Broadview
www.broadview.com.cn

Java技术大系

Spring 2.0 核心技术与最佳实践

廖雪峰 编著

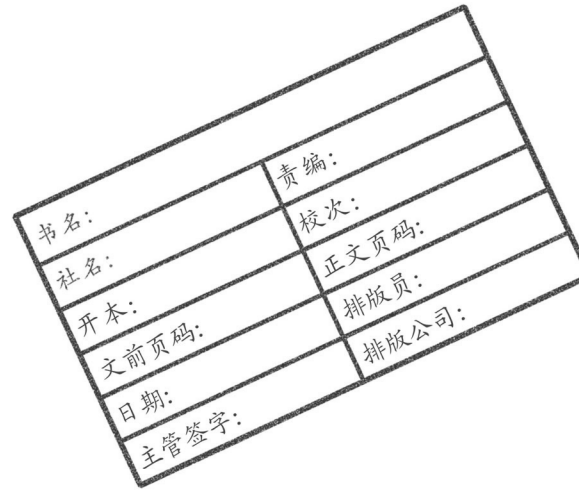
本书的全部源代码和Eclipse的视频教程 1.6GB DVD

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

Java 技术大系

Spring 2.0 核心技术与最佳实践

廖雪峰　编著



電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书注重实践而又深入理论，由浅入深且详细介绍了 Spring 2.0 框架的几乎全部的内容，并重点突出 2.0 版本的新特性。本书将为读者展示如何应用 Spring 2.0 框架创建灵活高效的 JavaEE 应用，并提供了一个真正可直接部署的完整的 Web 应用程序——Live 在线书店。

在介绍 Spring 框架的同时，本书还介绍了与 Spring 相关的大量第三方框架，涉及领域全面，实用性强。本书另一大特色是实用性强，易于上手，以实际项目为出发点，介绍项目开发中应遵循的最佳开发模式。

本书还介绍了大量实践性极强的例子，并给出了完整的配置步骤，几乎覆盖了 Spring 2.0 版本的新特性。

本书适合有一定 Java 基础的读者，对 JavaEE 开发人员特别有帮助。本书既可以作为 Spring 2.0 的学习指南，也可以作为实际项目开发的参考手册。

图书在版编目（CIP）数据

Spring 2.0 核心技术与最佳实践 / 廖雪峰编著.—北京：电子工业出版社，2007.5
（Java 技术大系）
ISBN 978-7-121-04262-1

I. S… Ⅱ. 廖… Ⅲ. JAVA 语言－程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2007）第 056302 号

责任编辑：韩 明
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：33.75 字数：681 千字
印 次：2007 年 4 月第 1 次印刷
印 数： 册 定价： 元（含光盘 1 张）

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

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

内容简介

Spring 正在成为最流行的轻量级 JavaEE 框架，其优秀的设计和强大的功能吸引了越来越多的 JavaEE 开发人员，最新的 Spring 2.0 版本更是增添了许多激动人心的功能。

本书注重实践而又深入理论，由浅入深且详细介绍了 Spring 2.0 框架的几乎全部的内容，并重点突出 2.0 版本的新特性。本书将为读者展示如何应用 Spring 2.0 框架创建灵活高效的 JavaEE 应用，并提供了一个真正可直接部署的完整的 Web 应用程序——Live 在线书店，读者可以从中学习设计并实现一个基于 Spring 2.0 的完整的多层 JavaEE 应用程序的方法，积累大量有用的经验和技巧。该演示程序可以通过 http://www. livebookstore.net 访问。

本书分为三大部分，第一部分从整体介绍 Spring 2.0 框架的设计和结构，并演示如何搭建开发环境和第一个 Spring 应用程序；第二部分由浅入深地分别介绍 Spring 框架的各个模块的功能，重点突出 2.0 版本的新特性；第三部分详细介绍了基于 Spring 2.0 框架的 Live 在线书店应用程序的设计和实现，集成了大量的第三方开源框架，具有极强的实践性。

在介绍 Spring 框架的同时，本书还介绍了与 Spring 相关的大量第三方框架，涉及领域全面，实用性强。例如，作为 ORM 框架的 Hibernate、iBatis、JDO 和最新的 JPA；作为 Web 框架或组件的 Struts、WebWork2、Velocity、FreeMarker、Tiles、JSF；作为 Web 服务框架的 Axis 和 XFire；作为安全框架的 Acegi 等。本书对每个框架的集成都给出了完整的 Eclipse 工程，这些示例可以直接作为基本的配置模型并应用到实际项目中。

本书另一大特色是实用性强，易于上手，以实际项目为出发点，介绍项目开发中应遵循的最佳开发模式。例如，在开发 Web 服务时，不是从编写复杂的 WSDL 文件入手，而是首先设计接口，然后采用 Java 5 注解来实现 Web 服务的自动部署。在集成 Hibernate 时，不是从编写配置文件或创建数据库结构入手，而是首先设计 Java 实体对象，然后通过 Java 5 注解并配合 Ant 自动完成数据库表的创建，这些都符合实际项目的开发。本书还提倡应用 Ant 并配合 XDoclet 实现配置文件的自动生成，减少项目的维护工作量。

本书还介绍了大量实践性极强的例子，并给出了完整的配置步骤，例如，基于泛型的 DAO 模型，结合 Lucene 和 Compass 实现全文搜索功能，利用 CAS 架设单点登录服务器，利用 JMX 实现对应用程序的远程监控，利用 Filter 实现无侵入的页面缓存等。

本书几乎覆盖了 Spring 2.0 版本的新特性，包括使用 AspectJ 5 注解实现 AOP、对 JPA 的完整支持、新的声明式事务配置方式、对动态语言的支持等。在选择某种解决方案时，优先考虑采用 Spring 2.0 的新特性并尽量使用 Java 5 注解进行配置，这也是本书有别于其他介绍 Spring 1.x 书籍的地方。

本书适合有一定 Java 基础的读者，对 JAVEEE 开发人员特别有帮助。本书既可以作为 Spring 2.0 的学习指南，也可以作为实际项目开发的参考手册。

前　言

Java 开发已经走过十年了！随着因特网的飞速发展，Java 技术获得了前所未有的广泛应用。从桌面系统到企业应用，从手机到智能卡，处处都能看到它的身影。从 1998 年 Sun 公司发布 JavaEE 1.0 版本开始，在此后的短短几年内，JaveEE 获得了巨大的发展，几乎成为企业开发的代名词。

作为 JavaEE 中最核心的 EJB 技术，也一度成为 JavaEE 应用的核心。不幸的是，EJB 在带来了全新的企业级开发模型的同时，也带来了不必要的复杂性：复杂的接口，难于测试和部署。越来越多的开发人员不断反思 EJB 开发的复杂性，并试图以更简单的 Java 技术来简化 JavaEE 应用的开发。Rod Johnson 总结了他数年的 JavaEE 项目经验，在《Expert-One-on-One: JaveEE Design & Development》一书中详细阐述了 EJB 带来的复杂性，并提出了一系列以轻量级框架为核心的全新的 JavaEE 设计思想，阐述了如何组合一系列现有的技术并形成了一个初步的框架，这个框架后来便发展为 Spring Framework。通过 Spring 这个轻量级框架，我们终于可以轻松实现过去必须使用复杂而烦琐的 EJB 才能实现的功能。

Spring 提出了以 JavaBean 为组件模型、针对接口编程、使用依赖注入等许多优秀的设计思想，并且 Spring 可以无缝整合许多流行的框架，如 Struts、Hibernate 等。人们很快意识到以 Spring 框架为基础来开发 JavaEE 应用程序可以大大简化应用程序的设计、调试和部署，并得到一个松散耦合的系统架构。因此，Spring 得到了越来越广泛的应用。随着 Spring 2.0 版本的推出，添加了大量新的功能，进一步强化了 Spring 框架在轻量级 JavaEE 开发领域的主导地位。

本书特色

本书以 Spring 2.0 版本为标准，试图向读者展示 Spring 框架的奥秘，引导读者由浅入深、一步一步地掌握 Spring 框架的使用方法和设计思想。此外，本书还特别注重实践，力图给出能够在实际项目中应用的解决方案，并给出完整的示例代码。在本书的最后一章中，还详细介绍了基于 Spring 2.0 框架设计并实现的一个完整的 Web 应用程序——Live 在线书店，并给出了许多有用的设计模式和技巧。

在介绍 Spring 框架的同时，本书也试图介绍与 Spring 相关的大量第三方框架，涉及领域全面，实用性强。例如，作为 ORM 框架的 Hibernate、iBatis、JDO 及最新的 JPA；作为 Web 框架或组件的 Struts、WebWork2、Velocity、FreeMarker、Tiles、JSF；作为 Web 服务框架的 Axis 和 XFire；作为安全框架的 Acegi 等。本书对每个框架的集成都给

出了完整的 Eclipse 工程，这些示例可以直接作为基本的配置模型并应用到实际项目中。

本书另一大特色是实用性强，以实际项目为出发点，介绍项目开发中应遵循的最佳开发模式。例如，在开发 Web 服务时，不是从编写复杂的 WSDL 文件入手，而是首先设计接口，然后采用 Java 5 注解来实现 Web 服务的自动部署；在集成 Hibernate 时，不是从编写配置文件或创建数据库结构入手，而是首先设计 Java 实体对象，然后通过 Java 5 注解并配合 Ant 自动完成数据库表的创建，这些都符合实际项目的开发。

本书还介绍了大量实践性极强的例子，并给出了完整的配置步骤，例如，基于泛型的 DAO 体系设计，结合 Lucene 和 Compass 实现全文搜索功能，利用 CAS 架设单点登录服务器，利用 JMX 实现对应用程序的远程监控，利用 Filter 实现无侵入的页面缓存等，这些都是在实际项目开发中经常需要用的，本书均给出了能够直接运行的配置，并配合屏幕截图尽量详细地给出配置步骤，能够最大限度地让初学者无痛起步。

本书的代码注释也非常详细，并且在书中尽量采用中文注释，便于初学者理解。对于许多复杂的模块设计，本书总是给出流程图或关系图，让读者从设计上能更好地整体把握。

本书还几乎覆盖了 Spring 2.0 版本的新特性，包括使用 AspectJ 5 注解实现 AOP、对 JPA 的完整支持、新的声明式事务配置方式、对动态语言的支持等。在选择某种解决方案时，优先考虑采用 Spring 2.0 的新特性并尽量使用 Java 5 注解进行配置，这也是本书有别于其他介绍 Spring 1.x 书籍的地方。

主要内容

本书按照由浅入深、从理论到实践的顺序来安排内容，主要包括以下内容。

第一部分：介绍 Spring 的诞生和主要功能，并指导读者在 Eclipse 中编写一个具体的 Spring 应用程序，以便读者能对 Spring 有一个初步认识。

第二部分：分别介绍 Spring 的各主要功能模块，按照由浅到深及各模块的依赖关系，首先介绍作为整个 Spring 框架核心基石的 IoC 容器，然后分别介绍 Spring 的 AOP 支持、数据访问策略、事务管理及 Web MVC 模块。紧接着介绍 Spring 框架的一些非核心但是可能在实际项目中应用的模块，包括远程访问、任务调度、JMS 支持、JMX 支持、动态语言支持及 Acegi 安全框架，读者可以根据实际需要有选择地学习。通过第二部分的介绍，读者能全面掌握 Spring 框架的几乎所有内容。

第三部分：开发一个完整的基于 Spring 框架的 Web 应用程序——Live 在线书店。这一部分详细介绍了如何利用 Spring 设计并实现一个多层 JaveEE 应用程序。在项目开发中，大量应用实际项目的开发方式，包括使用 Ant 作为构建工具，使用 XDoclet 自动生成配置文件等。在 Live 在线书店的实现细节上，还介绍了许多有用的模式和技巧，包括内存和静态文件的缓存模型、一些 JavaScript 技巧、应用 JMX 检测系统性能等。读者

完全可以将其应用到实际的项目开发中。

需要注意的是，本书中的图例并不是完全按照 UML 标准绘制的，图例的设计目的是为了突出问题并试图以最清晰的方式展示给读者，因此，读者不必有 UML 相关知识，只需明白图例的意义即可。

读者对象

本书适合有一定 Java 基础的读者，对 JaveEE 开发人员更是特别有帮助。本书既可以作为 Spring 2.0 的学习指南，也可以作为实际项目开发的参考手册。

本书不仅希望读者能掌握 Spring 框架的使用方法，还试图阐述 Spring 框架的实现原理，因此，许多章节都会涉及一些底层实现，例如，AOP 和 MVC 的手动实现方法。不理解这些原理虽然也不会影响 Spring 的学习，但是却失去了了解 Spring 框架底层运行机制的机会，也就无从学习 Spring 框架的设计思想。因此，强烈建议读者在掌握了 Spring 框架的使用方法后，更深入到 Spring 框架内部，最好能结合 Spring 源代码学习 Spring 的设计思想。如果在脱离 Spring 的环境下也能自然而然地应用 Spring 优秀的设计思想，例如，始终坚持针对接口编程，使用依赖注入，那才算真正掌握了 Spring 框架的精髓。

在本书的写作过程中，得到了家人和朋友的大力支持。在此，我要特别感谢我的妻子对我的大力支持，我还要感谢同事李江华，他为本书第 5 章的示例编写了 Swing 界面，最后，我还要感谢为本书提出宝贵意见的朋友和同事。

廖雪峰

2007.2.14

作者简介

廖雪峰，具有 5 年Java/J2EE/J2ME开发经验，早在大学本科时就参与了网易网上商城（http://mall.163.com）的开发，目前在摩托罗拉软件集团担任软件工程师，从事基于Eclipse的可视化建模工具的设计和开发。

目前，廖雪峰创建了国内讨论JavaEE技术的专业网站：JavaEE开发网（http://www.javaeedev.com），读者可以在JavaEE开发网的论坛中对本书提出中肯的批评和意见，作者将尽最大努力回复读者提出的问题。

作者的Blog地址是http://xuefeng.javaeedev.com，欢迎访问。

本书的勘误表也将在 JavaEE 开发网论坛中不定期发布，请读者关注。

目　录　CONTENTS

第 1 章

初识Spring

1.1 JavaEE平台的诞生和发展

Java 语言从诞生之日起，就受到了广泛关注。这个崭新的语言拥有许多优秀的特性：平台无关、垃圾回收机制、抛弃了 C/C++的指针、强大的网络处理功能和完全面向对象的开发模型。虽然许多特性并非由 Java 首次实现，然而，Java 诞生之日正是因特网开始发展之时。人们很快意识到 Java 语言的这些优秀特性非常适合快速开发基于因特网的健壮的分布式应用，尽管 Java 设计之初的本意是针对嵌入式设备。

今天，越来越多的企业开发人员希望能快速开发安全可靠的、可扩展的分布式企业应用，尤其是以浏览器为前端的 Web 应用，并借助因特网将服务尤其是电子商务扩展到全世界的范围。和过去的客户端/服务器（Client/Server）模式相比，基于浏览器/服务器（Browser/Server）模式的 B/S 应用越来越广泛。随着企业应用规模的快速增长，越来越多的企业将 JavaEE 平台作为企业开发的基础。短短的几年时间里，JavaEE 几乎成了企业开发的代名词。

作为 Java 语言的缔造者，Sun 公司也很快意识到企业对于 Java 的巨大需求，因此，在 1999 年底，一个基于标准 Java 虚拟机的企业级 Java 平台 J2EE——Java 2 Enterprise Edition 正式发布了。

随着 J2EE 1.5 标准的发布，Sun 将 J2EE 正式更名为 JavaEE，与此对应，J2SE 和 JavaME 平台也更名为 JavaSE 和 JavaME。考虑到 JavaEE 这一名称使用越来越广泛，在本书中，一律使用“JavaEE”这一术语，而不再使用“J2EE”这一名称，请读者注意。

值得注意的是，JavaEE 并非是一个产品，而是一系列技术和标准的集合。具体的 JavaEE 平台产品由各厂商实现并遵循同一个标准。JavaEE 平台继承了 Java 语言的安全性和高可移植性，为企业应用的设计、开发、部署和管理提供了一套完善的解决方案，它包括了从前端 Web 界面到中间件，再到后端数据库系统的一系列技术和规范。JavaEE 提供了一套标准的 API 和以组件为基础的企业架构，尤其值得注意的是，JavaEE 提出了一个新的“容器”的概念，通过容器来提供标准的系统底层服务，大大降低了企业级开发的复杂度。

概括来讲，JavaEE 包括了一系列针对组件和服务的平台标准和 API 接口，如图 1-1 所示。

多层的分布式架构是 JavaEE 的典型设计模型。通过将系统分割成多层，以便降低各层的复杂性，并实现一个松耦合的结构以便于扩展和维护。典型的 JavaEE 应用是一个三层结构的系统：表示层、业务层和持久层。表示层负责和用户打交道，接受用户输入并将结果显示给用户；业务层负责实现各种商业逻辑，即企业的业务模型；持久层则负责

将业务层的数据保存到永久的外部存储系统中，或者读取数据以供业务层使用。通常，最常用的数据存储系统是关系数据库，因此，持久层最重要的功能便是完成对象/关系映射（O/R Mapping）的实现，即实现系统中 Java 对象和数据库表记录的双向转换。

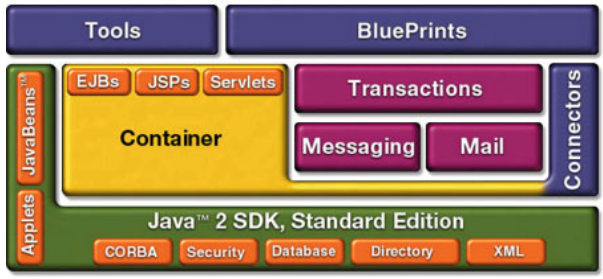


图 1-1

1.2 Spring的起源

在 JavaEE 平台诞生后的日子里，JavaEE 几乎就是企业级开发的代名词，当然，作为 JavaEE 中最核心的 EJB 技术，也一度成为 JavaEE 应用的核心。EJB 第一次提出了声明性事务管理的概念，通过在部署时指定 EJB 组件的声明性事务类型，事务管理被纳入了容器的功能，而不必由开发者来编写冗长的、不易维护的事务代码。这种事务管理模型大大简化了企业应用的开发，也被视为 EJB 最成功的特性。

不幸的是，EJB 在带来了全新的企业级开发模型的同时，也带来了不必要的复杂性：编写 EJB 组件是复杂而困难的。为了实现一个 EJB 组件，除了 Bean 本身的实现类外，还不得不编写额外的 Home 接口和 Remote 接口。大多数时候，为了避免 EJB 远程调用的开销，还常常需要编写 Local 接口。

为每个 EJB 都编写如此复杂的接口需要很多工作量。此外，部署和测试也是令人非常头疼的问题。随着测试驱动开发的流行，人们逐渐意识到单元测试的重要性。实际上，容易被测试的代码往往意味着更容易被维护，也更容易扩展。不幸的是，和普通的 Java 对象相比，EJB 组件更难于测试，因为简单的编码经由测试的开发循环变成了编码，然后进行部署和测试。每次测试时还需要启动 JavaEE 服务器，这中间又加入了一个“等待”的过程。

越来越多的开发人员不断反思 EJB 开发的复杂性，并试图以更简单的 Java 技术来简化 JavaEE 应用的开发。Rod Johnson 总结了他数年的 JavaEE 项目经验，在《Expert-One-on-One: JavaEE Design & Development》一书中详细阐述了 JavaEE 体系尤其是 EJB 带来的复杂性，并提出了一系列以轻量级框架为核心的全新的 JavaEE 设计思想，阐述了如何

组合一系列现有的技术并形成了一个初步的框架，这个框架后来便发展为 Spring Framework。通过 Spring 这个轻量级框架，我们终于可以轻松实现过去必须使用复杂而烦琐的 EJB 才能实现的功能。更重要的是，Spring 抛弃了 EJB 这种重量级组件，以 JavaBean 作为组件实现的一个轻量级框架。由于不再需要 EJB，基于 Spring 的轻量级 JavaEE 应用完全可以抛弃 EJB 服务器，而仅仅需要 Web 服务器即可。更重要的是，以 JavaBean 作为组件模型使得组件的开发和测试得到了极大的简化，而且 Spring 是一个耦合极为松散的框架。

1.3 Spring框架介绍

简单地说，Spring 就是一个实现了 AOP 功能的 IoC 容器，虽然这种说法不太全面，但确实概括了 Spring 框架的核心功能。图 1-2 显示了 Spring 框架包含的 7 个主要模块。

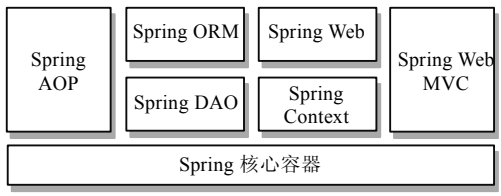


图 1-2

整个 Spring 框架都是轻量级的。和 EJB 这种重量级的组件需要在重量级的 EJB 容器中运行相比，Spring 为普通的 JavaBean 提供了一个轻量级的容器，因此，不需要全功能的 JavaEE 服务器，Spring 容器可以运行在仅支持 Web 容器的 JavaEE 服务器上，或者直接在普通的 main()方法中启动它。

1.3.1 Spring的核心IoC容器

如图 1-2 所示，Spring 核心容器是所有其他模块的基础。这个核心的 IoC 容器定义了如何创建、管理和配置 Bean。在一个 Spring 应用程序中，几乎所有的组件都被放到核心 IoC 容器中，并按照某种配置装配起来。作为开发人员，我们需要关心的是组件的编写，以及如何在 Spring 的 IoC 容器中正确地装配它们。

Spring 通过 Bean 工厂来实现基于依赖注入的组件装配。在第 3 章中，将详细介绍如何使用 Spring 的 IoC 容器来装配出应用程序，从而获得强大的灵活性和可维护性。

Spring 提供的应用程序上下文（Application Context）封装了许多基本的系统服务，例如，访问 JNDI、对国际化（I18N）的支持、事件传播、资源装载、电子邮件服务等。

1.3.2 Spring对AOP的支持

AOP（Aspect Oriented Programming）即面向切面编程，是近年越来越流行的一种新的编程模式。AOP 的编程思想和 OOP 不同，它是对 OOP 的一种强有力的补充。通过 AOP，在某些情况下能够实现更好的模块化结构，或者可以动态为系统增加新的功能，而不影响原有系统的结构。

Spring 的 AOP 模块提供了 AOP 联盟定义的 AOP 接口的实现，利用 Spring 提供的 AOP 支持，可以简化代码逻辑，分离应用程序关注点。此外，Spring 提供的许多底层服务，例如，对声明式事务管理的支持也是基于 AOP 实现的。

和其他 AOP 框架有所不同，Spring 的 AOP 哲学仍建立在 IoC 之上，这意味着 AOP 也是以 Bean 的方式在 Spring 的 IoC 容器中装配出来的。

1.3.3 Spring对数据访问的封装

Spring DAO 定义了一个访问数据库的一致的接口，对 JDBC 的模版化封装大大简化了 JDBC 代码的编写，并且可解析不同数据库厂商的特定错误代码。使用 Spring DAO 的目的之一就是为了隔离应用程序的业务逻辑和数据访问逻辑，从而获得较高的可扩展性。

对象/关系映射 ORM（Object/Relational Mapping）模块封装了多种 ORM 解决方案。但是，Spring 自身并没有提供任何 ORM 方案，相反，Spring 的 ORM 是为了集成许多流行的 ORM 框架而设计的，例如，Hibernate、iBatis、TopLink、JDO 等。通过 Spring 提供的集成方案，可以非常方便地将其纳入到 Spring 的事务管理中。

1.3.4 Spring的声明式事务

传统的 EJB 开发人员能够从 EJB 中获得的最强大的功能便是声明式事务管理。通过指定事务的属性，开发人员就不必手动编写复杂的事务管理代码，而将事务交给 EJB 容器处理，并且能够获得最高的可移植性。现在，Spring 也提供了完全可媲美 EJB 的声明式事务管理。和 EJB 不同，Spring 的声明式事务管理是建立在轻量级的 AOP 基础之上的，却提供了一致的事务模型。熟悉 EJB 声明式事务管理的开发人员可以立刻转到 Spring 提供的声明式事务管理，这也是整个 Spring 框架最具特色的功能之一。

1.3.5 Spring的Web MVC框架

Spring 的 Web 模块提供了一系列针对 Web 开发的基础功能，例如，文件上传、自动绑定参数等。这个模块还用于集成其他的 Web 框架，例如，Struts。

除了能与现有的多种流行 Web 框架集成之外，Spring 本身提供了一个全功能的 MVC（Model-View-Controller）框架。与其他 Web 框架相比，Spring 的 MVC 框架设计得更加适合 Spring 的 IoC 容器，并且具有极大的灵活性。此外，Spring 的 MVC 框架能无缝集成多种可替代 JSP 的视图技术，例如，Velocity 和 Freemaker。

以上仅对 Spring 框架做了一个大致的介绍。事实上，Spring 框架本身也是一个极为松散的耦合。Spring 并不强迫应用程序开发者必须使用 Spring 框架提供的全部功能。事实上，开发者可以使用 Spring 的某一部分功能，也可以完全抛弃 Spring 的其他功能。例如，本书的 Live 在线书店示例中，DAO 的设计就没有采用 Spring 的 DAO 模块，而是使用了基于范型的 DAO 实现。即使对于同一种功能，Spring 也提供了相当多的方案，尤其是 Spring MVC 框架，这种过多的选择往往令初学者不知所措，因此，只有在深入理解 Spring 原理的基础上，才能更好地运用 Spring 来简化设计和编码。

1.4 Spring的设计思想

Spring 不仅解决了 JavaEE 开发中常见的复杂问题，而且 Spring 一直强调良好的编程实践，例如，针对接口编程、利用依赖注入来解耦、鼓励编写单元测试等。对于学习和正在应用 Spring 框架的开发人员来说，他们更能体会到 Spring 倡导的优秀的设计思想。

1.4.1 使用松散耦合的JavaBean

如果使用 EJB 作为组件模型，那么除了实现组件本身的业务功能外，还引入了 EJB 的接口，并且查找 EJB 组件的唯一方法是使用 JNDI。这种较高的耦合度使得组件完全依赖于 EJB 容器。

Spring 抛弃了 EJB 这种具有特定接口的组件模型，相反，Spring 极力推荐使用普通的 JavaBean 作为构成系统的组件，因为 JavaBean 是普通的 Java 对象，它不依赖于任何特殊实现，也不需要实现任何特定接口。因此，运行在 Spring 容器内的 JavaBean 是松耦合的，并且组件通过依赖注入来实现装配，而不是主动通过 JNDI 来查找其依赖的组件。此外，和 EJB 相比，JavaBean 是轻量级的，运行更快，也更容易编写。

1.4.2 始终针对接口编程

Spring 非常注重良好的编程实践。作为 JavaEE 开发者，好的设计比具体的实现技术更重要。一条通用的设计准则便是将接口和实现相分离，上层的调用者对于底层实现应当一无所知。在 JDK 的类库中，我们随处可见这样的设计：JNDI 接口、集合类接口、SQL 接口等。在 Spring 中，针对接口编程更是发挥到了极致。

1.4.3 工厂模式和更好的Singleton解决方案

使用工厂模式来创建对象比直接用 new 关键字更加容易，因为工厂向客户端隐藏了创建对象的复杂的细节。在 Spring 中，随处可见各种工厂类。实际上，Spring 的 IoC 容器本身就是一个很好的工厂模式的实现。在 Spring 中，除了直接使用 Spring 自身提供的工厂类外，还可以非常容易地编写我们自己的工厂类，并纳入到统一的容器中管理。

在 JavaEE 应用程序中，经常需要用到单一实例的对象，这些对象的生命周期贯穿整个应用程序。为了保证正确实例化这些单例对象，如果采用 Singleton 模式，我们将不得不为每个单例组件实现复杂的 Singleton 模式。Spring 对此提出了更容易也更简单的解决办法：将单例对象纳入 Spring 的 IoC 容器中，由 IoC 容器保证对象的单一实例。实际上，这种方式正是将组件的生命周期管理从组件自身移至 Spring 的 IoC 容器中，使得我们只需关注对象本身的逻辑。默认配置下，Spring 的 IoC 容器管理的对象均是单例。

1.4.4 不重新发明轮子

在现有的组件基础之上构建应用程序比从零开始要容易得多。对此，Spring 的设计哲学之一便是不重新发明轮子。Spring 认为，如果一种框架已经很好地解决了某一领域的问题，那么应当直接使用它，而不是重复设计一个新的框架。为了尽可能方便地重用现有的框架，Spring 集成了大量已经广泛使用的开源框架，例如，Hibernate、Struts 等。Spring 通过包装或者适配将这些第三方组件也纳入到 Spring 的 IoC 容器之中，大大简化了开发人员使用这些第三方框架的难度。

除了集成现有的框架之外，Spring 还对一些通用的服务或 API 接口进行了包装，使之更容易使用，例如，JDBC 接口、处理 JMS 消息等。开发人员利用 Spring 提供的接口更容易实现模块化的组件，而不是过程化的代码。

1.4.5 代码应该很容易被测试

容易测试的代码才是好的代码。随着单元测试越来越深入人心，开发人员逐渐从中体会到单元测试本身不仅仅是测试代码的正确性，同时也是模拟客户端对组件接口的调用。如果一个组件很容易被测试，那么说明该组件也很容易被客户端使用，这意味着良好的接口设计和极低的耦合度。反之，不易被测试的代码也必定不易被客户端所调用。

编写单元测试时，常常需要为被测试的组件注入其依赖的对象，如果这些依赖对象能够很容易地被模拟，则说明整个系统组件之间的耦合度较低。反之，如果很难模拟其依赖的对象，则说明该组件大而臃肿，与其他组件之间有较高的耦合。

传统的 EJB 组件非常难以测试。由于 EJB 组件实现了 EJB 专有接口，即使一个很简单的测试也必须启动 EJB 容器。此外，由于 EJB 组件需要部署，编写模拟 EJB 组件的测试类也是非常困难的，这使得测试单个 EJB 组件往往非常复杂。

Spring 抛弃了复杂的 EJB 组件模型。由于 Spring 的 IoC 容器管理的全部都是普通的 JavaBean 对象，所以测试非常容易。不需要启动 Spring 就可以对每个 JavaBean 编写单元测试，并且直接在 IDE 中运行他们。这些易于测试的 JavaBean 组件实际上简化了开发，并且它们之间是松耦合的关系。

1.4.6 EJB 3.0 会终结Spring吗

毫无疑问，EJB 的标准制定者已经意识到了 EJB 的复杂性和较差的性能。在最新的 EJB 3.0 规范中，POJO 已经成为了 EJB 3.0 的基础， Remote 接口、Home 接口等则全部去掉了。可见，EJB 3.0 受到来自 Spring 等轻量级框架的影响是巨大的。

许多人担心，一旦 EJB 3.0 出炉，且作为 JavaEE 标准，是否会成为 Spring 强有力的竞争对手甚至取代 Spring？其实，这种担心是完全不必要的。首先，EJB 3.0 仍仅仅是一个中间件，它不是一个全面的 JavaEE 框架，无法覆盖如 Web 层的开发；其次，Spring 带来的远远不是一个轻量级框架，而是一种全新的设计和开发思想，例如，依赖注入、AOP、事务抽象、基于模版的异常封装。Spring 将“针对接口”的编程实践发挥到了极致，应用 Spring 重在领会其设计思想，正如学习 Struts 应掌握其 MVC 设计模式一样。如果仅仅将 Spring 单纯作为一个框架来学习，则不过多掌握了一个框架而已，无法从根本上提高面向对象的设计和开发能力。

1.5 如何学习Spring

Spring 是一个无侵略性或侵略性极小的框架，它不强迫你实现任何 Spring 特定的接口，因此，我们开发的组件不但可以非常轻易地通过 Spring 来组装和部署，也完全可以抛弃 Spring 而选择另一个 IoC 框架，甚至手动将他们简单地装配起来。

正如你将在本书后面的章节中所看到的，Spring 提供了从 Web 前端到中间层，再到后端数据库访问的全部功能，并且每一层都提供了相当多的可替换的解决方案。我们既可以使用 Spring 提供的全部功能实现完整的三层结构的应用程序，也可以仅仅使用 Spring 提供的部分功能，这完全取决于我们的需要和项目的需求。

本书的内容组织按照由浅入深、循序渐进的原则，先从 Spring 基础开始讲解，在第 3 章和第 4 章中分别讲述 Spring 框架的两个最核心、最基本的功能：依赖注入和 AOP，读者通过这两章应该对 Spring 有基本的了解。然后，在第 5 章和第 6 章讨论了 Spring 框架的数据访问功能，包括如何设计数据访问模型和管理事务等，使读者有能力应用 Spring 框架实现灵活可靠的数据访问。第 7 章介绍了 Spring Web MVC 框架，并给出了集成多个第三方 MVC 框架的方法。Spring 提供的 MVC 框架极为灵活，我们完全可以根据需要实现一个完整的 Web MVC 应用程序。

在学习 Spring 的过程中，应当按照学会使用、深入原理、剖析源码的步骤一步一步深入。除了介绍如何使用 Spring 框架本身外，本书许多地方都详细阐述了某一功能的实现原理，并深入剖析 Spring 的设计思想。读者不应当仅仅满足于“会用 Spring”，更要学习 Spring 框架本身的设计思想，从根本上提高面向对象的设计能力。

除了一个完整的 Live 在线书店 Web 应用程序外，本书所有章节中的示例代码都提供了完整的项目，可以在 Eclipse 环境下直接运行。建议读者一次性地从本书的配套光盘中导入全部项目。在学习每一章的过程中，可以分别打开对应的项目查看源代码，然后自己动手新建一个项目，编写代码实现相应的功能。

1.6 Spring示例：Live在线书店应用程序

为了让读者更好地学习 Spring 框架，本书还提供了一个基于 Spring 框架的完整的 Web 应用程序——Live 在线书店。和普通的演示代码不同，Live 在线书店是一个真正可以直接部署的 Web 应用程序，它类似于常见的网上书店，为用户提供了以下常见功能。

- 分类浏览各种书籍并对此评论；
- 通过关键字搜索相关书籍；

- 拥有一个购物车，能随时将书籍放入购物车；
- 生成订单并确认付款和送货方式；
- 管理自己的订单、收藏夹和个人信息等。

除了以 Spring 作为核心框架外，Live 在线书店还集成了相当多的开源框架。

（1）Hibernate 是目前应用最广泛的开源对象/关系映射（O/R Mapping）框架，其强大的功能和优秀的性能受到了 Java 开发者的广泛欢迎。Live 在线书店的持久层采用 Hibernate 3.2，配合 JPA 注解，最大程度地简化了数据访问逻辑。

（2）Velocity 是替代 JSP 的另一种表示层的视图解决方案，其最大的特点是以非常直观的表达式无缝嵌入 HTML 页面，避免了在 JSP 中由于大量标签的滥用而使得页面设计非常困难。

（3）Lucene 是一个完全基于 Java 的性能优秀的开源全文搜索引擎，不过考虑到 Lucene 提供的 API 使用比较复杂，采用 Compass 这个封装了 Lucene API 的开源框架即可轻松实现全文搜索。

（4）Acegi 是基于 Spring 开发的一个安全框架，能够实现完善的安全保护，但却能几乎避免对现有系统的改动。Live 在线书店采用 Acegi 作为安全框架，大大减少了实现安全逻辑的代码量。

在第 11 章中，我们将详细介绍 Live 在线书店的设计与实现。总之，以开源框架为基础，构建轻量级的 JavaEE Web 应用程序是 Live 在线书店最大的特点。读者可以通过站点 http://www.livebookstore.net 访问该示例应用程序。

1.7 小结

本章我们简单介绍了 Spring 框架的起源、各主要模块的功能和 Spring 的一些重要的设计思想。当前，Spring 这种轻量级框架正得到越来越广泛的应用，这给传统的 JavaEE 开发带来了一系列新的思考方式。在学习 Spring 的过程中，我们除了掌握 Spring 框架本身之外，更需要深入理解 Spring 的设计思想。

第 2 章

Spring快速入门

2.1 搭建开发环境

在学习 Spring 前，我们需要一个完整的 Java 开发环境。考虑到 Java 5.0 的许多优秀特性已经相当普及，因此在本书中，所有的代码都是基于 Java 5 版本的，许多代码用到了泛型，这意味着代码并不能兼容 Java 1.4 或更早的平台。因此，必须安装 JDK 5.0 开发环境。

对于开发工具而言，有许多免费的和商业的 IDE 开发工具，而 Eclipse 是一个使用最广泛的免费 IDE 开发环境，越来越多的公司和个人在 Eclipse 开发平台上创建了许多优秀的插件，使 JavaEE 的开发更加容易。本书的所有代码均是在 Eclipse 3.2 下开发完成的。但是，在一个好的项目结构中，编译和部署等任务不应该依赖于某个 IDE 环境，因此本书的项目构建使用标准的 Ant 脚本，保证了在脱离 IDE 环境下仍能以命令行方式编译并部署项目。由于流行的 Java IDE（例如，JBuilder、NetBeans 等）几乎全部支持 Ant 作为标准的项目构建工具，因此，读者也完全可以选择自己熟悉的 IDE。稍后还将详细介绍 Ant 脚本和一个第三方 XDoclet 库，用于提取源代码注释并自动生成配置文件。

有许多可选的免费 JavaEE 服务器可用于开发和商业运营，许多商业 JavaEE 服务器（例如，WebLogic）也提供用于开发的免费版本。本书选择 Resin 作为开发和运行环境。Resin 是一个优秀的 JavaEE 服务器，其标准版是免费且开源的。在本书中，所有的例子均在 Resin 标准版服务器上运行调试通过。

注意：所有的开发工具和服务器软件都不应当安装到带有中文的目录下，否则可能引起莫名其妙的错误。为了尽量减少不必要的麻烦，请安装到英文目录下，并且目录名最好不要带有空格。

读者可以在本书配套光盘中找到以上所有的开发工具和服务器软件。

2.1.1 安装JDK 5.0

首先，从 Java 官方站点 http://java.sun.com 下载最新的 JDK 5.0 发布版本。截至本书写作时，目前的最新稳定版本是 JDK 5.0 Update 08。下载 Windows 离线安装版本，运行安装程序，并选择一个合适的安装目录，例如，C:\java\jdk1.5.0_08。然后打开“控制面

板”→“系统”→“高级”→“环境变量”，在系统变量中新建一个环境变量 JAVA_HOME，变量值设定为 JDK 5.0 的安装目录，例如，C:\java\jdk1.5.0_08，如图 2-1 所示。

注意：如果不设置 JAVA_HOME 环境变量，许多依赖 Java 虚拟机的应用程序将无法启动，例如，Resin 服务器。

然后，在系统变量中找到 Path，编辑此变量，在变量值的最前面添加%JAVA_HOME%\bin，目的是将 JDK 可执行文件添加到 Path 中，以便相关应用程序（如 Eclipse）能启动 Java 虚拟机。

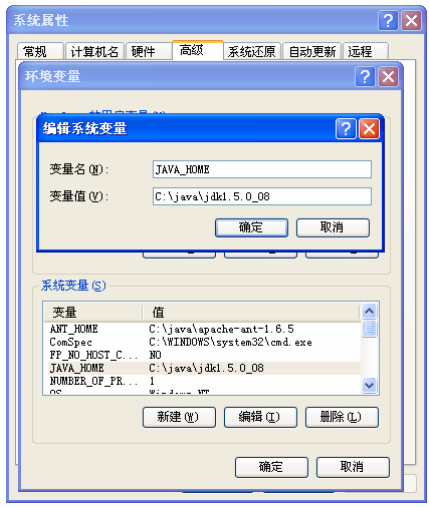


图 2-1

如果使用 Linux 操作系统，可能需要修改/etc/profile 文件，在最后加入如下代码。

```
JAVA_HOME=/usr/share/java/jdk1.5.0_08
PATH=$JAVA_HOME/bin:$PATH
export JAVA_HOME
export PATH
```

要测试以上设置是否正确，请打开命令提示符，输入 java –version，应该显示如下版本信息。

```
C:\Documents and Settings\Xuefeng>java -version
java version "1.5.0_08"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_08-b03)
Java HotSpot(TM) Client VM (build 1.5.0_08-b03, mixed mode, sharing)
```

2.1.2 安装Eclipse IDE

Eclipse 是一个非常优秀的免费 IDE。可以从 Eclipse 官方网站 http://www.eclipse.org 下载最新版本。截至本书写作时，最新稳定版本为 Eclipse SDK 3.2.1。Eclipse SDK 3.2.1 不仅包含了基本的 IDE 和 Java 开发环境，还包含了 Eclipse 平台本身的开发环境和源代码，因此比较庞大（120MB）。如果仅需要 Eclipse 作为 Java 应用程序的一个开发平台，可以选择下载单独的 Eclipse Platform Runtime Binary（34M）和 JDT Runtime Binary（19M），然后解压到相同的本地目录下。这样可以得到一个最精简的完整的 Java 开发环境。

第一次启动 Eclipse 时，需要指定 Eclipse 存放所有工程的根目录，这个根目录被称为 Eclipse 的工作区目录。确保选中“Use this as the default and do not ask again”复选框，以便让 Eclipse 记住该工作区目录，如图 2-2 所示。

Eclipse 有 Perspective（透视图）的概念，一个 Perspective 就是一系列 View 布局的组合，目的是方便开发者使用。例如，Java Perspective 的布局如图 2-3 所示。

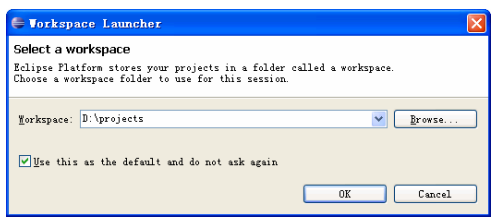


图 2-2

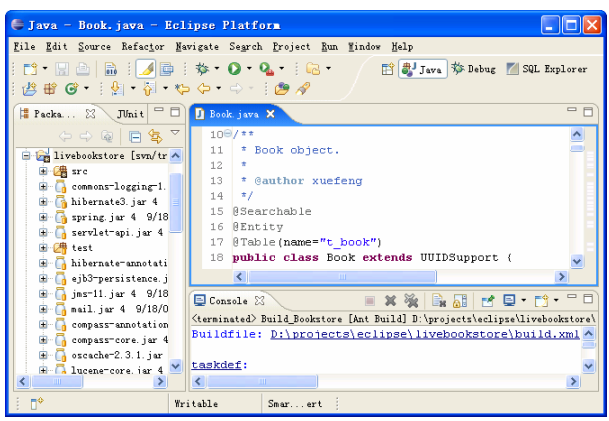


图 2-3

可以通过右上角的按钮非常方便地切换 Perspective，没有列出的 Perspective 可以通过菜单“Window”→“Open Perspective”→“Other...”打开对话框，选择需要的 Perspective。

一般在开发时使用 Java Perspective，启动调试器时，Eclipse 可以自动切换到 Debug Perspective。

要自定义 Eclipse 的设置，请选择菜单“Window”→“Preferences”，在弹出的 Preferences 对话框中设置。请读者务必注意，本书的所有源代码都是以 UTF-8 格式编码的，为了保证不出现乱码，请在 Eclipse 中设置所有的文本文件默认编码为“UTF-8”，如图 2-4 所示。

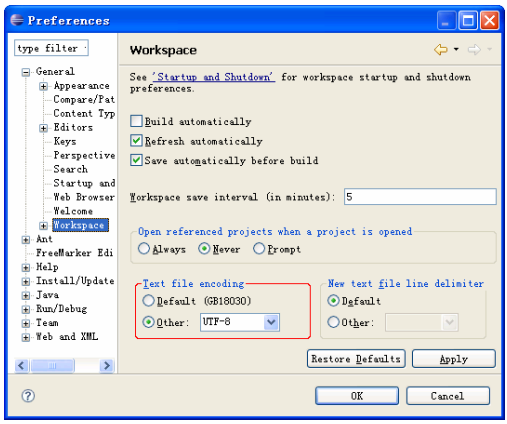


图 2-4

在 Eclipse 中，可以非常方便地导入一个已经存在的工程。选择菜单“File”→“Import...”出现如图 2-5 所示的对话框。

在弹出的对话框中选择“Existing Projects into Workspace”，然后选择本书配套光盘中的 source 目录，这里假定是“E:\source”，如图 2-6 所示。

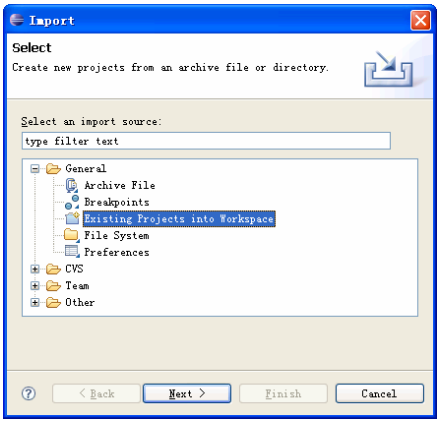


图 2-5

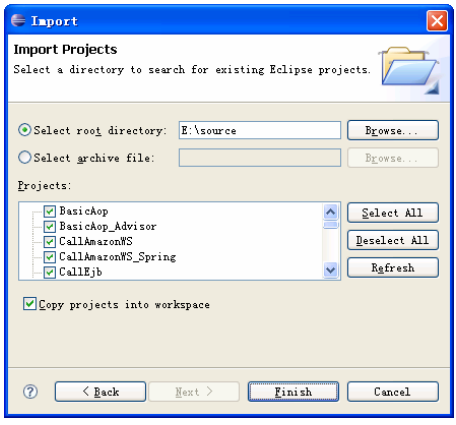


图 2-6

确保选中了“Copy projects into workspace”复选框，这样才能将光盘中的所有工程复制到本地硬盘上，否则将无法编译。单击“Finish”按钮，即可将本书配套光盘中的

示例工程全部导入至 Eclipse 中。

为了让初次使用 Eclipse 的读者快速上手，本书配套光盘中附带了 Eclipse 的 Flash 演示教程，对 Eclipse 不太熟悉的读者可以参考该 Flash 教程。

2.1.3 安装Resin

Resin 是一个非常优秀的 JavaEE 服务器。就性能而言，Resin 提供了目前速度最快的 Servlet 容器，远远超过了 Tomcat，并且 Resin 标准版也是免费的，还可获得 Resin 的源代码。从 Resin 的官方站点 http://www.caucho.com 可以下载最新的 3.1.0 版。解压到本地目录，然后添加一个环境变量 RESIN_HOME，指向 Resin 的根目录。

要启动 Resin，运行 Resin 根目录下的 httpd.exe，Resin 会弹出一个小窗口并打开一个控制台。启动成功后，控制台输出将显示“Resin started in XXXms”字样。

在 IE 浏览器中输入 http://localhost:8080，可以看到默认的首页，如图 2-7 所示。

如果不能正常启动，请检查 JAVA_HOME 和 RESIN_HOME 等环境变量是否正确设置。

Resin 的小窗口可以控制服务器的启动和停止。要完全关闭 Resin 服务器，可以单击“Quit”按钮，如图 2-8 所示。

图 2-7

图 2-8

2.1.4 下载Spring Framework 2.0

截至本书写作时，Spring 2.0 已经正式发布了，可以从 Spring 的官方站点 http://www.springframework.org 下载最新的 2.0.2 发布版本。需要注意的是，Spring 的发布版本提供了一个 with-dependencies 版本（60.1M），它包含了 Spring 依赖的许多第三方开源库，例如，Hibernate、Struts、Commons Logging 等，强烈推荐下载这个发布版本，可以省去查找第三方库的麻烦。下载后解压到某个文件夹，例如，C:\java\spring-framework-2.0.2，可以在 dist 目录下找到 Spring 的发布包 spring.jar 及 Spring 的全部源代码 spring-src.zip，lib 目录下存放了所有的第三方库。

2.2 第一个Spring应用程序

为了体验 Spring 的威力，我们以“Hello，World”为例，介绍如何编写并运行第一

个 Spring 应用程序。整个项目源代码可从本书的配套光盘中获得。

在 Eclipse 中新建一个工程的步骤如下。

① 切换到 Java Perspective。

② 选择菜单“File”→“New”→“Project...”。

③ 在弹出的向导中展开 Java，选择 Java Project。

④ 输入 Project name，例如，“Hello，World”。

⑤ 在 Project layout 中选择“Create separate source and output folders”，可以使源代码和编译后的 class 文件分开存放。

⑥ 单击“Finish”按钮后，可以看到已经创建的工程。

启动 Eclipse，新建一个 Java Project。在 Package Explorer 视图中，如果没有 src 目录，首先要新建一个 src 目录。在工程根节点上单击右键，在弹出的快捷菜单中选择“New”→“Source Folder”，如图 2-9 所示。

只有 Source Folder 才能被 Eclipse 编译，普通目录下存放的 Java 文件将不会被编译。HelloWorld 工程的整个项目结构如图 2-10 所示。

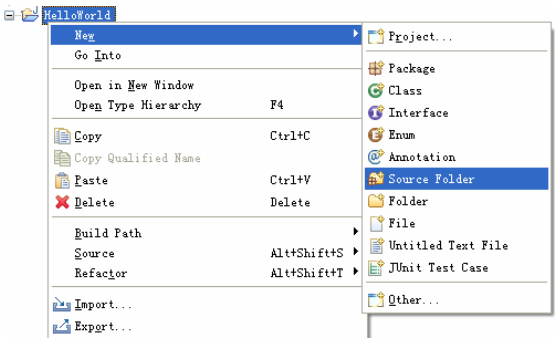


图 2-9

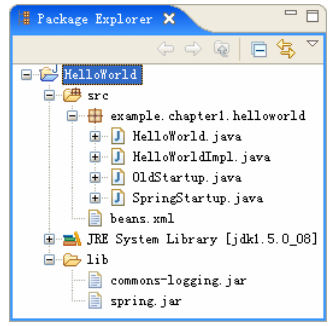


图 2-10

src 目录存放所有的 Java 源代码，lib 目录存放用到的库文件。对于 Spring 应用程序来说，仅需依赖 spring.jar 和 commons-logging.jar。JRE System Library 由 Eclipse 自动添加，其版本取决于目前安装的 JDK 版本。

在编译项目前，还需要将 lib 目录下的 spring.jar 和 commons-logging.jar 添加到项目的 Java Build Path 中。选中 HelloWorld 项目，然后选择菜单“Project”→“Properties”，在弹出的对话框中添加这两个必要的 jar 文件，如图 2-11 所示。

一旦将 jar 文件添加到 Java Build Path，lib 目录下的 jar 文件就会隐藏掉，如图 2-12 所示。

这仅仅影响 jar 文件在 Eclipse 中的显示，不会影响文件的实际存储位置。

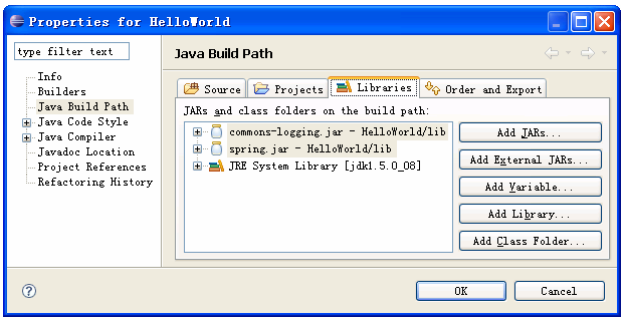


图 2-11

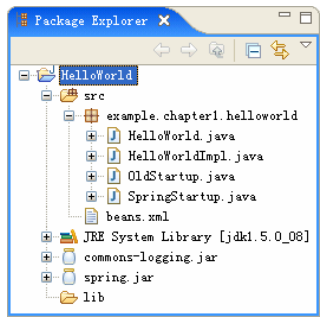


图 2-12

可以为 spring.jar 附加源代码，这样不仅可以在 Eclipse 中浏览 Spring 的源代码，还可以更方便地跟踪调试。右键单击 spring.jar，在弹出的快捷菜单中选择“Properties”，然后选择“Java Source Attachment”，在“Location path”中填入 Spring 的源代码目录即可，如图 2-13 所示。

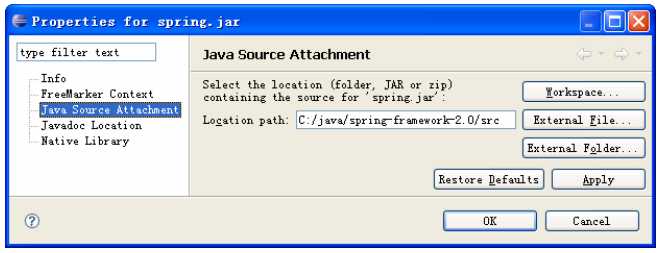


图 2-13

2.2.1 编写Java代码

以经典的“HelloWorld”为例，我们稍作扩展，实现一个标准的 Spring 组件。针对抽象编程是一种良好的面向对象设计原则，因此定义一个 HelloWorld 接口。

```
public interface HelloWorld {
    void setName(String name);
    String say();
}
```

然后针对接口编写实现类。

```
public class HelloWorldImpl implements HelloWorld {
    private String name = null;

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String say() {
        if(name==null)
            return "Hello, world!";
        return "Hello, " + name + "!";
    }
}
```

我们编写一个传统的命令行程序来调用 HelloWorld 组件。

```
public class OldStartup {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorldImpl();
        hello.setName("Spring");
        System.out.println(hello.say());
    }
}
```

要在 Eclipse 中运行一个应用程序，选中 OldStartup.java 文件，然后单击右键，在弹出的快捷菜单中选择“Run As”→“Java Application”，如图 2-14 所示。

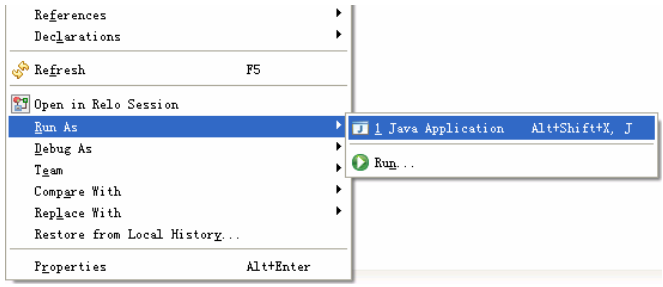


图 2-14

如果没有“Java Application”这一项，请检查选择的 Java 类是否有 main()方法。运行 OldStartup，Eclipse 会自动显示 Console 视图，结果如图 2-15 所示。

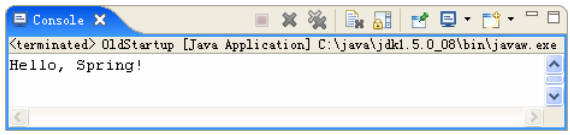


图 2-15

现在我们感兴趣的是如何以“Spring”的方式来调用 HelloWorld 组件。Spring 应用程序的运行流程一般如下。

① 初始化 Spring 的 Bean 工厂。

② 通过 Bean 工厂获得 Bean 实例。

③ 调用 Bean 实例实现应用程序所需的功能。

④ 销毁 Spring 的 Bean 工厂。

典型的代码如下。

```
public class SpringStartup {
    public static void main(String[] args) {
        XmlBeanFactory factory = new XmlBeanFactory(new ClassPathResource
("beans.xml"));
        HelloWorld hello = (HelloWorld)factory.getBean("hello");
        System.out.println(hello.say());
        factory.destroySingletons();
    }
}
```

在运行这个 Spring 应用程序之前，还需要一个 Spring 配置文件。

2.2.2 编写Spring配置文件

正如前面所介绍的，Spring 框架依赖一个 XML 配置文件来管理和装配应用程序的所有 Bean 组件。启动 Spring 前，还需要准备一个 beans.xml 配置文件。

```
<?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="hello" class="example.chapter1.helloworld.HelloWorldImpl">
        <property name="name" value="Spring" />
    </bean>
</beans>
```

XML 文件的根节点是<beans>，这是 Spring 配置文件必须的，在<beans>和</beans>之间通过<bean>节点定义所有的 Bean，每个 Bean 还可设置属性和依赖关系，具体的配置方法将在第 3 章详细讲解。本例中，注意到 bean 的 id 为“hello”，要在 Spring 的 Bean 工厂获取这个 Bean 的实例，只需提供相应的 id。

2.2.3 运行Spring应用程序

在 Eclipse 中运行 SpringStartup，可以看到如图 2-16 所示的结果。

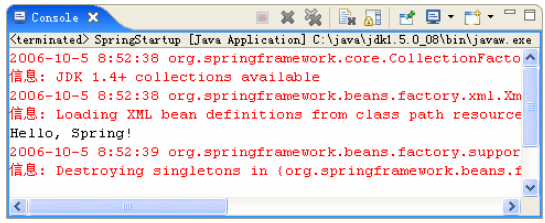


图 2-16

虽然运行结果和 OldStartup 一致，不过，组件的创建和配置却交给了 Spring 框架，应用程序只需使用组件。在第 3 章中，将详细讨论 Spring 是如何管理组件的。

2.2.4 调试Spring应用程序

对于可以单独启动的普通的 Java 命令行程序，Eclipse 提供了非常方便的调试支持。可以在程序中任意设置断点，然后切换到 Debug 模式，选择应用程序的启动类 SpringStartup，在右键弹出菜单中选择“Debug As”→“Java Application”，即进入调试模式。

对于 Web 应用程序，由于需要服务器支持才能启动应用程序，因此需要某种远程调试机制。幸运的是，Resin 服务器和 Eclipse 都提供了对标准远程调试的支持。所谓远程，即跨 Java 虚拟机，在不同的 Java 虚拟机中运行的 Java 应用程序都需要进行远程通信，不一定非得运行在不同的计算机上。Eclipse 和 Resin 本身也是 Java 应用程序。在启动 Resin 时加上以下命令行参数即可打开远程调试功能。

```
httpd -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,
address=12345
```

可以将以上命令写入批处理文件以方便启动。

先启动 Resin，然后在 Eclipse 中选择菜单“Run”→“Debug...”→“Remote Java Application”→“New”，新建一个 Remote Java Application，填入主机名“localhost”，端口号“12345”，如图 2-17 所示。

注意：这个端口号就是 Resin 启动的 address=12345 参数。现在，就可以利用 Eclipse 强大而方便的调试界面对 Web 应用程序进行断点调试和跟踪了。

Web 应用程序的设计和开发将在后续的章节中详细阐述，这里仅介绍如何在 Eclipse

中调试远程应用程序的方法。

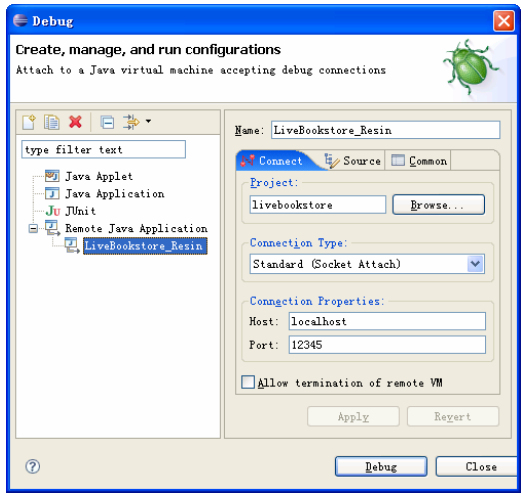


图 2-17

2.3 使用Ant构建项目

Ant 早已成为 Java 项目构建工具事实上的标准。Ant 通过一个 XML 格式的项目配置文件（通常是 build.xml）来构建项目，从而可以实现从编译到部署的全自动化的项目构建。一个项目构建任务由若干 Task 构成，每个 Task 可以指定其依赖的 Task，Ant 会按照依赖关系自动执行必要的 Task。

一个项目的构建过程一般可按照以下顺序进行。

① 定义各目录变量。

② 定义 Classpath 和外部任务。

③ 初始化目录。

④ 执行 javac 编译。

⑤ 生成各类配置文件（如 Spring 配置文件）。

⑥ 运行 JUnit 测试。

⑦ 生成项目部署文件。

对于上面的“Hello，World”示例，编写如下的 build.xml 并放到项目根目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="HelloWorld" default="compile" basedir=".">
    <!-- 定义目录变量 -->
    <property name="src.dir"    value="src"/>
    <property name="lib.dir"    value="lib"/>
    <property name="build.dir"  value="bin"/>
```

```
    <!-- 定义 Classpath -->
    <path id="master-classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
    </path>

    <!-- 初始化${build.dir}目录 -->
    <target name="init">
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- 编译 java 源代码 -->
    <target name="compile" depends="init">
        <mkdir dir="${build.dir}" />
        <javac destdir="${build.dir}">
            <classpath refid="master-classpath"/>
            <src path="${src.dir}"/>
        </javac>
        <!-- 复制 beans.xml -->
        <copy todir="${build.dir}">
            <fileset file="${src.dir}/beans.xml"/>
        </copy>
    </target>
</project>
```

Ant 可以通过<property>定义任意的变量，通常应当把多次出现的目录名、文件名定义为变量，在更改目录名或文件名时，只需更新变量值即可。变量名也应该有明确的意义，可以在任何地方通过${变量名}引用变量。

```
<property name="src.dir" value="src"/>
```

为了构建项目，通常我们将一个构建任务分拆为几个子任务，通过<target>定义。target 必须指定一个 name，还可以指定 depends，表示这个 target 必须依赖于其他的某些 target。例如，名称为“compile”的 target 依赖于名称为“init”的 target。

```
<target name="compile" depends="init">
```

在执行 compile 之前，Ant 会自动首先执行 init。因此，只要正确指定了各个 target 之间的依赖关系，然后在根元素<project>中指定默认的 target，Ant 就会以正确的顺序执行必须的 target。target 的定义顺序对 Ant 没有影响。

Eclipse 默认使用内置的 Java 编译器。为了能使用 Ant 构建项目，需要打开菜单“Project”→“Properties”，选择“Builders”，新建一个 Ant Build，如图 2-18 所示。

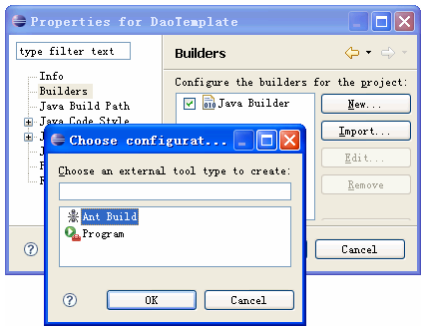


图 2-18

然后，指定 build.xml 的位置和项目的根目录，如图 2-19 所示。

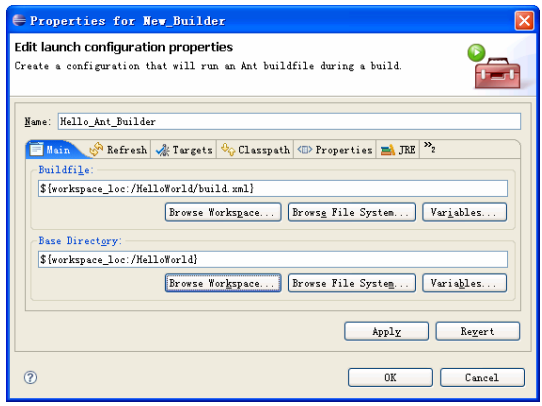


图 2-19

然后，在“Builder”对话框中，选中“Hello_Ant_Builder”复选框，同时取消选择“Java Builder”复选框，如图 2-20 所示。

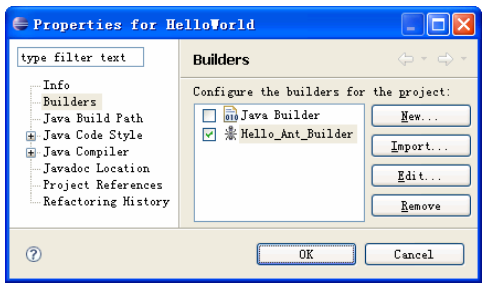


图 2-20

忽略警告信息，单击“OK”按钮保存设置。现在，选择菜单“Project”→“Build Project”，Eclipse 将使用 Ant 来构建项目，如图 2-21 所示。

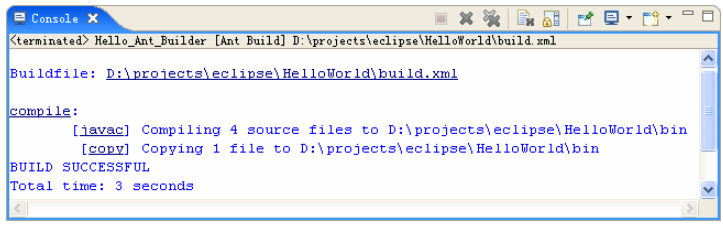


图 2-21

如果构建成功，Ant 会显示“**BUILD SUCCESSFUL**”字样。

要脱离 IDE 以独立的 Ant 来构建此项目，需要首先下载完整的 Ant 包。可以从 http://ant.apache.org/bindownload.cgi 下载。截至本书写作时，最新的稳定版本是 1.6.5。解压到本地目录，然后设置环境变量 ANT_HOME=<Ant 安装目录>，并将%ANT_HOME%\bin 添加到 Path 中。

在命令提示符下，切换到项目所在的根目录。由于我们使用了标准的项目配置文件 build.xml，因此只需简单地键入 ant，如图 2-22 所示。

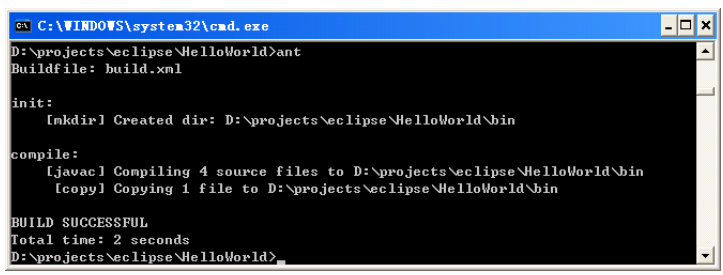


图 2-22

如果没有使用标准的 build.xml 作为文件名，需要在命令行指定 XML 文件。

```
ant -buildfile mybuild.xml
```

读者可能会觉得使用 Ant 构建项目较为烦琐。对于小项目而言，编写 Ant 所需的 build.xml 显得工作量较多，然而随着项目规模越来越大，使用 Ant 构建将大大简化编译、测试、部署和生成 API 文档等一系列工作，并且能够全自动进行，是 Java 项目不可或缺的工具。作为示例，本书的 Live 在线书店示例也使用 Ant 构建。

2.4 使用XDoclet自动生成配置文件

由于 Spring 需要一个 XML 配置文件来正确组装所有的组件，而手动编写 XML 配置文件是相当烦琐的，而且随着应用程序的组件越来越多（在本书的 Live 在线书店应用中，共有 50 多个组件需要配置），XML 文件变得难以维护，并且改动源代码后要相应地修改 XML 文件。因此，需要一种自动生成 XML 配置文件的方式。

XDoclet 是一个开源的自动化生成工具，可用于自动生成 Java 代码和各类配置文件，包括 Spring 的 XML 配置文件。XDoclet 通过提取源代码中的特殊的注释来生成配置文件，因此不必维护配置文件，只需在更改源代码后通过 XDoclet 重新生成一个新的配置文件即可，大大降低了开发和维护的难度。

XDoclet 已经提供了对 Spring 的内置支持，并且很容易扩展，在第 3 章中，还将详细介绍如何在 Java 代码中编写 XDoclet 注释，然后自动生成 Spring 配置文件。

可以从 http://xdoclet.sourceforge.net 下载最新的 XDoclet。截至本书写作时，XDoclet 的最新版本是 1.2.3 版本。

XDoclet 只能以 Ant 的扩展任务形式运行，因此只有使用 Ant 作为项目构建工具时，才能享受到 XDoclet 带来的便捷，这也是本书推荐的开发方式：依赖标准的项目构建工具 Ant，而非某个特定的 IDE。

XDoclet 的核心是 XJavaDoc，这个库负责解析 Java 源代码。由于 XDoclet 1.2.3 版本的 XJavaDoc 还不能正确解析 Java 5 代码，因此需要手动更新 XJavaDoc。下载最新的 XJavaDoc 1.5，然后将 XDoclet 目录下的 xjavadoc-1.2.3.jar 替换为 xjavadoc-1.5-snapshot 050611.jar，即可正确解析 Java 5 源代码。读者也可以在本书的配套光盘中找到支持 Java 5 的 XDoclet 版本，并直接使用。

2.5 Spring 2.0 的新特性

Spring 2.0 对 Spring 1.x 做了较大的修订，并引入了许多新的特性。下面列出了 Spring 2.0 新增的主要功能。

2.5.1 更容易的配置

在 Spring 2.0 中，XML 配置文件的语法变得更简单了。当然，由于本书仅针对 Spring 2.0 框架，因此用过 Spring 1.x 的读者可能才会感觉到 Spring 2.0 中提供的一些新的配置

方式使得 XML 文件不再那么烦琐，并且更加容易阅读。

在 Spring 2.0 中，对 AOP 的配置引入了 AspectJ 的切入点语法，因此可以将切面声明为 Java 5 的注解。不过，Spring 仅仅引入了 AspectJ 的库来解析和配置切入点，其 AOP 的实现并不依赖于 AspectJ 的编译器和织入器。

Spring 2.0 对事务的配置方式做了重大改进。虽然可以继续使用 1.x 版本的配置方式，但笔者推荐使用新的方式，因为它更加简洁。此外，还可以使用 AspectJ 的注解来声明事务属性。在第 6 章中，还将详细讨论关于事务的配置。

2.5.2 对JPA的支持

在最新的 JavaEE 5.0 规范中，引入了一个新的 Java 持久化 API——JPA（Java Persistent API）。JPA 是 EJB 3.0 的重要规范之一，但却独立于 EJB 3.0。它为 JavaEE 应用定义了一个标准的持久化申明，使用了 Java 5 的申明。对于 Java 1.4 或更早版本，JPA 也支持使用传统的 XML 配置文件。

Spring 2.0 提供了一个 JPA 抽象层，通过 JpaDaoSupport 实现标准的 DAO 模式。这与 Spring 的 DaoSupport 继承体系是一致的。

由于本书的代码均是基于 Java 5 平台的，因此，仅讨论如何使用 JPA 申明来简化持久化类的配置。在第 5 章还将详细讨论如何在 Hibernate 中应用 JPA。

2.5.3 对JMS的完整支持

JMS（Java Message Service，Java 消息服务）是 JavaEE 应用的标准消息服务，使用 JMS 能大大减少应用程序的耦合，从而提高应用程序的可维护性和可扩展性。

在 Spring 1.x 中，对 JMS 仅限于创建并发送消息，却没有对处理消息提供封装。这个缺陷在 Spring 2.0 框架中得到了完整解决。在第 8 章中还将详细讨论在 Spring 中发送和处理 JMS 消息。此外，还可以在 Live 在线书店示例中看到如何在 Spring 2.0 中应用 JMS 轻松处理电子邮件。

2.5.4 对Portlet支持

在 Spring 2.0 中，不仅支持传统的基于 Servlet 的 MVC Web 开发，也添加了对 Portlet 的支持。Portlet 是一种整合了多个视图的门户页面技术，用户可以自定义页面的布局。Portlet 不同于 Servlet 之处还在于 Portlet 运行于 Portal 容器中。Spring 2.0 完整地支持 JSR-168 定义的 Portlet 规范，并提供了一个 Portlet MVC 框架。

2.5.5 对动态语言的支持

Spring 2.0 开始支持在 Spring 应用程序中使用动态语言来定义类和对象，通过在 XML 配置文件中定义使用的动态语言，Spring 容器可以完全透明地实例化那些使用动态语言编写的类，并和现有的 Java 类配合实现依赖注入。目前，Spring 2.0 支持 JRuby、Groovy 和 BeanShell 这 3 种动态语言。

2.6 小结

本章主要介绍了 Spring 2.0 开发环境的建立，以及如何在 Eclipse 中编写第一个“HelloWorld”Spring 应用程序。考虑到实际项目的需要，我们还介绍了使用 Ant 构建工程的方法和使用 XDoclet 自动生成配置文件的方式，这些有用的工具可以大大降低项目的构建成本，在后续的章节中我们还会经常用到。

本章还介绍了 Spring 2.0 提供的许多新的特性，和 Spring 1.x 版本相比，2.0 版本增加了许多有用的新特性，除了对 XML 配置文件做了简化外，Spring 2.0 还新增了对 JPA、Portlet、动态语言和 JMS 的完整支持。读者可以在后续的章节中看到这些新特性在 Spring 2.0 中的应用。

从第 3 章开始，我们将依次介绍 Spring 2.0 框架的各个主要模块。IoC 容器是整个 Spring 框架的基础，也是 Spring 其他模块运行的环境，因此，在第 3 章，我们首先介绍 Spring 的 IoC 容器，以及如何使用它实现基于依赖注入的组件装配模式，以大大简化组件的开发和维护。

第 3 章

使用Spring的IoC容器管理Bean

Java 语言作为完全的面向对象的程序设计语言获得了巨大的成功。回顾软件开发的历史，其实就是不断地在更高层次上抽象。面向过程的软件开发以函数的方式来达到代码复用的目的，而面向对象的软件开发则进一步通过类来实现代码复用，更加接近现实世界中的模型。随着软件规模的不断扩大，以组件为基础来构建复杂的应用程序越来越受到重视。和类有所不同，组件是具有一定功能的已编译的类的实现，通常通过接口向外部提供服务，从而能在二进制级别实现复用，而类只能在源代码级别实现复用。

在 Java 中，JavaBean 是最简单的组件。随着 JavaEE 平台的发展，以 EJB 为代表的典型的重量级组件一度被认为是 JavaEE 应用中最重要的技术。EJB 最大的意义在于提供了一个组件的运行环境，即容器的概念。EJB 容器为运行在其中的 EJB 组件提供生命周期管理、事务支持、安全控制等一系列重要的服务。不过，随着轻量级容器的发展，EJB 组件所能享受到的诸如声明式事务管理等功能在普通的 JavaBean 组件上也能实现。Spring 容器就是要为普通的 JavaBean 组件提供一系列有价值的服务，它也是整个 Spring 框架的基石。

3.1 JavaBean概述

JavaBean 是一种符合特定规范的 Java 对象，也是最简单、最基本的 Java 组件。JavaBean 具有一个无参数的构造方法和由 getter/setter 提供的标准属性供外部访问。因此，JavaBean 既可以作为数据对象的载体，也可以被设计为业务组件实现商业逻辑。和 EJB 组件相比，JavaBean 组件的编写、测试要容易得多，占用的资源也较 EJB 少得多。因此，相对于 EJB 重量级组件而言，通常把 JavaBean 称为轻量级组件。

实际上，JavaBean 正是 Spring 框架大力推荐使用的组件。在 Spring 中，凡是可以被实例化或者可以从 JNDI 获得的对象都可以被 Spring 轻量级容器所管理，而并非一定要完全符合 JavaBean 规范。

3.2 IoC入门

3.2.1 容器的概念

容器是为某些组件的运行提供必要运行支持的一个软件环境。例如，Servlet 容器为 Servlet 和 JSP 组件提供运行环境，EJB 容器为 EJB 组件提供运行环境。通常，组件不能脱离容器单独运行。

除了提供一个组件运行环境之外，容器还提供了相当多的访问系统底层服务的简单

方法。例如，EJB 容器为 EJB 组件提供的声明式事务服务，使 EJB 组件的开发人员不必自己编写冗长的事务处理代码，而通过部署描述符申明需要如何使用事务。由于容器将许多底层服务封装为简单的接口或者声明式描述，因此能大大简化组件的开发。

传统的 EJB 容器仅为 EJB 这种重量级组件提供支持，而 Spring 提供的轻量级容器可以管理所有的轻量级 Java 组件，包括 JavaBean、JNDI 对象和所有能被实例化的 Java 对象，它同时也是一个支持依赖注入的 IoC 容器。Spring 容器提供的功能主要包括组件的生命周期管理、组件的配置和组装服务、AOP 支持，以及建立在 AOP 之上的声明式事务服务等。本章讨论的主要内容正是 Spring 容器对组件的生命周期管理和配置组装的服务。通过本章的介绍，读者可以看到，利用 Spring 提供的强大的 IoC 容器，组件的管理和配置变得非常容易。

3.2.2 理解IoC的概念

在最近几年盛行的一些软件开发的术语中，IoC 是出现频率较高的一个单词。IoC 全称为 Inversion of Control，直译为控制反转。何谓 IoC？在解释 IoC 的概念之前，我们来看看通常的 Java 组件是如何协作的。假定一个 BookService 组件，有一个 listBooksByAuthor 的方法，用于列出某个作者编写的所有书籍。

```
public class BookService {
    private BookDao bookDao = new DbBookDao();

    public List<Book> listBooksByAuthor(String author) {
        List<Book> books = bookDao.listAll();
        Iterator<Book> it = books.iterator();
        while(it.hasNext()) {
            if(!it.next().getAuthor().equals(author))
                it.remove();
        }
        return books;
    }
}
```

可以看到，listBooksByAuthor 的功能非常简单，即列出指定作者的书籍。而列出所有书籍的功能被委托给 bookDao 对象。考虑到书籍可能存储的多种形式，例如，数据库、XML 文件等，将 BookDao 申明为接口，并在某个子类中实现具体的功能符合面向对象编程的基本原则之一：针对抽象编程，因此，我们实现了一个具体的子类 DbBookDao。

现在需要考虑的是，BookService 如何持有 bookDao 对象。最简单的方式是，在 BookService 的内部持有一个 DbBookDao 的实例，上面的例子正是这么做的。

考察一下这种“直接实例化并持有”的方式，会发现以下缺点。

（1）在 BookService 中硬编码创建了 BookDao，如果需要另一种 BookDao 的实现，例如，XmlBookDao，则需要修改 BookService 的代码，换言之，BookService 组件不能脱离 BookDao 的具体实现。

（2）BookDao 的实例无法被其他组件共享。假设其他组件也需要引用 BookDao，则多个组件很难共享同一个 BookDao 的实例，因为该实例的生命周期定义在了 BookService 组件中，从而难以共享，实现一个 getBookDao()的方法以暴露 BookDao 虽然可行，却违反了组件之间的契约关系，因为 BookService 并非 BookDao 的工厂，两者不存在创建关系。

（3）如果 DbBookDao 仍需要引用其他资源，例如，DataSource，则 BookService 可能还需要负责管理和维护一个 DataSource，而这完全不是作为上层组件的 BookService 的职责。

（4）测试 BookService 是复杂的，因为必须首先编写 DbBookDao，倘若 DbBookDao 还依赖于 DataSource，则测试必须在真实的数据库环境下执行，而无法用模拟对象（Mock Object）来代替。

从以上几点可以看出，如果系统中有大量的组件，其生命周期和相互之间的依赖关系如果由组件自身来维护，不但大大增加了系统的复杂度，而且会导致组件之间极为紧密的耦合，继而给测试和维护带来极大的困难。

现在的核心问题是如何组装大量的组件，使之互相配合完成复杂的系统功能？解决这一问题的方案正是使用 IoC。

传统的应用程序中，控制权在应用程序本身，程序的控制流程完全由开发者控制。创建 BookService 组件，在创建 BookService 组件的过程中，又创建了 BookDao 组件，然后使用 BookService 组件为用户服务。

在 IoC 模式下，控制权发生了反转：从应用程序转移到了 IoC 容器。组件不再由应用程序负责创建和配置，而是由 IoC 容器负责，应用程序只需直接使用已经创建并配置好的组件。为了让组件能在 IoC 容器中被“装配”出来，需要某种“注入”的机制，才能将一种组件“注入”到另一种组件中。

在 Java 中，使用 set 方法可以非常简单地实现这一注入机制。将 BookService 修改如下。

```
public class BookService {
    private BookDao bookDao;
    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    public List<Book> listBooksByAuthor(String author) {
```

```
        List<Book> books = bookDao.listAll();
        Iterator<Book> it = books.iterator();
        while(it.hasNext()) {
            if(!it.next().getAuthor().equals(author))
                it.remove();
        }
        return books;
    }
}
```

修改后的 bookDao 实例不再由 BookService 创建，而是由 IoC 容器负责将某个 BookDao 实例通过 setBookDao()注入。

以上对 BookService 的修改虽然非常简单，但是却带来了一系列的好处。

（1）BookService 不必关心如何创建 BookDao 的实例，也不必关心 BookDao 的具体实现，只需要使用它就可以了，因此简化了 BookService 的编码。

（2）BookDao 的实例由 IoC 容器管理，因此，可在多个组件之间共享，只要它们也实现了相应的 setBookDao()的方法。

（3）测试 BookService 非常容易，因为可以自行实现一个 MockBookDao 模拟对象，然后注入到 BookService 中，即可测试 listBooksByAuthor 方法，而不需要真实的数据库环境。

许多人认为使用 DI（Dependency Injection，依赖注入）来描述比 IoC 更合适。的确，依赖注入的描述更加准确。不过，本书不打算讨论使用哪个术语更合适，一律将其称之为“依赖注入”。在提到容器时，仍使用“IoC 容器”一词，因为它更为流行。

简单来说，依赖注入解决了最主要的一个问题：将组件的配置与使用相分离，并且由 IoC 容器负责管理组件的生命周期。

由于 IoC 容器负责实例化所有的组件，因此，需要告诉容器如何创建组件和各组件之间的依赖关系。最常见的配置方式是通过一个 XML 文件，例如，Spring 的 IoC 容器所采用的方式，上述组件用 XML 配置如下。

```
<beans>
    <bean id="bookDao" class="DbBookDao" />
    <bean id="bookService" class="BookService">
        <property name="bookDao" ref="bookDao" />
    </bean>
</beans>
```

配置 bookService 组件时，ref 指向的就是 bookDao 组件，Spring 的 IoC 容器就根据该配置自动调用 bookService 的 setBookDao()方法，将 bookDao 组件注入到 bookService 组件中去。

3.2.3 依赖注入的方式

依赖注入的方式主要有 3 种：构造方法注入、设置属性注入和接口注入。3 种注入方式各有优劣。

构造方法注入在构造方法中注入所需的依赖组件。若采用构造方法注入，上面的 BookService 示例将被改写为如下形式。

```
public class BookService {
    private BookDao bookDao;
    public BookService(BookDao bookDao) {
        this.bookDao = bookDao;
    }
}
```

构造方法注入的好处是在组件实例化时就同时设置了所有依赖的组件，不会漏掉某个依赖的组件，组件的初始化代码可以在构造方法中完成。缺点是如果有多个参数个数相同的构造方法，IoC 容器仅依赖参数可能无法区分它们，此外，XML 配置文件也不太直观。使用构造方法注入的 IoC 容器有 PicoContainer，Spring 的 IoC 容器也支持构造方法注入，不过，Spring 更推荐使用设置属性注入。

设置属性注入通过简单的 set 方法注入一个符合参数类型的依赖组件。上面的示例采用的也是这种方法。这种方法的优点是 XML 配置非常直观，缺点是如果忘记注入某个组件，运行时将会抛出 NullPointerException。此外，要在所有注入完成后执行一些初始化代码，需要其他机制。Spring 采用的方法是指定初始化方法的名称，在所有注入完成后调用该方法。

第 3 种注入方式是接口注入。这种注入方式是在接口中定义需要注入的信息。对于上面的示例，要注入 BookDao，首先需要定义一个接口。

```
public interface InjectBookDao {
    void injectBookDao(BookDao bookDao);
}
```

对于需要注入的 BookService 组件，必须实现这个接口。然后，容器通过接口信息完成依赖注入。相对于构造方法注入或设置属性注入，接口注入的侵略性要强得多，而且需要更多额外的工作。使用接口注入的 IoC 容器有 Avalon。

读者可能在一些资料中看到关于依赖注入的 3 种方式分别是 Type 1、Type 2 和 Type 3，实际上这 3 种方式正是指接口注入、设置属性注入和构造方法注入。

3.3 Spring提供的IoC容器

Spring 提供了功能强大的 IoC 容器，支持设置属性注入和构造方法注入这两种依赖注入方式。通过 XML 配置文件可以指定所有组件的依赖关系， Spring 的 IoC 容器便能正确地初始化它们。

在设计上，Spring 的 IoC 容器是一个高扩展性的无侵入式的框架。所谓无侵入式，是指应用程序的组件无需实现特定的 Spring 专有接口便可纳入 Spring 的 IoC 容器进行管理。和 Spring 的 IoC 容器相比，EJB 容器则是高侵入式的，因为 EJB 组件必须实现 EJB 容器的回调接口。

由于 Spring 的无侵入式设计，使得开发者至少获得了两种好处：第一，应用程序组件不必依赖 Spring 框架也可单独进行测试，大大提高了开发效率，降低了测试成本；第二，即使脱离 Spring 环境，应用程序也可自行装配组件，避免了对 Spring 框架的过度依赖。

Spring 的 IoC 容器事实上就是一个实现了 BeanFactory 接口的可实例化类。事实上，Spring 提供了两种不同的容器：一种是最基本的 BeanFactory，另一种是扩展的 ApplicationContext。BeanFactory 仅仅提供了最简单的依赖注入支持，而 ApplicationContext 则扩展了 BeanFactory，提供了更多的额外功能。

实例化 Spring 的 IoC 容器非常简单，就是创建一个 BeanFactory 或者 Application Context 的实例。下面分别讲述如何实例化这两种 IoC 容器。

3.3.1 使用BeanFactory

顾名思义，Spring 的 BeanFactory 采用的是工厂模式，实现了 BeanFactory 接口的类负责创建并配置所有的 Bean。应用程序将 Bean 的创建和配置完全委托给 BeanFactory，然后从 BeanFactory 获取 Bean 并使用它们，如图 3-1 所示。

错误！

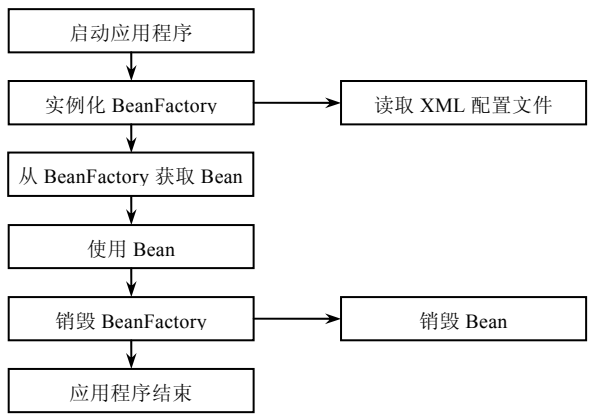


图 3-1

Spring 提供了好几种 BeanFactory 的实现，其中最常用的是 XmlBeanFactory。XmlBeanFactory 通过一个 XML 配置文件创建和配置 Bean。创建一个 XmlBeanFactory 的实例需要一个 Resource 对象，用于指定如何载入 XML 配置文件。

最简单的方法是从一个本地文件中载入 XML 配置文件。

```
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("C:\\test\\
config.xml"));
```

Config.xml 文件的内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
   <bean id="bookService" class="example.chapter3.BookService">
      <property name="bookDao" ref="bookDao" />
   </bean>

   <bean id="bookDao" class="example.chapter3.DbBookDao" />
</beans>
```

许多情况下，使用 ClassPath 定位 XML 配置文件更为方便，只需确保 XML 配置文件位于 ClassPath 路径之内。

```
BeanFactory factory = new XmlBeanFactory(
   new ClassPathResource ("config.xml"));
```

还可以使用 ByteArrayResource 提供已经读入内存的 XML 文件的内容。

在创建了 BeanFactory 的实例后，就可以随时获得 Bean 的实例，只需调用 BeanFactory 的 getBean()方法并传入 Bean 的 id。

```
BookService bookService = (BookService)factory.getBean("bookService");
```

为了能让示例代码运行，我们在 Eclipse 中新建一个完整的 IoC 工程，其结构如图 3-2 所示。

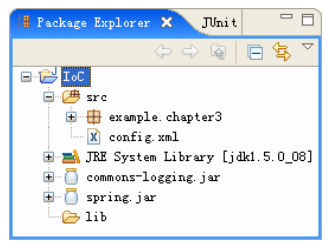


图 3-2

读者可以从本书的配套光盘中导入工程源代码并直接运行。

3.3.2 使用ApplicationContext

ApplicationContext 本质上仍然是一个 BeanFactory，因为 ApplicationContext 是从 BeanFactory 继承而来的。不过，ApplicationContext 提供了更多的功能，并能够与一些应用环境整合，例如，对于 Web 应用程序，提供了 XmlWebApplicationContext，用于简化 Web 应用程序的配置。

和基本的 BeanFactory 相比，ApplicationContext 还提供了国际化支持、事件的发布和通知机制等。

Spring 同样提供了好几种 ApplicationContext 的实现，包括 FileSystemXmlApplicationContext、ClassPathXmlApplicationContext、XmlWebApplicationContext 等。

使用 ApplicationContext 和使用基本的 BeanFactory 一样简单。将上面的 XmlBeanFactory 替换为 ClassPathXmlApplicationContext，其余的代码不用改动就能运行。

```
        ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
```

3.4 Bean初始化

在讨论了如何创建一个 Spring 的 IoC 容器后，我们要关注的是这个 XML 配置文件，因为 Spring 的 IoC 容器根据 XML 配置文件来初始化 Bean。需要注意的一点是，ApplicationContext 初始化 Bean 和基本的 BeanFactory 有所不同，基本的 BeanFactory 总是延迟加载 Bean，直到第一次调用 getBean("beanId")方法请求 Bean 的实例时，BeanFactory 才会创建这个 Bean，而 ApplicationContext 在自身初始化时就一次性创建了所有的 Bean，了解这一点区别非常重要，因为 ApplicationContext 在初始化时就能验证 XML 配置文件的正确性，而使用基本的 BeanFactory，直到调用 getBean("beanId")方法获取 Bean 实例时，才可能会发现配置错误而导致抛出异常。

只有在非常简单的情况下，使用基本的 BeanFactory 才可能满足我们的需求。绝大多数时候，ApplicationContext 是最佳选择。在启动的时候，ApplicationContext 就能检测出配置文件的错误，这比使用基本的 BeanFactory 在运行一段时间后调用 getBean()抛出异常要好得多。并且，延迟加载会带来性能上的损失。

相对于基本的 BeanFactory，ApplicationContext 唯一的缺点是由于在启动时需要一次性实例化所有的 Bean，如果定义的 Bean 很多，则启动时间会较长。

下面讨论如何在 XML 配置文件中定义 Bean。一个标准的 Spring XML 配置文件如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
    <bean id="bean1" class="package.ExampleBean1" />
    <bean id="bean2" class="package.ExampleBean2" />
    <bean id="bean3" class="package.ExampleBean3" />
</beans>
```

Spring 的 XML 配置文件的根节点必须是<beans>，其中可以包括多个<bean>节点，每个节点定义一个 Bean，上述 XML 文件一共定义了 3 个 Bean。

对于每个<bean>节点，class 指定了 Bean 的完整类名，以便 Spring 容器能通过反射生成一个 Bean 的实例，id 赋予了 Bean 唯一的名称，通过 getBean()方法从容器获取 Bean 时，需要告知 Bean 的名称。

由于 id 也是 XML 中节点的唯一标识，受到 XML 规范本身的限制，不能使用特殊字符作为 id。如果一定要使用特殊字符，可以用 name 来替代 id。

```
<bean name="@_@" class="package.ExampleBean1" />
```

下面的例子中，我们以 ClassPathXmlApplicationContext 为例，注意：XML 配置文件被命名为 config.xml 并放置在 src 目录下，这样可以确保编译后配置文件位于 ClassPath 中。读者可以从本书的配套光盘中获取项目的源代码。

注意：在 Spring 1.x 版本中，使用 DTD 来验证 XML 配置文件，一个典型的 XML 文件如下。

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

Spring 2.0 版本仍然可以继续使用上面的 DTD 验证，不过，某些新的特性（如 scope

属性）将无法通过 DTD 验证，从而导致 Spring 容器启动失败。对于 Spring 2.0 版本，强烈推荐使用 Schema 验证，典型的 XML 文件如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>
```

关于 DTD 和 Schema 的讨论已经超出了本书范围，读者只需复制上述内容即可。

3.4.1 Bean的初始化流程

当使用 Spring 的 IoC 容器时，例如，ClassPathXmlApplicationContext，在调用 getBean("beanId")方法获取 Bean 实例之前，Bean 就已经被 Spring 初始化了。当容器关闭时，Bean 会被正确地销毁。了解 Spring 的 IoC 容器的初始化流程非常重要，Spring 严格按照以下顺序初始化一个 Bean。

① 容器根据 XML 配置文件中 Bean 的定义实例化一个 Bean，并传入必要的构造方法参数。

② 容器根据 XML 配置文件使用依赖注入设置 Bean 的属性。

③ 如果 Bean 实现了 BeanNameAware 接口，调用其 setBeanName()方法。

④ 如果 Bean 实现了 BeanClassLoaderAware 接口，调用其 setBeanClassLoader()方法。

⑤ 如果 Bean 实现了 BeanFactoryAware 接口，调用其 setBeanFactory()方法。

⑥ 如果使用 ApplicationContext 并且 Bean 实现了 ResourceLoaderAware 接口，调用其 setResourceLoader()方法。

⑦ 如果使用 ApplicationContext 并且 Bean 实现了 ApplicationEventPublisherAware 接口，调用其 setApplicationEventPublisher ()方法。

⑧ 如果使用 ApplicationContext 并且 Bean 实现了 MessageSourceAware 接口，调用其 setMessageSource()方法。

⑨ 如果使用 ApplicationContext 并且 Bean 实现了 ApplicationContextAware 接口，调用其 setApplicationContext()方法。

⑩ 如果使用 WebApplicationContext 并且 Bean 实现了 ServletContextAware 接口，调用其 setServletContext()方法，仅对 Web 应用程序有效。

⑪ 如果关联了 BeanPostProcessor，调用 BeanPostProcessor 的 postProcessBeforeInitialization ()方法。

⑫ 如果 Bean 实现了 InitializingBean 接口，调用其 afterPropertiesSet()方法执行一些初始化工作。

⑬ 如果 Bean 定义了 init-method 方法，调用这个方法执行一些初始化工作。

⑭ 如果关联了 BeanPostProcessor，调用 BeanPostProcessor 的 postProcessAfterInitialization ()方法。

执行完以上步骤后，Bean 就处于就绪状态，随时可以被应用程序使用。上述步骤虽然烦琐，但是通常仅第 ①、② 和 ⑬ 步是必要的，因为其他步骤都会引入 Spring 的专有接口，如非特别必要，一般不要去实现它们。

如果编写的 Bean 可能会提供给别人使用，则第 ⑫ 步（即实现 InitializingBean）可能是必要的，因为这样可以保证 Bean 能被 Spring 自动调用其 afterPropertiesSet()方法正确地完成初始化工作。

容器关闭时，会销毁已创建的 Bean。销毁过程较之初始化过程简单得多：

1．如果 Bean 实现了 DisposableBean 接口，调用其 destroy()方法执行资源清理等工作；

2．如果 Bean 定义了 destroy-method 方法，调用这个方法执行资源清理等工作。

同样的道理，由于实现 DisposableBean 接口会引入 Spring 的专有接口，因此，建议指定 destroy-method 方法来清理资源，除非该 Bean 可能会提供给别人使用。

3.5 装配Bean

告知 Spring 容器如何正确创建并配置 Bean 的方式就是在 XML 配置文件中对每个<bean>节点进行正确的配置。对于以下的 ExampleBean：

```
package example.chapter3;
public class ExampleBean {
    private List list;
    private int size = 100;
    private String version;
    public void setSize(int size) {
        this.size = size;
    }
    public void setVersion(String version) {
        this.version = version;
    }
    public void init() {
        list = new ArrayList(size);
    }
```

```
}
```

假定这个 ExampleBean 需要设置 size 和 version 两个属性，然后调用 init()方法完成必要的初始化工作，定义这个 Bean 的 XML 片段如下。

```
<bean id="exampleBean"
      class="example.chapter3.ExampleBean"
      init-method="init"
>
    <property name="size" value="10" />
    <property name="version" value="1.0_beta" />
</bean>
```

可以通过 init-method 指定一个无参数的初始化方法，如果实现了 InitializingBean 接口，则不必指定 init-method，Spring 容器会自动调用 afterPropertiesSet()方法。不过，这种方式引入了 Spring 的专有类，因此不推荐使用。

3.5.1 注入基本类型

通过 set 方法可以注入指定的属性值。Spring 支持各种基本类型的属性，包括 boolean、int 等原始类型，以及 String、Class 和 URL 类型。

对于下面的 BasicBean：

```
public class BasicBean {
    private String title;
    public void setUseCache(boolean useCache) {}
    public void setMaxSize(int size) {}
    public String getTitle() { return title; }
    public void setTitle(String title) {this.title = title;}
}
```

对应的 XML 配置片段如下。

```
<bean id="basicBean" class="example.chapter3.BasicBean">
    <property name="useCache" value="true" />
    <property name="maxSize" value="100" />
    <property name="title" value="A Basic Bean" />
    <property name="file" value="config.xml" />
</bean>
```

Spring 会自动将 value 的值转化为合适的类型。如果无法完成转换，将抛出异常。对于 Class 注入，需要完整的类名；对于 URL 注入，必须要确保该文件可以通过 ClassPath 找到。

上述配置仅适用于 Spring 1.2 以上版本。对于更早期版本，配置文件稍显复杂。

```
<property name="useCache"><value>true</value></property>
```

两种方式都能在 Spring 2.0 中正常工作，不过，推荐使用 Spring 2.0 的方式配置，因为配置文件更加简洁。

3.5.2 注入引用类型

如果需要注入的属性不是基本类型，而是引用类型，则使用<ref>替代<value>。例如，在 RefBean 中注入 BasicBean。

```
public class RefBean {
    private BasicBean basicBean;
    public void setBasic(BasicBean basicBean) {
        this.basicBean = basicBean;
    }
}
```

对应的 XML 配置片段如下。

```
<bean id="refBean" class="example.chapter3.RefBean">
    <property name="basic" ref="basicBean" />
</bean>
```

上述配置同样仅适用于 Spring 2.0 版本。Spring 1.x 版本的配置应写为如下形式。

```
<property name="basic"><ref bean="basicBean" /></property>
```

3.5.3 注入null

如果对某个属性注入 null，则必须使用<null />。例如，对上述的 basic 属性显示设置为 null，需要这样写。

```
<property name="basic"><null /></property>
```

<null />同样也适用于下面将要讲解的集合类型。

3.5.4 注入List类型和数组类型

除了基本属性外，Spring 也支持各种集合类型的注入。对于 List 和数组来说，两者的配置形式是一样的。例如，以下的 ListBean 包含一个普通的 List、一个泛型 List 和一

个 int[]数组。

```
public class ListBean {
    public void setChildren(List children) {}
    public void setPrices(List<Float> prices) {
        for(Float f : prices) {
            System.out.println(f);
        }
    }
    public void setFibonacci(int[] fibonacci) {
        for(int n : fibonacci) {
            System.out.println(n);
        }
    }
}
```

对应的 XML 配置片段如下。

```
<bean id="listBean" class="example.chapter3.ListBean">
    <property name="children">
        <list>
            <value>A String</value>
            <ref bean="basicBean" />
        </list>
    </property>
    <property name="prices">
        <list>
            <value>12.54</value>
            <value>82</value>
            <value>27.9</value>
        </list>
     </property>
     <property name="fibonacci">
        <list>
            <value>1</value>
            <value>1</value>
            <value>2</value>
            <value>3</value>
            <value>5</value>
        </list>
    </property>
</bean>
```

设置 List 和数组类型的方法是在<list>和</list>之间包含若干<value>或<ref>节点，可以混合使用<value>和<ref>，不过，必须和 List 或数组的类型相符。

如果使用 Java 5 的范型，例如，上例的 List<Float>，Spring 2.0 可以通过反射自动获得强类型 List<Float>的泛型信息，从而强制将<value>转化为 Float 类型。这条规则对于其他集合类型也适用，前提是必须使用 Java 5 的泛型。

3.5.5 注入Set类型

Set 类型和 List 类似，只不过注入的类型是 Set，因此，可以包含任意的<value>或<ref>节点。例如，下面的 SetBean。

```
public class SetBean {
    public void setCups(Set cups) {
        for(Object obj : cups) {
            System.out.println(obj);
        }
    }
}
```

对应的 XML 配置片段如下。

```
<bean id="setBean" class="example.chapter3.SetBean">
    <property name="cups">
        <set>
            <value>spring</value>
            <value>hibernate</value>
            <value>hibernate</value>
            <ref bean="listBean" />
        </set>
    </property>
</bean>
```

由于 Set 中不能包含重复元素，因此，打印结果仅包含 3 项。

```
spring
hibernate
example.chapter3.ListBean@1cb25f1
```

3.5.6 注入Map类型

相对于 List 和 Set，Map 类型由于存在键-值映射，因此，格式稍微复杂一点，每一项都是由一个键和对应的值构成，通过<entry>节点来定义各项。

```
<bean id="mapBean" class="example.chapter3.MapBean">
    <property name="weekday">
        <map>
            <entry key="Monday">
                <value>do something</value>
            </entry>
```

```
            <entry key="Tuesday">
                <ref bean="setBean" />
            </entry>
        </map>
    </property>
</bean>
```

在上述配置中，注入的 Map 键只能为 String 类型，但实际上，Map 可以使用任意的 Object 作为键。为了使用另一个 Bean 作为键，需要按如下方式编写<entry>节点。

```
<entry>
    <key>
        <ref bean="basicBean" />
    </key>
    <value>basic</value>
</entry>
<entry>
    <key>
        <ref bean="listBean" />
    </key>
    <ref bean="setBean" />
</entry>
```

还可以进一步简写为如下形式。

```
<entry key-ref="basicBean" value="basic" />
<entry key-ref="listBean" value-ref="setBean" />
```

3.5.7 注入Properties类型

Properties 类型也是由键-值对构成的，注入和 Map 非常类似，不过，由于 Properties 的键-值都只能是 String，因此，不必再使用<value>。注入 Properties 的 XML 配置片段如下。

```
<bean id="propBean" class="example.chapter3.PropBean">
    <property name="env">
        <props>
            <prop key="target">1.5</prop>
            <prop key="encoding">UTF-8</prop>
            <prop key="debug">on</prop>
        </props>
    </property>
</bean>
```

在使用上述集合类型时，要注意注入属性的参数使用接口而非具体类，例如，使用 List 而非 ArrayList，这样才能保证 Spring 能正确地注入，这也符合针对抽象编程的原则。

3.5.8 注入Resource资源

除了基本类型、引用类型和集合类型外，在应用程序中还经常使用各种资源。Java 的标准库提供了 File、URL、InputStream 等常用的资源类型，但是这些资源无法方便地注入到 Bean 中，因此，Spring 提供了一个更强大、更方便的访问底层资源的 Resouce 抽象接口，大大简化了在 Bean 中注入各种 Resource。

Resource 接口定义了以下几个方法。

```
public interface Resource extends InputStreamSource {
    // 返回资源是否存在:
    boolean exists();

    // 返回 InputStream 是否已打开:
    boolean isOpen();

    // 返回一个 URL 对象:
    URL getURL() throws IOException;

    // 返回一个 File 对象:
    File getFile() throws IOException;

    // 根据相对路径创建一个 Resource:
    Resource createRelative(String relativePath) throws IOException;

    // 返回文件名,通常是不含路径的文件或目录名:
    String getFilename();

    // 返回资源描述,通常在出错后使用:
    String getDescription();
}
```

Spring 提供了几种 Resource 的实现，最常用的 Resouce 实现如下。

（1）UrlResource：封装了一个 URL 对象，它可以表示 HTTP、FTP 或者文件对象。

（2）ClassPathResource：可以从 ClassPath 中获得一个资源，对于访问那些位于 ClassPath 中的文件资源特别有用。

（3）FileSystemResource：封装了一个 File 对象，可以访问文件系统资源。

在 Spring 的 XML 配置文件中，注入的 Resource 资源总是由一个字符串来表示的。

我们无需关心底层的 Resource 究竟是何种具体类型。事实上，Spring 根据字符串来判断应该如何创建相应的 Resource。表 3-1 概述了从字符串到 Resource 的转化规则。

表 3-1

前　缀	说　明	示　例
classpath:	从 ClassPath 加载该资源	classpath:config.xml
http:	作为 URL 加载该资源	http://java.sun.com/
file:	作为 URL 加载该文件资源	file:///C:/WINDOWS/
无	根据具体的 ApplicationContext 判断	test.txt

我们以一个 Resource 工程来演示如何在 Bean 中注入各种 Resource。在 Eclipse 中建立 Resource 工程如图 3-3 所示。

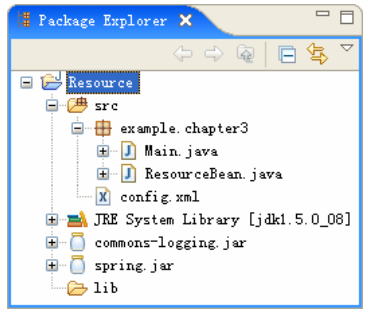


图 3-3

ResourceBean 是需要注入 Resource 的目标 Bean，我们定义 3 个属性，分别注入 URL 类型、ClassPath 类型和 File 类型的 Resource。

```
public class ResourceBean {
    public void setUrlResource(Resource resource) throws IOException {
        System.out.println(resource.getURL());
    }
    public void setClasspathResource(Resource resource) throws IOException {
        System.out.println(resource.getFile());
    }
    public void setFileResource(Resource resource) throws IOException {
        System.out.println(resource.getFile());
    }
}
```

在 Spring 的配置文件 config.xml 中，我们只需指定正确的前缀即可。

```
<bean id="resourceBean" class="example.chapter3.ResourceBean">
```

```
    <property name="urlResource" value="http://www.javaeedev.com/" />
    <property name="classpathResource" value="classpath:config.xml" />
    <property name="fileResource" value="C:/WINDOWS/NOTEPAD.EXE" />
</bean>
```

对于没有任何前缀的“C:/WINDOWS/NOTEPAD.EXE”，Spring 根据 ApplicationContext 的具体类型来决定是使用 FileSystemResource 还是 ClassPathResource。由于我们希望将该资源作为 FileSystemResource 使用，因此，在 main()方法中需要启动一个 FileSystemXmlApplicationContext。

```
public static void main(String[] args) {
    new FileSystemXmlApplicationContext("classpath:config.xml");
}
```

运行结果如下。

```
http://www.javaeedev.com/
D:\projects\Resource\bin\config.xml
C:\WINDOWS\NOTEPAD.EXE
```

如果采用 ClassPathXmlApplicationContext 容器，注入的“C:/WINDOWS/ NOTEPAD.EXE”将被转化为一个 ClassPathResource。

此外，还有一些 Resource 实现，例如，ServletContextResource、InputStreamResource、ByteArrayResource 等，读者可以参考 Spring 的 API。

3.6 构造方法注入

在本章开头提到的 3 种依赖注入的方式中，除了设置属性注入外，Spring 还支持构造方法注入。

使用构造方法注入的优点是，构造方法的参数是强制注入的，这样可以保证 Bean 在创建时就被正确初始化了。相反，属性可能由于多次注入而导致不正确的状态。

下面的 ConstrutorBean 定义的构造方法需要注入两个 int 参数。

```
public class ConstructorBean {
    public ConstructorBean(int min, int max) {
        System.out.println("(int, int)");
    }
}
```

在 XML 配置文件中，需要在<bean>和</bean>之间加入<constructor-arg>节点，按照构造方法的参数顺序排列。

```
<bean id="constructorBean" class="example.chapter3.ConstructorBean">
    <constructor-arg value="100" />
    <constructor-arg value="200" />
</bean>
```

当然，也可以混合使用构造方法注入和设置属性注入。

由于在 Java 语言中，构造方法可以重载，如果定义了多个构造方法，Spring 会根据参数个数和类型来自动选择合适的构造方法。不过，有些时候，Spring 也无法区分参数个数相同而类型不同的构造方法。例如，给 ConstrutorBean 添加另一个构造方法。

```
public class ConstructorBean {
    public ConstructorBean(int min, int max) {
        System.out.println("(int, int)");
    }
    public ConstructorBean(String min, String max) {
        System.out.println("(String, String)");
    }
}
```

XML 配置文件不变，但是注入的参数对于以上两个构造方法来说均符合要求。那么，Spring 容器到底调用的是哪一个呢？运行程序后结果表明调用的是第 2 个构造方法。

```
(String, String)
```

因此，需要告诉 Spring 更多的信息才能正确调用我们希望的构造方法。可以为<constructor-arg>指定一个 type 属性，明确指定构造方法的参数类型。

```
<bean id="constructorBean" class="example.chapter3.ConstructorBean">
    <constructor-arg type="int" value="100" />
    <constructor-arg type="int" value="200" />
</bean>
```

再次运行程序，现在 Spring 容器已有了足够的信息并正确调用了第一个构造方法。

```
(int, int)
```

由于 Spring 同时支持设置属性注入和构造方法注入，因此，选择哪一种方式就存在很大的争议。Spring 团队推荐优先考虑设置属性注入，因为设置属性注入简单明了，能清楚地从属性名称上看出依赖关系，而构造方法注入存在多个构造方法导致的不确定性，并且不利于实现 Bean 的继承。

事实上，哪种方式最符合应用情况就应该使用哪种。有些时候，必须使用构造方法注入才能正确初始化 Bean。大多数时候，选择设置属性注入会更简单明了。对于需要一些初始化工作的 Bean，不要忘了在注入属性后指定一个 init-method 来初始化 Bean。

对于接口注入，由于其侵入性较强，并且使用烦琐，在 Spring 中没有对其提供支持。

3.7 Bean的作用域

除了能在 XML 配置文件中定义 Bean 的依赖注入外，还可以定义 Bean 的作用域。Bean 的作用域定义了 Bean 的生命周期，在 Spring 2.0 中，一共定义了 5 种作用域，分别是 singleton、prototype、request、session 和 globalSession。其中，后 3 种仅对 Web 应用程序有效。

Spring 2.0 通过 scope= ""来定义 Bean 的作用域，而在 Spring 1.x 中，由于只有 singleton 和 prototype 这两种作用域，因此，Spring 1.x 中定义 Bean 的作用域是通过 singleton="true" 或 singleton="false"来定义。为了向后兼容，Spring 2.0 也支持 Spring 1.x 的定义方式，不过，应首先考虑用 scope=""来定义 Bean 的作用域，而且绝不要混合使用两种定义。

3.7.1 Singleton作用域

如果不指定 scope，默认值即为 scope= "singleton"。Spring 的 IoC 容器仅为每个 Bean 创建一个实例并保持 Bean 的引用，换言之，每次调用 getBean()方法请求某一个 Bean 时，Spring 总是返回相同的 Bean 实例。因此，每个 Bean 有且仅有一个实例，这类似于《设计模式》一书中的 Singleton（单例）模式。

Singleton 模式由 GoF 在《设计模式》一书中第一次提出，这个模式能保证一个类在整个应用程序的运行过程中有且仅有一个实例。例如，对于下面的 Singleton 类：

```
public class Singleton {
   private static Singleton instance = new Singleton();
   private Singleton() {};
   public static Singleton getInstance() {
      return instance;
   }
}
```

由于构造方法被申明为 private，因此，在类的外部不可实例化该类，唯一的实例持有在静态引用中，从外部获取该实例的唯一入口是调用 Singleton 的静态方法 Singleton.getInstance()。

通常，应用程序中的许多组件都只需要一个实例就足够了，例如，线程安全的 Dao、DataSource 等。如果将组件设计为 GoF 的 Singleton 模式，则存在以下问题。

（1）Singleton 组件的编码和普通 Bean 相比更加复杂。

（2）Singleton 组件无法直接实例化，难以实现依赖注入。

（3）Singleton 组件难于测试，尤其是状态经常变化的 Singleton。

Spring 解决这一问题的方法非常简单却极其有效：将组件交给 Spring 的 IoC 容器去创建和配置，应用程序只要通过 Spring 容器获取相应的组件，默认即为 Singleton。

读者可能已经注意到了，Spring 仅仅借用了“Singleton”一词，表示组件“有且仅有”一个实例，开发人员编写的组件仍然是普通的 Bean，Spring 并不能阻止应用程序自行通过 new 操作符生成 Bean 的实例。一个好的实践是始终坚持让 Spring 的 IoC 容器来创建和管理 Bean，不要试图在 Spring 容器的外部自行创建 Bean。

3.7.2 Prototype作用域

如果需要每次返回 Bean 的新实例，则需要告诉 Spring 容器采用 Prototype 作用域。

定义一个 Prototype 作用域需要设置 scope="prototype"。例如，要每次获取系统最新的时间，我们告诉 Spring 返回一个新的 java.util.Date 的实例。

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

我们写一个程序来验证每次返回的 Date 实例是否不同。

```
public static void main(String[] args) throws Exception {
   ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
   System.out.println("Get date: " + context.getBean("date"));
   Thread.sleep(10000);
   System.out.println("Get date: " + context.getBean("date"));
   Thread.sleep(10000);
   System.out.println("Get date: " + context.getBean("date"));
}
```

运行程序，控制台打印出的日期是不同的，说明返回的 3 个 Date 实例各不相同。

```
Get date: Mon Oct 09 11:31:49 CST 2006
Get date: Mon Oct 09 11:31:59 CST 2006
Get date: Mon Oct 09 11:32:09 CST 2006
```

由于采用 prototype 作用域时，Spring 容器总会返回一个新创建的实例，因此，Spring 容器一旦将实例交给客户端，就不再对其跟踪引用，所以无法对 prototype 作用域的 Bean 定义 destroy-method，不过仍可以定义 init-method。

如果将 XML 配置文件中的 scope="prototype"去掉，作用域就变成了 singleton。再次运行程序，可以看到打印出的 Date 内容是相同的。

```
Get date: Mon Oct 09 11:34:55 CST 2006
Get date: Mon Oct 09 11:34:55 CST 2006
Get date: Mon Oct 09 11:34:55 CST 2006
```

可见，使用 Spring 的 IoC 容器管理 Bean 是多么方便。许多情况下，只需要改动 XML 配置文件，甚至不需要改动代码。

3.7.3 其他作用域

Spring 2.0 添加了 3 种新的作用域：request、session 和 globalSession。这 3 种作用域由于仅对 Web 应用程序有效，因此，将在第 7 章讨论。

3.8 配置工厂Bean

通常，应用程序直接使用 new 操作符创建新的对象，然后做必要的初始化工作，此后，该对象就处于“就绪”状态并能随时服务。为了能将对象的创建和使用相分离，一个好的实践是采用工厂模式，应用程序将对象的创建及初始化的职责交给工厂对象，如图 3-4 所示。

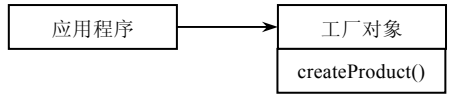


图 3-4

当我们使用 Spring 的 IoC 容器管理 Bean 时，Spring 容器就相当于一个复杂的工厂，它负责根据 XML 配置文件创建并配置 Bean。

很多时候，应用程序需要使用自己的工厂对象来创建 Bean，如果将这一功能也交给 Spring 容器，Spring 容器管理的就不是普通的 Bean 实例，而是“工厂”实例。我们把 Spring 管理的“工厂”实例暂且称为“工厂 Bean”。此时，应用程序调用 getBean()方法时，Spring 返回的不是直接创建的 Bean 实例，而是工厂 Bean 创建的 Bean 实例，如图 3-5 所示。

错误！

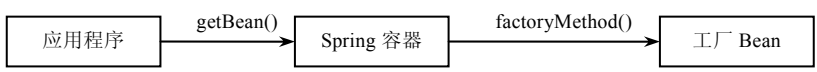


图 3-5

现在，问题变成了如何在 Spring 中配置工厂 Bean。在 Spring 中，可以定义 3 种不

同类型的工厂 Bean。下面分别讲解 3 种工厂 Bean 的配置。

3.8.1 使用静态工厂

静态工厂即 GoF《设计模式》一书中最简单的静态工厂方法模式。对象的创建被委托给工厂类的一个静态方法。例如，下面的例子使用静态工厂方法返回一个随机数。

```
public class StaticFactoryBean {
    public static Integer createRandom() {
        return new Integer(new Random().nextInt());
    }
}
```

如果直接使用这个静态工厂，典型的 Java 代码如下。

```
Integer rnd = StaticFactoryBean.createRandom();
```

如果要将其纳入 Spring 容器来管理，需要通过 factory-method 指定静态方法名称。XML 配置片断如下。

```
<bean id="random"
     class="example.chapter3.StaticFactoryBean"
     factory-method="createRandom"
/>
```

调用 getBean("random")返回的便是静态工厂方法产生的随机数。反复调用 getBean("random")可以发现，返回的随机数是相同的，因为默认的 Bean 的作用域是 singleton，Spring 将缓存并始终返回第一次返回的 Bean 实例。若要每次返回不同的随机数，指定 scope="prototype"即可。

在实际应用中，静态工厂方法仅能用于很简单的情况，适用范围很小。

3.8.2 使用实例工厂

和静态工厂方法相比，通过实例工厂创建一个对象时，必须首先实例化工厂对象，然后才能调用工厂对象的成员方法，这往往是因为对象的创建还依赖于工厂实例的内部状态。例如，定义一个 InstanceFactoryBean，返回一个预定义格式的当前日期，格式存储于工厂的私有字段中。

```
public class InstanceFactoryBean {
    private String format = "yy-MM-dd HH:mm:ss";
    public void setFormat(String format) {
```

```
        this.format = format;
    }

    public String createTime() {
        return new SimpleDateFormat(format).format(new Date());
    }
}
```

定义实例工厂需要分别定义两个 Bean。

```
<bean id="instanceFactoryBean"
    class="example.chapter3.InstanceFactoryBean">
    <property name="format" value="yyyy-MM-dd HH:mm:ss" />
</bean>
<bean id="currentTime"
      factory-bean="instanceFactoryBean"
      factory-method="createTime"
/>
```

第一个 Bean 定义了实例工厂本身，这和定义一个普通的 Bean 没有任何区别；第二个 Bean 定义如何通过实例工厂获取 Bean，需要指定实例工厂的名称和方法名。要获取格式化的当前时间，调用 getBean("currentTime")；调用 getBean("instanceFactoryBean") 呢？很容易想到，这将返回工厂实例本身，打印出来的内容类似于“example.chapter3.InstanceFactoryBean@10bc49d”。

3.8.3 实现FactoryBean接口

第 3 种方式是隐式地使用一个工厂 Bean 实例，该 Bean 实现了 Spring 专有的 org.springframework.beans.factory.FactoryBean 接口。下面的 PiFactoryBean 返回一个 Double 类型的 π 。

```
public class PiFactoryBean implements FactoryBean {
    public Object getObject() throws Exception {
        return new Double(3.14159265358979);
    }
    public Class getObjectType() {
        return Double.class;
    }
    public boolean isSingleton() {
        return true;
    }
}
```

定义这个 Bean 和普通的 Bean 没有任何区别。

```
<bean id="pi" class="example.chapter3.PiFactoryBean" />
```

然而，一旦某个 Bean 实现了 FactoryBean 接口，这个 Bean 就不能再被作为普通 Bean 使用。Spring 的 IoC 容器会自动检测，如果该 Bean 实现了 FactoryBean 接口，则调用 getBean("pi") 返回的将不是 PiFactoryBean 的实例，而是 PiFactoryBean 的工厂方法 getObject()返回的 Double 对象实例。

要获得 PiFactoryBean 本身的实例呢？Spring 提供了类似 C/C++的一个取地址符号“&”，用 getBean("&pi")返回的便是工厂实例。

注意：“&”符号只能用于实现了 FactoryBean 接口的工厂 Bean。对于上述第 2 种实例工厂，没有这样的用法，因为实例工厂是单独定义的，可以直接获得；对于上述第 1 种静态工厂，由于工厂本身都没有被实例化，因此，更不可能用“&”取得工厂实例了。

为避免与普通 Bean 混淆，如果需要自己编写这种特殊的工厂 Bean，请务必命名为 XxxFactoryBean，以便和普通 Bean 区分开来。

3.8.4 常用的FactoryBean

请读者注意，不要将这里的“FactoryBean”与“工厂 Bean”混淆，“工厂 Bean”是一个泛指，而这里的“FactoryBean”是指实现了 FactoryBean 接口的 Bean，即上文提到的第 3 种工厂 Bean。

Spring 框架本身大量使用了 FactoryBean，用来提供许多有用的服务。

1．JndiObjectFactoryBean

JndiObjectFactoryBean 用于获取指定的 JNDI 对象，这个 FactoryBean 配置非常简单，唯一需要指定的是 jndiName 属性。例如，下面的 XML 配置片断能够获取名为“java:comp/env/systemStartTime”的 JNDI 对象。

```
<bean id="systemStartTime" class="org.springframework.jndi.JndiObject
FactoryBean">
    <property name="jndiName" value="java:comp/env/systemStartTime" />
</bean>
```

通常，在 JavaEE 服务器环境下，已经由厂商提供了 JNDI 的完整实现，只要 JNDI 的路径有效，我们简单地调用 getBean("systemStartTime")就获得了该 JNDI 对象。

为了在命令行程序测试 Spring 的 JndiObjectFactoryBean，需要自行提供一个 JNDI

实现。在 JDK 5.0 中，一共提供了 4 种 JNDI 实现，分别是 LDAP、CORBA、RMI 和 DNS。由于没有真实的 LDAP 和 CORBA 环境，使用 RMI 是最简单的实现方式。下面的例子启动了一个基于 RMI 的 JNDI 服务并绑定一个 Date 对象。

```
        private static void bindJndi() throws Exception {
            // 启动 RMI:
            LocateRegistry.createRegistry(1099);
            // 设置 JNDI 环境:
            System.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.
registry.RegistryContextFactory");
            System.setProperty(Context.PROVIDER_URL, "rmi://localhost:1099");
            // 扩展 Date 实现 Remote 接口:
            class RemoteDate extends Date implements Remote {};
            // 绑定至 JNDI:
            InitialContext ctx = new InitialContext();
            ctx.bind("java:comp/env/systemStartTime", new RemoteDate());
            ctx.close();
        }
```

由于使用 RMI 的 JNDI 对绑定对象有所限制，被绑定对象必须实现 Remote 接口，因此我们扩展出了一个 RemoteDate。

测试代码如下。

```
        public static void main(String[] args) throws Exception {
            bindJndi();
            ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
            System.out.println("Get system start time from JNDI: "
                    + context.getBean("systemStartTime"));
        }
```

运行上述程序，顺利获得了绑定的 Date 对象。

```
        Get system start time from JNDI: Tue Oct 10 08:12:11 CST 2006
```

对比一下使用传统的方式获得一个 JNDI 对象。

```
        private static Object jndiLookup() {
            InitialContext ctx = null;
            try {
                ctx = new InitialContext();
                return ctx.lookup("java:comp/env/systemStartTime");
            }
            catch(NamingException ne) {
                throw new RuntimeException(ne);
            }
```

```
        finally {
            if(ctx!=null) {
                try {
                    ctx.close();
                }
                catch(NamingException e) {}
            }
        }
    }
```

可以看出，真正有用的代码仅仅两三行，但却不得不编写大量的异常处理语句。通过 Spring 容器，我们将 JNDI 的查找简化为几行配置文件，免去了开发者编写大量重复而枯燥的代码。

2．ProxyFactoryBean

ProxyFactoryBean 用于简化 AOP 的实现，通过 ProxyFactoryBean 创建的对象与其代理的对象有相同的接口，但返回给客户端的是经过增强的对象，而客户端却感觉不出任何区别。在第 4 章将详细讨论如何使用 ProxyFactoryBean 实现 AOP 功能。

3．TransactionProxyFactoryBean

TransactionProxyFactoryBean 实现了声明式事务管理，大大简化了事务的配置。TransactionProxyFactoryBean 与 ProxyFactoryBean 类似，不过增加的是声明式的事务功能。在第 6 章还将详细讨论 TransactionProxyFactoryBean 的使用方法。

4．LocalSessionFactoryBean

LocalSessionFactoryBean 能非常方便地生成 Hibernate 的 SessionFactory 对象。通过 Spring 提供的 LocalSessionFactoryBean，甚至不需要 Hibernate 的配置文件，真正实现了无缝集成。在第 5 章将讨论如何集成 Hibernate。

3.9 自动装配和模版装配

除了在 XML 配置文件中对每个 Bean 都显式地指定其依赖关系外，Spring 还提供了一些用于简化配置的装配方式，包括自动装配和模版装配。

3.9.1 使用自动装配

如果对于在 XML 配置文件中设置组件的各种注入关系感到烦琐，可以试试使用 Spring 容器的自动装配功能，它能减少 XML 配置文件的内容。

在上面介绍的 XML 配置中，我们总是显式地注入属性。如果使用自动装配，可以减少 XML 配置文件的部分内容。

Spring 容器支持好几种自动装配，分别是 byName、byType、constructor 和 autodetect。

若指定某个 Bean 采用 byName 方式的自动装配，例如：

```
<bean id="aBean" class="..." autowire="byName" />
```

则 Spring 容器负责自动注入该 Bean 的所有属性。容器将查找与属性名相同的 Bean，然后自动注入到该属性中，如果没有找到，则该属性将不会被注入。当然，使用自动装配时，也可以显式地注入 Bean 的部分属性。

若使用 byType 方式的自动装配，注入过程与 byName 类似，不同的是，容器查找的是与属性的设值方法参数类型兼容的 Bean。

若采用 constructor 方式的自动装配，则容器将试图找出所有与构造方法的参数类型兼容的 Bean，然后确定某个合适的构造方法。如果没有符合调用任何构造方法所需的 Bean，则容器将抛出 UnsatisfiedDependencyException。

若采用 autodetect 方式的自动装配，则容器根据该 Bean 是否有默认的构造方法来决定采用 byType 方式还是 constructor 方式。

默认的 autowire 设定是“no”，即不采用自动装配。如果在配置文件的根节点<beans>中指定了自动装配的方式，则所有的<bean>都将采用指定的自动装配方式，除非显式设置了 autowire。

虽然自动装配简化了配置，但是，组件之间的依赖关系将无法清晰地在 XML 配置文件中体现出来，不利于项目的理解和维护。此外，自动装配是否正确还依赖于 Bean 的命名等，因此增加了出错的几率和调试的成本。

Spring 还提供了一个 dependency-check 的属性，用来检查 Bean 的属性是否被注入了，这对于自动装配会比较有用。可以设置为 none、simple、object 和 all，分别表示不进行依赖检查，仅检查基本属性和集合属性，仅检查<ref>注入的属性，检查所有属性。默认设置为“none”，即不检查。使用 dependency-check 的好处之一是可以避免忘记注入某个属性而导致运行期的 NullPointerException。还可以在根节点<beans>定义 default-dependency-check，这将覆盖所有<bean>的 dependency-check 设置。

在下一节将会讲到，某些情况下，使用模版装配也能大大减化 XML 配置文件，还可以考虑使用 XDoclet 完全自动生成 XML 配置文件，因此我们推荐，除非确有必要，一般不要使用自动装配。

3.9.2 使用模版装配

如果需要配置大量的同一类型的 Bean，而这些 Bean 都有一些相同的属性，可以考虑使用模版来装配 Bean，这样能简化 XML 配置文件的编写。

例如，对于下面的 Employee 类。

```
public class Employee implements Serializable {
    private String name;
    private String title;
    private String company;
    private String department;

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

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getCompany() { return company; }
    public void setCompany(String company) { this.company = company; }

    public String getDepartment() { return department; }
    public void setDepartment(String department) { this.department =
department; }
}
```

如果需要配置多个 Employee，而这些 Employee 的 company 和 department 都是相同的，则可以考虑先定义一个 Bean 作为模版。

```
<bean id="abstractEmployee"
     class="example.chapter3.Employee"
     abstract="true"
>
    <property name="company" value="Live Bookstore Inc." />
    <property name="department" value="R&amp;D Center" />
</bean>
```

在 Bean 的定义中，指定 abstract="true"表示这个 Bean 仅作为模版使用，Spring 容器不会为标记为 abstract 的 Bean 创建实例。

然后，分别定义 3 个 Employee Bean，指定 parent="abstractEmployee"，则这 3 个 Bean 将自动继承 abstractEmployee 的 company 和 department 属性。

```
<bean id="engineer1" class="example.chapter3.Employee" parent="
abstractEmployee">
    <property name="name" value="Bill" />
    <property name="title" value="Software Engineer" />
</bean>
<bean id="engineer2" parent="abstractEmployee">
```

```
        <property name="name" value="Steven" />
        <property name="title" value="Hardware Engineer" />
    </bean>
    <bean id="manager" class="example.chapter3.Employee" parent="
abstractEmployee">
        <property name="name" value="Michael" />
        <property name="title" value="Manager" />
        <property name="department" value="HR" />
    </bean>
```

可以在继承的 Bean 中覆盖其 parent 的设置，例如，manager Bean 覆盖了模版中的 department 属性。

继承的 Bean 如果与其 parent 的 class 完全相同，则 class 可以省略；如果不同，则继承的 Bean 必须是其 parent 的子类，这样才能确保 parent 中的设置也能应用于子类，否则将得到一个 ClassCastException 异常。

使用模版装配 Bean 的意义在于简化同一组 Bean 的配置，这样可以避免每个 Bean 都重复编写相同的 XML 配置，还可以避免某一个 Bean 忘记了注入该属性。

有趣的是，如果一个 Bean 被定义为 abstract 作为模版使用，因为 Spring 容器不会去实例化它，所以也不会检查其属性是否合法，因此可随意注入任何属性。例如，上例的 Employee 由于实现了 Serializable 接口，将模版 Bean abstractEmployee 的 class 改为“java.io.Serializable”，XML 配置文件仍然有效，尽管 Serializable 接口根本不含 company 和 department 属性。

```
    <bean id="abstractEmployee"
         class="java.io.Serializable"
         abstract="true"
    >
        <property name="company" value="Live Bookstore Inc." />
        <property name="department" value="R&amp;amp;D Center" />
    </bean>
```

3.10 定制Bean

通常情况下，应用普通的 Bean 和工厂 Bean 足够满足绝大多数情况下的需求了。不过，在 Spring 容器中，还有一些 Bean 具有特殊身份，Spring 容器处理这些特殊 Bean 和其他普通 Bean 是区别对待的，这些特殊 Bean 可以实现一些特殊的功能以满足特殊的需求。

（1）获得自身在 Spring 容器中的配置信息。

（2）获得容器的实例。

（3）可以切入到 Bean 的生命周期中，以便检查或修改 Bean 的属性。

（4）监听并处理 Spring 容器和其他 Bean 发布的消息。

3.10.1 获取Bean的信息

对于 Bean 来说，通常，我们关心的是如何在 Spring 容器中组装他们，Bean 自身甚至不知道容器的存在。在某些情况下，如果 Bean 需要从容器获得自身的某些信息，则可以实现相应的 Spring 专有接口。

例如，对于实现了 BeanNameAware 的 Bean，该 Bean 在初始化时可以获得 Spring 容器传给它的名字。下面定义的 WhoAmI Bean 就可以打印出自身的名字。

```
public class WhoAmI implements BeanNameAware {
    public void setBeanName(String name) {
        System.out.println("My name is " + name);
    }
}
```

相应的 XML 配置片段如下。

```
<bean id="whoAmI" class="example.chapter3.WhoAmI" />
```

运行程序，可以找到输出的“My name is whoAmI”。

如果同时为 Bean 定义了 id 和 name 两个属性，那么 Spring 容器会传入哪个值？测试结果显示 Spring 容器总是优先使用 id 属性，在 id 属性不存在的情况下才使用 name 属性。不过，绝大多数情况下，Bean 根本不必关心自己的名字。

3.10.2 获取容器

如果 Bean 希望得到 Spring 容器的引用，可以实现 ApplicationContextAware 或 BeanFactoryAware 接口，它们分别对应使用 ApplicationContext 容器和使用基本的 BeanFactory 容器的情况。

例如，定义一个实现了 ApplicationContextAware 接口的 WhereAmI 的 Bean。

```
public class WhereAmI implements ApplicationContextAware {
    public void setApplicationContext(ApplicationContext context) throws
BeansException {
        System.out.println(context);
    }
}
```

打印出的 ApplicationContext 信息如下。

```
org.springframework.context.support.ClassPathXmlApplicationContext:
```

```
display name [org.springframework.context.support.ClassPathXmlApplicationContext;
hashCode=25276323]; startup date [Sat Oct 14 13:36:50 CST 2006]; root of context hierarchy
```

获得容器的引用未必是一件好事，因为 Bean 一旦获得了容器的引用，就可以随心所欲地访问其他 Bean，从而打破了 Bean 组件之间的松散耦合，抵销了使用 IoC 容器原本可以带来的好处。

3.10.3 使用BeanPostProcessor

在 Bean 的初始化阶段，Spring 容器也提供了一种切入机制，允许检查或者修改 Bean 的属性。要实现这一功能，我们需要编写一个特殊的 Bean，它实现了 Spring 的专有接口 BeanPostProcessor。

现在，我们就来实现一个自定义的 MyBeanPostProcessor，该类实现了 BeanPostProcessor 接口的两个方法：postProcessBeforeInitialization 和 postProcessAfterInitialization，它们分别在 Bean 的初始化方法执行前后调用，因此，可以在 postProcessAfterInitialization 方法中检查 Bean 的状态并修改。

还记得本章在依赖注入一节中的 BasicBean 的例子吗？BasicBean 有一个可读写的 title 属性，我们当时在 XML 配置文件中注入的字符串为“A Basic Bean”。现在，我们希望对这个 Bean 进行特殊的后处理，将其 title 属性字符串全部变为小写。这个功能正是在 MyBeanPostProcessor 中的 postProcessAfterInitialization 方法中完成的。

```
public class MyBeanPostProcessor implements BeanPostProcessor {
   public Object postProcessBeforeInitialization(Object bean, String name)
throws BeansException {
       System.out.println(">>postProcessBeforeInitialization: " + name);
       return bean;
   }
   public Object postProcessAfterInitialization(Object bean, String name)
throws BeansException {
       System.out.println(">>postProcessAfterInitialization: " + name);
       if(bean instanceof BasicBean) {
          System.out.println(">> Check BasicBean...");
          BasicBean bb = (BasicBean)bean;
          bb.setTitle(bb.getTitle().toLowerCase());
       }
       return bean;
   }
}
```

对于使用基本的 BeanFactory 和使用 ApplicationContext 这两种容器来说，要将我们

定义的这个特殊的 MyBeanPostProcessor 置入容器，方法有所不同。对于基本的 BeanFactory，由于 Bean 是延迟创建的，因此在获得一个 BeanFactory 实例后，必须立刻以编程的方式置入 MyBeanPostProcessor。

```
XmlBeanFactory factory = new XmlBeanFactory(
            new ClassPathResource("config.xml"));
factory.addBeanPostProcessor(new MyBeanPostProcessor());
```

在 ApplicationContext 中定义 MyBeanPostProcessor 就更方便了，只需在 XML 配置文件中定义，和普通 Bean 的定义没有任何区别。

```
<bean id="myBeanPostProcessor"
     class="example.chapter3.MyBeanPostProcessor" />
```

无论采用哪种方式，一旦向 Spring 容器中置入了 MyBeanPostProcessor，则所有的 Bean 的初始化过程都会被 MyBeanPostProcessor 截获，每个 Bean 的初始化流程都变为如下形式，如图 3-6 所示。

在 MyBeanPostProcessor 中，可以通过传入的两个参数 bean 和 name 判断当前正在初始化的 Bean 的类型和名称。在 MyBeanPostProcessor 中，我们打印出了每个 Bean 的名称，然后用 instanceof 判断当前正在初始化的 Bean 是否是 BasicBean，如果是，则在 postProcessAfterInitialization()方法中将已注入的 title 属性变为小写。

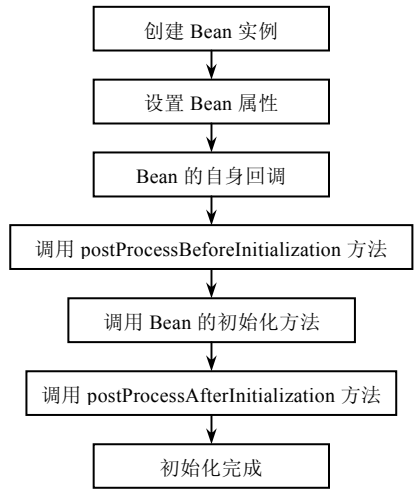


图 3-6

在 XML 配置文件中，basicBean 的 title 属性被注入为“A Basic Bean”。运行程序后，可以看到，basicBean 注入的 title 属性已被 MyPostProcessorBean 修改为“a basic bean”。

```
BasicBean basicBean = (BasicBean) context.getBean("basicBean");
```

```
System.out.println(basicBean.getTitle());
```

输出为：

```
a basic bean
```

3.10.4 使用@Required检查依赖注入

Spring 2.0 还提供了一个非常有用的 BeanPostProcessor： RequiredAnnotation BeanPostProcessor，它负责检查一个 Bean 的某个被标记为@Required 注解的属性是否被注入了。如果忘记注入这个属性，RequiredAnnotationBeanPostProcessor 将会抛出 BeanCreationException 异常。当然，这一特性必须要求使用 Java 5 或更高版本。

例如，在 RefBean 的源代码中，写上@Required 注解就要求这个 basic 属性必须被注入。

```
public class RefBean {
    private BasicBean basicBean;
    @Required
    public void setBasic(BasicBean basicBean) {
        this.basicBean = basicBean;
    }
}
```

要让这个检查生效，还需要在 XML 配置文件中声明这个 RequiredAnnotation BeanPostProcessor。

```
<bean id="requiredChecker"
class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProce
ssor" />
```

当然，如果使用基本的 BeanFactory，则需要以编程的方式加入这个 BeanPost Processor。

现在把 XML 配置文件中对 RefBean 的 basic 属性注入去掉，再次运行程序，在启动 ApplicationContext 容器的时候，将得到一个 BeanCreationException 异常。

```
Caused by: org.springframework.beans.factory.BeanInitializationException:
Property 'basic' is required for bean 'refBean'
```

这对于检查一些必须要注入的属性来说是非常有用的，它避免了在运行期由于忘记注入某个属性而导致的 NullPointerException，并且这种方式比指定 Bean 的 dependency-check 要灵活得多，我们推荐使用@Required 注解来检查依赖注入，而不使用 dependency-check。

3.10.5 使用BeanFactoryPostProcessor

BeanPostProcessor 使我们有能力在 Spring 容器对 Bean 初始化前后对所有的 Bean 进行检查或修改，而 BeanFactoryPostProcessor 允许我们在 Spring 容器对所有的 Bean 初始化之前对 Bean 的定义做一些修改，例如，更改 Bean 的 scope，甚至 class 类型。Spring 容器的初始化流程如图 3-7 所示。

为了理解这个神奇的魔法背后的原理，我们决定编写一个 ChangeTypeBean Factory PostProcessor，它能将一种类型的 Bean 变为另一种类型的 Bean。我们在 Eclipse 中创建一个 ModifyDefinition 工程，结构如图 3-8 所示。

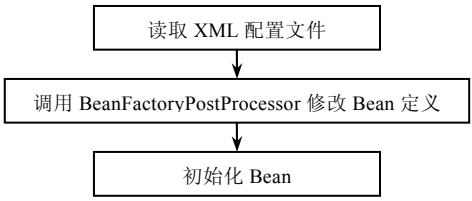


图 3-7

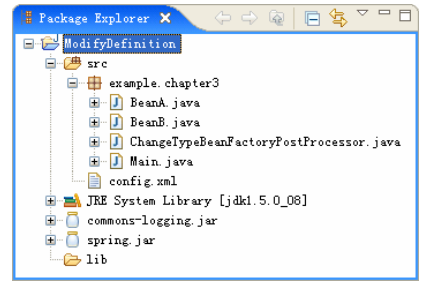


图 3-8

BeanA 和 BeanB 是两个简单的 JavaBean，定义如下。

```
/* src/example/chapter3/BeanA.java */
public class BeanA {
    public String toString() {
        return "I am BeanA";
    }
}

/* src/example/chapter3/BeanB.java */
public class BeanB {
    public String toString() {
        return "I am BeanB";
    }
}
```

然后，我们在 config.xml 中定义两个类型为 BeanA 的 Bean。

```
<bean id="a1" class="example.chapter3.BeanA" />
<bean id="a2" class="example.chapter3.BeanA" />
```

现在，我们考虑如何将这两个类型为 BeanA 的 Bean 变为类型为 BeanB 的 Bean。一旦 Spring 容器将这两个 Bean 初始化完毕，就无法更改其 Class 类型了。唯一的修改机会在 Spring 容器载入 config.xml 配置文件后，还没有初始化这两个 Bean 之前，修改其定

义，才能实现将 BeanA 类型更改为 BeanB 类型。

实现这一更改的关键在于创建一个 ChangeTypeBeanFactoryPostProcessor，它必须实现 BeanFactoryPostProcessor 接口。

```
public class ChangeTypeBeanFactoryPostProcessor implements BeanFactory
PostProcessor {
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException
    {
        String[] beanNames = beanFactory.getBeanDefinitionNames();
        for(String beanName : beanNames) {
            BeanDefinition bd = beanFactory.getBeanDefinition(beanName);
            if(bd.getBeanClassName().equals("example.chapter3.BeanA"))
                bd.setBeanClassName("example.chapter3.BeanB");
        }
    }
}
```

在 postProcessBeanFactory()方法中，我们可以获得 ConfigurableListableBeanFactory 的实例，继而获得所有的 Bean 的定义，然后就有机会根据需要修改 Bean 的定义。这里我们将类型为 example.chaper3.BeanA 类型的 Bean 全部改为 example.chapter3.BeanB 类型。对于使用 ApplicationContext 的容器来说，我们还需要在 config.xml 配置文件中声明这个 ChangeTypeBeanFactoryPostProcessor。

```
<bean id="a2b" class="example.chapter3.ChangeTypeBeanFactoryPostProcessor" />
```

使用 BeanFactory 容器则需要手动将这个 ChangeTypeBeanFactoryPostProcessor 应用到具有 ConfigurableListableBeanFactory 接口的 BeanFactory 上。

```
XmlBeanFactory factory = new XmlBeanFactory(
    new FileSystemResource("beans.xml"));
new ChangeTypeBeanFactoryPostProcessor().postProcessBeanFactory(factory);
```

在 main()方法中，我们将这两个 Bean 打印出来。

```
public static void main(String[] args) throws Exception {
    ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
    System.out.println(context.getBean("a1"));
    System.out.println(context.getBean("a2"));
}
```

运行 main()方法，我们看到打印的结果显示，两个原本定义为 BeanA 类型的 Bean 已经神奇地变为了 BeanB 类型的 Bean。

```
I am BeanB
```

```
I am BeanB
```

从这个例子我们可以看出，BeanFactoryPostProcessor 有能力动态修改 XML 配置文件中的原始定义。读者也许要问，这种修改有何作用？下面要介绍的将 XML 配置文件中的某些常量提取到外部 properties 文件的 PropertyPlaceholderConfigurer 就是通过 BeanFactoryPostProcessor 实现的，它能动态读取 properties 文件的内容，然后替换掉相应的占位符。

3.10.6 使用外部属性文件

大多数情况下，我们直接在 Spring 的 XML 配置文件中装配好所有的 Bean。对于常量，通常我们是直接在 XML 配置文件中注入的，不过，某些时候，将常量放到一个外部的 properties 属性文件中也许更合适。最典型的例子是装配一个 DataSource，在 Spring 中，可以直接使用 DriverManagerDataSource，在 XML 中装配如下。

```
        <bean id="dataSource" class="org.springframework.jdbc.datasource. Driver
ManagerDataSource">
            <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
            <property name="url" value="jdbc:hsqldb:mem:test" />
            <property name="username" value="sa" />
            <property name="password" value="password" />
        </bean>
```

如果要修改 DataSource 的配置，就必须直接在 Spring 的 XML 配置文件中修改这个 Bean 的定义。一个更好的方式是将数据库的配置信息放到一个单独的 properties 文件中，例如，jdbc.properties，然后在 XML 配置文件中读取该 properties 文件的相应的值。这样，在发布应用程序时，用户可以很方便地根据自己的数据库配置修改 jdbc.properties 的内容，而不是直接修改 Spring 的 XML 文件。

幸运的是，Spring 提供了一个 PropertyPlaceholderConfigurer，它能够读取外部 properties 文件的内容，然后将相应的值传递给 Spring 容器。我们以上面提到的 DataSource 的配置为例，在 Eclipse 中新建一个 PropertyPlaceholder 工程，结构如图 3-9 所示。

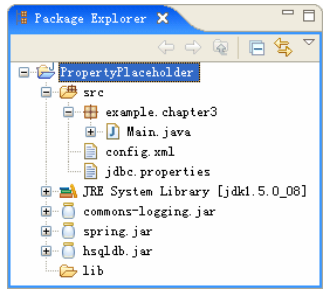


图 3-9

假如我们当前使用的是 HSQLDB 数据库，就需要编写如下的 jdbc.properties 属性文件。

```
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:mem:test
jdbc.username=sa
jdbc.password=password
```

然后，在 Spring 的 XML 配置文件中定义 DataSource 如下。

```
<bean id="dataSource" class="org.springframework.jdbc.datasource. Driver
ManagerDataSource">
    <property name="driverClassName" value="${jdbc.driver}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

读者可以很容易地看到，引用 properties 文件中的值采用“${变量名}”的形式，这样就避免了在 Spring 的 XML 配置文件中以硬编码的方式将常量注入到 Bean 中。

最后一步还需要声明一个类型为 PropertyPlaceholderConfigurer 的 Bean，用来告诉 Spring 容器在哪里可以找到这个 properties 文件。

```
<bean id="propertyConfigurer" class="org.springframework.beans.factory.
config.PropertyPlaceholderConfigurer">
    <property name="location" value="jdbc.properties" />
</bean>
```

PropertyPlaceholderConfigurer 是一个 BeanFactoryPostProcessor，这样，PropertyPlaceholderConfigurer就有机会在Spring容器读取XML配置文件之后、初始化Bean之前对Bean的定义进行修改，使用properties文件中相应的值代替“${变量名}”表达式。

如果有多个 properties 文件，就不能使用 location 属性，而是通过 locations 来配置多个 properties 文件。

```
<bean id="propertyConfigurer" class="org.springframework.beans.factory.
config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>jdbc.properties</value>
            <value>file.properties</value>
            <value>server.properties</value>
        </list>
    </property>
</bean>
```

我们在这个 PropertyPlaceholder 工程中使用 HSQLDB 作为测试数据库，HSQLDB 是一个纯 Java 编写的非常小巧的数据库，并且支持内存数据库，非常容易与 Java 应用程序集成，在第 5 章中，我们还会详细介绍它。要运行这个 PropertyPlaceholder 工程，需要将 hsqldb.jar 导入到工程的 Java Build Path 中，然后运行 main()方法测试。

使用外部属性文件看起来就好像是用 properties 文件来配置 Spring 的 XML 配置文件。当然，我们并不推荐将所有的常量都以 properties 文件的形式存放，然后在 XML 配置文件中引用。只有那些针对应用程序部署的特定配置信息才有必要将其单独放到外部 properties 文件中，例如，数据库配置信息、邮件服务器配置信息等，以便于修改。

3.10.7 国际化支持

国际化通常简称为 I18N，因为英文单词 Internationalization 中第一个字母 I 和最后一个字母 N 之间有 18 个字母。如果应用程序需要国际化支持（例如，对于不同国家或地区的用户显示不同的信息），就不能直接将文本硬编码在程序中。Java 平台本身就支持应用程序的国际化，它通过定义一个或多个属性文件保存应用程序中需要给用户显示的信息，每种语言或区域对应一个属性文件，从而实现根据用户语言及地区动态地绑定文本信息。语言和国家的代码与 ISO-639 及 ISO-3166 标准一致。

例如，我们想设计一个根据当前时间自动向用户问候的 GreetBean 组件，对于英文用户，我们需要一个 greeting_en.properties 的属性文件来保存不同时间的问候语。

```
morning=Good morning!
afternoon=Good afternoon!
evening=Good evening!
night=Good night!
```

而对于中国用户，则需要一个 greeting_zh_CN.properties 的属性文件来保存。

```
morning=\u65e9\u4e0a\u597d\uff01
afternoon=\u4e0b\u5348\u597d\uff01
evening=\u665a\u4e0a\u597d\uff01
night=\u665a\u5b89\uff01
```

其中，“\u65e9”之类的文本是经过编码的中文字符，稍后我们会介绍如何使用 JDK 提供的工具来实现编码转化。

Spring 的 ApplicationContext 对国际化的支持也正是依赖于 Java 平台的国际化机制，不过，我们不需要与 Java 平台本身提供的 ResourceBundle 打交道，而是直接通过 ApplicationContext 获得国际化信息。ApplicationContext 实现了 MessageSource 接口，为国际化提供了支持。

```
String getMessage(MessageSourceResolvable resolvable, Locale locale);
String getMessage(String code, Object[] args, Locale locale);
String getMessage(String code, Object[] args, String defaultMessage, Locale
locale);
```

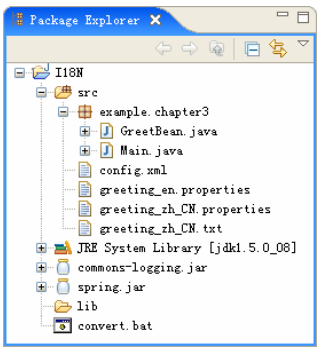


图 3-10

最常用的是后面两个 getMessage()方法，他们根据 locale 返回相应的文本，参数 args 用于格式化文本，例如，用实际值替换文本信息中的“{0}”、“{1,date}”等。为了避免 NoSuchMessageException 异常，可以使用第 3 个 getMessage()方法，提供一个默认的文本。

我们以下面的 I18N 工程为例，演示如何使用 Spring ApplicationContext 的国际化功能，如图 3-10 所示。

在使用 Spring 提供的国际化支持之前，我们还需要做两件事。首先，ApplicationContext 需要一个 MessageSource 的实例来实现国际化支持，它会在当前 XML 配置文件中查找一个名为“messageSource”的 Bean，因此，我们必须在 XML 配置文件中声明一个名为“messageSource”的 Bean。Spring 已经提供了一个 ResourceBundle MessageSource 的实现类，这个类的作用就是通过 Java 标准库的 java.util.ResourceBundle 来解析国际化信息。在 config.xml 中添加如下配置。

```
<bean id="messageSource" class="org.springframework.context.support.
ResourceBundleMessageSource">
    <property name="basename" value="greeting" />
</bean>
```

属性 basename 指定了国际化文本的“基本”文件名，对于本例的“greeting”而言，可能会有如下的几个文本文件。

（1）greeting.properties：默认的属性文本，当无法获得用户当前的语言和区域信息时，就是用这个默认的文本。

（2）greeting_en.properties：针对英语用户提供的国际化文本。

（3）greeting_en_US.properties：针对区域为美国的英语用户提供的国际化文本。

（4）greeting_zh_CN.properties：针对中国用户提供的国际化文本。

还可以添加更多的国际化属性文件，例如，“de_DE”、“fr”等。下一步是编写 GreetBean 组件，为了获取容器的引用，我们实现了 ApplicationContextAware 接口。

```
public class GreetBean implements ApplicationContextAware {
    private ApplicationContext context;
    private Locale locale = Locale.getDefault();
```

```
    public void setApplicationContext(ApplicationContext context) throws
BeansException {
        this.context = context;
    }
    public void setLocale(Locale locale) {
        this.locale = locale;
    }

    public String greet() {
        Calendar c = Calendar.getInstance();
        c.setTime(new Date());
        int hour = c.get(Calendar.HOUR_OF_DAY);
        if(hour<6)
            return context.getMessage("night", null, "Good night!", locale);
        if(hour<12)
            return context.getMessage("morning", null, "Good morning!", locale);
        if(hour<18)
            return context.getMessage("afternoon", null, "Good afternoon", locale);
        return context.getMessage("evening", null, "Good evening!", locale);
    }
}
```

由于 Java 的 ResourceBundle 对属性文件的限制，只能读取基本的 ASCII 字符和编码后的 UNICODE，所以，对于中文属性文本，需要用 JDK 提供的一个 native2ascii 工具，该命令行工具使用方法如下。

```
native2ascii -encoding encoding_name input output
```

我们编写一个 greeting_zh_CN.txt 文本文件，并且以 UTF-8 格式保存。

```
morning=早上好!
afternoon=下午好!
evening=晚上好!
night=晚安!
```

然后，输入命令：

```
native2ascii -encoding UTF-8 greeting_zh_CN.txt greeting_zh_CN.properties
```

现在，src 目录下自动生成了符合 ResourceBundle 要求的属性文件 greeting_zh_CN.properties。运行 main()方法测试，结果根据当前时间有所不同。如果计算机当前的 Windows 语言设置为中文，区域设置为中国，则运行结果如下。

```
2006-10-27 16:46:20 org.springframework.core.CollectionFactory <clinit>
信息: JDK 1.4+ collections available
```

```
...
下午好!
```

如果要将 GreetingBean 的 Locale 强制设置为 en_US，则修改配置文件 config.xml。

```
<bean id="greet" class="example.chapter3.GreetBean">
   <property name="locale" value="en_US" />
</bean>
```

再次运行，就可以看到打印出的信息已变为“Good afternoon!”。

3.10.8 定制属性编辑器

在前面介绍 Spring 强大的依赖注入特性时，读者一定注意到了，对于注入的<value>类型的值，在 Spring 的 XML 配置文件中均是以字符串形式存放的，无论该属性的类型是 String 还是 int、boolean，甚至是 Class、Locale、URL 类型。我们不禁要问，Spring 是如何将文本转化为实际的属性类型的呢？

实际上，这个从 String 到实际属性类型的转化并不是 Spring 的专利，而是 JavaBean 定义的规范。标准的 java.beans.PropertyEditor 接口定义了一个将 String 转化为任意类型的 setAsText()方法，java.beans.PropertyEditorSupport 则是实现了 PropertyEditor 接口的支持类。我们可以很方便地从 PropertyEditorSupport 派生出自定义的类型。

Spring 已经内置了许多 PropertyEditor 的实现，并且某些 PropertyEditor 已经由 BeanWrapperImpl 自动注册到容器中。除了基本类型注入外，常用的 PropertyEditor 如下。

（1）ByteArrayPropertyEditor：将 String 转化为一个 byte[]类型，默认为自动注册。

（2）CharArrayPropertyEditor：将 String 转化为一个 char[]类型，默认为自动注册。

（3）ClassEditor：将 String 转化为一个 Class 类型，String 必须为该类型的完整类名，默认为自动注册。

（4）CustomDateEditor：将 String 转化为一个 java.util.Date 类型，该类型默认没有被自动注册，需要手动配置。

（5）ResourceEditor：将 String 转化为一个 org.springframework.core.io.Resource 类型，使用 Spring 定义的表达式，例如，"file:C:/myfile.txt"、"classpath:myfile.txt"等。该 Resource 可以包含 File、URL、InputStream 等，默认为自动注册。

（6）InputStreamEditor：将以"http:// "、"ftp:// "等开头的合法的 URL 或以"classpath: "开头的文件转化为 java.io.InputStream 类型，注意 Spring 只负责创建 InputStream，关闭 InputStream 要由应用程序完成，默认为自动注册。

（7）LocaleEditor：将 String 转化为一个 Locale 对象，例如，"zh_CN"将转化为

Locale.CHINA，默认为自动注册。

（8）StringTrimmerEditor：可以对 String 实现 trim()操作，并能将一个空字符串转化为 null，该类型默认没有被自动注册，需要手动配置。

（9）URLEditor：将 String 转化为一个 URL 对象，可以是任意合法的 URL 字符串，或者以"classpath: "开头的文件，默认为自动注册。

如果需要编写自定义的属性编辑器，应该怎么办？例如，我们有一个订单 Order 对象，包含一个订单号 id 和一个联系人 Contact 对象。

```
public class Order {
    private String id;
    private Contact contact;

    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    public Contact getContact() { return contact; }
    public void setContact(Contact contact) { this.contact = contact; }
}
```

该 Contact 对象又由 name、address 和 zip 三个属性构成。

```
public class Contact {
    private String name;
    private String address;
    private String zip;

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

    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }

    public String getZip() { return zip; }
    public void setZip(String zip) { this.zip = zip; }

    public String toString() {
        return name + "; " + address + "; " + zip;
    }
}
```

如果要在 XML 配置文件中定义一个 Order 对象，我们不得不这么编写。

```
<bean id="order" class="example.chapter3.Order">
    <property name="id" value="12345678" />
```

```
        <property name="contact">
            <bean class="example.chapter3.Contact">
                <property name="name" value="Xuefeng" />
                <property name="address" value="北京市海淀区" />
                <property name="zip" value="100087" />
            </bean>
        </property>
    </bean>
```

对于用户而言，这样的 XML 配置太烦琐。考虑到 Contact 是 Order 的一个属性，如果能直接将 String 转化为 Contact 对象，就能大大简化 XML 配置，例如，可以按如下方式配置。

```
<bean id="order" class="example.chapter3.Order">
    <property name="id" value="12345678" />
    <property name="contact" value="Xuefeng; 北京市海淀区; 100087" />
</bean>
```

这就需要我们将字符串“Xuefeng; 北京市海淀区; 100087”转化为 Contact 对象。为了实现这个功能，我们需要编写一个自定义的属性编辑器 ContactEditor，该类从 PropertyEditorSupport 派生。

```
public class ContactEditor extends PropertyEditorSupport {
    public void setAsText(String text) throws IllegalArgumentException {
        String[] ss = text.split("[ ]*\\;[ ]*");
        if(ss.length!=3)
            throw new IllegalArgumentException(text);
        Contact contact = new Contact();
        contact.setName(ss[0]);
        contact.setAddress(ss[1]);
        contact.setZip(ss[2]);
        setValue(contact);
    }
}
```

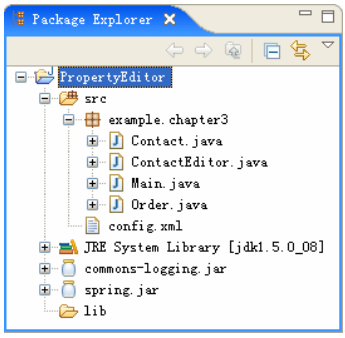


图 3-11

读者可以看到，编写一个自定义的属性编辑器是非常简单的，唯一需要覆写的方法是 setAsText(String)，在该方法中分析字符串，然后创建一个 Contact 对象，初始化后通过 setValue(Object)方法将其放入即可，Spring 容器就可以通过这个自定义的属性编辑器完成 String 到 Contact 对象的转化。整个工程的结构如图 3-11 所示。

另一个问题是 Spring 如何查找我们自定义的属性编辑器呢？当 Spring 遇到一个无法解析的类型时（例

如，example.chapter3.Contact 类型），就试图查找一个后缀为 Editor 的属性编辑器（例如，example.chapter3. ContactEditor）。如果没有找到，就抛出一个 TypeMismatchException 异常，提示无法将 String 转化为指定类型的对象。

如果没有遵守这种命名规则，例如，将其命名为 MyContactEditor，很不幸，我们只能告诉 Spring 这个 MyContactEditor 应该和什么类型的 Object 关联。这时，需要在 XML 配置文件中添加一个 CustomEditorConfigurer，这是一个 BeanFactory PostProcessor。

```
        <bean id="customEditorConfigurer" class="org.springframework.beans.factory.
config.CustomEditorConfigurer">
            <property name="customEditors">
                <map>
                    <entry key="example.chapter3.Contact">
                        <bean class="example.chapter3.MyContactEditor" />
                    </entry>
                </map>
            </property>
        </bean>
```

如果使用 BeanFactory 而不是 ApplicationContext，还必须在代码中调用 registerCustomEditor()方法注册自定义的属性编辑器。我们强烈建议使用一致的命名，避免在 XML 配置文件中声明自定义的属性编辑器。

3.10.9 发布和接收事件

Spring 的 ApplicationContext 容器可以支持发布和接收事件通知，其实现原理是基于标准的 Observer 模式。Spring 自身已经定义了 3 种事件：ContextClosedEvent、ContextRefreshedEvent 和 RequestHandledEvent，一般来说，Bean 不需要关心这些事件。

如果一个 Bean 需要向其他 Bean 发布事件通知，只需要调用 ApplicationContext 的 publishEvent()方法。下面的例子演示了如何发布一个事件。

```
        public class MessagePublisher implements ApplicationContextAware {
            private ApplicationContext context;
            public void setApplicationContext(ApplicationContext context) throws
BeansException {
                this.context = context;
            }
            public void sendMessage(final String message) {
                context.publishEvent(
                    new ApplicationEvent(this) {
                        public String toString() {
                            return message;
```

```
                }
            }
        );
    }
}
```

如前面所介绍，Bean 如果要获取容器的实例，则需要实现 ApplicationContextAware 接口，Spring 容器将把自身的引用注入给这个 MessagePublisher。

要接收其他 Bean 发布的事件，则仅需要实现 ApplicationListener 接口。

```
public class MessageListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent event) {
        System.out.println(event.toString());
    }
}
```

在 XML 配置文件中配置好以上两个 Bean（配置片段省略），然后在 Main 中编写测试代码，发布一个 String 消息。

```
MessagePublisher messagePublisher =
        (MessagePublisher) context.getBean("messagePublisher");
messagePublisher.sendMessage("Hello, everyone!");
```

运行应用程序，可以看到 MessageListener 打印出了其接收到的事件。

```
Hello, everyone!
```

如果作为事件发布的 Bean 也实现了 ApplicationListener 接口，则它将接收到自己发布的事件。可以通过 ApplicationEvent 的 getSource()方法获得该事件的发布者，以便忽略自己发布的事件。

由于 Spring 容器的事件发布机制是同步的，即调用 ApplicationContext.publishEvent()方法时，直到所有的 Bean 都处理完该事件，方法才会返回。因此，接收事件的 Bean 一定要迅速地处理完事件，以避免 publishEvent()方法长时间阻塞。

使用 Spring 容器提供的事件发布机制的好处是可以直接使用预定义的接口，不必做任何初始化工作。缺点是引入了 Spring 框架的专有接口，并且测试只能在 Spring 的容器内进行。如果要避免依赖 Spring 容器，可以考虑自己扩展 Java 标准库提供的 java.util.EventListener 接口，但这需要更多的编码工作。

3.11 分拆配置文件

对于大型项目，如果需要配置的 Bean 过多，会造成 XML 配置文件内容很多，不便

于各个小组之间的协作。这时可以考虑将一个 XML 配置文件拆为几个，Spring 支持用<import>导入一个配置文件。例如，某个小组负责 Web 模块的开发，并编写了 web-beans.xml，另一个小组则负责持久层的开发，并编写了 dao-beans.xml，则负责集成测试的小组可以在主配置文件中导入这两个单独的 XML 配置文件。

```
<beans>
   <import resource="web-beans.xml" />
   <import resource="dao-beans.xml" />
   <bean id="bean1" class="..." />
   <bean id="bean2" class="..." />
   ...
</beans>
```

需要注意的是，必须在定义任何<bean>之前用<import>导入其他 XML 配置文件，每个 XML 配置文件也必须是完整的包含 DTD 或 Schema 的 XML。

3.11.1 local的用法

由于可以导入其他的 XML 配置文件，因此，如果一个 Bean 需要注入某个依赖的 Bean（使用<property name="xxx" ref="refBean" />），则被注入的 Bean 可能被定义在导入的 XML 配置文件中。Spring 提供了 local 来明确指定注入的 Bean 必须在当前 XML 配置文件定义。

例如，若指定：

```
<property name="xxx">
   <ref local="refBean" />
</property>
```

则 Spring 容器在查找 refBean 时，仅在当前 XML 配置文件中查找，不在导入的 XML 配置文件中查找，若没有找到，则容器将抛出异常。

3.12 容器的继承

通常来说，一个应用程序只需要启动一个 Spring IoC 容器即可满足需求。不过，Spring 也提供了容器的继承功能，允许为容器指定其父容器，这样，子容器除了拥有自身定义的 Bean 以外，还可以直接访问其父容器中定义的所有的 Bean。反之，父容器则不能访问子容器中的任何 Bean，因为父容器对子容器一无所知，根本就没有持有子容器的引用。

实现具有继承功能的容器必须实现 HierarchicalBeanFactory 接口，对于基本的

BeanFactory 容器，只有 DefaultListableBeanFactory 和 XmlBeanFactory 具有 HierarchicalBeanFactory 接口，因此，可以实现容器的继承体系；对于 ApplicationContext 容器而言，由于 ApplicationContext 接口本身具有 HierarchicalBeanFactory 接口，因此，所有的 ApplicationContext 的实现都具有继承功能，使用起来更加方便。

我们仍以一个具体的例子来演示 Spring 容器的继承关系。在 Eclipse 中建立 Hierarchy 工程，结构如图 3-12 所示。

我们定义两个 XML 配置文件：config-1.xml 和 config-2.xml，分别作用于父容器和子容器。config-1.xml 配置文件中定义了 3 个 String 类型的 Bean，名称分别为“parent-1”、“parent-2”和“parent-3”：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
   <bean id="parent-1" class="java.lang.String">
      <constructor-arg value="Test parent-1 in config-1" />
   </bean>

   <bean id="parent-2" class="java.lang.String">
      <constructor-arg value="Test parent-2 in config-1" />
   </bean>

   <bean id="parent-3" class="java.lang.String">
      <constructor-arg value="Test parent-3 in config-1" />
   </bean>
</beans>
```

config-2.xml 配置文件中也定义了 3 个 String 类型的 Bean，名称分别为“child-1”、“child-2”和“child-3”。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
   <bean id="child-1" class="java.lang.String">
      <constructor-arg value="Test child 1 in config-2" />
   </bean>

   <bean id="child-2" class="java.lang.String">
```

```
        <constructor-arg value="Test child 2 in config-2" />
    </bean>

    <bean id="child-3" class="java.lang.String">
        <constructor-arg ref="parent-3" />
    </bean>
</beans>
```

注意到“child-3”引用了父容器配置文件中的 Bean“parent-3”，这是允许的，因为子容器可以访问父容器中定义的所有的 Bean。下一步，我们在 main()方法中先启动父容器，然后启动子容器，打印出子容器中的所有的 Bean。

```
public static void main(String[] args) {
    ApplicationContext parent = new ClassPathXmlApplicationContext("config-
1.xml");
    ApplicationContext child = new ClassPathXmlApplicationContext(
        new String[] {"config-2.xml"}, parent);
    for(String name : child.getBeanDefinitionNames()) {
        System.out.println("[Bean: " + name + "] " + child.getBean(name));
}
    System.out.println("----------");
    for(int i=1; i<=3; i++) {
        String name = "parent-" + i;
        System.out.println("[Bean: " + name + "] " + child.getBean(name));
    }
}
```

请注意，启动子容器时，需要调用其构造方法 ClassPathXmlApplicationContext(String[] configLocations, ApplicationContext parent)指定其父容器，否则，就需要手动调用 setParent(ApplicationContext parent)方法为其指定父容器。

我们在 main()方法中打印出子容器中定义的所有的 Bean，然后在子容器中访问父容器定义的 3 个 Bean，结果如下。

```
[Bean: child-1] Test child 1 in config-2
[Bean: child-2] Test child 2 in config-2
[Bean: child-3] Test parent-3 in config-1
----------
[Bean: parent-1] Test parent-1 in config-1
[Bean: parent-2] Test parent-2 in config-1
[Bean: parent-3] Test parent-3 in config-1
```

从结果可见，子容器中仅包含其自身 XML 配置文件中定义的 Bean，但是子容器可以访问父容器中的 Bean，因此可以从子容器中获得父容器中的 Bean，也可以在子容器的 XML 配置文件中注入父容器的 XML 配置文件中定义的 Bean。

读者要注意，Spring 中容器的继承和 Java 类的继承关系不同，后者是对类型的扩展，而前者仅仅是“父子”关系，图 3-13 很好地说明了我们上面的例子中两个容器之间的关系。

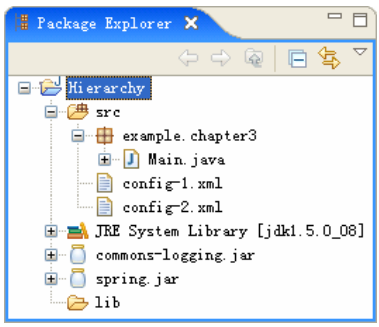


图 3-12

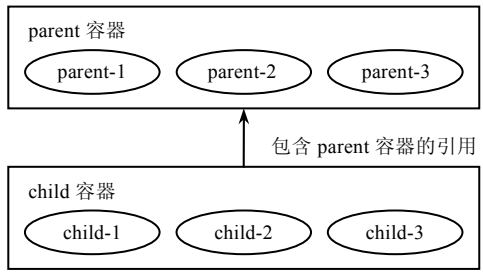


图 3-13

如果子容器和父容器中同时定义了相同名称的 Bean，则在子容器中查找该 Bean 时，返回的是哪个容器中的 Bean 呢？Spring 的容器总是先在自身定义的 Bean 中查找，若找到了，就直接返回该 Bean；若没有找到，才向父容器请求。所以，子容器总是优先返回自身定义的 Bean，父容器中相同名称的 Bean 永远无法从子容器中获得。

对于一般的应用程序，通常使用一个容器就足够了。这种继承体系的容器一般用于 Web 应用程序中。由于 JavaEE Web 应用程序中组件按照 Listener、Filter 和 Servlet 的顺序初始化，因此，某些时候必须首先在 Listener 中启动父容器，然后在 Servlet 中启动子容器。

3.13 使用XDoclet自动生成配置文件

对于大型项目来说，维护 XML 配置文件是一件非常困难的事情，因为要保证修改代码后对 XML 配置文件做必要的同步修改。我们希望能有一种根据源代码自动生成 XML 配置文件的方式，这样将大大降低维护的工作量。

幸运的是，XDoclet 正好符合这一要求，并且 XDoclet 还内置了对 Spring 的支持。XDoclet 是一种模版工具，它扫描源代码并提取特殊的注释，然后根据模版自动生成配置文件。XDoclet 最初的设计目的是为 EJB 自动生成接口和部署描述符，后来发展成一种通用的模版工具，可以支持 Spring、Hibernate、JDO、Portlet 等。

本节讨论如何应用 XDoclet 自动生成 Spring 所需的 XML 配置文件。

3.13.1 配置项目

由于 XDoclet 只能以 Ant 的扩展任务形式运行，因此必须使用 Ant 来构建项目，才能使用 XDoclet。我们将把上面的 IoC 工程改造为用 Ant 和 XDoclet 构建的 IoC_XDoclet 工程，其结构如图 3-14 所示。

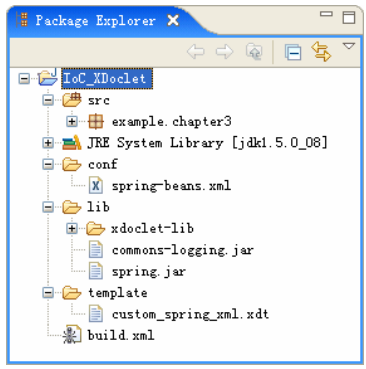


图 3-14

src 目录存放了所有源代码；conf 目录存放一个名为 spring-beans.xml 文件，它包含了非自动生成的所有 Spring 配置片断，XDoclet 在自动生成了 XML 配置文件后，会将该目录下的 spring-beans.xml 文件合并到最终的 XML 配置文件中；lib 目录除了必要的 commons-logging.jar 和 spring.jar 外，另外新建了一个 xdoclet-lib 目录，然后将 XDoclet-1.2.3 的 lib 目录下所有 jar 文件全部复制到该目录下。

请读者务必注意：为了支持 Java 5 代码，我们用 xjavadoc-1.5-snapshot050611.jar 替换了原有的 xjavadoc-1.2.3.jar。

template 目录存放用户自定义的模版，由于目前 XDoclet 还仅支持 Spring 1.x 版本，因此有必要修改 XDoclet 提供的模版，让其支持 Spring 2.0 版本。这个 custom_spring_xml.xdt 是从 xdoclet-spring-module-1.2.3.jar 中解压出的 /xdoclet/modules/spring/resources/spring_xml.xdt 文件。

由于使用 XDoclet 自动生成 XML 配置文件，因此，src 目录下不再需要手动编写 config.xml 文件。

在添加了 Ant 支持后，编写 build.xml 如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="HelloWorld" default="gen-spring-conf" basedir=".">
    <property name="src.dir"     value="src"     />
    <property name="lib.dir"     value="lib"     />
    <property name="conf.dir"    value="conf"    />
```

```
        <property name="build.dir"    value="bin"     />
        <property name="template.dir" value="template" />

        <!-- 定义 Classpath，包含了 XDoclet 的所有 jar 包 -->
        <path id="master-classpath">
            <fileset dir="${lib.dir}">
                <include name="**/*.jar"/>
            </fileset>
            <pathelement path="${build.dir}"/>
        </path>

        <!-- 编译 java 源代码 -->
        <target name="compile">
            <mkdir dir="${build.dir}"/>
            <javac destdir="${build.dir}" target="1.5" debug="on" debuglevel=
"lines">
                <classpath refid="master-classpath"/>
                <src path="${src.dir}"/>
            </javac>
        </target>

        <!-- 使用 XDoclet 生成配置文件 -->
        <target name="gen-spring-conf" depends="compile">
            <!-- 定义 Ant 任务 -->
            <taskdef name="springdoclet"
                classname="xdoclet.modules.spring.SpringDocletTask"
                classpathref="master-classpath"
            />
            <!-- 生成配置文件 -->
            <springdoclet
                destDir="${build.dir}"
                mergeDir="${conf.dir}"
                force="false"
                excludedtags="@version,@author,@todo"
            >
                <fileset dir="${src.dir}" includes="**/*.java" />
                <springxml
                    xmlencoding="UTF-8"
                    templateFile="${template.dir}/custom_spring_xml.xdt"
                    destinationFile="config.xml"
                />
            </springdoclet>
        </target>
    </project>
```

定义新的 Ant 任务

自动生成的配置文件存放目录

固定配置文件的存放目录

指定源代码文件

指定使用的模版文件

指定生成的配置文件名

XDoclet 提供的 spring_xml.xdt 模版仅支持 Spring 1.x 版本，可以从文件内容看出。

```
        <?xml version="1.0" encoding="<XDtConfig:configParameterValue paramName=
'Xmlencoding'/>"?>
        <!DOCTYPE beans PUBLIC
           "<XDtXml:publicId/>"
           "<XDtXml:systemId/>">
        <beans
          default-autowire="<XDtConfig:configParameterValue paramName="default
Autowire"/>"
          default-lazy-init="<XDtConfig:configParameterValue paramName=
"defaultLazyInit"/>"
          default-dependency-check="<XDtConfig:configParameterValue paramName=
"defaultDependencyCheck"/>"
        >
        ...
```

我们复制 spring_xml.xdt 到/conf/custom_spring_xml.xdt，并修改文件内容如下。

```
        <?xml version="1.0" encoding="<XDtConfig:configParameterValue paramName=
'Xmlencoding'/>"?>
        <beans xmlns="http://www.springframework.org/schema/beans"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
              default-autowire="<XDtConfig:configParameterValue paramName=
"defaultAutowire"/>"
              default-lazy-init="<XDtConfig:configParameterValue paramName=
"defaultLazyInit"/>"
              default-dependency-check="<XDtConfig:configParameterValue
paramName="defaultDependencyCheck"/>"
        >
        ...
```

通过我们的自定义模版生成的 XML 配置文件就符合 Spring 2.0 的要求。读者可以从本书的配套光盘中获得这个自定义的 custom_spring_xml.xdt 模版。下面我们详细讨论如何编写特定的 XDoclet 注释以便自动生成 Spring 配置文件。

3.13.2 定义Bean

为了定义一个<bean id="xxx" class="xxx.yyy.Zzz" />，在 class 的声明处编写如下的标准注释。

```
        /**
         * @spring.bean id="basicBean"
```

```
 */
public class BasicBean {
}
```

@spring.bean 定义了一个<bean>的声明，只需指定 id 或 name 的值，无需指定 class，XDoclet 能自动检测出完整类名。

运行 Ant，在自动生成的/bin/config.xml 中可以看到生成的<bean>定义。

```
<bean
   id="basicBean"
   class="example.chapter3.BasicBean"
>
</bean>
```

请注意，定义 Bean 时，只能针对具体类定义，不能在接口处定义 Bean，否则 XDoclet 生成的<bean>节点的 class 属性为接口类，Spring 的 IoC 容器将无法实例化该 Bean。

3.13.3 注入属性

注入属性也非常容易，在 set 方法的标准注释中加入@spring.property 定义，加上 value 或 ref。

```
/**
 * @spring.property value="100"
 */
public void setMaxSize(int size) {}
/**
 * @spring.property ref="basicBean"
 */
public void setBasic(BasicBean basicBean) {}
```

同样无需指定属性名称，XDoclet 能自动检测出来。

需要注意的是，在 get 方法前添加的注释是无效的，XDoclet 只认可 set 方法前的注释。要了解更多的 Spring 特有的注释，请参考附录 A 或者 XDoclet 的文档。

3.13.4 使用Merge功能

如果应用程序中使用的 Bean 不是自己开发的，而是现有的组件，如 JDK 或 Spring 自身的类，此时就无法通过编写特定的注释来自动生成配置文件。

解决这个问题的方法是手动编写一个固定的 XML 配置片段，这个固定配置文件的文件名必须是 spring-beans.xml，在 Ant 脚本中指定 mergeDir 后，XDoclet 就在这个目录

下查找 spring-beans.xml，如果找到，就将其嵌入到最终生成的 XML 配置文件中。

假定我们要定义一个 java.util.Date 对象，编写 spring-beans.xml 如下。

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

注意，该文件没有常规的 XML 头，只能定义<bean>节点。运行 Ant，生成的最终配置文件将会包含上述配置。

解决此问题的另一个办法是在一个固定的 XML 配置文件中使用<import>导入 XDoclet 动态生成的 XML 配置文件，这种方式较之使用 Merge 功能更加简单。

3.13.5 扩展XDoclet

前面我们虽然对 XDoclet 的模版做了一定的修改，使之符合 Spring 2.0 的 schema 验证规范，不过，XDoclet 还没有提供 Spring 配置文件的全部特性，例如，无法识别 scope、constructor-arg 中的 type，以及以 ref 构成的 list 等。幸运的是，XDoclet 的模版极具扩展性，可以轻松添加我们需要的新特性。

例如，对于 Spring 2.0 对 Bean 的 scope 定义，如果编写如下注释。

```
/**
 * @spring.bean id="basicBean" scope="prototype"
 */
public class BasicBean {
}
```

scope 属性将被忽略。要让 XDoclet 自动添加对 scope 属性的支持，我们可以修改 custom_spring_xml.xdt。参考现有的模版内容，很容易地添加对 scope 的支持。

```
...
<bean
    ...
    <XDtClass:ifHasClassTag tagName="spring.bean" paramName="scope">
       scope="<XDtClass:classTagValue tagName="spring:bean" paramName=
"scope" values="singleton,prototype,request,session,globalSession" default=
"singleton"/>"
    </XDtClass:ifHasClassTag>
>
...
```

再次运行 Ant，可以看到，scope 已经正确生成了。

```
<bean
    id="basicBean"
```

```
        class="example.chapter3.BasicBean"
        scope="prototype"
    >
```

在 IoC_XDoclet 项目中，我们对 custom_spring_xml.xdt 模版添加了 scope 和 listRef 的支持，读者可以尝试添加更多的 XDoclet 不支持的特性，例如，注入 Map 等。

也许不久，XDoclet 就会添加对 Spring 2.0 框架的支持，本节介绍的扩展 XDoclet 的目的不仅仅是为了实现 Spring 2.0 的配置自动化，更重要的意义在于如何主动对现有的技术和框架进行扩展，以满足项目开发的特定需求，而不是消极地等待新版本的支持。

在项目开发初期，配置 XDoclet 可能会花费一定的时间和精力，但是，随着组件越来越多，配置越来越复杂，开发人员可以切身体会到，通过脚本自动生成 XML 配置文件将节省大量的时间和维护成本。除了 Spring，XDoclet 还可以支持 Hibernate、JDO 等，因此，将 XDoclet 引入项目构建工具是完全值得的。

本书的附录 A 列出了详细的 XDoclet 使用说明，读者可以参考附录 A 以获得更详细的配置方法。

3.14 小结

本章主要讨论了如何应用 Spring 提供的功能强大的 IoC 容器以“组装积木”的组件配置方式来简化应用程序的开发。开发人员通过编写 Bean 组件并在 Spring 的 XML 配置文件中定义各组件及其依赖关系，就可以将组件的创建和销毁交给 Spring 的 IoC 容器管理，从而大大简化应用程序的开发，并有利于编写易于测试的组件。

我们详细讨论了如何在 XML 配置文件中定义 Bean 及其依赖关系，这是 Spring 框架设计的核心思想，读者必须深入理解依赖注入的思想，并在实践中体会依赖注入的强大威力。

我们还讲到了几种特殊的 Bean 和 Spring 容器提供的专有接口。大多数情况下，我们不会用到它们，应该优先考虑使用最基本的依赖注入，记住：Bean 的设计越简单越好。

本章还讨论了在实际项目中应用 XDoclet 来简化 Bean 的配置，在大型项目中，应用 Ant 配合 XDoclet 能大大提高开发效率，因此，花一定的时间来学习 Ant 和 XDoclet 是值得的。

第 4 章我们将学习在 Spring 中应用 AOP——面向方面编程。AOP 是对 OOP 的一种有益补充，在第 6 章中，读者还将看到，Spring 的声明式事务管理功能也是在 AOP 的基础上实现的。

第 4 章

使用Spring AOP

4.1 AOP入门

AOP（Aspect Oriented Programming）即面向切面编程，是近年来越来越流行的一种新的编程模式。AOP 的编程思想和 OOP 不同，作为对 OOP 的一种强有力的补充，本章将详细介绍 AOP 的概念、AOP 能做什么，以及如何在 Spring 容器中应用 AOP 实习强大的功能。

4.1.1 AOP概念

面向对象编程（OOP）已有几十年的历史。在软件开发领域，面向对象编程获得了巨大的成功。在软件系统中，OOP 通过将系统各功能封装为对象，通过对象的继承和多态，从而获得强大的代码复用能力，极大地提高了系统设计的能力。

而面向切面编程（AOP）则是最近几年开始流行的一种新的编程方式。AOP 最早由施乐公司帕洛阿尔托研究中心在 20 世纪 90 年代发明的一种编程范式，然而直到最近，AOP 才在 Java 平台上变得日益流行，这是因为 Java 平台能非常好地支持 AOP 的实现。

和 OOP 不同，AOP 以另一种方式来看待系统的结构。按照 AOP 的观点，一个系统被分解为不同的关注点（Concern），或者称之为切面（Aspect）。

通常，OOP 通过对象的封装、继承和多态能够很好地实现代码复用，从而获得优雅的系统设计。然而有些时候，我们无法通过 OOP 避免一些重复的冗余代码。例如，在一个 JavaEE 多层系统中，通常，业务组件除了实现核心的逻辑功能之外，还需要诸如日志记录、安全检查、事务管理等功能，如图 4-1 所示。

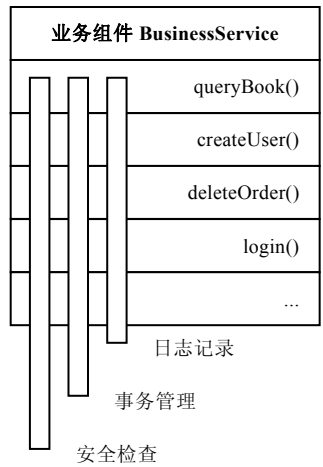


图 4-1

在一个 BusinessService 的组件中，我们不得不在每个业务方法中编写重复且类似的代码。

```
public void aBusinessOperation() {
    doSecurityCheck();
    Transaction tx = startTransaction();
    try {
        // 核心逻辑
        // TODO: ...
        tx.commit();
    }
    catch(Exception e) {
        tx.rollback();
    }
    log.info("aBusinessOperation done.");
}
```

这种重复的代码会出现在每个方法中。不幸的是，应用 OOP 很难将这些四处分散的代码模块化。

考察系统模型就可以发现，BusinessService 关注的是自身的核心逻辑，而整个系统还要求关注安全检查、事务管理、日志记录等切面，这种系统级的关注点是横跨多个核心逻辑的。为了实现系统级的关注点，就不得不在每一个逻辑方法中重复实现相同的或类似的系统关注点。

应用 OOP 很难剔出这些重复的代码。一个可能的解决方法是使用 Decorator（装饰器）模式，将系统关注点转移到另一个类中，或者通过派生一个子类，覆盖其每一个方法。

```
public class BusinessServiceEx extends BusinessService {
public void aBusinessOperation() {
    doSecurityCheck();
    Transaction tx = startTransaction();
    try {
        // 调用真正的业务组件:
        super.aBusinessOperation();
        tx.commit();
    }
    catch(Exception e) {
        tx.rollback();
    }
    log.info("aBusinessOperation done.");
}
```

然而，新的类仍然需要实现 BusinessService 的每个方法，因此仍然无法避免系统级

关注点出现在新类的每个方法中。一旦某个方法忘记了覆盖，则可能出现安全漏洞。

一种稍微可行的方法是通过某种方式自动地生成这个代理类。事实上，EJB 容器可以通过这个方法实现声明式事务管理，然而，这种代码生成的方法的限制很大，而且部署复杂。

AOP 以一种全新的视点来看待这些系统级的关注点。由于这些关注点横跨了多个核心逻辑，因此 AOP 将其称为 Aspect（切面）。AOP 通过将分散在各处的横切代码集中到一个独立的模块中，然后将这些横切模块以某种方式织入到核心逻辑的流程中，从而避免代码重复，提高整个系统的可维护性。

例如，对于上面的示例，应用 AOP 可以抽象出 3 个 Aspect：安全检查、事务管理和日志记录。如果将这 3 个 Aspect 集中到 3 个模块中，并织入到核心逻辑时，在调用 BusinessService 的每一个方法时，应用程序的执行流程如图 4-2 所示。

错误！

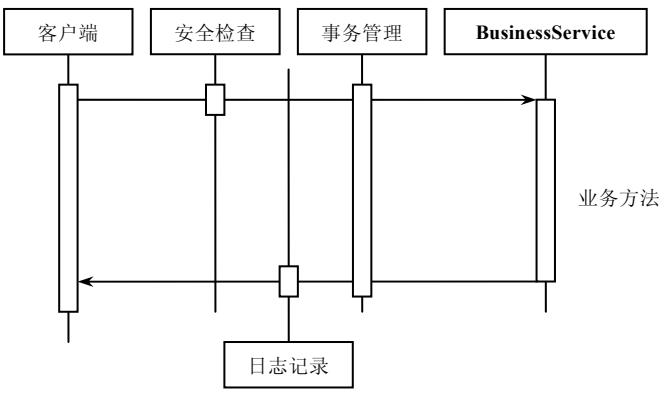


图 4-2

然而，AOP 并非解决问题的灵丹妙药。在实际应用中，有的系统级关注点非常容易通过 AOP 实现横切，例如，事务管理，这是由于分散在各处的事务代码几乎是完全相同的；有的系统级关注点就不那么容易实现，例如，日志记录。由于日志记录经常需要获取局部变量，这些内容对于系统维护常常具有重要意义，如果应用横切，则只能实现非常粗的记录（仅包括方法名及参数），无法实现细粒度的日志。因此，在应用 AOP 前，我们需要对 AOP 做全面的深入了解，并在实际项目中根据需要渐进地应用 AOP。

本章的例子大多以日志记录为例子，并不是为了鼓励以 AOP 的方式来编写日志模块，而是因为日志实现简单，能够非常方便地看到应用程序的运行流程，便于读者理解 AOP 的原理。

4.1.2 AOP的实现原理

如何实现这些横切模块在系统运行期能织入到核心逻辑中？这正是 AOP 技术需要

解决的问题，即各种 AOP 实现的原理。只要在调用目标对象的某一业务方法时，能够拦截该方法的调用，就可以将切面织入到应用程序的流程中。

在 Java 平台上，对于 AOP 模块的织入，有 3 种方式。

（1）编译期：切面在编译期织入，这类似于代码自动生成技术，不过，需要定义新的语法关键字并由特殊的编译器编译来实现。

（2）类装载器：在目标类被装载到 Java 虚拟机时，由一个特殊的类装载器对目标类的字节码进行增强。

（3）运行期：目标对象和切面都是标准的 Java 类，通过 Java 虚拟机的动态代理功能或 CGLIB 实现在运行期的动态织入。

类装载器的实现比较复杂，而且类装载器的继承体系可能会对应用程序的运行造成潜在的影响，尤其是在允许热部署的 Web 应用程序中，因此几乎没有采用这种方法实现 AOP 的框架。相比之下，编译期和运行期的实现方式较为普遍，最典型的代表分别就是 AspectJ 和 Spring AOP 的实现。

4.1.3 对比不同的AOP实现

AspectJ 和 Spring 的 AOP 分别是静态编译期实现和动态运行期织入的两种典型实现。让我们看看这两种实现的异同。

1．AspectJ 的 AOP 实现

AspectJ 是 Java 平台上最早的 AOP 实现，AspectJ 扩展了 Java 语法，它定义了额外的 AOP 语法关键字，因此，AspectJ 需要一个专门的编译器来生成 Java 字节码。实际上，AspectJ 对 AOP 的支持是在编译期实现的，AspectJ 提供的这个特殊的编译器就负责在编译期将切面加入到需要增强的类中。不过，一旦编译成为 class 文件，就和普通的 Java 类没有任何区别，因为 AspectJ 仅仅引入了新的语法关键字，没有对 class 字节码做任何改动。

AspectJ 也是目前最完善的 AOP 解决方案，它提供了 AOP 技术的所有特性，包括针对字段的拦截，目前，AspectJ 的最新版本是 5.0，其官方站点为 http://www.eclipse.org/aspectj/。AspectJ 还同时提供了一个 AJDT 插件以方便在 Eclipse 中开发，读者可以下载其最新版本。

2．Spring 的 AOP 实现

Spring 的 AOP 使用纯 Java 实现，不需要特殊的语法和专门的编译过程。Spring AOP 也不会更改类装载器，因此可以适用于 Web 应用程序。

Spring AOP 仅支持对方法的增强，这一点和 AspectJ 不同。Spring 的 AOP 追求的并

不是最完整的 AOP 实现方案，事实上，AspectJ 的 AOP 解决方案是最完整的，Spring AOP 的目标是让 AOP 技术在 Spring 的 IoC 容器中完美集成，因此，Spring AOP 通常都和 Spring 的 IoC 容器一起使用来解决企业应用开发中的常见问题。部分 AOP 功能也不被 Spring AOP 支持，例如，对于字段的拦截，尽管 Spring AOP 完全可以做到，但是 Spring 开发团队认为这样做破坏了对象的封装性，因此不提供这个功能。

Spring 框架提供的许多基础设施服务都依赖于 AOP 实现，例如，声明式事务管理，远程调用等，因此，AOP 在 Spring 中总是要和 Spring 的 IoC 容器整合使用，这是和其他 AOP 框架的一个重要区别，也正是这种整合，充分发挥了依赖注入和 AOP 这两种技术的威力，使得基于 Spring 的应用程序结构更加优雅，而编码则尽可能地被简化了。

虽然我们已经讲述了 AOP 的优点，AOP 技术看上去似乎比较神秘，但事实上，Spring 的 AOP 魔法并不神奇。下面，我们就先从原理入手，亲自动手编写一个最简单的 AOP 实现。

4.1.4 利用动态代理实现AOP

从 1.3 版本的 Java 平台开始，Java 虚拟机新增加了一个动态代理的功能。对于传统的面向对象设计，必须首先有实现了某个接口的具体类才能转型为接口类型，而动态代理则无需静态编译的具体类，就允许应用程序在运行期动态地创建一个实现了一个或多个接口的代理对象，并设置一个拦截器，调用该代理对象的任何方法都会被拦截器截获，从而可以动态地实现一个对象。

和继承机制不同的是，动态代理并没有具体地实现某个接口，它仅仅在运行期被创建，然后动态指定了其实现的接口。

动态代理的一个重要的作用是减少暴露给客户端的接口。例如，某个类同时实现了接口 A 和接口 B，但是对于某个客户端而言，我们仅希望提供接口 A 给它使用，尽管可以通过向上转型为接口 A，然而，客户端仍然可以使用强制转型为接口 B 来获得接口 B 的功能。通过动态代理的包装，就可以仅提供接口 A 给客户端，此时，客户端将无法强制转型为接口 B。

动态代理机制的核心是实现了 InvocationHandler 接口的拦截器。调用代理对象的方法及参数将被拦截器截获，方法返回值由拦截器返回。最简单的拦截器的实现是调用反射返回原始对象的方法调用结果。利用这一特性，就可以在调用对象的某一方法时将额外的功能动态织入进来。

我们设计了一个通用的 AopProxyFactory，通过一个 MethodBeforeAdvice 对一个对象动态地进行增强，即在调用实际对象的每个方法前，都首先调用 MethodBeforeAdvice

的 before()方法。

```
public class AopProxyFactory {
    public static Object createProxy(final Object target, final Method
BeforeAdvice methodBeforeAdvice) {
        return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(), // 代理类实现的接口
            new InvocationHandler() { // 指定拦截器
                public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                    // 首先调用切面的 before 方法:
                    methodBeforeAdvice.before(method, args, target);
                    // 然后调用实际对象的方法:
                    return method.invoke(target, args);
                }
            });
    }
}
```

现在，为了验证我们自己编写的 AopProxyFactory 是否真的能够对一个对象进行增强，我们在 Eclipse 中建立一个 MyAop 工程，结构如图 4-3 所示。

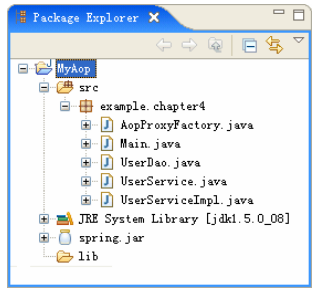


图 4-3

读者可以从本书的配套光盘中找到工程源代码，并直接在 Eclipse 中导入该项目。

这个工程模拟了一个用户管理的模块。UserDao 负责访问数据库，这里使用了一个简单的 Map 来模拟数据库。

```
public class UserDao {
    private Map<String, String> map = new HashMap<String, String>();
    public UserDao() {
        map.put("admin", "security");
        map.put("test", "123456");
    }
    public void create(String username, String password) {
```

```
        if(map.get(username)!=null)
            throw new RuntimeException("User exist!");
        map.put(username, password);
    }
    public void login(String username, String password) {
        String pw = map.get(username);
        if(pw==null || !pw.equals(password))
            throw new RuntimeException("Login failed.");
    }
    public void print() {
        System.out.println("User list:");
        Set<String> keySet = map.keySet();
        for(String key : keySet) {
            System.out.println(key);
        }
    }
}
```

UserService 定义了用户管理的接口，它仅有两个方法。

```
public interface UserService {
    void create(String username, String password);
    void login(String username, String password);
}
```

UserServiceImpl 则实现了 UserService 接口，它需要注入一个 UserDao 来完成业务方法。

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
    public void create(String username, String password) {
        userDao.create(username, password);
    }
    public void login(String username, String password) {
        userDao.login(username, password);
    }
}
```

现在，假定我们需要将日志记录这个功能动态地增加到 UserService 的每一个方法调用中。利用 AopProxyFactory 就可以动态地创建这个代理对象。

```
public static void main(String[] args) {
    // 创建 UserDao 和 UserService 对象并设置依赖注入：
```

```
    UserDao userDao = new UserDao();
    UserServiceImpl target = new UserServiceImpl();
    target.setUserDao(userDao);
    // 创建日志记录切面:
    MethodBeforeAdvice log = new MethodBeforeAdvice() {
        public void before(Method m, Object[] args, Object target) throws
Throwable {
            System.out.println("call method: " + m.getName());
        }
    };
    // 创建动态代理类:
    UserService userService = (UserService) AopProxyFactory.createProxy
(target, log);
    // 调用代理类的方法:
    userService.create("aop", "mypassword");
    userService.login("aop", "mypassword");
    // 检查 UserDao 中的数据:
    userDao.print();
}
```

我们仅仅借用了 Spring 定义的 MethodBeforeAdvice 接口来定义 log 对象，但是并没有用到 Spring 提供的任何现成的 AOP 功能。运行测试程序，可以看到日志对象已经打印出了每一个方法调用的记录。

```
call method: create
call method: login
```

为了验证动态代理是否真的调用了实际对象的方法，我们检查 UserDao 中的数据。

```
User list:
aop
test
admin
```

果然有通过动态代理添加的新用户“aop”!

现在，读者应该对 AOP 的原理有了一个基本的认识。事实上，Spring 框架提供了 JdkDynamicProxyFactory 来使用 JDK 动态代理，原理和上面的例子完全一样。用 Spring 提供的类来实现上述功能时，通过工厂类 ProxyFactory 来实现 AOP 的代码如下。

```
ProxyFactory factory = new ProxyFactory(target);
factory.addInterceptor(log);
UserService userService = (UserService)factory.getProxy();
```

当然，以编程的方式在 Spring 环境下实现 AOP 不但烦琐，也无法发挥 Spring IoC

容器依赖注入的威力，我们需要一种以 XML 或 Java 5 注解的方式来配置 AOP，然后直接使用代理对象即可。

从上面的例子可以看出，AOP 并非一种全新的技术，它仅仅是使用了更简单、更优雅的设计来实现传统的 Proxy、Decorator 等模式。对 AOP 框架而言，所要提供的就是一个可以灵活配置的装配 AOP 的方式。例如，对所有方法进行安全检查增强，但仅对以 create 开头的方法进行事务管理增强。

使用 Java 虚拟机的动态代理来实现 AOP 时的一个限制就是，只能对接口代理，而无法对一个实际类代理。例如，在上面的例子中，如果 UserServiceImpl 没有 UserService 接口，就无法实现代理类。但是，这通常不会是一个限制，因为针对接口编程是面向对象的一个非常好的实践，组件对外提供的服务也应该以接口的方式来声明。

在 Spring 中，默认情况下，即采用 JDK 动态代理机制来实现 AOP。如果要对具体的类代理，则 Spring 将使用 CGLIB 运行时的动态字节码生成技术来实现 AOP。这两种机制都可以由 Spring 自动选择，XML 配置文件的编写是一致的。绝大多数时候，我们应该针对接口编程，因此，采用默认的 JDK 动态代理机制来实现 AOP 完全能满足我们的需要。如果必须对具体类实现 AOP，则需要考虑设计是否合理。

值得注意的是，AOP 并非要取代 OOP。事实上，AOP 是对 OOP 的一种强有力的补充。在 JavaEE 系统中，仍然需要以 OO 的方式来设计和开发，但是，在处理某些“切面”的问题上，应用 AOP 可以大大简化设计，提高系统的模块化程度。

4.2 Spring AOP基础

前面我们通过一个具体的例子演示了如何使用 JDK 动态代理功能实现一个最简单的 AOP。Spring 的 AOP 也正是这一原理的应用，只不过将 AOP 的创建放到了 IoC 容器中。本节我们将详细讨论如何在 Spring 的 IoC 容器中使用 AOP。

4.2.1 术语解释

在开始 AOP 的深入了解前，我们先看一看 AOP 中经常使用的术语，这些术语大都出自最早的 AOP 研究团队——AspectJ 的创始人。

1．Aspect：切面

切面就是一个横跨多个核心逻辑的功能，或者称之为系统关注点，例如，日志记录、事务管理、安全检查等。以前，我们在一个组件的每个方法中重复编写这些代码。现在，我们将这些散落在各处的代码集中起来，放在一个模块中，这个模块（例如，日志记录）

就是一个切面。

2．Joinpoint：连接点

连接点就是定义在应用程序流程的何处插入切面的执行，最容易想到的就是在一个方法调用时，或者在访问一个字段时，或者在特定的异常抛出时。

3．Pointcut：切入点

切入点就是一组连接点的集合。例如，一个切面需要在调用每个以 query 开头的方法时执行，就可以用正则表达式（.*query.*）或通配符（query*）来表示这一组连接点，我们将其称之为切入点。

4．Advice：增强

在特定连接点上执行的动作，执行这个动作就相当于对原始对象的功能做了增强。

5．Introduction：引介

为一个已有的 Java 对象动态地增加新的接口。这个功能看起来有点玄乎，实际上也是 AOP 的一种功能，稍后我们还会讲到引介可以实现的一些功能。

6．Weaving：织入

织入就是将切面整合到程序的执行流程中。

还有一些术语是 Spring AOP 中常用的。

7．Interceptor：拦截器

拦截器是如何实现增强的一种方式。

8．Target Object：目标对象

目标对象就是真正执行核心逻辑的对象。没有 AOP 之前，我们可能不得不编写大量的重复代码散布于目标对象的每个方法中，有了 AOP 之后，重复代码就被集中到切面中，目标对象的代码被大大简化了。

9．AOP Proxy：AOP 代理

AOP 代理就是客户端持有的引用。由于 AOP 代理实现了与目标对象同样的接口，因此，客户端感觉不出使用这两者有何区别，但是却获得了动态增加的额外功能。

如果这些术语让你感到迷惑，不要紧，第一次接触 AOP 的人都会被这些术语弄得头晕脑胀，不需要立刻理解这些术语。实际上，读者只需要理清 3 个对象（**目标对象**、**切面**和 **AOP 代理**）**的关系**。目标对象就是我们自己编写的去除了重复代码的核心类，切面则是集中了重复代码的模块，然后在 Spring 的 IoC 容器中以合适的方式装配起来，构成一个 AOP 代理，交给客户端使用。

在4.1.4节我们自己实现的那个最简单的AOP例子中，目标对象就是UserServiceImpl对象，切面则是log对象，它在目标对象的每个方法调用前被执行。AOP代理对象则是我们通过动态代理机制创建的代理对象，它包装了目标对象并且知道应该在何处调用切面，并且这个AOP代理对象也实现了UserService接口，因此，客户端使用它和直接使用UserServiceImpl对象是一样的。

如果把我们自己创建AOP代理的方式改为在Spring IoC容器中配置，基本上就是一个完整的Spring AOP应用了。因此，下面要讲述的内容就是如何在Spring的IoC容器中将AOP代理对象装配出来。

4.2.2 在Spring中装配AOP

Spring大力提倡使用依赖注入的方式来装配Bean，对于AOP也不例外，依赖注入仍是极具威力的武器。Spring提供了ProxyFactoryBean来实现AOP的装配，免去了以编程方式生成AOP代理对象的烦琐步骤，这也是Spring AOP的设计目标之一。

在使用Spring AOP之前，还需要了解一些通用接口。AOP联盟（AOP Alliance）为了在Java语言中实现一致的AOP框架，定义了一系列AOP接口，以便为各种AOP框架的实现提供统一的接口。Spring的AOP实现也符合AOP联盟定义的接口，并且还做了一定的扩展。图4-4显示了Advice的继承体系。

错误！

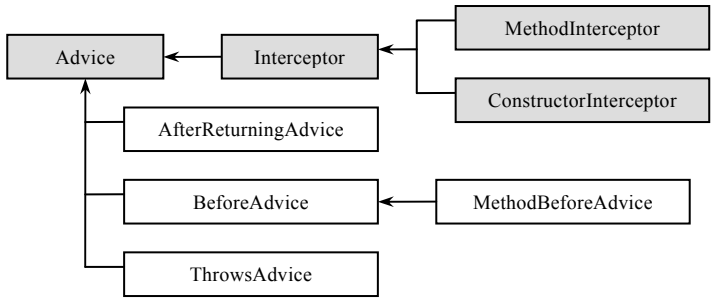


图4-4

Advice就是一个增强，可以在某个连接点上被执行。在Java中， Advice本质上就是一个拦截器，可以拦截方法调用，然后执行自己的一段代码来增强应用程序的逻辑，但是Advice并不能决定拦截哪些方法，这是由Pointcut决定的，关于Pointcut我们将在稍后讲解。

图4-4中灰色标记的是AOP联盟定义的接口，白色标记的则是Spring扩展的接口。

常用的Advice如下。

（1）MethodBeforeAdvice：在一个方法执行开始前增强。

（2）AfterReturningAdvice：在一个方法执行完毕后增强。

（3）ThrowsAdvice：在一个方法执行期抛出异常时增强。

（4）MethodInterceptor：在一个方法执行前后增强。

对于前 3 种 Advice，我们可以在方法执行前后进行一些增强，但是无法阻止方法被执行，也无法改变方法的返回值。而 MethodInterceptor 则完全可以控制整个方法的执行与否，甚至可以修改方法的返回值，因此功能最为强大，可以实现前 3 种 Advice 的所有功能。

我们在选择 Advice 的时候，使用尽量简单的 Advice 类型来实现需要的功能。例如，如果 MethodBeforeAdvice 能满足要求，就不要使用 MethodInterceptor，这样能避免由于忘记调用 invocation.proceed()而导致应用程序流程错误。

我们把 MyAOP 工程复制一份，命名为 BasicAop，然后以 Spring 的 AOP 方式来实现它。在 Eclipse 中，该工程的结构如图 4-5 所示。

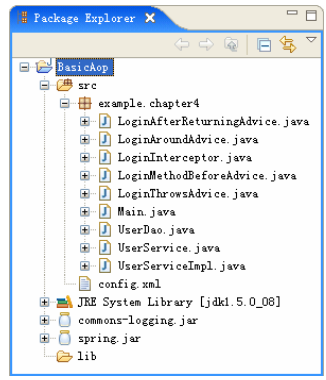


图 4-5

4.2.3 编写Advice

我们的第一个 Advice 仍以日志记录为例，先创建一个 LoginMethodBeforeAdvice，用于记录所有 login()方法的调用。

```
public class LoginMethodBeforeAdvice implements MethodBeforeAdvice {
  public void before(Method m, Object[] args, Object target) throws Throwable {
      System.out.println("[LoginMethodBeforeAdvice] User " + args[0] + " try
to login...");
    }
}
```

4.2.4 使用ProxyFactoryBean装配AOP

有了目标 Bean 和 Advice 后，就可以编写 config.xml 配置文件，将其在 Spring 的 IoC 容器中装配起来。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
```

```
    <bean id="userServiceTarget" class="example.chapter4.UserServiceImpl">
        <property name="userDao">
            <bean class="example.chapter4.UserDao" />
        </property>
    </bean>

    <!-- 定义 MethodBeforeAdvice -->
    <bean id="loginMethodBeforeAdvice"
        class="example.chapter4.LoginMethodBeforeAdvice" />

    <!-- 配置 AOP -->
    <bean id="userService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <!-- 注入目标对象 -->
        <property name="target" ref="userServiceTarget" />
        <!-- 注入 Advice/Interceptor -->
        <property name="interceptorNames">
            <list>
                <value>loginMethodBeforeAdvice</value>
            </list>
        </property>
    </bean>
</beans>
```

在 Spring 的 IoC 容器中，将 Advice 织入到目标对象需要依靠 ProxyFactoryBean 来完成，由于 ProxyFactoryBean 是一个工厂 Bean，因此，它返回的对象是封装了目标对象 userServiceTarget 和 loginMethodBeforeAdvice 的 AOP 代理。target 属性即是需要增强的目标对象，还可以使用 targetName 来设置目标对象的 id，而不是 ref 引用。interceptorNames 属性可以注入一系列 Advice，这里我们只有一个 loginMethodBeforeAdvice。我们没有设置 interfaces 属性，因此，ProxyFactoryBean 将返回 userServiceTarget 对象所具有的所有接口。如果需要返回某些指定的接口，则需要设置这一属性。在 XML 配置文件中装配好 AOP 之后，我们就获得了增强后的 AOP 对象 userService，客户端使用它和直接使用原始的 userServiceTarget 没有任何区别。

现在，编写 main()方法来测试我们在 Spring 的 IoC 容器中配置的 AOP。

```
public static void main(String[] args) {
  ApplicationContext context = new ClassPathXmlApplicationContext ("config.xml");
   UserService userService = (UserService) context.getBean("userService");
   userService.login("admin", "security");
   try {
       // nobody 用户不存在:
       userService.login("nobody", "invalid-password");
```

```
        }
        catch(Exception e) {}
    }
```

运行测试程序，可以看到 loginMethodBeforeAdvice 打印出了每次的方法调用。

```
[LoginMethodBeforeAdvice] User admin try to login...
[LoginMethodBeforeAdvice] User nobody try to login...
```

我们不仅希望记录用户尝试登录的信息，还希望记录用户是否登录成功。AfterReturningAdvice 和 ThrowsAdvice 正好派上用场，因此，我们继续编写以下两个 Advice，其中，一个是 LoginAfterReturningAdvice，用于记录已经成功登录的用户。

```
    public class LoginAfterReturningAdvice implements AfterReturningAdvice {
        public void afterReturning(Object returnValue, Method method, Object[] args,
Object target) throws Throwable {
            System.out.println("[LoginAfterReturningAdvice] User " + args[0] + "
logged in successfully.");
        }
    }
```

如果用户登录失败，login()方法将会抛出异常，因此，LoginThrowsAdvice 正好适合记录用户登录失败的情况。

```
    public class LoginThrowsAdvice implements ThrowsAdvice {
        public void afterThrowing(Method m, Object[] args, Object target, Throwable
subclass) {
            System.out.println("[LoginThrowsAdvice] An exception occur: " +
subclass.getClass().getSimpleName());
        }
    }
```

这里有一点比较特别，就是 ThrowsAdvice 并没有定义任何方法。查看 Spring 的 API 文档，可以看到文档中是这么写的。

> 实现类应当按照下面的形式实现方法。
> afterThrowing([method], [args], [target], Throwable subclass)
> 实现类应该实现一个 afterThrowing 方法，但是，除了最后一个参数 Throwable 是必须的以外，前 3 个参数可以视情况而定，Spring 根据反射来查找符合上述定义形式的方法。

例如，如果不需要知道方法及方法参数，但需要知道当前方法所属的调用对象，就可以定义。

```
    afterThrowing(Object target, Throwable subclass)
```

如果还需要获得方法参数，就加上 args 参数。

```
afterThrowing(Object[] args, Object target, Throwable subclass)
```

设计这样的接口的目的是为了不强迫实现类去实现每个可能的 afterThrowing 方法，从而简化实现。但是，由于缺少了编译期的接口契约检查，如果有错误，就只能在运行期才能发现。

在 config.xml 中添加上述两个 Advice 的定义。

```
<bean id="loginAfterReturningAdvice"
   class="example.chapter4.LoginAfterReturningAdvice" />
<bean id="loginThrowsAdvice" class="example.chapter4.LoginThrowsAdvice" />
```

然后，将其加入到 ProxyFactoryBean 的 interceptorNames 中。

```
<bean id="userService"
   class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="interceptorNames">
      <list>
         <value>loginMethodBeforeAdvice</value>
         <value>loginAfterReturningAdvice</value>
         <value>loginThrowsAdvice</value>
      </list>
   </property>
   <property name="target" ref="userServiceTarget" />
</bean>
```

再次运行测试程序，可以看到如下的输出。

```
[LoginMethodBeforeAdvice] User admin try to login...
[LoginAfterReturningAdvice] User admin logged in successfully.
[LoginMethodBeforeAdvice] User nobody try to login...
[LoginThrowsAdvice] An exception occur: RuntimeException
```

我们注意到，如果方法正常返回，则 ThrowsAdvice 将不会被执行，因为没有异常抛出，自然不会引发 ThrowsAdvice 的执行；如果方法抛出了异常，则 ThrowsAdvice 将被执行，而 AfterReturningAdvice 就不能被执行了，这一点也很容易理解，因为当异常抛出时，方法是没有任何返回值的，自然无法执行 AfterReturningAdvice。

可以有多个 ThrowsAdvice，当异常抛出时，将按顺序执行每个 ThrowsAdvice，这样可以使每个 ThrowsAdvice 都能各自处理其关心的特定异常。

现在，我们讨论了 MethodBeforeAdvice、AfterReturningAdvice 和 ThrowsAdvice，还有一种 MethodInterceptor，通常称为环绕通知。正如前面所说，环绕通知不仅能实现这 3 种 Advice 的所有功能，还可以控制目标对象的方法是否执行，因此功能更加强大。

一个最简单的 MethodInterceptor 的实现如下。

```
public class LoginAroundAdvice implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        try {
            // 这里可以实现 MethodBeforeAdvice 的功能...
            Object ret = invocation.proceed();
            // 这里可以实现 AfterReturningAdvice 的功能...
            return ret;
        }
        catch(Throwable t) {
            // 这里可以实现 ThrowsAdvice 的功能...
            throw t;
        }
    }
}
```

调用 invocation.proceed()方法就调用了目标对象的目标方法，并获得方法的返回值，然后返回给客户端即可。在调用 proceed()方法之前编写代码就能实现 MethodBeforeAdvice 的功能；在调用 proceed ()方法之后编写代码就能实现 AfterReturningAdvice 的功能；如果将 proceed()方法调用放在 try { ... }语句中并捕获所有 Throwable，就能实现 ThrowsAdvice 的功能。因此，MethodInterceptor 完全可以取代前 3 种 Advice，不过，如果忘记了调用 invocation.proceed()方法，则目标对象的方法将不会被执行。这一点可以用来控制是否需要调用目标对象的方法。例如，出于某种原因，我们希望 admin 用户不能直接登录，就可以编写。

```
public class LoginAroundAdvice implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        Object[] args = invocation.getArguments();
        if(args[0].equals("admin"))
            throw new SecurityException("Rejected");
        return invocation.proceed();
    }
}
```

将 LoginAroundAdvice 加入到 config.xml 中，然后修改 ProxyFactoryBean，将 loginAroundAdvice 也加入到 interceptorNames 中，运行测试程序，可以看到当试图以"admin"登录时，抛出了 SecurityException。

```
Exception in thread "main" java.lang.SecurityException: Rejected
```

由于 MethodBeforeAdvice、AfterReturningAdvice 和 ThrowsAdvice 这 3 种 Advice 的功能并不互相重叠，因此，在 ProxyFactoryBean 中配置它们时，其排列顺序一般不会影

响到执行顺序。但是，如果加入了 MethodInterceptor，并且放在最前面，则 MethodInterceptor 的实现将影响到这 3 种 Advice 是否执行。例如，若 MethodInterceptor 根本没有调用 invocation.proceed()方法，则这 3 种 Advice 将都不会被执行。

读者可能已经注意到了，我们讨论了各种 Advice 的实现，但是，却并没有指定 Advice 应当作用于目标对象的哪些方法上。如果在 main()方法中添加一个 userService.create()方法调用，则针对 login()方法编写的各种 Advice 仍将执行，这显然是不符合逻辑的，因为这些 Advice 目的是对 login()方法增强，而不是对 create 方法增强。更进一步地讲，我们还不知道如何让 Advice 应用于指定的某些方法，例如，仅应用于以 query 开头的方法。这时，Advisor 就派上用场了。

4.2.5 编写Advisor

Spring 定义了 Advisor 的概念，它包含了一个 Advice，并且定义了如何将 Advice 织入到目标对象。其中，最重要的 Advisor 便是 PointcutAdvisor。PointcutAdvisor 包含了一个 Advice 和一个 Pointcut，Pointcut 用于判断方法是否符合设定的条件，若符合，则执行相应的 Advice。因此，通过 PointcutAdvisor，我们便能指定切面应该作用在哪些方法上。

可以把 Advisor 简单地理解为一个 Advice 和一个 Pointcut 的组合。由于 Advisor 依赖 Pointcut 来判断是否应该将切面织入到目标方法中，因此，我们首先来看几个重要的 Pointcut 实现。一般来说，每个 Pointcut 实现都会有一个对应的 Advisor 实现，以方便同时指定 Pointcut 和 Advice，所以我们通常不用自己实现 Pointcut，只需要直接使用 Spring 提供的 Advisor 即可。

1．NameMatchMethodPointcut

NameMatchMethodPointcut 非常容易设置，它提供了两个非常方便的方法来定义方法切入点。

```
setMappedName(String mappedName)
setMappedNames(String[] mappedNames)
```

回到我们前面的例子，如果仅仅希望将 Advice 应用于 UserService 的 login()方法，可以定义一个 NameMatchMethodPointcutAdvisor。

```
<bean id= "loginBeforeMethodAdvisor" class= " org.springframework.aop.support.
NameMatchMethodPointcutAdvisor">
    <property name="mappedName" value="login" />
    <property name="advice" ref="loginBeforeMethodAdvice" />
</bean>
```

然后，将 loginBeforeMethodAdvisor 替代 loginBeforeMethodAdvice 注入到 Proxy

FactoryBean 的 interceptorNames 中。事实上，ProxyFactoryBean 的 interceptorNames 可以接受 Advice 和 Advisor 两种对象。

如果要匹配多个方法，将一个 String 数组注入到 mappedNames 属性中即可。

此外，还可以使用通配符来简化设置。例如，若希望将 Advice 应用于所有以 query 开头的方法，则可以注入“query*”到 mappedName 属性中。

2．AbstractRegexpMethodPointcut

AbstractRegexpMethodPointcut 根据正则表达式来计算切入点。如果对于正则表达式很熟悉，使用正则表达式可能会比较方便。

和 NameMatchMethodPointcut 不同，AbstractRegexpMethodPointcut 要求匹配完整的类名和方法名。例如，对于 UserService 的 login 方法，一个合乎条件的正则表达式为“.*login”。

Spring 提供了两种 AbstractRegexpMethodPointcut 的实现，对于 JDK 1.4 或更高版本，Spring 使用 JdkRegexpMethodPointcut；对于 JDK 1.4 以前的版本，由于还没有加入处理正则表达式的类，因此只能使用 Perl5RegexpMethodPointcut，这要求当前的 ClassPath 必须包含 Jakarta ORO 库。

AbstractRegexpMethodPointcut 对应的 Advisor 是 RegexpMethodPointcutAdvisor，RegexpMethodPointcutAdvisor 会根据当前 JDK 版本决定使用 JdkRegexpMethodPointcut 还是 Perl5RegexpMethodPointcut。通过注入一个 pattern 属性，就可以指定切入点。

```
        <bean id= "loginBeforeMethodAdvisor" class= " org.springframework.aop.support.
RegexpMethodPointcutAdvisor">
            <property name="pattern" value=".*login" />
            <property name="advice" ref="loginBeforeMethodAdvice" />
        </bean>
```

下面的例子扩展了上一个 BasicAop 的工程，演示如何使用 Advisor 来截获指定的方法。在 Eclipse 中，创建 BasicAop_Advisor 工程，结构如图 4-6 所示。

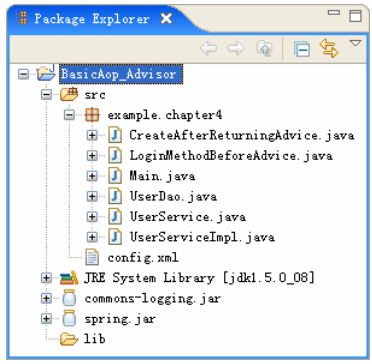


图 4-6

新增 CreateAfterReturningAdvice，用于对 create()方法增强。

```
public class CreateAfterReturningAdvice implements AfterReturningAdvice {
    public void afterReturning(Object returnValue, Method method, Object[] args,
Object target) throws Throwable {
        System.out.println("[CreateAfterReturningAdvice] New user created: "
+ args[0]);
    }
}
```

LoginMethodBeforeAdvice 不变。下面需要做的就是将 Advice 封装为 Advisor，这样就同时指定了 Advice 应该作用的方法。

首先，我们使用 NameMatchMethodPointcutAdvisor 来定义 LoginMethodBeforeAdvice 应当仅仅作用于 login()方法。

```
<bean id="loginBeforeAdvisor" class="org.springframework.aop.support.
NameMatchMethodPointcutAdvisor">
    <!--  指定切入方法为 login -->
    <property name="mappedName" value="login" />
    <!-- 指定 Advice -->
    <property name="advice">
        <bean class="example.chapter4.LoginMethodBeforeAdvice" />
    </property>
</bean>
```

然后，我们使用 RegexpMethodPointcutAdvisor 来定义 CreateAfterReturningAdvice 应当作用于所有以 create 开头的方法。

```
<bean id="createAfterAdvisor" class="org.springframework.aop.support.
RegexpMethodPointcutAdvisor">
    <!-- 指定切入方法为.*create.*，拦截所有以 create 开头的方法 -->
    <property name="pattern" value=".*create.*" />
    <!-- 指定 Advice -->
    <property name="advice">
        <bean class="example.chapter4.CreateAfterReturningAdvice" />
    </property>
</bean>
```

最后，将这两个 Advisor 装配起来。

```
<bean id="userService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="interceptorNames">
        <list>
            <value>loginBeforeAdvisor</value>
            <value>createAfterAdvisor</value>
```

```
        </list>
    </property>
    <property name="target" ref="userServiceTarget" />
</bean>
```

为了验证这两个 Advisor 是否针对特定的方法进行了增强，我们在 main()方法中调用 UserService 的不同方法。

```
public static void main(String[] args) {
  ApplicationContext context = new ClassPathXmlApplicationContext ("config.xml");
   UserService userService = (UserService) context.getBean("userService");
   userService.create("newuser", "newpassword");
   userService.login("admin", "security");
   userService.create("new_user", "a-test");
}
```

运行结果如下。

```
[CreateAfterReturningAdvice] New user created: newuser
[LoginMethodBeforeAdvice] User admin try to login...
[CreateAfterReturningAdvice] New user created: new_user
```

可见，create()方法和 login()方法分别被 createAfterAdvisor 和 loginBeforeAdvisor 增强，这样我们就实现了针对特定方法的增强，而不是对所有方法都进行增强。

回顾一下在 Spring 中创建 AOP 对象的过程，其实我们需要关注的主要是目标 Bean，Advice、Pointcut 和最终的 AOP 代理对象，它们的结构如图 4-7 所示。

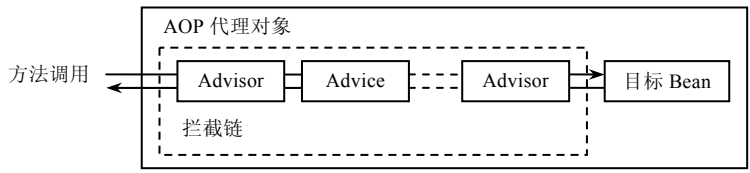
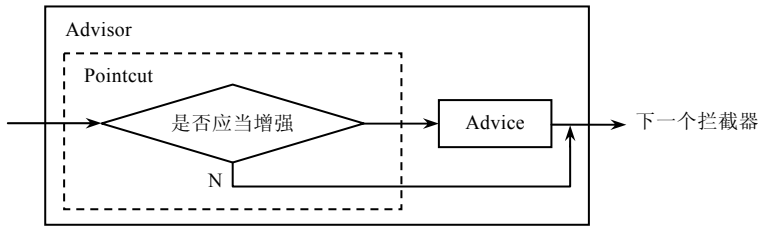


图 4-7

AOP 代理对象实际上包含了一个由 Advisor 和 Advice 组成的拦截链。当客户端调用 AOP 代理对象的某个方法时，拦截链上的每个 Advisor 和 Advice 都有机会对该方法调用进行增强。不同的是，Advice 总是增强所有的方法，而 Advisor 同时包含了 Advice 和 Pointcut。因此，Advisor 会通过 Pointcut 计算是否应该调用 Advice 对某个方法进行增强，如图 4-8 所示。

从图 4-8 可以看出，Spring 的 AOP 原理就这么简单！



错误！

图 4-8

4.3 使用自动代理

前面已经讲述了如何将一个目标 Bean 通过 AOP 变成一个代理 Bean，这样，客户端就可以使用代理 Bean，从而获得通过 AOP 实现的额外功能。不过，客户端同样也能绕过代理 Bean 直接访问目标 Bean，只要客户端知道目标 Bean 的 id。为了向客户端完全屏蔽目标 Bean，Spring 还提供了自动代理的功能，通过自动代理，可以实现下面两个功能。

（1）自动为多个目标 Bean 实现 AOP 代理。

（2）避免客户端直接访问目标 Bean。

Spring 的自动代理功能实际上是由 BeanPostProcessor 实现的。在容器载入 XML 配置文件后，具有自动代理功能的 BeanPostProcessor 就可以修改 Bean 的定义，将所有需要实现代理的目标 Bean 全部修改为代理 Bean，而 id 不变。这样，容器就不直接持有目标 Bean 的引用，而仅持有代理 Bean 的引用，客户端也就无从直接访问目标 Bean 了。

在第 3 章已经讲到了，使用 ApplicationContext 时，容器可以直接检测到定义在 XML 配置文件中的 BeanPostProcessor，然后自动调用它们，而使用 BeanFactory 则需要手动编程来添加 BeanPostProcessor，因此，在本节中，我们总是以 ApplicationContext 为例，在 XML 配置文件中定义实现自动代理的 BeanPostProcessor。

Spring 提供了几种常用的实现自动代理的 BeanPostProcessor。

（1）BeanNameAutoProxyCreator：根据 Bean 的 id 或 name 属性来查找目标 Bean 并自动为其代理。

（2）DefaultAdvisorAutoProxyCreator：根据当前容器中的 Advisor 决定每个 Bean 是否可以被代理，如果可以，就自动创建代理，并自动织入所有可用的 Advisor。

（3）AspectJInvocationContextExposingAdvisorAutoProxyCreator：根据 AspectJ 的语法规则来决定是否为一个 Bean 创建代理。

（4）AnnotationAwareAspectJAutoProxyCreator：根据 AspectJ 的注解来决定是否为一个 Bean 创建代理。

本节我们只讨论 BeanNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator，对于 AnnotationAwareAspectJAutoProxyCreator 将在稍后讨论。

4.3.1 使用BeanNameAutoProxyCreator

我们将 BasicAop_Advisor 工程复制一份，命名为 AopAutoProxy，然后利用 BeanNameAutoProxyCreator 来简化 AOP 的配置。

AopAutoProxy 工程的结构和 BasicAop_Advisor 工程并无区别，如图 4-9 所示。

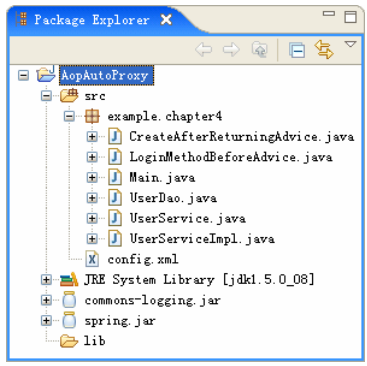


图 4-9

现在我们仅需要修改 config.xml 配置文件，将原来通过 ProxyFactoryBean 手动创建 AOP 代理对象的方式改为通过 BeanNameAutoProxyCreator 自动创建。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
    <bean id="userService" ... />
    <bean id="loginBeforeAdvisor" ... />
    <bean id="createAfterAdvisor" ... />

    <!-- 配置 BeanNameAutoProxyCreator -->
    <bean id="beanNameAutoProxy" class="org.springframework.aop.framework.
autoproxy.BeanNameAutoProxyCreator">
        <!-- 指定 Bean 的名称 -->
        <property name="beanNames" value="userService" />
        <!-- 指定 Interceptor -->
        <property name="interceptorNames">
            <list>
                <value>loginBeforeAdvisor</value>
                <value>createAfterAdvisor</value>
```

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

虽然使用 BeanNameAutoProxyCreator 和 ProxyBeanFactory 并无太多区别，但是 BeanNameAutoProxyCreator 允许通过通配符指定一系列目标 Bean，本例中我们指定的是“userService”。如果设置为“*Service”，就可以将以“Service”结尾的 Bean 全部自动代理为 AOP 对象，因此，对于需要配置多个 AOP 代理对象的 Bean，各 AOP 代理对象的配置均一致，使用 BeanNameAutoProxyCreator 就可以简化配置。

除 BeanNameAutoProxyCreator 外，DefaultAdvisorAutoProxyCreator 是一种更加强大的自动代理。

4.3.2 使用DefaultAdvisorAutoProxyCreator

使用 DefaultAdvisorAutoProxyCreator 只需要声明一行配置即可完成自动代理功能。

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

DefaultAdvisorAutoProxyCreator 将找出所有的 Advisor（注意：仅 Advisor 会被列举出来，Advice 都将被忽略）和普通的 Bean，然后根据 Advisor 计算它们能够适用于哪些 Bean，凡是能被适用的 Bean 都将被自动创建 AOP 代理，整个过程全部由 DefaultAdvisor AutoProxyCreator 完成，完全不需要手动配置任何 AOP 代理对象。

如果想将上例的 BeanNameAutoProxyCreator 改为 DefaultAdvisorAutoProxyCreator，删除 BeanNameAutoProxyCreator，然后添加 DefaultAdvisorAutoProxyCreator。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
>
    <bean id="userService" class="example.chapter4.UserServiceImpl">
        <property name="userDao">
            <bean class="example.chapter4.UserDao" />
        </property>
    </bean>

    <bean id="loginBeforeAdvisor" ... />
    <bean id="createAfterAdvisor" ... />
```

```
        <!-- 配置 DefaultAdvisorAutoProxyCreator -->
        <bean id="defaultAutoProxy" class="org.springframework.aop.framework.
autoproxy.DefaultAdvisorAutoProxyCreator" />
    </beans>
```

运行 main()方法，很不幸，我们得到了一个 AopConfigException，提示 CGLIB 没有找到。我们暂时不去管为何需要 CGLIB 库，添加一个 CGLIB，再次运行，得到如下结果。

```
[CreateAfterReturningAdvice] New user created: newuser
[CreateAfterReturningAdvice] New user created: newuser
[LoginMethodBeforeAdvice] User admin try to login...
[LoginMethodBeforeAdvice] User admin try to login...
[CreateAfterReturningAdvice] New user created: new_user
[CreateAfterReturningAdvice] New user created: new_user
```

分析结果，我们发现似乎每个 Advisor 都被应用了两次，这显然不符合我们的期望。实际上，并非 UserService 对象被应用了两次 Advisor，而是 UserDao 对象也被应用了 Advisor，因此，每个 Advice 都被调用了两次。UserDao 对象是一个类而非接口，这也是为什么 Spring 会提示需要 CGLIB 的原因。

我们观察 loginBeforeAdvisor 和 createAfterAdvisor 的定义就会发现，实际上，由于我们仅匹配了调用方法，因此，对于 UserService 和 UserDao，这两个 Advisor 均符合切入要求，所以，DefaultAdvisorAutoProxyCreator 不管三七二十一，将两个 Advisor 同时应用到了 UserService 和 UserDao，即使 UserDao 是一个 prototype 类型的 Bean。

```
    <bean id="loginBeforeAdvisor" class="org.springframework.aop.support.
NameMatchMethodPointcutAdvisor">
        <!-- login 同时也匹配 UserDao 的 login()方法 -->
        <property name="mappedName" value="login" />
        <property name="advice">
            <bean class="example.chapter4.LoginMethodBeforeAdvice" />
        </property>
    </bean>

    <bean id="createAfterAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <!-- .*create.*同时也匹配 UserDao 的 create()方法 -->
        <property name="pattern" value=".*create.*" />
        <property name="advice">
            <bean class="example.chapter4.CreateAfterReturningAdvice" />
        </property>
    </bean>
```

要避免 DefaultAdvisorAutoProxyCreator 将这两个 Advisor 也应用到 UserDao 对象上，我们需要定义更严格的切入点，因此，修改这两个 Advisor 如下。

```
        <bean id="loginBeforeAdvisor" class="org.springframework.aop.support.
RegexpMethodPointcutAdvisor">
            <!-- 限定切入点仅作用于 UserService 的 login()方法 -->
            <property name="pattern" value=".*UserService\.login" />
            <property name="advice">
                <bean class="example.chapter4.LoginMethodBeforeAdvice" />
            </property>
        </bean>

        <bean id="createAfterAdvisor" class="org.springframework.aop.support.
RegexpMethodPointcutAdvisor">
            <!-- 限定切入点仅作用于 UserService 的 create()方法 -->
            <property name="pattern" value=".*UserService\.create" />
            <property name="advice">
                <bean class="example.chapter4.CreateAfterReturningAdvice" />
            </property>
        </bean>
```

现在，由于我们使用了更严格的正则表达式来定义切入点，因此，UserDao 对象不再符合切入点要求。再次运行 main()方法，UserDao 不再被 DefaultAdvisorAutoProxyCreator 自动创建 AOP 代理，因此我们终于得到了正确的结果。

```
[CreateAfterReturningAdvice] New user created: newuser
[LoginMethodBeforeAdvice] User admin try to login...
[CreateAfterReturningAdvice] New user created: new_user
```

使用自动代理除了简化 XML 配置文件外，最重要的好处就是向客户端隐藏了原始的目标 Bean，客户端获取的永远只能是 AOP 代理对象，这样就避免了客户端直接通过获取原始 Bean 绕过 AOP 代理而破坏了应用程序逻辑的完整性。

4.4 使用引介

引介（Introduction）是一种特殊类型的拦截器，和普通的 Interceptor 不同，引介不能作用于任何切入点。引介只能作用于类，而非方法级别。换句话说，引介的作用就是要给一个已有的类动态增加接口。这听起来有点奇怪，为了让读者容易理解引介是如何实现动态地为一个类增加新的接口，我们仍以一个具体的例子来一步一步演示引介的原理和实现方式。

我们在 Eclipse 中建立如下的 Introduction 工程，其结构如图 4-10 所示。

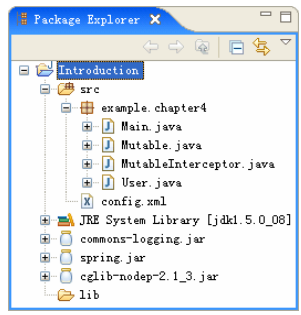


图 4-10

请注意我们定义的几个类和接口，User 类是一个普通的 JavaBean，它简单得不能再简单了。

```
public class User {
    private String username;
    private String password;
    private String email;

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

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

现在，我们的任务是为 User 类增加一个 Mutable 接口，以便能锁定 User 对象的内容。当调用了锁定方法后，外部客户端就不能修改 User 对象的状态。Mutable 接口如下。

```
public interface Mutable {
    boolean getReadonly();
    void setReadonly(boolean readonly);
}
```

但是，我们并不打算让 User 对象去实现 Mutable 接口，这样做虽然可行，却变成了在编译期就让 User 静态实现了 Mutable 接口，并且需要修改 User 对象所有的 setXxx() 方法，增加判断 readonly==true 的语句。如果要动态地为 User 对象增加 Mutable 接口，我们就需要使用 Spring 的引介。首先编写一个实现了 Mutable 接口的 MutableInterceptor，

稍后我们再详细讨论其内部实现机制。

```
public class MutableInterceptor extends DelegatingIntroductionInterceptor
        implements Mutable {
    private boolean readonly = false;

    public boolean getReadonly() {
        return readonly;
    }
    public void setReadonly(boolean readonly) {
        this.readonly = readonly;
    }
}
```

为了组装出一个具有 Mutable 接口的 User 对象，我们在 Spring 的 XML 配置文件中使用 ProxyFactoryBean 装配出这个特殊的 User 对象。config.xml 配置文件的内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd"
>
    <!-- 具有 Mutable 接口的代理 Bean -->
    <bean id="user" scope="prototype"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="target" />
        <!-- 对类代理,因此设置为 true,同时需要 CGLIB 运行库 -->
        <property name="proxyTargetClass" value="true" />
        <property name="proxyInterfaces">
            <list>
                <value>example.chapter4.Mutable</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>introductionAdvisor</value>
            </list>
        </property>
    </bean>

    <bean id="introductionAdvisor" scope="prototype"
    class="org.springframework.aop.support.DefaultIntroductionAdvisor">
        <constructor-arg>
            <bean class="example.chapter4.MutableInterceptor" />
```

```
        </constructor-arg>
    </bean>

    <!-- 原始 Bean -->
    <bean id="target" class="example.chapter4.User" scope="prototype">
        <property name="username" value="default-name" />
        <property name="password" value="PaSsWoRd" />
        <property name="email" value="default-email@abc.xyz" />
    </bean>
</beans>
```

最后，我们编写一个 main()方法测试一下，看看 User 对象能否转化为 Mutable 接口。

```
public static void main(String[] args) {
  ApplicationContext context = new ClassPathXmlApplicationContext ("config.xml");
   User user = (User)context.getBean("user");
   System.out.println(user.getEmail());
   ((Mutable)user).setReadonly(true);
   System.out.println(((Mutable)user).getReadonly());
   user.setEmail("new-email@abc.xyz");
   System.out.println(user.getEmail());
}
```

运行此 main()方法，控制台打印出了如下信息。

```
default-email@abc.xyz
true
new-email@abc.xyz
```

我们没有看到 ClassCastException 异常，说明我们获得的 User 对象确实具有 Mutable 接口，因此可以将其强制转型为 Mutable 类型，并调用 Mutable 接口的方法。但是 Mutable 接口虽然实现了，不过似乎并没有起作用，因为设置了 setReadonly(true)后，我们仍可以成功地调用 setEmail()为 User 对象设置新的 E-mail。这一功能将在稍后实现。

现在我们回过头来讨论这种“动态添加接口”是如何实现的。由于 ProxyFactoryBean 返回的是一个 AOP 代理对象，因此，它可以截获所有的方法调用，并委托给 MutableInterceptor 处理，MutableIntercepter 是从 Spring 的 DelegatingIntroduction Interceptor 派生而来，而 DelegatingIntroductionInterceptor 具有 MethodInterceptor 接口，因此可以实现方法拦截。

让我们看看 DelegatingIntroductionInterceptor 的 invoke(MethodInvocation)方法的实现。

```
public Object invoke(MethodInvocation mi) throws Throwable {
    // 调用的是否是新的接口的方法?
    if (isMethodOnIntroducedInterface(mi)) {
```

```
            return ...(调用 delegate 对象的方法);
        }
        return doProceed(mi);
    }
```

原来，当调用 User 代理对象的方法时，方法调用被 DelegatingIntroductionInterceptor 拦截后，它首先判断该方法是新接口定义的方法还是原始对象的方法。若是新接口定义的方法，则通过反射调用 MutableInterceptor 这个被称为 delegate 的对象的相应的方法；否则，直接通过 doProceed()调用原始对象的方法。从客户端的角度，整个调用过程如图 4-11 所示。

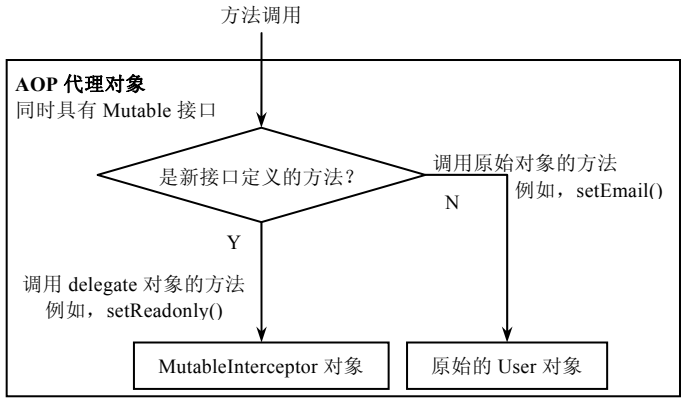


图 4-11

现在我们来实现 Mutable 接口的逻辑功能，即当用户调用了 setReadonly(true)后，User 对象应该被锁定，对其调用 setEmail()等方法应当抛出异常。既然 DelegatingIntroduction Interceptor 的 invoke()方法可以拦截整个方法的调用，那我们就在 MutableInterceptor 中覆写 invoke()方法。

```
public Object invoke(MethodInvocation invocation) throws Throwable {
    if(readonly && invocation.getMethod().getName().startsWith("set"))
        throw new UnsupportedOperationException("Object is set to readonly!");
    return super.invoke(invocation);
}
```

再次运行 main()方法，控制台果然得到了 UnsupportedOperationException 异常。

```
default-email@abc.xyz
true
Exception in thread "main" java.lang.UnsupportedOperationException: Object is
set to readonly!
```

不过，细心的用户也许已经发现了，一旦设置了 setReadonly(true)，就再也无法调用

setReadonly(false)来解除 User 对象的锁定了，因为 invoke()直接判断所有的 setXxx()方法，调用 setReadonly(false)也将得到一个 UnsupportedOperationException 异常，这是不正确的。

仔细观察 DelegatingIntroductionInterceptor 的 invoke()方法，其实我们希望拦截的是 User 对象的 set 方法，而不是 Mutable 接口定义的任何方法，因此，需要覆写的应该是 DelegatingIntroductionInterceptor 的 doProceed()方法，doProceed()方法处理的永远是原始对象的方法调用，而绝不会是 delegate 对象的方法调用。

```
protected Object doProceed(MethodInvocation mi) throws Throwable {
    if(readonly && mi.getMethod().getName().startsWith("set"))
        throw new UnsupportedOperationException("Object is set to readonly!");
    return super.doProceed(mi);
}
```

可见，如果要拦截所有的方法调用，就应当覆写 invoke()方法；如果仅仅希望拦截原始对象的方法，就应当覆写 doProceed()方法。到底覆写哪个方法应当根据应用程序的逻辑决定。

此外，我们看到每一个 AOP 代理对象都会包含一个原始的 User 对象和 MutableIntercepter 对象的引用，因此，将 MutableInterceptor 的 scope 设置为 prototype 是必要的，因为每个 AOP 代理都需要一个单独的 MutableInterceptor 实例来维护自身的状态。

必须注意的是，只有通过 ProxyFactoryBean 获得的 User 对象才被动态地添加了 Mutable 接口，如果应用程序自己通过 new User()生成了一个 User 对象，是绝不可能具有 Mutable 接口的。因此，引介虽然可以动态地为某个类添加新的接口，但是不要滥用，除非确有必要，否则很容易造成应用程序的逻辑混乱而不易理解。

4.4.1 在运行期改变AOP代理

在创建了 AOP 代理后，还可以将 AOP 代理转型为 org.springframework.aop.framework.Advised 接口，因为任何 AOP 代理都一定实现了这个接口。然后，可以调用 Advised 接口的方法在运行期动态改变 AOP 代理，例如，添加或删除 Advice 或 Advisor。

绝大多数情况下，在运行期修改一个 AOP 代理的业务对象是不合适的，如果将一个安全检查的 Advice 从 AOP 代理对象上移除了，就破坏了应用程序逻辑的完整性。有些时候，尤其是在开发阶段，可能需要在运行期修改 AOP 代理，不过这种情况也是很少见的。

为了防止在运行期修改 AOP 代理对象，可以设置 frozen 属性为 true，例如，设置 ProxyFactoryBean 的 frozen 属性，在这种情况下，Advised 接口的 isFrozen()方法将返回 true，任何增加或者移除通知的修改都会导致一个 AopConfigException 异常，这样就能确保 AOP 代理对象一旦被创建后就无法更改。

4.5 使用@AspectJ实现AOP

虽然BeanNameAutoProxyCreator和DefaultAdvisorAutoProxyCreator已经可以部分简化AOP的配置，不过在XML配置文件中要装配出AOP对象仍是比较麻烦的。了解AspectJ 5的读者一定知道，AspectJ 5引入了对Java 5注解的支持，这样使得对AOP的配置变得非常容易。

幸运的是，Spring 2.0也引入了AspectJ 5的这种注解，并使用AspectJ提供的一个库来解析并匹配Pointcut，这样，我们就可以通过Java 5注解以AspectJ的语法在Spring 2.0中配置AOP了。不过，Spring AOP在运行时仍是纯Java的实现，并不依赖于AspectJ的编译器和织入器。

使用AspectJ注解实现AOP最大的优点是将多个Advisor集中到一个类中，并且配置极为简单。我们仍以具体的例子来演示，在Eclipse中新建一个Spring_AspectJ工程，用AspectJ 5的注解来实现AOP。Spring_AspectJ工程的结构如图4-12所示。

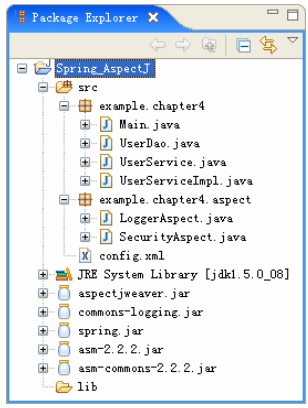


图4-12

我们的目标是将本章前文介绍的BasicAop工程和BasicAop_Advisor工程改造一下，用AspectJ注解来实现同样的功能，却可以大大简化代码的编写和XML配置文件。

4.5.1 声明Aspect

UserDao、UserService、UserServiceImpl和BasicAop_Advisor工程一致，这里不再多述。现在，我们编写一个LoggerAspect类，该类被标记为@Aspect，表示这是一个切面类。

```
@Aspect
public class LoggerAspect {
    ...
}
```

和直接继承 Advice 只能实现一个类对应一个 Advice 不同，一个切面类可以包含多个 Advice 和 Pointcut，每个 Advice 和 Pointcut 都是由一个独立的方法加上特定的注解来表示。下面，我们将为这个 Aspect 添加多个 Advice 和 Pointcut。

4.5.2 声明Advice

Advice 可以通过定义一个方法并加上特定的注释来指定 Pointcut。例如，我们需要添加一个 BeforeAdvice，回顾一下本章前面介绍的 MethodBeforeAdvice 的实现类。

```
public class LoginMethodBeforeAdvice implements MethodBeforeAdvice {
  public void before(Method m, Object[] args, Object target) throws Throwable {
      ...;
    }
}
```

使用 AspectJ 的注解后，以前需要编写的这个类就被简化为一个方法，加上特定的注解就可以同时实现 Advice 和 Pointcut 的声明。

```
@Aspect
public class LoggerAspect {
    @Before("execution(* example.chapter4.UserService.login(..))")
    public void logBefore() {
        System.out.println("[logger] User try to login...");
    }
}
```

现在，我们已经可以直接将这个 Aspect 在 Spring 的 XML 配置文件中装配起来了。首先，必须在 Spring 的 XML 配置文件中引入<aop>名字空间。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-2.0.xsd"
>
```

然后，声明这个 Aspect 和目标 Bean UserService。

```
<!-- 定义 Aspect -->
<bean id="loggerAspect" class="example.chapter4.aspect.LoggerAspect" />

<!-- 定义 Service -->
<bean id="userService" class="example.chapter4.UserServiceImpl">
    <property name="userDao">
        <bean class="example.chapter4.UserDao" />
    </property>
</bean>
```

最后，添加一行<aop:aspectj-autoproxy />，即开启了 AspectJ 的支持。

```
<!-- 启动 AspectJ 支持 -->
<aop:aspectj-autoproxy />
```

就这么简单！<aop:aspectj-autoproxy />实际上会引入一个类型为 AnnotationAwareAspectJAutoProxyCreator 的自动代理 Bean，这是一个 BeanPostProcessor，它会扫描 Spring 的 XML 配置文件中定义的所有具有@AspectJ 注解的 Bean，然后，利用 AspectJ 的库文件解析注解，并自动找出所有符合条件的 Pointcut，将它们装配为 AOP 代理对象。这一切都是自动完成的，配置非常简单！

现在我们继续为 LoggerAspect 添加更多的 Advice，对于记录用户登录成功的 AfterReturningAdvice，需要通过@AfterReturning 定义如下。

```
    @AfterReturning("execution(* example.chapter4.UserService.login(..)) &&
args(username,..)")
    public void logSuccess(String username) {
        System.out.println("[logger] User " + username + " login successfully!");
    }
```

请注意方法参数，如果要获得 login()方法的参数，就需要在指定 Pointcut 时加上“args(username,..)”，这样该 Advice 就可以获得 username 参数。

如果要记录用户登录失败，就需要定义一个 AfterThrowing 的 Advice。

```
    @AfterThrowing(
        pointcut="execution(* example.chapter4.UserService.login(..))",
        throwing="e"
    )
    public void logFailure(RuntimeException e) {
        System.out.println("[logger] Exception: " + e.getMessage());
    }
```

注意，pointcut 与 logBefore()方法的@Before 注解的表达式是一致的。此外，为了获

得异常的实例，还需要指定 throwing 的参数名称。

我们还差一个 Around 类型的 Advice。同样，通过@Around 注解声明这个 Advice 非常简单，我们将其定义在另一个单独的 SecurityAspect 中。

```
@Aspect
public class SecurityAspect {
    @Around("execution(* example.chapter4.UserService.login(..))")
    public Object securityCheck(ProceedingJoinPoint pjp) throws Throwable {
        String username = (String)pjp.getArgs()[0];
        System.out.println("[security check] username = " + username);
        if(!"admin".equals(username))
            return pjp.proceed();
        System.out.println("[security check] admin is forbidden!");
        throw new RuntimeException("Forbidden");
    }
}
```

Around 类型具有固定的方法签名，即：

```
public Object methodName(ProceedingJoinPoint pjp) throws Throwable { ... }
```

通过 ProceedingJoinPoint 参数就可以获得方法调用的所有信息，包括方法参数、目标对象等，然后，仍必须通过手动调用 proceed()方法来完成对目标对象的方法调用。

将 SecurityAspect 在 Spring 的 XML 配置文件中声明为一个 Bean，我们运行 main()测试方法就可以看到，使用 AspectJ 注解配置的各个 Aspect 和 Advice 都正确执行了。这里仅给出 main()方法的测试代码，读者可以自己运行看看结果是否符合期望的逻辑。

```
public static void main(String[] args) {
  ApplicationContext context = new ClassPathXmlApplicationContext ("config.xml");
    UserService userService = (UserService) context.getBean("userService");
    try {
        userService.create("new_user", "new_password");
        userService.login("new_user", "new_password");
        userService.login("bad_user", "bad_password");
    }
    catch(Exception e) {}
    try {
        userService.login("admin", "admin_password");
    }
    catch(Exception e) {}
}
```

现在，我们再回过头研究一下如何编写 Pointcut 的表达式。Spring 2.0 框架并不自己解析这些表达式，而是通过 AspectJ 的库来解析它们，因此，有必要熟悉一下 AspectJ

的语法。

Spring 支持以下几种主要的 AspectJ 风格的 Pointcut。

（1）execution：匹配方法执行的切入点，这也是在 Spring 中最常用的切入点定义方式。

（2）within：匹配特定类型的切入点；

（3）this：匹配特定实例的切入点。

此外，AspectJ 还支持诸如 call、initialization、if 等切入点类型。不过，在 Spring 中不能使用这些不被支持的定义，否则会得到一个 IllegalArgumentException 异常。

execution 是最常用的类型，通常能满足我们绝大多数的需求，因此，这里仅详细讨论一下 execution 表达式的编写方法。

execution 的完整表达式如下。

```
execution(修饰符? 返回类型 声明类型? 方法名称(参数类型) 异常类型?)
```

除了返回类型、方法名称和参数类型是必须的之外，其他都是可选的。使用最频繁的匹配模式是“*”，例如，用*表示任意的返回类型或者匹配部分命名模式（如“set*”表示所有以 set 开头的方法）。对于方法参数，“()”匹配一个没有任何参数的方法，而“(..)”匹配一个任意参数的方法（包括没有参数），“(*,String)”匹配一个有两个参数的方法，其中，第 2 个参数必须是 String 类型。

我们列举一些 execution 示例如下。

执行任意的 public 方法。

```
execution(public * * (..))
```

执行任何以 set 开头的方法。

```
execution(* set*(..))
```

执行任何至少有一个参数，且第 1 个参数为 String 的方法。

```
execution(* *(java.lang.String,..))
```

执行 Service 接口中任意方法。

```
execution(* example.Service.*(..))
```

执行 example 包或者子包的任意接口或类的任意方法。

```
execution(* example.*.*(..))
```

执行可能会抛出 IOException 的方法。

```
execution(* *(..) throws java.io.IOException)
```

4.5.3 声明Pointcut

在上面编写的几个 Advice 中，我们直接把 Pointcut 的表达式写在了相应的注解中。我们注意到 LoggerAspect 的几个 Advice 使用的 Pointcut 表达式其实是大同小异的，有没有办法将它们提取出来，放到一个单独的定义中呢？答案是使用@Pointcut 注解，将一个方法声明为 Pointcut。

```
@Pointcut("execution(* example.chapter4.UserService.login(..))")
public void loginMethod() {};
```

现在，我们就可以将@Before、@AfterReturning 和@AfterThrowing 的 Pointcut 表达式用统一的 loginMethod()替代。

```
@Aspect
public class LoggerAspect {
    @Pointcut("execution(* example.chapter4.UserService.login(..))")
    public void loginMethod() {};

    @Before(value="loginMethod()")
    public void logBefore() { ... }

    @AfterReturning("loginMethod() && args(username,..)")
    public void logSuccess(String username) { ... }

    @AfterThrowing(pointcut="loginMethod()", throwing="e")
    public void logFailure(RuntimeException e) { ... }
}
```

我们在 3 个作为 Advice 的方法中都复用了一个 Pointcut，这样使得这个 Aspect 类更容易维护，因为只需要修改@Pointcut 注解，就可以将更改同时应用到 3 个 Advice 中。

对比使用传统的 Spring 1.x 版本实现 AOP 的方式，我们可以总结出使用 AspectJ 注解来实现 AOP 的几大优势。首先，可以将几个相关的 Advice 统一放入一个 Aspect 类中，每个 Advice 即对应一个单独的方法，避免了实现多个只有一个方法的 Advice 类；其次，Pointcut 可以直接通过注解在 Advice 的方法层次上配置，不必在 XML 配置文件中定义额外的 Advisor 来组合 Advice 和 Pointcut，Pointcut 表达式的定义还可以复用，这样更增强了代码的可维护性；最后，在 XML 配置文件中装配 AOP 变得前所未有的简单，一行“<aop:aspectj-autoproxy />”配置就完成了所有的 AOP 装配！

如果要在应用程序中使用 AOP 功能，我们强烈建议采用以 AspectJ 5 注解来配置 AOP，当然，前提是必须使用 Java 5.0 或更高版本。

4.6 小结

本章我们详细介绍了 AOP 的基本概念，以及如何在 Spring 框架中应用 AOP 技术。我们首先通过 JDK 的动态代理机制给出了一个最简单的 AOP 实现，然后详细介绍了在 Spring 中使用 AOP 的方法。

通过编写 Advice 并在 XML 配置文件中装配出 Advisor 及最终的 AOP 代理对象是 Spring 1.x 和 Spring 2.0 中通用的方式。

通过使用自动代理可以简化部分 XML 配置文件的编写。

通过使用 AspectJ 5 的注解并使用<aop>名字空间实现 AOP 配置，这是最简便、最容易维护的方式。

在实际的项目中，只要采用 Java 5 或更高平台，我们强烈建议使用 AspectJ 5 的注解来配置 AOP，不仅代码更简洁，更重要的是，几乎无需在 XML 中进行烦琐的配置就可以非常简单地实现 AOP，大大提高了项目的可维护性。

要特别注意，AOP 是一种新的编程范式，它是为了解决 OOP 的某些不足而提出的解决方案，并非要取代 OOP 设计。在实际项目中仍必须坚持 OOP 设计为主，AOP 设计为辅，在合适的地方使用 AOP，而不要为了使用 AOP 而使用 AOP。

第 5 章

Spring数据访问策略

现在，几乎所有的应用程序都离不开数据库的支持。绝大多数的应用程序都将最终的业务数据存储在关系数据库中。本章我们将要介绍的就是如何在 Spring 中访问关系数据库，Spring 提供了许多访问关系数据库的方式，除了可以直接使用标准的 JDBC 接口外，Spring 还集成了众多的 O/R Mapping（对象/关系映射）框架。最为重要的是，无论采用何种方式访问数据库，Spring 均为我们提供了一致的编程模型，即统一的 DAO 模式和一致的异常体系，使得应用程序的结构层次更加清晰。

本章我们先介绍如何在 Spring 中使用 JDBC 接口访问数据库，然后分别介绍使用各种 O/R Mapping 框架，包括主要的开源 O/R Mapping 框架（如 Hibernate 和 iBatis），以及已经成为 JSR 标准的 JDO 和 JPA。

5.1 使用JDBC

JDBC（Java Database Connectivity）是 Java 访问 SQL 数据库的工业标准。JDBC 提供了一个统一的 API 接口，可以把 Java 应用程序从一个数据库移植到另一个数据库。和大多数 Java 规范一样，Sun 仅仅定义了 JDBC 接口，具体的实现则交由各数据库厂商。由于 Java 应用程序本身仅使用 JDBC API，从不使用针对特定数据库的 API，因此，在不同数据库之间移植对开发人员而言影响最小。

5.1.1 JDBC数据访问接口

JDBC 接口提供了以下数据库访问能力。

（1）创建数据库连接。

（2）向数据库提交 SQL 请求。

（3）处理数据库返回的结果集。

JDBC 驱动可以分为Ⅰ、Ⅱ、Ⅲ、Ⅳ共 4 种，不过，大多数 JavaEE 应用程序都使用第Ⅳ类 JDBC 驱动。

第一类（Type Ⅰ）JDBC 驱动通常被称为 JDBC-ODBC 桥，这是 Sun 公司提供的最原始的 JDBC 接口的实现，它使得开发人员可以利用现有的 ODBC 驱动程序，经过 JDBC-ODBC 的转化，封装为 JDBC 驱动。JDBC-ODBC 驱动已经内置在 JDK 内部，不过，这种实现的效率很低，除非无法找到数据库特有的 JDBC 驱动（例如，微软的 Access 数据库），一般不使用这类驱动。

第二类（Type Ⅱ）JDBC 驱动将 JDBC API 直接映射为数据库厂商提供的专用客户端 API，它需要一个本机代码库。例如，使用 Oracle Type Ⅱ驱动程序时，就需要 Oracle

客户端的接口库。

第三类（Type Ⅲ）JDBC 驱动用于充当 Applet 应用程序访问数据库的代理，因为在 Java 发展的早期，Applet 的安全模型禁止访问多台 Web 服务器的数据库资源。现在，一般不使用这类驱动。

第四类（Type Ⅳ）JDBC 驱动完全使用 Java 编写，它直接与数据库服务器通信，并使用数据库的低层协议，没有数据类型转化的开销，因此具有最高的效率，也是目前使用最广泛的 JDBC 驱动。

早期的 JDBC 应用程序通过 DriverManager 来创建 Connection（数据库连接），这种方式由于效率较低，已被 DataSource（数据源）取代。DataSource 可以看作获取 Connection 的工厂，它管理并维护一个 Connection 对象池，能支持大量并发的数据库连接请求。

使用 DataSource 时，一个典型的 JDBC 操作包括以下步骤。

① 从 DataSource 中获取 Connection。

② 通过 Connection 创建 Statememt。

③ 执行 Statement，处理返回的 ResultSet（结果集）。

④ 关闭 Connection 以释放资源。

由于 DataSource 可以管理并缓存 Connection，因此，关闭 Connection 并不意味着实际的数据库连接被关闭了，很可能仅仅是将 Connection 标记为“空闲”状态并返回给 DataSource，以便给下一个连接请求使用，这样就避免了反复创建和关闭数据库连接所带来的开销，大大提高了运行效率。而这些对客户端而言是透明的，客户端也不需要了解 DataSource 的实现内幕。

除了能缓存数据库连接以提供较高的效率外，使用 DataSource 还可以得到更多的好处。

（1）在 DriverManager 中必须将数据库连接的配置信息（如用户名和口令）硬编码到代码中，这种方式不够安全，而且缺乏灵活性，而 DataSource 的配置可以通过 JavaEE 服务器的配置完成，应用程序无需管理这些配置信息。

（2）应用程序也无需知道底层数据库，这样可以修改甚至替换低层的数据库系统，而不会对应用程序造成很大的影响。

（3）DataSource 可以限制应用程序的连接数目，这样可以有效地避免数据库服务器由于连接数目过多导致负载过重甚至崩溃。

通常，JavaEE 服务器会提供高效的 DataSource 的实现，应用程序通过 JNDI 查找获取 DataSource 的引用，然后使用它即可。还有许多开源的 DataSource 实现，例如，Apache DBCP。一些简单的 DataSource 更有助于应用程序的测试。

5.1.2 Spring封装的数据访问异常

与 SQLException 是一个 Checked Exception 不同，Spring 定义的基本的数据访问异常 DataAccessException 是一个 RuntimeException，这意味着在应用程序中不必强行捕获该异常，可以交由上层处理。如果直接处理 SQLException，我们需要获得特定数据库的错误代码，然后判断该 SQLException 的原因，包装成自定义异常交由上层处理。现在，由于 Spring 为我们提供了一个与底层数据库无关的异常体系，因此可以非常方便地处理各种数据访问异常。Spring 甚至试图翻译各个数据库厂商的错误代码，然后转换为 DataAccessException 异常体系中的某一个合适的子类。

表 5-1 显示了 Spring 框架常用的数据访问异常。

表 5-1

异　常	意　义
DataAccessException	Spring DAO 框架的异常体系根类，如果不在意数据访问的异常原因，则处理该异常就足够了
ConcurrencyFailureException	在多个并发操作时，无法乐观锁定或者获得数据库锁等
DataAccessResourceFailureException	访问数据彻底失败，例如，无法连接数据库
DataRetrievalFailureException	无法获取指定的数据，例如，根据某个主键无法查找到相应的记录
InvalidDataAccessResourceUsageException	无效的数据访问方法，最常见的情况是使用了语法错误的 SQL 语句
PermissionDeniedDataAccessException	没有数据访问权限，例如，当前数据库登录用户无权访问特定的表或执行特定的 SQL 语句
UncategorizedDataAccessException	无法归类的异常，通常是特定数据库的错误代码无法被 Spring 识别，于是将此 SQLException 归入此类

通常，应用程序只需捕获 DataAccessException 即可。由于 DataAccessException 是从 NestedRuntimeException 继承而来，因此，它完整地保存了原始异常信息（例如，SQLException）。

5.2 应用DAO模式

在多层应用程序设计中，将数据访问层和上层清晰地划分开是非常必要的。由于数据访问层包含了所有的数据库操作，因此，对上层而言，完全隔离了数据库访问。不但可以比较容易地使用另一种实现替换数据访问层，对上层而言，可以实现数据缓存，还可以使用模拟对象进行测试，有利于提高应用程序的可靠性和扩展性。

DAO 是 Data Access Object 的缩写，现在，DAO 已成为 JavaEE 系统的标准的数据访问模式。通过 DAO 接口，应用程序将底层的数据访问代码和上层的业务逻辑代码完全分离，使得各层保持相对独立，便于扩展和移植，如图 5-1 所示。

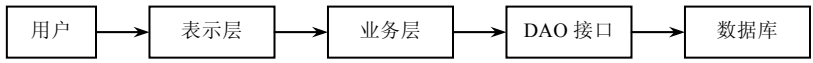


图 5-1

Spring 提供了标准的 DAO 模版，使用 Spring DAO 模版的一个最大的好处是 Spring 框架已经在 DAO 模版中封装了相当多的功能，例如，对于 JdbcDaoSupport 类，只需要注入 DataSource，JdbcDaoSupport 就自动为我们提供了 JdbcTemplate 的实例，并且无需关心 Connection 的获取和释放。如果完全从头编写自己的 DAO 实现类，则需要构造 JdbcTemplate 实例；如果不打算使用 JdbcTemplate，则必须手动管理 Connection 的获取和释放。

Spring DAO 模版的另一个好处是提供了一致的数据访问模式，无论是使用 Spring 提供的 JdbcTemplate，还是使用 Hibernate、Toplink 之类的 ORM 框架，都可以获得一致的编程模型。例如，对于一个操作用户实体 Account 的 DAO 对象，其定义可能如下。

```
public interface AccountDao {
    Account query(String username);
    void create(Account account);
    void update(Account account);
    void delete(Account account);
}
```

实现类可以根据具体的数据访问策略从 Spring 提供的 DaoSupport 派生。例如，若采用 Spring 提供的 JdbcTemplate，则实现类 AccountDaoImpl 可能如下。

```
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {
    ...
}
```

而使用 Hibernate 来访问数据库的另一种实现类 AccountDaoImpl 可能如下。

```
public class AccountDaoImpl extends HibernateDaoSupport implements AccountDao {
    ...
}
```

采用 Spring 提供的一致的 DAO 模式，则向上层屏蔽了底层数据访问的细节，上层持有的通常是 DAO 接口，因此，无需关心 DAO 的具体实现究竟采用何种数据访问策略。

5.2.1 准备数据库环境

为了创建基于 Spring 的数据库应用程序，需要首先准备数据库环境。有许多免费的数据库（如 MySQL、PostgreSQL、Oracle XE 等）可供选择，不过，这些数据库都需要复杂的安装和配置。为了把精力集中在学习如何使用 Spring 提供的 DAO 框架上，我们选择一款免费而小巧的 HSQLDB。

HSQLDB 是一个纯 Java 编写的数据库，其特点是体积小巧，600 多 KB 的 jar 包就同时包含了数据库和 JDBC 驱动程序，能无缝地嵌入到 Java 应用程序中，因此，特别适合数据库应用程序的测试和开发。

HSQLDB 还支持内存数据库模式。这意味着每次运行应用程序后，不必恢复数据库状态，因为应用程序一结束，内存数据库就自动消失了。在本章的示例中，应用程序启动时将启动 HSQLDB 的内存数据库，同时建立表并插入若干初始化表数据，这样可以反复运行应用程序，方便测试。

如果读者需要把数据以文件的形式保存下来，可以以文件模式启动 HSQLDB，具体方式请参考 HSQLDB 的文档，本书在此不做关于 HSQLDB 的更多介绍。

我们在 Eclipse 中建立如下的 DaoTemplate 工程，并将 hsqldb.jar 添加到 Java Build Path，如图 5-2 所示。

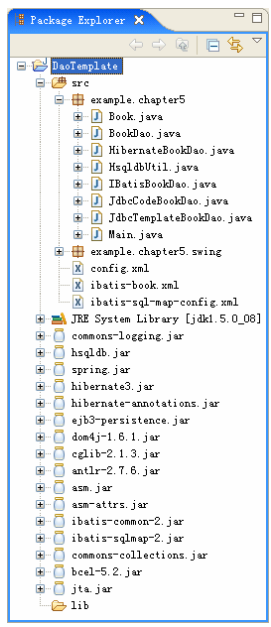


图 5-2

读者可以从本书的配套光盘中导入 DaoTemplate 工程。该工程有一个 HsqldbUtil 类负责启动 HSQLDB 数据库，并创建一个名为 bookstore 的测试数据库。

```
public static void startDatabase() {
    server = new Server();
    server.setDatabaseName(0, "test");
    server.setDatabasePath(0, "mem:bookstore");
    server.setLogWriter(null);
    server.setErrWriter(null);
    server.start();
}
```

HsqldbUtil 还负责创建一个 Book 表，然后插入两条初始记录。

现在，我们已经准备好数据库环境，还需要准备一个 DataSource 以供应用程序使用。Spring 提供了一个简单的 DataSource 实现，我们直接在 XML 配置文件中将 DataSource 定义为 Bean，以便能注入到其他需要 DataSource 的 Bean 中。

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:mem:bookstore" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

Spring 提供的这个 DriverManagerDataSource 是为了方便开发和调试。在实际的运行环境中，只需要在 JavaEE 服务器端配置好 DataSource 及其 JNDI 名称，应用第 3 章讲到的 JndiObjectFactoryBean，修改 XML 配置文件如下。

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/MyDataSource" />
</bean>
```

保证 Bean 的 id 不变，就完全不用修改应用程序代码，这正是 Spring 依赖注入的威力。

5.2.2 域对象模型

在多层应用程序中，由于数据库存储的是以记录为基本内容构成的表，而应用程序内部处理的数据都是类似 JavaBean 的实体对象，因此，在应用程序和数据库的存取过程中，就需要将这两者互相进行转化。通常，我们将这一类的实体对象称为域对象，在设计应用程序的结构时，应首先抽象出域对象模型，然后根据域对象创建数据库表的结构。

在 DaoTemplate 工程中，我们只有一个域对象，即 Book 对象，它代表一本书，包含 id、name 和 author 这 3 个属性，其中，id 是 Book 对象的唯一标识，对应的就是数据库表的主键。

Book 对象的定义就是一个普通的 JavaBean，包含一系列的 getter 和 setter 方法。

```
public class Book {
    private String id;
    private String name;
    private String author;

    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

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

    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
}
```

与数据库记录需要唯一由主键来标识一样，域对象也需要一个用于表示主键的属性。对于上面的 Book 对象来说，id 属性就是用于表示主键的标识属性。

5.2.3 主键生成策略

由于数据最终需要存储在关系数据库的表中，因此，需要确定记录的主键生成方式。常见的几种主键生成策略如下。

（1）使用数据库内置的自动增长类型的整型字段，一些常见的数据库（如 MS SQL Server、MySQL、Oracle 等）均支持这种自动增长的字段。

（2）使用整型字段，但自己管理主键的自动增长，在集群的环境下需要严格的同步，否则会造成重复主键导致数据插入失败。

（3）使用 UUID，UUID 是系统随机生成的 128 位整数，通常用十六进制的字符串表示。UUID 按照 IP 地址、网卡的 MAC 地址和随机数来保证生成的 128 位整数是全球唯一的。

许多数据库提供了自增主键的功能，例如，MS SQL Server，只要指定主键的列，每次插入新记录时，数据库系统自动为记录分配一个新的递增主键，以保证主键没有重复。这种方式的优点在于无需关心主键的生成算法，缺点是主键由数据库系统管理，在移植和数据的导入导出时可能会遇到很大的麻烦，并且主键只能为整型。使用第二种方式时，需要自行计算下一个主键，并且要保证没有重复，这需要编写更多的代码，并且在集群

环境下要进行严格的同步，否则极易出现重复主键导致插入失败。相比之下，使用 UUID 作为主键无需复杂的主键生成算法，并且由于 UUID 是根据计算机网卡的 MAC 地址、当前时间和随机数等计算出的，在任何时候都能保证是全局唯一的，因此，在集群环境下也能工作得非常好，很适合作为数据库记录的主键。从 Java 5 开始，JDK 提供了一个 UUID 类，调用其 randomUUID()方法就能非常方便地获得一个全局唯一的 UUID。UUID 的缺点是必须使用一个字符串来保存这个 128 位的整数，其占用的存储空间比整数类型的主键大。

综合比较，我们推荐使用 UUID 作为首选的主键生成策略，这样能以最小的代价获得最大的灵活性，并且当应用程序从单一服务器扩展到集群环境时，UUID 主键不受影响。

5.2.4 DAO接口

对于应用程序而言，对于每个域对象，我们都需要设计相应的 DAO 接口，以便能创建、查询、更新和删除域对象，这些操作在 DAO 接口的实现类中对应的便是实际的数据库操作，因此，通过 DAO 接口的这个抽象层，所有的数据库操作对于应用程序上层而言是完全不可见的，应用程序上层也不需要关心访问数据库的细节问题，如图 5-3 所示。

错误！

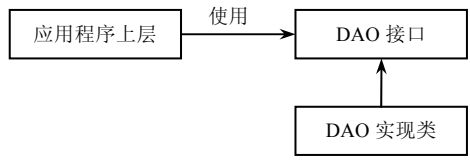


图 5-3

通常，需要对 DAO 接口定义 CRUD 操作。所谓 CRUD，就是 Create、Retrieve、Update 和 Delete 的缩写，代表了数据库操作中最常用的创建、查询、更新和删除操作。

例如，为了实现对 Book 对象的创建、查询、更新和删除操作，我们定义了 BookDao 接口。

```
public interface BookDao {
    List<Book> queryAll();
    List<Book> queryByAuthor(String author);
    void create(Book book);
    void update(Book book);
    void delete(String id);
}
```

一般只需要为创建、更新和删除操作各定义一个方法即可，但是，查询通常有多种

情况，可以根据需要定义多个查询方法。在上例中，我们定义了两个查询方法，分别是查询所有的 Book 和根据作者来查询 Book。

为了让读者看到对 BookDao 操作的实际效果，我们提供了一个 Swing 界面，可以非常方便地在窗口中进行书籍的查询、创建、更新和删除，如图 5-4 所示。

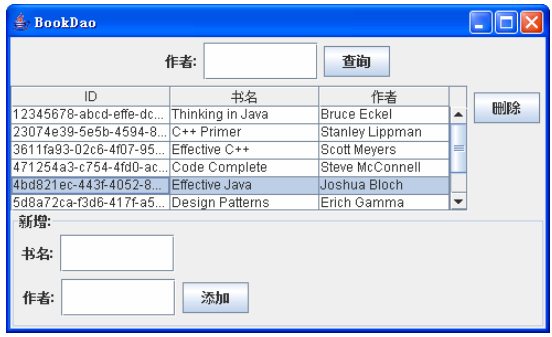


图 5-4

读者不需要了解 Swing 的相关知识，关于 Swing 的讨论也超出了本书的范围。所有的 Swing 相关代码都放在 example.chapter5.swing 中，我们需要关心的是 BookDao 接口和各种 BookDao 的实现类。BookDao 接口在 BookTableModel 类中被调用，并更新 Swing 窗口的显示。

```
public class BookTableModel extends AbstractTableModel {
    private List<Book> bookList;
    private BookDao bookDao;

    public BookTableModel(BookDao dao) {
        bookDao = dao;
        bookList = bookDao.queryAll();
    }

    public void refresh(String author) {
        if(author==null || author.trim().equals(""))
            bookList = bookDao.queryAll();
        else
            bookList = bookDao.queryByAuthor(author.trim());
        fireTableDataChanged();
    }

    public void removeBook(Book book) {
        bookDao.delete(book.getId());
        bookList.remove(book);
        fireTableDataChanged();
```

```
    }

    public void updateBook(Book book) {
        bookDao.update(book);
        fireTableDataChanged();
    }

    public void createBook(Book book) {
        bookDao.create(book);
        refresh(null);
    }
    ...
}
```

为了让读者对于 DAO 模式有更深入的理解，我们会在后面的讲述中将 Hibernate、iBatis 等各种 DAO 的实现都添加到 DaoTemplate 中，在应用程序启动时，可以手动选择一个 DAO 的实现，如图 5-5 所示。

图 5-5

这充分体现了 DAO 接口带来的好处。对于这个以 Swing 作为界面的应用程序，由于使用了 DAO 接口来分离用户界面和数据库访问逻辑，使得 DAO 层可以非常容易地实现可插拔式的替换，而上层的 GUI 仅调用 DAO 接口，不依赖于具体的数据库访问逻辑（无论是使用 JDBC，还是 Hibernate、iBatis、JDO、JPA 等）。对于多层结构的 Web 应用程序来说，道理是一样的，DAO 接口很好地隔离了中间的逻辑层和后端的数据访问代码，这不仅便于移植，更重要的是使每一层有明确的职责，便于测试和扩展，也有利于提高应用程序的整体性能，例如，可以在 DAO 接口之上加上一层缓存，以避免重复的数据库访问。

下面，我们分别介绍 Spring 框架中几种常见的数据库访问策略，包括使用 Spring 提供的 JdbcTemplate，以及常见的 ORM 框架，包括 Hibernate、iBatis、JDO 和 JPA。

5.3 使用JdbcTemplate

让我们先来看看直接使用 JDBC 接口来访问数据库需要编写的代码。对于一个普通

的 SELECT 查询操作，用 JDBC 访问数据库所需的代码如下。

```
Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
    conn = DriverManager.getConnection("url", "user", "password");
    ps = conn.prepareStatement("select * from Book where author = ?");
    ps.setString(1, "Erich Gamma");
    rs = ps.executeQuery();
    while(rs.next()) {
        rs.getString("column_name");
    }
}
catch(SQLException sqle) {
    // log...
}
finally {
    if(rs!=null) {
        try {
            rs.close();
        }
        catch(SQLException e) {}
    }
    if(ps!=null) {
        try {
            ps.close();
        }
        catch(SQLException e) {}
    }
    if(conn!=null) {
        try {
            conn.close();
        }
        catch(SQLException e) {}
    }
}
```

虽然真正有用的代码不过四五行，但是，为了保证数据库资源的正确释放，一个冗长的 try ... catch ... finally 是必不可少的。试想一下，假如应用程序中需要访问数据库的地方有上百处，如果每一次的数据库访问都需要编写上述重复而冗长的代码（声明引用，获得数据库资源，执行 SQL 语句，处理返回值，捕获 SQLException，然后在 finally 中依次释放数据库资源），将是多么复杂。如果在某一处的数据库访问中忘记了释放某个数

据库资源，则应用程序的健壮性将得不到保证，经过长时间运行很可能由于耗尽数据库资源而导致应用程序最终崩溃。

为了避免直接使用 JDBC 接口带来的复杂而冗长的代码，Spring 提供了一个强有力的模版类 JdbcTemplate 来简化 JDBC 操作，并且 DataSource 和 JdbcTemplate 全部都可以以 Bean 的形式定义在 XML 配置文件中，充分发挥了依赖注入的威力。

JdbcTemplate 的创建只需要一个 DataSource 接口。由于 DataSource 一般由管理员在服务器上部署，对于 Web 应用程序，只需要通过 JNDI 获得 DataSource 的引用即可。在开发阶段，我们要避免启动 JavaEE 服务器来使用 DataSource，因为这将大大增加开发和调试的成本。

Spring 提供了一个简单的 DriverManagerDataSource，可以直接以 Bean 的形式定义在 XML 配置文件中。

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:mem:bookstore" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

在产品部署阶段，只需用 JndiObjectFactoryBean 来替换这个 DriverManagerDataSource，应用程序的代码一行也不用更改。

下一步就是定义 JdbcTemplate。

```
<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>
```

然后将 JdbcTemplate 对象注入到自定义的 DAO 对象中，例如，MyBookDao。

```
public class MyBookDao implements BookDao {
    private JdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this. jdbcTemplate = jdbcTemplate;
    }
}
```

如果自定义的 DAO 是直接从 JdbcDaoSupport 派生的，就只需要注入一个 DataSource，JdbcTemplate 对象会自动生成，可以随时通过 getJdbcTemplate()方法获得 JdbcTemplate 对象的引用，例如，我们自定义的 JdbcTemplateBookDao。

```
<bean id="jdbcTemplateBookDao"
    class="example.chapter5.JdbcTemplateBookDao">
    <property name="dataSource" ref="dataSource" />
</bean>
```

如果存在多个 DAO 对象，则不必为每个 DAO 对象都创建一个 JdbcTemplate 对象，因为 JdbcTemplate 对象是线程安全的，可以被所有的 DAO 对象共享。JdbcTemplate 提供了以下主要方法来简化 JDBC 操作。

1．List query(String sql, Object[] args, RowMapper rowMapper)

执行一个 SQL 查询并返回一个 List，args 是传递给 SQL 语句的参数，而 rowMapper 负责将每一行记录转化为一个 Java 对象并放入 List 中。因此，最后获得的 List 就是包含了 Java 对象的结果集，而不是原始的 ResultSet 结果集。

例如，在 JdbcTemplateBookDao 中，为了将每一行数据库表记录映射为 Book 对象，就需要首先定义一个 RowMapper。

```
class BookRowMapper implements RowMapper {
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Book book = new Book();
        book.setId(rs.getString("id"));
        book.setName(rs.getString("name"));
        book.setAuthor(rs.getString("author"));
        return book;
    }
}
```

RowMapper 的编写非常简单。事实上，应当为每个实体对象编写相应的 RowMapper 实现，然后在所有的 DAO 实现类中复用它们。事实上，利用 RowMapper，我们已经自己手动实现了一个最简单的 O/R Mapping，尽管它远没有 Hibernate 之类的自动化 O/R Mapping 框架那么强大。然而，对于使用 JDBC 的应用程序而言，编写 RowMapper 也是一种良好的设计。

然后，我们实现 List<Book> queryByAuthor(String author)方法。

```
public List<Book> queryByAuthor(String author) {
    return getJdbcTemplate().query(
           "select * from Book where author=?",
           new Object[] { author },
           new BookRowMapper());
}
```

可以看到代码相当简洁。SQL 语句将被用于创建 PreparedStatement，而 Object[]数组的元素将自动被填充到 PreparedStatement 的参数中，因此，Object[]数组的元素个数必

须与 SQL 语句中的“?”个数相同。

有些时候，对于传入的参数，JdbcTemplate 无法自动转化为相应的数据库字段类型，例如，Date 对象可以映射为数据库字段的 DATE、TIME 和 TIMESTAMP 三种类型，此时，可以调用 query 的重载方法，指定每一个参数对应的数据库字段类型。数据库的字段类型被定义在 java.sql.Types 中。

```
List list = getJdbcTemplate().query(
    "select * from Book where pubDate<?",
    new Object[] { new Date() },
    new int[] { Types.DATE },
    new BookRowMapper());
}
```

如果确认查询的结果只有一行记录，例如，根据用户名和口令查找指定用户，则可以用 queryForObject()返回唯一的一个对象。如果返回值是通过 count()、sum()等 SQL 函数返回的整数类型，可以用 queryForLong()返回一个 long 型的整数，这两个方法都有多个重载形式，可以选择最合适的一个。

对于创建、更新和删除操作，通过 int update(String sql, Object[] args)方法来实现极其简单。例如，要实现 createBook(Book book)方法。

```
public void create(Book book) {
    getJdbcTemplate().update(
        "insert into Book (id, name, author) values (?, ?, ?)",
        new Object[] {book.getId(), book.getName(), book.getAuthor()});
}
```

update()方法同样有许多重载形式，可根据需要选择。

应用 query()和 update()方法，绝大多数的数据库操作都可以轻松完成。不过，有时候还需要更加复杂的操作，这时，可以考虑使用 execute 并传入一个 ConnectionCallback 对象。

```
getJdbcTemplate().execute(new ConnectionCallback() {
    public Object doInConnection(Connection connection)
        throws SQLException, DataAccessException
    {
        Statement stmt = connection.createStatement();
        ...
        return null;
    }
});
```

在回调方法 doInConnection()中，Connection 对象是以参数传入的，因此可以做任何

JDBC 相关的操作。由于 Connection 的获取与释放都是由 Spring 管理的，所以我们只需要使用 Connection，不要去关闭它。返回值也是任意的，只需要正确地做类型强制转化。

除了上面介绍的常用操作外，JdbcTemplate 还提供了一些有用的方法来简化 JDBC 调用。总之，JdbcTemplate 的设计目的就是要通过一定的封装来简化甚至自动创建 PreparedStatement，并通过一些回调对象避免手动管理 Connection 对象的获取和释放。

用 JdbcTemplate 编写一个 BookDao 完整实现的代码如下。

```
public class JdbcTemplateBookDao extends JdbcDaoSupport implements BookDao {
    class BookRowMapper implements RowMapper {
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Book book = new Book();
            book.setId(rs.getString("id"));
            book.setName(rs.getString("name"));
            book.setAuthor(rs.getString("author"));
            return book;
        }
    }

    public List<Book> queryAll() {
        return getJdbcTemplate().query(
            "select * from Book",
            new BookRowMapper());
    }

    public List<Book> queryByAuthor(String author) {
        return getJdbcTemplate().query(
            "select * from Book where author=?",
            new Object[] { author },
            new BookRowMapper());
    }

    public void create(Book book) {
        getJdbcTemplate().update(
            "insert into Book (id, name, author) values (?, ?, ?)",
            new Object[] {book.getId(), book.getName(), book.getAuthor()});
    }

    public void update(Book book) {
        getJdbcTemplate().update(
            "update Book set name=?, author=? where id=?",
            new Object[] {book.getName(), book.getAuthor(), book.getId()});
    }
```

```
    public void delete(String id) {
        getJdbcTemplate().update(
            "delete from Book where id=?",
            new Object[] {id});
    }
}
```

相比直接使用 JDBC 接口编写的 JdbcCodeBookDao，JdbcTemplateBookDao 的代码量从 193 行减少到了 56 行，不但大大提高了开发效率，还通过 RowMapper 的封装使得维护和扩展更加容易。

虽然可以直接使用 JDBC 操作数据库，并且通常能获得最高的效率，不过，其烦琐的代码不利于应用程序的快速开发和修改。在 JavaEE 领域，还有各种对象/关系映射框架（O/R Mapping）可供选用，在后面的几个小节中，我们还将介绍如何集成各种 O/R Mapping 框架，获得更高的开发效率。

5.4 集成Hibernate

在 B/S 应用中，面向对象是整个系统的设计基础，然而，系统后端的数据持久化仍然是由关系数据库实现的。把面向对象的软件和关系数据库一起使用是相当麻烦的，因为这是两种不同的设计和思维方式：一种是面向对象，一种是面向关系。在 JavaEE 领域，有许多连接 Java 环境和关系数据库表的映射工具，这些工具使用“对象/关系映射”（Object/Relational Mapping）的技术，实现 Java 对象模型和基于 SQL 语言的数据库关系模型的互相转化。

5.4.1 Hibernate简介

Hibernate 正是这些工具中非常出色的一个 ORM 框架，不仅因为其强大的功能和优秀的性能，还因为 Hibernate 是完全开源且免费的。Hibernate 不仅管理 Java 类到数据库表的映射，还提供了强大的基于对象的数据查询语言（HQL），可以大幅减少访问数据库需要编写的 JDBC 代码和烦琐的 SQL 语句。此外，Hibernate 通过“方言”（Dialect）最大限度地降低了对特定数据库厂商的依赖，同时又能利用其专有的 SQL 语法优化性能。

Hibernate 还提供了一系列复杂的功能来简化数据库操作，包括如下功能。

（1）延迟加载（Lazy Loading）：如果对象之间的关系极其复杂，我们很多时候不希望一下子取得所有相关联的对象，而是希望在访问到某个对象时再从数据库中读取，这样能在需要数据的时候才读取数据，不需要的数据将不会被读取。Hibernate 的这一特性

是依赖于 CGLIB 动态为实体对象生成子类并跟踪对象的属性读写实现的。

（2）主动抓取（Eager Fetching）：和延迟加载相反，主动抓取允许在一个查询操作中就获得所有关联的对象，这样可以避免多次的数据库连接造成的开销。主动抓取的性能依赖于底层数据库对连接操作（join）的支持好坏。

（3）缓存（Caching）：对于读数据远多于写数据的情况，如果能够将数据缓存在内存中，而不是每次读操作都请求数据库，将极大地提高整个系统的性能。Hibernate 提供了一级缓存和二级缓存来有效地实现数据缓存。

（4）级联操作（Cascading）：对于以外键关联的表，更新一个表时往往需要同时更新其关联的表，Hibernate 提供了级联更新和级联删除来自动完成相关表的更新，免去了手动操作的麻烦。

事实上，Hibernate 是对 JDBC 的一层较薄的封装。让我们看看 Hibernate 中主要的类与 JDBC 中主要的类的对应关系，如表 5-2 所示。

表 5-2

Hibernate	功　能	对应的 JDBC 类
SessionFactory	封装一个数据源，是线程安全的	DataSource
Session	封装一个较短的数据访问会话，非线程安全	Connection
Query	封装一个数据查询，非线程安全	PreparedStatement
Transaction	封装一个事务，非线程安全	JDBC Transaction 或 JTA

可以简单地从代码中看出 Hibernate 和 JDBC 的大致关系。

```
SessionFactory sf =
    config.buildSessionFactory();
Session session = sf.openSession();
Transaction tx = session.
beginTransaction();
Query query = session.createQuery(
    "select o from Book as o");
List<Book> books = query.list();

tx.commit();
session.close();
```

```
DataSource ds = jndiLookup("ds");

Connection conn = ds.getConnection();
conn.setAutoCommit(false);

PreparedStatement ps = conn. Prepare
Statement("select * from Book");
ResultSet rs = ps.executeQuery();
while(rs.next()) {...}
conn.commit();
conn.close();
```

当然，上述代码仅仅是一个非常简化的例子，Hibernate 自身还会做许多复杂的映射、缓存、优化等工作。

无论如何，Hibernate 最终都会将数据访问请求转化为实际的 SQL 语句。默认配置下，Hibernate 会打印出其发送的所有 SQL 语句，以方便调试。

关于 Hibernate 的详细用法，本节不会做太多的讨论。读者如果没有 Hibernate 基础，可以参考 Hibernate 的相关资料。本节将主要讨论如何在 Spring 框架中无缝地集成 Hibernate。读者可以看到，应用 Spring 还能进一步简化对 Hibernate 的操作。

5.4.2 配置Hibernate

应用 Spring 框架，可以非常方便地集成 Hibernate。Spring 同时支持 Hibernate 2.x 和 Hibernate 3.x 两个版本，在本书中，我们仅讨论如何使用 Hibernate 3.2 正式版，因为最新的 Hibernate 3.2 版本还支持 JPA 注解，配置起来更加方便。请读者注意，Spring 框架同时包含了对 Hibernate 2.x 和 Hibernate 3.x 的支持。许多模版类都有相同的名字，但是分别在不同的包中，因此要注意，导入的包应该为 org.springframework.orm.hibernate3.*。

在 Spring 中使用 Hibernate 有两种基本的方式，一种是完全不用 Spring 对 Hibernate 的封装，此时，需要开发者自己管理 Hibernate 中的资源，如 SessionFactory、Session、Query 等，Spring 不介入 Hibernate 代码。这种方式需要开发者对 Hibernate 非常熟悉，并且自行封装 DAO 接口。

典型的使用方式是编写一个 HibernateUtil。

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new AnnotationConfiguration()
                    .configure()
                    .buildSessionFactory();
        }
        catch(Exception e) {
            e.printStackTrace();
            throw new ExceptionInInitializerError(e);
        }
    }

    public static Session getCurrentSession() {
        return sessionFactory.getCurrentSession();
    }
}
```

为了使用 SessionFactory.currentSession() 方法，需要在配置文件中把事务（Transaction）绑定到 Thread 或 JTA 上。关于事务将在第 6 章中详细讨论，这里仅给出

一个典型的 Hibernate 的配置文件 hibernate.cfg.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="Bookstore">
      <property
name="connection.driver_class">org.hsqldb.jdbcDriver</ property>
        <property
name="connection.url">jdbc:hsqldb:mem:bookstore</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

      <property
name="dialect">org.hibernate.dialect.HSQLDialect</ property>
        <property
name="hibernate.cache.provider_class">org.hibernate.cache.
HashtableCacheProvider</property>
        <property
name="hibernate.transaction.factory_class">org.hibernate.
transaction.JDBCTransactionFactory</property>
        <property name="current_session_context_class">thread</property>
        <mapping package="example.chapter5"/>
        <mapping class="example.chapter5.Book"/>
    </session-factory>
</hibernate-configuration>
```

在这种方式下，对于每个数据库操作，典型的代码如下。

```
Session session = HibernateUtil.getSession();
Transaction tx = session.beginTransaction();
try {
    // TODO...
    tx.commit();
}
catch(Exception e) {
    tx.rollback();
}
// 不需要调用 session.close()，在事务提交或回滚时，Session 就自动关闭了
```

可以看到，对于每个数据库操作，都需要编写大量重复的 try ... catch 代码，因此，可以考虑使用 Spring 提供的 Hibernate 封装，把 Hibernate 的资源管理统统交给 Spring 去做。在这种方式下，就不要将 Hibernate 的事务绑定到 Thread 或 JPA 上，Spring 会自己

管理 Hibernate 事务。

Spring 提供了以下封装 Hibernate 的 Bean，可以非常方便地实现 Hibernate 操作，如表 5-3 所示。

表 5-3

LocalSessionFactoryBean	封装 Hibernate 的 SessionFactory
AnnotationSessionFactoryBean	支持 Annotaion 配置的 SessionFactory
HibernateTemplate	操作 Hibernate 的模版类，是操作 Hibernate 最重要的类
HibernateDaoSupport	实现 DAO 支持的支持类

在 Spring 中，使用 Hibernate 最简单的方法如下。

（1）定义 LocalSessionFactoryBean 或 AnnotationSessionFactoryBean，得到一个 SessionFactory 对象。

（2）从 HibernateDaoSupport 派生自定义的子类，如 HibernateBookDao。

```
public class HibernateBookDao extends HibernateDaoSupport
    implements BookDao {
}
```

然后，将 SessionFactory 注入到 HibernateBookDao 中。

```
<bean id="hibernateBookDao" class="example.chapter5.HibernateBookDao">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

在 HibernateBookDao 中，可以随时调用 getHibernateTemplate()方法获得一个 HibernateTemplate 对象，然后就可以非常方便地操作 Hibernate。

```
public class HibernateBookDao extends HibernateDaoSupport
    implements BookDao {

    public List<Book> queryAll() {
        return getHibernateTemplate().find("select b from Book as b");
    }

    public void create(Book book) {
        getHibernateTemplate().save(book);
    }

    ...
}
```

在 Spring 提供的 HibernateDaoSupport 超类中，如果注入 SessionFactory，HibernateDaoSupport 就会自动生成 HibernateTemplate 对象。由于 HibernateTemplate 是线

程安全的，如果有多个 DAO 对象，就没有必要在每个 DAO 对象中都生成一个 HibernateTemplate 对象，可以首先定义 HibernateTemplate 对象，然后分别注入到每个 DAO 对象中。

```
<bean id="hibernateTemplate" class="">
   <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="bookDao" class="...">
   <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>

<bean id="userDao" class="...">
   <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>

...
```

如果自定义的 DAO 没有从 HibernateDaoSupport 派生，也没有关系，只需要自己手动定义一个注入 HibernateTemplate 的方法。

```
public class MyDao {
   private HibernateTemplate hibernateTemplate;
   public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
      this.hibernateTemplate = hibernateTemplate;
   }
}
```

然后将 HibernateTemplate 注入。

```
<bean id="myDao" class="...">
   <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>
```

无论如何，我们最终的目的都是要获得一个 HibernateTemplate 的引用。由于 HibernateTemplate 本身包含了 SessionFactory 的引用，因此，只需要 HibernateTemplate 对象就可以完成所有的 Hibernate 操作。

HibernateTemplate 提供了一系列方法来实现数据库的 CRUD 操作。

（1）Serializable save(Object entity)

保存一个对象到数据库中，并返回其主键。

（2）Object get(Class entityClass, Serializable id)

通过主键获取一个指定类型的对象，若对应的数据库记录不存在，返回 null。

（3）Object load(String entityName, Serializable id)

和 get 类似，不同之处在于，如果对应的数据库记录不存在，将抛出 ObjectRetrieval FailureException 异常。

（4）void update(Object entity)

更新一个已存在的对象。

（5）void delete(Object entity)

删除一个已存在的对象。

如果要使用复杂的查询，可以使用 find()方法，这个方法有好几个重载形式，最常用的是 List find(String queryString, Object[] values)方法，非常类似 PreparedStatement，但是使用 Hibernate 特有的 HQL 查询，语法更加简单，例如：

```
List users = hibernateTemplate.find(
    "select u from User u where u.name=? and birth>?",
    new Object[] {"bill", new Date()} );
```

Object[]数组的个数必须和 HQL 语句中的“?”数量相同。

有些时候，HibernateTemplate 提供的上述方法仍不能满足更复杂的查询，例如，希望使用 Hibernate 的分页功能，或者一次执行多个批量操作，对此，可以使用 HibernateTemplate 提供的 Object execute(HibernateCallback action)方法，它允许执行任何 Hibernate 操作，并返回任意类型的值。

例如，为了实现一个分页操作，编写如下代码。

```
List users = (List) hibernateTemplate.execute(new HibernateCallback() {
    public Object doInHibernate(Session session)
            throws HibernateException, SQLException
    { // 取出第 100-120 条的记录：
        Query query = session.createQuery("select u from User u")
                    .setFirstResult(100).setMaxResults(120);
        return query.list();
    }
});
```

在 doInHibernate 中可以使用任意的 Hibernate 接口，如 Query、Criteria 等，返回值可以根据需要返回 List、Long 等对象，然后做类型强制转换。

由于 HibernateCallback 是一个回调接口，HibernateTemplate 在调用 HibernateCallback 前后会自动管理 Session 等相关资源，Session 的获取和释放由 Spring 负责，这正是为了让我们免去编写冗长的 try {...} catch {...} finally {...}代码。

读者从上面的讲述中可以看到，在 Spring 环境中使用 Hibernate 比直接使用 Hibernate 的 API 接口更加方便，尤其是编写 HQL 查询。甚至从来没有接触过 Hibernate 的开发人

员也能很快上手。当然，我们仍然强烈建议首先要具备扎实的 Hibernate 基础，只有彻底掌握了 Hibernate 的用法，在 Spring 中使用起来才会更加得心应手。

5.4.3 使用HibernateTemplate实现CRUD操作

在 5.3 节中，我们已经实现了两种 BookDao 的实现：一是直接使用 JDBC 接口，二是使用 Spring 提供的 JdbcTemplate。现在，我们为 DaoTemplate 增加一个新的 HibernateBookDao 的实现。

在编写 HibernateBookDao 之前，我们首先还需要定义如何将数据库表的字段映射到 Book 实体中。这里，我们抛弃了 XML 配置方式，直接以 JPA 注解来实现映射。打开 Book.java，添加如下必要的注解。

```
import javax.persistence.*;

@Entity(name="Book")
public class Book {
    private String id;
    private String name;
    private String author;

    @Id
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

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

    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }

}
```

@Entity 注解必须写在 class 定义处，表示这个 class 将作为数据库表的映射，name 属性指定了数据库的表名。

@Id 注解表示 Book 的 id 属性将被映射为数据库表的主键，没有任何标记的 getXxx 方法表示将数据库表中的对应字段映射到该属性上。这里，由于我们在创建数据库表的时候非常聪明地用 Book 的每个属性名称作为数据库表的字段名，因此，无需额外的注解就可以完成映射。如果数据库表的字段和对应的实体属性不一致，就必须采用 @Column 来定义将哪个字段映射到该属性上，例如：

```
@Column(name="BK_AUTHOR")
public String getAuthor() { return author; }
```

如果某些属性不是用于映射的，请务必使用@Transient 标识，Hibernate 才会忽略它。例如，age 属性并非从数据库映射而来（数据库表中根本没有 age 字段），而是根据 birth 计算出来的，此时，age 属性就应当被标识为@Transient，以便让 Hibernate 忽略它。

```
@Transient
public int getAge() { return getYear(new Date()) - getYear(birth); }
```

要注意的是，所有的注解（除了@Entity）都必须标记在 get 方法前。注解在 set 方法前是无效的。此外，注解被定义在 ejb3-persistence.jar 包中，请引入 javax.persistence.* 包。JPA 还将在 5.7 节中讨论，关于 JPA 注解更详细的用法请参考附录 B，这里仅使用了 JPA 注解来定义 Hibernate 的 O/R 映射规则。

为了使用 JPA 注解，还要引入 Hibernate Annotation 包，由于 Hibernate 依赖的包较多，请读者参考 DaoTemplate 工程的 ClassPath 设置。

为实体定义了注解后，通过 Spring 提供的 AnnotationSessionFactoryBean 就可以创建一个 SessionFactory 对象。

```
        <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
annotation. AnnotationSessionFactoryBean">
                <property name="dataSource" ref="dataSource" />
                <property name="annotatedClasses">
                    <list>
                        <value>example.chapter5.Book</value>
                    </list>
                </property>
                <property name="annotatedPackages">
                    <list>
                        <value>example.chapter5</value>
                    </list>
                </property>
                <property name="hibernateProperties">
                    <props>
                        <prop key="hibernate.dialect">org.hibernate.dialect.
HSQLDialect</prop>
                        <prop key="hibernate.show_sql">true</prop>
                        <prop key="hibernate.jdbc.fetch_size">10</prop>
                        <prop key="hibernate.cache.provider_class">org.hibernate.
cache.HashtableCacheProvider</prop>
                    </props>
                </property>
            </bean>
```

然后，从 HibernateDaoSupport 类派生 HibernateBookDao，使之能接受一个

SessionFactory 对象并自动创建一个 HibernateTemplate 对象。

```
<bean id="hibernateBookDao" class="example.chapter5.HibernateBookDao">
   <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

HibernateBookDao 的代码如下。

```
public class HibernateBookDao extends HibernateDaoSupport
   implements BookDao {

   public List<Book> queryAll() {
      return getHibernateTemplate().find("select b from Book as b");
   }

   public List<Book> queryByAuthor(String author) {
      return getHibernateTemplate().find("select b from Book as b where
b.author=?", author);
   }

   public void create(Book book) {
      getHibernateTemplate().save(book);
   }

   public void delete(String id) {
      Book book = (Book) getHibernateTemplate().load(Book.class, id);
      getHibernateTemplate().delete(book);
   }

   public void update(Book book) {
      getHibernateTemplate().update(book);
   }
}
```

和直接使用 JDBC（193 行代码）、使用 JdbcTemplate（56 行代码）方式比较，使用 HibernateTemplate 使 DAO 的实现更加简洁，代码行数仅 33 行。无论是开发效率，还是代码的可维护性都大大地提高了。

5.4.4 使用Hibernate注解验证数据

从 Hibernate 3.2 开始，除了支持 JPA 注解外，Hibernate 还提供了一套基于注解的验证机制，在插入和更新数据时，可以自动对实体对象进行验证。

例如，为了验证 Book 对象是否合法有效，我们可以分别对其 id、name 和 author 属性

编写 Hibernate 验证注解。注意：和 JPA 注解一样，只有针对 get 方法的注解才是有效的。

```
@Pattern(regex="[a-z0-9\\-]{36}", message="Invalid id!")
public String getId() { return id; }
```

上面的@Pattern 注解使用正则表达式限制了 id 属性的规则，只能使用小写字母、数字和“–”号，并且长度为 36 个字符。message 用于当验证失败时向用户返回的异常信息。

常见的 Hibernate 验证注解如表 5-5 所示。

表 5-5

注　解	适用类型	说　明	示　例
@Pattern	String	通过正则表达式来验证字符串	@Pattern(regex="[a-z]{8}")
@Length	String	验证字符串的长度	@Length(min=3, max=20)
@Email	String	验证一个 E-mail 地址是否有效	@Email
@Range	long	验证一个整型是否在指定范围内	@Range(min=0, max=100)
@Min	long	验证一个整型必须不小于指定值	@Min(value=10)
@Max	long	验证一个整型必须不大于指定值	@Max(value=20)
@Size	集合或数组	验证集合或数组的大小是否在指定范围内	@Size(min=1, max=255)

以上每个注解都可以有一个 message 属性，用于在验证失败后向用户返回的消息。还可以在一个属性上使用多个注解。

为了让验证生效，我们还必须向 Hibernate 的 SessionFactory 注册两个 Hibernate 内置的事件监听器，用于在插入和更新数据之前触发。

```
<property name="eventListeners">
    <map>
        <entry key="pre-update">
            <bean class="org.hibernate.validator.event.ValidatePreUpdate
EventListener" />
        </entry>
        <entry key="pre-insert">
            <bean class="org.hibernate.validator.event.ValidatePreInsert
EventListener" />
        </entry>
    </map>
</property>
```

现在，插入或更新实体时，若违反了我们定义的注解约束，Hibernate 将会抛出一个 InvalidStateException 异常，通过 getInvalidValues()方法就可以知道所有违反约束的属性。

5.5 集成iBatis

iBatis 也是一个开源且免费的 O/R Mapping 框架。和 Hibernate 这种全自动化的 O/R Mapping 框架相比，iBatis 只能算一个“半自动化”框架，因为所有的数据库操作都需要手动编写 SQL 语句，虽然增加了一定的开发难度，但是由于 SQL 语句完全掌握在开发人员自己的手中，因此 SQL 语句可以做最大的优化，其速度与直接使用 JDBC 相当。

iBatis 使用 XML 配置文件将 Java 实体映射到数据库表，并存储了所有的 SQL 语句。相对于 Hibernate，学习 iBatis 的成本要低很多，只要熟悉 SQL 语法，就可以执行任意的数据库操作。

iBatis 的配置也比较简单。首先，需要为每一个实体定义其各自的映射和所有用到的 SQL 语句，然后在一个根配置文件中包含它们。

还是以 DaoTemplate 工程为例，我们准备为 BookDao 接口再增加一个 IBatisBookDao 的实现。首先，将 ibatis-common-2.jar 和 ibatis-sqlmap-2.jar 放入 ClassPath 中，然后定义 Book 实体的映射文件 ibatis-book.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
    PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Book">
    <typeAlias alias="Book" type="example.chapter5.Book" />

    <select id="queryAll" resultClass="Book">
        select * from Book
    </select>

    <select id="queryByAuthor" parameterClass="java.lang.String" resultClass
="Book">
        select * from Book where author=#value#
    </select>

    <insert id="create" parameterClass="Book">
        insert into Book (id, name, author) values (#id#, #name#, #author#)
    </insert>

    <delete id="updateBook" parameterClass="Book">
        update Book set name=#name#, author=#author# where id=#id#
    </delete>
```

```
    <delete id="deleteBook" parameterClass="java.lang.String">
        delete from Book where id=#value#
    </delete>
</sqlMap>
```

为了简化 XML 配置文件的编写，我们首先通过 typeAlias 为 Book 实体起了一个简短的“别名”，以便在后面的配置中引用别名而非完整的类名。

```
<typeAlias alias="Book" type="example.chapter5.Book" />
```

查询操作通过<select>定义，其 id 属性将在程序中引用，表示执行相应的 SQL 语句。

```
<select id="queryByAuthor" parameterClass="java.lang.String"
    resultClass="Book">
        select * from Book where author=#value#
</select>
```

可选的属性有：parameterClass 表示传入的参数类型；resultClass 表示返回的结果类型。如果传入的参数为简单值类型（如 int、double、String 等），用“#value#”表示；如果传入的是实体对象（如 Book 对象），用“#属性名#”表示，iBatis 会自动将其替换为对应的属性值。

与插入、更新和删除对应的 SQL 分别以<insert>、<update>和<delete>定义，其形式与<select>类似，这里就不再多述。

由于我们创建的数据库表名称为 Book，各字段与 Book 实体的属性完全相同，因此，iBatis 默认就可以完成自动映射。如果表名或字段名与 Book 实体不完全匹配，就需要手动定义映射规则，通常紧接在<typeAlias>之后，在任何<select>、<update>等操作前。

```
<resultMap id="bookResultMap" class="Book">
    <result property="id" column="PK_ID"/>
    <result property="name" column="BK_NAME"/>
    <result property="author" column="BK_AUTHOR"/>
</resultMap>
```

我们建议尽量做到实体属性和数据库表的字段名完全一致，这样不仅看起来直观，而且免去了配置的麻烦。

最后，还需要一个根配置文件 ibatis-sql-map-config.xml，我们提供的是最简单的一个实现。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
```

```
    <!-- 引入对 Book 定义的映射 -->
    <sqlMap resource="ibatis-book.xml" />
</sqlMapConfig>
```

根配置文件还可以有更多的选项来控制缓存、批量执行的 BatchSize 等，可以在 iBatis 的文档中获得更详细的说明。所有的 XML 配置文件都必须放在 ClassPath 下，或者指定相对于 ClassPath 的相对路径。

完成了上述 XML 配置文件的编写，我们就可以开始编程使用 iBatis 了。iBatis 只提供了一个最核心的对象 SqlMapClient，它充当了 Hibernate 中 SessionFactory 和 Session 的功能，并且是线程安全的，因此，通过 SqlMapClient 就可以完成所有的 SQL 操作。例如，我们定义了 id 为“queryByAuthor”的查询为“select * from Book where author= #value#”，就可以通过 SqlMapClient 直接获得 SQL 语句的返回值。

```
try {
    List list = sqlMapClient.queryForList("queryByAuthor", "Erich Gamma");
}
catch(SQLException sqle) {
}
```

返回的 List 包含的是 iBatis 经过转换的 Book 对象。

比较麻烦的是构造 SqlMapClient 对象，此外，SqlMapClient 仍然需要处理 SQLException，因此，Spring 提供了一个 SqlMapClientTemplate 对象，它包装了一个 SqlMapClient，并且无需处理 SQLException。由于我们的 IBatisBookDao 派生自 IBatisDaoSupport，直接注入 SqlMapClient 就会自动生成一个 SqlMapClientTemplate，下面是 XML 配置片断，只需指定一个 DataSource 和 configLocation。

```
<bean id="iBatisBookDao" class="example.chapter5.IBatisBookDao">
    <property name="sqlMapClient">
      <bean class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
         <property name="configLocation" value="ibatis-sql-map-config.xml" />
          <property name="dataSource" ref="dataSource" />
       </bean>
    </property>
</bean>
```

在 IBatisBookDao 中，可以随时调用 getSqlMapClientTemplate()获得 SqlMapClient Template 对象，实现 CRUD 操作非常简单。

```
public class IBatisBookDao extends SqlMapClientDaoSupport implements BookDao {
    @SuppressWarnings("unchecked")
    public List<Book> queryAll() {
        return getSqlMapClientTemplate().queryForList("queryAll");
```

```
    }

    @SuppressWarnings("unchecked")
    public List<Book> queryByAuthor(String author) {
      return getSqlMapClientTemplate().queryForList("queryByAuthor", author);
    }

    public void create(Book book) {
        getSqlMapClientTemplate().insert("create", book);
    }

    public void delete(String id) {
        getSqlMapClientTemplate().delete("delete", id);
    }

    public void update(Book book) {
        getSqlMapClientTemplate().update("update", book);
    }
}
```

和 Hibernate 这种全自动化的 ORM 框架相比，使用 iBatis 需要自己手动编写 SQL 语句，这样带来了更多的复杂性，增加了调试 SQL 语句的开发成本，但是，好处是由于 SQL 语句完全由开发者自己控制，因此可以针对数据库做最大限度的优化，并可以使用底层数据库提供的所有功能，如果 SQL 优化得当，则可能获得很高的执行效率，因此，iBatis 很适合那些项目组中有 SQL 开发和调优经验成员的团队。

5.6 集成JDO

JDO 即 Java Data Object 的缩写，是 JavaEE 平台的另一个持久化规范。与作为 EJB 持久化机制的 Entity Bean 只能运行在 EJB 容器中相比，JDO 作为一种轻量级的持久化规范可用于任何 Java 环境，包括 J2SE 环境。目前，JDO 有两个版本：1.0 版和 2.0 版，2.0 版本对 1.0 版本又做了大量的修订和完善，这里我们只讨论如何在 Spring 中集成 JDO 2.0。

JDO 2.0 规范可以从 http://jcp.org/en/jsr/detail?id=243 下载，JDO 也定义了一种对象查询语言 JDOQL，与 Hibernate 的 HQL 大同小异。由于 JDO 只是一个持久化标准，各厂商都可以有各自的实现，为了把 DaoTemplate 工程的 BookDao 以 JDO 规范来实现，我们还需要一个实现了 JDO 标准的产品。JPOX 是 JDO 2.0 标准的参考实现，并且遵循 Apache 2.0 开源协议，因此可以免费用于开发和部署。

使用 JPOX 之前，需要首先下载以下 jar 文件。

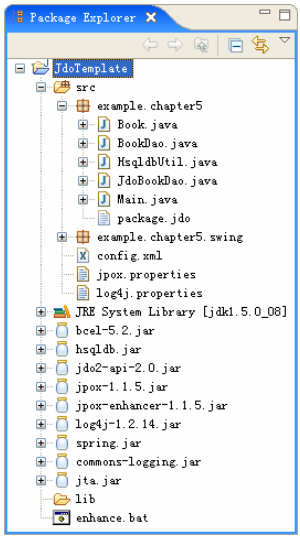


图 5-6

（1）jdo2-api-2.0.jar：从 http://db.apache.org/jdo/下载。

（2）jpox-1.1.5.jar 和 jpox-enhancer-1.1.5.jar：从 http://www.jpox.org/下载。

（3）log4j-1.2.x.jar：从 http://logging.apache.org/log4j/下载。

（4）bcel-5.2.jar：从 http://jakarta.apache.org/bcel/下载。

其中，jdo2-api-2.0.jar 是 JDO 2.0 标准的接口，jpox-1.1.5.jar 是 JPOX 对 JDO 2.0 的实现，jpox-enhancer- 1.1.5.jar 用于对持久化对象进行增强，需要用到 Apache 的 BCEL（Byte Code Engineering Library）库，log4j 是 JPOX 运行时必须的日志库。

由于 JDO 与其他 ORM 框架有所不同，它需要一个增强的步骤，并且没有默认的事务设置，因此我们不准备继续在 DaoTemplate 工程中添加 JDO，而是在 Eclipse 中建立一个单独的 JdoTemplate 工程，其结构如图 5-6 所示。

JdoTemplate 工程的结构和 DaoTemplate 工程一致，仍以 Swing 为界面，不过只提供了 JdoBookDao 的实现。

JDO 需要一个 XML 文件来定义需要持久化的对象应该如何映射到数据库表中，为了把 Book 类映射到数据库表，配置文件以 package.jdo 命名，并放到相应的包目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="example.chapter5" table="BOOK">
        <class name="Book">
            <field name="id" primary-key="true" column="ID" />
            <field name="name" column="NAME" />
            <field name="author" column="AUTHOR" />
        </class>
    </package>
</jdo>
```

现在，我们为 BookDao 新增一个采用 JDO 的 JdoBookDao 实现。由于 JdoBookDao 是从 JdoDaoSupport 派生的，所以可以随时调用 getJdoTemplate()获得 JdoTemplate 对象。

```
public class JdoBookDao extends JdoDaoSupport implements BookDao {
    public List<Book> queryAll() {
```

```
        Collection c = getJdoTemplate().find(Book.class);
        List<Book> books = new ArrayList<Book>();
        books.addAll(c);
        return books;
    }
    public List<Book> queryByAuthor(String author) {
        // query: class, filters, parameters, values
        Collection c = getJdoTemplate().find(Book.class,
               "author==a", "String a", new Object[] { author });
        List<Book> books = new ArrayList<Book>();
        books.addAll(c);
        return books;
    }
    public void create(Book book) {
        getJdoTemplate().makePersistent(book);
    }
    public void delete(String id) {
        Book book = (Book) getJdoTemplate().getObjectById(Book.class, id);
        getJdoTemplate().deletePersistent(book);
    }
    public void update(Book book) {
      Book b = (Book) getJdoTemplate().getObjectById(Book.class, book.getId());
        b.setName(book.getName());
        b.setAuthor(book.getAuthor());
        getJdoTemplate().makePersistent(b);
    }
}
```

要实现更复杂的操作，可以使用 JdoTemplate 提供的 execute(JdoCallback)方法，例如：

```
Object object = getJdoTemplate().execute(new JdoCallback() {
    public Object doInJdo(PersistenceManager pm) {
        // TODO:
        return null;
    }
});
```

由于获得了 PersistenceManager 对象，因此可以做任意的操作，但却无需关心如何获取 PersistenceManager、事务管理和处理各种异常。

下一步是在 Spring 的 XML 配置文件中配置 JDO，与 JDBC、Hibernate 及 iBatis 不同，由于 JDO 没有默认的事务设置，因此我们需要明确地定义事务模式，否则创建和更新操作将不会有任何效果。关于事务我们将在第 6 章详细讨论，这里只给出一个使用 Java 5 注解配置的事务。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-2.0.xsd"
>
    <bean id="pmFactory" class="javax.jdo.JDOHelper"
      factory-method="getPersistenceManagerFactory" destroy-method="close">
       <constructor-arg value="jpox.properties" />
    </bean>

  <bean id="jdoTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
       <property name="persistenceManagerFactory" ref="pmFactory" />
    </bean>

    <bean id="jdoBookDao" class="example.chapter5.JdoBookDao">
       <property name="persistenceManagerFactory" ref="pmFactory" />
    </bean>

    <tx:annotation-driven transaction-manager="jdoTxManager" />
</beans>
```

在接口 BookDao 的声明处添加@Transactional 注解，即可完成整个声明式事务的配置。

```
@Transactional
public interface BookDao {
    ...
}
```

注意到 JDO 的 PersistenceManagerFactory 可以由 JDOHelper 对象创建，它需要一个 JDO 配置文件，这里我们编写了一个 jpox.properties 来配置 JPOX，包括数据库连接信息等。

```
# jpox.properties
javax.jdo.PersistenceManagerFactoryClass=org.jpox.PersistenceManagerFactoryImpl
javax.jdo.option.ConnectionDriverName=org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionURL=jdbc:hsqldb:mem:bookstore
javax.jdo.option.ConnectionUserName=sa
javax.jdo.option.ConnectionPassword=

org.jpox.autoCreateSchema=true
```

```
org.jpox.validateTables=false
org.jpox.validateConstraints=false
```

在使用 JdoBookDao 之前，还有一件事情要做。为了能将 Book 对象存储到数据库表中，需要为 Book 对象增加 javax.jdo.spi.PersistenceCapable 接口，该接口有大量的方法需要实现，如果需要持久化的实体对象很多，为每一个对象手动增加该接口将非常麻烦，并且不利于源代码的维护，因为我们关心的是 Book 对象的基本属性，至于如何持久化到数据库的关系表中，应该由 JDO 的具体实现来完成。因此，JDO 标准还定义了一个字节码“增强”的步骤，通过字节码增强来自动完成这个步骤。

JPOX 提供了一个 jpox-enhancer-1.1.5.jar 来完成这一步骤。首先，在 Eclipse 中编译工程，然后打开命令提示符，切换到工程的根目录下，输入命令。

```
D:\project\JdoTemplate>java -cp bin;lib\jdo2-api-2.0.jar;lib\jpox-1.1.5.
jar;lib\jpox-enhancer-1.1.5.jar;lib\bcel-5.2.jar;lib\log4j-1.2.14.jar org.jpox.
enhancer.JPOXEnhancer bin\example\chapter5\package.jdo
```

JPOX 会将增强操作记录在 jpox.log 文件中，如果得到一个出错的消息，请务必查看 jpox.log 获得详细的错误信息。为了便于反复运行，我们将该命令以“enhance.bat”文件保存，存放在工程根目录下。

如果没有对实体对象增强，在运行期会得到一个 Spring 封装的 JdoUsageException 异常，这是由 ClassNotPersistenceCapableException 异常引起的，表示该实体没有实现 PersistenceCapable 接口。

可以配置工程以便让 Eclipse 自动完成增强这一任务。选择菜单“Project”→“Properties”，在弹出的“Properties for JdoTemplate”对话框中选择左侧的“Builders”，单击“New...”按钮，选择“Program”，新建一个 Builder，如图 5-7 所示。

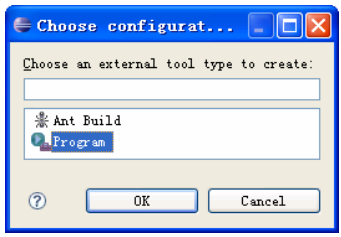


图 5-7

在弹出的对话框中配置这个 Builder，填入 Name：JDO Enhancer；Location：当前工程根目录下的 enhance.bat；Working Directory：当前工程根目录，如图 5-8 所示。

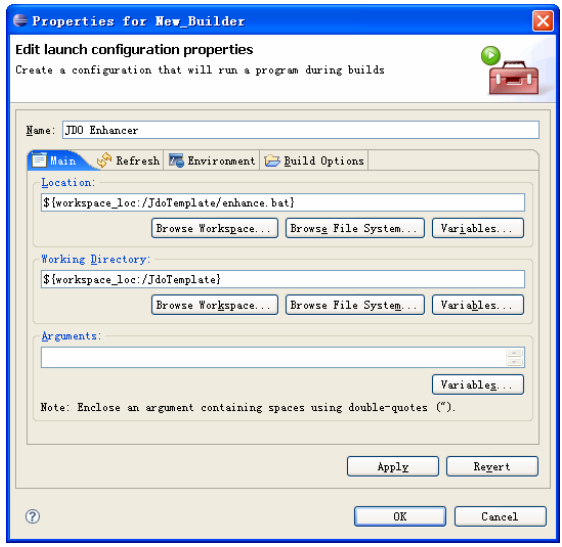


图 5-8

单击“OK”按钮后即完成了配置。确保 Java Builder 和 JDO Enhancer 都选中，如图 5-9 所示。

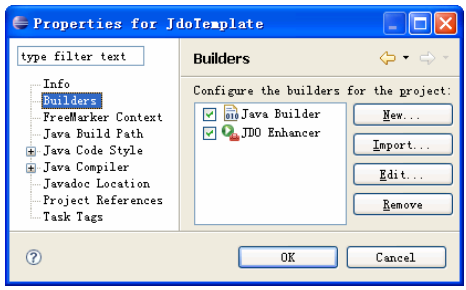


图 5-9

单击“OK”按钮，然后编译工程，Eclipse 就会按照 Builders 中的顺序首先用 Java Builder 编译工程源代码，然后调用我们自定义的 JDO Enhancer 对实体类进行增强，并在 Console 中输出结果，如图 5-10 所示。

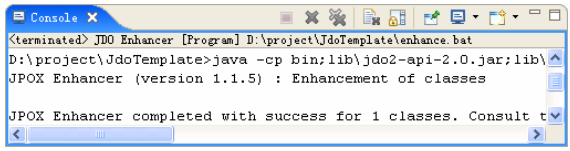


图 5-10

这样就可以不必手动完成增强这一过程，而直接在 Eclipse 中运行程序了。

事实上，几乎所有的 O/R Mapping 框架都需要对实体类进行增强，以便持久化框架能跟踪实体的变化，Hibernate 采用的是 CGLIB 在运行期动态增强实体的方式，因此无需手动增强这一步骤，使用起来更为方便。

使用 JDO 作为持久化机制的好处是它是一种标准 API，目前有许多可供选择的实现，如果想换用不同的 JDO 实现，一般无需更改代码，除非用到了特定 JDO 产品的扩展 API。麻烦之处在于需要增强的步骤。

其他开源和商业的 JDO 实现还有 Kodo JDO、intelliXO、JDOInstruments、Speedo、TriActive JDO 等。

5.7 集成JPA

JPA 即 Java Persistence API（Java 持久化接口），是 JavaEE 5.0 中引入的新的标准的持久化 API。JPA 是 EJB 3.0 的一部分，而 EJB 3.0 又是 JavaEE 5.0 规范的一部分。虽然 JPA 的设计目的主要是为 EJB 3.0 提供持久化支持，但是 JPA 并不依赖 EJB。从一开始，JPA 就被设计成可以在 EJB 容器之外独立使用。事实上，JPA 也可以在 J2SE 的环境中启动。

JPA 定义了两个最重要的规范：一是通过 JPA 注解，使得对实体的映射配置可以通过注解来完成，而不必编写复杂的 XML 配置文件。事实上，Hibernate 3.2 版本已经可以通过 Hibernate Annotation 库实现 JPA 注解配置映射。对于使用 JDK 1.4 或更低版本，JPA 也支持传统的 XML 配置文件。二是 JPA 定义了一个统一的数据访问接口，包括 EntityManagerFactory、EntityManager 等。熟悉 Hibernate 的读者很快可以发现，JPA 中的这些概念几乎与 Hibernate 完全相同，因为 JPA 规范本身就从 Hibernate、JDO 等技术中吸收了大量的成功之处。

Spring 从 2.0 版本开始对 JPA 提供支持。在 Spring 中，可以放心地使用 JPA 作为持久化方案，但却完全不使用 EJB 3.0 的其他规范。

已经有很多持久化框架完整地实现了 JPA 规范，如 Hibernate 3.2、TopLink 10g、BEA Kodo 4.1 等。在本节中，我们将以 Hibernate 3.2 为例，使用 JPA 为 BookDao 再增加一个 JpaBookDao 的实现。

Spring 对 JPA 的支持采用了 Adapter 模式，并且提供了 JapDaoSupport 和 JpaTemplate 支持类，使 JPA 和其他持久化机制（如 Hibernate、JDO）拥有一致的编程模型，如图 5-11 所示。

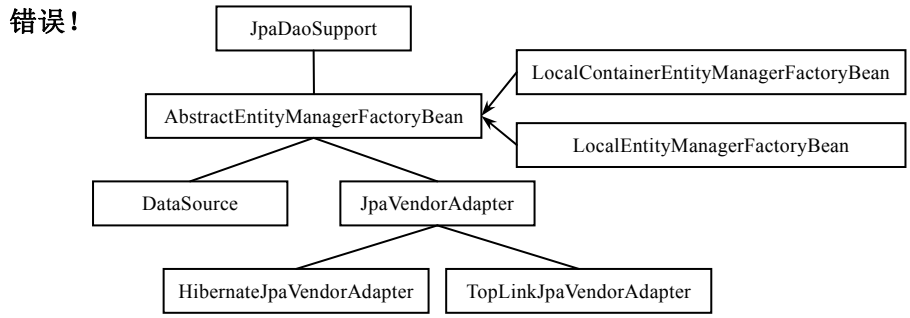


图 5-11

对于开发人员来说，在 Spring 中使用 JPA 非常容易，只需要从 JpaDaoSupport 派生出自定义的子类并实现自定义的接口，例如，派生出 JpaBookDao 并实现 BookDao 接口，其他所有组件均在 XML 配置文件中定义，并注入相应的依赖关系。

AbstractEntityManagerFactoryBean 负责创建一个 JPA 实现的 EntityManagerFactory，它根据 DataSource 和 JpaVendorAdapter 来自动查找并初始化 JPA 的 EntityManagerFactory。例如，若指定 JpaVendorAdapter 为 HibernateJpaVendorAdaptor，则使用 Hibernate EntityManager 作为 JPA 实现；若指定 TopLinkJpaVendorAdapter，则使用 TopLinkJpaVendorAdapter。

Spring 提供了两种 AbstractEntityManagerFactoryBean 的实现，LocalEntityManagerFactoryBean 提供了标准的 JPA 初始化配置，相比之下，LocalContainerEntityManagerFactoryBean 则更为灵活，可以注入 Spring 管理的 DataSource。我们以 LocalContainerEntityManagerFactoryBean 为例，详细介绍如何在 Spring 中实现 JPA。

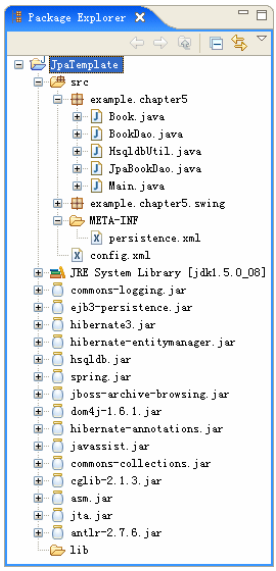


图 5-12

我们在 Eclipse 中新建一个 JpaTemplate 工程，结构和 DaoTemplate 工程类似，如图 5-12 所示。

首先，我们从 JpaDaoSupport 派生自定义的 JpaBookDao 并实现 BookDao 接口。

```
@Transactional
public class JpaBookDao extends JpaDaoSupport
implements BookDao {
    ...
}
```

然后，在 XML 配置文件中初始化所有依赖的组件。

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

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-2.0.xsd"
>

   <!-- 定义 DataSource -->
   <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
      <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
      <property name="url" value="jdbc:hsqldb:mem:bookstore" />
      <property name="username" value="sa" />
      <property name="password" value="" />
   </bean>

   <!-- 使用 JPA 实现的 DAO -->
   <bean id="jpaBookDao" class="example.chapter5.JpaBookDao">
      <property name="entityManagerFactory" ref="entityManagerFactory" />
   </bean>

   <bean id="entityManagerFactory" class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource"/>
      <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.HibernateJpa
VendorAdapter">
            <!-- 是否显示 SQL 语句：是 -->
            <property name="showSql" value="true" />
            <!-- 是否自动创建表：否 -->
            <property name="generateDdl" value="false" />
         </bean>
      </property>
   </bean>

<bean id="jpaTxManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManagerFactory" />
   </bean>

   <tx:annotation-driven transaction-manager="jpaTxManager" />
</beans>
```

和 JDO 类似，JPA 也需要正确配置事务，并将 BookDao 标注为@Transactional，这里仅给出和 JdoTemplate 类似的参考配置。

实际上，Hibernate 的 JPA 实现是由 Hibernate EntityManager 实现的，它通过 JPA 接口封装了 Hibernate 专有的 API，但是，我们不必直接与 Hibernate EntityManager 打交道，因为 Spring 已经为我们准备好了 JpaTemplate，可以随时在 JpaBookDao 中调用 getJpaTemplate()获得该对象，然后对其操作，就像使用 HibernateTemplate、SqlMapClient Template、JdoTemplate 一样。

JpaBookDao 的完整实现如下。

```
public class JpaBookDao extends JpaDaoSupport implements BookDao {
    public List<Book> queryAll() {
        return getJpaTemplate().find("select b from Book b");
    }
    public List<Book> queryByAuthor(String author) {
        return getJpaTemplate().find(
            "select b from Book b where b.author=?",
            new Object[] { author });
    }
    public void create(Book book) {
        getJpaTemplate().persist(book);
    }
    public void delete(String id) {
        Book book = getJpaTemplate().find(Book.class, id);
        getJpaTemplate().remove(book);
    }
    public void update(Book book) {
        getJpaTemplate().merge(book);
    }
}
```

可以看到，JpaTemplate 对象提供的 find()、persist()、remove()、merge()方法和 Spring 提供的其他 XxxTemplate 非常类似，用法几乎完全相同。如果需要更复杂的数据操作，同样有一个 JpaCallback 回调接口。

```
Object ret = getJpaTemplate().execute(
    new JpaCallback() {
        public Object doInJpa(EntityManager entityManager) throws Persistence
Exception {
            // TODO:任意 JPA 操作
            return null;
        }
    }
```

```
        );
```

运行 JPA 之前的最后一步是编写一个/META-INF/persistence.xml 配置文件，这是 JPA 规范要求所必须的，这里我们只配置需要映射的 Java 对象和 Hibernate 数据库方言。

```
        <?xml version="1.0" encoding="UTF-8"?>
        <persistence xmlns="http://java.sun.com/xml/ns/persistence"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.
sun.com/xml/ns/persistence/persistence_1_0.xsd"
            version="1.0"
        >
            <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
                <class>example.chapter5.Book</class>
                <properties>
                    <property name="hibernate.dialect" value="org.hibernate.dialect.
HSQLDialect"/>
                </properties>
            </persistence-unit>
        </persistence>
```

确保在 ClassPath 中能找到/META-INF/persistence.xml，然后把相关 jar 包添加进来，读者可以参考本书配套光盘的 DaoTemplate 工程源代码。运行 DaoTemplate 工程，选择 jpaBookDao，其实现功能和前面的各种 DAO 实现完全相同。

由于 Spring 对 JPA 的良好封装，我们仅针对 JPA 接口编程，完全没有涉及具体的 JPA 实现（如本例的 Hibernate 3.2 EntityManager）。如果要使用另一个 JPA 实现也是极为容易的。例如，如果需要将 Hibernate JPA 实现替换为 TopLink JPA 实现，只需要修改 XML 配置文件和/META-INF/persistence.xml 配置文件的相关配置即可，应用程序代码一行也不用改动。

5.8 小结

在本章中，我们学习到了如何在 Spring 环境下访问数据，并且应用 DAO 模式分离应用程序的底层数据访问逻辑和上层业务逻辑，使应用程序更具灵活性和可扩展性。

读者可能已经注意到了，虽然 JavaEE 平台定义了多种持久化规范，包括 EJB 的 Entity Bean、JDO 和最新的 JPA，但实际上，这些持久化解决方案都大同小异，底层 ORM 框架的实现原理是一致的。许多 ORM 框架经过一层适配便可实现标准的持久化规范，例如，Hibernate 除了自身的专有接口外，经过适配后就可以提供 Entity Bean 和 JPA 的实现，如图 5-13 所示。

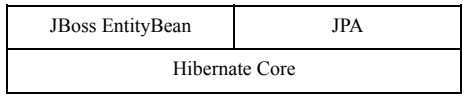


图 5-13

TopLink 除了自身的专有接口外，也可以适配为符合 JDO 和 JPA 规范的接口，如图 5-14 所示。

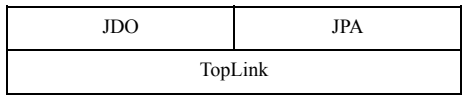


图 5-14

Kodo 最初是一个 JDO 的实现，不过 4.0 版本也可以支持符合 JPA 规范的接口，如图 5-15 所示。

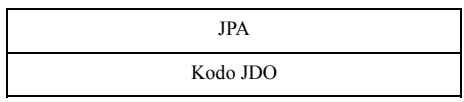


图 5-15

因此，我们需要关心的是 ORM 框架的原理和掌握一种 ORM 框架的使用方法。为了便于使用，Spring 框架提供了一种统一的编程模型和一致的 DAO 实现，大大方便了使用这些持久化框架。

我们还分别使用 JDBC、Spring 提供的 JdbcTemplate、HibernateTemplate、JdoTemplate 和 JpaTempate 实现了 BookDao 接口。读者可能已经注意到了 Spring 提供的各种 DAO Template 编程模式都非常相似，并且都以回调的方式避免直接处理资源的获取和释放，以及 SQLException 的问题。

我们看看这些不同的 DAO 实现的代码量，如表 5-6 所示。

表 5-6

实现方式	代 码 量
直接使用 JDBC	约 193 行
使用 JdbcTemplate	约 56 行
使用 HibernateTemplate	约 33 行
使用 IBatisTemplate	约 32 行
使用 JpaTemplate	约 33 行
使用 JdoTemplate	约 34 行

可见，直接使用 JDBC 不仅代码编写量很大，而且维护比较困难，而采用 Spring 提

供的 JdbcTemplate 就可以大大减少 JDBC 代码量。尽管如此，我们还是推荐采用一种 O/R Mapping 框架，无论是 Hibernate、IBatis 还是 JPA 实现，都可以极大地减少代码的编写，大大提高开发效率。由于 Hibernate 是一个功能强大且开源的免费 O/R Mapping 框架，我们没有理由拒绝使用它。在本书的示例应用程序 Live 在线书店中，就采用了 Hibernate 作为持久化框架。读者还将在第 11 章中看到更多的 Hibernate 应用。

在第 6 章中，我们将讨论另一个数据访问的关键问题——事务管理。通过事务，对数据访问的完整性和一致性就可以进一步得到保障。

第 6 章

Spring事务管理

数据库系统为了保证数据操作的完整性和一致性，引入了事务这个重要概念。所谓事务，就是将一系列的数据库操作作为一个整体来执行。本章将详细介绍数据库事务的概念，如何使用 Spring 来管理事务，以及 JavaEE 应用程序中常用的事务模式。

6.1 JavaEE事务概述

事务是数据库系统中的重要概念。其基本特点是要么全部执行，要么全部不执行。我们以现实生活中的银行转账操作为例子，可以更好地理解事务的概念。

例如，假定张三在银行办理转账业务，希望将 100 元人民币转移到李四的账户上，这个转账操作必须执行以下两步才能完成。

（1）从张三的账户上减去 100 元。

（2）给李四的账户上增加 100 元。

以上两步必须保证同时执行成功。假设第一步完成后，由于通信故障等原因，导致第二步没有执行，则张三的账户扣除了 100 元，而李四的账户却没有增加相应的 100 元，银行的客户显然不可以接受这种结果。

为了保证上述两步操作能够作为一个整体执行，就需要将这两步放入一个事务中执行，倘若第二步执行失败，则整个事务将全部撤销，张三的账户就不会有损失。

在计算机中，上述操作就是通过数据库系统的事务来保证正确执行的。在实际情况中，由于还涉及多用户并发操作数据库的情形，因此，对于数据库事务来说，必须具备 ACID 特性，ACID 是 Atomic（原子性）、Consistency（一致性）、Isolation（隔离性）和 Durability（持久性）的缩写，这几个特性的含义如下。

（1）原子性：原子性的含义是一个事务中的一系列操作都是不可分割的工作单元，只有所有的操作都成功执行完毕，整个事务才算成功；如果其中任何一步操作执行失败，则已经执行的操作也必须全部撤销，数据库的状态要恢复到没有执行事务前的状态。

（2）一致性：一致性是指事务不能破坏数据库中的数据完整性和逻辑上的完整性。例如，对于上述转账事务，无论事务是否成功，张三和李四的账户总额不应该变化。

（3）隔离性：由于实际的数据库系统可能有多个用户同时执行他们各自的事务，多个并发执行的事务操作同一数据出现冲突时，数据库必须协调并保证各事务的独立性。

（4）持久性：持久性是指事务成功完成后，对数据库状态的更改必须永久地保存下来。

在最常用的关系数据库中，数据库系统依赖日志和锁机制来保证事务具有 ACID 特性，其实现细节是非常复杂的。幸运的是，对于开发人员来说，并不需要了解数据库事务的底层细节，只需要通过数据库系统提供的接口，就可以按照业务需求来控制事务。

对于JavaEE应用程序来说，由于各数据库厂商都提供了符合JDBC接口标准的驱动程序，因此，在JavaEE中操作数据库事务就是对JDBC事务的操作。

6.1.1 事务的隔离级别

虽然事务可以保证ACID特性，但是，由于数据库系统总是对多个用户进行服务，即多个客户端可以同时操作数据库。如果并发执行的两个或多个事务操作的是同一数据，则仍可能出现逻辑错误。

为了保证多个并发事务的正确执行，我们还需要对每个数据库事务设置隔离级别，表6-1列出了数据库支持的常见的4种隔离级别。

表 6-1

隔离级别	是否有第一类丢失更新	是否出现脏读	是否出现虚读	是否出现不可重复读	是否有第二类丢失更新
Read Uncommitted	否	是	是	是	是
Read Commited	否	否	是	是	是
Repeatable Read	否	否	是	否	否
Serializable	否	否	否	否	否

（1）ISOLATION_READ_UNCOMMITTED：允许读到其他事务没有提交的数据，这可能会造成脏读。

（2）ISOLATION_READ_COMMITTED：只允许读已经提交的数据，避免了脏读。

（3）ISOLATION_REPEATABLE_READ：可以看到其他事务已经提交的新插入的记录，但不能看到对记录的更新；

（4）ISOLATION_SERIALIZABLE：最严格的隔离级别，操作同一数据的并发事务都只能以串行化方式执行，即第一个事务结束后才能开始下一个事务。

上述隔离级别逐个增高。隔离级别越高，就越能保证数据的完整性和一致性，但是事务的并发性就会降低，这将导致整个数据库系统的性能下降。所以，需要根据实际情况合理地选择事务的隔离级别。

如果没有指定任何隔离级别，即使用底层数据库系统的默认隔离级别ISOLATION_DEFAULT。不同数据库的默认的隔离级别通常有所不同。在关系数据库中，要实现事务的隔离级别，数据库系统就必须采用锁来保证并发事务的正确执行，当一个事务需要访问数据库的某些数据时，必须根据其隔离级别获得相应的共享锁或独占锁，数据库系统针对事务的隔离级别可以对整个表、某一行记录或索引列等实现加锁。

6.1.2 JDBC事务

在 JDBC 中，Connection 对象代表一个数据库连接，事务是与 Connection 对象绑定的。在获得一个 Connection 对象后，默认的事务模式为自动提交（Auto Commit），即对于每一个 SELECT、INSERT、UPDATE 和 DELETE 操作，都会自动启动事务，执行该 SQL 语句，然后提交事务。例如，对于一个 INSERT 操作。

```
Connection conn = dataSource.getConnection();
Statement stmt = conn.createStatement();
// 隐含地开始事务...
stmt.executeUpdate("insert into ...");
// 隐含地提交事务...
// 清理资源
stmt.close();
conn.close();
```

在执行 INSERT 语句前后，JDBC 事务就自动地开始和提交，如果该语句在执行过程中抛出了 SQLException，事务就自动回滚。

可以关闭默认的自动事务提交模式，此时，执行一系列 SQL 操作后，必须手动提交事务，或者在发生异常的情况下回滚事务，典型的手动提交 JDBC 事务的代码如下。

```
Connection conn = dataSource.getConnection();
// 显式地开始事务:
conn.setAutoCommit(false);
// 第一个操作:
Statement stmt1 = conn.createStatement();
stmt1.executeUpdate("insert ...");
stmt1.close();
// 第二个操作:
Statement stmt2 = conn.createStatement();
stmt2.executeUpdate("update ...");
stmt2.close();
// 显式地提交事务:
conn.commit();
// 或者通过 conn.rollback()回滚事务...
// 清理资源
conn.close();
```

在手动提交模式下，如果忘记调用 Connection 对象的 commit()方法，底层数据库根本就不会有任何更改。读者还需要注意，为了简化，上面两段代码都省略了异常处理。在实际的应用程序中，事务控制必须正确地在 try { ... } catch() { ... } finally { ... }中实现，以避免任何资源泄漏。

6.1.3 JTA事务

JTA 是 Java Transaction API 的缩写，即 Java 事务 API，也是 JavaEE 标准的一部分。事实上，JTA 标准是建立在 JTS（Java Transaction Service，Java 事务服务）标准之上的。JTS 是一个底层 API，供应用服务器厂商使用，而 JTA 是一个高层接口，供开发者使用。

既然 JDBC 标准已经完整地支持了数据库事务，为什么还需要另外一个 JTA 标准？因为 JDBC 事务是一个局部事务，只能应用于当前数据库，在分布式环境下，如果事务需要跨多个数据库，则只能使用支持分布式应用的 JTA 标准。

在本章开头的例子中，我们假定张三和李四的账户都存在同一家银行的同一个数据库中。在实际情况中，张三的账户在 A 银行而李四的账户在 B 银行是非常普遍的，同样必须保证转账操作作为一个事务在两个银行的数据库中都成功完成，因此，需要一种分布式的事务管理，而 JTA 则是 JavaEE 应用程序的分布式事务标准。

JTA 使用事务管理器（Transaction Manager）来管理分布式事务。一个 JTA 事务涉及一个事务管理器和多个资源管理器。一个资源管理器可以是任何类型的持久性数据存储系统，包括数据库、MIS 系统、JMS 等。事务管理器负责协调所有事务参与者之间的通信。

使用 JTA 控制事务和 JDBC 事务非常类似，通常，UserTransaction 对象是从 JNDI 中查找获得的，大多数 JavaEE 应用服务器将 UserTransaction 对象绑定在标准位置“java:comp/UserTransaction”。

```
UserTransaction tx = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
try {
    tx.begin();
    ...
    tx.commit();
}
catch(Exception e) {
    try {
        tx.rollback();
    }
    catch(Exception ex) {
        throw new RuntimeException(ex);
    }
}
```

如果要使用 JTA 事务，则一般有多个数据库参与 JTA 事务，这要求数据库的 JDBC 驱动程序实现 XAConnection 和 XAResource 接口，只有实现了这些接口的 JDBC 驱动才

能正确地参与 JTA 事务。由于并非所有的数据库驱动都支持 JTA 事务，一些高端 JavaEE 服务器支持将普通的 JDBC 驱动模拟为支持 XA 的 JDBC 驱动。

一旦使用了 JTA 事务，则 XAConnection 就会自动参与 JTA 事务。这意味着 XAConnection 和普通的 JDBC Connection 的事务操作是不同的，它们决不能自动提交，也绝不能调用 XAConnection 的 commit()和 rollback()方法。相反，只能对 UserTransaction 对象调用 begin()、commit()和 rollback()方法。

如果应用程序只使用一个数据库，则仅使用 JDBC 事务就足够了，尽管仍然可以使用 JTA 事务，但这会带来不必要的复杂性。许多应用程序考虑到将来可能会扩展为分布式应用，因此从一开始就使用 JTA 事务。幸运的是，对 Spring 应用程序来说，究竟选择 JDBC 事务还是 JTA 事务已经不重要了，因为 Spring 框架使用同一种抽象的事务编程模型，配合声明式事务管理，则选择 JDBC 事务还是 JTA 事务仅仅需要修改配置文件，一般不会涉及代码的改动。这种简洁而灵活的事务管理功能也是 Spring 框架最具特色的优秀功能之一。

6.1.4 Spring的事务模型

Spring 框架提供了极其强大而简便的事务处理功能，其核心便是 PlatformTransactionManager 抽象接口。Spring 将所有的事务管理都抽象为 PlatformTransactionManager、TransactionStatus 和 TransactionDefinition 这 3 个接口，而无论其底层关联的具体的事务究竟是 JDBC 事务、JTA 事务，还是 ORM 框架自定义的事务。

PlatformTransactionManager 定义了事务管理器，所有与事务相关的操作都由 PlatformTransactionManager 管理；TransactionStatus 定义了事务状态，PlatformTransactionManager 会根据 TransactionStatus 的状态来决定是否回滚事务；TransactionDefinition 则定义了事务的隔离级别和传播行为，在启动事务时，PlatformTransactionManager 根据 TransactionDefinition 来启动合适的事务。

事务的隔离级别实际上是由底层数据库系统实现的，在前面我们已经简单地讨论过了。事务的传播行为则是应用程序自己管理的，它决定了事务如何在应用程序中传播，一般总是由底层框架来完成。Spring 支持如下的事务传播行为，这和 EJB 声明式事务的传播行为是一致的。

（1）PROPAGATION_REQUIRED：必须在事务内执行，如果当前存在事务，就加入到当前事务中；如果当前没有事务，就创建一个新事务。这是最常见的选择，也是 Spring 默认的事务传播行为。

（2）PROPAGATION_SUPPORTS：支持当前事务，但如果当前没有事务，也可以

以非事务方式执行。

（3）PROPAGATION_MANDATORY：必须在当前事务内执行，如果当前没有事务，就直接抛出异常。

（4）PROPAGATION_REQUIRES_NEW：总是新建一个事务，如果当前存在事务，就把当前事务挂起，直到新事务执行完毕。

（5）PROPAGATION_NOT_SUPPORTED：不能在事务环境下执行，如果当前存在事务，就把当前事务挂起。

（6）PROPAGATION_NEVER：不能在事务环境下执行，如果当前存在事务，就直接抛出异常。

（7）PROPAGATION_NESTED：必须在事务内执行，如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则执行与 PROPAGATION_REQUIRED 类似的操作。

6.2 使用编程式事务管理

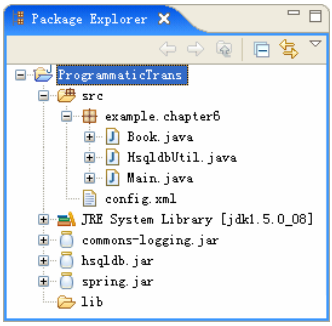


图 6-1

使用 Spring 的编程式事务只需要和上述 3 个核心接口打交道，我们在 Eclipse 中新建一个 ProgrammaticTrans 工程，结构如图 6-1 所示。

HsqldbUtil 和第 5 章的 DaoTemplate 工程中的 HsqldbUtil 一样，负责启动和初始化 HSQL 数据库，由于是以内存模式运行的，因此，程序结束后也无需关闭数据库资源，这样能让我们将注意力集中在 Spring 的事务管理上。

为了在 Spring 中使用编程式事务管理，需要在 XML 配置文件中定义 DataSource 和 PlatformTransactionM ananger。首先，定义 DataSource 如下。

```
        <bean id="dataSource" class="org.springframework.jdbc.datasource. Driver
ManagerDataSource">
            <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
            <property name="url" value="jdbc:hsqldb:mem:bookstore" />
            <property name="username" value="sa" />
            <property name="password" value="" />
        </bean>
```

对于 PlatformTransactionManager，需要根据我们当前的需求选择一个实现类。在这个工程中，由于我们直接采用 JDBC 操作数据库，因此必须选择 DataSource Transaction

Manager，定义如下。

```
        <bean id="transactionManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager">
            <property name="dataSource" ref="dataSource" />
        </bean>
```

现在，PlatformTransactionManager 已经和 DataSource 关联好了，使用 Spring 的编程式事务管理，在 Main 方法中编写的典型代码如下。

```
        public class Main {
            public static void main(String[] args) {
                HsqldbUtil.startDatabase();
                ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
                DataSource dataSource = (DataSource)context.getBean("dataSource");
                queryAllBooks(dataSource);
                // 获取 PlatformTransactionManager:
                PlatformTransactionManager transManager = (PlatformTransactionManager)
context.getBean("transactionManager");
                try {
                    doTransaction(transManager, dataSource);
                }
                finally {
                    queryAllBooks(dataSource);
                    System.exit(0);
                }
            }

            private static void doTransaction(PlatformTransactionManager transManager,
DataSource dataSource) {
                // 定义 TransactionDefinition:
                DefaultTransactionDefinition transDef = new DefaultTransaction Definition();
                // 定义 Transaction 隔离级别:
                transDef.setIsolationLevel(TransactionDefinition.ISOLATION_READ_ COMMITTED);
                // 定义 Transaction 传播行为:
                transDef.setPropagationBehavior(TransactionDefinition.PROPAGATION_
REQUIRED);
                // 开始一个 Transaction:
                TransactionStatus ts = transManager.getTransaction(transDef);
                try {
                    insertBook(dataSource);
                }
                catch(RuntimeException e) {
                    e.printStackTrace();
```

```
                // 回滚事务:
                transManager.rollback(ts);
                throw e;
            }
            catch(Error e) {
                e.printStackTrace();
                // 回滚事务:
                transManager.rollback(ts);
                throw e;
            }
            // 提交事务:
            transManager.commit(ts);
        }

        private static void queryAllBooks(DataSource dataSource) {
            List<Book> books = new JdbcTemplate(dataSource).query("select * from
Book", new BookRowMapper());
            System.err.println("-- All Books ---------------------------");
            for(Book book : books) {
                System.err.println("  " + book.getName() + ", by " + book.
getAuthor());
            }
            System.err.println("----------------------------------------");
        }

        private static void insertBook(DataSource dataSource) {
            JdbcTemplate jdbcTemp = new JdbcTemplate(dataSource);
            jdbcTemp.update(
                    "insert into Book(id, name, author) values(?, ?, ?)",
                    new Object[]{"123-456-789", "New Book", "New Author"});
        }
    }

    class BookRowMapper implements RowMapper {
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Book book = new Book();
            book.setId(rs.getString("id"));
            book.setName(rs.getString("name"));
            book.setAuthor(rs.getString("author"));
            return book;
        }
    }
```

为了简化 JDBC 代码，我们仍采用 JdbcTemplate 和 RowMapper 实现数据库操作，以便将注意力集中在事务管理上。

使用 Spring 的编程式事务管理时，典型的步骤如下。

① 从 Spring 的 IoC 容器中获取 PlatformTransactionManager 实例。

② 定义 TransactionDefinition 并设置好事务的隔离级别和传播方式。

③ 通过 PlatformTransactionManager.getTransaction()开始一个事务，并获得 TransactionStatus 对象。

④ 运行需要在事务环境下执行的代码并捕获异常。

⑤ 如果有异常发生，将 TransactionStatus 设置为 setRollbackOnly()，表示将要回滚事务。

⑥ 调用 PlatformTransactionManager.commit(TransactionStatus)方法提交事务，Spring 根据 TransactionStatus 的状态来决定如何提交事务，如果已经调用了 setRollbackOnly()，事务将回滚；如果没有调用过 setRollbackOnly()，则事务将被提交。

为了检查事务是否真的起作用，我们在事务执行前后分别打印出数据库表中的所有记录。程序运行结果如下。

```
-- All Books --------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
---------------------------------------
...
-- All Books --------------------------
  New Book, by New Author
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
---------------------------------------
```

可以看到，新的记录被正确插入了。如果我们调用两次 insertBook()方法，则肯定会由于插入重复主键导致第二次 INSERT 语句执行失败，此时，代码运行结果如下。

```
-- All Books --------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
---------------------------------------
...
Exception in thread "main" org.springframework.dao.DataIntegrityViolation
Exception:  ...;
Caused by: java.sql.SQLException: Unique constraint violation: Unique
constraint violation: ...
...
-- All Books --------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
```

```
----------------------------------------
```

可以看到，由于回滚了事务，两条新记录都没有被真正插入到数据库中。

我们仅捕获了 RuntimeException 和 Error，在这两种情况下将事务回滚，这也是 Spring 默认的事务回滚条件。

另一种无需捕获异常的方式是使用 TransactionTemplate，这样，TransactionTemplate 会自动启动和提交事务，并在异常抛出时回滚事务，代码如下。

```
TransactionTemplate tt = new TransactionTemplate(transManager);
tt.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
tt.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        // TODO: jdbc code here...
        return null;
    }
});
```

如果需要显式地回滚事务，则没有必要抛出一个 RuntimeException，可以调用 TransactionStatus 对象的 setRollbackOnly()方法。一旦该方法被调用过，在提交事务时，TransactionManager 就会自动回滚事务。

现在，我们已经对 Spring 的编程式事务管理有了一个初步的了解。Spring 将不同的事务类型都统一抽象为 PlatformTransactionManager，因此，我们只需要以同一种方式对事务进行管理，无需关心底层事务究竟是 JDBC 事务还是 JTA 事务，或是 Hibernate 封装的事务。事实上，Spring 的声明式事务管理也是建立在这种抽象事务之上的。

下面我们要讨论的一个问题是 Spring 的 PlatformTransactionManager 是如何与底层数据库资源关联起来的。在上面的例子中，我们使用的是 JDBC 数据源，其事务也是 JDBC 事务。在 XML 配置文件中，我们仅仅将 PlatformTransactionManager 和 DataSource 关联起来了，而真正的 JDBC 事务必须和一个 JDBC Connection 关联。现在，假定调用 PlatformTransactionManager 的 getTransaction()方法启动一个事务时，PlatformTransaction Manager 自动从 DataSource 中获得了一个 Connection，现在的问题是，接下来的数据库操作（不管是直接调用 JDBC 接口，还是使用 JdbcTemplate）如果不能保证其获取的 Connection 和 PlatformTransactionManager 获取的 Connection 是同一个 Connection，则整个事务管理肯定都是错误的。例如，如下代码。

```
TransactionStatus ts = transManager.getTransaction(transDef);
Connection newConn = dataSource.getConnection();
// TODO: 对 newConn 操作
transManager.commit(ts);
```

由于从 DataSource 返回的 newConn 肯定是另一个不同于 PlatformTransactionManager 获得的 Connection，因此，PlatformTransactionManager 的事务对 newConn 无效，newConn 连接根本不在 PlatformTransactionManager 的管理之下。

因此，为了使事务范围内的数据库操作能正确获得 PlatformTransactionManager 的 Connection，我们**决不能**直接调用 DataSource 的 getConnection()方法。Spring 提供了一个 DataSourceUtils 工具类，其 getConnection(DataSource)方法能返回预先和 PlatformTransactionManager 绑定的 Connection，这样才能保证数据库事务的正确执行。因此，上面的代码应修改如下。

```
Connection newConn = DataSourceUtils.getConnection(dataSource);
```

使用 JdbcTemplate 时，Spring 直接使用了 DataSourceUtils 来获取 Connection，所以 ProgrammaticTrans 工程中的代码是没有问题的。

事实上，PlatformTransactionManager 在启动事务时会将当前事务的连接注册到 TransactionSynchronizationManager，DataSourcUtils 也将首先试图从 TransactionSynchronizationManager 中获取已经关联到当前事务的 Connection。可见，不了解 Spring 框架的内部设计就无法正确地使用 Spring 提供的接口，因此，深入到 Spring 框架内部是很有必要的。

6.3 使用声明式事务管理

6.2 节我们讨论了在 Spring 中使用编程式事务管理，如果一个 DAO 对象有许多方法，每个方法都需要纳入事务管理，使用编程式事务管理就必须重复编写大量的代码，因此，我们很自然地想到，能否应用 AOP 将事务管理作为一个切面，动态插入到各个方法的执行前后。实际上，Spring 已经替我们做好了，这就是 Spring 提供的声明式事务管理，并且将整个事务配置都放入了 XML 配置文件中。本节将详细讨论如何在 Spring 中使用声明式事务管理来大大简化应用程序的事务模型。

我们复制一份 ProgrammaticTrans 工程，命名为 DeclarativeTrans，结构如图 6-2 所示。

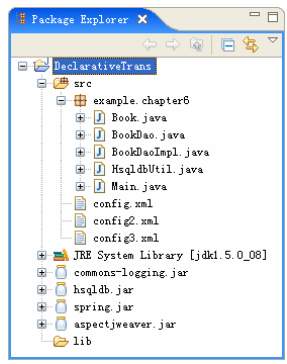


图 6-2

BookDao 接口和 BookDaoImpl 实现类是 DAO 模式的应用，和第 5 章 DaoTemplate 工程的 BookDao 接口和 JdbcTemplateBookDao 实现类完全相同。不过，在 DeclarativeTrans 工程中，我们需要将 BookDao 封装为具有事务功能的 DAO

接口。

我们先使用 Spring 提供的 TransactionProxy FactoryBean 来实现具有声明式事务功能的 BookDao，然后讨论如何使用 Spring 2.0 支持的<tx>命名空间更方便地实现声明式事务。

在 XML 配置文件 config.xml 中定义 DataSource 和 PlatformTransactionManager。

```
        <!-- 定义 DataSource -->
        <bean id="dataSource" class="org.springframework.jdbc.datasource.Driver
ManagerDataSource">
            <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
            <property name="url" value="jdbc:hsqldb:mem:bookstore" />
            <property name="username" value="sa" />
            <property name="password" value="" />
        </bean>

        <!-- 定义 TransactionManager -->
        <bean id="transactionManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager" >
            <property name="dataSource" ref="dataSource" />
        </bean>
```

然后，定义 BookDao 的一个 Singleton。

```
        <!-- 定义 BookDao -->
        <bean id="bookDaoTarget" class="example.chapter6.BookDaoImpl">
            <property name="dataSource" ref="dataSource" />
        </bean>
```

最后，我们通过 TransactionProxyFactoryBean 将 bookDaoTarget 封装为具有事务功能的 bookDao。

```
        <!-- 封装为事务的 BookDao -->
        <bean id="bookDao" class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean">
            <property name="target" ref="bookDaoTarget" />
            <property name="transactionManager" ref="transactionManager" />
            <property name="transactionAttributes">
                <props>
                    <!-- 对以 query 开头的方法要求只读事务 -->
                    <prop key="query*">PROPAGATION_REQUIRED,readOnly</prop>
                     <!-- 对于其他方法要求事务 -->
                    <prop key="*">PROPAGATION_REQUIRED</prop>
                </props>
            </property>
        </bean>
```

由于 TransactionProxyFactoryBean 是一个具有 AOP 代理功能的 FactoryBean，其返回的对象是 AOP 代理，具有和 bookDaoTarget 对象一致的接口，因此客户端使用 bookDao 时，就拥有了事务功能。

我们在 TransactionProxyFactoryBean 的配置中对于以 query 开头的方法配置为只读的事务，对于其他方法（如 insert 等）要求默认的事务，这样可以让底层 JDBC 驱动优化数据库事务，在读操作远多于写操作的应用中，能大大提高数据库事务的性能。

我们在 main()方法中编写如下测试代码。

```
public static void main(String[] args) {
    HsqldbUtil.startDatabase();
   ApplicationContext context = new ClassPathXmlApplicationContext ("config. xml");
    BookDao bookDao = (BookDao)context.getBean("bookDao");
    display(bookDao.queryAll());
    try {
        Book newBook = new Book();
        newBook.setId("new-id-1234567890");
        newBook.setName("New Book");
        newBook.setAuthor("New Author");
        bookDao.insert(newBook);
     }
    catch(Exception e) {
        System.out.println(e.getClass().getName());
    }
    display(bookDao.queryAll());
    System.exit(0);
}
```

上述代码运行结果如下。

```
-- All Books ---------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
----------------------------------------
-- All Books ---------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
  New Book, by New Author
----------------------------------------
```

但是，我们还不能确定配置的声明式事务管理是否有效。如果在 BookDaoImpl 类的 insert()方法中抛出 RuntimeException，我们看看事务是否会回滚。

```
public void insert(Book book) {
```

```
        getJdbcTemplate().update(
                "insert into Book(id, name, author) values(?,?,?)",
                new Object[] {book.getId(), book.getName(), book.getAuthor()});
        throw new UnsupportedOperationException("Cause rollback");
    }
```

再次运行 main()方法，可以看到，由于抛出了运行时异常，因此，INSERT 操作被回滚。

```
-- All Books ---------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
----------------------------------------
java.lang.UnsupportedOperationException
-- All Books ---------------------------
  Thinking in Java, by Bruce Eckel
  JUnit in Action, by Craig Walls
----------------------------------------
```

默认情况下，Spring 只在 RuntimeException 异常或 Error 抛出时回滚事务，如果抛出了 CheckedException 异常，Spring 仍会提交事务。如果需要在 CheckedException 异常抛出时也回滚事务，就需要在 TransactionProxyFactoryBean 配置中显式地将每个 CheckedException 都添加进来。

```
<prop key="*">PROPAGATION_REQUIRED,-RemoteException,-SocketException </prop>
```

使用 Spring 的声明式事务管理后，我们就无需直接调用 Spring 的事务 API 来手动控制事务，进一步大大简化了代码，但是，事务边界就被限制在每个具有事务功能的对象方法中，在方法调用时开始事务，在方法返回时提交事务，在 RuntimeException 抛出时回滚事务。在上面的例子中，由于我们将 BookDao 封装为具有声明式事务的对象，因此，事务边界被限制在 BookDao 的每一个方法调用前后。

6.3.1 使用<tx:>简化配置

下面我们讨论如何在 Spring 2.0 框架中采用新的更简化的配置方式来实现声明式事务。在 Spring 2.0 框架中，引入了新的<tx:>命名空间，可以使 XML 配置更加简化。为了使用<tx:>和<aop:>命名空间，必须修改 XML 的根元素<beans>。

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:tx="http://www.springframework.org/schema/tx"
```

```
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
```

定义好 dataSource 和 transactionManager 后：

```
<bean id="dataSource" class="..." />
<bean id="transactionManager" class="..." />
```

就可以直接声明一个 bookDao 对象，然后将其包装为具有声明式事务功能的 AOP 代理对象。

```
<!-- 定义 BookDao -->
<bean id="bookDao" class="example.chapter6.BookDaoImpl">
   <property name="dataSource" ref="dataSource" />
</bean>

<!-- 将 BookDao 封装成具有事务功能 -->
<!-- 声明 TxAdvice -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
   <tx:attributes>
      <!-- 对以 query 开头的方法要求只读事务 -->
      <tx:method name="query*" read-only="true" />
      <!-- 对于其他方法要求事务 -->
      <tx:method name="*" />
   </tx:attributes>
</tx:advice>

<aop:config>
   <!-- 使用 AspectJ 语法定义 Pointcut -->
   <aop:pointcut id="bookDaoOperation" expression="execution(* example.
chapter6.BookDao.*(..))" />
   <!-- 织入 -->
   <aop:advisor advice-ref="txAdvice" pointcut-ref="bookDaoOperation" />
</aop:config>
```

由于使用了 AspectJ 的语法实现 Pointcut，因此需要导入 aspectjweaver.jar。整个 XML 配置文件请参考 DeclarativeTrans 工程的 src/config2.xml 文件。

要注意的是，使用 Spring 2.0 新的配置方式时，客户端获取的 bookDao 是通过 AOP 织入后的具有事务功能的 AOP 代理对象，而不是原始的 BookDaoImpl 实例，这样就可

以保证客户端无法直接获取 BookDaoImpl 的实例。在上一个 config.xml 配置文件中，客户端是可以通过“bookDaoTarget”来直接获取 BookDaoImpl 的实例的，这样就绕过了 Spring 的声明式事务管理功能，应用程序的逻辑完整性就会遭到破坏。

<tx:method>除了 name 外，还可以有如表 6-2 所示的配置选项，以便更精确地配置事务行为。

表 6-2

属　性	必　需	默 认 值	描　述
name	是		与事务关联的方法名，可以使用通配符“*”，例如，query*
propagation	否	REQUIRED	定义事务的传播行为
isolation	否	DEFAULT	定义事务的隔离级别
read-only	否	false	是否是只读事务
time-out	否	–1	事务超时（以秒为单位）
rollback-for	否		将触发回滚操作的 CheckedException，如果有多个，以逗号分隔，例如，java.io.IOException,javax.servlet.ServletException.
no-rollback-for	否		不会触发回滚操作的 RuntimeException，例如，java.lang.ArithmeticException

6.3.2 使用Java 5 注解简化配置

Spring 2.0 框架提供的第 3 种声明式事务的配置是基于 Java 5 注解，此时，声明式事务通过 Java 5 注解在源代码中配置，在 XML 配置文件中只需要编写一行配置即可。当然，<tx:>的命名空间仍必不可少。下面的 config3.xml 配置文件列出了使用 Java 5 注解将 bookDao 包装为声明式事务的配置方式。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <!-- 定义 DataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource. Driver
ManagerDataSource">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
```

```
            <property name="url" value="jdbc:hsqldb:mem:bookstore" />
            <property name="username" value="sa" />
            <property name="password" value="" />
        </bean>

        <!-- 定义 TransactionManager -->
        <bean id="transactionManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager" >
            <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- 定义 BookDao -->
        <bean id="bookDao" class="example.chapter6.BookDaoImpl">
            <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- 将所有具有@Transactional 注解的 Bean 自动配置为声明式事务支持 -->
        <tx:annotation-driven transaction-manager="transactionManager"/>
    </beans>
```

只需要一行<tx:annotation-driven ... />就可以自动将所有具有@Transactional 注解的 Bean 全部自动包装为具有声明式事务功能的 Bean，大大简化了声明式事务的配置。如果正在使用 Java 5，应该首先考虑这种配置方式。

对于上述 XML 配置文件，我们还需要对 BookDao 接口编写适当的注释来声明事务。

```
@Transactional
public interface BookDao {
    @Transactional(readOnly=true)
    Book query(String id);

    @Transactional(readOnly=true)
    List<Book> queryAll();

    void insert(Book book);

    void update(Book book);

    @Transactional(propagation=Propagation.REQUIRES_NEW)
    void delete(Book book);
}
```

这种配置方式最大的大优点是，只要在接口或类的声明处编写一个@Transactional 注解，所有方法都将自动继承这个注解的设置，当然，具体的某一个方法还可以覆盖默认的设置。例如，query()方法将 readOnly 设置为 true，而 delete 方法将 propagation 设置

为 REQUIRES_NEW。@Transactional 的所有可选设置如表 6-3 所示。

表 6-3

属　性	类　型	默 认 值	描　述
propagation	Propagation 枚举	REQUIRED	事务传播属性
isolation	Isolation 枚举	DEFAULT	事务隔离级别
read-only	boolean	false	是否只读
timeout	int	–1	超时（秒）
rollbackFor	Class[]	{}	需要回滚的异常类
rollbackForClassName	String[]	{}	需要回滚的异常类名
noRollbackFor	Class[]	{}	不需要回滚的异常类
noRollbackForClassName	String[]	{}	不需要回滚的异常类名

例如，下面的注解声明了事务传播属性为 REQUIRES_NEW，隔离级别为 READ_COMMITED，超时为 5 秒，遇到 IOException 和 ServletException 将回滚，而遇到 ArithmeticException 则不回滚。

```
@Transactional(
    propagation=Propagation.REQUIRES_NEW,
    isolation=Isolation.READ_COMMITTED,
    timeout=5,
    rollbackFor={IOException.class,ServletException.class},
    noRollbackForClassName={"java.lang.ArithmeticException"}
)
```

读者可能会问，@Transactional 注解能否标注到具体实现类而不是接口，例如，BookDaoImpl？由于 Spring AOP 默认使用 JDK 动态代理，因此，既可以识别标注在接口的@Transactional 注解，也可以识别具体类的@Transactional 注解。但是，如果使用 CGLIB 实现 AOP 代理，就只能识别出具体类的@Transactional 注解，这是因为@Transactional 注解是不可继承的。因此，如果要保证无论使用何种 AOP 实现，都能正确配置声明式事务，最好直接在实现类上使用@Transactional 注解。

另一个需要注意的是，@Transactional 注解只能被应用到 public 方法上。对于其他非 public 方法，如果标注了@Transactional 注解，Spring 也不会报错，但是该方法不会被配置为具有声明式事务功能，因为 Spring 会忽略掉所有非 public 方法标注的@Transactional 注解。

需要注意的最后一点是，只使用@Transactional 注解并不能实现声明式事务。事实上，声明式事务的功能是由<tx:annotation-driven ... />这一行配置文件开启的，所以，千万不要忘记了在 XML 配置文件的最后添加<tx:annotation-driven ... />。

6.4 集成Hibernate事务

Hibernate 是一个轻量级的 O/R Mapping 框架，它对 JDBC API 进行了一层薄薄的封装。Hibernate 已经将 JDBC 事务或 JTA 事务纳入自身的管理范围之内。在使用 Hibernate 之前，我们需要首先理解 Hibernate 的事务模型。

请读者注意，本书仅讨论 Hibernate 3.2 版本的使用，因为 Spring 2.0 涉及的 JPA 等新特性仅在 Hibernate 3.2 或更高版本才能支持。

在第 5 章中，我们已经简单介绍了 Hibernate 中最常用的 SessionFactory、Session 和 Transaction 对象，但是我们还没有讨论如何使用 Hibernate 的 Transaction。

在单独使用 Hibernate 时，如果使用 Hibernate 默认配置，则一个典型的 Hibernate 事务代码如下。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
try {
    // TODO: do with session:
    tx.commit();
}
catch(Throwable e) {
    tx.rollback();
}
finally {
    session.close();
}
```

Hibernate 还允许将当前的 Session 和 JDBC 或 JTA 事务关联起来，这样就可以使用 SessionFactory.getCurrentSession()直接获得与当前事务关联的 Session，这在 Web 应用程序中尤其有用，如果事务需要跨多个对象的方法调用，则每个对象获取的 Session 都必须是同一个 Session，使用这种方式就保证了获取的 Session 总是在当前事务中，确保了事务的完整性，却不用在各个对象间传递 Session 引用。

要将 Session 绑定到 JDBC 事务上，需要在 Hibernate 配置文件中设置。

```
hibernate.transaction.factory_class = org.hibernate.transaction. JDBCTransaction
Factory
hibernate.current_session_context_class = thread
```

这样，当前的 Session 和关联的 Hibernate 事务被绑定到当前线程上。

要将 Session 绑定到 JTA 事务上，需要在 Hibernate 配置文件中设置。

```
        hibernate.transaction.factory_class = org.hibernate.transaction. JTATransaction
Factory
        hibernate.current_session_context_class = jta
```

一旦绑定了 JDBC 或 JTA 事务，就可以非常方便地使用 SessionFactory.getCurrent Session()获得与当前事务关联的 Session。通常，推荐编写一个简单的 HibernateUtil 来封装这个方法，并负责初始化 SessionFactory。

```
public final class HibernateUtil {
    // SessionFactory 实例:
    private static final SessionFactory sessionFactory;
    static {
        // 初始化 SessionFactory:
        try {
            sessionFactory = new AnnotationConfiguration()
                    .configure()
                    .buildSessionFactory();
        }
        catch(Exception e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    // 返回 currentSession:
    public static Session getCurrentSession() {
        return sessionFactory.getCurrentSession();
    }
}
```

在这种方式下，典型的 Hibernate 代码如下。

```
Transaction tx = HibernateUtil.getCurrentSession().beginTransaction();
try {
    // do with currentSession:
    process();
    tx.commit();
}
catch(Throwable t) {
    tx.rollback();
}
```

在启动一个 Hibernate Transaction 后，在 process()中调用 HibernateUtil.getCurrent Session()获得的 Session 就是与当前 Transaction 关联的 Session，这样，其他组件就可以直接使用这个 Session，当事务提交或回滚时，当前 Session 会被自动关闭，因此，不要

在 finally 中关闭 Session，否则将会得到一个 SessionException 异常。

使用上述 Hibernate 事务模型时，最大的优点是事务的边界非常清晰，在 Web 应用程序中，通常是一个请求对应一个 Session，此时，通过一个前端控制器（例如，Filter，将在第 7 章中详细讨论）开启并提交或回滚事务，其他组件只需要随时调用 HibernateUtil.getCurrentSession()获得与当前 Transaction 关联的 Session 并使用它即可，大大简化了事务管理，并且免去了在各个组件中编写烦琐的 try { ... } catch() { ... }语句。

6.4.1 在Spring中集成Hibernate事务

Spring 对 Hibernate 进行了很好的集成，在 Spring 中使用 Hibernate 事务非常容易。我们建立一个 HibernateTrans 工程，部分文件从 DeclarativeTrans 工程复制过来，然后将 Hibernate 依赖的 jar 包复制到 lib 目录下，整个工程结构如图 6-3 所示。

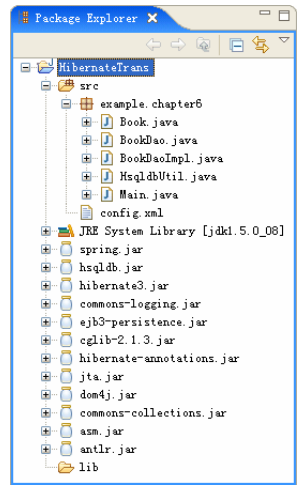


图 6-3

我们先讨论使用 Spring 的声明式事务管理，Hibernate Trans 工程需要实现以下功能。

（1）使用 Hibernate 作为持久化框架。

（2）使用 JPA 注解来配置实体 Bean，免去使用 XML 配置 Hibernate 的麻烦。

（3）使用 HibernateDaoSupport 实现 DAO。

（4）使用 Spring 的声明式事务管理和 Java 5 注解来配置声明式事务。

在第 5 章中我们已经介绍了如何使用 JPA 注解，请读者参考第 5 章 DaoTemplate 工程的 Book 类注解，这里不再重复。HibernateDaoSupport 的使用和 DaoTemplate 工程的 HibernateBookDao 非常类似，本例中，BookDaoImpl 如下。

```
public class BookDaoImpl extends HibernateDaoSupport implements BookDao {
    public Book query(String id) {
        Object obj = getHibernateTemplate().load(Book.class, id);
        if(obj==null)
            throw new DataRetrievalFailureException("id not found: " + id);
        return (Book) obj;
    }
    public List<Book> queryAll() {
        return getHibernateTemplate().find("select b from Book as b");
    }
    public void insert(Book book) {
```

```
            getHibernateTemplate().save(book);
        }
        public void update(Book book) {
            getHibernateTemplate().update(book);
        }
        public void delete(Book book) {
            getHibernateTemplate().delete(
                    getHibernateTemplate().load(Book.class, book.getId()));
        }
    }
```

我们在 BookDao 接口中对声明式事务进行注解。

```
@Transactional
public interface BookDao {
    @Transactional(readOnly=true)
    Book query(String id);
    @Transactional(readOnly=true)
    List<Book> queryAll();
    void insert(Book book);
    void update(Book book);
    void delete(Book book);
}
```

最后一步是编写 Spring 的 XML 配置文件，在 config.xml 中，我们需要定义的依次是 DataSource、SessionFactory、HibernateTransactionManager、BookDao 和<tx:annotation -driven ... />打开自动根据注解配置声明式事务的功能。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <!-- 定义 DataSource -->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
        <property name="url" value="jdbc:hsqldb:mem:bookstore" />
        <property name="username" value="sa" />
        <property name="password" value="" />
```

```
        </bean>

        <!-- 定义 SessionFactory -->
        <bean id="sessionFactory" class="org.springframework.orm.hibernate3.
annotation.AnnotationSessionFactoryBean">
            <property name="dataSource">
                <ref bean="dataSource"/>
            </property>
            <property name="annotatedClasses">
                <list>
                    <value>example.chapter6.Book</value>
                </list>
            </property>
            <property name="annotatedPackages">
                <list>
                    <value>example.chapter6</value>
                </list>
            </property>
         </bean>

         <!-- 定义 TransactionManager -->
        <bean id="transactionManager" class="org.springframework.orm.hibernate3.
HibernateTransactionManager" >
            <property name="sessionFactory" ref="sessionFactory" />
        </bean>

        <!-- 定义 BookDao -->
        <bean id="bookDao" class="example.chapter6.BookDaoImpl">
            <property name="sessionFactory" ref="sessionFactory" />
        </bean>

        <!-- 将所有具有@Transactional 注解的 Bean 自动配置为声明式事务支持 -->
        <tx:annotation-driven transaction-manager="transactionManager"/>
    </beans>
```

运行 main()方法，其效果与上一个 DeclarativeTrans 工程一样，不过我们已经成功地使用 Hibernate 替换了 JDBC 来实现持久化。

如果需要手动控制事务，我们仍然无需和 Hibernate 的事务 API 打交道，只需操作 Spring 提供的 PlatformTransactionManager，和前面的 ProgrammaticTrans 工程中操作 JDBC 事务的代码非常类似。这正是 Spring 提供的一致的事务编程模型带来的好处。典型的代码如下。

```
SessionFactory sf = (SessionFactory)context.getBean("sessionFactory");
PlatformTransactionManager tm = (PlatformTransactionManager)
        context.getBean("transactionManager");
```

```
// 启动事务:
TransactionStatus ts = tm.getTransaction(
       new DefaultTransactionDefinition());
try {
    Session session = SessionFactoryUtils.getSession(sf, false);
    // TODO: 在此执行任意的 Hibernate 操作
}
catch(RuntimeException e) {
    tm.rollback(ts);
    throw e;
}
catch(Error e) {
    tm.rollback(ts);
    throw e;
}
// 提交事务:
tm.commit(ts);
```

注意，不要关闭 Session，在提交或回滚事务时，Session 将被自动关闭。换句话说，Session 的获取和释放要全部交给 Spring 完成，除非自己手动从 SessionFactory 中获取了一个新的 Session（不推荐手动从 SessionFactory 中获取新的 Session）。

和前面我们提到的 JDBC Connection 问题类似，我们必须保证获取的 Session 和 PlatformTransactionManager 关联的 Session 是同一个实例，否则，Session 根本不在 PlatformTransactionManager 的事务控制之下。为此，务必使用 Spring 提供的 SessionFactory Utils 的 getSession()方法获取 Session，第二个参数 boolean allowCreate 表示是否允许创建新的 Session，在 PlatformTransactionManager 的事务范围内，绝不允许创建新的 Session，或者通过 SessionFactory.openSession()打开一个新的 Session。只有这样才能保证获取的 Session 和 PlatformTransactionManager 关联的是同一个 Session 实例。

使用 HibernateTemplate 时，其使用的 Session 正是通过 SessionFactoryUtils 的 getSession()方法获取的，因此可以放心使用。

另一个需要注意的问题是，使用 Spring 来管理 Hibernate 事务时，千万不要使用 Hibernate 自己的事务绑定，即不要配置将 Hibernate 的事务绑定到 Thread 或 JTA 上，也不可使用 Hibernate 自己的事务 API，这样会使 Spring 的资源管理混乱，造成难以预计的结果。Spring 对事务的绑定是通过 TransactionSynchronizationManager 实现的，无论对于何种事务均有一致的模型。

6.5 确定事务边界

在介绍了 Spring 的编程式事务管理和声明式事务管理模型后，我们来讨论一下事务

边界。所谓事务边界是指事务开始和结束的地方。使用编程式事务管理时，启动和提交事务都是由程序控制的，因此，事务的边界就被确定在启动事务和提交事务的代码处。使用声明式事务管理时，事务的边界就需要根据应用程序的实际情况确定。

通常，数据库操作由 DAO 对象封装，DAO 对象中的每个方法都封装了一个相对独立的数据库操作，如果使用 Spring 的声明式事务管理将 DAO 对象变为具有事务功能的代理对象（例如，上面的例子中的 BookDao 对象），则对于 DAO 对象的每个方法调用均是一个独立的事务，事务边界就被确定在 DAO 对象的每个方法的调用前后，如图 6-4 所示。

如果事务需要跨越多个 DAO 对象的方法，即将多个 DAO 对象的方法调用放到一个事务中执行，例如，在用户提交订单时，同时更新用户的积分信息。

```
orderDao.createOrder(order);
userDao.updateUser(user);
```

将事务边界定义在每个 DAO 方法调用上，就无法保证多个 DAO 方法的事务完整性，此时，必须将事务边界定义在上层的对象中，例如，在 Service 对象中。

```
void order(Order order, User user) {
    orderDao.createOrder(order);
    userDao.updateUser(user);
}
```

将 Service 对象通过声明式事务变为具有事务功能的对象后，事务的边界就变为 Service 对象的每个方法调用前后。在 Service 对象的一个方法内调用多个 DAO 对象的方法就可以保证这些操作作为一个事务正确执行，如图 6-5 所示。

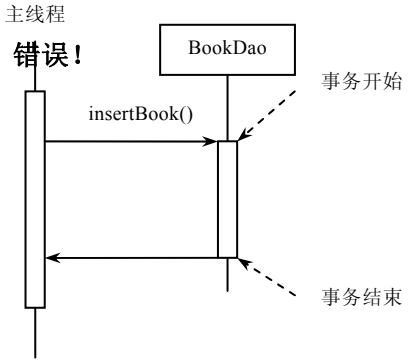


图 6-4

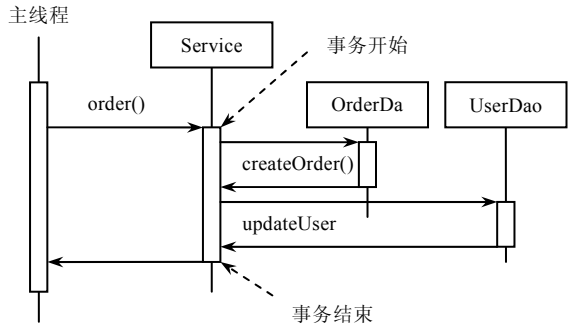


图 6-5

一些持久层框架（例如，Hibernate）还支持延迟加载的特性，在 Web 应用程序的视图渲染时才从数据库读取所需的数据，在这种设计下，事务边界就必须被扩大为每个 HTTP 请求的处理开始和结束时，这一点在第 11 章 Live 在线书店应用中还将详细讨论。

6.6 小结

本章我们讨论了事务的基本概念、JDBC 事务、JTA 事务和 Spring 提供的抽象事务模型。

我们还详细介绍了如何使用 Spring 提供的统一的抽象事务模型来针对各种事务实现统一的编程模型。通过 Spring 的事务抽象，我们便可以以一种模式来处理事务，不必关心底层事务究竟是 JDBC 还是 JTA 事务，或是 Hibernate 事务。

Spring 还提供了声明式事务管理的强大功能，这使得不必应用负责的 EJB 服务器，我们也同样有能力对普通的 Java Bean 组件实现声明式事务，极大地简化了事务的处理。在设计应用程序结构时，应当首先考虑使用 Spring 的声明式事务管理，绝大多数情况下，Spring 的声明式事务管理完全能满足应用程序的需求。

我们还讨论了 Hibernate 的事务模型，包括单独使用 Hibernate 和在 Spring 中集成 Hibernate。由于 Spring 将 Hibernate 事务也纳入了统一的 PlatformTransactionManager 抽象之下，因此，Spring 有自己的 Hibernate 事务绑定模式，读者务必注意，不要混合使用 Hibernate 自身提供的事务和 Spring 的 HibernateTransactionManager。

在应用程序中正确处理事务的关键是建立明确的事务边界。使用编程式事务管理（无论是使用 Spring 编程式事务管理，还是直接使用 JDBC 等事务）时，最好将事务的启动、提交和回滚操作置于一个统一的前端入口，而不要将这些代码散落在各个组件中。使用 Spring 的声明式事务管理时，事务的边界被明确地确立在方法的调用前后，我们必须保证在一个事务方法中能够正确处理用户请求，如果一个操作需要先后执行两个事务方法，就需要考虑是否有必要在更高一层的组件上实现 Spring 的声明式事务，例如，在 Service 层，而不是 DAO 层。

第 7 章

使用Spring MVC框架

JavaEE 为 Web 开发提供了强大的支持。为了实现拥有良好结构的、可扩展的 Web 应用程序，各种 Web 框架层出不穷。Spring 框架除了作为优秀的 IoC 容器之外，其本身也提供了一个完整的 Web MVC 模块。

本章将详细介绍 JavaEE Web 开发的基础知识，以及如何使用 Spring MVC 框架开发出灵活的、可扩展的 Web 应用程序。本章还将介绍如何集成现有的一些流行的 MVC 框架，例如，Struts 和 WebWork，并比较它们和 Spring MVC 框架的优劣。

7.1 JavaEE Web基础

7.1.1 HTTP协议简介

HTTP（HyperText Transfer Protocol，超文本传输协议）协议是 Web 应用所使用的最主要的协议。以浏览器为界面的 Web 应用程序均是以 HTTP 协议为基础的请求相应模式。浏览器作为客户端向服务器发送一个请求，服务器收到请求后，将响应返回给客户端。图 7-1 显示了浏览器访问 http://www.sina.com.cn/的请求和响应。

图 7-1

HTTP 是一个无状态协议，浏览器和服务器的交互包括以下步骤。

① 浏览器向服务器请求建立 TCP 连接。

② 连接建立后，浏览器发送 HTTP 请求给服务器。

③ 服务器将响应内容发送给浏览器。

④ 双方关闭 TCP 连接。

如果服务器支持 HTTP 1.1 版本，则第 ②、③ 步可以多次执行，以便减少 TCP 连接

的次数，从而提高网络效率。

HTTP 请求由请求方式、URL 和数据三部分构成，最常见的 HTTP 请求是 GET 请求和 POST 请求。

GET 请求仅仅给服务器发送一个 URL，可以在 URL 中包含参数，然后期待服务器返回相应的内容。一个完整的 GET 请求的 URL 格式如下。

http://www.livebookstore.net/listBooks.jspx

与 GET 请求相比，POST 请求的参数不包含在 URL 中，而是以附加的消息体发送给服务器。POST 请求的数据不会显示在浏览器的地址栏，因此用户无法看到。

由于 HTTP 协议是无状态的，而 Web 应用程序常常需要跟踪用户的身份，因此，服务器通常使用以下两种方式来保存用户状态。

（1）使用 Cookie 来标识用户。浏览器在第一次请求服务器时将获得服务器传递给它的 Cookie，此后的请求中，浏览器将 Cookie 附加在请求中，服务器就可以识别出用户身份。

（2）通过 URL 重写的方式来跟踪用户。服务器通过将响应页面中的 URL 链接附加上一个特定的标识符，就可以跟踪用户身份。

对于一个用户来说，在浏览器和服务器之间的反复的请求响应被称为一个会话。由于服务器的资源是有限的，因此，会话有一个超时设置。如果用户长时间没有通过浏览器请求服务器，服务器就认为此会话结束。选择一个合适的会话超时是必要的，过短的会话会导致用户操作不便，过长的会话会导致服务器负担过重。通常，JavaEE 服务器的默认会话超时（例如，30 分钟）是一个比较合理的设置。

在对 HTTP 协议有基本了解后，我们需要了解 JavaEE 的两种 Web 组件标准：Servlet 和 JSP。它们是整个 JavaEE Web 应用程序的基础。

7.1.2 Servlet组件

Servlet 组件是 JavaEE 中最核心的 Web 标准。Servlet 运行于 Web 容器中，按照请求/响应模式为用户提供服务。一个典型的 Servlet 代码如下。

```
        public class HelloServlet extends HttpServlet {
            protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
                PrintWriter pw = response.getWriter();
                pw.print("<html><boay><h1>Hello, world!</h1></body></html>");
            }
            protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
                doGet(request, response);
            }
        }
```

doGet 方法和 doPost 方法分别对应 HTTP 的 GET 请求和 POST 请求。Servlet API 定义了 HttpServletRequest 对象和 HttpServletResponse 对象，Web 容器负责将这两个对象传递给 Servlet 组件，开发人员需要从 HttpServletRequest 对象中获取需要的参数，然后将生成的页面写入 HttpServletResponse 对象的输出流中，即完成了一次完整的请求/响应。

由于 Servlet 组件必须运行在 Web 容器中，因此，要运行上面的示例，必须将其部署到 Web 服务器上。我们以 Resin 服务器为例，建立一个 HelloServlet 项目，结构如图 7-2 所示。

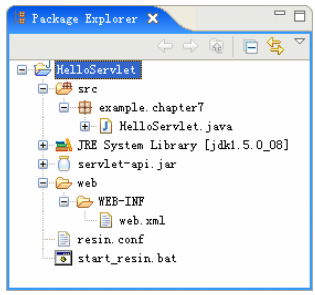


图 7-2

web 目录是网站的根目录，WEB-INF 目录（注意：区分大小写）存放了 Web 应用程序所需的全部文件，在 Web 应用程序运行期，服务器不允许用户直接通过 URL 访问 WEB-INF 目录下的任何文件，这就保证了服务器端代码不会被客户端所获得。在 WEB-INF 目录下，classes 目录存放编译好的 class 文件，这里我们直接设置项目输出路径为 HelloServlet/web/WEB-INF/classes，如图 7-3 所示。

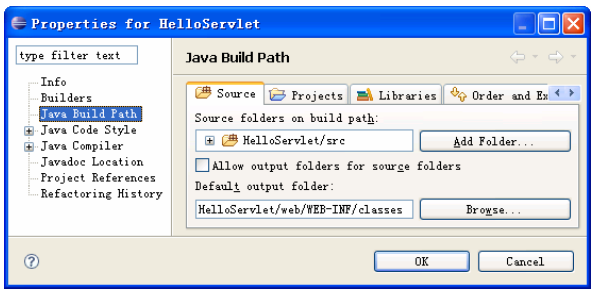


图 7-3

为了编译 HelloServlet，需要将 servlet-api.jar 引入到 Java Build Path 中。注意，这个文件仅在编译时需要用到，在 Web 应用程序的运行期不需要它，因为 Web 服务器内置了所有的 Servlet API。

web/WEB-INF/lib 目录存放了所有用到的第三方库文件，在这个简单的项目中我们

没有用到任何第三方库，在后面的章节中，如果用到了第三方库，都需要将其加入到工程的 Java Build Path 中。web.xml 是整个 Web 应用程序最基本的配置文件。要让 HelloServlet 工作，需要在 web.xml 中定义它，并为其配置 URL 映射。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>helloServlet</servlet-name>
        <servlet-class>example.chapter7.HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>helloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

在启动 Resin 前，请配置好环境变量 JAVA_HOME 和 RESIN_HOME，然后编写一个 resin.conf 配置文件，为了方便启动，我们还编写了一个 start_resin.bat 批处理文件来启动 Resin。读者可以导入本书配套光盘中的配置文件。

```
PATH=%JAVA_HOME%\bin;%PATH%
set RESIN_CONF=%CD%/resin.conf
CD web
%RESIN_HOME%\httpd -server-root %CD% -conf %RESIN_CONF%
```

运行 start_resin.bat 启动 Resin，由于我们配置的 HelloServlet 的映射路径为/hello，因此，在浏览器地址栏中输入“http://localhost:8080/hello”，即可看到 HelloServlet 在浏览器中运行的效果，如图 7-4 所示。

图 7-4

在每个 HTTP 请求到来时，Web 服务器都会创建 HttpServletRequest 对象和 HttpServletResponse 对象来封装 HTTP 请求和输出，然后传递给 Servlet 组件处理。除了 HttpServlet 作为核心的 Web 组件用于处理 HTTP 请求外，Web 容器还提供了

ServletContext 对象和 Session 对象来简化 Web 应用程序的开发。ServletContext 对象在一个 Web 应用程序中有且仅有一个，它封装了应用程序所需的常用信息；Session 对象负责管理一个会话，Web 容器为每一个客户端创建一个独立的 Session，Web 应用程序可以将客户端的相关信息放入其各自的 Session 中，以便管理。

7.1.3 JSP组件

由于在 Servlet 中输出 HTML 页面极其困难且难以维护，因此，JavaEE 还提供了另一种以 HTML 为主的 JSP 组件。在一个 JSP 页面中，大部分为 HTML 标签，仅用<% ... %>嵌入小部分 Java 代码，因此，JSP 降低了网页设计的难度。

当用户请求一个 JSP 页面时，JSP 页面首先要被 Web 容器转化为 Servlet 源代码，然后被编译为 class 文件并执行。通常，编译过程在首次请求时被执行。如果原始的 JSP 文件做了更新，则 Web 容器会检测到更新并自动对其重新编译。

以下是一个典型的 JSP 页面。

```
<html>
    <body>
        <h1><% out.print("Hello, world!"); %></h1>
    </body>
</html>
```

和 Servlet 相比，JSP 页面的部署就非常简单，不需要在 web.xml 中定义，直接在地址栏输入 JSP 页面的路径即可。可以从本书的配套光盘中导入 HelloJsp 项目，如图 7-5 所示。

然后启动 Resin，输入“http://localhost:8080/hello.jsp”，其执行效果和 7.1.2 节的 Servlet 示例完全一样，如图 7-6 所示。

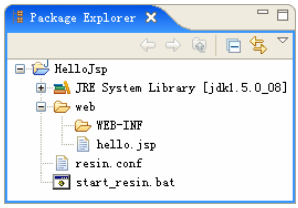


图 7-5

图 7-6

由于 JSP 在执行前将首先被转化为 Servlet 源代码，然后被编译为 class 文件并载入执行。第一次请求 JSP 时，需要一个转化和编译的过程，因此时间较长，而后续的请求就可以跳过转化和编译过程，因此速度会快得多。可以在/WEB-INF/work 目录下找到由

hello.jsp 页面转化的 Servlet 源代码和编译后的 class 文件。

JSP 和 Servlet 另一个不同之处在于，服务器会自动检测 JSP 页面的改动。若发现更改，则自动重新编译，而改动 Servlet 的 class 文件后，只有重新启动服务器或者重新加载 Web 应用程序后才会生效。

7.1.4 JSP标签

JSP 页面的输出通常是调用隐含的 out 对象的 print()方法，这样做会导致复杂的 Java 代码。为了简化输出，SUN 又在 JSP 中引入了标签的概念。标签就是封装了一些功能并能够输出 HTML 片段的 Java 类，使用起来却和 HTML 代码差不多。例如，使用了 JSTL（JSP Standard Tag Library）标签库的 JSP 页面输出“Hello, world!”和直接使用 out 对象的 JSP 页面相比，格式更简洁，但是必须在页面开头处声明使用的标签库。

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
    <body>
        <h1><c:out value="Hello, world!" /></h1>
    </body>
</html>
```

JSP 标签的功能相当强大，不但可以输出 HTML 片段，还可以执行任意 Java 代码，甚至包括执行 SQL 语句等，但是，滥用 JSP 标签将造成页面逻辑混乱，给后期维护造成极大的困难，此外，自己开发定制标签库也不是一件容易的事情。我们强烈建议，对标签库的使用应该严格限制那些只用于显示输出结果的标签，不要使用诸如执行 SQL 查询之类的标签。那么，处理请求的逻辑应该在放在何处呢？MVC 模式正是为了解决逻辑处理和显示输出相分离而提出的一种设计模式。稍后我们还将介绍 MVC 模式的原理和在 Web 应用中的实现方式。

7.1.5 Filter

从 Servlet 2.3 规范开始，还引入了另一个激动人心的特性——Filter（过滤器）。Filter 组件有能力在请求被传递给 Servlet 或 JSP 组件处理前截获请求，通过修改 HttpServletRequest 或 HttpServletResponse，实现诸如安全检查、定制输出等许多功能。此外，可以将多个 Filter 组合在一起形成过滤链，使请求能被链上的每一个 Filter 依次处理。

一个最简单的 Filter 实现如下。

```
public class SimpleFilter implements Filter {
    public void init(FilterConfig filterConfig) throws ServletException {
        // 在此初始化 Filter
    }
    public void destroy() {
        // 在此释放资源
    }
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        // 千万不要忘了调用 chain.doFilter(),否则请求无法被继续处理:
        chain.doFilter(request, response);
    }
}
```

过滤器在 doFilter 方法中处理请求，在 SimpleFilter 中，我们简单地调用 chain.doFilter() 将请求直接传递给下一个过滤器。要特别注意，如果忘记了调用 chain.doFilter()，那么请求处理就会到此结束，客户端很可能得不到任何输出。

下面的例子演示了如何利用 Filter 来实现安全检查。这个 SecurityFilter 将向客户端返回 403 禁止访问的错误。

```
public class SecurityFilter implements Filter {
    public void init(FilterConfig filterConfig) throws ServletException {}
    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        ((HttpServletResponse)response).sendError(403);
        // 没有调用 chain.doFilter()，因为已经向客户端发送了 403 错误
    }
}
```

在 web.xml 中，我们规定 SecurityFilter 将过滤/security 目录下的所有资源，即访问 /security 目录下的任何资源，都会得到一个 403 禁止访问的错误。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <filter>
        <filter-name>securityFilter</filter-name>
        <filter-class>example.chapter7.SecurityFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>securityFilter</filter-name>
```

```
        <url-pattern>/security/*.jsp</url-pattern>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
</web-app>
```

我们创建如下的 SecurityFilter 工程，在 web 目录下包含两个 jsp 文件，其中，security.jsp 放在/security 目录下，如图 7-7 所示。

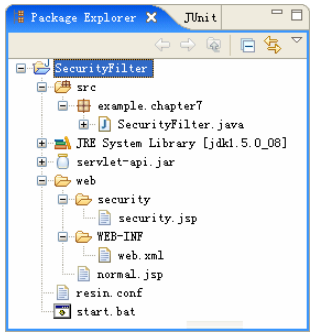


图 7-7

打开浏览器，访问/normal.jsp 页面是没有问题的，如图 7-8 所示。

如果访问/security/security.jsp，则会得到一个 403 错误，这说明 SecurityFilter 起作用了，如图 7-9 所示。

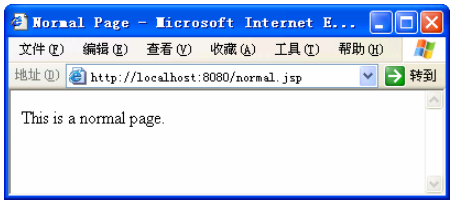


图 7-8

图 7-9

Filter 不仅能对请求实现预处理，还能定制输出。利用这个特性，可以对输出进行后处理，例如，对输出的内容进行 GZip 压缩，以加快网络传输。下面的 GZipFilter 的例子演示了如何对输出内容进行压缩，可以从本书的配套光盘中找到工程源代码。

实现定制输出的关键是对 HttpServletResponse 进行包装，截获所有的输出，等到过滤器链处理完毕后，再对截获的输出进行处理，并写入到真正的 HttpServletResponse 对象中。JavaEE 框架已经定义了一个 HttpServletResponseWrapper 类使得包装 HttpServletResponse 更加容易。我们扩展这个 HttpServletResponseWrapper，截获所有的输出，并保存到 ByteArrayOutputStream 中。

```
public class Wrapper extends HttpServletResponseWrapper {
```

```
        public static final int OT_NONE = 0, OT_WRITER = 1, OT_STREAM = 2;
        private int outputType = OT_NONE;

        private ServletOutputStream output = null;
        private PrintWriter writer = null;
        private ByteArrayOutputStream buffer = null;

        public Wrapper(HttpServletResponse resp) throws IOException {
            super(resp);
            buffer = new ByteArrayOutputStream();
        }
        public PrintWriter getWriter() throws IOException {
            if(outputType==OT_STREAM)
                throw new IllegalStateException();
            else if(outputType==OT_WRITER)
                return writer;
            else {
                outputType = OT_WRITER;
                writer = new PrintWriter(new OutputStreamWriter(buffer, get
CharacterEncoding()));
                return writer;
            }
        }
        public ServletOutputStream getOutputStream() throws IOException {
            if(outputType==OT_WRITER)
                throw new IllegalStateException();
            else if(outputType==OT_STREAM)
                return output;
            else {
                outputType = OT_STREAM;
                output = new WrappedOutputStream(buffer);
                return output;
            }
        }
        public void flushBuffer() throws IOException {
            if(outputType==OT_WRITER)
                writer.flush();
            if(outputType==OT_STREAM)
                output.flush();
        }
        public void reset() {
            outputType = OT_NONE;
            buffer.reset();
        }
        public byte[] getResponseData() throws IOException {
            flushBuffer();
            return buffer.toByteArray();
```

```
    }
    class WrappedOutputStream extends ServletOutputStream {
        private ByteArrayOutputStream buffer;
        public WrappedOutputStream(ByteArrayOutputStream buffer) {
            this.buffer = buffer;
        }
        public void write(int b) throws IOException {
            buffer.write(b);
        }
        public byte[] toByteArray() {
            return buffer.toByteArray();
        }
    }
}
```

然后，在 GZipFilter 的 doFilter()方法中对输出进行 GZip 压缩。

```
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
    HttpServletResponse resp = (HttpServletResponse)response;
    Wrapper wrapper = new Wrapper(resp);
    chain.doFilter(request, wrapper);
    byte[] gzipData = gzip(wrapper.getResponseData());
    resp.addHeader("Content-Encoding", "gzip");
    resp.setContentLength(gzipData.length);
    ServletOutputStream output = response.getOutputStream();
    output.write(gzipData);
    output.flush();
}
```

gzip()方法负责将一个 byte[]数组的内容进行 GZip 压缩。

```
private byte[] gzip(byte[] data) {
    ByteArrayOutputStream byteOutput = new ByteArrayOutputStream(10240);
    GZIPOutputStream output = null;
    try {
        output = new GZIPOutputStream(byteOutput);
        output.write(data);
    }
    catch (IOException e) {}
    finally {
        try {
            output.close();
        }
        catch (IOException e) {}
    }
    return byteOutput.toByteArray();
}
```

我们在 web.xml 中配置 GZipFilter 过滤的 URL 为*.html，然后通过一个 HTML 文件进行测试，在使用 GZipFilter 和不使用 GZipFilter 的情况下，通过 ieHTTPHeaders 这个插件查看 IE 浏览器获得的服务器返回，如图 7-10 和图 7-11 所示。

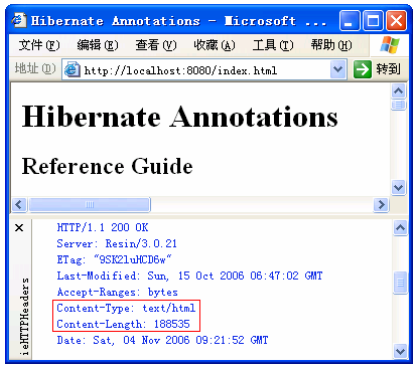


图 7-10

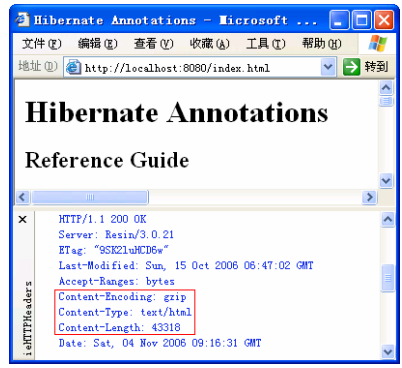


图 7-11

可以看到，没有使用 GZipFilter 时，原始 HTML 文件的大小是 188,535 字节，使用 GZipFilter 压缩后的文件大小是 43,318 字节，压缩了大约 77%，效果是非常显著的。

如果读者对于 Apache 的 mod_rewrite 熟悉就一定知道 URL Rewrite 在 Apache 中是非常强大的功能，它能很容易地使动态 URL 以.html 的静态形式更友好地展示给用户。在 JavaEE Web 应用程序中，应用 Filter 同样可以实现这一强大功能。

事实上，Resin 服务器已经自带了一个实现 URL 重写的 Filter，不过它依赖于 Resin 的某些 jar 包。不过，我们可以从 Resin 的源代码中获得这个 RewriteFilter 的源代码，然后稍做修改，就可以立刻应用在任何 JavaEE Web 应用程序中。

从 Resin 的官方站点 http://www.caucho.com/下载 Resin 3.0.x 的源代码，解压后将 RewriteFilter.java 提取出来，然后在 Eclipse 中建立以下 UrlRewrite 工程，如图 7-12 所示。

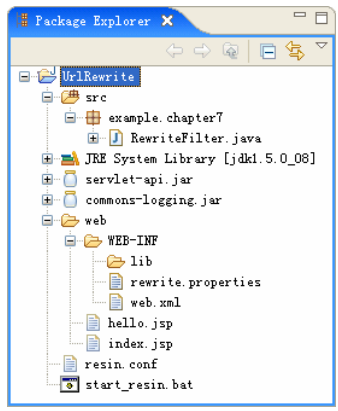


图 7-12

我们修改 RewriteFilter.java，主要将其与 Resin 的 jar 包依赖的类去掉，用 StringBuffer 替换 com.caucho.util.CharBuffer（CharBuffer 是 StringBuffer 的一个非线程安全版本，因此性能会稍微高一点），用 org.apache.commons.logging.Log 替换 com.caucho.log.Log，此外，Resin 的 RewriteFilter 是以非标准方式配置 Rewrite 规则的，我们将使用/WEB-INF/rewrite.properties 作为 Rewrite 规则的配置文件。整个 RewriteFilter 代码如下。

```
public class RewriteFilter implements Filter {
    private final Log log = LogFactory.getLog(RewriteFilter.class);
    private ServletContext _app;
    private ArrayList<RewriteEntry> _entries = new ArrayList<RewriteEntry>();

    public void init(FilterConfig config) throws ServletException {
        _app = config.getServletContext();
        // 读取配置文件:
       File conf_file = new File(
            _app.getRealPath("/WEB-INF/rewrite. properties"));
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(conf_file));
            for(;;) {
                String s = reader.readLine();
                if(s==null)
                    break;
                if(s.trim().equals(""))
                    continue;
                if(s.trim().startsWith("#"))
                    continue;
                String[] ss = s.trim().split(" ", 2);
                if(ss.length!=2) {
                    log.warn("Invaild rule: " + s);
                }
                else {
                    log.info("Add rewrite entry: " + s);
                    _entries.add(new RewriteEntry(ss[0], ss[1]));
                }
            }
        }
        catch(IOException ioe) {
            log.warn("Exception in init RewriteFilter.", ioe);
        }
        finally {
            if(reader!=null) {
                try {
                    reader.close();
```

```
                }
                catch(IOException e) {}
            }
        }
    }

    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain nextFilter)throws ServletException, IOException
    {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;
        String url = req.getRequestURI();
        for (int i = 0; i < _entries.size(); i++) {
            RewriteEntry entry = _entries.get(i);
            Pattern pattern = entry.getRegexp();
            Matcher matcher = pattern.matcher(url);
            if (! matcher.find(0))
                continue;
            String replacement = replace(matcher, entry.getTarget());
            String query = req.getQueryString();
            if (query != null) {
                if (replacement.indexOf('?') > 0)
                    replacement = replacement + '&' + query;
                else
                    replacement = replacement + '?' + query;
            }
            log.info("forwarding '" + url + "' to '" + replacement + "'");
            if (replacement.startsWith("/")) {
              RequestDispatcher disp = _app.getRequestDispatcher(replacement);
                disp.forward(request, response);
                return;
            }
            else {
                res.sendRedirect(res.encodeRedirectURL(replacement));
                return;
            }
        }
        nextFilter.doFilter(request, response);
    }

    private String replace(Matcher matcher, String target) {
        StringBuffer cb = new StringBuffer(512);
        for (int i = 0; i < target.length(); i++) {
```

```
            char ch = target.charAt(i);
            if (ch != '$' || i == target.length() - 1)
                cb.append(ch);
            else {
                ch = target.charAt(i + 1);
                if (ch >= '0' && ch <= '9') {
                    int group = ch - '0';
                    cb.append(matcher.group(group));
                    i++;
                }
                else if (ch == '$') {
                    cb.append('$');
                    i++;
                }
                else
                    cb.append('$');
            }
        }
        return cb.toString();
    }

    public static class RewriteEntry {
        private Pattern _pattern;
        private String _target;
        public RewriteEntry(String pattern, String target) {
            _pattern = Pattern.compile(pattern);
            _target = target;
        }
        public Pattern getRegexp() { return _pattern; }
        public String getTarget() { return _target; }
    }
}
```

URL 的重写规则和 Apache 的完全一致，但是不支持域名的重写。为了测试 RewriteFilter，我们编写了两个 jsp 文件，放在 web 目录下，然后在/WEB-INF/rewrite.properties 中定义了两条 URL 重写规则。

```
^\/index\-(.*)\.html$ /index.jsp?id=$1
^\/hello\-(.*)\.html$ /hello.jsp?name=$1
```

在 web.xml 中配置好 RewriteFilter 后，启动 Resin，打开浏览器测试，如图 7-13 和图 7-14 所示。

可以看到，后缀为.html 的 URL 被自动重写为带有参数的.jsp 的 URL，但是 JSP 页面获得的 URL 是重写后的 URL，因此不会影响 JSP 页面的正常运行。

这个 RewriteFilter 虽然简单，但是在 Web 应用程序中是极其有用的。使用 RewriteFilter 可以使开发阶段不必集成 Apache 即可测试 Web 应用程序。将来如果需要集成 Apache，只需要在 Apache 中配置好 mod_rewrite 模块，然后简单地将 RewriteFilter 移除即可。

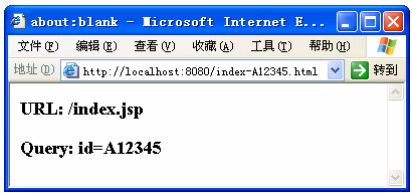


图 7-13

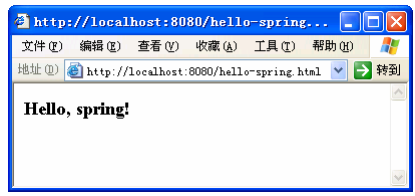


图 7-14

从上面 3 个例子可以看出，Filter 可以实现对输入和输出的定制，其设计类似于 AOP 的切面，能非常方便地插入需要的功能而不修改核心 Web 组件的代码。

7.2 MVC概述

比较 Servlet 和 JSP 可以看出，Servlet 是在 Java 代码中嵌入 HTML，而 JSP 则是在 HTML 中嵌入 Java 代码，两者的优缺点正好互补，如表 7-1 所示。

表 7-1

	Servlet	JSP
编写 Java 代码	容易	困难
编写 HTML 代码	困难	容易

如果仅使用 Servlet，则虽然应用程序逻辑编写简单，但是，编码输出的页面极其难以维护；如果仅使用 JSP，则虽然简化了页面设计，但大量的 Java 代码将使得应用程序逻辑难以维护。

因此，若能将 Servlet 和 JSP 结合使用，使得 Servlet 和 JSP 各司其职，就能大大降低 Web 应用程序的开发难度。

MVC（Model-View-Controller）即模型-视图-控制器，这种模式最早由 Smalltalk 语言引入，作为应用程序界面的标准设计模式，其应用范围极为广泛。MVC 模式通过将显示界面分为 3 个核心模块（模型、视图和控制器），分别负责不同的任务，协作完成一次完整的请求/响应。

图 7-15 显示了 MVC 模式的响应流程。

随着 Web 应用开发的兴起，MVC 也被引入了 Web 应用程序中。由于 Servlet 是纯 Java 代码，因此适合担任控制器，控制请求的流程。Servlet 通过查询模型的状态，获得需要的数据，然后传递结果给视图，由视图将页面渲染出来，返回给用户。由于 JSP 主要是 HTML 标签，因此适合作为视图，仅负责渲染出结果页面。

受到 HTTP 协议的限制，模型的状态如果发生了更改，无法主动通知视图，因此，每次请求必须由浏览器发起。相对于传统的主动 MVC 模式，在 Web 应用程序中必须使用被动的 MVC 模式，如图 7-16 所示。**错误！**

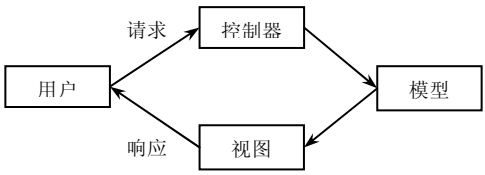


图 7-15

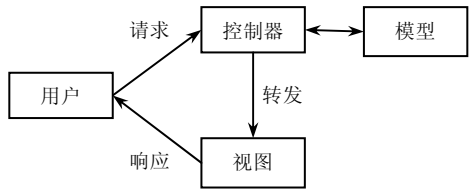


图 7-16

在 JavaEE Web 模型中，一个 HTTP 请求可以以 3 种转发方式。

（1）Forward：将一个请求传递给另一个组件处理，可以是 Servlet 或 JSP，而当前的组件将不再继续处理请求。这也是 MVC 模式中转发请求让视图来渲染模型时使用的方式。

（2）Include：将一个请求传递给另一个组件处理，但是其处理结果将被包含在当前组件的输出页面中，通常，Include 方式用来包含网站每个页面都共享的页眉和页脚，但是许多网页编辑工具本身就支持页面模版，因此，Include 方式的使用并不多。

（3）Error：当处理请求发生了异常时，将被导向到指定的错误页面。

JavaEE 平台上已经有许多成熟的 Web MVC 框架，例如，Struts 和 WebWork，其原理大同小异。在介绍如何使用 Spring 的 MVC 框架前，有必要首先弄清楚 MVC 的实现原理。

根据 MVC 的设计思想，一个完整的 HTTP 请求需要经过以下几步处理。

① 所有的 HTTP 请求都将映射到一个 Servlet 上。

② Servlet 根据 URL 选择合适的 Controller 来处理请求，获得返回的 Model 和 View。

③ Servlet 将 Model 绑定到 Request 中，然后交给 View 渲染。

④ View 将渲染后的页面返回给用户，完成请求处理。

由于存在多个控制器，如果直接用 Servlet 作为控制器，则需要实现多个 Servlet，这会导致非常复杂的配置和大量的重复代码。因此，通常将 Servlet 作为一个请求入口，或者称之为前置控制器（Front Controller），然后根据 URL 选择一个合适的控制器处理，具体流程如图 7-17 所示。

错误！

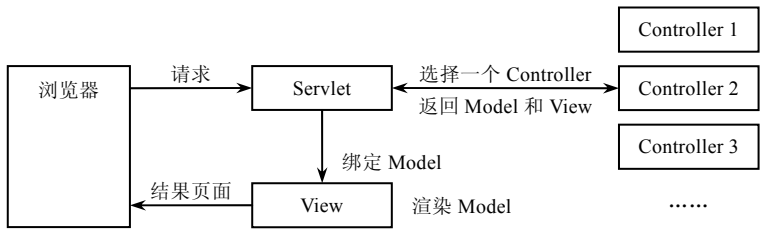


图 7-17

由于 Servlet 需要知道 URL 和控制器的映射关系，所以还需要首先抽象出 Controller。通常，在 Web MVC 框架中，我们将抽象的 Controller 称为控制器，将负责接收用户请求的 Servlet 称为前置控制器或请求转发器（DispatcherServlet）。下面，我们在 Eclipse 中建立一个 SimpleMVC 项目，自己动手实现一个最简单的 MVC 流程，如图 7-18 所示。

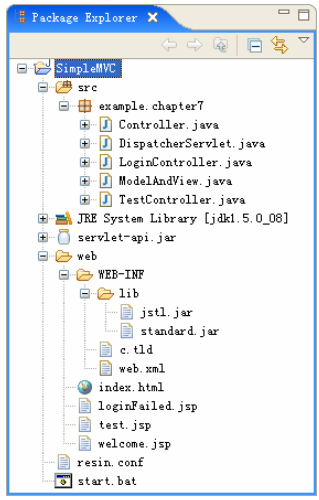


图 7-18

7.2.1 设计Controller

根据面向接口编程的良好实践，首先定义一个 Controller 接口。

```
public interface Controller {
    ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception;
}
```

Controller 需要返回视图和将要由视图渲染的 Model。视图通常是 JSP 页面，因此，返回 JSP 页面的路径即可，而 Model 可以简单地使用 Map 来实现。由于一个方法只能返

回一个对象，因此，有必要将 Model 和 View 一起返回。设计 ModelAndView 如下。

```
public class ModelAndView {
    private String view;
    private Map<String, Object> model;
    public ModelAndView(String view, Map<String, Object> model) {
        this.view = view;
        this.model = model;
    }
    public String getView() { return view; }
    public Map<String, Object> getModel() { return model; }
}
```

下一步是实现一个具体的 TestController，它从 request 中获得 URL 的参数，然后生成 Model 并返回 ModelAndView。

```
public class TestController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
Response response) throws Exception {
        String name = request.getParameter("name");
        if(name==null) name = "world";
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("message", "This is a TestController.");
        model.put("greet", "Hello, " + name + "!");
        model.put("time", new Date().toString());
        // 返回的 JSP 视图路径是"/test.jsp":
        return new ModelAndView("/test.jsp", model);
    }
}
```

在实现了 Controller 后，现在需要定义的就是如何在 Servlet 中控制整个请求的处理流程。

7.2.2 实现请求转发

通常，我们把接收所有请求的 Servlet 命名为 DispatcherServlet，在 SimpleMVC 项目中，我们把所有以“.cmd”结尾的 URL 都映射到 DispatcherServlet，然后在 DispatcherServlet 中，根据 URL 选择合适的 Controller，实现请求转发。我们假定现在定义了两个 Controller：TestController 和 LoginController，其对应的 URL 分别是“/test.cmd”和“/login.cmd”，在 DispatcherServlet 中处理如下。

```
public class DispatcherServlet extends HttpServlet {
    private ServletContext context;
```

```
        private Map<String, Controller> controllers;
        // 初始化 DispatcherServlet:
        public void init(ServletConfig config) throws ServletException {
            context = config.getServletContext();
            // 将 URL 和对应的 Controller 关联起来:
            controllers = new HashMap<String, Controller>();
            controllers.put("/test.cmd", new TestController());
            controllers.put("/login.cmd", new LoginController());
        }
        protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
            String contextPath = request.getContextPath();
            String uri = request.getRequestURI().substring(contextPath.length());
            Controller controller = controllers.get(uri);
            if(controller==null) {
                // 没有找到合适的 Controller 处理请求，直接返回一个 404 错误:
                response.sendError(404);
                return;
            }
            try {
                // 将请求交给 Controller 处理:
                ModelAndView mv = controller.handleRequest(request, response);
                // 将 Model 绑定到 Request 中:
                Map<String, Object> model = mv.getModel();
                Set<String> keys = model.keySet();
                for(String key : keys)
                    request.setAttribute(key, model.get(key));
                // 获得 JSP 视图的路径:
                String jsp = mv.getView();
                // 转发给 JSP 视图渲染:
                context.getRequestDispatcher(jsp).forward(request, response);
            }
            catch(Exception e) {
                // 处理过程中发生异常，返回一个 500 错误:
                response.sendError(500);
            }
        }
        protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
            // POST 请求的处理方式和 GET 一样:
            doGet(request, response);
        }
    }
```

DispatcherServlet 将 Controller 返回的 Model 绑定到 Request，然后使用 forward()将

Request 交给 JSP 页面渲染。在 JSP 页面中，只需要使用 JSP 标签把 Model 的数据显示出来，因此，不需要也不应该嵌入任何 Java 代码。test.jsp 页面源码如下。

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>TestController -&gt; test.jsp</title>
    </head>
    <body>
        <p>/test.cmd (TestController) -&gt; test.jsp </p>
        <p>Message: <c:out value="${message}" /></p>
        <p>Greet: <c:out value="${greet}" /></p>
        <p>Time: <c:out value="${time}" /></p>
    </body>
</html>
```

如果用户请求 URL “/test.cmd?name=spring”，则整个请求按照以下流程处理，如图 7-18 所示。

错误！

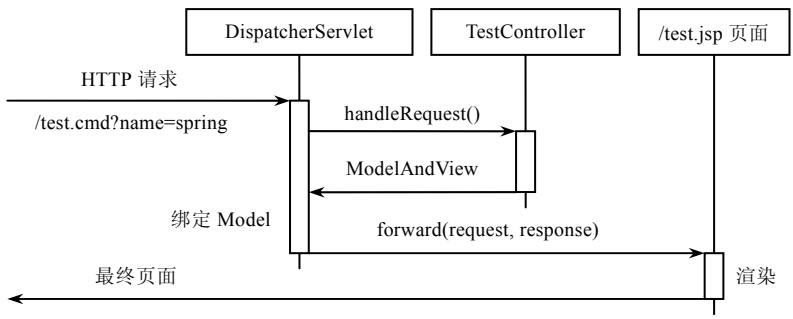


图 7-19

LoginController 则稍微复杂一点，它根据登录是否成功来决定返回哪一个 ModelAndView。

```
public class LoginController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
Response response) throws Exception {
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        if("admin".equals(username) && "password".equals(password)) {
            // 登录成功，将转入/welcome.jsp:
            Map<String, Object> model = new HashMap<String, Object>();
            model.put("username", username);
```

```
            model.put("greet", "Welcome!");
            model.put("time", new Date().toString());
            return new ModelAndView("/welcome.jsp", model);
        }
        // 登录失败，将转入/loginFailed.jsp:
        Map<String, Object> error = new HashMap<String, Object>();
        error.put("message", "Login failed.");
        error.put("time", new Date().toString());
        return new ModelAndView("/loginFailed.jsp", error);
    }
}
```

为了能在 Resin 中测试我们自己实现的这个 SimpleMVC，需要在 web.xml 中注册 DispatcherServlet，由于 JSP 页面还使用了 JSTL 标签库，因此，还要在 web.xml 中注册标签库。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>example.chapter7.DispatcherServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.cmd</url-pattern>
    </servlet-mapping>
    <taglib>
        <taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>
        <taglib-location>/WEB-INF/c.tld</taglib-location>
    </taglib>
</web-app>
```

其他依赖的 jar 包请参考本书配套光盘中的 SimpleMVC 项目设置。启动 Resin，输入 http://localhost:8080/test.cmd?name=spring 和 http:// localhost: 8080/index.html，分别测试 TestController 和 LoginController，看看是不是我们预期的结果，如图 7-20 所示。

图 7-20

Spring 的 MVC 实现原理和我们自己实现的简单的 MVC 类似，只不过添加了更多的功能和更简便的配置。下面，我们讲解如何使用 Spring MVC 开发 Web 应用程序。

7.3 Spring MVC基础

在 7.2 节中，我们自己实现了一个最简单的 MVC 架构。实际上，Spring MVC 的实现原理与此类似。Spring 提供了 DispatcherServlet，这个类不仅负责实现请求转发，还负责启动一个 WebApplicationContext 容器。按照 Spring 一贯的 IoC 哲学，所有的 Controller 都是 JavaBean，并由 IoC 容器统一管理。对于 View，则采取了更灵活的处理方式，Spring MVC 允许使用不同的 View 实现，除了 JSP 外，还可以使用 Velocity、Freemaker、XSLT 等多种 View 技术。

总的来讲，要使用 Spring MVC 框架，需要以下步骤。

① 在 web.xml 中配置 DispatcherServlet 及 URL 映射。

② 编写 IoC 容器需要的 XML 配置文件，命名为<servlet-name>-servlet.xml，放到/WEB-INF 目录下。例如，如果 DispatcherServlet 在 web.xml 中的配置名称为 dispatcher，则 Spring 将寻找 dispatcher-servlet.xml 配置文件。

③ 在 XML 配置文件中定义 URL 映射方式和使用哪种 View 技术。

我们仍以 SimpleMVC 为基础，用 Spring MVC 框架来实现这个 Web 应用程序。在 Eclipse 中创建一个 SpringMVC 项目，结构如图 7-21 所示。

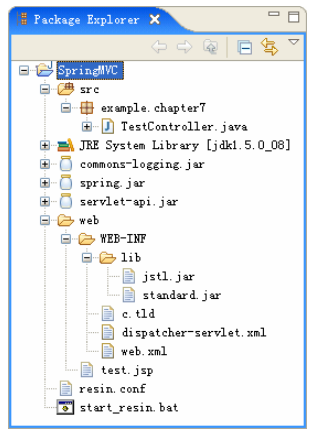


图 7-21

/web/WEB-INF/lib 目录下的 jstl.jar 和 standard.jar 是 JSP 标准标签库，将在 JSP 视图中用到，/web/WEB-INF/c.tld 是标签库的声明文件，稍后我们会在 JSP 视图文件中用到它们。

7.3.1 配置DispatcherServlet

首先，在 web.xml 中配置 DispatcherServlet，并将所有以“.html”结尾的 URL 全部映射到 DispatcherServlet 上，这样，用户仅从 URL 地址上无法得知服务器端后台使用了何种技术。

```
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
        <web-app>
            <!-- spring dispatch servlet -->
```

```
            <servlet>
                <servlet-name>dispatcher</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
                <load-on-startup>0</load-on-startup>
            </servlet>
            <servlet-mapping>
                <servlet-name>dispatcher</servlet-name>
                <url-pattern>*.html</url-pattern>
            </servlet-mapping>
            <!-- 申明 taglib,将在 JSP 中使用 -->
            <taglib>
                <taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>
                <taglib-location>/WEB-INF/c.tld</taglib-location>
            </taglib>
        </web-app>
```

和普通的 Spring 应用程序稍有不同，对于 Web 应用程序，Spring 的 ApplicationContext 是由 DispatcherServlet 加载的，它会在/WEB-INF/下查找一个名称为<servletName>-servlet.xml 的 XML 配置文件来初始化 Spring Web 应用程序的 ApplicationContext。对于上例，我们在 web.xml 中定义 DispatcherServlet 的名称为 dispatcher，因此，相应的 XML 配置文件就必须是/WEB-INF/dispatcher-servlet.xml。

下一步便是编写 dispatcher-servlet.xml 配置文件，首先定义 URL 的映射方式（HandlerMapping）。Spring 提供了几种常用的 HandlerMapping。

1．使用 SimpleUrlHandlerMapping

SimpleUrlHandlerMapping 提供了最简单的 URL 映射，通过 Properties 将 URL 和 Controller 对应起来，配置示例如下。

```
        <bean id="simpleUrlHandlerMapping" class="org.springframework.web.servlet.
handler.SimpleUrlHandlerMapping">
            <property name="mappings">
                <props>
                    <prop key="/a.html">controllerA</prop>
                    <prop key="/b.html">controllerB</prop>
                </props>
            </property>
        </bean>
```

这种方式和 Struts 的配置类似。当用户请求一个 URL 时，Spring 就在 SimpleUrl HandlerMapping 注入的 Properties 中查找对应的 Controller。

2．使用 BeanNameUrlHandlerMapping

BeanNameUrlHandlerMapping 的实现更为简单，每个 Controller 的 URL 与其 name 属性对应，因此，只需要对每个 Controller 以 URL 作为 name，就可以实现 URL 映射。配置示例如下。

```
        <bean id="beanNameUrlHandlerMapping" class="org.springframework.web.servlet.
handler.BeanNameHandlerMapping" />

        <bean name="/a.html" class="example.chapter7.ControllerA" />
        <bean name="/a.html" class="example.chapter7.ControllerB" />
```

之所以用 Bean 的 name 作为 URL 而不是 id，是因为 XML 规范不允许在 id 标识中使用特殊字符“/”。当用户请求一个 URL 时，Spring 将直接查找 name 为 URL 的 Controller。

使用 SimpleUrlHandlerMapping 的麻烦之处在于，添加或删除 Controller 时必须要对 SimpleUrlHandlerMapping 做相应的修改，而 BeanNameUrlHandlerMapping 则无需手工编写映射，只需要在每个 Controller 中仔细定义 name 属性。如果使用 XDoclet 自动生成配置文件，则可以将 name 在 Controller 的注释中定义，维护起来更加方便。因此，我们推荐首先考虑使用 BeanNameUrlHandlerMapping。事实上，如果没有在 XML 配置文件中定义任何 UrlHandlerMapping，则 Spring MVC 默认使用 BeanNameUrlHandlerMapping。

Spring 还提供了一个 ControllerClassNameHandlerMapping，它和 BeanNameUrlHandlerMapping 类似，不过是将 Controller 的 ClassName 和对应的 URL 关联起来，由于这种方式灵活性欠佳，实际使用较少。

也可以混合使用多种 UrlHandlerMapping，但是必须为每个 UrlHandlerMapping 指定 order 属性来表示优先级，order 值越小优先级越高，Spring 会先查询优先级高的 UrlHandlerMapping。若找到了对应的 Controller，就不再继续查询，否则，按照优先级依次查询，直到找到为止。若所有的 UrlHandlerMapping 都无法返回一个合适的 Controller，并且没有设置默认的 Controller 时，就会返回给客户端一个“404 Not Found”错误，表示不存在这个 URL。

下一步需要为 Spring MVC 指定一个 ViewResolver（视图解析器），指示使用何种视图技术，以及如何解析 ModelAndView 返回的逻辑视图名称。

这里我们直接给出使用 JSP 视图的配置，对于其他的视图技术将会在后面讲到。

```
        <bean id="viewResolver" class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
            <property name="viewClass" value="org.springframework.web.servlet.
view.JstlView" />
            <property name="prefix" value="/" />
```

```
        <property name="suffix" value=".jsp" />
    </bean>
```

prefix 和 suffix 将和逻辑视图名称一起组合成为实际视图的路径。例如，对于上例，若返回一个 new ModelAndView("test", model)，则实际的视图路径由 prefix+逻辑视图名+suffix 这 3 部分构成。

```
/test.jsp
```

定义前缀（prefix）使得视图文件无论放在何处都可以通过修改前缀来实现位置无关性（当然，必须在 web 目录内），许多应用程序将其放在/WEB-INF 目录下，使得用户无法通过 URL 直接访问视图文件以保证视图文件的安全；定义后缀（suffix）可以在将来用另一种视图技术（例如，Velocity）取代现在的 JSP 视图，只需将后缀从“.jsp”更改为“.vm”即可，而不必更改源代码中的逻辑视图名。总之，一切目标都是为了实现最大限度的解耦。

7.3.2 实现Controller

实现了 org.springframework.web.servlet.mvc.Controller 接口的 Bean 都可以作为有效的 Controller 来处理用户请求。例如，一个最简单的 TestController。

```
public class TestController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
Response response) throws Exception {
        String name = request.getParameter("name");
        if(name==null)
            name = "spring";
        Map model = new HashMap();
        model.put("name", name);
        model.put("time", new Date());
        return new ModelAndView("test", model);
    }
}
```

注意，上例的 Controller 接口和返回值 ModelAndView 都是在 Spring 框架中定义的，这和 SimpleMVC 项目中我们自己定义的 Controller 接口和 ModelAndView 类所在的包是不同的。在 SimpleMVC 项目中，我们并没有使用 Spring MVC，而是借用 Spring MVC 的概念自定义接口。在现在的 SpringMVC 项目中，我们没有自定义任何接口，而是直接使用 Spring MVC 框架来实现 Web 应用程序，这一点请读者务必区分清楚。

另外需要注意的是，ModelAndView 返回的逻辑视图是“test”，还记得在 viewResolver

中定义的 prefix 和 suffix 吗？实际的视图名称由这 3 部分构成便是“/test.jsp”。

最后将这个 TestController 作为 Bean 定义在 dispatcher-servlet.xml 中，由于我们准备使用默认的 BeanNameUrlHandlerMapping，因此，需要在 Bean 的 name 中指定 URL。

```
<bean name="/test.html" class="example.chapter7.TestController" />
```

完整的 dispatcher-servlet.xml 配置文件的内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="/test.html" class="example.chapter7.TestController" />

    <bean id="viewResolver" class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.
view.JstlView" />
        <property name="prefix" value="/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

注意，我们没有指定 UrlHandlerMapping，Spring 会自动使用默认的 BeanNameUrl HandlerMapping。

7.3.3 实现View

到目前为止，我们已经编写了 Controller 的实现和配置文件，最后一步是编写一个 JSP 文件作为视图。由于采用了 MVC 架构，视图的任务只有一个，就是将 Controller 返回的 Model 渲染出来。Spring MVC 会将 Model 中的所有数据全部绑定到 HttpServlet Request 中，然后将其转发给 JSP，JSP 只需将数据显示出来即可。

通过 JSTL 标签库能进一步简化显示逻辑，我们看看如何显示 TestController 返回的 Model。test.jsp 文件的内容如下。

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
    <head>
        <title>SpringMVC</title>
```

```
    </head>
    <body>
        <h3>Hello, <c:out value="${name}" />,
        it is <c:out value="${time}" /></h3>
    </body>
</html>
```

现在，Spring MVC 所需的所有组件都已编写并配置完毕，我们先来回顾一下 Spring MVC 的处理流程，如图 7-22 所示。

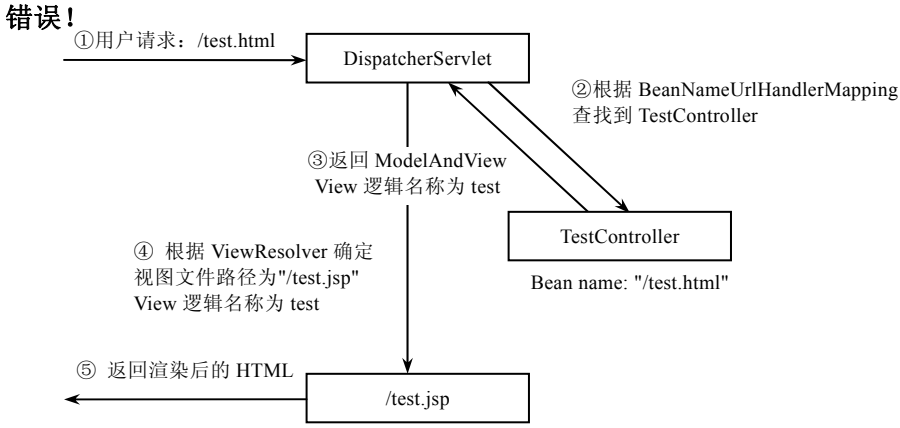


图 7-22

将 SpringMVC 工程的编译输出目录设置为/web/WEB-INF/classes，然后编译工程，打开浏览器，测试我们编写的 SpringMVC，结果如图 7-23 所示。

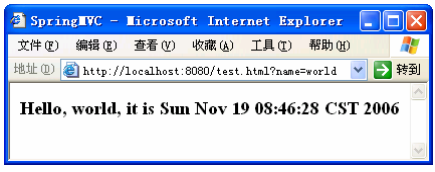


图 7-23

现在，读者对 Spring 的 MVC 框架应该有了一个初步的认识。实际上，除了基本的 MVC 标准流程外，Spring MVC 还提供了相当多的功能，下面我们将逐一介绍 Spring MVC 提供的各种丰富的 Controller、拦截器和异常处理机制。

7.4 Spring MVC提供的更多功能

除了直接实现 Controller 接口外，Spring 还提供了许多功能更多的 Controller 的实现，

可以选择继承一个合适类型的 Controller 来简化编码。相对于 Struts 或 WebWork，Spring 提供的 Controller 层次极为丰富，如图 7-24 所示。

错误！

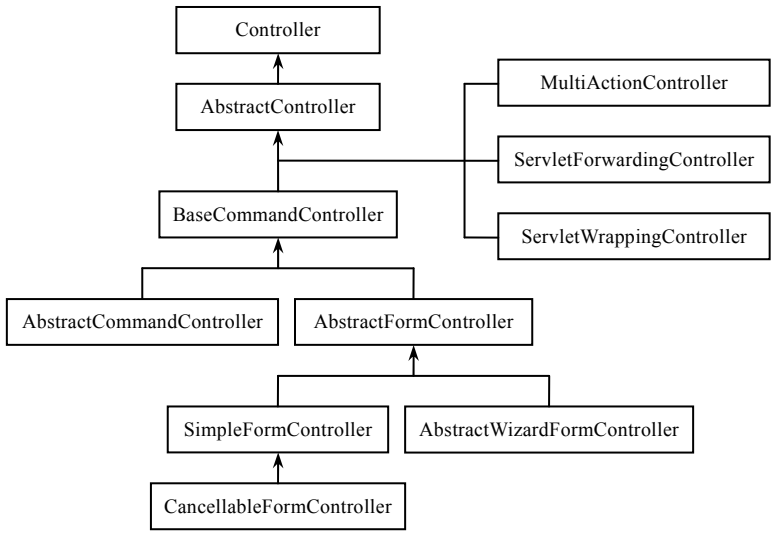


图 7-24

AbstractController 提供了一个最顶层的 Controller 模版，用来完成一些基本的任务。AbstractController 可以注入以下属性。

（1）supportedMethods：设定允许的 HTTP 请求方式，默认为 GET 和 POST。如果需要 PUT、HEAD 之类的请求，则需要覆盖默认的设定，不过，通常不需要设定其他 HTTP 请求方式。

（2）requireSession：设定是否需要 Session 支持，默认为 false。如果设定为 true，则要求当前请求必须和 Session 关联，这样可以保证子类在任何时候调用 request.getSession() 时不会得到 null。

（3）cacheSeconds：设定 HTTP 响应头的缓存，默认值为–1，表示不添加任何缓存指令到 HTTP 响应头；如果设为 0，表示完全不缓存；如果设为大于 0，表示应当缓存的秒数。

（4）synchronizeOnSession：表示来自同一用户的请求是否能并行处理，默认值为 false，表示允许同一用户同时向服务器发送多个请求。如果设定为 true，则同一用户的请求只能被依次处理，这个设置可以有效控制同一用户对服务器的并发请求，例如，禁止使用多线程下载由 Controller 生成的文件。

由于 AbstractController 位于 Controller 继承体系的上端，其他子类也可以非常方便地设定上述属性。

为了演示如何使用 Spring 内置的常用的 Controller，我们在 SpringMVC 工程的基础上扩展。在 Eclipse 中新建 SpringMVC_Controllers 工程，结构如图 7-25 所示。

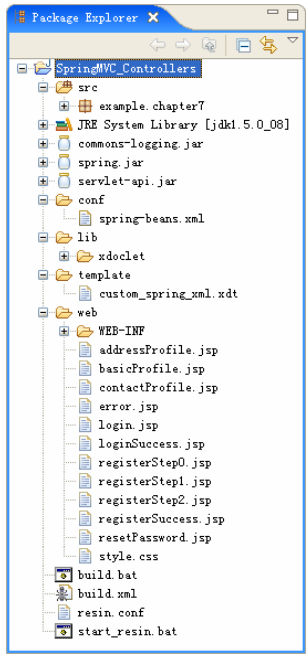


图 7-25

Spring 提供了一套标签库，能大大简化表单的绑定和验证任务，为了使用 Spring 内置的 Tag 和 JSP 标准标签库，需要将 c.tld 和 spring-form.tld 复制到/web/WEB-INF/目录下，并且在 web.xml 中声明。

```
<taglib>
    <taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>http://www.springframework.org/tags/form</taglib-uri>
    <taglib-location>/WEB-INF/spring-form.tld</taglib-location>
</taglib>
```

前面讲到了使用 BeanNameUrlHandlerMapping 能极大地简化从 URL 到 Controller 的映射，在实际的项目中，完全可以采用 Ant+XDoclet 自动生成 Spring Web 应用程序所需的 XML 配置文件，这样，对 URL 映射的配置就变成了在相应的 Controller 类的源代码中简单地添加一个 XDoclet 注释，极大地降低了手动维护 XML 配置文件带来的成本。

在第 3 章中我们已经介绍了如何使用 XDoclet 生成 Spring 的 XML 配置文件，并对

XDoclet 做了一定的扩展，使其支持 Spring 2.0 的 XML 配置文件。对于这个 SpringMVC_Controllers Web 应用程序也同样适用，先将 XDoclet 和 Ant 的相关文件复制到工程的 lib 目录下，然后编写 Ant 的 build.xml 脚本。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SpringMVC_Controllers" default="gen-spring-conf" basedir=".">
    <property name="src.dir"     value="src" />
    <property name="conf.dir"    value="conf" />
    <property name="web.dir"     value="web" />
    <property name="xdoclet.dir"  value="xdoclet" />
    <property name="template.dir" value="template" />
    <property name="lib.dir"     value="${web.dir}/WEB-INF/lib" />
    <property name="build.dir"    value="${web.dir}/WEB-INF/classes" />

    <!-- 定义编译期的 classpath -->
    <path id="master-classpath">
        <!-- 包含${lib.dir} -->
        <fileset dir="${lib.dir}">
            <include name="**/*.jar" />
        </fileset>
        <!-- 包含${xdoclet.dir} -->
        <fileset dir="${xdoclet.dir}">
            <include name="**/*.jar" />
        </fileset>
        <!-- 包含${src.dir} -->
        <pathelement path="${src.dir}"/>
    </path>

    <!-- 清理自动生成的资源 -->
    <target name="clean">
        <delete>
            <fileset dir="${build.dir}" />
            <filename>${web.dir}/WEB-INF/dispatcher-servlet.xml</filename>
        </delete>
    </target>

    <!-- 编译源代码 -->
    <target name="compile">
        <mkdir dir="${build.dir}"/>
       <javac destdir="${build.dir}" target="1.5" debug="on" debuglevel="lines">
            <classpath refid="master-classpath"/>
            <src path="${src.dir}"/>
        </javac>
    </target>
```

```
    <!-- 生成 Spring 配置文件 -->
    <target name="gen-spring-conf" depends="compile">
        <!-- 定义 Ant 任务 -->
        <taskdef name="springdoclet"
            classname="xdoclet.modules.spring.SpringDocletTask"
            classpathref="master-classpath"
        />
        <!-- 生成配置文件 -->
        <springdoclet
            destDir="${web.dir}/WEB-INF"
            mergeDir="${conf.dir}"
            force="true"
            excludedtags="@version,@author,@todo"
        >
            <fileset dir="${src.dir}" includes="**/*.java" />
            <springxml
                xmlencoding="UTF-8"
                templateFile="${template.dir}/custom_spring_xml.xdt"
                destinationFile="dispatcher-servlet.xml"
            />
        </springdoclet>
    </target>
</project>
```

不熟悉 XDoclet 的读者请参考第 3 章关于如何使用 XDoclet 自动生成 Spring 配置文件的相关章节，这里不再做更多的介绍。下面我们要讨论的是 Spring 提供的几种非常有用的控制器。

7.4.1 SimpleFormController

SimpleFormController 可以处理简单的表单，这和 Struts 的 ActionForm 类似，但是 Spring 对表单类不要求实现某个特定的接口。此外，SimpleFormController 可以同时完成显示表单和提交表单两个功能。显示表单无须编写代码，Spring 会自动处理，我们只需处理提交表单。

SimpleFormController 主要通过以下几个属性来决定如何显示和提交表单。

（1）commandClass：表单类（或称为命令类），Spring 据此来实例化一个表单类，请读者注意，在 Spring 中，表单对象被称为 Command 对象，这和 Struts 中的 ActionForm 类似，但 Spring 的 Command 对象不要求实现任何特定接口。

（2）formView：显示表单的视图名称。

（3）successView：提交表单成功后的视图名称。

以用户登录为例，我们设计一个 LoginController，从 SimpleFormController 派生。当用户以 GET 方式请求/login.do 时，LoginController 就会向用户显示一个登录表单，该视图的名称就是由 formView 指定的“login.jsp”，显示表单不需要编写任何代码，Spring 替我们自动完成这一步骤，唯一需要处理的是覆写 onSubmit 方法，处理表单提交。

```
/**
 * 处理用户登录表单
 * @spring.bean name="/login.do"
 * @spring.property name="commandClass" value="example.chapter7.User"
 * @spring.property name="formView" value="login"
 * @spring.property name="successView" value="loginSuccess"
 */
public class LoginController extends SimpleFormController {
    private UserService userService;
    /**
     * @spring.property ref="userService"
     */
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @Override
    protected ModelAndView onSubmit(HttpServletRequest request, HttpServlet
Response response, Object command, BindException errors) throws Exception {
        User user = (User)command;
        try {
            userService.login(user.getUsername(), user.getPassword());
            // 登录成功，在 Session 中标记:
          request.getSession().setAttribute("USERNAME", user.getUsername());
            // 然后返回 successView:
            Map model = new HashMap();
            model.put("username", user.getUsername());
            return new ModelAndView(getSuccessView(), model);
        }
        catch(RuntimeException e) {
            // 登录失败,返回 formView 让用户重新填写表单:
            Map model = new HashMap();
            model.put("command", command);
            model.put("error", e.getMessage());
            return new ModelAndView(getFormView(), model);
        }
    }
}
```

由于使用 XDoclet 在注释中就配置好了 LoginController，因此运行 Ant，生成的 XML 配置片断最终如下。

```
<bean name="/login.do" class="example.chapter7.LoginController">
    <property name="userService" ref="userService" />
    <property name="commandClass" value="example.chapter7.User" />
    <property name="formView" value="login" />
    <property name="successView" value="loginSuccess" />
</bean>
```

对于作为 formView 的 login.jsp 视图文件，为了让 Spring 自动绑定表单的内容，使用 Spring 的<form:>标签库即可完成此功能。

```
<form:form method="post" action="login.do">
    用户名：<form:input path="username" />
    口令：<form:password path="password" />
    <input type="submit" name="Submit" value="登录" />
</form:form>
```

<form:form>用于定义一个表单，和 HTML 的<form>标签对应，<form:input>对应一个 INPUT，其 path 属性指定了这个文本输入框对应 Command 对象的 username 属性，<form:password>对应一个类型为 PASSWORD 的 INPUT，其 path 属性指定了这个口令输入框对应 Command 对象的 password 属性。

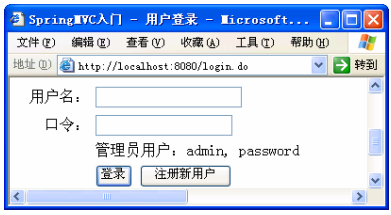


图 7-26

运行 Ant 编译，然后启动 Resin，输入 http://localhost:8080/login.do，就可以看到 Spring 自动绑定表单的效果，如图 7-26 所示。

可以看到，整个处理流程非常清晰。commandClass 必须是具有默认构造方法，可以被实例化的类，通常都是简单的 JavaBean 对象，这里我们使用的 commandClass 是 User 类，但是仅使用了 username 和 password 这两个属性，其余属性由于不出现在登录表单中，将直接被 Spring 忽略。

7.4.2 验证表单

对于用户输入的表单，通常需要在服务器端进行验证，以确保数据的完整性和一致性。Spring 提供了一个 Validation 框架来验证用户输入的表单，并可以将错误信息绑定到合适的字段上。

以登录表单为例，我们可以编写一个 LoginValidator 来验证登录表单。

```
public class LoginValidator implements Validator {
    public boolean supports(Class clazz) {
        return clazz==User.class;
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "username", "error.
username.required", "必须填写用户名");
        ValidationUtils.rejectIfEmpty(errors, "password", "error.password.
required", "必须填写口令");
    }
}
```

所有的验证器都必须实现 Validator 接口，该接口需要实现两个方法。

（1）boolean supports(Class clazz)：返回是否支持该类型。

（2）void validate(Object target, Errors errors)：验证表单对象。

Spring 提供了一个 ValidationUtils 来简化验证，如果某个字段未通过验证，就将其放入 Errors 对象中，也可以直接使用 Errors 的 rejectValue()方法来加入一个验证错误。

有了验证器后，就可以将其注入到 BaseCommandController 及其子类中。LoginController 也是从 BaseCommandController 继承而来的，一旦注入了 Validator，Spring 对表单的处理流程就如图 7-27 所示。

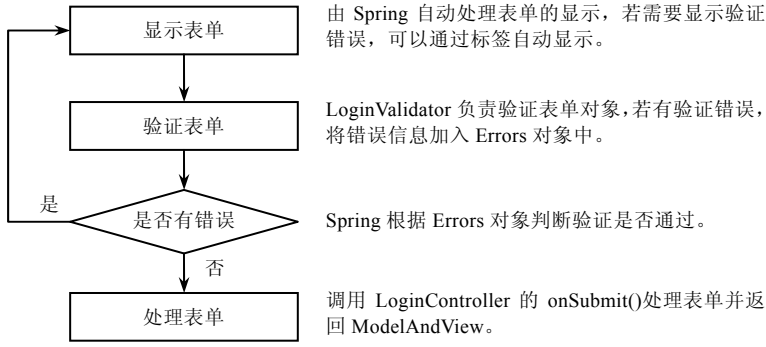


图 7-27

加入了验证功能的 login.jsp 由<form:errors>标签显示验证错误，其内容如下。

```
<form:form method="post" action="login.do">
    用户名：<form:input path="username" /> <form:errors path="username" />
    口令：<form:password path="password" /> <form:errors path="password" />
    <input type="submit" name="Submit" value="登录" />
</form:form>
```

图 7-28

其中，path 属性必须和 ValidationUtils.rejectIfEmpty（Errors errors, String field, String errorCode）或 Errors.rejectValue（String field, String errorCode, String defaultMessage）中的 field 参数一致，这样，Spring 的<form:errors>标签就可以自动绑定错误信息。

对于用户登录表单，如果没有输入用户名或口令，就会显示验证错误的信息，如图 7-28 所示。

7.4.3 MultiActionController

如果需要处理多个类似的请求，可以考虑使用一个 MultiActionController 来实现，而不必分别编写多个单一功能的 Controller。例如，在显示用户详细资料时，考虑到用户资料的内容较多，可以分为“基本资料”、“联系方式”和“详细地址”3 大类显示给用户，由于这 3 个页面都从 User 对象中获取数据，因此，将 3 个页面的显示放到一个 MultiActionController 中不仅更方便，也便于减少 Controller 的数量。

ViewProfileController 便是从 MultiActionController 派生的，能够查看 3 个页面。

```
/**
 * @spring.bean name="/*Profile.do"
 */
public class ViewProfileController extends MultiActionController {
public class ViewProfileController extends MultiActionController {
    private UserService userService;

    /**
     * @spring.property ref="userService"
     */
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    private String getUsername(HttpServletRequest request) {
       String username = (String) request.getSession().getAttribute("USERNAME");
        if(username==null)
            throw new NeedLoginException();
        return username;
    }

    public ModelAndView basicProfile(HttpServletRequest request, HttpServlet
Response response) throws Exception {
```

```
            String username = getUsername(request);
            User user = userService.query(username);
            Map model = new HashMap();
            model.put("username", user.getUsername());
            model.put("role", user.getRole()==User.ADMIN ? "Admin" : "User");
            return new ModelAndView("basicProfile", model);
        }

        public ModelAndView contactProfile(HttpServletRequest request, HttpServlet
Response response) throws Exception {
            String username = getUsername(request);
            User user = userService.query(username);
            Map model = new HashMap();
            model.put("email", user.getEmail());
            model.put("blog", user.getBlog());
            model.put("website", user.getWebsite());
            return new ModelAndView("contactProfile", model);
        }

        public ModelAndView addressProfile(HttpServletRequest request, HttpServlet
Response response) throws Exception {
            String username = getUsername(request);
            User user = userService.query(username);
            Map model = new HashMap();
            model.put("province", user.getProvince());
            model.put("city", user.getCity());
            model.put("zip", user.getZip());
            return new ModelAndView("addressProfile", model);
        }
    }
```

其中，basicProfile()、contactProfile()和 addressProfile()这 3 个方法都能分别独立地处理用户请求，并返回 User 对象中对应的部分数据，那么，Spring 如何根据 URL 来确定应该调用 ViewProfileController 的哪个方法来处理用户请求呢？答案是使用 methodName Resolver 的设定。在使用 ViewProfileController 之前，我们还需要配置一个 MethodName Resolver。

MultiActionController 默认使用 InternalPathMethodNameResolver，从 URL 中提取方法名，然后调用相应的方法来处理请求。对于上面的 ViewProfileController，传入“http://localhost:8080/basicProfile.do”、“http:// localhost:8080/contactProfile.do”和“http:// localhost:8080/addressProfile.do ”就可以分别调用 basicProfile()、contactProfile()和 addressProfile()方法来处理用户请求。

图 7-29

由于要从 URL 中提取 methodName，所以配置 ViewProfileController 的 name 为“/*Profile.do”，使用通配符“*”来匹配这 3 个方法名，因此，方法命名一定要符合一定的规则，才便于使用 URL 映射。

现在，我们在一个 ViewProfileController 中就同时实现了 3 个页面的处理。启动服务器，登录后可以看到 ViewProfileController 的效果如图 7-29~图 7-31 所示。

图 7-30

图 7-31

如果不喜欢通过 URL 来调用方法，Spring 同样提供了多种 MethodNameResolver 的实现，最常见的一种是 ParameterMethodNameResolver，它可以从参数中提取方法名。

```
        <bean id="methodNameResolver" class="org.springframework.web.servlet.mvc.
multiaction.ParameterMethodNameResolver">
            <property name="paramName" value="action" />
            <property name="defaultMethodName" value="basicProfile" />
        </bean>
```

在上面的 XML 配置片断中，ParameterMethodNameResolver 根据 URL 的 action 参数来确定方法名称。将 methodNameResolver 注入到 ViewProfileController 后，若用户请求“/viewProfile.do?action=basicProfile”，则 methodNameResolver 将根据参数 action=basicProfile 来决定调用 ViewProfileController 的 basicProfile()方法处理用户请求，若用户请求“/viewProfile.do?action=contactProfile”，则调用 contactProfile()方法，若没有找到 action 参数，则 methodNameResolver 根据 defaultMethodName 属性来调用 basicProfile()方法。

事实上，我们自己也可以手动编写代码来确定调用哪个方法。

```
        public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
Response response) throws Exception {
            String methodName = request.getParameter("action");
            if("basicProfile".equals(methodName))
```

```
        return basicProfile(request, response);
    if("contactProfile".equals(methodName))
        return contactProfile(request, response);
    if("addressProfile".equals(methodName))
        return addressProfile(request, response);
    // default method:
    return basicProfile(request, response);
}
```

Spring 为我们所做的工作不过是将上述代码以配置文件的形式更灵活地实现而已。除了 ParameterMethodNameResolver 外，Spring 还提供了 PropertiesMethodNameResolver，不过，最简单也最有用的还是 ParameterMethodNameResolver。

7.4.4 AbstractWizardFormController

从名字上就可以看出，AbstractWizardFormController 能够实现向导式的页面。如果用户需要填写的表单内容很多，就有必要将其拆为几个页面，使用户能通过“上一步”和“下一步”按钮方便地在向导页面间导航，例如，设计一个在线调查的向导，就可以方便地引导用户一步一步完成调查表单的填写。

我们以注册新用户为例，RegisterController 需要用户填写基本资料、联系方式和详细地址，由于表单内容较多，我们让用户分 3 个页面分步完成注册。

我们无须处理“下一步”和“上一步”按钮，Spring 会自动显示正确的页面，我们只需要处理最后用户单击“完成”按钮提交的整个表单对象。

```
/**
 * 用户注册向导
 * @spring.bean name="/register.do"
 * @spring.property name="commandClass" value="example.chapter7.User"
* @spring.property name="pages" list="registerStep0,registerStep1, registerStep2"
 */
public class RegisterController extends AbstractWizardFormController {
    private UserService userService;
    /**
     * @spring.property ref="userService"
     */
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    // 当用户单击"_finish"按钮时,触发 processFinish()方法:
    protected ModelAndView processFinish(HttpServletRequest request,
```

```
            HttpServletResponse response, Object command, BindException errors)
            throws Exception {
        User user = (User)command;
        userService.register(user);
        Map model = new HashMap();
        model.put("username", user.getUsername());
        return new ModelAndView("registerSuccess", model);
    }
}
```

那么，Spring 是如何知道下一个或上一个需要显示的页面呢？除了指定 command Class 为 User 对象外，我们还需要将几个 View 的逻辑名称注入到 RegisterController 的 pages 属性中，注意到 AbstractWizardController 的 pages 是从下标 0 开始计数的，因此，我们将注册的 3 个页面依次命名为 registerStep0.jsp、registerStep1.jsp 和 registerStep2.jsp。

除了指定 pages 属性外，我们还需要按照一定的规则来编写 JSP 页面，才能告诉 Spring 如何显示下一页或上一页。在表单的提交按钮上，必须以_target+索引命名按钮，例如：

<input type="submit" name="_target1" value="下一步">将前进到索引为 1 的页面，即 registerStep1.jsp。

<input type="submit" name="_target0" value="上一步">将返回到索引为 0 的页面，即 registerStep0.jsp。

最后一个“完成”按钮必须以“_finish”命名。

```
<input type="submit" name="_finish" value="完成">
```

当用户单击“完成”按钮后，Spring 将调用 processFinish()方法处理表单。

如果需要验证表单，在 AbstractWizardController 中，就无法使用 Validator 来进行验证，因为用户在每个页面仅填写了部分内容，直到用户单击“完成”按钮时，整个表单对象才被填充完毕，因此，在任何一个页面中验证 Command 都将失败，为此，验证必须在 AbstractWizardController 的 validatePage()方法中进行，Spring 将传入 page 参数，我们就根据这个参数对 command 对象进行部分验证。

```
// 每当用户单击"_target?"准备前进到下一步时,触发 validatePage()来验证当前页:
protected void validatePage(Object command, Errors errors, int page) {
    User user = (User)command;
    if(page==0) {
        // 验证 username,password:
        if(!user.getUsername().matches("[a-zA-Z0-9]{3,20}"))
            errors.rejectValue("username", "error.username", "用户名不符合要求");
        if(userService.isExist(user.getUsername()))
```

```
                errors.rejectValue("username", "error.username", "用户名已存在");
            if(user.getPassword()==null || user.getPassword().length()<6)
              errors.rejectValue("password", "error.password", "口令至少为 6 个字符");
        }
        else if(page==1) {
            // 验证 email,blog,website:
            if(user.getEmail()==null)
               errors.rejectValue("email", "error.email.empty", "电子邮件不能为空");
            else if(!user.getEmail().matches("[a-zA-Z0-9\\_\\-]+\\@[a-zA-Z0-9\\_
\\-]+[\\.[a-zA-Z0-9\\_\\-]+]{1,2}"))
                errors.rejectValue("email", "error.email", "电子邮件地址无效");
            if(user.getBlog()==null || user.getBlog().trim().equals(""))
                errors.rejectValue("blog", "error.blog", "博客地址不能为空");
            if(user.getWebsite()==null || user.getWebsite().trim().equals(""))
                errors.rejectValue("website", "error.website", "网址地址不能为空");
        }
        else if(page==2) {
            // 验证 province,city,zip:略过
        }
    }
```

若验证未通过，则将停留在当前页，并可以通过<form:errors>显示相应的错误信息，待用户更正后，才可以继续前进到下一页。编译工程，启动 Resin 服务器，可以看到整个用户注册的流程如图 7-32~图 7-35 所示。

图 7-32

图 7-33

图 7-34

图 7-35

读者也许注意到了，第一个页面有两个口令框，其中，第二个口令框名称为password2，在 User 对象中并没有对应的属性，Spring 不会自动绑定它。那么，如何验证用户两次输入的口令是否一致呢？我们一般不愿意更改 User 对象，因为 User 对象很可能对应数据库中的某个表，而数据库表不会存储同一用户的两份相同的口令。此时，可以通过 JavaScript 来验证，既方便，又能避免修改 User 对象。因此，在 Web 应用程序的设计中，不要仅仅拘泥于 JavaEE 框架，对于 JavaScript、AJAX 等技术也要充分利用。

为了让读者能更方便地看到 Controller 和对应的 JSP 视图，我们将 Controller 的源码也放入到每个页面中。在每一个页面中，读者都可以通过单击“查看源码”非常方便地阅读对应的 Controller 代码，如图 7-36 所示。

图 7-36

7.4.5 输出二进制内容

虽然大多数时候用户请求的都是 HTML 页面，不过，某些情况下仍然需要向用户发送动态生成的二进制内容，例如，图片认证码、Excel 报表等，从 HTTP 协议上看来，向用户发送二进制内容只需要设置好输出响应的 MIME 类型，然后直接写入二进制流即可。对应到 JavaEE Web 应用程序，就是设置 HttpServletResponse 对象的 ContentType，然后将二进制数据写入 OutputStream 即可。

我们以动态生成一个图片认证码为例，实现一个 ImageController 如下。

```
public class ImageController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        // 在内存中绘图:
        String code = String.valueOf((int)(Math.random() * 9000) + 1000);
```

```
        BufferedImage image = new BufferedImage(100, 50, BufferedImage. TYPE_INT_RGB);
            Graphics2D g = image.createGraphics();
            g.setColor(Color.GRAY);
            g.fillRect(0, 0, 100, 50);
            g.setColor(Color.RED);
            g.drawRect(0, 0, 99, 49);
            g.setColor(Color.BLACK);
            g.drawString(code, 20, 20);
            g.dispose();
            image.flush();
            // 设置 ContentType:
            response.setContentType("image/jpeg");
            response.setHeader("Cache-Control", "no-cache");
            // 输出到 ServletOutputStream:
            ServletOutputStream output = response.getOutputStream();
            ImageIO.write(image, "jpeg", output);
            return null;
        }
    }
```

上述代码的运行效果如图 7-37 所示。

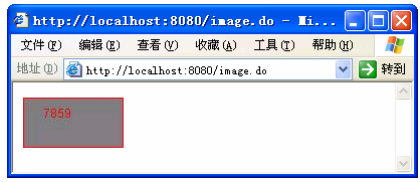


图 7-37

从上面的代码可以看出，输出二进制内容的关键是设置正确的 ContentType 及 ContentLength，然后获得输出流，将二进制内容写入即可。常见的 ContentType 类型如表 7-2 所示。

表 7-2

文件类型	ContentType
JPEG	image/jpeg
MS Word	application/msword
MS Excel	application/vnd.ms-excel
PDF	application/pdf
MP3	audio/mpeg
未知文件类型	application/octet-stream

虽然可以将所有的二进制内容都设置为 application/octet-stream，但是浏览器会根据 ContentType 做不同的处理。例如，若一个 Word 文档被设置为 application/octet-stream，则浏览器将直接提示将文件另存到本地，而设置为 application/msword 时，如果用户计算机上已装有 Word，就可以直接在浏览器中打开该 Word 文档。

在 Spring 中，生成二进制内容还可以用 AbstractView 来实现，即 Controller 仍返回一个 ModelAndView，最终输出由 View 实现，Spring 提供了好几种这样的 View，例如，AbstractPdfView 可以输出 PDF 文档，AbstractExcelView 可以输出 Excel 文档。使用 AbstractView 来输出二进制内容的关键是正确实现下面几个方法。

（1）String getContentType()：返回二进制内容的 ContentType。

（2）void renderMergedOutputModel (Map model, HttpServletRequest request, Http ServletResponse response)：输出二进制内容，所需的数据可以从 model 中获得，最终的输出仍然是通过调用 response.getOutputStream()获得输出流并写入。

使用 Spring 提供的 AbstractView 虽然也能实现输出二进制内容，并且能保持 Spring MVC 的一致性，不过，我个人认为，这样做反而增加了应用程序的复杂性。由于通常 Web 应用程序需要动态生成二进制内容的地方不多，完全可以自己在 Controller 中将二进制内容写入 response 对象，然后直接返回 null 结束处理，这样反而使流程更清晰，配置更少，代码更容易维护。有些时候，Spring 过于严密的封装反而增加了代码的复杂性。当然，如果需要输出 Excel 或 PDF 文档，则仍可以选择 AbstractExcelView 和 AbstractPdfView，因为它们提供了一些额外的辅助方法来简化文档的生成。

7.4.6 重定向URL

重定向 URL 会使服务器向客户端发送一个 Redirect 响应，并包含一个目标 URL。客户端接收到 Redirect 响应后，会立刻重新请求新的 URL，这一点和 Forward 不同。前者使客户端发送了两次独立的 HTTP 请求，而后者请求是在服务器内部处理的，客户端并不知道服务器端对 Request 是否做了 Forward 处理。

重定向功能的主要用途是为了在服务器端修改了某一资源的 URL 后，原有客户仍可以继续通过原来的 URL 访问该资源。由于重定向会使客户端发送两次请求，所以降低了网络效率，并且不便于用户在浏览器中单击“后退”按钮返回上一个页面。对于 Web 应用程序而言，决不能大量使用重定向功能。

在 Controller 中实现 Redirect 也非常容易。最简单的方法是直接调用 HttpServlet Response 对象的 sendRedirect()方法，然后返回 null。一旦返回的 ModelAndView 为 null，Spring 就认为 Controller 自己已经完成了请求处理，不再按照常规的 MVC 流程继续处理

请求。

例如，对于用户注销登录的操作，在清理了 Session 的内容后，就可以将用户重定向到登录页面。LogoutController 代码如下。

```
/**
 * @spring.bean name="/logout.do"
 */
public class LogoutController extends AbstractController {
   protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response) throws Exception {
      request.getSession().removeAttribute("USERNAME");
      response.sendRedirect("login.do");
      return null;
   }
}
```

另一种实现重定向的方法不用直接调用 HttpServletResponse 对象的 sendRedirect()方法，而是返回一个带有“redirect:”前缀的 View，这样，ViewResolver 就知道这是一个重定向操作，于是不再渲染视图，而是直接向客户端发送 Redirect 响应。

```
return new ModelAndView("redirect:login.do");
```

Spring 还提供了一个 RedirectView 对象，也可以实现重定向操作，不过使用 RedirectView 使 Controller 和 View 的耦合稍微紧密了一点，推荐的方法是使用“redirect:”前缀。

使用重定向要注意的一点是，重定向的资源不可位于/WEB-INF/目录下，因为用户无法通过 URL 直接访问位于/WEB-INF/目录下的资源，而使用 MVC 流程通过 forward 调用/WEB-INF/目录下的资源是允许的。

7.4.7 处理异常

如果 Controller 在处理用户请求时发生了异常，自己捕获异常并跳转到出错页面会使核心逻辑混乱。Spring 的 MVC 框架提供了一个 HandlerExceptionResolver，为所有的 Controller 抛出的异常提供一个统一的入口。

```
/**
 * @spring.bean id="handlerExceptionResolver"
 */
public class MyHandlerExceptionResolver implements HandlerExceptionResolver {
   private Log log = LogFactory.getLog(getClass());
```

```
    public ModelAndView resolveException(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex) {
        log.warn("Handle exception: " + ex.getClass().getName());
        if(ex instanceof NeedLoginException)
            return new ModelAndView("redirect:login.do");
        Map model = new HashMap();
        model.put("ex", ex.getClass().getSimpleName());
        model.put("message", ex.getMessage());
        return new ModelAndView("error", model);
    }
}
```

MyHandlerExceptionResolver 根据 Exception 类型判断如何处理异常，如果是 NeedLoginException，说明系统要求用户登录，这时直接将用户导向到登录页面；对于其他类型的异常，则直接将异常的错误信息显示给用户，注意返回的视图名称为“error”，实际的视图文件即为“/error.jsp”。

使用 HandlerExceptionResolver 可以避免在应用程序的每一个 Controller 中都去处理异常，将异常统一放到 HandlerExceptionResolver 中可以极大地简化异常处理逻辑，也便于在一个统一的地方记录异常日志。对于无法处理的异常，可以给用户显示一个友好的出错页面。

7.4.8 拦截请求

在本章的前几节，我们已经看到了使用 Filter 可以拦截用户请求，并实现相应的处理。Spring 的 MVC 框架也提供了一个拦截器链，可以由多个 HandlerInterceptor 构成，允许在 Controller 处理用户请求的前后有机会处理请求。和 Filter 相比，HandlerInterceptor 是在 Spring 的 IoC 容器中配置的，可以注入任意的组件，而 Filter 定义在 Spring 容器之外，因此，注入 IoC 组件比较困难，或者难以得到一个优雅的设计。

HandlerInterceptor 接口定义了以下 3 个方法。

```
    boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler)
    void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView)
    void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex)
```

（1）preHandle()方法在 Controller 执行前调用，其返回值指定了是否应当继续处理请求。若返回 false，Spring MVC 框架将不再继续调用下一个拦截器，也不会将请求交给

Controller 处理，整个请求处理将到此结束。

（2）postHandler()方法在 Controller 执行完毕后调用，此时 Controller 仅返回了 ModelAndView 对象，还没有对视图进行渲染，在这个方法中有机会对 ModelAndView 进行修改。

（3）afterCompletion()方法在整个请求全部完成后调用，通过判断参数 ex 是否为 null 就可以判断是否产生了异常。

通过 HandlerInterceptor，就有机会在一个请求执行的 3 个阶段对其进行拦截。例如，为了统计 Web 应用程序的性能，我们设计了一个性能拦截器，将每个用户请求的处理时间记录下来。PerformanceHandlerInterceptor 实现如下。

```
        /**
         * @spring.bean id="performanceHandler"
         */
        public class PerformanceHandlerInterceptor implements HandlerInterceptor {
          private final Log log = LogFactory.getLog(PerformanceHandlerInterceptor. class);
           private static final String START_TIME = "PERF_START";

           public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
               request.setAttribute(START_TIME, System.currentTimeMillis());
               return true;
           }

           public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
               // 不需要处理 postHandler, 保留空方法即可
           }

           public void afterCompletion(HttpServletRequest request, HttpServlet
Response response, Object handler, Exception ex) throws Exception {
               Long startTime = (Long)request.getAttribute(START_TIME);
               if(startTime!=null) {
                   long last = System.currentTimeMillis() - startTime.longValue();
                   String uri = request.getRequestURI();
                   String query = request.getQueryString();
                   if(query!=null)
                       uri = uri + '?' + query;
                   log.info("URL: " + uri);
                   log.info("Execute: " + last + "ms.");
               }
           }
        }
```

由于我们必须保证 PerformanceHandlerInterceptor 是线程安全的，因此，绝不可将起始时间记录在 PerformanceHandlerInterceptord 的成员变量中。由于每个请求都对应一个独立的 HttpServletRequest 实例，因此，将起始时间放入 HttpServletRequest 实例中就保证了线程安全。

然后，将其添加到 handlerMapping 中的 interceptor 列表中。

```
        <bean id="handlerMapping" class="org.springframework.web.servlet.handler.
BeanNameUrlHandlerMapping">
            <property name="interceptors">
               <list>
                  <ref bean="performanceHandler" />
               </list>
            </property>
        </bean>
```

运行应用程序，在浏览器中请求/login.do，查看控制台输出如下。

```
[2006/11/20 22:33:41.857] URL: /login.do
[2006/11/20 22:33:41.857] Execute: 3781ms.
[2006/11/20 22:33:45.325] URL: /login.do
[2006/11/20 22:33:45.325] Execute: 0ms.
```

可以看到 PerformanceHandlerInterceptor 记录的处理时间。首次执行/login.do 请求时，耗时 3 秒多，这是因为服务器需要编译 JSP 文件，随后刷新页面，由于可以跳过 JSP 的编译步骤，/login.do 请求在 1ms 内就完成了。

7.4.9 处理文件上传

文件上传是 Web 应用程序中常见的功能。本质上，浏览器在向服务器发送文件时，其 HTTP 请求必须以 multipart/form-data 的形式发送，该规范定义在 RFC 2388（http://www.ietf.org/rfc/rfc2388.txt）中，可以实现一次上传一个或多个文件。不过，JavaEE 的 Web 规范并没有内置处理 multipart 请求的功能，因此，要实现文件上传，就必须借助于第三方组件，或者自己手动编码解析 HttpServletRequest。

Apache Commons FileUpload（http://jakarta.apache.org/commons/fileupload）组件和 COS FileUpload（http://www.servlets.com/cos）组件都是常见的处理文件上传的组件，Spring 很好地对这两种组件进行了封装。在 Spring 中处理文件上传时，根本无须与这两个组件的 API 打交道，只需用到 Spring 提供的 MultipartHttpServletRequest 对象，就可以轻松实现文件上传的功能。

下面的例子演示了如何在 Spring 中实现文件上传。我们在 Eclipse 中建立如下的 WebUpload 工程，如图 7-38 所示。

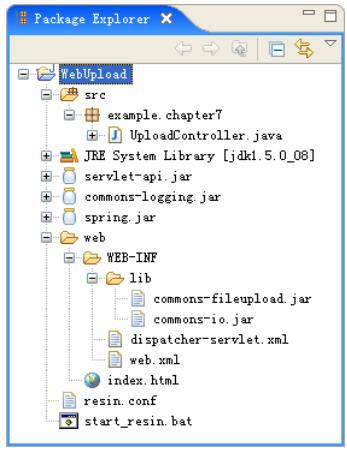


图 7-38

默认地，Spring 不会处理文件上传，即所有的以 multipart/form-data 形式发送的请求都不被处理，如果要处理 Multipart 请求，需要在 Spring 的 XML 配置文件中申明一个 MultipartResolver。

```
        <bean id="multipartResolver" class="org.springframework.web.multipart.
commons.CommonsMultipartResolver">
            <!-- 最大允许上传文件大小:1M -->
            <property name="maxUploadSize" value="1048576" />
        </bean>
```

maxUploadSize 属性指定了最大所能上传的文件大小，若超出了最大范围，Spring 将会直接抛出异常。

如果一个请求不是 Multipart 请求，它就会按照正常的流程处理；如果一个请求是 Multipart 请求，Spring 就会自动调用 MultipartResolver，然后将 HttpServletRequest 请求变为 MultipartHttpServletRequest 请求，开发者只需要处理 MultipartHttpServletRequest 对象就可以了。

如何得知一个请求是否是 MultipartHttpServletRequest 类型呢？通过 instanceof 操作就能非常简单地判断出来。我们在 UploadController 中实现文件上传的代码如下。

```
        public class UploadController implements Controller {
            private Log log = LogFactory.getLog(getClass());

            public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
                // 判断 request 是不是 multipart 请求:
                if(request instanceof MultipartHttpServletRequest) {
```

```
                    MultipartHttpServletRequest multipart = (MultipartHttpServlet
Request)request;
                    MultipartFile file = multipart.getFile("file");
                    if(file==null || file.isEmpty()) {
                        // 文件不存在:
                        response.sendError(HttpServletResponse.SC_BAD_REQUEST);
                        return null;
                    }
                    String filename = file.getOriginalFilename();
                    log.info("Upload file name: " + filename);
                    // 获取文件扩展名:
                    String ext = "";
                    int pos;
                    if((pos = filename.lastIndexOf('.'))!=(-1)) {
                      ext = URLEncoder.encode(filename.substring(pos).trim(), "UTF-8");
                    }
                    InputStream input = null;
                    OutputStream output = null;
                    // 确定服务器端写入文件的文件名:
                    String uploadFile = request.getSession()
                           .getServletContext()
                           .getRealPath("/upload" + System.currentTimeMillis() + ext);
                    try {
                        // 获得上传文件的输入流:
                        input = file.getInputStream();
                        // 写入到服务器的本地文件:
                     output = new BufferedOutputStream(new FileOutputStream(uploadFile));
                        byte[] buffer = new byte[1024];
                        int n;
                        while((n=input.read(buffer))!=(-1)) {
                            output.write(buffer, 0, n);
                        }
                    }
                    finally {
                        // 必须在 finally 中关闭输入/输出流:
                        if(input!=null) {
                            try {
                                input.close();
                            }
                            catch(IOException ioe) {}
                        }
                        if(output!=null) {
                            try {
                                output.close();
                            }
```

```
                    catch(IOException ioe) {}
                }
            }
            // 告诉浏览器文件上传成功:
            Writer writer = response.getWriter();
            writer.write("File uploaded successfully!");
            writer.flush();
        }
        else {
            // 非 multipart/form-data 请求,发送一个错误:
          response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }
        return null;
    }
}
```

仔细查看上面的代码，读者可能会发现，我们根本没有调用 Commons FileUpload 或 COS FileUpload 组件的 API，Spring 已经完全为我们封装好了。那么，Spring 如何确定使用 Commons FileUpload 还是使用 COS FileUpload 呢？答案是发现哪个就用哪个。如果在/WEB-INF/lib 目录下放置 Commons FileUpload 的 jar 包，Spring 就会自动使用 Commons FileUpload，COS FileUpload 也是如此，这样带来的好处是完全屏蔽了底层组件的 API，如果需要替换底层组件，只需要替换相应的 jar 包，甚至连 XML 配置文件都不用改动。

在 WebUpload 工程中，我们使用的是 Commons FileUpload，只需将 commons-fileupload.jar 和 commons-io.jar 放到/WEB-INF/lib 目录下，剩下的事情就由 Spring 处理了。使用任何文本编辑器编写一个最简单的上传文件的 index.html 页面。

```
<html><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Upload File</title>
</head>

<body>
<form action="upload.do" method="post" enctype="multipart/form-data"
name="form1">
<p>请选择需要上传的文件：<input type="file" name="file"></p>
<p><input type="submit" name="Submit" value="上传"></p>
</form>
</body>
</html>
```

配置好 DispatcherServlet 后，运行这个 Web 应用程序，打开 index.html，选择待上

传的文件，如图 7-39 所示。

文件上传成功后，就可以在服务器的 Web 应用的根目录下找到已上传的文件，如图 7-40 所示。

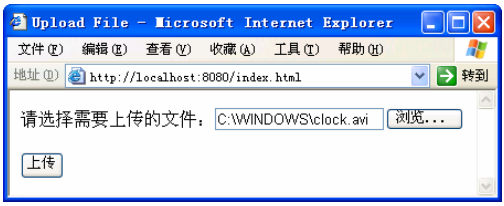


图 7-39

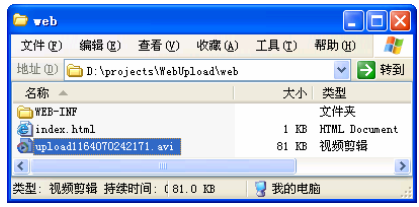


图 7-40

对于非 file 类型的表单字段，仍可以调用 MultipartHttpServletRequest 的 getParameter()方法获得相应的字段值，因为 MultipartHttpServletRequest 也实现了 HttpServletRequest 接口。

也可以在 SimpleFormController 中将表单中上传的文件绑定到 byte[]类型的属性中，不过，如果上传文件较大，则将消耗较大的服务器内存，因此，采用何种解决方案需要视情况而定。

7.5 使用其他视图技术

Spring 的 MVC 框架被设计为控制器和具体视图技术的完全解耦。控制器只需要返回视图的逻辑名称，至于如何解析这个名称，由 ViewResolver 负责，因此，从一种视图技术(例如，JSP)转到另一种视图技术时，控制器无需更改，只要设定合适的 ViewResolver 即可。

下面要介绍的是除了 JSP 之外的其他视图技术。Spring 提供了对 Velocity、Freemaker 和 XSLT 的完善支持。

7.5.1 Velocity

Velocity 是一种模版技术，能够通过模版生成任何文本内容，因此也非常适合作为视图。Velocity 提供了一种非常简单的模版语言 VTL，很容易同时被 Java 开发人员和网页设计人员轻松理解，和 JSP 相比，Velocity 在以下几点更具优势。

（1）Velocity 不提供 Java 代码支持，这意味着它只能作为视图，无法嵌入任何逻辑代码。Velocity 的这种设计就是为了强制分离 Java 逻辑和 Web 页面，使 Web 应用程序更容易维护，而 JSP 允许嵌入任意的 Java 代码，很容易造成 Java 代码的滥用。

（2）Velocity 不需要特定的标签，它使用简单的${var_name}来标记变量，这使得页面设计更加容易，因为 JSP 的标签代码通常在可视化的 HTML 编辑器中无法正常显示。Velocity 页面还能使用任意扩展名，包括.html，因此，可以直接在浏览器中预览页面的效果。

（3）通常，Velocity 还提供了比 JSP 更快的渲染速度。

要在 Spring MVC 框架中应用 Velocity，首先需要配置 Velocity 引擎。在 Spring 中使用 Velocity 比单独使用 Velocity 更加容易，也更加方便，我们甚至根本不需要与 Velocity 的 API 打交道，所有 Velocity 相关组件均是在 XML 配置文件中定义的。我们将 Spring MVC 项目复制一份，命名为 Spring_Velocity，结构如图 7-41 所示。

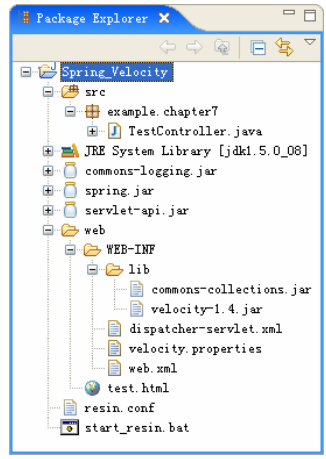


图 7-41

Velocity 需要 velocity.jar 和 commons-collections.jar 这两个 jar 包，将其与 Spring 的相关 jar 包一同放入/web/WEB-INF/lib 目录下。由于 Velocity 不需要 taglib，修改 web.xml，将 taglib 的声明删除。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
   <servlet>
      <servlet-name>dispatcher</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>0</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>*.html</url-pattern>
   </servlet-mapping>
</web-app>
```

然后修改 dispatcher-servlet.xml，添加 velocityConfigurer 配置 Velocity 引擎，并设置 viewResolver 为 VelocityViewResolver。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd">

            <bean name="/test.html" class="example.chapter7.TestController" />

            <!-- 使用 Velocity 视图解析器 -->
            <bean id="viewResolver" class="org.springframework.web.servlet.view.
velocity.VelocityViewResolver">
                <property name="contentType" value="text/html;charset=UTF-8" />
                <property name="prefix" value="/" />
                <property name="suffix" value=".html" />
            </bean>
            <!-- 配置 Velocity -->
            <bean id="velocityConfig" class="org.springframework.web.servlet.view.
velocity.VelocityConfigurer">
                <!-- 配置文件位置 -->
                <property name="configLocation" value="/WEB-INF/velocity.properties" />
                <!-- 视图资源位置 -->
                <property name="resourceLoaderPath" value="/" />
            </bean>
        </beans>
```

Velocity 还需要一个配置文件，用于设置 Velocity 引擎。配置文件的每个选项都可以注入到 Spring 的 XML 配置文件中，但是，由于 Velocity 的配置选项较多，放在单独的 velocity.properties 中比较合适。典型的配置如下，读者只需要注意几个重要的配置选项。

```
runtime.log.logsystem.class = org.apache.velocity.runtime.log.SimpleLog4JLogSystem

runtime.log = example.chapter7

runtime.log.error.stacktrace = true
runtime.log.warn.stacktrace = true
runtime.log.info.stacktrace = false
runtime.log.invalid.reference = true

# 设置输入/输出的编码:
input.encoding = UTF-8
output.encoding = UTF-8

directive.foreach.counter.name = velocityCount
# 设置 foreach 循环的 index 初始值:
directive.foreach.counter.initial.value = 1
directive.include.output.errormsg.start = <!-- include error :
directive.include.output.errormsg.end   =  see error log -->
directive.parse.max.depth = 3
```

```
        # 设置输入模版来自文件:
        resource.loader = file
        file.resource.loader.description = Velocity-File-Resource-Loader
        # 设置模版的加载类:
        file.resource.loader.class = org.apache.velocity.runtime.resource.loader.
FileResourceLoader
        # 设置是否使用 cache,在开发期可禁用 cache 以便随时编辑页面:
        file.resource.loader.cache = false
        # 设置检测文件改动的时间间隔,在运行期可设置较大的数,如 3600(1 小时):
        file.resource.loader.modificationCheckInterval = 1

        # 设置 macro 文件位置:
        velocimacro.library = /macro.txt
        velocimacro.library.autoreload = true
```

Velocity 使用一种 VTL 语言来渲染视图，VTL 有非常简单的语法，它通过${var_name}来输出变量，支持循环，条件判断和赋值。将 test.jsp 重命名为 test.html，修改内容如下。

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Spring_Velocity</title>
</head>
<body>
    <h3>Hello, ${name}, it is ${time}</h3>
</body>
</html>
```

虽然 Velocity 一般使用.vm 作为页面的扩展名，不过，Velocity 视图是纯 HTML 页面，因此我们使用.html 作为扩展名，这样在可视化 HTML 编辑器中编辑时非常直观，并且可以在不启动服务器的情况下就能预览页面效果，大大简化了页面的设计和实现，如图 7-42 所示。

启动 Resin 服务器，输入 http://localhost/test.html，可以看到由 Velocity 渲染的页面，如图 7-43 所示。

图 7-42

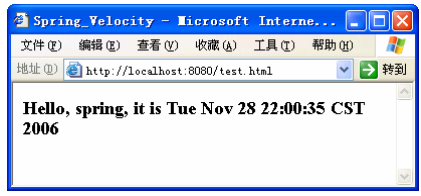


图 7-43

Velocity 还提供了 Macro（宏）的功能，能将反复引用的一组复杂的 HTML/Velocity 代码通过一个 Macro 定义并引用，极大地简化页面的设计。Macro 相当于 JSP 的自定义标签，但 JSP 的自定义标签编写极其复杂，而用户编写的 Macro 仍是 HTML/Velocity 代码，因此相对容易得多。

7.5.2 Freemarker

Freemaker 是取代 JSP 的又一种视图技术，和 Velocity 非常类似，但是它比 Velocity 多了一个格式化的功能，因此使用上较 Velocity 方便一点，但语法也稍微复杂一些。

将 Velocity 替换为 Freemarker 只需要改动一些配置文件，同样，在 Spring 中使用 Freemarker 也非常方便，根本无须与 Freemarker 的 API 打交道。我们将 Spring_Velocity 工程复制一份，命名为 Spring_Freemarker，结构如图 7-44 所示。

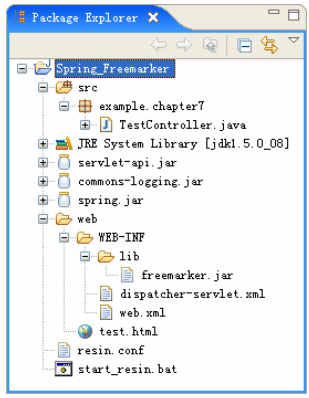


图 7-44

修改 dispatcher-servlet.xml，将 velocityConfig 删除，修改 viewResolver 为 FreeMarker ViewResolver，并添加一个 freemarkerConfig。

```
        <!-- 使用 Freemarker 视图解析器 -->
        <bean id="viewResolver" class="org.springframework.web.servlet.view.
freemarker.FreeMarkerViewResolver">
            <property name="contentType" value="text/html;charset=UTF-8" />
            <property name="prefix" value="/" />
            <property name="suffix" value=".html" />
        </bean>

        <!-- 配置 Freemarker -->
        <bean id="freemarkerConfig" class="org.springframework.web.servlet.view.
freemarker.FreeMarkerConfigurer">
```

```
    <!-- 视图资源位置 -->
    <property name="templateLoaderPath" value="/" />
    <property name="defaultEncoding" value="UTF-8" />
</bean>
```

模版 test.html 可以稍做修改，加入 Freemarker 内置的格式化功能来定制 Date 类型的输出格式。

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Spring_Freemarker</title>
</head>
<body>
    <h3>Hello, ${name}, it is ${time?string("yyyy-MM-dd HH:mm:ss")}</h3>
</body>
</html>
```

添加 freemarker.jar 到 web/WEB-INF/lib 目录后，启动 Resin，可以看到由 Freemarker 渲染的页面，如图 7-45 所示。

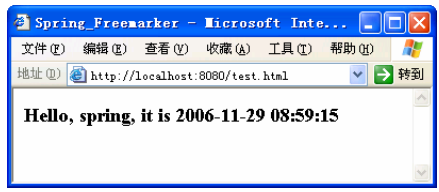


图 7-45

7.5.3 XSLT

XSLT（XSL 是 eXtensible Stylesheet Language 的缩写，而 XSLT 代表 XSL Transformations）技术是为了将 XML 格式的文档转化为任意格式的文本文档，当然，主要用途之一就是将 XML 转化为 HTML。使用 XML+XSLT 能实现最严格的数据和显示分离：XML 仅包含数据，而最终的渲染页面则交给 XSLT 来完成，如图 7-46 所示。

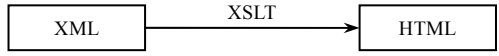


图 7-46

XSLT 可以实现 XML 到 HTML 的转化，转化过程既可以在服务器端完成，也可以在浏览器端完成。如果在服务器端转化，则客户端根本不知道服务器端使用的具体技术，

不过，由于 XSLT 的转化非常耗费 CPU 资源，因此，在访问量大的情况下，要考虑服务器是否能够负载，相比之下，在浏览器端转化就不需要耗费太多的服务器资源，服务器只需要向浏览器传送 XML 和 XSLT 文件，由浏览器自己完成 XML 到 HTML 的转化，这种方式减轻了服务器的负担，不过，客户端就得知了服务器端采用的技术，并且早期的浏览器还不支持 XML 的转化，但是现在绝大多数浏览器都没有问题。在浏览器端转化的另一个问题是搜索引擎无法正确地抓取页面，因为搜索引擎一般只抓取 HTML 格式的页面，对于仅包含纯数据的 XML，搜索引擎一般无法分析其内容。

在 Spring 中使用 XSLT 和其他视图技术类似，我们在 Eclipse 中新建一个 Spring_Xslt 工程，结构如图 7-47 所示。

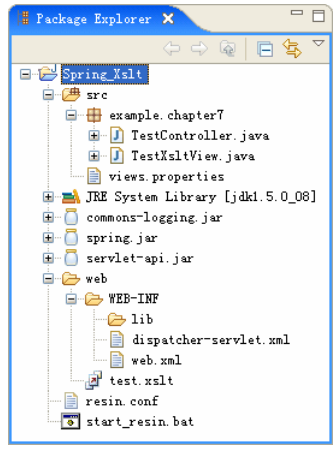


图 7-47

其中，TestController 不变，test.html 更改为 test.xslt，负责将 XML 转化为 HTML。这里的一个问题是，Spring 的 Controller 返回的 Model 是 Map 类型，必须将其首先转化为 XML 才能用 XSLT 完成到 HTML 的转化。遗憾的是，从 Map 类型到 XML 没有直接的转化过程，由于 Map 中包含的数据类型也各不相同，因此，还必须手动编写 TestXsltView 类，完成 Map 到 XML 的转化，该类必须从 AbstractXsltView 派生，并且复写 createXsltSource()方法。

```
        public class TestXsltView extends AbstractXsltView {
            protected Source createXsltSource(Map model, String rootName,
HttpServletRequest request, HttpServletResponse response) throws Exception {
                Document document = DocumentBuilderFactory.newInstance().newDocument
Builder().newDocument();
                Element root = document.createElement(rootName);
                // 添加<name>:
                String name = (String)model.get("name");
```

```
            Element nameNode = document.createElement("name");
            nameNode.appendChild(document.createTextNode(name));
            root.appendChild(nameNode);
            // 添加<time>:
            Date time = (Date)model.get("time");
            Element timeNode = document.createElement("time");
            timeNode.appendChild(document.createTextNode(time.toString()));
            root.appendChild(timeNode);
            return new DOMSource(root);
        }
    }
```

为了使用 XSLT，在 dispatcher-servlet.xml 中配置使用 ResourceBundleViewResolver 作为 ViewResolver。

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

        <bean name="/test.html" class="example.chapter7.TestController" />
        <bean id="viewResolver" class="org.springframework.web.servlet.view.
ResourceBundleViewResolver">
            <property name="basename" value="views" />
        </bean>
    </beans>
```

定义了 Map 到 XML 的转化后，还需要一个 views.properties 配置文件来告诉 Spring 该 TestXsltView 生成的 XML 应该用哪个 XSLT 来完成到 HTML 的转化，该文件必须放到 src 目录下。

```
    test.class=example.chapter7.TestXsltView
    #test.stylesheetLocation=/test.xslt
    test.root=DocRoot
```

我们先把 test.stylesheetLocation 注释掉，这样 Spring 就可以直接将 XML 返回，我们首先检查生成的 XML 是否正确。启动 Resin，输出如图 7-48 所示。

图 7-48

现在，我们编写 test.xslt 来转化生成的 XML 为 HTML，放到 web 目录之下。

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" />
    <xsl:template match="/">
        <html><head>
            <title>Spring_Xslt</title>
        </head>
        <body>
            <h3>Hello,
                <xsl:value-of select="DocRoot/name" />, it is
                <xsl:value-of select="DocRoot/time" /></h3>
        </body></html>
    </xsl:template>
</xsl:stylesheet>
```

将 views.properties 的注释去掉，然后重新启动 Resin，可以看到由 XSLT 转化而成的 HTML，如图 7-49 所示。

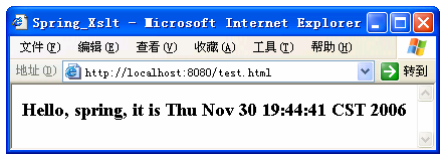


图 7-49

一般来说，如果使用 XSLT，由于视图部分总是要先设计好 HTML 页面，然后才能编写 XSLT 来转化 XML 到指定的 HTML 格式。如果 HTML 页面结构发生了较大的改动，则整个 XSLT 都必须全部重写，其维护成本是非常高的。目前，在还没有任何可视化编辑器辅助设计的情况下，不推荐使用 XSLT。如果应用程序本身已经有了 XML 数据，则可以考虑使用 XSLT 来完成 HTML 转化。

7.5.4 混合使用多种视图技术

如果需要混合使用多种视图技术，就需要设置合适的 ViewResolver，并且让其有能力解析不同的视图名称。

以 Web 形式启动的 Spring 应用程序会自动查找所有具有 ViewResolver 接口的 Bean，将它们作为一个视图解析链来使用。默认地，声明在前的 ViewResolver 具有优先解析视图的权力，也可以用 order 属性来定义 ViewResolver 的解析顺序。

Spring 会按照 ViewResolver 的解析链来让每个 ViewResolver 试图解析并返回一个

View，如果某个 ViewResolver 返回了 View，则解析过程结束，然后调用 View 对象的 render()方法开始真正的渲染任务。例如，InternalResourceViewResolver 会返回 JstlView 视图，VelocityViewResolver 会返回 VelocityView，FreeMarkerViewResolver 会返回 FreeMarkerView，不同的 View 会有不同的渲染方式。

不过，某些情况下，一个 ViewResolver 可能无法根据视图的逻辑名检测到一个 View 是否存在。例如，InternalResourceViewResolver 只能调用 RequestDispatcher 来检测 JSP 文件是否存在，不幸的是，该方法只能调用一次，这可能会使后续的 ViewResolver 的解析出现问题。

另一个不太令我们满意的原因是使用 ViewResolver 链效率较低，它是通过循环来实现的，可以在 DispatcherServlet 中看到源代码。

```
for (Iterator it = this.viewResolvers.iterator(); it.hasNext();) {
    ViewResolver viewResolver = (ViewResolver) it.next();
    View view = viewResolver.resolveViewName(viewName, locale);
    if (view != null) {
        return view;
    }
}
```

我们最好不要让 ViewResolver 链来解析视图，因为这样每个 ViewResolver 只能依次“猜测”View 是否存在，如果能根据视图的扩展名来决定由哪个 ViewResolver 来解析，则只需要一步就可完成视图的解析。

一个可行的方法是实现一个自定义的 MixedViewResolver，并直接和 Spring MVC 框架关联，然后 MixedViewResolver 根据视图的扩展名来决定将解析委托给哪个具体的 ViewResolver。例如，对于扩展名为.jsp 的视图，就用 InternalResourceViewResolver 解析；对于扩展名为.vm 的视图，就用 VelocityViewResolver 解析；对于扩展名为.ftl 的视图，就用 FreeMarkerViewResolver 来解析。这样，就实现了混合使用多种视图技术。

MixedViewResolver 仅实现 ViewResolver 接口，其实现的关键是将视图扩展名和具体的 ViewResolver 实例关联起来，从而实现根据视图扩展名来“转发”解析请求。

```
public class MixedViewResolver implements ViewResolver {
    private Map<String, ViewResolver> resolvers;
    public void setResolvers(Map<String, ViewResolver> resolvers) {
        this.resolvers = resolvers;
    }

  public View resolveViewName(String viewName, Locale locale) throws Exception {
        int n = viewName.lastIndexOf('.');
        if(n==(-1))
```

```
                throw new NoSuchViewResolverException();
            // 获得扩展名:
            String suffix = viewName.substring(n+1);
            // 取出对应的 ViewResolver:
            ViewResolver resolver = resolvers.get(suffix);
            if(resolver!=null)
                return resolver.resolveViewName(viewName, locale);
            // 没有找到对应的 ViewResolver 就抛异常:
            throw new NoSuchViewResolverException("No ViewResolver for " + suffix);
        }
    }
```

由于需要根据视图的扩展名来决定到底使用何种 ViewResolver，所以就不能配置 ViewResolver 的 suffix 属性，可以配置一个 prefix 属性，但是本例中为了简化，始终让 Controller 返回绝对路径的视图。

在 dispatcher-servlet.xml 配置文件中，定义 viewResolver 如下。

```
    <bean id="viewResolver" class="example.chapter7.MixedViewResolver">
        <property name="resolvers">
            <map>
                <entry key="jsp">
                    <bean class="org.springframework.web.servlet.view.Internal
ResourceViewResolver">
                        <property name="viewClass" value="org.springframework.
web.servlet.view.JstlView" />
                    </bean>
                </entry>
                <entry key="vm">
                    <bean class="org.springframework.web.servlet.view.velocity.
VelocityViewResolver">
                        <property name="contentType" value="text/html;charset=UTF-8" />
                    </bean>
                </entry>
                <entry key="ftl">
                    <bean class="org.springframework.web.servlet.view.freemarker.
FreeMarkerViewResolver">
                        <property name="contentType" value="text/html;charset=UTF-8" />
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
```

扩展名“.jsp”、“.vm”和“.ftl”分别对应 3 种 ViewResolver。当然，velocityConfig

和 freemarkerConfig 的定义也必不可少。然后修改 TestController，我们根据 URL 的参数来决定返回的视图名称。

```
        public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
Response response) throws Exception {
            String view = request.getParameter("view");
            if(view==null)
                view = "jsp"; // 默认使用 jsp
            String name = request.getParameter("name");
            if(name==null)
                name = "spring";
            Map model = new HashMap();
            model.put("name", name);
            model.put("time", new Date());
            // "/test." + view 就是视图的完整路径:
            return new ModelAndView("/test." + view, model);
        }
```

我们在 web/目录下分别放置了 test.jsp、test.vm 和 test.ftl 这 3 个视图文件，整个 Spring_Mixed 工程的结构如图 7-50 所示。

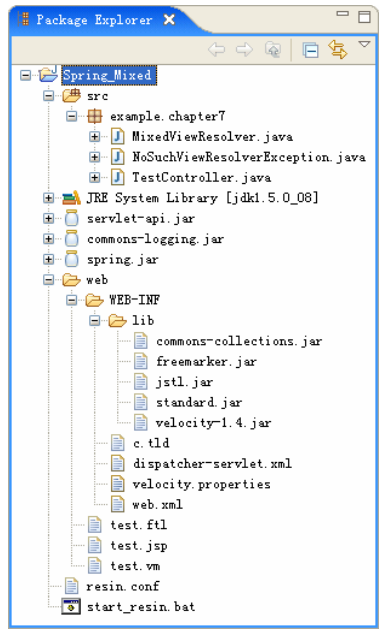


图 7-50

配置好 web.xml（不要忘记了声明 JSP 的 taglib）和 velocity.properites，确保所有的 jar 包都在 WEB-INF/lib 目录下，编译工程，启动 Resin 服务器，分别输入“http://localhost:

8080/test.html?view=jsp”、“http://localhost:8080/test. html?view=vm”和“http://localhost:8080/test.html? view=ftl”，观察浏览器输出，如图 7-51~图 7-53 所示。

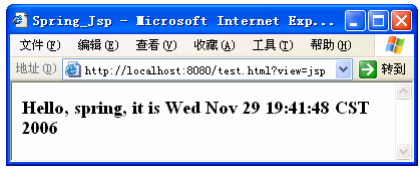


图 7-51

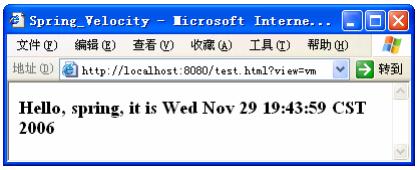


图 7-52

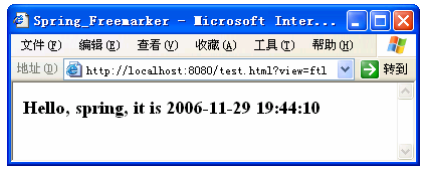


图 7-53

可以看到，对于 TestController 返回的同一个 Model，Spring MVC 使用了 3 个不同的 ViewResolver 来解析并渲染视图。

从 AbstractCachingViewResolver 继承而来的 ViewResolver 还可以具有 Cache 的能力，将解析过的视图缓存起来，某些情况下能大大提高解析速度。但是，在 MixedViewResolver 中我们并不需要缓存功能，因为其包含的其他 ViewResolver 可能已经具有缓存的功能了。MixedViewResolver 仅仅实现“转发”解析请求的功能，如果实际负责解析的 ViewResolver 具有缓存的功能，则缓存就会生效，因为相同的视图名称总是会转发给同一个 ViewResolver。

我们已经成功地让多种视图技术共存于 Spring MVC 框架中，这得益于 Spring MVC 框架极为灵活的低耦合设计，并且基于接口的设计使我们能够非常方便地插入自己的实现，从而扩展 Spring 的功能。

7.5.5 几种视图技术的比较

JSP 作为标准的 JavaEE 规范之一，已经获得了广泛的使用，然而，由于可以嵌入 Java 代码，如果不严格控制 Controller 和 View 的界限，很容易造成 JSP 页面混入逻辑代码，导致 Web 应用程序的维护成本大幅上升。此外，如果大量使用标签库，也容易造成页面布局混乱，使页面设计人员难以在可视化编辑器中设计页面。JSP 2.0 标准已经开始支持类似 Velocity 的 EL 语法，即采用${var.name}获得属性值，而非复杂的<c:out value="var.name" />标签。

Velocity 最大的优势是非常简单的页面，由于不可嵌入 Java 代码，因此强制实现了 MVC 架构。

Freemarker 与 Velocity 非常类似，但多了一个格式化的功能，因此，使用上可能稍微方便一点。

XSLT 是 XML 技术的一部分，从本质上来讲，只有 XML+XSLT 才能真正实现数据的内容和显示相分离，然而，就目前而言，支持 XSLT 的工具还非常少，编写 XSLT 极其困难，因此不推荐采用 XSLT 作为视图。

在实际项目中，采用何种视图要根据需求而定。考虑到设计页面的通常是设计人员而非开发人员，因此，页面设计要尽量做到纯 HTML 格式，能使用可视化 HTML 编辑器设计，而 Velocity 和 Freemarker 无疑在这方面更胜一筹。在本书的 Live 在线书店应用中，视图技术就采用了 Velocity 而非标准的 JSP。读者可以看到，设计一个 Velocity 视图和设计一个 HTML 页面几乎没有什么区别，借助于 Velocity 的 Macro 功能，还能极大地简化分页等功能的实现。

7.6 集成其他Web框架

虽然 Spring 本身也提供了一个功能非常强大的 MVC 框架，并且和 Spring 的 IoC 容器无缝集成，非常便于使用。不过，在实际情况中，我们还不得不考虑众多采用第三方 MVC 框架的 Web 应用程序，例如，采用 Strtus、WebWork 等。如果要将这些应用程序的逻辑组件和持久化组件纳入 Spring 的 IoC 容器中进行管理，就必须考虑如何集成这些第三方的 MVC 框架。Spring 强大的集成和扩展能力使得集成第三方 MVC 框架成为轻而易举的事情。

与第三方 MVC 框架集成时，需要考虑的问题是如何在第三方 MVC 框架中访问 Spring IoC 容器管理的 Bean，或者说，如何获取 Spring IoC 容器的实例。我们还要保证在第三方 MVC 框架开始处理用户请求之前，Spring 的 IoC 容器必须初始化完毕。

有两种方式可以自动加载 Spring 的 IoC 容器。一种是利用第三方框架的扩展点，实现加载 Spring 的 IoC 容器，例如，Struts 就提供了 Plugin 扩展。第二种方式是在 web.xml 中定义 Listener 或 Servlet，让 Web 应用程序一启动就自动加载 Spring 的 IoC 容器。这种方式较为通用，对于一些不太常见的第三方 MVC 框架，也可以用这种方式来尝试与 Spring 集成。

如果正在使用基于 Servlet 2.3 或更高规范的 Web 服务器，则应当使用 Spring 提供的 ContextLoaderListener 来加载 IoC 容器，因为根据 Servlet 2.3 规范，所有的 Listener 都会在 Servlet 被初始化之前完成初始化。由于我们希望尽可能早地完成 Spring IoC 容器的初

始化，因此，采用 ContextLoaderListener 加载 Spring 的 IoC 容器是最合适的。

采用 ContextLoaderListener 加载 Spring 的 IoC 容器时，在/WEB-INF/web.xml 中定义如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

   ...

   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

   ...

</web-app>
```

默认地，ContextLoaderListener 会在/WEB-INF/目录下查找名为 applicationContext.xml 文件，作为 Spring 的 XML 配置文件加载。如果使用其他文件名，或者有多个 XML 配置文件，就需要预先在<context-param>中指定。

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/cfg1.xml,/WEB-INF/cfg2.xml</param-value>
</context-param>
```

如果使用的 Web 服务器还不支持 Servlet 2.3 规范，则无法使用 Listener，也就无法通过 ContextLoaderListener 来加载 Spring 的 IoC 容器。为此，Spring 提供了另一个 ContextLoaderServlet，以 Servlet 的形式来加载 Spring 的 IoC 容器。

```
<servlet>
   <servlet-name>contextLoader</servlet-name>
      <servlet-class>
         org.springframework.web.context.ContextLoaderServlet
      </servlet-class>
      <load-on-startup>0</load-on-startup>
</servlet>
```

ContextLoaderServlet 查找 Spring 的 XML 配置文件的方式与 ContextLoaderListener 完全一致，不过，由于必须首先加载 ContextLoaderServlet，然后加载其他的 Servlet，才能保证 Spring 的 IoC 容器在其他 Servlet 处理用户请求之前初始化完毕。因此，设置

<load-on-startup>为 0，表示 Web 服务器一启动就加载 ContextLoaderServlet，对于其他的 Servlet，<load-on-startup>的值要设得大一些，保证 ContextLoaderServlet 有足够的时间初始化 Spring 的 IoC 容器。

一旦完成了 Spring IoC 容器的加载，另一个问题是如何在第三方应用程序中获得 Spring IoC 容器的实例？

事实上，不管采用何种方式加载 Spring 的 IoC 容器，Spring 最终都会将 IoC 容器的实例绑定到 ServletContext 上。由于 ServletContext 在一个 Web 应用程序中是全局唯一的，因此该 Web 应用程序中所有的 Servlet 都可以访问到这个唯一的 ServletContext，也就可以获得 Spring IoC 容器的实例。Spring 提供了一个辅助类 WebApplicationContextUtils，通过调用 getWebApplicationContext(ServletContext)方法就可以获得 Spring IoC 容器的实例。对于参数 ServletContext，可以在 Servlet 中随时调用 getServletContext()获得。

一旦获得了 Spring IoC 容器的实例，就可以获得 Spring 管理的所有的 Bean 组件。

下面，我们分别详细介绍如何在 Spring 中集成 Struts、WebWork2、Tiles 和 JSF。

7.6.1 集成Struts

Struts 是目前 JavaEE Web 应用程序中应用最广泛的 MVC 开源框架，自从 2001 年发布以来，Struts 作为 JavaEE 领域的第一个 MVC 框架，极大地简化了基于 JSP 和 Servlet 的 Web 开发，提供了统一的 MVC 编程模型。虽然从现在看来，Struts 的设计模型已不算先进，有许多其他 MVC 框架拥有更好的设计，但 Struts 仍具有庞大的社区支持和最多的开发人员，这些都使得 Struts 仍是 JavaEE 领域内 Web 开发的首选框架。

不过，Strtus 仅仅是一个用于表示层的 MVC 框架，它并没有提供一个完整的 JavaEE 框架的解决方案。如果要将 Struts 集成到 Spring 框架中，有两种方案可供选择。下面，我们将详细讲解如何将 Struts 集成到 Spring 框架中。

在集成 Struts 之前，我们假定已经有了一个基于 Struts 的 Web 应用程序，这里我们使用的例子是一个简单的处理用户登录的 Struts 应用，这个 Struts 应用在浏览器中的效果如图 7-54 所示。

图 7-54

在 Eclipse 中，我们建立了这个名为 Struts 的工程，其结构如图 7-55 所示。

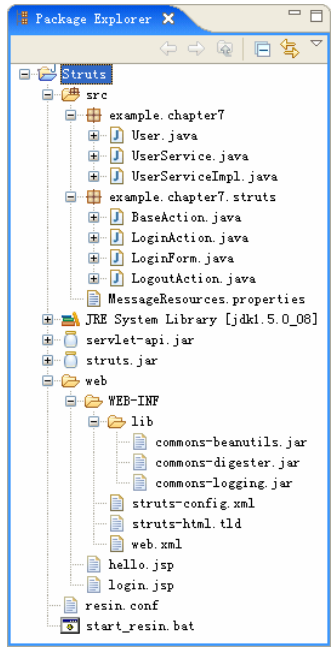


图 7-55

在 Struts 应用中，每个用户请求通过一个 Action 类来处理，这和 Spring 的 Controller 类似，但是 Struts 的 Action 是一个类而非接口，因此，所有的 Action 子类都只能从 Action 派生。由于 Struts 是一个 Web 层框架，需要考虑如何获得业务逻辑接口。一个较好的设计是首先设计一个 BaseAction，其中定义了获得业务逻辑接口的方法，其他所有的 Action 从 BaseAction 派生即可非常方便地调用业务逻辑接口。

```
public class BaseAction extends Action {
    private static final UserService userService = new UserServiceImpl();

    public UserService getUserService() {
        return userService;
    }
}
```

在 BaseAction 中，我们以静态变量持有业务逻辑接口 UserService 的引用，这是一个不太优雅的设计。如果使用 Spring 的 IoC 容器来配置和管理这些逻辑组件，则可以完全实现一个优雅的多层应用程序，Struts 只处理 Web 表示层，业务逻辑层和数据持久层都交由 Spring 管理，便于维护和扩展。

在 Spring 中有两种方式来集成 Struts，关于 Struts 的更多详细用法的讨论已经超出

了本书的范围。在本节中，假定读者对 Struts 已经有了一定的基础。下面我们分别介绍 Spring 集成 Struts 的两种方案，两种方案都需要 Spring 提供的一个名为 ContextLoader Plugin 的 Struts 插件来启动 IoC 容器。由于是 Web 应用程序，ContextLoaderPlugin 启动的是 WebApplicationContext 的实例。

第一种方案是通过 Spring 提供的一个 Struts 插件来初始化 IoC 容器，然后从 Spring 提供的 ActionSupport 派生所有的 Action，以便能通过 getWebApplicationContext()获得 ApplicationContext，一旦获得了 ApplicationContext 引用，就可以获得 Spring 的 IoC 容器中所有的 Bean。我们建立一个 Struts_Spring1 工程，首先复制 Struts 工程的所有文件，然后在 Struts 配置文件的最后添加 Spring 的插件声明。

```
<struts-config>
    ...
   <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn" />
</struts-config>
```

然后修改 BaseAction，将其超类从 Struts 的 Action 改为 Spring 的 ActionSupport。

```
public class BaseAction extends ActionSupport {
    public UserService getUserService() {
        return (UserService) getWebApplicationContext()
              .getBean("userService");
    }
}
```

现在，BaseAction 就可以随时通过 Spring 的 ApplicationContext 获得逻辑组件 UserService 的引用，这样所有的 Action 子类都可以直接通过 getUserService()获得 UserService 组件。

最后一步是编写 Spring 的 XML 配置文件，默认的文件名是<servlet-name>-servlet.xml，由于我们在 web.xml 中配置 Struts 的 ActionServlet 名称为 action，因此，Spring 的配置文件为 action-servlet.xml，放到 web/WEB-INF/目录下，定义所有的 Bean 组件，但不包括 Struts 的 Action 实例。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="userService" class="example.chapter7.UserServiceImpl" />
</beans>
```

如果 Spring 的 XML 配置文件没有使用默认的文件名，就必须在 Struts 配置文件中

声明 Plugin 时指定文件位置。

```
<struts-config>
    ...
   <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
       <set-property property="contextConfigLocation"
          value="/WEB-INF/my-spring-config.xml"/>
    </plug-in>
</struts-config>
```

整个 Struts_Spring1 工程的结构如图 7-56 所示。

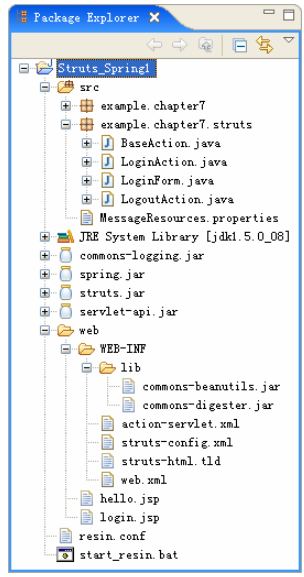


图 7-56

编译工程，然后启动 Resin 服务器，可以看到效果和原始的 Struts 工程一样，但是，业务逻辑层组件和持久层组件（在本例中，为了简化问题，没有设计持久层组件）已被纳入 Spring 的 IoC 容器之中，从而清晰地将表示层和业务层划分开来。

使用这种方式集成 Spring 和 Struts 时，如果合理地抽象出一个类似 BaseAction 的类作为所有 Action 的超类，在集成到 Spring 时，只需将 BaseAction 的超类从 Struts 的 Action 类改为 Spring 提供的 ActionSupport，然后在 Struts 中声明 Spring 的 Plugin，就可以在不修改任何 Action 的情况下实现和 Spring 的集成。

第二种集成 Struts 的方案是将 Struts 的所有 Action 都作为 Spring IoC 容器中的 Bean 来管理，就像 Spring 的 Controller 一样。然后通过 Spring 提供的 DelegatingRequestProcessor 来替代 Struts 的默认派发器，以便让 Spring 能截获派发给 Action 的请求。在这种方式下，

业务逻辑组件通过依赖注入的方式在 Spring 的 IoC 容器中就配置完毕了。

我们仍建立一个 Struts_Spring2 的工程，将 Struts 工程的所有文件复制过来，其目录结构和 Struts_Spring1 类似，如图 7-57 所示。

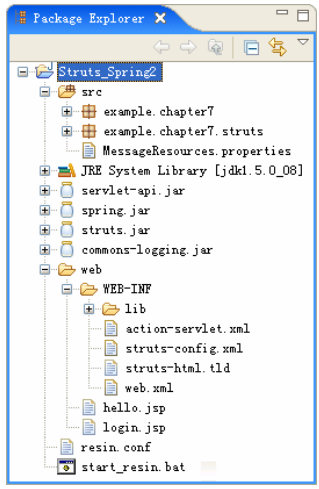


图 7-57

为了能将业务组件 UserService 注入到每个 Action 类中，抽象出一个 BaseAction 是非常重要的，这样可以使代码的修改被限制在 BaseAction 中，而不会涉及每一个 Action 类。修改 BaseAction 如下。

```
public class BaseAction extends Action {
    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    public UserService getUserService() {
        return userService;
    }
}
```

然后修改 Struts 的配置文件，添加 Spring 提供的 DelegatingRequestProcessor 和 Plugin 的声明。

```
<struts-config>
    ...
    <controller processorClass="org.springframework.web.struts.DelegatingRequestProcessor"/>
```

```
        ...
        <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn" />
    </struts-config>
```

Spring 提供的 Struts Plugin 仍负责启动 Spring 容器，而 DelegatingRequestProcessor 可以让 Spring 接管请求，从而将请求派发到 IoC 容器管理的 Action 实例。为此，在 Spring 的 XML 配置文件中，除了定义业务逻辑组件和其他组件外，还必须声明每一个 Action 类，其 name 属性和 Struts 配置文件中的 path 要完全一致。由于我们在 Struts 配置文件中定义了两个 Action：

```
    <action-mappings>
        <action path="/login" type="example.chapter7.struts.LoginAction" ... />
        <action path="/logout" type="example.chapter7.struts.LogoutAction" ... />
    </action-mappings>
```

因此，在 Spring 的配置文件中，除 UserService 组件外，还必须定义两个 Action，并且要和 Struts 配置文件中的 Action 一一对应。

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
        <bean id="userService" class="example.chapter7.UserServiceImpl" />

        <bean name="/login" class="example.chapter7.struts.LoginAction">
            <property name="userService" ref="userService" />
        </bean>

        <bean name="/logout" class="example.chapter7.struts.LogoutAction">
            <property name="userService" ref="userService" />
        </bean>
    </beans>
```

实际上，如果使用这种方案实现集成，Struts 配置文件中 Action 的 type 属性将被忽略，因为 Spring 会根据 path 属性查找对应 name 的 Action Bean。但是，仍然建议将 type 属性标识出来，便于查看 URL 和 Action 的关联。

使用这种方案时，由于 Action 被作为 Bean 纳入 Spring 的 IoC 容器管理，因此，可以获得完全的依赖注入能力。不过，最大的不便是所有的 Action 必须同时在 Struts 和 Spring 的配置文件中各配置一次，这增加了配置文件的维护成本。

在集成 Struts 时，选择何种方案需要根据实际情况决定。例如，由于 RequestProcessor 是 Struts 的一个扩展点，如果现有的 Web 应用程序已经扩展了 RequestProcessor，采用

第一种方式就比较合适。不过，无论采用哪种方案，基于 Struts 的整体设计至关重要，如果没有统一的业务逻辑接口，而是将业务逻辑散落在各个 Action 中，则对代码的修改将涉及所有的 Action 类。

7.6.2 集成WebWork2

WebWork 是一个非常简洁和优雅的 Web 框架，WebWork 的架构设计非常容易理解，它构建在一个命令模式的 XWork 框架之上，支持多种视图技术，并且 WebWork 也有一个丰富的标签库，能非常容易地实现校验。WebWork 最大的特色就是使用 IoC 容器来管理所有的 Action，并且拥有拦截器等一系列类似 AOP 的概念，因此，WebWork 的整体设计比 Struts 更优秀，也更容易扩展。

WebWork 目前有两个主要版本：WebWork 1.x 和 WebWork 2.x，两个版本的差异较大，本节介绍的均以最新的 WebWork 2.2 为例。可以从 WebWork 的官方站点 http://www.opensymphony.com/webwork/下载最新的 2.2.4 版。

在 WebWork 2.2 版本之前，WebWork 有一个自己的 IoC 容器，尽管仍可以和 Spring 集成，但是不那么方便。从 WebWork 2.2 开始，WebWork 的设计者就推荐使用 Spring 的 IoC 容器来管理 WebWork2 的 Action，这对于使用 Spring 框架的开发者来说无疑是一个好消息，因为这使得两者的集成更加容易，我们可以像对待 Spring 的 Bean 一样，在 Spring 的 IoC 容器中直接配置 WebWork2 的 Action。

WebWork2 的开发团队已经提供了一个 Spring 和 WebWork2 的集成方案，正因为如此，在 Spring 框架中并没有 WebWork2 相关的支持包。如果读者在集成 WebWork2 时遇到问题，请首先参考 WebWork 官方网站（http://www.opensymphony.com/webwork/）的相关文档。

如果读者正在考虑使用 WebWork2 作为 MVC 框架，则理所当然应当集成到 Spring 框架中。在下面的例子中，我们以最新的 WebWork 2.2.4 为例，首先创建一个基于 WebWork2 的 Web 应用程序，命名为 WebWork2，其 Eclipse 工程结构如图 7-58 所示。

WebWork2 所需的 jar 文件除了 webwork-2.2.4.jar 外，其余必需的 jar 包可以从解压后的 WebWork2 的 lib/default/目录下找到，将这些 jar 包（javamail.jar 可选）全部复制到/WEB-INF/lib 目录下。其中，xwork.jar 是编译时必须的，将其添加到 Eclipse 的 Build Path 中。由于 WebWork2 框架是基于 XWork 框架之上的，对开发者而言，开发 WebWork2 的 Action 甚至不用和 Servlet API 打交道，这种设计大大提高了 Web 应用程序的可测试性。

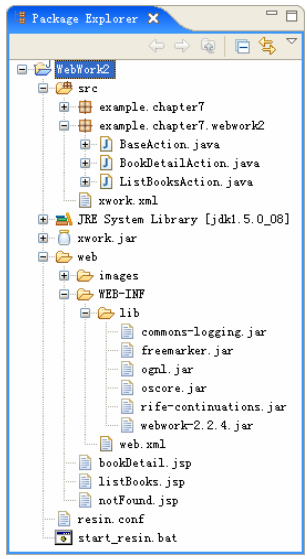


图 7-58

我们假定读者对 WebWork2 已经有了一定的了解。在这个 Web 应用程序中，我们定义了两个 Action，ListBooksAction 用于列出所有的书籍，BookDetailAction 用于查看书籍的详细信息，为此，我们还定义了一个 BookService 接口，并在所有 Action 的超类 BaseAction 中定义了获取 BookService 接口的方法 getBookService()。

```
public abstract class BaseAction implements Action {
    private static BookService bookService = new BookServiceImpl();

    public BookService getBookService() {
        return bookService;
    }
}
```

然后，在 xwork.xml 中配置好两个 Action，并放到 src 目录下，这样，编译后该文件就位于/WEB-INF/classes 目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xwork
        PUBLIC
        "-//OpenSymphony Group//XWork 1.0//EN"
        "http://www.opensymphony.com/xwork/xwork-1.0.dtd">
<xwork>
    <include file="webwork-default.xml"/>
    <package name="default" extends="webwork-default">
```

```
        <action name="listBooks" class="example.chapter7.webwork2.ListBooksAction">
            <result name="success" type="dispatcher">/listBooks.jsp</result>
        </action>

        <action name="bookDetail" class="example.chapter7.webwork2.
BookDetailAction">
            <result name="success" type="dispatcher">/bookDetail.jsp</result>
            <result name="error" type="dispatcher">/notFound.jsp</result>
        </action>
    </package>
</xwork>
```

在 web.xml 中，我们声明 WebWork2 的 ServletDispatcher，并映射所有以.action 结尾的 URL。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            com.opensymphony.webwork.dispatcher.ServletDispatcher
        </servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>

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

    <taglib>
        <taglib-uri>webwork</taglib-uri>
        <taglib-location>/WEB-INF/lib/webwork-2.2.4.jar</taglib-location>
    </taglib>
</web-app>
```

现在，作为一个独立的 WebWork2 的应用程序，我们已经可以在浏览器中看到实际效果了。启动 Resin 服务器，然后输入地址“http://localhost:8080/listBooks.action”，如图 7-59 所示。

我们还没有将 Spring 集成进来，并且和 7.6.1 节的 Struts 应用程序类似，获取 BookService 接口设计得不那么优雅。下一步，我们将 WebWork2 与 Spring 集成起来，让 Spring 来管理所有的 Action 和其他逻辑组件。

图 7-59

我们将 WebWork2 工程复制一份，命名为 WebWork2_Spring，然后开始一步一步地将其集成到 Spring 框架中。

在 WebWork2 中集成 Spring 是一件非常容易的事情。在 WebWork2 中，所有的 Action 也是由 IoC 容器管理的、默认地，WebWork2 自身有一个 IoC 容器，负责管理所有的 Action，但是，正如前面提到的，从 WebWork 2.2 开始，WebWork2 设计者推荐使用 Spring 的 IoC 容器来管理 Action。只要对配置文件稍做修改，就可以让 WebWork2 直接使用 Spring 的 IoC 容器。

第一步是编写一个 webwork.properties 的配置文件，放在 src 目录下，指定 WebWork2 使用的 IoC 容器名称。

```
webwork.objectFactory = spring
```

该配置文件非常简单，只需要以上一行内容即可。更多的配置选项可以参考 WebWork2 的文档。编译后，该文件就会被放到/WEB-INF/classes/目录下。

现在，只要 Spring 的 IoC 容器启动了，WebWork2 就会自动感知并使用 Spring 的 IoC 容器。要启动 Spring 的 IoC 容器，使用 ContextLoaderListener 最合适不过，在 web.xml 中添加 Spring 的 ContextLoaderListener 声明。

```
<listener>
   <listener-class>
      org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

下一步是在 Spring 默认的 XML 配置文件中定义所有的 Bean，包括 WebWork2 的 Action。由于使用 ContextLoaderListener 来启动 Spring 的 IoC 容器，因此，默认的 XML 配置文件名称为 applicationContext.xml，内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
>
    <bean id="bookService" class="example.chapter7.BookServiceImpl" />

    <bean id="listBooksAction" class="example.chapter7.webwork2. ListBooks
Action" scope="prototype">
        <property name="bookService" ref="bookService" />
    </bean>

    <bean id="bookDetailAction" class="example.chapter7.webwork2. BookDetail
Action" scope="prototype">
        <property name="bookService" ref="bookService" />
    </bean>
</beans>
```

要特别注意的是，WebWork2 使用的 Action 和 Struts 不同，对于每个用户请求，WebWork2 都会创建一个新的 Action 实例来处理用户请求，因此，必须将 Action 的 scope 定义为 prototype，使得每次 WebWork2 向 Spring IoC 容器请求一个 Action 时，Spring 都能返回一个新的实例。采用 Bean 模版也不失为一个好办法，具体配置请参考第 3 章 3.9.2 “使用模版装配”一节。

最后一步是修改 xwork.xml，将每个 Action 的 class 属性从类名改为 Spring 中对应的 id，这样，WebWork2 就会根据此 id 向 Spring 的 IoC 容器请求一个新的 Action 实例。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xwork
        PUBLIC
        "-//OpenSymphony Group//XWork 1.0//EN"
        "http://www.opensymphony.com/xwork/xwork-1.0.dtd">
<xwork>
    <include file="webwork-default.xml"/>
    <package name="default" extends="webwork-default">
        <action name="listBooks" class="listBooksAction">
            <result name="success" type="dispatcher">/listBooks.jsp</result>
        </action>

        <action name="bookDetail" class="bookDetailAction">
            <result name="success" type="dispatcher">/bookDetail.jsp</result>
            <result name="error" type="dispatcher">/notFound.jsp</result>
        </action>
```

```
        </package>
    </xwork>
```

完成了以上步骤后，将 spring.jar 放入/WEB-INF/lib 目录下，然后启动 Resion 服务器，就可以看到如下输出。

```
    [2006/12/06 16:41:43.138] Initializing WebWork-Spring integration...
    [2006/12/06 16:41:43.154] Setting autowire strategy to name
    [2006/12/06 16:41:43.154] ... initialized WebWork-Spring integration
successfully
```

打开浏览器测试，其结果与前面 WebWork2 工程的结果完全相同，不过，所有的 Action 和逻辑组件都由 Spring 的 IoC 来管理，这使得应用程序的层次更加清晰。WebWork2_Spring 工程的整个结构如图 7-60 所示。

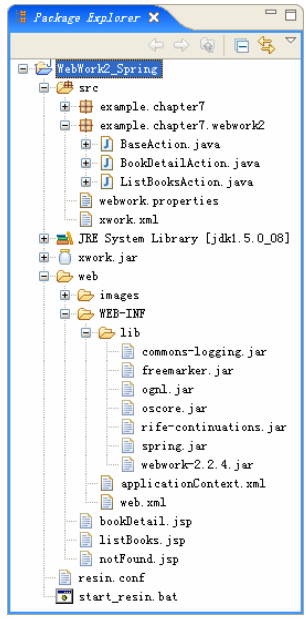


图 7-60

默认情况下，在使用 Spring 的 IoC 容器时，WebWork2 使用 byName 的自动装配功能来装配 WebWork2 的 Action。可以根据需要将其修改为 byType。如果 Action 较多，更好的配置方法是首先定义一个模版 Bean，作为所有 Action Bean 的模版。

```
    <bean id="webworkActionTemplate" abstract="true" scope="prototype">
        <property name="service1" ref="service1" />
        <property name="service2" ref="service2" />
    </bean>
```

其余的 Action Bean 通过继承此模版 Bean 就可以安全地获得 scope 设定和其他依赖注入的属性。

7.6.3 集成Tiles

前面我们讲到了 JavaEE Web 组件处理请求的 3 种方式：转发（Forward）、包含（Include）和错误（Error），Tiles 框架将 Include 方式发挥到了极致。Tiles 并不是一个 MVC 框架，从本质上说，Tiles 是一个模版框架，它将页面分成几个小的部分，然后动态地将它们组合在一起，从而允许更灵活地创建可以重用的页面组件。

虽然 Tiles 是 Struts 框架的一部分，但是 Tiles 从一开始就被设计为能够在 Struts 外独立适用。事实上，Tiles 可以用于任何 Web 框架，当然也包括 Spring 的 MVC 框架。

Spring 已经对 Tiles 做了非常好的集成，在 Spring 框架中使用 Tiles 更加方便。下面的例子我们创建了一个 Spring_Tiles 工程，结构如图 7-61 所示。

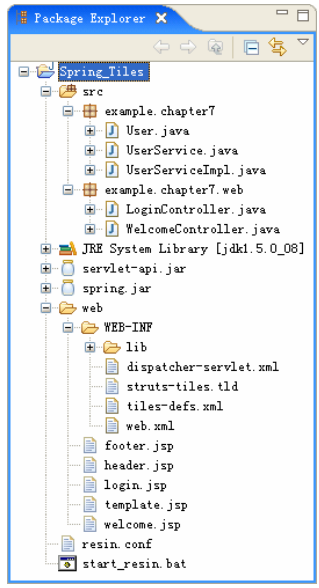


图 7-61

将 Tiles 所有依赖的 jar 包放入工程，包括 struts.jar、commons-beanutils.jar、commons-collections.jar、commons-digester.jar、commons-logging.jar 和 spring.jar。注意，Tiles 框架本身被包含在 struts.jar 中，但是我们并不使用 Struts 作为 MVC 框架。然后配置 web.xml，除了声明 Spring 的 DispatcherServlet 外，还要声明 Tiles 的 taglib。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
```

```
            "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
            "http://java.sun.com/dtd/web-app_2_3.dtd">
        <web-app>
            <servlet>
                <servlet-name>dispatcher</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
                <load-on-startup>0</load-on-startup>
            </servlet>

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

            <taglib>
                <taglib-uri>http://struts.apache.org/tags-tiles</taglib-uri>
                <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
            </taglib>
        </web-app>
```

我们先来看看如何在 Tiles 中组合出一个页面。在这个 Web 应用程序中，我们一共设计了两个页面，一个是登录页面，一个是欢迎页面，每个页面都被分为页眉（header）、主体（body）和页脚（footer）三部分。为了能在 Tiles 中组合出一个完整的页面，我们分别编写 header.jsp 作为页眉，footer.jsp 作为页脚，在登录页中，主体部分是 login.jsp，而在欢迎页中，主体是 welcome.jsp，如图 7-62 所示。

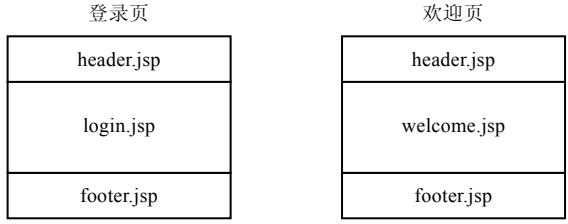


图 7-62

通过将页面拆为几个部分，我们就可以复用页面的公共部分（如 header.jsp 和 footer.jsp），在 Tiles 中，每个可重用的部分被称为一个可视化组件，通常是一个 JSP 文件，然后用一个模版将几个组件组合到一起，就构成了一个完整的页面。例如，template.jsp 将页眉、主体和页脚 3 个组件组合在一起，作为一个完整的页面展示给用户。

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="tiles" uri="http://struts.apache.org/tags-tiles" %>
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title><tiles:getAsString name="title" /></title>
</head>
<body>
<table width="100%" border="0" cellspacing="0" cellpadding="0">
   <tr><td><tiles:insert name="header" /></td></tr>
   <tr><td><tiles:insert name="body" /></td></tr>
   <tr><td><tiles:insert name="footer" /></td></tr>
</table>
</body>
</html>
```

组合 tile 非常直观，用<tiles:insert>标签就可以嵌入一个 tile，用<tiles:getAsString>可以将一个属性写到页面中。在 template.jsp 模版里，一共插入了 3 个组件和 1 个属性值作为标题。

在 template.jsp 中，我们只定义了每个组件的名称，并没有指定具体的 jsp 文件。在 Tiles 中，每个页面的布局都被定义在 XML 配置文件中，我们将 Tiles 的配置文件命名为 tiles-defs.xml，并放到/web/WEB-INF/目录下，在这个配置文件中，我们定义了 login 和 welcome 两个完整的页面。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 1.3//EN"
      "http://struts.apache.org/dtds/tiles-config_1_3.dtd">
<tiles-definitions>
   <definition name="default" path="/template.jsp">
      <put name="title"  value="Default Title" />
      <put name="header" value="/header.jsp" />
      <put name="footer" value="/footer.jsp" />
   </definition>

   <definition name="login" extends="default" >
      <put name="title" value="Please login" />
      <put name="body"  value="/login.jsp" />
   </definition>

   <definition name="welcome" extends="default">
      <put name="title" value="Welcome!" />
      <put name="body"  value="/welcome.jsp" />
   </definition>
</tiles-definitions>
```

每个<definition>定义一个完整的页面，在 Tiles 中，页面是可以继承的，例如，上述 default 页面定义的 header 和 footer 分别为 header.jsp 和 footer.jsp，并设置页面的 title 属性为“Default Title”，login 页面就可以从 default 继承，并定义 body 为 login.jsp，然后覆盖了 title 属性。通过继承，就使得页面定义更加简洁。

最后一步是在 Spring 的 XML 配置文件中配置 Tiles 框架，并选择一个合适的 ViewResolver，让它能解析 Tiles 视图。

```
        <bean id="tilesConfig" class="org.springframework.web.servlet.view.
tiles.TilesConfigurer">
            <property name="definitions">
                <list>
                    <value>/WEB-INF/tiles-defs.xml</value>
                </list>
            </property>
        </bean>

        <bean id="viewResolver" class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
            <property name="viewClass" value="org.springframework.web.servlet.view.
tiles.TilesView" />
        </bean>
```

TilesConfigurer 只需要指定 Tiles 配置文件就可以自动地配置好 Tiles，为了让视图解析器能识别 TilesView，使用 InternalResourceViewResolver 并将 TilesView 绑定即可。

LoginController 和 WelcomeController 的编写和前面的完全相同，在 LoginController 中，返回的视图名称是“login”，而 WelcomeController 返回的视图名称是“welcome”，细心的读者可能已经发现了，视图解析器的任务就是在 Tiles 配置文件中查找对应名称的页面，然后根据页面的定义，由 TilesView 将整个页面渲染出来。两个页面的效果如图 7-63 和图 7-64 所示。

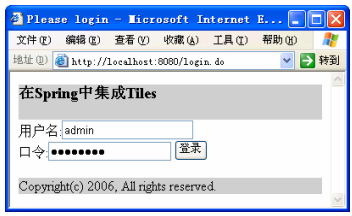


图 7-63

图 7-64

7.6.4 集成JSF

JSF 是 JavaServer Faces 的缩写，JSF 是 JavaEE 中构建 Web 应用程序的又一个全新

的 MVC 框架。与其他常见的 MVC 框架相比，JSF 提供了一种以组件为中心的方式来开发 Web 应用程序。JSF 的目标是提供一种类似 ASP.Net 的可视化 Web 组件，开发人员只需要将已有的 JSF 组件拖放到页面上，就可以迅速建立一个 Web 应用程序，而不必从头开始编写 UI 界面。因此，JSF 的易用性很大程度上取决于一个简单、高效的可视化开发环境。

和传统的 MVC 框架（如 Struts）相比，JSF 更像是一个 Web 版本的 Swing 应用程序。传统的 MVC 框架的 View 是以页面为中心的，而 JSF 的 View 则是一系列可复用的 UI 组件。因此，JSF 应用程序的生命周期更为复杂。一般来说，JSF 处理用户请求的流程如图 7-65 所示。

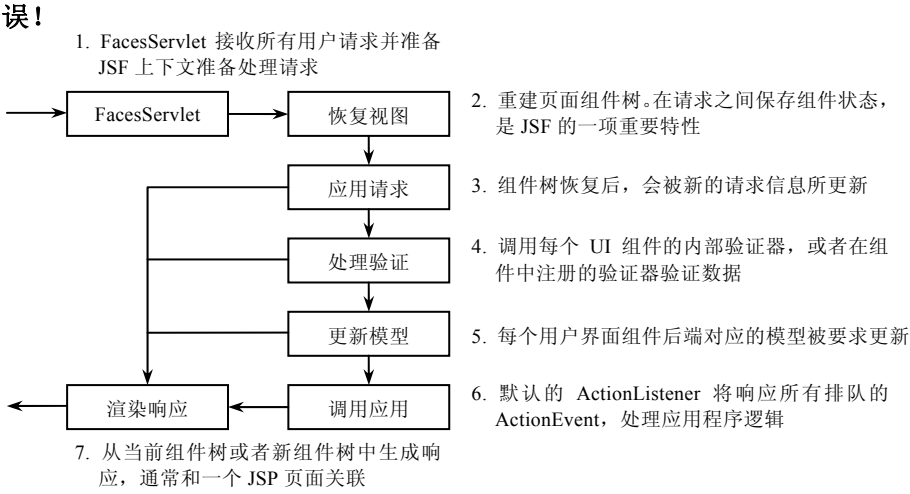


图 7-65

除了恢复视图和渲染响应是必须的之外，其他步骤都可以视情况跳过。例如，在验证失败后，就直接跳到渲染响应，将错误信息展示给用户。

JSF 的真正威力在于它的 UI 界面组件。与 ASP.NET 类似，JSF 的 UI 组件使开发人员能够使用已创建好的 UI 组件（如 JSF 内置的 UI 组件）来创建 Web 页面，而非完全从头创建，从而提供了前所未有的开发效率。JSF 的 UI 组件可以是简单到只是显示文本的 outputLabel，或者复杂到可以表示来自数据库表的表格。

此外，JSF 也使用类似 Spring IoC 容器的方式来管理 Model，即与 UI 组件绑定的 Bean，在 JSF 中称为 Managed-Bean，并且也支持依赖注入。Managed-Bean 的生命周期可以是 request、session 和 application，分别对应一次请求、一次会话和整个 Web 应用程序生存期。

在开发 JSF 应用之前，我们需要首先准备开发环境。由于 JSF 也是一个规范，不同

的厂商都可以有自己的实现。这里我们采用的是 JSF 1.1 规范（JSR 127），因为 JSF 1.1 仅需要 Servlet 2.3 规范的支持，在大多数 Web 服务器上都能正常运行，而最新的 JSF 1.2 则要求 Servlet 2.5 规范，目前除了少数服务器外，大多数服务器还无法支持。

我们还需要一个 JSF 1.1 的实现。SUN 给出了一个 JSF 1.1 的参考实现，可以用于开发，以保证我们的 JSF 应用程序将来可以移植到其他的 JSF 实现。Apache 也提供了一个 MyFaces 的 JSF 实现，读者可以参考 Apache 的官方站点获取详细信息，本书不对此做更多讨论。

下面的例子改自 IBM developerWorks 站点的一个 JSF 应用，它允许用户输入自己的姓名和电子邮件地址来订阅新闻。从 http://java.sun.com/javaee/javaserverfaces/download.html 下载 JSF 1.1 的参考实现并解压，然后在 Eclipse 中建立一个 Spring_JSF 工程，首先实现一个基于 JSF 的 Web 应用程序，将相关 jar 包放入/WEB-INF/lib 目录，工程结构如图 7-66 所示。

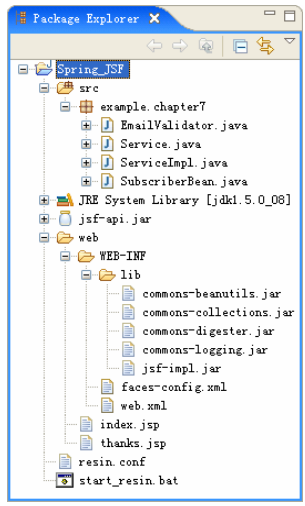


图 7-66

编译工程只需要引用 jsf-api.jar。几个主要的接口和类如下。

Service 接口是唯一的业务逻辑接口，它仅定义了一个 subscribe 方法。

```
public interface Service {
    void subscribe(SubscriberBean subscriber);
}
```

其实现类 ServiceImpl 仅仅简单地打印出用户的订阅信息。

```
public class ServiceImpl implements Service {
    public void subscribe(SubscriberBean subscriber) {
```

```
        System.out.println("User \"" + subscriber.getName()
                + "\" with email \"" + subscriber.getEmail()
                + "\" subscribed successfully with preferred language "
                + subscriber.getLanguage());
    }
}
```

SubscriberBean 是一个与前端 UI 绑定的 Session 范围的 Managed-Bean，其作用范围是 Session，在 SubscriberBean 中还定义了 submit()方法来处理 JSF 的 Action，因此，在 SubscriberBean 中必须要注入一个 Service 对象，才能完成实际业务方法的调用。

```
public class SubscriberBean {
    private String name;
    private String email;
    private String language;

    private Service service = new ServiceImpl();

    public void setService(Service service) {
        this.service = service;
    }

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

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public String getLanguage() { return language; }
    public void setLanguage(String language) { this.language = language; }

    public String submit() {
        // 处理订阅请求:
        service.subscribe(this);
        return "success";
    }
}
```

EmailValidator 是一个自定义的 JSF 验证器，读者可以参考 JSF 相关文档获得有关验证器的使用方法，这里仅给出实现。

```
public class EmailValidator implements Validator {
  public void validate(FacesContext context, UIComponent component, Object value)
          throws ValidatorException {
        String email = ((String)value).trim();
```

```
            if(!email.matches("[a-zA-Z0-9][\\w\\.\\-]*@[a-zA-Z0-9][\\w\\.\\-]
*\\.[a-zA-Z][a-zA-Z\\.]*")) {
                throw new ValidatorException(
                        new FacesMessage("Invalid email address"));
            }
        }
    }
```

下面我们需要配置 JSF，首先，在 web.xml 中声明 JSF 相关 Listener 和 FacesServlet。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
    </context-param>

    <listener>
        <listener-class>
            com.sun.faces.config.ConfigureListener
        </listener-class>
    </listener>

    <!-- Faces Servlet -->
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
</web-app>
```

FacesServlet 是整个 JSF 应用程序的前端入口，负责接收所有的用户请求。然后，需要编写一个默认的 faces-config.xml 配置文件，告诉 JSF 所有的配置信息，包括自定义的验证器、导航规则、所有的 Managed-Bean 和这些 Bean 之间的依赖关系。将

faces-config.xml 放到/WEB-INF/目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
    <!-- 声明自定义的 Validator -->
    <validator>
        <validator-id>emailValidator</validator-id>
        <validator-class>example.chapter7.EmailValidator</validator-class>
    </validator>

    <!-- 声明所有的 Managed Bean -->
    <managed-bean>
        <managed-bean-name>service</managed-bean-name>
       <managed-bean-class>example.chapter7.ServiceImpl</managed-bean-class>
       <managed-bean-scope>application</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>subscriber</managed-bean-name>
      <managed-bean-class>example.chapter7.SubscriberBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>service</property-name>
            <value>#{service}</value>
        </managed-property>
    </managed-bean>

    <!-- 定义导航规则 -->
    <navigation-rule>
        <from-view-id>/index.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/thanks.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

下一步是编写两个 JSP 页面，使用 JSF 标准的标签库，读者可以发现 JSF 使用的表达式和 Velocity 非常类似，不过将“$”改为了“#”。index.jsp 负责接受用户输入并验证表单。

```
<%@page contentType="text/html;charset=UTF-8" %>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<head>
<title>Subscribe Form</title>
</head>
<body>
    <f:view>
        <h:form>
            <h4>Subscribe</h4>
            Your name:
             <h:inputText id="id_name" value="#{subscriber.name}" required="true">
                 <f:validateLength minimum="3" maximum="20" />
            </h:inputText>
            <h:message for="id_name" />
            <br/>
            Your email:
           <h:inputText id="id_email" value="#{subscriber.email}" required="true">
                 <f:validator validatorId="emailValidator" />
            </h:inputText>
            <h:message for="id_email" />
            <br/>
            Preferred Language:
            <h:selectOneMenu value="#{subscriber.language}" required="true">
                <f:selectItem itemLabel="English" itemValue="English" />
                <f:selectItem itemLabel="Chinese" itemValue="Chinese" />
            </h:selectOneMenu>
            <br/>
            <h:commandButton type="submit" value="Submit" action="#{subscriber.
submit}" />
        </h:form>
    </f:view>
</body>
</html>
```

thanks.jsp 用于提示用户订阅成功。

```
<%@page contentType="text/html;charset=UTF-8" %>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<head><title>Thank you</title></head>
<body>
    <f:view>
        <h3>
            Thank you, <h:outputText value="#{subscriber.name}"/>!
```

```
            </h3>
            A confirm mail has been sent to your mail box <h:outputText
value="#{subscriber.email}" />.
        </f:view>
    </body>
    </html>
```

编译工程后，启动 Resin 服务器，就可以直接在浏览器中输入 http://localhost:8080/index.faces，其执行效果如图 7-66 和图 7-67 所示。

图 7-66

图 7-67

现在，我们来考虑如何将 JSF 集成到 Spring 框架中。我们注意到，在上面的配置中，Service 对象是由 JSF 定义为全局变量并注入到 SubscriberBean 中的，我们更希望能从 Spring 的 IoC 容器中获得 Service 对象，然后将其注入到 JSF 的 SubscriberBean 中，这样使得 JSF 仅作为 Web 层与用户打交道，真正的中间层逻辑和后端持久层的 Bean 都交给 Spring 管理，使整个应用程序的层次更加清晰。

在 Spring 中集成 JSF 非常简单，其关键在于声明 Spring 提供的一个 DelegatingVariableResolver，在 faces-config.xml 中添加下列内容。

```
<faces-config>
    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
    </application>
    ...
</faces-config>
```

我们看看 JSF 是如何实现依赖注入的。在前面的 faces-config.xml 中，SubscriberBean 需要注入一个 Service 对象，我们是这么注入的。

```
<managed-property>
    <property-name>service</property-name>
    <value>#{service}</value>
</managed-property>
```

JSF 根据名称#{service}去查找名称为 service 的 Managed-Bean，然后将其注入到 SubscriberBean 中。如果我们声明了 Spring 的 DelegatingVariableResolver，则由 DelegatingVariableResolver 负责查找这个名称为 service 的 Bean，DelegatingVariable Resolver 就有机会从 Spring 的 IoC 容器中获得名称为 service 的 Bean，然后交给 JSF，JSF 再将其注入到 SubscriberBean 中，图 7-68 很好地说明了 DelegatingVariableResolver 实现的功能。

错误！

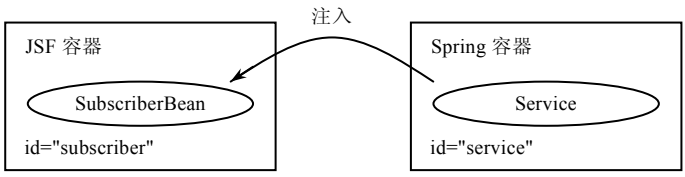


图 7-68

然后，删除 faces-config.xml 中定义的 Service Bean，因为这个 Bean 已被放入 Spring 容器中管理。在 Spring 的 applicationContext.xml 中定义这个 Service Bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
>
   <bean id="service" class="example.chapter7.ServiceImpl" />
</beans>
```

最后一步是在 web.xml 中通过声明 Spring 提供的 ContextLoaderListener 来启动 Spring 容器，注意：该 Listener 应当在其他 Listener 之前定义，以保证 Spring 容器首先被启动。

```
<listener>
   <listener-class>
      org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

集成后的整个工程结构如图 7-69 所示。

无需编译，重新启动 Resin 服务器后，可以看到和 Spring 集成的 JSF 应用的运行效果与前面的 JSF 应用一致，不同之处在于 Service 组件从 JSF 容器移动到 Spring 容器内了。

需要注意的几点是，DelegatingVariableResolver 将首先试图查找 faces-config.xml 中定义的 Managed-Bean，找不到才在 Spring 的 IoC 容器中查找。因此，Spring 容器中定义的 Bean 和 JSF 中定义的 Managed-Bean 千万不要有相同的名称，以免造成冲突。

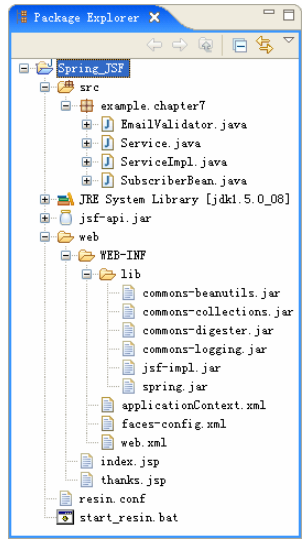


图 7-69

另一个值得注意的地方是，作为中间逻辑组件和后端持久化组件的 Bean 非常适合放在 Spring 的 IoC 容器中，以便获得 AOP、声明式事务等强大的支持。但是，在 JSF 中作为 UI 组件 Model 的 Managed-Bean 和 UI 组件的耦合程度较高，仍适合由 JSF 管理其生命周期，不推荐放在 Spring 的 IoC 容器中。这一点和 Struts 及 WebWork2 的 Action 不一样，后两者的 Action 要么是唯一实例，要么是对应每个请求的新实例，其生命周期远没有 JSF 的 Managed-Bean 那么复杂。

有些时候，需要在 JSF 中手动获取 Spring 容器的 Bean，读者可能已经想到了，由于 Spring 的 IoC 容器是绑定在 ServletContext 上的，因此可以首先通过 FacesContext 的 getExternalContext() 方法获得 ExternalContext 实例，再通过 ExternalContext 的 getApplicationMap() 方法获得所有 Application 级别的 Object，再通过查找名称为 WebApplicationContext.ROOT_ WEB_APPLICATION_CONTEXT_ATTRIBUTE 的对象就可以获得 Spring 的 IoC 容器的实例，一旦获得了 Spring 容器的实例，就可以获取所有的 Bean 实例。

事实上，Spring 已经提供了一个辅助类 FacesContextUtils 来获取 ApplicationContext 实例。

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(
    FacesContext.getCurrentInstance());
```

这样，在 JSF 的代码中，任何时候如果需要获得 ApplicationContext 的实例，就可以按照上述方法调用。

7.7 小结

本章我们介绍了以 Spring 为核心的 Web 应用程序的开发。我们首先介绍了 JavaEE Web 应用程序的基础知识，并给出了几个非常有用的 Filter 组件，这些组件都可以直接在实际的 Web 应用程序中应用。然后，我们介绍了 MVC 模式的原理，以及如何在 Web 应用程序中实现。我们甚至还手动实现了一个最简单的 MVC 框架，读者从这个例子中应该对 MVC 有了一个较为全面的认识。

由于 Spring MVC 框架的强大功能，并且能与 Spring 的 IoC 容器实现完美集成，因此，构建基于 Spring 框架的 Web 应用程序时，应当首先考虑采用 Spring 的 MVC 框架。我们详细介绍了 Spring MVC 框架的主要接口和常用的 Controller。我们还介绍了 Spring MVC 框架提供的一些高级功能，如输出二进制文件、重定向、文件上传、拦截请求、异常处理等。

由于 Spring 良好的集成能力，采用其他视图技术甚至混合使用多种视图技术也是完全可以的，因此，我们还介绍了除 JSP 外常用的视图技术（如 Velocity 和 Freemarker），并给出了一个混合使用多种视图技术的方案。

对于采用 Struts 等框架的已有的 Web 应用程序，如何与 Spring 集成从而获得结构清晰的多层应用程序是一个大问题。我们详细介绍了如何与 Struts、WebWork、Tiles 和 JSF 集成的策略及可能的多种方式，读者在实际项目中应根据需要选择最合适的方案。

在第 8 章中，我们将介绍 Spring 提供的多种远程访问服务，以便将应用程序的某些业务接口暴露给远程客户端，从而实现分布式应用。

第 8 章

Spring提供的远程访问

如果需要将应用程序的某些功能暴露给远程客户端访问，就需要某种远程调用机制。除了 Java 内置的标准的 RMI 远程调用机制外，还有众多可供选用的远程调用方案。Spring 框架提供了对各种远程访问的良好支持，包括 RMI、HTTP 远程调用和 Web 服务。在本章中，我们将详细介绍各种远程调用方式。

8.1 RMI远程调用

RMI 是 Java 标准的远程方法调用接口，即 Rmote Method Invocation 的缩写。RMI 从 JDK 1.1 就引入了，它基于 Java 的序列化机制实现远程方法调用。下面，我们分别来看看手动使用 RMI 和在 Spring 中使用 RMI 的异同。

8.1.1 实现RMI

让我们来看一个常见的例子。假定应用程序已经设计并实现了一个用户管理的模块，允许创建新用户并让用户登录，定义 Rmi 工程结构如图 8-1 所示。

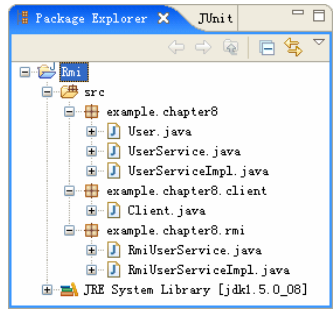


图 8-1

User 对象代表一个用户。

```
public class User implements java.io.Serializable {
    private String username;
    private String password;
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    public String getUsername() { return username; }
    public String getPassword() { return password; }
}
```

UserService 定义了用户服务的接口。

```
public interface UserService {
    User login(String username, String password);
    void create(String username, String password);
}
```

并且有一个默认的实现类。

```
public class UserServiceImpl implements UserService {
    private Map<String, String> users = new HashMap<String, String>();
    public void create(String username, String password) {
        if(username==null || password==null)
            throw new IllegalArgumentException("Invalid args.");
        if(users.get(username)!=null)
            throw new RuntimeException("User exist!");
        users.put(username, password);
    }
    public User login(String username, String password) {
        if(username==null || password==null)
            throw new IllegalArgumentException("Invalid args.");
        if(password.equals(users.get(username)))
            return new User(username, password);
        throw new RuntimeException("Login failed.");
    }
}
```

这个模块在应用程序中运行良好，现在，我们希望将它暴露为远程服务，以便其他远程应用程序也能通过这个模块来管理用户。为了以 RMI 的方式来实现这个远程服务，我们先看看实现 RMI 的必要条件。

（1）服务接口必须从 Remote 派生，并且在每个远程方法中抛出 RemoteException，很明显 UserService 不符合这个条件。

（2）实现类除了实现服务接口外，还必须从 UnicastRemoteObject 派生，因此，UserServiceImpl 也不符合这个条件。

（3）所有方法的参数和返回值都必须是基本类型，或者实现了 Serializable 接口，或者实现了 Remote 接口，幸运的是，对 User 对象添加一个 Serializable 接口是非常容易的。

为了实现 RMI 远程调用，按照传统的 RMI 调用方式，我们不得不修改 UserService 接口。然而，如果让 UserService 接口的每个方法都抛出 RemoteException，由于 RemoteException 是 CheckedException，势必造成应用程序其他依赖 UserService 接口的代码无法编译通过，其代价是巨大的。因此，只好考虑定义另一个 RmiUserService 接口。

```
public interface RmiUserService extends Remote {
```

```
        User login(String username, String password) throws RemoteException;
        void create(String username, String password) throws RemoteException;
    }
```

然后，为 RmiUserService 接口再编写一个实现类。

```
    public class RmiUserServiceImpl extends UnicastRemoteObject implements
RmiUserService {
        UserService service = new UserServiceImpl();
        // 必须定义构造方法，因为 UnicastRemoteObject 的构造方法会抛出 RemoteException:
        public RmiUserServiceImpl() throws RemoteException {}
        public void create(String username, String password) throws RemoteException {
            service.create(username, password);
        }
        public User login(String username, String password) throws RemoteException {
            return service.login(username, password);
        }
        // 启动 RMI 服务:
        public static void main(String[] args) throws Exception {
            LocateRegistry.createRegistry(1099);
            Naming.bind("rmi://localhost:1099/UserService", new RmiUser
ServiceImpl());
        }
    }
```

现在，我们终于可以编写客户端来调用这个 RmiUserService 了。编写一个 Client 类来调用 RMI 服务。

```
    package example.chapter8.client;
    import java.rmi.Naming;
    import example.chapter8.rmi.RmiUserService;
    public class Client {
        public static void main(String[] args) throws Exception {
            RmiUserService service = (RmiUserService)Naming.lookup("rmi://
localhost:1099/UserService");
            service.create("new_user", "a_test");
            System.out.println(service.login("new_user", "a_test"));
        }
    }
```

首先启动 RMI 服务，打开 Windows 的控制台窗口，切换到 bin 目录，输入：

```
    D:\workspace\Rmi\bin>java -cp . example.chapter8.rmi.RmiUserServiceImpl
```

然后打开另一个控制台窗口，切换到 bin 目录，输入：

```
D:\workspace\Rmi\bin>java -cp . example.chapter8.client.Client
```

控制台窗口立刻输出异常。

```
Exception in thread "main" java.lang.ClassCastException: $Proxy0
```

原因是客户端在调用 RMI 服务时，还必须首先要有 RMI 服务的“存根（Stub）”，也就是支持类，这些支持类是由 rmic 编译出来的。没有办法，只好首先编译出 Stub 类，输入：

```
D:\workspace\Rmi\bin>rmic example.chapter8.rmi.RmiUserServiceImpl
```

然后，再次运行客户端，这一次，我们终于成功地调用了一个 RMI 远程服务。

```
D:\workspace\Rmi\bin>java example.chapter8.client.Client
example.chapter8.User@17172ea
```

再运行一次客户端，由于重复添加用户，因此得到一个 RuntimeException。

```
D:\workspace\Rmi\bin>java example.chapter8.client.Client
Exception in thread "main" java.lang.RuntimeException: User exist!
        at example.chapter8.UserServiceImpl.create(UserServiceImpl.java:14)
        at example.chapter8.rmi.RmiUserServiceImpl.create(RmiUserServiceImpl.
java:19)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethod
AccessorImpl. java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethod
AccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:585)
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.
java:294)
        ...
```

这说明我们的 RMI 服务逻辑是正确的。

从上面的例子可以看到，为了实现一个简单的 RMI 远程调用，我们不得不遵循 RMI 规范，定义符合 RMI 调用的接口和实现类，还必须手动编译出 Stub 类供客户端使用。这实在是太麻烦了，为了避免这么复杂的步骤，我们看看用 Spring 能否简化 RMI 的调用。

8.1.2 在Spring中输出RMI

我们新建一个 Rmi_Spring 工程，复制 User、UserService 和 UserServiceImpl 这 3 个

类，然后我们看看如何在 Spring 中将 UserService 作为一个 RMI 服务输出。

Rmi_Spring 工程结构如图 8-2 所示。

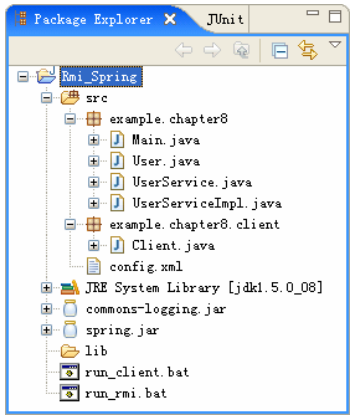


图 8-2

令我们非常兴奋的是，不用编写一行代码，就可以直接在 Spring 的 XML 配置文件中将 UserService 作为 RMI 服务输出。

```
    <bean id="userService" class="example.chapter8.UserServiceImpl" />

    <bean id="rmiService" class="org.springframework.remoting.rmi.RmiService
Exporter">
        <property name="serviceName" value="UserService" />
        <property name="service" ref="userService" />
        <property name="serviceInterface" value="example.chapter8.UserService" />
        <property name="registryPort" value="1099" />
    </bean>
```

唯一的代码是编写 main()方法启动 Spring 容器。

```
    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("config.xml");
    }
```

编写客户端代码稍微麻烦一点，需要用到 Spring 框架提供的 RmiProxyFactoryBean。

```
    public class Client {
        public static void main(String[] args) throws Exception {
            RmiProxyFactoryBean factory = new RmiProxyFactoryBean();
            factory.setServiceInterface(UserService.class);
            factory.setServiceUrl("rmi://localhost:1099/UserService");
            factory.afterPropertiesSet();
            // 获取 UserService 服务接口:
```

```
            UserService userService = (UserService)factory.getObject();
            // 调用 UserService:
            userService.create("test", "password");
            System.out.println(userService.login("test", "password"));
            try {
                userService.login("test", "bad-password");
            }
            catch(Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
```

由于启动 RMI 服务和 RMI 客户端都需要 Spring 的 jar 包和 commons-logging.jar，为了避免在控制台窗口输入很长的命令，我们创建了两个 BAT 文件分别用于启动 Spring 的 RMI 服务和调用 RMI 的客户端。首先，运行 run_rmi.bat。

```
        D:\workspace\Rmi_Spring>java -cp bin\;lib\spring.jar;lib\commons-logging.
jar example.chapter8.Main
        ...
        INFO: Looking for RMI registry at port '1099'
        Nov 8, 2006 9:17:42 PM org.springframework.remoting.rmi.RmiServiceExporter
getRe
        gistry
        WARNING: Could not detect RMI registry - creating new one
        Nov 8, 2006 9:17:42 PM org.springframework.aop.framework.DefaultAopProxy
Factory
        <clinit>
        INFO: CGLIB2 not available: proxyTargetClass feature disabled
        Nov 8, 2006 9:17:42 PM org.springframework.remoting.rmi.RmiServiceExporter
prepa
        re
        INFO: Binding RMI service 'UserService' to registry at port '1099'
```

屏幕显示出绑定 RMI 服务成功后，在运行 run_client.bat。

```
        D:\workspace\Rmi_Spring>java -cp bin\;lib\spring.jar;lib\com
        mons-logging.jar example.chapter8.client.Client
        Nov 8, 2006 9:15:18 PM org.springframework.remoting.rmi.RmiClientInterceptor
pre
        pare
        INFO: RMI stub [rmi://localhost:1099/UserService] is an RMI invoker
        Nov 8, 2006 9:15:18 PM org.springframework.aop.framework.DefaultAopProxy
Factory
        <clinit>
```

```
INFO: CGLIB2 not available: proxyTargetClass feature disabled
Nov 8, 2006 9:15:18 PM org.springframework.core.CollectionFactory <clinit>
INFO: JDK 1.4+ collections available
example.chapter8.User@14b7453
Login failed.
```

令人非常兴奋的是，整个过程没有引入任何 RMI 必须的接口，虽然客户端调用代码稍显复杂，并且需要 Spring 的 jar 包支持，但是，由于不需要手动调用 rmic 生成 Stub 类，整个调用过程被大大简化了。

8.1.3 访问RMI

事实上，如果客户端也在 Spring 容器中启动，则完全可以在 XML 配置文件中定义 UserService 并直接使用。

```
    <bean id="userService" class="org.springframework.remoting.rmi.RmiProxy
FactoryBean">
        <property name="serviceUrl" value="rmi://localhost:1099/UserService" />
        <property name="serviceInterface" value="example.chapter8.UserService" />
    </bean>
```

这样，客户端完全不必编写获取远程服务接口的代码，就可以直接将 userService 注入到需要的组件中去。

对于传统的 RMI 调用，服务端发布了 RMI 服务后，还必须通过 rmic 编译出存根供客户端使用，整个调用过程如图 8-3 所示。

错误！

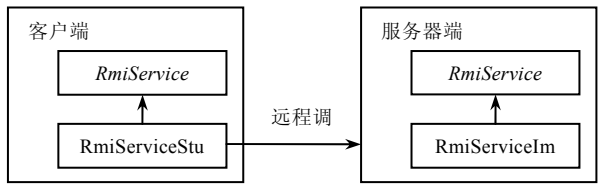


图 8-3

Stub 的作用便是将客户端对 RmiService 接口的调用转变成实际的远程调用，在 Spring 中，由于使用了 JDK 动态代理，因此可以在运行期动态实现 RmiServiceStub，免去了使用 rmic 编译存根的麻烦。

如果 RMI 服务不是由 Spring 包装后启动的，而是直接以 Remote 接口实现的，例如，在 Rmi 工程中的 RmiUserServiceImpl 类实现的 RMI 服务，能否在 Spring 中调用呢？答案是肯定的，只需要将 Client 代码修改为实际的 RMI 服务接口。

```
public class Client {
    public static void main(String[] args) throws Exception {
```

```
        RmiProxyFactoryBean factory = new RmiProxyFactoryBean();
        // 服务接口由 UserService 变为 RmiUserService:
        factory.setServiceInterface(RmiUserService.class);
        factory.setServiceUrl("rmi://localhost:1099/UserService");
        factory.afterPropertiesSet();
        RmiUserService userService = (RmiUserService)factory.getObject();
        userService.create("test", "password");
        System.out.println(userService.login("test", "password"));
        try {
            userService.login("test", "bad-password");
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

将 Rmi 工程中的 RmiUserService 和 RmiUserServiceImpl 复制过来，编译工程，然后用 rmic 编译 RmiUserServiceImpl 生成 Stub 存根，运行 Client。

```
Nov 8, 2006 10:33:55 PM org.springframework.remoting.rmi.RmiClientInterceptor
prepare
INFO: Using service interface [example.chapter8.rmi.RmiUserService] for RMI stub
 [rmi://localhost:1099/UserService] - directly implemented
Nov 8, 2006 10:33:55 PM org.springframework.aop.framework.DefaultAopProxy
Factory
 <clinit>
INFO: CGLIB2 not available: proxyTargetClass feature disabled
Nov 8, 2006 10:33:55 PM org.springframework.core.CollectionFactory <clinit>
INFO: JDK 1.4+ collections available
example.chapter8.User@18a992f
Login failed.
```

可以看到 Spring 在实现 RMI 远程调用时，会自动检查是否有 Stub 存根，如果找到了 Stub，就会直接使用预编译好的 Stub 类，并返回一个 RmiUserService 接口对象；如果没有找到，就使用 JDK 动态代理动态实现 Stub 的功能。

8.2 HTTP调用

RMI 虽然是 Java 标准的远程调用模式，但是 RMI 最大的缺点是使用特定的 JRMP（Java Remote Method Protocol，Java 远程方法协议）二进制协议，很难穿透防火墙，仅适合在企业内部网中使用。如果需要跨防火墙调用，则 HTTP 协议几乎是唯一的选择，

因此我们需要一种以 HTTP 协议为基础的远程调用。Spring 支持 3 种基于 HTTP 协议的远程调用，以及我们将要在 8.3 节中讨论的 Web 服务。在本节中，我们先讨论 Spring 支持的 3 种 HTTP 远程调用。

Hessian 和 Burlap 这两种基于 HTTP 的远程调用协议是由 Caucho 开发的，并且集成在 Resin 服务器内部，可以直接使用。Hessian 是一个二进制协议，而 Burlap 是一个基于 XML 的协议，两者没有本质的不同。区别在于，Hessian 由于使用了二进制协议，所以效率更高，但是很难被读懂，而 Burlap 使用 XML 作为载体，其传输内容很容易被开发人员所理解，并且，从理论上讲，任何其他语言只要能解析 XML，就可以使用 Burlap 和 Java 程序通信。不过，这两种协议由于是私有协议，并没有成为标准，因此仅适合 Java 系统内跨越防火墙进行调用。

除 Hessian 和 Burlap 外，Spring 本身还提供了一个 HTTP 远程调用协议，我们暂且称之为 HTTP Invoker。和 Hessian、Burlap 使用自定义的序列化机制有所不同，HTTP Invoker 使用 Java 标准的序列化机制，通过 HTTP 协议来实现远程调用，因此这种方式也仅适用于 Java 应用程序间的通信。

在 Spring 中，使用 Hessian、Burlap 和 HTTP Invoker 非常容易，我们甚至根本无须了解其 API 细节就可以直接使用它们。下面的 HttpCall_Server 工程演示了在 Spring 环境下将服务接口暴露成为远程服务的方法。由于 Hessian、Burlap 和 HTTP Invoker 都需要通过 Servlet 来实现服务接口，因此，该工程是一个基于 Spring MVC 框架的 Web 应用程序，如图 8-4 所示。

我们设计了一个非常简单的接口 TimeService，用于返回当前时间。

```
public interface TimeService {
    String getTime();
}
```

TimeServiceImpl 是实现类，其实现非常简单。

```
public class TimeServiceImpl implements TimeService {
    public String getTime() {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
                .format(new Date());
    }
}
```

现在，我们需要将 TimeService 暴露为远程接口，以便客户端可以通过 HTTP 实现远程调用。在 Spring 中，通过 Hessian、Burlap 和 HTTP Invoker 实现该服务非常简单，Spring 已经提供了现成的 HttpRequestHandler，只需要告诉 Spring 业务接口和实现类，就立刻可以将其输出为远程接口。在 Spring 的 XML 配置文件 dispatcher-servlet.xml 中定义如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="timeService" class="example.chapter8.TimeServiceImpl" />

    <bean name="/HessianService" class="org.springframework.remoting.caucho.
HessianServiceExporter">
        <property name="service" ref="timeService" />
        <property name="serviceInterface" value="example.chapter8. TimeService" />
    </bean>

    <bean name="/BurlapService" class="org.springframework.remoting.caucho.
BurlapServiceExporter">
        <property name="service" ref="timeService" />
        <property name="serviceInterface" value="example.chapter8.imeService" />
    </bean>

    <bean name="/SpringHttpService" class="org.springframework.remoting.
httpinvoker.HttpInvokerServiceExporter">
        <property name="service" ref="timeService" />
        <property name="serviceInterface" value="example.chapter8. TimeService" />
    </bean>
</beans>
```

以 Hessian 为例，只需首先定义好提供服务的 TimeService Bean，然后将 TimeService 接口及其实现类注入到 HessianServiceExporter，就立刻实现了一个 Hessian 远程调用服务。Burlap 和 HTTP Invoker 的配置完全相同。我们甚至在编译器没有用到任何额外的 jar 包时就实现了 3 个远程调用接口。

下一步只需要在 web.xml 中配置好 Spring 的 DispatcherServlet。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>

    <servlet-mapping>
```

```
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/remote/*</url-pattern>
    </servlet-mapping>
</web-app>
```

我们设置 URL 映射为/remote/*，由于默认使用 BeanNameUrlHandlerMapping，因此启动服务器后，我们通过 Hessian、Burlap 和 HTTP Invoker 输出的 TimeService 的 URL 地址分别为 http://localhost:8080/remote/HessianService、http://localhost:8080/remote/BurlapService 和 http://localhost:8080/remote/SpringHttpService。启动 Resin 服务器后，直接从浏览器访问上述地址会显示出错页面，因为我们没有传入必要的参数。另一个需要注意的地方是，由于 Hessian 和 Burlap 是集成在 Resin 服务器中的，因此不需要额外的 jar 包就可以直接运行。如果使用其他 Web 服务器（例如，Tomcat），就必须把 hessian.jar 放到 Web 应用程序的/WEB-INF/lib 目录下。

下一步，我们需要编写客户端来实现远程调用上述服务。在 Eclipse 中新建一个 HttpCall_Client 工程，结构如图 8-5 所示。

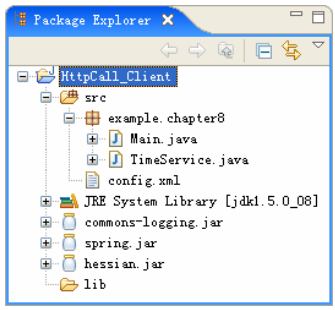


图 8-4

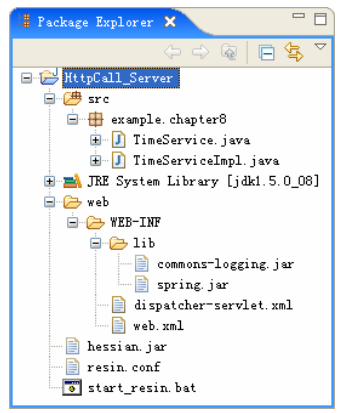


图 8-5

在 Spring 中调用 Hessian、Burlap 和 HTTP Invoker 也非常容易。首先，将 TimeService 接口从 HttpCall_Server 工程中复制过来，然后在 config.xml 中配置如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http: //www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="hessianService" class="org.springframework.remoting.caucho.
HessianProxyFactoryBean">
        <property name="serviceUrl" value="http://localhost:8080/remote/
HessianService" />
```

```
            <property name="serviceInterface" value="example.chapter8.TimeService"/>
        </bean>

        <bean id="burlapService" class="org.springframework.remoting.caucho.
BurlapProxyFactoryBean">
            <property name="serviceUrl" value="http://localhost:8080/remote/
BurlapService" />
            <property name="serviceInterface" value="example.chapter8.TimeService"/>
        </bean>

        <bean id="springHttpService" class="org.springframework.remoting.
httpinvoker.HttpInvokerProxyFactoryBean">
            <property name="serviceUrl" value="http://localhost:8080/remote/
SpringHttpService" />
            <property name="serviceInterface" value="example.chapter8.TimeService"/>
        </bean>
    </beans>
```

以 Hessian 为例，为了获得远程接口，只需要将服务器端地址和接口类注入到 HessianProxyFactoryBean 中，其返回的即是具有 TimeService 接口的远程对象，客户端只需要调用该对象即可，甚至不知道该对象到底是否是一个远程对象。Burlap 和 HTTP Invoker 的配置和 Hessian 完全相同。

为了让读者更直观地看到远程调用的效果，我们做了一个 Swing 界面，允许用户选择使用哪个远程对象。先启动 HttpCall_Server 工程，然后运行 HttpCall_Client 的 Main，效果如图 8-6 和图 8-7 所示。

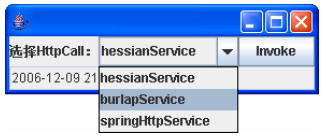


图 8-6

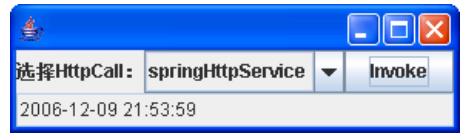


图 8-7

以上 3 种 HTTP 调用都可以跨防火墙，不过，由于使用的都是私有协议，因此只能用于 Java 应用程序之间的远程调用。如果希望和异构平台实现远程调用，就必须使用标准的 Web 服务（Web Services）。

8.3 Web服务

Web 服务（Web Service）是近年来炙手可热的技术，从本质上说，Web 服务就是一种能够实现跨平台远程调用的技术标准，即应用程序通过向外界暴露一个可以通过 Web

来远程调用的 API 接口，客户端就可以调用这个服务接口来实现相应的功能。例如，Amazon 站点提供了书籍查询的 Web 服务，因此，所有的应用程序都可以通过该 Web 服务来查找 Amazon 站点上相关书籍的信息。

可以把 Web 服务看成是基于因特网的远程调用，有了 Web 服务，Web 网站和 Web 网站之间就不再是简单地用超链接连起来，而是可以互相调用的 Web 应用程序。

例如，在没有 Web 服务前，如果需要在我们自己的网站上集成 Google 搜索功能，唯一的办法是通过 HTML 将用户导向到 Google 的站点，而有了 Web 服务后，网站可以通过调用 Google 的 Web 服务获得搜索结果，然后将结果以任意的方式展示给用户，对于用户而言，完全不知道该搜索结果实际上是从 Google 得到的。

Web 服务是一种面向服务的架构技术，由于 XML 是一种平台无关的描述语言，因此，在 Web 服务中，XML 被用来描述如何传递数据。这样，调用双方就可以是任意的异构平台，例如，在.NET 应用程序中调用 Java 应用发布的 Web 服务，或者在 Java 应用程序中调用.NET 应用程序发布的 Web 服务。

随着因特网的迅速发展，Web 服务也得到了越来越广泛的应用。目前，Web 服务大致可以应用在以下领域。

（1）面向企业的 Web 服务，包括企业内部的各种 ERP 系统和企业间合作伙伴的系统对接。由于大型企业内部或企业之间常常是异构平台，因此，采用 Web 服务来实现系统集成非常合适。

（2）面向消费的 Web 服务，尤其是 B2C 站点。通过 Web 服务，第三方桌面应用程序就可以为用户提供更方便的服务，例如，个人理财软件可以通过 Web 服务获得最新的股票价格。

（3）面向终端设备的 Web 服务。包括手机、PDA 等各种移动终端，可以通过 Web 服务轻松实现天气预报、酒店预订等服务。

使用 Web 服务带来的最大的好处是能实现跨平台的异构协作，由于 Web 服务使用了一系列标准的协议，因此，不同平台、不同语言之间都可以互相调用。图 8-8 显示了一个 Web 服务是如何在发布者和调用者之间实现的。

错误！

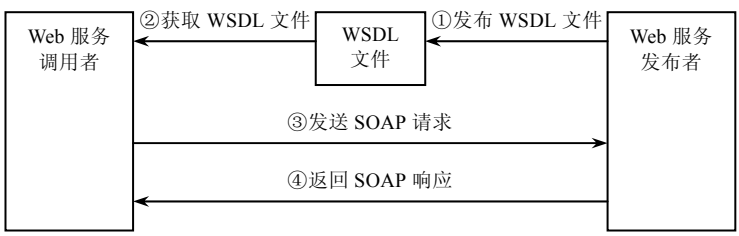


图 8-8

Web 服务的发布者首先必须提供一个 WSDL 文件，即 Web Services Description Languange（Web 服务描述语言），这个 XML 文件定义了调用 Web 服务的所有信息，包括所有的方法名称、参数类型、返回类型和数据类型的映射等。Web 服务的调用者只要获得了该 WSDL 文件，就可以根据 WSDL 文件通过相应的工具生成客户端支持类，因此，无论服务端以任何技术实现该 Web 服务，客户端都可以用任何编程语言来调用它。

JavaEE 平台从 1.3 版本开始就完整地支持 Web 服务，包括调用和发布 Web 服务两部分。为了让 Java 应用程序获得访问和发布 Web 服务的能力，SUN 定义了一系列相关的 API 标准，包括 SAAJ、JAXB、JAX-RPC 和最新的 JAX-WS。SAAJ（SOAP with Attachments API for Java）已经完整地实现了 SOAP 协议，可以直接使用 SAAJ 实现 Web 服务的底层传输。JAXB（Java Architecture for XML Binding）则实现了 XML 数据到 Java 类的绑定，通常我们不直接使用这两个 API，而是调用高层的 JAX-PRC（Java API for XML-Based RPC）和 JAX-WS（Java API for XML-Based Web Services）。JAX-WS 是 JavaEE 最新的 Web 服务标准，不过仍向下兼容 JAX-RPC。

和其他 Java 标准类似，为了在 Java 中实现 Web 服务，还必须获得一个具体的实现。Axis 和 XFire 都是实现了完整的 Web 服务协议的库，可以用于访问和发布 Web 服务。稍后我们会分别看到这两个库的使用方法。

下面我们首先介绍如何使用 Axis 访问 Amaon 站点提供的 Web 服务，然后介绍如何通过 XFire 对外发布我们自己的 Web 服务。

8.3.1 访问Amazon的Web服务

我们首先看看如何在 Java 应用程序中调用 Web 服务。以 Amazon 站点的 Web 服务为例，首先，我们需要获得 Amazon 站点的 Web 服务描述文档 WSDL，Amazon 提供了多个免费和收费的 Web 服务，可以在 Amazon 的网站上查看所有的 Web 服务信息。最常用的是免费的 AWSECommerceService 服务，可以搜索和查询各种商品的名称、价格和详细信息等。该 Web 服务的 WSDL 地址为 http://webservices.amazon.com/ AWSE CommerceService/2005-03-23/US/AWSECommerceService.wsdl。为了使用 Amazon 的 Web 服务，需要首先注册一个免费的 Amazon Web 服务账号，并获得相应的 Access Key。对于免费的 Web 服务，Amazon 对调用次数没有限制，但调用间隔必须大于 1 秒，这对于我们的示例代码来说是没有问题的，但是，如果需要频繁地调用，则应该考虑缓存调用结果。

让我们首先以手动方式来调用 Amazon 的 Web 服务，实现关键字搜索相关书籍的功能。

首先，在 Eclipse 中新建一个 CallAmazonWS 工程，然后添加以下必要的 jar 包到工程中：axis.jar、saaj.jar、wsdl4j.jar、jaxrpc.jar、commons-logging.jar 和 commons-discovery.jar。

其中，jaxrpc.jar 是 JavaEE 中 Web 服务的接口，我们编写代码时与之打交道的 API 正是 jaxrpc.jar 中定义的 Web 服务的 API。整个工程的结构如图 8-9 所示。

前面已经讲过，Web 服务是平台无关的，由于 WSDL 文件已经包含了足够的信息，在使用 Amazon 的 Web 服务之前，必须首先将 WSDL 文件映射为 Java 接口和类。为此，Axis 提供了一个 WSDL2Java 的命令来实现这个功能。在控制台窗口输入命令或者执行 wsdl2java.bat。

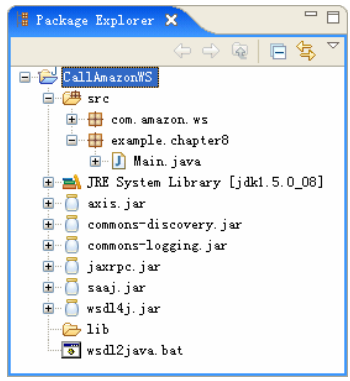


图 8-9

```
java -cp lib\axis.jar;lib\commons-logging.jar;lib\commons-discovery.jar;
lib\jaxrpc.jar;lib\saaj.jar;lib\wsdl4j.jar org.apache.axis.wsdl.WSDL2Java -v -W -o
src -p com.amazon.ws http://webservices.amazon.com/AWSECommerceService/2005-03-23
/AWSECommerceService.wsdl
```

上述命令可以在 src 目录下获得 Axis 自动生成的相关 Java 接口和类，包名为指定的“com.amazon.ws”。使用“-h”参数可以获得 WSDL2Java 所有参数的详细信息。

下一步，编写 main()方法查找关键字为“j2ee”的所有书籍。

```
public static void main(String[] args) throws Exception {
    String wsdl = "http://webservices.amazon.com/AWSECommerceService/2005-
03-23/US/AWSECommerceService.wsdl";
    String namespaceUri = "http://webservices.amazon.com/AWSECommerceService
/2005-03-23";
    String serviceName = "AWSECommerceService";
    String portName = "AWSECommerceServicePort";
    QName serviceQN = new QName(namespaceUri, serviceName);
    QName portQN = new QName(namespaceUri, portName);
    ServiceFactory factory = ServiceFactory.newInstance();
    Service srv = factory.createService(new URL(wsdl), serviceQN);
    AWSECommerceServicePortType service = (AWSECommerceServicePortType)srv.
getPort(portQN, AWSECommerceServicePortType.class);
    ItemSearch itemSearch = new ItemSearch();
    ItemSearchRequest request = new ItemSearchRequest();
```

```
            // 设定 Amazon Access Key，请替换为您的 Amazon Access Key:
            itemSearch.setSubscriptionId("1GY19EPMN1B149YS3VR2");

            request.setResponseGroup(new String [] {"Small"} );
            request.setSearchIndex("Books");
            request.setKeywords("j2ee");

            itemSearch.setRequest ( new ItemSearchRequest [] { request } );
            ItemSearchResponse response = service.itemSearch(itemSearch);
            if(response==null)
               throw new Exception("Server Error - no response recieved!");

            Items[] itemsArray = response.getItems();
            if(itemsArray==null)
               throw new Exception("Server Error - empty response!");

            if(itemsArray[0].getRequest().getErrors()!=null)
               throw new Exception (itemsArray [0].getRequest().getErrors()[0].
getMessage());

            for(Items items : itemsArray) {
               if(items!=null) {
                  System.out.println("TotalPages " + items.getTotalPages());
                  System.out.println("TotalResults " + items.getTotalResults());
                  Item[] itemArray = items.getItem();
                  if(itemArray!=null) {
                     for(Item item : itemArray) {
                        if(item!=null) {
                           ItemAttributes itemAttributes = item.getItemAttributes();
                           if(itemAttributes!=null) {
                              System.out.println("\nTitle: " + itemAttributes.
getTitle());
                              StringBuffer authors = new StringBuffer();
                              String[] authorArray = itemAttributes.getAuthor();
                              if(authorArray != null) {
                                 for(String author : authorArray) {
                                    authors.append(author).append(', ');
                                 }
                              }
                              System.out.println(authors.toString());
                           }
                        }
                     }
                  }
               }
            }
         }
```

运行应用程序，可以看到 Amazon 的 Web 服务返回了 503 个结果和第 1 页的 10 本书籍的信息。

```
        TotalPages 51
        TotalResults 503

        Title: Head First Servlets and JSP: Passing the Sun Certified Web Component
Developer Exam (SCWCD)
        Bryan Basham, Kathy Sierra, Bert Bates,

        Title: Core JavaEE Patterns: Best Practices and Design Strategies, Second
Edition
        Deepak Alur, Dan Malks, John Crupi,

        Title: Expert One-on-One JavaEE Design and Development (Programmer to
Programmer)
        Rod Johnson,

        Title: Enterprise Service Bus: Theory in Practice
        David Chappell,

        Title: Enterprise JavaBeans 3.0 (5th Edition)
        Bill Burke, Richard Monson-Haefel,

        Title: Rails Recipes (Pragmatic Programmers)
        Chad Fowler,

        Title: JavaEE Web Services
        Richard Monson-Haefel,

        Title: SCJA - Sun Certified Java Associate Certification Study Guide for Java
5, JavaEE and J2ME Technology from ExamScam.com - The Pre SCJP, Programmers Certification
        Cameron McKenzie,

        Title: Core Security Patterns: Best Practices and Strategies for JavaEE(TM),
Web Services, and Identity Management (Core Series)
        Christopher Steel, Ramesh Nagappan, Ray Lai,

        Title: Sun Certified Enterprise Architect for JavaEE Technology Study Guide
        Mark Cade, Simon Roberts,
```

读者可以看到，在客户端调用 Web 服务时，需要知道的唯一信息就是 WSDL 文件的 URL 地址，只要获取了该 WSDL 文件，就获得了调用 Web 服务的全部信息。在 Java 环境中调用 Web 服务前，还需要使用 wsdl2java 这个工具来根据 WSDL 自动生成客户端

支持类。需要注意的一点是，不同的 Web 服务框架的 wsdl2java 生成的代码是和该框架绑定的，例如，上面的例子使用了 Axis 的 wsdl2java，自动生成的客户端支持类就只能通过 Axis 来调用 Web 服务。如果使用其他的 Web 服务框架（例如，XFire），就必须使用 XFire 提供的 wsdl2java 工具。不过，客户端的调用代码仍符合标准的 JAX-RPC 接口，所以不会和具体的底层框架（例如，Axis）绑定。简言之，替换底层的 Web 服务框架只需要用新的 wsdl2java 工具重新自动生成一遍客户端支持类即可。

8.3.2 在Spring中调用Web服务

为了获得 Web 服务接口，我们必须手动编写烦琐而冗长的代码，在实际应用中，还必须捕获各种异常。为了简化调用 Web 服务，Spring 提供了 LocalJaxRpcServiceFactoryBean 和 JaxRpcPortProxyFactoryBean 来封装一个 Web 服务的调用。前者只能返回 JAX-RPC 的服务类，而后者的封装更好，可以返回一个自定义的接口类，使客户端完全和 Web 服务隔离开。

我们以 JaxRpcPortProxyFactoryBean 为例，首先创建一个 CallAmazonWS_Spring 工程，结构与 CallAmazonWS 类似，只是多了一个 spring.jar，如图 8-10 所示。

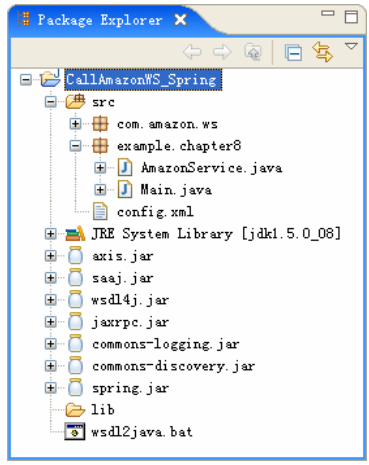


图 8-10

然后，在 Spring 的 XML 配置文件中定义 Amazon 的 Web 服务接口。

```
        <bean id="amazonWebService" class="org.springframework.remoting.jaxrpc.
JaxRpcPortProxyFactoryBean">
            <property name="serviceInterface" value=" com.amazon.ws.AWSECommerce
ServicePortType" />
            <property name="portInterface" value="com.amazon.ws.AWSECommerce
ServicePortType" />
```

```
            <property name="wsdlDocumentUrl" value="http://webservices.amazon.com/
AWSECommerceService/2005-03-23/US/AWSECommerceService.wsdl" />
            <property name="namespaceUri" value="http://webservices.amazon.com/
AWSECommerceService/2005-03-23" />
            <property name="serviceName" value="AWSECommerceService" />
            <property name="portName" value="AWSECommerceServicePort" />
        </bean>
```

由于返回的 serviceInterface 是 AWSECommerceServicePortType，从 Remote 接口派生而来，每个远程方法都将抛出 RemoteException。如果希望将 RemoteException 异常屏蔽掉，可以自定义一个与 AWSECommerceServicePortType 具有**相同方法签名**的接口，但不抛出 RemoteException。例如，定义 AmazonService 接口。

```
        public interface AmazonService {
            public com.amazon.ws.HelpResponse help(Help body);
            public ItemSearchResponse itemSearch(ItemSearch body);
            public ItemLookupResponse itemLookup(ItemLookup body);
            ...
        }
```

然后，让 Spring 返回我们自定义的 AmazonService 接口。

```
        <bean id="amazonWebService"
class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
            <property name="serviceInterface" value="example.chapter8.AmazonService" />
            <property name="portInterface" value="com.amazon.ws.AWSECommerceService
PortType" />
            <property name="wsdlDocumentUrl" value="http://webservices.amazon.com/
AWSECommerceService/2005-03-23/US/AWSECommerceService.wsdl" />
            <property name="namespaceUri" value="http://webservices.amazon.com/AWSE
CommerceService/2005-03-23" />
            <property name="serviceName" value="AWSECommerceService" />
            <property name="portName" value="AWSECommerceServicePort" />
        </bean>
```

现在，我们在 main()方法中就无需捕获 RemoteException。

```
        public static void main(String[] args) {
            ApplicationContext context = new ClassPathXmlApplicationContext("config.
xml");
            AmazonService service = (AmazonService)context.getBean("amazonWebService");
        }
```

相比直接使用 JAX-RPC，使用 Spring 最大的好处是免去了获取 Web 服务接口的烦琐代码，并且可以将远程 Web 服务接口转换为本地接口，从而使客户端无需捕获

RemoteException。从客户端的角度看，由于使用了自定义的接口，从而完全隔离了 Web 服务，客户端甚至不知道自己调用的接口实际上是一个 Web 服务，这样更便于使用模拟对象来代替 Web 服务，从而大大简化了测试。

如果调用 Web 服务时抛出了 RemoteException，Spring 会自动将 RemoteException 转化为 RemoteAccessException。由于 RemoteAccessException 是一个 RuntimeException，因此无需在客户端捕获。

8.3.3 发布Web服务

前面已经介绍了如何访问已有的 Web 服务，我们很自然地想到，如果需要发布自己的 Web 服务，应该如何实现。

发布 Web 服务本质上就是要最终生成 Web 服务描述文件，即 WSDL 文件，然后告知客户端该文件的 URL。客户端如果要调用我们发布的 Web 服务，只需要获取 WSDL 文件，然后从中获得调用 Web 服务所需的全部信息。如果查看 Amazon 的 WSDL 文件就会发现，该 XML 文件格式非常复杂，根本不可能手动编写，更不可能手动维护，因为 WSDL 文件的设计目标之一就是为了能使用工具自动生成让机器识别的 XML。所以，要发布 Web 服务，必须借助工具从现有的 Java 接口及支持类自动生成 WSDL 文件。

如果读者使用过 Microsoft Visual Studio 2005 来发布一个 Web 服务，通过几个简单的向导页面，整个发布过程将不超过一分钟。遗憾的是，对于 JavaEE 开发者来说，虽然 Web 服务的标准早已被引入到 JavaEE 体系中，然而，长期以来，在 JavaEE 环境中发布 Web 服务是极其困难的，如果使用 Axis 发布一个 Web 服务，至少需要手动编写一个复杂的 deploy.wsdd 配置文件，然后需要手动运行 AdminClient 来生成 Axis 需要的支持文件，整个过程仍比较复杂。

幸运的是，随着新一代的 Java Web 服务引擎 XFire 的发布，在 JavaEE 中发布 Web 服务将变得轻而易举。和其他 JavaEE 的 Web 服务引擎相比，XFire 的配置非常简单，它使得 JavaEE 开发人员终于可以获得和.Net 开发人员一样的开发效率，但是在功能上，XFire 却丝毫不逊色于其他 Web 服务引擎，由于使用了 StAX（the Streaming API for XML，基于流的 XML 解析）作为 XML 解析器，XFire 的运行速度又有了质的提高，并且 XFire 支持最新的 JSR 181 的 Web 服务注解。如果使用 Java 5，只需要在源代码中编写相应的 JSR 181 注解，XFire 就可以根据 Java 5 注解自动提取所需的全部信息。由于 JSR 181 也是 JavaEE Web 服务框架的一部分，使用它最大的好处在于不仅极大地简化了配置，而且避免了配置文件和某个特定的 Web 服务引擎的锁定（例如，deploy.wsdd 是 Axis 的配置文件，在其他 Web 服务引擎中不能使用）。

在下面的例子中，我们将看到如何使用 XFire 发布 Web 服务，并且在 Spring 环境中集成它。

1．使用 XFire 发布 Web 服务

我们首先在单独的 XFire 中发布一个 Web 服务。要获得最新的 XFire，请从 http://xfire.codehaus.org/下载，本书的配套光盘中也附带了 XFire 的发布包。然后，解压到本地即可。

在 Eclipse 中建立如图 8-11 所示的 ExportWS 工程。

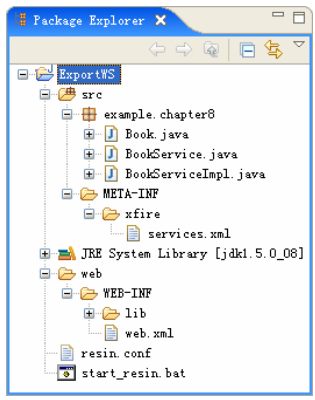


图 8-11

其中，web/WEB-INF/lib 目录下存放了 XFire 需要的库文件，请复制 XFire 根目录下的 xfire-all-1.2.2.jar 及 lib 目录下的所有 jar 文件。需要注意的是，单独使用 XFire 还需要 Spring 1.2.x 的 jar 文件，因为 XFire 用到了 Spring 框架提供的一些功能。不过不用担心，在与 Spring 2.0 框架集成时，就不需要 Spring 1.2.x 了，经过测试，XFire 与 Spring 2.0 配合得相当好。

有些 jar 文件是可选的，读者可以参考 XFire 的文档，删除不必要的 jar 文件，这样可以减小 Web 应用程序的发布文件。

以本书的 Live 在线书店为例，假定我们的 Web 应用程序也准备对外提供一个书籍查询的功能，可以根据关键字返回对应的书籍信息，因此，我们首先编写一个 BookService 接口，作为 Web 服务的接口。

```
public interface BookService {
    Book[] search(String keyword);
}
```

然后，编写一个相应的实现类。这里，我们的服务接口还只能返回以“j2ee”作为关键字的书籍信息，对于其他的关键字，一律返回一个空数组。

```
public class BookServiceImpl implements BookService {
    public Book[] search(String keyword) {
        if("j2ee".equalsIgnoreCase(keyword)) {
            return new Book[] {
                new Book("Core JavaEE Patterns", "Dan Malks"),
                new Book("The JavaEE Tutorial", "Bode Carson"),
                new Book("JavaEE Design Patterns", "William Crawford"),
                new Book("JavaEE Platform Web Services", "Ray Lai")
            };
        }
        return new Book[0];
    }
}
```

下一步，编写 XFire 必须的一个 Web 服务描述文件 services.xml，该文件必须放到 ClassPath 中的 META-INF/xfire /目录下，在这个工程中，我们将其放到 src/META-INF/xfire/目录下。这个文件将为 XFire 提供生成 Web 服务的描述信息，其内容非常简单，只需指定 Web 服务的接口类和实现类即可。

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
    <service>
        <name>BookService</name>
        <namespace>http://www.livebookstore.net/BookService</namespace>
        <serviceClass>example.chapter8.BookService</serviceClass>
        <implementationClass>example.chapter8.BookServiceImpl</implement
ationClass>
    </service>
</beans>
```

最后，在 web.xml 文件中加入 XFireConfigurableServlet 的声明，并映射到指定的路径。

```
<!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>
        <servlet-name>XFireServlet</servlet-name>
        <servlet-class>
            org.codehaus.xfire.transport.http.XFireConfigurableServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>XFireServlet</servlet-name>
        <url-pattern>/servlet/XFireServlet/*</url-pattern>
```

```
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>XFireServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

传统的处理 XML 的 API 有 DOM 和 SAX 两种，DOM 速度慢，占用内存极大，SAX 虽然比 DOM 快，但速度仍不理想。现在，一种新的基于流的 Stream API for XML（简称 StAX）极大地提高了 XML 的处理速度，并且已经成为 JSR 173 标准（http://www.jcp.org/en/jsr/detail?id=173）。目前，StAX 最好的开源实现是 Woodstox（http://woodstox.codehaus.org/），它完整地支持 StAX API。Resin 3.1 也内置了 StAX 的实现，但是并没有支持所有的 StAX API。XFire 在 ClassPath 上如果找到了 StAX API 的定义，就会试图定位一个 StAX 的实现。为了让 XFire 不使用 Resin 3.1 内置的 StAX 实现，而使用 Woodstox 的实现，就必须在 Resin 服务器启动时指定以下参数：

```
-Djavax.xml.stream.XMLInputFactory=com.ctc.wstx.stax.WstxInputFactory
-Djavax.xml.stream.XMLOutputFactory=com.ctc.wstx.stax.WstxOutputFactory
-Djavax.xml.stream.XMLEventFactory=com.ctc.wstx.stax.WstxEventFactory
```

我们编写了一个 start_resin_using_woodstox.bat 脚本来启动 Resin 3.1 并指定以上参数。同时要确保 ClassPath 包含了 StAX API 和 Woodtox 实现，即/web/lib/目录下必须有 stax-api.jar 和 wstx-asl-3.0.1.jar。

运行 start_resin_using_woodstox.bat 以启动 Resin 服务器，然后打开浏览器，输入地址：http://localhost: 8080/services/BookService?wsdl，就可以看到 BookService Web 服务的 WSDL 描述文档，如图 8-12 所示。

这正是客户端调用 BookService 所需要的唯一信息，任何需要调用 BookService 的客户端只需要知道这个 URL 即可调用它。为了测试我们发布的 Web 服务是否能够正常地在客户端被调用，下面使用 Visual Studio 2005 作为客户端开发环境，直接在 C#工程中调用该 Web 服务。

在 Visual Studio 2005 中新建一个 C# Console Application 工程，命名为“CallBookService”，然后选择菜单“Project”→“Add Web Reference...”，在弹出的对话框中输入 WSDL 的地址并回车，等待几秒钟后，Visual Studio 2005 就会自动发现 BookService 并找到 search() 方法，如图 8-13 所示。

图 8-12

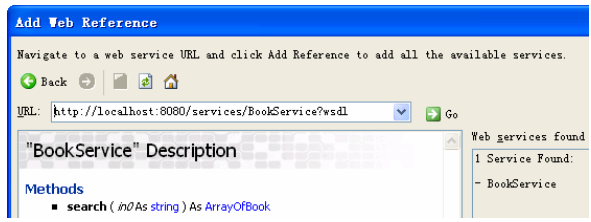


图 8-13

单击“Add Reference”按钮后，BookService 就被自动添加进了工程中，Visual Studio 2005 会自动为该 Web 服务生成客户端支持文件/Web References/localhost/Reference.cs，其命名空间为 CallBookService.localhost，如图 8-14 所示。

打开 Program.cs 文件，编写代码调用 BookService，如图 8-15 所示。

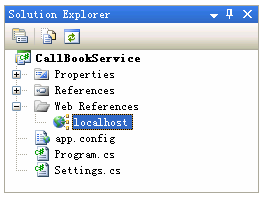


图 8-14

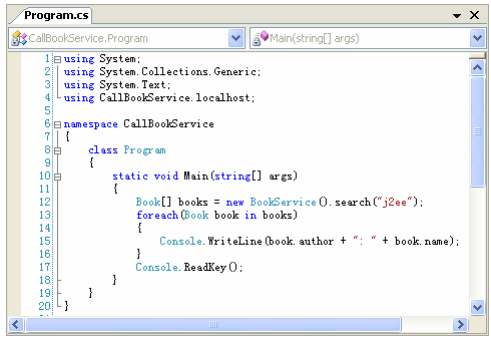


图 8-15

运行 C#代码，在控制台窗口可以看到如图 8-16 所示的输出。

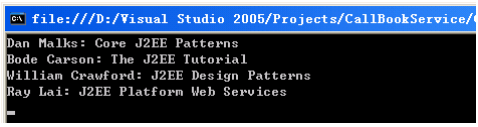


图 8-16

调用成功！说明我们发布的 Web 服务确实能够被异构平台正确调用。也可以使用其他任何支持 Web 服务的语言来调用该 Web 服务，包括前面介绍的在 Spring 中访问 Web 服务的方法。

2．在 Spring 中集成 XFire

在前面我们看到了 XFire 作为 Web 服务引擎能极大地简化 Web 服务的发布。由于 Web 服务不过是为了将系统内部的某些功能暴露给外部，使其他的异构客户端都能调用

系统的某些功能，因此，仅仅编写一个简单的 JavaBean 作为 Web 服务的实现是不符合实际应用的，一个实际应用程序的 Web 服务必然要涉及调用系统内部的逻辑组件，包括访问数据库以获取必要的数据等。在前面的章节中，我们已经介绍了如何在 Spring 的 IoC 容器中完成所有组件的部署，如果希望将某些组件的功能暴露为 Web 服务，就必须以某种方式将 XFire 集成到 Spring 应用程序中。

幸运的是，XFire 提供了对 Spring 的完善支持，并且在 Spring 2.0 环境下也没有任何问题。下面的例子将讲述如何在 Spring 2.0 框架中集成 XFire 并发布 Web 服务。

我们将 ExportWS 工程复制一份，重新命名为 ExportWS_Spring，然后将 web/WEB-INF/lib 目录下的 spring-1.2.x.jar 替换为 Spring 2.0 的 jar 文件，此外，src/META-INF/xfire/services.xml 也不需要了，直接将整个 META-INF 目录删掉，Web 服务的相关信息稍后将直接在 Spring 的 XML 配置文件 dispatcher-servlet.xml 中定义。

在上一个例子中，我们仍使用了传统的 XML 配置文件来声明 Web 服务，现在，我们使用 JSR 181 Web 服务注解来直接标注 Web 服务。对 BookServiceImpl 增加如下注解。

```
import javax.jws.*;

@WebService(
       name="BookService",
       serviceName="BookService",
       targetNamespace="http://www.livebookstore.net/BookService"
)
public class BookServiceImpl implements BookService {

   @WebMethod
   @WebResult
   public Book[] search(@WebParam String keyword) {
      if("j2ee".equalsIgnoreCase(keyword)) {
         return new Book[] {
            new Book("Core JavaEE Patterns", "Dan Malks"),
            new Book("The JavaEE Tutorial", "Bode Carson"),
            new Book("JavaEE Design Patterns", "William Crawford"),
            new Book("JavaEE Platform Web Services", "Ray Lai")
         };
      }
      return new Book[0];
   }
}
```

JSR 181 的注解非常简单，读者可以很容易地猜出各个注解的含义。

和普通的 Spring Web 应用程序一样，在 web.xml 文件中声明 Spring 的 Dispatcher Servlet。

```
<!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>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

然后编写 Spring 的 XML 配置文件 dispatcher-servlet.xml，内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- 引用 XFire 预定义的 Bean 配置 -->
    <import resource="classpath:org/codehaus/xfire/spring/xfire.xml" />

    <!-- Web 服务实现类 -->
    <bean id="bookService" class="example.chapter8.BookServiceImpl" />

    <!-- 定义 URL 映射 -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandler
Mapping">
        <property name="urlMap">
            <map>
                <entry key="/BookService" value="bookExporter" />
            </map>
        </property>
    </bean>

    <!-- bookExporter 将处理来自客户端的调用 -->
    <bean id="bookExporter" class="org.codehaus.xfire.spring.remoting.
XFireExporter">
```

```
        <!-- 对外提供的 Web 服务的接口 -->
        <property name="serviceClass" value="example.chapter8.BookService" />
        <!-- 实现 Web 服务的 Bean -->
        <property name="serviceBean" ref="bookService" />
        <!-- 下面引用的 Bean 都已被 import 引入了 -->
        <property name="serviceFactory" ref="xfire.serviceFactory" />
        <property name="xfire" ref="xfire" />
    </bean>

</beans>
```

需要注意的是，这个 XML 配置文件必须首先通过<import>引入 XFire 的一些预定义的 Bean，这样就可以直接使用 XFireExporter 来处理 Web 服务请求。整个配置相当简单。由于是在 Spring 的容器中配置的，对于 bookService 组件来说，只要注入了访问数据库的组件（例如，BookDao），就可以真正实现一个有用的 Web 服务。

读者仍可以使用 Microsoft Visual Studio 2005 来测试这个通过 Spring 集成 XFire 发布的 Web 服务，不过请注意，这个工程的 BookService Web 服务的 URL 地址变为 http://localhost: 8080/BookService?wsdl，如图 8-17 所示。

图 8-17

现在，我们已经详细介绍了 Web 服务的原理，如何调用和发布 Web 服务，以及如何在 Spring 框架中集成 Web 服务。在调用 Web 服务时，我们使用的是 Axis，但客户端代码使用的是标准的 JAX-RPC 接口，因此，不存在绑定 Axis 的问题。如果读者希望使用 XFire 来调用 Web 服务，可以使用 XFire 提供的 wsdl2java 工具来生成相关的支持类，客户端代码是不变的。在发布 Web 服务时，我们只介绍了如何使用 XFire 来实现，因为相比其他的 Web 服务引擎，XFire 在实现 Web 服务上有巨大的配置和性能优势，因此，XFire 应当作为 JavaEE 环境下发布 Web 服务的首选。

Spring 框架的另一个子项目 Spring Web Services（http://www.springframework.org/spring-ws）的目标就是提供一个与 Spring 紧密结合的 Web 服务模块，并与 Spring 的 MVC 框架集成，这个项目可以作为 Spring 框架下发布 Web 服务的又一个选择，不过，截至本书写作时，Spring Web Services 项目仍处于开发阶段，尚未有稳定版本发布。对

此项目感兴趣的读者可以参考 Spring 官方网站上的相关文档，本书对此将不再多做介绍。

在本章写作时，Java 6 已经正式发布了。Java 6 的一个重大特性就是对 Web 服务的内在支持，发布和访问 Web 服务都变得轻而易举。如果采用基于 Java 6 的 Web 服务器，那么不必使用任何第三方 Web 服务的支持库，就可以直接将 Web 服务完美地集成进来。不过，Java 6 从正式发布到各 Web 服务器对其支持尚需一段较长的时间，目前，在服务器端发布 Web 服务的最佳选择仍是 XFire。

8.4 小结

本章我们主要介绍了 JavaEE 平台的各种远程调用技术，以及如何在 Spring 框架中方便地集成它们，既包括了作为客户端调用远程服务，也包括了如何在 Spring 环境中发布远程服务。下面，我们来比较一下各种远程调用的特点和适用范围。

（1）RMI：RMI 协议是专门为 Java 设计的一种二进制协议，这意味着 RMI 只能在 Java 应用程序间通信，而无法与非 Java 应用程序通信。此外，RMI 使用专用的端口，这使得 RMI 很难穿透防火墙。RMI 的好处在于可以直接传输 Java 对象，因此实现起来非常简单。

（2）Hessian、Burlap 和 Spring HTTP Invoker：这 3 种协议都是基于 HTTP 的轻量级协议，因此，很容易穿透防火墙，并且理论上采用 XML 格式的 Burlap 可以和其他任意应用程序通信，只要对方能正确解析 XML。但是，由于这 3 种协议都是私有协议，没有成为标准，实际应用极少，因此不推荐使用这 3 种协议来实现远程调用。

（3）Web Services：Web 服务早已成为工业标准，被所有的大厂商支持，并且在因特网上得到了广泛的应用。由于以 HTTP 为基础，因此可以很容易穿透防火墙，并且由于 XML 的平台无关性，通信双方可以使用任何平台。Web Service 的缺点在于部署和调用都相对困难，而且效率较低。

基于上述考虑，我们建议，如果仅在企业内部网中使用，并且通信双方都是 Java 平台，可以考虑使用 RMI，如果需要通过因特网调用，或者通信双方是异构系统（例如，Java 系统和.Net 系统），则 Web 服务几乎是唯一的选择。

在第 9 章，我们将介绍 Spring 集成的其他常用的服务，包括电子邮件服务、任务调度服务、JMS 消息服务、JMX 集成、EJB 调用等，通过集成这些服务，加上 IoC 容器、AOP、数据访问模型、声明式事务管理和 Web MVC 框架，整个 Spring 框架就基本上涵盖了 JavaEE 企业级应用开发所需的绝大多数功能。

第 9 章

Spring集成的其他功能

在前面几章中，我们已经介绍了 Spring 框架提供的主要功能，包括 IoC 容器、AOP 支持、统一的数据访问模型、强大的事务管理和灵活的 MVC 框架。除了这些构成多层应用程序基础的模块外，通常，应用程序还需要一些其他的系统服务，例如，电子邮件服务等。本章主要介绍 Spring 集成的常用的服务，包括邮件服务、任务调度和 Java 消息服务。

9.1 集成邮件服务

JavaMail API 为 Java 应用程序提供了邮件发送和接收的服务。JavaMail 提供了一个平台无关和协议无关的框架，可以在 JavaMail API 的基础上构建一整套电子邮件应用。JavaMail 支持常见的电子邮件协议，包括 SMTP、IMAP 和 POP3。在 JavaEE 应用程序中，我们关心的是如何通过 SMTP 协议发送电子邮件，因此，本节仅讨论如何发送电子邮件，如果需要接收电子邮件，可以使用 POP3 协议，同样非常简单。

9.1.1 发送纯文本邮件

如果直接使用 JavaMail API 发送邮件，即使发送最简单的纯文本邮件，也不得不编写如下代码。

```
        public static void send(Properties props, final String username, final String
password, String from, String[] to, String subject, String text) throws
MessagingException {
            Session session = Session.getInstance(props, new Authenticator() {
                public PasswordAuthentication getPasswordAuthentication() {
                    return new PasswordAuthentication(username, password);
                }
            });
            session.setDebug(true);
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress(from));
            message.setSubject(subject);
            message.setText(text);
            message.setSentDate(new Date());
            Address[] addressTo = new Address[to.length];
            for(int i=0; i<to.length; i++) {
                addressTo[i] = new InternetAddress(to[i]);
            }
            message.setRecipients(Message.RecipientType.TO, addressTo);
            message.saveChanges();
```

```
        Transport.send(message, addressTo);
    }
```

调用这个方法还需要准备一个 Properties 对象，下面的代码将使用 GMail 的 SMTP 服务器发送邮件。

```
Properties props = new Properties();
// 设置 SMTP 服务器:
props.put("mail.smtp.host", "smtp.gmail.com");
props.put("mail.smtp.port", "465");
props.put("mail.smtp.auth", "true");
// 使用 SSL 安全连接:
props.put("mail.smtp.starttls.enable", "true");
props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
send(
    props,
    "livebookstore", // 用户名
    "LiVe-BoOkStOrE", // 口令
    "livebookstore@gmail.com", // 发件人
    new String[] {"livebookstore2@gmail.com"}, // 请修改收件人地址
    "A test mail", // 主题
    "测试使用 JavaMail API 发送邮件" // 邮件正文
);
```

这样的代码不仅冗长而且不便于封装为组件来复用。Spring 提供了非常方便的 Mail 抽象层。它通过一个 MailSender 接口封装了邮件发送 Bean，而 SimpleMailMessage 封装了纯文本的简单邮件，它是一个纯粹的 POJO。

Spring 的 MailSender 被设计为可插拔的接口，它支持两种邮件发送服务：标准的 JavaMail 和 CosMail。通常我们使用标准的 JavaMail 即可。JavaMail 的实现库并没有集成到 JDK 5.0 中，需要手动从 SUN 的官方站点下载 mail.jar 和 activation.jar 这两个库，读者也可以在本书的配套光盘中找到。建立如图 9-1 所示的 Mail 项目。

然后编写 config.xml 配置文件，定义 mailSender Bean。

```
    <bean id="mailSender" class="org.springframework.mail.javamail.JavaMail
SenderImpl">
        <property name="host" value="smtp.gmail.com" />
        <property name="port" value="465" />
        <property name="username" value="livebookstore" />
        <property name="password" value="LiVe-BoOkStOrE" />
        <property name="javaMailProperties">
            <props>
                <prop key="mail.smtp.auth">true</prop>
                <prop key="mail.smtp.starttls.enable">true</prop>
```

```
                <prop key="mail.smtp.socketFactory.class">javax.net.ssl.
SSLSocketFactory</prop>
            </props>
        </property>
    </bean>
```

发送邮件所需的所有 SMTP 配置都被注入到 Bean 中。我们编写测试程序，看看使用 Spring 提供的这个 JavaMailSender 发送邮件是多么方便。

```
        ApplicationContext context = new ClassPathXmlApplicationContext("config.
xml");
        JavaMailSender mailSender = (JavaMailSender)context.getBean
("mailSender");
        // 创建一个纯文本邮件:
        SimpleMailMessage mail = new SimpleMailMessage();
        mail.setFrom("livebookstore@gmail.com");
        mail.setTo("livebookstore2@gmail.com"); // 请修改收件人地址
        mail.setSubject("Another test mail");
        mail.setText("测试使用 Spring MailSender 发送邮件");
        // 发送:
        mailSender.send(mail);
```

请修改收件人地址，然后运行测试程序，过几分钟后打开邮箱，可以看到接收到的纯文本邮件，如图 9-2 所示。

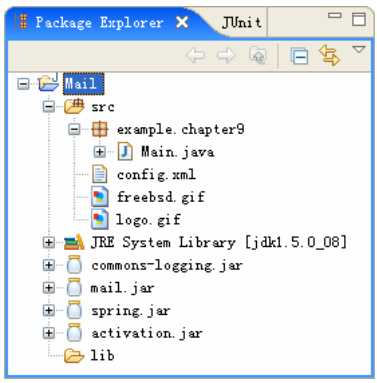


图 9-1

图 9-2

9.1.2 发送MIME邮件

在 JavaMail 中发送带有附件的 MimeMessage 是非常麻烦的，需要构造 Multipart 和多个 MimeBodyPart 对象，而 Spring 提供了一个助手类 MimeMessageHelper，能非常方便地操作 MimeMessage。下面的例子演示了在获得了一个 JavaMailSender Bean 之后，通

过 MimeMessageHelper 发送一个带有两个附件的 HTML 邮件。

```
MimeMessage mime = mailSender.createMimeMessage();
// 创建 Helper 实例并声明编码为 UTF-8，true 表示 Multipart:
MimeMessageHelper helper = new MimeMessageHelper(mime, true, "UTF-8");
helper.setFrom("livebookstore@gmail.com");
helper.setTo("livebookstore2@gmail.com"); // 请修改收件人地址
helper.setSubject("Test mime mail");
helper.setText("<html><body>访问 Live 在线书店："
        + "<a href='http://www.livebookstore.net' target='_blank'>"
        + "<img src='cid:logo'></a></body></html>",
        true); // true 表示设定为 HTML 格式
// 添加一个嵌入的图片：
helper.addInline("logo", new ClassPathResource("logo.gif"));
// 添加一个普通的附件：
helper.addAttachment("freebsd.gif", new ClassPathResource("freebsd.gif"));
// 发送：
mailSender.send(mime);
```

需要注意的一点是，在添加附件的时候，如果需要将附件嵌入到 HTML 中显示，则在<img>标签中用“cid:xxx”标记，对应的附件通过 addInline()方法添加。

请修改收件人地址然后运行测试程序，稍后打开邮箱，可以以 HTML 格式看到页面和嵌入的 Live 在线书店 Logo 的图片，如图 9-3 所示。

图 9-3

9.2 集成任务调度服务

并非所有的任务都是由用户发起请求而产生的，大多数系统都需要周期性地运行一些调度任务，比如，需要定时分析每天的日志记录，然后自动发送报告给系统管理员。要在 JavaEE 应用程序中实现这些调度任务，Spring 提供了一个非常实用的调度器，可以

方便地配置任务并定期执行。

根据 Spring“不重新发明轮子”的原则，Spring 框架没有自己实现调度器，而是提供了一个抽象层，它封装了 JDK 的 Timer 类和开源的 Quartz 调度器。

9.2.1 使用Timer调度任务

从 JDK 1.3 开始，提供了一个 Timer 类，可以实现周期性地执行某个任务。Spring 对其包装为 ScheduledTimerTask，设置起来非常简单。

为了能让系统周期性地分析日志，并发送邮件给管理员，在 Eclipse 中建立 ReportTimerTask 工程，如图 9-4 所示。

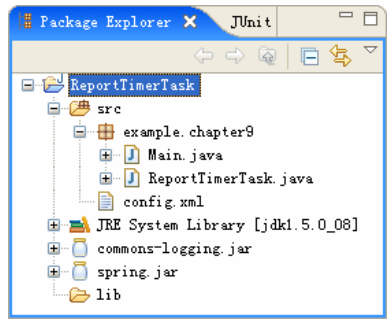


图 9-4

首先，需要定义一个任务，它派生自 TimerTask。

```
public class ReportTimerTask extends TimerTask {
    public void run() {
        String log = "read from log file at " + new Date();
        System.out.println(log);
        // analyse...
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
        // send mail...
    }
}
```

我们用 Thread.sleep()模拟一个耗时的任务。至于发送邮件，可以参考 9.1 节的内容，为 ReportTimerTask 注入一个定义好的 JavaMailSender 对象，读者可以自己添加相关的

依赖注入属性，完整地实现发送邮件的功能。

下一步是在 XML 配置文件中定义好 reportTask 和 scheduledTask，其配置都相当简单。

```
<bean id="reportTask" class="example.chapter9.ReportTimerTask" />

<bean id="scheduledTask" class="org.springframework.scheduling.timer.
ScheduledTimerTask">
    <!-- 启动后等待 10 秒，然后开始执行 -->
    <property name="delay" value="10000" />
    <!-- 每 60 秒执行一次 -->
    <property name="period" value="60000" />
    <property name="timerTask" ref="reportTask" />
</bean>
```

reportTask 定义了一个独立的任务，而 scheduledTask 定义了周期性的任务。最后，通过定义一个 timerFactory 来启动这个周期性任务。

```
<bean id="timerFactory" class="org.springframework.scheduling.timer.
TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <ref bean="scheduledTask" />
        </list>
    </property>
</bean>
```

在测试程序中，我们启动 Spring 容器，然后等待 5 分钟后结束程序。

```
public static void main(String[] args) {
    new ClassPathXmlApplicationContext("config.xml");
    // 等待 5 分钟,观察输出:
    try {
        Thread.sleep(300000);
    }
    catch(InterruptedException e) {}
    }
    System.exit(0);
}
```

观察输出，可以看到，从容器启动后 10 秒开始，ReportTask 以 1 分钟的间隔周期性运行。

```
2006-10-18 17:00:04 org.springframework.scheduling.timer.TimerFactoryBean
afterPropertiesSet
```

```
信息: Initializing Timer
read from log file at Wed Oct 18 17:00:14 CST 2006
read from log file at Wed Oct 18 17:01:14 CST 2006
read from log file at Wed Oct 18 17:02:14 CST 2006
read from log file at Wed Oct 18 17:03:14 CST 2006
read from log file at Wed Oct 18 17:04:14 CST 2006
```

如果只想运行一次任务，可以设定 period 的值为 0。

另一个创建 ScheduledTask 的方法更简单，即使用 MethodInvokingTimerTaskFactoryBean，它甚至不需要 reportTask 扩展 TimerTask，只要指定方法名称即可。

```
    <bean id="scheduledTask" class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
        <property name="targetObject" ref="reportTask" />
        <property name="targetMethod" value="run" />
    </bean>
```

注意：这个 MethodInvokingTimerTaskFactoryBean 是一个 FactoryBean，因此 Spring 容器创建的实际对象类型是 ScheduledTimerTask。

由于 JDK 提供的 Timer 功能有限，只能以固定的周期运行任务。如果希望以更灵活的方式运行任务，例如，希望每天凌晨 3:00 向管理员发送报告，每周星期一至星期五早上 6:00 向用户发送新闻，可以使用 Quartz 来实现更复杂的调度任务。

9.2.2 使用Quartz调度任务

Quartz 是一个功能极为强大的任务调度器，可以任意指定周期性的任务，Quartz 还支持将任务存储到数据库中，这样即使重启服务器也可以继续上次没有执行的任务。要使用 Quartz，请从 http://www.opensymphony.com/quartz/下载最新版本。

Quartz 实现任务调度的几个关键概念如下。

（1）Job：定义一个任务，Job 只管执行，不管什么时候执行，也不管执行多少次。

（2）Trigger：定义一个触发器，表示应当如何触发一个 Job 的执行。Quartz 提供了许多简单的 Trigger，可以实现某一时刻触发、周期性触发等多种触发方式，并且一个 Job 可以和多个 Trigger 关联，这样能更加灵活地执行 Job。

（3）Scheduler：真正调度任务的调度器，通过 scheduleJob(Job, Trigger)方法就把一个关联了 Trigger 的 Job 对象放入了调度器中执行。

简单的代码如下。

```
SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start(); // 启动调度器
JobDetail jobDetail = new JobDetail("myJob", null, AJob.class); // 定义任务
Trigger trigger = TriggerUtils.makeHourlyTrigger(); // 每小时执行一次
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date())); // 开始时间
trigger.setName("myTrigger");
sched.scheduleJob(jobDetail, trigger); // 调度执行
```

用 Quartz 实现上一个 Timer 版本的日志发送同样非常简单，却可以设置更多的调度方式。在 Eclipse 中建立 ReportQuartzTask 工程，如图 9-5 所示。

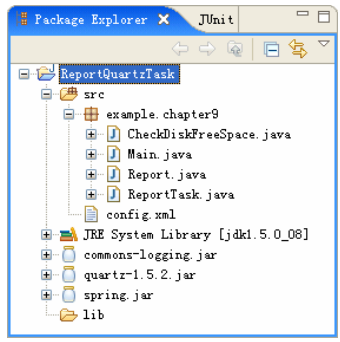


图 9-5

Spring 提供了两种方式来封装 Quartz 的 Job，一种是直接派生自 QuartzJobBean。例如，定义一个 Report 对象向管理员发送报告，并且可以注入管理员的名字。

```
public class Report extends QuartzJobBean {
    private String name;
    public void setName(String name) {
        this.name = name;
    }

    @Override
    protected void executeInternal(JobExecutionContext context) throws
JobExecutionException {
        System.out.println("Send report to " + name + " at " + new Date());
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在 XML 配置文件中将 Report 包装为 JobDetailBean。

```
    <bean name="reportJob" class="org.springframework.scheduling.quartz.
JobDetailBean">
        <property name="jobClass" value="example.chapter9.Report" />
        <property name="jobDataAsMap">
            <map>
                <entry key="name" value="Micheal" />
            </map>
        </property>
    </bean>
```

Spring 会自动将 jobDataAsMap 属性中注入的 Map 按照 key 注入到 Report 对象的相应的属性中。

另一种方式是完全用最简单的 POJO 实现，例如，设计一个检查磁盘剩余空间的 CheckDiskFreeSpace 对象。

```
    public class CheckDiskFreeSpace {
        public void check() {
            long freeSpace = Math.random() > 0.5 ? 100000000 : 200000000;
            System.out.println("Check disk free space at " + new Date());
            if(freeSpace<100*1024*1024) { // <100MB
                System.out.println("Warning! Low disk free space...");
            }
        }
    }
```

check()方法是执行方法，下一步是在 XML 配置文件中将其也包装为 JobDetailBean。

```
    <bean name="checkDiskFreeSpace"
    class="example.chapter9.CheckDiskFreeSpace" />

    <bean name="checkDiskJob" class="org.springframework.scheduling.quartz.
MethodInvokingJobDetailFactoryBean">
        <property name="targetObject" ref="checkDiskFreeSpace" />
        <property name="targetMethod" value="check" />
        <property name="concurrent" value="false" />
    </bean>
```

concurrent 属性指定了这个 checkDiskJob 是否能同时执行，默认为 true。若设置为 false，如果当前有一个 checkDiskJob 正在执行，则不启动相同的 checkDiskJob。如果一个 Job 执行时间较长，执行频率又比较高，定义 concurrent 为 false 可以避免多个相同的 Job 同时执行。

下一步是定义 Trigger，它定义了在何时触发 Job 的执行。Spring 封装了两种常用的 TriggerBean，一种是 SimpleTriggerBean，可以周期性地执行 Job。为了让 checkDiskJob 能每隔 5 分钟执行一次，配置如下。

```
        <bean id="repeatTrigger" class="org.springframework.scheduling.quartz.
SimpleTriggerBean">
            <property name="jobDetail" ref="checkDiskJob" />
            <!-- 1 分钟后启动 -->
            <property name="startDelay" value="60000" />
            <!-- 5 分钟检查一次 -->
            <property name="repeatInterval" value="300000" />
        </bean>
```

另一种 CronTriggerBean 功能更为强大，能指定何时执行 Job。例如，希望 reportJob 在周一至周五中午 12:00 执行，就可以在 XML 配置文件中按如下方式定义。

```
        <bean id="cronTrigger" class="org.springframework.scheduling.quartz.
CronTriggerBean">
            <property name="jobDetail" ref="reportJob" />
            <!-- 每周周一至周五中午 12:00 执行 -->
            <property name="cronExpression" value="0 0 12 ? * MON-FRI" />
        </bean>
```

cronExpression 表达式定义了 Job 的执行规则，稍后我们将详细讨论。现在，我们希望启动这些定义好的 Job，在 XML 配置文件中定义一个 Scheduler，然后将所有的 TriggerBean 都注入进去即可。

```
        <bean id="scheduler" class="org.springframework.scheduling.quartz.Scheduler
FactoryBean">
            <property name="triggers">
                <list>
                    <ref bean="repeatTrigger" />
                    <ref bean="cronTrigger" />
                </list>
            </property>
        </bean>
```

最后，编写 main()方法启动 Spring 容器。

```
        public static void main(String[] args) {
            new ClassPathXmlApplicationContext("config.xml");
        }
```

运行程序，观察输出。

```
        Nov 7, 2006 11:46:33 AM org.quartz.core.QuartzScheduler start
```

```
INFO: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED started.
Check disk free space at Tue Nov 07 11:47:33 CST 2006
Check disk free space at Tue Nov 07 11:52:33 CST 2006
Warning! Low disk free space...
Check disk free space at Tue Nov 07 11:57:33 CST 2006
Send report to Micheal at Tue Nov 07 12:00:00 CST 2006
Check disk free space at Tue Nov 07 12:02:33 CST 2006
Warning! Low disk free space...
```

可以看到 checkDiskJob 按照 5 分钟的间隔运行，而 reportJob 在周二的中午 12:00 执行了！为了能看到 reportJob 的执行效果，读者可能需要修改执行时间，否则最长要等待 24 小时才能看到 reportJob 第一次被执行。

现在我们再来讨论一下 cronExpression 的写法。cronExpression 用于定义触发 Job 的时间，由 7 个部分组成，分别代表：

（1）秒；

（2）分；

（3）小时；

（4）一月中的某天；

（5）月份；

（6）星期；

（7）年份（可选，通常不用指定，因为以年为周期执行的任务通常不需要 Quartz 来完成）

例如，“0 0 12 ? * MON”表示每周一 12:00:00 执行，?表示忽略月份中的天数，因为通常要么按照日期指定，要么按照星期指定，*表示全部，即每月都执行。

可以使用“,”指定一个列表，例如，“0 0 6 1,3,5 * ?”表示每月 1 号、3 号和 5 号的早上 6:00 执行，还可以使用“-”指定一个范围，例如，“0 0 6 ? * MON-FRI”表示周一至周五早上 6:00 执行。

可以使用“/”指定时间间隔，例如，设置分钟为“0/15”表示从 0 分钟开始，每隔 15 分钟执行一次，而“3/20”表示从 3 分钟开始，每隔 20 分钟执行一次，这和“3,23,43”的效果是完全一样的。

以下是一些常用的表达式。

“0 0/5 * * * ?”表示每隔 5 分钟执行一次。

“10 0/5 * * * ?”表示每隔 5 分钟，在 10 秒时执行，例如，00:10、05:10。

“0 30 10-12 ? * WED,FRI”表示每周三和每周五在 10:30、11:30、12:30 执行。

“0 0/30 9-17 1-5,10 * ?”表示在每月 1 号到 5 号，以及 10 号这几天，每天从 9:00 到 17:00 每隔 30 分钟执行一次。

一些更复杂的调度可能无法以一个 cronExpression 表示出来，此时，可以考虑将其分拆为几个 cronExpression，然后将这几个 CronTriggerBean 都添加到 Scheduler 中。

9.3 集成Java消息服务

Java 消息服务，即 JMS（Java Message Service），是 JavaEE 标准中为企业应用程序提供消息传递服务的 API 标准。JMS 使得异步发送和接收事件通知的应用程序变得容易设计和实现。

在 Spring 框架中，同样为 JMS 提供了非常好的封装，使 JMS 使用起来更加方便。

9.3.1 Java消息服务概述

大多数时候，应用程序内部的各个对象间都是以同步的方式进行方法调用的。当一个方法被调用时，当前线程的控制权就转移到这个方法中，直到方法执行完毕，线程的控制权才返回给调用者。

如果要执行一个非常耗时的方法，会使当前线程的执行变得缓慢，从而引起用户长时间地等待。为了避免同步调用来执行一个耗时的任务，可以在一个新的线程中执行这个耗时的任务，原来的线程则不必等待，就可以立刻继续执行下去。

相对于多线程模型，使用消息机制能更好地实现异步调用，而不必处理复杂的线程管理问题。JMS 为 Java 应用程序提供了完整的异步消息服务机制。

读者可能会问，Spring 的 ApplicationContext 容器本身也支持事件发布，类似消息传递，能否替代 JMS 呢？答案是否定的。因为 Spring 的 ApplicationContext 容器不支持异步事件处理，同步会使应用程序性能大打折扣。此外，JMS 并不要求通信双方一定要在同一台机器上，可以将发送消息和处理消息的应用程序部署在不同的机器上。JMS 还提供了持久化服务，这是 Spring 的 ApplicationContext 容器无法做到的。

直接使用 JMS API 会比较复杂，而 Spring 对 JMS 做了非常方便的封装。读者需要注意，在 Spring 1.x 版本中，仅对发送 JMS 消息进行了封装，而接收 JMS 消息的封装在 Spring 2.0 版本中才被添加进来。

9.3.2 JMS编程模型

JMS 支持两种消息发送模式：点对点模式和发布-订阅模式。

在点对点模式下，消息的发送是一对一的，消息在两个实体间进行单向传递。点对点模式使用 Queue 来传递消息，如图 9-6 所示。

错误！

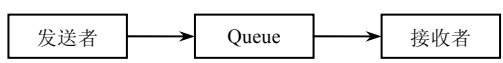


图 9-6

在发布-订阅模式下，消息的发送是一对多的。消息的发送者只需要发送消息，而凡是订阅了该消息的实体都可以接收到消息，如图 9-7 所示。

错误！

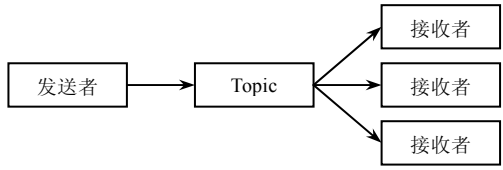


图 9-7

最初的 JMS 1.0 规范对这两种模式定义的接口是不同的，这意味着开发人员在编写 JMS 代码时，必须清楚地知道使用哪种模式来发送消息。幸运的是，在 JMS 1.1 标准中，这两种模式的接口完全统一了，开发人员不必关心究竟使用哪种模式来发送消息，而使用何种消息模式被定义在了服务器的相关配置中，也就是说，使用点对点模式还是发布-订阅模式只需要修改服务器的设置即可。JMS 1.1 中统一的 API 使得 JMS 代码更简单，更易于维护，并且代码复用性也更好。

表 9-1 显示了点对点模型和发布-订阅模型的主要接口，以及在 JMS 1.1 中统一的接口。

表 9-1

点对点模型	发布-订阅模型	JMS 1.1 的统一接口
QueueConnectionFactory	TopicConnectionFactory	ConnectionFactory
QueueConnection	TopicConnection	Connection
Queue	Topic	Destination
QueueSession	TopicSession	Session
QueueSender	TopicPublisher	MessageProducer
QueueReceiver	TopicSubscriber	MessageConsumer

使用 JMS 1.1 就无需关心两种模式的不同的接口，而只需使用统一的接口。

9.3.3 使用JMS API

为了在命令行程序中使用 JMS 测试，需要一个 JMS 服务器的实现。SUN 提供了一个免费的 mom4j 的 JMS 实现，它完整地支持 JMS 1.1 标准，可以从 http://mom4j.sourceforge.net 下载，不过，在使用 mom4j 之前，还需要手动下载 JMS 接口库和 mom4j 依赖的几个第三方库，然后用 Ant 编译源代码，具体请参考 README.txt 文件。读者可以从本书的配套光盘中找到已经编译好的 mom4j 1.1 版本，直接使用 lib 目录下的

mom4j.jar 即可。

在 Eclipse 中建立如图 9-8 所示的 JMS 工程。

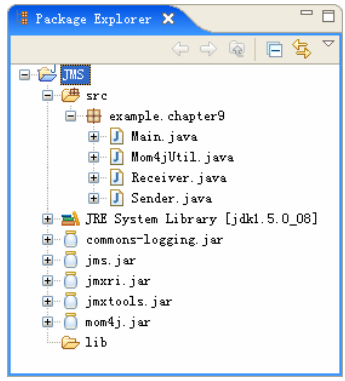


图 9-8

Mom4jUtil 负责初始化并启动 JMS 服务器，同时绑定 ConnectionFactory 和 Queue 到 JNDI 上，然后，在 Sender 类中，就可以发送 JMS 消息了。

```
        public class Sender extends Thread {
            public void run() {
                try {
                    // 查找 ConnectionFactory 和 Destination:
                    Context ctx = new InitialContext();
                    ConnectionFactory factory = (ConnectionFactory) ctx.lookup("Queue
ConnectionFactory");
                    Destination destination = (Destination) ctx.lookup("jms/queue");
                    for(;;) {
                        Connection connection = null;
                        try {
                            connection = factory.createConnection();
                            Session session = connection.createSession(false, Session.
AUTO_ACKNOWLEDGE);
                            MessageProducer producer = session.createProducer(destination);
                            String text = "Hello, it is " + new Date();
                            System.out.println("   Send: " + text);
                            Message message = session.createTextMessage(text);
                            producer.send(message);
                            producer.close();
                            session.close();
                        }
                        finally {
                            if(connection!=null)
                                connection.close();
```

```
                }
                Thread.sleep(1000 + (long)(Math.random() * 5000));
            }
        }
        catch(Exception e) {}
    }
}
```

Sender类以随机的间隔发送TextMessage消息。为了能够发送JMS消息，需要从JNDI获得 ConnectionFactory 和 Destination 对象。ConnectionFactory 将用于创建到 JMS 服务器的连接，Destination 则指定了消息的目的地，即消息将发向何处。

接收 JMS 消息和发送类似，也需要通过 ConnectionFactory 创建到 JMS 服务器的连接，Destination 指定从何处接收消息。此外，接收者必须实现 MessageListener 接口以便服务器回调。Receiver 类将接收到的 TextMessage 打印出来。

```
public class Receiver extends Thread implements MessageListener {
    public void run() {
        try {
            // 查找 ConnectionFactory 和 Destination:
            Context ctx = new InitialContext();
            ConnectionFactory factory = (ConnectionFactory) ctx.lookup("Queue
ConnectionFactory");
            Destination destination = (Destination) ctx.lookup("jms/queue");
            Connection connection = null;
            try {
                connection = factory.createConnection();
                Session session = connection.createSession(false, Session.AUTO_
ACKNOWLEDGE);
                MessageConsumer consumer = session.createConsumer(destination);
                // 设定 MessageListener 并开始接收消息:
                consumer.setMessageListener(this);
                connection.start();
                Thread.sleep(20000);
            }
            finally {
                if(connection!=null)
                    connection.close();
            }
        }
        catch(Exception e) {}
    }
    public void onMessage(Message message) {
        if(message instanceof TextMessage) {
            TextMessage text = (TextMessage) message;
```

```
            try {
                System.out.println("Receive: " + text.getText());
            }
            catch(JMSException e) {}
        }
    }
}
```

在 main 方法中启动 JMS 服务器，然后分别启动发送和接收线程。

```
public static void main(String[] args) throws Exception {
    Mom4jUtil.startJmsServer();
    new Sender().start();
    new Receiver().start();
    Thread.sleep(30000);
    System.exit(0);
}
```

可以看到如下的输出。

```
2006-10-19 15:53:21 org.mom4j.messaging.MessagingServer start
信息: admin server listening on port 8888
   Send: Hello, it is Thu Oct 19 15:53:21 CST 2006
Receive: Hello, it is Thu Oct 19 15:53:21 CST 2006
   Send: Hello, it is Thu Oct 19 15:53:25 CST 2006
Receive: Hello, it is Thu Oct 19 15:53:25 CST 2006
   Send: Hello, it is Thu Oct 19 15:53:27 CST 2006
Receive: Hello, it is Thu Oct 19 15:53:27 CST 2006
```

从上面的例子可以看到，直接使用 JMS API 发送和接收消息都是比较复杂的，并且没有很好地封装为组件。幸运的是，Spring 对 JMS 提供了非常简便的封装。通过使用 Spring 提供的 JmsTemplate 模版，操作 JMS 就易如反掌了。下面，我们看看在 Spring 中配置 JMS 组件的方法。

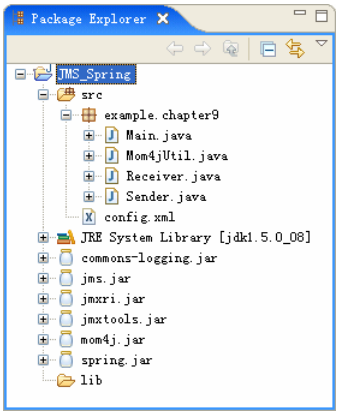


图 9-9

9.3.4 Spring如何封装JMS

为了演示在 Spring 环境下操作 JMS，我们建立一个 JMS_Spring 工程，如图 9-9 所示。

Spring 提供了 JmsTemplate 模版来简化 JMS 的操作。利用 JmsTemplate，Sender 只需被注入 JmsTemplate，发送的消息通过 MessageCreator 以回调的方式创建，整个发送过程非常简单。

```
public class Sender {
    private JmsTemplate jmsTemplate;
    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
    public void send(final String text) {
        System.out.println("  Send: " + text);
        jmsTemplate.send(new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(text);
            }
        });
    }
}
```

其所需的 JMS 资源全部定义在 XML 配置文件中。

```
    <bean id="jmsConnectionFactory" class="org.springframework.jndi.JndiObject
FactoryBean">
        <property name="jndiName" value="QueueConnectionFactory" />
    </bean>
    <bean id="jmsQueue" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="jms/queue" />
    </bean>
    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="jmsConnectionFactory" />
        <property name="defaultDestination" ref="jmsQueue" />
    </bean>
    <bean id="sender" class="example.chapter9.Sender">
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
```

对于接收消息的 Receiver 类，则仅实现了 MessageListener 接口，甚至没有注入任何 JMS 资源。

```
public class Receiver implements MessageListener {
    public void onMessage(Message message) {
        if(message instanceof TextMessage) {
            TextMessage text = (TextMessage) message;
            try {
                System.out.println("Receive: " + text.getText());
            }
            catch(JMSException e) {}
        }
    }
}
```

Spring 提供了 MessageListener 容器来包裹 MessageListener。通常情况下，使用 DefaultMessageListenerContainer 就足够了，只需要在 XML 配置文件中定义。

```
        <bean id="receiver" class="example.chapter9.Receiver" />
        <bean id="listenerContainer" class="org.springframework.jms.listener.
DefaultMessageListenerContainer">
            <property name="connectionFactory" ref="jmsConnectionFactory" />
            <property name="destination" ref="jmsQueue" />
            <property name="messageListener" ref="receiver" />
        </bean>
```

编写测试程序如下。

```
        public static void main(String[] args) throws Exception {
            Mom4jUtil.startJmsServer();
            ApplicationContext context = new ClassPathXmlApplicationContext("config.
xml");
            Sender sender = (Sender) context.getBean("sender");
            for(int i=0; i<10; i++) {
                sender.send("Hello, it is " + new Date());
                Thread.sleep(1000);
            }
            System.exit(0);
        }
```

运行程序，输出结果与直接使用 JMS 的工程类似，但是发送和接收 JMS 消息的编码被大大简化了。需要注意的一点是，必须保证 MessageListener 的实现类是线程安全的，因为 onMessage()方法可能被多个线程同时调用。

在 EJB 2.0 中使用消息驱动 Bean（Message-Driven Bean）也能大大简化 JMS 的编程，但是消息驱动 Bean 也是 EJB 的一种，必须运行在 EJB 容器中。相比之下，Spring 提供了以 POJO 形式的包裹 Bean，能够非常方便地处理 JMS 消息。

对于仅支持 JMS 1.02 版本规范的服务器，Spring 提供了一个兼容的 JmsTemplate102（表示 JMS 1.02 版本）模版类，JmsTemplate102 需要知道究竟使用 Queue 还是 Topic 方式，通过设置属性 pubSubDomain=true 就指定了 Topic 方式，如果不设定，则默认为 Queue 方式。

9.3.5 自动转化消息

大多数时候，除了简单的 TextMessage 外，需要发送的消息都应当被封装在 Java 类中，例如，一个电子邮件消息应该通过一个 MailMessage 对象来表示。通常，ObjectMessage 可以自动实现 Java 对象的序列化，不过，很多时候仍需要将消息以 BytesMessage 等其

他形式发送，为此，Spring 提供了一个 MessageConverter 接口来方便地实现 Java 类和 JMS 消息的转化。

```
public interface MessageConverter {
   Object fromMessage(Message message);
   Message toMessage(Object object, Session session);
}
```

同时，Spring 内置的 SimpleMessageConverter 已经能够满足大多数消息的转化，它支持 String 和 TextMessage、byte[]和 BytesMessage，以及 Map 和 MapMessage 之间的转化。要在发送消息时自动将 Java 对象转化为消息，可以调用 JmsTemplate 的 convertAndSend()。JmsTemplate 默认使用 SimpleMessageConverter 作为默认的 Message Converter，要编写一个自定义的 MessageConverter 也是极其容易的，这里不再给出示例代码。

9.3.6 同步接收消息

虽然绝大多数时候，JMS 消息都是异步传输的，但是某些时候也确实需要同步接收一条消息，JmsTemplate 提供了多个重载的 receive()方法，可以同步地接收消息。需要注意的是，使用 receive()方法要格外小心，在接收到消息之前，当前线程将被阻塞。

9.3.7 使用JMS发送E-mail通知

在 Web 应用程序中，常常需要给用户发送邮件通知，例如，注册成功的欢迎邮件、订单确认邮件，由于发送邮件是非常耗时的任务，对于需要实时发送的邮件，以同步的方式在一个 HTTP 请求中完成将影响用户浏览，在新的线程中发送则需要自己管理线程池等。这时，利用 JMS 提供的异步编程模型，借助 Spring 框架就可以在一个简单的 POJO 中非常方便地处理邮件发送的任务。

读者可以试着将 9.1 节讲述的邮件发送程序集成到 Receiver 类中，在 Sender 和 Receiver 之间使用 ObjectMessage 来传递邮件信息。

9.3.8 在服务器中发送消息

上面的例子中，我们使用的是独立的第三方 JMS 实现库，对于调试 JMS 程序来说是比较方便的。但是，一旦产品部署在服务器上，就应当使用服务器提供的 JMS 服务。常见的 JavaEE 服务器都提供了 JMS 的实现，少数高端 JavaEE 服务器还支持 JMS 集群。只需要对服务器做一定的配置，就可以直接在应用程序中使用 JMS 服务了。

在本节中，我们以 Resin 服务器为例，演示如何在服务器环境中使用 JMS。

首先，在 Eclipse 中新建 JMS_Servlet 工程，结构如图 9-10 所示。

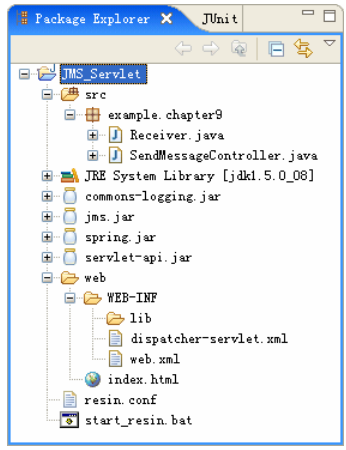


图 9-10

在 Resin 3.1 服务器中配置 JMS 相当简便，可以参考 Resin 根目录下的配置文件/conf/resin.conf。我们在工程根目录下创建一个新的 resin.conf，然后编写一个最简单的配置，只需加入下面的内容即可。

```
<resin xmlns="http://caucho.com/ns/resin"
      xmlns:resin="http://caucho.com/ns/resin/core">
   <log name="" level="info" path="stdout:"/>
   <cluster id="">
      <server id="">
         <http port="8080"/>
      </server>
      <resource jndi-name="jms/factory"
         type="com.caucho.jms.ConnectionFactoryImpl"/>
      <resource jndi-name="jms/queue"
         type="com.caucho.jms.memory.MemoryQueue"/>
      <resin:import path="${resin.home}/conf/app-default.xml"/>
      <host id="" root-directory=".">
         <web-app id="/" root-directory="."/>
      </host>
   </cluster>
</resin>
```

然后，编写 SendMessageController 以便接收用户输入，并将用户输入的内容作为 TextMessage 发送出去，为了简化程序，这里没有使用 MVC 架构，而是直接输出到 response 对象。

```
public class SendMessageController implements Controller {
    private JmsTemplate jmsTemplate;
    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
Response response) throws Exception {
        final String text = request.getParameter("text");
        response.setContentType("text/html;charset=UTF-8");
        response.setCharacterEncoding("UTF-8");
        PrintWriter writer = response.getWriter();
        if(text!=null && !text.equals("")) {
            jmsTemplate.send(new MessageCreator() {
                public Message createMessage(Session session) throws
JMSException {
                    return session.createTextMessage(text);
                }
            });
            writer.write("发送成功! <a href='index.html'>返回</a>");
            writer.flush();
        }
        else {
            writer.write("发送失败! <a href='index.html'>返回</a>");
            writer.flush();
        }
        return null;
    }
}
```

编写简单的 index.html，接收用户输入并将表单发送给 SendMessageController 处理。

```
<html><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>发送 JMS</title></head>
<body>
    <form name="form1" method="post" action="sendMessage.jms">
        输入消息：
        <input name="text" type="text" id="text" maxlength="255">
        <input type="submit" name="Submit" value="发送">
    </form>
</body>
</html
```

Receiver 类和 9.2 节的 JMS_Spring 工程中的完全相同，这里不再重复。最后一步便

是在 XML 配置文件中将它们全部装配出来。注意：由于是 Web 应用程序，XML 配置文件被放在/web/WEB-INF 目录下，命名为 dispatcher-servlet.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- JNDI 查找 -->
    <bean id="jmsConnectionFactory" class="org.springframework.jndi.
JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/jms/factory" />
    </bean>
    <bean id="jmsQueue" class="org.springframework.jndi.JndiObjectFactory
Bean">
        <property name="jndiName" value="java:comp/env/jms/queue" />
    </bean>
    <!-- 构造 JmsTemplate -->
    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="jmsConnectionFactory" />
        <property name="defaultDestination" ref="jmsQueue" />
    </bean>
    <!-- 接收消息 -->
    <bean id="receiver" class="example.chapter9.Receiver" />
    <bean id="listenerContainer" class="org.springframework.jms.listener.
DefaultMessageListenerContainer">
        <property name="connectionFactory" ref="jmsConnectionFactory" />
        <property name="destination" ref="jmsQueue" />
        <property name="messageListener" ref="receiver" />
    </bean>
    <!-- 默认使用 BeanNameViewResolver，只需将 Controller 的 Name 设置为 URL -->
    <bean name="/sendMessage.jms" class="example.chapter9.SendMessage
Controller">
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
</beans>
```

最后，在/web/WEB-INF/目录下配置好 web.xml。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
```

```
            <servlet-class>org.springframework.web.servlet.DispatcherServlet<
/servlet-class>
        </servlet>
        <servlet-mapping>
            <servlet-name>dispatcher</servlet-name>
            <url-pattern>*.jms</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
            <welcome-file>index.html</welcome-file>
        </welcome-file-list>
    </web-app>
```

运行 start_resin.bat，启动 Resin 服务器，然后打开浏览器，输入地址 http://localhost:8080/，如图 9-11 所示。

单击“发送”按钮，就将表单发送给 SendMessageController 处理，若发送成功，将显示发送成功的页面，如图 9-12 所示。

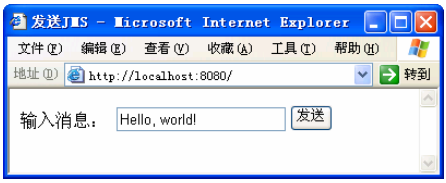


图 9-11

图 9-12

在 Resin 的控制台可以看到 Receiver 对象已经打印出了接收到的消息。

```
[2006/11/07 17:33:34.703] Servlet 'dispatcher' configured successfully
Receive: Hello, world!
```

如果这是一个在线聊天的应用，Receiver 就可以将消息转发给其他用户。

对于其他 JavaEE 服务器，请读者参考相应的说明文档，只要正确设置了 JNDI 名称，应用程序的代码不用更改就可以直接部署在相应的服务器上。

9.4 集成JMX

JMX 是 Java Management eXtensions 的缩写，JMX 为监控和管理 Java 应用程序提供了一个标准的接口和服务，使得不借助第三方软件，就能直接通过 Java 客户端来对 JVM 实施监控和管理。Spring 提供了对 JMX 很好的支持，能够使我们更方便地基于 JMX 对应用程序进行管理。

9.4.1 JMX概述

JMX 提供了标准的方式来管理 Java 应用程序，包括 Java 应用程序本身的对象、JVM 虚拟机状态、网络等一系列资源。使用 JMX 来管理 Java 应用程序时，被管理的 Java 应用程序和管理工具可以运行在不同的机器上，通过网络实现远程管理。

传统的应用程序会使用专门的软件来实现管理和监控，它们通常使用本地库嵌入到 JVM，并且有自己专用的协议，开发比较困难。很多应用程序也会自己编写管理界面，实现一套自定义的管理系统，不过，借助于 JMX，实现应用程序管理只需要关心如何收集被管理的数据即可，无需为管理程序开发界面，因为凡是符合 JMX 标准的管理程序都可以直接连接到被管理的应用程序，我们需要做的仅仅是按照 JMX 规范提供一些被管理和监控的类。

JMX 主要包括 JSR 3 和 JSR 160 标准，这两个 JSR 分别定义了被管理的资源和代理，以及实现远程管理的转化器和连接器。

由于 JMX 也是一个规范，SUN 只定义了接口，具体实现由各厂商自己完成。WebLogic、JBoss 等 JavaEE 服务器都有自己的 JMX 实现。在 Java 5 之前，如果要在自己的应用程序中使用 JMX，要么选择上述带有 JMX 实现的 JavaEE 服务器，要么集成一个开源的 JMX 实现。从 Java 5 开始，SUN 已经把 JMX 直接集成到了 JDK 中。使用 Java 5 平台就意味着可以直接使用 JMX 来实现应用程序的监管。

Java 5 平台还自带了一个 Swing 界面的 JConsole 管理程序，可以连接到任何符合 JMX 标准的 Java 应用上，对其进行监管，这样，我们就不必开发自己的管理界面。此外，还有许多第三方的 JMX 管理程序可供选择。

JMX 由 Instrumentation Level、Agent Level 和 Manager Level 三层实现，其架构如图 9-13 所示。

错误！

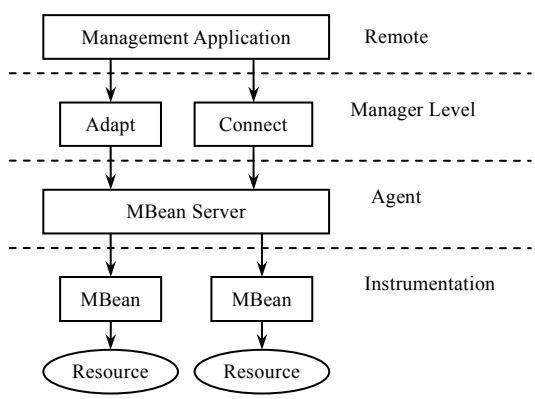


图 9-13

Instrumentation Level 包含被管理的 MBean，由开发人员或 JMX 提供商实现，Java 5 已经包含了几个有用的 MBean，可以对 JVM 状态进行监控，例如，内存使用情况、线程数量、CPU 使用时间、平台信息等。Agent Level 是访问 MBean 的代理，由 JMX 提供商实现。Manager Level 通过不同的 Adaptor 和 Connector 允许 JMX 管理软件以多种协议连接到 Agent，Java 5 JMX 内置的标准协议是 RMI，此外，还可以采用其他协议，如 HTTP 等。

9.4.2 手动注册MBean

为了能使用 JMX 远程管理应用程序，我们仍首先编写一个可以独立运行的 JMX 应用，稍后再将它集成到 Spring 环境中。现在，在 Eclipse 中新建如图 9-14 所示的 JMX 工程。

我们需要提供 JMX 所管理的 MBean，因此，第一步是编写自己的 MBean。这里，我们准备实现一个最简单的名为 Hello 的 MBean。首先，我们定义一个 HelloMBean 接口，声明一个 name 属性和 sayHello()方法。

```
public interface HelloMBean {
    String getName();
    void setName(String name);

    void sayHello();
}
```

然后，编写实现类 Hello。

```
public class Hello implements HelloMBean {
    private String name = "world";

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

    public void sayHello() {
        System.out.println("Hello, " + name);
    }
}
```

现在，我们已经有了一个 Mbean。但是管理程序并不直接和 MBean 打交道，而是通过 MBeanServer 和 MBean 的名称来访问 MBean。因此，下一步就要将这个 MBean 注册到 MBeanServer 中。前面已经提到了，Java 5 平台已经提供了一个内置的 MBeanServer，我们可以直接通过 ManagementFactory.getPlatformMBeanServer()方法获得该 Server，然

后注册我们自己的 Hello。

```
public static void main(String[] args) throws Exception {
    MBeanServer server = ManagementFactory.getPlatformMBeanServer();
    server.registerMBean(new Hello(), new ObjectName("test:name=Hello"));
    // 无限等待以便 JMX 管理程序连接:
    Thread.sleep(Long.MAX_VALUE);
}
```

虽然也可以通过 MBeanServerFactory 的 createMBeanServer()创建一个新的 Mbean Server，但是，在一个应用程序中创建多个 MBeanServer 是没有什么意义的，所以我们只需要使用 JVM 内置的这个 MBeanServer 就可以了。

为了能够让 JMX 管理程序连接到我们的应用程序，就必须提供合适的连接器和转化器。JMX 规范允许使用多种协议来实现远程连接，而最简单的 Java 5 JMX 内置的协议是使用 RMI 远程调用协议。要启动该 RMI 连接器，我们需要在应用程序启动时添加必要的参数。因此，选择 Eclipse 菜单“Run”→“Run...”，在弹出的对话框中新建一个 Java Application，在“Main”面板中填入 Project：JMX；Main class：example.chapter9.Main，然后切换到 Arguments 面板，填入以下参数，如图 9-15 所示。

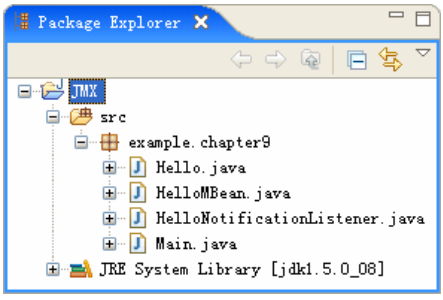


图 9-14

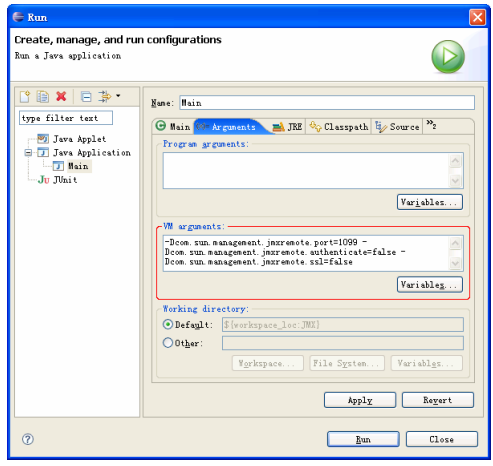


图 9-15

参数“com.sun.management.jmxremote.port”是 RMI 的监听端口，authenticate=false 指示无需认证，ssl=false 表示无需 SSL 安全连接。在这个测试应用程序中，这样可以最大限度地简化配置。在实际应用中，应当考虑使用口令保护和 SSL 安全连接。具体配置方式请参考 http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html，这里不再多述。

如果要在命令行模式下启动应用程序，请输入：

```
java -Dcom.sun.management.jmxremote.port=1099 -Dcom.sun.management.
jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false example.
chapter9.Main
```

应用程序启动后，就可以等待 JMX 管理程序连接。在 JDK 5.0 的 bin 目录下找到 jconsole.exe 并运行，JConsole 会自动弹出一个“连接到代理”对话框，如图 9-16 所示。

在对话框中选择“远程”面板，输入主机名或 IP：localhost，端口：1099，然后单击“连接”按钮，就可以连接到我们在 Eclipse 中启动的 JMX 应用程序。JConsole 自动显示如下“摘要”页，如图 9-17 所示。

图 9-16

图 9-17

还可以在“内存”、“线程”等页中查看更详细的信息，通过这些信息我们就可以随时监控 JVM 的运行状态。下面我们切换到“MBean”页，在左侧找到 Hello，就可以看到右侧的属性值，如图 9-18 所示。

在此可以任意修改 Hello 的属性，并且修改会直接反映到我们编写的 Hello 中。例如，将 Name 的值从“world”改为“spring”。

如果我们选择“操作”面板，JConsole 就会自动列出 Hello 可以调用的所有操作，如图 9-18 所示。

图 9-17

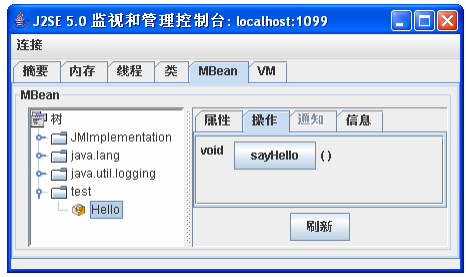


图 9-18

单击“sayHello”按钮，如果调用成功，JConsole 会提示“成功调用方法”。再回到

Eclipse 的控制台，可以看到我们编写的 Hello 已经打印出了“Hello，spring”字符串。这说明通过 JConsole 进行的调用实际上已经映射到了远程的应用程序中。

在一个实际应用程序中，完全可以将应用程序运行时需要的配置信息放在 MBean 中，例如，SMTP 的配置信息，这样我们就不必采用修改配置文件，再重新启动服务器这样烦琐的操作了，只需要通过 JConsole 修改 MBean，就可以立刻将配置信息更新。还可以在 MBean 中定义一系列方便的操作，例如，向管理员发送邮件报告等。采用 JMX 后，也不必针对运行期修改配置信息而专门编写一个管理页面，只需要利用标准的 JMX 管理程序就可以完成对远程应用程序的管理，大大简化了管理任务。

当 MBean 的状态改变后，例如，修改了某个属性的值，对这个 MBean 感兴趣的其他组件就应该能够得到通知，然后做一些必要的操作。对此，JMX 规范定义了 Notification 机制，凡是实现了 NotificationListener 接口的组件都可以被注册到 MBean 中，一旦该 MBean 的状态进行了更改，就能获得相应的通知。下面的 HelloNotificationListener 能够在 MBean 的属性修改后将属性值打印出来。

```
        public class HelloNotificationListener implements NotificationListener {
            public void handleNotification(Notification notification, Object handback) {
                if(notification instanceof AttributeChangeNotification) {
                    AttributeChangeNotification acn = (AttributeChangeNotification)
notification;
                    System.out.println("Attribute " + acn.getAttributeName()
                            + " changed: " + acn.getOldValue()
                            + " -> " + acn.getNewValue());
                }
            }
        }
```

除了实现 NotificationListener 之外，作为 MBean 本身还必须在属性被修改后向所有的 NotificationListener 发出通知。直接从 NotificationBroadcasterSupport 派生就可以实现该功能。我们修改 Hello 如下。

```
        public class Hello extends NotificationBroadcasterSupport implements
HelloMBean {
            private String name = "world";
            private long sequenceNumber;

            public String getName() { return name; }
            public synchronized void setName(String name) {
                String oldValue = this.name;
                this.name = name;
                sendNotification(new AttributeChangeNotification(
                        this,                       // 哪个对象的属性被更新了
```

```
                sequenceNumber++,           // 序列号
                System.currentTimeMillis(),  // 时间戳
                "Attribute 'name' changed!", // 描述
                "name",                     // 属性名称
                String.class.getName(),      // 属性的类型
                oldValue,                   // 更新前的属性值
                name                        // 更新后的属性值
            ));
        }

        public void sayHello() {
            System.out.println("Hello, " + name);
        }
    }
```

在修改了 name 属性后，Hello 就会广播一个 AttributeChangeNotification，凡是注册的 NotificationListener 都将收到这一通知。在应用程序中，注册过程需要手动完成，因此修改 main()方法。

```
public static void main(String[] args) throws Exception {
    Hello hello = new Hello();
    MBeanServer server = ManagementFactory.getPlatformMBeanServer();
    server.registerMBean(hello, new ObjectName("test:name=Hello"));
    hello.addNotificationListener(new HelloNotificationListener(), null, null);
    Thread.sleep(Long.MAX_VALUE);
}
```

再次运行应用程序，然后在 JConsole 中连接，修改 Hello 的 name，可以在 Eclipse 的控制台看到 HelloNotificationListener 打印出的信息。

```
Attribute name changed: world -> spring
```

细心的读者还可以在 JConsole 的 MBean 页中发现，“Hello”MBean 的“通知”页已变为可用，单击“订阅”还可以实现远程订阅。

9.4.3 在Spring中集成JMX

在上面的例子中，我们是以手动方式来配置 Mbean 和 NotificationListener 的。如果按照 Spring 的 IoC 哲学，这些 Mbean 和 NotificationListener 对象都应该作为组件在 XML 配置文件中装配起来。因此，在 Spring 中集成 JMX 就变成了一件很简单的事情：利用 Spring 提供的 IoC 容器，将这些 JMX 组件装配起来即可。

我们复制一份 JMX 工程，重新命名为 JMX_Spring，结构如图 9-19 所示。

和上一个版本有所不同，我们对 HelloNotificationListener 做了扩展，为其增加了一个 NotificationFilter 接口。

```
public class HelloNotificationListener
    implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        AttributeChangeNotification acn = (AttributeChangeNotification)
notification;
        System.out.println("Attribute " + acn.getAttributeName()
                + " changed: " + acn.getOldValue()
                + " -> " + acn.getNewValue());
    }

    public boolean isNotificationEnabled(Notification notification) {
        return (notification instanceof AttributeChangeNotification);
    }
}
```

NotificationFilter 允许过滤掉不感兴趣的通知，例如，我们只对来自属性变化的通知感兴趣，因此仅允许 AttributeChangeNotification 类型的通知。

下一步就是在 XML 配置文件中装配所有的 Bean。首先我们需要声明 Hello Mbean、HelloNotificationListener、一个 RmiRegistry 和一个基于 RMI 的连接器。

```
<!-- MBean -->
<bean name="test:name=Hello" class="example.chapter9.Hello" />

<!-- 监听器 -->
<bean id="helloListener" class="example.chapter9.HelloNotificationListener" />

<!-- 启动 RmiRegistry -->
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistry
FactoryBean">
    <property name="port" value="1099"/>
</bean>

<!-- 启动基于 RMI 的 Connector -->
<bean name="connector:name=rmi"
    class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="serviceUrl" value="service:jmx:rmi://localhost/jndi/rmi:
//localhost:1099/myconnector" />
</bean>
```

注意 RMI Connector 的 URL 格式，前半部分表示将使用 RMI 连接 JMX，后半部分

是 Connector 的 JNDI 名称。

最后一步是使用 Spring 提供的 MBeanExporter 将所有的 MBean 自动注册到 Mbean Server 中。

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
   <property name="autodetect" value="true" />
   <property name="notificationListenerMappings">
      <map>
         <entry key="test:name=Hello" value-ref="helloListener" />
      </map>
   </property>
</bean>
```

Spring 总是试图自动完成尽可能多的功能。由于我们指定了 autodetect 为 true，Spring 就自动查找可用的 MBeanServer，然后扫描 XML 中定义的所有的 Bean，凡是 name 符合“xyz:name=abc”格式的 Bean 均被认为是一个 MBean，并自动注册。而属性 notificationListenerMappings 可以将一个 NotificationListener 和一个指定的 MBean 名称关联。

如果有多个 NotificationListener 需要同时注册到一个 MBean 中，此时，以 MBean 的名称为 Key 的 Map 就无能为力了。这种情况下，需要使用另一个 notificationListeners 属性，该属性的优先级较 notificationListenerMappings 高，还可以同时关联 NotificationFilter。

```
<property name="notificationListeners">
   <list>
      <bean
         class="org.springframework.jmx.export.NotificationListener Bean">
          <property name="notificationListener" ref="helloListener"/>
          <property name="notificationFilter" ref="helloListener"/>
          <property name="mappedObjectName" value="test:name=Hello" />
      </bean>
   </list>
</property>
```

由于 RmiRegistry 和 RmiConnector 是应用程序启动后绑定到 MBeanServer 的，因此，无需为 JVM 增加启动参数，直接在 main()方法中启动 Spring 容器即可。

```
public static void main(String[] args) {
   new ClassPathXmlApplicationContext("config.xml");
}
```

运行该测试程序，利用 JConsole 连接时，切换到“高级”页，在 JMX URL 中填入 Connector 的 serviceUrl 属性即可连接，如图 9-20 所示。

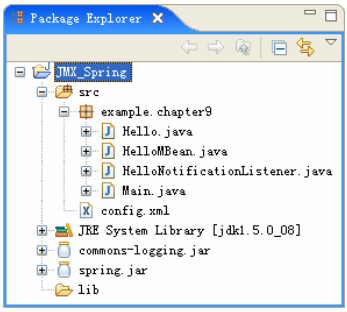


图 9-19

图 9-20

如果要添加其他协议，例如，以 HTML 格式来实现 JMX 管理，只需要相应的 Adaptor 即可。SUN 提供了一个 jmxtools.jar，包含了 HtmlAdaptorServer，从 http://java.sun.com/javase/ technologies/core/mntr-mgmt/javamanagement/download.jsp 下载该 jar 包后，向 Spring 的 XML 配置文件添加如下配置即可启动 HTML 支持。

```
<bean name="adaptor:name=html" class="com.sun.jdmk.comm.HtmlAdaptorServer"
    init-method="start">
    <property name="port" value="8080" />
</bean>
```

在浏览器中运行效果如图 9-21 所示。

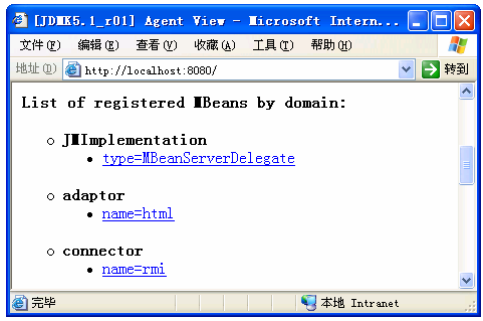


图 9-21

9.5 访问EJB

虽然 Spring 提倡使用轻量级的 JavaBean 组件，而反对使用 EJB 这种重量级的、复杂的组件，因为通过 Spring 这种轻量级框架，普通的 JavaBean 组件同样可以获得声明式事务处理的强大功能。但是，很多时候，尤其是与一个现有的 JavaEE 系统集成时，仍

然不可避免地需要访问已存在的 EJB 组件。Spring 也封装了用于访问 EJB 组件的类，以便简化对 EJB 的访问。

要访问一个已有的 EJB 组件，我们必须首先在 EJB 容器中部署一个 EJB 组件。本节以 JavaEE 1.4 SDK 为例，演示如何调用一个 Greeter 的无状态 SessionBean 组件。要运行本节的例子，请从 http://java.sun.com/javaee/downloads/previous/下载 Sun Java System Application Server PE 8.2 和 Samples Bundle 示例应用程序，或者从本书配套光盘中获得。Sun Java System Application Server PE 8.2 是 JavaEE 1.4 的参考实现，可以免费获得，并可免费用于开发和部署。安装好 Sun Java System Application Server PE 8.2 并解压缩 Sample Bundle 示例程序，确保 8080 端口没有被其他服务程序（例如，IISR、esin、Apache 等）占用，然后单击“开始”→“程序”→“Sun Microsystems”→“Application Server PE”→“Start Default Server”，启动默认的服务器。待服务器启动成功后，再单击“开始”→“程序”→“Sun Microsystems”→“Application Server PE”→“Deploytool”，启动部署工具，选择菜单“File”→“Open...”，打开示例应用程序 samples/ejb/stateless/apps/simple/stateless-simple.ear 后如图 9-22 所示。

选中“HelloWorld”，然后选择菜单“Tools”→“Deploy...”，准备将这个 EAR 应用程序部署到 Sun Java System Application Server PE 8.2 中，如图 9-23 所示。

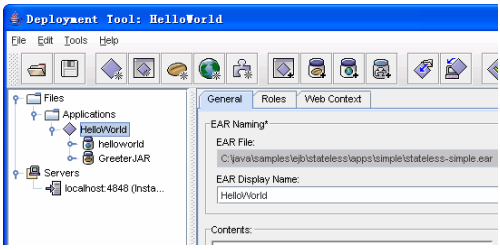


图 9-22

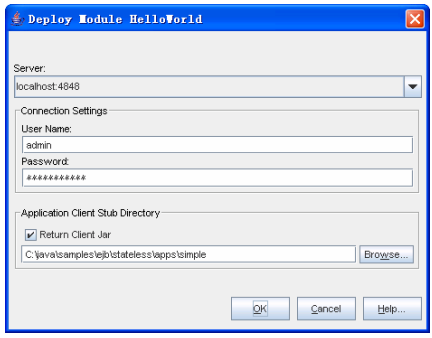


图 9-23

确保选中了“Return Client Jar”，这个 jar 文件将被客户端引用。然后在弹出的提示框中选择“Save and Deploy”，等待几秒钟，待部署成功后，就可以在“Server”→“localhost”中看到已部署的应用程序“stateless-simple”，如图 9-24 所示。

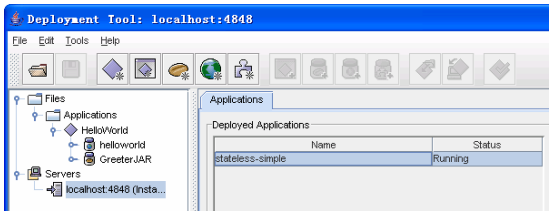


图 9-24

这个企业应用程序包含了一个简单的 Greeter EJB 组件，其接口仅包含一个 getGreeting 的方法。

```
// File: samples/ejb/stateless/apps/simple/simple-ejb/
//       src/java/samples/ejb/stateless/simple/ejb/Greeter.java
public interface Greeter extends javax.ejb.EJBObject {
    public String getGreeting() throws java.rmi.RemoteException;
}
```

下面，我们分别以传统的方式和“Spring”方式来访问这个 Greeter EJB 组件。

9.5.1 以传统方式访问EJB

我们首先来看看使用传统的方式来访问 EJB 组件需要编写的代码。打开 Eclipse，新建 CallEjb 工程，结构如图 9-25 所示。

其中，stateless-simpleClient.jar 是由部署工具返回的用于客户端访问的 jar 包，j2ee.jar 可以在 Sun Application Server PE 8.2 的安装目录下的 lib 目录中找到。

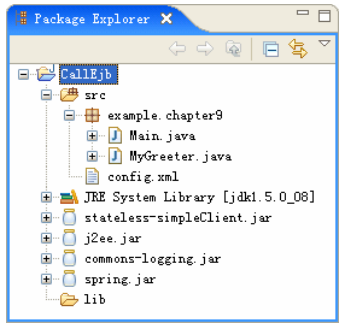


图 9-25

为了调用 Greeter EJB 组件的 getGreeting()方法，我们不得不在 main()方法中编写以下代码。

```
public static void main(String[] args) {
    GreeterHome home = getGreeterHome();
    try {
        Greeter greeter = home.create();
        System.out.println(greeter.getGreeting());
    }
    catch(RemoteException e) {
        e.printStackTrace();
    }
    catch(CreateException e) {
```

```
                e.printStackTrace();
            }
        }

        // 返回 Home 接口:
        private static GreeterHome homeCache = null;
        private static GreeterHome getGreeterHome() {
            if(homeCache==null) {
                InitialContext ctx = null;
                try {
                    System.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.
jndi.cosnaming.CNCtxFactory");
                    System.setProperty(Context.PROVIDER_URL, "iiop://localhost:3700");
                    ctx = new InitialContext();
                    Object ejbHome = ctx.lookup("greeter");
                    homeCache = (GreeterHome) PortableRemoteObject.narrow(ejbHome,
GreeterHome.class);
                }
                catch(Exception e) {
                    throw new RuntimeException(e);
                }
                finally {
                    if(ctx!=null) {
                        try {
                            ctx.close();
                        }
                        catch(NamingException e) {}
                    }
                }
            }
            return homeCache;
        }
```

其中，尽管我们已经封装了返回 Home 接口的方法，但该方法仍有 20 多行代码，在真正调用 Greeter 接口时，仍需要捕获 RemoteException 和 CreateException 异常。这种查找，使用，捕获异常，然后释放资源的方式和依赖注入思想是背离的。我们希望 Greeter 接口能直接被注入需要的 Bean 中，至于如何获得这个接口，组件通常是不关心的。

9.5.2 在Spring中访问EJB

幸运的是，在 Spring 中，访问 EJB 也变得简单了。Spring 提供了两个 FactoryBean，分别用来访问本地 EJB 组件和远程 EJB 组件。

（1）LocalStatelessSessionProxyFactoryBean：访问本地 EJB 组件。

（2）SimpleRemoteStatelessSessionProxyFactoryBean：访问远程 EJB 组件。

两种 EJB 组件的区别在于，访问本地 EJB 组件时，客户端和 EJB 容器必须运行在同一个 JVM 之内，而远程 EJB 则没有这个限制，但是，由于使用了远程调用，因此降低了速度。

对于上面的远程 Greeter EJB 组件，为了在 Spring 中访问它，需要首先在 XML 配置文件中定义它。

```
        <bean id="greeter" class="org.springframework.ejb.access.SimpleRemote
StatelessSessionProxyFactoryBean">
            <property name="jndiName" value="greeter" />
            <property name="businessInterface" value="samples.ejb.stateless.
simple.ejb.Greeter" />
        </bean>
```

注入 jndiName 和 businessInterface，就可以获得一个 Greeter 接口的引用。然后，只需要 main()方法中简单地编写几行代码。

```
        public static void main(String[] args) {
            System.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.
cosnaming.CNCtxFactory");
            System.setProperty(Context.PROVIDER_URL, "iiop://localhost:3700");
            ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
            Greeter greeter = (Greeter) context.getBean("greeter");
            try {
                System.out.println(greeter.getGreeting());
            }
            catch(RemoteException e) {}
        }
```

但是，由于从 Spring 容器中获得的“greeter”是一个远程 EJB 接口，因此，还必须在代码中捕获 RemoteException，这似乎还没有彻底地简化访问 EJB 的方法。不过，要消除 RemoteException 也是相当容易的。我们只需要定义一个自定义的 MyGreeter 接口，该接口和 Greeter 接口具有相同的方法签名，唯一不同的是不抛出 RemoteException。

```
        package example.chapter9;
        public interface MyGreeter {
            String getGreeting();
        }
```

然后，对 XML 配置文件稍做修改，将 businessInterface 改为“example.chapter9.

MyGreeter”。

```
        <bean id="greeter" class="org.springframework.ejb.access.SimpleRemote
StatelessSessionProxyFactoryBean">
            <property name="jndiName" value="greeter" />
            <property name="businessInterface" value="example.chapter9.MyGreeter" />
        </bean>
```

于是，main()方法被进一步简化为 5 行代码。

```
        public static void main(String[] args) {
            System.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.
cosnaming.CNCtxFactory");
            System.setProperty(Context.PROVIDER_URL, "iiop://localhost:3700");
            ApplicationContext context = new ClassPathXmlApplicationContext
("config.xml");
            MyGreeter greeter = (MyGreeter) context.getBean("greeter");
            System.out.println(greeter.getGreeting());
        }
```

实现这一魔法的原理仍然是 AOP，Spring 根据方法签名将调用委托给实际的 EJB 远程接口，只要保证自定义接口与 EJB 远程接口具有相同的方法签名（除了抛出任何异常的申明）。如果远程接口抛出了 RemoteException 怎么办？不用担心，Spring 会自动将 RemoteException 转化为 RemoteAccessException，这个 RemoteAccessException 异常是一个 RuntimeException，因此无需在客户端捕获该异常。

由于 EJB 组件以 JavaBean 的形式被定义在 XML 配置文件中，实现依赖注入便是小菜一碟了。

对于具有 Local 接口的 EJB 组件，其配置方式与具有 Remote 接口的 EJB 组件完全相同，并且，由于 Local 接口不抛出 RemoteException，所以自定义接口的步骤也免了。不过，需要注意的一点是，在使用 LocalStatelessSessionProxyFactoryBean 时，最好设置 Bean 的 **lazy-init** 为“true”，这是因为 Spring 程序和 EJB 组件都将部署在同一个 JVM 中，当 Spring 容器启动时，如果 EJB 容器还没有将 EJB 组件绑定到 JNDI，则会造成 Spring 查找 Local EJB 失败。设置 lazy-init="true"就是为了让 EJB 容器有充足的时间来完成 EJB 组件的初始化工作。对于远程 EJB 来说，总是应当首先启动远程 EJB 容器，除非 EJB 容器和 Web 容器在同一个 JVM 中，才需要设置 lazy-init 为“true”。

9.5.3 Spring中访问EJB的限制

在 Spring 中，由于仅提供了访问无状态 SessionBean 的方法，对于有状态 SessionBean

和 EntityBean，仍然只能使用传统的 JNDI 查找。不过，通常一个设计良好的 JavaEE 系统不应该暴露 EntityBean，并且总是应当提供一个无状态 SessionBean 作为 Façade（门面）来简化客户端对中间层的访问，因此，Spring 提供的对无状态 SessionBean 的支持在大多数情况下已经足够了。

Spring 还提供了对开发 EJB 的一些辅助类，如 AbstractStatelessSessionBean，虽然 Spring 不提倡 EJB，但仍然通过这些辅助类来倡导好的编程实践，例如，将业务逻辑放入 EJB 之后的 JavaBean 中，只把事务和远程调用这些任务交给 EJB 来完成。不过，我个人认为，如果使用 EJB 作为中间层，则没有必要再引入 Spring 的库。通常，IDE 工具甚至是一些 JavaEE 的服务器厂商都提供了简化 EJB 的方式，例如，BEA WebLogic 就提供了 GenericSessionBean 类来简化 EJB 的实现。在 EJB 3.0 中，就只需要编写 Bean 本身的 POJO 实现，因此，Spring 提供的辅助类就更没有必要了。

9.6 动态语言支持

Spring 2.0 开始提供对动态语言的支持，可以使用动态语言来定义类和对象，然后通过在 XML 配置文件中定义使用的动态语言，Spring 容器就可以完全透明地实例化那些使用动态语言编写的类，并和现有的 Java 类配合实现依赖注入。目前，Spring 2.0 支持 Jruby、Groovy 和 BeanShell 这 3 种动态语言。

（1）JRuby：JRuby 是 Ruby 语言的纯 Java 实现，可以将 Ruby 编译为 Java 字节码并动态执行。

（2）Groovy：Groovy 是来自 Java2 平台的敏捷动态语言，拥有很多如 Python、Ruby、Smalltalk 这类语言的特征，并以 Java 风格的语法展现给 Java 开发者。

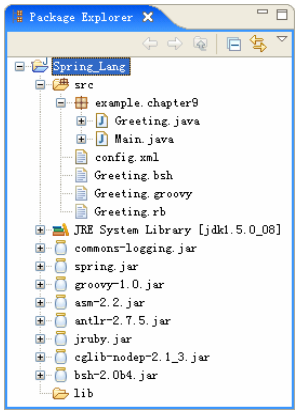


图 9-26

（3）BeanShell：BeanShell 是一个用 Java 实现的小型且免费的嵌入式 Java 解释器，支持动态执行标准的 Java 语法，并进行了扩展，带来一些常见的脚本的便利，如在 Perl 和 JavaScript 中宽松的类型等。

我们仍以一个具体的例子来演示如何在 Spring 2.0 框架中集成动态语言。在 Eclipse 中新建一个 Spring_Lang 工程，结构如图 9-26 所示。

请注意，我们需要的是使用动态语言来实现 Spring 应用程序的某个部分的功能，而不是直接使用动态语言的全部功能（例如，在命令行下解释执行 Ruby 语句），因此，接口的定义必不可少，而接口

的实现方式可以不是 Java 类，而是由动态语言来实现。

我们定义了一个 Greeting 接口，用于返回一个表示问候语的字符串。

```
public interface Greeting {
    String getMessage();
}
```

在 Spring 应用程序中，我们从容器中获取某个实现了此接口的 Bean，就可以直接使用它了，而这个 Bean 的具体实现则不一定是 Java 类。这里，我们分别使用 Jruby、Groovy 和 BeanShell 这 3 种动态语言来实现它。

Greeting.rb 是 JRuby 的实现。

```
# file: Greeting.rb
require 'java'
include_class 'example.chapter9.Greeting'

class JRubyGreeting < Greeting
  def getMessage()
    return @@msg
  end

  def setMessage(msg)
    @@msg = msg
  end
end
JRubyGreeting.new
```

Greeting.groovy 是 Groovy 的实现。

```
// file: Greeting.groovy
import example.chapter9.Greeting;

class GroovyGreeting implements Greeting {
    String msg;

    String getMessage() {
        return msg;
    }

    void setMessage(String msg) {
        this.msg = msg;
    }
}
```

Greeting.bsh 是 BeanShell 的实现。

```
// file: Greeting.bsh
String msg;

String getMessage() {
    return msg;
}

void setMessage(String aMsg) {
    msg = aMsg;
}
```

下一步是在 XML 配置文件中定义使用动态语言实现的这些 Bean。为了使用动态语言，我们必须引入 lang 命名空间，然后根据动态语言的类型，分别使用<lang:jruby />、<lang:groovy />和<lang:bsh />来定义 Bean，script-source 则用于指定文件路径，“classpath:”是最常用的前缀。除了 Groovy 外，使用 JRuby 和 BeanShell 还必须指定实现接口，因为 Spring 要通过 JDK 动态代理为它们实现这个接口。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/lang
      http://www.springframework.org/schema/lang/spring-lang-2.0.xsd"
>
    <!-- JRuby 实现的 Bean -->
    <lang:jruby id="jruby" script-source="classpath:Greeting.rb"
        script-interfaces="example.chapter9.Greeting">
        <lang:property name="message" value="Hello, JRuby!" />
    </lang:jruby>

    <!-- Groovy 实现的 Bean -->
    <lang:groovy id="groovy" script-source="classpath:Greeting.groovy">
        <lang:property name="message" value="Hello, Groovy!" />
    </lang:groovy>

    <!-- BeanShell 实现的 Bean -->
    <lang:bsh id="bsh" script-source="classpath:Greeting.bsh"
        script-interfaces="example.chapter9.Greeting">
        <lang:property name="message" value="Hello, BeanShell!" />
    </lang:bsh>
```

```
</beans>
```

一旦定义好动态语言实现的 Bean，我们就可以像使用普通的 JavaBean 一样来使用它们，方法仍然是通过 Spring 的 IoC 容器获取 Bean 的实例。

```
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("config.
xml");
    System.out.println(((Greeting)context.getBean("jruby")).getMessage());
    System.out.println(((Greeting)context.getBean("groovy")).getMessage());
    System.out.println(((Greeting)context.getBean("bsh")).getMessage());
}
```

导入必要的 jar 文件，就可以直接运行测试程序了。在 Eclipse 中运行结果如下。

```
Hello, JRuby!
Hello, Groovy!
Hello, BeanShell!
```

除了编写脚本文件并在 XML 配置文件中配置外，Spring 也支持直接在 XML 配置文件中嵌入脚本，不过，这样会大大增加 XML 配置文件的复杂性，除非脚本特别简单，否则不要这么做。

```
<lang:jruby id="jruby"
    script-interfaces="example.chapter9.Greeting">
    <lang:inline-script>
    <![CDATA[
    require 'java'
    include_class 'example.chapter9.Greeting'
    class JRubyGreeting < Greeting
        def getMessage()
            return @@msg
        end
        def setMessage(msg)
            @@msg = msg
        end
    end
    JRubyGreeting.new
    ]]>
    </lang:inline-script>
    <lang:property name="message" value="Hello, JRuby!" />
</lang:jruby>
```

Spring 2.0 还提供了一个对动态语言的自动更新的支持，它可以跟踪动态语言编写的脚本文件，一旦检测到更改，就可以自动重新加载。默认情况下，该特性是关闭的，可

以通过为某个 Bean 指定 refresh-check-delay 来开启自动更新功能，例如，设置每 3 秒检测一次脚本文件的改动。

```
<lang:bsh id="bsh" script-source="classpath:Greeting.bsh"
   refresh-check-delay="3000"
   script-interfaces="example.chapter9.Greeting">
   <lang:property name="message" value="Hello, BeanShell!" />
</lang:bsh>
```

请注意，自动更新只能检测独立的脚本文件的变化，对于内嵌到 XML 配置文件中的脚本，Spring 不会检测其更改。

在某些情况下，例如，验证用户输入的表单，使用动态语言比直接用 Java 实现要来得简便。并且利用 Spring 提供的对动态语言脚本文件自动更新的支持，可以在应用程序运行期动态修改验证规则，而不必重新启动应用程序。

9.7 小结

本章主要介绍了如何使用 Spring 框架更加方便地访问系统服务，包括邮件服务、任务调度、JMS 消息服务和访问 EJB 组件。我们还介绍了 Spring 框架对动态语言的支持，包括 JRuby、Groovy 和 BeanShell。

Spring 对 JavaMail API 做了非常好的封装，通过 Spring，我们能更加方便地发送电子邮件，尤其是带有附件的 HTML 格式的电子邮件。在例子中，我们以 Gmail 为目标邮箱，读者还可以看到如何使用 SSL 来保证电子邮件的传输安全。

任务调度也是常见的系统服务之一。Spring 对 Timer 和 Quartz 的封装使得我们几乎不必关心具体的 API，只需要配置好相关的 Bean 即可。更重要的是，任务调度器本身被纳入 Spring 的 IoC 容器中，免去了我们手动编写初始化和销毁资源等复杂的代码，便于集中精力处理好任务逻辑本身。

JMS 也是 JavaEE 体系中非常重要且应用广泛的 API。Spring 2.0 同时封装了发送和接收 JMS 消息，并提供对 JMS 1.1 的支持。本章介绍的 JMS 编程也是以 JMS 1.1 为准。通过异步处理请求，有些时候能大大简化系统设计。

我们还介绍了如何在 Spring 框架中访问 EJB 组件。我们对比了手动编码和 Spring 封装的 ProxyFactoryBean 这两种方式，使用 Spring 封装的 EJB 访问，就可以使 EJB 组件被当作一个普通的 Bean 来使用，更便于应用程序内部设计的一致性。

对动态语言的支持也是 Spring 2.0 的一个新特性。在某些情况下，使用动态语言比直接用 Java 来实现要来得简便，这种特性可能会对某些有特殊需求的项目带来便利，不

过，通常的 Java 项目中对动态语言的应用相当少见。

Spring 提供的抽象层对这些底层服务做了良好的封装，让开发者能以 Bean 的形式来使用这些服务。不过，弄明白如何直接调用这些服务是非常重要的，只有在能够手动调用 API 编写代码的基础上，才能更好地掌握 Spring 的封装原理，从而提高组件的设计能力。

由于这些需要用到的资源大部分都以 Bean 的形式定义在了 XML 配置文件中，免去了常见的 JNDI 查找、异常处理等烦琐的编码，开发人员就可以集中精力处理业务逻辑，从而提高开发效率。

在第 10 章中，我们将讨论企业级应用程序中一个重要的话题：安全。我们将详细介绍如何使用基于 Spring 的 Acegi 安全框架来构建应用程序的安全模型，以确保应用程序的安全。

第 10 章

Spring Acegi安全框架

10.1 JavaEE安全概述

安全永远是 Web 应用程序必须要面对和解决的头等大事。绝大多数的 JavaEE 企业级应用程序都有不同程度的安全需求。通常来说，应用程序都需要保证几个基本的安全需求。首先，应用程序必须能够识别访问者的身份；其次，应用程序必须对 Web 资源提供安全保证，拒绝未经授权的访问；最后，在多层应用程序模型中，应用程序还要有能力对业务逻辑组件提供保护，以防止非法用户绕过 Web 层调用了受保护的方法。

安全系统包括两个明显的问题：认证和授权。认证就是识别用户身份，一个用户通过认证来告诉系统“我是谁”，从而确定用户的身份。认证有多种形式，通常可以使用用户名和口令实现认证，但是，还可以使用数字证书等形式实现更安全的认证。一旦确定了用户的身份，下一步就是要确定用户是否有权进行某种给定的操作，即授权，例如，是否能够在网站首页发布一篇新闻。

10.1.1 基于角色的权限控制

由于用户通常很多，对每个用户进行授权会非常麻烦，因此，基于角色的权限控制（Role Based Access Control，RBAC）就成为一种最流行的授权方式。角色是一种抽象的逻辑用户分组，代表具有同等权限的用户组。由于角色一般不多，如一个部门中，角色可以分为管理员、经理、员工等几类，资源和角色关联，每个用户根据自身的角色获得相应的权限，这样就大大简化了授权的逻辑。比如，对于部门的工资系统，每个员工都可以浏览自己的工资信息，但是无权给自己加薪，而经理则具有调整员工工资的权限（经理是否具有给自己加薪的权限则视不同公司而定），管理员具有为新员工创建工资记录、定期备份数据等系统维护的权限。若用户 A 从员工被提升为经理，则该用户的角色就从员工变为经理，从而获得了经理级别的权限。因此，在基于角色的权限系统中，授权不和用户直接挂钩，而是和角色挂钩，用户再通过和角色关联，最终获得相应的授权，如图 10-1 所示。

错误！

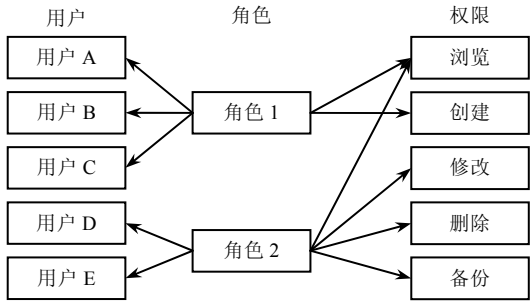


图 10-1

值得注意的是，用户和角色通常也不是简单的多对一关系，一个用户往往可以具有多个角色，例如，一个对计算机操作熟悉的员工被任命为管理员，就具有了员工和管理员这两个角色。

通常，不建议自己开发安全逻辑，因为开发防攻击的安全逻辑本身就不是一个简单的任务。此外，自定义的安全逻辑可能会导致较低的可重用性。在 Java 中，Java 验证与授权服务（Java Authentication and Authorization Service，JAAS）是标准的认证和授权服务，它建立在可插入的身份验证模块的基础上，通过定义插入验证模块的标准，系统管理员就能够插入和配置适当的身份验证服务，以满足安全需求却无需修改组件。不过，JAAS 使用的是现有的 Java 2 安全模型，它根据认证和授权来限制资源和代码的访问，不过，这些限制仍然局限在较低层次的系统资源，例如，读取文件或使用网络接口等。通常情况下，服务器环境下运行的代码都是可靠的且受信任的，因此，不太适合在服务器端应用 JAAS 授权机制。

10.2 Acegi安全框架

Acegi 是一个基于 Spring 开发的安全框架，为应用程序提供认证和授权的安全保护功能。该架构完全采用“Spring 方式”开发，包括依赖注入、AOP 拦截、针对接口编程等最佳实践。对于基于 Spring 的应用程序而言，Acegi 提供了一个非常有用的外置式的安全架构，能非常容易地集成到实际的应用程序中。

截至本书写作时，Acegi 已经发布了 1.0 正式版。不过，从 Acegi 开发过程来看，该架构从 0.5 版本到 1.0 版本的变化非常大，API 经过了大量的修改，甚至连包名都全部改过。本书采用的 Acegi 版本为最新的 1.0.3 版，请读者务必使用最新版本，以确保本章中的例子都能成功运行。

Acegi 的设计仍是一个基于角色的权限控制系统，它通过一系列可配置的组件构建了一个基于 Spring IoC 组件装配模式的安全框架。

Acegi 安全框架中也有 Principal 和 Credentials 的概念，它们可以是任何对象，并由 Authentication 对象封装，代表一个用户认证。正如前面介绍的，Principal 通常是用户名，Credentials 通常是口令，不过，在 Acegi 框架中，通常将 UserDetails 作为 Principal，除了存储了用户名外，UserDetails 还包含了用户角色等权限信息。此外，Authentication 对象还能存储一些与认证请求有关的附加信息，例如，用户的 IP 地址。

第二个问题是应用程序如何访问 Authentication 对象。一旦用户通过了认证，Authentication 对象就将存储在 HttpSession 对象中，不过，HttpSession 并非总是能被应用程序访问，因此，Acegi 使用 SecurityContext 来存储 Authentication，默认情况下，Acegi

使用 ThreadLocal 来实现 SecurityContext。在应用程序中，任何时候获取 Authentication 对象的方式就是调用 SecurityContextHolder.getContext().getAuthentication()。

有了 Authentication 对象，Acegi 安全框架再通过以下几个组件来确保被访问资源的安全，如图 10-2 所示。

错误！

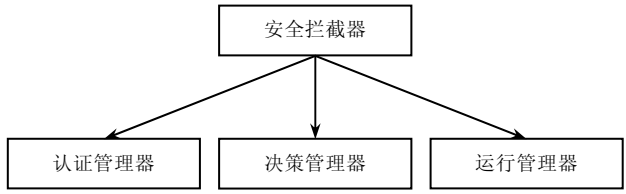


图 10-2

在访问任何受保护的资源时，安全拦截器都会首先截获该访问，然后通过认证管理器来确定用户是否已经通过了身份认证，紧接着，决策管理器将决定该用户是否有权限访问该资源，而运行管理器则可以用另一个身份替换当前用户的身份，从而允许访问系统内部更深处的受保护的资源。

通常，运行管理器在大多数应用中都不是必需的，一个最直观的例子是在 Windows 桌面上右键单击某个应用程序的快捷方式，例如，Eclipse，就可以看到菜单项“运行方式...”，如图 10-3 所示。

某些应用程序要求必须以管理员的身份运行，如果当前用户并不是管理员身份，就可以通过这种方式获得管理员身份来运行该应用程序。

图 10-3

在 Acegi 中，运行管理器可以不用配置，在本章中，我们也不会涉及运行管理器。我们的关注点主要放在认证管理器和决策管理器上。

现在，读者对 Acegi 安全框架有了一个整体的认识，我们还有一个疑问：Acegi 是如何与具体的应用程序整合的，它又到底保护哪些受保护的资源？

在一个 B/S 结构的企业级应用程序中，通常，Web 页面是整个应用程序与用户打交道的界面，因此，Acegi 首先要保护的便是 Web 页面，以确保未经授权的用户无法访问受保护的页面。在 JavaEE 平台上，Filter 具有对 URL 过滤的功能，因此，Acegi 对 Web 页面的安全保护正是通过 Filter 来实现的。

其次，多层应用程序中，核心的业务逻辑组件需要在方法调用的级别上进行安全保护，只有获得授权的用户才有权调用某些业务方法。由于 Acegi 本身也完全采用“Spring

方式”来开发和配置，很容易想到，对业务逻辑组件的方法级保护正是通过 AOP 实现的。

10.2.1 保护Web资源

对于 Web 应用程序而言，安全保护最重要的就是防止非授权用户访问受保护的页面。大多数手动实现安全保护的 Web 应用程序通常都使用 Filter 来过滤 URL，以实现授权访问。Acegi 安全框架对 Web 资源的保护也正是基于 Filter，不过，其设计更加灵活，很容易和现有的 Web 应用程序集成。

在本节中，我们首先在 Eclipse 中创建一个 Spring_Acegi 工程，它包含了几个简单的 JSP 页面，其结构如图 10-4 所示。

我们假定系统用户分为普通用户和管理员两类，访问规则如下。

（1）普通用户有权访问/user/目录下的所有资源，但无权访问/admin/目录下的任何资源。

（2）管理员有权访问/admin/和/user/目录下的所有资源。

（3）所有的用户（包括未登录的用户）都可以访问除/user/和/admin/之外的所有资源。

按照这个访问规则，我们开始配置 Acegi。

由于 Acegi 依赖 Filter 来实现对 Web 资源的保护，通常，我们需要多个 Filter 构成一个 FilterChain 来实现过滤，如图 10-5 所示。

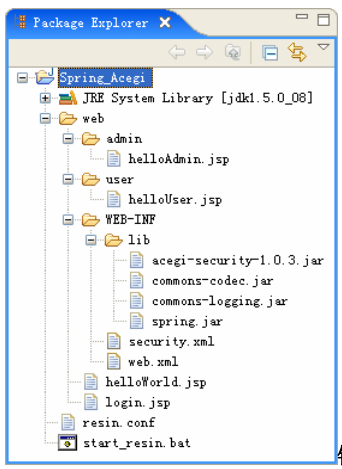
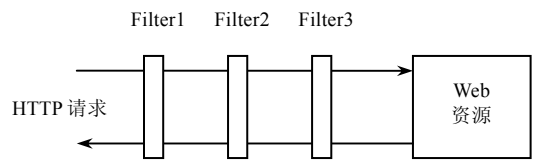


图 10-4



错误！

图 10-5

Acegi 虽然有多个 Filter，但是我们却只需要在 web.xml 中配置一个特殊的 Acegi Filter，其他的 Filter 通过依赖注入的方式在 Acegi 的 Spring XML 配置文件中定义。这个特殊的 Acegi Filter 可以被称为 FilterChainProxy，在其内部会依次调用每个注入的 Filter。

这样，就避免了在 web.xml 中配置多个 Acegi Filter 的麻烦。

我们将 Acegi 相关的 Bean 全部放入到 security.xml 中，然后在 web.xml 中配置 Listener 和 Filter。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- Acegi 配置文件的位置 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/security.xml</param-value>
    </context-param>

    <!-- 启动 Spring 容器 -->
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <!-- Acegi 过滤器 -->
    <filter>
        <filter-name>acegiFilterChain</filter-name>
        <filter-class>
            org.acegisecurity.util.FilterToBeanProxy
        </filter-class>
        <init-param>
            <param-name>targetClass</param-name>
            <param-value>
                org.acegisecurity.util.FilterChainProxy
            </param-value>
        </init-param>
    </filter>

    <!-- Acegi 过滤器 URL 映射 -->
    <filter-mapping>
        <filter-name>acegiFilterChain</filter-name>
        <url-pattern>*.jsp</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>acegiFilterChain</filter-name>
        <url-pattern>*.do</url-pattern>
    </filter-mapping>
```

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

请注意 acegiFilterChain 这个 Filter，我们指定了 targetClass 参数，在 Web 应用程序初始化时，acegiFilterChain 将从 Spring 容器中查找类型为 org.acegisecurity.util.FilterChainProxy 的 Bean，然后，Filter 将自身的任务委托给这个 Bean，即每当调用 Filter 执行过滤时，都会调用这个 Bean 的 doFilter()方法。

还可以指定 targetBean 参数，让 acegiFilterChain 直接从 Spring 容器中查找指定名称的 Bean。这样做唯一的缺点是 web.xml 和 security.xml 配置文件通过 Bean 的 id 关联起来了。

```
<filter>
    <filter-name>acegiFilterChain</filter-name>
    <filter-class>
        org.acegisecurity.util.FilterToBeanProxy
    </filter-class>
    <init-param>
        <param-name>targetBean</param-name>
        <param-value>filterChainProxy</param-value>
    </init-param>
</filter>
```

我们希望 acegiFilterChain 工作得更有效率一点，因此仅过滤*.jsp 和*.do 这两种 URL，不过滤图片、CSS 等静态资源。

现在，我们需要做的便是在 Acegi 的配置文件 security.xml 中配置好一系列的安全过滤器，以及认证管理器和决策管理器。

1．配置 AuthenticationManager

首先，我们来配置 Acegi 的认证管理器，即 AuthenticationManager，该组件的任务就是对用户进行认证。认证管理器通过识别 Principal（主体，通常是用户名）和 Credentials（凭证，通常是口令）来确定用户的身份。Acegi 提供了一个默认的 AuthenticationManager 的实现类 ProviderManager，可以直接使用。ProviderManager 并不直接去验证用户提供的 Principal 和 Credentials，而是将它们委托给一个或多个 AuthenticationProvider 来验证，例如：

```
<bean id="authenticationManager"
    class="org.acegisecurity.providers.ProviderManager">
    <property name="providers">
        <list>
            <ref bean="daoAuthenticationProvider" />
        </list>
    </property>
```

```
</bean>
```

ProviderManager 将逐一遍历每个 AuthenticationProvider，只要有一个 Authentication Provider 成功地认证了用户，该认证过程就结束。Acegi 已经提供了多个 Authentication Provider 可供选择。

（1）AuthByAdapterProvider：通过 Web 容器来验证用户身份。

（2）CasAuthenticationProvider：通过 CAS 服务器来验证用户身份，稍后我们还将详细讨论这种基于单点登录的认证方式。

（3）DaoAuthenticationProvider：通过数据库存储的用户名和口令信息来验证用户身份，这是 Web 应用程序最常见的认证方式。

（4）JaasAuthenticationProvider：通过 JAAS 服务来验证用户身份。

（5）PasswordDaoAuthenticationProvider：通过数据库认证，但是具体过程由底层数据源完成，例如，LDAP（Lightweight Directory Access Protocol，轻量级目录访问协议）。

（6）RememberMeAuthenticationProvider：通过浏览器提供的 Cookie 来验证用户上次是否已成功登录并在有效期内，若 Cookie 被接受，则通过认证。

（7）RemoteAuthenticationProvider：通过远程服务验证用户身份。

我们首先从最常用的 DaoAuthenticationProvider 开始，配置一个基于数据库的用户名和口令认证的 AuthenticationProvider。

```
<bean id="daoAuthenticationProvider"
   class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
   <property name="userDetailsService" ref="userDetailsService" />
</bean>
```

DaoAuthenticationProvider 还需要一个 UserDetailsService 对象，UserDatailsService 负责从数据库中读取用户名、口令、用户是否启用和角色信息，然后 DaoAuthentication Provider 匹配用户名和口令，然后构造出 UserDetails 对象，作为 Authentication 的 Principal。若设置 DaoAuthenticationProvider 的 forcePrincipalAsString 的属性为 true，则 Authentication 的 Principal 将被强制设置为用户的登录名，而不是 UserDetails 对象。

在开发阶段，我们也许更希望能直接在配置文件中配置一些固定的用户以方便测试，Acegi 提供了 InMemoryDaoImpl，允许通过配置直接定义一些用户。

```
<bean id="userDetailsService"
   class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
   <property name="userMap">
      <value>
         admin=password,enabled,ROLE_ADMIN,ROLE_USER
         test=test,enabled,ROLE_USER
```

```
                guest=guest,disabled,ROLE_USER
            </value>
        </property>
    </bean>
```

InMemoryDaoImpl 的 userMap 属性的每一行代表一个用户，其格式为“用户名=口令,<帐户是否有效>,<角色 1>,<角色 2>...”，在实际应用时，替换 InMemoryDaoImpl 是相当容易的，只需要实现 UserDetailsService 接口。下面的代码演示了从 Map 中查找用户名的一个实现。

```
    public class MyUserDetailsService implements UserDetailsService {
        private final GrantedAuthority ROLE_ADMIN = new GrantedAuthorityImpl
("ROLE_ADMIN");
        private final GrantedAuthority ROLE_USER = new
GrantedAuthorityImpl("ROLE_USER");

        private Map<String, String> users = new HashMap<String, String>();

        public MyUserDetailsService() {
            users.put("admin", "password");
            users.put("test", "test");
            users.put("guest", "locked");
        }

        public UserDetails loadUserByUsername(String username)
                throws UsernameNotFoundException, DataAccessException {
            String password = users.get(username);
            if(password==null)
                throw new UsernameNotFoundException("No such user");
            GrantedAuthority[] authorities = username.equals("admin") ?
                    new GrantedAuthority[] { ROLE_ADMIN, ROLE_USER } :
                    new GrantedAuthority[] { ROLE_USER };
            boolean enabled = !username.equals("guest");
            return new User(username, password, enabled, true, true, true,
authorities);
        }
    }
```

读者可以很容易地在 MyUserDetailsService 中注入 DataSource 等资源，从而轻易地使用 JDBC 从数据库中获得用户信息，最后将其包装为 UserDetails 对象返回即可。如果数据库的表结构符合 Acegi 的默认配置，还可以直接配置一个 JdbcDaoImpl 来实现 UserDetailsService。

我们的 AuthenticationManager 采用一个 DaoAuthenticationProvider 和一个 Remember

MeAuthenticationProvider，整个配置如下。

```
        <bean id="authenticationManager" class="org.acegisecurity.providers.
ProviderManager">
            <property name="providers">
               <list>
                  <ref bean="daoAuthenticationProvider" />
                  <ref bean="rememberMeAuthenticationProvider" />
               </list>
            </property>
        </bean>

        <!-- 基于 DAO 验证的 AuthenticationProvider -->
        <bean id="daoAuthenticationProvider"
            class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
            <property name="userDetailsService" ref="userDetailsService" />
        </bean>

        <!-- 使用内存 DAO,实际应用时可用 JdbcDao 代替 -->
        <bean id="userDetailsService"
            class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
            <property name="userMap">
               <value>
                  admin=password,enabled,ROLE_ADMIN,ROLE_USER
                  test=test,enabled,ROLE_USER
                  guest=guest,disabled,ROLE_USER
               </value>
            </property>
        </bean>

        <!-- Remember Me 的 AuthenticationProvider -->
        <bean id="rememberMeAuthenticationProvider"
            class="org.acegisecurity.providers.rememberme.RememberMeAuthentication
Provider">
            <property name="key" value="remember_Me" />
        </bean>
```

2．配置 AccessDecisionManager

身份认证是 Acegi 安全保护机制的第一步。在用户通过了身份认证后，Acegi 必须决定是否允许用户访问某一受保护的资源，这一步是由决策管理器（即 AccessDecision Manager）完成的。该接口的核心方法如下。

```
        public void decide(Authentication authentication, Object object, Config
AttributeDefinition config)
                throws AccessDeniedException, InsufficientAuthenticationException;
```

如果没有 AccessDeniedException 或 InsufficientAuthenticationException 抛出，即认为允许访问。

AccessDecisionManager 的决策过程相当有趣，它不是靠自己完成决策，而是基于投票的方式，让一个或多个具有投票权的 AccessDecisionVoter 对象进行投票，然后根据投票结果判断是否允许访问该资源。

Acegi 提供了 3 个基本的 AccessDecisionManager 实现。

（1）AffirmativeBased：至少有一个投票者允许访问时，就允许访问该资源，这是最常见的方式。

（2）ConsensusBased：按照“多数人同意”原则授权，即赞成票多于反对票时授权。如果赞成票与反对票相等，则根据 isAllowIfEqualGrantedDeniedDecisions 属性决定是否授权，默认该属性为 true，即相等情况下也授权。

（3）UnanimouseBased：如果没有投票者拒绝访问，就允许访问该资源。

这 3 个 AccessDecisionManager 还有一个共同的 allowIfAllAbstainDecisions 属性，表示“如果所有投票者都弃权，是否允许访问”。如果设置为 true，就建立了一个“沉默即同意”的策略。

每个投票者对于是否允许访问该资源并没有最终发言权，但是他们的投票将直接影响 AccessDecisionManager 做出最终授权。基于这一点，每个投票者可以以 3 种方式进行投票。

（1）ACCESS_GRANTED：赞成票，投票者希望允许访问受保护的资源。

（2）ACCESS_DENIED：拒绝票，投票者希望拒绝访问受保护的资源。

（3）ACCESS_ABSTAIN：弃权票，投票者不关心该资源，决定放弃投票。

要实现一个投票者就必须实现 AccessDecisionVoter 接口，虽然可以自己编写实现类，不过，Acegi 已经提供了一个非常有用的投票者实现：RoleVoter，它根据资源关联的角色进行投票。

例如，某个 JSP 页面被设置为仅允许角色为 ROLE_ADMIN 和 ROLE_USER 的用户访问，则 RoleVoter 的投票将根据当前用户的授权决定，若当前用户的角色为 ROLE_ADMIN 或 ROLE_USER，RoleVoter 将投赞成票；若当前用户的角色既非 ROLE_ADMIN，也非 ROLE_USER，RoleVoter 将投拒绝票；若某一资源关联的授权不与角色相关，则 RoleVoter 将投弃权票。

RoleVoter 默认只对以“ROLE_”开头的授权进行判断，也可以通过 rolePrefix 属性将其改为其他前缀，不过，不推荐这么做。

```
<bean id="roleVoter" class="org.acegisecurity.vote.RoleVoter"
   <property name="rolePrefix" value="GROUP_" />
</bean>
```

再回到 Spring_Acegi 工程，整个 AccessDecisionManager 采用 AffirmativeBased 和一个 RoleVoter，配置如下。

```
<bean id="accessDecisionManager"
   class="org.acegisecurity.vote.AffirmativeBased">
   <property name="decisionVoters">
      <list>
         <bean class="org.acegisecurity.vote.RoleVoter" />
      </list>
   </property>
   <property name="allowIfAllAbstainDecisions" value="false" />
</bean>
```

3．配置 FilterChain

现在，我们有了 AuthenticationManager 和 AccessDecisionManager，最后一步就是配置一个过滤器链，让每个过滤器各司其职，完成认证和授权，这也是最复杂的一步。

Acegi 已经提供了一系列非常有用的 Filter 供我们使用，常用的 Filter 如下。

（1）ChannelProcessingFilter：确保当前 URL 以指定的协议访问，例如，必须对以/secure/开头的 URL 使用 HTTPS 访问。

（2）ConcurrentSessionFilter：阻止同一用户在某一段时间内多次登录。

（3）HttpSessionContextIntegrationFilter：由于用户的认证信息存放在 Http Session 中，这个过滤器的作用就是从 Session 中获得用户的认证信息，然后将其关联到当前请求中，如果没有这个过滤器，后续的请求处理就无法获得已认证的用户身份信息。

（4）LogoutFilter：通过过滤特定的 URL（例如，/j_logout），LogoutFilter 可以实现用户注销的功能。

（5）AuthenticationProcessingFilter：通过过滤特定的 URL（例如，/j_security_check），AuthenticationProcessingFilter 可以验证用户名和口令，实现用户登录的功能。Acegi 提供了多种登录方式，除了由应用程序自身通过 JDBC 验证外，还有 CasProcessingFilter、JbossIntegrationFilter 等。

（6）SecurityContextHolderAwareRequestFilter：如果应用程序需要调用 HttpServlet Request 的 getRemoteUser()获得用户身份，就可以使用 SecurityContextHolder Aware RequestFilter 来包装原始的 HttpServletRequest，它使用一个代理模式返回 Acegi Authentication 对象的 Principal。

（7）RememberMeProcessingFilter：实现记住用户登录信息的功能，使用户在一段时间内都不必输入用户名和口令，Acegi 默认采用 Cookie 记住用户登录信息。

（8）AnonymouseProcessingFilter：如果当前用户没有登录，就将一个匿名用户身份放入 SecurityContext 中。

（9）ExceptionTranslationFilter：捕获任何与 Acegi 安全相关的异常，然后根据需要将用户导向到登录页面，或者直接发送一个 403 禁止访问的错误代码。

（10）FilterSecurityInterceptor：最终真正保护 Web 资源的拦截器。

Filter 的顺序非常重要，有的 Filter 对其他 Filter 有依赖关系，如果顺序不当，某些 Filter 将不会起作用，或者无法正常工作。总的来说，Filter 的顺序应该严格按照上述介绍的顺序排列，但可以有选择地使用。

ChannelProcessingFilter 和 ConcurrentSessionFilter 不访问 SecurityContextHolder，因此，应当放在 FilterChain 的最前面，这两个 Filter 都是可选的。在我们这个例子中，没有使用这两个 Filter。如果希望用户必须以 HTTPS 的方式访问/secure/目录下的所有资源，就可以配置一个 ChannelProcessingFilter，即使用户手动输入 http://访问/secure/目录，也会被强制重定向为 https://。

```
        <bean id="channelProcessingFilter"
           class="org.acegisecurity.securechannel.ChannelProcessingFilter">
           <property name="channelDecisionManager">
              <bean class="org.acegisecurity.securechannel.ChannelDecisionManager
Impl">
                 <property name="channelProcessors">
                    <list>
                       <bean class="org.acegisecurity.securechannel.SecureChannel
Processor" />
                       <bean class="org.acegisecurity.securechannel.Insecure
ChannelProcessor" />
                    </list>
                 </property>
              </bean>
           </property>
           <property name="filterInvocationDefinitionSource">
              <value>
                 CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                 PATTERN_TYPE_APACHE_ANT
                 /secure/**=REQUIRES_SECURE_CHANNEL
              </value>
           </property>
        </bean>
```

下一个必须配置的是 HttpSessionContextIntegrationFilter，它试图从 HttpSession 中将已经过认证的用户身份（即 Authentication 对象）取出，放入 SecurityContextHolder 中，以便后续的处理能通过 SecurityContextHolder 中随时取出 Authentication 对象。任何需要访问 SecurityContextHolder 的 Filter 都必须放在 HttpSessionContextIntegrationFilter 之后，否则获取的 Authentication 对象永远是 null。HttpSessionContextIntegrationFilter 的配置非

常简单。

```
<bean id="httpSessionContextIntegrationFilter"
   class="org.acegisecurity.context.HttpSessionContextIntegrationFilter" />
```

LogoutFilter 和 AuthenticationProcessingFilter 用于实现用户的注销和登录功能，它们仅过滤特定的 URL，我们在这里设置的注销和登录的 URL 分别为/j_logout.do 和/j_login.do。

```
<bean id="logoutFilter" class="org.acegisecurity.ui.logout.LogoutFilter">
   <!-- 注销后默认的跳转页面 -->
   <constructor-arg value="/helloWorld.jsp" />
   <constructor-arg>
      <list>
         <ref bean="rememberMeServices" />
         <bean class="org.acegisecurity.ui.logout.SecurityContextLogout
Handler" />
      </list>
   </constructor-arg>
   <!-- 用户注销的 URL -->
   <property name="filterProcessesUrl" value="/j_logout.do" />
   </bean>
<bean id="authenticationProcessingFilter"
   class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
   <property name="authenticationManager" ref="authenticationManager" />
   <!-- 登录失败的页面，通常是包含出错信息的登录页面 -->
   <property name="authenticationFailureUrl" value="/login.jsp?login_error=
Login%20failed." />
   <!-- 登录后默认的跳转页面 -->
   <property name="defaultTargetUrl" value="/helloWorld.jsp" />
   <!-- 用户登录的 URL -->
   <property name="filterProcessesUrl" value="/j_login.do" />
   <property name="rememberMeServices" ref="rememberMeServices" />
</bean>
```

接下来如果要使用“记住用户”的功能，就应该配置一个可选的 RememberMeProcessingFilter。Acegi 提供了一个以 Cookie 方式实现的 TokenBasedRememberMeServices 和一个不实现任何功能的 NullRememberMeServices。

当用户输入了正确的用户名和口令登录成功后，AuthenticationProcessingFilter 就调用 RememberMeServices 的 loginSuccess()方法，指示 RememberMeServices 记录用户的登录信息。TokenBasedRememberMeServices 按照如下方式构造一个字符串来记录用户的登录信息，经过 BASE64 编码后存放到 Cookie 中。

```
用户名:过期时间:MD5(用户名:过期时间:口令:Key)
```

请注意，Cookie 中并不存储用户口令，而是一个经过组合的字符串的 MD5 码。当 Session 失效后，用户再次访问受保护的资源时，autoLogin()方法就会被调用，并从 Cookie 中获得上次存储的字符串。通过拆分字符串，TokenBasedRememberMeServices 可以获得用户名、过期时间和一个 MD5 码，此时，再请求 UserDetailsService 就可以获得用户口令，然后计算字符串“用户名:过期时间:口令:key”的 MD5 码，若与 Cookie 中的一致，即认为登录成功。

key 的作用是让该字符串更加随机，以免根据 MD5 码很容易地获得原始字符串的值，因此可以设置一个复杂的字符串。不过，key 一经设置，在应用程序运行期就不要更改，否则，所有用户的 Cookie 都会失效。

rememberMeFilter 和 rememberMeServices 配合就实现了自动登录功能，这两个组件的配置如下。

```
<bean id="rememberMeFilter"
    class="org.acegisecurity.ui.rememberme.RememberMeProcessingFilter">
    <property name="authenticationManager" ref="authenticationManager" />
    <property name="rememberMeServices" ref="rememberMeServices" />
</bean>
<bean id="rememberMeServices"
    class="org.acegisecurity.ui.rememberme.TokenBasedRememberMeServices">
<property name="userDetailsService" ref="userDetailsService" />
    <property name="parameter" value="j_remember_me" />
    <property name="key" value="remember_Me" />
    <property name="tokenValiditySeconds" value="31536000" />
</bean>
```

不过，要让 RememberMeFilter 起作用，HTML 登录页表单中“Remember Me”字段默认的 name 是“_acegi_security_remember_me”，否则，必须设置为与 TokenBased RememberMeServices 对象的 parameter 属性一致。这个例子中我们设置的 parameter 属性为“j_remember_me”。RememberMeFilter 的 key 属性必须和前面配置的 remember MeAuthenticationProvider 的 key 属性一致，才能够成功地验证有效的 Cookie。默认的 Cookie 有效期是 14 天，可以通过 tokenValiditySeconds 设置 Cookie 有效期，单位为秒，例如，1 年：31536000。

AnonymouseProcessingFilter 也是可选的，如果请求处理到此，用户仍然没有登录，就可以使用 AnonymouseProcessingFilter 向 SecurityContextHolder 中放入一个匿名用户对象，否则，在后续调用 SecurityContextHolder.getContext().getAuthentication()时将返回 null。

ExceptionTranslationFilter 虽然是可选的，但是通常应该配置，它的作用是捕获

AuthenticationException 和 AccessDeniedException 这两个异常，若捕获到 AuthenticationException，默认将用户导向到登录页面；若捕获到 AccessDeniedException，默认向用户返回一个 403 Forbidden 的错误，也可以将用户导向到一个自定义的“您没有权限访问该资源”的提示页面。如果没有 ExceptionTranslationFilter，一旦发生安全相关的异常，用户就只能得到一个 500 ServletException。在这里，我们配置如下。

```
<bean id="exceptionFilter"
   class="org.acegisecurity.ui.ExceptionTranslationFilter">
   <!-- 出现 AuthenticationException 时的登录入口 -->
   <property name="authenticationEntryPoint">
      <bean class="org.acegisecurity.ui.webapp.AuthenticationProcessing
FilterEntryPoint">
         <!-- 将用户导向到登录页面 -->
         <property name="loginFormUrl" value="/login.jsp" />
         <!-- 是否强制使用 HTTPS -->
         <property name="forceHttps" value="false" />
      </bean>
   </property>
   <!-- 出现 AccessDeniedException 时的 Handler -->
   <property name="accessDeniedHandler">
      <!-- 还可以设置"errorPage"属性,例如"/denied.html" -->
      <bean class="org.acegisecurity.ui.AccessDeniedHandlerImpl" />
   </property>
</bean>
```

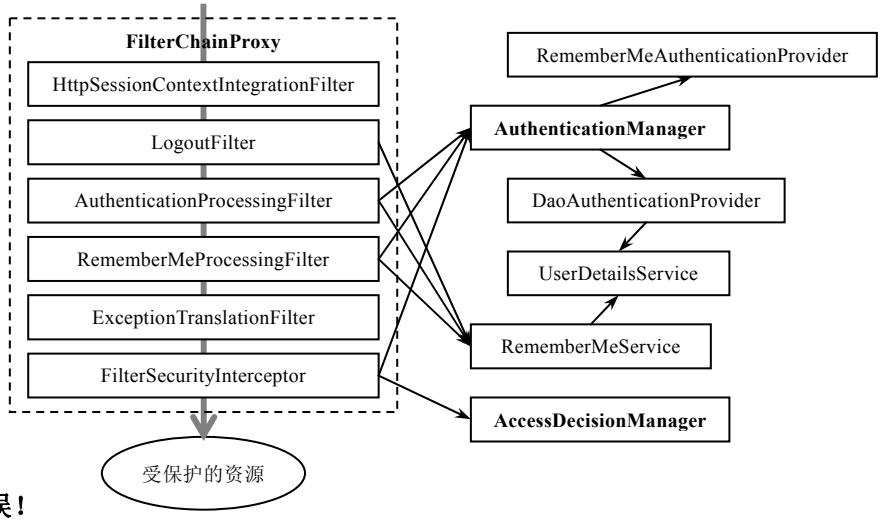
最后一个 FilterSecurityInterceptor 是必须的，因为真正保护 Web 资源的 Filter 正是这个 FilterSecurityInterceptor。FilterSecurityInterceptor 通过将 URL 和相应的角色关联起来，再使用 AccessDecisionManager 就实现了资源的授权访问。在我们这个例子中，为了保护指定的资源，定义如下。

```
<bean id="securityInterceptor"
      class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
   <property name="authenticationManager" ref="authenticationManager" />
   <property name="accessDecisionManager" ref="accessDecisionManager" />
   <property name="objectDefinitionSource">
      <value>
         CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
         PATTERN_TYPE_APACHE_ANT
         /admin/**=ROLE_ADMIN
         /user/**=ROLE_USER
      </value>
   </property>
</bean>
```

CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON 指示将 URL 转换为小写后再匹配，PATTERN_TYPE_APACHE_ANT 指示使用 ANT 风格的 URL，而不是默认的正则表达式，然后我们定义/admin/和/user/两个目录的权限分别为 ROLE_ADMIN 和 ROLE_USER。

不管使用哪种表达式语法，表达式都总是按照其定义的顺序进行处理，所以，越详细的定义要放在前面，这一点非常重要。例如，如果/secure/**出现在/secure/super/**模式之前，那么，/secure/super/目录下的任何资源总是首先和/secure/**匹配，因此，/secure/super/**将永远不会起作用。

至此，我们已经完成了整个 Acegi 安全框架的配置，让我们再回顾一下我们配置了哪些组件，以及它们之间的依赖关系，如图 10-6 所示。



错误！

图 10-6

图 10-6 看上去有点复杂，实际上，整个安全框架主要由 FilterChainProxy、AuthenticationManager 和 AccessDecisionManager 三大组件构成，它们各司其职，配合完成整个安全模块。

至此，我们还没有编写一行代码。下一步，我们创建两个测试用的 helloUser.jsp 和 helloAdmin.jsp，分别放在 user 目录和 admin 目录，再编写一个登录页面。

```
<html>
  <body>
    <h3>Please Login</h3>
    <%
      String error = request.getParameter("login_error");
      if(error!=null)
```

```
            out.println("<p><font color=\"red\">" + error + "</font></p>");
        %>
        <form action="j_login.do" method="POST">
          <p>Username: <input type="text" name="j_username" /></p>
          <p>Password: <input type="password" name="j_password"></p>
          <p><input type="checkbox" name="j_remember_me">Remember me</p>
          <p><input name="submit" type="submit" value="Login"></p>
        </form>
      </body>
    </html>
```

登录表单的 URL 必须和 AuthenticationProcessingFilter 的 filterProcessesUrl 一致，用户名和口令必须为“j_username”和“j_password”，这两个常量定义在 AuthenticationProcessingFilter 中，无法更改。“记住用户”的字段名称必须和 RememberMeServices 对象的 parameter 属性一致。

直接启动 Resin，然后打开浏览器，输入 http://localhost:8080/user/helloUser.jsp，此时，Acegi 自动拦截受保护的资源，并将用户导向到登录页面，如图 10-7 所示。

输入用户名“test”和口令“test”，登录成功后，就可以看到/user/helloUser.jsp 页面，如图 10-8 所示。

图 10-7

图 10-8

由于此时用户身份为 test，角色为 ROLE_USER，如果直接访问/admin/helloAdmin.jsp，则由于权限不够，导致 403 错误，如图 10-9 所示。

此时，必须注销登录，然后以用户“admin”身份登录，才能获得 ROLE_ADMIN 角色，此时，再访问/admin/helloAdmin.jsp 页面，就可以正常显示，如图 10-10 所示。

图 10-9

图 10-10

在测试中，我们发现，使用 Acegi 1.0.3 版本时，当用户注销后再次单击“logout”，将产生一个 NullPointerException。

```
        java.lang.NullPointerException
            at org.acegisecurity.ui.rememberme.TokenBasedRememberMeServices.logout
(TokenBasedRememberMeServices.java:295)
            at org.acegisecurity.ui.logout.LogoutFilter.doFilter(LogoutFilter.java:
98)
            at org.acegisecurity.util.FilterChainProxy$VirtualFilterChain.doFilter
(FilterChainProxy.java:274)
            at org.acegisecurity.context.HttpSessionContextIntegrationFilter.
doFilter(HttpSessionContextIntegrationFilter.java:229)
            at org.acegisecurity.util.FilterChainProxy$VirtualFilterChain.doFilter
(FilterChainProxy.java:274)
            ...
```

仔细查看 Acegi 的源代码，可以在 TokenBasedRememberMeServices.java 中看到如下代码。

```
        public void logout(HttpServletRequest request, HttpServletResponse response,
Authentication authentication) {
            // 295 行
            cancelCookie(request, response, "Logout of user " + authentication.
getName());
        }
```

当用户处于未登录状态时，如果当前过滤器链上也没有 AnonymouseProcessingFilter，则获得的 Authentication 对象为 null，在 TokenBasedRememberMeServices 第 295 行调用 authentication.getName()会抛出 NullPointerException。为了修复 Acegi 的这个 Bug，我们将 TokenBasedRememberMeServices.java 复制一份到 src 目录，放在 org.acegisecurity.ui.rememberme 包中，然后修改 logout()方法。

```
        public void logout(HttpServletRequest request, HttpServletResponse response,
Authentication authentication) {
            cancelCookie(request, response, "Logout of user "
                + (authentication==null ? "Anonymouse" : authentication.getName()));
        }
```

编译后，TokenBasedRememberMeServices 将被放到/WEB-INF/classes 目录下。由于 classes 目录的优先级比 lib 目录高，因此，ClassLoader 将会首先加载我们修改后的 TokenBasedRememberMeServices，避免了 NullPointerException。

如果无法获得源代码，或者该软件的许可证不允许更改源代码，就只能通过派生一

个自定义的子类，然后覆写超类的方法。

```
public class MyTokenBasedRememberMeServices
       extends TokenBasedRememberMeServices
{
    @Override
    public void logout(HttpServletRequest request, HttpServletResponse
response, Authentication authentication) {
        response.addCookie(makeCancelCookie(request));
    }
}
```

后续的 Acegi 版本可能会修复这个 Bug。读者可以根据当前使用的 Acegi 版本查看是否需要修复该 Bug。

10.2.2 保护Bean组件

除了对 Web 资源（如 JSP 页面等）进行 URL 级别的安全保护外，通常，业务逻辑组件也需要方法级别的安全保护。例如，我们设计了一个 Service 接口，任何登录后的用户都可以创建订单。

```
public interface Service {
    void createOrder(String orderId);
}
```

我们设计一个简单的实现类，仅仅将方法调用的信息打印出来。

```
public class ServiceImpl implements Service {
    private Log log = LogFactory.getLog(getClass());

    public void createOrder(String orderId) {
        log.info("createOrder");
    }
}
```

假设我们设计了一个 OrderController 来调用该业务组件。

```
public class OrderController implements Controller {
    private Service service;
    public void setService(Service service) {
        this.service = service;
    }

    public ModelAndView handleRequest(HttpServletRequest request, HttpServlet
```

```
Response response) throws Exception {
                service.createOrder("order_id");
                PrintWriter writer = response.getWriter();
                writer.write("<h3>createOrder() called.</h3>");
                return null;
            }
        }
```

然后，在 web.xml 中声明 Spring 的 DispatcherServlet。

```
        <servlet>
            <servlet-name>dispatcher</servlet-name>
            <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
            <load-on-startup>0</load-on-startup>
        </servlet>
        <servlet-mapping>
            <servlet-name>dispatcher</servlet-name>
            <url-pattern>*.do</url-pattern>
        </servlet-mapping>
```

并创建一个 dispatcher-servlet.xml 配置文件，将 OrderController 配置如下。

```
        <?xml version="1.0" encoding="UTF-8"?>
        <beans xmlns="http://www.springframework.org/schema/beans"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
        >
            <bean id="service" class="example.chapter10.ServiceImpl" />

            <bean name="/order.do" class="example.chapter10.OrderController">
                <property name="service" ref="service" />
            </bean>
        </beans>
```

很不幸，由于我们的疏忽，原本应当映射到/user/order.do 路径的 OrderController 却映射到了/order.do 上，于是，我们前面设置的 Filter 就无法起到安全保护的作用，一个未登录的用户通过访问 http://localhost:8080/order.do 就可以轻易地调用 Service 组件的 createOrder()方法，在一个实际的应用中，这将对整个系统的安全构成威胁。

我们需要的是将业务逻辑组件也保护起来，针对角色实现授权。正如前面提到的，Acegi 同样提供了针对组件的方法级别调用的保护，其实现原理便是 Spring AOP 机制。

针对业务组件实现保护的关键是声明一个 MethodSecurityInterceptor，使其有能力拦

截组件的方法调用，然后根据用户角色来决定是否允许调用该方法。

```
        <bean id="serviceSecurityInterceptor"
            class="org.acegisecurity.intercept.method.aopalliance.MethodSecurity
Interceptor">
            <property name="validateConfigAttributes" value="true" />
            <property name="authenticationManager" ref="authenticationManager" />
            <property name="accessDecisionManager" ref="accessDecisionManager" />
            <property name="objectDefinitionSource">
                <value>
                    example.chapter10.Service.createOrder=ROLE_USER
                </value>
            </property>
        </bean>

        <!-- 利用 Spring 的自动代理功能实现 AOP 代理 -->
        <bean id="autoProxyCreator"
            class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxy
Creator">
            <property name="interceptorNames">
                <list>
                    <value>serviceSecurityInterceptor</value>
                </list>
            </property>
            <property name="beanNames">
                <list>
                    <value>service</value>
                </list>
            </property>
        </bean>
```

现在，通过 Acegi 的 MethodSecurityInterceptor 实现的 AOP 拦截，就保证了 Service 组件的安全。再次访问 http://localhost:8080/order.do，这一次受保护的 createOrder()方法并未被调用，而未登录的用户将被导向到登录页面。

4．使用 Java 5 注解

读者可能已经注意到了，在 MethodSecurityInterceptor 中配置安全信息是通过 objectDefinitionSource 属性实现的，在上面的配置中，我们直接填入完整的方法调用和关联的角色。如果一个业务组件的方法很多，在配置文件中指定将相当麻烦，甚至很难避免由于拼写错误造成的安全漏洞。幸运的是，Acegi 还提供了另一种基于 Java 5 注解的配置方式，这种配置方式将安全配置以 Java 5 注解的形式和源代码放在一起，极大地提高了代码的可维护性。

使用 Java 5 注解来声明组件安全性时，objectDefinitionSource 属性被简单地设置如下。

```
        <property name="objectDefinitionSource">
           <bean class="org.acegisecurity.intercept.method.MethodDefinition
Attributes">
               <property name="attributes">
                  <bean
class="org.acegisecurity.annotation.SecurityAnnotationAttributes" />
               </property>
           </bean>
        </property>
```

然后，在 Service 接口中添加必要的注解。

```
        public interface Service {
            @Secured({"ROLE_USER"})
            void createOrder(String orderId);
        }
```

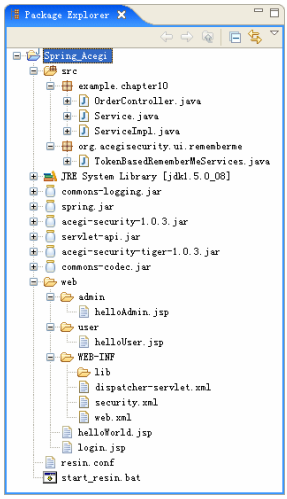


图 10-11

为了使用@Secured 注解，必须导入 Acegi 的 acegi-security-tiger-1.0.3.jar 包到工程的 Java Build Path 中。每个@Secured 注解指定一个或多个有权调用该方法的角色名称，编写该注解非常直观。

整个 Spring_Acegi 工程的最终结构如图 10-11 所示。

现在，即使我们仍没有将 OrderController 的 URL 映射改回正确的/user/order.do，由于 Acegi 在组件的方法级别的保护，仍然保证了整个系统的安全。

Acegi 还提供了基于 AspectJ 连接点的安全拦截器，不过，其配置远比使用 Java 5 注解复杂得多，并且还需要引入 AspectJ 运行期的库文件，因此不推荐使用，本书对此也不做更多介绍。

10.3 实现单点登录

单点登录即 Single Sign On，简称 SSO，是近年来兴起的一种为用户提供更好的用户体验的服务。它允许用户仅登录一次，就能使用多个 Web 应用，而不必重复登录。本节将详细介绍单点登录的概念、原理和最常用的一种开源 SSO 的实现方式。

10.3.1 SSO简介

在传统的 Web 应用中，如果要为用户提供多个服务，例如，商城、拍卖网站、Web 邮件、IM 系统等，则用户使用每个服务都需要登录到该 Web 应用上，还可能使用不同的用户名和口令，即使这些服务都是由一个公司提供的。对用户而言，这将造成极大的使用不便。因此，单点登录应运而生，它允许用户仅登录一次，就能使用多个 Web 应用，而不必重复登录。

除了登录一次，多次使用外，SSO 带来的另一个好处是每个用户只需记住一个用户名和口令，而不必针对每个 Web 应用重复申请，不仅方便了用户，还简化了整个系统的用户管理。

SSO 实现单点登录的关键在于每个 Web 应用并不直接认证用户，而是将用户认证委托给一个独立的 SSO 服务器完成。图 10-12 显示了用户访问一个 Web 应用时如何通过 SSO 服务器认证的整个流程。

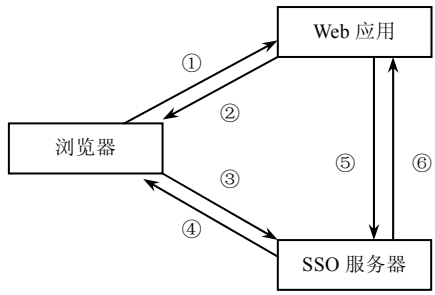


图 10-12

① 浏览器请求 Web 应用程序的某个受保护的资源。

② Web 应用如果发现用户尚未经过认证，就将用户导向到 SSO 服务器登录地址（例如，https://sso-server/login）。

③ 浏览器被重定向到 SSO 服务器的登录页面，并提示用户输入用户名和口令。

④ 如果用户名和口令被 SSO 服务器成功认证，则 SSO 服务器将浏览器重定向到原 Web 应用的服务器，并且在 URL 参数中包含一个票据。

⑤ Web 应用获得了浏览器传来的票据后，连接 SSO 服务器，检查票据是否有效。

⑥ SSO 服务器若验证票据成功，则返回一个肯定的回复给 Web 应用，整个验证过程结束。

从以上过程可以看到，用户认证并不是通过 Web 应用本身完成，而是委托给 SSO 服务器完成的，因此，有多个 Web 应用时，用户始终使用一个用户名和口令通过 SSO 服务器登录，这就是单点登录的实现。此外，用户看不到第 ⑤ 步和第 ⑥ 步，即 Web 应

用和 SSO 服务器通信的过程。由于只有 SSO 服务器才知道用户名和票据的关系，因此用户无法伪造票据。要保证用户无法猜测出 SSO 服务器生成的票据，SSO 服务器生成的票据就必须非常随机，以确保整个系统的安全。

如果有多个 Web 应用，每个 Web 应用都不必自己认证用户，而是由 SSO 服务器统一认证，用户名和口令也只需由 SSO 服务器存储并管理，这样就使用户能够实现一次登录，使用多个服务。

值得注意的是，单点登录只解决了身份认证问题，而授权仍需要由各个 Web 应用程序自行实现。换句话说，SSO 服务器只能告诉 Web 应用程序该用户通过了身份认证，但无法授予用户相应的权限，因为 SSO 服务器并不知道也无法获知用户在某个 Web 应用程序中究竟是什么角色。

目前，各大厂商都有自己的 SSO 产品，但这些产品大都价格昂贵。相比之下，开源的 SSO 实现也能满足大多数 Web 应用的需求，CAS 则是开源 SSO 解决方案中的佼佼者。

CAS 即 Central Authentication Service，是耶鲁大学发起的一个开源项目，完全采用 Java 实现，由于其配置简单，安全可靠，因此它是最为流行的一个免费的 SSO 解决方案。Acegi 已经集成了 CAS，能非常容易地实现单点登录。

CAS 目前最新版本为 3.0，1.0 和 2.0 都是由 Yale 开发的，而 3.0 是由 JA-SIG 开发的，引入了 Spring 的依赖注入、AOP 等特性以便于扩展。和 12MB 的 3.0 发布版相比，2.0 版本仅 200K 左右且配置简单，已经完全符合我们的需求，因此，在本节中，我们将采用 CAS 2.0 版本作为 SSO 服务器。

在使用 CAS 服务之前，可以从 CAS 的官方站点 http://www.ja-sig.org/products/cas/ 下载 CAS 服务器端和客户端。这里我们采用的服务器端为 cas-server-2.0.12，客户端为 cas-client-java-2.1.1。读者也可以从本书的配套光盘中获得服务器端和客户端的 ZIP 文件。

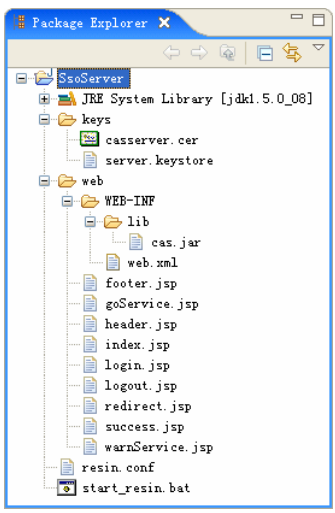


图 10-13

10.3.2 配置CAS服务器

由于我们的 CAS 服务器也是运行在本地的，因此，要搭建一个完整的测试环境，就需要启动两个 Resin 服务器，分别运行 CAS 服务和 Acegi Web 应用。我们首先来搭建 CAS 服务器。在 Eclipse 中建立如图 10-13 所示的 SsoServer 工程，然后解压 cas-server-2.0.12.zip，将 web 目录全部复制到 SsoServer 工程下。

由于 CAS 采用安全的 HTTPS 处理与浏览器及 Acegi 应用之间的通信，所以必须首先配置 SSL 使用的证书。我

们准备将证书放在 web/keys 目录下，以便 Resin 服务器能使用它。

在创建证书之前，首先必须确保环境变量 JAVA_HOME 和 PATH 的正确性。

```
D:\projects\SsoServer\keys>echo %JAVA_HOME%
C:\java\jdk1.5.0_08

D:\projects\SsoServer\keys>echo %PATH%
C:\java\jdk1.5.0_08\bin;C:\java\apache-ant-1.6.5\bin;C:\WINDOWS\system32;C
:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\java\jwsdp-2.0\jwsdp-shared\bin
```

下一步是创建 keystore，生成 CAS 服务器所需的证书。输入以下命令。

```
D:\projects\SsoServer\keys>keytool -genkey -alias casserver -keyalg RSA
-keystore server.keystore
输入 keystore 密码：  casserver
您的名字与姓氏是什么？
  [Unknown]:  localhost
您的组织单位名称是什么？
  [Unknown]:  www.javaeedev.com
您的组织名称是什么？
  [Unknown]:  javaeedev.com
您所在的城市或区域名称是什么？
  [Unknown]:  BJ
您所在的州或省份名称是什么？
  [Unknown]:  BJ
该单位的两字母国家代码是什么
  [Unknown]:  CN
CN=localhost, OU=www.javaeedev.com, O=javaeedev.com, L=BJ, ST=BJ, C=CN 正确吗？
  [否]:  y

输入<casserver>的主密码
        （如果和 keystore 密码相同，按回车）:
```

如果命令正确无误，应该在 keys 目录下找到 server.store 文件，该文件就存储了 CAS 服务器端的证书。要特别注意的是，在输入“您的名字与姓氏是什么”时，务必填入实际的 CAS 服务器域名（例如，sso.mycompany.com），这里我们要在本机上同时运行 CAS 服务器和 CAS 客户端，因此，只能填入“localhost”。CAS 客户端在验证 CAS 服务器证书时，若发现域名与证书不符，将直接抛出一个 javax.net.ssl.SSLHandshakeException，无法与 CAS 服务器建立 SSL 连接。

服务器端的证书现在已配置完毕，但是客户端还没有导入服务器端的证书，将来就无法验证服务器证书。由于我们的客户端实际上是运行在 JVM 之上的 Resin 服务器，因此，客户端证书必须导入到 JDK 的证书库中。首先，输入以下命令，将服务器端证书导

出为一个单独的证书文件。

```
D:\projects\SsoServer\keys>keytool -export -file casserver.cer -alias
casserver -keystore server.keystore
输入 keystore 密码：  casserver
保存在文件中的认证 <casserver.cer>
```

现在，在 keys 目录下应当看到一个 casserver.cer 证书文件，下一步是将其导入 JDK 的证书库中。输入以下命令。

```
D:\projects\SsoServer\keys>keytool -import -file casserver.cer -alias
casserver -keystore %JAVA_HOME%\jre\lib\security\cacerts
输入 keystore 密码：  changeit
Owner: CN=localhost, OU=www.javaeedev.com, O=javaeedev.com, L=BJ, ST=BJ, C=CN
发照者：CN=localhost, OU=www.javaeedev.com, O=javaeedev.com, L=BJ, ST=BJ, C=CN
序号：45888cae
有效期间：Wed Dec 20 09:06:54 CST 2006 至：Tue Mar 20 09:06:54 CST 2007
认证指纹：
        MD5:  F6:2E:7E:29:C1:EA:EA:CF:BF:57:64:EA:F2:E0:75:07
        SHA1:  57:92:1F:63:B5:75:08:0C:D2:E5:CC:57:7C:B8:A8:BC:E8:21:64:08
信任这个认证？[否]：  y
认证已添加至 keystore 中
```

如果没有更改过 JDK 的证书库，默认的初始密码是“changeit”。导入服务器证书后，Acegi 客户端才可以成功地和 CAS 服务器建立 SSL 安全连接，否则，将得到一个 sun.security.validator.ValidatorException：PKIX path building failed。

要检查 JDK 的证书库，输入命令 keytool -list，应当看到 casserver, 2006-12-20, trustedCertEntry。

```
C:\>keytool -list -keystore %JAVA_HOME%\jre\lib\security\cacerts
输入 keystore 密码：  changeit

Keystore 类型：jks
Keystore 提供者：SUN

您的 keystore 包含 44 输入

entrustclientca, 2003-1-9, trustedCertEntry,
认证指纹 (MD5)：0C:41:2F:13:5B:A0:54:F5:96:66:2D:7E:CD:0E:03:F4
...
casserver, 2006-12-20, trustedCertEntry,
认证指纹 (MD5)：F6:2E:7E:29:C1:EA:EA:CF:BF:57:64:EA:F2:E0:75:07
...
```

下一步是创建 Resin 的配置文件 resin.conf，告诉 Resin 证书库的位置和口令，以及

使用 HTTPS 的端口号。

```
<resin xmlns="http://caucho.com/ns/resin"
      xmlns:resin="http://caucho.com/ns/resin/core">
   <log name="" level="info" path="stdout:"/>
   <cluster id="">
      <server id="" port="6801">
         <http port="8443">
            <jsse-ssl>
               <key-store-file>../keys/server.keystore</key-store-file>
               <password>casserver</password>
            </jsse-ssl>
         </http>
      </server>
      <resin:import path="${resin.home}/conf/app-default.xml"/>
      <host id="" root-directory=".">
         <web-app id="/" root-directory="."/>
      </host>
   </cluster>
</resin>
```

测试期间我们使用 8443 端口号，在正式部署 CAS 服务器时可以使用标准的 443 端口。然后，编写 start_resin.bat 启动 Resin，以便运行 CAS 服务器。

```
cd web
%RESIN_HOME%\httpd -server-root . -conf ..\resin.conf
```

启动 Resin 服务器，然后打开浏览器，输入：https://localhost:8443/，应该首先看到浏览器弹出的安全警报提示，如图 10-14 所示。

由于我们的安全证书是自己创建的，不是通过信任机构颁发的，因此浏览器不能确定该证书是可以被信任的。在这个测试中，我们可以信任自己创建的证书，实际应用时，证书应当从第三方的信任机构（如 VerySign）获得，这样就不会有这个安全警报提示框。

单击“是”按钮，就可以看到 CAS 服务器的登录界面，如图 10-15 所示。

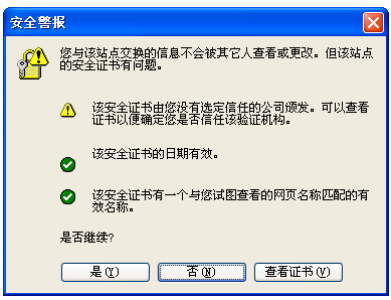


图 10-14

图 10-15

该登录页面对应的 JSP 为 login.jsp，可以很容易地将其改造为自己公司或网站相关的登录页面。

我们很容易想到的另一个问题就是 CAS 服务器在哪里验证用户的用户名和口令。查看/WEB-INF/web.xml 配置文件，可以看到如下配置信息。

```
<!-- Authentication handler -->
<context-param>
    <param-name>
        edu.yale.its.tp.cas.authHandler
    </param-name>
    <param-value>
        edu.yale.its.tp.cas.auth.provider.SampleHandler
    </param-value>
</context-param>
```

CAS 服务默认使用自带的 SampleHandler 来验证用户的用户名和口令。SampleHandler 的认证规则非常简单，如果用户名和口令相同，就通过认证。

```
public boolean authenticate(javax.servlet.ServletRequest request,
        String username, String password) {
    if(username.equals(password))
        return true;
    return false;
}
```

在实际应用中，大多数情况下我们应当从数据库中检查用户名和口令是否匹配。为此，需要自己编写验证类来替换 SampleHandler。一个自定义的验证类必须实现 PasswordHanlder 接口。

```
package edu.yale.its.tp.cas.auth;
public interface PasswordHandler extends AuthHandler {
    boolean authenticate(javax.servlet.ServletRequest request,
        String username, String password);
}
```

可以很容易地使用 JDBC 来验证用户名和口令是否匹配，然后修改 web.xml，使用自定义的这个 PasswordHandler 即可，这里不再多述。

10.3.3 集成CAS

现在，我们完成了 CAS 服务器的配置和服务器证书的导入。下一步，我们将使用 CAS 服务器实现单点登录。

我们首先将 Spring_Acegi 工程复制一份，命名为 Spring_Acegi_Cas，去掉 src 目录和 web 目录下的 login.jsp，因为我们不再需要自己的登录页面，而是依赖 CAS 服务器实现单点登录。整个工程最终的结构如图 10-16 所示。

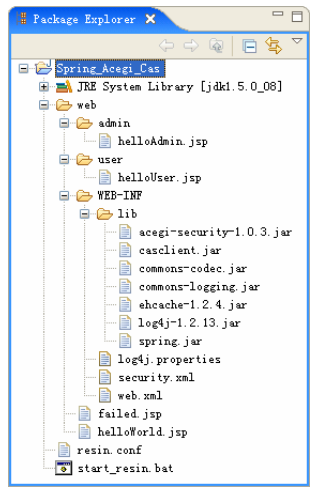


图 10-16

我们只需要配置 Acegi 就可以使用 CAS 来实现单点登录，但实际上，Acegi 与 CAS 服务器之间的交互是非常复杂的，如果读者不关心这些交互的细节，完全可以跳过下面的介绍。

Acegi 与 CAS 服务器交互的认证过程如下。

① 未登录的用户访问受保护的资源时，Acegi 的 ExceptionTranslationFilter 将捕获到 AuthenticationException，并根据 AuthenticationEntryPoint 设置导向登录入口。对于使用 CAS 认证的 Web 应用程序，其 AuthenticationEntryPoint 使用 CasProcessingFilterEntryPoint 登录入口。

② CasProcessingFilterEntry 将用户重定向到 CAS 服务器的登录 URL，并包含 Acegi 的回调 URL 地址，例如，https://cas-server/login?service=http%3A%2F%2Fweb-server%2Fj_acegi_cas _security_check。

③ 当用户被重定向到 CAS 服务器后，会被提示输入用户名和口令。如果用户之前已经登录过，并且持有一个有效的 Session Cookie，就不必再次输入用户名和口令。然后，CAS 服务器使用 PasswordHandler（如果使用 CAS 3.0 版本，则使用 AuthenticationHandler）来验证用户名和口令是否有效。

④ 如果登录成功，CAS 服务器将用户重定向到原始服务的地址，并包含一个票据（Ticket）参数，例如，http://web-server/j_acegi_cas_security_check?ticket=ABCDEFG。

⑤ Web 应用程序通过 CasProcessingFilter 截获了/j_acegi_cas_security_check 这个 URL，然后 CasProcessingFilter 根据票据参数创建一个 UsernamePassword Authentication Token 对象来表示 CAS 服务器的票据，由 AuthenticationManager 验证。

⑥ AuthenticationManager 必须使用 CasAuthenticationProvider 来验证 Username PasswordAuthenticationToken 对象，CasAuthenticationProvider 再使用 TicketValidator 对象来验证从 UsernamePasswordAuthenticationToken 对象中获得的 CAS 服务器的票据。

⑦ TicketValidator 使用 CAS 客户端的库文件通过 HTTPS 连接 CAS 服务器，请求 CAS 服务器验证该票据是否有效，其验证地址类似 https://cas-server/proxyValidate?service=http%3A%2F%2Fweb-server%2Fj_acegi_cas_security_check&ticket=ABCDEFG。

⑧ CAS 服务器根据票据是否有效向回调地址（即上一步 URL 中的 service 参数）

发送一个表示认证成功或失败的 XML。

⑨ Acegi 的 CasProxyTicketValidator 将解析 CAS 服务器返回的 XML，然后向 CasAuthenticationProvider 返回一个 TicketResponse 对象，该对象包含了用户的登录名和可能的代理服务器列表。

⑩ CasAuthenticationProvider 通过配置的 CasProxyDecider 决定是否接受代理服务器，对于简单的应用，无需使用代理服务器，直接配置一个 RejectProxyTickets 即可。

⑪ 至此，CasAuthenticationProvider 已经通过 CAS 服务器成功地认证了用户，但是，该用户的授权还必须由 Web 应用程序本身来完成，因此，CasAuthenticationProvider 再调用 CasAuthoritiesPopulator 对象来决定用户的授权。

⑫ 获得认证用户的授权后，CasAuthenticationProvider 将创建一个 CasAuthenticationToken 对象，它包含了用户的认证信息（用户名）和授权信息（角色），然后由 CasProcessingFilter 将其放入 HTTP Session 中，请注意，CasAuthenticationToken 对象实现了 Authentication 接口。

⑬ 现在，对于后续的每个用户请求，由 HttpSessionIntegrationFilter 首先将 Authentication 对象（实际上就是 CasAuthenticationToken 对象）从 HTTP Session 中取出并关联到 SecurityContextHolder，后续的 Filter 即可成功地获取 Authentication 对象。

和 Spring_Acegi 工程一样，所有的 Acegi 相关配置均放在/WEB-INF/security/目录下。首先我们需要配置的是 AuthenticationManager，不过，这里用 CasAuthenticationProvider 取代了 DaoAuthenticationProvider，并且为了简化 CAS 的配置，我们取消了 RememberMe 的功能。

```
<bean id="authenticationManager"
   class="org.acegisecurity.providers.ProviderManager">
   <property name="providers">
      <list>
         <ref bean="casAuthenticationProvider" />
      </list>
   </property>
</bean>
```

CasAuthenticationProvider 的配置稍微复杂一点，它一共需要注入以下几个对象。

（1）casAuthoritiesPopulator：当用户认证完毕后，由 casAuthoritiesPopulator 确定用户的权限。正如前面介绍的，CAS 服务器仅仅解决用户的身份认证问题，用户的授权仍然需要应用程序自行实现。

（2）ticketValidator：用于验证票据，Acegi 已经提供了一个 CasProxyTicketValidator 可供直接使用。

（3）casProxyDecider：用于决定是否使用代理，Acegi 提供了好几种实现，对于无需使用代理的简单应用，注入一个 RejectProxyTickets 即可。

（4）statelessTicketCache：用于缓存无状态的票据，Acegi 仅提供了一个基于 EhCache 的 EhCacheBasedTicketCache，这意味着我们不得不使用 EhCache 来构造该对象。

整个 casAuthenticationProvider 配置如下。

```
        <bean id="casAuthenticationProvider"
            class="org.acegisecurity.providers.cas.CasAuthenticationProvider">
            <property name="casAuthoritiesPopulator">
                <!-- 确定认证后用户的角色 -->
                <bean class="org.acegisecurity.providers.cas.populator.DaoCas
AuthoritiesPopulator">
                    <property name="userDetailsService" ref="userDetailsService" />
                </bean>
            </property>
            <property name="ticketValidator">
                <!-- 验证票据 -->
                <bean class="org.acegisecurity.providers.cas.ticketvalidator.Cas
ProxyTicketValidator">
                    <!-- 验证票据的 URL -->
                    <property name="casValidate" value="https://localhost:8443/proxy
Validate" />
                    <property name="serviceProperties" ref="serviceProperties" />
                </bean>
            </property>
            <property name="casProxyDecider">
                <!-- 是否使用代理 -->
                <bean class="org.acegisecurity.providers.cas.proxy.RejectProxy
Tickets" />
            </property>
            <property name="statelessTicketCache">
                <!-- 票据缓存 -->
                <bean class="org.acegisecurity.providers.cas.cache.EhCacheBased
TicketCache">
                    <property name="cache">
                        <!-- 构造一个 EhCache 的实例 -->
                        <bean class="net.sf.ehcache.Cache">
                            <constructor-arg index="0" value="CasTicketCache" />
                            <constructor-arg index="1" value="256" />
                            <constructor-arg index="2" value="false" />
                            <constructor-arg index="3" value="false" />
                            <constructor-arg index="4" value="600" />
                            <constructor-arg index="5" value="600" />
                        </bean>
```

```
            </property>
        </bean>
    </property>
    <property name="key" value="my_secret_cas_key" />
</bean>
```

注意到决定认证用户授权的 casAuthoritiesPopulator 还引用了一个 userDetailsService 对象，以便根据 userDetailsService 对象来确定用户的角色。我们仍采用 InMemoryDaoImpl 实现 UserDetailsService。

```
<bean id="userDetailsService"
    class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
    <property name="userMap">
        <value>
            admin=p,ROLE_ADMIN,ROLE_USER
            test=p,ROLE_USER
            guest=p,ROLE_USER
        </value>
    </property>
</bean>
```

请读者注意，当 CasAuthenticationProvider 委托 CasAuthoritiesPopulator 确定用户的授权时，用户已经通过了 CAS 服务器的认证，因此，CasAuthoritiesPopulator 仅向 userDetailsService 对象查询用户的角色，至于口令和账户是否有效等信息则一律忽略。

在实际的应用中，CAS 服务器存储了所有用户的登录用户名、口令等登录信息，而各 Web 应用程序通常也必须有相应的用户表才能确保数据库的完整性。通常，用户角色应该记录在 Web 应用程序自身的用户表中。当 CasAuthoritiesPopulator 向 userDetails Service 查询用户角色时，应当查找用户表，返回相应的角色。如果没有在用户表中查找到该用户，说明该用户虽然已经在 CAS 服务器上注册成功，但却是首次访问这个 Web 应用程序，此时，可以在 Web 应用的用户表中为该用户创建相应的记录，然后返回默认的角色，这样就实现了“首次登录并激活服务”的功能。

CasProxyTicketValidator 还引用了一个 ServiceProperties 对象，该对象包含一个 Web 应用程序自身的回调 URL 和一个 sendRenew 参数。

```
<bean id="serviceProperties"
    class="org.acegisecurity.ui.cas.ServiceProperties">
    <property name="service" value="http://localhost:8080/login.do" />
    <property name="sendRenew" value="true" />
</bean>
```

如果设置 sendRenew 为 true，将强迫 CAS 服务器再次认证用户身份，即使用户此前

已经通过了认证并仍在有效期内。该设置适用于对安全性要求较高的 Web 应用程序。

下一步是配置 AccessDecisionManager，这和 Spring_Acegi 工程是一致的。

```
<bean id="accessDecisionManager"
   class="org.acegisecurity.vote.AffirmativeBased">
   <property name="decisionVoters">
      <list>
         <bean class="org.acegisecurity.vote.RoleVoter" />
      </list>
   </property>
   <property name="allowIfAllAbstainDecisions" value="false" />
</bean>
```

然后，我们需要配置 FilterChainProxy，在这个工程中，为了强调 CAS 的配置，我们只使用 5 个必需的 Filter，顺序如下。

httpSessionContextIntegrationFilter 和 Spring_Acegi 工程中一样，用于从 HttpSession 中取出用户认证信息并放入 SecurityContextHolder 中。

```
<bean id="httpSessionContextIntegrationFilter"
   class="org.acegisecurity.context.HttpSessionContextIntegrationFilter" />
```

logoutFilter 也和 Spring_Acegi 工程一样，用于将用户身份从当前 HttpSession 中移除。

```
<bean id="logoutFilter" class="org.acegisecurity.ui.logout.LogoutFilter">
   <!-- 注销后导向的页面 -->
   <constructor-arg value="/helloWorld.jsp" />
   <constructor-arg>
      <list>
         <bean class="org.acegisecurity.ui.logout.SecurityContextLogout
Handler" />
      </list>
   </constructor-arg>
   <property name="filterProcessesUrl" value="/j_logout.do" />
</bean>
```

casProcessingFilter 则替代了 authenticationProcessingFilter，它监听回调 URL 地址 /login.do，然后由 authenticationManager 验证票据。

```
<bean id="casProcessingFilter"
   class="org.acegisecurity.ui.cas.CasProcessingFilter">
   <property name="authenticationManager" ref="authenticationManager" />
   <property name="defaultTargetUrl" value="/helloWorld.jsp" />
   <property name="filterProcessesUrl" value="/login.do" />
   <property name="authenticationFailureUrl" value="/failed.jsp" />
</bean>
```

exceptionFilter 仍负责捕获 AuthenticationException 和 AccessDeniedException，如果捕获到 AuthenticationException，就根据注入的 CasProcessingFilterEntryPoint 将用户导向到 CAS 服务器的登录页面。

```
        <bean id="exceptionFilter" class="org.acegisecurity.ui.ExceptionTranslation
Filter">
            <!-- 出现 AuthenticationException 时的登录入口 -->
            <property name="authenticationEntryPoint">
                <bean class="org.acegisecurity.ui.cas.CasProcessingFilterEntryPoint">
                    <property name="loginUrl" value="https://localhost:8443/login" />
                    <property name="serviceProperties" ref="serviceProperties" />
                </bean>
            </property>
            <!-- 出现 AccessDeniedException 时的 Handler -->
            <property name="accessDeniedHandler">
                <bean class="org.acegisecurity.ui.AccessDeniedHandlerImpl" />
            </property>
        </bean>
```

最后一个 securityInterceptor 的目的仍是通过 URL 保护 Web 资源，其配置和 Spring_Acegi 工程完全一致。

```
        <bean id="securityInterceptor"
            class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
            <property name="authenticationManager" ref="authenticationManager" />
            <property name="accessDecisionManager" ref="accessDecisionManager" />
            <property name="objectDefinitionSource">
                <value>
                    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                    PATTERN_TYPE_APACHE_ANT
                     /user/**=ROLE_USER
                    /admin/**=ROLE_ADMIN
                </value>
            </property>
        </bean>
```

最后一步，在 FilterChainProxy 中将这些 Filter 装配起来，就完成了 security.xml 的配置。

```
        <bean id="filterChainProxy"
            class="org.acegisecurity.util.FilterChainProxy">
            <property name="filterInvocationDefinitionSource">
                <value>
                    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                    PATTERN_TYPE_APACHE_ANT
```

```
                /**=httpSessionContextIntegrationFilter,logoutFilter,
casProcessingFilter,exceptionFilter,securityInterceptor
            </value>
        </property>
    </bean>
```

在这个工程中，我们还配置了 Log4J，这不是必须的，主要是为了方便地记录 Acegi 与 CAS 服务器之间的交互信息，以方便调试。此外，还需要创建一个 failed.jsp，如果整个认证失败时，将向用户显示错误信息。

```
    <%@ page contentType="text/html; charset=UTF-8" %>
    <html>
    <head><title>Login to CAS failed</title></head>
    <body>
        <h1>Login to CAS failed</h1>
        <font color="red">
            Your CAS credentials were rejected.<BR><BR>
            Reason: <%= ((org.acegisecurity.AuthenticationException)session.
getAttribute(org.acegisecurity.ui.AbstractProcessingFilter.ACEGI_SECURITY_LAST_EXCE
PTION_KEY)).getMessage() %>
        </font>
    </body>
    </html>
```

现在，整个 Spring_Acegi_Cas 工程配置完毕，首先运行 SsoServer 工程的 start_resin.bat，启动 CAS 服务器，然后运行 Spring_Acegi_Cas 工程，打开浏览器测试，其效果和 Spring_Acegi 工程一致，但是用户认证的功能却从 Web 应用程序本身移至 CAS 服务器。

某些读者可能会发现，直接在浏览器的地址栏输入“http://localhost:8080/user/helloUser.jsp”被导向到 CAS 服务器的登录页面后，输入正确的用户名和口令却得到了以下错误信息，如图 10-17 所示。

图 10-17

这是因为 Resin 在第一次为浏览器设置会话 Cookie 时，将 jsessionid 传递给了 CAS 服务器，扰乱了 CAS 服务器对浏览器的识别。解决这一问题的方法是禁用 Resin 的 URL

重写功能，在 Spring_Acegi_Cas 项目根目录下的 resin.conf 配置文件中做如下设置。

```
<resin xmlns="http://caucho.com/ns/resin"
      xmlns:resin="http://caucho.com/ns/resin/core">
   <log name="" level="info" path="stdout:"/>
   <cluster id="">
      <server id="" port="6800">
         <http port="8080"/>
      </server>
      <resin:import path="${resin.home}/conf/app-default.xml"/>
      <web-app-default>
         <session-config>
            <enable-url-rewriting>false</enable-url-rewriting>
         </session-config>
      </web-app-default>
      <host id="" root-directory=".">
         <web-app id="/" root-directory="."/>
      </host>
   </cluster>
</resin>
```

如果读者使用其他 Web 服务器，例如，Tomcat，请参考相应的服务器配置文档，确保 URL 重写的功能已经禁用。

10.4 小结

本章我们主要详细介绍了基于 Spring 的 Acegi 安全框架，从总体来说，Acegi 实现了一个声明式的安全保护机制，尽管其配置稍嫌复杂，但是能够很好地将安全逻辑和应用程序的核心逻辑分离开来，因此，很适合对安全需求较高的项目。

Acegi 安全框架通过基于 Filter 的 URL 过滤实现了针对 Web 资源的保护，通过基于 AOP 拦截的机制实现了针对组件的方法级别的保护，这两者在实际的项目中应用都非常广泛。最后，我们还给出了一个基于开源的 CAS 服务的单点登录的完整实现，完全可以应用在多个中小型网站以及企业内部网中。

通过这几章，我们基本上已经详细介绍了 Spring 2.0 框架的各个主要模块。在第 11 章中，我们将详细介绍基于 Spring 2.0 框架开发的一个完整的 Web 应用程序——Live 在线书店。这个 Web 应用程序完全基于 Spring 2.0 框架，并且集成了多种开源解决方案，除了 Acegi 安全框架外，还包括用于持久层的 Hibernate、用于全文搜索的 Lucene 和 Compass、用于视图的模版引擎 Velocity 等。读者可以从中体会到 Spring 框架各主要模块的应用，并学习到 Spring 强大的集成能力。

第 11 章

Spring 2.0 实战：Live在线书店

本书将介绍应用 Spring 2.0 框架开发的一个完整的 Web 应用程序——Live 在线书店。读者可以从 http://www.livebookstore.net 访问该应用程序。Live 在线书店和大多数网上书店类似，这个 Web 应用包含以下功能。

（1）允许用户分类浏览书籍信息。

（2）允许用户注册、登录及修改口令。

（3）允许用户收藏喜爱的书籍。

（4）允许用户在线下订单。

（5）允许管理员管理书籍、用户和订单。

（6）帮助信息。

Live 在线书店虽然是一个较为简化的系统，例如，它不包含复杂的订单管理和配送系统，但是，作为一个完整的示例，读者可以从中体会到如何以 Spring 框架为基础，快速开发出可靠的、灵活的、可扩展的多层应用程序。

11.1 配置开发环境

在前面的章节中，我们的示例程序都采用 Eclipse 3.2 这个免费而强大的 IDE 作为开发环境，并全部使用 JDK 5.0 平台，服务器则使用免费的 Resin 3.1 版本。对于 Live 在线书店应用也不例外。此外，在开发阶段，虽然我们使用 Windows XP SP2 作为系统平台，但是由于 Java 应用程序良好的跨平台性，我们可以轻易地将其移植到 Linux/UNIX 系统上。

此外，Live 在线书店还必须要有一个后端数据库。前面我们已经介绍并采用了 HSQLDB 这个纯 Java 编写的免费而小巧的数据库，在开发阶段，我们完全可以采用它作为数据库。在实际应用时，推荐免费的 MySQL 5 数据库，当然也可以使用商用的 Oracle 等数据库。读者也可以一开始就采用 MySQL 作为数据库，Live 在线书店默认使用 HSQLDB 作为数据库是为了免去安装和配置数据库的麻烦。

11.1.1 创建项目目录结构

我们在 Eclipse 中创建一个 livebookstore 工程，其结构如图 11-1 所示。

其中，各主要目录如下。

（1）src 目录：存放所有的 Java 源代码。

（2）test 目录：存放所有的 JUnit 测试代码。

（3）conf 目录：存放所有的配置文件，其中，unused 子目录存放 Hibernate 的配置文件，这是为了在 Eclipse 中调试 HQL 用的，并非在编译和运行期使用。

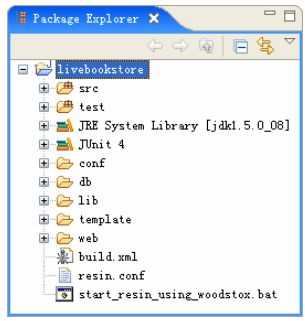


图 11-1

（4）db 目录：存放 HSQLDB 的数据文件。

（5）lib 目录：存放所有需要的第三方 jar 包，其中，core 子目录存放运行时必需的 jar 包，ext 子目录存放编译期必需的 jar 包，src 子目录存放某些 jar 包的源代码，这纯粹是为了在 Eclipse 中能方便地查看某些库的源代码，例如，Spring 和 Hibernate 的源代码。

（6）template 目录：存放自定义的 XDoclet 的模版文件。

（7）web 目录：存放所有的 Web 资源，包括 HTML、CSS、JavaScript 等。

11.1.2 配置HSQLDB数据库

为了配置 HSQLDB 数据库，只需要确保 hsqldb.jar 位于/lib/core/目录下，然后编写一个批处理 db_start.bat 启动数据库。

```
@SET CLASSPATH=.\lib\core\hsqldb.jar
java org.hsqldb.Server -database.0 file:.\db\bookstore -dbname.0 bookstore
```

结束数据库可以直接在控制台按下 Ctrl+C 组合键。

HSQLDB 还提供一个 Swing 界面的管理程序，要运行该管理程序，编写一个批处理 db_manager.bat 启动它。

```
SET CLASSPATH=.\lib\core\hsqldb.jar
java org.hsqldb.util.DatabaseManagerSwing %1 %2 %3 %4 %5 %6 %7 %8 %9
```

在管理界面中可以执行 SQL 语句，可以在此创建数据库表并插入测试数据等。

11.1.3 编写build.xml

由于 livebookstore 工程比较复杂，单纯依赖 Eclipse 的 Java Builder 是不能完成整个构建任务的。因此，我们采用 Ant 作为项目构建工具，编写 build.xml 脚本，分解成若干

个小的任务，互相依赖并完成最终的构建任务。

整个 build.xml 中定义的 Ant 任务依赖关系如图 11-2 所示。

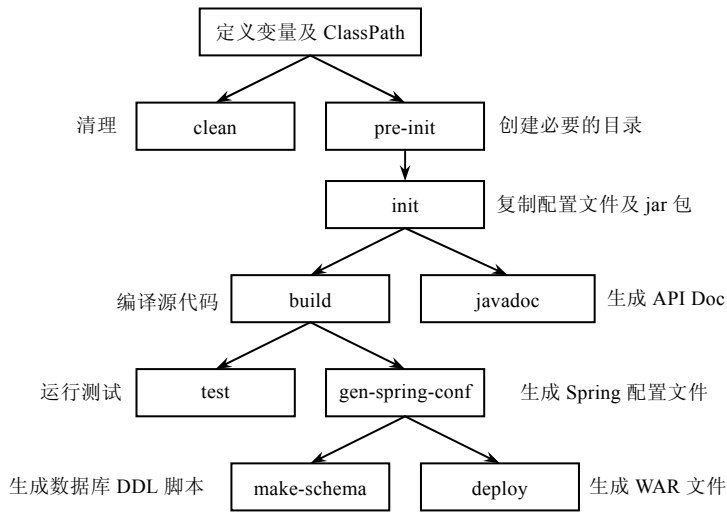


图 11-2

默认的任务是 deploy，因此，运行 Ant 后会生成一个完整的可直接运行的 Web 应用程序。

11.1.4 使用XDoclet自动生成配置文件

通过 XDoclet 自动生成 Spring 所需的 XML 配置文件能大大减少手动维护 XML 配置文件的工作量，但是，其他组件（例如，Spring 自身的 Bean 或第三方的 Bean）则需要手动编写到 XML 配置文件中，这部分配置文件必须固定下来。因此，在考虑使用 XDoclet 自动生成 Spring 的配置文件前，我们需要仔细考虑如何拆分 Spring 的配置文件，如何嵌入固定的 XML 配置文件。我们将整个配置文件分为 3 部分：services.xml、dispatcher-servlet.xml 和 security.xml，services.xml 和 dispatcher-servlet.xml 还引用了由 XDoclet 自动生成的 services-import-beans.xml 和 dispatcher-servlet-import-beans.xml，如图 11-3 所示。

其中，services.xml 是为逻辑层和持久层编写的固定的配置文件，定义了 dataSource、mailSender 等逻辑层和持久层的 Bean，并导入了 services-import-beans.xml。dispatcher-servlet.xml 是为 Web 层编写的固定的配置文件，定义了 urlMapping、velocity Config 等 Web 层的 Bean，并导入了 dispatcher-servlet-import-beans.xml 和 xfire.xml。security.xml 是 Acegi 的配置文件，由于其相对独立，因此和我们自定义的 Bean 没有联系。

错误！

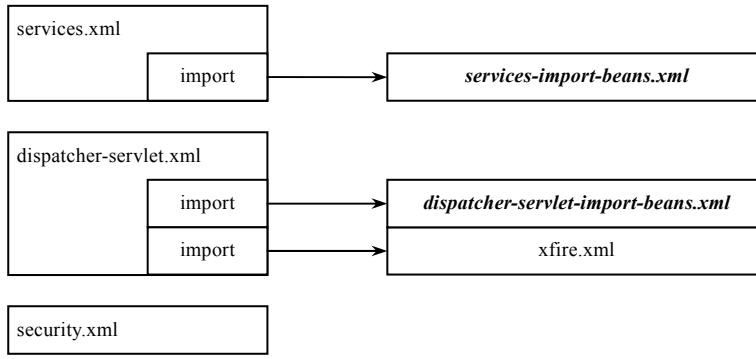


图 11-3

在第 8 章已经介绍了 xfire.xml 是使用 XFire 提供 Web 服务必须的配置文件，该文件由 XFire 提供，我们只需要导入即可。需要使用 XDoclet 自动生成的 XML 配置文件是 services-import-beans.xml 和 dispatcher-servlet-import-beans.xml。

此外，在第 3 章我们讲到了如何将某些配置信息从 Spring 的 XML 配置文件中提取出来，放到外部的 properties 文件中。在 Live 在线书店应用中，我们将 JDBC 和 SMTP 的相关配置放到外部 properties 文件中，在 services.xml 中引用它们。

jdbc.properties 配置如下。

```
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost/bookstore
jdbc.username=sa
jdbc.password=
```

smtp.properties 配置如下。

```
smtp.host=smtp.javaeedev.com
smtp.port=25
smtp.username=service@javaeedev.com
smtp.password=livebookstore
smtp.auth=true
smtp.encoding=GBK
```

读者需要根据自己的配置修改上述 properties 配置文件。

11.2 三层应用程序模型

分层结构是基于B/S架构的JavaEE应用程序的标准模式。通过对应用程序划分层次，可以获得各层清晰的功能和职责，简化代码的实现难度，并具有较高的扩展性。

通常，JavaEE 应用程序可分为 3 层：表示层、逻辑层和持久层，如图 11-4 所示。

错误！

图 11-4

持久层负责实现所有的数据访问功能，它将上层传入的数据写入到持久化存储系统中，并根据上层的要求读取或修改现有的数据。

最常见的持久化存储系统就是广泛使用的关系数据库。几乎所有的 Web 应用程序都离不开后台关系数据库的支持。不幸的是，面向对象的设计方法要求以对象为中心建模，这样才能设计出面向对象的、结构优雅的应用程序，而作为数据的最终存储地，数据库系统则要求以关系模型的表结构来储存应用程序的数据。这种对象-关系模型的“阻抗不匹配”成了持久层设计中最大的困难。

JavaEE 领域出现了大量解决对象-关系映射（O/R Mapping）的方案，例如，前面讲到的 Hibernate，它们都试图透明地完成对象到关系的双向映射。在一个 ORM 框架中，对象传入到 ORM 系统中，然后被自动转换为数据库表的记录并写入数据库。相反地，从数据库读出的表记录经过 ORM 系统就变成了可供应用程序直接使用的对象。

逻辑层负责完成应用程序的逻辑功能，包括调用持久层完成实体对象的存取、安全检查，事务控制等。抽象出逻辑层的好处是将应用程序的逻辑功能从表示层中剥离，这样就能复用逻辑层的功能，例如，增加新的应用程序用户接口（如 Web 服务）不会影响到表示层。此外，逻辑层也可以看作是对持久层的一个门面模式，简化表示层对这些逻辑功能的调用。

表示层则是与用户打交道的 UI 界面。通常，在 JavaEE 三层应用程序中，网页是最常见的用户界面，因为用户只需要一个浏览器即可以访问应用程序，因此部署成本最低。此外，还可以用 Swing 等作为表示层。

11.2.1 Java包结构

我们按照应用程序的层次结构来组织 Java 包，并将接口和实现类分别放在不同的包中。例如，持久层接口的定义全部在 net.livebookstore.dao 包中，而具体的实现类则放在 net.livebookstore.dao.hibernate 包中。逻辑层组件与之类似，接口全部放在 net.livebookstore.business 包中，而具体的实现类则放在 net.livebookstore.business.impl 包中。这样做有利于将接口和实现分离。

11.3 域模型设计

传统的应用程序总是先设计好数据库的表结构，然后根据表的结构设计应用程序中持有数据的对象。这种设计方式本质上仍是以关系数据库为基础的，不符合面向对象的设计方式，如图 11-5 所示。

在 Live 在线书店应用中，我们从域模型入手，首先设计域对象的模型，并确定其关联关系，然后通过 Hibernate 提供的 DDL 工具，自动地将域模型映射为数据库的表结构。这种方式符合面向对象的设计方法，而且更容易建立 UML 模型，如图 11-6 所示。

错误！

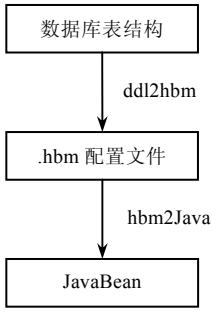


图 11-5

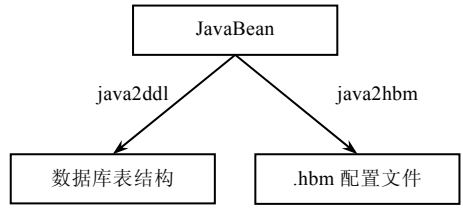


图 11-6

从 JavaBean 映射到数据库的表结构比从数据库的表结构映射到 JavaBean 要容易得多。作为实体对象的 JavaBean 非常简单，很容易维护，而数据库表的修改则不那么容易。

由于采用了 Hibernate 3.2 版本及 Java 5 Annotation 支持，使得传统的对应每个域对象的 Hibernate 映射文件（.hbm）不再需要了。这使得代码更容易被维护，因为同步代码和注解要比同步代码和 XML 配置文件容易。

如果读者仍使用 JDK 1.4.x 或更早的版本，无法使用 Hibernate 提供的 Annotation 支持，则需要考虑使用 XDoclet 自动生成.hbm 配置文件以减轻工作量。

考虑到 Live 在线书店的功能，我们设计了以下几个实体对象。

1．Account

Account 实体代表一个用户，由于用户的登录名是唯一的，因此可直接用作主键，privilege 属性用来标识用户权限，在这个系统中，只定义了普通用户和管理员两种类型，其他的属性（如地址、邮编、姓名、电话等）是为了让用户每次下订单时自动填入默认的送货信息，避免用户重复输入。

2．Book

Book 实体代表一本书，书名、作者、出版社、价格等都是很容易想到的属性。由于书名可能重复，因此作为主键就是不合适的，我们需要一种自动生成的主键。在第 5 章中，我们已经讨论了几种主键生成方式，并推荐 UUID 作为首选的主键生成策略，因为 UUID 不仅生成简单，而且由于其全球唯一性，在集群环境下也完全能够正常工作。

我们首先定义了一个 UUIDSupport。

```
public abstract class UUIDSupport {
    protected String id;
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
}
```

凡是以 UUID 作主键的实体对象均从此类派生，因此，Book 对象采用 UUID 作为主键，定义如下。

```
public class Book extends UUIDSupport {
    private String name;
    private String originalName;
    private int categoryId = Category.ROOT_ID;

    private String author;
    private String publisher;
    private String language;

    private float price;
    private int discount; // 1-100, default 100(no discount).
    private String isbn;
    private Date pubDate;
    private String description;

    private int state;
    private int stock;
    private int sold;

    private int rating;    // sum of all rating
    private int ratingCount; // how many user rate this book
}
```

rating 和 ratingCount 属性用于保存用户对书籍评价的总分和人次，这样就可以计算出书籍的平均分。

3．Category

Category 实体代表一个书籍分类，由于书籍分类是一种树型结构，并非关系数据库表的二维结构，因此，需要以一种合适的方式保存这种树状结构。

一种方式是为每个分类指定其父分类的 ID，对应的表结构如表 11-1 所示。

Id	Name	ParentId
aaa	Java	(null)
bbb	JavaEE	aaa
ccc	JavaME	aaa

这种方式表示直观，但是查询某一分类的书籍会比较麻烦。例如，“Struts”和“EJB”这两本书应归入 JavaEE 分类，由于 JavaEE 是 Java 的子分类，所以用户查询 Java 分类也应当能查到这两本书。如果用户浏览 Java 分类，就至少需要进行 3 次查询，分别查询 Java、JavaEE、JavaME 这 3 个分类下的所有书籍。

另一种方式是不指定父分类 ID，而是通过整型 ID 本身来表示这种层次关系。这种方式类似于 IP 地址的掩码，例如，Java 分类的 ID 为 0x0A00，则 JavaEE 和 JavaME 的分类为 0x0A01 和 0x0A02，用户查询 Java 分类时，通过使用掩码就可以一次查询到所有的书籍。

```
select * from Book where Book.categoryId & 0xFF00 = 0x0A00
```

这种方式的缺点是分类的树型层次深度是一定的，且每个节点能够包含的最大子节点的个数也是有限的。例如，采用 int（32 位）作为主键类型，定义每一层占用的位数为 8 位，则树的层次最多为 4 层，每个节点下最多可以有 255 个子节点，这对于 Live 在线书店应用来说已经足够了。如果需要更多的层次，可以采用 long（64 位）。Live 在线书店采用 int 是因为 HSQLDB 数据库对于 long 类型的“&”运算支持不好。

我们将整个分类定义为 4 层，并手动指定根节点的 ID 为 0x01000000。

```
public class Category {
    public static final int ROOT_ID = 0x01000000;
    private static final int[] MASK = {
        0xFF000000,
        0xFFFF0000,
        0xFFFFFF00,
        0xFFFFFFFF
    };

    private Integer id; // PK
    private String name;
```

```
    private int categoryOrder;
    private int mask;
    private int level;
    private int parentId = 0; // parentId;
    private List<Category> children = new ArrayList<Category>();
}
```

MASK 定义了每一层节点的掩码常量，除了 id、name 和 categoryOrder 是直接映射到数据库表外，mask、level、parentId 和 children 都是在初始化时设定的。

4．FavoriteBook

FavoriteBook 实体用于保存用户的收藏夹，除了包含 account 和 book 两个属性外，还增加了一个 createdDate 用于按照用户添加书籍到收藏夹的时间进行排序。

```
public class FavoriteBook extends UUIDSupport {
    private Account account;
    private Book book;
    private Date createdDate;
}
```

5．Order 和 OrderItem

Order 实体用于保存用户的一个订单，一个 Order 对象还包含若干 OrderItem 对象，表示订单每一项的书籍和数量。

```
public class Order extends UUIDSupport {
    private Account account; // FK
    private List<OrderItem> orderItems;

    // default value should be copied from Account:
    private String address;
    private String zip;
    private String name;
    private String telephone;
    private String mobile;

    private Date createdDate;
    private int deliver; // 送货方式
    private int payment; // 付款方式
    private int state; // 订单状态
}
```

OrderItem 保存了 Order 的每一项细节。

```
public class OrderItem extends UUIDSupport {
    private Order order;
```

```
    private Book book;
    private int number;
}
```

6．Comment

Comment 实体用于表示一条用户评论，除了包含 Account 和 Book 这两个属性外，还需要保存评论内容、评分和发表时间。

```
public class Comment extends UUIDSupport {
    private Book book;
    private Account account;
    private int rating;
    private String content;
    private Date createdDate;
}
```

在第 5 章中，虽然我们已经讨论了 DAO 模式和简单的 O/R 映射，但是，还没有涉及更复杂的实体的关联，例如，一对多、多对多关系模型。

在域模型中，实体之间存在以下几种关系。

（1）一对多关系

一对多关系和多对一关系是一样的，Account 实体和 Order 实体就是一个一对多关系，一个用户可以有多个订单，但是每个订单只能对应一个用户。反过来，Order 实体和 Account 实体就是多对一关系，因此，一对多和多对一是一个概念。为了在两个实体之间建立一对多关系，我们需要在 Order 实体中定义一个 Account 类型的属性，以便可以通过 Order 实体获得相应的 Account 实体。

```
public class Order extends UUIDSupport {
    private Account account;
    public Account getAccount() { return account; }
    public void setAccount(Account account) { this.account = account; }
}
```

这样，一旦获得了 Order 对象，就可以通过 Order 对象的 getAccount()方法获得与之关联的 Account 对象，而 setAccount()方法用于创建 Order 对象和 Account 对象的关联关系，如果需要，Hibernate 在读取 Order 对象时就可以同时读取相应的 Account 对象，再通过 setAccount()方法将 Account 对象注入到 Order 对象中，完成整个 Order 对象的初始化。

现在我们可以从 Order 对象获得 Account 对象，如果已有了 Account 对象，需要获得其对应的所有的 Order 对象，该怎么办？这时，需要在 Account 实体中定义一个集合类型，并映射到所有其对应的 Order 对象中。

```
public class Account {
```

```
    private Set<Order> orders;
    public Set<Order> getOrders() { return orders; }
    public void setOrders(Set<Orders> orders) { this.orders = orders; }
}
```

集合类型可以是 Set，表示无序的集合；也可以是 List，表示有序的集合。可以同时在 Account 实体和 Order 实体中定义这两种导航关联，我们将其称为双向关联，这样既可以通过 Order 对象获得对应的 Account 对象，也可以通过 Account 对象获得对应的所有 Order 对象。如果只在一个实体中定义关联，我们称之为单向关联，例如，仅在 Order 实体中定义 account 属性，就只能通过 Order 对象获得对应的 Account 对象，而不能通过 Account 对象获得对应的所有 Order 对象。如果需要通过 Account 对象获得对应的 Order 对象，就需要进行额外的查询操作，而关联导航是由 ORM 框架自动进行查询完成的。

在关系数据库中，一对多关系是通过外键表示的。由于 Account 实体和 Order 实体之间建立了一对多关系，因此，在其对应的数据库表中，Order 表将包含一个外键约束，指向 Account 表的主键，如图 11-7 所示。

错误！

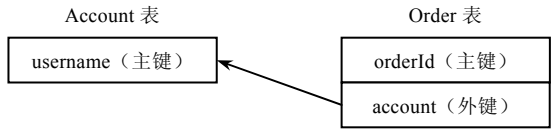


图 11-7

同样，Order 实体和 OrderItem 实体也是一对多关系，我们在这两个对象之间建立的映射如下。

```
public class Order extends UUIDSupport {
    private List<OrderItem> orderItems;

    @OneToMany(targetEntity=OrderItem.class, mappedBy="order")
    public List<OrderItem> getOrderItems() { return orderItems; }
    public void setOrderItems(List<OrderItem> orderItems) { this.orderItems =
orderItems; }
}
```

一对多关系中，我们需要为 oderItems 属性标注一个**@OneToMany** 的 JPA 注解，表示该属性是一个一对多关系，并通过 targetEntity 告诉 ORM 框架该集合的元素类型为 OrderItem。

在 OrderItem 中，需要定义多对一关系。

```
public class OrderItem extends UUIDSupport {
    private Order order;
```

```
    @ManyToOne
    @JoinColumn(nullable=false, updatable=false)
    public Order getOrder() { return order; }
    public void setOrder(Order order) { this.order = order; }
}
```

多对一关系中，要使用**@ManyToOne** 的 JPA 注解，表示该属性是一个多对一关系。和一对多不同，不需要告诉 ORM 框架多对一中的“一”是什么类型的实体，因为 ORM 框架可以从属性类型判断多对一中的“一”的类型。此外，为了描述其底层数据库表的列属性，不能使用普通的@Column 描述，而必须使用@JoinColumn 来描述，告诉 ORM 框架该属性将映射到数据库表的一个外键。这里我们设置@JoinColumn(nullable=false, updatable=false)，表示该属性不允许为空，并且一旦创建后就不允许更改，ORM 框架在执行更新该实体的操作时就会忽略这个属性，这样也可以避免在代码中错误地设置了该属性，或者由于安全漏洞导致恶意用户修改了该属性。

（2）多对多关系

考虑 Account 实体和 Book 实体，如果要表示用户的收藏夹，就需要在 Account 实体和 Book 实体之间建立多对多关系，因为一个用户可以收藏多本书籍，而一本书籍也可以被多个用户收藏。要建立多对多关系，可以直接在 Account 实体和 Book 实体分别建立相应的集合类型，以便通过 Account 对象获得 Book 对象的集合，或者通过 Book 对象获得 Account 对象的集合。不过，从关系数据库的角度看，要表示多对多关系，必须将其拆为两个一对多关系，利用一个中间表，建立 Account 表和 Book 表的多对多关系，如图 11-8 所示。

错误！

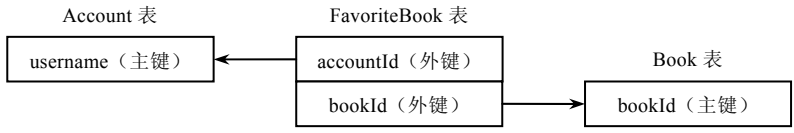


图 11-8

如果将多对多拆为两个一对多关系，就可以通过 Account、Book 和 FavoriteBook 这 3 个实体表示 Account 和 Book 的多对多关系，只需要在 Account 和 FavoriteBook，以及 Book 和 FavoriteBook 之间都建立一对多关系。Live 在线书店采用的就是这种方式，因为我们还需要为 FavoriteBook 添加额外的信息以保存用户将书籍加入收藏夹的时间，以便按照添加的顺序向用户返回收藏夹的书籍列表。

（3）一对一关系

相比一对多和多对多关系，一对一关系就简单得多。事实上，如果两个实体具有一

对一关系，则完全可以将其合并为一个实体，这样，无论是实体的设计还是数据库表的结构都可以最大限度地简化。不过，某些时候，一对一关系也是必要的。例如，随着业务的发展，Live 在线书店需要为用户提供博客空间，以便让用户发表一些书评。为了避免修改现有的 Account 实体和对应的底层数据库表，可以建立一个新的 Blog 实体，并与 Account 实体建立一对一关系，如图 11-9 所示。

错误！

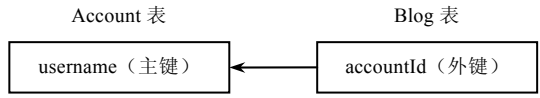


图 11-9

通过一对一关系，可以在用户需要时才为用户建立 Blog，并且避免了修改现有的实体模型和数据库表。

以下对象就是 Live 在线书店用到的全部实体对象，其关联关系如图 11-10 所示。

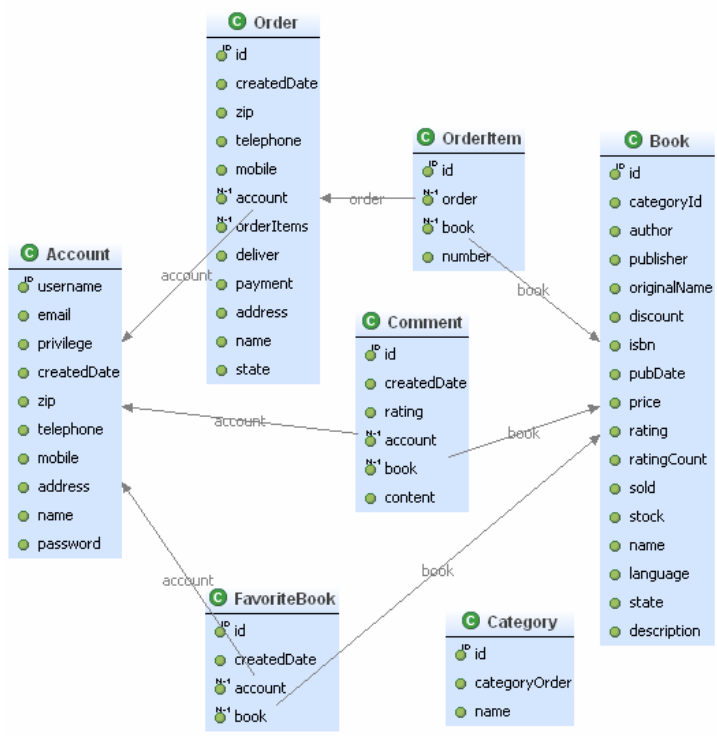


图 11-10

由于采用了 Hibernate 3.2 作为持久化机制，并且使用 JPA 注解来定义实体到数据库表的映射关系，使得应用程序更容易维护。

11.3.1 生成数据库表结构

有了域对象模型后，我们就可以利用工具来实现域对象模型到数据库关系模型的自动转化，从而获得数据库表结构的 DDL 脚本。Hibernate 提供了一个有用的工具——HibernateTools 来完成这一转化。HibernateTools 以 Ant 扩展任务的形式来实现 DDL 的自动生成，不过它需要一个额外的 Hibernate 配置文件，因此，我们需要以下两个步骤。

① 编写 hibernate.cfg.xml 配置文件。虽然在 Spring 框架中我们不必编写该配置文件就可以直接生成 Hibernate 的 SessionFactory，但是，在运行 Ant 脚本时，并没有任何 Spring 的 IoC 容器环境，因此，HibernateTools 需要根据该文件来创建 SessionFactory。我们将该文件放在/conf/unused/目录下，并使用 Annotation 配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="Bookstore">
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</
property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost/
bookstore</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <property name="dialect">net.livebookstore.hibernate.Custom
SQLDialect</property>
        <property name="hibernate.cache.provider_class">org.hibernate.
cache.HashtableCacheProvider</property>
       <property name="hibernate.transaction.factory_class">org.hibernate.
transaction.JDBCTransactionFactory</property>

        <!-- mapping files -->
        <mapping package="net.livebookstore.domain"/>
        <mapping class="net.livebookstore.domain.Account"/>
        <mapping class="net.livebookstore.domain.Book"/>
        <mapping class="net.livebookstore.domain.Category"/>
        <mapping class="net.livebookstore.domain.Comment"/>
        <mapping class="net.livebookstore.domain.FavoriteBook"/>
        <mapping class="net.livebookstore.domain.Order"/>
        <mapping class="net.livebookstore.domain.OrderItem"/>
    </session-factory>
```

```
</hibernate-configuration>
```

② 编写 Ant 任务，使用<hbm2ddl>生成 DDL 脚本。这里，需要确保 Ant 脚本中设置的 Classpath 包含 hibernate-tools.jar。我们将 hibernate-tools.jar 放入/lib/ext/目录下，因为该文件只在使用 Ant 生成 DDL 脚本时需要，在 Live 在线书店运行时并不需要。Ant 脚本中“make-schema”定义如下。

```
<target name="make-schema" depends="gen-spring-conf">
    <!-- 定义 task -->
    <taskdef name="hibernatetool"
        classname="org.hibernate.tool.ant.HibernateToolTask">
        <classpath refid="build-classpath"/>
    </taskdef>
    <taskdef name="annotationconfiguration"
        classname="org.hibernate.tool.ant.AnnotationConfigurationTask">
        <classpath refid="build-classpath"/>
    </taskdef>
    <!-- 生成 DDL -->
    <hibernatetool destdir="${gen.dir}">
        <classpath refid="build-classpath"/>
        <annotationconfiguration
            configurationfile="${conf.dir}/unused/hibernate.cfg.xml"/>
        <hbm2ddl
            export="false"
            drop="false"
            create="true"
            delimiter=";"
            outputfilename="schema.sql"
            destdir="${gen.dir}"
        />
    </hibernatetool>
</target>
```

自动生成的 DDL 脚本放在/auto-gen/schema.sql 文件中。可以使用文本编辑器查看，HibernateTools 生成的数据库表的创建脚本如下。

```
create table t_account ...
...
alter table t_comment ...
...
```

create 语句将首先创建所有的表结构，然后，alter 语句定义各表之间的外键关联。各表的字段类型和我们在域对象中定义的属性类型一致，字段大小由 HibernateTools 根据 JPA 注解自动判断，因此，从域对象模型到关系数据库模型的映射是完全一致的。

使用合适的数据库工具运行这个 DDL 脚本就完成了表结构的创建，我们甚至都没有关心 SQL 语句的具体内容。

11.4 持久层设计

在第 5 章中，我们已经初步介绍了 Hibernate 这个功能强大的 ORM 框架，Live 在线书店仍然采用 Hibernate 作为持久化解决方案。DAO 模式仍是持久层的标准模式，不同的是，我们不采用 Spring 提供的现成的 DAO 体系，而是设计一个类型安全的泛型 DAO，通过泛型 DAO，能够将公共代码以范型方式放入范型 DAO 超类中，从而进一步减少代码量。

对于每一个 Domain Object 来说，至少要实现基本的 CRUD 操作，即 Create（创建）、Retrieve（获取）、Update（更新）和 Delete（删除）4 种操作。我们将这 4 种操作全部以泛型方式实现，大大简化了各个子类的编码，同时还获得了类型安全的特性。

泛型 DAO 的核心是定义一个 GenericDao 接口，申明 CRUD 操作。

```
public interface GenericDao<T> {
    // 通过主键查询 T:
    T query(Serializable id);
    // 创建新的 T:
    void create(T t);
    // 删除 T:
    void delete(T t);
    // 更新 T:
    void update(T t);
}
```

然后，提供一个默认的 GenericHibernateDao 实现类，实现所有的 CRUD 操作，但不必实现 GenericDao 接口。

```
public abstract class GenericHibernateDao<T> {
    private final Class<T> clazz;
    protected HibernateTemplate hibernateTemplate;

    public GenericHibernateDao(Class<T> clazz) {
        this.clazz = clazz;
    }
    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }
    // 根据主键查询 T:
    public T query(Serializable id) {
        // 用 get()而不用 load()是因为 load()方法返回的是延迟加载的对象，
```

```
        // 可能造成 LazyInitializationException:
        T t = (T)hibernateTemplate.get(clazz, id);
        if(t==null)
            throw new DataRetrievalFailureException("Object not found.");
        return t;
    }
    // 创建 T:
    public void create(T t) {
        hibernateTemplate.save(t);
    }
    // 删除 T:
    public void delete(T t) {
        hibernateTemplate.delete(t);
    }
    // 更新 T:
    public void update(T t) {
        hibernateTemplate.update(t);
    }
}
```

GenericHibernateDao 受到 Hibernate 的唯一限制是必须获得域对象的 Class 实例，由于无法直接调用 **T.class**，因此，一个变通的方法是从构造方法的参数中传入 Class 实例，对于子类的继承会稍微麻烦一点。

对于真正的 DAO 接口，由 GenericDao 接口扩展，保证了类型安全。例如，对于 BookDao，由于扩展自 GenericDao<Book>，因此，定义的 CRUD 操作即为类型安全的，此外，还可以定义其他查询操作。

```
public class BookDaoImpl extends GenericHibernateDao<Book>
        implements BookDao {
    public BookDaoImpl() {
        super(Book.class);
    }
    // 基本的 CRUD 操作已经实现了!
    // 定义额外的 query 操作:
    public List<Book> query(final Category c, final Page page) {
        // TODO:
    }
}
```

BookDao 的实现类 BookDaoImpl 实现了 BookDao 接口，这是类型安全的。此外，BookDaoImpl 还从 GenericHibernateDao 扩展而来，因此，基本的 CRUD 操作已经全部实现了，BookDaoImpl 只需实现 BookDao 额外定义的查询操作。由于 Spring 提供的 HibernateTemplate 已被注入到 GenericHibernateDao 中，因此，BookDaoImpl 可以直接使

用 HibernateTemplate 来实现额外定义的查询操作。

这个泛型 DAO 的详细模式如图 11-11 所示。

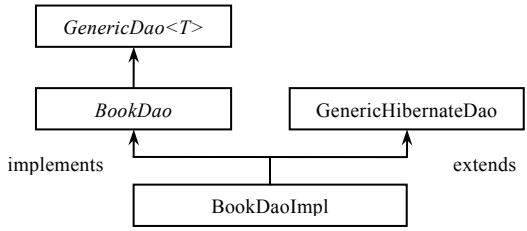


图 11-11

泛型 DAO 模式的最大的好处是消除了每个 DAO 对象中重复的 CRUD 操作，这些重复的 CRUD 操作被统一放入 GenericHibernateDao，以泛型方式实现了。

子类如果不希望继承超类的某个方法，例如，CommentDaoImpl 不希望客户端去调用 update()方法，就可以简单地覆盖它，直接抛出 UnsupportedOperationException 异常。

```
@Override
public void update(Comment t) {
    throw new UnsupportedOperationException();
}
```

将覆写的方法加上注解@Override，维护源代码时，很容易知道该方法覆盖了超类的相同签名的方法。使用 Eclipse 的菜单命令“Source”→“Override/Implements Methods...”生成的方法签名就会被自动标记为@Override。这是在 Java 5 中的一种良好的编程习惯。

在持久层中，我们一共定义了 5 个 DAO 对象，其层次关系如图 11-12 所示。

错误！

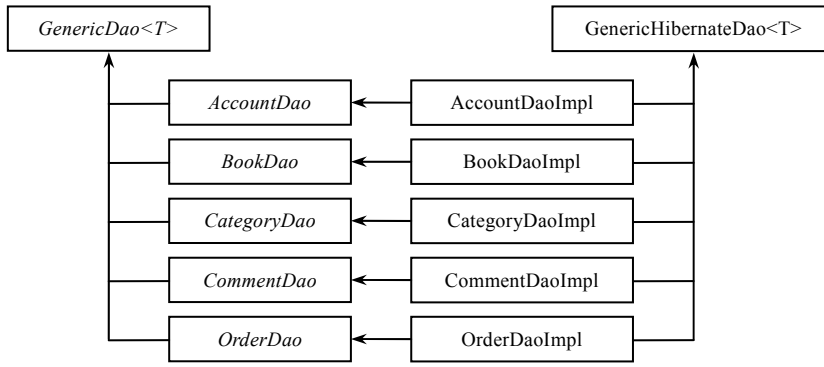


图 11-12

HibernateTemplate 对象是注入到 GenericHibernateDao<T>中的，因此，所有的实现

类都可以直接引用。注意到我们没有对 FavoriteBook 和 OrderItem 对象定义 DAO 操作，这两个对象的相关操作分别被定义在 BookDao 和 OrderDao 中。

现在我们设计好了各个 DAO 组件，下一步就需要在 Spring 的 XML 配置文件中装配起来。对于持久层来说，需要装配的一共有以下组件。

（1）DataSource：通过 Spring 提供的 DriverManagerDataSource，我们可以很容易地配置一个 DataSource 供开发和测试使用。在实际部署时，在服务器上配置好 DataSource，然后应用 JndiObjectFactoryBean 获得 DataSource 即可。

```
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="${jdbc.driver}" />
   <property name="url" value="${jdbc.url}" />
   <property name="username" value="${jdbc.username}" />
   <property name="password" value="${jdbc.password}" />
</bean>
```

JDBC 连接的配置信息放在外部 jdbc.properties 文件中，应用第 3 章介绍的 PropertyPlaceholderConfigurer 可以很容易地引入到 Spring 的配置文件中。

（2）SessionFactory：使用 AnnotationSessionFactoryBean 可以直接在 Spring 中配置一个 SessionFactory，而不必使用 Hibernate 特有的 hibernate.cfg.xml 配置文件。

```
<bean id="sessionFactory"  class="org.springframework.orm.hibernate3.
annotation.AnnotationSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="annotatedClasses">
      <list>
         <value>net.livebookstore.domain.Account</value>
         <value>net.livebookstore.domain.Book</value>
         <value>net.livebookstore.domain.Category</value>
         <value>net.livebookstore.domain.Comment</value>
         <value>net.livebookstore.domain.FavoriteBook</value>
         <value>net.livebookstore.domain.Order</value>
         <value>net.livebookstore.domain.OrderItem</value>
      </list>
   </property>
   <property name="annotatedPackages">
      <list>
         <value>net.livebookstore.domain</value>
      </list>
   </property>
   <property name="hibernateProperties">
      <props>
```

```
                <prop key="hibernate.dialect">
                    net.livebookstore.hibernate.CustomSQLDialect
                </prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.jdbc.fetch_size">10</prop>
                <prop key="hibernate.cache.provider_class">
                    org.hibernate.cache.HashtableCacheProvider
                </prop>
            </props>
        </property>
        <property name="eventListeners">
            <map>
                <entry key="pre-update">
                    <bean class="org.hibernate.validator.event.ValidatePreUpdate
EventListener" />
                </entry>
                <entry key="pre-insert">
                    <bean class="org.hibernate.validator.event.ValidatePreInsert
EventListener" />
                </entry>
            </map>
        </property>
    </bean>
```

（3）HibernateTemplate：由于我们的每个 DAO 组件并没有从 Spring 的 Hibernate DaoSupport 中派生，因此，需要定义一个 HibernateTemplate 实例，然后注入到每个 DAO 组件中。

```
    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory" ref="sessionFactory" />
        <property name="fetchSize" value="10" />
    </bean>
```

（4）HibernateTransactionManager：用于管理 Hibernate 事务，在这里我们只需配置这个 Bean，就可以直接使用声明式事务管理。

```
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
```

（5）各 DAO 组件：由于我们直接在 GenericHibernateDao 中通过 XDoclet 注释注入了 HibernateTemplate：

```
public abstract class GenericHibernateDao<T> {
    protected HibernateTemplate hibernateTemplate;
    /**
     * @spring.property name="hibernateTemplate" ref="hibernateTemplate"
     */
    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }
    ...
}
```

因此，在各个 DAO 的定义处加上**@spring.bean** 的注释，再运行 Ant，即可自动生成 DAO 组件的配置信息并自动注入 HibernateTemplate。

11.4.1 与运算(&)的实现

由于 Hibernate 3.2 不支持“&”运算，但实际上大部分数据库都支持“&”运算，例如，MySQL 支持“a & b”，而 HSQLDB 是通过“BITAND(a, b)”函数提供的“&”运算。因此，我们需要扩展 Hibernate 3.2，使其支持“&”运算，这样才能根据 Category 对象获得当前分类及其子类的所有书籍。

虽然 Hibernate 也支持直接执行原始的 SQL 语句，但是这样就丧失了 O/R Mapping 的能力，并且需要更多的转化工作。我们希望能直接在 HQL 语句中支持“&”运算。幸运的是，Hibernate 框架的设计非常具有扩展性。Hibernate 对不同数据库的“方言”支持就可以解析某种数据库的特定 SQL 函数，我们只需要利用 Hibernate 的自定义函数机制，自行编写一个 bitand()函数，将其解析为数据库对应的 SQL 语句，即可实现“&”运算。

Hibernate 通过 SQLFunction 接口实现自定义 SQL 函数，我们定义一个 BitAndFunction 如下。

```
public class BitAndFunction implements SQLFunction {
    // 根据需要返回 SQL 数据类型:
    public Type getReturnType(Type type, Mapping mapping) {
        return Hibernate.INTEGER;
    }

    public boolean hasArguments() {
        return true;
    }

    public boolean hasParenthesesIfNoArguments() {
        return true;
```

```
            }

            public String render(List args, SessionFactoryImplementor factory) throws
QueryException {
                if (args.size() != 2) {
                    throw new IllegalArgumentException("BitAndFunction requires 2
arguments!");
                }
                return args.get(0).toString() + " & " + args.get(1).toString();
            }
        }
```

对于 HQL 语句，上述自定义 SQL 函数会将“bitand(a,b)”翻译成“a & b”，这样，大多数支持“&”运算的数据库就可以正确执行。

由于 HSQLDB 比较特殊，它不是通过“&”实现的与运算，而是提供了一个 BITAND() 函数，因此，再定义一个 HsqlBitAndFunction。

```
        public class HsqlBitAndFunction extends BitAndFunction {
            public String render(List args, SessionFactoryImplementor factory) throws
QueryException {
                return "BITAND(" + args.get(0).toString() + ", " + args.get(1).toString()
+ ")";
            }
        }
```

现在，我们需要将自定义函数注册到 Hibernate 中，最简单的方法是从相应的方言派生一个自定义的 CustomSQLDialect，然后在构造方法中注册该 BitAndFunction。

```
        public class CustomSQLDialect extends HSQLDialect {
            public CustomSQLDialect() {
                super();
                LogFactory.getLog(CustomSQLDialect.class).info("Register bitand
function for bit-and operation. (e.g.: where a & b = :c)");
                if(HSQLDialect.class.isAssignableFrom(getClass()))
                    registerFunction("bitand", new HsqlBitAndFunction());
                else
                    registerFunction("bitand", new BitAndFunction());
            }
        }
```

在 Spring 的 Hibernate 相关配置中，将 dialect 指定为 CustomSQLDialect 就可以实现“&”运算。

11.4.2 分页的实现

分页是查询时最常见的操作。如果一次查询的数据过多，就很有必要分页显示给用户，一是因为一次查询数据量如果太大，例如，上万条记录，会对服务器造成很大的负担；二是用户希望看到的往往是最关心的少量数据，因此，应当尽量在前几页让用户看到他们最关心的数据。

实现分页查询的关键是获得所有符合条件的记录总数，这样就能根据每页的数量计算出页数。为此，我们设计一个 Page 对象，初始化每页需要显示的记录数和要显示的页号。

```
public class Page {
    public static final int DEFAULT_PAGE_SIZE = 10;

    private int pageIndex;
    private int pageSize;
    private int totalCount;
    private int pageCount;

    public Page(int pageIndex, int pageSize) {
        if(pageIndex<1)
            pageIndex = 1;
        if(pageSize<1)
            pageSize = 1;
        this.pageIndex = pageIndex;
        this.pageSize = pageSize;
    }
    public Page(int pageIndex) {
        this(pageIndex, DEFAULT_PAGE_SIZE);
    }

    public int getPageIndex() { return pageIndex; }
    public int getPageSize() { return pageSize; }
    public int getPageCount() { return pageCount; }

    public int getTotalCount() { return totalCount; }
    public int getFirstResult() { return (pageIndex-1)*pageSize; }

    public boolean getHasPrevious() { return pageIndex>1; }
    public boolean getHasNext() { return pageIndex<pageCount; }

    public void setTotalCount(int totalCount) {
```

```
            this.totalCount = totalCount;
            pageCount = totalCount / pageSize + (totalCount%pageSize==0 ? 0 : 1);
            if(totalCount==0) {
                if(pageIndex!=1)
                    throw new IndexOutOfBoundsException("Page index out of range.");
            }
            else {
                if(pageIndex>pageCount)
                    throw new IndexOutOfBoundsException("Page index out of range.");
            }
        }
        public boolean isEmpty() {
            return totalCount==0;
        }
    }
```

Page 在初始化时就确定了 pageSize 和 pageIndex 属性，待查询到记录总数后，通过 setTotalCount()方法就可以确定页数，从而通过 getFirstResult()返回第一行记录的起始位置。

为了在 Hibernate 中实现分页查询，还需要几个辅助方法。我们将这几个辅助方法放到 GenericHibernateDao 中，以方便子类调用。

queryForObject()方法用于执行一次查询，并返回一个唯一结果。

```
    protected Object queryForObject(final String select, final Object[] values) {
        HibernateCallback selectCallback = new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Query query = session.createQuery(select);
                if(values!=null) {
                    for(int i=0; i<values.length; i++)
                        query.setParameter(i, values[i]);
                }
                return query.uniqueResult();
            }};
        return hibernateTemplate.execute(selectCallback);
    }
```

queryForList(String, Object[], Page)方法实现一个分页查询。

```
    protected List queryForList(final String select, final Object[] values, final
Page page) {
        HibernateCallback selectCallback = new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Query query = session.createQuery(select);
                if(values!=null) {
```

```
                for(int i=0; i<values.length; i++)
                    query.setParameter(i, values[i]);
            }
            return query.setFirstResult(page.getFirstResult())
                        .setMaxResults(page.getPageSize())
                        .list();
        }
    };
    return (List) hibernateTemplate.executeFind(selectCallback);
}
```

另一个 queryForList(String, String, Object[], Page)重载方法实现了一个完整的分页查询，第一个查询定义了如何获得记录总数，然后填充到 Page 对象，再根据第二个查询获得指定页的记录。

```
    protected List queryForList(final String selectCount, final String select,
final Object[] values, final Page page) {
        Long count = (Long)queryForObject(selectCount, values);
        page.setTotalCount(count.intValue());
        if(page.isEmpty())
            return Collections.EMPTY_LIST;
        return queryForList(select, values, page);
    }
```

例如，要查询一本书的评论，由于评论可能很多，因此需要进行分页显示。queryComments()方法实现了这个功能。

```
    public List<Comment> queryComments(Book book, Page page) {
        return queryForList(
            "select count(*) from Comment as c where c.book=?",
            "select c from Comment as c where c.book=? order by c.createdDate desc",
            new Object[] {book},
            page
        );
    }
```

第一个“**select count(*) from ...**”查询定义了如何获得评论的总数，第二个“**select from ...**”查询根据 Page 对象的 pageIndex 和 pageSize 取出相应页面的评论。为了便于使用，这些辅助方法均定义在 GenericHibernateDao 中，子类可以方便地调用它们。

需要注意的是，与 Hibernate 3.1 及其以前的版本不同，从 Hibernate 3.2 开始，使用 count()等 SQL 函数返回的数据类型从 Integer 改为 Long，这是为了兼容 JPA 标准。

对于 Hibernate 来说，还提供了 Criteria 查询，通过 DetachedCriteria，可以先定义查询，然后关联 Session 执行查询。Criteria 可以通过投影操作方便地获得记录的总数，但

是，投影操作和查询的 Order 条件是冲突的。为了实现通过一条 DetachedCriteria 同时得到记录总数和对应页数的记录，可以通过反射实现。为此，封装一个 PaginationCriteria。

```
        class PaginationCriteria {
            private static Field orderEntriesField = getField(Criteria.class,
"orderEntries");
            public static List query(Criteria c, Page page) {
                // first we must detect if any Order defined:
                // Hibernate code: private List orderEntries = new ArrayList();
                List _old_orderEntries = (List)getFieldValue(orderEntriesField, c);
                boolean restore = false;
                if(_old_orderEntries.size()>0) {
                    restore = true;
                    setFieldValue(orderEntriesField, c, new ArrayList());
                }
                c.setProjection(Projections.rowCount());
                int rowCount = ((Long)c.uniqueResult()).intValue();
                page.setTotalCount(rowCount);
                if(rowCount==0) {
                    // no need to execute query:
                    return Collections.EMPTY_LIST;
                }
                // query:
                if(restore) {
                    // restore order conditions:
                    setFieldValue(orderEntriesField, c, _old_orderEntries);
                }
                return c.setFirstResult(page.getFirstResult())
                        .setMaxResults(page.getPageSize())
                        .setFetchSize(page.getPageSize())
                        .list();
            }
            private static Field getField(Class clazz, String fieldName) {
                try {
                    return clazz.getDeclaredField(fieldName);
                }
                catch (Exception e) {
                    throw new RuntimeException(e);
                }
            }
            private static Object getFieldValue(Field field, Object obj) {
                field.setAccessible(true);
                try {
                    return field.get(obj);
                }
```

```
            catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
        private static void setFieldValue(Field field, Object target, Object value) {
            field.setAccessible(true);
            try {
                field.set(target, value);
            }
            catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
```

然后，就可以通过一个 DetachedCriteria 实现分负查询。

```
    protected List queryForList(final DetachedCriteria dc, final Page page) {
        HibernateCallback callback = new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Criteria c = dc.getExecutableCriteria(session);
                if(page==null)
                    return c.list();
                return PaginationCriteria.query(c, page);
            }
        };
        return hibernateTemplate.executeFind(callback);
    }
```

我们定义的 DAO 对象全部是线程安全的，因此，可以在 Spring 中只定义一个实例，然后放心地在多个组件之间共享。所谓线程安全是指多个线程可以安全地执行某个对象实例的全部方法。由于方法的参数和方法内定义的局部变量的引用都存储在线程的堆栈中，因此各个线程之间互不影响。只有实例的字段是共享的，因此，只要保证实例的字段一经初始化就不再变化，这个实例就是线程安全的。

GenericHibernateDao 定义的字段只有 Class 和 HibernateTemplate，其中，Class 被定义为 final 类型，而 HibernateTemplate 一旦初始化完毕就不会更改，因此，这是一个线程安全的类，其子类只要保证各自的字段在运行期不变，也都是线程安全的。

11.4.3 调试HQL语句

由于我们使用了 Hibernate 作为持久化机制，因此，在 DAO 中，大量使用了 HQL 查询。如何调试这些 HQL 语句是应用程序开发中必须要解决的，否则，我们只能一遍

一遍地修改代码，再编译，重新运行。

调试 HQL 语句有两个方法，一是打开 Hibernate 的“hibernate.show_sql”功能，就可以在控制台看到 Hibernate 最终生成的所有 SQL 语句，不过这仍然不太直观，因此，Hibernate 还提供了 HibernateTools 插件，可以在 Eclipse 中方便地调试 HQL。

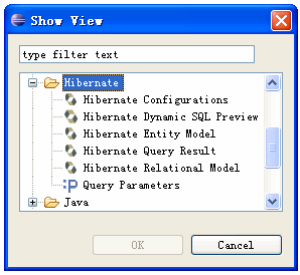


图 11-13

将 HibernateTools-3.2.0.beta7.zip 解压到 Eclipse 的安装目录，然后重新启动 Eclipse，选择菜单“Window”→“Show View”→“Other...”，在弹出的对话框中找到 Hibernate 组，展开就可以看到 HibernateTools 插件提供的视图，如图 11-13 所示。

还可以直接选择菜单“Window”→“Open Perspective”→“Other...”，打开 Hibernate Console 视角，如图 11-14 所示。

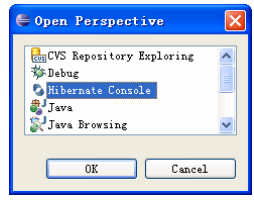


图 11-14

在“Hibernate Configurations”视图中单击⊞按钮添加一个 Hibernate 配置，在弹出的“Create Hibernate Console Configuration”对话框中填入如下内容。

Name：livebookstore；

Configuration file：\livebookstore\conf\unused\hibernate.cfg.xml；

选中“Enable hibernate ejb3/annotations”。

在 Classpath 中添加目录/livebookstore/web/WEB-INF/classes 和 jar 文件/livebookstore/lib/core/hsqldb.jar。

然后，在 Hibernate Configurations 视图中选中 livebookstore，单击右键，在弹出的快捷菜单中选择“HQL Editor”，即可输入 HQL 语句并查看运行结果，如图 11-15 所示。

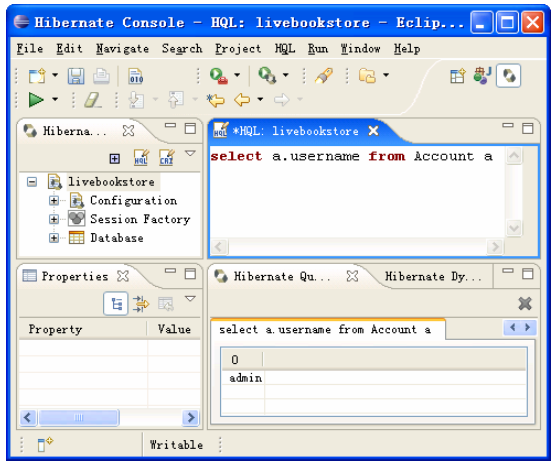


图 11-15

11.5 逻辑层设计

现在我们已经有了持久层的模型，对于持久层设计，我们始终坚持仅仅对数据库操作的封装，即任何与数据库相关的操作均封装在持久层中，任何与数据库操作无关的功能均不可放在持久层中，例如，权限控制等。持久层提供的功能就是将域对象持久化，而上层对持久层采用何种持久化机制则应一无所知。

逻辑层需要实现应用程序的逻辑，对于简单的操作，例如，注册新用户的功能在持久层中对应的是 AccountDao.createAccount()，逻辑层中仅仅简单地调用即可；还有一些操作则需要进行权限检查，例如，createComment()允许用户创建一条评论，就需要首先检查用户是否登录，然后根据用户评分更新 Book 对象，因此，对于调用后端持久层的逻辑层组件来说，可以将其看成是一个 Façade（门面模式）。

除了调用持久层实现对象的读取和存储外，Live 在线书店还需要有全文搜索功能和向用户发送电子邮件的功能，因此，我们设计了 3 个逻辑层组件，分别实现这 3 种功能，每个组件均定义一个接口和相应的实现类。

（1）BusinessService：负责封装基本的与实体对象操作相关的功能，例如，书籍浏览、用户注册、发表评论、订单确认等功能。

（2）MailService：负责向新的注册用户发送欢迎邮件，以及定期向所有注册用户发送促销信息等邮件。

（3）SearchService：负责实现全文搜索功能，使用户能根据关键字快速搜索到相关书籍。

其依赖关系如图 11-16 所示。

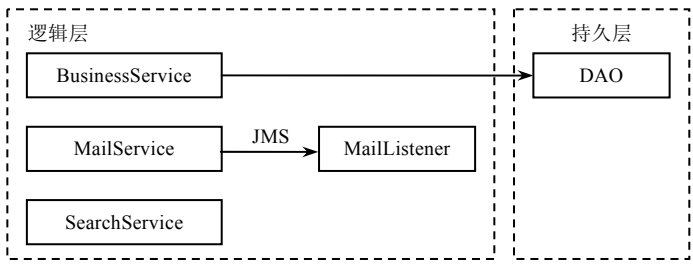


图 11-16

MailService 和 SearchService 由于其功能相对独立，与持久层关系不大，我们会在本章后半部分详细讨论。本节主要讨论 BusinessService 组件的设计和实现。

BusinessService 接口的定义如下。

```
public interface BusinessService {
    Account queryUserDetails(String username);
    Object query(Class clazz, Serializable id);
    List<Account> queryAccounts(Page page);
    List<Comment> queryComments(Book book, Page page);
    List<Order> queryOrders(Page page);
    List<Order> queryOrders(Account account, Page page);
    List<Book> queryFavorite(Account account, Page page);
    Category queryRoot();
    List<Book> queryBooks(Category c, Page page);
    List<Book> queryTopSales(Category c, int max);

    boolean createFavorite(Account account, Book book);
    void createAccount(Account account);
    void createBook(Book book);
    void createComment(Comment comment);
    void createCategory(Category category, Integer parentId);
    void createOrder(Order order, List<OrderItem> items);

    void updateAccount(Account account);
    void updateBook(Book book);
    void updateCategory(Category category);
    void updateOrder(Order order, int newState);
    void cancelOrder(Order order);

    void delete(Object object);
    void deleteFavorite(Account account, Book book);

   void changePassword(String username, String oldPassword, String newPassword);
}
```

BusinessServiceImpl 则是 BusinessService 接口的实现类，它需要调用持久层的所有 DAO 对象，因此，将 DAO 对象全部注入到 BusinessServiceImpl 中即可使用。

```
public class BusinessServiceImpl implements BusinessService {
    private AccountDao accountDao;
    private BookDao bookDao;
    private CategoryDao categoryDao;
    private CommentDao commentDao;
    private OrderDao orderDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
```

```
        }
        ...
    }
```

大多数方法都简单地对应 DAO 接口的某个方法，例如，创建一个新用户，其createAccount()方法的实现如下。

```
public void createAccount(Account account) {
    accountDao.create(account);
}
```

分页浏览书籍信息的 queryBooks()方法的实现如下。

```
public List<Book> queryBooks(Category c, Page page) {
    return bookDao.query(c, page);
}
```

我们注意到 BusinessService 提供的方法并没有针对用户实现权限检查，但事实上某些方法只有管理员才有权调用，例如，delete()方法；另一部分方法只有注册用户才有权调用，例如，changePassword()方法。因此，我们还需要对 BusinessService 组件进行保护，防止非授权的访问。

我们将在后面的安全模型一节中详细讨论如何保护 BusinessService 组件，这里我们暂时只关注 BusinessService 应当实现的逻辑功能。

11.5.1 确定事务模型

确定了持久层和逻辑层的结构后，我们还需要考虑采用何种事务模型。在第 6 章中我们已经讨论了事务的概念，以及 Spring 支持的编程式事务管理和声明式事务管理。

无论采用编程式事务管理还是声明式事务管理，首先需要确定的是事务边界，即在何处开始事务，何处结束事务。正如第 6 章所讲，如果将事务引入 DAO 对象，则 DAO 对象将到处充斥着事务代码，编写这些重复而枯燥的事务代码将很快导致代码的混乱。此外，如果一个用户的请求需要通过几步 DAO 调用完成，则将一个事务控制在 DAO 的方法内是不可行的。

一个好的方法是将事务控制在逻辑层的业务对象的方法内。用户通常只需执行业务对象的某一个方法，而在此方法中可能多次调用多个 DAO 对象的方法，因此，将事务的边界确定为业务对象的方法调用前后是正确的。

Hibernate 还支持延迟加载的技术，所谓延迟加载即读取数据库的表记录时，仅将主键或者基本字段填充到实体中，如果访问未经加载的字段，例如，外键关联的字段，则 Hibernate 在请求时才动态读取数据库并加载该字段，这样做的目的是避免加载的数据在

渲染视图时没有使用。

Hibernate 对延迟加载的实现原理是 CGLIB 动态字节码生成技术，即返回的实体并非真正的实体对象，而是经过 CGLIB 处理后的代理实体，当调用某一未经加载的属性时，代理实体就可以截获这一调用，然后由 Hibernate 实现动态加载。

如果要使用 Hibernate 的延迟加载特性，则渲染视图阶段不能关闭事务，因此，事务的范围变为整个 HTTP 请求的周期。

采用 OpenSessionInView 模式可以将事务范围界定在请求开始和渲染视图结束后，使得 Hibernate 的 Session 在视图渲染时仍有效。有两种方式实现 OpenSessionInView 模式，一种是使用 Spring 提供的 OpenSessionInViewInterceptor，如果采用 Spring MVC 框架，可以将这个 Interceptor 加入到 Controller 的拦截器链中，事务在 Controller 处理前开始，在视图渲染后结束，如图 11-17 所示。

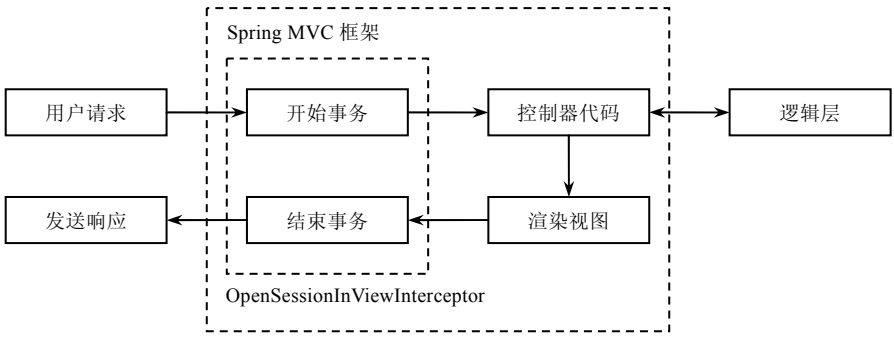


图 11-17

如果 Web 层没有采用 Spring 的 MVC 框架，而是使用 Struts 等其他 MVC 框架，甚至没有使用 MVC 框架，此时，就无法定义 Interceptor，只能采用 Filter 来实现 OpenSessionInView 模式。

OpenSessionInViewFilter 是 Spring 提供的一个 Filter。在 OpenSessionInViewFilter 模式下，所有的 HTTP 请求都将被 OpenSessionInViewFilter 截获，事务在请求处理前开始，在请求处理完毕后结束，而不管采用何种 MVC 框架，甚至直接使用 JSP，如图 11-18 所示。

错误！

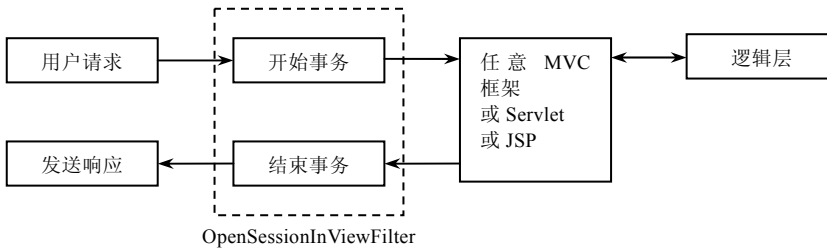


图 11-18

两种方式各有优劣。OpenSessionInViewInterceptor 只能用于 Spring MVC，但是配置简单，无须过滤 URL；OpenSessionInViewFilter 适用范围更广，但是必须手动配置 web.xml 文件，并且必须正确过滤 URL。

无论如何，采用以上两种方式的目的都是为了使用 Hibernate 的延迟加载特性。由于事务也是一种数据库资源，事务持续的时间越久，数据库资源被锁定也越久，应用程序的吞吐量就会降低。因此，要尽量将事务限定在最小的范围内。

在 Live 在线书店应用中，不考虑使用 Hibernate 的延迟加载特性，因此，事务被限定在逻辑层对象的每个方法内。

采用编程式事务管理需要编写大量额外的代码，并且需要更全面的测试，而采用声明式事务管理则无需编写代码，能够大大简化事务配置。毫无疑问，使用 Java 5 注解来配置声明式事务是最简单也是最容易维护的方案，因此，我们向 BusinessService 添加如下注解。

```
@Transactional
public interface BusinessService {

    @Transactional(readOnly=true)
    Account queryUserDetails(String username);

    @Transactional(readOnly=true)
    Object query(Class clazz, Serializable id);

    @Transactional(readOnly=true)
    List<Account> queryAccounts(Page page);

    ...
}
```

将@Transactional 标注在 BusinessService 接口的定义处，就表示该接口的所有方法均需要声明式事务。对于以 query 开头的方法，定义只读事务，可以使底层持久层根据数据库特性来优化数据库的事务性能，其他方法不必标上注解即表示采用默认的事务配置。

下一步是在 XML 配置文件中启动事务配置，声明<tx:>名字空间，然后利用<tx:annotation-driven ... />即可开启声明式事务。具体方法请参考第 6 章“使用 Java 5 注解简化配置”一节。

11.6 Web层设计

在多层应用程序中，表示层用于和用户打交道。表示层从用户获取请求，然后将请求交给逻辑层处理，最后将结果展现给用户。对于大多数 JavaEE 应用程序来说，Web 是最常用的表示层，它使得用户仅仅使用浏览器这种瘦客户端就能访问应用程序，从而

大大降低了客户端的部署成本。

对于以 Web 作为前端界面的应用程序而言，表示层的设计就是要实现一个灵活而优雅的 MVC 架构模型。Struts 是第一个也是应用最广泛的 Web MVC 框架，除此之外，还有许多其他的 MVC 框架，如 WebWork、Maverick 等，其设计与 Struts 大同小异。然而，这些 MVC 框架的局限性在于，与后端的逻辑层结合时，没有一种方便而优雅的配置方式。

Spring 框架的设计原则就是不重新发明轮子。如果现有的框架能很好地满足需求，Spring 只会提供一个集成方案。为何 Spring 还要额外提供一个 MVC 框架？因为 Spring 开发人员对 Struts 感到不是很满意，因此，Spring 提供了一个极其灵活的 MVC 框架。

和 Struts 等传统的 MVC 框架相比，Spring 的 MVC 框架从设计开始就将其纳入了 IoC 容器的管理之中。这意味着 Spring 的 MVC 组件和其他 IoC 组件一样，在 IoC 容器中部署并正确配置依赖属性，这使得 Web 层和后端的逻辑层是完全无缝集成的，因为只需要在 MVC 组件中注入逻辑层组件即可。

此外，Spring 的 MVC 框架还提供了 Interceptor 的拦截机制，这种机制类似于 Filter，不过，Interceptor 是在 IoC 容器中配置的，因此，依赖注入对 Interceptor 同样有效。借助 Interceptor，可以实现编程式事务管理、安全检查、日志记录等许多功能。

11.6.1 设计Controller体系

要应用 MVC 模式，我们要设计的主要是 Controller 和 View，Model 一般可以由普通的 Java 对象表示，在 Spring MVC 框架中，Model 总是以 Map 来表示，因此，我们不需关心 Model 的结构。

对于 Controller，我们并没有直接使用 Spring MVC 框架提供的许多现成的 Controller，而是直接从 Controller 接口派生，实现一套简单的自定义的 Controller 体系，如图 11-19 所示。

错误！

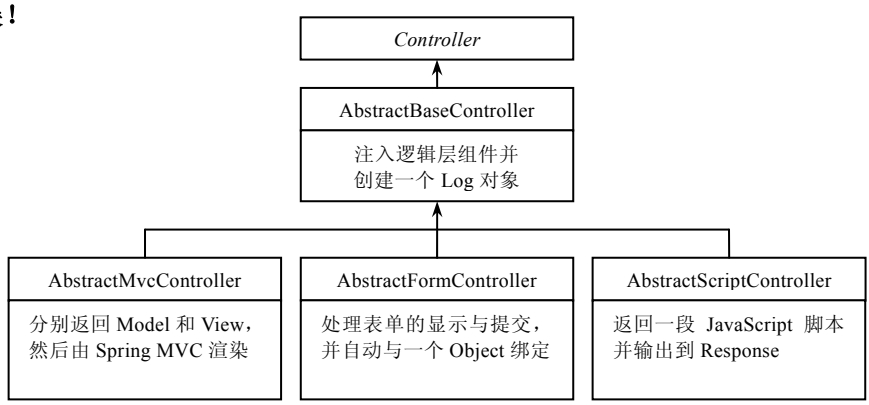


图 11-19

实现这种 Controller 体系的主要目的是将逻辑层组件直接注入到 AbstractBaseController 中，以便简化所有的 Controller 的配置。如果直接从 Spring MVC 提供的现成的 Controller 派生，就必须在每个 Controller 中添加注入逻辑层组件的重复代码。

另一个原因是我们抽象的 Controller 比较简单，不需要复杂的功能。如果需要向导页面之类复杂的功能，则最好还是从 Spring 提供的功能更完整的 Controller 体系中继承，麻烦在于要手动在子类注入逻辑层组件。

11.6.2 使用Template模式

Template 模式在超类定义了完整的模版方法的骨架，但是将部分步骤作为抽象方法的实现推迟到具体的子类中，其结构如图 11-20 所示。

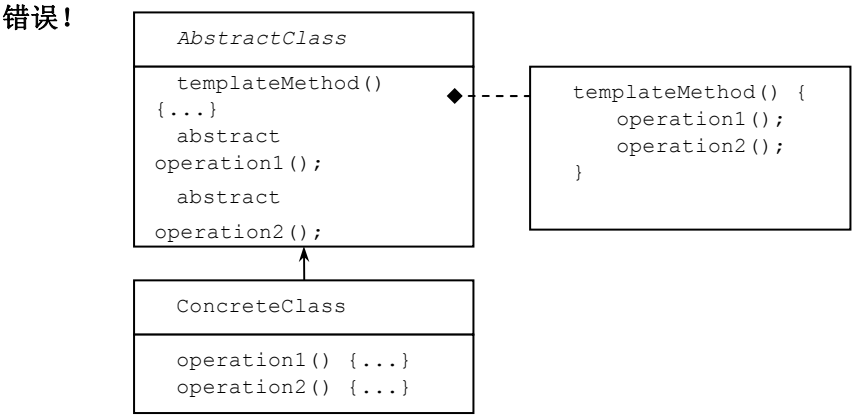


图 1-20

AbstractClass 定义了 templateMethod()方法，该方法用于实现一个完整的逻辑，但是调用了 operation1()和 operation2()这两个抽象方法。而 ConcreteClass 则实现了这两个抽象方法。

类似的，我们的 Controller 体系也采用了 Template 模式。以 AbstractMvcController 为例，如图 11-21 所示。

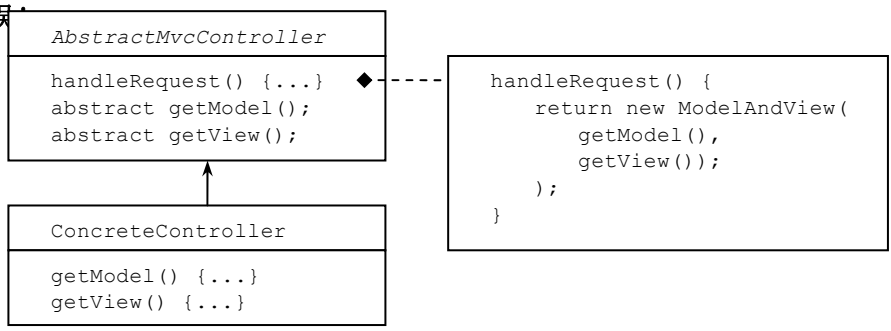


图 11-21

AbstractMvcController 已经实现了 Controller 接口必需的 handleRequest()方法，但是同时调用了 getModel()和 getView()这两个抽象方法。子类只需要实现这两个抽象方法，就可以实现一个完整的 Controller。AbstractMvcController 代码如下。

```
public abstract class AbstractMvcController extends AbstractBaseController {
    public final ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        Map map = getModel(request, response);
        if(map==null)
            return null;
        String view = getView(request, response);
        if(view==null)
            return null;
        return new ModelAndView(view, map);
    }

    public abstract Map getModel(HttpServletRequest request, HttpServlet
Response response) throws Exception;

    public abstract String getView(HttpServletRequest request, HttpServlet
Response response);
}
```

而一个具体的派生自 AbstractMvcController 的子类，例如，显示书籍详细信息的 Controller 可能如下。

```
public class BookDetailController extends AbstractMvcController {
    public Map getModel(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        // 查询 Book:
        Book book = ...
        // 构造 Model:
        Map map = new HashMap();
        map.put("book", book);
        // 其他相关 object:
        map.put(...);
        return map;
    }

    public String getView(HttpServletRequest request, HttpServletResponse
response) {
        return "/bookDetail.htm";
    }
}
```

另一个非常有用的 AbstractScriptController 同样采用了 Template 模式，凡是从 AbstractScriptController 派生的子类都有能力向客户端输出一段 JavaScript 脚本，而设置 ContentType、缓存时间等功能则已经在 AbstractScriptController 中实现了。

```
        public abstract class AbstractScriptController extends AbstractBaseController {
            public final ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
                String script = getScript(request, response);
                response.setContentType("text/javascript;charset=UTF-8");
                int cache = getCacheTime(request, response);
                if(cache<=0)
                    response.setHeader("Cache-Control", "no-cache");
                else {
                    response.setHeader("Cache-Control", "Private");
                    response.addDateHeader("Expires", System.currentTimeMillis() +
1000 * cache);
                }
                PrintWriter writer = response.getWriter();
                writer.write(script);
                writer.flush();
                return null;
            }

            // 获得具体的脚本内容,例如"document.write('hello')":
            public abstract String getScript(HttpServletRequest request, HttpServlet
Response response) throws Exception;

            // 获得缓存时间,以秒为单位:
            public abstract int getCacheTime(HttpServletRequest request, HttpServlet
Response response);
        }
```

子类则只需要关注如何生成 JavaScript 本身和简单地返回一个缓存时间即可。如果不采用 Template 模式，则每个子类都需要重复编写对 HttpServletResponse 对象的烦琐操作。

Template 模式将不同子类中相同的过程或一系列步骤上移到超类中，但是其中个别步骤在更详细的层次上实现有所不同，因此，个别步骤的实现就被推迟到子类中。

我们将这些基本的模版类放入 net.livebookstore.web.core 中，然后将具体的 Controller 根据访问权限归类，能够被匿名访问的 Controller（如 SearchBooksController、RegisterController 等）被放入 net.livebookstore.web 包中；只有登录后的用户才能访问的 Controller（如 ListOrdersController、PasswordController 等）被放入 net.livebookstore.web.user 包中；只有管理员身份才能访问的 Controller（如 AdminManageOrderController、

AdminRebuildIndexController 等）被放入 net.livebookstore.web.admin 包中，这样便于管理。

对于每个页面，都必须有一个单独的 Controller，我们定义了以下的 Controller，如表 11-2~表 11-4 所示。

表 11-2 允许匿名访问的 Controller

名 称	派 生 自	描 述
ListBooksController	AbstractMvcController	列出指定分类下的书籍
BookDetailController	AbstractMvcController	显示书籍详细信息
ListCommentsController	AbstractMvcController	列出书籍全部评论
CartController	AbstractMvcController	查看购物车
RegisterController	AbstractFormController	注册新用户
TopSalesController	AbstractMvcController	列出书籍销量排行榜
SearchBooksController	AbstractMvcController	根据关键字搜索书籍
CheckStateController	AbstractScriptController	检查当前用户状态

表 11-3 仅允许注册用户访问的 Controller

名 称	派 生 自	描 述
ListOrdersController	AbstractMvcController	列出用户的所有订单
OrderDetailController	AbstractMvcController	列出某一订单的详细信息
PreviewOrderController	AbstractMvcController	预览新订单
CancelOrderController	AbstractRedirectController	取消一个订单
FavoriteController	AbstractMvcController	列出收藏夹的书籍
PasswordController	AbstractMvcController	修改口令
ProfileController	AbstractMvcController	修改个人资料
MakeCommentController	AbstractRedirectControllers	对书籍发表评论

表 11-4 仅允许管理员访问的 Controller

名 称	派 生 自	描 述
AdminListOrdersController	AbstractMvcController	列出所有用户的 Order
AdminManageOrderController	AbstractRedirectController	管理订单
AdminBroadcastMailController	AbstractMvcController	群发邮件
AdminAccountController	AbstractMvcController	管理用户
AdminBookController	AbstractMvcController	管理书籍
AdminCategoryController	AbstractMvcController	管理书籍分类
AdminOutputController	AbstractMvcController	导出书籍分类
AdminRebuildIndexController	AbstractMvcController	重建所有书籍的索引

注意到没有处理登录和注销的 Controller，因为我们将采用 Acegi 安全框架实现登录机制，因此无需在 Web 层处理。

11.6.3 配置Controller

正如第 7 章介绍的，使用 Spring MVC 框架时，BeanNameUrlHandlerMapping 应当首先考虑使用。利用 XDoclet，我们只需要在每个 Controller 定义处加上 XDoclet 的注释，即可自动将所有的 Controller 添加到 XML 配置文件中，例如：

```
/**
 * @spring.bean name="/bookDetail.jspx"
 */
public class BookDetailController extends ... {
    ...
}
```

逻辑层组件的注入已经在超类 AbstractBaseController 中实现了，XDoclet 会自动提取超类的注释并合并到子类中。

整个 Web 层的 XML 配置文件为 dispatcher-servlet.xml，它同时导入了自动生成的配置文件 dispatcher-servlet-import-beans.xml 和稍后在 Web 服务中需要用到的 XFire 的配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
>
    <import resource="dispatcher-servlet-import-beans.xml" />
    <import resource="classpath:org/codehaus/xfire/spring/xfire.xml" />

    <bean id="beanNameUrlMapping"
     class="org.springframework.web.servlet.handler.BeanNameUrlHandler Mapping">
        <property name="alwaysUseFullPath" value="true" />
    </bean>

    <bean id="multipartResolver"
     class="org.springframework.web.multipart.commons.CommonsMultipart Resolver">
        <!-- 最大允许上传文件：1MB -->
        <property name="maxUploadSize" value="1048576" />
    </bean>
</beans>
```

11.6.4 设计View

在解决了 MVC 的 M（Model）和 C（Controller）外，View（视图）也是一个非常重要的组件。通常，JavaEE 应用程序会选择 JSP 作为 View，这不仅是因为 JSP 的广泛应用，还由于 JSP 是 JavaEE 标准的一个重要部分。然而，考察许多现有的 Web 应用程序就会发现，JSP 作为视图技术，通常应当由网页设计人员或者美工来开发，而 JavaEE 开发人员少有艺术天赋，难以设计出赏心悦目的页面。对于网页设计人员来讲，快速方便的可视化网页设计工具（如 Dreamwaver）是他们的首选，然而，对<% ... %>的 Java 代码和大量的自定义标签，则可能完全打乱了精心设计的页面。

在第 7 章中，除了 JSP 外，我们还介绍了其他几种常用的视图技术，包括 Velocity、FreeMarker、Xslt。Xslt 由于语法非常复杂，调试困难，我们不予考虑。Velocity 和 FreeMarker 则比较类似。综合考察各种视图技术后，Live 在线书店最终采用 Velocity 这种模版技术作为视图。采用 Velocity 而非 JSP 作为视图有如下好处。

（1）Velocity 采用简单而强大的模版语言 VTL 来渲染页面，能保证页面在 Dreamwaver 之类的可视化编辑器中正常显示。

（2）Velocity 模版可以是任意扩展名，在 Live 在线书店中，我们用.html 作为扩展名，设计页面时更为方便，因为可以随时在浏览器中预览页面效果，而使用 JSP 时，除非启动服务器，否则无法在浏览器中预览。

（3）Velocity 禁止在页面中写入任何逻辑代码，因此，强迫保证 MVC 划分清晰，而 JSP 由于允许嵌入 Java 代码和自定义标签，很容易引起逻辑混乱，不利于页面维护。

（4）Velocity 具有很高的速度，其渲染页面的速度甚至比 JSP 还要快。

采用 Velocity 也有如下缺点。

（1）Velocity 不是 JavaEE 标准，文档较少，通常只能从社区获得支持。

（2）Velocity 的 VTL 语法需要一定的学习时间，尽管相对比较容易。

下面我们简单地介绍一下 Velocity 模版的编写。Velocity 模版非常简单，它是一个纯文本文件，以“#”开头的被视为指令，以“$”开头的被视为变量，其他一律不变，直接输出到结果。一个最简单的 Velocity 模版如下。

```
<html>
<body>
#set($day="Sunday")
Today is $day.
</body>
</html>
```

以上模版的输出即为一个完整的 HTML。

1．引用变量

Velocity 使用“${表达式}”来引用一个变量，并可调用其属性或方法。在没有歧义的情况下，也可以简写为“$表达式”，例如：

```
$foo
$!foo
$foo.name
$foo.max()
```

$foo 表示引用 foo 变量，foo 可以是任意类型，包括基本类型，Velocity 将会正确地将 foo 显示出来。如果 foo 为 null，则显示的结果是$foo 而非 null，这一点很重要。在 HTML 表单显示时，我们更希望将 null 变量显示为空字符串，这时可以使用$!foo，若 foo 为 null，则什么也不显示。

```
<input type="text" name="foo" value="$!foo" />
```

$foo.name 有两种解释，既可以表示 foo 对象的一个 name 字段，又可以表示 foo 对象的 name 属性，即 getName()方法调用。结果是 foo 对象的 name 字段值或 getName()方法的返回值。

$foo.max()则明确地表示返回 foo 对象的 max()方法调用的结果。

在不引起歧义的情况下，省略“{}”是可以的，不过有些时候，必须使用“{}”来明确表示一个引用，例如：

```
get ${foo}tball
```

若去掉“{}”，Velocity 将试图解释$football 变量。

2．赋值语句

Velocity 允许通过#set 语句对一个变量赋值，例如：

```
#set($name="Spring 2.0")
#set($name="Spring 2.0 $author")
#set($valid=true)
#set($number=100)
#set($price=18.90)
#set($name=$foo.name)
```

可以直接将字符串或基本类型直接赋给变量，Velocity 会自动判断类型；也可以将一个变量赋给另一个变量。String 类型的变量可以直接嵌在一起。对于基本类型的变量，还可以做简单的运算如下例。

```
#set($value=$value + 1)
#set($value=$count * 100)
#set($value=$foo.max / $count)
```

3．条件语句

Velocity 使用#if()作为条件判断语句，可以使用#if、#elseif、#else 和#end 构成任意复杂的条件判断语句，如下例。

```
#if($foo)
   <strong>$foo.value</strong>
#elseif($bar.value>0)
   <strong>$bar.value</strong>
#else
   <strong>Nothing</strong>
#end
```

上例中，对于条件表达式$foo，若$foo 是一个 boolean 或 Boolean 类型的变量，则如果其值为 true，判断成功；若$foo 是一个其他类型的对象，则$foo 只要不为 null 就判断成功。对于数值比较，Velocity 会自动判断$bar.value 的类型。

4．循环语句

Velocity 通过#foreach 实现循环，这和 Java 5 的 for(var : collection)非常类似。例如：

```
#foreach($b in $books)
   <p>Book name: $b.name, price: $b.price</p>
#end
```

其中，$books 必须是数组类型或集合类型的对象，foreach 循环将依次取出$books 中的每一个元素并赋给$b 变量，在循环体中即可引用$b 变量。

5．宏

Velocity 提供了功能强大的#macro 指令来实现宏的功能，宏类似于函数，可以在模版中反复使用，还可以将多个宏集合到一个单独的文件中，便于管理和维护。例如：

```
#macro(drawTable $color $list)
   <table>
      #foreach($obj in $list)
         <tr><td bgcolor="$color">$obj</td></tr>
      #end
   </table>
#end
```

以上定义了一个名为“drawTable”的宏，包含两个变量$color 和$list，然后根据$color 设置背景色，并循环$list 依次显示在表格中。调用该宏的方法如下。

```
#set($list=["J2SE", "JavaEE", "JavaME"])
#set($color="red")
#drawTable($color $list)
```

利用宏可以极大的提高复杂页面的复用性。将某些部分抽象为宏就可以在各个页面中复用它们。推荐将所有的宏都集中到一个独立的文件中，以便于管理和维护。

6．注释

Velocity 提供两种注释类型，单行注释由##开头，到行尾自动结束，类似于 Java 的“//”注释。多行注释由#*开头，以*#结束，类似于 Java 的“/* ... */”注释。例如：

```
## 这是一个单行注释
#*
    这是一个多行注释
    更多注释...
*#
```

在视图渲染阶段，注释将被全部忽略，并且不会输出到浏览器端，因此，可以在任何地方使用注释。

11.6.5 简化分页逻辑

在持久层的设计中，我们已经使用了 Page 对象来帮助实现分页查询。现在，需要在表示层中方便地显示页数和上一页、下一页给用户。如果每个页面都重复编写分页代码，将造成逻辑维护复杂，并且无法做到统一更新。此时，利用 Velocity 提供的宏功能，就能大大简化分页的显示。

对于有分页显示功能的页面，Controller 总会将 Page 对象以“page”为名称放入 Model 中，因此，在 Velocity 模版中，我们首先定义一个 pagination 宏，它仅仅依赖 Model 中的这个 Page 对象就可以完成分页功能。

```
#macro(pagination)
    #if( ${page.totalCount} == 0 )
        <table border="0" align="center" cellpadding="3" cellspacing="3">
            <tr><td align="center"><font color="#ff0000"><strong>没有可显示的项
目</strong></font></td></tr>
        </table>
    #else
        #set( $p_count = $page.pageCount )
        #set( $p_index = $page.pageIndex )
        <table border="0" align="center" cellpadding="3" cellspacing="3">
            <tr>
                <td class="pageNonClickable">共<b>${page.totalCount}</b>项，
```

```
<b>${p_count}</b>页：</td>
                        ## 是否显示"上一页":
                        #if(${page.hasPrevious})
                            #set($p_tmp = $p_index - 1)
                            <td class="pageClickable" onMouseOver="hoverPage(this)"
onMouseOut="normalPage(this)" onClick="location.assign(getPageLocation(${p_tmp}))">
上一页</td>
                        #else
                            <td class="pageUnclickable">上一页</td>
                        #end
                        ## 如果前面页数过多,显示"...":
                        #if($p_index>5)
                            #set( $p_prevPages = $p_index - 5)
                            #set( $p_start = $p_index - 4)
                            <td class="pageClickable" onMouseOver="hoverPage(this)"
onMouseOut="normalPage(this)" onClick="location.assign(getPageLocation(${p_ prevPages}))
">…</td>
                        #else
                            #set($p_start = 1)
                        #end
                        ## 显示当前页附近的页
                        #set( $p_end = $p_index + 4 )
                        #if( $p_end > $p_count )
                            #set( $p_end = $p_count )
                        #end
                        #foreach($i in [$p_start..$p_end])
                            #if($i==$p_index)
                                <td class="pageNonClickable"><b><font color="#FF0084"
>${i}</font></b></td>
                            #else
                                <td class="pageClickable" onMouseOver="hoverPage(this)"
onMouseOut="normalPage(this)" onClick="location.assign(getPageLocation(${i}))">
${i}</td>
                            #end
                        #end
                        ## 如果后面页数过多,显示"...":
                        #if( $p_end < $p_count )
                            #set( $p_nextPages=$p_end + 1 )
                            <td class="pageClickable" onMouseOver="hoverPage(this)"
onMouseOut="normalPage(this)" onClick="location.assign(getPageLocation(${p_next
Pages}))">…</td>
                        #end
                        ## 显示"下一页":
                        #if(${page.hasNext})
                            #set($p_tmp=${p_index}+1)
                            <td class="pageClickable" onMouseOver="hoverPage(this)"
onMouseOut="normalPage(this)" onClick="location.assign(getPageLocation(${p_tmp}))">
```

```
下一页</td>
                    #else
                        <td class="pageUnclickable">下一页</td>
                    #end
                </tr>
            </table>
        #end
    #end
```

虽然#pagination()宏的定义比较复杂，但是，在各个页面中引用起来却极为简单，只需要在希望出现分页显示的地方写一句#pagination()即可，不需要任何参数。

```
<table width="600">
    <tr>
        <td>显示第 N 页的内容...</td>
    </tr>
    <tr>
        <td>#pagination()</td>
    </tr>
</table>
```

页面的显示效果如图 11-22 所示。

图 11-22

读者可能会问，#pagination()宏如何知道当前页面的 URL 并正确构造出任意页的 URL？这里用的小技巧是使用 JavaScript 获取当前页面的 URL，然后分析 URL 参数，重新构造出任意页的 URL。只要保证当前页面对应的 Controller 正确地放入了名称为“page”的 Page 对象，#pagination()宏就可以正常工作。

如果对分页显示的效果不满意，修改#pagination()宏即可立刻将更新反映到所有具有分页逻辑的页面上，大大降低了页面的维护成本。

11.6.6 配置Velocity

在第 7 章中，我们已经介绍了 Velocity 在 Spring 中的基本配置方法，这里仅给出 dispatcher-servlet.xml 配置文件中的相关片段。

```
        <bean id="viewResolver" class="org.springframework.web.servlet.view.
velocity.VelocityViewResolver">
            <!-- 保证输出为 UTF 编码 -->
            <property name="contentType"><value>text/html;charset=UTF-8</value>
```

```
</property>
        </bean>

        <bean id="velocityConfig" class="org.springframework.web.servlet.view.
velocity.VelocityConfigurer">
            <property name="configLocation" value="/WEB-INF/velocity.properties" />
            <!-- 以 Web 根目录为模版目录 -->
            <property name="resourceLoaderPath" value="/" />
        </bean>
```

另一个 velocity.properties 配置文件必须放在指定的/WEB-INF/目录下，其内容与第 7 章介绍的一致，这里不再列出具体内容，唯一要注意的是我们将所有的宏定义全部放在了/vm/macro.txt 中，因此，必须要有以下配置。

```
velocimacro.library = /vm/macro.txt
velocimacro.library.autoreload = true
```

设置 velocimacro.library.autoreload 为 true 是为了方便调试，以便让 Velocity 随时检查文件更改，这样我们不必重启服务器就可以修改宏定义。在产品部署时可以将其改为 false，以便提高一点性能。

11.6.7 配置MVC

最后一步是在 web.xml 中配置整个 MVC。我们使用.jspx 作为映射后缀。如果不希望用户得知服务器采用何种技术和平台，也完全可以使用.html 作为后缀。

在 web.xml 中配置 DispatcherServlet 的映射如下。

```
<servlet>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>

<!-- 映射.jspx 后缀 -->
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

11.7 设计安全模型

Live 在线书店是一个三层 Web 应用程序，我们需要对 Web 资源和业务逻辑组件进

行保护，避免非授权的访问。由于 Live 在线书店整个应用都是建立在 Spring 2.0 框架之上的，因此，采用 Acegi 基于角色的声明式安全控制方式最适合不过了。

根据 Live 在线书店的业务需求，所有的用户可以分为普通的注册用户和管理员两类，因此，我们定义以下两种用户角色。

（1）ROLE_ADMIN：代表系统管理员用户。

（2）ROLE_USER：代表普通的注册用户。

在第 10 章中，我们已经介绍了如何使用 Acegi 实现声明式的安全控制模型，下面我们详细介绍 Live 在线书店应用采用 Acegi 如何保护 Web 资源和逻辑层组件。

11.7.1 保护Web资源

首先我们需要保护的是 Web 资源，这是由 Acegi 的 Filter 实现的。此外，登录和注销也交给 Acegi 实现，而不需要我们自己手动编写用户登录的逻辑。此外，自动登录也被纳入 Live 在线书店提供的功能中，因此，在 Acegi 配置文件中，一共需要定义以下几个 Filter，如图 11-23 所示。

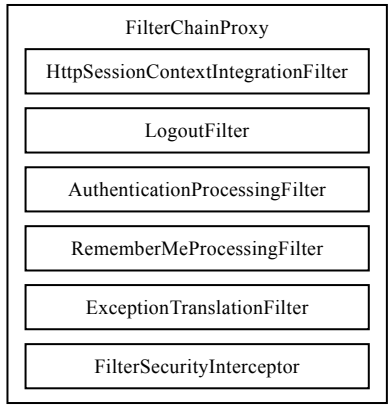


图 11-23

Filter 的具体配置请参考第 10 章。此外，认证管理器、决策管理器，以及相关的 Bean 配置和第 10 章介绍的也基本一致，这里不再重述，仅给出 XML 配置。

```
<!-- 认证管理器 -->
<bean id="authenticationManager"
   class="org.acegisecurity.providers.ProviderManager">
   <property name="providers">
      <list>
         <ref bean="daoAuthenticationProvider" />
         <ref bean="rememberMeAuthenticationProvider" />
```

```
        </list>
    </property>
</bean>

<!-- 基于 DAO 验证的 AuthenticationProvider -->
<bean id="daoAuthenticationProvider"
    class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
    <property name="userDetailsService" ref="userDetailsService" />
</bean>

<bean id="userDetailsService"
    class="net.livebookstore.security.JdbcUserDetailsService">
    <property name="dataSource" ref="dataSource" />
</bean>

<!-- 基于 RememberMe 验证的 AuthenticationProvider -->
<bean id="rememberMeAuthenticationProvider"
       class="org.acegisecurity.providers.rememberme.
RememberMeAuthentica tionProvider">
    <property name="key" value="RememberMeAtLiveBookstore" />
</bean>

<bean id="rememberMeServices"
    class="org.acegisecurity.ui.rememberme.TokenBasedRememberMeServices">
    <property name="userDetailsService" ref="userDetailsService" />
    <property name="parameter" value="j_remember_me" />
    <property name="key" value="RememberMeAtLiveBookstore" />
</bean>

<!-- 决策管理器 -->
<bean id="accessDecisionManager"
    class="org.acegisecurity.vote.AffirmativeBased">
    <property name="decisionVoters">
        <list>
            <bean class="org.acegisecurity.vote.RoleVoter" />
        </list>
    </property>
    <property name="allowIfAllAbstainDecisions" value="false" />
</bean>
```

在 Account 实体中，我们用 int privilege 表示用户权限，可以很容易地将其映射到 Acegi 的角色上。这个映射不是由 Account 类完成的，而是我们自定义的 UserDetails Service 实现的，因此，数据库表的结构和 Acegi 的角色模型是完全分离的。

UserDetailsService 也是唯一一个集成 Acegi 框架时需要我们手动编写的 Bean 组件。

因此，最后一步便是为 Acegi 提供一个 UserDetailsService 的实现。以 JDBC 的方式直接访问数据库实现一个 JdbcUserDetailsService 是最直接的方式，只需要注入一个 DataSource 即可。

```
        public class JdbcUserDetailsService implements UserDetailsService {
            private final Log log = LogFactory.getLog(getClass());

            private final GrantedAuthority ROLE_ADMIN = new GrantedAuthorityImpl
("ROLE_ADMIN");
            private final GrantedAuthority ROLE_USER = new GrantedAuthorityImpl
("ROLE_USER");

            private final GrantedAuthority[] AUTHORITY_ADMIN = new GrantedAuthority[]
{ROLE_ADMIN, ROLE_USER};
            private final GrantedAuthority[] AUTHORITY_USER = new GrantedAuthority[]
{ROLE_USER};

            private DataSource dataSource;

            public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
            }

            public UserDetails loadUserByUsername(String username)
                    throws UsernameNotFoundException, DataAccessException {
                if(log.isDebugEnabled())
                    log.debug("Loading UserDetails of username: " + username);
                Connection conn = null;
                PreparedStatement ps = null;
                ResultSet rs = null;
                try {
                    conn = dataSource.getConnection();
                    boolean autoCommit = conn.getAutoCommit();
                    conn.setAutoCommit(true);
                    ps = conn.prepareStatement("select password as pwd, privilege from
t_account where username=?");
                    ps.setString(1, username);
                    rs = ps.executeQuery();
                    if(!rs.next()) {
                        log.warn("UserDetails load failed: No such user with username
- " + username);
                        throw new UsernameNotFoundException("User name is not found.");
                    }
                    String password = rs.getString(1);
```

```
                int privilege = rs.getInt(2);
                if(!conn.getAutoCommit())
                    conn.commit();
                log.info("Username " + username + " loaded successfully.");
                conn.setAutoCommit(autoCommit);
                return new User(username, password, true, true, true, true,
privilege==Account.PRIVILEGE_ADMIN
                        ? AUTHORITY_ADMIN : AUTHORITY_USER);
            }
            catch(SQLException sqle) {
                log.error("Error retrieving UserDetails from database.", sqle);
             throw new DataRetrievalFailureException("Data retrieval failed.", sqle);
            }
            finally {
                if(rs!=null) {
                    try {
                        rs.close();
                    }
                    catch(SQLException e) {
                        log.warn("Close ResultSet exception.", e);
                    }
                }
                if(ps!=null) {
                    try {
                        ps.close();
                    }
                    catch(SQLException e) {
                        log.warn("Close PreparedStatement exception.", e);
                    }
                }
                if(conn!=null) {
                    try {
                        conn.close();
                    }
                    catch(SQLException e) {
                        log.warn("Close Connection exception.", e);
                    }
                }
            }
        }
    }
```

对应的 XML 配置片断如下。

```
    <bean id="userDetailsService"
```

```
            class="net.livebookstore.security.JdbcUserDetailsService">
            <property name="dataSource" ref="dataSource" />
        </bean>
```

使用 JDBC 实现 UserDetailsService 时，代码比较烦琐，且需要知道底层数据库表的结构，好处是只需要依赖一个 DataSource，对其他任何组件都没有依赖关系，适用范围较广。

另一种方式是直接调用逻辑层组件 BusinessService 的 queryUserDetails()方法，这种方式不需要使用复杂的 JDBC，代码更简单。我们通过这种方式来实现一个 SimpleUserDetailsService。

```
        public class SimpleUserDetailsService implements UserDetailsService {
            private final Log log = LogFactory.getLog(getClass());

            private final GrantedAuthority ROLE_ADMIN = new GrantedAuthorityImpl
("ROLE_ADMIN");
            private final GrantedAuthority ROLE_USER = new GrantedAuthorityImpl
("ROLE_USER");

            private final GrantedAuthority[] AUTHORITY_ADMIN = new GrantedAuthority[]
{ROLE_ADMIN, ROLE_USER};
            private final GrantedAuthority[] AUTHORITY_USER = new GrantedAuthority[]
{ROLE_USER};

            private BusinessService service;
            public void setBusinessService(BusinessService service) {
                this.service = service;
            }

            public UserDetails loadUserByUsername(String username)
                    throws UsernameNotFoundException, DataAccessException {
                if(log.isDebugEnabled())
                    log.debug("Loading UserDetails of username: " + username);
                Account account = service.queryUserDetails(username);
                return new User(account.getUsername(), account.getPassword(),
                        true, true, true, true,
                        account.getPrivilege()==Account.PRIVILEGE_ADMIN ? AUTHORITY_
ADMIN : AUTHORITY_USER);
            }
        }
```

和 JdbcUserDetailsService 相比，通过调用 BusinessService 组件获得用户信息的 SimpleUserDetailsService 更加简单，但是需要特别注意事务的配置。由于 Acegi 在读取

用户信息时，代码执行点还在 Acegi 的 Filter 内，此时就要保证事务的正确启动和关闭。当采用 OpenSessionInView 模式并使用 Filter 来控制事务时，就可能出现 Acegi 调用 BusinessService 组件时不在事务的控制范围之内，如图 11-24 所示。

错误！

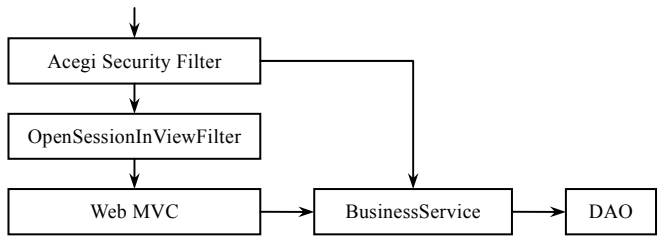


图 11-24

当采用 JDO 等作为持久层机制时，如果没有在事务控制之内执行数据库操作，即使是查询也不会得到任何结果，因此，这种方式需要合适的事务控制模型。

对于 Live 在线书店应用来说，由于我们是在 BusinessService 组件上定义的声明式事务，因此，可以安全地在 Acegi 中调用 BusinessService。如果采用 OpenSessionInView 模式，就需要考虑事务是否正确地启动和关闭。

在 web.xml 中配置好 Acegi 的 FilterToBeanProxy，然后配置映射为*.jspx，Web 层的安全配置即完毕。具体配置方式请参考第 10 章。下一步是编写登录页面表单，注意表单的 URL，以及文本框的 name 要与 Acegi 配置中的一致。这里，我们的配置如下。

（1）用户登录表单的提交 URL：/login.jspx。

（2）用户名文本框的名称：j_username。

（3）口令文本框的名称：j_password。

（4）记住登录的复选框的名称：j_remember_me。

（5）用户注销的 URL：/logout.jspx。

这样便可以顺利实现用户登录与注销的功能。注意：登录和注销的 URL 都由 Acegi 处理，并没有真正的 Controller 与之对应。

在 Live 在线书店应用中，为了保证用户口令的安全，我们在数据库中存储的并非用户的原始口令，而是“用户名+固定字符串+原始口令”的 MD5 码，因此，在登录页面中，就需要使用 JavaScript 自动将用户输入的原始口令转化成上述的 MD5 码，然后交给 Acegi，让其与数据库中存储的口令判断，从而进一步保证了系统安全。

11.7.2 保护BusinessService组件

除了对 Web 资源进行保护外，我们还需要保护逻辑层组件，即 BusinessService 组件。

在第 10 章中，我们已经详细介绍了如何使用 Acegi 提供的 Java 5 注解实现声明式的 AOP 保护，这里，针对两种不同权限的用户，我们在 BusinessService 接口定义处加上 Acegi 的@Secured 注解。

```
public interface BusinessService {
    @Secured({"ROLE_ADMIN"})
    List<Account> queryAccounts(Page page);

    @Secured({"ROLE_USER"})
    List<Order> queryOrders(Account account, Page page);

    @Secured({"ROLE_USER"})
    List<Book> queryFavorite(Account account, Page page);

    @Secured({"ROLE_USER"})
    boolean createFavorite(Account account, Book book);

    @Secured({"ROLE_ADMIN"})
    void createBook(Book book);
    ...
}
```

然后，通过定义一个 MethodSecurityInterceptor 和 Spring 提供的 BeanNameAuto ProxyCreator 即可完成 BusinessService 组件的安全配置。

```
<bean id="serviceSecurityInterceptor"
        class="org.acegisecurity.intercept.method.aopalliance.
MethodSecurityInterceptor">
    <property name="validateConfigAttributes" value="true" />
    <property name="authenticationManager" ref="authenticationManager" />
    <property name="accessDecisionManager" ref="accessDecisionManager" />
    <property name="objectDefinitionSource">
     <bean class="org.acegisecurity.intercept.method.MethodDefinition Attributes">
          <property name="attributes">
             <bean class="org.acegisecurity.annotation.SecurityAnnotation
Attributes" />
          </property>
       </bean>
    </property>
</bean>

<!-- 利用 Spring 的自动代理功能实现 AOP 代理 -->
<bean id="autoProxyCreator"
    class="org.springframework.aop.framework.autoproxy.BeanNameAuto
ProxyCreator">
```

```
    <property name="interceptorNames">
        <list>
            <value>serviceSecurityInterceptor</value>
        </list>
    </property>
    <property name="beanNames">
        <list>
            <value>businessService</value>
        </list>
    </property>
</bean>
```

下面我们要讨论的另一个安全问题是，某些时候，仅根据角色有时并不能完全确定用户权限。例如，在根据订单号查询 Order 对象时，并非具有 ROLE_USER 角色的用户就能成功查询到 Order 对象，还要根据 Order 对象关联的 Account 是否是当前用户来决定该查询是否能够成功，否则，用户 A 如果知道用户 B 的某一订单号，就可以查询到用户 B 的订单，这是不允许的。

解决该问题的最简单的方法是在 BusinessService 的实现类中加入相应的安全检查，进一步验证具体的用户身份。为了最大限度地简化 BusinessService 实现类中的安全检查代码，我们首先定义了一个 SecurityUtil 辅助类，主要提供 assertUsername(String username) 方法来检查当前用户是否是指定的用户身份。

```
public class SecurityUtil {
    public static void assertUsername(String username) {
      Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        _assertUsername(auth, username);
    }

    private static void _assertUsername(Authentication auth, String username) {
        String s = _getUsernameFromAuth(auth);
        if(s==null || !s.equals(username))
            throw new AccessDeniedException("Access denied.");
    }

    private static String _getUsernameFromAuth(Authentication auth) {
        if(auth==null)
            return null;
        if (auth.getPrincipal() instanceof UserDetails) {
            return ((UserDetails) auth.getPrincipal()).getUsername();
        }
        return auth.getPrincipal().toString();
    }
    ...
}
```

现在，在 BusinessServiceImpl 中，对于重要的方法，如果需要检查用户身份，就可以随时调用 SecurityUtil.assertUsername()。如果指定的用户身份与当前用户身份不符，将抛出一个 AccessDeniedException。例如，对于取消订单的方法，就需要首先检查当前订单的所有者和当前用户的身份是否一致。

```
public void cancelOrder(Order order) {
    SecurityUtil.assertUsername(order.getAccount().getUsername());
    if(!order.canCancel())
        throw new IllegalArgumentException("订单已在处理中，无法取消");
    order.setState(Order.STATE_CANCELLED);
    orderDao.update(order);
}
```

虽然在 BusinessService 组件中引入了安全逻辑代码，导致部分安全检查代码和应用程序的逻辑混在一起，但是，这是一种最简单且最直接的解决方式。另一种方式是针对具体的实体对象实现 ACL 授权操作，不过实现起来不仅麻烦，而且效率也更低。

此外，将 MailService 和 SearchService 组件保护起来也未尝不可。同样使用 Acegi 注解的方式对这两个组件进行保护，然后将其添加到 BeanNameAutoProxyCreator 的 beanNames 属性列表中即可。

11.7.3 阻止访问Velocity模版

由于模版文件是以.htm 的扩展名存储的，因此，用户如果知道模版的路径，就可以直接在浏览器地址栏输入地址来查看模版的原始内容，如图 11-25 所示。

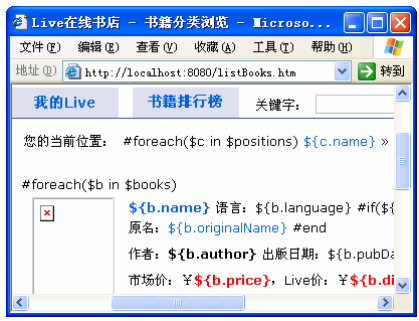


图 11-25

虽然我们设计的 Velocity 模版仅仅用于渲染 Model 的视图，没有任何业务逻辑在其中，因此，暴露模版文件并不会对系统安全构成威胁。不过，我们仍然希望能阻止用户直接查看模版文件。将所有的模版文件放入/WEB-INF/目录下即可做到这一点，但是，这样会影响页面的可视化设计，因为打乱了图像、CSS、JavaScript 文件的相对路径。另

一种方式是配置一个过滤器，阻止所有后缀为.htm 的 URL 访问。

在第 7 章中，我们已经介绍了如何设计一个 SecurityFilter 阻止用户访问某些受限资源，这里正好派上用场。我们将 SecurityFilter 简单地改造为 ForbiddenFilter，然后在 web.xml 中配置这个 Filter，过滤掉所有以*.htm 结尾的 URL。

```
<filter>
    <filter-name>forbiddenFilter</filter-name>
    <filter-class>
        net.livebookstore.web.filter.ForbiddenFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>forbiddenFilter</filter-name>
    <url-pattern>*.htm</url-pattern>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

重新编译，运行，输入模版文件的地址，将得到一个 403 禁止访问的错误，如图 11-26 所示。

图 11-26

对于真正的静态 HTML 文件，例如，登录页面，用户能否直接访问呢？这里我们采用了一个很小的技巧，即对 Velocity 模版采用.htm 作为扩展名，而真正的静态 HTML 页面采用.html 作为扩展名，因此，ForbiddenFilter 仅过滤.htm 页面，不会影响用户访问真正的静态 HTML 文件。从这一点也可以看出，在设计系统时一定要保持命名的一致性，这样才能在修改系统时涉及的改动最少。

与 Apache 等 Web 服务器集成时还要注意，必须将所有的*.htm 请求交给 Resin 处理，否则，前端的 Web 服务器将直接读取 Velocity 模版文件并返回给用户。

11.8 实现全文搜索

在 Live 在线书店应用中，我们需要实现一个非常有用的功能：让用户能通过关键字

搜索到相关的书籍，并且相关度高的书籍应排在搜索结果的前面。这种根据关键字搜索的功能通常称为全文搜索，其原理和 Google 搜索是一样的，不同之处在于 Google 搜索的内容是整个因特网的网页，而我们这个 Web 应用程序搜索的内容仅限于本地数据库存储的书籍信息。

11.8.1 全文搜索简介

全文搜索是根据关键字搜索的一种查询方式，能够快速定位包含该关键字的相关记录，其实现原理是分词+索引。举一个最简单的例子，许多书籍附录都有一个关键词汇的索引表，用户根据词汇表就能得到该词汇出现在书的哪一页，从而快速定位到该页。如果没有这个词汇索引表，我们就只能从书的第一页开始，一页一页地向后查找，其效率自然和通过索引查找相距甚远。全文搜索正是建立这样的一个词汇索引机制，通过扫描每一条记录，建立每个词汇与位置的映射关系，最终得到一个排好序的关键字列表。当用户查询某个关键字时，全文搜索就将这一模糊查询变成利用索引得到精确位置的过程，从而大大提高了关键字查询的效率。

许多读者会问，数据库表的记录也支持索引，并且数据库检索也支持 LIKE 模糊查询，一样可以实现全文搜索，为什么我们还需要一个全文搜索引擎？全文搜索和数据库检索究竟有何不同？

首先，数据库的索引不是为全文搜索准备的，它仅仅能对数据库记录实现排序，即通过某一列的精确值获得该记录的位置，从而快速定位该记录。使用 LIKE '%keyword%' 查询时，数据库索引是不起作用的，这一查询又变成了从头查找的过程，如果数据库记录非常多，使用 LIKE 查询的效率是非常低的，如果需要对多个关键字进行模糊匹配，例如，LIKE '%keyword1%' and LIKE '%keyword2%'，其效率就更低了。

因此，全文搜索和数据库索引机制是不同的，这也是大部分数据库对全文搜索支持有限的原因。虽然某些数据库也支持有限的全文搜索，不过其效率和灵活性仍不能与全文搜索引擎相比。我们可以对比一下全文搜索引擎和数据库 LIKE 查询的差异，如表 11-5 所示。

表 11-5

	全文搜索引擎	数 据 库
索引	将所有数据都建立关键字索引	无法使用索引进行 LIKE 查询
匹配效果	按单词匹配，使用“ant”不会匹配“planting”	无法按单词匹配，使用“%ant%”也会匹配“planting”
匹配度	有匹配度算法，搜索结果按匹配度从高到低排列	没有匹配度算法，无法按匹配度对搜索结果进行排序
可定制性	提供编程接口，容易定制索引规则	只能使用 SQL 查询，无法定制

全文搜索还需要解决的一个最大的问题是让与关键字相关度最高的结果出现在最前面，尽可能地把用户希望看到的结果排在首页，这是数据库 LIKE 查询无法做到的。

我们需要的是一个全文搜索引擎，幸运的是，在 Java 领域，已经有一个免费而出色的全文搜索引擎 Lucene，它是一个纯 Java 编写的全文搜索引擎，可以嵌入到任何 Java 应用程序中实现全文搜索功能。Lucene 的创始人 Doug Cutting 就是一位资深的搜索引擎专家。

许多应用程序已经内置了 Lucene 作为全文搜索引擎，包括 Eclipse、JIRA、TheServerSide.com 等。

Lucene 并不是一个完整的全文搜索应用，它只提供了全文搜索引擎和相关编程接口，没有涉及任何 UI 功能。和数据库相比，Lucene 提供了以下几个主要类来存储索引数据，如表 11-6 所示。

表 11-6

	Lucene	数 据 库
Document	索引中的存储单元，包含多个 Field	类似数据库表的 Record 记录
Field	一个字段的完整内容	类似数据库表的 Field 字段
Hits	代表一个搜索结果，它持有一个按相关度排序的 Document 列表	类似数据库查询的 ResultSet 结果集，由多个 Record 构成

和关系数据库中表的字段是固定的不同，Lucene 的 Document 可以包含任意数量的 Field。此外，Lucene 操作的所有数据都是 String 类型的字符串。使用 Lucene 实现全文搜索就是调用其提供的几个主要接口实现索引和搜索两大功能。

（1）Analyzer：在创建索引时分析文本，将文本拆为最小的词元。

（2）IndexReader：读取索引。

（3）IndexSearcher：根据关键字查询索引以实现根据全文搜索。

（4）IndexWriter：创建并维护索引。

（5）QueryParser：分析查询字符串以获得关键字表达式，例如，包含关键字 A 但不包含 B。

利用 Lucene 提供的以上主要接口即可实现全文搜索功能。此外，Lucene 还具有相当良好的扩展性，对中文支持只需添加一个 ChineseAnalyzer 即可。

11.8.2 集成Compass

由于 Lucene 提供的 API 都比较底层，需要我们手动编写大量的代码来实现索引和搜索两大部分的功能，并且操作的数据是 Lucene 提供的 Document 和 Field 类，需要手动转化我们自定义的域对象，因此使用起来较为烦琐。Compass 正是为了简化 Lucene

的使用而创建的框架。

在 Live 在线书店应用中，我们需要实现全文检索的是书籍信息，即 Book 对象，如图 11-27 所示。

错误！

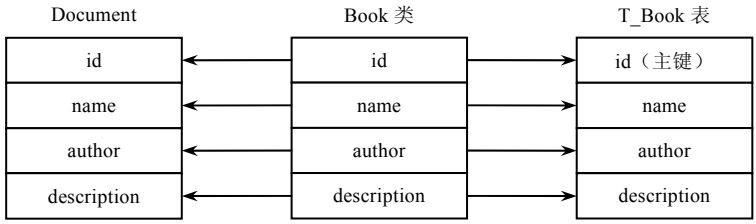


图 11-27

我们可以简单地将 Compass 看成是一个封装了 Lucene 的“ORM”框架，使用 Lucene 就好比直接使用 JDBC 操作数据库一样烦琐，而使用 Compass 就好比使用 Hibernate 操作数据库一样，可以自动完成实体对象到 Lucene 提供的 Document 对象的转化，正如 Hibernate 自动完成实体对象到数据库记录的转化一样。此外，Compass 还隐藏了 Lucene 复杂的多线程模型，避免了读写索引的线程同步问题。

11.8.3 实现全文搜索

正如 Hibernate 将 Book 对象映射到数据库的表记录，Compass 将 Book 对象映射到 Lucene 的 Document 对象中，因此，Compass 的 API 设计和 Hibernate 很类似，也提供 Compass、CompassSession、CompassTransaction 等对象，Compass 类似 Hibernate 的 SessionFactory，提供一个全局的 Compass 实例，并负责打开 CompassSession；CompassSession 提供了 create()和 delete()方法用于创建和删除索引，find()方法用于实现全文搜索；CompassTransaction 提供了事务支持，典型的代码如下。

```
CompassSession session = compass.openSession();
CompassTransaction tx = session.beginTransaction();
// 对 session 进行操作
// ...
tx.commit();
session.close();
```

使用 Compass 的第一步就是建立 Book 对象到 Document 的映射关系。Compass 同时支持 XML 和 Java 5 注解两种方式。由于我们已经使用了 Java 5 平台，因此采用 Java 5 注解是最简单方便的映射方式。在 Book 类中，对以 get 开头的属性添加 Compass 注解。

```
@Searchable
public class Book extends UUIDSupport {
```

```
    @SearchableProperty
    public String getAuthor() { return author; }

    @SearchableProperty(index=Index.UN_TOKENIZED, store=Store.YES)
    public String getPublisher() { return publisher; }

    @SearchableProperty(boost=2)
    public String getName() { return name; }

    @SearchableProperty(boost=2)
    public String getOriginalName() { return originalName; }

    @SearchableProperty
    public String getDescription() { return description; }

    @SearchableProperty(index=Index.NO, store=Store.YES)
    public String getLanguage() { return language; }

    @SearchableProperty(converter="date", store=Store.YES, index=Index.NO)
    public Date getPubDate() { return pubDate; }

    @SearchableProperty(index=Index.NO, store=Store.YES)
    public float getPrice() { return price; }

    @SearchableProperty(index=Index.NO, store=Store.YES)
    public int getDiscount() { return discount; }
    ...
}
```

将 Book 类标注为@Searchable 就告诉 Compass 该类将映射到 Lucene 的 Document，然后在需要实现搜索的属性上标注@SearchableProperty，就可以将该属性映射到 Document 中的 Field。@SearchableProperty 包含几个可选的属性，如表 11-7 所示。

表 11-7

参　数	默 认 值	含　义
index	Index.YES	是否索引该属性，若设置为 Index.NO，则该属性不会被索引
store	Store.YES	是否存储该属性，若设置为 Store.NO，则该属性不会被存储，要获得该属性就必须设法从数据库中查询
converter	""	如何实现该属性和 String 之间的转化，除了 String 和基本类型外，其他类型需要手动指定一个转化器，例如，java.util.Date 类型
boost	1.0F	定义该属性在索引中的重要性

对于那些需要实现全文搜索的属性，例如，Book 的 name 属性，直接标注为@SearchableProperty 即可；对于那些需要存储但不需要搜索的属性，例如，Book 的 price 属性，用户不会根据价格来搜索书籍，但是，搜索结果应当列出书籍的价格，因此，该属性被标记为@SearchableProperty(index=Index.NO, store=Store.YES)，表示不索引，但是要存储；对于非 String 类型或基本类型的属性，例如，Book 的 pubDate 属性，需要为其指定一个转化器，稍后我们会讨论 Compass 自带的几个常用的转化器。

最重要的 boost 参数用于表示该属性在搜索结果中的重要性，例如，当用户搜索“Spring”时，在标题中包含 Spring 关键字的书籍的重要性就比在描述中包含 Spring 关键字的书籍要高，因此，boost 将直接影响搜索的排序结果。默认的 boost 值为 1.0F，将 Book 的 name 属性标记为@SearchableProperty(boost=2)就提高了 name 属性的重要性，使搜索结果的排序更符合用户期望。

最后，为了唯一标识 Book 实体，不要忘了在 Book 的标识属性 id 上标注一个@SearchableId 注解。

```
public abstract class UUIDSupport {
    protected String id;
    @SearchableId
public String getId() { return id; }
public void setId(String id) { this.id = id; }
}
```

下一步，我们编写一个 SearchService 接口，定义创建、删除、更新索引和搜索的几个方法。

```
public interface SearchService {
    // 实现关键字搜索:
    List<Book> search(String q, Page page);
    // 为 Book 创建索引:
    void index(Book book);
    // 删除 Book 的索引:
    void unindex(Book book);
    // 重建 Book 的索引:
    void reindex(Book book);
}
```

SearchService 的实现类 SearchServiceImpl 需要使用 Compass 实现上述功能。SearchServiceImpl 持有一个 Compass 引用，并负责创建和销毁 Compass 实例。

```
/**
 * @spring.bean id="searchService" init-method="init" destroy-method=" destroy"
 */
```

```
public class SearchServiceImpl implements SearchService {
    private final Log log = LogFactory.getLog(getClass());

    private String directory;
    private Compass compass;

    /**
     * 设置索引的存储目录
     * @spring.property value="/WEB-INF/compass"
     */
    public void setIndexDirectory(Resource resouce) {
        try {
            File dir = resouce.getFile();
            if(!dir.isDirectory()) {
                if(!dir.mkdirs()) {
                    throw new ExceptionInInitializerError("Could not create
directory for compass: " + dir.getPath());
                }
            }
            directory = dir.getPath();
            log.info("Set compass directory: " + directory);
        }
        catch(IOException e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    /**
     * 初始化 Compass
     */
    public void init() {
        DateConverter dateConverter = new DateConverter();
        dateConverter.setFormat("yyyy-MM-dd");
        compass = new CompassAnnotationsConfiguration()
                .setConnection(directory)
                .addClass(Book.class)
                .registerConverter("date", dateConverter)
                .buildCompass();
    }

    /**
     * 销毁 Compass
     */
    public void destroy() {
        compass.close();
```

```
        }
        ...
    }
```

创建 Compass 的实例非常容易，设置好索引的存储目录，添加 Book 类，然后注册一个 Date 转化器即可。销毁 Compass 实例只需要调用 close()方法。下面需要实现的是索引的创建、删除、更新和搜索功能。

使用 Compass 为 Book 创建索引非常容易，编写下列代码。

```
public void index(Book book) {
    CompassSession session = null;
    CompassTransaction tx = null;
    try {
        session = compass.openSession();
        tx = session.beginTransaction();
        session.create(book);
        tx.commit();
    }
    catch(RuntimeException e) {
        tx.rollback();
        throw e;
    }
    finally {
        if(session!=null)
            session.close();
    }
}
```

读者可以看出，Compass 的 API 设计和 Hibernate 很类似，也有 Session、Transaction 的概念。删除一个索引的代码如下。

```
public void unindex(Book book) {
    CompassSession session = null;
    CompassTransaction tx = null;
    try {
        session = compass.openSession();
        tx = session.beginTransaction();
        session.delete(book);
        tx.commit();
    }
    catch(RuntimeException e) {
        tx.rollback();
        throw e;
    }
```

```
    finally {
        if(session!=null)
            session.close();
    }
}
```

在 Compass 中更新索引和 Hibernate 有所不同，由于 Lucene 没有更新索引的 API，更新索引的唯一方法就是先删除原有的索引，再创建新索引，因此 reindex(Book book) 方法的代码如下。

```
public void reindex(Book book) {
    log.info("Reindex book...");
    CompassSession session = null;
    CompassTransaction tx = null;
    try {
        session = compass.openSession();
        tx = session.beginTransaction();
        session.delete(book);
        session.create(book);
        tx.commit();
    }
    catch(RuntimeException e) {
        tx.rollback();
        throw e;
    }
    finally {
        if(session!=null)
            session.close();
    }
}
```

实现了索引的创建、删除和更新后，就可以实现全文搜索功能。

```
public List<Book> search(String q, Page page) {
    CompassSession session = compass.openSession();
    CompassTransaction tx = session.beginTransaction();
    try {
        CompassHits hits = session.find(q);
        int count = hits.length();
        page.setTotalCount(count);
        if(count==0) {
            tx.commit();
            return Collections.EMPTY_LIST;
        }
        // 取出 hits[start, end):
```

```
            int start = page.getFirstResult();
            int end = start + page.getPageSize();
            if(end > count)
                end = count;
            List<Book> results = new ArrayList<Book>(end-start);
            for(int i=start; i<end; i++) {
                results.add((Book)hits.data(i));
            }
            tx.commit();
            return results;
        }
        catch(RuntimeException e) {
            tx.rollback();
            throw e;
        }
        finally {
            if(tx!=null)
                tx.commit();
            if(session!=null)
                session.close();
        }
    }
```

用 Compass 实现全文搜索的核心代码只有一行，即 CompassHits hits = session.find (q)。然后，根据当前页码取出相应的结果集，放入 List 中返回即可。完成了上述搜索接口后，我们就可以在 Web 层添加搜索功能了。

首先，Book 对象的创建、删除和更新（这里指数据库操作）都需要相应地修改 Lucene 索引。在 Live 在线书店应用中，只有管理员才有权调用 BusinessSerivce 接口的 createBook()、delete()和 updateBook()方法，因此，只需要在相应的 Controller 中注入 SearchService，然后调用 SearchService 相应的方法即可。例如，在书籍管理的 AdminBook Controller 中为新的 Book 对象创建索引。

```
    public Map handle(HttpServletRequest request, HttpServletResponse response)
throws Exception {
        ...
        if(action.equals("add")) {
            ...
            businessService.createBook(book);
            searchService.index(book);
        }
        else if(action.equals("delete")) {
            ...
            businessService.delete(book);
```

```
                searchService.unindex(book);
            }
            ...
        }
```

最后，为了让用户能实现根据关键字搜索，我们编写了一个 SearchBooksController。

```
        /**
         * @spring.bean name="/searchBooks.jspx"
         */
        public class SearchBooksController extends AbstractMvcController {
            public Map handle(HttpServletRequest request, HttpServletResponse
response) throws Exception {
                int pageIndex = HttpUtil.getInt(request, "page", 1);
                String q = HttpUtil.getString(request, "q");
                Page page = new Page(pageIndex);
                long start_time = System.currentTimeMillis();
                List<Book> books = searchService.search(q, page);
                long cost_time = System.currentTimeMillis() - start_time;
                float cost = cost_time / 1000f;
                Map<String, Object> map = new HashMap<String, Object>();
                map.put("q", q);
                map.put("page", page);
                map.put("books", books);
                map.put("time", String.valueOf(cost));
                return map;
            }

            public String getView(HttpServletRequest request, HttpServletResponse
response) {
                return "/search.htm";
            }
        }
```

search.htm 是对应的 Velocity 模版文件。需要注意的是，只能调用 Book 对象中标记为@SearchableProperty 的属性，因为 Compass 从搜索结果的 Document 中构造 Book 对象时，只能为标记为@SearchableProperty 的属性设置相应的值，其他属性均未初始化。

用户在 Live 在线书店应用中搜索关键字“JavaEE”的结果如图 11-28 所示。

为了维护索引，当应用程序迁移到另一台机器上时，或者由于手动更改了数据库导致数据库记录和 Lucene 索引状态不一致时，就需要为所有的书籍重建索引。因此，Live 在线书店还为管理员提供了一个 AdminRebuildIndexController，用于重建当前所有书籍的索引，请读者参考源代码。

图 11-28

11.9 发送E-mail

电子邮件也是应用程序需要支持的重要功能。在 Live 在线书店中，用户注册、订单通知等均需要通过电子邮件告知用户，因此，组件 MailService 即封装 E-mail 服务，便于 Web 层方便地发送 E-mail。

在第 9 章中已经介绍了 Spring 提供的 JMS 支持，可以通过 JMS 非常方便地实现异步发送 E-mail，而不必自己实现线程池。Live 在线书店的 MailService 组件仍采用 JMS 实现 E-mail 的异步发送。MailService 接口如下。

```
public interface MailService {
    void sendRegistrationMail(Account account);
   void sendBroadcastMail(List<Account> accounts, String subject, String text);
    void sendOrderMail(Order order);
}
```

实现类 MailServiceImpl 的任务就是构造一个 SimpleMailMessage，然后通过 JMS 将其发送出去即可。

```
/**
 * @spring.bean id="mailService"
 */
public class MailServiceImpl implements MailService {

    private final Log log = LogFactory.getLog(getClass());
    private static final String ACCOUNT_USERNAME = "\\$\\{USERNAME\\}";
    private String from;
    private String signature;
```

```
    private final String registration_subject = "恭喜您成功注册成为 Live 在线书店
的会员！";
    private final String registration_body;
    private final String order_subject = "订单通知：您在 Live 在线书店的订单";
    private final String order_body;

    private JmsTemplate jmsTemplate;
    private Destination destination;

    public MailServiceImpl() {
        // load templates:
     String packagePath = getClass().getPackage().getName().replace('.', '/') + "/";
        this.signature = loadTemplate(packagePath + "signature.txt");
      this.registration_body = loadTemplate(packagePath + "registration. txt");
        this.order_body = loadTemplate(packagePath + "order.txt");
    }

    /**
     * @spring.property ref="jmsConnectionFactory"
     */
    public void setConnectionFactory(ConnectionFactory cf) {
        this.jmsTemplate = new JmsTemplate(cf);
    }

    /**
     * @spring.property ref="jmsDestination"
     */
    public void setDestination(Destination destination) {
        this.destination = destination;
    }

    /**
     * @spring.property value="service@crackJavaEE.com"
     */
    public void setFrom(String from) {
        this.from = from;
    }

    public void sendRegistrationMail(Account account) {
        String text = registration_body + signature;
        SimpleMailMessage mail = new SimpleMailMessage();
        mail.setFrom(from);
        mail.setTo(account.getEmail());
        mail.setSubject(registration_subject);
```

```
            mail.setText(text.replaceAll(ACCOUNT_USERNAME, account.getDisplayName()));
            sendMessage(mail);
        }

        public void sendBroadcastMail(List<Account> accounts, String subject,
String text) {
            ...
        }

        public void sendOrderMail(Order order) {
            ...
        }

        protected void sendMessage(final SimpleMailMessage message) {
            this.jmsTemplate.send(this.destination,
                    new MessageCreator() {
                        public Message createMessage(Session session) throws
JMSException {
                            return session.createObjectMessage(message);
                        }
                    }
            );
        }
        ...
    }
```

邮件内容的模版从文本文件中读取。注意到 MailServiceImpl 只负责发送 JMS 消息，并不处理真正的 E-mail 发送。处理 E-mail 的组件是 MailListener，它实现了 JMS 的 MessageListener 接口，一旦收到 ObjectMessage 消息，就将其转换回 SimpleMailMessage，然后通过 JavaMailSender 组件发送 E-mail。

```
    /**
     * @spring.bean id="mailListener"
     */
    public class MailListener implements MessageListener {

        private Log logger = LogFactory.getLog(getClass());
        private MailSender mailSender;

        /**
         * @spring.property ref="mailSender"
         */
        public void setMailSender(MailSender mailSender) {
            this.mailSender = mailSender;
```

```
    }

    public void onMessage(Message message) {
        if(message instanceof ObjectMessage) {
            try {
                Object object = ((ObjectMessage)message).getObject();
                if(object instanceof SimpleMailMessage) {
                    SimpleMailMessage mail = (SimpleMailMessage)object;
                    mailSender.send(mail);
                }
            }
            catch (JMSException e) {
                logger.warn("Send mail failed.", e);
            }
        }
    }
}
```

注意到 MailListener 组件和整个系统的其他组件几乎没有任何耦合关系，因此，如果发送 E-mail 的压力很大，完全可以将其独立出来，甚至放到另一台服务器上。

如果需要调度发送 E-mail 的任务，考虑使用 Quartz 来实现定时发送。请读者参考第 9 章自行实现。

11.9.1 配置JMS

Live 在线书店使用 Resin 3.1 作为服务器，因此，需要在 Resin 的配置文件 resin.conf 中定义 JMS Queue，并且使用可持久化的 Queue。

```
<!-- - Resin 3.1 configuration file. -->
<resin xmlns="http://caucho.com/ns/resin"
      xmlns:resin="http://caucho.com/ns/resin/core">
    ...
    <cluster id="">
        ...
        <!--
          - factory name is java:comp/env/jms/factory
          - queue name is java:comp/env/jms/queue
          -->
        <database jndi-name="jdbc/jms">
            <driver type="com.caucho.db.jdbc.ConnectionPoolDataSourceImpl">
                <path>WEB-INF/jms</path>
            </driver>
        </database>
```

```
            <resource jndi-name="jms/factory" type="com.caucho.jms.Connection
FactoryImpl">
                <init>
                    <data-source>jdbc/jms</data-source>
                </init>
            </resource>
            <resource jndi-name="jms/queue" type="com.caucho.jms.jdbc.JdbcQueue">
                <init>
                    <queue-name>queue</queue-name>
                    <data-source>jdbc/jms</data-source>
                </init>
            </resource>
            ...
        </cluster>
    </resin>
```

11.10 发布Web服务

在第 8 章中，我们已经介绍了多种远程访问机制，Web 服务则是应用最广泛的远程访问，能真正将 Web 网站变为 Web 应用。

由于有了 XFire，并且由于其与 Spring 良好的集成性，使得在 Spring 应用程序中发布 Web 服务变得轻而易举。

11.10.1 实现一个书籍搜索的Web服务

我们打算将书籍搜索发布为一个 Web 服务，以便其他任何第三方应用程序都可以通过这个 Web 服务实现关键字搜索。

我们将 Web 服务所需的接口和类全部放在 net.livebookstore.web.service 包中。搜索返回的书籍信息虽然可以由作为域对象的 Book 类表示，不过，由于 Book 还包含了许多额外的属性，因此，我们不希望客户看到他们不能使用的属性，所以需要一个 BookResult 对象来转换搜索结果。

```
public class BookResult {
    private String id;
    private String name;
    private String originalName;
    private String author;
    private String language;
    private Date pubDate;
    private String image;
```

```
    // getters and setters ...
}
```

由于方法的返回值只有一个，因此还需要一个 SearchResult 包装搜索结果。

```
public class SearchResult {
    private int totalCount;
    private int pageIndex;
    private int pageCount;
    private int pageSize;
    private BookResult[] books;

    // getters and setters ...
}
```

现在，我们来定义 Web 服务类的接口。

```
public interface BookstoreWebService {
    SearchResult search(String keyword, int pageIndex);
}
```

实现类 BookstoreWebServiceImpl 也相当简单，只需要注入 SearchService 组件，然后对搜索结果稍做简化，再使用标准的 JSR 181 注解，即可实现整个 Web 服务。

```
@WebService(
    name="BookstoreWebService",
    serviceName="BookstoreWebService",
    targetNamespace="http://www.livebookstore.net/BookstoreWebService"
)
public class BookstoreWebServiceImpl implements BookstoreWebService {
    private SearchService searchService;
    private String imageRootUrl;
    private static final SearchResult EMPTY_RESULT = new SearchResult(new
BookResult[0], 0, 0, 1, Page.DEFAULT_PAGE_SIZE);

    // 注入 Image 的根 URL
    public void setImageRootUrl(String imageRootUrl) {
        if(imageRootUrl==null || imageRootUrl.trim().equals(""))
            throw new IllegalArgumentException("Property imageRootUrl is null
or empty.");
        this.imageRootUrl = imageRootUrl.trim();
    }

    // 注入 SearchService
    public final void setSearchService(SearchService searchService) {
        this.searchService = searchService;
```

```
        }

        @WebMethod
        public @WebResult SearchResult search(@WebParam String keyword, @WebParam
int pageIndex) {
            Page page = new Page(pageIndex);
            List<Book> books = searchService.search(keyword, page);
            if(page.getTotalCount()==0)
                return EMPTY_RESULT;
            BookResult[] results = new BookResult[books.size()];
            for(int i=0; i<results.length; i++) {
                // 复制 Book 到 BookResult:
                BookResult result = new BookResult();
                Book book = books.get(i);
                result.setId(book.getId());
                result.setName(book.getName());
                result.setOriginalName(book.getOriginalName());
                result.setAuthor(book.getAuthor());
                result.setLanguage(book.getLanguage());
                result.setPubDate(book.getPubDate());
                result.setImage(imageRootUrl + book.getImage());
                results[i] = result;
            }
            return new SearchResult(results,
                   page.getTotalCount(), page.getPageCount(),
                   page.getPageIndex(), page.getPageSize());
        }
    }
```

下一步是将 Web 服务在 Spring 的 XML 配置文件中装配起来。参考第 8 章使用 XFire 发布 Web 服务，我们能很容易地实现该 Web 服务。由于这个使用 XFire 的 Web 服务的访问地址总是以“/bookstoreWebService”开头，因此，在 Spring Web MVC 中定义的*.jspx 映射对该 Web 服务无效。我们需要为 DispatcherServlet 再添加一个 URL 映射。

```
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/bookstoreWebService*</url-pattern>
    </servlet-mapping>
```

读者可以参考第 8 章使用 Visual Studio 2005 或其他任何客户端来访问这个 Web 服务。

11.11 监控系统运行状态

在第 9 章中，我们已经了解了 Spring 对 JMX 的良好集成。通过 JMX，我们就能够

获得一种标准的应用程序监控能力。为了能时刻了解 Live 在线书店的运行状态，我们仍采用 JMX 实现对系统的监控。

除了利用 JVM 内置的 MBean 对内存、CPU 使用率、线程状态进行监控外，对应用程序的性能进行监控也是非常必要的，因此，我们设计一个 SamplerMBean，实现对性能的采样。

```
public interface SamplerMBean {
    long getTotal(); // 运行总时间
    long getCount(); // 采样次数
    long getMax(); // 最大运行时间
    long getAverage(); // 平均运行时间
    void clear(); // 重置计数器
}
```

对应的实现类 Sampler 代码如下。

```
/**
 * @spring.bean name="sampler:name=HttpSampler"
 * @spring.bean name="sampler:name=BusinessSampler"
 * @spring.bean name="sampler:name=DaoSampler"
 */
public class Sampler implements SamplerMBean {
    private transient long total;
    private transient long count;
    private transient long max;

    public long getTotal() { return total; }
    public long getCount() { return count; }
    public long getMax() { return max; }

    public long getAverage() {
        return count==0 ? 0 : total / count;
    }

    public final void clear() {
        total = count = max = 0;
    }

    public final void sample(long time) {
        total += time;
        count ++;
        max = (time>max) ? time : max;
    }
}
```

Sampler 实现一个没有严格同步的采样器，并且我们通过 XDoclet 注释定义了 3 个 MBean，分别对 HTTP 请求、BusinessService 接口和 DAO 接口的方法调用时间进行采样。

利用 Filter 对整个 HTTP 请求的时间进行采样很容易实现，我们定义一个 PerformanceFilter 用于实现采样。

```
public class PerformanceFilter implements Filter {
    private static final String START_TIME = "ST_PERF_FILTER";
    private final Log log = LogFactory.getLog(getClass());
    private Sampler httpSampler;
    public void setHttpSampler(Sampler sampler) {
        this.httpSampler = sampler;
    }

    public void init(FilterConfig config) throws ServletException {}
    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        String url = HttpUtil.getURL((HttpServletRequest)request);
        if(url.endsWith("/checkState.jspx")) {
            chain.doFilter(request, response);
            return;
        }
      request.setAttribute(START_TIME, new Long(System.currentTimeMillis()));
        try {
            chain.doFilter(request, response);
        }
        finally {
            long httpTime = System.currentTimeMillis() - ((Long)request.
getAttribute(START_TIME)).longValue();
            httpSampler.sample(httpTime);
        }
    }
}
```

如果要对 BusinessService 接口和 DAO 接口的运行时间实现采样，就需要利用 AOP 记录每次调用时间。由于我们希望计算的是一次完整的 HTTP 请求中调用 BusinessService 接口和 DAO 接口所需的总时间，因此，利用 ThreadLocal 作为累加器是合适的。

```
public class PerformanceFilter implements Filter {
   public static final ThreadLocal<Long> businessPerf = new ThreadLocal< Long>();
    public static final ThreadLocal<Long> daoPerf = new ThreadLocal<Long>();

    private static final String START_TIME = "ST_PERF_FILTER";
```

```
    private final Log log = LogFactory.getLog(getClass());

    private Sampler httpSampler;
    private Sampler businessSampler;
    private Sampler daoSampler;

    /**
     * @spring.property ref="sampler:name=HttpSampler"
     */
    public void setHttpSampler(Sampler sampler) {
        this.httpSampler = sampler;
    }
    /**
     * @spring.property ref="sampler:name=BusinessSampler"
     */
    public void setBusinessSampler(Sampler sampler) {
        this.businessSampler = sampler;
    }
    /**
     * @spring.property ref="sampler:name=DaoSampler"
     */
    public void setDaoSampler(Sampler sampler) {
        this.daoSampler = sampler;
    }

    public void init(FilterConfig config) throws ServletException {}
    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        String url = HttpUtil.getURL((HttpServletRequest)request);
        if(url.endsWith("/checkState.jspx")) {
            chain.doFilter(request, response);
            return;
        }
       request.setAttribute(START_TIME, new Long(System.currentTimeMillis()));
        businessPerf.set(new Long(0));
        daoPerf.set(new Long(0));
        try {
            chain.doFilter(request, response);
        }
        finally {
            long httpTime = System.currentTimeMillis() - ((Long)request.
getAttribute(START_TIME)).longValue();
            long businessTime = businessPerf.get().longValue();
```

```
                long daoTime = daoPerf.get().longValue();
               long businessPercent = httpTime==0 ? 0 : businessTime * 100 / httpTime;
                long daoPercent = httpTime==0 ? 0 : daoTime * 100 / httpTime;
                httpSampler.sample(httpTime);
                businessSampler.sample(businessTime);
                daoSampler.sample(daoTime);
                log.info(url + " execution: " + httpTime + "ms.");
                log.info("Business execution: " + businessTime + "ms, " +
businessPercent + "%");
                log.info("Dao execution: " + daoTime + "ms, " + daoPercent + "%");
                businessPerf.remove();
                daoPerf.remove();
            }
        }
    }
```

具体的累加在每次执行特定方法时由 AOP 实现计时，代码如下。

```
    @Aspect
    public class SamplerAspect {
       @Around("execution(* net.livebookstore.business.BusinessService.*(..))")
       public Object businessExecution(ProceedingJoinPoint pjp) throws Throwable {
           return doSample(pjp, PerformanceFilter.businessPerf);
        }

        @Around("execution(* net.livebookstore.dao.*Dao.*(..))")
        public Object daoExecution(ProceedingJoinPoint pjp) throws Throwable {
           return doSample(pjp, PerformanceFilter.daoPerf);
        }

        private Object doSample(ProceedingJoinPoint pjp, ThreadLocal<Long> tl)
throws Throwable {
            long start = System.currentTimeMillis();
            try {
                return pjp.proceed();
            }
            finally {
                long last = System.currentTimeMillis() - start;
                Long acc = tl.get();
                if(acc!=null)
                    tl.set(new Long(acc.longValue() + last));
            }
        }
    }
```

最后一步是将 SampleAspect 织入到合适的组件中，利用<aop:aspectj-autoproxy />即

可实现。为了便于查看系统状态，利用 HtmlAdaptorServer 使得我们能通过浏览器直接连接 JMX。为此，添加 HtmlAdaptorServer，并设定一个用户名和口令。

```
        <!-- 启动 HTML Adaptor -->
        <bean name="adaptor:name=html" class="com.sun.jdmk.comm.HtmlAdaptorServer"
init-method="start">
            <!-- 监听端口 8088 -->
            <constructor-arg value="8088" />
            <constructor-arg>
                <list>
                    <!-- 用户认证信息 -->
                    <bean class="com.sun.jdmk.comm.AuthInfo">
                        <property name="login" value="jmxadmin" />
                        <property name="password" value="livebookstore" />
                    </bean>
                </list>
            </constructor-arg>
            <!-- 最多允许同时连接的客户端 -->
            <property name="maxActiveClientCount" value="10" />
        </bean>

       <bean id="mbeanExporter" class="org.springframework.jmx.export.MBeanExporter">
            <property name="autodetect" value="true" />
        </bean>
```

应用程序运行一段时间后，就可以通过浏览器在 8088 端口查看我们收集的系统性能统计数据，如图 11-29 所示。

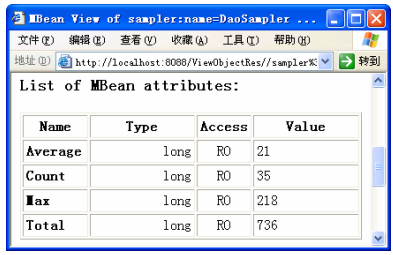


图 11-29

11.12 优化系统性能

为了尽量榨取硬件的潜能，我们需要在最后对应用程序进行优化。缓存是提高性能的最有效的手段。由于内存的速度远远快于磁盘访问速度，因此，将常用的数据缓存在内存中可以大大提高系统的性能。

11.12.1 OSCache缓存介绍

OSCache 是一个免费的开源缓存框架，其特点是支持多种缓存算法，配置简单。在 Spring 中，可在 XML 配置文件中直接配置 OSCache。

OSCache 的编程接口极为简单。OSCache 提供了一个简单的构造方法，用于构造一个 Cache 实例。

```
public Cache(boolean useMemoryCaching, // 是否使用内存缓存
       boolean unlimitedDiskCache, // 是否使用磁盘缓存
       boolean overflowPersistence, // 溢出后是否持久化
       boolean blocking, // 是否阻塞
       String algorithmClass, // 缓存算法
       int capacity // 缓存容量
)
```

OSCache 提供了好几种缓存算法，最常用的是 LRU 算法，即最长不使用的被优先移出，这种算法能满足绝大部分缓存模型。更新缓存的典型代码如下。

```
cache = new Cache(true, false, false, false,
       "com.opensymphony.oscache.base.algorithm.LRUCache",
       100);
expiresRefreshPolicy = new ExpiresRefreshPolicy(60); // 缓存 60 秒
...
try {
   Object content = cache.getFromCache("key");
}
catch(NeedsRefreshException e) {
   boolean updated = false;
   try {
      Object content = ... // 重建缓存
      cache.putInCache("key", newContent, expiresRefreshPolicy);
      updated = true;
   }
   finally {
      if(!updated)
         cache.cancelUpdate("key");
   }
}
```

在捕获到 NeedsRefreshException 后，如果没有更新缓存内容，务必调用 cancelUpdate() 方法以避免死锁。

11.12.2 设计缓存模型

考虑到数据库查询速度较慢，因此，如果能将数据库中经常使用的数据缓存，则可以大大提高应用程序的性能。

由于采用了多层结构，因此，在各层均可以缓存数据。对于不经常变化的数据，例如，Category 对象，在用户第一次请求时一次性全部读取并缓存在 CategoryDaoImpl 中，此后的访问完全不涉及数据库。在创建、更新或删除 Category 时，采用 ReadWriteLock 模式就可以保证读取和修改 Category 时不会冲突。Java 5 的 java.util.concurrent 包已经内置了 ReadWriteLock 的实现类，我们可以非常方便地直接使用。

```
public class CategoryDaoImpl extends GenericHibernateDao<Category>
        implements CategoryDao {
    private final ReadWriteLock wrLock = new ReentrantReadWriteLock();
    private final Lock readLock = wrLock.readLock();
    private final Lock writeLock = wrLock.writeLock();

    private Category root = null;
    private Map<Integer, Category> categories = null;

    public CategoryDaoImpl() {
        super(Category.class);
    }

    public void init() {
        writeLock.lock();
        try {
            ...
        }
        finally {
            writeLock.unlock();
        }
    }

    public Category queryRoot() {
        readLock.lock();
        try {
            return root;
        }
        finally {
            readLock.unlock();
        }
```

```
    }

    public Category query(Serializable id) {
        readLock.lock();
        try {
            ...
        }
        finally {
            readLock.unlock();
        }
    }

    public void create(Category category, Integer pId) {
        writeLock.lock();
        try {
            ...
        }
        finally {
            writeLock.unlock();
        }
    }

    public void delete(Category category) {
        writeLock.lock();
        ...
    }

    public void update(Category category) {
        writeLock.lock();
        ...
    }
}
```

采用 ReadWriteLock 模式可以保证任何时刻读写均不冲突，即允许同时读，但仅有一个线程能单独写，如表 11-8 所示。

对于读操作远多于写操作的情况，使用 ReadWriteLock 模式能大幅提高系统的读性能。关于 ReadWriteLock 模式的讨论已经超出了本书的范围。读者可以参考《Java 多线程设计模式》一书。

表 11-8

	读	写
读	允许	不允许
写	不允许	不允许

对于查询单个对象，可以考虑在逻辑层缓存。例如，按照主键查找的实体对象，对于这种缓存方式，应用 AOP 是最好的解决方案。然而，对于以 List 方式一次读取多个对象时，如获取某一分类下的书籍列表，放入缓存就不合适了，一是很难确定一个合适的 Key，二是对于后续查询难以获知哪些对象已被缓存，哪些对象没有被缓存。

在逻辑层难于实现的缓存还可以在 Web 层实现，并且越是在上层缓存，性能越好，因为无须调用底层提供的服务了。如果能直接针对输出页面缓存，则可以获得最好的性能。下面，我们主要详细讨论如何对最终输出页面实现缓存。

要最大限度地提高系统性能，可以将整个输出页面缓存，这样，当用户下次请求同一 URL 时，可直接输出缓存的页面，无须经过 Controller 处理，调用逻辑层，最后渲染页面的所有步骤。一个完整的页面缓存逻辑如图 11-30 所示。

错误！

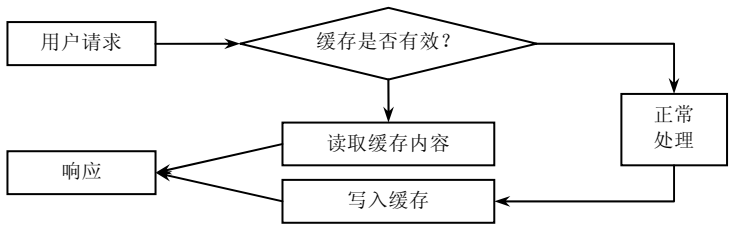


图 11-30

设计页面缓存时，需要考虑的问题是，只有不经常变化的页面将其缓存才有意义，如果页面与当前用户的状态有关，则难以进行缓存。例如，Live 在线书店的每个页面左上角几乎都有当前用户的登录状态信息，直接缓存整个页面是不行的，如图 11-31 所示。

图 11-31

这里用到的关键技巧是 JavaScript。由于 JavaScript 可以使用 document.write()动态生成一部分 HTML 代码，因此，将页面拆为两个部分，主页面是不与用户状态关联的静态页面，与用户状态关联的小部分 HTML 代码由 JavaScript 动态生成，而这个 JavaScript 又由服务器根据当前用户返回，如图 11-32 所示。

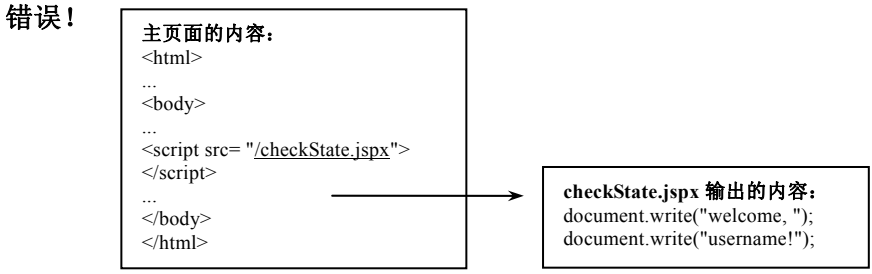


图 11-32

现在，就可以将主页面全部缓存了。

这个技巧还可以进一步扩展，使静态页面也能根据用户当前的登录状态显示不同的内容。例如，显示书籍详细信息的页面内容几乎是不变的，因此，完全可以生成静态页面来加快访问速度。但是，对于评论框的显示与否就要根据当前用户的登录状态决定。同样可以利用 JavaScript 动态判断。让/checkState.jspx 多返回一点内容。

```
document.write("welcome, ");
document.write("username!");
var g_login = true; // 如果没有登录，就返回 false
```

然后，在主页面中根据 g_login 变量判断是否显示评论框。

```
<div id="makeComment" style="display:none">
    请发表评论：<input type="text" />
    <input type="submit" value="发表">
</div>
<script>
    if(g_login)
        document.getElementById("makeComment").style.display="";
</script>
```

为了避免将缓存逻辑引入现有的 Controller 体系，采用 Filter 是最好的选择，可以透明地实现开启或禁用缓存，对系统的其他部分功能却不造成影响。

由于缓存总是通过键-值映射的，缓存的内容是整个页面的输出内容，而缓存的键只能是页面的 URL。

这里需要注意的是，由于需要缓存的常常是动态页面（静态页面常常由 Web 服务器自行实现缓存，大多数 Web 服务器都会自动缓存静态页面），而 URL 的参数经常变化，且有的参数还不一定是必须的，因此，必须有一种转化规则，将一个完整的 URL 转化为一个字符串，相同参数的 URL 转化后必须是相同的字符串，而无论参数顺序是否相同。例如，/list.jsp?a=A&page=1 和/list.jsp?page=/&a=A 实际上是同一页面。如果 page 参数的

默认值是 1，则/list.jsp?a=A 与上述两个 URL 也应该是相同的。此外，无效参数也必须能够被忽略，否则，恶意用户只需添加无效参数（如/list.jsp?a=A&useless=1,2,3...）即会破坏服务器端的缓存。

对于页面内容，有两种缓存地点：内存和静态页面。内存适合固定时长的缓存，例如，1 小时，而静态页面更适合变化极其缓慢的页面。为此，定义 MemoryCacheFilter 和 FileCacheFilter，并抽象出共同的超类 AbstractCacheFilter，如图 11-33 所示。

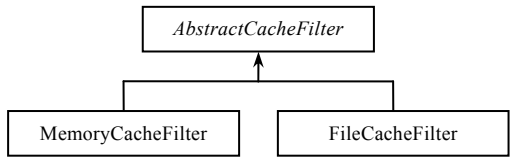


图 11-33

首先要为每个需要缓存的页面定义 CacheEntry，以便判断一个 HttpServletRequest 是否符合缓存要求。

```
class CacheEntry {
    private String url;
    private Map<String, String> parameters = new HashMap<String, String>();
    private int parametersCount;

    /**
     * URL 示例：/user/test.jspx=id,name,page "1"
     */
    public CacheEntry(String config) {
        String[] ss = config.trim().split("\\=");
        if(ss.length!=2)
            throw new IllegalArgumentException("Illegal config: " + config);
        url = ss[0].trim();
        if(url.equals(""))
            throw new IllegalArgumentException("Illegal config: " + config);
        String[] ps = ss[1].split("\\,");
        for(String p : ps)
            addParameter(p);
        parametersCount = parameters.size();
    }

    private void addParameter(String p) {
        p = p.trim();
        String[] ss = p.split("[ ]+", 2);
        String param = ss[0];
        String value = null;
```

```
            if(ss.length==2) {
                value = ss[1].trim();
                if(value.matches("\\\".*\\\""))
                    value = value.substring(1, value.length()-1);
            }
            parameters.put(param, value);
        }

        // 返回 Request 对应的 Key,不匹配将返回 null:
        public String buildKey(HttpServletRequest request) {
            if(!url.equals(request.getRequestURI()))
                return null;
            if(parametersCount==0)
                return url;
            StringBuffer sb = new StringBuffer(256);
            sb.append(this.url).append('?');
            for(String p : parameters.keySet()) {
                String value = request.getParameter(p);
                if(value!=null)
                    sb.append(p).append('=').append(value).append('&');
                else {
                    String dv = parameters.get(p);
                    if(dv==null)
                        return null;
                    sb.append(p).append('=').append(dv).append('&');
                }
            }
            return sb.toString();
        }
    }
```

为了简化配置，构造一个 CacheEntry 仅需要一个字符串，其格式为：

```
URI=parameter1 defaultValue,parameter2 defaultValue,...
```

例如，对于/user/test.jspx?id=12345&name=abc&page=1，则配置为：

```
/user/test.jspx=id,name,page
```

如果 page 有默认值 1，name 有默认值 abc，则加上默认值。

```
/user/test.jspx=id,name abc,page 1
```

这样，无论 URL 的参数顺序是否一致，可选参数是否完整，均能构造出相同的缓存 Key。

CacheDefinition 则包含了一组 CacheEntry，定义 CacheDefinition 对象是为了方便地

在 Spring 的 XML 配置文件中注入一组 CacheEntry。

```
public class CacheDefinition {
    private CacheEntry[] entries;

    public CacheDefinition(CacheEntry[] entries) {
        this.entries = entries;
    }

    public String getKey(HttpServletRequest request) {
        for(CacheEntry entry : entries) {
            String s = entry.buildKey(request);
            if(s!=null)
                return s;
        }
        return null;
    }
}
```

为了一次性注入一组 CacheEntry，例如：

```
<property name="cacheDefinition">
    <value>
        /user/test.jspx=id,name abc,page 1
        /list.jspx=id,category,page 1
        /product.jspx=id,category
    </value>
</property>
```

我们就需要一个自定义属性编辑器 CacheDefinitionEditor。

```
public class CacheDefinitionEditor extends PropertyEditorSupport {
    public void setAsText(String text) throws IllegalArgumentException {
        BufferedReader reader = new BufferedReader(new StringReader(text));
        List<CacheEntry> entries = new ArrayList<CacheEntry>();
        try {
            for(;;) {
                String s = reader.readLine();
                if(s==null)
                    break;
                s = s.trim();
                if(s.equals(""))
                    continue;
                entries.add(new CacheEntry(s));
            }
        }
        catch(IOException e) {}
```

```
            setValue(new CacheDefinition(entries.toArray(new CacheEntry[0])));
        }
    }
```

CacheDefinitionEditor 负责分析注入的多行字符串，然后将每一行转化为 CacheEntry，并构造出 CacheDefinition 对象，这样，AbstractCacheFilter 就可以直接注入 CacheDefinition。AbstractCacheFilter 的主要作用就是负责注入 CacheDefinition 和 contentType 属性。

```
public abstract class AbstractCacheFilter implements Filter, InitializingBean {
    protected Log log = LogFactory.getLog(getClass());

    private String contentType = "text/html;charset=UTF-8";
    private CacheDefinition cacheDefinition;

    public final void setContentType(String contentType) {
        this.contentType = contentType;
    }

    protected final String getContentType() {
        return contentType;
    }

    public final void setCacheDefinition(CacheDefinition cacheDefinition) {
        this.cacheDefinition = cacheDefinition;
    }

    protected final String getKey(HttpServletRequest httpRequest) {
        return cacheDefinition.getKey(httpRequest);
    }

    public void init(FilterConfig config) throws ServletException {}
    public void destroy() {}

    public void afterPropertiesSet() throws Exception {
        if(cacheDefinition==null)
          throw new NullPointerException("Property cacheDefinition must be set.");
        if(contentType==null || contentType.trim().equals(""))
         throw new NullPointerException("Property contentType is null or empty.");
        contentType = contentType.trim();
    }
}
```

具体的缓存实现则由 MemoryCacheFilter 和 FileCacheFilter 实现，分别用于内存缓存和静态页面缓存。

11.12.3 缓存页面到内存

如果某个页面的变化比较缓慢，或者对数据更新的要求不是特别及时，可以考虑将页面缓存到内存中。

例如，用户在浏览书籍时，页面的变化是非常缓慢的，因为只有管理员添加或删除了书籍后，相应的页面才会变化。用户并不一定要求立刻看到管理员添加的新书，因此，完全可以设定一个合适的缓存时间，例如，1 小时，这样，用户最多在 1 小时后就能够看到管理员添加的新书，这已经足够了，因为用户并不在意在新书添加后就必须立刻看到它。

MemoryCacheFilter 采用 OSCache 作为内存缓存的实现，默认采用 LRU 算法，即最近很久未使用的内存首先被移出。其原理为第 7 章介绍的截获 HttpServletResponse，即将一个 Wrapper Response 传递给下一个 FilterChain，然后截获输出的内容，并放入缓存中，再发送给客户端。

```
public class MemoryCacheFilter extends AbstractCacheFilter {
    private Cache cache;
    private int capacity = 100;
    private long expiresTime = 3600000;
    private EntryRefreshPolicy expiresRefreshPolicy;

    // 设置缓存的最大允许页面数量:
    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    // 设置最长缓存时间
    public void setExpiresTime(int expiresTime) {
        if(expiresTime<1 || expiresTime>3600*24*365)
            throw new IllegalArgumentException("Invalid expiresTime: " +
expiresTime);
        this.expiresTime = expiresTime * 1000;
    }

    public void afterPropertiesSet() throws Exception {
        super.afterPropertiesSet();
      expiresRefreshPolicy = new ExpiresRefreshPolicy((int)(expiresTime / 1000));
        // 构造 OSCache 实例:
        cache = new Cache(
                true,  /* 使用内存缓存 */
                false, /* 不使用硬盘缓存 */
```

```
                    false, /* 缓存溢出后不输出到硬盘 */
                    false, /* 是否阻塞 */
                    "com.opensymphony.oscache.base.algorithm.LRUCache", // 算法
                    capacity);
        }

        public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
            HttpServletRequest httpRequest = (HttpServletRequest)request;
            String key = getKey(httpRequest);
            if(key!=null) {
                // 允许缓存:
                HttpServletResponse httpResponse = (HttpServletResponse)response;
                try {
                    GZipResponseContent content = (GZipResponseContent) cache.
getFromCache(key);
                    // 找到了缓存的内容,直接输出:
                    sendResponse(httpRequest, httpResponse, content);
                }
                catch(NeedsRefreshException e) {
                    // 没有找到缓存内容,需要重新构造缓存内容:
                    boolean updated = false;
                    try {
                        // 构造 Wrapper:
                        CachedResponseWrapper wrapper = new CachedResponseWrapper
(httpResponse);
                        chain.doFilter(request, wrapper);
                        // 只对 200 OK 响应缓存:
                        if(wrapper.getStatus()!=HttpServletResponse.SC_OK)
                            return;
                        GZipResponseContent newContent = new GzipResponseContent
(wrapper.getResponseData());
                        cache.putInCache(key, newContent, expiresRefreshPolicy);
                        updated = true;
                        // 向客户端输出页面内容:
                        sendResponse(httpRequest, httpResponse, newContent);
                    }
                    finally {
                        if(!updated)
                            cache.cancelUpdate(key);
                    }
                }
            }
            else {
                // 该页面无法缓存:
```

```
                chain.doFilter(request, response);
            }
        }

        private void sendResponse(HttpServletRequest request, HttpServletResponse
response, GZipResponseContent content)
                throws IOException, ServletException
        {
            long ifModifiedSince = request.getDateHeader("If-Modified-Since");
            long lastModified = content.getLastModified();
            if(ifModifiedSince!=(-1) && ifModifiedSince>=lastModified) {
                // 客户端本地内容尚未过期,直接返回 302:
                response.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
            else {
                response.setContentType(getContentType());
                response.setHeader("Content-Encoding", "gzip");
                response.setHeader("Cache-Control", "Private");
                response.setDateHeader("Expires", lastModified + expiresTime);
                response.getOutputStream().write(content.getData());
            }
        }
    }

    class GZipResponseContent {
        private long lastModified;
        private byte[] data;

        public GZipResponseContent(byte[] data) {
            this.lastModified = System.currentTimeMillis();
            this.data = GZipUtil.gzip(data);
        }
        public long getLastModified() { return lastModified; }
        public byte[] getData() { return data; }
    }
```

注意，在向客户端输出缓存内容时，我们还判断了浏览器传递的 If-Modified-Since 计算浏览器的本地内容是否过期，若没有过期，直接返回 302 Not Modified，进一步减少不必要的传输。在向浏览器输出缓存内容时，同时设置 Expires 告诉浏览器该页面在多久的范围内不会过期，以便浏览器不必重新请求服务器。

如果对于数据更新的及时性要求较高，可以考虑在数据更新后强制清除缓存，这样，后续的下一次请求就会由于缓存命中失败而被迫查询数据库获得最新的数据。不过，这样做将把清除缓存的逻辑引入到逻辑层对象的某些更新的方法中，因此增加了系统维护

的难度。

为了减少内存占用，采用了 GZip 算法对页面进行压缩，这样还可以提高后续请求的网络传输效率。由于现在所有的浏览器都支持 GZip 压缩格式，因此，可以放心地压缩页面。

JDK 已经自带了处理 GZip 的相关 IO 类。我们编写一个 GZipUtil 封装压缩操作。

```
public final class GZipUtil {
    public static byte[] gzip(byte[] data) {
        ByteArrayOutputStream byteOutput = new ByteArrayOutputStream(10240);
        GZIPOutputStream output = null;
        try {
            output = new GZIPOutputStream(byteOutput);
            output.write(data);
        }
        catch (IOException e) {}
        finally {
            if(output!=null) {
                try {
                    output.close();
                }
                catch (IOException e) {}
            }
        }
        return byteOutput.toByteArray();
    }
}
```

注意：不需要编写解压缩的方法，因为发送的页面是压缩的数据，解压缩的工作由客户端的浏览器完成。

最后一步是借用 Acegi 的 FilterToBeanProxy 将 Spring 中定义的 Bean 包装为一个 Filter，在 web.xml 中配置：

```
<filter>
    <filter-name>memoryCacheFilter</filter-name>
    <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
    <init-param>
        <param-name>targetClass</param-name>
        <param-value>
            net.livebookstore.web.filter.MemoryCacheFilter
        </param-value>
```

```
        </init-param>
    </filter>
```

将页面缓存到内存后，对于相同的 URL 请求，服务器的处理速度得到了极大地提高。通常，后续的请求处理仅需 1ms 左右，而处理没有缓存的页面请求需要几十至几百毫秒，大部分时间花费在数据库的查询中。

11.12.4 缓存页面到文件

对于不变的页面，或者变化非常缓慢的页面，可以考虑生成静态文件，以空间换取时间。FileCacheFilter 负责实现缓存页面到静态文件。由于从 AbstractCacheFilter 继承，因此，FileCacheFilter 已经获得了注入的 CacheDefinition。

和 MemoryCacheFilter 的处理流程类似，FileCacheFilter 的实现如下。

```
public class FileCacheFilter extends AbstractCacheFilter {
    private String root;
    private final String SUFFIX = ".html";

    // 设置缓存文件的根目录:
    public final void setFileDir(Resource dir) {
        try {
            File f = dir.getFile();
            f.mkdirs();
            if(!f.isDirectory())
                throw new IllegalArgumentException("Invalid directoy: " +
f.getPath());
            if(!f.canWrite())
                throw new IllegalArgumentException("Cannot write to directory:
" + f.getPath());
            root = f.getPath();
            // make sure root is end with "/":
            if(!root.endsWith("/") && !root.endsWith("//"))
                root = root + "/";
        }
        catch(IOException e) {
            throw new IllegalArgumentException(e);
        }
    }
    public void afterPropertiesSet() throws Exception {
        super.afterPropertiesSet();
        if(!new File(root).isDirectory()) {
            throw new IllegalArgumentException("No directory: " + root);
        }
```

```
}
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException
{
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    String key = getKey(httpRequest);
    if(key==null) {
        // 该页面无法缓存:
        chain.doFilter(request, response);
    }
    else {
        File file = key2File(key);
        if(file.isFile()) {
            // 缓存文件已找到:
          HttpServletResponse httpResponse = (HttpServletResponse) response;
            httpResponse.setContentType(getContentType());
            httpResponse.setHeader("Content-Encoding", "gzip");
            httpResponse.setContentLength((int)file.length());
            // 读取文件内容:
            FileUtil.readFile(file, httpResponse.getOutputStream());
        }
        else {
            // 缓存文件未找到:
          HttpServletResponse httpResponse = (HttpServletResponse) response;
            // 构造 Wrapper:
        CachedResponseWrapper wrapper = new CachedResponseWrapper (httpResponse);
            chain.doFilter(request, wrapper);
            if(wrapper.getStatus()==HttpServletResponse.SC_OK) {
                byte[] data = GZipUtil.gzip(wrapper.getResponseData());
                // 写入文件:
                FileUtil.writeFile(file, data);
                // 发送到客户端:
                httpResponse.setContentType(getContentType());
                httpResponse.setHeader("Content-Encoding", "gzip");
                httpResponse.setContentLength(data.length);
                httpResponse.getOutputStream().write(data);
            }
        }
    }
}
// key->File 的转化:
private File key2File(String key) {
    int hash = key.hashCode();
    int dir1 = (hash & 0xff00) >> 8;
    int dir2 = hash & 0xff;
```

```
            String dir = root + dir1 + "/" + dir2;
            File fdir = new File(dir);
            if(!fdir.isDirectory()) {
                if(!fdir.mkdirs())
                    return null;
            }
            return new File(dir + "/" + encode(key) + SUFFIX);
        }
        // 确保文件名合法:
        private String encode(String key) {
            try {
                return URLEncoder.encode(key, "UTF-8");
            }
            catch(UnsupportedEncodingException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

读写文件封装在 FileUtil 中，采用静态方法，其原理很简单，这里就不再给出源代码了。

使用 FileCacheFilter 时，若页面内容变化后，就必须删除原有的已缓存的静态文件，以便强迫 FileCacheFilter 在下一次请求时重新生成新的静态文件，因此，FileCacheFilter 还有一个 remove()方法。

```
    // 删除一个已缓存的文件:
    public void remove(String url, Map<String, String> parameters) {
        String key = getKey(HttpServletRequestFactory.create(url, parameters));
        if(key!=null) {
            FileUtil.removeFile(key2File(key));
        }
    }
```

为了复用 getKey(HttpServletRequest)方法，我们需要构造一个 HttpServletRequest 对象。利用 JDK 的动态代理功能实现这一“伪造”的 HttpServletRequest 对象非常简便。

```
    public class HttpServletRequestFactory {
        private static final Class[] PROXY_INTERFACES =
            { HttpServletRequest.class };

        public static HttpServletRequest create(final String uri, final Map<String,
String> parameters) {
            return (HttpServletRequest) Proxy.newProxyInstance(
                    HttpServletRequestFactory.class.getClassLoader(),
```

```
                    PROXY_INTERFACES,
                    new InvocationHandler() {
                        public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                            String methodName = method.getName();
                            if(methodName.equals("getRequestURI"))
                                return uri;
                            if(methodName.equals("getParameter"))
                                return parameters.get(args[0]);
                            throw new UnsupportedOperationException("not impl");
                        }
                    }
                );
            }
        }
```

对于这个“伪造”的 HttpServletRequest，只能调用其 getRequestURI()和 getParameter()方法。事实上我们在 getKey()方法中也只调用了这两个方法。

配置 FileCacheFilter 和配置 MemoryCacheFilter 完全一致，也是利用 Acegi 的 FilterToBeanProxy，这里不再多述。

11.12.5 客户端缓存

由于 HTTP 协议还可以设置客户端缓存，因此，可以对某些静态资源设置合理的缓存时间，避免浏览器频繁请求静态资源。

用 ExpiresFilter 可以为静态资源（如 GIF 图像、CSS 样式表等）设置一个较大的缓存时间，如 1 周，这样浏览器会在本地自动缓存静态资源，只要本地的内容没有清空，1 周之内浏览器都不必向服务器请求这些静态资源。ExpiresFilter 根据 URL 判断请求资源的类型并读取 web.xml 配置文件来获得各种静态资源的缓存时间。

```
public class ExpiresFilter implements Filter {
    private Log log = LogFactory.getLog(getClass());
    private Map<String, Long> map = new HashMap<String, Long>();

    public void init(FilterConfig filterConfig) throws ServletException {
        Enumeration en = filterConfig.getInitParameterNames();
        while(en.hasMoreElements()) {
            String paramName = en.nextElement().toString();
            String paramValue = filterConfig.getInitParameter(paramName);
            try {
                int time = Integer.valueOf(paramValue);
```

```
                if(time>0) {
                    log.info("Set " + paramName + " expired seconds: " + time);
                    map.put(paramName, new Long(time));
                }
            }
            catch(Exception e) {
                log.warn("Exception in initilazing ExpiredFilter.", e);
            }
        }
    }
    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        String uri = ((HttpServletRequest)request).getRequestURI();
        int n = uri.lastIndexOf('.');
        if(n!=(-1)) {
            String ext = uri.substring(n);
            Long exp = map.get(ext);
            if(exp!=null) {
                HttpServletResponse resp = (HttpServletResponse)response;
                resp.setHeader("Cache-Control", "max-age=" + exp);
            resp.setDateHeader("Expires", System.currentTimeMillis() + exp * 1000);
            }
        }
        chain.doFilter(request, response);
    }
}
```

在 web.xml 中，配置将所有的 CSS、图片等资源缓存 1 小时。

```
<filter>
    <filter-name>expiresFilter</filter-name>
    <filter-class>
        net.livebookstore.web.filter.ExpiresFilter
    </filter-class>
    <init-param>
        <param-name>.css</param-name>
        <param-value>3600</param-value>
    </init-param>
    <init-param>
        <param-name>.jpg</param-name>
        <param-value>3600</param-value>
    </init-param>
    <init-param>
```

```
            <param-name>.gif</param-name>
            <param-value>3600</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>expiresFilter</filter-name>
        <url-pattern>*.css</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>expiresFilter</filter-name>
        <url-pattern>*.jpg</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>expiresFilter</filter-name>
        <url-pattern>*.gif</url-pattern>
    </filter-mapping>
```

如果集成了 Apache 等 Web 服务器，可以更方便地使用 mod_expires 模块来配置静态资源的缓存时间，而不必编写 ExpiresFilter。

配置好 ExpiresFilter 以后，用 ieHTTPHeaders 插件查看浏览器发送的请求，可以看到，对于同一静态资源，IE 浏览器仅会请求一次。但是请注意，不要使用 IE 的“刷新”按钮，因为“刷新”按钮无论缓存是否过期都会直接请求服务器。可以打开另一个 IE，然后输入相同的地址来查看缓存是否起作用了。

11.13 设置站点首页

现在，我们已经配置好了所有的组件，站点已经可以运行了！输入 http://localhost:8080/listBooks.jspx，我们就看到了站点首页。不过，还有一个小问题，假定我们的站点已经部署到了服务器上，并且绑定了域名 www.livebookstore.net，用户访问时输入 http://www.livebookstore.net/会看到什么呢？一个 404 Not Found: / was not found on this server.错误。我们可不希望用户输入 http://www.livebookstore.net/listBooks. jspx 这么长的地址，因此，最后一步就是要配置一个 URL 映射，把“/”映射到“/listBooks.jspx”上就大功告成了！

可以在 web.xml 中配置<welcome-file-list>，包含一个或多个<welcome-file>，当用户请求“/”时，Web 服务器就会按照顺序查找指定的 welcome-file 文件，如果找到了，就直接返回给用户。例如：

```
    <welcome-file-list>
```

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

不幸的是，<welcome-file>不能设置为 Servlet 映射，不信试试。

```
<welcome-file-list>
    <welcome-file>listBooks.jspx</welcome-file>
</welcome-file-list>
```

保证不起作用。某些 JavaEE 服务器可以通过特殊配置实现“/”到 Servlet 的映射，但这并不是标准的解决方案，当部署到不同的 JavaEE 服务器上时会导致问题。

一个变通的方法是编写一个 welcome.jsp 文件充当 welcome-file，然后在 JSP 中直接调用 response 的 sendRedirect()方法跳转到 listBooks.jspx。

```
<%
    response.sendRedirect("listBooks.jspx");
%>
```

现在，将 welcome.jsp 加入到<welcome-file-list>中。

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

再次启动服务器，输入 http://localhost:8080/，浏览器终于正确显示出了站点首页，如图 11-34 所示。

图 11-34

注意：浏览器的地址栏“http://localhost:8080/listBooks.jspx”显示的是自动跳转后的地址。

11.14 和外部服务器集成

Resin 3.1 可以和第三方的 Web 服务器集成。本节我们讲述如何将 Resin 集成到 Apache 和 IIS 中。

11.14.1 集成到Apache

Resin 可以方便地与 Apache 集成，目前支持的版本是 Apache 2.0 和 Apache 2.2。我们以 Apache 2.0 为例，讲解在 Windows 下如何将 Resin 与 Apache 集成。

在 Resin 的根目录下找到/win32/apache-2.0/mod_caucho.dll，复制到 Apache 安装目录下的 modules 目录，然后编辑 Apache 的配置文件/conf/httpd.conf，修改如下。

```
# 添加 mod_caucho.dll:
LoadModule caucho_module modules/mod_caucho.dll
ResinConfigServer localhost 6800
<Location /caucho-status>
    SetHandler caucho-status
</Location>
...
# 将 DocumentRoot 指向 livebookstore 的 Web 目录:
DocumentRoot "D:/projects/livebookstore/web"
<Directory "D:/projects/livebookstore/web">
    Options Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
...
# 将 welcome.jsp 添加到默认首页文件:
DirectoryIndex welcome.jsp index.html index.html.var
```

先启动 Resin，然后启动 Apache，即可直接通过 http://localhost/访问 Live 在线书店。

11.14.2 集成到IIS

Resin 同样可以和 IIS 集成。Resin 提供了一个 ISAPI 筛选器，将 Resin 根目录下的/win32/isapi_srun.dll 复制到 IIS 的默认站点下的 scripts 虚拟目录下，然后在此创建配置文件 resin.ini，内容如下。

```
ResinConfigServer localhost 6800
CauchoStatus yes
IISPriority high
```

打开默认网站的属性页，切换到 ISAPI 筛选器，将 scripts 虚拟目录下的 isapi_srun.dll 添加为一个新的筛选器，如图 11-35 所示。

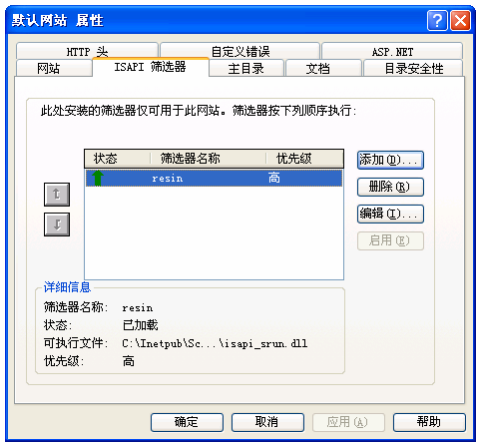


图 11-35

启动 Resin 服务器，然后重启 IIS 服务器，就可以通过 http://localhost/访问 Live 在线书店了。

如果出现“401.3 没有权限访问”的错误，请检查 scripts 目录和 isapi_srun.dll 文件是否具有 IIS 进程对应的 IUSER_XXX 用户的读权限，或者设置 Everyone 的权限为只读。

11.15 小结

本章我们通过一个完整的 Web 应用程序演示了如何利用 Spring 2.0 框架来构建灵活的、可扩展的 Web 应用程序。在 Live 在线书店应用中，我们严格按照了 JavaEE 多层架构模型，保证了各层之间相对独立，简化了每一层的设计和实现，并且使得后期的维护和扩展更加容易。对于每一层的所有组件，我们都是以 Bean 的形式配置在 Spring 的 IoC 容器中，这种依赖注入的组件装配模式非常强大，不仅最大限度地降低了各组件的耦合度，还大大简化了测试。

读者应当建立这样一个概念：在设计 Web 应用程序时，首先应该认识到，Web 应用程序和普通的桌面应用程序一样，也是应用程序的一种，绝不应当被简单地作为 Web 站点来设计。事实上，Web 站点的设计仅仅是一个完整的 Web 应用程序表示层 MVC 架构中的 View，而后台的持久层、逻辑层和前台 MVC 的实现都必须使用面向对象的基本设计方法，只有这样，才能够建立一个快速灵活且具有扩展性的 Web 应用程序。

在 Live 在线书店应用程序中，我们涉及了前面几章讲到的几乎所有的内容。

（1）使用 Spring 的 IoC 容器装配所有的组件，包括 Web 组件。

（2）使用 Hibernate 作为持久化方案并无缝集成到 Spring 中。

（3）使用泛型 DAO 实现类型安全的 DAO 设计。

（4）使用 Spring 的声明式事务管理，并通过 Java 5 注解简化事务配置。

（5）使用 Spring 的 MVC 框架设计灵活的 Web 层。

（6）使用 Spring 集成的 Velocity 实现视图，能方便地进行可视化的页面设计。

（7）使用 Acegi 实现安全控制，保护 Web 资源和逻辑业务组件。

（8）使用 Compass 实现基于 Lucene 的全文搜索。

（9）使用 Spring 集成的 JMS 和 Mail 组件方便地实现异步发送邮件。

（10）使用 JMX 并通过 AOP 实现性能监控。

（11）使用 Filter 实现了两种页面缓存，提升了系统性能。

（12）使用 Filter、GZip 压缩等方式最大限度地优化 HTTP 传输，减少不必要的请求。

（13）许多有用的模式和技巧，例如，用 JavaScript 分离页面的动态部分和静态部分。

通过这个完整的 Web 应用程序，有助于读者更好地应用 Spring 2.0 框架来构建自己的 JavaEE 应用程序。在开发过程中，我们还坚持使用测试驱动开发来保证代码质量，使用 Ant 脚本和 XDoclet 工具来构建项目，使所有的配置文件都能自动生成，大大减少了项目开发过程中维护这些配置文件的工作。

附录A　XDoclet参考

XDoclet 是一种模版工具，通过 XDoclet，可以非常容易地从源代码中提取注释，自动生成配置文件。目前，XDoclet 1.2.3 支持 Spring、Hibernate、JDO、EJB、Portlet 等多种配置文件的自动生成。

附录 A 仅介绍如何应用 XDoclet 自动为 Spring 2.0 项目生成 Spring 所需的 XML 配置文件。

A.1　创建带有@spring的注释

XDoclet 可以自动提取的所有注释必须被标记在 Java 源代码的标准注释中，形式如下。

```
/**
 * @spring.tag parameter1="value1" parameter2="value2" ...
 */
```

1．@spring.bean 见表 A-1

作用：定义一个 Bean，无需指定 class，XDoclet 将自动检测完整类名。

表 A-1

参数名称	描　　述	允许的设置	是否必须
id	定义 Bean 的 id	合法的 XML 标记字符	id 或 name 至少指定一个
name	定义 Bean 的 name	任意字符	
autowire	指定自动装配的方式，不指定默认为 no	no byName byType constructor autodetect	否
scope(*)	指定 Bean 的作用域，不指定默认为 singleton	singleton prototype request session globalSession	否

续表

参数名称	描　　述	允许的设置	是否必须
singleton	指定 Bean 是 singleton 还是 prototype，已不推荐使用，考虑用 scope 代替	True false	否
init-method	指定 Bean 的初始化方法	无参数的方法名	否
destroy-method	指定 Bean 的销毁方法	无参数的方法名	否
lazy-init	指定是否延迟初始化 Bean，不指定默认为 false	true false	否
dependency-check	指定是否检查依赖关系，不指定默认为 none	none simple object all	否
description	Bean 的描述	任意字符	否

请注意，标记有(*)的参数是自定义的（下同），需要自定义模版的支持，请参见后面的“自定义 XDoclet 模版”一节。

示例：

```
/**
 * @spring.bean id="myBean" scope="prototype" init-method="initDir"
 */
public class MyBean { ... }
```

2．@spring.property 见表 A-2

作用：定义一个依赖注入属性，需要标记在 set 方法前，无需指定属性名称，XDoclet 将自动检测属性名称。

表 A-2

参数名称	描　　述	允许的设置	是否必须
value	注入基本类型和 String、URL、Resource、Class 等类型	可以被转化为相应类型的字符串	至少指定其中一个
ref	注入一个 Bean 的引用	该 Bean 的 id	
list	注入由<value>类型构成的 List 或数组	可以被转化为相应类型的字符串，以逗号分隔	
listRef(*)	注入由<ref>类型构成的 List 或数组	Bean 的 id，以逗号分隔	
set(*)	注入由<value>类型构成的 Set	可以被转化为相应类型的字符串，以逗号分隔	
setRef(*)	注入由<ref>类型构成的 Set	Bean 的 id，以逗号分隔	

示例：

```
/**
 * @spring.property value="1234"
 */
public void setMaxAge(int age) { ... }
```

3．@spring.constructor-arg 见表 A-3

作用：定义一个构造方法的参数注入，每个标记对应一个参数，需要标记在构造方法前。

表 A-3

参数名称	描　　述	允许的设置	是否必须
value	注入基本类型和 String、URL、Resource、Class 等类型	可以被转化为相应类型的字符串	至少指定其中一个
ref	注入一个 Bean 的引用	该 Bean 的 id	
list	注入由<value>类型构成的List或数组	可以被转化为相应类型的字符串，以逗号分隔	
listRef(*)	注入由<ref>类型构成的 List 或数组	Bean 的 id，以逗号分隔	
type(*)	指定参数的类型，即完整类名	完整的类名或基本类型（如 int、float）	否

示例：

```
/**
 * @spring.constructor-arg value="1234" type="java.lang.String"
 * @spring.constructor-arg value="5678" type="int"
 * @spring.constructor-arg ref="anotherBean"
 */
public Constructor(String file, int size, AnotherBean bean) { ... }
```

A.2　自定义XDoclet模版

目前，XDoclet 1.2.3 仅支持 Spring 1.x 版本，要支持 Spring 2.0，必须使用自定义模版。可以在 XDoclet 提供的模版上修改，使之支持 Spring 2.0。本书使用的 XDoclet 模版是作者自己扩充的自定义模版，增加了一些新的参数类型，读者可以在配套光盘中找到自定义模版“custom_spring_xml.xdt”并直接使用。

A.3　创建Ant任务

XDoclet 只能以 Ant 的扩展任务形式执行，因此，需要在 build.xml 中加入任务。下面的 build.xml 仅给出一个 XDoclet 的任务，不包括其他（如 javac、jar 等）任务。请读者注意，这个例子的项目结构请参考第 3 章的 IoC_XDoclet 项目。

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

```
<project name="HelloWorld" default="gen-spring-conf" basedir=".">
    <-- 定义目录 -->
    <property name="src.dir"     value="src"     />
    <property name="lib.dir"     value="lib"     />
    <property name="conf.dir"    value="conf"    />
    <property name="template.dir" value="template" />

    <!-- 定义 Classpath -->
    <path id="master-classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
</path>

    <!-- 使用 XDoclet 生成配置文件 -->
    <target name="gen-spring-conf" depends="compile">
        <!-- 定义 Ant 任务 -->
        <taskdef name="springdoclet"
            classname="xdoclet.modules.spring.SpringDocletTask"
            classpathref="master-classpath"
        />
        <!-- 生成配置文件 -->
        <springdoclet
            destDir="${build.dir}"
            mergeDir="${conf.dir}"
            force="false"
            excludedTags="@version,@author,@todo"
        >
            <fileset dir="${src.dir}" includes="**/*.java" />
            <springxml
                xmlencoding="UTF-8"
                templateFile="${template.dir}/custom_spring_xml.xdt"
                destinationFile="config.xml"
            />
        </springdoclet>
    </target>
</project>
```

注意：使用<taskdef>在 build.xml 中定义 springdoclet 任务 ,就无需将 XDoclet 的所有 jar 文件复制到 Ant 的 lib 目录下，但是需要复制到${lib.dir}目录下，以便加入到 Classpath 中。

<springdoclet>就是自动生成 Spring XML 配置文件的任务，可以包含以下属性。

（1）destDir：自动生成的 XML 配置文件的存放目录。

（2）mergeDir：需要嵌入的静态配置文件的存放目录，要注意该文件必须是 spring-bean.xml 且仅包含<bean>的定义，不含 XML 文件头和根节点<beans>。如果没有找到 spring-bean.xml 文件，就不会包含静态配置的<bean>。

（3）force：是否每次都强制重新生成配置文件，若设为 false，如果没有检测到源文件的改动，就不会重新生成配置文件，若设为 true，则不管源文件是否改动过，每次都将重新生成配置文件。

（4）excludedTags：被忽略的注释类型，对于 springdoclet 没有效果，对于其他某些 XDoclet 任务会有效果，例如，ejbdoclet。

<fileset>是标准的 Ant 指令，指示要扫描哪些 Java 源文件，建议设定为${src.dir}目录下的所有.java 文件。

<springxml>定义了自动生成的配置文件的一些设置，包括：

（1）xmlencoding：设置自动生成的 XML 配置文件的编码，若不指定，默认为“UTF-8”。

（2）templateFile：设置使用哪个模版来生成配置文件，若不指定，将使用 XDoclet 自带的模版。强烈建议读者使用本书提供的修改后的模版，能支持 Spring 2.0 版本及更多的依赖注入特性。

（3）destinationFile：设置自动生成的目标文件名，若不指定，默认为“spring.xml”。

（4）defaultAutowire：设置根节点<beans>的 default-autowire 属性，若不指定，默认为“no”。

（5）defaultLazyInit：设置根节点<beans>的 default-lazy-init 属性，若不指定，默认为“false”。

（6）defaultDependencyCheck：设置根节点<beans>的 default-dependency-check 属性，若不指定，默认为“none”。

A.4 XDoclet资源

XDoclet 站点：http://xdoclet.sourceforge.net

XDoclet2 站点：http://xdoclet.codehaus.org

Ant 参考：http://ant.apache.org/manual/index.html

附录B　Java Persistent API注解

JPA 注解作为 EJB 3 标准的一部分，用于为 ORM 框架提供基于注解的配置方式，但 JPA 本身是独立于 EJB 3 的，可以被任何 ORM 框架利用。附录 B 列举了常用的 JPA 注解，能满足大多数实际项目的需求。

B.1　创建Java Persistent API注解

1．@Entity(name="EntityName")

必需

@Entity 用于标注一个实体，通常，该实体类型将对应数据库中的一个表。@Entity 只能标注在实体的 class 定义处。

name：可选，表示实体名称。实体名称将用于查询中，如 Hibernate 的 HQL 查询。默认的，实体名称和类名一致（不包含 package 名称）。

示例：

```
@Entity(name="Book")
public class Book {}
```

2．@Table(name="", catalog="", schema="")

可选

@Table 通常和@Entity 配合使用，只能标注在实体的 class 定义处，表示实体对应的数据库表的信息。

name：可选，表示表的名称。默认地，表名和实体名称一致，只有在不一致的情况下才需要指定表名。

catalog：可选，表示 Catalog 名称，默认为 Catalog（""）。

schema：可选，表示 Schema 名称，默认为 Schema（""）。

示例：

```
@Entity(name="Book")
@Table(name="T_BOOK")
public class Book {}
```

3．@Id

必需

@Id 定义了映射到数据库表的主键的属性。一个实体只能有一个属性被映射为主键。

示例：

```
@Id
public int getPk() { return pk; }
```

4．@GeneratedValue(strategy=GenerationType, generator="")

可选

@GeneratedValue 和@Id 配合使用，表示主键的生成策略。

strategy：表示主键生成策略，有 AUTO、IDENTITY、SEQUENCE 和 TABLE 4 种，分别表示让 ORM 框架自动选择，根据数据库表的 Identity 字段生成，根据数据库表的 Sequence 字段生成，以及根据一个额外的表生成主键，默认为 AUTO。

generator：表示主键生成器的名称，这个属性通常和 ORM 框架相关。例如，Hibernate 可以指定 uuid 等主键生成方式。

示例：

```
@Id
@GeneratedValue(strategy=StrategyType.SEQUENCE)
public int getPk() { return pk; }
```

5．@Basic(fetch=FetchType, optional=true)

可选

@Basic 表示一个简单的属性到数据库表的字段的映射，对于没有任何标注的 getX()方法，默认即为@Basic。

fetch：表示该属性的读取策略，有 EAGER 和 LAZY 两种，分别表示主动抓取和延迟加载，默认为 EAGER。

optional：表示该属性是否允许为 null，默认为 true。

示例：

```
@Basic(optional=false)
public String getAddress() { return address; }
```

6．@Column

可选

@Column 描述了数据库表中该字段的详细定义。这对于根据 JPA 注解生成表结构的工具非常有用。

name：表示数据库表中该字段的名称，默认与属性名称一致。

nullable：表示该字段是否允许为 null，默认为 true。

unique：表示该字段是否是唯一标识，默认为 false。

length：表示该字段的大小，仅对 String 类型的字段有效。

insertable：表示在 ORM 框架执行插入操作时，该字段是否应出现在 INSERT 语句中，默认为 true。

updatable：表示在 ORM 框架执行更新操作时，该字段是否应出现在 UPDATE 语句中，默认为 true。对于一经创建就不可更改的非主键字段，该属性非常有用。例如，对于 birthday 字段。

columnDefinition：表示该字段在数据库中的实际类型。通常 ORM 框架可以根据属性类型自动判断数据库中字段的类型，但是对于 Date 类型仍无法确定数据库中字段类型究竟是 DATE、TIME 还是 TIMESTAMP。此外，String 的默认映射类型为 VARCHAR，如果要将 String 类型映射到特定数据库的 BLOB 或 TEXT 字段类型，该属性十分有用。

示例：

```
@Column(name="BIRTH", nullable=false, columnDefinition="DATE")
public String getBirthday() { return birthday; }
```

7．@Transient

可选

@Transient 表示该属性并非一个到数据库表的字段的映射，ORM 框架将忽略该属性。如果一个属性并非数据库表的字段映射，就务必将其标识为@Transient，否则，ORM 框架默认其注解为@Basic。

示例：

```
// 根据 birth 计算出的 age 属性:
@Transient
public int getAge() { return getYear(new Date()) - getYear(birth); }
```

8．@ManyToOne(fetch=FetchType, cascade=CascadeType)

可选

@ManyToOne 表示一个多对一的映射，该注解标注的属性通常是数据库表的外键。

optional：是否允许该字段为 null，该属性应根据数据库表的外键约束来确定，默认为 true。

fetch：表示抓取策略，默认为 FetchType.EAGER。

cascade：表示默认的级联操作策略，可以指定为 ALL、PERSIST、MERGE、REFRESH 和 REMOVE 中的若干组合，默认为无级联操作。

targetEntity：表示该属性关联的实体类型。该属性通常不必指定，ORM 框架根据属

性类型自动判断 targetEntity。

示例：

```
// 订单 Order 和用户 User 是一个 ManyToOne 的关系
// 在 Order 类中定义：
@ManyToOne()
@JoinColumn(name="USER")
public User getUser() { return user; }
```

9．@JoinColumn

可选

@JoinColumn 和@Column 类似，但是描述的不是一个简单字段，而是一个关联字段。例如，描述一个@ManyToOne 的字段。

Name：该字段的名称。由于@JoinColumn 描述的是一个关联字段，如 ManyToOne，则默认的名称由其关联的实体决定。例如，实体 Order 有一个 user 属性来关联实体 User，则 Order 的 user 属性应为一个外键，其默认的名称为实体 User 的名称+下划线+实体 User 的主键名称。

示例：

见@ManyToOne。

10．@OneToMany(fetch=FetchType, cascade=CascadeType)

可选

@OneToMany 描述一个一对多的关联。该属性应当为集合类型，在数据库表中并没有实际字段。

fetch：表示抓取策略，默认为 FetchType.LAZY，因为其关联的多个对象通常不必从数据库预先读取到内存。

cascade：表示级联操作策略，对于 OneToMany 类型的关联非常重要，通常当该实体更新或删除时，其关联的实体也应当被更新或删除。例如，实体 User 和实体 Order 是 OneToMany 关系，则实体 User 被删除时，其关联的实体 Order 也应当被全部删除。

示例：

```
@OneToMany(cascade=ALL)
public Set getOrders() { return orders; }
```

11．@OneToOne(fetch=FetchType, cascade=CascadeType)

可选

@OneToOne 描述一个一对一的关联。

fetch：表示抓取策略，默认为 FetchType.LAZY。

cascade：表示级联操作策略。

示例：

```
@OneToOne(fetch=FetchType.LAZY)
public Blog getBlog() { return blog; }
```

12．@ManyToMany

可选

@ManyToMany 描述一个多对多的关联。多对多关联实际上是两个一对多关联，但是在 ManyToMany 描述中，中间表由 ORM 框架自动处理。

targetEntity：表示多对多关联的另一个实体类的全名，例如，package.Book.class。

mappedBy：表示多对多关联的另一个实体类的对应集合属性名称。

示例：

User 实体代表用户，Book 实体代表书籍，为了描述用户收藏的书籍，可以在 User 和 Book 之间建立 ManyToMany 关系。

```
@Entity
public class User {
    private List books;
    @ManyToMany(targetEntity=package.Book.class)
    public List getBooks() { return books; }
    public void setBooks(List books) { this.books = books; }
}

@Entity
public class Book {
    private List users;
    @ManyToMany(targetEntity=package.User.class, mappedBy="books")
    public List getUsers() { return users; }
    public void setUsers(List users) { this.users = users; }
}
```

两个实体间相互关联的属性必须标记为@ManyToMany，并相互指定 targetEntity 属性。需要注意的是，有且只能有一个实体的@ManyToMany 注解需要指定 mappedBy 属性，指向 targetEntity 的集合属性名称。

利用 ORM 工具自动生成的表除了 User 和 Book 表外，还自动生成了一个 User_Book 表，用于实现多对多关联。

13．@MappedSuperclass

可选

@MappedSuperclass 可以将超类的 JPA 注解传递给子类，使子类能够继承超类的 JPA

注解。

示例：

```
@MappedSuperclass
public class Employee { ... }

@Entity
public class Engineer extends Employee { ... }

@Entity
public class Manager extends Employee { ... }
```

14. @ Embedded

可选

@Embeded 将几个字段组合成一个类，并作为整个 Entity 的一个属性。例如，User 表包含 id、name、city、street、zip 属性，我们希望将 city、street、zip 属性映射为 Address 对象，这样，User 对象将具有 id、name 和 address 这 3 个属性。Addess 对象必须被定义为@Embeddable。

示例：

```
@Embeddable
public class Address { city, street, zip }

@Entity
public class User {
    @Embedded
    public Address getAddress() { ... }
    ...
}
```

附录C　光盘资源索引

C.1　flash目录

eclipse_introduction.exe：介绍 Eclipse 的 Flash 入门教程。

C.2　software目录

包含本书涉及的相关软件和开源框架。

C.3　source目录

包含本书介绍的全部源代码，均为完整的 Eclipse 3.2 版本的工程。

（1）chapter2～chapter10 目录：分别包含第 2 章~第 10 章的全部源代码。

（2）livebookstore 目录：Live 在线书店应用程序的全部源代码。

Broadview®
www.broadview.com.cn

《Spring 2.0 核心技术与最佳实践》读者调查表

尊敬的读者：

感谢您对我们的支持与爱护。为了今后为您提供更优秀的图书，请您抽出宝贵的时间将您的意见以下表的方式及时告知我们（可另附页）。我们将从中评选出热心读者若干名，免费赠阅我们以后出版的图书。

姓名：	性别：□ 男 □ 女	年龄：	职业：
通信地址：			邮政编码：
电话：	传真：	E-mail：	

1. 影响您购买本书的因素（可多选）：

□ 封面封底 □ 价格 □ 内容提要、前言和目录 □ 书评广告 □ 出版物名声
□ 作者名声 □ 正文内容 □ 其他_____

2. 您对本书的满意度：

从技术角度 □ 很满意 □ 比较满意 □ 一般 □ 较不满意 □ 不满意
□ 改进意见_____

从文字角度 □ 很满意 □ 比较满意 □ 一般 □ 较不满意 □ 不满意
□ 改进意见_____

从版面、封面设计角度 □ 很满意 □ 比较满意 □ 一般 □ 较不满意
□ 不满意 □ 改进意见_____

3. 您最喜欢书中的哪篇（或章、节）？请说明理由。

_____

_____

4. 您最不喜欢书中的哪篇（或章、节）？请说明理由。

_____

_____

5. 您希望本书在哪些方面进行改进？

_____

_____

6. 您感兴趣或希望增加的图书选题有：

_____

_____

Broadview®
www.broadview.com.cn
通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888

如果您对我们出版的图书有任何意见和建议，也可以发邮件给我们，我们将及时回复。

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

反侵权盗版声明