精通Spring 4.x
企业应用开发实战

Proficient in 4.x Spring
Enterprise Application Development

陈雄华 林开雄 文建国◎编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

陈雄华

毕业于厦门大学计算机与信息工程学院，倾心Spring技术研究多年，是ROP开源项目创始人，担任过多家公司的系统架构师、技术总监，主持过多个大型企业级应用及多家公司基础技术平台的研发，拥有丰富的一线实战经验。出版《精通JBuilder 2005》、《精通Spring 3.x》、《Spring就这么简单》等多本技术书籍，广受读者好评。

林开雄

资深软件开发经理，拥有10余年软件开发经验，对Spring、大数据、应用虚拟化、微服务等开源技术的应用和实现原理有深入研究，拥有丰富的产品研发实战经验，目前专注于大数据解决方案以及微服务的研究与实施，参与《精通Spring 3.x》、《Spring就这么简单》等多本技术书籍的创作。

文建国

系统架构设计师、高级项目经理，精通Spring等优秀开源技术在企业中的应用，主要研究方向为云计算、大数据、业务基础平台、分布式等技术。曾参与中国电信ITSP 3.0技术架构规范的编写，有多个大型全国集中项目的架构和管理经验。目前致力于“智能制造”的人机一体化系统研发，希望通过物联网与互联网的融合提升生产效率和增强企业原有系统。热衷于开源技术布道，曾译有《Spring Data实战》、《大规模Java平台虚拟化与调优》等书籍。

精通 Spring 4.x
企业应用开发实战

陈雄华　林开雄　文建国◎编著

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

内 容 简 介

Spring 4.0 是 Spring 在积蓄 4 年后，隆重推出的一个重大升级版本，进一步加强了 Spring 作为 Java 领域第一开源平台的翘楚地位。

Spring 4.0 引入了众多 Java 开发者翘首以盼的基于 Groovy Bean 的配置、HTML 5/WebSocket 支持等新功能，全面支持 Java 8.0，最低要求是 Java 6.0。这些新功能实用性强、易用性高，可大幅降低 Java 应用，特别是 Java Web 应用开发的难度，同时有效提升应用开发的优雅性。

本书是在《精通 Spring 3.x——企业应用开发详解》的基础上，历时一年的重大调整改版而成的，延续了上一版本“追求深度，注重原理，不停留在技术表面”的写作风格，力求使读者在熟练使用 Spring 的各项功能的同时透彻理解 Spring 的内部实现，真正做到知其然并知其所以然。此外，本书重点突出了“实战性”的主题，力求使全书内容体现“从实际项目中来，到实际项目中去”的写作原则。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

精通 Spring 4.x：企业应用开发实战 / 陈雄华，林开雄，文建国编著. —北京：电子工业出版社，2017.1

ISBN 978-7-121-30443-9

Ⅰ. ①精… Ⅱ. ①陈… ②林… ③文… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 284564 号

责任编辑：李　冰
特约编辑：田学清　赵海军等
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：51.25　　字数：1312 千字
版　　次：2017 年 1 月第 1 版
印　　次：2017 年 1 月第 1 次印刷
印　　数：3000 册　　定价：128.00 元

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）88254888，88258888。

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

本书咨询联系方式：libing@phei.com.cn。

前　言

本书小序

Spring 从 2004 年发布第一个版本以来，至今已有 12 载。12 年刚好是一个生肖轮回，但在一日千里的计算机领域，12 年基本上算是一个世纪了。都说“好花不常开，好景不常在”，但 Spring 这朵 Java 开源世界里芳香馥郁的奇葩不但没有零落成泥，反而满园春色历久弥艳，成为 Java 开发者无法回避的开源框架。

回顾 Spring 的光辉岁月，一路与时俱进，引领时代之潮流。总的来说，Spring 主要经历了三次重大的版本升级：一为 2006 年从 1.0 升级到 2.0，在 Spring 2.0 中新增了 XML 命名空间、AspectJ 及 Spring MVC 等功能，此外，在 Spring 2.5 中还引入了注解驱动配置的支持，同时进一步完善了 Spring MVC 功能；二为 2009 年从 2.5 升级到 3.0，新增了 SpEL、OXM、REST、验证/格式化等功能，全面支持 Java 5.0；三为 2013 年从 3.0 升级到 4.0，新增了 Groovy Bean 配置、HTML 5/WebSocket 支持等功能，全面支持 Java 8.0，最低要求是 Java 6.0。Spring 始终坚持以小版本快速推进、每三年左右发布一个大版本的演化策略，既保证版本的平稳有序，又能紧跟技术发展的潮流。难能可贵的是，Spring 即便发生了这么多次版本的升级，其整体框架依然是向下兼容的，在这一点上，Spring 明显区别于 Struts、Hibernate 等框架的升级风格。

笔者在 2007 年曾编写了《精通 Spring 2.x》，并在 2012 年出版了升级版本《精通 Spring 3.x》。感谢读者朋友的厚爱垂青，其中《精通 Spring 3.x》已经重印了 11 次，成为国内 Spring 领域的畅销书籍。2013 年年底 Spring 4.0 就已经发布了，从那时起，出版社的朋友多次力促笔者进行版本的同步升级，笔者也希望能与时俱进地对原书进行更新，但囿于工作繁忙，一直未能付诸行动。直到 2015 年 8 月左右，才与林开雄、文建国着手筹划《精通 Spring 4.x》的升级编写工作。林开雄和文建国是笔者多年的好朋友，二人都是拥有十多年工作经验的 Java 实战型技术高手，他们不但为人谦和、技术精湛，而且拥有丰厚的创作实力。林开雄早在 2012 年就参与了《精通 Spring 3.x》的撰写工作，文建国则于 2014 年翻译了《Spring Data 实战》。此外，林莉、何彩云、陈谋坤、项群、陈文炎、陈曦、康玉琳、蔡雪峰、康沿清、朱景、朱贤俊等也一起参与了本书的代码审查及测试工作，在此对大家的努力付出一并表示感谢！

本次改版，不但将全书内容同步更新到 Spring 4.0，还对全书结构进行了多方面的优化和调整。移除了两个章节，分别是“JavaMail 发送邮件”（考虑到比较简单）及“在 Spring 中开发 Web Service”（考虑到 Spring MVC 开发 REST WebService 再合适不过了）；新增了 3 个全新的章节，分别是“Spring Boot”、“Spring SpEL”及“Spring Cache”。

本书特点

- **揭示内幕，深入浅出：**笔者对 Spring 的源码进行了彻底分析，深刻揭示了 Spring 框架的技术内幕，让读者知其然，更知其所以然。Spring 中的许多设计经验、技巧、模式具有很高的借鉴性，在透彻学习 Spring 体系结构的同时，读者可以直接将这些方法借用到具体的应用开发中。
- **同步更新，与时俱进：**虽然在 2013 年 12 月就发布 Spring 4.0 的第一个候选版本，后来又发布了多个 RC 版本，并最终于 2015 年 8 月发布了 Spring 4.2 的正式版本，但新功能的添加及旧功能的调整从来就没有停止过。本书基于 Spring 4.2 版本讲解，保证全书内容与时俱进。
- **突出重点，淡化边缘：**虽然全书篇幅达 800 多页，但本书没有片面追求内容的面面俱到，相反，我们特别注意内容的剪裁和取舍；对于实用性强的知识点深入分析、深度挖掘，而对于不常用的知识点则点到为止，甚至不纳入本书的范围。举例来说，我们对使用 Spring JDBC、Spring Cache 及 Spring MVC 等实用性强的技术都进行了深入分析，而对如何集成 EJB、JMX、JCA 等不常用的功能完全不涉及，很好地做到了实用性和深入性的统一。
- **理论透彻，面向实践：**本书在透彻分析原理、讲解技术知识点的同时，特别注意与实际应用的结合。笔者将自身丰富的实战经验糅合到全书的相关知识点中，很好地做到了知识讲解和实践经验的结合，让读者在掌握纯技术知识的同时能够对如何活用技术做到胸有成竹。如笔者在第 16 章讲解任务调度的内容时，专门辟出 16.6 节讲解实际应用中任务调度的使用经验；在第 18 章中讲述使用实战项目开发时，专门通过 18.11 节讲述了笔者在实际项目中所总结的项目配置文件及数据源的规划方案。此外，我们还适时提供了“实战经验”的插文，它们在不影响上下文连贯性的同时让读者学到了相关技术的实战经验。诸如此类的以实际应用为导向的内容贯穿全书，这是本书区别于其他书籍的特色之一。
- **代码简洁，图例丰富：**全书代码在排版布局及内容剪裁上颇费心思，实例代码重点关注当前知识点涉及的内容，弱化边缘代码，并采用特殊的排版方式适时添加简明扼要的注释，方便程序代码的阅读和重点内容的把握。全书拥有大量精美的图表，这些图表很好地解构了上下文中的一些知识难点，大大提高了可读性，降低了理解的难度。
- **注重趣味，轻松阅读：**由于技术书籍的严谨性、知识性特点，阅读技术书籍往往是枯燥乏味的，更遑论趣味性。笔者对此深有感触，为寻求一些突破，我们

在全书大部分章节都精心设计了一个“轻松一刻”环节。它们和上下文内容存在某种程度的关联性，但其本身是一段趣味性的短文，以便在增强全书趣味性的同时还为读者提供了另一个思考问题的角度。

- **相关知识，一网打尽：**Spring 不但本身涉及众多 Java 技术，其集成的第三方技术本身也涵盖了丰富的知识。我们在介绍 Spring 相关技术时，都会简明扼要地讲解相关联的基础知识，其中包括 Java 的新知识和被集成技术的知识，而不是在完全脱离背景知识的情况下孤立讲解 Spring 的知识。
- **通力合作，倾力打造：**本书从筹划到改版完成，历时近 8 个月。我们交叉审核，多次优化重构，为保证全书质量，多次向出版社申请延迟提交稿件，直到 2016 年 6 月才完成所有稿件。

本书结构

本书分为 5 篇。第 1 篇为基础篇，包括第 1～3 章，讲解 Spring 概述性知识，以便读者快速建立对 Spring 的整体认识，能够使用 Spring 快速开发一个简单的项目，而更多深入的知识则在后续篇章中展开。第 2 篇为核心篇，包括第 4～9 章，讲解 Spring 的 IoC、AOP 及 SpEL 的知识，这些知识是 Spring 的核心，也是 Spring 所有衍生服务及功能的基石。第 3 篇为数据篇，包括第 10～14 章，讲解 Spring 的各种数据访问技术及事务管理的内容，对事务管理的实现机制和各种疑难问题进行剖析。第 4 篇为应用篇，包括第 15～18 章，讲解数据缓存、任务调度、Web 开发的内容，对于企业应用开发来说，数据缓存及任务调度是两个无法回避的问题，需要重点学习和掌握；此外，本篇还精心设计了一个实战案例，包含需求分析、数据库设计、项目开发、代码测试、应用部署的整体过程，让读者在项目的实战中整体串接 Spring 的知识点。第 5 篇为提高篇，包括第 19 章和第 20 章，讲解 Spring OXM 及单元测试的内容；全书工程代码放在本书配套网盘中，请读者下载网盘内容阅读，本书网盘下载地址如下。

百度云盘：http://pan.baidu.com/s/1boCl3d1。

下面简要介绍一下每章的内容。

第 1 章：对 Spring 框架进行了整体性的概述，可使读者快速建立起对 Spring 整体性的认识。

第 2 章：通过一个简单的例子展现开发 Spring Web 应用的整体过程，通过这个实例，读者可以快速跨入 Spring Web 应用的世界。

第 3 章：Spring Boot 的设计目的是用来简化新 Spring 应用的搭建和开发过程，本章通过实例向读者讲述了 Spring Boot 的使用技巧。

第 4 章：讲解了 Spring IoC 容器的知识，通过具体的实例详细地讲解了 IoC 概念；同时，对 Spring 框架的三个最重要的框架级接口进行了剖析，并对 Bean 的生命周期进行了讲解。

第 5 章：讲解了如何在 Spring 配置文件中使用各种配置方式配置 Bean 的内容，并对各个配置项的意义进行了深入说明。

第 6 章：对 Spring 容器进行了解构，从内部探究 Spring 容器的体系结构和运行流程。此外，还对 Spring 容器一些高级主题进行了深入阐述。

第 7 章：从 Spring AOP 的底层实现技术入手，一步步深入到 Spring AOP 的内核中，分析它的底层结构和具体实现。

第 8 章：对如何使用基于 AspectJ 配置 AOP 的知识进行了深入分析，包括使用 XML Schema 配置文件、使用注解进行配置等内容。

第 9 章：SpEL 不仅仅是一个动态语言，而且 Spring 容器的很多配置都直接依赖于 SpEL 工作，因此掌握 SpEL 是掌握 Spring 配置的必修课程。

第 10 章：介绍了 Spring 所提供的 DAO 封装层，包括 Spring DAO 的异常体系、数据访问模板等内容。

第 11 章：声明式事务配置是 Spring 的一项重要功能，本章介绍了 Spring 事务管理的工作机制，以及通过 XML、注解等方式进行事务管理配置等内容。

第 12 章：对实际应用中 Spring 事务管理的各种疑难问题进行了透彻剖析，让读者对 Spring 事务管理不再有云遮雾罩的感觉。

第 13 章：讲解了如何使用 Spring JDBC 进行数据访问操作，还重点讲述了 LOB 字段处理、主键的产生和获取等难点知识。

第 14 章：讲解了如何在 Spring 中集成 Hibernate、MyBatis 等数据访问框架，同时，读者还将学到 ORM 框架的混用和 DAO 层设计的知识。

第 15 章：数据缓存已经成为提高系统运行性能的一个重要方法，本章讲解了 Spring Cache 如何通过注解方式进行透明化数据缓存。

第 16 章：本章重点讲解了在 Spring 中如何使用 Quartz 进行任务调度，同时还涉及使用 JDK Timer 和 JDK 5.0 执行器等知识。

第 17 章：对 Spring MVC 框架进行了详细介绍，对 REST 风格的编程方式进行了重点讲解，同时还对 Spring 的校验和格式化框架如何与 Spring MVC 整合进行了说明。

第 18 章：以一个实际的项目为蓝本，带领读者从项目需求分析、项目设计、代码开发、单元测试直到应用部署体验一次接近实战的整体项目开发过程。

第 19 章：介绍 Spring OXM 的多种实现技术，同时对 XML 技术进行了整体说明。

第 20 章：有别于一般书籍的单元测试内容，本书以当前最具实战性的 TestNG+ Unitils+ Mockito 复合测试框架对测试基于数据库的 Web 应用进行了深入讲解。

如何使用本书

由于程序开发是一项实践性极强的工作，只有亲身体验才能掌握其真谛，因此我们强烈建议读者使用本书网盘的工程代码进行同步学习。本书项目基于 Java 7.0+，请

读者先安装好 Java SDK 7.0+；开发工具使用 Intllij IDEA，可从 http://www.jetbrains.com/idea 下载免费社区版进行安装。请下载本书配套网盘的内容，建议下载到本地的 D:\masterspring 目录下。本书所有章节对应的工程项目均以 Maven 方式组织，**请将网盘 tools\settings.xml 复制到 C:\Users\<操作系统用户名>\.m2 目录下**。在 settings.xml 中，我们使用了国内的 oschina Maven 公共仓库，下载依赖构件包速度很快；否则，Maven 项目将默认从国外的中央仓库下载，速度很慢。

当学习到某个章节时，请在 IDEA 中打开对应章节的工程项目，并依照下面的步骤创建章节所对应的 IDEA 工程项目：

（1）依次选择 File→New→Project From Existing Sources...命令，打开 Select File or Directory to Import 对话框。

（2）选择要打开的 D:\masterspring\chapterX 项目工程，打开 Import Project 向导。

（3）选择 Import project from external model 选项并在列表中选择 Maven，然后连续单击 Next 按钮，按向导指示创建章节的 Maven 项目工程（注意 SDK 选择 1.7）。

（4）由于本书工程项目采用 UTF-8 编码，所以在创建完项目工程后，**请按 Ctrl+Alt+S 组合键打开 Setting 对话框，找到 Editor→File Encodings，将 IDE Encoding、Project Encoding 及 Default encoding for properties files 三项都设置为 UTF-8**；否则将会因编码问题导致编译失败。

本书很多章节都用到了数据库，因此请在本机安装 MySQL 5.0+，本书假设 MySQL 的 root 密码为 123456。在每个需要数据库访问的项目工程中都拥有一个 schema 目录，其下是数据库初始化 SQL 脚本，请先执行脚本再运行章节对应的代码。

本书插文

本书会适时加入一些提示、实战经验和轻松一刻的小段插文，在不打断行文连续性的同时提供一些有益的开发经验、使用技巧并增强阅读的趣味性。这些插文都带有一个小图标加以凸显，说明如下。

	提示：在上下文中可能存在一些读者容易忽视或容易犯错的地方，在提示信息中给予针对性的帮助信息
	实战经验：笔者将多年的开发实战经验适时介绍给大家。这些知识往往是不能从一般的书籍或资料中获得的。本书会适时地在行文中将这些实战经验分享出来，相信可以使读者少走一些弯路
	轻松一刻：为了增强技术书籍阅读的趣味性，全书每章都有一到两个“轻松一刻”的短文。它们和上下文内容都存在某种程度的关联性，不但可为阅读带来了趣味性，还可以启发读者思考

此外，由于 Spring 4.x 拥有多个版本，为了保持行文的简洁，除非特别指出，本书的 Spring 或 Spring 4.0 即代表当前的最新版本（Spring 4.2.x）。

如何与作者联系

由于 Spring 内容涵盖面宽广，涉及的内容非常多，同时由于作者水平有限，错误之处在所难免。我们不但欢迎读者朋友来信交流，更期待各界高手、专家就不足之处给予赐教和斧正，您可以通过 itstamen@qq.com 与笔者联系。

陈雄华

2016 年 5 月 22 日于厦门

目　录

第 1 篇　基础篇

第 1 章　Spring 概述 2

1.1　认识 Spring 2

1.2　关于 SpringSource 4

1.3　Spring 带给我们什么 5

1.4　Spring 体系结构 6

1.5　Spring 对 Java 版本的要求 8

1.6　Spring 4.0 新特性 8

1.6.1　全面支持 Java 8.0 9

1.6.2　核心容器的增强 11

1.6.3　支持用 Groovy 定义 Bean 12

1.6.4　Web 的增强 12

1.6.5　支持 WebSocket 12

1.6.6　测试的增强 13

1.6.7　其他 13

1.7　Spring 子项目 13

1.8　如何获取 Spring 15

1.9　小结 16

第 2 章　快速入门 17

2.1　实例概述 17

2.1.1　比 Hello World 更适用的实例 18

2.1.2　实例功能简介 18

2.2　环境准备 20

2.2.1　构建工具 Maven 20

2.2.2　创建库表 22

2.2.3　建立工程 23

2.2.4　类包及 Spring 配置文件规划 28

2.3　持久层 29

2.3.1　建立领域对象 29

2.3.2　UserDao 30

2.3.3　LoginLogDao 33

2.3.4　在 Spring 中装配 DAO 34

2.4　业务层 35

2.4.1　UserService 35

2.4.2　在 Spring 中装配 Service 37

2.4.3　单元测试 38

2.5　展现层 40

2.5.1　配置 Spring MVC 框架 40

2.5.2　处理登录请求 42

2.5.3　JSP 视图页面 44

2.6　运行 Web 应用 46

2.7　小结 48

第 3 章　Spring Boot 49

3.1　Spring Boot 概览 49

3.1.1　Spring Boot 发展背景 50

3.1.2 Spring Boot 特点 50
3.1.3 Spring Boot 启动器 50
3.2 快速入门 52
3.3 安装配置 54
3.3.1 基于 Maven 环境配置 54
3.3.2 基于 Gradle 环境配置 56
3.3.3 基于 Spring Boot CLI 环境配置 57
3.3.4 代码包结构规划 58
3.4 持久层 59
3.4.1 初始化配置 59
3.4.2 UserDao 61
3.5 业务层 62
3.6 展现层 64
3.6.1 配置 pom.xml 依赖 64
3.6.2 配置 Spring MVC 框架 65
3.6.3 处理登录请求 65
3.7 运维支持 67
3.8 小结 70

第 2 篇 核心篇

第 4 章 IoC 容器 72
4.1 IoC 概述 72
4.1.1 通过实例理解 IoC 的概念 73
4.1.2 IoC 的类型 75
4.1.3 通过容器完成依赖关系的注入 77
4.2 相关 Java 基础知识 78
4.2.1 简单实例 78
4.2.2 类装载器 ClassLoader 80
4.2.3 Java 反射机制 83
4.3 资源访问利器 85
4.3.1 资源抽象接口 85
4.3.2 资源加载 88
4.4 BeanFactory 和 ApplicationContext 91
4.4.1 BeanFactory 介绍 92
4.4.2 ApplicationContext 介绍 94
4.4.3 父子容器 103
4.5 Bean 的生命周期 103
4.5.1 BeanFactory 中 Bean 的生命周期 103
4.5.2 ApplicationContext 中 Bean 的生命周期 112
4.6 小结 114
第 5 章 在 IoC 容器中装配 Bean 115
5.1 Spring 配置概述 116
5.1.1 Spring 容器高层视图 116
5.1.2 基于 XML 的配置 117
5.2 Bean 基本配置 120
5.2.1 装配一个 Bean 120
5.2.2 Bean 的命名 120
5.3 依赖注入 121
5.3.1 属性注入 121
5.3.2 构造函数注入 124
5.3.3 工厂方法注入 128
5.3.4 选择注入方式的考量 130
5.4 注入参数详解 130
5.4.1 字面值 130
5.4.2 引用其他 Bean 131
5.4.3 内部 Bean 133
5.4.4 null 值 133
5.4.5 级联属性 134

5.4.6 集合类型属性.....134
5.4.7 简化配置方式.....138
5.4.8 自动装配.....141
5.5 方法注入.....142
5.5.1 lookup 方法注入.....142
5.5.2 方法替换.....143
5.6 <bean>之间的关系.....144
5.6.1 继承.....144
5.6.2 依赖.....145
5.6.3 引用.....146
5.7 整合多个配置文件.....147
5.8 Bean 作用域.....148
5.8.1 singleton 作用域.....148
5.8.2 prototype 作用域.....149
5.8.3 与 Web 应用环境相关的 Bean 作用域.....150
5.8.4 作用域依赖问题.....152
5.9 FactoryBean.....153
5.10 基于注解的配置.....155
5.10.1 使用注解定义 Bean.....155
5.10.2 扫描注解定义的 Bean.....156
5.10.3 自动装配 Bean.....157
5.10.4 Bean 作用范围及生命过程方法.....162
5.11 基于 Java 类的配置.....164
5.11.1 使用 Java 类提供 Bean 定义信息.....164
5.11.2 使用基于 Java 类的配置信息启动 Spring 容器.....167
5.12 基于 Groovy DSL 的配置.....169
5.12.1 使用 Groovy DSL 提供 Bean 定义信息.....169
5.12.2 使用 GenericGroovyApplicationContext 启动 Spring 容器.....171
5.13 通过编码方式动态添加 Bean.....172
5.13.1 通过 DefaultListableBeanFactory.....172
5.13.2 扩展自定义标签.....173
5.14 不同配置方式比较.....175
5.15 小结.....177
第 6 章 Spring 容器高级主题.....178
6.1 Spring 容器技术内幕.....178
6.1.1 内部工作机制.....179
6.1.2 BeanDefinition.....182
6.1.3 InstantiationStrategy.....183
6.1.4 BeanWrapper.....183
6.2 属性编辑器.....184
6.2.1 JavaBean 的编辑器.....185
6.2.2 Spring 默认属性编辑器.....188
6.2.3 自定义属性编辑器.....189
6.3 使用外部属性文件.....192
6.3.1 PropertyPlaceholderConfigurer 属性文件.....192
6.3.2 使用加密的属性文件.....195
6.3.3 属性文件自身的引用.....198
6.4 引用 Bean 的属性值.....199
6.5 国际化信息.....201
6.5.1 基础知识.....201
6.5.2 MessageSource.....206
6.5.3 容器级的国际化信息资源.....209
6.6 容器事件.....210
6.6.1 Spring 事件类结构.....211
6.6.2 解构 Spring 事件体系的具体实现.....213

6.6.3 一个实例 214
6.7 小结 215
第 7 章 Spring AOP 基础 216
7.1 AOP 概述 216
7.1.1 AOP 到底是什么 217
7.1.2 AOP 术语 219
7.1.3 AOP 的实现者 221
7.2 基础知识 222
7.2.1 带有横切逻辑的实例 222
7.2.2 JDK 动态代理 224
7.2.3 CGLib 动态代理 228
7.2.4 AOP 联盟 229
7.2.5 代理知识小结 230
7.3 创建增强类 230
7.3.1 增强类型 230
7.3.2 前置增强 231
7.3.3 后置增强 235
7.3.4 环绕增强 236
7.3.5 异常抛出增强 237
7.3.6 引介增强 239
7.4 创建切面 243
7.4.1 切点类型 243
7.4.2 切面类型 244
7.4.3 静态普通方法名匹配切面 246
7.4.4 静态正则表达式方法匹配切面 248
7.4.5 动态切面 251
7.4.6 流程切面 254
7.4.7 复合切点切面 256
7.4.8 引介切面 258
7.5 自动创建代理 259
7.5.1 实现类介绍 259
7.5.2 BeanNameAutoProxyCreator 260
7.5.3 DefaultAdvisorAutoProxyCreator 261
7.5.4 AOP 无法增强疑难问题剖析 262
7.6 小结 267
第 8 章 基于@AspectJ 和 Schema 的 AOP 269
8.1 Spring 对 AOP 的支持 269
8.2 Java 5.0 注解知识快速进阶 270
8.2.1 了解注解 270
8.2.2 一个简单的注解类 271
8.2.3 使用注解 272
8.2.4 访问注解 273
8.3 着手使用@AspectJ 274
8.3.1 使用前的准备 275
8.3.2 一个简单的例子 275
8.3.3 如何通过配置使用@AspectJ 切面 277
8.4 @AspectJ 语法基础 278
8.4.1 切点表达式函数 278
8.4.2 在函数入参中使用通配符 279
8.4.3 逻辑运算符 280
8.4.4 不同增强类型 281
8.4.5 引介增强用法 282
8.5 切点函数详解 283
8.5.1 @annotation() 284
8.5.2 execution() 285
8.5.3 args()和@args() 287
8.5.4 within() 288
8.5.5 @within()和@target() 289
8.5.6 target()和 this() 290

8.6 @AspectJ 进阶 291
8.6.1 切点复合运算 292
8.6.2 命名切点 292
8.6.3 增强织入的顺序 294
8.6.4 访问连接点信息 294
8.6.5 绑定连接点方法入参 295
8.6.6 绑定代理对象 297
8.6.7 绑定类注解对象 298
8.6.8 绑定返回值 299
8.6.9 绑定抛出的异常 299
8.7 基于 Schema 配置切面 300
8.7.1 一个简单切面的配置 300
8.7.2 配置命名切点 302
8.7.3 各种增强类型的配置 303
8.7.4 绑定连接点信息 305
8.7.5 Advisor 配置 306
8.8 混合切面类型 307
8.8.1 混合使用各种切面类型 308
8.8.2 各种切面类型总结 308
8.9 其他 309
8.9.1 JVM Class 文件字节码转换基础知识 309
8.9.2 使用 LTW 织入切面 311
8.10 小结 314
第 9 章 Spring SpEL 316
9.1 JVM 动态语言 316
9.2 SpEL 表达式概述 318
9.3 SpEL 核心接口 319
9.3.1 EvaluationContext 接口 320
9.3.2 SpEL 编译器 321
9.4 SpEL 基础表达式 323
9.4.1 文本字符解析 323
9.4.2 对象属性解析 323
9.4.3 数组、集合类型解析 324
9.4.4 方法解析 326
9.4.5 操作符解析 327
9.4.6 安全导航操作符 329
9.4.7 三元操作符 330
9.4.8 Elvis 操作符 331
9.4.9 赋值、类型、构造器、变量 332
9.4.10 集合过滤 335
9.4.11 集合转换 335
9.5 在 Spring 中使用 SpEL 336
9.5.1 基于 XML 的配置 336
9.5.2 基于注解的配置 337
9.6 小结 338

第 3 篇 数据篇

第 10 章 Spring 对 DAO 的支持 340
10.1 Spring 的 DAO 理念 340
10.2 统一的异常体系 341
10.2.1 Spring 的 DAO 异常体系 341
10.2.2 JDBC 的异常转换器 343
10.2.3 其他持久化技术的异常转换器 344
10.3 统一数据访问模板 344
10.3.1 使用模板和回调机制 345
10.3.2 Spring 为不同持久化技术所提供的模板类 347
10.4 数据源 348
10.4.1 配置一个数据源 348
10.4.2 获取 JNDI 数据源 352
10.4.3 Spring 的数据源实现类 353
10.5 小结 353

第 11 章 Spring 的事务管理.....355
11.1 数据库事务基础知识 355
11.1.1 何为数据库事务.....356
11.1.2 数据并发的问题.....357
11.1.3 数据库锁机制.....359
11.1.4 事务隔离级别.....360
11.1.5 JDBC 对事务的支持.....361
11.2 ThreadLocal 基础知识..... 362
11.2.1 ThreadLocal 是什么 363
11.2.2 ThreadLocal 的接口方法 363
11.2.3 一个 TheadLocal 实例.....364
11.2.4 与 Thread 同步机制的比较....366
11.2.5 Spring 使用 ThreadLocal 解决线程安全问题.....366
11.3 Spring 对事务管理的支持..... 368
11.3.1 事务管理关键抽象.....369
11.3.2 Spring 的事务管理器实现类 ...371
11.3.3 事务同步管理器.....374
11.3.4 事务传播行为.....375
11.4 编程式的事务管理 376
11.5 使用 XML 配置声明式事务 377
11.5.1 一个将被实施事务增强的服务接口 379
11.5.2 使用原始的 TransactionProxy FactoryBean.....379
11.5.3 基于 aop/tx 命名空间的配置 ...382
11.6 使用注解配置声明式事务 385
11.6.1 使用@Transactional 注解.....385
11.6.2 通过 AspectJ LTW 引入事务切面 389
11.7 集成特定的应用服务器 390
11.7.1 BEA WebLogic 390
11.7.2 WebSphere.....390
11.8 小结..... 390
第 12 章 Spring 的事务管理难点剖析... 392
12.1 DAO 和事务管理的牵绊 393
12.1.1 JDBC 访问数据库 393
12.1.2 Hibernate 访问数据库 395
12.2 应用分层的迷惑..... 398
12.3 事务方法嵌套调用的迷茫 401
12.3.1 Spring 事务传播机制回顾 401
12.3.2 相互嵌套的服务方法 402
12.4 多线程的困惑..... 405
12.4.1 Spring 通过单实例化 Bean 简化多线程问题.....405
12.4.2 启动独立线程调用事务方法.....405
12.5 联合军种作战的混乱..... 408
12.5.1 Spring 事务管理器的应对 408
12.5.2 Hibernate+Spring JDBC 混合框架的事务管理 408
12.6 特殊方法成漏网之鱼 412
12.6.1 哪些方法不能实施 Spring AOP 事务.....412
12.6.2 事务增强遗漏实例 413
12.7 数据连接泄露..... 416
12.7.1 底层连接资源的访问问题 416
12.7.2 Spring JDBC 数据连接泄露.....417
12.7.3 事务环境下通过 DataSource Utils 获取数据连接 420
12.7.4 无事务环境下通过 DataSource Utils 获取数据连接.....422
12.7.5 JdbcTemplate 如何做到对连接泄露的免疫.....424

12.7.6 使用 TransactionAwareData SourceProxy.....425
12.7.7 其他数据访问技术的等价类...426
12.8 小结.....426
第 13 章 使用 Spring JDBC 访问数据库.....428
13.1 使用 Spring JDBC.....428
13.1.1 JdbcTemplate 小试牛刀.....429
13.1.2 在 DAO 中使用 Jdbc Template.....429
13.2 基本的数据操作.....431
13.2.1 更改数据.....431
13.2.2 返回数据库的表自增主键值...434
13.2.3 批量更改数据.....436
13.2.4 查询数据.....437
13.2.5 查询单值数据.....440
13.2.6 调用存储过程.....442
13.3 BLOB/CLOB 类型数据的操作.....444
13.3.1 如何获取本地数据连接.....445
13.3.2 相关的操作接口.....446
13.3.3 插入 LOB 类型的数据.....448
13.3.4 以块数据方式读取 LOB 数据.....450
13.3.5 以流数据方式读取 LOB 数据.....451
13.4 自增键和行集.....452
13.4.1 自增键的使用.....452
13.4.2 如何规划主键方案.....454
13.4.3 以行集返回数据.....456
13.5 NamedParameterJdbcTemplate 模板类.....456
13.6 小结.....459
第 14 章 整合其他 ORM 框架.....460
14.1 Spring 整合 ORM 技术.....460
14.2 在 Spring 中使用 Hibernate.....462
14.2.1 配置 SessionFactory.....462
14.2.2 使用 HibernateTemplate.....465
14.2.3 处理 LOB 类型的数据.....469
14.2.4 添加 Hibernate 事件监听器...470
14.2.5 使用原生的 Hibernate API.....471
14.2.6 使用注解配置.....472
14.2.7 事务处理.....474
14.2.8 延迟加载问题.....475
14.3 在 Spring 中使用 MyBatis.....476
14.3.1 配置 SqlMapClient.....476
14.3.2 在 Spring 中配置 MyBatis.....478
14.3.3 编写 MyBatis 的 DAO.....479
14.4 DAO 层设计.....482
14.4.1 DAO 基类设计.....482
14.4.2 查询接口方法设计.....484
14.4.3 分页查询接口设计.....486
14.5 小结.....487
第 4 篇 应用篇
第 15 章 Spring Cache.....490
15.1 缓存概述.....490
15.1.1 缓存的概念.....490
15.1.2 使用 Spring Cache.....493
15.2 掌握 Spring Cache 抽象.....498
15.2.1 缓存注解.....498
15.2.2 缓存管理器.....504
15.2.3 使用 SpEL 表达式.....506
15.2.4 基于 XML 的 Cache 声明.....506

15.2.5 以编程方式初始化缓存.....507
15.2.6 自定义缓存注解.....509
15.3 配置 Cache 存储.....509
15.3.1 EhCache.....510
15.3.2 Guava.....510
15.3.3 HazelCast.....511
15.3.4 GemFire.....511
15.3.5 JSR-107 Cache.....511
15.4 实战经验.....513
15.5 小结.....514
第 16 章 任务调度和异步执行器.....516
16.1 任务调度概述.....516
16.2 Quartz 快速进阶.....517
16.2.1 Quartz 基础结构.....518
16.2.2 使用 SimpleTrigger.....520
16.2.3 使用 CronTrigger.....522
16.2.4 使用 Calendar.....526
16.2.5 任务调度信息存储.....527
16.3 在 Spring 中使用 Quartz.....530
16.3.1 创建 JobDetail.....530
16.3.2 创建 Trigger.....533
16.3.3 创建 Scheduler.....534
16.4 在 Spring 中使用 JDK Timer.....536
16.4.1 Timer 和 TimerTask.....536
16.4.2 Spring 对 Java Timer 的支持.....539
16.5 Spring 对 Java 5.0 Executor 的支持.....540
16.5.1 了解 Java 5.0 的 Executor.....541
16.5.2 Spring 对 Executor 所提供的抽象.....543
16.6 实际应用中的任务调度.....544
16.6.1 如何产生任务.....545
16.6.2 任务调度对应用程序集群的影响.....547
16.6.3 任务调度云.....547
16.6.4 Web 应用程序中调度器的启动和关闭问题.....549
16.7 小结.....552
第 17 章 Spring MVC.....553
17.1 Spring MVC 体系概述.....554
17.1.1 体系结构.....554
17.1.2 配置 DispatcherServlet.....555
17.1.3 一个简单的实例.....560
17.2 注解驱动的控制器.....565
17.2.1 使用@RequestMapping 映射请求.....565
17.2.2 请求处理方法签名.....569
17.2.3 使用矩阵变量绑定参数.....570
17.2.4 请求处理方法签名详细说明.571
17.2.5 使用 HttpMessageConverter <T>.....575
17.2.6 使用@RestController 和 AsyncRestTemplate.....584
17.2.7 处理模型数据.....586
17.3 处理方法的数据绑定.....591
17.3.1 数据绑定流程剖析.....592
17.3.2 数据转换.....592
17.3.3 数据格式化.....598
17.3.4 数据校验.....602
17.4 视图和视图解析器.....611
17.4.1 认识视图.....611
17.4.2 认识视图解析器.....612
17.4.3 JSP 和 JSTL.....613
17.4.4 模板视图.....618

17.4.5 Excel 621
17.4.6 PDF 623
17.4.7 输出 XML 625
17.4.8 输出 JSON 626
17.4.9 使用 XmlViewResolver 626
17.4.10 使用 ResourceBundleView Resolver 627
17.4.11 混合使用多种视图技术 628
17.5 本地化解析 630
17.5.1 本地化的概念 630
17.5.2 使用 CookieLocaleResolver 631
17.5.3 使用 SessionLocaleResolver 632
17.5.4 使用 LocaleChange Interceptor 632
17.6 文件上传 633
17.6.1 配置 MultipartResolver 633
17.6.2 编写控制器和文件上传表单页面 633
17.7 WebSocket 支持 634
17.7.1 使用 WebSocket 634
17.7.2 WebSocket 的限制 638
17.8 杂项 639
17.8.1 静态资源处理 639
17.8.2 装配拦截器 643
17.8.3 异常处理 644
17.8.4 RequestContextHolder 的使用 646
17.9 小结 646
第 18 章 实战案例开发 648
18.1 论坛案例概述 648
18.1.1 论坛整体功能结构 648
18.1.2 论坛用例描述 649
18.1.3 主要功能流程描述 651
18.2 系统设计 655
18.2.1 技术框架选择 655
18.2.2 采用 Maven 构建项目 656
18.2.3 单元测试类包结构规划 657
18.2.4 系统架构图 658
18.2.5 PO 类设计 658
18.2.6 持久层设计 659
18.2.7 服务层设计 660
18.2.8 Web 层设计 661
18.2.9 数据库设计 662
18.3 开发前的准备 663
18.4 持久层开发 664
18.4.1 PO 类 664
18.4.2 DAO 基类 666
18.4.3 通过扩展基类定义 DAO 类 670
18.4.4 DAO Bean 的装配 672
18.4.5 使用 Hibernate 二级缓存 673
18.5 对持久层进行测试 675
18.5.1 配置 Unitils 测试环境 675
18.5.2 准备测试数据库及测试数据 676
18.5.3 编写 DAO 测试基类 677
18.5.4 编写 BoardDao 测试用例 678
18.6 服务层开发 680
18.6.1 UserService 的开发 680
18.6.2 ForumService 的开发 681
18.6.3 服务类 Bean 的装配 683
18.7 对服务层进行测试 684
18.7.1 编写 Service 测试基类 685
18.7.2 编写 ForumService 测试用例 685

18.8 Web 层开发 687
18.8.1 BaseController 的基类 687
18.8.2 用户登录和注销 689
18.8.3 用户注册 691
18.8.4 论坛管理 692
18.8.5 论坛普通功能 694
18.8.6 分页显示论坛版块的主题帖子 696
18.8.7 web.xml 配置 700
18.8.8 Spring MVC 配置 702
18.9 对 Web 层进行测试 703
18.9.1 编写 Web 测试基类 703
18.9.2 编写 ForumManageController 测试用例 704
18.10 开发环境部署 705
18.11 项目配置实战经验 708
18.11.1 “传统的”Web 项目属性文件 708
18.11.2 如何规划便于部署的 Web 项目属性文件 709
18.11.3 数据源的配置 710
18.12 小结 712

第 5 篇 提高篇

第 19 章 Spring OXM 714
19.1 认识 XML 解析技术 714
19.1.1 什么是 XML 714
19.1.2 XML 的处理技术 715
19.2 XML 处理利器：XStream 717
19.2.1 XStream 概述 717
19.2.2 快速入门 718
19.2.3 使用 XStream 别名 720
19.2.4 XStream 转换器 721
19.2.5 XStream 注解 723
19.2.6 流化对象 725
19.2.7 持久化 API 726
19.2.8 额外功能：处理 JSON 727
19.3 其他常见的 O/X Mapping 开源项目 729
19.3.1 JAXB 729
19.3.2 Castor 733
19.3.3 JiBX 738
19.3.4 总结比较 741
19.4 与 Spring OXM 整合 742
19.4.1 Spring OXM 概述 742
19.4.2 整合 OXM 实现者 744
19.4.3 如何在 Spring 中进行配置 744
19.4.4 Spring OXM 简单实例 747
19.5 小结 749
第 20 章 实战型单元测试 750
20.1 单元测试概述 751
20.1.1 为什么需要单元测试 751
20.1.2 单元测试之误解 752
20.1.3 单元测试之困境 754
20.1.4 单元测试基本概念 755
20.2 TestNG 快速进阶 757
20.2.1 TestNG 概述 757
20.2.2 TestNG 生命周期 758
20.2.3 使用 TestNG 758
20.3 模拟利器 Mockito 763
20.3.1 模拟测试概述 763
20.3.2 创建 Mock 对象 763

20.3.3 设定 Mock 对象的期望行为及返回值 764
20.3.4 验证交互行为 766
20.4 测试整合之王 Unitils 767
20.4.1 Unitils 概述 767
20.4.2 集成 Spring 770
20.4.3 集成 Hibernate 773
20.4.4 集成 DbUnit 774
20.4.5 自定义扩展模块 775
20.5 使用 Unitils 测试 DAO 层 776
20.5.1 数据库测试的难点 776
20.5.2 扩展 DbUnit 用 Excel 准备数据 776
20.5.3 测试实战 779
20.6 使用 Unitils 测试 Service 层 789
20.7 测试 Web 层 794
20.7.1 对 LoginController 进行单元测试 794
20.7.2 使用 Spring Servlet API 模拟对象 795
20.7.3 使用 Spring RestTemplate 测试 797
20.8 小结 798

第1篇

基础篇

第 1 章

Spring 概述

Spring 已经成为 Java 应用首选的 full-stack 开发框架，它本着“从实践中来，到实践中去”的原则，对传统 EJB 重量型框架的思想进行了颠覆性的革新，通过 Rod Johnson 天才般的演绎，使 Spring 在短时间内就成为用户众多、社群庞大、文档丰富、极具实用性的开源开发框架。目前，Spring 已经升级到 4.2 版本，全面支持 Java SE 8、Java EE 7，向下兼容 Java SE 6/Java EE 6，新添加如泛型依赖注入、Lambda 表达式的支持、Groovy DSL 定义 Bean、核心容器增强、Web 框架增强、WebSocket 模块的实现、测试增强等功能，全面支持 REST 风格的 Web 开发。Spring 家族系列的子项目更加丰富，Spring 在支持产品化开发、大数据、云计算及微服务架构（MSA）、Mobile 这些引领世界风潮的技术上拥有令人惊美的表现。

本章主要内容：

- 认识 Spring
- Spring 体系及文档结构
- Spring 4.0 新特性
- Spring 子项目介绍

本章亮点：

- Spring 子项目介绍
- Spring 4.0 新特性

1.1 认识 Spring

Spring——春天，春风又绿江南岸的春天，花团锦簇、草长莺飞的春天。一提到近几年的 Java 开源世界，喜爱历史的人也许马上就能联想到春秋战国时期那段百花齐放、

百家争鸣、思想文化空前繁荣的学术春天，而 Spring 正是百花齐放的 Java 开源世界里一朵芳香馥郁的奇葩。

Spring 是众多 Java 开源项目中的一员，唯一不同的是，它秉承着破除权威迷信，一切从实践中来到实践中去的理念，宛如阿基米德手中的杠杆，以一己之力撼动了 Java EE 传统重量级框架坚如磐石的大厦。

要用一两句话总结出 Spring 的所有内涵确实有点困难，但为了先给大家一个基本的印象，我们尝试着进行以下概括。

Spring 是分层的 Java SE/EE 应用一站式的轻量级开源框架，以 IoC（Inverse of Control，控制反转）和 AOP（Aspect Oriented Programming，切面编程）为内核，提供了展现层 Spring MVC、持久层 Spring JDBC 及业务层事务管理等一站式的企业级应用技术。此外，Spring 以海纳百川的胸怀整合了开源世界里众多著名的第三方框架和类库，逐渐成为使用最多的轻量级 Java EE 企业应用开源框架。

说起 Spring，我们不免要提到 Spring 的缔造者 Rod Johnson 这位 Java 奇才。Rod Johnson 不仅在悉尼大学获得了计算机学士学位，同时还是一位音乐学博士，也许是音乐的细胞赋予了他程序设计美学的灵感，让他成就了 Spring 的简约和优雅。他不但早在 1996 年就涉足了 Java 技术，同时也对 C/C++有着深厚的造诣。他参与过众多保险、电子商务和金融等行业大型项目的开发，具有丰富的实践经验。同时他还是 JCP 活跃的成员，是 JSR-154（Servlet 2.4）和 JDO 2.0 规范的专家。

Rod Johnson 在 2002 年编著的 *Expert One-to-One J2EE Design and Development* 一书中，对 Java EE 正统框架臃肿、低效、脱离现实的种种学院派做法提出了质疑，并积极寻求探索革新之道。以此书为指导思想，他编写了 interface21 框架，这是一个力图冲破 Java EE 传统开发的困境，从实际需求出发，着眼于轻便、灵巧，易于开发、测试和部署的轻量级开发框架。Spring 框架即以 interface21 框架为基础，经过重新设计，并不断丰富其内涵，于 2004 年 3 月 24 日发布了 1.0 正式版。同年他又推出了一部堪称经典的力作 *Expert One-to-One J2EE Development without EJB*，该书在 Java 世界掀起了轩然大波，改变了 Java 开发者程序设计和开发的思考方式。在该书中，他根据自己多年丰富的实践经验，对 EJB 的各项笨重臃肿的结构进行了逐一的分析和否定，并分别以简洁实用的方式替换之。至此一战功成，Rod Johnson 成为一个改变 Java 世界的大师级人物。

从 2004 年发布第一个版本以来，Spring 逐渐占据了 Java 开发人员的视线，获得了开源社区一片赞誉之声，Java 开源社区里“春”色满园关不住。在著名的 GitHub 开源网站上，Spring 是大家始终关注的热点项目，下载量名列前茅。在国内的开源中国网站（http://www.oschina.net）上，Spring 也是大家关注讨论的焦点。

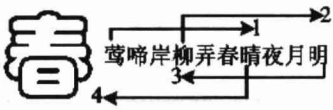
Spring 拥有庞大的社区、活跃的开发队伍、丰富的文档、众多的应用案例。国内的 Spring 社区也异常活跃，对 Spring 在国内的推广起到了推波助澜的作用。下面是其中的一些代表者。

首先提到的是 SpringFramework 中文论坛（http://springside.io），这是一个出色的 Spring 专业论坛，2008 年，论坛组织和满江红开放技术研究组织（http://www.redsaga.com）联手，快速并出色地完成了 Spring 开发手册的翻译工作，使众多国内 Spring 爱好者从中受益（由于种种原因，目前这两个网站均停止运营，但我们仍然不能忘记它们所做的贡献）。

其次，面向高阶 Java 开发人员的 IT 视线论坛（http://www.iteye.com）是 Spring 爱好者极好的交流平台，众多无私的开发者慷慨地分享他们的开发心得和经验，为 Spring 在国内的推广做出了重大贡献。

轻松一刻

在茶诗中，最有奇趣的要数回文诗。回文是利用汉语的词序、语法、词义灵活的特点构成的一种修辞方法。回文诗有多种形式，如通体回文、就句回文、双句回文、本篇回文、环复回文等。《四时山水诗》是明末浙江才女吴绛雪所作的回文诗，共包括春、夏、秋、冬四句就句回文诗，其中对应“春”的回文诗句是“莺啼岸柳弄春晴夜月明”，经回文拆解后，每句诗都包括一个“春”字，且很富诗意，颇具欣赏性。



莺啼岸柳弄春晴，
柳弄春晴夜月明。
明月夜晴春弄柳，
晴春弄柳岸啼莺。

1.2 关于 SpringSource

Rod Johnson 不但是一位技术奇才，也是一位商业奇才。当 Spring 1.0 发布时，Rod 就和他的骨干团队成立了 SpringSource 公司，以商业化的方式对开源的 Spring 进行运作。以 Spring 应用的开源框架为依托，成功开展了很多代表不同技术领域的子项目，将 Spring 的触角延伸到应用安全、云计算、批量数据处理等技术领域。同时，通过这些子项目的探索，不断改进和丰富 Spring 框架的内涵，使 Spring 的用户越来越多。

Rod 不但注重 Spring 框架“内功”的打造，还很关注市场的推广和宣传，不间断地在世界各地提供宣传、培训和咨询服务。还像 IBM、微软、RedHat 等公司一样开展技术认证服务，找到了商业盈利的模式。

2007 年 5 月，SpringSource 吸引了著名的 Benchmark Capital 风险投资商（Benchmark 成功投资过 MySQL、RedHat 和 eBay 等公司），Benchmark Capital 提供了 1000 万美元的资金。后来，Benchmark 还联合了 Accel 共同投资，后者在 2008 年年初提供了 1500 万美元的资金。

2008 年，SpringSource 收购了 G2One——Groovy 编程语言和 Grails Web 框架背后的公司，以及 Covalent——为 Apache 的 Tomcat 应用服务器提供支持的公司。

2009 年，SpringSource 收购了开源系统监测厂商 Hyperic。Hyperic 的核心产品是 Hyperic HQ，该产品提供了硬件和操作系统、虚拟机、数据库及应用服务器的可用性监测。SpringSource 的核心业务是销售 SpringSource 企业版服务器，使用 Hyperic 产品后，就相当于配备了系统监测的测量仪器。SpringSource 的最终目标是借此在云计算市场拥有越来越多的话语权，这可以通过 Rod 的这句话得到证实：“云的兴起使得消除开发者与运营者之间的隔阂更为重要了。我们相信我们的中间件作为 Java 云技术的基础是最好的，而 SpringSource/Hyperic 组合将让我们能够以一种独一无二的方式消除开发者和运营者之间的隔阂。”

SpringSource 一直致力于成为能够同时提供应用开发框架、应用服务器及应用服务监控的提供商。在收购 Covalent 和 Hyperic 之后，SpringSource 终于实现了这一宏图。更多的喜讯马上接踵而来：2009 年 8 月 11 日，商业软件生产商 VMware 宣布，斥资 4.2 亿美元收购 SpringSource 公司。这一消息无疑是 2009 年开源社区领域极其重大的消息之一。合并后，VMware 和 SpringSource 计划共同开发集成化的平台即服务（Platform as a Service，PaaS）解决方案，期望它能应用于客户数据中心或被云服务提供商采用。

2012 年，Spring 创始人 Rod 离开 SpringSource 和 VMware，“去从事其他一些感兴趣的事情”。

2013 年，SpringSource 团队发布了 Spring Framework 4.0 的相关计划，这是 Spring 框架的下一个升级版本，首个 4.0 里程碑版本的主要改进包括：首次支持 Java SE 8、Java EE 7 及 WebSocket 编程。2013 年 12 月，发布 Spring Framework 4.0 正式版本。

1.3　Spring 带给我们什么

也许有很多开发者曾经被 EJB 的过度宣传所迷惑，成为 EJB 的拥趸者，并因此拥有一段痛苦的开发经历。EJB 的复杂源于它对所有的企业应用采用统一的标准，它认为所有的企业应用都需要分布式对象、远程事务，因此造就了 EJB 框架的极度复杂。这种复杂不仅造成了陡峭的学习曲线，而且给开发、测试、部署工作造成了很多额外的要求和工作量。其中最大的诟病就是难于测试，因为这种测试不能脱离 EJB 容器，每次测试都需要进行应用部署并启动 EJB 容器，而部署和启动 EJB 容器是一项费时费力的重型操作，其结果是测试工作往往成为开发工作的瓶颈。

但 EJB 并非一无是处，它提供了很多可圈可点的服务，如声明事务、透明持久化等。Spring 承认 EJB 中存在优秀的东西，只是它的实现太复杂、要求过严过高，所以 Spring 在努力提供类似服务的同时尽量简化开发，Spring 认为 Java EE 的开发应该更容易、更简单。在实现这一目标时，Spring 一直贯彻并遵守“好的设计优于具体实现，代码应易

于测试”这一理念，最终带给我们一个易于开发、便于测试且功能齐全的开发框架。概括起来，Spring 给我们带来以下好处。

- 方便解耦，简化开发。通过 Spring 提供的 IoC 容器，用户可以将对象之间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合。有了 Spring，用户不必再为单实例模式类、属性文件解析等这些底层的需求编写代码，可以更专注于上层的应用。
- AOP 编程的支持。通过 Spring 提供的 AOP 功能，方便进行面向切面的编程，很多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应对。
- 声明式事务的支持。在 Spring 中，用户可以从单调烦闷的事务管理代码中解脱出来，通过声明的方式灵活地进行事务管理，提高开发效率和质量。
- 方便程序的测试。可以用非容器依赖的编程方式进行几乎所有的测试工作。在 Spring 里，测试不再是昂贵的操作，而是随手可做的事情。
- 方便集成各种优秀框架。Spring 不排斥各种优秀的开源框架，相反，Spring 可以降低各种框架的使用难度。Spring 提供了对各种优秀框架（如 Struts、Hibernate、Hessian、Quartz 等）的直接支持。
- 降低 Java EE API 的使用难度。Spring 对很多难用的 Java EE API（如 JDBC、JavaMail、远程调用等）提供了一个薄层封装，通过 Spring 的简易封装，这些 Java EE API 的使用难度大大降低。
- Java 源码是经典的学习范例。Spring 的源码设计精妙、结构清晰、匠心独运，处处体现着大师对 Java 设计模式的灵活运用及对 Java 技术的高深造诣。Spring 框架源码无疑是 Java 技术的最佳实践范例。如果想在短时间内迅速提高自己的 Java 技术水平和应用开发水平，学习和研究 Spring 源码将会收到意想不到的效果。

1.4 Spring 体系结构

Spring 核心框架由 4000 多个类组成，整个框架按其所属功能可以划分为 5 个主要模块，如图 1-1 所示。

从整体来看，这 5 个主要模块几乎为企业应用提供了所需的一切，从持久层、业务层到展现层都拥有相应的支持。就像吕布的赤兔马和方天画戟、秦琼的黄骠马和熟铜锏，IoC 和 AOP 是 Spring 所依赖的根本。在此基础上，Spring 整合了各种企业应用开源框架和许多优秀的第三方类库，成为 Java 企业应用 full-stack 的开发框架。Spring 框架的精妙之处在于对于开发者拥有自由的选择权，Spring 不会将自己的意志强加给开发者，因为针对某个领域的问题，Spring 往往支持多种实现方案。当希望选用不同的实现方案时，Spring 又能保证过渡的平滑性。

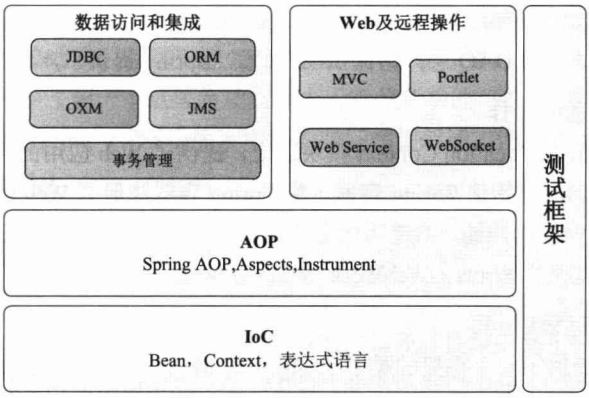


图 1-1　Spring 框架结构

1．IoC

Spring 核心模块实现了 IoC 的功能，它将类与类之间的依赖从代码中脱离出来，用配置的方式进行依赖关系描述，由 IoC 容器负责依赖类之间的创建、拼接、管理、获取等工作。BeanFactory 接口是 Spring 框架的核心接口，它实现了容器许多核心的功能。

Context 模块构建于核心模块之上，扩展了 BeanFactory 的功能，添加了 i18n 国际化、Bean 生命周期控制、框架事件体系、资源加载透明化等多项功能。此外，该模块还提供了许多企业级服务的支持，如邮件服务、任务调度、JNDI 获取、EJB 集成、远程访问等。ApplicationContext 是 Context 模块的核心接口。

表达式语言模块是统一表达式语言（Unified EL）的一个扩展，该表达式语言用于查询和管理运行期的对象，支持设置/获取对象属性，调用对象方法，操作数组、集合等。此外，该模块还提供了逻辑表达式运算、变量定义等功能，可以方便地通过表达式串和 Spring IoC 容器进行交互。

2．AOP

AOP 是继 OOP 之后，对编程设计思想影响极大的技术之一。AOP 是进行横切逻辑编程的思想，它开拓了考虑问题的思路。在 AOP 模块里，Spring 提供了满足 AOP Alliance 规范的实现，还整合了 AspectJ 这种 AOP 语言级的框架。在 Spring 里实现 AOP 编程有众多选择。Java 5.0 引入 java.lang.instrument，允许在 JVM 启动时启用一个代理类，通过该代理类在运行期修改类的字节码，改变一个类的功能，从而实现 AOP 的功能。

3．数据访问和集成

任何应用程序的核心问题是对数据的访问和操作。数据有多种表现形式，如数据表、XML、消息等，而每种数据形式又拥有不同的数据访问技术（如数据表的访问既可以直接通过 JDBC，也可以通过 Hibernate 或 MyBatis）。

首先，Spring 站在 DAO 的抽象层面，建立了一套面向 DAO 层的统一的异常体系，同时将各种访问数据的检查型异常转换为非检查型异常，为整合各种持久层框架提供基础。其次，Spring 通过模板化技术对各种数据访问技术进行了薄层封装，将模式化的代

码隐藏起来，使数据访问的程序得到大幅简化。这样，Spring 就建立起了和数据形式及访问技术无关的统一的 DAO 层，借助 AOP 技术，Spring 提供了声明式事务的功能。

4．Web 及远程操作

该模块建立在 Application Context 模块之上，提供了 Web 应用的各种工具类，如通过 Listener 或 Servlet 初始化 Spring 容器，将 Spring 容器注册到 Web 容器中。该模块还提供了多项面向 Web 的功能，如透明化文件上传、Velocity、FreeMarker、XSLT 的支持。此外，Spring 可以整合 Struts、WebWork 等 MVC 框架。

5．Web 及远程访问

Spring 自己提供了一个完整的类似于 Struts 的 MVC 框架，称为 Spring MVC。据说 Spring 之所以也提供了一个 MVC 框架，是因为 Rod Johnson 想证明实现 MVC 其实是一项简单的工作。当然，如果你不希望使用 Spring MVC，那么 Spring 对 Struts、WebWork 等 MVC 框架的整合，一定也可以给你带来方便。相对于 Servlet 的 MVC，Spring 在简化 Portlet 的开发上也做了很多工作，开发者可以从中受益。

6．WebSocket

WebSocket 提供了一个在 Web 应用中高效、双向的通信，需要考虑到客户端（浏览器）和服务器之间的高频和低时延消息交换。一般的应用场景有在线交易、游戏、协作、数据可视化等。

此外，Spring 在远程访问及 Web Service 上提供了对很多著名框架的整合。由于 Spring 框架的扩展性，特别是随着 Spring 框架影响性的扩大，越来越多的框架主动支持 Spring 框架，使得 Spring 框架应用的涵盖面越来越宽广。

1.5 Spring 对 Java 版本的要求

Spring 4.0 基于 Java 6.0，全面支持 Java 8.0。运行 Spring 4.0 必须使用 Java 6.0 以上版本，推荐使用 Java 8.0 及以上版本，如果要编译 Spring 4.0，则必须使用 Java 8.0。此外，Spring 保持和 Java EE 6.0 的兼容，同时也对 Java EE 7.0 提供一些早期的支持。

1.6 Spring 4.0 新特性

2009 年 12 月正式发布 Spring 3.0 版本，2013 年 12 月发布 Spring Framework 4.0 正式版本，在本书撰写时（2015 年 10 月）的最新版本是 Spring 4.2.2。相比 Spring 3.x，Spring 4.2.2 把众多新功能被添加到 Spring 中，全面支持 Java SE 8、Java EE 7，而且向下兼容到 Java SE 6/Java EE 6，并移除一些过时的类，添加如泛型依赖注入、Lambda 表

达式的支持、Groovy DSL 定义 Bean、核心容器增强、Web 框架增强、WebSocket 模块的实现、测试增强等功能，全面支持 REST 风格的 Web 开发。在进入 Spring 具体内容的学习之前，有必要了解一下这些新功能。由于有些新功能可能是在 Spring 4.0 中添加的，也有可能是在 Spring 4.x 等小版本中添加的，为了叙述方便，在一般情况下，我们统一称之为 Spring 4.0。

1.6.1　全面支持 Java 8.0

Spring 框架本身是由 Java 8.0 编译器编译的，编译时使用的是生成 Java 6 字节码的编译命令选项，因此可以使用 Java 6.0、7.0 或 8.0 来运行 Spring 4.0 的应用。

Java 8.0 编译器编译过的代码需要在 Java 8.0 或以上的 Java 虚拟机上运行。由于在 Spring 框架中大量应用了反射机制和 Asm、Cglib 等函数库，必须确保这些函数库能理解 Java 8.0 生成的新.class 文件。因此，Spring 将 Asm、Cglib 等函数库通过 jarjar（https://code.google.com/p/jarjar/）嵌入 Spring 框架中，这样 Spring 就可以同时支持 Java 6.0、7.0 和 8.0 的字节码而不会产生运行时错误。

1．Java 8.0 的 Lambda 表达式

Java 8.0 的设计者想保证它是向下兼容的，以使 Lambda 表达式能在旧版本的代码编译器中使用。向下兼容通过定义函数式接口来实现。

Spring 的代码里有很多函数式接口，因此，Lambda 表达式可以很容易地与 Spring 结合使用。即便 Spring 框架本身被编译成 Java 6.0 的.class 文件格式，仍然可以用 Java 8.0 的 Lambda 表达式编写应用代码。

总之，因为 Spring 框架早在 Java 8.0 之前就已经使用了函数式接口，所以在 Spring 里使用 Lambda 表达式非常容易。

2．Java 8.0 的时间与日期 API

Java 开发者对 java.util.Date 笨拙的设计忍耐已久，现在 Java 8.0 带来了全新的日期与时间 API，在 java.time 包中引入了一系列有用的新类，如 LocalDate、LocalTime 和 LocalDateTime 等，解决了那些久被诟病的问题。

Spring 有一个数据转换框架，它可以使字符串和 Java 数据类型相互转换。Spring 4.0 升级了这个转换框架，以支持 Java 8.0 日期与时间 API 里的类，如代码清单 1-1 所示。

代码清单 1-1　LocalDateController

```
@RestController
public class  LocalDateController {
  @RequestMapping("/date/{localDate}")
  public String get(@DateTimeFormat(iso = ISO.DATE) LocalDate localDate) {
     return localDate.toString();
  }
}
```

3. 重复注解支持

Java 8.0 增加了对重复注解的支持，Spring 4.0 也同样支持。目前 Spring 4.0 仅支持对注解@Scheduled 和@PropertySource 的重复。例如，可以在一个类中使用多个@PropertySource 注解来加载不同的资源配置文件，如代码清单 1-2 所示。

代码清单 1-2 Application

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
@PropertySource("classpath:/conf1.properties")
@PropertySource("classpath:/ conf2.properties")
public class Application {
    @Autowired
    private Environment env;

    @Bean
    public MyBean getBean() {
        return new MyBean(env.getProperty("app1.p1"),env.getProperty("app2.p1"));
    }
    public static void main(String[] args) {
        SpringApplication.run(AppServer.class, args);
    }
}
```

4. 空指针终结者：Optional<>

java.lang.NullPointerException 是最常见也是最令人讨厌的一种异常，如果一个对象可能为 null，在调用其方法之前就必须进行非空检查，否则就会引发 NullPointerException。但是，很多对象永远都不会为 null，如果能把那些可能会为 null 的对象明确标识出来，只对 null 嫌疑者进行判断，岂不是既可避免 NullPointerException 又可避免不必要的非空判断？Java 8.0 的 java.util.Optional 就是为此而生的，它明确指示开发者哪些对象是需要非空检查的。

目前，Spring 4.0 可在两种场合使用 Java Optional。在代码清单 1-3 中，假设 userDao 不一定会被注入进来，原来必须使用@Autowired(required=false)，但现在直接使用 Optional 即可。

代码清单 1-3 DefaultUserService

```
@Service
public class DefaultUserService implements UserService {

    @Autowired
     private Optional<UserDao> userDao;

     public User findUserByUserName(String userName) {
         if(userDao.isPresent()){
             userDao.get().findUserByUserName(userName);
         }
         return null;
     }
}
```

另一个使用 Optional 的地方是 Spring MVC 框架。例如，下面的代码表示 userName 参数是可选的，即请求参数可不包含 userName。

```
@RequestMapping("/user")
public User getUser(String id, Optional<String> userName){}
```

1.6.2　核心容器的增强

Spring 4.0 对核心容器进行了增强，支持泛型依赖注入，对 GgLib 类代理不再要求必须有空参构造器(这个特性带来很大便利)；在基于 Java 的配置里添加了@Description；提供@Conditional 注解来实现 Bean 的条件过滤；提供@Lazy 注解解决 Bean 延时依赖注入；支持 Bean 被注入 List 或者 Array 时可以通过@Order 注解或基于 Ordered 接口进行排序。如果使用 Spring 的注解支持，则可以使用自定义注解来组合多个注解，方便对外公开特定的属性。

- 泛型依赖注入：Spring 4.0 可以为子类的成员变量注入泛型类型。

```
public abstract class BaseService<M extends Serializable> {
    @Autowired
    protected BaseDao<M> dao;
    ...
}
@Service
public class UserService extends BaseService<User> {
}
@Service
public class ViewSpaceService extends BaseService<ViewSpace> {
}
```

- Map 依赖注入。

```
@Autowired
private Map<String, BaseService> map;
```

上述写法将 BaseService 类型注入 map 中。其中，key 是 Bean 的名字；value 是所有实现了 BaseService 的 Bean。

- @Lazy 延迟依赖注入。

```
@Lazy
@Service
public class UserService extends BaseService<User> {
}
```

也可以把@Lazy 放在@Autowired 之上，即依赖注入也是延时的，当调用 userService 时才会注入。同样适用于@Bean。

- List 注入。

```
@Autowired
private List<BaseService> list;
```

这样就会注入所有实现了 BaseService 的 Bean，但顺序是不确定的。在 Spring 4.0 中可以使用@Order 或 Ordered 接口来实现排序，例如：

```
@Order(value = 1)
@Service
```

```
public class UserService extends BaseService<User> {
}
```

- @Conditional 注解：@Conditional 类似于@Profile，一般用于在多个环境（开发环境、测试环境、正式机环境）中进行配置切换，即通过某个配置来开启某个环境。@Conditional 注解的优点是允许自己定义规则。可以指定在如@Component、@Bean、@Configuration 等注解的类上，以决定是否创建 Bean 等。
- CGLIB 代理类增强：在 Spring 4.0 中，基于 CGLIB 的代理类不再要求类必须有空参构造器，这是一个很好的特性。使用构造器注入有很多好处，比如，可以确保只在创建 Bean 时注入依赖，以保证 Bean 不可更改；又如，如果对 UserService 类进行事务增强，此时要求 UserService 类必须有空参构造器，就会造成很多不便。

1.6.3 支持用 Groovy 定义 Bean

Spring 4.0 支持使用 Groovy DSL 来进行 Bean 定义配置，其类似于 XML，但比 XML 更加灵活，可以通过 Groovy DSL 语法配置任何复杂的 Bean 依赖注入，但其目前也存在一些不足之处：

- Groovy DSL 错误提示不友好，给排查问题带来很多不便。
- 目前主流的 IDE 对 Groovy DSL 代码自动补全功能的支持较弱。
- Groovy DSL 语法学习曲线较高，加上目前 Spring 社区在这方面的支持力度不足，要全面掌握 DSL 配置需要一个很长的过程。

1.6.4 Web 的增强

从 Spring 4.0 开始，Spring MVC 基于 Servlet 3.0 开发，如需使用 Spring MVC 测试框架，则要依赖 Servlet 3.0 的相关.jar 包（因为 Mock 的对象都是基于 Servlet 3.0 的）。另外，为了方便 REST 开发，引入新的@RestController 控制器注解，这样就不需要在每个@RequestMapping 方法上加@ResponseBody 了。同时添加了一个 AsyncRestTemplate，支持 REST 客户端的异步无阻塞请求。

1.6.5 支持 WebSocket

Spring 4.0 的一个最大更新是增加了对 WebSocket 的支持。WebSocket 提供了一个在 Web 应用中高效、双向的通信，需要考虑到客户端（浏览器）和服务器之间的高频和低延时消息交换。一般的应用场景有在线交易、游戏、协作、数据可视化等。

使用 WebSocket 需要考虑浏览器版本（IE<10 不支持），目前主流的浏览器都能很好地支持 WebSocket。

WebSocket 有一些子协议，可以从更高的层次实现编程模型。这些子协议包括 STOMP、WAMP 等。

1.6.6　测试的增强

Spring-test 模块里的所有注解都可以用作 meta-annotation，这样就可以自定义组合注解来减少测试时的重复配置。org.springframework.mock.web 包与 Servlet 3.0 适配。

在 Spring 4.0 之前，需要通过继承 AbstractTransactionalJUnit 4SpringContextTests 类，然后调用 executeSqlScript()函数来进行测试。这里存在一个问题：如果要同时执行多个数据源的初始化，则该方法并不可靠，而且使用起来也不是很便利。Spring 4.0 提供了@Sql 注解来完成这个任务。

1.6.7　其他

Spring 4.0 提供了对 JCache（JSR-107）注解的支持，并对 Cache 抽象部分进行了增强。

Spring 4.0 添加了动态语言支持，对动态脚本语言（BeanShell、Groovy、JavaScript）计算表达式进行了抽象封装，对外统一接口 ScriptEvaluator，并废弃了对 JRuby 的支持。

Spring 4.0 添加了多线程并发处理支持，对 JDK 的 Future 进行了封装，简化了线程回调处理；提供了与 Guava 类似的 ListenableFuture 接口，通过 ListenableFutureTask 实现类可以很容易地处理线程回调。

Spring 4.0 增强了持久化处理，Transactional 支持 AspectJ，SimpleJdbcCallOperations 支持命名绑定；全面支持 Hibernate ORM 5.0，不再支持 Hibernate 3.6 以前的版本，并去除了对 iBaits 的直接支持。

1.7　Spring 子项目

打开 Spring 官方网站 http://spring.io/projects，可以看到 Spring 众多的子项目，它们构建起一个丰富的企业级应用解决方案的生态系统。在这个生态系统中，除 Spring 框架本身外，还有很多值得关注的子项目。从配置到安全，从普通 Web 应用到大数据，用户在构建应用基础设施的时候，总能从 Spring 子项目中找到一个适合自己的子项目。对 Spring 应用开发者来说，了解这些子项目，可以更好地使用 Spring；也可以通过阅读这些子项目的源代码，更深入地了解 Spring 的设计架构和实现原理。下面以表格形式对 Spring 的各个子项目进行简要介绍，如表 1-1 所示。

表 1-1 Spring子项目

子项目名称	子项目介绍
Spring IO Platform	Spring IO 是可集成的、构建现代化应用的版本平台。Spring IO 是模块化的、企业级的分布式系统，包括一系列依赖，使得开发者仅能对自己所需的部分进行完全的部署控制
Spring Boot	Spring 应用快速开发工具，用来简化 Spring 应用开发过程
Spring XD	Spring XD（eXtreme Data，极限数据）是 Pivotal 的大数据产品。它结合了 Spring Boot 和 Grails，组成 Spring IO 平台的执行部分
Spring Cloud	Spring Cloud 为开发者提供了在分布式系统（如配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性 Token、全局锁、决策竞选、分布式会话和集群状态）中操作的开发工具。使用 Spring Cloud，开发者可以快速实现上述这些模式
Spring Data	Spring Data 是为了简化构建基于 Spring 框架应用的数据访问实现，包括非关系数据库、Map-Reduce 框架、云数据服务等；另外，也包含对关系数据库的访问支持
Spring Integration	Spring Integration 为企业数据集成提供了各种适配器，可以通过这些适配器来转换各种消息格式，并帮助 Spring 应用完成与企业应用系统的集成
Spring Batch	Spring Batch 是一个轻量级的完整批处理框架，旨在帮助应用开发者构建一个健壮、高效的企业级批处理应用（这些应用的特点是不需要与用户交互，重复的操作量大，对于大容量的批量数据处理而言，这些操作往往要求较高的可靠性）
Spring Security	Spring Security 是一个能够为基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。它提供了一组可以在 Spring 应用上下文中配置的 Bean，充分利用 Spring IoC 和 AOP 功能，为应用系统提供声明式的安全访问控制功能，减少了为企业系统安全控制编写大量重复代码的工作
Spring Hateoas	Spring Hateoas 是一个用于支持实现超文本驱动的 REST Web 服务的开发库，是 Hateoas 的实现。Hateoas（Hypermedia as the engine of application state）是 REST 架构风格中最复杂的约束，也是构建成熟 REST 服务的核心。它的重要性在于打破了客户端和服务器之间严格的契约，使得客户端可以更加智能和自适应，而 REST 服务本身的演化和更新也变得更加容易
Spring Social	Spring Social 是 Spring 框架的扩展，用来方便开发 Web 社交应用程序，可通过该项目来创建与各种社交网站的交互，如 Twitter、Facebook、LinkedIn 和 TripIt 等
Spring AMQP	Spring AMQP 是基于 Spring 框架的 AMQP 消息解决方案，提供模板化的发送和接收消息的抽象层，提供基于消息驱动的 POJO。这个项目支持 Java 和.NET 两个版本。Spring Source 旗下的 Rabbit MQ（Erlang 语言开发）就是一个开源的基于 AMQP 的消息服务器
Spring for Android	Spring for Android 为 Android 终端开发应用提供 Spring 的支持，它提供了一个在 Android 应用环境中工作、基于 Java 的 REST 客户端
Spring Mobile	Spring Mobile 是基于 Spring MVC 构建的，为移动终端的服务器应用开发提供支持。比如，使用 Spring Mobile 可以在服务器端自动识别连接到服务器的移动终端的相关设备信息，从而为特定的移动终端实现应用定制
Spring Web Flow	Spring Web Flow（SWF）一个建立在 Spring MVC 基础上的 Web 页面流引擎。随着其自身项目的发展，Web Flow 比原来更为丰富，SWF 定义了一种特定的语言来描述页面流。其目标是成为管理 Web 应用页面流程的最佳方案。当你的应用需要复杂的导航控制，如向导，在一个比较大的事务过程中指导用户经过一连串的步骤的时候，SWF 是一个很好的解决方案框架
Spring Web Services	Spring Web Services（Spring WS）是基于 Spring 框架的 Web 服务框架，主要侧重于基于文档驱动的 Web 服务，提供 SOAP 服务开发，允许通过多种方式创建 Web 服务

续表

子项目名称	子项目介绍
Spring LDAP	Spring LDAP 是一个用于操作 LDAP 的 Java 框架，类似于 Spring JDBC 提供了 JdbcTemplate 方式来操作数据库。这个框架提供了一个 LdapTemplater 操作模板，可帮助开发人员简化 looking up、closing contexts、encoding/decoding values、filters 等操作
Spring Session	Spring Session 致力于提供一个公共基础设施会话，支持从任意环境中访问一个会话。在 Web 环境下支持独立于容器的集群会话，支持可插拔策略来确定 Session ID，WebSocket 活跃的时候可以简单地保持 HttpSession
Spring Shell	Spring Shell 提供交互式的 Shell，用户可使用简单的基于 Spring 的编程模型来开发命令

1.8　如何获取 Spring

在开始学习和使用 Spring 之前，必须先获取 Spring 的发布包。Spring 在以下 4 个地方提供发布版本的下载。

- Spring 下载社区：spring.io 自建的下载社区（http://spring.io/projects），不但提供 Spring 框架的下载，还提供 Spring 所有子项目的下载。
- Maven 中心：即 Maven 工具默认访问的仓库，许多 Spring 依赖的第三方类库也可以从这里获取。
- 企业模块仓库（Enterprise Bundle Repository，EBR）：是由 SpringSource 公司自己维护的一个企业模块仓库，类似于 Maven 的仓库。它拥有 Spring 所涉及的所有类库。不但可以被 Maven 使用，也可以被 Gradle 使用（基于 Groovy DSL 自动化构建工具，Spring 框架源码采用 Gradle 来构建）。
- Maven 公共仓库：众多 Maven 仓库都可获取到，如 https://repository.sonatype.org 等。

目前，Spring 不再提供直接下载方式，只能使用 Maven 或 Gradle 构建来下载 Spring 的构建包。在项目开发中，可通过配置 Maven 工程中的 pom.xml 文件来下载 Spring 相应的构建包（spring-context），如代码清单 1-4 所示。

代码清单 1-4　在pom.xml中配置Spring依赖

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>4.2.2.RELEASE</version>
   </dependency>
</dependencies>
```

Spring 框架的文档可以通过在线方式直接浏览或下载 PDF。

- 在线文档地址：http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/。

- 下载 PDF 文档地址：http://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/spring-framework-reference.pdf。

Spring 所有的子项目源码和实例工程代码都托管在 GitHub，可以通过 Git 客户端 TortoiseGit 到下列地址下载：

- 从 https://github.com/spring-projects/spring-framework 下载框架源码。
- 从 https://github.com/spring-projects/spring-petclinic 下载实例源码。

如果不想安装 Git 客户端，可以通过项目正式发布版本列表选择相应版本，直接单击下载，如图 1-2 所示。

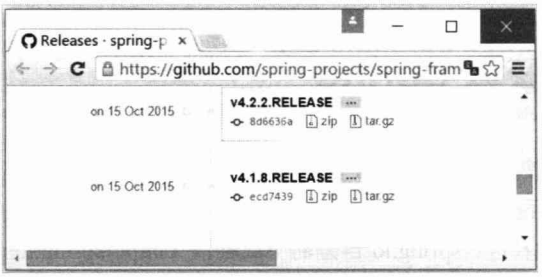


图 1-2　Spring 下载地址

1.9　小结

全世界成千上万的项目都构建于 Spring 技术框架之上，Spring 已然成为事实上标准的 Java 技术框架。它颠覆了传统 Java 开发笨重难用的学院派风格，给 Java 开发者带来了一股敏捷便利、灵活实用的编程之风。

Spring 4.0 在核心容器、Web、测试、缓存、数据访问等方面进行了重大升级，全面支持 Java 8.0、WebSocket、Groovy 动态语言等，项目源码以当前灵活的 Gradle 构建工具进行组织，进一步增强了 Spring 在 Java 开源领域第一开源框架的领导地位。

第 2 章

快速入门

本章通过一个简单的例子展现开发 Spring Web 应用的整体过程，通过这个实例，读者可以快速跨入 Spring Web 应用的世界。实例应用按持久层、业务层和展现层进行组织，从底层 DAO 到 Web 展现逐层演进，一步步地搭建起一个完整的实例。通过本章的学习，读者可以独立完成一个典型的基于 Spring 的 Web 应用。

本章主要内容：

- Maven 构建工具介绍
- 用户登录实例介绍
- 基于 Spring JDBC 的持久层实现
- 基于 Spring 声明式事务的业务层实现
- 基于 Spring MVC 的展现层实现
- 在 IDEA 中开发 Web 应用的过程
- 运行 Web 应用

本章亮点：

- 非传统 Hello World 的快速入门实例
- 通过 IDEA 开发工具讲解开发的过程
- 详尽的开发过程，使读者快速上手

2.1 实例概述

在进行实例项目具体开发之前，有必要先对项目的功能进行概述，以便对要实现的项目有一个整体性的认识。

2.1.1 比 Hello World 更适用的实例

为了让大家快速对 Spring 有一个切身的认识，没有什么比通过一个实际的例子更适合了。Hello World 是比较经典的入门实例，但笔者认为 Hello World 太过简单，很难展现 Spring 的全貌。为了让 Spring 的功能轮廓更加清晰，笔者试图通过一个功能涵盖面更广的论坛登录模块替换经典的 Hello World 实例。之所以选择登录功能模块，出于以下 3 种原因：

（1）读者对于登录模块的业务功能很熟悉，无须在业务功能介绍上花费时间。

（2）登录模块“麻雀虽小，五脏俱全”，它涵盖了持久层数据访问操作、业务层事务管理及展现层 MVC 等企业应用常见的功能。

（3）本书希望通过一个名为“小春”的论坛贯穿始终，以便能够由点及面，使读者在单纯技术性学习的酣战中深刻理解应用程序的整体开发流程。

Spring 拥有持久层、业务层和展现层的“原生技术”，分别是 Spring JDBC、声明式事务和 Spring MVC。为了充分展现 Spring 本身的魅力，本章仅使用 Spring 的这些“原生技术”，在后续章节中将学习其他的持久层和展现层技术，只要用户愿意，就可以平滑地将其过渡到其他实现技术中。

2.1.2 实例功能简介

论坛登录模块的功能很简单，首先登录页面提供一个带用户名/密码的输入表单，用户填写并提交表单后，服务器端程序检查是否有匹配的用户名/密码。如果用户名/密码不匹配，则返回登录页面，并给出提示；如果用户名/密码匹配，则记录用户的成功登录日志，更新用户的最后登录时间和 IP，并给用户增加 5 个积分，然后重定向到欢迎页面，如图 2-1 所示。

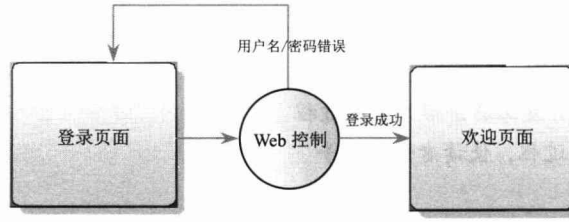


图 2-1 页面流程

在持久层拥有两个 DAO 类，分别是 UserDao 和 LoginLogDao，在业务层对应一个业务类 UserService，在展现层拥有一个 LoginController 类和两个 JSP 页面，分别是登录页面 login.jsp 和欢迎页面 main.jsp。

下面通过一张时序图来描述论坛登录模块的整体交互流程，如图 2-2 所示。

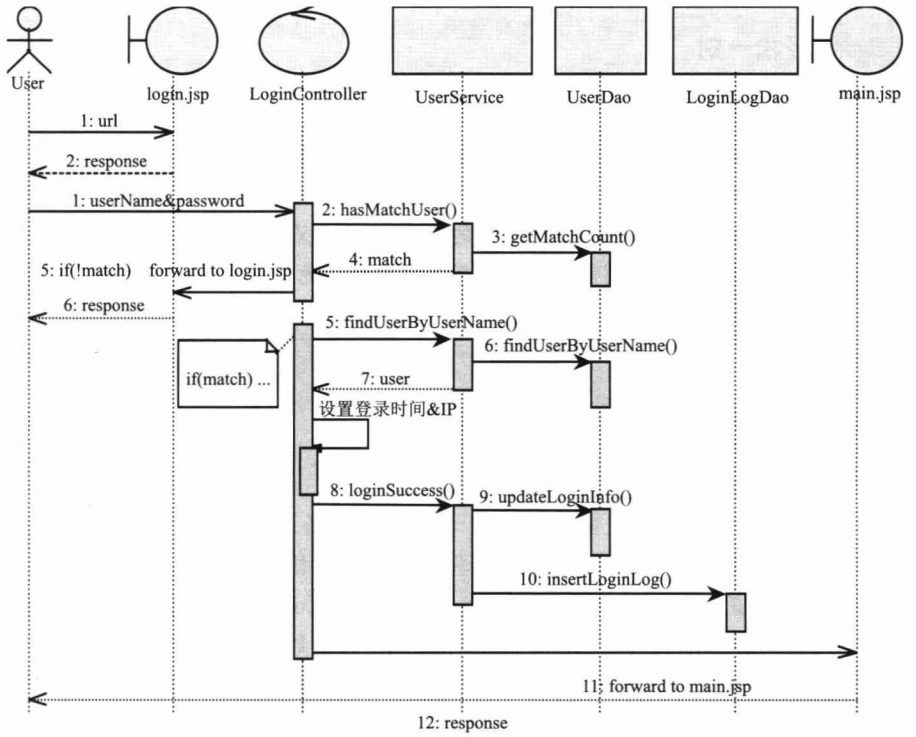


图 2-2　登录模块整体流程

（1）用户访问 login.jsp，返回带用户名/密码表单的登录页面。

（2）用户在登录页面输入用户名/密码，提交表单到服务器，Spring 根据配置调用 LoginController 控制器响应登录请求。

（3）LoginController 调用 UserService#hashMatchUser()方法，根据用户名和密码查询是否存在匹配的用户，UserService 内部通过调用持久层的 UserDao 完成具体的数据库访问操作。

（4）如果不存在匹配的用户，则重定向到 login.jsp 页面，并报告错误；否则进入下一步。

（5）LoginController 调用 UserService#findUserByUserName()方法，加载匹配的 User 对象，并更新用户最近一次登录时间和登录 IP。

（6）LoginController 调用 UserService#loginSuccess()方法，进行登录成功的业务处理：首先调用 UserDao#updateLoginInfo()方法为用户添加 5 个积分，然后创建一个 LoginLog 对象，并利用 LoginLogDao 将其插入数据库中。

（7）重定向到欢迎页面 main.jsp，欢迎页面产生响应返回给用户。

实例的所有程序位于配套网盘 chapter2 的目录下，本章后面的内容将逐一实现以上步骤的功能，完成这个实例的所有细节。

轻松一刻

虽然本书很少采用 foo 和 bar 对变量进行命名，但相信读者对于这两个单词再熟悉不过了，它们是计算机图书中经常使用的变量名。不同的字典对 foo 的解释相去甚远，一说来自中国“福”字的发音，又有解释为二战时期的一种武器。将 foo 和 bar 组合在一起所构成的 foobar 应该最能反映其原始的意思：foobar 又为 foo-bar，其中 bar 是 beyond all recognition 的缩写，意为超越认知，通俗点说就是“无法识别、一塌糊涂”的意思。而 foo 是 fu 的变体，fu 是英语习语 fuck-up 的缩写，同样是“一团糟”的意思。

2.2 环境准备

在进入实例的具体开发之前，需要做一些环境的准备工作，其中包括数据库表的创建、项目工程的创建、规划 Spring 配置文件等。本书采用 MySQL 5.x 数据库，如果用户机器中还未安装该数据库，则可以从 http://www.mysql.org/downloads 下载并安装。为了方便读者运行本书示例代码，本书所有章节示例中连接数据库的用户名统一采用 root，密码统一采用 123456。如果读者安装的 MySQL 数据库的 root 用户的密码与本书不一致，则在运行本书工程示例时，需要对工程连接数据库信息进行相应调整。

提示

MySQL 4.1.0 以前的版本不支持事务，MySQL 4.1.0 本身也只对事务提供有限的支持，Spring 的各种声明式事务需要底层数据库的支持，所以最好安装 5.0 或更高版本的 MySQL。各版本主要增加的特性如下：

MySQL 5.0 增加储存过程、视图、游标、触发器、XA 事务。

MySQL 5.1 增加事件调度器、分区、可插拔的存储引擎 API、行复制、全局动态查询日志修改。

MySQL 5.5 默认存储引擎更改为 InnoDB，提高了默认线程并发数，后台输入/输出线程控制，主线程输入/输出速率控制，操作系统内存分配程序使用控制，适应性散列索引控制，恢复组提交，多缓冲池实例，半同步复制，中继日志自动恢复，建立快速索引，高效的数据压缩等特性。

MySQL 5.6 中 InnoDB 性能加强，InnoDB 死锁信息可以记录到错误日志，支持主从延时复制，增强行级复制功能，基于 CRC32 校验的复制事件等。

2.2.1 构建工具 Maven

Maven 是 Apache 的一个顶级项目（http://maven.apache.org），也是一款强大的构建

工具，能够帮助用户建立一套有效的自动化构建体系。从清理、编译、测试到生成报告，再到打包和部署，无须一遍又一遍地输入命令，一次又一次地单击鼠标，只需按照 Maven 提供的 POM 配置好项目模块间的相互依赖关系及相关的 Maven 插件，然后输入简单的命令（如 mvn clean install），Maven 就能完成那些烦琐的构建任务。读者若想深入学习，可以参考《Maven 实战》一书。Maven 构建的模型如图 2-3 所示。

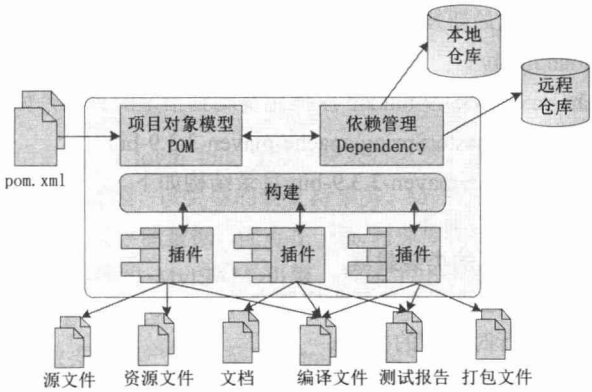


图 2-3　Maven 模型

1. Maven 基础概念

- **Project**：任何你想构建的事务，Maven 都可以认为它们是工程。这些工程被定义为工程对象模型（Project Object Model，POM）。一个工程可以依赖其他的工程，一个工程也可以由多个子工程构成。
- **POM**：POM（pom.xml）是 Maven 的核心文件，它是指示 Maven 如何工作的元数据文件，类似于 Ant 中的 build.xml 文件。POM 文件位于每个工程的根目录中。
- **GroupId**：GroupId 是一个工程在全局中的唯一标识符，一般地，它就是工程名。GroupId 有利于使用一个完全的包名将一个工程从其他有类似名称的工程中区别出来。
- **Artifact**：中文名为“构件”，是工程将要产生或需要使用的文件，它可以是.jar 文件、源文件、二进制文件、.war 文件，甚至是.pom 文件。每个 Artifact 都由 GroupId 和 ArtifactId 组合的标识符唯一识别。需要被使用的 Artifact 都要放在仓库（见 Repository）中，否则 Maven 无法找到它们。
- **Dependency**：为了能够构建或运行，一个典型的 Java 工程会依赖其他的包。在 Maven 中，这些被依赖的包就被称为 Dependency。Dependency 一般是其他工程的 Artifact。
- **Plug-in**：可以说 Maven 就是一堆插件的集合，它的每一个功能都是由插件完成的。插件提供 goal（类似于 Ant 中的 target），并根据在 POM 中找到的元数据去完成工作。主要的 Maven 插件是由 Java 编写而成的，同时支持用 Beanshell 或 Ant 脚本编写的插件。

- **Repository**：仓库，即放置 Artifact 的地方，有中央仓库、公共仓库、私有仓库及本地仓库之分。为了提高 Artifact 的下载速度，一般情况下，公司或开发者组织都需要部署一个私有仓库，可使用 Nexus（http://www.sonatype.org/nexus）创建 Maven 私有仓库。

2. Maven 安装

（1）从官方网站下载最新发布版本（本书下载版本为 3.3.9），下载地址为 http://maven.apache.org/download.html。

（2）把 apache-maven-3.3.9-bin.zip 压缩包复制到指定的目录，如 D:\ masterSpring，并解压缩到当前目录 D:\masterSpring\apache-maven-3.3.9-bin。

D:\masterSpring\apache-maven-3.3.9-bin 目录结构如下。

- bin：Maven 的运行脚本。
- boot：Maven 自己的类装载器。
- conf：该目录下包含了全局行为定制文件 setting.xml。
- lib：Maven 运行时所需的类库。

（3）设置“JAVA_HOME”环境变量，指定 JDK 安装目录。

（4）设置“M2_HOME”环境变量，如 D:\masterSpring\apache-maven-3.3.9-bin。

（5）编辑“Path”环境变量，把 Maven 的 bin 目录（%M2_HOME %\bin）添加到当前环境变量，方便后续 Maven 命令的使用。

（6）设置 MAVEN_OPTS=-Xms 512m –Xmx1024m（非必要项，但可防止内存溢出）。

（7）检查安装正确性。在命令行窗口中输入“mvn –v”，如果能看到 Maven 和 JDK 的版本号，则说明安装正确。

2.2.2 创建库表

（1）启动 MySQL 数据库后，用 DOS 命令窗口登录数据库。

```
mysql>mysql -uroot -p123456
```

分别指定用户名和密码，MySQL 默认运行在 3306 端口。如果 MySQL 没有运行在默认端口，则需要通过--port 参数指定端口号。

（2）运行以下脚本，创建实例对应的数据库。

```
mysql>DROP DATABASE IF EXISTS sampledb;
mysql>CREATE DATABASE sampledb DEFAULT CHARACTER SET utf8;
mysql>USE sampledb;
```

数据库名为 sampledb，默认字符集采用 UTF-8。

（3）创建实例所用的两张表。

```
#创建用户表
mysql>CREATE TABLE t_user (
      user_id   INT AUTO_INCREMENT PRIMARY KEY,
      user_name VARCHAR(30),
```

```
    credits INT,
    password  VARCHAR(32),
    last_visit datetime,
    last_ip  VARCHAR(23)
)ENGINE=InnoDB;

#创建用户登录日志表
mysql>CREATE TABLE t_login_log (
    login_log_id  INT AUTO_INCREMENT PRIMARY KEY,
    user_id   INT,
    ip  VARCHAR(23),
    login_datetime datetime
)ENGINE=InnoDB;
```

t_user 为用户信息表；t_login_log 为用户登录日志表。其中，ENGINE=InnoDB 指定了表的引擎为 InnoDB 类型，该类型的表支持事务。MySQL 默认采用 MyISAM 引擎，该类型的表不支持事务，仅存储数据，优点在于读/写速度快。对于论坛型应用系统的表来说，可以使用不支持事务的 MyISAM 引擎，但本书出于演示事务的目的，所有表均采用支持事务的 InnoDB 引擎。

（4）初始化一条数据，用户名/密码为 admin/123456。

```
#插入初始化数据
mysql>INSERT INTO t_user (user_name,password) VALUES('admin','123456');
mysql>COMMIT;
```

用户也可以通过直接运行脚本文件完成以上所有工作。创建数据库表的脚本文件位于 chapter2/schema/sampledb.sql。下面提供了两种运行脚本的方法。

- 直接通过 mysql 命令运行。

假设将网盘中的内容复制到 D:\masterSpring 目录下，则在 DOS 命令窗口下，运行以下命令：

```
D:\> mysql -u root -p123456 --port 3306
            < D:\masterSpring\code\chapter2\schema\sampledb.sql
```

- 在登录 MySQL 后，通过 source 命令运行脚本。

```
mysql>source D:\masterSpring\code\chapter2\schema\sampledb.sql
```

2.2.3　建立工程

考虑到 IntelliJ IDEA 是一款非常优秀且强大的 IDE 工具，越来越受到广大开发人员的欢迎，本书的所有示例都采用 IDEA 进行开发（采用 14.0 版本，目前分为商业版和社区版两种，其中社区版是免费的，可以到 http://www.jetbrains.org 下载）。将 IDEA 的工作空间设置于 D:\masterSpring。为了避免因路径不一致引起的各种问题，请尽量保证工作空间和本书的路径一致。网盘中的源文件和配置文件都使用 UTF-8 编码格式，UTF-8 可以很好地解决国际化问题，同时避免不受欢迎的中文乱码问题。用户可以通过 File→Settings→File Encodings 命令将 IDEA 的工作空间编码格式设置为 UTF-8，如图 2-4 所示。

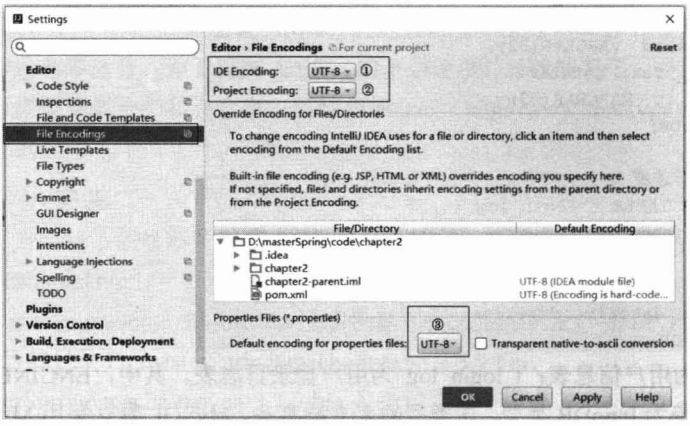


图 2-4　IDEA 工程编码设置

需要特别说明的是，图 2-4 中的①、②、③处编码必须全部设置为 UTF-8，否则会影响工程代码的正常编译。**后续章节示例工程编码都采用 UTF-8，设置方式与此一致。**

在 IDEA（File→New Project）中新建一个 Maven 项目，选择工程类型为 Maven，并选择工程 SDK 为 Java 7.0。如果没有，则单击右侧的 New...按钮，选择本地已安装 Java 7.0 的目录即可，如图 2-5 所示。

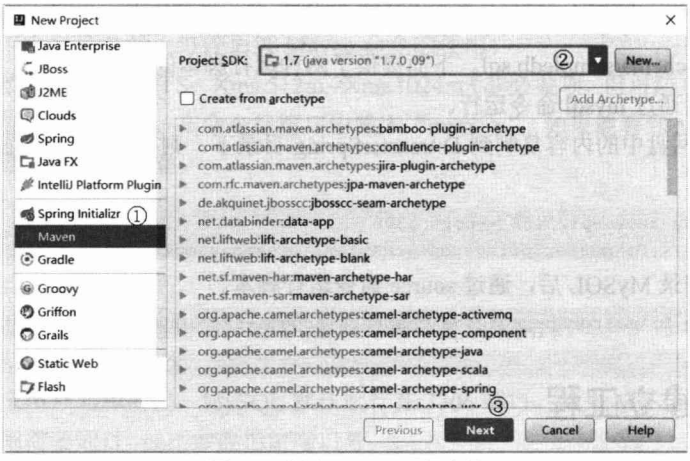


图 2-5　创建 Maven 工程向导

单击 Next 按钮，进入 Maven 工程设置向导，分别填写 GroupId，ArticfactId、Version 信息。本书的示例，GroupId 统一设置为 com.smart，章节模块名称设置为 chapterX（X 为对应章节的数字），Version 统一设置为 1.0，如图 2-6 所示。

设置 Maven 基本信息之后，单击 Next 按钮，进入工程信息配置界面，设置工程名称、目录及模块名称等信息。本书的示例工程路径统一设置为 D:\masterSpring\code，如图 2-7 所示。

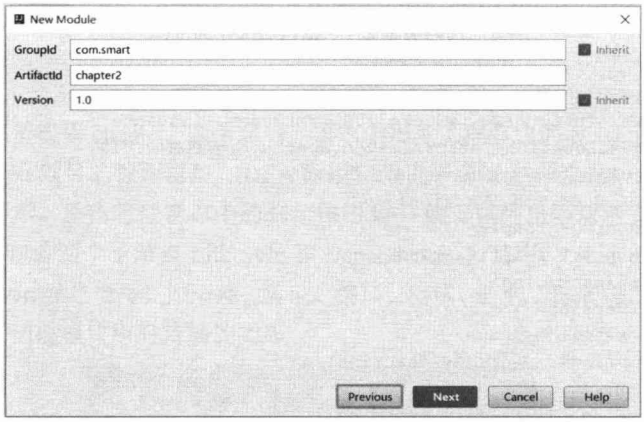


图 2-6　设置 Maven 信息

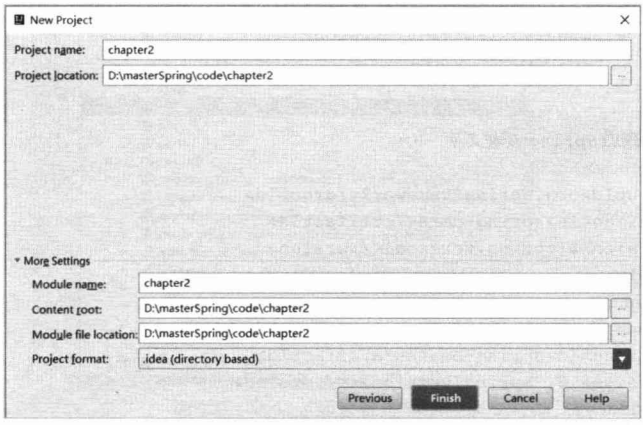


图 2-7　设置工程信息

单击 Finish 按钮完成本章节模块的创建。创建之后工程模块的目录结构如图 2-8 所示。

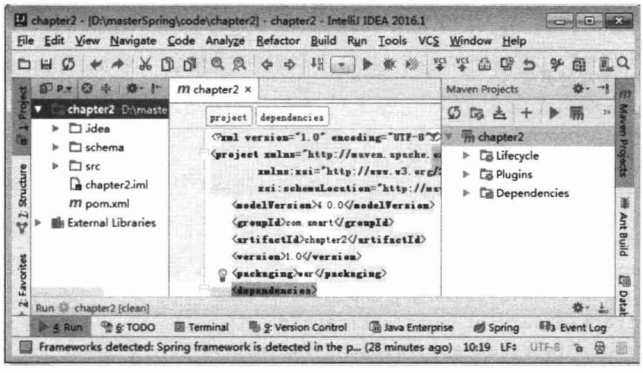


图 2-8　Maven 工程目录结构

创建示例工程之后，需要在 pom.xml 文件中配置 Spring、数据源、数据库连接驱动、Servlet 类库的依赖信息，如代码清单 2-1 所示。

代码清单 2-1　根模块pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.smart</groupId>
 <artifactId>chapter2 </artifactId>
 <packaging>war</packaging>
 <version>1.0</version>
 <name>Spring4.x 实战示例</name>
 <description>Spring4.x 实战示例</description>
 <properties>
     <file.encoding>UTF-8</file.encoding>
     <spring.version>4.2.1.RELEASE</spring.version>
     <mysql.version>5.1.29</mysql.version>
     <servlet.version>2.5</servlet.version>
       …
</properties>
<dependencies>
        <!—依赖的 Spring 模块类库 -->
       <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-core</artifactId>
          <version>${spring.version}</version>
       </dependency>
       <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-beans</artifactId>
          <version> ${ spring.version} </version>
       </dependency>
         <!-- 依赖的数据库驱动类库-->
       <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>${mysql.version}</version>
       </dependency>
       <!--依赖的连接池类库-->
       <dependency>
          <groupId>commons-dbcp</groupId>
          <artifactId>commons-dbcp</artifactId>
          <version>${commons-dbcp.version}</version>
       </dependency>
       <!--依赖的 Web 类库-->
       <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>servlet-api</artifactId>
          <version>${servlet-api.version}</version>
       </dependency>
       <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>jstl</artifactId>
          <version>${jstl.version}</version>
```

```
        </dependency>
        ...
</dependencies>
</project>
```

配置了章节子模块 chapter2 的信息后，单击 IDEA 工程右边的 Maven Projects 选项卡，将弹出 Maven 项目的管理窗口。IDEA 为用户提供了非常友好的 Maven 项目管理界面，如图 2-9 所示。单击管理窗口中的刷新按钮，就可以列出当前所有的 Maven 模块。每个模块都包含两个子节点 Lifecycle 和 Dependencies，其中，Lifecycle 子节点下提供了常用的 Maven 操作命令，如清理、验证、编译、测试、打包、部署等；Dependencies 子节点下列出了当前模块所有依赖的类库。

图 2-9　Maven 项目管理界面

基于 Maven 的模块化工程与传统的单一模块工程相比，最大的不同之处就是代码打包及运行方式。若是初次导入或打开 Maven 工程，则可能会在③、④处出现红色的下画线，读者不用担心，那是由于工程模块的依赖包未下载到本地。此时只需要在②处同时选中“clean”、“install”命令，并单击工具栏上的运行命令按钮，如①-2 所示，Maven 就会到远程中心仓库将工程所有的依赖 JAR 包下载到本地（默认存储在 C:\Users\用户名\.m2）。值得一提的是，最好将工具栏的选中，如①-1 所示，以便跳过单元测试；否则会自动运行单元测试，不但影响编译打包效率，而且如果单元测试有问题，那么整个构建过程会随之退出。本书所有章节的示例工程都采用独立的 Maven 工程进行组织，读者在运行章节工程的时候，需要按照此种方式编译、打包工程。

实战经验

在实际项目中，一个项目一般划分为多个子模块。为了简化每个模块对第三方依赖的配置管理，在创建项目工程的时候，会创建一个 Maven 的根模块，用来管理每个子模

块的公共依赖配置，在各个子模块中直接继承根模块的配置即可。由于本书各个章节的示例代码比较简单，为了方便读者对代码的阅读和调试，本书所示的章节示例工程不采用 Maven 继承配置。

2.2.4 类包及 Spring 配置文件规划

1. 类包规划

类包以分层的方式进行组织，共划分为 4 个包，分别是 dao、domain、service 和 web。领域对象从严格意义上讲属于业务层，但由于领域对象可能同时被持久层和展现层共享，所以一般将其单独划分到一个包中，如图 2-10 所示。

单元测试的类包和程序的类包对应，但放置在不同的文件夹下，如图 2-11 所示。本实例仅对业务层的业务类进行单元测试。将程序类和测试类放置在物理不同的文件夹下，方便将来程序的打包和分发，因为测试类仅在开发时有用，无须包含到部署包中。在本章的后续内容中，读者将会看到如何用 Maven 工具进行程序打包，体会这种包及目录的设计结构所带来的好处。

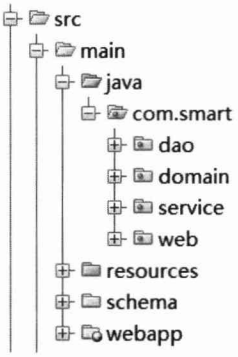


图 2-10 主程序包的规划

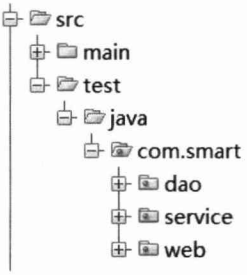


图 2-11 测试包的规划

实战经验

随着项目规模的增大，这种仅以分层思想规划的包结构就会显示出它的不足。一般情况下，需要在业务模块包下进一步按分层模块划分子包，如 user/dao、user/service、viewspace/dao、viewspace/service 等。对于由若干独立的子系统组成的大型应用，在业务分层包的前面一般还需要加上子系统的前缀。包的规划对于大型应用非常重要，它直接关系到应用部署和分发的便利性。

2. Spring 配置文件规划

Spring 可以将所有的配置信息统一到一个文件中，也可以放置到多个文件中。对于简单的应用来说，由于配置信息少，仅用一个配置文件就足以应付。随着应用规模的扩

大、配置信息量的增多，仅使用一个配置文件往往难以满足要求。如果不进行仔细规划，则将给配置信息的查看和团队协作带来负面影响。

配置文件在团队协作时是资源争用的焦点，对于大型应用一般要按模块进行划分，以在一定程度上降低争用，减少团队协作的版本控制冲突。由于我们的应用比较小，所以直接采用一个配置文件 spring-context.xml 即可。

2.3　持久层

持久层负责数据的访问和操作，DAO 类被上层的业务类调用。Spring 本身支持多种流行的 ORM 框架（第 14 章对此进行专门的讲解），这里使用 Spring JDBC 作为持久层的实现技术（关于 Spring JDBC 的详细内容，请参见第 13 章）。为了方便阅读，我们会对本章涉及的相关知识点进行必要的介绍，所以在不了解 Spring JDBC 的情况下，相信读者也可以轻松阅读本章的内容。

2.3.1　建立领域对象

领域对象（Domain Object）也被称为实体类，它代表了业务的状态，且贯穿展现层、业务层和持久层，并最终被持久化到数据库中。领域对象使数据库表操作以面向对象的方式进行，为程序扩展带来了更大的灵活性。领域对象不一定等同于数据库表，不过对于简单的应用来说，领域对象往往拥有对应的数据库表。

持久层的主要工作就是从数据库表中加载数据并实例化领域对象，或将领域对象持久化到数据库表中。论坛登录模块需要涉及两个领域对象：User 和 LoginLog，前者代表用户信息，后者代表日志信息，分别对应 t_user 和 t_login_log 数据库表，领域对象类的包为 com.smart.domain。

提示

领域模型中的实体类可细分为 4 种类型：VO、DTO、DO、PO。PO（Persistent Object）：持久化对象，表示持久层的数据结构（如数据库表）；DO（Domain Object）：领域对象，即业务实体对象；DTO（Data Transfer Object）：数据传输对象，原来的目的是为 EJB 的分布式应用提供粗粒度的数据实体，以降低分布式调用的次数，提高分布式调用的性能，后来一般泛指用于展示层与服务层之间的数据传输对象，因此可以将 DTO 看成一个组合版的 DO；VO（View Object）：视图对象，用于展示层视图状态对应的对象。从分层角度来说，PO、DO/DTO、VO 分别属于持久层、服务层和展现层。对于简单模块来说，有时 PO、DO 和 VO 并没有什么区别，这时就没有必要分别定义 DO 和 VO 了，直接复用 PO 即可。

1. 用户领域对象

用户信息领域对象很简单，可以看成对 t_user 表的对象映象，每个字段对应一个对象属性。User 类主要有 3 类信息，分别为用户名/密码（userName/password）、积分（credits）及最后一次登录的信息（lastIp、lastVisit），如代码清单 2-2 所示。

代码清单 2-2　User.java领域对象

```
package com.smart.domain;
import java.io.Serializable;
import java.util.Date;

//①领域对象一般要实现 Serializable 接口，以便可以序列化
public class User implements Serializable{
    private int userId;
    private String userName;
    private String password;
    private int credits;
    private String lastIp;
    private Date lastVisit;

    //省略 get/setXxx 方法
    ...
}
```

2. 登录日志领域对象

用户每次登录成功后，都会记录一条登录日志，该登录日志包括 3 类信息，分别是用户 ID、登录 IP 和登录时间。一般情况下，还必须包括退出时间。为了简化实例，我们仅记录登录时间。登录日志的领域对象如代码清单 2-3 所示。

代码清单 2-3　LoginLog.java

```
package com.smart.domain;
import java.io.Serializable;
import java.util.Date;
public class LoginLog implements Serializable {
    private int loginLogId;
    private int userId;
    private String ip;
    private Date loginDate;
   //省略 get/setXxx 方法
}
```

2.3.2　UserDao

首先来定义访问 User 的 DAO，它包括 3 个方法。

- getMatchCount()：根据用户名和密码获取匹配的用户数。等于 1 表示用户名/密码正确；等于 0 表示用户名或密码错误（这是最简单的用户身份认证方法，在实际应用中需要采用诸如密码加密等安全策略）。
- findUserByUserName()：根据用户名获取 User 对象。

❑ updateLoginInfo()：更新用户积分、最后登录 IP 及最后登录时间。

下面通过 Spring JDBC 技术实现这个 DAO 类，如代码清单 2-4 所示。

代码清单 2-4　UserDao

```
package com.smart.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
import org.springframework.stereotype.Repository;
import com.smart.domain.User;

@Repository //①通过 Spring 注解定义一个 DAO
public class UserDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired //②自动注入 JdbcTemplate 的 Bean
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public int getMatchCount(String userName, String password) {
        String sqlStr = " SELECT count(*) FROM t_user "
                + " WHERE user_name =? and password=? ";
        return jdbcTemplate.queryForInt(sqlStr, new Object[] { userName, password });
    }
    ...
}
```

在 Spring 2.5 以后，可以使用注解的方式定义 Bean。较之于 XML 配置方式，注解配置方式的简单性非常明显，已经被广泛接受，成为一种趋势。所以除非没有办法，否则我们应尽量采用注解的配置方式。

这里我们用@Repository 定义了一个 DAO Bean，使用@Autowired 将 Spring 容器中的 Bean 注入进来（关于 Spring 的注解配置，将在第 4 章详细讲述）。

传统的 JDBC API 太底层，即使用户执行一条最简单的数据查询操作，都必须执行如下过程：获取连接→创建 Statement→执行数据操作→获取结果→关闭 Statement→关闭结果集→关闭连接，除此之外还需要进行异常处理的操作。如果使用传统的 JDBC API 进行数据访问操作，则可能会产生 1/3 以上单调乏味的重复性代码。

Spring JDBC 对传统的 JDBC API 进行了薄层封装，将样板式的代码和那些必不可少的代码进行了分离，用户仅需要编写那些必不可少的代码，剩余的那些单调乏味的重复性工作则交由 Spring JDBC 框架处理。简单来说，Spring JDBC 通过一个模板类 org.springframework.jdbc.core.JdbcTemplate 封装了样板式的代码，用户通过模板类就可以轻松地完成大部分数据访问操作。

如 getMatchCount()方法，我们仅提供了一个查询 SQL 语句，直接调用模板的 queryForInt()方法就可以获取查询，用户不用担心获取连接、关闭连接、异常处理等烦琐的事务。

通过 JdbcTemplate 的支持，我们可以轻松地实现 UserDao 的另外两个接口，如代码清单 2-5 所示。

代码清单 2-5　UserDao的另外两个接口

```
package com.smart.dao.jdbc;
...
@Repository
public class UserDao {
    ...
    //①根据用户名查询用户的 SQL 语句
    private final static String MATCH_COUNT_SQL = " SELECT count(*) FROM " +
            " t_user WHERE user_name =? and password=? ";

    private final static String UPDATE_LOGIN_INFO_SQL = " UPDATE t_user SET " +
            " last_visit=?,last_ip=?,credits=?  WHERE user_id =?";

   public User findUserByUserName(final String userName) {


        final User user = new User();
        jdbcTemplate.query(MATCH_COUNT_SQL, new Object[] { userName },
                //②匿名类方式实现的回调函数
                new RowCallbackHandler() {
                    public void processRow(ResultSet rs) throws SQLException {
                        user.setUserId(rs.getInt("user_id"));
                        user.setUserName(userName);
                        user.setCredits(rs.getInt("credits"));
                    }
                });
        return user;
    }

    public void updateLoginInfo(User user) {
        jdbcTemplate.update(UPDATE_LOGIN_INFO_SQL, new Object[] { user.getLastVisit(),
                user.getLastIp(),user.getCredits(),user.getUserId()});
    }
}
```

findUserByUserName()方法稍微复杂一些。这里，我们使用了 JdbcTemplate#query()方法，该方法的签名为 query(String sql,Object[] args, RowCallbackHandler rch)，它有 3 个入参。

- sqlStr：查询的 SQL 语句，允许使用带“?”的参数占位符。
- args：SQL 语句中占位符对应的参数数组。
- RowCallbackHandler：查询结果的处理回调接口，该回调接口有一个方法 processRow(ResultSet rs)，负责将查询的结果从 ResultSet 装载到类似于领域对象的对象实例中。

在②处，findUserByUserName()通过匿名内部类的方式定义了一个 RowCallbackHandler 回调接口实例，将 ResultSet 转换为 User 对象。

updateLoginInfo()方法比较简单，主要通过 JdbcTemplate#update(String sql,Object[]) 进行数据的更新操作。

实战经验

在 DAO 中编写 SQL 语句时，通常将 SQL 语句写在类静态变量中，这样会使代码更具有可读性。如果编写的 SQL 语句比较长，那么一般会采用多行字符串的方式进行构造，如代码清单 2-5①处所示。在编写多行 SQL 语句时，由于上下行最终会组成一行完整的 SQL 语句，因而这种拼接方式很容易产生错误的 SQL 组合语句：假设在①处的第一行的 FROM 后不加空格，在第二行的 t_user 之前也无空格，组合的 SQL 将为“... FROMt_user ...”，由于 FROM 保留字和 t_user 连在一起，就产生了非法的 SQL 语句。以下是一种规避这种潜在错误的值得推荐的编程习惯：在每行 SQL 语句的句前和句尾都加一个空格，这样就可以避免分行 SQL 语句组合后的错误。

2.3.3　LoginLogDao

LoginLogDao 负责记录用户的登录日志，它仅有一个 insertLoginLog()接口方法。与 UserDao 相似，其实现类也通过 JdbcTemplate#update(String sql ,Object[] args)方法完成登录日志插入的操作，如代码清单 2-6 所示。

代码清单 2-6　LoginLogDao

```
package com.smart.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import com.smart.domain.LoginLog;

@Repository
public class LoginLogDao {
    private JdbcTemplate jdbcTemplate;

    //保存登录日志 SQL
    private final static String INSERT_LOGIN_LOG_SQL= "INSERT INTO
                                    t_login_log(user_id,ip,login_datetime)
VALUES(?,?,?)";

    public void insertLoginLog(LoginLog loginLog) {
        Object[] args = { loginLog.getUserId(), loginLog.getIp(),loginLog.
getLoginDate() };
        jdbcTemplate.update(INSERT_LOGIN_LOG_SQL, args);
    }

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

}
```

2.3.4 在 Spring 中装配 DAO

在编写 DAO 接口的实现类时，大家也许会有一个问题：在以上两个 DAO 实现类中都没有打开/释放 Connection 的代码，DAO 类究竟如何访问数据库呢？前面说过，样板式的操作都被 JdbcTemplate 封装起来了，JdbcTemplate 本身需要一个 DataSource，这样它就可以根据需要从 DataSource 中获取或返回连接。UserDao 和 LoginLog 都提供了一个带@Autowired 注解的 JdbcTemplate 变量，所以我们必须事先声明一个数据源，然后定义一个 JdbcTemplate Bean，通过 Spring 的容器上下文自动绑定机制进行 Bean 的注入。

在项目工程的 src\resources（在 Maven 工程中，资源文件统一放置在 resources 文件夹中）目录下创建一个名为 smart-context.xml 的 Spring 配置文件，该配置文件的基本结构如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- 引用 Spring 的多个 Schema 空间的格式定义文件 -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-4.0.xsd">
   ...
</beans>
```

在 IDEA 中，刷新工程目录树，在 resources 文件夹下即可看到该配置文件。双击 smart-context.xml 文件，在这个文件中添加如代码清单 2-7 所示的配置信息。

代码清单 2-7　DAO Bean的配置

```
...
<beans ...>
   <!-- ①扫描类包，将标注 Spring 注解的类自动转化为 Bean，同时完成 Bean 的注入 -->
   <context:component-scan base-package="com.smart.dao"/>

   <!--②定义一个使用 DBCP 实现的数据源-->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/sampledb"
        p:username="root"
        p:password="123456" />

    <!--③定义 JDBC 模板 Bean  -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
         p:dataSource-ref="dataSource" />
</beans>
```

在①处，我们使用 Spring 的<context:component-scan>扫描指定类包下的所有类，这样在类中定义的 Spring 注解（如@Repository、@Autowired 等）才能产生作用。

在②处，我们使用 Jakarta 的 DBCP 开源数据源实现方案定义了一个数据源，数据库驱动器类为 com.mysql.jdbc.Driver。由于这里 MySQL 数据库的服务端口为 3309，而非默认的 3306，所以在数据库 URL 中显式指定了 3309 端口的信息。

在③处配置了 JdbcTemplate Bean，将②处声明的 dataSource 注入 JdbcTemplate 中，而这个 JdbcTemplate Bean 将通过@Autowired 自动注入 LoginLog 和 UserDao 的 Bean 中，可见 Spring 可以很好地将注解配置和 XML 配置统一起来。

这样就完成了登录模块持久层所有的开发工作，接下来将着手业务层的开发和配置工作。我们将对业务层的业务类方法进行单元测试，到时就可以看到 DAO 的实际运行效果了，现在暂时把这两个 DAO 放在一边。

提示

在附录 A 中有各种常见数据库连接配置的实例，如果用户希望采用不同的数据库，那么，请参考附录 A 进行相关的设置。

2.4　业务层

在论坛登录实例中，业务层仅有一个业务类，即 UserService。UserService 负责将持久层的 UserDao 和 LoginLoginDao 组织起来，完成用户/密码认证、登录日志记录等操作。

2.4.1　UserService

UserService 业务接口有 3 个业务方法，其中，hasMatchUser()方法用于检查用户名/密码的正确性；findUserByUserName()方法以用户名为条件加载 User 对象；loginSuccess()方法在用户登录成功后调用，更新用户最后登录时间和 IP 信息，同时记录用户登录日志。

下面我们来实现这个业务类。UserService 的实现类需要调用 DAO 层的两个 DAO 完成业务逻辑操作，如代码清单 2-8 所示。

代码清单 2-8　UserService

```
package com.smart.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.smart.dao.LoginLogDao;
import com.smart.dao.UserDao;
import com.smart.domain.LoginLog;
import com.smart.domain.User;

@Service //①将 UserService 标注为一个服务层的 Bean
public class UserService {
    private UserDao userDao;
    private LoginLogDao loginLogDao;
```

```
    @Autowired //②
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    @Autowired //③
    public void setLoginLogDao(LoginLogDao loginLogDao) {
        this.loginLogDao = loginLogDao;
    }

    public boolean hasMatchUser(String userName, String password) {
        int matchCount =userDao.getMatchCount(userName, password);
        return matchCount > 0;
    }

    public User findUserByUserName(String userName) {
        return userDao.findUserByUserName(userName);
    }

    @Transactional //④
    public void loginSuccess(User user) {
        user.setCredits( 5 + user.getCredits());
        LoginLog loginLog = new LoginLog();
        loginLog.setUserId(user.getUserId());
        loginLog.setIp(user.getLastIp());
        loginLog.setLoginDate(user.getLastVisit());
      userDao.updateLoginInfo(user);
      loginLogDao.insertLoginLog(loginLog);
    }
}
```

首先在①处通过@Service 注解将 UserService 标注为一个服务层的 Bean；然后在②和③处注入 userDao 和 loginLogDao 这两个 DAO 层的 Bean；接着通过 hasMatchUser()和 findUserByUserName()业务方法简单地调用 DAO 完成对应的功能；最后在④处为 loginSuccess()方法标注@Transactional 事务注解，让该方法运行在事务环境中（因为我们在 Spring 事务管理器拦截切入表达式上加入了@Transactional 过滤），否则该方法将在无事务方法中运行。loginSuccess()方法根据入参 user 对象构造出 LoginLog 对象并将 user.credits 递增 5，即用户每登录一次赚取 5 个积分，然后调用 userDao 更新到 t_user 中，再调用 loginLogDao 向 t_login_log 表中添加一条记录。

实战经验

在实际应用中，一般不会直接在数据库中以明文的方式保存用户的密码，因为这样很容易造成密码泄露，所以需要将密码加密后以密文的方式进行保存；另外一种更有效的办法是仅保存密码的 MD5 摘要，由于相等的两个字符串的摘要值也相等，所以在登

录验证时，通过比较摘要的方式就可以判断用户所输入的密码是否正确。由于不能通过密码摘要反推出原来的密码，即使内部人员可以查看用户信息表，也无法知道用户的密码。所以，摘要存储方式已经成为大部分系统密码存储的通用方式。此外，为了防止黑客通过工具进行密码的暴力破解，目前大多数 Web 应用都使用了图片验证码功能，验证码具有一次性消费的特征，每次登录都不相同，这样工具暴力破解就无用武之地了。

loginSuccess()方法将两个 DAO 组织起来，共同完成一个事务性的数据操作：更新 t_user 表记录并添加 t_login_log 表记录。但我们从 UserService 中却看不出任何事务操作的影子，这正是 Spring 的高明之处，它让我们从事务操作单调、机械的代码中解脱出来，专注完成那些不可或缺的业务工作。通过 Spring 声明式事务配置即可让业务类享受 EJB 声明式事务的好处，下一节我们将了解如何赋予业务类事务管理的能力。

2.4.2　在 Spring 中装配 Service

事务管理的代码虽然无须出现在程序代码中，但我们必须以某种方式告诉 Spring 哪些业务类需要工作在事务环境下及事务的规则等内容，以便 Spring 根据这些信息自动为目标业务类添加事务管理的功能。

打开原来的 smart-context.xml 文件，进行如代码清单 2-9 所示的更改。

代码清单 2-9　smart-service.xml

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

<!--① 引入 aop 及 tx 命名空间所对应的 Schema 文件-->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    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-4.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-4.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">


   <context:component-scan base-package="com. smart.dao"/>
   <!—② 扫描 service 类包，应用 Spring 的注解配置 -->
   <context:component-scan base-package="com. smart.service"/>

    ...
    <!--③ 配置事务管理器 -->
    <bean id="transactionManager"
```

```
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource" />

    <!--④ 通过 AOP 配置提供事务增强，让 service 包下所有 Bean 的所有方法拥有事务 -->
    <aop:config proxy-target-class="true">
        <aop:pointcut id="serviceMethod"
                expression=" (execution(* com.smart.service..*(..))) and
                (@annotation(org.springframework.transaction.annotation.Transactional))" />
        <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
    </aop:config>
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" />
        </tx:attributes>
    </tx:advice>
</beans>
```

①处在<beans>的声明处添加 aop 和 tx 命名空间的 Schema 定义文件的说明，这样，在配置文件中就可以使用这两个空间下的配置标签了。

②处将 com.smart.service 添加到上下文扫描路径中，以便使 service 包中类的 Spring 注解生效。

③处定义了一个基于数据源的 DataSourceTransactionManager 事务管理器，该事务管理器负责声明式事务的管理。该管理器需要引用 dataSource Bean。

④处通过 aop 及 tx 命名空间的语法，以 AOP 的方式为 com.smart.service 包下所有类的所有标注@Transactional 注解的方法都添加了事务增强，即它们都将工作在事务环境中（关于 Spring 事务的配置，详见第 11 章）。

这样就完成了业务层的程序开发和配置工作，接下来需要对该业务类进行简单的单元测试，以便检验业务方法的正确性。

2.4.3 单元测试

TestNG 和 JUnit 相比有了重大的改进，本书示例所有的单元测试统一采用 TestNG 框架。请确保已经将 TestNG 依赖包添加到根模块 pom.xml 文件中。在 chapter2\src\test 测试目录下创建与 UserService 一致的包结构，即 com.smart.service，并创建 UserService 对应的测试类 UserServiceTest，编写如代码清单 2-10 所示的测试代码。

代码清单 2-10 UserServiceTest

```
package com.smart.service;

import java.util.Date;
import org.springframework.test.context.testng.AbstractTransactionalTestNGSpring
ContextTests;
import org.testng.annotations.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import com.smart.domain.User;
```

```
import static org.testng.Assert.*;

@ContextConfiguration("classpath*:/applicationContext.xml") //②启动 Spring 容器
public class UserServiceTest extends AbstractTransactionalTestNGSpringContextTests {
    private UserService userService;

    @Autowired //③注入 Spring 容器中的 Bean
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    //④标注测试方法
    @Test
    public void hasMatchUser() {
        boolean b1 = userService.hasMatchUser("admin", "123456");
        boolean b2 = userService.hasMatchUser("admin", "1111");
        assertTrue(b1);
        assertTrue(!b2);
    }

    @Test
    public void findUserByUserName() {
        User user = userService.findUserByUserName("admin");
        assertEquals(user.getUserName(), "admin");
    }

    ...
}
```

Spring 4.0 的测试框架很好地整合了 TestNG 单元测试框架，示例 UserServiceTest 通过扩展 Spring 测试框架提供测试基类 AbstractTransactionalTestNGSpringContextTests 来启动测试运行器。@ContextConfiguration 也是 Spring 提供的注解，用于指定 Spring 的配置文件。

可以使用 Spring 的@Autowired 将 Spring 容器中的 Bean 注入测试类中。在测试方法前通过 TestNG 的@Test 注解即可将方法标注为测试方法。

在 IDEA 中执行当前测试类，通过右键菜单 Run ‘UserServiceTest’来运行该测试用例，以检验业务类方法的正确性，如图 2-12 所示。

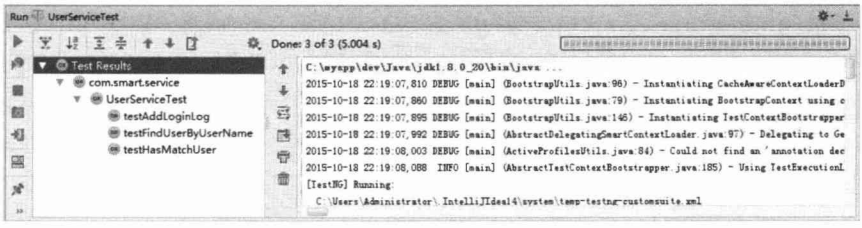


图 2-12　UserServiceTest 的运行结果

从单元测试的运行结果可以看到 3 个业务方法已经成功执行，在后台数据库中，用户会发现已经有一条新的登录日志添加到 t_login_log 表中。关于 Spring 应用单元测试内容，详见第 20 章的内容。

提示

在测试类中的空白处，通过右键菜单 Run 'UserServiceTest' 运行测试用例时，会执行当前测试用例的所有测试方法。如果在开发中只想运行测试用例的某一个测试方法，则可以将鼠标移至相应的测试方法块内，通过右键菜单运行当前的测试方法。

2.5 展现层

业务层和持久层的开发任务已经完成，该是为程序提供界面的时候了。Struts MVC 框架由于抢尽天时地利，成为当下主流的展现层框架。但也有很多人认为 Spring MVC 相比于 Struts 更简单、更强大、更优雅。此外，由于 Spring MVC 出自 Spring 之手，因此和 Spring 容器没有任何阻抗，显得天衣无缝。

Spring 3.0 提供了 REST 风格的 MVC，使 Spring MVC 变得更轻便、易用。Spring4.0 对 MVC 进行了全面增强，支持跨域注解@CrossOrigin 配置，Groovy Web 集成，Gson、Jackson、Protobuf 的 HttpMessageConverter 消息转换器等，使 Spring MVC 的功能更加丰富、强大（读者将在第 18 章学习到 Spring MVC 的详细内容）。

2.5.1 配置 Spring MVC 框架

首先需要对 web.xml 文件进行配置，以便 Web 容器启动时能够自动启动 Spring 容器，如代码清单 2-11 所示。

代码清单 2-11 自动启动Spring容器的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

<!--①从类路径下加载Spring配置文件，classpath关键字特指类路径下加载-->
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
        classpath:smart-context.xml
       </param-value>
    </context-param>
    <!--②负责启动Spring容器的监听器，它将引用①处的上下文参数获得Spring配置文件的地址-->
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
```

```
    ...
</web-app>
```

然后通过 Web 容器上下文参数指定 Spring 配置文件的地址，如①所示。多个配置文件可用逗号或空格分隔，建议采用逗号分隔的方式。然后在②处指定 Spring 所提供的 ContextLoaderListener 的 Web 容器监听器，该监听器在 Web 容器启动时自动运行，它会根据 contextConfigLocation Web 容器参数获取 Spring 配置文件，并启动 Spring 容器。注意，需要将 log4J.propertis 日志配置文件放置在类路径下，以便日志引擎自动生效。

最后需要配置 Spring MVC 相关的信息。Spring MVC 像 Struts 一样，也通过一个 Servlet 来截获 URL 请求，然后再进行相关的处理，如代码清单 2-12 所示。

代码清单 2-12　Spring MVC地址映射

```
...
<!-- Spring MVC的主控Servlet -->
<servlet> <!--①-->
    <servlet-name>smart</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>
   <!-- Spring MVC处理的URL -->
    <servlet-mapping> <!--②-->
        <servlet-name>smart</servlet-name>
        <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

在①处声明了一个 Servlet，Spring MVC 也拥有一个 Spring 配置文件（稍后将涉及），该配置文件的文件名和此处定义的 Servlet 名有一个契约，即采用<*Servlet 名*>-servlet.xml 的形式。在这里，Servlet 名为 smart，则在/WEB-INF 目录下必须提供一个名为 smart-servlet.xml 的 Spring MVC 配置文件，但这个配置文件无须通过 web.xml 的 contextConfigLocation 上下文参数进行声明，因为 Spring MVC 的 Servlet 会自动将 smart-servlet.xml 文件和 Spring 的其他配置文件（smart-dao.xml、smart-service.xml）进行拼装。

在②处对这个 Servlet 的 URL 路径映射进行定义，在这里让所有以.html 为后缀的 URL 都能被 smart Servlet 截获，进而转由 Spring MVC 框架进行处理。我们知道，在 Struts 框架中一般将 URL 后缀配置为*.do，而在 WebWork 中一般配置为*.action。其实，框架本身和 URL 模式没有任何关系，用户大可使用喜欢的任何后缀。使用.html 后缀，一方面，用户不能通过 URL 直接知道我们采用了何种服务器端技术；另一方面，.html 是静态网页的后缀，可以骗过搜索引擎，增加被收录的概率，所以我们推荐采用这种后缀。对于那些真正无须任何动态处理的静态网页，则可以使用.htm 后缀加以区分，以避免被框架截获。

请求被 Spring MVC 截获后，首先根据请求的 URL 查找到目标的处理控制器，并将请求参数封装“命令”对象一起传给控制器处理；然后，控制器调用 Spring 容器中的业务 Bean 完成业务处理工作并返回结果视图。

2.5.2 处理登录请求

1. POJO 控制器类

首先需要编写的是 LoginController，它负责处理登录请求，完成登录业务，并根据登录成功与否转向欢迎页面或失败页面，如代码清单 2-13 所示。

代码清单 2-13 LoginController

```
package com.smart.web;
import java.util.Date;
import javax.servlet.http.HttpServletRequest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import com.smart.domain.User;
import com.smart.service.UserService;

//①标注成为一个Spring MVC的Controller
@Controller
public class LoginController{
    private UserService userService;

    //②负责处理/index.html的请求
    @RequestMapping(value = "/index.html")
    public String loginPage(){
        return "login";
    }

    //③负责处理/loginCheck.html的请求
    @RequestMapping(value = "/loginCheck.html")
    public ModelAndView loginCheck(HttpServletRequest request,LoginCommand
loginCommand){
        boolean isValidUser =
              userService.hasMatchUser(loginCommand.getUserName(),
                                       loginCommand.getPassword());
        if (!isValidUser) {
            return new ModelAndView("login", "error", "用户名或密码错误。");
        } else {
            User user = userService.findUserByUserName(loginCommand
                    .getUserName());
            user.setLastIp(request.getLocalAddr());
            user.setLastVisit(new Date());
            userService.loginSuccess(user);
            request.getSession().setAttribute("user", user);
            return new ModelAndView("main");
        }
    }

   @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
```

```
    }
}
```

在①处通过 Spring MVC 的@Controller 注解可以将任何一个 POJO 的类标注为 Spring MVC 的控制器，处理 HTTP 的请求。当然，标注了@Controller 的类首先会是一个 Bean，所以可以使用@Autowired 进行 Bean 的注入。

一个控制器可以拥有多个处理映射不同 HTTP 请求路径的方法，通过@RequestMapping 指定方法如何映射请求路径，如②和③处所示。

请求参数会根据参数名称默认契约自动绑定到相应方法的入参中。例如，在③处的 loginCheck(HttpServletRequest request,LoginCommand loginCommand)方法中，请求参数会按名称匹配绑定到 loginCommand 的入参中。

请求响应方法可以返回一个 ModelAndView，或直接返回一个字符串，Spring MVC 会解析之并转向目标响应页面。

ModelAndView 对象既包括视图信息，又包括视图渲染所需的模型数据信息。在这里用户仅需要了解它代表一张视图即可，在后面的内容中，读者将了解到 Spring MVC 如何根据这个对象转向真正的页面。

前面用到的 LoginCommand 对象是一个 POJO，没有继承特定的父类或实现特定的接口。LoginCommand 类仅包括用户/密码这两个属性（和请求的用户/密码参数名称一样），如代码清单 2-14 所示。

代码清单 2-14　LoginCommand

```
package com.smart.web;
public class LoginCommand {
    private String userName;
    private String password;
    //省略get/setter方法
}
```

2. Spring MVC 配置文件

编写好 LoginCommand 后，需要在 smart-servlet.xml 中声明该控制器，扫描 Web 路径，指定 Spring MVC 的视图解析器，如代码清单 2-15 所示。

代码清单 2-15　smart-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"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <!--① 扫描Web包，应用Spring的注解 -->
    <context:component-scan base-package="com.smart.web"/>
```

```
    <!--② 配置视图解析器，将ModelAndView及字符串解析为具体的页面 -->
    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:viewClass="org.springframework.web.servlet.view.JstlView"
        p:prefix="/WEB-INF/jsp/"
        p:suffix=".jsp" />

</beans>
```

3. ModelAndView 的解析配置

在代码清单 2-13 的②和③处，控制器根据登录处理结果分别返回 ModelAndView ("login", "error", "用户名或密码错误。")和 ModelAndView("main")。ModelAndView 的第一个参数代表视图的逻辑名，第二、第三个参数分别为数据模型名称和数据模型对象，数据模型对象将以数据模型名称为参数名放置到 request 的属性中。

Spring MVC 如何将视图逻辑名解析为具体的视图页面呢？解决思路和上面的方法类似，需要在 smart-servlet.xml 中提供一个定义解析规则的 Bean，如代码清单 2-16 所示。

代码清单 2-16　smart-servlet.xml视图解析规则

```
...
<!--通过prefix指定在视图名前所添加的前缀，通过suffix指定在视图名后所添加的后缀-->
<bean   class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:viewClass="org.springframework.web.servlet.view.JstlView"
        p:prefix="/WEB-INF/jsp/"
        p:suffix=".jsp" />
```

Spring MVC 为视图名到具体视图的映射提供了许多可供选择的方法。在这里，我们使用 InternalResourceViewResolver，它通过为视图逻辑名添加前、后缀的方式进行解析。如视图逻辑名为“login”，将解析为/WEB-INF/jsp/login.jsp；视图逻辑名为“main”，将解析为/WEB-INF/jsp/main.jsp。

2.5.3　JSP 视图页面

论坛登录模块共包括两个 JSP 页面，分别是登录页面 login.jsp 和欢迎页面 main.jsp，我们将在这节完成这两个页面的开发工作。

1. 登录页面 login.jsp

登录页码如代码清单 2-17 所示。

代码清单 2-17　login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
    <head>
        <title>小春论坛登录</title>
    </head>
    <body>
```

```
        <c:if test="${!empty error}">①
          <font color="red"><c:out value="${error}" /></font>
        </c:if>
        <form action="<c:url value="/ loginCheck.html "/>" method= "post">②
            用户名:
            <input type="text" name="userName">
            <br>
            密 码:
            <input type="password" name="password">
            <br>
            <input type="submit" value="登录" />
            <input type="reset" value="重置" />
        </form>
    </body>
</html>
```

login.jsp 页面有两个用处，既作为登录页面，又作为登录失败后的响应页面。所以在①处，使用 JSTL 标签对登录错误返回的信息进行处理。在 JSTL 标签中引用了 error 变量，这个变量正是 LoginController 中返回的 ModelAndView("login", "**error**", "用户名或密码错误。") 对象所声明的 **error** 参数。

login.jsp 的登录表单提交到/loginController.html，如②所示。<c:url value= "/loginController.html"/>的 JSTL 标签会在 URL 前自动加上应用部署根目录。假设应用部署在网站的 bbt 目录下，则<c:url/>标签将输出/bbt/loginController.html。通过<c:url/>标签很好地解决了开发和应用部署目录不一致的问题。

由于 login.jsp 放置在 WEB-INF/jsp 目录下，无法直接通过 URL 进行调用，所以它由 LoginController 控制类中标注了@RequestMapping(value = "/index.html")的 loginPage() 进行转发，如代码清单 2-13 所示。

2．欢迎页面 main.jsp

登录成功的欢迎页面很简单，仅使用 JSTL 标签显示一条欢迎信息即可，如代码清单 2-18 所示。

代码清单 2-18　main.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
     <title>小春论坛</title>
  </head>
  <body>
     ${user.userName},欢迎您进入小春论坛，您当前积分为${user.credits}; ①
  </body>
</html>
```

①处访问 Session 域中的 user 对象，显示用户名和积分信息。这样就完成了实例所有的开发任务，下一节将通过 Ant 工具对 Web 应用进行打包和部署。

2.6 运行 Web 应用

基于 Maven 工程，运行 Web 应用有两种方式：第一种方式是在 IDE 工具中配置 Web 应用服务器；第二种方式是在 pom.xml 文件中配置 Web 应用服务器插件。推荐采用第二种方式，本书章节示例都将采用第二种方式。这里我们在 pom.xml 文件中配置 Jetty 应用服务器插件，如代码清单 2-19 所示。

代码清单 2-19　chapter2 模块pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
…
<build>
      <plugins>
         <!-- Jetty插件 -->
         <plugin>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>maven-jetty-plugin</artifactId>
            <version>6.1.25</version>
            <configuration><!-配置说明 -->
                  <connectors>
                   <connector implementation="org.mortbay.jetty.nio.
                      SelectChannelConnector">
                      <port>8000</port>
                      <maxIdleTime>60000</maxIdleTime>
                   </connector>
               </connectors>
               <contextPath>/bbs</contextPath>
               <scanIntervalSeconds>0</scanIntervalSeconds>
            </configuration>
         </plugin>
      </plugins>
</build>
</project>
```

Jetty 插件常用配置选项说明如下。

在 Connectors 中配置 Connector 对象，包含 Jetty 的监听端口。如果不配置连接器，如 org.mortbay.jetty.nio.SelectChannelConnector，则默认监听端口会被设置为 8080。本示例将通过连接器 port 选项，将监听端口设置为 8000。

contextPath 可选，用于配置 Web 应用上下文。如果不配置此项，则默认上下文采用 pom.xml 中设置的<artifactId>名称。本示例将上下文设置为 bbs。

overrideWebXml 可选，它是一个应用于 Web 应用的 web.xml 的备用 web.xml 文件。这个文件可以存放在任何地方。用户可以根据不同的环境（如测试、开发等），利用它增加或修改一个 web.xml 配置。

webDefaultXml 可选，webdefault.xml 文件用来代替 webapp 默认提供给 Jetty 的文件。

scanIntervalSeconds 可选[秒]，在设置间隔内检查 Web 应用是否有变化，如果发现有变更则自动热部署。默认为 0，表示禁用热部署。任何一个大于 0 的数字都将表示启用。

systemPropertie 可选，允许用户在设置一个插件的执行操作时配置系统属性（更多的信息请查阅 Setting System Properties）。

在 pom.xml 中配置好 Jetty 插件之后，在 IDEA 工程右边的 Maven Projects 管理窗口工具栏中单击 图标，在章节模块中的插件节点 Plugins 下会自动出现安装的 Jetty 插件，如图 2-13 所示。

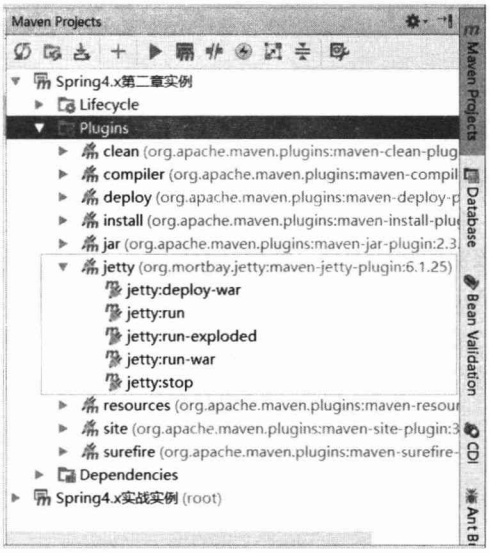


图 2-13　Jetty 插件

双击 jetty:run 或 jetty:run-exploded，将以运行模式启动 Jetty 服务器。如果想要以 Debug 模式运行应用，则通过右键菜单选择 Debug 运行应用即可。图 2-14 是论坛登录的首页面。

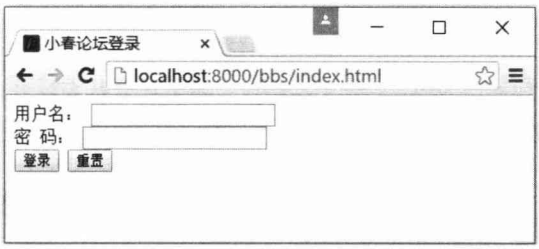


图 2-14　在浏览器中访问应用

如果输入错误的用户名/密码，则登录模块将给出错误提示。这里输入 admin/123456，单击“登录”按钮后，就可以登录到欢迎页面中，如图 2-15 所示。

图 2-15　登录成功后的欢迎页面

2.7　小结

在本章中，我们用 Spring MVC、Spring JDBC 及 Spring 的声明式事务等技术实现了一个常见的论坛登录模块，让大家对如何使用 Spring 框架构建 Web 应用拥有了切身体验，同时了解了开发一个简单的 Web 应用所需经历的开发过程。

也许用户会抱怨该实例功能的简单性和开发过程的复杂性不成正比，但对于一个具有扩展性、灵活性的 Web 应用来说，这些步骤往往都是必需的。其实我们在完成实例开发的同时也完成了 Web 框架的搭建，为新功能模块的添加夯实了地基，后继的模块开发仅需在此基础上进行添砖加瓦的工作，当新功能加入时，读者就会发现我们在这里所做的工作是值得的。

第 3 章

Spring Boot

Spring Boot 是由 Pivotal 团队设计的全新框架，其目的是用来简化 Spring 应用开发过程。本章使用 Spring Boot 重新开发第 2 章的实例，通过这个实例开发实战，读者不仅可以全面认识 Spring Boot 的作用、用法与价值，而且可以学会如何利用 Spring Boot 框架来简化日常的项目开发。本章将 Spring Boot 贯穿于安装配置、快速入门、运维支持及应用开发的全过程。通过本章的学习，读者可以应用 Spring Boot 快速、独立地完成一个典型的基于 Spring 的 Web 项目开发。

本章主要内容：

- Spring Boot 概览
- Spring Boot 快速入门
- Spring Boot 安装配置
- Spring Boot 应用实战
- Spring Boot 运维支持

本章亮点：

- Spring Boot 各种环境配置讲解
- Spring Boot 详细的开发过程讲解
- Spring Boot 运维支持能力讲解

3.1 Spring Boot 概览

Spring Boot 是由 Pivotal 团队设计的全新框架，其目的是用来简化 Spring 应用开发过程。该框架使用了特定的方式来进行配置，从而使得开发人员不再需要定义一系列样板化的配置文件，而专注于核心业务开发，项目涉及的一些基础设施则交由 Spring Boot 来解决。

3.1.1 Spring Boot 发展背景

多年来，Spring 配置复杂性一直为人所诟病，Spring IO 子项目试图化解这一问题，但由于其主要侧重于解决集成方面的问题，因此 Spring 配置复杂性并没有得到本质的改观，如何实现简化 Spring 配置的呼声依旧高亢，直到 Spring Boot 的出现。Spring Boot 可让开发人员不再需要编写复杂的 XML 配置文件，仅通过几行代码就能实现一个可运行的 Web 应用。

Spring Boot 不是去再造一个“轮子”，它的“革命宣言”是为 Spring 项目开发带来一种全新的体验，从而大大降低 Spring 框架的使用门槛。

Spring Boot 革新 Spring 项目开发体验之道，其实是借助强大的 Groovy 动态语言实现的，如借助 Groovy 强大的 MetaObject 协议、可插拔的 AST 转换器及内置的依赖解决方案引擎等。在其核心的编译模型中，Spring Boot 使用 Groovy 来构建工程文件，所以它可以轻松地利用导入模板及方法模板对类所生成的字节码进行改造，从而让开发者仅用很简洁的代码就可以完成很复杂的操作。

3.1.2 Spring Boot 特点

从 Spring Boot 项目名称中的 Boot 可以看出，Spring Boot 的作用在于创建和启动新的基于 Spring 框架的项目，其目的是帮助开发人员快速构建出基于 Spring 的应用。Spring Boot 像一个“管家”，它会在后台“智能地”整合项目所需的第三方依赖类库或框架，因此大部分基于 Spring Boot 的应用仅需要很少的配置就可以运行起来。

Spring Boot 包含如下特性：

- 为开发者提供 Spring 快速入门体验。
- 内嵌 Tomcat 和 Jetty 容器，不需要部署 WAR 文件到 Web 容器就可独立运行应用。
- 提供许多基于 Maven 的 pom 配置模板来简化工程配置。
- 提供实现自动化配置的基础设施。
- 提供可以直接在生产环境中使用的功能，如性能指标、应用信息和应用健康检查。
- 开箱即用，没有代码生成，也无须 XML 配置文件，支持修改默认值来满足特定需求。

通过 Spring Boot，创建一个新的 Spring 应用变得非常简单，只需几步即可完成。

3.1.3 Spring Boot 启动器

Spring Boot 是由一系列启动器组成的，这些启动器构成一个强大的、灵活的开发助

手。开发人员根据项目需要，选择并组合相应的启动器，就可以快速搭建一个适合项目需要的基础运行框架。例如，要开发一个基于 Maven 的 Web 项目，通常在模块的 pom.xml 文件中引入 spring-mvc、spring-webmvc、jackson、tomcat 等依赖模块；如果使用 Spring Boot 启动器，则只需引用一个 spring-boot-starter-web 模块即可。下面先来了解一下 Spring 提供了哪些有用的启动器，如表 3-1 所示。

表 3-1　Spring Boot启动器

启动器名称	启动器说明
spring-boot-starter	核心模块，包含自动配置支持、日志库和对 YAML 配置文件的支持
spring-boot-starter-amqp	支持 AMQP，包含 spring-rabbit
spring-boot-starter-aop	支持面向切面编程（AOP），包含 spring-aop 和 AspectJ
spring-boot-starter-artemis	通过 Apache Artemis 支持 JMS 的 API（Java Message Service API）
spring-boot-starter-batch	支持 Spring Batch，包含 HSQLDB
spring-boot-starter-cache	支持 Spring 的 Cache 抽象
spring-boot-starter-cloud-connectors	支持 Spring Cloud Connectors，简化了在像 Cloud Foundry 或 Heroku 这样的云平台上连接服务
spring-boot-starter-data-gemfire	支持 GemFire 分布式数据存储，包含 spring-data-gemfire
spring-boot-starter-data-jpa	支持 JPA，包含 spring-data-jpa、spring-orm 和 Hibernate
spring-boot-starter-data-elasticsearch	支持 ElasticSearch 搜索和分析引擎，包含 spring-data-elasticsearch
spring-boot-starter-data-solr	支持 Apache Solr 搜索平台，包含 spring-data-solr
spring-boot-starter-data-mongodb	支持 MongoDB，包含 spring-data-mongodb
spring-boot-starter-data-rest	支持以 REST 方式暴露 Spring Data 仓库，包含 spring-data-rest-webmvc
spring-boot-starter-redis	支持 Redis 键值存储数据库，包含 spring-redis
spring-boot-starter-jdbc	支持使用 JDBC 访问数据库
spring-boot-starter-jta-atomikos	通过 Atomikos 支持 JTA 分布式事务处理
spring-boot-starter-jta-bitronix	通过 Bitronix 支持 JTA 分布式事务处理
spring-boot-starter-security	包含 spring-security
spring-boot-starter-test	包含常用的测试所需的依赖，如 TestNG、Hamcrest、Mockito 和 spring-test 等
spring-boot-starter-velocity	支持使用 Velocity 作为模板引擎
spring-boot-starter-freemarker	支持 FreeMarker 模板引擎
spring-boot-starter-thymeleaf	支持 Thymeleaf 模板引擎，包括与 Spring 的集成
spring-boot-starter-mustache	支持 Mustache 模板引擎
spring-boot-starter-web	支持 Web 应用开发，包含 tomcat、spring-mvc、spring-webmvc 和 jackson
spring-boot-starter-websocket	支持使用 Tomcat 开发 WebSocket 应用
spring-boot-starter-ws	支持 Spring Web Services
spring-boot-starter-groovy-templates	支持 Groovy 模板引擎
spring-boot-starter-hateoas	通过 spring-hateoas 支持基于 Hateoas 的 RESTful Web 服务
spring-boot-starter-hornetq	通过 HornetQ 支持 JMS
spring-boot-starter-log4j	添加 Log4j 的支持
spring-boot-starter-logging	使用 Spring Boot 默认的日志框架 Logback
spring-boot-starter-integration	支持通用的 spring-integration 模块

续表

启动器名称	启动器说明
spring-boot-starter-jersey	支持 Jersey RESTful Web 服务框架
spring-boot-starter-mail	支持 javax.mail 模块
spring-boot-starter-mobile	支持 spring-mobile
spring-boot-starter-social-facebook	支持 spring-social-facebook
spring-boot-starter-social-linkedin	支持 pring-social-linkedin
spring-boot-starter-social-twitter	支持 pring-social-twitter
spring-boot-starter-actuator	添加适用于生产环境的功能，如性能指标和监测等功能
spring-boot-starter-remote-shell	支持远程 SSH 命令操作
spring-boot-starter-tomcat	使用 Spring Boot 默认的 Tomcat 作为应用服务器
spring-boot-starter-jetty	引入了 Jetty HTTP 引擎（用于替换 Tomcat）
spring-boot-starter-undertow	引入了 Undertow HTTP 引擎（用于替换 Tomcat）

Spring Boot 通过这些启动器将不同的第三方组件依赖进行了套件封装，为开发 Spring 应用提供了一个易用的套件库。Spring Boot 所选用的第三方组件库都是经过认真评估和谨慎选择的，开发者大可放心享用 Spring Boot 的“调制套餐”。

轻松一刻

计算机引导启动的英文单词是 boot，可是，boot 原意是靴子，“启动”与“靴子”有何关系？原来，这里的 boot 是 bootstrap（鞋带）的缩写，它来自西方一句“拉鞋带”的谚语“pull oneself up by one's bootstraps”，译为“拽着鞋带把自己拉起来”，这就相当于项羽坐在椅子上要把自己举起来的典故一样，当然是不可能的。计算机启动本身就是一个很矛盾的过程：必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序——就像鸡生蛋、蛋生鸡一样！所以，工程师把这个启动过程叫作“拉鞋带”，久而久之就简称为 boot 了。

3.2 快速入门

上一节学习了 Spring Boot 相关背景、特点、启动器的基础知识，下面通过一个示例，快速体验一下 Spring Boot 的功效。我们以 Maven 方式快速创建一个 Spring Web 应用，首先需要在 pom.xml 文件中引入 Spring Boot 依赖，如代码清单 3-1 所示。

代码清单 3-1　Spring Boot依赖配置pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.3.RELEASE</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>chapter3</artifactId>
  <name>Spring4.x第三章实例</name>
  <dependencies>
    <!--①添加一个Boot Web启动器-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>1.3.3.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

在①处引用了一个 spring-boot-starter-web 启动器依赖，不像在第 2 章中的示例一样，需要引入很多 Spring 子模块依赖。当使用 Maven“mvn dependency:tree”命令或 IDEA 依赖查看视图功能（在 pom.xml 文件视图中单击鼠标右键选择 Diagrams→Show Dependencies）时，会发现 spring-boot-starter-web 内部已经封装了 spring-web、spring-webmvc、jackson- databind 等模块依赖。

配置好 Spring Boot 相关依赖之后，接下来就可以通过几行代码，快速创建一个 Web 应用，如代码清单 3-2 所示。

代码清单 3-2　论坛应用BbsDaemon

```
package com.smart.web;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@EnableAutoConfiguration
public class BbsDaemon {

  @RequestMapping("/")
  public String index() {
    return "欢迎光临小春论坛!";
  }

  public static void main(String[] args) throws Exception {
    SpringApplication.run(BbsDaemon.class, args);
  }
}
```

其中，@EnableAutoConfiguration 注解是由 Boot 提供的，用于对 Spring 框架进行自动配置，减少了开发人员的工作量；@RestController 和@RequestMapping 注解是由 Spring MVC 提供的，用于创建 Rest 服务。

直接运行 BbsDaemon 类会启动一个运行于 8080 端口的内嵌 Tomcat 服务，在浏览器中访问“http://localhost:8080”，即可看到页面上显示“欢迎光临小春论坛!”。也就是

说，只需简单的两个步骤就完成了一个可独立启动运行的 Web 应用。既没有配置安装 Tomcat 或 Jetty 这样的应用服务器，也没有打包成 WAR 文件——与传统开发方式相比，Spring Boot 堪称犀利！

3.3 安装配置

Spring Boot 1.3.3 需要运行在 Java 7.0+及 Spring 4.1.5+版本中。如果要让其运行在 Java 6.0 版本中，则须做一些特殊的配置。配套的构建工具需要使用 Maven 3.2+或 Gradle 1.12+。虽然 Spring Boot 可以运行在 Java 6.0、7.0 版本中，但为了更好地使用 Spring Boot，官方推荐使用 Java 8.0。

目前 Spring Boot 支持内嵌 Servlet 容器，如表 3-2 所示。

表 3-2 支持内嵌Servlet容器

Web 容器名称	Servlet 版本	Java 版本
Tomcat 8	3.1	Java 7.0+
Tomcat 7	3.0	Java 6.0+
Jetty 9	3.1	Java 7.0+
Jetty 8	3.0	Java 6.0+
Undertow 1.1	3.1	Java 7.0+

可以发现，基于 Spring Boot 构建的 Web 应用可运行于任何支持 Servlet 3.0+及 Java 6.0+的 Web 容器中。

Spring Boot 实际上是一些类库的集合，它能够被任意项目的构建系统所引用。为简便起见，Spring Boot 提供了一个命令行客户端运行工具（Spring Boot CLI），可用来运行和测试 Spring Boot 应用。Spring Boot 发布版本包含集成的 CLI，可以在 Spring 仓库中手工下载和安装。此外，也可手工在项目类库中引入 spring-boot-*.jar 类库来使用 Spring Boot。但我们强烈建议使用 Maven 或 Gradle 构建系统来使用 Spring Boot。

3.3.1 基于 Maven 环境配置

Spring Boot 依赖 Maven 3.2+，如果基于 Maven 环境运行 Spring Boot，则需要确保机器环境已经安装配置好 Maven（http://maven.apache.org）。为了简化 Spring 依赖关系，Spring Boot 按照不同的功能需要进行了模块划分，开发人员可以根据需要导入相关的“spring-boot-starter-*”模块。为了更容易地管理依赖版本和使用默认配置，Spring Boot 提供了一个 pom 根配置，在项目工程中可以直接继承它，如代码清单 3-3 所示。

代码清单 3-3 Maven配置样例

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!--①继承 Spring Boot默认配置 -->
    <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
     <version>1.3.3.RELEASE</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>chapter3</artifactId>
     <name>Spring4.x第三章实例</name>

    <!--②根据应用需要添加不同类型的启动器依赖-->
    <dependencies>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
       </dependency>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-jetty</artifactId>
       </dependency>
    </dependencies>
    <build>
       <plugins>
          <!--③配置运行插件-->
          <plugin>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>
       </plugins>
    </build>
</project>
```

在①处继承了 Spring Boot 提供的根默认配置依赖，并且指定当前版本号为 1.3.3.RELEASE，这里引入 spring-boot-starter-parent 的好处是在②处添加启动器时不必再声明版本号。如果不想采用继承方式，也可以使用如下方式导入 Boot 提供的根配置。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>1.3.3.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

在③处引用了一个 Spring Boot 运行插件。刷新 IDEA 开发工具右边的 Maven Projects 选项卡，如图 3-1 所示，就可以看到 Spring Boot 运行命令。

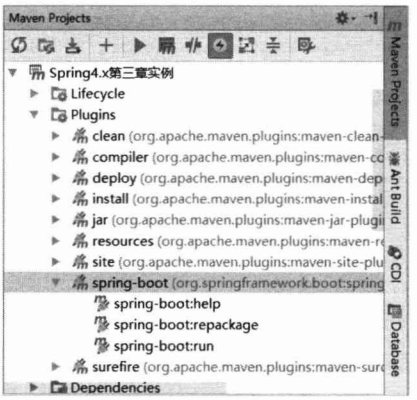


图 3-1　Spring Boot 插件

通过双击执行 spring-boot:run 命令，在默认情况下，Spring Boot 会采用内嵌的 Tomcat 运行当前应用。如果想使用 Jetty 运行当前应用，则只需在依赖中添加一个 Jetty 启动器（spring-boot-starter-jetty）即可。

3.3.2　基于 Gradle 环境配置

Spring Boot 依赖 Gradle 1.12+，如果基于 Gradle 环境运行 Boot，则需要确保机器环境已经安装配置好 Gradle（http://www.gradle.org）。为了实现更为简单的构建配置，开发人员可以使用 Gradle 提供的简洁的 Groovy DSL 来编写依赖，如代码清单 3-4 所示。

代码清单 3-4　Gradle配置示例

```
buildscript { //①中心仓库与Boot插件
  repositories {
     jcenter()
     maven { url "http://repo.spring.io/snapshot" }
     maven { url "http://repo.spring.io/milestone" }
  }
  dependencies {
     classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.3.RELEASE")
  }
}
apply plugin: 'java' //②指定打包方式
apply plugin: 'spring-boot'
jar { //③指定模块名称与版本号
    baseName = 'chapter3'
    version = '1.0'
}
repositories {
    jcenter()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}
dependencies { //④指定模块依赖包
```

```
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

在①处，在 buildscript 中指定了当前模块的中心仓库，并指定了基于 Gradle 插件，通过这个插件，可以将当前模块打成一个可运行的 JAR 包或 WAR 包（打成 WAR 包，需要将②处的 apply plugin: 'java'改为 apply plugin: 'war'）。在③处指定了当前模块应用所依赖的启动器。需要特别注意的是，由于在 Gradle 中没有提供类似于 Maven 的根配置，所以在基于 Gradle 的 Boot 运行环境中需要自动配置相关依赖。

3.3.3　基于 Spring Boot CLI 环境配置

Spring Boot 除提供了上述基于 Gradle、Maven 构建系统外，还提供了基于命令行工具的方式，可让用户快速创建基于 Spring 框架系统原型的项目。Spring Boot 发布版本包含了 CLI 命令工具包，也可以通过手工方式，从 Spring 仓库下载相关版本的客户端。

仓库地址：

http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/

Windows 环境下载：

spring-boot-cli-1.3.3.RELEASE-bin.zip

Linux 环境下载：

spring-boot-cli-1.3.3.RELEASE-bin.tar.gz

选择命令行客户端相应版本，单击并下载到本地磁盘，本书示例统一放在 D:\masterSpring 目录，解压缩命令行客户端，其目录结构如图 3-2 所示。

图 3-2　命令行工具目录结构

在 Windows 命令行中，进入 bin 目录，并运行 spring --version 命令，如果出现 Spring CLI v1.3.3.RELEASE，则说明安装成功，如图 3-3 所示。

图 3-3　运行测试图

为了快速体验 Boot 命令行的强大功能，下面通过一个实例来讲解。使用 Groovy 脚本重新编写上文中的快速入门实例，如代码清单 3-5 所示。

代码清单 3-5　BbsDaemon.groovy

```
@RestController
class BbsDaemon {
    @RequestMapping("/")
    String index() {
        "欢迎光临小春论坛!"
    }
}
```

为了方便测试，直接将编写好的 BbsDaemon.groovy 文件放到命令行工具的 bin 目录中，使用 spring run 命令运行当前的 BbsDaemon 应用，如图 3-4 所示。

图 3-4　运行应用

执行命令之后，Boot 首先会到中心仓库下载相关的依赖包，根据网络速度的不同，可能需要几分钟。待所有的依赖包下载成功后，Boot 会采用内嵌的 Tomcat 启动当前应用。在浏览器地址栏中输入“http://localhost:8080/”，如果返回“欢迎光临小春论坛!”，则说明当前应用已经成功启动。

3.3.4　代码包结构规划

虽然 Spring Boot 没有强制要求工程代码结构按某种方式进行组织，但为了编写代码的可读性，建议采用图 3-5 所示的包组织方式。

图 3-5　包组织方式

建议将应用的主类放在主包外层，将 Application 应用主类放在 com.smart 主包下。这个主类声明了 main()方法，并在类级别上标注@Configuration、@ComponentScan、

@EnableAutoConfiguration 注解。在 Spring Boot 1.2+中可以使用@SpringBootApplication 注解代替上面 3 个注解。其他子包规划包括 Web、Service、Domain、DAO 等。一个典型的 Application 主类如代码清单 3-6 所示。

代码清单 3-6　Application主类

```
package com.smart;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

//@Configuration
//@ComponentScan
//@EnableAutoConfiguration
@SpringBootApplication
public class Application {
   public static void main(String[] args) throws Exception {
      SpringApplication.run(Application.class, args);
   }
}
```

Spring Boot 启动应用非常简单，只需在 main()方法中通过 SpringApplication.run()方法启动即可。

3.4　持久层

Spring 框架提供了几种可选的操作数据库方式，可以直接使用 Spring 内置轻量级的 JdbcTemplate，也可以使用第三方持久化框架 Hibernate 或 MyBaits。Spring Boot 为这两种操作数据库方式分别提供了相应的启动器 spring-boot-starter-jdbc 和 spring-boot-starter-jpa。应用 Spring Boot 启动器使数据库持久化操作变得更加简单，因为 Spring Boot 会自动配置访问数据库相关设施。只需在工程模块 pom.xml 文件中添加 spring-boot-starter-data-jdbc 或 spring-boot-starter-data-jpa 依赖即可。下面将采用 Boot 提供的 JDBC 启动器来实现第 2 章登录示例的持久层。

3.4.1　初始化配置

要使用 Spring Boot 提供的 JDBC 启动器，首先要在模块 pom.xml 文件中导入 spring-boot-starter-data-jdbc 依赖及访问数据库的 JDBC 驱动器，如代码清单 3-7 所示。

代码清单 3-7　pom.xml依赖配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.3.RELEASE</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>chapter3</artifactId>
    <name>Spring4.x第三章实例</name>
    <dependencies>
        …
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
        </dependency>
    </dependencies>
    …
</project>
```

导入依赖包之后，为了让 Spring Boot 能够自动装配数据源的连接，需要在资源根目录 resources 下创建一个 application.properties，配置数据库的连接信息，如代码清单 3-8 所示。

代码清单 3-8　application.properties配置

```
#①-1配置数据库连接信息
spring.datasource.url=jdbc:mysql://localhost:3306/sampledb
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

#①-2指定自定义连接池
#spring.datasource.type=org.apache.commons.dbcp2.BasicDataSource

#①-3连接池配置信息
spring.datasource.max-wait=10000
spring.datasource.max-active=50
spring.datasource.max-idle=10
spring.datasource.min-idle=8
spring.datasource.max-wait=10000
spring.datasource.max-active=100
spring.datasource.test-on-borrow=true
spring.datasource.validation-query=select 1

#②配置JNDI数据源
#spring.datasource.jndi-name= java:comp/env/jdbc/sampleDs

#③初始化数据库脚本
#spring.datasource.initialize=true
#spring.datasource.platform=mysql
```

```
#spring.datasource.data=data
#spring.datasource.schema=schema
```

在 Spring Boot 中，可以通过两种方式配置数据库连接。一种是通过自定义连接的方式，如在配置文件①-1 处，通过配置 spring.datasource.*选项设定数据源的链接地址、连接驱动器、用户名及密码。在默认情况下，Boot 启动器自动创建 tomcat-jdbc 连接池。如果不想采用默认的连接池，则可以通过 spring.datasource.type 属性手工指定项目所需的连接池（如 DBCP、C3P0）。

另外一种是通过 JNDI 方式设置，在生产环境中通常会采用此种方式。如在示例②处，为 spring.datasource.jndi-name 属性指定一个 JNDI 连接名称即可。

在 Boot 中提供了灵活的数据库初始化方式，可以设定 DDL 脚本，也可以设定 DML 脚本。如示例③处，spring.datasource.initialize 属性设置启动的时候是否进行初始化；spring.datasource.platform 属性设置当前数据库类型（如 Oracle、MySQL、SQL Server 等）；spring.datasource.data 属性设置 DML 脚本文件名称，在启动的时候会从类根路径加载 data-${platform}.sql 文件执行，其中${platform}为当前数据库类型，本示例配置会加载 data-mysql.sql；spring.datasource.schema 属性设置 DDL 脚本文件名称，在启动的时候会从类根路径加载 schema-mysql.sql 文件执行。

3.4.2 UserDao

与第 2 章中的 UserDao 一样，首先来定义访问 User 的 DAO，它包括 3 个方法。

- getMatchCount()：根据用户名和密码获取匹配的用户数。等于 1 表示用户名/密码正确；等于 0 表示用户名或密码错误（这是最简单的用户身份认证方法，在实际应用中需要采用诸如密码加密等安全策略）。
- findUserByUserName()：根据用户名获取 User 对象。
- updateLoginInfo()：更新用户积分、最后登录 IP 及最后登录时间。

下面通过 Spring JDBC 技术实现这个 DAO 类，如代码清单 3-9 所示。

代码清单 3-9　UserDao

```
@Repository
public class UserDao {
    private JdbcTemplate jdbcTemplate;

    public int getMatchCount(String userName, String password) {
     …
    }

    public User findUserByUserName(final String userName) {
      …
    }

    public void updateLoginInfo(User user) {
      …
    }
```

```
    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

这里用@Repository 定义了一个 DAO Bean，用@Autowired 将 Spring 容器中的 Bean 注入进来。这个写法与第 2 章中的示例一致。细心的读者可能会有疑问：采用 Boot 方式与普通的方式不是一样的吗？

大家可以回顾一下第 2 章的持久层，写完了业务操作 DAO 之后，还有一个重要的步骤，就是要在 Spring 容器中装配 DAO。

在 Spring Boot 中，这个步骤就不需要了，Boot 会自动帮我们装配好。这就是 Spring Boot 的强大之处，开发人员只需关注业务的实现，而无须关注 Bean 装配等配置，这也是 Spring Boot 设计的初衷。

3.5 业务层

在论坛登录实例中，业务层仅有一个业务类，即 UserService。UserService 负责将持久层的 UserDao 和 LoginLogDao 组织起来，完成用户/密码认证、登录日志记录等操作。这一步与第 2 章的实现业务逻辑一致，这里就不再讲解，如代码清单 3-10 所示。

代码清单 3-10　UserService

```
package com.smart.service;

import com.smart.dao.LoginLogDao;
import com.smart.dao.UserDao;
import com.smart.domain.LoginLog;
import com.smart.domain.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class UserService {
    private UserDao userDao;
    private LoginLogDao loginLogDao;

    public boolean hasMatchUser(String userName, String password) {
      …
    }

    public User findUserByUserName(String userName) {
      …
    }

    @Transactional //事务增强
```

```
    public void loginSuccess(User user) {
        …
    }

    @Autowired
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    @Autowired
    public void setLoginLogDao(LoginLogDao loginLogDao) {
        this.loginLogDao = loginLogDao;
    }
}
```

在编写业务层代码时有两个重要的步骤：一是编写正确的业务逻辑；二是对业务事务的管控。在 Spring Boot 中，使用事务非常简单，首先在主类 Application 上标注@EnableTransactionManagement 注解（开启事务支持，相当于 XML 中的<tx:annotation-driven/>配置方式），然后在访问 Service 方法上标注@Transactional 注解即可。如果将@Transactional 注解标注在 Service 类级别上，那么当前 Service 类的所有方法都将被事务增强，建议不要在类级别上标注@Transactional 注解，如代码清单 3-11 所示。

代码清单 3-11　Application主类

```
package com.smart;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication
@EnableTransactionManagement //启用注解事务管理
public class Application {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```

通过@EnableTransactionManagement 注解，Boot 为应用自动装配了事务支持。这对用户并不透明，用户如果想自己定义事务管理器，则在 Application 类中添加一个即可，如代码清单 3-12 所示。

代码清单 3-12　Application主类

```
package com.smart;
…
@SpringBootApplication
@EnableTransactionManagement //启用注解事务管理
public class Application {
    @Bean
    public PlatformTransactionManager txManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
    …
}
```

在 Application 中添加自定义事务管理器方法 txManager()，并在方法上标注@Bean 注

解，此时 Spring Boot 会加载自定义的事务管理器，不会重新实例化其他事务管理器。

如果在实际的项目中需要分布式事务支持，那么，Boot 也提供了很好的支持，它集成了 Atomikos 和 Bitronix 分布式事务处理框架，可以根据需要导入相应的启动器（spring-boot-starter-jta-atomikos 或 spring-boot-starter-jta-bitronix）。

3.6 展现层

业务层和持久层的开发任务已经完成，该是为程序提供界面的时候了。与第 2 章示例中的展现层开发一样，仍然基于 Spring MVC 实现。

3.6.1 配置 pom.xml 依赖

由于在示例中使用 JSP 作为视图，且用到了 JSTL 标签，因此需要再添加相关的依赖包，如代码清单 3-13 所示。

代码清单 3-13 pom.xml依赖配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.3.RELEASE</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>chapter3</artifactId>
     <name>Spring4.x第三章实例</name>
     <packaging>war</packaging>
    <dependencies>
        ...
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-jasper</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
        </dependency>
    </dependencies>
    ...
</project>
```

为了支持 JSP 与 JSTL，我们添加了 tomcat-embed-jasper 及 jstl 两个模块依赖，并将模块的打包方式改为 WAR。

3.6.2　配置 Spring MVC 框架

在 Boot 环境中配置 MVC 很简单，只要将上面的应用启动类 Application 稍作修改即可，如代码清单 3-14 所示。

代码清单 3-14　Application主类

```
package com.smart;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.boot.context.web.SpringBootServletInitializer;
import org.springframework.boot.builder.SpringApplicationBuilder;

@SpringBootApplication
@EnableTransactionManagement
public class Application extends SpringBootServletInitializer {//①
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }

    //②
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }
}
```

在①处继承了 Spring Boot 提供的 Servlet 初始化器 SpringBootServletInitializer，在②处重写了 SpringBootServletInitializer 的 configure()方法。

3.6.3　处理登录请求

首先需要编写的是 LoginController，它负责处理登录请求，完成登录业务，并根据登录成功与否转向欢迎页面或失败页面。这与第 2 章示例 LoginController 一致，如代码清单 3-15 所示。

代码清单 3-15　LoginController

```
package com.smart.web;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
…
@RestController
public class LoginController{
    private UserService userService;

    @RequestMapping(value = {"/","/index.html"}) //可以配置多个映射路径
```

```
    public ModelAndView loginPage(){
        return new ModelAndView("login");
    }

    @RequestMapping(value = "/loginCheck.html")
    public ModelAndView loginCheck(HttpServletRequest request,LoginCommand
loginCommand){
        …
    }

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
}
```

编写好登录控制器 LoginController 之后，接下来配置 MVC 视图映射。首先创建一个文件夹用于存放 JSP 文件。为了统一规范，在 src/main/webapp/WEB-INF 目录下创建一个 jsp 文件夹，并将第 2 章示例中创建的两个页面（login.jsp 或 main.jsp）复制到此目录，如图 3-6 所示。

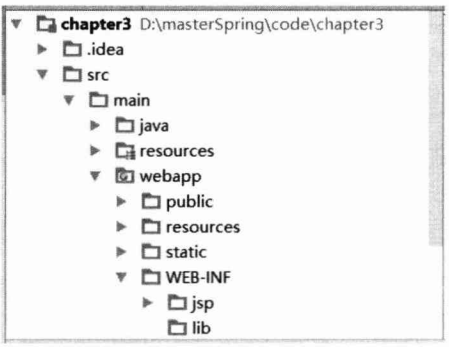


图 3-6　视图目录规划

在默认情况下，Spring Boot 对/static、/public、/resources 或/META-INF/ resources 目录下的静态文件提供支持，所以我们可以将应用中的静态文件（JS、CSS、Image 等）放到这几个目录中。

规划好视图目录后，最后一步就是在 application.properties 中配置创建好的视图的路径，如代码清单 3-16 所示。

代码清单 3-16　application.properties

```
…
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
…
```

通过 spring.mvc.view.prefix 属性指定视图路径的前缀，通过 spring.mvc.view.suffix 属性指定视图文件的后缀。至此，我们就完成了对第 2 章登录示例的改造工作。

最后，通过 Spring Boot 运行应用插件。双击 spring-boot:run 命令，即可启动应用。

启动成功后，可以在控制器中看到“Tomcat started on port(s): 8080 (http)”信息，如图 3-7 所示。

图 3-7 应用启动控制台

在浏览器地址栏中输入“http://localhost:8080/”或“http://localhost:8080/index.html”跳转到登录首页面，如图 3-8 所示。

图 3-8　登录首页面

实战经验

基于 Spring Boot 应用，由于当前应用包含了一个可直接运行的 Application 类，所以在开发过程中，大家很容易在 IDE（如 IDEA 工具）中单击鼠标右键运行当前类。虽然可以启动当前应用，在非 Web 应用中可能不会有什么问题，但在 Web 应用中，如果采用上述方法直接运行应用，那么在访问有视图的页面时（如 JSP），会一直报 404 错误。

因为直接运行当前启动类，Spring Boot 无法找到当前页面资源。因此，基于 Spring Boot 的应用在开发调试的时候，一定要基于 Spring Boot 提供的 spring-boot-maven-plugin 插件命令来运行应用或通过 Spring Boot 命令行来运行应用。

3.7　运维支持

与开发和测试环境不同的是，当应用部署到生产环境时，需要各种运维相关的功能的支持，如监控应用的各种性能指标、运行信息和应用状态等。目前可以通过第三方开

源库实现这些功能，但整合起来还是相当不便。Spring Boot 对这些运维监控相关的类库进行了整合，形成了一个功能完备和可定制的启动器，称为 Actuator。

基于 Spring Boot 应用，添加监控功能非常简单，只需在应用的 pom.xml 文件中添加 spring-boot-starter-actuator 依赖即可，如代码清单 3-17 所示。

代码清单 3-17　pom.xml依赖配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.3.RELEASE</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>chapter3</artifactId>
     <name>Spring4.x第三章实例</name>
     <packaging>war</packaging>
    <dependencies>
        …
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>
    …
</project>
```

Spring Boot 默认提供了对应用本身、关系数据库连接、MongoDB、Redis、Solr、ElasticSearch、JMS 和 Rabbit MQ 等服务的健康状态的检测功能。这些服务都可以在 application.properties 的 management.health.*选项中进行配置，如代码清单 3-18 所示。

代码清单 3-18　application.properties

```
…
#数据库监控配置
management.health.db.enabled=true
management.health.defaults.enabled=true

#应用磁盘空间检查配置
management.health.diskspace.enabled=true
management.health.diskspace.path= D:/masterSpring/code
management.health.diskspace.threshold=0

# ElasticSearch服务健康检查配置
management.health.elasticsearch.enabled=true
management.health.elasticsearch.indices=index1,index2
management.health.elasticsearch.response-timeout=100

# Solr服务健康检查配置
management.health.solr.enabled=true
```

```
#JMS服务健康检查配置
management.health.jms.enabled=true

# Mail服务健康检查配置
management.health.mail.enabled=true

# MongoDB服务健康检查配置
management.health.mongo.enabled=true

# Rabbit MQ服务健康检查配置
management.health.rabbit.enabled=true

# Redis服务健康检查配置
management.health.redis.enabled=true
management.health.status.order=DOWN, OUT_OF_SERVICE, UNKNOWN, UP
…
```

配置好 Actuator 相关依赖及服务健康检查属性配置，重新启动应用，就可以在控制台上看到很多服务映射，如“/health”、“/env”、“/info”等，如图 3-9 所示。

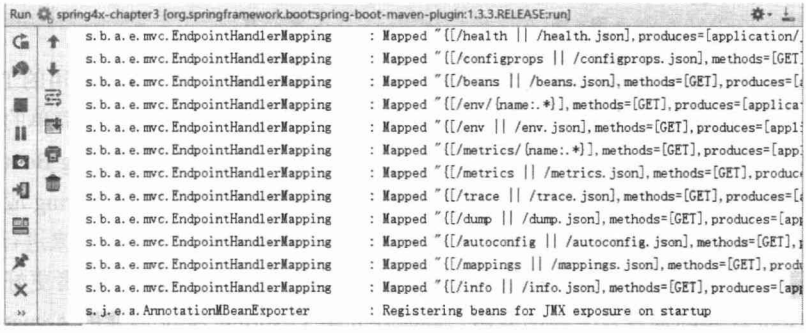


图 3-9　健康检查服务映射信息

在浏览器地址栏中输入其中的一个服务地址“http://localhost:8080/health”，就可以在浏览器中看到如图 3-10 所示的服务信息。

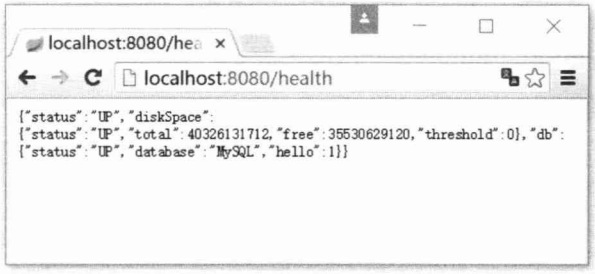


图 3-10　健康检查服务

下面将 Spring Boot 默认提供的健康检查关键服务分别进行说明，方便大家学习使用，如表 3-3 所示。

表 3-3 健康检查服务

服务名称	服务说明
/health	显示应用的健康状态信息
/configprops	显示应用中的配置参数的实际值
/beans	显示应用中包含的 Spring Bean 的信息
/env	显示从 ConfigurableEnvironment 得到的环境配置信息
/metrics	显示应用的性能指标
/trace	显示应用相关的跟踪（trace）信息
/dump	生成一个线程 dump
/autoconfig	显示 Spring Boot 自动配置的信息
/mappings	显示 Spring MVC 应用中通过“@RequestMapping”添加的路径映射
/info	显示应用的基本信息
/shutdown	关闭应用

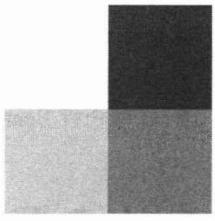
3.8 小结

对于使用 Spring 框架的开发人员来说，Spring Boot 无疑是一个非常实用的工具。本章介绍了 Spring Boot 的发展背景、特点及各种启动器的作用；通过一个简单的入门实例，演示了如何应用 Spring Boot 快速创建 Spring 应用；详细介绍了 Spring Boot 3 种方式的安装配置及代码包结构规划；通过基于依赖的自动配置功能，使得 Spring 应用的配置变得非常简单；在依赖的管理上也变得更加简单，不需要开发人员自己来进行整合。

本章使用 Spring Boot 改造了第 2 章的登录示例，一步步演进，让读者更好地检验 Spring Boot 在实际项目开发中的应用。同时还介绍了 Spring Boot 内建的 Actuator 提供的可以在生产环境中直接使用的性能指标、运行信息和应用管理等功能。Actuator 所提供的功能非常实用，在生产环境下对于应用的监控和管理大有裨益。Spring Boot 应该成为每个使用 Spring 框架的开发人员的首选工具。

第2篇

核心篇



第 4 章

IoC 容器

本章开始讲解 Spring IoC 容器的知识。为了理解 Spring 的 IoC 容器，我们将通过具体的实例详细地讲解 IoC 的概念。同时，本章将对 Java 反射技术进行快速学习，它是 Spring 实现依赖注入的 Java 底层技术，掌握 Java 反射技术有助于读者深刻理解 IoC 的知识，做到知其然并知其所以然。此外，本章还对 Spring 框架的 3 个重要的框架级接口进行了剖析，并对 Bean 的生命周期进行了讲解。通过本章的学习，读者可以掌握依赖注入的设计思想、实现原理，以及几个 Spring IoC 容器级接口的知识。

本章主要内容：

- IoC 概念所包含的设计思想
- Java 反射技术
- BeanFactory、ApplicationContext 及 WebApplicationContext 基础接口
- Bean 的生命周期

本章亮点：

- 通过简单明了的实例逐步讲解 IoC 概念和原理
- 详细分析 Bean 的生命周期并探讨生命周期接口的实际意义

4.1 IoC 概述

IoC（Inverse of Control，控制反转）是 Spring 容器的内核，AOP、声明式事务等功能在此基础上开花结果。但是 IoC 这个重要的概念却比较晦涩难懂，不容易让人望文生义，这不能不说是一大遗憾。不过 IoC 确实包括很多内涵，它涉及代码解耦、设计模式、代码优化等问题的考量，我们试图通过一个小例子来说明这个概念。

4.1.1　通过实例理解 IoC 的概念

贺岁大片在中国已经形成了一个传统，每到年底总会有多部贺岁大片纷至沓来，让人应接不暇。在所有的贺岁大片中，张之亮的《墨攻》算是比较出彩的一部。该片讲述了战国时期墨家人革离帮助梁国反抗赵国侵略的个人英雄主义故事，恢宏壮阔、浑雄凝重的历史场面相当震撼。其中有一个场景，当刘德华所饰演的墨者革离到达梁国都城下时，城上梁国守军问道："来者何人？"刘德华回答："墨者革离！"我们不妨通过 Java 语言为这个"城门叩问"的场景编写剧本，并借此理解 IoC 的概念，如代码清单 4-1 所示。

代码清单 4-1　MoAttack：直接使用演员编排剧本

```
public class MoAttack {
  public void cityGateAsk(){

      //①演员直接侵入剧本
      LiuDeHua ldh = new LiuDeHua();
      ldh.responseAsk("墨者革离！");
  }
}
```

我们会发现，以上剧本在①处，作为具体角色饰演者的刘德华直接侵入剧本，使剧本和演员直接耦合在一起，如图 4-1 所示。

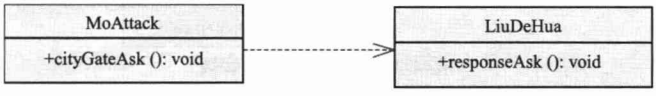


图 4-1　剧本和演员直接耦合

一个明智的编剧在剧情创作时应围绕故事的角色进行，而不应考虑角色的具体饰演者，这样才可能在剧本投拍时自由地遴选任何适合的演员，而非绑定在某一人身上。通过以上分析，我们知道需要为该剧本的主人公革离定义一个接口，如代码清单 4-2 所示。

代码清单 4-2　MoAttack：通过角色编排剧本

```
public class MoAttack {
  public void cityGateAsk(){

      //①引入革离角色接口
      GeLi geli = new LiuDeHua();

      //②通过接口展开剧情
      geli.responseAsk("墨者革离！");
  }
}
```

在①处引入了剧本的角色——革离，剧本的情节通过角色展开，在拍摄时角色由演员饰演，如②处所示。因此，墨攻、革离、刘德华三者的类图关系如图 4-2 所示。

从图 4-2 中可以看出，MoAttack 同时依赖于 GeLi 接口和 LiuDeHua 类，并没有达到我们所期望的剧本仅依赖于角色的目的。但是角色最终必须通过具体的演员才能完成拍摄，如何让 LiuDeHua 和剧本无关而又能完成 GeLi 的具体动作呢？当然是在影片投拍

时，导演将 LiuDeHua 安排在 GeLi 的角色上，导演负责剧本、角色、饰演者三者的协调控制，如图 4-3 所示。

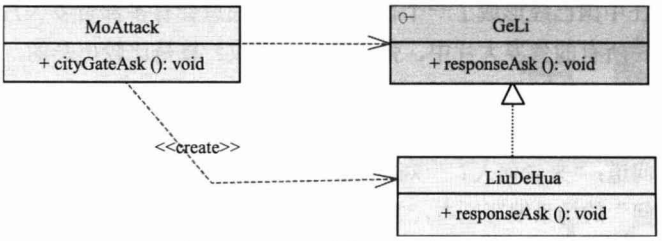


图 4-2　引入角色接口后的关系

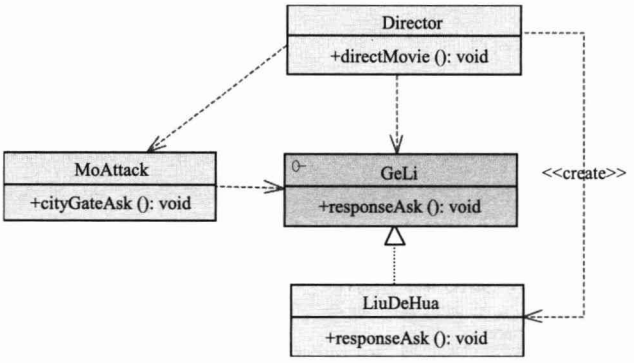


图 4-3　剧本和饰演者解耦

通过引入导演，使得剧本和具体饰演者解耦。对应到软件中，导演就像一台装配器，安排演员表演具体的角色。

现在我们可以反过来讲解 IoC 的概念了。IoC（Inverse of Control）的字面意思是控制反转，它包括两方面的内容：

- 其一是控制。
- 其二是反转。

那到底是什么东西的“控制”被“反转”了呢？对应到前面的例子，“控制”是指选择 GeLi 角色扮演者的控制权；“反转”是指这种控制权从《墨攻》剧本中移除，转交到导演的手中。对于软件来说，即某一接口具体实现类的选择控制权从调用类中移除，转交给第三方决定，即由 Spring 容器借由 Bean 配置来进行控制。

因为 IoC 确实不够开门见山，因此业界曾进行了广泛的讨论，最终软件界的泰斗级人物 Martin Fowler 提出了 DI（Dependency Injection，依赖注入）的概念用来代替 IoC，即让调用类对某一接口实现类的依赖关系由第三方（容器或协作类）注入，以移除调用类对某一接口实现类的依赖。“依赖注入”这个名词显然比“控制反转”直接明了、易于理解。

轻松一刻

名字原本只是一个代号，无所谓好坏，但历史上就有人因名得福，有人因名招祸。永乐二十二年，殿试的结果，状元是孙日恭，榜眼是邢宽。可是发榜时，邢宽成了状元，孙日恭成了探花。原来，日、恭二字连在一起是“暴”字，永乐帝认为不祥，便让他屈居第三。那么，谁做状元呢？永乐帝觉得邢宽这个名字好，“刑”政宽和，必得人心，于是便让邢宽取而代之。

4.1.2　IoC 的类型

从注入方法上看，IoC 主要可以划分为 3 种类型：构造函数注入、属性注入和接口注入。Spring 支持构造函数注入和属性注入。下面我们继续使用以上的例子说明这 3 种注入方式的区别。

1. 构造函数注入

在构造函数注入中，通过调用类的构造函数，将接口实现类通过构造函数变量传入，如代码清单 4-3 所示。

代码清单 4-3　MoAttack：通过构造函数注入革离饰演者

```
public class MoAttack {
  private GeLi geli;

  //①注入革离的具体饰演者
  public MoAttack(GeLi geli){
      this.geli = geli;
  }
  public void cityGateAsk(){
      geli.responseAsk("墨者革离！");
  }
}
```

MoAttack 的构造函数不关心具体由谁来饰演革离这个角色，只要在①处传入的饰演者按剧本要求完成相应的表演即可，角色的具体饰演者由导演来安排，如代码清单 4-4 所示。

代码清单 4-4　Director：通过构造函数注入革离饰演者

```
public class Director {
  public void direct(){

      //①指定角色的饰演者
      GeLi geli = new LiuDeHua();

      //②注入具体饰演者到剧本中
      MoAttack moAttack = new MoAttack(geli);
```

```
        moAttack.cityGateAsk();
    }
}
```

在①处导演安排刘德华饰演革离，并在②处将刘德华“注入”到《墨攻》剧本中，然后开始“城门叩问”剧情的演出工作。

2. 属性注入

有时，导演会发现，虽然革离是影片《墨攻》的第一主角，但并非每个场景都需要革离的出现，在这种情况下通过构造函数注入并不妥当，这时可以考虑使用属性注入。属性注入可以有选择地通过 Setter 方法完成调用类所需依赖的注入，更加灵活方便，如代码清单 4-5 所示。

代码清单 4-5　MoAttack：通过Setter方法注入革离饰演者

```
public class MoAttack {
    private GeLi geli;

    //①属性注入方法
    public void setGeli(GeLi geli) {
        this.geli = geli;
    }
    public void cityGateAsk() {
        geli.responseAsk("墨者革离");
    }
}
```

MoAttack 在①处为 geli 属性提供了一个 Setter 方法，以便让导演在需要时注入 geli 的具体饰演者，如代码清单 4-6 所示。

代码清单 4-6　Director：通过Setter方法注入革离饰演者

```
public class Director {
    public void direct(){

        MoAttack moAttack = new MoAttack();

        //①调用属性Setter方法注入
        GeLi geli = new LiuDeHua();
        moAttack.setGeli(geli);
        moAttack.cityGateAsk();
    }
}
```

和通过构造函数注入革离饰演者不同，在实例化 MoAttack 剧本时，并未指定任何饰演者，而是在实例化 MoAttack 后，在需要革离出场时，才调用其 setGeli()方法注入饰演者。按照类似的方式，还可以分别为剧本中的其他诸如梁王、巷淹中等角色提供注入的 Setter 方法，这样，导演就可以根据所拍剧段的不同，按需注入相应的角色。

3. 接口注入

将调用类所有依赖注入的方法抽取到一个接口中，调用类通过实现该接口提供相应的注入方法。为了采取接口注入的方式，必须先声明一个 ActorArrangable 接口，如下：

```
public interface ActorArrangable {
  void injectGeli(GeLi geli);
}
```

然后，MoAttack 通过 ActorArrangable 接口提供具体的实现，如代码清单 4-7 所示。

代码清单 4-7　MoAttack：通过接口方法注入革离饰演者

```
public class MoAttack implements ActorArrangable {
    private GeLi geli;

    //①实现接口方法
    public void injectGeli (GeLi geli) {
        this.geli = geli;
    }
    public void cityGateAsk() {
        geli.responseAsk("墨者革离");
    }
}
```

Director 通过 ActorArrangable 的 injectGeli()方法完成饰演者的注入工作，如代码清单 4-8 所示。

代码清单 4-8　Director：通过接口方法注入革离饰演者

```
public class Director {
  public void direct(){
      MoAttack moAttack = new MoAttack();
      GeLi geli = new LiuDeHua();
      moAttack. injectGeli (geli);
      moAttack.cityGateAsk();
  }
}
```

由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，因此我们不提倡采用这种注入方式。

4.1.3　通过容器完成依赖关系的注入

虽然 MoAttack 和 LiuDeHua 实现了解耦，MoAttack 无须关注角色实现类的实例化工作，但这些工作在代码中依然存在，只是转移到 Director 类中而已。假设某一制片人想改变这一局面，在选择某个剧本后，希望通过媒体“海选”或者第三方代理机构来选择导演、演员，让他们各司其职，那么剧本、导演、演员就都实现了解耦。

所谓媒体“海选”和第三方代理机构，在程序领域就是一个第三方的容器，它帮助完成类的初始化与装配工作，让开发者从这些底层实现类的实例化、依赖关系装配等工作中解脱出来，专注于更有意义的业务逻辑开发工作。这无疑是一件令人向往的事情。Spring 就是这样的一个容器，它通过配置文件或注解描述类和类之间的依赖关系，自动完成类的初始化和依赖注入工作。下面是 Spring 配置文件对以上实例进行配置的配置文件片段：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w4.org/2001/XMLSchema-instance"
```

```
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
       <!--①实现类实例化-->
   <bean id="geli" class="LiuDeHua"/>
   <bean id="moAttack" class="com.smart.ioc.MoAttack"
        p:geli-ref="geli"/><!--②通过 geli-ref 建立依赖关系-->
</beans>
```

通过 new XmlBeanFactory("beans.xml")等方式即可启动容器。在容器启动时，Spring 根据配置文件的描述信息，自动实例化 Bean 并完成依赖关系的装配，从容器中即可返回准备就绪的 Bean 实例，后续可直接使用之。

Spring 为什么会有这种“神奇”的力量，仅凭一个简单的配置文件，就能魔法般地实例化并装配好程序所用的 Bean 呢？这种“神奇”的力量归功于 Java 语言本身的类反射功能。下面我们独辟章节专门讲解 Java 语言的反射知识，为大家深刻理解 Spring 的技术内幕做好准备。

4.2 相关 Java 基础知识

Java 语言允许通过程序化的方式间接对 Class 进行操作。Class 文件由类装载器装载后，在 JVM 中将形成一份描述 Class 结构的元信息对象，通过该元信息对象可以获知 Class 的结构信息，如构造函数、属性和方法等。Java 允许用户借由这个与 Class 相关的元信息对象间接调用 Class 对象的功能，这就为使用程序化方式操作 Class 对象开辟了途径。

4.2.1 简单实例

我们将从一个简单的例子开始探访 Java 反射机制的征程。下面的 Car 类拥有两个构造函数、一个方法及 3 个属性，如代码清单 4-9 所示。

代码清单 4-9 Car

```
package com.smart.reflect;
public class Car {
    private String brand;
    private String color;
    private int maxSpeed;

    //①默认构造函数
    public Car(){}

    //②带参构造函数
    public Car(String brand,String color,int maxSpeed){
        this.brand = brand;
        this.color = color;
```

```
        this.maxSpeed = maxSpeed;
    }

    //③未带参的方法
    public void introduce() {
      System.out.println("brand:"+brand+";color:"+color+";maxSpeed:" +maxSpeed);
    }
    //省略参数的getter/Setter方法
    ...
}
```

一般情况下，我们会使用如下代码创建 Car 的实例：

```
Car car = new Car();
car.setBrand("红旗 CA72");
```

或者：

```
Car car = new Car("红旗 CA72","黑色");
```

以上两种方法都采用传统方式直接调用目标类的方法。下面我们通过 Java 反射机制以一种间接的方式操控目标类，如代码清单 4-10 所示。

代码清单 4-10　ReflectTest

```
package com.smart. reflect;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
public class ReflectTest {
    public static Car  initByDefaultConst() throws Throwable{

        //①通过类装载器获取Car类对象
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        Class clazz = loader.loadClass("com.smart.reflect.Car");

        //②获取类的默认构造器对象并通过它实例化Car
        Constructor cons = clazz.getDeclaredConstructor((Class[])null);
        Car car = (Car)cons.newInstance();

        //③通过反射方法设置属性
        Method setBrand = clazz.getMethod("setBrand",String.class);
        setBrand.invoke(car,"红旗CA72");
        Method setColor = clazz.getMethod("setColor",String.class);
        setColor.invoke(car,"黑色");
        Method setMaxSpeed = clazz.getMethod("setMaxSpeed",int.class);
        setMaxSpeed.invoke(car,200);
        return car;
    }

    public static void main(String[] args) throws Throwable {
        Car car = initByDefaultConst();
        car.introduce();
    }
}
```

运行以上程序，在控制台上将打印出以下信息：

```
brand:红旗 CA72;color:黑色;maxSpeed:200
```

这说明我们完全可以通过编程方式调用 Class 的各项功能，与通过构造函数和方法直接调用类功能的效果是一致的，只不过前者是间接调用，后者是直接调用罢了。

在 ReflectTest 中使用了几个重要的反射类，分别是 ClassLoader、Class、Constructor 和 Method，通过这些反射类就可以间接调用目标 Class 的各项功能。在①处，我们获取当前线程的 ClassLoader，然后通过指定的全限定类名“com.smart.beans.Car”装载 Car 类对应的反射实例。在②处，我们通过 Car 的反射类对象获取 Car 的构造函数对象 cons，通过构造函数对象的 newInstrance()方法实例化 Car 对象，其效果等同于 new Car()。在③处，我们又通过 Car 的反射类对象的 getMethod(String methodName,Class paramClass)获取属性的 Setter 方法对象，其中第一个参数是目标 Class 的方法名；第二个参数是方法入参的对象类型。在获取方法反射对象后，即可通过 invoke(Object obj,Object param)方法调用目标类的方法，该方法的第一个参数是操作的目标类对象实例，第二个参数是目标方法的入参。

在代码清单 4-10 中，粗体所示部分的信息即通过反射方法操控目标类的元信息，如果我们将这些信息以一个配置文件的方式提供，就可以使用 Java 语言的反射功能编写一段通用的代码，对类似于 Car 的类进行实例化及功能调用操作。

4.2.2 类装载器 ClassLoader

1. 类装载器的工作机制

类装载器就是寻找类的节码文件并构造出类在 JVM 内部表示对象的组件。在 Java 中，类装载器把一个类装入 JVM 中，需要经过以下步骤：

（1）装载：查找和导入 Class 文件。

（2）链接：执行校验、准备和解析步骤，其中解析步骤是可以选择的。

① 校验：检查载入 Class 文件数据的正确性。

② 准备：给类的静态变量分配存储空间。

③ 解析：将符号引用转换成直接引用。

（3）初始化：对类的静态变量、静态代码块执行初始化工作。

类装载工作由 ClassLoader 及其子类负责。ClassLoader 是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入 Class 字节码文件。JVM 在运行时会产生 3 个 ClassLoader：根装载器、ExtClassLoader（扩展类装载器）和 AppClassLoader（应用类装载器）。其中，根装载器不是 ClassLoader 的子类，它使用 C++语言编写，因而在 Java 中看不到它，根装载器负责装载 JRE 的核心类库，如 JRE 目标下的 rt.jar、charsets.jar 等。ExtClassLoader 和 AppClassLoader 都是 ClassLoader 的子类，其中 ExtClassLoader 负责装载 JRE 扩展目录 ext 中的 JAR 类包；AppClassLoader 负责装载 Classpath 路径下的类包。

这 3 个类装载器之间存在父子层级关系，即根装载器是 ExtClassLoader 的父装载器，ExtClassLoader 是 AppClassLoader 的父装载器。在默认情况下，使用 AppClassLoader 装载应用程序的类。我们可以做一个实验，如代码清单 4-11 所示。

代码清单 4-11　ClassLoaderTest

```
public class ClassLoaderTest {
    public static void main(String[] args) {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        System.out.println("current loader:"+loader);
        System.out.println("parent loader:"+loader.getParent());
        System.out.println("grandparent loader:"+loader.getParent(). getParent());
    }
}
```

运行以上代码，在控制台上将打印出以下信息：

```
current loader:sun.misc.Launcher$AppClassLoader@131f71a
parent loader:sun.misc.Launcher$ExtClassLoader@15601ea
//①根装载器在Java中访问不到，所以返回null
grandparent loader:null
```

通过以上输出信息，我们知道当前的 ClassLoader 是 AppClassLoader，其父 ClassLoader 是 ExtClassLoader，祖父 ClassLoader 是根装载器，因为在 Java 中无法获得它的句柄，所以仅返回 null。

JVM 装载类时使用“全盘负责委托机制”，“全盘负责”是指当一个 ClassLoader 装载一个类时，除非显式地使用另一个 ClassLoader，该类所依赖及引用的类也由这个 ClassLoader 载入；“委托机制”是指先委托父装载器寻找目标类，只有在找不到的情况下才从自己的类路径中查找并装载目标类。这一点是从安全角度考虑的，试想，如果有人编写了一个恶意的基础类（如 java.lang.String）并装载到 JVM 中，将会引起多么可怕的后果？但是由于有了“全盘负责委托机制”，java.lang.String 永远是由根装载器来装载的，这样就避免了上述安全隐患的发生。

实战经验

Java 的开发者想必都遇到过 java.lang.NoSuchMethodError 的错误信息吧。究其根源，这个错误基本上都是由 JVM 的“全盘负责委托机制”引发的问题。因为在类路径下放置了多个不同版本的类包，如 commons-lang 2.x.jar 和 commons-lang4.x.jar 都位于类路径中，代码中用到了 commons-lang4.x 类的某个方法，而这个方法在 commons-lang2.x 中并不存在，JVM 加载器碰巧又从 commons-lang 2.x.jar 中加载类，运行时就会抛出 NoSuchMethodError 的错误。

这种问题的排查是比较棘手的，特别是在 Web 应用的情况下，类路径的系统目录比较多，特别是在类包众多时，情况尤其复杂：你很难知道 JVM 到底从哪个类包中加载类文件。对于这个问题，奉上一个终极杀法。

本书配套网盘 tools 下有一个名为 srcAdd.jsp 的程序，将它放到 Web 应用的根路径下，通过如下方式即可查看 JVM 从哪个类包中加载指定类：

```
http://localhost/srcAdd.jsp?className=java.net.URL
```

在 tools 下还有一个 com\smart\utils\ClassLocationUtils.java 类，在 IDEA 断点调试时，

可按 Alt+F8 组合键，弹出 Evaluate Expression 对话框，在 Expression 处输入“ClassLocationUtils.where(<类名>.class)”即可获知当前类是从哪个 JAR 包中加载的。

2. ClassLoader 的重要方法

在 Java 中，ClassLoader 是一个抽象类，位于 java.lang 包中。下面对该类的一些重要接口方法进行介绍。

- Class loadClass(String name)：name 参数指定类装载器需要装载类的名字，必须使用全限定类名，如 com.smart. beans.Car。该方法有一个重载方法 loadClass (String name,boolean resolve)，resolve 参数告诉类装载器是否需要解析该类。在初始化类之前，应考虑进行类解析的工作，但并不是所有的类都需要解析。如果 JVM 只需要知道该类是否存在或找出该类的超类，那么就不需要进行解析。
- Class defineClass(String name, byte[] b, int off, int len)：将类文件的字节数组转换成 JVM 内部的 java.lang.Class 对象。字节数组可以从本地文件系统、远程网络获取。参数 name 为字节数组对应的全限定类名。
- Class findSystemClass(String name)：从本地文件系统载入 Class 文件。如果本地文件系统不存在该 Class 文件，则将抛出 ClassNotFoundException 异常。该方法是 JVM 默认使用的装载机制。
- Class findLoadedClass(String name)：调用该方法来查看 ClassLoader 是否已装入某个类。如果已装入，那么返回 java.lang.Class 对象；否则返回 null。如果强行装载已存在的类，那么将会抛出链接错误。
- ClassLoader getParent()：获取类装载器的父装载器。除根装载器外，所有的类装载器都有且仅有一个父装载器。ExtClassLoader 的父装载器是根装载器，因为根装载器非 Java 语言编写，所以无法获得，将返回 null。

除 JVM 默认的 3 个 ClassLoader 外，用户可以编写自己的第三方类装载器，以实现一些特殊的需求。类文件被装载并解析后，在 JVM 内将拥有一个对应的 java.lang.Class 类描述对象，该类的实例都拥有指向这个类描述对象的引用，而类描述对象又拥有指向关联 ClassLoader 的引用，如图 4-4 所示。

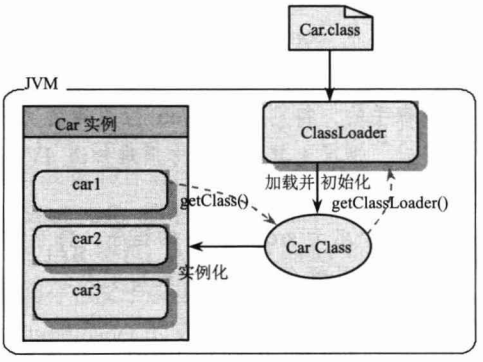


图 4-4 类实例、类描述对象及类装载器的关系

每个类在 JVM 中都拥有一个对应的 java.lang.Class 对象，它提供了类结构信息的描述。数组、枚举、注解及基本 Java 类型（如 int、double 等），甚至 void 都拥有对应的 Class 对象。Class 没有 public 的构造方法。Class 对象是在装载类时由 JVM 通过调用类装载器中的 defineClass()方法自动构造的。

4.2.3　Java 反射机制

Class 反射对象描述类语义结构，可以从 Class 对象中获取构造函数、成员变量、方法类等类元素的反射对象，并以编程的方式通过这些反射对象对目标类对象进行操作。这些反射对象类在 java.reflect 包中定义。下面介绍 3 个主要的反射类。

- Constructor：类的构造函数反射类，通过 Class#getConstructors()方法可以获取类的所有构造函数反射对象数组。在 Java 5.0 中，还可以通过 getConstructor(Class... parameterTypes)获取拥有特定入参的构造函数反射对象。Constructor 的一个主要方法是 newInstance(Object[] initargs)，通过该方法可以创建一个对象类的实例，相当于 new 关键字。在 Java 5.0 中，该方法演化为更为灵活的形式：newInstance (Object... initargs)。
- Method：类方法的反射类，通过 Class#getDeclaredMethods()方法可以获取类的所有方法反射类对象数组 Method[]。在 Java 5.0 中，可以通过 getDeclaredMethod (String name, Class... parameterTypes)获取特定签名的方法，其中 name 为方法名；Class...为方法入参类型列表。Method 最主要的方法是 invoke(Object obj, Object[] args)，其中 obj 表示操作的目标对象；args 为方法入参，代码清单 4-10 中的③处演示了这个反射类的使用方法。在 Java 5.0 中，该方法的形式调整为 invoke (Object obj, Object... args)。此外，Method 还有很多用于获取类方法更多信息的方法。
  - Class getReturnType()：获取方法的返回值类型。
  - Class[] getParameterTypes()：获取方法的入参类型数组。
  - Class[] getExceptionTypes()：获取方法的异常类型数组。
  - Annotation[][] getParameterAnnotations()：获取方法的注解信息，是 Java 5.0 中的新方法。
- Field：类的成员变量的反射类，通过 Class#getDeclaredFields()方法可以获取类的成员变量反射对象数组，通过 Class#getDeclaredField(String name)则可以获取某个特定名称的成员变量反射对象。Field 类最主要的方法是 set(Object obj, Object value)，其中 obj 表示操作的目标对象，通过 value 为目标对象的成员变量设置值。如果成员变量为基础类型，则用户可以使用 Field 类中提供的带类型名的值设置方法，如 setBoolean(Object obj, boolean value)、setInt(Object obj, int value)等。

此外，Java 还为包提供了 Package 反射类，在 Java 5.0 中还为注解提供了 AnnotatedElement 反射类。总之，Java 的反射体系保证了可以通过程序化的方式访问目标类中所有的元素，对于 private 或 protected 成员变量和方法，只要 JVM 的安全机制允许，也可以通过反射进行调用，请看下面的例子，如代码清单 4-12 所示。

代码清单 4-12　PrivateCar

```
package com.smart.reflect;
public class PrivateCar {

  //①private成员变量：使用传统的类实例调用方式，只能在本类中访问
   private String color;

   //②protected方法：使用传统的类实例调用方式，只能在子类和本包中访问
   protected void drive(){
      System.out.println("drive private car! the color is:"+color);
   }
}
```

color 变量和 drive()方法都是私有的，通过类实例变量无法在外部访问私有变量、调用私有方法，但通过反射机制则可以绕过这个限制，如代码清单 4-13 所示。

代码清单 4-13　PrivateCarReflect

```
...
public class PrivateCarReflect {
   public static void main(String[] args) throws Throwable{
       ClassLoader loader = Thread.currentThread().getContextClassLoader();
       Class clazz = loader.loadClass("com.smart.reflect.PrivateCar");
       PrivateCar pcar = (PrivateCar)clazz.newInstance();
       Field colorFld = clazz.getDeclaredField("color");

       //①取消Java语言访问检查以访问private变量
       colorFld.setAccessible(true);
       colorFld.set(pcar,"红色");

       Method driveMtd = clazz.getDeclaredMethod("drive",(Class[])null);
       //Method driveMtd = clazz.getDeclaredMethod("drive"); JDK 5.0下使用

       //②取消Java语言访问检查以访问protected方法
       driveMtd.setAccessible(true);
       driveMtd.invoke(pcar,(Object[])null);
   }
}
```

运行该类，打印出以下信息：

```
drive private car! the color is:红色
```

在访问 private 或 protected 成员变量和方法时，必须通过 setAccessible(boolean access)方法取消 Java 语言检查，否则将抛出 IllegalAccessException。如果 JVM 的安全管理器设置了相应的安全机制，那么调用该方法将抛出 SecurityException。

4.3　资源访问利器

4.3.1　资源抽象接口

JDK 所提供的访问资源的类（如 java.net.URL、File 等）并不能很好地满足各种底层资源的访问需求，比如缺少从类路径或者 Web 容器的上下文中获取资源的操作类。鉴于此，Spring 设计了一个 Resource 接口，它为应用提供了更强的底层资源访问能力。该接口拥有对应不同资源类型的实现类。先来了解一下 Resource 接口的主要方法。

- boolean exists()：资源是否存在。
- boolean isOpen()：资源是否打开。
- URL getURL() throws IOException：如果底层资源可以表示成 URL，则该方法返回对应的 URL 对象。
- File getFile() throws IOException：如果底层资源对应一个文件，则该方法返回对应的 File 对象。
- InputStream getInputStream() throws IOException：返回资源对应的输入流。

Resource 在 Spring 框架中起着不可或缺的作用，Spring 框架使用 Resource 装载各种资源，包括配置文件资源、国际化属性文件资源等。下面我们来了解一下 Resource 的具体实现类，如图 4-5 所示。

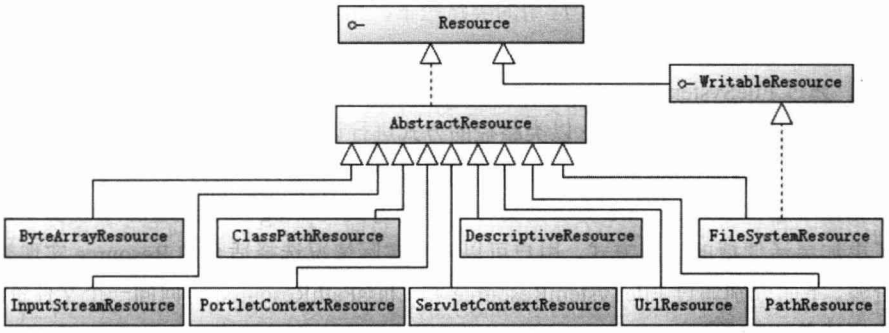


图 4-5　Resource 及其实现类的关系

- WritableResource：可写资源接口，是 Spring 3.1 版本新加的接口，有两个实现类，即 FileSystemResource 和 PathResource，其中 PathResource 是 Spring 4.0 提供的实现类。
- ByteArrayResource：二进制数组表示的资源，二进制数组资源可以在内存中通过程序构造。
- ClassPathResource：类路径下的资源，资源以相对于类路径的方式表示，如代码清单 4-14 所示。

- FileSystemResource：文件系统资源，资源以文件系统路径的方式表示，如D:/conf/bean.xml 等。
- InputStreamResource：以输入流返回表示的资源。
- ServletContextResource：为访问 Web 容器上下文中的资源而设计的类，负责以相对于 Web 应用根目录的路径加载资源。它支持以流和 URL 的方式访问，在 WAR 解包的情况下，也可以通过 File 方式访问。该类还可以直接从 JAR 包中访问资源。
- UrlResource：URL 封装了 java.net.URL，它使用户能够访问任何可以通过 URL 表示的资源，如文件系统的资源、HTTP 资源、FTP 资源等。
- PathResource：Spring 4.0 提供的读取资源文件的新类。Path 封装了 java.net.URL、java.nio.file.Path（Java 7.0 提供）、文件系统资源，它使用户能够访问任何可以通过 URL、Path、系统文件路径表示的资源，如文件系统的资源、HTTP 资源、FTP 资源等。

有了这个抽象的资源类后，就可以将 Spring 的配置信息放置在任何地方（如数据库、LDAP 中），只要最终可以通过 Resource 接口返回配置信息即可。

提示

Spring 的 Resource 接口及其实现类可以在脱离 Spring 框架的情况下使用，它比通过 JDK 访问资源的 API 更好用、更强大。

假设有一个文件位于 Web 应用的类路径下，用户可以通过以下方式对这个文件资源进行访问：

- 通过 FileSystemResource 以文件系统绝对路径的方式进行访问。
- 通过 ClassPathResource 以类路径的方式进行访问。
- 通过 ServletContextResource 以相对于 Web 应用根目录的方式进行访问。

相比于通过 JDK 的 File 类访问文件资源的方式，Spring 的 Resource 实现类无疑提供了更加灵活便捷的访问方式，用户可以根据实际情况选择适合的 Resource 实现类访问资源。下面分别通过 FileSystemResource 和 ClassPathResource 访问同一个文件资源，如代码清单 4-14 所示。

代码清单 4-14　FileSourceExample

```
package com.smart.resource;

import java.io.IOException;
import java.io.InputStream;
import org.springframework.core.io.*;

public class FileSourceExample {

    public static void main(String[] args) {
        try {
```

```
            String filePath =
"D:/masterSpring/code/chapter4/src/main/resources/conf/file1.txt";

            //①使用系统文件路径方式加载文件
            WritableResource res1 = new PathResource(filePath);

            //②使用类路径方式加载文件
            Resource res2 = new ClassPathResource("conf/file1.txt");

            //③使用WritableResource接口写资源文件
            OutputStream stream1 = res1.getOutputStream();
            stream1.write("欢迎光临\n小春论坛".getBytes());
            stream1.close();

            //④使用Resource接口读资源文件
            InputStream ins1 = res1.getInputStream();
            InputStream ins2 = res2.getInputStream();

            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int i;
            while((i=ins1.read())!=-1){
                baos.write(i);
            }
            System.out.println(baos.toString());

         System.out.println("res1:"+res1.getFilename());
         System.out.println("res2:"+res2.getFilename());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

在获取资源后，用户就可以通过 Resource 接口定义的多个方法访问文件的数据和其他信息。如可以通过 getFileName()方法获取文件名，通过 getFile()方法获取资源对应的 File 对象，通过 getInputStream()方法直接获取文件的输入流。通过 WritableResource 接口定义的多个方法向文件写数据，通过 getOutputStream()方法直接获取文件的输出流。此外，还可以通过 createRelative(String relativePath)在资源相对地址上创建新的文件。

在 Web 应用中，用户还可以通过 ServletContextResource 以相对于 Web 应用根目录的方式访问文件资源，如代码清单 4-15 所示。

代码清单 4-15　resource.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<jsp:directive.page
import="org.springframework.web.context.support.ServletContextResource"/>
<jsp:directive.page import="org.springframework.core.io.Resource"/>
<jsp:directive.page import="org.springframework.web.util.WebUtils"/>
<%
   //①注意文件资源地址以相对于Web应用根路径的方式表示
   Resource res3 = new
ServletContextResource(application,"/WEB-INF/classes/conf/file1.txt");
```

```
    out.print(res4.getFilename()+"<br/>");
    out.print(WebUtils.getTempDir(application).getAbsolutePath());
%>
```

对于位于远程服务器（Web 服务器或 FTP 服务器）的文件资源，用户可以方便地通过 UrlResource 进行访问。

资源加载时默认采用系统编码读取资源内容。如果资源文件采用特殊的编码格式，那么可以通过 EncodedResource 对资源进行编码，以保证资源内容操作的正确性，如代码清单 4-16 所示。

代码清单 4-16　FileSourceExample

```
package com.smart.resource;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.EncodedResource;
import org.springframework.util.FileCopyUtils;
public class EncodedResourceExample {
      public static void main(String[] args) throws Throwable  {
          Resource res = new ClassPathResource("conf/file1.txt");
          EncodedResource encRes = new EncodedResource(res,"UTF-8");
          String content  = FileCopyUtils.copyToString(encRes.getReader());
          System.out.println(content);
    }
}
```

4.3.2　资源加载

为了访问不同类型的资源，必须使用相应的 Resource 实现类，这是比较麻烦的。是否可以在不显式使用 Resource 实现类的情况下，仅通过资源地址的特殊标识就可以访问相应的资源呢？Spring 提供了一个强大的加载资源的机制，不但能够通过“classpath:”、“file:”等资源地址前缀识别不同的资源类型，还支持 Ant 风格带通配符的资源地址。

1．资源地址表达式

首先来了解一下 Spring 支持哪些资源类型的地址前缀，如表 4-1 所示。

表 4-1　资源类型的地址前缀

地址前缀	示　例	对应的资源类型
classpath:	classpath: com/smart/beanfactory/beans.xml	从类路径中加载资源，classpath:和 classpath:/是等价的，都是相对于类的根路径。资源文件可以在标准的文件系统中，也可以在 JAR 或 ZIP 的类包中
file:	file:/conf/com/smart/beanfactory/beans.xml	使用 UrlResource 从文件系统目录中装载资源，可采用绝对或相对路径
http://	http: //www. smart.com/resource/beans.xml	使用 UrlResource 从 Web 服务器中装载资源
ftp://	ftp: //www. smart.com/resource/beans.xml	使用 UrlResource 从 FTP 服务器中装载资源
没有前缀	com/smart/beanfactory/beans.xml	根据 ApplicationContext 的具体实现类采用对应类型的 Resource

其中，和“classpath:”对应的还有另一种比较难理解的“classpath*:”前缀。假设有多个 JAR 包或文件系统类路径都拥有一个相同的包名（如 com.smart）。“classpath:”只会在第一个加载的 com.smart 包的类路径下查找，而“classpath*:”会扫描所有这些 JAR 包及类路径下出现的 com. smart 类路径。

这对于分模块打包的应用非常有用。假设一个名为 smart 的应用共分成 3 个模块，一个模块对应一个配置文件，分别是 module1.xml、module2.xml 及 module4.xml，都放到 com.smart 目录下，每个模块单独打成 JAR 包。使用“classpath*:com/smart/module*.xml”可以成功加载这 3 个模块的配置文件，而使用“classpath:com/smart/module*.xml”只会加载一个模块的配置文件。

Ant 风格的资源地址支持 3 种匹配符。

- ?：匹配文件名中的一个字符。
- *：匹配文件名中的任意字符。
- **：匹配多层路径。

下面是几个 Ant 风格的资源路径的示例。

- classpath:com/t?st.xml：匹配 com 类路径下的 com/test.xml、com/tast.xml 或者 com/txst.xml 文件。
- file:D:/conf/*.xml：匹配文件系统 D:/conf 目录下所有以.xml 为后缀的文件。
- classpath:com/**/test.xml：匹配 com 类路径下（当前目录及其子孙目录）的 test.xml 文件。
- classpath:org/springframework/**/*.xml：匹配类路径 org/springframework 下所有以.xml 为后缀的文件。
- classpath:org/**/servlet/bla.xml：不仅匹配类路径 org/springframework/servlet/bla.xml，也匹配 org/springframework/testing/servlet/bla.xml，还匹配 org/servlet/bla.xml。

2．资源加载器

Spring 定义了一套资源加载的接口，并提供了实现类，如图 4-6 所示。

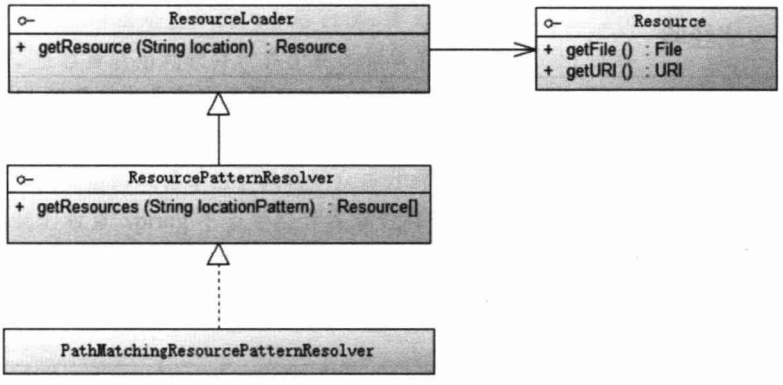


图 4-6　资源加载器接口及实现类

ResourceLoader 接口仅有一个 getResource(String location)方法，可以根据一个资源地址加载文件资源。不过，资源地址仅支持带资源类型前缀的表达式，不支持 Ant 风格的资源路径表达式。ResourcePatternResolver 扩展 ResourceLoader 接口，定义了一个新的接口方法 getResources(String locationPattern)，该方法支持带资源类型前缀及 Ant 风格的资源路径表达式。PathMatchingResourcePatternResolver 是 Spring 提供的标准实现类，来看一个例子，如代码清单 4-17 所示。

代码清单 4-17　ResourceUtilsExample

```
package com.smart.resource;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.core.io.support.ResourcePatternResolver;
import org.testng.annotations.*;
import static org.testng.Assert.*;

public class PatternResolverTest {
    @Test
    public void getResources() throws Throwable{
        ResourcePatternResolver resolver = new
PathMatchingResourcePatternResolver();

        //①加载所有类包com.smart（及子孙包）下以.xml为后缀的资源
        Resource resources[] =resolver.getResources("classpath*:com/smart/**/
*.xml");
        assertNotNull(resources);
        for(Resource resource:resources){
            System.out.println(resource.getDescription());
        }
    }
}
```

由于资源路径是“classpath*:”，所以 PathMatchingResourcePatternResolver 将扫描所有类路径下及 JAR 包中对应 com.smart 类包下的路径，读取所有以.xml 为后缀的文件资源。

实战经验

用 Resource 操作文件时，如果资源配置文件在项目发布时会被打包到 JAR 中，那么不能使用 Resource#getFile()方法，否则会抛出 FileNotFoundException。但可以使用 Resource#getInputStream()方法读取。

错误的读取方式：

```
(new DefaultResourceLoader()).getResource("classpath:conf/sys.properties").
getFile()
```

正确的读取方式：

```
(new DefaultResourceLoader()).getResource("classpath:conf/sys.properties").
getInputStream()
```

这个问题在实际的项目开发过程中很容易被忽视，因为在项目开发时，资源配置文件一般是在文件夹下的，所以 Resource#getFile()是可以正常工作的。但在发布时，如果资源配置文件被打包到 JAR 中，这时 getFile()就无法读取了，从而造成部署实施的时候出现意想不到的问题。因此，我们建议尽量以流的方式读取，避免环境不同造成的问题。

4.4 BeanFactory 和 ApplicationContext

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。Spring 的 IoC 容器在完成这些底层工作的基础上，还提供了 Bean 实例缓存、生命周期管理、Bean 实例代理、事件发布、资源装载等高级服务。

Bean 工厂（com.springframework.beans.factory.BeanFactory）是 Spring 框架最核心的接口，它提供了高级 IoC 的配置机制。BeanFactory 使管理不同类型的 Java 对象成为可能，应用上下文（com.springframework.context.ApplicationContext）建立在 BeanFactory 基础之上，提供了更多面向应用的功能，它提供了国际化支持和框架事件体系，更易于创建实际应用。我们一般称 BeanFactory 为 IoC 容器，而称 ApplicationContext 为应用上下文。但有时为了行文方便，我们也将 ApplicationContext 称为 Spring 容器。

对于二者的用途，我们可以进行简单的划分：BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合都可以直接使用 ApplicationContext 而非底层的 BeanFactory。

轻松一刻

程序开发思想的不断进步使得软件抽象层面越来越高。Spring 框架是生成类对象的工厂，而被创建的类对象本身也可能是一个工厂类，这就形成了所谓“创建工厂的工厂”。

站在钱塘江畔层层叠加的六和塔面前，作为一名富有经验的软件开发者，很容易联想到软件世界中许多相似的事物，如 OSI 网络分层模型、应用系统安全分层模型、Web 应用系统分层模型等。www.theserverside.com 上曾有一篇题为 *Why I Hate Frameworks* 的文章，以幽默诙谐的戏谑手法，讽刺了众多开源框架给开发者带来的困惑。我们希望 Spring 不会带给开发者这样的印象，因为对于使用 Spring 框架的应用来说，虽然软件开发的分层增加了，但框架所提供的底层操作对于上层开发是透明的。只要框架不对上层的应用提出侵入性的硬性要求，开发者就可以借此登高眺远、邀风揽月了。

4.4.1 BeanFactory 介绍

诚如其名，BeanFactory 是一个类工厂，但和传统的类工厂不同，传统的类工厂仅负责构造一个或几个类的实例；而 BeanFactory 是类的通用工厂，它可以创建并管理各种类的对象。这些可被创建和管理的对象本身没有什么特别之处，仅是一个 POJO，Spring 称这些被创建和管理的 Java 对象为 Bean。我们知道 JavaBean 是要满足一定规范的，如必须提供一个默认不带参的构造函数、不依赖于某一特定的容器等，但 Spring 中所说的 Bean 比 JavaBean 更宽泛一些，所有可以被 Spring 容器实例化并管理的 Java 类都可以成为 Bean。

1. BeanFactory 的类体系结构

Spring 为 BeanFactory 提供了多种实现，最常用的是 XmlBeanFactory，但在 Spring 3.2 中已被废弃，建议使用 XmlBeanDefinitionReader、DefaultListableBeanFactory 替代。BeanFactory 的类继承体系设计优雅，堪称经典。通过继承体系，我们可以很容易地了解到 BeanFactory 具有哪些功能，如图 4-7 所示。

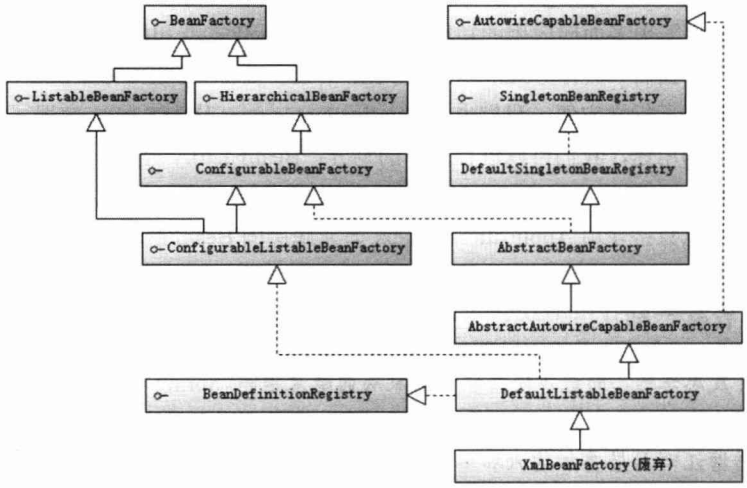


图 4-7 BeanFactory 类继承体系

BeanFactory 接口位于类结构树的顶端，它最主要的方法就是 getBean(String beanName)，该方法从容器中返回特定名称的 Bean。BeanFactory 的功能通过其他接口得到不断扩展。下面对图 4-7 中涉及的其他接口分别进行说明。

- ListableBeanFactory：该接口定义了访问容器中 Bean 基本信息的若干方法，如查看 Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等。
- HierarchicalBeanFactory：父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器。

- ConfigurableBeanFactory：这是一个重要的接口，增强了 IoC 容器的可定制性。它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法。
- AutowireCapableBeanFactory：定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法。
- SingletonBeanRegistry：定义了允许在运行期向容器注册单实例 Bean 的方法。
- BeanDefinitionRegistry：Spring 配置文件中每一个<bean>节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinition Registry 接口提供了向容器手工注册 BeanDefinition 对象的方法。

2. 初始化 BeanFactory

下面使用 Spring 配置文件为 Car 提供配置信息，然后通过 BeanFactory 装载配置文件，启动 Spring IoC 容器。Spring 配置文件如代码清单 4-18 所示。

代码清单 4-18　beans.xml：Car的配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w4.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
    <bean id="car1" class="com.smart.Car"
        p:brand="红旗CA72"
         p:color="黑色"
         p:maxSpeed="200" />
</beans>
```

下面通过 XmlBeanDefinitionReader、DefaultListableBeanFactory 实现类启动 Spring IoC 容器，如代码清单 4-19 所示。

代码清单 4-19　BeanFactoryTest

```
package com.smart.beanfactory;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.core.io.support.ResourcePatternResolver;
import com.smart.Car;
import org.testng.annotations.*;
import static org.testng.Assert.*;

public class BeanFactoryTest {

    @Test
     public void getBean() throws Throwable{
        ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
        Resource res = resolver.getResource("classpath:com/smart/beanfactory/beans.
xml");
        System.out.println(res.getURL());

        //被废弃，不建议使用
```

```
        //BeanFactory bf = new XmlBeanFactory(res);
        DefaultListableBeanFactory factory= new DefaultListableBeanFactory();
        XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions(res);

        System.out.println("init BeanFactory.");

        Car car = factory.getBean("car",Car.class);
        System.out.println("car bean is ready for use!");
        car.introduce();
    }
}
```

XmlBeanDefinitionReader 通过 Resource 装载 Spring 配置信息并启动 IoC 容器，然后就可以通过 BeanFactory#getBean(beanName)方法从 IoC 容器中获取 Bean。通过 BeanFactory 启动 IoC 容器时，并不会初始化配置文件中定义的 Bean，初始化动作发生在第一个调用时。对于单实例（singleton）的 Bean 来说，BeanFactory 会缓存 Bean 实例，所以第二次使用 getBean()获取 Bean 时，将直接从 IoC 容器的缓存中获取 Bean 实例。

Spring 在 DefaultSingletonBeanRegistry 类中提供了一个用于缓存单实例 Bean 的缓存器，它是一个用 HashMap 实现的缓存器，单实例的 Bean 以 beanName 为键保存在这个 HashMap 中。

值得一提的是，在初始化 BeanFactory 时，必须为其提供一种日志框架，我们使用 Log4J，即在类路径下提供 Log4J 配置文件，这样启动 Spring 容器才不会报错。

4.4.2 ApplicationContext 介绍

如果说 BeanFactory 是 Spring 的“心脏”，那么 ApplicationContext 就是完整的“身躯”了。ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。在 BeanFactory 中，很多功能需要以编程的方式实现，而在 ApplicationContext 中则可以通过配置的方式实现。

1. ApplicationContext 类体系结构

ApplicationContext 的主要实现类是 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext，前者默认从类路径加载配置文件，后者默认从文件系统中装载配置文件。下面了解一下 ApplicationContext 的类继承体系，如图 4-8 所示。

从图 4-8 中可以看出，ApplicationContext 继承了 HierarchicalBeanFactory 和 ListableBeanFactory 接口，在此基础上，还通过多个其他的接口扩展了 BeanFactory 的功能。这些接口如下。

- ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了 ApplicationListener 事件监听接口的 Bean 可以接收到容器事件，并对事件进行响应处理。在 ApplicationContext 抽象实现类 AbstractApplicationContext 中存在一个 ApplicationEventMulticaster，它负责保存

所有的监听器，以便在容器产生上下文事件时通知这些事件监听者。

- MessageSource：为应用提供 i18n 国际化消息访问的功能。
- ResourcePatternResolver：所有 ApplicationContext 实现类都实现了类似于 PathMatchingResourcePatternResolver 的功能，可以通过带前缀的 Ant 风格的资源文件路径装载 Spring 的配置文件。
- LifeCycle：该接口提供了 start()和 stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现，ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。

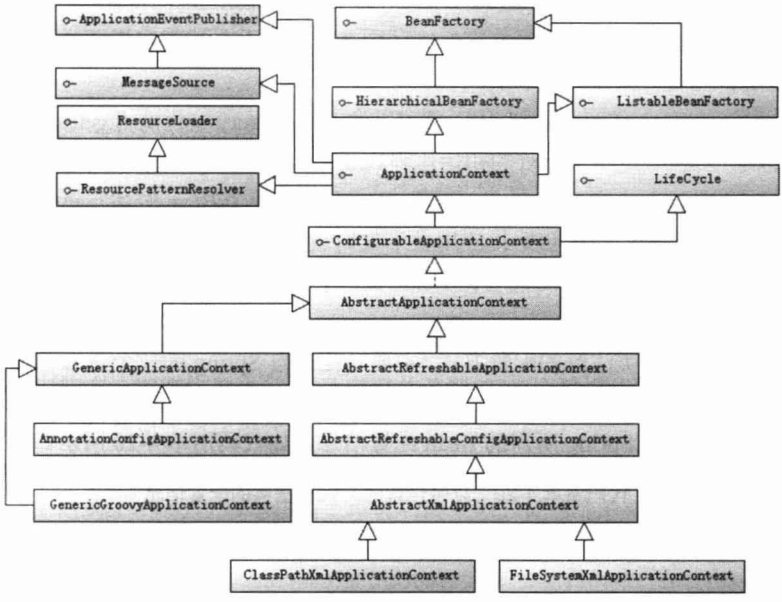


图 4-8　ApplicationContext 类继承体系

ConfigurableApplicationContext 扩展于 ApplicationContext，它新增了两个主要的方法：refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用 refresh()即可启动应用上下文，在已经启动的状态下调用 refresh()则可清除缓存并重新装载配置信息，而调用 close()则可关闭应用上下文。这些接口方法为容器的控制管理带来了便利，但作为开发者，我们并不需要过多关心这些方法。

和 BeanFactory 初始化相似，ApplicationContext 的初始化也很简单。如果配置文件放置在类路径下，则可以优先考虑使用 ClassPathXmlApplicationContext 实现类。

```
ApplicationContext ctx =
        new ClassPathXmlApplicationContext("com/smart/context/beans.xml");
```

对于 ClassPathXmlApplicationContext 来说，“com/smart/context/beans.xml”等同于“classpath: com/smart/context/beans.xml”。

如果配置文件放置在文件系统的路径下，则可以优先考虑使用 FilySystemXml ApplicationContext 实现类。

```
ApplicationContext ctx =
        new FileSystemXmlApplicationContext("com/smart/context/beans.xml");
```

对于 FileSystemXmlApplicationContext 来说，“com/smart/context/beans.xml”等同于“file: com/smart/context/beans.xml”。

还可以指定一组配置文件，Spring 会自动将多个配置文件在内存中“整合”成一个配置文件，如下：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
                        new String[]{"conf/beans1.xml","conf/beans2.xml"});
```

当然，FileSystemXmlApplicationContext 和 ClassPathXmlApplicationContext 都可以显式使用带资源类型前缀的路径，它们的区别在于如果不显式指定资源类型前缀，则分别将路径解析为文件系统路径和类路径。

在获取 ApplicationContext 实例后，就可以像 BeanFactory 一样调用 getBean(beanName) 返回 Bean 了。ApplicationContext 的初始化和 BeanFactory 有一个重大的区别：BeanFactory 在初始化容器时，并未实例化 Bean，直到第一次访问某个 Bean 时才实例化目标 Bean；而 ApplicationContext 则在初始化应用上下文时就实例化所有单实例的 Bean。因此，ApplicationContext 的初始化时间会比 BeanFactory 稍长一些，不过稍后的调用则没有“第一次惩罚”的问题。

Spring 支持基于类注解的配置方式，主要功能来自 Spring 的一个名为 JavaConfig 的子项目。JavaConfig 现已升级为 Spring 核心框架的一部分。一个标注@Configuration 注解的 POJO 即可提供 Spring 所需的 Bean 配置信息，如代码清单 4-20 所示。

代码清单 4-20　以带注解的Java类提供的配置信息

```
package com.smart.context;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.smart.Car;

//①表示是一个配置信息提供类
@Configuration
public class Beans {

    //②定义一个Bean
    @Bean(name = "car")
    public Car buildCar() {
        Car car = new Car();
        car.setBrand("红旗CA72");
        car.setMaxSpeed(200);
        return car;
    }
}
```

和基于 XML 文件的配置方式相比，类注解的配置方式可以很容易地让开发者控制 Bean 的初始化过程，比基于 XML 文件的配置方式更加灵活。

Spring 为基于注解类的配置提供了专门的 ApplicationContext 实现类：AnnotationConfigApplicationContext。来看一个使用 AnnotationConfigApplicationContext 启动 Spring 容器的示例，如代码清单 4-21 所示。

代码清单 4-21　通过带@Configuration的配置类启动容器

```
package com.smart.context;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.smart.Car;
import static org.testng.Assert.*;
import org.testng.annotations.*;

public class AnnotationApplicationContextTest {

    @Test
     public  void getBean() {

        //①通过一个带@Configuration的POJO装载Bean配置
        ApplicationContext ctx = new AnnotationConfigApplicationContext (Beans.class);
        Car car =ctx.getBean("car",Car.class);
        assertNotNull(car);
    }
}
```

AnnotationConfigApplicationContext 将加载 Beans.class 中的 Bean 定义并调用 Beans.class 中的方法实例化 Bean，启动容器并装配 Bean。关于使用 JavaConfig 配置方式的详细内容，将在第 5 章详细介绍。

Spring 4.0 支持使用 Groovy DSL 来进行 Bean 定义配置。其与基于 XML 文件的配置类似，只不过基于 Groovy 脚本语言，可以实现复杂、灵活的 Bean 配置逻辑，来看一个例子，如代码清单 4-22 所示。

代码清单 4-22　groovy-beans.groovy配置信息

```
package com.smart.context;
import com.smart.Car;

beans {
   car(Car) {//①名字（类型）
      brand = "红旗 CA72" //②注入属性
      maxSpeed = "200"
      color = "red"
   }
}
```

基于 Groovy 的配置方式可以很容易地让开发者配置复杂 Bean 的初始化过程，比基于 XML 文件、注解的配置方式更加灵活。

Spring 为基于 Groovy 的配置提供了专门的 ApplicationContext 实现类：GenericGroovyApplicationContext。来看一个如何使用 GenericGroovyApplicationContext 启动 Spring 容器的示例，如代码清单 4-23 所示。

代码清单 4-23　通过GenericGroovyApplicationContext启动容器

```
package com.smart.context;

import com.smart.Car;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericGroovyApplicationContext;
import org.testng.annotations.*;
import static org.testng.Assert.*;

public class GroovyApplicationContextTest {

   @Test
   public  void getBean(){
      ApplicationContext ctx = new
      GenericGroovyApplicationContext("classpath:com/smart/context/groovy-beans.groovy");
      Car car = (Car) ctx.getBean("car");
      assertNotNull(car);
      assertEquals(car.getColor(),"red");
   }
}
```

2．WebApplicationContext 类体系结构

WebApplicationContext 是专门为 Web 应用准备的，它允许从相对于 Web 根目录的路径中装载配置文件完成初始化工作。从 WebApplicationContext 中可以获得 ServletContext 的引用，整个 Web 应用上下文对象将作为属性放置到 ServletContext 中，以便 Web 应用环境可以访问 Spring 应用上下文。Spring 专门为此提供了一个工具类 WebApplicationContextUtils，通过该类的 getWebApplicationContext(ServletContext sc)方法，可以从 ServletContext 中获取 WebApplicationContext 实例。

在非 Web 应用的环境下，Bean 只有 singleton 和 prototype 两种作用域。WebApplicationContext 为 Bean 添加了三个新的作用域：request、session 和 global session。

下面来看一下 WebApplicationContext 的类继承体系，如图 4-9 所示。

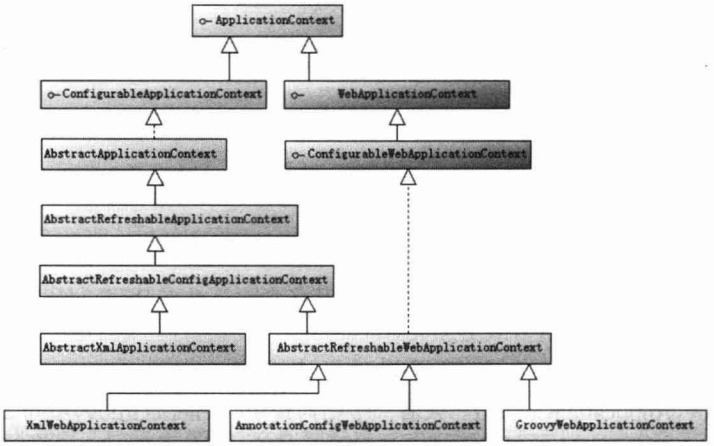


图 4-9　WebApplicationContext 类继承体系

由于 Web 应用比一般的应用拥有更多的特性，因此 WebApplicationContext 扩展了 ApplicationContext。WebApplicationContext 定义了一个常量 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，在上下文启动时，WebApplicationContext 实例即以此为键放置在 ServletContext 的属性列表中，可以通过以下语句从 Web 容器中获取 WebApplicationContext：

```
WebApplicationContext wac = (WebApplicationContext)servletContext.getAttribute(
WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
```

这正是前面提到的 WebApplicationContextUtils 工具类 getWebApplicationContext(ServletContext sc)方法的内部实现方式。这样，Spring 的 Web 应用上下文和 Web 容器的上下文应用就可以实现互访，二者实现了融合，如图 4-10 所示。

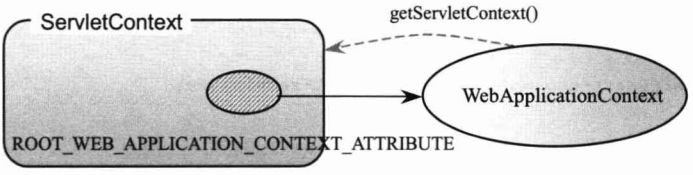


图 4-10　Spring 和 Web 应用的上下文融合

ConfigurableWebApplicationContext 扩展了 WebApplicationContext，它允许通过配置的方式实例化 WebApplicationContext，同时定义了两个重要的方法。

- setServletContext(ServletContext servletContext)：为 Spring 设置 Web 应用上下文，以便二者整合。
- setConfigLocations(String[] configLocations)：设置 Spring 配置文件地址，一般情况下，配置文件地址是相对于 Web 根目录的地址，如/WEB-INF/smart-dao.xml、/WEB-INF/smart-service.xml 等。但用户也可以使用带资源类型前缀的地址，如 classpath:com/smart/beans.xml 等。

3．WebApplicationContext 初始化

WebApplicationContext 的初始化方式和 BeanFactory、ApplicationContext 有所区别，因为 WebApplicationContext 需要 ServletContext 实例，也就是说，它必须在拥有 Web 容器的前提下才能完成启动工作。有过 Web 开发经验的读者都知道，可以在 web.xml 中配置自启动的 Servlet 或定义 Web 容器监听器（ServletContextListener），借助二者中的任何一个，就可以完成启动 Spring Web 应用上下文的工作。

提示

所有版本的 Web 容器都可以定义自启动的 Servlet，但只有 Servlet 2.3 及以上版本的 Web 容器才支持 Web 容器监听器。有些即使支持 Servlet 2.3 的 Web 服务器，也不能在 Servlet 初始化之前启动 Web 监听器，如 Weblogic 8.1、WebSphere 5.x、Oracle OC4J 9.0。

Spring 分别提供了用于启动 WebApplicationContext 的 Servlet 和 Web 容器监听器：

- org.springframework.web.context.ContextLoaderServlet。
- org.springframework.web.context.ContextLoaderListener。

二者的内部都实现了启动 WebApplicationContext 实例的逻辑，只要根据 Web 容器的具体情况选择二者之一，并在 web.xml 中完成配置即可。

代码清单 4-24 是使用 ContextLoaderListener 启动 WebApplicationContext 的具体配置。

代码清单 4-24　通过Web容器监听器引导

```
...
<!--①指定配置文件-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/smart-dao.xml, /WEB-INF/smart-service.xml
   </param-value>
</context-param>

<!--②声明Web容器监听器-->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

ContextLoaderListener 通过 Web 容器上下文参数 contextConfigLocation 获取 Spring 配置文件的位置。用户可以指定多个配置文件，用逗号、空格或冒号分隔均可。对于未带资源类型前缀的配置文件路径，WebApplicationContext 默认这些路径相对于 Web 的部署根路径。当然，也可以采用带资源类型前缀的路径配置，如“classpath*:/smart-*.xml”和上面的配置是等效的。

如果在不支持容器监听器的低版本 Web 容器中，则可以采用 ContextLoaderServlet 完成相同的工作，如代码清单 4-25 所示。

代码清单 4-25　通过自启动的Servlet引导

```
...
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/smart-dao.xml, /WEB-INF/smart-service.xml </param-value>
</context-param>
...
<!--①声明自启动的Servlet -->
<servlet>
   <servlet-name>springContextLoaderServlet</servlet-name>
    <servlet-class>org.springframework.web.context.ContextLoaderServlet
</servlet-class>

   <!--②启动顺序-->
   <load-on-startup>1</load-on-startup>
</servlet>
```

由于 WebApplicationContext 需要使用日志功能，所以用户可以将 Log4J 的配置文件放置在类路径 WEB-INF/classes 下，这时 Log4J 引擎即可顺利启动。如果 Log4J 配置文件放置在其他位置，那么用户必须在 web.xml 中指定 Log4J 配置文件的位置。Spring 为启动 Log4J 引擎提供了两个类似于启动 WebApplicationContext 的实现类：Log4jConfigServlet 和 Log4jConfigListener，不管采用哪种方式，都必须保证能够在装载 Spring 配置文件前先装载 Log4J 配置信息，如代码清单 4-26 所示。

代码清单 4-26　指定Log4J配置文件时启动Spring Web应用上下文

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
   /WEB-INF/smart-dao.xml,/WEB-INF/smart-service.xml
 </param-value>
</context-param>

<!--①指定Log4J配置文件的位置-->
<context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
</context-param>

<!--②装载Log4J配置文件的自启动Servlet -->
<servlet>
    <servlet-name>log4jConfigServlet</servlet-name>
<servlet-class>org.springframework.web.util.Log4jConfigServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
    <servlet-name> springContextLoaderServlet</servlet-name>
<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
<load-on-startup>2</load-on-startup>
   </servlet>
```

注意上面将 log4jConfigServlet 的启动顺序号设置为 1，而将 springContextLoaderServlet 的启动顺序号设置为 2。这样，前者将先启动，完成装载 Log4J 配置文件并初始化 Log4J 引擎的工作，紧接着后者再启动。如果使用 Web 监听器，则必须将 Log4jConfigListener 放置在 ContextLoaderListener 的前面。采用以上配置方式，Spring 将自动使用 XmlWebApplicationContext 启动 Spring 容器，即通过 XML 文件为 Spring 容器提供 Bean 的配置信息。

如果使用标注@Configuration 的 Java 类提供配置信息，则 web.xml 需要按以下方式配置，如代码清单 4-27 所示。

代码清单 4-27　使用标注@Configuration的Java类提供配置信息的配置

```
<web-app>

   <!--通过指定context参数，让Spring使用AnnotationConfigWebApplicationContext而非
   XmlWebApplicationContext启动容器 -->
    <context-param>
```

```
        <param-name>contextClass</param-name>
        <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- 指定标注了@Configuration的配置类，多个可以使用逗号或空格分隔-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
          com.smart.AppConfig1,com.smart.AppConfig1
       </param-value>
    </context-param>

    <!-- ContextLoaderListener监听器将根据上面的配置使用
       AnnotationConfigWebApplicationContext根据contextConfigLocation
       指定的配置类启动Spring容器-->
    <listener>
      <listener-class>
       org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
</web-app>
```

ContextLoaderListener 如果发现配置了 contextClass 上下文参数，就会使用参数所指定的 WebApplicationContext 实现类（AnnotationConfigWebApplicationContext）初始化容器，该实现类会根据 contextConfigLocation 上下文参数指定的标注@Configuration 的配置类所提供的 Spring 配置信息初始化容器。

如果使用 Groovy DSL 配置 Bean 信息，则 web.xml 需要按以下方式配置，如代码清单 4-28 所示。

代码清单 4-28　使用Groovy DSL配置Bean信息

```
<web-app>

  <!--通过指定context参数，让Spring使用GroovyWebApplicationContext而非
   XmlWebApplicationContext或AnnotationConfigWebApplicationContext启动容器 -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
        org.springframework.web.context.support.GroovyWebApplicationContext
        </param-value>
    </context-param>

    <!-- 指定标注了Groovy的配置文件-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
           Classpath*:conf/spring-mvc.groovy
       </param-value>
    </context-param>

    <!-- ContextLoaderListener监听器将根据上面的配置使用
       GroovyWebApplicationContext根据contextConfigLocation
```

```
        指定的配置类启动Spring容器-->
    <listener>
      <listener-class>
        org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
</web-app>
```

GroovyWebApplicationContext 实现类会根据 contextConfigLocation 上下文参数指定的 conf/spring-mvc.groovy 所提供的 Spring 配置信息初始化容器。

4.4.3　父子容器

通过 HierarchicalBeanFactory 接口，Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器中的 Bean。在容器内，Bean 的 id 必须是唯一的，但子容器可以拥有一个和父容器 id 相同的 Bean。父子容器层级体系增强了 Spring 容器架构的扩展性和灵活性，因为第三方可以通过编程的方式为一个已经存在的容器添加一个或多个特殊用途的子容器，以提供一些额外的功能。

Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层 Bean，而业务层和持久层 Bean 则看不到展现层 Bean。

4.5　Bean 的生命周期

我们知道 Web 容器中的 Servlet 拥有明确的生命周期，Spring 容器中的 Bean 也拥有相似的生命周期。Bean 生命周期由多个特定的生命阶段组成，每个生命阶段都开出了一扇门，允许外界借由此门对 Bean 施加控制。

在 Spring 中，可以从两个层面定义 Bean 的生命周期：第一个层面是 Bean 的作用范围；第二个层面是实例化 Bean 时所经历的一系列阶段。下面分别对 BeanFactory 和 ApplicationContext 中 Bean 的生命周期进行分析。

4.5.1　BeanFactory 中 Bean 的生命周期

1. 生命周期图解

由于 Bean 的生命周期所经历的阶段比较多，下面将通过图形化的方式进行描述。图 4-11 描述了 BeanFactory 中 Bean 生命周期的完整过程。

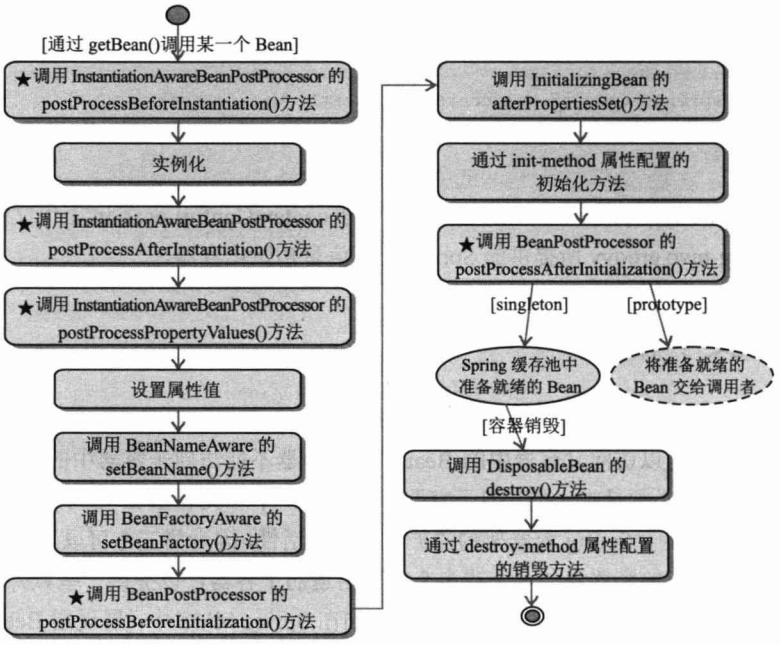


图 4-11　BeanFactory 中 Bean 的生命周期

具体过程如下。

（1）当调用者通过 getBean(beanName)向容器请求某一个 Bean 时，如果容器注册了 org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor 接口，则在实例化 Bean 之前，将调用接口的 postProcessBeforeInstantiation()方法。

（2）根据配置情况调用 Bean 构造函数或工厂方法实例化 Bean。

（3）如果容器注册了 InstantiationAwareBeanPostProcessor 接口，那么在实例化 Bean 之后，调用该接口的 postProcessAfterInstantiation()方法，可在这里对已经实例化的对象进行一些“梳妆打扮”。

（4）如果 Bean 配置了属性信息，那么容器在这一步着手将配置值设置到 Bean 对应的属性中，不过在设置每个属性之前将先调用 InstantiationAwareBeanPostProcessor 接口的 postProcessPropertyValues()方法。

（5）调用 Bean 的属性设置方法设置属性值。

（6）如果 Bean 实现了 org.springframework.beans.factory.BeanNameAware 接口，则将调用 setBeanName()接口方法，将配置文件中该 Bean 对应的名称设置到 Bean 中。

（7）如果 Bean 实现了 org.springframework.beans.factory.BeanFactoryAware 接口，则将调用 setBeanFactory()接口方法，将 BeanFactory 容器实例设置到 Bean 中。

（8）如果 BeanFactory 装配了 org.springframework.beans.factory.config.BeanPostProcessor 后处理器，则将调用 BeanPostProcessor 的 Object postProcessBeforeInitialization(Object bean, String beanName)接口方法对 Bean 进行加工操作。其中，入参 bean 是当前正在处

理的 Bean，而 beanName 是当前 Bean 的配置名，返回的对象为加工处理后的 Bean。用户可以使用该方法对某些 Bean 进行特殊的处理，甚至改变 Bean 的行为。BeanPostProcessor 在 Spring 框架中占有重要的地位，为容器提供对 Bean 进行后续加工处理的切入点，Spring 容器所提供的各种“神奇功能”（如 AOP、动态代理等）都通过 BeanPostProcessor 实施。

（9）如果 Bean 实现了 InitializingBean 接口，则将调用接口的 afterPropertiesSet()方法。

（10）如果在<bean>中通过 init-method 属性定义了初始化方法，则将执行这个方法。

（11）BeanPostProcessor 后处理器定义了两个方法：其一是 postProcessBeforeInitialization()，在第（8）步调用；其二是 Object postProcessAfterInitialization(Object bean, String beanName)，这个方法在此时调用，容器再次获得对 Bean 进行加工处理的机会。

（12）如果在<bean>中指定 Bean 的作用范围为 scope="prototype"，则将 Bean 返回给调用者，调用者负责 Bean 后续生命的管理，Spring 不再管理这个 Bean 的生命周期。如果将作用范围设置为 scope="singleton"，则将 Bean 放入 Spring IoC 容器的缓存池中，并将 Bean 引用返回给调用者，Spring 继续对这些 Bean 进行后续的生命管理.

（13）对于 scope="singleton"的 Bean（默认情况），当容器关闭时，将触发 Spring 对 Bean 后续生命周期的管理工作。如果 Bean 实现了 DisposableBean 接口，则将调用接口的 destory()方法，可以在此编写释放资源、记录日志等操作。

（14）对于 scope="singleton"的 Bean，如果通过<bean>的 destroy-method 属性指定了 Bean 的销毁方法，那么 Spring 将执行 Bean 的这个方法，完成 Bean 资源的释放等操作。

Bean 的完整生命周期从 Spring 容器着手实例化 Bean 开始，直到最终销毁 Bean。其中经过了许多关键点，每个关键点都涉及特定的方法调用，可以将这些方法大致划分为 4 类。

- Bean 自身的方法：如调用 Bean 构造函数实例化 Bean、调用 Setter 设置 Bean 的属性值及通过<bean>的 init-method 和 destroy-method 所指定的方法。
- Bean 级生命周期接口方法：如 BeanNameAware、BeanFactoryAware、InitializingBean 和 DisposableBean，这些接口方法由 Bean 类直接实现。
- 容器级生命周期接口方法：在图 4-11 中带“★”的步骤是由 InstantiationAwareBean PostProcessor 和 BeanPostProcessor 这两个接口实现的，一般称它们的实现类为“后处理器”。后处理器接口一般不由 Bean 本身实现，它们独立于 Bean，实现类以容器附加装置的形式注册到 Spring 容器中，并通过接口反射为 Spring 容器扫描识别。当 Spring 容器创建任何 Bean 的时候，这些后处理器都会发生作用，所以这些后处理器的影响是全局性的。当然，用户可以通过合理地编写后处理器，让其仅对感兴趣的 Bean 进行加工处理。
- 工厂后处理器接口方法：包括 AspectJWeavingEnabler、CustomAutowireConfigurer、ConfigurationClassPostProcessor 等方法。工厂后处理器也是容器级的，在应用上下文装配配置文件后立即调用。

Bean 级生命周期接口和容器级生命周期接口是个性和共性辩证统一思想的体现，前者解决 Bean 个性化处理的问题，而后者解决容器中某些 Bean 共性化处理的问题。

Spring 容器中是否可以注册多个后处理器呢？答案是肯定的。只要它们同时实现 org.springframework.core.Ordered 接口，容器将按特定的顺序依次调用这些后处理器。所以图 4-11 中带“★”的步骤都可能调用多个后处理器进行一系列加工操作。

InstantiationAwareBeanPostProcessor 其实是 BeanPostProcessor 接口的子接口，Spring 为其提供了一个适配器类 InstantiationAwareBeanPostProcessorAdapter，一般情况下，可以方便地扩展该适配器覆盖感兴趣的方法以定义实现类。下面将通过一个具体的实例来更好地理解 Bean 生命周期的各个步骤。

2. 窥探 Bean 生命周期的实例

依旧采用前面介绍的 Car 类，让它实现所有 Bean 级的生命周期接口。此外，还定义了初始化和销毁的方法，这两个方法将通过<bean>的 init-method 和 destroy-method 属性指定，如代码清单 4-29 所示。

代码清单 4-29　实现各种生命周期控制访问的Car

```
package com.smart;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

//①管理Bean生命周期的接口
public class Car implements BeanFactoryAware, BeanNameAware, InitializingBean,
        DisposableBean {
    private String brand;
    private String color;
    private int maxSpeed;

    private BeanFactory beanFactory;
    private String beanName;

    public Car() {
        System.out.println("调用Car()构造函数。");
    }
    public void setBrand(String brand) {
        System.out.println("调用setBrand()设置属性。");
        this.brand = brand;
    }

    public void introduce() {
        System.out.println("brand:" + brand + ";color:" + color + ";maxSpeed:"
                + maxSpeed);
    }
```

```
    //②BeanFactoryAware接口方法
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("调用BeanFactoryAware.setBeanFactory()。");
        this.beanFactory = beanFactory;
    }

    //③BeanNameAware接口方法
    public void setBeanName(String beanName) {
        System.out.println("调用BeanNameAware.setBeanName()。");
        this.beanName = beanName;
    }

    //④InitializingBean接口方法
    public void afterPropertiesSet() throws Exception {
        System.out.println("调用InitializingBean.afterPropertiesSet()。");
    }

    //⑤DisposableBean接口方法
    public void destroy() throws Exception {
        System.out.println("调用DisposableBean.destroy()。");
    }

    //⑥通过<bean>的init-method属性指定的初始化方法
    public void myInit() {
        System.out.println("调用init-method所指定的myInit()，将maxSpeed设置为240。");
        this.maxSpeed = 240;
    }

    //⑦通过<bean>的destroy-method属性指定的销毁方法
    public void myDestroy() {
        System.out.println("调用destroy-method所指定的myDestroy()。");
    }
}
```

Car 类在②、③、④、⑤处实现了 BeanFactoryAware、BeanNameAware、InitializingBean、DisposableBean 这些 Bean 级的生命周期控制接口；在⑥和⑦处定义了 myInit()和 myDestroy()方法，以便在配置文件中通过 init-method 和 destroy-method 属性定义初始化和销毁方法。

MyInstantiationAwareBeanPostProcessor 通过扩展 InstantiationAwareBeanPostProcessor 适配器 InstantiationAwareBeanPostProcessorAdapter 提供实现，如代码清单 4-30 所示。

代码清单 4-30　InstantiationAwareBeanPostProcessor实现类

```
package com.smart.beanfactory;
import java.beans.PropertyDescriptor;
import org.springframework.beans.BeansException;
import org.springframework.beans.PropertyValues;
import
org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessorAdapte
r;
import com.smart.Car;
public class MyInstantiationAwareBeanPostProcessor extends InstantiationAwareBeanPostProcessorAdapter{
```

```
    //①接口方法：在实例化Bean前调用
    public Object postProcessBeforeInstantiation(Class beanClass, String beanName)
         throws BeansException {

        //①-1仅对容器中的car Bean处理
         if("car".equals(beanName)){
            System.out.println("InstantiationAware BeanPostProcessor. postProcess
                ↪BeforeInstantiation");
         }
         return null;
    }

    //②接口方法：在实例化Bean后调用
    public boolean postProcessAfterInstantiation(Object bean, String beanName)
         throws BeansException {

        //②-1仅对容器中的car Bean进行处理
         if("car".equals(beanName)){
            System.out.println("InstantiationAware BeanPostProcessor.postProcess
                ↪AfterInstantiation");
         }
         return true;
    }

    //③接口方法：在设置某个属性时调用
    public PropertyValues postProcessPropertyValues(
            PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName)
            throws BeansException {

            //③-1仅对容器中的car Bean进行处理，还可以通过pdst入参进行过滤，
            //仅对car的某个特定属性值进行处理
             if("car".equals(beanName)){
              System.out.println("Instantiation AwareBeanPostProcessor.postProcess
                ↪PropertyValues");
         }
         return pvs;
    }
}
```

在 MyInstantiationAwareBeanPostProcessor 中，通过过滤条件仅对 car Bean 进行处理，对其他的 Bean 一概视而不见。

此外，还提供了一个 BeanPostProcessor 实现类，在该实现类中仅对 car Bean 进行处理，对配置文件所提供的属性设置值进行判断，并执行相应的“补缺补漏”操作，如代码清单 4-31 所示。

代码清单 4-31　BeanPostProcessor实现类

```
package com.smart.beanfactory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import com.smart.Car;
public class MyBeanPostProcessor implements BeanPostProcessor{
```

```
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if(beanName.equals("car")){
            Car car = (Car)bean;
            if(car.getColor() == null){
                System.out.println("调用BeanPostProcessor.postProcess
BeforeInitialization(),
                    color为空，设置为默认黑色。");
                car.setColor("黑色");
            }
        }
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        if(beanName.equals("car")){
            Car car = (Car)bean;
            if(car.getMaxSpeed() >= 200){
                System.out.println("调用BeanPostProcessor.postProcess
AfterInitialization(),
                    将maxSpeed调整为200。");
                car.setMaxSpeed(200);
            }
        }
        return bean;
    }
}
```

在 MyBeanPostProcessor 类的 postProcessBeforeInitialization()方法中，首先判断所处理的 Bean 是否名为 car，如果是，则进一步判断该 Bean 的 color 属性是否为空；如果为空，则将该属性设置为“黑色”。在 postProcessAfterInitialization()方法中，仅对名为 car 的 Bean 进行处理，判断其 maxSpeed 是否超过最大速度 200，如果超过，则将其设置为 200。

至于如何将 MyInstantiationAwareBeanPostProcessor 和 MyBeanPostProcessor 这两个后处理器注册到 BeanFactory 容器中，请参看代码清单 4-32。

代码清单 4-32　beans.xml：配置Car

```
<bean id="car" class="com.smart.Car"
      init-method="myInit"
      destroy-method="myDestroy"
      p:brand="红旗CA72"
     p:maxSpeed="200"/>
```

通过 init-method 指定 Car 的初始化方法为 myInit()；通过 destroy-method 指定 Car 的销毁方法为 myDestroy()；同时通过 scope 定义了 Car 的作用范围（关于 Bean 作用范围的详细讨论，请参见 5.8 节）。

下面让容器装载配置文件，然后分别注册上面所提供的两个后处理器，如代码清单 4-33 所示。

代码清单 4-33 BeanLifeCycle

```
package com.smart.beanfactory;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import com.smart.Car;

public class BeanLifeCycle {
    private static void LifeCycleInBeanFactory(){

      //①下面两句装载配置文件并启动容器
       Resource res = new ClassPathResource("com/smart/beanfactory/beans.xml");

      BeanFactory bf= new DefaultListableBeanFactory();
      XmlBeanDefinitionReader reader = new
                    XmlBeanDefinitionReader((DefaultListableBeanFactory)bf);
      reader.loadBeanDefinitions(res);

      //②向容器中注册MyBeanPostProcessor后处理器
      ((ConfigurableBeanFactory)bf).addBeanPostProcessor(new MyBeanPostProcessor());

      //③向容器中注册MyInstantiationAwareBeanPostProcessor后处理器
      ((ConfigurableBeanFactory)bf).addBeanPostProcessor(
            new MyInstantiationAwareBeanPostProcessor());

      //④第一次从容器中获取car，将触发容器实例化该Bean，这将引发Bean生命周期方法的调用
      Car car1 = (Car)bf.getBean("car");
      car1.introduce();
      car1.setColor("红色");

      //⑤第二次从容器中获取car，直接从缓存池中获取
      Car car2 = (Car)bf.getBean("car");

      //⑥查看car1和car2是否指向同一引用
      System.out.println("car1==car2:"+(car1==car2));

      //⑦关闭容器
      ((DefaultListableBeanFactory)bf).destroySingletons();
    }
     public static void main(String[] args) {
         LifeCycleInBeanFactory();
     }
}
```

在①处，装载了配置文件并启动容器。在②处，向容器中注册了 MyBeanPostProcessor 后处理器，注意对 BeanFactory 类型的 bf 变量进行了强制类型转换，因为用于注册后处理器的 addBeanPostProcessor()方法是在 ConfigurableBeanFactory 接口中定义的。如果有多个后处理器，则可以按照相似的方式调用 addBeanPostProcessor()方法进行注册。需要强调的是，后处理器的实际调用顺序和注册顺序是无关的，在具有多个后处理器的情况下，必须通过实现的 org.springframework.core.Ordered 接口来确定调用顺序。

在③处，按照注册 MyBeanPostProcessor 后处理器相同的方法注册 MyInstantiationAwareBeanPostProcessor 后处理器，Spring 容器会自动检查后处理器是否实现了 InstantiationAwareBeanPostProcessor 接口，并据此判断后处理器的类型。

在④处，第一次从容器中获取 car Bean，容器将按图 4-11 中描述的 Bean 生命周期过程，实例化 Car 并将其放入缓存池中，然后再将这个 Bean 引用返回给调用者。在⑤处，再次从容器中获取 car Bean，Bean 将从容器缓存池中直接取出，不会引发生命周期相关方法的执行。如果 Bean 的作用范围定义为 scope="prototype"，则第二次 getBean() 时，生命周期方法会再次被调用，因为 prototype 范围的 Bean 每次都返回新的实例。在⑥处，检验 car1 和 car2 是否指向相同的对象。

运行 BeanLifeCycle，在控制台上得到以下输出信息：

```
InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation
调用 Car()构造函数。
InstantiationAwareBeanPostProcessor.postProcessAfterInstantiation
InstantiationAwareBeanPostProcessor.postProcessPropertyValues
调用 setBrand()设置属性。
调用 BeanNameAware.setBeanName()。
调用 BeanFactoryAware.setBeanFactory()。
调用 BeanPostProcessor.postProcessBeforeInitialization()，color 为空，设置为默认黑色。
调用 InitializingBean.afterPropertiesSet()。
调用 myInit()，将 maxSpeed 设置为 240。
调用 BeanPostProcessor.postProcessAfterInitialization()，将 maxSpeed 调整为 200。
brand:奇瑞 QQ;color:黑色;maxSpeed:200
brand:奇瑞 QQ;color:红色;maxSpeed:200
2016-01-03 15:47:10,640 INFO [main] (DefaultSingletonBeanRegistry.java:272) -
Destroying singletons in {org.springframework.beans.factory.xml.XmlBeanFactory
defining beans [car]; root of BeanFactory hierarchy}
调用 DisposableBean.destroy()。
调用 myDestroy()。
```

仔细观察输出的信息，发现其验证了前面所介绍的 Bean 生命周期的完整过程。在⑦处，通过 destroySingletons()方法关闭了容器，由于 Car 实现了销毁接口并指定了销毁方法，所以容器将触发调用这两个方法。

3．关于 Bean 生命周期接口的探讨

通过实现 Spring 的 Bean 生命周期接口对 Bean 进行额外控制，虽然让 Bean 具有了更细致的生命周期阶段，但也带来了一个问题：Bean 和 Spring 框架紧密地绑定在一起，这和 Spring 一直推崇的“不对应用程序类作任何限制”的理念是相悖的。因此，如果用户希望将业务类完全 POJO 化，则可以只实现自己的业务接口，不需要和某个特定框架（包括 Spring 框架）的接口关联。可以通过<bean>的 init-method 和 destroy-method 属性配置方式为 Bean 指定初始化和销毁的方法，采用这种方式对 Bean 生命周期的控制效果和通过实现 InitializingBean 和 DisposableBean 接口所达到的效果是完全相同的。采用前者的配置方式可以使 Bean 不需要和特定的 Spring 框架接口绑定，达到了框架解耦的目的。此外，Spring 还拥有一个 Bean 后置处理器 InitDestroyAnnotationBeanPostProcessor，它负责对标注了@PostConstruct、@PreDestroy 的 Bean 进行处理，在 Bean 初始化后及销

毁前执行相应的逻辑。喜欢注解的读者，可以通过 InitDestroyAnnotationBeanPost Processor 达到和以上两种方式相同的效果（如果在 ApplicationContext 中，则已经默认装配了该处理器）。

对于 BeanFactoryAware 和 BeanNameAware 接口，前者让 Bean 感知容器（BeanFactory 实例），而后者让 Bean 获得配置文件中对应的配置名称。一般情况下，用户几乎不需要关心这两个接口。如果 Bean 希望获取容器中的其他 Bean，则可以通过属性注入的方式引用这些 Bean；如果 Bean 希望在运行期获知在配置文件中的 Bean 名称，则可以简单地将名称作为属性注入。

综上所述，我们认为，除非编写一个基于 Spring 之上的扩展插件或子项目之类的东西，否则用户完全可以抛开以上 4 个 Bean 生命周期的接口类，使用更好的方案替代之。

但 BeanPostProcessor 接口却不一样，它不要求 Bean 去继承它，可以完全像插件一样注册到 Spring 容器中，为容器提供额外的功能。Spring 容器充分利用了 BeanPostProcessor 对 Bean 进行加工处理，当我们讲到 Spring 的 AOP 功能时，还会对此进行分析，了解 BeanPostProcessor 对 Bean 的影响，对于深入理解 Spring 核心功能的工作机理将会有很大的帮助。很多 Spring 扩展插件或 Spring 子项目都是使用这些后处理器完成激动人心的功能的。

4.5.2 ApplicationContext 中 Bean 的生命周期

Bean 在应用上下文中的生命周期和在 BeanFactory 中的生命周期类似，不同的是，如果 Bean 实现了 org.springframework.context.ApplicationContextAware 接口，则会增加一个调用该接口方法 setApplicationContext()的步骤，如图 4-12 所示。

此外，如果在配置文件中声明了工厂后处理器接口 BeanFactoryPostProcessor 的实现类，则应用上下文在装载配置文件之后、初始化 Bean 实例之前将调用这些 BeanFactoryPostProcessor 对配置信息进行加工处理。Spring 框架提供了多个工厂后处理器，如 CustomEditorConfigurer、PopertyPlaceholderConfigurer 等，我们将在第 5 章中详细介绍它们的功用。如果在配置文件中定义了多个工厂后处理器，那么最好让它们实现 org.springframework.core.Ordered 接口，以便 Spring 以确定的顺序调用它们。工厂后处理器是容器级的，仅在应用上下文初始化时调用一次，其目的是完成一些配置文件的加工处理工作。

ApplicationContext 和 BeanFactory 另一个最大的不同之处在于：前者会利用 Java 反射机制自动识别出配置文件中定义的 BeanPostProcessor、InstantiationAwareBeanPost Processor 和 BeanFactoryPostProcessor，并自动将它们注册到应用上下文中；而后者需要在代码中通过手工调用 addBeanPostProcessor()方法进行注册。这也是为什么在应用开发时普遍使用 ApplicationContext 而很少使用 BeanFactory 的原因之一。

在 ApplicationContext 中，只需在配置文件中通过<bean>定义工厂后处理器和 Bean 后处理器，它们就会按预期的方式运行。

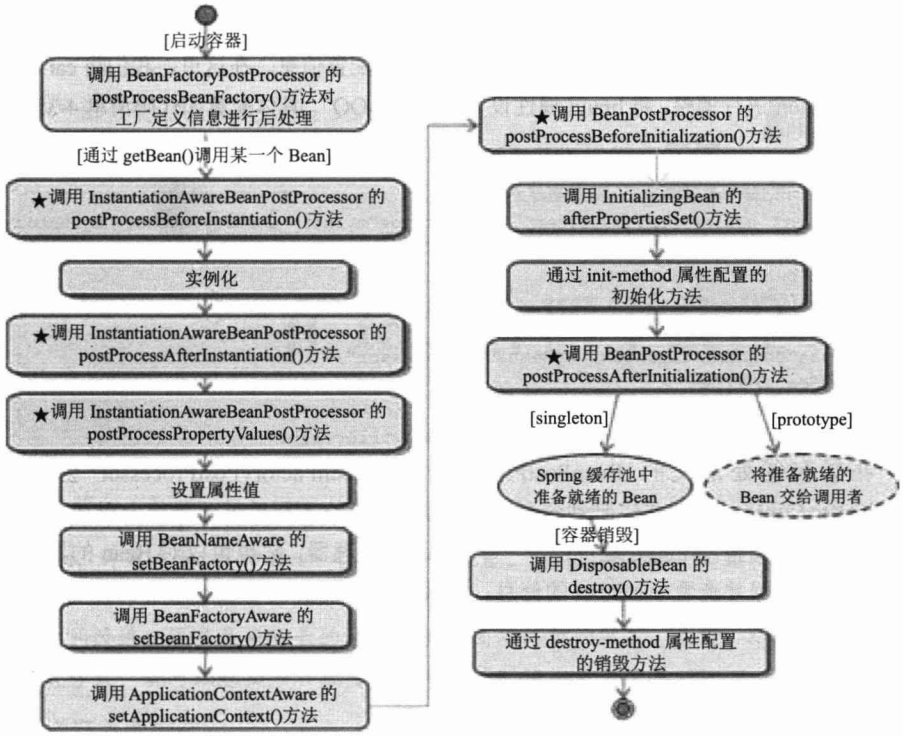


图 4-12　ApplicationContext 中 Bean 的生命周期

来看一个使用工厂后处理器的实例。假设我们希望对配置文件中 car 的 brand 配置属性进行调整，则可以编写一个如代码清单 4-34 所示的工厂后处理器。

代码清单 4-34　工厂后处理器：MyBeanFactoryPostProcessor.java

```
package com.smart.context;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanFactoryPost Processor;
import org.springframework.beans.factory.config.ConfigurableListable BeanFactory;
import com.smart.Car;

public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor{

  //①对car <bean>的brand属性配置信息进行“偷梁换柱”的加工操作
  public void postProcessBeanFactory(ConfigurableListableBeanFactory bf)
        throws BeansException {
        BeanDefinition bd = bf.getBeanDefinition("car");

       bd.getPropertyValues().addPropertyValue("brand", "奇瑞QQ");
        System.out.println("调用BeanFactoryPostProcessor.postProcessBean Factory()!");
    }
}
```

ApplicationContext 在启动时，将首先为配置文件中的每个<bean>生成一个 BeanDefinition 对象，BeanDefinition 是<bean>在 Spring 容器中的内部表示。当配置文件中

所有的<bean>都被解析成 BeanDefinition 时，ApplicationContext 将调用工厂后处理器的方法，因此，我们有机会通过程序的方式调整 Bean 的配置信息。在这里，我们将 car 对应的 BeanDefinition 进行调整，将 brand 属性设置为“奇瑞 QQ”，具体配置如代码清单 4-35 所示。

代码清单 4-35　beans.xml

```
<!--①这个brand属性的值将被工厂后处理器更改掉-->
<bean id="car" class="com.smart.Car" init-method="myInit" destroy-method="myDestory"
      p:brand="红旗CA72"
      p:maxSpeed="200"/>
<!--②工厂后处理器-->
<bean id="myBeanPostProcessor"
      class="com.smart.context.MyBeanPostProcessor"/>
<!--③注册Bean后处理器-->
<bean id="myBeanFactoryPostProcessor"
      class="com.smart.context.MyBeanFactoryPostProcessor"/>
```

在②和③处定义的 BeanPostProcessor 和 BeanFactoryPostProcessor 会自动被 ApplicationContext 识别并注册到容器中。在②处注册的工厂后处理器将会对在①处配置的属性值进行调整。在③处还声明了一个 Bean 后处理器，它也可以对 Bean 的属性进行调整。启动容器并查看 car Bean 的信息，将发现 car Bean 的 brand 属性成功被工厂后处理器更改了。

4.6　小结

在本章中，我们深入分析了 IoC 的概念。控制反转的概念其实包含两个层面的意思：“控制”是接口实现类的选择控制权；而“反转”是指这种选择控制权从调用类转移到外部第三方类或容器的手中。

为了揭开 Spring 依赖注入的神秘面纱，透视 Spring 的机理，我们对 Java 语言的反射技术进行了快速学习。掌握了这些知识，读者不但可以深刻理解 Spring 的内部实现机制，还可以自己动手编写一个 IoC 容器。

BeanFactory、ApplicationContext 和 WebApplicationContext 是 Spring 框架的 3 个核心接口，框架中其他大部分的类都围绕它们展开，为它们提供支持和服务。在这些支持类中，Resource 是一个不可忽视的重要接口，框架通过 Resource 实现了和具体资源的解耦，不论它们位于何种存储介质中，都可以通过相同的实例返回。与 Resource 配合的另一个接口是 ResourceLoader，ResourceLoader 采用了策略模式，可以通过传入资源地址的信息，自动选择适合的底层资源实现类，为上层对资源的引用提供了极大的便利。

Spring 为 Bean 提供了细致周全的生命周期过程，通过实现特定的接口或通过<bean>属性设置，都可以对 Bean 的生命周期过程施加影响。Bean 的生命周期不但和其实现的接口相关，还与 Bean 的作用范围有关。为了让 Bean 绑定在 Spring 框架上，我们推荐使用配置方式而非接口方式进行 Bean 生命周期的控制。

第 5 章

在 IoC 容器中装配 Bean

在使用 Spring 所提供的各项丰富而神奇的功能之前，必须在 Spring IoC 容器中装配好 Bean，并建立 Bean 和 Bean 之间的关联关系。Spring 的 Bean 配置文件虽然已经很简单，但广大的开发者希望它做得更好。Spring 对这个呼声给予了高度的重视，进行了许多重大的改进，很多原来冗长的配置拥有了简洁优雅的版本。此外，Spring 还提供了多种配置方式，既可以选择一种配置，也可以同时使用多种配置。

本章主要内容：

- 如何使用基于 Schema 格式的配置
- 依赖注入的类型和配置方式
- 各种注入参数的详细讲解
- Bean 的作用域
- FactoryBean 的作用
- 基于注解的配置
- 基于 Java 类的配置
- 基于 Groovy DSL 的配置
- 通过编码方式动态添加 Bean

本章亮点：

- 解答了因 XML 语法或 JavaBean 规范特殊知识点而引用的配置难题
- 对可达到相同目的的多种配置方式从实际应用角度进行了比较分析

5.1 Spring 配置概述

5.1.1 Spring 容器高层视图

要使应用程序中的 Spring 容器成功启动，需要同时具备以下三方面的条件：

- Spring 框架的类包都已经放到应用程序的类路径下。
- 应用程序为 Spring 提供了完备的 Bean 配置信息。
- Bean 的类都已经放到应用程序的类路径下。

Spring 启动时读取应用程序提供的 Bean 配置信息，并在 Spring 容器中生成一份相应的 Bean 配置注册表，然后根据这张注册表实例化 Bean，装配好 Bean 之间的依赖关系，为上层应用提供准备就绪的运行环境。

Bean 配置信息是 Bean 的元数据信息，它由以下 4 个方面组成：

- Bean 的实现类。
- Bean 的属性信息，如数据源的连接数、用户名、密码等。
- Bean 的依赖关系，Spring 根据依赖关系配置完成 Bean 之间的装配。
- Bean 的行为配置，如生命周期范围及生命周期各过程的回调函数等。

Bean 元数据信息在 Spring 容器中的内部对应物是由一个个 BeanDefinition 形成的 Bean 注册表，Spring 实现了 Bean 元数据信息内部表示和外部定义的解耦。Spring 支持多种形式的 Bean 配置方式。Spring 1.0 仅支持基于 XML 的配置，Spring 2.0 新增基于注解配置的支持，Spring 3.0 新增基于 Java 类配置的支持，而 Spring 4.0 则新增基于 Groovy 动态语言配置的支持。

图 5-1 描述了 Spring 容器、Bean 配置信息、Bean 实现类及应用程序四者的相互关系。

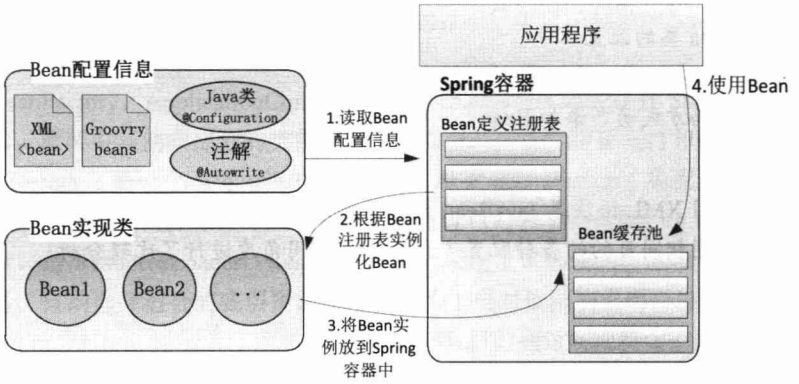


图 5-1 Spring 容器内部协作解构

Bean 配置信息首先定义了 Bean 的实现及依赖关系，Spring 容器根据各种形式的

Bean 配置信息在容器内部建立 Bean 定义注册表；然后根据注册表加载、实例化 Bean，并建立 Bean 和 Bean 之间的依赖关系；最后将这些准备就绪的 Bean 放到 Bean 缓存池中，以供外层的应用程序进行调用。

5.1.2　基于 XML 的配置

对于基于 XML 的配置，Spring 1.0 的配置文件采用 DTD 格式，Spring 2.0 以后采用 Schema 格式，后者让不同类型的配置拥有了自己的命名空间，使得配置文件更具扩展性。此外，Spring 基于 Schema 配置方案为许多领域的问题提供了简化的配置方法，配置工作因此得到了大幅简化。

采取基于 Schema 的配置格式，文件头的声明会复杂一些，先看一个简单的示例，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"
xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation=
   "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

    <!--默认命名空间的配置-->
    <bean id="foo" class="com.smart.Foo"/>

   <!--aop命名空间的配置-->
    <aop:config>
        <aop:advisor pointcut="execution(* *..PetStoreFacade.*(..))"
advice-ref="txAdvice"/>
    </aop:config>
</beans>
```

①默认命名空间

②xsi 标准命名空间，用于指定自定义命名空间的 Schema 文件

③-1 自定义命名空间，aop 是该命名空间的简称

③-2 命名空间全称，必须在 xsi 命名空间为其指定空间对应的 Schema 文件，参见④

④为每个命名空间指定具体的 Schema 文件

要了解文件头所声明的内容，需要学习一些 XML Schema 的知识。Schema 在文档根节点中通过 xmlns 对文档所引用的命名空间进行声明。在上面的代码中定义了 3 个命名空间。

① 默认命名空间：它没有空间名，用于 Spring Bean 的定义。

② xsi 标准命名空间：这个命名空间用于为每个文档中的命名空间指定相应的 Schema 样式文件，是 W3C 定义的标准命名空间。

③ aop 命名空间：这个命名空间是 Spring 配置 AOP 的命名空间，即一种自定义的命名空间。

命名空间的定义分为两个步骤：第一步指定命名空间的名称；第二步指定命名空间的 Schema 文档格式文件的位置，用空格或回车换行进行分隔。

在第一步中，需要指定命名空间的缩略名和全名，请看下面配置所定义的命名空间：

```
xmlns:aop="http://www.springframework.org/schema/aop"
```

aop 为命名空间的别名，一般使用简洁易记的名称，文档后面的元素可通过命名空间别名加以区分，如<aop:config/>等。而 http://www.springframework.org/schema/aop 为空间的全限定名，习惯上用文档发布机构的官方网站和相关网站目录作为全限定名。这种命名方式既可以标识文档所属的机构，又可以很好地避免重名的问题。但从 XML Schema 语法来说，别名和全限定名都可以任意命名。

如果命名空间的别名为空，则表示该命名空间为文档默认命名空间。文档中无命名空间前缀的元素都属于默认命名空间，如<beans/>、<bean/>等都属于在①处定义的默认命名空间。

在第二步中，为每个命名空间指定了对应的 Schema 文档格式的定义文件，定义的语法如下：

```
<命名空间 1>[ ] <命名空间 1Schema 文件>[ ]<命名空间 2>[ ]<命名空间 2Schema 文件>
```

命名空间使用全限定名，每个组织机构在发布 Schema 文件后，都会为该 Schema 文件提供一个引用的 URL 地址，一般使用这个 URL 地址指定命名空间对应的 Schema 文件。命名空间名称和对应的 Schema 文件地址之间使用空格或回车分隔，不同的命名空间之间也使用这种分隔方法。

指定命名空间的 Schema 文件地址有两个用途：其一，XML 解析器可以获取 Schema 文件并对文档进行格式合法性验证；其二，在开发环境下，IDE 可以引用 Schema 文件对文档编辑提供诱导功能（自动补全功能）。当然，这个 Schema 文件的远程地址并非一定能够访问，一般的 IDE 都提供了从本地类路径查找 Schema 文件的功能，只有找不到时才从远程加载。

Spring 4.0 配置的 Schema 文件放置在各模块 JAR 文件内一个名为 config 的目录下。表 5-1 对这些 Schema 文件的用途进行了说明。

表 5-1　Spring 4.0 的Schema文件

Schema 文件	说　明
spring-beans-4.0.xsd	[说明]：Spring 4.0 最主要的 Schema，用于配置 Bean [命名空间]：http://www.springframework.org/schema/beans [Schema 文件]：http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
spring-aop-4.0.xsd	[说明]：AOP 的配置定义的 Schema [命名空间]：http://www.springframework.org/schema/aop [Schema 文件]：http://www.springframework.org/schema/aop/spring-aop-4.0.xsd

续表

目　录	说　明
spring-tx-4.0.xsd	[说明]：声明式事务配置定义的 Schema [命名空间]：http://www.springframework.org/schema/tx [Schema 文件]：http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
spring-mvc-4.0.xsd	[说明]：MVC 配置的 Schema，是 Spring 3.0 新增的 [命名空间]：http://www.springframework.org/schema/mvc [Schema 文件]：http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
spring-util-4.0.xsd	[说明]：为简化某些复杂的标准配置提供的 Schema [命名空间]：http://www.springframework.org/schema/util [Schema 文件]：http://www.springframework.org/schema/util/spring-util-4.0.xsd
spring-jee-4.0.xsd	[说明]：为简化 Java EE 中 EJB、JNDI 等功能的配置而提供的 Schema [命名空间]：http://www.springframework.org/schema/jee [Schema 文件]：http://www.springframework.org/schema/jee/spring-jee-4.0.xsd
spring-jdbc-4.0.xsd	[说明]：为配置 Spring 内嵌数据库提供的 Schema，是 Spring 3.0 新增的 [命名空间]：http://www.springframework.org/schema/jdbc [Schema 文件]：http://www.springframework.org/schema/jdbc/spring-jdbc-4.0.xsd
spring-jms-4.0.xsd	[说明]：JMS 配置的 Schema [命名空间]：http://www.springframework.org/schema/jms [Schema 文件]：http://www.springframework.org/schema/jms/spring-jms-4.0.xsd
spring-lang-4.0.xsd	[说明]：增加了对 JRuby 和 Groovy 等动态语言的支持，该 Schema 是为集成动态语言而定义的 [命名空间]：http://www.springframework.org/schema/lang [Schema 文件]：http://www.springframework.org/schema/lang/spring-lang-4.0.xsd
spring-oxm-4.0.xsd	[说明]：配置对象 XML 映射的 Schema，是 Spring 3.0 新增的 [命名空间]：http://www.springframework.org/schema/oxm [Schema 文件]：http://www.springframework.org/schema/oxm/spring-oxm-4.0.xsd
spring-task-4.0.xsd	[说明]：任务调度的 Schema [命名空间]：http://www.springframework.org/schema/task [Schema 文件]：http://www.springframework.org/schema/task/spring-task-4.0.xsd
spring-tool-4.0.xsd	[说明]：为集成 Spring 的一些有用工具定义的 Schema [命名空间]：http://www.springframework.org/schema/tool [Schema 文件]：http://www.springframework.org/schema/tool/spring-tool-4.0.xsd

虽然 Spring 为 AOP、声明事务、Java EE 都提供了专门的 Schema XML 配置，但 Spring 也允许继续使用低版本的基于 DTD 的 XML 配置方式。Spring 4.0 配置的升级是向后兼容的，但我们强烈建议使用新的基于 Schema 的配置方式。

除支持 XML 配置方式外，Spring 还支持基于注解、Java 类及 Groovy 的配置方式，不同的配置方式在“质”上是基本相同的，只是存在“形”的区别。由于基于 XML 的配置方式是最基础、最传统的，所以我们主要以基于 XML 的配置方式讲解 Spring 的配置，其他 3 种配置方式则作简要介绍。

5.2 Bean 基本配置

在进行 Bean 配置的详细讲解之前，先来了解一下 Bean 配置的基础知识，以快速建立起 Bean 配置的初步概念。

5.2.1 装配一个 Bean

在 Spring 容器的配置文件中定义一个简要 Bean 的配置片段如图 5-2 所示。

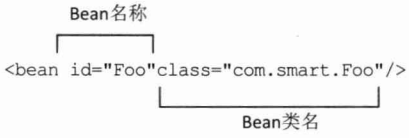


图 5-2 Bean 的定义

一般情况下，Spring IoC 容器中的一个 Bean 对应配置文件中的一个<bean>，这种镜像映射关系应该容易理解。其中，id 为这个 Bean 的名称，通过容器的 getBean("foo")即可获取对应的 Bean，在容器中起到定位查找的作用，是外部程序和 Spring IoC 容器进行交互的桥梁；class 属性指定了 Bean 对应的实现类。

下面基于 XML 的配置文件定义了两个简单的 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-4.0.xsd">
    <bean id="car" class="com.smart.simple.Car"/>
    <bean id="boss" class="com.smart.simple.Boss"/>
</beans>
```

正如读者所看到的，这段配置信息提供了实例化 Car 和 Boss 这两个 Bean 必需的信息，Spring IoC 容器完全可以据此创建这两个 Bean 的实例。

5.2.2 Bean 的命名

一般情况下，在配置一个 Bean 时，需要为其指定一个 id 属性作为 Bean 的名称。id 在 IoC 容器中必须是唯一的，而且 id 的命名需要满足 XML 对 id 的命名规范(id 是 XML 规定的特殊属性)：必须以字母开始，后面可以是字母、数字、连字符、下画线、句号、冒号等完整结束（full stops）的符号，逗号和空格这些非完整结束符是非法的。在实际情况下，id 命名约束并不会给用户带来影响，但如果用户确实希望用一些特殊字符进行 Bean 命名，则可以使用<bean>的 name 属性。name 属性没有字符上的限制，几乎可以使用任何字符，如?ab、123 等，示例如下：

```
<bean name="#car1" class="com.smart.simple.Car"/>
```

id 和 name 都可以指定多个名字，名字之间可用逗号、分号或者空格进行分隔，如下：

```
<bean name="#car1,123,$car" class="com.smart.simple.Car"/>
```

这里为 Bean 定义了 3 个名称：其一为#car1；其二为 123；其三为$car。用户可以使用 getBean("#car1")、getBean("123")或 getBean("$car")获取 IoC 容器中的 car Bean。

Spring 配置文件不允许出现两个相同 id 的<bean>，但却可以出现两个相同 name 的<bean>。如果有多个 name 相同的<bean>，那么通过 getBean(beanName)获取 Bean 时，将返回后面声明的那个 Bean，原因是后面的 Bean 覆盖了前面同名的 Bean。所以为了避免无意间 Bean 覆盖的隐患，应尽量使用 id 而非 name 命名 Bean。

如果 id 和 name 两个属性都未指定，如<bean class="com.smart.simple.Car"/>，那么 Spring 自动将全限定类名作为 Bean 的名称，这时用户可以通过 getBean("com.smart.simple.Car")获取 car Bean。如果存在多个实现类相同的匿名<bean>，如下：

```
<bean class="com.smart.simple.Car"/>
<bean class="com.smart.simple.Car"/>
<bean class="com.smart.simple.Car"/>
```

第一个 Bean 通过 getBean("com.smart.simple.Car")获得；第二个 Bean 通过 getBean("com.smart.simple.Car#1")获得；第三个 Bean 通过 getBean("com.smart. simple.Car#2")获得，以此类推。一般匿名<bean>在通过内部 Bean 为外层 Bean 提供注入值时使用，正如 Java 的匿名类一样。

提示

各种眼花缭乱、花拳绣腿式的命名方式着实让我们见识了 Spring 配置的灵活性和包容性，但在一般情况下，那些奇怪的命名大多是唬人的噱头，不值得在实际项目中使用，通过 id 为 Bean 指定唯一的名称才是“康庄大道”。

5.3　依赖注入

Spring 支持两种依赖注入方式，分别是属性注入和构造函数注入。除此之外，Spring 还支持工厂方法注入方式。在本节中，我们将了解到不同依赖注入方式的具体配置方法。

5.3.1　属性注入

属性注入指通过 setXxx()方法注入 Bean 的属性值或依赖对象。由于属性注入方式具有可选择性和灵活性高的优点，因此属性注入是实际应用中最常采用的注入方式。

1. 属性注入实例

属性注入要求 Bean 提供一个默认的构造函数，并为需要注入的属性提供对应的 Setter 方法。Spring 先调用 Bean 的默认构造函数实例化 Bean 对象，然后通过反射的方式调用 Setter 方法注入属性值。来看一个简单的例子，如代码清单 5-1 所示。

代码清单 5-1　Car：默认构造函数和Setter

```
package com.smart.ditype;
public class Car {
     private int maxSpeed;
     public String brand;
     private double price;
     public void setBrand(String brand) {
        this.brand = brand;
     }
     public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
     }
     public void setPrice(double price) {
        this.price = price;
     }
    ...
}
```

Car 类中定义了 3 个属性，并分别提供了对应的 Setter 方法。

提示

默认构造函数是不带参的构造函数。Java 语言规定，如果类中没有定义任何构造函数，则 JVM 会自动为其生成一个默认的构造函数；反之，如果类中显式定义了构造函数，则 JVM 不会为其生成默认的构造函数。所以假设 Car 类中显式定义了一个带参的构造函数，如 public Car(String brand)，则需要同时提供一个默认的构造函数 public Car()，否则使用属性注入时将抛出异常。

代码清单 5-2 是在 Spring 配置文件中对 Car 进行属性注入的配置片段。

代码清单 5-2　Car：属性注入配置

```
<bean id="car" class="com.smart.ditype.Car">
   <property name="maxSpeed"><value>200</value></property>
   <property name="brand"><value>红旗CA72</value></property>
   <property name="price"><value>20000.00</value></property>
</bean>
```

上述代码配置了一个 Bean，并为该 Bean 的 3 个属性提供了属性值。具体来说，Bean 的每一个属性对应一个<property>标签，name 为属性的名称，在 Bean 实现类中拥有与其对应的 Setter 方法：maxSpeed 对应 setMaxSpeed()，brand 对应 setBrand()。

需要指出的是，Spring 只会检查 Bean 中是否有对应的 Setter 方法，至于 Bean 中是否有对应的属性成员变更则不做要求。举个例子，配置文件中<property name="brand"/>的属性配置项仅要求 Car 类中拥有 setBrand()方法，但 Car 类不一定要拥有 brand 成员变量，如下：

```
public Class Car{
   private int maxSpeed;
   private double price;

  //①仅拥有 setBrand()方法，但类中没有 brand 成员变量
   public void setBrand(String brand){
      System.out.println("设置 brand 属性。");
   }
   ...
}
```

虽然如此，但在一般情况下，仍然按照约定俗成的方式在 Bean 中提供同名的属性变量。

2．JavaBean 关于属性命名的特殊规范

Spring 配置文件中<property>元素所指定的属性名和 Bean 实现类的 Setter 方法满足 Sun JavaBean 的属性命名规范：xxx 的属性对应 setXxx()方法。

一般情况下，Java 的属性变量名都以小写字母开头，如 maxSpeed、brand 等，但也存在特殊的情况。考虑到一些特定意义的大写英文缩略词（如 USA、XML 等），JavaBean 也允许以大写字母开头的属性变量名，不过必须满足**“变量的前两个字母要么全部大写，要么全部小写”**的要求，如 brand、IDCode、IC、ICCard 等属性变量名是合法的，而 iC、iCcard、iDCode 等属性变量名则是非法的。这个并不广为人知的 JavaBean 规范条款引发了众多让人困惑的配置问题。

为了更清楚地理解这个隐晦的问题，我们来看一个具体的实例。下面是一个“违反”了 JavaBean 属性命名规范的类：

```
public class Foo {

    //①非法的属性变量名，不过Java语言本身不会报错，因为它将iDCode看成普通的变量
        private String iDCode;

    //②该Setter方法对应IDCode属性而非iDCode属性
    public void setIDCode(String iDcode) {
        this.iDCode = iDcode;
    }
}
```

在 Spring 配置文件中，我们可能会想当然地为 Foo 提供以下配置：

```
<bean id="foo" class="com.smart.attr.Foo">
 <!--①这个属性变量名是非法的!! -->
    <property name="iDCode" value="070101"/>
 </bean>
```

当我们试图启动 Spring 容器时，将得到启动失败的结果，控制台输出以下错误信息：

```
Error setting property values; nested exception is org.springframework.
beans.NotWritablePropertyException: Invalid property 'iDCode' of bean class
[com.smart.attr.Foo]: Bean property 'iDCode' is not writable or has an invalid Setter
method. Did you mean 'IDCode'?
Caused by: org.springframework.beans.NotWritablePropertyException: Invalid property
'iDCode' of bean class
```

虽然 Spring 给出了启动失败的原因，但错误信息具有很强的误导性，因为它“报怨”Foo 中没有提供对应于 iDCode 的 Setter 方法，但事实上 Foo 已经提供了 setIDCode()方法。那真相到底是什么呢？其实真正的错误根源是我们在 Spring 配置文件中指定了一个非法的属性名 iDcode，这个非法的属性名永远不可能有对应的 Setter 方法，因此错误就产生了。

纠正的办法是将配置文件中的属性名改为 IDCode，如下：

```
<bean id="foo" class="com.smart.attr.Foo">
    <!--① IDCode对应setIDCode()属性设置方法-->
    <property name="IDCode" value="070101"/>
</bean>
```

Foo 类中的 iDCode 属性变量名不一定要修改，因为我们说过，Spring 配置文件的属性名仅对应于 Bean 实现类的 get/setXxx()方法。但是如果进一步探讨引发这个配置错误的根源，我们就会归咎于 Foo 类中 iDCode 的变量名。原因很简单，因为我们在编写了 Foo 的 iDCode 变量名后，通过 IDE 的代码自完成功能生成 setIDCode()属性设置方法，然后想当然地在 Spring 配置文件中使用 iDCode 属性名进行配置，最终造成了 Spring 容器的启动错误。

以大写字母开头的变量名总显得比较另类，为了避免这类诡异的错误，用户可以遵照以下的编程经验：像 QQ、MSN、ID 等正常情况下以大写字母出现的专业术语，在 Java 中一律将其调整为小写形式，如 qq、msn、id 等，以保证命名的统一性（变量名都以小写字母开头），减少出现错误的概率。

5.3.2 构造函数注入

构造函数注入是除属性注入外的另一种常用的注入方式，它保证一些必要的属性在 Bean 实例化时就得到设置，确保 Bean 在实例化后就可以使用。

1. 按类型匹配入参

如果任何可用的 Car 对象都必须提供 brand 和 price 的值，若使用属性注入方式，则只能人为地在配置时提供保证而无法在语法级提供保证，这时通过构造函数注入就可以很好地满足这一要求。使用构造函数注入的前提是 Bean 必须提供带参的构造函数。下面为 Car 提供一个可设置 brand 和 price 属性的构造函数。

```
package com.smart.ditype;
public class Car {
    ...
    public Car(String brand, double price) {
        this.brand = brand;
        this.price = price;
    }
}
```

构造函数注入的配置方式和属性注入的配置方式有所不同，下面在 Spring 配置文件中使用构造函数注入的配置方式装配这个 car Bean，如代码清单 5-3 所示。

代码清单 5-3　通过构造函数注入Car

```
<bean id="car1" class="com.smart.ditype.Car">
  <constructor-arg type="java.lang.String"> ①
    <value>红旗CA72</value>
  </constructor-arg>
  <constructor-arg type="double"> ②
    <value>20000</value>
  </constructor-arg>
</bean>
```

在<constructor-arg>的元素中有一个 type 属性，它为 Spring 提供了判断配置项和构造函数入参对应关系的“信息”。细心的读者可能会提出以下疑问：配置文件中<bean>元素的<constructor-arg>声明顺序难道不能用于确定构造函数入参的顺序吗？在只有一个构造函数的情况下当然是可以的，但如果在 Car 中定义了多个具有相同入参的构造函数，这种顺序标识方法就失效了。此外，Spring 的配置文件采用和元素标签顺序无关的策略，这种策略可以在一定程度上保证配置信息的确定性，避免一些似是而非的问题。因此，①和②处的<constructor-arg>位置并不会对最终的配置效果产生影响。

2．按索引匹配入参

我们知道 Java 语言通过入参的类型及顺序区分不同的重载方法。对于代码清单 5-3 中的 Car 类，Spring 仅通过 type 属性指定的参数类型就可以知道“红旗 CA72”对应 String 类型的 brand 入参，而“20000”对应 double 类型的 price 入参。但是如果 Car 构造函数有两个类型相同的入参，那么仅通过 type 就无法确定对应关系了，这时需要通过入参索引的方式进行确定。

提示

我们知道，在属性注入时，Spring 按 JavaBean 规范找到配置属性所对应的 Setter 方法，并使用 Java 反射机制调用 Setter 方法完成属性注入。但 Java 反射机制并不会记住构造函数的入参名，因此我们无法通过指定构造函数的入参名进行构造函数注入的配置，只能通过入参类型和索引信息间接确定构造函数配置项和入参的对应关系。

为了更好地演示按索引匹配入参的配置方式，我们特意对 Car 构造函数进行了以下调整：

```
//①该构造函数第一、第二入参都是 String 类型
public Car(String brand,String corp,double price){
    this.brand = brand;
    this.corp = corp;
    this.price = price;
}
```

因为 brand 和 corp 的入参类型都是 String，所以 Spring 无法确定 type 为 String 的<constructor-arg>到底对应的是 brand 还是 corp。但是通过显式指定参数的索引能够消除这种不确定性，如代码清单 5-4 所示。

代码清单 5-4　通过入参位置索引确定对应关系

```
<bean id="car2" class="com.smart.ditype.Car">
    <!--①注意索引从 0 开始-->
    <constructor-arg index="0" value="红旗CA72"/>
    <constructor-arg index="1" value="中国一汽"/>
    <constructor-arg index="2" value="20000"/>
</bean>
```

构造函数的第一个参数索引为 0，第二个为 1，以此类推，因此很容易知道“红旗 CA72”对应 brand 入参，而“中国一汽”对应 corp 入参。

3．联合使用类型和索引匹配入参

有时需要 type 和 index 联合使用才能确定配置项和构造函数入参的对应关系，来看下面的例子，如代码清单 5-5 所示。

代码清单 5-5　Car：入参数目相同的构造函数

```
...
public Car(String brand, String corp, double price) {
    this.brand = brand;
    this.corp = corp;
    this.price = price;
}
public Car(String brand, String corp, int maxSpeed) {
    this.brand = brand;
    this.corp = corp;
    this.maxSpeed = maxSpeed;
}
...
```

这里，Car 拥有两个重载的构造函数，它们都有两个入参。代码清单 5-4 按照入参位置索引的配置方式针对这种情况又难以满足要求了，这时需要联合使用<constructor-arg>的 type 和 index 才能解决问题，如代码清单 5-6 所示。

代码清单 5-6　Car：通过入参类型和位置索引确定对应关系

```
<!--①对应 Car (String brand,String corp, int maxSpeed)构造函数-->
<bean id="car3" class="com.smart.ditype.Car">
        <constructor-arg index="0" type="java.lang.String">
          <value>红旗CA72</value>
        </constructor-arg>
        <constructor-arg index="1" type="java.lang.String">
          <value>中国一汽</value>
        </constructor-arg>
        <constructor-arg index="2" type="int">
          <value>200</value>
        </constructor-arg>
</bean>
```

对于代码清单 5-5 中的两个构造函数，如果仅通过 index 进行配置，那么 Spring 将无法确定第三个入参配置项究竟是对应 int 的 maxSpeed 还是 double 的 price，所以在采用索引匹配配置时，真正引起歧义的地方是第三个入参，因此仅需要明确指定第三个入参的类型就可以取消歧义。所以在代码清单 5-6 中，第一、第二个<constructor-arg>元素

的 type 属性可以去除。

对于因参数数目相同而类型不同引起的潜在配置歧义问题，Spring 容器可以正确启动且不会给出报错信息，它将随机采用一个匹配的构造函数实例化 Bean，而被选择的构造函数可能并不是用户所期望的那个。因此，必须特别谨慎，以避免潜在的错误。

4．通过自身类型反射匹配入参

当然，如果 Bean 构造函数入参的类型是可辨别的（非基础数据类型且入参类型各异），由于 Java 反射机制可以获取构造函数入参的类型，即使构造函数注入的配置不提供类型和索引的信息，Spring 依旧可以正确地完成构造函数的注入工作。下面 Boss 类构造函数的入参就是可辨别的：

```
public Boss(String name, Car car, Office office) {
    this.name = name;
    this.car = car;
    this.office = office;
}
```

由于 car、office 和 name 入参的类型都是可辨别的，所以无须在构造函数注入的配置时指定<constructor-arg>的类型和索引，因此我们可以采用如下简易的配置方式：

```
<bean id="boss" class="com.smart.ditype.Boss">
   <!--①没有设置 type 和 index 属性，通过入参值的类型完成匹配映射-->
    <constructor-arg>
        <value>John</value>
    </constructor-arg>
    <constructor-arg>
      <ref bean="car"/>
    </constructor-arg>
    <constructor-arg>
      <ref bean="office"/>
    </constructor-arg>
</bean>
<bean id="car" class="com.smart.ditype.Car"/>
<bean id="office" class="com.smart.ditype.Office"/>
```

但是为了避免潜在配置歧义引起的张冠李戴的情况，如果 Bean 存在多个构造函数，那么使用显式指定 index 和 type 属性不失为一种良好的配置习惯。

5．循环依赖问题

Spring 容器能对构造函数配置的 Bean 进行实例化有一个前提，即 Bean 构造函数入参引用的对象必须已经准备就绪。由于这个机制的限制，如果两个 Bean 都采用构造函数注入，而且都通过构造函数入参引用对方，就会发生类似于线程死锁的循环依赖问题。来看一个发生循环依赖问题的例子：

```
public class Car {
  ...
  //①构造函数依赖于一个 boss 实例
  public Car(String brand,Boss boss){
      this.brand = brand;
      this.boss = boss;
  }
```

```
  ...
}

public class Boss {
  ...

  //②构造函数依赖于一个 car 实例
  public Boss(String name,Car car){
      this.name = name;
      this.car = car;
  }
  ...
}
```

假设在 Spring 配置文件中按照以下构造函数注入方式进行配置：

```
<bean id="car" class="com.smart.cons.Car">
  <constructor-arg index="0"  value="红旗CA72"/>
  <!--①引用②处的 boss -->
  <constructor-arg index="1" ref="boss"/>
</bean>
<bean id="boss" class="com.smart.cons.Boss">
  <constructor-arg index="0"  value="John"/>
  <!-- ②引用①处的 car -->
  <constructor-arg index="1" ref="car"/>
</bean>
```

当启动 Spring IoC 容器时，因为存在循环依赖问题，Spring 容器将无法成功启动。如何解决这个问题呢？用户只需修改 Bean 的代码，将构造函数注入方式调整为属性注入方式就可以了。

5.3.3 工厂方法注入

工厂方法是在应用中被经常使用的设计模式，它也是控制反转和单实例设计思想的主要实现方法。由于 Spring IoC 容器以框架的方式提供工厂方法的功能，并以透明的方式开放给开发者，所以很少需要手工编写基于工厂方法的类。正是因为工厂方法已经成为底层设施的一部分，因此工厂方法对于实际编码的重要性就降低了。不过在一些遗留系统或第三方类库中，我们还会遇到工厂方法，这时可以使用 Spring 工厂方法注入的方式进行配置。

1. 非静态工厂方法

有些工厂方法是非静态的，即必须实例化工厂类后才能调用工厂方法。下面为 Car 提供一个非静态的工厂类，如代码清单 5-7 所示。

代码清单 5-7　CarFactory：非静态工厂方法

```
package com.smart.ditype;
public class CarFactory {
  //①创建 Car 的工厂方法
  public Car createHongQiCar(){
```

```
        Car car = new Car();
        car.setBrand("红旗CA72");
        return car;
    }
}
```

工厂类负责创建一个或多个目标类实例，工厂类方法一般以接口或抽象类变量的形式返回目标类实例。工厂类对外屏蔽了目标类的实例化步骤，调用者甚至无须知道具体的目标类是什么。在代码清单 5-7 中，CarFactory 工厂类仅负责创建 Car 类型的对象，下面的配置片段使用 CarFactory 为 Car 提供工厂方法的注入，如代码清单 5-8 所示。

代码清单 5-8　通过工厂类注入Bean

```
...
<!--①工厂类Bean -->
<bean id="carFactory" class="com.smart.ditype.CarFactory"/>

<!-- factory-bean 指定①处的工厂类Bean; factory-method 指定工厂类
Bean 创建该Bean 的工厂方法-->
<bean id="car5"  factory-bean="carFactory"
                 factory-method="createHongQiCar"/>
```

由于 CarFactory 工厂类的工厂方法不是静态的，所以首先需要定义一个工厂类的 Bean，然后通过 factory-bean 引用工厂类实例，最后通过 factory-method 指定对应的工厂类方法。

2. 静态工厂方法

很多工厂类方法都是静态的，这意味着用户在无须创建工厂类实例的情况下就可以调用工厂类方法，因此，静态工厂方法比非静态工厂方法更易使用。下面对 CarFactory 进行改造，将其 createHongQiCar()方法调整为静态的，如代码清单 5-9 所示。

代码清单 5-9　CarFactory：静态工厂方法

```
package com.smart.ditype;
public class CarFactory {
  //①工厂类方法是静态的
  public static Car createHongQiCar(){
      ...
  }
}
```

当使用静态工厂类型的方法后，用户就无须在配置文件中定义工厂类的 Bean，只需按以下方式进行配置即可：

```
                                                    工厂类方法
                                                  ┌──────────────────┐
<bean id="car6" class="com.smart.ditype.CarFactory" factory-method="createCar" />
                └──────────────────────────────┘
                              工厂类
```

直接在<bean>中通过 class 属性指定工厂类，然后再通过 factory-method 指定对应的工厂方法。

5.3.4 选择注入方式的考量

Spring 提供了 3 种可供选择的注入方式，在实际应用中，究竟应该选择哪种注入方式呢？对于这个问题，仁者见仁，智者见智，并没有统一的标准。下面是支持使用构造函数注入的理由：

- 构造函数可以保证一些重要的属性在 Bean 实例化时就设置好，避免因为一些重要属性没有提供而导致一个无用 Bean 实例的情况。
- 不需要为每个属性提供 Setter 方法，减少了类的方法个数。
- 可以更好地封装类变量，不需要为每个属性指定 Setter 方法，避免外部错误的调用。

更多的开发者可能倾向于使用属性注入方式，他们反对构造函数注入的理由如下：

- 如果一个类的属性众多，那么构造函数的签名将变成一个庞然大物，可读性很差。
- 灵活性不强，在有些属性是可选的情况下，如果通过构造函数注入，也需要为可选的参数提供一个 null 值。
- 如果有多个构造函数，则需要考虑配置文件和具体构造函数匹配歧义的问题，配置上相对复杂。
- 构造函数不利于类的继承和扩展，因为子类需要引用父类复杂的构造函数。
- 构造函数注入有时会造成循环依赖的问题。

其实构造函数注入和属性注入各有自己的应用场景，Spring 并没有强制用户使用哪一种方式，用户完全可以根据个人偏好做出选择，在某些情况下使用构造函数注入，而在另一些情况下使用属性注入。对于一个全新开发的应用来说，我们不推荐使用工厂方法的注入方式，因为工厂方法需要额外的类和代码，这些功能和业务是没有关系的，既然 Spring 容器已经以一种更优雅的方式实现了传统工厂模式的所有功能，那么我们大可不必再去做这项重复性的工作。

5.4 注入参数详解

在 Spring 配置文件中，用户不但可以将 String、int 等字面值注入 Bean 中，还可以将集合、Map 等类型的数据注入 Bean 中，此外还可以注入配置文件中其他定义的 Bean。

5.4.1 字面值

所谓“字面值”一般是指可用字符串表示的值，这些值可以通过<value>元素标签进行注入。在默认情况下，基本数据类型及其封装类、String 等类型都可以采取字面值注入的方式。Spring 容器在内部为字面值提供了编辑器，它可以将以字符串表示的字面值

转换为内部变量的相应类型。Spring 允许用户注册自定义的编辑器，以处理其他类型属性注入时的转换工作（关于自定义编辑器的内容，请参见第 6 章）。

在下面的示例中，我们为 Car 注入了两个属性值，并在 Spring 配置文件中使用字面值提供配置值，如代码清单 5-10 所示。

代码清单 5-10　字面值注入字面值

```
<bean id="car" class="com.smart.attr.Car">
    <property name="maxSpeed">
        <value>200</value>
    </property>
    <property name="brand">①
        <value><![CDATA[红旗&CA72]]></value>
    </property>
</bean>
```

由于①处的 brand 属性值包含一个 XML 的特殊符号，因此我们特意在属性值外添加了一个 XML 特殊标签<![CDATA[ ]]>。<![CDATA[ ]]>的作用是让 XML 解析器将标签中的字符串当作普通的文本对待，以防止特殊字符串对 XML 格式造成破坏。

XML 中共有 5 个特殊的字符，分别是&、<、>、“、’。如果配置文件中的注入值包括这些特殊字符，就需要进行特别处理。有两种解决方法：其一，采用本例中的<![CDATA[ ]]>特殊标签，将包含特殊字符的字符串封装起来；其二，使用 XML 转义序列表示这些特殊字符，这 5 个特殊字符所对应的 XML 转义序列在表 5-2 中进行了说明。

表 5-2　XML特殊实体符号

特殊符号	转义序列	特殊符号	转义序列
<	<	“	"
>	>	‘	'
&	&		

如果使用 XML 转义序列，则可以使用以下配置替换代码清单 5-10 中的配置。

```
<property name="brand"><value>红旗&amp;CA72</value></property>
```

提示

一般情况下，XML 解析器会忽略元素标签内部字符串的前后空格，但 Spring 却不会忽略元素标签内部字符串的前后空格。如通过以下配置为 brand 属性提供注入值：<property name="brand"><value>␣红旗 CT72 ␣</value> </property>，那么 Spring 会将“红旗 CT72”连同其前后空格一起赋给 brand 属性。

5.4.2　引用其他 Bean

Spring IoC 容器中定义的 Bean 可以相互引用，IoC 容器则充当“红娘”的角色。下面创建一个新的 Boss 类，Boss 类中拥有一个 Car 类型的属性。

```
package com.smart.attr;
public class Boss {
    private Car car;
   //①设置 car 属性
    public void setCar(Car car) {
        this.car = car;
    }
    ...
}
```

boss 的 Bean 通过<ref>元素引用 car Bean，建立起 boss 对 car 的依赖。

```
<!--①car Bean -->
 <bean id="car" class="com.smart.attr.Car"/>
 <bean id="boss" class="com.smart.attr.Boss">
     <property name="car">
       <!--②引用①处定义的 car Bean -->
       <ref bean="car"></ref>
     </property>
</bean>
```

<ref>元素可以通过以下 3 个属性引用容器中的其他 Bean。

- bean：通过该属性可以引用同一容器或父容器中的 Bean，这是最常见的形式。
- local：通过该属性只能引用同一配置文件中定义的 Bean，它可以利用 XML 解析器自动检验引用的合法性，以便开发人员在编写配置时能够及时发现并纠正配置错误。
- parent：引用父容器中的 Bean，如<ref parent="car">的配置说明 car 的 Bean 是父容器中的 Bean。

为了说明子容器对父容器中 Bean 的引用，我们来看一个具体的例子。假设有两个配置文件 beans1.xml 和 beans2.xml，其中 beans1.xml 被父容器加载，其配置内容如下：

```
<!--①在父容器中定义的 car -->
<bean id="car" class="com.smart.attr.Car">
      <property name="brand" value="红旗CA72" />
      <property name="maxSpeed" value="200" />
      <property name="price" value="2000.00" />
</bean>
```

而 beans2.xml 被子容器加载，其配置内容如下：

```
<!--①该 Bean 和父容器的 car Bean 具有相同的 id -->
 <bean id="car" class="com.smart.attr.Car">
       <property name="brand" value="吉利CT5" />
       <property name="maxSpeed" value="100" />
       <property name="price" value="1000.00" />
</bean>
<bean id="boss" class="com.smart.attr.Boss">
       <property name="car">
        <!--②引用父容器中的 car，而非②处定义的 Bean。如
        果采用<ref bean="car"/>，则将引用本容器①处的 car -->
          <ref parent="car"/>
       </property>
</bean>
```

在 beans1.xml 中配置了一个 car Bean，在 bean2.xml 中也配置了一个 car Bean。分

别通过父、子容器加载 beans1.xml 和 beans2.xml，beans2.xml 中的 boss 通过<ref parent="car">引用父容器中的 car。

下面是分别使用父、子容器加载 beans1.xml 和 beans2.xml 配置文件的代码：

```
//①父容器
ClassPathXmlApplicationContext pFactory = new ClassPathXmlApplicationContext(
                new String[]{"com/smart/attr/beans1.xml"});
//②指定 pFactory 为该容器的父容器
ApplicationContext factory = new ClassPathXmlApplicationContext(
            new String[]{"com/smart/attr/beans2.xml"},pFactory);
Boss boss = (Boss)factory.getBean("boss");
System.out.println(boss.getCar().toString());
```

运行这段代码，在控制台中打印出以下信息：

```
brand:红旗CA72/maxSpeed:200/price:2000.0
```

5.4.3　内部 Bean

如果 car Bean 只被 boss Bean 引用，而不被容器中任何其他的 Bean 引用，则可以将 car 以内部 Bean 的方式注入 Boss 中。

```
<bean id="boss" class="com.smart.attr.Boss">
   <property name="car">
      <bean class="com.smart.attr.Car">
         <property name="maxSpeed" value="200"/>
         <property name="price" value="2000.00"/>
      </bean>
   </property>
</bean>
```

内部 Bean 和 Java 的匿名内部类相似，既没有名字，也不能被其他 Bean 引用，只能在声明处为外部 Bean 提供实例注入。

内部 Bean 即使提供了 id、name、scope 属性，也会被忽略，scope 默认为 prototype 类型。关于 Bean 的作用域，将在 5.8 节进行详细介绍。

5.4.4　null 值

如果用户尝试通过以下配置方式为 car 的 brand 属性注入一个 null 值，那么将会得到一个失望的结果。

```
<bean id="car" class="com.smart.attr.Car">
      <property name="brand"><value></value></property>
 </bean>
```

Spring 会将<value></value>解析为空字符串。那么，如何为属性设置一个 null 的注入值呢？答案是必须使用专用的<null/>元素标签，通过它可以为 Bean 的字符串或其他对象类型的属性注入 null 值。

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

上面的配置代码等同于调用 car.setBrand(null)方法。

5.4.5 级联属性

和 Struts、Hibernate 等框架一样，Spring 支持级联属性的配置。假设我们希望在定义 Boss 时直接为 Car 的属性提供注入值，则可以采取以下配置方式：

```
<bean id="boss3" class="com.smart.attr.Boss">
  <!--①以圆点(.)的方式定义级别属性-->
   <property name="car.brand" value="吉利 CT50"/>
</bean>
```

按照上面的配置，Spring 将调用 Boss.getCar().setBrand("吉利 CT50")方法进行属性的注入操作。这时必须对 Boss 类进行改造，为 car 属性声明一个初始化对象。

```
public class Boss {
   //①声明初始化对象
    private Car car = new Car();
    public Car getCar() {
        return car;
    }
    public void setCar(Car car) {
        this.car = car;
    }
}
```

在①处为 Boss 的 car 属性提供了一个非空的 Car 实例。如果没有为 car 属性提供 Car 对象，那么 Spring 在设置级联属性时将抛出 NullValueInNestedPathException 异常。

Spring 没有对级联属性的层级数进行限制，只要配置的 Bean 拥有对应于级联属性的类结构，就可以配置任意层级的级联属性，如<property name="car.wheel.brand" value="双星"/>定义了具有三级结构的级联属性。

5.4.6 集合类型属性

java.util 包中的集合类型是最常用的数据结构类型，主要包括 List、Set、Map、Properties，Spring 为这些集合类型属性提供了专属的配置标签。

1. List

为 Boss 添加一个 List 类型的 favorites 属性，如下：

```
package com.smart.attr;
...
public class Boss {
    private List favorites = new ArrayList();
    public List getFavorites() {
        return favorites;
    }
    public void setFavorites(List favorites) {
        this.favorites = favorites;
    }
    ...
}
```

对应 Spring 中的配置片段如下：

```
<bean id="boss1" class="com.smart.attr.Boss">
    <property name="favorites">
      <list>
        <value>看报</value>
        <value>赛车</value>
        <value>高尔夫</value>
      </list>
    </property>
</bean>
```

List 属性既可以通过<value>注入字符串，也可以通过<ref>注入容器中其他的 Bean。

提示

假设一个属性类型可以通过字符串字面值进行配置，那么该类型对应的数组类型的属性（如 String[]、int[]等）也可以采用<list>方式进行配置。

2. Set

如果 Boss 的 favorites 属性是 java.util.Set，则采用如下配置方式：

```
<bean id="boss1" class="com.smart.attr.Boss">
    <property name="favorites">
      <set>
        <value>看报</value>
        <value>赛车</value>
        <value>高尔夫</value>
      </set>
    </property>
</bean>
```

3. Map

下面为 Boss 添加一个 Map 类型的 jobs 属性：

```
public class Boss {
    ...
    private Map jobs = new HashMap();
    public Map getJobs() {
        return jobs;
    }
    public void setJobs(Map jobs) {
        this.jobs = jobs;
    }
    ...
}
```

在配置文件中可以通过以下方式为 jobs 属性提供配置值：

```
<bean id="boss1" class="com.smart.attr.Boss">
     <property name="jobs">
          <map>
            <entry>                                          ①
               <key><value>AM</value></key>
               <value>会见客户</value>
            </entry>
```

Map 第一个元素

```
            <entry>                                          ②
                <key><value>PM</value></key>
                <value>公司内部会议</value>
            </entry>
        </map>
    </property>
</bean>
```

Map 第二个元素

假如某一 Map 元素的键和值都是对象，则可以采取以下配置方式：

```
<entry>
    <key><ref bean="keyBean"/></key>
    <ref bean="valueBean"/>
</entry>
```

4. Properties

Properties 类型其实可以看作 Map 类型的特例。Map 元素的键和值可以是任何类型的对象，而 Properties 属性的键和值都只能是字符串。下面为 Boss 添加一个 Properties 类型的 mails 属性：

```
public class Boss {
    ...
    private Properties mails = new Properties();
    public Properties getMails() {
        return mails;
    }
    public void setMails(Properties mails) {
        this.mails = mails;
    }
    ...
}
```

下面的配置片段为 mails 提供了配置：

```
<bean id="boss1" class="com.smart.attr.Boss">
    <property name="mails">
        <props>
            <prop key="jobMail">john-office@smart.com</prop>
            <prop key="lifeMail">john-life@smart.com</prop>
        </props>
    </property>
</bean>
```

因为 Properties 键值对只能是字符串，因此其配置比 Map 的配置要简单一些，注意值的配置没有<value>子元素标签。

5. 强类型集合

Java 5.0 提供了强类型集合的新功能，允许为集合元素指定类型。如下面 Boss 类中的 jobTime 属性就采用了强类型的 Map 类型，元素的键为 String 类型，而值为 Integer 类型。

```
public class Boss {
    ...
    private Map<String,Integer> jobTime = new HashMap<String,Integer>();
    public Map<String, Integer> getJobTime() {
        return jobTime;
```

```
    }
    public void setJobTime(Map<String, Integer> jobTime) {
        this.jobTime = jobTime;
    }
    ...
}
```

在 Spring 中的配置和非强类型集合相同，如代码清单 5-11 所示。

代码清单 5-11　强类型集合配置

```
<bean id="boss1" class="com.smart.attr.Boss">
    <property name="jobTime">
          <map>
            <entry>
                <key><value>会见客户</value></key>
                <value>124</value> <!--①为 Integer 类型提供设置值-->
            </entry>
          </map>
        </property>
    </bean>
```

但 Spring 容器在注入强类型集合时会判断元素的类型，将设置值转换为对应的数据类型。如代码清单 5-11 中①处的设置项 124 将被转换为 Integer 类型。

6．集合合并

Spring 支持集合合并的功能，允许子<bean>继承父<bean>的同名属性集合元素，并将子<bean>中配置的集合属性值和父<bean>中配置的同名属性值合并起来作为最终 Bean 的属性值，如代码清单 5-12 所示。关于父子<bean>的内容，请参见 5.6.1 节。

代码清单 5-12　集合合并

```
<bean id="parentBoss" abstract="true"
        class="com.smart.attr.Boss"> <!--①父<bean> -->
        <property name="favorites">
            <set>
                <value>看报</value>
                <value>赛车</value>
                <value>高尔夫</value>
            </set>
        </property>
    </bean>
    <bean id="childBoss" parent="parentBoss"> <!--②指定父<bean>-->
        <property name="favorites">
            <set merge="true"> <!--③和父<bean>中的同名集合属性合并-->
                <value>爬山</value>
                <value>游泳</value>
            </set>
        </property>
</bean>
```

在代码清单 5-12 中，③处通过 merge="true"属性指示子<bean>和父<bean>中的同名属性值进行合并，即子 Bean 的 favorites 集合最终将拥有 5 个元素。如果设置为 merge="false"，则不会和父<bean>中的同名集合属性进行合并，即子 Bean 的 favorites 属性集合只有两个元素。

7. 通过 util 命名空间配置集合类型的 Bean

如果希望配置一个集合类型的 Bean，而非一个集合类型的属性，则可以通过 util 命名空间进行配置。首先需要在 Spring 配置文件头中引入 util 命名空间的声明。

```
<?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:util="http://www.springframework.org/schema/util" ①
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-
4.0.xsd
                        http://www.springframework.org/schema/util
                        http://www.springframework.org/schema/util/spring-util-
4.0.xsd">
    ...
</beans>
```

util 命名空间

其次配置一个 List 类型的 Bean，可以通过 list-class 显式指定 List 的实现类。

```
<util:list id="favoriteList1" list-class="java.util.LinkedList">
    <value>看报</value>
    <value>赛车</value>
    <value>高尔夫</value>
</util:list>
```

再次配置一个 Set 类型的 Bean，可以通过 set-class 指定 Set 的实现类。

```
<util:set id="favoriteSet1">
    <value>看报</value>
    <value>赛车</value>
    <value>高尔夫</value>
</util:set>
```

最后配置一个 Map 类型的 Bean，可以通过 map-class 指定 Set 的实现类。

```
<util:map id="emails1">
    <entry key="AM" value="会见客户" />
    <entry key="PM" value="公司内部会议" />
</util:map>
```

此外，<util:list>和<util:set>支持 value-type 属性，指定集合中的值类型；而<util:map>支持 key-type 和 value-type 属性，指定 Map 的键和值类型。

5.4.7 简化配置方式

前面几节我们采用完整配置格式的配置方式，也许读者已经发现这种方式显得比较拖沓。Spring 为字面值、引用 Bean 和集合都提供了简化的配置方式。如果没有用到完整配置格式的特殊功能，用户大可使用简化的配置方式。下面分别为上面提及的配置内容给出简化前和简化后的版本。

1．字面值属性（见表 5-3）

表 5-3　字面值属性简化配置

	简化前	简化后
字面值属性	<property name="maxSpeed"> <value>200</value> </property>	<property name="maxSpeed" value="200"/>
构造函数参数	<constructor-arg type="java.lang.String"> <value>红旗 CA72</value> </constructor-arg>	<constructor-arg type="java.lang.String" value="红旗 CA72" />
集合元素	<map> <entry> <key><value>AM</value></key> <value>会见客户</value> </entry> </map>	<map> <entry key="AM" value="会见客户"/> </map>

如果使用简化的方式，则将无法使用<![CDATA[ ]]>处理 XML 特殊字符，只能用 XML 转义序列对特殊字符进行转换，如 value="红旗&CA72"。

2．引用对象属性（见表 5-4）

表 5-4　引用对象属性简化配置

	简化前	简化后
字面值属性	<property name="car"> <ref bean="car"></ref> </property>	<property name="car" ref="car"/>
构造函数参数	<constructor-arg> <ref bean="car"/> </constructor-arg>	<constructor-arg ref="car"/>
集合元素	<map> <entry> <key><ref bean="keyBean"/></key> <ref bean="valueBean"/> </entry> </map>	<map> <entry key-ref="keyBean" value-ref="valueBean"/> </map>

<ref>的简化形式对应于<ref bean="xxx">，而<ref local="xxx">和<ref parent="xxx">则没有对应的简化形式。

3．使用 p 命名空间

为了简化 XML 文件的配置，越来越多的 XML 文件采用属性而非子元素配置信息。Spring 从 2.5 版本开始引入了一个新的 p 命名空间，可以通过<bean>元素属性的方式配置 Bean 的属性。使用 p 命名空间后，基于 XML 的配置方式将进一步简化。

使用 p 命名空间前，如代码清单 5-13 所示。

代码清单 5-13　未采用p命名空间的配置

```
<?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-4.0.xsd">
    <bean id="car" class="com.smart.ditype.Car">
        <property name="brand" value="红旗&amp;amp;CA72"/>
        <property name="maxSpeed" value="200"/>
        <property name="price" value="20000.00"/>
    </bean>
    <bean id="boss" class="com.smart.ditype.Boss">
        <property name="car" ref="car"/>
    </bean>
</beans>
```

使用 p 命名空间后，如代码清单 5-14 所示。

代码清单 5-14　采用p命名空间的配置

```
<?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:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
    <bean id="car" class="com.smart.ditype.Car"
        p:brand="红旗&amp;amp;CA72"
        p:maxSpeed="200"
        p:price="20000.00"/>
    <bean id="boss" class="com.smart.ditype.Boss"
        p:car-ref="car"/>
</beans>
```

①声明 p 命名空间

②采用 p 命名空间配置 Bean 属性值(字面值)

③采用 p 命名空间配置 Bean 属性值(引用对象)

未采用 p 命名空间前，<bean>使用<property>子元素配置 Bean 的属性；采用 p 命名空间后，采用<bean>的元素属性配置 Bean 的属性。

对于字面值属性，其格式为：

```
p:<属性名>="xxx"
```

对于引用对象的属性，其格式为：

```
p:<属性名>-ref="xxx"
```

正是由于 p 命名空间中的属性名是可变的，所以 p 命名空间没有对应的 Schema 定义文件，也就无须在 xsi:schemaLocation 中为 p 命名空间指定 Schema 定义文件。

实战经验

在 IDEA 开发工具中，对 Spring 配置文件默认提供诱导功能，对于 p 命名空间的属性配置，只要输入 p:就能动态分析 Bean 类的属性列表，开发者只要诱导选择即可。对于其他的开发工具，如 Eclipse，需要安装 Spring IDE for Eclipse 插件，并按 Alt+/组合键才能诱导。

5.4.8　自动装配

Spring IoC 容器知道所有 Bean 的配置信息，此外，通过 Java 反射机制还可以获知实现类的结构信息，如构造函数方法的结构、属性等信息。掌握所有 Bean 的这些信息后，Spring IoC 容器就可以按照某种规则对容器中的 Bean 进行自动装配，而无须通过显式的方式进行依赖配置。Spring 为厌恶配置的开发人员提供了一种轻松的方法，可以按照某些规则进行 Bean 的自动装配。

<bean>元素提供了一个指定自动装配类型的属性：autowire="<自动装配类型>"。Spring 提供了 4 种自动装配类型，用户可以根据具体情况进行选择，如表 5-5 所示。

表 5-5　自动装配类型

自动装配类型	说　明
byName	根据名称进行自动匹配。假设 Boss 有一个名为 car 的属性，如果容器中刚好有一个名为 car 的 Bean，Spring 就会自动将其装配给 Boss 的 car 属性
byType	根据类型进行自动匹配。假设 Boss 有一个 Car 类型的属性，如果容器中刚好有一个 Car 类型的 Bean，Spring 就会自动将其装配给 Boss 的这个属性
constructor	与 ByType 类似，只不过它是针对构造函数注入而言的。如果 Boss 有一个构造函数，构造函数包含一个 Car 类型的入参，如果容器中有一个 Car 类型的 Bean，则 Spring 将自动把这个 Bean 作为 Boss 构造函数的入参；如果容器中没有找到和构造函数入参匹配类型的 Bean，则 Spring 将抛出异常
autodetect	根据 Bean 的自省机制决定采用 byType 还是 constructor 进行自动装配。如果 Bean 提供了默认的构造函数，则采用 byType；否则采用 constructor

<beans>元素标签中的 default-autowire 属性可以配置全局自动匹配，default-autowire 属性的默认值为 no，表示不启用自动装配；其他几个可选配置值分别为 byName、byType、constructor 和 autodetect，这几个配置值的含义是不言自明的。不过在<beans>中定义的自动装配策略可以被<bean>的自动装配策略覆盖。

自动装配以四两拨千斤的方式完成容器中 Bean 之间的装配工作，这种省心省力的自动装配机制确实省却了大量配置工作。在实际开发中，XML 配置方式很少启用自动装配功能，而基于注解的配置方式默认采用 byType 自动装配策略。

轻松一刻

董永和七仙女所演绎的天仙配已经成为家喻户晓的故事。到过湖北省孝感市的朋友可能会觉得这个市名比较奇怪，其实孝感的市名就源自天仙配的故事。时下，很多婚恋网站都提供了名为“天仙配”的速配功能，能够根据用户设定的条件从众多候选者中选出一个匹配的对象。婚恋网站就像 Spring 容器，男女注册用户就是一个个 Bean，而身高、体重、居住地等各种择偶要求就是 byName、byType 等自动装配的约束条件，标榜缘分的“神奇功能”其实只是一个小小的算法。

5.5 方法注入

无状态 Bean 的作用域一般可以配置为 singleton（单例模式），如果我们往 singleton 的 Boss 中注入 prototype 的 Car，并希望每次调用 boss Bean 的 getCar()方法时都能够返回一个新的 car Bean，使用传统的注入方式将无法实现这样的要求。因为 singleton 的 Bean 注入关联 Bean 的动作仅有一次，虽然 car Bean 的作用范围是 prototype 类型，但 Boss 通过 getCar()方法返回的对象还是最开始注入的那个 car Bean。

如果希望每次调用 getCar()方法都返回一个新的 car Bean 的实例，一种可选的方法就是让 Boss 实现 BeanFactoryAware 接口，且能够访问容器的引用，这样 Boss 的 getCar()方法就可以采取以下实现方式来达到目的：

```
public Car getCar() {
    //通过 getBean()返回 prototype 的 Bean，每次都将返回新实例
     return (Car)factory.getBean("car");
}
```

但在第 4 章中指出，这种依赖 Spring 框架接口的设计将应用与 Spring 框架绑定在一起，部分开发者可能并不喜欢。针对前面提出的需求，是否有既不与 Spring 框架绑定，又可享受依赖注入好处的实现方案？Spring 没有让我们失望，可以通过方法注入的方案完美地解决这个问题。

5.5.1 lookup 方法注入

Spring IoC 容器拥有复写 Bean 方法的能力，这项魔术般的功能归功于 CGLib 类包。CGLib 可以在运行期动态操作 Class 字节码，为 Bean 动态创建子类或实现类。关于 CGLib 的进一步介绍，请参见第 7 章。

现在声明一个 MagicBoss 接口，并声明一个 getCar()的接口方法。

```
package com.smart.injectfun;
public interface MagicBoss {
   Car getCar();
}
```

下面不编写任何实现类，仅通过配置为该接口提供动态的实现，让 getCar()接口方法每次都返回新的 car Bean。

```
<!--① prototype 类型的 Bean -->
<bean id="car" class="com.smart.injectfun.Car"
     p: brand="红旗CA72" p: price="2000" scope="prototype"/>
<!--②实施方法注入-->
<bean id="magicBoss" class="com.smart.injectfun.MagicBoss">
   <lookup-method name="getCar" bean="car"/>
</bean>
```

通过 lookup-method 元素标签为 MagicBoss 的 getCar()提供动态实现，返回 prototype 类型的 car Bean，这样 Spring 将在运行期为 MagicBoss 接口提供动态实现，其效果等同于：

```
package com.smart.injectfun;
…
public class MagicBossImpl implements MagicBoss, ApplicationContextAware {
    private ApplicationContext ctx;
    public Car getCar() {
        return (Car) ctx.getBean("car");
    }
    public void setApplicationContext(ApplicationContext ctx)
            throws BeansException {
        this.ctx = ctx;
    }
}
```

因为每次调用 MagicBoss 的 getCar()方法都会从容器中获取 car Bean，由于 car Bean 的作用域为 prototype，所以每次都返回新的 car 实例。

如果将 car Bean 的作用域设置为 singleton，虽然以上配置仍然可以运行，但这时 lookup 所提供的方法注入就没有什么意义了。因为我们可以容易地编写一个 MagicBoss 接口实现类，用属性注入的方式达到相同的目的。所以 lookup 方法注入是有一定使用范围的，一般在希望通过一个 singleton Bean 获取一个 prototype Bean 时使用。

提示

由于方法注入时 Spring 需要用到 CGLib 类包，所以需要将 CGLib 类包加入到类路径中，否则无法使用方法注入的功能。

5.5.2　方法替换

在金庸笔下，“乾坤大挪移”是明教至高无上的神功，在《倚天屠龙记》里会九阳神功的张无忌最终修成了正果。在 Spring IoC 容器里，用户同样可以拥有这种“乾坤大挪移”的能力：可以使用某个 Bean 的方法去替换另一个 Bean 的方法。

在下面的例子中，Boss1 的 getCar()方法返回一辆宝马 Z4。

```
package com.smart.injectfun;
public class Boss1{
    public Car getCar() {
        Car car = new Car();
        car.setBrand("宝马Z4");
        return car;
    }
}
```

Boss2 实现了 Spring 的 org.springframework.beans.factory.support.MethodReplacer 接口，在接口方法 reimplement()中，返回一辆美人豹。

```
package com.smart.injectfun;
import java.lang.reflect.Method;
import org.springframework.beans.factory.support.MethodReplacer;
public class Boss2 implements MethodReplacer {
    public Object reimplement(Object arg0, Method arg1, Object[] arg2)
            throws Throwable {
```

```
        Car car = new Car();
        car.setBrand("美人豹");
        return car;
    }
}
```

用于替换他人的 Bean 必须实现 MethodReplacer 接口，Spring 将利用该接口的方法去替换目标 Bean 的方法。下面通过 Spring IoC 容器的“乾坤大挪移”术，用 Boss2 的方法去替换 Boss1 的 getCar()方法。

```
<bean id="boss1" class="com.smart.injectfun.Boss1">
   <replaced-method name="getCar" replacer="boss2"/>①
</bean>
<bean id="boss2" class="com.smart.injectfun.Boss2"/>
```

使用 boss2 的 MethodReplacer 接口方法替换该 Bean 的 getCar()方法

当从容器中返回 boss1 Bean 并调用其 getCar()方法时，将返回一辆“美人豹”的 Car，调包成功。

但这种高级功能就像《宋玉答楚王问》中所说的阳春白雪一样，在实际应用中很少使用，而属性注入、构造函数注入等“下里巴人”式的普通功能反而在实际项目中使用最多。

5.6 <bean>之间的关系

不但可以通过<ref>引用另一个 Bean，建立起 Bean 和 Bean 之间的依赖关系，<bean>元素标签之间也可以建立类似的关系，完成一些特殊的功能。

5.6.1 继承

OOP 思想告诉我们，如果多个类拥有相同的方法和属性，则可以引入一个父类，在父类中定义这些类共同的方法和属性，以消除重复的代码。同样，如果多个<bean>存在相同的配置信息，则 Spring 允许定义一个父<bean>，子<bean>将自动继承父<bean>的配置信息。

下面通过一个实例，对使用和未使用父子<bean>的配置进行比较，从中看出父子<bean>给配置带来的便利性，如代码清单 5-15 所示。

代码清单 5-15　未使用父子<bean>的配置

```
<!--①和②处的配置信息完全相同-->
<bean id="car1" class="com.smart.tagdepend.Car"
     p: brand="红旗CA72" p: price="2000.00"  p:color="黑色"/>
<bean id="car2" class="com.smart.tagdepend.Car"
     p: brand="红旗CA72" p: price="2000.00"  p:color="红色"/> <!--②-->
```

代码清单 5-15 中配置了两个 car Bean，我们发现这两个 Bean 的配置存在大量的重复信息。事实上，二者除了 color 属性配置值不一样外，其他配置信息都相同。通过父

子<bean>的继承关系就可以很好地消除这种重复的配置信息，如代码清单 5-16 所示。

代码清单 5-16　使用父子<bean>的配置

```
<!--①定义为抽象 bean-->
<bean id="abstractCar" class="com.smart.tagdepend.Car"
    p: brand="红旗CA72" p: price="2000.00"  p:color="黑色"  abstract= "true"/>
<!--②继承于 abstractCar -->
<bean id="car3" p:color="红色" parent="abstractCar"/>
<!--③继承于 abstractCar -->
<bean id="car4" p:color="白色" parent="abstractCar"/>
```

在代码清单 5-16 中，car3 和 car4 这两个<bean>都继承于 abstractCar 的<bean>，Spring 会将父<bean>的配置信息传递给子<bean>。如果子<bean>提供了父<bean>已有的配置信息，那么子<bean>的配置信息将覆盖父<bean>的配置信息。

父<bean>的主要功能是简化子<bean>的配置，所以一般声明为 abstract="true"，表示这个<bean>不实例化为一个对应的 Bean。在代码清单 5-16 的①处，如果用户没有指定 abstract="true"，则 Spring IoC 容器会实例化一个名为 abstractCar 的 Bean。

5.6.2　依赖

一般情况下，可以使用<ref>元素标签建立对其他 Bean 的依赖关系，Spring 负责管理这些 Bean 的关系。当实例化一个 Bean 时，Spring 保证该 Bean 所依赖的其他 Bean 已经初始化。

但在某些情况下，这种 Bean 之间的依赖关系并不那么明显。下面举一个例子。“小春论坛”拥有很多系统参数（如会话过期时间、缓存更新时间等），这些系统参数用于控制系统的运行逻辑。我们用一个 SystemSettings 类表示这些系统参数。

```
public class SystemSettings {
  public static int SESSION_TIMEOUT = 30;
  public static int.REFRESH_CYCLE = 60;
  ...
}
```

在 SystemSettings 类中为每个系统参数提供了默认值，但一个灵活的论坛必须提供一个管理后台，在管理后台中可以调整这些系统参数并保存到后台数据库中，在系统启动时，初始化程序从数据库后台加载这些系统参数的配置值以覆盖默认值。

```
public class SysInit {
  public SysInit(){
      SystemSettings.SESSION_TIMEOUT = 10;  ①
      SystemSettings.REFRESH_CYCLE = 100;
  }
}
```

模拟从数据库中加载系统参数设置值

假设论坛有一个缓存刷新管理器，它需要根据系统参数 SystemSettings. REFRESH_CYCLE 创建缓存刷新定时任务。

```
public class CacheManager {
  public CacheManager(){
```

```
        Timer timer = new Timer();
        TimerTask cacheTask = new CacheTask();
        timer.schedule(cacheTask,0,SystemSettings.REFRESH_CYCLE);
    }
}
```

在以上实例中，CacheManager 依赖于 SystemSettings，而 SystemSettings 的值由 SysInit 负责初始化。虽然 CacheManager 不直接依赖于 SysInit，但从逻辑上看，CacheManager 希望在 SysInit 加载并完成系统参数设置后再启动，以避免调用不到真实的系统参数值。如果这 3 个 Bean 都在 Spring 配置文件中定义，那么如何保证 SysInit 在 CacheManager 之前进行初始化呢？

Spring 允许用户通过 depends-on 属性显式指定 Bean 前置依赖的 Bean，前置依赖的 Bean 会在本 Bean 实例化之前创建好。

```
<bean id="manager" class="com.smart.tagdepend.CacheManager"
                  depends-on="sysInit" />①
<bean id="sysInit" class="com.smart.tagdepend.SysInit" /> ②
```

该 Bean 依赖了②处的 Bean

在①处通过 depends-on 属性将 sysInit 指定为 manager 前置依赖的 Bean，这样就可以保证 manager Bean 在实例化并运行时所引用的系统参数是最新的设置值，而非 SystemSettings 类中的默认值。如果前置依赖于多个 Bean，则可以通过逗号、空格或分号的方式创建 Bean 的名称。

5.6.3 引用

假设一个<bean>要引用另一个<bean> 的 id 属性值，则可以直接使用以下配置方式：

```
<bean id="car" class="com.smart.tagdepend.Car"/> ①
<bean id="boss" class="com.smart.tagdepend.Boss"
      p: carId="car" scope="prototype"/> ②
```

假设希望将 boss Bean 的 carId 设置为①处<bean>的 id 值，虽然可以通过②处的方式以字面值的形式进行设置，但二者之间并没有建立引用关系。一般情况下，在一个 Bean 中引用另一个 Bean 的 id 是希望在运行期通过 getBean(beanName)方法获取对应的 Bean。由于 Spring 并不会在容器启动时对属性配置值进行特殊检查，因此，即使编写错误，也需要等到具体调用时才会发现。

Spring 为此提供了一个<idref>元素标签，可以通过<idref>引用另一个<bean>的名字。在容器启动时，Spring 负责检查引用关系的正确性，这样就可以提前发现错误。因此，下面的配置是推荐的优化方案：

```
<bean id="car" class="com.smart.tagdepend.Car"/> ①
<bean id="boss" class="com.smart.tagdepend.Boss">
  <property name="carId" >
     <idref bean="car"/> ②
  </property>
</bean>
```

假设②处由于配置错误，误将<idref bean="car"/>写为<idref bean="cat"/>，那么

Spring 容器在启动时，将会抛出 BeanDefinitionStoreException，提示容器中没有名为 cat 的 Bean。

如果引用者和被引用者的<bean>位于同一个 XML 配置文件中，则可以使用<idref local="car">的配置方式，这时 IDE 的 XML 分析器就可以在开发期发现引用错误了。

5.7　整合多个配置文件

对于一个大型应用来说，可能存在多个 XML 配置文件，在启动 Spring 容器时，可以通过一个 String 数组指定这些配置文件。Spring 还允许通过<import>将多个配置文件引入到一个文件中，进行配置文件的集成。这样，在启动 Spring 容器时，仅需指定这个合并好的配置文件即可。代码清单 5-17 是 beans2.xml 的配置文件，它引入了 beans1.xml 配置文件。

代码清单 5-17　将多个配置文件组合到一起：beans2.xml

```
<import resource="classpath:com/smart/impt/beans1.xml"/>①
<bean id="boss1" class="com.smart.fb.Boss" p:name="John" p:car-ref="car1"/>
<bean id="boss2" class="com.smart.fb.Boss" p:name="John" p:car-ref="car2"/>
```

假设已经在 beans1.xml 中配置了 car1 和 car2 的 Bean，在①处通过<import>的 resource 属性引入 beans1.xml，beans2.xml 就拥有了完整的配置信息，Spring 容器仅需通过 beans2.xml 就可以加载所有的配置信息。

需要指出的是，如果一个配置文件 a.xml 定义的<bean>引用了另一个配置文件 b.xml 定义的<bean>，那么并不一定需要通过<import>引入 b.xml，只需在启动 Spring 容器时，a.xml 和 b.xml 都在配置文件列表中即可。区别在于，如果 a.xml 采用 import 引入了 b.xml，相当于 a.xml 一个文件就包含了 a.xml 和 b.xml 两个文件的内容，因此 Spring 容器启动时仅需加载 a.xml 即可；否则就需要在启动 Spring 容器时，同时加载 a.xml 和 b.xml 配置文件，以便在内存中对 a.xml 和 b.xml 进行合并。

一个 XML 配置文件可以通过<import>组合多个外部的配置文件，resource 属性支持 Spring 标准的资源路径，参见 4.3 节的说明。

实战经验

对于大型应用来说，为了防止开发时配置文件的资源竞争，或者为了使模块便于拆卸，往往每个模块都拥有自己独立的配置文件。应用层面提供了一个整合的配置文件，通过<import>将各个模块整合起来。这样，在容器启动时，只需加载这个整合的配置文件即可。

5.8 Bean 作用域

在配置文件中定义 Bean 时，用户不但可以配置 Bean 的属性值及相互之间的依赖关系，还可以定义 Bean 的作用域。作用域将对 Bean 的生命周期和创建方式产生影响。表 5-6 列出了 Spring 4.0 支持的所有作用域类型。

表 5-6 Bean作用域类型

类 型	说 明
singleton	在 Spring IoC 容器中仅存在一个 Bean 实例，Bean 以单实例的方式存在
prototype	每次从容器中调用 Bean 时，都返回一个新的实例，即每次调用 getBean()时，相当于执行 new XxxBean()操作
request	每次 HTTP 请求都会创建一个新的 Bean。该作用域仅适用于 WebApplicationContext 环境
session	同一个 HTTP Session 共享一个 Bean，不同的 HTTP Session 使用不同的 Bean。该作用域仅适用于 WebApplicationContext 环境
globalSession	同一个全局 Session 共享一个 Bean，一般用于 Portlet 应用环境。该作用域仅适用于 WebApplicationContext 环境

在低版本的 Spring 中，仅支持两个 Bean 作用域，所以采用 singleton="true|false"的配置方式。Spring 为了向后兼容，依然支持这种配置方式。不过，Spring 推荐采用新的配置方式：scope="<作用域类型>"。

除了以上 5 种预定义的 Bean 作用域外，Spring 还允许用户自定义 Bean 的作用域。可以先通过 org.springframework.beans.factory.config.Scope 接口定义新的作用域，再通过 org.springframework.beans.factory.config.CustomScopeConfigurer 这个 BeanFactoryPostProcessor 注册自定义的 Bean 作用域。在一般的应用中，Spring 所提供的作用域已经能够满足应用的要求，用户很少需要自定义新的 Bean 作用域。所以本书不对此进行深入讲解，感兴趣的读者可以自行阅读 Scope 接口的 Javadoc 文档。

5.8.1 singleton 作用域

单例模式是重要的设计模式之一。在传统的应用开发中，需要手工为每个单实例类编写特定代码，在这种情况下，类的业务逻辑代码和模式代码紧密耦合在一起。Spring 以容器的方式提供天然的单例模式功能，任何 POJO 无须编写特殊的代码，仅通过配置就可以享用单例模式的“大餐”。

一般情况下，无状态或者状态不可变的类适合使用单例模式，不过 Spring 对此实现了超越。在传统开发中，由于 DAO 类持有 Connection 这个非线程安全的变量，因此往往未采用单例模式。而在 Spring 环境下，对于所有的 DAO 类都可以采用单例模式，因为 Spring 利用 AOP 和 LocalThread 功能，对非线程安全的变量（或称状态）进行了特殊处理，使这些非线程安全的类变成了线程安全的类（将在第 7 章介绍这一功能的内部机理）。

因为 Spring 的这一超越，所以在实际应用中，大部分 Bean 都能以单实例的方式运行，这也是为什么 Spring 将 Bean 的默认作用域定为 singleton 的原因。

singleton 的 Bean 在同一 Spring IoC 容器中只有一个实例，请看下面的例子：

```
<bean id="car" class="com.smart.scope.Car" scope="singleton"/> ①
<bean id="boss1" class="com.smart.scope.Boss" p: car-ref="car"/>②
<bean id="boss2" class="com.smart.scope.Boss" p: car-ref="car"/>③
<bean id="boss3" class="com.smart.scope.Boss" p: car-ref="car"/>④
```

①处的 car Bean 声明为 singleton（因为默认是 singleton，所以无须显式指定），在容器中有 3 个其他的 Bean 引用了 car Bean，如②、③、④所示。在容器内部，boss1、boss2 和 boss3 的 car 属性都指向同一个 Bean，如图 5-3 所示。

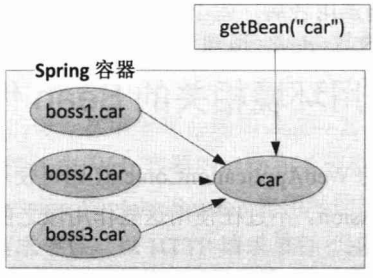


图 5-3　单例模式

不但在配置文件中通过配置注入的 car 引用相同的 car Bean，任何通过容器的 getBean("car")方法返回的实例也指向同一个 Bean。

在默认情况下，Spring 的 ApplicationContext 容器在启动时，自动实例化所有 singleton 的 Bean 并缓存于容器中。虽然启动时会花费一些时间，但它带来两个好处：首先，对 Bean 提前进行实例化操作会及早发现一些潜在的配置问题；其次，Bean 以缓存的方式保存，当运行时用到该 Bean 时就无须再实例化了，提高了运行的效率。如果用户不希望在容器启动时提前实例化 singleton 的 Bean，则可以通过 lazy-init 属性进行控制。

```
<bean id="boss1" class="com.smart.scope.Boss"  p:car-ref="car"  lazy-init="true"/>
```

lazy-init="true"的 Bean 在某些情况下依然会提前实例化：如果该 Bean 被其他需要提前实例化的 Bean 所引用，那么 Spring 将忽略延迟实例化的设置。

5.8.2　prototype 作用域

采用 scope="prototype"指定非单例作用域的 Bean，请看下面的配置：

```
<bean id="car" class="com.smart.scope.Car" scope="prototype"/> ①
<bean id="boss1" class="com.smart.scope.Boss" p:car-ref="car"/> ②
<bean id="boss2" class="com.smart.scope.Boss" p:car-ref="car"/> ③
<bean id="boss3" class="com.smart.scope.Boss" p:car-ref="car"/> ④
```

通过以上配置，boss1、boss2、boss3 所引用的都是一个新的 car 实例，每次通过容器的 getBean("car")方法返回的也是一个新的 car 实例，如图 5-4 所示。

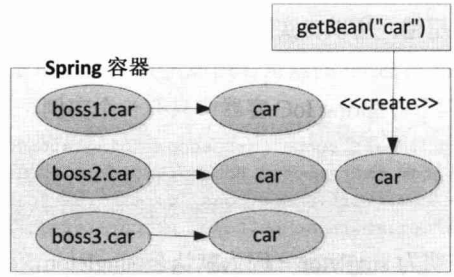


图 5-4　非单例模式

在默认情况下，Spring 容器在启动时不实例化 prototype 的 Bean。此外，Spring 容器将 prototype 的 Bean 交给调用者后，就不再管理它的生命周期。

5.8.3　与 Web 应用环境相关的 Bean 作用域

如果用户使用 Spring 的 WebApplicationContext，则可使用另外 3 种 Bean 的作用域：request、session 和 globalSession。不过在使用这些作用域之前，首先必须在 Web 容器中进行一些额外的配置。

1．在 Web 容器中进行额外配置

在低版本的 Web 容器中（Servlet 2.3 之前），用户可以使用 HTTP 请求过滤器进行配置。

```
<web-app>
  ...
  <filter>
      <filter-name>requestContextFilter</filter-name>
      <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
  </filter>
      <filter-mapping>
      <filter-name>requestContextFilter</filter-name>
      <!--①对所有的 URL 进行过滤拦截-->
      <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

在高版本的 Web 容器中，则可以利用 HTTP 请求监听器进行配置。

```
<web-app>
  ...
  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>
  ...
</web-app>
```

细心的读者可能会有一个疑问：在第 4 章介绍 WebApplicationContext 初始化时，已经通过 ContextLoaderListener（或 ContextLoaderServlet）将 Web 容器与 Spring 容器进行了整合，为什么在这里又要引入一个额外的 RequestContextListener 以支持 Bean 的另外 3 个作用域呢？通过分析两个监听器的源码，一切疑问就真相大白了，如图 5-5 所示。

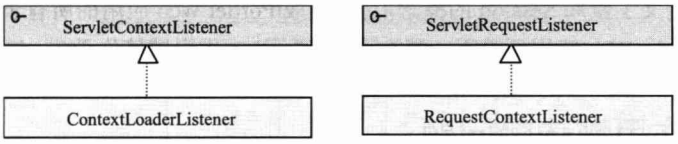


图 5-5　两个监听器的区别

在整合 Spring 容器时使用 ContextLoaderListener，它实现了 ServletContextListener 监听器接口，ServletContextListener 只负责监听 Web 容器启动和关闭的事件。而 RequestContextListener 实现了 ServletRequestListener 监听器接口，该监听器监听 HTTP 请求事件，Web 服务器接收的每一次请求都会通知该监听器。

Spring 容器启动和关闭操作由 Web 容器的启动和关闭事件触发，但如果 Spring 容器中的 Bean 需要 request、session 和 globalSession 作用域的支持，Spring 容器本身就必须获得 Web 容器的 HTTP 请求事件，以 HTTP 请求事件“驱动”Bean 作用域的控制逻辑。也就是说，通过配置 RequestContextListener，Spring 容器和 Web 容器的结合更加密切，Spring 容器对 Web 容器中的“风吹草动”都能够察觉，因而就可以实施 Web 相应 Bean 作用域的控制了。

当然，Spring 完全可以提供一个既实现 ServletContextListener 又实现 ServletContextListener 接口的监听器，这样我们仅需配置一次就可以了。探究 Spring 将二者分开的原因，可能出于两个方面的考虑：第一，考虑版本兼容的问题，毕竟针对 Web 应用的 Bean 作用域是从 Spring 2.0 开始提供的；第二，这 3 种新增的 Bean 作用域的适用场合并不多，用户往往并不真的需要这些新增的 Bean 作用域。

2．request 作用域

顾名思义，request 作用域的 Bean 对应一个 HTTP 请求和生命周期。考虑下面的配置：

```
<bean name="car" class="com.smart.scope.Car" scope="request"/>
```

这样，每次 HTTP 请求调用 car Bean 时，Spring 容器就会创建一个新的 car Bean；请求处理完毕后，就会销毁这个 Bean。

3．session 作用域

假设将以上 Car 的作用域调整为 session 类型，如下：

```
<bean name="car" class="com.smart.scope.Car" scope="session"/>
```

这样配置后，car Bean 的作用域横跨整个 HTTP Session，Session 中的所有 HTTP 请求都共享同一个 car Bean。当 HTTP Session 结束后，实例才被销毁。

4. globalSession 作用域

下面的配置片段将 car 的作用域设置为 globalSession：

```
<bean name="loginController" class="com.smart.scope.Car" scope= "globalSession"/>
```

globalSession 作用域类似于 session 作用域，不过仅在 Portlet 的 Web 应用中使用。Portlet 规范定义了全局 Session 的概念，它被组成 Portlet Web 应用的所有子 Portlet 共享。如果不在 Portlet Web 应用环境下，那么 globalSession 作用域等价于 session 作用域。

5.8.4 作用域依赖问题

假设将 Web 相关作用域的 Bean 注入 singleton 或 prototype 的 Bean 中，我们当然希望它能够按照预定的方式工作，即引用者应该从指定的域中取得它的引用。但如果没有进行一些额外的配置，那么我们将得到一个失望的结果。在这种情况下，需要 Spring AOP "出手相救"，如代码清单 5-18 所示。

代码清单 5-18　非Web相关作用域引用Web相关作用域的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"          ①声明 aop Schema
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/
schema/aop/spring-aop-4.0.xsd">
<bean name="car" class="com.smart.scope.Car" scope="request">
    <aop:scoped-proxy/> ②          创建代理
</bean>
<bean id="boss" class="com.smart.scope.Boss">
    <property name="car" ref="car"/> ③          引用 Web 相关作用域的 car Bean
</bean>
</beans>
```

在代码清单 5-18 中，car Bean 是 request 作用域，它被 singleton 作用域的 boss Bean 引用。为了使 boss 能够从适当作用域中获取 car Bean 的引用，需要使用 Spring AOP 的语法为 car Bean 配置一个代理类，如②所示。为了能够在配置文件中使用 AOP 的配置标签，需要在文档声明头中定义 aop 命名空间。

当 boss Bean 在 Web 环境下调用 car Bean 时，Spring AOP 将启用动态代理智能地判断 boss Bean 位于哪个 HTTP 请求线程中，并从对应的 HTTP 请求线程域中获取对应的 car Bean。我们通过图 5-6 对此进行剖析。

boss Bean 的作用域是 singleton，也就是说，在 Spring 容器中始终只有一个实例，而 car Bean 的作用域为 request，所以每个调用到 car Bean 的 HTTP 请求都会创建一个 car Bean。Spring 通过动态代理技术，能够让 boss Bean 引用到对应 HTTP 请求的 car Bean。

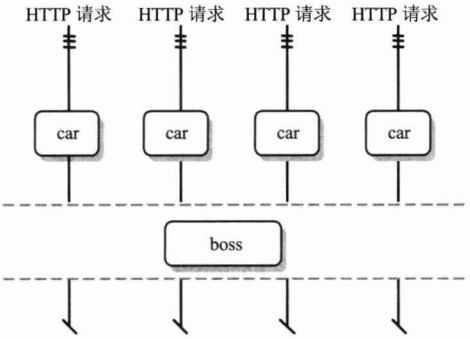


图 5-6　引用 Web 相关作用域的 Bean

反过来，在配置文件中添加<aop:scoped-proxy/>后，注入 boss Bean 中的 car Bean 已经不是原来的 car Bean，而是 car Bean 的动态代理对象。这个动态代理是 Car 类的子类（或实现类，假设 Car 是接口），Spring 在动态代理子类中加入一段逻辑，以判断当前的 boss 需要取得哪个 HTTP 请求相关的 car Bean。

提示

动态代理所添加的逻辑其实也很简单，即判断当前 boss 位于哪个线程中，然后根据这个线程找到对应的 HttpRequest，再从 HttpRequest 域中获取对应的 car。因为 Web 容器的特性，一般情况下，一个 HTTP 请求对应一个独立的线程。

Java 语言只能对接口提供自动代理，所以，如果需要对类提供代理，则需要在类路径中加入 CGLib 的类库，这时 Spring 将使用 CGLib 为类生成动态代理的子类。我们将在第 7 章和第 8 章讨论 Spring AOP 的相关知识。

5.9　FactoryBean

一般情况下，Spring 通过反射机制利用<bean>的 class 属性指定实现类实例化 Bean。在某些情况下，实例化 Bean 的过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会获得一个简单的方案。Spring 为此提供了一个 org.springframework.beans.factory.FactoryBean 工厂类接口，用户可以通过实现该工厂类接口定制实例化 Bean 的逻辑。

FactoryBean 接口对于 Spring 框架来说占有重要的地位，Spring 自身就提供了 70 多个 FactoryBean 的实现类。它们隐藏了实例化一些复杂 Bean 的细节，给上层应用带来了便利，本书后续章节会多次用到 Spring 自身提供的 FactoryBean 实现类。

从 Spring 3.0 开始，FactoryBean 开始支持泛型，即接口声明改为 FactoryBean<T>的形式。在该接口中共定义了 3 个接口方法。

- T getObject()：返回由 FactoryBean 创建的 Bean 实例。如果 isSingleton()返回 true，则该实例会放到 Spring 容器的单实例缓存池中。
- boolean isSingleton()：确定由 FactoryBean 创建的 Bean 的作用域是 singleton 还是 prototype。
- Class<?> getObjectType()：返回 FactoryBean 创建 Bean 的类型。

当配置文件中<bean>的 class 属性配置的实现类是 FactoryBean 时，通过 getBean()方法返回的不是 FactoryBean 本身，而是 FactoryBean#getObject()方法所返回的对象，相当于 FactoryBean#getObject()代理了 getBean()方法。

在前面的例子中，在配置 Car 时，Car 的每个属性分别对应一个<property>元素标签。假设我们认为这种方式不够简洁，而希望通过逗号分隔的方式一次性为 Car 的所有属性指定配置值，那么可以通过编写一个 FactoryBean 来达到目的，如代码清单 5-19 所示。

代码清单 5-19　自定义CarFactoryBean

```
package com.smart.fb;
import org.springframework.beans.factory.FactoryBean;
public class CarFactoryBean implements FactoryBean<Car> {

    private String carInfo;
    public String getCarInfo() {
        return carInfo;
    }
    //①接收逗号分隔的属性设置信息
    public void setCarInfo(String carInfo) {
        this.carInfo = carInfo;
    }

    //②实例化 car Bean
    public Car getObject() throws Exception {
        Car car = new Car();
        String[] infos = carInfo.split(",");
        car.setBrand(infos[0]);
        car.setMaxSpeed(Integer.parseInt(infos[1]));
        car.setPrice(Double.parseDouble(infos[2]));
        return car;
    }

    //③返回 Car 的类型
    public Class<Car> getObjectType() {
        return Car.class;
    }

    //④标识通过该 FactoryBean 返回的 Bean 是 singleton
    public boolean isSingleton() {
        return false;
    }
}
```

有了这个 CarFactoryBean 后，就可以在配置文件中使用以下自定义的配置方式配置 Car Bean：

```
<bean id="car1" class="com.smart.fb.CarFactoryBean"
        p:carInfo="红旗CA72,200,20000.00"/>
```

当调用 getBean("car")时，Spring 通过反射机制发现 CarFactoryBean 实现了 FactoryBean 的接口，这时 Spring 容器就调用接口方法 CarFactoryBean#getObject()返回工厂类创建的对象。如果用户希望获取 CarFactoryBean 的实例，则需要在使用 getBean(beanName)方法时显式地在 beanName 前加上“&”前缀，即 getBean("&car")。

5.10　基于注解的配置

5.10.1　使用注解定义 Bean

前面说过，不管是 XML 还是注解，它们都是表达 Bean 定义的载体，其实质都是为 Spring 容器提供 Bean 定义的信息，在表现形式上都是将 XML 定义的内容通过类注解进行描述。Spring 从 2.0 开始就引入了基于注解的配置方式，在 2.5 时得到了完善，在 4.0 时进一步增强。

我们知道，Spring 容器成功启动的三大要件分别是 Bean 定义信息、Bean 实现类及 Spring 本身。如果采用基于 XML 的配置，则 Bean 定义信息和 Bean 实现类本身是分离的；而如果采用基于注解的配置文件，则 Bean 定义信息通过在 Bean 实现类上标注注解实现。

下面是使用注解定义一个 DAO 的 Bean：

```
package com.smart.anno;
import org.springframework.stereotype.Component;
//①通过 Repository 定义一个 DAO 的 Bean
@Component ("userDao")
public class UserDao {
  ...
}
```

在①处使用@Component 注解在 UserDao 类声明处对类进行标注，它可以被 Spring 容器识别，Spring 容器自动将 POJO 转换为容器管理的 Bean。

它和以下 XML 配置是等效的：

```
<bean id="userDao" class="com.smart.anno.UserDao"/>
```

除@Component 外，Spring 还提供了 3 个功能基本和@Component 等效的注解，分别用于对 DAO、Service 及 Web 层的 Controller 进行注解。

- @Repository：用于对 DAO 实现类进行标注。
- @Service：用于对 Service 实现类进行标注。
- @Controller：用于对 Controller 实现类进行标注。

之所以要在@Component 之外提供这 3 个特殊的注解，是为了让标注类本身的用途

清晰化，完全可以用@Component 替代这 3 个特殊的注解。但是，我们推荐使用特定的注解标注特定的 Bean，毕竟这样一眼就可以看出 Bean 的真实身份。

5.10.2 扫描注解定义的 Bean

Spring 提供了一个 context 命名空间，它提供了通过扫描类包以应用注解定义 Bean 的方式，如代码清单 5-20 所示。

代码清单 5-20 定义扫描包

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--①声明 context 命名空间-->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.0.xsd">
  <!--②扫描类包以应用注解定义的 Bean-->
   <context:component-scan base-package="com.smart.anno"/>
</beans>
a
```

在①处声明 context 命名空间，在②处即可通过 context 命名空间的 component-scan 的 base-package 属性指定一个需要扫描的基类包，Spring 容器将会扫描这个基类包里的所有类，并从类的注解信息中获取 Bean 的定义信息。

如果仅希望扫描特定的类而非基包下的所有类，那么可以使用 resource-pattern 属性过滤出特定的类，如下：

```
<context:component-scan base-package="com.smart" resource-pattern="anno/*.class"/>
```

这里将基类包设置为 com.smart；默认情况下 resource-pattern 属性的值为“**/*.class”，即基类包里的所有类，将其设置为“anno/*.class”，则 Spring 仅会扫描基类包里 anno 子包中的类。

通过 resource-pattern 属性可以按资源名称对基类包中的类进行过滤。如果只使用 resource-pattern，就会发现很多时候它并不能满足要求，如仅需过滤基类包中实现了 XxxService 接口的类或标注了某个特定注解的类等。

不过这些需求可以很容易地通过<context:component-scan>的过滤子元素实现，如下：

```
<context:component-scan base-package="com.smart">
   <context:include-filter type="regex" expression="com\.smart\.anno.*"/>
   <context:exclude-filter type="aspectj" expression="com.smart..*Controller+"/>
</context:component-scan>
```

<context:include-filter>表示要包含的目标类，而<context:exclude-filter>表示要排除的目标类。一个<context:component-scan>下可以拥有若干个<context:exclude-filter>和

<context: include-filter>元素。这两个过滤元素均支持多种类型的过滤表达式，说明如表 5-7 所示。

表 5-7　过滤表达式

类　别	示　例	说　明
annotation	com.smart.XxxAnnotation	所有标注了 XxxAnnotation 的类。该类型采用目标类是否标注了某个注解进行过滤
assignable	com.smart.XxxService	所有继承或扩展 XxxService 的类。该类型采用目标类是否继承或扩展了某个特定类进行过滤
aspectj	com.smart..* Service+	所有类名以 Service 结束的类及继承或扩展它们的类（参见第 7 章关于 AspectJ 的内容）。该类型采用 AspectJ 表达式进行过滤
regex	com\.smart\.anno\..*	所有 com.smart.anno 类包下的类。该类型采用正则表达式根据目标类的类名进行过滤
custom	com.smart.XxxTypeFilter	采用 XxxTypeFile 代码方式实现过滤规则。该类必须实现 org.springframework.core.type.TypeFilter 接口

在所有这些过滤类型中，除 custom 类型外，aspectj 的过滤表达能力是最强的，它可以轻易实现其他类型所能表达的过滤规则。

<context:component-scan/>拥有一个容易被忽视的 use-default-filters 属性，其默认值为 true，表示默认会对标注@Component、@Controller、@Service 及@Reposity 的 Bean 进行扫描。<context:component-scan/>先根据<exclude-filter/>列出需要排除的黑名单，再通过<include-filter/>列出需要包含的白名单。由于 use-default-filters 属性默认值的作用，下面的配置片段不但会扫描@Controller 的 Bean，还会扫描@Component、@Service 及@Reposity 的 Bean。

```
<context:component-scan base-package="com.smart">
   <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

换言之，在以上配置中，加不加<context:include-filter/>的效果都是一样的。如果想仅扫描@Controller 的 Bean，则必须将 use-default-filters 属性设置为 false。

```
<context:component-scan base-package="com.smart" use-default-filters="false">
   <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

5.10.3　自动装配 Bean

1. 使用@Autowired 进行自动注入

Spring 通过@Autowired 注解实现 Bean 的依赖注入。来看一个 LogonService 的例子，如代码清单 5-21 所示。

代码清单 5-21　@Autowired注入

```
package com.smart.anno;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
//① 定义一个 Service 的 Bean
  @Service
  public class LogonService {
    //② 分别注入 LogDao 及 UserDao 的 Bean
     @Autowired
     private LogDao logDao;
     @Autowired
     private UserDao userDao;
     ...
  }
```

在①处使用@Service 将 LogonService 标注为一个 Bean，在②处通过@Autowired 注入 LogDao 及 UserDao 的 Bean。@Autowired 默认按类型（byType）匹配的方式在容器中查找匹配的 Bean，当有且仅有一个匹配的 Bean 时，Spring 将其注入@Autowired 标注的变量中。

2．使用@Autowired 的 required 属性

如果容器中没有一个和标注变量类型匹配的 Bean，那么 Spring 容器启动时将报 NoSuchBeanDefinitionException 异常。如果希望 Spring 即使找不到匹配的 Bean 完成注入也不要抛出异常，那么可以使用@Autowired(required=false)进行标注，如代码清单 5-22 所示。

代码清单 5-22　设置Autowired的required属性

```
...
@Service
public class LogonService {
    @Autowired(required=false)
    private LogDao logDao;
    ...
}
```

在默认情况下，@Autowired 的 required 属性值为 ture，即要求必须找到匹配的 Bean，否则将报异常。

3．使用@Qualifier 指定注入 Bean 的名称

如果容器中有一个以上匹配的 Bean 时，则可以通过@Qualifier 注解限定 Bean 的名称，如代码清单 5-23 所示。

代码清单 5-23　@Qualifier的使用

```
package com.smart.anno;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

@Service
public class LogonService {
```

```
    @Autowired
    private LogDao logDao;

    //①注入名为 userDao、类型为 UserDao 的 Bean
    @Autowired
    @Qualifier("userDao")
    private UserDao userDao;
}
```

这时，假设容器有两个类型为 UserDao 的 Bean，一个名为 userDao，另一个名为 otherUserDao，则①处会注入名为 userDao 的 Bean。

4．对类方法进行标注

@Autowired 可以对类成员变量及方法的入参进行标注，下面在类的方法上使用@Autowired 注解，如代码清单 5-24 所示。

代码清单 5-24　在类的方法上使用@Autowired

```
package com.smart.anno;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

@Service
public class LogonService {
    private LogDao logDao;
    private UserDao userDao;

    //①自动将 LogDao 传给方法入参
    @Autowired
    public void setLogDao(LogDao logDao) {
        this.logDao = logDao;
    }

    //②自动将名为 userDao 的 Bean 传给方法入参
    @Autowired
    @Qualifier("userDao")
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

如果一个方法拥有多个入参，则在默认情况下，Spring 将自动选择匹配入参类型的 Bean 进行注入。Spring 允许对方法入参标注@Qualifier 以指定注入 Bean 的名称，如下：

```
@Autowired
public void init(@Qualifier("userDao")UserDao userDao,LogDao logDao){
    System.out.println("multi param inject");
    this.userDao = userDao;
    this.logDao =logDao;
}
```

在以上例子中，UserDao 的入参注入名为 userDao 的 Bean，而 LogDao 的入参注入 LogDao 类型的 Bean。

一般情况下，在 Spring 容器中大部分 Bean 都是单实例的，所以一般无须通过

@Repository、@Service 等注解的 value 属性为 Bean 指定名称，也无须使用@Qualifier 注解按名称进行注入。

虽然 Spring 支持在属性和方法上标注自动注入注解@Autowired，但在实际项目开发中建议采用在方法上标注@Autowired 注解，因为这样更加“面向对象”，也方便单元测试的编写。如果将注解标注在私有属性上，则在单元测试时就很难用编程的办法设置属性值。

5．对集合类进行标注

如果对类中集合类的变量或方法入参进行@Autowired 标注，那么 Spring 会将容器中类型匹配的所有 Bean 都自动注入进来。下面来看一个具体的例子，如代码清单 5-25 所示。

代码清单 5-25　MyComponent

```
package com.smart.anno;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
@Component
public class MyComponent {

    //①Spring 会将容器中所有类型为 Plugin 的 Bean 注入这个变量中
    @Autowired(required=false)
    private List<Plugin> plugins;

    //②将Plugin类型的Bean注入Map中
    @Autowired
    private Map<String,Plugin> pluginMaps;

    public List<Plugin> getPlugins() {
        return plugins;
    }
}
```

Spring 如果发现变量是一个 List 和一个 Map 集合类，则它会将容器中匹配集合元素类型的所有 Bean 都注入进来。在②处将实现 Plugin 接口的 Bean 注入 Map 集合，是 Spring 4.0 提供的新特性，其中 key 是 Bean 的名字，value 是所有实现了 Plugin 的 Bean。

这里，Plugin 是一个接口，它拥有两个实现类，分别是 OnePlugin 和 TwoPlugin，其中 OnePlugin 如代码清单 5-26 所示。

代码清单 5-26　OnePlugin插件示例

```
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(value = 1)  //①指定此插件的加载顺序，值越小，优先被加载
public class OnePlugin implements Plugin{

}
```

通过@Component 标注为 Bean，Spring 会将 OnePlugin 和 TwoPlugin 这两个 Bean 都注入 plugins 中。在默认情况下，这两个 Bean 的加载顺序是不确定的，在 Spring4.0 中可以通过@Order 注解或实现 Ordered 接口来决定 Bean 加载的顺序，值越小，优先被加载。

6．对延迟依赖注入的支持

Spring 4.0 支持延迟依赖注入，即在 Spring 容器启动的时候，对于在 Bean 上标注@Lazy 及@Autowired 注解的属性，不会立即注入属性值，而是延迟到调用此属性的时候才会注入属性值，如代码清单 5-27 所示。

代码清单 5-27　延迟依赖示例

```
…
@Lazy //①此处需要标注延迟注解
@Repository
public class LogDao{
}

@Service
public class LogonService implements BeanNameAware {

   @Lazy//②此处需要标注延迟注解
   @Autowired(required=false)
    public void setLogDao(LogDao logDao){...}
}
```

对 Bean 实施延迟依赖注入，要注意@Lazy 注解必须同时标注在属性及目标 Bean 上，如示例的①和②处，二者缺一，则延迟注入无效。

实战经验

Spring 对集合类自动注入容器中所有匹配类型 Bean 的功能非常强大，笔者深深受惠于这项功能。笔者曾在某公司负责研发一个类似于普元的快速开发平台，该开发平台采用“模型驱动+插件”的体系架构，其中的插件体系就完全利用 Spring 集合注入的功能完成插件的识别和注入工作，大大简化了平台的研发难度。

7．对标准注解的支持

此外，Spring 还支持 JSR-250 中定义的@Resource 和 JSR-330 中定义的@Inject 注解，这两个标准注解和@Autowired 注解的功用类似，都是对类变更及方法入参提供自动注入功能。@Resource 注解要求提供一个 Bean 名称的属性，如果属性为空，则自动采用标注处的变量名或方法名作为 Bean 的名称，如代码清单 5-28 所示。

代码清单 5-28　@Resource示例

```
package com.smart.anno;
import javax.annotation.Resource;
import org.springframework.stereotype.Component;
@Component
```

```
public class Boss {
    private Car car;

    @Resource("car")
    private void setCar(Car car){
        System.out.println("execute in setCar");
        this.car = car;
    }
}
```

这时，如果@Resource 未指定“car”属性，则也可以根据属性方法得到需要注入的 Bean 名称。可见@Autowired 默认按类型匹配注入 Bean，@Resource 则按名称匹配注入 Bean。而@Inject 和@Autowired 同样也是按类型匹配注入 Bean 的，只不过它没有 required 属性。可见，不管是@Resource 还是@Inject 注解，其功能都没有@Autowired 丰富，因此，除非必要，大可不必在乎这两个注解。

5.10.4 Bean 作用范围及生命过程方法

通过注解配置的 Bean 和通过<bean>配置的 Bean 一样，默认的作用范围都是 singleton。Spring 为注解配置提供了一个@Scope 注解，可以通过它显式指定 Bean 的作用范围，如代码清单 5-29 所示。

代码清单 5-29 @Scope示例

```
package com.smart.anno;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

 //①指定 Bean 的作用范围为 prototype
@Scope("prototype")
@Component
public class Car {
      ...
}
```

@Scope 注解通过入参指定 Bean 的作用范围，可选择的值参见 5.8 节的说明。

在使用<bean>进行配置时，可以通过 init-method 和 destory-method 属性指定 Bean 的初始化及容器销毁前执行的方法。Spring 从 2.5 开始支持 JSR-250 中定义的@PostConstruct 和@PreDestroy 注解，在 Spring 中它们相当于 init-method 和 destory-method 属性的功能，不过在使用注解时，可以在一个 Bean 中定义多个@PostConstruct 和@PreDestroy 方法，如代码清单 5-30 所示。

代码清单 5-30 @PostConstruct和@PreDestroy示例

```
package com.smart.anno;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component
public class Boss {
    private Car car;
    public Boss(){
        System.out.println("construct...");
    }

    @Autowired
    private void setCar(Car car){
        System.out.println("execute in setCar");
        this.car = car;
    }

    @PostConstruct
    private void init1(){
        System.out.println("execute in init1");
    }

    @PostConstruct
    private void init2(){
        System.out.println("execute in init1");
    }

    @PreDestroy
    private void destory1(){
        System.out.println("execute in destory1");
    }

    @PreDestroy
    private void destory2(){
        System.out.println("execute in destory2");
    }
}
```

在 Boss 类中分别定义了两个@PostConstruct 和两个@PreDestroy 方法，运行如代码清单 5-31 所示的代码启动和关闭容器。

代码清单 5-31　运行测试

```
package com.smart.anno;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.util.Assert;

public class SimpleTest {
    public static void main(String[] args) {
      //①启动容器
      ApplicationContext ctx = new
                    ClassPathXmlApplicationContext("com/smart/anno/beans.xml");
      //②关闭容器
      ((ClassPathXmlApplicationContext)ctx).destroy();
    }
}
```

运行这个测试类，可以在控制台中看到如下输出信息：

```
construct...
execute in setCar
```

```
execute in init1
execute in init1
execute in destory1
execute in destory2
```

这说明 Spring 先调用 Boss 的构造函数实例化 Bean，再执行@Autowired 进行自动注入，然后分别执行标注了@PostConstruct 的方法，在容器关闭时，则分别执行标注了@PreDestroy 的方法。

5.11 基于 Java 类的配置

5.11.1 使用 Java 类提供 Bean 定义信息

JavaConfig 是 Spring 的一个子项目，它旨在通过 Java 类的方式提供 Bean 的定义信息，该项目早在 Spring 2.0 时就已经发布了 1.0 版本。Spring 4.0 基于 Java 类配置的核心就取材于 JavaConfig，JavaConfig 经过若干年的努力终于修成正果，成为 Spring 4.0 的核心功能。

普通的 POJO 只要标注@Configuration 注解，就可以为 Spring 容器提供 Bean 定义的信息，每个标注了@Bean 的类方法都相当于提供了一个 Bean 的定义信息，如代码清单 5-32 所示。

代码清单 5-32 @Configuration示例

```
package com.smart.conf;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

//①将一个 POJO 标注为定义 Bean 的配置类
@Configuration
public class AppConf {

    //②以下两个方法定义了两个 Bean，并提供了 Bean 的实例化逻辑
    @Bean
    public UserDao userDao(){
        return new UserDao();
    }
    @Bean
    public LogDao logDao(){
       return new LogDao();
    }

    //③定义了 logonService 的 Bean
    @Bean
    public LogonService logonService(){
        LogonService logonService = new LogonService();
       //④ 将②和③处定义的 Bean 注入 logonService Bean 中
        logonService.setLogDao(logDao());
```

```
        logonService.setUserDao(userDao());
        return logonService;
    }
}
```

在①处，在 AppConf 类的定义处标注了@Configuration 注解，说明这个类可用于为 Spring 提供 Bean 的定义信息。该类的方法可标注@Bean 注解，Bean 的类型由方法返回值的类型决定，名称默认和方法名相同，也可以通过入参显式指定 Bean 名称，如@Bean(name="userDao")。@Bean 所标注的方法体提供了 Bean 的实例化逻辑。

在②处，userDao()和 logDao()方法定义了一个 UserDao 和一个 LogDao 的 Bean，它们的 Bean 名称分别为 userDao 和 logDao。在③处，又定义了一个 logonService Bean，并且在④处注入在②处所定义的两个 Bean。

因此，以上配置和以下 XML 配置是等效的：

```
<bean id="userDao" class="com.smart.anno.UserDao"/>
<bean id="logDao" class="com.smart.anno.LogDao"/>
<bean id="logonService" class="com.smart.conf.LogonService"
    p:logDao-ref="userDao" p:userDao-ref="logDao"/>
```

基于 Java 类的配置方式和基于 XML 或基于注解的配置方式相比，前者通过代码编程的方式可以更加灵活地实现 Bean 的实例化及 Bean 之间的装配；后两者都是通过配置声明的方式，在灵活性上要稍逊一些，但在配置上要更简单一些。

如果 Bean 在多个@Configuration 配置类中定义，如何引用不同配置类中定义的 Bean 呢？例如，UserDao 和 LogDao 这两个 Bean 在 DaoConfig 中定义，而 logonService Bean 在 ServiceConfig 中定义，logonService Bean 需要引用 DaoConfig 中定义的两个 Bean。我们通过下面的例子说明，如代码清单 5-33 所示。

代码清单 5-33　DaoConfig配置

```
package com.smart.conf;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class DaoConfig {
    @Bean
    public UserDao userDao(){
        return new UserDao();
    }
    @Bean
    public LogDao logDao(){
       return new LogDao();
    }
}
```

由于@Configuration 注解类本身已经标注了@Component 注解，所以任何标注了@Configuration 的类，本身也相当于标注了@Component，即它们可以像普通的 Bean 一样被注入其他 Bean 中。DaoConfig 标注了@Configuration 注解后就成为一个 Bean，它可以被自动注入 ServiceConfig 中，如代码清单 5-34 所示。

代码清单 5-34　ServiceConfig配置

```
package com.smart.conf;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

@Configuration
public class ServiceConfig {

    //①像普通 Bean 一样注入 DaoConfig
    @Autowired
    private DaoConfig daoConfig;

    @Bean
    public LogonService logonService(){
        LogonService logonService = new LogonService();

       //②像普通 Bean 一样，调用 Bean 相关的方法
        logonService.setLogDao(daoConfig.logDao());
        logonService.setUserDao(daoConfig.userDao());
        return logonService;
    }
}
```

调用 daoConfig 的 logDao()和 userDao()方法，就相当于将 DaoConfig 配置类中定义的 Bean 注入进来。Spring 会对配置类所有标注@Bean 的方法进行“改造”（AOP 增强），将对 Bean 生命周期管理的逻辑植入进来。所以，在②处调用 daoConfig.logDao()及 daoConfig.userDao()方法时，不是简单地执行 DaoConfig 类中定义的方法逻辑，而是从 Spring 容器中返回相应 Bean 的单例。换句话说，多次调用 daoConfig.logDao()返回的都是 Spring 容器中相同的 Bean。在@Bean 处，还可以标注@Scope 注解以控制 Bean 的作用范围。如果在@Bean 处标注了@Scope("prototype")，则每次调用 daoConfig.logDao()都会返回一个新的 logDao Bean，如代码清单 5-35 所示。

代码清单 5-35　DaoConfig配置

```
package com.smart.conf;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
@Configuration
public class DaoConfig {

    @Scope("prototype")
    @Bean
    public LogDao logDao(){
        return new LogDao();
    }
    ...
}
```

由于 Spring 容器会自动对@Configuration 的类进行“改造”，以植入 Spring 容器对

Bean 的管理逻辑，所以使用基于 Java 类的配置必须保证将 Spring aop 类包和 CGLIB 类包加载到类路径下。

5.11.2　使用基于 Java 类的配置信息启动 Spring 容器

1. 直接通过@Configuration 类启动 Spring 容器

Spring 提供了一个 AnnotationConfigApplicationContext 类，它能够直接通过标注@Configuration 的 Java 类启动 Spring 容器，如代码清单 5-36 所示。

代码清单 5-36　JavaConfigTest示例

```
package com.smart.conf;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class JavaConfigTest {
    public static void main(String[] args) {
        //①使用@Configuration 类提供的 Bean 定义信息启动容器
        ApplicationContext ctx =
                        new AnnotationConfigApplicationContext(AppConf.class);
        LogonService logonService = ctx.getBean(LogonService.class);
        logonService.printHelllo();
    }
}
```

在①处，通过 AnnotationConfigApplicationContext 类的构造函数直接传入标注@Configuration 的 Java 类，直接用该类中提供的 Bean 定义信息启动 Spring 容器。

此外，AnnotationConfigApplicationContext 还支持通过编码的方式加载多个@Configuration 配置类，然后通过刷新容器应用这些配置类，如代码清单 5-37 所示。

代码清单 5-37　JavaConfigTest示例

```
package com.smart.conf;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class JavaConfigTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplication
Context();

        //①注册多个@Configuration 配置类
        ctx.register(DaoConfig.class);
        ctx.register(ServiceConfig.class);

        //②刷新容器以应用这些注册的配置类
        ctx.refresh();
        LogonService logonService = ctx.getBean(LogonService.class);
        logonService.printHelllo();
    }
}
```

可以通过代码一个一个地注册配置类，也可以通过@Import 将多个配置类组装到一

个配置类中，这样仅需要注册这个组装好的配置类就可以启动容器，如代码清单 5-38 所示。

代码清单 5-38 JavaConfigTest示例

```
package com.smart.conf;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import(DaoConfig.class)
public class ServiceConfig {
    @Bean
    public LogonService logonService(){
        LogonService logonService = new LogonService();
        return logonService;
    }
}
```

2. 通过 XML 配置文件引用@Configuration 的配置

标注了@Configuration 的配置类与标注了@Component 的类一样也是一个 Bean，它可以被 Spring 的<context:component-scan>扫描到。因此，如果希望将配置类组装到 XML 配置文件中，通过 XML 配置文件启动 Spring 容器，则仅需在 XML 中通过<context:component-scan>扫描到相应的配置类即可，如代码清单 5-39 所示。

代码清单 5-39 beans2.xml：装配了配置类的XML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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <!--① 通过上下文扫描加载到 AppConf 的配置类-->
    <context:component-scan base-package="com.smart.conf"
          resource-pattern="AppConf.class" />
</beans>
```

3. 通过@Configuration 配置类引用 XML 配置信息

假设在 beans3.xml 中定义了两个 Bean，如代码清单 5-40 所示。

代码清单 5-40 beans3.xml：定义了两个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-4.0.xsd">
   <bean id="userDao" class="com.smart.conf.UserDao"/>
   <bean id="logDao" class="com.smart.conf.LogDao"/>
</beans>
```

在@Configuration 配置类中可以通过@ImportResource 引入 XML 配置文件，在 LogonAppConfig 配置类中即可直接通过@Autowired 引用 XML 配置文件中定义的 Bean，如代码清单 5-41 所示。

代码清单 5-41　LogonAppConfig.java：引入并组装XML配置信息

```
package com.smart.conf;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

//①通过@ImportResource 引入 XML 配置文件
@Configuration
@ImportResource("classpath:com/smart/conf/beans3.xml")
public class LogonAppConfig {

    //②自动注入 XML 文件中定义的 Bean
    @Bean
    @Autowired
    public LogonService logonService(UserDao userDao,LogDao logDao){
        LogonService logonService = new LogonService();
        logonService.setUserDao(userDao);
        logonService.setLogDao(logDao);
        return logonService;
    }
}
```

在②处完成了两项功能：其一，定义了一个 logonService Bean；其二，通过方法入参自动注入 userDao 和 logDao Bean，这两个 Bean 是在 XML 配置文件中定义的。

需要说明的是，在①处引入定义 userDao 和 logDao Bean 的 XML 配置文件不是②处可成功自动注入 userDao 和 logDao Bean 的前提条件。只要不同形式的 Bean 定义信息能够加载到 Spring 容器中，Spring 就能足够“智能”地完成 Bean 之间的装配。

5.12　基于 Groovy DSL 的配置

5.12.1　使用 Groovy DSL 提供 Bean 定义信息

Groovy 是一种基于 JVM 的敏捷开发语言，它结合了 Python、Ruby 等动态语言的特性。Groovy 代码能够与 Java 代码很好地结合，也能用于扩展现有代码。由于其运行在 JVM 上的特性，所以 Groovy 可以使用其他 Java 语言编写的库。

目前 Groovy 在 Spring 框架构建、Bean 配置等方面的应用非常广泛。Spring 模块化构建采用的工具 Gradle 也是以 Groovy DSL 作为基础的。

Spring 4.0 支持使用 Groovy DSL 来进行 Bean 定义配置，类似于 XML 的配置，只不过配置信息是由 Groovy 脚本表达的，可以实现任何复杂的 Bean 配置，需要 Groovy 2.3.1 以上版本，如代码清单 5-42 所示。

代码清单 5-42　spring-context.groovy配置示例

```
import com.smart.anno.LogDao
import com.smart.groovy.LogonService
import com.smart.groovy.UserDao

beans {
     //①声明context命名空间
     xmlns context: "http://www.springframework.org/schema/context"

     //②与注解混合使用，定义注解Bean扫描包路径
     context.'component-scan'('base-package': "com.smart.groovy") {

          //③排除不需要扫描的包路径
          'exclude-filter'('type': "aspectj", 'expression': "com.smart.xml.*")
     }

    //④读取app-conf.properties配置文件
    def stream;
    def config = new Properties();
    try{
       stream = new ClassPathResource('conf/app-conf.properties').inputStream
       config.load(stream);
    }finally {
       if(stream!=null){
        stream.close()
     }
    }

     //⑤配置无参构造函数Bean
     logDao(LogDao){
          bean->
               bean.scope = "prototype"          //配置当前Bean的作用域
               bean.initMethod="init"            //配置当前Bean的初始化方法
               bean.destroyMethod="destory"      //配置当前Bean的销毁方法
               bean.lazyInit =true               //配置当前Bean的懒加载
     }

    //⑥根据条件注入Bean，这是Groovy DSL定义Bean的一大亮点
     if("db"==config.get("dataProvider")){
         userDao(DbUserDao)
     }else{
         userDao(XmlUserDao)
      }

     //⑦配置有参构造函数注入Bean，参数是userDao
     logonService(LogonService,userDao){
         logDao = ref("logDao") //⑧配置属性注入，引用Groovy定义Bean
         mailService = ref("mailService") //⑨配置属性注入，引用注解定义Bean
    }
}
```

在①处声明 context 命名空间，在 Groovy 脚本中，可以定义注解 Bean 扫描方式，也就是在 Groovy DSL 中可以灵活引用通过注解定义的 Bean。在②处定义注解 Bean 扫描包路径，并在③处定义不扫描的包路径。在④处，在 Groovy 脚本中加载资源配置文

件。在⑤处注入一个无参构造函数 Bean，其中 logDao 表示定义 Bean 的名称，括号内的 LogDao 表示要定义 Bean 的类名称。在⑥处根据应用配置文件所配置的 dataProvider 选项值决定要注入的 Bean，这是 Groovy DSL 配置 Bean 的一个重要灵活性的体现。在⑦处定义了一个有参构造函数 Bean，括号内的第一个参数表示定义 Bean 的类名称，第二个参数表示 LogonService 构造函数参数 userDao。在⑦处采用属性注入 logDao 引用，其中 ref()方法表示要引用容器中定义的 Bean；在⑧处也通过属性注入 mailService，与⑦处不同的是，此处引用的 Bean 是通过注解定义的。

5.12.2　使用 GenericGroovyApplicationContext 启动 Spring 容器

Spring 为基于 Groovy 的配置提供了专门的 ApplicationContext 实现类：GenericGroovyApplicationContext。来看一个如何使用 GenericGroovyApplicationContext 启动 Spring 容器的示例，如代码清单 5-43 所示。

代码清单 5-43　基于Groovy DSL的测试示例

```
package com.smart.groovy;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericGroovyApplicationContext;
import org.testng.annotations.Test;
import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertNotNull;

public class LogonServiceTest {

    @Test
    public  void getBean(){

        //①加载指定Groovy Bean配置文件来创建容器
        ApplicationContext ctx = new GenericGroovyApplicationContext("classpath:com/
            smart/groovy/spring-context.groovy");

        //②加载Groovy定义的Bean
        LogonService logonService = ctx.getBean(LogonService.class);
        assertNotNull(logonService);

        //③加载注解定义的Bean
        MailService mailService = ctx.getBean(MailService.class);
        assertNotNull(mailService);

        //④判断注入是否是DbUserDao
        UserDao userDao = ctx.getBean(UserDao.class);
        assertTrue(userDao instanceof DbUserDao);
}
```

5.13 通过编码方式动态添加 Bean

5.13.1 通过 DefaultListableBeanFactory

DefaultListableBeanFactory 实现了 ConfigurableListableBeanFactory 接口，提供了可扩展配置、循环枚举的功能，可以通过此类实现 Bean 动态注入。为了实现在 Spring 容器启动阶段能动态注入自定义 Bean，保证动态注入的 Bean 也能被 AOP 所增强，需要实现 Bean 工厂后置处理器接口 BeanFactoryPostProcessor。如示例中的 UserServiceFactoryBean 实现了 BeanFactoryPostProcessor#postProcessBeanFactory()方法，并在此方法中动态创建并注入 userService1 和 userService2 实例到容器中，如代码清单 5-44 所示。

代码清单 5-44 UserServiceFactoryBean

```
package com.smart.dynamic;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.stereotype.Component;

@Component
public class UserServiceFactoryBean  implements  BeanFactoryPostProcessor {

    public void postProcessBeanFactory(ConfigurableListableBeanFactory bf)
                                        throws BeansException {

        //①将ConfigurableListableBeanFactory转化为DefaultListableBeanFactory
        DefaultListableBeanFactory beanFactory = (DefaultListableBeanFactory) bf;

        //②通过BeanDefinitionBuilder创建Bean定义
        BeanDefinitionBuilder beanDefinitionBuilder =
                  BeanDefinitionBuilder.genericBeanDefinition(UserService.class);

        //③设置属性userDao，此属性引用已经定义的bean:userDao
        beanDefinitionBuilder.addPropertyReference("userDao","userDao");

        //④注册Bean定义
        beanFactory.registerBeanDefinition("userService1",
                                  beanDefinitionBuilder.getRawBeanDefinition());

        //⑤直接注册一个Bean实例
        beanFactory.registerSingleton("userService2",new UserService());
    }

}
```

在①处显式地将 ConfigurableListableBeanFactory 转化为 DefaultListableBeanFactory。在②处通过 BeanDefinitionBuilder 创建了一个 UserService Bean 定义，并通过

addPropertyReference()方法向 UserService Bean 注入已定义的 UserDao。可以通过 DefaultListableBeanFactory 的 registerBeanDefinition()方法注册 UserService 定义，也可以通过 registerSingleton()方法直接注入一个 Bean 实例。

5.13.2　扩展自定义标签

在开发产品级组件的时候，为了更好地封装组件、增强组件的易用性，一般都将组件进行标签化定义（如 Dubbo、Rop）。Spring 为第三方组件自定义标签提供了强有力的支持。在 Spring 中，自定义组件标签非常方便，只需经过以下几个步骤：

（1）采用 XSD 描述自定义标签的元素属性。

（2）编写 Bean 定义的解析器。

（3）注册自定义标签解析器。

（4）绑定命名空间解析器。

自定义标签的第一步是定义标签元素的 XML 结构，采用 XSD 描述自定义标签的元素属性，下面以用户服务作为自定义标签，如代码清单 5-45 所示。

代码清单 5-45　用户服务标签userservice.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
    <xsd:schema xmlns="http://www.smart.com/schema/service"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:beans="http://www.springframework.org/schema/beans"
                targetNamespace="http://www.smart.com/schema/service" ①
                elementFormDefault="qualified"
                attributeFormDefault="unqualified">
        <xsd:import namespace="http://www.springframework.org/schema/beans"/> ②
        <xsd:element name="user-service"> ③
            <xsd:complexType>
                <xsd:complexContent>
                    <xsd:extension base="beans:identifiedType">
                        <xsd:attribute name="dao" type="xsd:string" use="required"/> ④
                    </xsd:extension>
                </xsd:complexContent>
            </xsd:complexType>
        </xsd:element>
</xsd:schema>
```

在①处指定了一个自定义标签的命名空间。在②处导入了 Spring 本身的 beans 命名空间。在③处定义了一个 user-service 标签，并且在 beans:identifiedType 基础上定义了 user-service 标签的扩展属性“dao”，类似继承方式。

定义好标签的 XSD 结构，接下来编写用户服务标签解析类，如代码清单 5-46 所示。

代码清单 5-46　UserServiceDefinitionParser

```
package com.smart.dynamic;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.parsing.BeanComponentDefinition;
```

```
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Element;

public class UserServiceDefinitionParser implements BeanDefinitionParser {

    public BeanDefinition parse(Element element, ParserContext parserContext) {

        //①通过BeanDefinitionBuilder创建Bean定义
        BeanDefinitionBuilder beanDefinitionBuilder =
                BeanDefinitionBuilder.genericBeanDefinition(UserService.class);

        //②获取自定义标签的属性
        String dao = element.getAttribute("dao");
        beanDefinitionBuilder.addPropertyReference("userDao", dao);
        AbstractBeanDefinition beanDefinition = beanDefinitionBuilder.getBeanDefinition();

        //③注册Bean定义
        parserContext.registerBeanComponent(new
                        BeanComponentDefinition( beanDefinition,"userService"));

        return null;
    }

}
```

在①处通过 BeanDefinitionBuilder 创建 UserService 定义。在②处通过 element 元素获取自定义标签的属性 dao，并将已定义的 userDao 以引用的方式动态注入 UserService。在③处通过 parserContext 注册 UserService 定义。

现在可以将 UserServiceDefinitionParser 解析器注册到 Spring 命名空间解析器，如代码清单 5-47 所示。

代码清单 5-47　UserServiceNamespaceHandler

```
package com.smart.dynamic;
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class UserServiceNamespaceHandler extends NamespaceHandlerSupport {
      public void init() {
        registerBeanDefinitionParser("user-service", new UserServiceDefinition
Parser());
      }
}
```

绑定自定义 Bean 解析器很简单，只要继承 NamespaceHandlerSupport 类，实现 NamespaceHandler#init()方法，并在 init()方法中注册自定义 Bean 解析器即可。

最后需要告诉 Spring 如何解析自定义标签。在源码 resources 目录创建 META-INF 文件夹，并在其中创建 spring.handlers 和 spring.schemas 两个文件，告诉 Spring 自定义标签的文档结构及解析它的类，如代码清单 5-48 所示。

代码清单 5-48　spring.handlers 和spring.schemas

spring.schemas文件内容如下：

```
http\://www.smart.com/schema/service.xsd=com/smart/schema/userservice.xsd
```

spring.handlers文件内容如下：

```
http\://www.smart.com/schema/service=com.smart.dynamic.UserServiceNamespaceHandler
```

在 spring.schemas 文件中，告诉 Spring 描述自定义标签的文档结构文件所在的位置。在 spring.handlers 文件中，告诉 Spring 自定义命名空间所对应的解析器。

至此，自定义标签工作全部完成，接下来就可以在 Spring 配置文件中使用自定义的标签，如代码清单 5-49 所示。

代码清单 5-49　引用自定义标签

```
<?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:us="http://www.smart.com/schema/service" ①
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.smart.com/schema/service http://www.smart.com/schema/service.xsd">
    <bean id="userDao" class="com.smart.dynamic.UserDao" />
    <us:user-service dao="userDao"/> ②
</beans>
```

要使用自定义的标签，首先要在 beans 头部引用自定义标签的命名空间，并设置命名空间前缀“us”，之后就可以使用“us”标签声明定义的组件了。

5.14　不同配置方式比较

同一功用商品或服务的多种品牌供给性是市场健康的基本要素。对于 Spring 来说，为实现 Bean 信息定义，它提供了基于 XML、基于注解、基于 Java 类及基于 Groovy 这 4 种选项，同时还允许各种配置方式复合共存。Spring 张开虚怀若谷的胸怀包容气象万千的世界，同时让百态生象可以互通有无、取长补短，最终达到本质纯一、世界大同。Spring 优雅地实现了这个目标，让我们把赞誉毫无保留地敬献给 Spring 的大师们！

下面通过表 5-8 比较一下它们实现 Bean 配置的不同。

表 5-8　Bean不同配置方式比较

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
Bean 定义	在 XML 文件中通过 <bean> 元素定义 Bean，如： <bean class="com.smart.UserDao"/>	在 Bean 实现类处通过标注@Compoent 或衍型类（@Repository、@Service 及 @Controller ）定义 Bean	在标注了@Configuration 的 Java 类中，通过在类方法上标注@Bean 定义一个 Bean。方法必须提供 Bean 的实例化逻辑	在 Groovy 文件中通过 DSL 定义 Bean，如： userDao(UserDao)

续表

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
Bean 名称	通过<bean>的 id 或 name 属性定义，如： `<bean id="userDao" class="com.smart.UserDao "/>` 默认名称为 com.smart.UserDao#0	通过注解的 value 属性定义，如@Component ("userDao")。默认名称为小写字母开头的类名（不带包名）userDao	通过@Bean 的 name 属性定义，如@Bean("userDao")。默认名称为方法名	通过 Groovy 的 DSL 定义 Bean 的名称（Bean 的类型，Bean 构建函数参数），如： `logonService(LogonService, userDao)`
Bean 注入	通过<property>子元素或通过 p 命名空间的动态属性，如 p:userDao-ref="userDao" 进行注入	通过在成员变更或方法入参处标注@Autowired，按类型匹配自动注入。还可以配合使用@Qualifier 按名称匹配方式注入	比较灵活，可以在方法处通过@Autowired 使方法入参绑定 Bean，然后在方法中通过代码进行注入；还可通过调用配置类的@Bean 方法进行注入	比较灵活，可以在方法处通过 ref()方法进行注入，如 ref("logDao")
Bean 生命过程方法	通过<bean>的 init-method 和 destory-method 属性指定 Bean 实现类的方法名。最多只能指定一个初始化方法和一个销毁方法	通过在目标方法上标注 @PostConstruct 和 @PreDestroy 注解指定初始化或销毁方法，可以定义任意多个	通过@Bean 的 initMethod 或 destoryMethod 指定一个初始化或销毁方法。 对于初始化方法来说，可以直接在方法内部通过代码的方式灵活定义初始化逻辑	通过 bean-> bean.initMethod 或 bean.destoryMethod 指定一个初始化或销毁方法
Bean 作用范围	通过<bean>的 scope 属性指定，如： `<bean class=" com.smart.UserDao " scope=" prototype"/>`	通过在类定义处标注@Scope 指定，如@Scope ("prototype")	通过在 Bean 方法定义处标注@Scope 指定	通过 bean-> bean.scope = "prototype"指定
Bean 延迟初始化	通过 <bean> 的 lazy-init 属性指定，默认为 default，继承于 <beans> 的 default-lazy-init 设置，该值默认为 false	通过在类定义处标注@Lazy 指定，如@Lazy (true)	通过在 Beam 方法定义处标注@Lazy 指定	通过 bean-> bean. lazyInit = true 指定

这 4 种配置文件很难说孰优孰劣，只能说它们都有自己的舞台和适用场景，表 5-9 提供了一些参考意见。

表 5-9 Bean不同配置方式的适用场景

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
适用场景	（1）Bean 实现类来源于第三方类库，如 DataSource、JdbcTemplate 等，因无法在类中标注注解，所以通过 XML 配置方式较好。 （2）命名空间的配置，如 aop、context 等，只能采用基于 XML 的配置	Bean 的实现类是当前项目开发的，可以直接在 Java 类中使用基于注解的配置	基于 Java 类配置的优势在于可以通过代码方式控制 Bean 初始化的整体逻辑。如果实例化 Bean 的逻辑比较复杂，则比较适合基于 Java 类配置的方式	基于 Groovy DSL 配置的优势在于可以通过 Groovy 脚本灵活控制 Bean 初始化的过程。如果实例化 Bean 的逻辑比较复杂，则比较适合基于 Groovy DSL 配置的方式

笔者一般采用 XML 配置 DataSource、SessionFactory 等资源 Bean，在 XML 中利用 aop、context 命名空间进行相关主题的配置。其他所有项目中开发的 Bean 都通过基于注解配置的方式进行配置，即整个项目采用“基于 XML+基于注解”的配置方式，很少采用基于 Java 类的配置方式。

5.15　小结

在本章中，我们学习了在 Spring 配置文件中配置 Bean 的相关知识。对于一般的应用来说，这些知识已经可以满足开发的需要。现在对这些知识作一个简要的回顾。

用户不但可以通过属性注入的方式建立 Bean 和 Bean 之间的依赖，也可以通过构造函数的方式完成相同的任务。但前者不管是对于代码的编写还是对于 Bean 的配置都具有更大的灵活性，成为大多数开发者的首选。

用户可以通过字面值的方式设置 Bean 的属性，也可以通过 ref 引用容器中其他的 Bean。由于集合类型是最常使用的数据类型，因此，Spring 为集合类型提供了专门的配置标签。一般情况下，可以使用 Spring 的简化配置方式让配置文件更加紧凑。

不但容器中的 Bean 可以通过配置建立起关联关系，配置文档中的<bean>标签也可以建立继承、依赖、引用关系，合理地使用这些关系可以简化配置、提高配置质量。

Spring 提供了 5 个 Bean 作用范围。在 Web 应用环境下，可以使用 request、session 和 globalSession 的 Bean 作用域。此外，还允许通过编程的方式定义新的 Bean 作用域。

通过@Component 及另外 3 个衍型注解（@Repository、@Service 及@Controller），配合@Autowired 就可以很好地使用基于注解的配置进行 Bean 的定义和注入。这种方式比在 XML 文件中通过<bean>提供的配置更加简单。

任何 POJO 标注了@Configuration 注解后就可以为 Spring 容器提供 Bean 的定义信息，在类方法中标注@Bean 相当于定义了一个 Bean，同时还提供了 Bean 的实例化逻辑。由于 Bean 的实例化逻辑是在方法中定义的，因此，它可以应对一些复杂的 Bean 实例化场景。

不管使用何种配置方式，Bean 都可以很好地将它们整合起来。在 Spring 容器内部，这些不同方式的 Bean 定义信息是大体相同的，四者之间并不存在谁替代谁的问题，它们都有自己适合的应用场景。

第 6 章

Spring 容器高级主题

Spring 容器是一部设计精妙的机器，其优异的外在表现是通过精密的内部设计实现的。本章将对 Spring 容器进行解构，从内部探究 Spring 容器的体系结构和运行机理。此外，还将对 Spring 容器的一些高级主题进行深入阐述。

本章主要内容：

- 分析 Spring 容器的内部结构
- Spring 属性编辑器
- 使用外部属性文件
- 国际化信息
- 容器事件体系

本章亮点：

- 详细分析了 Spring 容器主要的构成类和内部流程
- 讲解了使用外部加密属性文件的技巧
- 简明扼要地介绍了相关的 Java 基础知识

6.1 Spring 容器技术内幕

Spring 容器就像一台构造精妙的机器，我们通过配置文件向机器传达控制信息，机器就能够按照设定的模式工作。如果将 Spring 容器比作一辆汽车，那么可以将 BeanFactory 看成汽车的发动机，而 ApplicationContext 则是一辆完整的汽车，它不但包括发动机，还包括离合器、变速器及底盘、车身、电气设备等其他组件。在 ApplicationContext 内，各个组件按部就班、有条不紊地完成汽车的各项功能。

第 4 章和第 5 章介绍了 Spring 容器的功用，现在让我们打开“机盖”，看看底下究竟隐藏着哪些秘密。

6.1.1　内部工作机制

Spring 的 AbstractApplicationContext 是 ApplicationContext 的抽象实现类，该抽象类的 refresh()方法定义了 Spring 容器在加载配置文件后的各项处理过程，这些处理过程清晰地刻画了 Spring 容器启动时所执行的各项操作。下面来看一下 refresh()内部定义了哪些执行逻辑，如代码清单 6-1 所示。

代码清单 6-1　AbstractApplicationContext#refresh()

```
//①初始化BeanFactory
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
...
//②调用工厂后处理器
invokeBeanFactoryPostProcessors();

//③注册Bean后处理器
registerBeanPostProcessors();

//④初始化消息源
initMessageSource();

//⑤初始化应用上下文事件广播器
initApplicationEventMulticaster();

//⑥初始化其他特殊的Bean：由具体子类实现
onRefresh();

//⑦注册事件监听器
registerListeners();

//⑧初始化所有单实例的Bean，使用懒加载模式的Bean除外
finishBeanFactoryInitialization(beanFactory);

//⑨完成刷新并发布容器刷新事件
finishRefresh();
```

（1）初始化 BeanFactory：根据配置文件实例化 BeanFactory，在 obtainFreshBeanFactory()方法中，首先调用 refreshBeanFactory()方法刷新 BeanFactory，然后调用 getBeanFactory()方法获取 BeanFactory，这两个方法都是由具体子类实现的。在这一步里，Spring 将配置文件的信息装入容器的 Bean 定义注册表（BeanDefinitionRegistry）中，但此时 Bean 还未初始化。

（2）调用工厂后处理器：根据反射机制从 BeanDefinitionRegistry 中找出所有实现了 BeanFactoryPostProcessor 接口的 Bean，并调用其 postProcessBeanFactory()接口方法。

（3）注册 Bean 后处理器：根据反射机制从 BeanDefinitionRegistry 中找出所有实现

了 BeanPostProcessor 接口的 Bean，并将它们注册到容器 Bean 后处理器的注册表中。

（4）初始化消息源：初始化容器的国际化消息资源。

（5）初始化应用上下文事件广播器。

（6）初始化其他特殊的 Bean：这是一个钩子方法，子类可以借助这个方法执行一些特殊的操作，如 AbstractRefreshableWebApplicationContext 就使用该方法执行初始化 ThemeSource 的操作。

（7）注册事件监听器。

（8）初始化所有单实例的 Bean，使用懒加载模式的 Bean 除外：初始化 Bean 后，将它们放入 Spring 容器的缓存池中。

（9）发布上下文刷新事件：创建上下文刷新事件，事件广播器负责将这些事件广播到每个注册的事件监听器中。

在 4.5 节中，我们观摩了 Bean 从创建到销毁的生命历程，这些过程都可以在上面的流程中找到对应的步骤。Spring 协调多个组件共同完成这个复杂的作业流程。图 6-1 描述了 Spring 容器从加载配置文件到创建出一个完整 Bean 的作业流程及参与的角色。

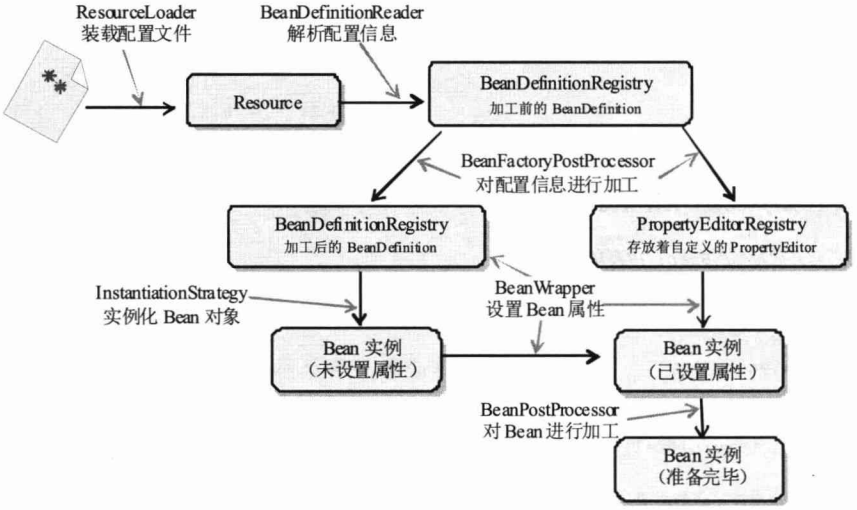


图 6-1　IoC 的流水线

（1）ResourceLoader 从存储介质中加载 Spring 配置信息，并使用 Resource 表示这个配置文件资源。

（2）BeanDefinitionReader 读取 Resource 所指向的配置文件资源，然后解析配置文件。配置文件中的每个<bean>解析成一个 BeanDefinition 对象，并保存到 BeanDefinitionRegistry 中。

（3）容器扫描 BeanDefinitionRegistry 中的 BeanDefinition，使用 Java 反射机制自动识别出 Bean 工厂后处理器（实现 BeanFactoryPostProcessor 接口的 Bean），然后调用这些 Bean 工厂后处理器对 BeanDefinitionRegistry 中的 BeanDefinition 进行加工处理。主要

完成以下两项工作：

① 对使用占位符的<bean>元素标签进行解析，得到最终的配置值。这意味着对一些半成品式的 BeanDefinition 对象进行加工处理并得到成品的 BeanDefinition 对象。

② 对 BeanDefinitionRegistry 中的 BeanDefinition 进行扫描，通过 Java 反射机制找出所有属性编辑器的 Bean（实现 java.beans.PropertyEditor 接口的 Bean），并自动将它们注册到 Spring 容器的属性编辑器注册表中（PropertyEditorRegistry）。

（4）Spring 容器从 BeanDefinitionRegistry 中取出加工后的 BeanDefinition，并调用 InstantiationStrategy 着手进行 Bean 实例化的工作。

（5）在实例化 Bean 时，Spring 容器使用 BeanWrapper 对 Bean 进行封装。BeanWrapper 提供了很多以 Java 反射机制操作 Bean 的方法，它将结合该 Bean 的 BeanDefinition 及容器中的属性编辑器，完成 Bean 属性注入工作。

（6）利用容器中注册的 Bean 后处理器（实现 BeanPostProcessor 接口的 Bean）对已经完成属性设置工作的 Bean 进行后续加工，直接装配出一个准备就绪的 Bean。

Spring 容器堪称一部设计精密的机器，其内部拥有众多的组件和装置。Spring 的高明之处在于，它使用众多接口描绘出了所有装置的协作蓝图，构建好 Spring 的骨架，继而通过继承体系层层推演、不断丰富，最终让 Spring 成为有血有肉的完整的框架。所以在查看 Spring 框架的源码时，有两条清晰可见的脉络：

（1）接口层描述了容器的重要组件及组件间的协作关系。

（2）继承体系逐步实现组件的各项功能。

接口层清晰地勾勒出 Spring 框架的高层功能，框架脉络呼之欲出。有了接口层抽象的描述后，不但 Spring 自己可以提供具体的实现，任何第三方组织也可以提供不同的实现，可以说 Spring 完善的接口层使框架的扩展性得到了很好的保证。纵向继承体系的逐步扩展，分步骤地实现框架的功能，这种实现方案保证了框架功能不会堆积在某些类身上，从而造成过重的代码逻辑负载，框架的复杂度被完美地分解开了。

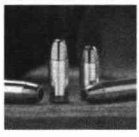
Spring 组件按其所承担的角色可以划分为两类。

（1）物料组件：Resource、BeanDefinition、PropertyEditor 及最终的 Bean 等，它们是加工流程中被加工、被消费的组件，就像流水线上被加工的物料一样。

（2）设备组件：ResourceLoader、BeanDefinitionReader、BeanFactoryPostProcessor、InstantiationStrategy 及 BeanWrapper 等。它们就像流水线上不同环节的加工设备，对物料组件进行加工处理。

第 4 章介绍了 Resource 和 ResourceLoader 两个组件，本章将对其他组件进行讲解。

轻松一刻



尼古拉斯·凯奇主演的《军火之王》具有《教父》般的磅礴气势。片头很有创意，镜头跟着一颗流水线上的子弹不断前进，经过一道道工序，最终子弹走下流水线，打包、装箱、运输、卸货、开包、上膛、发射，最

后击中了目标。对 Spring 容器进行深入分析后，读者也能感受到这种按部就班、有条不紊地预设流程的魅力。

6.1.2 BeanDefinition

org.springframework.beans.factory.config.BeanDefinition 是配置文件<bean>元素标签在容器中的内部表示。<bean>元素标签拥有 class、scope、lazy-init 等配置属性，BeanDefinition 则提供了相应的 beanClass、scope、lazyInit 类属性，BeanDefinition 就像<bean>的镜中人，二者是一一对应的。BeanDefinition 类的继承结构如图 6-2 所示。

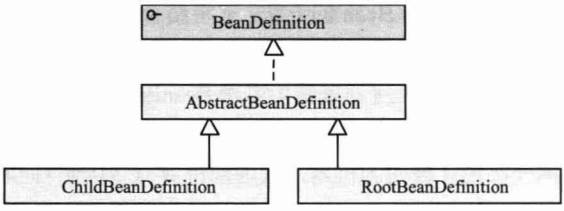


图 6-2 BeanDefinition 类继承结构

RootBeanDefinition 是最常用的实现类，它对应一般性的<bean>元素标签。我们知道，在配置文件中可以定义父<bean>和子<bean>，父<bean>用 RootBeanDefinition 表示，子<bean>用 ChildBeanDefiniton 表示，而没有父<bean>的<bean>则用 RootBeanDefinition 表示。AbstractBeanDefinition 对二者共同的类信息进行抽象。

Spring 通过 BeanDefinition 将配置文件中的<bean>配置信息转换为容器的内部表示，并将这些 BeanDefiniton 注册到 BeanDefinitonRegistry 中。Spring 容器的 BeanDefinitionRegistry 就像 Spring 配置信息的内存数据库，后续操作直接从 BeanDefinitionRegistry 中读取配置信息。一般情况下，BeanDefinition 只在容器启动时加载并解析，除非容器刷新或重启，这些信息不会发生变化。当然，如果用户有特殊的需求，也可以通过编程的方式在运行期调整 BeanDefinition 的定义。

创建最终的 BeanDefinition 主要包括两个步骤。

（1）利用 BeanDefinitionReader 读取承载配置信息的 Resource，通过 XML 解析器解析配置信息的 DOM 对象，简单地为每个<bean>生成对应的 BeanDefinition 对象。但是这里生成的 BeanDefinition 可能是半成品，因为在配置文件中，可能通过占位符变量引用外部属性文件的属性，这些占位符变量在这一步里还没有被解析出来。

（2）利用容器中注册的 BeanFactoryPostProcessor 对半成品的 BeanDefinition 进行加工处理，将以占位符表示的配置解析为最终的实际值，这样半成品的 BeanDefinition 就成为成品的 BeanDefinition。

提示

Spring 框架源码类包层次结构清晰，但包名很长。在 IDE 环境下，这不是问题，但却给书面表达带来了困难。为了行文方便，常常将 Spring 类的包名省略。如果用户希望了解类位于哪个具体的包中，在 IDEA 中，可以按 Ctrl+N 组合键，输入类名，IDEA 将自动查找出这个类。推荐另一个小工具：Search and Replace，它被称为文本搜索工具中的“倚天剑”。它能深入到 ZIP、JAR 等类型的文件中进行搜索，用户可以通过这个工具轻易地找出类所在的位置。

6.1.3　InstantiationStrategy

org.springframework.beans.factory.support.InstantiationStrategy 负责根据 BeanDefinition 对象创建一个 Bean 实例。Spring 之所以将实例化 Bean 的工作通过一个策略接口进行描述，是为了可以方便地采用不同的实例化策略，以满足不同的应用需求，如通过 CGLib 类库为 Bean 动态创建子类再进行实例化。InstantiationStrategy 类的继承结构如图 6-3 所示。

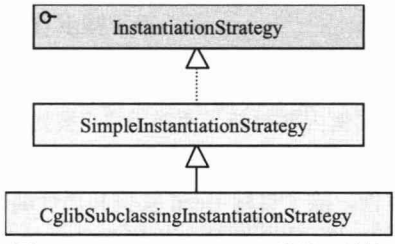


图 6-3　InstantiationStrategy 类继承结构

SimpleInstantiationStrategy 是最常用的实例化策略，该策略利用 Bean 实现类的默认构造函数、带参构造函数或工厂方法创建 Bean 的实例。

CglibSubclassingInstantiationStrategy 扩展了 SimpleInstantiationStrategy，为需要进行方法注入的 Bean 提供了支持。它利用 CGLib 类库为 Bean 动态生成子类，在子类中生成方法注入的逻辑，然后使用这个动态生成的子类创建 Bean 的实例。

InstantiationStrategy 仅负责实例化 Bean 的操作，相当于执行 Java 语言中 new 的功能，它并不会参与 Bean 属性的设置工作。所以由 InstantiationStrategy 返回的 Bean 实例实际上是一个半成品的 Bean 实例，属性填充的工作留待 BeanWrapper 来完成。

6.1.4　BeanWrapper

org.springframework.beans.BeanWrapper 是 Spring 框架中重要的组件类。BeanWrapper 相当于一个代理器，Spring 委托 BeanWrapper 完成 Bean 属性的填充工作。在 Bean 实例被 InstantiationStrategy 创建出来之后，容器主控程序将 Bean 实例通过 BeanWrapper 包

装起来，这是通过调用 BeanWrapper#setWrappedInstance(Object obj)方法完成的。BeanWrapper 类的继承结构如图 6-4 所示。

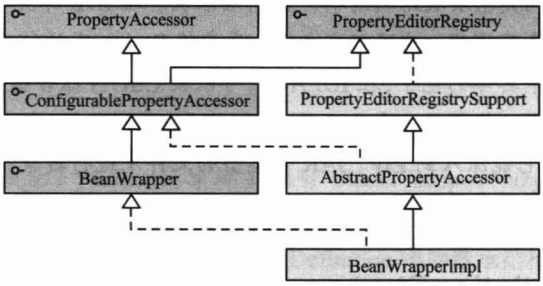


图 6-4 BeanWrapper 类继承结构

通过图 6-4 可以看出，BeanWrapper 还有两个顶级类接口，分别是 PropertyAccessor 和 PropertyEditorRegistry。PropertyAccessor 接口定义了各种访问 Bean 属性的方法，如 setPropertyValue(String,Object)、setPropertyValues(PropertyValues pvs)等；而 PropertyEditorRegistry 是属性编辑器的注册表。所以 BeanWrapper 实现类 BeanWrapperImpl 具有三重身份：

（1）Bean 包裹器。

（2）属性访问器。

（3）属性编辑器注册表。

一个 BeanWrapperImpl 实例内部封装了两类组件：被封装的待处理的 Bean，以及一套用于设置 Bean 属性的属性编辑器。

要顺利地填充 Bean 属性，除了目标 Bean 实例和属性编辑器外，还需要获取 Bean 对应的 BeanDefinition，它从 Spring 容器的 BeanDefinitionRegistry 中直接获取。Spring 主控程序从 BeanDefinition 中获取 Bean 属性的配置信息 PropertyValue，并使用属性编辑器对 PropertyValue 进行转换以得到 Bean 的属性值。对 Bean 的其他属性重复这样的步骤，就可以完成 Bean 所有属性的注入工作。BeanWrapperImpl 在内部使用 Spring 的 BeanUtils 工具类对 Bean 进行反射操作，设置属性。下一节将详细介绍属性编辑器的原理，并讲解如何通过配置的方式注册自定义的属性编辑器。

6.2 属性编辑器

在 Spring 配置文件里，往往通过字面值为 Bean 各种类型的属性提供设置值：不管是 double 类型还是 int 类型，在配置文件中都对应字符串类型的字面值。BeanWrapper 在填充 Bean 属性时如何将这个字面值正确地转换为对应的 double 或 int 等内部类型呢？我们可以隐约地感觉到一定有一个转换器在“暗中相助”，这个转换器就是属性编辑器。

“属性编辑器”这个名字可能会让人误以为是一个带用户界面的输入器，其实属性编辑器不一定非得有用户界面，任何实现 java.beans.PropertyEditor 接口的类都是属性编

辑器。属性编辑器的主要功能就是将外部的设置值转换为 JVM 内部的对应类型，所以属性编辑器其实就是一个类型转换器。

PropertyEditor 是 JavaBean 规范定义的接口，JavaBean 规范中还有其他一些 PropertyEditor 配置的接口。为了彻底地理解属性编辑器，必须对 JavaBean 中有关属性编辑器的规范进行学习，相信这些知识对学习和掌握 Spring 中的属性编辑器会大有帮助。

6.2.1　JavaBean 的编辑器

Sun 所制定的 JavaBean 规范很大程度上是为 IDE 准备的——它让 IDE 能够以可视化的方式设置 JavaBean 的属性。如果在 IDE 中开发一个可视化的应用程序，则需要通过属性设置的方式对组成应用的各种组件进行定制，IDE 通过属性编辑器让开发人员使用可视化的方式设置组件的属性。

一般的 IDE 都支持 JavaBean 规范所定义的属性编辑器，当组件开发商发布一个组件时，它往往将组件对应的属性编辑器捆绑发行，这样开发者就可以在 IDE 环境下方便地利用属性编辑器对组件进行定制工作。

JavaBean 规范通过 java.beans.PropertyEditor 定义了设置 JavaBean 属性的方法，通过 BeanInfo 描述了 JavaBean 的哪些属性是可定制的，此外还描述了可定制属性与 PropertyEditor 的对应关系。

BeanInfo 与 JavaBean 之间的对应关系通过二者之间的命名规范确立。对应 JavaBean 的 BeanInfo 采用如下的命名规范：*<Bean>*BeanInfo。如 ChartBean 对应的 BeanInfo 为 ChartBeanBeanInfo；Car 对应的 BeanInfo 为 CarBeanInfo。当 JavaBean 连同其属性编辑器一同注册到 IDE 中后，在开发界面中对 JavaBean 进行定制时，IDE 就会根据 JavaBean 规范找到对应的 BeanInfo，再根据 BeanInfo 中的描述信息找到 JavaBean 属性描述（是否开放、使用哪个属性编辑器），进而为 JavaBean 生成特定的开发编辑界面。

JavaBean 规范提供了一个管理默认属性编辑器的管理器 PropertyEditorManager，该管理器内保存着一些常见类型的属性编辑器。如果某个 JavaBean 的常见类型属性没有通过 BeanInfo 显式指定属性编辑器，那么 IDE 将自动使用 PropertyEditorManager 中注册的对应默认属性编辑器。

由于 JavaBean 对应的属性编辑器等 IDE 环境相关的资源和组件需要动态加载，所以在纯 Java 的 IDE 中开发基于组件的应用时，总会感觉 IDE 反应很迟钝，不像 Delphi、C++Builder 一样灵敏快捷。但在 IDEA 开发环境中，设计包括可视化组件的应用时却很快捷，原因是 IDEA 没有使用 Java 的标准用户界面组件库，当然也就没有按照 JavaBean 的规范开发设计 GUI 组件。

1．PropertyEditor

PropertyEditor 是属性编辑器的接口，它规定了将外部设置值转换为内部 JavaBean

属性值的转换接口方法。PropertyEditor 主要的接口方法说明如下。

- Object getValue()：返回属性的当前值，基本类型被封装成对应的封装类实例。
- void setValue(Object newValue)：设置属性的值，基本类型以封装类传入。
- String getAsText()：将属性对象用一个字符串表示，以便外部的属性编辑器能以可视化的方式显示。默认返回 null，表示该属性不能以字符串表示。
- void setAsText(String text)：用一个字符串去更新属性的内部值，这个字符串一般从外部属性编辑器传入。
- String[] getTags()：返回表示有效属性值的字符串数组（如 boolean 属性对应的有效 Tag 为 true 和 false），以便属性编辑器能以下拉框的方式显示出来。默认返回 null，表示属性没有匹配的字符值有限集合。
- String getJavaInitializationString()：为属性提供一个表示初始值的字符串，属性编辑器以此值作为属性的默认值。

可以看出，PropertyEditor 接口方法是内部属性值和外部设置值的沟通桥梁。此外，可以很容易地发现该接口的很多方法是专为 IDE 中的可视化属性编辑器提供的，如 getTags()、getJavaInitializationString()及另外一些未曾介绍的接口方法。

Java 为 PropertyEditor 提供了一个方便类 PropertyEditorSupport，该类实现了 PropertyEditor 接口并提供了默认实现。一般情况下，用户可以通过扩展这个方便类设计自己的属性编辑器。

2．BeanInfo

BeanInfo 主要描述了 JavaBean 的哪些属性可以编辑及对应的属性编辑器，每个属性对应一个属性描述器 PropertyDescriptor。PropertyDescriptor 的构造函数有两个入参：PropertyDescriptor(String propertyName, Class beanClass)，其中 propertyName 为属性名，beanClass 为 JavaBean 对应的 Class。此外，PropertyDescriptor 还有一个 setPropertyEditorClass (Class propertyEditorClass)方法，用于为 JavaBean 属性指定编辑器。BeanInfo 接口最重要的方法就是 PropertyDescriptor[] getPropertyDescriptors()，该方法返回 JavaBean 的属性描述器数组。

BeanInfo 接口有一个常用的实现类 SimpleBeanInfo，一般情况下，可以通过扩展 SimpleBeanInfo 实现自己的功能。

3．一个实例

下面来看一个具体的属性编辑器实例，该实例根据 *Core Java II* 中的一个例子改编而成。

ChartBean 是一个可定制图表组件，允许通过属性设置定制图表的样式，以得到满足各种不同使用场合要求的图表。我们忽略 ChartBean 的其他属性，仅关注其中的两个属性，如代码清单 6-2 所示。

代码清单 6-2　ChartBean

```
public class ChartBean extends JPanel{
   private int titlePosition = CENTER;
   private boolean inverse;
   //省略 get/setter 方法
}
```

下面为 titlePosition 属性提供一个属性编辑器。我们不去直接实现 PropertyEditor，而是通过扩展 PropertyEditorSupport 这个方便类来定义属性编辑器，如代码清单 6-3 所示。

代码清单 6-3　TitlePositionEditor

```
import java.beans.*;
public class TitlePositionEditor extends PropertyEditorSupport{
   private String[] options = { "Left", "Center", "Right" };

   //①代表可选属性值的字符串标识数组
   public String[] getTags() { return options; }

   //②代表属性初始值的字符串
   public String getJavaInitializationString() { return "" + getValue(); }

   //③将内部属性值转换为对应的字符串表示形式，供属性编辑器显示之用
   public String getAsText(){
      int value = (Integer) getValue();
      return options[value];
   }

   //④将外部设置的字符串转换为内部属性的值
   public void setAsText(String s){
      for (int i = 0; i < options.length; i++){
         if (options[i].equals(s)){
            setValue(i);
            return;
         }
      }
   }
}
```

①处通过 getTags()方法返回一个字符串数组，因此在 IDE 中该属性对应的编辑器将自动提供一个下拉框，下拉框中包含 3 个可选项：“Left”、“Center”、“Right”。而③和④处的两个方法分别完成属性值到字符串的双向转换功能。ChartBean 的 inverse 属性也有一个相似的编辑器 InverseEditor，我们忽略不讲。

下面编写 ChartBean 对应的 BeanInfo。根据 JavaBean 的命名规范，这个 BeanInfo 应该命名为 ChartBeanBeanInfo，它负责将属性编辑器和 ChartBean 的属性联系起来，如代码清单 6-4 所示。

代码清单 6-4　ChartBeanBeanInfo

```
import java.beans.*;
 public class ChartBeanBeanInfo extends SimpleBeanInfo{
```

```
    public PropertyDescriptor[] getPropertyDescriptors()
{
      try{
         PropertyDescriptor titlePositionDescriptor                                    ①
            = new PropertyDescriptor("titlePosition", ChartBean.class);
         titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
                                                        将 TitlePositionEditor 绑定到
                                                        ChartBean 的 titlePosition 属性中
       PropertyDescriptor inverseDescriptor
            = new PropertyDescriptor("inverse", ChartBean.class);                      ②
         inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
         return new PropertyDescriptor[]{titlePositionDescriptor, inverseDescriptor};
      }
      catch (IntrospectionException e){                  将 InverseEditor 绑定
         e.printStackTrace();                            到 ChartBean 的 inverse 属性中
         return null;
      }
    }
}
```

在 ChartBeanBeanInfo 中分别为 ChartBean 的 titlePosition 和 inverse 属性指定对应的属性编辑器。将 ChartBean 连同属性编辑器及 ChartBeanBeanInfo 打成 JAR 包，使用 IDE 组件扩展管理功能注册到 IDE 中。这样就可以像使用 TextField、Checkbox 等组件一样对 ChartBean 进行可视化的开发设计工作了。下面是 ChartBean 在 NetBeans IDE 中的属性编辑器效果图，如图 6-5 所示。

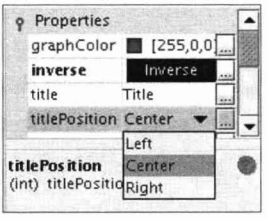


图 6-5　ChartBean 的属性编辑器

ChartBean 可设置的属性都列在属性查看器中。在单击 titlePosition 属性时，下拉框中列出了我们提供的 3 个选项。

6.2.2　Spring 默认属性编辑器

Spring 的属性编辑器和传统的用于 IDE 开发时的属性编辑器不同，它们没有 UI 界面，仅负责将配置文件中的文本配置值转换为 Bean 属性的对应值，所以 Spring 的属性编辑器并非传统意义上的 JavaBean 属性编辑器。

Spring 为常见的属性类型提供了默认的属性编辑器。从图 6-4 中可以看出，BeanWrapperImpl 类扩展了 PropertyEditorRegistrySupport 类，Spring 在 PropertyEditor RegistrySupport 中为常见属性类型提供了默认的属性编辑器，这些“常见的类型”共 32 个，可分为三大类，总结如表 6-1 所示。

表 6-1　Spring提供的默认属性编辑器

类　别	说　明
基础数据类型	分为几个小类： （1）基本数据类型，如 boolean、byte、short、int 等。 （2）基本数据类型封装类，如 Long、Character、Integer 等。 （3）两个基本数据类型的数组，即 char[]和 byte[]。 （4）大数类，即 BigDecimal 和 BigInteger
集合类	为 5 种类型的集合类 Collection、Set、SortedSet、List 和 SortedMap 提供了编辑器
资源类	用于访问外部资源的 8 个常见类，即 Class、Class[]、File、InputStream、Locale、Properties、Resource[]和 URL

PropertyEditorRegistrySupport 中有两个用于保存属性编辑器的 Map 类型变量。

- defaultEditors：用于保存默认属性类型的编辑器，元素的键为属性类型，值为对应的属性编辑器实例。
- customEditors：用于保存用户自定义的属性编辑器，元素的键值和 defaultEditors 相同。

PropertyEditorRegistrySupport 通过类似以下的代码定义默认属性编辑器：

```
defaultEditors.put(char.class, new CharacterEditor(false));
defaultEditors.put(Character.class, new CharacterEditor(true));
defaultEditors.put(Locale.class, new LocaleEditor());
defaultEditors.put(Properties.class, new PropertiesEditor());
```

这些默认的属性编辑器用于解决常见属性类型的注册问题。如果用户的应用包括一些特殊类型的属性，且希望在配置文件中以字面值提供配置，那么就需要编写自定义属性编辑器并注册到 Spring 容器中。这样，Spring 才能将配置文件中的属性配置值转换为对应的属性类型值。

6.2.3　自定义属性编辑器

Spring 的大部分默认属性编辑器都直接扩展于 java.beans.PropertyEditorSupport 类，开发者也可以通过扩展 PropertyEditorSupport 实现自己的属性编辑器。相对于用于 IDE 环境的属性编辑器来说，Spring 环境下使用的属性编辑器的功能非常单一，仅需将配置文件中的字面值转换为属性类型的对象即可，并不需要提供 UI 界面，因此仅需简单覆盖 PropertyEditorSupport 的 setAsText()方法即可。

1．一个实例

继续使用第 5 章中 Boss 和 Car 的例子。假设现在希望在配置 Boss 时不通过引用 Bean 的方式注入 Boss 的 car 属性，而希望直接通过字面值提供配置。为了方便阅读，这里再次列出 Car 和 Boss 类的简要代码，如代码清单 6-5 和 6-6 所示。

代码清单 6-5 Car

```
package com.smart.editor;
public class Car {
    private int maxSpeed;
    public String brand;
    private double price;
   //省略get/setter
}
```

代码清单 6-6 Boss

```
package com.smart.editor;
public class Boss {
    private String name;
    private Car car = new Car();
   //省略get/setter
}
```

Boss 有两个属性：name 和 car，分别对应 String 类型和 Car 类型。Spring 拥有 String 类型的默认属性编辑器，因此，我们无须关心 String 类型的属性。但 Car 类型是我们自定义的类型，要配置 Boss 的 car 属性，有两种方案。

（1）在配置文件中为 car 专门配置一个<bean>，然后在 Boss 的<bean>中通过 ref 引用 car Bean，这正是第 5 章中所用的方法。

（2）为 Car 类型提供一个自定义的属性编辑器，这样就可以通过字面值为 Boss 的 car 属性提供配置值。

第一种方案是常用的方法，但在某些情况下，这种方式需要将属性对象一步步肢解为最终可以用基本类型配置的 Bean，使配置文件变得不够清晰。直接为属性类提供一个对应的自定义属性编辑器可能会是更好的替代方案。

现在来为 Car 编写一个自定义的属性编辑器，如代码清单 6-7 所示。

代码清单 6-7 CustomCarEditor

```
package com.smart.editor;
import java.beans.PropertyEditorSupport;

public class CustomCarEditor extends PropertyEditorSupport {

   //①将字面值转换为属性类型对象
   public void setAsText(String text){
        if(text == null || text.indexOf(",") == -1){
            throw new IllegalArgumentException("设置的字符串格式不正确");
        }
        String[] infos = text.split(",");
        Car car = new Car();
        car.setBrand(infos[0]);
        car.setMaxSpeed(Integer.parseInt(infos[1]));
        car.setPrice(Double.parseDouble(infos[2]));

       //②调用父类的setValue()方法设置转换后的属性对象
        setValue(car);
    }
}
```

CustomCarEditor 很简单，它仅覆盖 PropertyEditorSupport 便利类的 setAsText(String text)方法，该方法负责将配置文件以字符串提供的字面值转换为 Car 对象。字面值采用逗号分隔格式字符串的同时为 brand、maxSpeed 和 price 属性值提供设置值，setAsText()方法解析这个字面值并生成对应的 Car 对象。由于并不需要将 Boss 内部的 car 属性回显到属性编辑器的 UI 界面中，因此不需要覆盖 getAsText()方法。

2．注册自定义的属性编辑器

在 IDE 环境下，自定义属性编辑器在使用之前必须通过扩展组件功能进行注册，在 Spring 环境下也需要通过一定的方法注册自定义的属性编辑器。

如果使用 BeanFactory，则用户需要手工调用 registerCustomEditor(Class requiredType, PropertyEditor propertyEditor)方法注册自定义的属性编辑器；如果使用 ApplicationContext，则只需在配置文件中通过 CustomEditorConfigurer 注册即可。CustomEditorConfigurer 实现了 BeanFactoryPostProcessor 接口，因而是一个 Bean 工厂后处理器。我们知道，Bean 工厂后处理器在 Spring 容器中加载配置文件并生成 BeanDefinition 半成品后就会被自动执行。因此，CustomEditorConfigurer 在容器启动时有机会注入自定义的属性编辑器。下面的配置片段定义了一个 CustomEditorConfigurer：

```
<!--①配置自动注册属性编辑器的CustomEditorConfigurer -->
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
               <!--② 属性编辑器对应的属性类型-->
               <entry key="com.smart.editor.Car"
                              value="com.smart.editor.CustomCarEditor"/>

                </entry>
            </map>
        </property>
    </bean>
 <bean id="boss" class="com.smart.editor.Boss">
     <property name="name" value="John"/>
     <!--③该属性将使用②处的属性编辑器完成属性填充操作-->
     <property name="car" value="红旗CA72,200,20000.00"/>
</bean>
```

在①处定义了用于注册自定义属性编辑器的 CustomEditorConfigurer，Spring 容器将通过反射机制自动调用这个 Bean。CustomEditorConfigurer 通过一个 Map 属性定义需要自动注册的自定义属性编辑器。在②处为 Car 类型指定了对应的属性编辑器 CustomCarEditor，其中键是属性类型，值是属性编辑器的类名。

最精彩的当然是③处的配置。原来通过一个<bean>元素标签配置好 car Bean，然后在 Boss 的<bean>中通过 ref 引用 car Bean，而现在直接通过 value 为 car 属性提供配置。BeanWrapper 在设置 Boss 的 car 属性时，将检索自定义属性编辑器的注册表，当发现 Car 属性类型拥有对应的属性编辑器 CustomCarEditor 时，就会利用 CustomCarEditor 将“红旗 CA72,200,20000.00”转换为 Car 对象。

提示

按照 JavaBean 的规范，JavaBean 的基础设施会在 JavaBean 的相同类包下查找是否存在<JavaBean>Editor 的规范类。如果存在，则自动使用<JavaBean>Editor 作为该 JavaBean 的 PropertyEditor。

如 com.smart.domain.UserEditor 会自动成为 com.smart.domain.User 对应的 PropertyEditor。Spring 也支持这个规范，如果采用这种规约命令 PropertyEditor，就无须显式在 CustomEditorConfigurer 中注册了，Spring 将自动查找并注册这个 PropertyEditor。

6.3 使用外部属性文件

在进行数据源或邮件服务器等资源的配置时，用户可以直接在 Spring 配置文件中配置用户名/密码、链接地址等信息。但一种更好的做法是将这些配置信息独立到一个外部属性文件中，并在 Spring 配置文件中通过形如${user}、${password}的占位符引用属性文件中的属性项。这种配置方式拥有两个明显的好处。

- 减少维护的工作量：资源的配置信息可以被多个应用共享，在多个应用使用同一资源的情况下，如果资源用户名/密码、链接地址等配置信息发生更改，则用户只需调整独立的属性文件即可。
- 使部署更简单：Spring 配置文件主要描述应用工程中的 Bean，这些配置信息在开发完成后就基本固定下来了。但数据源、邮件服务器等资源配置信息却需要在部署时根据现场情况确定。如果通过一个独立的属性文件存放这些配置信息，则部署人员只需调整这个属性文件即可，根本不需要关注结构复杂、信息量大的 Spring 配置文件。这不仅给部署和维护带来了方便，也降低了出错的概率。

Spring 提供了一个 PropertyPlaceholderConfigurer，它能够使 Bean 在配置时引用外部属性文件。PropertyPlaceholderConfigurer 实现了 BeanFactoryPostProcessorBean 接口，因而也是一个 Bean 工厂后处理器。

6.3.1 PropertyPlaceholderConfigurer 属性文件

1. 使用 PropertyPlaceholderConfigurer 属性文件

回忆一下在 2.3.4 节中定义的数据源，如下：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/sampledb"
    p:userName="root"
    p:password="123456"/>
```

驱动器类名、JDBC 的 URL 地址及数据库用户名/密码都直接写在 XML 文件中，部

署人员在部署应用时，必须先找出这个 Bean 配置 XML 文件，再找出数据源 Bean 定义的代码段进行调整，给部署工作带来了很大的困难。

根据实际应用中的最佳实战，可以将这些需要调整的配置信息抽取到一个配置文件中。这里使用一个名为 jdbc.properties 的配置文件，如代码清单 6-8 所示。

代码清单 6-8　jdbc.properties

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/sampledb
userName=root
password=123456
```

属性文件可以定义多个属性，每个属性都由一个属性名和一个属性值组成，二者用“=”隔开。下面通过 PropertyPlaceholderConfigurer 引入 jdbc.properties 属性文件，调整数据源 Bean 的配置。

```
<!--①引入jdbc.properties属性文件 -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
    p:location="classpath:com/smart/placeholder/jdbc.properties"
    p:fileEncoding="utf-8"/>

<!--②通过属性名引用属性值 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${driverClassName}"
    p:url="${url}"
    p:username="${userName}"
    p:password="${password}"/>
```

在①处通过 PropertyPlaceholderConfigurer 的 location 属性引入属性文件，这样，在 Bean 定义的时候就可以引用属性文件中的属性了，如②处的粗体部分所示。通过这样的调整后，部署人员在部署本应用时，仅需关注 jdbc.properties 这个配置文件即可，无须关心 Spring 的配置文件。

2. PropertyPlaceholderConfigurer 的其他属性

除了 location 属性外，PropertyPlaceholderConfigurer 还有一些常用的属性，在一些高级应用中可能会用到。

- locations：如果只有一个属性文件，则直接使用 location 属性指定就可以了；如果有多个属性文件，则可以通过 locations 属性进行设置。可以像配置 List 一样配置 locations 属性，参见 5.4.6 节。
- fileEncoding：属性文件的编码格式。Spring 使用操作系统默认编码读取属性文件。如果属性文件采用了特殊编码，则需要通过该属性显式指定。
- order：如果配置文件中定义了多个 PropertyPlaceholderConfigurer，则通过该属性指定优先顺序。
- placeholderPrefix：在上面的例子中，通过${属性名}引用属性文件中的属性项，其中“${”为默认的占位符前缀，可以根据需要改为其他的前缀符。
- placeholderSuffix：占位符后缀，默认为“}”。

3. 使用<context:property-placeholder>引用属性文件

可以使用 context 命名空间定义属性文件，相比于传统的 PropertyPlaceholderConfigurer 配置，这种方法更加优雅。

```
<context:property-placeholder
      location="classpath:com/smart/placeholder/jdbc.properties" />
```

以上配置就相当于在 Spring 容器中定义了一个 PropertyPlaceholderConfigurer 的 Bean。虽然这种方式比较优雅，但如果希望自定义一些额外的高级功能，如属性加密、使用数据库表来保存配置信息等，则必须使用扩展 PropertyPlaceholderConfigurer 的类并使用 Bean 的配置方式（参见 6.3.2 节）。

4. 在基于注解及基于 Java 类的配置中引用属性

在基于 XML 的配置文件中，通过“${propName}”形式引用属性值。类似的，基于注解配置的 Bean 可以通过@Value 注解为 Bean 的成员变量或方法入参自动注入容器已有的属性，如下：

```
package com.smart.placeholder;
import org.apache.commons.lang.builder.ToStringBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MyDataSource {

    @Value("${driverClassName}")
     private String driverClassName;

    @Value("${url}")
     private String url;

    @Value("${userName}")
     private String userName;

    @Value("${password}")
     private String password;

    public String toString(){
      return ToStringBuilder.reflectionToString(this);
    }
}
```

@Value 注解可以为 Bean 注入一个字面值，也可以通过@Value("${propName}")的形式根据属性名注入属性值。由于标注@Configuration 的类本身就相当标注了@Component，所以在标注@Configuration 的类中引用属性的方式和基于注解配置的引用方式是完全一样的，此处不再赘述。

使用@Value 注解来引用属性值带来很大的便利，但在使用过程中，一定要确保所引用的属性值在属性文件中已经存在且数值匹配，否则会造成 Bean 创建错误，引发意想不到的异常。

6.3.2　使用加密的属性文件

对于那些不敏感的属性信息，以明文形式出现在属性文件中是合适的。但如果属性信息是数据库用户名/密码等敏感信息，一般情况下则希望以密文的方式保存。虽然 Web 应用系统的客户端用户看不到服务器端的属性文件，但允许登录到 Web 应用系统所在服务器的内部人员却可以轻易查看到属性文件的内容。如果属性文件以明文形式保存着访问数据库的用户名/密码等信息，那么任何拥有服务器登录权限的人都可能查看到这些机密信息，容易造成数据库访问权限的泄露。

对于一些对安全要求特别高的应用系统（如电信、银行、公安的重点人口库等）来说，这些敏感信息应该只掌握在少数特定维护人员的手中，而不是毫无保留地对所有可以进入部署机器的人员开放。这就要求对应用程序配置文件的某些属性进行加密，让 Spring 容器在读取属性文件后，在内存中对属性进行解密，然后再将解密后的属性值赋给目标对象。

PropertyPlaceholderConfigurer 继承自 PropertyResourceConfigurer 类，后者有几个有用的 protected 方法，用于在属性使用之前对属性列表中的属性进行转换。

- void convertProperties(Properties props)：属性文件中的所有属性值都封装在 props 中，覆盖此方法，可以对所有的属性值进行转换处理。
- String convertProperty(String propertyName, String propertyValue)：在加载属性文件并读取文件中的每个属性时，都会调用此方法进行转换处理。
- String convertPropertyValue(String originalValue)：和上一个方法类似，只不过没有传入属性名。

在默认情况下，这 3 个方法内部都是空的，即不会对属性值进行任何转换。可以扩展 PropertyPlaceholderConfigurer，覆盖相应的属性转换方法，就可以支持加密版的属性文件了。

1．DES 加密解密工具类

信息的加密可分为对称和非对称两种方式，前者表示加密后的信息可以解密成原值，而后者则不能根据加密后的信息还原为原值。MD5 属于非对称加密，而 DES 属于对称加密。先用 DES 对属性值进行加密，在读取到属性值时，再用 DES 进行解密。下面是支持 DES 加密解密的工具类，如代码清单 6-9 所示。

代码清单 6-9　DESUtils.java：DES加密解密工具类

```
package com.smart.placeholder;

import java.security.Key;
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;
```

```
public class DESUtils {

    //①指定DES加密解密所用的密钥
    private static Key key;
    private static String KEY_STR = "myKey";
    static {
        try {
            KeyGenerator generator = KeyGenerator.getInstance("DES");
            generator.init(new SecureRandom(KEY_STR.getBytes()));
            key = generator.generateKey();
            generator = null;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    //②对字符串进行DES加密，返回BASE64编码的加密字符串
    public static String getEncryptString(String str) {
        BASE64Encoder base64en = new BASE64Encoder();
        try {
            byte[] strBytes = str.getBytes("UTF8");
            Cipher cipher = Cipher.getInstance("DES");
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] encryptStrBytes = cipher.doFinal(strBytes);
            return base64en.encode(encryptStrBytes);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    //③对BASE64编码的加密字符串进行解密，返回解密后的字符串
    public static String getDecryptString(String str) {
        BASE64Decoder base64De = new BASE64Decoder();
        try {
            byte[] strBytes = base64De.decodeBuffer(str);
            Cipher cipher = Cipher.getInstance("DES");
            cipher.init(Cipher.DECRYPT_MODE, key);
            byte[] decryptStrBytes = cipher.doFinal(strBytes);
            return new String(decryptStrBytes, "UTF8");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

    }

    //④对入参的字符串进行加密，打印出加密后的串
    public static void main(String[] args) throws Exception {
        if (args == null || args.length < 1) {
            System.out.println("请输入要加密的字符，用空格分隔.");
        } else {
            for (String arg : args) {
                System.out.println(arg + ":" + getEncryptString(arg));
```

```
            }
        }
    }
}
```

上述示例使用 DES 算法打造加密解密工具类。DES 加密解密的关键是加密密钥，在①处设置了一个值为“myKey”的加密密钥。②处的方法可以获取明文字符串的加密串，它以 BASE64 进行编码。BASE64 编码是以大小写字母、数字及其他几个字符组成的编码串，给书写表达带来了方便。③处的方法可以将加密后的字符串解密出来。④处的方法可以将 main()方法入参的值加密并打印出来。

2. 使用密文版的属性文件

可以通过 DESUtils 类对数据库用户名和密码进行加密。在编译好 DESUtils 工具类后，在 DOS 窗口下通过命令获取用户名及密码的密文，如图 6-6 所示。

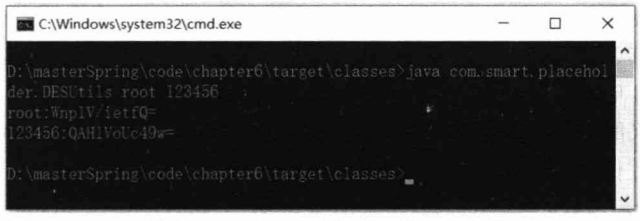


图 6-6　使用 DESUtils 通过命令方式对信息进行加密

在 DOS 窗口下输入 java com.smart.placeholder.DESUtils root 123456 命令，DESUtils 将使用 DES 对 root 及 123456 进行加密并返回加密后的密文。用此密文更改 jdbc.properties 的相关属性值，如下：

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/sampledb
userName=WnplV/ietfQ=
    password= QAHlVoUc49w=
```

PropertyPlaceholderConfigurer 本身不支持密文版的属性文件，不过我们扩展该类，覆盖 String convertProperty(String propertyName, String propertyValue)方法，对 userName 及 password 的属性值进行解密转换，如代码清单 6-10 所示。

代码清单 6-10　EncryptPropertyPlaceholderConfigurer.java

```
package com.smart.placeholder;
import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;

//①继承PropertyPlaceholderConfigurer定义支持密文版属性的属性配置器
public class EncryptPropertyPlaceholderConfigurer extends
PropertyPlaceholderConfigurer {
    private String[] encryptPropNames ={"userName","password"};

    //②对特定属性的属性值进行转换
    @Override
    protected String convertProperty(String propertyName, String propertyValue) {

        if(isEncryptProp(propertyName)){
```

```
            String decryptValue = DESUtils.getDecryptString(propertyValue);
            System.out.println(decryptValue);
            return decryptValue;
        }else{
            return propertyValue;
        }
    }

    //③判断是否是需要解密的属性
    private boolean isEncryptProp(String propertyName){
        for(String encryptPropName:encryptPropNames){
            if(encryptPropName.equals(propertyName)){
                return true;
            }
        }
        return false;
    }
}
```

EncryptPropertyPlaceholderConfigurer 使用前面的 DESUtils 类，对已经加密的属性进行解密转换，以便属性的最终读者可以获取解密版的内容。

使用自定义的属性加载器后，就无法使用<context:property-placeholder>引用属性文件了，必须通过传统的配置方式引用加密版的属性文件。

```
<bean class="com.smart.placeholder.EncryptPropertyPlaceholderConfigurer"
      p:location="classpath:com/smart/placeholder/jdbc.properties"
      p:fileEncoding="utf-8"/>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${driverClassName}"
      p:url="${url}"
      p:username="${userName}"
      p:password="${password}" />
```

属性文件加载器的实现类改为 EncryptPropertyPlaceholderConfigurer，其他配置和原来的相同。

6.3.3 属性文件自身的引用

Spring 既允许在 Bean 定义中通过${propName}引用属性值，也允许在属性文件中使用${propName}实现属性之间的相互引用。

```
dbName=sampledb
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/${dbName}
```

在上面的属性文件定义中，url 属性的值通过${dbName}引用了另一个属性的值。对于一些复杂的属性，可以通过这种方式将属性变化的部分抽取出来，实现配置的最小化。

提示

如果一个属性值太长，一行写不下，则可以通过在行后添加“\”将属性值划分为多行，如：

```
desc=desc content desc content desc content\
    desc content desc content
```

6.4　引用 Bean 的属性值

将应用系统的配置信息放在配置文件中并非总是最适合的。如果应用系统是以集群方式部署的，或者希望在运行期动态调整应用系统的某些配置，这时，将配置信息放到数据库中不但方便集中管理，而且可以通过应用系统的管理界面动态维护，有效增强应用系统的可维护性。

在早期版本中，要在配置文件中使 Bean 引用另一个 Bean 的属性值是比较麻烦的，Spring 3.0 则提供了优雅的解决方案。在 Spring 3.0 中，可以通过类似#{beanName.beanProp}的方式方便地引用另一个 Bean 的值，如代码清单 6-11 所示。

代码清单 6-11　SysConfig.java：提供配置信息的类

```
package com.smart.beanprop;

public class SysConfig {
    private int sessionTimeout;
    private int maxTabPageNum;
    private DataSource dataSource;

    //①模拟从数据库中获取配置值并设置相应的属性
    public void initFromDB(){
        ...
       //模拟从数据库获取配置值
        this.sessionTimeout = 30;
        this.maxTabPageNum = 10;
    }

    public int getSessionTimeout() {
        return sessionTimeout;
    }

    public int getMaxTabPageNum() {
        return maxTabPageNum;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

在①处的方法中可以更改代码，从数据库中获取相应的属性信息。为了简化代码，

仅采用直接设置属性值的方式演示属性引用的过程。

在 XML 配置文件中，先将 SysConfig 定义为一个 Bean，这样在定义数据源时即可通过#{beanName.propName}的方式引用 Bean 的属性值，如代码清单 6-12 所示。

代码清单 6-12　beans.xml：引用Bean的属性值

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.0.xsd">


    <context:property-placeholder
         location="classpath:com/smart/placeholder/jdbc.properties"/>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${driverClassName}"
        p:url="${url}"
        p:username="${userName}"
        p:password="${password}" />


    <!--① 通过initFromDB方法从数据源中获取配置属性值 -->
   <bean id="sysConfig" class="com.smart.beanprop.SysConfig"
         init-method="initFromDB"
         p:dataSource-ref="dataSource"/>

   <!--② 引用Bean的属性值-->
    <bean class="com.smart.beanprop.ApplicationManager"
       p:maxTabPageNum="#{sysConfig.maxTabPageNum}"
       p:sessionTimeout="#{sysConfig.sessionTimeout}"/>
</beans>
```

访问数据库获取配置属性值的前提是连接到数据库中，为此，需要使用外部属性文件配置数据库的连接信息。在生产环境下，一般通过 JNDI 引用应用容器的数据源，此时属性文件仅提供数据源 JNDI 名即可。然后通过 sysConfig 的 initFromDB()方法访问数据库，获取应用系统的配置信息，并将其保存在 sysConfig 的属性中。

其他需要访问应用系统配置信息的 Bean 即可通过#{beanName.propName}表达式引用 sysConfig Bean 的属性值，如②处所示。

在基于注解和基于 Java 类配置的 Bean 中，可以通过@Value("#{beanName.propName}")的注解形式引用 Bean 的属性值。

```
package com.smart.beanprop;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component
```

```
public class ApplicationManager {

    @Value("#{sysConfig.sessionTimeout}")
    private int sessionTimeout;

    @Value("#{sysConfig.maxTabPageNum}")
    private int maxTabPageNum;
    ...
}
```

6.5　国际化信息

假设我们正在开发一个支持多国语言的 Web 应用程序，要求系统能够根据客户端系统的语言类型返回对应的界面：英文的操作系统返回英文界面，而中文的操作系统则返回中文界面——这便是典型的 i18n 国际化问题。对于有国际化要求的应用系统，我们不能简单地采用硬编码的方式编写用户界面信息、报错信息等内容，而必须为这些需要国际化的信息进行特殊处理。简单来说，就是为每种语言提供一套相应的资源文件，并以规范化命名的方式保存在特定的目录中，由系统自动根据客户端语言选择适合的资源文件。

6.5.1　基础知识

“国际化信息”也称为“本地化信息”，一般需要两个条件才可以确定一个特定类型的本地化信息，分别是“语言类型”和“国家/地区类型”。如中文本地化信息既有中国大陆地区的中文，又有中国台湾、中国香港地区的中文，还有新加坡地区的中文。Java 通过 java.util.Locale 类表示一个本地化对象，它允许通过语言参数和国家/地区参数创建一个确定的本地化对象。

语言参数使用 ISO 标准语言代码表示，这些代码是由 ISO-639 标准定义的，每种语言由两位小写字母表示。在许多网站上都可以找到这些代码的完整列表，下面的网址提供了标准语言代码的信息：http://www.loc.gov/standards/iso639-2/php/English_list.php。

国家/地区参数也由标准的 ISO 国家/地区代码表示，这些代码是由 ISO-3166 标准定义的，每个国家/地区由两个大写字母表示。用户可以从以下网址查看 ISO-3166 的标准代码：http://www.iso.org/iso/country_codes。

表 6-2 给出了一些语言和国家/地区的标准代码。

表 6-2　语言和国家/地区代码示例

语　言	代　码	国家/地区代码	代　码
中文	zh	中国大陆	CN
英语	en	中国台湾	TW

续表

语　言	代　码	国家/地区代码	代　码
法语	Fr	中国香港	HK
德语	de	英国	EN
日语	ja	美国	US
韩语	ko	加拿大	CA

1. Locale

java.util.Locale 是表示语言和国家/地区信息的本地化类，它是创建国际化应用的基础。下面给出几个创建本地化对象的示例：

```
//①带有语言和国家/地区信息的本地化对象
Locale locale1 = new Locale("zh","CN");
//②只有语言信息的本地化对象
Locale locale2 = new Locale("zh");
//③等同于Locale("zh","CN")
Locale locale3 = Locale.CHINA
//④等同于Locale("zh")
Locale locale4 = Locale.CHINESE
//⑤获取本地系统默认的本地化对象
Locale locale 5= Locale.getDefault()
```

用户既可以同时指定语言和国家/地区参数定义一个本地化对象（见①处），也可以仅通过语言参数定义一个泛本地化对象（见②处）。Locale 类中通过静态常量定义了一些常用的本地化对象，③和④处直接通过引用常量返回本地化对象。此外，用户还可以获取系统默认的本地化对象，如⑤处所示。

提示

在测试时，如果希望改变系统默认的本地化设置，则可以在启动 JVM 时通过命令参数指定：java -Duser.language=en -Duser.region=US MyTest。

2. 本地化工具类

JDK 的 java.util 包中提供了几个支持本地化的格式化操作工具类，如 NumberFormat、DateFormat、MessageFormat。下面分别通过实例了解它们的用法，如代码清单 6-13～6-15 所示。

代码清单 6-13　NumberFormat

```
Locale locale = new Locale("zh", "CN");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(locale);
double amt = 123456.78;
System.out.println(currFmt.format(amt));
```

上面的实例通过 NumberFormat 按本地化的方式对货币金额进行格式化操作。运行实例，输出以下信息：

```
¥123,456.78
```

代码清单 6-14　DateFormat

```
Locale locale = new Locale("en", "US");
Date date = new Date();
DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM, locale);
System.out.println(df.format(date));
```

通过 DateFormat.getDateInstance(int style,Locale locale)方法按本地化的方式对日期进行格式化操作。该方法的第一个入参为时间样式，第二个入参为本地化对象。运行以上代码，输出以下信息：

```
Jan 8, 2007
```

MessageFormat 在 NumberFormat 和 DateFormat 的基础上提供了强大的占位符字符串的格式化功能，支持时间、货币、数字及对象属性的格式化操作。下面的实例演示了一些常见的格式化功能，如代码清单 6-15 所示。

代码清单 6-15　MessageFormat

```
//①格式化信息串
String pattern1 = "{0}，你好！你于{1}在工商银行存入{2} 元。";
String pattern2 = "At {1,time,short} On{1,date,long}, {0} paid {2,number, currency}.";

//②用于动态替换占位符的参数
Object[] params = {"John", new GregorianCalendar().getTime(),1.0E3};

//③使用默认的本地化对象格式化信息
String msg1 = MessageFormat.format(pattern1,params);

//④使用指定的本地化对象格式化信息
MessageFormat mf = new MessageFormat(pattern2,Locale.US);
String msg2 = mf.format(params);
System.out.println(msg1);
System.out.println(msg2);
```

pattern1 是简单形式的格式化信息串，通过{n}占位符指定动态参数的替换位置索引，{0}表示第一个参数，{1}表示第二个参数，以此类推。

pattern2 格式化信息串比较复杂一些，除参数位置索引外，还指定了参数的类型和样式。从 pattern2 中可以看出格式化信息串的语法是很灵活的，一个参数甚至可以出现在两个地方。如{1,time,short}表示从第二个入参中获取时分秒部分的值，显示为短样式时间；而{1,date,long}表示从第二个入参中获取日期部分的值，显示为长样式时间。关于 MessageFormat 更详细的使用方法，请参见 JDK 的 Javadoc 文档。

在②处定义了用于替换格式化占位符的动态参数，这里用到了 JDK 6.0 自动装包的语法，否则必须采用封装类表示基本类型的参数值。

在③处通过 MessageFormat 的 format()方法格式化信息串。它使用了系统默认的本地化对象，由于是中文平台，因此默认为 Locale.CHINA。而在④处显式指定了 MessageFormat 的本地化对象。

运行上面的代码，输出以下信息：

```
John，你好！你于16-3-15 下午5:46在工商银行存入1,000 元。
At 5:46 PM OnMarch 15, 2016, John paid $1,000.00.
```

3. ResourceBoundle

如果应用系统中的某些信息需要支持国际化功能，则必须为期望支持的不同本地化类型分别提供对应的资源文件，并以规范的方式进行命名。国际化资源文件的命名规范规定资源名称采用以下方式进行命名：

```
<资源名>_<语言代码>_<国家/地区代码>.properties
```

其中，语言代码和国家/地区代码都是可选的。<资源名>.properties 命名的国际化资源文件是默认的资源文件，即某个本地化类型在系统中找不到对应的资源文件，就采用这个默认的资源文件。<资源名>_<语言代码>.properties 命名的国际化资源文件是某一语言默认的资源文件，即某个本地化类型在系统中找不到精确匹配的资源文件，就采用相应语言默认的资源文件。

举个例子：假设资源名为“resource”，语言为英文，国家/地区为美国，则与其对应的本地化资源文件命名为 **resource_en_US.properties**。信息在资源文件以属性名/值的方式表示，如下：

```
greeting.common=How are you!
greeting.morning = Good morning!
greeting.afternoon = Good Afternoon!
```

对应语言为中文、国家/地区为中国大陆的本地化资源文件则命名为 **resource_zh_CN.properties**，资源文件内容如下：

```
greeting.common=\u60a8\u597d\uff01
greeting.morning=\u65e9\u4e0a\u597d\uff01
greeting.afternoon=\u4e0b\u5348\u597d\uff01
```

本地化不同的同一资源文件，虽然属性值各不相同，但属性名却是相同的，这样应用程序就可以通过 Locale 对象和属性名精确定位到某个具体的属性值。

读者可能已经注意到，上面中文的本地化资源文件内容采用了特殊的编码形式，这是因为资源文件对文件内容有严格的要求：只能包含 ASCII 字符。所以必须将非 ASCII 字符的内容转换为 Unicode 代码的表示方式。如上面中文的 resource_zh_CN.properties 资源文件的 3 个属性值分别是“您好!”、“早上好!”和“下午好!”这 3 个中文字符串所对应的 Unicode 代码串。

在应用开发时，直接采用 Unicode 编码编辑资源文件是很不方便的。所以，通常情况下直接使用正常的方式编写资源文件，在测试或部署时再采用工具进行转换。JDK 在 bin 目录下提供了一个完成此项功能的 native2ascii 工具，它可以将中文字符的资源文件转换为 Unicode 编码格式的文件，命令格式如下：

```
native2ascii [-reverse] [-encoding 编码] [输入文件 [输出文件]]
```

resource_zh_CN.properties 包含中文字符并且以 UTF-8 进行编码。假设将该资源文件放到 d:\目录下，通过下面的命令就可以将其转换为 Unicode 编码的形式：

```
D:\>native2ascii -encoding utf-8 d:\resource_zh_CN.properties
d:\resource_zh_CN_1.properties
```

由于原资源文件采用 UTF-8 编码，所以必须显式地通过-encoding 指定编码格式。

实战经验

通过 native2ascii 命令手工转换资源文件，不但在操作上不方便，转换后资源文件中的属性内容由于采用了 ASCII 编码，阅读起来也不方便。很多 IDE 开发工具都有属性文件编辑器的插件，插件会自动将资源文件内容转换为 ASCII 形式的编码，同时以正常的方式阅读和编辑资源文件的内容，这给开发和维护工作带来了很大的便利。Intellij IDEA 即支持这种透明化编辑资源文件的功能，不过默认未启用，可通过如下方式开启：Setting → Editor→File Encoding→勾选“Transparent native-to-ascii conversion”复选框；对于 MyEclipse 来说，可以使用 MyEclipse Properties Editor 编辑资源属性文件。

如果应用程序中拥有大量的本地化资源文件，则直接通过传统的 File 操作资源文件显然太过笨拙。Java 提供了用于加载本地化资源文件的方便类 java.util.ResourceBoundle。ResourceBoundle 为加载及访问资源文件提供了便捷的操作。下面的语句从相对于类路径的目录中加载一个名为 resource 的本地化资源文件：

```
ResourceBundle rb = ResourceBundle.getBundle("com/smart/i18n/resource", locale)
```

通过以下代码即可访问资源文件的属性值：

```
rb.getString("greeting.common")
```

来看下面的实例，如代码清单 6-16 所示。

代码清单 6-16　ResourceBoundle

```
ResourceBundle rb1 = ResourceBundle.getBundle("com/smart/i18n/resource", Locale.US);
ResourceBundle rb2 = ResourceBundle.getBundle("com/smart/i18n/resource", Locale.CHINA);
System.out.println("us:"+rb1.getString("greeting.common"));
System.out.println("cn:"+rb2.getString("greeting.common"));
```

rb1 加载了对应美国英语本地化的 resource_en_US.properties 资源文件；而 rb2 加载了对应中国大陆中文的 resource_zh_CN.properties 资源文件。运行上面的代码，将输出以下信息：

```
us:How are you!
cn:您好!
```

在加载资源文件时，如果不指定本地化对象，则将使用本地系统默认的本地化对象。所以，在中文系统中，ResourceBundle.getBundle("com/smart/i18n/resource")语句也将返回和代码清单 6-16 中 rb2 相同的本地化资源。

ResourceBundle 在加载资源时，如果指定的本地化资源文件不存在，则按以下顺序尝试加载其他资源：本地系统默认本地化对象对应的资源→默认的资源。在上面的例子中，假设使用 ResourceBundle.getBundle("com/smart/i18n/resource",Locale.CANADA)加载资源，由于不存在 resource_en_CA.properties 资源文件，它将尝试加载 resource_zh_CN.properties 资源文件。假设 resource_zh_CN.properties 资源文件也不存在，它将继续尝试加载 resource.properties 资源文件。如果这些资源文件都不存在，则将抛出 java.util.MissingResourceException 异常。

4．在资源文件中使用格式化串

在上面的资源文件中，属性值都是一个普通的字符串，它们不能结合运行时的动态参数构造出灵活的信息，而这种需求是很常见的。要解决这个问题很简单，只需使用带占位符的格式化串作为资源文件的属性值并结合使用 MessageFormat 就可以满足要求。

在上面的例子中，仅向用户提供了一般性问候。下面对资源文件进行改造，通过格式化串让问候语更具个性化，如下：

```
greeting.common=How are you!{0},today is {1}
greeting.morning = Good morning!{0},now is {1 time short}
greeting.afternoon = Good Afternoon!{0} now is {1 date long}
```

将该资源文件保存在 fmt_resource_en_US.properties 中，按照同样的方式编写对应的中文本地化资源文件 fmt_resource_zh_CN.properties。

下面联合使用 ResourceBoundle 和 MessageFormat 得到“接地气”的问候语，如代码清单 6-17 所示。

代码清单 6-17　资源文件格式化串处理

```
//①加载本地化资源
ResourceBundle rb1 =
          ResourceBundle.getBundle("com/smart/i18n/fmt_ resource",Locale.US);
ResourceBundle rb2 =
           ResourceBundle.getBundle("com/smart/i18n/fmt_ resource",Locale.CHINA);
Object[] params = {"John", new GregorianCalendar().getTime()};

                                                    用本地化对象进行格式化
String str1 = new MessageFormat(rb1.getString("greeting.common"),Locale.
US).format(params);                                                    ②
String str2 =new MessageFormat(rb2.getString("greeting.morning"),Locale.
CHINA).format(params);
String str3 =new MessageFormat(rb2.getString("greeting.afternoon"),Locale.
CHINA).format(params);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
```

运行以上代码，将输出以下信息：

```
How are you!John,today is 1/9/07 4:11 PM
早上好！John，现在是下午4:11
下午好！John，现在是2016年1月9日
```

6.5.2　MessageSource

Spring 定义了访问国际化信息的 MessageSource 接口，并提供了若干个易用的实现类。首先来了解一下该接口的几个重要方法。

- String getMessage(String code, Object[] args, String defaultMessage, Locale locale)：code 表示国际化信息中的属性名；args 用于传递格式化串占位符所用的运行期参数；当在资源中找不到对应的属性名时，返回 defaultMessage 参数指定的默认信息；locale 表示本地化对象。

- String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessage Exception：与上面的方法类似，只不过在找不到资源中对应的属性名时，直接抛出 NoSuchMessageException 异常。
- String getMessage(MessageSourceResolvable resolvable, Locale locale) throws NoSuchMessageException：MessageSourceResolvable 将属性名、参数数组及默认信息封装起来，它的功能和第一个接口方法相同。

1．MessageSource 的类结构

MessageSource 分别被 HierarchicalMessageSource 和 ApplicationContext 接口扩展，这里主要看一下 HierarchicalMessageSource 接口的几个实现类，如图 6-7 所示。

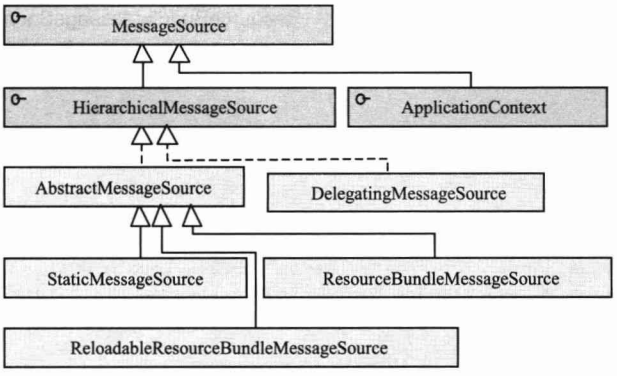


图 6-7　MessageSource 的类结构

HierarchicalMessageSource 接口添加了两个方法，建立父子层级的 MessageSource 结构，类似于前面介绍的 HierarchicalBeanFactory。该接口的 setParentMessageSource (MessageSource parent)方法用于设置父 MessageSource，而 getParentMessageSource()方法用于返回父 MessageSource。

HierarchicalMessageSource 接口最重要的两个实现类是 ResourceBundleMessageSource 和 ReloadableResourceBundleMessageSource。它们基于 Java 的 ResourceBundle 基础类实现，允许仅通过资源名加载国际化信息。ReloadableResourceBundleMessageSource 提供了定时刷新功能，允许在不重启系统的情况下更新资源的信息。StaticMessageSource 主要用于程序测试，允许通过编程的方式提供国际化信息。而 DelegatingMessageSource 是为方便操作父 MessageSource 而提供的代理类。

2．ResourceBundleMessageSource

该实现类允许用户通过 beanName 指定一个资源名（包括类路径的全限定资源名），或通过 beanNames 指定一组资源名。在代码清单 6-17 中通过 JDK 的基础类完成了本地化操作，下面使用 ResourceBundleMessageSource 来完成相同的任务，如代码清单 6-18 所示。读者可以比较两者的使用差别，并体会 Spring 所提供的国际化处理功能所带来的好处。

代码清单 6-18　通过ResourceBundleMessageSource配置资源

```
<bean id="myResource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
   <!--①通过基名指定资源，相对于类根路径-->
   <property name="basenames">
     <list>
       <value>com/smart/i18n/fmt_resource</value>
     </list>
   </property>
 </bean>
```

启动 Spring 容器，并通过 MessageSource 访问配置的国际化信息，如代码清单 6-19 所示。

代码清单 6-19　访问国际化信息：ResourceBundleMessageSource

```
String[] configs = {"com/smart/i18n/beans.xml"};
ApplicationContext ctx = new ClassPathXmlApplicationContext(configs);

//①获取 MessageSource 的 Bean
MessageSource ms = (MessageSource)ctx.getBean("myResource");
Object[] params = {"John", new GregorianCalendar(). getTime()};

//②获取格式化的国际化信息
String str1 = ms.getMessage("greeting.common",params,Locale.US);
String str2 = ms.getMessage("greeting.morning",params,Locale.CHINA);
String str3 = ms.getMessage("greeting.afternoon",params,Locale.CHINA);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
```

分析代码清单 6-19 中的代码，发现最主要的区别在于无须分别加载不同语言、不同国家/地区的本地化资源文件，仅仅通过资源名就可以加载整套国际化信息资源文件。此外，无须显式使用 MessageFormat 操作国际化信息，仅通过 MessageSource#getMessage()方法就可以完成操作。这段代码的运行结果与代码清单 6-17 的运行结果完全相同。

3. ReloadableResourceBundleMessageSource

前面提到该实现类相比于 ResourceBundleMessageSource 的唯一区别在于它可以定时刷新资源文件，以便在应用程序不重启的情况下感知资源文件的变化。很多生产系统都需要长时间地持续运行，系统重启会给运行带来很大的负面影响。这时，通过该实现类就可以解决国际化信息更新的问题。请看如代码清单 6-20 所示的配置。

代码清单 6-20　通过ReloadableResourceBundleMessageSource配置资源

```
<bean id="myResource"
      class="org.springframework.context.support.ReloadableResourceBundleMessage
Source">
   <property name="basenames">
     <list>
       <value>com/smart/i18n/fmt_resource</value>
     </list>
```

```
    </property>
    <property name="cacheSeconds" value="5"/> ①
  </bean>
```

刷新资源文件的周期，以秒为单位

在上面的配置中，通过 cacheSeconds 属性让 ReloadableResourceBundleMessageSource 每 5 秒钟刷新一次资源文件（在真实的应用中，刷新周期不能太短，否则频繁刷新将带来性能上的负面影响，一般建议不小于 1 分钟）。cacheSeconds 默认值为–1，表示永不刷新，此时，该实现类的功能就蜕化为 ResourceBundleMessageSource 的功能。

编写一个测试类对上面配置的 ReloadableResourceBundleMessageSource 进行测试，如代码清单 6-21 所示。

代码清单 6-21　刷新资源：ReloadableResourceBundleMessageSource

```
String[] configs = {"com/smart/i18n/beans.xml"};
ApplicationContext ctx = new ClassPathXmlApplicationContext(configs);

MessageSource ms = (MessageSource)ctx.getBean("myResource");
Object[] params = {"John", new GregorianCalendar().getTime()};

for (int i = 0; i < 2; i++) {
    String str1 = ms.getMessage("greeting.common",params,Locale.US);
    System.out.println(str1);
    Thread.currentThread().sleep(20000); ①
}
```

模拟程序应用，在此期间，更改资源文件

在①处让程序睡眠 20 秒，在此期间，将 fmt_resource_zh_CN.properties 资源文件的 greeting.common 键值调整为：

```
---How are you!{0},today is {1}---
```

可以看到两次输出的格式化信息分别对应更改前后的内容，即本地化资源文件的调整自动生效了。

```
How are you!John,today is 1/9/07 4:55 PM
---How are you!John,today is 1/9/07 4:55 PM---
```

6.5.3　容器级的国际化信息资源

在如图 6-7 所示的 MessageSource 类结构中，我们发现 ApplicationContext 实现了 MessageSource 的接口。也就是说，ApplicationContext 的实现类本身也是一个 MessageSource 对象。

将 ApplicationContext 和 MessageSource 整合起来，乍一看令人费解，Spring 这样设计的意图究竟是什么呢？原来 Spring 认为，在一般情况下，国际化信息资源应该是容器级的。一般不会将 MessageSource 作为一个 Bean 注入其他的 Bean 中，相反，MessageSource 作为容器的基础设施向容器中所有的 Bean 开放。只要考察一下国际化信息的实际消费场所，就更能理解 Spring 这样设计的用意了。国际化信息资源一般在系统输出信息时使用，如 Spring MVC 的页面标签、控制器（Controller）等，不同的模块都可能通过这些组件访问国际化信息资源，因此 Spring 将国际化信息资源作为容器的公共基础设施对所有组件开放。

既然一般情况下不会直接通过引用 MessageSource Bean 使用国际化信息资源，那么如何声明容器级的国际化信息资源呢？其实在 6.1.1 节讲解 Spring 容器的内部工作机制时已经埋下了伏笔：在介绍容器启动过程时，通过代码清单 6-1 对 Spring 容器启动时的步骤进行了剖析，④处的 initMessageSource()方法所执行的工作就是初始化容器中的国际化信息资源，它根据反射机制从 BeanDefinitionRegistry 中找出名为 **messageSource** 且类型为 **org.springframework.context.MessageSource** 的 Bean，将这个 Bean 定义的信息资源加载为容器级的国际化信息资源。请看如代码清单 6-22 所示的配置。

代码清单 6-22　容器级资源的配置

```
<!--①注册资源 Bean，其 Bean 名称只能为 messageSource -->
<bean id="messageSource"
     class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>com/smart/i18n/fmt_resource</value>
    </list>
  </property>
</bean>
```

下面通过 ApplicationContext 直接访问国际化信息资源，如代码清单 6-23 所示。

代码清单 6-23　通过ApplicationContext访问国际化信息资源

```
String[] configs = {"com/smart/i18n/beans.xml"};
ApplicationContext ctx = new ClassPathXmlApplicationContext(configs);
//①直接通过容器访问国际化信息资源
Object[] params = {"John", new GregorianCalendar().getTime()};

String str1 = ctx.getMessage("greeting.common",params,Locale.US);
String str2 = ctx.getMessage("greeting.morning",params,Locale.CHINA);
System.out.println(str1);
System.out.println(str2);
```

运行以上代码，输出以下信息：

```
How are you!John,today is 1/9/07 5:24 PM
早上好! John，现在是下午5:24
```

假设 MessageSource Bean 没有被命名为“messageSource”，那么以上代码将抛出 NoSuchMessageException 异常。

6.6　容器事件

Spring 的 ApplicationContext 能够发布事件并且允许注册相应的事件监听器，因此，它拥有一套完善的事件发布和监听机制。我们知道，Java 通过 java.util.EventObject 类和 java.util.EventListener 接口描述事件和监听器，某个组件或框架如需事件发布和监听机制，都需要通过扩展它们进行定义。在事件体系中，除了事件和监听器外，还有另外 3 个重要的概念。

- 事件源：事件的产生者，任何一个 EventObject 都必须拥有一个事件源。
- 事件监听器注册表：组件或框架的事件监听器不可能飘浮在空中，而必须有所依存。也就是说组件或框架必须提供一个地方保存事件监听器，这便是事件监听器注册表。一个事件监听器注册到组件或框架中，其实就是保存在事件监听器注册表中。当组件和框架中的事件源产生事件时，就会通知这些位于事件监听器注册表中的监听器。
- 事件广播器：它是事件和事件监听器沟通的桥梁，负责把事件通知给事件监听器。

通过图 6-8 可以看出这几个角色是如何各司其职的。

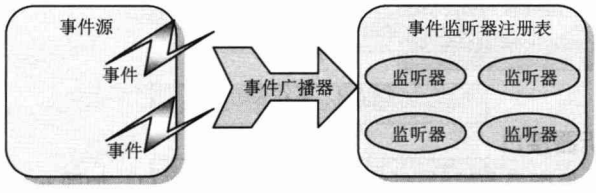


图 6-8　事件体系

事件源、事件监听器注册表和事件广播器这 3 个角色有时可以由同一个对象承担，如 java.swing 包中的 JButton、JCheckBox 等组件，它们分别集以上 3 个角色于一身。

在分析了事件体系后，我们会发现事件体系其实是观察者模式的一种具体实现方式，它并没有任何神秘之处。之所以组件或框架的事件会让一些开发者觉得神奇，就是因为组件或框架通过观察者模式很好地封装了事件模型并透明地提供给使用者，使用者只需按其设定的方式定义并注册事件监听器，事件体系就可以正常工作，因而我们很少会关注它的内部实现机理。

6.6.1　Spring 事件类结构

1. 事件类

首先来了解一下 Spring 的事件类结构。目前 Spring 框架本身仅定义了几个事件，如图 6-9 所示。

ApplicationEvent 的唯一构造函数是 ApplicationEvent(Object source)，通过 source 指定事件源，它有两个子类。

- ApplicationContextEvent：容器事件，它拥有 4 个子类，分别表示容器启动、刷新、停止及关闭的事件。
- RequestHandleEvent：这是一个与 Web 应用相关的事件，当一个 HTTP 请求被处理后，产生该事件。只有在 web.xml 中定义了 DispatcherServlet 时才会产生该事件。它拥有两个子类，分别代表 Servlet 及 Portlet 的请求事件。

也可以根据需要扩展 ApplicationEvent 定义自己的事件，完成其他特殊的功能。

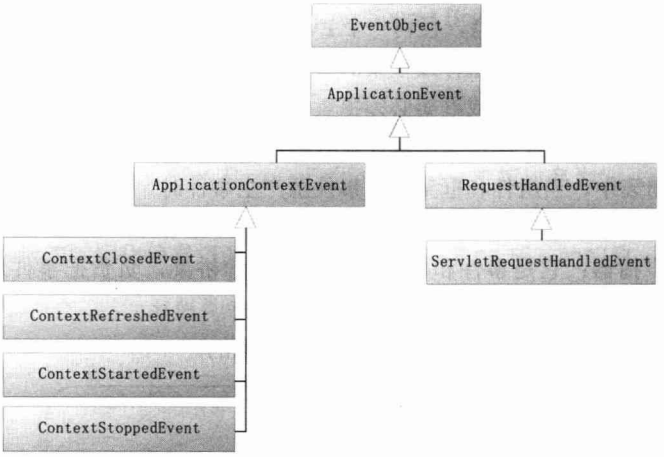


图 6-9　事件类结构

2. 事件监听器接口

Spring 的事件监听器都继承自 ApplicationListener 接口，如图 6-10 所示。

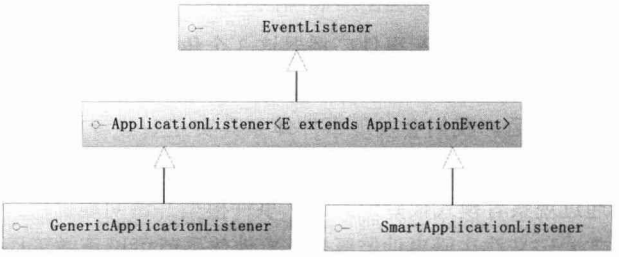


图 6-10　事件监听器接口

ApplicationListener 接口只定义了一个方法：onApplicationEvent(E event)，该方法接收 ApplicationEvent 事件对象，在该方法中编写事件的响应处理逻辑。而 SmartApplicationListener 接口是 Spring 3.0 新增的，它定义了两个方法。

- boolean supportsEventType(**Class<? extends ApplicationEvent> eventType**)：指定监听器支持哪种类型的容器事件，即它只会对该类型的事件做出响应。
- boolean supportsSourceType(Class<?> sourceType)：指定监听器仅对何种事件源对象做出响应。

其中，GenericApplicationListener 接口是 Spring 4.2 新增的。与 SmartApplicationListener 接口不同的是，它增强了对泛型事件类型的支持，supportsEventType()方法的参数不再仅限于 ApplicationEvent 子类型，而是采用可解析类型 ResolvableType。ResolvableType 是 Spring 4.0 提供的一个更加简单易用的泛型操作支持类。通过 ResolvableType 可以很容易地获取到泛型的实际类型信息，包括获取类级、字段级别、方法返回值、构造器参数及数组组件类型的泛型信息。在 Spring 4.0 框架中，很多核心类内部涉及的泛型操作都替换为 ResolvableType 类进行处理（如 BeanWrapperImpl、GenericTypeResolver 等）。

GenericApplicationListener 定义了两个方法。

- boolean supportsEventType(**ResolvableType eventType**)：指定监听器是否实际支持给定的事件类型，即它只会对该类型的事件做出响应。
- boolean supportsSourceType(Class<?> sourceType)：指定监听器仅对何种事件源对象做出响应。

3．事件广播器

当发生容器事件时，容器主控程序将调用事件广播器将事件通知给事件监听器注册表中的事件监听器，事件监听器分别对事件进行响应。Spring 为事件广播器定义了接口，并提供了实现类，如图 6-11 所示。

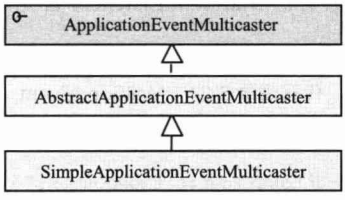


图 6-11　事件广播器类结构

6.6.2　解构 Spring 事件体系的具体实现

Spring 在 ApplicationContext 接口的抽象实现类 AbstractApplicationContext 中完成了事件体系的搭建。AbstractApplicationContext 拥有一个 applicationEventMulticaster 成员变量，applicationEventMulticaster 提供了容器监听器的注册表。AbstractApplicationContext 在 refresh()这个容器启动方法中通过以下 3 个步骤搭建了事件的基础设施。代码清单 6-1 中列出了 refresh()内部的整个过程，为了阅读方便，这里再次给出和事件体系有关的代码，如下：

```
//⑤初始化应用上下文事件广播器
initApplicationEventMulticaster();
...
//⑦注册事件监听器
registerListeners();
...
//⑨完成刷新并发布容器刷新事件
finishRefresh();
```

首先，在⑤处，Spring 初始化事件的广播器。用户可以在配置文件中为容器定义一个自定义的事件广播器，只要实现 ApplicationEventMulticaster 即可，Spring 会通过反射机制将其注册成容器的事件广播器。如果没有找到配置的外部事件广播器，则 Spring 自动使用 SimpleApplicationEventMulticaster 作为事件广播器。

在⑦处，Spring 根据反射机制，从 BeanDefinitionRegistry 中找出所有实现 org.springframework.context.ApplicationListener 的 Bean，将它们注册为容器的事件监听

器，实际操作就是将其添加到事件广播器所提供的事件监听器注册表中。

在⑨处，容器启动完成，调用事件发布接口向容器中所有的监听器发布事件。在publishEvent()内部可以看到，Spring 委托 ApplicationEventMulticaster 将事件通知给事件监听器。

6.6.3 一个实例

本节通过一个实例讲解事件发布和事件监听的整体过程。这个例子包括一个模拟的邮件发送器 MailSender，它在向目的地发送邮件时，将产生一个 MailSendEvent 事件，容器中注册了监听该事件的监听器 MailSendListener。首先来看一下 MailSendEvent 的代码，如代码清单 6-24 所示。

代码清单 6-24 MailSendEvent

```
package com.smart.event;

import org.springframework.context.ApplicationContext;
import org.springframework.context.event.ApplicationContextEvent;

public class MailSendEvent extends ApplicationContextEvent {
    private String to;

    public MailSendEvent(ApplicationContext source, String to) {
        super(source);
        this.to = to;
    }
    public String getTo() {

        return this.to;
    }
}
```

它直接扩展 ApplicationContextEvent，事件对象除 source 属性外，还具有一个代表发送目的地的 to 属性。

事件监听器 MailSenderListener 负责监听 MailSendEvent 事件，它的代码如下所示：

```
package com.smart.event;
import org.springframework.context.ApplicationListener;
public class MailSendListener implements ApplicationListener<MailSendEvent>{

  //①对MailSendEvent事件进行处理
    public void onApplicationEvent(MailSendEvent event) {
            MailSendEvent mse = (MailSendEvent) event;
            System.out.println("MailSendListener:向" + mse.getTo() + "发送完一封邮件");
    }
}
```

MailSenderListener 直接实现 ApplicationListener 接口，在接口方法中通过 instanceof 操作符判断事件的类型，仅对 MailSendEvent 类型的事件进行处理。

MailSender 要拥有发布事件的能力，就必须实现 ApplicationContextAware 接口，如下：

```
package com.smart.event;

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

public class MailSender implements ApplicationContextAware {
    private ApplicationContext ctx ;

   //①ApplicationContextAware的接口方法，以便容器启动时注入容器实例
    public void setApplicationContext(ApplicationContext ctx)
            throws BeansException {
        this.ctx = ctx;

    }
    public void sendMail(String to){
        System.out.println("MailSender:模拟发送邮件...");
        MailSendEvent mse = new MailSendEvent(this.ctx,to);
       //②向容器中的所有事件监听器发送事件
        ctx.publishEvent(mse);
    }
}
```

在 sendMail()方法中，首先模拟发送邮件，然后产生一个 MailSendEvent 事件，并通过容器句柄 ctx 向容器中的所有事件监听器发送事件。

在 Spring 的配置文件中，仅需进行如下配置：

```
<bean class="com.smart.event.MailSendListener"/>
<bean id="mailSender" class="com.smart.event.MailSender"/>
```

下面的代码启动容器并调用 mailSender Bean 发送一封邮件：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("com/smart/event/beans.xml");
MailSender mailSender = (MailSender)ctx.getBean("mailSender");
mailSender.sendMail("aaa@bbb.com");
```

当容器启动时，它会根据配置文件自动注册 MailSendListener 这个事件监听器。运行以上代码，输出以下信息：

```
MailSender:模拟发送邮件...
MailSendListener:向 aaa@bbb.com 发送完一封邮件
```

6.7　小结

在本章中，我们对 Spring 容器进行了深度剖析，不但分析了 Spring 容器的运行流程，还深入 Spring 的内部探究其实现机理，并介绍了组成 Spring 容器的重要组件。Spring 不但是一个优秀而实用的开发框架，该框架本身也是经典的程序设计范本，我们希望通过对其内部设计的深入分析，让读者从中吸取设计思想的精华并应用到自己的开发实践中去，而不仅仅使用 Spring 框架的功能。

此外，还介绍了 Spring 的属性编辑器、外部属性文件、国际化信息及容器事件的知识。在介绍这些知识前，我们学习了相关主题的 Java 知识，这些知识可以帮助读者加深对 Spring 相关知识的理解。

第 7 章

Spring AOP 基础

Spring AOP 是 AOP 技术在 Spring 中的具体实现，它是构成 Spring 框架的另一个重要基石。Spring AOP 构建于 IoC 之上，和 IoC “浑然天成”，统一于 Spring 容器之中。本章将从 Spring AOP 的底层实现技术入手，一步步深入 Spring AOP 的内核，分析它的底层结构和实现。

本章主要内容：

- AOP 概述
- Spring AOP 所涉及的 Java 基础知识
- Spring AOP 的增强类型
- Spring AOP 的切面类型
- 通过自动代理技术创建切面

本章亮点：

- 通过 Spring AOP 所涉及的底层 Java 知识的学习深刻理解 Spring AOP 的具体实现
- 深入 Spring AOP 的内核分析其组成和结构
- AOP 疑难问题剖析

7.1 AOP 概述

编程语言的终极目标就是能以更自然、更灵活的方式模拟世界，从原始机器语言到过程语言再到面向对象语言，编程语言一步步地用更自然、更灵活的方式编写软件。AOP 是软件开发思想发展到一定阶段的产物，但 AOP 的出现并不是要完全替代 OOP，而仅作为 OOP 的有益补充。虽然 AOP 作为一项编程技术已经有多年的历史，但长时间停留在学术领域，直到近几年，AOP 才作为一项真正的实用技术在应用领域开疆拓土。需要

指出的是，AOP 是有特定的应用场合的，它只适合那些具有横切逻辑的应用场合，如性能监测、访问控制、事务管理及日志记录（虽然有很多文章用日志记录作为讲解 AOP 的实例，但很多人认为很难用 AOP 编写实用的程序日志，笔者对此观点非常认同）。不过，这丝毫不影响 AOP 作为一种新的软件开发思想在软件开发领域所占有的地位。

7.1.1　AOP 到底是什么

AOP 是 Aspect Oriented Programing 的简称，最初被译为“面向方面编程”，这个翻译向来为人所诟病，但是由于先入为主的效应，受众广泛，所以这个翻译依然被很多人使用。但我们更倾向于用“面向切面编程”的译法，因为它更加达意。

按照软件重构思想的理念，如果多个类中出现相同的代码，则应该考虑定义一个父类，将这些相同的代码提取到父类中。比如 Horse、Pig、Camel 这些对象都有 run()和 eat()方法，通过引入一个包含这两个方法的抽象的 Animal 父类，Horse、Pig、Camel 就可以通过继承 Animal 复用 run()和 eat()方法。通过引入父类消除多个类中重复代码的方式在大多数情况下是可行的，但世界并非永远这样简单，请看如代码清单 7-1 所示的论坛管理业务类的代码。

代码清单 7-1　ForumService

```
package com.smart.concept;
public class ForumService {
    private TransactionManager transManager;
    private PerformanceMonitor pmonitor;
    private TopicDao topicDao;
    private ForumDao forumDao;

    public void removeTopic(int topicId) {
        pmonitor.start();
        transManager.beginTransaction();
        topicDao.removeTopic(topicId); //①
        transManager.commit();
        pmonitor.end();
    }
    public void createForum(Forum forum) {
        pmonitor.start();
        transManager.beginTransaction();
        forumDao.create(forum); //②
        transManager.commit();
        pmonitor.end();
    }
    ...
}
```

代码清单 7-1 中斜体的代码是方法性能监视代码，它在方法调用前启动，在方法调用返回前结束，并在内部记录性能监视的结果信息。而黑色粗体的代码是事务开始和事务提交的代码。我们发现①、②处的业务代码淹没在重复化非业务性的代码之中，性能

监视和事务管理这些非业务性代码葛藤缠树般包围着业务性代码。

如图 7-1 所示，假设将 ForumService 业务类看成一段圆木，将 removeTopic()和 createForum()方法分别看成圆木的一截，会发现性能监视和事务管理的代码就好像一个年轮，而业务代码是圆木的树心，这也正是横切代码概念的由来。

图 7-1　横切逻辑示意图

我们无法通过抽象父类的方式消除如上所示的重复性横切代码，因为这些横切逻辑依附在业务类方法的流程中，它们不能转移到其他地方去。

AOP 独辟蹊径，通过横向抽取机制为这类无法通过纵向继承体系进行抽象的重复性代码提供了解决方案。对于习惯了纵向抽取的开发者来说，可能不太容易理解横向抽取方法的工作机制，因为 Java 语言本身不直接提供这种横向抽取的能力。暂把具体实现放在一旁，先通过图解的方式归纳出 AOP 的解决思路，如图 7-2 所示。

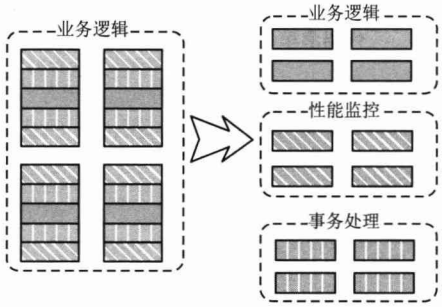


图 7-2　横向抽取

从图 7-2 中可以看出，AOP 希望将这些分散在各个业务逻辑代码中的相同代码通过横向切割的方式抽取到一个独立的模块中，还业务逻辑类一个清新的世界。

当然，将这些重复性的横切逻辑独立出来是很容易的，但如何将这些独立的逻辑融合到业务逻辑中以完成和原来一样的业务流程，才是事情的关键，这也正是 AOP 要解决的主要问题。

轻松一刻

现在常用“雁过拔毛”来形容某人爱贪便宜，对每件经手的事情都要获得一些好处。“雁过拔毛”就是现实生活中 AOP 的一个很形象的例子。其实“雁过拔毛”的原意是形容武艺高超，大雁飞过时也能拔下它的毛来。

7.1.2　AOP 术语

如学习电学要先学习电阻、电压、电容等专业术语一样，AOP 也有一些自己的行话。为了方便后面的学习，先来了解一下 AOP 的几个重要术语。

1．连接点（Joinpoint）

特定点是程序执行的某个特定位置，如类开始初始化前、类初始化后、类的某个方法调用前/调用后、方法抛出异常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些代码中的特定点就被称为“连接点”。Spring 仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时及方法调用前后这些程序执行点织入增强。我们知道，黑客攻击系统需要找到突破口，没有突破口就无法进行攻击。从某种程度上来说，AOP 也可以看成一个黑客（因为它要向目前类中嵌入额外的代码逻辑），连接点就是 AOP 向目标类打入楔子的候选锚点。

连接点由两个信息确定：一是用方法表示的程序执行点；二是用相对位置表示的方位。如在 Test.foo()方法执行前的连接点，执行点为 Test.foo()，方位为该方法执行前的位置。Spring 使用切点对执行点进行定位，而方位则在增强类型中定义。

2．切点（Pointcut）

每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序类中客观存在的事物。但在为数众多的连接点中，如何定位某些感兴趣的连接点呢？AOP 通过“切点”定位特定的接连点。借助数据库查询的概念来理解切点和连接点的关系再合适不过了：连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。

在 Spring 中，切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件，Spring AOP 的规则解析引擎负责解析切点所设定的查询条件，找到对应的连接点。确切地说，应该是执行点而非连接点，因为连接点是方法执行前、执行后等包括方位信息的具体程序执行点，而切点只定位到某个方法上，所以如果希望定位到具体的连接点上，还需要提供方位信息。

3．增强（Advice）

增强是织入目标类连接点上的一段程序代码，是不是觉得 AOP 越来越像黑客了，这不是往业务类中装入木马吗？我们大可按照这一思路去理解增强，因为这样更形象易懂。在 Spring 中，增强除用于描述一段程序代码外，还拥有另一个和连接点相关的信息，这便是执行点的方位。结合执行点的方位信息和切点信息，就可以找到特定的连接。正因为增强既包含用于添加到目标连接点上的一段执行逻辑，又包含用于定位连接点的方位信息，所以 Spring 所提供的增强接口都是带方位名的，如 BeforeAdvice、AfterReturningAdvice、ThrowsAdvice 等。BeforeAdvice 表示方法调用前的位置，而 AfterReturningAdvice 表示访问返回后的位置。所以只有结合切点和增强，才能确定特定的连接点并实施增强逻辑。

有很多书籍和文章将 Advice 译为“通知”，就像将“how old are you?”译为“怎么老是你”一样，明显是一种“望文生义”的译法。来看几个使用“通知”的语境：银行向张三发出了一个催款通知；班主任通知学生明天大扫除。从这些语境中可以知道，通知者只是把某个消息传达给被通知者，并不会替被通知者做任何事情。而 Spring 的 Advice 必须嵌入类的某连接点上，并完成一段附加的执行逻辑，这明显是去“增强”目标类的功能。当然，我们不能对这个翻译有过多的微词，毕竟 Advice 这个英文单词本身就有些不知所云，如果将其改为 Enhancer，相信理解起来会更容易一些。

轻松一刻

早期国外普遍采用韦氏拼音来拼写中国的人名、地名，一些译者在翻译国外作品时由于未注意到这一背景，闹出了不少笑话，如将蒋介石（Chiang Kai-shek）译为常凯申，将孟子（Mencius）译为门修斯，这些留洋归来的名人着实“洋气”了一把。

4. 目标对象（Target）

增强逻辑的织入目标类。如果没有 AOP，那么目标业务类需要自己实现所有的逻辑，就如代码清单 7-1 中的 ForumService 所示。在 AOP 的帮助下，ForumService 只实现那些非横切逻辑的程序逻辑，而性能监视和事务管理等这些横切逻辑则可以使用 AOP 动态织入特定的连接点上。

5. 引介（Introduction）

引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过 AOP 的引介功能，也可以动态地为该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。

6. 织入（Weaving）

织入是将增强添加到目标类的具体连接点上的过程。AOP 就像一台织布机，将目标类、增强或者引介天衣无缝地编织到一起。我们不能不说“织入”这个词太精辟了。根据不同的实现技术，AOP 有 3 种织入方式。

（1）编译期织入，这要求使用特殊的 Java 编译器。

（2）类装载期织入，这要求使用特殊的类装载器。

（3）动态代理织入，在运行期为目标类添加增强生成子类的方式。

Spring 采用动态代理织入，而 AspectJ 采用编译期织入和类装载期织入。

7. 代理（Proxy）

一个类被 AOP 织入增强后，就产生了一个结果类，它是融合了原类和增强逻辑的代理类。根据不同的代理方式，代理类既可能是和原类具有相同接口的类，也可能就是原类的子类，所以可以采用与调用原类相同的方式调用代理类。

8．切面（Aspect）

切面由切点和增强（引介）组成，它既包括横切逻辑的定义，也包括连接点的定义。Spring AOP 就是负责实施切面的框架，它将切面所定义的横切逻辑织入切面所指定的连接点中。

AOP 的工作重心在于如何将增强应用于目标对象的连接点上。这里包括两项工作：第一，如何通过切点和增强定位到连接点上；第二，如何在增强中编写切面的代码。本章大部分内容都将围绕这两点展开。

7.1.3　AOP 的实现者

AOP 工具的设计目标是把横切的问题（如性能监视、事务管理）模块化。使用类似 OOP 的方式进行切面的编程工作。位于 AOP 工具核心的是连接点模型，它提供了一种机制，可以定位到需要在哪里发生横切。

1．AspectJ

AspectJ 是语言级的 AOP 实现，2001 年由 Xerox PARC 的 AOP 小组发布，目前版本已经更新到 1.8.9。AspectJ 扩展了 Java 语言，定义了 AOP 语法，能够在编译期提供横切代码的织入，所以它有一个专门的编译器用来生成遵守 Java 字节编码规范的 Class 文件。其主页位于 http://www.eclipse.org/aspectj。

2．AspectWerkz

AspectWerkz 是基于 Java 的简单、动态、轻量级的 AOP 框架，该框架于 2002 年发布，由 BEA Systems 提供支持。它支持运行期或类装载期织入横切代码，所以它拥有一个特殊的类装载器。现在，AspectJ 和 AspectWerkz 项目已经合并，以便整合二者的力量和技术创建统一的 AOP 平台。它们合作的第一个发布版本是 AspectJ 5：扩展 AspectJ 语言，以基于注解的方式支持类似 AspectJ 的代码风格。

3．JBoss AOP

JBoss AOP 于 2004 年作为 JBoss 应用程序服务器框架的扩展功能发布，读者可以从以下地址了解到 JBoss AOP 的更多信息：http://www.jboss.org/products/aop。

4．Spring AOP

Spring AOP 使用纯 Java 实现，它不需要专门的编译过程，也不需要特殊的类装载器，它在运行期通过代理方式向目标类织入增强代码。Spring 并不尝试提供最完整的 AOP 实现，相反，它侧重于提供一种和 Spring IoC 容器整合的 AOP 实现，用以解决企业级开发中的常见问题。在 Spring 中可以无缝地将 Spring AOP、IoC 和 AspectJ 整合在一起。

7.2 基础知识

Spring AOP 使用动态代理技术在运行期织入增强的代码，为了揭示 Spring AOP 底层的工作机理，有必要学习涉及的 Java 知识。Spring AOP 使用了两种代理机制：一种是基于 JDK 的动态代理；另一种是基于 CGLib 的动态代理。之所以需要两种代理机制，很大程度上是因为 JDK 本身只提供接口的代理，而不支持类的代理。

7.2.1 带有横切逻辑的实例

下面通过具体化代码实现 7.1 节所介绍的例子的性能监视横切逻辑，并通过动态代理技术对此进行改造。在调用每一个目标类方法时启动方法的性能监视，在目标类方法调用完成时记录方法的花费时间，如代码清单 7-2 所示。

代码清单 7-2 ForumService：包含性能监视横切代码

```
package com.smart.proxy;
public class ForumServiceImpl implements ForumService {
    public void removeTopic(int topicId) {


        //①-1 开始对该方法进行性能监视
        PerformanceMonitor.begin(
                        "com.smart.proxy.ForumServiceImpl. removeTopic");
        System.out.println("模拟删除Topic记录:"+topicId);
        try {
            Thread.currentThread().sleep(20);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

       //①-2 结束对该方法的性能监视
        PerformanceMonitor.end();
    }

    public void removeForum(int forumId) {

        //②-1 开始对该方法进行性能监视
        PerformanceMonitor.begin(
                        "com.smart.proxy.ForumServiceImpl. removeForum");
        System.out.println("模拟删除Forum记录:"+forumId);
        try {
            Thread.currentThread().sleep(40);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }


       //②-2 结束对该方法的性能监视
```

```
        PerformanceMonitor.end();
    }
}
```

在代码清单 7-2 中，粗体表示的代码就是具有横切逻辑特征的代码，每个 Service 类和每个业务方法体的前后都执行相同的代码逻辑：方法调用前启动 PerformanceMonitor；方法调用后通知 PerformanceMonitor 结束性能监视并记录性能监视结果。

PerformanceMonitor 是性能监视的实现类，下面给出一个非常简单的实现版本，如代码清单 7-3 所示。

代码清单 7-3　PerformanceMonitor

```
package com.smart.proxy;
public class PerformanceMonitor {

    //①通过一个 ThreadLocal 保存与调用线程相关的性能监视信息
    private static ThreadLocal<MethodPerformace> performanceRecord =
                                    new ThreadLocal<MethodPerformance>();

    //②启动对某一目标方法的性能监视
    public static void begin(String method) {
        System.out.println("begin monitor...");
        MethodPerformance mp = new MethodPerformance(method);
        performanceRecord.set(mp);
    }
    public static void end() {
        System.out.println("end monitor...");
        MethodPerformance mp = performanceRecord.get();

        //③打印出方法性能监视的结果信息
        mp.printPerformance();
    }
}
```

ThreadLocal 是将非线程安全类改造为线程安全类的“法宝”（11.2 节将详细介绍）。PerformanceMonitor 提供了两个方法：通过调用 begin(String method)方法开始对某个目标类方法的监视，其中 method 为目标类方法的全限定名；而通过调用 end()方法结束对目标类方法的监视，并给出性能监视信息。这两个方法必须配套使用。

用于记录性能监视信息的 MethodPerformance 类的代码如代码清单 7-4 所示。

代码清单 7-4　MethodPerformance

```
package com.smart.proxy;
public class MethodPerformance {
    private long begin;
    private long end;
    private String serviceMethod;
    public MethodPerformance(String serviceMethod){
      this.serviceMethod = serviceMethod;
      this.begin = System.currentTimeMillis(); ①
    }
    public void printPerformance(){
      end = System.currentTimeMillis(); ②
      long elapse = end - begin;
```

记录目标类方法开始执行点的系统时间

获取目标类方法执行完成后的系统时间，进而计算出目标类方法的执行时间

```
        System.out.println(serviceMethod+"花费"+elapse+"毫秒。"); ③
    }                                                    报告目标类方法的执行时间
}
```

通过下面的代码测试拥有性能监视能力的 ForumServiceImpl 业务方法：

```
package com.smart.proxy;
public class TestForumService {
    public static void main(String[] args) {
        ForumService forumService = new ForumServiceImpl();
        forumService .removeForum(10);
        forumService .removeTopic(1012);
    }
}
```

得到以下输出信息：

```
begin monitor...
模拟删除Forum记录:10
end monitor...
com.smart.proxy.ForumServiceImpl.removeForum花费47毫秒。

begin monitor...
模拟删除Topic记录:1012
end monitor...
com.smart.proxy.ForumServiceImpl.removeTopic花费26毫秒。
```

正如代码清单 7-2 所示，当某个方法需要进行性能监视时，必须调整方法代码，在方法体前后分别添加开启性能监视和结束性能监视的代码。这些非业务逻辑的性能监视代码破坏了 ForumServiceImpl 业务逻辑的纯粹性。我们希望通过代理的方式将业务类方法中开启和结束性能监视的横切代码从业务类中完全移除，并通过 JDK 或 CGLib 动态代理技术将横切代码动态织入目标方法的相应位置。

7.2.2 JDK 动态代理

自 Java 1.3 以后，Java 提供了动态代理技术，允许开发者在运行期创建接口的代理实例。在 Sun 刚推出动态代理时，还很难想象它有多大的实际用途，现在终于发现动态代理是实现 AOP 的绝好底层技术。

JDK 的动态代理主要涉及 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。其中，InvocationHandler 是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态地将横切逻辑和业务逻辑编织在一起。

而 Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。这样描述一定很抽象，我们马上着手使用 Proxy 和 InvocationHandler 这两个“魔法戒”对 7.2.1 节中的性能监视代码进行革新。

首先从业务类 ForumServiceImpl 中移除性能监视的横切代码，使 ForumServiceImpl 只负责具体的业务逻辑，如代码清单 7-5 所示。

代码清单 7-5　ForumServiceImpl：移除性能监视横切代码

```
package com.smart.proxy;

public class ForumServiceImpl implements ForumService {

    public void removeTopic(int topicId) {
        //PerformanceMonitor.begin(…) ①
        System.out.println("模拟删除Topic记录:"+topicId);
        try {
            Thread.currentThread().sleep(20);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        //PerformanceMonitor.end();①
    }
    public void removeForum(int forumId) {
        //PerformanceMonitor.begin(…)②
        System.out.println("模拟删除Forum记录:"+forumId);
        try {
            Thread.currentThread().sleep(40);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
       //PerformanceMonitor.end();②
    }
}
```

在此位置原有的横切代码被移除（抽取切面中）

在此位置原有的横切代码被移除（抽取切面中）

在代码清单 7-5 中的①和②处，原来的性能监视代码被移除了，只保留了真正的业务逻辑。

从业务类中移除性能监视横切代码后，必须为它找到一个安身之所，InvocationHandler 就是横切代码的“安家乐园”。将性能监视横切代码安置在 PerformanceHandler 中，如代码清单 7-6 所示。

代码清单 7-6　PerformanceHandler

```
package com.smart.proxy;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class PerformanceHandler implements InvocationHandler { //①
   private Object target;
    public PerformanceHandler(Object target){//②
        this.target = target;
    }
    public Object invoke(Object proxy, Method method, Object[] args) //③
            throws Throwable {
        PerformanceMonitor.begin(
                    target.getClass().getName()+"."+ method. getName());//③-1
        Object obj = method.invoke(target, args); //③-2
        PerformanceMonitor.end();//③-1
        return obj;
    }
}
```

实现 InvocationHandler

target 为目标业务类

通过反射方法调用业务类的目标方法

③处 invoke()方法中粗体所示部分的代码为性能监视的横切代码，我们发现，横切代码只出现一次，而不是像原来那样散落各处。③-2 处的 method.invoke()语句通过 Java 反射机制间接调用目标对象的方法，这样 InvocationHandler 的 invoke()方法就将横切逻辑代码（③-1）和业务类方法的业务逻辑代码（③-2）编织到一起，所以，可以将 InvocationHandler 看成一个编织器。下面对这段代码作进一步的说明。

首先实现 InvocationHandler 接口，该接口定义了一个 invoke(Object proxy, Method method, Object[] args)方法，其中，proxy 是最终生成的代理实例，一般不会用到；method 是被代理目标实例的某个具体方法，通过它可以发起目标实例方法的反射调用；args 是被代理实例某个方法的入参，在方法反射调用时使用。

其次，在构造函数里通过 target 传入希望被代理的目标对象，如②处所示；在 InvocationHandler 接口方法 invoke(Object proxy, Method method, Object[] args)里，将目标实例传递给 method.invoke()方法，并调用目标实例的方法，如③处所示。

下面通过 Proxy 结合 PerformanceHandler 创建 ForumService 接口的代理实例，如代码清单 7-7 所示。

代码清单 7-7　ForumServiceTest：创建代理实例

```
package com.smart.proxy;
import java.lang.reflect.Proxy;
import org.testng.annotations.*;
public class ForumServiceTest {
    @Test
    public void proxy() {
        ForumService target = new ForumServiceImpl(); //①

        PerformanceHandler handler = new PerformanceHandler(target); //②

        ForumService proxy = (ForumService) Proxy.newProxyInstance( //③
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                handler);

        proxy.removeForum(10); //④
        proxy.removeTopic(1012);
    }
}
```

希望被代理的目标业务类

将目标业务类和横切代码编织到一起

根据编织了目标业务类逻辑和性能监视横切逻辑的 InvocationHandler 实例创建代理实例

调用代理实例

上面的代码完成了业务类代码和横切代码的编织工作并生成了代理实例。在②处，让 PerformanceHandler 将性能监视横切逻辑编织到 ForumService 实例中，然后在③处，通过 Proxy 的 newProxyInstance()静态方法为编织了业务类逻辑和性能监视逻辑的 handler 创建一个符合 ForumService 接口的代理实例。该方法的第一个入参为类加载器；第二个入参为创建代理实例所需实现的一组接口；第三个入参是整合了业务逻辑和横切逻辑的编织器对象。

按照③处的设置方式，这个代理实例实现了目标业务类的所有接口，即 Forum

ServiceImpl 的 ForumService 接口。这样就可以按照调用 ForumService 接口实例相同的方式调用代理实例，如④处所示。运行以上代码，输出以下信息：

```
begin monitor...
模拟删除Forum记录:10
end monitor...
com.smart.proxy.ForumServiceImpl.removeForum花费47毫秒。

begin monitor...
模拟删除Topic记录:1012
end monitor...
com.smart.proxy.ForumServiceImpl.removeTopic花费26毫秒。
```

我们发现，程序的运行效果和直接在业务类中编写性能监视逻辑的效果一致，但在这里，原来分散的横切逻辑代码已经被抽取到 PerformanceHandler 中。当其他业务类（如 UserService、SystemService 等）的业务方法也需要使用性能监视时，只要按照与代码清单 7-7 相似的方式分别为它们创建代理对象即可。下面通过时序图描述通过创建代理对象进行业务方法调用的整体逻辑，以进一步认识代理对象的本质，如图 7-3 所示。

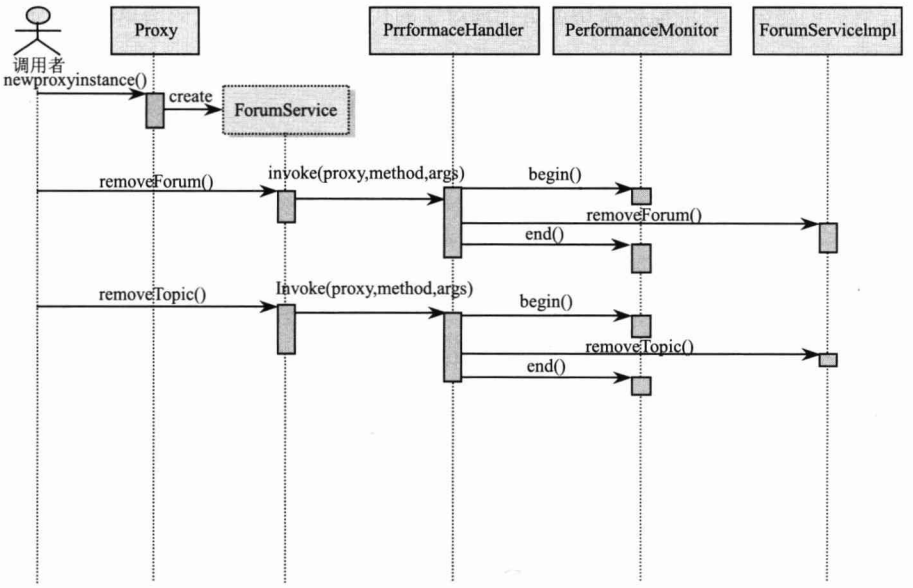


图 7-3　代理实例方法调用的时序图

在图 7-3 中使用虚线的方式对通过 Proxy 创建的 ForumService 代理实例加以突显，ForumService 代理实例内部利用 PerformaceHandler 整合横切逻辑和业务逻辑。调用者调用代理对象的 removeForum()和 removeTopic()方法时，图 7-3 所示的内部调用时序清晰地告诉我们实际上后台所发生的一切。

7.2.3 CGLib 动态代理

使用 JDK 创建代理有一个限制，即它只能为接口创建代理实例，这一点可以从 Proxy 的接口方法 newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h) 中看得很清楚：第二个入参 interfaces 就是需要代理实例实现的接口列表。虽然面向接口编程的思想被很多大师级人物（包括 Rod Johnson）所推崇，但在实际开发中，许多开发者也对此深感困惑：难道对一个简单业务表的操作也需要老老实实地创建 5 个类（领域对象类、DAO 接口、DAO 实现类、Service 接口和 Service 实现类）吗？难道不能直接通过实现类构建程序吗？对于这个问题，很难给出一个孰优孰劣的准确判断，但仍有很多不使用接口的项目也取得了非常好的效果。

对于没有通过接口定义业务方法的类，如何动态创建代理实例呢？JDK 动态代理技术显然已经黔驴技穷，CGLib 作为一个替代者，填补了这项空缺。

CGLib 采用底层的字节码技术，可以为一个类创建子类，在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势织入横切逻辑。下面采用 CGLib 技术编写一个可以为任何类创建织入性能监视横切逻辑代理对象的代理创建器，如代码清单 7-8 所示。

代码清单 7-8 CglibProxy

```
package com.smart.proxy;
import java.lang.reflect.Method;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class CglibProxy implements MethodInterceptor {
    private Enhancer enhancer = new Enhancer();
    public Object getProxy(Class clazz) {                    设置需要创建子类的类
        enhancer.setSuperclass(clazz);//①
        enhancer.setCallback(this);                          通过字节码技术动态
        return enhancer.create();//②                         创建子类实例
    }                                                                    拦截父类所有
    public Object intercept(Object obj, Method method, Object[] args, //③   方法的调用
            MethodProxy proxy) throws Throwable {
        PerformanceMonitor.begin(obj.getClass().getName()+"."+method. getName());//③-1
        Object result=proxy.invokeSuper(obj, args); //③-2
        PerformanceMonitor.end();//③-1                                   通过代理类调用
        return result;                                                   父类中的方法
    }
}
```

在上面的代码中，用户可以通过 getProxy(Class clazz)方法为一个类创建动态代理对象，该代理对象通过扩展 clazz 实现代理。在这个代理对象中，织入性能监视的横切逻辑（粗体部分）。intercept(Object obj, Method method, Object[] args,MethodProxy proxy)是 CGLib 定义的 Interceptor 接口方法，它拦截所有目标类方法的调用。其中，obj 表示目标类的实例；method 为目标类方法的反射对象；args 为方法的动态入参；proxy 为代理类实例。

下面通过 CglibProxy 为 ForumServiceImpl 类创建代理对象，并测试代理对象的方法，如代码清单 7-9 所示。

代码清单 7-9　ForumServiceTest：测试CGLib创建的代理类

```
package com.smart.proxy;
import java.lang.reflect.Proxy;
import org.testng.annotations.*;
public class ForumServiceTest {
    @Test
    public void proxy() {
      CglibProxy proxy = new CglibProxy();
      ForumServiceImpl forumService = //①
              (ForumServiceImpl )proxy.getProxy(ForumServiceImpl.class);
      forumService.removeForum(10);
      forumService.removeTopic(1023);
    }
}
```

通过动态生成子类的方式创建代理类

在①处通过 CglibProxy 为 ForumServiceImpl 动态创建了一个织入性能监视逻辑的代理对象，并调用代理类的业务方法。运行上面的代码，输出以下信息：

```
begin monitor...
模拟删除 Forum 记录:10
end monitor...
com.smart.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0.removeForum 花费 47 毫秒。
begin monitor...
模拟删除 Topic 记录:1023
end monitor...
com.smart.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0.removeTopic 花费 16 毫秒。
```

观察以上输出，除了发现两个业务方法中都织入了性能监控的逻辑外，还发现代理类的名字变成 com.smart.proxy.ForumServiceImpl$$EnhancerByCGLIB$$ 2a9199c0，这个特殊的类就是 CGLib 为 ForumServiceImpl 动态创建的子类。

值得一提的是，由于 CGLib 采用动态创建子类的方式生成代理对象，所以不能对目标类中的 final 或 private 方法进行代理。

7.2.4　AOP 联盟

AOP 联盟（http://aopalliance.sourceforge.net）是众多开源 AOP 项目的联合组织，该组织的目的是为了制定一套规范描述 AOP 的标准，定义标准的 AOP 接口，以便各种遵守标准的具体实现可以相互调用。

这种标准的制定本应当由 Sun 来做，但是因为 Sun 运作迟缓，AOP 联盟便捷足先登，而且它的影响力越来越大。现在大部分的 AOP 实现都采用 AOP 联盟的标准，所以 AOP 联盟制定的规范已经成为事实上的标准。

7.2.5 代理知识小结

Spring AOP 的底层就是通过使用 JDK 或 CGLib 动态代理技术为目标 Bean 织入横切逻辑的。这里对动态创建代理对象作一个小结。

虽然通过 PerformanceHandler 或 CglibProxy 实现了性能监视横切逻辑的动态织入，但这种实现方式存在 3 个明显需要改进的地方。

（1）目标类的所有方法都添加了性能监视横切逻辑，而有时这并不是我们所期望的，我们可能只希望对业务类中的某些特定方法添加横切逻辑。

（2）通过硬编码的方式指定了织入横切逻辑的织入点，即在目标类业务方法的开始和结束前织入代码。

（3）手工编写代理实例的创建过程，在为不同类创建代理时，需要分别编写相应的创建代码，无法做到通用。

以上 3 个问题在 AOP 中占用重要的地位，因为 Spring AOP 的主要工作就是围绕以上 3 点展开的：Spring AOP 通过 Pointcut（切点）指定在哪些类的哪些方法上织入横切逻辑，通过 Advice（增强）描述横切逻辑和方法的具体织入点（方法前、方法后、方法的两端等）。此外，Spring 通过 Advisor（切面）将 Pointcut 和 Advice 组装起来。有了 Advisor 的信息，Spring 就可以利用 JDK 或 CGLib 动态代理技术采用统一的方式为目标 Bean 创建织入切面的代理对象了。

JDK 动态代理所创建的代理对象，在 Java 1.3 下，性能差强人意。虽然在高版本的 JDK 中动态代理对象的性能得到了很大的提高，但有研究表明，CGLib 所创建的动态代理对象的性能依旧比 JDK 所创建的动态代理对象的性能高不少（大概 10 倍）。但 CGLib 在创建代理对象时所花费的时间却比 JDK 动态代理多（大概 8 倍）。对于 singleton 的代理对象或者具有实例池的代理，因为无须频繁地创建代理对象，所以比较适合采用 CGLib 动态代理技术；反之则适合采用 JDK 动态代理技术。

7.3 创建增强类

Spring 使用增强类定义横切逻辑，同时由于 Spring 只支持方法连接点，增强还包括在方法的哪一点加入横切代码的方位信息，所以增强既包含横切逻辑，又包含部分连接点的信息。

7.3.1 增强类型

AOP 联盟为增强定义了 org.aopalliance.aop.Advice 接口，Spring 支持 5 种类型的增强，先来了解一下增强接口继承关系图，如图 7-4 所示。

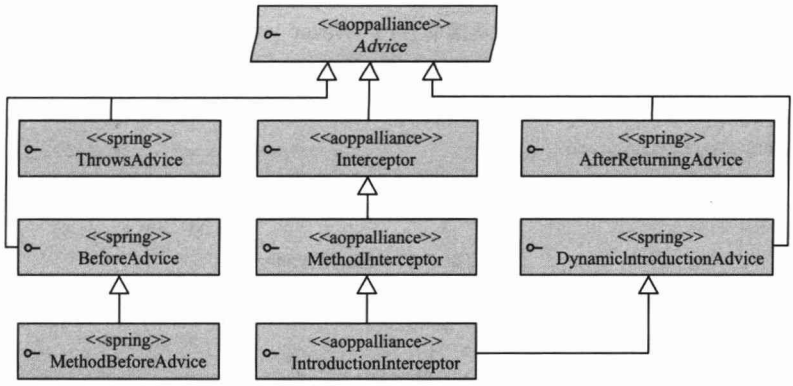


图 7-4　增强接口继承关系图

带<<spring>>标识的接口是 Spring 所定义的扩展增强接口；带<<aoppalliance>>标识的接口则是 AOP 联盟定义的接口。按照增强在目标类方法中的连接点位置，可以分为以下 5 类。

- 前置增强：org.springframework.aop.BeforeAdvice 代表前置增强。因为 Spring 只支持方法级的增强，所以 MethodBeforeAdvice 是目前可用的前置增强，表示在目标方法执行前实施增强，而 BeforeAdvice 是为了将来版本扩展需要而定义的。
- 后置增强：org.springframework.aop.AfterReturningAdvice 代表后置增强，表示在目标方法执行后实施增强。
- 环绕增强：org.aopalliance.intercept.MethodInterceptor 代表环绕增强，表示在目标方法执行前后实施增强。
- 异常抛出增强：org.springframework.aop.ThrowsAdvice 代表抛出异常增强，表示在目标方法抛出异常后实施增强。
- 引介增强：org.springframework.aop.IntroductionInterceptor 代表引介增强，表示在目标类中添加一些新的方法和属性。

这些增强接口都有一些方法，通过实现这些接口方法，并在接口方法中定义横切逻辑，就可以将它们织入目标类方法的相应连接点位置。

7.3.2　前置增强

“热情待客、礼貌服务”已经成为服务行业的基本经营理念，下面通过前置增强对服务生的服务用语进行强制规范。假设服务生只做两件事：第一，欢迎顾客；第二，对顾客提供服务。

1．保证使用礼貌用语的实例

来看一个保证使用礼貌用语的实例，如代码清单 7-10 所示。

代码清单 7-10 Waiter

```
package com.smart.advice;
public interface Waiter {
  void greetTo(String name);
  void serveTo(String name);
}
```

现在来看一个训练不足的服务生的服务情况，如代码清单 7-11 所示。

代码清单 7-11 NaiveWaiter

```
package com.smart.advice;
public class NaiveWaiter implements Waiter {
    public void greetTo(String name) {
        System.out.println("greet to "+name+"...");
    }
    public void serveTo(String name){
        System.out.println("serving "+name+"...");
    }
}
```

NaiveWaiter 只是简单地向顾客打招呼，闷不作声地走到顾客跟前，直接提供服务。下面对 NaiveWaiter 的服务行为进行规范，让他们在打招呼和提供服务之前，必须先对顾客使用礼貌用语，如代码清单 7-12 所示。

代码清单 7-12 GreetingBeforeAdvice

```
package com.smart.advice;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class GreetingBeforeAdvice implements MethodBeforeAdvice {

    public void before(Method method, Object[] args, Object obj) throws Throwable {//①
        String clientName = (String)args[0];
        System.out.println("How are you! Mr."+clientName+".");
    }
}
```

在目标类方法调用前执行

BeforeAdvice 是前置增强的接口，方法前置增强的 MethodBeforeAdvice 接口是其子类。Spring 目前只提供方法调用的前置增强，在以后的版本中可能会看到 Spring 提供的其他类型的前置增强，这正是 BeforeAdvice 接口存在的意义。MethodBeforeAdvice 接口仅定义了唯一的方法：before(Method method, Object[] args, Object obj) throws Throwable。其中，method 为目标类的方法；args 为目标类方法的入参；而 obj 为目标类实例。当该方法发生异常时，将阻止目标类方法的执行。

礼貌用语的前置增强制定好后，下面着手强制在服务生队伍中应用这个规定，来看具体的实施情况，如代码清单 7-13 所示。

代码清单 7-13 BeforeAdviceTest

```
package com.smart.advice;
import org.springframework.aop.BeforeAdvice;
import org.springframework.aop.framework.ProxyFactory;
import org.testng.annotations.*;
```

```
public class BeforeAdviceTest {

   @Test
    public void before() {
      Waiter target = new NaiveWaiter();
      BeforeAdvice advice = new GreetingBeforeAdvice();

        //①Spring 提供的代理工厂
      ProxyFactory pf = new ProxyFactory();

      // ②设置代理目标
      pf.setTarget(target);

      //③为代理目标添加增强
      pf.addAdvice(advice);

      //④生成代理实例
      Waiter proxy = (Waiter)pf.getProxy();
      proxy.greetTo("John");
      proxy.serveTo("Tom");}
}
```

运行上面的代码，可以看到以下输出信息：

```
How are you! Mr.John! ①
greet to John...
How are you! Mr.Tom! ②
serving Tom...
```

通过前置增强引入的礼貌用语

通过前置增强引入的礼貌用语

正如我们期望看到的一样，礼貌待客的优质服务理念得到了坚决、彻底的贯彻。

2．解剖 ProxyFactory

在 BeforeAdviceTest 中，使用 org.springframework.aop.framework.ProxyFactory 代理工厂将 GreetingBeforeAdvice 的增强织入目标类 NaiveWaiter 中。回想一下，前面介绍的 JDK 和 CGLib 动态代理技术是否有一些相似之处？不错，ProxyFactory 内部就是使用 JDK 或 CGLib 动态代理技术将增强应用到目标类中的。

Spring 定义了 org.springframework.aop.framework.AopProxy 接口，并提供了两个 final 类型的实现类，如图 7-5 所示。

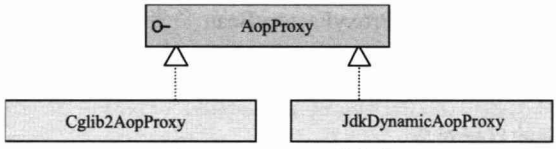


图 7-5　AopProxy 类结构

其中，Cglib2AopProxy 使用 CGLib 动态代理技术创建代理，而 JdkDynamicAopProxy 使用 JDK 动态代理技术创建代理。如果通过 ProxyFactory 的 setInterfaces(Class[] interfaces) 方法指定目标接口进行代理，则 ProxyFactory 使用 JdkDynamicAopProxy；如果是针对类的代理，则使用 Cglib2AopProxy。此外，还可以通过 ProxyFactory 的 setOptimize(true) 方法让 ProxyFactory 启动优化代理方式，这样，针对接口的代理也会使用 Cglib2AopProxy。

值得注意的一点是，在使用 CGLib 动态代理技术时，必须引入 CGLib 类库。

现在回过头来对代码清单 7-13 进行分析。BeforeAdviceTest 使用的是 CGLib 动态代理技术，当我们指定针对接口进行代理时，将使用 JDK 动态代理技术。

```
...
ProxyFactory pf = new ProxyFactory();
pf.setInterfaces(target.getClass().getInterfaces()); //①
pf.setTarget(target);
pf.addAdvice(advice);
...
```

指定对接口进行代理

如果指定启用代理优化，则 ProxyFactory 还将使用 Cglib2AopProxy 代理。

```
ProxyFactory pf = new ProxyFactory();
pf.setInterfaces(target.getClass().getInterfaces()); //①
pf.setOptimize(true);//②
pf.setTarget(target);
pf.addAdvice(advice);
```

指定对接口进行代理

启用优化

读者可能已经注意到，ProxyFactory 通过 addAdvice(Advice)方法添加一个增强，用户可以使用该方法添加多个增强。多个增强形成一个增强链，它们的调用顺序和添加顺序一致，可以通过 addAdvice(int, Advice)方法将增强添加到增强链的具体位置（第一个位置为 0）。

3. 在 Spring 中配置

使用 ProxyFactory 比直接使用 CGLib 或 JDK 动态代理技术创建代理省了很多事，如大家预想的一样，可以通过 Spring 的配置以“很 Spring 的方式”声明一个代理，如代码清单 7-14 所示。

代码清单 7-14　通过ProxyFactoryBean配置代理

```
<bean id="greetingAdvice" class="com.smart.advice.GreetingBeforeAdvice"/>①
<bean id="target" class="com.smart.advice.NaiveWaiter"/> ②
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="com.smart.advice.Waiter"③
    p:interceptorNames="greetingAdvice"④
    p:target-ref="target" ⑤
    />
```

指定代理的接口，如果是多个接口，请使用<list>元素

指定使用的增强（①处）

指定对哪个 Bean 进行代理（②处）

ProxyFactoryBean 是 FactoryBean 接口的实现类，5.9 节专门介绍了 FactoryBean 的功用，它负责实例化一个 Bean。ProxyFactoryBean 负责为其他 Bean 创建代理实例，它在内部使用 ProxyFactory 来完成这项工作。下面进一步了解一下 ProxyFactoryBean 的几个常用的可配置属性。

- target：代理的目标对象。
- proxyInterfaces：代理所要实现的接口，可以是多个接口。该属性还有一个别名属性 interfaces。
- interceptorNames：需要织入目标对象的 Bean 列表，采用 Bean 的名称指定。这些 Bean 必须是实现了 org.aopalliance.intercept.MethodInterceptor 或 org.springframework.aop.Advisor 的 Bean，配置中的顺序对应调用的顺序。
- singleton：返回的代理是否是单实例，默认为单实例。

- optimize：当设置为 true 时，强制使用 CGLib 动态代理。对于 singleton 的代理，我们推荐使用 CGLib；对于其他作用域类型的代理，最好使用 JDK 动态代理。原因是虽然 CGLib 创建代理时速度慢，但其创建出的代理对象运行效率较高；而使用 JDK 创建代理的表现正好相反。
- proxyTargetClass：是否对类进行代理（而不是对接口进行代理）。当设置为 true 时，使用 CGLib 动态代理。

运行如代码清单 7-15 所示的测试代码。

代码清单 7-15　测试增强

```
String configPath = "com/smart/advice/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter)ctx.getBean("waiter");
waiter.greetTo("John");
```

输出以下信息：

```
How are you! Mr.John.
greet to John...
```

这时，ProxyFactoryBean 使用了 JDK 动态代理技术。可以调整配置，使用 CGLib 动态代理技术通过动态创建 NaiveWaiter 的子类来代理 NaiveWaiter 对象，如下：

```
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="greetingAdvice"
    p:target-ref="target"
    p: proxyTargetClass ="true"/>
...
```

将 proxyTargetClass 设置为 true 后，无须再设置 proxyInterfaces 属性，即使设置也会被 ProxyFactoryBean 忽略。

7.3.3　后置增强

后置增强在目标类方法调用后执行。假设服务生在每次服务后也需要使用规范的礼貌用语，则可以通过一个后置增强来实施这一要求，如代码清单 7-16 所示。

代码清单 7-16　GreetingAfterAdvice

```
package com.smart.advice;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
public class GreetingAfterAdvice implements AfterReturningAdvice {

    //①在目标类方法调用后执行
    public void afterReturning(Object returnObj, Method method, Object[] args,
            Object obj) throws Throwable {
        System.out.println("Please enjoy yourself!");
    }
}
```

通过实现 AfterReturningAdvice 来定义后置增强的逻辑，AfterReturningAdvice 接口也仅定义了唯一的方法 afterReturning(Object returnObj, Method method, Object[]

args,Object obj) throws Throwable。其中，returnObj 为目标实例方法返回的结果；method 为目标类的方法；args 为目标实例方法的入参；而 obj 为目标类实例。假设在后置增强中抛出异常，如果该异常是目标方法声明的异常，则该异常归并到目标方法中；如果不是目标方法所声明的异常，则 Spring 将其转为运行期异常抛出。

下面将这个后置增强添加到上面的实例中，如代码清单 7-17 所示。

代码清单 7-17　添加后置增强

```
...
<bean id="greetingBefore" class="com.smart.advice.GreetingBeforeAdvice"/>
<bean id="greetingAfter" class="com.smart.advice.GreetingAfterAdvice"/>
<bean id="target" class="com.smart.advice.NaiveWaiter"/>
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="com.smart.advice.Waiter"
    p:target-ref="target"
    p:interceptorNames="greetingBefore,greetingAfter"/>
```

运行代码清单 7-15 中的代码，将输出以下信息：

```
How are you! Mr.John.①        ← 前置增强引入的逻辑
greet to John...
Please enjoy yourself! ②      ← 后置增强引入的逻辑
```

可见，前置、后置增强中的逻辑都成功地织入目标类 NaiveWaiter 方法所对应的连接点上。

实战经验

interceptorNames 是 String[]类型的，它接收增强 Bean 的名称而非增强 Bean 的实例。这是因为 ProxyBeanFactory 内部在生成代理类时，需要使用增强 Bean 的类，而非增强 Bean 的实例，以织入增强类中所写的横切逻辑代码，因而可以说增强是类级别的。

对于属性是字符数组类型且数组元素是 Bean 名称的配置，我们最好使用<idref local="xxx">标签，这样在一般的 IDE 环境下编辑 Spring 配置文件时，IDE 会即时发现配置错误并给出报警，以便开发者及早消除配置错误，如下：

```
<property name="interceptorNames">
    <list>
      <idref local="greetingBefore"/>
      <idref local="greetingAfter"/>
    </list>
</property>
```

当然，对于希望尽量简化配置文件的开发者来说，也可以采用逗号、分号或空格分隔的方式进行配置（字符串数组编辑器支持这种配置），如下：

```
<property name="interceptorNames" value="greetingBefore,greetingAfter">
```

7.3.4　环绕增强

介绍完前置、后置增强，环绕增强的作用就显而易见了。环绕增强允许在目标类方

法调用前后织入横切逻辑，它综合实现了前置、后置增强的功能。下面用环绕增强同时实现前礼貌用语和后礼貌用语。

```
package com.smart.advice;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class GreetingInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {//①

        Object[] args = invocation.getArguments();//目标方法入参
        String clientName = (String)args[0];
        System.out.println("How are you! Mr."+ clientName+".");②

        Object obj = invocation.proceed();//③

        System.out.println("Please enjoy yourself!"); //④
        return obj;
    }
}
```

截获目标类方法的执行，并在前后添加横切逻辑

在目标方法执行前调用

通过反射机制调用目标方法

在目标方法执行后调用

Spring 直接使用 AOP 联盟所定义的 MethodInterceptor 作为环绕增强的接口。该接口拥有唯一的接口方法 Object invoke(MethodInvocation invocation) throws Throwable。MethodInvocation 不但封装了目标方法及其入参数组，还封装了目标方法所在的实例对象，通过 MethodInvocation 的 getArguments()方法可以获取目标方法的入参数组，通过 proceed()方法反射调用目标实例相应的方法，如③处所示。通过在实现类中定义横切逻辑，可以很容易地实现方法前后的增强。

下面使用环绕增强替换前置和后置增强，如代码清单 7-18 所示。

代码清单 7-18　环绕增强配置

```
<bean id="greetingAround" class="com.smart.advice.GreetingInterceptor"/>
<bean id="target" class="com.smart.advice.NaiveWaiter"/>
<bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="com.smart.advice.Waiter"
    p:target-ref="target"
    p:interceptorNames="greetingAround"/>
```

运行代码清单 7-15 中的代码，将看到以下输出信息：

```
How are you! Mr.John.
greet to John...
Please enjoy yourself!
```

可见，环绕增强达到了前置和后置增强的联合效果。

7.3.5　异常抛出增强

异常抛出增强最适合的应用场景是事务管理，当参与事务的某个 DAO 发生异常时，事务管理器就必须回滚事务。在这里，仅仅给出一个模拟性的实例，用于说明异常抛出增强的使用方法，如代码清单 7-19 所示。

代码清单 7-19 ForumService

```
package com.smart.advice;
import java.sql.SQLException;
public class ForumService {
    public void removeForum(int forumId) {
        // do sth...
        throw new RuntimeException("运行异常。");
    }
    public void updateForum(Forum forum) throws Exception{
        // do sth...
        throw new SQLException("数据更新操作异常。");
    }
}
```

在模拟业务类 ForumService 中定义了两个业务方法，removeForum()抛出运行期异常，而 updateForum()抛出 SQLException。下面试图通过 TransactionManager 这个异常抛出增强对业务方法进行增强处理，统一捕捉抛出的异常并回滚事务，如代码清单 7-20 所示。

代码清单 7-20 TransactionManager

```
package com.smart.advice;
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;
public class TransactionManager implements ThrowsAdvice {

    //①定义增强逻辑
    public void afterThrowing(Method method, Object[] args, Object target,
            Exception ex) throws Throwable {
        System.out.println("-----------");
        System.out.println("method:" + method.getName());
        System.out.println("抛出异常:" + ex.getMessage());
        System.out.println("成功回滚事务。");
    }
}
```

ThrowsAdvice 异常抛出增强接口没有定义任何方法，它是一个标签接口，在运行期 Spring 使用反射机制自行判断，必须采用以下签名形式定义异常抛出的增强方法：

```
void afterThrowing([Method method, Object[] args, Object target], Throwable);
```

方法名必须为 afterThrowing，方法入参规定如下：前 3 个入参 Method method、Object[] args、Object target 是可选的（3 个入参要么提供，要么不提供），而最后一个入参是 Throwable 或其子类。如以下方法都是合法的：

- afterThrowing(SQLException e)。
- afterThrowing(RuntimeException e)。
- afterThrowing(Method method, Object[] args, Object target,RuntimeException e)。

而以下方法是非法的：

- afterThrowing(Object[] args, Object target,RuntimeException e)：缺少 Method。
- solveThrowing(SQLException e)：方法名非法。

可以在同一个异常抛出增强中定义多个 afterThrowing()方法，当目标类方法抛出异常时，Spring 会自动选用最匹配的增强方法。假设在增强中定义了两个方法：

- afterThrowing(SQLException e)。
- afterThrowing(Throwable e)。

当目标方法抛出一个 SQLException 时，将调用 afterThrowing(SQLException e)而非 afterThrowing(Throwable e)进行增强。在类的继承树上，两个类的距离越近，就说这两个类的相似度越高。目标方法抛出异常后，优先选取拥有异常入参和抛出的异常相似度最高的 afterThrowing()方法。

提示

标签接口是没有任何方法和属性的接口，它不对实现类有任何语义上的要求，仅仅表明它的实现类属于一个特定的类型。它非常类似于 Web 2.0 中 TAG 的概念，Java 使用它标识某一类对象。它主要有两个用途：第一，通过标签接口标识同一类型的类，这些类本身可能并不具有相同的方法，如 Advice 接口；第二，通过标签接口使程序或 JVM 采取一些特殊处理，如 java.io.Serializable，它告诉 JVM 对象可以被序列化。

在 Spring 中对这个异常抛出增强进行配置，如下：

```
<bean id="transactionManager" class="com.smart.advice.TransactionManager"/>
<bean id="forumServiceTarget" class="com.smart.advice.ForumService"/>
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
      p:interceptorNames="transactionManager"
      p:target-ref="forumServiceTarget"
      p:proxyTargetClass="true"/>①
```

因 ForumService 是类，使用 CGLib 代理

采用类似于代码清单 7-15 的代码测试这个异常抛出增强，将得到以下输出信息：

```
-----------
method:removeForum
抛出异常:运行异常。
成功回滚事务。
-----------
method:updateForum
抛出异常:数据更新操作异常。
成功回滚事务。
```

可见，ForumService 的两个方法所抛出的异常都被 TransactionManager 这个异常抛出增强捕获并成功处理。这样 ForumService 就从事务管理繁复的代码中解放出来，历史揭开了崭新的一页！

7.3.6　引介增强

引介增强是一种比较特殊的增强类型，它不是在目标方法周围织入增强，而是为目标类创建新的方法和属性，所以引介增强的连接点是类级别的，而非方法级别的。通过引介增强，可以为目标类添加一个接口的实现，即原来目标类未实现某个接口，通过引

介增强可以为目标类创建实现某接口的代理。这种功能富有吸引力，因为它能够在横向上定义接口的实现方法，思考问题的角度发生了很大的变化。

Spring 定义了引介增强接口 IntroductionInterceptor，该接口没有定义任何方法，Spring 为该接口提供了 DelegatingIntroductionInterceptor 实现类。一般情况下，通过扩展该实现类定义自己的引介增强类。

回到本章前面性能监视的例子，我们对所有的业务类都织入了性能监视的增强。由于性能监视会影响业务系统的性能，所以是否启用性能监视应该是可控的，即维护人员可以手工打开或关闭性能监视的功能。但原来的例子只简单地添加了运行性能监视逻辑，未提供任何控制的功能，现在可以用引介增强来实现这一诱人的功能。

首先定义一个用于标识目标类是否支持性能监视的接口，如代码清单 7-21 所示。

代码清单 7-21　Monitorable

```
package com.smart.introduce;
public interface Monitorable {
   void setMonitorActive(boolean active);
}
```

该接口仅包括一个 setMonitorActive(boolean active)方法，我们期望通过该接口方法控制业务类性能监视功能的激活和关闭状态。

下面通过扩展 DelegatingIntroductionInterceptor 为目标类引入性能监视的可控功能，如代码清单 7-22 所示。

代码清单 7-22　ControllablePerformanceMonitor

```
package com.smart.introduce;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.support.DelegatingIntroductionInterceptor;
public class ControllablePerformanceMonitor
         extends DelegatingIntroductionInterceptor
          implements Monitorable {
    private ThreadLocal<Boolean> MonitorStatusMap =new ThreadLocal <Boolean>();①
    public void setMonitorActive(boolean active) {②
        MonitorStatusMap .set(active);
    }

    //③拦截方法
    public Object invoke(MethodInvocation mi) throws Throwable {
        Object obj = null;

       //④对于支持性能监视可控代理，通过判断其状态决定是否开启性能监控功能
       if (MonitorStatusMap.get()!= null && MonitorStatusMap.get()) {
            PerformanceMonitor.begin(mi.getClass().getName() + "."
                    + mi.getMethod().getName());
            obj = super.invoke(mi);
            PerformanceMonitor.end();
        } else {
            obj = super.invoke(mi);
        }
```

```
        return obj;
    }
}
```

ControllablePerformanceMonitor 在扩展 DelegatingIntroductionInterceptor 的同时，还必须实现 Monitorable 接口，提供接口方法的实现。在①处定义了一个 ThreadLocal 类型的变量，用于保存性能监视开关状态。之所以使用 ThreadLocal 变量，是因为这个控制状态使代理类变成了非线程安全的实例，为了解决单实例线程安全的问题，通过 ThreadLocal 让每个线程单独使用一个状态。

在③处覆盖了父类中的 invoke()方法，该方法用于拦截目标类方法的调用，根据监视开关的状态有条件地对目标实例方法进行性能监视。④处的粗体代码所示部分可能有点难以理解，它使用了 Java 5.0 的自动拆包功能，MonitorStatusMap.get()方法返回的 Boolean 被自动拆包为 boolean 类型的值。

下面通过 Spring 的配置，将这个引介增强织入业务类 ForumService 中，具体配置如代码清单 7-23 所示。

代码清单 7-23　配置引介增强

```
<bean id="pmonitor" class="com.smart.introduce.Controllable
PerformanceMonitor"/>
<bean id="forumServiceTarget" class="com.smart.introduce.ForumService"/>
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
        p:interfaces="com.smart.introduce.Monitorable" ①
        p:target-ref="forumServiceTarget"
        p:interceptorNames="pmonitor"
        p:proxyTargetClass="true" />②
```

引介增强所实现的接口

由于引介增强一定要通过创建子类来生成代理，所以需要强制使用 CGLib，否则会报错

引介增强的配置与一般的配置有较大的区别：首先，需要指定引介增强所实现的接口，如①处所示，这里的引介增强实现了 Monitorable 接口；其次，由于只能通过为目标类创建子类的方式生成引介增强的代理，所以必须将 proxyTargetClass 设置为 true。

如果没有对 ControllablePerformanceMonitor 进行线程安全的特殊处理，就必须将 singleton 属性设置为 ture，让 ProxyFactoryBean 产生 prototype 作用域类型的代理。这就带来了一个严重的性能问题，因为CGLib动态创建代理的性能很低，而每次通过 getBean()方法从容器中获取作用域类型为 prototype 的 Bean 时都将返回一个新的代理实例，所以这种性能的影响是巨大的，这也是为什么在代码中通过 ThreadLocal 对 ControllablePerformanceMonitor 的开关状态进行线程安全化处理的原因。通过线程安全化处理后，就可以使用默认的 singleton Bean 作用域，这样创建代理的动作仅发生一次。

代码清单 7-24 所示的代码对织入性能监视控制接口业务类方法的调用情况进行测试。

代码清单 7-24　IntroduceTest：测试引介增强

```
package com.smart.introduce;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.testng.annotations.*;
```

```
public class IntroduceTest {
    @Test
     public void introduce(){
        String configPath = "com/smart/introduce/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext (configPath);
        ForumService forumService = (ForumService)ctx.getBean("forumService");

        forumService.removeForum(10);
        forumService.removeTopic(1022);

        Monitorable moniterable = (Monitorable)forumService;
        moniterable.setNeedMonitor(true);

        forumService.removeForum(10);
        forumService.removeTopic(1022);
    }
}
```

默认情况下，未开启性能监视功能 ①

开启性能监视功能 ②

③ *在性能监视功能开启的情况下，再次调用业务方法*

注意②处的(Monitorable)forumService 代码，强制性地将 forumService 转换为 Monitorable 类型。代码的成功执行表示从 Spring 容器中返回的代理确实引入了 Monitorable 接口方法的实现。

执行以上代码，可以看到以下输出信息：

```
模拟删除Forum记录:10
模拟删除Topic记录:1022

begin monitor...
模拟删除Forum记录:10
end monitor...
org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation. removeForum
花费47毫秒。
begin monitor...
模拟删除Topic记录:1022
end monitor...
org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation. removeTopic
花费16毫秒。
```

① *未激活监视功能*

② *激活监视功能*

在①处，只有业务逻辑被执行，性能监视功能没有被执行；而在②处，性能监视功能正常启用，两个业务方法都启用了性能监视功能。

提示

在 Spring 4.0 中，基于 CGLib 的类代理不再要求目标类必须有无参构造函数。这是一个不错的特性，这样在使用 CGLib 类时，不再需要特别关注目标类是否有无参构造函数。取消这个限制后，增强的目标 Bean 就可以使用构造函数注入了。Spring 到底如何实现这个功能？这就要归根于 Spring 内联了 objenesis 类库，感兴趣的读者可到其官网（http://objenesis.org）查看。

7.4　创建切面

在介绍增强时，读者可能会注意到一个问题：增强被织入目标类的所有方法中。假设我们希望有选择地织入目标类的某些特定方法中，就需要使用切点进行目标连接点的定位。描述连接点是进行 AOP 编程最主要的工作，为了突出强调这一点，再次给出 Spring AOP 如何定位连接点。

增强提供了连接点方位信息，如织入到方法前面、后面等，而切点进一步描述了织入哪些类的哪些方法上。

Spring 通过 org.springframework.aop.Pointcut 接口描述切点，Pointcut 由 ClassFilter 和 MethodMatcher 构成，它通过 ClassFilter 定位到某些特定类上，通过 MethodMatcher 定位到某些特定方法上，这样 Pointcut 就拥有了描述某些类的某些特定方法的能力。可以简单地用 SQL 复合查询条件来理解 Pointcut 的功用。Pointcut 类关系图如图 7-6 所示。

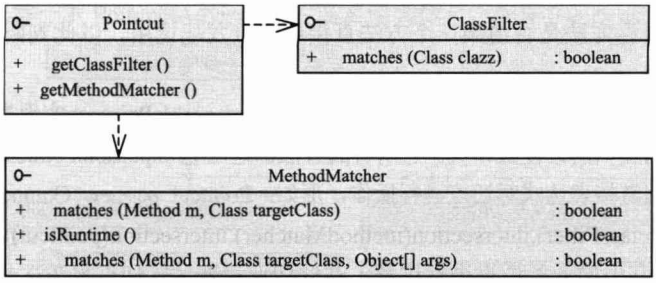


图 7-6　Pointcut 类关系图

可以看到 ClassFilter 只定义了一个方法 matches(Class clazz)，其参数代表一个被检测类，该方法判别被检测的类是否匹配过滤条件。

Spring 支持两种方法匹配器：静态方法匹配器和动态方法匹配器。所谓静态方法匹配器，仅对方法名签名（包括方法名和入参类型及顺序）进行匹配；而动态方法匹配器会在运行期检查方法入参的值。静态匹配仅会判别一次，而动态匹配因为每次调用方法的入参都可能不一样，所以每次调用方法都必须判断，因此，动态匹配对性能的影响很大。一般情况下，动态匹配不常使用。方法匹配器的类型由 isRuntime()方法的返回值决定，返回 false 表示是静态方法匹配器，返回 true 表示是动态方法匹配器。

此外，Spring 2.0 还支持注解切点和表达式切点，前者通过 Java 5.0 的注解定义切点，而后者通过字符串表达式定义切点，二者都使用 AspectJ 的切点表达式语言。

7.4.1　切点类型

Spring 提供了 6 种类型的切点，下面分别对它们的用途进行介绍。

- 静态方法切点：org.springframework.aop.support.StaticMethodMatcherPointcut 是静态方法切点的抽象基类，默认情况下它匹配所有的类。StaticMethodMatcherPointcut 包括两个主要的子类，分别是 NameMatchMethodPointcut 和 AbstractRegexp MethodPointcut，前者提供简单字符串匹配方法签名，而后者使用正则表达式匹配方法签名。
- 动态方法切点：org.springframework.aop.support.DynamicMethodMatcherPointcut 是动态方法切点的抽象基类，默认情况下它匹配所有的类。
- 注解切点：org.springframework.aop.support.annotation.AnnotationMatchingPointcut 实现类表示注解切点。使用 AnnotationMatchingPointcut 支持在 Bean 中直接通过 Java 5.0 注解标签定义的切点。
- 表达式切点：org.springframework.aop.support.ExpressionPointcut 接口主要是为了支持 AspectJ 切点表达式语法而定义的接口。
- 流程切点：org.springframework.aop.support.ControlFlowPointcut 实现类表示控制流程切点。ControlFlowPointcut 是一种特殊的切点，它根据程序执行堆栈的信息查看目标方法是否由某一个方法直接或间接发起调用，以此判断是否为匹配的连接点。
- 复合切点：org.springframework.aop.support.ComposablePointcut 实现类是为创建多个切点而提供的方便操作类。它所有的方法都返回 ComposablePointcut 类，这样就可以使用链接表达式对切点进行操作，形如：Pointcut pc=new ComposablePointcut().union(classFilter).intersection(methodMatcher).intersection(pointcut)。

本章仅对其中的 4 类切点进行讲解，注解切点和表达式切点将在下一章讲解。

7.4.2 切面类型

由于增强既包含横切代码，又包含部分连接点信息（方法前、方法后主方位信息），所以可以仅通过增强类生成一个切面。但切点仅代表目标类连接点的部分信息（类和方法的定位），所以仅有切点无法制作出一个切面，必须结合增强才能制作出切面。Spring 使用 org.springframework.aop.Advisor 接口表示切面的概念，一个切面同时包含横切代码和连接点信息。切面可以分为 3 类：一般切面、切点切面和引介切面，可以通过 Spring 所定义的切面接口清楚地了解切面的分类，如图 7-7 所示。

- Advisor：代表一般切面，仅包含一个 Advice。因为 Advice 包含了横切代码和连接点信息，所以 Advice 本身就是一个简单的切面，只不过它代表的横切的连接点是所有目标类的所有方法，因为这个横切面太宽泛，所以一般不会直接使用。
- PointcutAdvisor：代表具有切点的切面，包含 Advice 和 Pointcut 两个类，这样就可以通过类、方法名及方法方位等信息灵活地定义切面的连接点，提供更具适用性的切面。

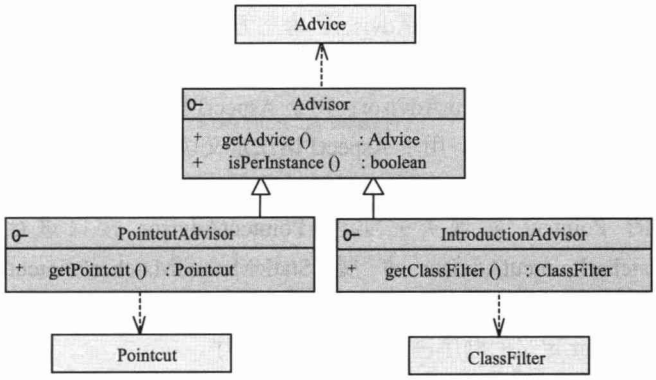


图 7-7　切面类继承关系

- IntroductionAdvisor：代表引介切面。7.3.6 节介绍了引介增强类型，引介切面是对应引介增强的特殊的切面，它应用于类层面上，所以引介切点使用 ClassFilter 进行定义。

下面再来看一下 PointcutAdvisor 的主要实现类体系，如图 7-8 所示。

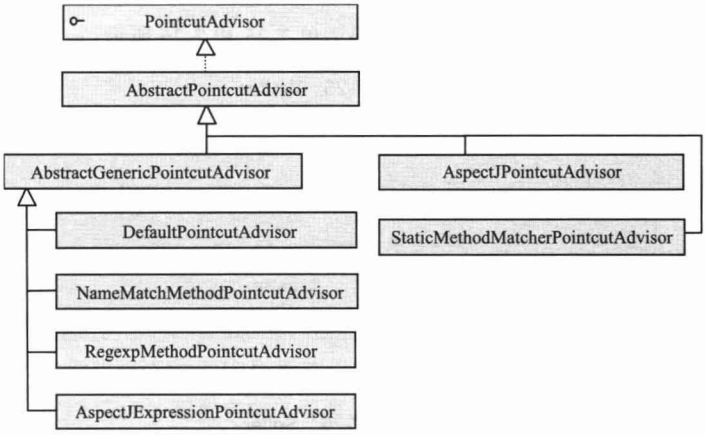


图 7-8　PointcutAdvisor 实现类体系

PointcutAdvisor 主要有 6 个具体的实现类，分别介绍如下。

- DefaultPointcutAdvisor：最常用的切面类型，它可以通过任意 Pointcut 和 Advice 定义一个切面，唯一不支持的是引介的切面类型，一般可以通过扩展该类实现自定义的切面。
- NameMatchMethodPointcutAdvisor：通过该类可以定义按方法名定义切点的切面。
- RegexpMethodPointcutAdvisor：对于按正则表达式匹配方法名进行切点定义的切面，可以通过扩展该实现类进行操作。RegexpMethodPointcutAdvisor 允许用户以正则表达式模式串定义方法匹配的切点，其内部通过 JdkRegexpMethodPointcut 构造出正则表达式方法名切点。

- StaticMethodMatcherPointcutAdvisor：静态方法匹配器切点定义的切面，默认情况下匹配所有的目标类。
- AspectJExpressionPointcutAdvisor：用于 AspectJ 切点表达式定义切点的切面。
- AspectJPointcutAdvisor：用于 AspectJ 语法定义切点的切面。

这些 Advisor 的实现类都可以在 Pointcut 中找到对应物，实际上，它们都是通过扩展对应的 Pointcut 实现类并实现 PointcutAdvisor 接口进行定义的。如 StaticMethodMatcherPointcutAdvisor 扩展 StaticMethodMatcherPointcut 类并实现 PointcutAdvisor 接口。此外，Advisor 都实现了 org.springframework.core.Ordered 接口，Spring 将根据 Advisor 定义的顺序决定织入切面的顺序。

在了解了切点和切面的知识后，下面将通过具体实例进一步了解它们的具体用法。

7.4.3 静态普通方法名匹配切面

StaticMethodMatcherPointcutAdvisor 代表一个静态方法匹配切面，它通过 StaticMethodMatcherPointcut 来定义切点，并通过类过滤和方法名来匹配所定义的切点。来看下面的 Waiter 和 Seller 业务类，如代码清单 7-25 和 7-26 所示。

代码清单 7-25 Waiter

```
package com.smart.advisor;
public class Waiter {
    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
    public void serveTo(String name){
        System.out.println("waiter serving "+name+"...");
    }
}
```

Waiter 有两个方法，分别是 greetTo()和 serveTo()。

代码清单 7-26 Seller

```
package com.smart.advisor;
public class Seller {
    public void greetTo(String name) {
        System.out.println("seller greet to "+name+"...");
    }
}
```

Seller 拥有一个和 Waiter 相同名称的方法 greetTo()。现在，我们希望通过 StaticMethodMatcherPointcutAdvisor 定义一个切面，在 Waite#greetTo()方法调用前织入一个增强，即连接点为 Waiter#greetTo()方法调用前的位置。具体的切面类的实现如代码清单 7-27 所示。

代码清单 7-27 GreetingAdvisor

```
package com.smart.advisor;
import java.lang.reflect.Method;
```

```
import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.StaticMethodMatcherPointcutAdvisor;
public class GreetingAdvisor extends StaticMethodMatcherPointcutAdvisor {
    public boolean matches(Method method, Class clazz) {//①
        return "greetTo".equals(method.getName());
    }
    public ClassFilter getClassFilter(){ //②
        return new ClassFilter(){
            public boolean matches(Class clazz){
                return Waiter.class.isAssignableFrom(clazz);
            }
        };
    }
}
```

切点方法匹配规则：方法名为 greetTo

切点类匹配规则：为 Waiter 的类或子类

StaticMethodMatcherPointcutAdvisor 抽象类唯一需要定义的是 matches()方法。在默认情况下，该切面匹配所有的类，这里通过覆盖 getClassFilter()方法，让它仅匹配 Waiter 类及其子类。

当然，Advisor 还需要一个增强类的配合，我们定义一个前置增强，如代码清单 7-28 所示。

代码清单 7-28　GreetingBeforeAdvice

```
package com.smart.advisor;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class GreetingBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object obj) throws Throwable {
        System.out.println(obj.getClass().getName()+"."+method.getName());//①
        String clientName = (String)args[0];
        System.out.println("How are you! Mr."+clientName+".");
    }
}
```

输出切点

当然，可以直接使用 ProxyFactory，通过手工编码的方式织入切面生成代理类，有兴趣的读者可以参考代码清单 7-13 所示的代码，在③处使用 addAdvisor()方法添加切面就可以了。下面使用 Spring 配置来定义切面，如代码清单 7-29 所示。

代码清单 7-29　配置切面：静态方法匹配切面

```
<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="sellerTarget" class="com.smart.advisor.Seller"/>
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice"/>
<bean id="greetingAdvisor" class="com.smart.advisor.GreetingAdvisor"
        p:advice-ref="greetingAdvice"/>①
<bean id="parent" abstract="true"②
        class="org.springframework.aop.framework.ProxyFactoryBean"
        p:interceptorNames="greetingAdvisor"
        p:proxyTargetClass="true"/>
<bean id="waiter" parent="parent"  p:target-ref="waiterTarget"/>③
<bean id="seller" parent="parent"  p:target-ref="sellerTarget"/>④
```

向切面注入一个前置增强

通过一个父<bean>定义公共的配置信息

waiter 代理

seller 代理

在①处，将 greetingAdvice 增强装配到 greetingAdvisor 切面中。StaticMethodMatcherPointcutAdvisor 除具有 advice 属性外，还可以定义另外两个属性。

- classFilter：类匹配过滤器，在 GreetingAdvisor 中用编码的方式设定了 classFilter。
- order：切面织入时的顺序，该属性用于定义 Ordered 接口表示的顺序。

由于需要分别为 waiter 和 seller 两个 Bean 定义代理器，且二者有很多公共的配置信息，所以使用了一个父<bean>简化配置，如②处所示。在③和④处，通过引用父<bean>轻松地定义了两个织入切面的代理。

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter)ctx.getBean("waiter");
Seller seller = (Seller)ctx.getBean("seller");
waiter.greetTo("John");
waiter.serveTo("John");
seller.greetTo("John");
```

运行以上代码，输出以下信息：

```
com.smart.advisor.Waiter.greetTo ①
How are you! Mr.John.
waiter greet to John...
waiter serving John...
seller greet to John...
```

这两行为切面引入的增强逻辑

可见切面只织入 Waiter.greetTo()方法调用前的连接点上，Waiter.serveTo()和 Seller.greetTo()方法没有织入切面。

7.4.4 静态正则表达式方法匹配切面

在 StaticMethodMatcherPointcutAdvisor 中，仅能通过方法名定义切点，这种描述方式不够灵活。假设目标类中有多个方法，且它们都满足一定的命名规范，使用正则表达式进行匹配描述就要灵活多了。RegexpMethodPointcutAdvisor 是正则表达式方法匹配的切面实现类，该类已经是功能齐备的实现类，一般情况下无须扩展该类。

1. 具体实例

下面直接使用 RegexpMethodPointcutAdvisor，通过配置的方式为 Waiter 目标类定义一个切面，如代码清单 7-30 所示。

代码清单 7-30 通过正则表达式定义切面

```
<bean id="regexpAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
    p:advice-ref="greetingAdvice">
    <property name="patterns">①
        <list>
            <value>.*greet.*</value>②
        </list>
    </property>
</bean>
<bean id="waiter1" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:interceptorNames="regexpAdvisor"
    p:target-ref="waiterTarget"
    p:proxyTargetClass="true"/>
```

用正则表达式定义目标类全限定方法名的匹配模式串

匹配模式串

在②处定义了一个匹配模式串“.*greet.*”，该模式串匹配 Waiter.greetTo()方法。值得注意的是，匹配模式串匹配的是目标类方法的全限定名，即带类名的方法名。

运行下列测试代码：

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter)ctx.getBean("waiter1");
waiter.greetTo("John");
waiter.serveTo("John");
```

输出以下信息：

```
com.smart.advisor.Waiter.greetTo ①
How are you! Mr.John.
waiter greet to John...
waiter serving John...
```

这两行为切面引入的增强逻辑

可见，Waiter.greetTo()方法被织入了切面，而 Waiter.serveTo()方法没有被织入切面。除了例子中所使用的 patterns 和 advice 属性外，还有另外两个属性，分别介绍如下。

- pattern：如果只有一个匹配模式串，则可以使用该属性进行配置。patterns 属性用于定义多个匹配模式串，这些匹配模式串之间是“或”的关系。
- order：切面在织入时对应的顺序。

2. 正则表达式语法

正则表达式的语法内容较多，这里仅对常见的正则表达式知识进行简单介绍（见表 7-1），相信这些知识足以应付 AOP 配置的日常所需。

表 7-1　正则表达式符号

符　号	说　明	实　例
.	匹配除换行符外的所有单个的字符	.n 匹配 nay, an apple is on the tree 中的 an 和 on，但不匹配 nay
*	匹配*前面的字符 0 次或 *n* 次	bo*匹配 A ghost booooed 中的 boooo 或 A bird warbled 中的 b，但不匹配 Agoat g runted 中的任何字符
+	匹配+前面的字符 1 次或 *n* 次。等价于{1,}	a+匹配 candy 中的 a 和 caaaaaaandy.中的所有 a
^	表示匹配的字符必须在最前边	^A 不匹配 an A 中的 A，但匹配 An A.中最前面的 A
$	与^类似，匹配最末的字符	t$不匹配 eater 中的 t，但匹配 eat 中的 t
?	匹配?前面的字符 0 次或 1 次	e?le?匹配 angel 中的 el 和 angle 中的 le
x|y	匹配 x 或者 y	green|red 匹配 green apple 中的 green 和 red apple.中的 red
[xyz]	一张字符列表，匹配列表中的任一字符。可以通过连字符“-”指出一个字符范围	[abc]跟[a-c]一样。它们匹配 brisket 中的 b 及 ache 中的 a 和 c
{n}	这里的 n 是一个正整数。匹配前面的 n 个字符	a{2}不匹配 candy 中的 a，但匹配 caandy 中的两个 a
{n,}	这里的 n 是一个正整数。匹配至少 n 个前面的字符	a{2,}不匹配 candy 中的 a，但匹配 caandy 中的所有 a 和 caaaaaaandy.中的所有 a
{n,m}	这里的 n 和 m 都是正整数。匹配至少 n 个最多 m 个前面的字符	a{1,3}不匹配 cndy 中的任何字符，但匹配 candy 中的 a 和 caandy 中的前面两个 a 和 caaaaaaandy 中前面的 3 个 a。注意，即使 caaaaaaandy 中有很多个 a，但只匹配前面的 3 个 a，即 aaa

续表

符　号	说　明	实　例
\	将下一个字符标记为一个特殊字符	例如，n 匹配字符 n。\n 匹配一个换行符。语法中的特殊字符需要通过转义符表示，如\.表示.，而\{表示{
转义字符		
\d 匹配一个数字字符。等价于 [0-9]		
\D 匹配一个非数字字符。等价于 [^0-9]		
\f 匹配一个换页符。等价于\x0c 和\cL		
\n 匹配一个换行符。等价于\x0a 和\cJ		
\r 匹配一个回车符。等价于\x0d 和\cM		
\s 匹配任何空白字符，包括空格、制表符、换页符等。等价于[\f\n\r\t\v]		
\S 匹配任何非空白字符。等价于[^ \f\n\r\t\v]		
\t 匹配一个制表符。等价于\x09 和\cI		
\v 匹配一个垂直制表符。等价于\x0b 和\cK		
\w 匹配包括下画线的任何单词字符。等价于[A-Za-z0-9_]		
\W 匹配任何非单词字符。等价于 [^A-Za-z0-9_]		

下面举几个例子，进一步认识正则表达式在配置匹配方法上的具体应用。

示例 1：.*set.*表示所有类中以 set 为前缀的方法，如 com.smart.Waiter.setSalary()、Person.setName()等。

示例 2：com\.smart\.advisor\..* 表示 com.smart.advisor 包下所有类的所有方法。

示例 3：com\.smart\.service\..*Service\..* 匹配 com.smart.service 包下所有类名以 Service 结尾的类的所有方法，如 com.smart.service.UserService.save(User user)、com.smart.service.ForumService.update(Forum forum)等。

示例 4：com\.smart\.service\..*\.save.+匹配所有以 save 为前缀的方法，该方法后还必须拥有至少一个字符，且这些方法位于 com.smart.service 包中以 Service 为后缀的类中。如匹配 com.smart.service.UserService 类的 saveUser()和 saveLoginLog()方法，但不匹配该类的 save()方法。

只要程序的类包具有良好的命名规范，就可以使用简单的正则表达式描述出目标方法。由于需要使用全限定名来定义方法名，所以不但方法名需要具有良好的规范性，包名也需要具体良好的规范性。对包名、类名、方法名按其功用进行规范命名并不是一件坏事，相反，规范命名可以增强程序的可读性和团队开发的协作性，降低沟通成本，是值得实践和提倡的编程方法。

实战经验

在编写正则表达式时，通过一些好用的工具可以取得事半功倍的效果。RegexBuddy 是使用正则表达式时最好的助手，借助这款小巧的工具可以很容易地创建符合用户要求的正则表达式。读者可以通过 http://www.regexbuddy.com 了解更多关于这款软件的信息。

7.4.5　动态切面

在低版本中，Spring 提供了用于创建动态切面的 DynamicMethodMatcherPointcut Advisor 抽象类，因为该类在功能上和其他类有重叠，会给开发者造成选择上的困惑，因此在 Spring 2.0 中已经过期。可以使用 DefaultPointcutAdvisor 和 DynamicMethod MatcherPointcut 来完成相同的功能。

DynamicMethodMatcherPointcut 是一个抽象类，它将 isRuntime()标识为 final 且返回 true，这样其子类就一定是一个动态切点。该抽象类默认匹配所有的类和方法，因此需要通过扩展该类编写符合要求的动态切点，如代码清单 7-31 所示。

代码清单 7-31　GreetingDynamicPointcut

```
package com.smart.advisor;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.DynamicMethodMatcherPointcut;
public class GreetingDynamicPointcut extends DynamicMethodMatcherPointcut {
    private static List<String> specialClientList = new ArrayList<String>();
    static {
        specialClientList.add("John");
        specialClientList.add("Tom");
    }
    public ClassFilter getClassFilter() {//①
        return new ClassFilter() {
            public boolean matches(Class clazz) {
                System.out.println("调用getClassFilter()对"+clazz.getName()+ "做静
态检查.");
                return Waiter.class.isAssignableFrom(clazz);
            }
        };
    }
   public boolean matches(Method method, Class clazz) { //②
        System.out.println("调用matches(method,clazz)"+clazz.getName()+"."+
                           method.getName()+"做静态检查.");
        return "greetTo".equals(method.getName());
    }
    public boolean matches(Method method, Class clazz, Object[] args) {//③
        System.out.println("调用matches(method,clazz)"+clazz.getName()+"."
                           +method.getName()+"做动态检查.");
        String clientName = (String) args[0];
        return specialClientList.contains(clientName);
    }
}
```


GreetingDynamicPointcut 类既有用于静态切点检查的方法，又有用于动态切点检查的方法。由于动态切点检查会对性能造成很大的影响，所以应当尽量避免在运行时每次都对目标类的各个方法进行动态检查。Spring 采用这样的机制：在创建代理时对目标类的每个连接点使用静态切点检查，如果仅通过静态切点检查就可以知道连接点是不匹配

的，则在运行时就不再进行动态检查；如果静态切点检查是匹配的，则在运行时才进行动态切点检查。

在动态切点类中定义静态切点检查的方法可以避免不必要的动态检查操作，从而极大地提高运行效率，这一点在稍后的运行测试中再作进一步分析。

在 GreetingDynamicPointcut 类中，在③处通过 matches(Method method, Class clazz, Object[] args)定义了动态切点检查的方法，只对目标方法为 greetTo(clientName)且 clientName 为特殊客户的方法启用增强，通过 specialClientList 模拟特殊的客户名单。

在编写好动态切点后，就可以着手在 Spring 配置文件中装配出一个动态切面，如代码清单 7-32 所示。

代码清单 7-32　动态切面配置

```
<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="dynamicAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
   <property name="pointcut">
     <bean class="com.smart.advisor.GreetingDynamicPointcut"/>①
   </property>
   <property name="advice">
     <bean class="com.smart.advisor.GreetingBeforeAdvice"/>
   </property>
</bean>
<bean id="waiter2" class="org.springframework.aop.framework.ProxyFactoryBean"
   p:interceptorNames="dynamicAdvisor"
   p:target-ref="waiterTarget"
   p:proxyTargetClass="true"/>
```

动态切面的配置和静态切面的配置没有什么区别。使用 DefaultPointcutAdvisor 定义切面，在①处使用内部 Bean 方式注入动态切点 GreetingDynamicPointcut，增强依旧使用前面定义的 GreetingBeforeAdvice。此外，DefaultPointcutAdvisor 还有一个 order 属性，用于定义切面的织入顺序。

一切准备就绪后，开始编写一个测试类，这样就可以看到动态切面的“庐山真面目”了，如代码清单 7-33 所示。

代码清单 7-33　动态切面测试代码

```
package com.smart.advisor;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.testng.annotations.*;
public class DynamicAdvisorTest {

    @Test
    public void dynamic() {
        String configPath = "com/smart/advisor/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter2");
        waiter.serveTo("Peter");
        waiter.greetTo("Peter");
        waiter.serveTo("John");
```

```
        waiter.greetTo("John"); //①
    }                                          John 是特殊客户
}
```

运行以上代码，在控制台上输出以下信息：

```
                                    以下 10 行输出信息反映了在织入切面前 Spring
①                                   对目标类中所有方法进行的静态切点检查
调用getClassFilter()对com.smart. advisor.Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.serveTo做静态检查.
调用getClassFilter()对com.smart.advisor.Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做静态检查.
调用getClassFilter()对com.smart.advisor.Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.hashCode做静态检查.
调用getClassFilter()对com.smart.advisor.Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.clone做静态检查.
调用getClassFilter()对com.smart.advisor.Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.toString做静态检查.
                                    对应 waiter.serveTo("Peter")：第一次
②                                   调用 serveTo()方法时，执行静态切点检查
调用getClassFilter()对com.smart.advisor. Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.serveTo做静态检查.
waiter serving Peter...
                                    以下 4 行对应 waiter.greetTo("Peter")：第一次
③                                   调用 greetTo()方法时，执行静态、动态切点检查
调用getClassFilter()对com.smart.advisor.Waiter做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做静态检查.
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做动态检查.
waiter greet to Peter...
                                    对应 waiter.serveTo("John")：第二次调用
④                                   serveTo()方法时，不再执行静态切点检查
waiter serving John...
                                    以下 4 行对应 waiter.greetTo("John")：第二
⑤                                   次调用 greetTo()方法时，只执行动态切点检查
调用matches(method,clazz)对com.smart. advisor.Waiter.greetTo做动态检查.
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
```

通过以上输出信息，对照 DynamicMethodMatcherPointcut 切点类，可以很容易发现，Spring 会在创建代理织入切面时，对目标类中的所有方法进行静态切点检查；在生成织入切面的代理对象后，第一次调用代理类的每一个方法时都会进行一次静态切点检查，如果本次检查就能从候选者列表中将该方法排除，则以后对该方法的调用就不再执行静态切点检查；对于那些在静态切点检查时匹配的方法，在后续调用该方法时，将执行动态切点检查。

在例子中，切点匹配的规则是：目标类为 com.smart.advisor.Waiter 或其子类；方法名为 greetTo；动态入参 clientName 必须是特殊名单中的客户。

基于这条规则，serveTo()及从 Object 中继承而来的 toString()、hashCode()和 clone()等方法通过静态切点检查就可以排除在候选者之外，只有 greetTo()方法是动态切点检查的候选者，每次调用都会进行动态切点检查。

如果将 GreetingDynamicPointcut 类的 getClassFilter()和 matches(Method method,

Class clazz)方法注释掉，重新运行代码清单 7-33 所示的测试代码，则将得到以下输出信息：

```
调用matches(method,clazz)对com.smart.advisor.Waiter.serveTo做动态检查.
waiter serving Peter...
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做动态检查.
waiter greet to Peter...
调用matches(method,clazz)对com.smart.advisor.Waiter.serveTo做动态检查.
waiter serving Peter...
调用matches(method,clazz)对com.smart.advisor.Waiter.greetTo做动态检查.
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
```

可以发现，每次调用代理对象的任何一个方法，都会执行动态切点检查，这将导致很大的性能问题。所以，在定义动态切点时，切勿忘记同时覆盖 getClassFilter()和 matches(Method method, Class clazz)方法，通过静态切点检查排除大部分方法。

提示

7.2 节介绍了动态代理的概念，在这里又碰到了动态切面的概念，这两个概念很容易混淆。其实动态代理的“动态”是相对于那些编译期生成代理和类加载期生成代理而言的。动态代理是运行时动态产生的代理。在 Spring 中，不管是静态切面还是动态切面，都是通过动态代理技术实现的。所谓静态切面，是指在生成代理对象时就确定了增强是否需要织入目标类的连接点上；而动态切面是指必须在运行期根据方法入参的值来判断增强是否需要织入目标类的连接点上。

7.4.6 流程切面

Spring 的流程切面由 DefaultPointcutAdvisor 和 ControlFlowPointcut 实现。流程切点代表由某个方法直接或间接发起调用的其他方法。来看下面的实例，假设通过一个 WaiterDelegate 类代理 Waiter 所有的方法，如代码清单 7-34 所示。

代码清单 7-34 WaiterDelegate

```
package com.smart.advisor;
public class WaiterDelegate {
    private Waiter waiter;
    public void service(String clientName) { //①  <-- waiter 的方法通过该方法发起调用
        waiter.greetTo(clientName);
        waiter.serveTo(clientName);
    }
    public void setWaiter(Waiter waiter) {
        this.waiter = waiter;
    }
}
```

如果希望所有由 WaiterDelegate#service()方法发起调用的其他方法都织入 GreetingBeforeAdvice 增强，就必须使用流程切面来完成目标。下面使用

DefaultPointcutAdvisor 配置一个流程切面来完成这一需求，如代码清单 7-35 所示。

代码清单 7-35　配置控制流程切面

```
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="waiterTarget" class="com.smart.advisor.Waiter"/>
<bean id="controlFlowPointcut"
class="org.springframework.aop.support.ControlFlowPointcut">
    <constructor-arg type="java.lang.Class"
                value="com.smart.advisor.WaiterDelegate" />①  指定流程切点的类
    <constructor-arg type="java.lang.String"
                value="service" /> ②  指定流程切点的方法
</bean>
<bean id="controlFlowAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor"
        p:pointcut-ref="controlFlowPointcut"
       p:advice-ref="greetingAdvice" />

<bean id="waiter3" class="org.springframework.aop.framework.ProxyFactoryBean"
     p:interceptorNames="controlFlowAdvisor"
       p:target-ref="waiterTarget"
       p:proxyTargetClass="true"/>
```

ControlFlowPointcut 有两个构造函数，分别是 ControlFlowPointcut(Class clazz)和 ControlFlowPointcut(Class clazz, String methodName)。第一个构造函数指定一个类作为流程切点；而第二个构造函数指定一个类和某一个方法作为流程切点。

在这里，指定 com.smart.advisor.WaiterDelegate#service()方法作为切点，表示所有通过该方法直接或间接发起的调用匹配切点。说到这里，可能还有一些模糊的地方，下面通过测试代码观察流程切面的运行效果，如代码清单 7-36 所示。

代码清单 7-36　流程切面测试

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter3");
WaiterDelegate wd = new WaiterDelegate();
wd.setWaiter(waiter);
waiter.serveTo("Peter");
waiter.greetTo("Peter");
wd.service("Peter");
```

运行上面的代码，在控制台上输出以下信息：

```
waiter serving Peter...  ①  对应 waiter.serveTo("Peter")
waiter greet to Peter... ②  对应 waiter.greetTo("Peter")
③  对应 wd.service("Peter")
com.smart.advisor.Waiter.greetTo
How are you! Mr.Peter.
waiter greet to Peter...
com.smart.advisor.Waiter.serveTo
How are you! Mr.Peter.
waiter serving Peter...
```

①处和②处的信息是直接通过 waiter 调用 serveTo()和 greetTo()方法的输出，此时增强没有起作用；③处是通过 WaiterDelegate#service()调用 Waiter 的 serveTo()和 greetTo()方法的输出，这时发现 Waiter 的两个方法都织入了增强。

流程切面和动态切面从某种程度上说可以算是一类切面，因为二者都需要在运行期判断动态的环境。对于流程切面来说，代理对象在每次调用目标类方法时，都需要判断方法调用堆栈中是否有满足流程切点要求的方法。因此，和动态切面一样，流程切面对性能的影响也很大。在 JVM 1.4 上，流程切点通常比别的切点要慢 5 倍，在 JVM 1.3 上要慢 10 倍。

7.4.7 复合切点切面

在前面的例子中定义的切面仅有一个切点，有时，一个切点可能难以描述目标连接点的信息。比如在前面流程切面的例子中，假设我们希望由 WaiterDelegate# service()发起调用且被调用的方法是 Waiter#greetTo()时才织入增强，这个切点就是复合切点，因为它由两个单独的切点共同确定，第一个切点是代码清单 7-35 所定义的流程切点，而另一个切点是方法名为“greetTo”的方法名切点。

当然，用户也可以只通过一个切点描述同时满足以上两个匹配条件的连接点，而更好的方法是使用 Spring 提供的 ComposablePointcut 把两个切点组合起来，通过切点的复合运算表示。ComposablePointcut 可以将多个切点以并集或交集的方式组合起来，提供了切点之间复合运算的功能。

ComposablePointcut 本身也是一个切点，它实现了 Pointcut 接口。下面先来了解一下 ComposablePointcut 的构造函数。

- ComposablePointcut()：构造一个匹配所有类所有方法的复合切点。
- ComposablePointcut(ClassFilter classFilter)：构造一个匹配特定类所有方法的复合切点。
- ComposablePointcut(MethodMatcher methodMatcher)：构造一个匹配所有类特定方法的复合切点。
- ComposablePointcut(ClassFilter classFilter, MethodMatcher methodMatcher)：构造一个匹配特定类特定方法的复合切点。

ComposablePointcut 提供了 3 个交集运算的方法。

- ComposablePointcut intersection(ClassFilter filter)：将复合切点和一个 ClassFilter 对象进行交集运算，得到一个结果复合切点。
- ComposablePointcut intersection(MethodMatcher mm)：将复合切点和一个 MethodMatcher 对象进行交集运算，得到一个结果复合切点。
- ComposablePointcut intersection(Pointcut other)：将复合切点和一个切点对象进行交集运算，得到一个结果复合切点。

ComposablePointcut 提供了两个并集运算的方法。

- ComposablePointcut union(ClassFilter filter)：将复合切点和一个 ClassFilter 对象进行并集运算，得到一个结果复合切点。

- ComposablePointcut union(MethodMatcher mm)：将复合切点和一个 MethodMatcher 对象进行交并集运算，则得到一个结果复合切点。

ComposablePointcut 没有提供直接对两个切点进行交并集运算的方法，如果需要对两个切点进行交并集运算，可以使用 Spring 提供的 org.springframework.aop.support.Pointcuts 工具类，该工具类中有两个非常好用的静态方法。

- Pointcut intersection(Pointcut a, Pointcut b)：对两个切点进行交集运算，返回一个结果切点，该切点即 ComposablePointcut 对象的实例。
- Pointcut union(Pointcut a, Pointcut b)：对两个切点进行并集运算，返回一个结果切点，该切点即 ComposablePointcut 对象的实例。

下面通过 ComposablePointcut 创建一个流程切点和方法名切点的相交切点，程序代码如代码清单 7-37 所示。

代码清单 7-37　GreetingComposablePointcut

```
package com.smart.advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.support.ComposablePointcut;
import org.springframework.aop.support.ControlFlowPointcut;
import org.springframework.aop.support.NameMatchMethodPointcut;
public class GreetingComposablePointcut {
   public Pointcut getIntersectionPointcut(){
       ComposablePointcut cp = new ComposablePointcut();//①
       Pointcut pt1 = new ControlFlowPointcut(WaiterDelegate.class,"service");//②
       NameMatchMethodPointcut pt2 = new NameMatchMethodPointcut();//③
       pt2.addMethodName("greetTo");
       return cp.intersection(pt1).intersection(pt2); //④
   }
}
```

创建一个复合切点

创建一个流程切点

创建一个方法名切点

将两个切点进行交集操作

通过 GreetingComposablePointcut#getIntersectionPointcut()方法，即可得到一个相交的复合切点。配置复合切点的切面和配置其他切点一样，如代码清单 7-38 所示。

代码清单 7-38　配置复合切点切面

```
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="gcp" class="com.smart.advisor.GreetingComposablePointcut" />
<bean id="composableAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor"
    p:pointcut-="#{gcp.intersectionPointcut}" ①
    p:advice-ref="greetingAdvice"/>
<bean id="waiter4" class="org.springframework.aop.framework.ProxyFactoryBean"
   p:interceptorNames="composableAdvisor" ②
   p:target-ref="waiterTarget"
   p:proxyTargetClass="true"/>
```

引用 gcp.getIntersectionPointcut() 方法所返回的复合切点

使用复合切点切面

在①处使用 util 命名空间的标签引用另一个 Bean 的属性（关于 util 命名空间标签的详细信息，请参看 5.4.6 节）。

下面编写相应的测试代码：

```
String configPath = "com/smart/advisor/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter4");
WaiterDelegate wd = new WaiterDelegate();
wd.setWaiter(waiter);
waiter.serveTo("Peter");
waiter.greetTo("Peter");
wd.service("Peter");
```

运行以上代码，输出以下信息：

```
waiter serving Peter... ①   ← 对应 waiter.serveTo("Peter")
waiter greet to Peter... ②  ← 对应 waiter.greetTo("Peter")
com.smart.advisor.Waiter.greetTo ③  ← 对应通过 wd.service("Peter") 调用的 Waiter.greetTo()方法
How are you! Mr.Peter.
waiter greet to Peter...    ← 对应通过 wd.service("Peter") 调用的 Waiter.serveTo()方法
waiter serving Peter... ④
```

通过以上输出信息发现，只有通过 WaiterDelegate#service()方法调用 Waiter#greetTo()方法时才织入了增强，而这正是复合交集切点所描述的接连点。

7.4.8 引介切面

7.3.6 节介绍了引介增强的知识，引介切面是引介增强的封装器，通过引介切面，可以更容易地为现有对象添加任何接口的实现。图 7-9 是引介切面的类继承关系图。

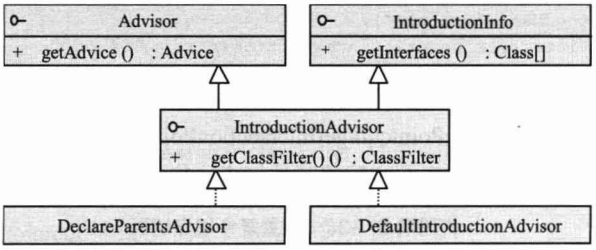


图 7-9 引介切面类继承关系图

从图 7-9 中可以看出，IntroductionAdvisor 接口同时继承 Advisor 和 IntroductionInfo 接口，IntroductionInfo 接口描述了目标类需要实现的新接口。IntroductionAdvisor 和 PointcutAdvisor 接口不同，它仅有一个类过滤器 ClassFilter 而没有 MethodMatcher，这是因为引介切面的切点是类级别的，而 Pointcut 的切点是方法级别的。

IntroductionAdvisor 有两个实现类，分别是 DefaultIntroductionAdvisor 和 DeclareParentsAdvisor，前者是引介切面最常用的实现类，后者用于实现使用 AspectJ 语言的 DeclareParent 注解表示的引介切面。

DefaultIntroductionAdvisor 拥有 3 个构造函数。

- DefaultIntroductionAdvisor(Advice advice)：通过一个增强创建的引介切面，引介切面将为目标对象新增增强对象中所有接口的实现。
- DefaultIntroductionAdvisor(DynamicIntroductionAdvice advice, Class clazz)：通过一个增强和一个指定的接口类创建引介切面，仅为目标对象新增 clazz 接口的实现。
- DefaultIntroductionAdvisor(Advice advice, IntroductionInfo introductionInfo)：通过一个增强和一个 IntroductionInfo 创建引介切面，目标对象需要实现哪些接口由 introductionInfo 对象的 getInterfaces()方法表示。

下面通过 DefaultIntroductionAdvisor 为代码清单 7-22 中的引介增强配置切面，会发现这种方式比前面的方式要更简洁、更清晰，如代码清单 7-39 所示。

代码清单 7-39　配置引介切面

```
<bean id="introduceAdvisor" class="org.springframework.aop.support.
DefaultIntroductionAdvisor">
   <constructor-arg> ①                    ControllablePerformanceMonitor 是一个 Advice 对象
     <bean class="com.smart.introduce.ControllablePerformanceMonitor"/>
   </constructor-arg>
</bean>
<bean id="forumServiceTarget" class="com.smart.introduce.ForumService"/>
<bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
      p:interceptorNames="introduceAdvisor"
      p:target-ref="forumServiceTarget"
      p:proxyTargetClass="true"/>
```

虽然引介切面和其他切面有很大的不同，但却可以采用相似的 Spring 配置方式配置引介切面。通过这种方式配置的代码在本质上和代码清单 7-23 中配置的效果是一样的。

7.5　自动创建代理

在前面所有的例子中，都通过 ProxyFactoryBean 创建织入切面的代理，每个需要被代理的 Bean 都需要使用一个 ProxyFactoryBean 进行配置，虽然可以使用父子<bean>进行改造，但还是很麻烦。对于小型系统，可以将就使用，但对于由拥有众多需要代理 Bean 的系统，原来的配置显然不尽如人意。

幸运的是，Spring 提供了自动代理机制，让容器自动生成代理，把开发人员从烦琐的配置工作中解放出来。在内部，Spring 使用 BeanPostProcessor 自动完成这项工作。

7.5.1　实现类介绍

这些基于 BeanPostProcessor 的自动代理创建器的实现类，将根据一些规则自动在容

器实例化 Bean 时为匹配的 Bean 生成代理实例。这些代理创建器可以分为 3 类。

- 基于 Bean 配置名规则的自动代理创建器：允许为一组特定配置名的 Bean 自动创建代理实例的代理创建器，实现类为 BeanNameAutoProxyCreator。
- 基于 Advisor 匹配机制的自动代理创建器：它会对容器中所有的 Advisor 进行扫描，自动将这些切面应用到匹配的 Bean 中（为目标 Bean 创建代理实例），实现类为 DefaultAdvisorAutoProxyCreator。
- 基于 Bean 中 AspectJ 注解标签的自动代理创建器：为包含 AspectJ 注解的 Bean 自动创建代理实例，实现类为 AnnotationAwareAspectJAutoProxyCreator。

图 7-10 是自动代理创建器实现类的类继承图。

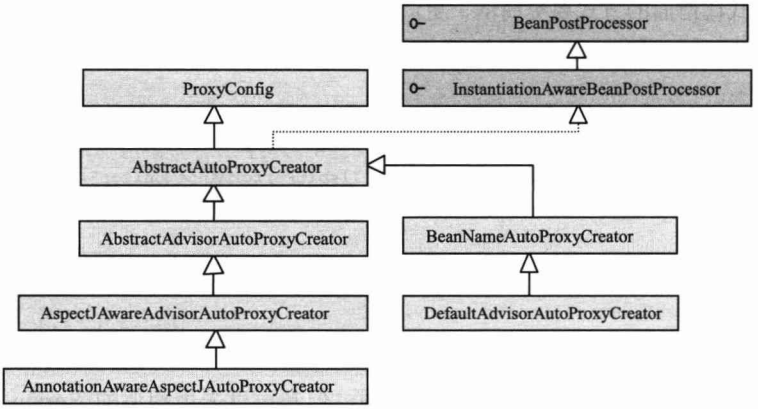


图 7-10　自动代理创建器实现类的类继承图

从图 7-10 中可以清楚地看到所有的自动代理创建器类都实现了 BeanPostProcessor，在容器实例化 Bean 时，BeanPostProcessor 将对它进行加工处理，所以，自动代理创建器有机会对满足匹配规则的 Bean 自动创建代理对象。

本章暂不涉及 AspectJ 的内容，下面只讲解 BeanNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator 的用法。

7.5.2　BeanNameAutoProxyCreator

在代码清单 7-29 中通过配置两个 ProxyFactoryBean 分别为 waiter 和 seller 的 Bean 创建代理对象。下面通过 BeanNameAutoProxyCreator 以更优雅、更便捷的方式完成相同的功能，如代码清单 7-40 所示。

代码清单 7-40　使用Bean名进行自动代理

```
<bean id="waiter" class="com.smart.advisor.Waiter" />
<bean id="seller" class="com.smart.advisor.Seller" />
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"
```

```
p:beanNames="*er" ①
p:interceptorNames="greetingAdvice"
p:optimize="true"/>
```

由于只有一个 Bean 名称，所以直接使用 value 属性进行配置，可以通过<list>子元素设定多个 Bean 名称，或通过逗号、空格、分号的方式设定多个 Bean 名称

BeanNameAutoProxyCreator 有一个 beanNames 属性，它允许用户指定一组需要自动代理的 Bean 名称，Bean 名称可以使用*通配符。假设 Spring 容器中除 waiter 和 seller 外还有其他的 Bean，如果以“er”为后缀的规则可以将这两个 Bean 和容器中其他的 Bean 区分开，就可以通过将 beanNames 属性设定为“*er”使 waiter 和 seller 这两个 Bean 被自动代理。当然，使用通配符会带来一定的风险，在例子中，假设一个其他的 Bean 名称也以“er”结尾，则自动代理创建器也会为该 Bean 创建代理。所以为了保险起见，用户可以使用下面的方式配置 beanNames 属性：value="waiter,seller"。

一般情况下不会为 FactoryBean 的 Bean 创建代理，如果刚好有这样一个需求，则需要在 beanNames 中指定添加$的 Bean 名称，如<property name="beanNames"value="$waiter "/>等。

BeanNameAutoProxyCreator 的 interceptorNames 属性指定一个或多个增强 Bean 的名称。此外，还有一个常用的 optimize 属性，如果将此属性设置为 true，则将强制使用 CGLib 动态代理技术。

通过这样的配置后，容器在创建 waiter 和 seller Bean 的实例时，就会自动为它们创建代理对象。而这一操作对于使用者来说是完全透明的，可以通过以下测试证实这一点：

```
String configPath = "com/smart/autoproxy/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter"); ①
Seller seller = (Seller) ctx.getBean("seller"); ②
waiter.greetTo("John");
seller.greetTo("Tom");
```

运行以上代码，在控制台上输出以下信息：

```
com.smart.advisor.Waiter.greetTo ①    被自动织入的增强逻辑
How are you! Mr.John.
waiter greet to John...
com.smart.advisor.Seller.greetTo ②    被自动织入的增强逻辑
How are you! Mr.Tom.
seller greet to Tom...
```

通过输出信息可以知道，从容器中返回的 waiter Bean 的 greetTo()方法及 seller Bean 的 greetTo()方法都被织入了增强，可见①处返回的 waiter Bean 和②处返回的 seller Bean 都是被代理过的对象。

7.5.3　DefaultAdvisorAutoProxyCreator

我们知道切面 Advisor 是切点和增强的复合体，Advisor 本身已经包含了足够的信息，如横切逻辑（要织入什么）及连接点（织入哪里）。

DefaultAdvisorAutoProxyCreator 能够扫描容器中的 Advisor，并将 Advisor 自动织入匹配的目标 Bean 中，即为匹配的目标 Bean 自动创建代理。

在代码清单 7-30 中通过 ProxyFactoryBean 为 waiter 配置了代理，在这里，引入 DefaultAdvisorAutoProxyCreator 为容器中所有带“greet”方法名的目标 Bean 自动创建代理。

```
<bean id="waiter" class="com.smart.advisor.Waiter" />
<bean id="seller" class="com.smart.advisor.Seller" />
<bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
<bean id="regexpAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
      p:patterns=".*greet.*"
      p:advice-ref="greetingAdvice" />

<bean class="org.springframework.aop.framework.autoproxy.
            DefaultAdvisorAutoProxyCreator"/>①
```

在①处，用 DefaultAdvisorAutoProxyCreator 定义了一个 Bean，它负责将容器中的 Advisor 织入匹配的目标 Bean 中。通过下面的测试代码检测一下自动代理创建器是否正常工作：

```
String configPath = "com/smart/autoproxy/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter waiter = (Waiter) ctx.getBean("waiter");
Seller seller = (Seller) ctx.getBean("seller");
waiter.serveTo("John");
waiter.greetTo("John");
seller.greetTo("Tom");
```

运行以上代码，输出以下信息：

```
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
com.smart.advisor.Seller.greetTo
How are you! Mr.Tom.
seller greet to Tom...
```

Waiter#serveTo()方法没有被织入增强，而 Waiter 和 Seller 的 greetTo()方法都被织入了增强，由此可见，增强被正确地织入匹配的接连点中。

7.5.4 AOP 无法增强疑难问题剖析

大家在使用 Spring AOP 时，或多或少会碰到一些方法无法被增强的问题，有时同一个类里面的方法有的可以被增强，有的却无法被增强。要分析其原因，首先要从 Spring AOP 的实现机制入手。从上文 Spring AOP 基础知识的学习可以知道，AOP 底层实现有两种方法：一种是基于 JDK 动态代理；另一种是基于 CGLib 动态代理。

在 JDK 动态代理中通过接口来实现方法拦截，所以必须确保要拦截的目标方法在接口中有定义，否则将无法实现拦截。

在 CGLib 动态代理中通过动态生成代理子类来实现方法拦截，所以必须确保要拦截的目标方法可被子类访问，也就是目标方法必须定义为非 final，则非私有实例方法。

是否注意了上面几点，就可以万事大吉了？要回答这个问题，先来看一个实例。首先对 7.5.3 节的示例进行简单的修改，如代码清单 7-41 所示。

代码清单 7-41　增强问题测试

```
public class AopAwareTest {
    @Test
    public void autoProxy() {
        String configPath = "com/smart/autoproxy/beans-aware.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter");
        waiter.serveTo("John");
        waiter.greetTo("John"); //①
    }
}
```

在这里，引入 DefaultAdvisorAutoProxyCreator 为容器中所有带“To”方法名的目标 Bean 自动创建代理，如代码清单 7-42 所示。

代码清单 7-42　beans-aware.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
…
http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.smart.aop" />
    <bean id="waiter" class="com.smart.advisor.Waiter" />
    <bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
    <!--通过Advisor自动创建代理-->
    <bean id="regexpAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
        p:patterns=".*To.*" p:advice-ref="greetingAdvice" />
    <bean
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAuto
      ProxyCreator"
        p:proxyTargetClass="true" />
</beans>
```

为了方便测试，这里只保留服务生。运行以上代码，输出以下信息：

```
com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
```

从运行结果来看，Waiter#serveTo()和 Waiter#greetTo()方法都被织入了增强，由此可见增强被正确地织入匹配的接连点中。现在改造一下 Waiter#serveTo()方法，让 Waiter#serveTo()方法内部直接调用 Waiter#greetTo()方法，并把代码清单 7-41 中①处的方法注释掉。模拟测试在一个类中一个可织入的方法 serveTo()调用另一个可织入的方法 greetTo()时，这两个方法是否都可以被织入增强，如代码清单 7-43 所示。

代码清单 7-43　增强问题测试

```
public class AopAwareTest {
    @Test
    public void autoProxy() {
        String configPath = "com/smart/autoproxy/beans-aware.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter) ctx.getBean("waiter");
        waiter.serveTo("John");
    }
}

public class Waiter {
    ...
    public void serveTo(String name){
        System.out.println("waiter serving "+name+"...");
        greetTo(name);
    }

    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
}
```

运行以上代码，输出以下信息：

```
com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
waiter greet to John...
```

从运行结果来看，只有 Waiter#serveTo()方法被织入了增强，而 Waiter#greetTo()方法没有被织入增强。原本希望这两个方法都被织入增强，但现在只有从外部调用的方法 serveTo()被增强，在同一个类的内部方法之间调用的方法 greetTo()无法被增强。笔者在实际的项目中也碰到过这个问题，经过反复的测试研究发现，原来在方法内部之间调用的时候，不会使用被增强的代理类，而是直接调用未被增强原类的方法，这也就是方法 greetTo()无法被织入增强的原因。

在实际项目中存在内部两个方法调用的问题，同时又希望它们都能够被增强，例如在一个 Service 类的两个方法 bus1()和 bus2()中都使用了缓存注解@Cacheable，也就是希望这两个方法都启用缓存。如果在业务处理过程中，bus2()方法只被 bus1()所调用，则 bus2()方法永远无法启用缓存，这在无形中给系统留下了隐患，而且不易被发现。

笔者为此研究了一套解决办法，就是想办法在内部方法调用时，让其通过代理类调用内部的方法。因此，需要让原来的 Waiter 实现一个可注入自身代理类的接口 BeanSelfProxyAware，如代码清单 7-44 所示。

代码清单 7-44　BeanSelfProxyAware接口

```
public interface BeanSelfProxyAware {
   void setSelfProxy(Object object); //织入自身代理类接口
}
```

有了这个代理类接口之后，需要对所有实现了 BeanSelfProxyAware 接口的 Bean 执

行自身代理 Bean 的注入，设计一个可复用的注入装配器 BeanSelfProxyAwareMounter，如代码清单 7-45 所示。

代码清单 7-45　BeanSelfProxyAwareMounter

```
package com.smart.aop;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
…
@Component
public class BeanSelfProxyAwareMounter implements SystemBootAddon,
ApplicationContextAware {
   private Logger logger = LoggerFactory.getLogger(this.getClass());
   private ApplicationContext applicationContext;

   //①注入Spring容器
   public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
      this.applicationContext = applicationContext;
   }

   public void onReady() { //②实现系统启动器接口中的装配就绪方法


      //②-1从容器中获取所有注入的自动代理Bean
      Map<String, BeanSelfProxyAware> proxyAwareMap =
               applicationContext.getBeansOfType(BeanSelfProxyAware.class);
      if(proxyAwareMap!=null){
         for (BeanSelfProxyAware beanSelfProxyAware : proxyAwareMap.values()) {
            beanSelfProxyAware.setSelfProxy(beanSelfProxyAware);
            if (logger.isDebugEnabled()) {
               logger.debug("{}注册自身被代理的实例。");
            }
         }
      }
   }

   public int getOrder() {
      return Ordered.HIGHEST_PRECEDENCE;
   }
}
```

在①处通过实现 ApplicationContextAware#setApplicationContext()接口方法来注入 Spring 容器上下文。在②处实现了系统启动器 SystemBootAddon#onReady()接口方法，此接口方法在下文中进行详细说明。这里的实现逻辑是从 Spring 容器中获取所有实现自动代理织入接口 BeanSelfProxyAware 的 Bean，循环迭代遍历这些 Bean，并调用 setSelfProxy()方法将自身代理类注入自身。

接下来需要编写一个系统所有组件都装载完成后、准备就绪前调用的插件接口 SystemBootAddon，用于在 Spring 容器启动完成之后触发调用注入装配器 BeanSelfProxyAwareMounter，如代码清单 7-46 所示。

代码清单 7-46　SystemBootAddon接口

```
public interface SystemBootAddon extends Ordered{

    //在系统就绪后调用的方法
    void onReady();
}
```

这个接口比较简单，只设计了一个接口方法 onReady()，用于在系统就绪后调用。同时实现了 Bean 加载顺序接口 Ordered，在插件中可以实现 Ordered#getOrder 方法，返回一个整型数字来指定插件的执行顺序。值越小优先被加载处理，值越大最后被加载处理。

最后需要设置一个启动管理器，告诉 Spring 什么时候触发 BeanSelfProxyAwareMounter 装配器，如代码清单 7-47 所示。

代码清单 7-47　设置启动管理器

```
...
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.core.OrderComparator;
import org.springframework.stereotype.Component;
import org.springframework.util.Assert;


@Component
public class SystemBootManager implements ApplicationListener<ContextRefreshedEvent> {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private List<SystemBootAddon> systemBootAddons = Collections.EMPTY_LIST;
    private boolean hasRunOnce = false;

    //①注入所有SystemBootAddon插件
    @Autowired(required = false)
    public void setSystemBootAddons(List<SystemBootAddon> systemBootAddons) {
        Assert.notEmpty(systemBootAddons);
        OrderComparator.sort(systemBootAddons);
        this.systemBootAddons =systemBootAddons;
    }

    //②触发所有插件
    public void onApplicationEvent(ContextRefreshedEvent event) {
        if (!hasRunOnce) {
            for (SystemBootAddon systemBootAddon : systemBootAddons) {
                systemBootAddon.onReady();
                if (logger.isDebugEnabled()) {
                    logger.debug("执行插件:{}",systemBootAddon.getClass().getCanonical
Name());
                }
            }
            hasRunOnce = true;
        }else{
            if (logger.isDebugEnabled()) {
                logger.debug("已执行过容器启动插件集，本次忽略之。");
            }
```

```
        }
    }
}
```

在①处通过自动注入方式注入所有实现 SystemBootAddon 接口的插件，在②处通过监听 Spring 容器的 ContextRefreshedEvent 事件调用容器中所有注册的 SystemBootAddon 插件。至此，设计的注入 Bean 自身代理类的通用型插件已经全部完成，接下来只需在需要注入自身代理类的 Bean 中实现 BeanSelfProxyAware 接口即可，如本示例中的 **Waiter** 类。下面对代码清单 7-43 中的 Waiter 类进行简单改造，如代码清单 7-48 所示。

代码清单 7-48　增强问题测试

```
...
public class Waiter implements BeanSelfProxyAware {
    private Waiter waiter;

    public void setSelfProxy(Object object) {
        waiter = (Waiter)object;
    }

    public void serveTo(String name){
        System.out.println("waiter serving "+name+"...");
        greetTo(name);
    }

    public void greetTo(String name) {
        System.out.println("waiter greet to "+name+"...");
    }
}
```

重新运行代码清单 7-43 中的 AopAwareTest 测试，输出以下信息：

```
com.smart.advisor.Waiter.serveTo
How are you! Mr.John.
waiter serving John...
com.smart.advisor.Waiter.greetTo
How are you! Mr.John.
waiter greet to John...
```

从运行结果来看，Waiter#serveTo()和 Waiter#greetTo()方法都被织入了增强，由此可见设计的注入 Bean 自身代理类的通用型插件有效。如果读者在实际项目中遇到同样的问题，则可以借鉴或使用笔者设计的这个插件。

如果不想使用上述方式，则也可以考虑使用 AspectJ，因为其支持编译期织入且不需要生成代理类，也就避免了因生成代理类产生的一些制约（static 和 final 方法不能被覆盖等问题）。关于 AspectJ 的用法，将在下一章进行详细讲解。

7.6　小结

AOP 是 OOP 的延伸，它为程序开发提供了一个崭新的思考角度，可以将重复性的

横切逻辑抽取到统一的模块中。通过 OOP 的纵向抽象和 AOP 的横向抽取，程序才可以真正解决重复性代码问题。

Spring 采用 JDK 动态代理和 CGLib 动态代理技术在运行期织入增强，所以不需要装备特殊的编译器或类装载器就可以使用 AOP 的功能。要使用 JDK 动态代理，目标类必须实现接口，而 CGLib 不对目标类作任何限制，它通过动态生成目标类子类的方式提供代理。JDK 在创建代理对象时的性能高于 CGLib，而生成的代理对象的运行性能却比 CGLib 的低。如果是 singleton 的代理，则推荐使用 CGLib 动态代理。

Spring 只能在方法级别上织入增强，Spring 提供了 4 种类型的方法增强，分别是前置增强、后置增强、环绕增强和异常抛出增强，此外还有一种特殊的引介增强。引介增强是类级别的，它为目标类织入新的接口实现。从广义上说，增强其实就是一种最简单的切面，它既包括横切代码又包括切点信息，只不过它的切点只是简单的方法相对位置信息。所以增强一般需要和切点联合才可以表示一个更具实用性的切面。

在 Spring 中，普通的切点通过目标类名和方法名描述连接点的信息。流程切点是比较特殊的切点，它通过方法调用堆栈的运行环境信息来决定连接点。有时需要通过切点的交叉或合并描述一个最终的切点，这时可以使用 ComposablePointcut 的复合切点。

切面是增强和切点的联合体，可以很方便地通过 Spring 提供的 ProxyBeanFactory 将切面织入不同的目标类中。当然，为每个目标类手工配置一个切面是比较烦琐的，Spring 利用 BeanPostProcessor 可干涉 Bean 生命周期的机制，提供了一些可以自动创建代理、织入切面的自动代理创建器，其中 DefaultAdvisorAutoProxyCreator 是功能强大的自动代理创建器，它可以将容器中的所有 Advisor 自动织入目标 Bean 中。

第 8 章

基于@AspectJ 和 Schema 的 AOP

在上一章的学习中我们发现，在 Spring 中定义一个切面是比较烦琐的，需要实现专门的接口，并进行一些较为复杂的配置，Spring AOP 的配置是被批评最多的地方。Spring 听到了这方面的声音，下决心解决这一问题，并取得了很好的突破。如今，Spring AOP 已经焕然一新，用户可以使用@AspectJ 注解非常容易地定义一个切面，而不需要实现任何接口。对于没有使用 Java 5.0 的项目，可以通过基于 Schema 的配置定义切面，其方便程度和基于@AspectJ 注解的配置相差无几。此外，Spring 还可以使用 AspectJ 语言编写切面，发扬二者的长处：用 AspectJ 织入切面，并让 Spring 容器管理这些切面。

本章主要内容：

- Java 5.0 注解知识
- 通过@AspectJ 定义切面
- 切点函数讲解
- 绑定连接点参数
- 基于 Schema 配置定义切面
- Spring LTW

本章亮点：

- 对切点表达式函数进行深入分析
- 深入讲解集成 AspectJ 的过程

8.1 Spring 对 AOP 的支持

Spring 在新版本中对 AOP 功能进行了重要的增强，主要表现在以下几个方面：

- 新增了基于 Schema 的配置支持，为 AOP 专门提供了 aop 命名空间。
- 新增了对 AspectJ 切点表达式语言的支持。@AspectJ 是 AspectJ 1.5 新增的功能，它通过 Java 5.0 的注解技术，允许开发者在 POJO 中定义切面。Spring 使用和@AspectJ 相同风格的注解，并通过 AspectJ 提供的注解库和解析库处理切点。当然，由于 Spring 只支持方法级的切点，所以仅对@AspectJ 提供了有限的支持。
- 可以无缝地集成 AspectJ。AspectJ 提供了语言级切面的实现，Spring 无意开发一个重复的东西，Spring 对开源世界里一切优秀的东西向来采取兼收并蓄的态度。

这里所说的 Spring AOP 包括基于 XML 配置的 AOP 和基于@AspcetJ 注解的 AOP，这两种方法虽然在配置切面时的表现方式不同，但底层都采用了动态代理技术（JDK 或 CGLib 动态代理）。Spring 可以集成 AspectJ，但 AspectJ 本身并不属于 Spring AOP 的范畴。

一般情况下，对于开发 JAVA EE 企业应用的开发者而言，Spring AOP 已经可以满足使用的要求。虽然 AspectJ 提供对 AOP 更为细致的实现，但像实例化切面、属性访问切面、条件切面等功能，在实际应用中并不常用。

如果是基于 Java 5.0 的项目，推荐使用 Spring 提供的@AspsctJ 配置方式，因为它能以更简单、更直接的方式应用切面。

8.2 Java 5.0 注解知识快速进阶

在进入本章学习之前，有必要对 Java 5.0 新增的注解（Annotation）技术进行简单的学习。因为 Spring 支持@AspectJ，而@AspectJ 本身就是基于 Java 5.0 的注解技术，所以学习 Java 5.0 的注解知识有助于更好地理解和掌握 Spring 的 AOP 技术。

8.2.1 了解注解

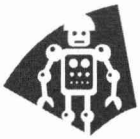
对于 Java 开发人员来说，在编写代码时，除源程序外，还会使用 Javadoc 标签对类、方法或成员变量进行注释，以便使用 Javadoc 工具生成和源码配套的 Javadoc 文档。这些@param、@return 等 Javadoc 标签就是注解标签，它们为第三方工具提供了描述程序代码的注释信息。使用过 Xdoclet 的读者对此更有感触，像 Struts、Hibernate 都提供了 Xdoclet 标签，使用它们可以快速地生成对应程序代码的配置文件。

Java 5.0 注解可以看作 Javadoc 和 Xdoclet 标签的延伸与发展。在 Java 5.0 中可以自定义这些标签，并通过 Java 语言的反射机制获取类中标注的注解，完成特定的功能。

注解是代码的附属信息，它遵循一个基本原则：注解不能直接干扰程序代码的运行，无论增加或删除注解，代码都能够正常运行。Java 语言解释器会忽略这些注解，而由第三方工具负责对注解进行处理。第三方工具可以利用代码中的注解间接控制程序代码的

运行，它们通过 Java 反射机制读取注解的信息，并根据这些信息更改目标程序的逻辑，而这正是 Spring AOP 对@AspectJ 提供支持所采取的方法。

轻松一刻



很多事物的设计都必须遵循最基本的原则。为了防止机器人伤害人类，科幻作家阿西莫夫于 1940 年提出了“机器人三原则”：第一，机器人不能伤害人类；第二，机器人应遵守人类的命令，与第一条违背的命令除外；第三，机器人应能保护自己，与第一条违背的命令除外。这是给机器人赋予的伦理性纲领，机器人学术界一直将这 3 条原则作为机器人开发的准则。

8.2.2　一个简单的注解类

通常情况下，第三方工具不但负责处理特定的注解，其本身还提供了这些注解的定义，所以通常仅需关注如何使用注解即可。但定义注解类本身并不困难，Java 提供了定义注解的语法。下面着手编写一个简单的注解类，如代码清单 8-1 所示。

代码清单 8-1　NeedTest注解类

```
package com.smart.aspectj.anno;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) //①声明注解的保留期限
@Target(ElementType.METHOD)//②声明可以使用该注解的目标类型
public @interface NeedTest {//③定义注解
    boolean value() default true;//④声明注解成员
}
```

Java 新语法规定使用@interface 修饰符定义注解类，如③处所示。一个注解可以拥有多个成员，成员声明和接口方法声明类似，这里仅定义了一个成员，如④处所示。成员声明有以下几点限制：

- 成员以无入参、无抛出异常的方式声明，如 boolean value(String str)、boolean value() throws Exception 等方式是非法的。
- 可以通过 default 为成员指定一个默认值，如 String level() default "LOW_LEVEL"、int high() default 2 是合法的，当然也可以不指定默认值。
- 成员类型是受限的，合法的类型包括原始类型及其封装类、String、Class、enums、注解类型，以及上述类型的数组类型，如 ForumService value()、List foo()是非法的。

在①和②处所看到的注解是 Java 预定义的注解，称为元注解（Meta-Annotation），它们被 Java 编译器使用，会对注解类的行为产生影响。@Retention(RetentionPolicy.

RUNTIME)表示 NeedTest 这个注解可以在运行期被 JVM 读取，注解的保留期限类型在 java.lang.annotation.Retention 类中定义，介绍如下。

- SOURCE：注解信息仅保留在目标类代码的源码文件中，但对应的字节码文件将不再保留。
- CLASS：注解信息将进入目标类代码的字节码文件中，但类加载器加载字节码文件时不会将注解加载到 JVM 中，即运行期不能获取注解信息。
- RUNTIME：注解信息在目标类加载到 JVM 后依然保留，在运行期可以通过反射机制读取类中的注解信息。

Target(ElementType.METHOD)表示 NeedTest 这个注解只能应用到目标类的方法上，注解的应用目标在 java.lang.annotation.ElementType 类中定义，介绍如下。

- TYPE：类、接口、注解类、Enum 声明处，相应的注解称为类型注解。
- FIELD：类成员变量或常量声明处，相应的注解称为域值注解。
- METHOD：方法声明处，相应的注解称为方法注解。
- PARAMETER：参数声明处，相应的注解称为参数注解。
- CONSTRUCTOR：构造函数声明处，相应的注解称为构造函数注解。
- LOCAL_VARIABLE：局部变量声明处，相应的注解称为局域变量注解。
- ANNOTATION_TYPE：注解类声明处，相应的注解称为注解类注解，ElementType.TYPE 包括 ElementType.ANNOTATION_TYPE。
- PACKAGE：包声明处，相应的注解称为包注解。

如果注解只有一个成员，则成员名必须取名为 value()，在使用时可以忽略成员名和赋值号（=），如@NeedTest(true)。当注解类拥有多个成员时，如果仅对 value 成员进行赋值，则也可不使用赋值号；如果同时对多个成员进行赋值，则必须使用赋值号，如 DeclareParents (value = "NaiveWaiter", defaultImpl = SmartSeller.class)。注解类可以没有成员，没有成员的注解称为标识注解，解释程序以标识注解存在与否进行相应的处理；此外，所有的注解类都隐式继承于 java.lang.annotation.Annotation，但注解不允许显式继承于其他的接口。

我们希望使用 NeedTest 注解对业务类的方法进行标注，以便测试工具可以根据注解情况激活或关闭对业务类的测试。在编写好 NeedTest 注解类后，就可以在其他类中使用它了。

8.2.3 使用注解

在 ForumService 中使用 NeedTest 注解，标注业务方法是否需要测试，如代码清单 8-2 所示。

代码清单 8-2　ForumService：使用注解

```
package com.smart.aspectj.anno;
public class ForumService {
   @NeedTest(value=true)//①标注注解
    public void deleteForum(int forumId){
        System.out.println("删除论坛模块："+forumId);
    }

   @NeedTest(value=false) //②标注注解
   public void deleteTopic(int postId){
        System.out.println("删除论坛主题："+postId);
    }
}
```

如果注解类和目标类不在同一个包中，则需要通过 import 引用注解类。在①和②处，使用 NeedTest 分别对 deleteForum()和 deleteTopic()方法进行标注。在标注注解时，可以通过以下格式对注解成员进行赋值：

```
@<注解名>(<成员名 1>=<成员值 1>,<成员名 2>=<成员值 2>,...)
```

如果成员是数组类型，则可以通过{}进行赋值，如 boolean 数组的成员可以设置为{true,false,true}。下面是几个注解标注的例子。

示例 1：多成员的注解。

```
@AnnoExample(id= 2868724, synopsis = "Enable time-travel",
engineer = "Mr. Peabody",date = "4/1/2007")
```

示例 2：一个成员的注解，成员名为 value。可以省略成员名和赋值符号。

```
@Copyright("2011 bookegou.com All Right Reserved")
```

示例 3：无成员的注解。

```
@Override
```

示例 4：成员为字符串数组的注解。

```
@SuppressWarnings(value={"unchecked","fallthrough"})
```

示例 5：成员为注解数组类型的注解。

```
@Reviews({@Review(grade=Review.Grade.EXCELLENT,reviewer="df"),
        @Review(grade=Review.Grade.UNSATISFACTORY,reviewer="eg",
                comment="This method needs an @Override annotation")})
```

@Reviews 注解拥有一个@Review 注解数组类型的成员。@Review 注解类型有 3 个成员，其中 reviewer、comment 都是 String 类型的成员，但 comment 有默认值，而 grade 是枚举类型的成员。

由于 NeedTest 注解的保留限期是 RetentionPolicy.RUNTIME 类型，因此，当 ForumService 被加载到 JVM 时，仍可通过反射机制访问到 ForumService 各个方法的注解信息。

8.2.4　访问注解

前面提到过，注解不会直接影响程序的运行，但是第三方程序或工具可以利用代码

中的注解完成特殊的任务，间接控制程序的运行。对于 RetentionPolicy.RUNTIME 保留期限的注解，可以通过反射机制访问类中的注解。

在 Java 5.0 中，Package、Class、Constructor、Method 及 Field 等反射对象都新增了访问注解信息的方法：<T extends Annotation>T getAnnotation(Class<T> annotationClass)，该方法支持通过泛型直接返回注解对象。

下面通过反射来访问注解，得出 ForumService 类中通过@NeedTest 注解所承载的测试需求，如代码清单 8-3 所示。

代码清单 8-3　ToolTest：访问代码中的注解

```
package com.smart.aspectj.anno;
import java.lang.reflect.Method;
import org.testng.annotations.*;
public class ToolTest {
    @Test
    public void tool() {

        //①得到ForumService对应的Class对象
        Class clazz = ForumService.class;

        //②得到 ForumSerivce 对应的 Method 数组
        Method[] methods = clazz.getDeclaredMethods();
        System.out.println(methods.length);
        for (Method method : methods) {

            //③获取方法上所标注的注解对象
            NeedTest nt = method.getAnnotation(NeedTest. class);
            if (nt != null) {
                if (nt.value()) {
                    System.out.println(method.getName() + "()需要测试");
                } else {
                    System.out.println(method.getName() + "()不需要测试");
                }
            }
        }
    }
}
```

在③处通过方法的反射对象，获取了方法上所标注的 NeedTest 注解对象，接着就可以访问注解对象的成员，从而得到 ForumService 类方法的测试需求。运行以上代码，输出以下信息：

```
deleteForum()需要测试
deleteTopic()不需要测试
```

8.3　着手使用@AspectJ

第 7 章分别使用 Pointcut 和 Advice 接口描述切点和增强，并用 Advisor 整合二者描

述切面，@AspectJ 则采用注解来描述切点、增强，二者只是表述方式不同，描述内容的本质是完全相同的，这就好比一个用中文、一个用英文讲述同一则伊索寓言。

8.3.1　使用前的准备

在使用@AspectJ 之前，首先必须保证所使用的 Java 是 5.0 及以上版本，否则无法使用注解技术。

Spring 在处理@Aspect 注解表达式时，需要将 Spring 的 asm 模块添加到类路径中。asm 是轻量级的字节码处理框架，因为 Java 的反射机制无法获取入参名，Spring 就利用 asm 处理@AspectJ 中所描述的方法入参名。

此外，Spring 采用 AspectJ 提供的@AspectJ 注解类库及相应的解析类库，需要在 pom.xml 文件中添加 aspectj.weaver 和 aspectj.tools 类包的依赖。

8.3.2　一个简单的例子

在做好准备工作后，就可以开始编写一个基于@AspectJ 的切面。首先来看一个简单的例子，以便对@AspectJ 有一个切身的体会。

@AspectJ 采用不同的方式对 AOP 进行描述，依旧使用 NaiveWaiter 的例子进行讲解。为了方便阅读，再次给出 NaiveWaiter 类的代码，如下：

```
package com.smart;
public class NaiveWaiter implements Waiter {
    public void greetTo(String clientName) {
        System.out.println("NaiveWaiter:greet to "+clientName+"...");
    }
    public void serveTo(String clientName){
        System.out.println("NaiveWaiter:serving "+clientName+"...");
    }
}
```

下面使用@AspectJ 注解定义一个切面，如代码清单 8-4 所示。

代码清单 8-4　PreGreetingAspect：切面

```
package com.smart.aspectj.aspectj;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@AspectJ //①通过该注解将PreGreetingAspect标识为一个切面
public class PreGreetingAspect{

    @Before("execution(* greetTo(..))") //②定义切点和增强类型
    public void beforeGreeting(){ //③增强的横切逻辑
        System.out.println("How are you");
    }
}
```

我们“惊奇”地发现这个切面没有实现任何特殊的接口，它只是一个普通的 POJO。它特殊的地方在于使用了@AspectJ 注解。

首先，在 PreGreetingAspect 类定义处标注了@AspectJ 注解，这样，第三方处理程序就可以通过类是否拥有@AspectJ 注解判断其是否为一个切面，如①处所示。

其次，在 beforeGreeting()方法定义处标注了@Before 注解，并为该注解提供了成员值“execution(* greetTo(..))”，如②处所示。②处的注解提供了两个信息：@Before 注解表示该增强是前置增强，而成员值是一个@ApsectJ 切点表达式。它的意思是：在目标类的 greetTo()方法上织入增强，greetTo()方法可以带任意的入参和任意的返回值。

最后，在③处的 beforeGreeting()方法是增强所使用的横切逻辑，该横切逻辑在目标方法前调用，可以通过图 8-1 描述这种关系。

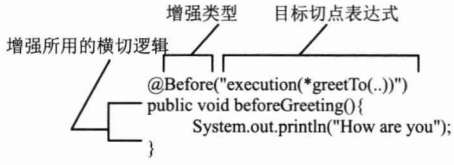


图 8-1　切面的信息构成

PreGreetingAspect 类通过注解和代码，将切点、增强类型和增强的横切逻辑糅合到一个类中，使切面的定义浑然天成。PreGreetingAspect 类相当于第 7 章中 BeforeAdvice、NameMatchMethodPointcut 及 DefaultPointcutAdvisor 三者联合表达的信息，在掌握@AspectJ 定义切面的技术后，再去使用第 7 章中基于接口的切面技术，一定会顿生“马后桃花马前雪”的感慨。不过，通过基于接口切面的学习可以更加深刻地了解 Spring AOP 的内核技术。

下面通过 AspectJProxyFactory 为 NaiveWaiter 生成织入 PreGreetingAspect 切面的代理，如代码清单 8-5 所示。

代码清单 8-5　AspectJProxyTest

```
package com.smart.aspectj.example;
import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;
import com.smart.NaiveWaiter;
import com.smart.Waiter;
public class AspectJProxyTest {
Waiter target = new NaiveWaiter();
        AspectJProxyFactory factory = new AspectJProxyFactory();

       //①设置目标对象
       factory.setTarget(target);

        //②添加切面类
        factory.addAspect(PreGreetingAspect.class);

        //③生成织入切面的代理对象
        Waiter proxy = factory.getProxy();
```

```
        proxy.greetTo("John");
        proxy.serveTo("John");
    }
}
```

第 7 章通过 ProxyFactory 织入基于接口描述的切面，此处则使用 AspectJProxyFactory 织入基于@AspectJ 的切面。在①处设置了目标对象；在②处添加了一个切面类，该类必须是带@AspectJ 注解的类；在③处就可以获取织入了切面的代理对象。

接下来直接通过代理对象调用 greetTo()和 serveTo()方法，将产生以下输出信息：

```
How are you ①
greet to John...
serving John...
```

通过①处的输出信息，可以知道代理对象的 greetTo()方法已经被织入了切面类所定义的增强逻辑。

细心的读者可能会发现，在 PreGreetingAspect 切面类中定义的前置增强方法和通过 BeforeAdvice 接口定义的前置增强方法不一样，一个明显的区别就是 PreGreetingAspect 切面类的 beforeGreeting()方法没有任何入参，而 BeforeAdvice 的接口方法 before(Method method, Object[] args, Object obj)的入参提供了一条访问目标对象方法和入参的途径。难道使用@AspectJ 定义的切面就没有办法访问目标对象连接点的信息了吗？当然不是，在本章后面的内容中，读者将学到相关的知识。

8.3.3　如何通过配置使用@AspectJ 切面

虽然可以通过编程的方式织入切面，但在一般情况下，都是通过 Spring 的配置完成切面织入工作的。

```
<!-- ①目标Bean -->
<bean id="waiter" class="com.smart.NaiveWaiter" />
<!-- ②使用了@AspectJ注解的切面类 -->
<bean class="com.smart. aspectj.example. PreGreetingAspect" />
<!-- ③自动代理创建器，自动将@AspectJ注解切面类织入目标Bean中-->
<bean class="org.springframework.aop.aspectj.annotation.
        AnnotationAwareAspectJAutoProxyCreator"/>
```

7.5.1 节介绍了几个自动代理创建器，其中 AnnotationAware AspectJAutoProxy Creator 能够将@AspectJ 注解切面类自动织入目标 Bean 中。在这里，PreGreetingAspect 是使用@AspectJ 注解描述的切面类，而 NaiveWaiter 是匹配切点的目标类。

如果使用基于 Schema 的 aop 命名空间进行配置，那么事情就更简单了，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"
    xmlns:aop="http://www.springframework.org/schema/aop" ①
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                        http://www.springframework.org/schema/aop ②
                        http://www.springframework.org/schema/aop/ spring-aop-4.0.xsd">
```

```
    <!--③基于@AspectJ切面的驱动器-->
    <aop:aspectj-autoproxy />
    <bean id="waiter" class="com.smart.NaiveWaiter" />
    <bean class="com.smart.aspectj.example.PreGreetingAspect" />
</beans>
```

首先在配置文件中引入 aop 命名空间，如①和②处所示；然后通过 aop 命名空间的<aop:aspectj-autoproxy/>自动为 Spring 容器中那些匹配@AspectJ 切面的 Bean 创建代理，完成切面织入。当然，Spring 在内部依旧采用 AnnotationAwareAspectJAutoProxyCreator 进行自动代理的创建工作，但具体的实现细节已经被<aop:aspectj-autoproxy />隐藏起来。

<aop:aspectj-autoproxy/>有一个 proxy-target-class 属性，默认为 false，表示使用 JDK 动态代理技术织入增强；当配置为<aop:aspectj-autoproxy proxy-target-class="true"/>时，表示使用 CGLib 动态代理技术织入增强。不过即使 proxy-target-class 设置为 false，如果目标类没有声明接口，则 Spring 将自动使用 CGLib 动态代理。

8.4 @AspectJ 语法基础

@AspectJ 使用 Java 5.0 注解和正规的 AspectJ 的切点表达式语言描述切面，由于 Spring 只支持方法的连接点，所以 Spring 仅支持部分 AspectJ 的切点语言。本节将学习 AspectJ 切点表达式语言，以 AspectJ 5.0 版本为准。

8.4.1 切点表达式函数

AspectJ 5.0 的切点表达式由关键字和操作参数组成，如切点表达式 execution (* greetTo(..))，execution 为关键字，而“* greetTo(..)”为操作参数。在这里，execution 代表目标类执行某一方法，而“* greetTo(..)”描述目标方法的匹配模式串，二者联合起来表示目标类 greetTo()方法的连接点。为了描述方便，将 execution()称作函数，而将匹配串“* greetTo(..)”称作函数的入参。

Spring 支持 9 个@ApsectJ 切点表达式函数，它们用不同的方式描述目标类的连接点。根据描述对象的不同，可以大致分为 4 种类型。

- 方法切点函数：通过描述目标类方法的信息定义连接点。
- 方法入参切点函数：通过描述目标类方法入参的信息定义连接点。
- 目标类切点函数：通过描述目标类类型的信息定义连接点。
- 代理类切点函数：通过描述目标类的代理类的信息定义连接点。

这 4 种类型的切点函数通过表 8-1 进行说明。

表 8-1　切点函数

类　别	函　数	入　参	说　明
方法切点函数	execution()	方法匹配模式串	表示满足某一匹配模式的所有目标类方法连接点。如 execution(* greetTo(..))表示所有目标类中的 greetTo()方法
	@annotation()	方法注解类名	表示标注了特定注解的目标类方法连接点。如@annotation(com.smart.anno.NeedTest)表示任何标注了@NeedTest 注解的目标类方法
方法入参切点函数	args()	类名	通过判别目标类方法运行时入参对象的类型定义指定连接点。如 args(com.smart.Waiter)表示所有有且仅有一个按类型匹配于 Waiter 入参的方法
	@args()	类型注解类名	通过判别目标类方法运行时入参对象的类是否标注特定注解来指定连接点。如@args(com.smart.Monitorable)表示任何这样的一个目标方法：它有一个入参且入参对象的类标注@Monitorable 注解
目标类切点函数	within()	类名匹配串	表示特定域下的所有连接点。如 within(com.smart.service.*)表示 com.smart.service 包中的所有连接点，即包中所有类的所有方法；而 within(com.smart.service.*Service)表示在 com.smart.service 包中所有以 Service 结尾的类的所有连接点
	target()	类名	假如目标类按类型匹配于指定类，则目标类的所有连接点匹配这个切点。如通过 target(com.smart.Waiter)定义的切点、Waiter 及 Waiter 实现类 NaiveWaiter 中的所有连接点都匹配该切点
	@within()	类型注解类名	假如目标类按类型匹配于某个类 A，且类 A 标注了特定注解，则目标类的所有连接点匹配这个切点。如@within(com.smart.Monitorable)定义的切点，假如 Waiter 类标注了@Monitorable 注解，则 Waiter 及 Waiter 实现类 NaiveWaiter 的所有连接点都匹配这个切点
	@target()	类型注解类名	假如目标类标注了特定注解，则目标类的所有连接点都匹配该切点。如@target (com.smart.Monitorable)，假如 NaiveWaiter 标注了@Monitorable，则 NaiveWaiter 的所有连接点都匹配这个切点
代理类切点函数	this()	类名	代理类按类型匹配于指定类，则被代理的目标类的所有连接点都匹配该切点。这个函数比较难以理解，这里暂不举例，留待后面详解

除了表 8-1 中所列的函数外，@AspectJ 还有 call()、initialization()、preinitialization()、staticinitialization()、get()、set()、handler()、adviceexecution()、withincode()、cflow()、cflowbelow()、if()、@this()及@withincode()等函数，这些函数在 Spring 中不能使用，否则会抛出 IllegalArgumentException 异常。在不特别声明的情况下，本书所讲的@AspectJ 函数均指表 8-1 中所列的函数。

8.4.2　在函数入参中使用通配符

有些函数的入参可以接受通配符，@AspectJ 支持 3 种通配符。

- *：匹配任意字符，但它只能匹配上下文中的一个元素。
- ..：匹配任意字符，可以匹配上下文中的多个元素，但在表示类时，必须和*联合使用，而在表示入参时则单独使用。

- +：表示按类型匹配指定类的所有类，必须跟在类名后面，如 com.smart.Car+。继承或扩展指定类的所有类，同时还包括指定类本身。

@AspectJ 函数按其是否支持通配符及支持的程度，可以为以下 3 类。

- 支持所有通配符：execution()和 within()，如 within(com.smart.*)、within(com.smart.service..*.*Service+)等。
- 仅支持“+”通配符：args()、this()和 target()，如 args(com.smart.Waiter+)、target(java.util.List+)等。虽然这 3 个函数可以支持“+”通配符，但其意义不大，因为对于这些函数来说，使用和不使用“+”都是一样的，如 target(com.smart.Waiter+)和 target(com.smart.aspectj.Waiter)是等价的。
- 不支持通配符：@args()、@within()、@target()和@annotation()，如@argscom.smart.anno.NeedTest)和@within(com.smart.anno.NeedTest)。

此外，args()、this()、target()、@args()、@within()、@target()和@annotation()这 7 个函数除了可以指定类名外，也可以指定变量名，并将目标对象中的变量绑定到增强的方法中。关于参数绑定的内容将在 8.6 节介绍，而函数的其他内容将在 8.5 节详解。

8.4.3 逻辑运算符

切点表达式由切点函数组成，切点函数之间还可以进行逻辑运算，组成复合切点。Spring 支持以下切点运算符。

- &&：与操作符，相当于切点的交集运算。如果在 Spring 的 XML 配置文件中使用切点表达式，由于&是 XML 特殊字符，所以使用转义字符&表示。为了使用方便，Spring 提供了一个等效的运算符“and”。如 within(com.smart..*) and args(String)表示在 com.smart 包下所有类（当前包及子孙包）拥有一个 String 入参的方法。
- ||：或操作符，相当于切点的并集运算，or 是等效的操作符。如 within(com. smart..*) || args(String)表示在 com.smart 包下所有类的方法，或者所有拥有一个 String 入参的方法。
- !：非操作符，相当于切点的反集运算，not 是等效的操作符。如!within(com. smart.*)表示所有不在 com.smart 包下的方法。

在标准的@AspectJ 中并不提供 and、or 和 not 操作符，它们是 Spring 为了在 XML 配置文件中方便定义切点表达式而特意添加的等价操作符。有意思的是，和一般的语法表达不一样，在 Spring 中使用 and、or 和 not 操作符时，允许不在前后添加空格，如 within(com.smart..*) andnotargs(String)和 within(com.smart..*) and not args(String)拥有相同的效果。虽然 Spring 接受这种表示方式，但为了保证程序的可读性，最好还是采用传统的习惯，在操作符的前后添加空格。

提示

如果 not 位于切点表达式的开头，则必须在开头添加一个空格，否则将产生解析错误。如“not within(com.smart..*)”将产生解析错误，这应该是 Spring 解析的一个 Bug，在表达式开头添加空格后则可以通过解析，即“␣not within (com.smart..*)”。

8.4.4　不同增强类型

第 7 章使用不同的接口描述各种增强类型，@AspectJ 也为各种增强类型提供了不同的注解类，它们位于 org.aspectj.lang.annotation.*包中。这些注解类拥有若干个成员，可以通过这些成员完成定义切点信息、绑定连接点参数等操作。此外，这些注解的存留期限都是 RetentionPolicy.RUNTIME，标注目标都是 ElementType.METHOD。下面逐一学习@AspectJ 所提供的几个增强注解。

1．@Before

前置增强，相当于 BeforeAdvice。Before 注解类拥有两个成员。

- value：该成员用于定义切点。
- argNames：由于无法通过 Java 反射机制获取方法入参名，所以如果在 Java 编译时未启用调试信息，或者需要在运行期解析切点，就必须通过这个成员指定注解所标注增强方法的参数名（注意二者名字必须完全相同），多个参数名用逗号分隔。

2．@AfterReturning

后置增强，相当于 AfterReturningAdvice。AfterReturning 注解类拥有 4 个成员。

- value：该成员用于定义切点。
- pointcut：表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值，可以将 pointcut 成员看作 value 的同义词。
- returning：将目标对象方法的返回值绑定给增强的方法。
- argNames：如前所述。

3．@Around

环绕增强，相当于 MethodInterceptor。Around 注解类拥有两个成员。

- value：该成员用于定义切点。
- argNames：如前所述。

4．@AfterThrowing

抛出增强，相当于 ThrowsAdvice。AfterThrowing 注解类拥有 4 个成员。

- value：该成员用于定义切点。
- pointcut：表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值。可以将 pointcut 成员看作 value 的同义词。

- throwing：将抛出的异常绑定到增强方法中。
- argNames：如前所述。

5. @After

Final 增强，不管是抛出异常还是正常退出，该增强都会得到执行。该增强没有对应的增强接口，可以把它看成 ThrowsAdvice 和 AfterReturningAdvice 的混合物，一般用于释放资源，相当于 try{}finally{}的控制流。After 注解类拥有两个成员。

- value：该成员用于定义切点。
- argNames：如前所述。

6. @DeclareParents

引介增强，相当于 IntroductionInterceptor。DeclareParents 注解类拥有两个成员。

- value：该成员用于定义切点，它表示在哪个目标类上添加引介增强。
- defaultImpl：默认的接口实现类。

除引介增强外，其他增强都很容易理解，将在本章后续内容中统一讲述。引介增强的使用比较特别，为此特别在 8.4.5 节准备了一个实例。

8.4.5 引介增强用法

请看以下两个接口及其实现类，如图 8-2 所示。

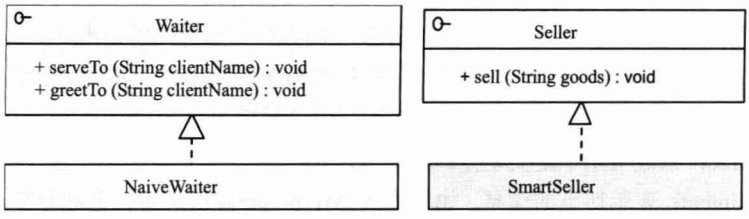


图 8-2 Waiter 和 Seller

假设希望 NaiveWaiter 能够同时充当售货员的角色，即通过切面技术为 NaiveWaiter 新增 Seller 接口的实现。在第 7 章中已经做过相似的工作，因此这里不在概念上作更多的解释，下面着手用@AspectJ 的引介增强来实现这一功能，如代码清单 8-6 所示。

代码清单 8-6 EnableSellerAspect

```
package com.smart.aspectj.basic;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;
import com.smart.Seller;
import com.smart.SmartSeller;

@Aspect
public class EnableSellerAspect {

  @DeclareParents(value="com.smart.NaiveWaiter",//①为NaiveWaiter添加接口实现
```

```
                defaultImpl=SmartSeller.class) //②默认的接口实现类
    public  Seller seller; //③要实现的目标接口
}
```

在 EnableSellerAspect 切面中，通过@DeclareParents 为 NaiveWaiter 添加了一个需要实现的 Seller 接口，并指定其默认实现类为 SmartSeller，然后通过切面技术将 SmartSeller 融合到 NaiveWaiter 中，这样 NaiveWaiter 就实现了 Seller 接口。

在 Spring 配置文件中配置好切面和 NaiveWaiter Bean，如下：

```
<aop:aspectj-autoproxy/>
<bean id="waiter" class="com.smart.NaiveWaiter"/>
<bean class="com.smart.aspectj.basic.EnableSellerAspect"/>
```

运行以下测试代码：

```
package com.smart.aspectj.basic;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.smart.Seller;
import com.smart.Waiter;
public class DeclaredParentsTest {
    public static void main(String[] args) {
        String configPath = "com/smart/aspectj/basic/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter waiter = (Waiter)ctx.getBean("waiter");
        waiter.greetTo("John");
        Seller seller = (Seller)waiter; //①可以成功地进行强制类型转换
        seller.sell("Beer", "John");
    }
}
```

代码成功执行，并输出以下信息：

```
NaiveWaiter:greet to John...
SmartSeller: sell Beer to John...
```

可见，NaiveWaiter 已经成功地新增了 Seller 接口的实现。

8.5　切点函数详解

切点函数是 AspectJ 表达式语言的核心，也是使用@AspectJ 进行切面定义的难点，本节通过具体实例来学习切点函数的相关知识。为了方便讲解，假设可供被增强的目标类包括 7 个类，这些目标类都位于 com.smart.*包中，如图 8-3 所示。

在这些类中，除 SmartSeller#showGoods()方法是 protected 外，其他的方法都是 public。

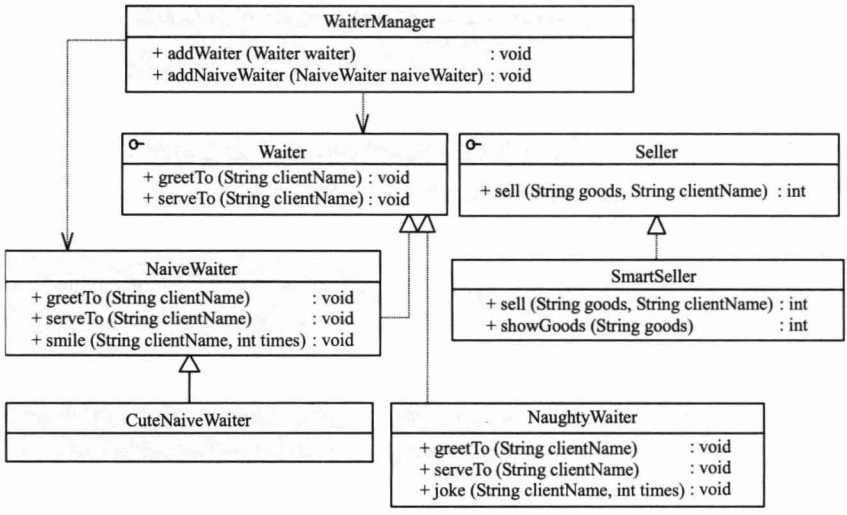


图 8-3　Waiter 和 Seller 类图

8.5.1　@annotation()

@annotation 表示标注了某个注解的所有方法。通过一个实例说明@annotation()的用法。TestAspect 定义了一个后置增强切面，该增强将应用到标注了 NeedTest 的目标方法中。

```
package com.smart.aspectj.fun;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
   @AfterReturning("@annotation(com.smart.anno.NeedTest)") //①后置增强切面
   public void needTestFun(){
      System.out.println("needTestFun() executed!");
   }
}
```

假设 NaughtyWaiter#greetTo()方法标注了@NeedTest 注解，而 NaiveWaiter#greetTo()方法没有标注@NeedTest 注解，如代码清单 8-7 所示。

代码清单 8-7　标注了@NeedTest注解的NaughtyWaiter

```
package com.smart;
import com.smart.anno.NeedTest;
public class NaughtyWaiter implements Waiter {
    @NeedTest
    public void greetTo(String clientName) {
        System.out.println("NaughtyWaiter:greet to "+clientName+"...");
    }
   ...
}
```

通过 Spring 配置自动应用切面。

```
<aop:aspectj-autoproxy />
<bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
<bean id="naughtyWaiter" class="com.smart.NaughtyWaiter" />
<bean class="com.smart.aspectj.fun.TestAspect" />
```

运行代码清单 8-8 中的代码。

代码清单 8-8　PointcutFunTest：测试代码

```
package com.smart.aspectj.fun;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.smart.Seller;
import com.smart.Waiter;
public class PointcutFunTest{
   @Test
    Public void pointcut {
        String configPath = "com/smart/aspectj/fun/beans.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
        Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
        Waiter naughtyWaiter = (Waiter) ctx.getBean("naughtyWaiter");
        naiveWaiter.greetTo("John");  //①该方法未被织入增强
        naughtyWaiter.greetTo("Tom");//②该方法被织入增强
    }
}
```

输出以下信息：

```
NaiveWaiter:greet to John...
NaughtyWaiter:greet to Tom... ①  ← 对应 NaughtyWaiter 的 greetTo() 方法
needTestFun() executed!
```

从以上信息中可以获知，切面被正确地织入 NaughtyWaiter#greetTo()方法中。

8.5.2　execution()

execution()是最常用的切点函数，其语法如下：

```
execution(<修饰符模式>? <返回类型模式> <方法名模式>(<参数模式>) <异常模式>?)
```

除了返回类型模式、方法名模式和参数模式外，其他项都是可选的。与其直接讲解该方法的使用规则，还不如通过一个个具体的例子来理解。下面给出各种使用 execution()函数的实例。

1．通过方法签名定义切点

- execution(public * *(..))：匹配所有目标类的 public 方法，但不匹配 SmartSeller 和 protected void showGoods()方法。第一个*代表返回类型；第二个*代表方法名；而..代表任意入参的方法。
- execution(* *To(..))：匹配目标类所有以 To 为后缀的方法，它匹配 NaiveWaiter 类和 NaughtyWaiter 的 greetTo()和 serveTo()方法。第一个*代表返回类型；而*To 代表任意以 To 为后缀的方法。

2．通过类定义切点

- execution(* com.smart.Waiter.*(..))：匹配 Waiter 接口的所有方法，它匹配 NaiveWaiter 和 NaughtyWaiter 类的 greetTo()和 serveTo()方法。第一个*代表返回任意类型；com.smart.Waiter.*代表 Waiter 接口中的所有方法。
- execution(* com.smart.Waiter+.*(..))：匹配 Waiter 接口及其所有实现类的方法，它不但匹配 NaiveWaiter 和 NaughtyWaiter 类的 greetTo()和 serveTo()这两个 Waiter 接口定义的方法，同时还匹配 NaiveWaiter#smile()和 NaughtyWaiter#joke()这两个不在 Waiter 接口中定义的方法。

3．通过类包定义切点

在类名模式串中，“.*”表示包下的所有类，而“..*”表示包、子孙包下的所有类。

- execution(* com.smart.*(..))：匹配 com.smart 包下所有类的所有方法。
- execution(* com.smart..*(..))：匹配 com.smart 包、子孙包下所有类的所有方法，如 com.smart.dao、com.smart.service 及 com.smart.dao.user 包下所有类的所有方法都匹配。当“..”出现在类名中时，后面必须跟“*”，表示包、子孙包下的所有类。
- execution(* com..*.*Dao.find*(..))：匹配包名前缀为 com 的任何包下类名后缀为 Dao 的方法，方法名必须以 find 为前缀。如 com.smart.UserDao#findByUserId()、com.smart.dao.ForumDao#findById()方法都匹配切点。

4．通过方法入参定义切点

切点表达式中的方法入参部分比较复杂，可以使用“*”和“..”通配符。其中，“*”表示任意类型的参数；而“..”表示任意类型的参数且参数个数不限。

- execution(* joke(String,int))：匹配 joke(String,int)方法，且 joke()方法的第一个入参是 String，第二个入参是 int。它匹配 NaughtyWaiter#joke(String,int)方法。如果方法中的入参类型是 java.lang 包下的类，则可以直接使用类名；否则必须使用全限定类名，如 joke(java.util.List,int)。
- execution(* joke(String,*))：匹配目标类中的 joke()方法，该方法的第一个入参为 String，第二个入参可以是任意类型，如 joke(String s1,String s2)和 joke(String s1,double d2)都匹配，但 joke(String s1,double d2,String s3)不匹配。
- execution(* joke(String,..))：匹配目标类中的 joke()方法，该方法的第一个入参为 String，后面可以有任意个入参且入参类型不限，如 joke(String s1)、joke(String s1,String s2)和 joke(String s1,double d2,String s3)都匹配。
- execution(* joke(Object+))：匹配目标类中的 joke()方法，方法拥有一个入参，且入参是 Object 类型或该类的子类。它匹配 joke(String s1)和 joke(Client c)。如果定义的切点是 execution(* joke(Object))，则只匹配 joke(Object object)，而不匹配 joke(String cc)或 joke(Client c)。

8.5.3　args()和@args()

args()函数的入参是类名，而@args()函数的入参必须是注解类的类名。虽然 args()允许在类名后使用“+”通配符，但该通配符在此处没有意义，添加和不添加效果都一样。

1．args()

该函数接收一个类名，表示目标类方法入参对象是指定类（包含子类）时，切点匹配，如下面的例子：

```
args(com.smart.Waiter)
```

表示运行时入参是 Waiter 类型的方法，它和 execution(* *(com.smart.Waiter))的区别在于后者是针对类方法的签名而言的，而前者则针对运行时的入参类型而言。如 args(com.smart.Waiter)既匹配 addWaiter(Waiter waiter)，又匹配 addNaiveWaiter(NaiveWaiter naiveWaiter)；而 execution(* *(com.smart.Waiter))只匹配 addWaiter(Waiter waiter)。实际上，args(com.smart.Waiter)等价于 execution(* *(com.smart.Waiter+))，当然也等价于 args(com.smart.Waiter+)。

2．@args()

该函数接收一个注解类的类名，当方法的运行时入参对象标注了指定的注解时，匹配切点。这个切点函数的匹配规则不太容易理解，通过图 8-4 对此进行详细讲解。

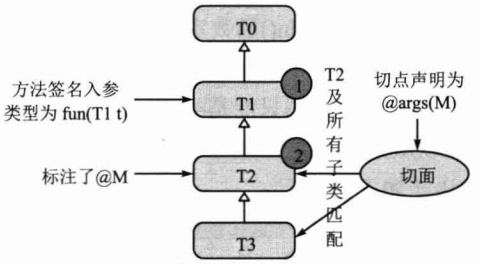


图 8-4　@arg(M)匹配示意图（1）

T0、T1、T2、T3 具有如图 8-4 所示的继承关系，假设目标类方法的签名为 fun(T1 t)，它的入参为 T1，而切面的切点定义为@args(M)，T2 类标注了@M。当 fun(T1 t)的传入对象是 T2 或 T3 时，方法匹配@args(M)声明所定义的切点。

假设方法签名是 fun(T1 t)，入参为 T1，而标注@M 的类是 T0，当 funt(T1 t)传入 T1、T2、T3 的实例时，均不匹配切点@args(M)。

在类的继承树中，①处为方法签名中入参类型在类继承树中的位置，称之为入参类型点；而②处为标注了@M 注解的类在类继承树中的位置，称之为注解点。判断方法在运行时是否匹配@agrs(M)切点，可以根据①和②在类继承树中的相对位置来判别。

（1）如果在类继承树中注解点②高于入参类型点①，则该目标方法不可能匹配切点@args(M)，如图 8-5 所示。

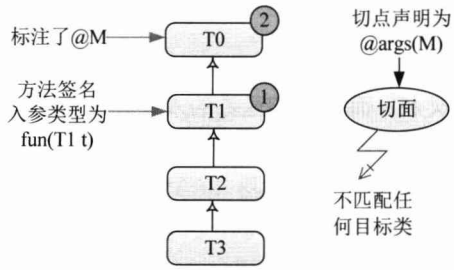


图 8-5　@arg(M)匹配示意图（2）

（2）如果在类继承树中注解点②低于入参类型点①，则注解点所在类及其子孙类作为方法入参时，该方法匹配切点@args(M)，如图 8-4 所示。

下面举一个具体的例子。假设定义这样的切点：@args(com.smart.Monitorable)，如果 NaiveWaiter 标注了@Monitorable，则对于 WaiterManager#addWaiter(Waiter w)方法来说，如果入参是 NaiveWaiter 或其子类对象，则该方法匹配切点；如果入参是 NaughtyWaiter 对象，则不匹配切点。如果 Waiter 标注了@Monitorable，但 NaiveWaiter 未标注@Monitorable，则 WaiterManager#addNaiveWaiter(NaiveWaiter w)不匹配切点，这是因为注解点（Waiter）高于入参类型点（NaiveWaiter）。

8.5.4　within()

通过类匹配模式串声明切点，within()函数定义的连接点是针对目标类而言的，而非针对运行期对象的类型而言，这一点和 execetion()是相同的。但和 execution()函数不同的是，within()所指定的连接点最小范围只能是类，而 execution()所指定的连接点可以大到包，小到方法入参。所以从某种意义上说，execution()函数的功能涵盖了 within()函数的功能。within()函数的语法如下：

```
within(<类匹配模式>)
```

形如 within(com.smart.NaiveWaiter)，是 within()函数所能表达的最小粒度。如果试图用 within()匹配方法级别的连接点，如 within(com.smart.NaiveWaiter.greet*)，那么将会产生解析错误。

下面是一些使用 within()函数的实例。

- within(com.smart.NaiveWaiter)：匹配目标类 NaiveWaiter 的所有方法。如果切点调整为 within(com.smart.Waiter)，则 NaiveWaiter 和 NaughtyWaiter 中的所有方法都不匹配。而 Waiter 本身是接口，不可能实例化，所以 within(com.smart.Waiter)的声明是无意义的。
- within(com.smart.*)：匹配 com.smart 包中的所有类，但不包括子孙包，所以 com.smart.service 包中类的方法不匹配这个切点。
- within(com.smart..*)：匹配 com.smart 包及子孙包中的类，所以 com.smart.service、com.smart.dao 及 com.smart.service.fourm 等包中所有类的方法都匹配这个切点。

8.5.5　@within()和@target()

除@annotation()和@args()函数外，还有另外两个用于注解的切点函数，分别是@target()和@within()。和@annotation()及@args()函数一样，它们也只接受注解类名作为入参。其中，@target(M)匹配任意标注了@M 的目标类，而@within(M)匹配标注了@M 的类及子孙类。

@target(M)切点的匹配规则如图 8-6 所示。

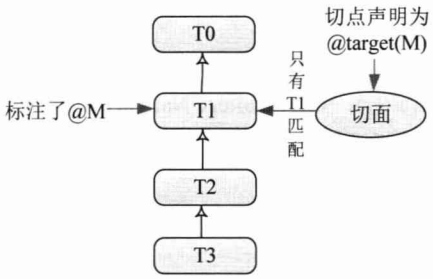


图 8-6　@target(M)匹配目标类示意图

假设 NaiveWaiter 标注了@Monitorable，而其子类 CuteNaiveWaiter 没有标注@Monitorable，则@target(com.smart.Monitorable)匹配 NaiveWaiter 类的所有方法，但不匹配 CuteNaiveWaiter 类的方法。

@within(M)切点的匹配规则如图 8-7 所示。

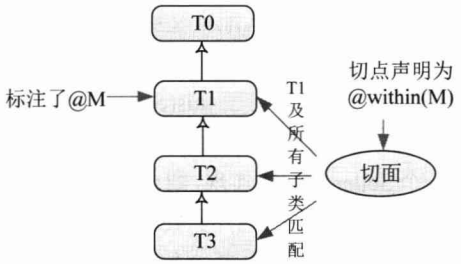


图 8-7　@within(M)匹配目标类示意图

假设 NaiveWaiter 标注了@Monitorable，而其子类 CuteNaiveWaiter 没有标注@Monitorable，则@within(com.smart.Monitorable)不但匹配 NaiveWaiter 类中的所有方法，也匹配 CuteNaiveWaiter 类中的所有方法。

有一个特别值得注意的地方是，如果标注@M 注解的是一个接口，则所有实现该接口的类并不匹配@within(M)。假设 Waiter 标注了@Monitorable 注解，但 NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter 这些接口实现类都没有标注@Monitorable，则@within(com.smart.Monitorable)和@target(com.smart.Monitorable)都不匹配 NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter。这是因为@within()、@target()及@annotation()函数都是针对目标类而言的，而非针对运行时的引用类型而言的，这点区别需要在开发中特别注意。

8.5.6 target()和 this()

target()切点函数通过判断目标类是否按类型匹配指定类来决定连接点是否匹配，而this()函数则通过判断代理类是否按类型匹配指定类来决定是否和切点匹配。二者都仅接受类名的入参，虽然类名可以带“+”通配符，但对于这两个函数来说，使用与不使用“+”通配符，效果完全相同。

1. target()

target(M)表示如果目标类按类型匹配于 M，则目标类的所有方法都匹配切点。下面通过一些例子理解 target(M)的匹配规则。

- target(com.smart.Waiter)：NaiveWaiter、NaughtyWaiter 及 CuteNaiveWaiter 的所有方法都匹配切点，包括那些未在 Waiter 接口中定义的方法，如 NaiveWaiter#simle()和 NaughtyWaiter#joke()方法。
- target(com.smart.Waiter+)：与 target(com.smart.Waiter)是等价的。

2. this()

根据 Spring 的官方文档，this()函数判断代理对象的类是否按类型匹配于指定类，如果匹配，则代理对象的所有连接点匹配切点。但通过实验发现，实际情况和文档有所出入。如声明一个切点 this(com.smart.NaiveWaiter)，如果不使用 CGLib 动态代理，则生成的代理对象属于 Waiter 类型，而非 NaiveWaiter 类型，这一点可以简单地通过 instanceof 操作符进行判断。但是，我们发现 NaiveWaiter 中所有的方法都被织入了增强。

一般情况下，使用 this()和 target()来匹配定义切点，二者是等效的。

（1）target(com.smart.Waiter)等价于 this(com.smart.Waiter)。

（2）target(com.smart.NaiveWaiter)等价于 this(com.smart.NaiveWaiter)。

二者的区别体现在通过引介切面产生代理对象时的具体表现。如果通过 8.4.5 节的方法为 NaiveWaiter 引介一个 Seller 接口的实现，则 this(com.smart.Seller)匹配 NaiveWaiter 代理对象的所有方法，包括 NaiverWaiter 本身的 greetTo()、serverTo()方法，以及通过 Seller 接口引入的 sell()方法；而 target(com.smart.Seller)不匹配通过引介切面产生的 NaiveWaiter 代理对象。

下面通过具体的实例来了解这一微妙的区别。EnableSellerAspect 是为 NaiveWaiter 添加 Seller 接口实现的引介切面，如下：

```
package com.smart.aspectj.fun;
...
@Aspect
public class EnableSellerAspect{
    @DeclareParents(value = "com.smart.NaiveWaiter",
                    defaultImpl = SmartSeller.class)
    public static Seller seller;
}
```

TestAspect 是通过判断运行期代理对象所属类型来定义切点的切面，如代码清单 8-9 所示。

代码清单 8-9　estAspect：通过this()指定切点

```
package com.smart.aspectj.fun;
...
@Aspect
public class TestAspect {
    //①后置增强，织入任何运行期对象为Seller类型的Bean中
    @AfterReturning("this(com.smart.Seller)")
    public void thisTest(){
        System.out.println("thisTest() executed!");
    }
}
```

在 Spring 中配置这两个切面和 NaiveWaiter，如下：

```
<aop:aspectj-autoproxy/>
<bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
<bean class="com.smart.aspectj.fun.EnableSellerAspect"/>
<bean class="com.smart.aspectj.fun.TestAspect" />
```

EnableSellerAspect 切面为 NaiveWaiter 引介 Seller 接口产生一个实现 Seller 接口的代理对象，TestAspect 在判断出 NaiveWaiter 这个代理对象实现 Seller 接口后，就将其切面织入这个代理对象中，所以最终 NaiveWaiter 的代理对象其实共织入了两个切面。在这里，我们忽略了多切面织入顺序的问题，读者将在本章后续的内容中了解到这方面的知识。

运行以下测试代码：

```
String configPath = "com/smart/aspectj/fun/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
naiveWaiter.serveTo("John");
((Seller)naiveWaiter).sell("Beer", "John");
```

输出以下信息：

```
NaiveWaiter:greet to John...
thisTest() executed!
NaiveWaiter:serving John...
thisTest() executed!
SmartSeller: sell Beer to John...
thisTest() executed!
```

可见代理对象的 3 个方法都织入了代码清单 8-9 中通过 this()函数定义的切面。

8.6　@AspectJ 进阶

@AspectJ 可以使用切点函数定义切点，还可以使用逻辑运算符对切点进行复合运算得到复合切点。为了在切面中重用切点，还可以对切点进行命名，以便在其他地方引

用定义过的切点。当一个连接点匹配多个切点时，需要考虑织入顺序的问题，另外一个重要的问题是如何在增强中访问连接点上下文的信息。

8.6.1 切点复合运算

使用切点复合运算符，将拥有强大而灵活的切点表达能力。以下是一个使用了复合切点的切面，如代码清单 8-10 所示。

代码清单 8-10 TestAspect：切点复合运算

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @After("within(com.smart.*) "
            + " && execution(* greetTo(..)))") //①与运算
    public void greeToFun() {
        System.out.println("--greeToFun() executed!--");
    }

    @Before(" !target(com.smart.NaiveWaiter) "+
            "&& execution(* serveTo(..))") //②非与运算
    public void notServeInNaiveWaiter() {
        System.out.println("--notServeInNaiveWaiter() executed!--");
    }

    @AfterReturning("target(com.smart.Waiter) || "+
                    " target(com.smart.Seller)") //③或运算
    public void waiterOrSeller(){
        System.out.println("--waiterOrSeller() executed!--");
    }
}
```

在①处，通过&&运算符定义了一个匹配 com.smart 包中所有 greetTo()方法的切点；在②处，通过!和&&运算符定义了一个匹配所有 serveTo()方法并且该方法不位于 NaiveWaiter 目标类的切点；在③处，通过||运算符定义了一个匹配 Waiter 和 Seller 接口实现类所有连接点的切点。

8.6.2 命名切点

在前面所举的例子中，切点直接声明在增强方法处，这种切点声明方式称为匿名切点，匿名切点只能在声明处使用。如果希望在其他地方重用一个切点，则可以通过@Pointcut 注解及切面类方法对切点进行命名。以下是一个具体的实例，如代码清单 8-11 所示。

代码清单 8-11　TestNamePointcut

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Pointcut;
public class TestNamePointcut {
    @Pointcut("within(com.smart.*)") ①
    private void inPackage(){ }

    @Pointcut("execution(* greetTo(..))") ②
    protected void greetTo(){ }

    @Pointcut("inPackage() and greetTo()") ③
    public void inPkgGreetTo(){ }
}
```

通过注解方法 inPackage()对该切点进行命名，方法可视域修饰符为 private，表明该命名切点只能在本切面类中使用

通过注解方法 greetTo()对该切点进行命名，方法可视域修饰符为 protected，表明该命名切点可以在当前包中的切面类、子切面类中使用

引用命名切点定义的切点，本切点也是命名切点，它对应的可视域为 public

在代码清单 8-11 中定义了 3 个命名切点，命名切点的使用类方法作为切点的名称，此外方法的访问修饰符还控制了切点的可引用性，这种可引用性和类方法的可访问性相同，如 private 的切点只能在本类中引用，public 的切点可以在任何类中引用。命名切点仅利用方法名及访问修饰符的信息，所以习惯上方法的返回类型为 void，并且方法体为空。可以通过图 8-8 更直观地了解命名切点的结构。

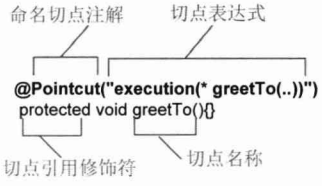


图 8-8　命名切点的结构

在③处，inPkgGreetTo()的切点引用了同类中的 greetTo()切点，而 inPkgGreetTo()切点可以被任何类引用。用户还可以扩展 TestNamePointcut 类，通过类的继承关系定义更多的切点。

命名切点定义好后，就可以在定义切面时通过名称引用切点，请看下面的实例：

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class TestAspect {
    @Before("TestNamePointcut.inPkgGreetTo()") //①
    public void pkgGreetTo(){
        System.out.println("--pkgGreetTo() executed!--");
    }
    @Before("!target(com.smart.NaiveWaiter) && TestNamePointcut. inPkgGreetTo()") //②
    public void pkgGreetToNotNaiveWaiter(){
        System.out.println("--pkgGreetToNotNaiveWaiter() executed!--");
    }
}
```

在①处，引用了 TestNamePointcut.inPkgGreetTo()切点；而在②处，在复合运算中使用了命名切点。

8.6.3 增强织入的顺序

一个连接点可以同时匹配多个切点，切点对应的增强在连接点上的织入顺序是如何安排的呢？这个问题需要分 3 种情况讨论。

- 如果增强在同一个切面类中声明，则依照增强在切面类中定义的顺序进行织入。
- 如果增强位于不同的切面类中，且这些切面类都实现了 org.springframework.core.Ordered 接口，则由接口方法的顺序号决定（顺序号小的先织入）。
- 如果增强位于不同的切面类中，且这些切面类没有实现 org.springframework.core.Ordered 接口，则织入的顺序是不确定的。

可以通过图 8-9 描述这种织入的规则。切面类 A 和 B 都实现了 Ordered 接口，切面类 A 对应序号为 1，切面类 B 对应序号为 2，切面类 A 按顺序定义了 3 个增强，切面类 B 按顺序定义两个增强，这 5 个增强对应的切点都匹配某个目标类的连接点，则增强织入的顺序为图中虚线所示。

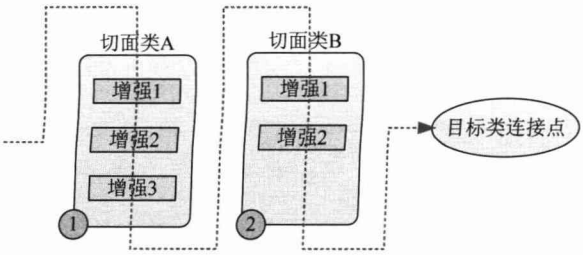


图 8-9 增强织入顺序

8.6.4 访问连接点信息

AspectJ 使用 org.aspectj.lang.JoinPoint 接口表示目标类连接点对象。如果是环绕增强，则使用 org.aspectj.lang.ProceedingJoinPoint 表示连接点对象，该类是 JoinPoint 的子接口。任何增强方法都可以通过将第一个入参声明为 JoinPoint 访问连接点上下文信息。先来了解一下这两个接口的主要方法。

1. JoinPoint

- java.lang.Object[] getArgs()：获取连接点方法运行时的入参列表。
- Signature getSignature()：获取连接点的方法签名对象。
- java.lang.Object getTarget()：获取连接点所在的目标对象。
- java.lang.Object getThis()：获取代理对象本身。

2. ProceedingJoinPoint

ProceedingJoinPoint 继承于 JoinPoint 子接口，它新增了两个用于执行连接点方法的方法。

- java.lang.Object proceed() throws java.lang.Throwable：通过反射执行目标对象的连接点处的方法。
- java.lang.Object proceed(java.lang.Object[] args) throws java.lang.Throwable：通过反射执行目标对象连接点处的方法，不过使用新的入参替换原来的入参。

来看一个具体的实例，如代码清单 8-12 所示。

代码清单 8-12　TestAspect：访问连接点对象

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class TestAspect {
    @Around("execution(* greetTo(..)) && target(com.smart.NaiveWaiter)") ①
    public void joinPointAccess(ProceedingJoinPoint pjp) throws Throwable{ ②
        System.out.println("------joinPointAccess-------");

        System.out.println("args[0]:"+pjp.getArgs()[0]);
        System.out.println("signature:"+pjp.getTarget(). getClass());
        pjp.proceed(); ④
        System.out.println("-------joinPointAccess-------");
    }
}
```

环绕增强

声明连接点入参

③ 访问连接点信息

通过连接点执行目标对象方法

在①处声明了一个环绕增强；在②处增强方法的第一个入参声明为 PreceedingJoinPoint 类型（注意一定要在第一个位置）；在③处通过连接点对象 pjp 访问连接点信息；在④处通过连接点调用目标对象的方法。

执行以下测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
```

输出以下信息：

```
------joinPointAccess-------
args[0]:John
signature:class com.smart.NaiveWaiter
NaiveWaiter:greet to John...  ①
-------joinPointAccess-------
```

对应 pjp.proceed();

8.6.5　绑定连接点方法入参

在介绍切点函数时说过 args()、this()、target()、@args()、@within()、@target()和@annotation()这 7 个函数除了可以指定类名外，还可以指定参数名，将目标对象连接点上的方法入参绑定到增强的方法中。

其中，args()用于绑定连接点方法的入参；@annotation()用于绑定连接点方法的注解

对象；而@args()用于绑定连接点方法入参的注解。来看一个 args()绑定参数的实例，如代码清单 8-13 所示。

代码清单 8-13　TestAspect：绑定连接点参数

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class TestAspect {

    @Before("target(com.smart.NaiveWaiter) && args(name,num,..)") ①
    public void bindJoinPointParams(int num,String name){②
      System.out.println("----bindJoinPointParams()----");
      System.out.println("name:"+name);
      System.out.println("num:"+num);
      System.out.println("----bindJoinPointParams()----");
    }
}
```

绑定连接点参数，首先，args(name, num,..)根据②处的增强方法入参找到 name 和 num 对应的类型，以得到真实的切点表达式：target(com.smart.NaiveWaiter) && args(String,int,..)；其次，在该增强方法织入目标连接点时，增强方法可以通过 num 和 name 访问到连接点的方法入参

增强方法接受连接点的参数

在①处，通过 args(name,num,..)进行连接点参数的绑定。和前面讲述的方式不同，当 args()函数入参为参数名时，共包括两方面的信息。

（1）连接点匹配规则信息：连接点方法的第一个入参是 String 类型，第二个入参是 int 类型。

（2）连接点方法入参和增强方法入参的绑定信息：连接点方法的第一个入参绑定到增强方法的 name 参数上，第二个入参绑定到增强方法的 num 参数上。

切点匹配和参数绑定的过程是这样的：首先，args()根据参数名称在增强方法中查到名称相同的入参并获知对应的类型，这样就知道了匹配连接点方法的入参类型；其次，连接点方法入参类型所在的位置则由参数名在 args()函数中声明的位置决定。代码清单 8-13 中 args(name,num) 匹配的目标类方法的第一个入参必须是 String 类型，第二个入参必须是 int 类型，如 smile(String name,int times)匹配，而 smile(int times ,String anme)不匹配。可以通过图 8-10 详细了解这一有趣的匹配过程。

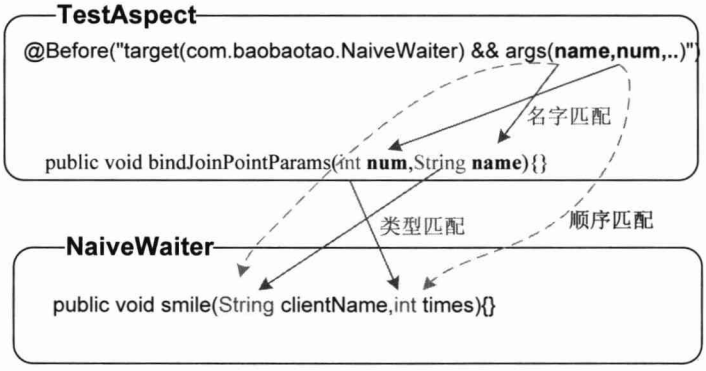


图 8-10　切点匹配和参数绑定过程

和 args()一样，其他可以绑定连接点参数的切点函数（如@args()和 target()等），当指定参数名时，就同时具有匹配切点和绑定参数双重功能。

运行下面的测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
NaiveWaiter naiveWaiter = (NaiveWaiter) ctx.getBean("naiveWaiter");
naiveWaiter.smile("John",2);
```

将看到以下输出信息：

```
----bindJoinPointParams()----
name:John
num:2
----bindJoinPointParams()----
NaiveWaiter:smile to  John2times...
```

可见，增强方法按预期绑定了 NaiveWaiter.smile(String name,int times)方法的运行期入参。

提示

为了保证实例能成功执行，必须启用 CGLib 动态代理：<aop:aspectj- autoproxy proxy-target-class="true" />。因为该实例需要对 NaiveWaiter 类进行代理（因为 NaiveWaiter#simle()方法不是 Waiter 接口的方法），所以必须使用 CGLib 动态代理生成子类的代理方法。

8.6.6　绑定代理对象

使用 this()或 target()函数可绑定被代理对象实例，在通过类实例名绑定对象时，依然具有原来连接点匹配的功能，只不过类名是通过增强方法中同名入参的类型间接决定罢了。这里通过 this()函数来了解对象绑定的用法。

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import com.smart.Waiter;
@Aspect
public class TestAspect {
    @Before("this(waiter)") ①
    public void bindProxyObj(Waiter waiter){ ②
      System.out.println("----bindProxyObj()----");
      System.out.println(waiter.getClass().getName());
      System.out.println("----bindProxyObj()----");
    }
}
```

通过②处查找出 waiter 对应的类型为 Waiter，因而切点表达式为 this(Waiter)。当增强方法织入目标连接点时，增强方法通过 waiter 入参绑定目标对象

①处的切点表达式首先按类变量名查找②处增强方法的入参列表，进而获取类变量名对应的类为 com.smart.Waiter，这样就知道了切点的定义为 this(com.smart.Waiter)，即所有代理对象为 Waiter 类的所有方法匹配该切点。②处的增强方法通过 waiter 入参绑定目标对象。

可见 NaiveWaiter 的所有方法都匹配①处的切点。运行以下测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
naiveWaiter.greetTo("John");
```

可以看到如下输出信息：

```
----bindProxyObj()----
com.smart.NaiveWaiter$$EnhancerByCGLIB$$6758891b
----bindProxyObj()----
NaiveWaiter:greet to John...
```

按相似的方法使用 target()函数进行绑定。

8.6.7 绑定类注解对象

@within()和@target()函数可以将目标类的注解对象绑定到增强方法中，下面通过@within()函数演示注解绑定的操作。

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import com.smart.Monitorable;
@Aspect
public class TestAspect {
    @Before("@within(m)")  ①
    public void bindTypeAnnoObject(Monitorable m){ ②
       System.out.println("----bindTypeAnnoObject()----");
       System.out.println(m.getClass().getName());
       System.out.println("----bindTypeAnnoObject()----");
    }
}
```

通过②处查找出 m 对应 Monitorable 类型的注解，因而真实的切点表达式为 @within (Monitorable)。当增强方法织入目标连接点时，增强方法通过 m 入参可以引用到连接点

在 NaiveWaiter 类中标注了@Monitorable 注解，所有的 NaiveWaiter Bean 都匹配切点，其 Monitorable 注解对象将绑定到增强方法中。运行以下代码，即可查看绑定注解对象：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((NaiveWaiter)naiveWaiter).greetTo("John");
```

输出以下信息：

```
----bindTypeAnnoObject()----
$Proxy3
----bindTypeAnnoObject()----
NaiveWaiter:greet to John...
```

从输出信息中我们发现了一个秘密，即使用 CGLib 代理 NaiveWaiter 时，其类的注解 Monitorable 对象也被代理了。

8.6.8　绑定返回值

在后置增强中，可以通过 returning 绑定连接点方法的返回值，如下：

```
@AfterReturning(value="target(com.smart.SmartSeller)",returning="retVal") //①
public void bingReturnValue(int retVal){ //②
   System.out.println("----bindException()----");
   System.out.println("returnValue:"+retVal);
   System.out.println("----bindException()----");
}
```

①处和②处的名字必须相同，此外，②处 retVal 的类型必须和连接点方法的返回值类型匹配。运行下面的测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
SmartSeller seller = (SmartSeller) ctx.getBean("seller");
seller.sell("Beer","John");
```

可以看到以下输出信息：

```
SmartSeller: sell Beer to John...
----bingReturnValue()----
returnValue:100
----bingReturnValue()----
```

可见目标连接点 Seller#sell()方法返回的入参被成功绑定到增强方法中。

8.6.9　绑定抛出的异常

和通过切点函数绑定连接点信息不同，连接点抛出的异常必须使用 AfterThrowing 注解的 throwing 成员进行绑定，如代码清单 8-14 所示。

代码清单 8-14　TestAspect：绑定异常对象

```
package com.smart.aspectj.advanced;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class TestAspect {
@AfterThrowing(value="target(com.smart.SmartSeller)",throwing="iae") //①
    public void bindException(IllegalArgumentException iae){ //②
      System.out.println("----bindException()----");
      System.out.println("exception:"+iae.getMessage());
      System.out.println("----bindException()----");
    }
}
```

①处 throwing 指定的异常名和②处入参的异常名相同，这个异常增强只在连接点抛出异常 instanceof IllegalArgumentException 时才匹配，增强方法通过 iae 参数可以访问抛出的异常对象。

在 SmartSeller 中添加一个抛出异常的测试方法，如下：

```
package com.smart;
public class SmartSeller implements Seller {
```

```
    public void checkBill(int billId){
        if(billId = = 1) throw new IllegalArgumentException("iae Exception");
        else throw new RuntimeException("re Exception");
    }
}
```

当 billId 为 1 时抛出 IllegalArgumentException，否则抛出 RuntimeException。运行以下测试代码：

```
String configPath = "com/smart/aspectj/advanced/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
SmartSeller seller = (SmartSeller) ctx.getBean("seller");
seller.checkBill(1); ①←——运行该方法将抛出 IllegalArgumentException
```

将看到以下输出信息：

```
----bindException()----
exception:iae Exception
----bindException()----
Exception in thread "main" java.lang.IllegalArgumentException: iae Exception
...
```

可见，当 seller.checkBill(1)抛出异常后，异常增强起作用，处理完成后，再向外抛出 IllegalArgumentException。如果将①处的代码调整为 seller.checkBill(2)，再次运行代码，将直接抛出 RuntimeException，异常增强并没有起作用。这是因为 RuntimeException 不按类型匹配于 IllegalArgumentException，切点不匹配。

8.7 基于 Schema 配置切面

如果读者的项目不能使用 Java 5.0，那么就无法使用基于@AspectJ 注解的切面。但是使用 AspectJ 切点表达式的大门依旧向我们敞开着，因为 Spring 提供了基于 Schema 配置的方法，它完全可以替代基于@AspectJ 注解声明切面的方式。

这是很容易理解的，因为基于@AspectJ 注解的切面，本质上是将切点、增强类型的信息使用注解进行描述，现在把这两个信息移到 Schema 的 XML 配置文件中，只是配置信息所在的地方不一样，表达的信息可以完全相同。XML 和注解，一个是“金刚”，一个是“罗汉”，就像那句俗语“猪往前拱，鸡往后刨”，做的是同一件事，只是方法形式不同而已。

使用基于 Schema 的切面定义后，切点、增强类型的注解信息从切面类中剥离出来，原来的切面类也就蜕变为真正意义上的 POJO。

8.7.1 一个简单切面的配置

首先来配置一个基于 Schema 的切面，它使用了 aop 命名空间。为了更准确地了解整个配置文件的全貌，给出一个结构上完整的切面示例，如代码清单 8-15 所示。

代码清单 8-15　基于Schema配置的切面示例

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring- aop-4.0.xsd">
  <aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods"> ①
      <aop:before pointcut="target(com.smart.NaiveWaiter) and execution (* greetTo(..))"②
          method="preGreeting"/> ③
    </aop:aspect>
  </aop:config>

  <bean id="adviceMethods" class="com.smart.schema.AdviceMethods" /> ④
  <bean id="naiveWaiter" class="com.smart.NaiveWaiter" />
  <bean id="naughtyWaiter" class="com.smart.NaughtyWaiter" />
</beans>
```

引用④处的 adviceMethods

声明切点表达式

增强方法使用 adviceMethods Bean 中的 preGreeting 方法

增强方法所在的 Bean

使用一个<aop:aspect>元素标签定义切面，其内部可以定义多个增强。在<aop:config>元素中可以定义多个切面。在①处，切面引用了 adviceMethods Bean，该 Bean 是增强方法所在的类。通过<aop:before>声明了一个前置增强，并通过 pointcut 属性定义切点表达式，切点表达式的语法和@AspectJ 中所用的语法完全相同，由于&&在 XML 中使用不便，所以一般用 and 操作符代替之。③处通过 method 属性指定增强的方法，该方法应该是 adviceMethods Bean 中的方法。

<aop:config>拥有一个 proxy-target-class 属性，当设置为 true 时，表示其中声明的切面均使用 CGLib 动态代理技术；当设置为 false 时，使用 Java 动态代理技术。一个配置文件可以同时定义多个<aop:config>，不同的<aop:config>可以采取不同的代理技术。

AdviceMethods 是增强方法所在的类，它是一个普通的 Java 类，没有任何特殊的地方。

```
package com.smart.schema;
public class AdviceMethods{
    public void preGreeting() { ①
        System.out.println("--how are you!--");
    }
}
```

该方法通过配置被用作增强的方法

通过以下代码测试这个使用 Schema 配置的切面：

```
String configPath = "com/smart/schema/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
Waiter naughtyWaiter = (Waiter) ctx.getBean("naughtyWaiter");
naiveWaiter.greetTo("John");
naughtyWaiter.greetTo("Tom")
```

输出以下信息：

```
--how are you!-- ①
NaiveWaiter:greet to John...
NaughtyWaiter:greet to Tom...
```

织入的增强逻辑

可见，切面被正确地实施到目标 Bean 中。

8.7.2 配置命名切点

在代码清单 8-15 的②处通过 pointcut 属性声明的切点是匿名切点，不能被其他增强或其他切面引用。Spring 提供了命名切点的配置方式，如代码清单 8-16 所示。

代码清单 8-16 命名切点配置

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:pointcut id="greetToPoinctcut" ①
                expression="target(com.smart.NaiveWaiter)
            and execution(* greetTo(..))" />
        <aop:before method="preGreeting" pointcut-ref="greetToPoinctcut"/>②
    </aop:aspect>
</aop:config>
```

定义切点，该切点的命名为 greetToPoinctcut

引用切点

在①处，使用<aop:pointcut>定义了一个切点，并通过 id 属性进行了命名；在②处，通过 pointcut-ref 引用这个命名的切点。和<aop:before>一样，除引介增强外，其他任何增强类型的元素都拥有 pointcut、pointcut-ref 及 method 这 3 个属性。

<aop:pointcut>如果位于<aop:aspect>元素中，则命名切点只能被当前<aop:aspect>内定义的元素访问到。为了能被整个<aop:config>元素中定义的所有增强访问，必须在<aop:config>元素下定义切点。

```
<aop:config proxy-target-class="true">
    <aop:pointcut id="greetToPoinctcut" ①
        expression="target(com.smart.NaiveWaiter) and execution(* greetTo(..))" />

    <aop:aspect ref="adviceMethods" >
        <aop:before method="preGreeting"  pointcut-ref="greetToPoinctcut" /> ②
    </aop:aspect>
    <aop:aspect ref="adviceMethods" >
        <aop:after method="postGreeting" pointcut-ref="greetToPoinctcut" /> ③
    </aop:aspect>
</aop:config>
```

当前<aop:config>下的所有增强均可以访问该切点

在①处定义的命名切点分别被②和③处不同的切面增强所访问。如果在<aop:config>元素下直接定义<aop:pointcut>，则必须保证<aop:pointcut>在<aop:aspect>之前定义。在<aop:config>元素下还可以定义<aop:advisor>（后面介绍），三者在<aop:config>中的配置有先后顺序的要求：首先是<aop:pointcut>，然后是<aop:advisor>，最后是<aop:aspect>。而在<aop:aspect>中定义的<aop:pointcut>则没有先后顺序的要求，可以在任何位置定义。可以通过 spring-aop-4.0.xsd Schema 样式定义文件了解关于 config 元素的配置规则。

实战经验

Schema 样式定义文件比 DTD 样式定义文件具有更强的文档样式定义能力。但由于 Schema 样式定义文件使用基于 XML 的文档样式描述语言定义文档格式，所以样式定义文件本身的内容比较复杂，可读性不高。为了快速理解 Spring 所提供的 Schema 样式定义文件（如 spring-aop-4.0.xsd 等）所描述的文档规则，可以借助一些可视化工具。Altova XMLSpy 就是这样的一款 Schema 可视化工具，图 8-11 就是从 XMLSpy 中导出的。

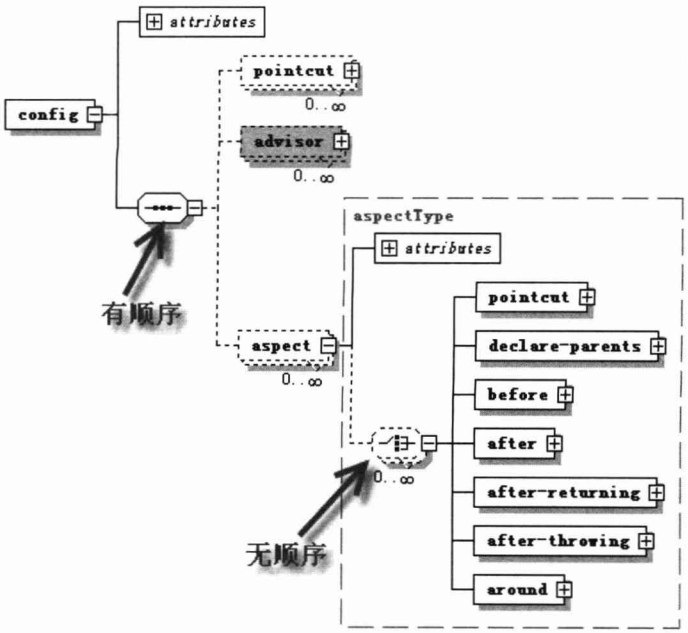


图 8-11　config 元素的 Schema 样式定义

8.7.3　各种增强类型的配置

基于 Schema 定义的切面和基于@AspectJ 定义的切面内容大抵一致，只是在表现形式上存在差异罢了。在上一节中介绍了前置增强，本节来学习如何通过 Schema 配置其余的增强类型。

1. 后置增强

通过<aop:after-returning>配置后置增强。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:after-returning method="afterReturning"
                pointcut="target(com.smart.SmartSeller)" returning= "retVal"/> ①
    </aop:aspect>
```

```
</aop:config>
...
```

returning 属性必须和增强方法的入参名一致。下面是后置增强对应的方法：

```
package com.smart.schema;
public class AdviceMethods {
    public void afterReturning(int retVal){ ①
        ...
    }
}
```

增强方法，retVal 和配置文件中的 returing 属性值相同

如果增强方法不希望接收返回值，将配置处的<aop:after-returning>的 returning 属性和增强方法的对应入参去除即可。

2. 环绕增强

通过<aop:around>配置环绕增强。

```
<aop:config proxy-target-class="true">
<aop:aspect ref="adviceMethods" >
     <aop:around method="aroundMethod"
           pointcut="execution(* serveTo(..) and within(com. smart.Waiter)"/>
     </aop:aspect>
</aop:config>
...
```

和前面介绍的一样，环绕增强对应的方法，可以将第一个入参声明为 ProceedingJoinPoint。

```
package com.smart.schema;
import org.aspectj.lang.ProceedingJoinPoint;
public class AdviceMethods {
    public void aroundMethod(ProceedingJoinPoint pjp){①
        ...
    }
}
```

环绕增强的方法，pjp 可以访问到环绕增强的连接点信息

3. 抛出异常增强

通过<aop:after-throwing>匹配抛出异常的增强。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:after-throwing method="afterThrowingMethod"
          pointcut="target(com.smart.SmartSeller) and execution(* checkBill(..))"
          throwing="iae"/> ①
    </aop:aspect>
</aop:config>
...
```

通过 iae 查找增强方法对应名字的入参，进而获取需要增强的连接点的匹配异常类型为 IllegalArgumentException

通过 throwing 属性声明需要绑定的异常对象，指定的异常名必须和增强方法对应的入参名一致。

```
package com.smart.schema;
public class AdviceMethods {
    public void afterThrowingMethod(IllegalArgumentException iae){
        ...
    }
}
```

4. Final 增强

通过<aop:after>配置 Final 增强。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods" >
        <aop:after method="afterMethod"
                   pointcut="execution(* com..*.Waiter.greetTo(..))"/>
    </aop:aspect>
</aop:config>
...
```

对应的 Final 增强方法如下：

```
package com.smart.schema;
public class AdviceMethods {
    public void afterMethod(){
       ...
    }
}
```

5. 引介增强

通过<aop:declare-parents>配置引介增强。引介增强和其他类型的增强不同，它没有 method、pointcut 和 pointcut-ref 属性。

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
        <aop:declare-parents
            implement-interface="com.smart.Seller" ①
            default-impl="com.smart.SmartSeller" ②
            types-matching="com.smart.Waiter+" /> ③
    </aop:aspect>
</aop:config>
...
```

要引介实现的接口

默认的实现类

哪些类需要引介接口的实现

<aop:declare-parents>通过 implement-interface 属性声明要实现的接口，通过 default-impl 属性指定默认的接口实现类，通过 types-matching 属性以 AspectJ 切点表达式语法指定哪些 Bean 需要引介 Seller 接口的实现。可以通过以下代码查看到 NaiveWaiter 已经实施了引介增强：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((Seller)naiveWaiter).sell("Beer","John"); ①
```

强制类型转换成功

需要注意的是，虽然<aop:declare-parents>没有 method 属性指定增强方法所在的 Bean，但<aop:aspect ref="adviceMethods">中的 ref 属性依然要指定一个增强 Bean。

8.7.4　绑定连接点信息

基于 Schema 配置的增强方法绑定连接点信息和基于@AspectJ 绑定连接点信息所使用的方式没有什么区别。

- 第一，所有增强类型对应的方法的第一个入参都可以声明为 JoinPoint（环绕增强可以声明为 ProceedingJoinPoint）访问连接点信息。

- 第二，<aop:after-returning>（后置增强）可以通过 returing 属性绑定连接点方法的返回值，<aop:after-throwing>（抛出异常增强）可以通过 throwing 属性绑定连接点方法所抛出的异常。
- 第三，所有增强类型都可以通过可绑定参数的切点函数绑定连接点方法的入参。

第一、第二种绑定参数的访问已经在上一节进行了介绍，下面通过一个实例来了解第三种绑定参数的方法，如代码清单 8-17 所示。

代码清单 8-17　绑定连接点参数到增强方法

```
<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
      <aop:before method="bindParams" ①
            pointcut="target(com.smart.NaiveWaiter) and args(name, num,..)"/> ②
    </aop:aspect>
</aop:config>
```

在②处，在切点表达式中通过 args(name,num,..)绑定了连接点的两个参数，对应的增强函数如代码清单 8-18 所示。

代码清单 8-18　AdviceMethods绑定参数的增强方法

```
package com.smart.schema;
import org.aspectj.lang.ProceedingJoinPoint;
public class AdviceMethods {
    public void bindParams(int num,String name){ //①
      System.out.println("----bindParams()----");
      System.out.println("name:"+name);
      System.out.println("num:"+num);
      System.out.println("----bindParams()----");
    }
}
```

①处的 bindParams(int num,String name)和代码清单 8-17 中的切点函数 args(name,num,..)声明的参数名必须相同。运行以下代码：

```
String configPath = "com/smart/schema/beans.xml";
ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);
Waiter naiveWaiter = (Waiter) ctx.getBean("naiveWaiter");
((NaiveWaiter)naiveWaiter).smile("John", 2);
```

输出以下信息：

```
----bindParams()----
name:John
num:2
----bindParams()----
NaiveWaiter:smile to  John2times...
```

可见，目标类 NaiveWaiter 的连接点方法 smile("John", 2)的入参被正确地绑定到增强方法中。

8.7.5　Advisor 配置

在第 7 章中学习了 Advisor 的知识，它是 Spring 中切面概念的对应物，是切点和增

强的复合体，不过它仅包含一个切点和一个增强。在 AspectJ 中没有对应的等价物，在 aop Schema 配置样式中，可以通过<aop:advisor>配置一个 Advisor。通过 advice-ref 属性引用基于接口定义的增强，通过 pointcut 定义切点表达式，或者通过 pointcut-ref 引用一个命名的切点。来看下面的例子：

```
<aop:config proxy-target-class="true">
   <aop:advisor advice-ref="testAdvice"  ①
                pointcut="execution(* com..*.Waiter.greetTo(..))"/> ②
</aop:config>
<bean id="testAdvice"
    class="com.smart.schema.TestBeforeAdvice"/> ③
```

引用③处的增强

切点表达式

前置增强

在③处定义了一个前置增强 Bean，而在①处引用这个增强，在②处使用切点表达式配置切点。TestBeforeAdvice 是一个实现了 MethodBeforeAdvice 接口的增强类，如下：

```
package com.smart.schema;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class TestBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
      System.out.println("------TestBeforeAdvice------");
      System.out.println("clientName:"+args[0]);
      System.out.println("------TestBeforeAdvice------");
    }
}
```

8.8　混合切面类型

现在我们已经掌握了 4 种定义切面的方式：

（1）基于@AspectJ 注解的方式。

（2）基于<aop:aspect>的方式。

（3）基于<aop:advisor>的方式。

（4）基于 Advisor 类的方式。

作为开发者，可能会觉得 Spring 在同一个问题上提供了太多的选择，在选择实现方案时反倒让人陷入了困境。其实 Spring 并不是在做一件吃力不讨好的事情，开发人员完全可以根据项目的具体情况做出选择：如果项目采用 Java 5.0，则可以优先考虑使用@AspectJ；如果项目只能使用低版本的 JDK，则可以考虑使用<aop:aspect>；如果正在升级一个基于低版本 Spring AOP 开发的项目，则可以考虑使用<aop:advisor>复用已经存在的 Advice 类；如果项目只能使用低版本的 Spring，那么就只能使用 Advisor 了。此外，值得注意的是，一些切面只能使用基于 API 的 Advisor 方式进行构建，如基于 ControlFlowPointcut 的流程切面。

8.8.1 混合使用各种切面类型

Spring 虽然提供了 4 种定义切面的方式，但其底层的实现技术却是一样的，那就是基于 CGLib 和 JDK 动态代理，所以在同一个 Spring 项目中可以混合使用 Spring 所提供的各种切面定义方式，如代码清单 8-19 所示。

代码清单 8-19 各种切面类型混合使用

```
<bean id="controlFlowAdvisor" ①                    使用 Advisor API 方式实现的流程控制切面
     class="org.springframework.aop.support. DefaultPointcutAdvisor">
    <property name="pointcut">
     <bean class="org.springframework.aop.support.ControlFlowPointcut">
        <constructor-arg type="java.lang.Class"
                         value="com.smart.advisor.WaiterDelegate"/>
        <constructor-arg type="java.lang.String" value="service"/>
     </bean>
   </property>
   <property name="advice">
     <bean class="com.smart.advisor.GreetingBeforeAdvice"/>
   </property>
</bean>                          使用@AspectJ 方式定义的切面

<aop:aspectj-autoproxy />②

<bean class="com.smart.aspectj.example.PreGreetingAspect" />
                                        使用基于 Schema 配置方式定义的切面
<aop:config proxy-target-class="true"> ③

    <aop:advisor advice-ref="testAdvice"  pointcut="execution(* com..*.Waiter.
greetTo(..))"/>④ ←——使用<aop:advisor>配置方式定义的切面

    <aop:aspect ref="adviceMethods"> ⑤ ←——使用<aop:aspect>配置方式定义的切面
      <aop:before pointcut="target(com.smart.NaiveWaiter) and execution (* greetTo(..))"
          method="preGreeting"/>
                                                    POJO 的增强类
    </aop:aspect>
</aop:config>
<bean id="adviceMethods" class="com.smart.schema. AdviceMethods" />⑥
<bean id="testAdvice" class="com.smart.schema. TestBeforeAdvice"/>⑦ ←  基于特定增强
                                                                  接口的增强类
```

虽然 Spring 可以混合使用各种切面类型达到统一的效果，但在一般情况下并不会在一个项目中同时使用。毕竟项目开发不是时装表演，也不是多军种联合演习，我们应该尽量根据项目的实际需要采用单一的实现方式，以保证技术的单一性。

8.8.2 各种切面类型总结

在学习完 Spring 的 4 种切面定义方式后，有必要对这 4 种实现方式进行比较，它们的本质是相同的，都是定义切点和增强，不同的只是在表现形式上，如表 8-2 所示。

表 8-2　切面不同定义方式具体实现比较

		@AspectJ	<aop:aspect>	Advisor	<aop:advisor>
增强类型	前置增强	@Before	<aop:before>	MethodBeforeAdvice	同 Advisor
	后置增强	@AfterReturning	<aop:after-returning>	AfterReturningAdvice	同 Advisor
	环绕增强	@Around	<aop:around>	MethodInterceptor	同 Advisor
	抛出异常增强	@AfterThrowing	<aop:after-throwing>	ThrowsAdvice	同 Advisor
	final 增强	@After	<aop:after>	无对应接口	同 Advisor
	引介增强	@DeclareParents	<aop:declare-parents>	IntroductionInterceptor	同 Advisor
切点定义		支持 AspectJ 切点表达式语法，可以通过@Pointcut 注解定义命名切点	支持 AspectJ 切点表达式语法，可以通过<aop:pointcut>定义命名切点	直接通过基于 Pointcut 的实现类定义切点	基本上和<aop:aspect>相同，不过切点函数不能绑定参数
连接点方法入参绑定		（1）使用 JoinPoint、Proceeding JoinPoint 连接点对象； （2）使用切点函数指定参数名绑定	同@AspectJ <aop:after-returning>	通过增强接口方法入参绑定	同 Advisor
连接点方法返回值或抛出异常绑定		（1）在后置增强中，使用@AfterReturning 的 returning 成员绑定方法返回值； （2）在抛出异常增强中，使用@AfterThrowing 的 throwing 成员绑定方法抛出的异常	（1）在后置增强中，使用 <aop:after-returning>的 returning 属性绑定方法返回值； （2）在抛出异常增强中，使用 <aop:after-throwing>的 throwing 属性绑定方法抛出的异常	通过增强接口方法入参绑定	同 Advisor

从表 8-2 中可以看出，<aop:advisor>其实是<aop:aspect>和 Advisor 的“混血儿”，它的切点表示方式和<aop:aspect>相同，增强定义方式则和 Advisor 相同。连接点方法入参的绑定方式和 Advisor 一样，通过增强接口方法入参进行调用，所以<aop:advisor>在切点表达式中不能使用切点函数绑定连接点方法入参，否则会产生错误。

在内部，Spring 使用 AspectJExpressionPointcut 为@AspectJ、<aop:aspect>及<aop:advisor>提供具体的切点实现。

8.9　其他

8.9.1　JVM Class 文件字节码转换基础知识

到目前为止，我们所接触到的 AOP 切面织入都是在运行期通过 JDK 或 CGLib 动态代理的方式实现的。我们知道，除了运行期织入切面的方式外，还可以在类加载期通过字节码编辑技术将切面织入目标类中，这种织入方式称为 LTW（Load Time Weaving）。

AspectJ LTW 使用 Java 5.0 所提供的代理功能（agent）完成加载期切面织入工作。JDK 的代理功能够让代理器访问到 JVM 的底层部件，借此向 JVM 注册类文件转换器，在类加载时对类文件的字节码进行转换。AspectJ LTW 由于基于 JDK 动态代理技术工作，而 JDK 动态代理的作用范围是整个 JVM，所以这种工作方式比较粗放，对于单一 JVM 多个应用的情况尤其不适合。

Spring 为 LTW 的过程提供了细粒度的控制，它支持在单个 ClassLoader 范围内实施类文件转换，且配置更为简单。

在进行 Spring LTW 的学习之前，先来了解一下 Java 5.0 中引入的 Instrument 的相关知识，为后面的学习打好基础。

java.lang.instrument 包的工作原理

Java 5.0 新增了一个 java.lang.instrument 包，该包中有两个能对 JVM 底层组件进行访问的类。具体地说，就是通过 JVM 的-javaagent 代理参数在启动时获取 JVM 内部组件的引用，以便在后续流程中使用。借助 JDK 动态代理，可以在 JVM 启动时装配并应用 ClassTransformer，对类字节码进行转换，实现 AOP 的功能。

java.lang.instrument 包中定义了两个重要的接口。

- ClassFileTransformer：Class 文件转换器接口，该接口有一个唯一的方法，如下：

```
byte[ ] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
ProtectionDomain protectionDomain, byte[ ] classfileBuffer)
```

该接口对 Class 文件的字节码进行转换，classfileBuffer 是类文件对应的字节码数组，返回的 byte[]为转换后的字节码。如果返回 null，则表示不进行字节码处理（并非将类的字节码数据置空）。

- Instrumentation：代表 JVM 内部的一个构件。这个术语不好翻译，它有“仪表、仪器、设备”等意思，我们不妨将其称为“组件”。

可以通过该接口的方法向 JVM 的内部“组件”注册一些 ClassFileTransformer，注册转换器的接口方法为 void addTransformer(ClassFileTransformer transformer)。

当 ClassFileTransformer 实例注册到 JVM 中后，JVM 在加载 Class 文件时，会先调用这个 ClassFileTransformer 的 transform()方法对 Class 文件的字节码进行转换。如果向 JVM 中注册多个 ClassFileTransformer，它们将按注册的顺序组成链式的调用。这样 ClassFileTransformer 的实现者就可以从 JVM 层面截获所有类的字节码，并引入希望添加的逻辑，如让每个类拥有性能监视的能力、织入特殊用途的增强代码等。

图 8-12 描述了拥有多个转换器的 JVM 从加载类到最终生成对应的类字节码的过程。

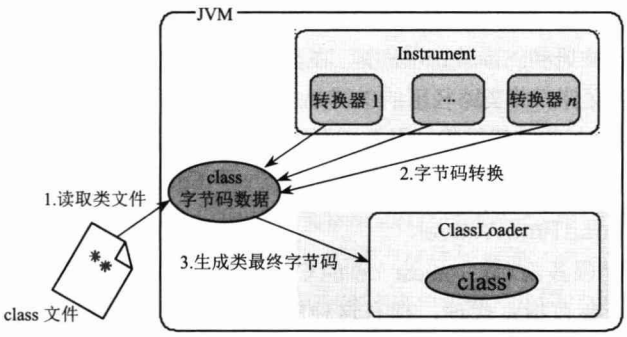


图 8-12　带转换器的 JVM 加载 Class 文件的处理过程

提示

目前已经有很多字节码处理的开源项目，ObjectWeb 的 ASM、Apache 的 BCEL 及 SourceForge 和 SERP 是其中优秀的代表者。TopLink JPA 就使用 ASM 编写它的 ClassTransformer。

8.9.2　使用 LTW 织入切面

Spring 的 LTW 仅支持 AspectJ 定义的切面，既可以是直接采用 AspectJ 语法定义的切面，也可以是采用基于@AspectJ 注解，通过 Java 类定义的切面。Spring LTW 直接采用了与 AspectJ LTW 相同的基础结构，即它利用类路径下的 META-INF/aop.xml 配置文件找到切面定义及切面所要实施的候选目标类的信息，通过 LoadTimeWeaver 在 ClassLoader 加载类文件时将切面织入目标类中。工作原理如图 8-13 所示。

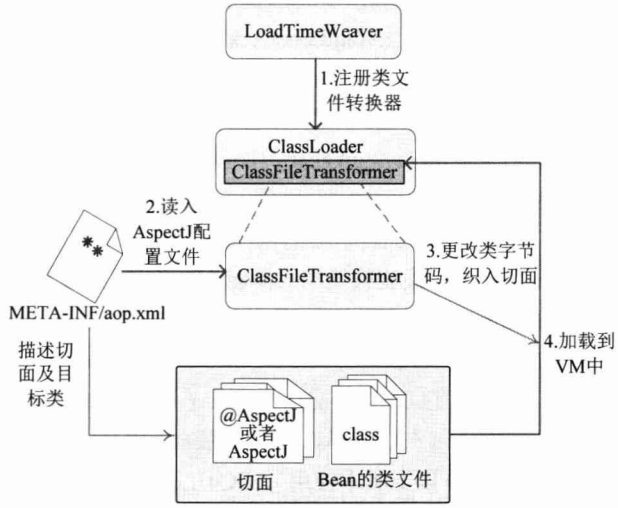


图 8-13　Spring LTW 工作原理

Spring 利用特定 Web 容器的 ClassLoader，通过 LoadTimeWeaver 将 Spring 提供的 ClassFileTransformer 注册到 ClassLoader 中。在类加载期，注册的 ClassFileTransformer 读取 AspectJ 的配置文件，即类路径下的 META-INF/aop.xml 文件，获取切面，对加载到 VM 中的 Bean 类进行字节码转换，织入切面。Spring 容器初始化 Bean 实例时，采用的 Bean 类就是已经被织入了切面的类。下面来了解一下 Spring 的 LoadTimeWeaver。

1. Spring 的 LoadTimeWeaver

大多数 Web 应用服务器（除 Tomcat 外）的 ClassLoader 都支持直接访问 Instrument，无须通过 javaagent 参数指定代理，拥有这种能力的 ClassLoader 称为“组件使能”（instrumentation-capable）。通过“组件使能”功能，可以非常方便地访问 ClassLoader 的 Instrument。Spring 利用 Web 应用服务器类加载器的这个特性，为它们分别提供了专门的 LoadTimeWeaver，以便向特定的 ClassLoader 注册 ClassFileTransformer，对类进行字节码转换，实施切面的织入。

Spring 的 org.springframework.instrument.classloading.LoadTimeWeaver 接口规定了类加载期织入器的高层协议。该接口有 3 个方法。

- void addTransformer(ClassFileTransformer transformer)：添加一个 ClassFileTransformer 到加载期织入器中。
- ClassLoader getInstrumentableClassLoader()：我们知道，JVM 拥有 Instrumentation 组件，但这是 JVM 级别的。Spring 对 ClassLoader 进行扩展，让它也具有 Instrumentation 组件，以便只对 ClassLoader 中的类应用 ClassFileTransformer。
- ClassLoader getThrowawayClassLoader()：返回一个“丢弃”的 ClassLoader，目的是使 Instrumentation 的作用范围仅局限在本 ClassLoader 中，而不影响父 ClassLoader。

Spring 为 LoadTimeWeaver 提供了多个实现类。

- InstrumentationLoadTimeWeaver：该装载期织入器使用-javaagent JVM 启动参数注册转换器，该类必须和 org.springframework.instrument.InstrumentationSavingAgent 结合使用。InstrumentationSavingAgent 是代理类，它获取 JVM 的 Instrumentation 后，以静态变量的方式保存这个引用，这样其他类就可以通过 getInstrumentation() 方法从 InstrumentationSavingAgent 中获取 JVM 的 Instrumentation。InstrumentationLoadTimeWeaver 即利用 InstrumentationSavingAgent 持有的 Instrumentation 引用完成添加转换器的操作。
- SimpleLoadTimeWeaver：该装载期织入器能为当前的 ClassLoader 创建一个相应的 SimpleInstrumentableClassLoader（简单组件使能的 ClassLoader 实现类），一般在测试或 IDE 环境下使用。
- 特定 Web 服务器的织入器：JBossLoadTimeWeaver（在 JBoss AS 5.0 以上版本使用）、GlassFishLoadTimeWeaver（在 GlassFish 3.0 以下版本使用）、WebLogicLoadTimeWeaver（在 BEA WebLogic 10.0 以上版本使用）、OC4JLoadTimeWeaver（在 Oracle OC4J 10.1.3.1 以上版本使用）。

- ReflectiveLoadTimeWeaver：有一些应用服务器不是组件使能的 ClassLoader，或者虽然是组件使能的，但却无法获取应用服务器的类加载器实例，用户只能以 ClassLoader 类型获取这些类加载器的句柄。在这种情况下，Spring 也没有办法将转换器添加到 ClassLoader 中。这时，只得对目标的 ClassLoader 进行改造，让它实现一些特殊的方法，并由 ReflectiveLoadTimeWeaver 在运行期通过反射的机制调用接口方法，以便注册 ClassFileTransformer。可以通过扩展需要改造的 ClassLoader 类，并实现以下两个特殊方法达到目的。
  - public void addTransformer(java.lang.instrument.ClassFileTransformer)：向 ClassLoader 中注册转换器。
  - public ClassLoader getThrowawayClassLoader()：返回需要丢弃的 ClassLoader。

Spring 只需在配置文件中添加一行配置就可以启用 LoadTimeWeaver，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"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.0.xsd">
   <!--①启用Spring装载期织入功能 -->
   <context:load-time-weaver/>
</beans>
```

通过①处的配置，向 Spring 容器中添加一个 LoadTimeWeaver，它负责向运行期的 ClassLoader 注册多个 ClassFileTransformer，以便实施 LTW 的功能。

2. 在 Tomcat 下的配置

Tomcat 服务器的 ClassLoader 不是组件使能的 ClassLoader，Spring 专门为 Tomcat 提供了一个 TomcatInstrumentableClassLoader，它扩展于 Tomcat 服务器的 org.apache.catalina.loader.WebappClassLoader，并实现了 LoadTimeWeaver 要求的两个特殊方法。

可以按照以下步骤使 Tomcat Web 应用启动 Spring 的 LoadTimeWeaver。

（1）在 Spring 配置文件中定义<context:load-time-weaver/>。

（2）将 org.springframework.instrument.tomcat-{version}.jar 复制到<TOMCAT_HOME>/lib 下，该 JAR 包中包含了 TomcatInstrumentableClassLoader 类。

（3）编写以下配置片段，让 Tomcat 服务器使用 TomcatInstrumentableClassLoader 作为其 ClassLoader。

```
<Context path="/myWebApp" docBase="/my/webApp/location">
<Loader loaderClass="org.springframework.instrument.classloading.tomcat.
                                ↪TomcatInstrumentableClassLoader"
      useSystemClassLoaderAsParent="false"/>
</Context>
```

如何让以上配置片段生效呢？对于 Tomcat 6.x 和 Tomcat 7.x 来说，可以使用以下 4 种方式。

（1）打开<TOMCAT_HOME>/conf/server.xml 文件，将以上配置片段添加到 server.xml 文件的相应位置上。

（2）将以上配置片段的内容保存为 context.xml 文件，并将其复制到<TOMCAT_HOME>/conf/context.xml 中。

（3）将以上配置片段的内容保存为<webapp_name>.xml 文件，并将其复制到<TOMCAT_HOME>/conf/ Catalina/localhost/<webapp_name>.xml 中。

（4）将以上配置片段的内容保存为 context.xml 文件，并将其和 Web 应用程序一起打包在 WAR 包中，放在 WAR 包的 META-INF 目录下：<webapp_name>.war/META-INF/context.xml。

前两种配置方式会产生全局，也就是说，Tomcat 应用中所有 Web 应用的类加载器都会调整为 TomcatInstrumentableClassLoader；而后两种配置方式只会对相应的 Web 应用产生影响，其他的 Web 应用依旧使用 Tomcat 默认的 WebappClassLoader。从部署的简易性到范围的影响性，最后一种配置方式是最佳选择。

3．在其他 Web 应用服务器下的配置

前面说过，有 4 种类型的 Web 服务器的类加载器支持组件使能功能。因此，它们既无须使用 javaagent 参数，也无须更改任何 Web 服务器的配置文件，只需在 Spring 的配置文件中配置<context:load-time-weaver/>，就可以使用 Spring 的 LTW。现将这 4 种 Web 服务器及其版本列出。

- BEA WebLogic（10.0 以上版本）。
- Oracle Containers for Java EE（OC4J 10.1.3.1 以上版本）。
- Resin（3.1 以上版本）。
- JBoss（5.x 以上版本）。

8.10 小结

在本章中，首先介绍了 Java 5.0 的注解知识，它是学习@AspectJ 的基础。使用@AspectJ 定义切面比基于接口定义切面更加直观、更加简洁，成为 Spring 所推荐的切面定义方式。

掌握切点表达式语法和切点函数是学习@AspectJ 的重心，我们分别对 9 个切点函数进行了详细的讲述。切点表达式非常灵活，拥有强大的切点表达能力，读者可以使用通配符、切点函数及切点运算符定义切点。

如果项目因某种原因无法使用 Java 5.0，则可以采用基于 Schema 配置的方式继续使用 AspectJ 的切点表达式和增强定义，基于 Schema 的配置采用<aop:aspect>描述@Aspect 类所描述的相同信息，只是换了一种方法而已。此外，还可以通过<aop:advisor>复用旧系统已有的 Advice，并配合使用 AspectJ 的切点表达式。

在切点表达式中，大多数切点函数都可以绑定连接点方法的入参，以便增强方法访问连接点信息。此外，用户也可以简单地将增强方法的第一个入参定义为 JoinPoint 访问连接点的上下文。

Spring AOP 虽然提供了 4 种切面定义方式，但其底层实现却是相同的，也就是说，表象不同，本质归一。所以如果需要，则可以放心地混合使用这些不同的定义方式。

Spring 还支持 LTW 的功能，允许通过 AspectJ 定义切面，在类加载期通过类文件转换器织入切面。

第 9 章

Spring SpEL

Spring 动态表达式语言（简称 SpEL）是一个支持运行时查询和操作对象图的强大动态语言。其语法类似于 EL 表达式，具有诸如显式方法调用和基本字符串模板函数等特性。本章对 JVM 动态语言进行简要概述，并对 SpEL 动态语言特性及其表达式的用法进行详细介绍，通过实例逐步揭开 SpEL 表达式的层层外衣。通过本章的学习，读者不仅可以运用 SpEL 表达式解决实际项目中的一些通用性表达式需求，还可以应用 SpEL 提供的扩展点来解决一些高阶问题。

本章主要内容：

- JVM 动态语言
- SpEL 表达式概述
- SpEL 核心接口
- SpEL 基础表达式
- 在 Spring 中使用 SpEL

本章亮点：

- 对 Java 动态表达式发展的总结
- 对 SpEL 各种用法进行讲解

9.1 JVM 动态语言

Java 是一门强类型的静态语言，所有代码在运行之前都必须进行严格的类型检查并编译成 JVM 字节码；因此虽然在安全、性能方面得到了保障，但牺牲了灵活性。这个特征就决定了 Java 在语言层面无法直接进行表达式语句的动态解析。而动态语言恰恰相反，其显著的特点是在程序运行时可以改变程序结构或变量类型。典型的动态语言有

Ruby、Python、JavaScript、Perl 等。这些动态语言能够被广泛应用于许多领域，得益于其动态、简单、灵活等特性。因为它们无须编译，即可被解释执行。它们可以在运行时动态改变表达式语句，非常适合编写复杂的动态表达式。

Java 在实现复杂业务系统、大型商业系统、分布式系统及中间件等方面有着非常强的优势。但在开发这些系统的过程中，有时需要引用动态语言的一些特性，以弥补其在动态性方面的不足。特别是在解决一些商业系统中动态规则的解析需求，如积分规则、各类套餐计费、活动促销等规则时，在 Java 的早期版本中，开发人员一般会使用 Rhino、BeanShell、MVEL 等类库实现。

为了简化在 Java 中使用动态脚本语言的难度，Java 6.0 开始提供对 JSR-223 规范的全面支持。JSR-223 中规范了在 Java 虚拟机上运行的动态脚本语言与 Java 程序之间的交互方式，并在 Java 6.0 中内置集成了 Mozilla Rhino 的 JavaScript 解析引擎，因此可以很方便地在 Java 中调用 JavaScript 编写的动态脚本，并通过 JavaScript 解释执行的特性来编写相关动态业务逻辑。

下例在 Java 中使用 Rhino 快速实现一个动态求和函数 sum，只需在脚本引擎中注册一个标准的 JavaScript 函数，就可以在 Java 应用上下文中调用注册的 JavaScript 函数，如代码清单 9-1 所示。

代码清单 9-1　ScriptSample：脚本实现动态函数

```
package com.smart.js;
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class ScriptSample {

   public static void main(String[] args) throws Exception{
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("JavaScript");
      String scriptText = "function sum(a,b) { return a+b; } ";
      engine.eval(scriptText);
      Invocable invocable = (Invocable) engine;
      Object result = invocable.invokeFunction("sum", 100,99);
      System.out.println(result); //打印199
   }
}
```

上例只用几行代码就实现了一个动态求和函数。首先使用 ScriptEngineManager 创建一个脚本引擎管理器，并通过其创建脚本引擎，调用 ScriptEngine#eval()方法注册 JavaScript 求和函数脚本。接下来就可以使用 Invocable#invokeFunction()方法调用注册的 sum 函数，第一个参数就是要调用的自定义函数名 sum，第二个参数开始对应自定义函数名 sum 的参数列表。

虽然目前在 JVM 中支持很多脚本语言（如 JavaScript、Jruby、Jython 等），但在使用的时候，还是需要对其进行相应的封装。对于仅仅需要一些简单的表达式需求的场景，使用脚本语言显得有些“笨重”，这也就是下文要重点介绍 SpEL 表达式的原因。

9.2 SpEL 表达式概述

Spring 动态语言（简称 SpEL）是一个支持运行时查询和操作对象图的强大的动态语言。其语法类似于 EL 表达式，具有诸如显式方法调用和基本字符串模板函数等特性。

同其他的 Java 动态语言相比(如国外的 OGNL、MVEL 和 JBoss EL，国内的 Aviator、IKExpression 和 FastEL 等)，SpEL 不但提供上述表达式的类似功能，而且更加简洁、灵活。加之出自 Spring 社区之手，与 Spring 框架及其子项目的结合显得更加顺畅自然。

SpEL 表达式的语言特性都是经过 Spring 框架及其子项目的需求一步步提炼而成的。在使用方面，IDE 工具可提供很好的代码自动补全功能。在集成方面，SpEL 抽象了一个通用表达式操作 API，因此可以很好地与其他动态语言进行集成。

SpEL 作为 Spring 家族中表达式求值的基础，它不直接依赖于 Spring 框架，可独立使用。只是在基于 Spring 的框架中使用 SpEL 更加便捷，一般情况下，无须手工调用 SpEL 提供的 API。因为 Spring 框架已经提供了许多直接使用 SpEL 表达式的方法，并且在框架层面屏蔽了表达式的运行设施创建过程，直接使用即可。例如，在 Bean 配置定义中，可以直接通过“#{}”编写 SpEL 表达式。

为了更好地学习 SpEL 动态语言，本章中的示例将 SpEL 作为一个独立的动态语言来使用，通过手工调用 SpEL API 来讲解其各种表达式的用法。下面先通过一个简单的示例来认识一下 SpEL。在使用 SpEL 之前，需要在 pom.xml 文件中添加 spring-expression 模块依赖，如代码清单 9-2 所示。

代码清单 9-2　在pom.xml文件中添加表达式模块依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.smart</groupId>
    <artifactId>chapter9</artifactId>
    <version>1.0</version>
    <name>Spring4.x第九章实例</name>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>
```

SpEL 被设计成一个可独立运行的模块，可以脱离 Spring 容器直接运行，因此只需引入 SpEL 的模块 spring-expression 即可，无须引入 Spring 框架的其他模块。接下来，在代码中就可以使用 SpEL 提供表达式解析类，如代码清单 9-3 所示。

代码清单 9-3　SpelHello示例

```
package com.smart.spel;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
public class SpelHello {
   public static void main(String[] args) {
      ExpressionParser parser =new SpelExpressionParser();
      Expression exp = parser.parseExpression("'Hello'+' World'");
      String message = (String) exp.getValue();
      System.out.println(message);
   }
}
```

在 SpEL 中，使用表达式非常简单，只需创建一个 SpelExpressionParser 实例即可。也就是首先创建一个 SpEL 表达式解析器，然后调用 ExpressionParser#parseExpression() 方法对表达式进行解析。其中，单引号表示 String 类型，如示例中的'Hello'、'World'。

轻松一刻

山东话与河南话的发音大致相同，但也有小小的差别。比如山东人称馒头为干粮，河南人称馒头为馍，发音与山东话的“么”相似。一个河南人在山东上大学，去食堂打饭，向大师傅说：“来个馍。”大师傅听了很是茫然，问：“你要么？”河南人答道：“对呀，我要馍。”大师傅听后就更糊涂了，这怎么来打饭的还不知道自己要什么的，就问道：“你到底要么啊？”河南人有点生气了，心想：你都知道我要什么了怎么还问啊。于是不耐烦地说：“你到底有没有馍？我就是要个馍啊!!”

9.3　SpEL 核心接口

上一节介绍了 SpEL 的简单用法，接下来逐步讲解 SpEL 的核心接口。SpEL 的所有类与接口都定义在 org.springframework.expression 包及其子包，以及 spel.support 中。

ExpressionParser 接口用来解析表达式字符串，表达式字符串是一个用单引号标注或者用转义的双引号标注的字符串。Expression 接口用来计算表达式字符串的值，当调用 ExpressionParser#parseExpression() 和 Expression#getValue() 方法时可能抛出解析异常 ParseException 和求值异常 EvaluationException。因此，在手工调用 API 时，需要处理这些可能的异常。

SpEL 支持一系列功能特性，如方法调用、属性调用及构造器调用等。下例是调用字符串类型的 concat()方法：

```
ExpressionParser parser= new SpelExpressionParser();
```

```
Expression exp=parser.parseExpression("'HelloWorld'.concat('!')");
String message=(String)exp.getValue(); // message的值为"Hello World!"
```

在上述代码中，Expression#getValue()方法返回的是一个 Object 对象，需要进行显式类型转换，使用起来感觉有些不够优雅。可以使用泛型方法 public <T> T getValue (Class<T> desiredResultType)，这个方法取值的时候无须进行显式类型转换。但需要注意的是，如果返回的类型不能被转换为泛型 T 或者被已注册的类型转换器所转换时，则会抛出求值 EvaluationException。

```
String message= exp.getValue(String.class);
```

在 SpEL 中更常见的用途是针对特定实例对象的属性进行求值。在下面的例子中，我们通过表达式获取 User 实例的 userName 属性。

```
User user = new User();
user.setUserName("tom");
user.setCredits(100);
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = new StandardEvaluationContext(user);
String username = (String)parser.parseExpression("userName").getValue(context);
```

在创建 StandardEvaluationContext 实例时，指定一个根对象作为求值目标对象，这样在求值表达式中就可以引用根对象属性。在求值内部可以使用反射机制从注册对象中获取相应的属性值。

在单独使用 SpEL 时，需要创建一个 ExpressionParser 解析器，并提供一个 EvaluationContext 求值上下文。在普通的基于 Spring 的应用开发中，这些 API 一般很少会涉及，仅需编写 SpEL 表达式字符串即可。例如，在 Bean 定义的时候使用 SpEL 表达式，只需写好相应的表达式字符串即可，至于解析器、求值上下文、根对象和其他预定义变量等基础设施，Spring 都会创建。

9.3.1 EvaluationContext 接口

EvaluationContext 接口提供了属性、方法、字段解析器及类型转换器。默认实现类 StandardEvaluationContext 的内部使用反射机制来操作对象。为了提高性能，在其内部会对已获取的 Method、Field 和 Constructor 实例进行缓存。

StandardEvaluationContext 类可以通过构造函数传递求值根对象或通过 setRootObject() 方法设置求值根对象，通过 setVariable()方法设置相关变量，通过 registerFunction()方法注册自定义函数。还可以向 StandardEvaluationContext 类注册自定义构造函数转换器 ConstructorResolvers、方法转换器 MethodResolvers 及属性转换器 PropertyAccessors 来解析 SpEL 表达式。

默认 SpEL 表达式使用的类型转换器是引用 Spring 核心包中的 ConversionService，ConversionService 转换服务会自动根据源类型与目标类型选择合适的转换器。Spring 内置了一系列常用的类型转换器，如 StringToBooleanConverter、NumberToCharacterConverter 等。如果内置的类型转换器未能满足需求，则可以编写并注册自定义类型转换器。当处

理表达式中涉及泛型时，SpEL 会尝试将当前值转换为对应的目标类型。例如，在表达式中通过 setValue()方法来设置 List 类型的元素时，如果目标类型是 List<Boolean>，那么 SpEL 会自动尝试将当前设置值转换为布尔类型，如下：

```
class Simple{
  public List<Boolean> booleanList= new ArrayList<Boolean>();
}
Simple simple= new Simple();
simple.booleanList.add(true);

//创建求值上下文
StandardEvaluationContext simpleContext= new StandardEvaluationContext(simple);

//自动将“false”转换为布尔类型
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

//b将被设置为false
Boolean b=simple.booleanList.get(0);
```

在上面的示例中，表达式“booleanList[0]”是一个 List<Boolean>类型。将一个字符串类型的“false”值设置给 booleanList 集合的第一个元素，由于目标类型是 Boolean，因而在 SpEL 内部会自动将原类型为“false”的字符串值转换为目标布尔类型。如果将“false”改为“false1”，再次运行的时候，则会报一个转换异常 ConversionFailedException，因为“false1”字符串无法转换为布尔类型。

9.3.2　SpEL 编译器

在默认情况下，SpEL 表达式只有在求值时才进行表达式计算，所以表达式可以在运行时进行动态修改。但对于同一个表达式而言，如果每次取值都要进行动态解析，则势必影响表达式的执行效率。对于一些调用频率不高的场景，对性能的影响不是很明显；反之，如果同一个表达式重复调用比较频繁，那么可能会对性能产生较大的影响。

SpelCompiler 编译器就是为了解决这个问题而诞生的，它可以将表达式直接编译成字节码，从而避免每次调用时进行语法解析所产生的时间消耗，有效提高执行效率。由于已经将表达式编译成字节码，如果在后续运行时表达式发生变化，则必须重新编译。因此，SpelCompiler 适用于表达式不经常发生变动且重复调用频率较高的场景。下面是基于 SpEL 编码的示例，如代码清单 9-4 所示。

代码清单 9-4　CompilerSample编译表达式示例

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.spel.SpelCompilerMode;
import org.springframework.expression.spel.SpelParserConfiguration;
import org.springframework.expression.spel.standard.SpelExpression;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
```

```
public class CompilerSample {

    public static void main(String[] args) {
        User user = new User();
        //①创建解析配置
        SpelParserConfiguration configuration = new SpelParserConfiguration(
                                    SpelCompilerMode.IMMEDIATE,
                                    CompilerSample.class.getClassLoader());
        //②创建解析器
        SpelExpressionParser parser = new SpelExpressionParser(configuration);

        //③创建取值上下文
        EvaluationContext context = new StandardEvaluationContext(user);

        //④表达式
        String expression = "isVipMember('tom') && isVipMember('jony')";

        //⑤解析表达式
        SpelExpression spelExpression = parser.parseRaw(expression);

        //⑥通过表达式求值
        System.out.println(spelExpression.getValue(context)); //第一次调用
        System.out.println(spelExpression.getValue(context)); //第二次调用
    }
}
```

在 SpEL 中启用编译模式，需要创建一个 SpelParserConfiguration 配置解析类，并指定编译模式和类加载器。其中，编译模式 SpEL 提供了一个枚举类型 SpelCompilerMode。该枚举共有 3 个值，分别是 SpelCompilerMode.OFF、SpelCompilerMode.MIXED 和 SpelCompilerMode. IMMEDIATE。

编译模式默认采用 SpelCompilerMode.OFF 值，表示不启用编译。想要全局开启，则可以通过配置类路径下的 spring.properties 配置文件，然后 SpelParserConfiguration 会在类加载时读取 spring.expression.compiler.mode 属性来进行配置。

SpelCompilerMode.MIXED 表示混合模式，表示在解释型和编译型之间进行转换；前面几次执行表达式取值采用解释型处理，直到达到 SpEL 的一个阈值（100）之后，才启用编译处理，也就是对表达式进行编译。

SpelCompilerMode. IMMEDIATE 表示立即启用编译。实际上，在 SpEL 内部不会立即启用编译，而是在第二次执行表达式取值时才会启用编译。

在②处通过配置解析对象创建解析器；在③处通过根对象实例 user 创建取值上下文；在④处编写一个求值表达式“isVipMember('tom') && isVipMember('jony')”，其中 isVipMember 是 User 对象中声明的方法，根据用户名判断是否是 VIP 会员，“&&”表示逻辑与操作；在⑤处通过调用 SpelExpressionParser#parseRaw()方法创建 SpelExpression 对象；在⑥处调用 Expression#getValue()方法对表达式求值。

9.4　SpEL 基础表达式

9.4.1　文本字符解析

文本表达式支持字符串、日期、数字（正数、实数及十六进制数）、布尔类型及 null，其中字符串需使用单引号或反斜杠+双引号包含起来，如"' Hello World '"、"\"Hello World \""，而单引号字符可使用“\”表示。来看一个具体的示例，如代码清单 9-5 所示。

代码清单 9-5　LiteralExprSample文本解析

```
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.*;

public class LiteralExprSample {
   public static void main(String[] args) {
      ExpressionParser parser = new SpelExpressionParser();

       //①解析字符串
      String helloWorld = (String) parser.parseExpression("\"Hello World\"").getValue();

       //②解析双精度浮点型
      double doubleNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

      //③解析整型
      int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

      //④解析整型布尔型
      boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

      //⑤解析空值
      Object nullValue = parser.parseExpression("null").getValue();
      …
   }
}
```

本示例演示了如何使用 SpEL 表达式将文本字面值解析为相应的数据类型。数据类型支持负数、小数、科学计数法及八进制、十六进制数等。默认情况下采用 Double.parseDouble()方法进行数值转换。

9.4.2　对象属性解析

在 SpEL 中，可使用类似“xxx.yyy.zzz”的对象属性路径轻松地访问对象属性值，如代码清单 9-6 所示。

代码清单 9-6 PropertyExprSample属性解析

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class PropertyExprSample {
    public static void main(String[] args) {

        //①构造一个对象
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100); //设置积分
        user.setPlaceOfBirth(new PlaceOfBirth("中国","厦门")); //设置出生地

        //②构造SpEL解析上下文
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //③基本属性值获取
        String username = (String)parser.parseExpression("userName").getValue(context);
        int credits = (Integer) parser.parseExpression("credits +
10").getValue(context);

        //④嵌套对象属性值获取
        String city = (String)parser.parseExpression("placeOfBirth.city").getValue
(context);
        …
    }
}
```

从上面的示例中可以看出，对象属性解析与文本字符解析有所不同，对象属性解析需要在取值的时候传递一个计算上下文参数 EvaluationContext。在这里，将 User 实例作为上下文的根对象传递给 EvaluationContext，这样 SpEL 表达式解析器就可以根据属性路径表达式获取上下文中根对象的属性值。

9.4.3 数组、集合类型解析

在 SpEL 中，支持数组、集合类型（Map、List）的解析。数组支持标准 Java 语言创建数组的方法，如“new int[]{1,2,3}”。List 支持大括号括起来的内容，数据项之间用逗号隔开，如“{1,2,3,4}”、“{{'a','b'},{'x','y'}}”。目前 SpEL 还不支持多维数组初始化，如“new int[2][3]{{1,2,3},{4,5,6}}”。Map 采用如下方式表达：{userName:'tom',credits:100}，如代码清单 9-7 所示。

代码清单 9-7　CollectionExprSample属性解析

```
package com.smart.spel;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.List;
import java.util.Map;

public class CollectionExprSample {
    public static void main(String[] args) {
        …
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①数组表达式解析
        int[] array1 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue
                        (context);
        int[][] array2 = (int[][]) parser.parseExpression("new int[2][3]").getValue
                        (context);

        //目前不支持多维数组初始化，以下语句将报错
        int[][] array3= (int[][]) parser.parseExpression("new
                        int[2][3]{{1,2,3},{4,5,6}}").getValue(context);

        //②List表达式解析
        List list = (List) parser.parseExpression("{1,2,3,4}").getValue(context);
        List listOfLists = (List)
parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

        //③列表字符串解析
        Map userInfo = (Map) parser.parseExpression("{userName:'tom',credits:100 }").
                getValue(context);
        List userInfo2 = (List) parser.parseExpression("{{userName:'tom',credits:100 },
                {userName:'tom',credits:100 }}").getValue(context);

        //④从数组，List，Map中取值
        String interest1 =
                (String)parser.parseExpression("interestsArray[0]").getValue(context);
        String interest2 =
                (String)parser.parseExpression("interestsList[0]"). getValue(context);
        String interest3 =
                (String)parser.parseExpression("interestsMap ['interest1']").getValue
                (context);
        …
    }

}
```

在 SpEL 中，要从解析器获取指定属性值，需要将根对象存储到 EvaluationContext 上下文中，如示例中的“new StandardEvaluationContext(user)”。

数组与 List 取值通过指定括号内的索引获取，如示例中的“interestsArray[0]”、“interestsList[0]”，其中 interestsArray、interestsList 均为 User 根对象的属性。

Map 取值通过键名获取，如示例中的“interestsMap['interest1']”，其中 interestsMap 也是 User 对象的属性，“interest1”为 Map 存储数据的键值。

9.4.4 方法解析

在 SpEL 中，方法调用支持 Java 可访问的方法，包括对象方法、静态方法，并支持可变方法参数；还可以调用 String 类型的所有可访问的方法，如 String#substring()，如代码清单 9-8 所示。

代码清单 9-8 MethodExprSample方法解析

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class MethodExprSample {
    public static void main(String[] args) {
        User user = new User();
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①调用String方法
        String substring = parser.parseExpression(
                "'Spring SpEL'.substring(7)").getValue(String.class);
        Integer index = parser.parseExpression(
                 "'Spring SpEL'.indexOf('SpEL')").getValue(Integer.class);

        //②调用实例方法
        boolean isCorrect = parser.parseExpression("validatePassword('123456')").
                                                getValue(context, Boolean.class);
        //③调用私有方法，将发生错误
        isCorrect = parser.parseExpression("validatePassword2('123456')").getValue
                                                (context, Boolean.class);
        //④调用静态方法
        isCorrect = parser.parseExpression("validatePassword3('123456')").getValue
                                                (context, Boolean.class);

       //⑤调用对象方法，可变参数列表
       parser.parseExpression("addInterests('Js','C')").getValue(context, Boolean.
class);
    }
}
```

在①处，在表达式中直接使用 String 提供的方法对文本字符进行各种操作。在②处，调用自定义类 User 的方法 User#validatePassword()。需要注意的是，私有方法是不能调用的，如示例中③处执行时将会报错。在④处，调用自定义类 User 的静态方法 User#validatePassword3()。在⑤处，调用自定义类 User 的可变参数方法 User# addInterests()。

9.4.5　操作符解析

SpEL 提供了丰富的操作符解析，支持关系操作符、逻辑操作符、算术运算操作符及正则表达式匹配等。

1. 关系操作符

在 SpEL 表达式中，可以使用标准 Java 操作符号，支持关系操作符：等于、不等于、小于、小于或等于、大于、大于或等于、正则表达式及 instanceof 操作符，如代码清单 9-9 所示。

代码清单 9-9　OperatorExprSample关系操作符解析

```
package com.smart.spel;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class OperatorExprSample {
   public static void main(String[] args) {
      ExpressionParser parser = new SpelExpressionParser();

      //①关系操作符
      boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);
      boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

      //②字符串关系比较
      trueValue = parser.parseExpression("\"black\" < \"block\"").getValue(Boolean.class);

      //③instanceof运算符
      falseValue = parser.parseExpression( "'xyz' instanceof
                                          T(int)").getValue(Boolean.class);

      //④正则匹配运算，前面为字符串，后面为正则表达式
      trueValue = parser.parseExpression(
                "'5.00' matches  '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
      falseValue = parser.parseExpression(
                "'5.0067' matches '\\^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
}
```

Java 语言本身并未提供字符串关系比较操作符，但 SpEL 则支持字符串关系比较，如示例中②处的“\"black\" < \"block\"”。instanceof 运算符是用来在运行时指出文本字符是否是特定类的一个实例，如示例中的“'xyz' instanceof T(int)”所示。需要注意的是 instanceof 操作符后面的类型表达格式：T(Java 类型)，如整型 T(int)、T(Integer)，字符类型 T(String)。在④处的表达式中使用 matches 关键字来对文本字符进行正则匹配运算，matches 后面是标准的 Java 正则表达式。

2. 逻辑操作符

逻辑操作符支持与操作（and 或&&）、或操作（or 或||）、非操作（!）。需要指出的是，在 SpEL 表达式中，既可以采用标准 Java 逻辑操作符，还可以采用“and”和“or”这两个操作符，如代码清单 9-10 所示。

代码清单 9-10　MethodExprSample逻辑操作符解析

```
package com.smart.spel;
import com.smart.User;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class MethodExprSample {
    public static void main(String[] args) {
        User user = new User();
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①与操作，结果为false
        boolean falseValue = parser.parseExpression("true && false").getValue(Boolean.
class);

        String expression = "isVipMember('tom') and isVipMember('jony')";
        boolean trueValue = parser.parseExpression(expression).getValue(context,
Boolean.class);

        //②或操作，结果为true
        trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

        //③取非操作，结果为false
        falseValue = parser.parseExpression("!true").getValue(Boolean.class);
    }
}
```

在 SpEL 表达式中，通过关系操作符关键字进行与操作，关键字的前后运算结果必须是 Java 标准的布尔类型。如下列表达式是非法的：true and 0；因为 and 操作符后面的“0”是整型值，而不是一个布尔值。

3．算术运算操作符

SpEL 表达式支持 Java 标准运算操作，其中加法运算符可用于数字、字符串和日期，减法运算符可用于数字和日期，乘法和除法运算符仅可用于数字。其他支持的数学运算包括取模（%）和指数幂（^），使用 Java 标准的运算符优先级规则。如代码清单 9-11 所示。

代码清单 9-11　OperatorExprSample算术运算操作符解析

```
package com.smart.spel;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class OperatorExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        //①加法操作，运行结果等于1
        int two = parser.parseExpression("1 + 1").getValue(Integer.class);
```

```
        String testString = parser.parseExpression(
                "\"test\" + ' ' + \"string\"").getValue(String.class);

        //②减法操作，运行结果等于4
        int four = parser.parseExpression("1 - -3").getValue(Integer.class);

        //减法操作，运行结果等于-9000
        double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class);

        //③乘法操作，运行结果等于6
        int six = parser.parseExpression("-2 * -3").getValue(Integer.class);

        //乘法操作，运行结果等于24.0
        double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double.
class);

        //④除法操作，运行结果等于-2
        int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);

        //除法操作，运行结果等于1.0
        double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class);

        //⑤求余操作，运行结果等于3
        int three = parser.parseExpression("7 % 4").getValue(Integer.class);

        // 求余操作，运行结果等于1
        one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class);

        //⑥优先级算术运算，运行结果等于-21
        int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.
class);
    }

}
```

9.4.6　安全导航操作符

安全导航操作符来源于 Groovy 语言，它避免了空指针异常。通常在访问对象方法或属性时需要验证该对象是否为空。为了避免烦琐的空对象验证，可采用安全导航操作符，它只会返回 null 而不是抛出异常。其格式是在获取对象属性操作符“.”前面添加一个“？”，如代码清单 9-12 所示。

代码清单 9-12　SafeExprSample安全导航操作符解析

```
package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.util.Date;
```

```
public class SafeExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国","厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);
        String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context,
String.class);
        System.out.println(city); //打印“厦门”

        //①将PlaceOfBirth对象设置为null
        user.setPlaceOfBirth(null);

        //②不会抛出异常
        city = parser.parseExpression("PlaceOfBirth?.City").getValue(context,
String.class);
        System.out.println(city); //打印“null”
    }

}
```

在上面的示例中，使用了 SpEL 提供的安全导航操作符，如“PlaceOfBirth?.City”，在获取 PlaceOfBirth 对象属性的时候，会先判断 PlaceOfBirth 对象是否为空；如果为空则直接返回 null，如果不为空则获取其属性值。在 Java 8.0 中提供了 java.util.Optional 来支持类似的功能。

9.4.7 三元操作符

可以在表达式内使用 if-then-else 条件逻辑三元操作符，也就是 Java 标准的条件表达式：<表达式 1>?<表达式 2>:<表达式 3>。如表达式“true?'value1': 'value2'”，将返回“value1”。一个真实的例子如代码清单 9-13 所示。

代码清单 9-13 IfThenElseExprSample三元操作符解析

```
package com.smart.spel;

import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

import java.util.Date;

public class IfThenElseExprSample {
    public static void main(String[] args) {
```

```
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国","厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);

        //三元操作表达式
        String expression = "UserName == 'tom'? Credits+10:Credits";

        Integer credits = parser.parseExpression(expression).getValue(context,
Integer.class);
        System.out.println(credits); // 输出“110”
    }
}
```

在示例中，创建了一个用户名为“tom”的用户实例“user”，并设置其积分为 100。在三元操作表达式中判断当前是否存在一个“tom”用户，如果存在，就添加 10 个积分。

9.4.8　Elvis 操作符

Elvis 操作符是 Groovy 语言中使用的三元操作符的缩写。在三元运算符中通常要重复变量两次，例如：

```
String name = "tom";
String displayName = name!=null?name:"Unknown";
```

可以使用 Elvis 操作符替代：name?:"Unknown"，代码非常简洁。下面来看一下具体的实例，如代码清单 9-14 所示。

代码清单 9-14　ElvisExprSample操作符解析

```
package com.smart.spel;

import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class ElvisExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");
        user.setLastVisit(new Date());
        user.setCredits(100);
        user.setPlaceOfBirth(new PlaceOfBirth("中国","厦门"));

        ExpressionParser parser = new SpelExpressionParser();
        StandardEvaluationContext context = new StandardEvaluationContext(user);
```

```
        String userName = parser.parseExpression("UserName?:'用户名为空'").getValue
                                                        (context, String.class);
        System.out.println(userName); //输出“tom”
        user.setUserName(null);
        userName = parser.parseExpression("UserName?:'用户名为空'").getValue(context,
                                                            String.class);
        System.out.println(userName); //输出“用户名为空”
    }
}
```

SpEL 提供的 Elvis 操作符非常简洁，其格式为“<var>?:<value>”。如果左边变量取值为 null，就取 value 值；否则就取 var 变量自身的值。

9.4.9 赋值、类型、构造器、变量

1. 赋值

属性设置是通过使用赋值运算符完成的，相当于调用 Expression#setValue()方法，也可以通过调用 Expression#getValue()方法赋值，如代码清单 9-15 所示。

代码清单 9-15 ObjectExprSample赋值

```
package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
import java.util.Date;

public class ObjectExprSample {
    public static void main(String[] args) {
        User user = new User();
        user.setUserName("tom");

        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①通过setValue赋值
        parser.parseExpression("userName").setValue(context,"jony");
        System.out.println(user.getUserName()); //输出“jony”

        //②通过表达式赋值
        parser.parseExpression("userName='anyli'").getValue(context);
        System.out.println(user.getUserName()); //输出“anyli”
    }
}
```

在①处通过 Expression 接口提供的 setValue()方法为表达式上下文对象属性赋值；在②处在调用 getValue()方法时通过赋值表达式直接赋值，如示例中的“userName='anyli'”。

2. 类型

T 操作符是一个非常实用的类型操作符，通过它可以从类路径加载指定类名称

（全限定名）的 Class 对象，操作表达式为“T(全限定类名)”；其功能类似于 ClassLoader#loadClass()方法，如代码清单 9-16 所示。

代码清单 9-16　ObjectExprSample类型

```
package com.smart.spel;
import com.smart.PlaceOfBirth;
import com.smart.User;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;

public class ObjectExprSample {
   public static void main(String[] args) {
      ExpressionParser parser = new SpelExpressionParser();

       //①加载java.lang.string
      Class stringClass=parser.parseExpression("T(java.lang.String)").getValue
(Class.class);
      System.out.println(stringClass==java.lang.String.class);  //输出 true

       //②加载com.smart.User
      Class userClass=parser.parseExpression("T(com.smart.User)").getValue(Class.
class);
      System.out.println(userClass == com.smart.User.class);  //输出 true
   }

}
```

在①处通过 T 操作符加载 java.lang.String 类，返回类 String 的 Class 对象；在②处通过 T 操作符加载自定义类 User 的 Class 对象。在 SpEL 内部，通过 StandardTypeLocator#findType()方法加载类。上述类的名称都使用全限定类名。如果加载的目标类位于 java.lang 包下，则可以不带包名；对于其他包下的类，则必须带上完整的包名。

T 操作符还可以直接调用类静态方法，操作表达式为“T(全限定类名).静态方法”。例如，可以使用 T 操作符直接调用 Math# random()方法，获取一个随机数，如下：

```
Object randomValue = parser.parseExpression("T(java.lang.Math).random()").getValue();
System.out.println(randomValue); //返回一个随机数
```

通过 T 操作符可以在 SpEL 表达式中调用工具类中的任意静态方法。这是一个非常实用的特性，如编写一个日期工具类 DateUtils，假设其中有一个解析日期的静态方法 parseDate()，则可以通过 T 操作符直接调用“T(com.smart.DateUtils). parseDate ('20160501')”。

3．构造器

可以使用 new 运算符调用构造器来创建一个对象实例。除了基本类型（如整型、浮点型）和字符串，其他类需要使用全限定类名，如代码清单 9-17 所示。

代码清单 9-17　ObjectExprSample构造器

```
package com.smart.spel;
…
public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();

        User user = parser.parseExpression("new com.smart.User('tom')").getValue(User.
class);
        System.out.println(user.getUserName()); //输出“tom”
    }

}
```

4. 变量

变量可以在表达式中使用语法“#变量名”引用，变量设置使用 EvaluationContex#setVariable(var, value)方法，其中 var 为变量名称，value 为变量对应的值，如代码清单 9-18 所示。

代码清单 9-18　ObjectExprSample变量解析

```
package com.smart.spel;
…
public class ObjectExprSample {
    public static void main(String[] args) {
        User  user = new User("tom");
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext(user);

        //①为newUserName变量设置新值
        context.setVariable("newUserName","jony");

        //②取变量值，并赋值
        parser.parseExpression("userName=#newUserName").getValue(context);
        System.out.println(user.getUserName()); //输出“jony”

        //③ this变量值使用
        List<Integer> credits= new ArrayList<Integer>();
        credits.addAll(Arrays.asList(150,100,90,50,110,130,70));
        context.setVariable("credits",credits);
        List<Integer> creditsGreater100=
              (List<Integer>)parser.parseExpression("#credits.?[#this>100]").
getValue(context);
    }
}
```

在①处通过调用 EvaluationContex#setVariable()方法在动态上下文中定义一个新的变量“newUserName”。在②处的表达式中通过“#变量名”访问这个变量的值，并将取到的值赋给 User 对象的 userName 属性。在③处创建了一个积分列表 credits，并将积分列表设置到动态上下文中。

9.4.10　集合过滤

集合过滤是动态语言的一个强大功能，它允许通过一个过滤条件获取原集合的子集。集合过滤的语法为“?[selectExpression]”。例如，从一个积分列表中找到大于 100 的积分，代码如下：

```
List<Integer> creditsGreater100=
     (List<Integer>)parser.parseExpression("#credits.?[#this>100]").getValue(context);
```

对于 List 或 Set 来说，过滤条件是针对集合内的每个元素进行比较的；而对于 Map 来说，则是针对每一个条目项（Java 类型的 Map.Entry 对象）进行运算的，如代码清单 9-19 所示。

代码清单 9-19　ObjectExprSample集合选择

```
package com.smart.spel;
…
public class ObjectExprSample {
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext();
        Map<String,Integer> creditsMap = new HashMap();
        creditsMap.put("Tom",95);
        creditsMap.put("Jony",110);
        creditsMap.put("Morin",85);
        creditsMap.put("Mose",120);
        creditsMap.put("Morrow",60);
        context.setVariable("credits",creditsMap);

        Map<String,Integer> creditsGreater100=

          (Map<String,Integer>)parser.parseExpression("#credits.?[value>90]").getVal
          ue(context);
    }
}
```

“#credits.?[value>90]”表达式将返回一个新的 Map，即过滤出值大于 100 的条目。默认返回所有匹配的结果，也可以仅返回第一个或最后一个匹配项：用^[…]获得第一个匹配值，如#credits.^[value>90]；用$[…]获得最后一个匹配值，如#credits. $[value>90]。

9.4.11　集合转换

集合转换允许使用一个算子对集合内的元素进行运算，得到一个新的集合。集合转换的语法为“！[projectionExpression]”。来看一个具体的例子，如代码清单 9-20 所示。

代码清单 9-20　ObjectExprSample集合转换

```
package com.smart.spel;
…
public class ObjectExprSample {
    public static void main(String[] args) {
```

```
        ExpressionParser parser = new SpelExpressionParser();
        EvaluationContext context = new StandardEvaluationContext();

        List<Integer> credits= new ArrayList<Integer>();
        credits.addAll(Arrays.asList(150,100,90,50,110,130,70));
        context.setVariable("credits",credits);

        List<Boolean> creditsGreater100=
              (List<Boolean>)parser.parseExpression("#credits.![#this>100]").getValue
(context);
    }
}
```

在示例中，对 List 集合元素进行转换，得到一个判断积分是否大于 100 的结果集合：[true,false,false,true,true,false]。类似的，Map 也可以进行转换，此时转换表达式是对 Map 中的每个项目运用算子进行求值。

9.5 在 Spring 中使用 SpEL

在 XML 配置方式或注解配置方式中都可以使用 SpEL 表达式进行一些高级配置，两种方式都采用统一的语法使用 SpEL 表达式：#{ <expression string> }。

9.5.1 基于 XML 的配置

在 Bean 配置中，可以使用 SpEL 表达式为 Bean 属性或构造函数入参注入动态值。下面以 Bean 属性使用 SpEL 表达式为例，如下：

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess"
    p:randomNumber= "#{T(java.lang.Math).random()*100.0}"
 />
```

在示例中，通过 SpEL 提供的 T 类型操作符，直接调用 java.lang.Math 的静态方法来生成一个随机数，并把生成的随机数乘以 100.0 后赋值给 NumberGuess 的 randomNumber 属性。还可以通过 systemProperties 获取各个系统环境变量，如下：

```
<bean id="systemPropertyBean" class="com.smart.spel.SystemPropertyBean"
      p:osName="#{systemProperties['os.name']}"
      p:javaHome="#{systemProperties['os.name']}"
       p:classPath="#{systemProperties['java.class.path']}"
       p:javaVersion="#{systemProperties['java.class.path']}"
/>
```

可以在 Bean 配置时引用任何一个在 Spring 容器中已定义的 Bean 的属性(属性提供了相应的 get 方法)，通过“Bean 名称.属性名”的方式来引用其他 Bean 的属性，如下：

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess"
    p:randomNumber= "#{T(java.lang.Math).random()*100.0}"
 />
<bean id="shapeGuess" class="org.spring.samples.ShapeGuess"
```

```
    p:initialShapeSeed = "#{numberGuess.randomNumber}"
/>
```

在上述代码中，定义了 NumberGuess 和 ShapeGuess 两个 Bean，ShapeGuess 中的 initialShapeSeed 属性通过表达式引用 NumberGuess 中定义的 randomNumber 属性。

9.5.2　基于注解的配置

@Value 注解可以标注在类的属性、方法及构造器函数上，用于从配置文件中加载一个参数值。下面是一个设置属性示例：

```
@Component
public class MyDataSource {

    @Value("#{properties['driverClassName']}")
    private String driverClassName;

    @Value("#{properties['url']}")
    private String url;

    @Value("#{properties['userName']}")
    private String userName;

    @Value("#{properties['password']}")
    private String password;

}
```

上述代码通过@Value 注解自动注入属性配置文件中的属性选项值，其中 properties 是定义的加载配置文件的 Bean 的名称。要让上述代码能够正常运行，需要在 Spring 中引入 util 工具命名空间，如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       …
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context- 4.0.xsd">
    <util:properties id="properties" location="classpath:jdbc.properties" />
</beans>
```

上述 jdbc.properties 配置文件的配置选项如下：

```
driverClassName=com.mysql.jdbc.Driver
 url=jdbc:mysql://localhost:3306/dbName
 userName=root
 password=1234
```

细心的读者可能会发现，类似“#{properties['password']}”这种写法比较容易出错。Spring 提供了一种更简便的写法——属性占位符，只要在 Spring 的配置方法中添加一个“**property-placeholder**”，就可以在表达式中使用“${属性}”，如下：

```
 <?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       …
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-4.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <util:properties id="properties" location="classpath:jdbc.properties" />
    <context:property-placeholder properties-ref="properties" />
</beans>
```

可以把上述代码改为如下形式：

```
@Component
public class MyDataSource {

    @Value("${driverClassName}")
    private String driverClassName;

    @Value("${url}")
    private String url;
  …
}
```

在 Bean 的方法及构造器函数上使用@Value 注解与类属性的使用方法类似，这里不再赘述。

9.6 小结

在实际项目开发中，对于动态表达式的需求场景还是比较多的。早期，大家可能会在 Java 中引入一些动态脚本语言如 Rhino、Jython、JRuby 等类库来实现。虽然这些动态脚本语言可以实现各类复杂的表达式需求，但需要自行封装实现，工作量很大。

SpEL 表达式的出现为我们提供了一个轻量级的表达式框架，它实现了一套丰富的操作表达式，支持文本、对象、集合、方法表达式解析，并提供了丰富的表达式操作，可以满足大多数表达式场景需求。SpEL 已深入整合到 Spring 框架的 Bean 配置中，使用 SpEL 可以完成众多高级的 Bean 配置问题。

第3篇

数 据 篇

第 10 章

Spring 对 DAO 的支持

随着持久化技术的持续发展，各种持久化框架已趋于成熟，Oracle 也发布了几个持久化规范。Spring 对多个持久化技术提供了集成支持，包括 Hibernate、MyBatis、JPA、JDO；此外，还提供一个简化 JDBC API 操作的 Spring JDBC 框架。Spring 面向 DAO 制定了一个通用的异常体系，屏蔽具体持久化技术的异常，使业务层和具体的持久化技术实现解耦。另外，Spring 提供了模板类简化各种持久化技术的使用。通用的异常体系及模板类是 Spring 整合各种持久化技术的不二法门，Spring 不但借此实现了对多种持久化技术的整合，还可以整合潜在的各种持久化框架，体现了开发模式中“开-闭原则”的经典应用。

本章主要内容：

- Spring DAO 异常体系
- Spring 数据访问模板
- 配置数据源

本章亮点：

- 了解 Spring DAO 层的设计思想
- 详细描述数据源的配置

10.1 Spring 的 DAO 理念

DAO（Data Access Object）是用于访问数据的对象，虽然在大多数情况下将数据保存在数据库中，但这并不是唯一的选择，也可以将数据存储到文件中或 LDAP 中。DAO 不但屏蔽了数据存储的最终介质的不同，也屏蔽了具体的实现技术的不同。

早期，JDBC 是访问数据库的主流选择。近几年，数据持久化技术获得了长足的发

展，Hibernate、MyBatis、JPA、JDO 成为持久层中争放异彩的实现技术。只要为数据访问定义好 DAO 接口，并使用具体的技术实现 DAO 接口的功能，就可以在不同的实现技术间平滑地切换。

图 10-1 是一个典型的 DAO 应用实例，在 UserDao 中定义访问 User 数据对象的接口方法，业务层通过 UserDao 操作数据，并使用具体的持久化技术实现 UserDao 接口方法，这样业务层和具体的持久化技术就实现了解耦。

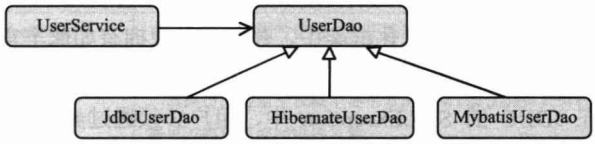


图 10-1　业务层通过 DAO 接口访问数据

提供 DAO 层的抽象可以带来一些好处：首先，可以很容易地构造模拟对象，方便单元测试的开展；其次，在使用切面时会有更多的选择，既可以使用 JDK 动态代理，又可以使用 CGLib 动态代理。

Spring 本质上希望以统一的方式整合底层的持久化技术，即以统一的方式进行调用及事务管理，避免让具体的实现侵入到业务层的代码中。由于每种持久化技术都有各自的异常体系，所以 Spring 提供了统一的异常体系，使不同异常体系的阻抗得以消弭，方便定义出和具体实现技术无关的 DAO 接口，以及整合到相同的事务管理体系中。

10.2　统一的异常体系

统一的异常体系是整合不同的持久化技术的关键。Spring 提供了一套和实现技术无关的、面向 DAO 层语义的异常体系，并通过转换器将不同持久化技术的异常转换成 Spring 的异常。

10.2.1　Spring 的 DAO 异常体系

在很多正统 API 或框架中，检查型异常被过多地使用，以至在使用 API 时，代码里充斥着大量 try/catch 样板式的代码。在很多情况下，除在 try/catch 中记录异常信息外，并没有做多少实质性的工作。引发异常的问题往往是不可恢复的，如数据连接失败、SQL 语句存在语法错误等。强制捕捉的检查型异常除限制开发人员的自由外，并没有提供什么有价值的东西。因此，Spring 的异常体系都是建立在运行期异常的基础上的，开发者可以根据需要捕捉感兴趣的异常。

JDK 的很多 API 之所以难用，一个很大的原因就是检查型异常的泛滥，如 JavaMail、EJB、JDBC 等。使用这些 API，一堆堆异常处理的代码喧宾夺主地侵入到业务代码中，

破坏了代码的整洁和优雅。

Spring 在 org.springframework.dao 包中提供了一套完备优雅的 DAO 异常体系，这些异常都继承于 DataAccessException，而 DataAccessException 本身又继承于 NestedRuntimeException，NestedRuntimeException 异常以嵌套的方式封装了源异常。因此，虽然不同持久化技术的特定异常被转换到 Spring 的 DAO 异常体系中，但原始的异常信息并不会丢失；只要用户愿意，就可以方便地通过 getCause()方法获取原始的异常信息。

Spring 的 DAO 异常体系并不和具体的实现技术相关，它从 DAO 概念的抽象层面定义了异常的目录树。在所有的持久化框架中，并没有发现拥有如此丰富语义的异常体系的框架。Spring 的这种设计无疑是独具匠心的，它使得开发人员关注某一特定语义的异常变得很容易。在 JDBC 的 SQLException 中，用户必须通过异常的 getErrorCode()或 getSQLState()方法获取错误代码，然后根据这些代码判断错误原因。这种过于底层的 API 不但带来了代码编程上的难度，而且也使代码的移植变得困难，因为 getErrorCode()方法是数据库相关的。

Spring 以分类手法建立了异常分类目录，对于大部分应用来说，这个异常分类目录对异常类型的划分具有适当的颗粒度。一方面，使开发者从底层细如针麻的技术细节中脱离出来；另一方面，可以从这个语义丰富的异常体系中选择感兴趣的异常加以处理。图 10-2 列出了那些位于 Spring DAO 异常体系第一层次的异常类，每个异常类下可能拥有众多的子异常类。

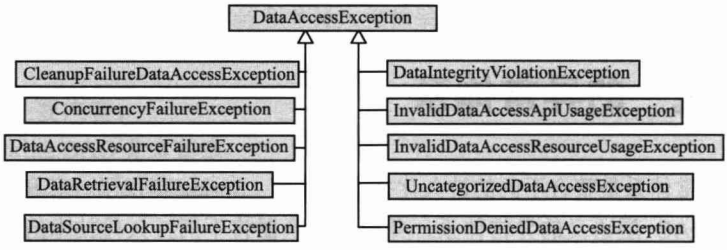


图 10-2　Spring DAO 异常体系

Spring DAO 异常体系类非常丰富，这里仅列出 DataAccessException 异常类下的子类。可以很容易地通过异常类的名字了解异常所代表的语义。下面通过表 10-1 对这些异常进行简单的描述。

表 10-1　Spring DAO异常体系类

异　常	说　明
CleanupFailureDataAccessException	DAO 操作成功执行，但在释放数据资源时发生异常，如关闭 Connection 时发生异常等
ConcurrencyFailureException	表示在进行并发数据操作时发生异常，如乐观锁无法获取、悲观锁无法获取、死锁引发的失败等
DataAccessResourceFailureException	访问数据资源时失败，如无法获取数据连接、无法获取 Hibernate 的会话等
DataRetrievalFailureException	获取数据失败，如找不到对应主键的数据、使用了错误的列索引等

续表

异　常	说　明
DataSourceLookupFailureException	无法从 JNDI 中查找到数据源
DataIntegrityViolationException	当数据操作违反了数据一致性限制时抛出的异常，如插入重复的主键、引用不存在的外键等
InvalidDataAccessApiUsageException	不正确地调用某一持久化技术时抛出的异常，如在 Spring JDBC 中查询对象，在调用前必须进行编译操作，如果忘记这项操作则会产生该异常。这种异常不是由底层数据资源产生的，而是由不正确地使用持久化技术产生的
InvalidDataAccessResourceUsageException	在访问数据源时使用了不正确的方法所抛出的异常，如 SQL 语句错误将抛出该异常
PermissionDeniedDataAccessException	数据访问时由于权限不足引发的异常，如仅拥有只读权限的用户试图进行数据更改操作时将抛出该异常
UncategorizedDataAccessException	其他未分类的异常都归到该异常中

为了进一步细化错误的问题域，Spring 对一级异常类进行了子类的细分，如 InvalidDataAccessResourceUsageException 就拥有十多个子类，下面是其中的两个子类，如图 10-3 所示。

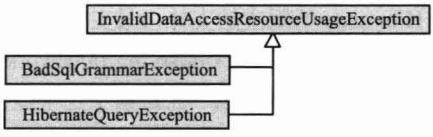


图 10-3　一级异常类的细分

对于 InvalidDataAccessResourceUsageException 异常，不同的持久化技术均有对应的子异常类。如 BadSqlGrammarException 对应 JDBC 实现技术的 SQL 语句语法错误异常，而 HibernateQueryException 对应 Hibernate 实现技术的查询语法异常。

Spring 的这个异常体系具有高度的可扩展性，当 Spring 需要对一种新的持久化技术提供支持时，只要为其定义一个对应的子异常就可以了，这种更改完全满足设计模式中的“开-闭原则”。

虽然 Spring 定义了如此丰富的异常类，但作为开发人员，仅需对感兴趣的异常进行处理即可。假设某个项目要求在发生乐观锁异常时，尝试再次获取乐观锁，而不是直接返回错误；那么，只需在代码中显式捕捉 ConcurrencyFailureException 异常，然后在 catch 代码块中编写满足需求的逻辑即可。其他众多的异常则可以简单地交由框架自动处理，如发生运行期异常时自动回滚事务。

10.2.2　JDBC 的异常转换器

传统的 JDBC API 在发生几乎所有的数据操作问题时都会抛出相同的 SQLException，它将异常的细节性信息封装在异常属性中。所以，如果希望了解异常的具体原因，则必须分析异常对象的信息。

SQLException 拥有两个代表异常具体原因的属性：错误码和 SQL 状态码。前者是数据库相关的，可通过 getErrorCode()方法返回，其值的类型是 int；而后者是一个标准的错误代码，可通过 getSQLState()方法返回，是一个 String 类型的值，由 5 个字符组成。

Spring 根据错误码和 SQL 状态码信息将 SQLException 译成 Spring DAO 的异常体系所对应的异常。在 org.springframework.jdbc.support 包中定义了 SQLExceptionTranslator 接口，该接口的两个实现类 SQLErrorCodeSQLExceptionTranslator 和 SQLStateSQLExceptionTranslator 分别负责处理 SQLException 中错误码和 SQL 状态码的翻译工作。将 SQLException 翻译成 Spring DAO 异常体系的工作是比较困难的，但 Spring 框架替我们完成了这项艰巨的工作并保证了转换的正确性，我们有充分的理由依赖这个转换的正确性。

10.2.3 其他持久化技术的异常转换器

由于各种框架级的持久化技术都拥有一个语义明确的异常体系，所以将这些异常转换为 Spring DAO 的体系相对轻松一些。下面将学习不同持久化技术的异常转换器。

Spring 4.0 移除了对 Hibernate 低版本的支持，只支持 Hibernate 3.6 之后的版本。另外，Spring 4.0 移除了对 TopLink 的支持。在 org.springframework.orm 包中，分别为 Spring 所支持的 ORM 持久化技术定义了一个子包，在这些子包中提供相应 ORM 技术的整合类。Spring 为各种 ORM 持久化技术所提供的异常转换器在表 10-2 中说明。

表 10-2 各ORM持久化技术异常转换器

ORM 持久化技术	异常转换器
Hibernate X.0（X 可为 3,4,5，下同）	org.springframework.orm.hibernateX.SessionFactoryUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

这些工具类除了具有异常转换的功能，在进行事务管理时，还提供了从事务上下文中返回相同会话的功能（将在第 11 章深入讲解事务管理的知识）。

Spring 也支持 MyBatis ORM 持久化技术，由于 MyBatis 抛出的异常是和 JDBC 相同的 SQLException 异常，所以直接采用和 JDBC 相同的异常转换器。

10.3 统一数据访问模板

到一个餐馆用餐，大抵会经历这样一个流程：进入餐馆→迎宾小姐问候并引到适合的位置→拿起菜单点菜→用餐→埋单→离开餐馆。之所以我们喜欢时不时到餐馆用餐，就是因为我们只要点菜→用餐→埋单就可以了，幕后的烹饪制作、刷锅洗盘等工作完全不用关心，一切已经由餐馆服务人员按照服务流程按部就班、有条不紊地执行了。衡量

一个餐馆服务质量好坏的一个重要标准是我们无须关心他们所负责的流程：不用催问菜为什么还没有上好（不但快而且服务态度佳），不用关心盘子为什么不干净（不但干净而且已经进行了消毒）。

从某种角度看，与其说餐馆为我们提供了服务，还不如说我们参与到餐馆的流程中：不管什么顾客点的菜都由相同的厨师烹制，不管什么顾客都按单付钱。在幕后，餐馆拥有一个服务模板，模板中定义的流程可以用于应付所有的顾客，只要为顾客提供几个专有需求（点的菜可不一样，座位可以自由选择），其他一切都按模板化的方式处理。

在直接使用具体的持久化技术时，大多需要处理整个流程，并没有享受餐馆用餐式的便捷。Spring 为支持的持久化技术分别提供了模板访问的方式，降低了使用各种持久化技术的难度，因此可以大幅度地提高开发效率。

10.3.1　使用模板和回调机制

下面是一段使用 JDBC 进行数据访问操作的简单代码，我们已经尽可能地简化了整个过程的处理，但以下步骤都是不可或缺的，如代码清单 10-1 所示。

代码清单 10-1　JDBC数据访问

```
public void saveCustomer(Customer customer) throws Exception {

   Connection con=null;
   PreparedStatement stmt=null;
    try {

     //①获取资源
     con=getConnection();

     //②启动事务
     con.setAutoCommit(false);

     //③具体的数据访问操作和处理
     stmt=con.prepareStatement("insert into CUSTOMERS(ID,NAME) values(?,?)");
     stmt.setLong(1,customerId);
     stmt.setString(2,customer.getName());
     stmt.execute();
     ...
     stmt.execute();

     //④提交事务
     con.commit();
    }catch(Exception e){
    try{
      //⑤回滚事务
      con.rollback();
     }catch(SQLException sqlex){
       sqlex.printStackTrace(System.out);
     }
     throw e;
```

```
    }finally{
     //⑥释放资源
     try{
        stmt.close();
        con.close();
      }catch(Exception e){
        e.printStackTrace();
      }
    }
}
```

如上述代码所示，JDBC 数据访问操作按以下流程进行：

（1）准备资源。

（2）启动事务。

（3）在事务中执行具体的数据访问操作。

（4）提交/回滚事务。

（5）关闭资源，处理异常。

按照传统的方式，在编写任何带事务的数据访问程序时，都需要重复编写上面的代码，而其中只有粗体部分所示的代码是业务相关的，其他代码都是在例行公事，因而导致大量八股文式的代码充斥着整个程序。

轻松一刻

八股文的每篇文章均由一定的格式、字数构成，即由破题、承题、起讲、入手、起股、中股、后股、束股八部分组成。破题是用两句话将题目的意义破开；承题是承接破题的意义而说明之；起讲为议论的开始，首二字用“意味”、“若曰”、“以为”、“且夫”、“尝思”等开端；入手为起讲；起股、中股、后股、束股才是正式议论，以中股为全篇重心。在这四股中，每股又有两股排比对偶的文字，合共八股，故名八股文。

Spring 将这个相同的数据访问流程固化到模板类中，并将数据访问中固定和变化的部分分开，同时保证模板类是线程安全的，以便多个数据访问线程共享同一个模板实例。固定的部分在模板类中已经准备好，而变化的部分通过回调接口开放出来，用于定义具体数据访问和结果返回的操作。图 10-4 描述了模板类拆分固定和变化部分的逻辑。

这样，只要编写好回调接口，并调用模板类进行数据访问，就可以得到预想的结果：数据访问成功执行，前置和后置的样板化工作也按顺序正确执行，在提高开发效率的同时保证了资源使用的正确性，彻底消除了因忘记进行资源释放而引起的资源泄露问题。

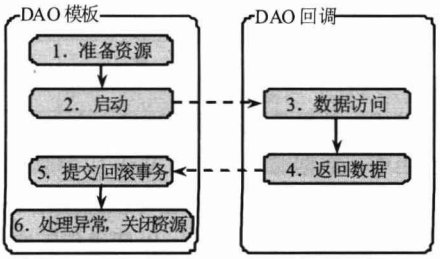


图 10-4　Spring DAO 模板和回调

10.3.2　Spring 为不同持久化技术所提供的模板类

Spring 为各种支持的持久化技术都提供了简化操作的模板和回调，在回调中编写具体的数据操作逻辑，使用模板执行数据操作，在 Spring 中，这是典型的数据操作模式。下面来了解一下 Spring 为不同的持久化技术所提供的模板类，如表 10-3 所示。关于模板类的具体使用，将在本书后续章节介绍。

表 10-3　不同持久化技术对应的模板类

ORM 持久化技术	模　板　类
JDBC	org.springframework.jdbc.core. JdbcTemplate
Hibernate X.0	org.springframework.orm.hibernateX.HibernateTemplate
JPA	org.springframework.orm.jpa.JpaTemplate
JDO	org.springframework.orm.jdo.JdoTemplate

如果直接使用模板类，则一般需要在 DAO 中定义一个模板对象并提供数据资源。Spring 为每种持久化技术都提供了支持类，支持类中已经完成了这样的功能。这样，只需扩展这些支持类，就可以直接编写实际的数据访问逻辑，因此更加方便。

不同持久化技术的支持类如表 10-4 所示。

表 10-4　持久化技术的支持类

ORM 持久化技术	支　持　类
JDBC	org.springframework.jdbc.core. JdbcDaoSupport
Hibernate X.0	org.springframework.orm.hibernateX.HibernateDaoSupport
JPA	org.springframework.orm.jpa.JpaDaoSupport
JDO	org.springframework.orm.jdo.JdoDaoSupport

这些支持类都继承于 dao.support.DaoSupport 类，DaoSupport 类实现了 InitializingBean 接口，在 afterPropertiesSet()接口方法中检查模板对象和数据源是否被正确设置，否则将抛出异常。

所有的支持类都是 abstract 的，其目的是希望被继承使用，而非直接使用。

10.4 数据源

不管采用何种持久化技术，都必须拥有数据连接。在 Spring 中，数据连接是通过数据源获得的。在以往的应用中，数据源一般是由 Web 应用服务器提供的。在 Spring 中，不但可以通过 JNDI 获取应用服务器的数据源，也可以直接在 Spring 容器中配置数据源。此外，还可以通过代码的方式创建一个数据源，以便进行无容器依赖的单元测试。

10.4.1 配置一个数据源

Spring 在第三方依赖包中包含了两个数据源的实现类包：其一是 Apache 的 DBCP；其二是 C3P0。可以在 Spring 配置文件中利用二者中的任何一个配置数据源。

1. DBCP 数据源

DBCP 是一个依赖 Jakarta commons-pool 对象池机制的数据库连接池，所以在类路径下还必须包括 commons-pool 的类包。下面是使用 DBCP 配置 MySQL 数据源的片段：

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"
     p: driverClassName="com.mysql.jdbc.Driver"
      p: url="jdbc:mysql://localhost:3309/sampledb"
      p: username="root"
      p: password="1234"/>
```

BasicDataSource 提供了 close()方法关闭数据源，所以必须设定 destroy-method="close"属性，以便 Spring 容器关闭时，数据源能够正常关闭。

假设数据库是 MySQL，如果数据源配置不当，则将可能发生经典的“8 小时问题”。原因是 MySQL 在默认情况下如果发现一个连接的空闲时间超过 8 小时，则将会在数据库端自动关闭这个连接。而数据源并不知道这个连接已经被数据库关闭了，当它将这个无用的连接返回给某个 DAO 时，DAO 就会报无法获取 Connection 的异常。

除以上必需的数据源属性外，还有一些常用的属性，如表 10-5 所示。

表 10-5 DBCP设置参数说明

分类	属 性	默认值	说 明
事务属性	defaultAutoCommit	true	连接池创建的连接的默认 auto-commit 状态
	defaultReadOnly	驱动默认	连接池创建的连接的默认 read-only 状态。如果没有设置，则 setReadOnly()方法将不会被调用
	defaultTransactionIsolation	驱动默认	连接池创建的连接的默认的 TransactionIsolation 状态，可选值包括：NONE、READ_COMMITTED、READ_UNCOMMITTED、REPEATABLE_READ 和 SERIALIZABLE

续表

分类	属　性	默认值	说　明
数据源连接数量	initialSize	0	初始化连接：连接池启动时创建的初始化连接数量
	maxActive	8	最大活动连接：连接池在同一时间能够分配的最大活动连接数量。如果设置为非正数，则表示不限制
	maxIdle	8	最大空闲连接：连接池中允许保持空闲状态的最大连接数量，超过的空闲连接将被释放。如果设置为负数，则表示不限制
	minIdle	0	最小空闲连接：连接池中允许保持空闲状态的最小连接数量，低于这个数量将创建新的连接。如果设置为 0，则不创建
	maxWait	无限	最大等待时间：当没有可用连接时，连接池等待连接被归还的最大时间（以毫秒计数），超过时间则抛出异常。如果设置为 -1，则表示无限等待
连接健康情况维护和检测	validationQuery	无默认值	SQL 查询语句。在将连接返回给调用者之前，用此 SQL 验证从连接池取出的连接是否可用。如果指定，则查询必须是一个 SQL SELECT，并且必须返回至少一行记录。在 MySQL 中可以设置为“select 1”，在 Oracle 中可以设置为“select 1 from dual”
	testOnBorrow	true	指明是否在从连接池中取出连接前进行检验，如果检验失败，则从连接池中去除该连接并尝试取出另一个新的连接。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	testOnReturn	false	指明是否在归还到连接池中前进行检验。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	testWhileIdle	false	指明连接是否被空闲连接回收器（如果有）进行检验。如果检测失败，则连接将被从连接池中去除。注意：设置为 true 后如果要生效，则 validationQuery 参数必须正确设置
	timeBetweenEvictionRunsMillis	-1	空闲连接回收器线程运行的周期，以毫秒为单位。如果设置为非正数，则不运行空闲连接回收器线程。注意：启用该参数时，validationQuery 参数必须正确设置
	numTestsPerEvictionRun	3	在每次空闲连接回收器线程（如果有）运行时检查的连接数量
	minEvictableIdleTimeMillis	1000 × 60 × 30	连接在可被空闲连接回收器回收前已经在连接池中的空闲时间，以毫秒为单位
缓存语句	poolPreparedStatements	false	开启连接池的 prepared statement 池功能。当设置为 true 时，所有 CallableStatement 和 PreparedStatemen 都会被缓存起来
	maxOpenPreparedStatements	无限制	PreparedStatement 池能够同时分配的打开的 statements 的最大数量。如果设置为 0，则表示不限制
连接泄露回收	removeAbandoned	false	标记是否删除泄露的连接。如果 removeAbandoned 设置为 true，那么“存在泄露嫌疑”的连接可能被连接池主动清除。这个机制在(getNumIdle() < 2) and (getNumActive() > getMaxActive() - 3)条件满足时被触发。举例来说：当 maxActive=20，活动连接为 18，空闲连接为 1 时，可以触发 removeAbandoned 动作。但是活动连接只有在未被使用的时间超过 removeAbandoned Timeout 时才被回收，默认为 300 秒。如果应用需要一个进行长操作的连接，则需要考虑将 removeAbandonedTimeout 设置得更长一些，否则可能发生正常连接被强制清除的情况

续表

分类	属　性	默认值	说　明
连接泄露回收	removeAbandonedTimeout	300	泄露的连接可以被回收的超时值，单位为秒
	logAbandoned	false	标记当 Statement 或连接被泄露时是否打印程序的 stack traces 日志。被泄露的 Statements 和连接的日志添加在每个连接打开或者生成新的 Statement，因为需要生成 stack trace

如果采用 DBCP 的默认配置，由于 testOnBorrow 属性的默认值为 true，数据源在将连接将交给 DAO 前，会事先检测这个连接是否是好的，如果连接有问题（在数据库端被关闭），则会取一个其他的连接给 DAO，所以并不会有“8 小时问题”。如果每次将连接交给 DAO 时都检测连接的有效性，那么在高并发的应用中将会带来性能问题，因为它会需要更多的数据库访问请求。

一种推荐的高效方式是：将 testOnBorrow 设置为 false，而将 testWhileIdle 设置为 true，再设置好 timeBetweenEvictionRunsMillis 值。这样，DBCP 将通过一个后台线程定时地对空闲连接进行检测，当发现无用的空闲连接（那些被数据库关闭的连接）时，就会将它们清除掉。只要将 timeBetweenEvictionRunsMillis 值设置为小于 8 小时，那些被 MySQL 关闭的空闲连接就可以被清除出去，从而避免了“8 小时问题”。

当然，MySQL 本身可以通过调整 interactive-timeout（以秒为单位）配置参数，更改空闲连接的过期时间。所以，在设置 timeBetweenEvictionRunsMillis 值时，必须首先获知 MySQL 空闲连接的最大过期时间。

2．C3P0 数据源

C3P0 是一个开放源码的 JDBC 数据源实现项目，实现了 JDBC3 和 JDBC2 扩展规范说明的 Connection 和 Statement 池。下面使用 C3P0 配置一个 Oracle 数据源，代码如下：

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-
                        method="close"
    p: driverClass=" oracle.jdbc.driver.OracleDriver "
    p: jdbcUrl=" jdbc:oracle:thin:@localhost:1521:ora9i "
    p:use="admin"
    p:password="1234"/>
```

ComboPooledDataSource 和 BasicDataSource 一样提供了一个用于关闭数据源的 close()方法，这样就可以保证 Spring 容器关闭时数据源能够被成功释放。

C3P0 拥有比 DBCP 更丰富的配置属性，通过这些属性，可以对数据源进行各种有效的控制。

- acquireIncrement：当连接池中的连接用完时，C3P0 一次性创建新连接的数目。
- acquireRetryAttempts：定义在从数据库获取新连接失败后重复尝试获取的次数，默认为 30。
- acquireRetryDelay：尝试获取连接的间隔时间，单位为毫秒，默认为 1000。
- autoCommitOnClose：连接关闭时默认将所有未提交的操作回滚。默认为 false。

- automaticTestTable：C3P0 将创建一张名为 Test 的空表，并使用其自带的查询语句进行测试。如果定义了这个参数，那么属性 preferredTestQuery 将被忽略。用户不能在这张 Test 表上进行任何操作，它仅为 C3P0 测试所用。默认为 null。
- breakAfterAcquireFailure：获取连接失败将会引起所有等待获取连接的线程抛出异常。但是数据源仍有效保留，并在下次调用 getConnection()方法时继续尝试获取连接。如果设为 true，那么在尝试获取连接失败后，该数据源将声明已断开并永久关闭。默认为 false。
- checkoutTimeout：当连接池用完时，客户端调用 getConnection()方法后等待获取新连接的时间，超时后将抛出 SQLException，如果设为 0 则无限期等待。单位为毫秒，默认为 0。
- connectionTesterClassName：通过实现 ConnectionTester 或 QueryConnectionTester 的类来测试连接，类名需要设置为全限定名。默认为 com.mchange.v2.C3P0.impl.DefaultConnectionTester。
- idleConnectionTestPeriod：间隔多少秒检查所有连接池中的空闲连接。默认为 0，表示不检查。
- initialPoolSize：初始化时创建的连接数，应在 minPoolSize 与 maxPoolSize 之间取值。默认为 3。
- maxIdleTime：最大空闲时间，超过空闲时间的连接将被丢弃。为 0 或负数则永不丢弃。默认为 0。
- maxPoolSize：连接池中保留的最大连接数。默认为 15。
- maxStatements：JDBC 的标准参数，用以控制数据源内加载的 PreparedStatement 数量。但由于预缓存的 Statement 属于单个 Connection 而不是整个连接池，所以设置这个参数需要考虑多方面的因素，如果 maxStatements 与 maxStatements PerConnection 均为 0，则缓存被关闭。默认为 0。
- maxStatementsPerConnection：连接池内单个连接所拥有的最大缓存 Statement 数。默认为 0。
- numHelperThreads：C3P0 是异步操作的，缓慢的 JDBC 操作通过帮助进程完成。扩展这些操作可以有效地提升性能，通过多线程实现多个操作同时被执行。默认为 3。
- preferredTestQuery：定义所有连接测试都执行的测试语句。在使用连接测试的情况下，这个参数能显著提高测试速度。测试的表必须在初始数据源的时候就存在。默认为 null。
- propertyCycle：用户修改系统配置参数执行前最多等待的秒数。默认为 300。
- testConnectionOnCheckout：因性能消耗大，请只在需要的时候使用它。如果设为 true，那么在每个 connection 提交的时候都将校验其有效性。建议使用

idleConnection TestPeriod 或 automaticTestTable 等方法来提升连接测试的性能。默认为 false。

- testConnectionOnCheckin：如果设为 true，那么在取得连接的同时将校验连接的有效性。默认为 false。

对于连接有效性的检测，请参照我们推荐的 DBCP 配置方式。

3．使用属性文件

数据源的配置信息有可能经常需要改动，同时可能被其他工程复用。此外，用户名/密码等信息比较敏感，可能需要使用特别的安全措施。所以一般将数据源的配置信息独立到一个属性文件中，通过<context:property-placeholder>引入属性文件，以${xxx}的方式引用属性。示例代码如下：

```
<context:property-placeholder
      location="/WEB-INF/jdbc.properties "/>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p: driverClassName="${jdbc.driverClassName}"
      p: url="${jdbc.url}"
      p: username="${jdbc.username}"
      p: password="${jdbc.password}"/>
```

在 jdbc.properties 属性文件中定义属性值，如下：

```
jdbc.driverClassName= com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3309/sampledb
jdbc.username=root
jdbc.password=1234
```

这里，属性文件的内容是以明文方式存放的。如果需要对属性文件的内容进行加密，请参见 6.3 节。

提示

经常有开发者在${xxx}前后不小心输入一些空格，这些空格字符将和变量合并后作为属性的值。如 p: username="␣${jdbc.username}␣"的属性配置项，在前后都有空格，被解析后，username 的值为“␣1234␣”，这将造成最终的错误，因此需要特别小心。

10.4.2 获取 JNDI 数据源

如果应用配置在高性能的应用服务器（如 WebLogic 或 WebSphere 等）上，则可能更希望使用应用服务器本身提供的数据源。应用服务器的数据源使用 JNDI 开放调用者使用，Spring 为此专门提供了引用 JNDI 数据源的 JndiObjectFactoryBean 类。下面是一个简单的配置：

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean"
      p: jndiName="java:comp/env/jdbc/bbt"/>
```

通过 jndiName 指定引用的 JNDI 数据源名称。

Spring 为获取 Java EE 资源提供了一个 jee 命名空间，通过 jee 命名空间，可以有效地简化 Java EE 资源的引用。下面是使用 jee 命名空间引用 JNDI 数据源的配置：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">
   <jee:jndi-lookup id="dataSource" jndi-name=" java:comp/env/jdbc/bbt"/>
</beans>
```

10.4.3　Spring 的数据源实现类

Spring 本身也提供了一个简单的数据源实现类 DriverManagerDataSource，它位于 org.springframework.jdbc.datasource 包中。这个类实现了 javax.sql.DataSource 接口，但它并没有提供池化连接的机制；每次调用 getConnection()方法获取新连接时，只是简单地创建一个新的连接。因此，这个数据源类比较适合在单元测试或简单的独立应用中使用，因为它不需要额外的依赖类。

下面来看一下 DriverManagerDataSource 类的简单使用，如下：

```
DriverManagerDataSource ds = new DriverManagerDataSource ();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3309/sampledb");
ds.setUsername("root");
ds.setPassword("1234");
Connection actualCon = ds.getConnection();
```

当然，也可以通过配置的方式直接使用 DriverManagerDataSource 类。

10.5　小结

Spring 支持目前大多数常用的数据持久化技术。Spring 定义了一套面向 DAO 层的异常体系，并为各种支持的持久化技术提供了异常转换器。这样，在设计 DAO 接口时，就可以抛开具体的实现技术，定义统一的接口。

不管采用何种持久化技术，访问数据的流程是相对固定的。Spring 将数据访问流程划分为固定和变化两部分，并以模板的方式定义好流程，用回调接口将变化的部分开放出来，留给开发者自行定义。这样，仅需提供业务相关的逻辑就可以完成整体的数据访问。Spring 为了进一步简化持久化模板类的使用，为各种持久化技术提供了便捷的支持

类。支持类不但包含数据访问模板，还包含数据源或会话等内容。通过扩展支持类定义自己的数据访问类是最简单的数据访问方式。

不管采用何种持久化技术，都需要定义数据源。在实际部署时，可能会直接采用应用服务器本身提供的数据源，这时可以通过 JndiObjectFactoryBean 或 jee 命名空间引用 JNDI 数据源。

第 11 章

Spring 的事务管理

在使用 Spring 开发应用时，Spring 的事务管理可能是被使用最多、应用最广的功能。对于大多数应用者来说，Spring 事务管理所带来的好处是实实在在的。学习 Spring 而忽略了事务管理，就像到了黄山不登天都峰、讲隋唐演义不讲李元霸一样，最精彩的部分被忽略了。Spring 不但提供了和底层事务源无关的事务抽象，还提供了声明性事务的功能，可以让程序从事务代码中解放出来。而这正是 EJB 自鸣得意的地方，但现在我们有了更好、更便捷的替代方案。

本章主要内容：

- 事务属性值的实际意义
- ThreadLocal 的工作机制
- Spring 事务管理的体系结构
- 基于 XML 的事务配置
- 基于注解的事务配置

本章亮点：

- 介绍数据库事务的基础知识，它们是掌握事务配置的基础
- 深入学习 ThreadLocal 知识，进而揭示 Spring 事务同步管理器的工作原理
- Spring 注解 AspectJ 织入的过程

11.1 数据库事务基础知识

Spring 虽然提供了灵活方便的事务管理功能，但这些功能都是基于底层数据库本身的事务处理机制工作的。要深入了解 Spring 的事务管理和配置，有必要先对数据库事务的基础知识进行学习。

11.1.1 何为数据库事务

“一荣俱荣，一损俱损”这句话很能体现事务的思想，很多复杂的事物要分步进行，但它们组成了一个整体，要么整体生效，要么整体失效。这种思想反映到数据库上，就是多条 SQL 语句，要么所有执行成功，要么所有执行失败。

数据库事务有严格的定义，它必须同时满足 4 个特性：原子性（Atomic）、一致性（Consistency）、隔离性（Isolation）和持久性（Durabiliy），简称为 ACID。下面是对每个特性的说明。

- 原子性：表示组成一个事务的多个数据库操作是一个不可分割的原子单元，只有所有的操作执行成功，整个事务才提交。事务中的任何一个数据库操作失败，已经执行的任何操作都必须撤销，让数据库返回到初始状态。
- 一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的，即数据不会被破坏。如从 A 账户转账 100 元到 B 账户，不管操作成功与否，A 账户和 B 账户的存款总额是不变的。
- 隔离性：在并发数据操作时，不同的事务拥有各自的数据空间，它们的操作不会对对方产生干扰。准确地说，并非要求做到完全无干扰。数据库规定了多种事务隔离级别，不同的隔离级别对应不同的干扰程度，隔离级别越高，数据一致性越好，但并发性越弱。
- 持久性：一旦事务提交成功后，事务中所有的数据操作都必须被持久化到数据库中。即使在提交事务后，数据库马上崩溃，在数据库重启时，也必须保证能够通过某种机制恢复数据。

在这些事务特性中，数据“一致性”是最终目标，其他特性都是为达到这个目标而采取的措施、要求或手段。

数据库管理系统一般采用重执行日志来保证原子性、一致性和持久性。重执行日志记录了数据库变化的每一个动作，数据库在一个事务中执行一部分操作后发生错误退出，数据库即可根据重执行日志撤销已经执行的操作。此外，对于已经提交的事务，即使数据库崩溃，在重启数据库时也能够根据日志对尚未持久化的数据进行相应的重执行操作。

和 Java 程序采用对象锁机制进行线程同步类似，数据库管理系统采用数据库锁机制保证事务的隔离性。当多个事务试图对相同的数据进行操作时，只有持有锁的事务才能操作数据，直到前一个事务完成后，后面的事务才有机会对数据进行操作。Oracle 数据库还使用了数据版本的机制，在回滚段为数据的每个变化都保存一个版本，使数据的更改不影响数据的读取。

11.1.2　数据并发的问题

一个数据库可能拥有多个访问客户端，这些客户端都可用并发的方式访问数据库。数据库中的相同数据可能同时被多个事务访问，如果没有采取必要的隔离措施，就会导致各种并发问题，破坏数据的完整性。这些问题可以归结为 5 类，包括 3 类数据读问题（脏读、不可重复读和幻象读）及 2 类数据更新问题（第一类丢失更新和第二类丢失更新）。下面分别通过实例讲解引发问题的场景。

1．脏读（dirty read）

A 事务读取 B 事务尚未提交的更改数据，并在这个数据的基础上进行操作。如果恰巧 B 事务回滚，那么 A 事务读到的数据根本是不被承认的。来看取款事务和转账事务并发时引发的脏读场景。

时　间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4		取出 500 元，把余额改为 500 元
T5	查询账户余额为 500 元（脏读）	
T6		撤销事务，余额恢复为 1000 元
T7	汇入 100 元，把余额改为 600 元	
T8	提交事务	

在这个场景中，B 希望取款 500 元，而后又撤销了动作，而 A 往相同的账户中转账 100 元，就因为 A 事务读取了 B 事务尚未提交的数据，因而造成账户白白丢失了 500 元。在 Oracle 数据库中，不会发生脏读的情况。

轻松一刻

一个有结巴的人在饮料店柜台前转悠，老板很热情地迎上来说："喝一瓶？"结巴连忙说："我…喝…喝…"老板麻利地打开易拉罐递给结巴，结巴终于憋出了他的那句话："我…喝…喝…喝不起啊！"

2．不可重复读（unrepeatable read）

不可重复读是指 A 事务读取了 B 事务已经提交的更改数据。假设 A 在取款事务的过程中，B 往该账户转账 100 元，A 两次读取账户的余额发生不一致。

时　间	取款事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元

续表

时　间	取款事务 A	转账事务 B
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	查询账户余额为 900 元（和 T4 读取的不一致）	

在同一事务中，T4 时间点和 T7 时间点读取的账户存款余额不一致。

3. 幻象读（phantom read）

A 事务读取 B 事务提交的新增数据，这时 A 事务将出现幻象读的问题。幻象读一般发生在计算统计数据的事务中。举个例子，假设银行系统在同一个事务中两次统计存款账户的总金额，在两次统计过程中，刚好新增了一个存款账户，并存入 100 元，这时，两次统计的总金额将不一致。

时　间	统计金额事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3	统计总存款数为 10000 元	
T4		新增一个存款账户，存款为 100 元
T5		提交事务
T6	再次统计总存款数为 10100 元（幻象读）	

如果新增数据刚好满足事务的查询条件，那么这个新数据就进入了事务的视野，因而产生了两次统计结果不一致的情况。

幻象读和不可重复读是两个容易混淆的概念，前者是指读到了其他已经提交事务的新增数据，而后者是指读到了已经提交事务的更改数据（更改或删除）。为了避免这两种情况，采取的对策是不同的：防止读到更改数据，只需对操作的数据添加行级锁，阻止操作中的数据发生变化；而防止读到新增数据，则往往需要添加表级锁——将整张表锁定，防止新增数据（Oracle 使用多版本数据的方式实现）。

4. 第一类丢失更新

A 事务撤销时，把已经提交的 B 事务的更新数据覆盖了。这种错误可能造成很严重的问题，通过下面的账户取款转账就可以看出来。

时　间	取款事务 A	转账事务 B
T1	开始事务	
T2		开始事务
T3	查询账户余额为 1000 元	
T4		查询账户余额为 1000 元
T5		汇入 100 元，把余额改为 1100 元
T6		提交事务
T7	取出 100 元，把余额改为 900 元	

续表

时　间	取款事务 A	转账事务 B
T8	**撤销事务**	
T9	余额恢复为 1000 元（丢失更新）	

A 事务在撤销时，“不小心”将 B 事务已经转入账户的金额给抹去了。

5. 第二类丢失更新

A 事务覆盖 B 事务已经提交的数据，造成 B 事务所做操作丢失。

时　间	转账事务 A	取款事务 B
T1		**开始事务**
T2	**开始事务**	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		**提交事务**
T7	汇入 100 元	
T8	**提交事务**	
T9	把余额改为 1100 元（丢失更新）	

在上面的例子中，由于支票转账事务覆盖了取款事务对存款余额所做的更新，导致银行最后损失了 100 元。相反，如果转账事务先提交，那么用户账户将损失 100 元。

11.1.3　数据库锁机制

数据并发会引发很多问题，在一些场合下有些问题是允许的，但在另一些场合下可能是致命的。数据库通过锁机制解决并发访问的问题，虽然不同的数据库在实现细节上存在差别，但原理基本上是一样的。

按锁定的对象的不同，一般可以分为表锁定和行锁定。前者对整张表进行锁定，而后者对表中的特定行进行锁定。从并发事务锁定的关系上看，可以分为共享锁定和独占锁定。共享锁定会防止独占锁定，但允许其他的共享锁定。而独占锁定既防止其他的独占锁定，也防止其他的共享锁定。为了更改数据，数据库必须在进行更改的行上施加行独占锁定，INSERT、UPDATE、DELETE 和 SELECT FOR UPDATE 语句都会隐式采用必要的行锁定。下面介绍一下 Oracle 数据库常用的 5 种锁定。

- 行共享锁定：一般通过 SELECT FOR UPDATE 语句隐式获得行共享锁定，在 Oracle 中用户也可以通过 LOCK TABLE IN ROW SHARE MODE 语句显式获得行共享锁定。行共享锁定并不防止对数据行进行更改操作，但是可以防止其他会话获取独占性数据表锁定。允许进行多个并发的行共享和行独占锁定，还允许进行数据表的共享或者采用共享行独占锁定。

- 行独占锁定：通过一条 INSERT、UPDATE 或 DELETE 语句隐式获取，或者通过一条 LOCK TABLE IN ROW EXCLUSIVE MODE 语句显式获取。这种锁定可以防止其他会话获取一个共享锁定、共享行独占锁定或独占锁定。
- 表共享锁定：通过 LOCK TABLE IN SHARE MODE 语句显式获得。这种锁定可以防止其他会话获取行独占锁定（INSERT、UPDATE 或 DELETE），或者防止其他表共享行独占锁定或表独占锁定，但它允许在表中拥有多个行共享和表共享锁定。该锁定可以让会话具有对表事务级一致性访问，因为其他会话在用户提交或者回滚该事务并释放对该表的锁定之前不能更改这张被锁定的表。
- 表共享行独占锁定：通过 LOCK TABLE IN SHARE ROW EXCLUSIVE MODE 语句显式获得。这种锁定可以防止其他会话获取一个表共享、行独占或者表独占锁定，但允许其他行共享锁定。这种锁定类似于表共享锁定，只是一次只能对一张表放置一个表共享行独占锁定。如果 A 会话拥有该锁定，则 B 会话可以执行 SELECT FOR UPDATE 操作；但如果 B 会话试图更新选择的行，则需要等待。
- 表独占锁定：通过 LOCK TABLE IN EXCLUSIVE MODE 语句显式获得。这种锁定可以防止其他会话对该表的任何其他锁定。

11.1.4 事务隔离级别

尽管数据库为用户提供了锁的 DML 操作方式，但直接使用锁管理是非常麻烦的，因此数据库为用户提供了自动锁机制。只要用户指定会话的事务隔离级别，数据库就会分析事务中的 SQL 语句，然后自动为事务操作的数据资源添加适合的锁。此外，数据库还会维护这些锁，当一个资源上的锁数目太多时，自动进行锁升级以提高系统的运行性能，而这一过程对用户来说完全是透明的。

ANSI/ISO SQL 92 标准定义了 4 个等级的事务隔离级别，在相同的数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可能导致不同的结果。不同的事务隔离级别能够解决的数据并发问题的能力是不同的，如表 11-1 所示。

表 11-1 事务隔离级别对并发问题的解决情况

隔离级别	脏 读	不可重复读	幻 象 读	第一类丢失更新	第二类丢失更新
READ UNCOMMITED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

事务的隔离级别和数据库并发性是对立的。一般来说，使用 READ UNCOMMITED 隔离级别的数据库拥有最高的并发性和吞吐量，而使用 SERIALIZABLE 隔离级别的数据库并发性最低。

SQL 92 定义 READ UNCOMMITED 主要是为了提供非阻塞读的能力。Oracle 虽然也支持 READ UNCOMMITED，但它不支持脏读；因为 Oracle 使用多版本机制彻底解决了在非阻塞读时读到脏数据的问题并保证读的一致性，所以，Oracle 的 READ COMMITTED 隔离级别就已经满足了 SQL 92 标准的 REPEATABLE READ 隔离级别。

SQL 92 推荐使用 REPEATABLE READ 以保证数据的读一致性，不过用户可以根据应用的需要选择适合的隔离等级。

11.1.5　JDBC 对事务的支持

并不是所有的数据库都支持事务，即使支持事务的数据库也并非支持所有的事务隔离级别。用户可以通过 Connection#getMetaData()方法获取 DatabaseMetaData 对象，并通过该对象的 supportsTransactions()、supportsTransactionIsolationLevel(int level)方法查看底层数据库的事务支持情况。

Connection 默认情况下是自动提交的，即每条执行的 SQL 语句都对应一个事务。为了将多条 SQL 语句当成一个事务执行，必须先通过 Connection#setAutoCommit(false)阻止 Connection 自动提交，并通过 Connection#setTransactionIsolation()设置事务的隔离级别。Connection 中定义了对应 SQL 92 标准 4 个事务隔离级别的常量。通过 Connection #commit()提交事务，通过 Connection#rollback()回滚事务。下面是典型的 JDBC 事务数据操作的代码，如代码清单 11-1 所示。

代码清单 11-1　JDBC事务代码

```
Connection conn ;
try{
    conn = DriverManager.getConnection();①          获取数据连接
    conn.setAutoCommit(false); ②                    关闭自动提交机制
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE); ③
    Statement stmt = conn.createStatement();                          设置事务隔离级别

    int rows = stmt.executeUpdate( "INSERT INTO t_topic VALUES(1,'tom') " );
    rows = stmt.executeUpdate( "UPDATE t_user set topic_nums = topic_nums +1 "+
                                                    "WHERE user_id = 1");

    conn.commit();④                                 提交事务
  }catch(Exception e){
    ...
    conn.rollback();⑤                               回滚事务
  }finally{
   ...
}
```

在 JDBC 2.0 中，事务最终只能有两个操作：提交和回滚。但是，有些应用可能需要对事务进行更多的控制，而不是简单地提交或回滚。JDBC 3.0（Java 1.4 及以后的版本）引入了一个全新的保存点特性，Savepoint 接口允许用户将事务分割为多个阶段，用户可以指定回滚到事务的特定保存点，而并非像 JDBC 2.0 一样只能回滚到开始事务的点，如图 11-1 所示。

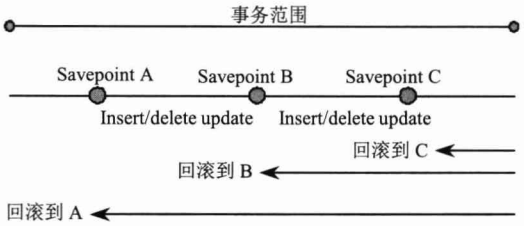


图 11-1　带 Savepoint 的事务

下面的代码使用了保存点功能，在发生特定问题时，回滚到指定的保存点，而非回滚整个事务，如代码清单 11-2 所示。

代码清单 11-2　使用保存点的事务代码

```
...
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate( "INSERT INTO t_topic VALUES(1,'tom')");

Savepoint svpt = conn.setSavepoint("savePoint1");①   ← 设置一个保存点
rows = stmt.executeUpdate( "UPDATE t_user set topic_nums = topic_nums +1 "+
                                                    "WHERE user_id = 1");
...
conn.rollback(svpt); ②   ← 回滚到①处的 savePoint1，①之前的 SQL 操作，在整个事务提交后依然提交，但①到②之间的 SQL 操作被撤销了
...
conn.commit();③   ← 提交事务
```

并非所有数据库都支持保存点功能，用户可以通过 DatabaseMetaData#supportsSavepoints()方法查看是否支持。

11.2　ThreadLocal 基础知识

在第 10 章中我们知道，Spring 通过各种模板类降低了开发者使用各种数据持久化技术的难度。这些模板类都是线程安全的，也就是说，多个 DAO 可以复用同一个模板实例而不会发生冲突。使用模板类访问底层数据，根据持久化技术的不同，模板类需要绑定数据连接或会话的资源。但这些资源本身是非线程安全的，也就是说它们不能在同一时刻被多个线程共享。虽然模板类通过资源池获取数据连接或会话，但资源池本身解决的是数据连接或会话的缓存问题，并非数据连接或会话的线程安全问题。

按照传统经验，如果某个对象是非线程安全的，在多线程环境下，对对象的访问必须采用 synchronized 进行线程同步。但模板类并未采用线程同步机制，因为线程同步会降低并发性，影响系统性能。此外，通过代码同步解决线程安全的挑战性很大，可能会增加几倍的实现难度。那么，模板类究竟仰仗何种“魔法神功”，可以在无须线程同步的情况下就化解线程安全的难题呢？答案就是 ThreadLocal！

ThreadLocal 在 Spring 中发挥着重要的作用，在管理 request 作用域的 Bean、事务管

理、任务调度、AOP 等模块中都出现了它的身影。要想了解 Spring 事务管理的底层技术，ThreadLocal 是必须攻克的“山头堡垒”。

11.2.1　ThreadLocal 是什么

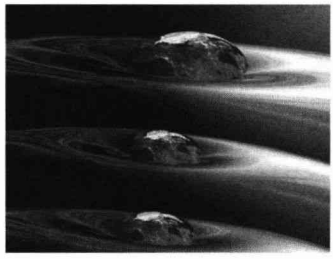
早在 Java 1.2 版本中就提供了 java.lang.ThreadLocal。ThreadLocal 为解决多线程程序的并发问题提供了一种新的思路，使用这个工具类可以很简洁地编写出优美的多线程程序。线程局部变量并不是 Java 的新发明，很多语言（如 IBM XL、FORTRAN）在语法层面就提供了线程局部变量。在 Java 中没有提供语言级支持，而是以一种变通的方法，通过 ThreadLocal 的类提供支持。所以，在 Java 中编写线程局部变量的代码相对来说要笨拙一些，这也是为什么线程局部变量没有在 Java 开发者中得到很好普及的原因。

ThreadLocal，顾名思义，它不是一个线程，而是保存线程本地化对象的容器。当运行于多线程环境的某个对象使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程分配一个独立的变量副本。所以每个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。从线程的角度看，这个变量就像线程专有的本地变量，这也是类名中“Local”所要表达的意思。

InheritableThreadLocal 继承于 ThreadLocal，它自动为子线程复制一份从父线程那里继承而来的本地变量：在创建子线程时，子线程会接收所有可继承的线程本地变量的初始值。当必须将本地线程变量自动传送给所有创建的子线程时，应尽可能地使用 InheritableThreadLocal，而非 ThreadLocal。

轻松一刻

ThreadLocal 很像平行宇宙（Parallel Universes）的概念。平行宇宙来自量子力学，是指多元宇宙中所包含的各个宇宙，一个宇宙从某个宇宙中分离出来，与原宇宙平行存在着既相似又不同的其他宇宙。它们可能处于同一空间体系，但时间体系不同，就好像同在一条铁路线上疾驰的先后两列火车。2011 年，邓肯·琼斯执导的《源代码》就是一部关于平行宇宙的科幻影片。



11.2.2　ThreadLocal 的接口方法

ThreadLocal 类接口很简单，只有 4 个方法，先来了解一下。

- void set(Object value)：设置当前线程的线程局部变量的值。
- public Object get()：返回当前线程所对应的线程局部变量。

- public void remove()：将当前线程局部变量的值删除，目的是为了减少内存的占用。该方法是 Java 5.0 新增的。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必需的操作，但它可以加快内存回收的速度。
- protected Object initialValue()：返回该线程局部变量的初始值。该方法是一个 protected 的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第一次调用 get()或 set(Object)时才执行，并且仅执行一次。ThreadLocal 中的默认实现直接返回一个 null。

值得一提的是，在 Java 5.0 中，ThreadLocal 已经支持泛型，该类的类名已经变为 ThreadLocal<T>。API 方法也相应进行了调整，新版本的 API 方法分别是 void set(T value)、T get()及 T initialValue()。

ThreadLocal 是如何做到为每个线程维护一份独立的变量副本呢？其实实现的思路很简单：在 ThreadLocal 类中有一个 Map，用于存储每个线程的变量副本，Map 中元素的键为线程对象，值为对应线程的变量副本。下面提供一个简单的实现版本，如代码清单 11-3 所示。

代码清单 11-3　SimpleThreadLocal

```
public class SimpleThreadLocal {
    private Map valueMap = Collections.synchronizedMap(new HashMap());
    public void set(Object newValue) {
        valueMap.put(Thread.currentThread(), newValue);①
    }
    public Object get() {
        Thread currentThread = Thread.currentThread();
        Object o = valueMap.get(currentThread); ②
      if (o == null && !valueMap.containsKey(currentThread)) {③
            o = initialValue();
            valueMap.put(currentThread, o);
        }
        return o;
    }
    public void remove() {
        valueMap.remove(Thread.currentThread());
    }
    public Object initialValue() {
        return null;
    }
}
```

键为线程对象，值为本线程的变量副本

返回本线程对应的变量

如果在 Map 中不存在，则放到 Map 中保存起来

虽然代码清单 11-3 中 ThreadLocal 的实现版本显得比较幼稚，但它和 JDK 所提供的 ThreadLocal 类在实现思路上是非常相近的。

11.2.3　一个 TheadLocal 实例

下面通过一个具体的实例了解一下 ThreadLocal 的具体使用方法，如代码清单 11-4 所示。

代码清单 11-4　SequenceNumber

```
package com.smart.basic;

public class SequenceNumber {

    private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>(){①
        public Integer initialValue(){
            return 0;
        }
    };

    public int getNextNum(){②
        seqNum.set(seqNum.get()+1);
        return seqNum.get();
    }

    public static void main(String[ ] args)
    {
        SequenceNumber sn = new SequenceNumber();

        TestClient t1 = new TestClient(sn);  ③
        TestClient t2 = new TestClient(sn);
        TestClient t3 = new TestClient(sn);
        t1.start();
        t2.start();
        t3.start();
    }
    private static class TestClient extends Thread
    {
        private SequenceNumber sn;
        public TestClient(SequenceNumber sn) {
            this.sn = sn;
        }
        public void run()
        {
            for (int i = 0; i < 3; i++) {④
            System.out.println("thread["+Thread.currentThread().getName()+
                    "] sn["+sn.getNextNum()+"]");
            }
        }
    }
}
```

通过匿名内部类覆盖 ThreadLocal 的 initialValue()方法，指定初始值

获取下一个序列值

3 个线程共享 sn，各自产生序列号

每个线程打出 3 个序列值

可通过匿名内部类的方式定义 ThreadLocal 的子类，提供初始的变量值，如①处所示。TestClient 线程产生一组序列号，在③处，生成 3 个 TestClient，它们共享同一个 SequenceNumber 实例。运行以上代码，在控制台上输出以下信息：

```
thread[Thread-2] sn[1]
thread[Thread-0] sn[1]
thread[Thread-1] sn[1]
thread[Thread-2] sn[2]
thread[Thread-0] sn[2]
thread[Thread-1] sn[2]
thread[Thread-2] sn[3]
```

```
thread[Thread-0] sn[3]
thread[Thread-1] sn[3]
```

考查输出的结果信息，发现每个线程所产生的序号虽然都共享同一个 SequenceNumber 实例，但它们并没有相互干扰，而是各自产生独立的序列号，这是因为通过 ThreadLocal 为每个线程提供了单独的副本。

11.2.4 与 Thread 同步机制的比较

ThreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。那么，ThreadLocal 和线程同步机制相比有什么优势呢？

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序缜密地分析什么时候对变量进行读/写、什么时候需要锁定某个对象、什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而 ThreadLocal 从另一个角度来解决多线程的并发访问。ThreadLocal 为每个线程提供了一个独立的变量副本，从而隔离了多个线程对访问数据的冲突。因为每个线程都拥有自己的变量副本，因而也就没有必要对该变量进行同步。ThreadLocal 提供了线程安全的对象封装，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

由于 ThreadLocal 中可以持有任何类型的对象，低版本 JDK 所提供的 get()方法返回的是 Object 对象，需要强制类型转换。但 Java 5.0 通过泛型很好地解决了这个问题，在一定程度上简化了 ThreadLocal 的使用，代码清单 11-2 就使用了 Java 5.0 新的 ThreadLocal<T>版本。

概括而言，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式：访问串行化，对象共享化；而 ThreadLocal 采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问；而后者为每个线程都提供了一份变量，因此可以同时访问而互不影响。

11.2.5 Spring 使用 ThreadLocal 解决线程安全问题

我们知道，在一般情况下，只有无状态的 Bean 才可以在多线程环境下共享。在 Spring 中，绝大部分 Bean 都可以声明为 singleton 作用域。正是因为 Spring 对一些 Bean（如 RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder 等）中非线程安全的“状态性对象”采用 ThreadLocal 进行封装，让它们也成为线程安全的“状态性对象”，因此，有状态的 Bean 就能够以 singleton 的方式在多线程中正常工作。

一般的 Web 应用划分为展现层、服务层和持久层 3 个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程，如图 11-2 所示。

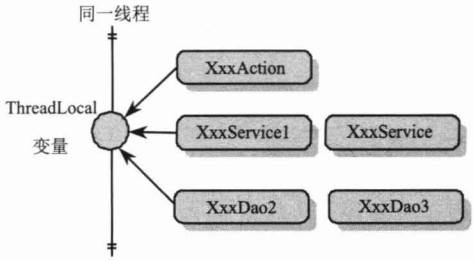


图 11-2　同一线程贯通三层

这样用户就可以根据需要，将一些非线程安全的变量以 ThreadLocal 存放，在同一次请求响应的调用线程中，所有对象所访问的同一 ThreadLocal 变量都是当前线程所绑定的。

下面的实例能够体现 Spring 对有状态 Bean 的改造思路，如代码清单 11-5 所示。

代码清单 11-5　TopicDao：非线程安全

```
public class TopicDao {

  //①一个非线程安全变量
  private Connection conn;

  public void addTopic(){

      //②引用非线程安全变量
      Statement stat = conn.createStatement();
      …
  }
}
```

由于①处的 conn 是非线程安全的成员变量，因此 addTopic()方法也是非线程安全的，所以必须在使用时创建一个新的 TopicDao 实例（非 singleton）。下面使用 ThreadLocal 对 conn 这个非线程安全的“状态”进行改造，如代码清单 11-6 所示。

代码清单 11-6　TopicDao：线程安全

```
import java.sql.Connection;
import java.sql.Statement;
public class TopicDao {

  //①使用ThreadLocal保存Connection变量
  private static ThreadLocal<Connection> connThreadLocal = new
ThreadLocal<Connection>();
  public static Connection getConnection(){

      //②如果connThreadLocal没有本线程对应的Connection，则创建一个新的Connection，
      //并将其保存到线程本地变量中
      if (connThreadLocal.get() == null) {
          Connection conn = ConnectionManager.getConnection();
          connThreadLocal.set(conn);
          return conn;
      }else{
```

```
            //③直接返回线程本地变量
            return connThreadLocal.get();
        }
    }
    public void addTopic() {

        //④从ThreadLocal中获取线程对应的Connection
        Statement stat = getConnection().createStatement();
    }
}
```

不同的线程在使用 TopicDao 时，先判断 connThreadLocal.get()是否为 null，如果为 null，则说明当前线程还没有对应的 Connection 对象，这时创建一个 Connection 对象并添加到本地线程变量中；如果不为 null，则说明当前线程已经拥有了 Connection 对象，直接使用就可以了。这样就保证了不同的线程使用自己独立的 Connection，而不会使用其他线程的 Connection。因此，这个 TopicDao 就可以做成 singleton 的 Bean 了。

当然，这个例子本身很粗糙，将 Connection 的 ThreadLocal 直接放在 DAO 中只能做到本 DAO 的多个方法共享 Connection 时不发生线程安全问题，但无法和其他 DAO 共用同一个 Connection。要做到同一事务多 DAO 共享同一个 Connection，必须在一个共同的外部类使用 ThreadLocal 保存 Connection。但这个实例基本上说明了 Spring 对所有状态类线程安全化的解决思路。在本章后面的内容中，将详细说明 Spring 如何通过 ThreadLocal 解决事务管理的问题。

11.3 Spring 对事务管理的支持

Spring 为事务管理提供了一致的编程模板，在高层次建立了统一的事务抽象。也就是说，不管是选择 Spring JDBC、Hibernate、JPA 还是选择 MyBatis，Spring 都可以让用户用统一的编程模型进行事务管理。

像 Spring DAO 为不同的持久化实现提供了模板类一样，Spring 事务管理继承了这一风格，也提供了事务模板类 TransactionTemplate。通过 TransactionTemplate 并配合使用事务回调 TransactionCallback 指定具体的持久化操作，就可以通过编程方式实现事务管理，而无须关注资源获取、复用、释放、事务同步和异常处理等操作。

Spring 事务管理的亮点在于声明式事务管理。Spring 允许通过声明方式，在 IoC 配置中指定事务的边界和事务属性，Spring 自动在指定的事务边界上应用事务属性。声明式事务是 EJB 烜赫一时的技术，Spring 让这种技术平民化，甚至可以说，Spring 的声明事务比 EJB 的更为强大。

EJB 事务建立在 JTA 的基础上，而 JTA 又必须通过 JNDI 获取。这意味着不管用户的应用是跨数据源的应用，还是单数据源的应用，EJB 都要求使用全局事务的方式加以处理，即基于 EJB 的应用无法脱离应用服务器所提供的容器环境。这种不加区分一概而论的做法无异于杀鸡杀牛都用一把刀。

Spring 深刻地认识到，大部分应用都是基于单数据源的，只有为数不多的应用需要使用多数据源的 JTA 事务。因此，在单数据源的情况下，Spring 直接使用底层的数据源管理事务。只有在面对多数据源的应用时，Spring 才寻求 Java EE 应用服务器的支持，通过引用应用服务器中的 JNDI 资源完成 JTA 事务。Spring 让人印象深刻的地方在于不管用户使用何种持久化技术，也不管用户是否使用了 JTA 事务，都可以采用相同的事务管理模型。这种统一的处理方式所带来的好处是不可估量的，用户完全可以抛开事务管理的问题编写程序，并在 Spring 中通过配置完成事务的管理工作。

11.3.1　事务管理关键抽象

在 Spring 事务管理 SPI（Service Provider Interface）的抽象层主要包括 3 个接口，分别是 PlatformTransactionManager、TransactionDefinition 和 TransactionStatus，它们位于 org.springframework.transaction 包中。通过图 11-3 可以描述这三者的关系。

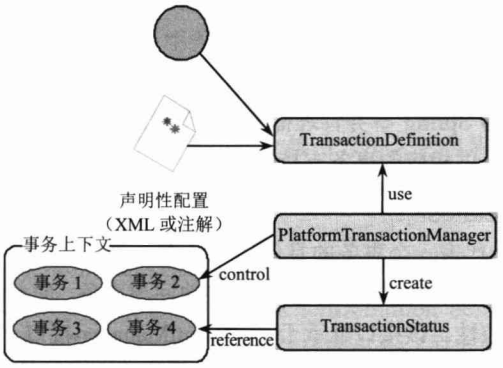


图 11-3　Spring 事务管理 SPI 抽象

TransactionDefinition 用于描述事务的隔离级别、超时时间、是否为只读事务和事务传播规则等控制事务具体行为的事务属性，这些事务属性可以通过 XML 配置或注解描述提供，也可以通过手工编程的方式设置。

PlatformTransactionManager 根据 TransactionDefinition 提供的事务属性配置信息创建事务，并用 TransactionStatus 描述这个激活事务的状态。下面分别来了解这些 SPI 接口内部的组成。

1．TransactionDefinition

TransactionDefinition 定义了 Spring 兼容的事务属性，这些属性对事务管理控制的若干方面进行配置。

- **事务隔离**：当前事务和其他事务的隔离程度。在 TransactionDefinition 接口中，定义了和 java.sql.Connection 接口中同名的 4 个隔离级别：ISOLATION_READ_UNCOMMITTED、ISOLATION_READ_COMMITTED、ISOLATION_REPEATABLE_

READ 和 ISOLATION_SERIALIZABLE（实际上它直接使用 Connection 的同名常量进行赋值），这些常量分别对应于 11.1.4 节所描述的隔离级别。此外，TransactionDefinition 还定义了一个默认的隔离级别——ISOLATION_DEFAULT，它表示使用底层数据库的默认隔离级别。

- **事务传播**：通常在一个事务中执行的所有代码都会运行于同一事务上下文中。但是 Spring 也提供了几个可选的事务传播类型，例如，简单地参与到现有的事务中，或者挂起当前的事务，创建一个新的事务。在 Spring 事务管理中，传播行为是一个重要的概念，将单独在 11.3.4 节详细讲述。Spring 提供了 EJB CMT 所支持的事务传播类型。
- **事务超时**：事务在超时前能运行多久，超过时间后，事务被回滚。有些事务管理器不支持事务过期的功能，这时，如果设置 TIMEOUT_DEFAULT 之外的其他值，则将抛出异常。
- **只读状态**：只读事务不修改任何数据，资源事务管理者可以针对可读事务应用一些优化措施，提高运行性能。只读事务在某些情况下（如使用 Hibernate 时）是一种非常有用的优化，试图在只读事务中更改数据将引发异常。

Spring 允许通过 XML 或注解元数据的方式为一个有事务要求的服务类方法配置事务属性，这些信息作为 Spring 事务管理框架的“输入”，Spring 将自动按事务属性信息的指示，为目标方法提供相应的事务支持。

2．TransactionStatus

TransactionStatus 代表一个事务的具体运行状态。事务管理器可以通过该接口获取事务运行期的状态信息，也可以通过该接口间接地回滚事务，它相比于在抛出异常时回滚事务的方式更具可控性。该接口继承于 SavepointManager 接口，SavepointManager 接口基于 JDBC 3.0 保存点的分段事务控制能力提供了嵌套事务的机制。

SavepointManager 接口拥有以下几个方法。

- Object createSavepoint()：创建一个保存点对象，以便在后面可以利用 rollbackToSavepoint(Object savepoint)方法使事务回滚到特定的保存点上，也可以通过 releaseSavepoint()方法释放一个已经不用的保存点。
- void rollbackToSavepoint(Object savepoint)：将事务回滚到特定的保存点上，被回滚的保存点将自动释放。
- void releaseSavepoint(Object savepoint)：释放一个保存点。如果事务提交，则所有的保存点会被自动释放，无须手工清除。

这 3 个方法在底层的资源不支持保存点时，都将抛出 NestedTransactionNotSupported Exception 异常。

TransactionStatus 扩展了 SavepointManager 接口，并提供了以下几个方法。

- boolean hasSavepoint()：判断当前事务是否在内部创建了一个保存点，该保存点是为了支持 Spring 的嵌套事务而创建的。

- boolean isNewTransaction()：判断当前事务是否是一个新的事务，如果返回 false，则表示当前事务是一个已经存在的事务，或者当前操作未运行在事务环境中。
- boolean isCompleted()：判断当前事务是否已经结束（已经提交或回滚）。
- boolean isRollbackOnly()：判断当前事务是否已经被标识为 rollback-only。
- void setRollbackOnly()：将当前事务设置为 rollback-only。通过该标识通知事务管理器只能将事务回滚，事务管理器将通过显式调用回滚命令或抛出异常的方式回滚事务。

3．PlatformTransactionManager

通过 JDBC 的事务管理知识可以知道，事务只能被提交或回滚（或回滚到某个保存点后提交），Spring 高层事务抽象接口 org.springframework.transaction.PlatformTransactionManager 很好地描述了事务管理这个概念，如代码清单 11-7 所示。

代码清单 11-7　PlatformTransactionManager

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

PlatformTransactionManager 只定义了 3 个接口方法，它们是 SPI（Service Provider Interface）高层次的接口方法。这些访问都没有和 JNDI 绑定在一起，可以像 Spring 容器中普通的 Bean 一样对待 PlatformTransactionManager 实现者。

下面来了解一下 PlatformTransactionManager 接口方法的功能。

- TransactionStatus getTransaction(TransactionDefinition definition)：该方法根据事务定义信息从事务环境中返回一个已存在的事务，或者创建一个新的事务，并用 TransactionStatus 描述这个事务的状态。
- commit(TransactionStatus status)：根据事务的状态提交事务。如果事务状态已经被标识为 rollback-only，则该方法将执行一个回滚事务的操作。
- rollback(TransactionStatus status)：将事务回滚。当 commit()方法抛出异常时，rollback()方法会被隐式调用。

11.3.2　Spring 的事务管理器实现类

Spring 将事务管理委托给底层具体的持久化实现框架来完成。因此，Spring 为不同的持久化框架提供了 PlatformTransactionManager 接口的实现类，如表 11-2 所示。

表 11-2　不同持久化技术对应的事务管理器实现类

事　务	说　明
org.springframework.orm.jpa.JpaTransactionManager	使用 JPA 进行持久化时，使用该事务管理器

续表

事　务	说　明
org.springframework.orm.hibernateX.HibernateTransactionManager	使用 Hibernate X.0（X 可为 3,4,5，下同）版本进行持久化时，使用该事务管理器
org.springframework.jdbc.datasource.DataSource TransactionManager	使用 Spring JDBC 或 MyBatis 等基于 DataSource 数据源的持久化技术时，使用该事务管理器
org.springframework.orm.jdo.JdoTransactionManager	使用 JDO 进行持久化时，使用该事务管理器
org.springframework.transaction.jta.JtaTransactionManager	具有多个数据源的全局事务使用该事务管理器（不管采用何种持久化技术）

这些事务管理器都是对特定事务实现框架的代理，这样就可以通过 Spring 所提交的高级抽象对不同种类的事务实现使用相同的方式进行管理，而不用关心具体的实现。

要实现事务管理，首先要在 Spring 中配置好相应的事务管理器，为事务管理器指定数据资源及一些其他事务管理控制属性。下面来看一下几个常见的事务管理器的配置。

1. Spring JDBC 和 MyBatis

如果使用 Spring JDBC 或 MyBatis，由于它们都基于数据源的 Connection 访问数据库，所以可以使用 DataSourceTransactionManager，只要在 Spring 中进行以下配置就可以了，如代码清单 11-8 所示。

代码清单 11-8　基于数据源的数据管理器

```
...
<bean id="dataSource"①                          配置一个数据源
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"/>
<bean id="transactionManager" ②          基于数据源的事务管理器
       class="org.springframework.jdbc.datasource. DataSourceTransactionManager"
       p:dataSource-ref="dataSource"③                    引用数据源
       />
</bean>
```

在幕后，DataSourceTransactionManager 使用 DataSource 的 Connection 的 commit()、rollback()等方法管理事务。

2. JPA

JPA 通过 javax.persistence.EntityTransaction 管理 JPA 的事务，EntityTransaction 对象可以通过 javax.persistence.EntityManager#getTransaction()方法获得，而 EntityManager 又通过一个工厂类方法 javax.persistence.EntityManagerFactory#createEntityManager()获取。

在底层，JPA 依然通过 JDBC 的 Connection 的事务方法完成最终的控制。因此，要配置一个 JPA 事务管理器，必须先提供一个 DataSource，然后配置一个 EntityManagerFactory，最后才配置 JpaTransactionManager，如代码清单 11-9 所示。

代码清单 11-9　JPA事务管理器配置

```
...
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
      p:dataSource-ref="dataSource"/>①  ← 指定一个数据源
      ...
</bean>
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager"
      p:entityManagerFactory-ref="entityManagerFactory"/>②  ← 指定实体管理器
```

3. Hibernate

Spring 4.0 已取消了对 Hibernate 3.6 之前版本的支持，并全面支持 Hibernate 5.0；因此，只为 Hibernate 3.6+提供了事务管理器。下面以 Hibernate 4.0 为例，如代码清单 11-10 所示。

代码清单 11-10　Hibernate 4.0 事务管理器配置

```
...
<bean id="sessionFactory"                                          指定一个数据源
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
       p:dataSource-ref="dataSource"①
       p:mappingResources="classpath:bbtForum.hbm.xml">②          指定 Hibernate 配置文件

        <property name="hibernateProperties"> ③  ← Hibernate 其他配置属性
            <props>
                <prop key="hibernate.dialect">${hibernate.dialect}</prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.generate_statistics">true</prop>
            </props>
        </property>
</bean>
<bean id="transactionManager"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager"
     p:sessionFactory-ref="sessionFactory"④  ← 注入会话工厂
```

大部分 ORM 框架都拥有自己事务管理的 API，它们对 DataSource 和 Connection 进行了封装。Hibernate 使用 org.hibernate.Session 封装 Connection，所以需要一个能够创建 Session 的 SessionFactory（更多内容请参见第 14 章）。

4. JTA

如果希望在 Java EE 容器里使用 JTA，则将通过 JNDI 和 Spring 的 JtaTransactionManager 获取一个容器管理的 DataSource。JtaTransactionManager 不需要知道 DataSource 和其他特定的资源，因为它引用容器提供的全局事务管理，如代码清单 11-11 所示。

代码清单 11-11　JTA事务管理器

```
<?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:jee="http://www.springframework.org/schema/jee"
```

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">

<jee:jndi-lookup id="accountDs" jndi-name="java:comp/env/jdbc/account"/>①
<jee:jndi-lookup id="orderDs"  jndi-name="java:comp/env/jdbc/order"/>
                                                    通过 jee 命名空间获取
                                                    Java EE 应用服务器容
                                                    器中的数据源
                                    指定 JTA 事务管理器
<bean id="transactionManager"
     class="org.springframework.transaction.jta.JtaTransactionManager"/>②
</beans>
```

这里使用了 jee 命名空间从 Java EE 容器中返回 JNDI 管理的资源。①处的配置相当于：

```
    <bean id="simple"
        class="org.springframework. jndi. JndiObjectFactoryBean"
        p:jndiName =" java:comp/env/jdbc/account "/>
```

显然，使用 jee 命名空间的配置更加简洁。jee 命名空间还可以获取 EJB 本地或远程的无状态 Session Bean（更多信息参见 Spring 参考手册的附录）。

11.3.3 事务同步管理器

Spring 将 JDBC 的 Connection、Hibernate 的 Session 等访问数据库的连接或会话对象统称为资源，这些资源在同一时刻是不能多线程共享的。为了让 DAO、Service 类可能做到 singleton，Spring 的事务同步管理器类 org.springframework.transaction.support.TransactionSynchronizationManager 使用 ThreadLocal 为不同事务线程提供了独立的资源副本，同时维护事务配置的属性和运行状态信息。事务同步管理器是 Spring 事务管理的基石，不管用户使用的是编程式事务管理，还是声明式事务管理，都离不开事务同步管理器。

Spring 框架为不同的持久化技术提供了一套从 TransactionSynchronizationManager 中获取对应线程绑定资源的工具类，如表 11-3 所示。

表 11-3 线程绑定资源获取工具

持久化技术	线程绑定资源获取工具
Spring JDBC 或 MyBatis	org.springframework.jdbc.datasource.DataSourceUtils
Hibernate X.0	org.springframework.orm.hibernateX.SessionFactoryUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

这些工具类都提供了静态的方法，通过这些方法可以获取和当前线程绑定的资源，如 DataSourceUtils.getConnection(DataSource dataSource)方法可以从指定的数据源中获取和当前线程绑定的 Connection，而 Hibernate 的 SessionFactoryUtils.getSession(SessionFactory

sessionFactory, boolean allowCreate)方法则可以从指定的 SessionFactory 中获取和当前线程绑定的 Session。

当需要脱离模板类，手工操作底层持久化技术的原生 API 时，就需要通过这些工具类获取线程绑定的资源，而不应该直接从 DataSource 或 SessionFactory 中获取。因为后者不能获得与本线程相关的资源，因此无法让数据操作参与到与本线程相关的事务环境中。

这些工具类还有另外一个重要的用途：将特定异常转换为 Spring 的 DAO 异常，已经在第 10 章的表 10-2 中对此进行了说明。

Spring 为不同的持久化技术提供了模板类，模板类在内部通过资源获取工具类间接访问 TransactionSynchronizationManager 中的线程绑定资源。所以，如果 DAO 使用模板类进行持久化操作，这些 DAO 就可以配置成 singleton。如果不使用模板类，也可以直接通过资源获取工具类访问线程相关的资源。

下面，让我们揭开 TransactionSynchronizationManager 的层层面纱，探寻其中的奥秘。

```
public abstract class TransactionSynchronizationManager {

    //①用于保存每个事务线程对应的Connection或Session等类型的资源
    private static final ThreadLocal resources = new ThreadLocal();

    //②用于保存每个事务线程对应事务的名称
    private static final ThreadLocal currentTransactionName = new ThreadLocal();

    //③用于保存每个事务线程对应事务的read-only状态
    private static final ThreadLocal currentTransactionReadOnly = new ThreadLocal();

    //④用于保存每个事务线程对应事务的隔离级别
    private static final ThreadLocal currentTransactionIsolationLevel= new
ThreadLocal();

    //⑤用于保存每个事务线程对应事务的激活态
    private static final ThreadLocal actualTransactionActive = new ThreadLocal();
     …
}
```

TransactionSynchronizationManager 将 DAO、Service 类中影响线程安全的所有“状态”统一抽取到该类中，并用 ThreadLocal 进行替换，从此 DAO（必须是基于模板类或资源获取工具类创建的 DAO）和 Service（必须采用 Spring 事务管理机制）摘掉了非线程安全的帽子，完成了脱胎换骨式的身份转变。

11.3.4　事务传播行为

当我们调用一个基于 Spring 的 Service 接口方法（如 UserService#addUser()）时，它将运行于 Spring 管理的事务环境中，Service 接口方法可能会在内部调用其他的 Service 接口方法以共同完成一个完整的业务操作，因此就会产生服务接口方法嵌套调用的情

况，Spring 通过事务传播行为控制当前的事务如何传播到被嵌套调用的目标服务接口方法中。事务传播是 Spring 进行事务管理的重要概念，其重要性怎么强调都不为过。但是事务传播行为也是被误解最多的地方，本节将详细分析不同事务传播行为的表现形式，掌握它们之间的区别。

Spring 在 TransactionDefinition 接口中规定了 7 种类型的事务传播行为，它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播，如表 11-4 所示。

表 11-4　事务传播行为类型

事务传播行为类型	说　明
PROPAGATION_REQUIRED	如果当前没有事务，则新建一个事务；如果已经存在一个事务，则加入到这个事务中。这是最常见的选择
PROPAGATION_SUPPORTS	支持当前事务。如果当前没有事务，则以非事务方式执行
PROPAGATION_MANDATORY	使用当前的事务。如果当前没有事务，则抛出异常
PROPAGATION_REQUIRES_NEW	新建事务。如果当前存在事务，则把当前事务挂起
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作。如果当前存在事务，则把当前事务挂起
PROPAGATION_NEVER	以非事务方式执行。如果当前存在事务，则抛出异常
PROPAGATION_NESTED	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

在使用 PROPAGATION_NESTED 时，底层的数据源必须基于 JDBC 3.0，并且实现者需要支持保存点事务机制。

11.4　编程式的事务管理

在实际应用中，很少需要通过编程来进行事务管理。即便如此，Spring 还是为编程式事务管理提供了模板类 org.springframework.transaction.support.TransactionTemplate，以满足一些特殊场合的需要。

TransactionTemplate 和那些持久化模板类一样是线程安全的，因此，可以在多个业务类中共享 TransactionTemplate 实例进行事务管理。TransactionTemplate 拥有多个设置事务属性的方法，如 setReadOnly(boolean readOnly)、setIsolationLevel(int isolationLevel)等。

虽然很少需要手工编写事务管理的程序，但作为深入认识 Spring 事务管理的一种方式，了解一下编程式事务也不是坏事。TransactionTemplate 有两个主要的方法：

- void setTransactionManager(PlatformTransactionManager transactionManager)：设置事务管理器。
- Object execute(TransactionCallback action)：在 TransactionCallback 回调接口中定义需要以事务方式组织的数据访问逻辑。

TransactionCallback 接口只有一个方法：Object doInTransaction(TransactionStatus status)。如果操作不会返回结果，则可以使用 TransactionCallback 的子接口 TransactionCallbackWithoutResult，如代码清单 11-12 所示。

代码清单 11-12　采用编程式的事务管理

```
...
@Service
public class ForumService1 {
    public ForumDao forumDao;
    public TransactionTemplate template;

    @Autowired          ①                                    通过 AOP 主动注入
    public void setTemplate(TransactionTemplate template) {
        this.template = template;
    }
    public void addForum(final Forum forum) {
        template.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                forumDao.addForum(forum); ②          需要在事务环境
                }                                    中执行的代码
        });
    }
  ...
}
```

由于 Spring 事务管理基于 TransactionSynchronizationManager 进行工作，所以，如果在回调接口方法中需要显式访问底层数据连接，则必须通过资源获取工具类得到线程绑定的数据连接。这是 Spring 事务管理的底层协议，不容违反。如果 ForumDao 是基于 Spring 提供的模板类构建的，由于模板类已经在内部使用了资源获取工具类获取数据连接，所以用户就不必关心底层数据连接的获取问题。

11.5　使用 XML 配置声明式事务

大多数 Spring 用户选择声明式事务管理的功能，这种方式对代码的侵入性最小，可以让事务管理代码完全从业务代码中移除，非常符合非侵入式轻量级容器的理念。

Spring 的声明式事务管理是通过 Spring AOP 实现的，通过事务的声明性信息，Spring 负责将事务管理增强逻辑动态织入业务方法的相应连接点中。这些逻辑包括获取线程绑定资源、开始事务、提交/回滚事务、进行异常转换和处理等工作。

Spring 提供了和 EJB CMT 相似的声明式事务管理，这不但体现在二者的最终执行效果上，还体现在二者事务声明的语法上。但二者也存在明显的不同，下面通过表 11-5 对二者进行比较。

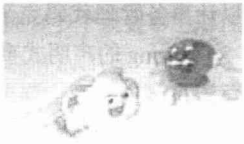
表 11-5　Spring和EJB CMT声明事务的不同

比较项	EJB CMT	Spring
是否绑定 JTA	绑定在 JTA 上，即使单数据源也是如此。所以 EJB 不能脱离容器运行	可以在任何环境下使用，包括直接在 Spring 中声明的数据源或在应用服务器 JNDI 中的 JTA 数据源
持久化技术支持	采用非开放的 EJB 自制持久化技术	通过少量配置即可和 JDBC、JDO、Hibernate 等持久化技术一起工作
目标类要求	必须是实现特定接口的特殊类	可以是任何 POJO，不过在内部必须使用资源获取工具类操作数据连接或会话。如果 DAO 使用模板类进行构建，则这种要求将自动得到满足
回滚规则	没有提供	提供声明式的回滚规则
开放性控制	使用 EJB CMT，除了使用 setRollbackOnly()方法，没有办法影响容器的事务管理	Spring 允许用户通过 AOP 定制事务行为。例如，如果需要，用户可以在事务回滚中插入定制的行为，也可以增加任意的增强，就和任何 AOP 的增强一样
分布式事务	支持分布式事务。一般应用并不需要使用这样的功能	Spring 不直接支持高端应用服务器所提供的跨越远程调用的事务上下文传播。此时，可以通过 Spring 的 Java EE 服务集成来提供。此外，在 Spring 中集成 JOTM 后，Spring 也可以提供 JTA 事务的功能

回滚规则的概念比较重要，它使用户能够指定什么样的异常导致自动回滚、什么样的异常不影响事务提交，这些规则可以在配置文件中通过声明的方式指定；同时，仍然可以通过调用 TransactionStatus#setRollbackOnly()方法编程式地回滚当前事务。通常我们定义一条规则，如声明 MyApplicationException 必须总是导致事务回滚。这种方式带来了显著的好处，它使用户的业务对象不必依赖于事务设施。典型的例子是用户不必在代码中导入 Spring API、事务代码等。

对 EJB 来说，默认的行为是 EJB 容器在遇到系统异常（通常指运行时异常）时自动回滚当前事务。EJB 遇到应用异常（通常指检查型异常）时并不会自动回滚。默认情况下，Spring 的声明式事务管理和 EJB 的相同（只在遇到运行期异常时自动回滚）。

轻松一刻



在讲解 Spring 时，经常会涉及的一个问题是“框架的侵入性”。从广义上讲，“侵入性”就是对应用编程行为的影响，从这个意义上说，任何框架都逃脱不了“侵入性”的控诉。不过反过来一想：框架不对应用产生任何影响，我们还需要框架做什么呢？关键问题是这种影响是怎样施加的。EJB 以一种强权政治的方式强力扭转程序员的固有习惯，让应用开发人员的日子很难过；而 Spring 则以一种春风化雨、喜闻乐见的方式引入它的“侵入性”——这不禁让我们想起了那则著名的“太阳和风”的寓言。

11.5.1　一个将被实施事务增强的服务接口

BbtForum 拥有 4 个方法，我们希望 addTopic()和 updateForum()方法拥有写事务的能力，而其他两个方法只需要拥有读事务的能力就可以了。以下是 BbtForum 的实现代码，如代码清单 11-13 所示。

代码清单 11-13　BbtForum

```
package com.smart.service;
...
@Service
@Transactional
public class BbtForum {
    public ForumDao forumDao;
    public TopicDao topicDao;
    public PostDao postDao;
    public void addTopic(Topic topic) {
        topicDao.addTopic(topic);
        postDao.addPost(topic.getPost());
    }
    public Forum getForum(int forumId) {
        return forumDao.getForum(forumId);
    }
    public void updateForum(Forum forum) {
        forumDao.updateForum(forum);
    }
    public int getForumNum() {
        return forumDao.getForumNum();
    }
    ...
}
```

BbtForum 是一个 POJO，只是简单地使用持久层的多个 DAO 类，通过它们的协作实现业务功能。在这里，我们看不到任何事务操作的代码，所以如果直接使用 BbtForum，那么这些方法都将以无事务的方式运行。现在，我们的任务是通过 Spring 声明式事务配置让这些业务方法拥有适合的事务功能。

11.5.2　使用原始的 TransactionProxyFactoryBean

在 Spring 的早期版本中，用户必须通过 TransactionProxyFactoryBean 代理类对需要事务管理的业务类进行代理，以便实施事务功能的增强。在 Spring 2.0 后，由于可以通过 aop/tx 命名空间声明事务，因此通过该代理类实施声明式事务的方式已经不被推荐。

1．声明式事务配置

从循序渐进的学习角度来看，了解 TransactionProxyFactoryBean 有助于我们更直观地理解 Spring 实施声明式事务的内部工作原理。所以我们不妨把使用这个代理类作为学习 Spring 事务管理的起始站，后面的章节会陆续介绍 Spring 的配置方式，如代码清单 11-14 所示。

代码清单 11-14　使用TransactionProxyFactoryBean配置

```
<!--①引入DAO和DataSource的配置文件-->
<import resource="classpath:applicationContext-dao.xml" />

<!--②声明事务管理器-->
<bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!--③需要实施事务增强的目标业务Bean -->
<bean id="bbtForumTarget"
        class="package com.smart.service.BbtForum"
        p:forumDao-ref="forumDao"
        p:topicDao-ref="topicDao"
        p:postDao-ref="postDao"/>

<bean id="bbtForum" ④ <-- 使用事务代理工厂类为目标业务 Bean 提供事务增强
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean
"
    p:transactionManager-ref="txManager" ④-1 <-- 指定事务管理器
    p:target-ref="bbtForumTarget"> ④-2
                                   <-- 指定目标业务 Bean
    <property name="transactionAttributes">④-3
                                   <-- 事务属性配置
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>④-4 <-- 只读事务
            <prop key="*">PROPAGATION_REQUIRED</prop>④-5
                                   <-- 可写事务
        </props>
    </property>
</bean>
```

当然，必须首先配置好 DAO Bean、数据源等基础设施，这些信息统一在 applicationContext-dao.xml 配置文件中定义，如①处所示（读者可以参看本书配套网盘中的具体配置）。在这个示例中，我们使用 Spring JDBC 的持久化技术，所以它的资源池是 JDBC 数据源，而事务管理器是 DataSourceTransactionManager。如果使用其他持久化技术，则需要对基础设施和事务管理器进行相应的调整。

按照约定的习惯，需要事务增强的业务类一般将 id 取名为 xxxTarget，如③处所示，这可以在字面上表示该 Bean 是要被代理的目标 Bean。

通过 TransactionProxyFactoryBean 对业务类进行代理，织入事务增强功能，如④处所示。首先，需要为该代理类指定事务管理器，这些事务管理器实现了 PlatformTransactionManager 接口；其次，通过 target 属性指定需要代理的目标 Bean；最后，为业务 Bean 的不同方法配置事务属性。Spring 允许通过键值配置业务方法的事务属性信息，键可以使用通配符，如 get*代表目标类中所有以 get 为前缀的方法，它匹配 BbtForum 的 getForum(int forumId)和 getForumNum()方法；而 key="*"匹配 BbtForum 接口的所有方法。

提示

如果将 TransactionProxyFactoryBean 的 optimize 属性设为 true，即添加<property name="optimize" value="true"/>，则 Spring 自动通过 CGLib 动态代理为 BbtForumImpl 生成基于子类的代理。这时，key="*"不但匹配 BbtForum 接口的所有方法，也匹配 BbtForumImpl 中的其他方法，如 setForumDao()、setTopicDao()等方法都将织入事务管理的增强。

2．异常回滚/提交规则

<prop>内的值为事务属性信息，其配置格式如图 11-4 所示。

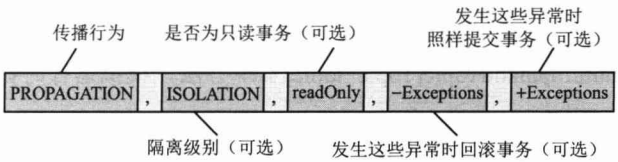


图 11-4　事务属性设置格式

传播行为是唯一必须提供的配置项，当<prop>的值为空时，也不会发生配置异常，但对应的匹配方法不会应用事务，相当于没有配置。可选的值参看表 11-4 中的定义。

隔离级别配置项是可选的，默认为 ISOLATION_DEFAULT，表示使用数据库默认的隔离级别。隔离级别其他可选的设置值为（具体意义参看 11.1.4 节的解释）：

- ISOLATION_READ_UNCOMMITTED。
- ISOLATION_READ_COMMITTED。
- ISOLATION_REPEATABLE_READ。
- ISOLATION_SERIALIZABLE。

如果希望将匹配方法设置为只读事务，则可添加 readOnly 配置项，如代码清单 11-14 中的④-4 所示。

当事务运行过程中发生异常时，事务可以被声明为回滚或继续提交。在默认情况下，当发生运行期异常时，事务将被回滚；当发生检查型异常时，既不回滚也不提交，控制权交给外层调用。这种默认的回滚规则在大多数情况下是适用的，不过用户也可以通过配置显式指定回滚规则：指定带正号（+）或负号（-）的异常类名（或异常名匹配片段）。当抛出负号型异常时，将触发事务回滚；当抛出正号型异常时，即使这个异常是检查型异常，事务也会提交。抛出的异常或该异常的祖先类的类名匹配规则中指定了异常类名（或异常名片段），规则就生效，如下面的设置：

```
<prop key="add*">PROPAGATION_REQUIRED, -Exception</prop>
```

只要业务方法运行时抛出的异常或其父类异常的类名包括“Exception”，事务都回滚。以下异常都符合这条规则：SQLException、ParseException。正因为 Spring 采用名称字符串包含的比较方式，所以用户甚至可以将回滚的规则设置为“-tion”。

因为 Spring 默认的事务回滚规则为“运行期异常回滚，检查型异常不回滚”，所以带负号的异常设置仅对检查型异常有意义。

此外，用户可以指定多个带正号或负号的事务回滚/提交规则，如下：

```
<prop key="add*">
  PROPAGATION_REQUIRED, -XxxException,-YyyException
</prop>
```

3. 异常提交的示例

举一个关于异常回滚的具体例子。假设希望 BbtForum#addTopic(Topic topic)在抛出 PessimisticLockingFailureException 异常时，依旧提交事务，则对应的配置如下：

```
<prop key="addTopic">
  PROPAGATION_REQUIRED, +PessimisticLockingFailureException
</prop>
```

调整 BbtForumImpl#addTopic(Topic topic)实现方法，模拟运行时抛出 PessimisticLockingFailureException。

```
public class BbtForum {
    ...
public void addTopic(Topic topic) throws Exception {
        topicDao.addTopic(topic);①
        if(true) throw new PessimisticLockingFailureException("fail");
        postDao.addPost(topic.getPost());②
    }
    ...
}
```

依旧会提交此处的数据持久化操作

此处的数据持久化操作无法到达

在运行这个方法时，虽然在①和②之间抛出了异常，但 Spring 最终将使①处的数据持久化操作生效，而②处的数据持久化操作将因为前面抛出了异常而无法执行。

11.5.3 基于 aop/tx 命名空间的配置

使用 TransactionProxyFactoryBean 代理工厂类为业务类添加事务性支持，在学习了动态代理的知识后，理解起来应该比较直观，但它拥有以下一些明显的缺点：

- 需要对每个需要事务支持的业务类进行单独的配置。
- 在指定事务方法时，只能通过方法名进行定义，无法利用方法签名的其他信息进行定位（如方法入参、访问域修饰符等）。
- 事务属性的配置串的规则比较麻烦，规则串虽然包括多项信息，但统一由逗号分隔的字符串来描述，不能利用 IDE 中的诱导输入功能，容易出错。
- 在为业务类 Bean 添加事务支持时，在容器中既需要定义业务类 Bean（通常命名为 xxxTarget），又需要通过 TransactionProxyFactoryBean 对其进行代理以生成支持事务的代理 Bean。实际上，我们只会从容器中返回代理的 Bean，而业务类 Bean 仅是为了能被代理才定义的；这样就造成相似的东西有两份配置，增加了配置信息量。

这一切都是因为在低版本的 Spring 中没有引入强大的 AOP 切面描述语言而造成的。

Spring 2.0 的一个重要改进是引入了 AspectJ 切面定义语言，借由这股强劲的“东风”，事务方法切面描述的难题就迎刃而解了。

Spring 在基于 Schema 的配置中添加了一个 tx 命名空间，在配置文件中以明确结构化的方式定义事务属性，大大提高了配置事务属性的便利性。配合 aop 命名空间所提供的切面定义这把“利剑”，业务类方法事务配置得到了极大的简化，而在描述能力上也得到了很大的提升。

下面通过 tx 和 aop 命名空间对上一节中基于 FactoryBean 的事务配置方式进行替换，如代码清单 11-5 所示。

代码清单 11-15　applicationContext-tx.xml：使用aop/tx命令空间

```
<?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"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
    <import resource="classpath:applicationContext-dao.xml" />

   <!--不再需要为了事务AOP增强的实施而改名换姓-->

    <!--①事务管理器-->
    <bean id="txManager"
         class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
         p:dataSource-ref="dataSource"/>

   <!--②使用强大的切点表达式语言轻松定义目标方法-->
   <aop:config>
        <!--②-1通过aop定义事务增强切面-->
        <aop:pointcut id="serviceMethod"
                 expression="execution(* com.smart.service.*Forum.*(..))" />
       <!--②-2引用事务增强-->
        <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
    </aop:config>

    <!--③事务增强-->
    <tx:advice id="txAdvice" transaction-manager="txManager">
      <!--③-1事务属性定义-->
      <tx:attributes>
         <tx:method name="get*" read-only="false"/>
         <tx:method name="add*" rollback-for="Exception" />
         <tx:method name="update*"/>
      </tx:attributes>
   </tx:advice>
</beans>
```

首先需要在配置文件中引入 tx 命名空间的声明，如<beans>元素粗体部分所示。采

用 aop/tx 定义事务方法时，它站在“局外人”的角度对 IoC 容器中的 Bean 进行事务管理配置定义，再由 Spring 将这些配置织入对应的 Bean 中。

在这一过程中我们看到了 3 种角色：通过 aop/tx 定义的声明式事务配置信息、业务 Bean、Spring 容器。Spring 容器自动将第一者应用于第二者，从容器中返回的业务 Bean 已经是被织入事务增强的代理 Bean，即第一者和第二者在配置时不直接发生关系。

而在使用 TransactionProxyFactoryBean 进行事务配置时，TransactionProxyFactoryBean 需要直接通过 target 属性引用目标业务 Bean，结果造成目标业务 Bean 往往需要使用 target 进行命名（如 userServiceTarget），以避免和最终代理 Bean 名称（如 userService）冲突。使用 aop/tx 方式后，业务 Bean 的名称不需要做任何“配合性”的调整，aop 直接通过切点表达式语言就可以对业务 Bean 进行定位。从这个意义上来说，aop/tx 的配置方式对业务 Bean 是“无侵入”的，而 TransactionProxyFactoryBean 的配置显然是“侵入式”的。

在 aop 命名空间中，通过切点表达式语言，将 com.smart.service 包下所有以 Forum 为后缀的类纳入了需要进行事务增强的范围，并配合<tx:advice>的<aop:advisor>完成了事务切面的定义，如②处所示。

<aop:advisor>引用的 txAdvice 增强是在 tx 命名空间上定义的，如③处所示。首先，事务增强一定需要一个事务管理器的支持，<tx:advice>通过 transaction-manager 属性引用了①处定义的事务管理器（它默认查找名为 transactionManager 的事务管理器，所以如果事务管理器命名为 transactionManager，则可以不指定 transaction-manager 属性）。在③-1 中我们看到，原来掺杂在一起，以逗号分隔字符串定义的事务属性，现在变成了一个结构清晰的 XML 片段。相信无须进行过多的解释，大家就可以明白这个片段中元素及属性的配置语义了。<tx:method>元素拥有的属性如表 11-6 所示。

表 11-6　<tx:method>元素属性表

属　性	是否必需	默 认 值	描　述
name	是	与事务属性关联的方法名，可使用通配符（*）	如“get*”、“handle*”、“on*Event”等
propagation	否	REQUIRED	事务传播行为，可选的值为：REQUIRED SUPPORTS、MANDATORY、REQUIRES_NEW、NOT_SUPPORTED、NEVER、NESTED
isolation	否	DEFAULT	事务隔离级别，可选的值为：DEFAULT、READ_UNCOMMITTED READ_COMMITTED、REPEATABLE_READ SERIALIZABLE
timeout	否	-1	事务超时的时间（以秒为单位）。如果设置为-1，则事务超时的时间由底层的事务系统所决定
read-only	否	false	事务是否只读
rollback-for	否	所有运行期异常回滚	触发事务回滚的 Exception，用异常名称的片段进行匹配。可以设置多个，以逗号分开，如“Exception1, Exception2”

续表

属　性	是否必须	默 认 值	描　述
no-rollback-for	否	所有检查型异常不回滚	不触发事务回滚的 Exception，用异常名称的片段进行匹配。可以设置多个，以逗号分开，如“Exception1, Exception2”

如果需要为不同的业务类 Bean 应用不同的事务管理风格，则可以在<aop:config>中定义另外多套事务切面，具体的配置方法在现有基础上演绎即可，此处不再赘述。关于基于 Schema 配置切面的基本知识，请参见第 8 章。

提示

基于 aop/tx 配置的声明式事务管理是实际应用中最常使用的事务管理方式，它的表达能力最强且使用最为灵活。

11.6　使用注解配置声明式事务

除了基于 XML 的事务配置，Spring 还提供了基于注解的事务配置，即通过@Transactional 对需要事务增强的 Bean 接口、实现类或方法进行标注；在容器中配置基于注解的事务增强驱动，即可启用基于注解的声明式事务。笔者在实际项目中一般采用这种配置方式。

11.6.1　使用@Transactional 注解

顺着原来的思路，使用@Transactional 对基于 aop/tx 命名空间的事务配置进行改造，比较二者在使用方式上的差异，如代码清单 11-16 所示。

代码清单 11-16　使用@Transactional注解的业务类

```
package com.smart.service;
import org.springframework.transaction.annotation.Transactional;
...
//①对业务类进行事务增强的标注
@Service
@Transactional
public class BbtForum {
    public void addTopic(Topic topic) throws Exception {
        topicDao.addTopic(topic);
        postDao.addPost(topic.getPost());
    }
    public Forum getForum(int forumId) {
        return forumDao.getForum(forumId);
    }
    ...
}
```

因为注解本身具有一组普适性的默认事务属性，所以往往只要在需要事务管理的业务类中添加一个@Transactional 注解，就完成了业务类事务属性的配置。

当然，注解只提供元数据，它本身并不能完成事务切面织入的功能。因此，还需要在 Spring 配置文件中通过一行小小的配置“通知”Spring 容器对标注@Transactional 注解的 Bean 进行加工处理，如代码清单 11-17 所示。

代码清单 11-17　applicationContext-anno.xml：使事务注解生效

```
<?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-4.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    <context:component-scan base-package="com.smart"/>
    <import resource="classpath:applicationContext-dao.xml" />
    <bean id="txManager"
        class="org.springframework.jdbc. datasource. DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>

    <!--①对标注@Transactional注解的Bean进行加工处理，以织入事务管理切面-->
    <tx:annotation-driven transaction-manager="txManager"/>
</beans>
```

在默认情况下，<tx:annotation-driven>会自动使用名为“transactionManager”的事务管理器。所以，如果用户的事务管理器 id 为“transactionManager”，则可以进一步将①处的配置简化为“<tx:annotation-driven/>”。

<tx:annotation-driven>还有另外两个属性。

- proxy-target-class：如果为 true，则 Spring 将通过创建子类来代理业务类；如果为 false，则使用基于接口的代理。如果使用子类代理，则需要在类路径中添加 CGLib.jar 类库。
- order：如果业务类除事务切面外，还需要织入其他的切面，则通过该属性可以控制事务切面在目标连接点的织入顺序。

1．关于@Transactional 的属性

基于@Transactional 注解的配置和基于 XML 的配置方式一样，它拥有一组普适性很强的默认事务属性，往往可以直接使用这些默认的属性。

- 事务传播行为：PROPAGATION_REQUIRED。
- 事务隔离级别：ISOLATION_DEFAULT。
- 读写事务属性：读/写事务。
- 超时时间：依赖于底层的事务系统的默认值。
- 回滚设置：任何运行期异常引发回滚，任何检查型异常不会引发回滚。

因为这些默认设置在大多数情况下都是适用的，所以一般不需要手工设置事务注解的属性（见表 11-7）。当然，Spring 允许通过手工设定属性值覆盖默认值。

表 11-7　@Transactional注解属性说明

属　性　名	说　　明
propagation	事务传播行为，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Propagation 例如：@Transactional(propagation=Propagation.REQUIRES_NEW)
isolation	事务隔离级别，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Isolation 例如：@Transactional(isolation=Isolation.READ_COMMITTED)
readOnly	事务读写性，布尔型。例如：@Transactional(readOnly=true)
timeout	超时时间，int 型，以秒为单位。例如：@Transactional(timeout=10)
rollbackFor	一组异常类，遇到时进行回滚，类型为：Class<? extends Throwable>[]，默认值为{}。例如：@Transactional(rollbackFor={SQLException.class})。多个异常之间可用逗号分隔
rollbackForClassName	一组异常类名，遇到时进行回滚，类型为 String[]，默认值为{}。例如：@Transactional(noRollbackForClassName={"Exception"})
noRollbackFor	一组异常类，遇到时不回滚，类型为 Class<? extends Throwable>[]，默认值为{}
noRollbackForClassName	一组异常类名，遇到时不回滚，类型为 String[]，默认值为{}

2．在何处标注@Transactional 注解

@Transactional 注解可以被应用于接口定义和接口方法、类定义和类的 public 方法上。

但 Spring 建议在业务实现类上使用@Transactional 注解。当然也可以在业务接口上使用@Transactional 注解，但这样会留下一些容易被忽视的隐患。因为注解不能被继承，所以在业务接口中标注的@Transactional 注解不会被业务实现类继承。如果通过以下配置启用子类代理：

```
<tx :annotation-driven transaction-manager="txManager" proxy-target-class="true"/>
```

那么业务类不会添加事务增强，照样工作在非事务环境下。举一个具体的实例：如果使用子类代理，假设用户为 BbtForum 接口标注了@Transactional 注解，那么其实现类 BbtForumImpl 依旧不会启用事务机制。

因此，Spring 建议在具体业务类上使用@Transactional 注解。这样，不管<tx:annotation-driven>将 proxy-target-class 属性配置为 true 或 false，业务类都会启用事务机制。

3．在方法处使用注解

方法处的注解会覆盖类定义处的注解。如果有些方法需要使用特殊的事务属性，则可以在类注解的基础上提供方法注解，如下：

```
@Transactional ①                    类级注解，适用于
public class BbtForum {             类中所有 public 的方法
    @Transactional(readOnly=true) ②   提供额外的注解信息，它
    public Forum getForum(int forumId) {   将覆盖①处的类级注解
```

```
        return forumDao.getForum(forumId);
    }
...
}
```

② 处的方法注解提供了 readOnly 事务属性的设置，它将覆盖类级注解中默认的 readOnly=false 设置。

4. 使用不同的事务管理器

一般情况下，一个应用仅需使用一个事务管理器。如果希望在不同的地方使用不同的事务管理器，则可以通过如下方式实现：

```
@Service
public class MultiForumService {
   //①使用名为topic的事务管理器
    @Transactional("topic")
    public void addTopic(Topic topic) throws Exception {
       ...
    }

   //②使用名为forum的事务管理器
    @Transactional("forum")
    public void updateForum(Forum forum) {
      ...
    }
}
```

而 topic 和 forum 的事务管理器可以在 XML 中分别定义，如下：

```
<bean id="forumTxManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="forumDataSource">①
     <qualifier value="forum"/> ②
</bean>

<bean id="topicTxManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="topicDataSource">
    <qualifier value="topic"/>
</bean>
<tx:annotation-driven/>
```

可以使用不同的数据源

为事务管理器指定一个名字

在①处，为事务管理器指定了一个数据源，每个事务管理器都可以绑定一个独立的数据源。在②处，指定了一个可被@Transactional 注解引用的事务管理器标识。

在一两处使用带标识的@Transactional 注解也许是合适的，但是如果到处都使用，则显得比较啰唆。可以自定义一个绑定到特定事务管理器的注解，然后直接使用这个自定义的注解进行标识，如下：

```
package com.smart;
...
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("forum") //①绑定到forum的事务管理器中
public @interface ForumTransactional {
}
```

按相似的方法，还可以定义一个绑定到 topic 事务管理器的@TopicTransactional 注解。完成定义后，就可以用以下方式对原来的代码进行调整：

```
public class MultiForumService {
   //①使用名为topic的事务管理器
    @TopicTransactional
    public void addTopic(Topic topic) throws Exception {
       ...
    }

   //②使用名为forum的事务管理器
    @ForumTransactional
    public void updateForum(Forum forum) {
      ...
    }
}
```

11.6.2　通过 AspectJ LTW 引入事务切面

Spring 还提供了对 AspectJ 切面的支持。在 org.springframework.aspects-{version}.jar 中有一个用 AspectJ 语法编写的 AnnotationTransactionAspect 切面类，用于为使用了@Transactional 注解的业务类提供事务增强。

使用-javaagent: D:\masterSpring\spring\dist\org.springframework.aspects-{version}.jar 的 JVM 参数，在类路径 META-INF 目录下提供如下 AspectJ 配置文件：

```
<?xml version="1.0"?>
<aspectj>
    <aspects>
        <aspect  name="org.springframework.transaction.aspectj.
                 AnnotationTransactionAspect" />
      </aspects>
    <weaver
        options="-showWeaveInfo -XmessageHandlerClass:
      org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
        <include within="com.smart.service.impl.*" />
    </weaver>
</aspectj>
```

在类加载期，对标注@Transactional 注解的类织入 AnnotationTransactionAspect 事务增强切面。

由于 AnnotationTransactionAspect 切面类需要使用事务管理器，所以必须利用 Spring IoC 为切面类提供事务管理器的注入，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-4.0.xsd">
    <context:component-scan base-package="com.smart"/>
    <import resource="classpath:applicationContext-dao.xml" />
```

```
    <bean id=" transactionManager "①
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>

    <tx:annotation-driven/>
</beans>
```

如①处所示，通过 transactionManager 属性注入事务管理器 Bean，这样切面类就能为目标业务类提供事务管理功能。

11.7 集成特定的应用服务器

一般来说，Spring 的事务抽象与应用服务器是无关的。不过，如果用户希望事务管理器使用特定的 UserTransaction 和 TransactionManager 对象（可以被自动探测），以获取更多的事务控制功能（如事务挂起等），则可以采用 Spring 为集成这些应用服务器所提供的适配器。

11.7.1 BEA WebLogic

在一个使用 WebLogic 9.0 或更高版本的环境下，用户一般会优先选用特定于 WebLogic 的 WebLogicJtaTransactionManager 类取代基础的 JtaTransactionManager 类；因为在 WebLogic 环境中，该类提供了对 Spring 事务定义的完全支持，超过了标准的 JTA 语义。用户可以使用以下配置达到目的：

```
<bean id="txManager"
  class="org.springframework.transaction.jta.WebLogicJtaTransactionManager"/>
```

它的特性包括：支持事务名、支持为每个事务定义隔离级别，以及在任何环境下正确地恢复事务的能力。

11.7.2 WebSphere

在 WebSphere 6.1 及以上版本的环境下，用户可以使用 Spring 的 WebSphereUow TransactionManager 类。这个特殊的适配器支持 IBM 的 UOWManager 的 API，在该适配器中，Spring 所支持的事务挂起特性得到了 IBM 官方的支持。

```
<bean id="txManager"
   class="org.springframework.transaction.jta.WebSphereUowTransactionManager"/>
```

11.8 小结

Spring 声明式事务管理是 Spring 中的亮点，也是被使用最多的功能之一。Spring 使

声明式事务平民化，现在可以在 Spring 轻量级容器中享受这项曾经只能在臃肿、厚重的 EJB 应用服务器中才拥有的功能。

Spring 事务管理是 Spring AOP 技术的精彩应用的案例。事务管理作为一个切面织入目标业务方法的周围，业务方法完全从事务代码中解脱出来，代码的复杂度大大降低。被织入的事务代码基于 Spring 事务同步管理器进行工作，事务同步管理器维护着业务类对象线程相关的资源。DAO 类需要利用资源获取工具访问底层数据连接，或者直接建立在相应的持久化模板类的基础上。要了解它们的内部机制，就必须事先了解 ThreadLocal 的工作机制。

Spring 的事务配置相对来说比较简单，这些配置主要提供两方面的信息：其一，切点信息，用于定位实施事务切面的业务类方法；其二，控制事务行为的事务属性，这些属性包括事务隔离级别、事务传播行为、超时时间、回滚规则等。理解事务属性具体配置值的实际意义是非常关键的，因此 11.1 节对此专门进行了讲解。

Spring 通过 aop/tx Schema 命名空间和@Transactional 注解技术，大大简化了声明式事务配置的难度。同时，了解基于 TransactionProxyFactoryBean 代理类定义声明式事务的工作机制，有助于理解事务增强的内部过程。

第 12 章

Spring 的事务管理难点剖析

Spring 最成功、最吸引人的地方莫过于轻量级的声明式事务管理，仅此一点，Spring 就宣告了重量级 EJB 容器的覆灭。Spring 声明式事务管理将开发者从繁复的事务管理代码中解脱出来，专注于业务逻辑的开发上，这不啻为一件可以被拿来顶礼膜拜的事情。但是，世界并未从此太平，开发人员需要面对的是层出不穷的应用场景，这些场景往往逾越了普通 Spring 技术书籍的理想界定。因此，随着应用开发的深入，在使用经过 Spring 层层封装的声明式事务时，开发人员越来越觉得自己坠入了迷雾、陷入了沼泽，体会不到外界所宣称的那种畅快淋漓。本章的目标旨在整理并剖析实际应用中种种让我们迷茫的场景，让阳光照进云遮雾障的山头。

本章主要内容：

- DAO、应用分层和事务管理的关系
- 事务方法嵌套调用的分析
- 在多线程环境下事务方法的调用问题
- 使用多种数据访问技术的问题
- Spring AOP 事务增强有哪些限制
- 数据连接泄露的原因及规避

本章亮点：

- 以实战的角度，深入剖析事务管理的众多难点
- 解构 Spring 数据连接维护的内部原理
- 如何彻底告别数据连接泄露

12.1　DAO 和事务管理的牵绊

很少有使用 Spring 而不使用 Spring 事务管理器的应用，因此常常有人会问：是否用了 Spring，就一定要用 Spring 事务管理器，否则就无法进行数据的持久化操作呢？事务管理器和 DAO 是什么关系呢？

也许是 DAO 和事务管理如影随形的缘故吧，这个看似简单的问题实实在在地存在着，从初学者心中涌出，萦绕在老把式的脑际。答案当然是否定的。我们都知道，事务管理的目的是保证数据操作的事务性（原子性、一致性、隔离性、持久性，即所谓的 ACID），脱离了事务性，DAO 照样可以顺利地进行数据操作。

12.1.1　JDBC 访问数据库

下面来看一段使用 Spring JDBC 进行数据访问的代码，如代码清单 12-1 所示。

代码清单 12-1　UserJdbcWithoutTransManagerService

```
package com.smart.withouttx.jdbc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.apache.commons.dbcp.BasicDataSource;

@Service("userService")
public class UserJdbcWithoutTransManagerService {
    private JdbcTemplate jdbcTemplate;

   @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
         this.jdbcTemplate = jdbcTemplate;
    }

    public void addScore(String userName,int toAdd){
       String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
       jdbcTemplate.update(sql,toAdd,userName);
    }

    public static void main(String[] args) {
       ApplicationContext ctx = new
          ClassPathXmlApplicationContext("com/smart/withouttx/jdbc/jdbcWithoutTx.xml");
       UserJdbcWithoutTransManagerService service =
                  (UserJdbcWithoutTransManagerService)ctx.getBean("userService");
       JdbcTemplate jdbcTemplate = (JdbcTemplate)ctx.getBean("jdbcTemplate");
       BasicDataSource basicDataSource = (BasicDataSource)jdbcTemplate.getDataSource();
```

```
        //①检查数据源autoCommit的设置
        System.out.println("autoCommit:"+ basicDataSource.getDefaultAutoCommit());

        //②插入一条记录，初始分数为10
        jdbcTemplate.execute("INSERT INTO t_user(user_name,password,score,last_logon_time)
                             VALUES('tom','123456',10,"+System.currentTimeMillis()+")");

        //③调用工作在无事务环境下的服务类方法，将分数添加20分
        service.addScore("tom",20);

        //④查看此时用户的分数
        int score = jdbcTemplate.queryForInt(
                "SELECT score FROM t_user WHERE user_name ='tom'");
        System.out.println("score:"+score);
        jdbcTemplate.execute("DELETE FROM t_user WHERE user_name='tom'");
    }
}
```

其中，jdbcWithoutTx.xml 配置文件如代码清单 12-2 所示。

代码清单 12-2　jdbcWithoutTx.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"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
   http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.withouttx.jdbc"/>
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:driverClassName="${jdbc.driverClassName}"
          p:url="${jdbc.url}"
          p:username="${jdbc.username}"
          p:password="${jdbc.password}"/>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
         p:dataSource-ref="dataSource"/>
</beans>
```

运行 UserJdbcWithoutTransManagerService，在控制台上打印出如下结果：

```
defaultAutoCommit:true
score:30
```

在 jdbcWithoutTx.xml 中没有配置任何事务管理器，但是数据已经成功持久化到数据库中。在默认情况下，dataSource 数据源的 autoCommit 被设置为 true——这也意味着所有通过 JdbcTemplate 执行的语句马上提交，没有事务。如果将 dataSource 的 defaultAutoCommit 设置为 false，再次运行 UserJdbcWithoutTransManagerService，将抛出错误，原因是新增及更改数据的操作都没有提交到数据库，所以代码清单 12-1 中④处的语句因无法从数据库中查询到匹配的记录而引发异常。

对于强调读速度的应用，数据库本身可能就不支持事务，如使用 MyISAM 引擎的 MySQL 数据库。这时，无须在 Spring 应用中配置事务管理器，因为即使配置了，也是没有实际用处的。

12.1.2　Hibernate 访问数据库

对于 Hibernate 来说，情况就有点复杂了。因为 Hibernate 的事务管理拥有其自身的意义，它和 Hibernate 一级缓存存在密切的关系：在调用 Session 的 save、update 等方法时，Hibernate 并不直接向数据库发送 SQL 语句，只在提交事务（commit）或 flush 一级缓存时才真正向数据库发送 SQL。所以，即使底层数据库不支持事务，Hibernate 的事务管理也是有一定好处的，不会对数据操作的效率造成负面影响。所以，如果使用 Hibernate 数据访问技术，则没有理由不配置 HibernateTransactionManager 事务管理器。

但是，如果不使用 Hibernate 事务管理器，Spring 就会采取默认的事务管理策略搜索（PROPAGATION_REQUIRED, readOnly）。如果有修改操作是不允许的，就会抛出异常。来看下面的例子，如代码清单 12-3 所示。

代码清单 12-3　UserHibernateWithoutTransManagerService

```
package com.smart.withouttx.hiber;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.apache.commons.dbcp.BasicDataSource;
import org.springframework.test.jdbc.SimpleJdbcTestUtils;

import com.smart.User;

@Service("hiberService")
public class UserHibernateWithoutTransManagerService {
    private HibernateTemplate hibernateTemplate;

   @Autowired
    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
         this.hibernateTemplate = hibernateTemplate;
    }


    public void addScore(String userName,int toAdd){
       User user = hibernateTemplate.get(User.class,userName);

       //①以下两条语句取消注释后，由于默认事务管理器不支持数据更改，将报异常
```

```
        //user.setScore(user.getScore()+toAdd);
        //hibernateTemplate.update(user);
    }

    public static void main(String[] args) {
        ApplicationContext ctx = new
            ClassPathXmlApplicationContext("com/smart/withouttx/hiber/
hiberWithoutTx.xml");
        UserHibernateWithoutTransManagerService service =
            (UserHibernateWithoutTransManagerService)ctx.getBean("hiberService");

        JdbcTemplate jdbcTemplate = (JdbcTemplate)ctx.getBean("jdbcTemplate");
        BasicDataSource basicDataSource = (BasicDataSource)jdbcTemplate.
getDataSource();

        //②检查数据源autoCommit的设置
        System.out.println("autoCommit:"+ basicDataSource.getDefaultAutoCommit());

        //③插入一条记录，初始分数为10
        jdbcTemplate.execute("INSERT INTO t_user(user_name,password,score,last_
logon_time)
                    VALUES('tom','123456',10,"+System.currentTimeMillis()+")");

        //④调用工作在无事务环境下的服务类方法，将分数添加20分
        service.addScore("tom",20);

        //⑤查看此时用户的分数
        int score = jdbcTemplate.queryForInt(
                "SELECT score FROM t_user WHERE user_name ='tom'");
        System.out.println("score:"+score);
        jdbcTemplate.execute("DELETE FROM t_user WHERE user_name='tom'");
    }
}
```

此时，采用 hiberWithoutTx.xml 配置文件，其配置内容如代码清单 12-4 所示。

代码清单 12-4　hiberWithoutTx.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"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.withouttx.hiber"/>
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:driverClassName="${jdbc.driverClassName}"
          p:url="${jdbc.url}"
          p:username="${jdbc.username}"
          p:password="${jdbc.password}"/>
```

```
    <bean id="jdbcTemplate"
          class="org.springframework.jdbc.core.JdbcTemplate"
          p:dataSource-ref="dataSource"/>

    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
          p:dataSource-ref="dataSource">
        <property name="annotatedClasses">
            <list>
                <value>com.smart.User</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

    <bean id="hibernateTemplate"
          class="org.springframework.orm.hibernate4.HibernateTemplate"
          p:sessionFactory-ref="sessionFactory"/>
</beans>
```

com.smart.User 是使用了 Hibernate 注解的领域对象，如代码清单 12-5 所示。

代码清单 12-5　User

```
package com.smart;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import javax.persistence.Id;
import java.lang.reflect.Field;
import java.io.Serializable;

@Entity
@Table(name="T_USER")
public class User implements Serializable{
    @Id
    @Column(name = "USER_NAME")
    private String userName;

    private String password;

    private int score;

    @Column(name = "LAST_LOGON_TIME")
    private long lastLogonTime = 0;

    …
}
```

运行 UserHibernateWithoutTransManagerService，如果代码清单 12-3 的①处取消注

释，那么程序将报错。这说明 Hibernate 在 Spring 中，在没有事务管理器的情况下，只能读数据而无法更改数据。

12.2 应用分层的迷惑

Web、Service 及 DAO 三层是 Web 应用开发常见的模式，但有些开发人员却错误地认为，如果要使用 Spring 的事务管理就一定要先进行三层的划分，更有甚者认为每层一定要先定义一个接口，然后再定义一个实现类。对将“面向接口编程”奉为圭臬，认为放之四海而皆准的论调，笔者并不是很赞同。是的，“面向接口编程”是 Martin Fowler、Rod Johnson 这些大师所提倡的行事原则。如果拿这条原则去开发框架、产品或大型项目，怎么强调都不为过。但是，对于一般的开发人员来说，也许做的是一个普通工程项目，往往只是一些对数据库增、删、查、改的功能。此时，过分强制“面向接口编程”除了会带来更多的类文件，并不会有什么好处。

Spring 框架所提供的各种好处（如 AOP、注解增强、注解 MVC 等）的唯一前提就是让 POJO 的类变成一个受 Spring 容器管理的 Bean。下面的实例用一个 POJO 完成所有的功能，既是 Controller，又是 Service，还是 DAO，如代码清单 12-6 所示。

代码清单 12-6 MixLayerUserService

```
package com.smart.mixlayer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

//①将POJO类通过注解变成Spring MVC的Controller
@Controller
public class MixLayerUserService {

    //②自动注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    //③通过Spring MVC注解映射成为处理HTTP请求的函数，同时作为一个拥有事务性的方法
    @RequestMapping("/logon.do")
    @Transactional
    public String logon(String userName,String password){
        if(isRightUser(userName,password)){
            String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
            jdbcTemplate.update(sql,20,userName);
            return "success";
        }else{
            return "fail";
        }
    }
```

```
    private boolean isRightUser(String userName,String password){
        //do sth
        return true;
    }
}
```

通过@Controller 注解将 MixLayerUserService 变成 Web 层的 Controller，同时也是 Service 层的服务类。此外，由于直接使用 JdbcTemplate 访问数据，所以 MixLayerUserService 还是一个 DAO。来看一下对应的 Spring 配置文件，如代码清单 12-7 所示。

代码清单 12-7　applicationContext.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"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
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-4.0.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    …
    <!--①事务管理配置->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
          p:dataSource-ref="dataSource"/>
    <tx:annotation-driven/>


    <!--②启动Spring MVC的注解功能-->
    <bean class="org.springframework.web.servlet.mvc.annotation.
              AnnotationMethodHandlerAdapter"/>
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          p:prefix="/WEB-INF/jsp/" p:suffix=".jsp"/>
</beans>
```

在①处，通过事务注解驱动使 MixLayerUserService 的 logon()工作在事务环境下；在②处，配置了 Spring MVC 的一些基本设施。要使程序能够运行起来，还必须进行 web.xml 的相关配置，如代码清单 12-8 所示。

代码清单 12-8　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

```
    <servlet>
        <servlet-name>user</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:com/smart/mixlayer/applicationContext.xml
</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>user</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

这个配置文件很简单，唯一需要注意的是 DispatcherServlet 的配置。默认情况下，Spring MVC 根据 Servlet 的名字查找 WEB-INF 下的<servletName>-servlet.xml 作为 Spring MVC 的配置文件，在此，通过 contextConfigLocation 参数显式指定 Spring MVC 配置文件的确切位置。

将 org.springframework.jdbc 及 org.springframework.transaction 的日志级别设置为 DEBUG，启动项目，并访问 http://localhost/forum/logon.do?userName=tom 应用，MixLayerUserService#logon()方法将做出响应，查看后台输出日志，如下：

```
  Returning cached instance of singleton bean 'transactionManager'
  Creating new transaction with name [com.smart.mixlayer.MixLayerUserService.logon]:
   PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
    (DataSourceTransactionManager.java:204) - Acquired Connection [jdbc:mysql://
localhost:3306/sampledb, UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC
transaction
    (DataSourceTransactionManager.java:221) - Switching JDBC Connection
[jdbc:mysql://localhost:3306/sampledb, UserName=root@localhost, MySQL-AB JDBC Driver]
to manual commit
    (JdbcTemplate.java:810) - Executing prepared SQL update
    (JdbcTemplate.java:569) - Executing prepared SQL statement [UPDATE t_user u SET
u.score = u.score + ? WHERE user_name =?]
    (JdbcTemplate.java:819) - SQL update affected 0 rows
    (AbstractPlatformTransactionManager.java:752) - Initiating transaction commit
    (DataSourceTransactionManager.java:264) - Committing JDBC transaction on
Connection [jdbc:mysql://localhost:3306/sampledb, UserName=root@localhost, MySQL-AB
JDBC Driver]
    …
```

日志中的粗体部分说明 MixLayerUserService#logon()方法已经成功地运行在事务上下文中。

Spring 框架本身不应是代码复杂化的理由，使用 Spring 的开发者应该是无拘无束的：从实际应用出发，根据实际需要进行编程。

12.3　事务方法嵌套调用的迷茫

12.3.1　Spring 事务传播机制回顾

Spring 事务的一个被讹传很广的说法是：一个事务方法不应该调用另一个事务方法，否则将产生两个事务。结果造成开发人员在设计事务方法时束手束脚，生怕一不小心就踩到地雷。

其实这是未正确认识 Spring 事务传播机制而造成的误解。Spring 对事务控制的支持统一在 TransactionDefinition 类中描述，该类有以下几个重要的接口方法。

- int getPropagationBehavior()：事务的传播行为。
- int getIsolationLevel()：事务的隔离级别。
- int getTimeout()：事务的过期时间。
- boolean isReadOnly()：事务的读/写特性。

很明显，除了事务的传播行为，对于事务的其他特性，Spring 是借助底层资源的功能来完成的，Spring 无非充当了一个代理的角色。但是事务的传播行为却是 Spring 凭借自身的框架提供的功能，是 Spring 提供给开发者最珍贵的礼物，讹传的说法玷污了 Spring 事务框架最美丽的光环。

所谓事务传播行为，就是多个事务方法相互调用时，事务如何在这些方法间传播。Spring 支持以下 7 种事务传播行为。

- PROPAGATION_REQUIRED：如果当前没有事务，就新建一个事务；如果已经存在一个事务，就加入到这个事务中。这是最常见的选择。
- PROPAGATION_SUPPORTS：支持当前事务。如果当前没有事务，就以非事务方式执行。
- PROPAGATION_MANDATORY：使用当前事务。如果当前没有事务，就抛出异常。
- PROPAGATION_REQUIRES_NEW：新建事务。如果当前存在事务，就把当前事务挂起。
- PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作。如果当前存在事务，就把当前事务挂起。
- PROPAGATION_NEVER：以非事务方式执行。如果当前存在事务，就抛出异常。
- PROPAGATION_NESTED：如果当前存在事务，就在嵌套事务内执行；如果当前没有事务，就执行与 PROPAGATION_REQUIRED 类似的操作。

Spring 默认的事务传播行为是 PROPAGATION_REQUIRED，它适合绝大多数情况。如果多个 ServiveX#methodX()均工作在事务环境下（均被 Spring 事务增强），且程序中存在调用链 Service1#method1()->Service2#method2()->Service3# method3()，那么这 3 个服务类的 3 个方法通过 Spring 的事务传播机制都工作在同一个事务中。

12.3.2 相互嵌套的服务方法

来看一下实例，UserService#logon()方法内部调用了 UserService#updateLastLogonTime()和 ScoreService#addScore()方法，这两个类都继承于 BaseService。它们之间的类结构如图 12-1 所示。

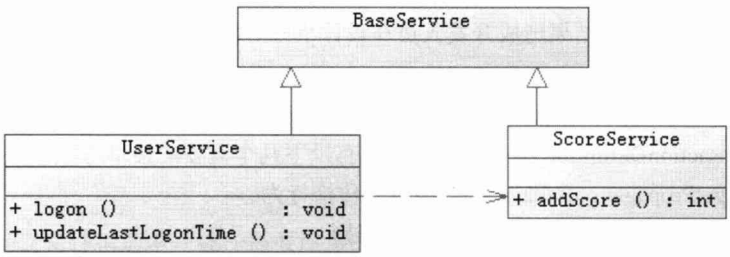


图 12-1 两个相互调用的服务类

UserService#logon()方法内部调用了 ScoreService#addScore()方法，二者分别通过 Spring AOP 进行了事务增强，则它们工作在同一事务中。来看具体的代码，如代码清单 12-9 所示。

代码清单 12-9 UserService

```
package com.smart.nestcall;
…
@Service("userService")
public class UserService extends BaseService {
    private JdbcTemplate jdbcTemplate;
    private ScoreService scoreService;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Autowired
    public void setScoreService(ScoreService scoreService) {
        this.scoreService = scoreService;
    }

    //①该方法嵌套调用了本类的其他方法及其他服务类的方法
    public void logon(String userName) {
        System.out.println("before userService.updateLastLogonTime...");
        updateLastLogonTime(userName);//①-1本服务类的其他方法
        System.out.println("after userService.updateLastLogonTime...");

        System.out.println("before scoreService.addScore...");
        scoreService.addScore(userName, 20); //①-2其他服务类的方法
        System.out.println("after scoreService.addScore...");

    }
    public void updateLastLogonTime(String userName) {
```

```
        String sql = "UPDATE t_user u SET u.last_logon_time = ? WHERE user_name =?";
        jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
    }
```

在 UserService 中注入了 ScoreService 的 Bean，而 ScoreService 的代码如代码清单 12-10 所示。

代码清单 12-10　ScoreService

```
package com.smart.nestcall;
…
@Service("scoreService")
public class ScoreService extends BaseService{

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void addScore(String userName, int toAdd) {
        String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
        jdbcTemplate.update(sql, toAdd, userName);
    }
}
```

通过 Spring 配置为 ScoreService 及 UserService 中的所有公有方法都添加 Spring AOP 的事务增强，让 UserService 的 logon()和 updateLastLogonTime()方法及 ScoreService 的 addScore()方法都工作在事务环境下。下面是关键的配置代码，如代码清单 12-11 所示。

代码清单 12-11　nestcall\applicationContext.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"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
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-4.0.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    <context:component-scan base-package="com.smart.nestcall"/>
     …
    <bean id="jdbcManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
          p:dataSource-ref="dataSource"/>

    <!--①通过以下配置为所有继承BaseService类的所有子类的所有public方法都添加事务增强-->
    <aop:config proxy-target-class="true">
```

```
        <aop:pointcut id="serviceJdbcMethod"
                      expression="within(com.smart.nestcall.BaseService+)"/>
        <aop:advisor pointcut-ref="serviceJdbcMethod" advice-ref="jdbcAdvice"
order="0"/>
    </aop:config>
    <tx:advice id="jdbcAdvice" transaction-manager="jdbcManager">
        <tx:attributes>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>
</beans>
```

将日志级别设置为 DEBUG，启动 Spring 容器并执行 UserService#logon()方法，仔细观察如下输出日志：

```
    before userService.logon method...

    //①创建了一个事务
    Creating new transaction with name [com.smart.nestcall.UserService.logon]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
    Acquired Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
    Switching JDBC Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver] to manual commit
    before userService.updateLastLogonTime...

    <!--②updateLastLogonTime()和logon()在同一个Bean中，并未发生加入已存在事务上下文的
     动作，而是“天然”地工作在相同的事务上下文中-->
    Executing prepared SQL update
    Executing prepared SQL statement [UPDATE t_user u SET u.last_logon_time = ? WHERE
user_name =?]
    SQL update affected 1 rows
    after userService.updateLastLogonTime...
    before scoreService.addScore...

    //③ScoreService#addScore()方法加入到①处启动的事务上下文中
    Participating in existing transaction
    Executing prepared SQL update
    Executing prepared SQL statement [UPDATE t_user u SET u.score = u.score + ? WHERE
user_name =?]
    SQL update affected 1 rows
    after scoreService.addScore...
    Initiating transaction commit
    Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver]
    …
    after userService.logon method...
```

从上面的输出日志中可以清楚地看到，Spring 为 UserService#logon()方法启动了一个新的事务，而 UserSerive#updateLastLogonTime()和 UserService#logon()方法在相同的类中，没有观察到有事务传播行为的发生，其代码块好像“直接合并”到 UserService#logon()方法中。

然而在执行到 ScoreService#addScore()方法时，我们观察到发生了一个事务传播行为：“Participating in existing transaction”；这说明 ScoreService#addScore()方法添加到了

UserService#logon()方法的事务上下文中，二者共享同一个事务。所以最终的结果是 UserService 的 logon()和 updateLastLogonTime()方法及 ScoreService 的 addScore()方法工作在同一个事务中。

12.4　多线程的困惑

12.4.1　Spring 通过单实例化 Bean 简化多线程问题

由于 Spring 的事务管理器是通过线程相关的 ThreadLocal 来保存数据访问基础设施（Connection 实例）的，再结合 IoC 和 AOP 实现高级声明式事务的功能，所以 Spring 的事务天然地和线程有着千丝万缕的联系。

我们知道 Web 容器本身就是多线程的，Web 容器为一个 HTTP 请求创建一个独立的线程（实际上大多数 Web 容器采用共享线程池），所以由此请求所涉及的 Spring 容器中的 Bean 也运行在多线程环境下。在绝大多数情况下，Spring 的 Bean 都是单实例的（singleton），单实例 Bean 的最大好处是线程无关性，不存在多线程并发访问的问题，也就是线程安全的。

一个类能够以单实例的方式运行的前提是“无状态”，即一个类不能拥有状态化的成员变量。我们知道，在传统的编程中，DAO 必须持有一个 Connection，而 Connection 就是状态化的对象。所以传统的 DAO 不能做成单实例的，每次要用时都必须创建一个新的实例。传统的 Service 由于内部包含了若干有状态的 DAO 成员变量，所以其本身也是有状态的。

但在 Spring 中，DAO 和 Service 都以单实例的方式存在。Spring 通过 ThreadLocal 将有状态的变量（如 Connection 等）本地线程化，达到另一个层面上的“线程无关”，从而实现线程安全。Spring 不遗余力地将有状态的对象无状态化，就是要达到单实例化 Bean 的目的。

由于 Spring 已经通过 ThreadLocal 的设施将 Bean 无状态化，所以 Spring 中的单实例 Bean 对线程安全问题拥有了一种天生的免疫能力。不但单实例的 Service 可以成功运行在多线程环境中，Service 本身还可以自由地启动独立线程以执行其他的 Service。

12.4.2　启动独立线程调用事务方法

下面对 logon()方法进行改造，让其在方法内部再启动一个新线程，在这个新线程中执行积分添加的操作，看看究竟会发生哪些事务行为，如代码清单 12-12 所示。

代码清单 12-12 UserService

```
package com.smart.multithread;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.stereotype.Service;
import org.apache.commons.dbcp.BasicDataSource;

@Service("userService")
public class UserService extends BaseService {
    private JdbcTemplate jdbcTemplate;
    private ScoreService scoreService;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Autowired
    public void setScoreService(ScoreService scoreService) {
        this.scoreService = scoreService;
    }

    public void logon(String userName) {
        System.out.println("before userService.updateLastLogonTime method...");
        updateLastLogonTime(userName);
        System.out.println("after userService.updateLastLogonTime method...");

        //①在同一个线程中调用scoreService#addScore(), 将运行在同一个事务中
        //scoreService.addScore(userName, 20);

        //②在一个新线程中执行scoreService#addScore(), 将启动一个新的事务
        Thread myThread = new MyThread(this.scoreService, userName, 20);
        myThread.start();
    }

    public void updateLastLogonTime(String userName) {
        String sql = "UPDATE t_user u SET u.last_logon_time = ? WHERE user_name =?";
        jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
    }

    //③负责执行scoreService#addScore()的线程类
    private class MyThread extends Thread {
        private ScoreService scoreService;
        private String userName;
        private int toAdd;
        private MyThread(ScoreService scoreService, String userName, int toAdd) {
            this.scoreService = scoreService;
            this.userName = userName;
            this.toAdd = toAdd;
        }
```

```
        public void run() {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("before scoreService.addScor method...");
            scoreService.addScore(userName, toAdd);
            System.out.println("after scoreService.addScor method...");
        }
    }
}
```

将日志级别设置为 DEBUG，执行 UserService#logon()方法，观察以下输出日志：

```
    before userService.logon method...

    //①创建一个事务
    Creating new transaction with name [com.smart.multithread.UserService.logon]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
    Acquired Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
    ...
    SQL update affected 1 rows
    after userService.updateLastLogonTime method...
    Initiating transaction commit

    //②提交①处开启的事务
    Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver]
    Releasing JDBC Connection [jdbc:mysql://localhost:3306/sampledb, UserName=
root@localhost, MySQL-AB JDBC Driver] after transaction
    ...
    Returning JDBC Connection to DataSource
    before scoreService.addScor method...

   //③创建一个事务
    Creating new transaction with name [com.smart.multithread.ScoreService.addScore]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
    Acquired Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver] for JDBC transaction
    ...
    SQL update affected 0 rows
    Initiating transaction commit

    //④提交③处开启的事务
    Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver]
    Releasing JDBC Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver] after transaction...
```

在①处，在主线程（main）中执行的 UserService#logon()方法的事务启动；在②处，其对应的事务提交。而在子线程（Thread-2）中执行的 ScoreService#addScore()方法的事务在③处启动；在④处提交其对应的事务。

所以可以得出这样的结论：在相同线程中进行相互嵌套调用的事务方法工作在相同的事务中。如果这些相互嵌套调用的方法工作在不同的线程中，则不同线程下的事务方法工作在独立的事务中。

12.5 联合军种作战的混乱

12.5.1 Spring 事务管理器的应对

Spring 抽象的 DAO 体系兼容多种数据访问技术，它们各有特色、各有千秋。如 Hibernate 是一个非常优秀的 ORM 实现方案，但对底层 SQL 的控制不太方便；而 MyBatis 则通过模板化技术让用户方便地控制 SQL，但没有 Hibernate 那样高的开发效率；自由度最高的当然是直接使用 Spring JDBC 了，但它也是底层的，灵活的代价是代码的繁复。很难说哪种数据访问技术是最优秀的，只有在某种特定的场景下才能给出答案。所以在一个应用中往往采用多种数据访问技术，一般是两种，一种采用 ORM 技术框架，而另一种采用偏 JDBC 的底层技术；二者珠联璧合，形成联合军种，共同御敌。

但是，这种联合军种如何应对事务管理的问题呢？我们知道，Spring 为每种数据访问技术提供了相应的事务管理器，难道需要分别为它们配置对应的事务管理器吗？它们到底是如何协同工作的呢？这些层出不穷的问题往往压制了开发人员使用联合军种的想法。

其实，在这个问题上，我们低估了 Spring 事务管理的能力。如果用户采用了一种高端的 ORM 技术（Hibernate、JPA、JDO），同时还采用了一种 JDBC 技术（Spring JDBC、MyBatis）；由于前者的会话（Session）是对后者连接（Connection）的封装，Spring 会“足够智能地”在同一个事务线程中让前者的会话封装后者的连接。所以，只要直接采用前者的事务管理器就可以了。表 12-1 给出了混合数据访问技术框架所对应的事务管理器。

表 12-1 事务隔离级别对并发问题的解决情况

序　号	混合数据访问技术框架	事务管理器
1	Hibernate+ Spring JDBC 或 MyBatis	org.springframework.orm.hibernate*X*.HibernateTransactionManager
2	JPA+Spring JDBC 或 MyBatis	org.springframework.orm.jpa.JpaTransactionManager
3	JDO+Spring JDBC 或 MyBatis	org.springframework.orm.jdo.JdoTransactionManager

12.5.2 Hibernate+Spring JDBC 混合框架的事务管理

由于一般不会出现同时使用多个 ORM 框架的情况（如 Hibernate+JPA），所以我们不拟对此命题展开论述，只重点研究 ORM 框架+JDBC 框架的情况。Hibernate+Spring

JDBC 可能是被使用得最多的组合，本节通过实例观察事务管理的运作情况，如代码清单 12-13 所示。

代码清单 12-13　UserService

```
package com.smart.mixdao;

…
@Service("userService")
public class UserService extends BaseService {
    private HibernateTemplate hibernateTemplate;
    private ScoreService scoreService;

    @Autowired
    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    @Autowired
    public void setScoreService(ScoreService scoreService) {
        this.scoreService = scoreService;
    }

    public void logon(String userName) {

        //①通过Hibernate技术访问数据
        System.out.println("before updateLastLogonTime()..");
        updateLastLogonTime(userName);
        System.out.println("end updateLastLogonTime()..");

        //②通过JDBC技术访问数据
        System.out.println("before scoreService.addScore()..");
        scoreService.addScore(userName, 20);
        System.out.println("end scoreService.addScore()..");
    }

    public void updateLastLogonTime(String userName) {
        User user = hibernateTemplate.get(User.class,userName);
        user.setLastLogonTime(System.currentTimeMillis());
        hibernateTemplate.update(user);

        //③这句很重要，请看下文的分析
        hibernateTemplate.flush();
    }

}
```

在①处使用 Hibernate 技术操作数据，而在②处调用 ScoreService#addScore()，该方法内部使用 Spring JDBC 技术操作数据。

在③处显式调用了 flush()方法，将 Session 中的缓存同步到数据库中（马上向数据库发送一条更新记录的 SQL 语句）。之所以要显式执行 flush()方法，是因为在默认情况下，Hibernate 对数据的更改只记录在一级缓存中，要等到事务提交或显式调用 flush()方法时才会将一级缓存中的数据同步到数据库中，而提交事务的操作发生在 logon()方法

返回前。如果所有针对数据库的更改操作都使用 Hibernate，那么这种数据同步的延迟机制并不会产生任何问题。但是，我们在 logon()方法中同时采用了 Hibernate 和 Spring JDBC 数据访问技术，而 Spring JDBC 无法自动感知 Hibernate 一级缓存；所以如果不及时调用 flush()方法将记录数据更改的一级缓存同步到数据库中，则②处通过 Spring JDBC 进行数据更改的结果将被 Hibernate 一级缓存中的更改覆盖掉，因为 Hibernate 一级缓存要等到 logon()方法返回前才同步到数据库中。

ScoreService 使用了 Spring JDBC 数据访问技术，如代码清单 12-14 所示。

代码清单 12-14　UserService

```
package com.smart.mixdao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.apache.commons.dbcp.BasicDataSource;

@Service("scoreService")
public class ScoreService extends BaseService{
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

   public void addScore(String userName, int toAdd) {
       String sql = "UPDATE t_user u SET u.score = u.score + ? WHERE user_name =?";
       jdbcTemplate.update(sql, toAdd, userName);
       BasicDataSource basicDataSource = (BasicDataSource)
jdbcTemplate.getDataSource();

       //①查看此处数据库激活的连接数量
       System.out.println("[scoreService.addScore]激活连接数量："
                   +basicDataSource.getNumActive());
   }
}
```

Spring 关键的配置文件代码如代码清单 12-15 所示。

代码清单 12-15　mixdao\applicationContext.xml

```
…
<!--①使用Hibernate事务管理器 -->
<bean id="hiberManager"
     class="org.springframework.orm.hibernate4.HibernateTransactionManager"
     p:sessionFactory-ref="sessionFactory"/>

<!--②使UserService及ScoreService的共用方法都拥有事务 -->
<aop:config proxy-target-class="true">
    <aop:pointcut id="serviceJdbcMethod"
                expression="within(com.smart.mixdao.BaseService+)"/>
    <aop:advisor pointcut-ref="serviceJdbcMethod"
                advice-ref="hiberAdvice"/>
```

```
    </aop:config>
    <tx:advice id="hiberAdvice" transaction-manager="hiberManager">
        <tx:attributes>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

</beans>
```

启动 Spring 容器，执行 UserService#logon()方法，可以看到如下执行日志：

```
    before userService.logon()..

    ①在执行userService.logon()方法后，Spring开启一个事务
    Creating new transaction with name [com.smart.mixdao.UserService.logon]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
    opened session at timestamp: 13009379637
    Opened new Session [org.hibernate.impl.SessionImpl@c5f468] for Hibernate
transaction
    ...
    Exposing Hibernate transaction as JDBC transaction
[jdbc:mysql://localhost:3306/sampledb, UserName=root@localhost, MySQL-AB JDBC Driver]
    before userService.updateLastLogonTime()..

    ②userService.updateLastLogonTime()方法执行时自动绑定到①处开启的Session中
    Found thread-bound Session for HibernateTemplate
    loading entity: [com.smart.User#tom]
    ...
    Not closing pre-bound Hibernate Session after HibernateTemplate
    end updateLastLogonTime()..

    before scoreService.addScore()..

    ③scoreService.addScore()方法执行时绑定到①处开启的Session中，并加入其所对应的事务中
    Found thread-bound Session [org.hibernate.impl.SessionImpl@c5f468] for Hibernate
     transaction
    Participating in existing transaction
    ...
    SQL update affected 1 rows

    ④此时数据源只打开了一个连接
    [scoreService.addScore]激活连接数量：1
    end scoreService.addScore()..
    Initiating transaction commit

    ⑤提交Hibernate的事务，它将触发一级缓存到数据库的同步
    Committing Hibernate transaction on Session [org.hibernate.impl.SessionImpl
@c5f468]
    commit
    ...
    com.smart.User{lastLogonTime=1300937963882,score=10,userName=tom,password=
123456}
    re-enabling autocommit

    ⑥提交Session底层所绑定的JDBC Connection所对应的事务
    committed JDBC Connection
```

```
    transaction completed on session with on_close connection release mode; be sure
to close the session to release JDBC resources!
    ...
    transaction completed on session with on_close connection release mode; be sure
to close the session to release JDBC resources!
    after userService.logon()..
```

仔细观察这段输出日志，在①处 UserService#logon()方法开启一个新的事务。②处的 UserService#updateLastLogonTime()方法绑定到事务上下文的 Session 中。③处的 ScoreService#addScore()方法加入到①处开启的事务上下文中。④处的输出是 ScoreService#addScore()方法内部的输出信息，汇报此时数据源激活的连接数为 1，这清楚地告诉我们，Hibernate 和 JDBC 这两种数据访问技术在同一事务上下文中“共用”一个连接。在⑤处提交 Hibernate 事务，接着在⑥处触发调用底层的 Connection 提交事务。

从以上运行结果可以得出这样的结论：使用 Hibernate 事务管理器后，可以混合使用 Hibernate 和 Spring JDBC 数据访问技术，它们将工作在同一事务上下文中。但是在使用 Spring JDBC 访问数据时，Hibernate 的一级或二级缓存得不到同步。此外，一级缓存延迟数据同步机制可能会覆盖 Spring JDBC 数据更改的结果。

由于混合数据访问技术方案存在“事务同步而缓存不同步”的情况，所以最好用 Hibernate 进行读/写操作，而只用 Spring JDBC 进行读操作。如用 Spring JDBC 进行简要列表的查询，而用 Hibernate 对查询出来的数据进行维护。

如果确实要同时使用 Hibernate 和 Spring JDBC 读/写数据，则必须充分考虑到 Hibernate 缓存机制引发的问题：必须整体分析数据维护逻辑，根据需要及时调用 Hibernate 的 flush()方法，以免覆盖 Spring JDBC 的更改，在 Spring JDBC 更改数据库时，维护 Hibernate 的缓存。由于方法调用顺序的不同可能会影响数据的同步性，因而很容易发生问题，这会极大地提高数据访问程序的复杂性。所以笔者郑重建议不要同时使用 Spring JDBC 和 Hibernate 对数据进行写操作。

可以将以上结论推广到其他混合数据访问技术的方案中，如 Hibernate+MyBatis、JPA+Spring JDBC、JDO+Spring JDBC 等。

12.6 特殊方法成漏网之鱼

12.6.1 哪些方法不能实施 Spring AOP 事务

由于 Spring 事务管理是基于接口代理或动态字节码技术，通过 AOP 实施事务增强的，虽然 Spring 依然支持 AspectJ LTW 在类加载期实施增强，但这种方法很少使用，所以我们不予关注。

对于基于接口动态代理的 AOP 事务增强来说，由于接口的方法都必须是 public 的，这就要求实现类的实现方法也必须是 public 的（不能是 protected、private 等），同时不

能使用 static 修饰符。所以，可以实施接口动态代理的方法只能是使用 public 或 public final 修饰符的方法，其他方法不可能被动态代理，相应地也就不能实施 AOP 增强。换句话说，即不能进行 Spring 事务增强。

基于 *CGLib* 字节码动态代理的方案是通过扩展被增强类，动态创建其子类的方式进行 AOP 增强植入的。由于使用 final、static、private 修饰符的方法都不能被子类覆盖，相应地这些方法将无法实施 AOP 增强。所以方法签名必须特别注意这些修饰符的使用，以免使方法不小心成为事务管理的“漏网之鱼”。

12.6.2　事务增强遗漏实例

本节通过具体的实例说明基于 CGLib 字节码动态代理无法享受 Spring AOP 事务增强的特殊方法，如代码清单 12-16 所示。

代码清单 12-16　UserService

```
package com.smart.special;
import org.springframework.stereotype.Service;
@Service("userService")
public class UserService {
    private void method1() {//① private方法因访问权限的限制，无法被子类覆盖
        System.out.println("method1");
    }
    public final void method2() {//② final方法无法被子类覆盖
        System.out.println("method2");
    }
    public static void method3() {//③ static是类级别的方法，无法被子类覆盖
        System.out.println("method3");
    }
    public void method4() { //④ public方法可以被子类覆盖，因此可以被动态字节码增强
        System.out.println("method4");
    }
    public final void method5() {//⑤final方法不能被子类覆盖
        System.out.println("in method5");
    }
    protected void method6(){//⑥protected方法可以被子类覆盖，因此可以被动态字节码增强
        System.out.println("in method6");
    }
}
```

Spring 通过 CGLib 动态代理技术对 UserService Bean 实施 AOP 事务增强的关键配置具体如代码清单 12-17 所示。

代码清单 12-17　applicationContext.xml

```
...
    <aop:config proxy-target-class="true"><!-- ①显式使用CGLib动态代理 -->

        <!-- ②希望对UserService的所有方法实施事务增强 -->
        <aop:pointcut id="serviceJdbcMethod"
                      expression="execution(* com.smart.special.UserService.*(..))"/>
```

```
        <aop:advisor pointcut-ref="serviceJdbcMethod" advice-ref="jdbcAdvice"
order="0"/>
    </aop:config>
    <tx:advice id="jdbcAdvice" transaction-manager="jdbcManager">
        <tx:attributes>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>
…
```

在①处通过 proxy-target-class="true"显式使用 CGLib 动态代理技术；在②处通过 AspectJ 切点表达式表达 UserService 的所有方法，希望对 UserService 的所有方法都实施 Spring AOP 事务增强。

在 UserService 中添加一个可执行的方法，如代码清单 12-18 所示。

代码清单 12-18 UserService

```
package com.smart.special;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("userService")
public class UserService {
    …
    public static void main(String[] args) {
        ApplicationContext ctx =
new ClassPathXmlApplicationContext("user/special/applicationContext.xml");
        UserService service = (UserService) ctx.getBean("userService");

        System.out.println("before method1");
        service.method1();
        System.out.println("after method1");

        System.out.println("before method2");
        service.method2();
        System.out.println("after method2");

        …

        System.out.println("before method5");
        service.method5();
        System.out.println("after method5");

        System.out.println("before method6");
        service.method6();
        System.out.println("after method6");
    }
}
```

在运行 UserService 之前，将 Log4J 日志级别设置为 DEBUG。运行以上代码查看输出日志，如下：

```
①未启用事务
 before method1
```

```
    in method1
    after method1

    ②未启用事务
    before method2
    in method2
    after method2

    ③未启用事务
    before method3
    in method3
    after method3

    ④启用事务
    before method4
    Creating new transaction with name [com.smart.special.UserService.method4]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
    ...
    in method4
    Initiating transaction commit
    Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver]
    Releasing JDBC Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver] after transaction
    Returning JDBC Connection to DataSource
    after method4

    ⑤未启用事务
    before method5
    in method5
    after method5

    ⑥启用事务
    before method6
    Creating new transaction with name [com.smart.special.UserService.method6]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
    ...
    in method6
    Initiating transaction commit
    Committing JDBC transaction on Connection [jdbc:mysql://localhost:3306/sampledb,
UserName=root@localhost, MySQL-AB JDBC Driver]
    ...
    after method6
```

观察以上输出日志，很容易发现 method1 至 method3 及 method5 这 4 个方法都没有被实施 Spring 的事务增强，而 method4 和 method6 被实施了事务增强。该结果验证了我们之前的论述。

表 12-2 描述了哪些特殊方法将成为 Spring AOP 事务增强的“漏网之鱼”。

表 12-2　不能被Spring AOP事务增强的方法

序　号	动态代理策略	不能被事务增强的方法
1	基于接口的动态代理	除 public 外的其他所有方法。此外，public static 也不能被增强
2	基于 CGLib 的动态代理	private、static、final 方法

不过，需要特别指出的是，这些不能被 Spring 事务增强的特殊方法并非就不工作在事务环境下。只要它们被外层的事务方法调用了，由于 Spring 事务管理的传播级别，内部方法也可以工作在外部方法所启动的事务上下文中。我们说这些方法不能被 Spring 进行 AOP 事务增强，是指这些方法不能启动事务，但是外层方法的事务上下文依旧可以顺利地传播到这些方法中。

这些不能被 Spring 事务增强的方法和可被 Spring 事务增强的方法的唯一区别在于“是否可以主动启动一个新事务”：前者不能而后者可以。对于事务传播行为来说，二者是完全相同的，前者也和后者一样不会造成数据连接的泄露问题。换句话说，如果这些“特殊方法”被无事务上下文的方法调用，则它们就工作在无事务上下文中；反之，如果被具有事务上下文的方法调用，则它们就工作在事务上下文中。

对于 private 方法，由于最终都会被 public 方法封装后再开放给外部调用，而 public 方法是可以被事务增强的，所以基本上没有什么问题。在实际开发中，最容易造成隐患的是基于 CGLib 动态代理的 public static 和 public final 这两种特殊方法。原因是它们本身是 public 的，因此可以直接被外部类（如 Web 层的 Controller 类）调用，只要调用者没有事务上下文，这些特殊方法也就以无事务的方式运作。

12.7 数据连接泄露

12.7.1 底层连接资源的访问问题

对于应用开发者来说，数据连接泄露无疑是一个可怕的梦魇。只要你开发的应用存在数据连接泄露的问题，应用系统最终都将因数据连接资源耗尽而崩溃，甚至还可能引起数据库的崩溃。数据连接泄露像一个黑洞那样让开发者避之唯恐不及。

Spring DAO 对所有支持的数据访问技术框架都使用模板化技术进行了薄层封装。**只要你的程序都使用 Spring DAO 的模板（如 JdbcTemplate、HibernateTemplate 等）进行数据访问，就一定不会存在数据连接泄露的问题**——这是 Spring 给予我们的郑重承诺！如果使用 Spring DAO 模板进行数据操作，则无须关注数据连接（Connection）及其衍生品（如 Hibernate 的 Session 等）的获取和释放操作，模板类已经通过其内部流程替我们完成了，且对开发者是透明的。

但是由于集成第三方产品、整合遗留代码等原因，可能需要直接访问数据源或直接获取数据连接及其衍生品。这时，如果使用不当，就可能在无意中创造出一个魔鬼般的连接泄露问题。

我们知道，当 Spring 事务方法运行时，就产生一个事务上下文，该上下文在本事务执行线程中针对同一个数据源绑定了一个唯一的数据连接（或其衍生品），所有被该事务上下文传播的方法都共享这个数据连接。这个数据连接从数据源获取到返回给数据源

都在 Spring 的掌控之中，不会发生问题。如果在需要数据连接时能够获取这个被 Spring 管控的数据连接，则使用者可以放心使用，无须关注连接释放的问题。

那么，如何获取这些被 Spring 管控的数据连接呢？Spring 提供了两种方法：其一是使用数据资源获取工具类；其二是对数据源（或其衍生品，如 Hibernate 的 SessionFactory）进行代理。

12.7.2　Spring JDBC 数据连接泄露

如果从数据源直接获取连接，且在使用完成后不主动归还给数据源（调用 Connection#close()方法），则将造成数据连接泄露的问题，如代码清单 12-19 所示。

代码清单 12-19　JdbcUserService

```
package com.smart.connleak;
…
@Service("jdbcUserService")
public class JdbcUserService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Transactional
    public void logon(String userName) {
        try {
            //①直接从数据源获取连接，后续程序没有显式释放该连接
            Connection conn = jdbcTemplate.getDataSource().getConnection();
            String sql = "UPDATE t_user SET last_logon_time=? WHERE user_name =?";
            jdbcTemplate.update(sql, System.currentTimeMillis(), userName);

            //②模拟程序代码的执行时间
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

JdbcUserService 通过 Spring AOP 事务增强的配置，让所有 public 方法都工作在事务环境中，即让 logon()和 updateLastLogonTime()方法拥有事务功能。在 logon()方法内部，在①处通过调用 jdbcTemplate.getDataSource().getConnection()方法显式获取一个连接，这个连接不是 logon()方法事务上下文线程绑定的连接，所以如果开发者没有手工释放这个连接（显式调用 Connection#close()方法），则这个连接将永久被占用（处于 active 状态），造成连接泄露。下面编写模拟运行的代码，查看方法执行对数据连接的实际占用情况，如代码清单 12-20 所示。

代码清单 12-20　JdbcUserService

```
package com.smart.connleak;
…
@Service("jdbcUserService")
public class JdbcUserService {
    …
    //①以异步线程的方式执行JdbcUserService#logon()方法，以模拟多线程的环境
    public static void asynchrLogon(JdbcUserService userService, String userName) {
        UserServiceRunner runner = new UserServiceRunner(userService, userName);
        runner.start();
    }
    private static class UserServiceRunner extends Thread {
        private JdbcUserService userService;
        private String userName;
        public UserServiceRunner(JdbcUserService userService, String userName) {
            this.userService = userService;
            this.userName = userName;
        }
        public void run() {
            userService.logon(userName);
        }
    }

    //②让主执行线程睡眠一段指定的时间
    public static void sleep(long time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    //③汇报数据源的连接占用情况
    public static void reportConn(BasicDataSource basicDataSource) {
        System.out.println("连接数[active:idle]-[" +
                basicDataSource.getNumActive()+":"+basicDataSource.getNumIdle()+"]");
    }

    public static void main(String[] args) {
        ApplicationContext ctx =
         new ClassPathXmlApplicationContext("com/smart/connleak/applicatonContext.xml");
        JdbcUserService userService = (JdbcUserService) ctx.getBean("jdbcUserService");

        BasicDataSource basicDataSource = (BasicDataSource) ctx.getBean("dataSource");

        //④汇报数据源初始连接占用情况
        JdbcUserService.reportConn(basicDataSource);

        JdbcUserService.asynchrLogon(userService, "tom");//启动一个异步线程A
        JdbcUserService.sleep(500);

        //⑤此时线程A正在执行JdbcUserService#logon()方法
        JdbcUserService.reportConn(basicDataSource);
```

```
        JdbcUserService.sleep(2000);

        //⑥此时线程A所执行的JdbcUserService#logon()方法已经执行完毕
        JdbcUserService.reportConn(basicDataSource);

        JdbcUserService.asynchrLogon(userService, "john");//启动一个异步线程B
        JdbcUserService.sleep(500);

        //⑦此时线程B正在执行JdbcUserService#logon()方法
        JdbcUserService.reportConn(basicDataSource);

        JdbcUserService.sleep(2000);

        //⑧此时线程A和B都已完成JdbcUserService#logon()方法的执行
        JdbcUserService.reportConn(basicDataSource);

    }
```

在 JdbcUserService 中添加一个可异步执行 logon()方法的 asynchrLogon()方法，通过异步执行 logon()方法及让主线程睡眠的方式模拟多线程环境下的执行场景。在不同的执行点，通过 reportConn()方法汇报数据源连接的占用情况。

通过 Spring 事务声明，对 JdbcUserService 的 logon()方法进行事务增强，配置代码如代码清单 12-21 所示。

代码清单 12-21　applicatonContext.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    …
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    <context:component-scan base-package="com.smart.connleak"/>
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:driverClassName="${jdbc.driverClassName}"
          p:url="${jdbc.url}"
          p:username="${jdbc.username}"
          p:password="${jdbc.password}"/>

    <bean id="jdbcTemplate"
          class="org.springframework.jdbc.core.JdbcTemplate"
          p:dataSource-ref="dataSource"/>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
          p:dataSource-ref="dataSource"/>

    <!--①启用注解驱动的事务增强-->
    <tx:annotation-driven/>
</beans>
```

然后运行 JdbcUserService，在控制台上将观察到如下输出信息：

```
连接数[active:idle]-[0:0]
连接数[active:idle]-[2:0]
```

```
连接数[active:idle]-[1:1]
连接数[active:idle]-[3:0]
连接数[active:idle]-[2:1]
```

下面通过表 12-3 对数据源连接的占用和泄露情况进行描述。

表 12-3　执行过程数据源连接占用情况

时间	执行线程 1	执行线程 2	数据源连接		
			active	idle	leak
T0	未启动	未启动	0	0	0
T1	正在执行方法	未启动	2	0	0
T2	执行完毕	未启动	1	1	1
T3	执行完毕	正在执行方法	3	0	1
T4	执行完毕	执行完毕	2	1	2

可见，在执行线程 1 执行完毕后，只释放了一个数据连接，还有一个数据连接处于 active 状态，这说明泄露了一个连接。类似的，在执行线程 2 执行完毕后，也泄露了一个连接，原因是直接通过数据源获取连接（jdbcTemplate.getDataSource().getConnection()）而没有显式释放。

12.7.3　事务环境下通过 DataSourceUtils 获取数据连接

Spring 提供了一个能从当前事务上下文中获取绑定的数据连接的工具类，即 DataSourceUtils。Spring 强调必须使用 DataSourceUtils 获取数据连接，Spring 的 JdbcTemplate 内部也是通过 DataSourceUtils 来获取连接的。DataSourceUtils 提供了若干获取和释放数据连接的静态方法，说明如下。

- static Connection doGetConnection(DataSource dataSource)：首先尝试从事务上下文中获取连接，失败后再从数据源获取连接。
- static Connection getConnection(DataSource dataSource)：和 doGetConnection()方法的功能一样，实际上，其内部就是通过调用 doGetConnection()方法获取连接的。
- static void doReleaseConnection(Connection con, DataSource dataSource)：释放连接，放回到连接池中。
- static void releaseConnection(Connection con, DataSource dataSource)：和 doRelease Connection()方法的功能一样，实际上，其内部就是通过调用 doReleaseConnection()方法获取连接的。

下面看一下 DataSourceUtils 从数据源获取连接的关键代码，如代码清单 12-22 所示。

代码清单 12-22　DataSourceUtils

```
public abstract class DataSourceUtils {
…
public static Connection doGetConnection(DataSource dataSource) throws SQLException {
```

```
        Assert.notNull(dataSource, "No DataSource specified");

    //①首先尝试从事务同步管理器中获取数据连接
    ConnectionHolder conHolder =
         (ConnectionHolder) TransactionSynchronizationManager.getResource(dataSource);
    if (conHolder != null && (conHolder.hasConnection() ||
                    conHolder.isSynchronizedWithTransaction())) {
            conHolder.requested();
            if (!conHolder.hasConnection()) {
                logger.debug("Fetching resumed JDBC Connection from DataSource");
                conHolder.setConnection(dataSource.getConnection());
            }
            return conHolder.getConnection();
        }

        //②如果获取不到连接，则直接从数据源中获取连接
        Connection con = dataSource.getConnection();

        //③如果拥有事务上下文，则将连接绑定到事务上下文中
        if (TransactionSynchronizationManager.isSynchronizationActive()) {
            ConnectionHolder holderToUse = conHolder;
            if (holderToUse == null) {
                holderToUse = new ConnectionHolder(con);
            }
            else {holderToUse.setConnection(con);}
            holderToUse.requested();
            TransactionSynchronizationManager.registerSynchronization(
                        new ConnectionSynchronization(holderToUse, dataSource));
            holderToUse.setSynchronizedWithTransaction(true);
            if (holderToUse != conHolder) {
                TransactionSynchronizationManager.bindResource(
                                                dataSource, holderToUse);
            }
        }
        return con;
    }
    …
}
```

首先查看当前是否存在事务管理上下文，并尝试从事务管理上下文中获取连接。如果获取失败，则直接从数据源中获取连接。在获取连接后，如果当前拥有事务上下文，则将连接绑定到事务上下文中。

在代码清单 12-23 的 JdbcUserService 中，使用 DataSourceUtils.getConnection()方法替换直接从数据源中获取连接的代码。

代码清单 12-23　JdbcUserService

```
package com.smart.connleak;
…
@Service("jdbcUserService")
public class JdbcUserService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
```

```
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Transactional
    public void logon(String userName) {
        try {
            //①使用DataSourceUtils获取数据连接
          Connection conn =
                DataSourceUtils.getConnection(jdbcTemplate.getDataSource());
            //Connection conn = jdbcTemplate.getDataSource().getConnection();

            String sql = "UPDATE t_user SET last_logon_time=? WHERE user_name =?";
            jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

重新运行代码，得到如下执行结果：

```
连接数[active:idle]-[0:0]
连接数[active:idle]-[1:0]
连接数[active:idle]-[0:1]
连接数[active:idle]-[1:0]
连接数[active:idle]-[0:1]
```

对照代码清单 12-23 的输出日志，可以看到已经没有连接泄露现象了。一个执行线程在运行 JdbcUserService#logon()方法时只占用一个连接，而且方法执行完毕后，该连接马上释放。这说明通过 DataSourceUtils.getConnection()方法确实获取了方法所在事务上下文绑定的那个连接，而不是像原来那样从数据源中获取一个新的连接。

12.7.4 无事务环境下通过 DataSourceUtils 获取数据连接

然而，是否使用 DataSourceUtils 获取数据连接就可以高枕无忧了呢？理想很丰满，现实很骨感：如果 DataSourceUtils 在没有事务上下文的方法中使用 getConnection()方法获取连接，那么依然会造成数据连接泄露。

保持代码清单 12-23 的代码不变，将 Spring 配置文件（代码清单 12-20）中①处的 Spring AOP 事务增强配置的代码注释掉，让代码运行在无事务环境下。重新运行代码清单 12-23 的代码，将得到如下输出日志：

```
连接数[active:idle]-[0:0]
连接数[active:idle]-[1:1]
连接数[active:idle]-[1:1]
连接数[active:idle]-[2:1]
连接数[active:idle]-[2:1]
```

下面通过表 12-4 对数据源连接的占用和泄露情况进行描述。

表 12-4　执行过程中数据源连接占用情况

时间	执行线程 1	执行线程 2	数据源连接		
			active	idle	leak
T0	未启动	未启动	0	0	0
T1	正在执行方法	未启动	1	1	0
T2	执行完毕	未启动	1	1	1
T3	执行完毕	正在执行方法	2	1	1
T4	执行完毕	执行完毕	2	1	2

仔细对照表 12-3 的执行过程，我们发现，在 T1 时刻，有事务上下文时的 active 为 2，idle 为 0，而此时由于没有事务管理，则 active 为 1，而 idle 也为 1。这说明有事务上下文时，需要等到整个事务方法（logon()）返回后，事务上下文绑定的连接才会被释放。但在没有事务上下文时，logon()方法调用 JdbcTemplate 执行完数据操作后，会立即释放连接。

在 T2 时刻执行线程完成 logon()方法的调用后，有一个连接没有被释放（active），所以发生了连接泄露。到 T4 时刻，两个执行线程都完成了 logon()方法的调用，但是出现了两个未释放的连接。

要堵上这个连接泄露的漏洞，需要对 logon()方法进行如下改造，如代码清单 12-24 所示。

代码清单 12-24　JdbcUserService

```
package com.smart.connleak;
…
@Service("jdbcUserService")
public class JdbcUserService {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
          this.jdbcTemplate = jdbcTemplate;
    }

    @Transactional
    public void logon(String userName) {
       Connection conn = null;
       try {
          conn = DataSourceUtils.getConnection(jdbcTemplate.getDataSource());
          String sql = "UPDATE t_user SET last_logon_time=? WHERE user_name =?";
          jdbcTemplate.update(sql, System.currentTimeMillis(), userName);
          Thread.sleep(1000);
          //①
       }catch (Exception e) {
          e.printStackTrace();
       }finally {
          //②显式使用DataSourceUtils释放连接
          DataSourceUtils.releaseConnection(conn,jdbcTemplate.getDataSource());
```

```
        }
    }
}
```

在②处显式调用 DataSourceUtils.releaseConnection()方法释放获取的连接。需要特别指出的是，一定不能在①处释放连接。因为如果 logon()方法在获取连接后，①处代码前这段代码执行时发生异常，则①处释放连接的动作将得不到执行。这将是一个非常具有隐蔽性的连接泄露的隐患点。

12.7.5 JdbcTemplate 如何做到对连接泄露的免疫

分析 JdbcTemplate 的代码，可以清楚地看到它开放的每个数据操作方法：首先使用 DataSourceUtils 获取连接，然后在方法返回之前使用 DataSourceUtils 释放连接。

下面看一下 JdbcTemplate 最核心的数据操作方法 execute()，如代码清单 12-25 所示。

代码清单 12-25 JdbcTemplate#execute()

```
public <T> T execute(StatementCallback<T> action) throws DataAccessException {

        //①首先根据DataSourceUtils获取数据连接
        Connection con = DataSourceUtils.getConnection(getDataSource());
        Statement stmt = null;
        try {
            Connection conToUse = con;
            …
            handleWarnings(stmt);
            return result;
        }
        catch (SQLException ex) {
            JdbcUtils.closeStatement(stmt);
            stmt = null;

           //②发生异常时，使用DataSourceUtils释放数据连接
           DataSourceUtils.releaseConnection(con, getDataSource());
            con = null;
            throw getExceptionTranslator().translate(
                        "StatementCallback", getSql(action), ex);
        }
        finally {
            JdbcUtils.closeStatement(stmt);

           //③最后再使用DataSourceUtils释放数据连接
            DataSourceUtils.releaseConnection(con, getDataSource());
        }
}
```

在①处通过 DataSourceUtils.getConnection()方法获取连接，在②和③处通过 DataSourceUtils.releaseConnection()方法释放连接。所有 JdbcTemplate 开放的数据访问 API 最终都是直接或间接由 execute(StatementCallback<T> action)方法执行数据访问操作的，因此这个方法代表了 JdbcTemplate 数据操作的最终实现方式。

正是因为 JdbcTemplate 严谨的获取连接及释放连接的模式化流程保证了 JdbcTemplate 对数据连接泄露问题的免疫性。所以，如有可能，应尽量使用 JdbcTemplate、HibernateTemplate 等模板进行数据访问操作，避免直接获取数据连接的操作。

12.7.6　使用 TransactionAwareDataSourceProxy

如果不得已要显式获取数据连接，除了可以使用 DataSourceUtils 获取事务上下文绑定的连接，还可以通过 TransactionAwareDataSourceProxy 对数据源进行代理。数据源对象被代理后就具有了事务上下文感知的能力，通过代理数据源的 getConnection()方法获取连接和使用 DataSourceUtils.getConnection()方法获取连接的效果是一样的。

下面是使用 TransactionAwareDataSourceProxy 对数据源进行代理的配置，如代码清单 12-26 所示。

代码清单 12-26　applicatonContext.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"
      …
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
   <context:component-scan base-package="com.smart.connleak"/>
   <context:property-placeholder location="classpath:jdbc.properties"/>

   <!--①未被代理的数据源 -->
   <bean id="originDataSource" class="org.apache.commons.dbcp.BasicDataSource"
         destroy-method="close"
         p:driverClassName="${jdbc.driverClassName}"
         p:url="${jdbc.url}"
         p:username="${jdbc.username}"
         p:password="${jdbc.password}"/>

    <!--②对数据源进行代理，使数据源具有事务上下文感知性 -->
    <bean id="dataSource"
         class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy"
         p:targetDataSource-ref="originDataSource" />

   <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate"
        p:dataSource-ref="dataSource"/>

   <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>
   <tx:annotation-driven/>
</beans>
```

对数据源进行代理后，就可以通过数据源代理对象的 getConnection()方法获取事务上下文中绑定的数据连接了。因此，如果数据源已经进行了 TransactionAwareDataSourceProxy

代理，而且方法存在事务上下文，那么代码清单 12-19 的代码也不会生产连接泄露的问题。

12.7.7 其他数据访问技术的等价类

理解了 Spring JDBC 的数据连接泄露问题，其中的道理可以平滑地推广到其他框架中。Spring 为每个数据访问技术框架都提供了一个获取事务上下文绑定的数据连接（或其衍生品）的工具类和数据源（或其衍生品）的代理类。

表 12-5 列出了不同数据访问技术框架对应 DataSourceUtils 的等价类。

表 12-5 不同数据访问技术框架DataSourceUtils的等价类

数据访问技术框架	连接（或衍生品）获取工具类
Spring JDBC	org.springframework.jdbc.datasource.DataSourceUtils
Hibernate	org.springframework.orm.hibernate*X*.SessionFactoryUtils
MyBatis	org.springframework.jdbc.datasource.DataSourceUtils
JPA	org.springframework.orm.jpa.EntityManagerFactoryUtils
JDO	org.springframework.orm.jdo.PersistenceManagerFactoryUtils

表 12-6 列出了不同数据访问技术框架下 TransactionAwareDataSourceProxy 的等价类。

表 12-6 不同数据访问技术框架下TransactionAwareDataSourceProxy的等价类

数据访问技术框架	连接（或衍生品）获取工具类
Spring JDBC	org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy
Hibernate	org.springframework.orm.hibernate*X*.LocalSessionFactoryBean
MyBatis	org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy
JPA	无
JDO	org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy

12.8 小结

Spring 声明式事务是 Spring 最核心、最常用的功能。由于 Spring 通过 IoC 和 AOP 非常透明地实现了声明式事务的功能，所以一般的开发者基本上无须了解 Spring 声明式事务的内部细节，仅需懂得如何配置即可。

但在实际应用开发过程中，Spring 的这种透明的高阶封装在带来便利的同时，也带来了迷惑。就像通过流言传播的消息，最终听众已经不清楚事情的真相，而这对于应用开发来说是很危险的。本章剖析了实际应用中给开发者造成迷惑的各种难点，通过分析 Spring 事务管理的内部运作机制将真相还原出来。

通过本章的学习，我们知道了以下真相。

- 在没有事务管理的情况下，DAO 照样可以顺利地进行数据操作。
- 将应用分成 Web、Service 及 DAO 层只是一种参考的开发模式，并非是事务管理工作的前提条件。
- Spring 通过事务传播机制可以很好地应对事务方法嵌套调用的情况，开发者无须为了事务管理而刻意改变服务方法的设计。
- 由于单实例的对象不存在线程安全问题，所以经过事务管理增强的单实例 Bean 可以很好地工作在多线程环境下。
- 混合使用多个数据访问技术框架的最佳组合是一个 ORM 技术框架（如 Hibernate 或 JPA 等）+一个 JDBC 技术框架（如 Spring JDBC 或 MyBatis）。直接使用 ORM 技术框架对应的事务管理器就可以了，但必须考虑 ORM 缓存同步的问题。
- Spring AOP 事务增强有两个方案：其一为基于接口的动态代理；其二为基于 CGLib 动态生成子类的代理。由于 Java 语法的特性，有些特殊方法不能被 Spring AOP 代理，所以也就无法享受 AOP 织入带来的事务增强；
- 在使用 Spring JDBC 时如果直接获取 Connection，则可能会造成连接泄露。为了降低连接泄露的可能性，尽量使用 DataSourceUtils 获取数据连接。也可以对数据源进行代理，以便使数据源拥有感知事务上下文的能力。
- 可以将 Spring JDBC 防止连接泄露的解决方案平滑地应用到其他数据访问技术框架中。

第 13 章

使用 Spring JDBC 访问数据库

Spring JDBC 是 Spring 所提供的持久层技术。它的主要目的是降低使用 JDBC API 的门槛，以一种更直接、更简洁的方式使用 JDBC API。在 Spring JDBC 里，仅需做那些与业务相关的 DML 操作的事，而将资源获取、Statement 创建、资源释放及异常处理等繁杂而乏味的工作交给 Spring JDBC。

虽然 ORM 的框架已经成熟丰富，但 JDBC 的灵活、直接的特性，依然让它拥有自己的用武之地。如在完全依赖查询模型动态产生查询语句的综合查询系统中，Hibernate、MyBatis、JPA 等框架都无法使用，这里 JDBC 是唯一的选择。

本章主要内容：

- 使用 JdbcTemplate 模板类进行 CRUD 数据操作
- BLOB 及 CLOB 类型数据的操作
- 支持命名参数绑定 NamedParameterJdbcTemplate 模拟类的使用
- 关于主键产生、获取的知识

本章亮点：

- 详细分析并讨论了主键的产生、获取方式，并比较了它们的优劣
- 对 LOB 数据操作进行详细分析，这些讨论在其他 ORM 框架中具有普适性
- 以 MySQL 和 Oracle 两个颇具代表性的数据库为基础进行实例设计

13.1 使用 Spring JDBC

但凡 Java 开发者都有过直接使用 JDBC 编写数据库程序的经历，由于 JDBC API 过于底层，开发者不但需要编写数据操作代码，还需要编写获取 JDBC 连接、处理异常、释放资源等代码。即使一个再简单不过的数据库操作，也需要至少十几行代码。Spring

JDBC 通过模板和回调机制大大降低了使用 JDBC 的复杂度，借由 JdbcTemplate 的帮助，仅需要编写那些“必不可少”的代码就可以进行数据库操作。

13.1.1　JdbcTemplate 小试牛刀

几乎可以用 JdbcTemplate 完成任何数据访问操作，并充分享受 JdbcTemplate 所带来的简洁。下面是使用 JdbcTemplate 创建一张表的例子，如代码清单 13-1 所示。

代码清单 13-1　使用JdbcTemplate创建一张表

```
DriverManagerDataSource ds = new DriverManagerDataSource();①
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3309/sampledb");
ds.setUsername("root");
ds.setPassword("1234");
JdbcTemplate jdbcTemplate = new JdbcTemplate();②
jdbcTemplate.setDataSource(ds);
String sql = "create table t_user(user_id int primary key,user_name varchar(60))";③
jdbcTemplate.execute(sql);
```

创建一个数据源

生成一个 JdbcTemplate 实例

创建一张表

在代码清单 13-1 中，首先使用 DriverManagerDataSource()方法创建了一个数据源，接着创建了一个 JdbcTemplate 对象，最后使用该对象执行 SQL 语句。运行以上代码，将在 MySQL 数据库中创建一张 t_user 表。

通过以上实例可以感受到 JdbcTemplate 所开创的全新的编程风格，代码清单 13-1 中粗体显示的部分就是移除闲杂代码后简洁到极致的代码。当然，从代码行上看，代码清单 13-1 依旧拥有 9 行代码，似乎并没有做到真正的简洁。这是因为为了保证实例的完整性，特意将数据源创建和模板实例创建的代码都列在这个例子中，但在实际应用中，用户一般不会在 DAO 中做这些事情。由于 JdbcTemplate 是线程安全的，因而所有的 DAO 都可以共享同一个 JdbcTemplate 实例，这样①和②部分的代码就可以从 DAO 中移除了，转而在 Spring 配置文件中统一定义即可。

提示

本章后面内容所涉及的数据库表的脚本位于本书配套网盘中的 chapter13/schema 目录下，提供了 MySQL 和 Oracle 两个数据库版本的 SQL 脚本。

13.1.2　在 DAO 中使用 JdbcTemplate

一般情况下都是在 DAO 类中使用 JdbcTemplate，JdbcTemplate 在 XML 配置文件中配置好后，直接在 DAO 中注入即可。

```
package com.smart.dao;
...
```

```
//①声明一个 DAO
@Repository
public class ForumDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired  //②注入JdbcTemplate实例
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    public void initDb() {

        //注意：在实际项目中，如果SQL不是动态组装的，则应将其定义成一个类级的静态final常量，
        //此处仅为方便代码阅读，所以放在方法内部定义，下同
        String sql = "create table t_user(user_id int primary key,user_name varchar(60))";
        jdbcTemplate.execute(sql);
    }
}
```

按照相同的方式，就可以方便地创建其他的 DAO 类，如 TopicDao、PostDao 等。在 Spring 配置文件中定义 JdbcTemplate 并注入每个 DAO 中。

```
 <!--①扫描包以注册注解声明的 Bean -->
<context:component-scan base-package="com.smart"/>

 <!--②配置数据源 -->
 <context:property-placeholder location="classpath:jdbc.properties" />
 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}" />

<!--③声明 JdbcTemplate Bean -->
 <bean id="jdbcTemplate"
     class="org.springframework.jdbc.core.JdbcTemplate"
     p:dataSource-ref="dataSource"/>
```

在 Spring 配置文件中配置 DAO 一般分为 4 个步骤。

（1）定义 DataSource。

（2）定义 JdbcTemplate。

JdbcTemplate 拥有几个可用于控制底层 JDBC API 的属性，合理地设置这些属性可以获得良好的程序性能。

- queryTimeout：设置 JdbcTemplate 所创建的 Statement 查询数据时的最大超时时间。默认为 0，表示使用底层 JDBC 驱动程序的默认设置。
- fetchSize：设置底层的 ResultSet 每次从数据库返回的行数。该属性对程序的性能影响很大，如果设置过大，因为一次性载入的数据都放到内存中，所以内存的消耗很大；反之，如果设置过小，从数据库读取的次数将增大，也会影响性能。默认值为 0，表示使用底层 JDBC 驱动程序的默认设置。Oracle 驱动程序的 fetchSize 的默认值为 10。

- maxRows：设置底层的 ResultSet 从数据库返回的最大行数。默认值为 0，表示使用底层 JDBC 驱动程序的默认设置。
- ignoreWarnings：是否忽略 SQL 的警告信息。默认为 true，即所有的警告信息都被记录到日志中；如果设置为 false，则 JdbcTemplate 将抛出 SQLWarningException。

（3）声明一个抽象的<bean>，以便所有 DAO 复用配置 JdbcTemplate 属性的配置。

（4）配置具体的 DAO。

实战经验

Spring 几乎为所有的模板类都提供了相应的支持类，与 JdbcTemplate 对应的支持类为 JdbcDaoSupport。JdbcDaoSupport 内部定义了 JdbcTemplate 的成员变量，开发者可以通过扩展 JdbcDaoSupport 定义自己的 DAO，这也是早期 Spring 开发者常用的做法。但是随着 Bean 的注解配置逐渐成为主流配置方式，这种做法就显得不合时宜了，因为直接继承 JdbcDaoSupport 无法对 JdbcTemplate 成员变量应用@Autowired 注解。所以我们推荐的做法是自己定义一个 BaseDao，在 BaseDao 中定义 JdbcTemplate 成员变量，并在其 setter 方法上标注@Autowired 注解。

在实际项目中，还会在 BaseDao 中定义一些通用的功能，如声明 JdbcTemplate、分页查询等。

13.2　基本的数据操作

数据库的增、删、查、改及存储过程调用是最常见的数据库操作，JdbcTemplate 提供了众多方法，通过 JdbcTemplate 可以用简单的方法完成这些数据操作。

13.2.1　更改数据

JdbcTemplate 提供了若干个 update()方法，允许对数据表记录进行更改和删除操作。下面的 ForumDao 定义了一个新增论坛版块的方法，如代码清单 13-2 所示。

代码清单 13-2　更新数据

```
package com.smart.dao;
…
@Repository
public class ForumDao {
    private JdbcTemplate jdbcTemplate;

   @Autowired
   public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
```

```
    }

    public void addForum(Forum forum) {
        String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)"; ①
        Object[ ] params = new Object[ ]{forum.getForumName(),forum. getForumDesc()};②
        jdbcTemplate.update(sql, params); ③
    }
    ...
}
```

由于 JdbcTemplate 在内部通过 PreparedStatement 执行 SQL 语句，所以可以使用绑定参数的 SQL 语句，如①处所示，每个“?”占位符可接受一个参数。在②处定义了用于填充占位符的参数数组，在③处直接通过 JdbcTemplate 的 int update(String sql, Object[] args)方法进行表数据的更新。

JdbcTemplate 已经把原来复杂的数据更新操作简化为定义 SQL 语句、准备参数、调用 update()这 3 个至简的步骤，这不由得让我们想起了《登徒子好色赋》中的东家之子——增之一分则太长，减之一分则太短。

实战经验

尽量使用可绑定参数的 SQL 语句，以便数据库可以复用 SQL 的执行计划，提高数据库的执行效率。此外，应尽量在 DAO 中使用类级别的静态常量（final static）定义 SQL 字符串，不应在方法内部声明 SQL 字符串变量，以提高 JVM 的内存使用效率。

在通过 update(String sql, Object[] args)方法为 SQL 语句的占位符绑定参数时，并没有显式指定对应字段的数据类型，此时，Spring 直接让 PreparedStatement 根据参数的类型进行“猜测”。在上例中，由于 t_forum 的 forum_name 和 forum_desc 都是 varchar 类型的，所以并不会发生错误。但一种更好的做法是使用 int update(String sql, Object[] args, int[] argTypes)显式指定每个占位符所对应的字段数据类型，这样就可以保证类型安全，当参数值为 null 时，这种形式提供了更好的支持。可以对代码清单 13-2 进行以下调整：

```
package com.smart.dao;
import java.sql.Types; ①
...
@Repository
public class ForumDao {
    ...
    public void addForum(Forum forum) {
        ...
        jdbcTemplate.update(sql, params,new int[]{Types.VARCHAR, Types.VARCHAR});
    }
}
```

使用该类中的常量属性定义参数类型

除了以上两个 update()方法外，JdbcTemplate 还提供了以下几个功能相似的重载方法。

- int update(String sql)：为不带占位符的 SQL 语句所提供的便利方法。
- int update(String sql, Object... args)：使用不定参数的方法，和 update(String sql, Object[] args)相似。

- int update(String sql, PreparedStatementSetter pss)：PreparedStatementSetter 是一个回调接口，它定义了一个 void setValues(PreparedStatement ps)接口方法。JdbcTemplate 使用 SQL 语句创建出 PreparedStatement 实例后，将调用该回调接口执行绑定参数的操作。可以按如下方式使用该回调接口：

```
//①为了 forum 可以在内部类中使用，必须声明为 final
public void addForum(final Forum forum) {
  ...
  //②通过匿名内部类定义回调实例
  jdbcTemplate.update(sql, new PreparedStatementSetter() {
     public void setValues(PreparedStatement ps) throws SQLException {
         ps.setString(1, forum.getForumName());
         ps.setString(2, forum.getForumDesc());
     }
  });
}
```

提示

PreparedStatement 绑定参数时，参数索引从 1 开始而非从 0 开始，第一个参数索引为 1，第二个参数索引为 2，以此类推。

- int update(PreparedStatementCreator psc)：PreparedStatementCreator 也是一个回调接口，它负责创建一个 PreparedStatement 实例。该回调接口定义了一个 PreparedStatement createPreparedStatement(Connection con)方法。同样可以使用匿名内部类定义一个 PreparedStatementCreator 实例，如下：

```
public void addForum(final Forum forum) {
    //①由于 sql 变量需要在内部类中使用，所以将其声明为 final
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)";

    jdbcTemplate.update(new PreparedStatementCreator(){
        public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, forum.getForumName());
            ps.setString(2, forum.getForumDesc());
            return ps;
        }
    });
}
```

- protected int update(PreparedStatementCreator psc, PreparedStatementSetter pss)：联合使用 PreparedStatementCreator 和 PreparedStatementSetter 回调。

需要指出的是，在实际使用中，应优先考虑使用不带回调接口的 JdbcTemplate 方法。首先，回调使代码显得臃肿；其次，回调并不能带来额外的好处。当使用 update(String sql,Object[] params)等简洁版的方法时，JdbcTemplate 会在内部自动创建这些回调实例，我们没有必要越俎代庖。

13.2.2 返回数据库的表自增主键值

有些开发者喜欢使用数据库自增字段作为表主键，即主键值不在应用层产生，而是在新增记录时由数据库产生。这样，应用层在保存对象前并不知道对象主键值，而必须在保存数据后才能从数据库中返回主键值。在很多情况下，我们需要获取新对象持久化后的主键值。在 Hibernate、JPA 等 ORM 框架中，新对象在持久化之后，主键值会自动绑定到对象上，给程序的开发带来了很多方便。

在 JDBC 3.0 规范中，当新增记录时，允许将数据库自动产生的主键值绑定到 Statement 或 PreparedStatement 中。在使用 Statement 时，可以通过以下方法绑定主键值：

```
int executeUpdate(String sql,int autoGeneratedKeys)
```

也可以通过 Connection 创建绑定自增主键值的 PreparedStatement，如下：

```
PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)
```

当 autoGeneratedKeys 参数设置为 Statement.RETURN_GENERATED_KEYS 时，即可绑定数据库产生的主键值；当设置为 Statement.NO_GENERATED_KEYS 时，不绑定主键值。下面的代码演示了 Statement 绑定并获取数据库产生的主键值的过程：

```
Statement stmt = conn.createStatement();
String sql = "INSERT INTO t_topic(topic_title,user_id) VALUES('测试主题','123')";

//①指定绑定表自增主键值
stmt.executeUpdate(sql,Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if ( rs.next() ) {
   int key = rs.getInt();//②获取对应的表自增主键值
}
```

Spring 利用这一技术，提供了一个可以返回新增记录对应主键值的方法，如下：

```
int update(PreparedStatementCreator psc, KeyHolder generatedKeyHolder)
```

org.springframework.jdbc.support.KeyHolder 是一个回调接口，Spring 使用它保存新增记录对应的主键，该接口的接口方法描述如下。

- Number getKey() throws InvalidDataAccessApiUsageException：当仅插入一行数据，主键不是复合键而是数字类型时，通过该方法可以直接返回新的主键值。如果是复合主键，或者有多个主键返回时，则该方法抛出 InvalidDataAccessApiUsageException。该方法是最常用的方法，因为在一般情况下，一次仅插入一条数据，并且主键字段类型为数字类型。
- Map<String,Object> getKeys() throws InvalidDataAccessApiUsageException：如果是复合主键，则列名和列值构成 Map 中的一个 Entry。如果返回的是多个主键，则该方法抛出 InvalidDataAccessApiUsageException 异常。
- List<Map<String,Object>> getKeyList()：如果返回多个主键，即 PreparedStatement 新增了多条记录，则每个主键对应一个 Map，多个 Map 构成一个 List。

Spring 为 KeyHolder 接口指代了一个通用的实现类 GeneratedKeyHolder，该类返回

新增记录时的自增长主键值。假设希望在新增论坛版块对象后，将主键值加载到对象中，则可以按以下代码进行调整：

```
public void addForum(final Forum forum) {
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)";

    KeyHolder keyHolder = new GeneratedKeyHolder();//①创建一个主键持有者
    jdbcTemplate.update(new PreparedStatementCreator(){
        public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
                PreparedStatement ps = conn.prepareStatement(sql);
                ps.setString(1, forum.getForumName());
                ps.setString(2, forum.getForumDesc());
                return ps;
            }
      },keyHolder);
        forum.setForumId(keyHolder.getKey().intValue());//②从主键持有者中获取主键
}
```

这样，在调用 addForum(final Forum forum)新增 forum 领域对象后，forum 将拥有对应的主键值，方便后继的使用。

在 JDBC 3.0 之前的版本中，PreparedStatement 不能绑定主键，如果采用表自增键（如 MySQL 的 auto increment 或 SQL Server 的 identity），则将给获取正确的主键值带来挑战，因为用户必须在插入数据后立即执行另一条获取新增主键的查询语句。表 13-1 给出了不同数据库获取最新自增主键值的查询语句。

表 13-1　不同数据库获取新增加的主键值

数 据 库	获取新增主键值的查询语句
DB2	IDENTITY_VAL_LOCAL()
Informix	SELECT dbinfo('sqlca.sqlerrd1') FROM <TABLE>
Sybase	SELECT @@IDENTITY
SQLServer	SELECT SCOPE_IDENTITY()或 SELECT @@IDENTITY
MySQL	SELECT LAST_INSERT_ID()
HSQLDB	CALL IDENTITY()
Cloudscape	IDENTITY_VAL_LOCAL()
Derby	IDENTITY_VAL_LOCAL()
PostgreSQL	SELECT nextval('<TABLE>_SEQ')

如果数据库的并发率很高，如在插入记录后执行查询主键前，数据库又执行了若干条插入记录的 SQL 语句，这时，通过表 13-1 返回的主键值就是最后一条插入语句的主键值，而非我们希望的主键值。所以使用查询语句获取表自增键值是不安全的，这也是为什么有些数据库（如 Oracle、Firebird）故意不提供自增键，而只提供序列的原因，序列强制要求用户在新增记录前先获取主键值。Oracle 通过 SELECT <*SEQUENCE_NAME*>.nextval FROM DUAL 获取序列的下一个值，而 Firebird 通过 SELECT GEN_ID (<*SEQUENCE_NAME*> 1) FROM RDB$DATABASE 获取序列的下一个值。13.4.1 节将讲解应用层自增键的相关知识。

实战经验

在实际开发中，我们并不太建议使用表自增键，因为这种方式会让开发更加复杂且降低程序移植性，在应用层中创建主键才是主流的方式，可以使用 UUID 或者通过一个编码引擎获取主键值。我们建议尽量少用数据库的自增键、外键、触发器、存储过程、数据库函数等高级功能，让数据库只负责数据存储和查询（在进行表设计时只应关注表空间、索引、非空限制及唯一值限制等和存储查询相关的特性），不要让它负责业务逻辑，业务逻辑处理应在应用层中进行。

13.2.3 批量更改数据

如果需要一次性插入或更新多条记录，当然可以简单地通过多次调用 update()方法完成任务，但这不是最好的实现方案。更好的选择是使用 JdbcTemplate 批量数据更改的方法。一般情况下，后者拥有更好的性能，因为更新的数据将被批量发送到数据库中，它减少了对数据库的访问次数。下面先来认识一下 JdbcTemplate 的两个批量数据操作方法。

- public int[] batchUpdate(String[] sql)：多条 SQL 语句组成一个数组（这些 SQL 语句不带参数），该方法以批量方式执行这些 SQL 语句。Spring 在内部使用 JDBC 提供的批量更新 API 完成操作。如果底层的 Jdbc Driver 不支持批量更新操作，那么 Spring 将采用逐条更新的方式模拟批量更新。
- int[] batchUpdate(String sql, BatchPreparedStatementSetter pss)：使用该方法对于同一结构的带参SQL语句多次进行数据更新操作。通过BatchPreparedStatementSetter回调接口进行批量参数的绑定工作。BatchPreparedStatementSetter 定义了两个方法。
  - int getBatchSize()：指定本批次的大小。
  - void setValues(PreparedStatement ps, int i)：为给定的 PreparedStatement 设置参数。

假设需要一次性新增多个论坛版块，则可以通过编写如下方法来满足要求，如代码清单 13-3 所示。

代码清单 13-3 批量插入操作

```
public void addForums(final List<Forum> forums) {
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc) VALUES(?,?)";

    jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

        //①指定该批的记录数
      public int getBatchSize() {
          return forums.size();
      }

        //②绑定插入的参数
```

```
        public void setValues(PreparedStatement ps, int index)
                throws SQLException {
            Forum forum = forums.get(index);
            ps.setString(1, forum.getForumName());
            ps.setString(2, forum.getForumDesc());
        }
    });
}
```

有的开发者认为 BatchPreparedStatementSetter 内部会自动将需要更改的数据分多批传送到数据库中，而每批的大小通过 getBatchSize()方法指定。其实，这样理解是错误的。实际上，BatchPreparedStatementSetter 是一次性批量提交数据的，而不会分批提交，getBatchSize()是整批的大小。所以，如果希望将一个 List 中的数据通过 BatchPrepared StatementSetter 批量更新到数据库中，getBatchSize()就应该设置为 List 的大小。如果 List 非常大，希望分多次批量提交，则可以分段读取这个大的 List 并暂存到一个小的 List 中，再将这个小的 List 通过 BatchPreparedStatementSetter 批量保存到数据库中。

13.2.4　查询数据

请回想一下使用原生 JDBC 查询数据时有哪些操作是必不可少的——是的，从结果集中获取数据是每个查询功能不可或缺的操作。在 Spring JDBC 中，仅需要指定 SQL 查询语句并定义好如何从结果集中返回数据就可以了。

1．使用 RowCallbackHandler 处理结果集

Spring 提供了 org.springframework.jdbc.core.RowCallbackHandler 回调接口，通过该接口可以定义如何从结果集中获取数据。RowCallbackHandler 接口很简单，仅有一个方法：void processRow(ResultSet rs) throws SQLException。Spring 会遍历结果集，对结果集中的每一行调用 RowCallbackHandler 回调接口处理数据。所以无须调用 ResultSet 的 next()方法，而只需定义好如何获取结果集数据的逻辑就可以了。

ForumDao 需要提供一个根据 forumId 获取 Forum 对象的方法，下面是该方法的具体实现，如代码清单 13-4 所示。

代码清单 13-4　单条数据结果集的处理

```
public Forum getForum(final int forumId) {
    String sql = "SELECT forum_name,forum_desc FROM t_forum WHERE forum_id=?";
    final Forum forum = new Forum();

    //①将结果集数据行中的数据抽取到 forum 对象中
    jdbcTemplate.query(sql,new Object[]{forumId}, new RowCallbackHandler(){
        public void processRow(ResultSet rs) throws SQLException {
            forum.setForumId(forumId);
            forum.setForumName(rs.getString("forum_name"));
            forum.setForumDesc(rs.getString("forum_desc"));
        }});
    return forum;
}
```

虽然 processRow(ResultSet rs)方法会抛出 SQLException 异常，但我们无须关注它，因为 JdbcTemplate 会将 SQLException 转换为 Spring DAO 通用层的运行期异常。

如果需要获取多条记录，依然可以使用 RowCallbackHandler 完成任务，只需稍微调整一下结果集的处理逻辑就可以了。下面的代码可以通过论坛版块 ID 的范围值获取 Forum 对象的列表，如代码清单 13-5 所示。

代码清单 13-5　多条数据结果集的处理

```
public List<Forum> getForums(final int fromId,final int toId) {
        String sql = "SELECT forum_id,forum_name,forum_desc FROM t_forum WHERE
                      forum_id between ? and ?";
        final List forums = new ArrayList ();

        jdbcTemplate.query(sql,new Object[]{fromId,toId},new RowCallback
Handler(){①  将结果集中的数据映射到 List 中
            public void processRow(ResultSet rs) throws SQLException {
                Forum forum = new Forum();
                forum.setForumId(rs.getInt("forum_id"));
                forum.setForumName(rs.getString("forum_name"));
                forum.setForumDesc(rs.getString("forum_desc"));
                forums.add(forum);
            }});
        return forums;
}
```

当结果集中没有数据时，并不会抛出异常，只是此时 RowCallbackHandler 回调接口中定义的处理逻辑没有得到调用罢了。

代码清单 13-4 和代码清单 13-5 都使用了 JdbcTemplate 的 void query(String sql, Object[] args, RowCallbackHandler rch)方法。该方法的参数类型安全的重载版本是 void query(String sql, Object[] args, **int[] argTypes**, RowCallbackHandler rch)。此外还有以下几个功能相似的重载版本：

- void query(PreparedStatementCreator psc, RowCallbackHandler rch)。
- void query(String sql, PreparedStatementSetter pss, RowCallbackHandler rch)。
- void query(String sql, RowCallbackHandler rch)。

2．使用 RowMapper<T>处理结果集

Spring 还提供了一个和 RowCallbackHandler 功能上类似的 RowMapper<T>接口，也可以使用 RowMapper<T>接口定义结果集映射逻辑。在结果集为多行记录时，该接口更容易使用。RowMapper<T>接口只有一个接口方法：T mapRow(ResultSet rs, int rowNum)。

JdbcTemplate 提供了若干个使用 RowMapper<T>查询数据的方法，这些方法的返回类型为 List。下面通过 JdbcTemplate 的 List query(String sql, Object[] args, RowMapper<T> rowMapper)方法对代码清单 13-5 进行改造，如代码清单 13-6 所示。

代码清单 13-6　使用RowMapper映射多行数据

```
public List<Forum> getForums(final int fromId,final int toId) {
        String sql = "SELECT forum_id,forum_name,forum_desc FROM t_forum WHERE
                      forum_id between ? and ?";
```

```
        return jdbcTemplate.query(sql,new Object[]{fromId,toId},new RowMapper
<Forum>(){
            public Forum mapRow(ResultSet rs, int index) throws SQLException {
                Forum forum = new Forum();
                forum.setForumId(rs.getInt("forum_id"));
                forum.setForumName(rs.getString("forum_name"));
                forum.setForumDesc(rs.getString("forum_desc"));
                return forum;
            }
        });
}
```

粗体部分为 RowMapper<T>的匿名内部类，参照代码清单 13-5，可以发现，在 RowCallbackHandler 的 void processRow(ResultSet rs)方法中，需要将行数据映射对象 Forum 手工添加到外部定义的 forums 列表中；而在 RowMapper<T>的 T mapRow(ResultSet rs, int index)方法中，仅需简单地定义结果集行和对象的映射关系即可，创建 List<T>对象、将行数据映射对象添加到 List<T>中等操作都由 JdbcTemplate 代劳了。这也是为什么说 RowMapper<T>更适合在多行结果集时使用的原因。

JdbcTemplate 使用 RowMapper<T>的其他几个接口方法说明如下。

- <T> List<T> query(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)。
- <T> List<T> query(String sql, RowMapper<T> rowMapper)。
- <T> List<T> query(String sql, PreparedStatementSetter pss, RowMapper<T> rowMapper)。
- <T> List<T> query(PreparedStatementCreator psc, RowMapper<T> rowMapper)。

3．RowCallbackHandler 和 RowMapper<T>的比较

从功能上来说，RowCallbackHandler 和 RowMapper<T>并没有太大的区别，它们都用于定义结果集行的读取逻辑，将 ResultSet 中的数据映射到对象或 List 中。

Spring 宣称 RowCallbackHandler 接口实现类可以是有状态的，而 RowMapper<T>的实现类应该是无状态的。如果 RowCallbackHandler 接口实现类是有状态的，就不能在多个地方复用，只有无状态的实例才能在不同的地方复用。

这样讲未免过于抽象，来看一下 Spring 自身的几个接口实现类。Spring 中有两个 RowCallbackHandler 实现类，分别介绍如下。

- RowCountCallbackHandler：计算结果集行数。请看下面的代码：

```
RowCountCallbackHandler countCallback = new RowCountCallbackHandler();
jdbcTemplate.query("select * from user", countCallback);
int rowCount = countCallback.getRowCount();
```

可见，RowCountCallbackHandler 包含了一个记录结果集行数的“状态”。在多线程环境下，如果没有进行特殊的处理，就不能在多个地方复用 countCallback 实例。

- SimpleRowCountCallbackHandler：简单地遍历结果集，不作任何处理，在测试时使用。

Spring 也提供了几个 RowMapper 实现类，如 ColumnMapRowMapper 和 SingleColumnRowMapper。ColumnMapRowMapper 将结果集中的每一行映射为一个 Map，而

SingleColumnRowMapper 则将结果集中的某一列映射为一个 Object。它们都只定义了映射逻辑，而没有保持状态。

我们知道，通过 JDBC 查询返回一个 ResultSet 结果集时，JDBC 并不会一次性将所有匹配的数据都加载到 JVM 中，而是只返回一批次的数据（由 JDBC 驱动程序决定，如 Oracle 的 JDBC 驱动默认返回 10 行），当通过 ResultSet#next()游标滚动结果集超过数据范围时，JDBC 再获取一批数据。这样以一种“批量化+串行化”的处理方式避免大结果集处理时 JVM 内存的过大开销。

当处理大结果集时，如果使用 RowMapper，那么**采用的方式是将结果集中的所有数据都放到一个 List<T>对象中，这样将会占用大量的 JVM 内存**，甚至可能直接引发 OutOfMemoryException 异常。这时，可使用 RowCallbackHandler 接口，在 processRow()接口方法内部一边获取数据一边完成处理，这样数据就不会在内存中堆积，可大大减少对 JVM 内存的占用。

举例来说，如果程序要求给所有系统用户发送一封邮件，而系统用户数量为 100 万。一种方案是采用 RowMapper，返回一个 List<User>的集合，再通过遍历这个 List<User>，逐个发送邮件；而另一种方案是采用 RowCallbackHandler 接口，在 processRow()接口方法内部逐行获取 User 数据后，立即调用邮件服务发送邮件。虽然这两种方案都达到了相同的目的，但第一种方案会在程序运行过程中，在 JVM 中产生一个系统用户数大小为 100 万条的 List<User>，从而导致极低的系统性能和巨大的内存开销，甚至引起系统崩溃。

采用 RowMapper 的操作方式是先获取数据，然后再处理数据；而 RowCallbackHandler 的操作方式是一边获取数据一边处理，处理完就丢弃之。因此，可以将 RowMapper 看作采用批量化数据处理策略，而 RowCallbackHandler 则采用流化处理策略。

轻松一刻

三国时期，徐庶推荐庞统去辅佐刘备。刘备见庞统貌丑，就派他当耒阳县令。庞统感觉大材小用很不爽，终日醉酒，三月不理政事。百姓状告到刘备那里，刘备立即派张飞去耒阳考察实情。张飞一进县衙，便斥问庞统：“你整天醉酒，哪能不误政事?”庞统说：“百里小县，有啥大事，何难决断！将军少坐，待我发落。”随即唤来公吏，将百余日所积公务都取来剖断。公吏抱来成堆案卷上厅，诉词被告人等，环跪阶下。庞统手中批判，口中发落，耳内听词，曲直分明，毫无差错。

13.2.5 查询单值数据

如果查询的结果集仅有一个值，如 SELECT COUNT(*) FROM t_forum 等，这时可以使用更简单的方式获取结果集的值。JdbcTemplate 为获取结果集中的单值数据提供了

3 组方法，分别用于获取 int、long 的单值，其他类型的单值则以 Object 类型返回。

1. int 类型的单值查询接口

- int queryForInt(String sql)。
- *int queryForInt(String sql, Object... args)*。
- int queryForInt(String sql, Object[] args, int[] argTypes)。

使用下面简单的代码就可以获取论坛版块的总数：

```
public int getForumNum() {
    String sql = "SELECT COUNT(*) FROM t_forum";
    return getJdbcTemplate().queryForInt(sql);
}
```

值得注意的是，如果返回空结果集，则将抛出 EmptyResultDataAccessException 异常；如果返回的结果集多于一行，则会抛出 IncorrectResultSizeDataAccessException 异常。

2. long 类型的单值查询接口

- long queryForLong(String sql)。
- long queryForLong(String sql, Object... args)。
- long queryForLong(String sql, Object[] args, int[] argTypes)。

当返回的整数超过 int 的值范围时，使用这套接口方法将返回长整型数值。

3. 其他类型的单值查询接口

- <T> T queryForObject(String sql, Class<T> requiredType)。
- <T> T queryForObject(String sql, Object[] args, Class<T>requiredType)。
- <T> T queryForObject(String sql, Object[] args, int[] argTypes, Class<T> requiredType)。
- <T> T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)。
- <T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)。
- <T> T queryForObject(String sql, RowMapper<T> rowMapper)。

在使用带 Class<T> requiredType 参数的方法时，结果集必须仅拥有一行一列，且结果集可以被构造成 T 类型的对象。如果结果集包括多列，则需要使用带 RowMapper<T> rowMapper 参数的方法，在接口方法中手工完成对象构造。

可以通过分析用户主题的回帖率来判断用户的被关注度，被关注度越高的用户是论坛中最有引导力的用户。通过下面的方法来实现这一要求，如代码清单 13-7 所示。

代码清单 13-7　使用RowMapper获取单值对象

```
package com.smart.dao;
  ...
public class TopicDao {
    public double getReplyRate(int userId) {
        String sql = "SELECT topic_replies,topic_views FROM t_topic WHERE user_id=?";
        double rate = jdbcTemplate.queryForObject(sql, new Object[] {userId},
                new RowMapper<Double>() {
                    public Double mapRow(ResultSet rs, int index)
```

```
                        throws SQLException {
                    int replies = rs.getInt("topic_replies");
                    int views = rs.getInt("topic_views");
                    if (views > 0)
                        return new Double((double) replies / views);
                    else
                        return new Double(0.0);
                }
            });
        return rate;
    };
}
```

上面的代码使用 Object queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)接口方法获取某一用户主题的回帖率。粗体部分定义了根据结果集数据构造 Double 型返回值的逻辑。在这里，我们把问题复杂化了，主要是为了演示接口方法的使用。实际上，可以通过 SQL 语句，在数据库中直接计算回帖率，如下：

```
String sql = "SELECT topic_replies/topic_views reply_rate FROM t_topic WHERE user_id=?"
```

实战经验

读者可能已经注意到，我们在编写 SQL 语句时，特意用大写形式编写 SQL 语言的关键字和函数，而用小写形式编写表名、字段名等非语义的元素。在 SQL 语句中用大小写区分语义和非语义的元素是比较好的编码习惯。为了防止编写过程中大小写的不断切换所造成的麻烦，用户也可以整句统一先按小写或大写的方式进行编写，然后再对部分代码进行大小写的转换。一般的 IDE 都提供了大小写转换的快捷键，如在 IDEA 中，用户可以通过 Ctrl+Shift+U 组合键对所选的代码进行大小写转换。

13.2.6 调用存储过程

JdbcTemplate 提供了两个调用存储过程的接口方法，分别介绍如下。

- <T> T execute(String callString, CallableStatementCallback<T> action)：用户通过 callString 参数指定调用存储过程的 SQL 语句；第二个参数 CallableStatement Callback<T>是一个回调接口，该接口只有一个方法 T doInCallableStatement (CallableStatement cs)，用户可以在该方法中进行输入参数绑定、输出参数注册及返回数据处理等操作。
- <T> T execute(CallableStatementCreator csc, CallableStatementCallback<T> action)：该接口方法使用 CallableStatementCreator 创建 CallableStatement，CallableStatement Creator 定义了一个方法 CallableStatement createCallableStatement(Connection con)，它使用 Connection 实例创建 CallableStatement 对象。CallableStatementCreator 负责创建 CallableStatement 实例、绑定参数、注册输出参数等工作，而

CallableStatementCallback<T>负责处理存储过程的返回结果。Spring 提供了创建 CallableStatementCreator 的工厂类 CallableStatementCreatorFactory，通过该工厂类可以简化 CallableStatementCreator 的实例创建工作。

下面在 MySQL 5.0 中创建一个存储过程，并使用 JdbcTemplate 来调用该存储过程。

```
delimiter // ①          将语句的结束符调整为//，否则存储过程中的“;”语句结束符会被错误解析
CREATE PROCEDURE P_GET_TOPIC_NUM(IN in_user_id INT,OUT out_num INT) ②   定义了一个入参和一个出参
BEGIN
   SELECT COUNT(*) INTO out_num  FROM t_topic WHERE user_id = in_user_id;
END
//
delimiter ; ③          重新将语句结束符调整为“;”
```

在 TopicDao 类中，添加一个 getUserTopicNum()方法，执行 P_GET_TOPIC_NUM 存储过程，如代码清单 13-8 所示。

代码清单 13-8　调用存储过程：1

```
public int getUserTopicNum(final int userId) {
    String sql = "{call P_GET_TOPIC_NUM(?,?)}";//①调用存储过程的SQL语句
    Integer num = jdbcTemplate.execute(sql,new CallableStatementCallback<Integer>() {
                public Integer doInCallableStatement(CallableStatement cs)
                        throws SQLException, DataAccessException {
                    cs.setInt(1, userId);//②绑定入参
                    cs.registerOutParameter(2, Types.INTEGER); //③注册输出参数
                    cs.execute();
                    return cs.getInt(2); //④获取输出参数的值
                }
            });
    return num;
}
```

使用 JDBC API 调用存储过程的标准 SQL 语句的格式如下：

```
{?= call <procedure-name>[<arg1>,<arg2>, ...]}
```

或

```
{call <procedure-name> [<arg1>,<arg2>, ...]}
```

通过占位符（?）的索引号绑定入参和注册出参。在②处声明了一个入参并绑定入参的值，在③处定义了一个整型的输出参数。如果某个参数既是入参又是出参，则采用相似的方式将该参数声明为入参后，再将其声明为出参就可以了。

调用 CallableStatement#execute()方法执行存储过程，存储过程执行后，就可以通过 CallableStatement 所提供的一套 getXxx()方法获取输出参数的值。值得注意的是，输出参数的位置索引必须和占位符（?）一致。如果存储过程返回结果集，则可以通过 CallableStatement #getResultSet()方法获取结果集对象，并按传统的方式获取结果集中的数据。

如果使用<T> T execute(CallableStatementCreator csc, CallableStatementCallback<T> action)接口，则代码清单 13-8 调整为代码清单 13-9 所示。

代码清单 13-9 调用存储过程：2

```
public int getUserTopicNum2(final int userId) {
    String sql = "{call P_GET_ TOPIC_NUM(?,?)}";

    //①通过 CallableStatementCreatorFactory 创建 CallableStatementCreator
    CallableStatementCreatorFactory fac = new CallableStatementCreatorFactory(sql);
    fac.addParameter(new SqlParameter("userId",Types.INTEGER)); ②
    fac.addParameter(new SqlOutParameter("topicNum",Types.INTEGER)); ③
    Map<String,Integer> paramsMap = new HashMap<String,Integer>();
    paramsMap.put("userId",userId); ④
    CallableStatementCreator csc = fac.newCallableStatementCreator (paramsMap);

    Integer num = jdbcTemplate.execute(csc,new CallableStatementCallback<Integer>(){
        public Integer doInCallableStatement(CallableStatement cs)
            throws SQLException, DataAccessException {
            cs.execute();
            return cs.getInt(2);
        }
    });
   return num;
}
```

通过 CallableStatementCreatorFactory 创建 CallableStatementCreator 时，一般按照以下顺序进行：

（1）以一条 SQL 语句为入参创建 CallableStatementCreatorFactory 实例。

（2）设置存储过程的入参、出参或者出入参，注意添加的顺序非常关键，必须和“？”占位符顺序一致。Spring 在 org.springframework.jdbc.core 包中定义了 SqlParameter、SqlOutParameter 和 SqlInOutParameter，分别代表入参、出参和出入参。在②处的 SqlParameter("userId",Types.INTEGER)代表一个入参，“userId”是逻辑名，以便稍后赋值，可以是任意的名称。同理，在③处定义了一个出参。在④处为②处定义的入参指定参数值。虽然不需要在后面为 SqlOutParameter 指定值，但 Spring 却没有提供一个无须指定参数名的构造函数，让人颇为不解。

（3）通过 newCallableStatementCreator(Map<String,?> map) 方法创建一个 CallableStatementCreator 实例。

13.3 BLOB/CLOB 类型数据的操作

我们已经学习了使用 JdbcTemplate 进行 CRUD（Create Retrieve Update Delete，增删查改）操作的知识，本节将进一步学习一些高级的数据库操作知识，包括获取本地数据连接进行数据库相关的操作，此外还要学习如何操作 BLOB 和 CLOB 这些 LOB 数据。

13.3.1　如何获取本地数据连接

从数据源返回的数据连接对象是本地 JDBC 对象（如 OracleConnection、SQLServer Connection）的代理类，这是因为数据源需要改变数据连接原有的行为以便施加额外的控制。如在调用 Connection#close()方法时，将数据连接返回到连接池中而非将其关闭。在某些情况下，我们希望得到被代理前的本地 JDBC 对象，如 OracleConnection 或 OracleResultSet，以便调用这些驱动程序厂商相关的 API 完成一些特殊的操作。

为了获取本地 JDBC 对象，Spring 在 org.springframework.jdbc.support.nativejdbc 包下定义了 NativeJdbcExtractor 接口并提供了实现类。NativeJdbcExtractor 接口定义了从数据源的 JDBC 对象抽取本地 JDBC 对象的方法，下面是几个重要的接口方法。

- Connection getNativeConnection(Connection con) ：获取本地 Connection 对象。
- Connection getNativeConnectionFromStatement(Statement stmt)：获取本地 Statement 对象。
- PreparedStatement getNativePreparedStatement(PreparedStatement ps)：获取本地 PreparedStatement 对象。
- ResultSet getNativeResultSet(ResultSet rs)：获取本地 ResultSet 对象。
- CallableStatement getNativeCallableStatement(CallableStatement cs)：获取本地 CallableStatement 对象。

有些简单的数据源仅对 Connection 对象进行代理，这时可以直接使用 SimpleNativeJdbc Extractor。有些数据源（如 Jakarta Commons DBCP）会对所有的 JDBC 对象进行代理，这时就需要根据具体情况选择适合的抽取器实现类。表 13-2 列出了不同数据源的本地 JDBC 对象抽取器类。

表 13-2　不同数据源的本地JDBC对象抽取类

数据源类型	本地 JDBC 对象抽取类
C3P0 数据源	org.springframework.jdbc.support.nativejdbc.C3P0NativeJdbcExtractor
DBCP 数据源	org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor
JBoss 3.2.4 及以上版本的数据源	org.springframework.jdbc.support.nativejdbc.JBossNativeJdbcExtractor
WebLogic 8.1+及以上版本的数据源	org.springframework.jdbc.support.nativejdbc.WebLogicNativeJdbcExtractor
WebSphere 5.1 及以上版本的数据源	org.springframework.jdbc.support.nativejdbc.WebSphereNativeJdbcExtractor
ObjectWeb'的 XAPool 数据源	org.springframework.jdbc.support.nativejdbc.XAPoolNativeJdbcExtractor

代码清单 13-10 中的代码将从 DBCP 数据源中获取 Oracle 的本地数据连接对象。

代码清单 13-10　获取本地数据连接

```
package com.smart.dao;
...
@Repository
public class PostDao {
    private JdbcTemplate jdbcTemplate;
```

```
    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    public void getNativeConn(){
        try {

            Connection conn =
DataSourceUtils.getConnection(getJdbcTemplate().getDataSource());①

            conn = jdbcTemplate.getNativeJdbcExtractor(). ②
                                getNativeConnection(conn);

            OracleConnection oconn = (OracleConnection) conn; ③
              ...
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

使用 DataSourceUtils 从模板类中获取连接

使用模板类的本地 JDBC 抽取器获取本地连接

这时可以强制进行类型转换了

在①处通过 DataSourceUtils 获取当前线程绑定的数据连接。为了使用线程上下文相关的事务，通过 DataSourceUtils 从数据源中获取连接是正确的做法。如果直接通过 dateSource 获取数据连接，则得到一个和当前线程上下文无关的新的数据连接实例。

JdbcTemplate 可以在配置时注入一个本地 JDBC 对象抽取器，要使代码清单 13-10 正确运行，则必须调整 JdbcTemplate 的配置，如下：

```
<bean id="nativeJdbcExtractor"  lazy-init="true"
 class="org.springframework.jdbc.support.nativejdbc.
CommonsDbcpNativeJdbcExtractor"/>①
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
    <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/> ②
</bean>
```

定义 DBCP 数据源的 JDBC 本地对象抽取器

设置抽取器

在获取 Oracle 的本地 Connection 实例后，就可以使用该对象的一些特有功能了，如使用 OracleConnection 对 LOB 对象进行操作。

13.3.2 相关的操作接口

LOB 代表大对象数据，包括 BLOB 和 CLOB 两种类型，前者用于存储大块的二进制数据，如图片数据、视频数据等（一般不宜将文件存储到数据库中，而应存储到文件服务器中）；而后者用于存储长文本数据，如论坛的帖子内容、产品的详细描述等。值得注意的是，在不同的数据库中，大对象对应的字段类型往往并不相同，如 Oracle 对应 BLOB/CLOB；MySQL 对应 BLOB/LONGTEXT；SQL Server 对应 IMAGE/TEXT。需要指出的是，有些数据库的大对象类型可以像简单类型一样访问，如 MySQL 的 LONGTEXT 的操作方式和 VARCHAR 类型一样。一般情况下，LOB 类型数据的访问方式不同于其他简单类型的数据，用户可能会以流的方式操作 LOB 类型的数据。此外，

LOB 类型数据的访问不是线程安全的，需要分配相应的数据库资源，并在操作完成后释放。另外，Oracle 非常有个性地采用非 JDBC 标准的 API 操作 LOB 数据。所有这些情况给编写 LOB 访问程序带来挑战，Spring 为此在 org.springframework.jdbc.support.lob 包中提供了帮助类，以便开发者轻松应对以上问题。

1．LobCreator

虽然 JDBC 定义了两个操作 LOB 类型的接口 java.sql.Blob 和 java.sql.Clob，但有些厂商的 JDBC 驱动程序并不支持这两个接口。为此，Spring 定义了一个独立于 java.sql.Blob/Clob 接口，以统一的方式操作各种数据库 LOB 类型数据的 LobCreator 接口。因为 LobCreator 本身只有 LOB 所对应的数据库资源，所以它不是线程安全的，一个 LobCreator 只能操作一个 LOB 数据。

为了方便在 PreparedStatement 中使用 LobCreator，用户可以直接使用 JdbcTemplate#execute(String sql,AbstractLobCreatingPreparedStatementCallback lcpsc) 方法。下面是 LobCreator 接口方法的简要说明。

- void close()：关闭会话，并释放 LOB 资源。
- void setBlobAsBinaryStream(PreparedStatement ps, int paramIndex, InputStream contentStream, int contentLength)：通过流填充 BLOB 数据。
- void setBlobAsBytes(PreparedStatement ps, int paramIndex, byte[] content)：通过二进制数据填充 BLOB 数据。
- void setClobAsAsciiStream(PreparedStatement ps, int paramIndex, InputStream asciiStream, int contentLength)：通过 ASCII 字符流填充 CLOB 数据。
- void setClobAsCharacterStream(PreparedStatement ps, int paramIndex, Reader characterStream, int contentLength)：通过 Unicode 字符流填充 CLOB 数据。
- void setClobAsString(PreparedStatement ps, int paramIndex, String content)：通过字符串填充 CLOB 数据。

2．LobHandler

LobHandler 接口为操作大二进制字段和大文本字段提供了统一访问接口，不管底层数据库究竟是以大对象的方式还是以一般数据类型的方式进行操作。此外，LobHandler 还充当 LobCreator 的工厂类。

Spring 提供 LobHandler 类主要是为了迁就 Oracle 特立独行的作风，Oracle 必须使用 OracleLobHandler 实现类，而其他的数据库使用 DefaultLobHandler 就可以了。Oracle 9i 直接使用自己的 API 操作 LOB 数据，并且不允许通过 PreparedStatement 的 setAsciiStream()、setBinaryStream()、setCharacterStream()等方法设置流数据，所以需要使用 Oracle LobHandler 调用 Oracle 本地的 JDBC 对象方法完成底层的操作。其他数据库可以统一使用 DefaultLobHandler。Oracle 10g 改正了 Oracle 9i 个性化的作风，终于“天下归一”了，所以 Oracle 10g 也可以使用 DefaultLobHandler。下面来看一下 LobHandler 接口的几个重要方法。

- InputStream getBlobAsBinaryStream(ResultSet rs, int columnIndex)：从结果集中返回 InputStream，通过 InputStream 读取 BLOB 数据。
- byte[] getBlobAsBytes(ResultSet rs, int columnIndex)：以二进制数据的方式获取结果集中的 BLOB 数据。
- InputStream getClobAsAsciiStream(ResultSet rs, int columnIndex)：从结果集中返回 InputStream，通过 InputStreamn 以 ASCII 字符流的方式读取 BLOB 数据。
- Reader getClobAsCharacterStream(ResultSet rs, int columnIndex)：从结果集中获取 Unicode 字符流 Reader，并通过 Reader 以 Unicode 字符流的方式读取 CLOB 数据。
- String getClobAsString(ResultSet rs, int columnIndex)：从结果集中以字符串的方式获取 CLOB 数据。
- LobCreator getLobCreator()：生成一个会话相关的 LobCreator 对象。

13.3.3 插入 LOB 类型的数据

用于保存论坛帖子的 t_post 表拥有两个 LOB 字段，其中 post_text 为 CLOB 类型，而 post_attach 为 BLOB 类型。下面来编写插入一个帖子记录的代码，如代码清单 13-11 所示。

代码清单 13-11 添加LOB字段数据

```
package com.smart.dao;

...
import com.smart.domain.Post;

@Repository
public class PostDao {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    //①定义 LobHandler 属性
    private LobHandler lobHandler;

    @Autowired
    public void setLobHandler(LobHandler lobHandler) {
        this.lobHandler = lobHandler;
    }
   public void addPost(final Post post){
    String sql = " INSERT INTO t_post(post_id,user_id,post_text,post_attach)"
                 + " VALUES(?,?,?,?)";
    jdbcTemplate.execute(sql,
          new AbstractLobCreatingPreparedStatementCallback(this.lobHandler) {//②
         protected void setValues(PreparedStatement ps,LobCreator lobCreator)
```

```
            throws SQLException {
                ps.setInt(1, 1);
                ps.setInt(2, post.getUserId());

                //③设置 CLOB 字段
                lobCreator.setClobAsString(ps, 3, post.getPostText());
                //④设置 BLOB 字段
                lobCreator.setBlobAsBytes(ps, 4, post.getPostAttach());
            }
        });

    };
    ...
}
```

首先在 PostDao 中引入一个 LobHandler 属性，如①处所示，并通过 Jdbc Template# execute(String sql,AbstractLobCreatingPreparedStatementCallback lcpsc)方法完成插入 LOB 数据的操作。然后通过匿名内部类的方式定义 LobCreatingPreparedStatementCallback 抽象类的子类，其构造函数需要一个 LobHandler 入参，如②处所示。最后在匿名类中实现了父类的抽象方法 setValues(PreparedStatement ps,LobCreator lobCreator)，在该方法中，通过 lobCreator 操作 LOB 对象，如③和④处所示，分别通过字符串和二进制数组填充 BLOB 和 CLOB 的数据。用户同样可以使用流的方式填充 LOB 数据，仅需调用 lobCreator 相应的流填充方法即可。

需要调整 Spring 的配置文件以配合刚刚定义的 PostDao。假设底层数据库是 Oracle，可以采用以下配置方式：

```
...
<bean id="nativeJdbcExtractor"
    class="org.springframework.jdbc.support.
nativejdbc.CommonsDbcpNativeJdbcExtractor"
    lazy-init="true"/>

<!--①使用设置本地 JDBC 对象抽取-->
<bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler"
    lazy-init="true"
    p:nativeJdbcExtractor-ref="nativeJdbcExtractor"/>
```

读者可能已经注意到 NativeJdbcExtractor 和 JobHandler Bean 都设置为 lazy-init= "true"，这是因为 NativeJdbcExtractor 需要通过运行期的反射机制获取底层的 JDBC 对象，所以需要避免让 Spring 容器启动时就实例化这两个 Bean。

LobHandler 需要访问本地 JDBC 对象，因此需要通过一个 NativeJdbcExtractor Bean 来完成任务，在①处为 JobHandler 注入了一个 NativeJdbcExtractor。

最后，可以把 lobHandler Bean 设置给需要进行 LOB 数据访问的 PostDao。

只要不是 Oracle 9i 的数据库，即 Oracle 10g 或其他数据库，则只要简单地配置一个 DefaultLobHandler 就可以了，如下：

```
<bean id="lobHandler "
class="org.springframework.jdbc.support.lob.DefaultLobHandler"
    lazy-init="true"/>
```

DefaultLobHandler 只是简单地代理标准 JDBC 的 PreparedStatement 和 ResultSet 对象，由于并不需要访问数据库驱动本地的 JDBC 对象，所以它不需要 NativeJdbcExtractor 的帮助。

用户可以通过以下代码测试 PostDao 的 addPost()方法，如代码清单 13-12 所示。

代码清单 13-12　测试PostDao的addPost()方法

```
package com.smart.dao;

...

@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
@Rollback
@Transactional
public class TestPostDao extends AbstractTransactionalTestNGSpringContextTests {

    @Autowired
    private PostDao postDao;

   @Test
    public void testAddPost() throws Throwable{
        Post post = new Post();
        post.setUserId(2);

        //①获取图片资源
        ClassPathResource res = new ClassPathResource("temp.jpg");

        //②读取图片文件的数据
        byte[] mockImg = FileCopyUtils.copyToByteArray(res.getFile());
        post.setPostAttach(mockImg);
        post.setPostText("测试帖子的内容");
        postDao.addPost(post);
    }
}
```

在这里，有几个知识点需要解释一下：我们使用 TestNG 编写测试类，通过@ContextConfiguration 加载 Spring 容器，它能够直接从 IoC 容器中装载 Bean。关于 Spring 中测试的知识，请参见第 20 章。

此外，我们使用 ClassPathResource 加载图片资源，并通过 FileCopyUtils 读取文件的数据。

13.3.4　以块数据方式读取 LOB 数据

用户可以直接以数据块的方式读取 LOB 数据：以 String 读取 CLOB 字段的数据，而以 byte[]读取 BLOB 字段的数据。在 PostDao 中添加一个 getAttachs()方法，以便获取某一用户所有带附件的帖子，代码如下：

```
public List getAttachs(final int userId){
    String sql = "SELECT post_id,post_attach FROM t_post where user_id =? and
            post_attach is not null";
```

```
    return jdbcTemplate.query(sql, new Object[] { userId },
            new RowMapper<Post>() {
                public Post mapRow(ResultSet rs, int rowNum)
                        throws SQLException {
                    int postId = rs.getInt(1);

                    //①以二进制数组方式获取 BLOB 数据
                    byte[] attach = lobHandler.getBlobAsBytes(rs, 2);
                    Post post = new Post();
                    post.setPostId(postId);
                    post.setPostAttach(attach);
                    return post;
                }
            });
}
```

通过 JdbcTemplate 的 List<T> query(String sql, Object[] args, RowMapper rowMapper)接口处理行数据的映射。在 RowMapper 回调的 mapRow()接口方法中，通过 LobHandler 以 byte[]获取 BLOB 字段的数据。

13.3.5　以流数据方式读取 LOB 数据

由于 Lob 数据的体积可能很大（如 100MB），如果直接以块的方式操作 LOB 数据，则需要消耗大量的内存，直接影响到程序的整体运行。对于体积很大的 LOB 数据，可以使用流的方式进行访问，以减少内存的占用。JdbcTemplate 为此提供了一个 Object query(String sql, Object[] args, ResultSetExtractor rse)方法，ResultSetExtractor 接口拥有一个处理流数据的抽象类 org.springframework.jdbc.core.support.AbstractLobStreaming ResultSetExtractor，可以通过扩展此类用流的方式操作 LOB 字段的数据。下面为 PostDao 添加一个以流的方式获取某个帖子附件的方法，代码如下：

```
...
public void getAttach(final int postId,final OutputStream os){①
    String sql = "SELECT post_attach FROM t_post WHERE post_id=? ";
    jdbcTemplate.query(
        sql, new Object[] {postId},
        new AbstractLobStreamingResultSetExtractor() {②
            protected void handleNoRowFound() throws LobRetrieval FailureException {③
                System.out.println("Not Found result!");
            }

            public void streamData(ResultSet rs) throws SQLException, IOException {④
                InputStream is = lobHandler.getBlobAsBinaryStream(rs, 1);
                if (is != null) {
                    FileCopyUtils.copy(is, os);
                }
            }
        }
    );
}
```

用于接收 LOB 数据的输出流

匿名内部类

处理未找到数据行的情况

以流的方式处理 LOB 字段

通过扩展 AbstractLobStreamingResultSetExtractor 抽象类，在 streamData(ResultSet rs)方法中以流的方式读取 LOB 字段数据，如④处所示。这里又用到了 Spring 的工具类 FileCopyUtils，将输入流的数据复制到输出流中。在 getAttach()方法中通过入参 OutputStream os 接收 LOB 的数据，如①处所示。用户可以同时覆盖抽象类中的 handleNoRowFound()方法，定义未找到数据行时的处理逻辑。

13.4 自增键和行集

Spring JDBC 提供了对自增键及行集的支持，自增键对象让用户可以不依赖数据库的自增键，在应用层为新记录提供主键。在 Java 1.4 中引入了 RowSet，它允许在连接断开的情况下操作数据。本节将介绍如何在 Spring JDBC 中使用 RowSet。

13.4.1 自增键的使用

一般数据库都提供了自增键的功能，如 MySQL 的 auto_increment、SQL Server 的 identity 字段等。Spring 允许用户在应用层产生主键值，为此定义了 org.springframework.jdbc.support.incrementer.DataFieldMaxValueIncrementer 接口，提供两种产生主键的方案：第一，通过序列产生主键；第二，通过表产生主键。根据主键产生方式及数据库类型的不同，Spring 提供了若干实现类，如图 13-1 所示。

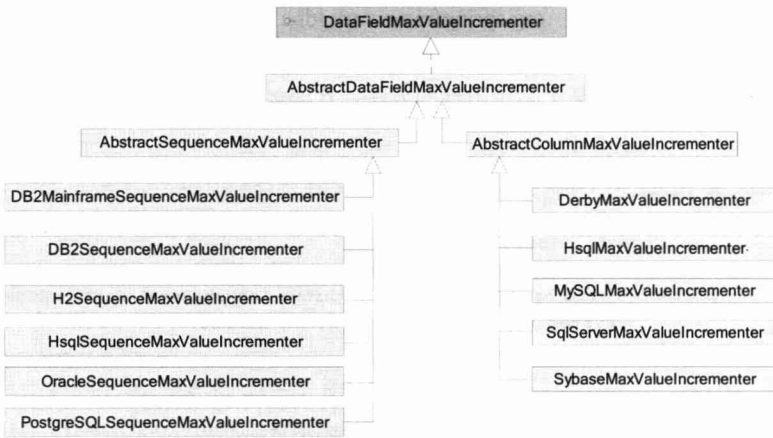


图 13-1 DateFieldMaxValueIncrementer 继承类图

根据不同的主键产生方式，可能需要配置表名、主键字段名或序列名等信息。下面以 Oracle 和 MySQL 数据库为例，分别讲解使用序列及表字段产生主键值的方式。

DataFieldMaxValueIncrementer 接口定义了 3 个获取下一个主键值的方法。

- int nextIntValue()：获取下一个主键值，主键数据类型为 int。

- long nextLongValue()：获取下一个主键值，主键数据类型为 long。
- String nextStringValue()：获取下一个主键值，主键数据类型为 String。

在其抽象实现类 AbstractDataFieldMaxValueIncrementer 中，提供了几个重要的属性，其中 incrementerName 属性用于定义序列或模拟序列表的名称；如果返回的主键值是 String 类型，则 paddingLength 属性可能会派上用场，它允许用户指定返回主键值的长度，不足的部分前面补 0。AbstractSequenceMaxValueIncrementer 类使用标准的数据库序列产生主键值，而 AbstractColumnMaxValueIncrementer 类使用一张模拟序列的表产生主键值。AbstractColumn MaxValueIncrementer 类可以通过 cacheSize 属性指定缓存的主键个数，当内存中的主键值用完后，递增器将一次性获取 cacheSize 个主键值，这样就可以减少数据访问的次数，提高应用的性能。

在代码清单 13-13 中，手工指定 t_post 表的主键值为 1，第二次运行该方法时就会抛出主键冲突的异常，现在通过 DateFieldMaxValueIncrementer 从数据库中获取主键值来弥补这个缺陷。调整 PostDao 的代码，添加 DateFieldMaxValueIncrementer 属性，并通过它从序列中得到下一个主键值。

代码清单 13-13　使用DateFieldMaxValueIncrementer产生主键

```
@Repository
public class PostDao {
    ...
    private DataFieldMaxValueIncrementer incre; ①          主键值产生器

    @Autowired
    public void setIncre(DataFieldMaxValueIncrementer incre) {
        this.incre = incre;
    }
    public void addPost(final Post post) {
        ...
        getJdbcTemplate().execute(
                sql,new AbstractLobCreatingPreparedStatementCallback(
                        this.lobHandler) {
                    protected void setValues(PreparedStatement ps,
                            LobCreator lobCreator) throws SQLException {
                        ps.setInt(1, incre.nextIntValue());②
                        ...                                    获取下一个主键值
                    }
                });
    }
    ...//省略get/setter方法
}
```

在②处，通过 incre.nextIntValue()方法获取下一个主键值。

1. 以序列方式产生主键值

在 Oracle 数据库中创建一个 seq_post_id 序列，使用这个序列为 t_post 表提供主键值。以下是创建 seq_post_id 序列的脚本：

```
create sequence seq_post_id
increment by 1
start with 1;
```

接着，调整 Spring 的配置，使用 OracleSequenceMaxValueIncrementer 作为主键产生器，代码如下：

```
<bean id="incre"
      class="org.springframework.jdbc.support.incrementer.
                  OracleSequenceMaxValueIncrementer"
      p:incrementerName="seq_post_id" ①  指定序列名
      p:dataSource-ref="dataSource"/> ②  设置数据源
```

2. 以表方式产生主键值

在 MySQL 数据库中创建一张用于维护 t_post 主键的 t_post_id 表。以下是创建该表及插入初始化的 SQL 脚本：

```
create table t_post_id(sequence_id  int) type = MYISAM;
insert into t_post_id values(0);
```

由于主键维护表的并发访问量很大，所以最好将其声明为 MYISAM 类型。此外，需要为该表提供初始值，以便后续主键值在此之上进行递增。

调整为 MySQL 数据库后，仅需对 Spring 配置文件进行小小的调整就可以了，如下：

```
<bean id="incre"
    class="org.springframework.jdbc.support.incrementer.MySQLMaxValueIncrementer"
    p:incrementerName="t_post_id"  ①  设置维护主键的表名
    p:columnName="sequence_id"     ②  用于生成主键值的列名
    p:cacheSize="10"               ③  缓存大小
    p:dataSource-ref="dataSource"/>
```

incrementerName 和 columnName 都很容易理解，cacheSize 决定一次返回的主键个数，这里设置为 10。当第一次通过 MySQLMaxValueIncrementer# nextIntValue()方法获取主键值时，MySQLMaxValueIncrementer 将使 t_post_id. sequence_id 递增 10；而后续 9 次调用 nextIntValue()方法时，都从缓存中获取主键值；直到第 10 次调用 nextIntValue()方法时，才会再次将 t_post_id. sequence_id 字段值递增 10，如此循环反复。

13.4.2 如何规划主键方案

在实际应用中必须慎重规划好主键方案，它直接影响到应用系统的运行效率、扩展性及维护性等。一个欠考虑的主键方案不但会带来并发性的问题，还可能造成系统难于编程、不易维护。

从主键创建者的角度，可以粗略地将主键创建方案分为两类：其一为“应用层主键方案”，新数据的主键分配由应用层负责，如采用 UUID 或者使用 DataFieldMax ValueIncrementer 生成主键都属于这一类型；其二为“数据库层主键方案”，新数据的主键分配由数据库负责，即在定义表结构时将主键列设置为 auto increment，或者通过表的触发器分配主键。

笔者认为，“数据库层主键方案”已经成为历史遗留的产物，它的缺点和不足已经随着应用的发展不断凸显：其一，它会给应用开发带来不便，因为必须通过一个查询获取新增数据的主键值；其二，不方便主键值的全局管理和控制，使系统丧失灵活性；其

三，不方便数据的整合和迁移。所以，除非有充足的理由，否则应当尽量避免采用“数据库层主键方案”。

在传统的应用中，数据表主键往往只承担本表内数据唯一性标识的任务。随着应用复杂性的不断提高，数据表主键开始肩负起更多的功能，包括数据库切分、资源定位等。

笔者从事的一个电力生产管理系统，对 100 多种电网设备提供信息管理，包括变电站、线路、标上塔、变压器等，每台设备对应一张表。系统各地区独立部署，但数据必须定时抽取到省公司中汇合。不但要求单个系统内设备表主键是唯一的，还必须要求全局内是唯一的，以便将各地区的设备表数据合并到一起而不产生主键冲突；否则就必须在数据整合时进行主键转换，而主键转换是极麻烦的事，会给系统带来很大的复杂度。

因此笔者当时采用了“应用层主键方案”，使用 UUID 为各设备表产生主键值，这样就可以保证设备 ID 的全局唯一性，为后期的数据整合带来便利。采用 UUID 的好处是，每台设备的 ID 都是唯一的，不但相同设备各地区的表数据合并不会有主键冲突问题，甚至不同设备的表数据合并也不会有主键冲突问题。

当然，采用 UUID 也有不好的地方，即 UUID 是一个 36 位的字符串，会占用较多的存储空间。所以，另一个候选的主键方案就是采用分段长整型编码方案，将主键编码分为 3 段：第一段代表地区；第二段代表设备类型；第三段代表设备序号，这样就可以创建一个全局唯一的整数型主键值。当然，这时我们并不能直接使用 Spring 的 DataFieldMaxValueIncrementer 完成主键创建的任务，因为 DataFieldMaxValueIncrementer 只能为一张表创建主键。但道理是相通的，我们可以创建一张包含 3 个字段的主键表，这 3 个字段分别存储地区编码、设备类型编码及当前主键最大值，编写一个类似于 DataFieldMaxValueIncrementer 的接口以获取对应地区对应设备类型的主键值。

实战经验

在高并发的系统中，如果使用基于序列表的方式创建主键值，则应该考虑两个层面的并发问题：其一是应用层获取主键的并发问题，Spring 的 DataFieldMaxValueIncrementer 实现类已经对获取主键值的代码进行了同步，因此保证了同一 JVM 内应用不会产生并发问题；其二是全局的并发问题，如果应用是集群部署的，所有集群节点都通过同一张序列表获取主键值，那么就必须对这张序列表进行乐观锁定（序列表必须添加一个版本或时间戳字段），以防止集群节点的并发问题。但是很可惜，Spring 的 DataFieldMaxValueIncrementer 实现类并没有对序列表进行乐观锁定，因此，如果应用需要集群部署，那么 Spring 的 DataFieldMaxValueIncrementer 实现类是有全局并发问题的，必须自己实现 DataFieldMaxValueIncrementer 接口，以解决全局并发的问题。

另外，DataFieldMaxValueIncrementer 接口只能为一张表提供主键，即每张表都必须配置一个单独的 DataFieldMaxValueIncrementer，因此这种方案太死板了。如果需要采用序列表创建主键的方案，笔者建议参照 DataFieldMaxValueIncrementer 接口，自行编写一个可为多张表提供主键的接口。

13.4.3 以行集返回数据

行集对象可以绑定一个数据连接并在其整个生命周期中维持该连接，在此情况下，该行集对象被称为“连接的行集”。行集对象还可以先绑定一个数据连接，获取数据后就关闭它，这种行集被称为“非连接行集”。非连接行集可以在断开连接时更改数据，然后重新绑定数据连接，并将对数据的更改同步到数据库中。

由于 java.sql.ResultSet 方法抛出的异常是 SQLException，所以 Spring 在 org.springframework.jdbc.support.rowset 包中提供了 SqlRowSet 和 SqlRowSetMetaData 接口，对 RowSet 和 RowSetMetaData 进行了封装，以便将 SQLException 转换为 Spring DAO 体系的异常。

JdbcTemplate 为了获得基于行集的结果集，提供了以下查询方法：

- SqlRowSet queryForRowSet(String sql)。
- SqlRowSet queryForRowSet(String sql, Object... args)。
- SqlRowSet queryForRowSet(String sql, Object[] args, int[] argTypes)。

下面在 TopicDao 中添加一个以 SqlRowSet 形式返回用户所有主题的方法，代码如下：

```
public SqlRowSet getTopicRowSet(int userId) {
    String sql = "SELECT topic_id,topic_title FROM t_topic WHERE user_id=?";
    return getJdbcTemplate().queryForRowSet(sql, userId);
}
```

在 TestTopicDao 测试类中，通过以下代码对返回的 SqlRowSet 进行访问：

```
@Test
public void testGetTopicRowSet(){
    SqlRowSet srs = topicDao.getTopicRowSet(1);

    //①这时，数据连接已经断开
    while (srs.next()) {
      System.out.println(srs.getString("topic_id"));
    }
}
```

在 topicDao 查询并返回 SqlRowSet 的结果集后，数据连接就已经断开了，但是结果集的数据已经保存在 SqlRowSet 中。因此，我们可以访问到 SqlRowSet 的结果集数据。值得注意的是，RowSet 会一次性装载所有的匹配数据，而不像 ResultSet 一样，分批次返回一批数据（一批的行数为 fetchSize）。

所以，对于大结果集的数据，使用 SqlRowSet 会造成很大的内存消耗，这一点要铭记。不过 JdbcTemplate 的 maxSize 属性依旧会限制 SqlRowSet 的返回记录数。

13.5 NamedParameterJdbcTemplate 模板类

除了标准的 JdbcTemplate 外，Spring 还提供了两个易用的 JDBC 模板类，分别是

NamedParameterJdbcTemplate 和 SimpleJdbcTemplate。前者提供命名参数绑定的功能；而后者封装了 JdbcTemplate，将常用的 API 开放出来。

在低版本的 Spring 中，用户只能使用“?”占位符声明参数，并使用索引号绑定参数。在使用这种方法绑定参数时必须足够小心，以保证参数的索引号和 SQL 语句中占位符（?）的位置正确匹配。这种编程模式被认为是弱稳定的，因为当新增一个“?”占位符时，可能导致原来所有的参数绑定方法都需要因此调整索引号，这极有可能引入一些不容易发现的错误。

NamedParameterJdbcTemplate 模板类支持命名参数变量的 SQL，它位于 org.springframework.jdbc.core.namedparam 包中。该包中还定义了一个用于承载命名参数的 SqlParameterSource 接口，该接口拥有两个实现类。

- BeanPropertySqlParameterSource：该实现类将一个 JavaBean 对象封装成一个参数源，以便通过 JavaBean 属性名和 SQL 语句中命名参数匹配的方式绑定参数。
- MapSqlParameterSource：该实现类内部有一个 Map 存储参数，可以通过 addValue(String paramName, Object value)或 addValues(Map values)方法添加参数，并通过参数键名和 SQL 语句中命名参数匹配的方式绑定参数。

来看一个使用 NamedParameterJdbcTemplate 的实例，如代码清单 13-14 所示。

代码清单 13-14　使用命名参数绑定的模板类

```
package com.smart.dao;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
...
@Repository
public class ForumDao {
    ...
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Autowired
    public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate
                                              namedParameterJdbcTemplate) {
        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
    }
    public void addForumByNamedParams(final Forum forum) {
        final String sql = "INSERT INTO t_forum(forum_name, forum_desc)
                VALUES(:forumName,:forumDesc)"; ①        定义参数源
        SqlParameterSource sps = new BeanPropertySqlParameterSource(forum); ②
        namedParameterJdbcTemplate.update(sql, sps); ③        定义命名参数
    }                                  使用模板类方法
    ...
}
```

在 SQL 语句中声明命名参数的格式是“:paramName”，如①处所示。在这个实例中，使用 BeanPropertySqlParameterSource 提供参数源，它接收一个 JavaBean 作为构造函数的入参，如②处所示。和标准的 JdbcTemplate 一样，NamedParameterJdbcTemplate 提供了众多数据访问方法，这些方法大都拥有一个 SqlParameterSource 入参，用以提供参数

源，这里使用模板类的 int update(String sql, SqlParameterSource paramSource)方法执行插入数据的操作，如③处所示。

必须在 applicationContext.xml 中定义 NamedParameterJdbcTemplate 的 Bean，如下：

```
<bean id="namedParamJdbcTemplate"
     class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
     <constructor-arg ref="dataSource"/>
</bean>
```

Forum 领域对象拥有 3 个属性，其中 forumName 和 forumDesc 这两个属性和 SQL 语句中的命名参数匹配，参数即按照这个匹配关系进行绑定。为了更清楚地看出参数绑定规则，下面给出 Forum 领域对象的代码：

```
package com.smart.domain;
...
public class Forum implements Serializable{
    private int forumId;
    private String forumName;
    private String forumDesc;
   //省略get/setter
}
```

如果数据表记录没有对应的领域对象，则用户可以直接使用 MapSqlParameterSource 达到参数绑定的目的。可以按照如下方式使用 MapSqlParameterSource：

```
public void addForum(final Forum forum) {
    final String sql = "INSERT INTO t_forum(forum_name,forum_desc)
                        VALUES(:forumName,:forumDesc)";
    MapSqlParameterSource sps = new MapSqlParameterSource(). ①
                        addValue("forumName", forum.getForumName()).
                        addValue("forumDesc",forum.getForumDesc());
    namedParameterJdbcTemplate.update(sql,sps);
}
```

提供参数源

由于 MapSqlParameterSource 中的大多数方法能够返回对象本身，所以可以将几个方法的调用串成一个链，如①处所示。由于这个原因，使用方法调用链模式设计的 API 受到越来越多开发者的青睐。

值得一提的是，NamedParameterJdbcTemplate 可支持 in，而 PrepareStatement 对 in 的支持只能动态拼接。如下例假设想找 id 在 1,2,3,4 范围内的数据：

```
String in_data="1,2,3,4,5,6";
pst = conn.prepareStatement("select id,name from B where id in ( ? )");
pst.setString(1,in_data);
```

使用传统的 PrepareStatement 是动态设定参数的，也就是现在生成 select id,name from B where id in (?)，一个“?”代表一个参数，pst.setString(1,"1,2,3,4")就相当于 select id,name from B where id in ('1,2,3,4')。我们期望找到 id 在 1,2,3,4 中的数据，而这时候生成的 SQL 语句是取查找 id 为 1,2,3,4 的记录。使用 NamedParameterJdbcTemplate 可以很好地解决这个问题，如下例：

```
String sql = "select id, name  from B where id in (:ids )";
MapSqlParameterSource parameters = new MapSqlParameterSource();
parameters.addValue("ids", ids);
```

这时候使用 NamedParameterJdbcTemplate 就相当于 select id,name from B where id in ('1','2','3','4')，符合我们的期望，而且不需要拼接 SQL 语句。

13.6　小结

本章以 JdbcTemplate 模板类为中心，详细讲解了各种使用 JdbcTemplate 对数据进行 CRUD 操作的方法。JdbcTemplate 使用大量的回调接口完成数据的访问操作，StatementCallback、PreparedStatementCallback、CallableStatementCallback、BatchPrepared StatementSetter 及 RowMapper 是其中常用的回调接口，一般可以通过匿名内部类实现这些回调接口，使代码更加紧凑。

此外，还重点讲述了在 Spring JDBC 中操作 LOB 数据类型的方法。由于有些数据库使用了自己的 API 操作 LOB 数据，因此需要从数据源的代理 JDBC 对象中抽取本地数据库的 JDBC 对象。LOB 数据可以使用块数据和流数据的方式进行访问。如果 LOB 数据比较大，则通过流操作方式可以减少内存的占用，同时保证程序的性能。

在本章中，我们花费了较大的篇幅讲述主键的知识，主键的产生方式从产生地点上可以分为应用层产生和数据库产生两种。应用层借助数据库的序列或表产生主键值，这种方式可以保证程序的可移植性和安全性，同时可以通过缓存机制提高运行效率。有些数据库支持数据表自增主键值产生机制，在 JDBC 3.0 以前的版本中，无法通过 Statement 自动获取新增记录对应的主键值。这时需要在插入数据后，立即执行一条数据库相关的主键获取 SQL 语句以得到对应的主键值。在数据库高并发的情况下，有可能获取到不正确的主键值。在这种情况下，在插入数据前在应用层准备好主键值是很好的备选方案。

Spring 持续降低 Spring JDBC 的使用难度，它提供支持命名参数绑定的 NamedParameterJdbcTemplate，以减少编程的出错概率。

第 14 章

整合其他 ORM 框架

罗素在《西方哲学史》中有一句经典的话：参差多态乃是幸福的本源。按照这样的说法，Spring 开发者是幸福的，因为在 Spring 中不但可以选择 Spring JDBC 作为持久化技术，还可以选择 Hibernate、JPA、JDO 等多种类型的持久化技术。由于篇幅所限，与其大同小异地介绍 Spring 支持的所有 ORM，不如选择其中两个最具代表性的（Hibernate 和 MyBatis）进行重点学习。同时，读者还将学习到 ORM 框架的混用和 DAO 层设计的知识。

本章主要内容：

- Spring 对 ORM 框架整合的基本思想
- 在 Spring 中使用 Hibernate
- 在 Spring 中使用 MyBatis
- 如何在 Spring 封装的基础上进行 DAO 层的基础设计

本章亮点：

在不同的 ORM 框架中处理 LOB 数据

14.1 Spring 整合 ORM 技术

Spring 的开放性和扩展性在持久化技术领域得到了充分的证明，Spring 不但直接提供了 Spring JDBC，还整合了 JPA、Hibernate、JDO 等 ORM 领域的代表者，使用者完全可以根据需要做出适合自己的选择。

第 13 章详细讲解了 Spring 对 JDBC API 的封装，其中大部分的思路和整合其他 ORM 框架是一脉相通的：在资源管理、DAO 模板类、事务管理、DAO 异常体系等方面，Spring 始终保持整合方式上的统一。在 Spring 中掌握了一种持久化技术后，切换到另一种持久化技术并不需要花费什么代价。

Spring 在集成 ORM 框架时，始终走“民主”、“亲民”的路线。它不仅提供了方便的模板类对原 ORM 进行简化封装，以一种更具 Spring 的风格使用 ORM 技术；同时，在保证继续享受 Spring 通用功能的情况下，开发者还可以使用 ORM 框架原生的 API 编写程序。这让一直使用 ORM 原生 API 编程的开发者感到很自然，不会存在过渡上的障碍。

在使用某种 ORM 框架时，为了让其更适合项目的需要，降低使用难度，一般情况下都会在原有 ORM API 的基础上编写一套封装类。Spring 高屋建瓴地为我们做了类似的工作，所以在决定花费力气并冒着风险去构造类似的底层架构之前，最好事先参考一下 Spring 的解决方案，而不要急着另起炉灶。

使用 Spring 所提供的 ORM 整合方案，可以获得许多好处。

1．方便基础设施的搭建

不同的 ORM 技术都有一套自己的方案以初始化框架、搭建基础设施等。在搭建基础设施的过程中，数据源是不可或缺的，不同 ORM 框架的实现方式各不相同，如在 JPA 中通过 persistence.xml 定义数据源，而 Hibernate 在 hibernate.cfg.xml 中配置数据源，目的相同但方法迥异。在数据源的基础上，不同的 ORM 框架拥有自己的基础实例，如 Hibernate 的 SessionFactory，它们是 ORM 程序运行时的底层设施，在程序级别代表着 ORM 框架本身。在学习不同的 ORM 框架知识时，往往会有一个独立的篇章专门讲解如何初始化这些实例。在 Spring 中，对于不同的 ORM 框架，首先，始终可以采用相同的方式配置数据源；其次，Spring 为不同的 ORM 框架提供了相应的 FactoryBean，用以初始化 ORM 框架的基础设施，可以将它们当成普通的 Bean 对待，唯一的差别是属性不同。

2．异常封装

Spring 能够转换各种 ORM 框架所抛出的异常，将 ORM 框架专有的或检查型异常转换为 Spring DAO 异常体系中的标准异常。这样用户就可以有选择地在适当的地方处理感兴趣的异常，忽略大部分不可恢复的异常，从而避免了很多令人讨厌的异常声明和异常捕捉代码。

3．统一的事务管理

通过使用基于 Spring DAO 模板的编程风格，甚至使用 ORM 框架原生的 API，只要遵循 Spring 所提出的很少的编程要求，就可以使用 Spring 提供的事务管理功能。Spring 为不同的 ORM 框架提供了对应的事务管理器，可以采用声明的方式进行事务管理，并且可以透明地实现本地事务管理到全局 JTA 事务管理的切换。此外，作为一项附加功能，JDBC 代码能够在事务级别上与用户使用的 ORM 框架一起使用。这一功能对于诸如批量处理、LOB 操作等并不适合单独采用 ORM 完成的地方尤其有用。

4．允许混合使用多个 ORM 框架

由于 Spring 在 DAO 异常、事务、资源等高级层次建立了抽象，因而可以让业务层

对 DAO 具体实现的技术不敏感。这给开发者带来了一个好处：他们可以在底层选用适合的实现方式，甚至可以混合使用多种 ORM，一般地，CRUD 使用 Hibernate，而数据查询使用 MyBatis 或 Spring JDBC，博采众长，强强联合。同时，用户也可以使用这种特性整合一些遗留的代码，而不必一切从头开始。

5．方便单元测试

Spring 容器使得替换不同的实现和配置变得非常简单，这些内容包括 Hibernate SessionFactory 的位置、JDBC DataSource、事务管理器及映射对象的实现（如果需要）等，这样就很容易隔离并测试不同的 DAO 类。

14.2 在 Spring 中使用 Hibernate

Hibernate 在 ORM 领域具有广泛的影响，拥有广大的使用群体。它提供了 ORM 最完整、最丰富的实现，在 Spring 4.0 中目前全面支持 Hibernate 5.0，不再支持 Hibernate 3.6 之前的版本。因为 iBatis 的升级版 MyBatis 自身已经提供了对 Spring 整合的支持，所以 Spring 不再为 MyBatis 提供直接支持，大家直接使用 MyBatis 自带的整合功能即可。

14.2.1 配置 SessionFactory

使用 Hibernate 框架的首要工作是编写 Hibernate 的配置文件，其次是如何使用这些配置文件实例化 SessionFactory，创建 Hibernate 的基础设施。

Spring 为创建 SessionFactory 提供了一个好用的 FactoryBean 工厂类：org.springframework.orm.hibernateX.LocalSessionFactoryBean，通过配置一些必要的属性，就可以获取一个 SessionFactory Bean。

LocalSessionFactoryBean 配置灵活度很高，支持开发者的不同习惯，让开发者拥有充分的选择权——这是 Spring 一贯的风格。

1．零过渡障碍的配置方式

让我们回忆一下使用 Hibernate API 创建一个 SessionFactory 的过程：首先编写好对象关系的映射文件 xxx.hbm.xml；然后通过 Hibernate 的配置文件 hibernate.cfg.xml 将所有的 xxx.hbm.xml 映射文件组装起来；最后通过以下两行经典的代码得到 SessionFactory 的实例：

```
Configuration cfg = new Configuration().configure("hibernate.cfg.xml");
SessionFactory sessionFactory = cfg.buildSessionFactory();
```

hibernate.cfg.xml 配置文件拥有创建 Hibernate 基础设施所需的配置信息，来看一个最简单的 Hibernate 配置文件，如代码清单 14-1 所示。

代码清单 14-1　hibernate.cfg.xml：Hibernate配置文件

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-5.0.dtd">
<hibernate-configuration>
   <session-factory>
         <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
         <property name="connection.url">jdbc:mysql://localhost:3309/sampledb
</property>
         <property name="connection.username">root</property>
         <property name="connection.password">1234</property>
         <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
         <property name="show_sql">true</property>
         <property name="format_sql">true</property>
         <property name="current_session_context_class">thread</property>
         <mapping resource="com/smart/orm/domain/Forum.hbm.xml"/>
         <mapping resource="com/smart/orm/domain/Topic.hbm.xml"/>
         <mapping resource="com/smart/orm/domain/Post.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

这个配置文件定义了 3 个方面的信息：数据源、映射文件及 Hibernate 控制属性。既然在 Hibernate 中可以使用一个配置文件创建一个 SessionFactory 实例，在 Spring 中也可以顺理成章地通过指定一个 Hibernate 配置文件，利用 LocalSessionFactoryBean 来达到相同的目的。

```
<bean id="sessionFactory"
     class= "org.springframework.orm. hibernate5.LocalSessionFactoryBean"
     p:configLocation="classpath: hibernate.cfg.xml"/>①
```

如①处所示，通过 configLocation 属性指定了一个 Hibernate 配置文件。如果有多个 Hibernate 配置文件，则可以通过 configLocations 属性指定，多个文件之间用逗号分隔。

LocalSessionFactoryBean 将利用 Hibernate 配置文件创建一个 SessionFactory 代理对象，以便和 Spring 的事务管理机制配合工作：当数据访问代码使用 SessionFactory 时，可以获取线程绑定的 Session，不管工作在本地或全局的事务，都能正确参与到当前的 Spring 事务管理中去。

2．更具 Spring 风格的配置

Spring 对 ORM 技术的一个重要支持就是提供统一的数据源管理机制，也许更多的开发者更愿意使用 Spring 配置数据源，即在 Spring 容器中定义数据源、指定映射文件、设置 Hibernate 控制属性等信息，完成集成组装的工作，完全抛开 hibernate.cfg.xml 配置文件，如代码清单 14-2 所示。

代码清单 14-2　移除Hibernate配置文件的配置

```
<?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:p="http://www.springframework.org/schema/p"
```

```
    xmlns:context="http://www.springframework.org/schema/context"
    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-4.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-4.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

   <context:property-placeholder location="classpath:jdbc.properties" />
   <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"/>
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
        p:dataSource-ref="dataSource">①←数据源

       <!--②指定 Hibernate 实体类映射文件-->
       <property name="mappingLocations">
            <list>
                <value>classpath*:/com/smart/orm/domain/Forum.hbm.xml</value>
                <value>classpath*:/com/smart/orm/domain/Topic.hbm.xml</value>
                <value>classpath*:/com/smart/orm/domain/Post.hbm.xml</value>
            </list>
        </property>

      <!--③指定Hibernate配置属性-->
      <property name="hibernateProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.MySQLDialect
            </prop>
            <prop key="hibernate.show_sql">
               true
            </prop>
         </props>
      </property>
    </bean>
   ...
</bean>
```

数据源、映射文件及 Hibernate 控制属性这三方面的信息在 LocalSessionFactoryBean 中得到了完美集成，完全替代了 hibernate.cfg.xml 的作用，但这种配置对于 Spring 开发者而言更加亲切。

首先，①处指定的数据源是 Spring 容器中的数据源，不管是直接在 Spring 容器中配置，还是通过<jee:jndi-lookup>从 EJB 容器中获取，对引用者而言是完全透明的。

其次，凭借 Spring 资源处理的强大功能，指定 Hibernate 映射文件变得相当灵活。

在②处采用了逐个指定映射文件的方法，其实这个方法是最笨拙的。由于 mappingLocations 属性的类型是 Resource[]，因此它还支持以下简洁的配置方式：

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    p:dataSource-ref="dataSource"
    p:mappingLocations="classpath:/com/smart/orm/domain/**/*.hbm.xml">
  ...
</bean>
```

除了 mappingLocations 属性外，也可以选择通过其他资源属性指定 Hibernate 映射文件。

- mappingJarLocations：如果映射文件位于 JAR 文件中，则可以通过该属性指定，如 WEB-INF/lib/example.hbm.jar。
- p:mappingDirectoryLocations：可以指定多个放置 Hibernate 映射文件的目录，Spring 将加载这些目录下的所有 Hibernate 映射文件，如 classpath:/com/smart/orm/domain。
- mappingResources：通过相对于类路径的方式指定映射文件，属性类型为 String[]。既可以通过内嵌多个<value>的<list>元素指定映射文件，也可以通过逗号分隔的方式指定，如“topic.hbm.xml,post.hbm.xml”。

其他 Hibernate 属性通过键值对的方式提供，如③处所示，键的名称请参考 Hibernate 的说明文档。

由于这种配置方式将所有的配置信息都统一到 Spring 中，给管理、维护和调整工作带来了方便，因而被广泛接受。

14.2.2　使用 HibernateTemplate

基于模板类使用 Hibernate 是最简单的方式，它可以在不牺牲 Hibernate 强大功能的前提下，以一种更简洁的方式使用 Hibernate，极大地降低了 Hibernate 的使用难度。按照 Spring 的风格，它提供了使用模板的支持类 HibernateDaoSupport，并通过 getHibernateTemplate()方法向子类开放模板类实例的调用。

为了能够使用注解配置的功能，我们自己编写一个 BaseDao，如下：

```
package com.smart.orm.dao.hibernate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate5.HibernateTemplate;
public class BaseDao {
    @Autowired
    private HibernateTemplate hibernateTemplate;

    ...
}
```

然后通过扩展这个 BaseDao 基类创建一个使用 HibernateTemplate 的 Dao，如代码清单 14-3 所示。

代码清单 14-3　使用HibernateTemplate的DAO

```
package com.smart.orm.dao.hibernate;

import java.sql.SQLException;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.springframework.orm.hibernate5.HibernateCallback;
import org.springframework.stereotype.Repository;
import com.smart.orm.domain.Forum;

@Repository
public class ForumHibernateDao extends BaseDao{

    public void addForum(Forum forum) {
        getHibernateTemplate().save(forum);//①保存实体对象
    }

    public void updateForum(Forum forum) {
        getHibernateTemplate().update(forum); //②更改实体对象
    }

    public Forum getForum(int forumId) {
        return getHibernateTemplate().get(Forum.class, forumId); //③获取实体对象
    }

    public List<Forum> findForumByName(String forumName) {//④使用HQL查询
        return (List<Forum>)getHibernateTemplate().find(
                "from Forum f where f.forumName like ?",forumName+"%");
    }
    public long getForumNum() {//⑤使用Iterate返回结果
        Object obj =getHibernateTemplate()
                    .iterate("select count(f.forumId) from Forum f")
                    .next();
       return (Long)obj;
   }
}
```

HibernateTemplate 代理了 Hibernate Session 的大多数持久化操作，并以一种更简洁的方式提供调用。HibernateTemplate 所提供的大部分方法对于 Hibernate 开发者来说都是熟悉亲切的，因为模板类的方法大都可以在 Session 接口中找到镜像。

1. 常用的 API 方法

下面来了解一下 HibernateTemplate 开放的一些有代表性的 API 方法，其他更多的方法请参见 Spring 的 Javadoc 文档。

- Serializable save(Object entity)：保存实体对象，并返回主键值。还有一个和该方法功能类似的 void persist(Object entity)方法，后者是 JSR-220 规定的方法。
- void update(Object entity)：更新实体对象。
- void saveOrUpdate(Object entity)：保存或更新一个实体。还有一个和该方法功能类似的<T> T merge(T entity)方法，后者是 JSR-220 规定的方法。

- void delete(Object entity)：删除一个实体。
- List find(String queryString)：根据 HQL 查询实体。该方法拥有几个带参的重载版本，如 find(String queryString, Object value)、List find(String queryString, Object... values)。在内部，模板类会自动创建 Query 并执行查询。
- List findByNamedQuery(String queryName)：执行命名查询。其也拥有多个可绑定参数的重载版本。
- List findByCriteria(DetachedCriteria criteria)：Criteria 版本的查询。该方法有一个可限定范围的重载版本：List findByCriteria(org.hibernate.criterion.DetachedCriteriacriteria, int firstResult, int maxResults)。

2．使用回调接口

一般情况下，使用模板类的简单代理方法就可以满足要求了，如果希望使用更多 Hibernate 底层的功能，则可以使用回调接口。Spring 定义了一个回调接口 org.springframework.orm.hibernate5.HibernateCallback<T>，该接口拥有唯一的方法，如下：

```
T doInHibernate(org.hibernate.Session session) throws HibernateException,SQLException
```

该接口配合 HibernateTemplate 进行工作，它无须关心 Hibernate Session 的打开/关闭等操作，仅需定义数据访问逻辑即可。可以通过该接口返回结果，结果可以是一个实体对象或一个实体对象的 List。回调接口中抛出的异常将传播到模板类中并被转换成 Spring DAO 异常体系的对应类。

HibernateTemplate 定义了两个使用 HibernateCallback 回调接口的方法。

- <T> T execute(HibernateCallback<T> action)：一般使用该方法执行数据更新、新增等操作。
- List executeFind(HibernateCallback<?> action)：一般使用该方法执行数据查询操作，返回的结果是一个 List。

使用回调接口对代码清单 14-3 中的 getForumNum()方法提供另一个版本的实现，如下：

```
public long getForumNum2() {
    Long forumNum = getHibernateTemplate().execute(
        new HibernateCallback<Long>() {
            public Long doInHibernate(Session session)
                    throws HibernateException, SQLException {
                Object obj = session
                 .createQuery("select count(f.forumId) from Forum f")
                   .list()
                   .iterator()
                     .next();
                return (Long) obj;
            }
        });
    return forumNum;
}
```

代码中粗体部分以匿名类的方式提供了 HibernateCallback 回调的实现，直接使用 Hibernate Session 接口完成查询的工作并返回结果。

3. 在 Spring 中配置 DAO

在编写好基于 HibernateTemplate 的 DAO 类后，接下来要做的就是在 Spring 中进行具体配置，使该 DAO 生效，代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    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-4.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-4.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

    <!--①扫描类包以启动注解驱动的Bean-->
    <context:component-scan base-package="com.smart"/>
    <context:property-placeholder location="classpath:jdbc.properties" />
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"/>

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
        p:dataSource-ref="dataSource"
        p:mappingDirectoryLocations="classpath:/com/smart/orm/domain"
        >
      <property name="hibernateProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.MySQLDialect
            </prop>
            <prop key="hibernate.show_sql">
               true
            </prop>
         </props>
      </property>
    </bean>

   <!--②配置HibernateTemplate Bean-->
   <bean id="hibernateTemplate"
       class="org.springframework.orm.hibernate5.HibernateTemplate"
```

```
        p:sessionFactory-ref="sessionFactory"/>

    <!--③配置Hibernate的事务管理器-->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager"
        p:sessionFactory-ref="sessionFactory" />

    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

在 Spring 配置文件中，首先需要配置一个 HibernateTemplate Bean，它基于 SessionFactory 工作，如②处所示。在①处使用<context:component-scan>扫描特定的类包以启用注解驱动的 Bean，这样 ForumHibernateDao 类的@Repository 及@Autowired 注解就可以起作用，将 ForumHibernateDao 装配为 Spring 容器中的 Bean。

14.2.3　处理 LOB 类型的数据

对 LOB 类型的数据的处理始终是各种 ORM 框架比较头疼的事，需要认真对待。Hibernate 为处理特殊数据类型字段定义了一个接口：org.hibernate.usertype.UserType。Spring 在 org.springframework.orm.hibernate5.support 包中为 BLOB 和 CLOB 类型提供了几个 UserType 的实现类。因此，可以在 Hibernate 的映射文件中直接使用这两个实现类轻松处理 LOB 类型的数据。

- BlobByteArrayType：将 BLOB 数据映射为 byte[]类型的属性。
- BlobStringType：将 BLOB 数据映射为 String 类型的属性。
- BlobSerializableType：将 BLOB 数据映射为 Serializable 类型的属性。
- ClobStringType：将 CLOB 数据映射为 String 类型的属性。

前面使用的 Post 领域对象有两个分别对应 CLOB 和 BLOB 字段类型的属性，下面使用 Spring 的 UserType 为 Post 配置 Hibernate 的映射文件，如代码清单 14-4 所示。

代码清单 14-4　post.hbm.xml：LOB数据映射配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-5.0.dtd">
<hibernate-mapping auto-import="true" default-lazy="false">
    <class name="com.smart.orm.domain.Post" table="t_post">
        <id name="postId" column="post_id">
            <generator class="identity" />
        </id>
        <property name="userId" column="user_id"/>
        <property name="postText" column="post_text"  ①
        type="org.springframework.orm.hibernate5.support.ClobStringType"/>

        <property name="postAttach" column="post_attach" ②
         type="org.springframework.orm.hibernate5.support.BlobByteArrayType"/>
        <property name="postTime" column="post_time" type="date" />
        <many-to-one name="topic" column="topic_id" class="com.smart.orm.domain.Topic" />
```

对应 CLOB 字段

对应 BLOB 字段

```
    </class>
</hibernate-mapping>
```

postText 为 String 类型的属性，对应数据库的 CLOB 类型；而 postAttach 为 byte[] 类型的属性，对应数据库的 BLOB 类型。分别使用 Spring 所提供的相应 UserType 实现类进行配置，如①和②处所示。

在配置好映射文件后，还需要在 Spring 配置文件中定义 LOB 数据处理器，让 SessionFactory 拥有处理 LOB 数据的能力。

```
<bean id="lobHandler"
      class="org.springframework.jdbc.support.lob.DefaultLobHandler"
      lazy-init="true" />

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    p:dataSource-ref="dataSource"
    p:mappingDirectoryLocations="classpath:/com/smart/orm/domain"
    p:lobHandler-ref="lobHandler">①
    ...
</bean>
```

设置 LOB 数据处理器

这样，仅需简单地使用 HibernateTemplate#save(Object entity)等方法，就可以正确地保存 LOB 数据。如果是 Oracle 9i 数据库，还需要配置一个本地 JDBC 抽取器，并使用特定的 LobHandler 实现类。

在使用 LobHandler 操作 LOB 数据时，需要在事务环境下才能工作，所以必须事先配置事务管理器，否则会抛出异常。

提示

在使用 Spring JDBC 时，除了可以按 byte[]、String 类型处理 LOB 数据外，还可以使用流的方式操作 LOB 数据，当数据超过一定的大小时（比如 100MB），流操作是唯一可行的方式。可惜，Spring 并未提供以流方式操作 LOB 数据的 UserType（Spring 开发组成员认为在实现上存在难度）。不过，www.atlassian.com 替 Spring 完成了这件难事，用户可以通过以下地址了解到这个满足要求的 BlobInputStream Type：

http://docs.atlassian.com/atlassian-confluence/latest/org/springframework/orm/hibernate/support/BlobInputStreamType.html。

14.2.4 添加 Hibernate 事件监听器

Hibernate 设计了一个功能完备的事件模型，允许为事件装配一个或多个事件监听器。通过 Configuration#setListener(String type, Object listener)等方法向 Hibernate 框架注册事件监听器。Hibernate 在 org.hibernate.event 包中定义了事件及对应的事件监听器接口，并在 org.hibernate.event.def 包中提供了事件监听器接口的默认实现。

LocalSessionFactoryBean 允许用户通过 eventListeners 属性向 Hibernate 注册事件监听

器。Spring 本身就提供了一个 Hibernate 的事件监听器 IdTransferringMergeEventListener。下面通过简单的配置将其注册到 Hibernate 中。

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    ...
    <property name="eventListeners">①
        <map>
            <entry key="merge">②-1 ← 事件监听器类型
                <bean class="org.springframework.orm.hibernate5.support.
                    IdTransferringMergeEventListener" />②-2 ← 事件监听器实现类
            </entry>
        </map>
    </property>
</bean>
```

eventListeners 属性是 Map 类型的，它以键值对的方式接收事件监听器的注册信息，其中键为事件类型，而值为事件监听器实现类。事件类型必须是 Hibernate 预定义的类型，包括 auto-flush、merge、create、delete、dirty-check、evict、flush、flush-entity、load、load-collection、lock、refresh、replicate 和 save-update 等。注册监听器时需要指定事件类型，在这一点上，Hibernate 做得实在不够友好。它完全可以利用类反射机制，根据监听器实现类自动判断对应的事件类型，就像在 web.xml 中注册 Servlet 容器事件监听器一样，但 Hibernate 却要求必须显式指定事件的名称。

IdTransferringMergeEventListener 是 Spring 提供的一个 Hibernate 事件监听器，它必须和 HibernateTemplate#merge(Object entity)方法配合使用。因为 Hibernate 的 merge()方法在对一个新对象进行操作时，并不会将 ID 值传递给原对象，Spring 通过该事件监听器完成这项工作。

14.2.5　使用原生的 Hibernate API

Hibernate 3.0 引入了一个新的特性：通过 SessionFactory#getCurrentSession()方法能够获取和当前线程绑定的 Session。这一特性使得 Hibernate 自身具备了获取和事务线程绑定的 Session 对象的功能，这与在 Spring 的 HibernateTemplate 中使用和事务绑定的 Session 是相同。

因此，可以在 Spring 中使用原生的 Hibernate API 编写 DAO，它同样可以正确地和 Spring 的事务管理器一起工作。下面使用原生的 Hibernate API 实现 ForumDao 接口，如代码清单 14-5 所示。

代码清单 14-5　ForumHibernateDao：基于原生Hibernate API的实现

```
package com.smart.orm.dao.hibraw;
...
@Repository
public class ForumHibernateDao {
```

```
    //①直接注入Hibernate原生的SessionFactory对象
    @Autowired
     private SessionFactory sessionFactory;
     public void addForum(Forum forum) {
         sessionFactory.getCurrentSession().save(forum);
     }
     public void updateForum(Forum forum) {
         sessionFactory.getCurrentSession().update(forum);
     }
     ...
}
```

DAO 注入了一个 SessionFactory 对象，如①处所示。这样，DAO 中的所有数据访问方法都可以通过 SessionFactory#getCurrentSession()方法获取和当前线程绑定的 Session，以便 Spring 的事务管理器能够正确地工作。

和使用 HibernateTemplate 不同的是，使用原生 Hibernate API 所抛出的异常是 Hibernate 异常（也是运行期异常）。这意味着 DAO 的调用者只能以普通的错误来处理这些异常，而无法在声明式事务中使用通用的 Spring DAO 异常体系进行回滚设置。一般情况下，这种差异是可以接受的，并不会带来什么问题。

14.2.6 使用注解配置

和 Spring 类似，Hibernate 不但可以使用 XML 提供 ORM 的配置信息，也可以直接在领域对象类中通过注解定义 ORM 映射信息。Hibernate 不但自己定义了一套注解，还支持 JSR-220 的 JPA 注解。

下面使用注解对 Forum 进行 ORM 的配置，如代码清单 14-6 所示。

代码清单 14-6 使用JPA注解的Forum

```
package com.smart.orm.domain;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="T_FORUM")
public class Forum implements Serializable{

    @Id
    @Column(name = "FORUM_ID")
    private int forumId;

    @Column(name = "FORUM_NAME")
    private String forumName;

    @Column(name = "FORUM_DESC")
    private String forumDesc;
```

```
    public String getForumDesc() {
        return forumDesc;
    }
}
```

Hibernate 通过 AnnotationConfiguration 的 addAnnotatedClass()或 addPackage()方法加载使用 JPA 注解的实体类，获取映射的元数据信息，并在此基础上创建 SessionFactory 实例。

需要特别注意的是，使用 addPackage 并不是加载类包下所有标注了 ORM 注解的实体类，而是加载类包下 package-info.java 文件中定义的 Annotation，而该类包下的所有持久化类仍然需要通过 addAnnotatedClass()方法加载。

Spring 专门提供了一个配套的 AnnotationSessionFactoryBean，用于创建基于 JPA 注解的 SessionFactory。

```
<!--①通过AnnotationSessionFactoryBean定义SessionFactory -->
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5. annotation.
          AnnotationSession FactoryBean"
      p:dataSource-ref="dataSource">
     <property name="annotatedClasses"> ②
        <list>
            <value>com.smart.orm.domain.Forum</value>
        </list>
    </property>
    <property name="hibernateProperties">
        ...
    </property>
</bean>
```

AnnotationSessionFactoryBean 扩展了 LocalSessionFactoryBean 类，增强的功能是：可以根据实体类的注解获取 ORM 的配置信息。也可以混合使用 XML 配置和注解配置对象关系映射，Hibernate 内部自动将这些元数据信息进行整合，并不会产生冲突。

annotatedClasses 属性指定使用 JPA 注解的实体类名，如②处所示。如果实体类比较多，不要想当然地以为通过 annotatedPackages 属性指定实体类所在包名就可以了，annotatedPackages 在内部通过调用 Hibernate 的 AnnotationConfiguration 的 addPackage()方法加载包中 package-info.java 文件定义的 Annotation，而非包中标注注解的实体类。

Spring 为了通过扫描方式加载带注解的实体类，提供了一个易用的 packagesToScan 属性，可以指定一系列包名，Spring 将扫描并加载这些包路径（包括子包）的所有带注解实体类。

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.annotation.AnnotationSessionFactoryBean"
    p:dataSource-ref="dataSource">
    <property name="packagesToScan" value="com.smart.orm.domain"/>
    <property name="hibernateProperties">
        ...
    </property>
</bean>
```

packagesToScan 属性可接收多个类包路径，用逗号分隔即可，例如：

```
<property name="packagesToScan" value="packeage1,package2"/>
```

14.2.7 事务处理

Spring 的通用事务管理模型对 Hibernate 是完全适用的，包括编程式事务、基于 TransactionProxyFactoryBean、基于 aop/tx 及基于@Transaction 注解的事务管理。在这里，仅给出基于@Transaction 注解的事务管理，其他的方式请参照第 11 章的内容。

首先，为 BbtForumSerive 添加@Transactional 注解。

```
package com.smart.orm.service.hibernate;

...
@Transactional
@Service
public class BbtForumSerive{

    @Autowired
    private ForumHibernateDao forumDao;

    @Autowired
    private TopicHibernateDao topicDao;

    @Autowired
    private PostHibernateDao postDao;
    ...
}
```

其次，在 Spring 配置文件中配置 Hibernate 事务管理器，并启用注解驱动事务。

```
...
<bean id="transactionManager"
     class="org.springframework.orm.hibernate5.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory"/>①  ← 为事务管理器指定 sessionFactory

                                  默认查找名为 transactionManager
<tx:annotation-driven />②  ← 的事务管理器，因此无须显式指定
```

Hibernate 的事务管理器需要注入一个 SessionFactory 实例，将其命名为 transactionManager 后，在<tx:annotation-driven />中就无须通过 transaction-manager 默认显式指定了。不管 BbtForumImpl 所用的 DAO 是基于 HibernateTemplate 还是基于 Hibernate 原生的 API，BbtForumImpl 中的所有方法都具有事务性。

轻松一刻

传说晋朝的旌阳县曾有一名道术高深的县令，叫许逊。他能施符作法，替人驱鬼治病，百姓见他像仙人一样神，就称他为“许真君”。一次，由于年成不好，农民缴不起赋税。许逊便叫大家把石头挑来，然后施展法术，用手指一点，使石头都变成了金子。这些金子足以应付百姓拖欠的赋税。成语“点石成金”据此而来。Spring 通过@Transaction 注解，就可以让业务类这个无事务性的“石头”变成具有事务性的“金子”，Spring 真是一个不折不扣的“许真君”。

14.2.8　延迟加载问题

Hibernate 允许对关联对象、属性进行延迟加载，但是必须保证延迟加载的操作限于同一个 Hibernate Session 范围之内。如果 Service 层返回一个启用了延迟加载功能的领域对象给 Web 层，当 Web 层访问到那些需要延迟加载的数据时，由于加载领域对象的 Hibernate Session 已经关闭，因而将导致延迟加载数据的访问异常。

Spring 为此专门提供了一个 OpenSessionInViewFilter 过滤器，它的主要功能是使每个请求过程绑定一个 Hibernate Session，即使最初的事务已经完成，也可以在 Web 层进行延迟加载操作。

OpenSessionInViewFilter 过滤器将 Hibernate Session 绑定到请求线程中，它将自动被 Spring 的事务管理器探测到。所以 OpenSessionInViewFilter 适用于 Service 层使用 HibernateTransactionManager 或 JtaTransactionManager 进行事务管理的环境，也可用于非只读事务的数据操作中。

要启用这个过滤器，必须在 web.xml 中进行如下配置：

```
...
<filter>
    <filter-name>hibernateFilter</filter-name>
<filter-class>org.springframework.orm.hibernate5.support.OpenSessionInViewFilter
</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
...
```

在上面的配置中，假设使用.html 后缀作为 Web 框架的 URL 匹配模式。如果用户使用 Struts 等 Web 框架，则可以将其改为对应的“*.do”模型。

实战经验

对于小型应用来说，使用 OpenSessionInViewFilter 确实可以降低延迟加载所引发的各种问题，使 Service 层的代码更易开发和维护。但对于大型且高并发的应用来说，笔者强烈建议不要使用 OpenSessionInViewFilter。因为 OpenSessionInViewFilter 会让每个 Web 请求线程都绑定一个 Hibernate 的 Session，即绑定一个数据连接，直到完成 Web 请求处理后才释放数据连接。这会造成两个性能问题：其一，加大了对数据连接资源访问的并发性（因为原本有些 Web 请求可能并不需要使用数据连接）；其二延长了每个 Web 请求对数据连接资源占用的时长。

我们知道，应用系统的瓶颈往往都出现在数据库上，使用 OpenSessionInViewFilter 会使系统更容易出现数据库资源的性能瓶颈。如果要抛弃 OpenSessionInViewFilter，则开发者必须设计好 Service 层的服务接口，使延迟加载的工作在 Service 层内完成。此外，

还要求提供满足不同需求的 Service 接口。假设 User 和 Dept 两个实体对象是 Many-to-One 的关系，User 中的 Dept 使用了延迟加载方案，那么 UserService 可能就需要提供两个不同的获取 User 对象的服务接口，以应对不同的应用场景：其一为 getUserWithDept()，返回带 Dept 的 User；其二为 getUserWithoutDept()，返回不带 Dept 的 User。如果使用 OpenSessionInViewFilter，则仅需提供一个 getUser()服务接口，因为延迟加载的操作在 Web 层由具体调用操作自动触发。

14.3 在 Spring 中使用 MyBatis

使用 MyBatis 提供的 ORM 机制，对业务逻辑实现人员而言，面对的是纯粹的 Java 对象，这一点与使用 Hibernate 框架基本一致。对于具体的数据操作，Hibernate 会自动生成 SQL 语句，而 MyBatis 则要求开发者编写具体的 SQL 语句。相对于 Hibernate 等“全自动”的 ORM 机制而言，MyBatis 在 SQL 开发的工作量和数据库移植性上做出了让步，为数据持久化操作提供了更大的自由空间。相对于“全自动”的 ORM 实现方案来说，MyBatis 的出现显得别有创意。

2010 年 5 月，iBATIS 更名为 MyBatis 并迁移到 Google Code，可以把 MyBatis 看作 iBatis 3.0。从公告上可以看出，开发团队想脱离 Apache 基金会寻求独立发展。开发团队保证不会修改授权协议（Apache License）、代码完全兼容、包名不会更改、不会删除 Apache 站上的任何资源。

由于 Spring 越来越成为 Java 事实标准的技术框架，因此 MyBatis 团队开发出整合类，可以让开发者直接在 Spring 中使用 MyBatis。因此，Spring 4.0 移除了对 iBatis 的直接支持，Spring 更乐于让第三方框架自身提供整合支持。

事务管理可以由 Spring 标准机制进行处理。对于 MyBatis 来说，没有特别的事务策略，除了 JDBC Connection 外，也没有特别的事务资源。它和 Spring JDBC 事务管理的方式完全一致，采用和 Spring JDBC 相同的 DataSourceTransactionManager 事务管理器。

14.3.1 配置 SqlMapClient

每个 MyBatis 的应用程序都以一个 SqlSessionFactory 对象的实例为核心。SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象来获得。SqlSessionFactoryBuilder 对象可以从 XML 配置文件或 Configuration 类的实例中构建 SqlSessionFactory 对象。

和 Hibernate 相似，MyBatis 拥有多个 SQL 映射文件，并通过一个配置文件对这些 SQL 映射文件进行装配，同时在该文件中定义一些控制属性的信息。下面是一个简单的 MyBatis 配置文件，如代码清单 14-7 所示。

代码清单 14-7　mybatisConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
   PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <settings>  ①
        <setting name="lazyLoadingEnabled" value="false" />
    </settings>
    <typeAliases>②
        <typeAlias alias="Forum" type="com.smart.orm.domain.Forum" />
        <typeAlias alias="Topic" type="com.smart.orm.domain.Topic" />
        <typeAlias alias="Post" type="com.smart.orm.domain.Post" />
    </typeAliases>
   <mappers>   ③
     <mapper resource="com/smart/orm/domain/mybatis/Forum.xml"/>
     <mapper resource="com/smart/orm/domain/mybatis/Topic.xml"/>
     <mapper resource="com/smart/orm/domain/mybatis/Post.xml"/>
   </mappers>
</configuration>
```

在①处提供可控制 MyBatis 框架运行行为的属性信息。在②处定义全限定类名的别名，在映射文件中可以通过别名代替具体的类名，简化配置。在③处将 MyBatis 的所有映射文件组装起来。

在③处通过<mappers>标签引用了 3 个 SQL 映射文件，下面来了解一下其中的 Forum.xml 文件，如代码清单 14-8 所示。

代码清单 14-8　Forum.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.smart.orm.dao.mybatis.ForumMybatisDao">①
  <select id="getForum" resultType="Forum" parameterType="int" > ②
        SELECT
         forum_id  forumId,
         forum_name  forumName,
         forum_desc  forumDesc
        FROM t_forum
        WHERE forum_id = #{forumId}
   </select>
   ...
  <insert id="addForum" parameterType="Forum">③
        INSERT INTO t_forum(forum_id,forum_name,forum_desc)
        VALUES(#{forumId},#{forumName}, #{forumDesc})
  </insert>
  <update id="updateForum" parameterType="Forum">④
        UPDATE t_forum f
        SET forum_name=#{forumName},forum_desc=#{forumDesc}
        WHERE f.forum_id = #{forumId}
  </update>
</mapper>
```

该文件定义了对 Forum 实体类进行数据操作时所需的 SQL 语句，同时还定义了查询结果和对象属性的映射关系。在①处指定了映射所在的命名空间，每个具体的映射项都有一个 id，可以通过命名空间和映射项的 id 定位到具体的映射项。如通过如下语句可以调用 getForum 的映射语句：

```
SqlSession session = sqlMapper.openSession();
try {
    Forum forum = (Forum) session.selectOne(
    " com.smart.orm.dao.mybatis.ForumMybatisDao. getForum ", 1);
} finally {
    session.close();
}
```

在②、③和④处分别定义了一条 SELECT、INSERT 及 UPDATE 语句映射项，映射项的 parameterType 指定传入的参数对象，可以是全限定名的类，也可以是类的别名，类的别名在 MyBatis 的主配置文件中定义，如代码清单 14-7 中的②处所示。如果映射项的入参是基础类型或 String 类型，则可以使用如 int、long、string 的基础类型名。SELECT 映射项拥有返回类型对象，通过 resultType 指定。在映射项中通过#{xxx}绑定 parameterType 指定参数类的属性，支持级联属性，如#{topic.forumId}。

14.3.2 在 Spring 中配置 MyBatis

可以使用 MyBatis 提供的 mybatis-spring 整合类包实现 Spring 和 MyBatis 的整合，从功能上来说，mybatis-spring 完全符合 Spring 的风格。要在 Spring 中整合 MyBatis，必须将 mybatis-spring 构件添加到 pom.xml 中（具体参见本章工程的 pom.xml 文件），如代码清单 14-9 所示。

代码清单 14-9 applicationContext-mybatis.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans …>
   <context:component-scan base-package="com.smart.orm.dao.mybatis" />
   <context:component-scan base-package="com.smart.orm.service.mybatis" />
    <context:property-placeholder location="classpath:jdbc.properties" />
    <bean id="dataSource"
       class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />

    <bean id="sqlSessionFactory"
       class="org.mybatis.spring.SqlSessionFactoryBean" ①
       p:dataSource-ref="dataSource"
       p:configLocation="classpath:myBatisConfig.xml"/> ②
</beans>
```

mybatis-spring 类包提供了一个 SqlSessionFactoryBean，以便通过 Spring 风格创建

MyBatis 的 SqlSessioinFactory，如①处所示。只需注入数据源并指定 MyBatis 的总装配置文件就可以了，如②处所示。

如果在 MyBatis 的总装配置文件 mybatisConfig.xml 中指定了 SQL 映射文件，则必须逐个列出所有的 SQL 映射文件，比较烦琐。是否可以像 Spring 加载 Hibernate 映射文件一样按资源路径匹配规则扫描式加载呢？答案是肯定的。SqlSessionFactoryBean 提供了 mapperLocations 属性，支持扫描式加载 SQL 映射文件。

首先将映射文件匹配从 mybatisConfig.xml 中移除，然后通过如下便捷方式加载 SQL 映射文件：

```
<bean id="sqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="dataSource"
      p:configLocation="classpath:myBatisConfig.xml"
      p:mapperLocations="classpath:com/smart/orm/domain/mybatis/*.xml"/>
```

这样，SqlSessionFactoryBean 将扫描 com/smart/orm/domain/mybatis 类路径并加载所有以.xml 为后缀的映射文件。

14.3.3　编写 MyBatis 的 DAO

1. 使用 SqlSessionTemplate

mybatis-spring 效仿 Spring 的风格提供了一个模板类 SqlSessionTemplate，可以通过模板类轻松地访问数据库。

首先在 applicationContext-mybatis.xml 中配置好 SqlSessionTemplate Bean。

```
<bean class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg ref="sqlSessionFactory"/>
</bean>
```

然后就可以使用 SqlSessionTemplate 调用 SQL 映射项完成数据访问操作，如代码清单 14-10 所示。

代码清单 14-10　ForumMybatisTemplateDao.java

```
package com.smart.orm.dao.mybatis;

import org.mybatis.spring.SqlSessionTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.smart.orm.domain.Forum;

@Repository
public class ForumMybatisTemplateDao{

  @Autowired
  private SqlSessionTemplate sessionTemplate;

  public Forum getForum(int forumId){ ①
      return (Forum)sessionTemplate.selectOne(
```

```
                "com.smart.orm.dao.mybatis.ForumMybatisDao.getForum",
                forumId);
    }
}
```

在①处，SqlSessionTemplate 通过 selectOne()方法调用在 Forum.xml 映射文件中定义的命名空间为 com.smart.orm.dao.mybatis.ForumMybatisDao、映射项 id 为 getForum 的 SQL 映射项，并传入参数，返回映射成 Forum 对象的查询结果。

在 SqlSessionTemplate 模板类中提供了多个方便调用的方法，常用方法介绍如下。

- List<?> selectList(String statement, Object parameter)：调用 select 映射项，返回一个结果对象集合。其中，statement 为映射项全限定名，即包括命名空间和映射项 id（下同）；而 parameter 为传递给映射项的入参。
- int insert(String statement, Object parameter)：调用 insert 映射项，返回插入的记录数。
- int update(String statement, Object parameter)：调用 update 映射项，返回更改的记录数。

2．使用映射接口

代码清单 14-10 在①处使用字符串指定映射项，这种方式很容易引起错误。因为字符串本身没有语义性，如果存在编写错误，则在编译期无法识别，只能等到运行期才能发现。MyBatis 为解决这个问题，特别提供了一种可将 SQL 映射文件中的映射项通过名称匹配接口进行调用的方法：接口名称和映射命名空间相同，接口方法和映射元素的 id 相同。

下面为 Forum.xml 文件的映射项定义一个调用接口，如代码清单 14-11 所示。

代码清单 14-11　ForumMybatisDao.java

```
package com.smart.orm.dao.mybatis;

import java.util.List;

import com.smart.orm.domain.Forum;

public interface ForumMybatisDao{
       void addForum(Forum forum);
       void updateForum(Forum forum) ;
      Forum getForum(int forumId) ;
       long getForumNum() ;
      List<Forum> findForumByName(String forumName);
}
```

类名为 com.smart.orm.dao.mybatis.ForumMybatisDao，Forum.xml 文件中的每个映射项对应一个接口方法，接口方法的签名和映射项的声明匹配。

在定义好 ForumMybatisDao 接口后，该如何通过该接口进行数据访问呢？毕竟 ForumMybatisDao 接口没有任何实现类。一种简单的方式是通过 SqlSessionTemplate 获取接口的实例。

```
package com.smart.orm.dao.mybatis;
...
@Repository
public class ForumMybatisTemplateDao{

   @Autowired
   private SqlSessionTemplate sessionTemplate;
   public Forum getForum2(int forumId){
       ForumMybatisDao forumMybatisDao =
           sessionTemplate.getMapper(ForumMybatisDao.class);
       return forumMybatisDao.getForum(forumId);
   }
}
```

SqlSessionTemplate 提供了一个可以根据接口类返回接口实例的方法 getMapper(Class<T> type)，直接访问接口实例的方法，即可调用 SQL 映射文件定义的映射项。

这种方法虽然比直接通过字符串指定映射项的方法安全便捷，但还不是最优的方法。对于 Spring 应用来说，更希望在 Service 类中通过@Autowired 注解直接注入接口实例，而非每次都通过 getMapper(Class<T> type)方法获取实例。

mybatis-spring 提供了一个“神奇”的转换器 MapperScannerConfigurer，它可以将映射接口直接转换为 Spring 容器中的 Bean，这样就可以在 Service 中注入映射接口的 Bean 了。假设已经为 3 个 SQL 映射文件分别定义了对应的接口类，这些接口类位于 com.smart.orm.dao.mybatis 包中，接口名分别为 ForumMybatisDao、TopicMybatisDao 及 PostMybatisDao。使用如下配置即可将接口转换为 Bean：

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"
     p:sqlSessionFactory-ref="sqlSessionFactory"
     p:basePackage="com.smart.orm.dao.mybatis"/>
```

MapperScannerConfigurer 将扫描 basePackage 所指定的包下的所有接口类（包括子包），如果它们在 SQL 映射文件中定义过，则将它们动态定义为一个 Spring Bean，这样就可以在 Service 中直接注入映射接口的 Bean 了，如代码清单 14-12 所示。

代码清单 14-12　ForumMybatisDao.java

```
package com.smart.orm.service.mybatis;
...
import com.smart.orm.dao.mybatis.ForumMybatisDao;
import com.smart.orm.dao.mybatis.PostMybatisDao;
import com.smart.orm.dao.mybatis.TopicMybatisDao;

@Transactional
@Service
public class BbtForumSerive{

    @Autowired
    private ForumMybatisDao forumDao;

    @Autowired
    private TopicMybatisDao topicDao;
```

```
    @Autowired
    private PostMybatisDao postDao;

    public void addForum(Forum forum) {
        forumDao.addForum(forum);
    }
    ...
}
```

如粗体代码所示，BbtForumSerive 可以直接使用@Autowired 注入这些映射接口的 Bean，然后即可顺利地通过接口方法调用 MyBatis 的映射项访问数据。

14.4 DAO 层设计

Spring 为其支持的各种 ORM 框架提供了基于模板模式的 Template 基类，此外还为模板类提供了方便的支持类，其内部包含一个模板类，Spring 推荐开发者直接继承这个 Support 类定义自己的 DAO。但是，在实际应用中，直接继承 Spring 的 Support 定义实体类 DAO 存在一些不足之处，下面将探讨通过引入一个基类简化具体 DAO 类的设计思路。

此外，Spring 提供的各种 ORM 模板类的查询方法使用 Object[]传递查询条件参数。很多开发者发现使用 Object[]传递查询条件参数并非一个最优的办法，因此出现了很多查询方法的设计版本。本节对此进行了汇总，读者可以通过对比，找到自己认为最适合的方式。

14.4.1 DAO 基类设计

虽然 Spring 通过模板类和支持类为各种 ORM 框架提供了出色的支持，但也存在一些不足。首先，这些模板类对泛型的支持是方法级别的，需要通过方法入参指定泛型的类型。对于保存、更改、删除等操作，泛型的意义并不是很大；但对于加载实体、查询实体的操作，泛型可以带来很大的便利。其次，由于支持类通过类似于 getHibernateTemplate()、getJdbcTemplate()的方式向子类开放模板类，所以子类在执行数据操作时，必须采取诸如 getJdbcTemplate ().execute ()的形式，子类代码显得比较拖沓。

在实际应用中，开发者一般会在 Spring 支持类的基础上编写自己的 DAO 基类，进行自己的封装，以获得泛型的支持，并提供自己的代理方法，使子类避免通过显式调用模板类实例进行数据操作。DAO 基类并不需要对模板类的所有方法进行代理，仅需对一些常用的方法（如 CRUD 操作方法）进行代理即可，对不常用的方法则可要求子类显式调用模板实例完成。这样，一方面简化了常用方法的调用，另一方面避免了使基类过于复杂。

由于一般实体类对应的 DAO 都必须拥有 CRUD 操作，与其在每个实体 DAO 接口中重复定义这些方法，不如提供一个通用的 DAO 接口。具体的实体 DAO 接口可以通过扩展这个通用的 DAO 接口并定义实体类相关的其他操作方法来实现。

在没有类级别泛型支持的情况下查询实体，不但要指定主键值，还需要指定实体类型；由于方法入参比较复杂，调用者还需要对返回的结果对象进行强制类型转换。这两种缺点都可以通过在基类中引入泛型得到彻底解决。

BaseDao<T>基类对所有实体 DAO 接口的通用方法进行了抽象并提供了泛型支持，下面以 Hibernate 为例，说明项目 DAO 基类的设计方式，如代码清单 14-13 所示。

代码清单 14-13　BaseDao<T>：DAO基类

```
package com.smart.orm.dao;

import java.io.Serializable;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate5.HibernateTemplate;

public class BaseDao<T> { //①提供DAO类级别的泛型支持

   @Autowired
   private HibernateTemplate hibernateTemplate; //②注入Hibernate模板类

   private Class entityClass; //③DAO的泛型类，即子类所指定的T所对应的类型

   public BaseDao(){//④通过反射方式获取子类DAO对应的泛型实体类
       Type genType = getClass().getGenericSuperclass();
       Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
       entityClass = (Class) params[0];
    }

    public T get(Serializable id){
       return (T)hibernateTemplate.get(entityClass, id); //⑤直接使用entityClass
    }

    public void save(T entity){
       hibernateTemplate.save(entity);
    }

    public void update(T entity){
       hibernateTemplate.update(entity);
    }

     public HibernateTemplate getHibernateTemplate() {
         return hibernateTemplate;
     }
    ...
}
```

基类 BaseDao<T>通过泛型方式允许子类 DAO 指定操作的实体类，以便简化常用的

数据操作方法。同时，在②处，BaseDao 注入了一个 HibernateTemplate，这样子类只要标注@Repository 注解就会拥有 HibernateTemplate 成员变量，无须各自声明。BaseDao 的构造方法通过 Java 反应机制自动解析出子类 T 所对应的类型，以便 HibernateTemplate 直接利用这个信息进行数据访问操作。如⑤处的 hibernateTemplate.get(entityClass, id)就使用泛型的类型，得到了一个更为便捷的 get()方法。

此外，还可以在该类中提供更多方便的操作方法化解子类的实现难度。这样，子类仅需指定泛型对应的实体类，就拥有了各种通用的数据操作能力。如果实体 DAO 的数据操作仅是一些常见的 CRUD 操作，那么子类甚至可以不编写任何代码，DAO 类的编码生产率将得到极大的提高，代码复用率也将上升一个台阶。通过扩展 BaseDao<T>基类可以得到一个被极大简化的 ForumDao 版本。

```
package com.smart.orm.dao;

import com.smart.orm.domain.Forum;

public class ForumDao extends BaseDao<Forum> {//①通过泛型指定实体类为Forum

  //②子类仅需编写特定的DAO方法就可以了，其他通用的方法从基类继承
public long getForumNum() {
        Object obj = getHibernateTemplate().iterate(
        "select count(f.forumId) from Forum f").next();
      return (Long) obj;
    }
}
```

通过扩展 BaseDao<T>基类并借由泛型指定实体类，ForumDao 立即拥有了对 Forum 实例进行 CRUD 操作的功能，并且这些功能都是泛型版本的。如当用户调用 ForumDao#get(Serializable id)方法加载实体对象时，将直接返回 Forum 类型的实体对象。因此，在 ForumDao 类中，开发者仅需编写个性化的接口方法即可，普通的增、删、查、改方法直接从 BaseDao 中继承。按照相似的方式，还可以轻松完成 TopicDao、PostDao 的编写，用户将发现 DAO 层的代码质量拥有了一个不小的飞跃。

14.4.2 查询接口方法设计

DAO 层除了 CRUD 的数据操作外，另一个重要的操作就是根据查询条件执行数据查询，不同的 ORM 框架都允许用户动态绑定参数确定查询条件。查询条件项的数目往往是不固定的，如既可能要求以 userName 为条件查询 User，也可能要求以 userName+status 等组合条件查询 User。条件项数目的不定性给查询接口方法的设计造成了一定的困难。在实体 DAO 定义带参的查询方法时，一般有 5 种方式，下面分别进行介绍。

方式 1：每个条件项参数对应一个入参

在查询方法中为每个查询条件项定义一个对应的入参，例如：

```
List  findOrder(String hql,Date startTime,Date endTime,int deptId)
```

这种方法签名含义清晰，可读性强，内部的逻辑处理简单，但接口稳定性差。如果findOrder()方法需要添加一个 userName 条件，则有两种重构的方式。第一，调整方法的签名，新增一个 String userName 入参，这种重构被认为是不健康的重构，因为它违反了软件设计中经典的“开-闭原则”。此外，如果条件项个数很多，方法签名将显得过于拖沓。第二，在 DAO 类中新增一个重载查询方法。但如果查询条件项的组合数过多，则 DAO 类的方法数目将直线上升，整个实体 DAO 类将显得臃肿笨重。

方式 2：使用数组传递条件项参数

即通过数组的方式传递查询条件项参数。由于参数类型具有不一致性，因而要求数组类型采用 Object[]。

```
List  findOrder(String hql,Object[] params)
```

这种方式可以应对查询条件项参数组合的多样性并保持接口的稳定性，开发者必须在方法内部从数组中获取参数再传递给查询引擎。其缺点是方法的可读性不强，调用者往往需要通过查看接口对应的 Javadoc 文档才能正确使用。此外，在 Java 5.0 以下的版本中，因为没有自动拆/解包的语法特性，所以在调用前必须对基本类型的参数使用封装类进行封装，有时在方法内部还需要进行相反的过程，在使用上较为不便。不过由于 Spring 为支持的 ORM 框架都提供了类似的查询接口（如 HibernateTemplate#find(String queryString, Object[] values)），所以 DAO 方法内部的处理相对还是比较简单的。

方式 3：使用 Java 5.0 的不定参数

如果是在 Java 5.0 中，则可以采用不定参数进行方法签名。这种方式在逻辑上和方式 2 并无多大的区别。

```
List  findOrder(String hql,Object... params)
```

方式 4：将查询条件项参数封装成对象

为了提高方式 1 中方法签名的简洁性，增强方式 2、3 中方法签名的可读性，方式 4 提出将查询条件项参数封装成一个对象的思路，即查询方法使用查询条件对象传递查询条件。

```
List<Order> findOrder(String hql,OrderQueryParam param)
```

OrderQueryParam 查询条件对象封装了 hql 查询语句可能会用到的条件项参数，在查询方法内部，开发者必须判断查询条件对象的属性并正确绑定条件项参数。由于需要为条件项参数定义一个类，因此会造成类数目的膨胀，有时甚至一个实体 DAO 需要对应多个查询条件参数类。另外，当需要添加一个新的条件项参数时，条件封装类还需要进行相应调整。

方式 5：使用 Map 传递条件项参数

另一种被广泛使用的方式是采用 Map 传递条件项参数，键为参数名，值为参数值。

```
List<Order> findOrder(String hql,Map params)
```

使用这种方式，接口方法签名可以在条件项发生变化的情况下保持稳定，同时通过

键指定条件项参数名在一定程度上解决了接口的健壮性（当调用者指定错误参数名时容易得到错误报警）。但和方式 2、3 一样，调用者必须通过接口 Javadoc 文档才能明白不同条件项要以什么名称进行设置。

14.4.3 分页查询接口设计

绝大多数应用系统都采用分页的方式展现业务数据，并在客户端业务列表页面中提供分页导航栏，用户可以通过分页导航栏查看各页的业务数据。按获取数据方式的不同，分页技术大致可分为以下 3 种。

- 客户端分页：直接将全部或多页结果数据一次性返回给客户端，客户端通过展现组件进行数据分页的控制。
- 数据库分页：在进行数据查询时，数据库仅返回一页数据给客户端。
- 服务器端分页：从数据库返回全部或多页数据，在服务器端缓存多页数据，但只返回一页数据给客户端。

客户端分页可以减小和服务器交互的次数，在进行分页切换时，直接从客户端的缓存中获取数据，无须和服务器端进行再次交互，提高了系统交互性；但会增加第一次交互的负荷。数据库分页要求每次切页时都访问数据库，这增加了数据库访问的并发性；但每次从数据库返回的数据较少，当次交互的负荷较轻。而服务器端分页在以上两者之间寻求平衡，它既减少了数据库访问的并发性，同时使服务器端返回给客户端的当次负荷也较小；但服务器端分页技术需要考虑到数据缓存、数据同步等问题，提高了系统的复杂性。

DAO 不可避免地需要涉及分页查询接口，分页查询接口包括查询参数、分页设置及分页结果等对象。下面是一个简单的分页查询接口的设计方案，如图 14-1 所示。

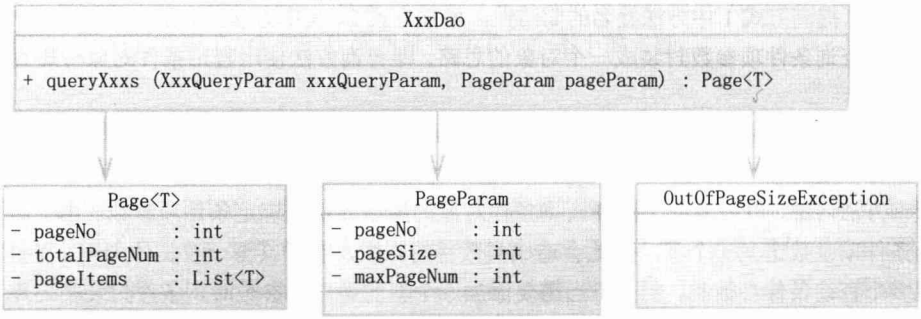


图 14-1 分页查询接口的设计方案

queryXxxs 有两个入参，其中 XxxQueryParam 为查询参数对象，而 PageParam 为分页的设置信息，包括页码（pageNo）、每页记录数（pageSize）及允许查询的最大查询页数（maxPageNum）等。Page<T>为当前页数据的封装类，其中 pageNo 为页码，totalPageNum 为结果集总页数（客户端的分页导航栏需要用到 pageNo 和 totalPageNum），pageItems 为当

前页的业务数据集合。当需要查询的页码超过结果集的最大页数，或者超过允许查询的最大页数时，将抛出 OutOfPageSizeException 运行期异常。

14.5　小结

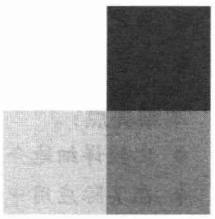
Spring 为其支持的 ORM 框架提供了方便易用的 FactoryBean，用以创建 ORM 框架的基础设施。Spring 通过模板类在不损失框架功能性的前提下大大降低了使用这些 ORM 技术的难度，因此成为 Spring 推荐的使用模式。不过，Spring 并不强制使用模板类，对于任意一种 ORM 技术，Spring 都允许用户使用原生的 API 构造 DAO。在使用原生 API 时，Spring 能保证用户获取到事务绑定的资源，Spring 的事务管理机制同样有效。

Spring 对 Hibernate 所提供的支持应该是最丰富的，以创建 SessionFactory 来例，就有多种配置方式，如直接使用 Hibernate 配置文件、使用混合型的 Spring 风格配置及通过注解的配置方式。而对 MyBatis 的集成，目前只能通过 myBatis-spring 来完成，不过依然方便易用。

虽然 Spring 提供了使用各种 ORM 框架的方便类，但这离实际的应用开发还有一段距离。具体的应用一般会定义一个项目级的 DAO 基类，简化接口方法，添加泛型支持。一个设计良好的 DAO 基类可以大大减少 DAO 层的代码总量，提高项目的开发效率。

第4篇

应用篇



第 15 章

Spring Cache

伴随着信息量的爆炸式增长，以及构建的应用系统越来越多样化、复杂化，特别是伴随着近年来企业级应用互联网化的趋势，缓存（Cache）对应用程序性能的优化变得越来越重要。将所需服务请求的数据放到缓存中，既可以提高应用程序的访问效率，又可以减少数据库服务器的压力，从而让用户获得更为极致的体验。

Spring 开发者正是因为看到了缓存在应用中的重要地位，从 Spring 3.1 开始，就以一贯的优雅风格提供了一种透明的缓存解决方案，这使得 Spring 可以在后台使用不同的缓存框架（如 EhCache、GemFire、HazelCast 和 Guava）时保持编程的一致性。而从 Spring 4.0 开始则全面支持 JSR-107 annotations 和自定义的缓存标签。

本章主要内容：

- 了解缓存的基本概念
- Spring 对 Cache 的支持
- 自定义 Cache 注解的使用
- 与企业级缓存的集成

本章亮点：

- 比较详细地介绍了 Cache 在系统中的使用场景和方式
- 在实际应用中开发缓存程序的实战经验

15.1 缓存概述

15.1.1 缓存的概念

缓存作为系统架构中提升性能的一种重要支撑技术，在企业级应用中的地位越来越

突显。缓存技术日新月异，可选的缓存套件也琳琅满目，让人应接不暇。本节先不急于介绍 Spring Cache 的相关知识，而是先介绍缓存的一些基本概念，再引出 Spring Cache，以求先知“庐山真面目”，再从 Spring 的角度“一览众山小”。

其实在现实生活中也可以找到很多“缓存”的影子。比如，京东的物流为什么那么快，甚至可以实现当日送达，就因为其在全国各地都有分仓库，在发货的时候，会找离客户最近的仓库，如果该仓库有货物，则安排就近送货。用过 Maven 的朋友应该知道，在找依赖构件的时候，先从本机仓库找，如果没有再从本地服务器仓库找，最后才到远程服务器仓库找。

所以可将缓存定义为一种存储机制，它将数据保存在某个地方，并以一种更快的方式提供服务。较为常见的一种情况是在应用中使用缓存机制，以避免方法的多次执行，从而克服性能缺陷，也可减少应用服务器或者数据库的压力。缓存的策略有很多种，在应用系统中可根据实际情况选择，通常会把一些静态数据或者变化频率不高的数据放到缓存中，如配置参数、字典表等。而有些场景可能要寻求替代方案，比如，想提升全文检索的速度，在复杂的场景下建议使用搜索引擎，如 Solr 或 ElasticSearch。

通常在 Web 应用开发中，不同层级对应的缓存要求和缓存策略全然不同。如图 15-1 所示列举了系统不同层级对应的缓存技术选型。

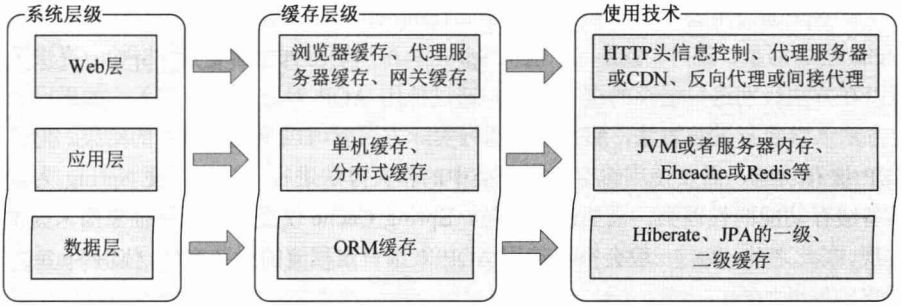


图 15-1　系统层级对应的缓存层级技术

要了解缓存，必须对其中的一些基本概念有所了解。下面先来了解一下缓存中两个比较重要的基本概念。

1. 缓存命中率

即从缓存中读取数据的次数与总读取次数的比率。一般来说，命中率越高越好。

命中率 = 从缓存中读取的次数 /(总读取次数[从缓存中读取的次数 + 从慢速设备上读取的次数])

Miss 率 = 没有从缓存中读取的次数 /(总读取次数[从缓存中读取的次数 + 从慢速设备上读取的次数])

这是一个非常重要的监控指标，如果要做缓存，就一定要监控这个指标，来看缓存是否工作良好。还是之前举的例子，假设京东的某个分仓库通过监控发现某个仓库的命

中率一直很低，经常无法找到待发货的商品，那么这时候就需要调整策略，否则就会造成搏击里面“出拳百次，无一命中”的笑话。

2．过期策略

即如果缓存满了，从缓存中移除数据的策略。常见的有 LFU、LRU、FIFO。

- FIFO（First In First Out）：先进先出策略，即先放入缓存的数据先被移除。
- LRU（Least Recently Used）：最久未使用策略，即使用时间距离现在最久的那个数据被移除。
- LFU（Least Frequently Used）：最近最少使用策略，即一定时间段内使用次数（频率）最少的那个数据被移除。
- TTL（Time To Live）：存活期，即从缓存中创建时间点开始直至到期的一个时间段（不管在这个时间段内有没有访问都将过期）。
- TTI（Time To Idle）：空闲期，即一个数据多久没被访问就从缓存中移除的时间。

至此，我们基本了解了缓存的一些基本知识。在 Java 中，一般会对调用方法进行缓存控制。比如调用“findUserById(String id)”，应该在调用这个方法之前先从缓存中查找有没有符合查询条件的数据，如果没有，则执行该方法从数据库中查找该用户，然后添加到缓存中，下次调用时将会从缓存中获取该数据。

从 Spring 3.1 开始，提供了类似于@Transactional 事务注解的缓存注解，且提供了 Cache 层的抽象。此外，JSR-107 也从 Spring 4.0 开始得到全面支持。Spring 提供了一种可以在方法级别进行缓存的缓存抽象。通过使用 AOP 对方法进行织入，如果已经为特定方法入参执行过该方法，那么不必执行实际方法就可以返回被缓存的结果。为了启用 AOP 缓存功能，需要使用缓存注解对类中的相关方法进行标记，以便 Spring 为其生成具备缓存功能的代理类。需要注意的是，Spring Cache 仅提供了一种抽象而未提供具体实现。在此之前，我们一般会自己使用 AOP 来做一定程度的封装实现，使用 Spring Cache 带来的好处如下：

- 支持开箱即用（Out-Of-The-Box），并提供基本的 Cache 抽象，方便切换各种底层 Cache。
- 类似于 Spring 提供的数据库事务管理，通过 Cache 注解即可实现缓存逻辑透明化，让开发者关注业务逻辑。
- 当事务回滚时，缓存也会自动回滚。
- 支持比较复杂的缓存逻辑。
- 提供缓存编程的一致性抽象，方便代码维护。

提示

Java 缓存规范 JCache API（JSR-107）对 Java 对象缓存进行了标准化，方便高效开发，让程序员摆脱了实现缓存有效期、互斥、假脱机（Spooling）和缓存一致性等负担。该规范提供了 API、RI（参考实现）和 TCK（技术兼容性套件）。读者可以通

过以下地址详细了解该规范：https://jcp.org/en/jsr/detail?id=107。同时，本章后面也会详细介绍该规范。有兴趣的读者可以关注 Java 的另一个规范 JSR-347——Java 数据网格规范。这两个规范容易让人混淆，但是 JSR-347 更多关注的是数据处理的分布式特性，如分布式存储、Map/Reduce 分布式计算等。

需要注意的是，Spring Cache 并不针对多进程的应用环境进行专门的处理，也就是说，当应用程序处于分布式或者集群环境下时，需要针对具体的缓存进行相应的配置。如 EhCache 可以通过 RMI、JGroups 及 EhCache Server 等方式来配置其多播集群环境。另外，在 Spring Cache 抽象的操作中没有锁的概念，当多线程并发操作（更新或者删除）同一个缓存项时，将可能得到过期的数据。有些缓存实现提供了锁的功能，如果需要考虑如上场景，则可以详细了解具体缓存的一些相关特性，如 EhCache 就提供了针对缓存元素 key 的 Read（读）、Write（写）锁。

轻松一刻

小情侣在打电话。女："你说话的声音怎么怪怪的？"男："在清除呼吸系统的缓存呢。"女："说人话!!"男："我在挖鼻孔……"

15.1.2　使用 Spring Cache

这里先展示一个自定义的缓存实现，即不通过任何第三方组件来实现对象的内存缓存，然后通过 Spring Cache 来实现缓存操作，以体会 Spring Cache 所带来的优雅和便捷性。

假设一个场景：在日常所见的项目中，用户查询是一个非常频繁的动作，从性能优化的角度，自然会想到对一个用户的查询方法做缓存，以避免频繁的数据库访问操作，提高页面的响应速度。通常的做法是以用户的 userId 作为 key，以返回的用户信息对象作为 value 值存储。而当以相同的 userId 查询用户时，程序将直接从缓存中获取结果并返回；否则更新缓存。

首先定义一个 User 实体类，该类具备基本的 userId、age 及 userName 属性，且具备 getter 和 setter 方法，如代码清单 15-1 所示。

代码清单 15-1　User.java

```
package com.smart.cache.domain;
import java.io.Serializable;

public class User implements Serializable {
    private String userId;
    private String userName;
    private int age;
```

```
    //省略get/setXxx方法
}
```

Java 对象的缓存和序列化是息息相关的，一般情况下，需要被缓存的实体类需要实现 Serializable，只有实现了 Serializable 接口的类，JVM 才可以对其对象进行序列化。对于 Redis、EhCache 等缓存套件来说，被缓存的对象应该是可序列化的，否则在网络传输、硬盘存储时都会抛出序列化异常。当然，由于这里直接采用 Map 以内存的方式存储缓存对象，所以 User 不实现 Serializable 接口也不会有问题，但是实体类始终实现 Serializable 接口是一个好的编程习惯。实现 Serializable 接口的实体类，一般要求声明一个 serialVersionUID 成员变量，以表明该实体类的版本。如果实体类的结构发生变化，则可以修改 serialVersionUID 值以支持反序列化工作。关于对象序列化和反序列化的更多知识，读者可自行查找相关资料学习，在此不再展开。

接下来定义一个缓存管理器，这个管理器负责实现缓存逻辑，支持对象的增加、修改和删除，并且支持值对象的泛型，具体实现如代码清单 15-2 所示。

代码清单 15-2　CacheManager.java

```
package com.smart.cache.mycache;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class CacheManager<T> {
  private Map<String,T> cache =
      new ConcurrentHashMap<String,T>();

  public T getValue(Object key) {
    return cache.get(key);
  }

  public void addOrUpdateCache(String key,T value) {
    cache.put(key, value);
  }
                                          根据 key 来删除缓存中的一条记录
  public void evictCache(String key) { ①
    if(cache.containsKey(key)) {
      cache.remove(key);
    }
  }
                                    清空缓存中的记录
  public void evictCache() {②
    cache.clear();
  }
}
```

现在有了实体类（User）和一个缓存管理器（CacheManager），还需要一个提供用户查询的服务类，此服务类使用缓存管理器来支持用户查询，如代码清单 15-3 所示。

代码清单 15-3　UserService.java

```
package com.smart.cache.mycache;
import com.smart.cache.domain.User;
```

```
public class UserService {
    private CacheManager<User> cacheManager;

    public UserService() {

        //构造一个缓存管理器
        cacheManager = new CacheManager<User>();
    }

    public User getUserById(String userId) {

     //首先查询缓存
     User result = cacheManager.getValue(userId);
        if(result!=null) {
            System.out.println("get from cache..."+userId);

            //如果在缓存中，则直接返回缓存的结果
            return result;
        }

        // 否则到数据库中查询
        result = getFromDB(userId);
        if(result!=null) {

            //将数据库查询的结果更新到缓存中
            cacheManager.addOrUpdateCache(userId, result);
        }
        return result;
    }

    public void reload() {
        cacheManager.evictCache();
    }

    private User getFromDB(String userId) {
        System.out.println("real querying db..."+userId);
        return new User(userId);
    }
}
```

现在开始写一个测试类，用于测试刚才的方法，如代码清单 15-4 所示。

代码清单 15-4　UserMain.java

```
package com.smart.cache.mycache;

public class UserMain {
    public static void main(String[] args) {
      UserService userService = new UserService();

      //开始查询账号
      userService.getUserById("001001");//第一次查询，应该是数据库查询
      userService.getUserById("001001");//第二次查询，应该直接从缓存返回

       //重置缓存
      userService.reload();
```

```
        System.out.println("after reload...");

        //应该是数据库查询
        userService.getUserById("001001");

        //第二次查询，应该直接从缓存返回
        userService.getUserById("001001");
    }
}
```

按照分析，执行结果应该是：首先从数据库加载，然后直接返回缓存中的结果；重置缓存后，应该首先从数据库加载，然后返回缓存中的结果。实际的执行结果如下：

```
从数据库中查询...001001//第一次从数据库加载
从缓存中查询...001001//第二次从缓存加载
重置缓存...//清空缓存
从数据库中查询...001001//又从数据库加载
从缓存中查询...001001//从缓存加载
```

可以看出，自定义缓存可以正常工作，但这种实现方式并不优雅：缓存代码和业务代码高度耦合，业务代码中穿插着大量的缓存控制逻辑，并且代码显式依赖缓存的具体实现。此外，本实现方案并不支持按条件缓存，比如只有符合某种条件的用户账号才需要缓存。如果要新增这一功能，代码将进一步复杂化。

下面使用 Spring Cache 来实现上面这个例子，基于刚才自定义缓存方案的实体类 User，改造 UserService 和 UserMain，如代码清单 15-5 所示。因为 Spring 已经提供了默认的缓存管理器，所以这个例子不用实现自定义的缓存管理器。

代码清单 15-5　UserService.java

```
package com.smart.cache.simplecache;
import com.smart.cache.domain.User;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service(value = "userServiceBean")
public class UserService {

    //使用一个名为 users的缓存
    @Cacheable(cacheNames = "users")
    public User getUserById(String userId) {

        //方法内部实现不考虑缓存逻辑，直接实现业务
        System.out.println("real query user." + userId);
        return getFromDB(userId);
    }

    private User getFromDB(String userId) {
        System.out.println("real querying db..." + userId);
        return new User(userId);
    }
}
```

getUserById()方法标注了一个注解：@Cacheable(cacheNames ="users")，当调用这个方法的时候，会先从 users 缓存中查询匹配的缓存对象，如果存在则直接返回；如果不

存在，则执行方法体内的逻辑（查询数据库），并将返回值放入缓存中。对应缓存的 key 为userId 的值，value 就是userId 所对应的User 对象，缓存名称需要在applicationContext.xml 中定义。

现在还需要一个 Spring 配置文件来支持基于注解的缓存，如代码清单 15-6 所示。

代码清单 15-6　支持基于注解的缓存

```
<?xml version="1.0" encoding="UTF-8" ?>
   <cache:annotation-driven />
   <bean id="accountServiceBean" class="com.smart.cache.simplecache.UserService"/>
   <bean id="cacheManager"
       class="org.springframework.cache.support.SimpleCacheManager">
      <property name="caches">
         <set>
            <bean
    class="org.springframework.cache.concurrent.ConcurrentMapCache
                  FactoryBean"  p:name="default" />
            <bean
    class="org.springframework.cache.concurrent.ConcurrentMapCache
                  FactoryBean"  p:name="users" />
         </set>
      </property>
   </bean>
</beans>
```

Spring 通过<cache:annotation-driven />即可启用基于注解的缓存驱动，这个配置项默认使用了一个定义为 cacheManager 的缓存管理器。SimpleCacheManager 是这个缓存管理器的默认实现，它通过对其属性 caches 的配置来实现刚刚自定义的缓存管理器逻辑。从上面的代码中可以看到，除了默认的 default 缓存外，还自定义了一个名为 users 的缓存，使用了默认的内存存储方案 ConcurrentMapCacheFactoryBean，它是一个基于 java.util.concurrent.ConcurrentHashMap 的内存缓存实现方案。

接下来便可以重写 UserMain，以观察使用 Spring Cache 运行的效果，如代码清单 15-7 所示。

代码清单 15-7　UserMain.java

```
package com.smart.cache.simplecache;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserMain {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext(
            "applicationContext.xml");//加载 Spring 配置文件
      UserService userService = (UserService) context.getBean("userServiceBean");

      //第一次查询，应该查询数据库
      System.out.print("first query...");
      userService.getUserById("somebody");

      //第二次查询，应该不查询数据库，直接返回缓存中的值
```

```
        System.out.print("second query...");
        userService.getUserById("somebody");
    }
}
```

上面的测试代码进行了两次查询，第一次应该查询数据库，第二次应该返回缓存中的值而不再查询数据库。执行上述代码，结果如下：

```
第一次查询...执行用户业务查询方法查找.用户//第一次查询
从数据库中查询...用户//对数据库进行了查询
第二次查询...//第二次查询，没有打印数据库查询日志，直接返回了缓存中的值
```

可以看出，我们配置的基于注解的缓存起作用了，而在 UserService 的代码中没有看到任何缓存逻辑代码，只需一个注解@Cacheable(cacheNames="users")，就实现了基本的缓存方案，代码变得非常优雅、简洁。事实上，使用 Spring Cache 非常简单，开发人员只需完成两个步骤。

- 缓存定义：确定需要缓存的方法和缓存策略。
- 缓存配置：配置缓存。

接下来我们将深入了解 Spring Cache，以探究其中的奥秘。

15.2 掌握 Spring Cache 抽象

15.2.1 缓存注解

Spring Cache 提供了 5 种可在方法级别或类级别上使用的缓存注解。这些注解定义了哪些方法的返回值会被缓存或者从缓存中移除。注意，只有使用 public 定义的方法才可以被缓存，而 private 方法、protected 方法或者使用 default 修饰符的方法都不能被缓存。当在一个类上使用注解时，该类中每个公共方法的返回值都将被缓存到指定的缓存项中或者从中移除。本节将详细介绍这 5 种缓存注解的用法及所支持的属性。

1. @Cacheable

@Cacheable 是最主要的注解，它指定了被注解方法的返回值是可被缓存的。其工作原理是 Spring 首先在缓存中查找数据，如果没有则执行方法并缓存结果，然后返回数据。缓存名是必须提供的，可以使用引号、Value 或者 cacheNames 属性来定义名称。下面的定义展示了 users 缓存的声明及其注解的使用：

```
@Cacheable("users")
//Spring 3.x
@Cacheable(value= "users")
//Spring 4.0 新增了 value 的别名 cacheNames，比 value 命名更达意，推荐使用
@Cacheable(cacheNames = "users")
```

此外，还可以以列表的形式提供多个缓存，在该列表中使用逗号分隔缓存名称，并用花括号括起来。以下代码展示了在@Cacheable 注解中定义的两个缓存 cache1 和 cache2：

```
@Cacheable(cacheNames = {"cache1","cache2"})
```

下面所示的代码片段展示了如何在方法上应用@Caheable 注解：getUser(String id)方法以 id 值为键值将用户缓存到 users 缓存段中。此外，还可以提供自定义键，以便获取存储在某一缓存中的数据。

```
@Cacheable(cacheNames ="users")
public User getUser(String id) {
    return user.get(id);
}
```

1）键生成器

缓存的本质就是键/值对集合。在默认情况下，缓存抽象使用方法签名及参数值作为一个键值，并将该键与方法调用的结果组成键/值对。如果在 Cache 注解上没有指定 key，则 Spring 会使用 KeyGenerator 来生成一个 key。

```
public interface KeyGenerator {
    Object generate(Object target, Method method, Object... params);
}
```

Sping 默认提供了 SimpleKeyGenerator 生成器。Spring 4.0 废弃了 3.x 的 DefaultKeyGenerator 而用 SimpleKeyGenerator 取代，原因是 DefaultKeyGenerator 在有多个入参时只是简单地把所有入参放在一起使用 hashCode()方法生成 key 值，这样很容易造成 key 冲突。SimpleKeyGenerator 使用一个复合键 SimpleKey 来解决这个问题。接下来通过 Spring 源码来看看 Spring 生成 key 的规则。

```
public static Object generateKey(Object... params) {
    if (params.length == 0) {
        return SimpleKey.EMPTY;
    }
     if (params.length == 1) {
        Object param = params[0];
            if (param != null && !param.getClass().isArray()) {
                return param;
            }
    }
    return new SimpleKey(params);
}
```

通过源码来学习 Spring 往往简单明了，从上面的代码中可以发现其生成规则如下：

（1）如果方法没有入参，则使用 SimpleKey.EMPTY 作为 key。

（2）如果只有一个入参，则使用该入参作为 key。

（3）如果有多个入参，则返回包含所有入参的一个 SimpleKey。

此外，还可以在声明中指定键值。@Cacheable 注解提供了实现该功能的 key 属性，通过该属性，可以使用 SpEL 指定自定义键，如下：

```
@Cacheable(cacheNames ="users", key = "#user.userCode")
public User getUser(User user, boolean checkLogout)
```

如上所示，入参中有一个 boolean 值用于区分用户是否已经注销，而这个 boolean 值并不想作为 key 的一部分，那么可以根据 key 属性指定使用 userCode 作为缓存键。

当然，如果 key 生成策略涉及一些比较复杂的算法，而这个 key 生成策略又是通用的，则可以通过实现 org.springframework.cache.interceptor.KeyGenerator 接口来定义个性

化的 key 生成器。如自定义了一个 MyKeyGenerator 类并且实现了 KeyGenerator 接口以实现自定义的 key 生成器，那么便可如下使用：

```
@Cacheable(cacheNames ="users",  keyGenerator= "myKeyGenerator")
public User getUser(User user,  boolean checkWork)
```

2）带条件的缓存

使用@Cacheable 注解的 condition 属性可按条件进行缓存，condition 属性使用了 SpEL 表达式动态评估方法入参是否满足缓存条件。关于 SpEL 表达式的相关内容，后文会进行介绍。

对于使用了@Cacheable 注解声明的 getUser()方法，我们启用了一个缓存判断条件：仅对年龄小于 35 周岁的用户启用缓存。

```
@Cacheable(cacheNames = "users", condition = "#user.age < 35")
public User getUser(User user){
    return users.get(user.getUserId());
}
```

在上述代码中，#user 引用方法的同名入参变量，接着通过.age 访问 user 入参对象的 age 属性值。在调用方法前，将对注解中声明的条件进行评估，满足条件才缓存。

与 condition 属性相反，可使用 unless 属性排除某些不希望缓存的对象。下面的示例拒绝了对年龄大于或等于 35 周岁的用户进行缓存。

```
@Cacheable(value="users", unless = "#user.age >= 35")
 public User getUser(User user){
    return users.get(user.getUserId());
 }
```

下面通过表 15-1 对@Cacheable 注解的参数进行详细说明。

表 15-1　@Cacheable注解参数说明

参　数	说　明	示　例
value/cacheNames	缓存的名称，在 Spring 配置文件中定义，必须指定至少一个	@Cacheable(cacheNames="cachename")　或　者 @Cacheable(cacheNames={"cache1","cache2"}
key	缓存的 key，可以为空。如果指定，则按照 SpEL 表达式编写；如果不指定，则默认按照方法的所有参数进行组合	@Cacheable(cacheNames="cachename",key="#user Name")
condition	缓存的条件，可以为空，使用 SpEL 表达式编写，返回 true 或者 false，只有返回 true 才进行缓存。unless 属性与 condition 属性相反，满足条件则不进行缓存	@Cacheable(cacheNames="cachename",condition= "#userName.length()>2")

2．@CachePut

@CachePut 注解与@Cacheable 注解效果几乎一样，它首先执行方法，然后将返回值放入缓存。当希望使用方法返回值来更新缓存时，便可以选择这种方法，如下：

```
@CachePut(value = "users")
public User getUser(int id) {
System.out.println("User with id " + id + " requested.");
    return users.get(id);
}
```

在上述代码中，首先执行 getUser()方法，然后将结果值放入 users 缓存。与@Cacheable 注解一样，@CachePut 注解也提供了 key、condition 和 unless 属性。

下面通过表 15-2 对@CachePut 注解的参数进行详细说明。

表 15-2　@CachePut注解参数说明

参　数	说　明	示　例
value/cacheNames	缓存的名称，在 Spring 配置文件中定义，必须指定至少一个	@CachePut(cacheNames="cachename")或者 @CachePut (cacheNames={"cache1","cache2"}
key	缓存的 key，可以为空。如果指定，则按照 SpEL 表达式编写；如果不指定，则默认按照方法的所有参数进行组合	@CachePut(cacheNames="cachename",key= "#userName")
condition	缓存的条件，可以为空，使用 SpEL 表达式编写，返回 true 或者 false，只有返回 true 才进行缓存。unless 属性与 condition 属性相反，满足条件则不进行缓存	@CachePut(cacheNames="cachename", condition= "#userName.length()>2")

需要注意的是，在同一个方法内不能同时使用@CachePut 和@Cacheable 注解，因为它们拥有不同的特性。当@Cacheable 注解跳过方法直接获取缓存时，@CachePut 注解会强制执行方法以更新缓存，这会导致意想不到的情况发生，如当注解都带入了条件属性，就会使得它们彼此排斥。还需要注意的是，@CachePut 注解的 condition 属性设置的缓存条件也不应该依赖于方法返回的结果（如 condition="#result"），因为缓存条件是在方法执行前预先验证的。

3．@CacheEvict

@CacheEvict 注解是@Cachable 注解的反向操作，它负责从给定的缓存中移除一个值。大多数缓存框架都提供了缓存数据的有效期，使用该注解可以显式地从缓存中删除失效的缓存数据。该注解通常用于更新或者删除用户的操作。下面的方法定义从数据库中删除一个用户，而@CacheEvict 注解也完成了相同的工作，从 users 缓存中删除了被缓存的用户。

```
@CacheEvict("users")
public void removeUser(int id) {
    users.remove(id);
}
```

与@Cacheable 注解一样，@CacheEvict 注解也提供了 key 和 condition 属性，通过这些属性可以使用 SpEL 表达式指定自定义的键和条件。

此外，@CacheEvict 注解还具有两个与@Cacheable 注解不同的属性：allEntries 属性定义了是否移除缓存的所有条目，其默认行为是不移除这些条目；beforeInvocation 属性定义了在调用方法之前还是在调用方法之后完成移除操作。与@Cacheable 注解不同的是，在默认情况下，@CacheEvict 注解在方法调用之后运行。

轻松一刻

五彩缤纷的宏观世界都是由质子、中子和电子所构成的，这些粒子因而被称为基本粒子，意指它们是构造世间万物的基本砖块。事实上，基本粒子世界并没有这么简单，任何基本粒子都存在相应的反粒子。当正反物质相遇时，双方就会相互湮灭抵消，发生爆炸并产生巨大能量。在丹·布朗的小说《天使与魔鬼》里，恐怖分子企图从欧洲核子中心盗取 0.25 克反物质，进而欲炸毁整座梵蒂冈城。

1）allEntries 属性

allEntries 是布尔类型的，用来表示是否需要清除缓存中的所有元素。默认值为 false，表示不需要。当指定 allEntries 为 true 时，Spring Cache 将忽略指定的 key，清除缓存中的所有内容。具体使用方法如下：

```
@CacheEvict(cacheNames="users", allEntries=true)
public void delete(String userId) {
    System.out.println("delete user by id: " + userId);
}
```

2）beforeInvocation 属性

需要知道，清除操作默认是在对应方法执行成功后触发的，即方法如果因为抛出异常而未能成功返回时则不会触发清除操作。使用 beforeInvocation 属性可以改变触发清除操作的时间。当指定该属性值为 true 时，Spring 会在调用该方法之前清除缓存中的指定元素。具体使用方法如下：

```
@CacheEvict(cacheNames="users", beforeInvocation=true)
public void delete(String userId) {
    System.out.println("delete user by id: " + userId);
}
```

需要注意的是，在相同的方法上使用@Caheable 和@CacheEvict 注解并使用它们指向相同的缓存没有任何意义，因为这相当于数据被缓存之后又被立即移除了，所以需要避免在同一方法上同时使用这两个注解。

下面通过表 15-3 对@CacheEvict 注解的参数进行详细说明。

表 15-3　@CacheEvict注解参数说明

参　数	说　明	示　例
value/cacheNames	缓存的名称，在 Spring 配置文件中定义，必须指定至少一个	@CachEvict(cacheNames="cachename") 或者 @CachEvict(cacheNames={"cache1","cache2"}
key	缓存的 key，可以为空。如果指定，则按照 SpEL 表达式编写；如果不指定，则默认按照方法的所有参数进行组合	@CachEvict(cacheNames="cachename",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 表达式编写，返回 true 或者 false，只有返回 true 才清空缓存。unless 属性与 condition 属性相反，满足条件则不进行缓存	@CachEvict(cacheNames="cachename",condition="#userName.length()>2")

续表

参　数	说　明	示　例
allEntries	是否清空所有缓存内容，默认为 false。如果指定为 true，则方法调用后将立即清空所有缓存	@CachEvict(cacheNames="cachename", allEntries=true)
beforeInvocation	是否在方法执行前就清空，默认为 false。如果指定为 true，则在方法还没有执行的时候就清空缓存。在默认情况下，如果方法执行抛出异常，则不会清空缓存	@CachEvict(cacheNames="cachename", beforeInvocation=true)

4. @Caching

@Caching 是一个组注解，可以为一个方法定义提供基于@Cacheable、@CacheEvict 或者@CachePut注解的数组。为了方便说明@Caching注解的使用方法，示例定义了 User、Member（会员）和 Visitor（游客）3 个实体类，它们彼此之间有一个简单的层次结构：User 是一个抽象类，而 Member 和 Visitor 类扩展了该类。

在代码清单 15-8 中，UserService 类是一个 Spring 服务 Bean，包含了 getUser()方法。同时声明了两个@Cacheable 注解，并使其指向两个不同的缓存项：members 和 visitors。然后根据两个@Cacheable 注解定义中的条件对方法的参数进行检查，并将对象存储在 members 或 visitors 缓存中。

代码清单 15-8　UseService.java

```
package com.smart.cache.cachegroup;
import com.smart.cache.domain.User;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.cache.annotation.Caching;
import org.springframework.stereotype.Service;
import java.util.HashMap;
import java.util.Map;


@Service(value = "cacheGroupUserServcie")
public class UserService {
    private Map<Integer, User> ppl = new HashMap<Integer, User>();
    {
        ppl.put(1, new Member("1", "w1"));
        ppl.put(2, new Visitor("2", "w2"));
    }

     //@Caching组的使用
    @Caching(cacheable = {
            @Cacheable(value = "members", condition = "#obj instanceof
                T(com.smart.cache.cachegroup.Member)"),
            @Cacheable(value = "visitors", condition = "#obj instanceof
     T(com.smart.cache.cachegroup.Visitor)")
    })
    public User getUser(User obj) {
        return ppl.get(Integer.valueOf(obj.getUserId()));
    }
}
```

5. @CacheConfig

在 Spring 4.0 之前并没有类级别的全局缓存注解。前面我们所了解的注解都是基于方法的，如果在同一个类中需要缓存的方法注解属性都相似，则需要一个个地重复增加。Spring 4.0 增加了@CacheConfig 类级别的注解来解决这个问题。一个典型的使用示例如代码清单 15-9 所示。

代码清单 15-9 UseService.java

```
package com.smart.cache.config;
import com.smart.cache.domain.User;
import org.springframework.cache.annotation.CacheConfig;
import org.springframework.cache.annotation.Cacheable;

@CacheConfig(cacheNames = "users",keyGenerator="MyKeyGenerator ")
public class UserService {
    @Cacheable
    public User findA(User user) {
        ...
    }
   @Cacheable
    public User findB(User user) {
        ...
    }
}
```

可以看到，在@CacheConfig 注解中定义了类级别的缓存 users 和自定义 key 生成器，那么在 findA()和 findB()方法中不再需要重复指定，而是默认使用类级别的定义。

15.2.2 缓存管理器

CacheManager 是 SPI（Service Provider Interface，服务提供程序接口），提供了访问缓存名称和缓存对象的方法，同时也提供了管理缓存、操作缓存和移除缓存的方法。本节将列举 Spring Cache 框架所提供的不同的缓存管理器实现。

1. SimpleCacheManager

通过使用 SimpleCacheManager 可以配置缓存列表，并利用这些缓存进行相关的操作。因为 SimpleCacheManager 是缓存管理器的简化版本，所以在本章的所有示例中都将使用该实现类。下面的代码片段是针对该缓存管理器的一个配置示例。对应缓存的定义，我们使用了 ConcurrentMapCaheFactoryBean 类来对 ConcurrentMapCache 进行实例化，该实例使用了 JDK 的 ConcurrentMap 实现。

```
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean id="users"
                class="org.springframework.cache.concurrent.ConcurrentMapCache
                                    FactoryBean" />
        </set>
```

```
    </property>
</bean>
```

2．NoOpCacheManager

NoOpCacheManager 主要用于测试目的，但实际上它并不缓存任何数据。下面的代码给出了该缓存管理器的配置定义，我们没有为该管理器提供缓存列表，因为它仅仅作为测试目的。

```
<bean id="cacheManager" class="org.springframework.cache.support.NoOpCache Manager" />
```

3．ConcurrentMapCacheManager

ConcurrentMapCacheManager 使用了 JDK 的 ConcurrentMap。它提供了与前面介绍的 SimpleCacheManager 类似的功能，但并不需要像前面那样定义缓存。该缓存定义如下：

```
<bean id="cacheManager"
    class="org.springframework.cache.concurrent.ConcurrentMapCacheManager" />
```

4．CompositeCacheManager

CompositeCacheManager 能够定义多个缓存管理器。当在应用程序上下文中声明<cache:annotation-driven>标记时，它只提供一个缓存管理器，这往往并不能满足用户的需求。而 CompositeCacheManager 定义将多个缓存管理器定义组合在一起，从而扩展了该功能。此外，CompositeCacheManager 还提供了一种机制，通过使用 fallbackToNoOpCache 属性回到 NoOpCacheManager。下面所示的定义是一个 CompositeCacheManager 定义，将一个简单的缓存管理器与 HazelCast 缓存管理器捆绑在一起。简单的缓存管理器定义了 members 缓存，而 HazelCast 缓存管理器则为 visitors 定义了缓存管理器。配置 HazelCast 缓存管理器的详细过程将在本章稍后介绍。下面的示例展示了可在不同的缓存中存储不同类型的对象，而不同的缓存则由不同的缓存管理器进行管理。

```
<bean id="cacheManager"
class="org.springframework.cache.support.CompositeCacheManager">
   <property name="cacheManagers">
      <list>
         <bean class="org.springframework.cache.support.SimpleCacheManager">
            <property name="caches">
               <set>
                  <bean id="members"  class="org.springframework.cache.concurrent
.                                    Concurrent MapCacheFactoryBean" />
               </set>
            </property>
         </bean>
         <bean class="com.hazelcast.spring.cache.HazelcastCacheManager">
            <constructor-arg ref="hazelcast" />
         </bean>
      </list>
   </property>
</bean>
```

15.2.3 使用 SpEL 表达式

在 Spring Cache 注解属性中（如 key、condition 和 unless），Spring 的缓存抽象使用了 SpEL 表达式，从而提供了属性值的动态生成及足够的灵活性。下面代码所示的方法根据用户编码进行了缓存。对于 key 属性，使用了表达式来自定义键的生成。

```
@Cacheable(cacheNames="users",key = "#user.userCode")
public User getUser(User user) {
    return users.get(user.getUserId());
}
```

在下面的代码片段中，还应用了一个条件，以便对那些年龄小于 35 周岁的用户进行缓存。

```
@Cacheable(value = "users", condition = "#user.age < 35")
public User getUser(User user) {
    System.out.println("User with id " + user.getUserId() + " requested.");
    return users.get(user.getUserId());
}
```

SpEL 表达式可基于上下文并通过使用缓存抽象，提供与 root 对象相关联的缓存特定的内置参数。表 15-4 描述了该表达式。

表 15-4 SpEL表达式

名 字	位 置	描 述	示 例
methodName	root 对象	当前被调用的方法名	#root.methodname
method	root 对象	当前被调用的方法	#root.method.name
target	root 对象	当前被调用的目标对象实例	#root.target
targetClass	root 对象	当前被调用的目标对象的类	#root.targetClass
args	root 对象	当前被调用的方法的参数列表	#root.args[0]
caches	root 对象	当前方法调用使用的缓存列表	#root.caches[0].name
Argument name	执行上下文	当前被调用的方法的参数，如 findByUser(User user)，可以通过#user.id 获得参数	#user.id
result	执行上下文	方法执行后的返回值（仅当方法执行之后的判断有效，如'unless'、'cache evict'的 beforeInvocation=false）	#result

15.2.4 基于 XML 的 Cache 声明

如果不想使用注解或者由于其他原因而无法获得项目的源码（如二次开发时经常没有源码的访问权限），也可以用 XML 的方式配置 Spring Cache。其配置方式和 transaction 管理器的 advice 类似，如下例：

```
<!-- 定义需要使用缓存的类 -->
<bean id="userService" class="com.smart.service.UserService"/>
<!-- 缓存定义 -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="users">
    <cache:cacheable method="findUser" key="#userId"/>
```

```
    <cache:cache-evict method="loadUsers" all-entries="true"/>
    </cache:caching>
</cache:advice>
<aop:config>
<aop:advisor advice-ref="cacheAdvice"
    pointcut="execution(*com.smart.service.UserService.*(..))"/>
</aop:config>
```

如上配置文件为 UserService 开启了缓存。使用 cache:advice 定义包装了两个需要使用缓存的方法，其中 findUser()定义了 Cacheable，而 loadUsers()定义了 CacheEvict，并且定义了公共的缓存 users。

aop:config 定义了 cacheAdvice 的切入点（关于 AOP 的定义，具体请参考第 8 章）。XML 声明式配置支持所有注解的方法，因此二者之间可以很容易地替换，当然也可以在项目中同时使用这两种方式。

15.2.5　以编程方式初始化缓存

在实际项目中，有时可能需要在使用之前就完成缓存的初始化。最典型的示例是当启动并且运行应用程序时将数据加载到缓存中。实现该方法很简单，首先访问缓存管理器，然后将数据手工加载到不同的缓存中（这些缓存根据缓存名称进行区分）。在下面的示例中，当初始化应用程序上下文时，将用户列表加载到缓存区域中。

代码清单 15-10 展示了在 Sping Bean 的@PostConstrut 注解方法中初始化缓存 users。此外，该 Bean 还包含了 getUser()方法，并且已被注解用于缓存操作。

代码清单 15-10　UserService.java

```
package com.smart.cache.initcache;
import com.smart.cache.domain.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.Cache;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
import javax.annotation.PostConstruct;
import java.util.HashMap;
import java.util.Map;

@Service(value = "initUserServcie")
public class UserService {
   private Map<Integer, User> users = new HashMap<Integer, User>();
   {
      users.put(1, new User("1", "w1"));
      users.put(2, new User("2", "w2"));
   }

   private CacheManager cacheManager;

   @Autowired
   public void setCacheManager(CacheManager cacheManager) {
```

```
        this.cacheManager = cacheManager;
    }

    @PostConstruct
    public void setup() {
        Cache usersCache = cacheManager.getCache("users");
        for (Integer key : users.keySet()) {
            usersCache.put(key, users.get(key));
        }
    }

    @Cacheable(value = "users")
    public User getUser(int id) {
        System.out.println("User with id " + id + " requested.");
        return users.get(id);
    }
}
```

接下来创建 UserService 类作为 Spring 服务 Bean，并且自动注入 CacheManager，然后在@PostConstruct 中通过键值把所有数据放置到缓存中，该键值将用于检索。

下面通过基于 Java 类的配置方式准备好 Spring 容器配置信息，如代码清单 15-11 所示。

代码清单 15-11　ApplicationConfig.java

```
package com.smart.cache.initcache;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCache;
import org.springframework.cache.support.SimpleCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import java.util.Arrays;

@Configuration
@ComponentScan(basePackages = {"com.smart.cache"})
@EnableCaching
public class ApplicationConfig {
    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Arrays.asList(new ConcurrentMapCache("users")));
        return cacheManager;
    }
}
```

用 UserMain 测试以上缓存实现的效果，如代码清单 15-12 所示。

代码清单 15-12　UserMain.java

```
package com.smart.cache.initcache;
import com.smart.cache.domain.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
public class UserMain {
    public static void main(String... args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(ApplicationConfig.class);
        UserService userService =  (UserService) context.getBean("initUserServcie");

        User userFetch1 = userService.getUser(1);
        System.out.println(userFetch1);
        User userFetch2 = userService.getUser(2);
        System.out.println(userFetch2);
    }
}
```

在代码清单 15-12 中，首先从上下文中查找 initUserServcie Bean，然后连续调用 getUser(1)和 getUser(2)方法。可以看到，此时已查找到缓存中的相关信息，如下：

```
User{id=1, name='w1', age=0}
User{id=2, name='w2', age=0}
```

15.2.6　自定义缓存注解

之前介绍的@Caching 组合也许会让方法上的注解显得比较杂乱，Spring 提供了自定义注解，可把这些注解组合到一个注解类中，从而解决这个问题，如下代码所示：

```
@Caching(
    put = {
            @CachePut(cacheNames = "user", key = "#user.id"),
            @CachePut(cacheNames = "user", key = "#user.username"),
            @CachePut(cacheNames = "user", key = "#user.email")
    }
)
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface UserSaveCache {
}
```

这样只需在方法上使用如下代码即可，整个代码显得比较干净。

```
@UserSaveCache
public User save(User user)
```

15.3　配置 Cache 存储

在企业级应用中往往会有更复杂的功能和性能需求，所以在日常开发过程中，大部分情况下会使用第三方的缓存实现，而不是 SimpleCacheManager 的简单实现。在企业级 Java 领域，Spring 缓存提供了与不同缓存框架的集成支持。

15.3.1 EhCache

EhCache 是广泛使用的 Java 开源缓存框架之一，本节将介绍如何通过 Spring Cache 来集成 EhCache，以便在项目中便捷地使用。

在项目中增加 EhCache 的依赖。我们选择的是 2.8.3 版本，该版本最值得关注的新属性包括 Off-Heap 存储和 JSR-107 兼容。

```
<dependency>
   <groupId>net.sf.ehcache</groupId>
   <artifactId>ehcache</artifactId>
   <version>2.8.3</version>
</dependency>
```

接着需要在 applicationContext-ehcache 配置文件中对 EhCache 进行如下配置：

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager"
     p:cacheManager="ehcache"/>
<bean id="ehcache" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
    p:configLocation="classpath:ehcache.xml"/>
```

在配置文件中声明了 cacheManager Bean。通过使用 EhCache，定义 cacheManager 是非常简单和直接的。它包含了另外一个名为 ehcache 的 Bean，并使用配置文件 ehcache.xml 对其进行配置。下面给出一个最简单的 ehcache.xml 配置文件，其中仅包含了 users 缓存的定义。

```
<ehcache>
   <cache name="users"  maxElementsInMemory="1000" />
</ehcache>
```

Ehcache 还提供了 TTL 或者 Eviction Policy 之类的功能，并且可以对这些功能进行配置。而相关配置应该直接通过缓存提供程序完成，因为缓存抽象没有为这些功能提供任何配置（这些功能不可能被不同的提供程序所支持，如 JDK 的 ConcurrentMap）。如果想进一步配置，请参考 EhCache 的相关文档。

15.3.2 Guava

从 Spring 4.0 开始支持 Guava，Guava 是 Google 贡献的一个开源公共库（Guava 非常好用，建议开发者了解并使用之），它提供了缓存功能。下面的代码片段显示了针对 Guava 的 Maven 依赖项，版本是 18.0。

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>18.0</version>
</dependency>
```

配置 GuavaCacheManager 很简单，只需定义 cacheManager Bean 就可以获取配置并运行。同时，没必要定义缓存，因为它们将在需要时被创建。

```
<bean id="cacheManager" class="org.springframework.cache.guava.GuavaCacheManager" />
```

15.3.3　HazelCast

HazelCast 是业界流行的 Java 分布式内存网格框架之一。与其他框架一样，HazelCast 也提供了基于 Spring 缓存抽象的缓存管理器，这里使用的是其 3.3 版本。

```
<dependency>
    <groupId>com.hazelcast</groupId>
     <artifactId>hazelcast-all</artifactId>
     <version>3.3</version>
</dependency>
```

目前，HazellcastCacheManager 位于 hazelcast-all 的工具包中，但并没有包括在 spring-context-support 中。如下面的代码所示，HazellcastCacheManager 引用了另一个 Bean 作为其打包的 cacheManager。

```
<bean id="cacheManager" class="com.hazelcast.spring.cache.HazelcastCacheManager">
     <constructor-arg ref="hazelcast" />
</bean>
```

借助 hazelcast Bean，通过创建普通的 users 缓存为 HazelCast 完成配置。

```
<hz:hazelcast id="hazelcast">
  <hz:config>
    <hz:map name="users">
      <hz:map-store enabled="true" class-name="com.smart.cache.domain.User
          ↪" write-delay-seconds="0"/>
    </hz:map>
  </hz:config>
</hz:hazelcast>
```

HazelCast 需要实体类实现 Serializable 接口，并包含默认的构造函数。

15.3.4　GemFire

GemFire 是一个需要商业许可的数据管理平台，它提供了基于分布式体系结构的数据访问功能。GemFire 同时提供了一个内存数据网格，具备极高的吞吐量、低延迟的数据访问及可扩展性。

配置 GemFire 也很简单，如下代码所示：

```
<gfe:cache id="gemfire-cache"/>
<bean id="cacheManager" class="org.springframework.data.gemfire.support.Gemfire
      ↪CacheManager" p:cache-ref="gemfire-cache">
</bean>
```

关于 GemFire，如果读者想深入了解，则可以参考 Spring Data 项目中关于 GemFire 的描述（可参考官方链接：http://docs.spring.io/spring-data/gemfire/docs/1.5.2.RELEASE/reference/html/）。

15.3.5　JSR-107 Cache

从 Spring 4.0 开始全面支持 JSR-107 规范，该规范包含@CacheResult、@CacheEvict、

@CacheRemove 和 @CacheRemoveAll 及配套的 @CacheDefaults、@CacheKey 和 @CacheValue 注解。无须把缓存存储迁移到 JSR-107 中，就可以使用 JSR-107 规范的缓存注解，因为 Spring 的缓存抽象为 JSR-107 提供了默认的 CacheResolver 和 KeyGenerator 实现。

表 15-5 梳理了 Spring 缓存注解和相对应的 JSR-107 规范注解之间的一些差异。

表 15-5 Spring Cache和JSR-107 的对比

Spring	JSR-107	备 注
@Cacheable	@CacheResult	基本一致，但是@CacheResult 提出了缓存异常的概念，可以通过 exceptionCacheName 属性指定缓存异常，当异常再次发生时会被抛出，而不会再去调用方法
@CachePut	@CachePut	Spring 会在调用更新方法后缓存结果，而 JCache 需要在方法入参中传递一个@CacheValue 注解参数，JCache 允许通过注解属性 afterInvocation 指定缓存在方法执行前或执行后更新缓存
@CacheEvict	@CacheRemove	@CacheRemove 支持当调用方法抛出异常时，可以按照条件移除缓存
@CacheEvict (allEntries=true)	@CacheRemoveAll	和@CacheRemove 一样
@CacheConfig	@CacheDefaults	类级别的注解，用于配置类级别的通用属性

Java Caching 定义了 4 个核心接口，分别是 CachingProvider、CacheManager、Cache 和 Entry。下面对这 4 个核心接口进行简单介绍。

- CachingProvider 定义了创建、配置、获取、管理和控制多个 CacheManager。一个应用可以在运行期访问多个 CachingProvider。
- CacheManager 定义了创建、配置、获取、管理和控制多个唯一命名的 Cache，这些 Cache 存在于 CacheManager 的上下文中。一个 CacheManager 仅被一个 CachingProvider 所拥有。
- Cache 是一个类似 Map 的数据结构并存储以 key 为索引的值。一个 Cache 仅被一个 CacheManager 所拥有。
- Entry 是一个存储在 Cache 中的键值对。每个存储在 Cache 中的条目都有一个定义的有效期，即 Expiry Duration。一旦超过这个时间，条目即为过期状态。一旦过期，条目将不可访问。缓存有效期可以通过 ExpiryPolicy 设置。

JCache 的 javax.cache.annotation.CacheResolver 和 Spring Cache 的 CacheResolver 接口类似，唯一不同的是 JCache 只支持单个 Cache，这点从注解属性命名上就可以看出，Spring 是 cacheNames，而 JCache 是 cacheName。一般情况下通过注解参数中的定义去获取缓存；如果未指定，则 JCache 默认取入参的第一个参数作为缓存名称。

CacheResolver 的实例从 CacheResolverFactory 中获取，可以为每个方法定制不同的 CacheResolver，如下：

```
@CacheResult(cacheNames="users",cacheResolverFactory=MyCacheResolverFactory. class)
public User finduser(User user)
```

JCache 的生成器和 Spring 的 KeyGenerator 一样，JCache 定义了 javax.cache.annotation.CacheKeyGenerator 接口。如果没有在参数中注入@CacheKey，则默认取所有的参数用于生成 key，这点和 Spring Cache 也极为类似，如下：

```
@Cacheable(cacheNames="users", key="#userId")
public User finduser(User user, boolean checkLogout)
@CacheResult(cacheName="users")
public User finduser(@CacheKey user user, boolean checkLogout)
```

JCache 可以管理在注解方法中抛出的异常，以便阻止缓存的更新。它也可以根据缓存指定的异常，在下次抛出异常时不再执行方法本身。

假设有一个错误的入参被传入，而会导致抛出 InvalidUserNotFoundException 异常，那么在下次该错误入参被再次传入时，Spring 会直接抛出异常而不会再调用方法本身，如下：

```
@CacheResult(cacheName="users", exceptionCacheName="failures"
        cachedExceptions = InvalidUserNotFoundException.class)
public User findUser(User user)
```

JSR-107 的缓存实现同样可以使用 Spring 的缓存抽象。JCache 的缓存实现在 org.springframework.cache.jcache 包中。要使用它，只需要配置相应的缓存管理器即可，如下：

```
<bean id="cacheManager"
    class="org.springframework.cache.jcache.JCacheCacheManager"
    p:cache-manager-ref="jCacheManager"/>
<!-- JSR-107 cache manager  -->
<bean id="jCacheManager" .../>
```

15.4　实战经验

1. 非 public 方法问题

和内部调用问题类似，非 public 方法如果想实现基于注解的缓存，则必须采用基于 AspectJ 的 AOP 机制。这里限于篇幅不再详述，可参考第 8 章。

2. 基于 Proxy 的 Spring AOP 带来的内部调用问题

前面介绍了 Spring Cache 的原理，即它是基于动态生成子类的代理机制来对方法的调用进行切面的，这里的关键点是对象的引用问题。如果对象的方法是内部调用（this 引用）而不是外部引用，则会导致代理失效，那么切面就失效，也就是说上面定义的各种注解，包括@Cacheable、@CachePut 和@CacheEvict 都会失效。来看下面的示例：

```
public User getUsertByName2(String userName) {
    return getUserByName(userName);
}

// 使用了一个缓存，名为 users
@Cacheable(cacheNames ="users")
public User getUserByName(String userName) {
```

```
    //方法内部实现不考虑缓存逻辑，直接实现业务
    return getFromDB(userName);
}
```

上面定义了一个新的方法 getUserByName2()，其自身调用了 getUserByName() 方法，这时发生的是内部调用（this），所以没有使用代理类，导致 Spring Cache 失效。要避免这个问题，就要避免对缓存方法的内部调用，或者避免使用基于代理机制的 AOP 模式，也可以使用基于 AspectJ 的 AOP 模式来解决这个问题。如果想使用基于代理机制的 AOP 模式，要想避免此问题，则可以参考 7.5.4 节介绍的方法。

3．@CacheEvict 的可靠性问题

我们看到，@CacheEvict 注解有一个属性 beforeInvocation，默认为 false，即默认情况下都是在实际的方法执行完成后才对缓存进行清空操作。期间如果执行方法出现异常，则会导致缓存清空而不被执行。

```
//清空 users 缓存
@CacheEvict(cacheNames ="users",allEntries=true)
 public void reload() {
    throw new RuntimeException();
 }
```

注意上面的代码，在 reload 的时候抛出了运行期异常，这会导致清空缓存失败。

4．运行期开发

有的时候，在代码迁移、调试或者部署的时候，恰好没有可运行的缓存服务容器，比如 MemCache 还不具备条件、GemFire 还没有安装好等，如果这时候想调试代码，基本上非常困难。这里有一个办法，在不具备缓存条件的时候，在不更改代码的前提下，禁用缓存。

方法就是修改 spring*.xml 配置文件，设置一个找不到缓存就不执行任何操作的标志位，如下：

```
<bean id="cacheManager"
class="org.springframework.cache.support.CompositeCacheManager">
    <property name="cacheManagers">
        <list>
            <ref bean="jdkCache"/>
            <ref bean="gemfireCache"/>
        </list>
    </property>
    <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

15.5 小结

本章介绍了缓存的基本概念，以及 Spring 通过其缓存抽象所提供的相关功能。同时介绍了 5 个可用来缓存数据或者从缓存中移除数据的重要注解（@Cacheable、

@CacheEvict、@CachePut、@Caching、@CacheConfig）。此外还介绍了采用 XML 声明式和注解式两种方式来使用 Spring Cache。列举了可抽象使用的不同缓存管理器，同时通过 SpEL 示例强调了定义缓存存储时表达式的重要性。

我们还学习了在应用程序启动期间自动初始化缓存，而这恰恰是企业应用程序中常见的用例。最后一节列举了主流的缓存程序实现，如 EhCache、Guava 和 HazelCast、GemFire、JSR-107，并详细介绍了它们与 Spring Cache 如何集成。最后笔者总结了在实际项目中常见的一些问题，并加以分析。本章对 Spring Cache 的源码和原理也有一定的分析，相信读者在理解之后，也能基于 Spring Cache 抽象定制出符合自己应用的缓存方案。

第 16 章

任务调度和异步执行器

任务调度是大多数应用系统的常见需求之一，直接自己编写基于线程的调度程序，不但容易出错，而且实现难度很大。借助巨人的肩膀，可以站得更高、看得更远。Quartz是任务调度领域享誉盛名的开源框架，Spring提供了集成Quartz的功能，可以让开发人员以更面向Spring的方式创建基于Quartz的任务调度应用。此外，Spring也为JDK Timer、Java 5.0和Executor提供了有益的支持。

本章主要内容：

- Quartz知识和Spring的支持
- JDK Timer知识和Spring的支持
- Executor知识和Spring所提供的抽象
- 在实际应用中开发任务调度程序所需注意的问题

本章亮点：

- 比较详细地讲述了Quartz框架
- 在实际应用中开发任务调度程序的实战经验

16.1 任务调度概述

各种企业应用几乎都会遇到任务调度的需求，以论坛为例，每隔半小时生成精华文章的RSS文件，每天凌晨统计论坛用户的积分排名，每隔30分钟对锁定到期的用户进行解锁。对于一个典型的企业应用系统来说，每月1日凌晨统计上个月各部门的业务数据生成月报表，每隔半小时查询用户是否有快到期的待处理业务等，这样的例子俯拾皆是、不胜枚举。

以上所举调度场景的核心都是以时间为关注点的调度，即在特定的时间点执行指定

的操作。如果将任务调度的范围稍微扩大一些，则还应该包括资源上的调度。如 Web Server 在接收到请求时，会立即创建一个新的线程服务该请求。但资源是有限的，无限制地使用必然耗尽亏空，大多数系统都要对资源使用进行控制。首先必须限制服务线程的最大数目；其次可以考虑使用线程池以便共享服务的线程资源，降低频繁创建、销毁线程的消耗。

任务调度本身涉及多线程并发、运行时间规则制定及解析、运行现场保持与恢复、线程池维护等诸多方面的工作。如果直接使用自定义线程这种刀耕火种的原始办法，则开发任务调度程序是一项颇具挑战性的工作。Java 开源的好处就是，领域问题都能找到现成的解决方案。

OpenSymphony 所提供的 Quartz 自 2001 年发布以来，已经被众多项目作为任务调度的解决方案。Quartz 在提供灵活性的同时并未牺牲其简单性，它所提供的强大功能使开发者可以轻松应对绝大多数任务调度的功能需求。

Sun 从 Java 1.3 开始，通过 java.util.Timer 和 TimerTask 提供了简单的调度功能，允许用户调度一个按固定时间间隔运行的任务。

针对前面提到的资源调度的要求，Java 5.0 通过 java.util.concurrent 这个新包中的若干类和接口，提供了方便的线程池调用功能。

Spring 对以上三者都提供了支持，通过 Spring 所提供的一系列 FactoryBean，可以很容易地创建任务调度框架的实例。此外，Spring 还提供了几个工具类，可以将某个具体 Bean 的方法直接作为被调度的任务，从而简化了任务的定义。Spring 还提供了支持线程池的调度执行器，它提供了一个抽象层，屏蔽了 Java 1.3、Java 1.4、Java 5.0 和 Java EE 的环境差异。

本章选取 Quartz 1.8.6 版本进行示例说明（有关 Quartz 的详细说明，可查看其官网 http://www.quartz-scheduler.org/）。需要在 pom.xml 中加入如下版本依赖：

```
<dependency>
    <groupId>org.quartz-scheduler</groupId>
    <artifactId>quartz</artifactId>
    <version>1.8.6</version>
</dependency>
```

16.2　Quartz 快速进阶

Quartz 是开源任务调度框架中的翘楚，它提供了强大的任务调度机制。Quartz 允许开发人员灵活地定义触发器的调度时间表，并可对触发器和任务进行关联映射。此外，Quartz 提供了调度运行环境的持久化机制，可以保存并恢复调度现场，即使系统因故障关闭，任务调度现场数据也不会丢失。此外，Quartz 还提供了组件式的侦听器、各种插件、线程池等功能。

16.2.1 Quartz 基础结构

Quartz 对任务调度的领域问题进行了高度抽象，提出了调度器、任务和触发器这 3 个核心的概念，并在 org.quartz 中通过接口和类对核心概念进行了描述。

- Job：是一个接口，只有一个方法 void execute(JobExecutionContext context)，开发者通过实现该接口来定义需要执行的任务，JobExecutionContext 类提供了调度上下文的各种信息。Job 运行时的信息保存在 JobDataMap 实例中。
- JobDetail：Quartz 在每次执行 Job 时，都重新创建一个 Job 实例，所以它不是直接接收一个 Job 实例，而是接收一个 Job 实现类，以便运行时通过 newInstance() 的反射调用机制实例化 Job。因此需要通过一个类来描述 Job 的实现类及其他相关的静态信息，如 Job 名称、描述、关联监听器等信息，而 JobDetail 承担了这一角色。通过该类的构造函数 JobDetail(java.lang.String name, java.lang.String group, java.lang.Class jobClass)，可以更具体地了解它的功用。该构造函数要求指定 Job 的实现类，以及任务在 Scheduler 中的组名和 Job 名称。
- Trigger：是一个类，描述触发 Job 执行的时间触发规则。主要有 SimpleTrigger 和 CronTrigger 这两个子类。当仅需要触发一次或者以固定间隔周期性执行时，SimpleTrigger 是最适合的选择；而 CronTrigger 则可以通过 Cron 表达式定义出各种复杂的调度方案，如每天早晨 9:00 执行，每周一、周三、周五下午 5:00 执行等。
- Calendar：org.quartz.Calendar 和 java.util.Calendar 不同，它是一些日历特定时间点的集合（可以简单地将 org.quartz.Calendar 看作 java.util.Calendar 的集合——java.util.Calendar 代表一个日历时间点，若无特殊说明，后面的 Calendar 即指 org.quartz.Calendar）。一个 Trigger 可以和多个 Calendar 关联，以便排除或包含某些时间点。假设安排每周一早晨 10:00 执行任务，但是如果遇到法定节假日，则不执行任务，这时就需要在 Trigger 触发机制的基础上使用 Calendar 进行定点排除。针对不同的时间段类型，Quartz 在 org.quartz.impl.calendar 包下提供了若干个 Calendar 的实现类，如 AnnualCalendar、MonthlyCalendar、WeeklyCalendar 分别针对每年、每月和每周进行定义。
- Scheduler：代表一个 Quartz 的独立运行容器，Trigger 和 JobDetail 可以注册到 Scheduler 中，二者在 Scheduler 中拥有各自的组及名称。组及名称是 Scheduler 查找定位容器中某一对象的依据，Trigger 的组及名称的组合必须唯一，JobDetail 的组及名称的组合也必须唯一（但可以和 Trigger 的组及名称相同，因为它们是不同类型的，处在不同的集合中）。Scheduler 定义了多个接口方法，允许外部通过组及名称访问和控制容器中的 Trigger 和 JobDetail。Scheduler 可以将 Trigger 绑定到某一 JobDetail 中，这样，当 Trigger 被触发时，对应的 Job 就被执行。一个 Job 可以对应多个 Trigger，但一个 Trigger 只能对应一个 Job。可以通过

SchedulerFactory 创建一个 Scheduler 实例。Scheduler 拥有一个 SchedulerContext，保存着 Scheduler 上下文信息，可以对照 ServletContext 来理解 SchedulerContext。Job 和 Trigger 都可以访问 SchedulerContext 内的信息。SchedulerContext 内部通过一个 Map，以键值对的方式维护这些上下文数据。SchedulerContext 为保存和获取数据提供了多个 put()和 getXxx()方法。可以通过 Scheduler#getContext()方法获取对应的 SchedulerContext 实例。

- ThreadPool：Scheduler 使用一个线程池作为任务运行的基础设施，任务通过共享线程池中的线程来提高运行效率。

Job 有一个 StatefulJob 子接口，代表有状态的任务。该接口是一个没有方法的标签接口，其目的是让 Quartz 知道任务的类型，以便采用不同的执行方案。无状态任务在执行时拥有自己的 JobDataMap 复制，对 JobDataMap 的更改不会影响下次的执行。而有状态任务共享同一个 JobDataMap 实例，每次任务执行对 JobDataMap 所做的更改会保存下来，后面的执行可以看到这个更改，即每次执行任务后都会对后面的执行产生影响。

正因为这个原因，无状态的 Job 可以并发执行，而有状态的 StatefulJob 不能并发执行。这意味着如果前次的 StatefulJob 还没有执行完毕，则下次的任务将阻塞等待，直到前次任务执行完毕。有状态任务比无状态任务需要考虑更多的因素，程序往往更复杂，因此，除非必要，应尽量避免使用有状态的 Job。

如果 Quartz 使用了数据库持久化任务调度信息，则无状态的 JobDataMap 仅会在 Scheduler 注册任务时保存一次，而有状态任务对应的 JobDataMap 在每次执行任务后都会进行保存。

Trigger 自身也可以拥有一个 JobDataMap，其关联的 JobDataMap 可以通过 JobExecutionContext#getTrigger().getJobDataMap()方法获取。不管是有状态还是无状态的任务，在任务执行期间对 Trigger 的 JobDataMap 所做的更改都不会进行持久化，即不会对下次任务的执行产生影响。

Quartz 拥有完善的事件和监听体系，大部分组件都拥有事件，如任务执行前事件、任务执行后事件、触发器触发前事件、触发器触发后事件、调度器开始事件、调度器关闭事件等，可以注册相应的监听器处理感兴趣的事件。

图 16-1 描述了 Scheduler 的内部组件结构。SchedulerContext 提供 Scheduler 全局可见的上下文信息，每个任务都对应一个 JobDataMap，虚线框中的 JobDataMap 表示有状态的任务。

一个 Scheduler 可以拥有多个 Trigger 和多个 JobDetail，它们可以分到不同的组中。在注册 Trigger 和 JobDetail 时，如果不显式指定所属的组，那么 Scheduler 将放入默认组中，默认组的组名为 Scheduler.DEFAULT_GROUP。组名和名称组成了对象的全名，同一类型对象（Job 或 Trigger）的全名不能相同。

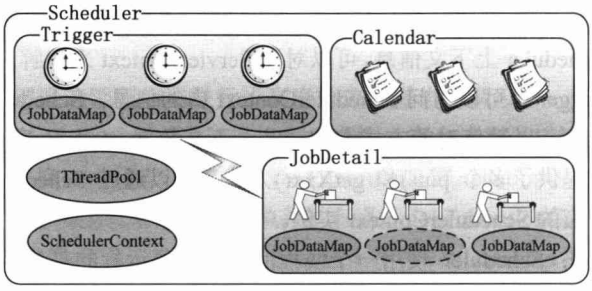


图 16-1　Scheduler 内部组件结构图

Scheduler 本身就是一个容器，它维护着 Quartz 的各种组件并实施调度的规则。Scheduler 还拥有一个线程池，线程池为任务提供执行线程。这比执行任务时简单地创建一个新线程要拥有更高的效率，同时通过共享机制可以减少资源的占用。基于线程池组件的支持，对于繁忙度高、压力大的任务调度，Quartz 可以提供良好的伸缩性。

提示

Quartz 完整下载包 examples 目录下拥有 10 多个实例，它们是快速掌握 Quartz 应用很好的实例。

16.2.2　使用 SimpleTrigger

SimpleTrigger 拥有多个重载的构造函数，用于在不同场合下构造出对应的实例。

- SimpleTrigger(String name, String group)：通过该构造函数指定 Trigger 所属组和名称。
- SimpleTrigger(String name, String group, Date startTime)：除指定 Trigger 所属组和名称外，还可以指定触发的开始时间。
- SimpleTrigger(String name, String group, Date startTime, Date endTime, int repeatCount, long repeatInterval)：除指定以上信息外，还可以指定结束时间、重复执行次数、时间间隔等参数。
- SimpleTrigger(String name, String group, String jobName, String jobGroup, Date startTime, Date endTime, int repeatCount, long repeatInterval)：这是最复杂的一个构造函数，在指定触发参数的同时，通过 jobGroup 和 jobName，使该 Trigger 和 Scheduler 中的某个任务关联起来。

通过实现 org.quartz.Job 接口，可以使 Java 类化身为可调度的任务。代码清单 16-1 提供了 Quartz 任务的一个示例。

代码清单 16-1　SimpleJob：简单的Job实现类

```
package com.smart.basic.quartz;
import java.util.Date;
import org.quartz.Job;
```

```
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
public class SimpleJob implements Job {
    public void execute(JobExecutionContext jobCtx)                    实现 Job 接口方法
                    throws JobExecutionException {①
        System.out.println(jobCtx.getTrigger().getName()+ " triggered. time is:" +
(new Date()));
    }
}
```

这个演示类的 execute(JobExecutionContext context) 方法只有一条简单的输出语句，可以在这个方法中包含任何想要执行的代码。下面通过 SimpleTrigger 对 SimpleJob 进行调度，如代码清单 16-2 所示。

代码清单 16-2　SimpleTriggerRunner：使用SimpleTrigger进行调度

```
package com.smart.basic.quartz;
import java.util.Date;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleTrigger;
import org.quartz.impl.StdSchedulerFactory;
public class SimpleTriggerRunner {
    public static void main(String args[]) {
        try {
                                                    创建一个 JobDetail 实例，指定
                                                    SimpleJob
            JobDetail jobDetail = new JobDetail("job1_1","jgroup1", SimpleJob.class); ①

                                                    通过 SimpleTrigger 定义调度规则：马上启
                                                    动，每 2 秒运行一次，共运行 100 次
            SimpleTrigger simpleTrigger = new SimpleTrigger("trigger1_1","tgroup1"); ②
            simpleTrigger.setStartTime(new Date());
            simpleTrigger.setRepeatInterval(2000);
            simpleTrigger.setRepeatCount(100);          通过 SchedulerFactory
                                                        获取一个调度器实例
            SchedulerFactory schedulerFactory = new StdSchedulerFactory(); ③
            Scheduler scheduler = schedulerFactory.getScheduler();

            scheduler.scheduleJob(jobDetail, simpleTrigger); ④ ← 注册并进行调度
            scheduler.start();⑤ ← 调度启动
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

首先在①处通过 JobDetail 封装 SimpleJob，同时指定 Job 在 Scheduler 中的所属组及名称。在这里，组名为 jgroup1，而名称为 job1_1。

然后在②处创建一个 SimpleTrigger 实例，指定该 Trigger 在 Scheduler 中的所属组及名称。接着设置调度的时间规则。

最后创建 Scheduler 实例，并将 JobDetail 和 Trigger 实例注册到 Scheduler 中。在这

里，通过 StdSchedulerFactory 获取一个 Scheduler 实例，并通过 scheduleJob(JobDetail jobDetail, Trigger trigger)完成两件事。

（1）将 JobDetail 和 Trigger 注册到 Scheduler 中。

（2）用 Trigger 对 JobDetail 中的任务进行调度。

在 Scheduler 启动后，Trigger 将定期触发并执行 SimpleJob 的 execute(JobExecution Context jobCtx)方法，每 10 秒重复一次，直到任务被执行 100 次。

还可以通过 SimpleTrigger 的 setStartTime(java.util.Date startTime)和 setEndTime(java.util.Date endTime)方法指定运行的时间范围。当运行次数和时间范围产生冲突时，超过时间范围的任务不被执行。如可以通过 simpleTrigger.setStartTime(new Date(System.Current TimeMillis() + 60000L))方法指定 60 秒后开始运行。

除了通过 scheduleJob(jobDetail, simpleTrigger)方法建立 Trigger 和 JobDetail 的关联外，还可以通过如下方式建立关联：

```
JobDetail jobDetail = new JobDetail("job1_1","jgroup1", SimpleJob.class);
SimpleTrigger simpleTrigger = new SimpleTrigger("trigger1_1","tgroup1");
…
simpleTrigger.setJobGroup("jgroup1");①-1 ← 指定关联的 Job 组名
simpleTrigger.setJobName("job1_1");①-2 ← 指定关联的 Job 名称
scheduler.addJob(jobDetail, true); ② ← 注册 JobDetail
scheduler.scheduleJob(simpleTrigger); ③ ← 注册指定了关联 JobDetail 的 Trigger
```

在这种方式中，Trigger 通过指定 Job 所属组及 Job 名称，使用 Scheduler 的 scheduleJob(Trigger trigger)方法注册 Trigger。在这里，有两个值得注意的地方。

（1）通过这种方式注册的 Trigger 必须已经指定了 Job 组名和 Job 名称，否则调用 scheduleJob(simpleTrigger)方法将抛出异常。

（2）引用的 JobDetail 对象必须已经存在于 Scheduler 中，即代码中①、②和③的顺序不能随意变动。

实战经验

在构造 Trigger 实例时，可以考虑使用 org.quartz.TriggerUtils 工具类，该工具类不但提供了众多获取特定时间的方法，还拥有多个获取常见 Trigger 的方法。如 makeSecondlyTrigger(String trigName)方法将创建一个每秒执行一次的 Trigger，makeWeeklyTrigger(String trigName, int dayOfWeek, int hour, int minute)方法将创建一个每星期某一特定时间点执行一次的 Trigger，而 getEvenMinuteDate(Date date)方法将返回某一时间点一分钟以后的时间。

16.2.3 使用 CronTrigger

CronTrigger 能够提供比 SimpleTrigger 更有具体实际意义的调度方案，调度规则基

于 Cron 表达式。CronTrigger 支持日历相关的周期时间间隔（比如每月第一个周一执行），而不是简单的周期时间间隔。因此，相对于 SimpleTrigger 而言，CronTrigger 在使用上也要复杂一些。

1. Cron 表达式

Quartz 使用类似于 Linux 下的 Cron 表达式定义时间规则。Cron 表达式由 6 或 7 个空格分隔的时间字段组成，如表 16-1 所示。

表 16-1　Cron 表达式时间字段

位　置	时间域名	允　许　值	允许的特殊字符
1	秒	0-59	, - * /
2	分钟	0-59	, - * /
3	小时	0-23	, - * /
4	日期	1-31	, - * ? / L W C
5	月份	1-12	, - * /
6	星期	1-7	, - * ? / L C #
7	年（可选）	空值 1970-2099	, - * /

Cron 表达式的时间字段除允许设置数值外，还可以使用一些特殊的字符，提供列表、范围、通配符等功能，详细介绍如下。

- 星号（*）：可用在所有字段中，表示对应时间域的每一个时刻。例如，*在分钟字段时，表示“每分钟”。
- 问号（?）：该字符只在日期和星期字段中使用，它通常指定为“无意义的值”，相当于占位符。
- 减号（-）：表达一个范围。如在小时字段中使用“10-12”，则表示从 10 点到 12 点，即 10,11,12。
- 逗号（,）：表示一个列表值。如在星期字段中使用“MON,WED,FRI”，则表示星期一、星期三和星期五。
- 斜杠（/）：x/y 表达一个等步长序列，x 为起始值，y 为增量步长值。如在分钟字段中使用 0/15，则表示为 0,15,30 和 45 秒；而 5/15 在分钟字段中表示 5,20,35,50。用户也可以使用*/y，它等同于 0/y。
- L：该字符只在日期和星期字段中使用，代表“Last”的意思，但它在两个字段中的意思不同。如果 L 用在日期字段中，则表示这个月份的最后一天，如 1 月 31 日，非闰年 2 月 28 日；如果 L 用在星期字段中，则表示星期六，等同于 7（注意，这里的规则是星期六为一星期的最后一天）。但是，如果 L 出现在星期字段里，而且前面有一个数字 N，则表示“这个月的最后 *N* 天”。例如，6L 表示该月的最后一个星期五。
- W：该字符只能出现在日期字段里，是对前导日期的修饰，表示离该日期最近的工作日。例如，15W 表示离该月 15 日最近的工作日，如果该月 15 日是星期

六，则匹配 14 日星期五；如果 15 日是星期日，则匹配 16 日星期一；如果 15 日是星期二，那结果就是 15 日星期二。但必须注意关联的匹配日期不能跨月，如用户指定 1W，如果 1 日是星期六，结果匹配的是 3 日星期一，而非上个月最后一天。W 字符串只能指定单一日期，而不能指定日期范围。

- LW 组合：在日期字段中可以组合使用 LW，它的意思是当月的最后一个工作日。
- 井号（#）：该字符只能在星期字段中使用，表示当月的某个工作日。如 6#3 表示当月的第三个星期五（6 表示星期五，#3 表示当前的第三个），而 4#5 表示当月的第五个星期三。假设当月没有第五个星期三，则忽略不触发。
- C：该字符只在日期和星期字段中使用，代表“Calendar”的意思。它的意思是计划所关联的日期，如果日期没有被关联，则相当于日历中的所有日期。例如，5C 在日期字段中相当于 5 日以后的那一天，1C 在星期字段中相当于星期日后的第一天。

Cron 表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。表 16-2 给出了一些完整的 Cron 表示式示例。

表 16-2　Cron表示式示例

表 示 式	说　明
"0 0 12 * * ? "	每天 12:00 运行
"0 15 10 ? * *"	每天 10:15 运行
"0 15 10 * * ?"	每天 10:15 运行
"0 15 10 * * ? *"	每天 10:15 运行
"0 15 10 * * ? 2008"	在 2008 年的每天 10:15 运行
"0 * 14 * * ?"	每天 14 点到 15 点每分钟运行一次，开始于 14:00，结束于 14:59
"0 0/5 14 * * ?"	每天 14 点到 15 点每 5 分钟运行一次，开始于 14:00，结束于 14:55
"0 0/5 14,18 * * ?"	每天 14 点到 15 点每 5 分钟运行一次，此外每天 18 点到 19 点每 5 分钟也运行一次
"0 0-5 14 * * ?"	每天 14:00 到 14:05，每分钟运行一次
"0 10,44 14 ? 3 WED"	3 月每周三的 14:10 到 14:44，每分钟运行一次
"0 15 10 ? * MON-FRI"	每周一、二、三、四、五的 10:15 运行
"0 15 10 15 * ?"	每月 15 日的 10:15 运行
"0 15 10 L * ?"	每月最后一天的 10:15 运行
"0 15 10 ? * 6L"	每月最后一个星期五的 10:15 运行
"0 15 10 ? * 6L 2014-2016"	2014 年、2015 年、2016 年每月最后一个星期五的 10:15 运行
"0 15 10 ? * 6#3"	每月第三个星期五的 10:15 运行

轻松一刻

Cron（译为克龙）代表 100 万年，是英文单词中最大的时间单位。另一个我们熟悉的 google 则来源于 Googol（译为古戈尔），代表 10 的 100 次方，足可穷尽宇宙万物。不过现在最大

的数是 googolplex，代表 10 的 googol 次方。在汉语里，在亿、兆之后还有很多大数，如京（10 的 16 次方，后略）、垓（20）、秭（24）、穰（28）、沟（32）、涧（36）、正（40）、载（44）。此外还有很多来自佛学的大数，如恒河沙（10 的 52 次方，后略）、频波罗（56）、矜羯罗（112，这个数已经大于 googol）、不可说不可说转（10 的 7×2122 次方）。

2. CronTrigger 实例

下面使用 CronTrigger 对 SimpleJob 进行调度，通过 Cron 表达式制定调度规则，让它每 5 秒运行一次，如代码清单 16-3 所示。

代码清单 16-3　CronTriggerRunner：使用CronTrigger进行调度

```
package com.smart.basic.quartz;
import org.quartz.CronExpression;
import org.quartz.CronTrigger;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.impl.StdSchedulerFactory;
public class CronTriggerRunner {
    public static void main(String args[]) {
        try {
            JobDetail jobDetail = new JobDetail("job1_2", "jGroup1",SimpleJob.class);
                                                    创建 CronTrigger，指定组及名称
         CronTrigger cronTrigger = new CronTrigger("trigger1_2", "tgroup1");①-1
            CronExpression cexp = new CronExpression("0/5 * * * * ?"); ①-2
            cronTrigger.setCronExpression(cexp); ①-3
                                                    设置 Cron 表达式
                                                                    定义 Cron 表达式
          SchedulerFactory schedulerFactory = new StdSchedulerFactory();
            Scheduler scheduler = schedulerFactory.getScheduler();
            scheduler.scheduleJob(jobDetail, cronTrigger);
            scheduler.start();
          //②
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行 CronTriggerRunner，每 5 秒将触发 SimpleJob 运行一次。在默认情况下，Cron 表达式对应当前的时区，可以通过 CronTriggerRunner 的 setTimeZone(java.util.TimeZone timeZone)方法显式指定时区。用户也可以通过 setStartTime(java.util.Date startTime)和 setEndTime(java.util.Date endTime)方法分别指定开始和结束时间。

实战经验

在代码清单 16-3 的②处需要通过 Thread.currentThread.sleep()方法让主线程睡眠一段时间，使调度器可以继续执行任务调度的工作；否则在调度器启动后，因为主线程立即退出，寄生于主线程的调度器也将关闭，调度器中的任务都将相应销毁，这将导致看

不到实际的运行效果。在单元测试的时候，使主线程睡眠一段时间以便让任务线程不被提前终止是经常使用的测试方法。对于测试某些长周期执行的调度任务，开发者可以简单地调整操作系统时间进行模拟。

16.2.4 使用 Calendar

在实际任务调度中，不可能一成不变地按照某个特定周期调度任务，必须考虑到实现生活中日历上的特殊日期。

下面安排一个任务，每小时运行一次，并将五一劳动节和国庆节排除在外，如代码清单 16-4 所示。

代码清单 16-4 CalendarExample：使用Calendar

```
package com.smart.basic.quartz;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import org.quartz.impl.calendar.AnnualCalendar;
import org.quartz.TriggerUtils;
  ...
public class CalendarExample {
    public static void main(String[] args) throws Exception {
        SchedulerFactory sf = new StdSchedulerFactory();
        Scheduler scheduler = sf.getScheduler();
                                                    法定节日是以每年为周期的，所以
                                                    使用 AnnualCalendar
        AnnualCalendar holidays = new AnnualCalendar();①

        Calendar laborDay = new GregorianCalendar();②    五一劳动节
        laborDay.add(Calendar.MONTH,5);
        laborDay.add(Calendar.DATE,1);
                                                    国庆节
        Calendar nationalDay = new GregorianCalendar();③
        nationalDay.add(Calendar.MONTH,10);
        nationalDay.add(Calendar.DATE,1);

        ArrayList<Calendar> calendars = new ArrayList<Calendar>();
        calendars.add(laborDay);
        calendars.add(nationalDay);          排除这两个特殊日期
        holidays.setDaysExcluded(calendars);④

        scheduler.addCalendar("holidays", holidays, false, false); ⑤  向 Scheduler
                                                                      注册日历
        Date runDate = TriggerUtils.getDateOf(0,0, 10, 1, 4); ⑥  4月1日上午10点
        JobDetail job = new JobDetail("job1", "group1", SimpleJob.class);
        SimpleTrigger trigger = new SimpleTrigger("trigger1", "group1",
                runDate,
                null,
                SimpleTrigger.REPEAT_INDEFINITELY,
                60L * 60L * 1000L);
        trigger.setCalendarName("holidays");⑦   让 Trigger 应用指定的日历规则
```

```
        scheduler.scheduleJob(job, trigger);
        scheduler.start();
        //在实际应用中主线程不能停止，否则Scheduler得不到执行，此处省略
    }
}
```

由于节日是每年重复的，所以使用 org.quartz.Calendar 的 AnnualCalendar 实现类，通过②和③处的代码，指定五一劳动节和国庆节这两个节日，并通过 AnnualCalendar#setDaysExcluded (ArrayList days)方法添加这两个日期。

在定制好 org.quartz.Calendar 后，还需要通过 Scheduler#addCalendar(String calName, Calendar calendar, boolean replace, boolean updateTriggers)方法进行注册。如果 updateTriggers 为 true，则 Scheduler 中已引用 Calendar 的 Trigger 将得到更新，如④处所示。

在⑦处，让一个 Trigger 指定使用 Scheduler 中代表节日的 Calendar，这样 Trigger 就会避开五一劳动节和国庆节这两个特殊日子了。

16.2.5　任务调度信息存储

在默认情况下，Quartz 将任务调度的运行信息保存在内存中。这种方法提供了最佳的性能，因为在内存中数据访问速度最快；不足之处是缺乏数据的持久性，当程序中途停止或系统崩溃时，所有运行的信息都会丢失。比如希望安排一个执行 100 次的任务，如果执行到 50 次时系统崩溃了，那么系统重启时任务的执行计数器将从 0 开始计数。在大多数实际应用中，往往并不需要保存任务调度的现场数据，因为很少需要规划一个指定执行次数的任务。对于仅执行一次的任务来说，其执行条件信息本身应该是已经持久化的业务数据（如对锁定到期的用户执行解锁的任务，锁定到期的时间点应该就是业务数据），当任务执行完成后，条件信息也会相应改变。当然，调度现场信息不仅包括运行次数，而且包括调度规则、JobDataMap 中的数据等。

如果确实需要持久化任务调度信息，则 Quartz 允许用户通过调整其属性文件，将这些信息保存到数据库中。在使用数据库保存了任务调度信息后，即使系统崩溃后重新启动，任务调度信息仍将得到恢复。如前面所说的例子，执行 50 次系统崩溃后重新运行，计数器将从 51 开始计数。使用数据库保存信息的任务称为持久化任务。

1．通过配置文件调整任务调度信息的保存策略

其实 Quartz JAR 文件的 org.quartz 包下就包含了一个 quartz.properties 属性配置文件，并提供了默认设置。如果需要调整默认配置，则可以在类路径下建立一个新的 quartz.properties 属性，它将自动被 Quartz 加载并覆盖默认的设置。

先来了解一下 Quartz 的默认属性配置文件，如代码清单 16-5 所示。

代码清单 16-5　quartz.properties：默认配置

```
org.quartz.scheduler.instanceName = DefaultQuartzScheduler ①
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.scheduler.wrapJobExecutionInUserTransaction = false
```

集群的配置，这里不使用集群

```
                                                    配置调度器的线程池
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool ②
org.quartz.threadPool.threadCount = 10
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore ③   配置任务调度现
                                                            场数据保存机制
```

Quartz 的属性配置文件主要包括三方面的信息：

（1）集群信息。

（2）调度器线程池。

（3）任务调度现场数据的保存。

如果任务数目很大，则可以通过增大线程池获得更好的性能。在默认情况下，Quartz 采用 org.quartz.simpl.RAMJobStore 保存任务的现场数据，顾名思义，信息保存在 RAM 内存中，可以通过以下设置将任务调度现场数据保存到数据库中，如代码清单 16-6 所示。

代码清单 16-6　quartz.properties：使用数据库保存任务调度现场数据

```
...
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.tablePrefix = QRTZ_ ①   数据库表前缀
org.quartz.jobStore.dataSource = qzDS ②     数据源名称
                                                    定义数据源的具体属性
org.quartz.dataSource.qzDS.driver = com.mysql.jdbc.Driver                    ③
org.quartz.dataSource.qzDS.URL =jdbc:mysql://localhost:3306/sampledb
org.quartz.dataSource.qzDS.user = stamen
org.quartz.dataSource.qzDS.password = abc
org.quartz.dataSource.qzDS.maxConnections = 10
```

要将任务调度数据保存到数据库中，就必须使用 org.quartz.impl.jdbcjobstore.JobStoreTX 代替原来的 org.quartz.simpl.RAMJobStore，并提供相应的数据库配置信息。首先在①处指定了 Quartz 数据库表的前缀，然后在②处定义了一个数据源，最后在③处具体定义这个数据源的连接信息。

用户必须事先在相应的数据库中创建 Quartz 的数据表（共 8 张），在 Quartz 的完整发布包的 docs/dbTables 目录下拥有对应不同数据库的 SQL 脚本，我们已经将 Quartz 的不同数据库脚本文件放到<工程项目>/schema 目录下。对于 MySQL innodb 引擎来说，直接运行 tables_mysql_innodb.sql 就可以了。

2．查询数据库中的运行信息

任务的现场保存对于上层的 Quartz 程序来说是完全透明的，在 src 目录下编写一个如代码清单 16-6 所示的 quartz.properties 文件后，重新运行代码清单 16-2 或代码清单 16-3 的程序，在数据库表中将看到对应的持久化信息。当调度程序运行过程中途停止后，任务调度的现场数据将记录在数据库表中，在系统重启时就可以在此基础上继续进行任务的调度，如代码清单 16-7 所示。

代码清单 16-7　JDBCJobStoreRunner：从数据库中恢复任务的调度

```
package com.smart.basic.quartz;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
import org.quartz.SimpleTrigger;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;
public class JDBCJobStoreRunner {
    public static void main(String args[]) {
        try {
            SchedulerFactory schedulerFactory = new StdSchedulerFactory();
            Scheduler scheduler = schedulerFactory.getScheduler();
            String[] triggerGroups = scheduler.getTriggerGroupNames();①
                                                          重新恢复在 tgroup1 组中名为
                                                          trigger1_1 的触发器的运行
          for (int i = 0; i < triggerGroups.length; i++) {②
                String[] triggers = scheduler.getTriggerNames(triggerGroups[i]);
                for (int j = 0; j < triggers.length; j++) {
                    Trigger tg =
scheduler.getTrigger(triggers[j],triggerGroups[i]);
                    if (tg instanceof SimpleTrigger
                      && tg.getFullName().equals("tgroup1.trigger1_1")) {②-1
                        scheduler.rescheduleJob(triggers[j],triggerGroups[i],
tg); ②-1
                    }
                }
            }
            scheduler.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

获取调度器中所有的触发器组

根据名称判断

恢复运行

当代码清单 16-2 中的 SimpleTriggerRunner 执行到一段时间后非正常退出，就可以通过 JDBCJobStoreRunner 根据记录在数据库中的现场数据恢复任务的调度。Scheduler 中的所有 Trigger 及 JobDetail 的运行信息都会保存在数据库中，这里仅恢复 tgroup1 组中名为 trigger1_1 的触发器，这可以通过如②-1 所示的代码进行过滤，触发器采用 GROUP.TRIGGER_NAME 的全名格式。通过 Scheduler#rescheduleJob(String triggerName, String groupName,Trigger newTrigger)方法即可重新调度关联某个 Trigger 的任务。

下面来观察一下不同时期 qrtz_simple_triggers 表的数据。

(1)运行代码清单 16-2 中的 SimpleTriggerRunner 一段时间后退出，这时 qrtz_simple_triggers 表中的数据如下：

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERE
▶ 1	trigger1_1 …	tgroup1 …	100	2000	4

其中，REPEAT_COUNT 表示需要运行的总次数，而 TIMES_TRIGGER 表示已经运行的次数。

（2）运行代码清单 16-7 中的 JDBCJobStoreRunner 恢复 trigger1_1 的触发器，运行一段时间后退出，这时 qrtz_simple_triggers 表中的数据如下：

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERE
▶ 1	trigger1_1	tgroup1	96	2000	22

首先 Quartz 会将原 REPEAT_COUNT 的值减去 TIMES_TRIGGER 的值得到新的 REPEAT_COUNT 值，并记录已经运行的次数（重新从 0 开始计算）。

（3）重新启动 JDBCJobStoreRunner 运行后，数据又将发生相应的变化，如下：

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERE
▶ 1	trigger1_1	tgroup1	74	2000	7

TIMES_TRIGGER 重新从 0 开始计数，而 REPEAT_COUNT 在原有值的基础上进行调整。

（4）继续运行直至完成所有剩余的次数，再次查询 qrtz_simple_triggers 表，如下：

	TRIGGER_NAME	TRIGGER_GROUP	REPEAT_COUNT	REPEAT_INTERVAL	TIMES_TRIGGERE

这时，该表中的记录已经变空。

值得注意的是，如果用户使用 JDBC 保存任务调度数据，那么运行代码清单 16-2 中的 SimpleTriggerRunner 然后退出，当再次希望运行 SimpleTriggerRunner 时，系统将抛出 JobDetail 重名的异常，如下：

```
Unable to store Job with name: 'job1_1' and group: 'jgroup1', because one already exists
with this identification.
```

因为每次调用 Scheduler#scheduleJob()方法时，Quartz 都会将 JobDetail 和 Trigger 的信息保存到数据库中。如果数据库中已经有同名的 JobDetail 或 Trigger，就会产生异常。

16.3 在 Spring 中使用 Quartz

Spring 为创建 Quartz 的 Scheduler、Trigger 和 JobDetail 提供了便利的 FactoryBean 类，以便能够在 Spring 容器中享受注入的好处。此外，Spring 还提供了一些便利工具类，用于直接将 Spring 中的 Bean 包装成合法的任务。Spring 进一步降低了使用 Quartz 的难度，能以更具 Spring 风格的方式使用 Quartz。概括来说，Spring 提供了两方面的支持。

（1）为 Quartz 的重要组件类提供更具 Bean 风格的扩展类。

（2）提供创建 Scheduler 的 BeanFactory 类，方便在 Spring 环境下创建对应的组件对象，并结合 Spring 容器生命周期执行启动和停止的动作。

16.3.1 创建 JobDetail

用户可以直接使用 Quartz 的 JobDetail 在 Spring 中配置一个 JobDetail Bean，但是 JobDetail 使用带参的构造函数，对于习惯通过属性配置的 Spring 用户来说存在使用上的

不便。为此，Spring 通过扩展 JobDetail 提供了一个更具 Bean 风格的 JobDetailFactoryBean。此外，Spring 还提供了一个 MethodInvokingJobDetailFactoryBean，通过这个 FactoryBean 可以将 Spring 容器中 Bean 的方法包装成 Quartz 任务，这样开发者就不必为 Job 创建对应的类。

1. JobDetailFactoryBean

JobDetailFactoryBean 扩展于 Quartz 的 JobDetail。使用该 Bean 声明 JobDetail 时，Bean 的名字即任务的名字，如果没有指定所属组，就使用默认组。除了 JobDetail 中的属性外，还定义了以下属性。

- jobClass：类型为 Class，实现 Job 接口的任务类。
- beanName：默认为 Bean 的 id 名，通过该属性显式指定 Bean 名称，它对应任务的名称。
- jobDataAsMap：类型为 Map，为任务所对应的 JobDataMap 提供值。之所以需要提供这个属性，是因为用户无法在 Spring 配置文件中为 JobDataMap 类型的属性提供信息，所以 Spring 通过 jobDataAsMap 设置 JobDataMap 的值。
- applicationContextJobDataKey：用户可以将 Spring ApplicationContext 的引用保存到 JobDataMap 中，以便在 Job 的代码中访问 ApplicationContext。为了达到这个目的，用户需要指定一个键，用于在 jobDataAsMap 中保存 ApplicationContext。如果不设置此键，JobDetailBean 就不会将 ApplicationContext 放入 JobDataMap 中。
- jobListenerNames：类型为 String[]，指定注册在 Scheduler 中的 JobListeners 名称，以便让这些监听器对本任务的事件进行监听。

下面的配置片段使用 JobDetailBean 在 Spring 中配置一个 JobDetail。

```
<bean name="jobDetail" class="org.springframework.scheduling.quartz.JobDetail FactoryBean "
    p:jobClass="com.smart.quartz.MyJob"
    p:applicationContextJobDataKey="applicationContext">
    <property name="jobDataAsMap">
        <map>
            <entry key="size" value="10" />
        </map>
    </property>
</bean>
```

JobDetailFactoryBean 封装了 MyJob 任务类，并为 Job 对应的 JobDataMap 设置了一个键为 size 的数据。此外，通过指定 applicationContextJobDataKey，让 Job 的 JobDataMap 持有 Spring ApplicationContext 的引用。

这样，MyJob 在运行时就可以通过 JobDataMap 访问到 size 和 ApplicationContext。来看一下 MyJob 的代码，如代码清单 16-8 所示。

代码清单 16-8　MyJob

```
package com.smart.quartz;
...
import org.quartz.Job;
import org.quartz.JobExecutionContext;
```

```
import org.quartz.JobExecutionException;
import org.springframework.context.ApplicationContext;
public class MyJob implements Job {
    public void execute(JobExecutionContext jctx) throws  JobExecutionException {
        Map dataMap = jctx.getJobDetail().getJobDataMap();①
        String size =(String)dataMap.get("size");②
        ApplicationContext ctx = (ApplicationContext)dataMap.get("applicationContext");③
        System.out.println("size:"+size);
        dataMap.put("size",size+"0");④
        //do sth...
    }
}
```

获取 JobDetail 关联的 JobDataMap

对 JobDataMap 所做的更改是否会被持久化取决于任务的类型

在②处获取 size 值，在③处可以根据键 applicationContext 获取 ApplicationContext，有了 ApplicationContext 的引用，Job 就可以毫无障碍地访问 Spring 容器中的任何 Bean。MyJob 可以在 execute()方法中对 JobDataMap 进行更改，如④处所示。如果 MyJob 实现了 Job 接口，则这种更改对于下一次执行是不可见的；如果 MyJob 实现了 StatefulJob 接口，则这种更改对于下一次执行是可见的。

2. MethodInvokingJobDetailFactoryBean

通常情况下，任务都定义在一个业务类方法中。这时，为了满足 Quartz Job 接口的规定，还需要定义一个引用业务类方法的实现类。为了避免创建这个只包含一行调用代码的 Job 实现类，Spring 提供了 MethodInvokingJobDetailFactoryBean，借由该 FactoryBean，可以将一个 Bean 的某个方法封装成满足 Quartz 要求的 Job。来看一个具体的例子：

```
<bean id="jobDetail_1"
    class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"
    p:targetObject-ref="myService" ①
    p:targetMethod="doJob" ②
    p:concurrent="false"/>③

<bean id="myService"  class="com.smart.service.MyService"/>
```

引用一个 Bean

指定目标 Bean 的方法

指定最终封装出的任务是否有状态

jobDetail_1 将 MyService#doJob()封装成一个任务，同时通过 concurrent 属性指定任务的类型。默认情况下封装为无状态的任务。如果希望封装为有状态的任务，仅需将 concurrent 属性设置为 false 就可以了。Spring 通过名为 concurrent 的属性指定任务的类型，能够更直接地描述任务执行的方式（有状态的任务不能并发执行，无状态的任务可并发执行），对于不熟悉 Quartz 内部机制的用户来说，比起 stateful，concurrent 显然更简明达意些。

MyService 服务类拥有一个 doJob()方法，它的代码如下所示：

```
package com.smart.service;
public class MyService {
  public void doJob(){①
      System.out.println("in MyService.dojob().");
  }
}
```

被封装成任务的目标方法

doJob()方法既可以是 static 的，也可以是非 static 的，但不能拥有方法入参。通过 MethodInvokingJobDetailFactoryBean 产生的 JobDetail 不能被序列化，所以不能被持久化到数据库中。如果希望使用持久化任务，则用户只能创建正规的 Quartz 的 Job 实现类。

16.3.2　创建 Trigger

Quartz 中另一个重要的组件就是 Trigger，Spring 按照相似的思路分别为 SimpleTrigger 和 CronTrigger 提供了更具 Bean 风格的 SimpleTriggerFactoryBean 和 CronTriggerFactoryBean 扩展类，通过这两个扩展类可以更容易地在 Spring 中以 Bean 的方式配置 Trigger。

1. SimpleTriggerFactoryBean

在默认情况下，通过 SimpleTriggerFactoryBean 配置的 Trigger 名称即为 Bean 的名称，属于默认组。SimpleTriggerFactoryBean 在 SimpleTrigger 的基础上新增了以下属性。

- jobDetail：对应的 JobDetail。
- beanName：默认为 Bean 的 id 名，通过该属性显式指定 Bean 名称，它对应 Trigger 的名称。
- jobDataAsMap：以 Map 类型为 Trigger 关联的 JobDataMap 提供值。
- startDelay：延迟多少时间开始触发，单位为毫秒，默认值为 0。
- triggerListenerNames：类型为 String[]，指定注册在 Scheduler 中的 TriggerListener 名称，以便让这些监听器对本触发器的事件进行监听。

下面的实例使用 SimpleTriggerFactoryBean 定义了一个 Trigger，该 Trigger 和 jobDetail 相关联，延迟 1 秒后启动，时间间隔为 2 秒，重复执行 100 次。此外，还为 Trigger 设置了 JobDataMap 数据。

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz. Simple
TriggerFactoryBean "
        p:jobDetail-ref="jobDetail"
        p:startDelay="1000"
        p:repeatInterval="2000"
        p:repeatCount="100">
        <property name="jobDataAsMap">①
            <map>
                <entry key="count" value="10" />
            </map>
        </property>
</bean>
```

需要特别注意的是，在①处配置的 Map 数据将填充到 Trigger 的 JobDataMap 中，执行任务时必须通过以下方式获取配置的值：

```
package com.smart.quartz;
…
public class MyJob implements StatefulJob {
    public void execute(JobExecutionContext jctx) throws JobExecutionException {
```

```
        Map dataMap = jctx.getTrigger().getJobDataMap();①  ← 获取 Trigger 的 JobDataMap
        String count = dataMap.get("count");
        dataMap.put("count","30") ②  ← 对 JobDataMap 的更改不会被持久化，不影响下次的执行
        ...
    }
}
```

2．CronTriggerFactoryBean

CronTriggerFactoryBean 扩展于 CronTrigger，触发器的名称即为 Bean 的名称，保存在默认组中。在 CronTrigger 的基础上，新增的属性和 SimpleTriggerFactoryBean 大致相同，配置的方法也和 SimpleTriggerFactoryBean 相似。下面给出一个简单的例子：

```
<bean id="checkImagesTrigger"
    class="org.springframework.scheduling.quartz. CronTriggerFactoryBean "
    p:jobDetail-ref="jobDetail"
    p:cronExpression="0/5 * * * * ?"/>
```

16.3.3 创建 Scheduler

Quartz 的 SchedulerFactory 是标准的工厂类，不太适合在 Spring 环境下使用。此外，为了保证 Scheduler 能够感知 Spring 容器的生命周期，在 Spring 容器启动后，Scheduler 自动开始工作，而在 Spring 容器关闭前，自动关闭 Scheduler。为此，Spring 提供了 SchedulerFactoryBean，这个 FactoryBean 大致拥有以下功能。

- 以更具 Bean 风格的方式为 Scheduler 提供配置信息。
- 让 Scheduler 和 Spring 容器的生命周期建立关联，相生相息。
- 通过属性配置的方式代替 Quartz 自身的配置文件。

来看一个 SchedulerFactoryBean 配置的例子，如代码清单 16-9 所示。

代码清单 16-9　SchedulerFactoryBean配置

```
<bean id="scheduler"
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers"> ①  ← 注册多个 Trigger
    <list>
        <ref bean="simpleTrigger" />
    </list>
    </property>
    <property name="schedulerContextAsMap">②  ← 以 Map 类型设置 SchedulerContext 数据
    <map>
        <entry key="timeout" value="30" />
    </map>
    </property>

    <property name="configLocation"
  value="classpath:com/smart/quartz/quartz.properties" />③  ← 显式指定 Quartz 的配置文件地址
</bean>
```

SchedulerFactoryBean 的 triggers 属性为 Trigger[]类型，可以通过该属性注册多个 Trigger，在①处注册了一个 Trigger。Scheduler 拥有一个类似于 ServletContext 的

SchedulerContext。SchedulerFactoryBean 允许用户以 Map 的形式设置 SchedulerContext 的参数值，如②处所示。在默认情况下，Quartz 在类路径下查询 quartz.properties 配置文件，用户也可以通过 configLocation 属性显式指定配置文件的位置，如③处所示。

除了实例中所用的属性外，SchedulerFactoryBean 还拥有一些常见的属性。

- calendars：类型为 Map，通过该属性向 Scheduler 注册 Calendar。
- jobDetails：类型为 JobDetail[]，通过该属性向 Scheduler 注册 JobDetail。
- autoStartup：SchedulerFactoryBean 在初始化后是否马上启动 Scheduler，默认为 true。如果设置为 false，则需要手工启动 Scheduler。
- startupDelay：在 SchedulerFactoryBean 初始化完成后，延迟多少秒启动 Scheduler，默认值为 0，表示马上启动。除非拥有需要立即执行的任务，一般情况下，可以通过 startupDelay 属性让 Scheduler 延迟一小段时间后启动，以便让 Spring 能够更快初始化容器中剩余的 Bean。

SchedulerFactoryBean 的一个重要功能是允许用户将 Quartz 配置文件中的信息转移到 Spring 配置文件中，由此带来的好处是配置信息的集中化管理，同时我们不必熟悉多种框架有差异的配置文件结构。回忆一个 Spring 集成的 Hibernate 框架，就知道这是 Spring 在集成第三方框架时采用的惯用招数。SchedulerFactoryBean 通过以下属性代替框架的自身配置文件。

- dataSource：当需要使用数据库来持久化任务调度数据时，用户可以在 Quartz 中配置数据源，也可以直接在 Spring 中通过 dataSource 指定一个 Spring 管理的数据源。如果指定了该属性，那么，即使在 quartz.properties 中已经定义了数据源，也会被 dataSource 覆盖。
- transactionManager：可以通过该属性设置一个 Spring 事务管理器。在设置 dataSource 时，Spring 强烈推荐用户使用一个事务管理器，否则数据表锁定可能无法正常工作。
- nonTransactionalDataSource：在全局事务的情况下，如果用户不希望 Scheduler 执行的相关数据操作参与到全局事务中，则可以通过该属性指定数据源。在 Spring 本地事务的情况下，使用 dataSource 属性就足够了。
- quartzProperties：类型为 Properties，允许用户在 Spring 中定义 Quartz 的属性，其值将覆盖 quartz.properties 配置文件中的设置。这些属性必须是 Quartz 能够识别的合法属性，在配置时，需要查看 Quartz 的相关文档。下面是一个配置 quartzProperties 属性的例子：

```
<bean id="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    ...
    <property name="quartzProperties">
        <props>
            <prop key="org.quartz.threadPool.class">①
                org.quartz.simpl.SimpleThreadPool
            </prop>
            <prop key="org.quartz.threadPool.threadCount">10</prop>②
```

Quartz 属性项 1

Quartz 属性项 2

```
        </props>
    </property>
</bean>
```

在实际应用中，我们并不总是在程序部署的时候就确定需要哪些任务，往往需要在运行期根据业务数据动态产生触发器和任务。用户完全可以在运行期通过代码调用SchedulerFactoryBean 获取 Scheduler 实例，然后动态注册触发器和任务。

16.4 在 Spring 中使用 JDK Timer

在 Java 1.3 以后的版本中，通过 java.util.Timer 和 java.util.TimerTask 这两个类提供了简单的任务调度功能，称之为 Java Timer。Java Timer 允许按照固定频率重复执行某项任务，这比直接通过编写底层线程程序进行任务调度要轻松许多，但是对于诸如“在每周周一 8:00 执行”这种和日历相关的任务调度需求来说，Java Timer 就无能为力了。

此外，JDK Timer 只适合对执行时间非常短的任务进行调度，因为在 Timer 中所有的 TimerTask 都在同一背景线程中执行，长时间的任务会严重影响到 Timer 的调度工作。所以 Java Timer 的使用范围很窄，在业务需求超过 Java Timer 的能力范围时，用户应该考虑前面介绍的 Quartz 框架。

16.4.1 Timer 和 TimerTask

TimerTask 代表一个需要多次执行的任务，它实现了 Runnable 接口，可以在 run()方法中定义任务逻辑。而 Timer 负责制定调度规则并调度 TimerTask。

1. TimerTask

TimerTask 相当于 Quartz 中的 Job，代表一个被调度的任务。二者最主要的区别在于，每当执行任务时，Quartz 都会创建一个 Job 实例，而 JDK Timer 则使用相同的 TimerTask 实例。所以，如果 TimerTask 类中拥有状态，那么这些状态对于后面的执行是可见的。从这点上来说，TimerTask 更像 StatefulJob 而非 Job。TimerTask 实现了 Runnable 接口，是一个抽象类，它只有以下 3 个方法。

- abstract void run()：子类覆盖这个方法并定义任务执行逻辑，每次执行任务时，run()方法就被调用一次。
- boolean cancel()：取消任务。假设任务被安排执行 *N* 次，那么在调用该方法后，后续的执行安排将取消。
- long scheduledExecutionTime()：返回此任务的计划执行时间点。如果在任务执行过程中调用此方法，则返回此次执行所对应的计划执行时间（一个任务的实际执行时间和计划执行时间可能不一致）。该方法一般在 run()方法内调用，用户可以通过该方法判断本次执行的时间点是否过晚，并据此决定是否要取消本次执行。该方法一般在固定频率执行时使用才会有意义。

2. Timer

Timer 只能以这样的方式对任务进行调度：在延迟一段时间或在指定时间点后运行一次任务或周期性地运行任务。实际上，在 Timer 内部使用 Object#wait(long time)进行任务的时间调度。这种机制不能保证任务的实时执行，只是一个粗略的近似值。

每个 Timer 对象都有一个对应的“背景线程”，它负责调度并执行 Timer 中所有的 TimerTask。由于所有的 TimerTask 都在这个线程中执行，所以 TimerTask 的执行时间应该比较短。如果一个 TimerTask 的执行占用了过多的时间，那么后面的任务就会受到影响。由于后续任务在调度时间轴上受到了“挤压”，所以可能会造成“扎堆”执行的情况。

当 Timer 中所有的 TimerTask 已经执行完成并且 Timer 对象没有外部引用时，Timer 的任务执行线程才会结束，但这可能需要很长的时间。因此，Timer 在默认情况下使用非守护线程（daemon Thread），这样用户就可以在应用程序中通过 Timer#cancel()方法手工结束 Timer。如果希望尽快结束 Timer 中的任务，则可以调用 TimerTask#cancel()方法来达到目的。

提示

Java 有两种线程：用户线程（User Thread）和守护线程（Daemon Thread）。守护线程是指在程序后台运行，提供一种通用服务的线程，垃圾回收线程就是一种典型的守护线程。守护线程是为主程序服务的，因此，当所有的用户线程都结束时，守护线程就会自动终止。因此，也可以将守护线程形象地称为奴仆线程，“主在我存，主亡我死”。将线程转换为守护线程可以通过调用 Thread 对象的 setDaemon(true)方法来实现。

Timer 的构造函数在创建 Timer 对象的同时将启动一个 Timer 背景线程。先来了解一下 Timer 的几个构造函数。

- Timer()：创建一个 Timer，背景线程是非守护线程。
- Timer(boolean isDaemon)：创建一个 Timer，当 isDaemon 为 true 时，背景线程为守护线程，守护线程将在应用程序主线程停止后自动退出。该方法是 Java 5.0 新增的。
- Timer(String name)：与 Timer()类似，只是通过 name 为关联背景线程指定名称。
- Timer(String name, boolean isDaemon)：与 Timer(boolean isDaemon)类似并为关联背景线程指定名称。该方法是 Java 5.0 新增的。

通过以下方法执行一次任务。

- schedule(TimerTask task, Date time)：在特定时间点执行一次任务。
- schedule(TimerTask task, long delay)：延迟指定时间后执行一次任务，delay 的单位为毫秒。

通过以下方法按固定间隔执行任务，间隔时长为上次任务执行完成的时间点到下次任务开始执行的时间点，任务的执行可能产生时间的漂移。

- schedule(TimerTask task, Date firstTime, long period)：从指定时间点开始周期性地执行任务，period 的单位为毫秒，后一次执行将在前一次执行完成后才开始计时。如任务被安排每 2 秒执行一次，假设第一次任务在 0 秒时间点开始执行并花费了 1.5 秒，则第二次将在第 3.5 秒时执行。
- schedule(TimerTask task, long delay, long period)：在延迟指定时间后，周期性地执行任务。

通过以下方法按固定频率执行任务。

- scheduleAtFixedRate(TimerTask task, Date firstTime, long period)：在指定时间点后，以指定频率执行任务。假设一个任务被安排每 2 秒执行一次，如果第一次执行花费了 1.5 秒，则在 0.5 秒后，第二次任务又会开始执行，以保证固定的执行频率。
- scheduleAtFixedRate(TimerTask task, long delay, long period)：在延迟一段时间后，以指定频率执行任务。

此外，Timer 还拥有几个控制方法。

- cancel()：取消 Timer 的执行，并丢弃所有被调度的 TimerTask，不过正在执行的任务不受影响。Timer 被取消后，不能调度新的 TimerTask。
- purge()：将所有已经取消的 TimerTask 从 Timer 列队中清除。如果 TimerTask 没有外部引用，就可以被垃圾回收。一般情况下无须调用该方法，只有在某些特殊情况下，当一次性取消多个 TimerTasks 后，调用该方法才有意义。

3．Java Timer 实例

首先创建一个任务，并在执行 10 次以后退出，代码如下：

```
package com.smart.basic.timer;
import java.util.Date;
import java.util.TimerTask;
public class SimpleTimerTask  extends TimerTask{
    private int count = 0;
    public void run() {
        System.out.println("execute task.");
        Date exeTime = (new Date(scheduledExecutionTime()));①
        System.out.println("本次任务安排执行时间点为："+exeTime);
       if(++count > 10){cancel();}②
    }
}
```

获取本次安排执行的时间点

在任务执行 10 次后主动退出

通过扩展 TimerTask 并实现 run()抽象方法定义一个任务。在 JDK Timer 中没有指定执行特定次数任务的方法。用户可以在任务的 run()方法中通过自定义代码实现。

下面通过 Timer 以固定延迟时间的方式每 5 秒执行一次任务。

```
package com.smart.basic.timer;
import java.util.Timer;
import java.util.TimerTask;
```

```
public class TimerRunner {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask task = new SimpleTimerTask();
        timer.schedule(task,1000L,5000L); ①  在延迟 1 秒后，每 5 秒执行一次任务
    }
}
```

运行以上代码，将输出以下信息：

```
execute task.
本次任务安排执行时间点为：Thu Mar 15 18:39:37 CST 2007
execute task.
本次任务安排执行时间点为：Thu Mar 15 18:39:42 CST 2007
execute task.
本次任务安排执行时间点为：Thu Mar 15 18:39:47 CST 2007
```

16.4.2　Spring 对 Java Timer 的支持

Spring 在 org.springframework.scheduling.timer 中提供了几个 JDK Timer 的支持类，主要在以下 3 个方面对 JDK Timer 提供支持。

（1）ScheduledTimerTask，它对 TimerTask 提供封装并提供相关的配置。

（2）通过 MethodInvokingTimerTaskFactoryBean 类可以将一个 Bean 的方法封装为 TimerTask。

（3）通过 TimerFactoryBean 可以更方便地配置 Timer。此外，让 Timer 的生命周期和 Spring 容器的生命周期相关，在初始化 TimerFactoryBean 后启动 Timer，在 Spring 容器关闭前取消 Timer。

1．ScheduledTimerTask

JDK Timer 标准的 API 要求在使用 Timer 方法进行任务调度时才指定调度的规则，这种方式不太适合进行 Bean 的配置，因此，Spring 提供了 ScheduledTimerTask，通过属性指定任务和调度规则。请看下面的代码：

```
<bean id="timerTask" class="com.smart.basic.timer.SimpleTimerTask" />

<bean id="scheduledTask"
      class="org.springframework.scheduling.timer.ScheduledTimerTask"
    p:timerTask-ref="timerTask1" ①  指定调度任务
      p:delay="1000"   ②  延迟时间，单位为毫秒
      p:period="1000""/>③  周期时间，单位为毫秒
```

如果希望只运行一次任务，则将 period 设置为 0 或负值。在默认情况下，采用固定时间间隔的调度方式，可以通过 fixedRate 属性设置以固定频率的方式执行任务。SimpleTimerTask 还可以将实现了 Runnable 接口的类封装成一个任务，用户可以通过 runnable 属性进行设置。

2．MethodInvokingTimerTaskFactoryBean

类似于 Quartz 的 MethodInvokingJobDetailFactoryBean，Spring 也为 JDK Timer 提供

了一个方便类，用于将 Bean 方法封装成一个 TimerTask。

```
<bean id="myService" class="com.smart.service.MyService" />

<bean id="timerTask1"①                    将返回一个 TimerTask 实例
    class="org.springframework.scheduling.timer.MethodInvokingTimerTask FactoryBean"
    p:targetObject-ref="myService" ②
    p:targetMethod="doJob"/> ③    业务方法  业务 Bean
```

3. TimerFactoryBean

类似于 Quartz 的 SchedulerFactoryBean，Spring 为 Timer 提供了 TimerFactoryBean 类。用户可以将多个 ScheduledTimerTask 注册到 TimerFactoryBean 中，TimerFactoryBean 将返回一个 Timer 实例。在 TimerFactoryBean 初始化完成后，对应的 Timer 启动，在 Spring 容器关闭之前，TimerFactoryBean 将取消 Timer。请看下面的配置代码：

```
<bean id="timer" class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <ref bean="scheduledTask" />
        </list>
    </property>
</bean>
```

scheduledTimerTasks 属性的类型为 ScheduledTimerTask[]，可以注入多个 ScheduledTimerTask。此外，TimerFactoryBean 还拥有一个 daemon 属性，指定生成 Timer 的背景线程是否为守护线程。

16.5 Spring 对 Java 5.0 Executor 的支持

Java 5.0 新增了一个并发工具包 java.util.concurrent，该工具包由 Doug Lea 设计并作为 JSR-166 添加到 Java 5.0 中。这是一个非常流行的并发工具包。它提供了功能强大的、高层次的线程构造器，包含执行器、线程任务框架、线程安全队列、计时器、锁（包含原子级别的锁）和其他一些同步的基本类型。

执行器 Executor 是并发工具包中一个重要的类，它对 Runnable 实例的执行进行了抽象，实现者可以提供具体的实现，如简单地以一个线程来运行 Runnable，或者通过一个线程池为 Runnable 提供共享线程。

因为 Executor 是 Java 5.0 新增的类，所以 Java 5.0 提供的实现类大多拥有线程池的内在支持。Spring 为 Executor 处理引入了一个新的抽象层，以便将线程池引入 Java 1.3 和 Java 1.4 环境中，同时屏蔽掉 Java 1.3、1.4、5.0 及 Java EE 环境中线程池实现的差异。

16.5.1　了解 Java 5.0 的 Executor

java.util.concurrent.Executor 接口的主要目的是将“任务提交”和“任务执行”分离解耦。该接口定义了任务提交的方法，实现者可以提供不同的任务执行机制，指定不同的线程使用规则和调度方案。

Executor 只有一个方法：void execute(Runnable command)，它接收任何实现了 Runnable 的实例，这个实例代表一个待执行的任务。可以使用如下代码提交任务：

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());①  ← 提交一个任务
executor.execute(new RunnableTask2());②  ← 提交另一个任务
```

Executor 本身并没有要求实现者以何种方式执行这些任务，一个简单的实现类甚至可以在提交任务时，立即在主线程中执行它们。下面是一个 Executor 最简单的实现：

```
public class SimpleExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();①  ← 在提交时直接执行任务
    }
}
```

但在更多的情况下，需要在一个异步线程中执行任务，而非在主线程中执行任务。下面是一个稍微有意义的实现，它为每个任务开启一个新的执行线程。

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();①  ← 在新的执行线程中执行任务
    }
}
```

但以上这些粗陋的实现并没有多大的实际应用价值，这也不符合 Executor 的设计初衷。真正有意义的实现需要引入线程、列队、调度等机制，这样的执行器才更贴近现实的需求。

Executor 接口还有两个子接口：ExecutorService 和 ScheduledExecutorService。ExecutorService 添加了结束任务的管理方法，此外，在提交任务时还可以获取一个 Future 实例，以便通过这个实例跟踪异步任务的执行情况。而 ScheduledExecutorService 可以对任务进行调度，如指定执行的延迟时间及执行的周期。如 ScheduledFuture<?> schedule (Runnable command, long delay, TimeUnit unit)方法将安排一个任务在延迟一段时间后执行。

Java 5.0 本身提供的 ThreadPoolExecutor 类实现了 Executor 和 ExecutorService 这两个接口，它使用一个线程池对提交的任务进行调度。如果需要处理数量巨大的短小并发任务（如 Web 服务器、数据库服务器、邮件服务器之类的应用程序需要处理来自远程的大量短小的任务），则采用线程池可以带来明显的好处。为每个请求创建一个新线程的开销很大，对于短小的任务，线程创建和销毁的时间可能比处理实际任务的时间还要长。除此以外，活动的线程需要消耗资源，过多的线程将导致大量的内存占用。线程池能让多个任务重用同一个线程，线程创建的开销被分摊到多个任务上。此外，由于在任

务到达时线程已经存在，所以也消除了任务执行时因创建线程所带来的延迟。可以通过适当地调整线程池中的参数，当任务的数目超过某个阈值时，强制新任务排队等待，直到获得一个线程来处理，从而防止资源无限制消耗而引发的系统问题。

ThreadPoolExecutor 的子类 ScheduledThreadPoolExecutor 实现了 ScheduledExecutorService 接口，添加了对任务的调度功能，如指定延迟一小段时间后执行任务，让任务周期性执行。该类明显优于 Java 1.3 中的 Timer，因为它通过内建的线程池让每个任务在独立的执行线程中执行，而非让所有任务在单一的背景线程中执行。Timer 经常出现的时间漂移、任务挤压等问题基本上得到了规避。

在 java.util.concurrent 中为创建这些接口实例提供了一个综合性的工厂类 Executors，它拥有以下众多方便的静态工厂方法。

- public static ExecutorService newFixedThreadPool(int nThreads)：创建一个线程池，重复使用一组固定的线程执行任务。
- public static ExecutorService newCachedThreadPool()：线程池是动态的，不够用时将创建新的线程，长时间不用的线程将被收回。
- public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, Thread Factory threadFactory)：创建一个线程池，可在指定延迟后执行或者定期执行。

来看一个具体的例子，如代码清单 16-10 所示。

代码清单 16-10　ExecutorExample

```
package com.smart.basic.executor;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
public class ExecutorExample {
    private Executor executor; ①          声明一个执行器
    public void setExecutor(Executor executor) {
        this.executor = executor;
    }
    public void executeTasks() {②          用执行器执行多个任务
        for (int i = 0; i < 6; i++) {
            executor.execute(new SimpleTask("task" + i));
        }
    }
    public static void main(String[] args) {          通过工厂类创建一个带 3 个
        ExecutorExample ee = new ExecutorExample();   线程的固定线程池的执行器
        ee.setExecutor(Executors.newFixedThreadPool(3)); ③
        ee.executeTasks();
    }
}
class SimpleTask implements Runnable {④          任务类
    private String taskName;
    public SimpleTask(String taskName) {
        this.taskName = taskName;
    }
    public void run() {
```

```
        System.out.println("do "+taskName+"... in Thread:" + Thread.currentThread().
getId());
    }
}
```

运行以上代码，输出以下信息：

```
do task0... in Thread:7
do task1... in Thread:8
do task2... in Thread:9
do task3... in Thread:7
do task5... in Thread:9
do task4... in Thread:8
```

可见，这 6 个任务共享了线程池中的 3 个线程。由于 ExecutorService 用线程池中的 3 个线程服务于提交的任务，从而避免了为每个任务创建独立线程的代价，具有更高的运行性能。

16.5.2　Spring 对 Executor 所提供的抽象

Spring 的 org.springframework.core.task.TaskExecutor 接口等同于 java.util.concurrent.Executor 接口。该接口和 Java 5.0 的 Executor 接口拥有相同的 execute(Runnable task)方法。TaskExecutor 拥有一个 SchedulingTaskExecutor 子接口，新增了任务调度规则定制的功能。

在 Spring 发行包中预定义了一些 TaskExecutor 实现，它们可以满足大部分应用要求，一般情况下，用户不必自行编写实现类。下面是 TaskExecutor 的实现类。

SyncTaskExecutor：位于 org.springframework.core.task 包中，实现了 TaskExecutor 接口。这个实现不会异步执行任务；相反，每次调用都在发起调用的主线程中执行。

下面是 SchedulingTaskExecutor 的实现类。

- SimpleAsyncTaskExecutor：位于 org.springframework.core.task 包中。该类没有使用线程池，在每次执行任务时都创建一个新线程。但是，它依然支持对并发总数设限，当超过并发总数限制时，阻塞新的任务直到有可用的资源。
- ConcurrentTaskExecutor：位于 org.springframework.scheduling.concurrent 包中。该类是 Java 5.0 的 Executor 的适配器，以便将 Java 5.0 的 Executor 当作 Spring 的 TaskExecutor 使用。
- SimpleThreadPoolTaskExecutor：位于 org.springframework.scheduling.quartz 包中。该类实际上是继承于 Quartz 的 SimpleThreadPool 类的子类，它将监听 Spring 的生命周期回调。当用户有线程池，需要在 Quartz 和非 Quartz 组件中共用时，该类可以发挥它的用处。
- ThreadPoolTaskExecutor：位于 org.springframework.scheduling.concurrent 包中。该类只能在 Java 5.0 中使用，它暴露了一些属性，方便在 Spring 中配置一个 java.util.concurrent.ThreadPoolExecutor，并把它包装成 TaskExecutor。

- TimerTaskExecutor：位于 org.springframework.scheduling.timer 包中。该类使用一个 Timer 作为其后台实现。

代码清单 16-10 中的 ExecutorExample 只能在 Java 5.0 下运行，如果用 Spring 的 TaskExecutor 替换①处的 Executor，当程序需部署到低版本的 Java 环境中时，仅需要选择一个合适的实现就可以了。

```
package com.smart.executor;
import org.springframework.core.task.SimpleAsyncTaskExecutor;
import org.springframework.core.task.TaskExecutor;
public class ExecutorExample {                          使用 Spring 的 TaskExecutor
    private TaskExecutor executor; ①                    替换 Java 5.0 的 Executor
    public void setExecutor(TaskExecutor executor) {
        this.executor = executor;
    }
    public void executeTasks() {
        for (int i = 0; i < 6; i++) {
            executor.execute(new SimpleTask("task" + i));
        }
    }
    public static void main(String[] args) {            使用 Spring 的 TaskExecutor
        ExecutorExample ee = new ExecutorExample();      实现类
        ee.setExecutor(new SimpleAsyncTaskExecutor());②
        ee.executeTasks();
    }
}
class SimpleTask implements Runnable {
    ...
}
```

如果将 ExecutorExample 配置成一个 Bean，通过注入的方式提供 executor 属性，就可以方便地选用不同的实现版本。如果是在 Java 5.0 版本中，则用户可以选用 ThreadPoolTaskExecutor；而在 JDK 低版本中则可以使用 SimpleAsyncTaskExecutor。这样，程序就可以在不同的 JDK 版本中进行移植。

16.6 实际应用中的任务调度

对于那些运行规则固定的静态任务(如每隔 30 分钟更新缓存)，当然可以通过 Spring 配置文件定义调度规则并在 Spring 容器中启动运行调度。但事情并非总是这么简单，在实际应用中，往往需要根据业务数据动态产生任务。举个例子：在论坛系统中，当发现某些用户发表一些非法的内容时，作为惩罚手段，往往需要将该用户锁定一段时间。这个功能就需要通过操作业务功能动态地创建任务来完成：在锁定用户后，经过一段时间，通过任务自动将其解锁。随着业务系统的运行，任务源源不断地产生，又源源不断地得到执行。

在实际的应用系统中，任务的执行时间点往往是非常关键的，不允许其发生时间漂

移(如电力控制系统)。此外,任务的执行往往是与日历相关的(如在本月 15 日上午 10:00 执行),所以 Quartz 往往是更适合的选择。

16.6.1　如何产生任务

在实际应用中,有两种不同的创建动态任务的方式。

(1) 业务流程产生型。

(2) 扫描线程产生型。

前者表示在运行一个业务时,在业务操作过程中产生任务;而后者由一个任务线程定时扫描业务数据库的任务表,并根据业务数据产生任务。

1. 在业务流程中产生

某个业务需要用到任务,如果任务的执行时间点离业务的操作时间点不是太长,则可以直接在业务流程中安排好任务。假设在电力传输管理系统中有一个功能,可以将一条传输线路在某段时间内停止供电。安排停电的时间点离业务的操作时间点一般不会太长,因此,可以在用户执行线路停电安排的业务时,立即向 Scheduler 中注册两个任务:一个是在某一时间点执行断电的任务;另一个是在某一时间点执行恢复供电的任务。

当然,考虑到系统重启的情况,可能需要使用持久化任务,以便在系统重启后能够恢复已经安排的任务。

2. 由扫描任务周期性产生

但在有些情况下不适合在业务流程中产生任务,而应当通过定期扫描功能,根据业务数据动态产生任务。

在论坛系统中,为了清除一些无效的注册账号,可以定义这样的规则:如果账号注册后在半年内都没有激活,则将这些账号删除。对于系统来说,引发这个潜在账号清除任务的业务是用户注册功能,但是不应该在注册账号后就安排一个清除账号的任务:首先,在注册用户时,并不知道这个账号在将来是否会满足清除的条件;其次,任务的执行时间点离当前点太远,现在就安排显然“为时过早”。

一般来说,像清除账号这样的任务对执行时间点的要求并不会太高,所以只要安排一个以天为周期定时执行的任务,将当天满足条件的账号删除即可。

现在来考虑另一种比较极端的需求。假设对清除账号的执行时间点有严格的要求:清除账号的时间点只能精确地发生在一年之后的那个时间点上(365×24×60×60×1000 毫秒之后)。这时,定期周期执行的清除任务就不再适用了。因为如果清除任务执行频率过小,就不能满足精确执行点的要求;如果频率过大,则会对业务数据库产生频繁的访问,对数据库的性能产生不良影响。此外,在需要访问数据库的情况下,单纯从技术角度来看,任务执行频率本身是受限的,用户很难安排秒级的任务。

为了保证严格的执行时间点并尽量减小对数据库的影响,需要一个用于产生最近执

行任务的扫描任务定期查询数据库，并为那些在一小段时间后就要执行的潜在任务进行动态安排，如图 16-2 所示。

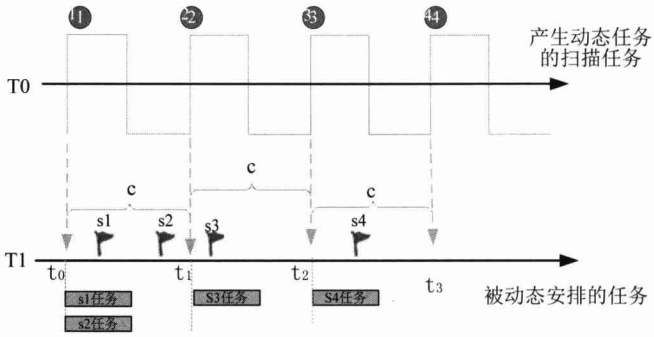


图 16-2　通过周期性扫描动态安排近期某个时间点执行的任务

T0 对应一个定时的任务，它负责周期性地扫描业务表，查找出在后续的扫描周期时间范围内将要执行的任务，创建并安排这些任务。通过这种方式可以带来 3 个好处。

（1）降低对数据库的影响：扫描任务自身的执行频率不必太高，如可以让扫描任务 30 分钟执行一次，将 30 分钟后所有需要执行的任务都安排好。这样，扫描任务在 30 分钟之内只会进行一次数据库访问，对数据库的影响微乎其微。

（2）缩短调度器中任务列队的长度：由于不是将所有的潜在任务提前很长时间就进行安排，而仅对一个扫描周期内的任务进行安排，所以调度器中任务列表的长度可以得到有效的控制。

（3）保证任务在精确的时间点执行：由于任务提前得到安排，任务执行时间点的精确性可以得到很好的保证。当然，由于扫描任务本身需要一定的执行时间，如果任务的执行时间点离扫描任务的扫描点很近，则执行时间点的精确性将会受到一些影响。这时，用户可以将安排任务的范围调大一些，以取消这种任务执行点和扫描周期点过近的问题。不过，一般情况下，这种影响是可以接受的。

如果系统存在许多依赖一个主任务动态创建出来的子任务，那么最好通过一张任务表来维护这些任务，而不要将它们分散在不同的业务表中。假设用户账号清除任务的条件信息在用户表中，锁定用户任务的信息在锁定用户表中，商品过期任务的信息在商品表中。这时就需要分别为这些任务定义一个扫描任务，或者在扫描任务内遍历所有相关的业务表。如果和任务有关的业务表非常多（如 100 张），为了动态创建任务，扫描任务就会对数据库产生很大的影响。如果通过一张任务表记录所有潜在的任务，并在业务操作过程中动态维护这张任务表，则仅需一个扫描任务查询这张任务表就可以了，有效地降低了创建动态任务对数据库的影响。使用统一任务表的方式也是有代价的，任务有关的业务模块在业务功能之外需要额外地考虑维护任务表中的数据。比如，在用户注册模块，在用户注册完成后，就向任务表添加一条清除账号的任务；而在用户激活模块，则需要将清除该用户账号的任务记录删除。

16.6.2　任务调度对应用程序集群的影响

对于有集群要求的 Web 应用程序来说，如果应用系统本身有任务调度的功能，就必须在系统设计初期仔细分析任务调度功能是否适合集群。按任务执行结果影响的范围，可以将任务为两种类型。

- 全局任务：指那些执行结果会影响到应用系统全局的任务。例如，每天凌晨生成业务报表、定期调用短信接口发送短信、定期清除系统过期数据等任务，它们的执行都会给整个系统带来“全局可见”的结果。所以，在传统的集群系统中，全局任务最好在一个独立部署的服务节点上执行，否则可能会因重复多次执行而引发系统逻辑的错误。
- 本地任务：和全局任务相对，指执行结果的影响范围仅限于本地，不会造成全局影响的任务。如定期刷新本地缓存、定期清除本地节点临时文件等任务，它们的执行结果只对本地服务节点有影响，需要在每个本地服务节点部署任务。

Quartz 可支持集群部署，其原理很简单，即让多个调度节点互为热备，在同一时刻只有一个调度节点是激活的，任务只在这个激活的调度节点中执行，其他的调度节点处于“休眠”状态；当激活的调度节点崩溃时，则唤醒某一个“休眠”的调度节点，以接管任务调度的工作。

Quartz 可通过两种方式实现集群：其一是通过一个中间数据库，使集群节点相互感知，以实现故障切换；其二是通过 Terracotta 实现集群。Terracotta 是一个 JVM 级的开源集群框架，简单地说，就是将多个 JVM 透明化为一个 JVM 使用。如何配置 Quartz 集群已经超出了本书的讨论范围，读者可查阅相关资料了解。

16.6.3　任务调度云

虽然 Quartz 支持集群方案，但其集群方案存在一个比较大的问题，即无法区分全局任务和本地任务，只能以调度节点为单位开启或关闭任务。对于既拥有全局任务又拥有本地任务的应用系统来说，Quartz 集群很难满足要求。此外，Quartz 缺少集群调度节点和任务管理控制台，不便进行节点调度和任务监控。

鉴于此，笔者曾基于 Quartz 研发了一个任务调度集群方案，称其为“任务调度云”，它实现了以下几个目标：

- 任务调度服务无须双机热备硬件支持，具有“软故障迁移”功能，避免了系统的单点故障。
- 任务调度服务无须独立部署，且不会造成“单任务多次运行”的问题。
- 任务调度的服务器可分配、可管理、可切换、可监控。
- 任务本身可管理、可控制、可监控。
- 任务的程序开发简单易行，对调度运行环境完全透明。

任务调度云的基础设施由任务调度资源识别器、故障迁移监控器、控制命令执行器及运行日志记录器组成，如图 16-3 所示。其基础实现原理是通过协商机制将云中的某个服务节点标志为“主节点”，全局任务只在主节点中执行，本地任务在各个节点都执行。通过心跳机制监测各节点的活动状态，如果主节点不活动了，则由协商机制选取云中的下一个活动节点作为主节点。

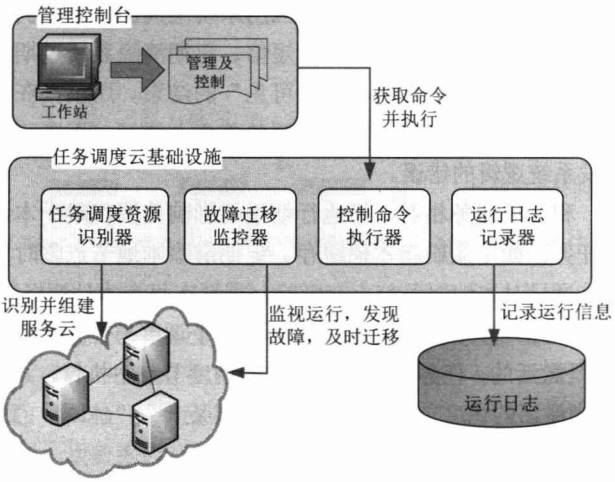


图 16-3　任务调度云

在任务调度云启动时，任务调度云的资源识别器将首先开始工作，根据管理控制台的资源定义将指定的服务器加入任务调度云中，以完成任务调度云的组建。

而故障迁移器将负责监控云中任务调度服务器的运行状态。当发生故障时，及时将任务迁移到备用任务调度服务器中，实现调度服务的热切换，达到故障迁移的目的。

任务调度管理员可在管理控制台向任务调度云发布控制命令，控制命令执行器在发现新命令后，下载命令并在云中执行命令。

运行日志记录器负责记录任务调度云中任务运行的详细日志，包括运行开始时间、结束时间、运行结果及运行在云中的具体位置等，以便管理员对任务运行进行审查和监控。

从任务的构成上看，一个任务由两个方面组成：其一是被调度的执行逻辑；其二是调度规则。执行逻辑可以描述为一个接口方法，而调度规则可通过周期表达式或 Cron 表达式指定。在任务调度云中，调度规则在管理控制台中定义和维护，而执行逻辑通过 Job 接口描述。Job 接口定义了如下方法。

- execute()：该方法承载任务执行的逻辑，开发者只需实现该方法并在其中定义需要进行调度的执行逻辑即可。
- isGlobal()：确定任务是全局任务还是本地任务，返回 true 表示是全局任务，否则为本地任务。本地任务在所有的任务调度节点都被执行，而全局任务只会在集群中某个激活的调度节点中执行。

- isNeedLog()：是否要让任务调度云记录任务的运行日志，可以在管理控制台查看到任务运行的日志。
- getJobGroupName()和 getJobName()：任务全限定名，由分组名+任务名组成。任务调度云通过任务全限定名将调度规则应用于特定的任务上，同时还通过全限定名进行名称的定位和控制。

在任务调度云中开发一个任务非常简单，只需扩展任务抽象类实现 execute()方法就可以了。以下就是一个具体任务的例子：

```
@Component("simpleJob")
public class SimpleJob extends AbstractBeanJob {
    public void execute(JobExecutionContext jobExecutionContext) {
        System.out.println("执行任务逻辑!");
    }
}
```

将任务类以一个 Bean 的方式进行定义，而任务全限定名直接取自 Bean 的名称，无须额外设定。Job 的接口方法在 AbstractBeanJob 抽象类中都有默认值，默认值指定这个 SimpleJob 是一个全局任务，需要任务运行日志且任务的全限定名为 DEFAULT.simpleJob。在管理控制台注册一个名为 DEFAULT.simpleJob 的任务，它提供了任务对应的调度规则。

笔者已经将任务调度云的完整方案整理成一篇技术文章《面向云计算的 EPGIS 平台中的任务调度服务研究》，发表在《电力信息化》2011 年第 5 期中，感兴趣的读者可以参考阅读。

16.6.4　Web 应用程序中调度器的启动和关闭问题

我们知道，静态变量是 ClassLoader 级别的，如果 Web 应用程序停止，那么这些静态变量也会从 JVM 中清除。**但线程是 JVM 级别的，如果用户在 Web 应用中启动了一个线程，那么这个线程的生命周期并不会和 Web 应用程序保持同步**。也就是说，即使停止了 Web 应用，这个线程依旧是活动的。正是因为这个很隐晦的问题，所以很多有经验的开发者不太赞成在 Web 应用中私自启动线程。

如果手工使用 JDK Timer（Quartz 的 Scheduler），在 Web 容器启动时启动非守护线程的 Timer，当 Web 容器关闭时，除非用户手工关闭这个 Timer，否则 Timer 中的任务还会继续执行。

下面通过一个小例子来演示这个“灵异现象”。通过 ServletContextListener 在 Web 容器启动时创建一个 Timer 并周期性地执行一个任务，如代码清单 16-11 所示。

代码清单 16-11　StartCycleRunTask：容器监听器

```
package com.smart.web;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
```

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
public class StartCycleRunTask implements ServletContextListener {
    private Timer timer;
    public void contextDestroyed(ServletContextEvent arg0) {①
        System.out.println("Web应用程序启动关闭...");
    }
    public void contextInitialized(ServletContextEvent arg0) {②
        System.out.println("Web应用程序启动...");
        timer = new Timer();②-1
        TimerTask task = new Simple TimerTask();
        timer.schedule(task, 1000L, 5000L); ②-2
    }
}
class SimpleTimerTask extends TimerTask {③
    private int count;
    public void run() {
        System.out.println((++count)+"execute task..."+(new Date()));
    }
}
```

该方法在 Web 容器关闭时执行

在 Web 容器启动时自动执行该方法

创建一个 Timer，在 Timer 内部自动创建一个背景线程

注册一个每隔 5 秒执行一次的任务

任务

在 web.xml 中声明这个 Web 容器监听器。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
...
    <listener>
        <listener-class>com.smart.web.StartCycleRunTask</listener-class>
    </listener>
</web-app>
```

在 Tomcat 中部署这个 Web 应用并启动后，用户将看到任务每隔 5 秒执行一次，在控制台上输出如图 16-4 所示的信息。

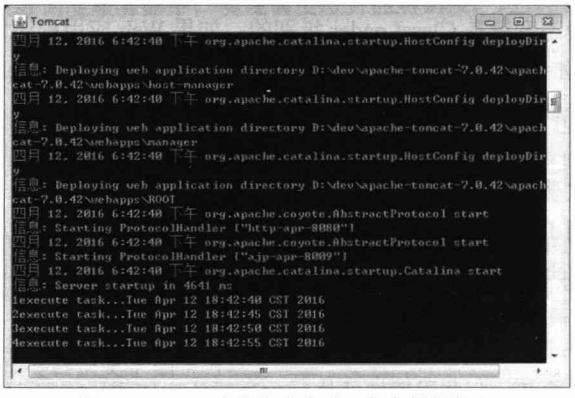


图 16-4　Web 应用程序启动，任务周期执行

运行一段时间后，登录 Tomcat 管理后台，将对应的 Web 应用（chapter16）关闭，Tomcat 的后台管理界面如图 16-5 所示。

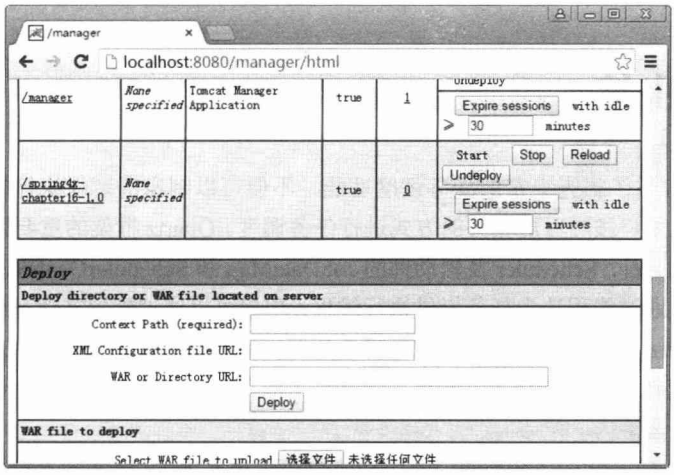


图 16-5　在 Tomcat 管理后台关闭相应的 Web 应用

转到 Tomcat 控制台，用户将看到，虽然 Web 应用已经关闭，但 Timer 任务还在“我行我素”地执行（见图 16-6）——舞台已拆，表演照旧。

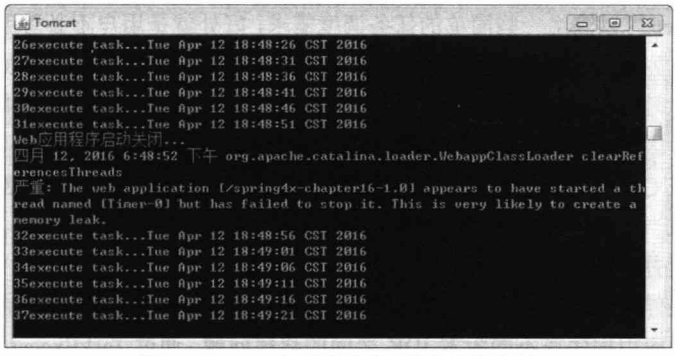


图 16-6　Web 应用关闭后，任务继续执行

可以通过改变代码清单 16-11 中的代码，在 contextDestroyed(ServletContextEvent arg0)中添加 timer.cancel()，在 Web 容器关闭后手工停止 Timer 来结束任务。

Spring 为 JDK Timer 和 Quartz Scheduler 所提供的 TimerFactoryBean 和 SchedulerFactoryBean 能够与 Spring 容器的生命周期关联，在 Spring 容器启动时启动调度器，而在 Spring 容器关闭时停止调度器。所以在 Spring 中通过这两个 FactoryBean 配置调度器，再从 Spring IoC 中获取调度器的引用进行任务调度，这样就不会出现这种 Web 容器关闭而任务依然执行的问题。而如果用户在程序中直接使用 Timer 或 Scheduler，如不进行额外的处理，则会出现这一问题。

16.7 小结

Quartz 提供了极为丰富的任务调度功能，不但可以制定周期性执行的任务调度方案，还可以让用户按照日历相关的方式进行任务调度。Quartz 框架的重要组件包括 Job、JobDetail、Trigger、Scheduler 及辅助性的 JobDataMap 和 SchedulerContext。Quartz 拥有一个线程池，通过线程池为任务提供执行线程，用户可以通过配置文件对线程池进行参数定制。Quartz 的另一个重要功能是可将任务调度信息持久化到数据库中，以便系统重启时能够恢复已经安排的任务。此外，Quartz 还拥有完善的事件体系，允许用户注册各种事件的监听器。

Spring 为 Quartz 的 JobDetail 和 Trigger 提供了更具 Bean 风格的支持类，这使得用户能够更方便地在 Spring 中通过配置定制这些组件实例。Spring 的 SchedulerFactoryBean 让用户可以脱离 Quartz 自身的配置体系，而以更具 Spring 风格的方式定义 Scheduler。此外，还可以享受 Scheduler 生命周期和 Spring 容器生命周期绑定的好处。

JDK Timer 可以满足一些简单的任务调度需求，好处就是用户不必引用 JDK 之外的第三方类库，学习也很简单；但必须保证任务是短小型的任务，任务应该很快就能完成，否则将产生“时间漂移”的问题。此外，使用 JDK Timer 的任务对执行时间点应该没有严格的要求，因为 JDK Timer 只能做到近似的时间安排。

不管是在 Quartz 中还是在 JDK Timer 中，Spring 都可以将 Spring 容器中 Bean 的方法直接封装成一个任务，方便任务实例的制定。

对于大量并发的短小型任务，使用线程池进行任务调度可以带来明显的好处，如节省资源、增强伸缩性、获得更快的响应速度。Java 5.0 新增加了支持线程池的 Executor，Spring 考虑到应用程序在不同版本 JDK 之间的移植问题，通过 TaskExecutor 对 Executor 进行了抽象，这样，当应用部署到不同版本的 JDK 中时，仅需选择适合的实现类就可以了。

开发任务调度的应用程序并非想象中那样简单，在实际应用中，需要考虑到动态任务的产生机制，以照顾数据库性能、任务队列长度、任务执行时间点精度等方方面面的问题。此外，对于需要集群部署的 Web 应用程序，应该将业务应用和任务调度应用分别部署，以避免一个任务多次执行的问题。Spring 提供了用于创建 Quartz、JDK Timer 基础设施的 FactoryBean，可以保证 Quartz 和 JDK Timer 调度线程在 Spring 容器关闭时自动停止。

第 17 章

Spring MVC

大部分 Java 应用都是 Web 应用，展现层是 Web 应用不可忽略的重要环节。Spring 为展现层提供了一个优秀的 Web 框架——Spring MVC。和众多其他的 Web 框架一样，它基于 MVC 的设计理念。此外，它采用了松散耦合、可插拔的组件结构，比其他的 MVC 框架更具扩展性和灵活性。Spring MVC 通过一套 MVC 注解，让 POJO 成为处理请求的控制器，无须实现任何接口。同时，Spring MVC 还支持 REST 风格的 URL 请求：注解驱动及 REST 风格的 Spring MVC 是 Spring 的出色功能之一。此外，Spring MVC 在数据绑定、视图解析、本地化处理及静态资源处理上都有许多不俗的表现。它在框架设计、扩展性、灵活性等方面全面超越了 Struts、WebWork 等 MVC 框架，从原来的追赶者一跃成为 MVC 的领跑者。

本章主要内容：

- Spring MVC 体系概述
- 注解驱动的控制器
- 数据绑定、输入/输出格式化及数据校验
- 视图解析
- 本地化解析
- 文件上传
- WebSocket 支持
- 静态资源处理、请求拦截器、异常处理

本章亮点：

- 深入分析并图解 Spring MVC 体系结构
- 对处理方法入参绑定及视图解析进行详细分析

17.1 Spring MVC 体系概述

Spring MVC 框架围绕 DispatcherServlet 这个核心展开，DispatcherServlet 是 Spring MVC 的总导演、总策划，它负责截获请求并将其分派给相应的处理器处理。Spring MVC 框架包括注解驱动控制器、请求及响应的信息处理、视图解析、本地化解析、上传文件解析、异常处理及表单标签绑定等内容。

17.1.1 体系结构

Spring MVC 是基于 Model 2 实现的技术框架，Model 2 是经典的 MVC（Model、View、Control）模型在 Web 应用中的变体，这个改变主要源于 HTTP 协议的无状态性。Model 2 的目的和 MVC 一样，也是利用处理器分离模型、视图和控制，达到不同技术层级间松散层耦合的效果，提高系统灵活性、复用性和可维护性。在大多数情况下，可以将 Model 2 与 MVC 等同起来。

在利用 Model 2 之前，把所有的展现逻辑和业务逻辑集中在一起，有时也称这种应用模式为 Model 1。Model 1 的主要缺点就是紧耦合，复用性差，维护成本高。

由于 Spring MVC 是基于 Model 2 实现的框架，所以它底层的机制也是 MVC，通过图 17-1 描述 Spring MVC 的整体架构。

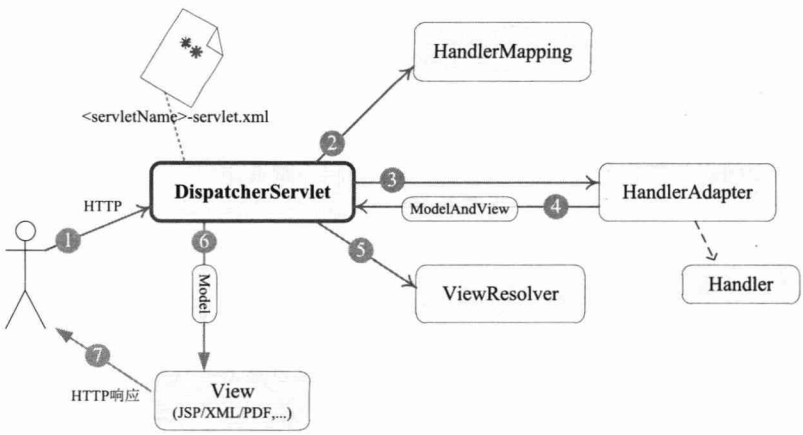


图 17-1 Spring MVC 框架模型

从接收请求到返回响应，Spring MVC 框架的众多组件通力配合、各司其职，有条不紊地完成分内的工作。在整个框架中，DispatcherServlet 处于核心的位置，它负责协调和组织不同组件以完成请求处理并返回响应的工作。和大多数 Web MVC 框架一样，Spring MVC 通过一个前端 Servlet 接收所有的请求，并将具体工作委托给其他组件进行

处理，DispatcherServlet 就是 Spring MVC 的前端 Servlet。下面对 Spring MVC 处理请求的整体过程进行讲解。

（1）整个过程始于客户端发出一个 HTTP 请求，Web 应用服务器接收到这个请求。如果匹配 DispatcherServlet 的请求映射路径（在 web.xml 中指定），则 Web 容器将该请求转交给 DispatcherServlet 处理。

（2）DispatcherServlet 接收到这个请求后，将根据请求的信息（包括 URL、HTTP 方法、请求报文头、请求参数、Cookie 等）及 HandlerMapping 的配置找到处理请求的处理器（Handler）。可将 HandlerMapping 看作路由控制器，将 Handler 看作目标主机。值得注意的是，在 Spring MVC 中并没有定义一个 Handler 接口，实际上，任何一个 Object 都可以成为请求处理器。

（3）当 DispatcherServlet 根据 HandlerMapping 得到对应当前请求的 Handler 后，通过 HandlerAdapter 对 Handler 进行封装，再以统一的适配器接口调用 Handler。HandlerAdapter 是 Spring MVC 的框架级接口，顾名思义，HandlerAdapter 是一个适配器，它用统一的接口对各种 Handler 方法进行调用。

（4）处理器完成业务逻辑的处理后将返回一个 ModelAndView 给 DispatcherServlet，ModelAndView 包含了视图逻辑名和模型数据信息。

（5）ModelAndView 中包含的是“逻辑视图名”而非真正的视图对象，DispatcherServlet 借由 ViewResolver 完成逻辑视图名到真实视图对象的解析工作。

（6）当得到真实的视图对象 View 后，DispatcherServlet 就使用这个 View 对象对 ModelAndView 中的模型数据进行视图渲染。

（7）最终客户端得到的响应消息可能是一个普通的 HTML 页面，也可能是一个 XML 或 JSON 串，甚至是一张图片或一个 PDF 文档等不同的媒体形式。

以上每个步骤都包含丰富的知识点，本章将逐步揭示每个组件的“庐山真面目”。不过现在请收好所有的好奇心，我们第一步要做的是在 web.xml 中配置好 DispatcherServlet，让 Spring MVC 的“心脏”跳动起来。

17.1.2　配置 DispatcherServlet

DispatcherServlet 是 Spring MVC 的“灵魂”和“心脏”，它负责接收 HTTP 请求并协调 Spring MVC 的各个组件完成请求处理的工作。和任何 Servlet 一样，用户必须在 web.xml 中配置好 DispatcherServlet。我们在第 2 章中已经配置了一个简单的 DispatcherServlet，这里进一步分析其具体的配置。

要了解 Spring MVC 框架的工作机理，必须回答以下 3 个问题。

（1）DispatcherServlet 框架如何截获特定的 HTTP 请求并交由 Spring MVC 框架处理？

（2）位于 Web 层的 Spring 容器（WebApplicationContext）如何与位于业务层的 Spring

容器（ApplicationContext）建立关联，以使 Web 层的 Bean 可以调用业务层的 Bean？

（3）如何初始化 Spring MVC 的各个组件，并将它们装配到 DispatcherServlet 中？

1. 配置 DispatcherServlet，截获特定的 URL 请求

大家知道，我们可以在 web.xml 中配置一个 Servlet，并通过<servlet-mapping>指定其处理的 URL。这是传统的 DispatcherServlet 配置方式。而 Spring 4.0 已全面支持 Servlet 3.0，因此也可以采用编程式的配置方式。这里先采用传统的 web.xml 的方式进行讲解，然后介绍基于 Servlet 3.0 的新方式。假设我们希望 Spring MVC 的 DispatcherServlet 能截获并处理所有以.html 结束的 URL 请求，那么可以在 web.xml 中按如下方式进行配置，如代码清单 17-1 所示。

代码清单 17-1　web.xml：配置DispatcherServlet

```
<context-param>
    <param-name>contextConfigLocation</param-name>                    ①
    <param-value>classpath:/applicationContext.xml</param-value>
</context-param>                                                      业务层和持久层的Spring配置文件，这些配置文件被父Spring容器所使用
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>

<servlet> ②                                                           声明 DispatcherServlet
    <servlet-name>smart</servlet-name>
    <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping> ③                                                   名为 DispatcherServlet 匹配的 URL 模式
    <servlet-name>smart</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

在①处，通过 contextConfigLocation 参数指定业务层 Spring 容器的配置文件（多个配置文件使用逗号分隔）。ContextLoaderListener 是一个 ServletContextListener，它通过 contextConfigLocation 参数所指定的 Spring 配置文件启动“业务层”的 Spring 容器。

在②处配置了名为 smart 的 DispatcherServlet，它默认自动加载/WEB-INF/smart-servlet.xml（<servlet-Name>-servlet.xml）的 Spring 配置文件，启动 Web 层的 Spring 容器。

在③处，通过<servlet-mapping>指定 DispatcherServlet 处理所有以.html 为后缀的 HTTP 请求，即所有带.html 后缀的 HTTP 请求都会被 DispatcherServlet 截获并处理。

我们知道，多个 Spring 容器之间可设置为父子级的关系，以实现良好的解耦。在这里，“Web 层”Spring 容器将作为“业务层”Spring 容器的子容器，即“Web 层”容器可以引用“业务层”容器的 Bean，而“业务层”容器却访问不到“Web 层”容器的 Bean。

需要提醒的是，一个 web.xml 可以配置多个 DispatcherServlet，通过其<servlet-mapping>配置，让每个 DispatcherServlet 处理不同的请求。

DispatcherServlet 遵循“契约优于配置”的原则，在大多数情况下，用户无须进行额外的配置，只需按契约行事即可。

如果确实要对 DispatcherServlet 的默认规则进行调整，则 DispatcherServlet 是“敞开胸怀”的。下面是常用的一些配置参数，可通过<servlet>的<init-param>指定。

- namespace：DispatcherServlet 对应的命名空间，默认为<servlet-name>-servlet，用于构造 Spring 配置文件的路径。在显式指定该属性后，配置文件对应的路径为 WEB-INF/<namespace>.xml，而非 WEB-INF/<servlet-name>-servlet.xml。如果将 namespace 设置为 sample，则对应的 Spring 配置文件为 WEB-INF/sample.xml。
- contextConfigLocation：如果 DispatcherServlet 上下文对应的 Spring 配置文件有多个，则可以使用该属性按照 Spring 资源路径的方式指定。如“classpath:sample1.xml, classpath:sample2.xml”，DispatcherServlet 将使用类路径下的 sample1.xml 和 sample2.xml 这两个配置文件初始化 WebApplicationContext。
- publishContext：布尔类型的属性，默认值为 ture。DispatcherServlet 根据该属性决定是否将 WebApplicationContext 发布到 ServletContext 的属性列表中，以便调用者可借由 ServletContext 找到 WebApplicationContext 实例，对应的属性名为 DispatcherServlet#getServletContextAttributeName()方法的返回值。
- publishEvents：布尔类型的属性。当 DispatcherServlet 处理完一个请求后，是否需要向容器发布一个 ServletRequestHandledEvent 事件，默认值为 true。如果容器中没有任何事件监听器，则可以将该属性设置为 false，以便提高运行性能。

下面的代码显式指定 Web 层的 Spring 配置文件。

```
<servlet>
      <servlet-name>smart</servlet-name>
      <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
              /WEB-INF/spring/webApplicationContext.xml
            </param-value>
     </init-param>
     <load-on-startup>1</load-on-startup>
</servlet>
```

之前提到 Spring 4.0 已全面支持 Servlet 3.0，因此，在 Servlet 3.0 环境中，也可以使用编程的方式来配置 Servlet 容器。下面的代码可达到和代码清单 17-1 同样的效果。

```
public class SmartApplicationInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new
            DispatcherServlet());
```

```
        registration.setLoadOnStartup(1);
        registration.addMapping("*.html*");
    }
}
```

接下来看看 Servlet 3.0 的实现原理。在 Servlet 3.0 环境中，容器会在类路径中查找实现 javax.servlet.ServletContainerInitializer 的类，如果发现已有实现类，就会调用它来配置 Servlet 容器。在 Spring 中，org.springframework.web.SpringServletContainerInitializer 类实现了该接口，同时这个类又会查找实现 org.springframework.web.WebApplicationInitializer 接口的类，并将配置任务交给这些实现类去完成。另外，Spring 提供了一个便利的抽象类 AbstractAnnotationConfigDispatcherServletInitializer 来实现这个接口，使得它在注册 DispatcherServlet 时只需简单地指定它的 Servlet 映射即可。在上述示例中，当应用部署到 Servlet 3.0 容器中时，容器启动时会自动发现它，并使用它来配置 Servlet 上下文。

2. 探究 DispatcherServlet 的内部逻辑

现在剩下的最后一个问题是：Spring 如何将上下文中的 Spring MVC 组件装配到 DispatcherServlet 中？通过查看 DispatcherServlet 的 initStrategies()方法的代码，一切真相就大白于天下了。

```
protected void initStrategies(ApplicationContext context) {
    initMultipartResolver();//①初始化上传文件解析器（直译为多部分请求解析器）
    initLocaleResolver();  //②初始化本地化解析器
    initThemeResolver();  //③初始化主题解析器
    initHandlerMappings(); //④初始化处理器映射器
    initHandlerAdapters();  //⑤初始化处理器适配器
    initHandlerExceptionResolvers();//⑥初始化处理器异常解析器
    initRequestToViewNameTranslator();//⑦初始化请求到视图名翻译器
    initViewResolvers();//⑧初始化视图解析器
}
```

initStrategies()方法将在 WebApplicationContext 初始化后自动执行，此时 Spring 上下文中的 Bean 已经初始化完毕。该方法的工作原理是：通过反射机制查找并装配 Spring 容器中用户显式自定义的组件 Bean，如果找不到，则装配默认的组件实例。

Spring MVC 定义了一套默认的组件实现类，也就是说，即使在 Spring 容器中没有显式定义组件 Bean，DispatcherServlet 也会装配好一套可用的默认组件。在 spring-webmvc-4.x.jar 包的 org/springframework/web/servlet 类路径下拥有一个 DispatcherServlet.properties 配置文件，该文件指定了 DispatcherServlet 所使用的默认组件。

```
##本地化解析器
org.springframework.web.servlet.LocaleResolver=
org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

##主题解析器
org.springframework.web.servlet.ThemeResolver=
org.springframework.web.servlet.theme.FixedThemeResolver

##处理器映射（共2个）
org.springframework.web.servlet.HandlerMapping=
```

```
org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
    org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

##处理器适配器（共3个）
org.springframework.web.servlet.HandlerAdapter=
org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    org.springframework.web.servlet.mvc.method.annotation
                                                .RequestMappingHandlerAdapter

##异常处理器（共3个）
org.springframework.web.servlet.HandlerExceptionResolver=
org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\
    org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

##视图名称翻译器
org.springframework.web.servlet.RequestToViewNameTranslator=
org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

##视图解析器
org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver
```

如果用户希望采用非默认类型的组件，则只需在 Spring 配置文件中配置自定义的组件 Bean 即可。Spring MVC 一旦发现上下文中有用户自定义的组件，就不会使用默认的组件。下面通过表 17-1 进一步了解 DispatcherServlet 装配每种组件的过程。

表 17-1　DispatcherServlet装配各型组件的逻辑

组件类型	发现机制
文件上传解析器（☆）	（1）查找名为 multipartResolver、类型为 MultipartResolver 的 Bean 作为该类型的组件。 （2）没有默认的实现类。 如果用户没有在上下文中显式定义这一类型的组件，则 DispatcherServlet 中将不会拥有该类型的组件
本地化解析器（☆）	（1）查找名为 localeResolver、类型为 LocaleResolver 的 Bean 作为该类型的组件。 （2）如果（1）找不到，则使用默认的实现类（AcceptHeaderLocaleResolver）创建该类型的组件
主题解析器（☆）	（1）查找名为 themeResolver、类型为 LocaleResolver 的 Bean 作为该类型的组件。 （2）如果（1）找不到，则使用默认的实现类（FixedThemeResolver）创建该类型的组件
处理器映射器（★）	（1）如果 detectAllHandlerMappings 属性为 true（默认为 true），则根据类型匹配机制查找上下文及父 Spring 容器中所有类型为 HandlerMapping 的 Bean，将它们作为该类型的组件。 （2）如果 detectAllHandlerMappings 属性为 false，则查找名为 handlerMapping、类型为 HandlerMapping 的 Bean 作为该类型的组件。 （3）如果通过以上两种方式都找不到，则使用 BeanNameUrlHandlerMapping 实现类创建该类型的组件
处理器适配器（★）	（1）如果 detectAllHandlerAdapters 属性为 true（默认为 true），则根据类型匹配机制查找上下文及父 Spring 容器中所有类型为 HandlerAdapter 的 Bean，将它们作为该类型的组件。 （2）如果 detectAllHandlerAdapters 属性为 false，则查找名为 handlerAdapter、类型为 HandlerAdapter 的 Bean 作为该类型的组件。 （3）如果通过以上两种方式都找不到，则使用 DispatcherServlet.properties 配置文件中指定的 3 个实现类分别创建一个适配器，并添加到适配器列表中

续表

组件类型	发现机制
处理器异常解析器（★）	（1）如果 detectAllHandlerExceptionResolvers 属性为 true（默认为 true），则根据类型匹配（HandlerExceptionResolver）机制查找上下文及父 Spring 容器中所有匹配的 Bean 作为该类型的组件。 （2）如果 detectAllHandlerExceptionResolvers 属性为 false，则查找名为 handlerExceptionResolver、类型为 HandlerExceptionResolver 的 Bean 作为该类型的组件。 （3）如果通过以上两种方式都找不到，则查找 DispatcherServlet.properties 中定义的默认实现类，不过该文件中没有对应处理器异常解析器的默认实现类（用户可以更改属性文件）
视图名翻译器（☆）	（1）查找名为 viewNameTranslator、类型为 RequestToViewNameTranslator 的 Bean 作为该类型的组件。 （2）如果（1）找不到，则使用默认的实现类（DefaultRequestToViewNameTranslator）创建该类型的组件
视图解析器（★）	（1）如果 detectAllViewResolvers 属性为 true（默认为 true），则根据类型匹配（ViewResolver）机制查找上下文及父 Spring 容器中所有匹配的 Bean 作为该类型的组件。 （2）如果 detectAllViewResolvers 属性为 false，则查找名为 viewResolver、类型为 ViewResolver 的 Bean 作为该类型的组件。 （3）如果通过以上两种方式都找不到，则通过 DispatcherServlet.properties 中定义的默认实现类（InternalResourceViewResolver）创建该类型的组件

有些组件最多允许存在一个实例，如 MultipartResolver、LocaleResolver 等，在表 17-1 中使用☆进行标注；而另一些组件允许存在多个实例，如 HandlerMapping、HandlerAdapter 等，在表 17-1 中使用★进行标注。同一类型的组件如果存在多个，那么它们之间的优先级顺序如何确定呢？这些组件都实现了 org.springframework.core.Ordered 接口，可通过 order 属性确定优先级顺序，值越小优先级越高。

简言之，当 DispatcherServlet 初始化后，就会自动扫描上下文的 Bean，根据名称或类型匹配的机制查找自定义的组件，找不到时则使用 DispatcherServlet.properties 定义的默认组件。

17.1.3 一个简单的实例

在学习了 Spring MVC 框架的整体结构后，下面通过一个简单的实例讲解 Spring MVC 开发的基本过程。Spring MVC 应用开发一般包括以下几个步骤。

（1）配置 web.xml，指定业务层对应的 Spring 配置文件，定义 DispatcherServlet。

（2）编写处理请求的控制器（处理器）。

（3）编写视图对象，这里使用 JSP 作为视图。

（4）配置 Spring MVC 的配置文件，使控制器、视图解析器等生效。

17.1.2 节已经详细讲解了如何在 web.xml 中配置 Spring 业务层容器及定义 DispatcherServlet 的知识，所以这里直接从第（2）步开始。

1. 编写处理请求的控制器

Spring MVC 通过@Controller 注解即可将一个 POJO 转化为处理请求的控制器，通

过@RequestMapping 为控制器指定处理哪些 URL 的请求。UserContoller 是一个负责用户处理的控制器，其代码如代码清单 17-2 所示。

代码清单 17-2　UserController.java

```
package com.smart.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller ①                          使UserController成为处理请求的控制器
@RequestMapping("/user") ②             处理来自/user URI的请求
public class UserController {

    @RequestMapping("/register") ③
    public String register(){          返回一个String类型的逻辑视图名
        return "user/register";
    }
}
```

首先使用@Controller 对 UserController 类进行标注，使其成为一个可处理 HTTP 请求的控制器。然后使用@RequestMapping 对 UserController 及其 register()方法进行标注，确定 register()对应的请求 URL。

在 UserController 类定义处标注的@RequestMapping 限定了 UserController 类处理所有 URI 为/user 的请求，它相对于 Web 容器部署根路径。UserController 类可以定义多个处理方法，处理来自/user URI 的请求。假设 Web 容器的部署根路径为/chapter17，则代码清单 17-2 中的 register()方法将处理所有来自/chapter17/user/ register.html 的请求。值得说明的是，②处类级的@RequestMapping 不是必需的，可以直接在方法中标注@RequestMapping，此时，方法处@RequestMapping 指定的 URI 则是相对于部署根路径的。

register()方法返回一个字符串 user/register，它代表一个逻辑视图名，将由视图解析器解析为一个具体的视图对象。在本例中，它将被映射为/WEB-INF/views/user/register.jsp。稍后，读者将了解如何装配完成这一任务的视图解析器。

2．编写视图对象

我们使用一个 register.jsp 作为用户的注册页面，UserController#register()方法处理完成后，将转向这个 register.jsp 页面，如代码清单 17-3 所示。

代码清单 17-3　register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>新增用户</title>
</head>                                          将表单提交到/user.html控制器中
<body>
  <form method="post" action="<c:url value="/user.html"/>">①
    <table>
        <tr>
          <td>用户名：</td>
```

```
        <td><input type="text" name="userName"/></td>
      </tr>
      <tr>
       <td>密码：</td>
        <td><input type="password" name="password"/></td>
      </tr>
      <tr>
       <td>姓名：</td>
        <td><input type="text" name="realName"/></td>
      </tr>
      <tr>
       <td colspan="2"><input type="submit" name="提交"/></td>
      </tr>
   </table>
  </form>
</body>
</html>
```

register.jsp 很简单，它包括了一个表单，单击“提交”按钮后，表单提交到/user.html 进行处理。UserController 添加了一个 createUser()方法用于处理表单提交的请求，如代码清单 17-4 所示。

代码清单 17-4　UserController.java

```
package com.smart.web;
…
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserService userService;①          注入业务层的Bean

    @RequestMapping(method=RequestMethod.POST)②   处理/user的请求，不过请求的方法必须为POST
    public ModelAndView createUser(User user) {   ③
        userService.createUser(user);
        ModelAndView mav = new ModelAndView();
        mav.setViewName("user/createSuccess");
        mav.addObject("user", user);
        return mav;
    }
   …
}
```

将表单值映射到User对象中，调用UserService保存user，返回ModelAndView

createUser()方法处的@RequestMapping 注解让 createUser()处理 URI 为/user.html 且请求方法为 POST 的请求。Spring MVC 自动将表单中的数据按参数名和 User 属性名匹配的方式进行绑定，将参数值填充到 User 的相应属性中。调用业务层的 UserService 进行业务处理，进而返回 ModelAndView 对象，逻辑视图名为 user/createSuccess，user 作为模型数据暴露给视图对象。

User 对象的代码如代码清单 17-5 所示。

代码清单 17-5　User.java

```
package com.smart.domain;
public class User {
    private String userName;
    private String password;
    private String realName;
    …
}
```

注意到 User 中的参数名和代码清单 17-3 中的表单组件是相同的，这样 Spring MVC 即可将表单组件值填充到 User 的相应属性中。

视图解析器将 user/createSuccess 解析为/WEB-INF/views/user/createSuccess.jsp 的视图对象，createSuccess.jsp 可以访问到模型中的数据。createSuccess.jsp 页面代码如代码清单 17-6 所示。

代码清单 17-6　createSuccess.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
   <head>
     <title>用户创建成功</title>
   </head>
   <body>
    恭喜，用户${user.userName}创建成功。①
  </body>
</html>
```

访问Model中的属性

由于 UserController#createUser()通过@ModelAttribute("user")将 User 对象放到模型中，所以 createSuccess.jsp 可以通过${user.userName}访问模型中的数据。

3．配置 Spring MVC 的配置文件

要使以上实例正常工作，需要在 Spring MVC 配置文件中进行简单的配置，如代码清单 17-7 所示。

代码清单 17-7　smart-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"
      xmlns:p="http://www.springframework.org/schema/p"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.web"/> ①
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" ②
          p:prefix="/WEB-INF/views/"
          p:suffix=".jsp"/>
</beans>
```

在①处通过<context:component-scan>使 Spring 扫描 com.smart.web 包下所有的类，

让标注 Spring 注解的类生效。而在②处定义了一个视图名称解析器，将视图逻辑名解析为/WEB-INF/views/<viewName>.jsp 的视图对象，如图 17-2 所示。

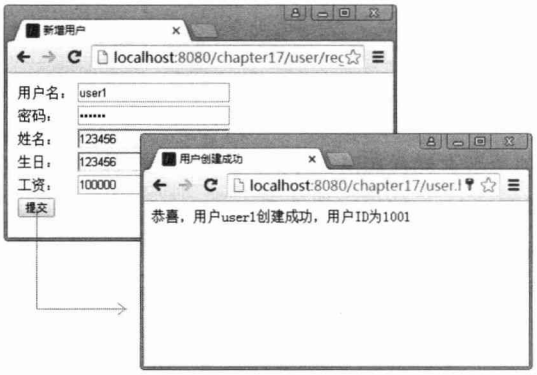


图 17-2　注册模块页面

4．运行用户注册模块

将用户模块部署到 Web 服务器中，Web 服务器服务端口为 8080，部署根路径为/chapter17，通过 http://localhost:8080/chapter17/user/register.html 即可访问用户注册页面。填写用户注册信息，单击“提交”按钮将表单提交到 http://localhost:8080/chapter17/user.html 地址中，UserController 控制器的 createUser()方法响应这个请求，并导向 createSuccess.jsp 页面中。

对于用户注册表单提交直到返回响应这一过程，用实物对象对组件接口进行替换，得到如图 17-3 所示的交互图。

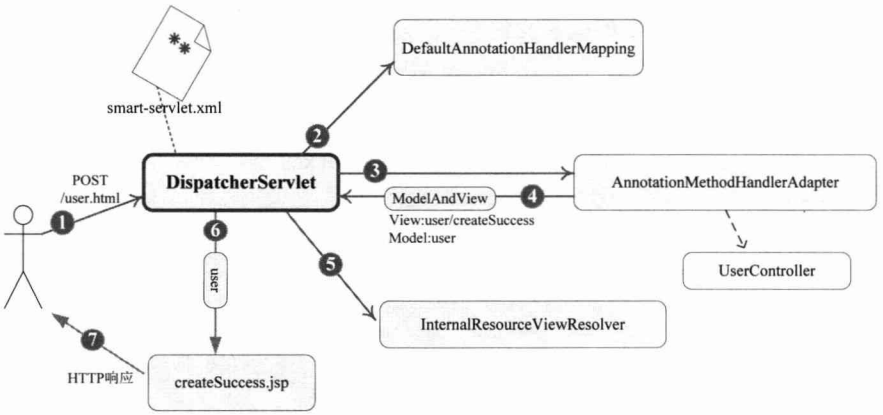


图 17-3　处理/user.html 表单提交的整体过程

再简要描述一下 Spring MVC 处理/user.html 的整个过程。

① DispatcherServlet 接收到客户端的/user.html 请求。

② DispatcherServlet 使用 DefaultAnnotationHandlerMapping 查找负责处理该请求的处理器。

③ DispatcherServlet 将请求分发给名为/user.html 的 UserController 处理器。

④ 处理器完成业务处理后，返回 ModelAndView 对象，其中 View 的逻辑名为/user/createSuccess，而模型包含一个键为 user 的 User 对象。

⑤ DispatcherServlet 调用 InternalResourceViewResolver 组件对 ModelAndView 中的逻辑视图名进行解析，得到真实的/WEB-INF/view/user/createSuccess.jsp 视图对象。

⑥ DispatcherServlet 使用/WEB-INF/view/user/createSuccess.jsp 对模型中的 user 模型对象进行渲染。

⑦ 返回响应页面给客户端。

通过这个例子，我们了解了开发一个 Spring MVC 应用所需经历的大体过程。这个例子太过简单，每个步骤都是最简的实现。在本章的后续章节中，我们将对以上各个步骤进行深入分析。

17.2　注解驱动的控制器

17.2.1　使用@RequestMapping 映射请求

在 POJO 类定义处标注@Controller，再通过<context:component-scan/>扫描相应的类包，即可使 POJO 成为一个能处理 HTTP 请求的控制器。

用户可以创建数量不限的控制器，分别处理不同的业务请求，如 LogonController、UserController、ForumController 等。每个控制器可拥有多个处理请求的方法，每个方法负责不同的请求操作。如何将请求映射到对应的控制器方法中是 Spring MVC 框架的重要任务之一，这项任务由@RequestMapping 承担。

在控制器的类定义及方法定义处都可以标注@RequestMapping，类定义处的@RequestMapping 提供初步的请求映射信息，方法定义处的@RequestMapping 提供进一步的细分映射信息。DispatcherServlet 截获请求后，就通过控制器上@RequestMapping 提供的映射信息确定请求所对应的处理方法。

将请求映射到控制器处理方法的工作包含一系列映射规则，这些规则是根据请求中的各种信息制定的，具体包括请求 URL、请求参数、请求方法、请求头这 4 个方面的信息项。

1．通过请求 URL 进行映射

@RequestMapping 使用 value 值指定请求的 URL，如@RequestMapping("/user")、@RequestMapping("/register")等。需要注意的是，@RequestMapping 在类定义处指定的 URL 相对于 Web 应用的部署路径，而在方法定义处指定的 URL 则相对于类定义处指定的 URL。如果在类定义处未标注@RequestMapping，则仅在处理方法处标注@RequestMapping，此时，方法处指定的 URL 则相对于 Web 应用的部署路径，如代码清单 17-8 所示。

代码清单 17-8　UserController.java：在类定义处不使用@RequestMapping

```
package com.smart.web;
import org.springframework.web.bind.annotation.RequestMapping;
...
@Controller
public class UserController {

    @RequestMapping(path="/user/createUser")
    public String createUser(@ModelAttribute("user") User user){
        ...
        return "user/createSuccess";
    }

    @RequestMapping("/user/register")
    public String register(@ModelAttribute("user") User user){
        return "user/register";
    }
}
```

这样，/user/register.html 请求将由 register()方法处理，而/user/createUser.html 请求将由 createUser()方法处理。注意，它们都相对于 Web 应用的部署路径。

同一控制器的多个处理方法负责处理相同业务模块的不同操作，但凡设计合理的 Web 应用都会将这些操作请求安排在某一相同的 URL 之下。所以除非特别的原因，建议不要舍弃类定义处的@RequestMapping。

@RequestMapping 不但支持标准的 URL，还支持 Ant 风格（?、*和**字符，参见 4.3.2 节）的和带{xxx}占位符的 URL。以下 URL 都是合法的。

- /user/*/createUser：匹配/user/aaa/createUser、/user/bbb/createUser 等 URL。
- /user/**/createUser：匹配/user/createUser、/user/aaa/bbb/createUser 等 URL。
- /user/createUser??：匹配/user/createUseraa、/user/createUserbb 等 URL。
- /user/{userId}：匹配 user/123、user/456 等 URL。
- /user/**/{userId}：匹配 user/aaa/bbb/123、user/aaa/456 等 URL。
- company/{companyId}/user/{userId}/detail：匹配 company/123/user/456/detail 等 URL。

通过@PathVariable 可以将 URL 中的占位符参数绑定到控制器处理方法的入参中，如代码清单 17-9 所示。

代码清单 17-9　UserController.java：使用@PathVariable

```
package com.smart.web;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.servlet.ModelAndView;
...
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/{userId}")
    public ModelAndView showDetail(@PathVariable("userId") String userId){
        ModelAndView mav = new ModelAndView();
```

```
        mav.setViewName("user/showDetail");
        mav.addObject("user", userService.getUserById(userId));
        return mav;
    }
}
```

URL 中的{xxx}占位符可以通过@PathVariable("xxx")绑定到操作方法的入参中。类定位处@RequestMapping 的 URL 如果使用占位符的参数，则也可以绑定到处理方法的入参中，如代码清单 17-10 所示。

代码清单 17-10　UserController.java：使用@PathVariable

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  @RequestMapping("/pets/{petId}")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model
model) {
    ...
  }
}
```

在默认情况下，Java 类的反射对象并未包含方法入参的名称，在 Java 8.0 中可以通过 javac -parameters 生成方法入参的元数据信息，在低版本的 Java 中则可以通过 javac -g 打开生成所有调试信息的开发，这样也会包含方法入参名的元数据信息。所以要使代码清单 17-10 中的 findPet()入参成功绑定 URL 中的占位符参数，必须保证在编译时输出方法名元信息。在 Maven 中可以显式配置 maven-compiler-plugin 编译插件，开启编译输出调试信息的开关。

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>${java.version}</source>
        <target>${java.version}</target>
        <encoding>UTF-8</encoding>
        <debug>true</debug>
    </configuration>
</plugin>
```

不过编译时打开 debug 开关会使目标类变大，对运行效率也有一定的负面影响。正式编译部署时往往将此开关取消，所以最好在@PathVariable 中显式指定绑定的参数名，以避免因编译方式不同造成参数绑定失败的隐患。

2. 通过请求参数、请求方法或请求头进行映射

HTTP 请求报文除 URL 外，还拥有其他众多的信息。以下是一个标准的 HTTP 请求报文，如图 17-4 所示。

图 17-4 HTTP 请求报文

①是请求方法，GET 和 POST 是最常见的 HTTP 方法，除此以外还包括 DELETE、HEAD、OPTIONS、PUT、TRACE。不过，当前的大多数浏览器只支持 GET 和 POST，Spring 提供了一个 HiddenHttpMethodFilter，允许通过_method 表单参数指定这些特殊的 HTTP 方法（实际上还是通过 POST 提交表单）。服务器端配置了 HiddenHttpMethodFilter 后，Spring 会根据_method 参数指定的值模拟出相应的 HTTP 方法，这样就可以使用这些 HTTP 方法对处理方法进行映射了。

②是请求对应的 URL 地址，它和报文头的 Host 属性组成完整的请求 URL。

③是协议名称及版本号。

④是 HTTP 的报文头，报文头包含若干个属性，格式为“属性名:属性值”，服务器端据此获取客户端的信息。

⑤是报文体，它将一个页面表单中的组件值通过 param1=value1&param2=value2 的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求 URL 也可以通过类似于/chapter17/user.html?param1=value1&param2=value2 的方式传递请求参数。

实战经验

HttpWatch 是强大的网页数据分析工具，安装后将集成到 Internet Explorer 工具栏中。它无须代理服务器或一些复杂的网络监控工具，就能抓取请求及响应的完整信息，包括 Cookies、消息头、查询参数、响应报文等，是 Web 应用开发人员的必备工具。笔者有篇网文对 HTTP 有较全面的介绍，地址为 http://www.iteye.com/topic/1124408，欢迎阅读。

@RequestMapping 除了可以使用请求 URL 映射请求外，还可以使用请求方法、请求头参数及请求参数（报文体和 URL 包含的请求参数）映射请求，如代码清单 17-11 所示。

代码清单 17-11　UserController.java：使用其他信息映射请求

```
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path="/delete",method=RequestMethod.POST,params="userId") ①
     public String test1(@RequestParam("userId") String userId){
         //do sth                                    使用请求方法及请求参数映射请求
        return "user/test1";
     }

     @RequestMapping(path="/show",headers="content-type=text/*")②
     public String test2(@RequestParam("userId") String userId){
         //do sth                                    使用报文头映射请求
        return "user/test2";
     }
}
```

@RequestMapping 的 value、method、params 及 headers 分别表示请求 URL、请求方法、请求参数及报文头的映射条件，它们之间是与的关系，联合使用多个条件项可让请求映射更加精确化。

params 和 headers 分别通过请求参数及报文头属性进行映射，它们支持简单的映射表达式。下面以 params 表达式为例进行说明，headers 可以参照 params 来理解。

- "param1"：表示请求须包含名为 param1 的请求参数。
- "!param1"：表示请求不能包含名为 param1 的请求参数。
- "param1!=value1"：表示请求包含名为 param1 的请求参数，但其值不能为 value1。
- {"param1=value1","param2"}：表示请求必须包含名为 param1 和 param2 的两个请求参数，且 param1 参数的值必须为 value1。

17.2.2　请求处理方法签名

@RequestMapping 像一个迎宾，将 HTTP 请求正确地迎接到主人的面前（负责处理该请求的方法）；主人起身相迎，进而与 HTTP 请求交杯换盏、把酒言欢；最后在欢声笑语中送走 HTTP 请求（返回响应）。Spring MVC 通过分析处理方法的签名，将 HTTP 请求信息绑定到处理方法的相应入参中，然后再调用处理方法得到返回值，最后对返回值进行处理并返回响应。

Spring MVC 对控制器处理方法签名的限制是很宽松的，用户几乎可以按自己喜欢的方式进行方法签名。在必要时对方法及方法入参标注相应的注解（如@PathVariable、@RequestParam、@RequestHeader 等）即可，Spring MVC 会优雅地完成剩下的工作：将 HTTP 请求的信息绑定到相应的方法入参中，并根据方法返回值类型做出相应的后续处理。

一般情况下，处理方法的返回值类型为 ModelAndView 或 String，前者包含模型和

逻辑视图名，而后者仅代表一个逻辑视图名。下面来看几个典型的方法签名，如代码清单 17-12 所示。

代码清单 17-12　几种典型的处理方法签名

```
//①请求参数按名称匹配的方式绑定到方法入参中，方法返回的字符串代表逻辑视图名
@RequestMapping(path="/handle1")
public String handle1(@RequestParam("userName") String userName,
                @RequestParam("password") String password,
                @RequestParam("realName") String realName){
    return "success";
}

//②将Cookie值及报文头属性绑定到入参中，方法返回ModelAndView
@RequestMapping(path="/handle2")
public ModelAndView handle2(@CookieValue("JSESSIONID") String sessionId,
                    @RequestHeader("Accept-Language") String accpetLanguage){
    ModelAndView mav = new ModelAndView();
    mav.setViewName("success");
    mav.addObject("user", new User());
    return mav;
}

//③请求参数按名称匹配的方式绑定到user的属性中，方法返回的字符串代表逻辑视图名
@RequestMapping(path="/handle3")
public String handle3(User user){
    return "success";
}


//④直接将HTTP请求对象传递给处理方法，方法返回的字符串代表逻辑视图名
@RequestMapping(path="/handle4")
public String handle4(HttpServletRequest request){
    return "success";
}
```

从代码清单 17-12 中可以发现，Spring MVC 会结合方法入参类型、入参的注解、返回值的类型，按照“契约”进行相应的处理。

17.2.3　使用矩阵变量绑定参数

RFC3986 定义了在 URI 中包含 name-value 的规范。随之在 Spring MVC 3.2 中出现了@MatrixVariable 注解，该注解的出现使得开发人员能够将请求中的矩阵变量（MatrixVariable）绑定到处理器的方法参数中。而 Spring 4.0 更全面地支持这个规范，这也是 Spring 4.0 众多吸引人的新特性之一。接下来我们就一起来了解这个新特性的使用方式。

在 Matrix Variable 中，多个变量可以使用“;”（分号）分隔，例如：

```
/books;author=Tom;year=2016
```

如果一个变量对应多个值，那么可以使用“,”（逗号）分隔，例如：

```
author =smart1, smart2, smart3
```

或者使用重复的变量名，例如：

```
author = smart1; author = smart2; author = smart3
```

下面举一个例子来说明，代码如下：

```
// GET /books/22;a=11;b=22
@RequestMapping(value = "/books/{bookId}", method = RequestMethod.GET)
public void findBookId(@PathVariableString bookId, @MatrixVariable int a) {
    ...
}
```

相应的 bookId 和 a 都会被映射到这个方法中，如果匹配不到，则会报“bad request”。如果 URI 只是“/books /11”，则也可以映射到这个方法中，但需要指定空值不报错：@MatrixVariable（required =false）。

再来看一个更复杂的例子，以深入理解，代码如下：

```
//GET /books/42;a=11/authors/21;q=22
@RequestMapping(value = "/ books /{bookId}/ authors /{ authorId}",
            method = RequestMethod.GET)
public void findBook(
  @MatrixVariable(value="a", pathVar=" bookId ") int q1,
  @MatrixVariable(value="a", pathVar=" authorId ") int q2) {
    // q1 == 11
    // q2 == 22
}
```

针对每个 Path Variable 绑定一个 Matrix Variable，然后使用 value 和 pathVar 属性就能找到该值。

另外，Matrix Variable 也自带了一些属性可供选择，例如，是否必需，默认值。举一个例子来说明，代码如下：

```
// GET /books/42
@RequestMapping(value = "/books/{bookId}", method = RequestMethod.GET)
public voidfindBook(@MatrixVariable(required=true, defaultValue="1") int q) {
   // q == 1
}
```

默认 Matrix Variable 功能是开启的，如果不希望开启该功能，则需要手工将 RequestMappingHandlerMapping 中的 removeSemicolonContent 属性设置为 true，即<mvc: annotation-driven enable-matrix-variables="true"/>。

17.2.4　请求处理方法签名详细说明

本节将学习如何对处理方法进行签名，包括如何设置方法入参以绑定请求信息、如何定义返回值类型、Spring MVC 对不同签名的处理方法如何进行调用等内容。

1. 使用@RequestParam 绑定请求参数值

在 17.2.1 节中我们指出，Java 类反射对象默认不记录方法入参的名称，因此需要在

方法入参处使用@RequestParam 注解指定其对应的请求参数。@RequestParam 注解有以下 3 个参数。

- value：参数名。
- required：是否必需，默认为 true，表示请求中必须包含对应的参数名，如果不存在则抛出异常。
- defaultValue：默认参数名，在设置该参数时，自动将 required 设为 false。极少情况需要使用该参数，也不推荐使用该参数。

下面的实例将 userName 和 age 请求参数分别绑定到 handle11()方法的 userName 和 age 中，并自动完成类型转换。如果不存在 age 请求参数，则将抛出异常。

```
@RequestMapping(path = "/handle11")
public String handle11(
        @RequestParam(value = "userName", required = false) String userName,
        @RequestParam("age") int age) {
    ...
}
```

2．使用@CookieValue 绑定请求中的 Cookie 值

使用@CookieValue 可让处理方法入参绑定某个 Cookie 的值，它和@RequestParam 拥有 3 个一样的参数。来看一个使用@CookieValue 的实例，代码如下：

```
@RequestMapping(path = "/handle12")
public String handle12(
        @CookieValue(value="sessionId",required=false) String sessionId,
        @RequestParam("age") int age) {
    ...
}
```

@CookieValue 的 value 属性指定了 Cookie 的名称，required 为 false，表示请求中没有相应的 Cookie 时也不会报错。

3．使用@RequestHeader 绑定请求报文头的属性值

如图 17-4 所示，请求报文包含了若干个报文头属性，服务器可据此获知客户端的信息，通过@RequestHeader 即可将报文头属性值绑定到处理方法的入参中。

```
@RequestMapping(path = "/handle13")
public String handle13(@RequestHeader("Accept-Encoding") String encoding,
                       @RequestHeader("Keep-Alive") long keepAlive) {
    ...
}
```

@RequestHeader 和@RequestParam 拥有 3 个一样的参数，此处不再赘述。

4．使用命令/表单对象绑定请求参数值

所谓命令/表单对象并不需要实现任何接口，仅是一个拥有若干属性的 POJO。在代码清单 17-12 中，③处的 User 就是一个命令/表单对象。Spring MVC 会按请求参数名和命令/表单对象属性名匹配的方式，自动为该对象填充属性值。支持级联的属性名，如 dept.deptId、dept.address.tel 等。

```
@RequestMapping(path = "/handle14")
public String handle14(User user) {
    ...
}
```

假设 User 类的结构如图 17-5 所示，则如下的 URL 请求，其请求参数将正确地填充到 User 对象中。

```
/handle14.html?userName=tom&dept.deptId=1&dept.address.tel=102
```

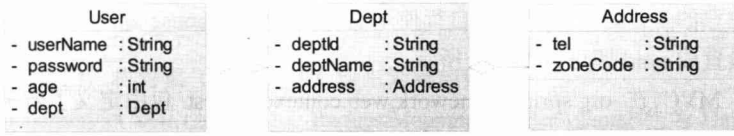


图 17-5　HTTP 请求报文

5. 使用 Servlet API 对象作为入参

在 Spring MVC 中，控制器类可以不依赖任何 Servlet API 对象，但是 Spring MVC 并不能阻止我们使用 Servlet API 的类作为处理方法的入参。以下处理方法都可以正确地工作，如代码清单 17-13 所示。

代码清单 17-13　使用Servlet API作为入参

```
package com.smart.web;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.util.WebUtils;
...
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/handle21")
    public void handle21(HttpServletRequest request,HttpServletResponse response) {①
        String userName = WebUtils.findParameterValue(request, "userName");
        response.addCookie(new Cookie("userName", userName));
    }

    @RequestMapping(path = "/handle22")
    public ModelAndView handle22(HttpServletRequest request) {②
        String userName = WebUtils.findParameterValue(request, "userName");
        ModelAndView mav = new ModelAndView();
        mav.setViewName("success");
        mav.addObject("userName", userName);
        return mav;
    }

    @RequestMapping(path = "/handle23")
    public String handle23(HttpSession session) {③
        session.setAttribute("sessionId", 1234);
        return "success";
    }
    @RequestMapping(path = "/handle24")
    public String handle24(HttpServletRequest request,
                           @RequestParam("userName")String userName) {④
```

同时使用HttpServletRequest/HttpServletResponse作为入参

仅使用HttpServletRequest作为入参

使用HttpSession 作为入参

既使用HttpServletRequest，又使用基本类型的入参

```
        ...
        return "success";
    }
}
```

在使用 Servlet API 的类作为入参时，Spring MVC 会自动将 Web 层对应的 Servlet 对象传递给处理方法的入参。处理方法入参可以同时使用 Servlet API 类的入参和其他符合要求的入参，它们之间的位置顺序没有特殊要求。

值得注意的是，如果处理方法自行使用 HttpServletResponse 返回响应，则处理方法的返回值设置成 void 即可，如①处所示。

Spring MVC 在 org.springframework.web.context.request 包中定义了若干个可代理 Servlet 原生 API 类的接口，如 WebRequest 和 NativeWebRequest，它们也允许作为处理类的入参，通过这些代理类可访问请求对象的任何信息，如下：

```
@RequestMapping(path = "/handle25")
public String handle25(WebRequest request) {
    String userName = request.getParameter("userName");
    return "success";
}
```

6. 使用 I/O 对象作为入参

Servlet 的 ServletRequest 拥有 getInputStream()和 getReader()方法，可以通过它们读取请求的信息。相应的，Servlet 的 ServletResponse 拥有 getOutputStream()和 getWriter()方法，可以通过它们输出响应信息。

Spring MVC 允许控制器的处理方法使用 java.io.InputStream/java.io.Reader 及 java.io.OutputStream/java.io.Writer 作为方法的入参，Spring MVC 将获取 ServletRequest 的 InputStream/Reader 或 ServletResponse 的 OutputStream/Writer，然后传递给控制器的处理方法，如代码清单 17-14 所示。

代码清单 17-14　使用OutputStream输出一张图片

```
package com.smart.web;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.util.FileCopyUtils;

@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/handle31")
    public void handle31(OutputStream os) throws IOException{
        Resource res = new ClassPathResource("/image.jpg");//读取类路径下的图片文件
        FileCopyUtils.copy(res.getInputStream(), os);      //将图片写到输出流中
    }
}
```

7. 其他类型的参数

控制器处理方法的入参除支持以上类型的参数外，还支持 java.util.Locale、java.

security.Principal，可以通过 Servlet 的 HttpServletRequest 的 getLocale()和 getUserPrincipal()方法得到相应的值。如果处理方法的入参类型为 Locale 或 Principal，则 Spring MVC 自动从请求对象中获取相应的对象并传递给处理方法的入参。

17.2.5　使用 HttpMessageConverter<T>

HttpMessageConverter<T>是 Spring 的一个重要接口，它负责将请求信息转换为一个对象（类型为 T），将对象（类型为 T）输出为响应信息。

DispatcherServlet 默认已经安装了 RequestMappingHandlerAdapter 作为 HandlerAdapter 的组件实现类，HttpMessageConverter 即由 RequestMappingHandlerAdapter 使用，将请求信息转换为对象，或将对象转换为响应信息。

HttpMessageConverter<T>接口定义了以下几个方法。

- Boolean canRead(Class<?> clazz, MediaType mediaType)：指定转换器可以读取的对象类型，即转换器可将请求信息转换为 clazz 类型的对象；同时指定支持的 MIME 媒体类型（如 text/html、application/json 等），MIME 媒体类型在 RFC2616 中定义（MIME 类型说明可参见 http://www.w3school.com.cn/media/media_mimeref.asp）。
- Boolean canWrite(Class<?> clazz, MediaType mediaType)：指定转换器可以将 clazz 类型的对象写到响应流中，响应流支持的媒体类型在 mediaType 中定义。
- List<MediaType> getSupportedMediaTypes()：该转换器支持的媒体类型。
- T read(Class<? extends T> clazz, HttpInputMessage inputMessage)：将请求信息流转换为 T 类型的对象。
- void write(T t, MediaType contentType, HttpOutputMessage outputMessage)：将 T 类型的对象写到响应流中，同时指定响应的媒体类型为 contentType。

1．HttpMessageConverter<T>的实现类

Spring 为 HttpMessageConverter<T>提供了众多的实现类，它们组成了一个功能强大、用途广泛的 HttpMessageConverter<T>家族，具体说明如表 17-2 所示。

表 17-2　HttpMessageConverter实现类

实　现　类	功能说明
StringHttpMessageConverter	用途：将请求信息转换为字符串。 （1）T 为 String 类型。 （2）可读取所有媒体类型（*/*）的请求信息，可通过设置 supportedMediaTypes 属性指定媒体类型。 （3）响应信息的媒体类型为 text/plain（即 Content-Type 的值）
FormHttpMessageConverter	用途：将表单数据读取到 MultiValueMap 中。 （1）T 为 org.springframework.util.MultiValueMap<String,?>类型。 （2）支持读取 application/x-www-form-urlencoded 的媒体类型，但不支持读取 multipart/form-data 的媒体类型。

续表

实 现 类	功能说明
FormHttpMessageConverter	（3）可写 application/x-www-form-urlencoded 及 multipart/form-data 媒体类型的响应信息
AllEncompassingFormHttpMessage Converter	扩展于 FormHttpMessageConverter。如果部分表单属性是 XML 数据，则可用该转换器进行读取
ResourceHttpMessageConverter	用途：读/写 org.springframework.core.io.Resource 对象。 （1）T 为 org.springframework.core.io.Resource 类型。 （2）可读取所有媒体类型（*/*）的请求信息。 （3）如果类路径下提供了 JAF（Java Activation Framework），则根据 Resource 的类型指定响应的媒体类型；否则响应的媒体类型为 application/octet-stream
BufferedImageHttpMessageConverter	用途：读/写 BufferedImage 对象。 （1）T 为 BufferedImage 类型。 （2）可以读取所有媒体类型。 （3）返回 BufferedImage 相应的媒体类型，也可以通过 contentType 显式指定
ByteArrayHttpMessageConverter	用途：读/写二进制数据。 （1）T 为 byte[]类型。 （2）可读取所有媒体类型（*/*）的请求信息，可通过设置 supportedMediaTypes 属性指定媒体类型。 （3）响应信息的媒体类型为 application/octet-stream
SourceHttpMessageConverter	用途：读/写 javax.xml.transform.Source 类型的数据。 （1）T 为 javax.xml.transform.Source 类型及其扩展类，包括 javax.xml.transform.dom.DOMSource 、 javax.xml.transform.sax.SAXSource 及 javax.xml.transform.stream.StreamSource。 （2）可读取 text/xml 和 application/xml 媒体类型的请求。 （3）响应信息的媒体类型为 text/xml 或 application/xml
MarshallingHttpMessageConverter	用途：通过 Spring 的 org.springframework.oxm.Marshaller （将 Java 对象转换为 XML）和 Unmarshaller（将 XML 解析为 Java 对象）读/写 XML 消息。 （1）T 为 Object 类型。 （2）可读取 text/xml 和 application/xml 媒体类型的请求。 （3）响应信息的媒体类型为 text/xml 或 application/xml
Jaxb2RootElementHttpMessageConverter	用途：通过 JAXB2 读/写 XML 消息，并将请求消息转换到标注 XmlRootElement 和 XmlType 注解的类中。 （1）T 为 Object 类型。 （2）可读取 text/xml 和 application/xml 媒体类型的请求。 （3）响应信息的媒体类型为 text/xml 或 application/xml
MappingJackson2HttpMessageConverter	用途：利用 Jackson 开源类包的 ObjectMapper 读/写 JSON 数据。 （1）T 为 Object 类型。 （2）可读取 application/json 类型的数据。 （3）响应信息的媒体类型为 application/json
RssChannelHttpMessageConverter	用途：能够读/写 RSS 种子消息。 （1）T 为 com.sun.syndication.feed.rss.Channel 类型。 （2）可读取 application/rss+xml 类型的数据。 （3）响应信息的媒体类型为 application/rss+xml

续表

实　现　类	功能说明
AtomFeedHttpMessageConverter	用途：和 RssChannelHttpMessageConverter 一样，能够读/写 RSS 种子消息。 （1）T 为 com.sun.syndication.feed.atom.Feed 类型。 （2）可读取 application/atom+xml 类型的数据。 （3）响应信息的媒体类型为 application/atom+xml

RequestMappingHandlerAdapter 默认已经装配了以下 HttpMessageConverter：

- StringHttpMessageConverter。
- ByteArrayHttpMessageConverter。
- SourceHttpMessageConverter。
- AllEncompassingFormHttpMessageConverter。

如果需要装配其他类型的 HttpMessageConverter，则可在 Spring 的 Web 容器上下文中自行定义一个 RequestMappingHandlerAdapter，如代码清单 17-15 所示。

代码清单 17-15　smart-servlet.xml

```
...
<!--① 定义一个RequestMappingHandlerAdapter -->
<bean class="org.springframework.web.servlet.mvc
                .method.annotation.RequestMappingHandlerAdapter "
                p:messageConverters-ref="messageConverters"/>

<!--① HttpMessageConverter列表-->
<util:list id="messageConverters">
    <bean class="org.springframework.http.converter.BufferedImageHttpMessageConverter" />
    <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <bean class="org.springframework.http.converter.StringHttpMessageConverter" />
    <bean class="org.springframework.http.converter
                        .support.AllEncompassingFormHttpMessageConverter" />
</util:list>
...
```

如果在 Spring Web 容器中显式定义了一个 RequestMappingHandlerAdapter，则 Spring MVC 将使用它覆盖默认的 RequestMappingHandlerAdapter。

2．使用 HttpMessageConverter<T>

如何使用 HttpMessageConverter<T>将请求信息转换并绑定到处理方法的入参中呢？Spring MVC 提供了两种途径。

- 使用@RequestBody/@ResponseBody 对处理方法进行标注。
- 使用 HttpEntity<T>/ResponseEntity<T>作为处理方法的入参或返回值。

下面分别通过实例进行说明。首先来看使用@RequestBody/@ResponseBody 的例子，如代码清单 17-16 所示。

代码清单 17-16　UserController.java：使用@RequestBody/@ResponseBody

```
package com.smart.web;
...
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.util.FileCopyUtils;

@Controller
@RequestMapping("/user")
public class UserController {

    ...
    @RequestMapping(path = "/handle41")
    public String handle41(@RequestBody String requestBody ) {①
        System.out.println(requestBody);
        return "success";
    }

    @ResponseBody  ②
    @RequestMapping(path = "/handle42/{imageId}")
    public byte[] handle42(@PathVariable("imageId") String imageId) throws IOException {
        System.out.println("load image of "+imageId);
        Resource res = new ClassPathResource("/image.jpg");
        byte[] fileData =FileCopyUtils.copyToByteArray(res.getInputStream());
        return fileData;
    }
}
```

将请求报文体转换为字符串绑定到 requestBody 入参中

读取一张图片，并将图片数据输出到响应流中，客户端将显示这张图片

在代码清单 17-15 中，已经为 RequestMappingHandlerAdapter 注册了若干个 HttpMessageConverter。handle41()方法的 requestBody 入参标注了一个@RequestBody 注解，如①处所示，Spring MVC 将根据 requestBody 的类型查找匹配的 HttpMessageConverter。由于 StringHttpMessageConverter 的泛型类型对应 String，所以 StringHttpMessageConverter 将被 Spring MVC 选中，用它将请求体信息进行转换并将结果绑定到 requestBody 入参上。

handle42()方法拥有一个@ResponseBody 注解，如②处所示。由于方法返回值类型为 byte[]，所以 Spring MVC 根据类型匹配的查找规则将使用 ByteArrayHttpMessageConverter 对返回值进行处理，即将图片数据流输出到客户端。

下面编写一个测试用例，通过 RestTemplate 对 handle41()及 handle42()这两个方法进行测试，如代码清单 17-17 所示。

代码清单 17-17　UserControllerTest.java

```
package com.smart.web;
…import org.junit.Test;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.util.*;
import org.springframework.web.client.RestTemplate;

public class UserControllerTest {

    @Test
    public void testHandle41() {
        RestTemplate restTemplate = new RestTemplate();
```

```
        MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
        form.add("userName", "tom");        第一个参数为URL，第二个参数通过MultiValueMap
        form.add("password", "123456");     准备报文体的参数数据
        form.add("age", "45");
        restTemplate.postForLocation(①
                "http://localhost:8080/chapter17/user/handle41.html",form);
    }

    @Test                                                    第二个参数为报文体参数数据，第三
    public void testHandle42() throws IOException{           个参数指定方法的返回值类型，第四
        RestTemplate restTemplate = new RestTemplate();      个参数为URL占位符参数的值
        byte[] response = restTemplate.postForObject( ②
                "http://localhost:8080/chapter17/user/handle42/{itemId}.html",
              null,byte[].class,"1233");
        Resource outFile = new FileSystemResource("d:/image_copy.jpg");
        FileCopyUtils.copy(response, outFile.getFile());
    }
}
```

RestTemplate 是 Spring 的模板类，在客户端程序中可使用该类调用 Web 服务器端的服务，它支持 REST 风格的 URL。此外，它像 RequestMappingHandlerAdapter 一样拥有一张 HttpMessageConverter 的注册表，RestTemplate 默认已经注册了以下 HttpMessageConverter：

- ByteArrayHttpMessageConverter。
- StringHttpMessageConverter。
- ResourceHttpMessageConverter。
- SourceHttpMessageConverter。
- AllEncompassingFormHttpMessageConverter。

所以，在默认情况下，RestTemplate 就可以利用这些 HttpMessageConverter 对响应数据进行相应的转换处理。可通过 RestTemplate 的 setMessageConverters(List<HttpMessage-Converter<?>> messageConverters)方法手工注册 HttpMessageConverter。

和@RequestBody/@ResponseBody 类似，HttpEntity<?>不但可以访问请求和响应报文体的数据，还可以访问请求和响应报文头的数据。Spring MVC 根据 HttpEntity 的泛型类型查找对应的 HttpMessageConverter。

使用 HttpEntity<?>对代码清单 17-16 中的两个方法进行改造，完成相似的功能，如代码清单 17-18 所示。

代码清单 17-18　UserController.java：使用HttpEntity<?>

```
@RequestMapping(path = "/handle43")
public String handle43(HttpEntity<String> httpEntity){                    ①
    long contentLen = httpEntity.getHeaders().getContentLength();
    System.out.println(httpEntity.getBody());
    return "success";
}                          使用StringHttpMessageConverter将请求报文体及报文头的
                           信息绑定到httpEntity中，在方法中可以对相应信息进行访问


@RequestMapping(path = "/handle44/{imageId}")
public ResponseEntity<byte[]> handle44(
```

```
            @PathVariable("imageId")String imageId) throws Throwable {
    Resource res = new ClassPathResource("/image.jpg");
    byte[] fileData =FileCopyUtils.copyToByteArray(res.getInputStream());
    ResponseEntity<byte[]> responseEntity =
            new ResponseEntity<byte[]>(fileData,HttpStatus.OK);
    return responseEntity;
}
```

②

在方法中创建 HttpEntity<byte[]> 对象并返回，ByteArrayHttpMessage Converter 负责将其输出到响应流中

在①处使用 HttpEntity<String>指定入参的类型，Spring MVC 分析出泛型类型为 String，使用 StringHttpMessageConverter 将请求体内容绑定到 httpEntity 中，返回的 String 类型的值为逻辑视图名。

②处的处理方法返回值类型为 ResponseEntity<byte[]>，Spring MVC 分析出泛型类型为 byte[]，使用 ByteArrayHttpMessageConverter 输出图片数据流。

通过以上两个实例，可以得出以下几条结论。

- 当控制器处理方法使用@RequestBody/@ResponseBody 或 HttpEntity<T>/ResponseEntity<T>时，Spring MVC 才使用注册的 HttpMessageConverter 对请求/响应消息进行处理。
- 当控制器处理方法使用@RequestBody/@ResponseBody 或 HttpEntity<T>/ResponseEntity<T>时，Spring 首先根据请求头或响应头的 Accept 属性选择匹配的 HttpMessageConverter，然后根据参数类型或泛型类型的过滤得到匹配的 HttpMessageConverter，如果找不到可用的 HttpMessageConverter 则报错。
- @RequestBody 和@ResponseBody 不需要成对出现。如果方法入参使用了@RequestBody，则 Spring MVC 选择匹配的 HttpMessageConverter 将请求消息转换并绑定到该入参中。如果处理方法标注了@ResponseBody，则 Spring MVC 选择匹配的 HttpMessageConverter 将方法返回值转换并输出响应消息。
- HttpEntity<T>/ResponseEntity<T>的功能和@RequestBody/@ResponseBody 相似。

3．处理 XML 和 JSON

Spring MVC 提供了几个处理 XML 和 JSON 格式的请求/响应消息的 HttpMessage Converter。

- MarshallingHttpMessageConverter：处理 XML 格式的请求或响应消息。
- Jaxb2RootElementHttpMessageConverter：同上，底层使用 JAXB。
- MappingJackson2HttpMessageConverter：处理 JSON 格式的请求或响应消息。

因此，只要在 Spring Web 容器中为 RequestMappingHandlerAdapter 装配好相应的处理 XML 和 JSON 格式的请求/响应消息的 HttpMessageConverter，并在交互中通过请求的 Accept 指定 MIME 类型，Spring MVC 就可使服务器端的处理方法和客户端透明地通过 XML 或 JSON 格式的消息进行通信，开发者几乎无须关心通信层数据格式的问题，可以将精力集中到业务层的处理上。单就这一点而言，其他 MVC 框架和 Spring MVC 相比，就如诸葛亮给关云长的评语一样：“犹未及美髯公之绝伦超群也。”

首先为 RequestMappingHandlerAdapter 装配可处理 XML 和 JSON 格式的请求/响应消息的 HttpMessage Converter，如代码清单 17-19 所示。

代码清单 17-19　smart-servlet.xml

```
...
<bean class="org.springframework.web.servlet.mvc
      .method.annotation.RequestMappingHandlerAdapter"
      p:messageConverters-ref="messageConverters"/>
<util:list id="messageConverters">
    <bean class="org.springframework.http.converter.BufferedImageHttpMessageConverter" />
    <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <bean class="org.springframework.http.converter.StringHttpMessageConverter" />
    <bean class="org.springframework.http.converter.
                        support.AllEncompassingFormHttpMessageConverter"/>
    <bean class="org.springframework.http.converter.xml
                                .MarshallingHttpMessageConverter"
          p:marshaller-ref="xmlMarshaller"
          p:unmarshaller-ref="xmlMarshaller">
    </bean>
    <bean class="org.springframework.http.converter.json
                              .MappingJackson2HttpMessageConverter"/>
</util:list>

<!--①声明Marshaller，使用XStream技术-->
<bean id="xmlMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="streamDriver">
        <bean class="com.thoughtworks.xstream.io.xml.StaxDriver"/>②
    </property>                                          使用 STAX 对 XML 消息进行处理，
                                                         STAX 占用内存少，响应速度也很快

<property name="annotatedClasses">          使用 XStream 的注解定义 XML 转换规则，
                                            使用 XStream 注解的类在此声明
        <list>
            <value>com.smart.domain.User</value> ③
        </list>
    </property>
</bean>
...
```

然后在控制器中编写相应的方法，如代码清单 17-20 所示。

代码清单 17-20　UserController.java：支持XML和JSON格式的消息处理方法

```
@RequestMapping(path = "/handle51")
public ResponseEntity<User> handle51(HttpEntity<User> requestEntity){
    User user = requestEntity.getBody();
    user.setUserId("1000");
    return new ResponseEntity<User>(user,HttpStatus.OK);
}
```

对于服务器端的处理方法而言，除使用@RequestBody/@ResponseBody 或 HttpEntity<T>/ResponseEntity<T>进行方法签名外，不需要进行任何额外的处理，借由 Spring MVC 中装配的 HttpMessageConverter，它便拥有了处理 XML 及 JSON 格式的消息的能力。

在接收到一个 HTTP 请求时，handle51()如何知道请求消息的格式？在处理完成后，又根据什么确定响应消息的格式？答案很简单：通过请求消息头的 Content-Type 及

Accept 属性确定。下面使用 RestTemplate 编写调用 handle51()方法的客户端程序，如代码清单 17-21 所示。

代码清单 17-21　UserControllerTest.java：使用XML格式的请求/响应消息

```
package com.smart.web;
…
import com.smart.domain.User;
import com.thoughtworks.xstream.io.xml.StaxDriver;

public class UserControllerTest {

    //①使用RestTemplate测试UserController#handle51()方法
    @Test
    public void testHandle51() throws IOException{
        RestTemplate restTemplate = buildRestTemplate();

        User user = new User();
        user.setUserName("tom");
        user.setPassword("1234");
        user.setRealName("汤姆");                    ①-1

        HttpHeaders entityHeaders = new HttpHeaders();
        entityHeaders.setContentType(MediaType.valueOf("application/xml;UTF-8"));   ①-2
        entityHeaders.setAccept(Collections.singletonList(MediaType.
APPLICATION_XML));

        HttpEntity<User> requestEntity = new HttpEntity<User>(user,entityHeaders);
        ResponseEntity<User> responseEntity = restTemplate.exchange(                ①-3
                "http://localhost:8080/chapter17/user/handle51.html",
                HttpMethod.POST,requestEntity,User.class);

       User responseUser = responseEntity.getBody();
        Assert.assertNotNull(responseUser);
        Assert.assertEquals("1000", responseUser.getUserId());
        Assert.assertEquals("tom", responseUser.getUserName());
        Assert.assertEquals("汤姆", responseUser.getRealName());                    ①-4
    }

    //②创建RestTemplate实例
    private RestTemplate buildRestTemplate() {
        RestTemplate restTemplate = new RestTemplate();

        XStreamMarshaller xmlMarshaller = new XStreamMarshaller();
        xmlMarshaller.setStreamDriver(new StaxDriver());
        xmlMarshaller.setAnnotatedClasses(new Class[]{User.class});                 ②-1

        MarshallingHttpMessageConverter xmlConverter =
                    new MarshallingHttpMessageConverter();
        xmlConverter.setMarshaller(xmlMarshaller);
        xmlConverter.setUnmarshaller(xmlMarshaller);
        restTemplate.getMessageConverters().add(xmlConverter);                      ②-2
```

创建 User 对象，它将通过 RestTemplate 流化为 XML 请求报文

指定请求的报文头信息

将 User 流化为 XML，放到报文体中，同时指定请求方法及报文头

将请求响应消息转换为 User 对象，并对响应值进行判断

使用 XStream 流化器，使用 STAX 技术处理 XML，同时加载使用了 XSteam 注解的 User 类

创建处理 XML 报文的 HttpMessageConverter，将其组装到 RestTemplate 中

```
                                        创建处理JSON报文的HttpMessageConverter，
                                        将其组装到RestTemplate中
        MappingJackson2HttpMessageConverter jsonConverter =
                new MappingJackson2HttpMessageConverter();
        restTemplate.getMessageConverters().add(jsonConverter);  ②-3
        return restTemplate;
    }
}
```

服务器端启动 Web 服务，运行 testhandle51WithXml()测试方法，使用网络监控工具（如 TcpTrace）拦截请求响应报文，如图 17-6 所示。

请求报文	POST /chapter17/user/handle51.html HTTP/1.1 **Accept: application/xml** **Content-Type: application/xml;charset=UTF-8** User-Agent: Java/1.6.0_14 Host: localhost:8080 Connection: keep-alive Content-Length: 101 <?xml version="1.0" ?><message userName="tom" password="1234" realName="汤姆" salary="0"></message>
响应报文	HTTP/1.1 200 OK Server: Apache-Coyote/1.1 **Content-Type: application/xml** Transfer-Encoding: chunked Date: Mon, 12 Sep 2015 12:01:46 GMT 6f <?xml version="1.0" ?><message id="1000" userName="tom" password="1234" realName="汤姆" salary="0"></message> 0

图 17-6　HTTP 请求响应报文（XML 格式）

通过以上 HTTP 请求/响应报文，我们清楚地知道客户端的 User 对象被流化为一段对应的 XML 报文（阴影部分），同时通过报文头属性 Accept 和 Content-Type 指定接收的 MIME 类型和本请求的报文内容均为 application/xml。

请求报文被服务器端的 UserController#handle51()方法正确处理，它根据请求的报文头属性 Accept 决定将服务器端的 User 对象流化为 XML 并返回 HTTP 响应报文（阴影部分），同时指定响应报文的 Content-Type 属性为 application/xml。

如果希望通过 JSON 方式进行通信，则仅需对客户端代码进行轻微的调整即可，服务器端代码无须作任何更改，如代码清单 17-22 所示。

代码清单 17-22　UserControllerTest.java：使用JSON格式的请求/响应消息

```
public void testhandle51(){
    @Test
    RestTemplate restTemplate = buildRestTemplate();

    ...
    HttpHeaders entityHeaders = new HttpHeaders();
    entityHeaders.setContentType(MediaType.valueOf("application/json;UTF-8"));
    entityHeaders.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));
    HttpEntity<User> requestEntity = new HttpEntity<User>(user,entityHeaders);
```

```
    ...
}
```

将请求报文头的 Content-Type 及 Accept 属性更改为 application/json 即可。再次执行 testhandle51()方法，观察 HTTP 请求/响应报文，如图 17-7 所示。

请求报文

```
POST /chapter17/user/handle51.html HTTP/1.1
Accept: application/json
Content-Type: application/json
User-Agent: Java/1.6.0_14
Host: localhost:8080
Connection: keep-alive
Content-Length: 109

{"userId":null,"realName":"tomson","birthday":null,"salary":0,"dept":null,"password":"1234","userName":"tom"}
```

响应报文

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json
Transfer-Encoding: chunked
Date: Mon, 12 Sep 2015 12:41:24 GMT

6f
{"password":"1234","userName":"tom","birthday":null,"salary":0,"realName":"tomson","userId":"1000","dept":null}
0
```

图 17-7　HTTP 请求响应报文（JSON 格式）

可见，请求报文头的 Content-Type 及 Accept 属性更改为 application/json，User 对象的数据以 JSON 格式进行传递。

17.2.6　使用@RestController 和 AsyncRestTemplate

1. @RestController

从 Spring 4.0 开始，Spring 以 Servlet 3.0 为基础进行开发。如果使用 Spring MVC 测试框架，则需要指定 Servlet 3.0 兼容的 JAR 包（因为其 Mock 的对象都是基于 Servlet 3.0 的）。为方便 Rest 的开发，Spring 引入了一个新的@RestController 注解，该注解已经标注了@ResponseBody 和@Controller。

```
@ Controller
@ ResponseBody
public @interface RestController {
}
```

这样，通过直接在控制器上标注新的@RestController，就不需要在每个@RequestMapping 方法上添加@ResponseBody 了。

```
@RestController
public class UserController {
}
```

当我们使用 REST 风格开发应用程序时，Spring MVC 仅需以下两行配置就可以了：

```
<context:component-scan base-package="com.smart.* "/>
<mvc:annotation-driven/>
```

2. AsyncRestTemplate

Spring 4.0 添加了一个 AsyncRestTemplate，支持以异步无阻塞方式进行服务访问。以下是服务器端的 Rest 服务实现类，如代码清单 17-23 所示。

代码清单 17-23　UserController.java

```
@RestController
public class UserController {
   private UserService userService;
   @Autowired
   public UserController(UserService userService) {
      this.userService = userService;
   }
   @RequestMapping("/api")
   public Callable<User> api() {
      System.out.println("=====hello");
      return new Callable<User>() {
         @Override
         public User call() throws Exception {
            Thread.sleep(10L * 1000); //①暂停10秒
            User user = new User();
            user.setId(1L);
            user.setName("haha");
            return user;
         }
      };
   }
}
```

在这里，我们模拟一个执行时间为 10 秒的服务器方法，如果客户端使用 RestTemplate，则将以同步方式进行调用，即客户端代码需要等待服务器端返回后才继续执行。下面使用 AsyncRestTemplate 以异步的方式进行服务调用，如代码清单 17-24 所示。

代码清单 17-24　main()方法

```
public static void main(String[] args) {
       AsyncRestTemplate template = new AsyncRestTemplate();

       //①调用完后立即返回（没有阻塞）
       ListenableFuture<ResponseEntity<User>> future =
            template.getForEntity("http://localhost:8080/chapter17/api",User.class);

       //②处理服务器端响应的异步回调方法
       future.addCallback(new ListenableFutureCallback<ResponseEntity<User>>() {
          @Override
          public void onSuccess(ResponseEntity<User> result) {
             System.out.println("======client get result : " + result.getBody());
          }

          @Override
          public void onFailure(Throwable t) {
             System.out.println("======client failure : " + t);
          }
       });
       System.out.println("==no wait");
}
```

①处的执行会立即返回，不会同步阻塞。待服务器端返回请求响应后，②处注册的回调函数会被自动异步调用。

AsyncRestTemplate 默认使用 SimpleClientHttpRequestFactory 进行 HTTP 操作，其底层通过 java.net.HttpURLConnection 实现。也可以使用其他的实现方式，如 template.setAsyncRequestFactory(new HttpComponentsAsyncClientHttpRequestFactory())语句将使用 Apache 的 http components 作为底层访问组件。

17.2.7 处理模型数据

对于 MVC 框架来说，模型数据是最重要的，因为控制（C）是为了产生模型数据（M），而视图（V）则是为了渲染模型数据。

通过前面的学习我们已经知道，Spring MVC 通过@RequestMapping 将请求引导到处理方法上，使用合适的方法签名将请求消息绑定到入参中。方法入参绑定请求消息只是处理方法的第一步，还有更为重要的任务等待完成，即根据入参执行相应的逻辑，产生模型数据，导向到特定视图中。

将模型数据暴露给视图是 Spring MVC 框架的一项重要工作。Spring MVC 提供了多种途径输出模型数据，介绍如下。

- ModelAndView：当处理方法返回值类型为 ModelAndView 时，方法体即可通过该对象添加模型数据。
- @ModelAttribute：在方法入参标注该注解后，入参的对象就会放到数据模型中。
- Map 及 Model：如果方法入参为 org.springframework.ui.Model、org.springframework.ui.ModelMap 或 java.util.Map，则当处理方法返回时，Map 中的数据会自动添加到模型中。
- @SessionAttributes：将模型中的某个属性暂存到 HttpSession 中，以便多个请求之间可以共享这个属性。

1. ModelAndView

控制器处理方法的返回值如果为 ModelAndView，则其既包含视图信息，又包含模型数据信息，这样 Spring MVC 就可以使用视图对模型数据进行渲染了。可以简单地将模型数据看成一个 Map<String,Object>对象。

在处理方法的方法体中，可以使用如下方法添加模型数据。

- ModelAndView addObject(String attributeName, Object attributeValue)。
- ModelAndView addAllObjects(Map<String,?> modelMap)。

可以通过如下方法设置视图。

- void setView(View view)：指定一个具体的视图对象。
- void setViewName(String viewName)：指定一个逻辑视图名。

ModelAndView 的使用非常简单，请参照 17.1.3 节的实例。

2. @ModelAttribute

如果希望将方法入参对象添加到模型中，则仅需在相应入参前使用@ModelAttribute注解即可。来看一个具体的实例，如代码清单 17-25 所示。

代码清单 17-25　UserController.java：在方法入参前使用@ModelAttribute注解

```
@RequestMapping(path = "/handle61")
public String  handle61(@ModelAttribute("user") User user){
    user.setUserId("1000");
    return "/user/createSuccess";
}
```

Spring MVC 将请求消息绑定到 User 对象中，然后再以 user 为键将 User 对象放到模型中。在准备对视图进行渲染前，Spring MVC 还会进一步将模型中的数据转储到视图的上下文中并暴露给视图对象。对于 JSP 视图来说，Spring MVC 会将模型数据转储到 ServletRequest 的属性列表中（通过 ServletRequest#setAttribute(String name, Object o)方法保存）。

handle61()方法返回的逻辑视图名为/user/createSuccess，对应 createSuccess.jsp 视图对象，这样 createSuccess.jsp 就可以使用${user.userName}等方式顺利地访问到模型中的数据了。

除了可以在方法入参上使用@ModelAttribute 注解外，还可以在方法定义中使用@ModelAttribute 注解。Spring MVC 在调用目标处理方法前，会先逐个调用在方法级上标注了@ModelAttribute 注解的方法，并将这些方法的返回值添加到模型中。下面是在方法级上使用@ModelAttribute 注解的实例，如代码清单 17-26 所示。

代码清单 17-26　UserController.java：在方法定义处使用@ModelAttribute注解

```
@ModelAttribute("user")
public User getUser(){
    User user = new User();
    user.setUserId("1001");
    return user;
}

@RequestMapping(path = "/handle62")
public String  handle62(@ModelAttribute("user") User user){
    user.setUserName("tom");
    return "/user/showUser";
}
```

在访问 UserController 中的任何一个请求处理方法前，Spring MVC 先执行该方法，并将返回值以 user 为键添加到模型中

①

在此，模型数据会赋给 User 的入参，然后再根据 HTTP 请求消息进一步填充覆盖 user 对象

②

在访问 UserController 中的任何一个请求处理方法前，都会事先执行标注了@ModelAttribute 的 getUser()方法，并将其返回值以 user 为键添加到模型中。

由于②处的 handle62()方法使用了入参级的@ModelAttribute 注解，且属性名和①处方法级@ModelAttribute 的属性名相同。这时，Spring MVC 会将①处获取的模型属性先赋值给②处的入参 user，然后再根据 HTTP 请求消息对 user 进行填充覆盖，得到一个整合版本的 user 对象。

提示

处理方法入参最多只能使用一个 Spring MVC 的注解，如 handle62(@ModelAttribute("user") User user)的 user 入参使用了@ModelAttribute，就不能再使用@RequestParam 或@CookieValue。如果使用了两个注解，则 Spring MVC 将抛出异常。

3. Map 及 Model

Spring MVC 在内部使用一个 org.springframework.ui.Model 接口存储模型数据，它的功能类似于 java.util.Map，但它比 Map 易用。org.springframework.ui.ModelMap 实现了 Map 接口，而 org.springframework.ui.ExtendedModelMap 扩展于 ModelMap 的同时实现了 Model 接口。

Spring MVC 在调用方法前会创建一个隐含的模型对象，作为模型数据的存储容器，我们称之为“隐含模型”。如果处理方法的入参为 Map 或 Model 类型，则 Spring MVC 会将隐含模型的引用传递给这些入参。在方法体内，开发者可以通过这个入参对象访问到模型中的所有数据，也可以向模型中添加新的属性数据。来看一个简单的例子，如代码清单 17-27 所示。

代码清单 17-27　UserController.java：使用ModelMap的入参

```
@ModelAttribute("user")
public User getUser(){
    User user = new User();
    user.setUserId("1001");
    return user;
}

@RequestMapping(path = "/handle63")
public String  handle63(ModelMap modelMap){
    modelMap.addAttribute("testAttr","value1");
    User user = (User)modelMap.get("user");
    user.setUserName("tom");
    return "/user/showUser";
}
```

Spring MVC 将请求对应的隐含模型对象传递给 modelMap，因此在方法中可以通过它访问模型中的数据

①

Spring MVC 一旦发现处理方法有 Map 或 Model 类型的入参，就会将请求内在的隐含模型对象传递给这些参数，因此在方法体中可以通过这个入参对模型中的数据进行读/写操作。

4. @SessionAttributes

如果希望在多个请求之间共用某个模型属性数据，则可以在控制器类中标注一个@SessionAttributes，Spring MVC 会将模型中对应的属性暂存到 HttpSession 中。来看一个实例，如代码清单 17-28 所示。

代码清单 17-28　UserController.java：使用@SessionAttributes注解

```
package com.smart.web;

import org.springframework.web.bind.annotation.ModelAttribute;
```

```
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

@Controller
@RequestMapping("/user")
@SessionAttributes("user") ①
public class UserController {

    @RequestMapping(path = "/handle71")
    public String  handle71(@ModelAttribute("user") User user){ ②
        user.setUserName("John");
        return "redirect:/user/handle72.html";
    }

    @RequestMapping(path = "/handle72")
    public String  handle72(ModelMap modelMap,SessionStatus sessionStatus){
        User user = (User)modelMap.get("user"); ③
        if(user != null){
            user.setUserName("Jetty");
            sessionStatus.setComplete();④
        }
        return "/user/showUser";
    }
}
```

将②处的模型属性自动保存到 HttpSession 中

读取模型中的数据

让 Spring MVC 清除本处理器对应的会话属性

在①处标注的@SessionAttributes("user")会自动将本处理器中任何处理方法属性名为 user 的模型属性透明地存储到 HttpSession 中。在②处，handle71()方法的 User user 入参会添加到隐含模型中，于是这个模型属性在 handle71()方法执行时，会由 Spring MVC 将其透明地保存到 HttpSession 中。

handle71()返回的逻辑视图名为 redirect:handle72.html，它将发起另一个请求，而这个请求由 handle72()负责处理。handle72()和 handle71()位于不同的请求上下文中，之所以在③处可以获取名为 user 的模型属性，就是因为@SessionAttributes("user")透明地将 handle71()的 user 模型属性存储到 HttpSession 中，而 handle72()的隐含模型又自动从 HttpSession 中获取到这个模型属性。

提示

我们知道，在一般情况下，控制器方法返回字符串类型的值会被当成逻辑视图名处理。但如果字符串带“forward:”或“redirect:”前缀，则 Spring MVC 将对它们进行特殊处理：将“forward:”或“redirect:”当成指示符，其后的字符作为 URL 处理，如“redirect:handle72.html”或“forward:http://www.baidu.com”。redirect 会让浏览器发起一个新的请求，而 forward 请求与当前请求同属一个请求。

handle72()方法还包含一个 SessionStatus 入参，当调用 SessionStatus#setComplete()方法时，Spring MVC 会清除该控制器类的所有会话属性；否则这个会话属性会一直保存在 HttpSession 中。

很可惜，当启动 Web 应用并向 handle71()发送请求时，Spring MVC 会抛出以下异常信息：

```
org.springframework.web.HttpSessionRequiredException: Session attribute 'user'
required - not found in session
...
```

这个异常曾令笔者大伤脑筋，因为 Spring 仅宣称，@SessionAttributes 的作用是将处理方法对应的模型属性透明地保存到 HttpSession 中，并没有要求 HttpSession 中必须事先拥有对应的模型属性。通过研究 Spring MVC 的源码，才找到了问题的答案。

原来 Spring MVC 对@ModelAttribute 及@SessionAttributes 的处理遵循一个特定的流程，当流程条件不满足时就会报错。这个处理流程简单说明如下。

① Spring MVC 在调用处理方法前，在请求线程中自动创建一个隐含的模型对象。

② 调用所有标注了@ModelAttribute 的方法，并将方法返回值添加到隐含模型中。

③ 查看 Session 中是否存在@SessionAttributes("xxx")所指定的 xxx 属性，如果有，则将其添加到隐含模型中。如果隐含模型中已经有 xxx 属性，则该步操作会覆盖模型中已有的属性值。

④ 对标注了@ModelAttribute("xxx")处理方法的入参按如下流程处理。

4.1 如果隐含模型拥有名为 xxx 的属性，则将其赋给该入参，再用请求消息填充该入参对象直接返回，否则转到4.2。

4.2 如果 xxx 是会话属性，即在处理类定义处标注了@SessionAttributes("xxx")，则尝试从会话中获取该属性，并将其赋给该入参，然后再用请求消息填充该入参对象。**如果在会话中找不到对应的属性，则抛出 HttpSessionRequiredException 异常。**否则转到4.3。

4.3 如果隐含模型中不存在 xxx 属性，且 xxx 也不是会话属性，则创建入参的对象实例，然后再用请求消息填充该入参。

分析代码清单 17-28，由于在①处标注了@SessionAttributes("user")，因此 user 为会话属性，Spring MVC 在对 handle71(@ModelAttribute("user") User user)进行处理时，会先在隐含模型中查询是否有对应的属性，如果不存在，则继续在会话中查询该属性。由于在会话中也不存在该属性，因此报 HttpSessionRequiredException 异常。

解决该异常的方法很简单，只需添加一个标注@ModelAttribute("user")的方法，以便让 Spring MVC 在处理 handle71(@ModelAttribute("user") User user)方法前先向隐含模型中添加 user 属性，这样4.1步就会执行，而4.2步不会执行，这样就不会抛出 HttpSessionRequiredException 异常，如代码清单 17-29 所示。

代码清单 17-29　UserController.java：解决HttpSessionRequiredException异常问题

```
package com.smart.web;

import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

@Controller
@RequestMapping("/user")
@SessionAttributes("user")
```

```
public class UserController {

    @ModelAttribute("user")
    public User getUser(){
        User user = new User();
        user.setUserId("1001");
        return user;
    }

    @RequestMapping(path = "/handle71")
    public String  handle71(@ModelAttribute("user") User user){
        user.setUserName("John");
        return "redirect:handle72.html";
    }

    @RequestMapping(path = "/handle72")
    public String  handle72(ModelMap modelMap,SessionStatus sessionStatus){
        User user = (User)modelMap.get("user");
        if(user != null){
            user.setUserName("Jetty");
            sessionStatus.setComplete();
        }
        return "/user/showUser";
    }
}
```

该方法会向隐含模型中添加一个名为 user 的模型属性

①

@SessionAttributes 除了可以通过属性名指定需要放到会话中的属性外，还可以通过模型属性的对象类型指定哪些模型属性需要放到会话中。如@SessionAttributes(types=User.class)会将隐含模型中所有类型为 User.class 的属性添加到会话中。

此外，@SessionAttributes 还可以通过属性名及 types 一起指定，二者都允许多值，放到会话中的属性是二者的并集。如以下声明方式都是合法的。

- @SessionAttributes(value={"user1","user2"})：将名为 user1 及 user2 的模型属性添加到会话中。
- @SessionAttributes(types={User.class,Dept.class})：将模型中所有类型为 User 及 Dept 的属性添加到会话中。
- @SessionAttributes(value={"user1","user2"},types={Dept.class})：将名为 user1 及 user2 的模型属性添加到会话中，同时将所有类型为 Dept 的模型属性添加到会话中。

17.3　处理方法的数据绑定

在 17.2 节中我们知道，Spring 会根据请求方法签名的不同，将请求消息中的信息以一定的方式转换并绑定到请求方法的入参中。当请求消息到达真正需要调用的方法时（如指定的业务方法），Spring MVC 还有很多工作要做，包括数据转换、数据格式化及数据校验等。

17.3.1 数据绑定流程剖析

Spring MVC 通过反射机制对目标处理方法的签名进行分析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 DataBinder，其运行机制描述如图 17-8 所示。

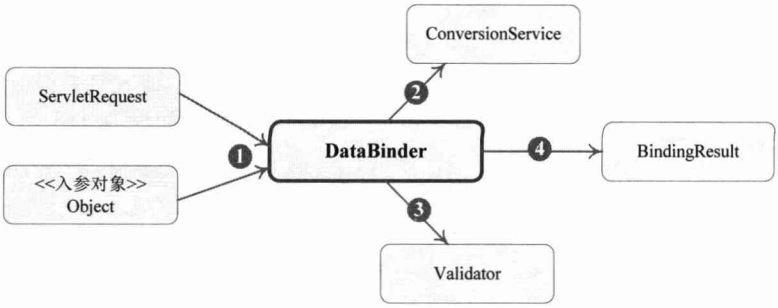


图 17-8 数据绑定

Spring MVC 主框架将 ServletRequest 对象及处理方法的入参对象实例传递给 DataBinder，DataBinder 首先调用装配在 Spring Web 上下文中的 ConversionService 组件进行数据类型转换、数据格式化等工作，将 ServletRequest 中的消息填充到入参对象中，然后调用 Validator 组件对已经绑定了请求消息数据的入参对象进行数据合法性校验，最终生成数据绑定结果 BindingResult 对象。BindingResult 包含了已完成数据绑定的入参对象，还包含相应的校验错误对象。Spring MVC 抽取 BindingResult 中的入参对象及校验错误对象，将它们赋给处理方法的相应入参。

后面我们将对数据绑定过程中发生的数据转换、数据格式化及数据校验等工作进行详细阐述。

17.3.2 数据转换

在第 6 章中，我们已经学习了如何编写并装配自定义属性编辑器的知识。Java 标准的 PropertyEditor 的核心功能是将一个字符串转换为一个 Java 对象，以便根据界面的输入或配置文件中的配置字符串构造出一个 JVM 内部的 Java 对象。

但是 Java 原生的 PropertyEditor 存在以下不足：

- 只能用于字符串到 Java 对象的转换，不适用于任意两个 Java 类型之间的转换。
- 对源对象及目标对象所在的上下文信息（如注解、所在宿主类的结构等）不敏感，在类型转换时不能利用这些上下文信息实施高级的转换逻辑。

鉴于此，Spring 在核心模型中添加了一个通用的类型转换模块，类型转换模块位于 org.springframework.core.convert 包中。Spring 希望用这个类型转换体系替换 Java 标准的 PropertyEditor。但由于历史原因，Spring 将同时支持两者，在 Bean 配置、Spring MVC 处理方法入参绑定中使用它们。

1. ConversionService

ConversionService 是 Spring 类型转换体系的核心接口，它位于 org.springframework.core.convert 包中，也是该包中的唯一一个接口。ConversionService 接口定义了以下 4 个方法。

- boolean canConvert(Class<?> sourceType, Class<?> targetType)：判断是否可以将一个 Java 类转换为另一个 Java 类。
- Boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType)：需转换的类将以成员变量的方式出现在宿主类中。TypeDescriptor 不但描述了需转换类的信息，还描述了从宿主类的上下文信息，如成员变量上的注解，成员变量是否以数组、集合或 Map 的方式呈现等。类型转换逻辑可以利用这些信息做出各种灵活的控制。
- <T> T convert(Object source, Class<T> targetType)：将原类型对象转换为目标类型对象。
- Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType)：将对象从原类型对象转换为目标类型对象，此时往往会用到所在宿主类的上下文信息。

第一个和第三个接口方法类似于 PropertyEditor，它们不关注类型对象所在的上下文信息，只简单地完成两个类型对象的转换，唯一的区别在于这两个方法支持任意两个类型的转换。而第二个和第四个接口方法会参考类型对象所在宿主类的上下文信息，并利用这些信息进行类型转换。

可以利用 org.springframework.context.support.ConversionServiceFactoryBean 在 Spring 的上下文中定义一个 ConversionService。Spring 将自动识别出上下文中的 ConversionService，并在 Bean 属性配置及 Spring MVC 处理方法入参绑定等场合使用它进行数据转换。具体配置如下：

```
<bean id="conversionService"
     class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

该 FactoryBean 创建 ConversionService 内建了很多转换器，可完成大多数 Java 类型的转换工作。除了包括将 String 对象转换为各种基础类型的对象外，还包括 String、Number、Array、Collection、Map、Properties 及 Object 之间的转换器。

可通过 ConversionServiceFactoryBean 的 converters 属性注册自定义的类型转换器，代码如下：

```
<bean id="conversionService"
     class="org.springframework.context.support.ConversionServiceFactoryBean">
     <property name="converters">
        <list>
            <bean class="com.smart.MyCustomConverter1"/>
          <bean class="com.smart.MyCustomConverter2"/>
        </list>
     </property>
</bean>
```

在 6.2 节中我们知道，通过 CustomEditorConfigurer 注册的自定义属性编辑器必须实现 PropertyEditor 接口。现在注册到 ConversionServiceFactoryBean 中的自定义转换器必须实现哪些接口，并满足哪些约定呢？我们将在接下来的小节中进行说明。

2. Spring 支持哪些转换器

Spring 在 org.springframework.core.convert.converter 包中定义了 3 种类型的转换器接口，实现任意一个转换器接口都可以作为自定义转换器注册到 ConversionServiceFactoryBean 中。这 3 种类型的转换器接口分别为：

- Converter<S,T>。
- GenericConverter。
- ConverterFactory。

首先需要了解的转换器接口是 Converter 接口，它是 Spring 中最简单的一个转换器接口，仅包括一个接口方法。该接口定义如下：

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> {
   T convert(S source);
}
```

T convert(S source)负责将 S 类型的对象转换为 T 类型的对象。如果希望将一种类型的对象转换为另一种类型及其子类的对象，举例来说，将 String 转换为 Number 及 Number 子类（Integer、Long、Double 等）对象，就需要一系列的 Converter，如 StringToInteger、StringToLong 及 StringToDouble 等。Spring 提供了一个将相同系列多个“同质”Converter 封装在一起的 ConverterFactory 接口，定义如下：

```
package org.springframework.core.convert.converter;
public interface ConverterFactory<S, R> {
   <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

S 为转换的源类型，R 为目标类型的基类，而 T 为扩展于 R 基类的类型。如 Spring 的 StringToNumberConverterFactory 就实现了 ConverterFactory 接口，封装了 String 转换到各种数据类型的 Converter。

Converter 只负责将一个类型的对象转换为另一个类型的对象，并没有考虑类型对象所在宿主类上下文的信息，因此并不能完成“复杂”类型转换工作。GenericConverter 接口会根据源类对象及目标类对象所在宿主类的上下文信息进行类型转换工作，其接口定义如下：

```
package org.springframework.core.convert.converter;
public interface GenericConverter {
   public Set<ConvertiblePair> getConvertibleTypes();
   Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

ConvertiblePair 封装了源类型和目标类型，组成一个“对子”，而 TypeDescriptor 包含了需转换类型对象所在宿主类的信息，因此 GenericConverter 的 convert()接口方法可以利用这些上下文信息进行类型转换工作。

ConditionalGenericConverter 扩展于 GenericConverter 接口，并添加了一个接口方法，如下：

```
boolean  matches(TypeDescriptor sourceType, TypeDescriptor targetType)
```

该接口方法根据源类型及目标类型所在宿主类的上下文信息决定是否要进行类型转换，只有该接口方法返回 true 时，才调用 convert()方法完成类型转换。正是因为在转换之前有一个是否要进行类型转换的条件判断动作，因此该接口命名为 ConditionalGenericConverter，即带“条件”的通用转换器。

ConversionServiceFactoryBean 的 converters 属性可接受 Converter、ConverterFactory、GenericConverter 或 ConditionalGenericConverter 接口的实现类，并把这些转换器的转换逻辑统一封装到一个 ConversionService 实例对象中（GenericConversionService）。Spring 在 Bean 属性配置及 Spring MVC 请求消息绑定时将利用这个 ConversionService 实例完成类型转换工作。以上过程可通过图 17-9 来理解。

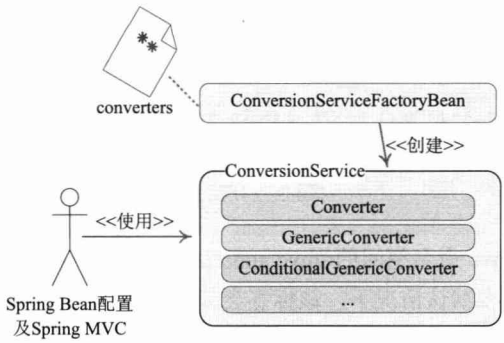


图 17-9　ConversionService 创建过程示意图

3．在 Spring MVC 中使用 ConversionService

假设处理方法有一个 User 类型的入参，我们希望将一个格式化的请求参数字符串直接转换为 User 对象，该字符串的格式为：

```
<userName>:<password>:<realName>
```

这就要求我们定义一个负责将格式化的 String 转换为 User 对象的自定义转换器，如代码清单 17-30 所示。

代码清单 17-30　StringToUserConverter.java：User对象转换器

```
package com.smart.domain;
import org.springframework.core.convert.converter.Converter;
public class StringToUserConverter implements Converter<String,User>{

    public User convert(String source) {
        User user = new User();
        if(source != null){
            String[] items = source.split(":");
            user.setUserName(items[0]);
            user.setPassword(items[1]);
            user.setRealName(items[2]);
```

将<userName>:<password>:<realName>的格式化串转换为 User 对象

①

```
        }
        return user;
    }
}
```

编写好 StringToUserConverter 后，接下来将其安装到 Spring 上下文中，如代码清单 17-31 所示。

代码清单 17-31　smart-servlet.xml：将StringToUserConverter安装到Spring上下文中

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="...                                    装配自定义的
      http://www.springframework.org/schema/mvc                ConversionService
      http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd">
    ...
    <mvc:annotation-driven conversion-service="conversionService"/> ①
    <bean id="conversionService"
        class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
          <list>
            <bean class="com.smart.domain.StringToUserConverter"/>②
          </list>
        </property>
    </bean>                          装配 StringToUserConverter
</beans>
```

在①处使用了 mvc 命名空间的<mvc:annotation-driven/>标签，该标签可简化 Spring MVC 的相关配置。在默认情况下，该标签会创建并注册一个默认的 DefaultAnnotation HandlerMapping 和一个 RequestMappingHandlerAdapter 实例。如果上下文中存在自定义的对应组件 Bean，则 Spring MVC 会自动利用自定义的 Bean 覆盖默认的 Bean。

除此之外，<mvc:annotation-driven/>标签还会注册一个默认的 ConversionService，即 FormattingConversionServiceFactoryBean（参见 17.3.3 节），以满足大多数类型转换的需求。现在由于要注册一个自定义的 StringToUserConverter，因此，需要显式定义一个 ConversionService 覆盖<mvc:annotation-driven/>中的默认实现，这是通过设置<mvc:annotation-driven/>的 conversion-service 属性来完成的。

在装配好 StringToUserConverter 后，就可以在任何控制器的处理方法中使用这个转换器了。在 UserController 中添加一个使用 StringToUserConverter 的 handle81()方法，如代码清单 17-32 所示。

代码清单 17-32　UserController.java：使用StringToUserConverter

```
@RequestMapping(path = "/handle81")
public String  handle81(@RequestParam("user") User user,ModelMap modelMap){
    modelMap.put("user", user);
    return "/user/showUser";
}
```

启动 Web 容器，发送一个如下 URL 请求：

```
http://localhost:8080/chapter17/user/handle81.html?user=tom:1234:tomson
```

user 请求参数的 tom:1234:tomson 值将会被 StringToUserConverter 正式地转换并绑定到 handle81()方法的 User 入参中。

4．使用@InitBinder 和 WebBindingInitializer 装配自定义编辑器

Spring MVC 在支持新的转换器框架的同时，也支持 JavaBeans 的 PropertyEditor。可以在控制器中使用@InitBinder 添加自定义的编辑器，也可以通过 WebBindingInitializer 装配在全局范围内使用的编辑器，如代码清单 17-33 所示。

代码清单 17-33　UserController.java：使用@InitBinder

```
package com.smart.web;
import org.springframework.web.bind.annotation.InitBinder;
import com.smart.domain.UserEditor;

@Controller
@RequestMapping("/user")
public class UserController {
    @InitBinder ①          在控制器初始化时调用
    public void initBinder(WebDataBinder binder) {
        binder.registerCustomEditor(User.class, new UserEditor());②
    }                                          注册一个自定义的编辑器
}
```

另外，Spring 4.0 可以使用 addCustomFormatter 指定格式化程序实现，这样就不需要实现一个 PropertyEditor 的实例，如下：

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
}
```

Spring MVC 使用 WebDataBinder 处理请求消息和处理方法入参的绑定工作。在②处通过 registerCustomEditor()方法为 User 注册一个自定义的编辑器，UserEditor 是实现 PropertyEditor 接口的编辑器。

如果希望在全局范围内使用 UserEditor 编辑器，则可实现 WebBindingInitializer 接口并在该实现类中注册 UserEditor，如代码清单 17-34 所示。

代码清单 17-34　MyBindingInitializer.java：注册全局的自定义编辑器

```
package com.smart.web;

import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

import com.smart.domain.User;
import com.smart.domain.UserEditor;

public class MyBindingInitializer implements WebBindingInitializer{
  public void initBinder(WebDataBinder binder, WebRequest request) {
      binder.registerCustomEditor(User.class, new UserEditor()); ①
  }
}
```

上述代码在 initBinder()接口方法中注册了 UserEditor 编辑器。接下来还需要在 Spring 上下文中通过 RequestMappingHandlerAdapter 装配 MyBindingInitializer，如代码清单 17-35 所示。

代码清单 17-35　smart-servlet.xml：将MyBindingInitializer安装到Spring上下文中

```
...
<bean class="org.springframework.web.servlet.mvc
                          .method.annotation.RequestMappingHandlerAdapter">
    <property name="webBindingInitializer">
      <bean class="com.smart.web.MyBindingInitializer" />
    </property>
</bean>
```

对于同一个类型对象来说，如果既在 ConversionService 中装配了自定义转换器，又通过 WebBindingInitializer 装配了自定义编辑器，同时还在控制器中通过@InitBinder 装配了自定义编辑器，那么 Spring MVC 将按以下优先顺序查找对应类型的编辑器：

- 查询通过@InitBinder 装配的自定义编辑器。
- 查询通过 ConversionService 装配的自定义转换器。
- 查询通过 WebBindingInitializer 装配的自定义编辑器。

17.3.3　数据格式化

Spring 使用转换器进行源类型对象到目标类型对象的转换，Spring 的转换器并不提供输入及输出信息格式化的工作。如果需要转换的源类型数据（一般是字符串）是从客户端界面中传递过来的，为了方便使用者观看，这些数据往往具有一定的格式。举例来说，像日期、时间、数字、货币等数据都是具有一定格式的，在不同的本地化环境中，同一类型的数据还会相应地呈现不同的显示格式。

如何从格式化的数据中获取真正的数据以完成数据绑定，并将处理完成的数据输出为格式化的数据，是 Spring 格式化框架要解决的问题。Spring 引入了一个新的格式化框架，这个框架位于 org.springframework.format 类包中。首先来了解一下格式化框架最重要的 Formatter<T>接口。

1. Formatter<T>

Formatter<T>接口扩展于 Printer<T>和 Parser<T>接口。

```
package org.springframework.format;
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

Printer<T>负责对象的格式化输出，而 Parser<T>负责对象的格式化输入，在接口中各定义了一个方法。先来看一下 Printer<T>的接口方法。

```
String print(T fieldValue, Locale locale);
```

print()接口方法将类型为 T 的成员对象根据本地化的不同输出为不同的格式化的字符串。

Parser<T>的接口方法也很简单。

```
T parse(String clientValue, Locale locale) throws ParseException;
```

parse()接口方法参考本地化信息将一个格式化的字符串转换为 T 类型的对象，即完成格式化对象的输入工作。

Spring 的 org.springframework.format.datetime 包中提供了一个用于时间对象格式化的 DateFormatter 实现类；而 org.springframework.format.number 包中提供了 3 个用于数字对象格式化的实现类。

- NumberFormatter：用于数字类型对象的格式化。
- CurrencyFormatter：用于货币类型对象的格式化。
- PercentFormatter：用于百分数数字类型对象的格式化。

可以手工调用这些 Formatter 接口实现类进行对象数据输入/输出的格式化工作，但是随着 Java 技术的发展，这种硬编码的格式化方式显然已经过时。Spring 提供了注解驱动的属性对象格式化功能，可在 Bean 属性设置、Spring MVC 处理方法入参数据绑定、模型数据输出时自动通过注解应用格式化功能。

2．注解驱动格式化的重要接口

为了让注解和格式化的属性类型关联起来，Spring 在 Formatter<T>所在的包中还提供了一个 AnnotationFormatterFactory<A extends Annotation>接口。来看一下该接口的几个方法。

- Set<Class<?>> getFieldTypes()：注解 A 的应用范围，即哪些属性类可以标注 A 注解。
- Parser<?> getParser（A annotation, Class<?> fieldType）：根据注解 A 获取特定属性类型的 Parser。
- Printer<?> getPrinter（A annotation, Class<?> fieldType）：根据注解 A 获取特定属性类型的 Printer。

Spring 提供了两个内建的实现类，分别支持数字及日期类型的注解驱动格式化。

- NumberFormatAnnotationFormatterFactory：支持对数字类型的属性使用@NumberFormat 注解（位于 org.springframework.format.annotation 包中）。
- JodaDateTimeFormatAnnotationFormatterFactory：支持对日期类型的属性使用@DateTimeFormat 注解，该注解和@NumberFormat 注解位于相同的包中。

现在剩下的问题是：如何在 Spring 上下文中启用这个基于注解驱动的格式化模块，并在实际应用中使用格式化功能？

3．启用注解驱动格式化功能

聪明的读者已经发现，对属性对象的输入/输出进行格式化，从本质上讲依然属于“类型转换”的范畴。不错！Spring 就是基于对象转换框架植入“格式化”功能的。

Spring 在格式化模块中定义了一个实现 ConversionService 接口的 FormattingConversionService 实现类，该实现类扩展了 GenericConversionService，因此它既具有类型转换功能，

又具有格式化功能。

相对于 ConversionService 的 ConversionServiceFactoryBean 工厂类，FormattingConversionService 也拥有一个对应的 FormattingConversionServiceFactoryBean 工厂类，后者用于在 Spring 上下文中构造一个 FormattingConversionService。通过这个工厂类，既可以注册自定义的转换器，还可以注册自定义的注解驱动逻辑。

由于 FormattingConversionServiceFactoryBean 在内部会自动注册 NumberFormatAnnotationFormatterFactory 和 JodaDateTimeFormatAnnotationFormatterFactory，因此，装配了 FormattingConversionServiceFactoryBean 后，就可以在 Spring MVC 入参绑定及模型数据输出时使用注解驱动的格式化功能。

首先在 Spring 上下文中装配 FormattingConversionServiceFactoryBean，如代码清单 17-36 所示。

代码清单 17-36　smart-servlet.xml：装配FormattingConversionServiceFactoryBean

```
...
<mvc:annotation-driven conversion-service="conversionService"/>
<bean id="conversionService"
   class="org.springframework.format.support.FormattingConversionServiceFactory
Bean">①
    <property name="converters">
      <list>
        <bean class="com.smart.domain.StringToUserConverter"/>
      </list>
    </property>
</bean>
```

替换原来的 ConversionServiceFactoryBean

值得注意的是，<mvc:annotation-driven/>标签内部默认创建的 ConversionService 实例就是一个 FormattingConversionServiceFactoryBean。

装配好 FormattingConversionServiceFactoryBean 后，Spring MVC 对处理方法的入参绑定就支持注解驱动功能了。

4．注解驱动格式化实例

在 User 中添加两个新属性，并标注格式化注解，如代码清单 17-37 所示。

代码清单 17-37　User.java：使用注解驱动格式化功能

```
package com.smart.domain;

import java.util.Date;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.NumberFormat;
...

public class User {
    ...
    @DateTimeFormat(pattern="yyyy-MM-dd")①
    private Date birthday;

    @NumberFormat(pattern="#,###.##") ②
    private long salary;
```

可将形如 1980-09-01 的字符串转换到 Date 类型的 birthday 属性中

可将形如 4,500.00 的字符串转换到 long 类型的 salary 属性中

```
    //省略get/setter方法
}
```

birthday 属性标注了@DateTimeFormat，并指定其格式为“yyyy-MM-dd”；而 salary 属性标注了@NumberFormat，并指定其格式为“#,###.##”。

@DateTimeFormat 注解可对 java.util.Date、java.util.Calendar、java.long.Long 及 Joda 时间类型的属性进行标注。因 Spring 4.0 已全面支持 Java SE 8.0，所以也可以使用 JDK 8.0 的 java.time 包。它支持以下几个互斥的属性。

- iso：类型为 DateTimeFormat.ISO。以下是几个常用的可选值。
  - DateTimeFormat.ISO.DATE：格式为 yyyy-MM-dd。
  - DateTimeFormat.ISO.DATE_TIME：格式为 yyyy-MM-dd hh:mm:ss.SSSZ。
  - DateTimeFormat.ISO.TIME：格式为 hh:mm:ss.SSSZ。
  - DateTimeFormat.ISO.NONE：表示不应该使用 ISO 格式的时间。
- pattern：类型为 String，使用自定义的时间格式化串，如 yyyy/mm/dd h:mm:ss。
- style：类型为 String，通过样式指定日期/时间的格式，由两位字符组成，第一位表示日期的样式，第二位表示时间的样式。以下是几个常用的可选值。
  - S：短日期/时间的样式。
  - M：中日期/时间的样式。
  - L：长日期/时间的样式。
  - F：完整日期/时间的样式。
  - -：忽略日期或时间的样式。

而@NumberFormat 可对类似于数字类型的属性进行标注，它拥有两个互斥的属性，具体说明如下。

- pattern：类型为 String，使用自定义的数字格式化串，如“##,###.##”。
- style：类型为 NumberFormat.Style。以下是几个常用的可选值。
  - NumberFormat.Style.CURRENCY：货币类型。
  - NumberFormat.Style.NUMBER：正常数字类型。
  - NumberFormat.Style.PERCENT：百分数类型。

我们在 UserController 中新增了一个 handle82(User user)处理方法，在 UserControllerTest 中通过如下方法进行测试：

```
@Test
public void testHandle82() {
    RestTemplate restTemplate = new RestTemplate();
    MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
    form.add("userName", "tom");
    form.add("password", "123456");
    form.add("age", "45");
    form.add("birthday", "1980-01-01");
    form.add("salary", "4,500.00");          ①
    String html = restTemplate.postForObject(
```

①← 请求参数的值都是格式化的字符串

```
            "http://localhost:8080/chapter17/user/handle82.html",form,String.class);
    Assert.assertNotNull(html);
    Assert.assertTrue(html.indexOf("tom")>-1);
}
```

如果希望在视图页面中将模型属性数据也以格式化的方式进行渲染，则可通过 Spring 的页面标签显示模型数据来达到目的。

17.3.4 数据校验

应用程序在执行业务逻辑前，必须通过数据校验保证接收到的输入数据是正确合法的，如代表生日的日期应该是一个过去的时间、工资的数值必须是一个正数等。一般情况下，应用程序的开发是分层的，不同层的代码由不同的开发人员负责。很多时候，同样的数据验证会出现在不同的层中，这样就会导致代码冗余，违反了 DRY 原则。为了避免这样的情况，最好将验证逻辑和相应的域模型进行绑定，将代码验证的逻辑集中起来管理。

1. JSR-303

JSR-303 是 Java 为 Bean 数据合法性校验所提供的标准框架，它已经包含在 Java EE 6.0 中。JSR-303 通过在 Bean 属性上标注类似于@NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证。可以通过 http://jcp.org/en/jsr/detail?id=303 了解 JSR-303 的详细内容。

JSR-303 定义了一套可标注在成员变量、属性方法上的校验注解，说明如表 17-3 所示。

表 17-3 JSR-303 注解

注　解	功能说明
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期

Hibernate Validator 是 JSR-303 的一个参考实现，它除了支持所有标准的校验注解外，还支持如表 17-4 所示的扩展注解。

表 17-4　Hibernate Validator扩展注解

注　解	功能说明
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串必须非空
@Range	被注释的元素必须在合适的范围内

JSR-303 的核心接口是 javax.validation.Validator，该接口根据目标对象类中所标注的校验注解进行数据校验，并得到校验结果。

2．Spring 校验框架

Spring 拥有自己独立的数据校验框架，同时支持 JSR-303 标准的校验框架。Spring 的 DataBinder 在进行数据绑定时，可同时调用校验框架完成数据校验工作。在 Spring MVC 中，则可直接通过注解驱动的方式进行数据校验。

Spring 的 org.springframework.validation 是校验框架所在的包，下面来了解一下校验框架的几个重要接口和类。

首先要了解的是 Validator，该接口拥有以下两个方法。

- boolean supports（Class<?> clazz）：该校验器能够对 clazz 类型的对象进行校验。
- void validate（Object target, Errors errors）：对目标类 target 进行校验，并将校验错误记录在 errors 中。

LocalValidatorFactoryBean 既实现了 Spring 的 Validator 接口，又实现了 JSR-303 的 Validator 接口。只要在 Spring 容器中定义了一个 LocalValidatorFactoryBean，即可将其注入需要数据校验的 Bean 中。定义 LocalValidatorFactoryBean 的 Bean 非常简单，代码如下：

```
<bean id="validator"
  class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

值得注意的是，Spring 本身没有提供 JSR-303 的实现，所以必须将 JSR-303 的实现者（如 Hibernate Validator）的 JAR 文件放到类路径下，Spring 将自动加载并装配好 JSR-303 的实现者。

3．Spring MVC 数据校验

<mvc:annotation-driven/>会默认装配一个 LocalValidatorFactoryBean，通过在处理方法的入参上标注@Valid 注解，即可让 Spring MVC 在完成数据绑定后执行数据校验工作。

下面对代码清单 17-37 进行更改，添加 JSR-303 的校验注解，如代码清单 17-38 所示。

代码清单 17-38　User.java：添加校验注解

```
package com.smart.domain;

...
```

```
import javax.validation.constraints.*;

import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.NumberFormat;

public class User {
    private String userId;

    @Pattern(regexp="w{4,30}") ①
    private String userName;

    @Pattern(regexp="S{6,30}")②
    private String password;

    @Length(min=2,max=100) ③
    private String realName;

    @Past ④
    @DateTimeFormat(pattern="yyyy-MM-dd")
    private Date birthday;

    @DecimalMin(value="1000.00")
    @DecimalMax(value="100000.00") ⑤
    @NumberFormat(pattern="#,###.##")
    private long salary;
}
```

通过正则表达式进行校验，匹配 4～30 个包含数字、字母及下画线的字符

通过正则表达式进行校验，匹配 6～30 个非空白的字符

这是 Hibernate Validator 的扩展度，将属性值的长度限制在 2～100 之间

时间值必须是一个过去的时间

数据必须在 1000.00～100000.00 之间

在 User 类的属性上标注校验注解后，接下来的问题是如何在 Spring MVC 中使用注解所声明的限制规则进行数据校验。我们在 UserController 处理器中添加一个处理方法进行演示，如代码清单 17-39 所示。

代码清单 17-39　UserController.java：对处理方法的入参数据进行校验

```
package com.smart.web;
import javax.validation.Valid;
import org.springframework.validation.BindingResult;
…
@Controller
@RequestMapping("/user")
public class UserController {
    …
    @RequestMapping(path = "/handle91")
    public String handle91(@Valid @ModelAttribute("user")User user, ①
                           BindingResult bindingResult){
        if(bindingResult.hasErrors()){②
            return "/user/register3";
        }else{
            return "/user/showUser";
        }
    }
}
```

在入参对象前添加@Valid 注解，同时在其后声明一个 BindingResult 入参

根据 BindingResult 即可判断是否存在错误，如果有错误则转到注册页面

在已经标注了 JSR-303 注解的表单/命令对象前标注一个@Valid，Spring MVC 框架在将请求数据绑定到该入参对象后，就会调用校验框架根据注解声明的校验规则实施校

验。这里的关键问题是：校验所产生的校验结果保存在什么地方？又如何传递给请求处理方法？Spring MVC 是通过对处理方法签名的规约来保存校验结果的：前一个表单/命令对象的校验结果保存在其后的入参中，这个保存校验结果的入参必须为 BindingResult 或 Errors 类型，这两个类都位于 org.springframework.validation 包中。其签名约定可通过图 17-10 来说明。

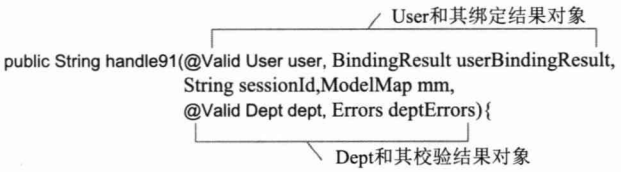


图 17-10　数据校验时处理方法入参的签名规约

概括来说，需校验的表单/命令对象和其绑定结果对象或错误对象是成对出现的，它们之间不允许声明其他入参。

Errors 接口提供了获取错误信息的方法，如 getErrorCount()方法获取错误的数量，getFieldErrors(String field)方法得到成员属性的校验错误列表。而 BindingResult 接口扩展了 Errors 接口，以便可以使用 Spring 的 org.springframework.validation.Validator 对数据进行校验，同时获取数据绑定结果对象的信息。BindingResult 接口通过 DataBinder.getBindingResult()方法获取。

4．如何获取校验结果

在表单/命令对象类的属性中标注校验注解，在处理方法对应的入参前添加@Valid，Spring MVC 就会实施校验并将校验结果保存在被校验入参对象之后的 BindingResult 或 Errors 入参中。

在处理方法内部可以通过 BindingResult 或 Errors 入参对象获取错误信息，如代码清单 17-36 中的②处所示，这样就可以通过 bindingResult.hasErrors()方法判断 User 入参对象是否存在校验错误。还可以通过接口方法获取更多的信息，以下是几个常用的方法。

- FieldError getFieldError(String field)：根据属性名获取对应的校验错误。
- List<FieldError> getFieldErrors()：获取所有的属性校验错误。
- Object getFieldValue(String field)：获取属性值。
- int getErrorCount()：获取错误数量。

5．如何在页面中显示错误

由于表单/命令对象所对应的请求一般是从客户端的网页中传送过来的，因此，如果发生了错误，则必须通过网页显示错误，提示用户更正错误。

Spring MVC 除了将表单/命令对象的校验结果保存到对应的 BindingResult 或 Errros 对象中外，还将所有的校验结果保存到“隐含模型”中（参见 17.2.5 节）。也就是说，即使处理方法的签名中没有对应表单/命令对象的校验结果入参，校验结果也不会丢失，它们始终可以从“隐含模型”对象中获取。校验结果对象和被校验的表单/命令对象是一

对一的关系，但“隐含模型”除了存储模型数据外，还保存了所有被校验的表单/命令对象的校验结果。隐含模型中的所有数据最终将通过 HttpServletRequest 的属性列表暴露给 JSP 视图对象，因此我们可以很容易地在 JSP 中获取校验错误信息。

抛开 Spring MVC 的所有组件，单从输入/输出的层面分析 Spring MVC 框架的数据流，如图 17-11 所示。

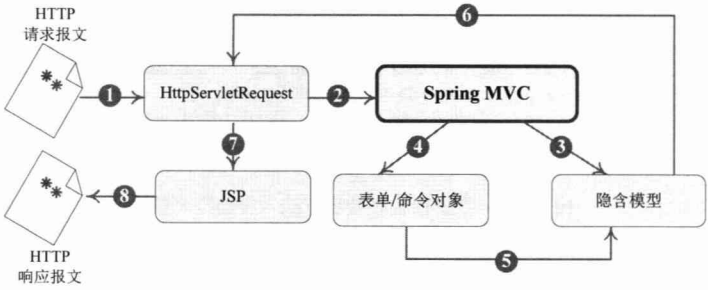


图 17-11　Spring MVC 的数据流图

① HTTP 请求报文到达 Web 服务器，Web 服务器将其封装成一个 HttpServletRequest 对象。

② Spring MVC 框架截获这个 HttpServletRequest 对象。

③ Spring MVC 创建一个隐含模型对象，作为处理本次请求的上下文数据存放处。

④ Spring MVC 将 HttpServletRequest 对象数据绑定到处理方法的入参对象中（表单/命令对象）。

⑤ 将绑定错误信息、检验错误信息都保存到隐含模型中。

⑥ 将本次请求的对应隐含模型数据存放到 HttpServletRequest 属性列表中，暴露给视图对象。

⑦ 视图对象对已经存放在 HttpServletRequest 属性列表中的模型数据进行渲染。

⑧ 将渲染后的 HTTP 响应报文发送给客户端。

理解了 Spring MVC 的整体处理过程后，可以肯定的是，校验错误信息一定包含在隐含模型中，且会通过 HttpServletRequest 暴露给视图对象。

可通过 Spring 的<form:errors path="propName"/>标签在 JSP 页面中显示错误信息，如代码清单 17-40 所示。

代码清单 17-40　register3.jsp：显示错误信息

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c"      uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form"   uri="http://www.springframework.org/tags/form" %>①
<html>
<head>                                                    引入 Spring 表单标签
<title>注册用户</title>
  <style>.errorClass{color:red}</style>
</head>
```

```
<body>
  <form:form modelAttribute="user"  action="/chapter17/user/handle91.html">
    <form:errors path="*"/>②
    <table>
        <tr>
          <td>用户名：</td>
          <td>
            <form:errors path="userName" cssClass="errorClass"/>③
            <form:input path="userName" />
          </td>
        </tr>
        <tr>
         <td>密码：</td>
          <td>
            <form:errors path="password" cssClass="errorClass" />④
            <form:password path="password" />
          </td>
        </tr>
         …
        <tr>
         <td colspan="2"><input type="submit" name="提交"/></td>
        </tr>
    </table>
  </form:form>
</body>
</html>
```

显示表单对象所有的错误信息

显示 userName 属性的错误信息

显示 password 属性的错误信息

在②处，当检测到存在错误信息时，转到 register3.jsp 视图页面。register3.jsp 在①处引入了 Spring 的标签库，可以通过如②处所示的方式显示所有的错误信息，也可以通过如③或④处所示的方式显示特定属性的错误信息。显示错误信息的页面如图 17-12 所示。

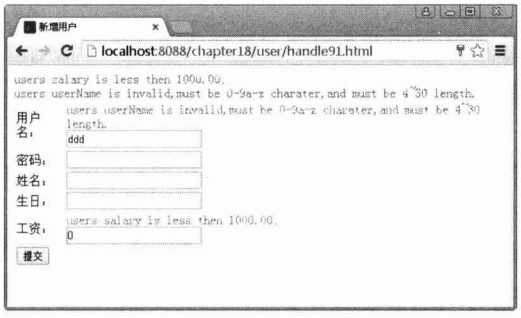


图 17-12　显示错误信息的页面

从图 17-12 中可以看出，Spring MVC 已经对表单的数据进行了合法性校验，并通过<form:errors path="*" />标签在页面顶部显示所有的校验错误信息，而通过<form:errors path="propName" />标签可显示某个属性的校验错误信息。

通过观察页面上的错误信息，可以发现这些错误信息都比较生硬和机械。实际上，这些校验错误信息都是校验框架根据校验规则自动产生的。在实际应用中，不但希望错误信息更加人性化、更具可读性，还希望可显示本地化的错误信息。Spring MVC 支持对错误信息进行本地化显示。

6. 通过国际化资源显示错误信息

每个属性在数据绑定和数据校验发生错误时，都会生成一个对应的 FieldError 对象。FieldError 对象实现了 org.springframework.context.MessageSourceResolvable 接口，MessageSourceResolvable 顾名思义是可用国际化资源进行解析的对象。MessageSource Resolvable 拥有 3 个接口方法。

- Object[] getArguments()：返回一组参数对象。
- String[] getCodes()：返回一组消息代码，每个代码对应一个资源属性。可以使用 getArguments()方法返回的参数对资源属性值进行参数替换。
- String getDefaultMessage()：默认的消息。由于我们没有装配相应的国际化资源，因此图 17-12 中显示的错误信息都是默认的。

当一个属性校验失败后，校验框架会为该属性生成 4 个消息代码，这些消息代码以校验注解类名为前缀，结合类名、属性名及属性类型名生成多个对应的消息代码。

如 User 类的 password 属性标注了一个@Pattern 注解，当该属性值不满足@Pattern 所定义的限制规则时，就会产生以下 4 个错误代码。

- Pattern.user.password：根据类名、属性名产生的错误代码。
- Pattern.password：根据属性名产生的错误代码。
- Pattern.java.lang.String：根据属性类型产生的错误代码。
- Pattern：根据验证注解名产生的错误代码。

当使用 Spring MVC 标签显示错误信息时，Spring MVC 会查看 Web 上下文是否装配了对应的国际化消息。如果没有，则显示默认的错误信息；否则使用国际化消息对错误代码进行翻译。

值得注意的是，如果在数据类型转换或数据格式转换时发生错误，或者该有的参数不存在，或调用处理方法时发生错误，则都会在隐含模型中创建错误信息。其错误代码前缀说明如下。

- required：必要的参数不存在，如@RequestParam("param1")标注了一个入参，但是请求参数不存在 param1 参数。
- typeMismatch：在数据绑定时，发生数据类型不匹配的问题。
- methodInvocation：Spring MVC 在调用处理方法时发生了错误。

举例来说，如果将“aaa”的非数值参数传递给 User 对象的 salary 属性，则将发生数据转换错误，Spring 将为该错误生成以下错误代码：

- typeMismatch.user.salary。
- typeMismatch.salary。
- typeMismatch.long。
- typeMismatch。

错误代码的产生逻辑可以通过 DataBinder 的 setMessageCodesResolver()方法注入一个自定义的 MessageCodesResolver 对象来调整，不过一般情况下采用默认的产生器就可以了。

在知道错误对象的错误代码是对应国际化消息的键名后，接下来的工作就非常简单了：定义一个国际化资源，在国际化资源文件中为错误代码定义相应的本地化消息。

我们在 i18n/下添加基名为 messages 的国际化资源，一个是默认的 messages.properties，另一个是对应中国大陆的 messages_zh_CN.properties。来看一下 messages_zh_CN.properties 资源文件的内容，如图 17-13 所示。

name	value
Pattern.user.userName	用户名不正确，必须是4~30个英数及_的字符
Pattern.user.password	密码不正确，必须是6~30个字符，不允许空格
Length.user.realName	姓名不合法，长度必须是2~100个字符
Past.user.birthday	日期格式不正确，必须是一个过去的日期
DecimalMin.user.salary	工资必须在1000~100000之间
DecimalMax.user.salary	${DecimalMin.user.salary}

图 17-13　messages_zh_CN.properties（使用 MyEclipse Properties Editor 查看）

接下来的工作是在 smart-servlet.xml 中装配好这个国际化资源。

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource"
    p:basename="i18n/messages"/>
```

可以通过 ResourceBundleMessageSource 的 basename 属性指定一个国际化资源，还可以通过 basenames 属性指定多个国际化资源。重新启动 Web 容器，当发生错误时，register3.jsp 所显示的错误页面如图 17-14 所示。

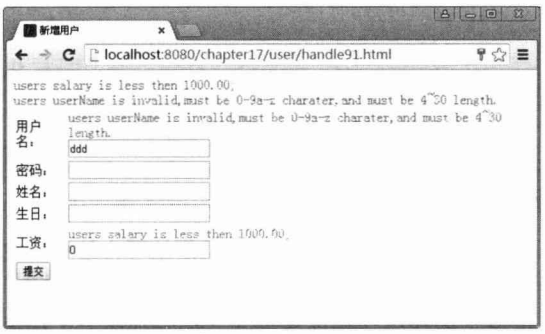


图 17-14　使用国际化资源显示错误信息

7．自定义校验规则

JSR-303 允许定义自定义校验注解及对应的校验器类，Spring 的校验框架完全支持 JSR-303 的自定义扩展注解。关于 JSR-303 自定义校验注解的内容已经超出本书的范畴，感兴趣的读者可以阅读相关材料。

这里要介绍的是如何给请求处理类装配一个自定义的 Validator，或者直接在处理方法中使用自定义的 Validator 对入参进行校验。可以在 UserController 中标注@InitBinder 注解的 initBinder()方法中装配自定义的 MyValidator。

```
@InitBinder
public void initBinder(WebDataBinder binder) {
  …
  binder.setValidator(new UserValidator());//在进行数据绑定时使用的校验器
}
```

当然，还可以借助请求处理方法的签名传递一个 Errors 或 BindingResult 对象进来，然后在处理方法中直接校验，如代码清单 17-41 所示。

代码清单 17-41　UserController.java：在处理方法中通过代码进行校验

```
package com.smart.web;

import org.springframework.validation.ValidationUtils;
import org.springframework.validation.BindingResult;


@Controller
@RequestMapping("/user")
public class UserController {


    @RequestMapping(path = "/handle92")
    public String  handle92(@ModelAttribute("user")User user,
                      BindingResult bindingResult){ ①
     ValidationUtils.rejectIfEmptyOrWhitespace(bindingResult, "userName",
"required");②
        if("aaaa".equalsIgnoreCase(user.getUserName())){
            bindingResult.rejectValue("userName", "reserved");③
        }
        if(bindingResult.hasErrors()){
            return "/user/register4";
        }else{
            return "/user/showUser";
        }
    }
}
```

声明一个 BindingResult 入参

使用校验器工具类进行校验

手工生成一条错误信息

在①处，处理方法签名声明了一个 BindingResult 入参，Spring MVC 框架将把对应本处理方法的 BindingResult 对象传递给该入参，以便处理方法借由这个入参访问绑定结果对象。

ValidationUtils 提供了空值验证的便捷方法，如②处所示，如果对应属性值为空或空白字符，则将生成一条错误信息。而在③处，通过手工判断 userName 是否等于“aaaa”，如果等于“aaaa”，则将产生一条错误信息。

在②处产生的错误信息对应的错误代码包括：

- required.user.userName。
- required.userName。
- required.java.lang.String。
- required。

相应的，在③处产生的错误信息对应的错误代码包括：

- reserved.user.userName。
- reserved.userName。

- reserved.java.lang.String。
- reserved。

可以从错误代码组中任选一个错误代码，并在国际化资源文件中定义错误代码对应的信息内容。这样，当发生错误时，就会显示相应的本地化错误信息。

提示

通过 binder.setValidator()方法设置自定义的 Validator 后，Spring MVC 将使用它对入参对象进行校验，而不再使用 Spring MVC 框架装配的 Validator 对入参对象进行校验。换句话说，即使在入参上标注了@Valid 注解，也不会再根据入参对象类中的 JSR-303 注解进行校验。

17.4　视图和视图解析器

请求处理方法执行完成后，最终返回一个 ModelAndView 对象。对于那些返回 String、View 或 ModelMap 等类型的处理方法，Spring MVC 也会在内部将它们装配成一个 ModelAndView 对象，该对象包含了视图逻辑名和模型对象的信息。

Spring MVC 借助视图解析器（ViewResolver）得到最终的视图对象（View），这可能是我们常见的 JSP 视图，也可能是一个基于 FreeMarker、Velocity 模板技术的视图，还可能是 PDF、Excel、XML、JSON 等各种形式的视图。

对于最终究竟采取何种视图对象对模型数据进行渲染，处理器并不关心，处理器的工作重点聚焦在生产模型数据的工作上，从而实现 MVC 的充分解耦。

17.4.1　认识视图

视图的作用是渲染模型数据，将模型里的数据以某种形式呈现给客户。视图对象可以是常见的 JSP，还可以是 Excel 或 PDF 等形式不一的媒体形式。为了实现视图模型和具体实现技术的解耦，Spring 在 org.springframework.web.servlet 包中定义了一个高度抽象的 View 接口，该接口中定义了两个方法。

- String getContentType()：视图对应的 MIME 类型，如 text/html、image/jpeg 等。
- void render(Map model, HttpServletRequest request, HttpServletResponse response)：将模型数据以某种 MIME 类型渲染出来。

视图对象是一个 Bean，通常情况下，视图对象由视图解析器负责实例化。由于视图 Bean 是无状态的，所以它们不会有线程安全的问题。

不同类型的视图实现技术对应不同的 View 实现类，这些视图实现类都位于 org.springframework.web.servlet.view 包中，通过表 17-5 来说明。

表 17-5 不同的视图实现类

大　类	视图类型	说　明
URL 资源视图	InternalResourceView	将 JSP 或其他资源封装成一个视图，这是 InternalResourceViewResolver 默认使用的视图实现类
	JstlView	如果 JSP 文件中使用了 JSTL 国际化标签的功能，则需要使用该视图类，而非 InternalResourceView 视图类
XSTL 视图	XsltView	XSTL 驱动的视图
Tiles 视图	TilesView	基于 Tiles 页面布局的视图
	TilesJstlView	如果 Tiles 模板的 JSP 组成文件使用了 JSTL，则需要使用该视图替换 TilesView 视图
文档视图	AbstractExcelView	Excel 文档视图的抽象类，开发者可以通过该抽象类实现自己的 Excel 文档视图。该视图实现类基于 POI 构造 Excel 文档
	AbstractJExcelView	和 AbstractExcelView 类似，只不过它基于 JExcelAPI
	AbstractPdfStamperView	PDF 文档视图的抽象类，通过 AcroForm 技术对 PDF 文档进行操作
	AbstractPdfView	PDF 文档视图的抽象类，可以通过该抽象类实现自己的 PDF 文档视图。该视图实现类基于 iText 构造 PDF 文档
模板视图	FreeMarkerView	使用 FreeMarker 模板引擎的视图
	VelocityLayoutView VelocityToolboxView VelocityView	几个使用 Velocity 模板引擎的视图
报表视图	ConfigurableJasperReportsView JasperReportsCsvView JasperReportsHtmlView JasperReportsMultiFormatView JasperReportsPdfView JasperReportsXlsView	几个使用 JasperReports 报表技术的视图
XML 和 JSON 视图	MarshallingView	通过 oxm 的 Marshaller 技术将模型数据以 XML 方式输出
	MappingJackson2JsonView	将模型数据通过 Jackson 开源框架的 ObjectMapper 以 JSON 方式输出
其他视图	RedirectView	进行重定向的视图，可以重定向到上下文的绝对路径或相对路径下，也可以重定向到当前请求的相对路径下

17.4.2 认识视图解析器

Spring MVC 为逻辑视图名的解析提供了不同的策略，可以在 Spring Web 上下文中配置一种或多种解析策略，并指定它们之间的先后顺序。每种解析策略对应一个具体的视图解析器实现类。视图解析器的工作比较单一，即将逻辑视图名解析为一个具体的视图对象。所有视图解析器都实现了 ViewResolver 接口，该接口仅有一个方法。

```
View resolveViewName(String viewName, Locale locale)
```

resolveViewName()方法的签名清楚地向我们传达了视图解析器工作的内涵：根据逻

辑视图名和本地化对象得到一个视图对象。Spring 拥有众多的视图解析器实现类，通过表 17-6 进行概括性说明。

表 17-6　视图解析器实现类

大　类	视图解析器类型	说　明
解析为 Bean 名字	BeanNameViewResolver	将逻辑视图名解析为一个 Bean，Bean 的 id 等于逻辑视图名
	XmlViewResolver	和 BeanNameViewResolver 类似，只不过目标视图 Bean 对象定义在一个独立的 XML 文件中，而非定义在 DispatcherServlet 上下文的主配置文件中
国际化解析	ResourceBundleViewResolver	在国际化资源文件中定义视图实现类及相关的信息。使用该视图解析器可以为不同的本地化类型提供不同的解析结果
解析为 URL 文件	InternalResourceViewResolver	将视图名解析为一个 URL 文件，一般使用该解析器将视图名映射为保存在 WEB-INF 目录中的程序文件（如 JSP）
	XsltViewResolver	将视图名解析为一个指定 XSLT 样式表的 URL 文件
	JasperReportsViewResolver	JasperReports 是一个基于 Java 的开源报表工具，该解析器将视图名解析为报表文件所对应的 URL
模板文件视图	FreeMarkerViewResolver	解析为基于 FreeMarker 模板技术的模板文件
	VelocityViewResolver VelocityLayoutViewResolver	解析为基于 Velocity 模板技术的模板文件
内容协商	ContentNegotiatingViewResolver	它不负责具体的视图解析，而是作为一个中间人的角色根据请求所要求的 MIME 类型，从上下文中选择一个适合的视图解析器，再将视图解析工作委托其负责

用户可以选择一种视图解析器或混用多种视图解析器，每个视图解析器都实现了 Ordered 接口并开放出一个 orderNo 属性，可以通过该属性指定解析器的优先顺序，值越小优先级越高。有些视图解析器默认为最高优先级（如 ContentNegotiatingViewResolver），而有些视图解析器默认为最低优先级（如 InternalResourceViewResolver、XsltViewResolver 等），具体请参考 API 文档。

Spring MVC 会按照视图解析器的优先级顺序对逻辑视图名进行解析，直到解析成功并返回视图对象，否则将抛出 ServletException 异常。

17.4.3　JSP 和 JSTL

1．使用 InternalResourceViewResolver

JSP 是最常见的视图技术，在 17.1.3 节中就使用了 InternalResourceViewResolver 作为视图解析器，其配置如下：

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/views/"  p:suffix=".jsp"/>
```

代码清单 17-4 中的 createUser()处理方法返回名为 user/createSuccess 的逻辑视图名，InternalResourceViewResolver 负责对此进行解析，将得到如图 17-15 所示的解析结果。

图 17-15　逻辑视图名解析为 URI 资源

InternalResourceViewResolver 默认使用 InternalResourceView 作为视图实现类。如果 JSP 文件使用了 JSTL 的国际化功能，确切地说，当 JSP 页面使用 JSTL 的<fmt:message/>标签时，用户需要使用 JstlView 替换默认的视图实现类，如下：

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:viewClass="org.springframework.web.servlet.view.JstlView"
      p:prefix="/WEB-INF/views/" p:suffix=".jsp"/>
```

下面通过一个例子演示使用 JSTL 的<fmt:message/>标签让 JSP 页面实现本地化输出。首先在 UserController 中添加一个方法。

```
@RequestMapping(path = "/showUserList")
 public String  showUserList(ModelMap mm){
     Calendar calendar = new GregorianCalendar();
     List<User> userList = new ArrayList<User>();
     User user1 = new User();
     user1.setUserName("tom");
     user1.setRealName("汤姆");
     calendar.set(1980, 1, 1);
     user1.setBirthday(calendar.getTime());
     User user2 = new User();
     user2.setUserName("john");
     user2.setRealName("约翰");
     user2.setBirthday(calendar.getTime());
     userList.add(user1);
     userList.add(user2);
     mm.addAttribute("userList",userList);
     return "user/userList";
}
```

userList.jsp 的代码如代码清单 17-42 所示。

代码清单 17-42　userList.jsp：通过JSTL的<fmt:message>标签使用国际化资源

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
    <head>
        <title><fmt:message key="website.title"/></title>①
    </head>
    <body>
        <fmt:message  key="user.userList.title"/>
        <table>
            <c:forEach items="${userList}" var="user">
```

显示本地化的信息

```
                <tr>
                    <td>
                    <a href="<c:url value="/user/showUser/${user.userName}.
html"/>">                                                                    ②
                        ${user.userName}
                    </a>
                    </td>
                    <td>
                        ${user.realName}
                    </td>
                    <td>
                     <fmt:formatDate value="${topic.createDate}" pattern=
"yyyy-MM-dd"/>
                    </td>
                </tr>
            </c:forEach>
        <table>                                              使用 JSTL 标签
    </body>
</html>
```

userList.jsp 所使用的国际化资源在 content 资源文件中定义，更改 smart-servlet.xml 中 ResourceBundleMessageSource 的配置，添加此国际化资源。

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>i18n/messages</value>
        <value>i18n/content</value>
      </list>
    </property>
</bean>
```

这样，userList.jsp 就可以根据客户端的不同显示相应的本地化页面。除了可以使用 JSTL 的标签外，Spring 也提供了一个轻量级的标签库，可以在 JSP 文件中使用这些标签。

2. 使用 Spring 表单标签

通过 Spring 表单标签，可以很容易地将模型数据中的表单/命令对象绑定到 HTML 表单元素中。在本章前面的内容中，我们已经使用了一些 Spring 表单标签，本节将对 Spring 表单标签进行全面的介绍，首先从<form:form>标签开始。

和使用任何 JSP 扩展标签一样，在使用 Spring 表单标签之前，用户必须在 JSP 页面中添加一行引用 Spring 表单标签的声明。

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding= "UTF-8"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%> ①
<html>                                                          引用表单标签的声明
  … ②  声明后，在页面中就可以使用 Spring 表单标签了
</html>
```

下面是一个使用<form:form>表单标签的实例，它将最终生成一个 HTML 的 form 表单。

```
<form:form modelAttribute="user">
    用户名: <form:input path="userName" />   <br>
```

```
    密  码: <form:password path="password" /><br>
    Email:  <form:input path="email" /><br>
    <input type="submit" value="注册" name="testSubmit"/>
    <input type="reset" value="重置" />
</form:form>
```

一般情况下，通过 GET 请求获取表单页面，通过 POST 请求提交表单页面，因此，获取表单页面和提交表单页面的 URL 地址是相同的。只要满足这条最佳实践的契约，<form:form>标签就无须通过 action 属性指定表单提交的目标 URL。

可以通过 modelAttribute 指定绑定的模型属性。如果 modelAttribute 属性不指定，那么默认从模型中尝试获取名为 command 的表单对象。如果不存在此表单对象，则将发生错误。通过 action 指定处理表单提交的 URL 地址。此外，<form:form>标签还拥有多个可设置的属性，这些属性大都是 HTML 表单标签属性的镜像，如 id、onclick、ondblclick、tabindex 等。需要注意的是，这些属性都是小写的，而对应 HTML 标签的属性名则没有这个限制。

Spring 提供了十多个表单组件标签，如<form:input/>、<form:select/>等，用以绑定表单对象的属性值。这些标签大都拥有以下属性。

- path：用属性路径表示的表单对象属性，如 userName、dept.deptName、dept.address.telephone、userName 等。
- htmlEscape：绑定的表单属性值是否要对 HTML 特殊字符进行转换，默认值为 true。
- cssClass：表单组件对应的 CSS 样式类名。
- cssErrorClass：当表单组件的数据存在相应的错误时（提交表单后由服务器端产生）采用的 CSS 样式类。
- cssStyle：表单组件对应的 CSS 样式串。

此外，表单组件标签也拥有 HTML 镜像标签的各种属性，如 id、onclick、ondblclick、tabindex 等。下面通过表 17-7 对 Spring 表单标签进行说明。

表 17-7　Spring 表单标签

Spring 表单标签	说　明
<form:input/>	输入框组件标签，如<form:input path="userName"/>
<form:password/>	密码框组件标签，如<form:password path="password" />
<form:hidden/>	隐藏框组件标签，如<form:hidden path="userId" />
<form:textarea/>	多行输入框组件标签，如<form:textarea path="notes" rows="3" cols="20" />

<form:radiobutton/>	单选框组件标签，当表单对象对应的属性值和 value 值相等时，单选框选中。下面是一个代表性别的单选框：
	`<form:radiobutton path="sex" value="0"/>男`
	`<form:radiobutton path="sex" value="1"/>女`
	当表单对象的 sex 属性为 0 时，第一个单选框选中

<form:radiobuttons/>	单选框组件组标签，用于构造多个单选框，例如： `<form:radiobuttons path="sex" items="${sexOptions}" cssClass="radioCls" delimiter=" "/>`

续表

Spring 表单标签	说　明
<form:radiobuttons/>	sexOptions 可以是一个 List、String[]或 Map，该标签将产生和此变量元素相同的单选框组件。如果 sexOptions 为 List 类型，则单选框的 value 和 lable 相同；如果 sexOptions 为 Map 类型，则默认情况下 value 为 Map 的 key，而 lable 为 Map 的 value。可以通过标签的 itemValue 及 itemLabel 属性更改这一默认行为： `<form:radiobuttons path="sex" items="${sexOptions}" itemValue="value" itemLabel="key"/>` key 和 value 分别代表 Map 的键和值。<form:checkboxs/>及<form:select/>标签也拥有相似的功能。 当单选框的 path 属性值和 sexOptions 中的某个相同时，单选框选中。多个单选框通过 delimiter 指定分隔符
<form:checkbox/>	复选框组件标签，如<form:checkbox path="favorites" value="1"/>
<form:checkboxs/>	复选框组件组标签，用于构造多个复选框，例如： `<form:checkboxes path="preferences.interests" items="${interests}" cssClass="note" delimiter=" "/>` 该标签的大部分特性和<form:radiobuttons/>类似
<form:select/>	下拉框组件标签，例如： `<form:select path="city" items="${citys}"/>` 该标签的大部分特性和<form:radiobuttons />类似

<form:option/>	下拉框选项组件标签，例如：
	`<form:select path="city">`
	`<form:option value="" label="--请选择--"/>`
	`<form:options items="${cityMap}" itemValue="key" itemLabel="value"/>`
	`</form:select>`
	该标签的大部分特性和<form:radiobuttons />类似

<form:errors/>	显示表单数据转换或数据校验所对应的错误，如<form:errors path="*"/>显示表单所有的错误，<form:errors path="user*"/>显示所有以 user 为前缀的属性对应的错误，而<form:errors path="userName"/>显示特定表单对象属性的错误。可以通过 cssClass 属性指定错误的 CSS 样式类，如<form:errors path="userName" cssClass="errorClass"/>

3．关于复选框、单选框及下拉框和表单对象属性的映射问题

对于以下复选框标签：

```
<form:checkboxes path="preferences.interests" items="${interests}"
```

其生成的对应 HTML 如下：

```
<input id="favorites1" name="favorites" type="checkbox" value="1" checked="checked"/>
<input id="favorites2" name="favorites" type="checkbox" value="2" />
<input id="favorites3" name="favorites" type="checkbox" value="3" checked="checked"/>
<input id="favorites4" name="favorites" type="checkbox" value="4"/>
<input type="hidden" value="1" name="_favorites"/>literature
```

读者可能已经注意到复选框组件的后面附加了一个 hidden 组件，如粗体部分所示。这是因为当 HTML 页面中的所有复选框都没有勾选时，表单提交所对应的 HTTP 请求报文不会包含该复选框的参数名，这给 Spring 的表单数据绑定机制带来了麻烦，因为无法触发 setFavorites()方法的调用（如果这个表单对象已经缓存在 Session 中，且该属性已经有值，那么这个属性值将不会被设置为空）。

解决方法就是在每个复选框后添加一个隐藏组件，并且在对应的复选框名字前添加

一个下画线作为隐藏组件的名字。这样一来，相当于告诉 Spring MVC：在这个表单中存在这样一个复选框，Spring MVC 可以据此保证服务器端的表单对象和页面中的表单组件数据的一致性。

复选框、单选框及下拉框的组件标签都拥有类似的功能。如果手工编写 HTML 代码，在碰到单选框、复选框及下拉框时，也应该采用类似的方式，以保证 Spring MVC 数据绑定机制正常工作。当然，如果表单对象没有缓存在 Session 中，而是每次提交都重新创建，则不存在这个问题，也就无须添加一个带下画线的隐藏组件。

17.4.4 模板视图

FreeMarker 和 Velocity 是除 JSP 外被使用最多的页面模板技术。页面模板编写好页面结构，模板页面中使用一些特殊的变量标识符绑定 Java 对象的动态数据。

Spring 对 FreeMarker 和 Velocity 都提供了支持，用户可以使用它们渲染模型中的数据。Velocity 视图的使用过程和 FreeMarker 大同小异，所以我们主要以 FreeMarker 为主进行学习。

FreeMarker 是一个模板引擎，是一个基于模板生成文本输出的通用工具，FreeMarker 可以基于模板产生 HTML、XML、Java 源代码等多种类型的输出内容。用户可以使用 FreeMarker 生成基于模板的一段字符内容、直接输出为一个文件，或作为结果在程序中使用。虽然 FreeMarker 具有一些编程能力，但通常由 Java 程序准备模型数据，FreeMarker 仅负责基于模板对模型数据进行渲染的工作。

1. 编写一个 FreeMarker 模板

使用 FreeMarker 技术对代码清单 17-42 中的页面进行重写，并将它存放在 WEB-INF/ftl 目录下，如代码清单 17-43 所示。

代码清单 17-43 userListFtl.ftl：显示用户列表数据的页面

```
<html>
    <head>
        <title>smart</title>
    </head>
    <body>
        用户列表
        <table>
        <#list userList as user>①
                <tr>
                   <td>
                <a href="/user/showUser/${user.userName}.html">${user.userName}
</a>
                 </td>
                   <td>${user.realName}</td>
                 <td> ${user.birthday?string("yyyy-MM-dd")}</td>③
                </tr>
        </#list>
        <table>
```

对模型数据中的 userList 属性对象进行迭代

②

显示模型属性对象中的值

格式化日期

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

在 UserController 中添加一个处理方法，其返回的逻辑视图名为 userListFtl，模型数据中包含了一个名为 userList 的属性对象，如代码清单 17-44 所示。

代码清单 17-44　UserController.java：显示用户列表数据的处理方法

```
@RequestMapping(path = "/showUserListByFtl")
public String showUserListInFtl(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList); ①
    return "userListFtl";                 ②
}
```

以 userList 为属性名，将 User 列表添加到模型数据中。

逻辑视图名为 userListFtl

接下来的工作就是在 Spring Web 上下文中装配好 FreeMarker 的基础设施和 FreeMarker 视图解析器，使 Spring MVC 能够将 userListFtl 解析为 userListFtl.ftl 对应的视图对象。

2．如何在 Spring Web 上下文中配置 FreeMarker

在 Spring Web 上下文中配置 FreeMarker 的过程如代码清单 17-45 所示。

代码清单 17-45　smart-servlet.xml：装配FreeMarker基础设施和解析器

```
...
<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer"
    p:templateLoaderPath="/WEB-INF/ftl" ①-1
    p:defaultEncoding="UTF-8">          ①-2
    <property name="freemarkerSettings"> ①-3
        <props>
            <prop key="classic_compatible">true</prop>
        </props>
  </property>
</bean>

<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver"
    p:order="5"          ②-1
    p:suffix=".ftl"      ②-2
    p:contentType="text/html; charset=utf-8"/>②-3
```

在①处通过 FreeMarkerConfigurer 配置了 FreeMarker 环境，首先在①-1 处指定了模板文件存放的路径。由于我们的模板文件采用 UTF-8 编码格式，所以必须显式配置 defaultEncoding 属性；否则将采用系统默认的编码，造成乱码现象。

FreeMarker 拥有丰富的自定义属性，用户可以通过 freemarkerSettings 进行统一的设置（需要阅读 FreeMarker 的参考文档，以获取可配置的属性信息），这些属性名称都采用小写并用“_”连接。在①-3 处配置的 classic_compatible 属性非常重要，否则当 FreeMarker 模板碰到值为 null 的对象属性时，将抛出异常。将 classic_compatible 属性设置为 true 后，FreeMarker 将采用类似于 JSTL 标签的行为处理模型属性数据，返回一个空白字符串，而非抛出系统异常。

在搭建好 FreeMarker 环境后，就可以通过 FreeMarkerViewResolver 视图解析器将

userListFtl 的逻辑视图名解析为一个对应的 FreeMarkerView 视图对象。

在②-1 处指定该视图解析器的优先级，它将优先于 InternalResourceViewResolver 执行（因为 InternalResourceViewResolver 默认的优先级最低）。而在②-2 处指定了后缀，这样 userListFt 的逻辑视图名将解析为/WEB-INF/ftl/userListFtl.ftl 视图对象。由于 userListFtl.ftl 模板最终产生的是 HTML，且我们希望使用 UTF-8 编码格式输出内容，所以需要通过②-3 处的 contentType 属性进行相应配置。如果不指定该属性，则输出的 HTML 将产生中文乱码问题。

3．使用 Spring 为 FreeMarker 提供的宏

Spring 为 FreeMarker 提供了许多有用的宏，这些宏类似于 Spring 的 JSP 表单标签，此外还提供了一些其他有用的功能。考虑一下代码清单 17-43 中②处的以下代码：

```
<a href="/user/showUser/${user.userName}.html">${user.userName}</a>
```

这里使用的 URL 地址没有考虑部署路径，如果部署路径不是“/”，则将发生 URL 地址错误。由于应用部署目录需要在最终部署时才能确定，因此不能直接通过硬编码的方式指定。在 JSP 中，一般通过 JSTL 的<c:url>将 URL 转换为包含应用部署路径的绝对 URL。

```
<a href="<c:url value="/user/showUser/${user.userName}.html"/>">${topic.title} </a>
```

但在 FreeMarker 的模板中，该如何完成这一功能呢？答案是使用 Spring 的 FreeMarker 宏。此外，Spring 的 FreeMarker 宏还提供了本地化信息显示功能。使用该宏对代码清单 17-43 中的代码进行调整，如代码清单 17-46 所示。

代码清单 17-46　userListFtl.ftl：使用Spring的宏

```
<#import "spring.ftl" as spring /> ①
<html>                                    引入 Spring 的 FreeMarker 宏定义文件
    <head>
        <title><@spring.message "website.title"/></title>②
    </head>                                                    引用国际化资源
    <body>
        <@spring.message "user.userList.title"/>
        <table>
         <#list userList as user>
                <tr>                                  使用 Spring 的宏对 URL 进行格式化
                    <td>
                     <a href="<@spring.url '/user/showUser/${user.userName}.html'/>">③
                       ${user.userName}
                      </a>
                    </td>
                  <td>${user.realName}</td>
                  <td> ${user.birthday?string("yyyy-MM-dd")}</td>
                </tr>
         </#list>
          <table>
    </body>
</html>
```

首先在①处引入 Spring 的 FreeMarker 宏定义文件，并指定一个命名空间名，一般

设置为 spring。然后在②处通过<@spring.message/>引用国际化资源显示本地化信息。最后在③处通过<@spring.url/>宏将'/user/showUser/${user.userName}.html'转换为包含部署路径的绝对路径。

Spring 为 FreeMarker 提供的表单宏和其他宏统一在表 17-8 中介绍。

表 17-8　Spring为FreeMarker提供的宏

宏 类 型	说　明
国际化信息，code 代表国际化资源的代码	<@spring.message code/> 示例：<@spring.message "user.userName"/>
国际化信息，如果没有对应代码，则使用默认值（text）指定	<@spring.messageText code, text/> 示例：<@spring. messageText "user.userName" "用户名"/>
获取相对于应用服务器的绝对路径	<@spring.url relativeUrl/> 示例：<@spring.url "/register.html"/>
输入框组件	<@spring.formInput path, attributes, fieldType/> 示例：<@spring.formInput "user.userName" "class='inputSty'" />
隐藏组件	<@spring.formHiddenInput path, attributes/> 示例：<@spring. formHiddenInput "user.userName"/>
密码组件	<@spring.formPasswordInput path, attributes/> 示例：<@spring.formPasswordInput "user.password"/>
多行文本框	<@spring.formTextarea path, attributes/> 示例：<@spring.formTextarea "user.desc" "cols=3 rows=2"/>
单选下拉框	<@spring.formSingleSelect path, options, attributes/> 示例：<@spring.formSingleSelect "user.sex" "sexList"/>
多选框	<@spring.formMultiSelect path, options, attributes/> 示例：<@spring. formMultiSelect "user.sex" "sexList"/>
单选框	<@spring.formRadioButtons path, options separator, attributes/> 示例：<@spring. formRadioButtons "user.sex" "sexList" "|"/>
复选框	<@spring.formCheckboxes path, options, separator, attributes/> 示例：<@spring. formCheckboxes "user.sex" "sexList" "|"/>
绑定组件的错误信息（需要和组件配合使用）	<@spring.showErrors separator, classOrStyle/> 示例：<@spring.formInput "user.userName" "class='inputSty'" /> <@spring.showErrors " " "errorSty"/>

此外，也可以使用 Spring 低版本中的<@spring.bind "command.userName"/>绑定表单对象的某个属性（绑定到名为 status 的请求参数中），其后可以通过${spring.status.expression}和${spring.status.value}引用这个属性的名字和值。不过，这种方式显然已经过时，直接使用 Spring 的 FreeMarker 宏是更简单的方式。

17.4.5　Excel

如果希望使用 Excel 展示用户列表，则仅需扩展 Spring 的 AbstractExcelView 或

AbstractJExcelView 即可。实现 buildExcelDocument()方法，在方法中使用模型数据对象即可构造 Excel 文档。AbstractExcelView 基于 POI API，而 AbstractJExcelView 基于 JExcelAPI。下面通过扩展 AbstractExcelView 定义显示用户列表的 Excel 视图类，如代码清单 17-47 所示。

代码清单 17-47　UserListExcelView.java

```
package com.smart.web;
...
import org.apache.commons.lang.time.DateFormatUtils;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.springframework.web.servlet.view.document.AbstractExcelView;

public class UserListExcelView extends AbstractExcelView {①

    @Override
    protected void buildExcelDocument(Map<String, Object> model,
            HSSFWorkbook workbook, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        response.setHeader("Content-Disposition", "inline; filename="+
                new String("用户列表".getBytes(), "iso8859-1")); ②
        List<User> userList = (List<User>) model.get("userList");
        HSSFSheet sheet = workbook.createSheet("users");
        HSSFRow header = sheet.createRow(0);
        header.createCell(0).setCellValue("账号");
        header.createCell(1).setCellValue("姓名");
        header.createCell(2).setCellValue("生日");

        int rowNum = 1;
        for (User user : userList) {
            HSSFRow row = sheet.createRow(rowNum++);
            row.createCell(0).setCellValue(user.getUserName());
            row.createCell(1).setCellValue(user.getRealName());
            String createDate = DateFormatUtils.format(user.getBirthday(),
                    "yyyy-MM-dd");
            row.createCell(2).setCellValue(createDate);
        }
    }
}
```

继承于 AbstractExcelView，使用 POI 技术创建 Excel 文档

Excel 文档名称必须编码为 iso8859-1，否则会显示乱码

需要特别注意的是②处对响应报文头的设置，它让浏览器在页面中直接显示 Excel 文件，而非显示一个对话框提示用户下载或打开文件。如果使用的是 IE 浏览器，则在使用该报文头时，IE 会在页面中直接显示对应的 Excel 文件；不过 Chrome 等浏览器还是采用下载的方式，其主要出于安全方面的考虑。如果希望在 IE 浏览器中显示提示用户下载或打开文件的对话框，则可以对响应报文头进行调整：

```
response.setHeader("Content-Disposition", "attachment; filename="+
                new String("用户列表".getBytes(), "iso8859-1"));
```

编写好 Excel 视图类后，必须在 smart-servlet.xml 中进行相应的配置。

```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"p:order="10"/>
<bean id="userListExcel" class="com.smart.web.UserListExcelView"/>
```

由于 UserListExcelView 代表的 Excel 视图对象是一个 Bean，因此，我们使用 BeanNameViewResolver 作为视图解析器，将其优先级序号设置为 10，这样它的优先级将排在 17.4.4 节所介绍的 FreeMarkerViewResolver 之后。

在 UserController 中添加一个相应的请求处理方法。

```
@RequestMapping(path = "/showUserListByXls")
public String  showUserListInExcel(ModelMap mm){
    ...
    userList.add(user1);
    userList.add(user2);
    mm.addAttribute("userList",userList);
    return "userListExcel";
}
```

这样，在浏览器地址栏中输入如下 URL 时，根据浏览器的不同，将显示或下载一个 Excel 文件。

```
http://localhost:8080/chapter17/user/showUserListByXls.html
```

17.4.6　PDF

PDF 视图和 Excel 类似，也使用一个 Bean 作为视图对象，如代码清单 17-48 所示。

代码清单 17-48　UserListPdfView.java

```
package com.smart.web;
...
import org.apache.commons.lang.time.DateFormatUtils;
import org.springframework.web.servlet.view.document.AbstractPdfView;
import com.lowagie.text.Cell;
import com.lowagie.text.Document;
import com.lowagie.text.Element;
import com.lowagie.text.Font;
import com.lowagie.text.Phrase;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class UserListPdfView extends AbstractPdfView {

    @Override
    protected void buildPdfDocument(Map<String, Object> model,
            Document document, PdfWriter writer, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        response.setHeader("Content-Disposition", "inline; filename="+
                new String("用户列表".getBytes(), "iso8859-1"));
        List<User> userList = (List<User>) model.get("userList");
        Table table = new Table(3);
        table.setWidth(80);
        table.setBorder(1);
        table.getDefaultCell().setHorizontalAlignment(Element.ALIGN_CENTER);
```

```
        table.getDefaultCell().setVerticalAlignment(Element.ALIGN_MIDDLE);
                                                                    使用中文字体
        BaseFont cnBaseFont = BaseFont.createFont("STSongStd-Light",
                                        "UniGB-UCS2-H", false);          ①
        Font cnFont = new Font(cnBaseFont, 10, Font.NORMAL, Color.BLUE);

        table.addCell(buildFontCell("账号",cnFont));  ②
        table.addCell(buildFontCell("姓名",cnFont));   对于中文字符，要使用中文字段构
        table.addCell(buildFontCell("生日",cnFont));   造 Cell 对象，否则会产生乱码
        for (User user : userList) {
            table.addCell(user.getUserName());③ ← 英文字符可直接添加到 Cell 中
            table.addCell(buildFontCell(user.getRealName(),cnFont));
            String createDate = DateFormatUtils.format(user.getBirthday(),
                    "yyyy-MM-dd");
            table.addCell(createDate);
        }
        document.add(table);

    }

    private Cell buildFontCell(String content,Font font) throws RuntimeException{ ④
        try {
            Phrase phrase = new Phrase(content, font);
            return new Cell(phrase);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
  }
}
```

默认的 lowagie 类包不支持亚洲文字（包括简体中文、繁体中文、日文、韩文等），当输出内容遇到亚洲文字时，直接输出空白字符。针对这个问题，lowagie 提供了一个附加的 iTextAsian.jar 包，它包含了亚洲文字字体，需要在 pom.xml 中添加这个构件（由于 Maven 公服上没有这个构件，所以本书配套网盘的 libs 目录下提供了该构件）。

```
<dependency>
    <groupId>com.lowagie</groupId>
    <artifactId>iTextAsian</artifactId>
    <version>2.0.7</version>
    <scope>system</scope>
    <systemPath>${basedir}/../libs/iTextAsian.jar</systemPath>
</dependency>
```

在①处创建了一个中文字体对象。在④处添加了一个 buildFontCell()方法，它将需要输出的内容通过 Phrase 对象使用特殊字体进行封装，只要传入的字体对象是中文字体，返回的 Cell 对象中的中文内容就可以正常显示。

UserListPdfView 和 UserListExcelView 视图一样，可以采用 BeanNameViewResolver 作为视图解析器，因此仅需在 smart-servlet.xml 中添加视图 Bean 的声明即可。

```
<bean id="userListPdf" class="com.smart.web.UserListPdfView"/>
```

在 UserController 中添加一个 showUserListInPdf()方法。

```
@RequestMapping(path = "/showUserListByPdf")
public String  showUserListInPdf(ModelMap mm){
```

```
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList);
    return "userListPdf";
}
```

在浏览器地址栏中输入如下 URL 时，根据浏览器的不同，将显示或下载一个 PDF 文件。

```
http://localhost:8080/chapter17/user/showUserListByPdf.html
```

17.4.7　输出 XML

Spring MVC 还可以将模型中的数据以 XML 的形式输出，其对应的视图对象为 MarshallingView。下面在 smart-servlet.xml 中添加 MarshallingView 的配置，如代码清单 17-49 所示。

代码清单 17-49　smart-servlet.xml：添加MarshallingView

```
...
<bean id="userListXml"
class="org.springframework.web.servlet.view.xml.MarshallingView"
        p:modelKey="userList"
        p:marshaller-ref="xmlMarshaller"/>
<bean id="xmlMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="streamDriver">
      <bean class="com.thoughtworks.xstream.io.xml.StaxDriver" />
    </property>
    <property name="annotatedClasses">
        <list>
            <value>com.smart.domain.User</value>
        </list>
    </property>
</bean>
```

MarshallingView 使用 Marshaller 将模型数据转换为 XML，通过 marshaller 属性注入一个 Marshaller 实例。在默认情况下，MarshallingView 会将模型中的所有属性都转换为 XML。由于模型属性包含很多隐式数据，直接将模型中的所有数据全部输出往往并不是我们所期望的。MarshallingView 允许通过 modelKey 指定模型中的哪个属性需要输出为 XML。

在 UserController 中添加一个处理方法，返回 userListXml 逻辑视图名。

```
@RequestMapping(path = "/showUserListByXml")
public String  showUserListInXml(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList);
    return "userListXml";
}
```

在浏览器地址栏中输入如下 URL 时，将返回 XML 内容的响应。

```
http://localhost:8080/chapter17/user/showUserListByXml.html
```

17.4.8 输出 JSON

Spring 4.0 MVC 的 MappingJackson2JsonView 借助 Jackson 框架的 ObjectMapper 将模型数据转换为 JSON 格式输出。由于 MappingJackson2JsonView 也是一个 Bean，可以通过 BeanNameViewResolver 进行解析，因此仅需在 smart-servlet.xml 中直接配置即可。

```
<bean id="userListJson"
      class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"
      p:modelKeys="userList"/>
```

在默认情况下，MappingJackson2JsonView 会将模型中的所有数据全部输出为 JSON，这显然是不适合的，可以通过 modelKeys 指定模型中的哪些属性需要输出。

在 UserController 中添加一个返回 userListJson 逻辑视图名的方法。

```
@RequestMapping(path = "/showUserListByJson")
public String  showUserListInJson(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    ...
    mm.addAttribute("userList",userList);
    return "userListJson";
}
```

在浏览器地址栏中输入 http://localhost:8080/chapter17/user/showUserListByJson.html，将以 JSON 格式输出模型中名为 userList 的属性对象。

17.4.9 使用 XmlViewResolver

如果视图对象的 Bean 数目太多，那么直接在 smart-servlet.xml 文件中配置，势必影响主配置文件的简洁性。XmlViewResolver 和 BeanNameViewResolver 功能相似，唯一不同的是它可以将视图 Bean 定义在一个独立的 XML 文件中。要使用 XmlViewResolver，必须在 smart-servlet.xml 添加以下片段：

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver"
      p:order="20"
      p:location="WEB-INF/views/smart-views.xml"/>
```

在默认情况下，XmlViewResolver 在 WEB-INF/views.xml 中查找视图 Bean 的定义文件。我们将视图 Bean 定义在 WEB-INF/views/smart-views.xml 文件中，所以需要通过 location 显式指定。

定义视图 Bean 的 smart-views.xml 文件，其格式和普通的 Spring 配置文件没有任何区别。

```
<?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:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
    <bean id="userListJson1"
```

```
        class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" ①
        p:modelKeys="userList" />
    <bean id="userListExcel1" class="com.smart.web.UserListExcelView"/>
    <bean id="userListPdf1" class="com.smart.web.UserListPdfView"/>
</beans>
```

定义视图 Bean

不过在 smart-views.xml 文件中定义的 Bean 不能被 Spring Web 上下文的其他 Bean 引用，它是被 XmlViewResolver 独享的。

17.4.10　使用 ResourceBundleViewResolver

对于同一个逻辑视图名，如果希望为不同地区的用户提供不同类型的视图，则应该考虑使用 ResourceBundleViewResolver。和 XmlViewResolver 将视图定义在 XML 文件中不同，ResourceBundleViewResolver 通过一个国际化资源文件定义视图对象。

例如，假设小春论坛需要考虑不同国家和地区客户的不同展现需求：中国人喜欢使用 HTML 页面的展现方式，而美国人喜欢使用 PDF 的展现方式——虽然这个需求很“怪诞”，但不影响我们把它作为一个例子。可以通过 ResourceBundleViewResolver 来实现这个需求，在 smart-servlet.xml 中添加以下代码：

```
<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver"
    p:order="30"
    p:basename="i18n/views"/>
```

通过 basename 属性指定视图国际化资源文件的基名。对应于中国大陆的视图资源文件为 views_**zh_CN**.properties，其内容为：

```
userListi18n.(class)=org.springframework.web.servlet.view.JstlView ①
userListi18n.url=/WEB-INF/views/user/userList.jsp ②
```

其中，使用<逻辑视图名>.(class)属性设置视图实现类，如①处所示。如果视图实现需要一个 URL 资源的支持，则可以通过<逻辑视图名>.url 设置这个 URL 资源。对于 JstlView 视图实现类来说，其实是通过调用 JstlView#setUrl()设置<逻辑视图名>.url 指定 JSP 文件的。

对应于美国的视图资源文件为 views_**en_US**.properties，其内容为：

```
userListi18n.(class)=com.smart.web.UserListPdfView
```

当美国用户查看论坛的用户列表时，ResourceBundleViewResolver 视图解析器将其解析为一个 PDF 文档。

在一个视图国际化资源文件中可以定义多个视图对象，如下：

```
welcomeView.(class)=org.springframework.web.servlet.view.JstlView ①
welcomeView.url=/WEB-INF/jsp/welcome.jsp

vetsView.(class)=org.springframework.web.servlet.view.JstlView ①
vetsView.url=/WEB-INF/jsp/vets.jsp
...
```

第一个视图

第二个视图

17.4.11 混合使用多种视图技术

1. ContentNegotiatingViewResolver

Spring 最重要的一个新功能就是对 REST 编程风格的支持。REST 风格的应用对资源的 URL 定义有严格的要求：一个资源对象对应唯一的 URL。在前面的例子中，我们使用不同的视图显示用户列表，如果把“用户列表”看作一个资源，那么我们希望通过一个相同的 URL 访问这个资源，而不是使用不同的 URL 访问该资源的不同视图。

17.2.5 节介绍了使用 HttpMessageConverter 对标注了@ResponseBody 或返回值为 ResponseEntity 的处理器方法进行响应信息转换的内容，Spring MVC 可以根据请求报文头的 Accept 属性选择适合的 HttpMessageConverter 将处理方法的返回值以 XML、JSON 等不同形式输出响应。也就是说，调用者可以通过设置请求报文头的 Accept 值控制服务器端返回的数据格式，从而实现对同一资源采用相同 URL 的 REST 编程风格。

但是基于 HttpMessageConverter 的实现方式存在以下限制。

- 只能通过请求报文头的 Accept 值控制服务器端返回的数据格式。如果客户端是浏览器，那么，除非使用 AJAX，否则很难控制 Accept 报文头的值。一般情况下，该值是由浏览器自己决定的。
- 无法通过 URL 扩展名或请求参数控制服务器端的资源输出形式，因此无法将其对应一个 URL 发布出去。
- 如果希望使用 XML、JSON、一个网页等形式输出资源，则 HttpMessageConverter 很难达到要求。因为 HttpMessageConverter 很难调用一个视图对象渲染模型，它直接负责将资源输出为某一内容形式。

所以，如果希望将资源以 XML、JSON 等纯数据的格式输出，且不在意使用报文头控制资源输出，那么适合选择 HttpMessageConverter 实现方式。否则，建议采用 Spring MVC 的 ContentNegotiatingViewResolver 视图解析器，虽然它和 HttpMessageConverter 在功能上有一些重叠，但在使用上更加灵活。

ContentNegotiatingViewResolver 名为“视图解析器”，但它并不是一个传统意义上的视图解析器，它像一个仲裁机构或一个代理人，根据请求信息从上下文中选择一个适合的视图解析器负责解析。因此，一般将 ContentNegotiatingViewResolver 的优先级设为最高，以保证优先调用 ContentNegotiatingViewResolver。

ContentNegotiatingViewResolver 根据请求所要求的 MIME 类型决定由哪个视图解析器负责处理，它按照如下方式工作。

① 如果其 favorPathExtension 属性设置为 true（默认为 true），则根据 URL 中的文件扩展名确定 MIME 类型（如 userList.xml,userList.json 等）。

② 如果其 favorParameter 属性设置为 true（默认为 false），则根据请求参数的值确定 MIME 类型。默认的请求参数是 formart。可以通过 parameterName 属性指定一个自定义的参数。

③ 如果还没有找到对应的 MIME 类型，且 Java Activation Framework（JAF）位于类路径下，则通过 JAF 的 FileTypeMap.getContentType(url)方法对 URL 进行判断，以获取对应的 MIME 类型。

④ 如果以上步骤都失败，且 ignoreAcceptHeader 属性设置为 false（默认为 false），则采用 Accept 请求报文头的值确定 MIME 类型。由于不同浏览器产生的 Accept 头都是不一样的，所以一般不建议采用 Accept 确定 MIME 类型。

2. 使用同一 URL 获取不同形式的返回内容

假设希望使用以下 REST 风格的 URL 以不同的 MIME 格式获取相同的资源(用户列表)。

- /user/showUserListMix.html：返回一个 HTML 页面显示的用户列表。
- /user/showUserListMix.html?content=xml：返回 XML 格式的用户列表。
- /user/showUserListMix.html?content=json：返回 JSON 格式的用户列表。

首先，在 UserController 中添加一个处理方法。

```
@RequestMapping(path = "/showUserListMix")
public String  showUserListMix(ModelMap mm){
    List<User> userList = new ArrayList<User>();
    …
    mm.addAttribute("userList",userList);
    return "userListMix";
}
```

接着，在 smart-servlet.xml 中添加以下配置片段，配置一个 ContentNegotiatingViewResolver，采用请求参数指定内容资源的返回类型，如代码清单 17-50 所示。

代码清单 17-50　smart-servlet.xml：添加ContentNegotiatingViewResolver

```
<bean id="contentNegotiationManager"

class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean"
        p:ignoreAcceptHeader="true"
        p:favorPathExtension="false"
        p:favorParameter="true"
        p:parameterName="format"
        p:defaultContentType="text/html">                    ①
      <property name="mediaTypes">
         <value>
            html=text/html
            xml=application/xml
            json=application/json
         </value>
      </property>
   </bean>

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver"
    p:order="0" >                                            ②
    <property name="contentNegotiationManager" ref="contentNegotiationManager"/>
    <property name="defaultViews">
        <list>
```

不支持扩展文件名，不支持 Accept 报文头指定 MIME 类型，通过请求参数指定 MIME 类型，参数名为 content，请求类型的参数值和 MIME 类型的映射列表

使该视图解析器优先级最高

```
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2
JsonView"
                  p:modelKeys="userList" />③
            <bean class="org.springframework.web.servlet.view.xml.MarshallingView"
                  p:modelKey="userList" p:marshaller-ref="xmlMarshaller" />④
        </list>
    </property>
</bean>
<!--以下是上下文中已经配置的视图解析器 -->
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"
      p:order="10" />
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"⑤
      p:order="100"
      p:viewClass="org.springframework.web.servlet.view.JstlView"
      p:prefix="/WEB-INF/views/"
      p:suffix=".jsp" />
```

ContentNegotiatingViewResolver 会在上下文中查找和 MIME 类型匹配的视图解析器，并委托它们进行视图解析。这样通过 content 请求参数就可以控制资源的输出格式。

③处配置的是 JSON 视图对象，而④处配置的是 XML 视图对象，它们都是默认的候选视图对象，会覆盖对应视图解析器返回的视图对象。换句话说，当资源 URL 带 content=json 参数时，直接返回③处对应的视图对象；当资源 URL 带 content=xml 参数时，直接返回④处对应的视图对象；当资源 URL 带 content=html 参数时，由⑤处的 InternalResourceViewResolver 解析；当资源 URL 不带 content 参数时，由于已经通过 defaultContentType="text/html" 指定了默认的 MIME 类型，因此也由⑤处的 InternalResourceViewResolver 解析。

17.5 本地化解析

17.5.1 本地化的概念

一般情况下，Web 应用根据客户端浏览器的设置判断客户端的本地化类型，用户可以通过 IE 菜单“工具→Internet 选项→语言”，在打开的“语言首选项”对话框中选择本地化类型。浏览器中设置的本地化类型会包含在 HTML 请求报文头中发送给 Web 服务器，确切地说是通过报文头的 Accept-Language 参数将“语言首选项”对话框中选择的语言发送到服务器，成为服务器判别客户端本地化类型的依据。

如果 Web 应用基于这种方式提供本地化页面，则用户只得通过改变浏览器的设置进行本地化类型的切换。读者是否见过一些网站在页面的右上角提供了语言切换的图标，如英文、中文、繁体等字样。这种语言切换功能不要求用户更改浏览器的设置，它们通过 Cookie、Session 或请求参数即可切换本地化类型，为用户的使用带来了便利。

在默认情况下，Spring MVC 根据 Accept-Language 参数判断客户端的本地化类型。此外，它还提供了多种指定客户端本地化类型的方式，如通过 Cookie、Session 指定。事实上，当收到请求时，Spring MVC 在上下文中寻找一个本地化解析器（LocaleResolver），找到后使用它获取请求所对应的本地化类型信息。

除此之外，Spring MVC 还允许装配一个动态更改本地化类型的拦截器，这样通过指定一个请求参数就可以控制单个请求的本地化类型。本地化解析器和拦截器都定义在 org.springframework.web.servlet.i18n 包中，用户可以在 DispatcherServlet 上下文中配置它们。

Spring 提供了以下 4 个本地化解析器。

- AcceptHeaderLocaleResolver：根据 HTTP 报文头的 Accept-Language 参数确定本地化类型。如果没有显式定义本地化解析器，则 Spring MVC 默认采用 AcceptHeaderLocaleResolver。
- CookieLocaleResolver：根据指定的 Cookie 值确定本地化类型。
- SessionLocaleResolver：根据 Session 中特定的属性值确定本地化类型。
- LocaleChangeInterceptor：从请求参数中获取本次请求对应的本地化类型。

在 17.3.4 节的实例中，我们为表单校验错误信息提供了国际化的支持，当表单数据校验存在错误时，Spring MVC 使用默认的 AcceptHeaderLocaleResolver 抽取请求报文头的 Accept-Language 属性以确定客户端本地化类型，并返回对应的本地化错误信息。下面采用其他的本地化解析器解析客户端的本地化类型。

17.5.2　使用 CookieLocaleResolver

要使用 Cookie 保存本地化类型信息，服务器端要做的事情很简单，只需在 Spring MVC 上下文中配置一个 CookieLocaleResolver 就可以了，DispatcherServlet 会自动识别本地化解析器并装配它。

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
    p:cookieName="clientLanguage"①
    p:cookieMaxAge="100000"②
    p:cookiePath="/"    ③
    p:defaultLocale="zh_CN"/>④
```

客户端保存本地化类型的 Cookie 名

Cookie 最大存活秒数

Cookie 保存路径

默认本地化类型

这样，客户端只要通过 JavaScript 更改 clientLanguage 这个 Cookie 的值，就可以控制服务器端返回相应的本地化页面。

实战经验

在实际的 Web 应用中，一般将用户个性化配置信息（包括本地化类型信息）保存在数据库中。当用户登录系统时，先加载这些个性化信息并保存在特定的媒介中（如

Cookie 或 Session),这样用户的个性化设置就可以永远生效而不会随着 Cookie 或 Session 的清除或过期而消失。

17.5.3 使用 SessionLocaleResolver

SessionLocaleResolver 查找 Session 中属性名为 SessionLocaleResolver.LOCALE_SESSION_ATTRIBUTE_NAME 的属性，并将其转换为 Locale 对象，以此作为客户端的本地化类型。如果应用程序使用 Session 维护客户的信息，则 SessionLocaleResolver 是最适合不过的。

用户仅需将 SessionLocaleResolver 的配置添加到 Spring MVC 上下文的配置文件中即可。

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>
```

SessionLocaleResolver 和 CookieLocaleResolver 的区别是，前者一般要求用户登录后生成相应的用户会话才有效，而后者只要浏览器有 Cookie 存在即可生效。

17.5.4 使用 LocaleChangeInterceptor

很多国际型的网站都允许通过一个请求参数控制网站的本地化，如 www.xxx.com?locale=zh_CN 返回对应中国大陆的本地化网页，而 www.xxx.com?locale=en 返回本地化为英语的网页。这样，网站使用者可以通过对 URL 的控制返回不同本地化的页面，非常灵活。在 Spring MVC 中，这一需求可以通过 LocaleChangeInterceptor 过滤器来完成。

在 Spring MVC 上下文中通过<mvc:interceptors>配置过滤器。

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
      p:cookieName="clientLanguage"
      p:cookieMaxAge="100000"
      p:cookiePath="/"
      p:defaultLocale="zh_CN"/>①
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />②
</mvc:interceptors>
```

LocaleChangeInterceptor 在默认情况下通过 locale 请求参数获取本次请求对应的本地化类型，用户可以通过其 paramName 属性指定一个其他参数名。由于 LocaleChangeInterceptor 的主要任务是从请求中获取本地化类型并将其设置给真正的本地化解析器，所以在配置 LocaleChangeInterceptor 之前，必须在上下文中先配置一个本地化解析器，如①处所示。值得注意的是，由于 AcceptHeaderLocaleResolver 是从请求报文头获取本地化信息的，因此不能被动态更改。如果配置了 LocaleChangeInterceptor，则只能选择 CookieLocaleResolver 或 SessionLocaleResolver。

17.6　文件上传

Spring MVC 为文件上传提供了直接支持，这种支持是通过即插即用的 MultipartResolver 实现的。Spring 使用 Jakarta Commons FileUpload 技术实现了一个 MultipartResolver 实现类：CommonsMultipartResolver。

在 Spring MVC 上下文中默认没有装配 MultipartResolver，因此默认情况下不能处理文件的上传工作。如果想使用 Spring 的文件上传功能，则需要先在上下文中配置 MultipartResolver。

17.6.1　配置 MultipartResolver

下面使用 CommonsMultipartResolver 配置一个 MultipartResolver 解析器。

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
      p:defaultEncoding="UTF-8"①  ←— 请求的编码格式，默认为 ISO-8859-1
      p:maxUploadSize="5242880"②  ←— 上传文件大小上限，单位为字节(5MB)
      p:uploadTempDir="file:/d:/temp"/>③  ←— 上传文件的临时路径
```

defaultEncoding 必须和用户 JSP 的 pageEncoding 属性一致，以便正确读取表单的内容。uploadTempDir 是文件上传过程中所使用的临时目录，文件上传完成后，临时目录中的临时文件会被自动清除。

为了让 CommonsMultipartResolver 正常工作，必须先将 Jakarta Commons FileUpload 及 Jakarta Commons io 的类包添加到类路径下。

17.6.2　编写控制器和文件上传表单页面

在 UserController 中添加一个用于处理用户头像上传的方法，如代码清单 17-51 所示。

代码清单 17-51　UserController.java：上传文件

```
package com.smart.web;
...
import org.springframework.web.multipart.MultipartFile;
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(path = "/uploadPage") ①
    public String updatePage() {
        return "uploadPage";
    }
    @RequestMapping(path = "/upload")
    public String updateThumb(@RequestParam("name") String name, ②
                              @RequestParam("file") MultipartFile file)throws Exception{
```

上传的文件自动绑定到 MultipartFile 中

```
        if (!file.isEmpty()) {
            file.transferTo(new File("d:/temp/"+file.getOriginalFilename()));
            return "redirect:success.html";
        }else{
            return "redirect:fail.html";
        }
    }
}
```

Spring MVC 会将上传文件绑定到 MultipartFile 对象中。MultipartFile 提供了获取上传文件内容、文件名等方法，通过其 transferTo()方法还可将文件存储到硬件中，具体说明如下。

- byte[] getBytes()：获取文件数据。
- String getContentType()：获取文件 MIME 类型，如 image/pjpeg、text/plain 等。
- InputStream getInputStream()：获取文件流。
- String getName()：获取表单中文件组件的名字。
- String getOriginalFilename()：获取上传文件的原名。
- long getSize()：获取文件的字节大小，单位为 Byte。
- boolean isEmpty()：是否有上传的文件。
- void transferTo(File dest)：可以使用该文件将上传文件保存到一个目标文件中。

负责上传文件的表单和一般表单有一些区别，表单的编码类型必须是 multipart/form-data 类型。

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c"  uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <head>
        <title>请上传用户头像</title>
    </head>
    <body>
        <h1>
            请选择上传的头像文件
        </h1>
        <form method="post" action="<c:url value="/user/upload.html"/>"
                            enctype="multipart/form-data">①
            <input type="text" name="name" />
            <input type="file" name="file" />②
            <input type="submit" />
        </form>
    </body>
</html>
```

指定表单内容类型，以便支持文件上传

上传文件的组件名

17.7 WebSocket 支持

17.7.1 使用 WebSocket

通常应用程序之间发送消息会使用诸如 JMS、AMQP 等技术，但是如果要实现浏览

器与服务器的全双工通信，则以上技术是不适用的，但在互联网时代，这样的需求又是普遍存在的。随着 HTML 5 的诞生，一个新的协议也应运而生，这便是 WebSocket 协议，它很好地解决了浏览器与服务器全双工通信的问题，而且相对于传统的解决方案能更好地节省服务器资源和带宽并达到实时通信。Java EE 7.0 已经支持 WebSocket 协议，相应的，Spring 4.0 也为 WebSocket 通信提供了支持。

Spring 4.0 主要为 WebSocket 通信提供了以下几个方面的支持。

- 发送和接收消息的 API。
- 用来发送消息的模板。
- 支持 SockJS，用来解决浏览器、服务器及代理不支持 WebSocket 的问题。

事实上，WebSocket 通信可以用在任何类型的应用中，其最简单的形式是在两个应用之间建立通信的通道，如图 17-16 所示。

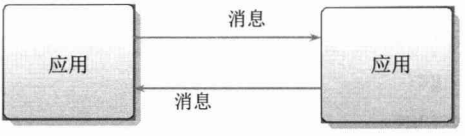


图 17-16 应用之间的 WebSocket 通信

如图 17-16 所示，位于 WebSocket 一端的应用发送消息，而另一端的应用处理消息。因为它是全双工通信的，所以每一端既可以发送又可以处理消息。但是 WebSocket 最为常见的应用场景是实现服务器和基于浏览器的应用之间的通信。在浏览器中使用 JavaScript 开启一个到达服务器的连接，服务器通过这个连接发送更新到浏览器中。相对于传统的在 Web 端轮询服务器及使用 Flash 中的 Socket 和 XMLSocket，这种方式更加高效、便捷。

下面通过一个在 Web 端使用 JavaScript 与服务器端相互传递“hello world!”文本信息的例子来展示 WebSocket 的用法。

Spring 提供了一个 WebSocketHandler 接口，该接口定义了 5 个 WebSocket 相关的接口方法。一如 Spring 惯有的优雅和便捷性，Spring 拥有一个 WebSocketHandler 接口的抽象实现类 AbstractWebSocketHandler，继承该类便可以选择性地实现感兴趣的方法，如代码清单 17-52 所示。

代码清单 17-52　MyWebSocketHandler.java

```
public class MyWebSocketHandler extends AbstractWebSocketHandler {
    @Override
    protected void handleTextMessage(
            WebSocketSession session, TextMessage message)  throws Exception {
        //处理文本消息
        System.out.println("收到消息" + message.getPayload());
        //模拟延时
        Thread.sleep(2000);
        //发送文本消息
        System.out.println("发送消息: hello world!");
        session.sendMessage(new TextMessage("hello world!"));
```

```
    }

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws Exception {
        System.out.println("建立连接");
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status)
                                                    throws Exception {
        System.out.println("关闭连接");
    }
}
```

可以看到，尽管 AbstractWebSocketHandler 是一个抽象类，但是我们并不需要重载所有的方法，在该抽象类中所有的方法都以空操作的方式实现，所以我们可选择性地重载相应的方法。除了重载 WebSocketHandler 中所定义的 5 个方法外，AbstractWebSocketHandler 还定义了以下 3 个方法：

- handleTextMessage。
- handleBinaryMessage。
- handlePongMessage。

这 3 个方法是 handleMessage()方法针对具体不同类型消息处理的实现，顾名思义，handleTextMessage()是处理文本消息类型的方法，handleBinaryMessage()是处理二进制消息类型的方法，handlePongMessage()是处理 Pong 消息类型的方法。因为 MyWebSocketHandler 将处理文本类型的消息“hello world!”，所以我们选择重载 handleTextMessage()方法，在接收文本消息后再简单地输出消息。这样，一个简单的消息处理类就完成了，需要在 applicationContext.xml 中增加相关的配置。

```
//将helloHandler映射到“/hello”
<websocket:handlers>
    <websocket:mapping  handler="helloHandler" path="/hello"/>
</websocket:handlers>

  //声明MyWebSocketHandler Bean
<bean id="helloHandler" class="com.smart.web.MyWebSocketHandler"/>
```

接下来看看客户端是怎么实现的。它会发送“hello world!”文本消息到服务器，并监听来自服务器的文本消息。代码清单 17-53 展示了如何使用 JavaScript 开启一个 WebSocket 并使用它发送消息给服务器。

代码清单 17-53　hello.jsp：客户端代码

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
  <head>
    <title>websocket测试</title>
    <script type="text/javascript">
      var url = 'ws://' +window.location.host + '<%=request.getContextPath()%>/hello';
      var sock = new WebSocket(url);  //打开WebSocket
```

```
    sock.onopen = function() {    //处理连接开启事件
      console.log('开启WebSocket连接！');
      sayHello();
    }

    sock.onmessage = function(e) {    //处理消息
      console.log('接收消息：', e.data);
      setTimeout(function(){sayHello()}, 2000);
    }

    sock.onclose = function() {  //连接关闭事件
      console.log('关闭WebSocket连接！');
    }

    function sayHello() {  //发送消息
      console.log('发送消息：hello world！')
      sock.send('hello world!');
    }
  </script>
 </head>
 <body>
  hello world!
 </body>
</html>}
```

在上述代码清单中，使用原生的 JavaScript 创建了 WebSocket 实例。通过创建 WebSocket 实例，实际上打开了一个给定 URL 的 WebSocket。注意，URL 使用了“ws://”前缀，表明这是一个 WebSocket 连接。

WebSocket 创建完毕后，接下来的代码实现了 WebSocket 的事件处理功能。WebSocket 的 onopen、onmessage 和 onclose 事件分别对应 MyWebSocketHandler 类的 afterConnectionEstablished()、handleTextMessage()和 afterConnectionClosed()方法。在 onopen 事件中会调用 sayHello()方法，在该 WebSocket 上发送“hello world!”消息，通过发送“hello world!”，这个双向通信即建立起来。服务器端的 MyWebSocketHandler 作为响应将“hello world!”发送回来，当客户端收到来自服务器端的消息后，onmessage 又会发送一个“hello world!”给服务器端，如此循环反复。

因为推送信息打印在 Web 控制台上，所以读者可以打开浏览器的开发者调试工具来观察效果。如果是 Chrome 浏览器，则按 F12 键后单击 Console 即可观察控制台上的输出，如图 17-17 所示。

图 17-17　Web 客户端控制台输出信息

在服务器端，也可以看到相关的日志信息，如图 17-18 所示。

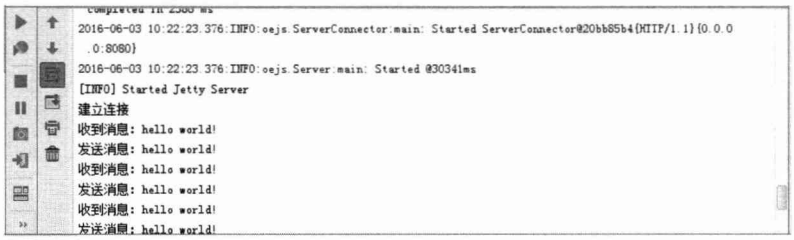


图 17-18　服务器端控制台输出信息

17.7.2　WebSocket 的限制

值得注意的是，WebSocket 是一个比较新的规范，在 Web 浏览器和应用服务器上依然没有得到全面的支持。FireFox 和 Chrome 早已完整支持 WebSocket，但是其他浏览器才刚刚开始支持 WebSocket。下面是主流浏览器对 WebSocket 支持的情况说明。

- Internet Explorer：10.0（完整支持）。
- FireFox：4.0（部分支持），6.0（完整支持）。
- Chrome：4.0（部分支持），13.0（完整支持）。
- Safari：5.0（部分支持），12.10（完整支持）。
- Opera：11.0（部分支持），12.10（完整支持）。
- iOS Safari：4.2（部分支持），6.0（完整支持）。。
- Android Browser：4.4（完整支持）。

可以看出，使用 WebSocket 在浏览器端存在一定的限制，其实服务器端的情况也并不乐观，Tomcat 从 7.0.26 版本开始才部分支持 WebSocket，直到 8.0 版本后才真正地全面支持 WebSocket，Jetty 也是到 7.0 版本才开始支持 WebSocket 的。因本书其他章节使用的 Jetty 版本为 6.1.25，并不支持 WebSocket，所以本章所使用的 Jetty 插件特地使用 9.2.1 版本。

即便在浏览器和服务器都支持 WebSocket 的情况下也可能出现问题，比如防火墙通常会限制 HTTP 以外的流量，有可能还没有配置允许进行 WebSocket 通信。但是不要灰心，只要是好的东西总会有完美的替代方案，这就是 SockJS。SockJS 是一个在浏览器上运行的 JavaScript 库，如果浏览器不支持 WebSocket，则该库可以模拟对 WebSocket 的支持，实现浏览器和 Web 服务器之间低延迟、全双工、跨域的通信通道，其 API 和 WebSocket 几乎一样。关于 SockJS 本书不再详细介绍，有兴趣的读者可在网上查阅相关资料自行研究学习。

17.8　杂项

17.8.1　静态资源处理

优雅 REST 风格的资源 URL 不希望带.html 或.do 等后缀，以下是几个优雅的 URL:

- /blog/tom：用户 tom 的 blog 资源。
- /forum/java：java 论坛版块资源。
- /order/4321：订单号为 4321 的订单资源。

由于早期的 Spring MVC 不能很好地处理静态资源，所以在 web.xml 中配置 DispatcherServlet 的请求映射时，往往采用*.do、*.xhtml 等方式。这就决定了请求 URL 必须是一个带后缀的 URL，而无法采用真正 REST 风格的 URL。

如果将 DispatcherServlet 请求映射配置为“/”，则 Spring MVC 将捕获 Web 容器所有的请求，包括静态资源的请求，Spring MVC 会将它们当成一个普通请求处理，因找不到对应的处理器而导致错误。

如何让 Spring 框架能够捕获所有 URL 的请求，同时又将静态资源的请求转由 Web 容器处理，是可将 DispatcherServlet 的请求映射配置为“/”的前提。由于 REST 是 Spring 的重要功能之一，所以 Spring 团队很看重静态资源处理这项任务，给出了堪称经典的两种解决方案。

在学习这两个方案之前，先调整 web.xml 中 DispatcherServlet 的配置，使其可以捕获所有的请求。

```
<servlet>
    <servlet-name>smart</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>smart</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

通过<url-pattern>/</url-pattern>的配置，所有 URL 请求都将被 Spring MVC 的 DispatcherServlet 截获。

1. 采用<mvc:default-servlet-handler/>

在 smart-servlet.xml 中配置<mvc:default-servlet-handler/>后，会在 Spring MVC 上下文中定义一个 org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler，它将充当一个检查员的角色，对进入 DispatcherServlet 的 URL 进行筛查。如果发现是静态资源的请求，就将该请求转由 Web 应用服务器默认的 Servlet 处理；如果不是静态资源的请求，则由 DispatcherServlet 继续处理。

一般 Web 应用服务器（包括 Tomcat、Jetty、Glassfish、JBoss、Resin、WebLogic

和 WebSphere）默认的 Servlet 名称都是 default，因此，DefaultServletHttpRequestHandler 可以找到它。如果用户所使用的 Web 应用服务器的默认 Servlet 名称不是 default，则需要通过 default-servlet-name 属性显式指定。

```
<mvc:default-servlet-handler default-servlet-name="yourServerDefaultServlet Name"/>
```

2．采用<mvc:resources/>

<mvc:default-servlet-handler/>将静态资源的处理经由 Spring MVC 框架交回 Web 应用服务器。而<mvc:resources/>更进一步，由 Spring MVC 框架自己处理静态资源，并添加一些有用的附加功能。

首先，<mvc:resources/>允许静态资源放置在任何地方，如 WEB-INF 目录下、类路径下等，甚至可以将 JavaScript 等静态文件打包到 JAR 包中。通过 location 属性指定静态资源的位置，由于 location 属性是 Resource 类型，因此可以使用诸如“classpath:”等的资源前缀指定资源位置。传统 Web 容器的静态资源只能放在 Web 容器的根路径下，<mvc:resources/>则完全打破了这个限制。

其次，<mvc:resources/>依据当前著名的 Page Speed、YSlow 等浏览器优化原则对静态资源提供优化。可以通过 cacheSeconds 属性指定静态资源在浏览器端的缓存时间，一般可将该时间设置为一年，以充分利用浏览器端的缓存。在输出静态资源时，会根据配置设置好响应报文头的 Expires 和 Cache-Control 值。

在接收到静态资源的获取请求时，会检查请求头的 Last-Modified 值。如果静态资源没有发生变化，则直接返回 303 响应状态码，指示客户端使用浏览器缓存的数据，而非将静态资源的内容输出到客户端，以充分节省带宽，提高程序性能。

在 smart-servlet.xml 中添加以下配置：

```
<mvc:resources mapping="/resources/**"location="/,classpath:/META-INF/publicResources/"/>
```

以上配置将 Web 根路径“/”及类路径/META-INF/publicResources/下的目录映射为/resources 路径。假设 Web 根路径下拥有 images 和 js 这两个资源目录，则可以通过如图 17-19 所示的方式引用静态资源。

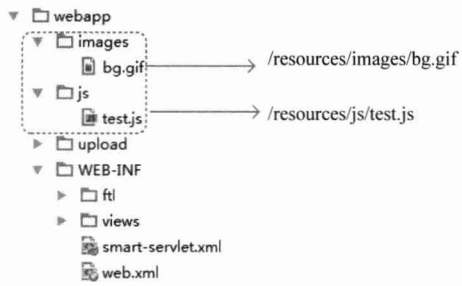


图 17-19　静态资源路径映射

假设类路径/META-INF/publicResources/下还拥有 images/bg1.gif 和 js/test1.js，则也可以在网页中通过/resources/images/bg1.gif 和/resources/js/test1.js 进行引用，如代码清单 17-54 所示。

代码清单 17-54　test.jsp：引用静态资源文件

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c"  uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>静态资源测试页面</title>
    <script src="<c:url value="/resources/js/test.js"/>" type="text/javascript">
</script>                                          通过逻辑映射路径引用资源文件
  </head>
  <body>
    <script>test();</script>
  </body>
</html>
```

由于<mvc:resources/>可以将多个物理路径映射为一个逻辑路径，因此，一个用逻辑路径表示的资源在多个物理路径下都存在。对于这个问题，<mvc:resources/>的处理机制是，只要在一个物理路径下找到匹配的资源后就返回，查找的顺序和物理路径在 location 中的配置顺序一致。

聪明的读者可能会问：既然将 Web 根路径“/”映射为“/resources/**”，是否可以在网页中通过“/resources/WEB-INF/web.xml”访问这个敏感的文件呢？答案是否定的。Spring MVC 在处理映射的静态资源时，会查看引用路径是否包含 WEB-INF 或 META-INF。如果包括，则直接返回 null 值，以保护安全文件不泄露出去。当然，如果将/WEB-INF/设置在 location 属性中，则可以通过/resources/web.xml 的 URL 查看到 web.xml。

```
<mvc:resources mapping="/resources/**" location="/WEB-INF/"/>
```

所以使用<mvc:resources/>时需要特别注意，不要一不小心将不期望暴露的资源泄露出去。

通过<mvc:resources/>的 cache-period 属性可以设置静态资源在客户端浏览器中的缓存有效时间。

```
<mvc:resources mapping="/resources/**"
   location="/,classpath:/META-INF/publicResources/"  cache-period="31536000"/>
```

一般情况下，将 cache-period 设置为一年，以便充分利用客户端的缓存数据。

在发布新版本的应用时，即使服务器端的 JavaScript、CSS 等静态资源文件已经发生了变化，但是由于客户端浏览器本身缓存管理机制的问题，客户端并不会从服务器端下载新的静态资源。一个好的解决办法是：网页中引用静态资源的路径添加应用的发布版本号，这样在发布新的部署版本时，由于版本号的变更造成网页中静态资源路径发生更改，从而使这些静态资源成为“新的资源”，客户端浏览器就会下载这个“新的资源”，而不会使用缓存中的数据。针对这个解决思路，可以通过<mvc:resources/>的静态资源逻辑路径给出一个通用的解决方案。

将发布版本号包含到<mvc:resources/>的静态资源逻辑路径中。首先创建一个 ServletContextAware 实现类，如代码清单 17-55 所示。

代码清单 17-55　ResourcePathExposer.java：上传文件

```
package com.smart.web;
import javax.servlet.ServletContext;
import org.springframework.web.context.ServletContextAware;
public class ResourcePathExposer implements ServletContextAware {
    private ServletContext servletContext;
    private String resourceRoot;
    public void init() {
        String version = "1.2.1";①
        resourceRoot = "/resources-" + version; ②

        getServletContext().setAttribute("resourceRoot", ③
                getServletContext().getContextPath()+resourceRoot);
    }
    public void setServletContext(ServletContext servletContext) {
        this.servletContext = servletContext;
    }

    public String getResourceRoot() {
        return resourceRoot;
    }

    public ServletContext getServletContext() {
        return servletContext;
    }
}
```

在实际应用中，可以在外部属性文件或数据库中保存应用的发布版本号，在此处获取之。此处仅仅提供了一个模拟值

资源逻辑路径带上应用的发布版本号

将资源逻辑路径暴露到 ServletContext 的属性列表中

在 ResourcePathExposer 中获取应用程序的发布版本号，产生一个带版本号的静态资源路径 resourceRoot，同时将其值发布到 ServletContext 中，这样 JSP 文件就可以通过 ${resourceRoot}引用其值了。

接下来要调整 smart-servlet.xml 中的配置，以便使用带版本的静态资源逻辑路径。

```
<bean id="rpe" class="com.smart.web.ResourcePathExposer" ①
    init-method="init"/>
<mvc:resources mapping="#{rpe.resourceRoot}/**" ②
          location="/" cache-period="31536000"/>
```

在①处配置好 ResourcePathExposer，并指定其初始化方法为 init()，以便在容器启动时让其初始化 resourceRoot 的值。由于其实现了 ServletContextAware 接口，因此，Spring 会在初始化该 Bean 时将 ServletContext 引用注入进来。

在②处通过 Spring EL 表达式引用 ResourcePathExposer 的 resourceRoot 属性值，生成动态的静态资源逻辑路径。

最后调整网页中引用静态资源的方式，如代码清单 17-56 所示。

代码清单 17-56　test.jsp：引用静态资源文件

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
  <head>
    <title>静态资源测试页面</title>
```

```
    <script src="${resourceRoot}/js/test.js" type="text/javascript"> </script>
  </head>                          ①
  <body>                                引用 ResourcePathExposer 通过 ServletContext
    <script>test();</script>            暴露的 resourceRoot 属性值
  </body>
</html>
```

由于①处引用的 resourceRoot 值和<mvc:resources/>通过#{rpe.resourceRoot}引用的值是一样的，所以可以正确访问到物理静态资源。这样，在每次发布新版本后，随着发布版本号的更改，客户端就会自动下载新的静态资源。

实战经验

要想成为一名 Web 开发高手，不能满足于知道如何做，而要抛开现象探究本质。笔者认为，要成为一名 Web 开发高手，必须熟悉以下内容：

（1）每次请求和响应的背后究竟发生了哪些步骤，客户端服务器是如何通过 HTTP 请求报文进行交互的。

（2）深刻掌握 MIME 类型的知识。

（3）深刻掌握 HTTP 响应状态码的知识，如 404、303 究竟代表什么。

17.8.2　装配拦截器

当收到请求时，DispatcherServlet 将请求交给处理器映射（HandlerMapping），让它找出对应该请求的 HandlerExecutionChain 对象。在讲解 HandlerMapping 之前，有必要认识一下这个 HandlerExecutionChain 对象。

HandlerExecutionChain 顾名思义是一个执行链，它包含一个处理该请求的处理器（Handler），同时包括若干个对该请求实施拦截的拦截器（HandlerInterceptor）。当 HandlerMapping 返回 HandlerExecutionChain 后，DispatcherServlet 将请求交给定义在 HandlerExecutionChain 中的拦截器和处理器一并处理。

HandlerExecutionChain 是负责处理请求并返回 ModelAndView 的处理执行链，其结构如图 17-20 所示。请求在被 Handler 执行的前后，链中装配的 HandlerInterceptor 会实施拦截操作。

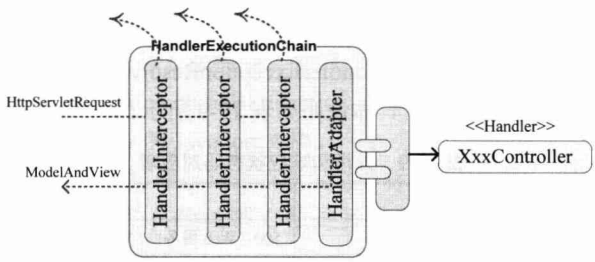


图 17-20　HandlerExecutionChain 的结构

拦截器到底做了什么事情？我们通过考查拦截器的几个接口方法进行了解。

- boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)：在请求到达 Handler 之前，先执行这个前置处理方法。当该方法返回 false 时，请求直接返回，不会传递到链中的下一个拦截器，更不会传递到处理器链末端的 Handler 中。只有返回 true 时，请求才向链中的下一个处理节点传递。
- void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)：在请求被 HandlerAdapter 执行后，执行这个后置处理方法。
- void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)：在响应已经被渲染后，执行该方法。

位于处理器链末端的是一个 Handler，DispatcherServlet 通过 HandlerAdapter 适配器对 Handler 进行封装，并按统一的适配器接口对 Handler 处理方法进行调用。

```
<mvc:interceptors>
   <mvc:interceptor>
      <mapping path="/secure/*"/>
      <bean class="com.smart.web.MyInterceptor" />
   </mvc:interceptor>
</mvc:interceptors>
```

可以在 smart-servlet.xml 中配置多个拦截器，每个拦截器都可以指定一个匹配的映射路径，以限制拦截器的作用范围。

17.8.3 异常处理

Spring MVC 通过 HandlerExceptionResolver 处理程序的异常，包括处理器映射、数据绑定及处理器执行时发生的异常。HandlerExceptionResolver 仅有一个接口方法。

```
ModelAndView resolveException(HttpServletRequest request,HttpServletResponse response,
                              Object handler,Exception ex)
```

当发生异常时，Spring MVC 将调用 resolveException()方法，并转到 ModelAndView 对应的视图中，作为一个异常报告页面反馈给用户。

HandlerExceptionResolver 拥有 4 个实现类，分别是 DefaultHandlerExceptionResolver、*SimpleMappingExceptionResolver*、*AnnotationMethodHandlerExceptionResolver* 及 *Response* StatusExceptionResolver。

1. DefaultHandlerExceptionResolver

Spring MVC 默认装配了 DefaultHandlerExceptionResolver，它会将 Spring MVC 框架的异常转换为相应的响应状态码，具体说明如表 17-9 所示。

表 17-9 异常和响应状态码对应表

异常类型	响应状态码
ConversionNotSupportedException	500（Web 服务内部错误）
HttpMediaTypeNotAcceptableException	406（无和请求 accept 匹配的 MIME 类型）

续表

异常类型	响应状态码
HttpMediaTypeNotSupportedException	415（不支持的 MIME 类型）
HttpMessageNotReadableException	400（坏的请求）
HttpMessageNotWritableException	500
HttpRequestMethodNotSupportedException	405（不支持的请求方法）
MissingServletRequestParameterException	400
NoSuchRequestHandlingMethodException	404（找不到匹配的资源）
TypeMismatchException	400

可以在 web.xml 中为响应状态码配置一个对应的页面，代码如下：

```
<error-page>
   <error-code>404</error-code>
   <location>/404.htm</location>
</error-page>
```

2. AnnotationMethodHandlerExceptionResolver

Spring MVC 默认注册了 AnnotationMethodHandlerExceptionResolver，它允许通过@ExceptionHandler 注解指定处理特定异常的方法，如代码清单 17-57 所示。

代码清单 17-57　UserController .java

```
package com.smart.web;
import org.springframework.web.bind.annotation.ExceptionHandler;

@Controller
@RequestMapping("/user")
public class UserController {
   @RequestMapping(path = "/throwException")①
    public String throwException() {
        if(2>1){
            throw new RuntimeException("ddd");
        }
        return "success";
    }

    @ExceptionHandler(RuntimeException.class)②
    public String handleException(RuntimeException re, HttpServletRequest request) {
        return "forward:/error.jsp";
    }
}
```

①处的处理方法在调用时抛出一个 RuntimeException 异常，它会被处于同一处理器类中②处的 handleException() 方法捕获。@ExceptionHandler 可以指定多个异常，如@ExceptionHandler({AException.class,BException.class})。异常处理方法的标签非常灵活，请参见 ExceptionHandler 的 Javadoc 文档。

不清楚是 Spring MVC 的 Bug 还是有意为之，标注@ExceptionHandler 的异常处理方法只能对同一处理类中的其他处理方法进行异常响应处理，笔者现在还受困于@ExceptionHandler 的这个限制中。

ResponseStatusExceptionResolver 和 ResponseStatusExceptionResolver 类似，允许通过@ResponseStatus 注解标注一个方法，用于处理特定类型的响应状态码。

3．SimpleMappingExceptionResolver

如果希望对所有异常进行统一处理，则可以使用 SimpleMappingExceptionResolver，它将异常类名映射为视图名，即发生异常时使用对应的视图报告异常。

请看下面的异常映射配置片段：

```
<bean id=" handlerExceptionResolver "
   class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
     <props>
        <prop key="org.springframework.dao.DataAccessException">dataAccessFailure
</prop>①
       <prop key="org.springframework.transaction.TransactionException"> dataAccessFailure
        </prop>②
     </props>
    </property>
</bean>
```

在①处指定当控制器发生 DataAccessException 异常时，使用 dataAccessFailure 视图显示。在②处指定当控制器发生 TransactionException 异常时，使用 dataAccessFailure 视图显示。

当然，用户也可以自己实现 ExceptionResolver 覆盖 resolveException()接口方法，编写自己的异常解析器，执行一些特定的工作，如将异常信息保存到数据库中等。编写好自定义的 ExceptionResolver，在 smart-servlet.xml 中注册成一个 Bean 即可工作。

17.8.4 RequestContextHolder 的使用

在 Spring API 中提供了一个非常便捷的工具类 RequestContextHolder，能够在 Controller 中获取 request 和 session 对象。其使用方法如下：

```
HttpServletRequest request = ((ServletRequestAttributes)
                RequestContextHolder.getRequestAttributes()).getRequest();
```

需要注意的是，如果直接使用该工具类，则会抛出一个空指针异常，原因是还需要在 web.xml 中配置一个监听器，代码如下：

```
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
     </listener-class>
</listener>
```

17.9 小结

Spring MVC 4.0 和早期版本相比有了一个质的飞跃，如更全面和方便地支持

REST 风格的 Web 编程、注解驱动、处理方法签名非常灵活、处理器方法不依赖于 Servlet API 等。

由于 Spring MVC 框架在后台做了很多隐性工作，所以想深入掌握 Spring MVC 4.0 并非易事。本章在学习 Spring MVC 的各项功能的同时，深入内部了解了其后台的运作机理。只有了解了这些运作机理，才能更好地使用这个先进的 MVC 框架。

第 18 章

实战案例开发

本章将带领大家开发一个完整的论坛应用案例，体会实际应用开发所需的各项技术及关注要点。学习完本案例后，读者即可胜任使用 Spring+Hibernate 经典框架开发实际应用的工作。

本章主要内容：

- 如何通过 UML 图描述应用的需求和设计
- 对于大型的 Web 应用，应该如何设计类和 Web 目录的结构
- 如何设计 Web 应用的持久层、服务层和 Web 层
- 如何测试 Web 应用的持久层、服务层和 Web 层

本章亮点：

- 如何描述 Web 应用需求和设计
- 如何对 Web 应用各分层实施单元测试

18.1 论坛案例概述

本章将通过一个具体的“小春论坛”案例开发来讲解使用 Spring+Hibernate 集成框架的开发过程。下面先来了解一下这个论坛的整体功能。

18.1.1 论坛整体功能结构

相信大多数读者都使用过网络论坛，论坛的内容主题可谓五花八门，但对于开发人员来讲，它们的功能却都是相近的。可以通过图 18-1 了解论坛的基本功能组成。

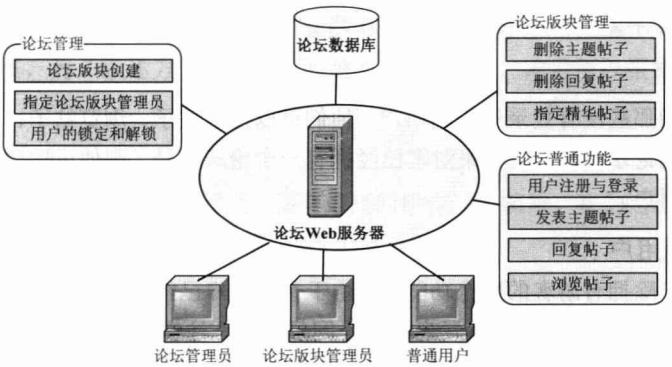


图 18-1　论坛整体功能结构

具体来讲，大部分论坛都包括以下功能模块。

- 论坛普通功能：包括用户注册与登录、发表主题帖子、回复帖子、浏览帖子等功能，这些功能是普通用户所拥有的。
- 论坛版块管理：包括删除主题帖子、删除回复帖子、指定精华帖子等功能，这些功能是论坛版块管理员及论坛管理员所拥有的。
- 论坛管理：包括论坛版块创建、指定论坛版块管理员、用户的锁定和解锁等功能，这些功能是论坛管理员所拥有的。

18.1.2　论坛用例描述

可以将论坛的用户角色划分为 4 种类型：游客、普通用户、论坛版块管理员及论坛管理员。这 4 种类型角色的操作权限是依次递增的，举例来说：所有普通用户拥有的操作功能，论坛版块管理员都拥有；而所有论坛版块管理员拥有的功能，论坛管理员也都拥有。通过如图 18-2 所示的系统用例图来描述这 4 个系统角色和用例的关系。

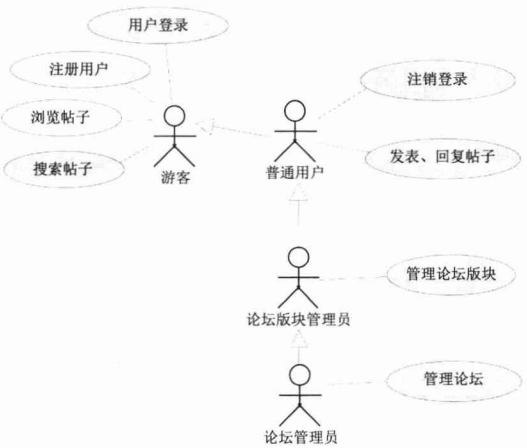


图 18-2　系统一级用例图

下面分别对这些角色及他们的操作功能进行说明。

1．游客

游客是指那些没有登录论坛的用户，他们可以搜索帖子、浏览帖子、调用论坛注册功能注册成为论坛的用户。如果游客已经拥有一个论坛账号，则他可以登录论坛，成为特定类型的用户。

2．普通用户

普通用户除拥有游客的所有功能外（由于普通用户已经登录，所以不能进行用户注册和用户登录操作），他还可以发表帖子、回复帖子、注销登录。当用户注销登录后，他就成为游客。普通用户的二级用例图如图 18-3 所示。

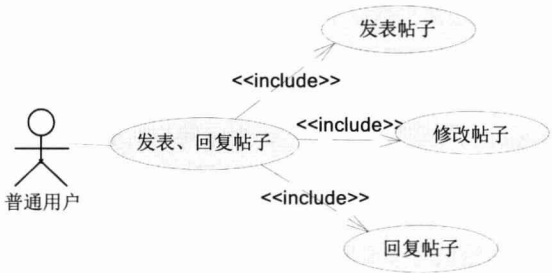


图 18-3　普通用户的二级用例图

3．论坛版块管理员

一个论坛一般拥有多个论坛版块，每个论坛版块可以拥有一个或多个论坛版块管理员。论坛版块管理员负责管理论坛版块的帖子。论坛版块管理员的二级用例图如图 18-4 所示。

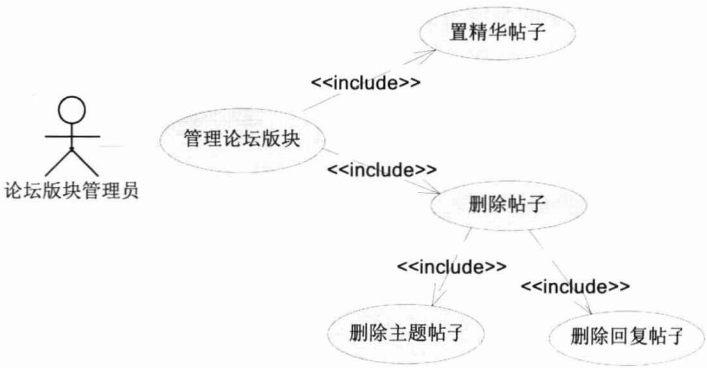


图 18-4　论坛版块管理员的二级用例图

在这里，我们要明确两个概念：主题帖子和回复帖子，前者指用户在论坛版块中发表的一个话题所对应的内容帖，一个话题可以拥有多个跟帖，后者就是回复帖子。所以

删除帖子包括删除主题帖子和删除回复帖子两个功能。置精华帖子的操作对象是主题帖子而非回复帖子，为了叙述方便，我们直接称之为“帖子”，读者可以根据上下文确定具体所指。

4．论坛管理员

该角色用户是整个论坛的管理员，拥有最高的操作功能权限。我们通过以下二级用例图描述论坛管理员的操作功能，如图 18-5 所示。

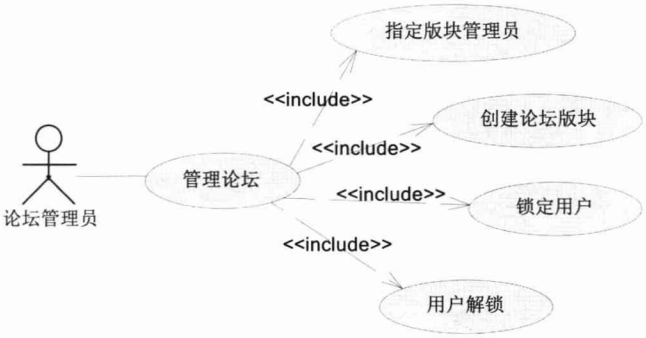


图 18-5　论坛管理员的二级用例图

论坛管理员可以创建一个论坛版块，为论坛版块指定若干个管理员，还可以锁定某些不遵守规则的用户，当然也可以对已经锁定的用户进行解锁。

18.1.3　主要功能流程描述

本小节将对论坛的主要功能进行描述，为页面设计和程序设计提供依据。

1．用户登录

和第 2 章所讲解的用户登录不同，本案例的用户登录功能涉及的步骤比较复杂，它更接近于真实应用的用户登录功能。可以通过如图 18-6 所示的活动图对用户登录进行具体描述。

用户登录需要判断许多种可能的情况，如用户名不正确、用户密码不正确、用户已经被锁定不允许登录等。在通过以上所有检查后，用户登录才算成功，这时需要完成登录成功后的业务操作，如添加用户积分、记录用户登录日志、更新用户最后登录时间等。为了简化实例功能，这里仅进行添加用户积分的操作。操作完成后，需要将用户对象添加到 HTTP Session 中，以便后续操作可以直接从 Session 中获取用户信息。

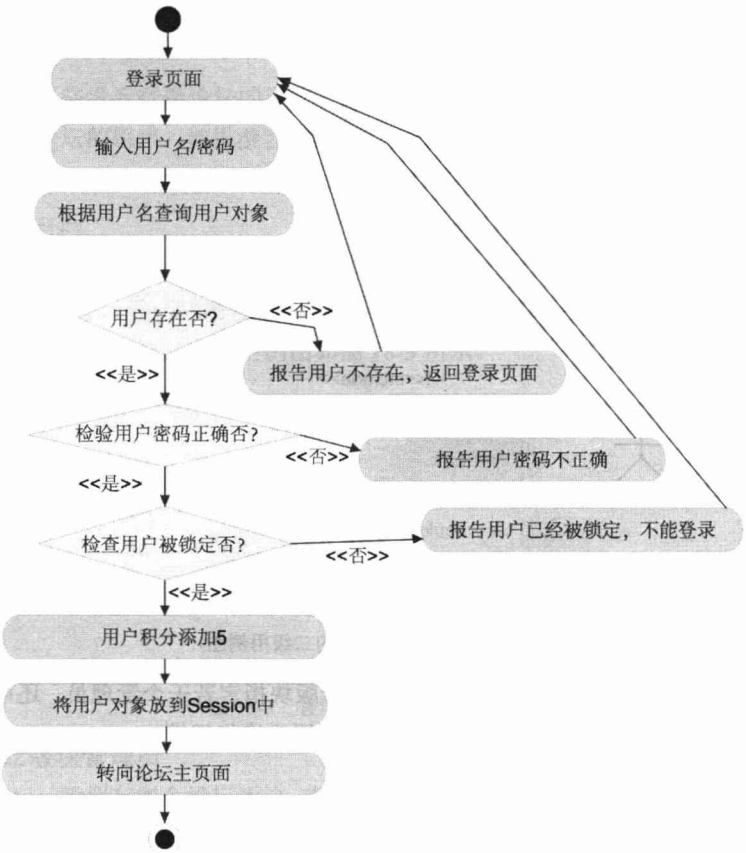


图 18-6　论坛用户登录

2．发表主题帖子

发表主题帖子功能的整体流程如图 18-7 所示。

在提交并成功保存主题帖子后，需要进行一些相关的后续操作，如将论坛版块的帖子数加 1，给主题帖子作者添加 10 个积分，然后刷新论坛版块帖子列表。

3．回复主题帖

回复一个主题帖子即新建一个回复帖子，这个功能可以通过如图 18-8 所示的活动图进行描述。

和发表主题帖子类似，回复帖子保存成功后，还必须进行一些相关的操作，如用户积分数加 5、主题帖子的回复数加 1，并更新主题帖子的最后回复时间，以便这个刚被回复过的主题帖子能够排到主题帖子列表的最前面。

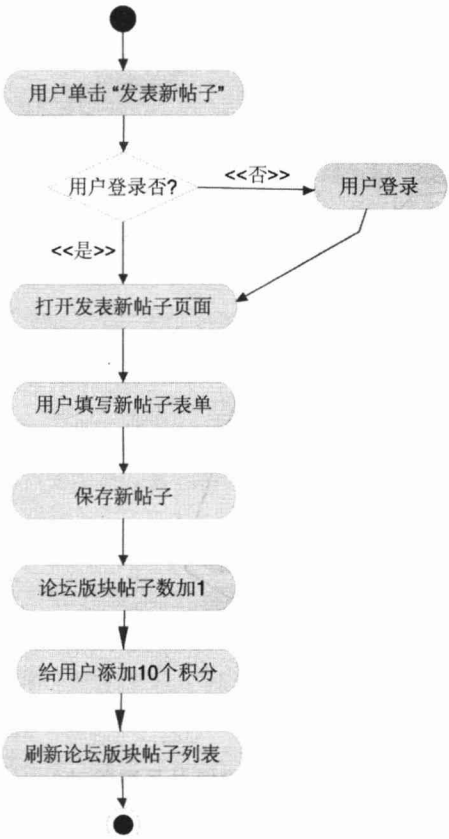


图 18-7　发表主题帖子

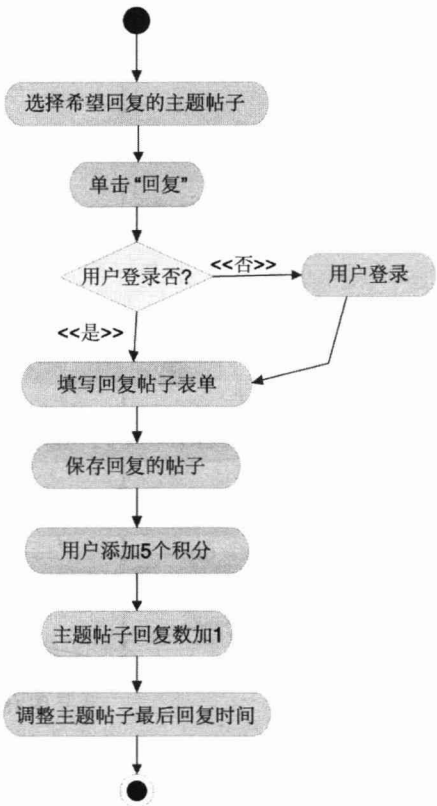


图 18-8　回复主题帖子

4．删除帖子

论坛版块管理员、论坛管理员都可以删除帖子，包括删除主题帖子和删除回复帖子。可以通过如图 18-9 所示的活动图对删除帖子的整体流程进行描述。

删除帖子基本上是执行发表主题帖子和回复主题帖子的反过程，不过为了防止论坛用户恣意发表一些不合法或垃圾性质的帖子，提高论坛内容的整体质量，被删除帖子的作者需要惩罚性地扣除较多的积分，如回复帖子被删除时将被扣除 20 个积分，而主题帖子被删除时将被扣除 50 个积分。

5．置精华帖子

论坛管理员、论坛版块管理员可以将一些质量较高的主题帖子置为精华帖子，方便其他用户快速查看到这些帖子，这一功能的操作流程如图 18-10 所示。

当主题帖子被设置为精华帖子时，发表该帖子的作者将被追加 100 个积分，以示奖赏。

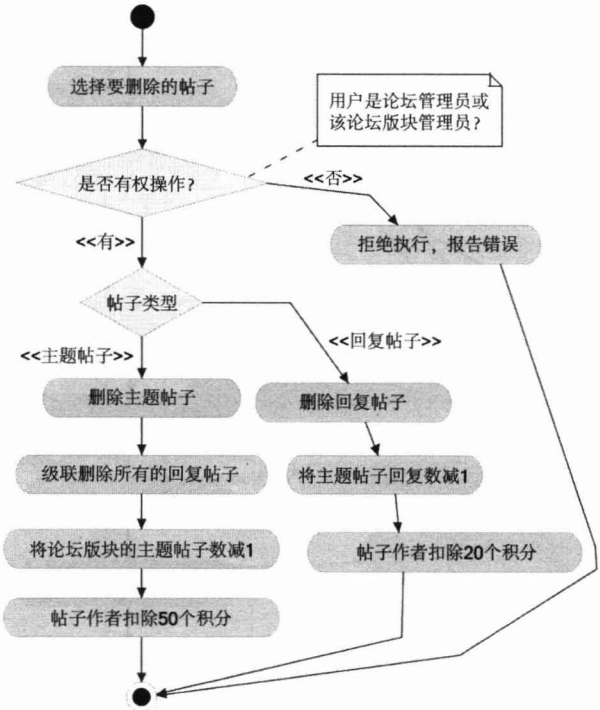


图 18-9　删除帖子

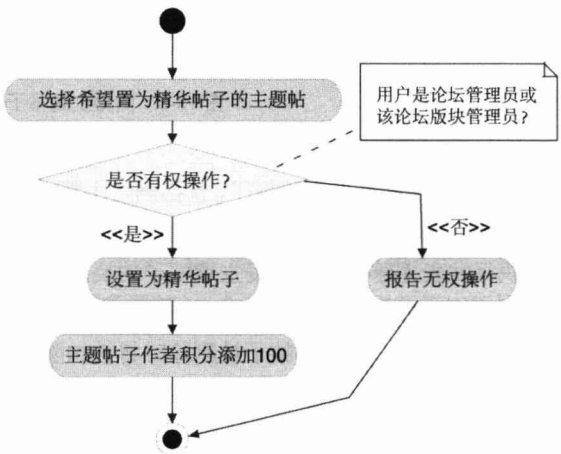


图 18-10　置为精华帖子

6．指定论坛版块管理员

指定论坛版块管理员的权限由论坛管理员完成，用户可以被指定为若干个论坛版块的管理员，一个论坛版块也可以拥有多个管理员。指定论坛版块管理员功能的操作流程如图 18-11 所示。

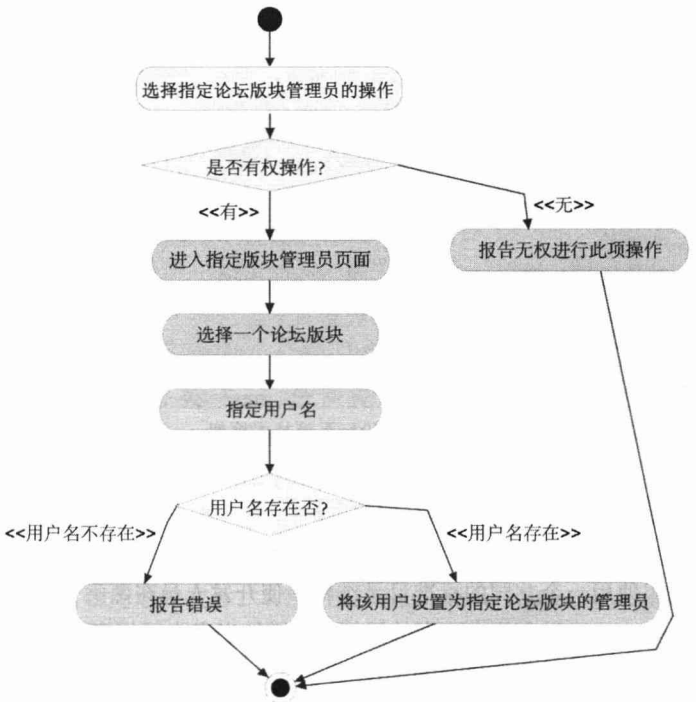


图 18-11　指定论坛版块管理员

论坛管理员从列表中选择一个论坛版块，然后输入希望成为该论坛版块管理员的用户名，系统必须判断这个用户名是否存在，如果不存在则必须报告错误，以便论坛管理员调整用户名。

18.2　系统设计

18.2.1　技术框架选择

我们拟使用如图 18-12 所示的技术框架来完成论坛应用程序的开发。为了解决中文乱码问题，我们在 Web 层提供一个字符编码转换过滤器。Web 层使用 Spring MVC 进行请求的处理和响应，视图采用 JSP 2.0 和 JSTL 技术处理。

服务层采用 Spring 4.0，持久层采用 Hibernate 4.2.0，而持久层通过 Spring 提供的支持类集成到 Spring 中。系统严格采取 Web 层、服务层和持久层三层体系结构，上一层的程序可以调用下一层的程序，反之则不行，达到层与层之间松耦合的目的。

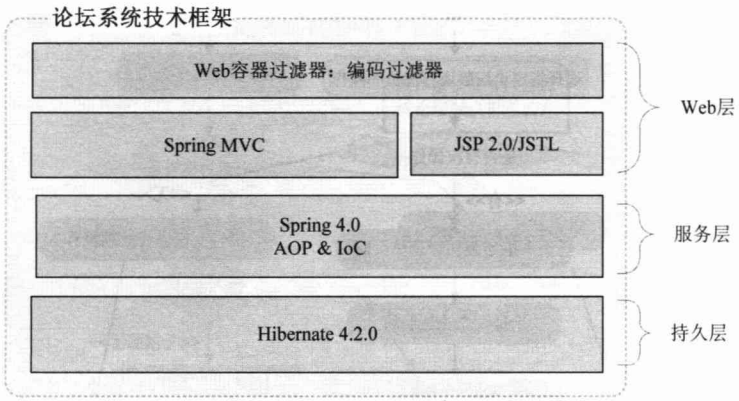


图 18-12　论坛系统技术框架

18.2.2　采用 Maven 构建项目

Maven 提倡使用一个共同的标准目录结构，使开发人员在熟悉了一个 Maven 工程后，对其他的 Maven 工程也能有清晰的了解，这样做也省去了很多设置上的麻烦。本工程使用 Maven 推荐的标准目录结构，如图 18-13 所示。

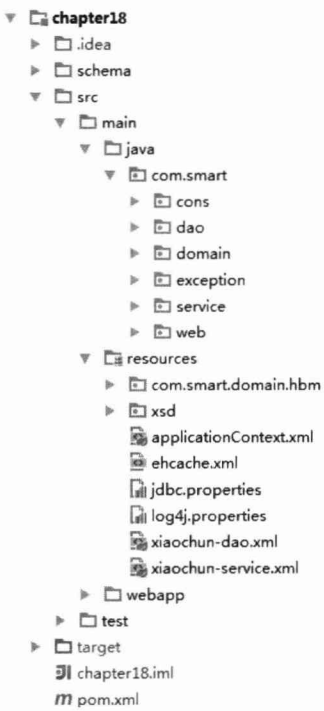


图 18-13　源码包目录结构

所有的源代码文件位于 src/main 文件夹中，在 main 文件夹中规划了 3 个子文件夹，其中 resources 文件夹用于放置系统配置文件，java 文件夹用于放置 Java 源代码文件，webapp 文件夹用于放置 web 应用的程序文件。所有类位于 com.smart 包中，该类包下为每个分层提供了一个相应的类包，如 dao 对应持久层的程序，而 service 和 web 分别对应服务层和 Web 层的程序。由于 PO 会在多个层中出现，因此我们为其提供了一个单独的 domain 包。为了避免在程序中直接使用字面值常量，需要通过常量定义的方式予以规避，我们在 cons 包中定义了应用级的常量。为了统一管理应用系统异常体系，我们在 exception 包中定义了业务异常类及系统异常等。可以在分层包下再按功能模块定义子包，由于我们的论坛应用比较简单，所以每个分层包下不再设子包。

我们为 DAO 和服务类 Bean 分别提供了一个 Spring 配置文件，前者为 xiaochun-dao.xml，后者为 xiaochun-service.xml。jdbc.properties 属性文件提供了数据库连接的信息，这个属性文件将被 xiaochun-service.xml 使用。log4j.properties 属性文件是 Log4J 的配置文件。我们将这些配置文件直接放置在类路径下。

webapp 目录结构很简单，我们将大部分的 JSP 放置在 WEB-INF/jsp 目录中，防止用户直接通过 URL 调用这些文件。WEB-INF/xiaochun-servlet.xml 为 Spring MVC 的配置文件。如果项目的 JSP 文件数目很多，则可以在 WEB-INF/jsp 目录下按功能模块划分为多个子文件夹。一般的 Web 应用都会在 Web 根目录下创建 images、css、js 等文件夹，分别放置图片、CSS 及 JS 的资源文件。我们的论坛应用比较简单，没有这些资源，所以这些文件夹没有出现在目录结构中。

18.2.3　单元测试类包结构规划

规划好程序的类包结构之后，需要根据应用程序分层结构规划相应的单元测试结构。为了使单元测试模块清晰、可读，一般情况下，可以根据应用程序分层建立相应的单元测试目录结构。在 Maven 目录规范中，test 和 main 为同级目录，如图 18-14 所示。

图 18-14　单元测试类包结构

与应用程序代码结构一样，测试所用的源代码文件使用专门的文件夹进行管理，所

有与单元测试相关的文件都放置在 src/test 文件夹中，在 test 文件夹中规划了两个子文件夹，其中 resources 文件夹用于放置测试资源文件，java 文件夹用于放置测试 Java 源代码文件。所有的测试类位于 com.smart 包中，该类包下为每个分层提供了一个相应的类包，如 dao 对应持久层的测试代码，而 service 和 web 分别对应服务层和 Web 层的程序测试代码。

18.2.4 系统架构图

可以将论坛划分为 4 个功能模块，分别是用户管理、论坛管理、版块管理及论坛基础功能，如图 18-15 所示。

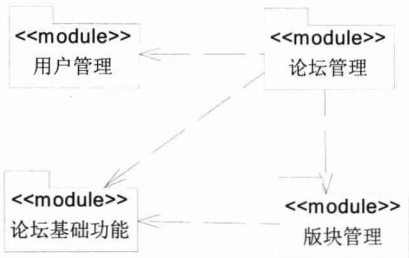


图 18-15 系统架构图

用户管理模块包括用户注册、登录、注销、用户个人信息维护、密码更改等功能（有些功能本案例未实现）；而论坛管理模块包括论坛版块创建、论坛版块管理员指定、用户锁定/解锁等功能；版块管理模块包括主题帖子删除、回复帖子删除、置精华帖子等功能；论坛基础功能模块则包括帖子搜索、论坛版块列表、论坛版块主题帖子列表、帖子浏览、发表主题帖子、发表回复帖子等基础性论坛功能。

18.2.5 PO 类设计

论坛应用共有 7 个 PO 类，其关系通过类图说明，如图 18-16 所示。

BaseDomain 是所有 PO 类的基类，它实现了 Serializable 接口。所有的 PO 类分别介绍如下。

- Board：论坛版块 PO 类。
- Topic：论坛主题 PO 类，包含主题帖子的作者、所属论坛版块、创建时间、浏览数、回复数等信息，其中 mainPost 对应主题帖子。
- Post：帖子 PO 类，一个 Topic 拥有一个 MainPost（主题帖子），但拥有若干个 Post（回复帖子）。
- User：论坛用户 PO 类。
- LoginLog：论坛用户登录日志 PO 类。

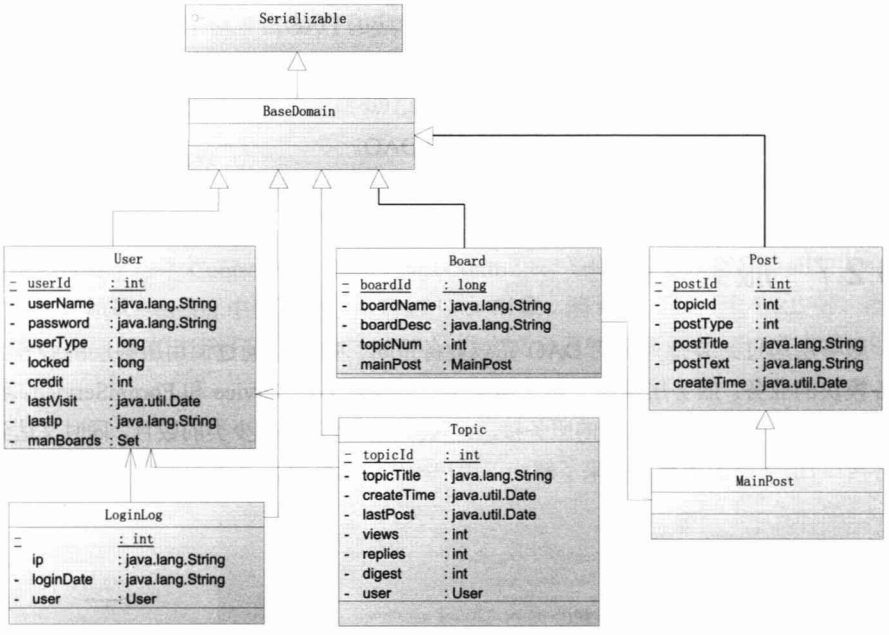


图 18-16　PO 类图

18.2.6　持久层设计

持久层采用 Hibernate 技术，创建所有 DAO 的基类 BaseDao<T>，并注入 Spring 为 Hibernate 提供的 HibernateTemplate 模板类。BaseDao<T>提供了常见数据的操作方法，子类仅需定义那些个性化的数据操作方法即可。BaseDao<T>使用了 JDK 5.0 泛型的技术，T 为 DAO 操作的 PO 类类型，子类在继承 BaseDao<T>时仅需指定 T 的类型，BaseDao<T>中的方法就可以确定操作的 PO 类型。持久层的类图如图 18-17 所示。

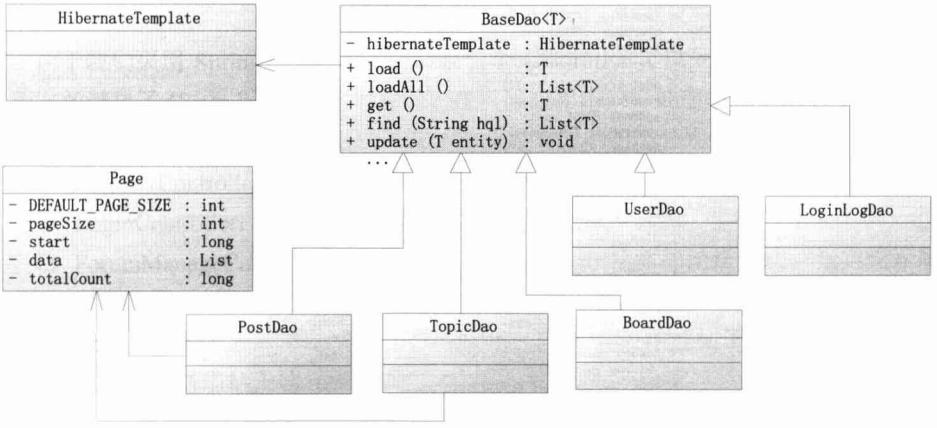


图 18-17　持久层的类图

- BoardDao：论坛版块 Board 持久化类对应的 DAO。
- TopicDao：主题 Topic 持久化类对应的 DAO。
- PostDao：帖子 Post 持久化类对应的 DAO。
- UserDao：用户 User 持久化类对应的 DAO。
- LoginLogDao：用户登录日志 LoginLog 持久化类对应的 DAO。

18.2.7 服务层设计

服务层通过封装持久层的 DAO 完成业务逻辑，Web 层通过调用服务层的服务类完成各模块的业务。服务层提供了两个服务类，分别是 UserService 和 ForumService。我们直接使用服务类，不提供相应的服务接口，这样可以有效地减少类的数目，同时又达到了服务接口+服务类的效果。先来了解一下用户操作的服务类，其类图如图 18-18 所示。

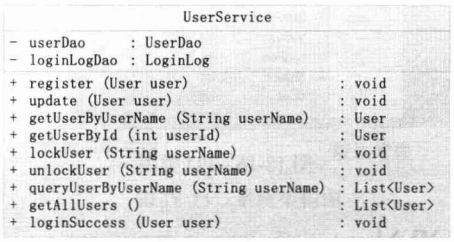


图 18-18 用户操作的服务类

UserService 通过调用持久层的 UserDao 操作持久化对象，它提供了保存、更新、锁定、解锁等对 User 持久类的操作方法，同时提供了根据用户名或者用户 ID 查询单个用户及根据用户名模糊查询多个用户的方法。

操作论坛版块、主题、帖子等论坛功能使用的服务方法封装在 ForumService 中，图 18-19 即 ForumService 的类图。

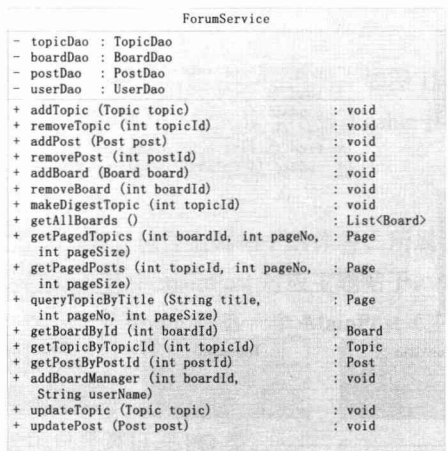


图 18-19 论坛操作功能服务类

ForumService 类中联合使用了 TopicDao、UserDao、BoardDao 及 PostDao 这 4 个 DAO 类，利用这些 DAO 类共同完成论坛功能的各项业务操作。这些功能包括论坛管理功能、论坛版块管理功能及发表主题帖子、回复帖子等。

18.2.8　Web 层设计

我们定义了一个 Controller 的基类：BaseController，它提供了其他 Controller 共有的一些方法，如从 Session 中获取登录用户的 User 对象、将请求转向一个 URL 等。所有具体的 Controller 都继承于这个 BasController，并定义自己的请求方法，其类图如图 18-20 所示。

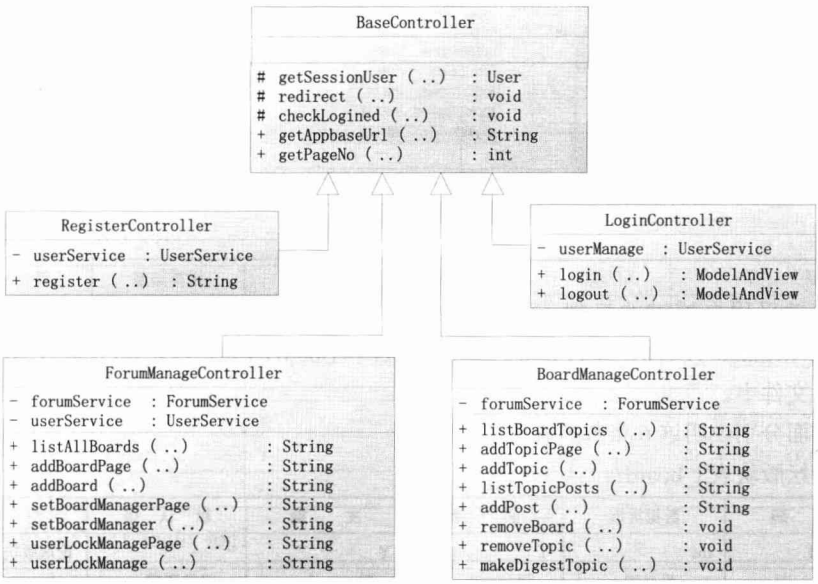


图 18-20　Web 层类图

由于我们采用 Spring 注解 MVC，所以一个 Controller 可以处理多种不同的请求，从而有效避免了 Controller 类数量的膨胀。在实际应用中，可处理多种请求的 Controller 比仅处理一种请求的 Controller 更受青睐。下面对类图中的类分别进行说明。

- RegisterController：用户注册的控制器。
- LoginController：用户登录、登录注销的控制器。
- ForumManageController：论坛管理的控制器，包括添加论坛版块、指定论坛版块管理员、对用户进行锁定/解锁。
- BoardManageController：论坛的基本功能，包括发表主题帖子、回复帖子、删除帖子、置精华帖子等。

18.2.9 数据库设计

论坛应用共包括 6 张数据表，其中 t_board_manager 用于维护 t_board 和 t_user 的多对多关系。表结构如图 18-21 所示。

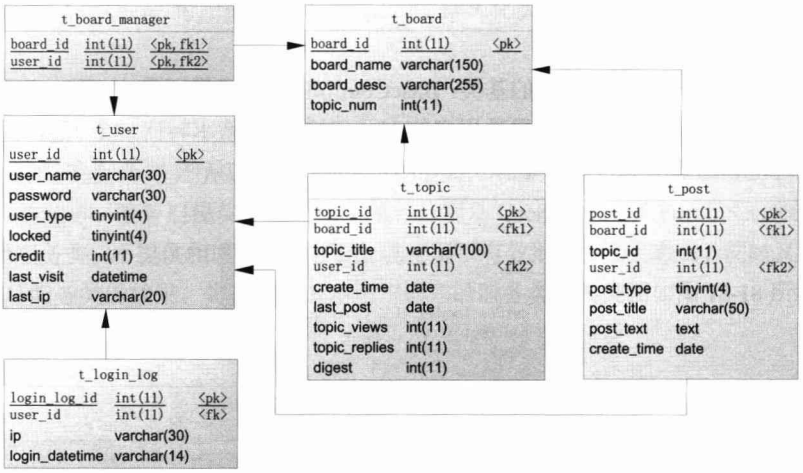


图 18-21 表结构图

主键采用自增键的机制，不采用处键。这些表都可以找到对应的 PO 类，但 t_board_manager 没有对应的 PO 类，它对应 User 和 Board 的多对多关系，反映在 Hibernate 的映射文件中。

下面分别说明这 6 张数据表的字段。

论坛版块表 t_board：

代　码	数据类型	必　填	主　键	默 认 值	注　释
board_id	int	Y	Y		论坛版块 ID
board_name	varchar(150)	Y	N	''	论坛版块名
board_desc	varchar(255)	N	N	NULL	论坛版块描述
topic_num	int	Y	N	0	帖子数目

用户管理版块关联表 t_board_manager：

代　码	数据类型	必　填	主　键	默 认 值	注　释
board_id	int	Y	Y		
user_id	int	Y	Y		

话题表 t_topic：

代　码	数据类型	必　填	主　键	默 认 值	注　释
topic_id	int	Y	Y		帖子 ID
board_id	int	Y	N		所属论坛
topic_title	varchar(100)	Y	N	''	帖子标题，该列建立索引
user_id	int	Y	N	0	发表用户，该列建立索引

续表

代　码	数据类型	必　填	主　键	默 认 值	注　释
create_time	datetime	Y	N		发表时间
last_post	datetime	Y	N		最后回复时间
topic_views	int	Y	N	1	浏览数
topic_replies	int	Y	N	0	回复数
digest	int	Y	N		0：不是精华话题；1：是精华话题

帖子表 t_post：

代　码	数据类型	必　填	主　键	默 认 值	注　释
post_id	int	Y	Y		帖子 ID
board_id	int	Y	N	0	论坛 ID
topic_id	int	Y	N	0	话题 ID，该列建立索引
user_id	int	Y	N	0	发表者 ID
post_type	tinyint	Y	N	2	帖子类型。1：主题帖子；2：回复帖子
post_title	varchar(50)	Y	N		帖子标题
post_text	text	Y	N		帖子内容
create_time	datetime	Y	N		创建时间

论坛用户表 t_user：

代　码	数据类型	必　填	主　键	默认值	注　释
user_id	int	Y	Y		用户 ID
user_name	varchar(30)	Y	N		用户名，该列建立索引
password	varchar(30)	Y	N	''	密码
user_type	tinyint	Y	N	1	1：普通用户；2：管理员
locked	tinyint	Y	N	0	0：未锁定；1：锁定
credit	int	N	N		积分

登录日志表 t_login_log：

代　码	数据类型	必　填	主　键	默认值	注　释
login_log_id	int	Y	Y		日志 ID
user_id	int	Y	N	0	发表者 ID
ip	varchar(30)	Y	N		登录 IP
login_datetime	datetime	Y	N		登录时间

18.3　开发前的准备

① 通过 mysql -uroot -p123456 登录 MySQL 数据库，运行 source <项目地址>/schema/sampledb.sql 脚本创建论坛数据库。该脚本还同时初始化了两个用户：一个是 john/1234（普通用户），另一个是 tom/123456（系统管理员）。

② 使用 IDEA 建立一个名为 chapter18 的 Maven 模块，使用 Java 1.8 版本。

③ 在 pom.xml 中增加依赖配置。pom.xml 文件可从本书配套网盘中的对应章节获取。

18.4 持久层开发

一般来说，我们将 PO 和 DAO 的类统一划归到持久层中，持久层既负责将 PO 持久化到数据中，也负责从数据库中加载数据到 PO 对象中。

18.4.1 PO 类

所有的 PO 类都直接或间接地继承 BaseDomain 类，这个 PO 基类的代码如代码清单 18-1 所示。

代码清单 18-1 BaseDomain.java

```
package com.smart.domain;
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;

//① 实现了Serializable接口，以便JVM可以序列化PO实例
public class BaseDomain implements Serializable{

   //② 统一的toString()方法
   public String toString() {
      return ToStringBuilder.reflectionToString(this);
   }
}
```

一般情况下，PO 类最好都实现 Serializable 接口，这样 JVM 就能够方便地将 PO 实例序列化到硬盘中，或者通过流的方式进行发送，为缓存、集群等功能带来便利。我们往往需要将 PO 对象打印为一个字符串，这是由对象的 toString()方法来完成的，这里通过 Apache 的 ToStringBuilder 工具类提供统一的实现。

下面先看一下 Board PO 类及 Hibernate JPA 注解配置，如代码清单 18-2 所示。

代码清单 18-2 Board.java

```
package com.smart.domain;
...
@Entity
@Cache(usage = CacheConcurrencyStrategy. NONSTRICT_READ_WRITE)
@Table(name = "t_board")
public class Board extends BaseDomain {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "board_id")
    private int boardId;
```

```
    @Column(name = "board_name")
    private String boardName;

    @Column(name = "board_desc")
    private String boardDesc;

    @Column(name = "topic_num")
    private int topicNum ;

   //省略属性的get/setter方法
}
```

每个持久化 PO 类都是一个实体 Bean，通过在类的定义中使用@Entity 注解来进行声明；通过@Table 注解为 Board 指定对应数据库表、目录和 schema 的名字；通过@Cache 注解为 Board 设置缓存策略。Hibernate 提供了以下几种缓存策略。

- CacheConcurrencyStrategy.NONE：不使用缓存。
- CacheConcurrencyStrategy.READ_ONLY：只读模式，在此模式下，如果对数据进行更新操作，则会产生异常。
- CacheConcurrencyStrategy.READ_WRITE：读/写模式，该模式在更新缓存的时候会对缓存里面的数据加锁，其他事务如果去取相应的缓存数据，发现被锁了，就会直接去数据库查询。
- CacheConcurrencyStrategy.NONSTRICT_READ_WRITE：不严格的读/写模式，使用该模式不会对缓存数据加锁。
- CacheConcurrencyStrategy.TRANSACTIONAL：事务模式，指缓存支持事务，当事务回滚时，缓存也能回滚，目前只支持 JTA 环境。

通过@Id 注解可将 Board 中的 boardId 属性定义为主键，使用@GenerateValue 注解定义主键生成策略（分别是 AUTO、TABLE、IDENTITY、SEQUENCE）。通过@Column 注解将 Board 的各个属性映射到数据库表 t_board 中相应的列。

下面再来看一下 Post PO 类及 Hibernate JPA 注解配置，如代码清单 18-3 所示。

代码清单 18-3　Post.java

```
package com.smart.domain;
...
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Table(name = "t_post")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "post_type", discriminatorType =
DiscriminatorType.STRING)
@DiscriminatorValue("1")
public class Post extends BaseDomain {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "post_id")
    private int postId;
```

```
    @Column(name = "post_title")
    private String postTitle;

    @Column(name = "post_text")
    private String postText;

    @Column(name = "board_id")
    private int boardId;

    @Column(name = "create_time")
    private Date createTime;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

   @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
   @JoinColumn(name="topic_id")
    private Topic topic;

   //省略属性的get/setter方法
}
```

Post（回复的帖子）和其子类 MainPost（主题帖）都映射到 t_post 表中，t_post 表通过 post_type 字段值分别两者。当 post_type=1 时，对应 MainPost；当 post_type=2 时，对应 Post。

通过@Inheritance 注解指定了 PO 映射继承关系。Hibernate 共提供了 3 种方式：每个类一张表（InheritanceType.TABLE_PER_CLASS）、连接的子类（InheritanceType.JOINED）和每个类层次结构一张表（InheritanceType.SINGLE_TABLE）。通过@DiscriminatorColumn 注解定义了辨别符列。对于继承层次结构中的每个类，@DiscriminatorValue 注解指定了用来辨别该类的值。辨别符列名字默认为 DTYPE，其默认值为实体名，类型为 DiscriminatorType.STRING。

通过@ManyToOne 注解定义了多对一关系，通过@JoinColumn 注解定义了多对一的关联关系。如果没有@JoinColumn 注解，则系统自动处理，在主表中将创建连接列，列名为“主题的关联属性名 + 下画线 + 被关联端的主键列名”。

其他的 PO 类和 Board、Post 类似，都是一堆属性的集合，并通过 JPA 注解配置 PO 类与数据表的映射关系，这里就不一一列出阐述。

18.4.2 DAO 基类

1. DAO 基类的基本方法

在编写完 PO 类及相应的 hbm 映射文件后，我们着手编写负责持久化 PO 和查找 PO 的 DAO 类。由于每个 PO 的 DAO 类都需要执行一些相同的操作，如保存、更新、删除 PO 及根据 ID 加载 PO 等，所以可以编写一个提供这些通用操作的基类，让所有 PO 的 DAO 类都继承这个 DAO 基类，其代码如代码清单 18-4 所示。

代码清单 18-4　BaseDao.java

```
package com.smart.dao;
import org.hibernate.Query;
...
// DAO基类，其他DAO可以直接继承这个DAO，不但可以复用共用的方法，还可以获得泛型的好处
public class BaseDao<T>{
    private Class<T> entityClass;
    @Autowired
    private HibernateTemplate hibernateTemplate;

     //通过反射获取子类确定的泛型类
    public BaseDao() {
        Type genType = getClass().getGenericSuperclass();
        Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
        entityClass = (Class) params[0];
    }

    //根据ID加载PO实例
    public T load(Serializable id) {
        return (T) getHibernateTemplate().load(entityClass, id);
    }

    //根据ID获取PO实例
    public T get(Serializable id) {
        return (T) getHibernateTemplate().get(entityClass, id);
    }

    //获取PO的所有对象
    public List<T> loadAll() {
        return getHibernateTemplate().loadAll(entityClass);
    }

    //保存PO
    public void save(T entity) {
        getHibernateTemplate().save(entity);
    }

    //删除PO
    public void remove(T entity) {
        getHibernateTemplate().delete(entity);
    }

    //更改PO
    public void update(T entity) {
        getHibernateTemplate().update(entity);
    }

    //执行HQL查询
    public List find(String hql) {
        return this.getHibernateTemplate().find(hql);
    }

    //执行带参的HQL查询
    public List find(String hql, Object... params) {
```

```
        return this.getHibernateTemplate().find(hql,params);
    }
    //对延迟加载的实体PO进行初始化
    public void initialize(Object entity) {
        this.getHibernateTemplate().initialize(entity);
    }
    ...
}
```

基类直接注入 Spring 为 Hibernate 提供的 HibernateTemplate 模板操作类，这样就可以借由这个 HibernateTemplate 模板操作类执行 Hibernate 的各项操作了。读者可能已经注意到基类的类名（BaseDao<T>）使用了 JDK 5.0 的泛型技术，这是为了让 DAO 子类可以使用泛型技术绑定特定类型的 PO 类，避免强制类型转换带来的麻烦。通过扩展这个 DAO 基类，DAO 子类仅需要声明泛型对应的 PO 类并实现那些非通用性的方法即可，大大减少了 DAO 子类的代码量。

2. 对数据分页的支持

除此以外，我们还在 BaseDao 中提供了对数据分页的支持。代码清单 18-5 列出了 BaseDao 中和数据分页相关的一些方法。

代码清单 18-5　BaseDao.java

```
package com.smart.dao;
import java.util.*;
import org.hibernate.Query;
...
public class BaseDao<T>{
    ...
    //分页查询函数，使用HQL
    public Page pagedQuery(String hql, int pageNo, int pageSize, Object... values) {
        Assert.hasText(hql);
        Assert.isTrue(pageNo >= 1, "pageNo should start from 1");
        // Count查询
        String countQueryString = " select count (*) " + removeSelect(removeOrders(hql));
        List countlist = getHibernateTemplate().find(countQueryString, values);
        long totalCount = (Long) countlist.get(0);

        if (totalCount < 1)
            return new Page();
        // 实际查询返回分页对象
        int startIndex = Page.getStartOfPage(pageNo, pageSize);
        Query query = createQuery(hql, values);
        List list = query.setFirstResult(startIndex).setMaxResults(pageSize).list();
        return new Page(startIndex, totalCount, pageSize, list);
    }

    //创建Query对象
    public Query createQuery(String hql, Object... values) {
        Assert.hasText(hql);
        Query query = getSession().createQuery(hql);
        for (int i = 0; i < values.length; i++) {
            query.setParameter(i, values[i]);
        }
```

```
        return query;
    }

    //去除HQL的select子句
    private static String removeSelect(String hql) {
        Assert.hasText(hql);
        int beginPos = hql.toLowerCase().indexOf("from");
        Assert.isTrue(beginPos != -1, " hql : " + hql + " must has a keyword 'from'");
        return hql.substring(beginPos);
    }

    //去除HQL的orderby子句
  private static String removeOrders(String hql) {
        Assert.hasText(hql);
        Pattern p = Pattern.compile("order\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_
INSENSITIVE);
        Matcher m = p.matcher(hql);
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            m.appendReplacement(sb, "");
        }
        m.appendTail(sb);
        return sb.toString();
    }
    …
}
```

这样，仅需提供 HQL 及分页的一些配置信息，就可以获取特定页面的数据。特定页面的信息通过 Page 类进行表达。下面来看一下 Page 类的代码，如代码清单 18-6 所示。

代码清单 18-6　Page.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
…
//分页对象，包含当前页数据及分页信息，如总记录数
public class Page implements Serializable {
    private static int DEFAULT_PAGE_SIZE = 20;
    private int pageSize = DEFAULT_PAGE_SIZE; // 每页的记录数
    private long start; // 当前页第一条数据在List中的位置，从0开始
    private List data; // 当前页中存放的记录，类型一般为List
    private long totalCount; // 总记录数

    //构造方法，只构造空页
    public Page() {
        this(0, 0, DEFAULT_PAGE_SIZE, new ArrayList());
    }

    //默认的构造方法
    public Page(long start, long totalSize, int pageSize, List data) {
        this.pageSize = pageSize;
        this.start = start;
        this.totalCount = totalSize;
        this.data = data;
    }
```

```
    //取总页数
    public long getTotalPageCount() {
        if (totalCount % pageSize == 0)
            return totalCount / pageSize;
        else
            return totalCount / pageSize + 1;
    }

    //取该页当前页码，页码从1开始
    public long getCurrentPageNo() {
        return start / pageSize + 1;
    }

    //该页是否有下一页
    public boolean isHasNextPage() {
        return this.getCurrentPageNo() < this.getTotalPageCount();
    }

    //该页是否有上一页
    public boolean isHasPreviousPage() {
        return this.getCurrentPageNo() > 1;
    }

    //获取任意一页第一条数据在数据集中的位置，每页条数使用默认值
    protected static int getStartOfPage(int pageNo) {
        return getStartOfPage(pageNo, DEFAULT_PAGE_SIZE);
    }

    //获取任意一页第一条数据在数据集中的位置
    public static int getStartOfPage(int pageNo, int pageSize) {
        return (pageNo - 1) * pageSize;
    }
    ...
}
```

这个 Page 分页类持有两部分信息：一部分信息是分页数据，另一部分信息是分页控制信息，如每页的数据行数、当前的页码、总页数等。也就是说，当调用分页查询方法时，将返回包含业务数据和分页信息的 Page 对象，而非仅包含业务数据的 List 对象，这一点值得读者注意。

18.4.3 通过扩展基类定义 DAO 类

来看一下 BoardDao 类的代码，如代码清单 18-7 所示。

代码清单 18-7 BoardDao.java

```
package com.smart.dao;
import java.util.Iterator;
import com.smart.domain.Board;
...
//① 扩展BaseDao，并确定泛型的类为Board
```

```
@Repository
public class BoardDao extends BaseDao<Board>{
  private static final String GET_BOARD_NUM = "select count(f.boardId) from Board f";

  //② 获取论坛版块数目的方法
  public long getBoardNum() {
      Iterator iter = getHibernateTemplate().iterate(GET_BOARD_NUM);
      return ((Long)iter.next());
   }
}
```

BoardDao 是操作 Board 的 DAO 类，它扩展于 BoardDao<T>，同时指定泛型类型 T 为 Board，这样在基类中定义的 save(T obj)和 update(T obj)等通用方法的入参就确定为 Board。由于通用性的方法已经在基类中实现，所以 BoardDao 仅需实现一个非通用性的 getBoardNum()方法就可以了，这个方法返回所有论坛版块的数目。

再来看一下 TopicDao 类的代码，如代码清单 18-8 所示。

代码清单 18-8　TopicDao.java

```
package com.smart.dao;
import java.util.List;
import com.smart.domain.Topic;
@Repository
public class TopicDao extends BaseDao<Topic> {
    private static final String GET_BOARD_DIGEST_TOPICS = "from Topic t where t.boardId = ?
                        and digest > 0 order by t.lastPost desc,digest desc";
    private static final String GET_PAGED_TOPICS = "from Topic where boardId = ?
                                            order by lastPost desc";
    private static final QUERY_TOPIC_BY_TITILE = "from Topic  where topicTitle like ?
                                            order by lastPost desc";

    //①获取论坛版块某一页的精华主题帖子，按最后回复时间及精华级别降序排列
    public Page getBoardDigestTopics(int boardId,int pageNo,int pageSize){
       return pagedQuery(GET_BOARD_DIGEST_TOPICS,pageNo,pageSize, boardId);
    }

    //② 获取论坛版块某一页的主题帖子
    public Page getPagedTopics(int boardId,int pageNo,int pageSize) {
        return pagedQuery(GET_PAGED_TOPICS,pageNo,pageSize, boardId);
    }

    //③ 获取和主题帖子标题模糊匹配的主题帖子（某一页的数据）
    public Page queryTopicByTitle(String title, int pageNo, int pageSize) {
        return pagedQuery(QUERY_TOPIC_BY_TITILE,pageNo,pageSize);
    }
}
```

由于需要列出一个论坛版块的主题帖子，所以我们提供了 getPagedTopics(int boardId,int pageNo,int pageSize)方法，而 getBoardDigestTopics(int boardId)方法用于获取论坛版块精华主题帖子。由于用户需要以关键字为条件查询匹配的帖子，所以我们提供了一个对主题帖子的标题执行模糊查询的 queryTopicByTitle(String title, int pageNo, int pageSize)方法。由此可以知道，DAO 的方法需要根据具体的业务需求确定，它为服务

层的 Service 类提供数据获取的实现。DAO 层还有另外 3 个 DAO 类，分别是 UserDao、LoginLogDao 和 PostDao，读者可以通过本章案例查看它们的具体代码。

18.4.4 DAO Bean 的装配

在完成 DAO 的开发后，需要在 Spring 配置文件中将它们定义为 Bean。我们在 src/main/resources 目录下创建一个用于配置 DAO 的 Spring 配置文件 xiaochun-dao.xml。在定义这些 DAO 之前，需要先定义一些基础设施，如数据源、HibernateTemplate 等，如代码清单 18-9 所示。

代码清单 18-9 xiaochun-dao.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
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    ...>
   <!-- 扫描com.smart.dao包下所有标注@Repository的DAO组件 -->
   <context:component-scan base-package="com.smart.dao"/>

   <!--① 引入定义JDBC连接的属性文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--② 定义一个数据源-->
    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />

    <!--③ 定义Hibernate的Session工厂-->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate4.annotation.AnnotationSession
FactoryBean">
        <property name="dataSource" ref="dataSource" />
       <property name="packagesToScan"><!--③-1 扫描基于JPA注解的PO类目录-->
          <list>
             <value>com.smart.domain</value>
          </list>
       </property>
        <!--③-2 指定Hibernate的属性信息-->
        <property name="hibernateProperties">
            <props>
              <!--③-2-1 指定数据库的类型为MySQL-->
               <prop key="hibernate.dialect">
                   org.hibernate.dialect.MySQLDialect
               </prop>
```

```
            <!--③-2-2 在提供数据库操作里显示SQL,
                    方便开发期的调试，在部署时应该将其设计为false-->
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

    <!--④ 定义HibernateTemplate-->
    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate4.HibernateTemplate"
         p:sessionFactory-ref="sessionFactory" />
</beans>
```

在①处引入了一个外部的属性文件，这个属性文件定义了 JDBC 连接的相关信息，其内容如代码清单 18-10 所示。

代码清单 18-10　jdbc.properties

```
#Mysql
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/sampledb?useUnicode=true&characterEncoding=UTF-8
jdbc.username=root
jdbc.password=123456
```

在②处定义了一个数据源，需要特别指出的是 jdbc.url 属性，它附带了两个参数：useUnicode=true&characterEncoding=UTF-8，这两个参数告诉 JDBC 在和 MySQL 数据库通信时需要使用特定的编码。这是因为 JDBC 在默认情况下会采用操作系统的默认编码和数据库通信（如中文操作系统一般是 GBK），如果数据库采用的编码不是系统默认的编码，则需要显式指定通信的编码格式，否则就会产生中文乱码问题。由于我们的数据库采用了 UTF-8 编码格式，所以需要添加这两个参数。从最优项目实践来讲，这种直接在 Spring 中定义数据源的方式并不是最佳方式，至于如何优化改进，请参见 18.11 节。

在③处定义了一个 Hibernate Session 工厂，这个 Session 工厂 Bean 需要使用数据源，此外还必须为其指定 Hibernate 注解包扫描路径。由于论坛的 PO 是基于 Hibernate JPA 注解的，所以仅需定义 packagesToScan 属性就可以了。Hibernate 的配置文件可以定义诸多 Hibernate 属性，在 Spring 中可以通过 hibernateProperties 定义这些属性，这里仅定义了两个 Hibenate 属性。

在④处定义了一个 HibernateTemplate 的实现，HibernateTemplate 是 Spring 提供的旨在简化 Hibernate API 调用的模板类。

18.4.5　使用 Hibernate 二级缓存

Hibernate 拥有一级和二级缓存。一级缓存是由 Session 实现的，天生拥有并且不可拆卸，所以无须关注。Hibernate 使用插件的方式实现二级缓存，在默认情况下，二级缓存是关闭的。合理地使用二级缓存可以有效地减少对数据库的访问次数，提升应用的整体性能。

对于一个版块 Board 对象来说，其实例数目比较少且不常发生更改，User 对象的实例数目比较多但也不经常发生变化，而 Topic 和 Post 的实例数目比较多且较常发生变化。我们将根据这些 PO 的特点使用不同的缓存策略。

配置二级缓存主要有两个步骤。

① 选择需要使用的第三方二级缓存组件（如 EHCache、MemCached 等），在基于 JPA 注解的实体对象或 SessionFactory 的配置中定义缓存策略。

② 配置所选第三方缓存组件的配置文件。每种缓存组件都有自己的配置文件，因此需要手工编辑它们的配置文件，并将它们放置在类路径下。对于 EHCache 来说，其配置文件为 ehcache.xml，而 JBossCache 的配置文件为 treecache.xml。

我们采用 EHCache 缓存实现方案，首先通过 SessionFactory 的配置启用二级缓存并定义缓存策略，这需要调整 xiaochun-dao.xml 对于 sessionFactory Bean 的配置，如代码清单 18-11 所示。

代码清单 18-11　xiaochun-dao.xml

```
...
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        ...
        <property name="hibernateProperties">
            <props>
                ...
              <!--① 采用EHCache缓存实现方案-->
                <prop key="hibernate.cache.region.factory_class">
                    org.hibernate.cache.ehcache.EhCacheRegionFactory
                </prop>
                <prop key="hibernate.cache.use_query_cache">false</prop>
            </props>
        </property>
</bean>
...
```

在①处启用了二级缓存，通过 hibernate.cache.region.factory_class 指定了缓存实现类。在这里，我们采用 EHCache 缓存实现方案。

最后还需要配置 EHCache 的配置文件，将其命名为 ehcache.xml 并放置到类路径下（chapter18/ src/main/resources/）。来看一下 ehcache.xml 的配置内容，如代码清单 18-12 所示。

代码清单 18-12　ehcache.xml

```
<ehcache>
    <diskStore path="java.io.tmpdir" />
    <defaultCache maxElementsInMemory="10000" eternal="false"
        overflowToDisk="false" timeToIdleSeconds="0" timeToLiveSeconds="0"
        diskPersistent="false" diskExpiryThreadIntervalSeconds="120" />
    <!--①存放Board的缓存区-->
    <cache name="fixedRegion" maxElementsInMemory="100"
         eternal="true" overflowToDisk="false"/>
```

```
    <!--② 存放User、Topic和Post的缓存区-->
    <cache name="freqChangeRegion" maxElementsInMemory="5000" eternal="false"
        overflowToDisk="true" timeToIdleSeconds="300" timeToLiveSeconds="1800"/>
</ehcache>
```

在①处定义的 fixedRegion 缓存区不使用硬盘缓存，所有对象都在内存中，缓存区中的对象永不过期，这非常适合缓存类似于 Board 的实例。在②处定义的 freqChangeRegion 缓存区使用硬盘缓存，对象在闲置 300 秒后就从缓存中清除，且对象的最大存活期限为 30 分钟，缓存区中最大的缓存实例个数为 5000 个，超出此限制的实例将被写到硬盘中。这样就完成了 Hibernate 二级缓存的所有配置，当启用 Spring 时，二级缓存就开始工作了。

需要注意的是，上述配置的 EHCache 只支持单机缓存（更改 ehcache.xml 配置，可以支持分布式集群），也就是在集群环境中，每个应用节点的缓存都是相互独立、无法共享的，这导致缓存的命中率不高。如果 Hibernate 二级缓存要运行在集群环境中，则需要第三方缓存组件支持集群能力，目前比较常用的有 EHCache、MemCached、JBossCache。其中 EHCache 和 JBossCache 是基于 Java 语言的高效缓存组件，它们都支持分布式集群，支持多种方式（如 JGroup、JMS、RMI）进行应用节点缓存的同步，缓存可以存储在内存或硬盘中。MemCached 服务器端是基于 C 语言的高性能集中式缓存组件，客户端支持多种语言（如 Java、C、PHP 等），缓存只能存储在内存中。与分布式缓存不同的是，集中式缓存为每个应用节点提供了统一的缓存服务，因此，每个应用节点不涉及缓存同步问题。

18.5　对持久层进行测试

按照 Kent Back 的观点，单元测试的重要特性之一应该是可重复性。不可重复的单元测试是没有价值的。因此，好的单元测试应该具备独立性和可重复性。DAO 层因为是和数据库打交道的层，所以 DAO 层的单元测试依赖于数据库中的数据。要实现 DAO 层单元测试的可重复性，就需要对每次因单元测试引起的数据库中的数据变化进行还原，也就是保护单元测试数据库的数据现场。Spring 测试框架并不能很好地解决所有问题，要解决这些问题，必须整合多方资源。下面使用第 20 章介绍的 DbUnit、Unitils 等测试框架来测试论坛 DAO 层。

18.5.1　配置 Unitils 测试环境

首先在测试资源目录 src/test/resources 中创建一个项目级别的 unitils.properties 配置文件，并对 Unitils 进行相应的配置，如 Unitils 模块、数据库连接信息、数据维护策略等。详细的配置如代码清单 18-13 所示。

代码清单 18-13　unitils.properties

```
#① 启用Unitils所需模块
unitils.modules=database,dbunit,hibernate,spring

#② 配置数据库连接
database.driverClassName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/sampledb?useUnicode=true&characterEncoding=
UTF-8
database.dialect = mysql
database.userName=root
database.password=123456
database.schemaNames=sampledb

#③ 配置数据库维护策略
updateDataBaseSchema.enabled=true

#④ 配置数据库表创建策略
dbMaintainer.autoCreateExecutedScriptsTable=true
dbMaintainer.script.locations=D:/masterSpring/chapter18/src/test/resources/dbscripts

#⑤配置数据集工厂
DbUnitModule.DataSet.factory.default=com.smart.test.dataset.excel
                                        .MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=com.smart.test.dataset.excel
                                        .MultiSchemaXlsDataSetFactory
```

对 DAO 层进行测试，需要与测试数据库、持久层 Hibernate 框架、运行 Spring 容器集成，因此在①处配置 Unitils 的加载模块有 database、dbunit、hibernate、spring。在②处配置测试数据库的连接信息。在③处和④处配置测试数据库的维护策略。在⑤处准备数据集及验证数据集工厂。

18.5.2　准备测试数据库及测试数据

配置好 Unitils 加载模块、测试数据库连接信息、数据库维护策略之后，开始测试数据库及测试数据准备工作。首先在测试目录 src/test/resources/dbscripts 中创建一个数据库创建脚本文件 001_create_sampledb.sql，里面分别是创建论坛版块表 t_board、帖子表 t_post、话题表 t_topic 等创建数据库脚本信息，如代码清单 18-14 所示。

代码清单 18-14　001_create_sampledb.sql

```
CREATE TABLE t_board (
   board_id int(11) NOT NULL auto_increment COMMENT '论坛版块ID',
   board_name varchar(150) NOT NULL default '' COMMENT '论坛版块名',
   board_desc varchar(255) default NULL COMMENT '论坛版块描述',
   topic_num int(11) NOT NULL default '0' COMMENT '帖子数目',
   PRIMARY KEY  (board_id),
   KEY AK_Board_NAME (board_name)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8;

CREATE TABLE t_post (
```

```
  post_id int(11) NOT NULL auto_increment COMMENT '帖子ID',
  board_id int(11) NOT NULL default '0' COMMENT '论坛ID',
  topic_id int(11) NOT NULL default '0' COMMENT '话题ID',
  user_id int(11) NOT NULL default '0' COMMENT '发表者ID',
  post_type tinyint(4) NOT NULL default '2' COMMENT '帖子类型 1:主帖子 2:回复帖子',
  post_title varchar(50) NOT NULL COMMENT '帖子标题',
  post_text text NOT NULL COMMENT '帖子内容',
  create_time date NOT NULL COMMENT '创建时间',
  PRIMARY KEY  (post_id),
  KEY IDX_POST_TOPIC_ID (topic_id)
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT CHARSET=utf8 COMMENT='帖子';
```

准备好测试数据库的脚本文件之后，接下来使用 Excel 准备测试数据及验证数据（Excel 测试数据的格式要求详见第 17 章）。这里分别对 BoardDao、TopicDao、PostDao、UserDao 4 个 DAO 进行测试，需要为每个 DAO 创建相应的测试数据及验证数据，并放置到 DAO 相应的类路径下，如图 18-22 所示。

下面列举论坛话题的 Excel 测试验证数据集结构，如图 18-23 所示。其他测试数据集结构与论坛话题验证数据集结构相似，这里就不一一列出了。

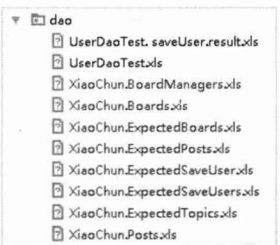


图 18-22　DAO 测试数据目录结构图

	A	B	C	D	E	F	G
1	topic_id	board_id	topic_title	user_id	create_til	last_post	topic
2	1	1	AOP背后的故事	1	2011/5/7	2011/5/16	
3	2	4	IOC的原理一	1	2011/5/7	2011/5/16	
4	3	4	IOC的原理二	1	2011/5/7	2011/5/16	
5	4	1	SpringMVC集成	1	2011/5/7	2011/9/18	
6	5	1	SpringMVC集成	1	2011/5/7	2011/9/18	
7							

t_topic

图 18-23　论坛话题验证数据集结构

18.5.3　编写 DAO 测试基类

配置好 Unitils 测试环境及准备好测试数据之后，开始着手编写 DAO 测试用例。为了简化每个 DAO 测试，我们编写了一个 DAO 测试基类，所有的 DAO 测试用例都需要扩展该基类。具体实现如代码清单 18-15 所示。

代码清单 18-15　BaseDaoTest.java

```
package com.smart.dao;

import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;

@SpringApplicationContext( {"classpath:/xiaochun-dao.xml" })
public class BaseDaoTest extends UnitilsTestNG{

}

...
```

测试 DAO 只需用到 DAO 层的配置信息，因此，在 DAO 测试基类中，只需通过

@SpringApplicationContext 注解加载 xiaochun-dao.xml 配置文件即可。由于我们的 DAO 测试用例基于 Unitils、TestNG 测试框架，因此需要扩展 Unitils 提供的 UnitilsTestNG 测试基类。

18.5.4 编写 BoardDao 测试用例

编写好 DAO 测试基类之后，接下来开始编写每个 DAO 相应的测试用例，首先来看一下 BoardDao 测试用例，具体实现如代码清单 18-16 所示。

代码清单 18-16 BoardDaoTest.java

```
package com.smart.dao;
import org.testng.annotations.Test;
import org.unitils.dbunit.annotation.DataSet;
import com.smart.test.dataset.util.XlsDataSetBeanFactory;
...
public class BoardDaoTest extends BaseDaoTest{

    //① 注入论坛版块DAO
    @SpringBean("boardDao")
    private BoardDao boardDao;

    //② 测试添加版块
    @Test
    @ExpectedDataSet("Xiaochun.ExpectedBoards.xls") //②-1 验证数据
    public void addBoard()throws Exception {
        //②-2 通过XlsDataSetBeanFactory数据集绑定工厂创建测试实体
        List<Board> boards  = XlsDataSetBeanFactory.createBeans(BoardDaoTest.class,
       "Xiaochun.SaveBoards.xls", "t_board", Board.class);
        //②-3 持久化Board实例
       for(Board board:boards){
           boardDao.save(board);
        }
    }

    //③ 删除测试版块
    @Test
    @DataSet("Xiaochun.Boards.xls")//③-1 准备测试数据
    @ExpectedDataSet("Xiaochun.ExpectedBoards.xls")//③-2 准备验证数据
    public void removeBoard(){

        //③-3 加载指定的版块
        Board board = boardDao.get(7);

        //③-4 删除指定的版块
        boardDao.remove(board);
    }

    //④ 测试加载版块
    @Test
    @DataSet("Xiaochun.Boards.xls")//④-1准备测试数据
```

```
    public void getBoard(){
        //④-2 加载版块
        Board board = boardDao.load(1);

        //④-3 验证结果
        assertNotNull(board);
        assertEquals (board.getBoardName(),"育儿");
    }
}
...
```

在①处，通过 Unitils 提供的@SpringBean 注解，从 Spring 容器中加载 BoardDao 实例。

在②处，测试保存版块功能 BoardDao#save()，首先通过 TestNG 提供的@Test 注解将当前方法标志为测试方法。在②-1 处通过 Unitils 提供的@ExpectedDataSet 注解，从当前测试用例所在的类路径中加载 Xiaochun.ExpectedBoards.xls 数据集文件。在②-2 处通过 XlsDataSetBeanFactory#createBeans()方法从 Xiaochun.SaveBoards.xls 数据集中读取数据并实例化 Board 实体。在②-3 处，通过 BoardDao#save()方法，将所有的 Board 实例持久化到数据库中。最后通过@ExpectedDataSet 注解验证加载的数据是否与数据库中的数据匹配。

在③处，对删除版块 BoardDao#remove()方法进行测试，首先通过 Unitils 提供的@DataSet 注解从当前测试用例所在的类路径中加载 Xiaochun.Boards.xls 数据集文件，并将 Xiaochun.Boards.xls 数据集中的数据保存到相应的数据库表中。由于我们采用 Unitils 默认数据集先清理后保存策略，也就是先删除当前 Xiaochun.Boards.xls 对应的数据库表中的数据，然后将当前 Xiaochun.Boards.xls 数据集中的数据同步到数据库中。在③-3 处通过 BoardDao#get()方法加载一个指定版块，并在③-4 处通过 BoardDao#remove()方法将指定版块删除。最后通过@ExpectedDataSet 注解加载的验证数据与当前数据库中的数据进行匹配验证。

在④处，对加载版块 BoardDao#load()方法进行测试，首先通过 Unitils 提供的@DataSet 注解从当前测试用例所在的类路径中加载 Xiaochun.Boards.xls 数据集文件，然后通过 BoardDao#load()方法加载一个版块，并通过 TestNG 提供的断言对加载 Board 实体进行验证。最后在 IDE 中执行 BoardDaoTest 测试用例，测试结果如图 18-24 所示。

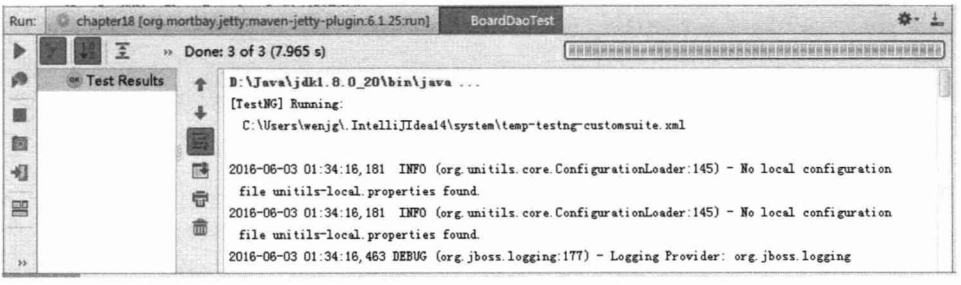


图 18-24　BoardDao 测试结果

至此，我们完成了对 BoardDao 测试用例的编写。由于每个 DAO 测试用例的编写方法都一样，此处就不一一阐述了，感兴趣的读者可以查看本书配套网盘中提供的相应章节案例代码。

18.6 服务层开发

服务层位于 Web 层和 DAO 层之间，服务类调用 DAO 层的类完成各项业务操作，并开放给 Web 层进行调用。我们在服务层中定义了两个服务类，分别是负责用户操作的 UserService 和负责论坛功能的 ForumService。

18.6.1 UserService 的开发

首先来看一下 UserService 的代码，具体实现如代码清单 18-17 所示。

代码清单 18-17 UserService.java

```
package com.smart.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
...
//用户管理服务器，负责执行查询用户、注册用户、锁定用户等操作
@Service
public class UserService {
    @Autowired
    private UserDao userDao;
    @Autowired
    private LoginLogDao loginLogDao;

    //注册一个新用户。如果用户名已经存在，则抛出UserExistException异常
    public void register(User user) throws UserExistException{
        User u = this.getUserByUserName(user.getUserName());
        if(u != null){
            throw new UserExistException("用户名已经存在");
        }else{
            user.setCredit(100);
          user.setUserType(1);
          userDao.save(user);
        }
    }

   //根据用户名/密码查询User对象
   public User getUserByUserName(String userName){
      return userDao.getUserByUserName(userName);
   }

    //根据userId加载User对象
   public User getUserById(int userId){
```

```
        return userDao.get(userId);
    }

    //将用户锁定，锁定的用户不能登录
    public void lockUser(String userName){
        User user = userDao.getUserByUserName(userName);
        user.setLocked(User.USER_LOCK);
       userDao.update(user);
    }

    //解除用户的锁定
    public void unlockUser(String userName){
        User user = userDao.getUserByUserName(userName);
        user.setLocked(User.USER_UNLOCK);
        userDao.update(user);
    }

   ...
}
...
```

UserService 使用 UserDao、LoginLog 完成论坛用户的各项业务操作，这些方法都非常直观，相信读者可以轻松看懂。通过 Spring 提供的@Autowired 注解，自动从 Spring 容器中加载 UserDao 和 LoginLog 这两个实例。UserService 事务管理通过 Spring 声明式事务管理的功能实现，通过事务的声明性信息，Spring 负责将事务管理增强逻辑动态织入业务方法相应的连接点中。这些逻辑包括获取线程绑定资源、开始事务、提交/回滚事务、进行异常转换和处理等工作。整个论坛服务层的事务管理统一在 xiaochun-service.xml 文件中进行配置，下文将会对其进行详细讲解。

18.6.2　ForumService 的开发

在学习完 UserService 后，再来了解一下提供论坛功能的 ForumService 服务类的代码，具体实现如代码清单 18-18 所示。

代码清单 18-18　ForumService.java

```
package com.smart.service;
import com.smart.dao.*;
import com.smart.domain.*;
...
@Service
public class ForumService {
    @Autowired
    private TopicDao topicDao;
    ...
    //①发表一个主题帖子，用户积分加10，论坛版块的主题帖子数加1
    public void addTopic(Topic topic) {
        Board board = (Board) boardDao.get(topic.getBoardId());
        board.setTopicNum(board.getTopicNum() + 1);
        topicDao.save(topic);
```

```
        //①-1创建主题帖子
        topic.getMainPost().setTopic(topic);
        MainPost post = topic.getMainPost();
        post.setCreateTime(new Date());
        post.setUser(topic.getUser());
        post.setPostTitle(topic.getTopicTitle());
        post.setBoardId(topic.getBoardId());

        //①-2 持久化主题帖子
        postDao.save(topic.getMainPost());

        //①-3 更新用户积分
        User user = topic.getUser();
        user.setCredit(user.getCredit() + 10);
        userDao.update(user);
    }

    //② 删除一个主题帖子，用户积分减50，论坛版块的主题帖子数减1，
    //删除主题帖子所有关联的帖子
    public void removeTopic(int topicId) {
        Topic topic = topicDao.get(topicId);

        // 将论坛版块的主题帖子数减1
        Board board = boardDao.get(topic.getBoardId());
        board.setTopicNum(board.getTopicNum() - 1);

        // 发表该主题帖子的用户扣除50个积分
        User user = topic.getUser();
        user.setCredit(user.getCredit() - 50);

        // 删除主题帖子及其关联的帖子
        topicDao.remove(topic);
        postDao.deleteTopicPosts(topicId);
    }

    //③ 添加一个回复帖子，用户积分加5，主题帖子回复数加1，并更新最后回复时间
    public void addPost(Post post){
        postDao.save(post);
        User user = post.getUser();
        user.setCredit(user.getCredit() + 5);
        userDao.update(user);
        Topic topic = topicDao.get(post.getTopicId());
        topic.setReplies(topic.getReplies() + 1);
        topic.setLastPost(new Date());

        //topic处于Hibernate受管状态，无须显示更新
        //topicDao.update(topic);
    }
    ...
}
...
```

ForumService 是论坛的核心服务类，它实现了论坛的大部分功能。ForumService 引用了 BoardDao、TopicDao、PostDao 和 UserDao 这 4 个 DAO 类。

大部分服务方法都比较简单，它们完成业务逻辑并进行数据持久化操作。下面仅对一些复杂的方法进行说明。先来看一下①处的 addTopic()方法：在①-1 处通过 Topic 实例创建主题帖子，在①-2 处调用基类的 save()方法进行持久化操作，在①-3 处为发表者添加 10 个积分，并调用 UserDao 的 update()方法更新到数据库中。

③处的 addPost()方法可能也会存在理解上的障碍。在该方法中，我们添加了一个回复帖子，同时更新主题帖子的回复帖子数及主题的最后回复时间，但是并没有调用 TopicDao 的 update()方法更新 Topic。也许会有读者怀疑这里的代码是否有误，其实这个方法是可以正确工作的。因为我们通过 topicDao.get(post.getTopicId())方法从数据表中加载 Topic 实例，所以这个 Topic 实例处于受管的状态，在方法中调整其 replies 和 lastPost 属性，Hibernate 会将 Topic 状态更改自动同步到数据表中，无须显式调用 topicDao.update()方法。

实战经验

由于数据库自增键会带来诸多的不便或潜在的问题，如在一些基于配置型的应用中，经常会涉及一些配置模型数据的导入/导出操作，此时很容易造成主键冲突问题。因此，建议读者在实际开发中尽量采用 UUID 的主键，或使用一个服务在应用层中获取领域对象的主键，省却这种因数据库产生主键而造成的麻烦。

18.6.3　服务类 Bean 的装配

在编写完 UserService 和 ForumService 后，需要在 Spring 配置文件中进行装配，以便注入 DAO Bean 并实施事务管理增强。我们在 src/main/resources 目录下创建一个 Spring 的配置文件 xiaochun-service.xml，该文件的内容如代码清单 18-19 所示。

代码清单 18-19　xiaochun-service.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
      xmlns:tx="http://www.springframework.org/schema/tx"
      ...
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-4.0.xsd">
```

```
    <!--①扫描com.smart.service包下所有标注@Service的服务组件 -->
    <context:component-scan base-package="com.smart.service"/>
    <!--②事务管理器-->
     <bean id="transactionManager"
           class="org.springframework.orm.hibernate4.HibernateTransactionManager"
           p:sessionFactory-ref="sessionFactory" />
     <!--③使用强大的切点表达式语言轻松定义目标方法-->
     <aop:config>
        <!--③-1通过AOP定义事务增强切面-->
         <aop:pointcut id="serviceMethod"
             expression="execution(* com.smart.service.*Service.*(..))" />
        <!--③-2引用事务增强-->
         <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
     </aop:config>
     <!--④事务增强-->
     <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <!--④-1事务属性定义-->
         <tx:attributes>
             <tx:method name="*" />
         </tx:attributes>
     </tx:advice>

    <!--⑤ 基于EHCache的系统缓存管理器-->
    <bean id="cacheManager"
         class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
         p:configLocation="classpath:ehcache.xml"/>
</beans>
...
```

首先在①处扫描 com.smart.service 包下所有标注@Service 注解的 Service Bean，Service Bean 引用 xiaochun-dao.xml 扫描 DAO Bean。接着对论坛服务层配置事务管理，需要在配置文件中引入 tx 命名空间的声明，如<beans>元素粗体部分所示。采用 aop/tx 定义事务方法时，它站在“局外人”的角度对 IoC 容器中的 Bean 进行事务管理配置定义，再由 Spring 将这些配置织入对应的 Bean 中。

在 aop 命名空间中，通过切点表达式语言，我们将 com.smart.service 包下所有以 Service 为后缀的类纳入了需要进行事务增强的范围，并配合<tx:advice>的<aop:advisor>完成了事务切面的定义，如③处所示。

<aop:advisor>引用的 txAdvice 增强是在 tx 命名空间中定义的，如④处所示。事务增强必须有一个事务管理器的支持，<tx:advice>通过 transaction-manager 属性引用了在②处定义的事务管理器（默认查找名为 transactionManager 的事务管理器，所以如果事务管理器命名为 transactionManager，则可以不指定 transaction-manager 属性）。

18.7 对服务层进行测试

服务层位于 Web 层和 DAO 层之间，要对服务层实施测试，需要与 DAO 层进行交互。但在测试过程中，这些需要被调用的真实对象常常很难被实例化，或者这些对象在

某些情况下无法被用来测试，例如，真实对象的行为无法预测、真实对象的行为难以触发，或者真实对象的运行速度很慢。这时候就需要使用模拟对象技术（Mock），利用一个模拟对象来模拟代码所依赖的真实对象，以帮助完成测试，或者对服务层和 DAO 层进行集成测试。下面使用第 20 章介绍的 Unitils、TestNG 等框架对论坛服务层进行集成测试。

18.7.1　编写 Service 测试基类

编写好论坛 UserService 和 ForumService 两个服务类之后，开始编写相应的测试用例。为了简化每个 Service 测试，我们编写了一个 Service 测试基类，所有的 Service 测试用例都需要扩展该基类，具体实现如代码清单 18-20 所示。

代码清单 18-20　BaseServiceTest.java

```
package com.smart.service;
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
@SpringApplicationContext( {"xiaochun-service.xml", "xiaochun-dao.xml" })
public class BaseServiceTest extends UnitilsTestNG {
    @SpringBean("hibernateTemplate")
    public  HibernateTemplate hibernateTemplate;

}
...
```

测试 Service 层一般有两种方法：一种是通过 Mockito 等模拟测试框架对 DAO 层进行模拟，完成 Service 层独立性测试；另一种通过 Unitils 等框架对 Service 层和 DAO 层进行集成测试。本案例采用集成测试方法，首先通过 Unitils 提供的@SpringApplication Context 注解加载 Service 层和 DAO 层的配置文件 xiaochun-service.xml 和 xiaochun-dao.xml，然后通过@SpringBean 注解从 Spring 容器中加载 HibernateTemplate 实例。

18.7.2　编写 ForumService 测试用例

编写好 Service 测试基类之后，接下来编写每个 Service 相应的测试用例（在第 20 章中会对 UserService 测试用例进行讲解，这里不再阐述）。下面重点看一下论坛的核心服务 ForumService 测试用例，具体实现如代码清单 18-21 所示。

代码清单 18-21　UserServiceTest.java

```
package com.smart.service;
import static org.hamcrest.Matchers.*;
import com.smart.test.dataset.util.XlsDataSetBeanFactory;
...
public class ForumServiceTest extends BaseServiceTest {
```

```
    @SpringBean("forumService")
    private ForumService forumService;
    @SpringBean("userService")
    private UserService userService;

    //① 在测试初始化中，消除Hibernate二级缓存，避免影响测试
    @BeforeMethod
    public void init(){
        SessionFactory sf  = hibernateTemplate.getSessionFactory();
        Map<String, CollectionMetadata> roleMap = sf.getAllCollectionMetadata();
        for (String roleName : roleMap.keySet()) {
            sf.evictCollection(roleName);
        }
        Map<String, ClassMetadata> entityMap = sf.getAllClassMetadata();
        for (String entityName : entityMap.keySet()) {
            sf.evictEntity(entityName);
        }
        sf.evictQueries();
    }


    //② 测试新增一个版块
    @Test
    @DataSet("Xiaochun.DataSet.xls")
    public void addBoard() throws Exception {
        Board board = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
                "Xiaochun.DataSet.xls", "t_board", Board.class);
        forumService.addBoard(board);
        Board boardDb = forumService.getBoardById(board.getBoardId());
        assertEquals (boardDb.getBoardName(), "育儿");
    }

    //③ 测试新增一个主题帖子
    @Test
    @DataSet("Xiaochun.DataSet.xls")
    public void addTopic() throws Exception {
        Topic topic = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
                "Xiaochun.DataSet.xls", "t_topic", Topic.class);
        User user = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
                "Xiaochun.DataSet.xls", "t_user", User.class);
        topic.setUser(user);
        forumService.addTopic(topic);
        Board boardDb = forumService.getBoardById(1);
        User userDb = userService.getUserByUserName("tom");
        assertEquals(boardDb.getTopicNum(), 1);
        assertEquals(userDb.getCredit(), 110);
        assertEquals(topic.getTopicId(), 0);
    }
    ...
}
```

上面的测试用例都比较好理解，首先使用 Excel 准备测试数据（格式要求详见第 20 章，这里不再阐述），如案例中的 Xiaochun.DataSet.xls 文件。然后通过 Unitils 提供的 @DataSet 注解从当前测试用例的所有类路径中加载 Xiaochun.DataSet.xls 数据集文件并

初始化测试数据库。通过自己编写的 XlsDataSetBeanFactory 工具类，加载 Xiaochun.DataSet.xls 测试数据，并实例化测试对象实例。接着通过 TestNG 和 Hamcrest 提供的断言及匹配方法对结果进行验证。由于在论坛中启用了 Hibernate 二级缓存，为了不影响测试结果，在执行测试方法之前，需要消除 Hibernate 二级缓存，如①处所示。最后在 IDE 中执行 ForumServiceTest 测试用例，可以看到测试运行结果。

至此，我们完成了对 ForumService 和 UserService 测试用例的编写。由于每个 Service 测试用例的编写方法都一样，在这里就不一一阐述了，感兴趣的读者可以查看本书配套网盘中提供的相应章节案例代码。

18.8　Web 层开发

至此，DAO 和服务类都已经准备就绪，并且通过单元测试对其进行了全面测试。接下来需要开发 Web 层将服务和页面关联起来。

18.8.1　BaseController 的基类

由于 Web 层的每个 Controller 都有一些常用的操作，所以我们提供了一个 Controller 的基类，如代码清单 18-22 所示。

代码清单 18-22　BaseController.java

```
package com.smart.web;
import com.smart.domain.User;
import com.smart.exception.NotLoginException;
...
public class BaseController {
    protected static final String ERROR_MSG_KEY = "errorMsg";

    //① 获取保存在Session中的用户对象
    protected User getSessionUser(HttpServletRequest request) {
        return (User) request.getSession().getAttribute(
                CommonConstant.USER_CONTEXT);
    }

    //②将用户对象保存到Session中
    protected void setSessionUser(HttpServletRequest request,User user) {
        request.getSession().setAttribute(CommonConstant.USER_CONTEXT,
                user);
    }

    //③ 获取基于应用程序的URL绝对路径
    public final String getAppbaseUrl(HttpServletRequest request, String url) {
        Assert.hasLength(url, "url不能为空");
        Assert.isTrue(url.startsWith("/"), "必须以/打头");
        return request.getContextPath() + url;
```

```
    }
    ...
}
```

由于论坛的 Controller 基于 Spring MVC 注解方式，因此我们无须扩展任何基类。在 BaseController 类中定义了 3 个方法，其中 setSessionUser()方法负责将 User 对象保存到 Session 中；getSessionUser()方法负责获取保存在 Session 中的 User 对象；getAppbaseUrl()方法负责获取基于应用程序的 URL 绝对路径。

Web 层的每个 Controller 都有可能涉及登录验证处理逻辑，如论坛中只有登录用户才能发表新话题，所以我们提供了一个过滤器来进行处理，如代码清单 18-23 所示。

代码清单 18-23　ForumFilter.java

```
package com.smart.web;
import javax.servlet.Filter;
import com.smart.domain.User;
import static com.smart.cons.CommonConstant.*;

...
public class ForumFilter implements Filter {
    private static final String FILTERED_REQUEST = "@@session_context_filtered_
request";

    //①不需要登录即可访问的URI资源
    private static final String[] INHERENT_ESCAPE_URIS = { "/index.jsp","/index.html",
                                "/login.jsp","/login/doLogin.html","/register.jsp",
                        "/register.html","/board/listBoardTopics-","/board/list
TopicPosts-"};
    //②执行过滤
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {

        //②-1保证该过滤器在一次请求中只被调用一次
        if (request != null && request.getAttribute(FILTERED_REQUEST) != null) {
            chain.doFilter(request, response);
        } else {

            //②-2设置过滤标识，防止一次请求多次过滤
            request.setAttribute(FILTERED_REQUEST, Boolean.TRUE);
            HttpServletRequest httpRequest = (HttpServletRequest) request;
            User userContext = getSessionUser(httpRequest);

           //②-3用户未登录，且当前URI资源需要登录才能访问
            if (userContext == null&& ! isURILogin (httpRequest.getRequestURI(),
httpRequest)) {
                        String toUrl = httpRequest.getRequestURL(). toString();
                if (!StringUtils.isEmpty(httpRequest.getQueryString())) {
                    toUrl += "?" + httpRequest.getQueryString();
                }

                //②-4将用户的请求URL保存在Session中，用于登录成功后跳到目标URL
                httpRequest.getSession().setAttribute(LOGIN_TO_URL, toUrl);

                //②-5转发到登录页面
```

```
                request.getRequestDispatcher("/login.jsp").forward(request,response);
                return;
            }
            chain.doFilter(request, response);
        }
    }

    //③当前URI资源是否需要登录才能访问
    private boolean isURILogin (String requestURI, HttpServletRequest request) {
        if (request.getContextPath().equalsIgnoreCase(requestURI)
                || (request.getContextPath() + "/").equalsIgnoreCase(requestURI))
            return true;
        for (String uri : INHERENT_ESCAPE_URIS) {
            if (requestURI != null && requestURI.indexOf(uri) >= 0) {
                return true;
            }
        }
        return false;
    }
    ...
}
```

在过滤器中，首先设置论坛中所有不需要登录即可访问的 URI 资源，如①处所示。在过滤处理过程中，设置一个当前请求的过滤标识，防止一次请求多次过滤的情况，如②-2 处所示。如果用户未登录，且当前 URI 资源需要登录才能访问，则保存当前请求的 URI 到会话中，并转发到登录页面。

18.8.2　用户登录和注销

用户登录和注销功能由 LoginController 负责，LoginController 通过调用服务层的 UserService 类完成相应的业务操作。来看一下 LoginController 的实现，如代码清单 18-24 所示。

代码清单 18-24　LoginController.java

```
package com.smart.web;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import static com.smart.cons.CommonConstant.*;
...
@Controller
@RequestMapping("/login")
public class LoginController extends BaseController {

    private UserService userService;

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    //① 用户登录
```

```
    @RequestMapping("/doLogin")
    public ModelAndView login(HttpServletRequest request, User user) {
        User dbUser = userService.getUserByUserName(user.getUserName());
        ModelAndView mav = new ModelAndView();
        mav.setViewName("forward:/login.jsp");
        if(dbUser == null){
            mav.addObject("errorMsg", "用户名不存在");
         }else if(!dbUser.getPassword().equals(user.getPassword())){
            mav.addObject("errorMsg","用户密码不正确");
         }else if(dbUser.getLocked() == User.USER_LOCK){
            mav.addObject("errorMsg","用户已经被锁定，不能登录。");
         }else{
            dbUser.setLastIp(request.getRemoteAddr());
            dbUser.setLastVisit(new Date());
            userService.loginSuccess(dbUser);
            setSessionUser(request,dbUser);
            String toUrl = (String)request.getSession().getAttribute(LOGIN_TO_URL);
            request.getSession().removeAttribute(LOGIN_TO_URL);

            //如果当前会话中没有保存登录之前的请求URL，则直接跳转到主页
            if(StringUtils.isEmpty(toUrl)){
                toUrl = "/index.html";
            }
            mav.setViewName("redirect:"+toUrl);
         }
        return mav;
    }

    //② 登录注销
    @RequestMapping("/doLogout")
    public String logout(HttpSession session) {
        session.removeAttribute(USER_CONTEXT);
        return "forward:/index.jsp";
    }
   ...
}
```

LoginController 直接扩展于 BaseController，login()方法负责处理用户登录操作，当用户名不存在、用户密码不正确或者用户已经被锁定时，都直接转到登录页面并报告相关的错误信息；否则用户添加 5 个积分并将其保存到 HTTP Session 中，然后转向成功页面。

在①处，首先根据用户名获取到 User 对象实例，然后对当前用户实例状态进行判断。如果当前用户实例既不为空，也不是被锁定状态，则说明当前用户合法，表示登录成功，调用 UserService#loginSuccess()方法添加当前用户积分并保存登录日志。最后判断当前会话中是否存在登录之前的请求 URL（这个请求 URL 在论坛的过滤器中设置），如果存在则跳转到这个 URL，如果不存在就直接跳转到主页。

②处的用户登录注销方法 logout()很简单，其主要工作是将 User 从 Session 中移除，并转到论坛主页中。这里使用了 LoginController 中定义的方法，将请求重定向到/index.jsp 页面中。

18.8.3　用户注册

要成为论坛的用户，首先必须进行注册。用户注册是论坛的一个非常重要的功能。下面来看一下负责用户注册的 RegisterController，如代码清单 18-25 所示。

代码清单 18-25　RegisterController.java

```
package com.smart.web;
import com.smart.domain.User;
import com.smart.exception.UserExistException;
import com.smart.service.UserService;
...
@Controller
public class RegisterController extends BaseController {
    @Autowired
    private UserService userService;

    //用户登录
    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public ModelAndView register(HttpServletRequest request,User user){
        ModelAndView view = new ModelAndView();
        view.setViewName("/success");
        try {
            userService.register(user);
        } catch (UserExistException e) {
            view.addObject("errorMsg", "用户名已经存在，请选择其他的名字。");
            view.setViewName("forward:/register.jsp");
        }
        setSessionUser(request,user);
        return view;
    }
   ...
}
```

在注册用户之前需要判断用户名是否已经存在，如果已经存在，则必须告之用户，以便用户调整注册的用户名。在 Web 2.0 时代，为了增强用户体验，最好在页面端采用 AJAX 技术，当用户输入用户名时即可自动告之用户是否已经存在，而不是等到用户提交注册表单后再进行判断。不过，不管注册用户名是否已经通过 AJAX 校验，仍有必要在服务器端再验证一次。

接下来的工作是配置用户注册的 JSP 页面，如代码清单 18-26 所示。

代码清单 18-26　register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户注册</title>
<script>
  function mycheck(){
    if(document.all("user.password").value != document.all("again").value){
```

```
        alert("两次输入的密码不正确，请更正。");
        return false;
      }else{
        return true;
      }
    }
</script>
</head>
<body>
用户注册信息:
<form action="<c:url value="/register.html" />" method="post" onsubmit="return mycheck()">
<c:if test="${!empty errorMsg}">
  <div style="color=red">${errorMsg}</div>
</c:if>
<table border="1px" width="60%">
    <tr>
       <td width="20%">用户名</td>
        <td width="80%"><input type="text" name="userName"/></td>
    </tr>
    <tr>
        <td width="20%">密码</td>
        <td width="80%"><input type="password" name="password"/></td>
    </tr>
    <tr>
        <td width="20%">密码确认</td>
        <td width="80%"><input type="password" name="again"></td>
    </tr>
    <tr>
        <td colspan="2">
          <input type="submit" value="保存"> <input type="reset" value="重置">
        </td>
    </tr>
</table>
</form>
</body>
</html>
```

用户注册表单的信息非常简单，仅包含用户名和密码两个信息。当提交表单时，表单信息填充到 User 属性中。

18.8.4 论坛管理

论坛管理模块对应论坛管理员所使用的各项操作功能，具体来说，包括创建论坛版块、指定论坛版块管理员、用户锁定/解锁等功能。ForumManageController 负责处理这些操作请求，如代码清单 18-27 所示。

代码清单 18-27 ForumManageController.java

```
package com.smart.web;
import com.smart.service.ForumService;
import com.smart.service.UserService;
...
```

```
//论坛管理，这部分功能由论坛管理员操作，包括创建论坛版块、指定论坛版块管理员、用户锁定/解锁
@Controller
public class ForumManageController extends BaseController {
    @Autowired
    private ForumService forumService;
    @Autowired
    private UserService userService;

    //① 列出所有的论坛模块
    @RequestMapping(value = "/index", method = RequestMethod.GET)
    public String listAllBoards() {
        List<Board> boards = forumService.getAllBoards();
        request.setAttribute("boards", boards);
        return "/listAllBoards";
    }

    //② 添加一个主题帖子页面
    @RequestMapping(value = "/forum/addBoardPage", method = RequestMethod.GET)
    public String addBoardPage() {
        return "/addBoard";
    }

    //③ 添加一个主题帖子
    @RequestMapping(value = "/forum/addBoard", method = RequestMethod.POST)
    public String addBoard(Board board) {
        forumService.addBoard(board);
        return "/addBoardSuccess";
    }

    //④ 指定论坛管理员的页面
    @RequestMapping(value = "/forum/setBoardManagerPage", method = RequestMethod.GET)
    public ModelAndView setBoardManagerPage() {
        ModelAndView view =new ModelAndView();
        List<Board> boards = forumService.getAllBoards();
        List<User> users = userService.getAllUsers();
        view.addObject("boards", boards);
        view.addObject("users", users);
        view.setViewName("/setBoardManager");
        return view;
    }
    ...
}
```

ForumManageController 通过调用服务层的 UserService 和 ForumService 完成相应业务逻辑。由于进行用户锁定、指定论坛版块管理员等操作都需要一个具体的操作页面，所以 ForumManageController 的另一项工作是将请求导向一个具体的操作页面中，如②处和④处的 addBoardPage()和 setBoardManagerPage 方法所示。

ForumManageController 共有 4 个转向页面。

- userLockManagePage：对应 WEB-INF/jsp/userLockManage.jsp 页面，即用户解锁和锁定的操作页面。
- setBoardManagerPage：对应 WEB-INF/jsp/setBoardManager.jsp 页面，这是设置论坛版块管理员的处理页面。

- listAllBoards：对应 WEB-INF/jsp/listAllBoards.jsp 页面，该页面显示论坛版块列表。
- addBoardPage：对应 WEB-INF/jsp/addBoard.jsp 页面，这是新增论坛版块的表单页面。

这些JSP页面的代码都比较简单，读者可自行通过本书配套网盘中的代码进行学习，这里不再展开阐述。

18.8.5 论坛普通功能

论坛普通功能包括显示论坛版块列表、显示论坛版块主题列表、发表主题帖子、回复帖子、删除帖子、设置精华帖子等，这些功能由 BoardManageController 负责处理，具体实现如代码清单 18-28 所示。

代码清单 18-28 BoardManageController.java

```
package com.smart.web;
import com.smart.domain.*;
import com.smart.service.ForumService;
…
@Controller
public class BoardManageController extends BaseController {
    @Autowired
    private ForumService forumService;

    //① 列出论坛模块下的主题帖子
    @RequestMapping(value = "/board/listBoardTopics-{boardId}",
method = RequestMethod.GET)
    public ModelAndView listBoardTopics(@PathVariable Integer boardId,
                @RequestParam(value= "pageNo", required = false) Integer pageNo)
{
        ModelAndView view =new ModelAndView();
        Board board = forumService.getBoardById(boardId);
        pageNo = pageNo==null?1:pageNo;
        Page pagedTopic = forumService.getPagedTopics(boardId, pageNo,
                CommonConstant.PAGE_SIZE);
        view.addObject("board", board);
        view.addObject("pagedTopic", pagedTopic);
        view.setViewName("/listBoardTopics");
        return view;
    }

    //② 新增主题帖子页面
    @RequestMapping(value = "/board/addTopicPage-{boardId}", method = RequestMethod. GET)
    public ModelAndView addTopicPage(@PathVariable Integer boardId) {
        ModelAndView view =new ModelAndView();
        view.addObject("boardId", boardId);
        view.setViewName("/addTopic");
        return view;
    }
```

```
    //③ 添加一个主题帖子
    @RequestMapping(value = "/board/addTopic", method = RequestMethod.POST)
    public String addTopic(HttpServletRequest request,Topic topic) {
        User user = getSessionUser(request);
        topic.setUser(user);
        Date now = new Date();
        topic.setCreateTime(now);
        topic.setLastPost(now);
        forumService.addTopic(topic);
        String targetUrl = "/board/listBoardTopics-" + topic.getBoardId()+ ".html";
        return "redirect:"+targetUrl;
    }

    //④ 列出主题的所有帖子
    @RequestMapping(value = "/board/listTopicPosts-{topicId}", method = RequestMethod. GET)
      public ModelAndView listTopicPosts(@PathVariable Integer topicId,
          @RequestParam(value ="pageNo", required = false) Integer pageNo) {
        ModelAndView view =new ModelAndView();
        Topic topic = forumService.getTopicByTopicId(topicId);
        pageNo = pageNo==null?1:pageNo;
        Page pagedPost = forumService.getPagedPosts(topicId, pageNo,
                                                CommonConstant.PAGE_SIZE);
        // 为回复帖子表单准备数据
        view.addObject("topic", topic);
        view.addObject("pagedPost", pagedPost);
        view.setViewName("/listTopicPosts");
        return view;
    }
   ...
}
```

Controller 的主要职责包括页面流程的控制、业务数据的准备，而业务逻辑处理则直接调用服务层的类来完成。如③处的 addTopic()请求处理方法首先应用 Spring MVC 自动绑定功能从表单获取提交的数据并绑定到 Topic 对象实例，接着补充那些需要在服务层产生的数据，如主题帖子的发表人、发表时间、最后回复时间等，然后调用服务层的 ForumService#addTopic()方法新增主题帖子，最后将请求重定向到论坛版块主题帖子列表页面中。

读者可能已经注意到，有很多请求方法处理完成后并未返回一个视图名称，而是直接返回 redirect:targetUrl 完成重定向请求。这是因为目标 URL 是动态的，不方便直接定义返回视图，因此直接利用 Spring MVC 提供的重定向功能进行请求的转向。

BoardManageController 共有 3 个转向页面。

- listBoardTopics：对应 WEB-INF/jsp/listBoardTopics.jsp 页面，该页面显示论坛版块的所有主题帖子。
- listTopicPosts：对应 WEB-INF/jsp/listTopicPosts.jsp 页面，该页面显示主题所对应的所有帖子。
- addTopicPage：对应 WEB-INF/jsp/addTopic.jsp 页面，它是发表新主题帖子的表单页面。

本书配套网盘里包括了所有 JSP 页面的代码，读者可以学习相关知识。

18.8.6 分页显示论坛版块的主题帖子

通过前面的讲述，我们知道论坛分别通过 listBoardTopics.jsp 和 listTopicPosts.jsp 页面分页显示论坛版块下的主题帖子和主题帖子下的所有帖子。本小节通过 listBoardTopics.jsp 页面讲解展现层分页功能的实现过程。先来看一下 listBoardTopics.jsp 页面代码，如代码清单 18-29 所示。

代码清单 18-29 listBoardTopics.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!--① 引入一个自定义的Tag，该Tag用于生成分页导航的代码-->
<%@taglib prefix="xiaochun" tagdir="/WEB-INF/tags" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>论坛版块页面</title>
    </head>
    <body>
    <%@ include file="includeTop.jsp"%>
    <div>
        <table border="1px" width="100%">
            <tr>
              <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                <td></td>
              </c:if>
                <td bgcolor="#EEEEEE">
                    ${board.boardName}
                </td>
                <td colspan="4" bgcolor="#EEEEEE" align="right">
                <a  href="<c:url value="/board/addTopicPage-${board.boardId}.html"/>">
                  发表新话题
              </a>
                </td>
            </tr>
            <tr>
              <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                <td>
                  <script>
                      function switchSelectBox(){
                         var selectBoxs = document.all("topicIds");
                         if(!selectBoxs) return ;
                         if(typeof(selectBoxs.length)=="undefined"){//only one checkbox
                            selectBoxs.checked = event.srcElement.checked;
                         }else{//many checkbox ,so is a array
                           for(var i = 0 ; i < selectBoxs.length ; i++){
                             selectBoxs[i].checked = event.srcElement.checked;
                           }
                         }
                      }
                  </script>
```

```
                <input type="checkbox" onclick="switchSelectBox()"/>
              </td>
            </c:if>
              <td width="50%">标题</td>
              <td width="10%">发表人</td>
              <td width="10%">回复数</td>
              <td width="15%">发表时间</td>
              <td width="15%">最后回复时间</td>
          </tr>
        <!--② 判断用户是否是该版块的管理员-->
          <c:set var="isboardManager" value="${false}" />
          <c:forEach items="${USER_CONTEXT.manBoards}" var="manBoard">
              <c:if test="${manBoard.boardId == board.boardId}">
                  <c:set var="isboardManager" value="${true}" />
              </c:if>
          </c:forEach>
        <!--③ 对保存在Page对象中的分页数据进行渲染，以显示一页的数据-->
          <c:forEach var="topic" items="${pagedTopic.result}">
              <tr>
                <!--③-1 如果是论坛版块管理员或者论坛管理员，则显示批量操作的复选框-->
                 <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                  <td>
                  <input type="checkbox" name="topicIds" value="${topic.topicId}"/>
                  </td>
                 </c:if>
                  <td>
                      <a  href="<c:url value="/board/listTopicPosts-${topic.
topicId}.html"/>">
                          <!--③-2 如果是精华帖子，则附加★号标志-->
                         <c:if test="${topic.digest > 0}">
                            <font color=red>★</font>
                          </c:if>
                          ${topic.topicTitle}
                      </a>
                  </td>
                  <td>${topic.user.userName}<br><br></td>
                  <td>${topic.replies}<br><br>
                  </td>
                  <td><fmt:formatDate pattern="yyyy-MM-dd HH:mm"
                          value="${topic.createTime}" /></td>
                  <td><fmt:formatDate pattern="yyyy-MM-dd HH:mm"
                          value="${topic.lastPost}" /></td>
              </tr>
          </c:forEach>
      </table>
      </div>
    <!--④ 分页显示导航栏-->
      <xiaochun:PageBar
            pageUrl="/board/listBoardTopics-${board.boardId}.html"
            pageAttrKey="pagedTopic"/>
        <!--⑤ 如果是论坛版块管理员或者论坛管理员，则显示批量操作的按钮-->
        <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
        <script>
          function getSelectedTopicIds(){
             var selectBoxs = document.all("topicIds");
```

```
            if(!selectBoxs) return null;
            if(typeof(selectBoxs.length) == "undefined" && selectBoxs.checked){
               return selectBoxs.value;
            }else{
              var ids = "";
              var split = ""
              for(var i = 0 ; i < selectBoxs.length ; i++){
                 if(selectBoxs[i].checked){
                    ids += split+selectBoxs[i].value;
                    split = ",";
                 }
              }
              return ids;
            }
          }
          function deleteTopics(){
            var ids = getSelectedTopicIds();
            if(ids){
              var url = "<c:url value="/board/removeTopic.html"/>?topicIds="+
                    ids+"&boardId=${board.boardId}";
              location.href = url;
            }
          }
          function setDefinedTopis(){
            var ids = getSelectedTopicIds();
            if(ids){
              var url = "<c:url value="/board/makeDigestTopic.html"/>? topicIds="+
                        ids+"&boardId=${board.boardId}";
              location.href = url;
            }
          }
        </script>
        <input type="button" value="删除" onclick="deleteTopics()">
        <input type="button" value="置精华帖" onclick="setDefinedTopis()">
      </c:if>
    </body>
</html>
```

在 BoardManageController 中，通过调用服务层的 ForumService#getPagedTopics() 方法获取了一页的数据，如果未指定 pageNo 参数，则默认为第一页数据。在获取页数据对象后，将其以“pagedTopic”为名称放置到 Request 属性列表中，并调用 listBoardTopics.jsp 页面渲染这一页的数据。最值得我们关注的可能是<xiaochun:PageBar/>这个自定义标签。

<xiaochun:PageBar/>标签接收两个参数，分别是 pageUrl 和 pageAttrKey。前者是转到其他页面的处理 URL；而后者是分页对象在 Request 属性列表中的键名称，< xiaochun:PageBar />需要根据这个名称从 Request 中获取 Page 对象。

我们使用 JSP 2.0 的 Tag 语法编写<xiaochun:PageBar/>，即在 WEB-INF/tags 目录下创建一个名为 PageBar.tag 的标签页面文件，其标签代码如代码清单 18-30 所示。

代码清单 18-30　PageBar.tag

```
<%@ tag pageEncoding="UTF-8" %>
```

```
<!--① 声明JSTL标签，以便在本标签中使用之-->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!--② 定义了两个标签属性-->
<%@ attribute name="pageUrl" required="true" rtexprvalue="true"
                                          description="分页页面对应的URL" %>
<%@ attribute name="pageAttrKey" required="true" rtexprvalue="true"
                                 description="Page对象在Request域中的键名称" %>
<c:set var="pageUrl" value="${pageUrl}" />
<!--③ 将一些标签中需要引用的对象放置到PageContext属性列表中,
                                          以便后面可以直接通过EL表达式引用之-->
<%
  String separator = pageUrl.indexOf("?") > -1?"&":"?";
  jspContext.setAttribute("pageResult", request.getAttribute(pageAttrKey));
  jspContext.setAttribute("pageUrl", pageUrl);
  jspContext.setAttribute("separator", separator);
%>
<!--④ 构造分页导航栏-->
<div style="font:12px;background-color:#DDDDDD">
    共${pageResult.totalPageCount}页，第${pageResult.currentPageNo}页
    <c:if test="${pageResult.currentPageNo <=1}">
      首页  
    </c:if>
    <c:if test="${pageResult.currentPageNo >1 }">
      <a href="<c:url value="${pageUrl}"/>
       ${separator}pageNo=1">首页
      </a>  
    </c:if>
    <c:if test="${pageResult.hasPreviousPage}">
     <a href="<c:url value="${pageUrl}"/>
     ${separator}pageNo=${pageResult.currentPageNo -1 }">上一页
     </a>  
    </c:if>
    <c:if test="${!pageResult.hasPreviousPage}">
     上一页  
    </c:if>
    <c:if test="${pageResult.hasNextPage}">
     <a href="<c:url value="${pageUrl}"/>
     ${separator}pageNo=${pageResult.currentPageNo +1 }">下一页
     </a>  
    </c:if>
    <c:if test="${!pageResult.hasNextPage}">
     下一页  
    </c:if>
    <c:if test="${pageResult.currentPageNo >= pageResult.totalPageCount}">
      末页  
    </c:if>
    <c:if test="${pageResult.currentPageNo < pageResult.totalPageCount}">
      <a href="<c:url value="${pageUrl}"/>
      ${separator}pageNo=${pageResult.totalPageCount }">末页
      </a>  
    </c:if>
</div>
```

在定义好 PageBar 标签后，不需要定义 TLD 文件，可直接在 JSP 页面中使用该标签，不过在使用之前必须声明标签的前缀：<%@taglib prefix="xiaochun" tagdir="/WEB-INF/tags" %>。

18.8.7 web.xml 配置

至此，论坛应用开发工作已经基本完成，但还未编写 web.xml 这个重要的文件。我们需要在 web.xml 中声明 Spring 配置文件和启动监听器，并配置 Spring 的请求 Servlet。此外，为了避免中文乱码问题，需要提供一个编码转换过滤器。由于论坛应用在 Web 层的 Action 中使用了 Hibernate 延迟加载机制，所以需要配置一个 OpenSessionInViewFilter 过滤器，以便 Hibernate 的 Session 能够在控制器请求方法中保持打开状态。下面是 web.xml 配置文件的所有内容，如代码清单 18-31 所示。

代码清单 18-31 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns=http://java.sun.com/xml/ns/j2ee
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  version="2.4"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <!--① 覆盖default servlet的/，Spring MVC Servlet将处理原来处理静态资源的映射 -->
    <description>小春论坛</description>
    <display-name>xiaochun</display-name>
    <!--② 指定Spring配置文件和初始化监听器-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
    </listener>
    <filter>
    <!--③ 在Web层打开Hibernate Session，以便可以在Web层访问到Hibernate延迟加载的数据-->
    <filter-name>hibernateFilter</filter-name>
    <filter-class>org.springframework.orm.hibernate4.support.OpenSessionInView
Filter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>hibernateFilter</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <!--④论坛登录验证过滤器-->
    <filter>
        <filter-name>forumFilter</filter-name>
        <filter-class>com.smart.web.ForumFilter</filter-class>
    </filter>
    <filter-mapping>
```

```
        <filter-name>forumFilter</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>forumFilter</filter-name>
        <url-pattern>*.jsp</url-pattern>
    </filter-mapping>
    <!--⑤使用Spring的编码转换过滤器，将请求信息的编码统一转换为UTF-8，以避免中文乱码问题-->
    <filter>
        <filter-name>encodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter
</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encodingFilter</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <servlet>
        <servlet-name>xiaochun</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>3</load-on-startup>
    </servlet>
        <!--⑥ 将以.html为后缀的URL由xiaochun Servlet处理-->
    <servlet-mapping>
        <servlet-name>xiaochun</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    <!--⑦ 浏览器不支持PUT、DELETE等方法，由该过滤器将xxx?_method=delete
    或 xxx?_method=put 转换为标准的HTTP DELETE、PUT方法 -->
    <filter>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter
</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <servlet-name>xiaochun</servlet-name>
    </filter-mapping>
    <!--⑧ 网站默认的页面-->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

在⑥处对这个 Servlet 的 URL 路径映射进行定义，在这里让所有以.html 为后缀的 URL 都能被 xiaochun Servlet 截获，进而转由 Spring MVC 框架进行处理。由于浏览器不

支持 PUT、DELETE 等方法，我们在⑦处配置一个 HiddenHttpMethodFilter 过滤器，将 xxx?_method=delete 或 xxx?_method=put 转换为标准的 HTTP PUT、DELETE 方法。

为了避免中文乱码问题，一个不错的解决办法是所有的程序文件都采用 UTF-8 编码方式，因为不但中文系统支持 UTF-8，其他类型的操作系统也支持 UTF-8。如果统一采用 UTF-8，则令人反感的中文乱码问题可以得到有效解决。

18.8.8 Spring MVC 配置

编写好控制器、页面及 web.xml 配置之后，剩下的工作就是在 Spring 控制器配置文件 xiaochun-servlet.xml 中扫描 Web 路径，启动 Spring 控制器注解解析，指定 Spring MVC 的视图解析器及配置控制器异常统一拦截处理，如代码清单 18-32 所示。

代码清单 18-32 xiaochun-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans" …
        xsi:schemaLocation=" …
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd ">

    <!--① 自动扫描com.smart.web包下标注@Controller注解的类控制器类 -->
    <context:component-scan base-package="com.smart.web" />

    <!--② 启动Spring MVC的注解功能，完成请求和注解POJO的映射 -->
    <mvc:annotation-driven/>

    <!--③ 对模型视图名称的解析，在请求时模型视图名称添加前后缀 -->
    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />
    <bean id="multipartResolver"

    class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
        p:defaultEncoding="utf-8" />
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource"
        p:basename="i18n/messages" />

   <!--④ Web异常解析处理-->
   <bean id="exceptionResolver"
       class="com.smart.web.controller.ForumHandlerExceptionResolver">
        <property name="defaultErrorView">
            <value>fail</value>
        </property>
        <property name="exceptionMappings">
            <props>
                <prop key="java.lang.RuntimeException">fail</prop>
            </props>
        </property>
    </bean>
</beans>
```

Spring MVC 为视图名到具体视图的映射提供了多种可供选择的方法。在这里，我们使用 InternalResourceViewResolver，它通过为视图逻辑名添加前后缀的方式进行解析。如视图逻辑名为“login”，将解析为/WEB-INF/jsp/login.jsp；名为“listAllBoards”，将解析为/WEB-INF/jsp/listAllBoards.jsp。

18.9　对 Web 层进行测试

完成 Web 层控制器编写及 Spring MVC 配置之后，接下来开始编写论坛各控制器相应的测试用例。为了提高单元测试的运行效率，使用 Spring 在 org.springframework.mock 包中为一些依赖于容器的接口提供模拟类，这样就可以在不启动容器的情况下执行单元测试。

18.9.1　编写 Web 测试基类

为了简化每个控制器测试，我们编写了一个 Web 测试基类，所有的 Web 测试用例都需要扩展该基类，具体实现如代码清单 18-33 所示。

代码清单 18-33　BaseWebTest.java

```
package com.smart.web.controller;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;
…
@SpringApplicationContext( { "classpath:/applicationContext.xml",
"classpath:/xiaochun-servlet.xml" })
public class BaseWebTest extends UnitilsTestNG {
    //① 根据类型注入Bean
   @SpringBeanByType
    public AnnotationMethodHandlerAdapter handlerAdapter;

    //② 声明模拟对象
    public MockHttpServletRequest request;
    public MockHttpServletResponse response;
    public MockHttpSession session;

    //③ 执行测试前先初始化模拟对象
    @BeforeMethod
    public void before() {
        request = new MockHttpServletRequest();
        response = new MockHttpServletResponse();
        session = new MockHttpSession();
        request.setCharacterEncoding("UTF-8");
    }

}
```

在①处，使用 Unitils 从 Spring 容器中加载 AnnotationMethodHandlerAdapter 实例，用于向控制器发送请求。在②处，声明 Spring 提供的 Servlet API 模拟类 MockHttpServletRequest、MockHttpServletResponse 及 MockHttpSession，并在测试初始化方法中进行实例化，用于发送请求信息及接收响应信息。

18.9.2 编写 ForumManageController 测试用例

编写好 Web 测试基类之后，接下来开始编写每个 Controller 相应的测试用例。在第 20 章中会对 LoginController 测试用例进行讲解，这里不再阐述。下面重点看一下论坛的核心控制器 ForumManageController 测试用例，如代码清单 18-34 所示。

代码清单 18-34 ForumManageControllerTest.java

```
package com.smart.web.controller;
import static org.hamcrest.Matchers.*;
import static org.testng.Assert.*;
…
public class ForumManageControllerTest  extends BaseWebTest {

    //① 注入论坛管理控制器
    @SpringBeanByType
     private ForumManageController controller;

     //② 测试论坛首页处理控制器
    @Test
     public void listAllBoards()throws Exception {
        //②-1设置请求URI及方法
         request.setRequestURI("/index");
         request.setMethod("GET");

        //②-2调用控制器
         ModelAndView mav = handlerAdapter.handle(request, response, controller);
         List<Board> boards = (List<Board>)request.getAttribute("boards");

        //②-3验证结果
         assertNotNull(mav);
         assertEquals(mav.getViewName(), "/listAllBoards");

         assertNotNull(boards);
         assertThat(boards.size(), greaterThan(1));
     }

     //③ 测试跳转添加版块页面
    @Test
     public void addBoardPage()throws Exception {
         //③-1设置请求URI及方法
        request.setRequestURI("/forum/addBoardPage");

        //请求方法要与控制器中@RequestMapping的设置方法一致
         request.setMethod("GET");
```

```
        //③-2验证结果
        ModelAndView mav = handlerAdapter.handle(request, response, controller);
        assertNotNull(mav);
        assertEquals(mav.getViewName(), "/addBoard");
    }
    ...
}
```

从 ForumManageController 测试用例来看，所有测试方法都比较好理解。首先通过模拟类 MockHttpServletRequest 设置请求 URI、参数及请求方法（如 POST、GET）。然后通过 Spring 提供的注解方法处理适配器向测试控制器 ForumManageController 发起请求。最后通过 TestNG 提供的断言及 Hamcrest 提供的匹配方法验证返回的结果。验证的返回结果主要有两个：一是验证返回视图的名称；二是验证返回视图的数据，如②-3 处所示。在编写 Web 控制器测试用例的过程中，唯一需要注意的是，测试方法中设置的请求方法与控制器中@RequestMapping 的设置方法一致。版块管理控制器 BoardManageController 测试用例编写方法与论坛管理控制器一样，这里不再阐述，感兴趣的读者可以参考本书配套网盘中的示例代码。在 IDE 工具中，运行 ForumManageController 测试用例，将看到测试用例可在不启动 Web 容器的情况下顺利执行。

至此，我们全部完成了论坛持久层、服务层、Web 层的开发及单元测试工作，最一步就是部署和运行论坛应用。

18.10　开发环境部署

由于我们是基于 Maven 构建的工程，所以在 pom.xml 文件中配置 Jetty 应用服务器插件，如代码清单 18-35 所示。

代码清单 18-35　spring4x-chapter18 模块pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
   org/xsd/maven-4.0.0.xsd">
…
<build>
       <plugins>
          <!-- Jetty插件 -->
          <plugin>
             <groupId>org.mortbay.jetty</groupId>
             <artifactId>maven-jetty-plugin</artifactId>
             <version>6.1.25</version>
             <configuration><!-配置说明 -->
                   <connectors>
                    <connector implementation="org.mortbay.jetty.nio.SelectChannel
                       Connector">
                       <port>80</port>
```

```
                    <maxIdleTime>60000</maxIdleTime>
                </connector>
            </connectors>
            <contextPath>/forum</contextPath>
            <scanIntervalSeconds>0</scanIntervalSeconds>
        </configuration>
    </plugin>
</plugins>
</build>
</project>
```

在章节模块中的插件节点 Plugins 下双击 jetty:run 或 jetty:run-exploded，将以运行模式启动 Jetty 服务器。如果想要以 Debug 模式运行应用，则通过右键菜单选择 Debug 运行应用即可。启动 Jetty 服务器，在浏览器地址栏中输入 http://localhost/forum/，将看到如图 18-25 所示的页面。

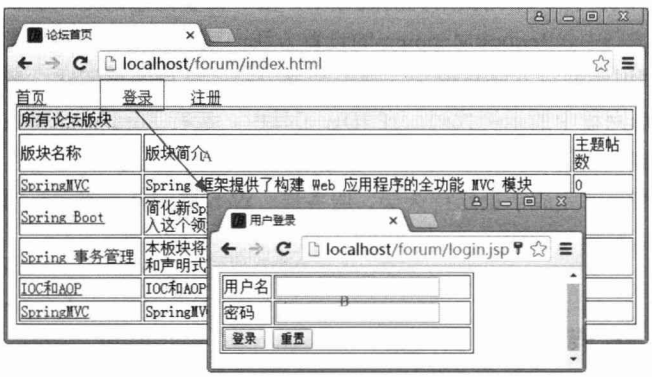


图 18-25　论坛首页及登录页面

单击“登录”链接，转到 B 页面中，登录完成后重新返回论坛首页，这时将看到登录用户的信息，原来的“登录”、“注册”链接变为“注销”链接。

单击论坛版块的链接，转到论坛版块的主题帖子列表页面，列出该版块所有的主题帖子，如图 18-26 所示。

图 18-26　论坛版块页面

单击某一个主题帖子，将看到该主题帖子及其所有回帖的列表，如图 18-27 所示。

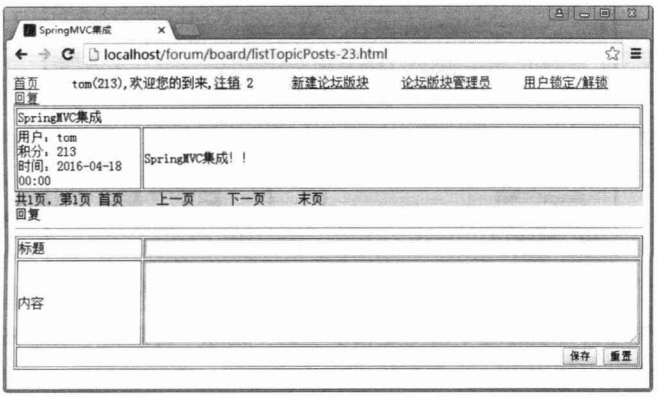


图 18-27　主题帖子页面

在所有回复帖子的下面有一个回复帖子的表单，可以通过这个表单新增回复帖子。如果登录用户是系统管理员，则其对应的操作页面如图 18-28 所示。

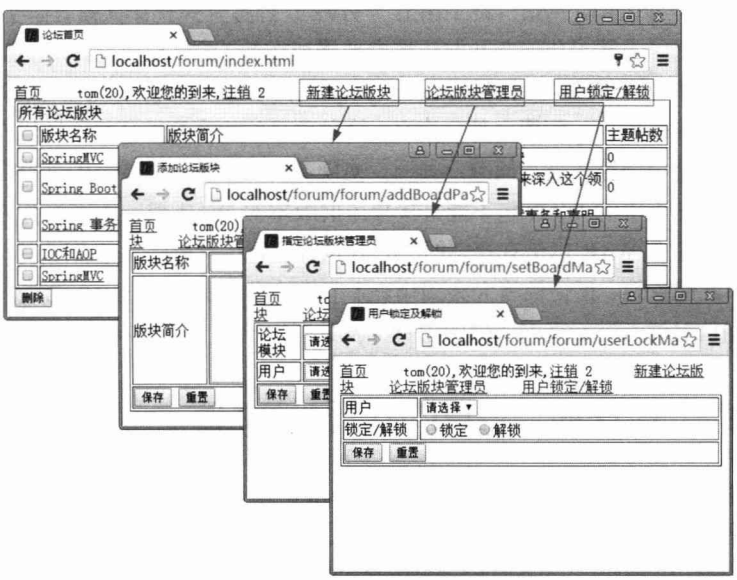


图 18-28　系统管理员的操作页面

论坛管理员登录后，论坛首页的顶部会自动显示出相应操作的链接，如新建论坛版块、论坛版块管理员及用户锁定/解锁，单击这些操作链接可以到达相应的操作页面。

如果用户是某一论坛版块的管理员，则论坛版块的主题帖子列表会自动显示出删除帖子和置精华帖子的操作链接，如图 18-29 所示。

图 18-29 论坛版块管理员所看到的主题帖子列表页面

论坛版块管理员可以通过单击主题帖子下的“删除”链接删除主题帖子，单击“置精华帖”链接将对应的主题帖子置为精华帖子。

18.11 项目配置实战经验

之前我们已经了解了如何在开发环境中部署和运行工程，然而项目的配置文件位于项目的类路径下（如 WEB-INF\classes\conf.properties），数据源也直接采用 DBCP、C3P0 等数据源，直接在 Spring 配置文件中定义。这种项目结构在开发环境下并没有问题，但如果在生产环境中则存在很大问题：其一，实施部署人员需要到 WAR 包中去更改配置信息；其二，新版本的 WAR 包不能直接覆盖旧版本的 WAR 包，否则原有的配置文件会被覆盖。即这个 WAR 包是有状态的，它既包含了程序代码，又包含了生产环境的配置信息，因此配置性、便捷性将大大弱化。本节将给出笔者在实践中总结出来的一个实战方案，相信对读者进行真刀实枪的项目开发具有指导意义。

18.11.1 “传统的”Web 项目属性文件

以下是一个传统的 Web 项目，在其 Spring 配置文件中通过<context:property-placeholder/>引用一个基于类路径的属性文件，如下：

```
<context:property-placeholder location="classpath:conf.properties"/>
<rop:annotation-driven
  core-pool-size="${rop.corePoolSize}"
  max-pool-size="${rop.maxPoolSize}"
  queue-capacity="${rop.queueCapacity}"
  keep-alive-seconds="${rop.keepAliveSeconds}"/>
```

将这个 Web 项目打包成 WAR 包后，在生产环境下如果要更改 conf.properties 配置文件中的参数，则必须在打开的 WAR 包中更改，虽然大部分 Web 应用服务器都会将 WAR 解压到一个文件目录中，但即便如此，对于开发和实施人员来说，仍需担心配置被升级覆盖的问题。

对于开发人员来说，每次打 WAR 包时都需要使用生产环境下的 conf.properties 替换开发环境下的 conf.properties，而开发人员往往并不清楚生产环境的具体情况。此时，开发人员只能请求实施人员不要用这个 WAR 包直接覆盖生产环境下的 WAR 包。

对于实施人员来说，他得战战兢兢地升级 WAR 包，先将生产环境下 WAR 包的 conf.properties 备份出来，然后使用新版 WAR 包，最后再将备份的 conf.properties 覆盖回来，以免生产环境下的 conf.properties 被覆盖了。当然，平常要更改配置参数也很麻烦，需要到每个 WAR 包中去更改。

18.11.2　如何规划便于部署的 Web 项目属性文件

为了方便 Web 项目的部署，必须实现属性文件和 WAR 包解耦，即 WAR 包中不包含配置文件。但这样问题又来了：难道在开发环境下，需要将配置文件放到一个项目之外的目录下？如此一来，项目团队开发岂不是很麻烦：团队成员从 SVN 迁出项目代码后，还需要手工复制项目的配置文件到一个工程外的指定配置文件目录下。

为了使项目既具有开发便捷性（SVN 一迁出工程，mvn jetty:run 即可启动项目），又具有部署便捷性（上传覆盖升级的 WAR 包即可，无须其他动作），只需引入一个 JVM 参数指定配置文件的位置，即可完美地满足开发便捷性和部署便捷性的需求。代码如下：

```
<plugin>
   <groupId>org.mortbay.jetty</groupId>
   <artifactId>maven-jetty-plugin</artifactId>
   <version>6.1.25</version>
   <configuration>
      <webDefaultXml>src/resources/dev/webdefault.xml</webDefaultXml>
      <systemProperties>
            <!-- ①注意这里，我们用一个JVM系统参数指定配置文件的路径 -->
         <systemProperty>
            <name>CONFFILE_PATH</name>
            <value>classpath:conf.properties</value>
         </systemProperty>
      </systemProperties>
      <webAppSourceDirectory>src/main/webapp</webAppSourceDirectory>
      <scanIntervalSeconds>0</scanIntervalSeconds>
      <contextPath>/forum</contextPath>
      <connectors>
         <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
            <port>80</port>
         </connector>
      </connectors>
   </configuration>
</plugin>
```

我们在项目工程 src/resources 中拥有一个项目配置文件 conf.properties，该配置文件是和项目工程在一起的，专门供开发期使用。在 pom.xml 中采用 Jetty 插件，在①处通过 CONFFILE_PATH 的 JVM 系统参数指向类路径下的 conf.properties。

接下来对 Spring 配置文件进行调整，通过这个 CONFFILE_PATH 的 JVM 系统参数引用配置文件。

```
<!-- ① 注意这里，通过Spring EL表达式访问系统参数的值 -->
<context:property-placeholder location="#{systemProperties.CONFFILE_PATH}"/>
<rop:annotation-driven
      core-pool-size="${rop.corePoolSize}"
      max-pool-size="${rop.maxPoolSize}"
      queue-capacity="${rop.queueCapacity}"
      keep-alive-seconds="${rop.keepAliveSeconds}"/>
```

也就是说，在开发环境下，使用工程类路径下的 conf.properties 配置文件。在生产环境下，在 Web 应用服务器中定义 CONFFILE_PATH 的 JVM 系统参数，其值设为一个独立于 WAR 包的系统文件（如 file:/D:/etc/conf.properties），在 WebSphere 中定义 JVM 系统参数的界面如图 18-30 所示。

图 18-30　在 WebSphere 中定义 JVM 系统参数界面

提示

在 WebSphere 中定义 JVM 系统参数的操作步骤为：服务器→应用程序服务器（选择对应的应用程序服务器）→服务器基础结构（Java 和进程管理/进程定义）→其他属性（Java 虚拟机）→其他属性（定制属性）。对于 Tomcat，Jetty 都可在应用服务器的配置文件中定义 JVM 参数。

这样，在生产环境下，Spring 将根据 CONFFILE_PATH 的 JVM 系统参数读取 file:/D:/etc/conf.properties 配置文件。由于配置文件已经放在独立于 WAR 包的外部操作系统的文件目录下，如此一来，生产环境下的 WAR 包升级时，直接覆盖旧版 WAR 包即可，不用再考虑新版 WAR 包覆盖配置文件的问题。

18.11.3　数据源的配置

Web 应用基本上离不开数据源，传统的 Web 项目大多数采用 DBCP、C3P0 等数据源直接在 Spring 配置文件中定义，这样会给部署带来问题。

（1）无法共享数据源。如果在一个 Web 应用服务器中存在多个 WAR 项目，那么这些 WAR 项目都使用同一个数据库。由于采用在各自项目中独立配置数据源的方式，因而会造成创建多个数据源实例的问题，无法实现数据源的共享。

（2）数据源的优化不方便。如果在 Spring 配置文件中配置 DBCP、C3P0 等数据源，则数据源的优化参数将被抽取到配置文件中。由于数据源的优化参数很多，往往并不能将所有的配置参数都抽取到配置文件中，所以数据源的优化将受限。

因此，应让 Spring 引用 JNDI 数据源，而不应在 Spring 配置文件中通过 DBCP、C3P0 等自行定义数据源。

当然，一个很现实的问题是如何在开发环境下使用 JNDI 数据源。其实这已经不是问题了，Tomcat、Jetty 很早就可以定义 JNDI 数据源，并不是只有 WebSphere、WebLogic 才可以定义 JNDI 数据源。

下面是使用 Maven 的 maven-jetty-plugin 插件定义 JNDI 数据源的配置。

```
<plugin>
   <groupId>org.mortbay.jetty</groupId>
   <artifactId>maven-jetty-plugin</artifactId>
   <version>6.1.25</version>
   <configuration>
      <webDefaultXml>src/resources/dev/webdefault.xml</webDefaultXml>
            <!--①添加一个Jetty的额外配置文件 -->
            <jettyEnvXml>src/resources/jetty.xml</jettyEnvXml>
      <systemProperties>
         <systemProperty>
            <name>CONFFILE_PATH</name>
            <value>classpath:conf.properties</value>
         </systemProperty>
      </systemProperties>
      <webAppSourceDirectory>src/main/webapp</webAppSourceDirectory>
      <scanIntervalSeconds>0</scanIntervalSeconds>
      <contextPath>/forum</contextPath>
      <connectors>
         <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
            <port>80</port>
         </connector>
      </connectors>
   </configuration>
</plugin>
```

在①处引用了一个与 pom.xml 位于相同目录的 Jetty 配置文件 jetty.xml，其配置内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
      "http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">

   <New class="org.mortbay.jetty.plus.naming.Resource">
      <Arg>jdbc/myds</Arg>
      <Arg>
         <New class="org.apache.commons.dbcp.BasicDataSource">
```

```
            <Set name="driverClassName">com.mysql.jdbc.Driver</Set>
            <Set name="url">
              <![CDATA[jdbc:mysql://localhost:3306/sampledb]]>
            </Set>
            <Set name="username">root</Set>
            <Set name="password">123456</Set>
          </New>
        </Arg>
    </New>
</Configure>
```

这里定义了一个名为 jdbc/myds 的 JNDI 数据源，这样项目的 Spring 配置文件就可直接引用这个 JNDI 数据源了。

```
<?xml version="1.0" encoding="UTF-8" ?>
    ...
    <jee:jndi-lookup id=" dataSource"  jndi-name="java:comp/env/jdbc/myds"/>
</beans>
```

由于 JNDI 数据源名称不是固定的，比如在某个部署环境下可能为 jdbc/yourds，因此需要将 JNDI 的名字属性化。结合上面所说的配置文件，可以在配置文件中添加一个属性，暂且将该属性命名为 ds.jndi。

```
conf.properties
ds.jndi=java:comp/env/jdbc/yourds
```

然后在 Spring 配置文件中通过参数名引用 JNDI 数据源。

```
<?xml version="1.0" encoding="UTF-8" ?>
    ...
    <context:property-placeholder location="#{systemProperties.CONFFILE_PATH}"/>
    <jee:jndi-lookup id="dataSource" jndi-name="${ds.jndi}"/>
</beans>
```

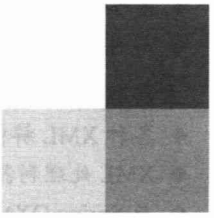
至此，开发和部署人员都皆大欢喜了。在 JNDI 数据源已经存在的情况下，只需在 conf.properties 中定义好 JNDI 数据源名称，项目即可启动。

18.12 小结

在本章中，我们开发了一个实际论坛的应用案例，该案例的技术框架采用 Spring+ Hibernate 的组合框架。我们从需求、设计、开发、单元测试、部署（开发环境和生产环境部署）典型软件开发过程讲解论坛应用的整体开发过程。当然，由于篇幅的限制，还留有一些未尽的工作，如严格的权限控制、用户帖子管理、投票帖子等，有兴趣的读者可以在本案例的基础上进一步完善这个论坛，让它更接近实际应用。

第5篇

提高篇



第 19 章

Spring OXM

本章从 XML 各种解析技术的发展历程讲起，并介绍另外一些主流 O/X Mapping 组件的使用方法。通过这些实例讲解，读者可以快速了解目前各种 O/X Mapping 技术。本章筛选了目前一些流行的 O/X Mapping 组件，如 XStream、Castor、JiBX、JAXB 等，从独立使用到与 Spring 整合逐步演进，逐步揭开各 O/X Mapping 组件的神秘面纱。通过本章的学习，读者可以根据需要，选用合适的 O/X Mapping 组件来处理对象 XML 之间的转换，为开发 Web Services 应用打下良好的基础。

本章主要内容：

- 认识 XML 解析技术
- XML 处理利器：XStream
- 基于 Castor 对象的 XML 映射
- 基于 JiBX 对象的 XML 映射
- 基于 JAXB 对象的 XML 映射
- 与 Spring OXM 整合

本章亮点：

- 各种 XML 解析技术特点分析
- XML 处理利器：XStream 及其他 O/X Mapping 组件讲解
- 与 Spring OXM 整合

19.1 认识 XML 解析技术

19.1.1 什么是 XML

XML（Extensible Markup Language）即可扩展标记语言，从 1996 年开始有其雏形，

并向 W3C（全球信息网联盟）提案，于 1998 年 2 月发布为 W3C 的标准（XML 1.0）。XML 的前身是 SGML（the Standard Generalized Markup Language），是 IBM 从 20 世纪 60 年代就开始发展的 GML（Generalized Markup Language）标准化后的名称。其目的是促进 Internet 上结构化文档的交换。简单地说，XML 是一组规则和准则的集合，用来描述结构化数据。

XML 为描述结构良好的文档提供了一整套灵活的语法。正因为它的这种灵活性，我们需要一些方法来验证 XML 文档是否与我们预计的格式保持一致。于是人们就逐步提出 DTD 和 XML Schema。

DTD 是一套关于标记符的语法规则，它是 XML 1.0 规范的一部分，是 XML 文件的验证机制，属于 XML 文件组成的一部分。DTD 是一种保证 XML 文档格式正确的有效方法，可以通过比较 XML 文档和 DTD 文件来看文档是否符合规范，以及元素和标签使用是否正确。

XML Schema 指定 XML Schema 定义语言，该语言提供了描述 XML 1.0 文档结构和限制其内容的工具，其中包括那些利用 XML Namespace 的工具。模式语言自身用 XML 1.0 表示并使用名称空间，它在很大程度上重构了 XML 1.0 DTD 的能力，并在一定程度上解决了 DTD 的许多局限性，如：

- 不支持名称空间。
- 对模块化和重用的支持非常有限。
- 不支持对声明的扩展或继承。
- 编写、维护和读取大型 DTD 以及定义系列相关模式都很困难。
- 没有嵌入式、结构化自我文档编制。
- 内容和属性声明不能依靠属性或元素上下文。

XML 不仅仅是一种标记语言，而是一系列技术，如 DTD、XSD、DOM、SAX、XSL、XLink、XPointer、SMIL 等的集合体。这一技术家族为我们开发可扩展性和互操作性的软件提供了一种解决方案，目前广泛应用于系统配置、数据存储及数据交换的格式等，它是 Web 服务的基础，也是现代面向服务的架构（Service-Oriented Architecture，SOA）设计模式的基础。

19.1.2　XML 的处理技术

为了有效使用 XML，需要通过一个 XML 处理器或 XML API 来访问其数据。目前 JAXP 1.6（JSR 206）的两种处理 XML 文档的方法已经得到广泛应用，分别是 DOM（Document Object Model）和 SAX（Simple API for XML）。

DOM 文档对象模型是一种通过编程方式对 XML 文档中的数据及结构进行访问的标准，基于 XML 文档在内存中的树状结构，当一个 XML 文件被装入处理器时，内存中建立起一棵相应的树。DOM 还定义了用来遍历一棵 XML 树及管理各个元素、值和

属性的编程接口（包括方法和属性的名字）。

DOM 标准的一个主要不足在于将整个 XML 文档装入内存所引起的巨大内存开销。当文件的数据量非常大时，这会带来很大的性能瓶颈。于是人们就开始创立一种新的标准，这就是 SAX。

SAX 是一种非常简单的 XML API，它允许开发者使用事件驱动的 XML 解析。与 DOM 不同，SAX 并不要求将整个 XML 文件一起装入内存。它的想法十分简单，一旦 XML 处理器完成对 XML 元素的操作，它就立刻调用一个自定义事件处理器及时处理这个元素及相关数据。

虽然 SAX 解决了 DOM 速度慢、内存占用大的问题，但在灵活性方面受到很大制约，如无法随机访问文档。于是又有一种新的基于流的 Stream API for XML（简称 StAX）逐步出现在人们的视野中，它不仅提高了 XML 的处理速度，而且较好地兼顾了灵活性。StAX 是 JSR 173 标准，目前已经加入 Java 6.0 的 JAXP 1.4 里面。

StAX 如其名称所暗示的那样，把重点放在流上。实际上，StAX 与其他方法的区别就在于应用程序能够把 XML 当作一个事件流来处理。将 XML 当作一个事件流来处理的想法并不新颖（事实上 SAX 已经提出来了），但 StAX 并不使用 SAX 的“推”模型，而是使用“拉”模型进行事件处理。此外，StAX 解析器也不使用回调机制，而是根据应用程序的要求返回事件。StAX 还提供了用户友好的 API，用于读入和写出。

StAX 实际上包括两套处理 XML 的 API，分别提供了不同程度的抽象。基于指针的 API 允许应用程序把 XML 作为一个标记（或事件）流来处理。应用程序可以检查解析器的状态，获得解析的上一个标记信息，然后再处理下一个标记，依此类推。这是一种低层 API，尽管效率高，但是没有提供底层 XML 结构的抽象。较为高级的基于迭代器的 API 允许应用程序把 XML 作为一系列事件对象来处理，每个对象和应用程序交换 XML 结构的一部分。应用程序只需确定解析事件的类型，将其转换成对应的具体类型，然后利用其方法获得属于该事件的信息即可。

DOM、SAX、StAX 技术都是从 XML 的角度来处理文档和建立模型，这对于只关注文档 XML 结构的应用程序来说是适用的，但是很多应用程序仅仅将 XML 作为数据交换的媒介，更多关注的是文档数据本身，为此人们又提出了一种 XML 数据绑定技术，可以使应用程序在很大程度上忽略 XML 文档的实际结构，而直接使用文档的数据内容。

数据绑定是指将数据从一些存储媒介（如 XML 文档和数据库）中取出，并通过程序表示这些数据的过程，即把数据绑定到虚拟机能够理解并且可以操作的某种内存结构中。数据绑定并不是一个新鲜的概念，它在关系数据库上早已得到了广泛的应用，如 Hibernate 就是针对数据库的轻量级数据绑定框架。而针对 XML 数据绑定的 Castor 框架在 2000 年就已经出现，目前已经涌现出许多类似的框架，如 JAXB、JiBX、Quick 和 Zeus 等。

19.2　XML 处理利器：XStream

19.2.1　XStream 概述

XStream 是一套简洁易用的开源类库，用于将 Java 对象序列化为 XML 或者将 XML 反序列化为 Java 对象，是 Java 对象和 XML 之间的一个双向转换器。2004 年 1 月发布了第一个版本 XStream 0.3，2016 年 3 月发布了 XStream 1.4.9 版本。

1．XStream 主要特点

- 灵活易用：提供了简单、灵活、易用的统一接口，用户无须了解底层细节。
- 无须映射：大多数对象都可以在无须映射的情况下进行序列化与反序列化的操作。
- 高速稳定：解析速度快，占用内存少。
- 灵活转换：转换策略是可以定制的，允许自定义类型存储为指定的 XML 格式。
- 易于集成：通过实现特定的接口，可以直接与其他任何树形结构进行序列化与反序列化操作。

2．XStream 架构组成

1）Converters（转换器）

当 XStream 遇到需要转换的对象时，它会委派给合适的转换器实现。XStream 为通用类型提供了多种转换器实现，包括基本数据类型、String、Collections、Arrays、null、Date 等。

2）I/O（输入/输出）

XStream 是通过接口 HierarchicalStramWriter 和 HierarchialStreamReader 从底层 XML 数据中抽象而来的，分别用于序列化和反序列化操作。

3）Context（上下文引用）

在 XStream 序列化或反序列化对象时，它会创建两个类 MarshallingContext 和 UnmarshallingContext，由它们来处理数据并委派合适的转换器。XStream 提供了三种上下文的默认实现，它们之间存在细微的差别。默认值可以通过 XStream.setMode()方法调整，可选值如下。

（1）XStream.XPATH_REFERENCES：（默认的）通过 XPath 引用来标识重复的引用。

（2）XStream.ID_REFERENCES：使用 ID 引用来标识重复的引用。

（3）XStream.NO_REFERENCES：对象作为树形结构，重复的引用被视为两个不同的对象，循环引用会导致异常产生。这种模式速度快，占用内存少。

4）Facade（统一入口）

作为 XStream 的统一入口点，它将上面所提及的重要组件集成在一起，以统一的接口开放出来。

19.2.2 快速入门

本小节通过一个实例开始 XStream 的应用之旅，采用 User 用户领域对象及 LoginLog 登录日志领域对象作为 XStream 实例转换对象，如代码清单 19-1 和代码清单 19-2 所示。

代码清单 19-1 User.java

```
package com.smart.oxm.domain;
…
public class XStreamSample {
    private int userId;
    private String userName;
    private String password;
    private int credits;
    private String lastIp;
    private Date lastVisit;
    private List logs;

    //set、get方法省略
}
```

代码清单 19-2 LoginLog.java

```
package com.smart.oxm.domain;
…
public class XStreamSample {
    private int loginLogId;
    private int userId;
    private String ip;
    private Date loginDate;

    //set、get方法省略
}
```

在开始实例的开发之前，要先将 XStream 的依赖加入 pom.xml 文件中，本实例采用的是 XStream 1.4.9 版本。代码如下：

```
<dependency>
   <groupId>com.thoughtworks.xstream</groupId>
   <artifactId>xstream</artifactId>
   <version>1.4.9</version>
</dependency>
```

接下来开始使用 XStream 进行对象与 XML 之间的互换，如代码清单 19-3 所示。

代码清单 19-3 XStreamSample.java

```
package com.smart.oxm.xstream;
…
public class XStreamSample {
    private static XStream xstream;
    static{
      //① 创建XStream实例并指定一个XML解析器
      xstream = new XStream(new DomDriver());
    }

   //初始化转换对象
```

```
    public static User getUser(){
        LoginLog log1 = new LoginLog();
        log1.setIp("192.168.1.91");
        log1.setLoginDate(new Date());
        User user = new User();
        user.setUserId(1);
        user.setUserName("xstream");
        user.addLoginLog(log1);
        return user;
    }

    //② Java对象转换为XML
    public static void objectToXML()throws Exception{

        //②-1 获取转换的User对象实例
        User user = getUser()

        //②-2 实例化一个文件输出流
        FileOutputStream outputStream = new FileOutputStream("out/XStreamSample.xml");
        //②-3 将User对象转换为XML，并保存到指定文件
        xstream.toXML(user, outputStream);
    }

    //③ XML转换为Java对象
    public static void XMLToObject()throws Exception{

        //③-1 实例化一个文件输入流
        FileInputStream inputStream= new FileInputStream("out/XStreamSample.xml ");

        //③-2 将XML文件输入流转换为对象
        User user = (User) xstream.fromXML(inputStream);

        for(LoginLog log : user.getLogs()){
          if(log!=null){
              System.out.println("访问IP: " + log.getIp());
              System.out.println("访问时间: " + log.getLoginDate());
          }
        }
    }
  …
}
```

在①处，创建一个 XStream 实例，并指定一个 DOM XML 解析器。如果不指定一个 XML 解析器，则 XStream 会采用默认的 XPP（XML Pull Parser，一种高速解析 XML 文件的方式，速度要比传统方式快很多）解析器来解析 XML 文件。

在②-1 处，首先实例化需要转换为 XML 的类，如实例中的 User、LoginLog。在②-2 处，创建一个用于输出 XML 文件的文件输出流。在②-3 处，调用 XStream#toXML() 方法将 User 实例转换为 XML 格式，并转出到指定的文件流中。同理，如果将 XML 文件转换为对象，则需要先读取要转换的 XML 文件到文件流中，如③-1 处所示，然后调用 XStream#fromXML()方法将输入文件流转换为对象。在 IDE 工具中执行当前实例，在类路径下生成一个 XML 文件，如下所示：

```
<com.smart.oxm.domain.User>
  <userId>1</userId>
  <userName>xstream</userName>
  <credits>0</credits>
  <logs>
    <com.smart.oxm.domain.LoginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.91</ip>
      <loginDate>2016-03-06 21:05:37.879 UTC</loginDate>
    </com.smart.oxm.domain.LoginLog>
    <com.smart.oxm.domain.LoginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.92</ip>
      <loginDate>2016-03-06 21:05:37.879 UTC</loginDate>
    </com.smart.oxm.domain.LoginLog>
  </logs>
</com.smart.oxm.domain.User>。
```

19.2.3 使用 XStream 别名

在默认情况下，Java 对象到 XML 的映射是 Java 对象属性名对应 XML 的元素名，Java 类的全名对应 XML 根元素的名字。在实际应用中，如果 XML 和 Java 类都已经存在相应的名字，那么，在进行转换时，需要设置别名进行映射。

XStream 别名配置包含三种情况。

- 类别名：用 alias(String name, Class type)。
- 类成员别名：用 aliasField(String alias, Class definedIn, String fieldName)。
- 类成员作为属性别名：用 aliasAttribute(Class definedIn, String attributeName, String alias)。单独命名没有意义，还要通过 useAttributeFor(Class definedIn, String fieldName) 应用到某个类上。

从上一个实例生成的 XML 文件来看，生成的 XML 元素结构不是很友好。接下来使用 XStream 提供的别名机制来修饰生成 XML 元素的结构，如代码清单 19-4 所示。

代码清单 19-4　XStreamAliasSample.java

```
package com.smart.oxm.xstream.alias;
…
public class XStreamAliasSample{
…

    static{
      …

      //① 设置类别名，默认为当前类名称加上包名
      xstream.alias("loginLog", LoginLog.class);
      xstream.alias("user", User.class);
```

```
        //② 设置类成员别名
        xstream.aliasField("id",User.class,"userId");

        //③ 把LoginLog的userId属性视为 XML属性，默认为XML的元素
        xstream.aliasAttribute(LoginLog.class, "userId","id");
        xstream.useAttributeFor(LoginLog.class, "userId");

        //④ 去掉集合类型生成XML的父节点，即忽略XML中的 <logs></logs>标记
        xstream.addImplicitCollection(User.class, "logs");
    }
    …
}
```

在①处，通过 XStream#alias ()方法来设置类别名，默认为当前类的全限定名。在②处，通过 XStream#aliasField()方法将 User 类的 userId 属性设置为 id。在③处，通过 XStream#aliasAttribute()和 XStream#useAttributeFor()方法将 LoginLog 类的 userId 属性设置为 id，并设置为 LoginLog 元素的属性，默认为 LoginLog 元素的子元素。在④处，通过 XStream#addImplicitCollection()方法删除集合节点 logs，即忽略 XML 中的<logs></logs>标记。

在 IDE 工具中执行当前实例，在类路径下生成一个 XML 文件，如下所示：

```
<user>
  <id>1</id>
  <userName>xstream</userName>
  <credits>0</credits>
  <loginLog id="0">
    <loginLogId>0</loginLogId>
    <ip>192.168.1.91</ip>
    <loginDate>2016-03-17 05:12:39.826 UTC</loginDate>
  </loginLog>
  …
</user>
```

19.2.4　XStream 转换器

在开发过程中，有时可能需要转换一些自定义类型，此时默认的映射方式可能无法满足需要。不用担心，XStream 已经提供了丰富的扩展点，用户可以实现自己的转换器，然后调用 registerConverter()方法注册自定义的转换器。实现自定义的转换器很简单，只要实现 XStream 提供的 Converter 接口并实现其方法即可。

上一个实例已成功地应用 XStream 进行对象与 XML 的相互转换，并应用 XStream 提供的别名机制设置生成 XML 的结构。下面使用 XStream 提供的转换器扩展接口，对生成的 XML 文件继续优化。

首先需要实现 Converter 接口来编写一个日期转换器，具体的实现如代码清单 19-5 所示。

代码清单 19-5 DateConverter.java

```
package com.smart.oxm.xstream.converters;
…
public class DateConverter implements Converter {
    private Locale locale;
    public DateConverter(Locale locale) {
        super();
        this.locale = locale;
    }
    //①实现该方法，判断要转换的类型
    public boolean canConvert(Class clazz) {
        return Date.class.isAssignableFrom(clazz);
    }

    //②实现该方法，编写Java对象到XML的转换逻辑
    public void marshal(Object value, HierarchicalStreamWriter writer,
                MarshallingContext context){
        DateFormat formatter = DateFormat.getDateInstance(DateFormat.DATE_FIELD,
                                                          this.locale);
       writer.setValue(formatter.format(value));
    }

    //③实现该方法，编写XML到Java对象的转换逻辑
    public Object unmarshal(HierarchicalStreamReader reader,
    UnmarshallingContext context) {
        GregorianCalendar calendar = new GregorianCalendar();
        DateFormat formatter = DateFormat.getDateInstance(DateFormat.DATE_FIELD,
                                                          this.locale);
        try {
            calendar.setTime(formatter.parse(reader.getValue()));
        } catch (ParseException e) {
            throw new ConversionException(e.getMessage(), e);
        }
        return calendar.getGregorianChange();
    }
}
```

在①处，通过实现 ConverterMatcher#canConvert()方法来判断转换逻辑。在②处，通过实现 Converter#marshal()方法，完成对象编组（对象转换为 XML）操作时的处理逻辑。在③处，通过实现 Converter#unmarshal()方法，完成对象反编组（XML 转换为对象）操作时的处理逻辑。编写好转换器之后，剩下的工作就是调用 XStream#registerConverter()方法注册自定义的日期转换器。

在 IDE 工具中执行当前实例，将会在类路径下生成一个 XML 文件，如下所示：

```
<user>
    <userId>1</userId>
    <userName>xstream</userName>
    <credits>0</credits>
    <logs>
        <loginLog>
    <loginLogId>0</loginLogId>
    <userId>0</userId>
    <ip>192.168.1.91</ip>
```

```
    <loginDate> 2016年3月6日 星期六 </loginDate>
        </loginLog>
        …
    </logs>
</user>
```

19.2.5　XStream 注解

XStream 不但可以通过编码的方式对 XML 进行转换，而且支持基于注解的方式进行转换。下面使用 XStream 提供的注解（见表 19-1）来改造上面的实例。首先需要对 User 和 LoginLog 添加相应的注解，如代码清单 19-6 所示。

代码清单 19-6　User.java

```
package com.smart.oxm.xstream.annotations;
…
@XStreamAlias("user")
public class User {
    @XStreamAsAttribute
    @XStreamAlias("id")
     private int userId;

     @XStreamAlias("username")
     private String userName;

     @XStreamAlias("password")
     private String password;

     @XStreamAlias("credits")
     private int credits;

     @XStreamAlias("lastIp")
     private String lastIp;

    @XStreamConverter(DateConverter.class)
     private Date lastVisit;

     @XStreamImplicit
     private List logs;
     …
}
```

表 19-1　XStream常用注解

注　解	说　明	作用目标
@XStreamAlias	别名注解	类，字段
@XStreamAsAttribute	转换成属性	字段
@XStreamOmitField	忽略字段	字段
@XStreamConverter	注入转换器	对象
@XStreamImplicit	隐式集合	集合字段

其中，@XStreamAlias 为别名注解，一般作用于目标类或字段，如实例中的@XStreamAlias("user")、@XStreamAlias("id")。@XStreamConverter 注解用于注入自定义的转换器，如实例中的@XStreamConverter(DateConverter.class)，注入一个日期转换器。对于集合类型，可以使用@XStreamImplicit 注解来隐藏，其作用与 XStream#addImplicitCollection()方法一样。@XStreamAsAttribute 注解将 Java 对象属性映射为 XML 元素的一个属性。@XStreamOmitField 注解标注 Java 对象的属性将不再出现在 XML 中。LoginLog 对象添加注解的方式与 User 一样，这里不再阐述。下面来看一下如何应用 XStream 提供的注解进行 Java 对象与 XML 的相互转换，如代码清单 19-7 所示。

代码清单 19-7　XStreamAnnotationSample.java

```
package com.smart.oxm.xstream.annotations;
package com.smart.oxm.xstream.annotations;
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.io.xml.DomDriver;
…
public class XStreamAnnotationSample {
    private static XStream xstream;
    …
    static{
        xstream = new XStream(new DomDriver());

        //① 实现该方法，判断要转换的类型
        xstream.processAnnotations(User.class);
        xstream.processAnnotations(LoginLog.class);
        //② 自动加载注解Bean
        //xstream.autodetectAnnotations(true);
    }

   //Java对象转换为XML
   public static void objectToXml()throws Exception{
       User user = getUser();
       FileOutputStream fs = new FileOutputStream("out/XStreamAnnotationSample.xml");
       OutputStreamWriter writer = new OutputStreamWriter(fs, Charset.forName("UTF-8"));
       xstream.toXML(user, writer);
   }
   …
}
```

要启用 XStream 提供的注解功能，需要在执行对象与 XML 转换之前先注册标注了 XStream 注解的 Java 对象，如实例中在①处通过 XStream#processAnnotations()方法手工注册 User 和 LoginLog 对象。XStream 除了手工注册外，还提供一个自动检测标注了 XStream 注解的 Java 对象的方法 XStream#autodetectAnnotations()，如实例中的②处所示。自动检测方法不仅使用方便，而且还提供了缓存机制来缓存所有标注了 XStream 注解的 Java 对象。

19.2.6　流化对象

XStream 为 java.io.ObjectInputStream 和 java.io.ObjectOutputStream 提供替代的实现，允许以对象流方式进行 XML 序列化或反序列化操作。这对于处理集合对象非常有用（List<User> users），在内存中只保留一个 User 对象流。很显然，我们应该使用基于流而非 DOM 的 XML 解析器读取 XML，以提高性能。

创建一个输出流，至于怎么输出可以使用多种方法，其原理是一样的。在这里就不得不提到 HierarchicalStreamWriter。HierarchicalStreamWriter 是一个接口，从字面意思上来说它是有层级关系的输出流。XStream 默认提供了几个常用的实现类用于输出，如 CompactWriter、PrettyPrintWriter。下面应用 XStream 流化对象来处理 XML 序列化及反序列化，如代码清单 19-8 所示。

代码清单 19-8　ObjectStreamSample.java

```
package com.smart.oxm.xstream;
…
public class ObjectStreamSample {
    private static  XStream xstream;
    static{
        xstream = new XStream();
    }

    //Java对象转换为XML
    public void objectToXml() throws Exception{
       //① 实例化序列化对象
        User user = getUser();

       //② 创建一个PrintWriter对象，用于输出
       PrintWriter pw=new PrintWriter("out/ObjectStreamSample.xml");

       //③ 选用一个HierarchicalStreamWriter的实现类来创建输出
        PrettyPrintWriter ppw=new PrettyPrintWriter(pw);
        //CompactWriter cw=new CompactWriter(pw);

       //④ 创建对象输出流
        ObjectOutputStream out = xstream.createObjectOutputStream(ppw);
        out.writeObject(user);
        out.close();
    }

    //XML转换为Java对象
    public User xmlToObject()throws Exception{

        //⑤ 通过对象流进行输入操作
        FileReader reader=new FileReader("out/ObjectStreamSample.xml");
        BufferedReader bufferedReader =new BufferedReader(reader);

        //⑥ 创建对象输入流
        ObjectInputStream input = xstream.createObjectInputStream(bufferedReader);
```

```
        //⑦ 从XML文件中读取对象
        User user=(User)input.readObject();
        return user;
    }
    …
}
```

通过对象流进行输出操作，首先需要实例化转换的对象，如实例中的①处所示。然后在②处创建一个 PrintWriter 对象，将对象序列化到指定的 XML 文件中。在③处，选用一个 HierarchicalStreamWriter 的实现类 PrettyPrintWriter 来创建输出，也可以采用 CompactWriter 实现类。CompactWriter 与 PrettyPrintWriter 的区别在于，用 CompactWriter 方法输出的是连续的没有分隔的 XML 文件，而用 PrettyPrintWriter 方法输出的是有分隔的有一定格式的 XML 文件。在④处，调用 XStream#createObjectOutputStream()方法创建对象输出流。

通过对象流进行输入操作，需要使用 FileReader 和 BufferedReader 读取指定的 XML 文件，如实例中的⑤处所示。在⑥处，调用 XStream#createObjectInputStream()方法创建对象输入流，最后从该输入流读取对象。

19.2.7 持久化 API

如果需要将一个集合中的所有对象持久化到文件系统中，通常会使用 java.io 的 API 将集合的对象逐个输入到文件中。虽然这个方法很简单，但过程比较复杂。XStream 提供了相关集合类的接口实现类，如 XmlArrayList、XmlSet、XmlMap，用一个简单的方法就可以将集合中的每个对象持久化到文件中。下面应用 XmlArrayList 来持久化一个集合中的所有对象，具体实现如代码清单 19-9 所示。

代码清单 19-9　PersistenceSample.java

```
package com.smart.oxm.xstream. persistence;
…
public class PersistenceSample {
    …
    public void persist() {

        //① 实例化需要持久化的对象
        users = new ArrayList<User>();
        User user = new User();
        user.setUserId(1);
        user.setCredits(10);
        user.setUserName("tom");
        user.setPassword("123456");
        users.add(user);

        //② 创建持久化策略
        File file = new File("out");
        PersistenceStrategy strategy = new FilePersistenceStrategy(file);
```

```
        //③ 持久化集合对象
         List list = new XmlArrayList(strategy);
         list.addAll(users);
    }
   …
}
```

XStream 为 XmlArrayList、XmlSet、XmlMap 实现类提供统一的创建接口，在创建 XmlArrayList（及相关的集合）时，需要指定一个持久化策略 PersistenceStrategy。XStream 提供一个持久化到文件的策略 FilePersistenceStrategy。这个策略将集合中的每个对象持久化到指定目录不同的文件中，如 int@0.xml、int@1.xml。

19.2.8　额外功能：处理 JSON

目前，在 Web 开发领域，主要的数据交换格式有 XML 和 JSON，相信每个 Web 开发者对二者都不会感到陌生。

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于阅读和编写，同时也易于机器解析和生成。它是基于 1999 年颁布的 ECMA-262 标准的 JavaScript 语言的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于高级语言的一些习惯（如 Java、C#等），这些特性使 JSON 成为理想的数据交换语言。虽然目前 XML 是业界数据交换标准，在可扩展性、数据类型的描述方面有着明显的优势，但相比于 JSON 这种轻量级的数据交换格式，XML 显得有些“笨重”。

对于大多数 Web 应用来说，根本不需要复杂的 XML 来传输数据，XML 宣称的扩展性在此没有多大优势。大多数 AJAX 应用直接使用 JSON 发送和接收数据，以构建动态页面的内容，非常灵活易用。如果使用 XML，则程序代码将会复杂不少。当然，在 Web Service 领域中，XML 目前仍有不可动摇的地位。

在 Java 的世界里有不少处理 Java 对象与 JSON、XML 相互转换的组件，如 JSON-lib、XMLBeans。但同时支持两种数据格式转换的轻量级组件并不多，XStream 组件就是少数中的佼佼者。

XStream 的 JettisonMappedXmlDriver 和 JsonHierarchicalStreamDriver 都可以很好地完成 Java 对象和 JSON 的相互转换工作。值得注意的是，在使用 JettisonMappedXmlDriver 时，必须事先将 jettison 依赖包添加到工程 pom 中，如下：

```
<dependency>
    <groupId>org.codehaus.jettison</groupId>
    <artifactId>jettison</artifactId>
    <version>1.3.2</version>
 </dependency>
```

下面应用这两个驱动来将对象转换为 JSON，如代码清单 19-10 所示。

代码清单 19-10　XStreamJSONSample.java

```
package com.smart.oxm.xstream.json;
…
```

```
public class XStreamJSONSample {
    private XStream xstream;
    …
    //①连续的没有分隔的JSON串
    public static void toJsonByJettisonMappedXmlDriver()throws Exception {
       User user = getUser();
        FileOutputStream outputSteam = new
              FileOutputStream("out/JettisonMappedSample.json");
        OutputStreamWriter writer = new
                        OutputStreamWriter(outputSteam,Charset.forName("UTF-8"));
        xstream = new XStream(new JettisonMappedXmlDriver());
        xstream.setMode(XStream.NO_REFERENCES);
        xstream.alias("user", User.class);
        xstream.toXML(user,writer);
    }

    //② 格式化良好的JSON串
    public void toJsonByJsonHierarchicalStreamDriver()throws Exception {
       User user = getUser();
        FileOutputStream outputSteam = new
                        FileOutputStream("out/JsonByJsonHierarchicalSample.json");
       OutputStreamWriter writer = new
                       OutputStreamWriter(outputSteam, Charset.forName("UTF-8"));
        xstream = new XStream(new JsonHierarchicalStreamDriver());
        xstream.alias("user", User.class);
        xstream.toXML(user,writer);
    }
    …
}
```

JSON 的转换和 XML 的转换用法一样，只需在创建 XStream 实例时，传递一个 XML 到 JSON 映射转换的驱动器如 JettisonMappedXmlDriver、JsonHierarchicalStreamDriver 即可。使用 JettisonMappedXmlDriver 驱动生成的是连续的没有分隔的 JSON 串，而使用 JsonHierarchical StreamDriver 驱动生成一个格式化后的 JSON 串。如果将 JSON 转换为对象，则只能使用 JettisonMappedXmlDriver 驱动。

在 IDE 工具中执行当前实例，在类路径下生成两个 JSON 文件。使用 JettisonMapped XmlDriver 驱动输出的 JSON 如下：

```
{"user":{"userId":1,"userName":"xstream","credits":0,"logs":[{"com.smart.oxm.domai
n.LoginLog":[{"loginLogId":0,"userId":0,"ip":"192.168.1.91","loginDate":"2016-03-1
3 15:56:30.517
UTC"},{"loginLogId":0,"userId":0,"ip":"192.168.1.92","loginDate":"2016-03-13
15:56:30.517 UTC"}]}]}}
```

而使用 JsonHierarch icalStreamDriver 驱动输出的 JSON 如下：

```
{"user": {
  "userId": 1,
  "userName": "xstream",
  "credits": 0,
  "logs": [
    {
      "loginLogId": 0,
      "userId": 0,
```

```
    "ip": "192.168.1.91",
    "loginDate": "2016-03-13 15:41:27 "
  },

  {
    "loginLogId": 0,
    "userId": 0,
    "ip": "192.168.1.92",
    "loginDate": "2016-03-13 15:41:27 "
  }
 ]
}}
```

19.3　其他常见的 O/X Mapping 开源项目

19.3.1　JAXB

JAXB(Java Architecture for XML Binding)是一个业界的标准，是一项可以根据 XML Schema 产生 Java 类的技术。JAXB 也提供了将 XML 文件反向生成 Java 对象树的方法，并能将 Java 对象树的内容重新写到 XML 文件。从另一方面来讲，JAXB 提供了快速而简便的方法将 XML 模式绑定到 Java 对象，从而使得开发者在 Java 应用程序中能方便地结合 XML 数据和处理函数。

利用 JAXB 技术，无须深入 XML 编程细节，就能在 Java 应用程序中灵活操作 XML 数据，而且可以充分利用 XML 的优势而不用依赖于复杂的 XML 处理模型，如 SAX 或 DOM 等。JAXB 隐藏了底层操作细节，并且取消了 SAX 和 DOM 中无用的关系，生成的 JAXB 类仅描述原始模型中定义的关系。其结果是整合高度可移植的 Java 代码和高度可移植的 XML 文档。通过这些代码可以创建灵活、轻便的应用程序和 Web 服务。

下面使用 JAXB 组件处理 Java 对象与 XML 之间的相互转换，采用 User 和 LoginLog 两个实体来创建 Schema 文档，通过 Schema 文档创建相应的 Java 源代码。在进行 Java 对象编组与反编组操作之前，需要编写 XML Schema 文档，并通过模式文档生成相应的 Java 代码。

1. 编写 XML Schema

XML Schema 文件是一个 XML 的约束文件，它定义了 XML 文件的结构和元素，以及对元素和结构的相关约束，JAXB 使用这个 Schema 文件生成相应的 Java 代码。User 模式结构图如图 19-1 所示，文档结构代码如代码清单 19-11 所示。

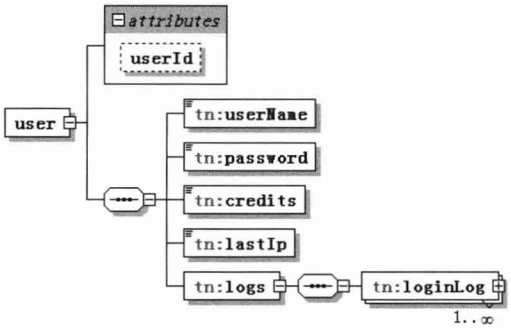


图 19-1　User 模式结构图

代码清单 19-11　user.xsd

```
<?XML version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" version="1.0"
 XMLns:xs="http://www.w3.org/2001/XMLSchema"
 XMLns:JAXB="http://Java.sun.com/xml/ns/JAXB"
    XMLns:tn="http://www.smart.oxm.com/domain/JAXB"
    targetNamespace="http://www.smart.oxm.com/domain/JAXB"
    JAXB:version="2.0" >
    <xs:annotation>
        <xs:appinfo>
            <JAXB:globalBindings>
                <JAXB:JavaType name="Java.util.Calendar" XMLType="xs:date"
                    parseMethod="javax.xml.bind.DatatypeConverter.parseDate"
                    printMethod="javax.xml.bind.DatatypeConverter.printDate" />
            </JAXB:globalBindings>
        </xs:appinfo>
    </xs:annotation>
    <xs:element name="user">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="userName" type="xs:string" />
                <xs:element name="password" type="xs:string" />
                <xs:element name="credits" type="xs:int" />
                <xs:element name="lastIp" type="xs:string" />
                <xs:element name="logs">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element maxOccurs="unbounded" name="loginLog"
                                type="tn:LoginLog" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="userId" type="xs:int" />
        </xs:complexType>
    </xs:element>
    <xs:complexType name="LoginLog">
        <xs:sequence>
            <xs:element name="userId" type="xs:string" />
            <xs:element name="ip" type="xs:string" />
```

```
            <xs:element name="loginDate" type="xs:date" />
        </xs:sequence>
        <xs:attribute name="loginLogId" type="xs:int" />
    </xs:complexType>
</xs:schema>
```

JAXB 在生成代码时，如果元素类型是日期类型，也就是 type="xs:date"时，默认情况下生成的 Java 对象属性类型为 javax.xml.datatype.XMLGregorianCalendar。这明显不是我们想要的类型，不过无须担心，JAXB 已经为我们提供了一个全局的类型转换机制，如实例在 Schema 文档中设置一个全局解析日期类型。

```
<xs:annotation>
    <xs:appinfo>
        <JAXB:globalBindings>
            <JAXB:javaType name="java.util.Calendar"  xmlType="xs:date"
                parseMethod="javax.xml.bind.DatatypeConverter.parseDate"
                printMethod="javax.xml.bind.DatatypeConverter.printDate" />
        </JAXB:globalBindings>
    </xs:appinfo>
</xs:annotation>
```

此时 JAXB 生成的对应实例类属性类型就变成 java.util.Calendar。细心的读者可能会有疑问：如果想要的类型是 java.util.Date，那么该如何设置？由于 JAXB 提供的日期解析器只能转换成 Calendar 类型，所以我们需要自己编写一个 Date 类型的适配器来替换 javax.xml.bind.DatatypeConverter.parseDate。接下来编写一个 Date 类型的适配器，如代码清单 19-12 所示。

代码清单 19-12　DateAdapter.java

```
package com.smart.oxm.JAXB;
…
public class DateAdapter {
    public static Date parseDate(String s) {
        return DatatypeConverter.parseDate(s).getTime();
    }
    public static String printDate(Date dt) {
        Calendar cal = new GregorianCalendar();
        cal.setTime(dt);
        return DatatypeConverter.printDate(cal);
    }
```

2. 生成 Java 类

xjc 工具基于 XML Schema 文档定义来绑定一个模式到 Java 类，也就是根据 XML Schema 文档定义生成相应的 Java 实体类。针对当前 XML 的模式来进行绑定的命令是：

```
xjc  user.xsd
```

xjc 命令行一些常用参数选项说明如下。

- -nv：对于输入的模式不执行严格的 XML 验证。
- -b <file>：指定外部的绑定文件。
- -d <dir>：指定生成文件的存放路径。
- -p <pkg>：指定目标包。

- -classpath <arg>：指定 classpath。
- -XMLschema：输入的模式是一个 W3C XML 模式（默认）。

xjc 命令提供了包参数选项-p，用来设置生成 Java 实体类的包结构。如果该参数没有输入，则 xjc 将利用 Schema 文档中定义的 namespace 自动产生包名（package name）。如实例中设置的 targetNamespace="http://www.smart.oxm.com/domain/jaxb"，生成的包结构为 com.smart.oxm.domain.jaxb。

使用参数-p 指定包名，生成的源代码结构如下：

```
D:\masterspring\code\chapter19\bin>
                          xjc -p com.smart.oxm.domain.jaxb  user.xsd -d src
parsing a schema...
compiling a schema...
com\smart\oxm\domain\ jaxb\Adapter1. java
com\smart\oxm\domain\ jaxb\LoginLog. java
com\smart\oxm\domain\ jaxb\ObjectFactory. java
com\smart\oxm\domain\ jaxb\User. java
com\smart\oxm\domain\ jaxb\package-info. java
```

使用 Schema 中定义的 namespace 作为包名，生成的源代码结构如下：

```
D:\masterspring\code\chapter19\bin>xjc user.xsd -d src
parsing a schema...
compiling a schema...
com\smart\oxm\domain\ jaxb\LoginLog. java
com\smart\oxm\domain\ jaxb\ObjectFactory. java
com\smart\oxm\domain\ jaxb\User. java
com\smart\oxm\domain\ jaxb\package-info. java
org\w3\_2001\XMLschema\Adapter1. java
```

在使用 JAXB 进行编组与反编组操作之前，需要创建一个 JAXBContext 对象，即 JAXB 的上下文，如实例中的 JAXBContext context = JAXBContext.newInstance(User.class)，通过其 createMarshaller()方法创建一个编组器，编组时调用 Marshaller#marshal()方法完成转换操作。如果希望生成的 XML 文件格式良好，则需要调用 Marshaller#setProperty()方法设置输出 XML 文件的格式，反编组时调用其 createUnmarshaller()方法创建一个反编组器，然后调用 Unmarshaller#unmarshal()方法即可完成 XML 到 Java 对象的转换，如代码清单 19-13 所示。

代码清单 19-13　JAXBSample.xml

```
package com.smart.oxm.JAXB;
…
public class JAXBSample {

    // Java转换为XML对象
    public static void objectToXML() throws Exception {
        User user = getUser();
        JAXBContext context = JAXBContext.newInstance(User.class);
        Marshaller m = context.createMarshaller();
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        FileWriter writer = new FileWriter("out/JaxbSample.xml");
        m.marshal(user, writer);
    }
```

```
    // XML转换为Java对象
    public static void XMLToObject() throws Exception {
        JAXBContext context = JAXBContext.newInstance(User.class);
        FileReader reader = new FileReader("out/JaxbSample.xml");
        Unmarshaller um = context.createUnmarshaller();
        User u = (User) um.unmarshal(reader);
        …
    }
  …
}
```

实战经验

目前 XML 数据绑定组件基本都支持两种方式：一是映射绑定；二是代码生成。如果采用代码生成，即根据 XML 样本数据文法生成 Java 语言代码，则必须编写 XML Schema 文件。对于熟悉 XML 的开发人员来说，可以自己编写这个 Schema 文件；而对于不熟悉 XML 的开发人员来说，则可以通过一些工具来完成，比较有名的如 Trang、XMLSpy、Stylus Studio 都可以通过 XML 样本数据来生成相应的 Schema 文件。

19.3.2　Castor

Castor 是 ExoLab Group 下一个开放源代码的项目，其主要目标是在 XML 数据、Java 对象和数据库关系数据之间提供一种直接的映射，使得这 3 种对象数据可以相互转换。Castor 项目主要包括 XML 与 Java 对象的映射（Castor XML）、Java 对象与关系数据库表映射（Castor JDO）两个关键功能。在 Castor 中，通过定义数据映射（Mapping）文件，在 XML 文档元素（节点、属性、文本等）、Java 类（类、属性等）和数据库表（关联表、表、字段）之间建立一一映射。通过 XML Schema 定义可以自动生成实体 Java 类定义，通过实体 Java 类定义可以自动生成 XML 元素与 Java 类之间的映射文件。

1. Castor 主要特性

- 通过定义映射文件在 XML 文档与 Java 对象实体间建立映射。
- Castor XML 在 Java 对象模型与 XML InfoSet 之间建立相互转换和数据绑定功能。
- 提供从 XML 文档定义自动生成 Java 类定义文件的功能（Code Generator）。
- 通过自省方式在 XML 文档与 Java 对象间建立映射。
- 提供基于 Java 类定义生成映射文件的功能。
- 提供基于 XML 输入文档生成 Schema 定义文件的功能。
- Castor JDO 为 Java 对象与数据库关系数据之间提供一种序列化与反序列化的框架。
- 基于 XML 映射文件的方式定义 3 种不同数据对象模型之间的映射。
- 提供内存缓存和提交写入方式，减少数据库 JDBC 操作。

- 支持两阶段事务、对象操作回滚和数据库锁定侦测。
- 支持 OQL 与标准 SQL 之间的映射，Castor JDO 对象查询采用 OQL。

Castor 组件提供了几种进行 Java 对象与 XML 相互转换的方式：一是采用 XML Schema 生成代码方式；二是采用自省方式；三是采用映射文件方式。下面通过这几种不同方式体验 Castor 组件的灵活与强大。

在开发实例之前，我们需要在 pom.xml 中引入 Castor 的依赖包，包含 castor-xml 和 castor-core。本实例选用 Castor 1.3 版本，代码如下：

```
<dependency>
    <groupId>org.codehaus.castor</groupId>
    <artifactId>castor-xml</artifactId>
    <version>1.3</version>
</dependency>
<dependency>
     <groupId>org.codehaus.castor</groupId>
      <artifactId>castor-core</artifactId>
      <version>1.3</version>
 </dependency>
```

2. XML Schema 生成代码

创建 XML Schema 文件，Castor 使用这个 Schema 文件生成一系列相关的 Java 实体类及相应的类描述符。本例 User 模式结构图如图 19-2 所示。

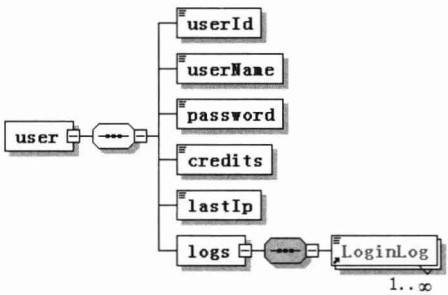


图 19-2　User 模式结构图

为了应用 Castor 生成 Java 实体类，不仅需要把 castor-1.3.jar、castor-1.3-codegen.jar 和 castor-1.3-XML-schema.jar 包放到类路径下，还需要编写一个批处理文件或者 Ant 构建文件。首先编写一个批处理文件 user.bat，如代码清单 19-14 所示。

代码清单 19-14　user.bat

```
@echo off
REM Change the following line to set your JDK path
set JAVA_HOME=%JAVA_HOME%
set JAVA=%JAVA_HOME%\bin\Java
set JAVAC=%JAVA_HOME%\bin\Javac
@echo Create the classpath
set CP=.
for %%i in (lib\*.jar) do call cp.bat %%i
set CP=%CP%;%JDK_BIN%\lib\tools.jar
```

```
@echo.
@echo Using classpath: %CP%
@echo Castor Test Cases
@echo.
@echo Generating classes...
@rem Java 2 style collection types
@rem %JAVA% org.exolab.castor.builder.SourceGeneratorMain -i invoice.xsd -f -types j2
-binding-file bindingInvoice.xml
@rem Java 1.1 collection types
%JAVA% -cp %CP% org.exolab.castor.builder.SourceGeneratorMain -i user.xsd -types j2
-package com.smart.oxm.domain.castor
@echo.
@pause
```

①生成代码入口

其中，org.exolab.castor.builder.SourceGeneratorMain 生成代码入口，参数-i 指定 Schema 文件，参数-package 指定生成实例类所在的包名。

运行 user.bat 文件，生成一系列 Java 实体类及实体类相应的类描述符。如本实例将生成 3 个实例类（User.java、LoginLog.java、Logs.java）及 3 个实体类对应的类描述符（UserDescriptor. java、LoginLogDescriptor. java、LogsDescriptor. Java）。

与 XMLBeans 一样，Castor 也可以通过生成 Java 代码的方式完成对象与 XML 相互转换，但 Castor 生成实体类及类描述符之后，转换工作比 XMLBeans 更加简洁，把一个实体对象转换为 XML，只要创建当前对象的实例，调用实体对象 marshal(Writer out)方法，就完成了对象编组工作，也就是把当前对象转换成一个 XML 文件。反编组也一样简洁，只需调用实例类反编组静态方法 unmarshal(Reader reader)就可以完成转换工作，如代码清单 19-15 所示。

代码清单 19-15　CastorGeneratorSampe.java

```
package com.smart.oxm.castor;
…
public class CastorGeneratorSampe {

    // Java对象转换为XML
    public static void objectToXML()throws Exception {
        User user = getUser();
        FileWriter writer = new FileWriter("out/CastorSample.xml");
        user.marshal(writer);
    }

    // XML转换为Java对象
    public static void XMLToObject()throws Exception {
         FileReader reader = new FileReader("out/CastorSample.xml");
         User u = User.unmarshal(reader);
         …
    }
…
}
```

3．自省方式的编组与反编组

Castor 提供自省的方式在 Java 对象与 XML 文档之间实现转换，也就是说，我们不

需要编写 Java 对象与 XML 文档之间的映射文件，Castor 在编组的时候会根据实体对象属性生成相应的 XML 元素。

Castor 提供了 Marshaller 和 Unmarshaller 两个类处理编组与反编组操作，在编组的时候，调用 Marshaller#marshal()方法完成对象转换为 XML 的工作；在反编组的时候，调用 Unmarshaller 静态反编组方法 Unmarshaller#unmarshal()即可。如实例中的 Unmarshaller# unmarshal()方法，通过 FileReader 把 XML 文件内容读入并映射到 User 对象，如代码清单 19-16 所示。

代码清单 19-16　CastorSample.java

```
package com.smart.oxm.castor;
…
public class CastorSample {

    // Java 对象转换为 XML
    public static void objectToXML()throws Exception {
        User user = getUser();
        FileWriter writer = new FileWriter("out/CastorSample.xml");
        Marshaller marshaller = new Marshaller(writer);
        marshaller.setEncoding("GBK");
        marshaller.marshal(user);
    }

    // XML 转换为 Java 对象
    public static void XMLToObject()throws Exception {
        FileReader reader = new FileReader("out/CastorSample.xml");
        User u = (User) Unmarshaller.unmarshal(User.class, reader);
        …
    }
    …
}
```

4．映射文件方式的编组与解编组

Castor 虽然提供自省的方式在 Java 对象与 XML 文档之间进行转换，但当 Java 对象属性与 XML 文档元素名称不一致时，就无法采用自省方式，这时就需要编写它们之间的映射文件，告诉 Castor 它们之间的对应关系。本实例仍然使用 User. java 及 LoginLog.java 作为实例的操作对象。

首先需要编写一个映射文件 mapping.xml，如代码清单 19-17 所示。

代码清单 19-17　mapping.xml

```
<?XML version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                    "http://castor.org/mapping.dtd">
<mapping>
    <class name="com.smart.oxm.domain.User">
        <map-to xml="user" /><!-①设置XML文件的根元素 -->
        <field name="userId" type="int">
            <bind- xml name="id" node="attribute" />
        </field>
        …
```

```
        <field name="userName" type="Java.lang.String">
            <bind- xml  name="userName"  />
        </field>
        <field name="logs" type="com.smart.oxm.domain.LoginLog" collection=
"arraylist">
            <bind-xml name="log" />
        </field>
    </class>

    <!-- ②通过在该元素中加入auto-complete属性并把值设为true。您可以告诉Castor对于该类的任何
属性，只要没有在这个元素中专门列出，就使用默认映射 -->
    <class name="com.smart.oxm.domain.LoginLog" identity="userId"
auto-complete="true">
        <map-to xml="logs" />
        <field name="loginLogId" type="int">
            <bind-xml name="id" node="attribute" />
        </field>
        <field name="userId" type="int">
            <bind-xml name="userId" />
        </field>
    </class>
</mapping>
```

与 Castor 自省方式编组及反编组操作方式一样，在编组的时候，也调用 Marshaller#marshal()方法完成 Java 对象转换为 XML 的工作，不同的是在编组之前需要先把映射文件加载到 Mapping 中，然后调用 Marshaller#setMapping()方法设置到编组器中。在反编组的时候，与自省方式不同，需要先创建 Unmarshaller 实例，并把映射文件加载到其实例中，然后调用 Unmarshaller 对象方法 unmarshal(FileReader reader)进行反编组操作，而不是调用静态方法。

Castor 使用 Marshaller 和 Unmarshaller 类的静态方法以自省的方式实现 XML 文档与 Java 对象之间的编组和反编组，以及使用 Mapping 映射文件通过调用 Marshaller 和 Unmarshaller 类的实例方法实现 XML 文档与 Java 对象之间的编组和反编组两种方式。前一种方式不需要构造 Marshaller 和 Unmarshaller 类的实例，后一种方式需要构造 Marshaller 和 Unmarshaller 类的实例，调用非静态方法完成上述功能。在调用 Marshaller 和 Unmarshaller 类的实例方法时，必须提供 XML 文档结构和 Java 类模型间的映射定义文件。

Castor 通过类描述符和字段描述符几乎可以编组/解编组所有的 Java 对象。当某个 Java 对象类描述符不存在时，编组框架使用反射（Reflection）方式获取对象的结构信息，并在内存中为此对象构造类描述符和字段描述符。当实体类的描述符存在时，Castor 使用这些描述符提供的 Java 对象模型信息处理编组和解编组。在编组与解编组过程中，Castor 要求实体类定义必须提供 public 类型的默认构造函数（不含参数）声明以及必要的 getter 和 setter 方法。

19.3.3 JiBX

JiBX 是一款非常优秀的 XML 数据绑定框架。它提供灵活的绑定映射文件，以实现数据对象与 XML 文件之间的转换，并不需要修改既有的 Java 类。不管是在灵活性还是在效率上，JiBX 是目前很多开源项目都无法比拟的，如具有转换效率高、配置绑定文件简单、不需要操作 XPath 文件、不需要写属性的 get/set 方法、对象属性名与 XML 文件元素名不需要相同等优点。

JiBX 使用 Java 字节码的增强技术，即 BCEL（Byte Code Engineering Library），在编译期，根据绑定配置文件，生成对应对象实例的方法和添加被绑定标记的属性；在运行期，它已经与绑定配置文件无关了，不需要再作任何配置，因为其已经嵌入到相应的 Java 类中。2005 年 12 月发布了 JiBX 1.0 版本，目前的最新版本是 2015 年 1 月发布的 JiBX 1.2.6。

JiBX 具有如下主要特性：

- JiBX 使用 BCEL 技术在编译时对相应的类字节码进行增强。
- JiBX 使用 XPP（XML Pull Parse）技术，它提供了一个更自然的接口来处理文档中的元素序列，能够使用相对简单的代码来处理数据。
- JiBX 可以自由灵活地定义 XML 中的每个字段和 Java 类的对应关系。

在开发实例之前，需要在 pom.xml 中引入 JiBX 的依赖包，本实例采用 JiBX 1.1.5 版本，代码如下：

```
<dependency>
    <groupId>net.sourceforge.jibx</groupId>
    <artifactId>com.springsource.org.jibx.runtime</artifactId>
    <version>1.1.5</version>
</dependency>
```

在使用 JiBX 进行编组及反编组操作时，需要编写 Java 对象与 XML 之间的绑定映射文件。在运行程序之前，需要先配置绑定文件并进行绑定，在绑定过程中将会动态修改程序中相应的.class 文件，主要是生成对应对象实例的方法和添加被绑定标记的属性 JiBX_bindingList 等。它有如此大的魔法，得益于 BCEL 技术。BCEL 是 Apache Software Foundation 的 Jakarta 项目的一部分，BCEL 在单独的 JVM 指令级别上进行操作，可以深入 Java 类的字节码，用它转换现有的类或者构建新的类，如代码清单 19-18 所示。

代码清单 19-18 binding.xml

```
<binding>
  <mapping name="user" class="com.smart.oxm.domain.User">
    <value name="userName" field="userName"/>
    <!-name设置XML元素名称，field设置Object对象属性名称 -->
    <value name="password" field="password" usage="optional" />
    <value name="credits" field="credits" usage="optional" />
   <collection field="logs" factory="com.smart.oxm.jibx.JiBXInterfaceFactory.
getArrayListInstance">
     <structure name="log" type="com.smart.oxm.domain.LoginLog">
          <value name="ip" field="ip"/>
```

```
        <value name="loginDate" field="loginDate"/>
      </structure>
    </collection>
  </mapping>
</binding>
```

在 JiBX 绑定 Java 对象时，如果使用了基本类型，又设置字段为可选值，如<value name="credits" field="credits" usage="optional" />中的 usage="optional"，则 JiBX 根据 credits 类型的默认值来决定是否编组当前元素。如果编组的时候 credits 的值刚好是 0，则 credits 无法显示在 XML 文件中，也就是说基本类型为可选项时，是无法输出默认值的。为了避免出现这个问题，我们在使用基本类型时，需要把字段设置为不可选。

使用接口处理从 XML 到 Java 对象的转换时，先要创建该类的实例。如果转换的是一个实体类，则创建实例不会有什么问题；但如果我们要使用接口编程，则转换的对象必须是一个接口，否则会出现错误，因为 JiBX 并不清楚你需要创建这个接口的哪个实例，所以我们需要在绑定文件中指明该接口的创建工厂方法。例如，本例 User 中声明一个 List 类型的属性 logs，想要让它指向一个 ArrayList 的实例，则需要写一个返回 ArrayList 实例的工厂类 JiBXInterfaceFactory，如代码清单 19-19 所示。

代码清单 19-19　JiBXInterfaceFactory.java

```
package com.smart.oxm.jibx;
…
public class JiBXInterfaceFactory {
    //获取List实现类实例
    public static List getArrayListInstance(){
        return new ArrayList();
    }
}
```

编写完绑定映射文件后，还要编写一个 Ant 构建文件，来完成 Java 类编译及字节码增强的工作，以增强实体对象编组及反编组的能力，如代码清单 19-20 所示。

代码清单 19-20　build.xml

```
<?XML version="1.0" encoding="gb2312"?>
<project name="oxm" default="run">
    <property name="project" value="D:/masterspring/code/chapter19"/>
    <property name="src" value="D:/masterspring/code /chapter19/src/main/java "/>
    <property name="lib" value="D:/masterspring/code/libs"/>
    <property name="classes"
    value="D:/masterspring/code/chapter19/target/classes"/>
    <target name="clean">
        <delete includeEmptyDirs="true">
            <fileset dir="${classes}" includes="**/*" defaultexcludes="no"/>
        </delete>
    </target>
     …
    <taskdef name="bind" classname="org.jibx.binding.ant.CompileTask"
classpath="${lib}/jibx-bind.jar"/>
    <target name="jibx-binding" depends="compile">
          <bind verbose="false" load="true"
```

```
            binding="${classes}/com/smart/oxm/jibx/binding.xml">
              <classpath>
                  <path location="${classes}">
                  </path>
                  <pathelement location="${lib}/jibx-bind.jar"/>
              </classpath>
          </bind>
      </target>
      <target name="run" depends="jibx-binding" description="run">
          <Java classname="com.smart.oxm.jibx.JibxSample">
              <classpath>
                  <path location="${classes}">
                  </path>
                  …
              </classpath>
          </Java>
      </target>
</project>
```

用 Ant 运行当前构建文件，在 User.class 同目录下，我们可以看到 JiBX 自动编译生成 JiBX_MungeAdapter.class 等 3 个类，用反编译工具查看 User.class 就会发现，User 已经自动实现 IMarshallable 和 IUnmarshallable（编组和反编组）两个接口。

使用 JiBX 进行编组及反编组操作时，需要通过 BindingDirectory 获取一个相应实体类的绑定工厂实例。如实例中的 IBindingFactory bfact = BindingDirectory.getFactory (User.class)，在编组的时候，先调用其 createMarshallingContext()方法创建一个编组上下文，然后调用编组上下文实例的 marshalDocument()方法进行编组；在反编组的时候，调用其 createUnmarshallingContext()方法创建一个反编组上下文，并通过实例的 unmarshalDocument()方法来进行反编组，如代码清单 19-21 所示。

代码清单 19-21　JibxSample.xml

```
package com.smart.oxm.jibx;
…
public class JibxSample {
    // Java对象转换为XML
    public static void objectToXML() throws Exception {
        User user = getUser();
        IBindingFactory bfact = BindingDirectory.getFactory(User.class);
        IMarshallingContext ctx = bfact.createMarshallingContext();
        FileOutputStream os = new FileOutputStream("out/JibxSample.xml");
        ctx.marshalDocument(user, "UTF-8", null, os);
    }

    // XML转换为Java对象
    public static void XMLToObject() throws Exception {
        IBindingFactory bfact = BindingDirectory.getFactory(User.class);
        IUnmarshallingContext uctx = bfact.createUnmarshallingContext();
        File dataFile = new File("out/JibxSample.xml");
        InputStream in = new FileInputStream(dataFile);
        User user = (User) uctx.unmarshalDocument(in, null);
        …
```

```
    }
    ...
}
```

JiBX 是一款高性能的数据绑定框架。如果 XML 文件格式比较固定，同时数据转换比较频繁，则可以考虑使用它来助你一臂之力。需要注意的是，JiBX 对中文的支持不是很好，目前 JiBX 只支持 Java 标准的字符集。如果使用的是 UTF-8 编码来处理中文，则没有任何问题；但如果使用 GB2312 或 GBK 编码，则需要自己实现 GB2312 和 GBK 的 Escaper 类，重写 writeAttribute(String, Writer)、writeCData(String, Writer)和 writeContent (String, Writer)方法。

19.3.4　总结比较

上文详细介绍了目前一些主流 O/X Mapping 组件的特点及使用方法，不同 O/X Mapping 组件都有各自的应用场景。如在构建基于 Spring 轻量级 RESTful 服务的过程中，可以使用 XStream 组件处理服务请求响应的报文（XML 或 JSON 数据格式）；在构建基于 Spring 契约优先的 Web Services 时，可以使用 XMLBeans、JAXB 等组件来处理服务请求响应的报文。表 19-2 列举了目前各 O/X Mapping 组件的优势、劣势及其应用场景，读者可以根据实际需求选择合适的 O/X Mapping 组件。

表 19-2　各O/X Mapping组件对比

XML 编组框架	优　势	劣　势	应用场景
XStream	简单易用； 同时支持XML和JSON数据格式	非标准； 默认 Driver 不支持 CDATA； 映射集合时对 XML 的格式要求很严格； 修改映射就需要修改代码	应用对象各种序列化及反序列化操作； 基于 Spring 轻量级 RESTful 服务中的对象映射处理
Castor	提供丰富的工具，如根据 XML 文档生成 Java 代码、根据 Java 类定义生成映射文件等； 支持自省方式在 XML 与 Java 对象间建立映射； 支持绑定任意类； 完全支持 JDO	不支持使用注解	应用各种数据绑定需求； 应用 JPA 相关实现； 使用 XML 交换数据的应用程序
JiBX	支持 CDATA； 对集合的兼容性很好； 允许定义空标签； 在配置文件中映射，而不是在代码中映射； 基于字节码增强技术，运行速度非常快	需要对 POJO 进行增强，每次重新编译后都必须重新进行增强。不过可以预编译，也可以运行时编译； 配置文件相对复杂些； 不支持灵活的验证机制	广泛应用各种数据绑定需求； Web Services 常用数据绑定框架

续表

XML 编组框架	优　势	劣　势	应用场景
JAXB	业界的标准	XML 模式（XML Schema）发生变化，就必须重新编译；JAXB 重新编译，可能会引起由 XML 模式生成的 JAXB 代码产生变化	广泛应用各种数据绑定需求； Web Services 常用数据绑定框架； 使用 XML 交换数据的应用程序

19.4 与 Spring OXM 整合

19.4.1 Spring OXM 概述

Spring OXM 是 Spring 3.0 的一个新特性，为主流 O/X Mapping 组件提供了统一层抽象和封装，而在 Spring 4.0 中 OXM 没有增加更多新的特性。O/X 映射器这个概念并不新鲜，O 代表 Object，X 代表 XML，目的是在 Java 对象和 XML 之间进行转换操作。Spring OXM 不仅屏蔽了各 O/X Mapping 组件实现的差异性，而且还提供了统一、高效的编程模型。Spring OXM 的结构框架如图 19-3 所示。

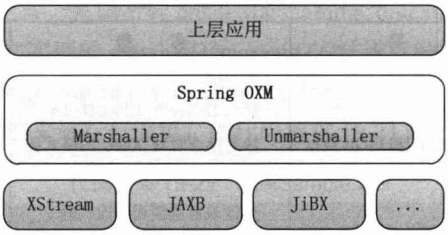


图 19-3　Spring OXM 的结构框架

Marshaller 和 Unmarshaller 是 Spring OXM 的两个核心接口。实现 Marshaller 接口可以实现从 Java 对象到 XML 的映射转换，实现 Unmarshaller 接口可实现从 XML 到 Java 对象的映射转换。下面从 Spring OXM 源代码中摘录部分 Marshaller 接口的定义，如代码清单 19-22 所示。

代码清单 19-22　Marshaller.java

```
package org.springframework.oxm;
import java.io.IOException;
import javax.xml.transform.Result;
public interface Marshaller {

    //① 判断支持的编组类类型
    boolean supports(Class<?> clazz);

    //② 对传入的对象进行编组操作
```

```
    void marshal(Object graph, Result result) throws IOException, XmlMappingException;
}
```

Marshaller 接口中提供了两个方法，其中 Marshaller#supports()方法用于判断支持的编组类类型；Marshaller#marshal()方法用于对对象进行编组操作，其中 graph 参数为目标转换对象，result 参数为 JDK 提供的 XML 转换接口，实现此接口的对象包含构建转换结果树所需的信息。Spring OXM 支持的 Result 转换实现类有 DOMResult、SAXResult、StreamResult、StaxResult，其中 DOMResult、SAXResult、StreamResult 为 JDK 提供的实现类，StaxResult 为 Spring OXM 提供的扩展类。

下面从 Spring OXM 源代码中摘录部分 Unmarshaller 接口的定义，如代码清单 19-23 所示。

代码清单 19-23　Unmarshaller.java

```
package org.springframework.oxm;
import java.io.IOException;
import javax.xml.transform. Source;
public interface Unmarshaller{

    //① 判断支持的反编组类类型
    boolean supports(Class<?> clazz);

    //② 对输入的源对象进行反编组操作
    Object unmarshal(Source source) throws IOException, XmlMappingException;

}
```

Unmarshaller 接口中提供了两个方法，其中 Marshaller#supports()方法用于判断支持的反编组类类型；Marshaller#unmarshal()方法用于将输入源对象进行反编组操作，其中 source 参数为目标输入源对象，Source 参数为 JDK 提供的输入源接口，实现此接口的对象包含充当源输入（XML 源或转换指令）所需的信息。Spring OXM 支持的 Source 输入源实现类有 DOMSource、SAXSource、StreamSource、StaxSource，其中 DOMSource、SAXSource、StreamSource 为 JDK 提供的实现类，StaxSource 为 Spring OXM 提供的扩展类。

Spring OXM 统一封装底层的 O/X Mapping 组件异常，将各 O/X Mapping 组件原始异常对象包装到 Spring 自身专为 OXM 建立的运行时异常 XmlMappingException 中，并通过 MarshallingFailureException 和 UnmarshallingFailureException 处理编组和反编组操作之间的区别。Spring OXM 异常层次结构如图 19-4 所示。

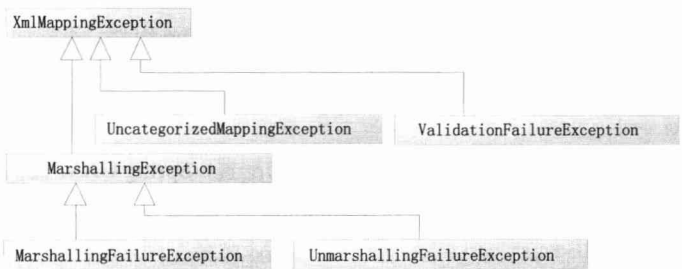


图 19-4　Spring OXM 异常层次结构

19.4.2 整合 OXM 实现者

Spring OXM 默认提供目前几个主流 O/X Mapping 组件的实现，包括 XStream、XMLBeans、Castor、JiBX、JAXB。这些 O/X Mapping 组件统一实现 Spring OXM 的两个核心接口 Marshaller 和 Unmarshaller。具体的实现体系类图如图 19-5 所示。

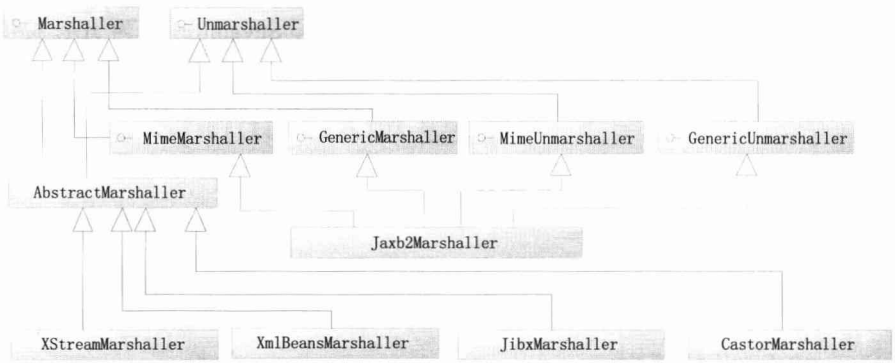


图 19-5 Spring OXM 核心接口实现

要使用 Spring OXM 的 O/X 功能，首先需要一个在 Java 对象和 XML 之间来回转换的组件，并将上文提到的 XStream、JAXB 等相应的类库添加到工程中。各 O/X Mapping 组件对应的实现类如表 19-3 所示。

表 19-3 各O/X Mapping组件包装器

O/X Mapping 组件	Spring OXM 实现类
XStream	org.springframework.oxm.xstream.XStreamMarshaller
Castor	org.springframework.oxm.castor.CastorMarshaller
JiBX	org.springframework.oxm.jibx.JibxMarshaller
JAXB 2.0	org.springframework.oxm.jaxb.JAXB2Marshaller

19.4.3 如何在 Spring 中进行配置

到目前为止，我们已经介绍了各 O/X Mapping 组件的使用方法以及 Spring OXM 总体框架，接下来介绍如何在 Spring 中整合这些 O/X Mapping 组件。不同的 O/X Mapping 组件在 Spring 中的配置略有不同，下面分别对各主流 O/X Mapping 组件的配置进行讲解。

1. OXM 命名空间

为了简化 Spring OXM 各 O/X Mapping 组件的配置，Spring OXM 提供了一个 OXM 命名空间，目前支持 OXM 命名空间的 O/X Mapping 组件有 3 个，分别是 Jaxb2Marshaller、JibxMarshaller 和 XmlBeansMarshaller。要启用 OXM 命名空间，首先需要在 Spring 配置文件中引用 OXM 模式定义文件，如代码清单 19-24 中的粗体所示。

代码清单 19-24　OXM命名空间

```
<?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:oxm="http://www.springframework.org/schema/oxm"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/oxm
http://www.springframework.org/schema/oxm/spring-oxm.xsd">
   …
</beans>
```

2. XStreamMarshaller 配置

要启用 Spring OXM 的 XStreamMarshaller 功能，首先需要将 XStream O/X Mapping 组件相应的类库添加到工程 pom.xml 的依赖中，本章中使用 XStream 1.4.9 版本。

```
<dependency>
   <groupId>com.thoughtworks.xstream</groupId>
   <artifactId>xstream</artifactId>
   <version>1.4.9</version>
</dependency>
```

在默认情况下，不需要对 XStreamMarshaller 作任何配置，直接在 Spring 应用程序上下文中配置即可。如果需要自定义的 XML 格式，则可以设置类别名映射及属性别名映射，如代码清单 19-25 所示。

代码清单 19-25　XStreamMarshaller配置实例

```
<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller"
        p:autodetectAnnotations="true">
        <!-- 设置类名别名 -->
        <property name="aliases">
            <map>
                <!-- User这个类的别名就变成了user -->
                <entry key="user" value="com.smart.oxm.domain.User" />
                <entry key="LoginLog" value="com.smart.oxm.domain.LoginLog" />
            </map>
        </property>
        <!-- 设置类属性别名 -->
        <property name="fieldAliases">
            <map>
                <!-- User中的lastVisit属性 -->
                <entry key="com.smart.oxm.domain.User.lastVisit" value="lastVisitDate" />
            </map>
        </property>
</bean>
```

在上面的配置实例中，通过 aliases 属性设置类名别名，通过 fieldAliases 属性设置类属性别名，通过 autodetectAnnotations 属性设置自动加载 XStream 注解 Bean。

3. Jaxb2Marshaller 配置

要启用 Spring OXM 的 Jaxb2Marshaller 功能，首先需要将 Jaxb O/X Mapping 组件相应的类库增加到 pom.xml 的依赖中，如本章中使用 jaxb-api-2.1.4 和 jaxb-impl-2.1.8。

Jaxb2Marshaller 可以使用与 Jaxb1Marshaller 相同配置属性的 contextPath 来指定编组对象。Jaxb2Marshaller 提供了一个 classesToBeBound 属性，用来设置编组对象数组。Schema 属性指定一个或多个模式资源，如代码清单 19-26 所示。

代码清单 19-26　Jaxb2Marshaller配置实例

```
<!--① 传统的配置方法 -->
<bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <!--<property name="contextPath"
value="com.smart.oxm.domain.jaxb"></property>-->
    <property name="classesToBeBound">
        <array>
            <value>com.smart.oxm.domain.jaxb.User</value>
            <value>com.smart.oxm.domain.jaxb.LoginLog</value>
        </array>
    </property>
    <property name="schema" value="classpath:com/smart/oxm/jaxb/user.xsd" />
</bean>
<!--② 基于OXM命名空间的配置方法 -->
<oxm:jaxb2-marshaller id="jaxb2Marshaller2">
    <oxm:class-to-be-bound name="com.smart.oxm.domain.jaxb.User" />
    <oxm:class-to-be-bound name="com.smart.oxm.domain.jaxb.LoginLog" />
</oxm:jaxb2-marshaller>
```

Jaxb1Marshaller 在 Spring OXM 3.0 之后不被支持，如果没有特殊需求，则推荐使用 Jaxb2Marshaller 作为 O/X Mapping 编组与反编组包装器。

4．CastorMarshaller 配置

要启用 Spring OXM 的 CastorMarshaller 功能，首先需要将 Castor O/X Mapping 组件相应的依赖包添加到工程的 pom.xml 中，如本章中使用 castor-1.3-core、castor-1.3-xml 和 castor-1.3-commons。不需要对 CastorMarshaller 作任何配置，直接在 Spring 应用程序上下文中配置即可。如果需要自定义的映射配置文件，则需要配置 mappingLocation 属性，如代码清单 19-27 所示。

代码清单 19-27　CastorMarshaller配置实例

```
<bean id="castorMarshaller"
     class="org.springframework.oxm.castor.CastorMarshaller"
     p:mappingLocation="classpath:/config/mapping.xml" />
```

虽然 Castor 支持自省方式在 XML 与 Java 对象间建立映射，但有时可能需要自定义设置它们之间的映射关系，这时就需要使用 Castor 的映射文件。可以通过 mappingLocation 属性来指定自定义的配置文件。

5．JibxMarshaller 配置

要启用 Spring OXM 的 JibxMarshaller 功能，首先需要将 JiBX O/X Mapping 组件相应的依赖包添加到工程 pom.xm 的依赖中，如本章中使用 jibx-run、jibx-bind 和 jibx-extras。配置 JibxMarshaller 很简单，只需在 Spring 应用程序上下文中配置一个 JibxMarshaller，并通过 targetClass 属性配置目标编组类即可。如果需要设置绑定名称，则可以通过 bindingName 属性指定一个绑定配置文件，如代码清单 19-28 所示。

代码清单 19-28　JibxMarshaller配置实例

```
<!--① 传统的配置方法 -->
<bean id="jibxMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
    <property name="targetClass" value="com.smart.oxm.domain.User" />
    <!--<property name="bindingName"
            value="classpath:com/smart/oxm/jibx/binding.xml" />-->
</bean>
<!--② 基于OXM命名空间的配置方法 -->
<oxm:jibx-marshaller id="jibxMarshaller2"  target-class="com.smart.oxm.domain.User" />
```

一个 JibxMarshaller 只能配置一个目标编组类。如果需要配置多个编组类，则只能为不同的编组类各配置一个 JibxMarshaller。

19.4.4　Spring OXM 简单实例

到目前为止，我们已经介绍了各 O/X Mapping 组件的使用方法及 Spring OXM 框架，接下来通过一个简单的实例来讲解如何应用 Spring OXM 并整合第三方 O/X Mapping 组件。实例使用 Castor 这个 O/X Mapping 组件处理 Java 对象与 XML 之间的相互转换映射。首先在 Spring 上下文（applicationContext.xml）中配置一个 Castor 编组器并注入实例 SpringOxmSample 中，如代码清单 19-29 所示。

代码清单 19-29　applicationContext.xml

```
<beans XMLns="http://www.springframework.org/schema/beans"
      XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      XMLns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="springOxm" class="com.smart.oxm.SpringOxmSample"
            p:marshaller-ref="castorMarshaller"
            p:unmarshaller-ref="castorMarshaller" />
    <bean id="castorMarshaller"
class="org.springframework.oxm.castor.CastorMarshaller"
        p:mappingLocation="classpath:/config/mapping.xml" />
</beans>
```

Spring OXM 仍然遵守不重造轮子的原则，只是为各个绑定映射组件提供一致的访问接口及异常处理机制。与单独使用 Castor 组件一样，我们需要编写一个映射文件，如代码清单 19-30 所示。

代码清单 19-30　mapping.xml

```
<?XML version="1.0"?>
<mapping>
    <class name="com.smart.oxm.domain.User">
      <map-to XML="user"/>
      <field name="userId" type="integer">
        <bind-XML name="userId" node="element"/>
      </field>
      …
      <field name="logs" type="com.smart.oxm.domain.User"  collection="arraylist">
```

```
        <bind-XML name="logs" />
      </field>
    </class>
</mapping>
```

Spring OXM 屏蔽第三方映射组件中编组器及反编组器的实现细节，也就是说映射转换程序与具体映射组件解耦，如果有一天需求突然发生变化，则当前的映射组件不能满足需求，只需修改 Spring 配置文件，换成另一个合适的映射组件即可，而不需要修改映射转换程序，如代码清单 19-31 所示。

代码清单 19-31　SpringOxmSample.java

```
package com.smart.oxm;
 …
public class SpringOxmSample {
   private Marshaller marshaller;
   private Unmarshaller unmarshaller;
    //Java转换为XML对象
   public  void objectToXML()throws Exception{
      User user = getUser();
      FileOutputStream os = null;
      try {
         os = new FileOutputStream("out/SpringOxmSample.xml");
          this.marshaller.marshal(user, new StreamResult(os));
      } finally {
        …
      }
   }
   //XML转换为Java对象
   public  void XMLToObject()throws Exception{
       FileInputStream is = null;
       User user = null;
       try {
          is = new FileInputStream("out/SpringOxmSample.xml");
           user = (User) this.unmarshaller.unmarshal(new StreamSource(is));
       } finally {
         …
       }
    }
}
```

Spring OXM 的一个最直接的好处是可以通过使用 Spring 框架的其他特性简化配置。Spring 的 Bean 支持将实例化的 O/X 编组器注入（前面提到过的“依赖项注入”）使用那些编组器的对象。遵循坚实的面向对象的设计实践，Spring OXM 框架只定义两个接口：Marshaller 和 Unmarshaller，它们用于执行 O/X 功能。另一个重大好处是，这些接口的实现完全对开发人员开放，开发人员可以轻松地切换它们而无须修改代码。例如，如果你一开始使用 Castor 进行 O/X 转换，但后来发现它缺乏你需要的某个功能，这时可以切换到 XMLBeans 而无须更改任何代码。唯一需要做的就是更改 Spring 配置文件以使用新的 O/X Mapping 组件。

使用 Spring OXM 的另一个好处是统一的异常层次结构。Spring OXM 遵循使用它的数据访问模块建立的模式，将原始异常对象包装到 Spring 自身专为 OXM 建立的运行时

异常中。由于第三方 O/X Mapping 组件的原始异常被包装到 Spring 运行时异常中，这也是我们能够查明出现异常的根本原因。不必费心修改代码以捕获异常，因为异常已经包装到一个运行时异常中。

19.5　小结

在本章中，首先分析了 XML 及其解析技术的发展过程，详细介绍了 XML 处理利器 XSteam O/X Mapping 组件的使用，并介绍了目前流行的另外几个 O/X Mapping 组件的使用方法，并进行了总结比较。接着结合 Spring 提供的 OXM 特性，分析了 Spring OXM 框架及两个重要的接口 Marshaller 和 Unmarshaller，并详细介绍了 O/X Mapping 组件的配置方法。最后通过一个简单的实例，介绍了如何应用 Spring OXM 处理 Java 对象与 XML 之间的相互映射。不管采用何种方式处理，Java 应用程序的 XML 数据绑定可以归纳为两种方式：一是根据 XML 文档文法生成 Java 语言代码（如 JAXB、Castor 等）；二是使用某种形式的映射绑定方法，也就是设定 Java 类如何与 XML 进行关联（如 XStream、Castor、JiBX）。

在实际应用过程中，这两种方式都有各自的长处。如果使用由 Schema 或 DTD 定义的稳定文档结构，并且该结构适合应用程序的需要，则代码生成方法可能是最佳的选择。如果使用现有的 Java 实体类，或者希望使用类的结构，该结构反映应用程序对数据的用法，而不是 XML 结构，则映射方法是最佳的选择。

第 20 章

实战型单元测试

单元测试对保障应用程序正确性而言，其重要性怎样强调都不为过。TestNG 是必须事先掌握的基础测试框架，大多数测试框架和测试工具都在此基础上扩展而来。Spring 测试框架为集成 TestNG、JUnit 等单元测试框架提供了很好的支持，并为测试 Spring 应用提供了许多基础设施。在项目单元测试过程中，不可避免地要涉及测试环境准备，如模拟接口测试、测试数据准备等繁杂工作。Mockito、Unitils、DbUnit 等框架的出现，使得这些问题有了很好的解决方案，特别是 Unitils 结合 DbUnit 对测试 DAO 层提供了强大的支持，大大提高了编写测试用例的效率和质量。

本章主要内容：

- 概述单元测试相关概念及意义
- 简要分析对单元测试存在的误解
- Spring 测试框架简介
- TestNG、Mockito、Unitils 测试框架简介
- 使用 TestNG、Mockito、Unitils 及 Spring 进行单元测试
- 面向数据库应用的测试
- 测试实战

本章亮点：

- 对单元测试存在的误解进行全面分析
- 简明扼要地介绍了 TestNG、Mockito、Unitils 的使用
- 使用 Excel 准备测试数据及验证数据来简化 DAO 测试

20.1　单元测试概述

一种商品只有通过严格检测才能投放市场，一架飞机只有经过严格测试才能上天，同样，一款软件只有对其各项功能进行严格测试后才能交付使用。不管一款软件多么复杂，它都是由相互关联的方法和类组成的，每个方法和类都可能隐藏着 Bug。只有防微杜渐、小步前进，才可以保证软件大厦的稳固性；否则隐藏在类中的 Bug 随时都有可能像打开的潘多拉魔盒一样让程序陷于崩溃之中，难以驾驭。

按照软件工程思想，软件测试可以分为单元测试、集成测试、功能测试、系统测试等。功能测试和系统测试一般来说是测试人员的职责，但单元测试和集成测试则必须由开发人员保证。

20.1.1　为什么需要单元测试

软件开发的标准过程包括以下几个阶段：『需求分析阶段』→『设计阶段』→『实现阶段』→『测试阶段』→『发布』。其中，测试阶段通过人工或者自动手段来运行或测试某个系统的过程，其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别。测试过程按 4 个步骤进行，即单元测试、集成测试、系统测试及发版测试。其中，功能测试主要检查已实现的软件是否满足需求规格说明中确定的各种需求，以及软件功能是否完全、正确。系统测试主要对已经过确认的软件纳入实际运行环境中，与其他系统成分组合在一起进行测试。单元测试、集成测试由开发人员进行，是我们关注的重点，下文对两者进行详细说明。

1．单元测试

单元测试是开发者编写的一小段代码，用于检验目标代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试用于判断某个特定条件或特定场景下某个特定函数的行为。例如，用户可能把一个很大的值放入一个有序 List 中，然后确认该值出现在 List 的尾部。或者，用户可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些字符。

单元测试由程序员自己来完成，最终受益的也是程序员自己。可以这么说，程序员有责任编写功能代码，同时也有责任为自己的代码编写单元测试。执行单元测试，就是为了证明这段代码的行为和我们期望的一致。

在一般情况下，一个功能模块往往会调用其他功能模块完成某项功能，如业务层的业务类可能会调用多个 DAO 完成某项业务。在对某个功能模块进行单元测试时，我们希望屏蔽对外在功能模块的依赖，以便将焦点放在目标功能模块的测试上。这时模拟对象将是最有力的工具，它根据外在模块的接口模拟特定的操作行为，这样单元测试就可以在假设关联模块正确工作的情况下验证本模块逻辑的正确性。

2．集成测试

单元测试和开发工作是并驾齐驱的，甚至是前置性的工作。除了一些显而易见的功能外，大部分功能（类的方法）都必须进行单元测试，通过单元测试可以保障功能模块的正确性。而集成测试则是在功能模块开发完成后，为验证功能模块之间匹配调用的正确性而进行的测试。在单元测试时，往往需要通过模拟对象屏蔽外在模块的依赖，而集成测试恰恰是要验证模块之间集成后的正确性。

举个例子，当对 UserService 这个业务层的类进行单元测试时，可以通过创建 UserDao 和 LoginLogDao 模拟对象，在假设 DAO 类正确工作的情况下对 UserService 进行测试。而对 UserService 进行集成测试时，则应该注入真实的 UserDao 和 LoginLogDao 进行测试。

所以，一般来讲，集成测试面向的层面要更高一些，一般对业务层和 Web 层进行集成测试；单元测试则面向一些功能单一的类（如字符串格式化工具类、数据计算类）。当然，我们可能对某个类既进行单元测试又进行集成测试，如 UserService 在模块开发期间进行单元测试，而在关联的 DAO 类开发完成后再进行集成测试。

3．测试的好处

在编写代码的过程中，一定会反复调试以保证它能够编译通过。但代码通过编译，只是说明了它的语法正确，而无法保证它的语义也一定正确，没有任何人可以轻易承诺这段代码的行为一定是正确的。幸运的是，单元测试会为我们的承诺提供保证。编写单元测试就是为了验证这段代码的行为是否与我们期望的一致。有了单元测试，我们就可以自信地交付自己的代码，减少后顾之忧。总之，进行单元测试，会带来以下好处：

- 软件质量最简单、最有效的保证。
- 是目标代码最清晰、最有效的文档。
- 可以优化目标代码的设计。
- 是代码重构的保障。
- 是回归测试和持续集成的基石。

20.1.2 单元测试之误解

认为单元测试影响开发进度，一是借口，拒绝对单元测试相关知识进行学习（单元测试、代码重构、版本管理是开发人员的必备技能）；二是单元测试是“先苦后甜”，刚开始搭建环境，引入额外工作，看似“影响进度”，但从长远来看，由于程序质量提升、代码返工减少、后期维护工作量缩小、项目风险降低，从而在整体上赢了回来。

误解一：影响开发进度

一旦编码完成，开发人员总是会迫切希望进行软件的集成工作，这样他们就能够看到系统的实际运行效果。这在外表上看来好像加快了进度，而像单元测试这样的活动被看作影响进度的原因之一，推迟了对整个系统进行集成测试的时间。

在实践中，这种开发步骤常常会导致这样的结果：软件甚至无法运行。更进一步的结果是大量的时间将被花费在跟踪那些包含在独立单元里的简单 Bug 上面。在个别情况下，这些 Bug 也许是琐碎和微不足道的，但总的来说，它们会推迟软件产品交付的时间，而且也无法确保软件产品能够可靠运行。

在实际工作中，进行完整计划的单元测试和编写实际的代码所花费的精力大致上是相同的。一旦完成了这些单元测试工作，很多 Bug 将被纠正，开发人员能够进行更高效的系统集成工作。这才是真实意义上的进步，所以说完整计划下的单元测试是对时间的更高效利用。

误解二：增加开发成本

如果不重视程序中那些未被发现的 Bug 可能带来的后果，则这种后果的严重程度可以从一个 Bug 引起的用户使用不便到系统崩溃。这种后果可能常常会被软件的开发人员所忽视，这种情况会长期损害软件开发商的声誉，并且会对未来的市场产生负面影响。相反，一个可靠的软件系统的良好声誉将有助于一个软件开发商获取未来的市场。

很多研究成果表明，无论什么时候做出修改都要进行完整的回归测试，在生命周期中尽早地对软件产品进行测试将使效率和质量得到最好的保证。Bug 发现得越晚，修改它所需的费用就越高，因此，从经济角度来看，应该尽可能早地查找和修改 Bug。而单元测试就是一个在早期抓住 Bug 的机会。

相比后阶段的测试，单元测试的创建更简单，且维护更容易，同时可以更方便地进行重构。从全程的费用来考虑，相比于那些复杂且旷日持久的集成测试，或不稳定的软件系统，单元测试所需的费用是很低的。

误解三：我是个编程高手，无须进行单元测试

在每个开发团队中至少有一个这样的开发人员，他非常擅长于编程，他开发的软件总是在第一时间就可以正常运行，因此不需要进行测试。你是否经常听到这样的借口？在现实世界里，每个人都会犯错误。即使某个开发人员可以抱着这种态度在很少的一些简单程序中应付过去，但真正的软件系统是非常复杂的。真正的软件系统不可以寄希望于没有进行广泛的测试和 Bug 修改过程就可以正常工作。编码不是一个可以一次性通过的过程。在现实世界中，软件产品必须进行维护以对功能需求的改变做出及时响应，并且要对最初的开发工作遗留下来的 Bug 进行修改。你希望依靠那些原始作者进行修改吗？这些制造出未经测试的代码的资深工程师还会继续在其他地方制造这样的代码。在开发人员做出修改后进行可重复的单元测试，可以避免产生那些令人不快的副作用。

误解四：测试人员会测出所有 Bug

一旦软件可以运行，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在创造一种单元调用的测试条件时，要全面考虑单元被调用时的各种入口参数。在软件集成阶段，对单元功能全面测试的复杂程度远远超过独立进行的单元测试过程。

最后的结果是测试将无法达到它应有的全面性，一些缺陷将被遗漏，并且很多 Bug 将被忽略。让我们类比一下：假设我们要清理一台电脑主机中的灰尘，如果没有把主机中的各个部件（显卡、内存等）拆开，无论你用什么工具，一些灰尘还会隐藏在主机的某些角落无法清理。但我们换个角度想想，如果把主机的每个部件一一拆开，这些死角里的灰尘就容易被发现和接触到了，并且每个部件的灰尘都可以毫不费力地进行清理。

20.1.3 单元测试之困境

测试在软件开发过程中一直都是备受关注的，测试不仅仅局限于软件开发中的一个阶段，它已经开始贯穿于整个软件开发过程。大家普遍认识到，如果测试能在开发阶段进行有效执行，程序的 Bug 就会被及早发现，其质量就能得到有效的保证，从而减少软件开发总成本。但是，相对于测试这个词的流行程度而言，大家对单元测试的认知普遍存在一些偏差，特别是一些程序员很容易陷入一些误区，导致测试并没有在他们所在的开发项目中起到有效的作用。下面对一些比较具有代表性的误区困境进行剖析，并对测试背后所蕴含的一些设计思考进行阐述，希望能够起到抛砖引玉的作用。

误区、困境一：使用 System.out.print 跟踪和运行程序就够了

这个误区可以说是程序员的一种通病，认为使用 System.out.print 就可以确保编写代码的正确性，无须编写测试用例，他们觉得编写测试用例是在“浪费时间”。使用 System.out.print 输出结果，以肉眼观察这种刀耕火种的方式进行测试，不仅效率低下，而且容易出错。

误区、困境二：存在太多无法测试的东西

在编码的时候，确实存在一些看起来比较难测试的代码，但并非无法测试。并且在大多数情况下，是因为被测试的代码在设计时没有考虑到可测试性的问题。编写的程序不仅与第三方一些框架耦合过紧，而且过于依赖其运行环境，从而表现出被测试的代码本身很难测试。

误区、困境三：测试代码可以随意写

编写测试代码时抱着一种随意的态度，没有弄清测试的真正意图。编写测试代码只是为了应付任务而已，先编写程序实现代码，然后再去编写一些单元测试。表现出来的结果是测试过于简单，只走形式和花架，将大量 Bug 传递给系统测试人员。

误区、困境四：不关心测试环境

手工搭建测试环境，测试数据，造成维护困难，占据了大量时间，严重影响效率。对测试产生的“垃圾”不清除、不处理，造成测试不能重复进行，导致脆弱的测试。需要维护好测试环境，做一个“低碳环保”的测试者。

误区、困境五：测试环境依赖性大

测试环境依赖性大，没有有效隔离测试目标及其依赖环境。一是使测试不聚焦；二是常因依赖环境的影响造成失败；三是因依赖环境太厚重从而降低测试的效率（依赖数据库或依赖网络资源，如邮件系统、Web 服务）。

20.1.4　单元测试基本概念

被测系统：SUT（System Under Test）

被测系统（System Under Test，SUT）表示正在被测试的系统，目的是测试系统能否正确操作。这一词语常用于软件测试中。软件系统测试的一个特例是对应用软件的测试，称为被测应用程序（Application Under Test，AUT）。

SUT 也表明软件已经到了成熟期，因为系统测试在测试周期中是集成测试的后一阶段。

测试替身：Test Double

在进行单元测试时，使用 Test Double 减少对被测对象的依赖，使得测试更加单一。同时，让测试案例执行的时间更短，运行更加稳定，同时能对 SUT 内部的输入/输出进行验证，让测试更加彻底深入。但是，Test Double 也不是万能的，Test Double 不能被过度使用，因为实际交付的产品是使用实际对象的，过度使用 Test Double 会让测试变得越来越脱离实际。

测试夹具：Test Fixture

所谓测试夹具（Fixture），就是测试运行程序（Test Runner）会在测试方法之前自动执行初始化、回收资源的工作。在 TestNG 中，通过@BeforeMethod 注解标注的方法进行初始化工作；通过@AfterMethod 注解标注的方法进行资源的回收工作。在一个测试类中，甚至可以使用多个@BeforeMethod 注解标注多个方法，这些方法都是在每个测试之前运行的。需要说明一点，@BeforeMethod 是在每个测试方法运行前均初始化一次；同理，@AfterMethod 是在每个测试方法运行完毕后均执行一次。也就是说，经过这两个注解的初始化和注销，可以保证各个测试之间的独立性而互不干扰，其缺点是效率低。另外，不需要在超类中显式调用初始化和清除方法，只要它们不被覆盖，测试运行程序将根据需要自动调用这些方法。超类中@BeforeMethod 注解标注的方法在子类的@BeforeMethod 方法之前调用（与构造函数调用顺序一致），@AfterMethod 注解标注的方法是子类在超类之前运行。

一个测试用例可以包含若干个标注@Test 注解的测试方法，测试用例测试一个或多个类 API 接口的正确性。当然，在调用类 API 时，需要事先创建这个类的对象及一些关联的对象，这组对象就被称为测试夹具，相当于测试用例的“工作对象”。

前面讲过，一个测试用例类可以包含多个标注@Test 注解的测试方法，在运行时，每个测试方法都对应一个测试用例类的实例。当然，用户可以在具体的测试方法中声明并实例化业务类的实例，在测试完成后销毁它们。但是，这样一来就要在每个测试方法中都重复这些代码，因为 TestCase 实例依照以下步骤运行。

① 创建测试用例的实例。

② 使用@BeforeMethod 注解修饰用于初始化夹具的方法。

③ 使用@AfterMethod 注解修饰用于注销夹具的方法。

④ 保证这两种方法不能带有任何参数。

TestCase 实例运行过程如图 20-1 所示。

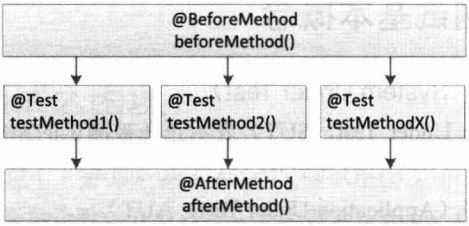


图 20-1　方法级别夹具执行示意图

之所以每个测试方法都需要按以上流程运行，是为了防止测试方法相互之间的影响，因为在同一个测试用例类中不同的测试方法可能会使用相同的测试夹具，前一个测试方法对测试夹具的更改会影响后一个测试方法的现场。而通过如上的运行步骤后，因为每个测试方法在运行前都重建了运行环境，所以测试方法相互之间就不会有影响了。

但是，这种夹具设置方式还是引来了批评，因为它效率低下，特别是在设置 Fixture 非常耗时的情况下（如设置数据库连接）。而且对于不会发生变化的测试环境或者测试数据来说，是不会影响到测试方法的执行结果的，也就没有必要针对每个测试方法重新设置一次夹具。因此，在 TestNG 中引入了类级别的夹具设置方法，编写规范说明如下。

① 创建测试用例的实例。

② 使用@BeforeClass 注解修饰用于初始化夹具的方法。

③ 使用@AfterClass 注解修饰用于注销夹具的方法。

④ 保证这两种方法不能带有任何参数。

类级别的夹具仅会在测试类中的所有测试方法执行之前执行初始化，并在全部测试方法测试完毕之后执行注销方法，如图 20-2 所示。

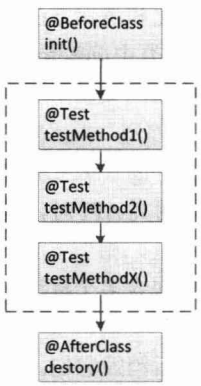


图 20-2　类级别夹具执行示意图

此外，TestNG 还提供了分组级、套件级注解 BeforeGroups、AfterGroups、BeforeSuite、AfterSuite 用来处理测试分组、套件初始化及注销夹具的方法。

测试用例：Test Case

有了测试夹具，就可以开始编写测试用例的测试方法了。当然也可以不需要测试夹具而直接编写测试用例的测试方法。在 TestNG 中编写测试方法非常简单，只需在每个测试方法中标注@Test 注解即可。

可以在一个测试用例中添加多个测试方法，运行器会为每个方法生成一个测试用例实例并分别运行。

测试套件：Test Suite

如果每次只能运行一个测试用例，那么又陷入了传统测试（使用 main()方法进行测试）的窘境：手工运行一个个测试用例，这是非常烦琐和低效的。测试套件专门为解决这一问题而生。它通过 TestSuite 对象将多个测试用例组装成一个测试套件，则测试套件批量运行。需要特别指出的是，可以把一个测试套件整个添加到另一个测试套件中，就像小筐装进大筐里变成一个筐一样。

20.2　TestNG 快速进阶

20.2.1　TestNG 概述

TestNG 是一个设计用来简化广泛的测试需求的测试框架，其灵感来自 JUnit 和 NUnit，但引入了一些新的功能，使其功能更强大，使用更方便。TestNG 消除了大部分旧框架的限制，使开发人员能够编写更加灵活和强大的测试。因为它在很大程度上借鉴了 Java 注解（Java 5.0 引入的）来定义的测试，它也可以告诉你如何在真实的 Java 语言生产环境中使用这个新功能。TestNG 添加了诸如灵活的装置、测试分类、参数测试、依赖方法、数据驱动等特性，让开发人员编写测试更加灵活、简便。

TestNG 设计的出发点是不仅可以用于单元测试，而且可以用于集成测试。相比于 JUnit 只适合单元测试，TestNG 无疑会走得更远。

编写一个测试的过程有 3 个典型步骤。

① 编写测试的业务逻辑并在代码中插入 TestNG 注解。

② 将测试信息添加到 testng.xml 或者 build.xml 文件中。

③ 运行 TestNG。

20.2.2 TestNG 生命周期

TestNG 测试用例的完整生命周期要经历以下阶段：类级初始化资源处理、方法级初始化资源处理、执行测试用例中的方法、方法级销毁资源处理、类级销毁资源处理。其中，类级初始化、销毁资源处理方法在一个测试用例类中只能运行一次；方法级初始化、销毁资源处理方法在执行测试用例的每个测试方法中都会运行一次，以防止测试方法相互之间的影响。测试用例的执行过程如图 20-3 所示。

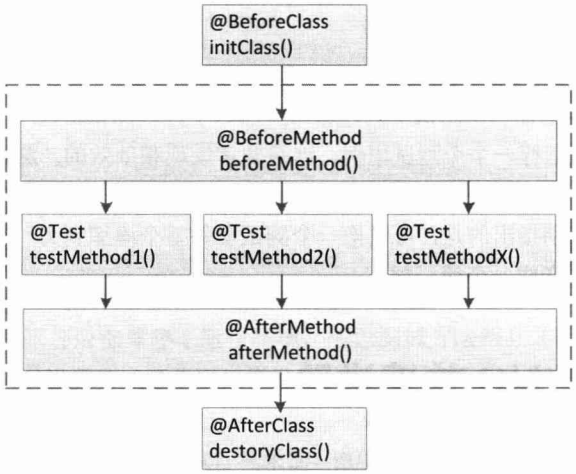


图 20-3 TestNG 测试用例执行示意图

如果在一个测试用例中编写了多个初始化处理方法，则运行时先执行位于最后面的初始化方法，然后往前一个个执行初始化方法。对于多个销毁资源处理方法，则按照方法的顺序一个个往后执行。

20.2.3 使用 TestNG

1. 测试方法

在 TestNG 中使用@Test 注解来标注一个测试方法。此外可以采用 Java 5.0 的静态导入功能导入断言 Assert 类，这样就可以很方便地在测试方法中使用断言方法。下面通过一个实例来快速体验 TestNG 测试方法，如代码清单 20-1 所示。

代码清单 20-1 测试方法

```
package sample.testng;
import org.testng.annotations.*;
import static org.testng.Assert.*;
public class MoneyTest{
    private Money f12CHF;//12 瑞士法郎
    private Money f14CHF; //14 瑞士法郎
     private Money f28USD; //28 美国美元
```

```
    @BeforeMethod
    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
                              测试方法，在测试方法中标注@Test 注解，方法
                              签名可以任意取名，可以声明抛出异常
    @Test
    public void moneyBag(){//①
                                                    可以设定若干个断言方法，这些断言
                                                    是评判被测试功能是否正确的依据
        Money bag[]= { f26CHF, f28USD };
        MoneyBag expected= new MoneyBag(bag);
        assertEquals(expected, f12CHF.add(f28USD.add(f14CHF))); //②
    }

    @AfterMethod
    protected void tearDown(){}
}
```

在 TestNG 中，只要在每个测试方法中添加@Test 注解即可。像②处的 assertYyy()断言方法就是测试 Money 的 add()方法功能运行正确性的测试规则。

可以在 MoneyTest 中添加多个测试方法，运行器会为每个方法生成一个测试用例实例并分别运行。

2．@BeforeClass 和@AfterClass

在 TestNG 中加入了两个注解：@BeforeClass 和@AfterClass，使用这两个注解的方法，在一个 Test 类的所有测试方法执行前后各执行一次。这是为了能在 @BeforeClass 中初始化一些昂贵的资源，如数据库连接，然后执行所有的测试方法，最后在@AfterClass 中释放资源。对于初学者来讲，很容易混淆@BeforeClass/@AfterClass 与@BeforeMethod/@AfterMethod，为此表 20-1 对它们作了一下对比。

表 20-1　@BeforeClass/@AfterClass与@BeforeMethod/@AfterMethod的区别

@BeforeClass/@AfterClass	@BeforeMethod/@AfterMethod
在一个类中只能出现一次	在一个类中可以出现多次，执行顺序不确定
方法名不作限制	方法名不作限制
在类中只执行一次	在每个测试方法之前或者之后都会执行一次
@BeforeClass 父类中标识了该注解的方法将会先于当前类中标识了该注解的方法执行。 @AfterClass 父类中标识了该注解的方法将会在当前类中标识了该注解的方法之后执行	@BeforeMethod 父类中标识了该注解的方法将会先于当前类中标识了该注解的方法执行。 @AfterMethod 父类中标识了该注解的方法将会在当前类中标识了该注解的方法之后执行
必须声明为 public static	必须声明为 public，并且非 static
所有标识为@AfterClass 的方法一定会被执行，即使在标识为@BeforeClass 的方法抛出异常的情况下也一样会被执行	所有标识为@AfterMethod 的方法一定会被执行，即使在标识为@BeforeMethod 或者@Test 的方法抛出异常的情况下也一样会被执行

3. 异常测试

因为使用了注解特性，所以 TestNG 测试异常非常简单明了。通过对@Test 传入 expected 参数值，即可测试异常。在传入异常类后，测试类如果没有抛出异常或者抛出一个不同的异常，本测试方法就将失败。如代码清单 20-2 所示为一个简单的异常测试实例。

代码清单 20-2 异常测试

```
package sample.testng;
import java.util.*;
…
public class TestNGExceptionTest {
   private User user;

   @BeforeClass
   public void init() {
      user = null;
   }

  @Test(enabled = true, expectedExceptions =NullPointerException.class)
   public void testUser(){
     assertNotNull(user.getUserName());
   }
}
```

预期抛出空指针异常

4. 超时测试

通过在@Test 注解中为 timeOut 参数指定时间值，即可进行超时测试。如果测试运行时间超过指定的毫秒数，则测试失败。超时测试对网络链接类非常重要。通过 timeOut 进行超时测试非常简单，如代码清单 20-3 所示为一个简单的超时测试实例。

代码清单 20-3 超时测试

```
package sample.testng;
import java.util.*;
…
public class TestNGTimeoutTest {
    …
    @Test(timeOut = 10)
    public void testUser(){
      assertNotNull(user);
      assertEquals( user.getUserName(),"admin");
    }
}
```

指定测试方法的超时时间

5. 参数化测试

为了测试程序的健壮性，可能需要模拟不同的参数对方法进行测试。如果为每个类型的参数创建一个测试方法，则是一件很难接受的事。幸好 TestNG 提供了参数化测试，它能够创建由参数值供给的通用测试，从而为每个参数都运行一次，而不必创建多个测试方法，如代码清单 20-4 所示。

代码清单 20-4　参数化测试

```
package sample.testng;
import static org.testng.Assert.*;
import org.testng.annotations.*;

public class TestNGParameterTest {
    private SimpleDateFormat simpleDateFormat;

                                          参数数据提供者
    @DataProvider(name = "testParam")
    public static Object[][] getParamters() {
        String[][] params = {
            { "2016-02-01 00:30:59", "yyyyMMdd", "20160201" },
            { "2016-02-01 00:30:59", "yyyy年MM月dd日", "2016年02月01日" },
            { "2016-02-01 00:30:59", "HH时mm分ss秒", "00时30分59秒" } };
        return params;
    }
                              @Test 属性 dataProvider 的值要与参数
                              数据提供者的 dataProvider 值保持一致
   @Test(dataProvider = "testParam")
   public void testSimpleDateFormat() throws ParseException{
      SimpleDateFormat df  = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
       Date d = df.parse(this.date);
      simpleDateFormat = new SimpleDateFormat(this.dateformat);
      String result = simpleDateFormat.format(d);
      assertEquals( result, expectedDate);
   }
}
```

首先编写测试类的参数数据提供者方法，然后用此方法进行参数初始化。该方法返回一个 Object[][]类型。用@DataProvider 注解来标注该方法，并设置 name 值。之后在需要测试的方法中设置@Test 的 dataProvider 属性，其值要和上面@DataProvider 修饰的方法中的 name 值保持一致。

6. 分组测试

TestNG 支持执行复杂的分组测试。不仅可以声明单个测试用例内的测试方法的分组，而且还可以声明不同测试用例类级的分组。在执行 TestNG 测试时，可以指定要执行的测试分组及排除不执行的测试分组，如代码清单 20-5 所示。

代码清单 20-5　分组测试

```
package sample.testng;
import org.testng.annotations.*;
…
@Test(groups = {"class-group"})
public class TestNGGroupsTest1 {
                                  设置用例级分组

   @Test(groups = {"group1", "group2"})
   public void testMethod1() {
   }
                                  设置测试方法级分组，一个
                                  方法可以属于多个分组

   @Test(groups = {"group1", "group2"})
```

```
    public void testMethod2() {
    }

    @Test(groups = {"group1"})
    public void testMethod3() {
    }
}

@Test(groups = {"class-group"})
public class TestNGGroupsTest2 {

    @Test(groups = {"group1", "group2"})
    public void testMethod3() {
    }
    …
}
```

7．依赖测试

有些时候，我们需要测试方法按照一个特定的顺序被调用。这非常有用，比如，在运行某个测试方法前需要先运行特定的测试方法，或希望初始化方法也作为测试方法（因为被标注为@BeforeXXX/@AfterXXX 的方法不作为测试报告的一部分）。为了实现这些需求，TestNG 为@Test 注解提供了 dependsOnMethods 或 dependsOnGroups 属性来实现测试方法间的依赖关系，如代码清单 20-6 所示。

代码清单 20-6　依赖测试

```
package sample.testng;
import org.testng.annotations.*;
…
public class TestNGGroupsTest1 {

    @Test(dependsOnMethods= {" testMethod1", " testMethod2"})
    public void testMethod3() {
    }

    @Test
    public void testMethod1() {
    }

    @Test
    public void testMethod2() {
    }
}
```

设置在当前测试方法执行之前被调用的测试方法

测试方法 testMethod3()设置了两个依赖的测试方法 testMethod1()及 testMethod2()，当执行 testMethod3()测试方法时，会先调用两个依赖的测试方法。如果 testMethod1()或 testMethod2()中的任何一个方法测试失败，则 testMethod3()将不被执行。只有依赖的方法测试全部通过时，当前方法才会被调用执行，这种依赖关系被称为强依赖，也是 TestNG 默认的依赖关系。可以通过@Test 提供的 alwaysRun 属性来改变这种强依赖，如示例中改成@Test(dependsOnMethods= {" testMethod1", " testMethod2"}, alwaysRun=true)之后，

不管测试方法 testMethod1()或 testMethod2()有没有通过测试，testMethod3()总会被执行，这种依赖关系被称为软依赖。

20.3　模拟利器 Mockito

20.3.1　模拟测试概述

目前支持 Java 语言的 Mock 测试工具有 EasyMock、JMock、Mockito、MockCreator、Mockrunner、MockMaker 等，其中 Mockito 是一个针对 Java 的 Mocking 框架。它与 EasyMock 和 JMock 很相似，是一套通过简单的方法对指定的接口或类生成 Mock 对象的类库，避免了手工编写 Mock 对象的烦琐。但 Mockito 通过在执行后校验是否已经被调用，它消除了对期望行为（Expectations）的需要。使用 Mockito，在准备阶段只需花费很少的时间，可以使用简洁的 API 编写出漂亮的测试，而且可以对具体的类创建 Mock 对象，并且有“监视”非 Mock 对象的能力。

Mockito 使用起来非常简单，学习成本很低，而且具有非常简洁的 API，测试代码的可读性很高，因而十分受欢迎，用户群越来越多，很多开源软件也选择了 Mockito。要想了解更多有关 Mockito 的信息，可以访问其官方网站 http://www.mockito.org/。在开始使用 Mockito 之前，先简单了解一下 Stub 和 Mock 的区别。相比于 EasyMock、JMock，其编写出来的代码更加容易阅读。无须录制 mock()方法调用就返回默认值是一个很大的优势。目前最新的版本是 1.10.19。

Stub 对象用来提供测试时所需的测试数据，可以对各种交互设置相应的回应。例如，可以设置方法调用的返回值等。在 Mockito 中，when(…).thenReturn(…) 这样的语法便是设置方法调用的返回值。另外也可以设置方法在何时调用会抛出异常等。

Mock 对象用来验证测试中所依赖对象间的交互是否能够达到预期。在 Mockito 中用 verify(…).methodXxx(…) 语法来验证 methodXxx()方法是否按照预期进行了调用。有关 Stub 和 Mock 的详细论述请见 Martin Fowler 的文章 *Mocks Aren't Stub*，地址为 http://martinfowler.com/articles/mocksArentStubs.html。在 Mocking 框架中，所谓的 Mock 对象实际上是作为上述 Stub 和 Mock 对象同时使用的，因为它既可以设置方法调用的返回值，又可以验证方法的调用。

20.3.2　创建 Mock 对象

可以对类和接口进行 Mock 对象的创建，创建的时候可以为 Mock 对象命名，也可以忽略命名参数。为 Mock 对象命名的好处就是调试的时候会很方便。比如，我们 Mock 多个对象，在测试失败的信息中会把有问题的 Mock 对象打印出来，有了名字就可以很容易地定位和辨认出是哪个 Mock 对象出现了问题。另外，其使用也有限制，对于 final

类、匿名类和 Java 的基本类型，是无法进行 Mock 的。除了用 Mock 方法来创建模拟对象，如 mock(Class<T> classToMock)，也可以使用@mock 注解定义 Mock。下面通过实例来介绍一下如何创建一个 Mock 对象，如代码清单 20-7 所示。

代码清单 20-7　MockitoSampleTest.java创建Mock对象

```
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
import static org.mockito.Mockito.*;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.testng.annotations.*;
import static org.testng.Assert.*;
…
public class MockitoSampleTest{

    //① 对接口进行模拟
    UserService mockUserService = mock(UserService.class);

    //② 对类进行模拟
    UserServiceImpl mockServiceImpl = mock(UserServiceImpl.class);

   //③ 基于注解模拟类
   @Mock
   User mockUser;

   @BeforeClass
   public void initMocks() {

      //④ 初始化当前测试类所有@Mock注解模拟对象
      MockitoAnnotations.initMocks(this);
   }
    …
}
…
```

在①处和②处，通过 Mockito 提供的 mock()方法创建 UserService 用户服务接口、用户服务实现类 UserServiceImpl 的模拟对象。在③处，通过@Mock 注解创建用户 User 类的模拟对象，并且需要在测试类初始化方法中，通过 MockitoAnnotations.initMocks()方法初始化当前测试类中所有标注@Mock 注解的模拟对象。如果没有执行这一步初始化操作，则测试时会报模拟对象为空对象异常。

20.3.3　设定 Mock 对象的期望行为及返回值

从上文中我们已经知道，可以通过 when(mock.someMethod()).thenReturn(value)来设定 Mock 对象的某个方法调用时的返回值，但它同样有限制条件，即对于 static 和 final 修饰的方法是无法进行设定的。下面通过实例来介绍一下如何调用方法及设定返回值，如代码清单 20-8 所示。

代码清单 20-8　MockitoSampleTest.java设定模拟对象的期望行为及返回值

```
import org.testng.annotations.*;
import org.mockito.Mock;
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
…
public class MockitoSampleTest {
    …
    //① 模拟接口UserService测试
    @Test
    public void testMockInterface() {

        //①-1 对方法设定返回值
    when(mockUserService.findUserByUserName("tom")).thenReturn(
                new User("tom", "1234"));

       //①-2 对方法设定返回值
       doReturn(true).when(mockServiceImpl).hasMatchUser("tom", "1234");

        //①-3 对void方法进行方法预期设定
       User u = new User("John", "1234");
        doNothing().when(mockUserService).registerUser(u);

       //①-4 执行方法调用
        User user = mockUserService.findUserByUserName("tom");
        boolean isMatch = mockUserService.hasMatchUser("tom","1234");
        mockUserService.registerUser(u);

        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);
    }

    //② 模拟实现类UserServiceImpl测试
   @Test
    public void testMockClass() {

        //对方法设定返回值
        when(mockServiceImpl.findUserByUserName("tom"))
                              .thenReturn(new User("tom", "1234"));
        doReturn(true).when(mockServiceImpl).hasMatchUser("tom", "1234");

        User user = mockServiceImpl.findUserByUserName("tom");
        boolean isMatch = mockServiceImpl.hasMatchUser("tom","1234");
        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);
    }

    //③ 模拟User类测试
   @Test
    public void testMockUser() {
```

```
        when(mockUser.getUserId()).thenReturn(1);
        when(mockUser.getUserName()).thenReturn("tom");
        assertEquals(mockUser.getUserId(),1);
        assertEquals(mockUser.getUserName(), "tom");
    }
```

在①处，模拟测试接口 UserService 的 findUserByUserName()、hasMatchUser()及 registerUser()方法。在①-1 处通过 when().thenReturn()语法，模拟方法调用及设置方法的返回值，实例通过模拟调用 UserService 用户服务接口的查找用户 findUserByUserName()方法，查询用户名为“tom”的详细信息，并设置返回 User 对象：new User("tom", "1234")。在①-2 处通过 doReturn (). when ()语法，模拟判断用户 hasMatchUser()方法的调用，判断用户名为“tom”、密码为“1234”的用户是否存在，并设置返回值为 true。在①-3 处对 void 方法进行方法预期设定，如实例中调用注册用户 registerUser()方法。在设定调用方法及返回值之后，就可以执行接口方法调用验证。在②处和③处，模拟测试用户服务实现类 UserServiceImpl，测试的方法与模拟接口一致。

20.3.4 验证交互行为

Mock 对象一旦建立便会自动记录自己的交互行为，所以我们可以有选择地对其交互行为进行验证。在 Mockito 中验证 Mock 对象交互行为的方法是 verify(mock). xxx()，于是用此方法验证了 findUserByUserName()方法的调用。因为只调用了一次，所以在 verify 中指定了 times 或 atLeastOnce()参数。最后验证返回值是否和预期一样，如代码清单 20-9 所示。

代码清单 20-9 MockitoSampleTest.java验证交互行为

```
import org.testng.annotations.*;
import org.mockito.Mock;
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
…
public class MockitoSampleTest {
    …

    //① 模拟接口UserService测试
    @Test
    public void testMockInterface() {
        …
    when(mockUserService.findUserByUserName("tom"))
                        .thenReturn(new User("tom", "1234"));
       User user = mockServiceImpl.findUserByUserName("tom");

        //①-4 验证返回值
         assertNotNull(user);
         assertEquals(user.getUserName(), "tom");
         assertEquals(isMatch, true);
```

```
        //①-5 验证交互行为
        verify(mockUserService).findUserByUserName("tom");

        //①-6 验证方法至少调用一次
        verify(mockUserService, atLeastOnce()).findUserByUserName("tom");
        verify(mockUserService, atLeast(1)).findUserByUserName("tom");

        //①-7 验证方法至多调用一次
        verify(mockUserService, atMost(1)).findUserByUserName("tom");
    }
...
```

Mockio 为我们提供了丰富调用方法次数的验证机制，如被调用了特定次数 verify(xxx, times(x))、至少 *x* 次 verify(xxx, atLeast (*x*))、最多 *x* 次 verify(xxx, atMost (*x*))、从未被调用 verify(xxx, never())。在①-6 处，验证 findUserByUserName()方法至少被调用一次。在①-7 处，验证 findUserByUserName()方法至多被调用一次。

20.4　测试整合之王 Unitils

20.4.1　Unitils 概述

Unitils 测试框架的目的是让单元测试变得更加容易和可维护。Unitils 构建在 DbUnit 与 EasyMock 项目之上，并与 JUnit 和 TestNG 相结合。它不仅支持数据库测试，而且支持利用 Mock 对象进行测试，并且可以与 Spring 和 Hibernate 相集成。Unitils 被设计成以一种高度可配置和松散耦合的方式来添加这些服务到单元测试中，目前最新版本是 3.4.2。要使用 Unitils，首先需要在章节模块 pom.xml 中添加 Unitils 的相关依赖。

1. Unitils 功能特点

- 自动维护和强制关闭单元测试数据库（支持 Oracle、HSQLDB、MySQL、DB2）。
- 简化单元测试数据库连接的设置。
- 简化利用 DbUnit 测试数据的插入。
- 简化 Hibernate session 管理。
- 自动测试与数据库相映射的 Hibernate 映射对象。
- 易于把 Spring 管理的 Bean 注入单元测试中，支持在单元测试中使用 Spring 容器中的 Hibernate SessionFactory。
- 简化 EasyMock Mock 对象创建。
- 简化 Mock 对象注入，利用反射等式匹配 EasyMock 参数。

2. Unitils 模块组件

Unitils 通过模块化的方式来组织各个功能模块，采用类似于 Spring 的模块划分方式，如 unitils-core、unitils-database、unitils-mock 等，比以前整合在一个工程里面显得更加清晰。目前包含的所有模块如下。

- unitils-core：核心内核包。
- unitils-database：维护测试数据库及连接池。
- unitils-DbUnit：使用 DbUnit 来管理测试数据。
- unitils-easymock：支持创建 Mock 和宽松的反射参数匹配。
- unitils-inject：支持在一个对象中注入另一个对象。
- unitils-mock：整合各种 Mock，在 Mock 的使用语法上进行了简化。
- unitils-orm：支持 Hibernate、JPA 的配置和自动数据库映射检查。
- unitils-spring：支持加载 Spring 的上下文配置，并加载 Spring Bean 注入。

Unitils 的核心架构中包含 Moudule 和 TestListener 两个概念，类似于 Spring 中粘连其他开源软件中的 FactoryBean 概念，可以看成第三方测试工具的一个黏合剂。整体框架如图 20-4 所示。

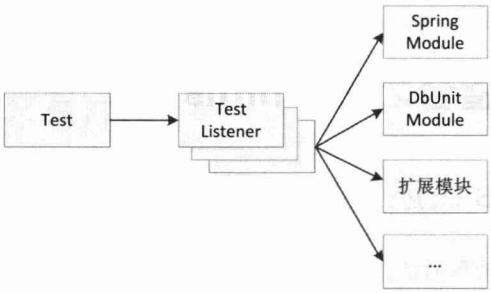


图 20-4　Unitils 架构图

通过 TestListener 可以在测试运行的不同阶段注入某些功能，同时某个 TestListener 又被一个对应的 Module 所持有。Unitils 也可以被看作一个插件体系结构，TestListener 在整个 Unitils 中又充当了插件中扩展点的角色。从 TestListener 接口中可以看到，它可以在 crateTestObject、before(after)Class、before(after)TestMethod、beforeSetup、afterTeardown 的不同切入点添加不同的动作。

3．Unitils 配置文件

- unitils-defaults.properties：默认配置文件，开启所有功能。
- unitils.properties：项目级配置文件，用于项目通用属性配置。
- unitils-local.properties：用户级配置文件，用于个人特殊属性配置。

Unitils 定义了通用的配置文件名 unitils.properties 和用户自定义配置文件名 unitils-local.properties，并给出了默认的模块及模块对应的 className，便于 Unitils 加载对应的模块。但是如果用户分别在 unitils.properties 和 unitils -local.properties 文件中对相同属性配置不同值，则将会以 unitils-local.properties 的配置内容为主。

4．Unitils 断言

典型的单元测试一般都包含一个重要的组成部分：对比实际产生的结果和希望的结果是否一致的方法，即断言方法（assertEquals）。Unitils 为我们提供了一个非常实用的

断言方法，我们以第 2 章中编写的用户领域对象 User 为蓝本，比较两个 User 对象的实例，来开始认识 Unitils 的断言之旅。

5．assertReflectionEquals：反射断言

在 Java 世界中，要比较现有两个对象实例是否相等，如果类没有重写 equals()方法，则用两个对象的引用是否一致作为判断依据。有时候，我们并不需要关注两个对象是否引用同一个对象，只要两个对象的属性值一样就可以了。在 TestNG 单元测试中，有两种测试方式进行这样的场景测试：一是在比较实体类中重写 equals()方法，然后进行对象比较；二是把对象实例的属性一个一个进行比较。不管采用哪种方法，都比较烦琐。Unitils 为我们提供了一种非常简单的方法，即使用 ReflectionAssert.assertReflectionEquals 方法，如代码清单 20-10 所示。

代码清单 20-10　assertReflectionEquals反射断言测试

```
package sample.unitils;
import org.testng.annotations.*;
import com.smart.domain.User;
import org.unitils.reflectionassert.ReflectionAssert;

public class AssertReflectionEqualsTest {
    @Test
    public void testReflection(){
        User user1 = new User("tom","1234");
        User user2 = new User("tom","1234");
        ReflectionAssert.assertReflectionEquals(user1, user2);
    }

}
```

ReflectionAssert. AssertReflectionEquals(期望值,实际值,比较级别)方法为我们提供了各种级别的比较断言，下面依次介绍这些级别的比较断言。

- ReflectionComparatorMode.LENIENT_ORDER：忽略要断言的集合 collection 或者 array 中元素的顺序。
- ReflectionComparatorMode.IGNORE_DEFAULTS：忽略 Java 类型的默认值。如当引用类型为 null，整型类型为 0，或者布尔类型为 false 时，断言忽略这些值的比较。
- ReflectionComparatorMode.LENIENT_DATES：比较两个实例的 Date 是不是都被设置了值或者都为 null，而忽略 Date 的值是否相等。

6．assertLenientEquals：断言

ReflectionAssert 类为我们提供了两种比较断言：一种是可忽略顺序的；另一种是可忽略默认值的断言 assertLenientEquals，使用这种断言就可以进行简单比较。下面通过实例来学习其基本用法，如代码清单 20-11 所示。

代码清单 20-11　assertLenientEquals断言测试

```
package sample.unitils;
import java.util.*;
```

```
…
public class AssertReflectionEqualsTest {
    @Test
    public void testLenientEquals(){
        Integer orderList1[] = new Integer[]{1,2,3};
        Integer orderList2[] = new Integer[]{3,2,1};

        //① 测试两个数组的值是否相等，忽略顺序
        //assertReflectionEquals(orderList1, orderList2,LENIENT_ORDER);
        assertLenientEquals(orderList1, orderList2);

        //② 测试两个对象的值是否相等，忽略默认值
        User user1 = new User("tom","1234");
        User user2 = new User("tom","1234");
        assertLenientEquals(user1, user2);

    }
}
```

7．assertPropertyXxxEquals：属性断言

assertLenientEquals 和 assertReflectionEquals 这两个方法是把对象作为整体进行比较，ReflectionAssert 类还为我们提供了只比较对象特定属性的方法：assertPropertyReflectionEquals() 和 assertPropertyLenientEquals()。下面通过实例学习其具体的用法，如代码清单 20-12 所示。

代码清单 20-12　assertPropertyXxxEquals属性断言

```
package sample.unitils;
import java.util.*;
…
public class AssertReflectionEqualsTest {
      User user = new User("tom","1234");
      assertPropertyReflectionEquals("userName", "tom", user);
      assertPropertyLenientEquals("lastVisit", null, user);
}
```

assertPropertyReflectionEquals()断言是默认严格比较模式但是可以手动设置比较级别的断言，assertPropertyLenientEquals()断言是既可忽略顺序又可忽略默认值的断言。

20.4.2　集成 Spring

Unitils 提供了一些在 Spring 框架下进行单元测试的特性。在基于 Spring 框架的项目开发过程中，编写代码要保持代码的非侵入性，也就是设计的类在非 Spring 容器中仍然易于测试。但是很多时候在 Spring 容器中进行测试还是非常有用的。

Unitils 提供了以下支持 Spring 的特性：

- ApplicationContext 配置的管理。
- 在单元测试代码中注入 Spring 的 Beans。

- 使用定义在 Spring 配置文件中的 Hibernate SessionFactory。
- 引用在 Spring 中定义的 Unitils 数据源。

ApplicationContext 配置

可以简单地在一个类、方法或者属性上标注@SpringApplicationContext 注解，并用 Spring 的配置文件作为参数来加载 Spring 应用上下文。下面通过实例来介绍一下如何创建 ApplicationContext，如代码清单 20-13 所示。

代码清单 20-13　加载Spring应用上下文

```
import org.testng.annotations.*;
import org.springframework.context.ApplicationContext;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import com.smart.service.UserService;
import static org.testng.Assert.*;

//①用户服务测试
public class UserServiceTest extends UnitilsTestNG {

    //①-1 加载Spring配置文件
    @SpringApplicationContext({"smart-service.xml", "smart-dao.xml"})
    private ApplicationContext applicationContext;

    //①-1 加载Spring容器中的Bean
    @SpringBean("userService")
    private UserService userService;

   //①-3 测试Spring容器中的用户服务Bean
   @Test
    public void testUserService (){
        assertNotNull(applicationContext);
        assertNotNull(userService.findUserByUserName("tom"));
    }
}
```

在①-1 处，通过@SpringApplicationContext 注解加载 smart-service.xml 和 smart-dao.xml 两个配置文件，生成一个 Spring 应用上下文，这样就可以在注解的范围内引用 applicationContext 这个上下文。在①-2 处，通过@SpringBean 注解注入当前 Spring 容器中相应的 Bean，如实例中加载 ID 为 userService 的 Bean 到当前测试范围。在①-3 处，通过 TestNG 断言验证是否成功加载 applicationContext 和 userService。Unitils 加载 Spring 上下文的过程是：首先扫描父类的@SpringApplicationContext 注解，如果找到了，就在加载子类的配置文件之前加载父类的配置文件，这样就可以让子类重写配置文件和加载特定的配置文件。

细心的读者可能会发现，采用这种方式加载 Spring 应用上下文，在每次执行测试时，都会重复加载 Spring 应用上下文。Unitils 提供了在类上加载 Spring 应用上下文的能力，以避免重复加载的问题，如代码清单 20-14 所示。

代码清单 20-14　通过基类加载ApplicationContext

```
…
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"})
public class BaseServiceTest extends UnitilsTestNG {

    //加载Spring应用上下文
    @SpringApplicationContext
    public ApplicationContext applicationContext;

}
```

在父类 BaseServiceTest 里指定了 Spring 配置文件，Spring 应用上下文只会创建一次，然后在子类 SimpleUserServiceTest 里会重用这个应用上下文。加载 Spring 应用上下文是一项非常繁重的操作，如果重用这个 Spring 应用上下文，就会大大提升测试的性能，如代码清单 20-15 所示。

代码清单 20-15　通过继承使用父类的ApplicationContext

```
…
public class SimpleUserServiceTest extends BaseServiceTest {

    //① Spring容器中加载ID为userService的Bean
    @SpringBean("userService")
    private UserService userService1;

    //② 从Spring容器中加载与UserService相同类型的Bean
    @SpringBeanByType
    private UserService userService2;

    //③ 从Spring容器中加载与userService相同名称的Bean
    @SpringBeanByName
    private  UserService userService;

    //④ 使用父类的Spring应用上下文
    @Test
    public void testApplicationContext(){
        assertNotNull(applicationContext);
    }

    @Test
    public void testUserService(){
        assertNotNull(userService.findUserByUserName("tom"));
        assertNotNull(userService1.findUserByUserName("tom"));
        assertNotNull(userService2.findUserByUserName("tom"));
    }
}
…
```

在①处，使用@SpringBean 注解从 Spring 容器中加载一个 ID 为 userService 的 Bean。在②处，使用@ SpringBeanByType 注解从 Spring 容器中加载一个与 UserService 相同类型的 Bean，如果找不到相同类型的 Bean，就会抛出异常。在③处，使用@SpringBeanByName 注解从 Spring 容器中加载一个与当前属性名称相同的 Bean。

20.4.3　集成 Hibernate

Hibernate 是一个优秀的 O/R 开源框架，它极大地简化了应用程序的数据访问层开发。虽然我们在使用一个优秀的 O/R 框架，但并不意味着我们无须对数据访问层进行单元测试。单元测试仍然非常重要，它不仅可以确保 Hibernate 映射类的映射正确性，也可以很便捷地测试 HQL 查询等语句。Unitils 为方便测试 Hibernate，提供了许多实用的工具类，如 HibernateUnitils，使用 assertMappingWithDatabaseConsistent()方法就可以方便地测试映射文件的正确性。

SessionFactory 配置

可以简单地在一个类、方法或者属性上标注@HibernateSessionFactory 注解，并用 Hibernate 的配置文件作为参数来加载 Hibernate 应用上下文。下面通过实例来介绍一下如何创建 SessionFactory，如代码清单 20-16 所示。

代码清单 20-16　通过基类加载SessionFactory

```
…
@HibernateSessionFactory("hibernate.cfg.xml")
public class BaseDaoTest extends UnitilsTestNG {
    @HibernateSessionFactory
    public SessionFactory sessionFactory;


    @Test
    public void testSessionFactory(){
        assertNotNull(sessionFactory);
    }
}
```

在父类 BaseDaoTest 里指定了 Hibernate 配置文件，Hibernate 应用上下文只会创建一次，然后在子类 SimpleUserDaoTest 里会重用这个应用上下文。加载 Hibernate 应用上下文是一项非常繁重的操作，如果重用这个 Hibernate 应用上下文，就会大大提升测试的性能，如代码清单 20-17 所示。

代码清单 20-17　通过继承使用父类的SessionFactory

```
…
public class SimpleUserDaoTest extends BaseDaoTest {
    private UserDao userDao;

    //① 初始化UserDao
    @BeforeClass
    public void init(){
        userDao = new WithoutSpringUserDaoImpl();
        userDao.setSessionFactory(sessionFactory); //使用父类的SessionFactory
    }

    //② Hibernate映射测试
    @Test
    public void testMappingToDatabase() {
```

```
        HibernateUnitils.assertMappingWithDatabaseConsistent();
    }

    //③ 测试UserDao
    @Test
    public void testUserDao(){
        assertNotNull(userDao);
        assertNotNull(userDao.findUserByUserName("tom"));
        assertEquals("tom", userDao.findUserByUserName("tom").getUserName());
    }
}
```

为了更好地演示如何应用 Unitils 测试基于 Hibernate 的数据访问层，在这个实例中不使用 Spring 框架。所以在执行测试时，需要先创建相应的数据访问层实例，如实例中的 userDao。其创建过程如①处所示，先手工实例化一个 UserDao，然后获取父类中创建的 SessionFactory，并设置到 UserDao 中。在②处，使用 Unitils 提供的工具类 HibernateUnitils 中的方法测试 Hibernate 映射文件。在③处，通过 TestNG 的断言验证 UserDao 相关方法，看是否与预期的结果一致。

20.4.4 集成 DbUnit

DbUnit 是一个基于 TestNG 扩展的数据库测试框架。它提供了大量的类，对数据库相关的操作进行了抽象和封装。DbUnit 通过使用用户自定义的数据集及相关操作使数据库处于一种可知的状态，从而使得测试自动化、可重复和相对独立。虽然不用 DbUnit 也可以达到这种目的，但是我们必须为此付出代价（编写大量代码、测试及维护）。既然有了这么优秀的开源框架，我们又何必再造轮子。目前其最新的版本是 2.4.8。

随着 Unitils 的出现，将 Spring、Hibernate、DbUnit 等整合在一起，使得 DAO 层的单元测试变得非常容易。Unitils 采用模块化方式来整合第三方框架，通过实现扩展模块接口 org.unitils.core.Module 来实现扩展功能。在 Unitils 中已经实现了一个 DbUnitModule，很好地整合了 DbUnit。通过这个扩展模块，就可以在 Unitils 中使用 DbUnit 强大的数据集功能，如用于准备数据的@DataSet 注解、用于验证数据的@ExpectedDataSet 注解。Unitils 集成 DbUnit 的流程图如图 20-5 所示。

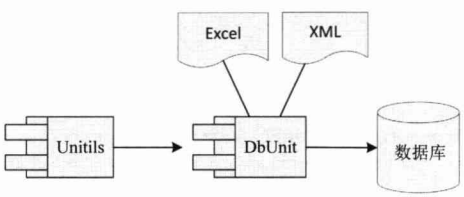


图 20-5　Unitils 集成 DbUnit 示意图

20.4.5　自定义扩展模块

Unitils 通过模块化的方式来组织各个功能模块，对外提供一个统一的扩展模块接口 org.unitils.core.Module 来实现与第三方框架的集成及自定义扩展。在 Unitils 中已经实现了目前一些主流框架的模块扩展，如 Spring、Hibernate、DbUnit、TestNG 等。如果这些内置的扩展模块无法满足需求，则可以实现自己的一些扩展模块。扩展 Unitils 模块很简单，如代码清单 20-18 所示。

代码清单 20-18　CustomExtModule

```
package sample.unitils.module;
import java.lang.reflect.Method;
import org.unitils.core.TestListener;
import org.unitils.core. Module;

//① 实现Module接口
public class CustomExtModule implements Module {

    //② 实现获取测试监听的方法
    public TestListener getTestListener() {
        return new CustomExtListener();
    }

    //② 新建监听模块
    protected class CustomExtListener extends TestListener {

       //③ 重写TestListener里的相关方法，完成相关扩展的功能
        @Override
        public void afterTestMethod(Object testObject, Method testMethod,
                Throwable testThrowable) {
           …
        }

        @Override
        public void beforeTestMethod(Object testObject, Method testMethod) {
          …
        }
    }
    …
}
```

在①处新建自定义扩展模块 CustomExtModule 实现 Module 接口；在②处新建自定义监听模块继承 TestListener；在③处重写（@Override）TestListener 里的相关方法完成相关扩展的功能。实现自定义扩展模块之后，剩下的工作就是在 Unitils 配置文件 unitils.properties 中注册这个自定义扩展的模块。

```
unitils.modules=…,custom
unitils.module. custom.className= sample.unitils.module.CustomExtModule
```

20.5 使用 Unitils 测试 DAO 层

Spring 的测试框架为我们提供了一个强大的测试环境，解决了日常单元测试中遇到的大部分测试难题，例如：运行多个测试用例和测试方法时，Spring 上下文只需创建一次；数据库现场不受破坏；方便手工指定 Spring 配置文件、手工设定 Spring 容器是否需要重新加载等。但也存在不足的地方，基本上所有的 Java 应用都涉及数据库，带数据库应用系统的测试难点在于数据库测试数据的准备、维护、验证及清理。Spring 测试框架并不能很好地解决所有问题。要解决这些问题，必须整合多方资源，如 DbUnit、Unitils、Mokito 等。其中 Unitils 正是这样的一个测试框架。

20.5.1 数据库测试的难点

按照 Kent Back 的观点，单元测试的重要特性之一应该是可重复性。不可重复的单元测试是没有价值的。因此，好的单元测试应该具备独立性和可重复性。对于业务逻辑层，可以通过 Mockito 底层对象和上层对象来获得这种独立性和可重复性。而 DAO 层因为是和数据库打交道的层，其单元测试依赖于数据库中的数据。要实现 DAO 层单元测试的可重复性，就需要对每次因单元测试引起的数据库中的数据变化进行还原，也就是保护单元测试数据库的数据现场。

20.5.2 扩展 DbUnit 用 Excel 准备数据

在测试数据访问层（DAO）时，通常需要经过测试数据的准备、维护、验证及清理过程。这个过程不仅烦锁，而且容易出错，如数据库现场容易遭受破坏、如何对数据操作正确性进行检查等。虽然 Spring 测试框架在这一方面为我们减轻了很多工作，如通过事务回滚机制来保存数据库现场等，但在测试数据及验证数据准备方面还没有一种很好的处理方式。Unitils 框架的出现改变了难测试 DAO 的局面，它将 SpringModule、DatabaseModule、DbUnitModule 等整合在一起，使得 DAO 的单元测试变得非常容易。基于 Unitils 框架的 DAO 测试过程如图 20-6 所示。

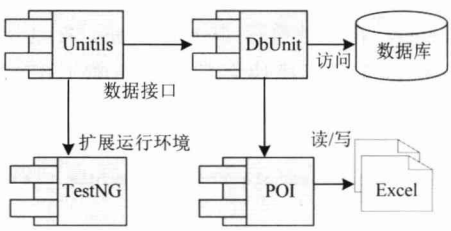


图 20-6 基于 Unitils 框架的 DAO 测试流程

以 TestNG 作为整个测试的基础框架，并采用 DbUnit 作为自动管理数据库的工具，以 XML、Excel 作为测试数据及验证数据准备，最后通过 Unitils 的数据集注解从 Excel、XML 文件中加载测试数据。使用一个注解标签就可以完成加载、删除数据操作。由于 XML 作为数据集的易用性不如 Excel，在这里就不对 XML 数据集进行讲解。下面主要讲解如何应用 Excel 作为准备及验证数据的载体，简化 DAO 单元测试。由于 Unitils 没有提供访问 Excel 的数据集工厂，因此需要编写插件支持 Excel 格式的数据源。Unitils 提供了一个访问 XML 的数据集工厂 MultiSchemaXmlDataSetFactory，其继承自 DbUnit 提供的数据集工厂接口 DataSetFactory。我们可以参考这个 XML 数据集工厂类，编写一个访问 Excel 的数据集工厂 MultiSchemaXlsDataSetFactory 及 Excel 数据集读取器 MultiSchemaXlsDataSetReader，然后在数据集读取器中调用 Apache POI 类库来读/写 Excel 文件，如代码清单 20-19 所示。

代码清单 20-19　MultiSchemaXlsDataSetFactory.java Excel数据集工厂

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
…
public class MultiSchemaXlsDataSetFactory implements DataSetFactory {
    protected String defaultSchemaName;

    //① 初始化数据集工厂
    public void init(Properties configuration, String defaultSchemaName) {
        this.defaultSchemaName = defaultSchemaName;
    }

    //② 从Excel文件中创建数据集
    public MultiSchemaDataSet createDataSet(File... dataSetFiles) {
        try {
            MultiSchemaXlsDataSetReader xlsDataSetReader =
                    new MultiSchemaXlsDataSetReader(defaultSchemaName);
            return xlsDataSetReader.readDataSetXls(dataSetFiles);
        } catch (Exception e) {
            throw new UnitilsException("创建数据集失败: "
                    + Arrays.toString(dataSetFiles), e);
        }
    }

    //③ 获取数据集文件的扩展名
    public String getDataSetFileExtension() {
        return "xls";
    }
}
…
```

与 XML 的数据集工厂 MultiSchemaXmlDataSetFactory 一样，Excel 的数据集工厂也需要实现数据集工厂接口 DataSetFactory 的 3 个方法：init(...)、createDataSet(File... dataSetFiles)和 getDataSetFileExtension()。在①处，初始化数据集工厂，需要设置一个默认的数据库表模式名称 defaultSchemaName。在②处，执行创建多数据集，具体读取构

建数据集的过程封装在 Excel 读取器 MultiSchemaXlsDataSetReader 中。在③处，获取数据集文件的扩展名，对 Excel 文件而言就是“xls”。下面来看一下这个数据集读取器的实现代码，如代码清单 20-20 所示。

代码清单 20-20　MultiSchemaXlsDataSetReader.java Excel数据集读取器

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
…
// Excel 数据集读取器
public class MultiSchemaXlsDataSetReader {
    private String defaultSchemaName;

    public MultiSchemaXlsDataSetReader(String defaultSchemaName) {
        this.defaultSchemaName = defaultSchemaName;
    }
    // Excel 数据集读取器
    public MultiSchemaDataSet readDataSetXls(File... dataSetFiles) {
        try {
            Map<String, List<ITable>> tableMap = getTables(dataSetFiles);
            MultiSchemaDataSet dataSets = new MultiSchemaDataSet();
            for (Entry<String, List<ITable>> entry : tableMap.entrySet()) {
                List<ITable> tables = entry.getValue();
                try {
                    DefaultDataSet ds = new DefaultDataSet(tables
                            .toArray(new ITable[] {}));
                    dataSets.setDataSetForSchema(entry.getKey(), ds);
                } catch (AmbiguousTableNameException e) {
                    throw new UnitilsException("构造DataSet失败!",  e);
                }
            }
            return dataSets;
        } catch (Exception e) {
            throw new UnitilsException("解析Excel文件出错: ", e);
        }
    }
    …
}
…
```

根据传入的多个 Excel 文件构造一个多数据集。其中，一个数据集对应一个 Excel 文件，一个 Excel 的 Sheet 表对应一个数据库 Table。通过 DbUnit 提供的 Excel 数据集构造类 XlsDataSet，可以很容易地将一个 Excel 文件转换为一个数据集：XlsDataSet(new FileInputStream(xlsFile))。最后将得到的多个 DataSet 用 MultiSchemaDataSet 进行封装。

下面就以一个用户 DAO 的实现类 WithoutSpringUserDaoImpl 为例，介绍如何使用我们实现的 Excel 数据集工厂。为了让 Unitils 使用自定义的数据集工厂，需要在 unitils.properties 配置文件中指定自定义的数据集工厂，如代码清单 20-21 所示。

代码清单 20-21　unitils.properties配置文件

```
…
DbUnitModule.DataSet.factory.default=sample.unitils.dataset.excel.
```

```
MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=sample.unitils.dataset.
                                                excel.MultiSchemaXlsData
                                                SetFactory
```

其中，DbUnitModule.DataSet.factory.default 是配置数据集工厂类，在测试方法中可以使用@DataSet 注解加载指定的准备数据。默认是 XML 的数据集工厂，这里指定自定义数据集工厂全限定类名为 sample.unitils.dataset.excel.MultiSchemaXlsDataSetFactory。

DbUnitModule. ExpectedDataSet.factory.default 是配置验证数据集工厂类，也是指定自定义数据集工厂类，使用@ExpectedDataSet 注解加载验证数据，如代码清单 20-22 所示。

代码清单 20-22　UserDaoTest.java 用户DAO测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
…
public class UserDaoTest extends UnitilsTestNG {
    @Test
    @DataSet //① 准备测试数据
    public void getUser() {
        …
    }

    @Test
    @DataSet("Smart.SaveUser.xls") //② 准备测试数据
    @ExpectedDataSet //③ 准备验证数据
    public void saveUser()throws Exception  {
        …
    }

}
…
```

@DateSet 注解表示测试时需要寻找 DbUnit 的数据集文件进行加载。如果没有指明数据集的文件名，则 Unitils 自动在当前测试用例所在的类路径下加载文件名为测试用例类名的数据集文件，如实例中的①处，将到 UserDaoTest.class 所在目录加载 WithExcelUserDaoTest.xls 数据集文件。

@ExpectedDataSet 注解用于加载验证数据集文件。如果没有指明数据集的文件名，则会在当前测试用例所在的类路径下加载文件名为 testClassName.methodName-result.xls 的数据集文件。如实例中的③处将加载 UserDaoTest. saveUser.result.xls 数据集文件。

20.5.3　测试实战

使用 TestNG 作为基础测试框架，结合 Unitils、DbUnit 管理测试数据，并使用上文编写的 Excel 数据集工厂（见代码清单 20-19）。从 Excel 数据集文件中获取准备数据及验证数据，并使用 MySQL 作为测试数据库。下面详细介绍如何应用 Excel 准备数据集及验证数据集来测试 DAO。

在进行 DAO 层的测试之前，先来认识一下需要测试的 UserDaoImpl 用户数据访问类。UserDaoImpl 用户数据访问类中拥有一个获取用户信息和保存注册用户信息的方法，其代码如代码清单 20-23 所示。

代码清单 20-23　UserDaoImpl

```
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate4.HibernateTemplate;
import com.smart.dao.UserDao;
import com.smart.domain.User;
public class UserDaoImpl implements UserDao {

    //通过用户名获取用户信息
    public User findUserByUserName(String userName) {
          String hql = " from User u where u.userName=?";
          List<User> users = getHibernateTemplate().find(hql, userName);
          if (users != null && users.size() > 0)
              return users.get(0);
          else
              return null;
     }

    //保存用户信息
     public void save(User user) {
          getHibernateTemplate().saveOrUpdate(user);
     }
     …
}
```

在认识了需要测试的 UserDaoImpl 用户数据访问类之后，还需要认识一下用于表示用户领域的对象 User，在演示测试保存用户信息及获取用户信息时需要用到此领域对象，其代码如代码清单 20-24 所示。

代码清单 20-24　User

```
import javax.persistence.Column;
import javax.persistence.Entity;
…
@Entity
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Table(name = "t_user")
public class User implements Serializable{
    @Id
    @Column(name = "user_id")
    protected int userId;

    @Column(name = "user_name")
    protected String userName;

    protected String password;

    @Column(name = "last_visit")
    protected Date lastVisit;
```

```
    @Column(name = "last_ip")
    protected String lastIp;

    @Column(name = "credit")
    private int credit;
    …
}
```

用户登录日志领域对象 LoginLog 与用户领域对象 Hibernate 的注解配置一致，这里就不再列出，读者可以参考本书配套网盘中的实例代码。在实例测试中，直接使用 Hibernate 进行持久化操作，所以还需要对 Hibernate 进行相应配置，详细的配置如代码清单 20-25 所示。

代码清单 20-25　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>
        <!--① SQL方言，这里设定的是MySQL -->
        <property name="dialect"> org.hibernate.dialect.MySQL5Dialect</property>
        <!--② 数据库连接配置 -->
        <property name="hibernate.connection.driver_class">
          com.mysql.jdbc.Driver
       </property>
        <property name="hibernate.connection.url">
           jdbc:mysql://localhost:3306/sampledb?useUnicode=true&characterEncoding=UTF-8
       </property>
        <!--设置连接数据库的用户名-->
        <property name="hibernate.connection.username">root</property>
        <!--设置连接数据库的密码-->
        <property name="hibernate.connection.password">root</property>
        <!--③ 设置显示SQL语句方便调试-->
        <property name="hibernate.show_sql">true</property>
        <!--④ 配置映射 -->
        <property name="configurationClass">
           org.hibernate.cfg.Configuration
       </property>
        <mapping class="com.smart.domain.User" />
        <mapping class="com.smart.domain.LoginLog" />
    </session-factory>
</hibernate-configuration>
```

选用 MySQL 作为测试数据库，在①处，配置 MySQL 的 SQL 方言 MySQL5Dialect。在②处，对连接数据库驱动及数据库连接进行相应的配置。为了方便测试调试，在③处设置显示 Hibernate 生成的 SQL 语句。在④处启用 Hibernate 的注解功能，并配置相应的领域对象，如实例中的 User 和 LoginLog。将配置好的 hibernate.cfg.xml 放在 src 目录下。

1. 配置 Unitils 测试环境

要在单元测试中使用 Unitils，首先需要在测试目录 src\test\resources 中创建一个

项目级 unitils.properties 配置文件。实例中 unitils.properties 文件的详细配置如代码清单 20-26 所示。

代码清单 20-26　unitils.properties

```
#① 启用Unitils所需模块
unitils.modules=database,dbunit,hibernate,spring

#自定义扩展模块，并解决DbUnit对MySQL处理的Bug
unitils.module.dbunit.className=org.dbunit.MySqlDbUnitModule

#② 配置MySQL数据库连接
database.driverClassName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/sampledb?useUnicode=true&characterEncoding=UTF-8
database.dialect =mysql
database.userName=root
database.password=123456
database.schemaNames=sampledb

#③ 配置数据库维护策略
updateDataBaseSchema.enabled=true

#④ 配置数据库表创建策略
dbMaintainer.autoCreateExecutedScriptsTable=true
dbMaintainer.script.locations= D:/masterSpring/code/chapter20/schema

#⑤ 数据集加载策略
#CleanInsertLoadStrategy:先删除dateSet中有关表的数据，然后再插入数据
#InsertLoadStrategy: 只插入数据
#RefreshLoadStrategy:有同样key的数据更新，没有的插入
#UpdateLoadStrategy:有同样key的数据更新，没有的不作任何操作
#DbUnitModule.DataSet.loadStrategy.default=org.unitils.dbunit.datasetloadstrategy.
InsertLoadStrategy


#⑥ 配置数据集工厂
DbUnitModule.DataSet.factory.default=sample.unitils.dataset.excel.
MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=sample.unitils.dataset.excel.
MultiSchemaXlsDataSetFactory

#⑦ 配置事务策略
#commit是单元测试方法过后提交事务
#rollback是回滚事务
#disabled是没有事务，默认情况下，事务管理是disabled
DatabaseModule.Transactional.value.default=disabled

#⑧ 配置数据集结构模式XSD的生成路径
dataSetStructureGenerator.xsd.dirName=resources/xsd
```

我们知道，unitils.properties 文件中配置的属性是整个项目级别的，整个项目都可以使用这些全局的属性配置。特定用户使用的属性可以设置在 unitils-local.properties 文件中，如 user、password 和 schema，这样每个开发者就可以使用自定义的测试数据库的 schema，而且彼此之间不会产生影响，实例的详细配置如代码清单 20-27 所示。

代码清单 20-27　unitils-local.properties

```
...
database.userName=root
database.password=123456
database.schemaNames=sampledb
...
```

如果用户分别在 unitils.properties 和 unitils-local.properties 文件中对相同属性配置不同值，则将会以 unitils-local.properties 配置内容为主。如在 unitils.properties 配置文件中也配置了 database.schemaNames=xxx，则测试时启用的是用户自定义配置中的值 database.schemaNames=sampledb。

2．配置数据集加载策略

默认的数据集加载机制采用先清理后插入的策略，也就是数据在被写入数据库的时候会先删除数据集中有对应表的数据，然后将数据集中的数据写入数据库。这个加载策略是可配置的，可以通过修改 DbUnitModule.DataSet.loadStrategy.default 的属性值来改变加载策略。如代码清单 20-26 中⑤处的配置策略，这时加载策略就由先清理后插入变成了插入，数据已经存在于表中将不会被删除，测试数据只进行插入操作。可选的加载策略列表如下。

- CleanInsertLoadStrategy：先删除 dateSet 中有关表的数据，然后再插入数据。
- InsertLoadStrategy：只插入数据。
- RefreshLoadStrategy：有同样 key 的数据更新，没有的插入。
- UpdateLoadStrategy：有同样 key 的数据更新，没有的不作任何操作。

3．配置事务策略

在测试 DAO 的时候都会填写一些测试数据，每个测试在运行时都会修改或者更新数据，当下一个测试运行的时候，都需要将数据恢复到原有状态。如果使用的是 Hibernate 或者 JPA，则需要每个测试都运行在事务中，以保证系统的正常工作。在默认情况下，事务管理是 disabled 的，可以通过修改 DatabaseModule.Transactional.value.default 配置选项，如代码清单 20-26 中⑧处的配置策略，这时每个测试都将执行 commit 操作。其他可选的配置属性值有 rollback 和 disabled。

4．准备测试数据库及测试数据

配置好 Unitils 基本配置、加载模块、数据集创建策略、事务策略之后，我们就着手开始测试数据库及测试数据准备工作，首先创建测试数据库。

5．创建测试数据库

在本章节模块 D:\masterSpring\code\chapter20\目录下创建一个 dbscripts 文件夹，且这个文件夹必须与在 unitils.properties 文件中 dbMaintainer.script.locations 配置项指定的位置一致，如代码清单 20-26 中的④处所示。

在这个文件夹中创建一个数据库创建脚本文件 001_sampledb.sql，里面包含创建用户表 t_user 及登录日志表 t_login_log，详细的脚本如代码清单 20-28 所示。

代码清单 20-28　001_create_sampledb.sql

```
CREATE TABLE t_user (
            user_id INT generated by default as identity (start with 100),
            user_name VARCHAR(30),credit INT,
            credit INT,
            password  VARCHAR(32),last_visit timestamp,
            last_ip  VARCHAR(23), primary key (user_id));

CREATE TABLE t_login_log (
            login_log_id  INT generated by default as identity (start with 1),
            user_id   INT,
            ip  VARCHAR(23),
            login_datetime timestamp,
            primary key (login_log_id));
```

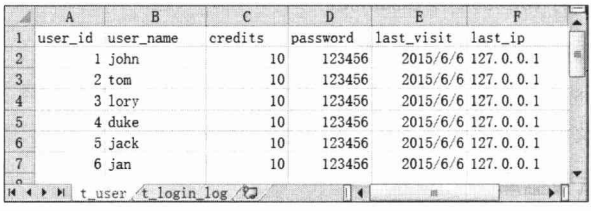
细心的读者可能会发现这个数据库创建脚本文件名好像存在一定的规则，是的，这个数据库创建脚本文件命名需要按以下规则命名：版本号+“_”+“自定义名称”+“.sql”。

6．连接到测试数据库

在测试 DAO 时，读者要有个疑问：测试数据库用到的数据源来自哪里？怎么让我们测试的 DAO 类来使用我们的数据源？在执行测试实例的时候，Unitils 会根据我们定义的数据库连接属性来创建一个数据源实例连接到测试数据库。随后的 DAO 测试会重用相同的数据源实例。建立连接的细节定义在 unitils.properties 配置文件中，如代码清单 20-26 中的②处所示。

7．用 Excel 准备测试数据

准备好测试数据库之后，剩下的工作就是用 Excel 来准备测试数据及验证数据。回顾一下我们要测试的 UserDaoImpl 类（见代码清单 20-23），需要对其中的获取用户信息方法 findUserByUserName()及保存用户信息方法 saveUser()进行测试，所以至少需要准备 3 个 Excel 数据集文件，分别是供查询用户用的数据集文件 UserDao.Users.xls、供保存用户信息用的数据集文件 UserDao.SaveUser.xls 及供保存用户信息用的验证数据集文件 UserDao.ExpectedSaveUser.xls。下面以用户数据集文件 UserDao.Users.xls 为例来进行说明，如图 20-7 所示。



	A	B	C	D	E	F
1	user_id	user_name	credits	password	last_visit	last_ip
2	1	john	10	123456	2015/6/6	127.0.0.1
3	2	tom	10	123456	2015/6/6	127.0.0.1
4	3	lory	10	123456	2015/6/6	127.0.0.1
5	4	duke	10	123456	2015/6/6	127.0.0.1
6	5	jack	10	123456	2015/6/6	127.0.0.1
7	6	jan	10	123456	2015/6/6	127.0.0.1

t_user　t_login_log

图 20-7　UserDao.Users.xls 查询用户数据集

其中，Excel 表格底部 Sheet 页 t_user 表示数据库对应的表名称。表格第一行表示数据库中 t_user 表对应的字段名称，从表格第二行开始表示测试的模拟数据。一个数据集文件可以对应多张表，一个 Sheet 就对应一张表。把创建好的数据集文件放到与测试类

相同的目录中，如实例中的 UserDaoTest 类位于 com.smart.dao 包中，则数据集文件需要放到当前包中。

8. 编写 UserDaoImpl 的测试用例

完成了 Unitils 环境配置、准备好测试数据库及测试数据之后，就可以开始编写用户 DAO 单元测试类了。下面为用户数据访问 UserDaoImpl 编写测试用例类，如代码清单 20-29 所示。

代码清单 20-29　UserDaoTest用户DAO测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
…
@SpringApplicationContext( {"smart-dao.xml" }) //① 初始化Spring容器
public class UserDaoTest extends UnitilsTestNG {

    @SpringBean("jdbcUserDao")  //② 从Spring容器中加载DAO
    private UserDao userDao;

    @BeforeClass
    public void init() {

    }
    …
}
```

在①处，通过 Unitils 提供的@SpringApplicationContext 注解加载 Spring 配置文件，并初始化 Spring 容器。在②处，通过@SpringBean 注解从 Spring 容器加载一个用户 DAO 实例。编写好 UserDaoTest 测试基础模型之后，接下来就开始编写查询用户信息的测试方法 findUserByUserName()，如代码清单 20-30 所示。

代码清单 20-30　UserDaoTest.findUserByUserName()测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
…
public class UserDaoTest extends UnitilsTestNG {
    …

    @Test //① 标记为测试方法
    @DataSet("Smart.Users.xls") //② 加载准备用户测试数据
    public void findUserByUserName() {
        User user = userDao.findUserByUserName("tony"); //③ 从数据库中加载tony用户
        assertNull("不存在用户名为tony的用户!", user);
        user = userDao.findUserByUserName("jan"); //④ 从数据库中加载jan用户
        assertNotNull("jan用户存在! ", user);
        assertEquals("jan", user.getUserName());
       assertEquals("123456",user.getPassword());
        assertEquals(10,user.getCredit());
    }
    …
}
```

在①处，通过 TestNG 提供的@Test 注解把当前方法标记为测试方法。在②处，通过 Unitils 提供的@DataSet 注解从当前测试类 UserDaoTest.class 所在的目录寻找支持 DbUnit 的数据集文件并进行加载。在执行测试逻辑之前，会把加载的数据集先持久化到测试数据库中（具体加载数据集的策略详见上文“配置数据集加载策略”部分）。实例中采用的是默认加载策略，即先删除测试数据库对应表的数据，再插入数据集中的测试数据。这种策略可以避免不同测试方法加载数据集时的相互干扰。在③处执行查询用户方法时，测试数据库中 t_user 表的数据与图 20-8 中 Smart.Users.xls 的数据一致，因此查询不到“tony”用户信息。在④处，执行查询“jan”用户信息，从测试数据集中可以看出，可以加载到“jan”的详细信息。最后在 IDE 中执行 UserDaoTest.findUserByUserName() 测试方法，按我们的预期通过测试，测试结果如图 20-8 所示。

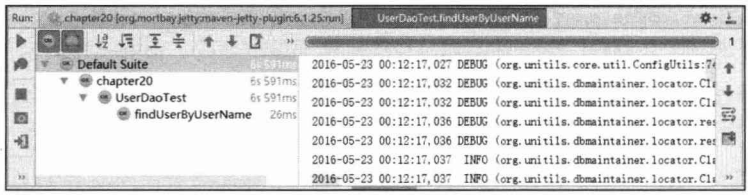


图 20-8　UserDaoTest. findUserByUserName()测试结果

完成了查询用户的测试之后，我们开始着手编写保存用户信息的测试方法，详细的实现代码如代码清单 20-31 所示。

代码清单 20-31　UserDaoTest.saveUser()测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
…
public class UserDaoTest extends UnitilsTestNG {
    …

    @Test  //① 标记为测试方法
    @ExpectedDataSet("UserDao.ExpectedSaveUser.xls") //准备验证数据
    public void saveUser()throws Exception  {
        User u = new User();
        u.setUserId(1);
        u.setUserName("tom");
        u.setPassword("123456");
        u.setLastVisit(getDate("2016-03-06 08:00:00","yyyy-MM-dd HH:mm:ss"));
        u.setCredit(30);
        u.setLastIp("127.0.0.1");
        userDao.save(u);  //执行用户信息更新操作
    }
    …
}
```

在①处，通过 TestNG 提供的@Test 注解把当前方法标记为测试方法。在②处，通过 Unitils 提供的@ExpectedDataSet 注解从当前测试类 UserDaoTest.class 所在的目录寻找支持 DbUnit 的验证数据集文件并进行加载，之后验证数据集里的数据和数据库中的数

据是否一致。在 UserDaoTest.saveUser()测试方法中创建一个 User 实例，并设置验证数据集数据，然后执行保存用户操作。最后在 IDE 中执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-9 所示。

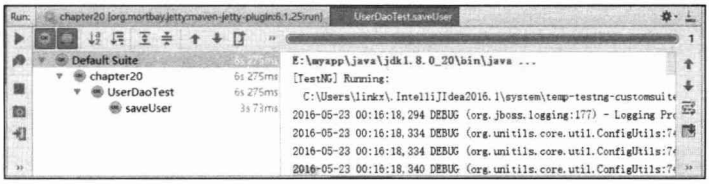


图 20-9　UserDaoTest. saveUser()测试结果（1）

虽然已经成功地完成了保存用户信息 UserDaoTest.saveUser() 方法测试，但还是存在不足的地方。我们的测试数据通过硬编码方式直接设置在 User 实例中，如果需要更改测试数据，则只能更改测试代码，大大削减了测试的灵活性。如果能直接从 Excel 数据集中获取测试数据，并自动绑定到目标对象，那我们的测试用例就更加完美了。为此，笔者编写了一个获取 Excel 数据集的 Bean 工厂 XlsDataSetBeanFactory，用于自动绑定数据集到测试对象。我们对上面的测试方法进行整改，实现代码如代码清单 20-32 所示。

代码清单 20-32　UserDaoTest.java

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
import sample.unitils.dataset.util.XlsDataSetBeanFactory;
…
public class UserDaoTest extends UnitilsTestNG {
    …

    @Test  //① 标记为测试方法
    @ExpectedDataSet("UserDao.ExpectedSaveUser.xls") //准备验证数据
    public void saveUser()throws Exception  {

        userDao.clean();//清理测试数据

        //② 从保存数据集中创建Bean
         User u  = XlsDataSetBeanFactory.createBean("UserDao.SaveUser.xls"
                                                    ,"t_user", User.class);
         userDao.save(u); //③ 执行用户信息更新操作
    }
   …
}
```

在②处，通过 XlsDataSetBeanFactory.createBean()方法从当前测试类所在目录加载 UserDao.SaveUser.xls 数据集文件。把 UserDao.SaveUser.xls 中名为 t_user 的 Sheet 页中的数据绑定到 User 对象，如果当前 Sheet 页有多条记录，则可以通过 XlsDataSetBean Factory.createBeans()方法获取用户列表 List<User>。最后在 IDE 中重新执行 UserDaoTest. saveUser()测试方法，执行结果如图 20-10 所示。

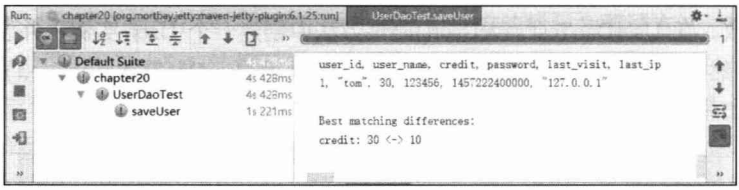


图 20-10　UserDaoTest. saveUser()测试结果（2）

从测试结果可以看出，执行 UserDaoTest.saveUser()测试失败。从右边的失败报告信息可以看出，是由于模拟用户的积分与我们期望的数据不一致造成的，期望用户积分是 30，而我们保存用户的积分是 10。重新对比一下 UserDao.SaveUser.xls 数据集的数据与 UserDao.ExpectedSaveUser.xls 数据集的数据，确实我们准备保存数据集的数据与验证结果的数据不一致。把 UserDao.SaveUser.xls 数据集中的用户积分更改为 30，最后在 IDE 中重新执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-11 所示。

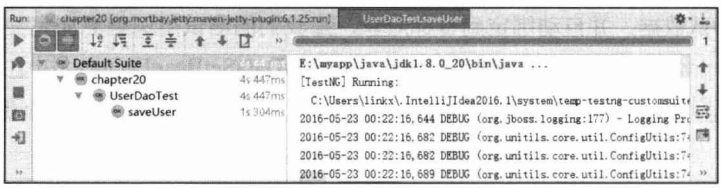


图 20-11　UserDaoTest. saveUser()测试结果（3）

从测试结果可以看出，保存用户通过测试。从上述的测试实战我们已经体验到用 Excel 准备测试数据与验证数据带来的便捷性。至此，我们完成了 DAO 测试的整个过程。对于 XlsDataSetBeanFactory 的具体实现，读者可以查看本章的实例源码，这里就不再详细分析了。代码清单 20-33 是实现的基本骨架。

代码清单 20-33　XlsDataSetBeanFactory

```
import org.dbunit.dataset.Column;
import org.dbunit.dataset.DataSetException;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.ITable;
import org.dbunit.dataset.excel.XlsDataSet;
…
public class XlsDataSetBeanFactory {

    //从Excel数据集文件中创建多个Bean
    public static <T> List<T> createBeans(String file, String tableName,
            Class<T> clazz) throws Exception {
        BeanUtilsBean beanUtils = createBeanUtils();
        List<Map<String, Object>> propsList = createProps(file, tableName);
        List<T> beans = new ArrayList<T>();
        for (Map<String, Object> props : propsList) {
            T bean = clazz.newInstance();
            beanUtils.populate(bean, props);
            beans.add(bean);
        }
        return beans;
```

```
    }

    //从Excel数据集文件中创建多个Bean
    public static <T> T createBean(String file, String tableName, Class<T> clazz)
            throws Exception {
        BeanUtilsBean beanUtils = createBeanUtils();
        List<Map<String, Object>> propsList = createProps(file, tableName);
        T bean = clazz.newInstance();
        beanUtils.populate(bean, propsList.get(0));
        return bean;
    }
    …
}
```

20.6　使用 Unitils 测试 Service 层

在进行服务层的测试之前，先来认识一下需要测试的 UserServiceImpl 服务类。UserServiceImpl 服务类中有一个处理用户登录的服务方法，其代码如代码清单 20-34 所示。

代码清单 20-34　UserService.java

```
package com.smart.service;
import com.smart.domain.LoginLog;
import com.smart.domain.User;
import com.smart.dao.UserDao;
import com.smart.dao.LoginLogDao;
@Service("userService")
public class UserServiceImpl implements UserService {
    private UserDao userDao;
    private LoginLogDao loginLogDao;

    public void loginSuccess(User user) {
        user.setCredit( 5 + user.getCredit());
        LoginLog loginLog = new LoginLog();
        loginLog.setUserId(user.getUserId());
        loginLog.setIp(user.getLastIp());
        loginLog.setLoginTime(user.getLastVisit());
       userDao.updateLoginInfo(user);
       loginLogDao.insertLoginLog(loginLog);
    }
  …
}
```

UserServiceImpl 需要调用 DAO 层的 UserDao 和 LoginLogDao 以及 User 和 LoginLog 这两个 PO 完成业务逻辑，User 和 LoginLog 分别对应 t_user 和 t_login_log 这两张数据库表。

在用户登录成功后，调用 UserServiceImpl 中的 loginSuccess()方法执行用户登录成功后的业务逻辑。

① 登录用户添加 5 个积分（t_user.credit）。

② 将登录用户的最后访问时间（t_user.last_visit）和 IP（t_user.last_ip）更新为当前值。

③ 在日志表（t_login_log）中为用户添加一条登录日志。

这是一个需要访问数据库并存在数据更改操作的业务方法，它工作在事务环境下。下面是装配该服务类 Bean 的 Spring 配置文件，如代码清单 20-35 所示。

代码清单 20-35　smart-service.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
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    <context:component-scan base-package="com.smart.service"/>
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource" />
    <tx:annotation-driven />
    <!-- 使用aop/tx命名空间配置事务管理，这里对service包下的服务类方法提供事务-->
    <aop:config>
        <aop:pointcut id="jdbcServiceMethod"
            expression= "within(com.smart.service..*)" />
        <aop:advisor pointcut-ref="jdbcServiceMethod" advice-ref="jdbcTxAdvice" />
    </aop:config>
    <tx:advice id="jdbcTxAdvice" transaction-manager="transactionManager">
        <tx:attributes>
          <tx:method name="*"/>
      </tx:attributes>
    </tx:advice>
</beans>
```

UserServiceImpl 所关联的 DAO 类和 PO 类都比较简单，这里就不一一列出了，读者可以参考本文配套网盘中的实例代码。

下面为 UserServiceImpl 编写一个简单的测试用例类，此时的目标是让这个基于 Unitils 测试框架的测试类运行起来，并联合 Mockito 框架创建 Dao 模拟对象。首先编写测试 UserService#findUserByUserName()方法的测试用例，如代码清单 20-36 所示。

代码清单 20-36　UserServiceTest.java

```
package com.smart.service;

import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.springframework.test.util.ReflectionTestUtils;
import org.unitils.spring.annotation.SpringBean;
import org.testng.annotations.*;
import com.smart.domain.User;
import java.util.Date;
…
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsTestNG{

    private UserDao userDao; //② 声明用户Dao
    private LoginLogDao loginLogDao;

    @BeforeClass //③ 创建Dao模拟对象
    public void init(){
    userDao = mock(UserDao.class);
    loginLogDao = mock(LoginLogDao.class);
    }

    @Test  //④ 设置为TestNG测试方法
    public void findUserByUserName() {

        //④-1 模拟测试数据
        User user = new User();
        user.setUserName("tom");
        user.setPassword("1234");
        user.setCredit(100);
        doReturn(user).when(userDao).findUserByUserName("tom");

        //④-2 实例化用户服务实例类
        UserServiceImpl userService = new UserServiceImpl();

        //④-3 通过Spring测试框架提供的工具类为目标对象私有属性赋值
        ReflectionTestUtils.setField(userService, "userDao", userDao);

        //④-4 验证服务方法
        User u = userService.findUserByUserName("tom");
        assertNotNull(u);
        assertThat(u.getUserName(),equalTo(user.getUserName()));

        //④-5 验证交互行为
        verify(userDao,times(1)).findUserByUserName("tom");
    }

}
```

在这里，我们让 UserServiceTest 直接继承于 Unitils 提供的 UnitilsTestNG 的抽象测试类，该抽象测试类的作用是让 Unitils 测试框架可以在 TestNG 测试框架的基础上运行起来。在①处标注了一个类级的@SpringApplicationContext 注解，这里 Unitils 将从类路

径中加载 Spring 配置文件，并使用该配置文件启动 Spring 容器。在③处通过 Mockito 创建两个模拟 DAO 实例。在④-1 处模拟测试数据并通过 Mockito 录制 UserDao#findUserByUserName()行为。在④-2 处实例化用户服务实例类，并在④-3 处通过 Spring 测试框架提供的工具类 org.springframework.test.util.ReflectionTestUtils 为 userService 私有属性 userDao 赋值（ReflectionTestUtils 是一个访问测试对象中私有属性非常好用的工具类）。在④-4 处调用服务 UserService#findUserByUserName()方法，并验证返回结果。在④-5 处通过 Mockito 验证模拟 userDao 对象是否被调用，且只调用一次。最后在 IDE 中执行 UserServiceTest 测试用例，测试结果如图 20-12 所示。

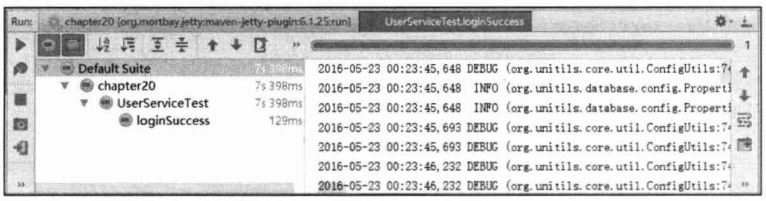


图 20-12　UserServceTest. findUserByUserName()测试结果

从运行结果可以看出，我们已经成功地对 UserServceTest.findUserByUserName()执行了单元测试。下面通过 Unitils 提供的@DataSet 注解来准备测试数据，并测试 UserService#loginSuccess()方法。UserDao.SaveUsers.xls 数据集如图 20-13 所示。

	A	B	C	D	E	F
1	user_id	user_name	credits	password	last_visi	last_ip
2	1	john	10	123456	2015/6/6	127.0.0.1
3	2	tom	10	123456	2015/6/7	127.0.0.1
4	3	lory	10	123456	2015/6/8	127.0.0.1

t_user / t_login_log /

图 20-13　Smart.SaveUsers 数据集

准备好测试数据集之后，就可以开始为 UserServiceImpl 编写测试用例类，此时的目标是通过 Unitils 提供的@DataSet 注解准备测试数据，以此来保证测试数据的独立性，避免手工通过事务回滚维护测试数据的状态。测试 UserService#loginSuccess()方法的代码如代码清单 20-37 所示。

代码清单 20-37　UserServiceTest.java

```
package com.smart.service;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import org.testng.annotations.*;
import com.smart.domain.User;
import java.util.Date;
…
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsTestNG{

    //② 从Spring容器中加载UserService实例
    @SpringBean("userService")
```

```
    private UserService userService;

    @Test
    @DataSet("Smart.SaveUsers.xls")//③ 准备验证数据
    public void loginSuccess() {
        User user = userService.findUserByUserName("tom"); //④-1 加载tom用户信息
        Date now = new Date();
        user.setLastVisit(now); //④-2 设置当前登录时间
        userService.loginSuccess(user); //④-3 user登录成功，更新其积分及添加日志
        User u = userService.findUserByUserName("tom");
        assertThat(u.getCredit(),is(105)); //⑤ 验证登录成功后用户积分是否是105分
    }
}
```

在①处通过加载 Unitils 的@SpringApplicationContext 注解加载 Spring 配置文件，并初始化 Spring 容器。在②处通过@SpringBean 注解从 Spring 容器中获取 UserService 实例。在③处通过@DataSet 注解从当前测试用例所在的类路径中加载 Smart.SaveUsers.xls 数据集，并将数据集中的数据保存到测试数据库相应的表中。从上面的数据集中可以看出，我们为 t_user 表准备了两条用户信息测试数据。在④-1 处从测试数据库中获取“tom”用户信息，模拟当前登录的用户。在④-2 处设置当前“tom”用户的登录时间。在④-3 处调用 UserService#loginSuccess()方法，更新“tom”用户积分，并持久化到测试数据库中。在⑤处，验证“tom”用户当前积分是否是 105 分。至此，完成测试用例的编写，最后在 IDE 中执行 UserServiceTest 测试用例，测试结果如图 20-14 所示。

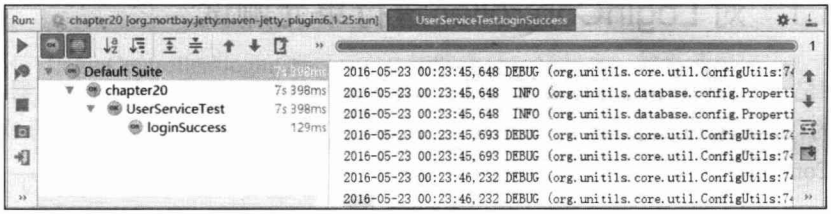


图 20-14　UserServiceTest. loginSuccess()测试结果

从运行结果可以看出，我们已经成功地对 UserServce#loginSuccess()执行了单元测试。重复执行当前单元测试，测试结果仍然通过。细心的读者可能会有疑问：没有 UserServce#loginSuccess()测试方法实施事务回滚，执行多次之后，“tom”用户的积分不应该是 105 分，那为何测试还是通过呢？这是因为 Unitils 帮我们维护测试数据库中的数据状态。Unitils 这个强大的魔力归根于 Unitils 强大的数据集更新策略。至此，我们成功地完成了 UserService 单元测试。从上面为用户服务 UserService 编写两个测试方法可以看出，对 Service 层的测试，既可以采用 TestNG+Unitils+Mockito 组合，运用 Mockito 强大的模块能力，完成 Service 层的独立性测试；也可以采用 TestNG+Unitils+DbUnit 组合，运用 DbUnit 强大的数据库维护能力，完成 Service 层+DAO 层的集成测试。

20.7 测试 Web 层

Spring 在 org.springframework.mock 包中为一些依赖于容器的接口提供了模拟类，这样用户就可以在不启动容器的情况下执行单元测试，提高单元测试的运行效率。Spring 提供的模拟类分属于以下 3 个包中。

- org.springframework.mock.jndi：为 JNDI SPI 接口提供的模拟类，以便可以脱离 Java EE 容器进行测试。
- org.springframework.mock.web：为 Servlet API 接口（如 HttpServletRequest、ServletContext 等）提供的模拟类，以便可以脱离 Servlet 容器进行测试。
- org.springframework.mock.web.portlet：为 Portlet API 接口（如 PortletRequest、PortalContext 等）提供的模拟类，以便可以脱离 Portlet 容器进行测试。

Spring 将这些模拟类单独打包成 spring-mock.jar，读者可以在 Spring 发布包的 dist 目录下找到这个类包。

在正常情况下，我们无法获得那些依赖于容器创建的接口实例，如 HttpServletRequest、ServletContext 等。模拟类实现了这些接口，它在一定程度上模拟了这些接口的输入/输出功能，借助这些模拟类的支持，我们就可以轻松地测试那些依赖于容器的功能模块。

20.7.1 对 LoginController 进行单元测试

回忆一下我们在第 2 章中编写的用于处理用户登录的 LoginController 控制器，处理请求的 handle()方法依赖于 Servlet API 接口。为了方便阅读，我们再次列出 LoginController 的代码（对原代码进行了细微调整），如代码清单 20-38 所示。

代码清单 20-38 LoginController：用户登录控制器

```
//①标注为一个Spring MVC的Controller
@Controller
public class LoginController{

    private UserService userService;

    //② 负责处理/index.html的请求
    @RequestMapping(value = "/index.html")
    public String loginPage(){
        return "login";
    }

    //③ 负责处理/loginCheck.html的请求
    @RequestMapping(value = "/loginCheck.html")
    public ModelAndView loginCheck(HttpServletRequest request,LoginCommand
loginCommand){
        boolean isValidUser =
                userService.hasMatchUser(loginCommand.getUserName(),
                                         loginCommand.getPassword());
```

```
        if (!isValidUser) {
            return new ModelAndView("login", "error", "用户名或密码错误。");
        } else {
            User user = userService.findUserByUserName(loginCommand
                    .getUserName());
            user.setLastIp(request.getLocalAddr());
            user.setLastVisit(new Date());
            userService.loginSuccess(user);
            request.getSession().setAttribute("user", user);
            return new ModelAndView("main");
        }
    }
    …
}
```

现在需要对 LoginController#loginCheck()方法进行单元测试，验证以下几种情况的正确性：

（1）/loginCheck.html 的请求路径是否能正确映射到 LoginController#loginCheck()。

（2）在向 Request 添加用户名 userName 参数及密码参数，并设置不存在的用户名及密码时，返回 ModelAndView("login", "error", "用户名或密码错误。")对象。

（3）在向 Request 添加用户名 userName 参数及密码参数，并设置正确的用户名及密码时，返回 ModelAndView("main")对象。

（4）当登录成功后，检查当前的 Session 属性中是否存在“user”，并验证 user 值的正确性。

要使 LoginController#loginCheck()方法能够成功运行起来，就必须保证该方法所依赖的对象事先准备好。我们通过以下方案解决这一问题：

- 通过 Unitils 从 Spring 容器中加载 RequestMappingHandlerAdapter 和 LoginController 实例。
- 通过 Spring 提供的 Servlet API 模拟类创建 HttpServletRequest 和 HttpServletResponse 实例。

20.7.2　使用 Spring Servlet API 模拟对象

下面联合使用 Spring 模拟类及 Unitils 对 LoginController 进行单元测试，具体实现如代码清单 20-39 所示。

代码清单 20-39　LoginControllerTest

```
package com.smart.web;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.smart.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml",
```

```
        "/web/smart-servlet.xml"})
public class LoginControllerTest extends UnitilsTestNG{

    //① 从Spring容器中加载RequestMappingHandlerAdapter
    @SpringBeanByType
    private RequestMappingHandlerAdapter handlerAdapter;

    //② 从Spring容器中加载LoginController
    @SpringBeanByType
    private LoginController controller;

   //③ 声明Request与Response模拟对象
   private MockHttpServletRequest request;
   private MockHttpServletResponse response;

    //④ 执行测试前先初始化模拟对象
    @BeforeClass
    public void before() {
        request = new MockHttpServletRequest();
        request.setCharacterEncoding("UTF-8");
        response = new MockHttpServletResponse();
    }

    //⑤ 测试LoginController#loginCheck()方法
    @Test
    public void loginCheck() throws Exception{

        request.setRequestURI("/loginCheck.html");
        request.addParameter("userName", "tom");      //⑥ 设置请求URL及参数
        request.addParameter("password", "123456");

       //⑦ 向控制发起请求
       ModelAndView mav = controller.loginCheck(request);
        User user = (User)request.getSession().getAttribute("user");

        assertNotNull(mav);
        assertEquals(mav.getViewName(),"main");
        assertNotNull(user);
        assertThat(user.getUserName(),equalTo("tom"));  ⑧ 验证返回结果
        assertThat(user.getCredit(),greaterThan(5));
    }

}
```

在①处和②处，使用 Unitils 提供的@SpringBeanByType 注解从 Spring 容器中加载 RequestMappingHandlerAdapter 和 LoginController 实例。在③处，声明 Spring 提供的 Servlet API 模拟类 MockHttpServletRequest 及 MockHttpServletResponse，并在测试初始化方法中进行实例化。在⑥处，在模拟类 MockHttpServletRequest 中设置请求 URL 及参数。在⑦处，通过 Spring 提供的注解方法处理适配器向 LoginController#loginCheck()发起请求。在⑧处，通过 TestNG 提供的断言及 Hamcrest 提供的匹配方法验证返回的结果。注意，当运行 LoginControllerTest 测试用例时，并不需要启动 Servlet 容器，用户可以在 IDE 环境下运行该测试用例。

20.7.3　使用 Spring RestTemplate 测试

上文通过 Spring 提供的模拟类并联合 Unitils 测试框架，顺利地完成了 LoginController 的单元测试。下面尝试应用 Spring 提供的 RestTemplate 并联合 Unitils 框架对登录模块的 Web 层进行集成测试。

RestTemplate 是用来在客户端访问 Web 服务的类。和其他 Spring 中的模板类（如 JdbcTemplate、JmsTemplate）很相似，还可以通过提供回调方法和配置 HttpMessageConverter 类来自定义该模板。客户端的操作可以完全使用 RestTemplate 和 HttpMessageConveter 类来执行。要使用 Spring RestTemplate，首先需要在 Spring 应用上下文中进行相应的配置，具体配置如代码清单 20-40 所示。

代码清单 20-40　smart-servlet.xml

```
...
    <bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
        <property name="messageConverters">
            <list>
                <bean id="stringHttpMessageConverter"
                      class="org.springframework.http.converter.StringHttp
    MessageConverter" />
                <bean id="formHttpMessageConverter"
                    class="org.springframework.http.converter.FormHttp
    MessageConverter" />
            </list>
        </property>
    </bean>
...
```

发送给 RestTemplate 方法的对象以及从 RestTemplate 方法返回的对象需要使用 HttpMessageConverter 接口转换成 HTTP 消息，因此，我们在上面配置两个消息转换器 StringHttpMessageConverter 和 FormHttpMessageConverter。配置好 RestTemplate 模板操作类之后，就可以开始编写登录控制器 LoginController 测试用例，具体配置如代码清单 20-41 所示。

代码清单 20-41　LoginControllerTest

```
package com.smart.web;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.smart.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml",
"/web/smart-servlet.xml"})
public class LoginControllerTest extends UnitilsTestNG{

    //① 从Spring容器中加载restTemplate
    @SpringBeanByType
    private RestTemplate restTemplate;
```

```
    //② 从Spring容器中加载LoginController
    @SpringBeanByType
    private LoginController controller;

    //③ 测试LoginController#loginCheck()方法
    @Test
    public void loginCheck() throws Exception{

        //③-1 构造请求提交参数
        MultiValueMap<String, String> map = new LinkedMultiValueMap<String, String>();
         map.add("userName", "tom");
         map.add("password", "123456");

        //③-2 发送客户访问请求
         result = restTemplate.postForObject(
                 "http://localhost:8000/bbs/loginCheck.html", map, String.class);

          //③-3 验证响应结果
         assertNotNull(result);
         assertThat(result, containsString("tom,欢迎您进入小春论坛"));
    }
}
```

在①处和②处，使用 Unitils 提供的@SpringBeanByType 注解从 Spring 容器中加载 RestTemplate 和 LoginController 实例。在③-1 处，构造一个提交请求的参数列表，如实例中设置的用于登录的用户名及密码。在③-2 处，使用 RestTemplate 模板操作类提供的 postForObject()方法发送访问请求。最后在③-3 处，对响应返回的结果进行验证。在 IDE 工具中，在启动 Web 容器之后，运行 LoginControllerTest 测试用例，测试结果如图 20-15 所示。

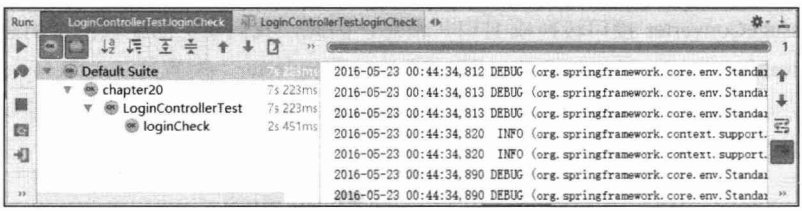


图 20-15 LoginControllerTest.loginCheck()测试结果

20.8 小结

本章讲述了单元测试所需的相关知识，简要分析了目前普遍存在的对单元测试的一些误解、误区及困境。TestNG 测试框架是必须掌握的基础内容，我们对 TestNG 进行了提纲挈领的学习，并着重学习了 TestNG 提供的分组测试、参数测试、依赖测试、数据驱动等特性。

在单元测试中，应该尽量在不依赖外部模块的情况下，重点测试模块内程序逻辑的

正确性。这时，可以通过 Mockito 为测试目标类所依赖的外部模块接口创建模拟对象，通过录制、回放及验证的方式使用模拟对象。通过 Mockito，我们达到了两个看似相对立的目标：一是使测试用例不依赖外部模块；二是使用到外部模块类的目标对象可以正常工作。

Spring 建议用户不要在单元测试时使用 Spring 容器，应该在集成测试时再使用 Spring 容器。手工创建测试夹具或者手工装配测试夹具的工作是单调乏味、没有创意的。通过使用 Unitils，用户就可以享受测试夹具自动装配的好处，将精力集中到目标类逻辑测试编写的工作上。

应该说大部分 Java 应用都是 Web 应用，而大部分 Java Web 应用都是数据库相关的应用，对数据库应用进行测试经常要考虑数据准备、数据库现场恢复、灵活访问数据以验证数据操作正确性等问题。这些问题如果没有一个很好的支持工具，则将给编写测试用例造成挑战。幸好 Unitils 框架为我们搭建好满足这些需求的测试平台，用户仅需在此基础上结合相关框架如 Spring、Hibernate、DbUnit 等就可以满足各种测试场景需要。

为了提高测试 DAO 层的效率，结合 Unitils、DbUnit 框架，编写一个支持 Excel 格式的数据集工厂类，实现使用 Excel 准备测试所需的数据及验证数据，从而大大减少测试 DAO 层的工作量。

对 Service 层的测试，既可以采用 TestNG +Unitils+Mockito 组合，运用 Mockito 强大的模块能力，完成 Service 层独立性测试，也可以采用 TestNG +Unitils+DbUnit 组合，运用 DbUnit 强大的数据库维护能力，完成 Service 层+DAO 层的集成测试。

对 Web 层的测试，我们既可以采用 TestNG +Unitils+Spring Mock 组合，运用 Spring Mock 模拟依赖容器的接口实例，如 HttpServletRequest、ServletContext 等，完成 Web 层中控制器的独立性测试；也可以采用 TestNG +Unitils+Spring RestTemplate，完成 Web 层的集成测试。

《Spring 3.x 企业应用开发实战》读者评价

2012 年 2 月出版

❍当当金卡会员　Doray

一直对国内的技术书籍抱有偏见，所以我经常购买外国人写的技术书籍，决定购买这本书时纠结了很久。看完这本书后，我才发现国内的技术书籍也有相当高的水平。

❍当当金卡会员　轻声曦语

相当有功底的一本书，以通俗易懂的语言剖析了 Spring 的核心功能。本书不仅有实例代码告诉读者怎么使用 Spring 完成代码，还将 Spring 的构建思想表述出来，让人知其然，亦知其所以然。对我的帮助非常大。

❍当当金卡会员　foreveriuu

我越来越喜欢原创作品了，尤其是这种厚积薄发，能够从使用到架构、从相关技术到实际问题的书。

❍当当钻石会员　为了找到更好的

这本书可以说很适合初学者，对于熟悉 Spring 的人来说，也可以发现自己想要的。一直以为自己了解 Spring，看完这本书我才明白 Spring 为什么这么火。

❍京东金牌会员　边走边忘

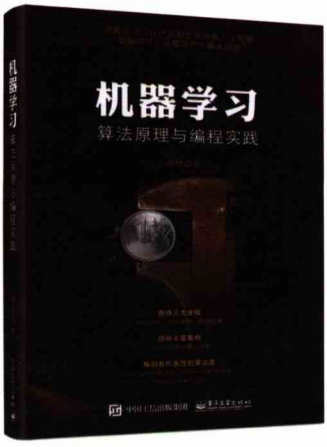
这本书对于理解 Spring 有较大的帮助。作者虽是“程序猿”，但文笔确实不凡，读起来很舒服！

❍京东铜牌会员　lijied

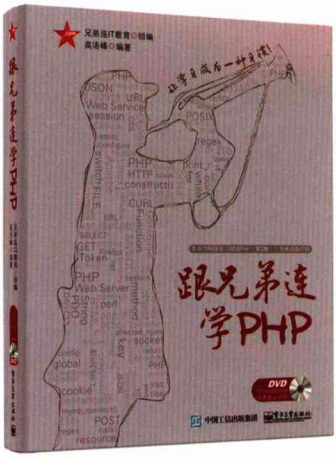
经典，值得购买，幽默，有趣，深入浅出。

❍豆瓣　空气+

无论是从技术的广度还是深度来看，该书都已做到极致，将 Spring 的相关思想、原理、细节描述得深入浅出、淋漓尽致，如最重要的 IoC、复杂抽象的 AOP、很多的实战经验、完整的 Spring MVC 描述，等等，不一而足。该书我仔细读了三遍，作者对 Spring 的相关原理和本质理解得相当深入和透彻。



ISBN：978-7-121-27367-4
2015年10月出版



ISBN：978-7-121-27801-3
2015年12月出版

精通Spring 4.x
企业应用开发实战

本书是在《精通Spring 3.x——企业应用开发详解》的基础上，历时一年的重大调整改版而成的，延续了上一版本“追求深度，注重原理，不停留在技术表面”的写作风格，力求使读者在熟练使用Spring的各项功能的同时透彻理解Spring的内部实现，真正做到知其然并知其所以然。此外，本书重点突出了“实战性”的主题，力求使全书内容体现“从实际项目中来，到实际项目中去”的写作原则。

本书核心内容：

第1篇为基础篇，包括第1～3章，讲解Spring概述性知识，以便读者快速建立对Spring的整体认识，能够使用Spring快速开发一个简单的项目，而更多深入的知识则在后续篇章中展开。

第2篇为核心篇，包括第4～9章，讲解Spring的IoC、AOP及SpEL的知识，这些知识是Spring的核心，也是Spring所有衍生服务及功能的基石。

第3篇为数据篇，包括第10～14章，讲解Spring的各种数据访问技术及事务管理的内容，对事务管理的实现机制和各种疑难问题进行剖析。

第4篇为应用篇，包括第15～18章，讲解数据缓存、任务调度、Web开发的内容，对于企业应用开发来说，数据缓存及任务调度是两个无法回避的问题，需要重点学习和掌握；此外，本篇还精心设计了一个实战案例，包含需求分析、数据库设计、项目开发、代码测试、应用部署的整体过程，让读者在项目的实战中整体串接Spring的知识点。

第5篇为提高篇，包括第19章和第20章，讲解Spring OXM及单元测试的内容。

全书工程代码放在本书配套网盘中，请读者下载网盘内容阅读，本书网盘下载地址如下。
百度云盘：http://pan.baidu.com/s/1boCl3d1。

ISBN 978-7-121-30443-9
9 787121 304439 >
定价：128.00元

责任编辑：李　冰
封面设计：朝天世纪