:

```
//***** TransactionDefinition.java*****  
public interface TransactionDefinition{  
    //获得事务的传播行为  
    int getPropagationBehavior();  
    //获得事务的隔离层次  
    int getIsolationLevel();  
    //判断事务是否超时  
    int getTimeout();  
    //判断是否为只读事务  
    boolean isReadOnly();  
    //返回一个事务的名字  
    String getName();  
}
```

`TransactionStatus` 代表了目前的事务, 通常不直接使用它, 可以借助它的 `setRollbackOnly()` 方法来设定只读事务。`TransactionStatus` 的示例代码如下:

```
//***** TransactionStatus.java*****  
public interface TransactionStatus {  
    //判断是否是一个新事务  
    boolean isNewTransaction();  
    //设定为只读事务  
    void setRollbackOnly();  
    //判断是否为只读事务  
    boolean isRollbackOnly();  
    //判断一个事务是否完成  
    boolean isCompleted();  
}
```

6.3.2 编程式事务处理

Spring 提供的 `TransactionTemplate` 能够以编程的方式实现事务控制。同样的，开发人员也是通过实现 `callback` 接口来使用它。`TransactionTemplate` 也是无状态且线程安全的。创建 `TransactionTemplate` 的实例需要提供一个 `PlatformTransactionManager` 的实例，前面讲过，Spring 提供了很多 `PlatformTransactionManager` 的实现类，提供多种事务控制的途径。

`TransactionTemplate` 的示例代码如下：

```
//***** TransactionTemplate.java*****
public class TransactionTemplate extends DefaultTransactionDefinition implements InitializingBean{
    private PlatformTransactionManager transactionManager = null;
    //通过依赖注入
    public TransactionTemplate(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }
    //执行完毕后调用
    public void afterPropertiesSet() {
        if (this.transactionManager == null) {
            throw new IllegalArgumentException("transactionManager is required");
        }
    }
    //在这里进行事务处理
    public Object execute(TransactionCallback action) throws TransactionException {
        TransactionStatus status = this.transactionManager.getTransaction(this);
        Object result = null;
        try {
            //执行具体的方法
            result = action.doInTransaction(status);
        }
        catch (RuntimeException ex) {
            // Transactional code threw application exception -> rollback
            rollbackOnException(status, ex);
            throw ex;
        }
        catch (Error err) {
            // Transactional code threw error -> rollback
            rollbackOnException(status, err);
            throw err;
        }
        this.transactionManager.commit(status);
        return result;
    }
}
```

```

    }
    //如果有异常就 rollback
    private void rollbackOnException(TransactionStatus status, Throwable ex) throws
    TransactionException {
        try {
            this.transactionManager.rollback(status);
        }
        catch (RuntimeException ex2) {
            throw ex2;
        }
        catch (Error err) {
            throw err;
        }
    }
}

```

代码说明：

- ❑ 使用 TransactionTemplate 很简单，只需要调用 execute()方法即可。
- ❑ 要实现 TransactionCallback 接口，并且在 doInTransaction()方法里进行各种数据库操作。
- ❑ 如果操作成功，则 this.transactionManager.commit(status)。
- ❑ 如果有异常，则相应地进行事务回滚 rollbackOnException(status, ex)。
- ❑ result 是执行 doInTransaction()方法后的返回值。

下面列举一个示例代码，来看一下 Spring 是如何利用 TransactionTemplate 来实现事务处理的，当然还需要一个 PlatformTransactionManager 的实现类，这里采用 DataSourceTransactionManager 类。首先来看一下在 HelloDAO 类的 create()方法里使用 TransactionTemplate 来进行事务处理，然后来看在 Spring 配置文档里是如何来配置事务的。具体实现步骤如下：

(1) 在 HelloDAO 类的 create()方法里使用 TransactionTemplate 来进行事务处理。HelloDAO.java 的示例代码如下：

```

//***** HelloDAO.java *****
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;

public class HelloDAO {
    private DataSource dataSource;
    private PlatformTransactionManager transactionManager;
    //通过依赖注入来完成管理
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}

```



```


    }

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    //该方法进行事务处理
    public int create(String msg) {
        TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
        Object result = transactionTemplate.execute(
            new TransactionCallback() {
                public Object doInTransaction(TransactionStatus status) {
                    // 执行新增的操作，向数据库新增一笔记录
                    .....
                    // 返回值是 resultObject
                    return resultObject;
                }
            });
    }
}

```

代码说明：

- ❑ 使用 TransactionTemplate 对 create()方法进行事务管理。
- ❑ 调用 TransactionTemplate 的 execute()方法，并覆写了 TransactionCallback 类的 doInTransaction()方法，在该方法里进行对数据库的新增操作。

 注意：TransactionTemplate 的 execute()方法中回调函数的使用，编码方式与通常的编码方式不太一样。

(2) 因为在 HelloDAO 中 DataSource 和 TransactionManager 采用注入的方式实现，所以还需要配置 XML。Spring 配置文档的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--设定 dataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--使用 SQL Server 数据库 -->
        <property name="driverClassName">
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
        </property>
        <!--设定 URL -->
        <property name="url">
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
        </property>
        <!--设定用户名-->
        <property name="name">
            <value>admin</value>
        </property>
    </bean>

```



```

        <!-- 设定密码 -->
        <property name="msg">
            <value>admin</value>
        </property>
    </bean>
    <!-- 设定 transactionManager -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
    </bean>
    <!-- 示例中的一个 DAO -->
    <bean id="helloDAO" class="com.gc.action.HelloDAO">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
        <property name="transactionManager">
            <ref bean="transactionManager"/>
        </property>
    </bean>
</beans>

```

代码说明：

- ❑ id 为 dataSource 的 Bean 是采用 DriverManagerDataSource 的数据源，后面会详细讲解，这里读者只要知道就可以了。
- ❑ id 为 transactionManager 的 Bean 采用的是 DataSourceTransactionManager 类，需要依赖 dataSource Bean。
- ❑ 定义 HelloDAO 的 id 为 helloDAO，并依赖注入 dataSource 和 transactionManager。

(3) 这样当业务逻辑程序调用 HelloDAO 的 create() 方法往数据库新增一笔记录时，Spring 会自动对这个操作使用事务处理，而不用再写 rollback 和 commit 的代码了。是不是比前面使用传统的 JDBC 方便多了，代码量也少多了？

(4) 当然如果开发人员想自己进行 rollback 和 commit，Spring 也提供了相应的功能来实现，只需要修改 HelloDAO，而配置文档和前面的一样，不需要任何改变。HelloDAO.java 的示例代码如下：

```

//***** HelloDAO.java *****
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;

public class HelloDAO {
    private DataSource dataSource;

```

```

private PlatformTransactionManager transactionManager;
//依赖注入
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

public void setTransactionManager(PlatformTransactionManager transactionManager) {
    this.transactionManager = transactionManager;
}
//往数据库表 hello 里新增一笔数据
public int create(String msg) {
    DefaultTransactionDefinition def = new DefaultTransactionDefinition();
    TransactionStatus status = transactionManager.getTransaction(def);
    try {
        //使用 JdbcTemplate 往数据库里新增数据
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update("INSERT INTO hello VALUES(1, 'gf', 'HelloWord')");
    } catch (DataAccessException ex) {
        // 也可以执行 status.setRollbackOnly();
        transactionManager.rollback(status);
        throw ex;
    } finally {
        transactionManager.commit(status);
    }
}
}

```

代码说明:

- ❑ 使用 Spring 预定义的 DefaultTransactionDefinition, 然后通过 TransactionManager 的 getTransaction()方法首先声明事务开始。
- ❑ 如果有异常发生, 则使用 TransactionManager 的 rollback()方法进行事务回滚。
- ❑ 如果成功, 则执行 TransactionManager 的 commit()方法进行事务提交。
- ❑ JdbcTemplate, 这里读者可以不用深究, 第7章会详细讲解它的使用方法。

(5) TransactionCallback 还有另外一个实现类 TransactionCallbackWithoutResult, 这个实现类中定义了 doInTransactionWithoutResult()方法, 从名字也可以看出, 这个没有结果返回。下面再展示一下 TransactionCallbackWithoutResult 的使用方法, 和使用 TransactionCallback 类似。HelloDAO.java 的示例代码如下:

```

//***** HelloDAO.java *****
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;

public class HelloDAO {

```

```
private DataSource dataSource;
private PlatformTransactionManager transactionManager;
//依赖注入
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

public void setTransactionManager(PlatformTransactionManager transactionManager) {
    this.transactionManager = transactionManager;
}
//使用回调的方式进行事务处理
public void create(String msg) {
    TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
    transactionTemplate.execute(new TransactionCallbackWithoutResult() {
        public void doInTransactionWithoutResult(TransactionStatus status) {
            JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
            jdbcTemplate.update("INSERT INTO hello VALUES(1, 'gf', 'HelloWord')");
        }
    });
}
```

上面这 3 种实现方法，从本质上讲是没有区别的，读者使用哪一个完全根据实际情况来决定，一般来说，推荐使用第一种方式。

6.3.3 声明式事务处理

Spring 的编程式事务处理虽然比其传统的 JDBC 有了很大的改进，但还是有些麻烦。因为要实现具体的方法，Spring 的声明式事务处理提供了更好的解决方法。

Spring 的事务处理需要借助于 AOP 的实现，所以需要用到开源的 aopalliance.jar 和 cglib.jar，因此需要把 spring-framework-2.0-m1/lib/aopalliance 文件夹中的 aopalliance.jar 和 spring-framework-2.0-m1/lib/cglib 文件夹中的 cglib-nodep-2.1_3.jar 加入到 CLASSPATH 中，因为前面在讲 AOP 时已经把 cglib-nodep-2.1_3.jar 加入到 CLASSPATH 中了，所以这里只需要把 aopalliance.jar 加入到 CLASSPATH 中就可以了。

下面列举一个示例代码，来看一下 Spring 是如何实现声明式事务处理的，当然还需要一个 PlatformTransactionManager 的实现类，同样采用 DataSourceTransactionManager 类，并使用 Spring 提供的 TransactionProxyFactoryBean 类来实现事务代理。首先在 HelloDAO 类的 create() 方法里使用 JdbcTemplate 往数据库新增一笔数据，然后来看在 Spring 配置文档中是如何来配置事务的。具体实现步骤如下：

(1) 在 HelloDAO 类的 create() 方法里使用 JdbcTemplate 往数据库新增一笔数据。HelloDAO.java 的示例代码如下：

```
//***** HelloDAO.java*****
package com.gc.action;
```



```
import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
public class HelloDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplate;
    //通过依赖注入
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    //注意这里看不到事务处理的代码，一切都在配置文件中
    public void create(String msg) {
        jdbcTemplate.update("INSERT INTO hello VALUES(1, 'gf', 'HelloWord')");
    }
}
```

(2) 因为在 HelloDAO 中 DataSource 采用注入的方式实现，而且还要使用 TransactionProxyFactoryBean 进行事务代理，所以还需要配置 XML。Spring 配置文档的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--设定 dataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--使用 SQL Server 数据库 -->
        <property name="driverClassName">
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
        </property>
        <!--设定 URL -->
        <property name="url">
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
        </property>
        <!--设定用户名-->
        <property name="name">
            <value>admin</value>
        </property>
        <!--设定密码-->
        <property name="password">
            <value>admin</value>
        </property>
    </bean>
    <!--设定 transactionManager -->
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-
TransactionManager">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
    </bean>
</beans>
```



```

</bean>
<!-- 示例中的一个 DAO -->
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
<!-- 声明式事务处理 -->
<bean id="helloDAOProxy" class="org.springframework.transaction.interceptor.Transaction-
ProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="target">
        <ref bean="helloDAO"/>
    </property>
    /* 如果当前没有事务，就新建一个事务 */
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
</beans>

```

代码说明：

- ❑ id 为 helloDAOProxy 的 Bean 的 target 属性，指明代理的是 HelloDAO 类。
- ❑ <prop key="create*">PROPAGATION_REQUIRED</prop>，表示对 HelloDAO 类中的 create() 方法进行事务管理，并指明如果当前没有事务，就新建一个事务。

下面罗列一下 Spring 自定义的一些指定事务的属性值。

- ❑ PROPAGATION_REQUIRED：如果当前没有事务，就新建一个事务。
- ❑ PROPAGATION_SUPPORTS：如果当前没有事务，就以非事务方式执行。
- ❑ PROPAGATION_MANDATORY：如果当前没有事务，就抛出异常。
- ❑ PROPAGATION_REQUIRES_NEW：新建事务，如果当前存在事务，把当前事务挂起。
- ❑ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- ❑ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

(3) 这样当业务逻辑程序调用 HelloDAO 的 create() 方法往数据库新增一笔记录时，Spring 会自动对这个操作使用事务处理，而在 HelloDAO 类的 create() 方法里却只有一行代码。

(4) 当然还有另外一种配置方式，HelloDAO 类不用更改。配置文档的示例代码如下所示：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

```

```

"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!-- 设定 dataSource -->
  <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <!-- 使用 SQL Server 数据库 -->
    <property name="driverClassName">
      <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
    </property>
    <!-- 设定 URL -->
    <property name="url">
      <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
    </property>
    <!-- 设定用户名 -->
    <property name="name">
      <value>admin</value>
    </property>
    <!-- 设定密码 -->
    <property name="msg">
      <value>admin</value>
    </property>

  </bean>
  <!-- 设定 transactionManager -->
  <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
  </bean>
  <!-- 示例中的一个 DAO -->
  <bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
  </bean>
  <!-- 声明式事务处理 -->
  <bean id="transactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
    /* 如果当前没有事务，就新建一个事务 */
    <property name="transactionAttributeSource">
      <value>
        com.gc.action.HelloDAO.create*=PROPAGATION_REQUIRED
      </value>
    </property>
  </bean>
  <bean id="helloDAOProxy" class="org.springframework.aop.framework.ProxyFactory-

```

```
Bean">  
    <property name="interceptorNames">  
        <value>transactionInterceptor,helloDAO</value>  
    </property>  
</bean>  
</beans>
```

上述代码和前面实现的功能是一样的，只是这种方式更加灵活。

6.4 使用编程式还是声明式事务处理

在一个小型软件系统中，如果只有很少的事务操作，使用编程式事务处理比较好。如果是一个大型的软件系统，而且有大量的事务操作，使用声明式事务处理则比较好，而且它使得事务处理与具体的业务逻辑分离。

6.5 小 结

本章首先讲述了事务处理的基础知识，接着对 Spring 提供的事务处理进行了讲解，使用 Spring 的 AOP 来实现事务处理，由此可以看出 Spring 的事务处理给开发人员带来了极大的方便，最后对 Spring 提供的两种事务处理方式进行了比较。

本章注重对概念的讲解，并提供了简单的示例代码，第 7 章将结合 Spring 对持久层的封装，更加详细地讲解 Spring 提供的事务处理。



第 7 章 Spring 的持久层封装

第 6 章介绍了事务处理的一些理论知识，本章主要结合 Spring 的持久层来讲解一下如何使用 Spring 的事务处理。首先对传统的数据库连接作一介绍，然后再介绍使用 Spring 怎样来进行持久层封装。

7.1 传统的 JDBC 数据访问技术

前面讲过，传统的 JDBC 数据访问技术的一般流程是：首先获取数据源，然后根据数据源获取数据连接，接着设定事务开始，执行相应的操作，最后执行成功则提交，执行失败则回滚。下面，通过示例来看 JDBC 中是怎么使用事务处理的。示例代码如下：

```
/** ***** TimeBook.java ***** */
Public Class HelloWorld {
    Connection conn = null;
    Statement stmt = null;
    try {
        //获取数据连接
        Class.forName(com.microsoft.jdbc.sqlserver.SQLServerDriver);
        conn = DriverManager.getConnection(jdbc:microsoft:sqlserver://localhost:1433/stdb, admin,
admin);
        //开始启动事务
        conn.setAutoCommit(false);
        stmt = conn.createStatement();
        //执行相应操作
        stmt.executeUpdate("insert into hello values(1,'gf', 'HelloWorld')");
        //执行成功则提交事务
        conn.commit();
    } catch (SQLException e) {
        if (conn != null) {
            try {
                //执行不成功，则回滚
                conn.rollback();
            } catch (SQLException ex) {
                System.out.println("数据连接有异常" + ex);
            }
        }
    }
    } finally {
        if (stmt != null) {
            try {
```



```
        stmt.close();
    } catch (SQLException ex) {
        System.out.println("执行操作有异常" + ex);
    }
}
if (conn != null) {
    try {
        conn.close();
    } catch (SQLException ex) {
        System.out.println("数据连接有异常" + ex);
    }
}
}
```

上面代码只是一个使用 JDBC 连接的示例，在实际的应用中，是不会这样用的，一方面编写代码繁琐，一方面效率太低，而 Spring 在持久层这方面提供了更好的支持，对 JDBC 进行了良好的封装。

🔔 注意：本章示例中使用的数据库表名为 hello，该表包含 3 个字段：id、name 和 msg。其中 id 为整型，其余两个都是字符串类型，以后不再重复说明。

7.2 通过 XML 实现 DataSource（数据源）注入

这里介绍 Spring 提供的 3 种通过 XML 实现 DataSource（数据源）注入的方式：使用 Spring 自带的 DriverManagerDataSource、使用 DBCP 连接池和使用 Tomcat 提供的 JNDI。下面分别进行介绍。

7.2.1 使用 Spring 自带的 DriverManagerDataSource

在第 6 章的例子中，所有示例的配置文档对于 DataSource 的注入使用的都是 Spring 提供的 DriverManagerDataSource。使用 DriverManagerDataSource 在效率上和直接使用 JDBC 没有多大的区别。使用 DriverManagerDataSource 的配置文档示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 设定 dataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!-- 使用 SQL Server 数据库 -->
        <property name="driverClassName">
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
        </property>
        <!-- 设定 URL -->
```

```
<property name="url">
    <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
</property>
<!--设定用户名-->
<property name="name">
    <value>admin</value>
</property>
<!--设定密码-->
<property name="msg">
    <value>admin</value>
</property>
</bean>
<!--设定 transactionManager -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-
TransactionManager">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
<!--示例中的一个 DAO -->
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
</beans>
```

配置文档中 id 为 helloDAO 的 Bean 的示例代码在第 6 章已经讲解过，这里只是把示例代码展示出来，以示过程的完整性。HelloDAO.java 的示例代码如下：

```
/** ***** HelloDAO.java ***** */
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;

public class HelloDAO {
    private DataSource dataSource;
    private PlatformTransactionManager transactionManager;
    //通过依赖注入实现
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

```

public void setTransactionManager(PlatformTransactionManager transactionManager) {
    this.transactionManager = transactionManager;
}
//新增数据
public int create(String msg) {
    TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
    Object result = transactionTemplate.execute(
        new TransactionCallback() {
            public Object doInTransaction(TransactionStatus status) {
                // 执行新增的操作，向数据库新增一笔记录

                .....
                // 返回值是 resultObject
                return resultObject;
            }
        }
    );
}
}

```

7.2.2 使用 DBCP 连接池

Spring 也提供了对 DBCP 连接池的支持，可以直接在配置文档中配置 DBCP 数据库连接池。要在 Spring 中使用 DBCP 连接池，需要将 spring-framework-2.0-m1\lib\jakarta-commons 文件夹中的 commons-collections.jar、commons-dbc.jar 和 commons-pool.jar 用前面介绍的方法加入到 ClassPath 中。使用 DBCP 连接池的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--设定 dataSource -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <!--使用 SQL Server 数据库 -->
        <property name="driverClassName">
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
        </property>
        <!--设定 URL -->
        <property name="url">
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
        </property>
        <!--设定用户名-->
        <property name="name">
            <value>admin</value>
        </property>
        <!--设定密码-->
        <property name="msg">
            <value>admin</value>
        </property>
    </bean>

```



```
</bean>
<!-- 设定 transactionManager -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-
TransactionManager">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
<!-- 示例中的一个 DAO -->
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
</beans>
```

HelloDAO 类的代码和上面的一样，不用改变，这里就不再展示了。

7.2.3 使用 Tomcat 提供的 JNDI

与使用 DBCP 连接池相比，使用 Spring 来进行 Web 开发，更多的是使用 Web 容器提供的数据库连接池功能，这里以使用 Tomcat 容器为例，来讲解一下在 Spring 中使用 Tomcat 提供的 JNDI 应该如何配置。首先要在 Tomcat 的 server.xml 中添加一下代码：

```
<Context path="/myApp" reloadable="true"
docBase="D:\eclipse\workspace\myApp" workDir="D:\eclipse\workspace\myApp\work" >
<Resource name="jdbc/opensdb" auth="Container"
type="javax.sql.DataSource"
factory="org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory"
driverClassName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
url="jdbc:microsoft:sqlserver://localhost:1433/stdb"
<!-- 设定用户名 -->
name="admin"
<!-- 设定密码 -->
msg="admin"
<!-- 设定最大连接数 -->
maxActive="10000"
<!-- 连接最大空闲时间 -->
maxIdle="10000"
<!-- 连接最大等待时间 -->
maxWait="10000"
removeAbandoned="true"
removeAbandonedTimeout="10"
logAbandoned="true"
/></Context>
```


然后 Spring 的配置文档的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!-- 设定 dataSource -->
  <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>jdbc/opensdb</value>
    </property>
  </bean>
  <!-- 设定 transactionManager -->
  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-
TransactionManager">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
  </bean>
  <!-- 示例中的一个 DAO -->
  <bean id="helloDAO" class="com.gc.action.HelloDAO">
    <!-- 依赖注入 dataSource -->
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
    <!-- 依赖注入 transactionManager -->
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
  </bean>
</beans>
```

同样，HelloDAO 的代码不用改变。

上面介绍的这 3 种实现 DataSource 注入的方式，给开发人员的 JDBC 编程带来了极大的方便，主要是因为 Spring 对 JDBC 进行了良好的封装。

7.3 使用 JdbcTemplate 访问数据

在介绍 JdbcTemplate 之前，有必要先介绍一下 Template 模式，因为在 Spring 中，这种模式有着大量的应用，比如前面讲过的 TransactionTemplate。然后再介绍如何使用 JdbcTemplate 查询、更新数据库。

7.3.1 Template 模式简介

Template 模式指的就是在父类中定义一个操作中算法的骨架或者说操作顺序，而将一

些步骤的具体实现延迟到子类中。这个模式可能是目前最简单的模式了，这里以一个示例加以说明。

假如要设计一个事务处理类，该类要处理事务，这个是确定的，但是要处理哪些事务却是不知道的，所以可以在父类中编写事务处理的骨架代码，指明事务处理的操作顺序，而具体的事务则可以延迟到子类中去实现。具体实现步骤如下：

(1) 在父类 Transaction 中编写事务处理的骨架代码，指明事务处理的操作顺序。Transaction.java 的示例代码如下：

```
/****** Transaction.java*****  
public abstract class Transaction{  
    //事务处理的骨架代码，指明事务处理的操作顺序  
    public Object execute() throws TransactionException {  
        this.transactionManager.getTransaction(this);  
        Object result = null;  
        try {  
            result = doInTransaction();  
        } catch (Error err) {  
            // Transactional code threw error -> rollback  
            this.transactionManager.rollback(this);  
            throw err;  
        }  
        this.transactionManager.commit(this);  
        return result;  
    }  
    //负责传入具体的事务  
    Public abstract Object doInTransaction ();  
}
```

(2) 在子类 SubTransaction 中编写具体要进行事务处理的代码。SubTransaction.java 的示例代码如下：

```
/****** SubTransaction.java*****  
public class SubTransaction extends Transaction {  
    //负责传入具体的事务  
    Public Object doInTransaction () {  
        //具体对数据库进行新增、修改、删除的代码  
        .....  
    }  
}
```

(3) 这样当有具体的业务逻辑调用 SubTransaction.Execute()方法时，就会进行事务处理。

⚠注意：在子类中，必须要延迟到子类进行时才在子类中做，否则没有必要在子类中做。而接口则无法像抽象类似的这样做，因为接口并不能具体实现任何操作。

7.3.2 回顾事务处理中 TransactionTemplate 的实现方式

然而 Spring 提供了另外一种实现 Template 模式的方法，利用接口回调函数，也是很有意思的，来看一下前面讲过的 Spring 的事务处理方式：

(1) TransactionTemplate 的示例代码如下：

```
//***** TransactionTemplate.java*****
public class TransactionTemplate extends DefaultTransactionDefinition implements InitializingBean{
    .....
    //进行事务处理的骨架代码，指明了事务处理的顺序
    public Object execute(TransactionCallback action) throws TransactionException {
        TransactionStatus status = this.transactionManager.getTransaction(this);
        Object result = null;
        try {
            result = action.doInTransaction(status);
        }
        catch (RuntimeException ex) {
            // Transactional code threw application exception -> rollback
            rollbackOnException(status, ex);
            throw ex;
        }
        catch (Error err) {
            // Transactional code threw error -> rollback
            rollbackOnException(status, err);
            throw err;
        }
        this.transactionManager.commit(status);
        return result;
    }
    //事务处理回调时调用
    private void rollbackOnException(TransactionStatus status, Throwable ex) throws
    TransactionException {
        try {
            //进行 rollback
            this.transactionManager.rollback(status);
        }
        catch (RuntimeException ex2) {
            throw ex2;
        }
        catch (Error err) {
            throw err;
        }
    }
}
```

代码说明：

□ 这里 TransactionTemplate 没有使用抽象类，在它的 execute()方法里定义事务处理的

骨架代码。

- 但是 execute()方法的 TransactionCallback 参数却是个接口，在这个接口中定义了 doInTransaction()方法。

(2) 接着来看一下 TransactionCallback 的实现。TransactionCallback.java 的示例代码如下：

```
/** ***** TransactionCallback.java ***** */
public interface TransactionCallback {
    Object doInTransaction(TransactionStatus status);
}
```

(3) 只要实现 TransactionCallback 接口，并在 doInTransaction()方法里编写具体要进行事务处理的代码就可以了。简单的示例代码如下：

```
/** ***** HelloDAO.java ***** */
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;

public class HelloDAO {
    private DataSource dataSource;
    private PlatformTransactionManager transactionManager;
    //依赖注入
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    //新增数据
    public int create(String msg) {
        TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
        Object result = transactionTemplate.execute(
            new TransactionCallback() {
                public Object doInTransaction(TransactionStatus status) {
                    // 执行新增的操作，向数据库新增一笔记录
                    .....
                    // 返回值是 resultObject
                    return resultObject;
                }
            }
        );
    }
}
```


代码说明：在 HelloDAO 的 create() 方法里，对 TransactionCallback 接口的 doInTransaction() 方法进行了实现。

多多钻研 Spring 的源代码对提高开发人员的编程能力有很大的帮助，接下来将要讲解的 JdbcTemplate 也同样用类似的方式实现了 Template 模式。

7.3.3 JdbcTemplate 的实现方式

在前面的很多示例中，都会看到 JdbcTemplate 的身影。JdbcTemplate 封装了传统 JDBC 的功能，却提供了更强大的功能，从名字可以看出 JdbcTemplate 也实现了 Template 模式，但它和 TransactionTemplate 又有了些不同。JdbcTemplate 的示例代码如下：

```
/****** JdbcTemplate.java *****  
public class JdbcTemplate extends JdbcAccessor implements JdbcOperations {  
    .....  
    //使用回调方法  
    public Object execute(ConnectionCallback action) throws DataAccessException {  
        Connection con = DataSourceUtils.getConnection(getDataSource());  
        try {  
            Connection conToUse = con;  
            if (this.nativeJdbcExtractor != null) {  
                // Extract native JDBC Connection, castable to OracleConnection or the like.  
                conToUse = this.nativeJdbcExtractor.getNativeConnection(con);  
            }  
            else {  
                // 创建一个连接代理  
                conToUse = createConnectionProxy(con);  
            }  
            return action.doInConnection(conToUse);  
        }  
        catch (SQLException ex) {  
            // 释放连接, to avoid potential connection pool deadlock  
            // in the case when the exception translator hasn't been initialized yet.  
            DataSourceUtils.releaseConnection(con, getDataSource());  
            con = null;  
            throw getExceptionTranslator().translate("ConnectionCallback", getSql(action), ex);  
        }  
        finally {  
            DataSourceUtils.releaseConnection(con, getDataSource());  
        }  
    }  
    //使用回调方法  
    public Object execute(StatementCallback action) throws DataAccessException {  
        Connection con = DataSourceUtils.getConnection(getDataSource());  
        Statement stmt = null;  
        try {  
            Connection conToUse = con;
```

```

        if (this.nativeJdbcExtractor != null &&
this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {
            conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
        }
        //创建 Statement
        stmt = conToUse.createStatement();
        applyStatementSettings(stmt);
        Statement stmtToUse = stmt;
        if (this.nativeJdbcExtractor != null) {
            stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
        }
        //初始化 Statement
        Object result = action.doInStatement(stmtToUse);
        SQLWarning warning = stmt.getWarnings();
        throwExceptionOnWarningIfNotIgnoringWarnings(warning);
        return result;
    }
    catch (SQLException ex) {
        //释放连接, to avoid potential connection pool deadlock
        // in the case when the exception translator hasn't been initialized yet.
        JdbcUtils.closeStatement(stmt);
        stmt = null;
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        throw getExceptionTranslator().translate("StatementCallback", getSql(action), ex);
    }
    finally {
        JdbcUtils.closeStatement(stmt);
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}
//直接传入 SQL 语句执行
public void execute(final String sql) throws DataAccessException {
    if (logger.isDebugEnabled()) {
        logger.debug("Executing SQL statement [" + sql + "]");
    }
    class ExecuteStatementCallback implements StatementCallback, SqlProvider {
        public Object doInStatement(Statement stmt) throws SQLException {
            stmt.execute(sql);
            return null;
        }
        //获取 SQL
        public String getSql() {
            return sql;
        }
    }
    //执行回调函数
    execute(new ExecuteStatementCallback());
}

```

```
}  
.....  
}
```

代码说明：

- ❑ 这里只是简单地罗列出 JdbcTemplate 的一些代码以供分析使用。
- ❑ 和 TransactionTemplate 类似，同样有 execute() 方法，并且 execute() 方法的产生是一个接口。
- ❑ 但和 TransactionTemplate 不同的是，JdbcTemplate 提供了更简单的操作方式，不需要在代码中使用回调方法，可以只是把 SQL 语句传入，直接执行。当然，如果使用回调方法也是可以的。

上面介绍了 JdbcTemplate 的实现方式，那 JdbcTemplate 该如何使用呢？下面主要通过示例代码来进行讲解。实现思路是：首先编写配置文档，然后通过在使用 JdbcTemplate，并和事务处理结合在一起。具体编写步骤如下：

(1) Spring 配置文档的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
  "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
  <!-- 设定 dataSource -->  
  <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <!-- 使用 SQL Server 数据库 -->  
    <property name="driverClassName">  
      <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>  
    </property>  
    <!-- 设定 URL -->  
    <property name="url">  
      <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>  
    </property>  
    <!-- 设定用户名 -->  
    <property name="name">  
      <value>admin</value>  
    </property>  
    <!-- 设定密码 -->  
    <property name="password">  
      <value>admin</value>  
    </property>  
  </bean>  
  <!-- 设定 transactionManager -->  
  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-  
TransactionManager">  
    <property name="dataSource">  
      <ref bean="dataSource"/>  
    </property>  
  </bean>  
  <!-- 示例中的一个 DAO -->
```



```
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <!--依赖注入 dataSource -->
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    <!--依赖注入 transactionManager -->
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
</beans>
```

因为在第6章进行过讲解，所以这里不再解释配置文档中的具体含义。

(2) 在类 HelloDAO 中使用 JdbcTemplate，并和事务处理结合在一起。HelloDAO.java 的示例代码如下：

```
/** ***** HelloDAO.java ***** */
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;

public class HelloDAO {
    private DataSource dataSource;
    private PlatformTransactionManager transactionManager;
    //设定 dataSource
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    //设定 transactionManager
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    //使用 JdbcTemplate
    public int create(String msg) {
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);
        try {
            JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
            jdbcTemplate.update("INSERT INTO hello VALUES(1, 'gf', 'HelloWord')");
        } catch (DataAccessException ex) {
            // 也可以执行 status.setRollbackOnly();
            transactionManager.rollback(status);
            throw ex;
        } finally {
            transactionManager.commit(status);
        }
    }
}
```



```

    }
}

```

代码说明：首先把配置文档中定义的 `dataSource` 通过 `JdbcTemplate` 的构造方法进行注入，然后直接执行 `JdbcTemplate` 的 `update()` 方法即可实现对数据库的操作，是不是比传统的 JDBC 方式简单多了？只用了两行代码就实现了执行数据库的操作。

(3) 开发人员还可以把配置文档中定义的 `dataSource`，通过 `JdbcTemplate` 的构造方法进行注入也省掉，直接把 `JdbcTemplate` 在配置文档中配置，并使 `JdbcTemplate` 依赖于 `dataSource`。配置文档的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--设定 dataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--使用 SQL Server 数据库 -->
        <property name="driverClassName">
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
        </property>
        <!--设定 URL -->
        <property name="url">
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
        </property>
        <!--设定用户名-->
        <property name="name">
            <value>admin</value>
        </property>
        <!--设定密码-->
        <property name="msg">
            <value>admin</value>
        </property>
    </bean>
    <!--设定 transactionManager -->
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-
TransactionManager">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
    </bean>
    <!--设定 jdbcTemplate -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
    </bean>
    <!--示例中的一个 DAO -->
    <bean id="helloDAO" class="com.gc.action.HelloDAO">

```

```

<!-- 依赖注入 jdbcTemplate -->
<property name=" jdbcTemplate ">
    <ref bean=" jdbcTemplate "/>
</property>
<!-- 依赖注入 transactionManager -->
<property name="transactionManager">
    <ref bean="transactionManager"/>
</property>
</bean>
</beans>

```

代码说明：

- ❑ 这里把 JdbcTemplate 通过配置文档定义，把 dataSource 注入到 JdbcTemplate 中。
- ❑ 把 JdbcTemplate 通过配置注入到 HelloDAO 中。

(4) 在类 HelloDAO 中使用 JdbcTemplate，但是不用编写代码把 dataSource 注入到 jdbcTemplate 中，并和事务处理结合在一起。HelloDAO.java 的示例代码如下：

```

//***** HelloDAO.java *****
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;
import org.springframework.jdbc.core.JdbcTemplate;

public class HelloDAO {
    private JdbcTemplate jdbcTemplate;
    private PlatformTransactionManager transactionManager;
    //设定 jdbcTemplate
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    //设定 transactionManager
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    //这里把 jdbcTemplate 通过依赖注入来实现
    public int create(String msg) {
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);
        try {
            jdbcTemplate.update("INSERT INTO hello VALUES(1, 'gf', 'HelloWord')");
        } catch (DataAccessException ex) {
            // 也可以执行 status.setRollbackOnly();
            transactionManager.rollback(status);
            throw ex;
        }
    }
}

```

```
        } finally {  
            transactionManager.commit(status);  
        }  
    }  
}
```

这里只有 1 行代码就实现了对数据库的操作，由此可以看出 Spring IoC 功能的强大，通过依赖注入，大大简化了开发人员的编码工作量，而且在 Spring 的配置文档中配置也是非常容易的。

(5) 开发人员甚至还可以将 SQL 语句也通过配置文档来进行配置，这样当需要更改 SQL 语句时，就不需要改变代码了，而只需要修改配置文档。通过配置文档进行 SQL 语句注入的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <!-- 设定 dataSource -->  
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <!-- 使用 SQL Server 数据库 -->  
        <property name="driverClassName">  
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>  
        </property>  
        <!-- 设定 URL -->  
        <property name="url">  
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>  
        </property>  
        <!-- 设定用户名 -->  
        <property name="name">  
            <value>admin</value>  
        </property>  
        <!-- 设定密码 -->  
        <property name="msg">  
            <value>admin</value>  
        </property>  
    </bean>  
    <!-- 设定 transactionManager -->  
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
        <property name="dataSource">  
            <ref bean="dataSource"/>  
        </property>  
    </bean>  
    <!-- 设定 jdbcTemplate -->  
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  
        <property name="dataSource">  
            <ref bean="dataSource"/>  
        </property>  
    </bean>
```



```
<!--示例中的一个 DAO -->
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="jdbcTemplate">
        <ref bean="jdbcTemplate"/>
    </property>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="sql">
        <value> INSERT INTO hello VALUES(1, 'gf', 'HelloWord')</value>
    </property>
</bean>
</beans>
```

代码说明：把 SQL 语句通过 IoC 注入到 HelloDAO 中。

(6) HelloDAO.java 的示例代码如下：

```
//***** HelloDAO.java*****
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;
import org.springframework.jdbc.core.JdbcTemplate;

public class HelloDAO {
    private JdbcTemplate jdbcTemplate;
    private PlatformTransactionManager transactionManager;
    private String sql;
    //通过依赖注入实现
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    //设定 transactionManager
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    //设定 SQL
    public void setSql(String sql) {
        this.sql = sql;
    }
    //把 SQL 语句也通过依赖注入的方式来实现
    public int create(String msg) {
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);
        try {
            //执行 SQL
```



```
        jdbcTemplate.update(this.sql);
    } catch (DataAccessException ex) {
        // 也可以执行 status.setRollbackOnly();
        transactionManager.rollback(status);
        throw ex;
    } finally {
        transactionManager.commit(status);
    }
}
```

上述 3 种方式都实现了同样的功能，当业务逻辑调用类 HelloDAO 中的 create() 方法时，都会向数据库增加一笔数据，而且代码量逐渐地减少，因为在 Spring 的配置文档中都通过依赖注入来进行设定。

7.3.4 使用 JdbcTemplate 查询数据库

通过对 JdbcTemplate 源代码的研读，可以看到，JdbcTemplate 提供了很多用来查询的数据库，比如 queryForMap()、queryForLong()、queryForInt() 和 queryForList() 等。这里只是简单地进行示例说明，使用 queryForList 查询的示例代码如下：

```
List rows = jdbcTemplate.queryForList("select * from hello");
Iterator it = rows.iterator();
//通过 Iterator 获取 list 中的值
while(it.hasNext()) {
    Map map = (Map) it.next();
    String id = map.get("id");
    String name = map.get("name");
    String msg = map.get("msg");
}
```

使用 queryForInt 查询获得 hello 表中记录数量的示例代码如下：

```
int count = jdbcTemplate.queryForInt("select count(*) from hello");
```

这里只简单介绍这两个查询的使用方法，如果想获得更多的查询功能，研读 JdbcTemplate 的源代码是很有用的。

7.3.5 使用 JdbcTemplate 更改数据库

使用 JdbcTemplate 的 update() 方法是进行数据库更改常用的方式。比如要往数据库插入一笔数据，则可以使用以下示例代码：

```
jdbcTemplate.update("inset into hello values('1', 'gf', 'HelloWorld')");
```

可以使用下面的这种示例代码来实现同样的功能：

```
jdbcTemplate.update("inset into hello values (?, ?, ?)",
```

```
new Object[] {1, 'gf', 'HelloWorld'});
```

还可以使用下面的这种示例代码来实现同样的功能：

```
jdbcTemplate.update("inset into hello values (?, ?, ?)",
    new PreparedStatementSetter() {
        public void setValues(PreparedStatement ps) throws SQLException {
            ps.setInt(1, 1);
            ps.setString(2, 'gf');
            ps.setString(3, 'HelloWorld');
        }
    });
```

对数据库的修改，同样可以使用上面的方法：

```
jdbcTemplate.update("update hello set name = 'gd', msg = 'HelloWorld' where id = 1");
```

可以使用下面的这种示例代码来实现同样的功能：

```
jdbcTemplate.update("update hello set name = ?, msg = ? where id = ?",
    new Object[] { 'gf', 'HelloWorld', 1 });
```

 注意：新增和修改时 Object[] 中参数的先后顺序是不一样的。

7.4 使用 ORM 工具访问数据

前面讲述了使用 JdbcTemplate 对数据库进行操作的方法，可是很多情况下，开发人员使用 JdbcTemplate 还是有一定的麻烦。Spring 还提供了与其他 ORM 工具结合的功能，本节就主要讲述 Spring 与 Hibernate、iBatis 的无缝结合。在讲解 Spring 与 Hibernate、iBatis 的结合前，先来看一下什么是 ORM。

7.4.1 ORM 简述

假如读者对 ORM 已经有一定的了解，则可以跳过本节。

ORM 的英文全称是 Object-Relational Mapping，中文名称是对象关系映射。

为什么要使用 ORM 呢？是因为开发人员使用的技术是面向对象技术，而使用的数据库却是关系型数据库。使用 SQL 带来了很大的麻烦，可是目前面向对象的数据库还没有完全成熟，那么有什么办法既使用 SQL 又能去掉或减少这些麻烦呢？使用 ORM，通过在对象和关系型之间建立起一座桥梁。Hibernate、iBatis 就是这样的 ORM 工具。

ORM 包括以下 4 部分：

- ☐ 一个对持久类对象进行 CRUD 操作的 API。
- ☐ 一个语言或 API 用来规定与类和类属性相关的查询。
- ☐ 一个规定 mapping metadata 的工具。
- ☐ 一种技术可以让 ORM 的实现同事务对象一起进行 dirty checking, lazy association

fetching 以及其他的优化操作。

ORM 模型的简单性简化了数据库查询过程。使用 ORM 查询工具，用户可以访问期望数据，而不必理解数据库的底层结构，开发人员不再需要关心底层数据库是怎么工作的了，甚至不需要知道数据库的结构，一切都交给 ORM 去管理了。

7.4.2 使用 Hibernate

Hibernate 也是一个开源的框架，主要应用在持久层方面。关于 Hibernate，在后面的章节中还有详细的介绍，本节只是简单地列出示例代码，让读者快速了解 Spring 和 Hibernate 结合在一起的应用。这里仍然介绍如何配置 Spring 的配置文档和 HelloDAO 如何编写，另外因为使用 Hibernate，需要增加一个 Hibernate 的配置文件 Hello.hbm.xml 和存放数据的类 Hello。具体步骤如下：

(1) 加入 Hibernate 后的 Spring 配置文档的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--设定 dataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--使用 SQL Server 数据库 -->
        <property name="driverClassName">
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
        </property>
        <!--设定 URL -->
        <property name="url">
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>
        </property>
        <!--设定用户名-->
        <property name="name">
            <value>admin</value>
        </property>
        <!--设定密码-->
        <property name="msg">
            <value>admin</value>
        </property>
    </bean>
    <!--使用 Hibernate 的 sessionFactory -->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactory-
Bean">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
        <property name="mappingResources">
            <list>
                <value>com/gc/action/ Hello.hbm.xml</value>
            </list>
        </property>
    </bean>
</beans>
```

```

        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <!--设定方言-->
            <prop key="hibernate.dialect">
                net.sf.hibernate.dialect.SQLServerDialect
            </prop>
            <!--是否显示 hql-->
            <prop key="hibernate.show_sql">
                true
            </prop>
        </props>
    </property>
</bean>
<!--设定 transactionManager -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-
TransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactory"/>
    </property>
</bean>
<!--示例中的一个 DAO -->
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="sessionFactory">
        <ref bean="sessionFactory"/>
    </property>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
</beans>

```

代码说明:

❑ Hibernate 中通过 SessionFactory 创建和维护 Session, 而 Spring 对 SessionFactory 的配置进行了整合。

❑ com/gc/action/Hello.hbm.xml, 表示 Hello.hbm.xml 放在包 com.gc.action 下。

(2) 把配置文件 Hello.hbm.xml 放在包 com.gc.action 下。Hello.hbm.xml 的示例代码如下:

```

<hibernate-mapping>
<class name="com.gc.action.Hello" table="hello" dynamic-update="false" dynamic-insert="false">
<id name="id" column="id" type="java.lang.Integer"/>
<property name="name"
type="java.lang.String"
update="true"
insert="true"
access="property"
<!--栏位名称为 msg-->

```



```
column="msg"
<!--字段长度为 50-->
length="50"/>
<property name="msg"
type="java.lang.String"
update="true"
insert="true"
access="property"
<!--栏位名称为 name-->
column="name"
<!--字段长度为 50-->
length="50" />
</class>
</hibernate-mapping>
```

(3) 把类 Hello 放在包 com.gc.action 下。Hello.java 的示例代码如下：

```
/** ***** Hello.java ***** */
/**
 * @hibernate.class table="hello"
 */
public class Hello {
    public Integer id;
    public String name;
    public String msg;
    /**
     * @hibernate.id
     * column="id"
     * type="java.lang.Integer"
     */
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    /**
     * @hibernate.property column="msg" length="50"
     */
    public String getMsg() {
        return msg;
    }
    public void setMsg(String msg) {
        this.msg = msg;
    }
    /**
     * @hibernate.property column="name" length="50"
     */
    public String getName() {
        return name;
    }
}
```

```
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

(4) 在类 HelloDAO 中使用 SessionFactory，并和事务处理结合在一起。HelloDAO.java 的示例代码如下：

```
//***** HelloDAO.java*****  
package com.gc.action;  
  
import javax.sql.DataSource;  
import org.springframework.jdbc.core.*;  
import org.springframework.transaction.*;  
import org.springframework.transaction.support.*;  
import org.springframework.dao.*;  
import org.springframework.orm.*;  
  
public class HelloDAO {  
    private SessionFactory sessionFactory;  
    private PlatformTransactionManager transactionManager;  
    //依赖注入 sessionFactory  
    public void setSessionFactory(DataSource sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
    //依赖注入 transactionManager  
    public void setTransactionManager(PlatformTransactionManager transactionManager) {  
        this.transactionManager = transactionManager;  
    }  
    //使用 HibernateTemplate 代替 JdbcTemplate  
    public int create(String msg) {  
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();  
        TransactionStatus status = transactionManager.getTransaction(def);  
        try {  
            HibernateTemplate hibernateTemplate = new HibernateTemplate(sessionFactory);  
            //定义一个 hello  
            Hello hello = new Hello();  
            hello.setId(1);  
            hello.setName("gf");  
            hello.setMsg("HelloWorld");  
            //保存或更新 hello  
            hibernateTemplate.saveOrUpdate(hello);  
        } catch (DataAccessException ex) {  
            // 也可以执行 status.setRollbackOnly();  
            transactionManager.rollback(status);  
            throw ex;  
        } finally {  
            transactionManager.commit(status);  
        }  
    }  
}
```

```
}  
}
```

上述代码实现的功能和前面使用 JdbcTemplate 是一样的。

7.4.3 使用 iBatis

iBatis 也是一个开源的框架，和 Hibernate 一样主要应用在持久层方面。关于 iBatis，在后面的章节中还有详细的介绍，本节只是简单地列出示例代码，让读者快速了解 Spring 和 iBatis 结合在一起的应用。这里仍然介绍如何配置 Spring 的配置文档和 HelloDAO 如何编写，另外因为使用 iBatis，需要增加一个 iBatis 的配置文件 sqlMapConfig.xml、Hello.xml 和存放数据的类 Hello。其具体步骤如下：

(1) 加入 iBatis 后的 Spring 配置文档的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <!-- 设定 dataSource -->  
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <!-- 使用 SQL Server 数据库 -->  
        <property name="driverClassName">  
            <value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>  
        </property>  
        <!-- 设定 URL -->  
        <property name="url">  
            <value>jdbc:microsoft:sqlserver://localhost:1433/stdb</value>  
        </property>  
        <!-- 设定用户名 -->  
        <property name="name">  
            <value>admin</value>  
        </property>  
        <!-- 设定密码 -->  
        <property name="password">  
            <value>admin</value>  
        </property>  
    </bean>  
    <!-- 使用 iBatis -->  
    <bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">  
        <property name="configLocation">  
            <value>WEB-INF/sqlMapConfig.xml</value>  
        </property>  
    </bean>  
    <!-- 设定 transactionManager -->  
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSource-  
TransactionManager">  
        <property name="dataSource">
```



```

        <ref bean="dataSource"/>
    </property>
</bean>
<!--示例中的一个 DAO -->
<bean id="helloDAO" class="com.gc.action.HelloDAO">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="sqlMap">
        <ref bean="sqlMap"/>
    </property>
</bean>
</beans>

```

代码说明：sqlMapConfig.xml，是 iBatis 的配置文件，放在 WEB-INF 下。

(2) sqlMapConfig.xml 的示例代码如下：

```

<sqlMapConfig>
    <sqlMap resource="com/gc/action/Hello.xml"/>
</sqlMapConfig>

```

(3) 把配置文件 Hello.xml 放在包 com.gc.action 下。Hello.xml 的示例代码如下：

```

<sqlMap namespace="Hello">
    <typeAlias alias="hello" type="com.gc.action.Hello" />
    <insert id="insertHello" parameterClass="hello">
        insert into hello ( id,name, msg) values ( #id#,#name#,#msg# )
    </insert>
</sqlMap>

```

(4) 类 Hello 和前面 Hibernate 使用的一样。Hello.java 的示例代码如下：

```

//***** Hello.java*****
public class Hello {
    public Integer id;
    public String name;
    public String msg;
    //获取 id
    public Integer getId() {
        return id;
    }
    //设定 id
    public void setId(Integer id) {
        this.id = id;
    }
    //获取 msg
    public String getMsg() {

```



```
        return msg;
    }
    //设定 msg
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获取 name
    public String getName() {
        return name;
    }
    //设定 name
    public void setName(String name) {
        this.name = name;
    }
}
```

(5) 使类 HelloDAO 继承，并和事务处理结合在一起。HelloDAO.java 的示例代码如下：

```
/****** HelloDAO.java *****/
package com.gc.action;

import javax.sql.DataSource;
import org.springframework.jdbc.core.*;
import org.springframework.transaction.*;
import org.springframework.transaction.support.*;
import org.springframework.dao.*;
import org.springframework.orm.*;
//继承 SqlMapClientDaoSupport
public class HelloDAO extends SqlMapClientDaoSupport {
    private PlatformTransactionManager transactionManager;
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    //通过 getSqlMapClientTemplate 来代替 JdbcTemplate
    public int create(String msg) {
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);
        try {
            //定义对象 hello
            Hello hello = new Hello();
            hello.setId(1);
            hello.setName("gf");
            hello.setMsg("HelloWorld");
            //往数据库新增对象 hello 中的数据
            getSqlMapClientTemplate().update("insertHello", hello);
        } catch (DataAccessException ex) {
            // 也可以执行 status.setRollbackOnly();
            transactionManager.rollback(status);
            throw ex;
        }
    }
}
```

```
    } finally {  
        transactionManager.commit(status);  
    }  
}
```

上述代码实现的功能和前面使用 JdbcTemplate 以及 Hibernate 是一样的,读者可以仔细比较一下它们之间的用法。

7.5 小 结

本章讲述了 Spring 对 JDBC 的封装,其核心是 JdbcTemplate,又讲了 Spring 与 Hibernate、iBatis 的无缝结合,主要目的是让读者了解 Spring 对持久层的封装,并学会 JdbcTemplate 和事务处理结合在一起使用的方法。

通过前面几章的介绍,读者对 Spring 的全貌有了大概的了解,当然 Spring 目前应用最广泛的应该是进行 Web 开发了,第 8 章将主要介绍 Spring 的 Web 框架。



第 8 章 Spring 的 Web 框架

经过前面几章的介绍，读者应该对 Spring 的基础知识有了大概的了解。当然 Spring 目前应用最广泛的应该是进行 Web 开发了，本章将主要介绍 Spring 的 Web 框架。本章首先从 MVC 的实现原理入手，然后通过一个实例快速了解 Spring MVC 的开发流程，接着讲解了 Spring 的模型、视图和控制器，以及数据验证和对国际化的支持，最后通过一个实例详细讲解了 Spring MVC 的开发过程。

8.1 Web 框架介绍

利用 Java 进行 Web 开发，或者说利用 JSP 进行 Web 开发，笔者经历了单纯的利用 JSP、利用 JSP+Servlet、利用自己实现的 MVC 框架、利用 Struts 和利用 Spring 的 MVC 这 5 个阶段。每个阶段都会有不同的收获，其中后 3 个阶段都是实现了 MVC 模式，对笔者的影响最大。因此在讲 Spring 的 MVC 之前，读者有必要了解一下 MVC 的实现原理，这种思想在某种意义上来说比使用 Spring 的 MVC 更重要。

8.1.1 MVC 模式简介

先来了解一下什么是模式：模式，其实就是解决某一类问题的方法论。把解决某类问题的方法总结归纳到理论高度，这就是模式。因为模式是一种指导，在一个良好的指导下，有助于开发人员完成任务，作出一个优良的设计方案，达到事半功倍的效果，而且会得到解决问题的最佳办法。

MVC 模式起源于 Smalltalk 语言，MVC 是 Model—View—Controller 的简写。它由以下 3 个部分组成：模型（Model）、视图（View）和控制器（Controller）。

（1）Model 代表的是应用的业务逻辑，包含应用程序的核心功能。模型封装了应用程序的状态，有时它包含的唯一功能就是状态，它对视图或控制器一无所知。

（2）View 是应用的视图层，提供模型的表示。它是应用程序的外观，视图可以访问模型的读方法，但不能访问写方法。此外，它对控制器一无所知。当更改模型时，视图应得到通知。

（3）Controller 是提供应用的处理过程控制，控制器对用户的输入作出反应，它创建并设置模型。控制器的作用是从客户端接收请求，并且选择执行相应的业务逻辑，然后把响应结果送回到客户端。

MVC 减弱了业务逻辑接口和数据接口之间的耦合。

8.1.2 MVC 模式的结构图

MVC 模式的结构图如图 8.1 所示。

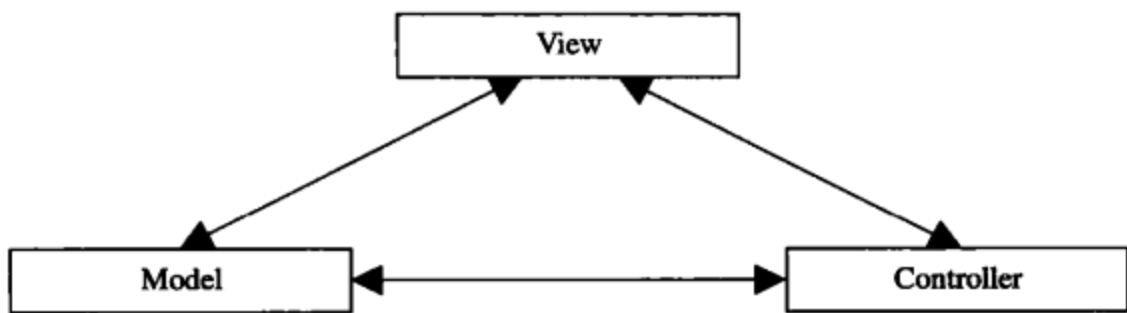


图 8.1 MVC 模式的结构图

8.1.3 MVC 模式的功能示意图

MVC 模式的功能示意图如图 8.2 所示。

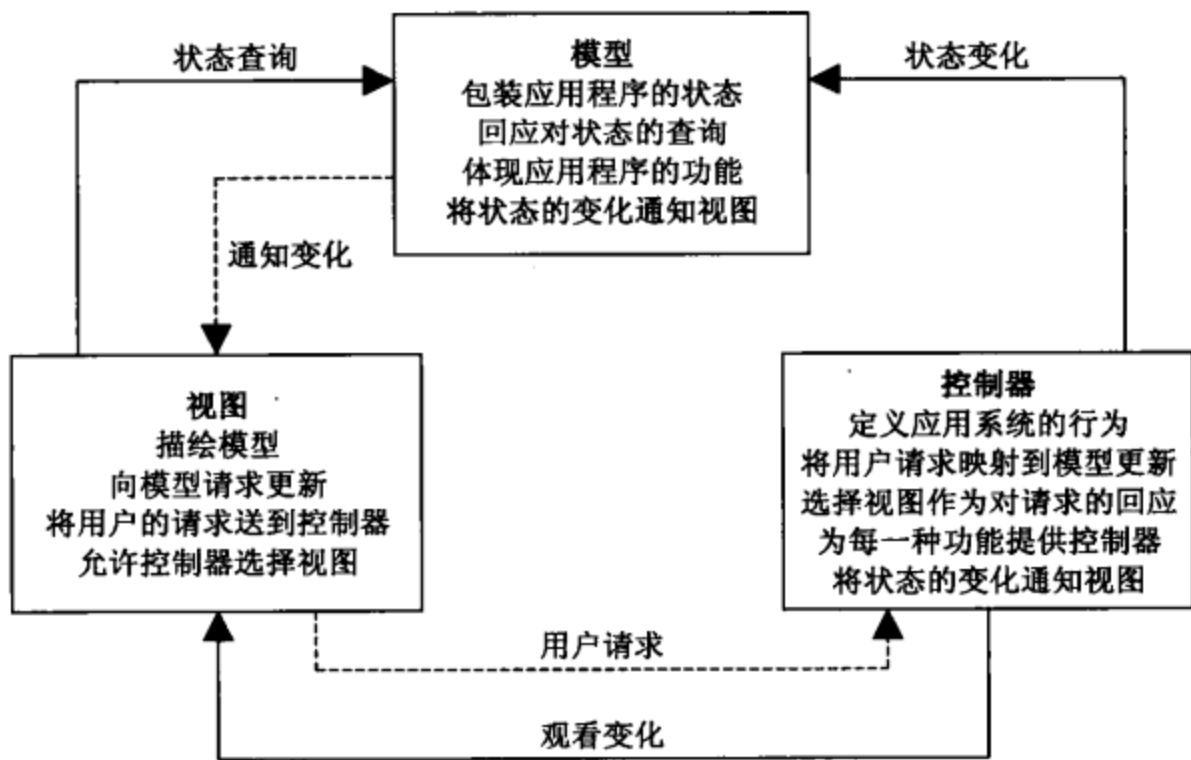


图 8.2 MVC 模式的功能示意图

8.1.4 使用 MVC 模式的好处

- ❑ 可靠性：表示层和业务层分离，这样就允许用户更改自己的表示层代码而不用重新编译模型（Model）和控制器（Controller）代码。
- ❑ 高重用和可适应性：MVC 模式允许用户使用各种不同样式的视图来访问同一个服务器端的代码。它包括任何 Web（HTTP）浏览器或者无线浏览器（WAP）。
- ❑ 较低的生命周期成本：MVC 使降低开发和维护用户接口的技术含量成为可能。
- ❑ 快速的部署：开发时间会得到相当大的缩减，使程序员（Java 开发人员）集中精力

于业务逻辑，界面程序员（HTML 和 JSP 开发人员）集中精力于表现形式上。

- 可维护性：分离表示层和业务逻辑层也使得基于 Struts 的 Web 应用更易于维护和修改。

MVC 模式并不能自动保证一个结构设计的正确，如何在一个系统的设计中正确地使用 MVC 模式与系统所使用的技术密切相关。

8.1.5 Model1 规范

MVC 设计模式很早就被提出，但在 Web 项目的开发中引入 MVC 却是步履维艰。主要原因：一是在早期的 Web 项目的开发中，程序语言和 HTML 的分离一直难以实现。CGI 程序以字符串输出的形式动态地生成 HTML 内容。后来随着脚本语言的出现，前面的方式又被倒了过来，改成将脚本语言书写的程序嵌入在 HTML 内容中。这两种方式有一个相同的不足之处，即它们总是无法将程序语言和 HTML 分离。二是脚本语言的功能相对较弱，缺乏支持 MVC 设计模式的一些必要的技术基础。直到基于 J2EE 的 JSP Model 2 问世时才得以改观。它用 JSP 技术实现视图的功能，用 Servlet 技术实现控制器的功能，用 JavaBean 技术实现模型的功能。

SUN 在 JSP 出现早期制定了两种规范，称为 Model1 和 Model2，它们是对采用 JSP 技术构成 Web 应用的不同模型的描述。

在使用 Java 技术建立 Web 应用的实例中，由于 JSP 技术的发展，很快这种便于掌握和可实现快速开发的技术就成了创建 Web 应用的主要技术。JSP 页面中可以非常容易地结合业务逻辑（jsp:useBean）、服务端处理过程（jsp:scriptlet）和 HTML（<html>），在 JSP 页面中同时实现显示、业务逻辑和流程控制，从而可以快速地完成应用开发。现在很多的 Web 应用就是由一组 JSP 页面构成的。这种以 JSP 为中心的开发模型可以称之为 Model1，Model1 的架构图如图 8.3 所示。

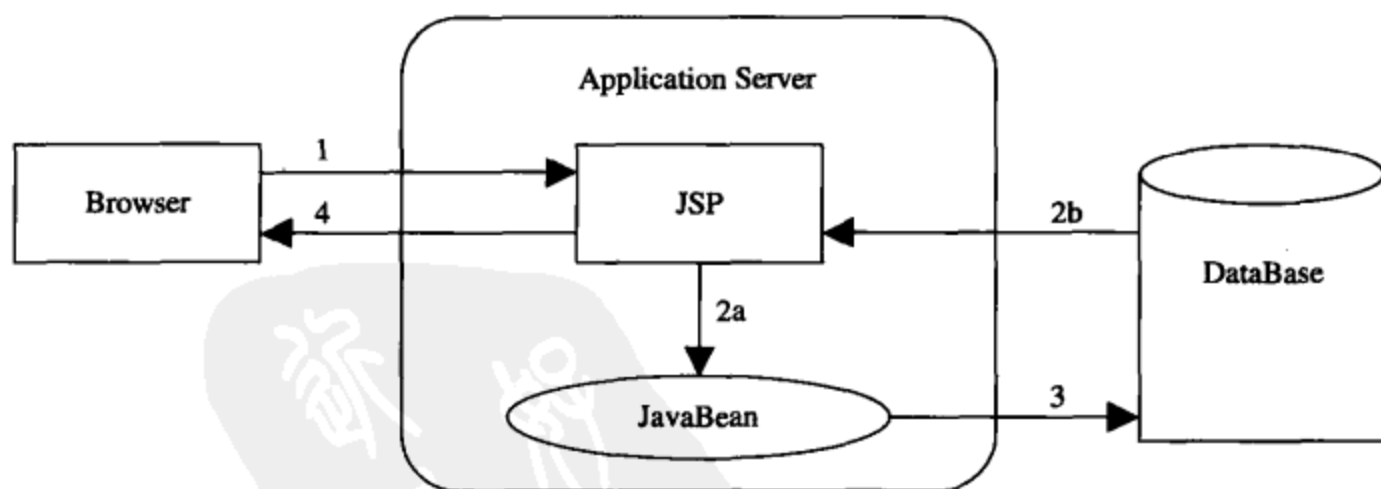


图 8.3 Model1 的架构图

当然这种开发模式在进行快速和小规模的应用开发时，是有非常大的优势，但是从工程化的角度考虑，它也有一些不足之处：

（1）应用的实现一般是基于过程的，一组 JSP 页面实现一个业务流程，如果要进行改动，必须在多个地方进行修改。这样非常不利于应用扩展和更新。

(2) 由于应用不是建立在模块上的, 业务逻辑和表示逻辑混合在 JSP 页面中没有进行抽象和分离, 所以非常不利于应用系统业务的重用和改动。

8.1.6 Model2 规范

考虑到这些问题在开发大型的 Web 应用时必须采用不同的设计模式——这就是 Model2。Model2 表示的是基于 MVC 模式的框架。通过这种设计模型把应用逻辑、处理过程和显示逻辑分成不同的组件实现。这些组件可以进行交互和重用, 从而弥补了 Model1 的不足。Model2 的架构图如图 8.4 所示。

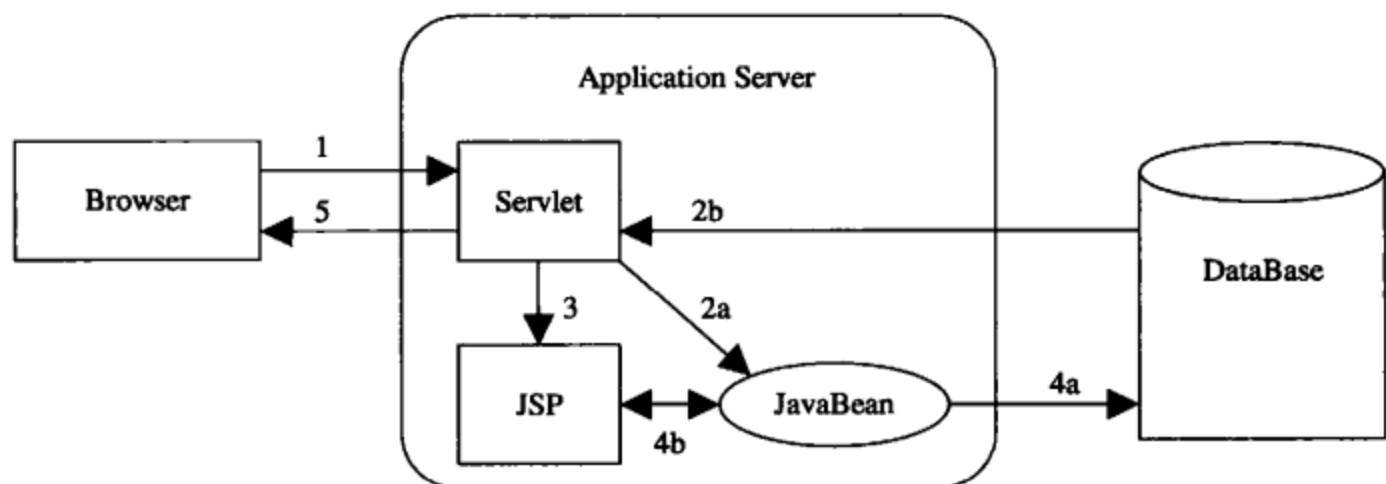


图 8.4 Model2 的架构图

Model2 具有组件化的优点, 从而更易于实现对大规模系统的开发和管理, 但是开发 MVC 系统比简单的 JSP 开发要复杂许多, 它需要更多的时间学习和掌握, 同时新东西的引入会带来新的问题:

(1) 必须基于 MVC 组件的方式重新思考和设计应用结构。原来通过建立一个简单的 JSP 页面就能实现的应用现在变成了多个步骤的设计和实现过程。

(2) 所有的页面和组件必须在 MVC 框架中实现, 所以必须进行附加的开发工作。

(3) 客户机和服务器的无状态连接。这种无状态行为使得模型很难将更改通知视图。在 Web 上, 为了发现对应用程序状态的修改, 浏览器必须重新查询服务器。

MVC 本身就是一个非常复杂的系统, 所以采用 MVC 实现 Web 应用时, 最好选一个现成的 MVC 框架, 在此之下进行开发, 从而取得事半功倍的效果。现在有很多可供使用的 MVC 框架, Spring MVC 就是其中比较优秀的一个。Spring MVC 本质上就是在 Model2 的基础上实现的一个 MVC 架构, Spring MVC 的 Model2 示意图如图 8.5 所示。

将 MVC 模式应用到一个系统设计中的过程就是对系统中不同责任的划分过程。

8.1.7 Spring MVC 的特点

就目前所出现的那些实现了 MVC 的框架来说, 它们提供了大体一致的功能, 因此如果会使用其中的一个框架, 则对于另外框架的学习和使用也将是很容易的, 只是 Spring 在 MVC 这方面还提供了一些与众不同的特点:

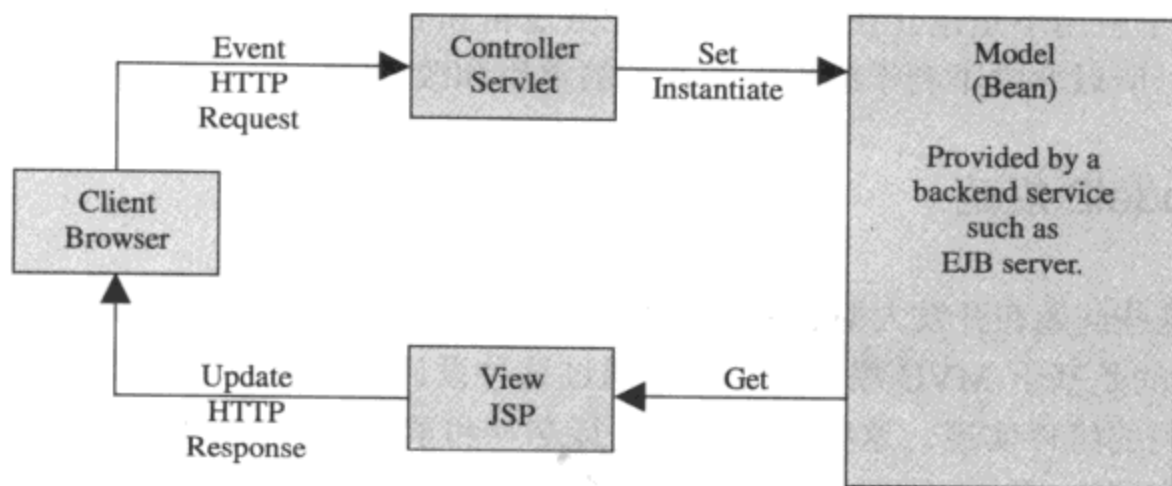


图 8.5 Spring MVC 的 Model2 示意图

(1) 清晰的角色划分，Spring 在 Model、View 和 Controller 方面提供了一个非常清晰的划分，这 3 个方面真正是各司其职，各负其责。

(2) 灵活的配置功能，因为 Spring 的核心是 IoC，同样在实现 MVC 上，也可以把各种类当作 Bean 来通过 XML 进行配置。

(3) 提供了大量的控制器接口和实现类，这样开发人员可以使用 Spring 提供的控制器实现类，也可以自己实现控制器接口。

(4) Spring MVC 是真正的 View 层的实现无关的，它不会强制开发人员使用 JSP，开发人员还可以使用 Velocity、Xslt 等技术。

(5) 国际化支持，前面讲过 Spring 的 ApplicationContext 提供了对国际化的支持，在这里可以很方便地使用。

(6) 面向接口编程，其实这不仅是 Spring MVC 的特点，就整个 Spring 来看，这个特点都是很明显的，因为它使开发人员对程序易于进行测试，并且很方便地进行管理。

(7) Spring 提供了 Web 应用开发的一整套流程，而不仅仅是 MVC，而且它们之间可以很方便地结合在一起。

以上总结的 Spring 的这 7 个特点，是笔者认为 Spring 提供的 MVC 和其他 MVC 的实现框架最主要的区别。

一个好框架要减轻开发者处理复杂问题的负担，内部要有良好的扩展，并且有一个支持它的强大的用户团体。

8.2 一个在 JSP 页面输出 “HelloWorld” 的 Spring MVC 实例

在详细讲解 Spring 的 MVC 之前，来看一个在 JSP 页面输出 “HelloWorld” 的 Spring MVC 实例，先大体认识一下使用 Spring MVC 的流程。

这个实例的实现思路是：使用第 2 章中配置好的 myApp 工程，首先新建一个 web.xml 文件，然后编写实现输出的 JSP 页面 index.jsp，接着编写控制器 HelloWorldAction.java，最

后配置 Spring 文档 dispatcherServlet-servlet.xml，并运行实例。

8.2.1 配置 web.xml

web.xml 的主要作用是：装载 DispatcherServlet 类，读取 Spring 配置文件，设置一些初始参数，加入标记库，设置如 *.do、*.form 的映射等。用记事本新建一个 web.xml，示例代码如下：

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

<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--初始化参数-->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/dispatcherServlet-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!--拦截所有以 do 结尾的请求-->
  <servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>
```

代码说明：

- ❑ web.xml 的开头几行是固定的，它定义了程序部署描述文件的字符编码、版本等，最顶层的元素为<web-app>，其余元素都要定义在<web-app>内。
- ❑ <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">，采用的是 Servlet 2.4 规定的 Web 程序部署描述格式。以前的 Servlet 经常使用的描述格式是：<!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>。
- ❑ <servlet>用来定义一个 Servlet。
- ❑ <servlet-name>是<servlet>的属性，用来定义 Servlet 的名称，这里是 dispatcherServlet。
- ❑ <servlet-class>是<servlet>的属性，用来指定上面定义 Servlet 的具体实现类。这里是 org.springframework.web.servlet.DispatcherServlet。

- ❑ `<init-param>`是`<servlet>`的属性，用来定义 Servlet 的初始化参数，这里指定要初始化 WEB-INF 文件夹下的 `dispatcherServlet-servlet.xml`，如果 `dispatcherServlet-servlet.xml` 的命名方式是前面定义 `servlet-name+“-servlet.xml”`，则可以不用定义这个初始化参数，Spring 默认会处理这个配置文件，这里其实是不用定义的。一个`<servlet>`可以有多个`<init-param>`。
- ❑ `<load-on-startup>`是`<servlet>`的属性，指定当 Web 启动时，加载 Servlet 的顺序，当它的值大于等于零时，Servlet 容器先加载数值小的 Servlet，再加载大的；如果它的值小于零或没有设定，则 Servlet 容器将在 Web 首次访问时加载这个 Servlet。
- ❑ `<servlet-mapping>`用来指定访问 Servlet 时的 URL。
- ❑ `<servlet-name>`用来定义 Servlet 的名称，和前面`<servlet>`中定义的名称相对应。
- ❑ `<url-pattern>`指定访问 Servlet 时 URL 的相对路径，这里表示所有以 `do` 结尾的都要经过指定的 `dispatcherServlet`。

8.2.2 编写实现输出的 JSP 页面 index.jsp

这里先采用普通的 JSP 页面，在 `myApp/WEB-INF` 下新建一个 JSP 文件夹，把 `index.jsp` 放在这个文件夹下。`index.jsp` 的示例代码如下：

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title>第一个 Spring MVC 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <H1><%=str%></H1>
</body>
</html>
```

8.2.3 编写控制器 HelloWorldAction.java

在 Spring 里，所有的控制器最终都可以看作是实现了 `Controller` 接口。为示例目的，这里的控制器类 `HelloWorldAction` 也简单实现了 `Controller` 接口。在 `com.gc.action` 包里新建一个类 `HelloWorldAction`，`HelloWorldAction.java` 的示例代码如下。

其中 `handleRequest()` 方法返回一个 `ModelAndView`，用来显示返回的页面和页面上的内容。

```
//***** HelloWorldAction .java*****
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

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

import org.apache.log4j.Logger;
import org.springframework.web.bind.RequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
//实现了 Controller 接口
public class HelloWorldAction implements Controller {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private String helloWorld;
    private String viewPage;
    //实现 Controller 接口中的 handleRequest()方法
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        Map model = new HashMap();
        model.put("helloWorld", getHelloWorld());
        return new ModelAndView(getViewPage(), model);
    }
    //依赖注入要返回的页面
    public void setViewPage(String viewPage) {
        this.viewPage = viewPage;
    }
    //获取要返回的页面
    public String getViewPage() {
        return viewPage;
    }
    //依赖注入也显示在页面中的文字
    public void setHelloWorld(String helloWorld) {
        this.helloWorld = helloWorld;
    }
    //获取要显示在页面中的文字
    public String getHelloWorld() {
        return helloWorld;
    }
}
```

8.2.4 配置 Spring 文档 dispatcherServlet-servlet.xml

因为前面定义 web.xml 时，定义的初始化参数是初始化 WEB-INF 文件夹下的 dispatcherServlet-servlet.xml，所以在 WEB-INF 文件夹下建立 dispatcherServlet-servlet.xml。其示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
```

```
<beans>
  <!--定义映射-->
  <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
    <property name="mappings">
      <props>
        <prop key="helloWorld.do">helloWorldAction</prop>
      </props>
    </property>
  </bean>
  <!--定义视图-->
  <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
    <property name="viewClass">
      <value>org.springframework.web.servlet.view.InternalResourceView</value>
    </property>
  </bean>
  <!--定义控制器-->
  <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="helloWorld">
      <value>HelloWorld</value>
    </property>
    <property name="viewPage">
      <value>/WEB-INF/jsp/index.jsp</value>
    </property>
  </bean>
</beans>
```

代码说明：

- ❑ id 为 urlMapping 的 Bean，用来定义一个映射，对 helloWorld.do 的访问将会映射到 id 为 helloWorldAction 的 Bean 中。
- ❑ id 为 viewResolver 的 Bean，用来定义视图解析器，通过 viewClass 属性来指定使用的视图，这里指定为 org.springframework.web.servlet.view.InternalResourceView，表示使用的是 JSP/Servlet 技术。
- ❑ id 为 helloWorldAction 的 Bean，指定对 helloWorld.do 访问时使用的控制器为 com.gc.action.HelloWorldAction。
- ❑ helloWorld 属性为控制器通过依赖注入要在页面中显示的内容，这里要在页面中显示 HelloWorld。
- ❑ viewPage 属性为控制器通过依赖注入要返回的页面，这里要返回的页面是 /WEB-INF/jsp/index.jsp。

8.2.5 启动 Tomcat

因为本书使用的开发工具是 Eclipse，在第 2 章中已经配置好 Tomcat 和 Eclipse，所以这里直接单击 Eclipse 工具按钮  启动 Tomcat。Tomcat 启动成功后显示的信息如图 8.6 所示。

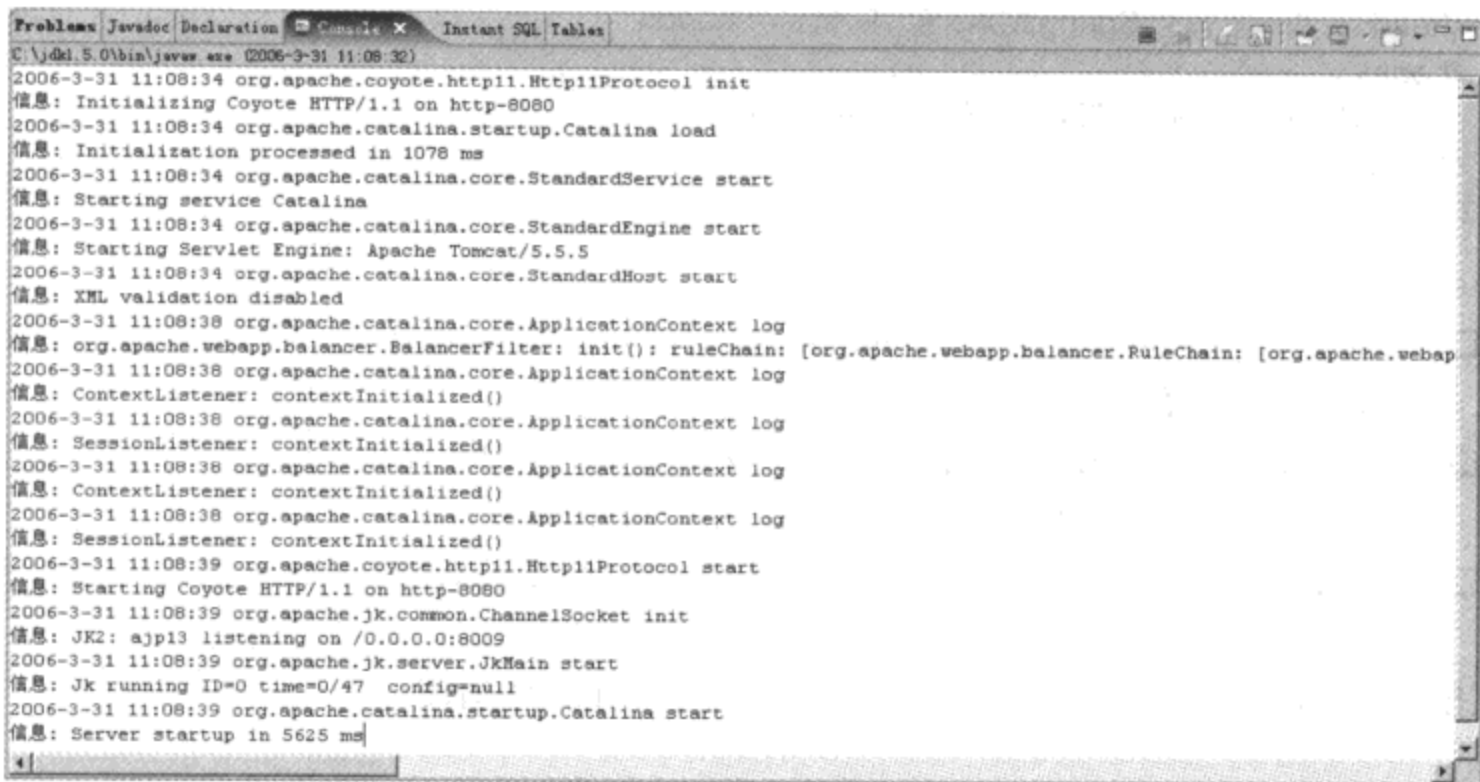


图 8.6 Tomcat 启动成功后显示的信息

8.2.6 运行程序

在浏览器的地址栏中输入 `http://localhost:8080/myApp/helloWorld.do`，按 Enter 键即可看到在浏览器中输出“HelloWorld”，如图 8.7 所示。

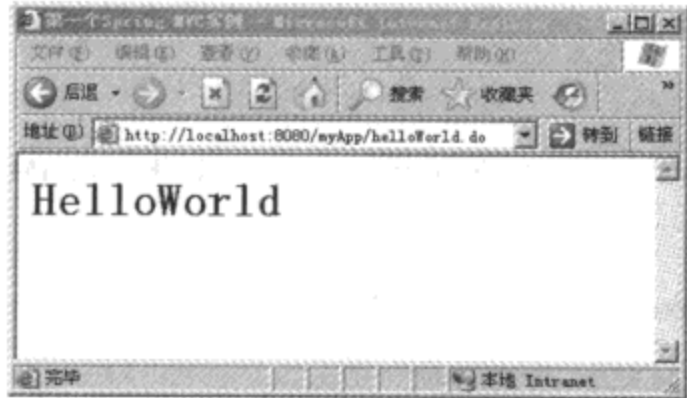


图 8.7 在浏览器中输出“HelloWorld”

8.2.7 把 index.jsp 改为使用 Jstl

Jstl 给开发人员提供了更为方便地在页面中输出内容的方式，下面笔者就把刚才的 `index.jsp` 改为使用 Jstl，首先将 `spring-framework-2.0-m1/lib/j2ee` 文件夹下的 `jstl.jar` 和 `spring-framework-2.0-m1/lib/jakarta-taglibs` 文件夹下的 `standard.jar` 复制到 `myApp/WEB-INF/lib` 下，加入到 `CLASSPATH` 中。修改后的 `index.jsp` 代码如下：

```
<%@ taglib prefix="c" url="http://java.sun.com/jstl/core_rt" %>
<%@ page contentType="text/html; charset=GBK" %>
<%@ page isELIgnored="false" %>
<html>
```



```
<head><title>第一个 Spring MVC 实例</title></head>
<body>
  <H1><c:out value="${helloWord}"/></H2>
</body>
</html>
```

代码说明：

- ❑ `<%@ taglib prefix="c" url="http://java.sun.com/jstl/core_rt" %>`表示使用的是 Jstl 的 core_rt，前缀是 c。
- ❑ `<%@ page isELIgnored="false" %>`，表示是否使用 EL 语法，设定为 false，表示使用 EL 语法，如果不写这句，则默认使用 EL 语法
- ❑ `<c:out value="${helloWord}"/>`，通过 EL 语法 `${helloWord}` 输出处理器中的内容。

8.2.8 修改配置文档 dispatcherServlet-servlet.xml

从上面的配置文档可以看出，如果页面比较多，在 Bean 里都设定每个页面的具体路径 /WEB-INF/jsp/index.jsp 时，如果将来要改变页面文件的存放位置，则每个这样的路径都要进行修改，比较麻烦。Spring 提供了更简单的方法来设定页面文件的存放路径。修改后的配置文档 dispatcherServlet-servlet.xml 示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!--定义映射-->
  <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="helloWorld.do">helloWorldAction</prop>
      </props>
    </property>
  </bean>
  <!--定义视图及 JSP 存放的路径-->
  <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
      <value>org.springframework.web.servlet.view.JstlView </value>
    </property>
    <!--JSP 存放的目录-->
    <property name="prefix">
      <value>/WEB-INF/jsp/</value>
    </property>
    <!--JSP 文件的后缀-->
    <property name="suffix">
      <value>.jsp</value>
    </property>
  </bean>
</beans>
```

```
</bean>
<!--定义控制器-->
<bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="helloWorld">
        <value>HelloWorld</value>
    </property>
    <property name="viewPage">
        <value>index</value>
    </property>
</bean>
</beans>
```

代码说明：

- ☐ id 为 urlMapping 的 Bean，用来定义一个映射，对 helloWorld.do 的访问将会映射到 id 为 helloWorldAction 的 Bean 中。
- ☐ id 为 viewResolver 的 Bean，用来定义视图解析器，通过 viewClass 属性来指定使用的视图，这里指定为 org.springframework.web.servlet.view.JstlView，表示使用的是 Jstl 技术。
- ☐ prefix 属性指明页面文件存放的文件夹。
- ☐ suffix 属性指明页面文件的后缀。
- ☐ viewPage 属性为控制器通过依赖注入要返回的页面，这里要返回的页面是 index。

8.2.9 运行修改后的程序

在浏览器的地址栏中输入 `http://localhost:8080/myApp/helloWorld.do`，按 Enter 键即可看到在浏览器中输出“HelloWorld”，如图 8.8 所示。



图 8.8 在浏览器中输出“HelloWorld”

可以看到，输出了和前面一样的信息。

8.2.10 使用 Log4j 时应该注意的问题

在启动 Tomcat 后，有时会看到这样的日志异常输出信息，如图 8.9 所示。

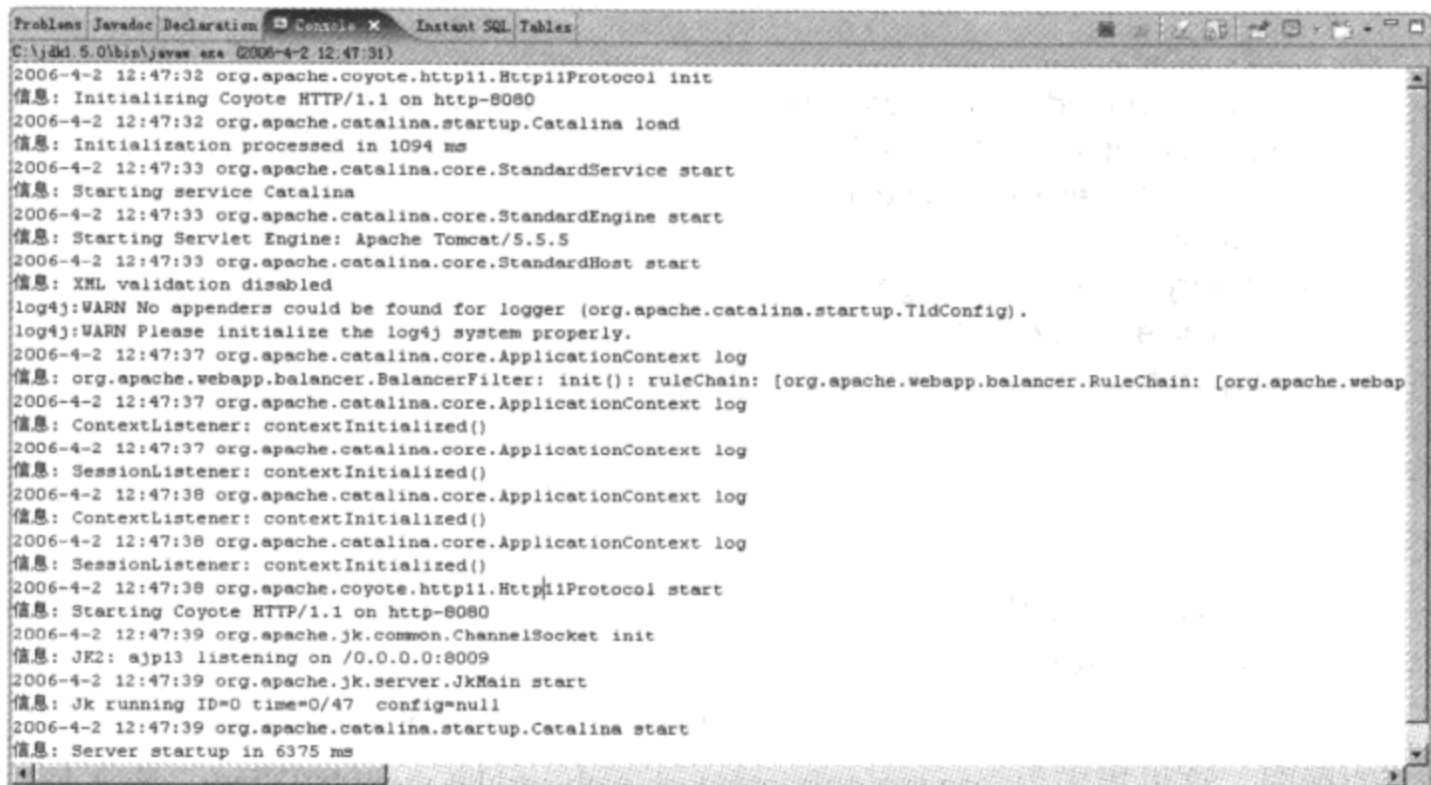


图 8.9 启动 Tomcat 后的日志异常输出信息

在输出的信息里多了以下信息：

```
log4j:WARN No appenders could be found for logger (org.apache.catalina.session.ManagerBase).
log4j:WARN Please initialize the log4j system properly.
```

这是因为没有把 log4j 的 log4j.properties 文件放在 myApp/WEB-INF/src 下的缘故，这时只要把 log4j.properties 放在 myApp/WEB-INF/src 下，重启 Tomcat 就不会出现这样的信息了。

8.3 Spring MVC 的模型和视图（ModelAndView）

就笔者所了解的那些实现 MVC 的框架来说，包括 Spring 在内，它们的重点都在 View 和 Controller 上，而对于 Model 层基本上都没有怎么实现，这是因为 Model 层在实现上没有什么特别固定的方式，对于 Model 层不外乎就是采用面向对象的技术来实现，而这一方面并不需要那些实现 MVC 的框架来做什么，靠的是开发设计人员对面向对象技术的掌握，这里 Spring 也是仅仅提供了一个 ModelAndView 类，其实就是利用 Map 来存储 Model 层处理后的结果集合。

8.3.1 模型和视图

这里主要详细讲解 Spring 提供的 org.springframework.web.servlet.ModelAndView 类，所有的控制器都会返回一个 ModelAndView，用来负责传递 Model 层处理后的结果集合和指定 View 层的信息。ModelAndView 的示例代码如下：

```
//***** ModelAndView.java *****
public class ModelAndView {
```

```

private Object view;
private Map model;
.....
//带参数 View 的构造函数
public ModelAndView(View view) {
    this.view = view;
}
//带参数 viewName 的构造函数
public ModelAndView(String viewName) {
    this.view = viewName;
}
//带参数 View 和 model 的构造函数
public ModelAndView(View view, Map model) {
    this.view = view;
    this.model = model;
}
//带参数 viewName 和 model 的构造函数
public ModelAndView(String viewName, Map model) {
    this.view = viewName;
    this.model = model;
}
//带参数 View、modelName 和 modelObject 的构造函数
public ModelAndView(View view, String modelName, Object modelObject) {
    this.view = view;
    addObject(modelName, modelObject);
}
//带参数 viewName、modelName 和 modelObject 的构造函数
public ModelAndView(String viewName, String modelName, Object modelObject) {
    this.view = viewName;
    addObject(modelName, modelObject);
}
//获取 model
public Map getModel() {
    if (this.model == null) {
        this.model = new HashMap(1);
    }
    return this.model;
}
//增加 model
public ModelAndView addObject(String modelName, Object modelObject) {
    getModel().put(modelName, modelObject);
    return this;
}
.....
}

```

代码说明:

- ❑ Object view, 用来存储一个视图类或者视图名称。
- ❑ Map model, 用来存储模型层处理后的结果集合, 使用 Map 接口。

- ❑ ModelAndView(View view), 只指定要返回的视图层, 表示视图层不需要返回结果集合。
- ❑ ModelAndView(String viewName), 只指定要返回的视图层的名称, 而具体的路径则可以在配置文档中设定。
- ❑ ModelAndView(View view, Map model), 指定要返回的视图层, 并将结果集合存放在 Map 中, 供视图层显示使用。
- ❑ ModelAndView(String viewName, Map model), 指定要返回的视图层的名称, 而具体的路径则可以在配置文档中设定, 并将结果集合存放在 Map 中, 供视图层显示使用。
- ❑ ModelAndView(View view, String modelName, Object modelObject), 指定要返回的视图层, 并将供视图层显示使用的结果和存放结果的名称通过 addObject(String modelName, Object modelObject) 存放在 Map 中。
- ❑ ModelAndView(String viewName, String modelName, Object modelObject), 指定要返回的视图层的名称, 而具体的路径则可以在配置文档中设定, 并将供视图层显示使用的结果和存放结果的名称通过 addObject(String modelName, Object modelObject) 存放在 Map 中。

在 ModelAndView 类中定义的 View 既可以表示一个 View 的名称, 也可以表示一个实现 View 接口的类, View 接口主要用来为请求做准备, 并将请求结果传递给视图层。View.java 的示例代码如下:

```
//*****View.java*****  
public interface View {  
    String getContentType();  
    void render(Map model, HttpServletRequest request, HttpServletResponse response) throws  
Exception;  
}
```

代码说明: render(Map model, HttpServletRequest request, HttpServletResponse response), 通过它将结果传递给视图层。

Spring 实现了很多 View 接口的实现类, 分别表示不同的视图层处理, 比如 JSP、velocity、xslt、tiles、freemarker、jasperreports 和 Jstl 等。由此也可以看出 Spring 提供了对众多视图层实现的支持, 真正做到了与视图层的具体实现无关。InternalResourceView.java 就是表示使用的是 JSP/Servlet。

InternalResourceView 是一个常用的视图, 表示用来处理 Servlet 和 JSP, JstlView 继承于它, 表示用来处理 Jstl。

8.3.2 Jstl 简介

Jstl 是 JSP 的标准标记库, 英文全称是 JSP Standard Tag Library。Jstl 是实现 Web 应用程序中通用功能的定制标记库集, 这些功能包括条件判断、循环、XML 操作以及数据库访

问等。

Jstl 实现大量服务器端 Java 应用程序常用的基本功能。通过为视图层的条件判断、循环等提供标准实现, Jstl 使 JSP 开发人员可以专注于特定于应用程序的开发需求。通常开发人员在 JSP 页面中是这样来实现条件判断的, 示例代码如下:

```
<% if ("gd".equals(compId)) { %>
    <p>欢迎您来到 gd 公司</p>
<% } else if ("gf".equals(compId)) { %>
    <p>欢迎您来到 gf 公司</p>
<% } %>
```

这样导致在页面中出现了很多逻辑代码, 如果页面交给别人或专门负责页面设计的人员来维护, 将会非常麻烦, 而通过将常用功能封装到定制标记库的标准集合中, Jstl 使 JSP 开发人员减少了页面的负责程度。

Jstl 1.0 于 2002 年 6 月发布, 包括 4 个定制标记库 (core、format、XML 和 SQL) 和一对通用标记库验证器 (ScriptFreeTLV 和 PermittedTaglibsTLV)。

(1) core 标记库提供了定制操作。

(2) format 标记库定义了用来格式化数字和日期的操作, 它还可以使用本地化资源对 JSP 页面进行国际化支持。

(3) XML 标记库用来操作通过 XML 表示的数据。

(4) SQL 标记库定义了用来查询关系数据库的操作。

(5) ScriptFreeTLV 验证器可以在 JSP 页面中禁止使用各种类型的 JSP 脚本元素: scriptlet、表达式等。

(6) PermittedTaglibsTLV 验证器可以用来限制由应用程序的 JSP 页面访问的定制标记库集。

使用 Jstl 有一些特定的语法, 说明 EL 表达式定界符的 Jstl 操作, 示例代码如下:

```
<c:out value="${msg}"/>
```

而在以前可能要这么编写 PGN 代码:

```
<%=request.getParameter("msg")%>
```

EL 还包括了几个用来操作和比较 EL 表达式所访问数据的运算符:

(1) 算术运算符: +、-、*、/、%。

(2) 关系运算符: ==、!=、< (或 lt)、> (或 gt)、<= (或 le) 和 >= (或 ge)。

(3) 逻辑运算符: && (或 and)、|| (或 or) 和 ! (或 not)。

(4) 验证运算符: empty。

利用运算符的 EL 表达式的示例代码如下:

```
${(i >= 1) && (i <= 5)}
```

用于 Jstl core 库 EL 版本的 taglib 伪指令的示例代码如下:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

实际上，对应于 Jstlcore 库的 taglib 伪指令有两种，因为在 Jstl 1.0 中，EL 是可选的。所有 4 个 Jstl 1.0 定制标记库都有使用 JSP 表达式指定动态属性值的备用版本。因为这些备用库依赖于 JSP 的更传统的请求时属性值，所以它们被称为 RT 库，而那些使用表达式语言的则被称为 EL 库。开发人员用不同的 taglib 伪指令来区分每个库的这两个版本。

用于 Jstl core 库 RT 版本的 taglib 伪指令：

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
```

<c:out>的示例代码如下：

```
<c:out value="expression" default="expression" escapeXml="boolean"/>
```

通过提供大多数 Web 应用程序常用功能的标准实现，Jstl 有助于加速开发周期。与 EL 结合起来，Jstl 可以不需要对表示层程序编写代码，从而极大地简化了 JSP 应用程序的维护。

8.3.3 视图解析

上面讲到，Spring 提供了很多 View 接口的实现类，真正做到了与视图层的具体实现无关。其实这个技术是通过视图解析器来实现的，在 Spring 中就是接口 ViewResolver，它提供了视图名和实际视图之间的映射。ViewResolver.java 的示例代码如下：

```
//***** ViewResolver.java*****  
public interface ViewResolver {  
    View resolveViewName(String viewName, Locale locale) throws Exception;  
}
```

代码说明：View resolveViewName(String viewName, Locale locale)，负责解析视图名称。

InternalResourceViewResolver 是常用的一个视图解析器。在 Spring 的配置文档中通过 ViewClass 来指定具体的视图类。

📌说明：更详细的有关视图解析器的内容可以参看 Spring 提供的文档和研究 Spring 的源代码。

8.4 Spring MVC 的控制器（Controller）

控制器负责建立模型层和视图层之间的联系，Spring 对控制器进行了精心的设计，从而使得开发人员既可以使用 Spring 提供的控制器，也可以自己很方便地实现控制器。这里首先对 Controller 的整体架构进行说明，并给出 Controller 的框架图，然后举几个控制器的实例来进行说明。

8.4.1 Controller 架构

Spring 首先定义了一个接口 `org.springframework.web.servlet.mvc.Controller`，所有 Spring 提供的控制器和开发人员自己实现的控制器都必须实现这个接口。`Controller.java` 的示例代码如下：

```
//***** Controller.java*****
public interface Controller {
    //所有的控制器实现类都必须实现这个方法
    ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
    throws Exception;
}
```

代码说明：`ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)`，所有的控制器实现类都必须实现这个方法，把 `HttpServletRequest`、`HttpServletResponse` 当作参数传入，经模型层处理后，返回一个 `ModelAndView`。

Controller 架构图如图 8.10 所示。

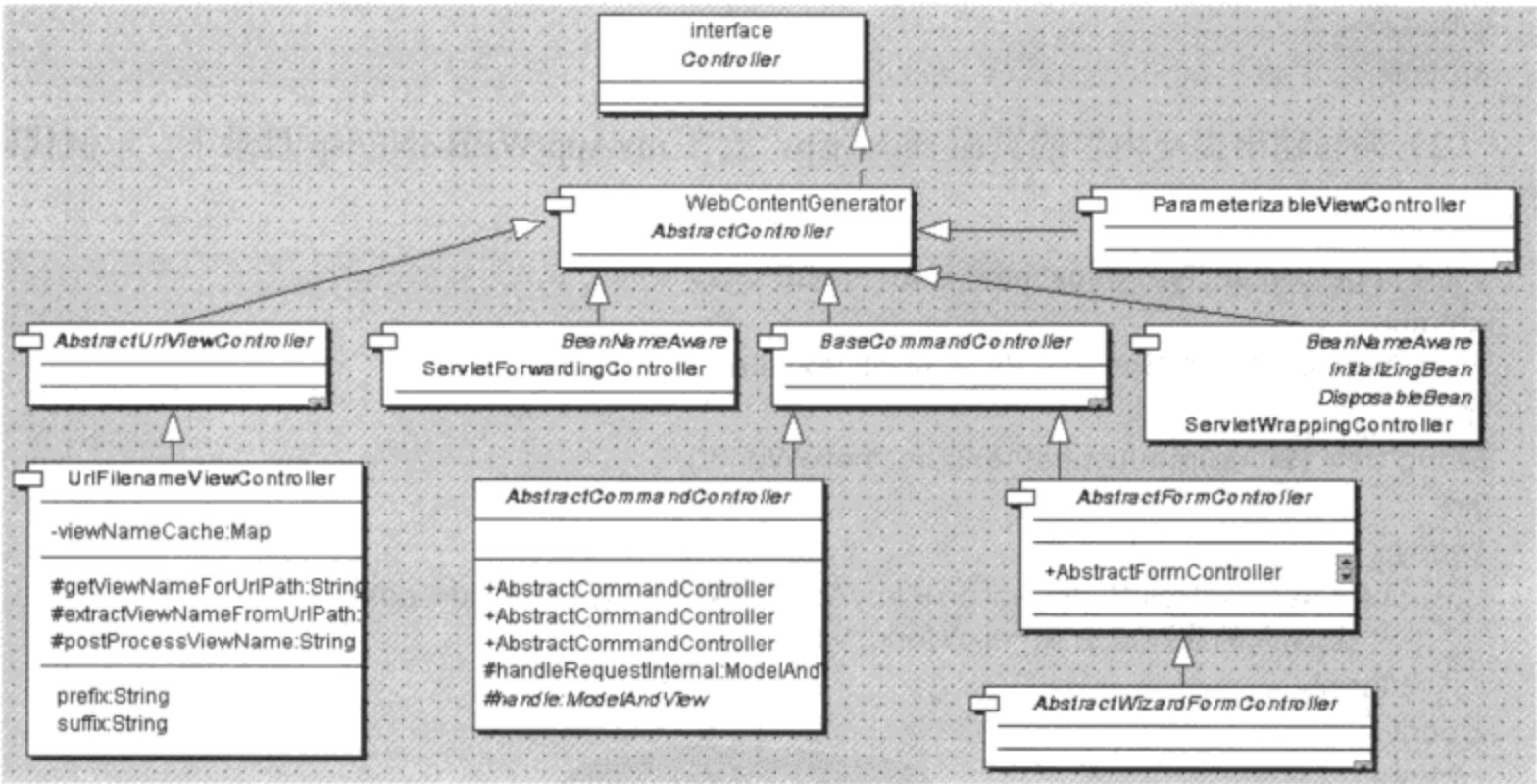


图 8.10 Controller 架构图

下面仅以表单控制器和多动作控制器进行说明，因为这两个控制器在实际编程中比较常用。

8.4.2 表单控制器（SimpleFormController）

如果每次传入参数时都使用 `HttpServletRequest` 获取页面元素的值，假如要获取页面的值很少时，还可以这样做，如果要获取的页面元素很多，那将是一件很痛苦的事情。因为

要写很多的 `HttpServletRequest.getParameter()`，为了解决这个问题，Spring 提供了表单控制器 `SimpleFormController`，把页面中 form 中的元素名称设定为和 Bean 中的一样，当传入的时候 Spring 会自动抓取 form 中和 Bean 名称一样的元素值，把它转换成一个 Bean，使得开发人员可以很方便地使用。

下面的示例就是控制器通过 `SimpleFormController` 来实现，读者可以从中看出它的方便性。实现思路是：首先编写提交 form 的页面和输出提交内容的页面，编写一个用来存放提交内容的 Bean，然后修改配置文档 `dispatcherServlet-servlet.xml`，接着修改控制器代码，使其继承 `SimpleFormController`，最后运行演示程序。具体步骤如下：

(1) 编写提交 form 的页面 `input.jsp`，放在 `myApp` 目录下。示例代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>第二个 Spring MVC 实例</title></head>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        欢迎语 <input type="text" name="msg" value=""/><br>
        <input type="submit" value="提交"/>
    </form>
</body>
</html>
```

(2) 编写输出提交内容的页面 `show.jsp`，放在 `myApp/WEB-INF/jsp` 目录下。示例代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>第二个 Spring MVC 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        您输入的欢迎语是"${helloWorld}"<br>
    </form>
</body>
</html>
```

(3) 在 `com.gc.action` 包中，编写用来存放提交内容的类 `HelloWorld`。`HelloWorld.java` 的示例代码如下：

```
//***** HelloWorld.java*****
package com.gc.action;
public class HelloWorld {
    //定义变量 msg
    private String msg = null;
    //设定 msg
    public void setMsg(String msg) {
```

```

        this.msg = msg;
    }
    //获得 msg
    public String getMsg() {
        return this.msg;
    }
}

```

(4) 修改配置文档 dispatcherServlet-servlet.xml, 示例代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    <!-- 定义映射 -->
    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
        <property name="mappings">
            <props>
                <prop key="hellWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!-- 定义视图及 JSP 存放路径 -->
    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
        <!-- 定义 JSP 存放的路径 -->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <!-- 定义 JSP 页面后缀 -->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!-- 定义控制器 -->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
        <property name="commandClass">
            <value>com.gc.action.HelloWorld</value>
        </property>
        <property name="viewPage">
            <value>show</value>
        </property>
    </bean>
</beans>

```

(5) 修改控制器代码, 使其继承 SimpleFormController。HelloWorldAction.java 的示例

代码如下：

```
/****** HelloWorldAction.java*****  
package com.gc.action;  
  
import java.io.IOException;  
import java.util.HashMap;  
import java.util.Map;  
  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
import org.apache.log4j.Logger;  
import org.springframework.web.bind.RequestUtils;  
import org.springframework.web.servlet.ModelAndView;  
import org.springframework.web.servlet.mvc.Controller;  
import org.springframework.web.servlet.mvc.SimpleFormController;  
//继承 SimpleFormController  
public class HelloWorldAction extends SimpleFormController {  
    private Logger logger = Logger.getLogger(this.getClass().getName());  
    private String viewPage;  
    //覆写 onSubmit 方法  
    public ModelAndView onSubmit(Object command) throws Exception {  
        HelloWorld helloWorld = (HelloWorld)command;  
        Map model = new HashMap();  
        model.put("helloWorld", helloWorld.getMsg());  
        //返回依赖注入定义的页面  
        return new ModelAndView(getViewPage(), model);  
    }  
    //设定 viewPage  
    public void setViewPage(String viewPage) {  
        this.viewPage = viewPage;  
    }  
    //获取 viewPage  
    public String getViewPage() {  
        return viewPage;  
    }  
}
```

(6) 运行演示程序，在浏览器的地址栏中输入 `http://localhost:8080/myApp/input.jsp`，按 Enter 键即可看到在浏览器中显示用来输入问候语的页面，如图 8.11 所示。

(7) 在页面的文本框中输入“HelloWorld”，然后单击“提交”按钮，即可看到输入的“HelloWorld”显示在页面上，如图 8.12 所示。

(8) 因为使用 Log4j 输出日志，也可以看到 Eclipse 的控制台输出了这个过程中 Spring 所做的相关动作信息：首先创建一个 HelloWorld 实例，然后执行 HelloWorldAction 类里的 `onSubmit()` 方法，如图 8.13 所示。



图 8.11 用来输入问候语的页面



图 8.12 输入的“HelloWorld”显示在页面上

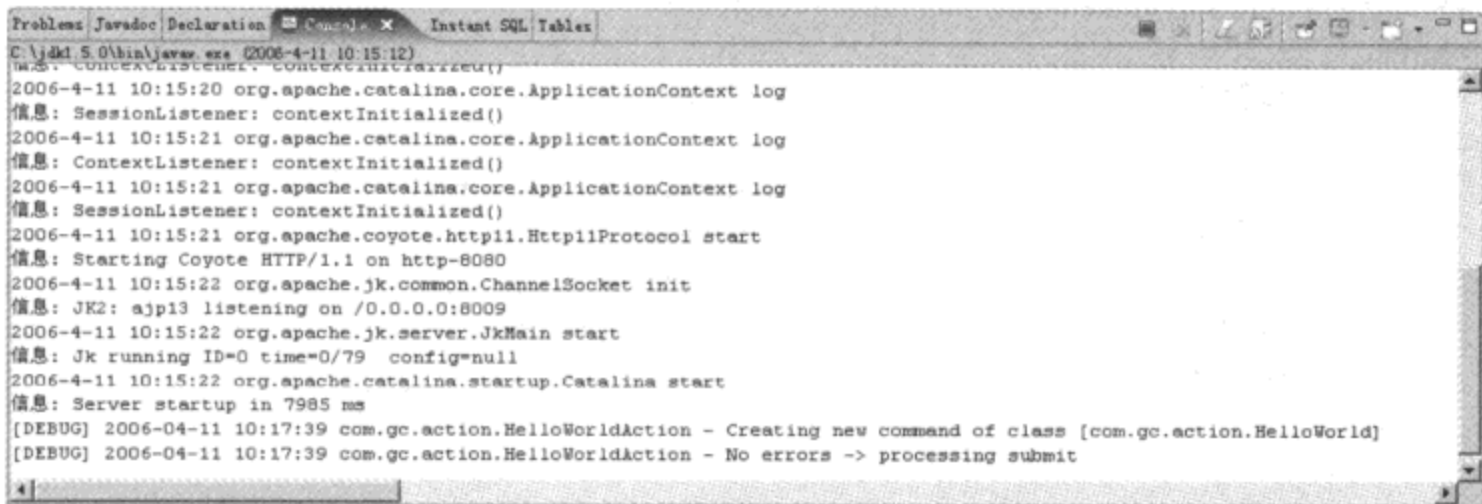



图 8.13 Spring 所做的相关动作信息

(9) 在 SimpleFormController 里,除了提供上面示例使用的 onSubmit(Object command) 方法外,还提供了以下方法给开发人员使用,示例代码如下:

```
protected ModelAndView onSubmit(  
    HttpServletRequest request, HttpServletResponse response, Object command,  
    BindException errors);  
protected ModelAndView onSubmit(Object command, BindException errors);
```

说明: 在上面的示例中,用来输入问候语的页面 input.jsp 放在了 myApp 目录下,而显示问候语的页面 show.jsp 放在了 myApp/WEB-INF/jsp 目录下,是因为放在 myApp/WEB-INF/jsp 下更加安全,即不能直接通过 *.jsp 来访问页面,但是这样开发人员就必须多编写一个控制器用来访问页面,为了简单起见,就把 input.jsp 放在了 myApp 目录下,这样可以直接通过 *.jsp 来访问页面。

8.4.3 多动作控制器 (MultiActionController)

在以上的示例中,每个控制器只能对应页面中的一个按钮,这样对于一个大项目来说,将会产生很多的控制器,不利于管理, Spring 提供了一个多动作控制器 MultiActionController,开发人员可以把处理一个业务逻辑类似的动作,比如对用户进行查询、新增、修改、删除等,放在这个控制器中,从而减少控制器的数量。MultiActionController 的实现方式有两种:一种是继承 MultiActionController,一种是在配置文档中定义一个代理 Bean,由它来定义哪个控制器是多动作的。

下面在前面示例中增加一个新增、修改、删除按钮，来展示 `MultiActionController` 的两种实现方法。实现思路是：首先修改页面，增加新增、修改、删除按钮，输出页面还使用前面示例的，然后修改配置文档，接着修改控制器，使其继承 `MultiActionController`，测试该程序，最后改为另一种方式演示，即控制器不继承 `MultiActionController`，改为在配置文档中通过配置来实现。具体步骤如下：

(1) 编写有新增、修改、删除按钮的页面 `input.jsp`，放在 `myApp` 目录下，示例代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>第二个 Spring MVC 实例</title></head>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        欢迎语 <input type="text" name="msg" value=""/><br>
        <input type="submit" name="method" value="insert"/>
        <input type="submit" name="method" value="update"/>
        <input type="submit" name="method" value="delete"/>
    </form>
</body>
</html>
```

(2) 输出提交内容的页面 `show.jsp`，还使用前面示例的，放在 `myApp/WEB-INF/jsp` 目录下，示例代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>第二个 Spring MVC 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        您输入的欢迎语是"${helloWorld}"<br>
    </form>
</body>
</html>
```

(3) 修改配置文档 `dispatcherServlet-servlet.xml`，示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义映射 -->
    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="hellWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
</beans>
```

```

        </props>
    </property>
</bean>
<!--定义视图及 JSP 文件存放路径-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.InternalResourceView</value>
    </property>
    <!--定义 JSP 存放的路径-->
    <property name="prefix">
        <value>/WEB-INF/jsp/</value>
    </property>
    <!--定义 JSP 页面后缀-->
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
<!--定义控制器-->
<bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="methodNameResolver">
        <ref bean="paraMethodResolver"/>
    </property>
    <property name="viewPage">
        <value>show</value>
    </property>
</bean>
<!--定义多动作参数-->
<bean id="paraMethodResolver" class="org.springframework.web.servlet.mvc.multiaction.
ParameterMethodNameResolver">
    <property name="paramName"><value>method</value></property>
</bean>
</beans>

```

(4) 修改控制器代码，使其继承 MultiActionController。HelloWorldAction.java 的示例代码如下：

```

//***** HelloWorldAction.java *****
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.web.bind.RequestUtils;

```

```
import org.springframework.web.bind.ServletRequestBindingException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.springframework.web.servlet.mvc.multiaction.MultiActionController;
//多动作控制器继承 MultiActionController
public class HelloWorldAction extends MultiActionController{
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private String viewPage;
    //新增动作
    public ModelAndView insert(HttpServletRequest req, HttpServletResponse res) throws
    ServletRequestBindingException {
        String helloWorld = RequestUtils.getRequiredStringParameter(req, "msg");
        Map model = new HashMap();
        model.put("helloWorld", "insert:" + helloWorld);
        return new ModelAndView(getViewPage(), model);
    }
    //修改动作
    public ModelAndView update(HttpServletRequest req, HttpServletResponse res) throws
    ServletRequestBindingException {
        String helloWorld = RequestUtils.getRequiredStringParameter(req, "msg");
        Map model = new HashMap();
        model.put("helloWorld", "update:" + helloWorld);
        return new ModelAndView(getViewPage(), model);
    }
    //删除动作
    public ModelAndView delete(HttpServletRequest req, HttpServletResponse res) throws
    ServletRequestBindingException {
        String helloWorld = RequestUtils.getRequiredStringParameter(req, "msg");
        Map model = new HashMap();
        model.put("helloWorld", "delete:" + helloWorld);
        return new ModelAndView(getViewPage(), model);
    }
    //依赖注入返回页面
    public void setViewPage(String viewPage) {
        this.viewPage = viewPage;
    }

    public String getViewPage() {
        return viewPage;
    }
}
```

代码说明: 为了区分执行的是 insert、update 和 delete 方法, 分别在问候语前加上“insert:”、“update:”、“delete:”。

(5) 运行演示程序, 在浏览器的地址栏中输入 `http://localhost:8080/myApp/input.jsp`, 按 Enter 键即可看到包含有 insert、update 和 delete 这 3 个按钮的页面, 如图 8.14 所示。

(6) 在页面的文本框中输入“HelloWorld”, 然后单击 insert 按钮, 即可看到“insert:HelloWorld”显示在页面上, 如图 8.15 所示。



图 8.14 用来输入问候语的页面



图 8.15 “insert:HelloWorld” 显示在页面上

(7) 同样单击 update 或 delete 按钮，也可看到 “update:HelloWorld” 或 “delete:HelloWorld” 显示在页面上，分别如图 8.16 和图 8.17 所示。



图 8.16 “update:HelloWorld” 显示在页面上



图 8.17 “delete:HelloWorld” 显示在页面上

(8) 另一种方式演示，即控制器不继承 MultiActionController，改为在配置文档中通过配置来实现。首先更改控制器代码，不继承 MultiActionController。HelloWorldAction.java 的示例代码如下：

```
//***** HelloWorldAction.java*****
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.web.bind.RequestUtils;
import org.springframework.web.bind.ServletRequestBindingException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
//另一种实现多动作的方式，控制器不继承任何类
public class HelloWorldAction {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private String viewPage;
```



```

//新增动作
public ModelAndView insert(HttpServletRequest req, HttpServletResponse res) throws
ServletRequestBindingException {
    String helloWorld = RequestUtils.getRequiredStringParameter(req, "msg");
    Map model = new HashMap();
    model.put("helloWorld", "insert:" + helloWorld);
    return new ModelAndView(getViewPage(), model);
}
//修改动作
public ModelAndView update(HttpServletRequest req, HttpServletResponse res) throws
ServletRequestBindingException {
    String helloWorld = RequestUtils.getRequiredStringParameter(req, "msg");
    Map model = new HashMap();
    model.put("helloWorld", "update:" + helloWorld);
    return new ModelAndView(getViewPage(), model);
}
//删除动作
public ModelAndView delete(HttpServletRequest req, HttpServletResponse res) throws
ServletRequestBindingException {
    String helloWorld = RequestUtils.getRequiredStringParameter(req, "msg");
    Map model = new HashMap();
    model.put("helloWorld", "delete:" + helloWorld);
    return new ModelAndView(getViewPage(), model);
}
//依赖注入返回页面
public void setViewPage(String viewPage) {
    this.viewPage = viewPage;
}

public String getViewPage() {
    return viewPage;
}
}

```

(9) 修改配置文档 dispatcherServlet-servlet.xml, 示例代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    <!--定义映射-->
    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
        <property name="mappings">
            <props>
                <prop key="hellWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!--定义视图及页面路径-->

```

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.InternalResourceView</value>
    </property>
    <!-- 定义 JSP 存放的路径-->
    <property name="prefix">
        <value>/WEB-INF/jsp/</value>
    </property>
    <!-- 定义 JSP 页面后缀-->
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
<!-- 定义多动作控制器-->
<bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="viewPage">
        <value>show</value>
    </property>
</bean>
<!-- 定义多动作控制器的参数-->
<bean id="paraMethodResolver" class="org.springframework.web.servlet.mvc.multiaction.
ParameterMethodNameResolver">
    <property name="paramName"><value>method</value></property>
</bean>
<!-- 使用多动作控制器代理类-->
<bean id="action" class="org.springframework.web.servlet.mvc.multiaction.MultiActionController">
    <property name="methodNameResolver">
        <ref bean="paraMethodResolver"/>
    </property>
    <property name="delegate">
        <ref bean="helloWorldAction"/>
    </property>
</bean>
</beans>
```

(10) 运行演示程序，在浏览器的地址栏中输入 `http://localhost:8080/myApp/input.jsp`，在页面的文本框中输入“HelloWorld”，然后单击 insert 按钮，即可看到“insert:HelloWorld”显示在页面上，如图 8.18 所示。



图 8.18 “insert:HelloWorld”显示在页面上

(11) 同样单击 update 或 delete 按钮，也可看到“update:HelloWorld”或“delete:HelloWorld”显示在页面上，分别如图 8.19 和图 8.20 所示。



图 8.19 “update:HelloWorld”显示在页面上



图 8.20 “delete:HelloWorld”显示在页面上

上面两种实现多动作控制器的方法，从运行结果来看是一样的。

8.5 Spring MVC 的分发器（DispatcherServlet）

大部分的 Web 框架都是请求驱动的，Spring 也不例外，其设计围绕一个能将请求分发到控制器的 Servlet，即 DispatcherServlet。和其他框架不一样的地方是，这个 Servlet 在 Spring 中是通过配置文档来定义的，开发人员一般不会直接使用它，而且更重要的是，它可以使用 Spring 的其他功能。

8.5.1 分发器的定义方式

在 Spring 中，DispatcherServlet 定义在 web.xml 中。web.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <!--定义所有以 do 结尾的请求都由 DispatcherServlet 来处理-->
    <servlet-mapping>
        <servlet-name> dispatcherServlet </servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    .....
</web-app>
```

在 web.xml 中，我们定义了一个 DispatcherServlet 的实例 dispatcherServlet，所有以“.do”结尾的请求都会由 DispatcherServlet 来处理。当然也可以把“*.do”改为“*.form”，或者

其他字符, 这样所有以“.form”结尾或其他字符结尾的请求都会由 DispatcherServlet 来处理。改为“*.form”的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <!--定义所有以 form 结尾的请求都由 DispatcherServlet 来处理-->
    <servlet-mapping>
        <servlet-name> dispatcherServlet </servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    .....
</web-app>
```

8.5.2 分发器的初始化参数

除了在 web.xml 中定义所有以“.do”结尾的请求都会由 DispatcherServlet 来处理外, 开发人员还可以在 web.xml 中定义一些 DispatcherServlet 的初始化参数。

(1) contextConfigLocation, 用来指定 Spring 配置文档的位置。示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!--初始化参数-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/ha-servlet.xml</param-value>
        </init-param>
    </servlet>
    <!--定义所有以 do 结尾的请求都由 DispatcherServlet 来处理-->
    <servlet-mapping>
        <servlet-name> dispatcherServlet </servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    .....
</web-app>
```


代码说明：<param-value>/WEB-INF/ha-servlet.xml</param-value>，指定 Spring 的配置文档是 WEB-INF 目录下的 ha-servlet.xml。

当然也可以指定多个 Spring 配置文档的位置，但要以逗号分隔。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!--初始化参数-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/ha-servlet.xml, /WEB-INF/hb-servlet.xml </param-value>
        </init-param>
    </servlet>
    <!--定义所有以 form 结尾的请求都由 DispatcherServlet 来处理-->
    <servlet-mapping>
        <servlet-name> dispatcherServlet </servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    .....
</web-app>
```

代码说明：<param-value>/WEB-INF/ha-servlet.xml, /WEB-INF/hb-servlet.xml </param-value>，指定 Spring 的配置文档是 WEB-INF 目录下的 ha-servlet.xml 和 hb-servlet.xml。

🔔说明：分隔符逗号是英文的逗号，不要用中文的逗号。

(2) namespace，Spring 默认为 servlet 名+“-servlet.xml”，并放在 WEB-INF 目录下，比如，前面的示例如果不指定 contextConfigLocation，则默认的就应该是放在 WEB-INF 目录下的 dispatcherServlet-servlet.xml。

8.5.3 分发器的工作流程

当 DispatcherServlet 配置好后，一旦 DispatcherServlet 接收到请求，DispatcherServlet 就开始处理请求了。研究 DispatcherServlet 的源代码，就会发现具体处理的步骤如下：

(1) 首先搜索 WebApplicationContext，并将它绑定到请求的一个属性上，以便控制器能够使用 WebApplicationContext。

(2) 接着会绑定本地化的信息、主题信息等信息。

(3) 然后会搜索合适的处理器，并准备 ModelAndView。

(4) 业务逻辑处理完毕后，根据 WebApplicationContext 中绑定的视图信息显示对应的

视图。

8.5.4 分发器与视图解析器的结合

在前面的示例中，只指定 id 为 viewResolver 的 Bean 的 viewClass 属性，所以还需要在配置文档中 id 为 helloUserAction 的 Bean 的 viewPage 属性中指定要返回页面的绝对路径。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    .....
    <!--定义视图技术-->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
    </bean>
    <!--定义控制器-->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
        <property name="helloWord">
            <value>H helloWord </value>
        </property>
        <property name="viewPage">
            <value>/WEB-INF/jsp/index.jsp</value>
        </property>
    </bean>
    .....
</beans>
```

其实 Spring 提供了更为方便的方法，不需要指定返回页面的绝对路径，而只需要指定 id 为 viewResolver 的 Bean 的 prefix 和 suffix 属性，然后在 id 为 helloUserAction 的 Bean 的 viewPage 属性中只需指定要返回页面的名称即可。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    .....
    <!--定义视图及路径-->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
```

```
<!--定义 JSP 存放的路径-->
<property name="prefix">
    <value>/WEB-INF/jsp/</value>
</property>
<!--定义 JSP 页面后缀-->
<property name="suffix">
    <value>.jsp</value>
</property>
</bean>
<!--定义控制器-->
<bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="helloWord">
        <value>HelloWorld</value>
    </property>
    <property name="viewPage">
        <value>index</value>
    </property>
</bean>
.....
</beans>
```

代码说明:

- ☐ prefix 和 suffix 属性, 指明系统要处理页面的目录和后缀, 这里指的是处理 WEB-INF/jsp 目录下以 jsp 为后缀的文件。
- ☐ viewPage 属性, 只需指定页面的名称即可。

8.5.5 在一个 Web 应用中使用不同的视图层技术

上面的示例中, 在视图层使用的只是 Jsp/Servlet 技术, 所以使用的 View 实现类是 InternalResourceView, 解析器是 InternalResourceViewResolver, 如果在一个项目中要求使用不同的视图层技术, 那就需要使用 Spring 提供的 ResourceBundleViewResolver 解析器。具体示例如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    .....
    <!--定义使用不同的视图技术-->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
        <property name="basename">
            <value>views</value>
        </property>
    </bean>
    <!--定义控制器-->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
```

```
<property name="helloWord">
    <value>HelloWorld</value>
</property>
<!--指定使用的是 JSP/Servlet 技术-->
<property name="viewPage1">
    <value>index1</value>
</property>
<!--指定使用的是 Jstl 技术-->
<property name="viewPage2">
    <value>index2</value>
</property>
<!--指定使用的是 Velocity 技术-->
<property name="viewPage3">
    <value>index3</value>
</property>
</bean>
.....
</beans>
```

然后在 views.properties 指定使用的视图层技术。示例代码如下：

```
index1.class=org.springframework.web.servlet.view.InternalResourceView
index1.url=/WEB-INF/jsp/ index1.jsp
index2.class=org.springframework.web.servlet.view.JstlView
index2.url=/WEB-INF/jsp/ index2.jsp
index3.class=org.springframework.web.servlet.view.VelocityView
index3.url=/WEB-INF/vm/ index3.vm
```

代码说明：

- ☐ index1.class, 指定使用的是 JSP/Servlet 技术。
- ☐ index2.class, 指定使用的是 Jstl 技术。
- ☐ index3.class, 指定使用的是 Velocity 技术。

8.5.6 在 DispatcherServlet 中指定处理异常的页面

在前面 AOP 中，笔者实现了一个利用 AOP 实现异常输出的实例，但那是输出到日志中，如果开发人员想把异常信息集中显示在页面中，就可以通过在 DispatcherServlet 中指定处理异常的页面来实现。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    .....
    <!--定义视图及路径-->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
```



```
        </property>
        <!--定义 JSP 存放的路径-->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <!--定义 JSP 页面后缀-->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!--定义异常处理页面-->
    <bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMapping-
ExceptionResolver">
        <property name="exceptionMappings">
            <props>
                <prop key="java.sql.SQLException">outException</prop>
                <prop key="java.sql.IOException">outException</prop>
            </props>
        </property>
    </bean>
    .....
</beans>
```

代码说明：只要发生了 SQLException 异常或 IOException 异常，就会连接至 /WEB-INF/jsp/outException.jsp。

outException.jsp 的示例代码如下：

```
<html>
<head><title>抛送异常页面</title></head>
<body>
<% Exception ex = (Exception)request.getAttribute("Exception"); %>
<H2>Exception: <%=ex.getMessage();%></H2>
</body>
</html>
```

8.6 处理器映射

前面分别讲解了 MVC 模式的 3 个方面之间的关系，又讲解了 Spring 的分发器用来定义什么样的请求可以被接收到控制器处理，具体的某个请求会被某个具体的控制器处理，还需要通过处理器映射来进行。

8.6.1 映射示例

例如前面示例中的配置文档，示例代码如下：

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

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    .....
    <!-- 定义映射 -->
    <bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/helloWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!-- 定义视图及路径 -->
    <bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
    <!-- 定义 JSP 存放的路径 -->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
    <!-- 定义 JSP 页面后缀 -->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!-- 定义控制器 -->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
        <property name="helloWord">
            <value>HelloWorld</value>
        </property>
        <!-- 指定返回的页面 -->
        <property name="viewPage">
            <value>index</value>
        </property>
    </bean>
    .....
</beans>
```

代码说明:

- ❑ <prop key="/helloWorld.do">helloWorldAction</prop>, 就是表示对于/helloWorld.do 的请求会转向处理器 helloWorldAction。
- ❑ org.springframework.web.servlet.handler.SimpleUrlHandlerMapping, 就是用来具体负责请求转换的类, 它实现了 HandlerMapping 接口。

8.6.2 映射原理

HandlerMapping 接口的示例代码如下：

```
public interface HandlerMapping {
    HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
}
```

而要实现把对*.do 的请求转换给配置文档中设定的相应处理器，则依赖于 org.springframework.web.servlet.DispatcherServlet 中的 getHandler()方法，示例代码如下：

```
protected HandlerExecutionChain getHandler(HttpServletRequest request, boolean cache) throws
Exception {
    // HANDLER_EXECUTION_CHAIN_ATTRIBUTE 在 DispatcherServlet 类中被定义为
    static final String, 其值为 DispatcherServlet.class.getName() + ".HANDLER"
    HandlerExecutionChain handler =
        (HandlerExecutionChain)
request.getAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE);
    //假如获取的 handler 不为 null
    if (handler != null) {
        //再判断 cache 是否为 false
        if (!cache) {
            request.removeAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE);
        }
        return handler;
    }
    //获取 handler
    Iterator it = this.handlerMappings.iterator();
    while (it.hasNext()) {
        HandlerMapping hm = (HandlerMapping) it.next();
        //调试时使用的日志信息
        if (logger.isDebugEnabled()) {
            logger.debug("Testing handler map [" + hm + "] in DispatcherServlet with
name "" +
                                getServletName() + "");
        }
        //通过 HandlerMapping 的 getHandler()方法获取 handler
        handler = hm.getHandler(request);
        //假如获取的 handler 不为 null
        if (handler != null) {
            if (cache) {
                request.setAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE,
handler);
            }
            return handler;
        }
    }
    return null;
}
```


8.6.3 添加拦截器

正因为有了处理器映射，所以开发人员可以在映射请求时添加一个拦截器，以做某种动作或检查。要在映射请求时添加一个拦截器，自定义的拦截器就必须实现 `org.springframework.web.servlet.HandlerInterceptor` 接口。

`HandlerInterceptor.java` 的示例代码如下：

```
//***** HandlerInterceptor.java*****  
public interface HandlerInterceptor {  
    //在处理器执行前被调用  
    boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object  
handler) throws Exception;  
    //在处理器执行后被调用  
    void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  
ModelAndView modelAndView) throws Exception;  
    //在整个请求处理完后调用  
    void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object  
handler, Exception ex) throws Exception;  
}
```

代码说明：`preHandle()`方法的返回值是 `boolean`。当返回 `true` 时，处理器将继续执行，当返回 `false` 时，`DispatcherServlet` 认为拦截器本身将处理请求，而不继续执行处理器。

假如企业需要用户只在正常上班时间之外才能在企业的 BBS 上留言，则就可以使用这个功能。假定正常上班时间是早上 8 点到下午 5 点，如果员工登录 BBS，则自动转向内网首页，即不允许在这段时间内留言。实现思路是：编写一个拦截器，实现 `HandlerInterceptor` 接口，然后配置 Spring 文档，在处理器映射中添加拦截器。具体实现步骤如下：

(1) 编写一个拦截器 `NotLeaveWord`，实现 `HandlerInterceptor` 接口。`NotLeaveWord` 的示例代码如下：

```
//***** NotLeaveWord.java*****  
package com.gc.action;  
import java.util.Calendar;  
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;  
//实现 HandlerInterceptor 接口  
public class NotLeaveWord extends HandlerInterceptorAdapter{  
    //在执行具体转发前截取  
    public boolean preHandle(  
        HttpServletRequest request,  
        HttpServletResponse response,  
        Object handler)  
        throws Exception {  
        //获取当前时间  
        Calendar cal = Calendar.getInstance();  
        int hour = cal.get(HOUR_OF_DAY);
```



```

//假如在 8 点到 17 点之间，则转向内网
if (8 <= hour < 17) {
    response.sendRedirect("http://localhost:8080/jlerp/index.jsp");
    return false;
} else {
    return true;
}
}
}

```

(2) 配置 Spring 文档，在处理器映射中添加拦截器。配置文档的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    .....
    <!-- 定义映射 -->
    <bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <!-- 定义拦截器 -->
        <property name="interceptors">
            <list>
                <ref bean="NotLeaveWord"/>
            </list>
        </property>
        <!-- 如果是 helloWorld.do 请求，则转发至 helloWorldAction 处理器 -->
        <property name="mappings">
            <props>
                <prop key="/helloWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!-- 定义视图及路径 -->
    <bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
        <!-- JSP 都放在 WEB-INF/jsp 目录下 -->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <!-- JSP 的后缀名都为 jsp -->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!-- 定义控制器 -->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">

```

```
<property name="helloWorld">
    <value>HelloWorld</value>
</property>
<!--定义要指向的页面-->
<property name="viewPage">
    <value>index</value>
</property>
</bean>
<bean id="NotLeaveWord"
      class="com.gc.action.NotLeaveWord">
</bean>
.....
</beans>
```

(3) 当然，开发人员可以充分利用 Spring 提供的注入功能，把时间范围设定在配置文档中，这样可以很方便地对时间点进行修改。修改 NotLeaveWord，示例代码如下：

```
//***** NotLeaveWord.java*****
package com.gc.action;
import java.util.Calendar;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;
public class NotLeaveWord extends HandlerInterceptorAdapter {
    //通过依赖注入完成
    private int startTime;
    private int endTime;
    public void setStartTime(int startTime) {
        this.startTime = startTime;
    }
    public void setEndTime(int endTime) {
        this.endTime = endTime;
    }
    //进行拦截处理
    public boolean preHandle(
        HttpServletRequest request,
        HttpServletResponse response,
        Object handler)
        throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        //假如在规定的時間之間，則轉向內網
        if (startTime <= hour < endTime) {
            response.sendRedirect("http://localhost:8080/jlerp/index.jsp");
            return false;
        } else {
            return true;
        }
    }
}
```

(4) 修改配置文档，示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    .....
    <!-- 定义映射 -->
    <bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="NotLeaveWord"/>
            </list>
        </property>
        <!-- 如果是 helloWorld.do 请求，则转发至 helloWorldAction 处理器 -->
        <property name="mappings">
            <props>
                <prop key="/helloWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!-- 定义视图及路径 -->
    <bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
        <!-- JSP 都放在 WEB-INF/jsp 目录下 -->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!-- 定义控制器 -->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
        <property name="helloWord">
            <value>HelloWorld</value>
        </property>
        <!-- 定义要指向的页面 -->
        <property name="viewPage">
            <value>index</value>
        </property>
    </bean>
    // 设定开始和结束时间
    <bean id="NotLeaveWord"
        class="com.gc.action.NotLeaveWord">
        <property name="startTime"><value>8</value></property>
        <property name="endTime"><value>17</value></property>
```



```
        </bean>
        .....
    </beans>
```

这样，如果想修改为 8 点到 12 点之间不允许在 BBS 上留言，直接修改配置文档即可，而不用修改程序代码。

8.7 数据绑定

在上面的示例中，如果要进行数据验证，则必须把验证的代码写在逻辑里，这不是一个好的做法，Spring 提供了分离业务逻辑和数据验证的方法，即数据绑定。

8.7.1 绑定原理

在 Spring 中，有两种数据绑定的方法：一种是使用自定义标签，一种是使用 Validator。

(1) 使用自定义标签，可以对页面、错误信息进行数据绑定。要使用自定义标签，需要将 spring-framework-2.0-m1/dist 目录下的 spring.tld 复制到 WEB-INF 文件夹下，并在 web.xml 中加入：

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

(2) 使用 SimpleFormController 时，可以使用 Validator，负责具体的验证工作。org.springframework.validation.Validator 的示例代码如下：

```
//***** Validator.java *****
public interface Validator {
    boolean supports(Class clazz);
    //在此方法里进行验证
    void validate(Object obj, Errors errors);
}
```

代码说明：

- ❑ supports()方法，表示是否支持对传进来的对象进行验证。如果返回值是 true，则表示支持对传进来的类进行验证；如果返回值是 false，则表示不支持对传进来的类进行验证。
- ❑ validate()方法，表示对传入的对象进行验证，如果有错，则可以把错误放在 Errors 里。

下面把前面实现 SimpleFormController 的示例分别添加自定义标签和使用 Validator，来展示数据绑定的方法。

8.7.2 使用自定义标签

使用自定义标签进行数据绑定，实现思路是：首先在 web.xml 中添加 Spring 的自定义标签库 spring.tld，修改提交 form 的页面、输出提交内容的页面，添加自定义标签的内容，用来存放提交内容的 Bean，仍然使用前面示例中的 Bean，配置文档 dispatcherServlet-servlet.xml 不改变，接着修改控制器代码，使其继承 SimpleFormController，最后运行演示程序。具体步骤如下：

(1) 在 web.xml 中添加 Spring 的自定义标签库 spring.tld，示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <!--拦截所有以 do 结尾的请求-->
    <servlet-mapping>
        <servlet-name> dispatcherServlet </servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!--定义自定义标签-->
    <taglib>
        <taglib-uri>/spring</taglib-uri>
        <taglib-location>/WEB-INF/spring.tld</taglib-location>
    </taglib>
    .....
</web-app>
```

(2) 仍然使用 myApp 目录下提交 form 的页面 input.jsp，这里只是做一个提交的动作，对提交的验证放在输出提交内容的页面来实现。示例代码如下：

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title> Spring MVC 实例</title></head>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        欢迎语 <input type="text" name="msg" value=""/><br>
        <input type="submit" value="提交"/>
    </form>
</body>
</html>
```

(3) 修改输出提交内容的页面 show.jsp，放在 myApp/WEB-INF/jsp 目录下，当从提

交页面转入这个页面后，这个页面也有提交功能，在这个页面进行数据验证。示例代码如下：

```
<%@taglib prefix="spring" uri="/spring"%>
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>Spring MVC 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        您输入的欢迎语是"${helloWorld}"<br>
        <spring:bind path="command.msg">
            输入验证<input type="text" name="${status.expression}" value="${status.value}"/><br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
    </form>
</body>
</html>
```

代码说明：<spring:bind>的 path 属性设定了要绑定窗体对象名称，这个名称是设定在 loginAction 中的 commandName 属性，预设名称是 command，当设定为 command.* 时，表示绑定窗体对象所有相关的数据，status 的 errorMessage 会显示 Controller 中设定的错误信息，status 的 expression 会显示绑定的属性名称，status 的 value 则显示窗体对象中所储存的值。

(4) 仍然使用 com.gc.action 包中的类 HelloWorld。HelloWorld.java 的示例代码如下：

```
//***** HelloWorld.java *****
package com.gc.action;
public class HelloWorld {
    //定义变量 msg
    private String msg = null;
    //定义 msg 的 set 方法
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //定义 msg 的 get 方法
    public String getMsg() {
        return this.msg;
    }
}
```

(5) 仍然使用配置文档 dispatcherServlet-servlet.xml，示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
```

```

<beans>
  <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
    <property name="mappings">
      <!--如果是 helloWorld.do 请求，则转发至 helloWorldAction 处理器-->
      <props>
        <prop key="hellWorld.do">helloWorldAction</prop>
      </props>
    </property>
  </bean>
  <!--定义视图及路径-->
  <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
      <value>org.springframework.web.servlet.view.InternalResourceView</value>
    </property>
    <!--JSP 都放在 WEB-INF/jsp 目录下-->
    <property name="prefix">
      <value>/WEB-INF/jsp/</value>
    </property>
    <!--JSP 页面的后缀都是 jsp-->
    <property name="suffix">
      <value>.jsp</value>
    </property>
  </bean>
  <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="commandClass">
      <value>com.gc.action.HelloWorld</value>
    </property>
    <!--指定要返回的页面-->
    <property name="viewPage">
      <value>show</value>
    </property>
  </bean>
</beans>

```

(6) 修改控制器代码，使其继承 SimpleFormController。HelloWorldAction.java 的示例代码如下：

```

//***** HelloWorldAction.java*****
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

```



```
import org.apache.log4j.Logger;
import org.springframework.validation.BindException;
import org.springframework.web.bind.RequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.springframework.web.servlet.mvc.SimpleFormController;
import org.springframework.web.servlet.view.RedirectView;
//该类继承 SimpleFormController
public class HelloWorldAction extends SimpleFormController {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private String viewPage;
    //覆写 onSubmit
    public ModelAndView onSubmit(Object command, BindException errors) throws Exception {
        HelloWorld helloWorld = (HelloWorld)command;
        Map model = errors.getModel();
        model.put("helloWorld", helloWorld.getMsg());
        //假如输入的问候语长度大于 10 个字符
        if (helloWorld.getMsg().length() > 10 ) {
            errors.rejectValue("msg", "", null, "问候语不能大于 10 个字符");
            return new ModelAndView(getViewPage(), model);
        } else {
            return new ModelAndView(getViewPage(), model);
        }
    }
    //依赖注入返回页面
    public void setViewPage(String viewPage) {
        this.viewPage = viewPage;
    }

    public String getViewPage() {
        return viewPage;
    }
}
```

代码说明: `rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage)`, `rejectValue()` 的 `field` 指定对象的属性, `errorCode` 指定消息文件中的 `key` 值, `errorArgs` 用于指定消息文件中的参数, 而预设消息指的是在消息文件中找不到指定的 `errorCode` 时使用。

(7) 运行演示程序, 在浏览器的地址栏中输入 `http://localhost:8080/myApp/input.jsp`, 按 Enter 键即可看到在浏览器中显示用来输入问候语的页面, 如图 8.21 所示。

(8) 在页面的文本框中输入“HelloWorld”, 单击“提交”按钮, 即可看到负责数据验证的页面, 如图 8.22 所示。

(9) 修改文本框中的字符为“HelloWorld gf”, 然后单击“提交”按钮, 即可看到输出“欢迎语不能大于 10 个字符”提示的页面, 如图 8.23 所示。

(10) 修改文本框中的字符为“HelloWorld”, 符合欢迎语小于 10 个字符, 然后单击“提交”按钮, 即可看到输出“HelloWorld”的页面, 如图 8.24 所示。



图 8.21 用来输入问候语的页面

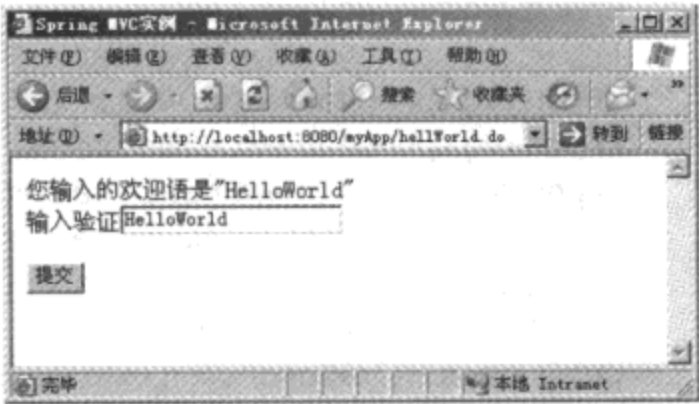


图 8.22 负责数据验证的页面



图 8.23 输出“欢迎语不能大于 10 个字符”提示的页面



图 8.24 输出“HelloWorld”的页面

8.7.3 Validator 应用

使用 Validator 进行数据绑定，实现思路是：首先提交 form 的页面、输出提交内容的页面，仍然使用前面示例中的页面；用来存放提交内容的 Bean，仍然使用原来的 Bean；编写一个实现 Validator 的类 HelloWorldValidator，用来实现数据绑定；然后修改配置文档 dispatcherServlet-servlet.xml；接着修改控制器代码，使其继承 SimpleFormController；最后运行演示程序。其具体步骤如下：

(1) 仍然使用原来提交 form 的页面 input.jsp，放在 myApp 目录下。示例代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title> Spring MVC 实例</title></head>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        欢迎语 <input type="text" name="msg" value=""/><br>
        <input type="submit" value="提交"/>
    </form>
</body>
</html>
```

(2) 仍然使用原来输出提交内容的页面 show.jsp，放在 myApp/WEB-INF/jsp 目录下。示例代码如下：

```
<%@taglib prefix="spring" uri="/spring"%>
<%@page contentType="text/html;charset=GBK"%>
```

```

<html>
<head><title>Spring MVC 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HellWorld" action="/myApp/hellWorld.do" method="post">
        您输入的欢迎语是"${helloWorld}"<br>
        <spring:bind path="command.msg">
            输入验证<input type="text" name="${status.expression}" value="${status.value}"/><br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
    </form>
</body>
</html>

```

(3) 仍然使用原来 com.gc.action 包中用来存放提交内容的类 HelloWorld。HelloWorld.java 的示例代码如下:

```

//***** HelloWorld.java*****
package com.gc.action;
public class HelloWorld {
    //定义变量 msg
    private String msg = null;
    //设定 msg
    public void setMsg(String msg) {
        this.msg = msg;
    }
    //获得 msg
    public String getMsg() {
        return this.msg;
    }
}

```

(4) 在 com.gc.action 包中, 编写用来进行数据绑定的类 HelloWorldValidator。HelloWorldValidator.java 的示例代码如下:

```

//***** HelloWorldValidator.java*****
package com.gc.action;

import org.springframework.validation.Validator;
import org.springframework.validation.Errors;
//实现 Validator 接口, 对数据进行验证
public class HelloWorldValidator implements Validator {
    public boolean supports(Class clazz) {
        return clazz.equals(HelloWorld.class);
    }
    //进行数据验证
    public void validate(Object obj, Errors errors) {

```

```

        HelloWorld helloWorld = (HelloWorld)obj;
        If (helloWorld.getMsg().length() > 10 ) {
            errors.rejectValue("msg", "", null, "问候语不能大于 10 个字符");
        }
    }
}

```

(5) 修改配置文档 dispatcherServlet-servlet.xml, 示例代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    <!-- 定义映射 -->
    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
        <property name="mappings">
            <props>
                <prop key="hellWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!-- 定义视图 -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
        <!-- JSP 都放在 WEB-INF/jsp 目录下 -->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <!-- JSP 页面的后缀都是 jsp -->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!-- 定义控制器 -->
    <bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
        <property name="commandClass">
            <value>com.gc.action.HelloWorld</value>
        </property>
        <!-- 定义进行验证的类 -->
        <property name="validator">
            <ref bean="helloWorldValidator"/>
        </property>
        <!-- 验证失败时返回的页面 -->
        <property name="formView">
            <value>show</value>
        </property>
    </bean>

```



```
</bean>
<!--定义验证类-->
<bean id="helloWorldValidator" class="com.gc.action.HelloWorldValidator"/>
</beans>
```

代码说明：注意把 id 为 helloWorldAction 的 Bean 的 viewPage 属性改为 formView 属性，这个属性是 SimpleFormController 的属性，使用 Validator 时，如果验证有错误会返回到 formView 属性设定的页面。

(6) 修改控制器代码，使其继承 SimpleFormController。HelloWorldAction.java 的示例代码如下：

```
//***** HelloWorldAction.java*****
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.web.bind.RequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.springframework.web.servlet.mvc.SimpleFormController;
//该类继承 SimpleFormController
public class HelloWorldAction extends SimpleFormController {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //覆写 onSubmit
    public ModelAndView onSubmit(Object command) throws Exception {
        HelloWorld helloWorld = (HelloWorld)command;
        Map model = new HashMap();
        model.put("helloWorld", helloWorld.getMsg());
        //带着 model 返回
        return new ModelAndView(getFormView(), model);
    }
}
```

(7) 运行演示程序，在浏览器的地址栏中输入 `http://localhost:8080/myApp/input.jsp`，按 Enter 键即可看到在浏览器中显示用来输入问候语的页面，如图 8.25 所示。

(8) 在页面的文本框中输入“HelloWorld”，单击“提交”按钮，即可看到负责数据验证的页面，如图 8.26 所示。

(9) 修改文本框中的字符为“HelloWorld gf”，然后单击“提交”按钮，即可看到输出“欢迎语不能大于 10 个字符”提示的页面，如图 8.27 所示。

(10) 修改文本框中的字符为“HelloWorld”，符合欢迎语小于 10 个字符，然后单击“提交”按钮，即可看到输出“HelloWorld”的页面，如图 8.28 所示。



图 8.25 用来输入问候语的页面



图 8.26 负责数据验证的页面



图 8.27 输出“欢迎语不能大于 10 个字符”提示的页面



图 8.28 输出“HelloWorld”的页面

由此可以看出，使用自定义标签和使用 Validator 都能达到相同的效果，只是使用 Validator 时能很清晰地把数据验证和业务逻辑分开。

说明：这里只是为了示例方便，其实这里的显示页面和提交页面是可以放在一起的，使用 `http://localhost:8080/myApp/helloWorld.do` 即可直接访问，演示示例。

8.8 本地化支持

前面讲过，Spring 提供了对本地化的支持，这主要依赖于 `org.springframework.web.servlet.LocaleResolver`。 `LocaleResolver.java` 的示例代码如下：

```
//***** LocaleResolver.java*****  
public interface LocaleResolver {  
    Locale resolveLocale(HttpServletRequest request);  
    //设定 locale  
    void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale);  
}
```

代码说明：

- ❑ `resolveLocale()`方法，用来取得 Locale 信息。
- ❑ `setLocale()`方法，用来设定 Locale 信息。

Spring 可以根据客户端浏览器语言设定, 自动切换符合客户端浏览器语言设定的信息。从而灵活地实现本地化支持。根据浏览器的语言设定, 实现本地化支持, 有 3 种方式, 分别是:

- (1) 在头信息中包含客户端的本地化信息。
- (2) 根据 cookie 获取本地化信息。
- (3) 从会话中获取本地化信息。

Spring 提供了很方便的方法来使用这 3 种方式, 只是简单地在配置文件中配置就可以了, 下面分别来进行介绍。

8.8.1 在头信息中包含客户端的本地化信息

在头信息中包含客户端的本地化信息, 使用非常简单, 只需在配置文件中增加一个 Bean 即可。示例代码如下:

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver">
</bean>
```

Spring 就会根据客户端浏览器的 Locale 设定决定返回界面所采用的语言种类。AcceptHeaderLocaleResolver 实现了 LocaleResolver 接口, 示例代码如下:

```
/** ***** AcceptHeaderLocaleResolver.java ***** */
public class AcceptHeaderLocaleResolver implements LocaleResolver {
    public Locale resolveLocale(HttpServletRequest request) {
        return request.getLocale();
    }
    //设定 Locale
    public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale) {
        throw new UnsupportedOperationException(
            "Cannot change HTTP accept header - use a different locale resolution strategy");
    }
}
```

从 AcceptHeaderLocaleResolver.java 的代码可以看出, AcceptHeaderLocaleResolver 只是实现了 resolveLocale 方法, 因此通过 AcceptHeaderLocaleResolver.resolveLocale 方法即可获得当前语言设定, 但不能对客户端的 Local 进行设定。

8.8.2 根据 cookie 获取本地化信息

根据 cookie 获取本地化信息, 使用也非常简单, 只需在配置文件中增加一个 Bean 即可。示例代码如下:

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
```

```

<!--设定 cookieName 的名字-->
<property name="cookieName">
    <value>clientlanguageLocal</value>
</property>
<property name="cookieMaxAge">
    <value>100000</value>
</property>
<!--设定 cookiePath -->
<property name="cookiePath">
    <value>myApp</value>
</property>
</bean>

```

代码说明：

- ❑ CookieLocaleResolver 配置包含了 3 个属性——cookieName、cookieMaxAge 和 cookiePath，分别指定了用于保存 Locale 设定的 cookie 的名称、最大保存时间和路径。
- ❑ cookieMaxAge 的默认值是 Integer.MAX_INT，表示 cookie 在客户端存在的最大时间。如果该值是-1，这个 cookie 一直存在，直到客户关闭浏览器。
- ❑ cookiePath 表示开发人员可以限制 cookie 只有一部分网站页面可以访问。当 cookiePath 被指定，cookie 只能被该目录以及子目录的页面访问。

CookieLocaleResolver 实现了 LocaleResolver 接口，示例代码如下：

```

//***** CookieLocaleResolver.java*****
public class CookieLocaleResolver extends CookieGenerator implements LocaleResolver {
    public static final String LOCALE_REQUEST_ATTRIBUTE_NAME = CookieLocaleResolver.
class.getName() + ".LOCALE";
    public static final String DEFAULT_COOKIE_NAME = CookieLocaleResolver.class.getName()
+ ".LOCALE";
    public CookieLocaleResolver() {
        setCookieName(DEFAULT_COOKIE_NAME);
    }
    public Locale resolveLocale(HttpServletRequest request) {
        // Check request for pre-parsed or preset locale.
        Locale locale = (Locale) request.getAttribute(LOCALE_REQUEST_ATTRIBUTE_NAME);
        if (locale != null) {
            return locale;
        }
        Cookie cookie = WebUtils.getCookie(request, getCookieName());
        if (cookie != null) {
            locale = StringUtils.parseLocaleString(cookie.getValue());
            if (logger.isDebugEnabled()) {
                logger.debug("Parsed cookie value [" + cookie.getValue() + "] into locale '" +
locale + "'");
            }
            if (locale != null) {
                request.setAttribute(LOCALE_REQUEST_ATTRIBUTE_NAME, locale);
            }
        }
    }
}

```

```

        return locale;
    }
}
// Fall back to accept-header locale.
return request.getLocale();
}
public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale
locale) {
    if (locale != null) {
        // Set request attribute and add cookie.
        request.setAttribute(LOCALE_REQUEST_ATTRIBUTE_NAME, locale);
        addCookie(response, locale.toString());
    }
    else {
        // Set request attribute to fallback locale and remove cookie.
        request.setAttribute(LOCALE_REQUEST_ATTRIBUTE_NAME,
request.getLocale());
        removeCookie(response);
    }
}
}
}

```

8.8.3 从会话中获取本地化信息

根据 cookie 获取本地化信息, 使用也非常简单, 只需在配置文件中增加一个 Bean 即可。示例代码如下:

```

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
</bean>

```

SessionLocaleResolver 实现了 LocaleResolver 接口, 示例代码如下:

```

//***** SessionLocaleResolver.java*****
public class SessionLocaleResolver implements LocaleResolver {
    public static final String LOCALE_SESSION_ATTRIBUTE_NAME = SessionLocaleResolver.
class.getName() + ".LOCALE";
    public Locale resolveLocale(HttpServletRequest request) {
        Locale locale = (Locale) WebUtils.getSessionAttribute(request, LOCALE_SESSION_
ATTRIBUTE_NAME);
        // specific locale, or fallback to accept header locale?
        return (locale != null ? locale : request.getLocale());
    }
    public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale
locale) {
        WebUtils.setSessionAttribute(request, LOCALE_SESSION_ATTRIBUTE_NAME, locale);
    }
}

```


8.8.4 根据参数修改本地化信息

除了前面 3 种很方便实现本地化信息的方法外，Spring 还提供了更加灵活的设定方法，使开发人员可以根据参数修改本地化信息，当然这要借助于拦截器的支持，也是只需要在配置文档中设定即可。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    <!--根据参数修改本地化信息-->
    <bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.Locale-
ChangeInterceptor">
        <property name="paramName"><value>language</value></property>
    </bean>
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
</bean>
    <!--定义映射-->
    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
        <property name="interceptors">
            <list>
                <ref local="localeChangeInterceptor"/>
            </list>
        </property>
        <!--如果 URL 为 helloWorld.do ， 则转向 helloWorldAction -->
        <property name="mappings">
            <props>
                <prop key="helloWorld.do">helloWorldAction</prop>
            </props>
        </property>
    </bean>
    <!--定义视图及路径-->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.JstlView </value>
        </property>
        <!--JSP 都放在 WEB-INF/jsp 目录下-->
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <!--JSP 页面的后缀都是 jsp-->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!--定义控制器-->
```

```

<bean id="helloWorldAction" class="com.gc.action.HelloWorldAction">
    <property name="helloWorld">
        <value>HelloWorld</value>
    </property>
    <!--指定要返回的页面-->
    <property name="viewPage">
        <value>index</value>
    </property>
</bean>
</beans>

```

代码说明：使用 localeChangeInterceptor，表示如果请求是 helloWorld.do?language=，则会根据 language 的设定修改本地化信息。

localeChangeInterceptor 继承了 HandlerInterceptorAdapter，示例代码如下：

```

//***** LocaleChangeInterceptor.java*****
public class LocaleChangeInterceptor extends HandlerInterceptorAdapter {
    public static final String DEFAULT_PARAM_NAME = "locale";
    private String paramName = DEFAULT_PARAM_NAME;
    public void setParamName(String paramName) {
        this.paramName = paramName;
    }
    //该方法用来提取拦截，并设定 language
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler)
        throws ServletException {
        //把 locale 作为参数获取新的 locale
        String newLocale = request.getParameter(this.paramName);
        if (newLocale != null) {
            //解析新的 locale
            LocaleResolver localeResolver = RequestContextUtils.getLocaleResolver(request);
            if (localeResolver == null) {
                throw new IllegalStateException("No LocaleResolver found: not in a
DispatcherServlet request?");
            }
            //设定新的 locale
            LocaleEditor localeEditor = new LocaleEditor();
            localeEditor.setAsText(newLocale);
            localeResolver.setLocale(request, response, (Locale) localeEditor.getValue());
        }
        // Proceed in any case.
        return true;
    }
}

```

8.9 一个用 Spring MVC 实现用户登录的完整示例

前面的示例都是针对某一特点问题的，为了使开发人员对使用 Spring MVC 开发应用程序的全过程有一个整体了解，下面将从在 Eclipse 中建立一个 Tomcat 工程开始，一步一步地讲解开发 Spring MVC 的全过程。

8.9.1 在 Eclipse 中建立 Tomcat 工程项目 myMVC

利用第 2 章介绍的方法，在 Eclipse 中建立 Tomcat 工程项目 myMVC，并把 commons-logging.jar、jstl.jar、log4j-1.2.9.jar、spring.jar、standard.jar 放在 myMVC/WEB-INF/lib 目录下，并加入到编译路径中，把 spring.tld 放在 myMVC/WEB-INF 目录下。下面建立相关的目录：

(1) 在 myMVC 下建立文件夹 gc，在 gc 下再建立两个文件夹 src 和 jsp，在 src 下再建立文件夹 com，在 com 下再建立文件夹 gc，在 gc 下再建立文件夹 action，建立好的 myMVC 目录环境如图 8.29 所示。

(2) 在上面目录中的 myMVC 上单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，弹出 myMVC 的属性对话框，如图 8.30 所示。



图 8.29 建立好的 myMVC 目录环境

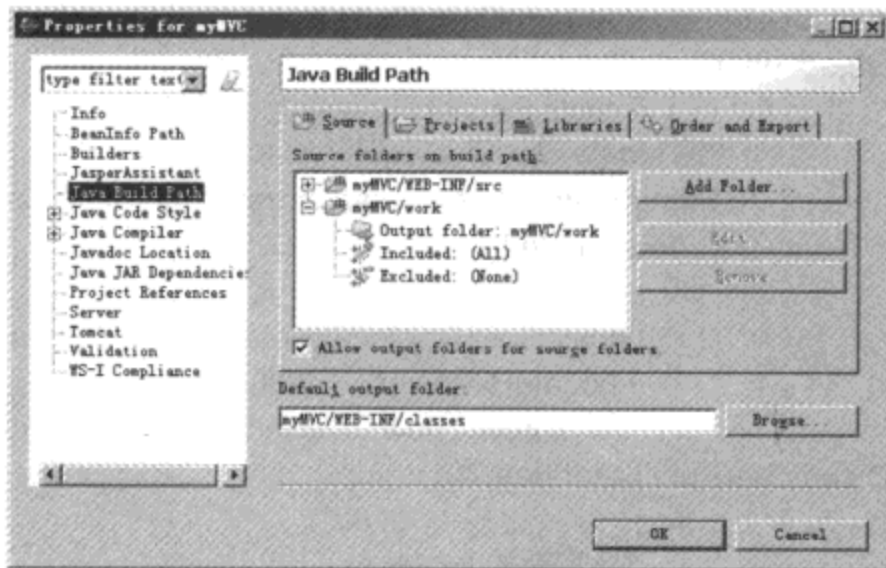


图 8.30 myMVC 的属性对话框

(3) 在 myMVC 的属性对话框中，单击 Add Folder 按钮，弹出 Source Folder Selection 对话框，如图 8.31 所示。

(4) 在该对话框中单击 gc 前面的加号图标，出现 gc 下的文件夹 jsp 和 src，如图 8.32 所示。

(5) 选中 src，然后单击 OK 按钮，返回到 Properties for myMVC 对话框，如图 8.33 所示。

(6) 在 Properties for myMVC 对话框中，单击 OK 按钮，最终形成的目录结构如图 8.34 所示。



图 8.31 Source Folder Selection 对话框



图 8.32 出现 gc 下的文件夹 jsp 和 src

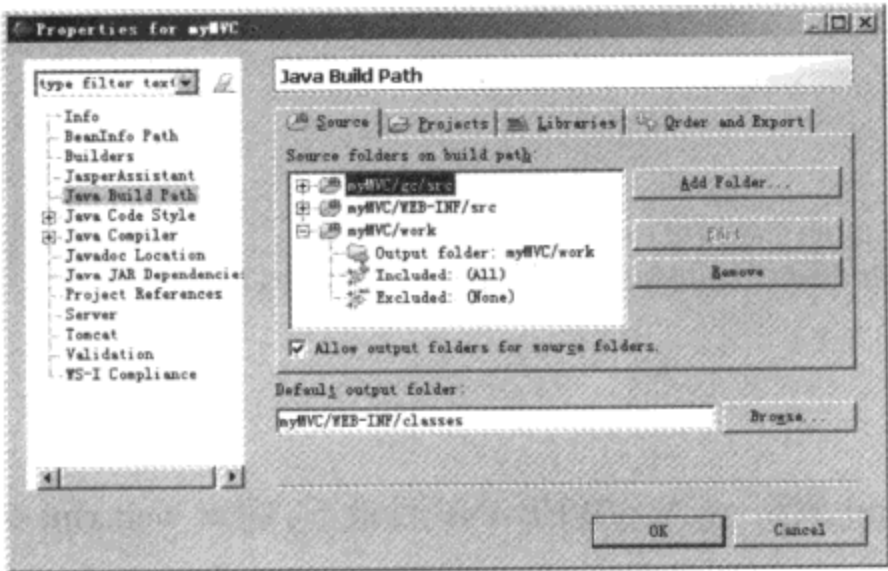


图 8.33 Properties for myMVC 对话框



图 8.34 最终形成的目录结构

8.9.2 编写日志文件放在 myMVC/WEB-INF/src 下

新建一个文件 log4j.properties，把 log4j.properties 放到 myApp/WEB-INF/src 目录下。编辑 log4j.properties 文件，内容如下：

```
log4j.rootLogger=DEBUG,stdout
log4j.logger.org=ERROR,A1
log4j.logger.com.gc=DEBUG,gc
#定义 log4j 的显示方式
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
#指定日志输入文件的名称
log4j.appender.stdout.File=gf.log
#指定日志输入文件的大小
log4j.appender.stdout.MaxFileSize=500KB
log4j.appender.stdout.MaxBackupIndex=50
log4j.appender.stdout.Append=true
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#定义 log4j 的显示方式
log4j.appender.A1=org.apache.log4j.RollingFileAppender
```



```

#指定日志输入文件的名称
log4j.appender.A1.File=org.log
#指定日志输入文件的大小
log4j.appender.A1.MaxFileSize=500KB
log4j.appender.A1.MaxBackupIndex=50
log4j.appender.A1.Append=true
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n

#定义 log4j 的显示方式
log4j.appender.A1=org.apache.log4j.RollingFileAppender
#指定日志输入文件的名称
log4j.appender.A1.File=gc.log
#指定日志输入文件的大小
log4j.appender.gc.MaxFileSize=500KB
log4j.appender.gc.MaxBackupIndex=50
log4j.appender.gc.Append=true
log4j.appender.gc.layout=org.apache.log4j.PatternLayout
log4j.appender.gc.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n

```

8.9.3 配置 web.xml

新建一个文件 web.xml，把 web.xml 放到 myApp/WEB-INF 目录下。编辑 web.xml 文件，内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2/EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <!--处理所有以 do 结尾的请求-->
    <servlet-mapping>
        <servlet-name>dispatcherServlet </servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!--定义该应用引用的自定义标签-->
    <taglib>
        <taglib-uri>/spring</taglib-uri>
        <taglib-location>/WEB-INF/spring.tld</taglib-location>
    </taglib>
    .....
</web-app>

```

8.9.4 编写登录页面 login.jsp

编写登录页面 login.jsp，放在 myMVC/gc/jsp 目录下。示例代码如下：

```
<%@taglib prefix="spring" uri="/spring"%>
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>Spring MVC 实例</title></head>
<body>
    <form name="User" action="/myMVC/login.do" method="post">
        <spring:bind path="command.username">
            输入用户名: <input type="text" name="${status.expression}" value="${status.value}"/>
        <br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
        <spring:bind path="command.password">
            输入密码: <input type="password" name="${status.expression}" value="${status.value}"/>
        <br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
        <spring:bind path="command.password2">
            确认密码: <input type="password" name="${status.expression}" value="${status.value}"/>
        <br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
        <input type="submit" value="提交"/>
    </form>
</body>
</html>
```

8.9.5 编写显示成功登录的页面 success.jsp

编写显示成功登录的页面 success.jsp，放在 myMVC/gc/jsp 目录下。示例代码如下：

```
<%@taglib prefix="spring" uri="/spring"%>
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>Spring MVC 实例</title></head>
<body>
    <spring:bind path="command.username">
        <H3>欢迎${status.value}登录成功</H3><br>
    </spring:bind>
</body>
</html>
```

8.9.6 编写存放用户登录信息的 Bean

在 com.gc.action 包中新建一个 Java 文件 User.java，用来存放用户登录信息。示例代码如下：

```
/****** User.java *****/
package com.gc.action;
public class User{
    //定义用户名
    private String username = null;
    //定义密码
    private String password = null;
    //定义确认密码
    private String password2 = null;
    //设定用户名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取用户名
    public String getUsername () {
        return this.username;
    }
    //设定密码
    public void setPassword (String password) {
        this.password = password;
    }
    //获取密码
    public String getPassword () {
        return this.password;
    }
    //设定确认密码
    public void setPassword2 (String password) {
        this.password2 = password;
    }
    //获取确认密码
    public String getPassword2 () {
        return this.password2;
    }
}
```

8.9.7 编写用户输入信息验证 UserValidator.java

在 com.gc.action 包中新建一个 Java 文件 UserValidator.java，用来验证用户输入的信息。示例代码如下：

```
/****** UserValidator.java *****/
package com.gc.action;
```



```

import org.springframework.validation.Validator;
import org.springframework.validation.Errors;
public class UserValidator implements Validator {
    public boolean supports(Class clazz) {
        return clazz.equals(User.class);
    }
    //对输入数据进行验证
    public void validate(Object obj, Errors errors) {
        User user = (User)obj;
        if(!"gf".equals(user.getUsername())) {
            errors.rejectValue("username", "", null, "用户名不正确");
        }
        if(!"123456".equals(user.getPassword())) {
            errors.rejectValue("password", "", null, "密码不正确");
        }
        if(!user.getPassword().equals(user.getPassword2())) {
            errors.rejectValue("password2", "", null, "2 次输入的密码不一致");
        }
    }
}

```

代码说明：如果用户输入用户名为“gf”，密码为“123456”，则正确。

说明：这里没有使用数据库，所以只是简单地在逻辑里判断一下数据是否正确。

8.9.8 编写用户登录逻辑 Login.java

在 com.gc.action 包中新建一个 Java 文件 Login.java，表示用户登录逻辑。示例代码如下：

```

//***** Login.java*****
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
//该类继承 SimpleFormController
public class Login extends SimpleFormController {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //覆写 onSubmit

```



```
public ModelAndView onSubmit(Object command , BindException errors) throws Exception {
    User user = (User)command;
    Map model = errors.getModel();
    model.put("user", user);
    return new ModelAndView(getSuccessView(), model);
}
}
```

8.9.9 编写配置文件 dispatcherServlet-servlet.xml

编写配置文档 dispatcherServlet-servlet.xml，放在 myMVC/ WEB-INF 目录下。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义映射-->
    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="login.do">login</prop>
            </props>
        </property>
    </bean>
    <!--定义视图及路径-->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.InternalResourceView</value>
        </property>
        <!--JSP 都放在 WEB-INF/jsp 目录下-->
        <property name="prefix">
            <value>/WEB-INF/jsp</value>
        </property>
        <!--JSP 页面的后缀都是 jsp-->
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
    <!--定义控制器-->
    <bean id="login" class="com.gc.action.Login">
        <property name="commandClass">
            <value>com.gc.action.User</value>
        </property>
        <!--指定验证类-->
        <property name="validator">
            <ref bean="userValidator"/>
        </property>
    </bean>

```

```
</property>
<!--指定验证失败要返回的页面-->
<property name="formView">
    <value>login</value>
</property>
<property name="successView">
    <value>success</value>
</property>
</bean>
<!--定义验证类-->
<bean id="userValidator" class="com.gc.action.UserValidator"/>
</beans>
```

以上内容配置完成后，所有的代码就编写完了，目前 myMVC 的目录结构如图 8.35 所示。

8.9.10 运行程序

(1) 启动 Tomcat，在浏览器地址栏中输入 `http://localhost:8080/myMVC/login.do`，按 Enter 键即可看到在浏览器中显示用来输入用户信息的页面，如图 8.36 所示。

(2) 输入用户名“gd”，密码“1234”，确认密码“123”，然后单击“提交”按钮，则会显示出错误提示信息，如图 8.37 所示。



图 8.35 目前 myMVC 的目录结构



图 8.36 用来输入用户信息的页面



图 8.37 显示错误提示信息的页面

(3) 输入用户名“gf”，密码“123456”，确认密码“123456”，然后单击“提交”按钮，则会显示登录成功信息的页面，如图 8.38 所示。

8.9.11 国际化支持

前面的示例中，文字都是直接写在页面中的，如果要换成其他语言，就会很麻烦，因为每个页面都要修改，这里加入该程序对国际化的支持。



图 8.38 显示登录成功信息的页面

(1) 用记事本编写存放信息资源的文档 messages.properties，存放在 myMVC/WEB-INF/src 目录下。messages.properties 的内容如下：

```
title=Spring MVC 实例
username=输入用户名:
password=输入密码:
password2=确认密码:
submit=提交
welcome=欢迎
loginSuccess=登录成功
usernameerror=用户名不正确
passworderror=密码不正确
doublepassword=2 次输入的密码不一致
```

- (2) 再把 messages.properties 放在 C 盘的根目录下。
- (3) 打开 Windows 的“开始”菜单，选择“运行”命令，弹出“运行”对话框，如图 8.39 所示。
- (4) 在“运行”对话框中，输入“cmd”，然后单击“确定”按钮，弹出“cmd 命令”对话框，如图 8.40 所示。

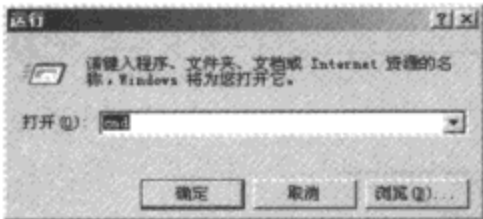


图 8.39 “运行”对话框

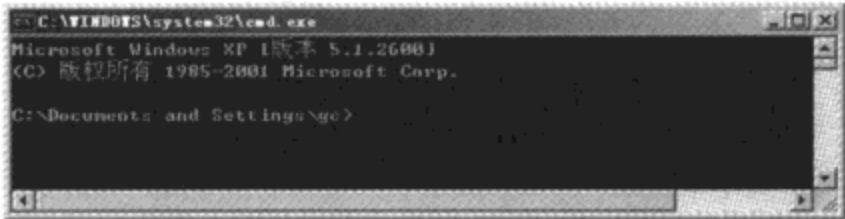


图 8.40 “cmd 命令”对话框

- (5) 在“cmd 命令”对话框中，输入“cd\”回到 C 盘根目录下。
- (6) 再输入“native2ascii messages.properties messages.txt”，然后按 Enter 键，如图 8.41 所示。



图 8.41 输入“native2ascii messages.properties messages.txt”后的“cmd 命令”对话框

(7) 这时在 C 盘的根目录下就会产生一个 messages.txt, messages.txt 的内容如下:

```
title=Spring MVC\u5b9e\u4f8b
username=\u8f93\u5165\u7528\u6237\u540d\u540d
password=\u8f93\u5165\u5bc6\u7801\u540d
password2=\u786e\u8ba4\u5bc6\u7801\u540d
submit=\u63d0\u4ea4
welcome=\u6b22\u8fce
loginSuccess=\u767b\u5f55\u6210\u529f
usernameerror=\u7528\u6237\u540d\u4e0d\u6b63\u786e
passworderror=\u5bc6\u7801\u4e0d\u6b63\u786e
doublepassword=2\u6b21\u8f93\u5165\u7684\u5bc6\u7801\u4e0d\u81f4
```

(8) 将 messages.txt 中的内容复制到 myApp/WEB-INF/src/messages.properties 中, 覆盖原来的内容。

(9) 用记事本编写存放信息资源的文档 messages_en_US.properties, 存放在 myMVC/WEB-INF/src 目录下。messages_en_US.properties 的内容如下:

```
title=Spring MVC
username=username
password=password
password2=password2
submit=submit
welcome=welcome
loginSuccess=loginSuccess
usernameerror=username error
passworderror=password error
doublepassword=doublepassword error
```

(10) 修改用户登录页面 login.jsp。示例代码如下:

```
<%@taglib prefix="spring" uri="/spring"%>
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title><spring:message code="title"/></title></head>
<body>
    <form name="User" action="/myMVC/login.do" method="post">
        <spring:bind path="command.username">
            <spring:message code="username"/><input type="text" name="${status.expression}"
value="${status.value}"/><br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
        <spring:bind path="command.password">
            <spring:message code="password"/> <input type="password" name="${status.expression}"
value="${status.value}"/><br>
            <font color="red"><b>${status.errorMessage}</b></font><br>
        </spring:bind>
        <spring:bind path="command.password2">
            <spring:message code="password2"/> <input type="password" name="${status.expression}"
value="${status.value}"/><br>
```



```

        <font color="red"><b>${status.errorMessage}</b></font><br>
    </spring:bind>
    <input type="submit" value="<spring:message code="submit"/>" />
</form>
</body>
</html>

```

(11) 修改用户登录页面 success.jsp。示例代码如下：

```

<%@taglib prefix="spring" uri="/spring"%>
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title><spring:message code="title"/></title></head>
<body>
    <spring:bind path="command.username">
        <H3><spring:message code="welcome"/></title>${status.value}<spring:message
code="loginSuccess"/></title></H3><br>
    </spring:bind>
</body>
</html>

```

(12) 修改用户输入信息验证 UserValidator.java。示例代码如下：

```

//***** UserValidator.java *****
package com.gc.action;
import org.springframework.validation.Validator;
import org.springframework.validation.Errors;
public class UserValidator implements Validator {
    public boolean supports(Class clazz) {
        return clazz.equals(User.class);
    }
    //对输入数据进行验证
    public void validate(Object obj, Errors errors) {
        User user = (User)obj;
        if(!"gf".equals(user.getUsername()) ) {
            errors.rejectValue("username", "usernameerror", null, "用户名不正确");
        }
        if(!"123456".equals(user.getPassword()) ) {
            errors.rejectValue("password", "passworderror", null, "密码不正确");
        }
        if(!user.getPassword().equals(user.getPassword2()) ) {
            errors.rejectValue("password2", "password2error", null, "2 次输入的密码不一致");
        }
    }
}

```

(13) 修改配置文件 dispatcherServlet-servlet.xml。示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>

```

```
<!--定义国际化消息-->
<bean id="messageSource" class="org.springframework.context.support.ResourceBundle-
MessageSource">
    <property name="basename">
        <value>messages</value>
    </property>
</bean>
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver">
</bean>
<!--定义映射-->
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
    <property name="mappings">
        <props>
            <prop key="login.do">login</prop>
        </props>
    </property>
</bean>
<!--定义视图及路径-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.InternalResourceView</value>
    </property>
<!--定义 JSP 存放的路径-->
    <property name="prefix">
        <value>/gc/jsp/</value>
    </property>
<!--定义 JSP 的后缀-->
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
<!--定义控制器-->
<bean id="login" class="com.gc.action.Login">
    <property name="commandClass">
        <value>com.gc.action.User</value>
    </property>
<!--定义验证类-->
    <property name="validator">
        <ref bean="userValidator"/>
    </property>
<!--验证失败返回的页面-->
    <property name="formView">
        <value>login</value>
    </property>
<!--验证成功返回的页面-->
    <property name="successView">
        <value>success</value>
    </property>
</bean>
```

```
</property>
</bean>
<!--定义验证类-->
<bean id="userValidator" class="com.gc.action.UserValidator"/>
</beans>
```

8.9.12 运行国际化后的程序

(1) 重启 Tomcat，在浏览器地址栏中输入 `http://localhost:8080/myMVC/login.do`，按 Enter 键即可看到浏览器中显示用来输入用户信息的页面，如图 8.42 所示：

(2) 输入用户名“gd”，密码“1234”，确认密码“123”，然后单击“提交”按钮，则会显示出错误提示信息，如图 8.43 所示。



图 8.42 用来输入用户信息的页面



图 8.43 显示错误提示信息的页面

(3) 输入用户名“gf”，密码“123456”，确认密码“123456”，然后单击“提交”按钮，则会显示登录成功信息的页面，如图 8.44 所示。

(4) 可以看出，和前面没有任何改变，如果把浏览器的默认语言改为美国英语，修改方法如下：

选择浏览器菜单栏里“工具”→Internet 选项，弹出“Internet 选项”对话框，如图 8.45 所示。



图 8.44 显示登录成功信息的页面



图 8.45 “Internet 选项”对话框

(5) 在“Internet 选项”对话框中，单击“语言”按钮，弹出“语言首选项”对话框，如图 8.46 所示。

(6) 在“语言首选项”对话框中，添加美国英语，把中文删除掉，然后再单击“确定”按钮。

(7) 在浏览器地址栏中输入 `http://localhost:8080/myMVC/login.do`，按 Enter 键即可看到浏览器中显示用来输入用户信息的英文页面，如图 8.47 所示。

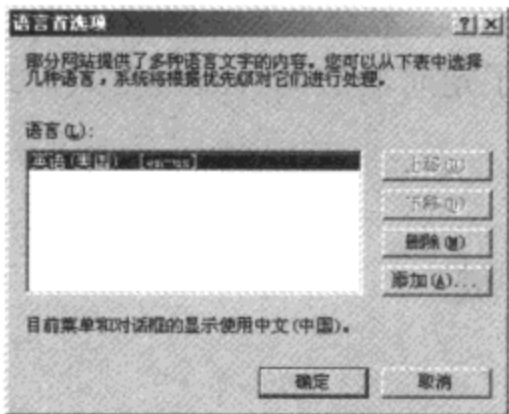


图 8.46 “语言首选项”对话框



图 8.47 用来输入用户信息的英文页面

(8) 输入用户名“gd”，密码“1234”，确认密码“123”，然后单击“提交”按钮，则会显示出错提示信息，如图 8.48 所示。

(9) 输入用户名“gf”，密码“123456”，确认密码“123456”，然后单击“提交”按钮，则会显示登录成功信息的英文页面，如图 8.49 所示。



图 8.48 显示错误提示信息的页面



图 8.49 显示登录成功信息的英文页面

8.10 小 结

本章讲述了 Spring 关于 Web 框架的相关知识，首先从 MVC 模式讲起，讲解 MVC 的原理，接着通过示例的方式介绍了 Spring MVC 各方面的知识点，最后通过一个完整实现用户登录的实例，把 Spring MVC 各方面的知识点衔接起来，从而使读者完整地了解了 Spring MVC 的开发过程。

第 9 章 Spring 的定时器

前面几章主要介绍了 Spring 的一些重要的基础知识。此外，Spring 还提供了在实际开发应用过程中经常用到的一些非常有用的工具，如定时器。本章主要讲解有关 Spring 定时器的一些知识，首先对定时器进行了介绍，接着讲解了定时器的两种实现方式，最后讲解了在 Spring 中如何实现定时器。

9.1 定时器简述

在实际开发应用过程中，开发人员经常需要定时或者重复执行一定的工作。例如，在固定的时间进行员工考勤刷卡数据的提取、审核，财务数据的入账、报表的产生等任务，这就需要用到定时器的功能。当笔者第一次遇到此类问题的时候，第一个想到的是用一个无限循环来完成，示例代码如下：

```
//***** HdjcMain.java*****
package com.gc.action;
public class HdjcMain extends Thread {
    public void run() {
        while (true){
            try{
                Thread.sleep ((int) (Math.random () * 1000));//停顿一秒
                date = new Date();
                //下面是定时执行的任务
                .....
            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

因为要判断是不是已经执行了定时任务，而且占用内存太大，所以这种方法不可取，下面将介绍定时器的两种实现方式，并给出使用 Spring 定时器的实例。

9.2 定时器的两种实现方式

在实际的应用中定时服务实现方式很多，最常见的就是 Windows 定时任务。在 Java 中

定时服务一般借助以下两种方式来实现：

- (1) 借助 java.util.Timer 来实现。
- (2) OpenSymphony 社区提供的 Quartz 来实现。

9.2.1 Timer

利用 Timer 开发定时任务分为两步：第 1 步是创建定时任务类；第 2 步是运行定时任务。而运行定时任务分为两种方式：一种是程序直接启动，一种是 Web 监听方式，如图 9.1 所示。

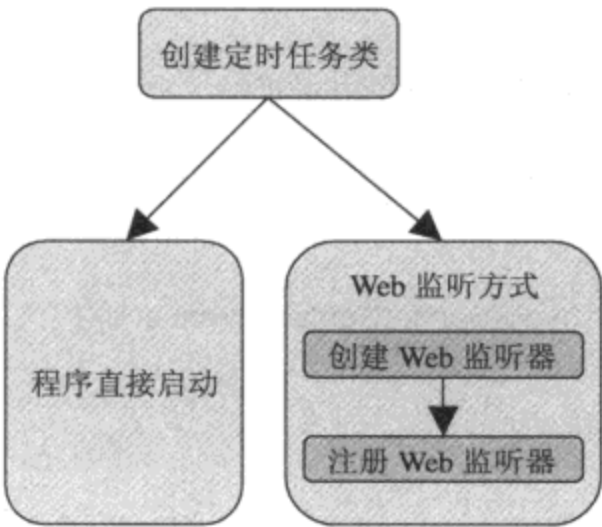


图 9.1 利用 Timer 开发定时任务步骤

(1) 创建定时任务类的示例代码如下：

```
//***** MainTask.java*****
package com.gc.action;
import java.util.TimerTask;
public class MainTask extends TimerTask {
    .....
    public void run() {
        //执行的定时器任务
    }
    .....
}
```

(2) 采用程序直接启动的方式启动定时任务的示例代码如下：

```
//***** Main.java*****
package com.gc.action;
public class Main {
    .....
    public void run() {
        //执行定时器任务
        Timer timer = new Timer();
    }
}
```

```
        timer.schedule(new MainTask(), 0, 1*1000);
    }
    .....
}
```

代码说明：timer.schedule()方法参数说明——timer.schedule（定时任务类，首次启动时间，间隔时间）。

（3）采用 Web 监听的方式启动定时任务，首先要创建监听类，示例代码如下：

```
/****** BindLoader.java*****
package com.gc.action;
public class BindLoader implements ServletContextListener {
    private Timer timer = null;
    //监听器初始化时执行事件
    public void contextInitialized(ServletContextEvent sce) {
        timer = new Timer();
        timer.schedule(new MainTask(), 0, 1*1000);
    }
    //监听器停止时执行事件
    public void contextDestroyed(ServletContextEvent sce) {
        timer.cancel();
    }
}
```

代码说明：监听器必须实现 javax.servlet.ServletContextListener 接口。

（4）创建监听类后在 web.xml 中注册该监听类。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
.....
    <listener>
        <listener-class>
            com.gc.action.BindLoader
        </listener-class>
    </listener>
    .....
</web-app>
```

9.2.2 Quartz

Quartz 是 OpenSymphony 开源组织在 Job scheduling 领域又一个开源项目，Quartz 可以用来创建简单或复杂的定时任务。利用 Quartz 开发定时任务同样分为两步：第 1 步是创建定时任务类；第 2 步是运行定时任务。而运行定时任务又分为两种方式：一种是程序直接启动，一种是 Web 监听方式，如图 9.2 所示。

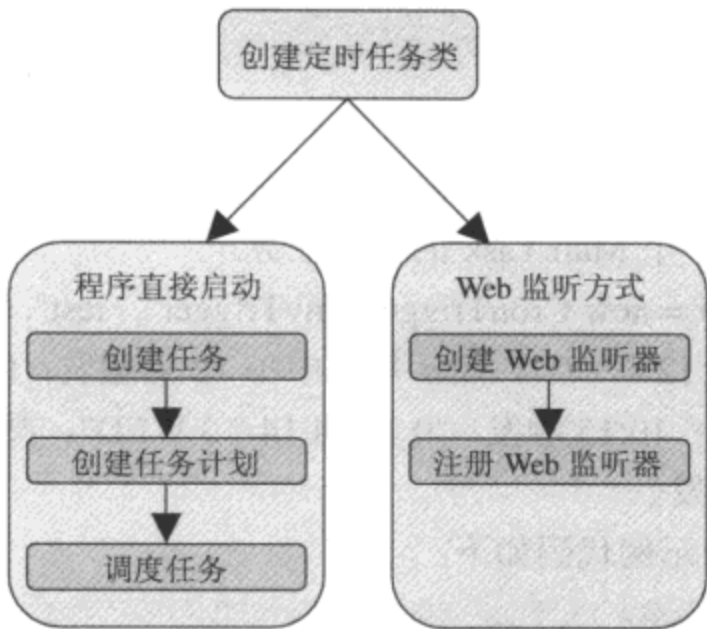


图 9.2 利用 Quartz 开发定时任务步骤

(1) 创建定时任务类的示例代码如下：

```

//***** MainTask.java *****
package com.gc.action;
public class MainTask implements Job {
    .....
    public void execute (JobExecutionContext context) throws JobExecutionException {
        //执行的定时器任务
        .....
    }
    .....
}

```

(2) 采用程序直接启动的方式启动定时任务，还要创建任务调度器及配置相应的任务计划，示例代码如下：

```

//***** MainTask.java *****
package com.gc.action;
public class MainShedule {
    private static Scheduler sched;
    public static void run() throws Exception {
        // 创建任务
        JobDetail jobDetail = new JobDetail("myJob", sched.DEFAULT_GROUP,
MainTask.class);
        // 创建任务计划
        CronTrigger trigger = new CronTrigger("myTrigger", "test", "0/10 * * * * ?");
        // 调度任务
        sched = new org.quartz.impl.StdSchedulerFactory().getScheduler();
        sched.scheduleJob(jobDetail, trigger);
        sched.start();
    }
    public static void stop() throws Exception {
        sched.shutdown();
    }
}

```



```

    }
}

```

代码说明：

- ❑ `JobDetail jobDetail = new JobDetail("myJob", sched.DEFAULT_GROUP, MainTask.class)`，首先创建一个 `MainTask` 的定时任务。
- ❑ `CronTrigger trigger = new CronTrigger("myTrigger", "test", "0/10 * * * * ?")`，创建任务计划。例如，"`0 0 12 * * ?`"，表示每天中午 12 点触发；"`0 15 10 * * ? 2006`"，表示 2006 年的每天上午 10:15 触发；"`0 10,44 14 ? 3 WED`"，表示每年三月的星期三的下午 2:10 和 2:44 触发。

(3) 执行定时任务的示例代码如下：

```

//***** Main.java *****
package com.gc.action;
public class Main {
    .....
    public void run() {
        //执行定时器任务
        MainShedule.run();
    }
    .....
}

```

(4) 采用 Web 监听的方式启动定时任务，首先要创建监听类，示例代码如下：

```

//***** ServiceLoader.java *****
package com.gc.action;
public class BindLoader implements ServletContextListener {
    //监听器初始化时执行事件
    public void contextInitialized(ServletContextEvent sce) {
        MainShedule.run();
    }
    //监听器停止时执行事件
    public void contextDestroyed(ServletContextEvent sce) {
        MainShedule.stop();
    }
}

```

代码说明：监听器必须实现 `javax.servlet.ServletContextListener` 接口。

(5) 创建监听类后在 `web.xml` 中注册该监听类，示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    .....
    <listener>
        <listener-class>
            com.gc.action.BindLoader
        </listener-class>
    </listener>

```

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

9.2.3 两种方式的比较

使用 Timer 方式实现定时器，原理简单、实现方便，在执行简单重复任务时比较方便。其不足之处是无法知道系统在几点几分执行，而且必须继承指定类。

使用 Quartz 方式实现定时器，可以方便、清楚地设定定时器启动的时间，定时器参数比较灵活，而且很容易实现各种复杂的定时任务。其不足之处是实现指定接口，而且需要加载相应的框架。两种定时器应用在不同的情况下。在实际应用中，Quartz 方式定时器能够方便、快捷地实现所有定时任务，因此应用比较广泛。

9.3 利用 Spring 简化定时任务的开发

Spring 对上述的两种定时器方式都提供了支持，并且实现这两种定时器的步骤基本一样。利用 Spring 实现定时器的开发过程分为以下 3 步：

- (1) 创建定时任务类。
- (2) 注册定时任务类，并配置任务计划和任务调度器。
- (3) 在 Web 项目中启动定时服务。

下面分别来介绍在 Spring 中使用 Timer 和 Quartz 这两种定时器方式的方法。

9.3.1 在 Spring 中使用 Timer 实现定时器

(1) 按照上面介绍的步骤，首先创建定时任务类 MainTask。MainTask.java 的示例代码如下：

```
//***** MainTask.java*****
package com.gc.action;
import java.util.TimerTask;
public class MainTask extends TimerTask {
    .....
    public void run() {
        //执行的定时器任务
        .....
    }
    .....
}
```

(2) 注册定时任务类，并配置任务计划和任务调度器。首先新建一个文件 TimerConfig.xml，放在 WEB-INF 文件夹下。TimerConfig.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">
<beans>
    <!--注册定时执行实体-->
    <bean id="mainTask" class="com.gc.action.MainTask" />
    <!--注册定时器信息-->
    <bean id="stTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <!--首次执行任务前需要等待 2 秒-->
        <property name="delay">
            <value>2000</value>
        </property>
    <!--任务执行的周期为 4 秒-->
        <property name="period">
            <value>4000</value>
        </property>
    <!--具体的执行任务-->
        <property name="timerTask">
            <ref local="mainTask" />
        </property>
    </bean>
    <!--配置任务调度器-->
    <bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
    <!--注入定时器列表-->
        <property name="scheduledTimerTasks">
            <list>
                <ref local="stTask" />
                .....
            </list>
        </property>
    </bean>
</beans>

```

(3) 在 Web 项目中启动定时服务。示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    .....
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/TimerConfig.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    .....
</web-app>

```

9.3.2 在 Spring 中使用 Quartz 实现定时器

(1) 按照上面介绍的步骤，首先创建定时任务类 MainTask。MainTask.java 的示例代码如下：

```
//***** MainTask.java*****  
package com.gc.action;  
public class MainTask {  
    .....  
    public void execute() {  
        //执行的定时器任务  
        .....  
    }  
    .....  
}
```

(2) 注册定时任务类，并配置任务计划和任务调度器，首先新建一个文件 TimerConfig.xml，放在 WEB-INF 文件夹下。TimerConfig.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/  
spring-beans.dtd">  
<beans>  
    <!--注册定时执行实体-->  
    <bean id="mainTask" class="com.gc.action.MainTask" />  
    <!--注册定时器信息-->  
    <bean id="mainJob"  
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"  
">  
        <!--指定要执行的定时任务类-->  
        <property name="targetObject">  
            <ref bean="mainTask"/>  
        </property>  
        <!--指定执行任务的方法名称-->  
        <property name="targetMethod">  
            <value>execute</value>  
        </property>  
    <bean id="timeTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">  
        <!--声明要运行的实体-->  
        <property name="jobDetail">  
            <ref bean="mainJob"/>  
        </property>  
        <!--设置要运行的时间-->  
        <property name="cronExpression">  
            <value>12,23 * * * * ?</value>  
        </property>  
    </bean>  
    <!--注册定时器-->
```



```
<bean id="sfb" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <!--注入定时器实体-->
  <property name="triggers">
    <list>
      <ref local="timeTrigger"/>
    </list>
  </property>
</bean>
</beans>
```

(3) 在 Web 项目中启动定时服务。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  .....
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/TimerConfig.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  .....
</web-app>
```

9.4 小 结

本章主要讲述了定时器的两种实现方式：使用 Timer 定时器方式和使用 Quartz 定时器方式，并讲解了它们在 Spring 中的实现方式。

在 Spring 中使用 Timer 实现定时器的方式，并没有减少代码量，只提供了一种使用 Timer 的解决方案。而 Quartz 实现定时器的方式，Spring 的优势就比较明显，因为它取消了原有的实现 Job 接口，可将任意类中的任意方法注册为定时任务，从而利用简单的配置代替了原有的代码量，提高了系统的灵活性，减少文件数量。

第 3 篇



Spring 与其他工具 整合应用

第 10 章 Spring 与 Struts 的整合

第 11 章 Spring 与 Hibernate 的整合

第 12 章 在 Spring 中使用 Ant

第 13 章 在 Spring 中使用 Junit



第 10 章 Spring 与 Struts 的整合

前面讲过，Spring MVC 实现了 MVC 模式，可以使开发人员很方便地进行 Web 开发。同样，另一个开源项目 Struts 更早地实现了 MVC 模式，开发人员当中使用得也比较多，使用 Spring 可以很方便地和 Struts 整合在一起。本章首先对 Struts 进行介绍，然后通过一个示例使读者快速上手 Struts，接着讲解 Struts 提供的几个比较重要的类及 Struts 对国际化的支持和自定义标签功能，再给出 Spring 与 Struts 整合的几种实现方式，最后通过实例展示 Spring 与 Struts 的整合方式。

10.1 Struts 介绍

10.1.1 Struts 的历史

通过前面对 Model 的介绍可以知道，Model2 会导致多个 Controller 的出现，并且对页面导航的处理比较复杂。所以 Craig R. McClanahan 于 2000 年 5 月提交了一个框架给 Java Community，这就是后来的 Struts。2001 年 7 月，Struts 1.0 正式发布，该项目也成为了 Apache Jakarta 的子项目之一。Struts 其实就是在 Model2 的基础上实现的一个 MVC 框架，它只有一个中心控制器，采用 XML 定制转向的 URL，采用 Action 来处理逻辑。

10.1.2 Struts 的体系结构

Struts 的体系结构如图 10.1 所示。

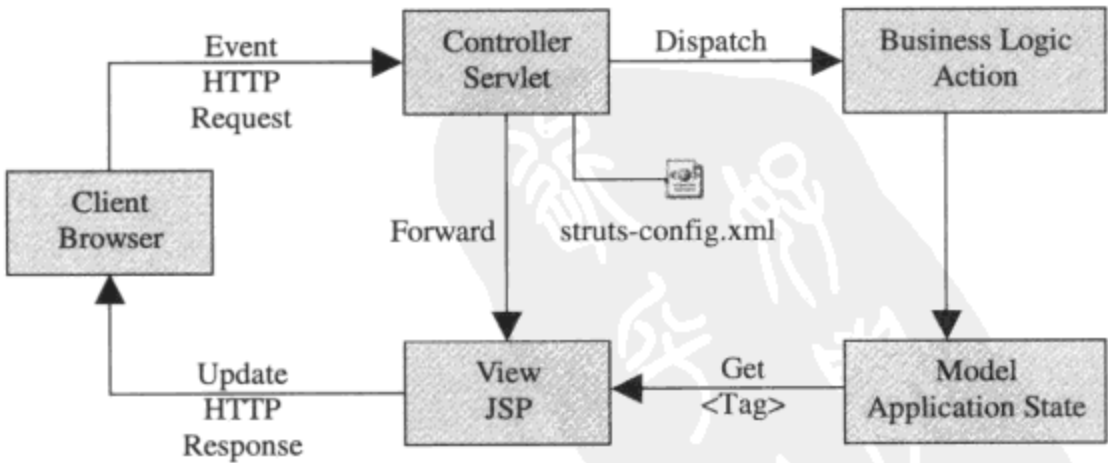


图 10.1 Struts 的体系结构

10.2 Struts 的下载和配置

同样，在详细讲解 Struts 之前，先来了解一下 Struts 的下载和配置方法。

10.2.1 下载 Struts

从 <http://struts.apache.org/download.cgi> 下载 Struts 的最新版本。Struts 1.2.9 版本的下载画面如图 10.2 所示。

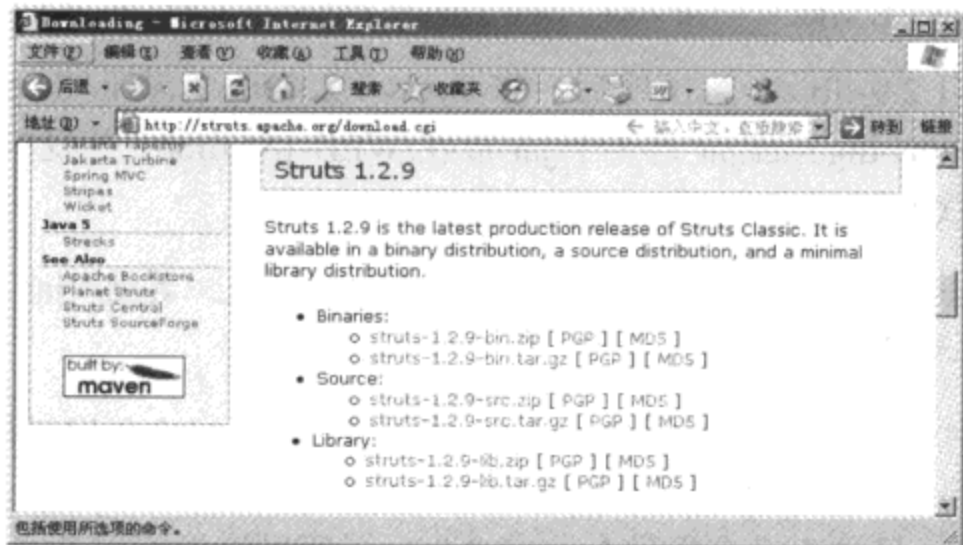


图 10.2 Struts 1.2.9 版本的下载画面

在 Struts 1.2.9 版本的下载画面中，struts-1.2.9-bin 中包含 Struts 运行所需要的 jar 和一些示例程序；struts-1.2.9-src 中包含 Struts 的源代码和文档；struts-1.2.9-lib 中只包含 Struts 运行所需要的 jar。下载 struts-1.2.9-src.zip 和 struts-1.2.9-lib.zip 到本地硬盘，并解压缩。

10.2.2 配置 Struts

当 Struts 解压缩完毕后，就可以在 Eclipse 中配置 Struts 了。这里主要从 3 个方面讲解。首先在 Eclipse 中建立一个项目 myStruts，然后把 Struts 相关的 jar 配置到该项目中，最后在项目中建立 3 个包。这 3 个包的作用分别为 com.gc.action 用来存放实体类，com.gc.impl 用来存放接口类，com.gc.test 用来存放测试类。具体步骤如下：

(1) 运行 Eclipse，选择 File→New→Project 命令，Eclipse 将弹出 New Project 对话框，如图 10.3 所示。

(2) 选择列表框中 Java 下的 Tomcat Project，然后单击 Next 按钮，弹出 New Tomcat Project 对话框，如图 10.4 所示。

(3) 在 New Tomcat Project 对话框的 Project name 文本框中输入“myStruts”，然后单击 Finish 按钮，项目即建立成功。myStruts 的目录结构如图 10.5 所示。

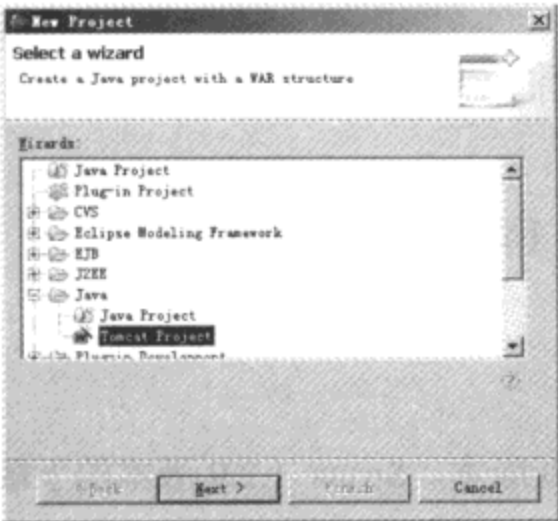


图 10.3 New Project 对话框



图 10.4 New Tomcat Project 对话框

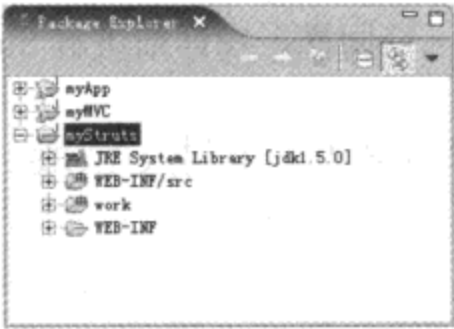


图 10.5 myStruts 的目录结构

(4) 把 struts-1.2.9-lib.zip 解压缩后，将 struts-1.2.9-lib 目录下的 struts.jar、commons-beanutils.jar、commons-digester.jar 这 3 个 jar 放在 /WEB-INF/lib/ 下并复制到 myStruts /WEB-INF/lib 目录下，即 CLASSPATH 中。

注意：这 3 个 jar 是必需的，否则会出现莫名其妙的问题。其他 *.jar 如果需要则用到再放，不需要用可以不放。

(5) 在程序中需要 jakarta 的日志元件，所以把 struts-1.2.9-lib 目录下的 commons-logging.jar 也复制到 myStruts /WEB-INF/lib 目录下。

(6) 因为 Struts 目录下没有提供 log4j-1.2.9.jar，所以这里使用 Spring 下的 log4j-1.2.9.jar，把 spring-framework-2.0-m1/lib/log4j 下的 log4j-1.2.9.jar 复制到 myApp /WEB-INF/lib 目录下。

(7) 用 Windows 自带的文本编辑器建立一个文件 log4j.properties，把 log4j.properties 放到 myStruts /WEB-INF/src 目录下。

(8) 编辑 log4j.properties 文件。内容如下：

```
log4j.rootLogger=DEBUG,stdout
log4j.logger.org=ERROR, A1
#定义 log4j 的显示方式
log4j.appender.A1=org.apache.log4j.RollingFileAppender
#指定日志输入文件的名称
log4j.appender.A1.File=org.log
#指定日志输入文件的大小
log4j.appender.A1.MaxFileSize=500KB
log4j.appender.A1.MaxBackupIndex=50
log4j.appender.A1.Append=true
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
#指定日志输入文件的内容格式
#定义 log4j 的显示方式
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
#指定日志输入文件的名称
log4j.appender.stdout.File=gf.log
#指定日志输入文件的大小
```

```
log4j.appender.stdout.MaxFileSize=500KB
log4j.appender.stdout.MaxBackupIndex=50
log4j.appender.stdout.Append=true
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
#指定日志输入文件的内容格式
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
```

(9) 在 myStruts 上单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，弹出 Properties for myStruts 对话框，如图 10.6 所示。

(10) 在 Properties for myStruts 对话框中，选择对话框左边列表框中的 Java Build Path。

(11) 在 Libraries 选项卡中，单击 Add JARs...按钮，弹出 JAR Selection 对话框，如图 10.7 所示。

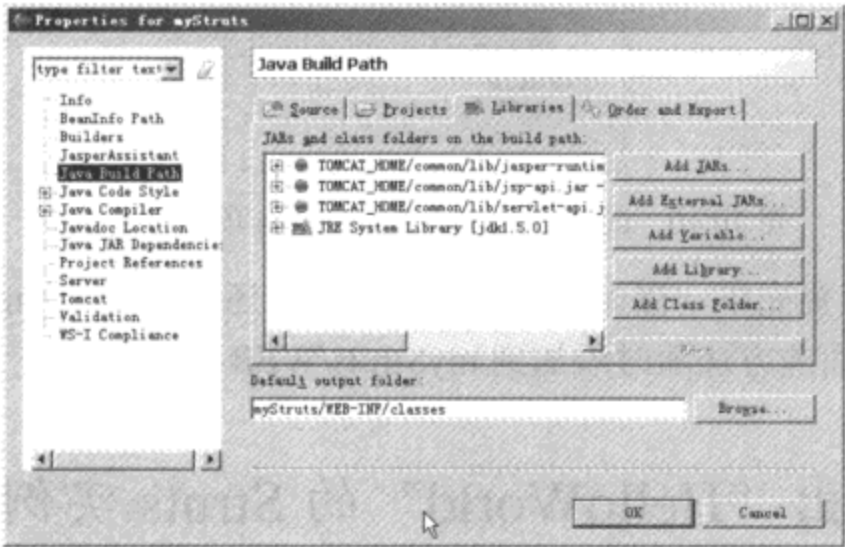


图 10.6 Properties for myStruts 对话框



图 10.7 JAR Selection 对话框

(12) 在 JAR Selection 对话框中，打开列表框中 myStruts 一直到 lib 目录下出现 5 个 jar: struts.jar、commons-beanutils.jar、commons-digester.jar、commons-logging.jar 和 log4j-1.2.9.jar。

(13) 按住 Ctrl 键，选中这 5 个 jar，然后单击 OK 按钮，返回到 Properties for myStruts 对话框，如图 10.8 所示。

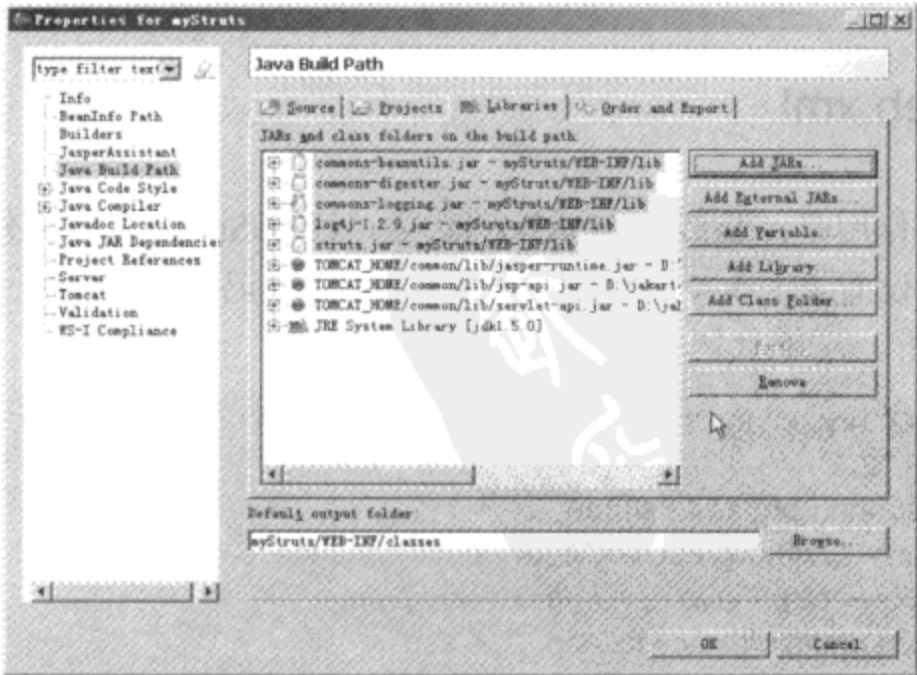


图 10.8 Properties for myStruts 对话框

(14) 在 Properties for myStruts 对话框中单击 OK 按钮，即可完成对 Struts 的配置。

(15) 再次在 myStruts 上单击鼠标右键，在弹出的快捷菜单中选择 New→Package 命令，将弹出 New Java Package 对话框，如图 10.9 所示。

(16) 在 New Java Package 对话框的 Name 文本框中输入“com.gc.action”，然后单击 Finish 按钮，即可建立 com.gc.action 包。

(17) 用同样的方法建立 com.gc.impl 包和 com.gc.test 包。

(18) 最终配置好 Struts 的 myStruts 项目目录结构如图 10.10 所示。

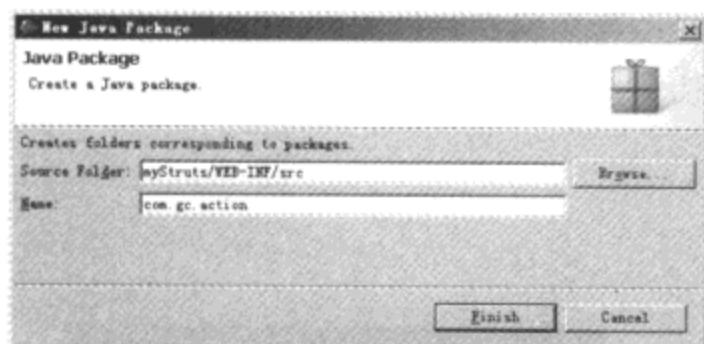


图 10.9 New Java Package 对话框



图 10.10 配置好 Struts 的 myStruts 项目目录结构

至此，一个简单的 Struts 运行环境配置完毕。假如在后面的项目中，还需要用到 Struts 所依赖的其他开源项目的 jar，可以用上面的方法随时添加到 myStruts/WEB-INF/lib 下。

10.3 一个在 JSP 页面输出“HelloWorld”的 Struts 实例

在详细讲解 Struts 之前，先来看一个在 JSP 页面输出“HelloWorld”的 Struts 实例，大体认识一下使用 Struts 的流程。

这个实例的实现思路是：使用前面配置好的 myStruts 工程，首先新建一个 web.xml 文件，然后编写实现输出的 JSP 页面 index.jsp，接着编写控制器 HelloWorldAction.java，最后配置 Struts 文档 struts-config.xml，并运行实例。

10.3.1 配置 web.xml

web.xml 的主要作用是：装载 ActionServlet 类，读取 Struts 配置文件，设置一些初始参数，加入标记库，设置如 *.do、*.form 的映射等。用记事本新建一个 web.xml，放在 myStruts/WEB-INF 下。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
```

```
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
<!--初始化配置文档-->
<init-param>
  <param-name>config</param-name>
  <param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>actionServlet</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

代码说明:

- ❑ web.xml 的开头几行是固定的, 它定义了程序部署描述文件的字符编码、版本等, 最顶层的元素为<web-app>, 其余元素都要定义在<web-app>内。
- ❑ <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">, 采用的是 Servlet 2.4 规定的 Web 程序部署描述格式。以前的 Servlet 经常使用的描述格式是: <!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>。
- ❑ <servlet>用来定义一个 servlet。
- ❑ <servlet-name>是<servlet>的属性, 用来定义 Servlet 的名称, 这里是 actionServlet。
- ❑ <servlet-class>是<servlet>的属性, 用来指定上面定义 Servlet 的具体实现类。这里是 org.apache.struts.action.ActionServlet。
- ❑ <init-param>是<servlet>的属性, 用来定义 Servlet 的初始化参数, 这里指定要初始化 WEB-INF 文件夹下的 struts-config.xml。如果不指定, 默认也是 WEB-INF 文件夹下的 struts-config.xml。
- ❑ <load-on-startup>是<servlet>的属性, 指定当 Web 启动时, 加载 Servlet 的顺序, 当它的值大于等于 0 时, Servlet 容器先加载数值小的 Servlet, 再加载大的; 如果它的值小于 0 或没有设定, 则 Servlet 容器将在 Web 首次访问时加载这个 Servlet。
- ❑ <servlet-mapping>用来指定访问 Servlet 时的 URL。
- ❑ <servlet-name>用来定义 Servlet 的名称, 和前面<servlet>中定义的名称相对应。
- ❑ <url-pattern>指定访问 Servlet 时 URL 的相对路径, 这里表示所有以 do 结尾的都要经过指定的 actionServlet。

10.3.2 编写实现输出的 JSP 页面 index.jsp

在 myStruts/WEB-INF 下新建一个 JSP 文件夹, 把 index.jsp 放在这个文件夹下。index.jsp 的示例代码如下:


```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>第一个 Struts 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <H1><%=str%></H1>
</body>
</html>
```

10.3.3 编写控制器 HelloWorldAction.java

在 Struts 里,所有的控制器都继承了 `org.apache.struts.action.Action` 类,并实现了 `execute()` 方法。在 `com.gc.action` 包里新建一个类 `HelloWorldAction`。`HelloWorldAction.java` 的示例代码如下:

```
/** ***** HelloWorldAction .java ***** */
package com.gc.action;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.apache.struts.action.*;
//继承 Action
public class HelloWorldAction extends Action {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //覆写 execute
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
                                request, HttpServletResponse response) throws Exception {
        request.setAttribute("helloWorld", "HelloWorld");
        return mapping.findForward("index");
    }
}
```

代码说明:

- ❑ `execute()`方法返回一个 `ActionForward`, 用来显示返回的页面。
- ❑ `return mapping.findForward("index")`, 表示返回的页面是在配置文件中定义的 `forward` 属性的名称为 `index` 的页面。

10.3.4 配置 Struts 文档 struts-config.xml

因为前面定义 web.xml 时，定义的初始化参数是初始化 WEB-INF 文件夹下的 struts-config.xml，所以在 WEB-INF 文件夹下建立 struts-config.xml。示例代码如下：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
  <action-mappings>
    <action path="/helloWorld" type="com.gc.action.HelloWorldAction">
      <forward name="index" path="/WEB-INF/jsp/index.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

代码说明：

- 这里定义的 path="/helloWorld"表明 URL 的路径是 helloWorld.do，要访问的类是 com.gc.action.HelloWorldAction。
- <forward name="index" path="/WEB-INF/jsp/index.jsp"/>，定义了访问 Action 后返回的页面是 WEB-INF/jsp 下的 index.jsp。

10.3.5 启动 Tomcat

直接单击 Eclipse 工具按钮，启动 Tomcat。Tomcat 启动成功后显示的信息如图 10.11 所示。

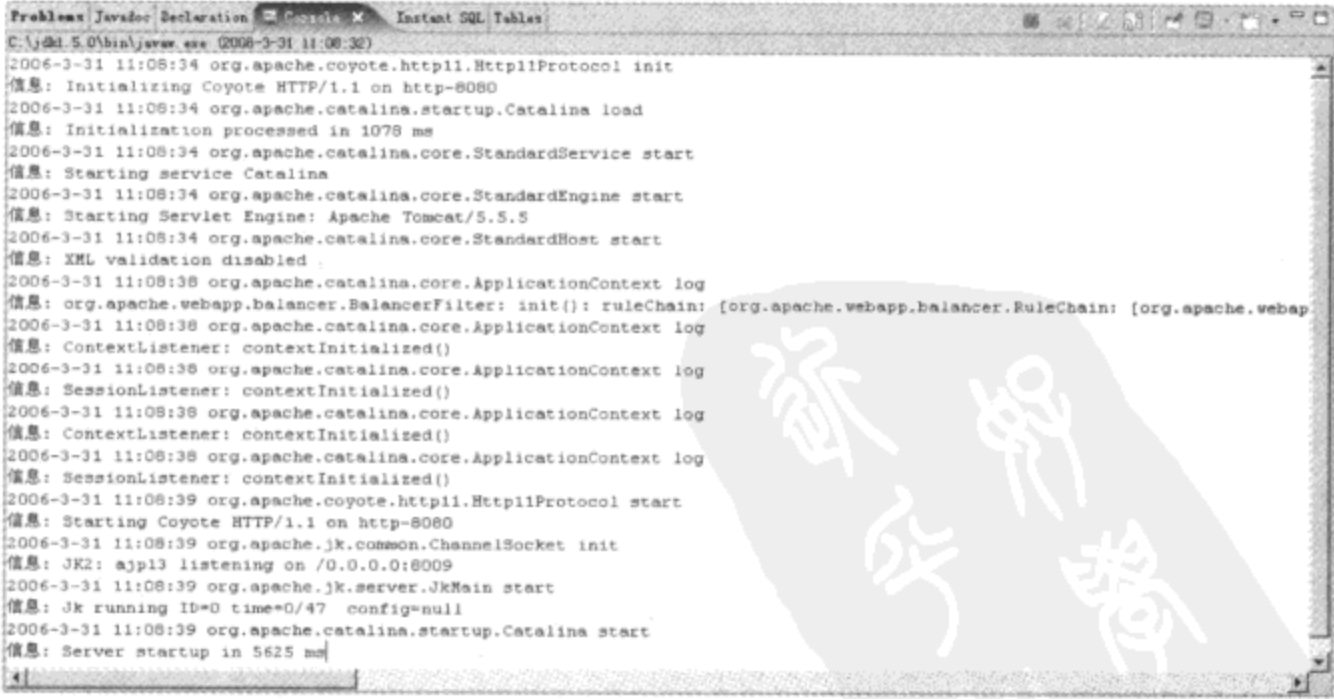


图 10.11 Tomcat 启动成功后显示的信息

10.3.6 运行程序

在浏览器的地址栏中输入“`http://localhost:8080/myStruts/helloWorld.do`”，按 Enter 键即可看到在浏览器中输出“HelloWorld”，如图 10.12 所示。



图 10.12 在浏览器中输出“HelloWorld”

10.4 Struts 的几个主要类简介

在前面的示例中，用到了 Struts 提供的几个主要的类，包括 `ActionServlet`、`Action` 和 `Action Mapping` 等。这几个类在 Struts 中起到了很重要的作用，下面分别进行讲解。

10.4.1 ActionServlet（控制器）

`org.apache.struts.action.ActionServlet` 继承于 `javax.servlet.http.HttpServlet`。`ActionServlet` 就是 MVC 模式中的 Controller，它截获用户的 Http 请求，根据配置文件的描述，转发到适当的处理器。如果这是该处理器收到的第一个请求，则将初始化实例并缓存起来。接着会调用 `Action` 实例的 `execute()` 方法，并将 `request` 和 `response` 对象传给该方法。执行该方法后会返回一个 `ActionForward`。它的示例代码如下：

```
//***** ActionServlet.java*****
public class ActionServlet extends HttpServlet {
    .....
    // doGet()方法调用了 process
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        process(request, response);
    }
    // doPost ()方法调用了 process
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {

        process(request, response);
    }
    .....
}
```

```

//负责请求转发
protected void process(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    ModuleUtils.getInstance().selectModule(request, getServletContext());
    ModuleConfig config = getModuleConfig(request);
    根据 ModuleConfig 获得 RequestProcessor 的一个对象
    RequestProcessor processor = getProcessorForModule(config);
    if (processor == null) {
        processor = getRequestProcessor(config);
    }
    //具体负责请求转发的方法
    processor.process(request, response);

}
}

```

从示例代码可以看出，在 ActionServlet 的 doGet() 和 doPost() 方法中，都调用了 process() 方法，从而将请求交给 RequestProcessor 的 process() 方法来进行处理。

RequestProcessor 中 process() 方法的示例代码如下：

```

public void process(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    // 处理 contentType 为 multipart/form-data 的 POST 请求
    request = processMultipart(request);
    //获取 URL 路径
    String path = processPath(request, response);
    if (path == null) {
        return;
    }
    //输出日志
    if (log.isDebugEnabled()) {
        log.debug("Processing a " + request.getMethod() +
            " for path " + path + "");
    }
    // 为当前用户选择一个 Locale
    processLocale(request, response);
    //确定 content type
    processContent(request, response);
    processNoCache(request, response);
    //前置处理，类似于 AOP 的前置处理
    if (!processPreprocess(request, response)) {
        return;
    }
    this.processCachedMessages(request, response);
    //确定 ActionMapping
    ActionMapping mapping = processMapping(request, response, path);
    if (mapping == null) {
        return;
    }
}

```



```
}
// Check for any role required to perform this action
if (!processRoles(request, response, mapping)) {
    return;
}
// 创建一个 ActionForm, 并一直使用它
ActionForm form = processActionForm(request, response, mapping);
//将页面中获取的表单值填入 ActionForm
processPopulate(request, response, form, mapping);
//检查是否进行 Validate 判断
try {
    if (!processValidate(request, response, form, mapping)) {
        return;
    }
} catch (InvalidCancelException e) {
    ActionForward forward = processException(request, response, e, form, mapping);
    processForwardConfig(request, response, forward);
    return;
} catch (IOException e) {
    throw e;
} catch (ServletException e) {
    throw e;
}
// 假如配置文档中 action 的 forward 属性被设定, 则不再执行 type 属性
if (!processForward(request, response, mapping)) {
    return;
}
//假如配置文档中 action 的 include 属性被设定, 则不再执行 type 属性
if (!processInclude(request, response, mapping)) {
    return;
}
// 执行 action 的 type 属性, 如果没有则创建一个, 以后一直使用, 单例模式
Action action = processActionCreate(request, response, mapping);
if (action == null) {
    return;
}
// 执行 Action 的 execute()或 perform()方法
ActionForward forward =
    processActionPerform(request, response,
        action, form, mapping);
// 返回 ActionForward
processForwardConfig(request, response, forward);
}
```

通过上面的示例代码, 希望读者能够了解 Struts 中 Controller 角色的内部机理, 从而能够更加熟练地掌握 Struts。

在 Struts 中, 这部分和 Spring 一样, 主要是通过配置文件来完成的。在前面的示例中, 已经有了一个 web.xml 的简单示例。这里给出一个完整的示例代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <!-- 配置文件的位置和名称-->
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
      <!-- 应用程序的资源集合的类-->
      <param-name> application </param-name>
      <param-value>null</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>actionServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>

```

代码说明：这里的初始化参数<init-param>可以定义多个。

10.4.2 Action（适配器）

前面的示例中，HelloWorldAction 类继承了 org.apache.struts.action.Action，并实现了 execute() 方法。它取得请求中的参数，并验证资料的正确性，最后把执行的结果返回给 ActionForward。Action 的示例代码如下：

```

//***** Action.java*****
public class Action {
  .....
  // execute 的返回值都是 ActionForward
  public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    ServletRequest request,
    ServletResponse response)
    throws Exception {
    try {
      return execute(
        mapping,
        form,
        (HttpServletRequest) request,

```

```

        (HttpServletResponse) response);
    } catch (ClassCastException e) {
        return null;
    }
}
// execute 的返回值都是 ActionForward
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    return null;
}
.....
}

```

代码说明：

- ❑ Action 中定义了两个 execute()方法，都返回 ActionForward，当开发人员使用时，直接覆写其中的一个即可。
- ❑ 在 Struts 中，采用的是单例模式，一个 Action 创建之后，会一直使用，所以要注意线程安全，不要使用静态变量。

10.4.3 ActionMapping（映射）

ActionServlet 将 ActionMapping 传递到 Action，主要是通过配置文件来实现的。在 Struts 中，默认的配置文件是 WEB-INF 下的 struts-config.xml。通过配置文件 struts-config.xml 可以定义全局转发的方式、ActionMapping、ActionForm 和数据源。

先来看前面的示例代码：

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <action-mappings>
        <!-- 请求的 URL 对应于< action>中的 path 属性-->
        <action path="/helloWorld" type="com.gc.action.HelloWorldAction">
            <!-- forward 表示执行完毕后返回的页面-->
            <forward name="index" path="/WEB-INF/jsp/index.jsp"/>
        </action>
    </action-mappings>
</struts-config>

```

代码说明：

- ❑ <action-mappings>用来描述 ActionMapping，在<action-mappings>下的每一个<action>都是一个 ActionMapping 对象，当客户端发出请求至 ActionServlet 时，请求的 URL 对应于< action>中的 path 属性，而要执行的 Action 则是 type 属性所设定

的内容, 执行完 Action 后, 返回的 ActionForward, 则在< forward >中设定。

□ 上面配置文件的访问路径是: <http://localhost:8080/myStruts/helloWorld.do>。

1. 全局转发的定义方式

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
  <!-- forward 的属性 name 表示全局转发的名称, path 表示全局转发的相对路径-->
  <global-forwards>
    <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
  </global-forwards>
  <action-mappings>
    <!-- 请求的 URL 对应于< action>中的 path 属性-->
    <action path="/helloWorld" type="com.gc.action.HelloWorldAction">
      <!-- forward 表示执行完毕后返回的页面-->
      <forward name="index" path="/WEB-INF/jsp/index.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

代码说明:

- <global-forwards>中, forward 的属性 name 表示全局转发的名称, path 表示全局转发的相对路径。
- forward 除了 name 和 path 属性外, 还可以设定 redirect 属性。如果 redirect 属性设置为 true, 则 ActionServlet 会使用 sendRedirect()方法来转发资源。

2. 定义 ActionMapping

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
  <!-- 全局路径-->
  <global-forwards>
    <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
  </global-forwards>
  <action-mappings>
    <!-- 请求的 URL 对应于< action>中的 path 属性-->
    <action path="/helloWorld" type="com.gc.action.HelloWorldAction">
      <!-- forward 表示执行完毕后返回的页面-->
      <forward name="index" path="/WEB-INF/jsp/index.jsp"/>
    </action>
    <action path="/regedit" type="com.gc.action.RegeditAction">
      <!-- forward 表示执行完毕后返回的页面-->
      <forward name="regedit" path="/WEB-INF/jsp/regedit.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```


代码说明:

- ❑ <action-mappings>中的每一个<action>都对应一个 ActionMapping 对象。
- ❑ <action>除了 path 和 type 属性外, 还可以设定很多属性。name 属性, 用来表示与 Action 相关联的 ActionForm; scope 属性, 用来表示 ActionForm 的作用域; prefix 属性, 用来表示 ActionForm 的前缀; suffix 属性, 用来表示 ActionForm 的后缀; input 属性, 用来表示当 ActionForm 发生错误时, 必须返回的表单路径; unknown 属性, 设为 true 时, 所有没有定义的 ActionMapping 的操作都将转到这里来; validate 属性, 设为 true 时, 则在执行 Action 前, 会调用 ActionForm 的 validate()方法来检查输入值。

3. 定义 ActionForm

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
  <!--全局路径-->
  <global-forwards>
    <!-- forward 表示执行完毕后返回的页面-->
    <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
  </global-forwards>
  <form-beans>
    <form-bean name="user" type="com.gc.action.User"/>
  </form-beans>
  <action-mappings>
    <!--请求的 URL 对应于< action>中的 path 属性-->
    <action path="/helloWorld" type="com.gc.action.HelloWorldAction">
      <!-- forward 表示执行完毕后返回的页面-->
      <forward name="index" path="/WEB-INF/jsp/index.jsp"/>
    </action>
    <action path="/regedit" type="com.gc.action.RegeditAction" name="user">
      <!-- forward 表示执行完毕后返回的页面-->
      <forward name="regedit" path="/WEB-INF/jsp/regedit.jsp"/>
      <forward name="success" path="/WEB-INF/jsp/success.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

代码说明:

- ❑ <form-beans>用来定义用到的 ActionForm, <form-bean>的属性包括 name 和 type。
- ❑ 在 path 属性为 regedit 的 action 中, 定义 name 属性为 user。

4. 定义数据源

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
```

```
<!--全局路径-->
<global-forwards>
    <forward name="welcome" path="/WEB-INF/jsp/welcome.jsp"/>
</global-forwards>
<form-beans>
    <form-bean name="user" type="com.gc.action.User"/>
</form-beans>
<action-mappings>
    <!--请求的 URL 对应于< action>中的 path 属性-->
    <action path="/helloWorld" type="com.gc.action.HelloWorldAction">
        <forward name="index" path="/WEB-INF/jsp/index.jsp"/>
    </action>
    <action path="/regedit" type="com.gc.action.RegeditAction" name="user">
        <!-- forward 表示执行完毕后返回的页面-->
        <forward name="regedit" path="/WEB-INF/jsp/regedit.jsp"/>
        <forward name="success" path="/WEB-INF/jsp/success.jsp"/>
    </action>
</action-mappings>
<!--在 Action 中可以通过 getDataSource(request, "conPool")来取得 DataSource-->
<data-sources>
    <data-source key="conPool" type="org.apache.commons.dbcp.BasicDataSource"
        <set-property
            autoCommit="true"
            description="数据库连接池"
            driverClass="com.microsoft.jdbc.sqlserver.SQLServerDriver"
            maxCount="150"
            minCount="20"
            url="jdbc:microsoft:sqlserver://localhost:1433/stdb"
            username="admin"
            password="admin" />
    </data-source>
</data-sources>
</struts-config>
```

代码说明:

- ❑ 定义<data-source>的属性 key 为 conPool, 则在 Action 中可以通过 getDataSource(request, "conPool")来取得 DataSource。
- ❑ 定义<data-source>的属性 type 为 org.apache.commons.dbcp.BasicDataSource, 表示使用的是 DBCP 连接池。
- ❑ 在<set-property>中定义了与数据库相关的属性, 因为相关属性和其他数据库类似, 这里就不再解释了。

10.4.4 ActionForm (数据存储)

ActionForm 就是一个简单的 JavaBean, 每一个 ActionForm 都必须继承于 org.apache.struts.action.ActionForm。除了提供简单的 set 和 get 方法外, 还提供了 reset 和 validate 方法。

前面已经讲过在配置文档中配置 ActionForm 的示例,这里只展示一个使用 ActionForm 的示例。

下面在前面示例中增加一个提交按钮,来展示 ActionForm 的实现方法。实现思路是:首先新建一个输入提交页面,增加提交按钮,接着新建一个 ActionForm 为 HelloWorld.java,然后修改配置文档,接着修改控制器,测试该程序,最后增加验证功能。具体步骤如下:

(1) 新建一个输入提交页面 input.jsp, 增加提交按钮, 放在 myStruts/jsp 目录下。示例代码如下:

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>ActionForm 的 Struts 实例</title></head>
<body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
        欢迎语 <input type="text" name="msg" value=""/><br>
        <input type="submit" name="method" value="提交"/>
    </form>
</body>
</html>
```

(2) 输出提交内容的页面 show.jsp, 放在 myStruts/WEB-INF/jsp 目录下。示例代码如下:

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title> ActionForm 的 Struts 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HelloWorld" action="/ myStruts /helloWorld.do" method="post">
        您输入的欢迎语是"${helloWorld}"<br>
    </form>
</body>
</html>
```

(3) 在 com.gc.action 包上, 新建一个 ActionForm 为 HelloWorld.java。HelloWorld.java 的示例代码如下:

```
//***** HelloWorld.java *****
package com.gc.action;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
//该类继承 ActionForm
public class HelloWorld extends ActionForm {
    //定义变量 msg
    private String msg = null;
    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```



```

    }
    public String getMsg() {
        return this.msg;
    }
    //重置变量
    public void reset(ActionMapping mapping, HttpServletRequest req) {
        this.msg = null;
    }
}

```

(4) 修改配置文件 struts-config.xml。示例代码如下：

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <!--定义 formbean-->
    <form-beans>
        <form-bean name="helloWorld" type="com.gc.action.HelloWorld"/>
    </form-beans>
    <action-mappings>
        <!--请求的 URL 对应于< action>中的 path 属性-->
        <action path="/helloWorld" type="com.gc.action.HelloWorldAction" name="helloWorld">
            <forward name="show" path="/WEB-INF/jsp/show.jsp"/>
        </action>
        <!--请求的 URL 对应于< action>中的 path 属性-->
        <action
            path="/input"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/jsp/input.jsp"/>
        </action>
    </action-mappings>
</struts-config>

```

(5) 修改控制器代码。HelloWorldAction.java 的示例代码如下：

```

//***** HelloWorldAction.java *****
package com.gc.action;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import org.apache.struts.actions. Action;
public class HelloWorldAction extends Action {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
        request, HttpServletResponse response) throws Exception {
        //将页面提交的进行转换成 form
    }
}

```



```
String msg = ((HelloWorld) form).getMsg();
request.setAttribute("helloWorld", msg);
return mapping.findForward("show");
}
}
```

(6) 运行演示程序。在浏览器的地址栏中输入 `http://localhost:8080/myStruts/input.do`，按 Enter 键即可看到包含有“提交”按钮的页面，如图 10.13 所示：

(7) 在页面的文本框中输入“HelloWorld”，然后单击“提交”按钮，即可看到“HelloWorld”显示在页面上，如图 10.14 所示。



图 10.13 用来输入问候语的页面



图 10.14 “HelloWorld”显示在页面上

10.4.5 DispatchAction（多动作控制器）

和 Spring 中的 `MultiActionController` 类似，Struts 也提供了多动作放在一个 Action 中的功能，就是 `DispatchAction`。如果要实现多动作放在一个 Action 里，则该 Action 必须继承于 `DispatchAction`。下面实现一个多动作控制的示例。读者可以和 Spring 的实现方式作一比较。

下面在前面示例中增加一个新增、修改、删除按钮，来展示 `DispatchAction` 的实现方法。实现思路是：首先修改页面，增加新增、修改、删除按钮，然后修改配置文档，接着修改控制器，使其继承 `DispatchAction`，测试该程序。具体步骤如下：

(1) 编写有新增、修改、删除按钮的页面 `input.jsp`，放在 `myStruts/jsp` 目录下。示例代码如下：

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title>第二个 Struts 实例</title></head>
<body>
  <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
    欢迎语 <input type="text" name="msg" value=""/><br>
    <input type="submit" name="method" value="insert"/>
    <input type="submit" name="method" value="update"/>
    <input type="submit" name="method" value="delete"/>
  </form>
</body>
</html>
```

(2) 输出提交内容的页面 `show.jsp`，还是使用前面示例的，放在 `myStruts/WEB-INF/jsp` 目录下。示例代码如下：

```

<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>第二个 Struts 实例</title></head>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HelloWorld" action="/ myStruts/helloWorld.do" method="post">
        您输入的欢迎语是"${helloWorld}"<br>
    </form>
</body>
</html>

```

(3) 修改配置文档 struts-config.xml。示例代码如下：

```

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

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <action-mappings>
        <action
            path="/helloWorld"
            type="com.gc.action.HelloWorldAction"
            parameter="method">
            <forward name="show" path="/WEB-INF/jsp/show.jsp"/>
        </action>
        <!--请求的 URL 对应于< action>中的 path 属性-->
        <action
            path="/input"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/jsp/input.jsp"/>
    </action-mappings>
</struts-config>

```

(4) 修改控制器代码，使其继承 DispatchAction。HelloWorldAction.java 的示例代码如下：

```

//***** HelloWorldAction.java*****
package com.gc.action;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import org.apache.struts.actions.DispatchAction;
public class HelloWorldAction extends DispatchAction {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //新增一笔数据
    public ActionForward insert(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception {
        String msg = request.getParameter("msg");

```

```
        request.setAttribute("helloWorld", "insert:" + msg);
        return mapping.findForward("show");
    }
    //修改一笔数据
    public ActionForward update(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception {
        String msg = request.getParameter("msg");
        request.setAttribute("helloWorld", "update:" + msg);
        return mapping.findForward("show");
    }
    //删除一笔数据
    public ActionForward delete(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception {
        String msg = request.getParameter("msg");
        request.setAttribute("helloWorld", "delete:" + msg);
        return mapping.findForward("show");
    }
}
```

代码说明：为了区分执行的是 insert、update、delete 方法，分别在问候语前加上“insert:”、“update:”、“delete:”。

(5) 运行演示程序。在浏览器的地址栏中输入 <http://localhost:8080/myStruts/input.do>，按 Enter 键即可看到包含有 insert、update 和 delete 这 3 个按钮的页面，如图 10.15 所示。

(6) 在页面的文本框中输入“HelloWorld”，然后单击 insert 按钮，即可看到“insert: HelloWorld”显示在页面上，如图 10.16 所示。



图 10.15 用来输入问候语的页面



图 10.16 “insert:HelloWorld”显示在页面上

(7) 同样单击 update 或 delete 按钮，也可看到“update:HelloWorld”或“delete:HelloWorld”显示在页面上，分别如图 10.17 和图 10.18 所示。



图 10.17 单击 update 按钮的结果



图 10.18 单击 delete 按钮的结果

10.5 国际化支持

Struts 也有很强大的国际化支持功能，使用方法也很简单。下面在前面 ActionForm 示例的基础上修改该示例进行说明。

实现思路是：首先生成国际化的消息文件，然后添加并配置自定义标签，最后修改原有程序，运行程序，演示示例。具体步骤如下：

(1) 用记事本编写存放信息资源的文档 messages.properties，存放在 myStruts/WEB-INF/src 目录下。messages.properties 的内容如下：

```
title=Struts 实例
submit=提交
error=欢迎语不能为空
input=您输入的欢迎语是
welcome=欢迎语
```

(2) 把 messages.properties 放在 C 盘的根目录下。

(3) 打开 Windows 的“开始”菜单，选择“运行”命令，弹出“运行”对话框，如图 10.19 所示。

(4) 在“运行”对话框中输入“cmd”，然后单击“确定”按钮，弹出“cmd 命令”对话框，如图 10.20 所示。

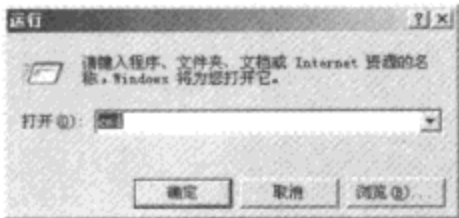


图 10.19 “运行”对话框

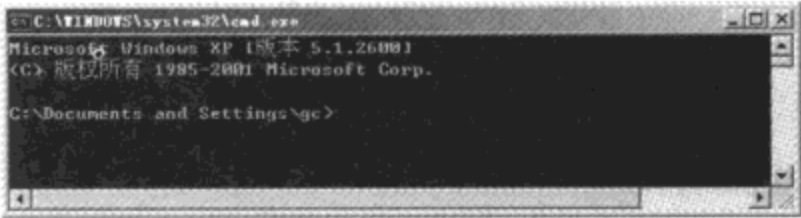


图 10.20 “cmd 命令”对话框

(5) 在“cmd 命令”对话框中输入“cd\”回到 C 盘根目录下。

(6) 再输入“native2ascii messages.properties messages.txt”，然后按 Enter 键，如图 10.21 所示。



图 10.21 输入“native2ascii messages.properties messages.txt”后的“cmd 命令”对话框

(7) 这时在 C 盘的根目录下就会产生一个 messages.txt。messages.txt 的内容如下：

```
title=Struts\u5b9e\u4f8b
submit=\u63d0\u4ea4
error=\u6b22\u8fce\u8bed\u4e0d\u80fd\u4e3a\u7a7a
input=\u60a8\u8f93\u5165\u7684\u6b22\u8fce\u8bed\u662f
welcome=\u6b22\u8fce\u8bed
```



```
input=\u60a8\u8f93\u5165\u7684\u6b22\u8fce\u8bed\u662f
welcome=\u6b22\u8fce\u8bed
```

(8) 将 messages.txt 中的内容复制到 myStruts/WEB-INF/src/messages.properties 中，覆盖原来的内容。

(9) 修改配置文件，增加对国际化的支持。修改配置文档 struts-config.xml，示例代码如下：

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

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
  <!-- 定义 formbean -->
  <form-beans>
    <form-bean name="helloWorld" type="com.gc.action.HelloWorld"/>
  </form-beans>
  <action-mappings>
    <action path="/helloWorld" type="com.gc.action.HelloWorldAction" name="helloWorld">
      <forward name="show" path="/WEB-INF/jsp/show.jsp"/>
    </action>
    <!-- 国际化支持 -->
    <action
      path="/input"
      type="org.apache.struts.actions.ForwardAction"
      parameter="/WEB-INF/jsp/input.jsp"/>
    </action-mappings>
    <!-- 国际化支持 -->
    <message-resources parameter="messages"/>
  </struts-config>
```

代码说明：<message-resources parameter="messages"/>，表示使用 WEB-INF/src 目录下以 messages 开头，扩展名为 properties 的消息文件。

(10) 把 struts-1.2.9-lib 目录下的 struts-html.tld、struts-bean.tld、struts-logic.tld、struts-nested.tld、struts-tiles.tld 复制到 WEB-INF 下，并在 web.xml 里进行定义。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
```

```

        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--拦截所有以 do 结尾的请求-->
    <servlet-mapping>
        <servlet-name>actionServlet</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!--定义该应用使用的 Struts 的 Bean 自定义标签库的路径-->
    <taglib>
        <taglib-uri>/tags/struts-bean</taglib-uri>
        <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
    </taglib>
    <!--定义该应用使用的 Struts 的 html 自定义标签库的路径-->
    <taglib>
        <taglib-uri>/tags/struts-html </taglib-uri>
        <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
    </taglib>
    <!--定义该应用使用的 Struts 的 logic 自定义标签库的路径-->
    <taglib>
        <taglib-uri>/tags/struts-logic </taglib-uri>
        <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
    </taglib>
    <!--定义该应用使用的 Struts 的 nested 自定义标签库的路径-->
    <taglib>
        <taglib-uri>/tags/struts-nested </taglib-uri>
        <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
    </taglib>
    <!--定义该应用使用的 Struts 的 tiles 自定义标签库的路径-->
    <taglib>
        <taglib-uri>/tags/struts-tiles </taglib-uri>
        <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
    </taglib>
</web-app>

```

(11) 增加验证信息。修改 HelloWorld.java, 添加 validate()函数。HelloWorld.java 的示例代码如下:

```

//***** HelloWorld.java*****
package com.gc.action;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
public class HelloWorld extends ActionForm {
    //定义变量 msg
    private String msg = null;
    public void setMsg(String msg) {
        this.msg = msg;
    }
    public String getMsg() {

```

```

        return this.msg;
    }
    public void reset(ActionMapping mapping, HttpServletRequest req) {
        this.msg = null;
    }
    //对用户提交的信息进行验证
    public ActionErrors validate(ActionMapping mapping,
        HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if("").equals(getMsg())) {
            errors.add("msg", new ActionError("error"));
        }
        return errors;
    }
}

```

(12) 修改输入页面 input.jsp。input.jsp 的示例代码如下：

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>
<body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
        <bean:message key="welcome"/><input type="text" name="msg" value=""/><br>
        <input type="submit" name="method" value="<bean:message key="submit"/>"/>
    </form>
</body>
</html:html>

```

(13) 修改显示页面 show.jsp。show.jsp 的示例代码如下：

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>
<%

```



```
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
        <bean:message key="input"/> "${helloWorld}" <br>
    </form>
</body>
</html:html>
```

(14) 运行演示程序。在浏览器的地址栏中输入 `http://localhost:8080/myStruts/input.do`，按 Enter 键即可看到包含有“提交”按钮的页面，如图 10.22 所示。

(15) 在页面的文本框中输入“HelloWorld”，然后单击“提交”按钮，即可看到“HelloWorld”显示在页面上，如图 10.23 所示。



图 10.22 用来输入问候语的页面



图 10.23 “HelloWorld”显示在页面上

(16) 从上面可以看到和前面的示例显示效果一样。

(17) 如果不输入欢迎语，然后单击“提交”按钮，即可看到“欢迎语不能为空”的错误提示，如图 10.24 所示。



图 10.24 “欢迎语不能为空”的错误提示

10.6 Struts 的自定义标签

在 Struts 中，自定义标签占了很大一部分比例。Struts 提供了大量的自定义标签来帮助开发人员实现其功能。主要分为 4 种类型的标签：Bean 标签、Logic 标签、Html 标签和 Template 标签。

Struts 的标签功能很强大，但是耦合性太强，所以这里只是简单讲解，有关 Struts 标签更详细的使用方法可参考相关专门介绍 Struts 的书籍。

10.6.1 Bean 标签

Struts 提供了多种自定义标签用来在 JSP 页中处理 JavaBean。它们统一被封装在一个 Bean 标签库中，struts-bean.tld 是该标签库的描述器，所以如果要想实现该标签，则要将 struts-bean.tld 复制到 WEB-INF/lib 下，并在 web.xml 里进行定义。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!--初始化参数-->
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>actionServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!--定义该应用使用的 Struts 的自定义标签库的路径-->
  <taglib>
    <taglib-uri>/tags/struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/lib/struts-bean.tld</taglib-location>
  </taglib>
</web-app>
```

如果要在 JSP 中使用 Bean 标签库，则要在 JSP 的开头声明和加载 Bean 标签库。示例代码如下：

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

代码说明：uri 属性表示唯一标识标签库描述符（tld）的 uri，在标签库描述符中描述了 uri。这个 uri 可以是直接或者非直接的。prefix 属性定义了区分指定标签库所定义的标签与其他标签库提供的标签的前缀。

Struts 自定义标签是用 XML 语法编写的。它们有一个开始标签和结束标签，可能还有正文。示例代码如下：

```
<bean:tag>
  正文
</bean:tag>
```

Bean 标签库和 JavaBean 有很强的关联性，它在 JSP 和 JavaBean 之间提供了一个接口。下面介绍 Bean 标签中的几个元素：message、write、size。

(1) <bean:message>：用于输出本地化的文本内容，它的 key 属性指定消息 key，这个 key 值和国际化资源文件一致。示例代码如下：

如果在国际化资源文件中的定义：

```
error.name = 用户名错误{0}
```

那么标签的使用：

```
<bean:message key="error.name" arg0="gf"/>
```

结果在 JSP 页面的显示：

```
用户名错误 gf
```

但是在资源文件中最多只支持 4 个参数。

(2) <bean:write>：用于输出 JavaBean 的属性值。示例代码如下：

```
<bean:write name="user" property="name"/>
```

上面示例代码的意思是：输出 user 对象中 name 属性的值。

(3) <bean:size>：得到存储在 array、collection 或 map 中的数目大小。示例代码如下：

```
<bean:size id="count" name="userMap" />
```

上面示例代码的意思是：输出类型为 map 的 userMap 的大小。

10.6.2 Logic 标签

Struts 的 Logic 标签可以根据特定的逻辑条件来判断网页的内容，或者循环遍历集合元素。它的功能主要是进行运算比较，判断内容的存在，进行字符串的匹配，循环遍历集合，进行请求转发和重定向等。它们统一被封装在一个 Bean 标签库中，struts-logic.tld 是该标签库的描述器，所以如果要实现该标签，则要将 struts-logic.tld 复制到 WEB-INF/lib 下，并在 web.xml 里进行定义。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!--初始化参数-->
    <init-param>
      <param-name>config</param-name>
```

```

        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>actionServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
<!--定义该应用使用的 Struts 的 logic 自定义标签库的路径-->
<taglib>
    <taglib-uri>/tags/struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/lib/struts-logic.tld</taglib-location>
</taglib>
</web-app>

```

如果要在 JSP 中使用 Logic 标签库，则要在 JSP 的开头声明和加载 Logic 标签库。示例代码如下：

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

下面介绍 Logic 标签中的几个元素：equal、match、empty、present 和 iterate。

(1) <logic:equal>：用于判断变量是否等于指定的常量。示例代码如下：

如果在国际化资源文件中的定义：

```
<logic:equal name="name" value="gf"/>
```

上面示例代码的意思是：变量 name 的值是否等于指定的常量 gf，相等则返回 true，否则返回 false。

(2) <logic:match>：用于判断变量中是否包含常量字符串。示例代码如下：

```
<logic:match name="name" value="gf"/>
```

上面示例代码的意思是：变量 name 的值是否包含指定的常量 gf，包含则返回 true，否则返回 false。

(3) <logic:empty>：用于判断指定的字符串变量是否为 null。示例代码如下：

```
< logic:empty name="name" />
```

上面示例代码的意思是：字符串变量 name 的值是否为 null，是则返回 true，否则返回 false。

(4) <logic:present>：用来判断 JavaBean 在特定的范围内是否存在。示例代码如下：

```

<logic:present name="user" scope="request">
    <bean:write name="user" property="name" />!<p>
</logic:present>

```

上面示例代码的意思是：判断在 request 范围内是否存在 user 对象，如果存在，就输出 user 的 name 属性值。

(5) <logic:iterate>：用来处理在页面上输出集合类。示例代码如下：

```

<logic:iterate id="user" name="list" type="com.gc.vo.User ">
    <bean:write name="user" property="name"/>

```

```
<bean:write name="user" property="password"/>
</logic:iterate>
```

上面示例代码的意思是：输出名称为 list、存储的类型为 com.gc.vo.User 中的对象 user 的 name 和 password 属性。

10.6.3 Html 标签

Html 标签库主要用来显示 Html 元素。它们统一被封装在一个 Html 标签库中，struts-html.tld 是该标签库的描述器，所以如果要想实现该标签，则要将 struts-html.tld 复制到 WEB-INF/lib 下，并在 web.xml 里进行定义。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!--初始化参数-->
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>actionServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!--定义该应用使用的 Struts 的 html 自定义标签库的路径-->
  <taglib>
    <taglib-uri>/tags/struts-html</taglib-uri>
    <taglib-location>/WEB-INF/lib/struts-html.tld</taglib-location>
  </taglib>
</web-app>
```

如果要在 JSP 中使用 Html 标签库，则要在 JSP 的开头声明和加载 Html 标签库。示例代码如下：

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

下面介绍 Html 标签中的几个元素：errors、form、select。

(1) <html:errors>：用于显示系统中产生的错误消息。示例代码如下：

```
<html:errors/>
```


上面示例代码的意思是：输出错误信息。

(2) `<html:form>`：用于创建 Html 表单，它能够把 Html 表单的字段和 ActionForm Bean 的属性关联起来。示例代码如下：

```
<html:form action="/login" >
    用户名: <html:text property="userName"/><br>
    密码: <html:text property="password"/><br>
    <html:submit/>
</html:form>
```

上面示例代码的意思是：输出一个表单，有用户名和密码，还有一个“提交”按钮。

(3) `<html:select>`：用于显示下拉选单。示例代码如下：

```
<html:select property="name" size="3">
    <html:option value="1">gf</html:option>
    <html:option value="2">gd</html:option>
    <html:option value="3">gc</html:option>
</html:select>
```

上面示例代码的意思是：输出一个下拉选单，共有 3 个选项，分别是 gf、gd、gc。

其实关于 Struts 的标签还有很多，因为 Struts 标签的耦合性太大，所以这里只简单介绍 Bean、Logic、Html 这 3 种标签。如果读者有更多的需求，可参考专业介绍 Struts 的书籍。

10.7 Spring 与 Struts 整合的 3 种方式

前面的讲解都是为本节做准备的，因为本书主要是讲解 Spring，所以对 Struts 只是作一个粗略的介绍，但这些对于读者了解并使用 Struts 已经足够了。虽然使用 Spring 即可实现 Struts 所提供的功能，但由于市场上熟悉 Struts 的开发人员还是很多的，所以 Spring 也提供了对 Struts 的支持，开发人员可以使用 Struts 代替 Spring MVC。

既然要使用 Struts 代替 Spring MVC，那肯定需要用到 Struts 的 `ActionServlet`，这里出现的问题就是 Struts 的 `ActionServlet` 如何装载 Spring 的应用程序环境，方法就是使用 Spring 提供的 `org.springframework.web.struts.ContextLoaderPlugIn`，在 Struts 的配置文件 `struts-config.xml` 中，注册 `ContextLoaderPlugIn` 插件即可。示例代码如下：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation" value="/WEB-INF/config.xml" />
</plug-in>
```

代码说明：`/WEB-INF/config.xml`，指的就是加载 `WEB-INF` 目录下 Spring 的配置文件 `config.xml`。

Spring 与 Struts 整合，主要有 3 种方式：通过 Spring 的 `ActionSupport` 类、通过 Spring 的 `DelegatingRequestProcessor` 类、通过 Spring 的 `DelegatingActionProxy` 类。下面分别来进行讲解。

10.7.1 通过 Spring 的 ActionSupport 类

通过 Spring 的 ActionSupport 类, 开发人员可以很方便地整合 Spring 和 Struts。方法就是: Action 不再继承 Struts 提供的 Action, 而是继承 Spring 提供的 ActionSupport, 然后在 Action 中获得 Spring 的 ApplicationContext。在前面使用 Struts 表单示例的基础上进行修改, 整合 Spring 与 Struts。具体实现步骤如下:

(1) 把 spring.jar 放在 myStruts/WEB-INF/lib/下, 并加入 CLASSPATH 中。

(2) 修改 Struts 的配置文件 struts-config.xml, 注册 ContextLoaderPlugIn 插件。struts-config.xml 的示例代码如下:

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

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <!-- 定义 formbean -->
    <form-beans>
        <form-bean name="helloWorld" type="com.gc.action.HelloWorld"/>
    </form-beans>
    <action-mappings>
    <!-- path 指的是 URL 路径 -->
        <action path="/helloWorld" type="com.gc.action.HelloWorldAction" name="helloWorld">
            <forward name="show" path="/WEB-INF/jsp/show.jsp"/>
        </action>
        <action
            path="/input"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/jsp/input.jsp"/>
        </action-mappings>
    <!-- 注册插件 -->
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation" value="/WEB-INF/config.xml" />
    </plug-in>
</struts-config>
```

(3) 编写 Spring 的配置文件 config.xml, 放在 WEB-INF 目录下。config.xml 的示例代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="helloWorldService" class="com.gc.service.impl.HelloWorldServiceImpl">
    </bean>
</beans>
```

(4) 使用前面介绍的方法在 Eclipse 中建立包 com.gc.service 和 com.gc.service.impl。

(5) 在包 com.gc.service 中建立类 HelloWorldService。HelloWorldService.java 的示例代码如下：

```
//***** HelloWorldService.java*****
package com.gc.service;
import com.gc.action.HelloWorld;
public interface HelloWorldService {
    public abstract String addMsg(HelloWorld helloWorld);
}
```

(6) 在包 com.gc.service.impl 中建立类 HelloWorldServiceImpl，实现 HelloWorldService 接口。HelloWorldServiceImpl.java 的示例代码如下：

```
//***** HelloWorldServiceImpl.java*****
import com.gc.action.HelloWorld;
import com.gc.service.HelloWorldService;
public class HelloWorldServiceImpl implements HelloWorldService {
    public String addMsg(HelloWorld helloWorld) {
        helloWorld.setMsg("欢迎使用 Spring");
        return helloWorld.getMsg();
    }
}
```

(7) 修改 HelloWorldAction，使其继承 ActionSupport。HelloWorldAction.java 的示例代码如下：

```
//***** HelloWorldAction.java*****
package com.gc.action;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import org.apache.struts.actions.Action;
import org.springframework.context.ApplicationContext;
import org.springframework.web.struts.ActionSupport;
import com.gc.service.HelloWorldService;
public class HelloWorldAction extends ActionSupport {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
        request, HttpServletResponse response) throws Exception {
        //通过 ApplicationContext 获取配置文件
        ApplicationContext ctx = getWebApplicationContext();
        HelloWorldService helloWorldService =
        (HelloWorldService)ctx.getBean("helloWorldService");
        //设定消息
    }
}
```



```

String msg = helloWorldService.addMsg((HelloWorld)form);
request.setAttribute("helloWorld", msg);
return mapping.findForward("show");
}
}

```

(8) 仍然使用原来的输入页面 input.jsp。input.jsp 的示例代码如下：

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>
<body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
        <bean:message key="welcome"/><input type="text" name="msg" value=""/><br>
        <input type="submit" name="method" value="<bean:message key="submit"/>" />
    </form>
</body>
</html:html>

```

(9) 仍然使用原来的显示页面 show.jsp。show.jsp 的示例代码如下：

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>
<%
String str = (String)request.getAttribute("helloWorld");
%>
<body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
        <bean:message key="input"/>"${helloWorld}"<br>
    </form>
</body>
</html:html>

```

(10) 启动 Tomcat，在浏览器的地址栏中输入 <http://localhost:8080/myStruts/input.do>，

按 Enter 键即可看到包含有“提交”按钮的页面，如图 10.25 所示：

（11）在页面的文本框中输入“HelloWorld”，然后单击“提交”按钮，即可看到“欢迎使用 Spring HelloWorld”显示在页面上，如图 10.26 所示。



图 10.25 用来输入问候语的页面



图 10.26 “欢迎使用 Spring HelloWorld” 显示在页面上

通过上面的示例可以看出，只需在 Struts 的配置文件 struts-config.xml 中注册 Spring 的 ContextLoaderPlugIn 插件，然后使 Action 继承 Spring 提供的 ActionSupport 类，即可实现 Spring 与 Struts 的整合。

上面这种方法，虽然方便，但缺点也是很明显的，那就是 Struts 的 Action 与 Spring 耦合在一起，并且 Struts 的 Action 不在 Spring 的控制之中，这样如果更换别的框架，或想使用 Spring 的 AOP 都是比较困难的，而且如果是多个动作放在一个 Action 中，则这种方式就无能为力了。下面这种方法解决了这个问题。

10.7.2 通过 Spring 的 DelegatingRequestProcessor 类

在 Struts 中，担任控制器角色的是 ActionServlet，当有请求发送至 ActionServlet 时，ActionServlet 的 doGet()或 doPost()方法会执行 RequestProcessor 的 process()方法。

Spring 提供了 DelegatingRequestProcessor 类，用来代替 Struts 的 RequestProcessor 类，从而把 Struts 的 Action 与 Spring 分离开，并把 Struts 的动作置于 Spring 框架的控制之下。在前面使用 ActionSupport 整合 Spring 和 Struts 示例的基础上进行修改。具体实现步骤如下：

（1）修改 Struts 的配置文件 struts-config.xml，注册 ContextLoaderPlugIn 插件，并使用 Spring 的 DelegatingRequestProcessor 代替 Struts 的 RequestProcessor。struts-config.xml 的示例代码如下：

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

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
  <!--定义 formbean-->
  <form-beans>
    <form-bean name="helloWorld" type="com.gc.action.HelloWorld"/>
  </form-beans>
  <action-mappings>
```

```

<!--path 表示 URL 的路径-->
<action path="/helloWorld" type="com.gc.action.HelloWorldAction" name="helloWorld">
    <forward name="show" path="/WEB-INF/jsp/show.jsp"/>
</action>
<!--path 表示 URL 的路径-->
<action
    path="/input"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/jsp/input.jsp"/>
</action-mappings>
<!--注册 ContextLoaderPlugin 插件-->
<controller processorClass="org.springframework.web.struts.DelegatingRequestProcessor"/>
<plug-in className="org.springframework.web.struts.ContextLoaderPlugin">
    <set-property property="contextConfigLocation" value="/WEB-INF/config.xml" />
</plug-in>
</struts-config>

```

(2) 修改前面放在 WEB-INF 目录下的 Spring 配置文件 config.xml, 新建 Bean 和 Struts 的配置文件中的动作对应。config.xml 的示例代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="helloWorldService" class="com.gc.service.impl.HelloWorldServiceImpl">
    </bean>
    <!--注意和 struts 路径的对应-->
    <bean name="/helloWorld" class="com.gc.action.HelloWorldAction">
        <property name="helloWorldService">
            <ref bean="helloWorldService"/>
        </property>
    </bean>
</beans>

```

(3) 使用原来的 HelloWorldService.java。示例代码如下:

```

//***** HelloWorldService.java*****
package com.gc.service;
import com.gc.action.HelloWorld;
//定义一个接口
public interface HelloWorldService {
    public abstract String addMsg(HelloWorld helloWorld);
}

```

(4) 使用原来的 HelloWorldServiceImpl.java。示例代码如下:

```

//***** HelloWorldServiceImpl.java*****
import com.gc.action.HelloWorld;
import com.gc.service.HelloWorldService;
public class HelloWorldServiceImpl implements HelloWorldService {
    public String addMsg(HelloWorld helloWorld) {
        helloWorld.setMsg("欢迎使用 Spring");
    }
}

```

```

        return helloWorld.getMsg();
    }
}

```

(5) 修改 HelloWorldAction, 仍然使其继承 Struts 提供的 Action。HelloWorldAction.java 的示例代码如下:

```

//***** HelloWorldAction.java *****
package com.gc.action;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import org.apache.struts.actions.Action;
import com.gc.service.HelloWorldService;

public class HelloWorldAction extends Action {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //依赖注入
    private HelloWorldService helloWorldService;
    public HelloWorldService getHelloWorldService () {
        return helloWorldService;
    }
    public void setHelloWorldService (HelloWorldService helloWorldService) {
        this.helloWorldService = helloWorldService;
    }
    //控制器都会执行 execute 方法
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
                                request, HttpServletResponse response) throws Exception {
        String msg = getHelloWorldService().addMsg((HelloWorld)form);
        request.setAttribute("helloWorld", msg);
        return mapping.findForward("show");
    }
}

```

(6) 仍然使用原来的输入页面 input.jsp。input.jsp 的示例代码如下:

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>

```



```
<body>
  <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
    <bean:message key="welcome"/><input type="text" name="msg" value=""/><br>
    <input type="submit" name="method" value="<bean:message key="submit"/>" />
  </form>
</body>
</html:html>
```

(7) 仍然使用原来的显示页面 show.jsp。show.jsp 的示例代码如下：

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ page contentType="text/html; charset=GBK" %>
<html:html locale="true">
  <head>
    <title><bean:message key="title"/></title>
    <html:base/>
  </head>
  <html:messages id="msg">
    <bean:write name="msg"/>
  </html:messages>
  <%
String str = (String)request.getAttribute("helloWorld");
%>
  <body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
      <bean:message key="input"/> "${helloWorld}" <br>
    </form>
  </body>
</html:html>
```

(8) 启动 Tomcat，在浏览器的地址栏中输入 `http://localhost:8080/myStruts/input.do`，按 Enter 键即可看到包含有“提交”按钮的页面，如图 10.27 所示：

(9) 在页面的文本框中输入“HelloWorld”，然后单击“提交”按钮，即可看到“欢迎使用 Spring HelloWorld”显示在页面上，如图 10.28 所示。



图 10.27 用来输入问候语的页面

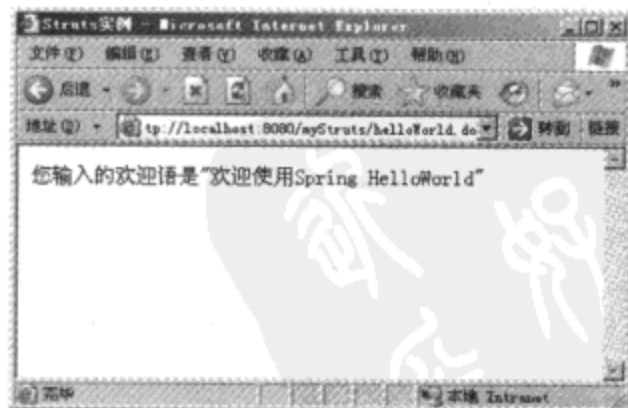


图 10.28 “欢迎使用 Spring HelloWorld”显示在页面上

使用 Spring 的 `DelegatingRequestProcessor` 类代替 Struts 的 `RequestProcessor` 类这种方法，和前面使用 Spring 的 `ActionSupport` 类的方法相比有很大的优势：在 Action 中不再有 Spring 的代码，这样如果要更换为别的 IoC 容器，可以很方便地转换而不用修改代码。

但是这种方法还是有缺点的，那就是开发人员可以自己定义 `RequestProcessor`，这样如果没有使用 Struts 默认的 `RequestProcessor`，则需要手工来整合 Spring 和 Struts 了。

Spring 的 `DelegatingActionProxy` 类提供了另外一种方法来整合 Spring 和 Struts，并避免了这样的问题。

10.7.3 通过 Spring 的 `DelegatingActionProxy` 类

Spring 提供了 `DelegatingActionProxy` 类，用来代理 Struts 中的动作，负责在 Spring 配置文档中查找对应的动作映射，从而把 Struts 的 Action 与 Spring 分离开，并把 Struts 的动作置于 Spring 框架的控制之下。在前面使用 `DelegatingRequestProcessor` 整合 Spring 和 Struts 示例的基础上进行修改，具体实现步骤如下：

(1) 修改 Struts 的配置文件 `struts-config.xml`，注册 `ContextLoaderPlugIn` 插件。在注册动作时，动作的 `type` 属性不再注册为类的实名，而是注册为 Spring 提供的代理类 `DelegatingActionProxy`。`struts-config.xml` 的示例代码如下：

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

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
  <!--定义 formbean-->
  <form-beans>
    <form-bean name="helloWorld" type="com.gc.action.HelloWorld"/>
  </form-beans>
  <action-mappings>
    <!--path 定义的是 URL 的路径-->
    <action path="/helloWorld" type="org.springframework.web.struts.DelegatingActionProxy"
name="helloWorld">
      <forward name="show" path="/WEB-INF/jsp/show.jsp"/>
    </action>
    <!--path 定义的是 URL 的路径-->
    <action
      path="/input"
      type="org.apache.struts.actions.ForwardAction"
      parameter="/WEB-INF/jsp/input.jsp"/>
    </action-mappings>
    <!--注册 ContextLoaderPlugIn 插件-->
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
      <set-property property="contextConfigLocation" value="/WEB-INF/config.xml" />
    </plug-in>
  </struts-config>
```

(2) 使用前面的 Spring 配置文件 `config.xml`，注意定义动作映射的 Bean 和 Struts 的配置文档中的动作的对应关系。`config.xml` 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
```

```
spring-beans.dtd">
<beans>
    <bean id="helloWorldService" class="com.gc.service.impl.HelloWorldServiceImpl">
    </bean>
    <!--注意映射和 Struts 里的动作的对应-->
    <bean name="/helloWorld" class="com.gc.action.HelloWorldAction">
        <property name="helloWorldService">
            <ref bean="helloWorldService"/>
        </property>
    </bean>
</beans>
```

(3) 使用原来的 HelloWorldService.java。示例代码如下：

```
/** ***** HelloWorldService.java ***** */
package com.gc.service;
import com.gc.action.HelloWorld;
public interface HelloWorldService {
    public abstract String addMsg(HelloWorld helloWorld);
}
```

(4) 使用原来的 HelloWorldServiceImpl.java。示例代码如下：

```
/** ***** HelloWorldServiceImpl.java ***** */
import com.gc.action.HelloWorld;
import com.gc.service.HelloWorldService;
public class HelloWorldServiceImpl implements HelloWorldService {
    //用来增加消息
    public String addMsg(HelloWorld helloWorld) {
        helloWorld.setMsg("欢迎使用 Spring");
        return helloWorld.getMsg();
    }
}
```

(5) 使用原来的 HelloWorldAction。HelloWorldAction.java 的示例代码如下：

```
/** ***** HelloWorldAction.java ***** */
package com.gc.action;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import org.apache.struts.actions.Action;
import com.gc.service.HelloWorldService;
public class HelloWorldAction extends Action {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //依赖注入
    private HelloWorldService helloWorldService;
```

```

    public HelloWorldService getHelloWorldService () {
        return helloWorldService;
    }
    public void setHelloWorldService (HelloWorldService helloWorldService) {
        this.helloWorldService = helloWorldService;
    }
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
                                request, HttpServletResponse response) throws Exception {
        //通过 addMsg 方法设定 msg
        String msg = getHelloWorldService().addMsg((HelloWorld)form);
        request.setAttribute("helloWorld", msg);
        return mapping.findForward("show");
    }
}

```

(6) 仍然使用原来的输入页面 input.jsp。input.jsp 的示例代码如下：

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>
<body>
    <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
        <bean:message key="welcome"/><input type="text" name="msg" value=""/><br>
        <input type="submit" name="method" value="<bean:message key="submit"/>"/>
    </form>
</body>
</html:html>

```

(7) 仍然使用原来的显示页面 show.jsp。show.jsp 的示例代码如下：

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@page contentType="text/html; charset=GBK"%>
<html:html locale="true">
<head>
<title><bean:message key="title"/></title>
<html:base/>
</head>
<html:messages id="msg">
    <bean:write name="msg"/>
</html:messages>
<%
String str = (String)request.getAttribute("helloWorld");
%>

```



```
<body>
  <form name="HelloWorld" action="/myStruts/helloWorld.do" method="post">
    <bean:message key="input"/> "${helloWorld}" <br>
  </form>
</body>
</html:html>
```

(8) 启动 Tomcat, 在浏览器的地址栏中输入 `http://localhost:8080/myStruts/input.do`, 按 Enter 键即可看到包含有“提交”按钮的页面, 如图 10.29 所示。

(9) 在页面的文本框中输入“HelloWorld”, 然后单击“提交”按钮, 即可看到“欢迎使用 Spring HelloWorld”显示在页面上, 如图 10.30 所示。



图 10.29 用来输入问候语的页面

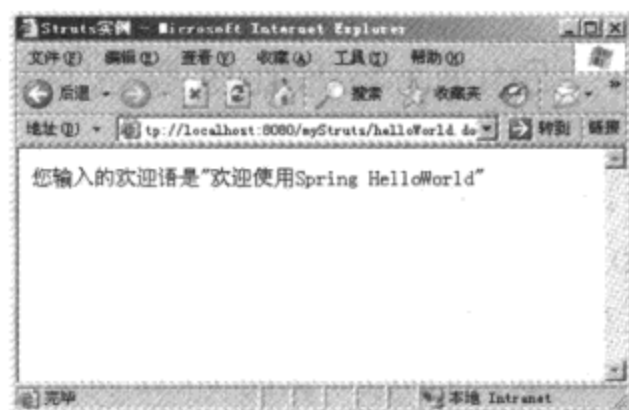


图 10.30 “欢迎使用 Spring HelloWorld”显示在页面上

10.7.4 比较 3 种整合方式

这 3 种整合方式都需要在 Struts 的配置文件 `struts-config.xml` 中注册 Spring 的 `ContextLoaderPlugIn` 插件。

- ❑ 使用 Spring 提供的 `ActionSupport` 类, 只需要使 Action 继承 Spring 提供的 `ActionSupport` 类, 即可实现 Spring 与 Struts 的整合, 使用起来简单方便, 但是 Struts 的 Action 与 Spring 耦合在一起, 并且 Struts 的 Action 不在 Spring 的控制之中, 这样如果更换成别的 IoC 容器, 或想使用 Spring 的 AOP 都是比较困难的, 而且如果是多个动作放在一个 Action 中, 则这种方式就无能为力了。
- ❑ 使用 Spring 的 `DelegatingRequestProcessor` 类, 需要在 Struts 的配置文件中, 使 `DelegatingRequestProcessor` 代替 `RequestProcessor` 类, 并在 Spring 的配置文件中定义和 Struts 配置文件中动作映射对应的 Bean, 使用这种方式在 Action 中不再有 Spring 的代码, 这样如果要更换为别的 IoC 容器, 可以很方便地转换而不用修改代码, 但这种方式的缺点是依赖于 Struts 的 `RequestProcessor` 类。
- ❑ 使用 Spring 的 `DelegatingActionProxy` 类, 需要在 Struts 的配置文件中, 定义动作映射的 `type` 属性为 `DelegatingActionProxy` 而不是类的实际名称, 并在 Spring 的配置文件中定义和 Struts 配置文件中动作映射对应的 Bean, 这种方式和 Struts 的耦合性最小。

三者中, 使用 Spring 的 `DelegatingActionProxy` 类来整合 Spring 和 Struts 的方式最为强大和灵活。

10.8 采用第 3 种整合方式编写一个用户注册的例子

这个示例主要实现的功能是：用户登录到注册页面，填写要注册的用户名和密码，然后单击“提交”按钮，如果注册成功，则返回到注册成功的画面，并显示注册的相关信息，如果注册不成功，则返回到原来注册的页面，并提示注册不成功。

这个示例采用第三种整合方式：使用 Spring 的 `DelegatingActionProxy` 类来整合 Spring 和 Struts。把这个整合项目命名为 mySS，具体实现步骤如下。

10.8.1 Spring 与 Struts 整合环境的配置

(1) 运行 Eclipse，选择 `File→New→Project` 命令，Eclipse 将弹出 New Project 对话框，如图 10.31 所示。

(2) 用鼠标选择列表框中 Java 下的 Tomcat Project，然后单击 Next 按钮，弹出 New Tomcat Project 对话框，如图 10.32 所示。



图 10.31 New Project 对话框

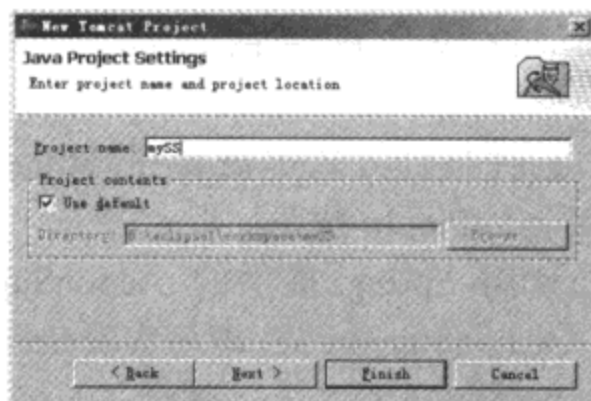


图 10.32 New Tomcat Project 对话框

(3) 在 New Tomcat Project 对话框的 Project name 文本框中输入 mySS，然后单击 Finish 按钮，项目即建立成功，mySS 的目录结构如图 10.33 所示。



图 10.33 mySS 的目录结构

(4) 把 `struts.jar`、`commons-beanutils.jar`、`commons-digester.jar`、`log4j-1.2.9.jar`、`commons-logging.jar` 和 `spring.jar` 这 6 个 jar 放在 `/WEB-INF/lib/` 下复制到 `mySS/WEB-INF/lib` 目录下，

即 CLASSPATH 中。

(5) 用 Windows 自带的文本编辑器，建立一个文件 log4j.properties，把 log4j.properties 放到 mySS\WEB-INF\src 目录下。

(6) 编辑 log4j.properties 文件，内容如下：

```
log4j.rootLogger=DEBUG,stdout,R

#定义 log4j 的显示方式
log4j.appender.A1=org.apache.log4j.RollingFileAppender
#指定日志输入文件的名称
log4j.appender.A1.File=org.log
#指定日志输入文件的大小
log4j.appender.A1.MaxFileSize=500KB
log4j.appender.A1.MaxBackupIndex=50
log4j.appender.A1.Append=true
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
log4j.appender.A2=org.apache.log4j.RollingFileAppender
log4j.appender.A2.File=gc.log
log4j.appender.A2.MaxFileSize=500KB
log4j.appender.A2.MaxBackupIndex=50
log4j.appender.A2.Append=true
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
#-----stdout-----
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#-----R-----
#log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
#this log file will be stored in web server's /bin directory,modify to your path which want to store.
log4j.appender.R.File=gf.log
#log4j.appender.R.datePattern='.'yyyy-MM-dd-HH-mm
log4j.appender.R.datePattern='.'yyyy-MM-dd
log4j.appender.R.append=true
## Keep one backup file
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n
```

(7) 在 mySS 上单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，将弹出 Properties for mySS 对话框，如图 10.34 所示。

(8) 在 Properties for mySS 对话框中，选择对话框左边列表框中的 Java Build Path。

(9) 在 Libraries 选项卡中，单击 Add JARs...按钮，弹出 JAR Selection 对话框，如图 10.35 所示。

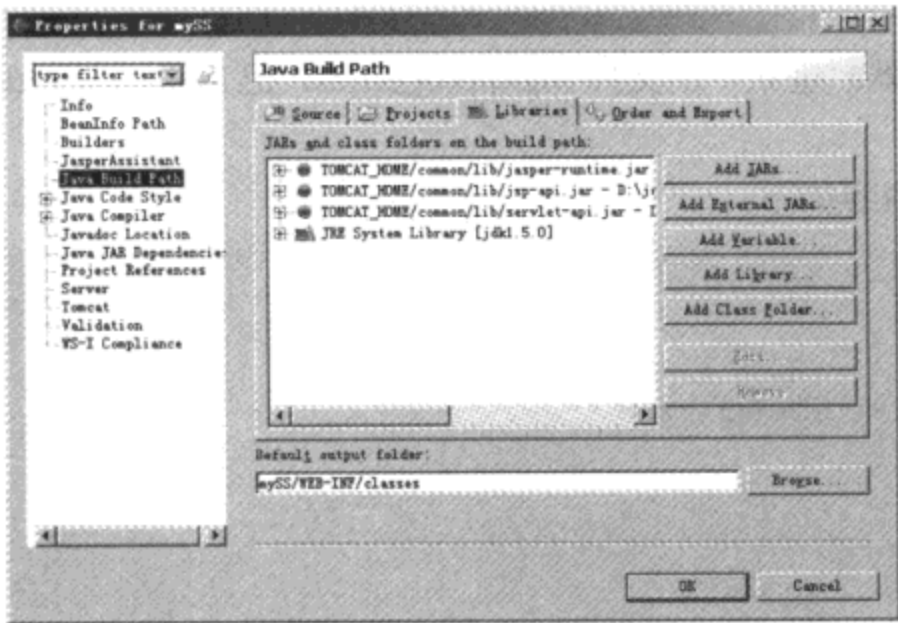


图 10.34 Properties for mySS 对话框



图 10.35 JAR Selection 对话框

(10) 在 JAR Selection 对话框中，打开列表框中 mySS 一直到 lib 目录下出现 6 个 jar: struts.jar、commons-beanutils.jar、commons-digester.jar、commons-logging.jar、log4j-1.2.9.jar 和 spring.jar。

(11) 按住 Ctrl 键选中这 6 个 jar，然后单击 OK 按钮，返回到 Properties for mySS 对话框，如图 10.36 所示。

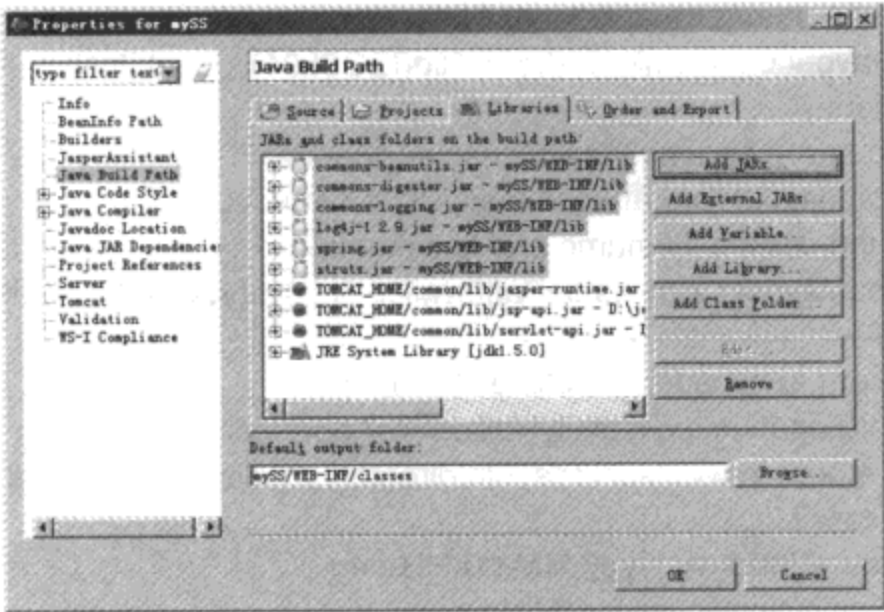


图 10.36 Properties for mySS 对话框

(12) 在 Properties for mySS 对话框中单击 OK 按钮，即可完成对 Spring 和 Struts 的配置。

(13) 再次在 mySS 上单击鼠标右键，在弹出的快捷菜单中选择 New→Package 命令，将弹出 New Java Package 对话框，如图 10.37 所示。

(14) 在 New Java Package 对话框的 Name 文本框中输入“com.gc.action”，然后单击 Finish 按钮，即可建立 com.gc.action 包。

(15) 用同样的方法建立 com.gc.service 包、com.gc.service.impl 包和 com.gc.vo 包。

(16) 在 WEB-INF 目录下建立 jsp 文件夹。

(17) 最终配置好 Spring 和 Struts 的 mySS 项目的目录结构如图 10.38 所示。

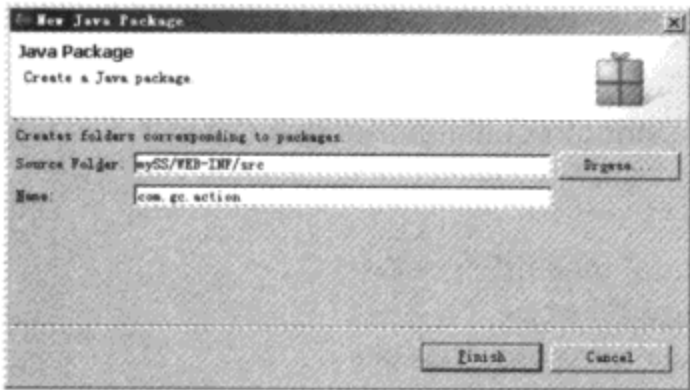


图 10.37 New Java Package 对话框



图 10.38 配置好 Spring 和 Struts 的 mySS 项目的目录结构

10.8.2 编写 web.xml

建立 web.xml 文件，放在 mySS/WEB-INF 目录下。web.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!--初始化参数-->
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!--定义拦截所有以 do 结尾的请求-->
  <servlet-mapping>
    <servlet-name>actionServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>
```

代码说明：/WEB-INF/struts-config.xml，说明 struts 的配置文件在 WEB-INF 目录下。

10.8.3 编写用户注册页面 regedit.jsp

新建一个用户注册页面 regedit.jsp，包含“提交”按钮，放在 mySS/WEB-INF/jsp 目录下。示例代码如下：


```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title>Spring 和 Struts 整合实例</title></head>
<%
String msg = request.getAttribute("msg") == null ? "" : (String)request.getAttribute("msg");
%>
<body> <%= msg%>
    <form name="HelloWorld" action="/mySS/regedit.do" method="post">
        用户名 <input type="text" name="name" value=""/><br>
        密 码 <input type="password" name="password" value=""/><br>
        <input type="submit" name="method" value="提交"/>
    </form>
</body>
</html>
```

代码说明：action="/mySS/regedit.do"，表示在配置文件中要设定的动作映射为 regedit。

10.8.4 编写用户注册成功页面 success.jsp

新建一个用户注册成功页面 success.jsp，放在 mySS/WEB-INF/jsp 目录下。示例代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<html>
<head><title> Spring 和 Struts 整合实例</title></head>
<body>
    ${msg}: 您输入的用户名是: ${user.name}, 密码是: ${user.password}
</body>
</html>
```

10.8.5 编写用户类 User.java

在 com.gc.vo 包上，新建一个 ActionForm 为 User 的用户类。User.java 的示例代码如下：

```
//***** User.java *****
package com.gc.vo;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
//该类继承 ActionForm
public class User extends ActionForm {
    //定义用户名
    private String name = null;
    //定义密码
    private String password = null;
    //设定用户名
    public void setName(String name) {
        this.name = name;
    }
}
```

```
}  
//获取用户名  
public String getName() {  
    return this.name;  
}  
//设定密码  
public void setPassword (String password) {  
    this.password = password;  
}  
//获取  
public String getPassword () {  
    return this.password;  
}  
//重置用户名和密码  
public void reset(ActionMapping mapping, HttpServletRequest req) {  
    this.name = null;  
    this.password = null;  
}  
}
```

10.8.6 编写 Struts 的配置文件 struts-config.xml

新建一个配置文档 struts-config.xml，放在 WEB-INF 目录下。示例代码如下：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>  
  
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration  
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">  
  
<struts-config>  
    <!-- 定义 formbean -->  
    <form-beans>  
        <form-bean name="user" type="com.gc.vo.User"/>  
    </form-beans>  
    <action-mappings>  
        <!-- path 代表 URL 的路径 -->  
        <action path="/regedit" type="org.springframework.web.struts.DelegatingActionProxy"  
name="user">  
            <forward name="success" path="/WEB-INF/jsp/success.jsp"/>  
            <forward name="input" path="/WEB-INF/jsp/regedit.jsp"/>  
        </action>  
        <!-- path 代表 URL 的路径 -->  
        <action  
            path="/input"  
            type="org.apache.struts.actions.ForwardAction"  
            parameter="/WEB-INF/jsp/regedit.jsp"/>  
    </action-mappings>  
    <!-- 注册插件 -->  
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">  
        <set-property property="contextConfigLocation" value="/WEB-INF/config.xml" />  
    </plug-in>  
</struts-config>
```

```
</plug-in>
</struts-config>
```

代码说明：

- ❑ 注册 `org.springframework.web.struts.ContextLoaderPlugIn`，并说明 Spring 的配置文件是 `WEB-INF` 目录下的 `config.xml`。
- ❑ 使用 Spring 提供的代理类 `org.springframework.web.struts.DelegatingActionProxy`。

10.8.7 编写 Spring 的配置文件 config.xml

新建一个 Spring 的配置文件 `config.xml`，放在 `WEB-INF` 目录下。`config.xml` 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="regedit" class="com.gc.service.impl.RegeditImpl">
    </bean>
    <!--注意定义动作映射的 Bean 和 Struts 的配置文件中的动作的对应关系-->
    <bean name="/regedit" class="com.gc.action.RegeditAction">
        <property name="regedit">
            <ref bean="regedit"/>
        </property>
    </bean>
</beans>
```

代码说明：注意定义动作映射的 Bean 和 Struts 的配置文件中的动作的对应关系。

10.8.8 编写控制器 RegeditAction.java

在 `com.gc.action` 包上，新建一个控制器 `RegeditAction` 类。`RegeditAction.java` 的示例代码如下：

```
//***** RegeditAction.java *****
package com.gc.action;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.apache.struts.action.*;
import com.gc.service.Regedit;
import com.gc.vo.User;
public class RegeditAction extends Action {
```



```

private Logger logger = Logger.getLogger(this.getClass().getName());
//通过依赖注入的方式设定
private Regedit regedit;
public Regedit getRegedit () {
    return regedit;
}
public void setRegedit (Regedit regedit) {
    this.regedit = regedit;
}
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
                             request, HttpServletResponse response) throws Exception {
    //保存用户提交的信息
    String msg = getRegedit().saveUser((User)form);
    request.setAttribute("msg", msg);
    request.setAttribute("user", (User)form);
    if ("注册成功".equals(msg)) {
        return mapping.findForward("success");
    } else {
        return mapping.findForward("input");
    }
}
}

```

10.8.9 编写业务逻辑接口 Regedit.java

在 com.gc.service 包上，新建一个业务逻辑接口 Regedit 类。Regedit.java 的示例代码如下：

```

//***** Regedit.java*****
package com.gc.service;
import com.gc.vo.User;
public interface Regedit {
    public abstract String saveUser(User user);
}

```

10.8.10 编写具体的业务逻辑类 RegeditImpl.java

在 com.gc.service.impl 包上，新建一个具体的业务逻辑类 RegeditImpl 类。RegeditImpl.java 的示例代码如下：

```

//***** RegeditImpl.java*****
package com.gc.service.impl;

import com.gc.service.Regedit;
import com.gc.vo.User;
public class RegeditImpl implements Regedit {
    //保存用户

```



```
public String saveUser(User user) {
    String str = "";
    if ("gf".equals(user.getName()) && "12345".equals(user.getPassword())) {
        str = "注册成功";
    } else if ("gf".equals(user.getName())) {
        str = "密码填写错误";
    } else if ("12345".equals(user.getPassword())) {
        str = "用户名填写错误";
    } else {
        str = "用户名和密码都填写错误";
    }
    return str;
}
```

代码说明：如果用户输入用户名为“gf”，密码为“12345”，则表示注册成功。所有代码编写完成后，mySS 的目录结构如图 10.39 所示。

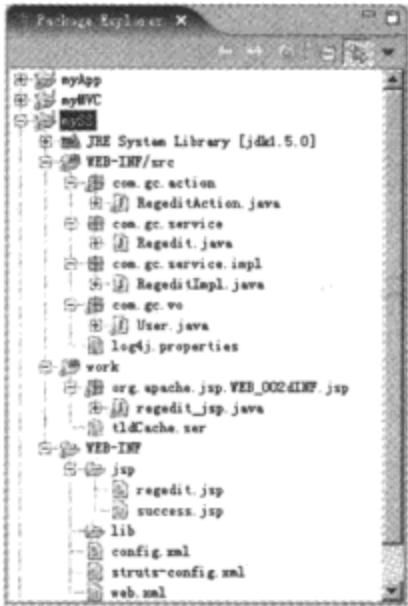


图 10.39 代码完成后 mySS 的目录结构

10.8.11 运行用户注册的例子

- (1) 启动 Tomcat，在浏览器的地址栏中输入 `http://localhost:8080/mySS/input.do`，按 Enter 键即可看到用户注册的页面，如图 10.40 所示。
- (2) 在用户名文本框中输入“gg”，密码输入“123”，然后单击“提交”按钮即可看到提示用户名和密码都填写错误的页面，如图 10.41 所示。
- (3) 在用户名文本框中输入“gf”，密码输入“123”，然后单击“提交”按钮即可看到提示密码填写错误的页面，如图 10.42 所示。
- (4) 在用户名文本框中输入“gg”，密码输入“12345”，然后单击“提交”按钮即可看到提示用户名填写错误的页面，如图 10.43 所示。
- (5) 在用户名文本框中输入“gf”，密码输入“12345”，然后单击“提交”按钮即可看到提示注册成功的页面，如图 10.44 所示。



图 10.40 用户注册的页面



图 10.41 用户名和密码都填写错误的页面



图 10.42 密码填写错误的页面



图 10.43 用户名填写错误的页面



图 10.44 注册成功的页面

10.9 小 结

本章首先讲述了 Struts 的历史、体系，又展示了如何下载和配置 Struts，并通过示例教会读者使用 Struts 的相关功能，接着通过示例的方式介绍了 Spring 和 Struts 整合的 3 种方式，并对这 3 种整合方式进行了比较，最后通过一个实现用户注册的实例，介绍了 Spring 和 Struts 整合的完整过程，从而使读者详细地了解了使用 Spring 和 Struts 共同开发应用程序的过程。

第 11 章 Spring 与 Hibernate 的整合

在当前的 Java 持久层框架中，最流行的 O/R Mapping 产品分别是 Hibernate、JDO 和 TopLink。从 EJB3 和 JDO 联合发表的公开信可以看出，新的持久层规范仍将以 Hibernate 的设计理念为基础，无疑在这 3 种 O/R Mapping 产品中，Hibernate 是最有前途的。本章首先对 Hibernate 进行了介绍，并给出了一个示例使读者快速了解 Hibernate，接着介绍了 Hibernate 的配置方式和映射方法，以及几个辅助工具，然后给出了 Spring 与 Hibernate 的整合方式，并通过实例展示了这种整合方式。

11.1 Hibernate 介绍

这里首先讲一下为什么要使用 ORM (Object/Relational Mapping)。这是因为开发人员使用的技术是面向对象技术，而使用的数据库却是关系型数据库，使用 SQL 带来了很大的麻烦，可是目前面向对象的数据库还没有完全成熟。

那么有什么办法既使用 SQL 又能去掉或减少这些麻烦呢？使用 ORM，通过在对象和关系型之间建立起一座桥梁。这样做的好处有很多，比如可以完全用 OO 的思想去设计，而不是首先去考虑表的结构和关系；降低中间层对数据库的耦合度，方便数据库的移植等。

而 Hibernate 正是这样一个工具，它在对象和关系型之间建立起一座桥梁，使得开发人员可以方便、快捷地进行持久化开发。

Hibernate 的第一个正式版本发布于 2001 年末；2003 年 6 月 Hibernate 2 发布，这一版本提供了对大多数数据库的支持；2003 年末 Hibernate 被 JBoss 吸纳，2005 年 3 月 Hibernate 3 正式发布，至此，Hibernate 获得了巨大的成功。Hibernate 使用起来非常简单，这也是 Hibernate 作者 Gavin King 的一贯思想。

11.2 Hibernate 的下载和配置

仍然从最基础的下载和配置讲起。这里除了需要下载 Hibernate 本身外，还需要下载 Hibernate 的两个支持工具：Middlegen-Hibernate 和 hibernate-extensions。

11.2.1 下载 Hibernate

任何人都可以从 http://prdownloads.sourceforge.net/hibernate/?sort_by=date&sort=desc 自

由下载 Hibernate。Hibernate 及其相关支持工具的下载画面如图 11.1 所示。

目前 Hibernate 提供的最新版本是 hibernate-3.2.0.cr2 版本，Middlegen-Hibernate 和 hibernate-extensions 的最新版本分别是 Middlegen-Hibernate-r5 和 hibernate-extensions-2.1.3，分别下载它们，下载后解压缩即可。

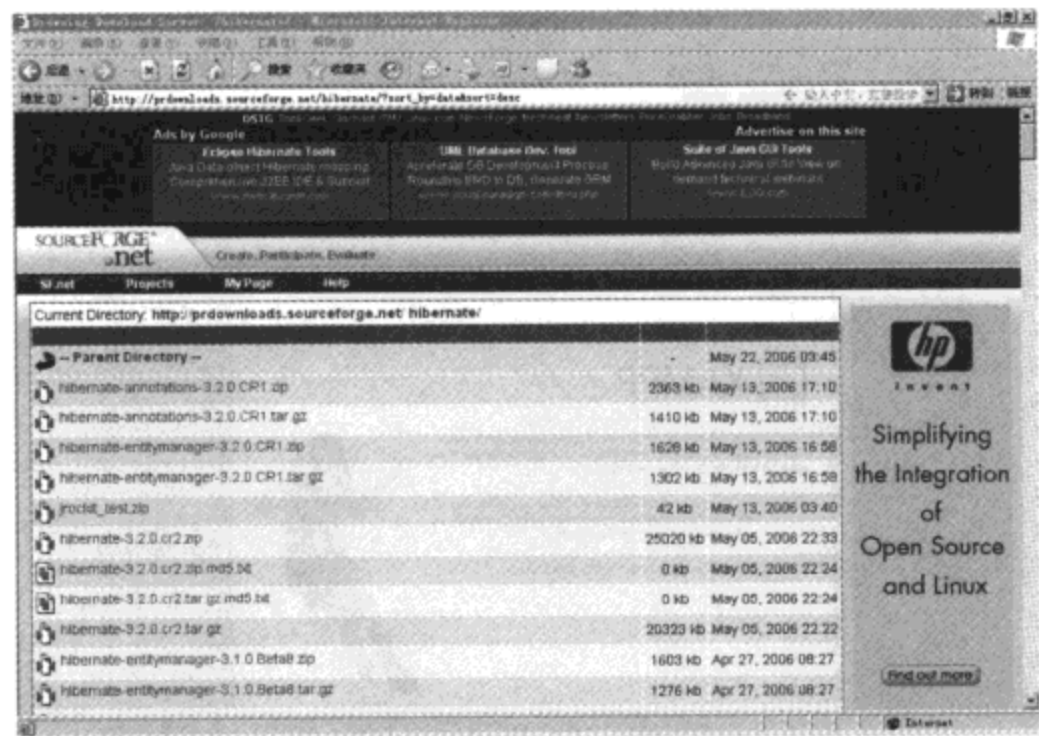


图 11.1 Hibernate 及其相关支持工具的下载画面

11.2.2 配置 Hibernate

使用 Hibernate，自然要用到数据库，本书使用 MySQL 数据库，读者可以到 <http://www.mysql.com> 网站下载 MySQL 数据库和 MySQL 相关的驱动程序。MySQL 的下载画面如图 11.2 所示。

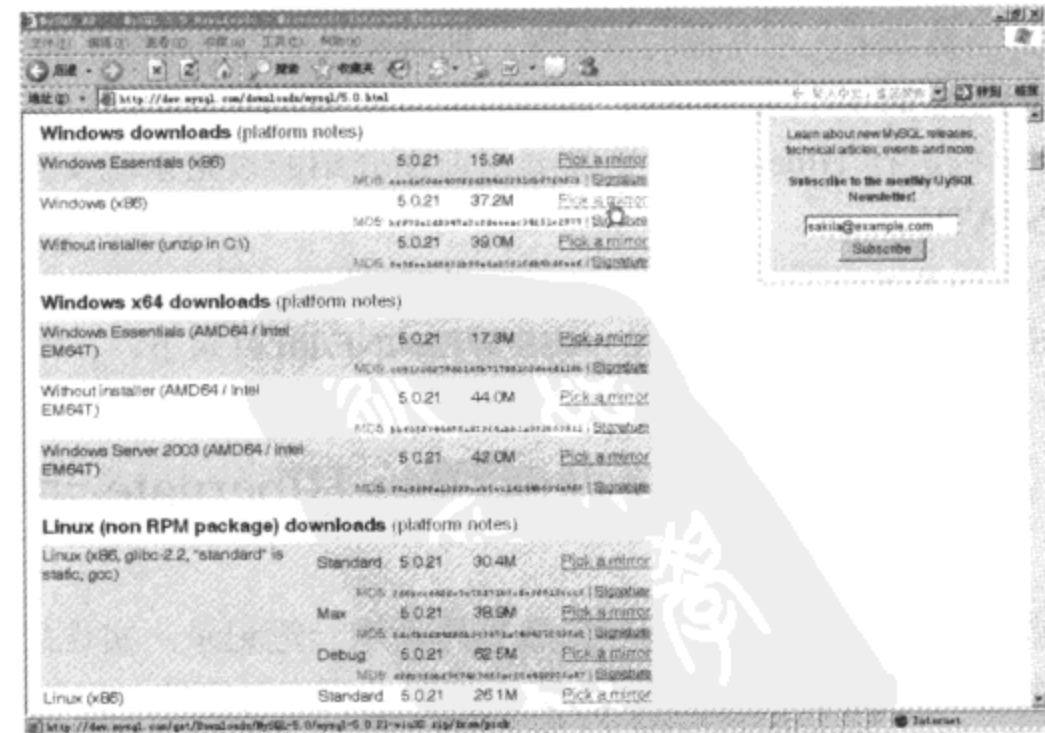


图 11.2 MySQL 的下载画面

也可以到 <http://www.apachefriends.org/en/xampp.html> 网站下载 XAMPP，它集成了 MySQL 5.0.20、PHP 5.1.1、PHP 4.4.2-pl、phpMyAdmin 2.8.0.3、XAMPP Control Panel Version 2.2 和 FileZilla FTP Server 0.9.14a 等工具，提供了对 MySQL 进行操作的可视化环境。XAMPP 的下载画面如图 11.3 所示。

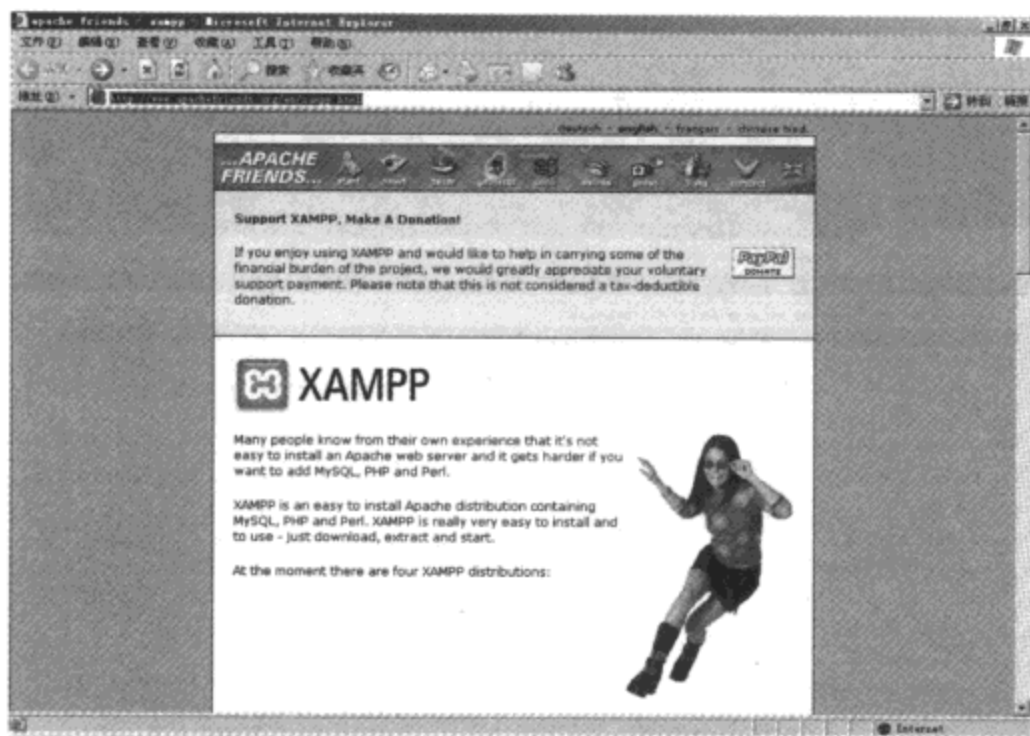


图 11.3 XAMPP 的下载画面

MySQL 数据库安装完毕后，要使用 Hibernate，还需要 Hibernate 相关的一些 jar：antlr.jar、asm.jar、asm-attrs.jar、cglib.jar、commons-collections.jar、commons-logging.jar、log4j.jar、dom4j.jar、ehcache.jar 和 jta.jar。这些 jar 在 hibernate-3.2.0 解压缩的 lib 目录里，当然最重要的是解压缩目录下的 hibernate3.jar。

要使用 MySQL，还需要 MySQL 的驱动程序，同样在 <http://www.mysql.com> 网站上下下载，下载完毕后，解压缩可以得到 mysql-connector-java-5.0.0-beta-bin.jar，也是运行 Hibernate 所需要的一个 jar。当然如果使用其他的数据库，则需要其他数据库相关的驱动程序。

获取上面所介绍的 jar 后，要对这些 jar 进行配置，有两种方法：

(1) 如果要单机运行 Hibernate，则需要用前面介绍的设定系统变量的方法，把上述 jar 设定在 CLASSPATH。

(2) 如果是进行 Web 开发，则要将这些 jar 放在项目工程的 WEB-INF/lib 目录下。这里仍然使用 mySS 工程，把这些 jar 放在 mySS/WEB-INF/lib 目录下。

11.3 一个实现数据新增的 Hibernate 示例

这个示例主要的功能是通过测试程序向数据库新增一笔记录。实现思路是：首先建立一个数据库和一个用来存放用户名和密码的表，然后编写表对应的 POJO 和 XML，接着编写 Hibernate 的配置文件，最后编写测试案例，对程序进行测试。这个示例展示了 Hibernate 的基本功能。具体步骤如下：

(4) 接下来根据向导，创建一个数据库表，名为 user，包含 3 个字段：id、username 和 password，设定 id 为主键，SQL 语句如下所示：

```
CREATE TABLE `user` (  
  `id` VARCHAR( 32 ) NOT NULL ,  
  `username` VARCHAR( 32 ) NOT NULL ,  
  `password` VARCHAR( 32 ) NOT NULL ,  
  PRIMARY KEY ( `id` )  
) ENGINE = MYISAM ;
```

(5) 最终的 user 表结构如图 11.6 所示。



图 11.6 user 表结构

11.3.2 编写表对应的 POJO

在包 com.gc.vo 上编写 User 类。User.java 的示例代码如下：

```
//***** User.java*****  
package com.gc.vo;  
  
public class User {  
    //定义 id  
    private String id = null;  
    //定义用户名  
    private String username = null;  
    //定义密码  
    private String password = null;  
    //设定 id  
    public void setId(String id) {  
        this.id = id;  
    }  
    //获取 id
```

```
public String getId() {  
    return this.id;  
}  
//设定用户名  
public void setUsername (String username) {  
    this.username = username;  
}  
//获取用户名  
public String getUsername () {  
    return this.username;  
}  
//设定密码  
public void setPassword (String password) {  
    this.password = password;  
}  
//获取密码  
public String getPassword () {  
    return this.password;  
}  
}
```

11.3.3 编写 POJO 对应的 XML

编写和 User.java 对应的映射文件 User.hbm.xml，放在包 com.gc.vo 中，和 User.java 同一个目录。User.hbm.xml 的示例代码如下：

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
<hibernate-mapping>  
    <!--定义类及对应的表-->  
    <class name="com.gc.vo.User" table="user">  
        <id name="id" type="string" unsaved-value="null">  
            <column name="id" sql-type="char(32)" />  
            <!--定义主键的生成方式-->  
            <generator class="uuid.hex"/>  
        </id>  
        <!--定义密码变量-->  
        <property  
            name="password"  
            type="java.lang.String"  
            update="true"  
            insert="true"  
            access="property"  
            column="password"  
            length="32"  
        />  
    </class>  
</hibernate-mapping>
```



```
<!--定义用户名-->
<property
    name="username"
    type="java.lang.String"
    update="true"
    insert="true"
    access="property"
    column="username"
    length="32"
/>
</class>
</hibernate-mapping>
```

代码说明：<generator class="uuid.hex"/>，是主键的生成方式。

11.3.4 编写 Hibernate 的配置文件

编写 Hibernate 的配置文件 hibernate.cfg.xml，放在 WEB-INF/src 目录下。hibernate.cfg.xml 的示例代码如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!--定义方言-->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!--定义驱动-->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <!--定义 URL-->
        <property name="connection.url">jdbc:mysql://localhost/mySSH</property>
        <!--定义用户名-->
        <property name="connection.username">root</property>
        <!--定义密码-->
        <property name="connection.password">root</property>
        <!--定义是否显示 SQL-->
        <property name="show_sql">true</property>
        <!-- Mapping files -->
        <mapping resource="com/gc/vo/User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

11.3.5 编写测试案例

在 mySS 工程上新建一个包 com.gc.test，然后建立一个测试类 TestHibernate。

TestHibernate.java 的示例代码如下：

```
//***** TestHibernate.java*****
package com.gc.test;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import com.gc.vo.User;
public class TestHibernate {
    private SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    public static void main(String[] args) throws HibernateException {
        //通过配置文件获取 SessionFactory
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
        User user = new User();
        user.setPassword("gd");
        user.setUsername("gd");
        //一个 session 类似一个连接
        Session session = sessionFactory.openSession();
        //进行事务处理
        Transaction transaction = session.beginTransaction();
        session.save(user);
        transaction.commit();
        session.close();
        sessionFactory.close();
    }
}
```

代码说明：向数据库新增一记录，用户名和密码都是 gd。

11.3.6 运行测试程序

运行测试程序 TestHibernate，即可看到 Hibernate 执行成功的显示信息，如图 11.7 所示。



图 11.7 Hibernate 执行成功的显示信息

从图中可以看到 Hibernate 生成的 SQL 语句如下：

```
insert into user (password, username, id) values (?, ?, ?)
```

查看 MySQL 数据库，可以看到已经有一笔记录存在了，id 为自动生成，用户名和密码都为 gd，如图 11.8 所示。

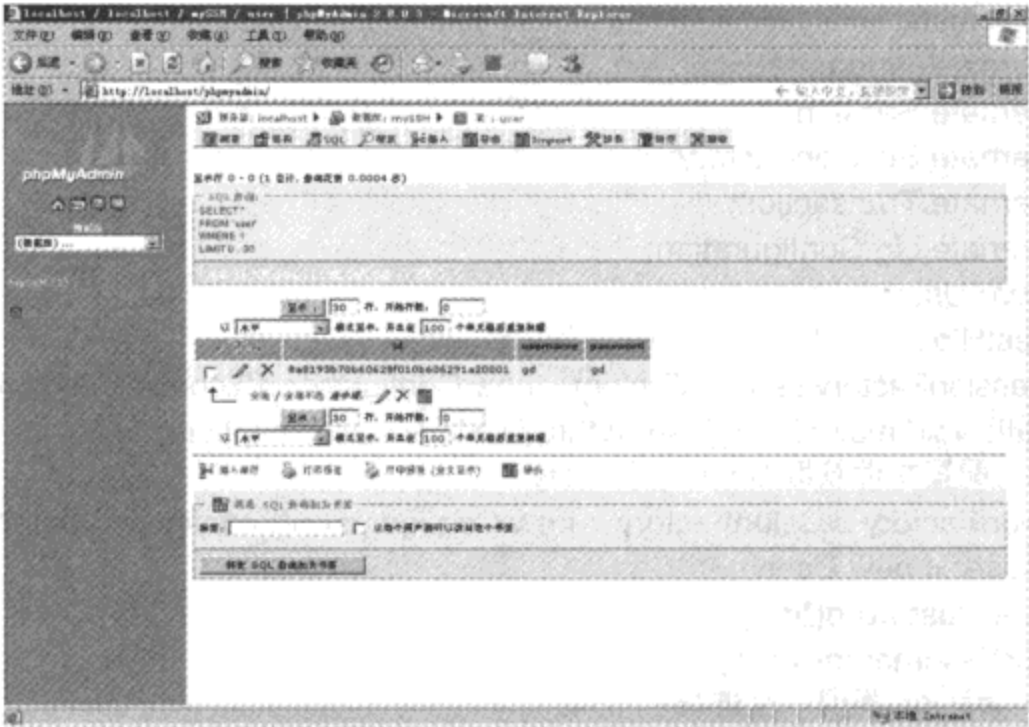


图 11.8 MySQL 中的记录

11.4 Hibernate 的配置

前面讲过，Hibernate 是在面向对象和关系型数据库之间建立一座桥梁，其实质是通过一个后缀名为 hbm.xml 的文件来建立面向对象和关系型数据库之间的联系。

从上可以看出，使用 Hibernate 是非常简单的一件事情，开发人员只需要编写 POJO 和相对应的映射文件、Hibernate 的配置文件即可。

- (1) 对于 POJO，前面已经讲过，这里不再多讲。
- (2) 和 POJO 相对应的映射文件。在 Hibernate 自带的文档的第 5 章中讲解得非常清楚，包括每个属性的含义都列了出来，读者可以参看。
- (3) Hibernate 的配置文件有两种表达方式：一个使用 XML，一个使用 properties。Hibernate 默认的配置文件名为 hibernate.cfg.xml，示例代码如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!--定义方言，使用 MySQL-->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!--定义 MySQL 的驱动-->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <!--定义 URL-->
    <property name="connection.url">jdbc:mysql://localhost/mySSH</property>
```

```

    <!--定义数据库用户名-->
    <property name="connection.username">root</property>
    <!--定义数据库密码-->
    <property name="connection.password">root</property>
    <!--定义是否显示生成的 SQL-->
    <property name="show_sql">true</property>
    <!--定义映射文件 -->
    <mapping resource="com/gc/vo/User.hbm.xml"/>
  </session-factory>
</hibernate-configuration>

```

代码说明：

- ❑ connection.driver_class, 表示 JDBC 驱动类。
- ❑ connection.url, 表示 URL。
- ❑ connection.username, 表示数据库用户。
- ❑ connection.password, 表示数据库用户密码。

如果需要使用 Hibernate 自带的连接池, 则需要定义 connection.pool_size, 表示连接池容量上限数目。如果需要使用其他的连接池, 比如 DBCP, 则可以用如下配置代替 connection.pool_size, 示例代码如下:

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!--定义方言, 使用 MySQL-->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!--定义 MySQL 的驱动-->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <!--定义 URL-->
    <property name="connection.url">jdbc:mysql://localhost/mySSH</property>
    <!--定义数据库用户名-->
    <property name="connection.username">root</property>
    <!--定义数据库密码-->
    <property name="connection.password">root</property>
    <!--定义连接池最小连接数-->
    <property name="dbcp.min_size">10</property>
    <!--定义连接池最大连接数-->
    <property name="dbcp.max_size">200</property>
    <!--定义连接池空闲时间-->
    <property name="dbcp.timeout">1000</property>
    <!--定义连接池最大 statements 数-->
    <property name="dbcp.max_statements">400</property>
    <!--定义是否显示生成的 SQL-->
    <property name="show_sql">true</property>
    <!-- Mapping files -->
    <mapping resource="com/gc/vo/User.hbm.xml"/>
  </session-factory>

```



```
</hibernate-configuration>
```

当然也可以使用 JNDI 进行连接，首先要在 Tomcat 的 server.xml 中添加以下代码：

```
<Context path="/mySSH" reloadable="true"
docBase="D:\eclipse\workspace\mySSH" workDir="D:\eclipse\workspace\mySSH\work" >
<Resource name="jdbc/opendb" auth="Container"
type="javax.sql.DataSource" factory="org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory"
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost/mySSH"
name="root" msg="root" maxActive="400"
maxIdle="1000" maxWait="400" removeAbandoned="true" removeAbandonedTimeout="10"
logAbandoned="true"
/></Context>
```

然后 Hibernate 的配置文档的示例代码如下：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!--定义方言，使用 MySQL-->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!--使用 jndi-->
    <property name="connection.datasource">java:comp/env/jdbc/opendb</property>
    <property name="show_sql">true</property>
    <!-- Mapping files -->
    <mapping resource="com/gc/vo/User.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

以上内容都是在 XML 里配置的，如果要在 properties 里配置，只需要在属性名称前加上 hibernate 即可，示例代码如下：

```
hibernate.dialect =org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class= com.mysql.jdbc.Driver
hibernate.connection.url =jdbc:mysql://localhost/mySSH
hibernate.connection.username= root
hibernate.connection.password= root
```

更详细的配置信息，可参考 Hibernate 提供的参考文档。

11.5 Hibernate 的映射

在前面的示例中，POJO 中都是一些简单的数据类型，它们对应一个简单的映射文件，然而在实际的应用中，POJO 就不会这么简单了，比如可能会用到集合，Bean 之间可能还

有继承、组合等，这些元素的映射文件都有其特殊的配置方式，下面将简单进行讲解。

11.5.1 集合映射

这里所说的集合包括 Set、Map 和 List。下面分别来说明这 3 种映射文件的配置方式，演示一个老师拥有多名学生。

(1) Set 方式，POJO 是这样的，包括有 Set 类型的对象。教师类的示例代码如下：

```
//***** Teacher.java*****
package com.gc.vo;
import java.util.HashSet;
import java.util.Set;

public class Teacher {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义 Set 类型的学生
    private Set students = new HashSet();
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取用户名
    public String getUsername () {
        return this.username;
    }
    //设定 set
    public void setStudents (Set students) {
        this.students = students;
    }
    //获取 set
    public Set getStudents () {
        return this.students;
    }
}
```

学生类的示例代码如下：

```
//***** Students.java*****
```

```
package com.gc.vo;
import java.util.HashSet;
import java.util.Set;

public class Students {
    //定义用户名
    private String username = null;
    //设定用户名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取用户名
    public String getUsername () {
        return this.username;
    }
}
```

那么这两个 Bean 对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义 Set 类型的表示方式-->
        <set name="students" table="students">
            <key column="id"/>
            <composite-element class="com.gc.vo.Students">
                <property name="username" column="username" not-null="true"/>
            </composite-element>
        </set>
    </class>
</hibernate-mapping>
```


上述代码对应的两个数据库表结构如下：

- ❑ 数据库表名为 teacher 的表，包括的字段为 id、username。
- ❑ 数据库表名为 student 的表，包括的字段为 id、username。

这里 student 表中 id 存储的值和 teacher 表中 id 存储的值相同。

上述代码把教师和学生分成了两个类，更符合面向对象的思想，如果只是简单地在教师类里存储学生的姓名，即此时只有教师类，没有学生类，则教师类的示例代码如下：

```
/****** Teacher.java *****  
package com.gc.vo;  
import java.util.HashSet;  
import java.util.Set;  
  
public class Teacher {  
    //定义 id  
    private String id = null;  
    //定义用户名  
    private String username = null;  
    //定义 Set 类型的学生  
    private Set students = new HashSet();  
    //设定 id  
    public void setId(String id) {  
        this.id = id;  
    }  
    //获取 id  
    public String getId() {  
        return this.id;  
    }  
    //设定用户名  
    public void setUsername (String username) {  
        this.username = username;  
    }  
    //获取用户名  
    public String getUsername () {  
        return this.username;  
    }  
    //设定 set  
    public void setStudents (Set students) {  
        this.students = students;  
    }  
    //获取 set  
    public Set getStudents () {  
        return this.students;  
    }  
}
```

那么此时对应的映射文件代码如下：

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC
```



```

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
  <!--定义类及对应的表-->
  <class name="com.gc.vo.Teacher " table="teacher">
    <id name="id" type="string" unsaved-value="null">
      <column name="id" sql-type="char(32)" />
      <generator class="uuid.hex"/>
    </id>
    <!--定义 username 变量-->
    <property
      name="username"
      type="java.lang.String"
      update="true"
      insert="true"
      access="property"
      column="username"
      length="32"
    />
    <!--定义 Set 类型的表示方式-->
    <set name="students" table="students">
      <key column="id"/>
      <element type="string" column="username" not-null="true"/>
    </set>
  </class>
</hibernate-mapping>

```

虽然前面的 Bean 和映射文件改变了，但数据库表结构并没有改变。

(2) Map 方式，POJO 是这样的，包括有 Map 类型的对象。教师类的示例代码如下：

```

//***** Teacher.java *****
package com.gc.vo;
import java.util.HashMap;
import java.util.Map;

public class Teacher {
  //定义 id
  private String id = null;
  //定义用户名
  private String username = null;
  //定义 Map 类型的学生
  private Map students = new HashMap ();
  //设定 id
  public void setId(String id) {
    this.id = id;
  }
  //获取 id
  public String getId() {
    return this.id;
  }
}

```

```
//设定用户名
public void setUsername (String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
//设定 Map
public void setStudents (Mapstudents) {
    this.students = students;
}
//获取 Map
public MapgetStudents () {
    return this.students;
}
}
```

学生类的示例代码如下：

```
/******* Students.java*****
package com.gc.vo;

public class Students {
    //姓名
    private String username = null;
    //身高
    private float length = 0f;
    //设定姓名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取姓名
    public String getUsername () {
        return this.username;
    }
    //设定身高
    public void setLength(float length) {
        this.length = length;
    }
    //获取身高
    public float getLength () {
        return this.length;
    }
}
```

那么这两个 Bean 对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
  <!--定义类及对应的表-->
  <class name="com.gc.vo.Teacher" table="teacher">
    <id name="id" type="string" unsaved-value="null">
      <column name="id" sql-type="char(32)" />
      <generator class="uuid.hex"/>
    </id>
    <!--定义 username 变量-->
    <property
      name="username"
      type="java.lang.String"
      update="true"
      insert="true"
      access="property"
      column="username"
      length="32"
    />
    <!--定义 Map 类型的表示方式-->
    <map name="students" table="students">
      <key column="id"/>
      <index column="length" type="string"/>
      <composite-element class="com.gc.vo.Students">
        <property name="username" column="username" not-null="true"/>
      </composite-element>
    </map>
  </class>
</hibernate-mapping>

```

上述代码对应的两个数据库表结构如下：

- ❑ 数据库表名为 teacher 的表，包括的字段为 id、username。
- ❑ 数据库表名为 student 的表，包括的字段为 id、username、length。

这里 student 表中 id 存储的值和 teacher 表中 id 存储的值相同。

上面同样是把教师和学生分成了两个类，更符合面向对象的思想，如果只是简单地在教师类里存储学生的姓名，即此时只有教师类，没有学生类，则教师类的示例代码如下：

```

//***** Teacher.java *****
package com.gc.vo;
import java.util.HashMap;
import java.util.Map;

public class Teacher {
  //定义 id
  private String id = null;
  //定义用户名
  private String username = null;
  //定义 Map 类型的学生
  private Map students = new HashMap ();
  //设定 id

```



```

public void setId(String id) {
    this.id = id;
}
//获取 id
public String getId() {
    return this.id;
}
//设定用户名
public void setUsername (String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
//设定 Map
public void setStudents (Mapstudents) {
    this.students = students;
}
//获取 Map
public MapgetStudents () {
    return this.students;
}
}

```

那么此时对应的映射文件代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher " table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义 Map 类型的表示方式-->
        <map name="students" table="students">

```



```

        <key column="id"/>
        <index column="length" type="string"/>
        <element type="string" column="username" not-null="true"/>
    </map>
</class>
</hibernate-mapping>

```

(3) List 方式，POJO 是这样的，包括有 List 类型的对象。教师类的示例代码如下：

```

//***** Teacher.java*****
package com.gc.vo;
import java.util.ArrayList;
import java.util.List;

public class Teacher {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义 List 类型的学生
    private List = new ArrayList ();
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取用户名
    public String getUsername () {
        return this.username;
    }
    //设定 List
    public void setStudents (List students) {
        this.students = students;
    }
    //获取 List
    public List getStudents () {
        return this.students;
    }
}

```

学生类的示例代码如下：

```

//***** Students.java*****
package com.gc.vo;

```

```
public class Students {  
    //姓名  
    private String username = null;  
    //身高  
    private float length = 0f;  
    //设定姓名  
    public void setUsername (String username) {  
        this.username = username;  
    }  
    //获取姓名  
    public String getUsername () {  
        return this.username;  
    }  
    //设定身高  
    public void setLength(float length) {  
        this.length = length;  
    }  
    //获取身高  
    public float getLength () {  
        return this.length;  
    }  
}
```

那么这两个 Bean 对应的映射文件代码如下：

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
<hibernate-mapping>  
    <!-- 定义类及对应的表-->  
    <class name="com.gc.vo.Teacher" table="teacher">  
        <id name="id" type="string" unsaved-value="null">  
            <column name="id" sql-type="char(32)" />  
            <generator class="uuid.hex"/>  
        </id>  
        <!-- 定义 username 变量-->  
        <property  
            name="username"  
            type="java.lang.String"  
            update="true"  
            insert="true"  
            access="property"  
            column="username"  
            length="32"  
        />  
        <!-- 定义 List 类型的表示方式-->  
        <list name="students" table="students">  
            <key column="id"/>
```

```
<index column="length" type="string"/>
<composite-element class="com.gc.vo.Students ">
    <property name="username" column="username" not-null="true"/>
</composite-element>
</list>
</class>
</hibernate-mapping>
```

上述代码对应的两个数据库表结构如下：

- ❑ 数据库表名为 teacher 的表，包括的字段为 id、username。
- ❑ 数据库表名为 student 的表，包括的字段为 id、username、length。

这里 student 表中 id 存储的值和 teacher 表中 id 存储的值相同。

上面同样是把教师和学生分成了两个类，更符合面向对象的思想，如果只是简单地在教师类里存储学生的姓名，即此时只有教师类，没有学生类。教师类的示例代码如下：

```
//***** Teacher.java*****
package com.gc.vo;
import java.util.ArrayList;
import java.util.List;

public class Teacher {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义 List 类型的学生
    private List = new ArrayList ();
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取用户名
    public String getUsername () {
        return this.username;
    }
    //设定 List
    public void setStudents (List students) {
        this.students = students;
    }
    //获取 List
    public List getStudents () {
```



```

        return this.students;
    }
}

```

那么此时对应的映射文件代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义 List 类型的表示方式-->
        <list name="students" table="students">
            <key column="id"/>
            <index column="length" type="string"/>
            <element type="string" column="username" not-null="true"/>
        </list>
    </class>
</hibernate-mapping>

```

从上可以看出，Set、Map 和 List 的配置方式类似，只是 Map 和 List 的配置方式比 Set 多了一个用来记录位置的列。

11.5.2 组件映射

在前面集合映射中，当分成教师类和学生类时，其实就已经使用了组件映射，只不过那是和集合映射一起使用的。这里演示把组件映射单独来进行配置，仍然以教师和学生为例来进行演示。

```

//***** Teacher.java*****
package com.gc.vo;

public class Teacher {

```



```
//定义 id
private String id = null;
//定义用户名
private String username = null;
//定义 Students 类型的学生
private Students = null;
//设定 id
public void setId(String id) {
    this.id = id;
}
//获取 id
public String getId() {
    return this.id;
}
//设定用户名
public void setUsername (String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
//设定 List
public void setStudents (Students students) {
    this.students = students;
}
//获取 Students
public Students getStudents () {
    return this.students;
}
}
```

学生类的示例代码如下：

```
//***** Students.java*****
package com.gc.vo;

public class Students {
    //姓名
    private String username = null;
    //身高
    private float length = 0f;
    //设定姓名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取姓名
    public String getUsername () {
        return this.username;
    }
}
```

```

//设定身高
public void setLength(float length) {
    this.length = length;
}
//获取身高
public float getLength () {
    return this.length;
}
}

```

那么这两个 Bean 对应的映射文件代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 定义类及对应的表-->
    <class name="com.gc.vo.Teacher " table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!-- 定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!-- 定义 component 类型的表示方式-->
        <component name="students" class="com.gc.vo.Students">
            <property name="username" column="studentsusername" not-null="true"/>
            <property name="length" column="length" not-null="true"/>
        </component>
    </class>
</hibernate-mapping>

```

上述代码对应的一个数据库表结构如下：数据库表名为 teacher 的表，包括的字段为 id、username、studentsusername 和 length。

11.5.3 关联映射

关联映射分为单向关联和双向关联，在单向关联和双向关联中又分为一对一、一对多、多对一和多对多关联，下面仍然用教师和学生的示例来分别进行演示它们的配置方式。

(1) 单向关联一对一，表示一个教师只对应一个学生。教师类的示例代码如下：

```
//***** Teacher.java*****  
package com.gc.vo;  
  
public class Teacher {  
    //定义 id  
    private String id = null;  
    //定义用户名  
    private String username = null;  
    //定义 Students 类型的学生  
    private Students = null;  
    //设定 id  
    public void setId(String id) {  
        this.id = id;  
    }  
    //获取 id  
    public String getId() {  
        return this.id;  
    }  
    //设定用户名  
    public void setUsername (String username) {  
        this.username = username;  
    }  
    //获取用户名  
    public String getUsername () {  
        return this.username;  
    }  
    //设定 List  
    public void setStudents (Students students) {  
        this.students = students;  
    }  
    //获取 Students  
    public Students getStudents () {  
        return this.students;  
    }  
}
```

学生类的示例代码如下：

```
//***** Students.java*****  
package com.gc.vo;  
  
public class Students {  
    //姓名  
    private String username = null;  
    //身高  
    private float length = 0f;  
    //设定姓名  
    public void setUsername (String username) {
```



```
        this.username = username;
    }
    //获取姓名
    public String getUsername () {
        return this.username;
    }
    //设定身高
    public void setLength(float length) {
        this.length = length;
    }
    //获取身高
    public float getLength () {
        return this.length;
    }
}
```

那么教师类对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--一对一关联-->
        <one-to-one name="students"
            class="com.gc.vo.Students"
            cascade="all"/>
    </class>
</hibernate-mapping>
```

代码说明：

- ☐ <one-to-one name="students", 表示是一对一关联。
- ☐ cascade="all"/>, 表示级联关系。如果设定为 all, 表示所有情况均进行级联；设定

为 none，表示所有情况均不进行级联；设定为 save-update，表示在执行 save-update 时进行级联；设定为 delete，表示在执行 delete 时进行级联。

上述代码对应的一个数据库表结构如下：数据库表名为 teacher 的表，包括的字段为 id 和 username；id 为主键。

学生类对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 定义类及对应的表 -->
    <class name="com.gc.vo.Students" table="student">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="foreign">
                <param name="property">teacher</param>
            </generator>
        </id>
        <!-- 定义 username 变量 -->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!-- 定义 length 变量 -->
        <property
            name="length"
            type="java.lang.Float"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
    </class>
</hibernate-mapping>
```

代码说明：<generator class="foreign">，表示 student 表主键的生成方式，和 teacher 表的主键保持一致。

上述代码对应的一个数据库表结构如下：数据库表名为 student 的表，包括的字段为 id、username 和 length；id 为主键。

上面示例使用的是主键关联的方式。另外还有一种单向一对一的关联是使用外键关联

的方式, Teacher 类和 Students 类不改变, 教师类对应的映射文件代码如下:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!-- 定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!-- 一对一关联-->
        <one-to-one name="students"
            class="com.gc.vo.Students"
            column="student_id"
            unique="true"/>
    </class>
</hibernate-mapping>
```

代码说明:

- ❑ <one-to-one name="students", 表示是一对一关联。
- ❑ unique="true", 表示在数据库表中该字段为唯一约束。

上述代码对应的一个数据库表结构如下: 数据库表名为 teacher 的表, 包括的字段为 id、username 和 student_id; id 为主键。

学生类对应的映射文件代码如下:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 定义类及对应的表-->
    <class name="com.gc.vo.Students" table="student">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
    </class>
</hibernate-mapping>
```

```
</id>
<!--定义 username 变量-->
<property
    name="username"
    type="java.lang.String"
    update="true"
    insert="true"
    access="property"
    column="username"
    length="32"
/>
<!--定义 length 变量-->
<property
    name="length"
    type="java.lang.Float"
    update="true"
    insert="true"
    access="property"
    column="username"
    length="32"
/>
</class>
</hibernate-mapping>
```

上述代码对应的一个数据库表结构如下：数据库表名为 student 的表，包括的字段为 id、username 和 length；id 为主键，与 teacher 表的 student_id 相关联。

(2) 双向关联一对一，表示一个教师只对应一个学生，一个学生只对应一个教师。教师类的示例代码如下：

```
/** ***** Teacher.java ***** */
package com.gc.vo;

public class Teacher {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义 Students 类型的学生
    private Students = null;
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
```



```
public void setUsername (String username) {  
    this.username = username;  
}  
//获取用户名  
public String getUsername () {  
    return this.username;  
}  
//设定 List  
public void setStudents (Students students) {  
    this.students = students;  
}  
//获取 Students  
public Students getStudents () {  
    return this.students;  
}  
}
```

学生类的示例代码如下：

```
//***** Students.java*****  
package com.gc.vo;  
  
public class Students {  
    //定义 id  
    private String id = null;  
    //定义用户名  
    private String username = null;  
    //定义身高  
    private float length = 0f;  
    //定义 Teacher  
    private Teacher teacher = null;  
    //设定 id  
    public void setId(String id) {  
        this.id = id;  
    }  
    //获取 id  
    public String getId() {  
        return this.id;  
    }  
    //设定用户名  
    public void setUsername (String username) {  
        this.username = username;  
    }  
    //获取用户名  
    public String getUsername () {  
        return this.username;  
    }  
    //设定身高  
    public void setLength(float length) {  
        this.length = length;  
    }  
}
```



```
}  
//获取身高  
public float getLength () {  
    return this.length;  
}  
//设定 Teacher  
public void setTeacher (Teacher teacher) {  
    this.teacher = teacher;  
}  
//获取 Teacher  
public Teacher getTeacher () {  
    return this.teacher;  
}  
}
```

那么教师类对应的映射文件代码如下：

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
<hibernate-mapping>  
    <!--定义类及对应的表-->  
    <class name="com.gc.vo.Teacher " table="teacher">  
        <id name="id" type="string" unsaved-value="null">  
            <column name="id" sql-type="char(32)" />  
            <generator class="uuid.hex"/>  
        </id>  
        <!--定义 username 变量-->  
        <property  
            name="username"  
            type="java.lang.String"  
            update="true"  
            insert="true"  
            access="property"  
            column="username"  
            length="32"  
        />  
        <!--一对一关联-->  
        <one-to-one name="students"  
            class="com.gc.vo.Students"  
            cascade="all"/>  
    </class>  
</hibernate-mapping>
```

代码说明：

- ☐ <one-to-one name="students"，表示是一对一关联。
- ☐ cascade="all"/>，表示级联关系。如果设定为 all，表示所有情况均进行级联；设定为 none，表示所有情况均不进行级联；设定为 save-update，表示在执行 save-update

时进行级联；设定为 delete，表示在执行 delete 时进行级联；

上述代码对应的一个数据库表结构如下：数据库表名为 teacher 的表，包括的字段为 id 和 username；id 为主键。

学生类对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Students" table="student">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="foreign">
                <param name="property">teacher</param>
            </generator>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义 length 变量-->
        <property
            name="length"
            type="java.lang.Float"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--一对一关联-->
        <one-to-one name="teacher"
            class="com.gc.vo.Teacher"
            constrained="true"/>
    </class>
</hibernate-mapping>
```

代码说明：

- ❑ <one-to-one name="teacher"，表示是一对一关联。
- ❑ constrained="true"/>，表示在 teacher 表上存在一个主键对 student 表的主键进行约束，

student 表引用的是 teacher 表的主键。

- ❑ `<generator class="foreign">`, 表示 student 表主键的生成方式, 和 teacher 表的主键保持一致。

上述代码对应的一个数据库表结构如下: 数据库表名为 student 的表, 包括的字段为 id、username 和 length; id 为主键。

上面示例使用的是主键关联的方式。另外还有一种一对一的关联是使用外键关联的方式, Teacher 类和 Students 类不改变。教师类对应的映射文件代码如下:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--一对一关联-->
        <one-to-one name="students"
            class="com.gc.vo.Students"
            column="student_id"
            unique="true"/>
    </class>
</hibernate-mapping>
```

代码说明:

- ❑ `<one-to-one name="students">`, 表示是一对一关联。
- ❑ `unique="true"`, 表示在数据库表中该字段为唯一约束。

上述代码对应的一个数据库表结构如下: 数据库表名为 teacher 的表, 包括的字段为 id、username 和 student_id; id 为主键。

学生类对应的映射文件代码如下:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
  <!--定义类及对应的表-->
  <class name="com.gc.vo.Students" table="student">
    <id name="id" type="string" unsaved-value="null">
      <column name="id" sql-type="char(32)" />
      <generator class="uuid.hex"/>
    </id>
    <!--定义 username 变量-->
    <property
      name="username"
      type="java.lang.String"
      update="true"
      insert="true"
      access="property"
      column="username"
      length="32"
    />
    <!--定义 length 变量-->
    <property
      name="length"
      type="java.lang.Float"
      update="true"
      insert="true"
      access="property"
      column="username"
      length="32"
    />
    <!--一对一关联-->
    <one-to-one name="teacher"
      class="com.gc.vo.Teacher"
      property-ref="student"/ >
  </class>
</hibernate-mapping>

```

代码说明:

- ❑ <one-to-one name="teacher", 表示是一对一关联。
- ❑ property-ref="student"/ >, 因为不是建立在主键基础上, 所以这里设定为 student, 表示 student 和 teacher 表相关联。

上述代码对应的一个数据库表结构如下: 数据库表名为 student 的表, 包括的字段为 id、username 和 length; id 为主键, 与 teacher 表的 student_id 相关联。

(3) 单向关联多对一, 表示多个学生对应一个教师, 教师类的示例代码如下:

```

//***** Teacher.java*****
package com.gc.vo;

public class Teacher {
  //定义 id

```



```
private String id = null;
//定义用户名
private String username = null;
//设定 id
public void setId(String id) {
    this.id = id;
}
//获取 id
public String getId() {
    return this.id;
}
//设定用户名
public void setUsername (String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
}
```

学生类的示例代码如下：

```
/****** Students.java*****
package com.gc.vo;

public class Students {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义身高
    private float length = 0f;
    //定义 Teacher
    private Teacher teacher = null;
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
    public void setUsername (String username) {
        this.username = username;
    }
    //获取用户名
    public String getUsername () {
        return this.username;
    }
}
```

```
//设定身高
public void setLength(float length) {
    this.length = length;
}
//获取身高
public float getLength () {
    return this.length;
}
//设定 Teacher
public void setTeacher (Teacher teacher) {
    this.teacher = teacher;
}
//获取 Teacher
public Teacher getTeacher () {
    return this.teacher;
}
}
```

那么教师类对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
    </class>
</hibernate-mapping>
```

上述代码对应的一个数据库表结构如下：数据库表名为 teacher 的表，包括的字段为 id 和 username；id 为主键。

学生类对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
  <!-- 定义类及对应的表 -->
  <class name="com.gc.vo.Students" table="student">
    <id name="id" type="string" unsaved-value="null">
      <column name="id" sql-type="char(32)" />
      <generator class="uuid.hex"/>
    </id>
    <!-- 定义 username 变量 -->
    <property
      name="username"
      type="java.lang.String"
      update="true"
      insert="true"
      access="property"
      column="username"
      length="32"
    />
    <!-- 定义 length 变量 -->
    <property
      name="length"
      type="java.lang.Float"
      update="true"
      insert="true"
      access="property"
      column="username"
      length="32"
    />
    <!-- 定义多对一 -->
    <many-to-one name="teacher"
      column="teacher_id"
      class="com.gc.vo.Teacher"/>
  </class>
</hibernate-mapping>

```

上述代码对应的一个数据库表结构如下：数据库表名为 student 的表，包括的字段为 id、username、length 和 teacher_id；id 为主键。teacher_id 和 teacher 表中的 id 相对应。

(4) 单向关联一对多，表示一个教师对应多个学生，教师类的示例代码如下：

```

//***** Teacher.java *****
package com.gc.vo;
import java.util.Set;
import java.util.HashSet;
public class Teacher {
  //定义 id
  private String id = null;
  //定义用户名
  private String username = null;
  //定义 Set 类型的 students

```



```
private Set students = new HashSet();
//设定 id
public void setId(String id) {
    this.id = id;
}
//获取 id
public String getId() {
    return this.id;
}
//设定用户名
public void setUsername(String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
//设定 students
public Set getStudents () {
    return students;
}
//获取 students
public void setStudents (Set students) {
    this.students = students;
}
}
```

学生类的示例代码如下：

```
/******* Students.java*****
package com.gc.vo;

public class Students {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义身高
    private float length = 0f;
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
    public void setUsername(String username) {
        this.username = username;
    }
}
```



```
}  
//获取用户名  
public String getUsername () {  
    return this.username;  
}  
//设定身高  
public void setLength(float length) {  
    this.length = length;  
}  
//获取身高  
public float getLength () {  
    return this.length;  
}  
}
```

那么教师类对应的映射文件代码如下：

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
<hibernate-mapping>  
    <!--定义类及对应的表-->  
    <class name="com.gc.vo.Teacher" table="teacher">  
        <id name="id" type="string" unsaved-value="null">  
            <column name="id" sql-type="char(32)" />  
            <generator class="uuid.hex"/>  
        </id>  
        <!--定义 username 变量-->  
        <property  
            name="username"  
            type="java.lang.String"  
            update="true"  
            insert="true"  
            access="property"  
            column="username"  
            length="32"  
        />  
        <!--定义一对多-->  
        <set name="students" table="student">  
            <key column="teacher_id"/>  
            <one-to-many class="com.gc.vo.Student"/>  
        </set>  
    </class>  
</hibernate-mapping>
```

上述代码对应的一个数据库表结构如下：数据库表名为 teacher 的表，包括的字段为 id 和 username；id 为主键。

学生类对应的映射文件代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 定义类及对应的表 -->
    <class name="com.gc.vo.Students" table="student">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!-- 定义 username 变量 -->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!-- 定义 length 变量 -->
        <property
            name="length"
            type="java.lang.Float"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
    </class>
</hibernate-mapping>

```

上述代码对应的一个数据库表结构如下：数据库表名为 student 的表，包括的字段为 id、username、length 和 teacher_id；id 为主键。teacher_id 和 teacher 表中的 id 相对应。

(5) 双向关联一对多或多对一，表示一个教师对应多个学生，教师类的示例代码如下：

```

//***** Teacher.java *****
package com.gc.vo;
import java.util.Set;
import java.util.HashSet;
public class Teacher {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义 Set 类型的 students
    private Set students = new HashSet();
    //设定 id

```

```
public void setId(String id) {
    this.id = id;
}
//获取 id
public String getId() {
    return this.id;
}
//设定用户名
public void setUsername(String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
//设定 students
public Set getStudents () {
    return students;
}
//获取 students
public void setStudents (Set students) {
    this.students = students;
}
}
```

学生类的示例代码如下：

```
//***** Students.java*****
package com.gc.vo;

public class Students {
    //定义 id
    private String id = null;
    //定义用户名
    private String username = null;
    //定义身高
    private float length = 0f;
    //定义 Teacher
    private Teacher teacher = null;
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定用户名
    public void setUsername(String username) {
        this.username = username;
    }
}
```



```
}  
//获取用户名  
public String getUsername () {  
    return this.username;  
}  
//设定身高  
public void setLength(float length) {  
    this.length = length;  
}  
//获取身高  
public float getLength () {  
    return this.length;  
}  
//设定 Teacher  
public void setTeacher (Teacher teacher) {  
    this.teacher = teacher;  
}  
//获取 Teacher  
public Teacher getTeacher () {  
    return this.teacher;  
}  
}
```

那么教师类对应的映射文件代码如下：

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
<hibernate-mapping>  
    <!-- 定义类及对应的表-->  
    <class name="com.gc.vo.Teacher" table="teacher">  
        <id name="id" type="string" unsaved-value="null">  
            <column name="id" sql-type="char(32)" />  
            <generator class="uuid.hex"/>  
        </id>  
        <!-- 定义 username 变量-->  
        <property  
            name="username"  
            type="java.lang.String"  
            update="true"  
            insert="true"  
            access="property"  
            column="username"  
            length="32"  
        />  
        <!-- 定义一对多关联-->  
        <set name="students" table="student">  
            <key column="teacher_id"/>  
            <one-to-many class="com.gc.vo.Student"/>  
        </set>  
    </class>  
</hibernate-mapping>
```



```
        </set>
    </class>
</hibernate-mapping>
```

上述代码对应的一个数据库表结构如下：数据库表名为 teacher 的表，包括的字段为 id 和 username；id 为主键。

学生类对应的映射文件代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Students" table="student">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义 length 变量-->
        <property
            name="length"
            type="java.lang.Float"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义多对一关联-->
        <many-to-one name="teacher"
            column="teacher_id"
            class="com.gc.vo.Teacher"/>
    </class>
</hibernate-mapping>
```

上述代码对应的一个数据库表结构如下：数据库表名为 student 的表，包括的字段为 id、username、length 和 teacher_id；id 为主键。teacher_id 和 teacher 表中的 id 相对应。

(6) 双向关联多对多，表示一个教师对应多个学生，一个学生对应多个教师。教师类

的示例代码如下：

```
//***** Teacher.java*****  
package com.gc.vo;  
import java.util.Set;  
import java.util.HashSet;  
public class Teacher {  
    //定义 id  
    private String id = null;  
    //定义用户名  
    private String username = null;  
    //定义 Set 类型的 students  
    private Set students = new HashSet();  
    //设定 id  
    public void setId(String id) {  
        this.id = id;  
    }  
    //获取 id  
    public String getId() {  
        return this.id;  
    }  
    //设定用户名  
    public void setUsername(String username) {  
        this.username = username;  
    }  
    //获取用户名  
    public String getUsername () {  
        return this.username;  
    }  
    //设定 students  
    public Set getStudents () {  
        return students;  
    }  
    //获取 students  
    public void setStudents (Set students) {  
        this.students = students;  
    }  
}
```

学生类的示例代码如下：

```
//***** Students.java*****  
package com.gc.vo;  
import java.util.Set;  
import java.util.HashSet;  
public class Students {  
    //定义 id  
    private String id = null;  
    //定义用户名  
    private String username = null;
```

```

//定义身高
private float length = 0f;
//定义 Teacher
private Set teacher = new HashSet();
    //设定 id
public void setId(String id) {
    this.id = id;
}
//获取 id
public String getId() {
    return this.id;
}
//设定用户名
public void setUsername(String username) {
    this.username = username;
}
//获取用户名
public String getUsername () {
    return this.username;
}
//设定身高
public void setLength(float length) {
    this.length = length;
}
//获取身高
public float getLength () {
    return this.length;
}
//设定 Teacher
public void setTeacher (Set teacher) {
    this.teacher = teacher;
}
//获取 Teacher
public Set getTeacher () {
    return this.teacher;
}
}

```

那么教师类对应的映射文件代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Teacher" table="teacher">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>

```



```

    </id>
    <!--定义 username 变量-->
    <property
        name="username"
        type="java.lang.String"
        update="true"
        insert="true"
        access="property"
        column="username"
        length="32"
    />
    <!--定义多对多关联-->
    <set name="students" table="teacherstudent" inverse="true"
        cascade="save-update" >
        <key column="teacher_id"/>
        <many-to-many class="com.gc.vo.Student"
            column="student_id"/>
    </set>
</class>
</hibernate-mapping>

```

学生类对应的映射文件代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Students" table="student">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 username 变量-->
        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="32"
        />
        <!--定义 length 变量-->
        <property
            name="length"
            type="java.lang.Float"
            update="true"
            insert="true"

```



```
        access="property"
        column="username"
        length="32"
    />
    <!--定义多对多关联-->
    <set name="teacher" table="teacherstudent" inverse="true"
        cascade="save-update" >
        <key column="student_id"/>
        <many-to-many class="com.gc.vo.Teacher"
            column="teacher_id"/>
    </set>
</class>
</hibernate-mapping>
```

上述代码对应的 3 个数据库表结构如下：

- ❑ 数据库表名为 teacher 的表，包括的字段为 id 和 username；id 为主键。
- ❑ 数据库表名为 student 的表，包括的字段为 id、username 和 length、；id 为主键。
- ❑ 数据库表名为 teacherstudent 的表，包括的字段为 teacher_id 和 student_id。teacher_id、student_id 都为键。

11.5.4 继承映射

这里以动物类来举例说明，有一个父类 Animal.java，一个子类 Dog.java，一个子类 Bird.java。Animal.java 的示例代码如下：

```
/** ***** Animal.java ***** */
package com.gc.vo;

public class Animal {
    //定义 id
    private String id = null;
    //定义头
    private String head = null;
    //定义身体
    private String body = null;
    //设定 id
    public void setId(String id) {
        this.id = id;
    }
    //获取 id
    public String getId() {
        return this.id;
    }
    //设定头
    public void setHead(String head) {
        this.head = head;
    }
}
```

```
//获取头
public String getHead () {
    return this.head;
}
//设定身体
public void setBody (String body) {
    this.body = body;
}
//获取身体
public String getBody () {
    return this.body;
}
}
```

Dog.java 的示例代码如下：

```
//***** Dog.java*****
package com.gc.vo;

public class Dog extends Animal {
    //定义 bite
    private String bite = null;
    //设定 bite
    public void setBite (String bite) {
        this.bite = bite;
    }
    //获取 bite
    public String getBite () {
        return this.bite;
    }
}
```

Bird.java 的示例代码如下：

```
//***** Bird.java*****
package com.gc.vo;

public class Bird extends Animal {
    //定义 fly
    private String fly = null;
    //设定 fly
    public void setFly (String fly) {
        this.fly = fly;
    }
    //获取 fly
    public String getFly () {
        return this.fly;
    }
}
```

这里给出两种映射方法：

(1) 所有内容存储在一个表中，映射文件的示例代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 定义类及对应的表 -->
    <class name="com.gc.vo.Animal" table="animal" discriminator-value="animal" >
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!-- 定义 head 变量 -->
        <property
            name="head"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="head"
            length="32"
        />
        <!-- 定义 body 变量 -->
        <property
            name="body"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="body"
            length="32"
        />
        <!-- 定义子类 -->
        <discriminator column="type" type="string"/>
        <subclass name="com.gc.vo.Dog" discriminator-value="dog">
            <property name="bite" type="string" column="bite"/>
        </subclass>
        <subclass name="com.gc.vo.Bird" discriminator-value="bird">
            <property name="fly" type="string" column="fly"/>
        </subclass>
    </class>
</hibernate-mapping>
```

上述代码对应的一个数据库表结构如下：数据库表名为 animal 的表，包括的字段为 id、head、body、bite、fly 和 type，其中使用 type 来区分存储的是 dog、bird 还是 animal。

(2) 读者应该可以看出，这种方式的缺点是显而易见的，因为 Dog 类只需要父类和它本身的 bite 属性，而不需要 fly 属性；而 Bird 类同样只需要父类和它本身的 fly 属性，而不需要 bite 属性，这样就会造成很多的数据冗余。下面给出另外一种解决方法，把不同的数

据放在不同的表里，映射文件的示例代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping>
    <!--定义类及对应的表-->
    <class name="com.gc.vo.Animal" table="animal">
        <id name="id" type="string" unsaved-value="null">
            <column name="id" sql-type="char(32)" />
            <generator class="uuid.hex"/>
        </id>
        <!--定义 head 变量-->
        <property
            name="head"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="head"
            length="32"
        />
        <!--定义 body 变量-->
        <property
            name="body"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="body"
            length="32"
        />
        <joined-subclass name="com.gc.vo.Dog" table="dog">
            <key column="id"/>
            <property name="bite" type="string" column="bite"/>
        </joined-subclass>
        <joined-subclass name="com.gc.vo.Bird" table="bird">
            <key column="id"/>
            <property name="fly" type="string" column="fly"/>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

上述代码对应的 3 个数据库表结构如下：

- ☐ 数据库表名为 animal 的表，包括的字段为 id、head 和 body。
- ☐ 数据库表名为 dog 的表，包括的字段为 id 和 bite。
- ☐ 数据库表名为 bird 的表，包括的字段为 id 和 fly。

其中这 3 个表的 id 存储的值相同。

11.6 Hibernate 的工具

前面讲述了 Hibernate 的一些基本原理，并手工编写了 POJO 和映射文件，其实这些代码都可以通过工具来自动生成。与 Hibernate 相关的配置文件主要有 3 个：数据库定义文件、POJO 和映射文件。Hibernate 提供的工具可以在这 3 种文件之间进行转换。

(1) 从数据库定义文件通过 Middlegen 到映射文件，从映射文件通过 hbm2java 到 POJO。

(2) 从 POJO 通过 XDoclet 到映射文件，从映射文件通过 SchemaExport 到数据库定义文件。

这里讲解第一种方法。

11.6.1 从数据库到映射

把前面下载的 Middlegen-Hibernate-r5.zip 解压缩到 D 盘的根目录下。使用 Middlegen 的具体配置步骤如下：

(1) 进入 Middlegen-Hibernate-r5/config/database 目录，可以看到目录下有不同数据库的 XML 配置文件，本书使用的是 MySQL，所以这里对 mysql.xml 进行修改。mysql.xml 示例代码如下：

```
<property name="database.script.file" value="${src.dir}/sql/${name}-mysql.sql"/>
  <property name="database.driver.file" value="${lib.dir}/mysql-connector-java-5.0.0-
beta-bin.jar"/>
  <property name="database.driver.classpath" value="${database.driver.file}"/>
  <property name="database.driver" value="org.gjt.mm.mysql.Driver"/>
  <property name="database.url" value="jdbc:mysql://localhost/mySSH"/>
  <property name="database.userid" value="root"/>
  <property name="database.password" value="root"/>
  <property name="database.schema" value=""/>
  <property name="database.catalog" value=""/>
  <property name="jboss.datasource.mapping" value="mySQL"/>
```

代码说明：

- ☐ 修改 database.url 为 jdbc:mysql://localhost/mySSH。
- ☐ 修改 database.userid 为 root。
- ☐ 修改 database.password 为 root。

(2) 把前面下载的 MySQL 的驱动 mysql-connector-java-5.0.0-beta-bin.jar 放在 Middlegen-Hibernate-r5/lib 目录下。

(3) 进入 Middlegen-Hibernate-r5 的根目录，打开 build.xml，修改其中的代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE project [
```

```

<!ENTITY database SYSTEM "file:./config/database/hsqldb.xml">
]>
<!-- 定义一个 project , basedir="."代表根目录是 build.xml 所在的目录-->
<project name="Middlegen Hibernate" default="all" basedir=".">
<!-- project name="Middlegen Hibernate" default="all" basedir="." -->
  <property file="{basedir}/build.properties"/>
  <property name="name" value="com.gc.vo"/>
  <!-- This was added because we were several people (in a course) deploying to same app
server>
  <property environment="env"/>
  <property name="unique.name" value="{name}.{env.COMPUTERNAME}"/-->
  <property name="gui" value="true"/>
  <property name="unique.name" value="{name}"/>
  <property name="appxml.src.file" value="{basedir}/src/application.xml"/>
  <property name="lib.dir" value="{basedir}/lib"/>
  <property name="src.dir" value="{basedir}/src"/>
  <property name="java.src.dir" value="{src.dir}/java"/>
  <property name="web.src.dir" value="{src.dir}/web"/>
  <property name="build.dir" value="{basedir}/build"/>
  <property name="build.java.dir" value="{build.dir}/java"/>
  <property name="build.gen-src.dir" value="{build.dir}/gen-src"/>
  <property name="build.classes.dir" value="{build.dir}/classes"/>
  &database;
  <!-- define the datasource.jndi.name in case the imported ejb file doesn't -->
  <property name="datasource.jndi.name" value="{name}/datasource"/>
  <path id="lib.class.path">
    <pathelement path="{database.driver.classpath}"/>
    <fileset dir="{lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <!-- The middlegen jars -->
    <!--fileset dir="{basedir}/.."-->
    <fileset dir="{basedir}/middlegen-lib">
      <include name="*.jar"/>
    </fileset>
  </path>
  <target name="init">
    <available
      property="xdoclet1.2+"
      classname="xdoclet.modules.hibernate.HibernateDocletTask" classpathref="lib.class.path"/>
  </target>
  <!-- ===== -->
  <!-- Fails if XDoclet 1.2.x is not on classpath -->
  <!-- ===== -->
  <target name="fail-if-no-xdoclet-1.2" unless="xdoclet1.2+">
    <fail>
      You must download several jar files before you can build Middlegen.
    </fail>
  </target>
  <!-- ===== -->
  <!-- Create tables -->

```

```

<!-- ===== -->
<target
  name="create-tables"
  depends="init,fail-if-no-xdoclet-1.2,check-driver-present,panic-if-driver-not-present"
  description="Create tables"
>
  <echo>Creating tables using URL ${database.url}</echo>
  <sql
    classpath="${database.driver.classpath}"
    driver="${database.driver}"
    url="${database.url}"
    userid="${database.userid}"
    password="${database.password}"
    src="${database.script.file}"
    print="true"
    output="result.txt"
  />
</target>
<target name="check-driver-present">
  <available file="${database.driver.file}" type="file" property="driver.present"/>
</target>
<target name="panic-if-driver-not-present" unless="driver.present">
  <fail>
    The JDBC driver you have specified by including one of the files in ${basedir}/config/database
    doesn't exist. You have to download this driver separately and put it in ${database.driver.file}
    Please make sure you're using a version that is equal or superior to the one we looked for.
    If you name the driver jar file differently, please update the database.driver.file property
    in the ${basedir}/config/database/xxx.xml file accordingly.
  </fail>
</target>
<!-- ===== -->
-->
<!-- Run Middlegen -->
<!-- ===== -->
-->
<target
  name="middlegen"
  description="Run Middlegen"
  unless="middlegen.skip"
  depends="init,fail-if-no-xdoclet-1.2,check-driver-present,panic-if-driver-not-present"
>
  <mkdir dir="${build.gen-src.dir}"/>
  <echo message="Class path = ${basedir}"/>
  <taskdef
    name="middlegen"
    classname="middlegen.MiddlegenTask"
    classpathref="lib.class.path"
  />
  <middlegen
    appname="${name}"

```



```

prefdir="${src.dir}"
gui="${gui}"
databaseurl="${database.url}"
initialContextFactory="${java.naming.factory.initial}"
providerURL="${java.naming.provider.url}"
datasourceJNDIName="${datasource.jndi.name}"
driver="${database.driver}"
username="${database.userid}"
password="${database.password}"
schema="${database.schema}"
catalog="${database.catalog}"
>
<!--
We can specify what tables we want Data generated for.
If none are specified, Data will be generated for all tables.
Comment out the <table> elements if you want to generate for all tables.
Also note that table names are CASE SENSITIVE for certain databases,
so on e.g. Oracle you should specify table names in upper case.
-->
<!--table generate="true" name="flights" pktable="flights_pk"/>
<table name="reservations"/-->
<!--
If you want m:n relations, they must be specified like this.
Note that tables declare in multiple locations must all have
the same value of the generate attribute.
-->
<!--many2many>
    <tablea generate="true" name="persons"/>
    <jointable name="reservations" generate="false"/>
    <tableb generate="true" name="flights"/>
</many2many-->
<!-- Plugins - Only Hibernate Plugin has been included with this special distribution -->

<!--
If you want to generate XDoclet markup for hbm2java to include in the POJOs then
set genXDocletTags to true. Also, composite keys are generated as external classes
which is
recommended. If you wish to keep them internal then set genIntergratedCompositeKeys
to true.
Since r4 the ability to customise the selection of JavaTypes is now provided. There is a
recommended type mapper provided as shown. It is optional - if not provided then
Middlegen
itself will select the Java mapping (as it did previously).
These settings are optional thus if they are not define here values default to false.
-->
<hibernate
    destination="${build.gen-src.dir}"
    package="${name}.hibernate"
    genXDocletTags="false"
    genIntergratedCompositeKeys="false"

```



```

        javaTypeMapper="middlesgen.plugins.hibernate.HibernateJavaTypeMapper"
    />

</middlesgen>
<mkdir dir="${build.classes.dir}"/>
</target>
<!-- ===== -->
<!-- Compile business logic (hibernate) -->
<!-- ===== -->
<target name="compile-hibernate" depends="middlesgen" description="Compile hibernate
Business Domain Model">
    <javac
        srcdir="${build.gen-src.dir}"
        destdir="${build.classes.dir}"
        classpathref="lib.class.path"
    >
        <include name="**/hibernate/**/*" />
    </javac>
</target>
<!-- ===== -->
<!-- Run hbm2java depends="middlesgen" -->
<!-- ===== -->
<target name="hbm2java" description="Generate .java from .hbm files.">
    <taskdef
        name="hbm2java"
        classname="net.sf.hibernate.tool.hbm2java.Hbm2JavaTask"
        classpathref="lib.class.path"
    />

    <hbm2java output="${build.gen-src.dir}">
        <fileset dir="${build.gen-src.dir}">
            <include name="**/*.hbm.xml" />
        </fileset>
    </hbm2java>
</target>
<!-- ===== -->
<!-- Build everything -->
<!-- ===== -->
<target name="all" description="Build everything" depends="compile-hibernate"/>
<!-- ===== -->
<!-- Clean everything -->
<!-- ===== -->
<target name="clean" description="Clean all generated stuff">
    <delete dir="${build.dir}"/>
</target>
<!-- ===== -->
<!-- Brings up the hsqldb admin tool -->
<!-- ===== -->
<target name="hsqldb-gui" description="Brings up the hsqldb admin tool">
    <property name="database.urlparams" value="?user=${database.userid}&password=$

```

```

{database.password}"/>
    <java
      classname="org.hsqldb.util.DatabaseManager"
      fork="yes"
      classpath="${lib.dir}/hsqldb-1.7.1.jar;${database.driver.classpath}"
      failonerror="true"
    >
      <arg value="-url"/>
      <arg value="${database.url}${database.urlparams}"/>
      <arg value="-driver"/>
      <arg value="${database.driver}"/>
    </java>
  </target>
  <!-- ===== -->
  <!-- Validate the generated xml mapping documents -->
  <!-- ===== -->
  <target name="validate">
    <xmlvalidate failonerror="no" lenient="no" warn="yes">
      <fileset dir="${build.gen-src.dir}/airline/hibernate" includes="*.xml" />
    </xmlvalidate>
  </target>
</project>

```

代码说明:

- ☐ 修改<!ENTITY database SYSTEM "file:./config/database/hsqldb.xml">中的 hsqldb.xml 为 mysql.xml。
- ☐ 修改<property name="name" value="airline"/>中的 value 为 com.gc.vo。
- ☐ 修改 package="\${name}.hibernate" 为 package="\${name} "。
- ☐ 修改 genXDocletTags="false" 为 genXDocletTags="true", 使其可以产生 XDoclet。

⚠注意: 在使用 Middlegen 之前, 应先配置 Ant, 具体配置方法, 参考第 12 章的介绍。

(4) 通过 cmd 控制台, 进入 D:\Middlegen-Hibernate-r5 目录下, 然后输入 ant, 运行 Ant, 如图 11.9 所示。

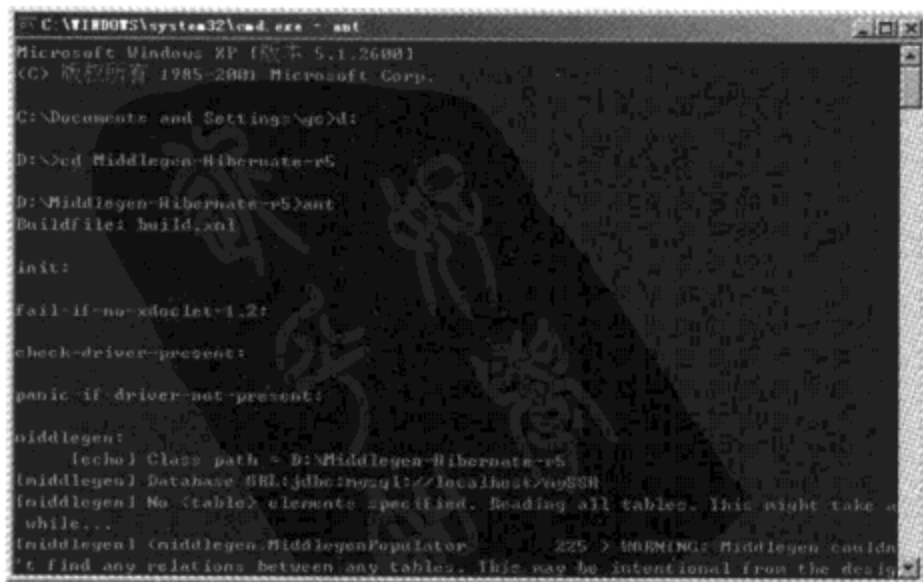


图 11.9 运行 Ant 的画面

(5) 运行 Ant 成功后，即可出现 Middlegen 的画面，它已经把前面建立的 user 表结构导入，如图 11.10 所示。

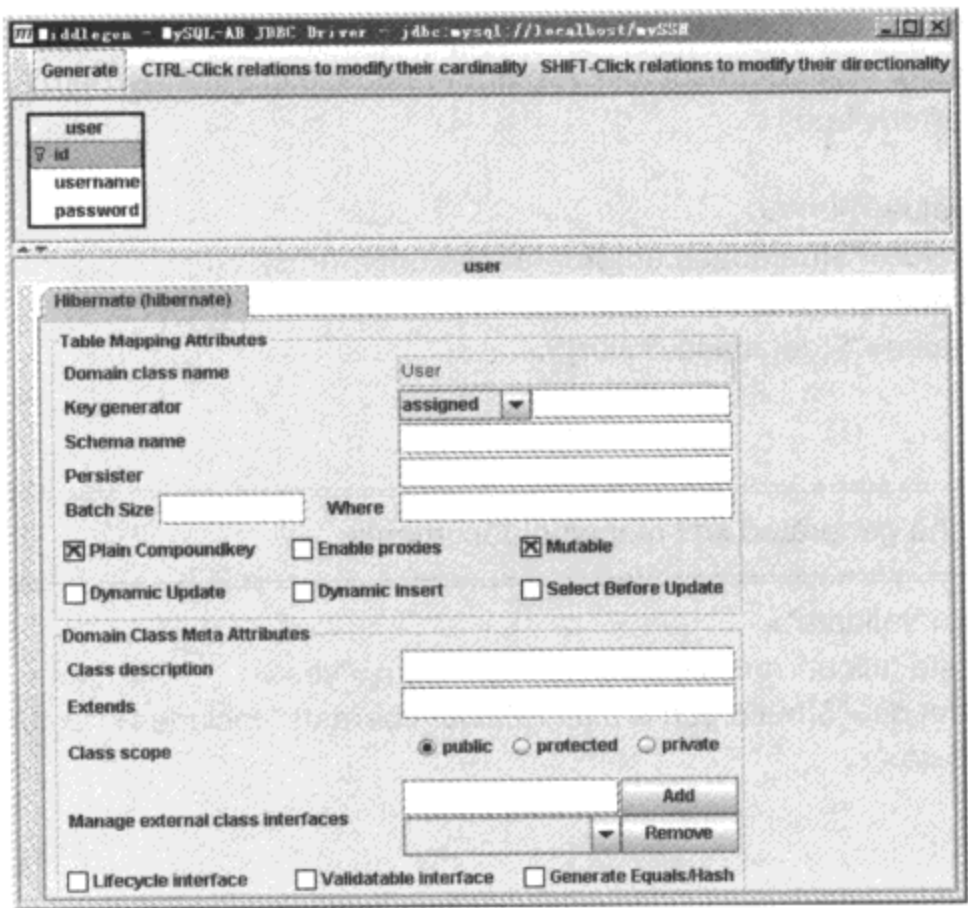


图 11.10 Middlegen 的画面

(6) 单击上面表结构中的 user，在 Middlegen 的画面的下半部分会出现与 user 相关的一些属性，这里只修改 Key generator（主键生成方式）为 uuid.hex，其他不变，如图 11.11 所示。

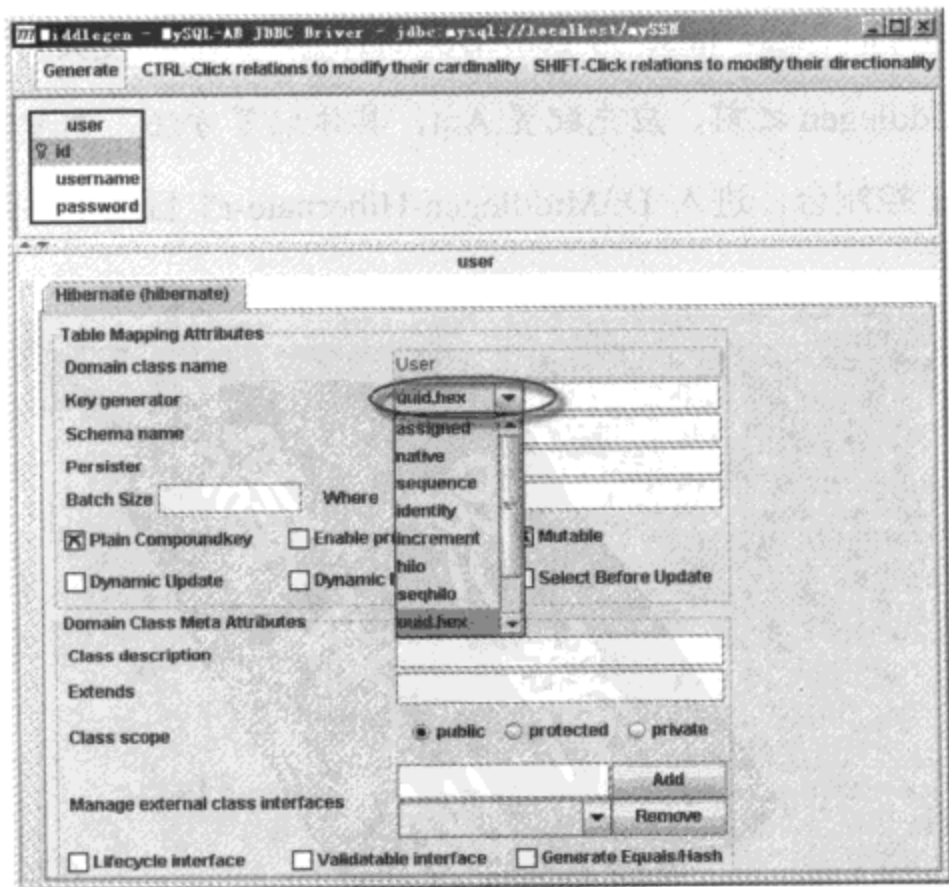


图 11.11 修改 Key generator（主键生成方式）为 uuid.hex

(7) 单击上面表结构中的 username 或 password，在 Middlegen 的画面的下半部分会出现与 username 或 password 相关的一些属性，这里不做任何修改，如图 11.12 所示。

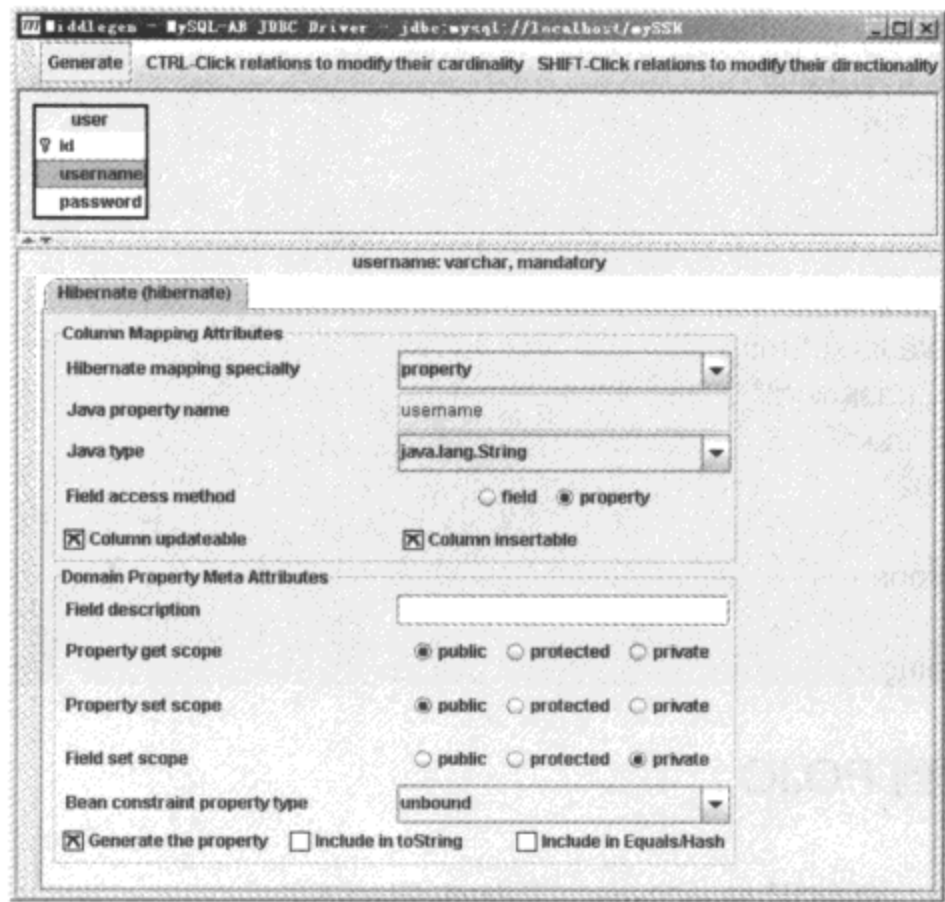


图 11.12 与 username 或 password 相关的一些属性

(8) 单击 Middlegen 画面左上方的 Generate 按钮，即可生成与 user 表相对应的映射文件 User.hbm.xml，该文件存放在 Middlegen-Hibernate-r5/build/gen-src/com/gc/vo 目录下。User.hbm.xml 的示例代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >
<hibernate-mapping>
<!--
    Created by the Middlegen Hibernate plugin 2.1
    http://boss.bekk.no/boss/middlegen/
    http://www.hibernate.org/
-->
<class
    name="com.gc.vo.User"
    table="user"
>
    <id
        name="id"
        type="java.lang.String"
        column="id"
    >
        <generator class="uuid.hex" />
    </id>
</class>
</hibernate-mapping>
</!DOCTYPE>
```



```

</id>
<property
    name="username"
    type="java.lang.String"
    column="username"
    not-null="true"
    length="32"
/>
<property
    name="password"
    type="java.lang.String"
    column="password"
    not-null="true"
    length="32"
/>
<!-- Associations -->
</class>
</hibernate-mapping>

```

11.6.2 从映射到 POJO

从映射到 POJO，要使用 hbm2java，具体步骤如下：

(1) 首先检查 D:\hibernate-extensions-2.1.3\tools\bin 目录下 setenv.bat 中 CP 的设定是否准确，用记事本打开 setenv.bat，然后查看 CP 中各种 jar 的设定是否正确，并增加 set HIBERNATE_HOME=D:\hibernate-3.2。setenv.bat 的示例代码如下：

```

@echo off
rem -----
rem Setup environment for hibernate tools
rem -----
set
JDBC_DRIVER=C:\Progra~1\SQLLIB\java\db2java.zip;C:\mm.mysql-2.0.14\mm.mysql-2.0.14-bin.j
ar
set HIBERNATE_HOME=D:\hibernate-3.2
set HIBERNATETOOLS_HOME=%~dp0..
echo HIBERNATETOOLS_HOME set to %HIBERNATETOOLS_HOME%
if "%HIBERNATE_HOME%" == "" goto noHIBERNATEHome
set CORELIB=%HIBERNATE_HOME%\lib
set LIB=%HIBERNATETOOLS_HOME%\lib
set
CP=%CLASSPATH%;%JDBC_DRIVER%;%HIBERNATE_HOME%\hibernate2.jar;%CORELIB%\c
ommons-logging-1.0.4.jar;%CORELIB%\commons-lang-1.0.1.jar;%CORELIB%\cglib-2.1.3.jar;%C
ORELIB%\dom4j-1.6.1.jar;%CORELIB%\odmg-3.0.jar;%CORELIB%\xml-apis.jar;%CORELIB%\xe
rces-2.6.2.jar;%CORELIB%\xalan-2.4.0.jar;%LIB%\jdom.jar;%CORELIB%\commons-collections-2.
1.1.jar;%LIB%\..\hibernate-tools.jar
if not "%HIBERNATE_HOME%" == "" goto end
:noHIBERNATEHome
echo HIBERNATE_HOME is not set. Please set HIBERNATE_HOME.

```

```
goto end
:end
```

⚠注意：因为 hibernate-extensions-2.1.3 目前还不支持 Hibernate 3，所以这里要麻烦一点。按照下载 Hibernate 3 同样的方法把 Hibernate 2 下载下来，并解压缩，把解压缩后 hibernate-2.1 目录下的 hibernate2.jar 放在 hibernate-3.2 目录下，把 hibernate-2.1/lib 目录下的 commons-lang-1.0.1.jar、odmg-3.0.jar 和 xalan-2.4.0.jar 这 3 个 jar 放在 hibernate-3.2/lib 目录下。然后修改 CP 中各个 jar 的版本号，使其与 hibernate-3.2/lib 目录下相应 jar 的版本号一致。

(2) 通过 cmd 控制台，进入 D:\hibernate-extensions-2.1.3\tools\bin 目录下，然后输入 hbm2java D:\Middlegen-Hibernate-r5\build\gen-src\com\gc\vo\User.hbm.xml --output=D:\Middlegen-Hibernate-r5\build\gen-src\，即可在 D:\Middlegen-Hibernate-r5\build\gen-src\com\gc\vo 目录下生成 User.java，如图 11.13 所示。

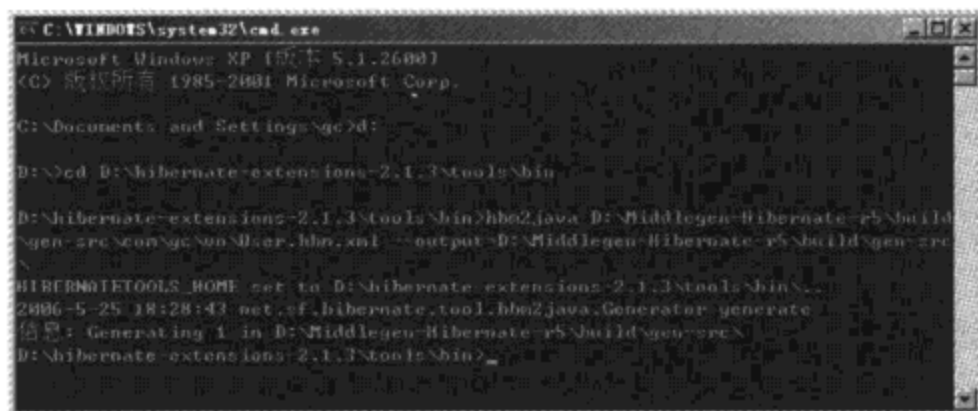


图 11.13 运行 hbm2java

⚠注意：output=D:\Middlegen-Hibernate-r5 中=两边不能有空格。

(3) 生成的 User.java 的示例代码如下：

```
//***** User.java*****
package com.gc.vo;
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;
/** @author Hibernate CodeGenerator */
public class User implements Serializable {
    /** identifier field */
    private String id;
    /** persistent field */
    private String username;
    /** persistent field */
    private String password;
    /** full constructor */
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    /** default constructor */
```

```
public User() {  
}  
public String getId() {  
    return this.id;  
}  
public void setId(String id) {  
    this.id = id;  
}  
public String getUsername() {  
    return this.username;  
}  
public void setUsername(String username) {  
    this.username = username;  
}  
public String getPassword() {  
    return this.password;  
}  
public void setPassword(String password) {  
    this.password = password;  
}  
public String toString() {  
    return new ToStringBuilder(this)  
        .append("id", getId())  
        .toString();  
}  
}
```

上面通过 Middlegen 和 hbm2java 这两个工具完成了从数据库定义文件到映射文件，从映射文件到 POJO 的转换。

11.7 Hibernate 的几个主要类简介

Hibernate 的正常工作是建立在 Configuration、SessionFactory 和 Session 等这些类的基础上。

11.7.1 Configuration（管理 Hibernate）

Configuration 类负责 Hibernate 的配置工作，它管理着 Hibernate 运行时需要的一些信息，如数据库连接、用户名、密码、方言（dialect）等。在前面的示例中，笔者使用

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

来获取相关的配置信息，当程序调用上述代码时，系统会自动寻找 hibernate.cfg.xml 文件。如果映射文件在 CLASSPATH 里，则开发人员也可以使用

```
Configuration cfg = new Configuration().addResource("User.hbm.xml");
```

来获取配置文件。这时如果 User.hbm.xml 不在 CLASSPATH 中，则可以使用下面这种方式来获取，示例代码如下：

```
Configuration cfg = new Configuration().addResource("com.gc.vo.User.class");
```

11.7.2 SessionFactory（创建 Session）

SessionFactory 用来根据配置好的 Configuration 创建连接，即创建 Session，代码如下：

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

11.7.3 Session（提供 Connection）

Session 类似于 Connection，是 Hibernate 的核心，它的创建方法如下：

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();  
Session session = sessionFactory.openSession();
```

如果单独使用 Hibernate，则下面的这段代码提供了一个很好的工具类，它把事务处理都包含了进去，示例代码如下：

```
/** ***** HibernateSessionUtil.java ***** */  
import java.io.Serializable;  
import org.hibernate.HibernateException;  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.Transaction;  
public class HibernateSessionUtil implements Serializable  
{  
    //保证连接是线程安全的  
    public static final ThreadLocal threadLocals = new ThreadLocal();  
    public static final ThreadLocal threadLocal = new ThreadLocal();  
    //获取 Session  
    public static Session currentSession(){  
        Session session = (Session) threadLocals.get();  
        try{  
            //没有则新建  
            if (session == null){  
                session = openSession();  
                threadLocals.set(session);  
            }  
        }catch (HibernateException e){  
            throw new InfrastructureException(e);  
        }  
        return session;  
    }  
    //关闭 Session  
    public static void closeSession(){  
        Session session = (Session) threadLocals.get();
```



```
threadLocals.set(null);
try{
    //当 session 不为空且开着时才关闭
    if (session != null && session.isOpen()){
        session.close();
    }
}catch (HibernateException e){
    throw new InfrastructureException(e);
}
}
//开始事务处理
public static void beginTransaction(){
    Transaction tx = (Transaction) threadLocal.get();
    try{
        if (tx == null){
            //事务处理
            tx = currentSession().beginTransaction();
            threadLocal.set(tx);
        }
    }catch (HibernateException e){
        throw new InfrastructureException(e);
    }
}
//结束事务处理
public static void commitTransaction(){
    Transaction tx = (Transaction) threadLocal.get();
    try{
        if (tx != null && !tx.wasCommitted() && !tx.wasRolledBack())
            //进行 commit
            tx.commit();
        threadLocal.set(null);
    }catch (HibernateException e){
        throw new InfrastructureException(e);
    }
}
//如果有异常, 则 rollback
public static void rollbackTransaction(){
    Transaction tx = (Transaction) threadLocal.get();
    try{
        threadLocal.set(null);
        if (tx != null && !tx.wasCommitted() && !tx.wasRolledBack()){
            //调用 rollback
            tx.rollback();
        }
    }catch (HibernateException e){
        throw new InfrastructureException(e);
    }
}
//打开 session
private static Session openSession() throws HibernateException{
```

```
        return getSessionFactory().openSession();
    }
    //获取 session 工厂
    private static SessionFactory getSessionFactory() throws HibernateException{
        return SingletonSessionFactory.getInstance();
    }
}
```

上述代码和使用 JDBC 时类似。更详细的内容可以查看 Heibernate 的参考文档和源代码。

11.8 通过 XML 来整合 Spring 和 Hibernate

其实质就是通过 IoC 把 Hibernate 纳入 Spring 来进行管理。

配置文件 Hibernate-Context.xml 的编写方式

Hibernate-Context.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring-beans.dtd">

<beans>
    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!-- 设定驱动 -->
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <!-- 设定 URL -->
        <property name="url">
            <value>jdbc:mysql://localhost/mySSH</value>
        </property>
        <!-- 设定用户名 -->
        <property name="username">
            <value>root</value>
        </property>
        <!-- 设定密码 -->
        <property name="password">
            <value>root</value>
        </property>
    </bean>
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
    </bean>
</beans>
```

```
</property>
<property name="mappingResources">
  <list>
    <value>com\gc\vo\User.hbm.xml</value>
  </list>
</property>
<property name="hibernateProperties">
  <props>
    <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
    <prop key="hibernate.show_sql">true</prop>
  </props>
</property>
</bean>
<bean id="transactionManager"
  class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>
<bean id="userDAOProxy"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager" />
  </property>
  <property name="target">
    <ref local="userDAO" />
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="insert*">PROPAGATION_REQUIRED</prop>
      <prop key="is*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
<bean id="userDAO" class="com.gc.dao.impl.UserDAOImpl">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>
</beans>
```

11.9 整合 Struts、Spring 和 Hibernate 实现用户注册的示例

这个示例主要实现的功能是：用户登录到注册页面，填写要注册的用户名和密码，然后单击“提交”按钮。如果注册成功，则返回到注册成功的页面，并显示注册的相关信息；如果注册不成功，则返回原来注册的页面，并提示注册不成功。

这个示例使用 Spring 的 `DelegatingActionProxy` 类来整合 Spring 和 Struts，使用 11.8 节

讲的 Hibernate-Context.xml 来整合 Spring 和 Hibernate。把这个整合项目命名为 mySSH，下面讲解具体实现步骤。

11.9.1 Spring、Struts 和 Hibernate 整合环境的配置

作为一个完整实现 Spring、Struts 和 Hibernate 整合的示例，在此再次从开始配置环境、建立 Tomcat 工程讲起。

(1) 运行 Eclipse，选择 File→New→Project 命令，Eclipse 将弹出 New Project 对话框，如图 11.14 所示。

(2) 用鼠标选择列表框中 Java 下的 Tomcat Project，然后单击 Next 按钮，弹出 New Tomcat Project 对话框，如图 11.15 所示。

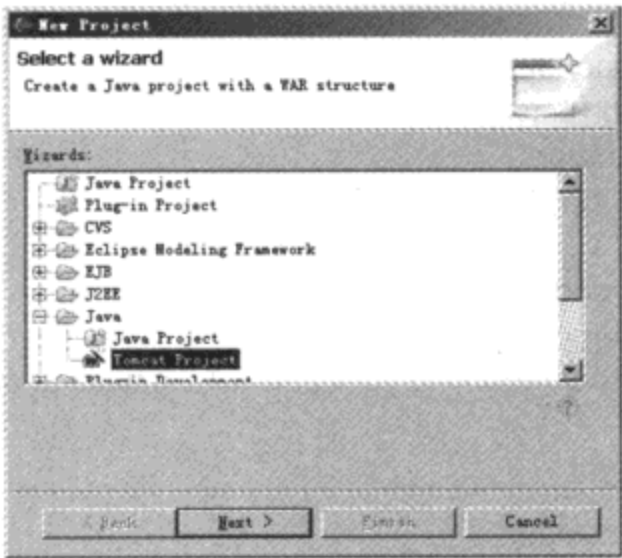


图 11.14 New Project 对话框

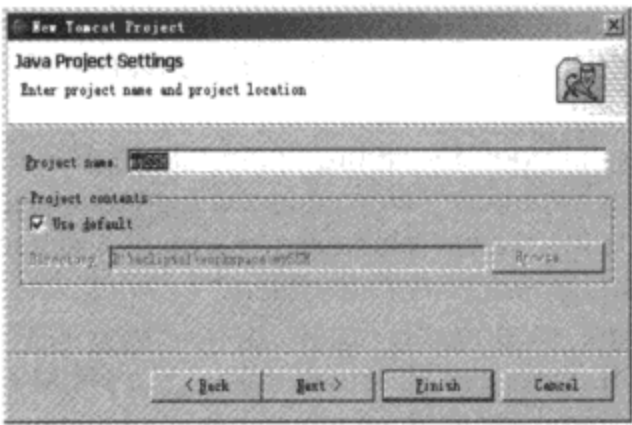


图 11.15 New Tomcat Project 对话框

(3) 在 New Tomcat Project 对话框的 Project name 文本框中输入“mySSH”，然后单击 Finish 按钮，项目即建立成功，mySSH 的目录结构如图 11.16 所示。

(4) 把 struts.jar、commons-beanutils.jar、commons-digester.jar、log4j-1.2.9.jar、commons-logging.jar、spring.jar、antlr.jar、asm.jar、spring-hibernate3.jar、asm-attrs.jar、cglib.jar、commons-collections.jar、dom4j.jar、ehcache.jar、jta.jar、mysql-connector-java-5.0.0-beta-bin.jar 和 hibernate3.jar 这 17 个 jar 放在 /WEB-INF/lib/ 下并复制到 mySSH /WEB-INF/lib 目录下，即 CLASSPATH 中。

(5) 用 Windows 自带的文本编辑器，建立一个文件 log4j.properties，把 log4j.properties 放到 mySSH /WEB-INF/src 目录下。

(6) 编辑 log4j.properties 文件，内容如下：

```
log4j.rootLogger=DEBUG,stdout,R
log4j.logger.org=ERROR, A1
```

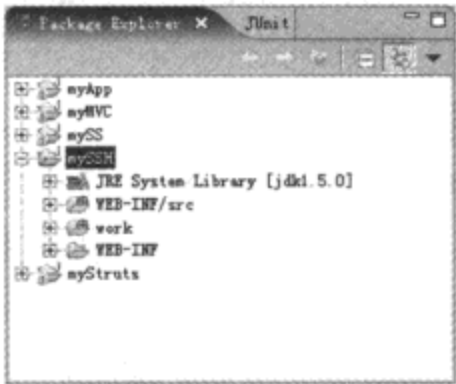


图 11.16 mySSH 的目录结构


```
log4j.logger.com.gc =DEBUG,A2
log4j.appender.A1=org.apache.log4j.RollingFileAppender
log4j.appender.A1.File=org.log
log4j.appender.A1.MaxFileSize=500KB
log4j.appender.A1.MaxBackupIndex=50
log4j.appender.A1.Append=true
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
log4j.appender.A2=org.apache.log4j.RollingFileAppender
log4j.appender.A2.File=gc.log
log4j.appender.A2.MaxFileSize=500KB
log4j.appender.A2.MaxBackupIndex=50
log4j.appender.A2.Append=true
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
#-----stdout-----
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#-----R-----
#log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
#this log file will be stored in web server's /bin directory,modify to your path which want to store.
log4j.appender.R.File=gf.log
#log4j.appender.R.datePattern='.'yyyy-MM-dd-HH-mm
log4j.appender.R.datePattern='.'yyyy-MM-dd
log4j.appender.R.append=true
## Keep one backup file
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n
```

(7) 在 mySSH 上单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，将弹出 Properties for mySSH 对话框，如图 11.17 所示。

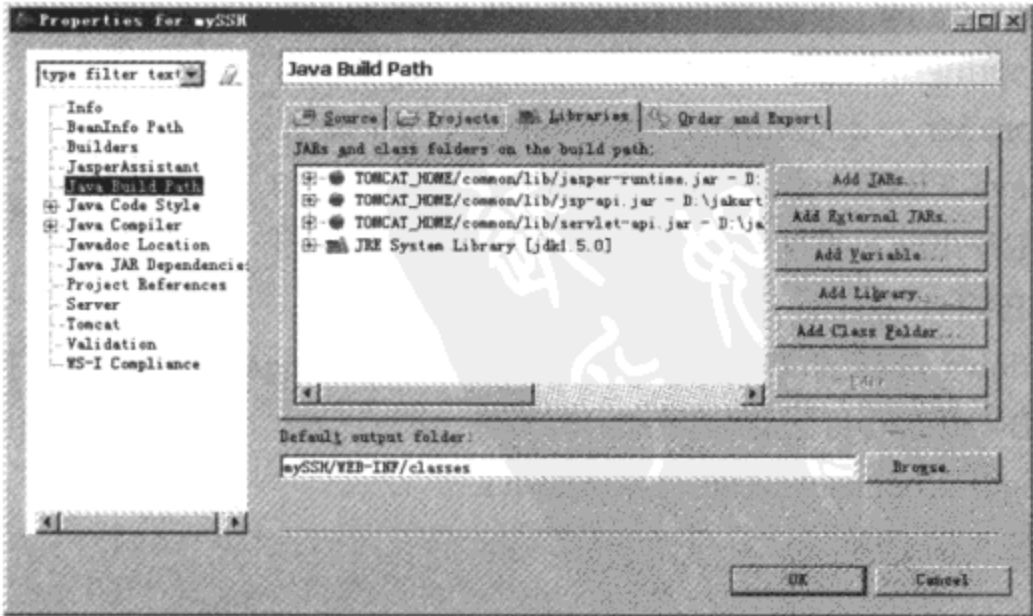


图 11.17 Properties for mySSH 对话框

(8) 在 Properties for mySSH 对话框中, 选择对话框左边列表框中的 Java Build Path。

(9) 在 Libraries 选项卡中, 单击 Add JARs...按钮, 弹出 JAR Selection 对话框, 如图 11.18 所示。

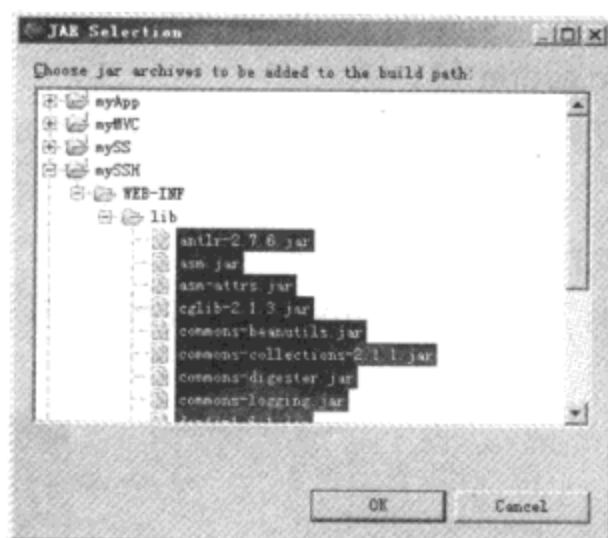


图 11.18 JAR Selection 对话框

(10) 在 JAR Selection 对话框中, 打开列表框中的 mySSH 一直到 lib 目录下出现 17 个 jar : struts.jar 、 commons-beanutils.jar 、 commons-digester.jar 、 log4j-1.2.9.jar 、 commons-logging.jar、spring.jar、antlr.jar、spring-hibernate3.jar、asm.jar、asm-attrs.jar、cglib.jar、commons-collections.jar、dom4j.jar、ehcache.jar、jta.jar、mysql-connector-java-5.0.0-beta-bin.jar、hibernate3.jar。

(11) 按住 Ctrl 键, 选中这 17 个 jar, 然后单击 OK 按钮, 返回到 Properties for mySSH 对话框中, 如图 11.19 所示。

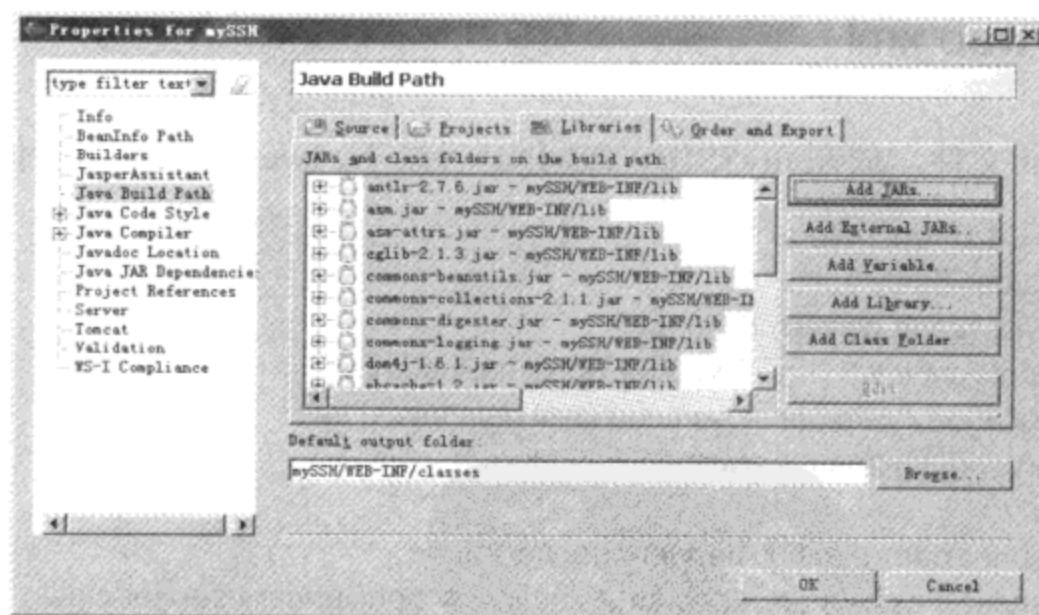


图 11.19 Properties for mySSH 对话框

(12) 在 Properties for mySSH 对话框中单击 OK 按钮, 即可完成对 Struts、Spring 和 Hibernate 的配置。

(13) 再次在 mySSH 上单击鼠标右键, 在弹出的快捷菜单中选择 New→Package 命令, 弹出 New Java Package 对话框, 如图 11.20 所示。

注意：在 spring.jar 中，没有包含对 Hibernate 的支持，如果需要使用 Hibernate，则需要把 spring-framework-2.0-m1/dist/extmodules/spring-hibernate3.jar 也包含进 CLASSPATH 中。

- (14) 在 New Java Package 对话框的 Name 文本框中输入 “com.gc.action”，然后单击 Finish 按钮，即可建立 com.gc.action 包。
- (15) 用同样的方法建立 com.gc.service 包、com.gc.service.impl 包、com.gc.dao 包和 com.gc.vo 包。
- (16) 在 WEB-INF 目录下建立 JSP 文件夹。
- (17) 最终配置好 Struts、Spring 和 Hibernate 的 mySSH 项目的目录结构，如图 11.21 所示。

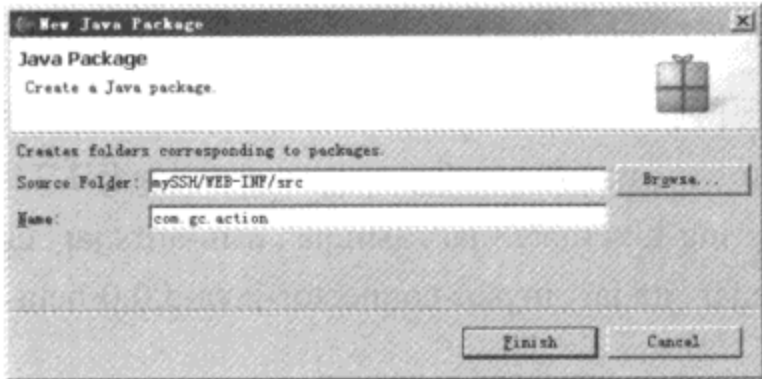


图 11.20 New Java Package 对话框



图 11.21 目录结构

11.9.2 编写 web.xml

建立 web.xml 文件，放在 mySSH/WEB-INF 目录下。web.xml 的示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>actionServlet</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!--初始参数-->
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!--处理所有后缀为 do 的请求-->
  <servlet-mapping>
```

```
<servlet-name>actionServlet</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

代码说明：/WEB-INF/struts-config.xml，说明 Struts 的配置文件在 WEB-INF 目录下。

11.9.3 编写用户注册页面 regedit.jsp

新建一个用户注册页面 regedit.jsp，包含“提交”按钮，放在 mySSH/WEB-INF/jsp 目录下。示例代码如下：

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title>Spring、Struts 和 Hibernate 整合</title></head>
<%
String msg = request.getAttribute("msg") == null ? "" : (String)request.getAttribute("msg");
%>
<body> <%= msg %>
    <form name="user" action="/mySSH/regedit.do" method="post">
        用户名 <input type="text" name="username" value="${user.username}"/><br>
        密 码 <input type="password" name="password" value=""/><br>
        <input type="submit" name="method" value="提交"/>
    </form>
</body>
</html>
```

代码说明：action="/mySSH/regedit.do"，表示在配置文件中要设定的动作映射为 regedit。

11.9.4 编写用户注册成功页面 success.jsp

新建一个用户注册成功页面 success.jsp，放在 mySSH/WEB-INF/jsp 目录下。示例代码如下：

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head><title>Spring、Struts 和 Hibernate 整合</title></head>
<body>
    提交成功：您输入的用户名是：${user.username}，密码是：${user.password}
</body>
</html>
```

11.9.5 建立数据库表结构

使用前面介绍的方法，在 MySQL 中创建一个数据库为 mySSH，用户名和密码都为 root，数据库中表名为 user，包含 3 个字段：id、username 和 password，设定 id 为主键。SQL 语

句如下:

```
CREATE TABLE `user` (  
  `id` VARCHAR( 32 ) NOT NULL ,  
  `username` VARCHAR( 32 ) NOT NULL ,  
  `password` VARCHAR( 32 ) NOT NULL ,  
  PRIMARY KEY ( `id` )  
) ENGINE = MYISAM ;
```

11.9.6 生成映射文件 User.hbm.xml 和 POJO

使用前面介绍的方法, 利用 Middlegen 生成映射文件 User.hbm.xml, 放在包 com.gc.vo 下。User.hbm.xml 的示例代码如下:

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
  "-//Hibernate/Hibernate Mapping DTD 2.0//EN"  
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >  
<hibernate-mapping>  
<!--  
  Created by the Middlegen Hibernate plugin 2.1  
  http://boss.bekk.no/boss/middlegen/  
  http://www.hibernate.org/  
-->  
<class  
  name="com.gc.vo.User"  
  table="user"  
>  
  <id  
    name="id"  
    type="java.lang.String"  
    column="id"  
  >  
    <generator class="uuid.hex" />  
  </id>  
  <property  
    name="username"  
    type="java.lang.String"  
    column="username"  
    not-null="true"  
    length="32"  
  />  
  <property  
    name="password"  
    type="java.lang.String"  
    column="password"  
    not-null="true"  
    length="32"  
  />  
</class>
```

```
<!-- Associations -->
</class>
</hibernate-mapping>
```

使用 hbm2java 根据映射文件生成 POJO，放在包 com.gc.vo 下。User.java 的示例代码如下：

```
/** ***** User.java ***** */
package com.gc.vo;
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;
/** @author Hibernate CodeGenerator */
public class User implements Serializable {
    /** identifier field */
    private String id;
    /** persistent field */
    private String username;
    /** persistent field */
    private String password;
    /** full constructor */
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    /** default constructor */
    public User() {
    }
    public String getId() {
        return this.id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getUsername() {
        return this.username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return this.password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String toString() {
        return new ToStringBuilder(this)
            .append("id", getId())
            .toString();
    }
}
```

11.9.7 编写接口 UserDao.java 和实现类 UserDaoImpl.java

在包 com.gc.dao 中编写一个接口类 UserDao.java。UserDao.java 示例代码如下：

```
/****** UserDao.java*****  
package com.gc.dao;  
import com.gc.vo.*;  
public interface UserDao {  
    //新增用户  
    public abstract void createUser(User user);  
    //修改用户  
    public abstract void updateUser(User user);  
    //删除用户  
    public abstract void deleteUser(User user);  
    //查询用户  
    public abstract User queryUser(String name);  
}
```

然后新建一个包 com.gc.dao.impl，在该包内编写接口 UserDao 的实现类 UserDaoImpl.java。UserDaoImpl.java 示例代码如下：

```
/****** UserDaoImpl.java*****  
package com.gc.dao.impl;  
import com.gc.dao.*;  
import com.gc.vo.*;  
import java.util.*;  
import org.apache.commons.logging.*;  
import org.hibernate.SessionFactory;  
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;  
  
public class UserDaoImpl extends HibernateDaoSupport implements UserDao {  
    private Logger logger = Logger.getLogger(this.getClass().getName());  
    //依赖注入 sessionFactory  
    private SessionFactory sessionFactory;  
    //定义查询语句 SQL  
    private String SQL = "from user u where u.username = ?";  
    //新增用户  
    public void createUser(User user) {  
        this.getHibernateTemplate().save(user);  
    }  
    //修改用户  
    public void updateUser(User user) {  
        this.getHibernateTemplate().update(user);  
    }  
    //删除用户  
    public void deleteUser(User user) {  
        this.getHibernateTemplate().delete(user);  
    }  
}
```



```
//查询用户
public User queryUser(String name){
    List userList;
    if (this.getHibernateTemplate().find(hql, name) == null )
        userList = new ArrayList();
    else
        userList = this.getHibernateTemplate().find(hql, name);
    return (User)userList.get(0);
}
```

11.9.8 编写 Struts 的配置文件 struts-config.xml

新建一个配置文档 struts-config.xml，放在 WEB-INF 目录下。示例代码如下：

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

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <!-- 定义 formbean -->
    <form-beans>
        <form-bean name="user" type="com.gc.vo.User"/>
    </form-beans>
    <action-mappings>
        <!-- path 为 URL 的路径 -->
        <action path="/regedit" type="org.springframework.web.struts.DelegatingActionProxy" name=
"user">
            <forward name="success" path="/WEB-INF/jsp/success.jsp"/>
            <forward name="input" path="/WEB-INF/jsp/regedit.jsp"/>
        </action>
        <action
            path="/input"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/jsp/regedit.jsp"/>
    </action-mappings>
    <!-- 注册 Struts 插件，与 Spring 相结合 -->
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation" value="/WEB-INF/config.xml" />
    </plug-in>
</struts-config>
```

代码说明：

- ☐ 注册 org.springframework.web.struts.ContextLoaderPlugIn，并说明 Spring 的配置文件是 WEB-INF 目录下的 config.xml。
- ☐ 使用 Spring 提供的代理类 org.springframework.web.struts.DelegatingActionProxy。

11.9.9 编写 Spring 的配置文件 config.xml

新建一个 Spring 的配置文件 config.xml，放在 WEB-INF 目录下。config.xml 的示例代码如下：

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

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="regedit" class="com.gc.service.impl.RegeditImpl">
        <property name="userDAO">
            <ref local="userDAO" />
        </property>
    </bean>
    <bean name="/regedit" class="com.gc.action.RegeditAction">
        <property name="regedit">
            <ref bean="regedit"/>
        </property>
    </bean>
    <bean name="/input" class="com.gc.action.RegeditAction">
        <property name="regedit">
            <ref bean="regedit"/>
        </property>
    </bean>
    <!--定义数据源-->
    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!--设定驱动-->
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <!--设定 URL-->
        <property name="url">
            <value>jdbc:mysql://localhost/mySSH</value>
        </property>
        <!--设定用户名-->
        <property name="username">
            <value>root</value>
        </property>
        <!--设定密码-->
        <property name="password">
            <value>root</value>
        </property>
    </bean>
    <!--和 Hibernate 联系起来-->
```

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref local="dataSource" />
  </property>
  <property name="mappingResources">
    <list>
      <value>com\gc\vo\User.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
</bean>
<!--进行事务处理-->
<bean id="transactionManager"
  class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>
<!--进行 DAO 代理-->
<bean id="userDAOProxy"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager" />
  </property>
  <property name="target">
    <ref local="userDAO" />
  </property>
  <!--当前没有事务，则新建一个事务-->
  <property name="transactionAttributes">
    <props>
      <prop key="create*">PROPAGATION_REQUIRED</prop>
      <prop key="update*">PROPAGATION_REQUIRED</prop>
      <prop key="delete*">PROPAGATION_REQUIRED</prop>
      <prop key="query*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
<bean id="userDAO" class="com.gc.dao.impl.UserDAOImpl">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>
</beans>
```

代码说明：注意定义动作映射的 Bean 和 Struts 的配置文件中的动作的对应关系。

11.9.10 编写控制器 RegeditAction.java

在 com.gc.action 包上，新建一个控制器 RegeditAction 类。RegeditAction.java 的示例代码如下：

```
/****** RegeditAction.java*****  
package com.gc.action;  
import java.io.IOException;  
import java.util.HashMap;  
import java.util.Map;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.apache.log4j.Logger;  
import org.apache.struts.action.*;  
import com.gc.service.Regedit;  
import com.gc.vo.User;  
public class RegeditAction extends Action {  
    private Logger logger = Logger.getLogger(this.getClass().getName());  
    //根据接口进行依赖注入  
    private Regedit regedit;  
    public Regedit getRegedit () {  
        return regedit;  
    }  
    public void setRegedit (Regedit regedit) {  
        this.regedit = regedit;  
    }  
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest  
                                request, HttpServletResponse response) throws Exception {  
        //保存页面提交的用户信息  
        getRegedit().saveUser((User)form);  
        request.setAttribute("user", (User)form);  
        return mapping.findForward("success");  
    }  
}
```

11.9.11 编写业务逻辑接口 Regedit.java

在 com.gc.service 包上，新建一个业务逻辑接口 Regedit 类。Regedit.java 的示例代码如下：

```
/****** Regedit.java*****  
package com.gc.service;  
import com.gc.vo.User;  
public interface Regedit {  
    //保存用户
```



```
public abstract void saveUser(User user);  
}
```

11.9.12 编写具体的业务逻辑类 RegeditImpl.java

在 com.gc.service.impl 包上，新建一个具体的业务逻辑类 RegeditImpl 类。RegeditImpl.java 的示例代码如下：

```
//***** RegeditImpl.java *****  
package com.gc.service.impl;  
  
import com.gc.service.Regedit;  
import com.gc.vo.User;  
public class RegeditImpl implements Regedit {  
    private UserDAO userDao ;  
    //保存用户  
    public void saveUser(User user) {  
        userDao.createUser(user);  
    }  
    //修改用户  
    public void updateUser(User user) {  
        userDao.updateUser(user);  
    }  
    //删除用户  
    public void deleteUser(User user) {  
        userDao.deleteUser(user);  
    }  
    //查询用户  
    public User queryUser(String username) {  
        return userDao.queryUser(username);  
    }  
    //依赖注入  
    public void setUserDAO (UserDAO userDao) {  
        this.userDao = userDao;  
    }  
    public UserDAO getUserDAO () {  
        return this.userDao;  
    }  
}
```

代码说明：private UserDAO userDao，增加接口定义，便于使用 Spring 来进行管理。所有代码编写完成后，mySSH 的目录结构如图 11.22 所示。

11.9.13 运行用户注册的例子

(1) 启动 Tomcat，在浏览器的地址栏中输入 <http://localhost:8080/mySSH/input.do>，按

Enter 键即可看到用户注册的页面，如图 11.23 所示。

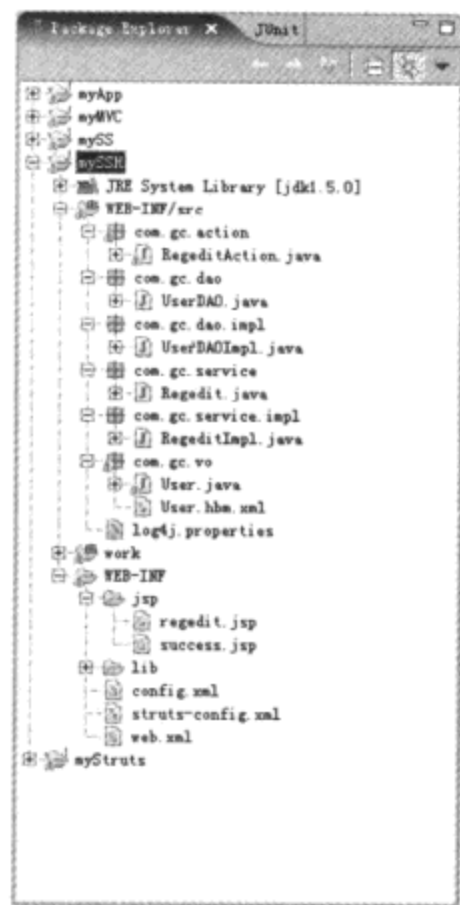


图 11.22 代码完成后 mySSH 的目录结构



图 11.23 用户注册的页面

(2) 在“用户名”文本框中输入“gd”，密码输入“123456”，然后单击“提交”按钮即可看到提交成功的页面，如图 11.24 所示。

(3) 再来输入一个用户名和密码，以做测试，在“用户名”文本框中输入“gf”，密码输入“123”，然后单击“提交”按钮即可看到再次提交成功的页面，如图 11.25 所示。



图 11.24 提交成功的页面



图 11.25 用户名文本框中输入“gf”，密码输入“123”提交成功的页面

(4) 启动 Tomcat，在浏览器的地址栏中输入 `http://localhost/phpmyadmin`，然后查看 MySQL 中的数据库 mySSH，表名为 user 中的内容，即可看到刚才新增成功的两笔数据，如图 11.26 所示。

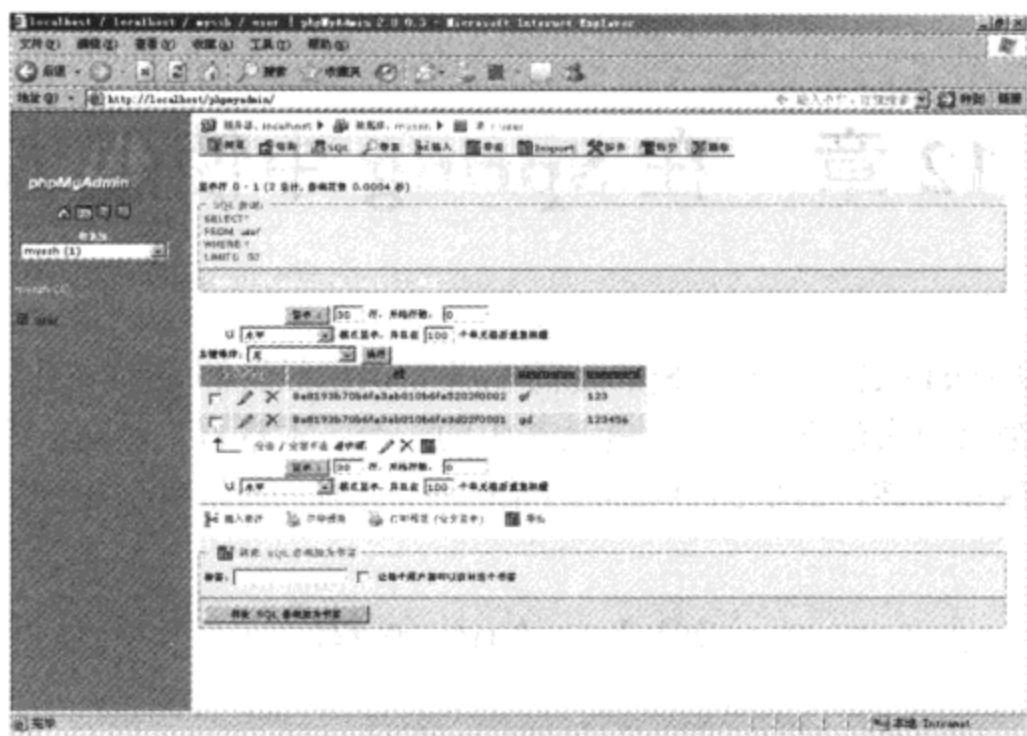


图 11.26 新增成功的两笔数据

11.10 小 结

本章首先讲述了 Hibernate 的下载和安装，接着讲解了 Hibernate 的配置和映射，以及 Hibernate 相关工具的使用，最后以一个 Struts、Spring 和 Hibernate 整合的示例为例，演示了如何整合 Struts、Spring 和 Hibernate。在接下来的两章里，将介绍开发人员在编程过程中经常使用的两个工具 Ant 和 Junit。



第 12 章 在 Spring 中使用 Ant

在介绍了 Hibernate 之后，关于 Spring 的相关知识已经讲解完了，但是有一些工具的使用会给开发人员带来很大的帮助，Ant 就是其中的一个。本章首先对 Ant 进行介绍，然后着重介绍 Ant 的下载和安装，最后结合前面的 Spring 示例，介绍 Ant 的使用方法。

12.1 Ant 介绍

Ant 是著名 Java 开源组织 Apache 的一个项目，是一个基于 Java 的 build 工具，用来编译、运行、测试 Java 程序。构建、包装和发布过程中几乎每一件事都可以由 Ant 的任务来处理。

它可以使开发人员通过 XML 发布项目，或者生成一些代码，执行 SQL 语言。总之，它可以帮助开发人员完成项目开发中除了开发代码以外的大部分辅助性工作。Ant 有如下优势：

- Ant 是基于 Java 的实现，具有良好的跨平台性。
- 可以通过增加新的 Java 类来扩展 Ant 的功能，而无须去了解不同平台上不同的脚本语言。
- Ant 的配置文件是基于 XML 的任务树，文件结构清晰、易读易写，能让开发人员运行各种各样的任务。

12.2 Ant 的下载和安装

仍然从最基础的下载和安装讲起。

12.2.1 下载 Ant

任何人可以从 <http://archive.apache.org/dist/ant/> 自由下载 Ant。Ant 的下载画面如图 12.1 所示。

单击 binaries 目录进去，可以下载 Ant 提供的最新稳定版本；单击 source 目录进去，可以下载 Ant 提供的最新源代码版本。

单击 binaries 目录进去，下载 Ant 的最新版本 1.6.5 版 apache-ant-1.6.5-bin.zip，下载后解压缩即可。

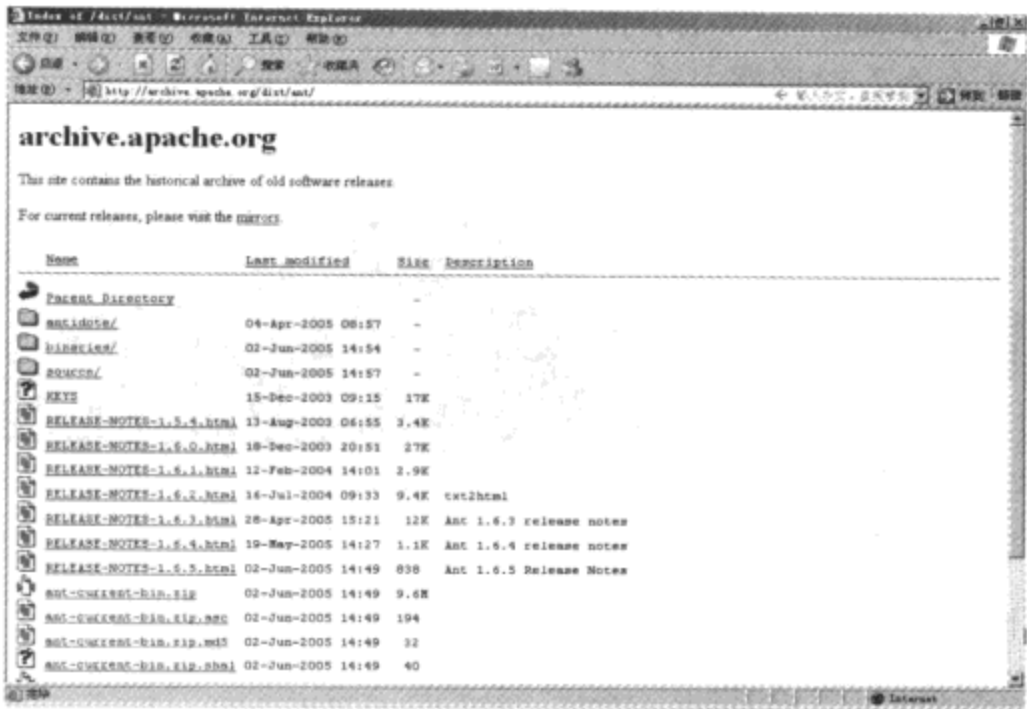


图 12.1 Ant 的下载画面

12.2.2 安装 Ant

下载 apache-ant-1.6.5-bin.zip 完毕，并解压缩到 D 盘根目录下后，即可开始配置环境变量。

- (1) 在 Windows 桌面中，右击“我的电脑”，在弹出的快捷菜单中选择“属性”命令，弹出“系统属性”对话框。
- (2) 在“系统属性”对话框中，单击“高级”选项卡下的“环境变量”按钮，弹出“环境变量”对话框，如图 12.2 所示。
- (3) 在“系统变量”选项区域中查找 Path 变量，选中 Path 变量，然后单击“编辑”按钮，如果没有则单击“新建”按钮，在弹出的“编辑系统变量”对话框中设定系统变量 Path，Path=D:\apache-ant-1.6.5\bin，如图 12.3 所示。



图 12.2 “环境变量”对话框

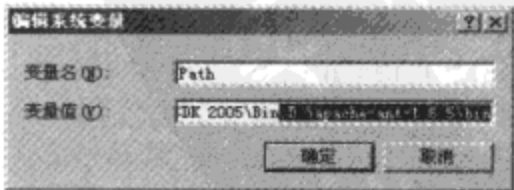


图 12.3 设定系统变量 Path

- (4) 单击“系统变量”选项卡中的“新建”按钮，在弹出的“编辑系统变量”对话框

中设定系统变量 ANT_HOME, ANT_HOME=D:\apache-ant-1.6.5, 如图 12.4 所示。

(5) 在“cmd 命令”对话框中输入 ant 命令, 即可查看 Ant 是否安装成功, 如图 12.5 所示。

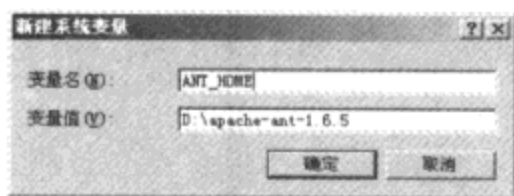


图 12.4 设定系统变量 ANT_HOME

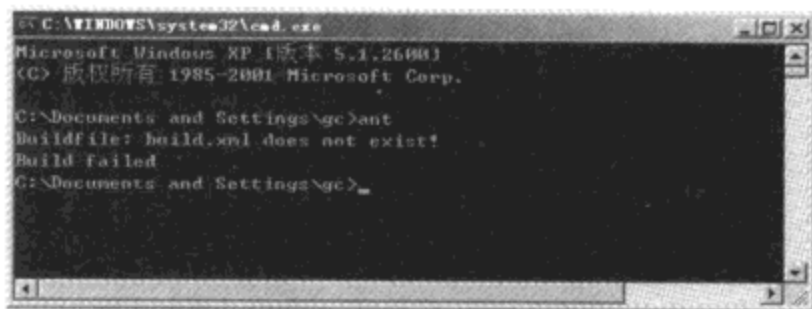


图 12.5 查看 Ant 是否安装成功

如果输出以下内容, 则说明 Ant 已经安装成功。

```
Buildfile: build.xml does not exist!  
Build failed
```

⚠注意: 因为在本书第 2 章已经设定好了 JAVA_HOME 系统变量, 所以这里不再设定它, 如果要单独使用 Ant, 则在配置环境变量之前, 应确认已经正确设置了 JAVA_HOME 系统变量。

12.3 在 Spring 中使用 Ant

这里以第 10 章中最后一个 Spring 和 Struts 整合的示例为例, 来演示如何使用 Ant 构建 jar 和 war。

12.3.1 在 Eclipse 中配置 Ant

因为 Eclipse 3.0 版本以上都把 Ant 集成进去了, 所以不用在 Eclipse 中配置 Ant, 即可直接使用。

12.3.2 建立 build.xml

在需要打包的工程中, 新建一个文件, 示例代码如下:

```
<?xml version="1.0"?>  
<!-- 定义一个 Ant 工程 mySS -->  
<project name="mySS" default="init" basedir=".">  
  <!-- 定义一些变量供后面使用 -->  
  <property name="mySS.home" value="." />  
  <property name="mySS.lib" value="${mySS.home}/WEB-INF/lib" />  
  <property name="mySS.jar" value="${mySS.home}/WEB-INF/lib" />  
  <property name="mySS.classes" value="${mySS.home}/WEB-INF/classes" />
```

```

<property name="tomcat.home" value="D:\jakarta-tomcat-5.5.5" />
<!--<property file="build.properties" />以上内容还可以定义在 build.properties 中-->
<!--定义 target，如果没有指定 target，则初始化该 target -->
<target name="init">
    <path id="all">
        <fileset dir="${mySS.lib}">
            <include name="**/*.jar" />
        </fileset>
        <fileset dir="${tomcat.home}/common/lib">
            <include name="*.jar" />
        </fileset>
    </path>
    <!--创建目录-->
    <mkdir dir="${mySS.classes}" />
</target>
<!--定义 target -->
<target name="clean">
    <!--删除目录-->
    <delete dir="${mySS.classes}">
</delete>
</target>
<!--定义 target，该 target 依赖于 init -->
<target name="compile" depends="init">
    <javac srcdir="${mySS.home}/WEB-INF/src" destdir="${mySS.classes}" target="1.5">
        <classpath refid="all" />
    </javac>
</target>
<!--定义 target，该 target 依赖于 compile -->
<target name="jar" depends="compile">
    <jar jarfile="${mySS.jar}/gc.jar" basedir="${mySS.classes}" includes="com/gc/**">
</jar>
</target>
<!--将 mySS 项目打包成 war 文件-->
<target name="war" depends="jar">
    <war destfile="${mySS.home}/mySS.war" webxml="${mySS.home}/WEB-INF/web.xml">
        <!--包含该文件夹下所有的内容 -->
        <fileset dir="${mySS.home}" casesensitive="yes">
            <include name="WEB-INF/**" />
            <exclude name="*.war" />
        </fileset>
        <lib dir="${mySS.home}/WEB-INF/lib">
            <include name="*.jar" />
        </lib>
    </war>
</target>
</project>

```

12.3.3 运行 Ant

在 Eclipse 中的 Outline 面板中，选中 jar，然后单击鼠标右键，在 Run As 的下一级菜单中选择 Ant Build 命令，即可构建 jar，如图 12.6 所示。

构建 jar 后，在 Eclipse 的控制面板会输出以下构建 jar 的信息，如图 12.7 所示。

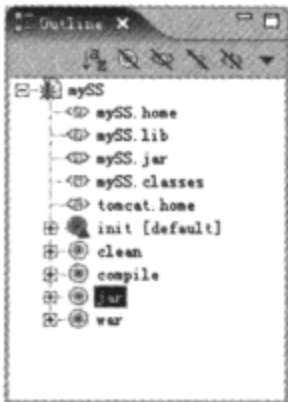


图 12.6 Eclipse 中的 OutLine 面板

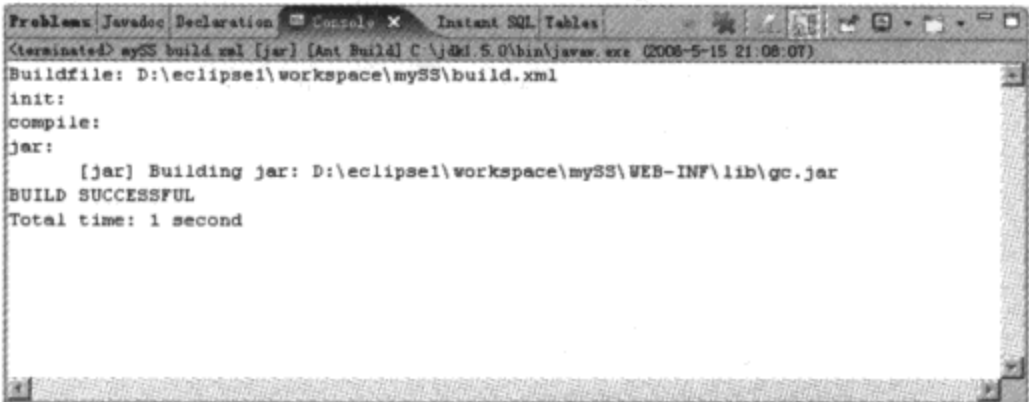


图 12.7 执行 build.xml 构建 jar 后输出的信息

如果选中 war，用同样的方法即可构建 war。在 Eclipse 的控制面板会输出以下构建 war 的信息，如图 12.8 所示。



图 12.8 执行 build.xml 构建 war 后输出的信息

12.4 小 结

本章首先讲述了 Ant 的下载和安装，接着又讲解了 Ant 的相关知识，最后以 Spring 和 Struts 整合的示例为例，演示了如何使用 Ant 构建 jar 和 war。

总体来说，Ant 的使用还是比较简单的，也是很有用的，同样，第 13 章中要讲的 Junit 也是一个很有用的工具。

第 13 章 在 Spring 中使用 Junit

第 12 章介绍的 Ant 主要用来构建、包装和发布 Java 程序，而对于 Java 程序的测试则由另外一个工具来完成，那就是 Junit。本章首先对 Junit 进行介绍，然后着重介绍 Junit 的下载和安装，最后结合前面 Spring 示例，介绍 Junit 的使用方法。

13.1 Junit 介绍

Junit 是由 Erich Gamma 和 Kent Beck 编写的回归测试框架，供 Java 编码人员做单元测试之用。在以前，开发人员写一个方法，代码如下：

```
//***** MathComputer.java*****
public Class MathComputer {
    public static int computer(int m, int n) {
        int num = m + n;
        return num;
    }
}
```

如果要对上述方法 computer()进行测试，通常这样来做，代码如下：

```
//***** MathComputer.java*****
public Class MathComputer {
    public static int computer(int m, int n) {
        int num = m + n;
        return num;
    }
    public static void main(String args[]) {
        if (computer(1, 2) == 3) {
            System.out.println("Test Ok");
        } else {
            System.out.println("Test Fail");
        }
    }
}
```

上面的测试依赖于结果的输出，这对于上面简单的方法还可以，但如果方法比较复杂，而且还牵涉到其他的方法，这样的测试方法就不能满足要求了。Junit 是这样一个测试工具，即它提供了更好的方法来进行单元测试。下面就主要来讲解 Junit 的使用方法。

13.2 Junit 的下载和安装

仍然从最基础的下载和安装讲起。

13.2.1 下载 Junit

任何人都可以从 <http://www.junit.org> 自由下载 Junit, Junit 的下载画面如图 13.1 所示。

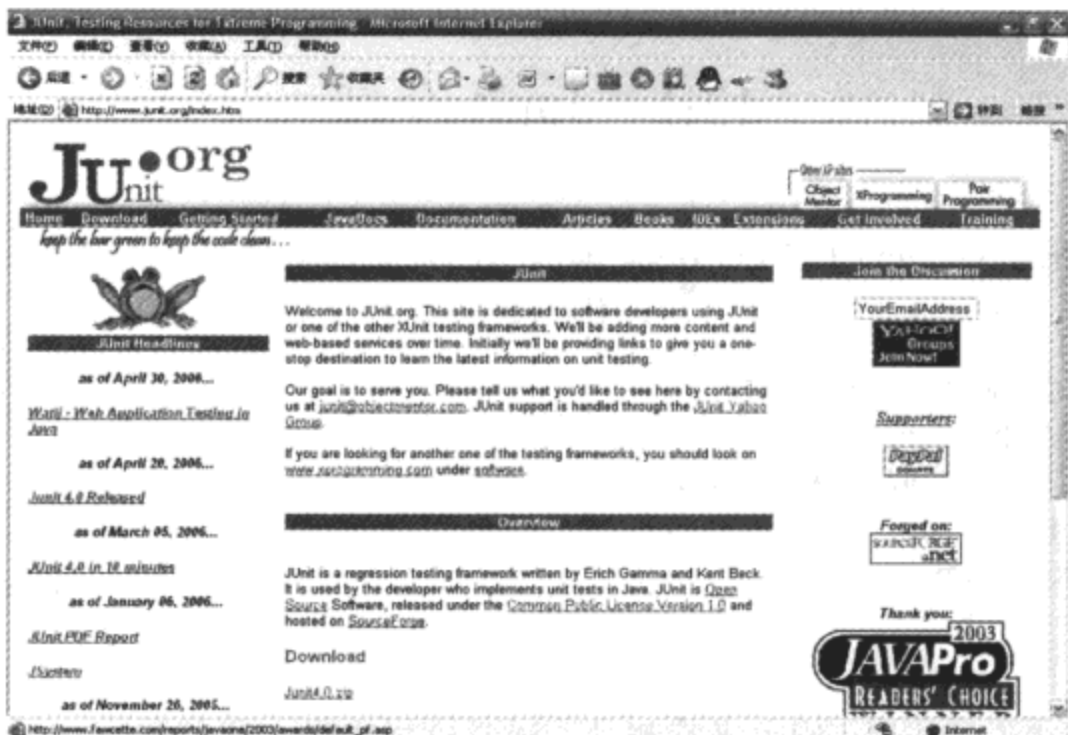


图 13.1 Junit 的下载画面

目前提供的最新版本是 4.0 版本, 下载 Junit 的最新版本 4.0 版 Junit4.0.zip, 下载后解压缩即可。

13.2.2 安装 Junit

下载 Junit4.0.zip 完毕, 并解压缩到 D 盘根目录下后, 即可开始配置环境变量。用前面介绍的设定系统变量的方法, 设定 CLASSPATH, CLASSPATH=***;D:\junit\junit.jar, 如图 13.2 所示。

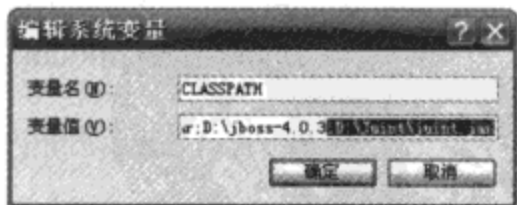


图 13.2 设定系统变量 CLASSPATH

13.3 在 Spring 中使用 Junit

这里以第 11 章中一个实现数据新增的 Hibernate 示例为例, 来演示如何使用 Junit 测试

程序。

13.3.1 在 Eclipse 中配置 Junit

因为 Eclipse 2.1 版本以上都把 Junit 集成进去了，所以不用在 Eclipse 中配置 Junit，即可直接使用。这里采用测试驱动的方式来编写程序，因为测试程序体现了设计的意图，首先编写测试程序导致编写的方法必须是可以测试的，这将有助于降低方法的耦合性。

13.3.2 扩展 TestCase 类

如果要使用 Junit，则测试类都必须继承 TestCase，然后使用 test+×××的方式来编写测试方法。对前面进行数学运算的方法编写测试案例，示例代码如下：

```
//***** TestMath.java*****  
import junit.framework.TestCase;  
public Class TestMath extends TestCase {  
    public void testcomputer() {  
        //断言计算结果与 2 是否相等  
        assertEquals(2, MathComputer.computer(1, 2));  
    }  
    public static void main(String args[]){  
        junit.swingui.TestRunner.run(TestMath.class);    }  
}
```

如果在进行单元测试之前需要初始化一些变量，在测试之后需要销毁一些变量，则可以使用 TestCase 类的 setUp()与 tearDown()方法。

在第 11 章中，针对实现数据新增的 Hibernate 示例演示时的测试代码如下：

```
//***** TestHibernate.java*****  
package com.gc.test;  
import org.hibernate.HibernateException;  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.Transaction;  
import org.hibernate.cfg.Configuration;  
import com.gc.vo.User;  
public class TestHibernate {  
    private SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();  
    public static void main(String[] args) throws HibernateException {  
        //根据配置文件获取 SessionFactory  
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();  
        User user = new User();  
        user.setPassword("gd");  
        user.setUsername("gd");  
        //一个 session 相当于一个连接  
        Session session = sessionFactory.openSession();
```

```
        //进行事务处理
        Transaction transaction = session.beginTransaction();
        session.save(user);
        transaction.commit();
        session.close();
        sessionFactory.close();
    }
}
```

如果改写为使用 Junit，则测试代码如下：

```
/****** TestHibernate.java*****
package com.gc.junit;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import com.gc.vo.User;
import junit.framework.TestCase;
public class TestHibernate extends TestCase {
    private SessionFactory sessionFactory = null;
    private Session session = null;
    //执行测试代码前进行初始化
    protected void setUp() throws Exception {
        super.setUp();
        sessionFactory = new Configuration().configure().buildSessionFactory();
        session = sessionFactory.openSession();
    }
    //具体的测试方法
    public void testCreate() {
        Transaction transaction = session.beginTransaction();
        User user = new User();
        user.setPassword("gd");
        user.setUsername("gd");
        session.save(user);
        transaction.commit();
    }
    //测试后的销毁方法
    protected void tearDown() throws Exception {
        super.tearDown();
        session.close();
        sessionFactory.close();
        sessionFactory = null;
    }
}
```

13.3.3 运行 Junit

在测试类上单击鼠标右键，然后在弹出的快捷菜单中选择 Run As→Junit Test 命令即可。

运行 Junit 后的视图如图 13.3 所示。

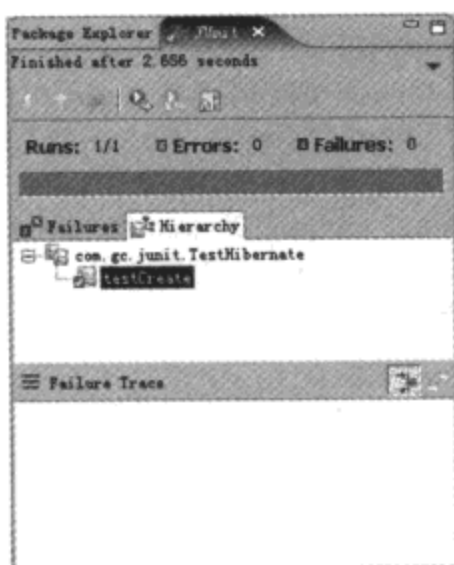


图 13.3 运行 Junit 后的视图

13.4 使用 Junit 时常用的一些判定方法

这里把使用 Junit 时常用的一些判定方法罗列出来：

- ☐ assertEquals (期望值, 实际值), 检查两个值是否相等。
- ☐ assertEquals (期望对象, 实际对象), 检查两个对象是否相等, 利用对象的 equals() 方法进行判断。
- ☐ assertSame (期望对象, 实际对象), 检查两个对象是否相等, 利用内存地址进行判断, 注意和上面 assertEquals 方法的区别。
- ☐ assertNotSame (期望对象, 实际对象), 检查两个对象是否不相等。
- ☐ assertNull (对象 1, 对象 2), 检查一个对象是否为空。
- ☐ assertNotNull (对象 1, 对象 2), 检查一个对象是否不为空。
- ☐ assertTrue (布尔条件), 检查布尔条件是否为真。
- ☐ assertFalse (布尔条件), 检查布尔条件是否为假。

13.5 利用 Ant 和 Junit 进行自动化测试

第 12 章讲过, Ant 可以用来自动构建 Java 程序, 当然也可以利用 Ant 和 Junit 进行自动化测试。

13.5.1 修改 build.xml

```
<?xml version="1.0"?>

<!--定义一个 Ant 工程 mySS-->
```



```

<project name="mySS" default="init" basedir=".">
  <!-- 定义一些变量供后面使用-->
  <property name="mySS.home" value="." />
  <property name="mySS.lib" value="${mySS.home}/WEB-INF/lib" />
  <property name="mySS.jar" value="${mySS.home}/WEB-INF/lib" />
  <property name="mySS.classes" value="${mySS.home}/WEB-INF/classes" />
  <property name="tomcat.home" value="D:/jakarta-tomcat-5.5.5" />
  <!--<property file="build.properties" />以上内容还可以定义在 build.properties 中-->
  <!-- 定义 target ， 如果没有指定 target ， 则初始化该 target -->
  <target name="init">
    <path id="all">
      <fileset dir="${mySS.lib}">
        <include name="**/*.jar" />
      </fileset>
      <fileset dir="${mySS.home}/WEB-INF/src">
        <include name="hibernate.cfg.xml" />
      </fileset>
    </path>
    <!-- 创建目录-->
    <mkdir dir="${mySS.classes}" />
  </target>
  <!-- 定义 target -->
  <target name="clean">
    <!-- 删除目录-->
    <delete dir="${mySS.classes}">
    </delete>
  </target>
  <!-- 定义 target ， 该 target 依赖于 init -->
  <target name="compile" depends="init">
    <javac srcdir="${mySS.home}/WEB-INF/src" destdir="${mySS.classes}" target="1.5">
      <classpath refid="all" />
    </javac>
  </target>
  <!-- 定义 target ， 该 target 依赖于 compile -->
  <target name="jar" depends="compile">
    <jar jarfile="${mySS.jar}/gc.jar" basedir="${mySS.classes}" includes="com/gc/**">
    </jar>
  </target>
  <!-- 将 mySS 项目打包成 war 文件-->
  <target name="war" depends="jar">
    <war destfile="${mySS.home}/mySS.war" webxml="${mySS.home}/WEB-INF/web.xml">
      <!-- 包含该文件夹下所有的内容 -->
      <fileset dir="${mySS.home}" casesensitive="yes">
        <include name="WEB-INF/**" />
        <exclude name="*.war" />
      </fileset>
      <lib dir="${mySS.home}/WEB-INF/lib">
        <include name="*.jar" />
      </lib>
    </war>
  </target>

```

```
<!--定义 target , 该 target 依赖于 compile -->
<target name="test" depends="compile">
    <junit printsummary="yes">
        <test name="com.gc.test.TestRegeditImpl"/>
        <classpath refid="all" />
        <classpath>
            <pathelement location="${mySS.classes}"/>
        </classpath>
    </junit>
</junitreport>
<!--产生报表的名称 -->
<fileset dir=".">
    <include name="TEST-*.xml"/>
</fileset>
<!--产生报表的路径 -->
<report format="frames" todir="reports/html"/>
</junitreport>
</target>
</project>
```

代码说明:

- ☐ printsummary="yes", 表示将测试结果显示出来。
- ☐ <test name="com.gc.test.TestRegeditImpl"/>, 定义测试类。
- ☐ <pathelement location="\${mySS.classes}"/>, 指定编译后 class 的存放位置。

13.5.2 运行 Ant

在 Eclipse 中的 Outline 面板中, 选中 test, 然后单击鼠标右键, 在弹出的快捷菜单中选择 Run As→Ant Build 命令, 即可自动测试, 如图 13.4 所示。此时在 Eclipse 中会看到出错的信息, 如图 13.5 所示。

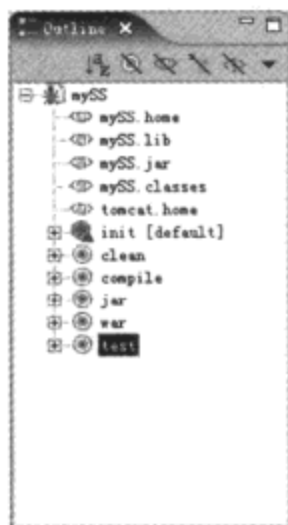


图 13.4 Eclipse 中的 Outline 面板



图 13.5 在 Eclipse 中会看到出错的信息

这是因为在 Ant 中<junit>任务是一个可选任务,所以需要 junit.jar。在 Ant 中添加 junit.jar 的方法如下:

(1) 在 Eclipse 中,选择 Window→Preferences 命令,会弹出 Preferences 对话框,如图 13.6 所示。

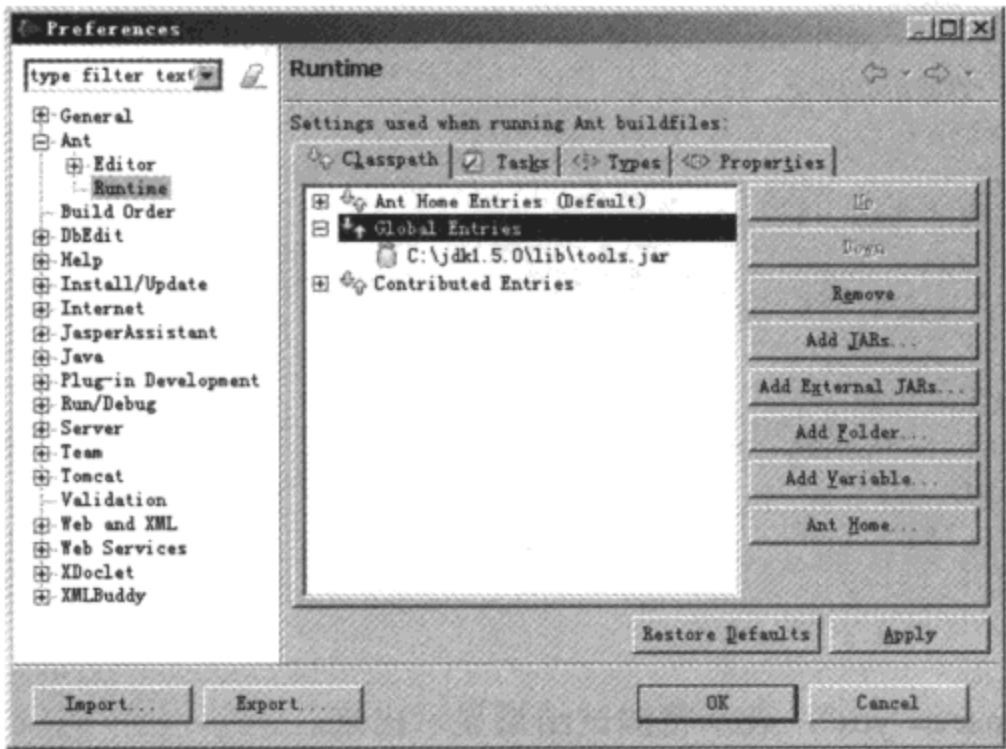


图 13.6 Preferences 对话框

(2) 在 Preferences 对话框中,选择 Ant 下的 Runtime,然后在对话框的右边选择 Global Entries,然后单击 add JARs...或 Add External JARs...按钮加入 junit.jar 即可。

13.5.3 自动生成测试报告

- (1) 把 D:\jakarta-tomcat-5.5.5\common\lib 下的 jar 复制到 mySS/WEB-INF/lib 下。
- (2) 运行 Ant 后,在 Eclipse 的控制面板会输出以下测试编译的信息,如图 13.7 所示。

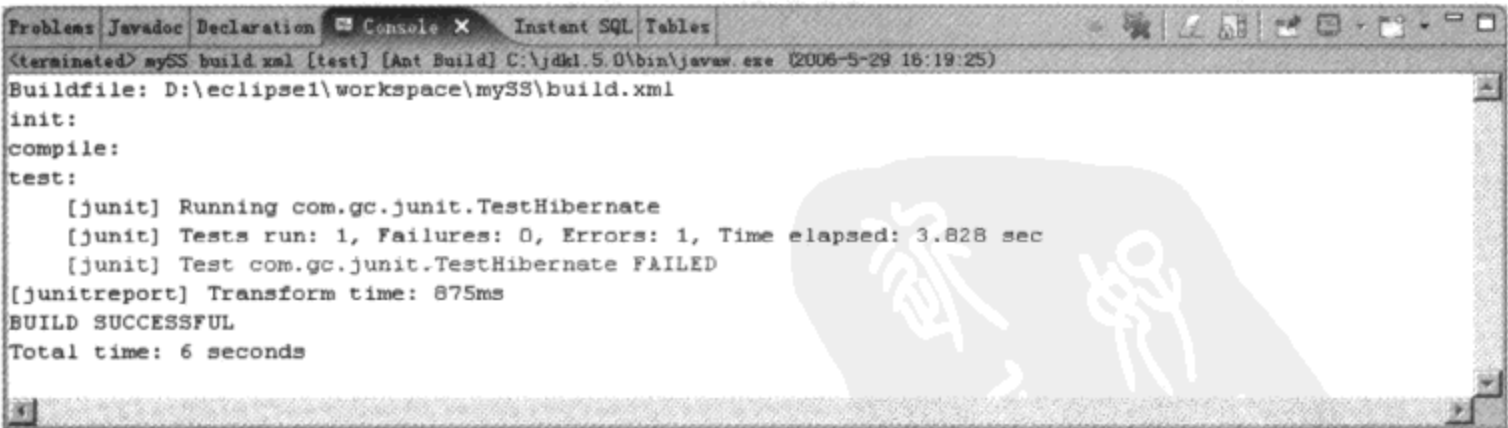


图 13.7 执行 build.xml 后输出的信息

(3) 打开 mySS/reports/html/index.html,即可看到自动生成的测试报告,如图 13.8 所示。

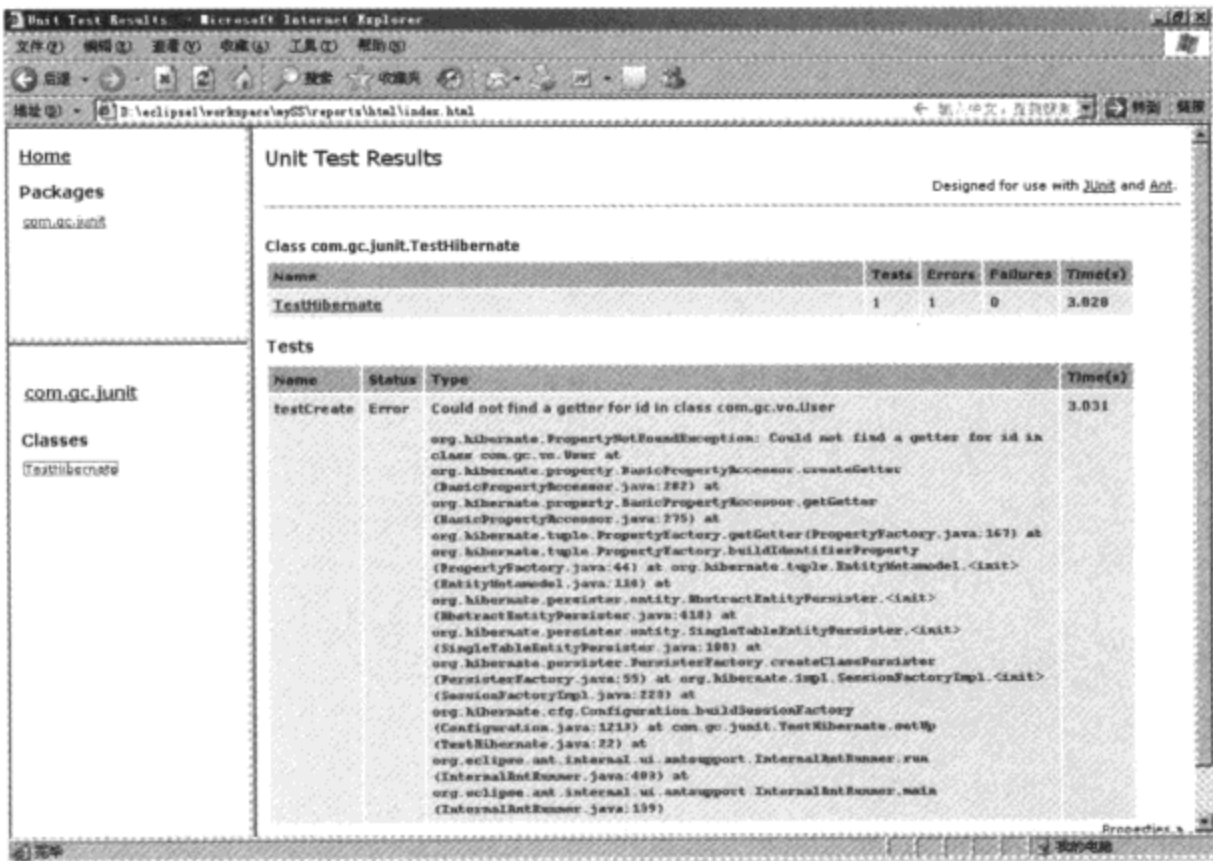


图 13.8 自动生成的测试报告

13.6 小 结

本章首先讲述了 Junit 的下载和安装，接着又讲解了 Junit 的相关知识，最后以第 10 章的 Spring 和 Struts 整合的示例为例，演示了如何使用 Junit 测试程序。

总体来说，Junit 的使用还是比较简单的，也是很有用的。与 Spring 相关的知识，到这里就已经讲完了，在第 14 章，笔者会通过一个比较大的实例，把涉及到 Spring 的相关知识串起来，从而使读者对此有一个全面的了解。



第 4 篇



Spring 实例

第 14 章 用 Spring 实现新闻发布系统实例

设计实例

第 14 章 用 Spring 实现新闻发布系统实例

前面的 13 章内容都是对 Spring 及其相关工具的介绍，通过对这些内容的讲述，让读者对 Spring 有一个全面的了解。前面讲过，实现实例是对 Spring 最好的理解方式，本章就是通过一个实例来对 Spring 进行一个整体的演示。在此主要使用 Spring 和 Hibernate 来实现新闻发布系统，从而使读者对 Spring 和 Hibernate 有一个全面的掌握。

因为 Spring 的 MVC 完全可以代替 Struts，所以这里只做 Spring 和 Hibernate 的整合，而不再考虑 Struts。

14.1 新闻发布系统的介绍

这个实例是一个新闻发布系统，主要包括用户的注册、权限的划分、新闻类别的设定、新闻的发布和新闻的浏览。首先注册用户，然后对用户进行授权，没有设定权限的用户只能浏览新闻，授权用户可以发布新闻、浏览新闻，普通用户不用授权即可浏览新闻。

14.2 检查环境配置

前面的章节中，都是在实例中进行环境配置的，这里再从头进行一下总结。很多时候，程序不能正常运行，都是因为环境配置的问题造成的。

这里主要检查 JDK、Tomcat、Spring、Hibernate 和 Ant 是否配置成功。

14.2.1 检查 JDK 配置

具体的安装步骤可以参看第 2 章的介绍。查看 JDK 是否配置成功，可以通过“cmd 命令”对话框来进行检查。在“cmd 命令”对话框中输入 Java，如果出现 java 相关选项的介绍信息，则说明 JDK 安装成功，如图 14.1 所示。

14.2.2 检查 Tomcat 配置

具体的安装步骤可以参看第 2 章的介绍。Tomcat 启动后，在 IE 地址栏中输入 <http://localhost:8080>，即可测试是否配置成功。Tomcat 启动成功的画面如图 14.2 所示。

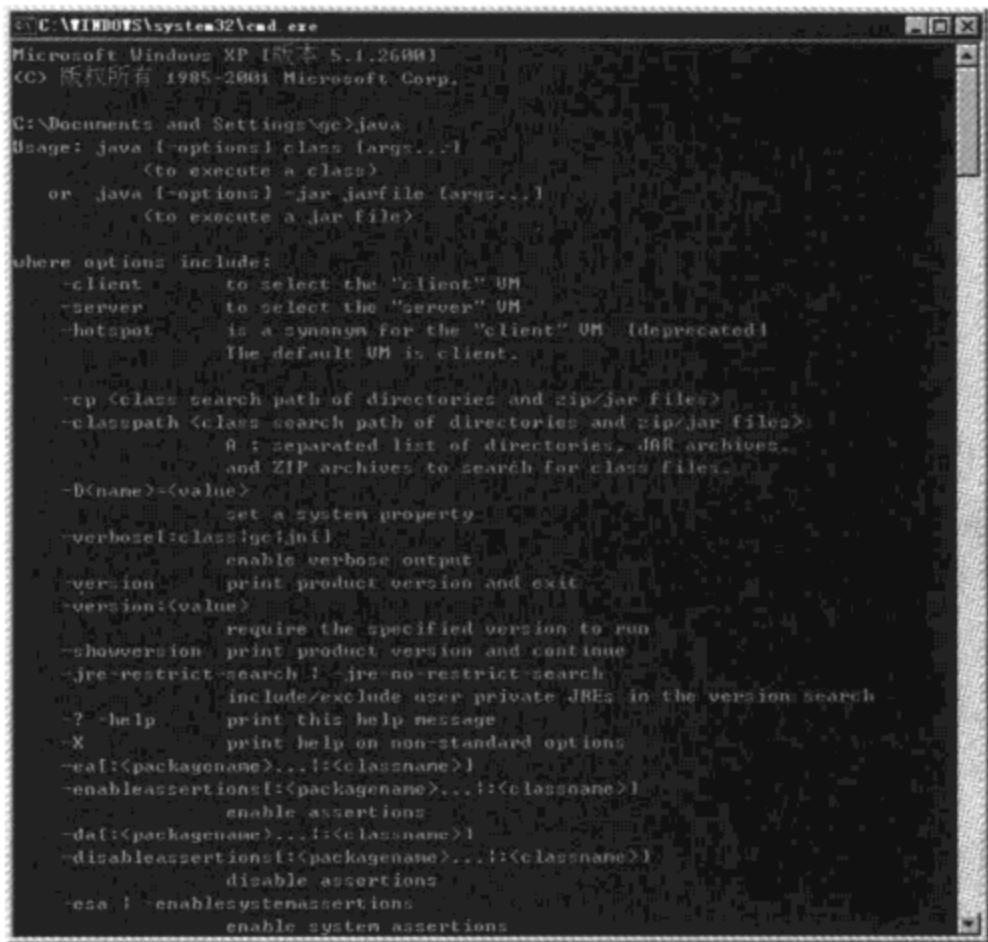


图 14.1 出现 Java 相关选项的介绍信息

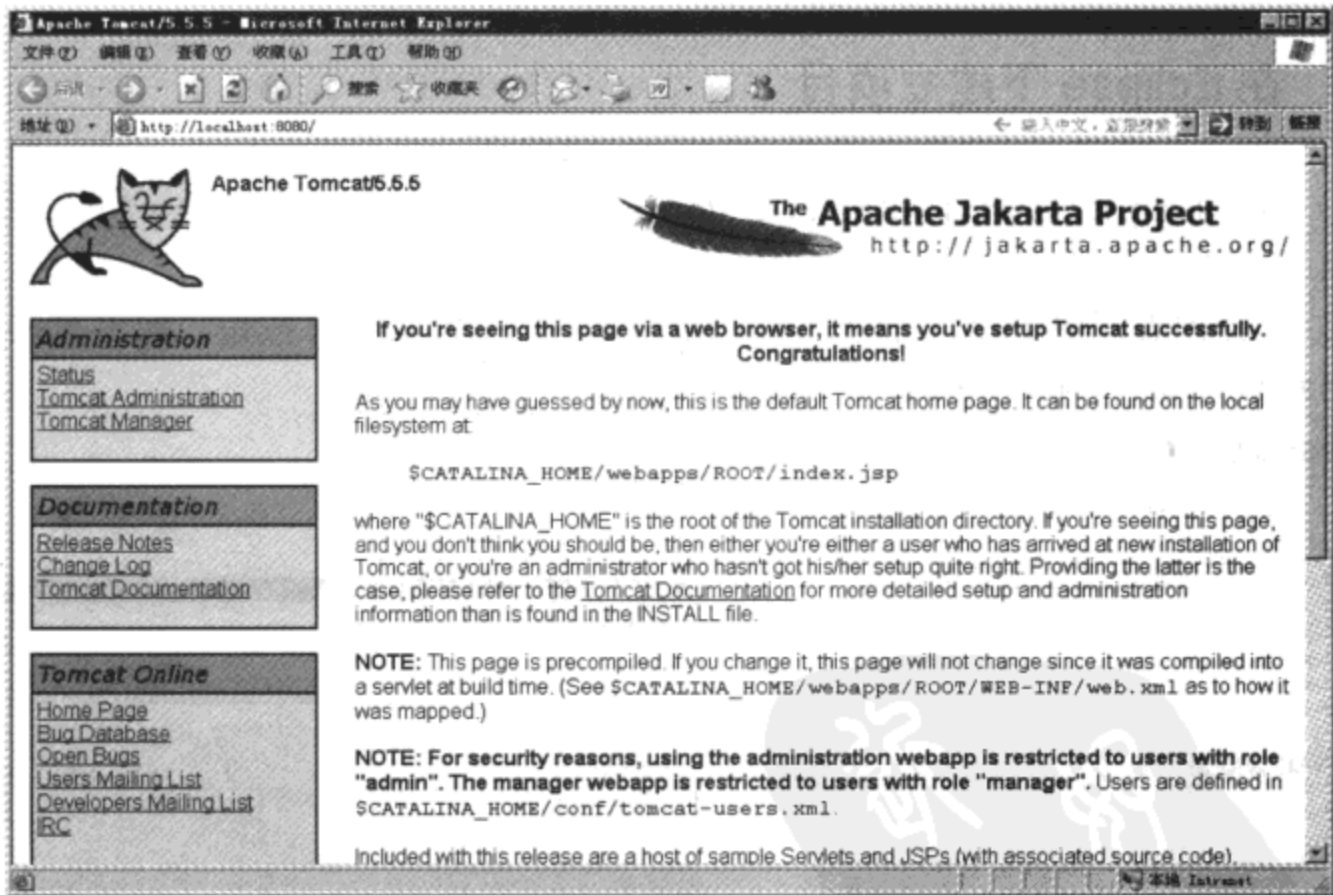


图 14.2 Tomcat 启动成功的画面

14.2.3 检查 Ant 配置

在“cmd 命令”对话框中输入 ant 命令，即可查看 Ant 是否安装成功，如图 14.3 所示。



图 14.3 查看 Ant 是否安装成功

如果输出以下内容，则说明 Ant 已经安装成功。

```
Buildfile: build.xml does not exist!  
Build failed
```

注意：虽然输出内容为 Build 失败，只是说明没有找到 build.xml，所以 Build 失败，但说明 Ant 已经可以运行。

14.3 在 Eclipse 下建立项目 myNews

在 Eclipse 下建立项目 myNews，并配置 Spring 和 Hibernate。

14.3.1 在 Eclipse 下建立项目

- (1) 运行 Eclipse，选择 File→New→Project 命令，Eclipse 将弹出 New Project 对话框，如图 14.4 所示。
- (2) 选择列表框中 Java 下的 Tomcat Project，然后单击 Next 按钮，弹出 New Tomcat Project 对话框，如图 14.5 所示。



图 14.4 New Project 对话框

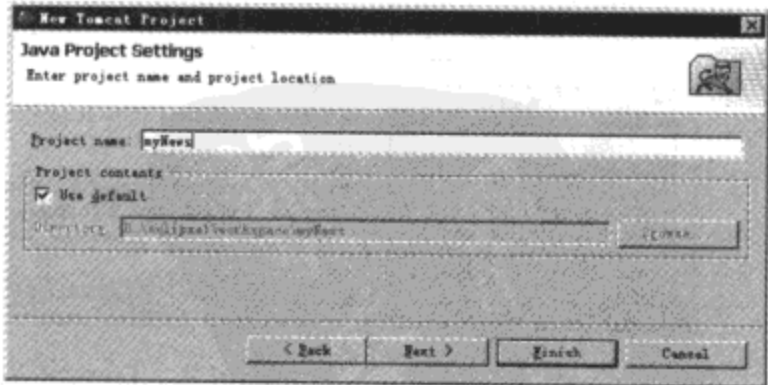


图 14.5 New Tomcat Project 对话框

(3) 在 New Tomcat Project 对话框中的 Project name 文本框中输入“myNews”，然后单击 Finish 按钮，项目即建立成功。myNews 的目录结构如图 14.6 所示。

(4) 把 log4j-1.2.9.jar、commons-logging.jar、spring.jar、antlr.jar、asm.jar、spring-hibernate3.jar、asm-attrs.jar、cglib.jar、commons-collections.jar、dom4j.jar、ehcache.jar、jta.jar、mysql-connector-java-5.0.0-beta-bin.jar、hibernate3.jar 这 14 个 jar 放在 /WEB-INF/lib/ 下以将其复制到 myNews/WEB-INF/lib 目录下，即 CLASSPATH 中。

(5) 用 Windows 自带的文本编辑器建立一个文件 log4j.properties，把 log4j.properties 放到 myNews/WEB-INF/src 目录下。

(6) 编辑 log4j.properties 文件。内容如下：

```
log4j.rootLogger=DEBUG,stdout,R

log4j.logger.org=ERROR, A1
log4j.logger.com.gd =DEBUG,A2
log4j.appender.A1=org.apache.log4j.RollingFileAppender
log4j.appender.A1.File=org.log
log4j.appender.A1.MaxFileSize=500KB
log4j.appender.A1.MaxBackupIndex=50
log4j.appender.A1.Append=true
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
log4j.appender.A2=org.apache.log4j.RollingFileAppender
log4j.appender.A2.File=gc.log
log4j.appender.A2.MaxFileSize=500KB
log4j.appender.A2.MaxBackupIndex=50
log4j.appender.A2.Append=true
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%d{ISO8601} - [%p] [%C{1}] - %m%n
#-----stdout-----
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#-----R-----
#log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
#this log file will be stored in web server's /bin directory,modify to your path which want to store.
log4j.appender.R.File=gf.log
#log4j.appender.R.datePattern='.'yyyy-MM-dd-HH-mm
log4j.appender.R.datePattern='.'yyyy-MM-dd
log4j.appender.R.append=true
## Keep one backup file
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss} %c - %m%n
#[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n
```



图 14.6 myNews 的目录结构

(7) 在 myNews 上单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，弹出 Properties for myNews 对话框，如图 14.7 所示。

(8) 在 Properties for myNews 对话框中，选择对话框左边列表框中的 Java Build Path。

(9) 在 Libraries 选项卡中，单击 Add JARs...按钮，弹出 JAR Selection 对话框，如图 14.8 所示。

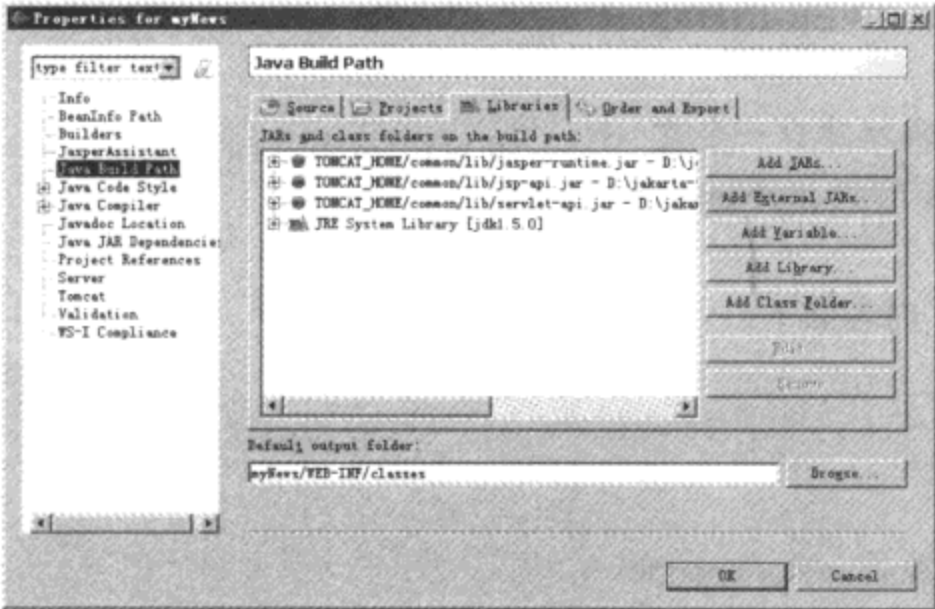


图 14.7 Properties for myNews 对话框



图 14.8 JAR Selection 对话框

(10) 在 JAR Selection 对话框中，打开列表框中的 myNews 一直到 lib 目录下出现 14 个 jar: log4j-1.2.9.jar、commons-logging.jar、spring.jar、antlr.jar、spring-hibernate3.jar、asm.jar、asm-attrs.jar、cglib.jar、commons-collections.jar、dom4j.jar、ehcache.jar、jta.jar、mysql-connector-java-5.0.0-beta-bin.jar、hibernate3.jar。

(11) 按住 Ctrl 键，选中这 14 个 jar，然后单击 OK 按钮返回到 Properties for myNews 对话框，如图 14.9 所示。

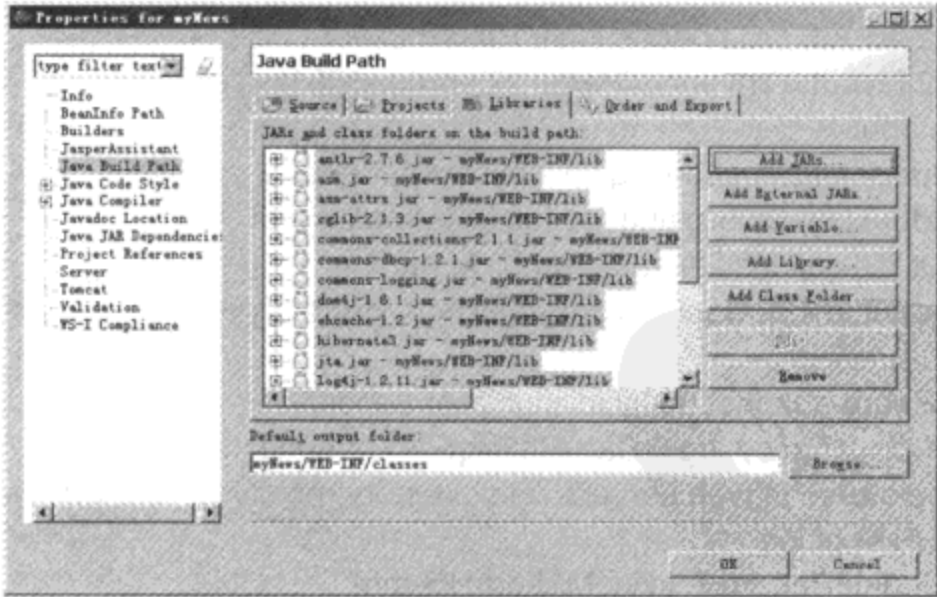


图 14.9 Properties for myNews 对话框

(12) 在 Properties for myNews 对话框中单击 OK 按钮，即可完成对 Spring 和 Hibernate 的配置。

(13) 再次在 myNews 上单击鼠标右键，在弹出的快捷菜单中选择 New→Package 命

令，将弹出 New Java Package 对话框，如图 14.10 所示。

注意：在 spring.jar 中，没有包含对 Hibernate 的支持。如果需要使用 Hibernate，则需要把 spring-framework-2.0-m1\dist\extmodules\spring-hibernate3.jar 也包含进 CLASSPATH 中。

(14) 在 New Java Package 对话框的 Name 文本框中输入“com.gd.action”，然后单击 Finish 按钮，即可建立 com.gd.action 包。

(15) 用同样的方法建立 com.gd.service 包、com.gd.service.impl 包、com.gd.dao 包、com.gd.dao.impl 包和 com.gd.vo 包、com.gd.po 包。

(16) 在 WEB-INF 目录下建立 jsp 文件夹。

(17) 最终配置好 Spring 和 Hibernate 的 myNews 项目的目录结构如图 14.11 所示。

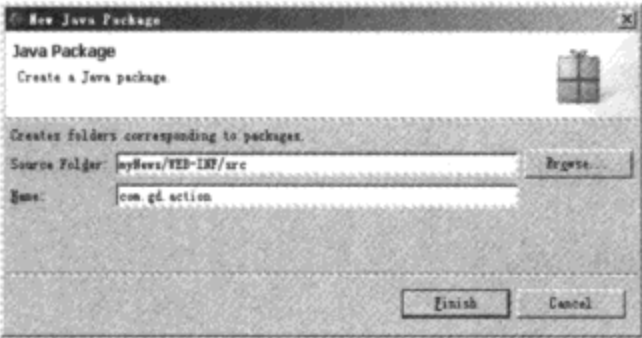


图 14.10 New Java Package 对话框



图 14.11 配置好 Spring 和 Hibernate 的 myNews 项目的目录结构

14.3.2 编写 Ant build 文件

在 myNews 目录下新建一个 Ant 文件 build.xml。示例代码如下：

```
<?xml version="1.0"?>

<project name="myNews" default="init" basedir=".">
  <property name="myNews.home" value="." />
  <property name="myNews.lib" value="${myNews.home}/WEB-INF/lib" />
  <property name="myNews.jar" value="${myNews.home}/WEB-INF/lib" />
  <property name="myNews.classes" value="${myNews.home}/WEB-INF/classes" />
  <property name="tomcat.home" value="D:\jakarta-tomcat-5.5.5" />
  <!--<property file="build.properties" />以上内容还可以定义在 build.properties 中-->
  <target name="init">
    <path id="all">
      <fileset dir="${myNews.lib}">
        <include name="**/*.jar" />
      </fileset>
      /*用来定义所包含的 jar*/
      <fileset dir="${tomcat.home}/common/lib">
        <include name="*.jar" />
      </fileset>
    </path>
    <mkdir dir="${myNews.classes}" />
  </target>
```



```

<target name="clean">
    <delete dir="${myNews.classes}">
    </delete>
</target>
<target name="compile" depends="init">
    /*用来定义要编译的源文件和编译后的路径*/
    <javac srcdir="${myNews.home}/WEB-INF/src" destdir="${myNews.classes}"
target="1.5">
        <classpath refid="all" />
    </javac>
</target>
<target name="jar" depends="compile">
    /*用来定义打包后 jar 的名字和路径*/
    <jar jarfile="${myNews.jar}/gd.jar" basedir="${myNews.classes}" includes="com/gd/**">
    </jar>
</target>
<!--将 myNews 项目打包成 war 文件-->
<target name="war" depends="jar">
    <war destfile="${myNews.home}/myNews.war"
webxml="${myNews.home}/WEB-INF/web.xml">
        <fileset dir="${myNews.home}" casesensitive="yes">
            <include name="WEB-INF/**" />
            <exclude name="*.war" />
        </fileset>
        <lib dir="${myNews.home}/WEB-INF/lib">
            <include name="*.jar" />
        </lib>
    </war>
</target>
</project>

```

14.3.3 配置 web.xml 文件

建立 web.xml 文件，放在 myNews/WEB-INF 目录下。web.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    .....
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/conf/dispatcherServlet-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>

```

```
<servlet-name>dispatcherServlet </servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
<taglib>
  <taglib-uri>/spring</taglib-uri>
  <taglib-location>/WEB-INF/tld/spring.tld</taglib-location>
</taglib>
.....
</web-app>
```

代码说明：

- ❑ /WEB-INF/conf/dispatcherServlet-servlet.xml，为了统一管理，在 WEB-INF 下新建一个文件夹 conf，把配置文件放在 /WEB-INF/conf/ 下。
- ❑ /WEB-INF/tld/spring.tld，为了统一管理，在 WEB-INF 下新建一个文件夹 tld，把自定义标签文件放在 /WEB-INF/tld/ 下。

14.4 设计新闻发布系统

上面对环境配置完毕。在开始编码之前，先来设计新闻发布系统，包括设计页面、设计业务逻辑和设计数据库。

14.4.1 设计页面

为了示例方便，这里的页面都没有使用图片。

从前面的实例说明可以知道，这个实例需要如下一些页面：新闻发布的展示页面 show.html、发布新闻页面 release.html、用户注册页面 regedit.html、管理员登录页面 login.html 和错误处理页面 error.html。

(1) 新闻发布的展示页面 show.html，如图 14.12 所示。

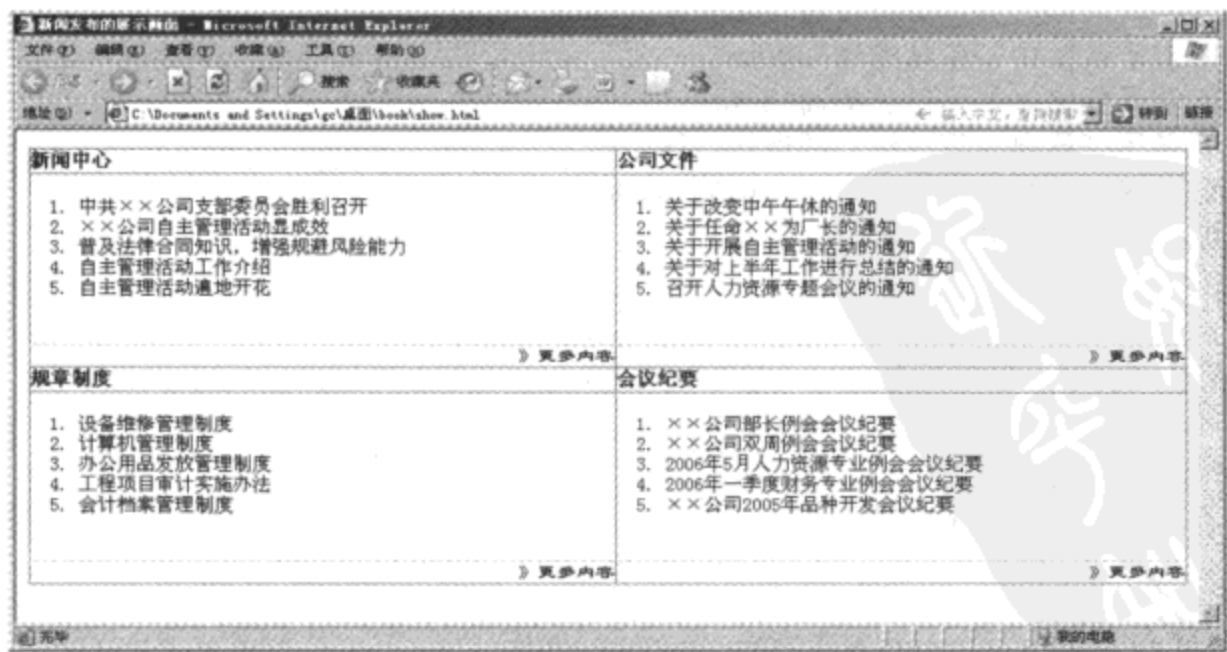


图 14.12 新闻发布的展示页面

该页面主要用来显示用户发布的新闻，并按照新闻类别来显示。每个新闻类别下显示 5 条新闻，其他新闻使用“更多内容”来进行查看。其中新闻类别分为新闻中心、公司文件、规章制度和会议纪要。

show.html 的源代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>新闻发布的展示页面</title>

<style type="text/css">
<!--
.style1 {font-family: "隶书"}
-->
</style>
</head>

<body>
<table width="100%" height="100%" border="1" cellpadding="0" cellspacing="0" >
  <tr height="100%">
    <td height="20"><strong>新闻中心</strong></td>
    <td><strong>公司文件</strong></td>
  </tr>
  <tr height="100%">
    <td height="150"><ol>
      <li>中共××公司支部委员会胜利召开</li>
      <li>××公司自主管理活动显成效</li>
      <li>普及法律合同知识，增强规避风险能力</li>
      <li>自主管理活动工作介绍</li>
      <li>自主管理活动遍地开花</li>
    </ol></td>
    <td><ol>
      <li>关于改变中午午休的通知</li>
      <li>关于任命××为厂长的通知</li>
      <li>关于开展自主管理活动的通知</li>
      <li>关于对上半年工作进行总结的通知</li>
      <li>召开人力资源专题会议的通知</li>
    </ol></td>
  </tr>
  <tr height="100%" style="border-top-width:0">
    <td height="15" style="border-top-width:0"><div align="right" class="style1" >》更多内容</div></td>
    <td style="border-top-width:0"><div align="right" class="style1" >》更多内容</div></td>
  </tr>
  <tr height="100%">
    <td height="20"><strong>规章制度</strong></td>
    <td><strong>会议纪要</strong></td>
```



```
</tr>
<tr height="100%">
  <td height="150"><ol>
    <li>设备维修管理制度</li>
    <li>计算机管理制度</li>
    <li>办公用品发放管理制度</li>
    <li>工程项目审计实施办法</li>
    <li>会计档案管理制度</li>
  </ol></td>
  <td><ol>
    <li>××公司部长例会会议纪要</li>
    <li>××公司双周例会会议纪要</li>
    <li>2006 年 5 月人力资源专业例会会议纪要</li>
    <li>2006 年一季度财务专业例会会议纪要</li>
    <li>××公司 2005 年品种开发会议纪要</li>
  </ol></td>
</tr>
<tr height="100%">
  <td height="15" style="border-top-width:0"><div align="right" class="style1">》更多内容</div></td>
  <td style="border-top-width:0"><div align="right" class="style1">》更多内容</div></td>
</tr>
</table>
</body>
</html>
```

(2) 发布新闻页面 release.html，如图 14.13 所示。

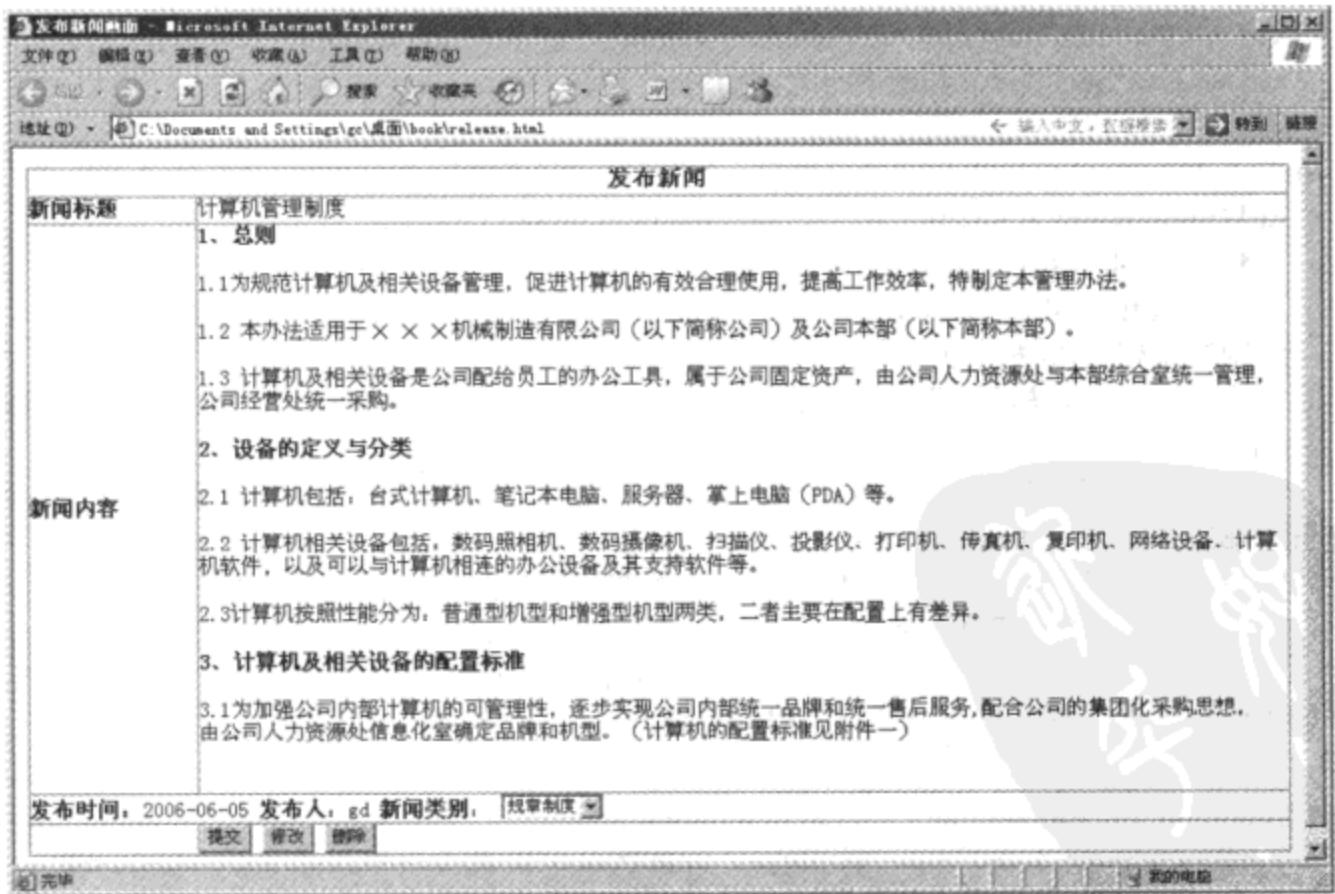


图 14.13 发布新闻页面

该页面主要用来发布新闻，用户填写新闻标题、新闻内容和选择新闻类别，系统自动输出发布时间和发布人。

release.html 的源代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>发布新闻页面</title>
<style type="text/css">
<!--
.style1 {
    font-size: large;
    font-weight: bold;
}
-->
</style>
</head>

<body>
<form name="form1" method="post" action="">
    <table width="100%" height="160" border="1" cellpadding="0" cellspacing="0">
        <tr>
            <td height="17" colspan="2"><div align="center" class="style1">发布新闻</div></td>
        </tr>
        <tr>
            <td width="126" height="19"><strong>新闻标题</strong></td>
            <td width="560">计算机管理制度</td>
        </tr>
        <tr>
            <td height="73"><strong>新闻内容</strong></td>
            <td><p><strong>1</strong><strong>、总则</strong></p>
                <p align="left">1.1 为规范计算机及相关设备管理，促进计算机的有效合理使用，提高工作效率，特制定本管理办法。</p>
                <p>1.2 本办法适用于×××机械制造有限公司（以下简称公司）及公司本部（以下简称本部）。</p>
                <p>1.3 计算机及相关设备是公司配给员工的办公工具，属于公司固定资产，由公司人力资源处与本部综合室统一管理，公司经营处统一采购。</p>
                <p><strong>2</strong><strong>、设备的定义与分类</strong></p>
                <p>2.1 计算机包括：台式计算机、笔记本电脑、服务器、掌上电脑（PDA）等。</p>
                <p>2.2 计算机相关设备包括：数码照相机、数码摄像机、扫描仪、投影仪、打印机、传真机、复印机、网络设备、计算机软件，以及可以与计算机相连的办公设备及其支持软件等。</p>
                <p>2.3 计算机按照性能分为：普通型机型和增强型机型两类，二者主要在配置上有差异。<strong>&nbsp;</strong></p>
                <p><strong>3</strong><strong>、计算机及相关设备的配置标准</strong></p>
                <p>3.1 为加强公司内部计算机的可管理性，逐步实现公司内部统一品牌和统一售后服务,配合公司的集团化采购思想，由公司人力资源处信息化室确定品牌和机型。（计算机的配置标准见附件一）</p>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

```

        <p>&nbsp;</p>
    </td>
</tr>
<tr>
    <td height="19" colspan="2"><strong>发布时间: </strong>2006-06-05 <strong>发布人
</strong>: gd <strong>新闻类别</strong>:
        <select name="select">
            <option>新闻中心</option>
            <option selected>规章制度</option>
            <option>会议纪要</option>
            <option>公司文件</option>
        </select></td>
</tr>
<tr>
    <td height="18">&nbsp;</td>
    <td><input type="submit" name="Submit" value="提交">
    <input type="submit" name="Submit" value="修改">
    <input type="submit" name="Submit" value="删除"></td>
</tr>
</table>
</form>
</body>
</html>
```

(3) 用户注册页面 regedit.html，如图 14.14 所示。

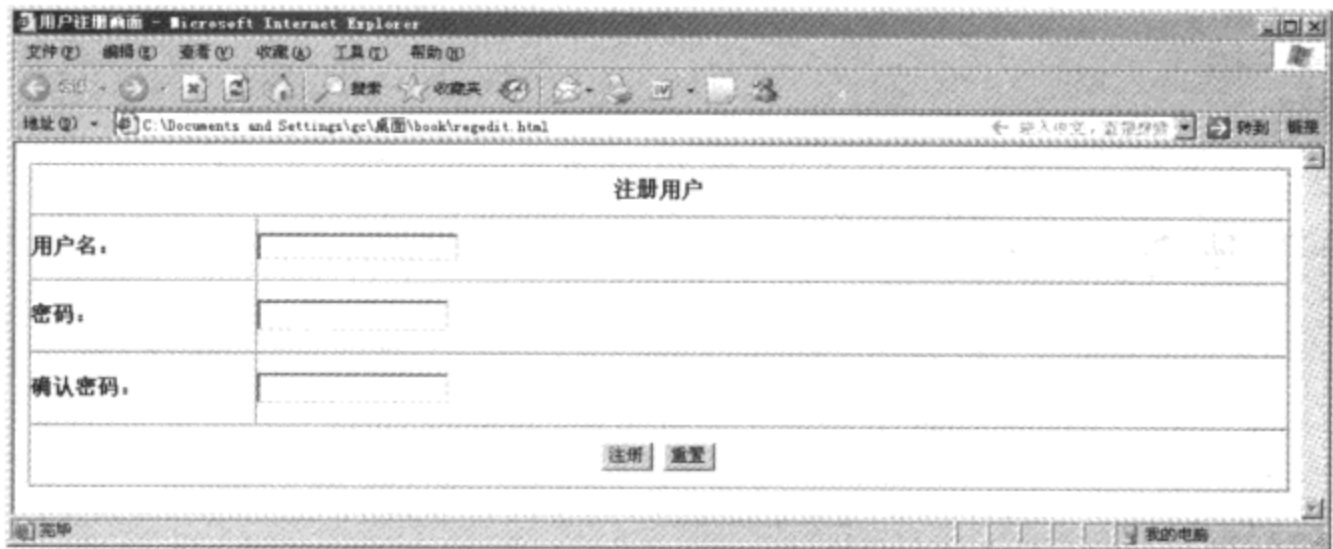


图 14.14 用户注册页面

该页面主要用来注册用户，包括注册用户名和密码。
regedit.html 的源代码如下：

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>用户注册画面</title>
</head>
```

```
<body>
<form name="form1" method="post" action="">
  <table width="100%" height="251" border="1" cellpadding="0" cellspacing="0">
    <tr>
      <td height="17" colspan="2"><div align="center"><strong>注册用户</strong></div></td>
    </tr>
    <tr>
      <td width="18%"><strong>用户名: </strong></td>
      <td width="82%"><input type="text" name="textfield"></td>
    </tr>
    <tr>
      <td><strong>密码: </strong></td>
      <td><input type="password" name="textfield"></td>
    </tr>
    <tr>
      <td><strong>确认密码: </strong></td>
      <td><input type="password" name="textfield"></td>
    </tr>
    <tr>
      <td colspan="2"><div align="center">
        <input type="submit" name="Submit" value="注册">
        <input type="reset" name="Submit" value="重置">
      </div></td>
    </tr>
  </table>
</form>
</body>
</html>
```

(4) 管理员登录页面 login.html，如图 14.15 所示。

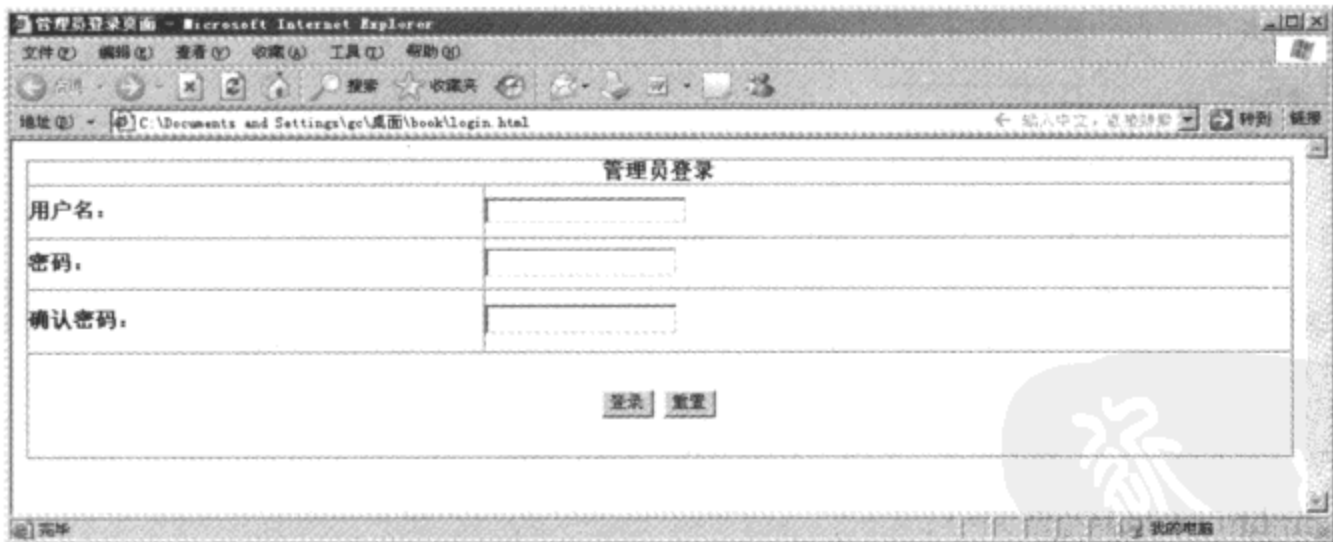


图 14.15 管理员登录页面

该页面主要用来进行新闻发布前的登录，如果管理员登录成功，则直接跳转至发布新闻的页面。

login.html 的源代码如下：

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



```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>管理员登录页面</title>
</head>

<body>
<form name="form1" method="post" action="">
  <table width="100%" border="1" cellpadding="0" cellspacing="0">
    <tr>
      <td colspan="2"><div align="center"><strong>管理员登录</strong></div></td>
    </tr>
    <tr>
      <td height="41"><strong>用户名: </strong></td>
      <td><input type="text" name="textfield"></td>
    </tr>
    <tr>
      <td height="40"><strong>密码: </strong></td>
      <td><input type="password" name="textfield"></td>
    </tr>
    <tr>
      <td height="49"><strong>确认密码: </strong></td>
      <td><input type="password" name="textfield"></td>
    </tr>
    <tr>
      <td height="83" colspan="2"><div align="center">
        <input type="submit" name="Submit" value="登录">
        <input type="reset" name="Submit" value="重置">
      </div></td>
    </tr>
  </table>
</form>
</body>
</html>
```

(5) 错误处理页面 error.html, 如图 14.16 所示。

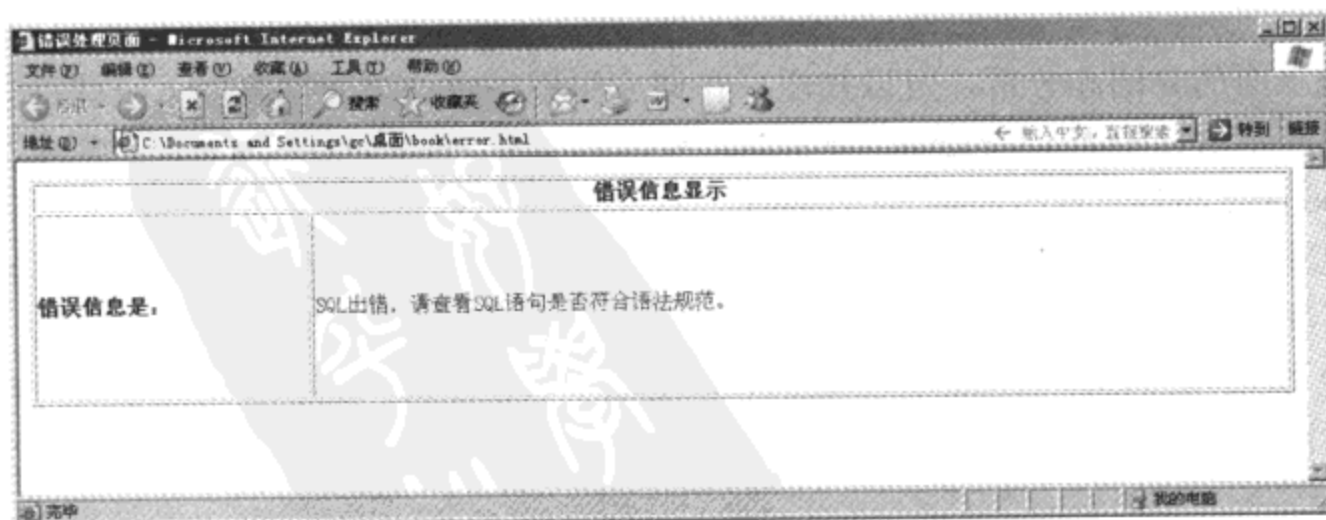


图 14.16 错误处理页面

该页面主要用来显示在示例运行过程中的错误信息。

error.html 的源代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>错误处理页面</title>
<style type="text/css">
<!--
.style1 {
    color: #000000;
    font-weight: bold;
}
.style2 {color: #FF0000}
-->
</style>
</head>

<body>
<table width="100%" border="1">
  <tr>
    <td colspan="2"><div align="center"><strong>错误信息显示</strong></div></td>
  </tr>
  <tr>
    <td width="22%" height="141"><span class="style1">错误信息是: </span></td>
    <td width="78%"><span class="style2">SQL 出错, 请查看 SQL 语句是否符合语法规范。
</span></td>
  </tr>
</table>
</body>
</html>
```

14.4.2 设计持久化类

通过上面的介绍和分析, 可以知道, 在该应用程序中, 至少需要以下一些持久化类: 负责用户基本信息的类 User.java、负责用户权限的类 UsersAuthor.java、负责新闻类别的类 NewsType.java 和负责新闻的类 News.java。

这里使用 Together 来画 UML 图。对于 Together 的使用方法这里不再进行讲解, 读者可以使用其他的工具, 方法基本类似。这里首先画一个整体包结构图, 如图 14.17 所示。

(1) 负责用户基本信息的类 User.java, 主要用来存储用户的基本信息以及对用户的信息进行验证。用户类的 UML 图如图 14.18 所示。

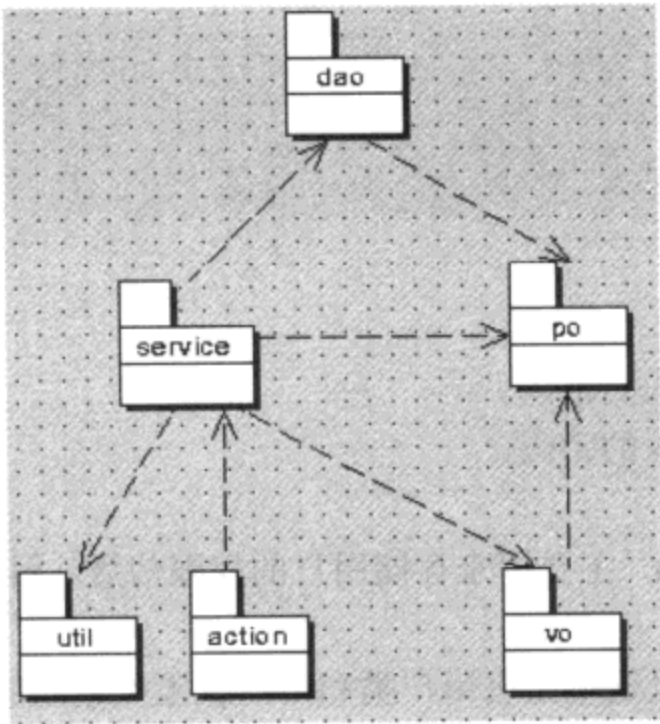


图 14.17 整体包结构图

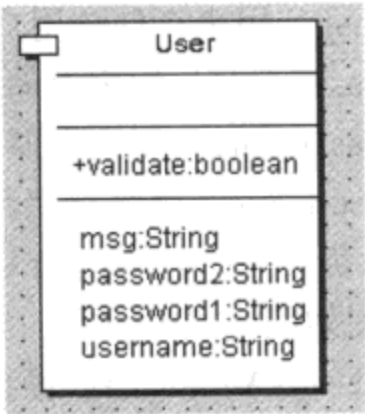


图 14.18 用户类的 UML 图

由 Together 自动生成的用户类的源代码如下：

```
//***** User.java*****
package com.gd.vo;
public class User {
    public String getMsg(){
return msg;
}

    public void setMsg(String msg){
this.msg = msg;
}

    public String getPassword2(){
return password2;
}

    public void setPassword2(String password2){
this.password2 = password2;
}

    public String getPassword1(){
return password1;
}

    public void setPassword1(String password1){
this.password1 = password1;
}

    public String getUsername(){
return username;
}

    public void setUsername(String username){
this.username = username;
}

    public boolean validate() {
}
```

```
private String msg;
private String password2;
private String password1;
private String username;
}
```

代码说明：

- ❑ msg，用来存储该用户类的消息。
- ❑ password1，用来存储用户第一次输入的密码。
- ❑ password2，用来存储用户第二次输入的密码。
- ❑ username，用来存储用户名。

(2) 负责用户权限的类 UsersAuthor.java，主要用来存储用户的权限信息。用户权限类的 UML 图如图 14.19 所示。

因为 UsersAuthor 类依赖于 User 类，所以这里给出用户类和用户权限类之间的关联图，如图 14.20 所示。

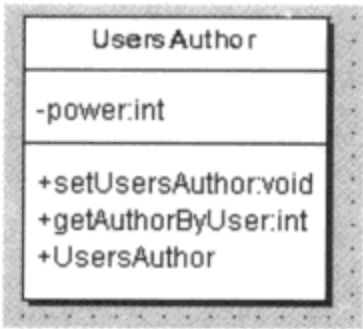


图 14.19 用户权限类的 UML 图

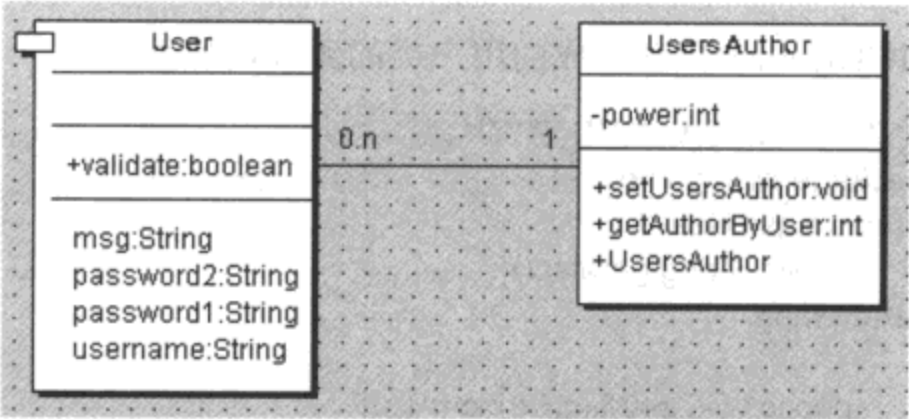


图 14.20 用户类和用户权限类之间的关联图

由 Together 自动生成的用户权限类的源代码如下：

```
//***** UsersAuthor.java*****
package com.gd.vo;
public class UsersAuthor {
    public void setUsersAuthor(User lnkUser, int power) {
        lnkUser = lnkUser;
        power = power;
    }
    public int getAuthorByUser(User lnkUser) {
        return power;
    }
    public UsersAuthor(User lnkUser, int power) {
        lnkUser = lnkUser;
        power = power;
    }
    private int power;
    /**
     * @clientCardinality 1
     * @supplierCardinality 0..n
     */
}
```

```
private User InkUser;  
}
```

代码说明：power，表示用户拥有的权限，如果 power 为 0 表示普通用户，如果 power 为 1 表示超级用户。

(3) 负责新闻类别的类 NewsType.java，主要用来存储新闻类别的信息。新闻类别类的 UML 图如图 14.21 所示。

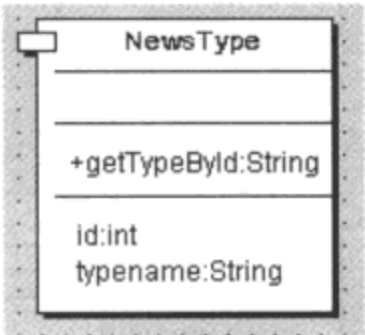


图 14.21 新闻类别类的 UML 图

由 Together 自动生成的新闻类别类的源代码如下：

```
/******* NewsType.java*****  
  
package com.gd.vo;  
public class NewsType {  
    public int getId(){  
        return id;  
    }  
    public void setId(int id){  
        this.id = id;  
    }  
    public String getTypename(){  
        return typename;  
    }  
    public void setTypename(String typename){  
        this.typename = typename;  
    }  
    public String getTypeById (int id) {  
        return typename;  
    }  
    private int id;  
    private String typename;  
}
```

(4) 负责新闻的类 News.java，主要用来存储新闻的信息。新闻类的 UML 图如图 14.22 所示。

因为 News 类依赖于 User 类和新闻类别类，所以这里给出用户类、新闻类和新闻类别类之间的关联图，如图 14.23 所示。

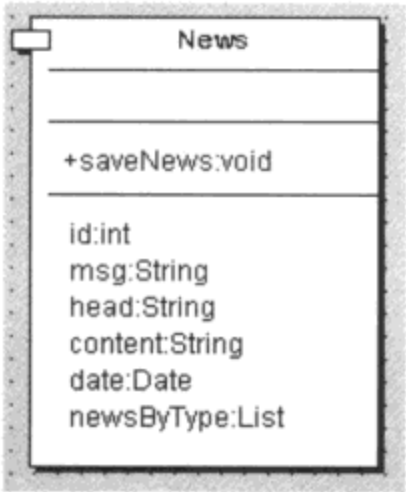


图 14.22 新闻类的 UML 图

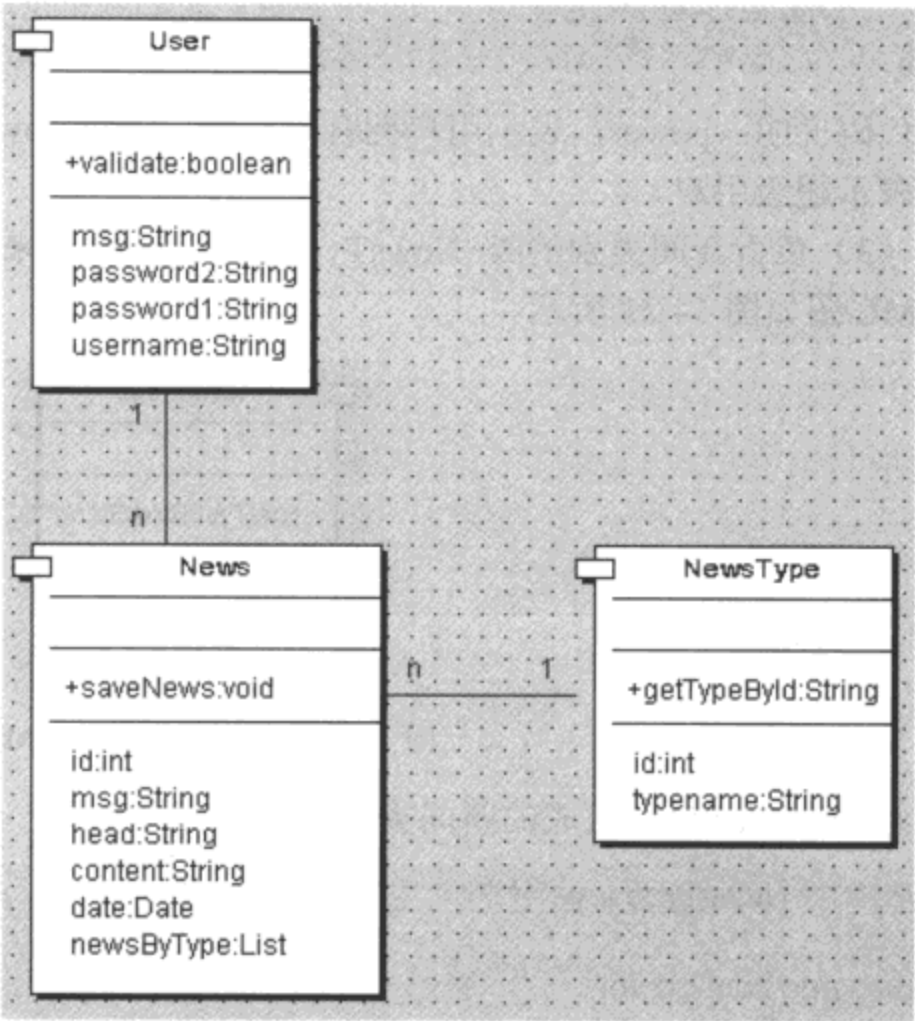


图 14.23 用户类、新闻类和新闻类别类之间的关联图

由 Together 自动生成的新闻类的源代码如下：

```
//***** News.java *****

package com.gd.vo;
public class News {
public int getId(){
    return id;
}
public void setId(int id){
    this.id = id;
}
public String getMsg(){
    return msg;
}
public void setMsg(String msg){
    this.msg = msg;
}
public String getHead(){
    return head;
}
public void setHead(String head){
    this.head = head;
}
public String getContent(){
```

```

        return content;
    }
    public void setContent(String content){
        this.content = content;
    }
    public Date getDate(){
        return date;
    }
    public void setDate(Date date){
        this.date = date;
    }
    //根据新闻类别的 id 获取新闻
    public List getNewsByType(int id) {
    }
    //保存新闻
    public void saveNews(New new) {
    }
    private int id;
    private String msg;
    private String head;
    private String content;
    private Date date;
    /**
     * @clientCardinality n
     * @supplierCardinality 1
     */
    private NewsType InkNewsType;
    /**
     * @clientCardinality n
     * @supplierCardinality 1
     */
    private User InkUser;
}

```

14.4.3 设计数据库

经过分析可以知道,在该应用中,主要有用户信息、用户的授权信息、新闻类别和新闻等内容需要存储,所以对数据库表的设计也主要从这几个方面来考虑。

(1) 存储用户信息的表,表名为 user,主要字段有 username 和 password,主键为 username。

(2) 存储用户授权信息的表,表名为 userAuthor,主要字段有 username 和 power,主键为 username 和 power。

(3) 存储新闻类别的表,表名为 newsType,主要字段有 id 和 type,主键为 id。

(4) 存储新闻的表,表名为 news,主要字段有 id、head、content、issuedate、issueuser 和 newstype,主键为 id。

注意：上面设计的数据库表是为了演示使用。如果是在实际的应用中，笔者建议每个表都加上 id，然后表之间的关联用 id 来实现。这样就把表之间的管理和业务逻辑的变化分离开了，降低了业务逻辑和表之间的耦合性。

14.4.4 新闻发布系统在持久层的整体 UML 图

经过上面对页面、持久类和数据库的设计，最终可以得到一个新闻发布系统在持久层的整体 UML 图，如图 14.24 所示。

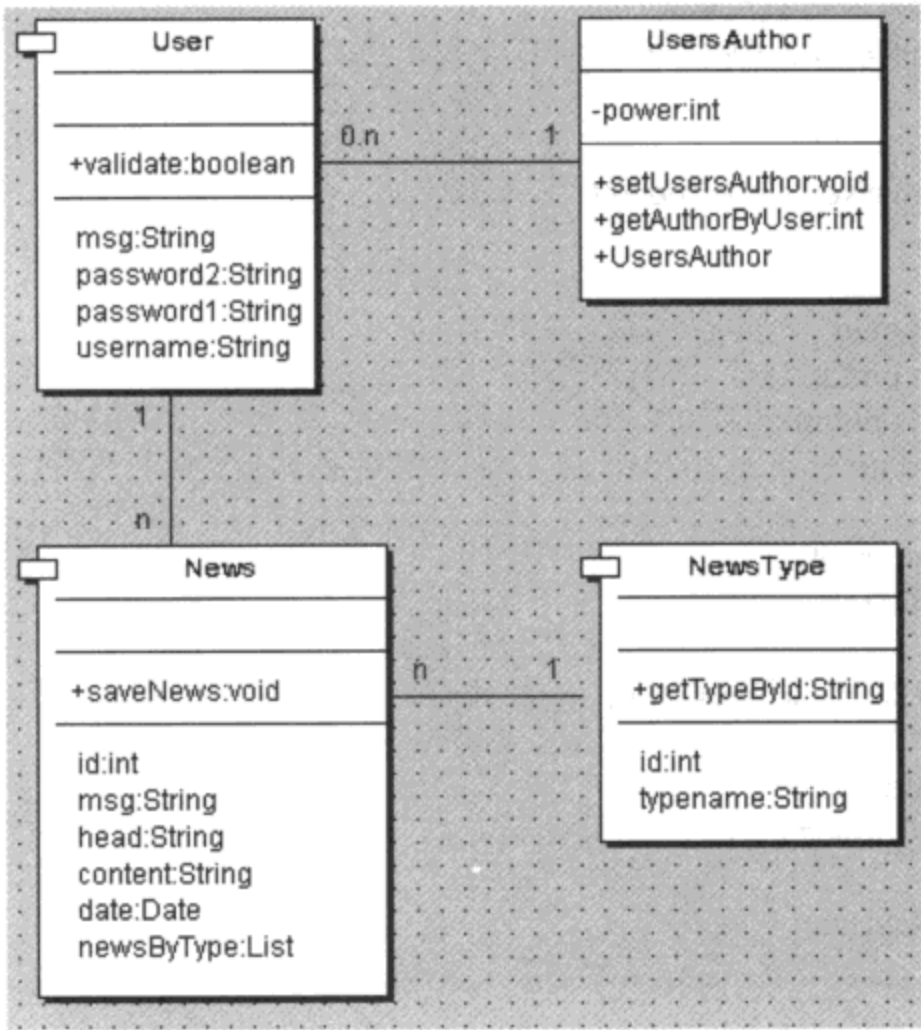


图 14.24 新闻发布系统在持久层的整体 UML 图

14.5 编写新闻发布系统的 JSP 页面

根据上面设计的新闻发布系统，编写其 JSP 页面。

14.5.1 新闻发布的展示页面 show.jsp

该页面存放在 WEB-INF/jsp 下，用来展示已经发布的新闻，并按照新闻类别进行显示。这里将新闻标题放在 Map 数组中，建立新闻类别 id 和新闻标题之间的对应关系。首先将新闻类别通过循环展示出来，同时每显示一个新闻类别，将新闻类别对应的新闻标题再通过

循环显示出来。show.jsp 的代码如下：

```
<%@page contentType="text/html;charset=GBK"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@page import="java.util.*,com.gd.util.*,com.gd.vo.*,com.gd.po.Newstype,com.gd.po.New"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>新闻发布的展示页面</title>

<style type="text/css">
<!--
.style1 {font-family: "隶书"}
-->
</style>
</head>
<%
List listNewsType = (List)request.getAttribute("listNewsType");
Map mapNews = (Map)request.getAttribute("mapNews");
%>
<body>
<table width="100%" height="100%" border="1" cellpadding="0" cellspacing="0" >
<%
    for (int i = 0; listNewsType != null && i < listNewsType.size(); i++) {
%>
<tr height="100%">
<td height="20"><strong><%=((Newstype)listNewsType.get(i)).getType()%></strong></td>
<td><strong><%=((Newstype)listNewsType.get(i + 1)).getType()%></strong></td>
</tr>
<tr height="100%">
<td height="150"><ol>
<%
        List newsHeads = (List)mapNews.get((((Newstype)listNewsType.get(i)).getId()));
        for (int j = 0; newsHeads != null && j < newsHeads.size(); j++) {
%>
<li><%=((New)newsHeads.get(j)).getContent() %></li>
<%}%>
</ol></td>
<td><ol>
<%
        newsHeads = (List)mapNews.get((((Newstype)listNewsType.get(++i)).getId()));
        for (int j = 0; newsHeads != null && j < newsHeads.size(); j++) {
%>
<li><%=((New)newsHeads.get(j)).getContent() %></li>
<%}%>
</ol></td>
</tr>
<tr height="100%" style="border-top-width:1">
<td height="15" style="border-top-width:1"><div align="right" class="style1">》更多内容
```



```

</div></td>
    <td style="border-top-width:1"><div align="right" class="style1">》更多内容</div></td>
</tr>
<%}%>
</table>
</body>
</html>

```

代码说明：Map mapNews = (Map)request.getAttribute("mapNews")，这里通过 Map 来循环把每种新闻类别的内容显示出来。

14.5.2 发布新闻页面 release.jsp

该页面存放在 WEB-INF/jsp 下，主要通过表单来提交用户填写的新闻标题和内容及发布人，这里添加了一个辅助类 NewsUtil.java，主要用来获取当前日期。release.jsp 的代码如下：

```

<%@page contentType="text/html; charset=GBK"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@page import="java.util.List,com.gd.util.*,com.gd.vo.User,com.gd.po.Newstype"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>发布新闻页面</title>
<style type="text/css">
<!--
.style1 {
    font-size: large;
    font-weight: bold;
}
-->
</style>
</head>
<%
List newsTypes = (List)request.getAttribute("newsTypes");
User user = (User)request.getAttribute("user");
%>
<body>
<form name="form1" method="post" action="/myNews/release.do">
    <table width="100%" height="160" border="1" cellpadding="0" cellspacing="0">
        <tr>
            <td height="17" colspan="2"><div align="center" class="style1">发布新闻</div></td>
        </tr>
        <tr>
            <td width="126" height="19"><strong>新闻标题</strong></td>
            <td width="560"><input name="head" type="text" size="100%"></td>
        </tr>
    </table>

```

```

<td height="73"><strong>新闻内容</strong></td>
<td><p>
  <textarea name="content" cols="100%" rows="30"></textarea>
</p>
</td>
</tr>
<tr>
  <td height="19" colspan="2"><strong>发布时间: </strong><%=NewsUtil.getCurrentDate()%>
<strong>发布人</strong>: <%=user.getUsername()%> <strong>新闻类别</strong>:
  <select name="newsType">
    <%
      for (int i = 0; newsTypes != null && i < newsTypes.size(); i++) {
        Newstype newsType = (Newstype)newsTypes.get(i);
      %>
    <option value = '<%=newsType.getId()%>'><%=newsType.getType()%></option>
    <%=}%>
  </select></td>
</tr>
<tr>
  <td height="18">&nbsp;</td>
  <td><input type="submit" name="insert" value="提交">
  <input type="submit" name="update" value="修改">
  <input type="submit" name="delete" value="删除"></td>
</tr>
</table>
</form>
</body>
</html>

```

代码说明: NewsUtil.getCurrentDate(), 这里添加一个辅助类 NewsUtil, 用来负责 myNews 系统的一些辅助方法, 该类在包 com.gd.util 里。

14.5.3 用户注册页面 regedit.jsp

该页面存放在 WEB-INF/jsp 下, 主要通过表单来提交用户填写的用户名和密码。regedit.jsp 的示例代码如下:

```

<%@page contentType="text/html; charset=GBK"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>用户注册页面</title>
</head>

<body>
<form name="form1" method="post" action="/myNews/regedit.do">

```

```
<table width="100%" height="251" border="1" cellpadding="0" cellspacing="0">
  <tr>
    <td height="17" colspan="2"><div align="center"><strong>注册用户</strong></div></td>
  </tr>
  <tr>
    <td width="18%"><strong>用户名: </strong></td>
    <td width="82%"><input type="text" name="username"></td>
  </tr>
  <tr>
    <td><strong>密码: </strong></td>
    <td><input type="password" name="password1"></td>
  </tr>
  <tr>
    <td><strong>确认密码: </strong></td>
    <td><input type="password" name="password2"></td>
  </tr>
  <tr>
    <td colspan="2"><div align="center">
      <input type="submit" name="Submit" value="注册">
      <input type="reset" name="Submit" value="重置">
    </div></td>
  </tr>
</table>
</form>
</body>
</html>
```

14.5.4 管理员登录页面 login.jsp

该页面存放在 WEB-INF/jsp 下，主要通过表单来提交用户填写的用户名和密码，用于验证用户是否填写正确。login.jsp 的示例代码如下：

```
<%@page contentType="text/html; charset=GBK"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>管理员登录页面</title>
</head>

<body>
<form name="form1" method="post" action="/myNews/login.do">
  <table width="100%" height="251" border="1" cellpadding="0" cellspacing="0">
    <tr>
      <td height="17" colspan="2"><div align="center"><strong>管理员登录</strong></div></td>
    </tr>
    <tr>
```



```
<td width="18%"><strong>用户名: </strong></td>
<td width="82%"><input type="text" name="username"></td>
</tr>
<tr>
<td><strong>密码: </strong></td>
<td><input type="password" name="password1"></td>
</tr>
<tr>
<td><strong>确认密码: </strong></td>
<td><input type="password" name="password2"></td>
</tr>
<tr>
<td colspan="2"><div align="center">
<input type="submit" name="Submit" value="登录">
<input type="reset" name="Submit" value="重置">
</div></td>
</tr>
</table>
</form>
</body>
</html>
```

14.5.5 错误处理页面 error.jsp

该页面存放在 WEB-INF/jsp 下, 主要用来捕捉并显示程序出现的 Exception 信息。error.jsp 的示例代码如下:

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>错误处理页面</title>
<style type="text/css">
<!--
.style1 {
    color: #000000;
    font-weight: bold;
}
.style2 {color: #FF0000}
-->
</style>
</head>
<% Exception ex = (Exception)request.getAttribute("Exception"); %>

<body>
<table width="100%" border="1">
<tr>
<td colspan="2"><div align="center"><strong>错误信息显示</strong></div></td>
</tr>
```



```
<tr>
  <td width="22%" height="141"><span class="style1">错误信息是: </span></td>
  <td width="78%"><span class="style2"><%=ex.getMessage();%></span></td>
</tr>
</table>
</body>
</html>
```

如果要使用该页面，则需要在 Spring 的配置文档中增加以下代码：

```
<bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMapping-
ExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="java.sql.SQLException">error</prop>
      <prop key="java.sql.IOException">error</prop>
    </props>
  </property>
</bean>
```

代码说明：只要发生了 SQLException 异常或 IOException 异常，就会连接至 /WEB-INF/jsp/error.jsp。

14.6 建立数据库表并生成 XML 和 POJO

这里仍然通过 XAMPP 来建立 MySQL 数据库，步骤如下：

(1) 在浏览器地址栏中输入“http://localhost/xampp/”，进入 XAMPP 的管理画面，如图 14.25 所示。

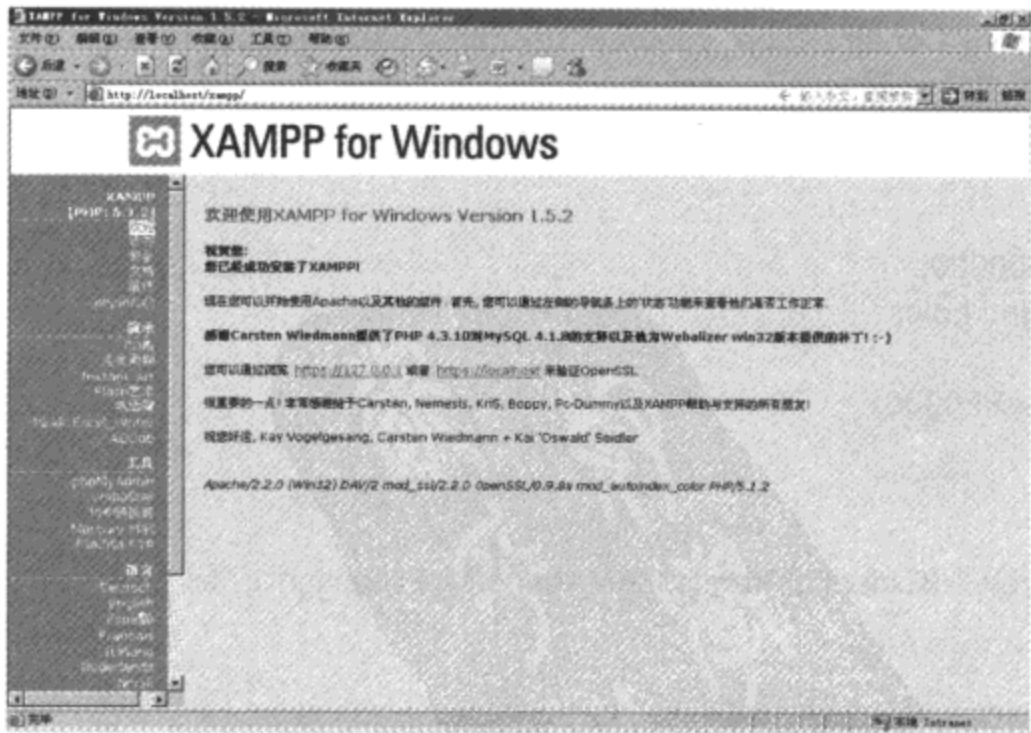


图 14.25 XAMPP 的管理画面

(2) 在 XAMPP 画面左边单击工具下的 phpMyAdmin，将会出现创建数据库的向导画面，如图 14.26 所示。

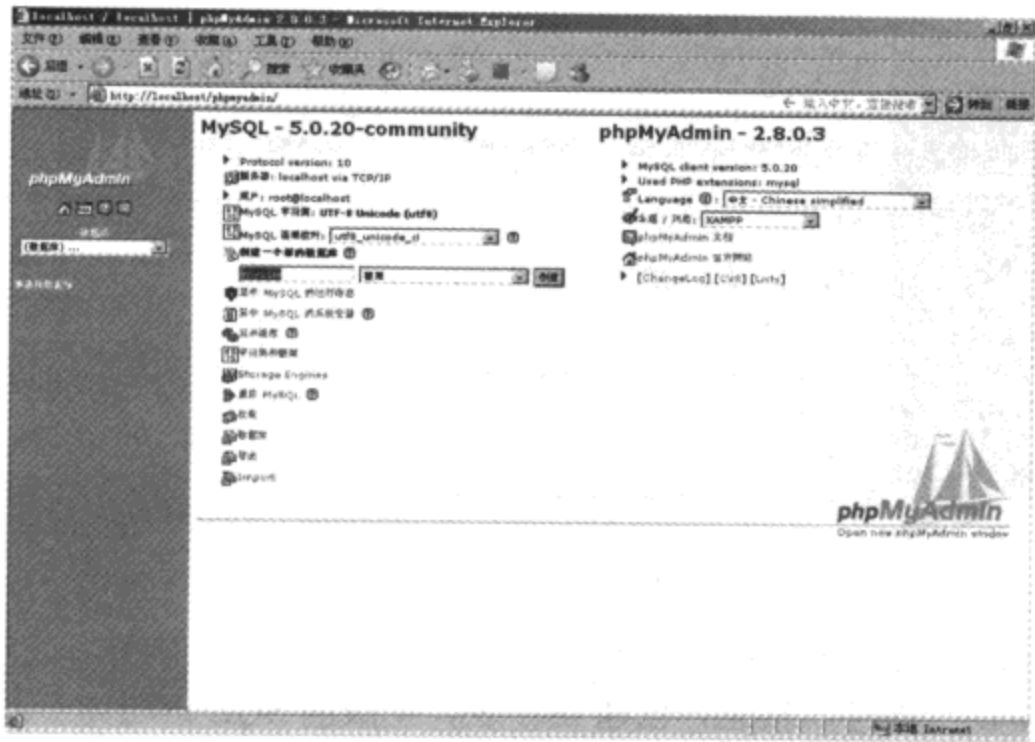


图 14.26 创建数据库的向导画面

(3) 这里设定创建的数据库名为 myNews，然后单击“创建”按钮，即可创建数据库 myNews。

接下来根据向导，分别创建前面设计的数据库表。

14.6.1 存放用户信息的数据库表

(1) 创建一个存储用户信息的表，表名为 user，主要包括 2 个字段，创建 user 表的向导如图 14.27 所示。

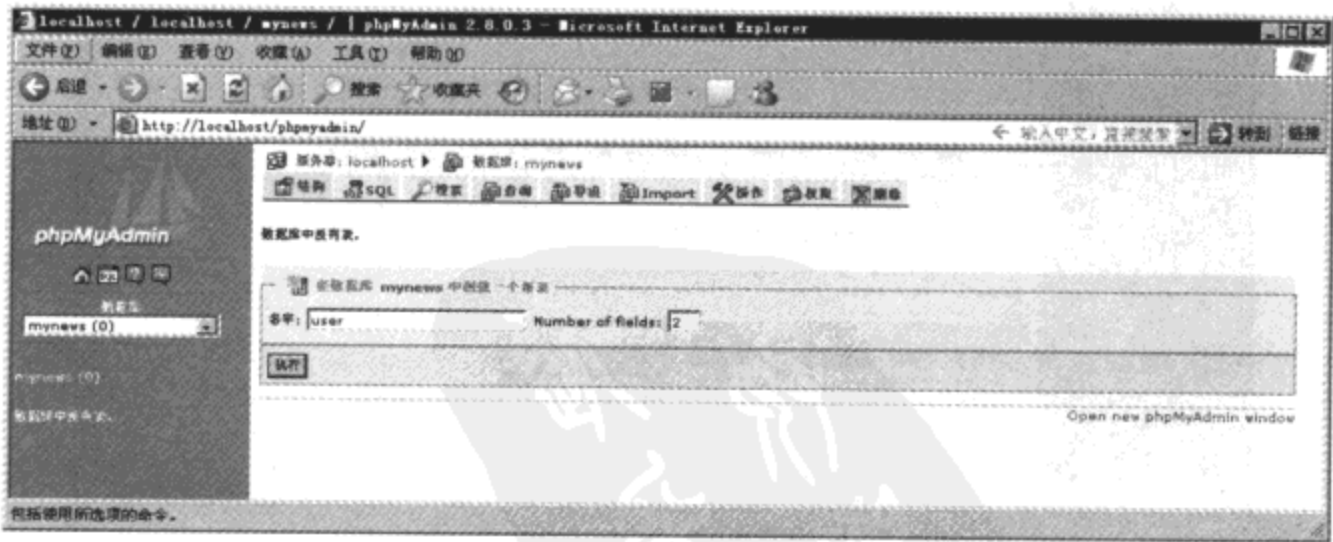


图 14.27 创建 user 表的向导

(2) 单击创建 user 表的向导画面中的“执行”按钮，出现设定 user 表字段的画面，如图 14.28 所示。

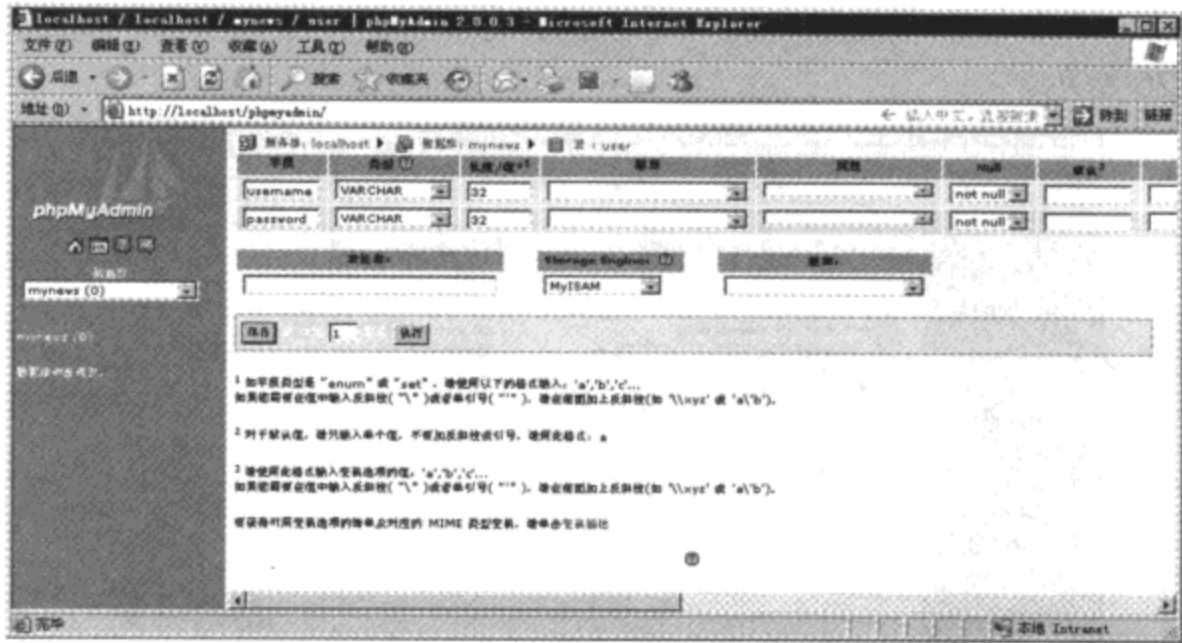


图 14.28 设定 user 表字段的画面

(3) 输入字段名称：username、password，都为 varchar 类型，长度为 32，主键为 username，然后单击“保存”按钮，创建 user 表，生成的 SQL 语句如下：

```
CREATE TABLE `user` (
  `username` VARCHAR( 32 ) NOT NULL ,
  `password` VARCHAR( 32 ) NOT NULL ,
  PRIMARY KEY ( `username` )
) ENGINE = MYISAM ;
```

(4) 最终的 user 表结构如图 14.29 所示。

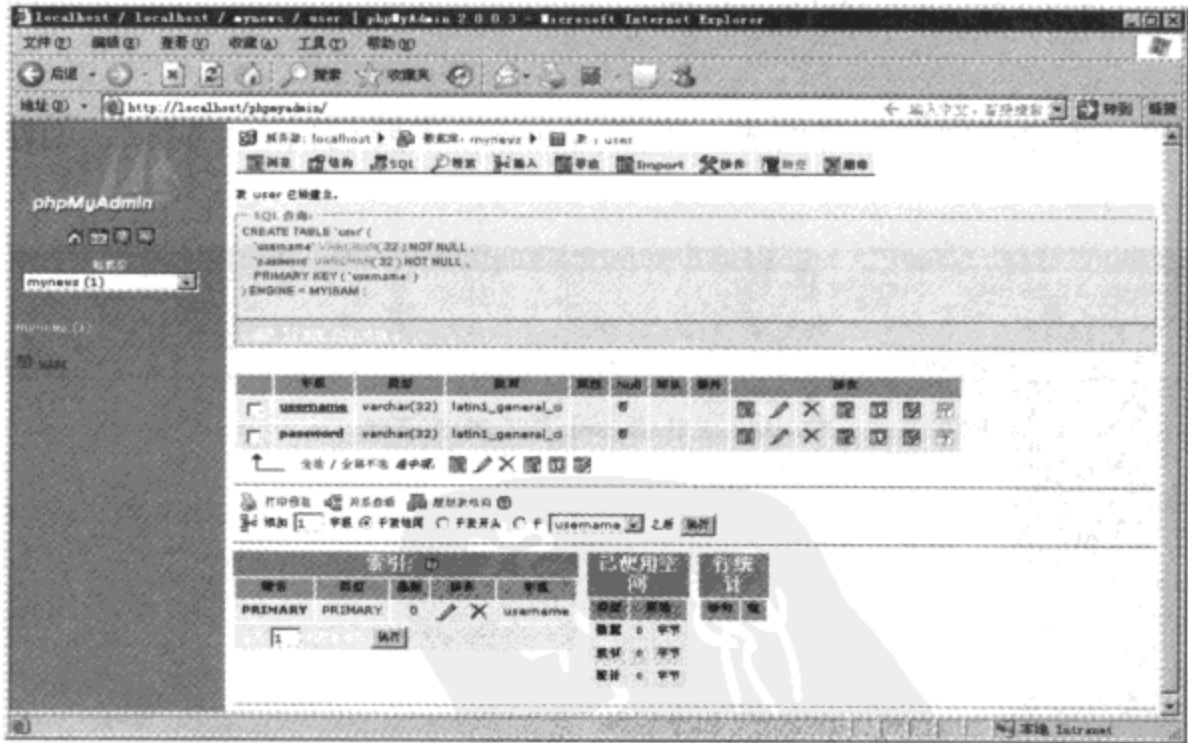


图 14.29 user 表结构

14.6.2 存放新闻的数据库表

(1) 创建一个存储新闻的表，表名为 news，主要包括 6 个字段，创建 news 表的向导

如图 14.30 所示。

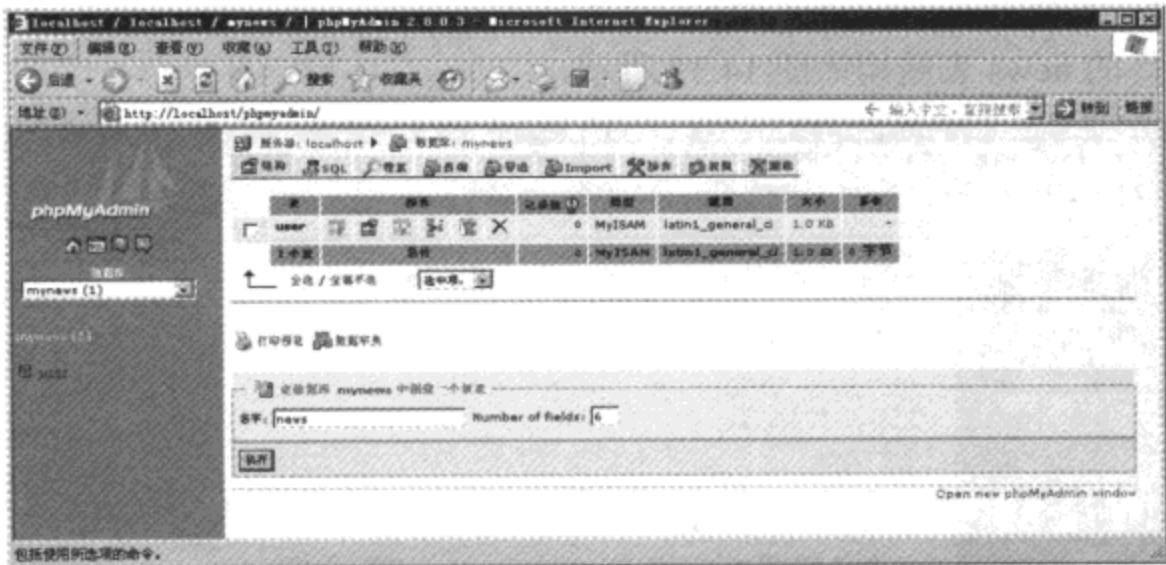


图 14.30 创建 news 表的向导

(2) 单击创建 news 表的向导画面中的“执行”按钮，出现设定 news 表字段的画面，如图 14.31 所示。

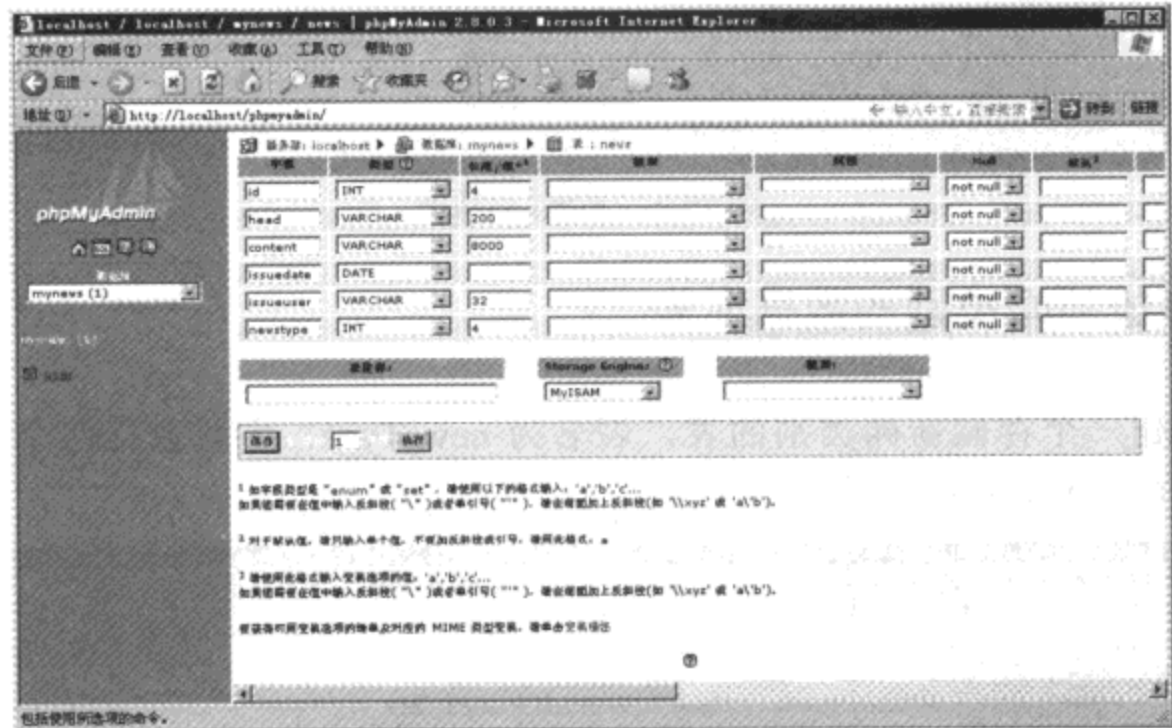


图 14.31 设定 news 表字段的画面

(3) 输入字段名称：id、head、content、issuedate、issueuser 和 newstype，id 为 int 类型，长度为 4；head 为 varchar 类型，长度为 200；content 为 varchar 类型，长度为 8000；issuedate 为 Date 类型，长度为 10；issueuser 为 varchar 类型，长度 32；newstype 为 int 类型，长度为 4，主键为 id，然后单击“保存”按钮，创建 news 表。生成的 SQL 语句如下：

```
CREATE TABLE `news` (  
  `id` INT(4) NOT NULL,  
  `head` VARCHAR(200) NOT NULL,  
  `content` VARCHAR(8000) NOT NULL,  
  `issuedate` DATE NOT NULL,  
  `issueuser` VARCHAR(32) NOT NULL,  
  `newstype` INT(4) NOT NULL,
```


PRIMARY KEY ('id')
) ENGINE = MYISAM ;

(4) 最终的 news 表结构如图 14.32 所示。

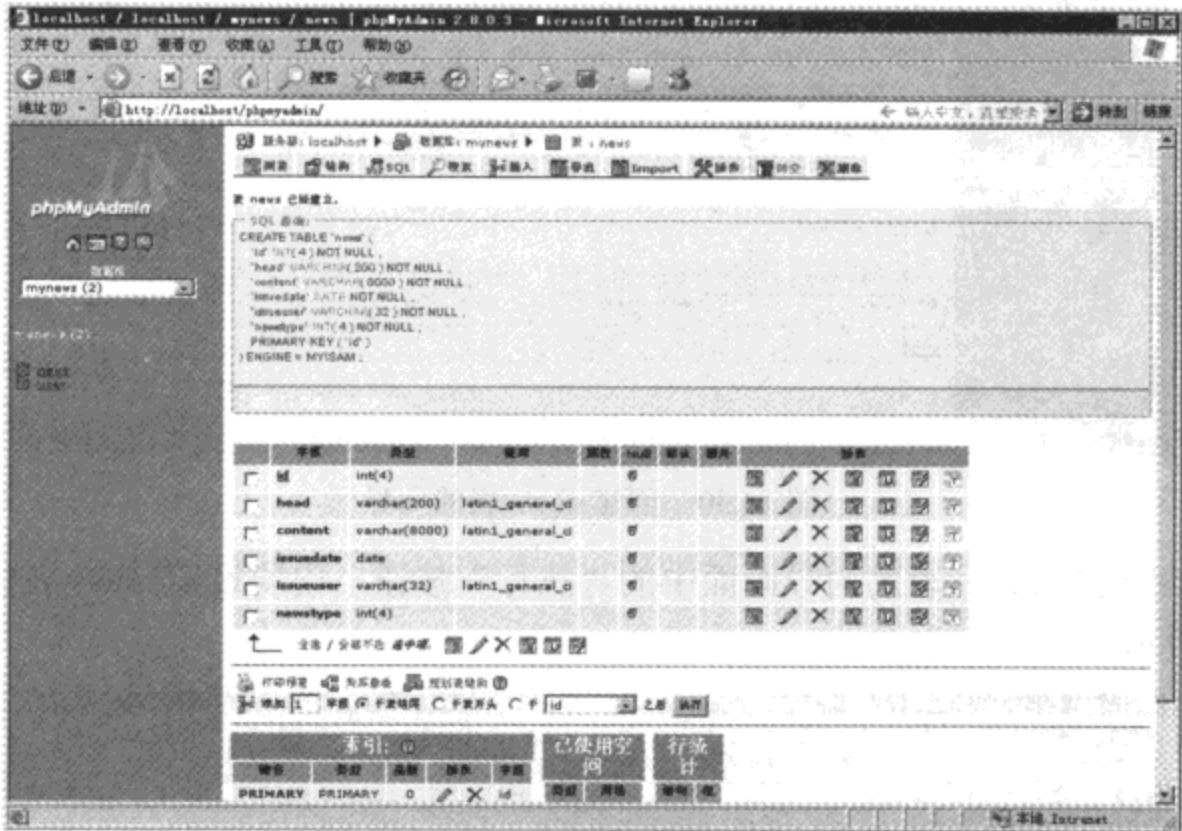


图 14.32 news 表结构

14.6.3 存放新闻类别的数据库表

(1) 创建一个存储新闻类别的表，表名为 newsType，主要包括两个字段。创建 newsType 表的向导如图 14.33 所示。



图 14.33 创建 newsType 表的向导

(2) 单击创建 newsType 表的向导画面中的“执行”按钮，出现设定 newsType 表字段的画面，如图 14.34 所示。



图 14.34 设定 newsType 表字段的画面

(3) 输入字段名称: id、type, id 为 int 类型, 长度为 4; type 为 varchar 类型, 长度为 32, 主键为 id, , 然后单击“保存”按钮, 创建 newsType 表。生成的 SQL 语句如下:

```
CREATE TABLE `newsType` (  
  `id` INT(4) NOT NULL ,  
  `type` VARCHAR(32) NOT NULL ,  
  PRIMARY KEY (`id` )  
) ENGINE = MYISAM ;
```

(4) 最终的 newsType 表结构如图 14.35 所示。

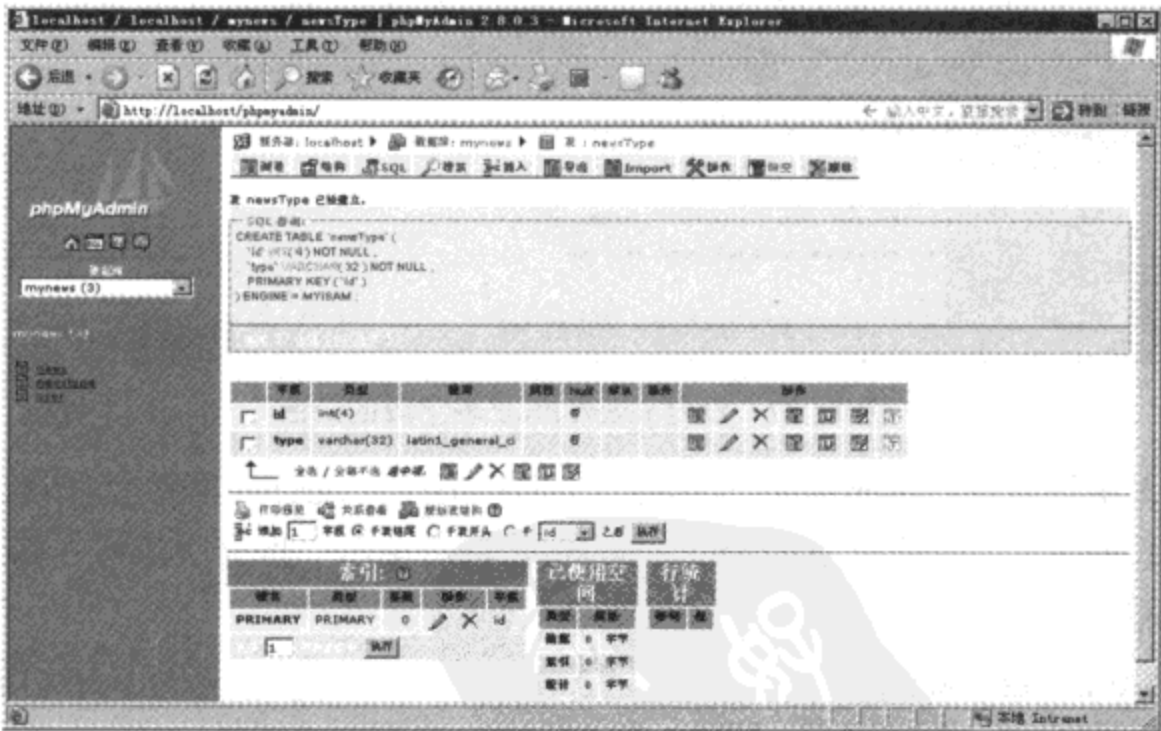


图 14.35 newsType 表结构

14.6.4 存放用户权限的数据库表

(1) 创建一个存储用户授权信息的表, 表名为 userAuthor, 主要包括两个字段。创建

userAuthor 表的向导如图 14.36 所示。

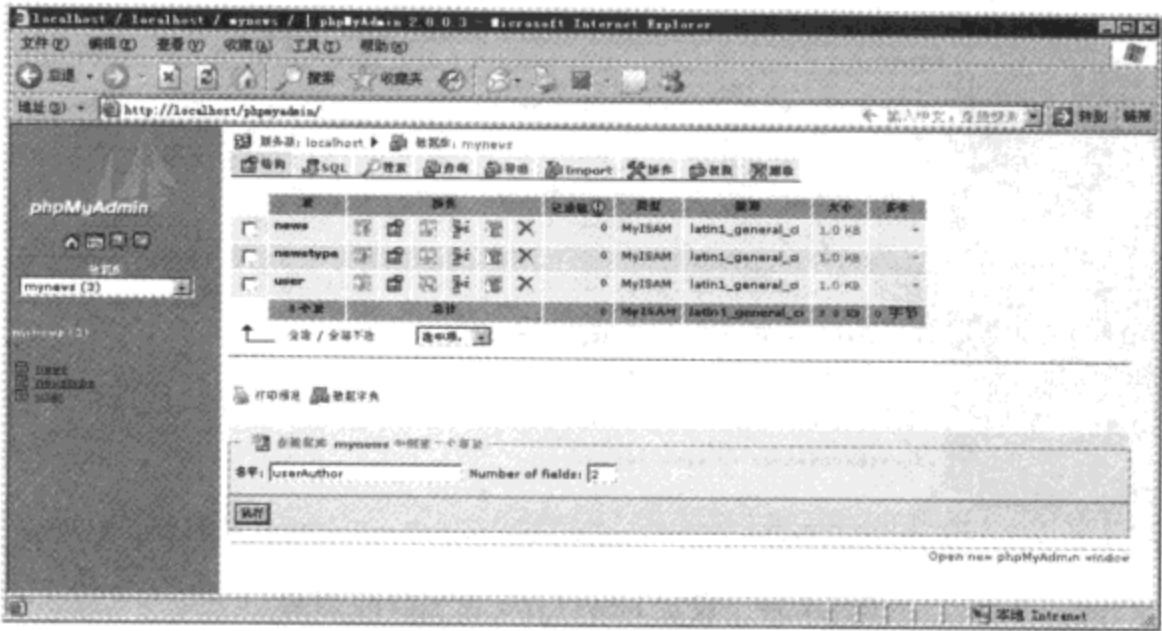


图 14.36 创建 userAuthor 表的向导

(2) 单击创建 userAuthor 表的向导画面中的“执行”按钮，出现设定 userAuthor 表字段的画面，如图 14.37 所示。

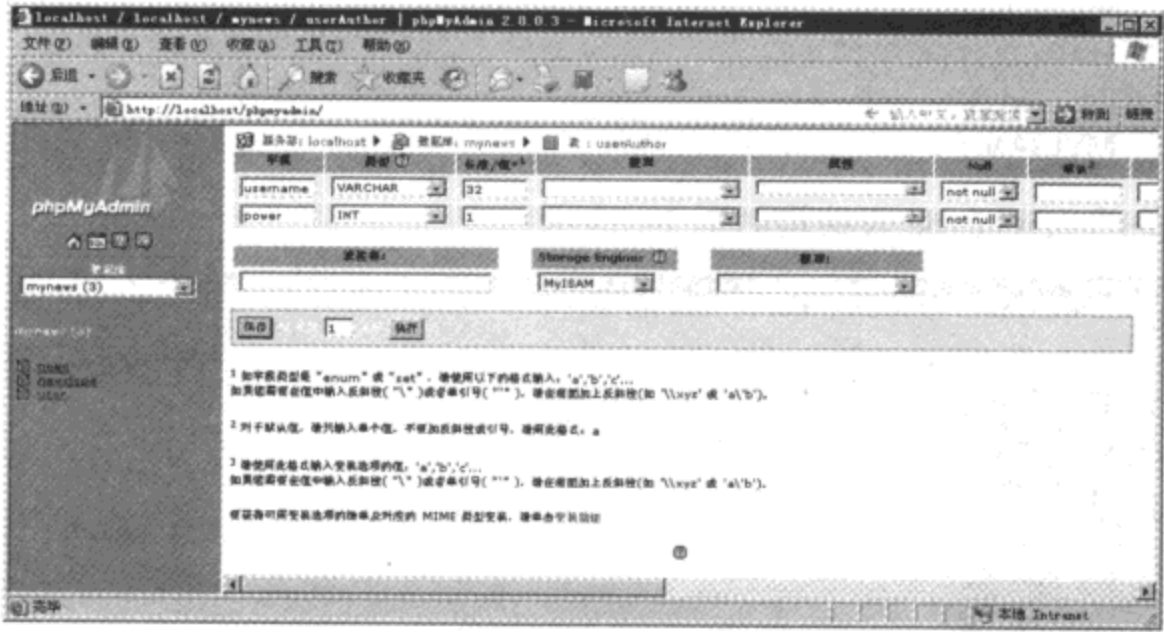


图 14.37 设定 userAuthor 表字段的画面

(3) 输入字段名称：username、power，username 为 varchar 类型，长度为 32；power 为 int 类型，长度为 1，主键为 username、power，然后单击“保存”按钮，创建 userAuthor 表。生成的 SQL 语句如下：

```
CREATE TABLE `userAuthor` (
  `username` VARCHAR( 32 ) NOT NULL ,
  `power` INT( 1 ) NOT NULL ,
  PRIMARY KEY ( `username` , `power` )
) ENGINE = MYISAM ;
```

(4) 最终的 userAuthor 表结构如图 14.38 所示。

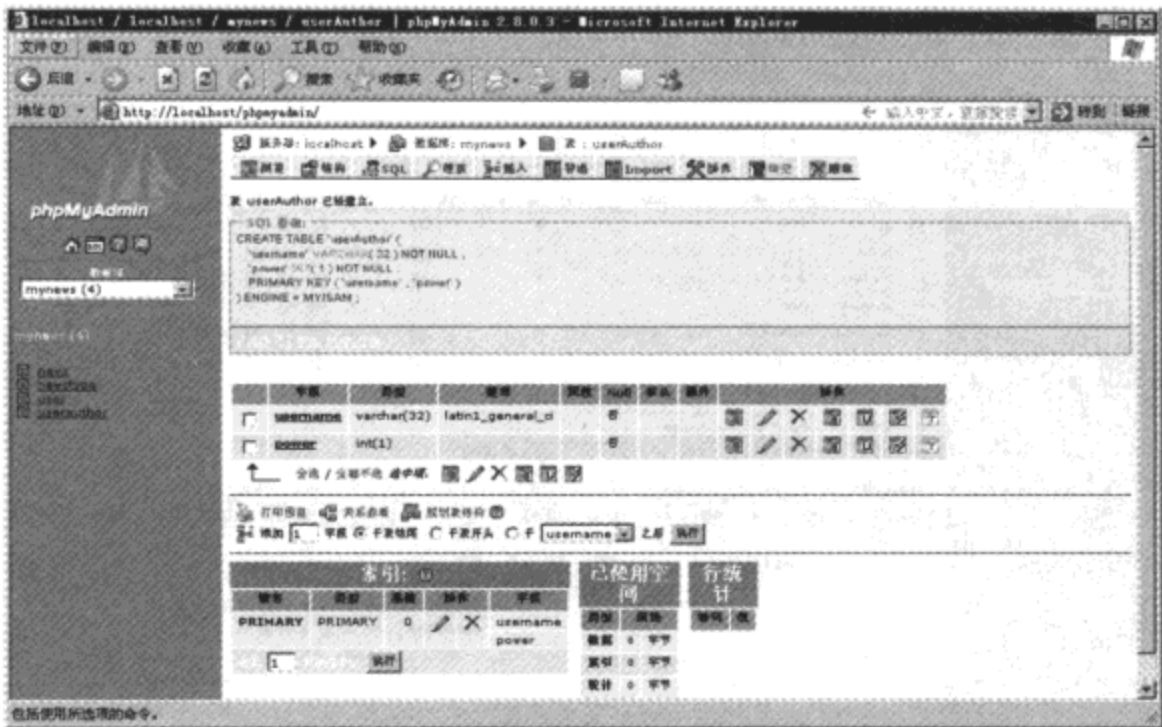


图 14.38 userAuthor 表结构

14.6.5 建立表之间的关系

在这 4 个表中，其中 news 表中的 issueuser 字段和 user 表中的 username 字段相关联；news 表中的 newstype 字段和 newsType 表中的 id 字段相关联；userAuthor 表中的 username 字段和 user 表中的 username 字段相关联。具体设置方法如下：

(1) 单击 phpMyAdmin 画面上方的数据库 myNews，进入 myNews 数据库的整体结构图，如图 14.39 所示。



图 14.39 myNews 数据库的整体结构图

(2) 单击画面左边的 news 表，进入 news 表的整体结构图，如图 14.40 所示。

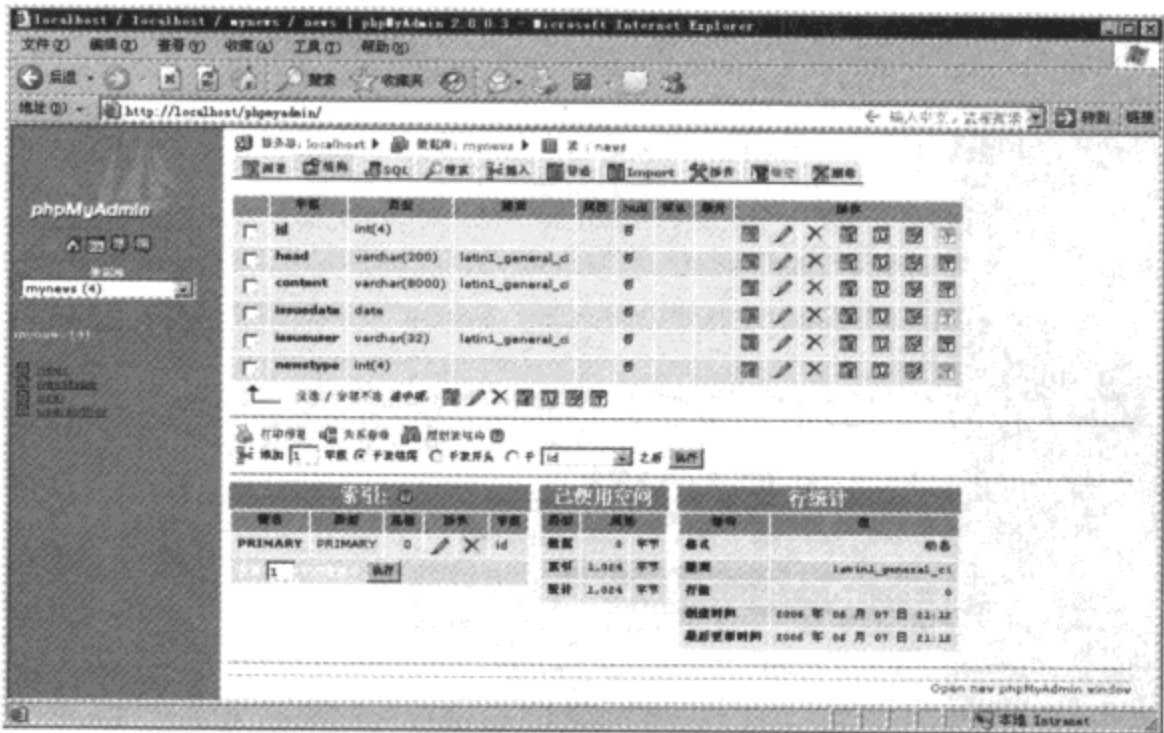


图 14.40 news 表的整体结构图

(3) 单击画面中间部分的关系查看，进入 news 表和其他表进行关系设定的画面，如图 14.41 所示。

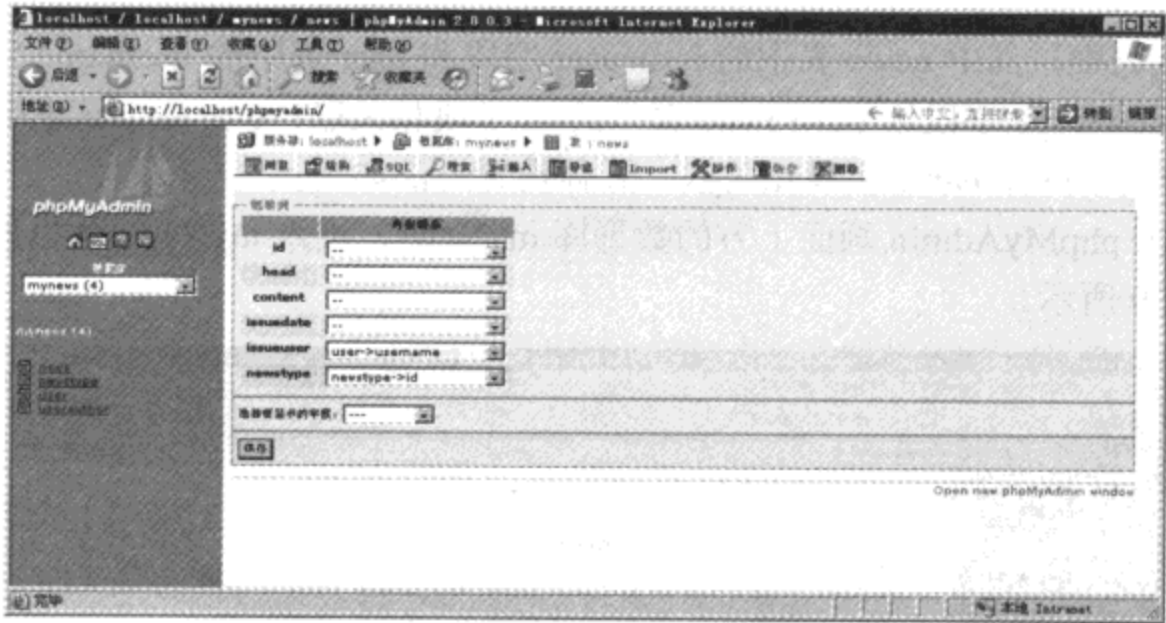


图 14.41 news 表和其他表进行关系设定的画面

(4) 选择 issueuser 与 user 表的 username 相关联，newstype 与 newsType 表的 id 相关联，然后单击“保存”按钮，即可建立关系。

(5) 再设定 userAuthor 表的关系，单击画面左边的 userAuthor 表，进入 userAuthor 表的整体结构图，如图 14.42 所示。

(6) 单击画面中间部分的关系查看，进入 userAuthor 表和其他表进行关系设定的画面，如图 14.43 所示。

(7) 选择 username 与 user 表的 username 相关联，然后单击“保存”按钮，即可建立关系。

(8) 最终 myNews 数据库的数据字典，如图 14.44 所示。

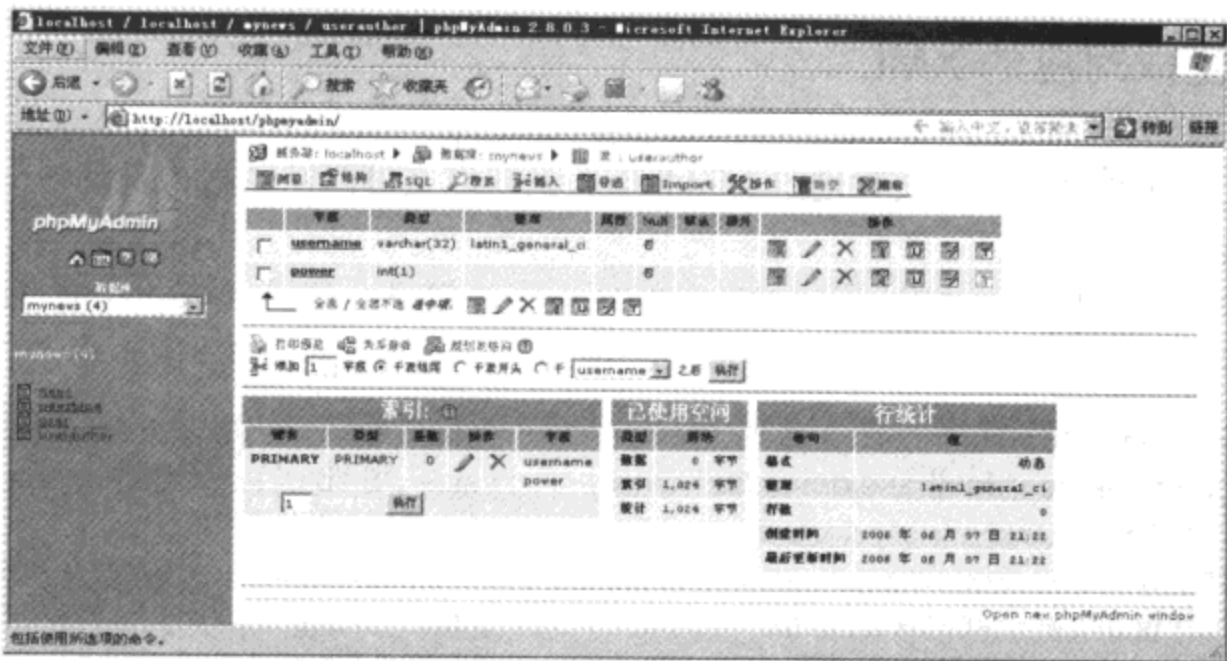


图 14.42 userAuthor 表的整体结构图

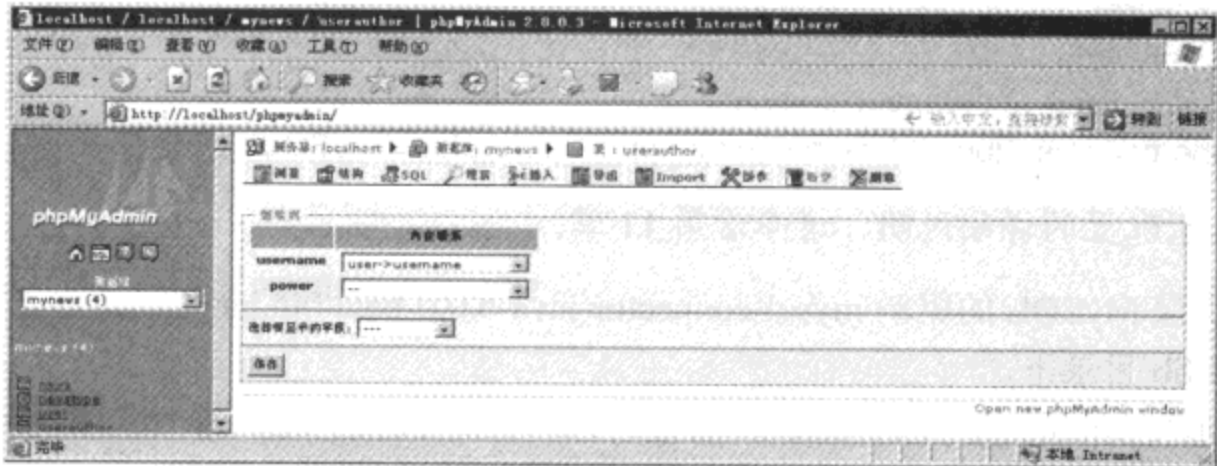


图 14.43 userAuthor 表和其他表进行关系设定的画面

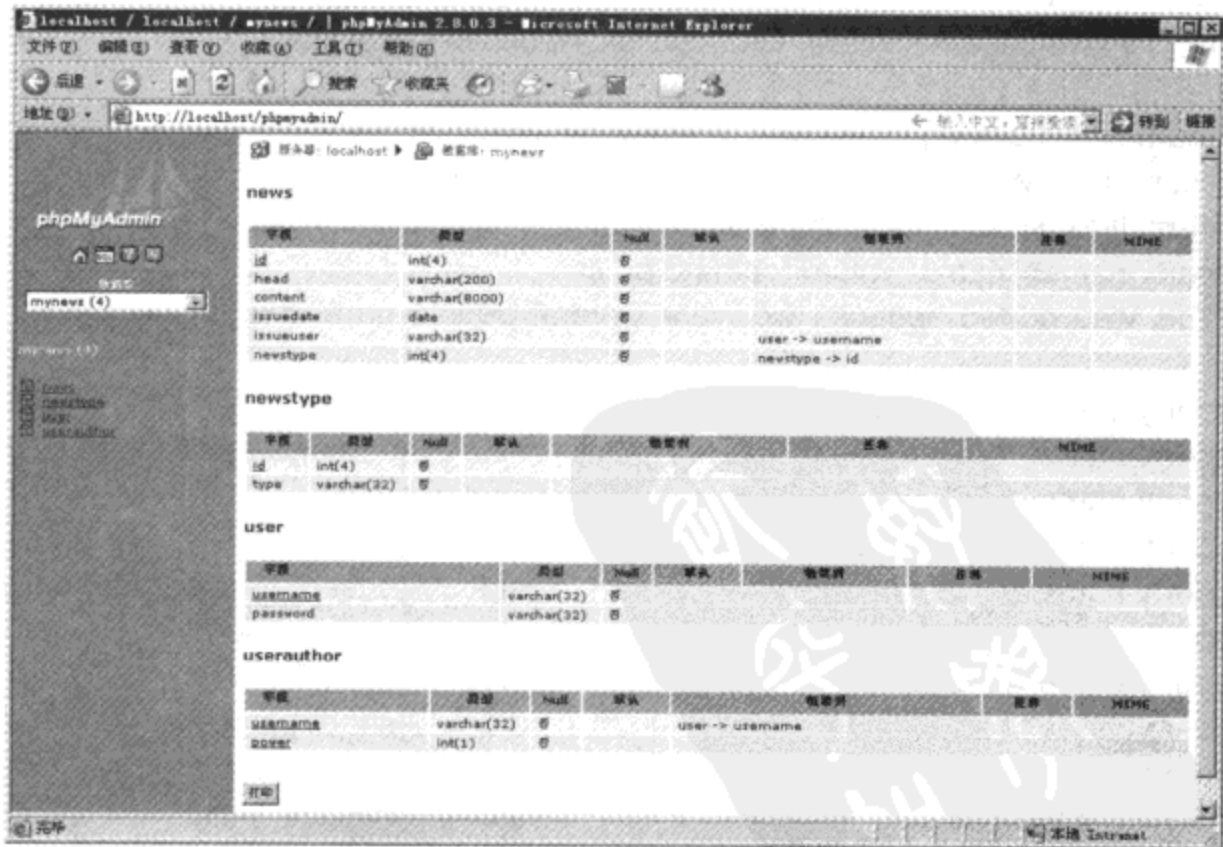


图 14.44 myNews 数据库的数据字典

14.6.6 生成对应的 XML

这里采用从数据库定义文件通过 Middlegen 到映射文件，从映射文件通过 hbm2java 到 POJO 的方法。

(1) 进入 Middlegen-Hibernate-r5/config/database 目录，修改 mysql.xml。mysql.xml 示例代码如下：

```
<property name="database.script.file" value="${src.dir}/sql/${name}-mysql.sql"/>
  <property name="database.driver.file" value="${lib.dir}/mysql-connector-java-5.0.0-
beta-bin.jar"/>
  <property name="database.driver.classpath" value="${database.driver.file}"/>
  <property name="database.driver" value="org.gjt.mm.mysql.Driver"/>
  <property name="database.url" value="jdbc:mysql://localhost/myNews"/>
  <property name="database.userid" value="root"/>
  <property name="database.password" value="root"/>
  <property name="database.schema" value=""/>
  <property name="database.catalog" value=""/>
  <property name="jboss.datasource.mapping" value="mySQL"/>
```

说明：关于配置的详细说明，请参看第 11 章。

(2) 确保 MySQL 的驱动 mysql-connector-java-5.0.0-beta-bin.jar 已经放在了 Middlegen-Hibernate-r5/lib 目录下。

(3) 进入 Middlegen-Hibernate-r5 的根目录，打开 build.xml，修改其中的代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE project [
  <!ENTITY database SYSTEM "file:./config/database/hsqldb.xml">
]>
<project name="Middlegen Hibernate" default="all" basedir=".">
<!-- project name="Middlegen Hibernate" default="all" basedir="." -->
  <property file="${basedir}/build.properties"/>
  <property name="name" value="com.gd.po"/>
  <!-- This was added because we were several people (in a course) deploying to same app
server>
  <property environment="env"/>
  <property name="unique.name" value="${name}.${env.COMPUTERNAME}"/-->
  <property name="gui" value="true"/>
  <property name="unique.name" value="${name}"/>
  <property name="appxml.src.file" value="${basedir}/src/application.xml"/>
  <property name="lib.dir" value="${basedir}/lib"/>
  <property name="src.dir" value="${basedir}/src"/>
  <property name="java.src.dir" value="${src.dir}/java"/>
  <property name="web.src.dir" value="${src.dir}/web"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.java.dir" value="${build.dir}/java"/>
  <property name="build.gen-src.dir" value="${build.dir}/gen-src"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>
```

```

&database;
<!-- define the datasource.jndi.name in case the imported ejb file doesn't -->
<property name="datasource.jndi.name" value="${name}/datasource"/>
<path id="lib.class.path">
  <pathelement path="${database.driver.classpath}"/>
  <fileset dir="${lib.dir}">
    <include name="*.jar"/>
  </fileset>
  <!-- The middlegen jars -->
  <!--fileset dir="${basedir}/.."-->
  <fileset dir="${basedir}/middlegen-lib">
    <include name="*.jar"/>
  </fileset>
</path>
<target name="init">
  <available
    property="xdoclet1.2+"
    classname="xdoclet.modules.hibernate.HibernateDocletTask" classpathref="lib.class.path"/>
</target>
<!-- =====
-->
<!-- Fails if XDoclet 1.2.x is not on classpath -->
<!-- =====
-->
<target name="fail-if-no-xdoclet-1.2" unless="xdoclet1.2+">
  <fail>
    You must download several jar files before you can build Middlegen.
  </fail>
</target>
<!-- =====
-->
<!-- Create tables -->
<!-- =====
-->
<target
  name="create-tables"
  depends="init,fail-if-no-xdoclet-1.2,check-driver-present,panic-if-driver-not-present"
  description="Create tables"
>
  <echo>Creating tables using URL ${database.url}</echo>
  <sql
    classpath="${database.driver.classpath}"
    driver="${database.driver}"
    url="${database.url}"
    userid="${database.userid}"
    password="${database.password}"
    src="${database.script.file}"
    print="true"
    output="result.txt"
  />

```



```

</target>
<target name="check-driver-present">
  <available file="${database.driver.file}" type="file" property="driver.present"/>
</target>
<target name="panic-if-driver-not-present" unless="driver.present">
  <fail>
    The JDBC driver you have specified by including one of the files in ${basedir}/config/database
    doesn't exist. You have to download this driver separately and put it in ${database.driver.file}
    Please make sure you're using a version that is equal or superior to the one we looked for.
    If you name the driver jar file differently, please update the database.driver.file property
    in the ${basedir}/config/database/xxx.xml file accordingly.
  </fail>
</target>
<!-- =====
-->
<!-- Run Middlegen -->
<!-- =====
-->
<target
  name="middlegen"
  description="Run Middlegen"
  unless="middlegen.skip"
  depends="init,fail-if-no-xdoclet-1.2,check-driver-present,panic-if-driver-not-present"
>
  <mkdir dir="${build.gen-src.dir}"/>
  <echo message="Class path = ${basedir}"/>
  <taskdef
    name="middlegen"
    classname="middlegen.MiddlegenTask"
    classpathref="lib.class.path"
  />
  <middlegen
    appname="${name}"
    prefsdir="${src.dir}"
    gui="${gui}"
    databaseurl="${database.url}"
    initialContextFactory="${java.naming.factory.initial}"
    providerURL="${java.naming.provider.url}"
    datasourceJNDIName="${datasource.jndi.name}"
    driver="${database.driver}"
    username="${database.userid}"
    password="${database.password}"
    schema="${database.schema}"
    catalog="${database.catalog}"
  >
    <!--
      We can specify what tables we want Data generated for.
      If none are specified, Data will be generated for all tables.
      Comment out the <table> elements if you want to generate for all tables.
    -->

```

Also note that table names are CASE SENSITIVE for certain databases, so on e.g. Oracle you should specify table names in upper case.

```
-->
<!--table generate="true" name="flights" pktable="flights_pk"/>
<table name="reservations"/-->
<!--
```

If you want m:n relations, they must be specified like this. Note that tables declare in multiple locations must all have the same value of the generate attribute.

```
-->
<!--many2many>
  <tablea generate="true" name="persons"/>
  <jointable name="reservations" generate="false"/>
  <tableb generate="true" name="flights"/>
</many2many-->
<!-- Plugins - Only Hibernate Plugin has been included with this special distribution -->
```

<!--
If you want to generate XDoclet markup for hbm2java to include in the POJOs then set genXDocletTags to true. Also, composite keys are generated as external classes

which is

recommended. If you wish to keep them internal then set genIntergratedCompositeKeys to true.

Since r4 the ability to customise the selection of JavaTypes is now provided. There is a recommended type mapper provided as shown. It is optional - if not provided then Middlegen

itself will select the Java mapping (as it did previously).

These settings are optional thus if they are not define here values default to false.

```
-->
<hibernate
  destination="${build.gen-src.dir}"
  package="${name}.hibernate"
  genXDocletTags="false"
  genIntergratedCompositeKeys="false"
  javaTypeMapper="middlegen.plugins.hibernate.HibernateJavaTypeMapper"
/>
```

```
</middlegen>
<mkdir dir="${build.classes.dir}"/>
```

```
</target>
```

```
<!-- =====
```

```
-->
```

```
<!-- Compile business logic (hibernate) -->
```

```
<!-- =====
```

```
-->
```

```
<target name="compile-hibernate" depends="middlegen" description="Compile hibernate Business Domain Model">
```

```
<javac
  srcdir="${build.gen-src.dir}"
```

```

        destdir="${build.classes.dir}"
        classpathref="lib.class.path"
    >
        <include name="**/hibernate/**/*" />
    </javac>
</target>
<!-- =====
-->
<!-- Run hbm2java      depends="middlegen"      -->
<!-- =====
-->
<target name="hbm2java" description="Generate .java from .hbm files.">
    <taskdef
        name="hbm2java"
        classname="net.sf.hibernate.tool.hbm2java.Hbm2JavaTask"
        classpathref="lib.class.path"
    />

    <hbm2java output="${build.gen-src.dir}">
        <fileset dir="${build.gen-src.dir}">
            <include name="**/*.hbm.xml" />
        </fileset>
    </hbm2java>
</target>
<!-- =====
-->
<!-- Build everything      -->
<!-- =====
-->
<target name="all" description="Build everything" depends="compile-hibernate"/>
<!-- =====
-->
<!-- Clean everything      -->
<!-- =====
-->
<target name="clean" description="Clean all generated stuff">
    <delete dir="${build.dir}" />
</target>
<!-- =====
-->
<!-- Brings up the hsqldb admin tool      -->
<!-- =====
-->
<target name="hsqldb-gui" description="Brings up the hsqldb admin tool">
    <property name="database.urlparams"
value="?user=${database.userid}&password=${database.password}" />
    <java
        classname="org.hsqldb.util.DatabaseManager"
        fork="yes"

```



```

        classpath="${lib.dir}/hsqldb-1.7.1.jar;${database.driver.classpath}"
        failonerror="true"
    >
        <arg value="-url"/>
        <arg value="${database.url}${database.urlparams}"/>
        <arg value="-driver"/>
        <arg value="${database.driver}"/>
    </java>
</target>
<!-- =====
-->
<!-- Validate the generated xml mapping documents -->
<!-- =====
-->
<target name="validate">
    <xmlvalidate failonerror="no" lenient="no" warn="yes">
        <fileset dir="${build.gen-src.dir}/airline/hibernate" includes="*.xml" />
    </xmlvalidate>
</target>
</project>

```

代码说明:

- ☐ 修改 `<!ENTITY database SYSTEM "file:./config/database/hsqldb.xml">` 中的 `hsqldb.xml` 为 `mysql.xml`。
- ☐ 修改 `<property name="name" value="airline"/>` 中的 `value` 为 `com.gd.po`。
- ☐ 修改 `package="${name}.hibernate"` 为 `package="${name}"`。
- ☐ 修改 `genXDocletTags="false"` 为 `genXDocletTags="true"`，使其可以产生 XDoclet。

(4) 通过 cmd 控制台，进入 `D:\Middlegen-Hibernate-r5` 目录下，然后输入 `ant`，运行 Ant，如图 14.45 所示。

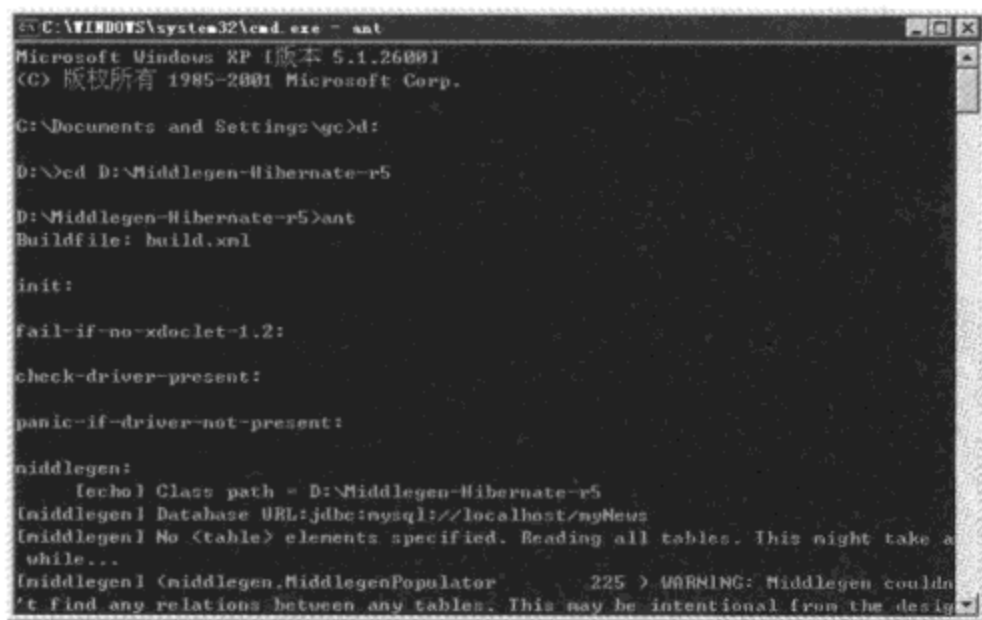


图 14.45 运行 Ant 的画面

(5) 运行 Ant 成功后，即可出现 Middlegen 的画面，它已经把前面建立的 4 个表结构导入，如图 14.46 所示。

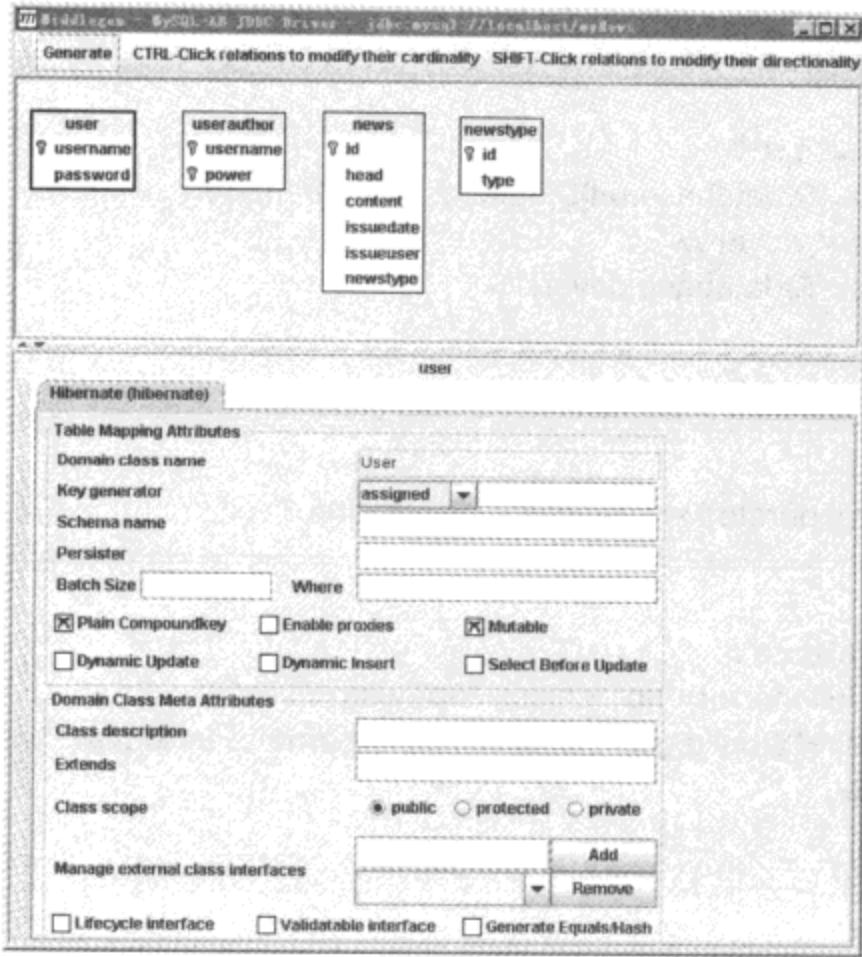


图 14.46 Middlegen 的画面

(6) 单击上面表结构中的 user，在 Middlegen 的画面的下半部分会出现与 user 相关的一些属性，这里默认 Key generator（主键生成方式）为 assigned，意思为主键生成方式由外部程序决定，Hibernate 不参与，其他不变，如图 14.47 所示。

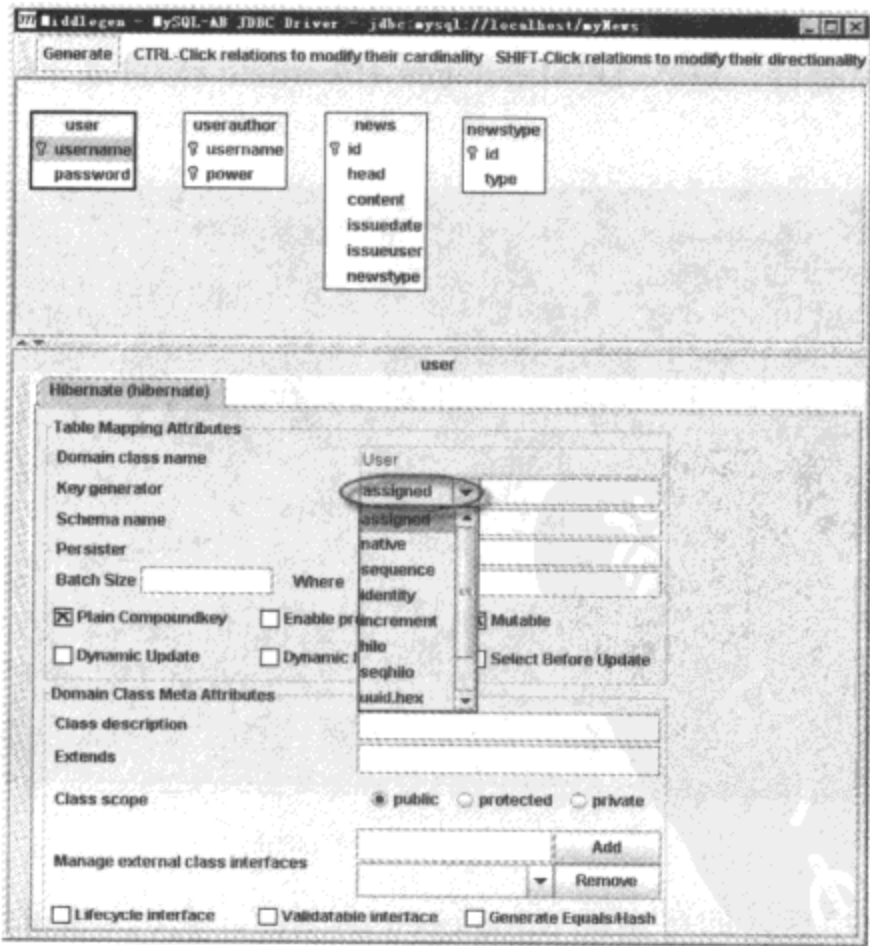


图 14.47 默认 Key generator（主键生成方式）为 assigned

(7) 同样默认 userAuthor 表的主键生成方式为 assigned，而 news 表和 newsType 表的主键生成方式则选择为 increment，意思为按数值顺序递增，news 表的主键生成方式如图 14.48 所示。

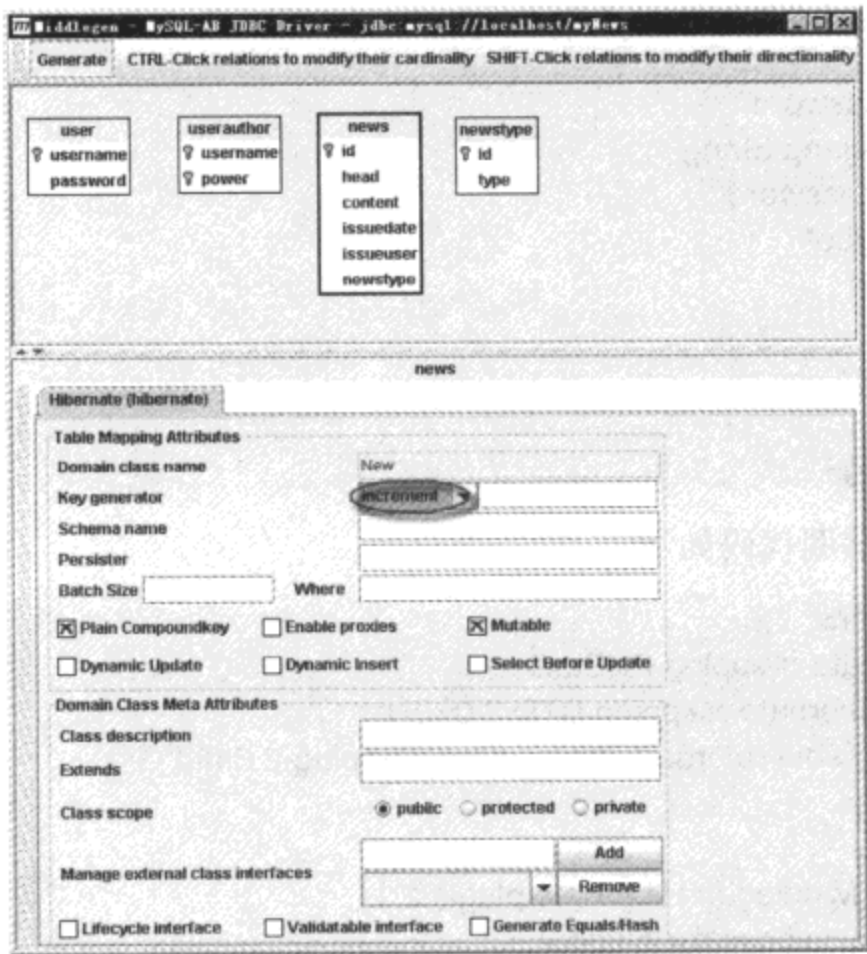


图 14.48 news 表的主键生成方式

注意：每设定一个表的主键和其他属性，都要先单击对应的表。

(8) 单击 Middlegen 画面左上方的 Generate 按钮，即可生成与这 4 个表相对应的映射文件 User.hbm.xml、New.hbm.xml、Newstype.hbm.xml、Userauthor.hbm.xml，这些文件存放在 Middlegen-Hibernate-r5/build/gen-src/com/gdc/po 目录下。User.hbm.xml 的示例代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >
<hibernate-mapping>
<!--
    Created by the Middlegen Hibernate plugin 2.1
    http://boss.bekk.no/boss/middlegen/
    http://www.hibernate.org/
-->
<class
    name="com.gd.po.User"
    table="user"
>
    <id
        name="username"
```

```

        type="java.lang.String"
        column="username"
    >
        <generator class="assigned" />
    </id>
    <property
        name="password"
        type="java.lang.String"
        column="password"
        not-null="true"
        length="32"
    />
    <!-- Associations -->
</class>
</hibernate-mapping>

```

New.hbm.xml 的示例代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >
<hibernate-mapping>
<!--
    Created by the Middlegen Hibernate plugin 2.1
    http://boss.bekk.no/boss/middlegen/
    http://www.hibernate.org/
-->
<class
    name="com.gd.po.New"
    table="news"
>
    <id
        name="id"
        type="java.lang.Integer"
        column="id"
    >
        <generator class="increment" />
    </id>
    <property
        name="head"
        type="java.lang.String"
        column="head"
        not-null="true"
        length="200"
    />
    <property
        name="content"
        type="java.lang.String"
        column="content"
    />

```

```

        not-null="true"
        length="8000"
    />
    <property
        name="issuedate"
        type="java.sql.Date"
        column="issuedate"
        not-null="true"
        length="10"
    />
    <property
        name="issueuser"
        type="java.lang.String"
        column="issueuser"
        not-null="true"
        length="32"
    />
    <property
        name="newstype"
        type="int"
        column="newstype"
        not-null="true"
        length="4"
    />
    <!-- Associations -->
</class>
</hibernate-mapping>

```

Newstype.hbm.xml 的示例代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >
<hibernate-mapping>
<!--
    Created by the Middlegen Hibernate plugin 2.1
    http://boss.bekk.no/boss/middlegen/
    http://www.hibernate.org/
-->
<class
    name="com.gd.po.Newstype"
    table="newstype"
>
    <id
        name="id"
        type="java.lang.Integer"
        column="id"
    >
        <generator class="increment" />
    </id>

```



```
</id>
<property
    name="type"
    type="java.lang.String"
    column="type"
    not-null="true"
    length="32"
/>
<!-- Associations -->
</class>
</hibernate-mapping>
```

Userauthor.hbm.xml 的示例代码如下：


```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >
<hibernate-mapping>
<!--
    Created by the Middlegen Hibernate plugin 2.1
    http://boss.bekk.no/boss/middlegen/
    http://www.hibernate.org/
-->
<class
    name="com.gd.po.Userauthor"
    table="userauthor"
>
    <composite-id name="comp_id" class="com.gd.po.UserauthorPK">
        <key-property
            name="username"
            column="username"
            type="java.lang.String"
            length="32"
        />
        <key-property
            name="power"
            column="power"
            type="java.lang.Integer"
            length="1"
        />
    </composite-id>
    <!-- Associations -->
    <!-- derived association(s) for compound key -->
    <!-- end of derived association(s) -->
</class>
</hibernate-mapping>
```

14.6.7 生成 POJO

把上面产生的 4 个映射文件转换成 POJO，要使用 hbm2java，具体步骤如下：

(1) 首先检查 D:\hibernate-extensions-2.1.3\tools\bin 目录下 setenv.bat 中 CP 的设定是否准确，用记事本打开 setenv.bat，然后查看 CP 中各种 jar 的设定是否正确，并增加 set HIBERNATE_HOME=D:\hibernate-3.2。setenv.bat 的示例代码如下：

```
@echo off
rem -----
rem Setup environment for hibernate tools
rem -----
set
JDBC_DRIVER=C:\Progra~1\SQLLIB\java\db2java.zip;C:\mm.mysql-2.0.14\mm.mysql-2.0.14-bin.j
ar
set HIBERNATE_HOME=D:\hibernate-3.2
set HIBERNATETOOLS_HOME=%~dp0..
echo HIBERNATETOOLS_HOME set to %HIBERNATETOOLS_HOME%
if "%HIBERNATE_HOME%" == "" goto noHIBERNATEHome
set CORELIB=%HIBERNATE_HOME%\lib
set LIB=%HIBERNATETOOLS_HOME%\lib
set
CP=%CLASSPATH%;%JDBC_DRIVER%;%HIBERNATE_HOME%\hibernate2.jar;%CORELIB%\c
ommons-logging-1.0.4.jar;%CORELIB%\commons-lang-1.0.1.jar;%CORELIB%\cglib-2.1.3.jar;%C
ORELIB%\dom4j-1.6.1.jar;%CORELIB%\odmg-3.0.jar;%CORELIB%\xml-apis.jar;%CORELIB%\xe
rces-2.6.2.jar;%CORELIB%\xalan-2.4.0.jar;%LIB%\jdom.jar;%CORELIB%\commons-collections-2.
1.1.jar;%LIB%\..\hibernate-tools.jar
if not "%HIBERNATE_HOME%" == "" goto end
:noHIBERNATEHome
echo HIBERNATE_HOME is not set. Please set HIBERNATE_HOME.
goto end
:end
```

 注意：因为 hibernate-extensions-2.1.3 目前还不支持 Hibernate 3，所以这里要麻烦一点。按照下载 Hibernate 3 同样的方法把 Hibernate 2 下载下来，并解压缩，把解压缩后 hibernate-2.1 目录下的 hibernate2.jar 放在 hibernate-3.2 目录下，把 hibernate-2.1\lib 目录下的 commons-lang-1.0.1.jar、odmg-3.0.jar、xalan-2.4.0.jar 这 3 个 jar 放在 hibernate-3.2\lib 目录下。然后修改 CP 中各个 jar 的版本号，使其与 hibernate-3.2\lib 目录下相应 jar 的版本号一致。

(2) 通过 cmd 控制台，进入 D:\hibernate-extensions-2.1.3\tools\bin 目录下，然后输入“hbm2java D:\Middlegen-Hibernate-r5\build\gen-src\com\gd\po*.xml --output=D:\Middlegen-Hibernate-r5\build\gen-src\”，即可在 D:\Middlegen-Hibernate-r5\build\gen-src\com\gd\po 目录下生成 User.java、New.java、Newstype.java、Userauthor.java 和 UserauthorPK.java，如图 14.49 所示。

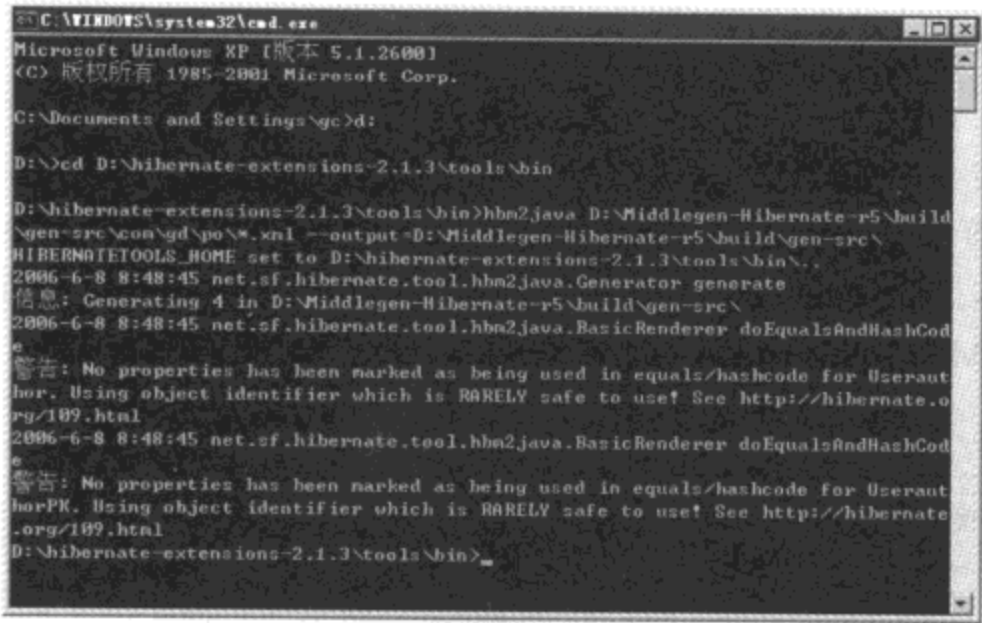


图 14.49 运行 hbm2java

(3) User.java 的示例代码如下：

```

//***** User.java *****
package com.gd.po;
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;
/** @author Hibernate CodeGenerator */
public class User implements Serializable {
    /** identifier field */
    private String username;
    /** persistent field */
    private String password;
    /** full constructor */
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    /** default constructor */
    public User() {
    }
    public String getUsername() {
        return this.username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return this.password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String toString() {
        return new ToStringBuilder(this)

```

```
        .append("username", getUsername())
        .toString();
    }
}
```

(4) New.java 的示例代码如下:

```
/****** New.java*****
package com.gd.po;

import java.io.Serializable;
import java.util.Date;
import org.apache.commons.lang.builder.ToStringBuilder;

/** @author Hibernate CodeGenerator */
public class New implements Serializable {

    /** identifier field */
    private Integer id;

    /** persistent field */
    private String head;

    /** persistent field */
    private String content;

    /** persistent field */
    private Date issuedate;

    /** persistent field */
    private String issueuser;

    /** persistent field */
    private int newstype;

    /** full constructor */
    public New(String head, String content, Date issuedate, String issueuser, int newstype) {
        this.head = head;
        this.content = content;
        this.issuedate = issuedate;
        this.issueuser = issueuser;
        this.newstype = newstype;
    }

    /** default constructor */
    public New() {
    }

    public Integer getId() {
        return this.id;
    }
}
```



```
}

public void setId(Integer id) {
    this.id = id;
}

public String getHead() {
    return this.head;
}

public void setHead(String head) {
    this.head = head;
}

public String getContent() {
    return this.content;
}

public void setContent(String content) {
    this.content = content;
}

public Date getIssuedate() {
    return this.issuedate;
}

public void setIssuedate(Date issuedate) {
    this.issuedate = issuedate;
}

public String getIssueuser() {
    return this.issueuser;
}

public void setIssueuser(String issueuser) {
    this.issueuser = issueuser;
}

public int getNewstype() {
    return this.newstype;
}

public void setNewstype(int newstype) {
    this.newstype = newstype;
}

public String toString() {
    return new ToStringBuilder(this)
        .append("id", getId())
        .toString();
}
```

```
}  
}
```

(5) Newstype.java 的示例代码如下:

```
/****** Newstype.java *****/  
package com.gd.po;  
  
import java.io.Serializable;  
import org.apache.commons.lang.builder.ToStringBuilder;  
  
/** @author Hibernate CodeGenerator */  
public class Newstype implements Serializable {  
  
    /** identifier field */  
    private Integer id;  
  
    /** persistent field */  
    private String type;  
  
    /** full constructor */  
    public Newstype(String type) {  
        this.type = type;  
    }  
  
    /** default constructor */  
    public Newstype() {  
    }  
  
    public Integer getId() {  
        return this.id;  
    }  
  
    public void setId(Integer id) {  
        this.id = id;  
    }  
  
    public String getType() {  
        return this.type;  
    }  
  
    public void setType(String type) {  
        this.type = type;  
    }  
  
    public String toString() {  
        return new ToStringBuilder(this)  
            .append("id", getId())  
            .toString();  
    }  
}
```

```
}  
}
```

(6) Userauthor.java 的示例代码如下:

```
/** ***** Userauthor.java ***** */  
package com.gd.po;  
  
import java.io.Serializable;  
import org.apache.commons.lang.builder.EqualsBuilder;  
import org.apache.commons.lang.builder.HashCodeBuilder;  
import org.apache.commons.lang.builder.ToStringBuilder;  
  
/** @author Hibernate CodeGenerator */  
public class Userauthor implements Serializable {  
  
    /** identifier field */  
    private com.gd.po.UserauthorPK comp_id;  
  
    /** full constructor */  
    public Userauthor(com.gd.po.UserauthorPK comp_id) {  
        this.comp_id = comp_id;  
    }  
  
    /** default constructor */  
    public Userauthor() {  
    }  
  
    public com.gd.po.UserauthorPK getComp_id() {  
        return this.comp_id;  
    }  
  
    public void setComp_id(com.gd.po.UserauthorPK comp_id) {  
        this.comp_id = comp_id;  
    }  
  
    public String toString() {  
        return new ToStringBuilder(this)  
            .append("comp_id", getComp_id())  
            .toString();  
    }  
  
    public boolean equals(Object other) {  
        if ( (this == other) ) return true;  
        if ( !(other instanceof Userauthor) ) return false;  
        Userauthor castOther = (Userauthor) other;  
        return new EqualsBuilder()  
            .append(this.getComp_id(), castOther.getComp_id())  
            .isEquals();  
    }  
}
```



```
}

public int hashCode() {
    return new HashCodeBuilder()
        .append(getComp_id())
        .toHashCode();
}
}
```

(7) UserauthorPK.java 的示例代码如下:

```
/** ***** UserauthorPK.java ***** */
package com.gd.po;

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/** @author Hibernate CodeGenerator */
public class UserauthorPK implements Serializable {

    /** identifier field */
    private String username;

    /** identifier field */
    private Integer power;

    /** full constructor */
    public UserauthorPK(String username, Integer power) {
        this.username = username;
        this.power = power;
    }

    /** default constructor */
    public UserauthorPK() {}

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

    public void setUsername(String username) {
        this.username = username;
    }

    public Integer getPower() {
        return this.power;
    }
}
```



```
public void setPower(Integer power) {
    this.power = power;
}

public String toString() {
    return new ToStringBuilder(this)
        .append("username", getUsername())
        .append("power", getPower())
        .toString();
}

public boolean equals(Object other) {
    if ( (this == other) ) return true;
    if ( !(other instanceof UserauthorPK) ) return false;
    UserauthorPK castOther = (UserauthorPK) other;
    return new EqualsBuilder()
        .append(this.getUsername(), castOther.getUsername())
        .append(this.getPower(), castOther.getPower())
        .isEquals();
}

public int hashCode() {
    return new HashCodeBuilder()
        .append(getUsername())
        .append(getPower())
        .toHashCode();
}
```

把生成的 XML 和 POJO 放在 myNews 工程里的 com.gd.po 包下。

⚠注意：这里需要把 commons-lang-1.0.1.jar 也放在 myNews 工程的 WEB-INF/lib 下，并加入到 CLASSPATH 中。

14.7 编写新闻发布系统的 VO 和 DAO

前面通过 Hibernate，生成了 POJO，在该应用程序中，这里把 PO 和 VO 区分开，PO 是用来持久化数据的，而 VO 是用来传递数据的。

PO 放在 com.gd.po 包里，VO 放在 com.gd.vo 包里。

14.7.1 用户类 User.java

该类放在 com.gd.vo 包里，主要用来负责从页面中返回用户填写的有关用户名和密码的相关信息，并把业务逻辑处理后的用户信息返回页面。User.java 的示例代码如下：

```
//***** User.java*****
package com.gd.vo;
import com.gd.dao.UserDAO;
import com.gd.service.Login;
public class User {
    public String getMsg(){
        return msg;
    }
    public void setMsg(String msg){
        this.msg = msg;
    }
    public String getPassword2(){
        return password2;
    }
    public void setPassword2(String password2){
        this.password2 = password2;
    }
    public String getPassword1(){
        return password1;
    }
    public void setPassword1(String password1){
        this.password1 = password1;
    }
    public String getUsername(){
        return username;
    }
    public void setUsername(String username){
        this.username = username;
    }
    //用来验证用户填写的信息是否正确
    public boolean validate(Login login) {
        User user = login.queryUser(getUsername());
        //用来验证用户填写的密码和数据库中的密码是否一致
        if (user != null && getPassword1().equals(user.getPassword1())) {
            return true;
        } else {
            return false;
        }
    }
    private String msg;
    private String password2;
    private String password1;
    private String username;
}
```

代码说明：Validate()方法，主要用来负责验证用户在页面中填写的用户名和密码是否和数据库中存储的一致。

14.7.2 用户权限类 UsersAuthor.java

该类放在 com.gd.vo 包里，主要用来负责处理用户的权限。UsersAuthor.java 的示例代码如下：

```
/****** UsersAuthor.java*****  
package com.gd.vo;  
public class UsersAuthor {  
    //构造函数  
    public void setUsersAuthor(User InkUser, int power) {  
        InkUser = InkUser;  
        power = power;  
    }  
    //根据用户类获取权限  
    public int getAuthorByUser(User InkUser) {  
        return power;  
    }  
    public UsersAuthor(User InkUser, int power) {  
        InkUser = InkUser;  
        power = power;  
    }  
    private int power;  
    /**  
     * @clientCardinality 1  
     * @supplierCardinality 0..n  
     */  
    private User InkUser;  
}
```

14.7.3 新闻类 News.java

该类放在 com.gd.vo 包里，主要用来负责从页面返回用户填写的有关新闻标题和内容等信息，然后把业务逻辑处理后的新闻有关信息返回页面。News.java 的示例代码如下：

```
/****** News.java*****  
package com.gd.vo;  
import java.util.ArrayList;  
import java.util.Date;  
import java.util.List;  
public class News {  
    public int getId(){  
        return id;  
    }  
    public void setId(int id){  
        this.id = id;  
    }  
}
```



```
public String getMsg(){
    return msg;
}
public void setMsg(String msg){
    this.msg = msg;
}
public String getHead(){
    return head;
}
public void setHead(String head){
    this.head = head;
}
public String getContent(){
    return content;
}
public void setContent(String content){
    this.content = content;
}
public Date getDate(){
    return date;
}
public void setDate(Date date){
    this.date = date;
}
//根据新闻类别 id 获取新闻
public List getNewsByType(int id) {

    return new ArrayList();
}
public void saveNews() {
}
private int id;
private String msg;
private String head;
private String content;
private Date date;
/**
 * @clientCardinality n
 * @supplierCardinality 1
 */
private NewsType lnkNewsType;
/**
 * @clientCardinality n
 * @supplierCardinality 1
 */
private User lnkUser;

public User getLnkUser(){
    return lnkUser;
}
```



```
public void setLnkUser(User lnkUser){
    this.lnkUser = lnkUser;
}

public NewsType getLnkNewsType(){
    return lnkNewsType;
}

public void setLnkNewsType(NewsType lnkNewsType){
    this.lnkNewsType = lnkNewsType;
}
}
```

14.7.4 新闻类别类 NewsType.java

该类放在 com.gd.vo 包里，主要用来负责从数据库中获取新闻类别的相关信息并显示在页面上。NewsType.java 的示例代码如下：

```
/** ***** NewsType.java ***** */
package com.gd.vo;
public class NewsType {
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getTypeName(){
        return typename;
    }
    public void setTypeName(String typename){
        this.typename = typename;
    }
    public String getTypeById (int id) {
        return typename;
    }
    private int id;
    private String typename;
}
```

14.7.5 用户 DAO 接口 UserDAO.java

该类放在 com.gd.dao 包里，主要用来负责新增、修改、删除以及查询用户的基本信息，这是一个接口，主要是为了实现 IoC。UserDAO.java 的示例代码如下：

```
/** ***** UserDAO.java ***** */
package com.gd.dao;
import com.gd.po.*;
```

```
public interface UserDao {  
    //新增用户  
    public abstract void createUser(User user);  
    //修改用户  
    public abstract void updateUser(User user);  
    //删除用户  
    public abstract void deleteUser(User user);  
    //根据名字查询用户  
    public abstract User queryUser(String name);  
}
```

14.7.6 新闻 DAO 接口 NewsDAO.java

该类放在 com.gd.dao 包里，主要用来负责新增、修改、删除以及查询新闻的基本信息，这是一个接口，主要是为了实现 IoC。NewsDAO.java 的示例代码如下：

```
//***** NewsDAO.java*****  
package com.gd.dao;  
import java.util.List;  
import com.gd.po.*;  
public interface NewsDAO {  
    //创建新闻  
    public abstract void createNews(New news);  
    //修改新闻  
    public abstract void updateNews(New news);  
    //删除新闻  
    public abstract void deleteNews(New news);  
    //根据 id 查询新闻  
    public abstract New queryNews(Integer id);  
    //根据新闻类别 id 获取新闻  
    public abstract List getNewsByType(Integer id);  
}
```

14.7.7 新闻类别 DAO 接口 NewsTypeDAO.java

该类放在 com.gd.dao 包里，主要用来负责新增、修改、删除以及查询新闻类别的信息，这是一个接口，主要是为了实现 IoC。NewsTypeDAO.java 的示例代码如下：

```
//***** NewsTypeDAO.java*****  
package com.gd.dao;  
import java.util.List;  
import com.gd.po.*;  
  
public interface NewsTypeDAO {  
    //创建新闻类别  
    public abstract void createNewsType(Newstype Newstype);  
    //修改新闻类别
```

```
public abstract void updateNewsType(Newstype Newstype);  
//删除新闻类别  
public abstract void deleteNewsType(Newstype Newstype);  
//获取新闻类别  
public abstract List queryNewsType();  
}
```

14.7.8 用户 DAO 实现类 UserDAOImpl.java

该类放在 com.gd.dao.impl 包里，主要用来负责新增、修改、删除以及查询用户的基本信息，这是接口 UserDAO 的实现类，主要是为了实现 IoC。UserDAOImpl.java 的示例代码如下：

```
//***** UserDAOImpl.java*****  
package com.gd.dao.impl;  
  
import java.util.ArrayList;  
import java.util.List;  
  
import org.apache.log4j.Logger;  
import org.hibernate.SessionFactory;  
  
import com.gd.dao.UserDAO;  
import com.gd.po.User;  
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;  
  
public class UserDAOImpl extends HibernateDaoSupport implements UserDAO {  
    private Logger logger = Logger.getLogger(this.getClass().getName());  
    //定义 SessionFactory，负责对数据库的连接  
    private SessionFactory sessionFactory;  
    //定义 Hibernate 的 hql 语句，用来根据用户名查询用户  
    private String hql = "from User u where u.username = ?";  
    //创建用户  
    public void createUser(User user) {  
        this.getHibernateTemplate().save(user);  
    }  
    //修改用户  
    public void updateUser(User user) {  
        this.getHibernateTemplate().update(user);  
    }  
    //删除用户  
    public void deleteUser(User user) {  
        this.getHibernateTemplate().delete(user);  
    }  
    //根据用户名查询用户  
    public User queryUser(String name){  
        List userList;  
        //假如查询结果为空，则 new 一个新的用户
```



```

        if (this.getHibernateTemplate().find(hql, name) == null ) {
            userList = new ArrayList();
            userList.add(new User());
        } else {
            userList = this.getHibernateTemplate().find(hql, name);
        }
        return (User)userList.get(0);
    }
}

```

14.7.9 新闻 DAO 实现类 NewsDAOImpl.java

该类放在 com.gd.dao.impl 包里，主要用来负责新增、修改、删除以及查询新闻的基本信息，这是接口 NewsDAO 的实现类，主要是为了实现 IoC。NewsDAOImpl.java 的示例代码如下：

```

//***** NewsDAOImpl.java*****
package com.gd.dao.impl;

import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Logger;
import org.hibernate.SessionFactory;

import com.gd.dao.NewsDAO;
import com.gd.po.New;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
//实现 NewsDAO 接口，实现依赖注入
public class NewsDAOImpl extends HibernateDaoSupport implements NewsDAO {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //定义 SessionFactory，负责对数据库的连接
    private SessionFactory sessionFactory;
    //定义 Hibernate 的 hql 语句，用来根据新闻的 id 查询新闻
    private String hql = "from New n where n.id = ?";
    //定义 Hibernate 的 hql 语句，用来根据新闻类别的 id 查询新闻
    private String hqlByNewsType = "from New n where n.newstype = ?";
    //创建新闻
    public void createNews(New news) {
        this.getHibernateTemplate().save(news);
    }
    //修改新闻
    public void updateNews(New news) {
        this.getHibernateTemplate().update(news);
    }
    //删除新闻
    public void deleteNews(New news) {

```



```

        this.getHibernateTemplate().delete(news);
    }
    //根据新闻 id 获取新闻
    public New queryNews(Integer id){
        List newsList;
        if (this.getHibernateTemplate().find(hql, id) == null )
            newsList = new ArrayList();
            userList.add(new New());
        else
            newsList = this.getHibernateTemplate().find(hql, id);
        return (New)newsList.get(0);
    }
    //根据新闻类别 id 获取同一类型的新闻
    public List getNewsByType(Integer id){
        List newsList;
        if (this.getHibernateTemplate().find(hqlByNewsType, id) == null )
            newsList = new ArrayList();
        else
            newsList = this.getHibernateTemplate().find(hqlByNewsType, id);
        return newsList;
    }
}

```

14.7.10 新闻类别 DAO 实现类 NewsTypeDAOImpl.java

该类放在 com.gd.dao.impl 包里，主要用来负责新增、修改、删除以及查询新闻类别的信息，这是接口 NewsTypeDAO 的实现类，主要是为了实现 IoC。NewsTypeDAOImpl.java 的示例代码如下：

```

//***** NewsTypeDAOImpl.java*****
package com.gd.dao.impl;

import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Logger;
import org.hibernate.SessionFactory;

import com.gd.dao.NewsTypeDAO;
import com.gd.po.Newstype;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

public class NewsTypeDAOImpl extends HibernateDaoSupport implements NewsTypeDAO {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //定义 SessionFactory，负责对数据库的连接
    private SessionFactory sessionFactory;
    //定义 Hibernate 的 hql 语句，用来查询新闻类别
    private String hql = "from Newstype ";
}

```

```
//创建新闻类别
public void createNewsType(Newstype newsType) {
    this.getHibernateTemplate().save(newsType);
}
//修改新闻类别
public void updateNewsType(Newstype newsType) {
    this.getHibernateTemplate().update(newsType);
}
//删除新闻类别
public void deleteNewsType(Newstype newsType) {
    this.getHibernateTemplate().delete(newsType);
}
//查询新闻类别
public List queryNewsType(){
    List newsTypeList;
    if (this.getHibernateTemplate().find(hql) == null )
        newsTypeList = new ArrayList();
    else
        newsTypeList = this.getHibernateTemplate().find(hql);
    return newsTypeList;
}
}
```

14.8 编写新闻发布系统的业务逻辑类

上面的持久层程序主要是负责持久化数据的存入和取出，而对业务逻辑的处理这里主要放在 service 包中。

14.8.1 登录接口 Login.java

该类放在 com.gd.service 包里，主要用来负责查询用户的信息并验证，这是一个接口，主要是为了实现 IoC。Login.java 的示例代码如下：

```
/****** Login.java *****/
package com.gd.service;

import com.gd.dao.UserDAO;
import com.gd.vo.User;

public interface Login {
    //定义根据用户名查询用户的虚函数，用于依赖注入
    public abstract User queryUser(String username);
}
```

14.8.2 注册接口 Regedit.java

该类放在 com.gd.service 包里，主要用来负责新增、修改和删除用户的信息，这是一个接口，主要是为了实现 IoC。Regedit.java 的示例代码如下：

```
/** ***** Regedit.java ***** */
package com.gd.service;

import com.gd.vo.User;

public interface Regedit {
    //定义保存用户的虚函数，用于依赖注入
    public abstract void saveUser(User user);
    //定义修改用户的虚函数，用于依赖注入
    public abstract void deleteUser(User user);
    //定义删除用户的虚函数，用于依赖注入
    public abstract void updateUser(User user);
}
```

14.8.3 发布接口 Release.java

该类放在 com.gd.service 包里，主要用来负责新增、修改和删除新闻的信息，这是一个接口，主要是为了实现 IoC。Release.java 的示例代码如下：

```
/** ***** Release.java ***** */
package com.gd.service;

import com.gd.vo.News;
import com.gd.vo.User;

public interface Release {
    //定义保存新闻的虚函数，用于依赖注入
    public abstract void saveNews(News news);
    //定义删除新闻的虚函数，用于依赖注入
    public abstract void deleteNews(News news);
    //定义修改新闻的虚函数，用于依赖注入
    public abstract void updateNews(News news);
}
```

14.8.4 登录实现类 LoginImpl.java

该类放在 com.gd.service.impl 包里，主要用来负责查询用户的信息，这是接口 Login 的实现类，主要是为了实现 IoC。LoginImpl.java 的示例代码如下：

```
/** ***** LoginImpl.java ***** */
package com.gd.service.impl;
```



```
import com.gd.dao.UserDAO;
import com.gd.service.Login;
import com.gd.vo.User;
//实现 Login 接口，用来根据用户名获取用户的信息
public class LoginImpl implements Login {
    private UserDAO userDAO ;
    //根据用户名获取用户的信息
    public User queryUser(String username) {
        //把从数据库得到的 po 转换为 vo
        User user = new User();
        user.setUsername(userDAO.queryUser(username).getUsername());
        user.setPassword1(userDAO.queryUser(username).getPassword());
        return user;
    }

    public void setUserDAO (UserDAO userDAO) {
        this.userDAO = userDAO;
    }
    public UserDAO getUserDAO () {
        return this.userDAO;
    }
}
```

代码说明：设定 userDAO 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。

14.8.5 注册实现类 RegeditImpl.java

该类放在 com.gd.service.impl 包里，主要用来负责新增、修改和删除用户的信息，这是接口 Regedit 的实现类，主要是为了实现 IoC。RegeditImpl.java 的示例代码如下：

```
/** ***** RegeditImpl.java ***** */
package com.gd.service.impl;

import com.gd.dao.UserDAO;
import com.gd.service.Regedit;
import com.gd.vo.User;
//实现 Regedit 接口，用来在注册时保存、修改、删除和查询用户信息
public class RegeditImpl implements Regedit {
    private UserDAO userDao ;
    //保存用户信息
    public void saveUser(User user) {
        //把从数据库得到的 vo 转换为 po
        com.gd.po.User userpo = new com.gd.po.User();
        userpo.setUsername(user.getUsername());
        userpo.setPassword(user.getPassword1());
        userDao.createUser(userpo);
    }
}
```



```

//修改用户信息
public void updateUser(User user) {
    //把从数据库得到的 vo 转换为 po
    com.gd.po.User userpo = new com.gd.po.User();
    userpo.setUsername(user.getUsername());
    userpo.setPassword(user.getPassword1());
    userDao.updateUser(userpo);
}
//修改删除信息
public void deleteUser(User user) {
    //把从数据库得到的 vo 转换为 po
    com.gd.po.User userpo = new com.gd.po.User();
    userpo.setUsername(user.getUsername());
    userpo.setPassword(user.getPassword1());
    userDao.deleteUser(userpo);
}
//根据用户名查询用户信息
public User queryUser(String username) {
    //把从数据库得到的 po 转换为 vo
    User user = new User();
    user.setUsername(userDao.queryUser(username).getUsername());
    user.setPassword1(userDao.queryUser(username).getPassword());
    return user;
}
//用于实现依赖注入，把 userDao 注入该类
public void setUserDao (UserDAO userDao) {
    this.userDao = userDao;
}
public UserDAO getUserDao () {
    return this.userDao;
}
}

```

代码说明：设定 userDao 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。

14.8.6 发布实现类 ReleaseImpl.java

该类放在 com.gd.service.impl 包里，主要用来负责新增、修改和删除新闻的信息，这是接口 Release 的实现类，主要是为了实现 IoC。ReleaseImpl.java 的示例代码如下：

```

//***** ReleaseImpl.java*****
package com.gd.service.impl;

import com.gd.dao.NewsDAO;
import com.gd.dao.UserDAO;
import com.gd.service.Login;
import com.gd.service.Release;
import com.gd.vo.News;

```

```
import com.gd.vo.User;
//实现 Release 接口，用来在注册时保存、修改、删除和查询新闻信息
public class ReleaseImpl implements Release {
    private NewsDAO newsDao ;
    //保存新闻
    public void saveNews(News news) {
        //把从数据库得到的 po 转换为 vo
        com.gd.po.New newpo = new com.gd.po.New();
        newpo.setHead(news.getHead());
        newpo.setContent(news.getContent());
        newpo.setIssueuser(news.getLnkUser().getUsername());
        newpo.setIssuedate(news.getDate());
        newpo.setNewstype(news.getLnkNewsType().getId());
        newsDao.createNews(newpo);
    }
    //修改新闻
    public void updateNews(News news) {
        //把从数据库得到的 po 转换为 vo
        com.gd.po.New newpo = new com.gd.po.New();
        newpo.setHead(news.getHead());
        newpo.setContent(news.getContent());
        newpo.setIssueuser(news.getLnkUser().getUsername());
        newpo.setIssuedate(news.getDate());
        newpo.setNewstype(news.getLnkNewsType().getId());
        newsDao.updateNews(newpo);
    }
    //删除新闻
    public void deleteNews(News news) {
        //把从数据库得到的 po 转换为 vo
        com.gd.po.New newpo = new com.gd.po.New();
        newpo.setHead(news.getHead());
        newpo.setContent(news.getContent());
        newpo.setIssueuser(news.getLnkUser().getUsername());
        newpo.setIssuedate(news.getDate());
        newpo.setNewstype(news.getLnkNewsType().getId());
        newsDao.deleteNews(newpo);
    }
    //根据新闻 id 获取新闻
    public News queryNews(int id) {
        //把从数据库得到的 po 转换为 vo
        News news = new News();
        news.setHead(newsDao.queryNews(id).getHead());
        news.setContent(newsDao.queryNews(id).getContent());
        news.setLnkUser(getLogin().queryUser(newsDao.queryNews(id).getIssueuser()));
        news.setDate(newsDao.queryNews(id).getIssuedate());
        return news;
    }
    //用于实现依赖注入，把 newsDao 注入该类
    public void setNewsDao (NewsDAO newsDao) {
        this.newsDao = newsDao;
    }
}
```

```
}  
public NewsDAO getNewsDao () {  
    return this.newsDao;  
}  
//用于实现依赖注入，把 login 注入该类  
private Login login;  
public Login getLogin () {  
    return login;  
}  
public void setLogin (Login login) {  
    this.login = login;  
}  
}
```

代码说明：

- ❑ 设定 newsDao 和 login 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。
- ❑ 在业务逻辑类里，进行了从 PO 到 VO 和从 VO 到 PO 的转换。

14.9 编写新闻发布系统的控制器类

新闻发布系统的控制器类，主要用来转发从浏览器那里接收到的用户的请求。

14.9.1 登录控制器类 LoginController.java

该类放在 com.gd.service.action 包里，主要用来负责对页面传来的用户登录请求进行转发，这个类继承了 Spring 提供的 SimpleFormController 类。LoginController.java 的示例代码如下：

```
/** ***** LoginController.java *****  
package com.gd.action;  
  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import javax.servlet.http.HttpSession;  
  
import org.apache.log4j.Logger;  
import org.springframework.validation.BindException;  
import org.springframework.web.servlet.ModelAndView;  
import org.springframework.web.servlet.mvc.SimpleFormController;  
  
import com.gd.dao.NewsTypeDAO;  
import com.gd.dao.UserDAO;
```



```
import com.gd.service.Login;
import com.gd.vo.User;
//继承 SimpleFormController, 用来负责对页面传来的用户登录请求进行转发
public class LoginController extends SimpleFormController {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    //覆写 onSubmit()方法, 用来接收用户传来的提交信息
    public ModelAndView onSubmit(HttpServletRequest req, HttpServletResponse res, Object
command, BindException errors) throws Exception {
        HttpSession session = req.getSession(true);
        //将提交的信息转换为 User 类存储
        User user = (User)command;
        Map model = errors.getModel();
        model.put("user", user);
        //用于验证用户提交的信息
        if (user.validate(getLogin())) {
            //如果验证成功, 则将用户信息存储在 session 中, 供其他页面使用
            session.setAttribute("user", user);
            //获取新闻类别, 用于在发布页面中显示新闻类别的下拉菜单
            List newsTypes = getNewsTypeDAO().queryNewsType();
            model.put("newsTypes", newsTypes);
            //验证成功, 返回在配置文档中定义的成功页面
            return new ModelAndView(getSuccessView(), model);
        } else {
            //验证失败, 返回在配置文档中定义的失败页面
            return new ModelAndView(getFormView(), model);
        }
    }
}
//下面都是为了实现依赖注入
private NewsTypeDAO newsTypeDAO;
public NewsTypeDAO getNewsTypeDAO () {
    return newsTypeDAO;
}
public void setNewsTypeDAO (NewsTypeDAO newsTypeDAO) {
    this.newsTypeDAO = newsTypeDAO;
}
private Login login;
public Login getLogin () {
    return login;
}
public void setLogin (Login login) {
    this.login = login;
}
}
```

代码说明: 设定 newsTypeDAO 和 login 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。

14.9.2 注册控制器类 RegeditController.java

该类放在 com.gd.service.action 包里，主要用来负责对页面传来的用户注册请求进行转发，这个类继承了 Spring 提供的 SimpleFormController 类。RegeditController.java 的示例代码如下：

```
//***** RegeditController.java*****
package com.gd.action;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;

import com.gd.dao.NewsTypeDAO;
import com.gd.service.Regedit;
import com.gd.vo.NewsType;
import com.gd.vo.User;
//继承 SimpleFormController,用来负责对页面传来的用户注册请求进行转发
public class RegeditController extends SimpleFormController {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private Regedit regedit;
    public Regedit getRegedit () {
        return regedit;
    }
    public void setRegedit (Regedit regedit) {
        this.regedit = regedit;
    }
    //覆写 onSubmit()方法，用来接收用户传来的注册信息
    public ModelAndView onSubmit(HttpServletRequest req, HttpServletResponse res, Object
command, BindException errors) throws Exception {
        HttpSession session = req.getSession(true);
        //将提交的信息转换为 User 类存储
        User user = (User)command;
        Map model = errors.getModel();
        getRegedit().saveUser(user);
        //获取新闻类别，用于在发布页面中显示新闻类别的下拉菜单
        List newsTypes = getNewsTypeDAO().queryNewsType();
        model.put("newsTypes", newsTypes);
    }
}
```

```

        model.put("user", user);
        //注册成功, 则将用户信息存储在 session 中, 供其他页面使用
        session.setAttribute("user", user);
        return new ModelAndView(getSuccessView(), model);
    }
    private NewsTypeDAO newsTypeDAO;
    public NewsTypeDAO getNewsTypeDAO () {
        return newsTypeDAO;
    }
    public void setNewsTypeDAO (NewsTypeDAO newsTypeDAO) {
        this.newsTypeDAO = newsTypeDAO;
    }
}

```

代码说明：设定 newsTypeDAO 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。

14.9.3 发布控制器类 ReleaseController.java

该类放在 com.gd.service.action 包里, 主要用来负责对页面传来的发布新闻的请求进行转发, 这个类继承了 Spring 提供的 SimpleFormController 类。ReleaseController.java 的示例代码如下:

```

//***** ReleaseController.java*****
package com.gd.action;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;

import com.gd.dao.NewsDAO;
import com.gd.dao.NewsTypeDAO;
import com.gd.po.Newstype;
import com.gd.service.Release;
import com.gd.util.NewsUtil;
import com.gd.vo.News;
import com.gd.vo.NewsType;
import com.gd.vo.User;
//继承 SimpleFormController, 用来负责对页面传来的发布新闻的请求进行转发
public class ReleaseController extends SimpleFormController {

```

```

private Logger logger = Logger.getLogger(this.getClass().getName());
private Release release;
public Release getRelease () {
    return release;
}
public void setRelease (Release release) {
    this.release = release;
}
//覆写 onSubmit()方法，用来接收用户提交的新闻信息
public ModelAndView onSubmit(HttpServletRequest req, HttpServletResponse res, Object
command, BindException errors) throws Exception {
    HttpSession session = req.getSession(true);
    //获取页面中用户选择的新闻类别 id
    NewsType newsType = new NewsType();
    newsType.setId(Integer.parseInt(req.getParameter("newsType")));
    //把从页面中抓取的新闻信息转换成 News 类存储
    News news = (News)command;
    //获取保存在 session 中的用户信息
    User user = (User)session.getAttribute("user");
    ///通过辅助类获取新闻发布日期
    news.setDate(NewsUtil.parseDateDayFormat(NewsUtil.getCurrentDate()));
    news.setLnkUser(user);
    news.setLnkNewsType(newsType);
    //保存新闻
    getRelease().saveNews(news);
    Map model = errors.getModel();
    Map mapNews = new HashMap();
    //获取新闻类别
    List listNewsType = getNewsTypeDAO().queryNewsType();
    //根据新闻类别获取对应的新闻，并将其存储在 Map 中
    for (int i = 0; listNewsType != null && i < listNewsType.size(); i++) {
        Newstype newsTypeTemp = (Newstype)listNewsType.get(i);
        //根据新闻类别获取对应的新闻
        List listNews = getNewsDAO().getNewsByType(newsTypeTemp.getId());
        mapNews.put(newsTypeTemp.getId(), listNews);
    }
    //将信息存储在 model 中，供返回页面使用
    model.put("mapNews", mapNews);
    model.put("listNewsType", listNewsType);
    model.put("news", news);
    return new ModelAndView(getSuccessView(), model);
}
//下面都是为了实现依赖注入
private NewsDAO newsDAO;
public NewsDAO getNewsDAO () {
    return newsDAO;
}
public void setNewsDAO (NewsDAO newsDAO) {

```



```

        this.newsDAO = newsDAO;
    }
    private NewsTypeDAO newsTypeDAO;
    public NewsTypeDAO getNewsTypeDAO () {
        return newsTypeDAO;
    }
    public void setNewsTypeDAO (NewsTypeDAO newsTypeDAO) {
        this.newsTypeDAO = newsTypeDAO;
    }
}

```

代码说明：设定 newsDAO 和 newsTypeDAO 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。

14.9.4 显示控制器类 ShowController.java

该类放在 com.gd.service.action 包里，主要用来负责对页面传来的显示新闻的请求进行转发，这个类继承了 Spring 提供的 SimpleFormController 类。ShowController.java 的示例代码如下：

```

//***** ShowController.java*****
package com.gd.action;

import java.io.IOException;
import java.util.AbstractMap;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

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

import org.apache.log4j.Logger;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.View;
import org.springframework.web.servlet.mvc.Controller;

import com.gd.dao.NewsDAO;
import com.gd.dao.NewsTypeDAO;
import com.gd.po.Newstype;
import com.gd.vo.User;
//实现 Controller 接口，用来负责对页面传来的显示新闻的请求进行转发
public class ShowController implements Controller {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    private String viewPage;
    //实现了 handleRequest()方法，用来负责对页面传来的显示新闻的请求进行转发并返回

```



```
public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
    Map model = new HashMap();
    Map mapNews = new HashMap();
    //获取新闻类别，供页面使用
    List listNewsType = getNewsTypeDAO().queryNewsType();
    //根据新闻类别获取对应的新闻，并将其存储在 Map 中
    for (int i = 0; listNewsType != null && i < listNewsType.size(); i++) {
        Newstype newsType = (Newstype)listNewsType.get(i);
        //根据新闻类别获取对应的新闻
        List listNews = getNewsDAO().getNewsByType(newsType.getId());
        mapNews.put(newsType.getId(), listNews);
    }
    //将信息存储在 model 中，供返回页面使用
    model.put("mapNews", mapNews);
    model.put("listNewsType", listNewsType);
    return new ModelAndView(getViewPage(), model);
}

//依赖注入要返回的页面
public void setViewPage(String viewPage) {
    this.viewPage = viewPage;
}

//获取要返回的页面
public String getViewPage() {
    return viewPage;
}

private NewsDAO newsDAO;
public NewsDAO getNewsDAO () {
    return newsDAO;
}

public void setNewsDAO (NewsDAO newsDAO) {
    this.newsDAO = newsDAO;
}

private NewsTypeDAO newsTypeDAO;
public NewsTypeDAO getNewsTypeDAO () {
    return newsTypeDAO;
}

public void setNewsTypeDAO (NewsTypeDAO newsTypeDAO) {
    this.newsTypeDAO = newsTypeDAO;
}
}
```

代码说明：设定 viewPage、newsDAO 和 newsTypeDAO 的 set 和 get 方法主要是为了使用 Spring 的依赖注入。

14.10 编写辅助类 NewsUtil.java

新建一个包 com.gd.util, 该包下面的类是 myNews 工程的一些辅助类。在此介绍用来获取当前日期的辅助类 NewsUtil, NewsUtil.java 示例代码如下:

```
/** ***** NewsUtil.java ***** */
package com.gd.util;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.List;
//用来获取当前日期, 以后可以添加其他辅助信息
public class NewsUtil {
    /**
     * 得到当前系统日期,格式: yyyy-mm-dd
     *
     * @return String
     */
    public static String getCurrentDate() {
        String currentDate = "";

        SimpleDateFormat format1 = new SimpleDateFormat("yyyy-MM-dd");
        format1.setLenient(false);
        currentDate = format1.format(new Date());

        return currentDate;
    }
    /**
     * 得到当前系统日期,格式: yyyymmdd
     *
     * @return String
     */
    public static String getCurDate() {
        String currentDate = "";

        SimpleDateFormat format1 = new SimpleDateFormat("yyyyMMdd");
        format1.setLenient(false);
        currentDate = format1.format(new Date());

        return currentDate;
    }
}
```

```
* 得到当前时间 (HH:mm:ss)
* @param cal
* @return String
*/
public static synchronized String getCurTime() {
    String pattern = "HHmm";
    return getDateFormat(getCalendar(), pattern);
}

/**
 * 得到当前时间 (HHmm)
 * @param cal
 * @return String
 */
public static synchronized String getCurrentTime() {
    String pattern = "HH:mm:ss";
    return getDateFormat(getCalendar(), pattern);
}

/**
 * @param cal
 * @return String
 */
public static synchronized String getDateFormat(java.util.Calendar cal) {
    String pattern = "yyyy-MM-dd HH:mm:ss";
    return getDateFormat(cal, pattern);
}

/**
 * @param date
 * @return String
 */
public static synchronized String getDateFormat(java.util.Date date) {
    String pattern = "yyyy-MM-dd HH:mm:ss";
    return getDateFormat(date, pattern);
}

/**
 * @param strDate
 * @return java.util.Calendar
 */
public static synchronized Calendar parseCalendarFormat(String strDate) {
    String pattern = "yyyy-MM-dd HH:mm:ss";
    return parseCalendarFormat(strDate, pattern);
}

/**
 * @param strDate
 * @return java.util.Date
 */
```



```
public static synchronized Date parseDateFormat(String strDate) {
    String pattern = "yyyy-MM-dd HH:mm:ss";
    return parseDateFormat(strDate, pattern);
}

/**
 * @param cal
 * @param pattern
 * @return String
 */
public static synchronized String getDateFormat(java.util.Calendar cal,
    String pattern) {
    return getDateFormat(cal.getTime(), pattern);
}

/**
 * 得到当前时间 (HHmm)
 * @param cal
 * @return String
 */
public static synchronized String getCurrentTime(String pattern) {
    return getDateFormat(getCalendar(), pattern);
}

/**
 * @param date
 * @param pattern
 * @return String
 */
public static synchronized String getDateFormat(java.util.Date date,
    String pattern) {
    synchronized (sdf) {
        String str = null;
        sdf.applyPattern(pattern);
        str = sdf.format(date);
        return str;
    }
}

/**
 * 该方法将字符串格式化为标准日期格式
 *
 * @param String
 * @return String
 */
public static String getFormatDate(String time) {
    SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    Date date;
```



```

        String strDate = "";
        try {
            date = df.parse(time);
            df.applyPattern("yyyy-MM-dd");
            strDate = df.format(date);
        } catch (ParseException e) {
        }

        return strDate;
    }

    /**
     * 该方法得到与当天差任意天的格式化时间,
     * offset 表示与当天相差的天数, splitdate 表示日期间的分隔符, splittime 表示时间间的分隔符。
     *
     * @param int
     *         offset
     * @param String
     *         splitdate
     * @param String
     *         splittime
     * @return String
     */
    public static String getPriorDay(int offset, String splitdate,
                                     String splittime) {
        SimpleDateFormat df = new SimpleDateFormat("yyyyMMddHHmmss");
        Calendar theday = Calendar.getInstance();
        theday.add(Calendar.DATE, offset);

        df.applyPattern("yyyy" + splitdate + "MM" + splitdate + "dd" + " "
                        + "HH" + splittime + "mm" + splittime + "ss");

        return df.format(theday.getTime());
    }

    public static synchronized Date parseDateDayFormat(String strDate) {
        String pattern = "yyyy-MM-dd";
        return parseDateFormat(strDate, pattern);
    }

    private static SimpleDateFormat sdf = new SimpleDateFormat();
    public static synchronized Date parseDateFormat(String strDate,
                                                    String pattern) {
        synchronized (sdf) {
            Date date = null;
            sdf.applyPattern(pattern);
            try {
                date = sdf.parse(strDate);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

```

```

        return date;
    }
}
public static synchronized Calendar getCalendar() {
    return GregorianCalendar.getInstance();
}
/**
 * @param strDate
 * @param pattern
 * @return java.util.Calendar
 */
public static synchronized Calendar parseCalendarFormat(String strDate,
    String pattern) {
    synchronized (sdf) {
        Calendar cal = null;
        sdf.applyPattern(pattern);
        try {
            sdf.parse(strDate);
            cal = sdf.getCalendar();
        } catch (Exception e) {
        }
        return cal;
    }
}
}
}

```

代码说明：该辅助类主要是用来显示和转换各种不同格式的日期。

14.11 编写配置文件 dispatcherServlet-servlet.xml

配置文件 dispatcherServlet-servlet.xml 放在 WEB-INF/conf 目录下，包括对 Hibernate 的配置也在其中。dispatcherServlet-servlet.xml 的示例代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    /*定义国际化支持信息*/
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename">
            <value>messages</value>
        </property>
    </bean>
    /*定义映射处理*/
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver"></bean>

```

```
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandler-
Mapping">
    <property name="mappings">
        <props>
            <prop key="login.do">loginController</prop>
            <prop key="regedit.do">regeditController</prop>
            <prop key="release.do">releaseController</prop>
            <prop key="show.do">showController</prop>
        </props>
    </property>
</bean>
/*定义视图显示*/
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
Resolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.InternalResourceView</value>
    </property>
    <property name="prefix">
        <value>/WEB-INF/jsp/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
/*定义控制器*/
<bean id="loginController" class="com.gd.action.LoginController">
    <property name="commandClass">
        <value>com.gd.vo.User</value>
    </property>
    <property name="formView">
        <value>login</value>
    </property>
    <property name="successView">
        <value>release</value>
    </property>
    <property name="newsTypeDAO">
        <ref local="newsTypeDAO" />
    </property>
    <property name="login">
        <ref local="login" />
    </property>
</bean>
<bean id="regeditController" class="com.gd.action.RegeditController">
    <property name="regedit">
        <ref bean="regedit"/>
    </property>
    <property name="commandClass">
        <value>com.gd.vo.User</value>
    </property>
```



```
<property name="formView">
    <value>regedit</value>
</property>
<property name="successView">
    <value>release</value>
</property>
<property name="newsTypeDAO">
    <ref local="newsTypeDAO" />
</property>
</bean>
<bean id="releaseController" class="com.gd.action.ReleaseController">
    <property name="commandClass">
        <value>com.gd.vo.News</value>
    </property>
    <property name="formView">
        <value>Realse</value>
    </property>
    <property name="successView">
        <value>show</value>
    </property>
    <property name="release">
        <ref bean="release"/>
    </property>
<property name="newsDAO">
    <ref local="newsDAO" />
</property>
<property name="newsTypeDAO">
    <ref local="newsTypeDAO" />
</property>
</bean>
<bean id="showController" class="com.gd.action.ShowController">
    <property name="viewPage">
        <value>show</value>
    </property>
    <property name="newsDAO">
        <ref local="newsDAO" />
    </property>
    <property name="newsTypeDAO">
        <ref local="newsTypeDAO" />
    </property>
</bean>
/*实现依赖注入*/
<bean id="regedit" class="com.gd.service.impl.RegeditImpl">
    <property name="userDao">
        <ref local="userDAO" />
    </property>
</bean>
<bean id="release" class="com.gd.service.impl.ReleasImpl">
    <property name="newsDao">
        <ref local="newsDAO" />
```



```
</property>
  <property name="login">
    <ref local="login" />
  </property>
</bean>

<bean id="login" class="com.gd.service.impl.LoginImpl">
  <property name="userDAO">
    <ref local="userDAO" />
  </property>
</bean>
/*定义数据源*/
<bean id="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost/myNews</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
  <property name="password">
    <value>root</value>
  </property>
</bean>
/*定义 sessionFactory */
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref local="dataSource" />
  </property>
  <property name="mappingResources">
    <list>
      <value>com\gd\po\User.hbm.xml</value>
      <value>com\gd\po\New.hbm.xml</value>
      <value>com\gd\po\Newstype.hbm.xml</value>
      <value>com\gd\po\Userauthor.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
</bean>
```

```

/*定义事务处理 */
<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
</bean>
/*定义事务处理的代理 */
<bean id="userDAOProxy"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="target">
        <ref local="userDAO" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
            <prop key="query*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
/*定义各种 DAO */
<bean id="userDAO" class="com.gd.dao.impl.UserDAOImpl">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
</bean>
<bean id="newsDAO" class="com.gd.dao.impl.NewsDAOImpl">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
</bean>
<bean id="newsTypeDAO" class="com.gd.dao.impl.NewsTypeDAOImpl">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
</bean>
/*定义异常处理机制*/
<bean
    id="exceptionResolver"
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="java.sql.SQLException">errpr</prop>
            <prop key="java.sql.IOException">error</prop>
        </props>
    </property>

```

```
</bean>
</beans>
```

至此编码结束，myNews 最终的目录结构如图 14.50 所示。

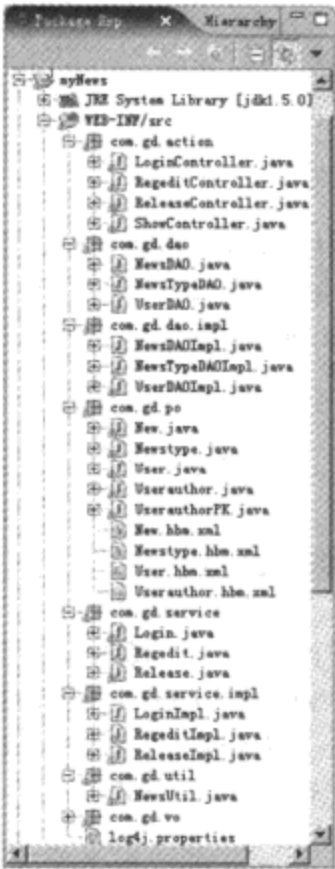


图 14.50 myNews 最终的目录结构

14.12 运行验证程序

(1) 首先通过数据库在 newsType 表中插入新闻类别的数据：新闻中心、规章制度、会议纪要、公司文件，如图 14.51 所示。

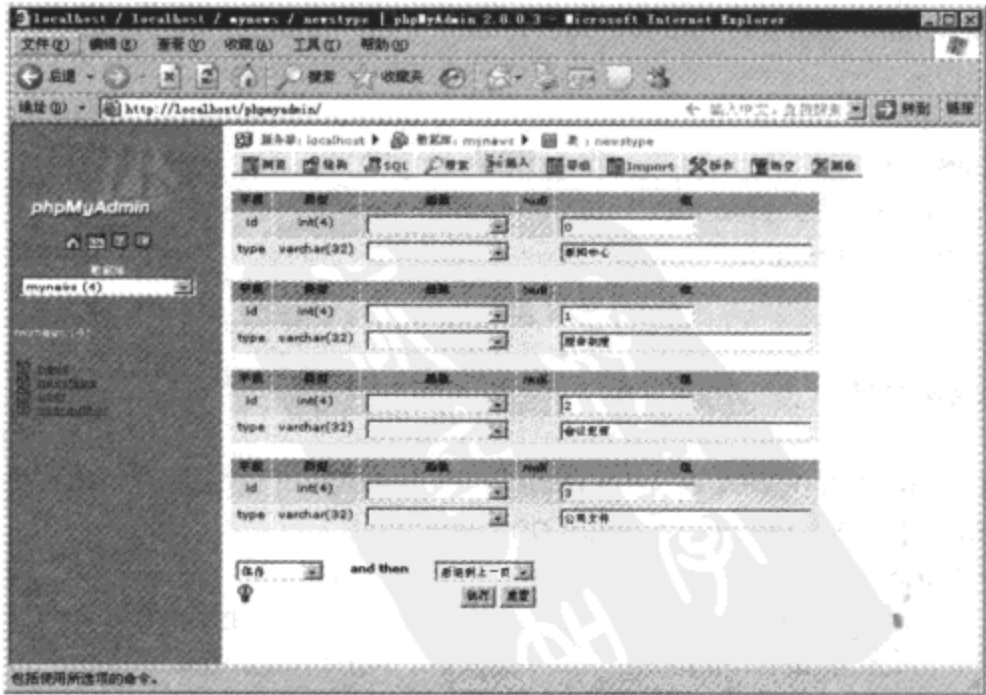


图 14.51 插入新闻类别的数据

(2) 单击“执行”按钮，即可向 newsType 表插入数据。

(3) 启动 Tomcat，在地址栏中输入“http://localhost:8080/myNews/regedit.do”，即可看到用户注册的页面，如图 14.52 所示。

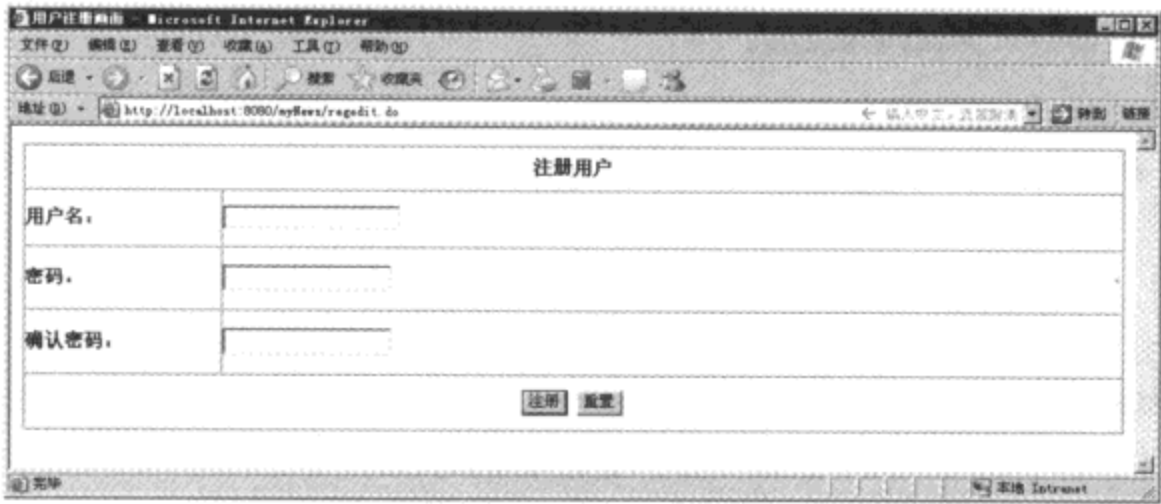


图 14.52 用户注册的页面

(4) 输入用户名“gd”、密码“12345”、确认密码“12345”，然后单击“注册”按钮，如果注册成功则会转入发布新闻的页面，如图 14.53 所示。

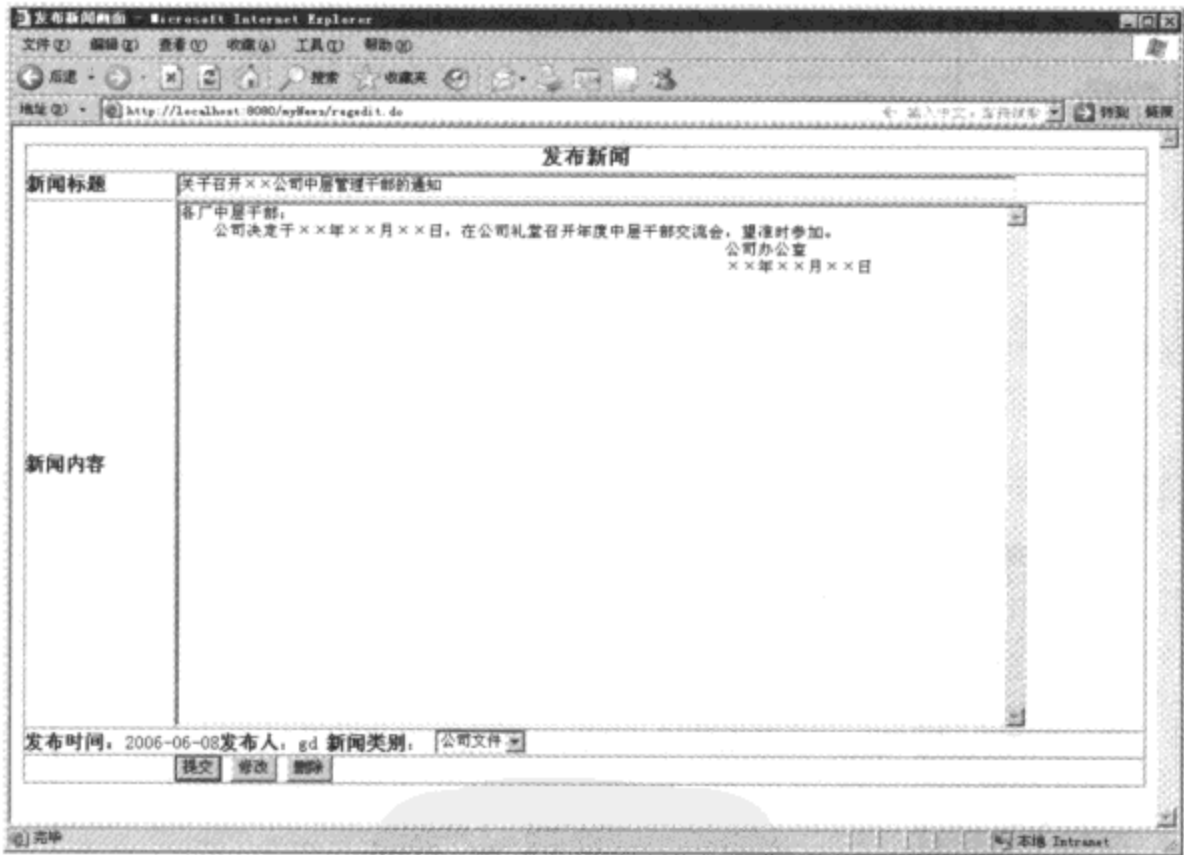


图 14.53 注册成功则会转入发布新闻的页面

(5) 输入新闻标题和内容，选择新闻类别，然后单击“提交”按钮，如果新闻发布成功，则会返回到新闻展示的页面，如图 14.54 所示。

(6) 在地址栏中输入“http://localhost:8080/myNews/login.do”，则会出现管理员登录页面，如图 14.55 所示。

(7) 输入刚才注册成功的用户名“gd”、密码“12345”、确认密码“12345”，然后单击“提交”按钮，如果验证成功则会转入发布新闻的页面。

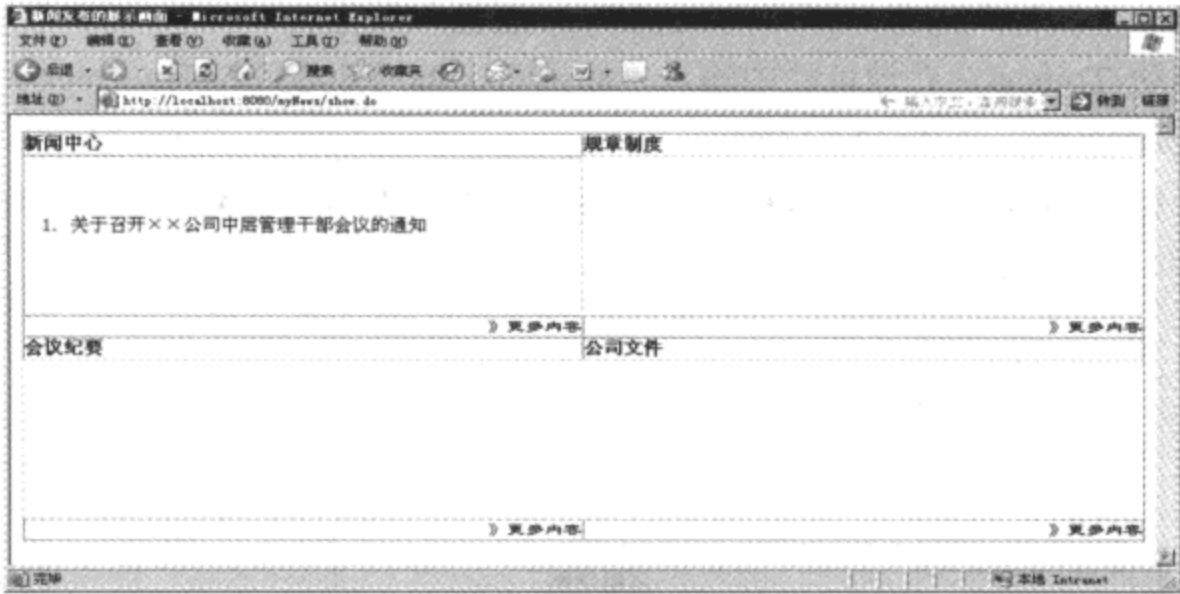


图 14.54 新闻展示页面

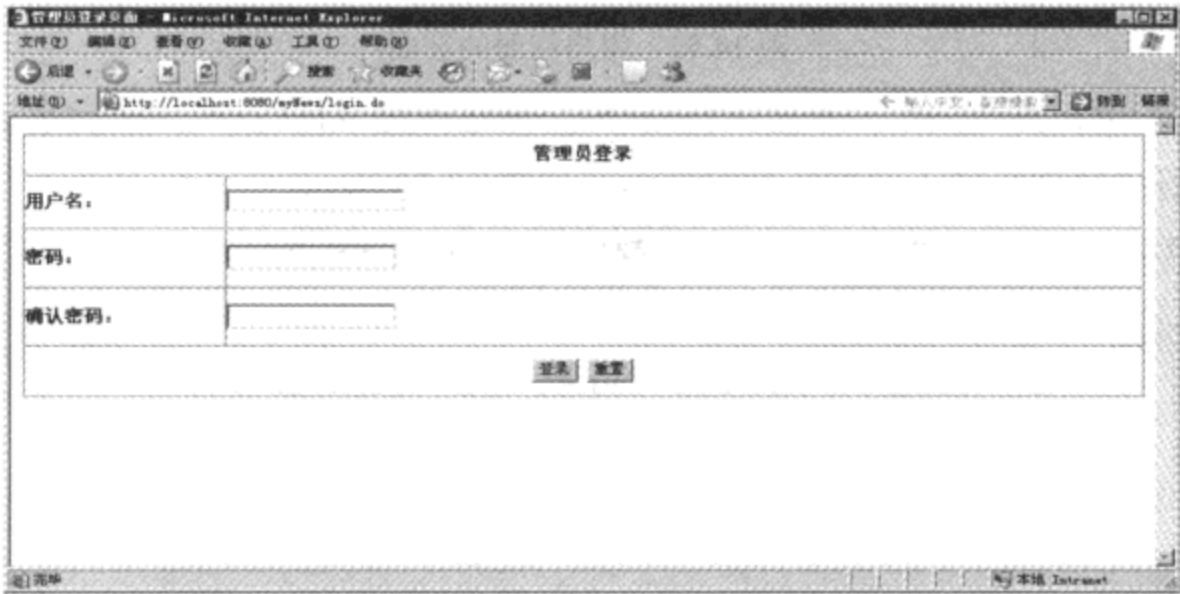


图 14.55 管理员登录页面

(8) 查看数据库即可看到刚才注册用户成功的数据，如图 14.56 所示。

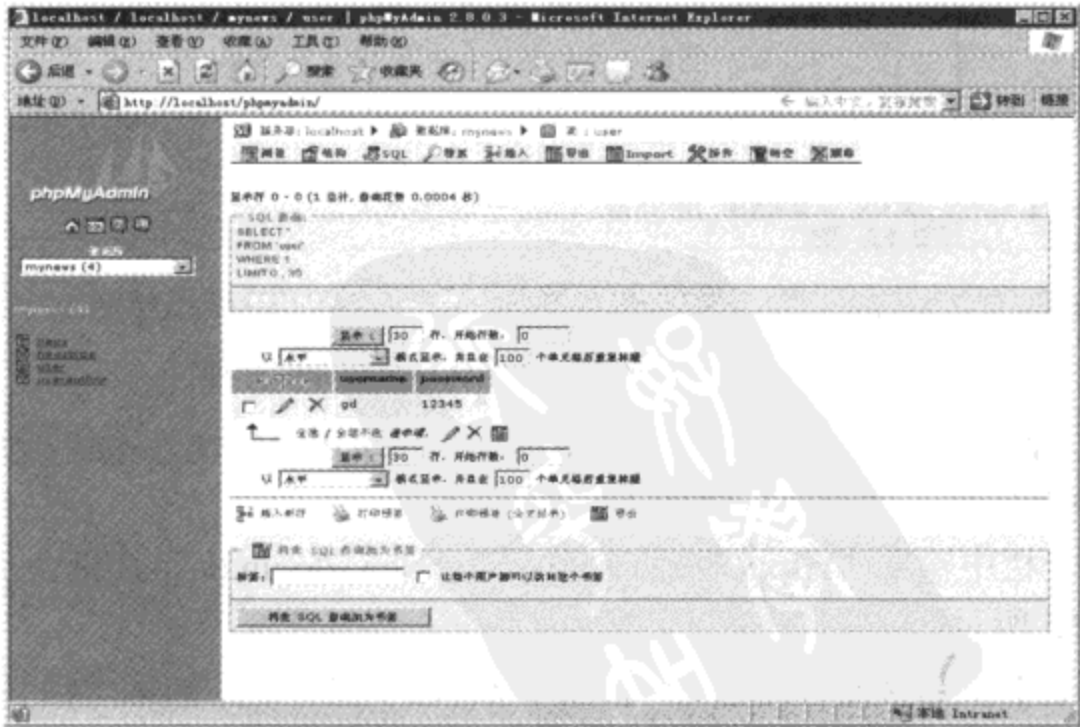


图 14.56 刚才注册用户成功的数据

(9) 同样查看数据库即可看到刚才发布新闻成功的数据，如图 14.57 所示。

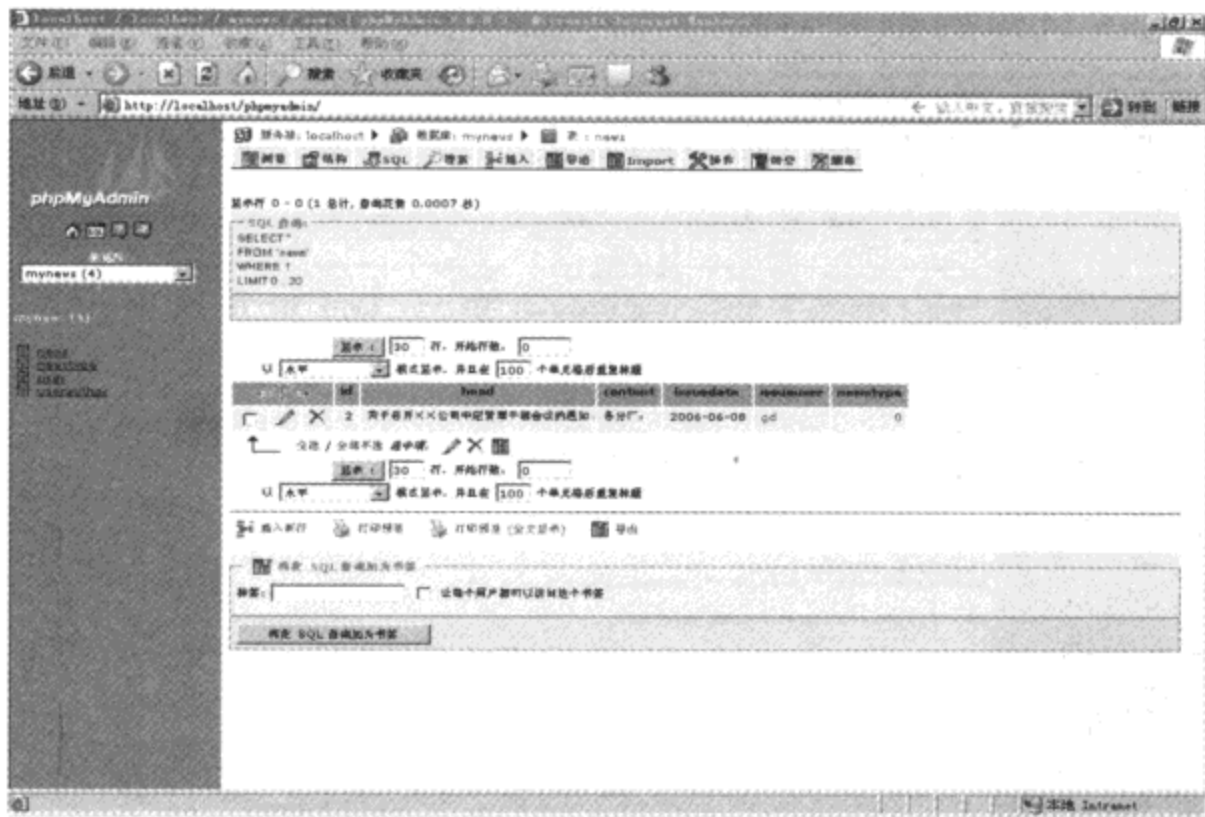


图 14.57 刚才发布新闻成功的数据

注意：在一些计算机里可能会出现乱码问题，这时可以把数据库中存放中文的字段属性改为 utf，即可在 MySQL 中存放中文而不会造成乱码；如果页面中的文字出现乱码，最好的方法就是实现国际化，读者可以根据前面介绍的方法尝试把上面的示例改为支持国际化。

(10) 把数据库中存放中文的字段属性改为 utf 的方法：首先单击 MySQL 管理画面左边的 news 数据库表，然后单击结构，出现 news 表的结构图，如图 14.58 所示。

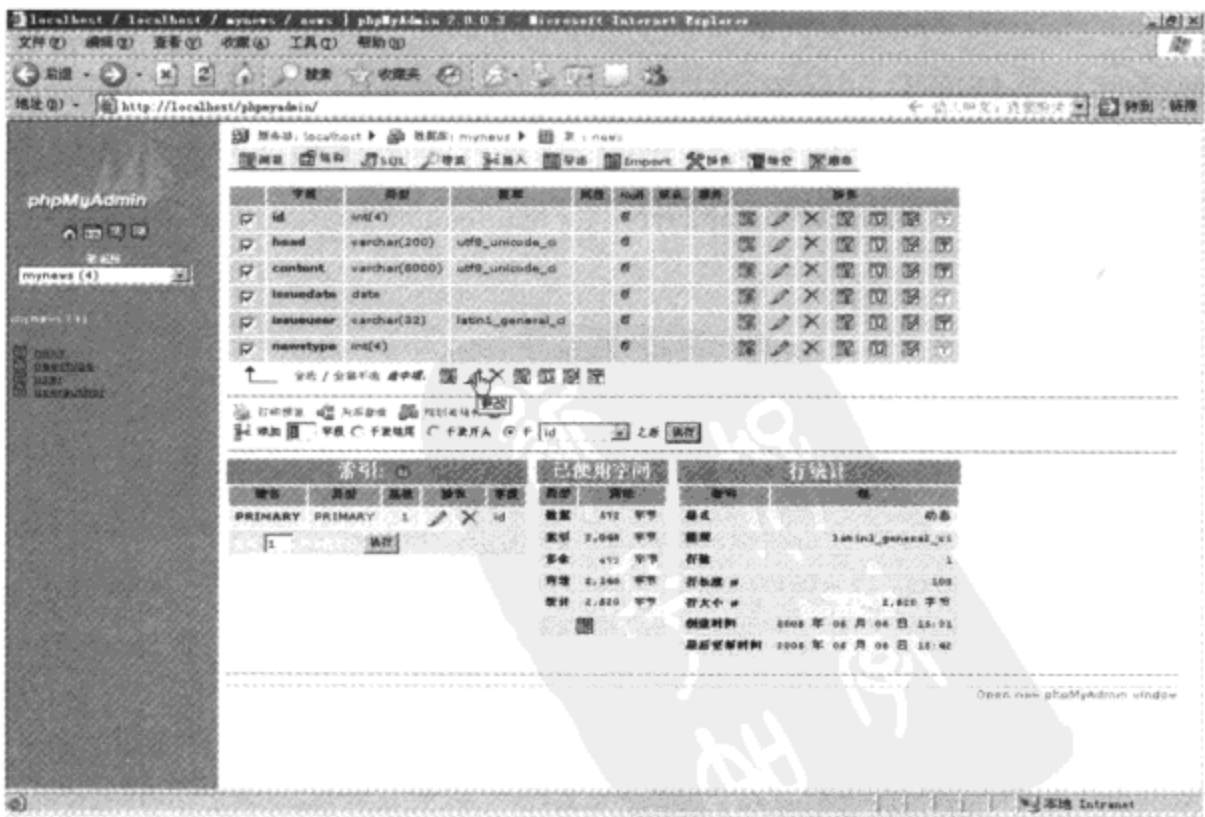


图 14.58 news 表的结构图

(11) 然后在 news 表的结构图中全选所有字段，单击更改图标，将出现更改 news 表结构的画面，如图 14.59 所示。

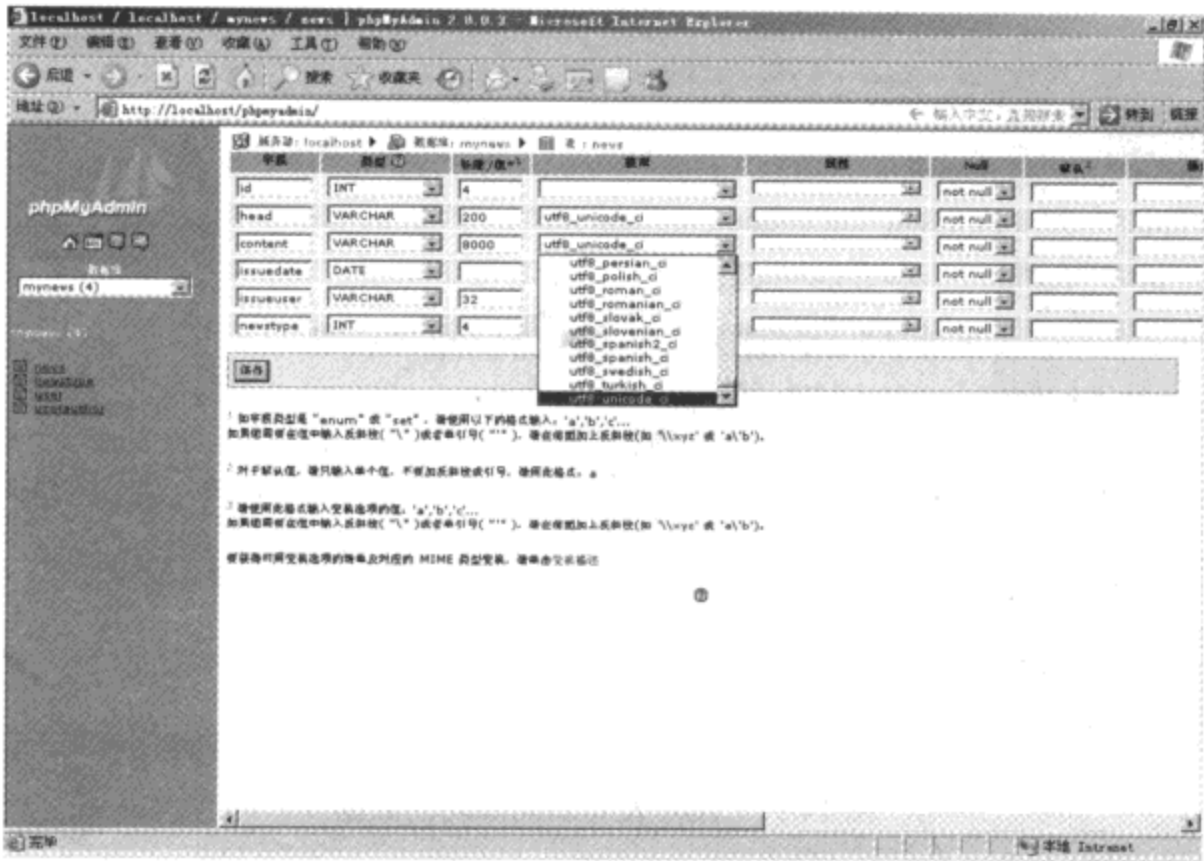


图 14.59 更改 news 表结构的画面

(12) 在需要更改存储方式的字段属性中，在整理属性下面的下拉菜单中选择 utf，然后单击“保存”按钮即可。

14.13 小 结

本章主要通过一个新闻发布系统的实例，从头到尾讲解了 Spring 和 Hibernate 的使用及配置方法。这个实例虽然简单，但有关 Spring 的方方面面基本都涉及到了，有兴趣的读者可以对此实例进行扩充，以满足实际的应用。



有关此电子图书的说明

本人由于一些便利条件,可以帮您提供各种中文电子图书资料,且质量均为清晰的 PDF 图片格式,质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新,文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书,我都可以帮您找到电子版本。所以,当你想要看什么图书时,可以联系我。我的 QQ 是: 85013855,大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作,请各位爱书之人尊重个人劳动,敬请您不要修改此 PDF 文件。因为这些图书都是有版权的,请各位怜惜电子图书资源,不要随意传播,否则,这些资源更难以得到。