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。

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

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。

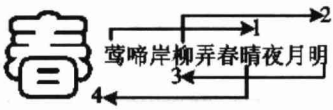
Spring 拥有庞大的社区、活跃的开发队伍、丰富的文档、众多的应用案例。国内的 Spring 社区也异常活跃，对 Spring 在国内的推广起到了推波助澜的作用。下面是其中的一些代表者。

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。

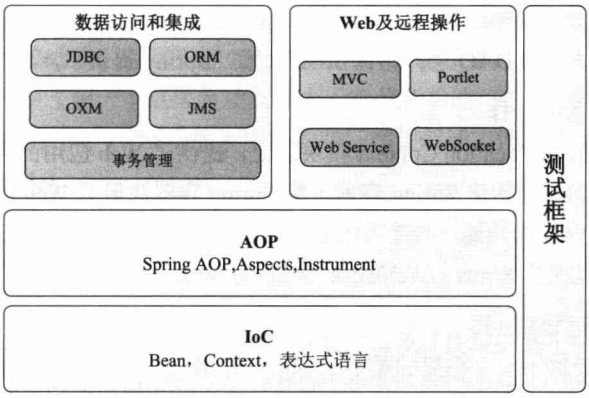


图 1-1　Spring 框架结构

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。

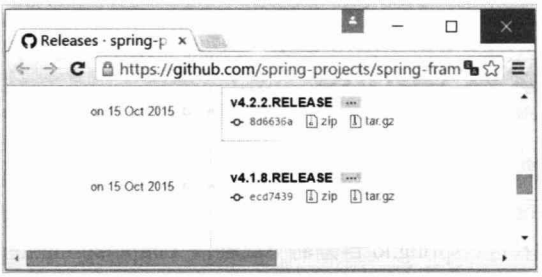


图 1-2　Spring 下载地址

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。

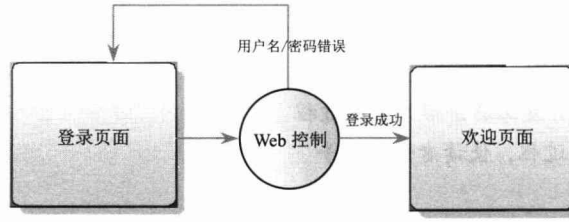


图 2-1 页面流程

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。

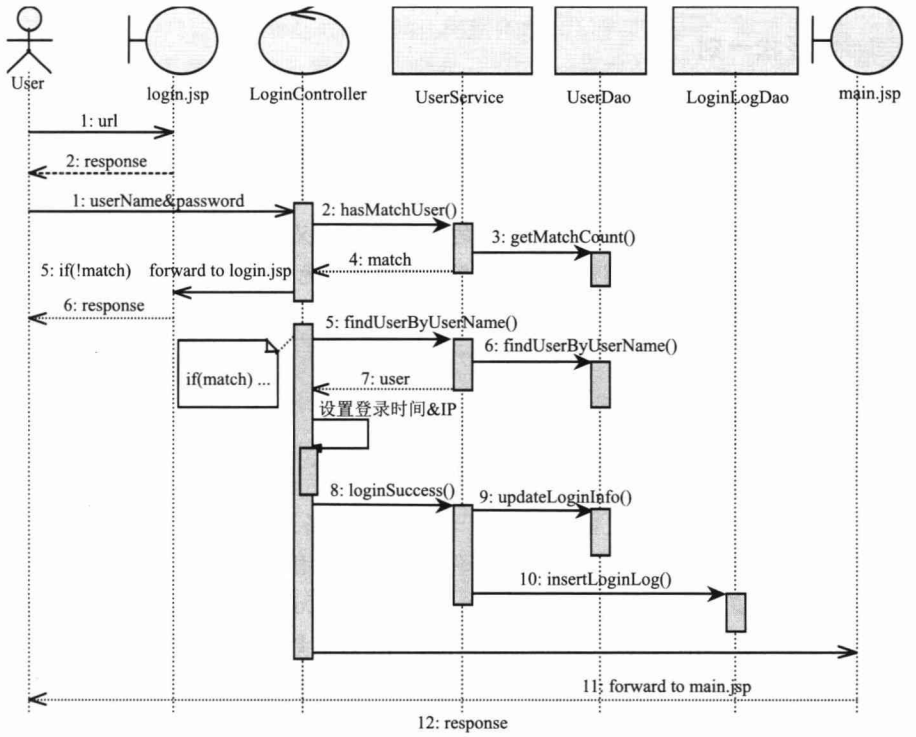


图 2-2　登录模块整体流程

（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。

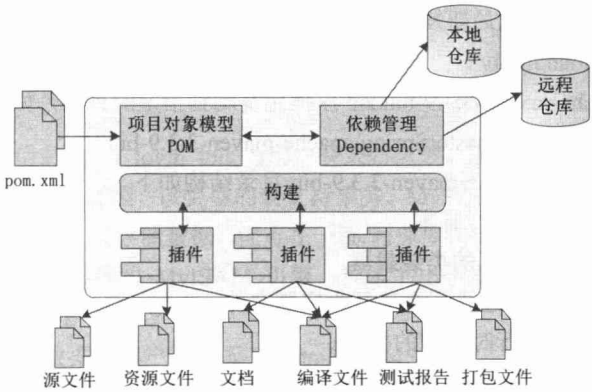


图 2-3　Maven 模型

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。

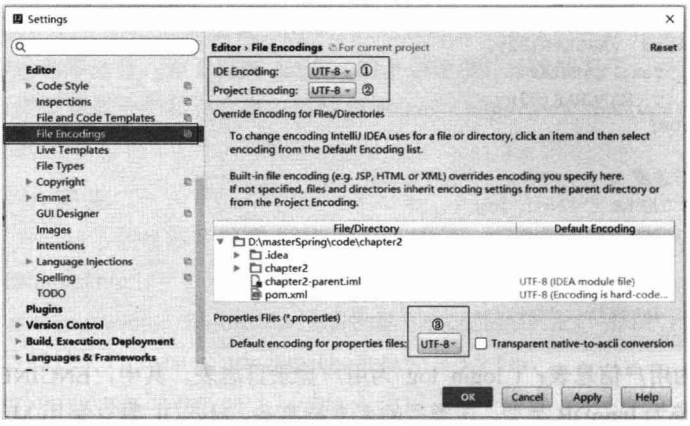


图 2-4　IDEA 工程编码设置

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。

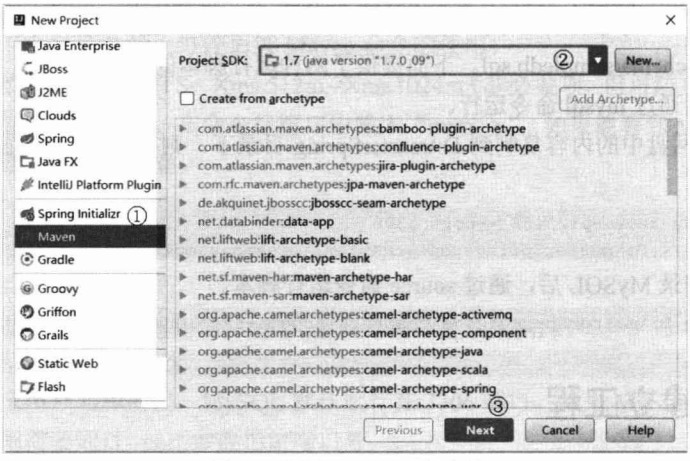


图 2-5　创建 Maven 工程向导

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。

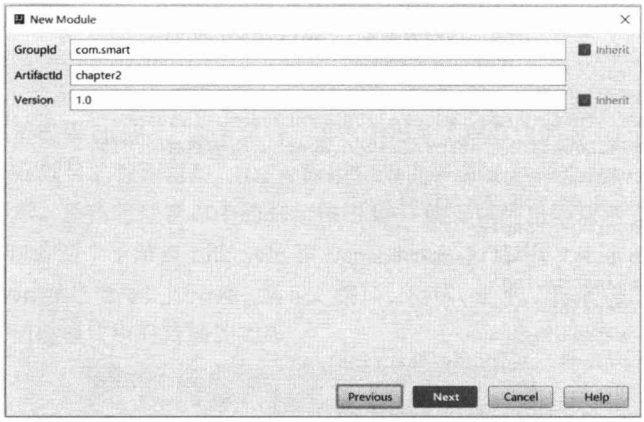


图 2-6　设置 Maven 信息

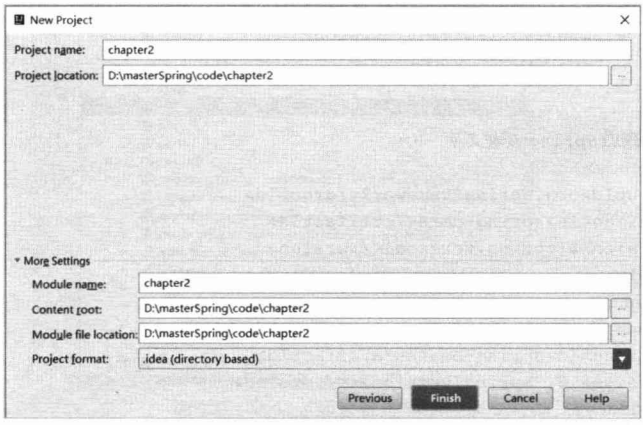


图 2-7　设置工程信息

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。

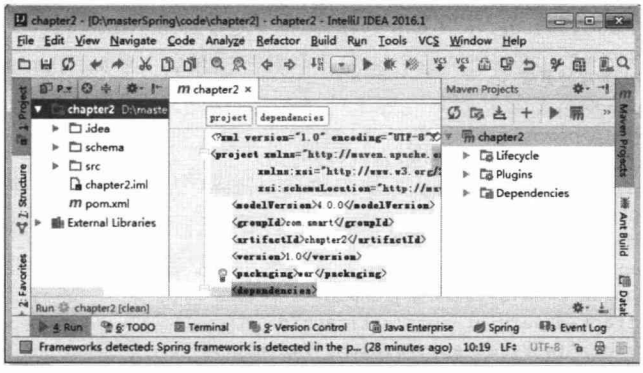


图 2-8　Maven 工程目录结构

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。

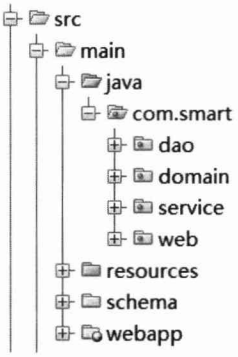


图 2-10 主程序包的规划

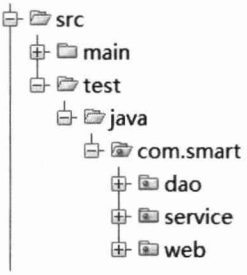


图 2-11 测试包的规划

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。

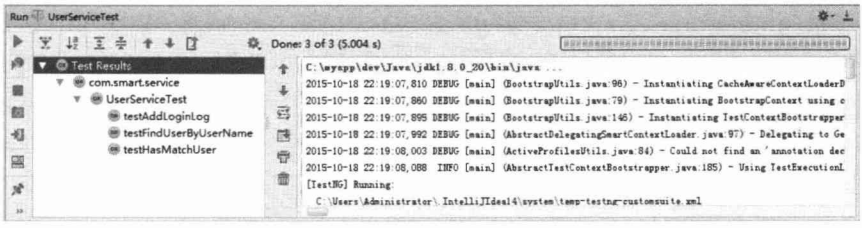


图 2-12　UserServiceTest 的运行结果

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。

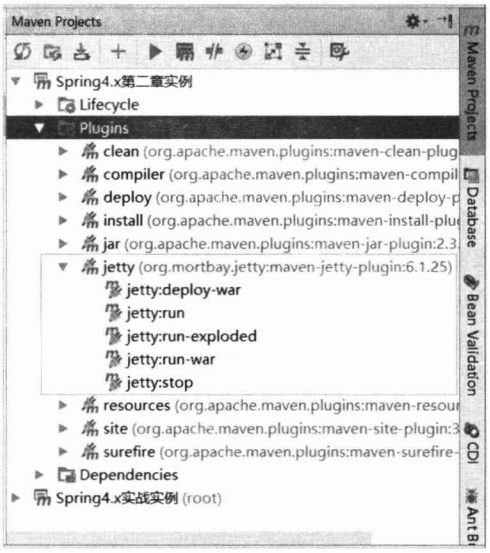


图 2-13　Jetty 插件

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。

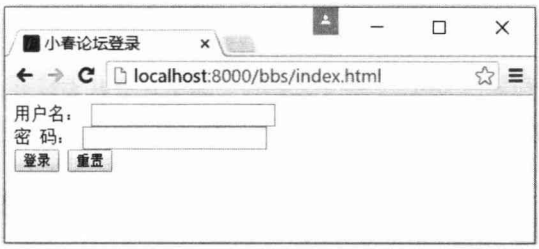


图 2-14　在浏览器中访问应用

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。

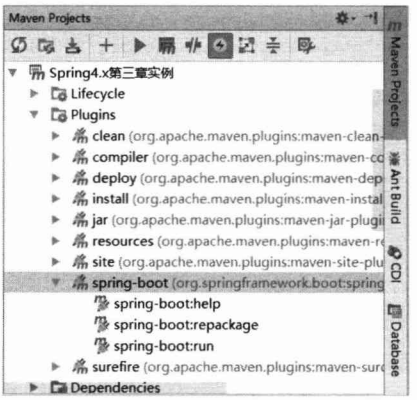


图 3-1　Spring Boot 插件

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。

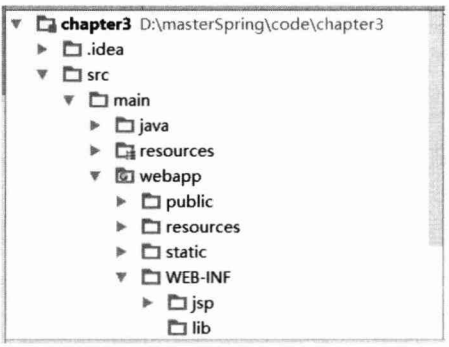


图 3-6　视图目录规划

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。

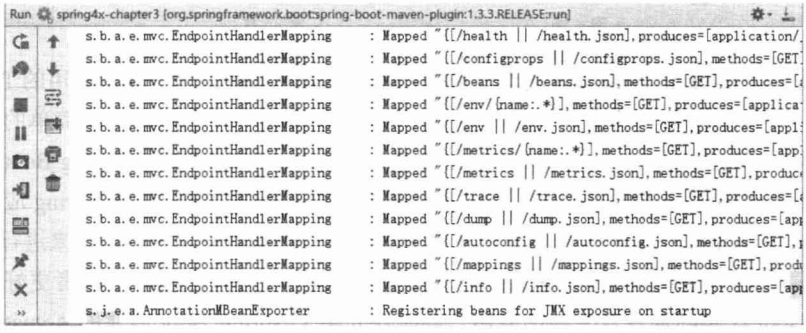


图 3-9　健康检查服务映射信息

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。

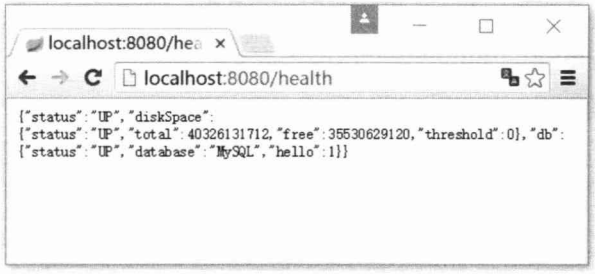


图 3-10　健康检查服务

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

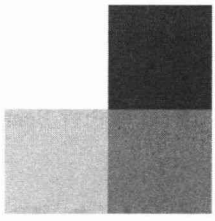
3.8 小结

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。

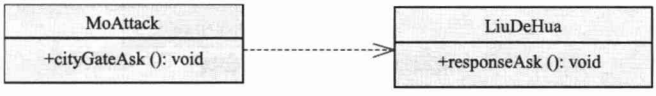


图 4-1　剧本和演员直接耦合

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。

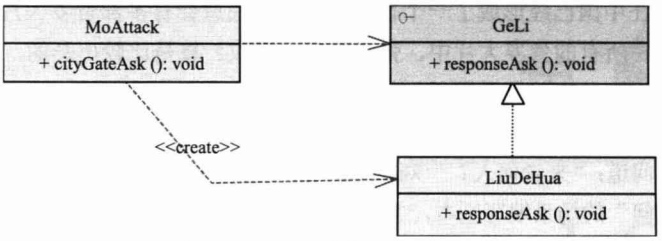


图 4-2　引入角色接口后的关系

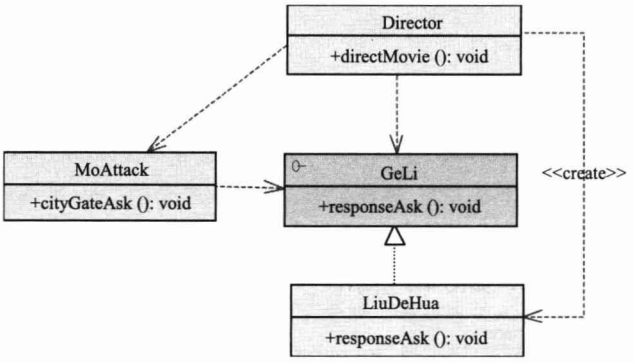


图 4-3　剧本和饰演者解耦

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。

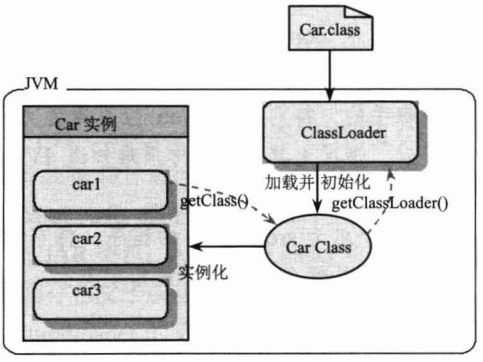


图 4-4 类实例、类描述对象及类装载器的关系

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。

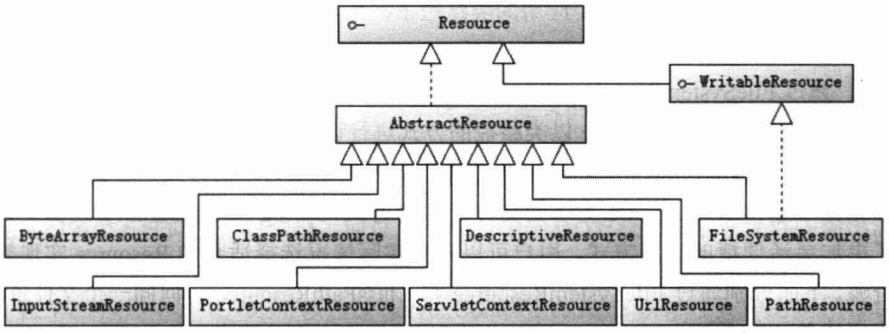


图 4-5　Resource 及其实现类的关系

- 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。

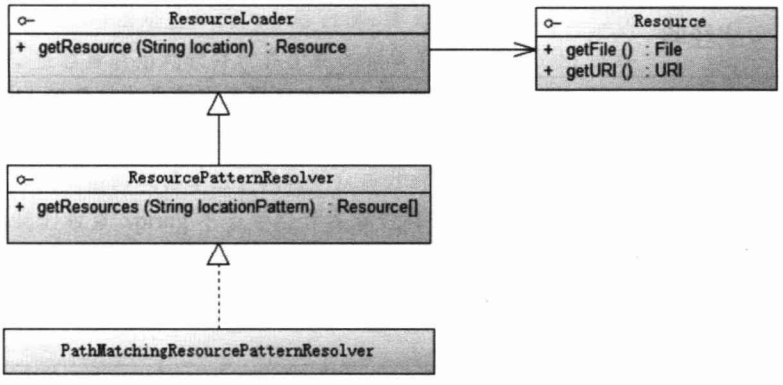


图 4-6　资源加载器接口及实现类

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。

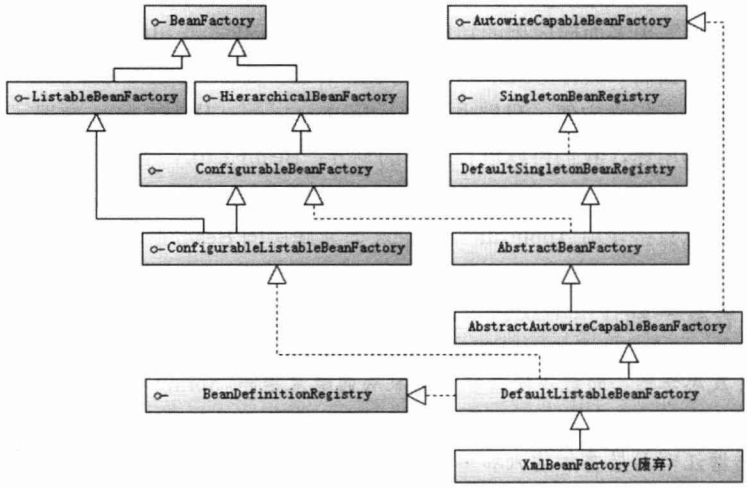


图 4-7 BeanFactory 类继承体系

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。

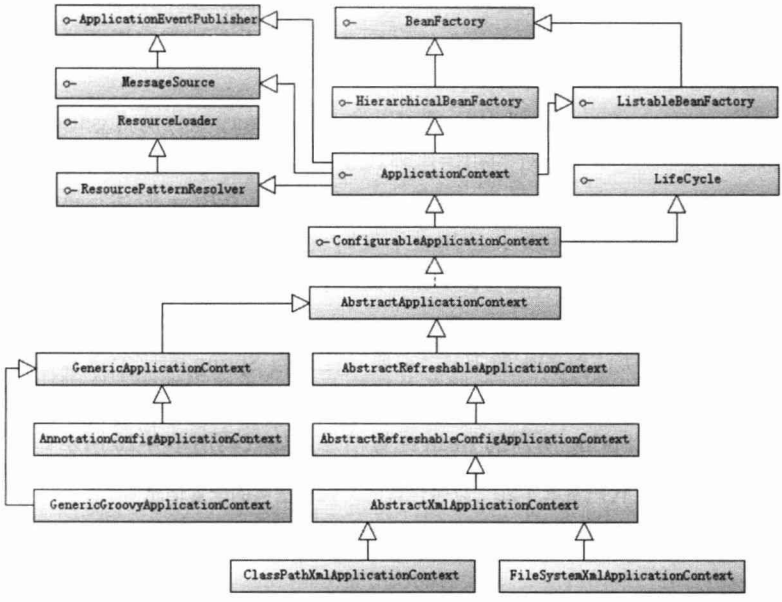


图 4-8　ApplicationContext 类继承体系

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。

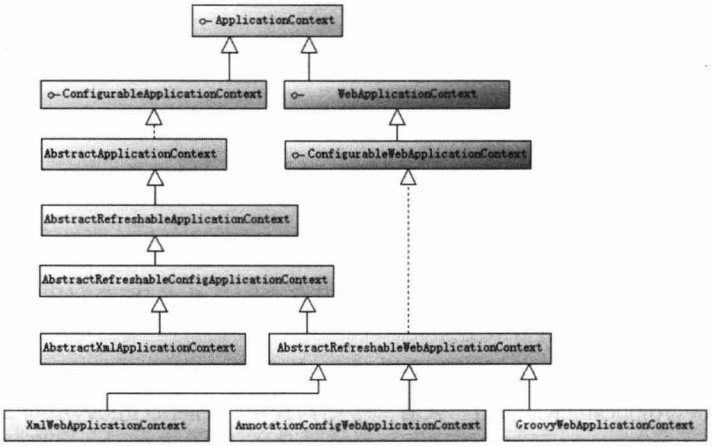


图 4-9　WebApplicationContext 类继承体系

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。

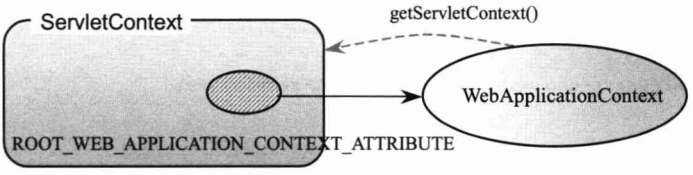


图 4-10　Spring 和 Web 应用的上下文融合

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。

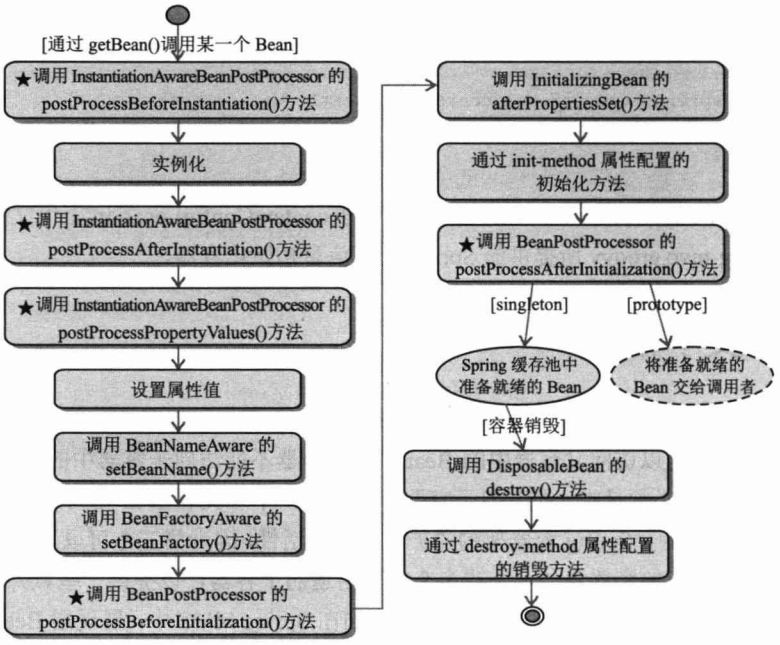


图 4-11　BeanFactory 中 Bean 的生命周期

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。

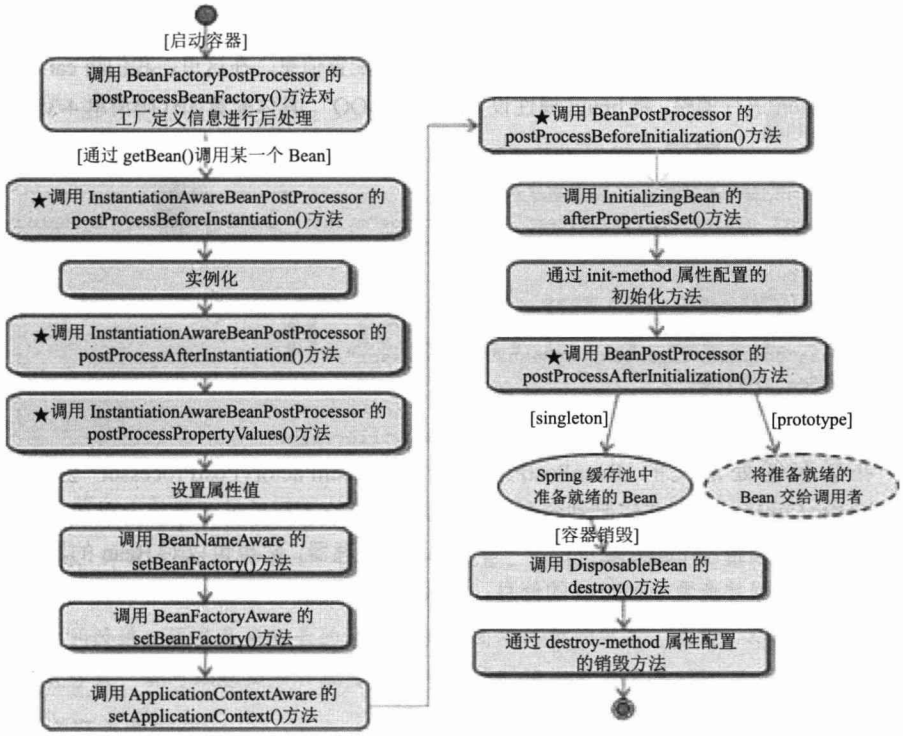


图 4-12　ApplicationContext 中 Bean 的生命周期

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。

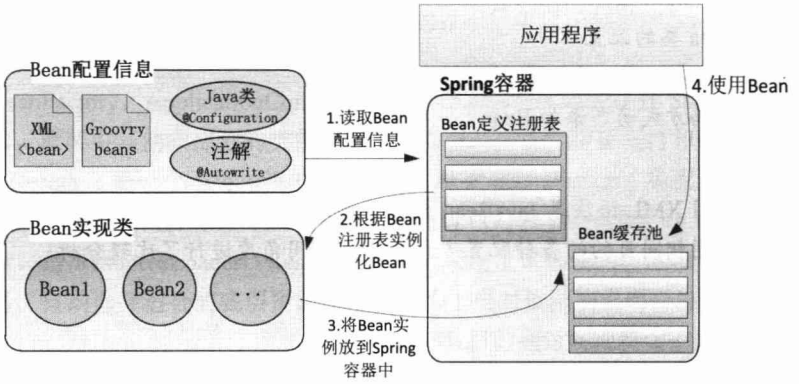


图 5-1 Spring 容器内部协作解构

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。

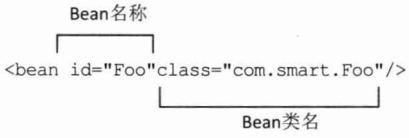


图 5-2 Bean 的定义

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
<	&lt;	“	&quot;
>	&gt;	‘	&apos;
&	&amp;		

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。

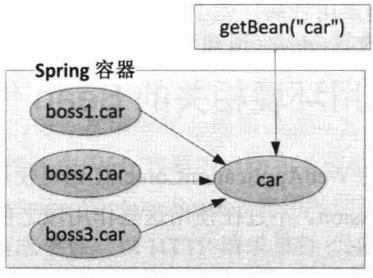


图 5-3　单例模式

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。

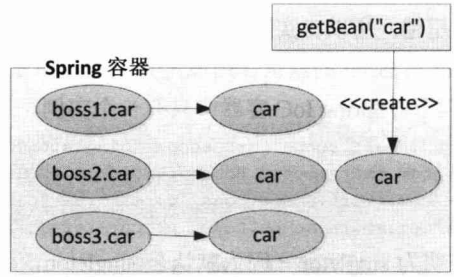


图 5-4　非单例模式

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。

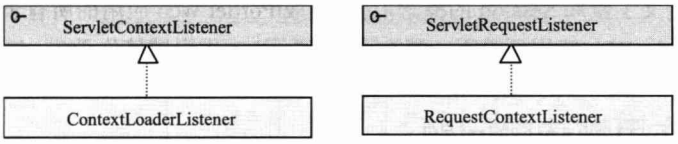


图 5-5　两个监听器的区别

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。

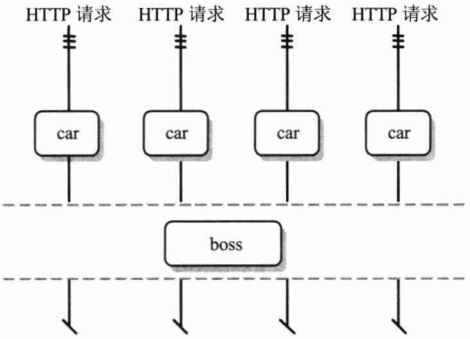


图 5-6　引用 Web 相关作用域的 Bean

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。

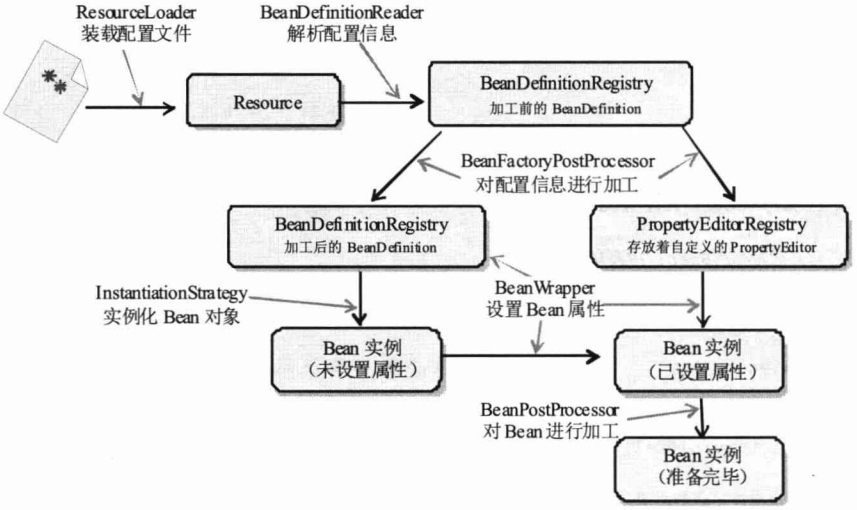


图 6-1　IoC 的流水线

（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。

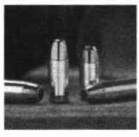
Spring 组件按其所承担的角色可以划分为两类。

（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。

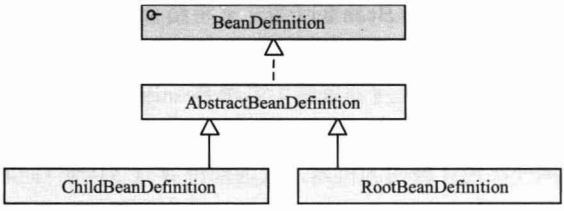


图 6-2 BeanDefinition 类继承结构

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。

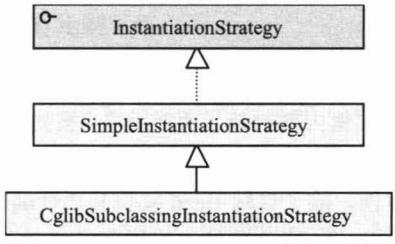


图 6-3　InstantiationStrategy 类继承结构

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。

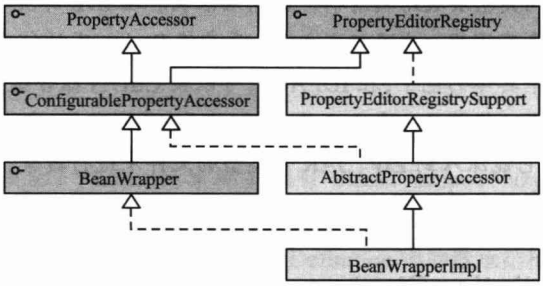


图 6-4 BeanWrapper 类继承结构

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
        = new PropertyDescriptor("titlePosition", ChartBean.class);
      titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
                                                    ①
                                                    将 TitlePositionEditor 绑定到
                                                    ChartBean 的 titlePosition 属性中
      PropertyDescriptor inverseDescriptor
        = new PropertyDescriptor("inverse", ChartBean.class);
      inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
      return new PropertyDescriptor[]{titlePositionDescriptor, inverseDescriptor};
    }                                               ②
    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。

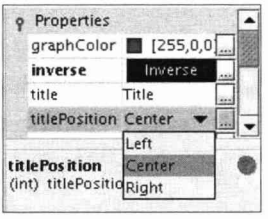


图 6-5　ChartBean 的属性编辑器

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 PropertyEditorRegistrySupport 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。

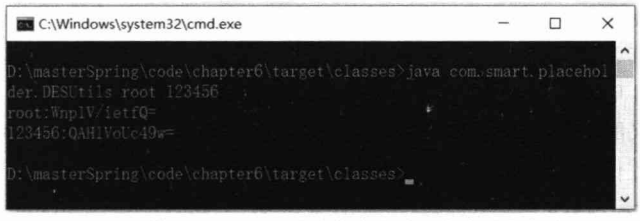


图 6-6　使用 DESUtils 通过命令方式对信息进行加密

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。

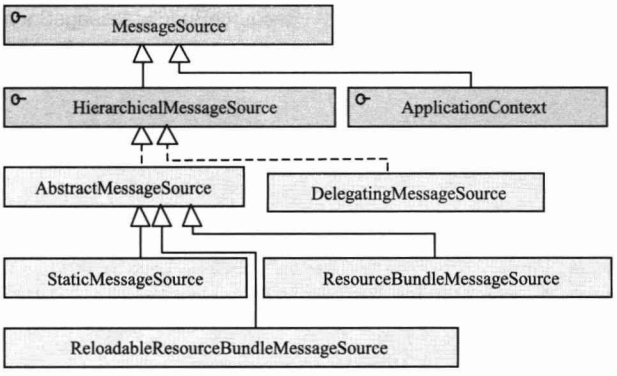


图 6-7　MessageSource 的类结构

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。

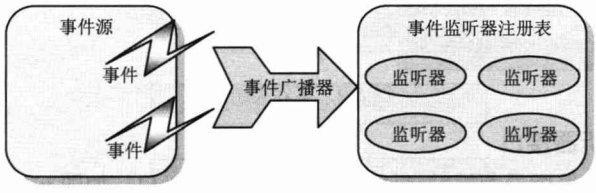


图 6-8　事件体系

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。

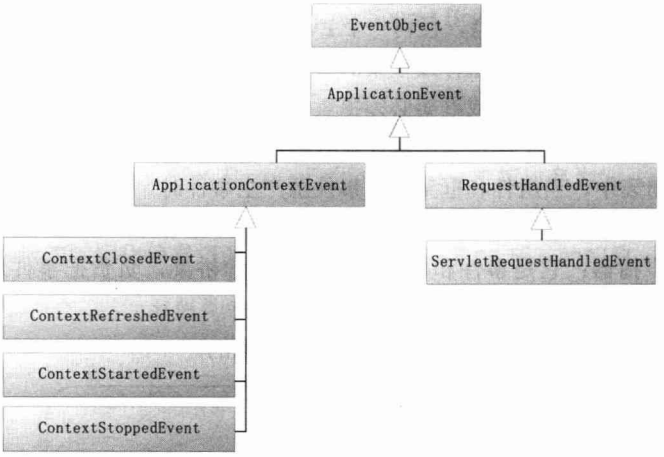


图 6-9　事件类结构

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。

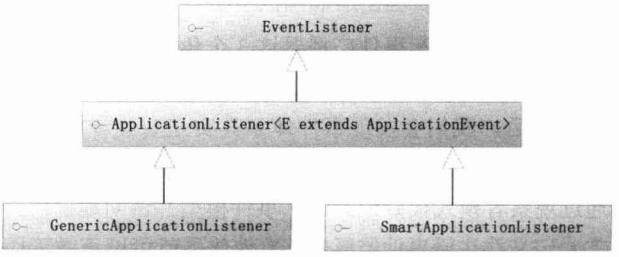


图 6-10　事件监听器接口

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。

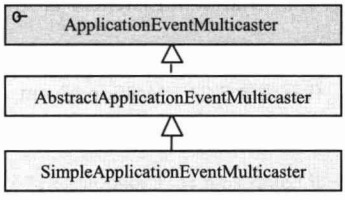


图 6-11　事件广播器类结构

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。

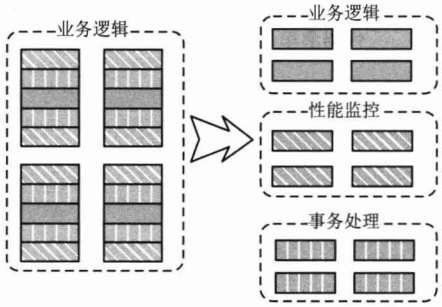


图 7-2　横向抽取

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。

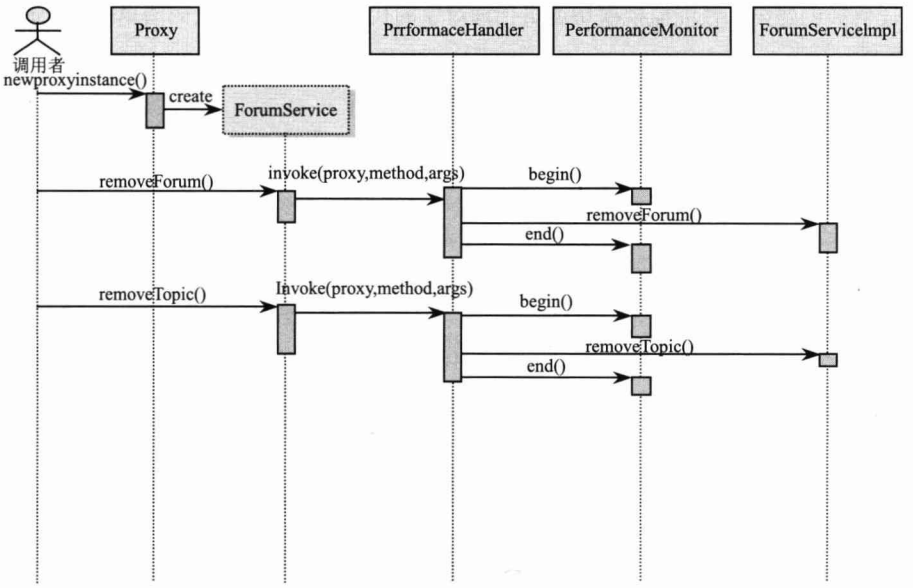


图 7-3　代理实例方法调用的时序图

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。

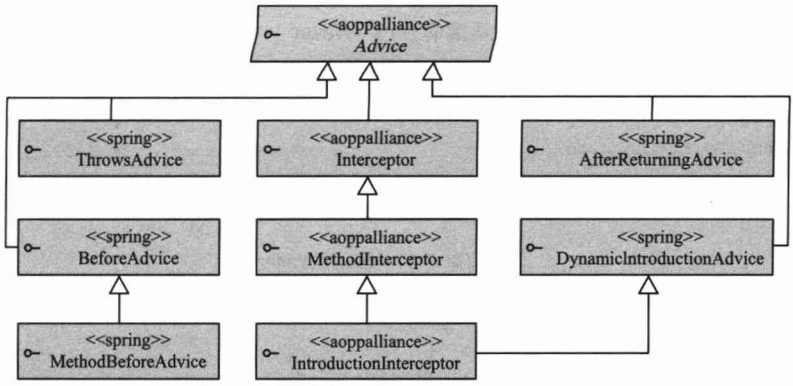


图 7-4　增强接口继承关系图

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。

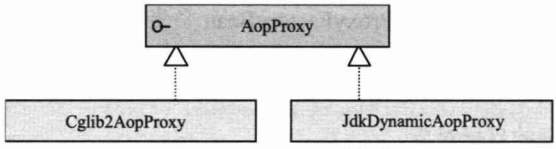


图 7-5　AopProxy 类结构

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。

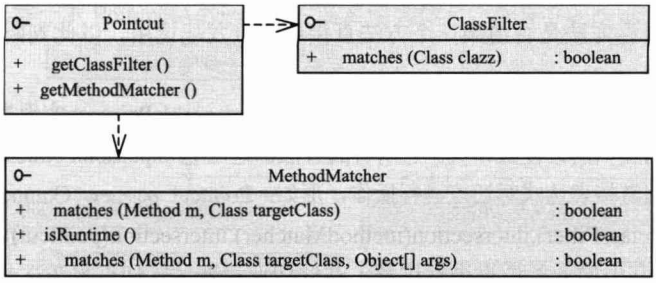


图 7-6　Pointcut 类关系图

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。

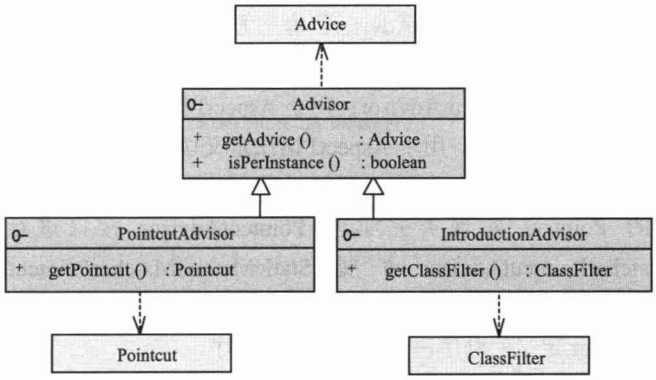


图 7-7　切面类继承关系

- 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。

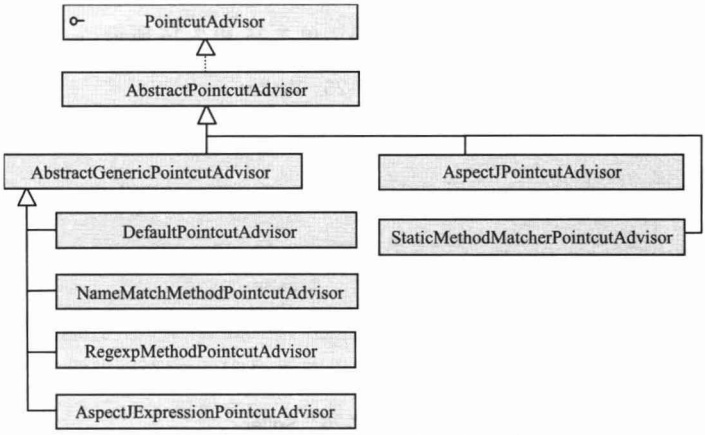


图 7-8　PointcutAdvisor 实现类体系

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