深度解读Spring 3.0源代码，Java社区和Spring社区一致鼎力推荐！

Spring Internals

Spring技术内幕

深入解析Spring架构与设计原理

计文柯◎著

机械工业出版社
China Machine Press

第 1 章

准备源代码环境

半亩方塘一鉴开，天光云影共徘徊。
问渠哪得清如许，为有源头活水来。
——【宋】朱熹 《观书有感》

1.1 安装 JDK

在开发 Spring 应用和分析 Spring 源代码之前，需要做一些准备工作，其中搭建 Java 环境是必不可少的。Spring 3.0 要求 Java 5 以上版本，JDK 需要 1.5 或 1.5 以上版本。如果 JDK 的版本低于 1.5，则可进入网站 http://Java.sun.com/javase/downloads 下载最新的 JDK 安装程序。

提示 安装完后，要检查 JDK 是否配置正确。某些第三方的程序会把自己的 JDK 路径加到系统 PATH 环境变量中。这样，即便安装最新版本的 JDK，系统还是会使用第三方程序所带的旧版本 JDK。这种情况下，Java 环境可能无法正常运行，手工修改系统 PATH 路径可以解决这个问题，比如重新设置 PATH 路径指向新 JDK 的安装路径。

1.2 安装 Eclipse

运行 Spring 只需要 JDK，但是一个好的开发环境和源代码阅读环境可以事半功倍。Eclipse 是最流行的 Java 集成开发环境之一，Eclipse 的 JDT 提供了良好的代码分析功能，它们是我们在分析 Spring 的实现原理过程中会用到的基本工具。比如，可以用 Eclipse 分析 Java 类和接口的继承关系、查看 Java 方法的调用关系、搜索代码等。

在 Eclipse 中分析 Java 类和接口的继承关系的具体做法如下：在代码区中选择需要的类和接口定义，然后使用右键选取 Open Type Hierarchy 或按快捷键【F4】，可以在 Hierarchy View 中看到继承关系，如图 1-1 所示。

在 Eclipse 中分析 Java 方法的调用关系的具体做法如下：在代码区中选择相应的方法定义，然后用右键选取 Open Call Hierarchy 或者按下快捷键【CTRL+ALT+H】，可以在 Call Hierarchy 视图中看到方法的调用关系，提供了逐层的方法调用追溯的功能，对查找方法的相互调用关系非常有用，如图 1-2 所示。

在 Eclipse 中使用搜索功能，使用菜单上的 Search 可以打开搜索对话框，搜索结果在 Search View 中显示。如果双击 Search View 中的搜索结果列表中的某一项，就可以直接打开该搜索结果对

Download at http://www.pin5i.com/

应的出处，具体如图 1-3 所示。

图 1-1　在 Eclipse 中查看类的继承关系

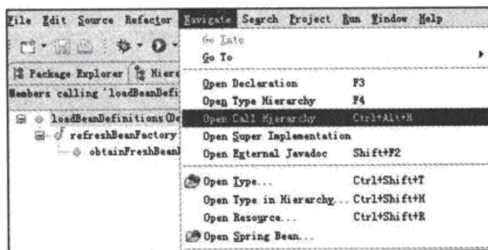


图 1-2　在 Eclipse 中参看方法调用关系

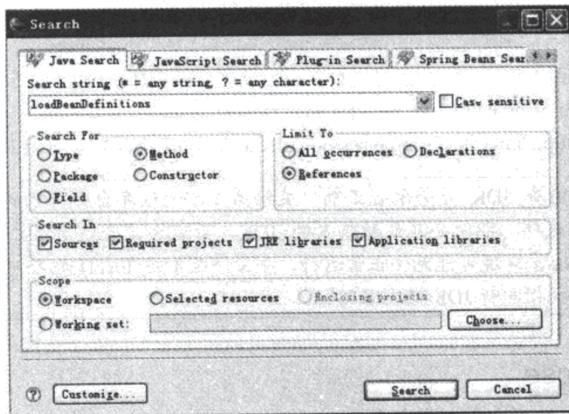


图 1-3　在 Eclipse 中使用搜索功能

1.3　安装辅助工具

作为集成的 IDE 环境，Eclipse 为许多相关的插件提供了应用平台。例如，SVN 的 Eclipse 插件为开发者提供了很好的源代码管理功能，Spring 的 Eclipse IDE 插件为开发 Spring 应用提供了很好的帮助。对应的插件在 Eclipse 中安装以后，这些功能就可以直接在 Eclipse IDE 环境中使用；通常这些插件是作为 Eclipse 的一个 View 或者一个 Perspective 出现在集成 IDE 环境中的。下面我们来看看这些插件在 Eclipse 中的安装。我们先对在 Eclipse 中安装 SVN 插件做一个简要的说明，关于 Spring Eclipse IDE 的安装和简要使用，将在第 8 章中详细介绍。

关于源代码管理工具的具体使用，本书就不做详细的说明了，记得早在 10 几年前我使用的版本管理工具是 CVS。当时是在一台 SUN Ultra60 工作站上与版本管理工具 CVS 有了第一次亲密接触。它有一个简单的图形界面，非常直观，使用起来也很方便，从那时候起，我才开始有了源代码

Download at http://www.pin5i.com/

管理的基本概念。后来发现，这些概念在其他工具上也是同样适用的，例如版本、分支、基线、合并、比较等。

慢慢更深入地接触了软件产品开发以后，知道这些操作都是在进行软件配置管理计划时需要定义的基本过程，这些基本过程保证了软件产品在协同开发和构建过程中的一致性，成为软件开发中不可缺少的变更管理的重要组成部分。无论是使用 CVS、SVN 还是 ClearCase，是使用命令行方式还是图形界面的方式，基本的概念都需要深刻的理解和掌握。ClearCase 在大型商业软件开发领域被普遍使用，而 CVS 和 SVN 却是开源领域和中小型企业应用开发领域的主角。

值得注意的是，CVS 和 SVN 本身也是开源软件，提供的功能已经足以满足开源软件开发的需求。这里有个有趣的问题就是，不知道 CVS 和 SVN 本身的开发项目是不是用自己做的代码管理？感兴趣的读者不妨去做个调研来告诉大家。如果读者对开源软件的源代码管理感兴趣，可以阅读 *Open Source Development with CVS*（作者 Karl Fogel 和 Moshe Bar）这本书，书中详细介绍了如何使用 CVS 对开源软件的源代码进行管理。

Eclipse 中自带的源代码管理客户端就是 CVS 的客户端，Eclipse 使用者可以直接使用。此外，这个 CVS 客户端因为有 Eclipse 的各种视图的支持，所以使用起来也是非常的方便。但是，由于 Spring 3.0 的源代码是用 Subversion 来进行管理的，而在 Eclipse 的默认安装中是不带 SVN 客户端这个插件的，所以这里有必要对 SVN 客户端的安装和使用做简要的介绍。这些源代码管理工具是开源软件开发中常用的，所以掌握这些工具的基本使用对我们理解开源软件的开发方式有很大的帮助。我们在这里使用的是一个开源的 SVN 客户端——Subclipse，一个在 Eclipse IDE 中使用的 SVN 插件。其官方网站是 http://subclipse.tigris.org，其下载页面如图 1-4 所示。

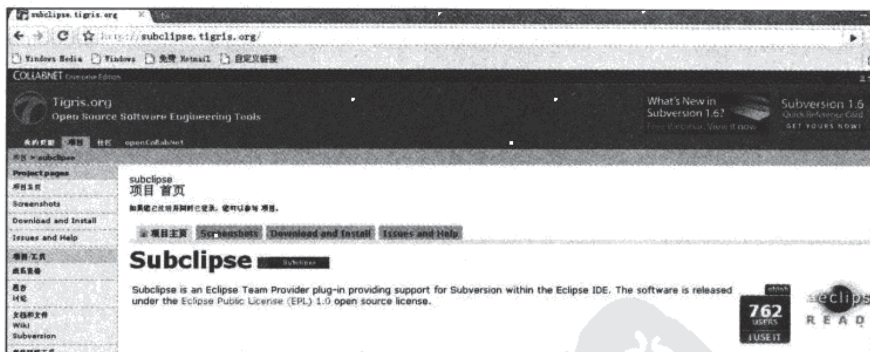


图 1-4　SVN 客户端下载

在 Eclipse 的 Help 菜单下选择 Software Updates，如图 1-5 所示。

打开 Software Updates and Add-ons 对话框，选择 Available Software 面板，可以看到在我们的 Eclipse IDE 环境中已经安装的插件，如图 1-6 所示。

单击 Add Site...按钮打开 Add Site 对话框。具体的 Eclipse 更新地址可以在 http://subclipse.tigris.org/单击 Download and Install 页面找到，我们在 Location 中输入插件更新地址，结果如图 1-7 所示。

Download at http://www.pin5i.com/

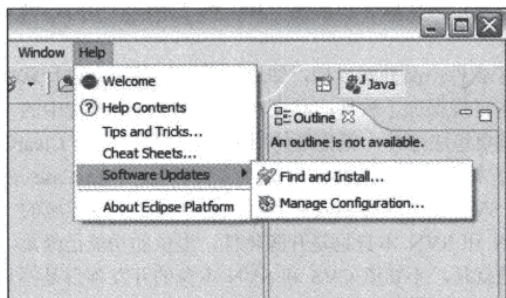


图 1-5　在 Eclipse 中安装插件

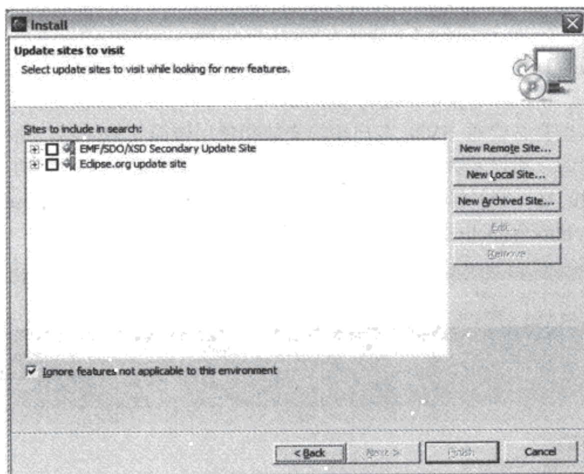


图 1-6　在 Eclipse 中安装插件

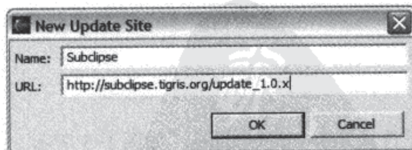


图 1-7　在 Eclipse 中安装插件

单击 OK 按钮关闭 Add Site 对话框。Eclipse 获取到插件列表，更新 Software Updates and Add-ons 对话框的内容，并选择 Subdipse update sites，如图 1-8 所示。

Download at http://www.pin5i.com/

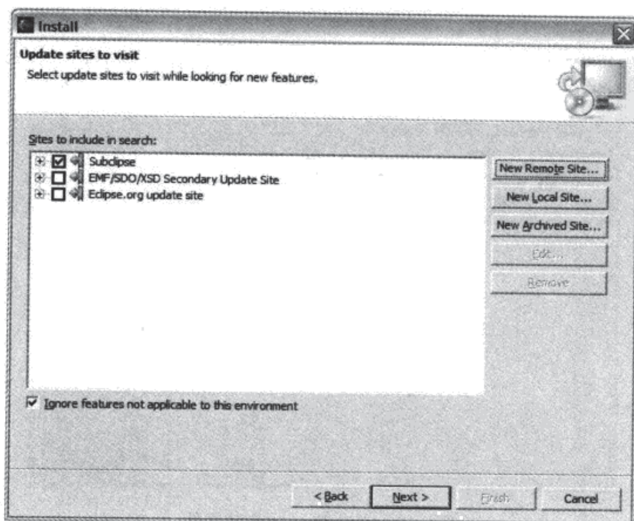


图 1-8　在 Eclipse 中安装插件

勾选 Subclipse 插件及其子项，单击 Next 按钮，如图 1-9 所示。

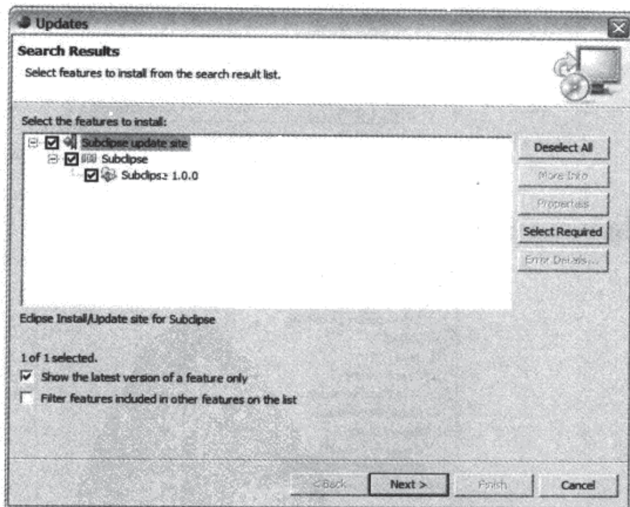


图 1-9　在 Eclipse 中安装插件

单击 Install 按钮，显示 Progress Information 进度对话框开始下载安装包，下载完成后的界面如图 1-10 所示。

Download at http://www.pin5i.com/

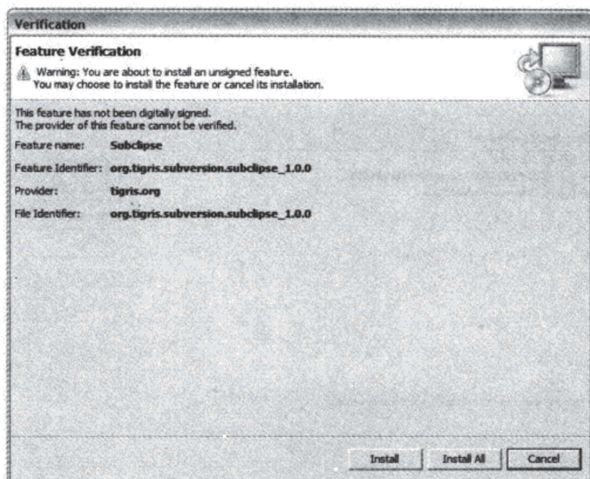


图 1-10　在 Eclipse 中安装插件

插件安装完成，重新启动 Eclipse，SVN Eclipse 插件安装完成，如图 1-11 所示。

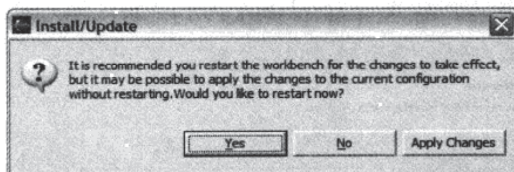


图 1-11　在 Eclipse 中安装插件

这时可以看到在 Eclipse 中 SVN 的相关视图，如图 1-12 所示。

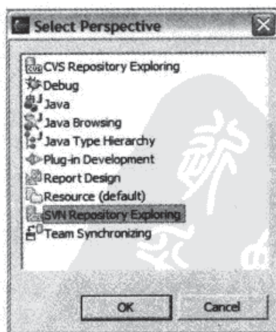


图 1-12　Eclipse 中安装好的 SVN 视图

Download at http://www.pin5i.com/

如果你不喜欢在 Eclipse 中使用 SVN 客户端，还可以直接下载 SVN 的独立客户端进行使用，比如 TortoiseSVN（一只可爱的小海龟），这也是一个开源的源代码管理工具，可以在 http://tortoisesvn.tigris.org/下载，如图 1-13 所示。

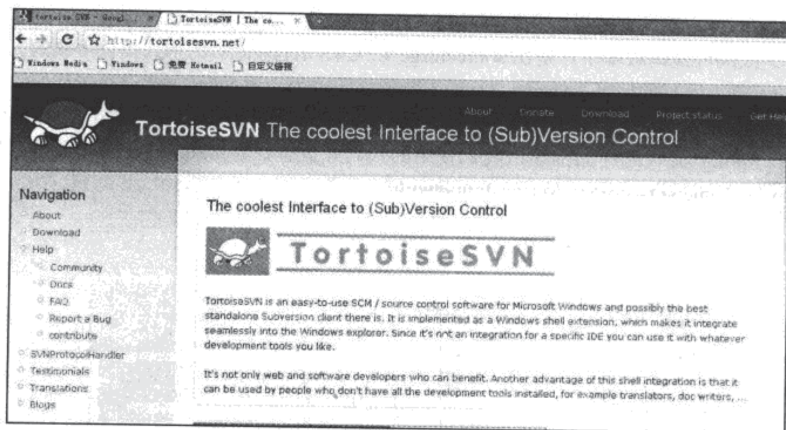


图 1-13　下载 SVN 独立客户端 TortoiseSVN

下载安装完成后，如果打开 Windows 的文件管理器，点击右键，可以看到 SVN 的菜单选项已经在操作列表里出现了，由于集成在 Windows 的文件管理器中，这个小海龟使用起来也是非常方便的。如图 1-14 所示，可以看到小海龟的 SVN 操作选项列表。比如 checkout 和创建代码库等等。从这里看到的 SVN 的基本功能，和我们在 Eclipse 中使用的 SVN 客户端的基本功能是类似的，但在这里的 TortoiseSVN 做为独立的 SVN 客户端，是脱离 Eclipse 而独立使用的，所以在 Eclipse 中浏览代码的时候需要注意对代码的更新和同步。

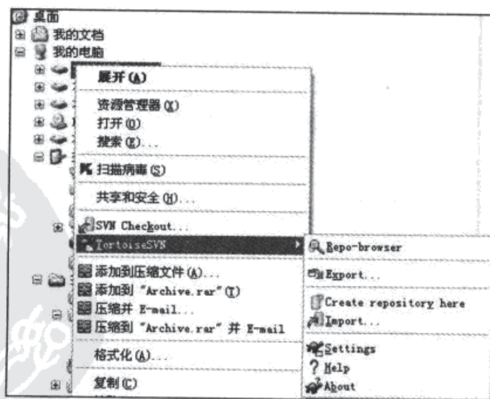


图 1-14　SVN 客户端在 Windows 系统中的使用

Download at http://www.pin5i.com/

1.4　获取 Spring 源代码

有了 SVN 客户端的支持后，我们可以到 Spring 的官方网站上获取 Spring 3.0 的源代码。在 Spring 3.0 之前的源代码版本中，是在 sourceforge 以 CVS repository 的形式提供下载的。但是，根据当前 Spring 官方网站的信息，Spring 3.0 版本的源代码改为使用 SVN 方式进行源代码管理。进入 Spring 官方网站 http://www.springsource.org/，打开 Project，然后选择 Spring（如图 1-15 所示），可以看到关于 Spring 源代码库的描述，我们使用的代码库位置在 Spring 官方网站上可以找到，即 https://src.springframework.org/svn/spring-framework/。

图 1-15　Spring 官方网站

确定了 Spring 源代码的下载位置后，我们开始使用 Eclipse SVN 客户端进行源代码下载，具体过程是：在 Eclipse→Windows→Open Perspective→Others，打开 SVN Perspective，如图 1-16 所示。

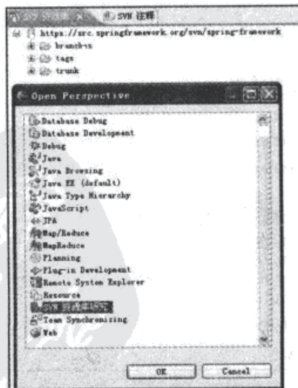


图 1-16　下载 Spring 源代码

Download at http://www.pin5i.com/

单击添加 SVN 资源库，添加 Spring 源代码下载的代码位置，单击 Finish，如图 1-17 所示。

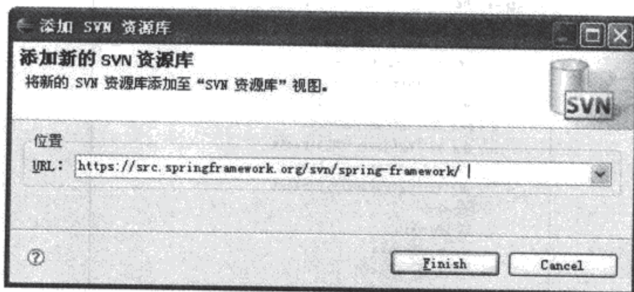


图 1-17 下载 Spring 源代码

这时在 SVN 资源库视图中可以看到详细的 Spring 源代码的代码库结构，因为是开源项目，Spring 的代码库对所有人开放 check out 权限。但是，如果要 check in 代码，则需要相应的 SVN 权限作为 developer 才可以，用右键点击 trunk，检出代码，单击 Finish，如图 1-18 所示。

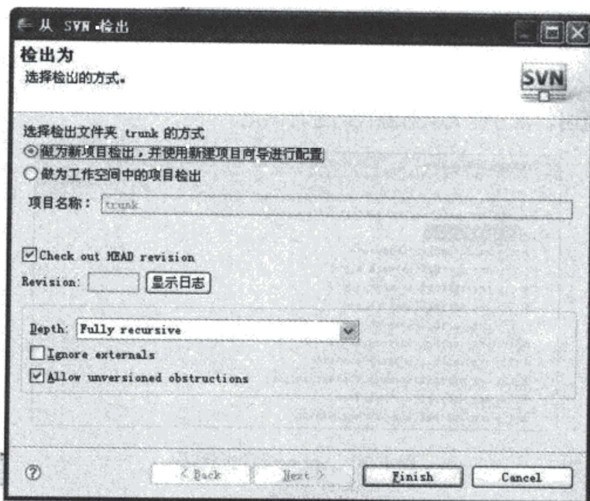


图 1-18 下载 Spring 源代码

这时会经历较长时间的代码检出过程，这个过程把 Spring 代码从服务器检出到 Eclipse 的本地工作环境。代码检出完毕后，可以把 Eclipse 切换到 Java Perspective，这时已经可以在 Packet Explorer 中看到与 Spring 代码库同步的代码了，如图 1-19 所示：

这时就可以像在你自己的项目里使用 Eclipse 浏览代码那样来浏览 Spring 的源代码了，整个 Spring 的源代码已经在我们面前毫无保留、一览无遗地呈现了。

Download at http://www.pin5i.com/

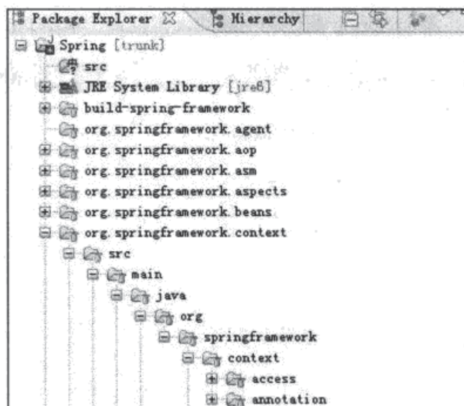


图 1-19 下载 Spring 源代码

1.5 Spring 源代码的组织结构

我们已经把 Spring 的源代码检出到了本地，整个代码的结构如图 1-20 所示，我们对它的目录结构做一些简要的说明，以方便大家对源代码进行浏览。

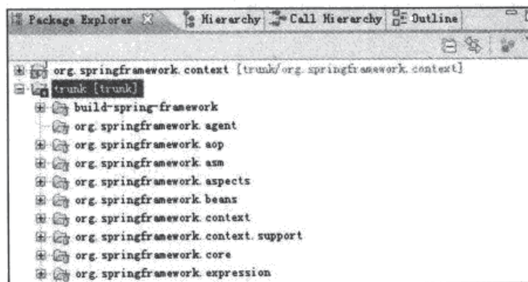


图 1-20 Spring 源代码结构

`Build-spring-framework` 是整个 Spring 源代码的构建目录，里面是项目的构建脚本，如果要自己动手构建 Spring，可以进入这个目录使用 ANT 进行构建。其他的目录从名字上就可以很容易地看出来是 Spring 的各个组成部分，比如，`org.springframework.context` 是 IoC 容器的源代码目录，`org.springframework.aop` 是 AOP 实现的源代码目录，`org.springframework.jdbc` 是 JDBC 的源代码部分，`org.springframework.orm` 是 O/R Mapping 对应的源代码实现部分，`org.springframework.samples.petclinic` 是 Spring 提供的一个应用示例的源代码，便于我们开发 Spring 应用时参考。

Download at http://www.pin5i.com/

Spring 源代码中的每个包（比如 org.springframework.context）都以一个相对独立的子项目存在于代码库中。之所以说这些包是子项目，是因为每个包都可以作为独立的项目导入到 Eclipse 中，都有 Eclipse 的项目配置文件，有针对这些包的代码的测试用例，这些测试用例组织在 src/test 目录中。另外还有针对自己包的 build 构建文件，这些构建文件同时也是构成整个 Spring 项目构建的一部分。这种代码组织结构使得包之间的相互耦合相对较小，非常有利于各个子模块的并行开发、集成与测试。在每个源代码包中，都有着类似的代码结构划分，比如 src 是源代码目录，其中的 main 目录用来存放产品代码，test 用来存放测试代码。main 里面的 java 目录用来存放 java 源文件，而 resources 目录用来存放资源文件。target 目录用来存放编译好的 classes 文件，这个 target 名字让我想起了在嵌入式软件的开发系统中也常看到这样的目录，在那些系统里，这些目录常用来存放目标代码，往往还可以针对不同的处理器结构和平台（比如 X86 平台、PPC 平台、ARM 平台等）。在这里，因为 Java 的跨平台特性，所以只要一个 target 即可，也许这些名字也是 Java 起源于嵌入式系统开发的佐证之一吧。这些代码的组织规划很统一，让整个 Spring 的源代码看起来非常整齐，浏览起来非常方便。图 1-21 以 org.springframework.context 包为例，大家可以体会一下。

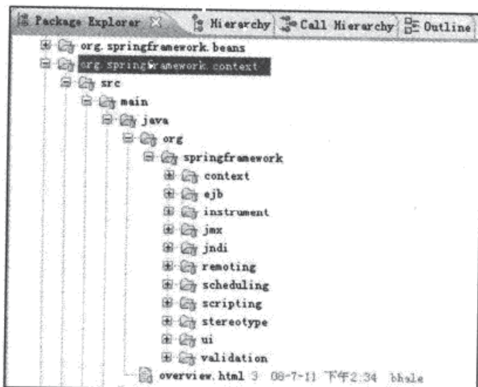


图 1-21　Spring 源代码包的内部结构

下面看看整个 Spring 的代码库结构。在整个代码库中，我们前面 check out 的是 trunk 上的代码，也就是当前 Spring 最新的提交代码，相当于 CVS 中的 HEAD 版本。如果要获取 Spring 3.0 的某个基线版本的代码，可以在 tags 目录中看到，然后 check out 到本地即可，这个过程和 check out 在 trunk 上的代码是一样的。在使用 SVN 和 CVS 做代码管理的时候，它的基本思路也和我们使用 ClearCase 的时候不太一样，在 ClearCase 里，如果要检出代码，一般需要先拉一个分支，然后再做 check out，这样对并行开发没有影响，同时使用 ClearCase 的时候往往需要对各种分支的权限做比较详细的规划；但在 CVS 和 SVN 中，管理过程有很大的不同，往往代码的检出和提交是比较自由的，但是对基线版本的构建却是比较严格的，所以在这里不太能够看到分支的使用；这也是由开源软件的开发特点决定的。让我们回到前面的目录结构，在 tags 目录下，我们可以看到现在已经开发好的基线版本，比如 Spring3.0 M1/M2/M3 等，这些都是一些重要的开发里程碑，熟悉软件配置管理的读者看到这里一定会发出会心的一笑，感觉又回到了我们在日常产品代码开发中熟悉的配置环

Download at http://www.pin5i.com/

境中来了。如果你觉得基线版本的代码更新太慢，而你又急切地想了解代码的最新改动，可以直接参看 trunk 的代码，这绝对是 Spring 开发团队最新出炉的作品。从这点上看，互联网真的是把整个世界都变平了。另外一点值得注意的是，从现在的代码组织结构上可以看到，这个代码仓库是从 Spring 3.0 开始为 Spring 服务的，因而无法在这里找到 Spring 3.0 版本以前的 Spring 源代码。如果需要获取之前版本的源代码，需要找到以前代码库的具体位置，这些信息可以到 Spring 的官方网站上获取。现在我们看到的代码库的快照如图 1-22 所示。

图 1-22　Spring 源代码库的结构

这就是我们看到的 Spring 源代码！经过这么多年的发展，其核心已经比较稳定了，包括各个基本包的设计和命名。同时，我们从这些源代码的组织也隐约地看到了 Spring 的配置管理和构建过程，比如项目组织、测试管理、构建工具以及依赖关系管理工具的使用等，这些都为 Spring 代码的高质量开发奠定了一个良好的工程环境。有兴趣的读者不妨自己做个研究，看看 Spring 的构建过程是怎样完成的。

1.6　小结

问渠哪得清如许，为有源头活水来。本章我们从 Spring 的源头开始，对 Spring 源代码的工程环境进行了介绍，通过这些介绍，读者已经具备自己动手对源代码进行分析的能力。另外，我们使用 Eclipse 开发环境，对源代码进行分析的一些实践经验，以及和开源软件开发过程紧密相关的一些基本知识进行介绍，这些知识不仅对 Spring 适用，而且对其他的开源软件开发也具有非常好的借鉴意义。

有了软件工程环境的有力支持，以及在这个环境基础上，对 Spring 源代码结构的基本了解，我们已经具备了阅读 Spring 源代码的基本条件，就是这些源代码，是我们深入了解 Spring 实现原理的有力武器，也是开源软件最宝贵的财富。让我们养成动手到源代码中去看个究竟以解迷惑的习惯吧，从个人的切身体验来看，这可是一个深入了解软件实现原理的好习惯。对实现的商业软件产品也是一样如此。让我们瞭望一下这片生机勃勃而又有些神秘的代码丛林，就像探险者面对茂密的热带雨林那样，一起做个深呼吸，准备开始这充满乐趣和挑战的 Spring 源代码之旅吧！

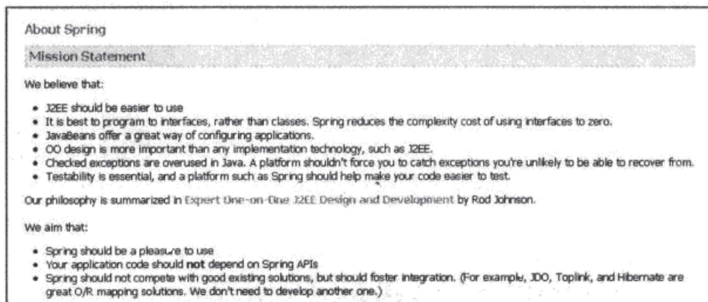
Download at http://www.pin5i.com/

第一部分

Spring核心实现篇

Download at http://www.pin5i.com/

本篇将对 Spring 的核心 IoC 容器和 AOP 的实现原理进行阐述。IoC 容器和 AOP 是 Spring 的核心，是 Spring 系统中其他组件模块和应用开发的基础。从两个核心模块的设计和实现上可以了解到 Spring 倡导的对企业应用开发所应秉持的思路，比如使用 POJO 开发企业应用、提供一致的编程模型、强调对接口编程等。对于这些 Spring 背后的开发思想和设计理念，大家都不会陌生，在 Rod Johnson 的经典著作里都有全面和深刻的讲解。作为参考，我们可以看到 Spring 官方网站对 Spring 项目的描述。如下图所示，Spring 的目标和愿景写得很清楚。


About Spring

Mission Statement

We believe that:

- J2EE should be easier to use
- It is best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero.
- JavaBeans offer a great way of configuring applications.
- OO design is more important than any implementation technology, such as J2EE.
- Checked exceptions are overused in Java. A platform shouldn't force you to catch exceptions you're unlikely to be able to recover from.
- Testability is essential, and a platform such as Spring should help make your code easier to test.

Our philosophy is summarized in Expert One-on-One J2EE Design and Development by Rod Johnson.

We aim that:

- Spring should be a pleasure to use
- Your application code should **not** depend on Spring APIs
- Spring should not compete with good existing solutions, but should foster integration. (For example, JDO, Toplink, and Hibernate are great O/R mapping solutions. We don't need to develop another one.)

首先，Spring 的目标在于让 Java EE 的开发变得更容易，这也就意味着 Spring 框架的使用也应该是容易的。对于开发人员而言，易用性是第一位的。为什么要让 Java EE 开发变得更容易，难道以前的 Java EE 开发很艰难？Spring 究竟是如何让 Java EE 的开发变得更容易的呢？了解 Java EE 开发历史的读者都知道，正如 Rod Johnson 在他的著作 *Expert One-on-One Java EE Design and Development* 中提到的那样，EJB 模型为 Java EE 开发引入了过度的复杂性，这个开发模型对 Java EE 的开发并不友好。有没有更好的开发模型呢？有，就是 POJO！它让 Java 洗净铅华，恢复其自然的风采。使用 POJO 不仅能开发复杂的 Java 企业应用，而且还可以让 Java EE 开发在开发成本、开发周期、可维护性和性能上获得更大优势。对一般的企业应用需求而言，重要的是如何方便地使用应用需要的服务，而不是各种各样的开发模型和模式。虽然这些模式为我们描绘了设计高可靠性分布式应用的美妙场景，但这些场景是不是大多数企业应用开发者所要面对的呢？

世上都说 Java 好，唯有 Spring 忘不了。喜欢 Java，是因为它简洁，不但包含了面向对象的语言特性，同时还可以跨平台，可谓是简洁而又强大。但是，进入到企业应用后，作为门外汉的自己一看到复杂的 EJB 模型就心生畏惧。这时候，我接触到了 Spring，她给人的第一印象就是简洁却又具有丰富的内涵，就像第一次遇到 Java 一样，被她的这种特质深深地吸引了。她降低了企业应用开发的门槛，还原了 POJO 的本色，让我们直接依赖于 Java 语言，直接依赖于面向对象编程，使用无所不在的单元测试来保证代码质量，这样我们就有信心能够开发出高质量的企业应用。

也就是说，我们如何才能让开发既变得容易，又能享受到 Java EE 中提供的各种服务呢？Spring 的目标就是通过自己的努力，让用户体会到这种简单之中的强大。同时，作为应用框架，Spring 不想把自己作为另外一种复杂开发模型的替代，也就是说不是用另一种复杂性去替代现有的复杂性，那是换汤不换药，并不能解决问题。这就意味着需要有新的突破。要解决这个问题，需要降低应用的负载和框架的侵入性，Spring 是怎样做到这一点的呢？

Spring 为我们提供的解决方案就是 IoC 容器和 AOP 支持。作为依赖反转模式的具体实现，IoC

Download at http://www.pin5i.com/

容器很好地降低了框架的侵入性，同时也可以认为依赖反转模式是 Spring 体现出来的核心模式。这些核心模式是软件架构设计中非常重要的因素，比如说，我们常常看到的 MVC 模式就是这样的核心模式。不要小看这些体系结构模式的作用和影响，它们就是框架背后所谓的“道”。有了 IoC 容器和 AOP 的支持，用户的开发方式发生了很大的变化，具体说来，就是可以使用 POJO 来完成开发，对用户来说是简化了，但由于有平台的支持，依然能够实现复杂的企业应用开发。对于依赖反转，在 Spring 中，Java EE 的服务都被抽象到 IoC 容器和 AOP 中并进行了有效地封装，而且因为依赖注入的特性，这些复杂的依赖关系的管理被反转了，它们的管理交给了容器。

Spring 中各个模块的依赖关系可以用简单的 IoC 配置文件进行描述，信息集中并且明了。在使用其他组件服务时，只需要在配置文件中配置这些服务与应用组件的依赖关系。对应用开发而言，只需要了解服务的接口和依赖关系的配置。这样一来又很好地体现了 Spring 的第二个信条：让应用开发对接口编程，而不是对类编程。这样 POJO 使用 Java EE 服务时，可以将对这些服务实现的依赖降到最低，尽可能地降低框架的侵入性。

在处理与现有优秀解决方案的关系时，根据 Spring 的既定策略，它不会与这些第三方的解决方案发生竞争，而是致力于为应用提供使用优秀方案的集成平台。真正地把 Spring 定位在应用平台的地位，使得自己成为一个兼容并包的开放体系的同时，最大程度地降低开发者对 Spring API 的依赖，这是怎样实现的呢？答案还是 IoC 容器和 AOP 技术，也就是说，Spring API 在开发过程中并不是必须使用的。

Download at http://www.pin5i.com/

第2章

Spring Framework 的核心：IoC 容器的实现

朝辞白帝彩云间，千里江陵一日还。
两岸猿声啼不住，轻舟已过万重山。
——【唐】李白《早发白帝城》

2.1 Spring IoC 容器概述

2.1.1 IoC 容器和依赖反转模式

子曰：温故而知新。在这里，我们先简要地回顾一下有关依赖反转的相关概念。我们选取维基百科中关于依赖反转的叙述，把这些文字作为我们理解依赖反转概念的参考。这里不会对这些原理进行学理上的考究，只是希望提供一些有用的信息，以便给读者一些启示。这个模式非常重要，它是 IoC 容器得到广泛应用的基础。

维基百科对“依赖反转”相关概念的叙述

早在 2004 年，Martin Fowler 就提出了“哪些方面的控制被反转了？”这个问题。他得出的结论是：依赖对象的获得被反转了。基于这个结论，他为控制反转创造了一个更好的名字：依赖注入。许多非凡的应用（比 HelloWorld.java 更加优美、更加复杂）都是由两个或多个类通过彼此的合作来实现业务逻辑，这使得每个对象都需要与其合作的对象（也就是它所依赖的对象）的引用。如果这个获取过程要靠自身实现，那么如你所见，这将导致代码高度耦合并且难以测试。

以上的这段话概括了依赖反转的要义，如果合作对象的引用或依赖关系的管理要由具体对象来完成，会导致代码的高度耦合和可测试性降低，这对复杂的面向对象系统的设计是非常不利的。在面向对象系统中，对象封装了数据和对数据的处理，对象的依赖关系常常体现在对数据和方法的依赖上。这些依赖关系可以通过把对象的依赖注入交给框架或 IoC 容器来完成，这种从具体对象手中交出控制的做法是非常有价值的，它可以在解耦代码的同时提高代码的可测试性。极限编程中对单元测试和重构等实践的强调体现了软件开发过程中对质量的承诺，这是软件项目成功的一个重要因素。

依赖控制反转的实现方式有很多种。在 Spring 中，IoC 容器是实现这个模式的载体，它可以在对象生成或初始化时直接将数据注入到对象中，也可以通过将对象引用注入到对象数据域中的方式来注入对方法调用的依赖。这种依赖注入是可以递归的，对象被逐层注入。就此而言，这种方案有一种完整而简洁的美感，它把对象的依赖关系有序地建立起来，简化了对象依赖关系的管理，在很大程度上简化了面向对象系统的复杂性。

Download at http://www.pin5i.com/

关于如何反转对依赖的控制，把控制权从具体业务对象手中转交到平台或者框架中，是解决面向对象系统设计复杂性和提高面向对象系统可测试性的一个有效的解决方案。它促进了 IoC 设计模式的发展，是 IoC 容器要解决的核心问题。同时，也是产品化的 IoC 容器出现的推动力。

注意　IoC 亦称为"依赖倒置原理"（Dependency Inversion Principle），几乎所有框架都使用了倒置注入（Martin Fowler）技巧，是 IoC 原理的一项应用。SmallTalk、C++、Java 或.NET 等面向对象语言的程序员已使用了这些原理。控制反转是 Spring 框架的核心。

IoC 原理的应用在不同的语言中有许多实现，比如 SmallTalk、C++、Java 等。在同一语言的实现中也会有多个具体的产品，Spring 是 Java 语言实现中最著名的一个。同时，IoC 也是 Spring 框架要解决的核心问题。

注意　应用控制反转后，当对象被创建时，由一个调控系统内的所有对象的外界实体将其所依赖的对象的引用传递给它。也就是说，依赖被注入到对象中。所以，控制反转是关于一个对象如何获取它所依赖的对象的引用的，在这里，反转指的是责任的反转。

我们可以认为上面提到的调控系统是应用平台，或者更具体地说是 IoC 容器。通过使用 IoC 容器，对象依赖关系的管理被反转了，转到 IoC 容器中来了，对象之间的相互依赖关系由 IoC 容器进行管理，并由容器完成对象的注入。这样就在很大程度上简化了应用的开发，把应用从复杂的对象依赖关系管理中解放出来。简单地说，因为很多对象的依赖关系的建立和维护并不需要和系统运行状态有很强的关联性，所以可以把我们在面向对象编程中常常需要执行的诸如新建对象、给对象引用赋值等操作交由容器统一完成。这样一来，这些散落在不同代码中的功能相同的部分就集中成为容器的一部分，也就是成为面向对象系统的基础设施的一部分。

如果对面向对象系统中的对象进行简单地分类，会发现除了一部分是数据对象外，其他有很大一部分对象都是用来处理数据的。这些对象并不会经常发生变化，是系统中基础的部分。在很多情况下，这些对象在系统中以单件的形式存在就可以满足应用的需求，而且它们也不常涉及数据和状态共享的问题。如果涉及数据共享方面的问题，需要在这些单件的基础上做进一步的处理。

同时，这些对象之间的相互依赖关系也是比较稳定的，一般不会随着应用的运行状态的改变而改变。这些特性使得这些对象非常适合由 IoC 容器来管理，虽然它们存在于应用系统中，但是应用系统并不承担管理这些对象的责任，而是通过依赖反转把责任交给了容器（或者说平台）。了解了这些背景，Spring IoC 容器的原理也就不难理解了。在原理的具体实现上，Spring 有着自己的独特思路、实现技巧和丰富的产品特性。关于这些原理的实现，下面会进行详细的分析。

第 1 章中，我们已经对建立本地源代码环境做了简要的介绍，该源代码环境是我们分析 Spring 原理前要做的重要准备工作。同时，我们还需要针对 IoC 容器做一些额外的事情：根据 Spring 3.0 的源代码组织特点，每个模块作为独立的 Eclipse 项目存在，所以现在需要在 Eclipse 中建立与 IoC 容器和上下文相关的代码项目。这样就可以方便地使用 Eclipse 的代码分析工具来对相关模块的实现进行分析。这个额外的准备过程在分析其他模块时也是需要的，所以这里会做一个说明。

准备过程如图 2-1 所示，打开 Eclipse，依次选择 File→Import→General→Existing Projects into Workspace，然后再选择 `org.springframework.beans` 和 `org.springframework.context`

Download at http://www.pin5i.com/

两个目录，并将其导入到 Eclipse 本地环境中。这时即可看到在 Package Explorer View 中的 Spring IoC 容器的源代码项目。

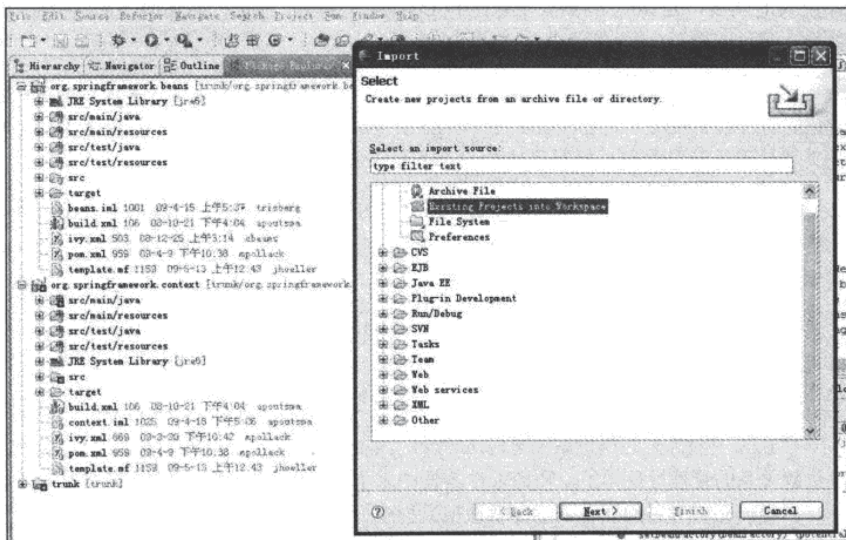


图 2-1　打开 IoC 容器的源代码包

2.1.2 Spring 的 IoC 容器系列

IoC 容器为开发者管理对象之间的依赖关系提供了很多便利和基础服务，有许多 IoC 容器供开发者选择，SpringFramework 的 IoC 核心就是其中的一个，它是开源的。那具体什么是 IoC 容器呢？它在 Spring 框架中到底长什么样？其实对 IoC 容器的使用者来说，我们经常接触到的 BeanFactory 和 ApplicationContext 都可以看成是容器的具体表现形式。我们通常所说的 IoC 容器，如果深入到 Spring 的实现去看，会发现 IoC 容器实际上代表着一系列功能各异的容器产品，只是容器的功能有大有小，有各自的特点。我们举水桶为例子，在商店中出售的水桶有大有小，制作材料也各不相同，有金属的、塑料的等，总之是各式各样，但只要能装水，具备水桶的基本特性，那就可以作为水桶来出售，来让用户使用。这在 Spring 中也是一样，Spring 有各式各样的 IoC 容器的实现供用户选择和使用。使用什么样的容器完全取决于用户的需要，但在使用之前如果能够了解容器的基本情况，那对容器的使用是非常有帮助的，就像我们在购买商品前对商品进行考察和挑选那样。图 2-2 展示了这个容器系列的概况。

就像商品需要有产品规格说明一样，同样，作为 IoC 容器，也需要为它的具体实现指定基本的功能规范，这个功能规范的设计表现为接口类 BeanFactory，它体现了 Spring 为提供给用户使用的 IoC 容器所设定的最基本功能规范。还是举前面我们说的百货商店出售的水桶为例子，如果把 IoC 容器看成一个水桶，那么这个 BeanFactory 就定义了可以作为水桶的基本功能，比如至少能

Download at http://www.pin5i.com/

装水，有个提手什么的。满足了基本的功能，为了不同场合的需要，水桶的生产厂家还在这个基础上为用户设计了其他各式各样的水桶产品，来满足不同的用户需求。这些水桶会提供更丰富的功能，有简约型的，有豪华型的，等等。但是，不管什么水桶，它都需要有一项最基本的功能：能够装水。那对 Spring 的具体 IoC 容器实现来说，它需要满足的基本特性是什么呢？它需要满足 BeanFactory 这个基本的接口定义，所以在图 2-2 中可以看到，这个 BeanFactory 接口在继承体系中的地位，它是作为一个最基本的接口类出现在 Spring 的 IoC 容器体系中的。

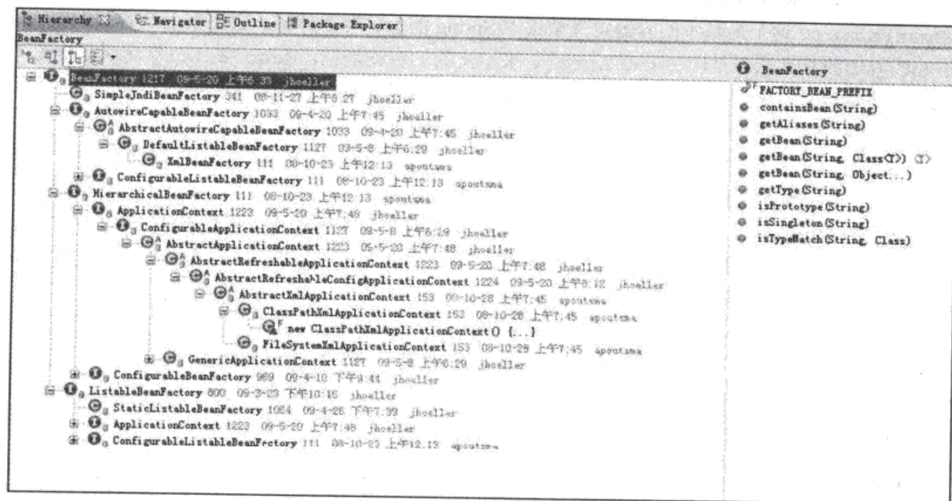


图 2-2　Spring 的 IoC 容器系列概况

在这些 Spring 提供的基本 IoC 容器的接口定义和实现的基础上，Spring 通过定义 BeanDefinition 来管理基于 Spring 的应用中的各种对象以及它们之间的相互依赖关系。BeanDefinition 抽象了我们对 Bean 的定义，是让容器起作用的主要数据类型。我们都知道，在计算机的世界里，所有的功能都是建立在用数据对现实进行抽象的基础上完成的。IoC 容器是用来管理对象依赖关系的，对 IoC 容器来说，BeanDefinition 就是对依赖反转模式中管理的对象依赖关系的数据抽象，也是容器实现依赖反转功能的核心数据结构，依赖反转功能都是围绕对这个 BeanDefinition 的处理上完成的。这些 BeanDefinition 就像是容器里装的水，有了这些基本数据，容器才能够发挥作用。在下面的分析中，BeanDefinition 的上镜次数会很多，我们在这里先简单地打个招呼。

同时，在使用 IoC 容器时，了解 BeanFactory 和 ApplicationContext 之间的区别对我们理解和使用 IoC 容器也是比较重要的。弄清楚了这两种重要容器之间的区别和联系，意味着我们具备辨别容器系列中不同容器产品的能力。还有一个好处就是，如果需要定制特定功能的容器实现，也能比较方便地在容器系列中找到一款恰当的产品作为参考。

Download at http://www.pin5i.com/

2.2 IoC 容器系列的实现：BeanFactory 和 ApplicationContext

2.2.1 BeanFactory 对 IoC 容器的功能定义

从前面的介绍，我们知道 BeanFactory 定义了 IoC 容器的基本功能规范，所以，下面我们就从 BeanFactory 这个最基本的容器定义来进入 Spring 的 IoC 容器体系，去了解 IoC 容器的实现原理。IoC 容器的基本接口是由 BeanFactory 来定义的，也就是说，BeanFactory 定义了 IoC 容器的最基本的形式，并且提供了 IoC 容器所应该遵守的最基本的服务契约。同时，这也是我们使用 IoC 容器所应遵守的最底层和最基本的编程规范，这些接口定义勾画出了 IoC 的基本轮廓。很显然，在 Spring 的代码实现中，BeanFactory 只是一个接口类，并没有给出容器的具体实现，而我们在图 2-2 中看到的各种具体类，比如 DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext 等都可以看成是容器的附加了某种功能的具体实现，也就是容器体系中的具体容器产品。下面我们来看看 BeanFactory 是怎样定义 IoC 容器的基本接口的。下面介绍这个基本接口为用户提供的基本功能。

用户使用容器时，可以使用转义符“&”来得到 FactoryBean 本身，用来区分通过容器来获取 FactoryBean 产生的对象和获取 FactoryBean 本身。举例来说，如果 myJndiObject 是一个 FactoryBean，那么使用&myJndiObject 得到的是 FactoryBean，而不是 myJndiObject 这个 FactoryBean 产生出来的对象。

注意 理解上面这段话需要很好地区分 FactoryBean 和 BeanFactory 这两个在 Spring 中使用频率很高的类，它们在拼写上非常相似。一个是 Factory，也就是 IoC 容器或对象工厂；一个是 Bean。在 Spring 中，所有 Bean 都是由 BeanFactory（也就是 IoC 容器）来进行管理的。但对 FactoryBean 而言，这个 Bean 不是简单的 Bean，而是一个能产生或者修饰对象生成的工厂 Bean，它的实现与设计模式中的工厂模式和修饰器模式类似。

BeanFactory 接口设计了 getBean 方法，这个方法是使用 IoC 容器 API 的主要方法，通过这个方法，可以取得 IoC 容器中管理的 Bean，Bean 的取得是通过指定名字来进行索引的。如果需要在获取 Bean 时对 Bean 的类型进行检查，BeanFactory 接口定义了带有参数的 getBean 方法，这个方法的使用与 getBean 方法类似，不同的是增加了对 Bean 检索的类型的要求。

用户可以通过 BeanFactory 接口方法 getBean 来使用 Bean 名字，从而当获取 Bean 时，如果需要获取的 Bean 是 prototype 类型的，用户还可以为这个 prototype 类型的 Bean 生成指定构造函数的对应参数。这使得在一定程度上可以控制生成 prototype 类型的 Bean。有了 BeanFactory 的定义，用户可以执行以下操作：

- 通过接口方法 containsBean 让用户能够判断容器是否含有指定名字的 Bean。
- 通过接口方法 isSingleton 来查询指定了名字的 Bean 是否是 Singleton 类型的 Bean。对于 Singleton 属性，用户可以在 BeanDefinition 中指定。
- 通过接口方法 isPrototype 来查询指定了名字的 Bean 是否是 prototype 类型的。与 Singleton 属性一样，这个属性也可以由用户在 BeanDefinition 中指定。

Download at http://www.pin5i.com/

- 通过接口方法 isTypeMatch 来查询指定了名字的 Bean 的 Class 类型是否是特定的 Class 类型。这个 Class 类型可以由用户来指定。
- 通过接口方法 getType 来查询指定了名字的 Bean 的 Class 类型。
- 通过接口方法 getAliases 来查询指定了名字的 Bean 的所有别名，这些别名都是用户在 BeanDefinition 中定义的。

这些定义的接口方法勾画出了 IoC 容器的基本特性，因为 BeanFactory 接口定义了 IoC 容器，所以下面给出它定义的全部内容来让大家参考，如代码清单 2-1 所示。

代码清单 2-1 BeanFactory 接口

```
public interface BeanFactory {
 /**
  * Used to dereference a {@link FactoryBean} instance and distinguish it from
  * beans <i>created</i> by the FactoryBean. For example, if the bean named
  * <code>myJndiObject</code> is a FactoryBean, getting <code>&myJndiObject </code>
  * will return the factory, not the instance returned by the factory.
  */
 String FACTORY_BEAN_PREFIX = "&";
 /**
  * Return an instance, which may be shared or independent, of the specified bean.
  * <p>This method allows a Spring BeanFactory to be used as a replacement for the
  * Singleton or Prototype design pattern. Callers may retain references to
  * returned objects in the case of Singleton beans.
  * <p>Translates aliases back to the corresponding canonical bean name.
  * Will ask the parent factory if the bean cannot be found in this factory instance.
  * */
 Object getBean(String name) throws BeansException;
 /**
  * Return an instance, which may be shared or independent, of the specified bean.
  * <p>Behaves the same as {@link #getBean(String)}, but provides a measure of type
  * safety by throwing a BeanNotOfRequiredTypeException if the bean is not of the
  * required type. This means that ClassCastException can't be thrown on casting
  * the result correctly, as can happen with {@link #getBean(String)}.
  * <p>Translates aliases back to the corresponding canonical bean name.
  * Will ask the parent factory if the bean cannot be found in this factory instance.
  *  */
 <T> T getBean(String name, Class<T> requiredType) throws BeansException;
 /**
  * Return an instance, which may be shared or independent, of the specified bean.
  * <p>Allows for specifying explicit constructor arguments / factory method arguments,
  * overriding the specified default arguments (if any) in the bean definition.
  *  */
 Object getBean(String name, Object... args) throws BeansException;
 /**
  * Does this bean factory contain a bean with the given name? More specifically,
  * is {@link #getBean} able to obtain a bean instance for the given name?
  * <p>Translates aliases back to the corresponding canonical bean name.
  * Will ask the parent factory if the bean cannot be found in this factory instance.
  */
 boolean containsBean(String name);
 /**
  * Is this bean a shared singleton? That is, will {@link #getBean} always
  * return the same instance?
  * <p>Note: This method returning <code>false</code> does not clearly indicate
  * independent instances. It indicates non-singleton instances, which may correspond
  * to a scoped bean as well. Use the {@link #isPrototype} operation to explicitly
```

Download at http://www.pin5i.com/

```
	 * check for independent instances.
	 * <p>Translates aliases back to the corresponding canonical bean name.
	 * Will ask the parent factory if the bean cannot be found in this factory instance.
	 */
	boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
	/**
	 * Is this bean a prototype? That is, will {@link #getBean} always return
	 * independent instances?
	 * <p>Note: This method returning <code>false</code> does not clearly indicate
	 * a singleton object. It indicates non-independent instances, which may correspond
	 * to a scoped bean as well. Use the {@link #isSingleton} operation to explicitly
	 * check for a shared singleton instance.
	 * <p>Translates aliases back to the corresponding canonical bean name.
	 * Will ask the parent factory if the bean cannot be found in this factory instance.
	 */
	boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
	/**
	 * Check whether the bean with the given name matches the specified type.
	 * More specifically, check whether a {@link #getBean} call for the given name
	 * would return an object that is assignable to the specified target type.
	 * <p>Translates aliases back to the corresponding canonical bean name.
	 * Will ask the parent factory if the bean cannot be found in this factory instance.
	 */
	boolean isTypeMatch(String name, Class targetType) throws NoSuchBeanDefinitionException;
	/**
	 * Determine the type of the bean with the given name. More specifically,
	 * determine the type of object that {@link #getBean} would return for the given name.
	 * <p>For a {@link FactoryBean}, return the type of object that the FactoryBean creates,
	 * as exposed by {@link FactoryBean#getObjectType()}.
	 * <p>Translates aliases back to the corresponding canonical bean name.
	 * Will ask the parent factory if the bean cannot be found in this factory instance.
	 */
	Class getType(String name) throws NoSuchBeanDefinitionException;
	/**
	 * Return the aliases for the given bean name, if any.
	 * All of those aliases point to the same bean when used in a {@link #getBean} call.
	 * <p>If the given name is an alias, the corresponding original bean name
	 * and other aliases (if any) will be returned, with the original bean name
	 * being the first element in the array.
	 * <p>Will ask the parent factory if the bean cannot be found in this factory instance.
	 */
	String[] getAliases(String name);
}
```

2.2.2 IoC 容器 XmlBeanFactory 的工作原理

这个 BeanFactory 接口提供了使用 IoC 容器的规范。在这个基础上，Spring 还提供了符合这个 IoC 容器接口的一系列容器的实现供开发人员使用。例如，在图 2-2 中，我们可以看到 BeanFactory 的相关部分的实现。为简单起见，我们浏览一下图 2-2 的 BeanFactory 的继承体系，注意 AutowireCapableBeanFactory→AbstractAutowireCapableBeanFactory→DefaultListableBeanFactory→XmlBeanFactory IoC 容器的实现系列。

我们从这个容器系列的最底层实现 XmlBeanFactory 开始，这个容器的实现与我们在 Spring 应用中用到的那些上下文相比，有一个非常明显的特点，它只提供了最基本的 IoC 容器的功能。从它的名字中可以看出，这个 IoC 容器可以读取以 XML 形式定义的 BeanDefinition。理解这一

Download at http://www.pin5i.com/

点有助于理解 ApplicationContext 与基本的 BeanFactory 之间的区别和联系。我们可以认为直接的 BeanFactory 实现是 IoC 容器的基本形式，而各种 ApplicationContext 的实现是 IoC 容器的高级表现形式。关于 ApplicationContext 的分析，以及它与 BeanFactory 相比的增强特性都会在下面进行详细的分析。

让我们回顾一下这个继承体系，从中可以清楚地看到它们之间的联系，它们都是 IoC 容器系列的组成部分。在设计这个容器系列时，我们可以从继承体系的发展上看到 IoC 容器各项功能的实现过程。如果要扩展自己的容器产品，建议读者最好在这个继承体系中检验一下，看看 Spring 是不是已经提供了现成的或相近的容器实现供我们参考。下面就从我们比较熟悉的 XmlBeanFactory 的实现入手进行分析，来看看一个基本的 IoC 容器是怎样实现的。

如果仔细阅读 XmlBeanFactory 的源码，在一开始的注释里面已经对 XmlBeanFactory 的功能做了简要的说明，从代码的注释还可以看到，这是 Rod Johnson 在 2001 年就写下的代码，可见这个类应该是 Spring 的元老类了。它是继承 DefaultListableBeanFactory 这个类的，而且它非常重要，在以后的分析中这个类会经常用到。我们会看到这个 DefaultListableBeanFactory 实际上包含了 IoC 容器的重要功能，也是在很多地方都会用到的容器系列中的一个基本产品。

从名字上就可以看出来，在 Spring 中，实际上是把它作为一个默认的完整功能的 IoC 容器来使用的。XmlBeanFactory 在继承了 DefaultListableBeanFactory 容器的功能的同时，给 DefaultListableBeanFactory 增加的功能很容易从 XmlBeanFactory 的名字上猜到。它是一个与 XML 相关的 BeanFactory，也就是说它可以读取以 XML 文件方式定义的 BeanDefinition 的一个 IoC 容器。

如果说 XmlBeanFactory 是一个可以读取 XML 文件方式定义的 BeanDefinition 的 IoC 容器，那么这些实现 XML 读取的功能是怎样实现的呢？对这些 XML 文件定义信息的处理并不是由 XmlBeanFactory 来直接处理的。在 XmlBeanFactory 中，初始化了一个 XmlBeanDefinitionReader 对象，有了这个 Reader 对象，那些以 XML 的方式定义的 BeanDefinition 就有了处理的地方。我们可以看到，对这些 XML 形式的信息的处理实际上是由这个 XmlBeanDefinitionReader 来完成的。

构造 XmlBeanFactory 这个 IoC 容器时，需要指定 BeanDefinition 的信息来源，而这个信息来源需要封装成 Spring 中的 Resource 类来给出。Resource 是 Spring 用来封装 IO 操作的类。比如，我们的 BeanDefinition 信息是以 xml 文件形式存在的，那么可以使用像 ClassPathResource res = new ClassPathResource("beans.xml"); 这样具体的 ClassPathResource 来构造需要的 Resource，然后作为构造参数传递给 XmlBeanFactory 构造函数。这样，IoC 容器就可以方便地定位到需要的 BeanDefinition 信息来对 Bean 完成容器的初始化和依赖注入过程。

XmlBeanFactory 的功能是建立在 DefaultListableBeanFactory 这个基本容器的基础上的，在这个基本容器的基础上实现了其他诸如 XML 读取的附加功能。对于这些功能的实现原理，看一看 XmlBeanFactory 的代码实现就能很容易地理解。如代码清单 2-2 所示，在 XmlBeanFactory 构造方法中需要得到 Resource 对象。对 XmlBeanDefinitionReader 对象的初始化，以及使用这个对象来完成 loadBeanDefinitions 的调用，就是这个调用启动了从

Download at http://www.pin5i.com/

Resource 中载入 BeanDefinitions 的过程，loadBeanDefinitions 同时也是 IoC 容器初始化的重要组成部分。

代码清单 2-2 XmlBeanFactory 的实现

```
public class XmlBeanFactory extends DefaultListableBeanFactory {
private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);
 public XmlBeanFactory(Resource resource) throws BeansException {
     this(resource, null);
 }
 public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws
     BeansException {
     super(parentBeanFactory);
     this.reader.loadBeanDefinitions(resource);
 }
}
```

我们看到 XmlBeanFactory 使用了 DefaultListableBeanFactory 作为基类，DefaultListableBeanFactory 是很重要的一个 IoC 实现，在其他 IoC 容器中，比如 ApplicationContext，其实现的基本原理和 XmlBeanFactory 一样，也是通过持有或者扩展 DefaultListableBeanFactory 来获得基本的 IoC 容器的功能的。

参考 XmlBeanFactory 的实现，我们以编程的方式使用 DefaultListableBeanFactory，从中我们可以看到 IoC 容器使用的一些基本过程。尽管我们在应用中使用 IoC 容器时很少会使用这样原始的方式，但是了解一下这个基本的过程，对我们了解 IoC 容器的工作原理却是非常有帮助的。因为这个编程式使用容器的过程很清楚地揭示了在 IoC 容器实现中的那些关键的类（比如 Resource、DefaultListableBeanFactory 以及 BeanDefinitionReader）之间的相互关系，例如它们是如何把 IoC 容器的功能解耦的，又是如何结合在一起为 IoC 容器服务的，等等。在代码清单 2-3 中可以看到编程式使用 IoC 容器的过程。

代码清单 2-3 编程式使用 IoC 容器

```
ClassPathResource res = new ClassPathResource("beans.xml"); DefaultListableBeanFactory
    factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(res);
```

这样，我们就可以通过 factory 对象来使用 DefaultListableBeanFactory 这个 IoC 容器了。在使用 IoC 容器时，需要如下几个步骤：

1）创建 IoC 配置文件的抽象资源，这个抽象资源包含了 BeanDefinition 的定义信息。

2）创建一个 BeanFactory，这里使用 DefaultListableBeanFactory。

3）创建一个载入 BeanDefinition 的读取器，这里使用 XmlBeanDefinitionReader 来载入 XML 文件形式的 BeanDefinition，通过一个回调配置给 BeanFactory。

4）从定义好的资源位置读入配置信息，具体的解析过程由 XmlBeanDefinitionReader 来完成。完成整个载入和注册 Bean 定义之后，需要的 IoC 容器就建立起来了。这个时候 IoC 容器就可以直接使用了。

2.2.3 ApplicationContext 的特点

我们了解了 IoC 容器建立的基本步骤。现在可以很方便地通过编程的方式来手工控制这些配置

Download at http://www.pin5i.com/

和容器的建立过程了。但是，在 Spring 中系统已经为用户提供了许多已经定义好的容器实现，而不需要开发人员事必躬亲。相比那些简单拓展 BeanFactory 的基本 IoC 容器，开发人员常用的 ApplicationContext 除了能够提供在上面看到的容器的基本功能外，还为用户提供了以下的附加服务，可以让客户更方便地使用。所以说，ApplicationContext 是一个高级形态意义的 IoC 容器，如图 2-3 所示，可以看到 ApplicationContext 在 BeanFactory 的基础上添加的附加功能，这些功能为 ApplicationContext 提供了以下 BeanFactory 不具备的新特性。

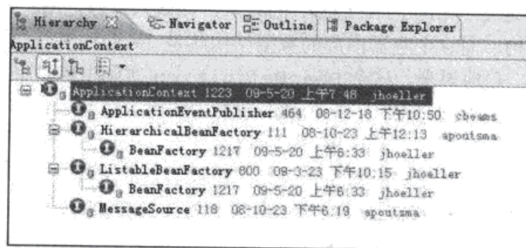


图 2-3 ApplicationContext 的接口关系

- 支持不同的信息源。我们看到 ApplicationContext 扩展了 MessageSource 接口，这些信息源的扩展功能可以支持国际化的实现，为开发多语言版本的应用提供服务。
- 访问资源。体现在对 ResourceLoader 和 Resource 的支持上，这样我们可以从不同地方得到 Bean 定义资源。这种抽象使用户程序可以灵活地定义 Bean 定义信息，尤其是从不同的 IO 途径得到 Bean 定义信息。这在接口关系上看不出来，一般来说，具体 ApplicationContext 都是继承了 DefaultResourceLoader 的子类。因为 DefaultResourceLoader 是 AbstractApplicationContext 的基类，关于 Resource 在 IoC 容器中的使用，在 2.3 节中有详细的讲解。
- 支持应用事件。继承了接口 ApplicationEventPublisher，这样在上下文中引入了事件机制。这些事件和 Bean 的生命周期的结合为 Bean 的管理提供了便利。
- 在 ApplicationContext 中提供的附加服务。这些服务使得基本 IoC 容器的功能更丰富。因为具备了这些丰富的附加功能，使得 ApplicationContext 与简单的 BeanFactory 相比，对它的使用是一种面向框架的使用风格，所以一般建议在开发应用时使用 ApplicationContext 作为 IoC 容器的基本形式。

2.3 IoC 容器的初始化

IoC 容器的初始化包括 BeanDefinition 的 Resouce 定位、载入和注册这三个基本的过程。在前面的编程式地使用 DefaultListableBeanFactory 中，我们可以看到定位和载入过程的接口调用。这里将详细分析这三个过程的实现。值得注意的是，Spring 在实现中是把这三个过程分开并使用不同的模块来完成的，这样可以让用户更加灵活地对这三个过程进行剪裁和扩展，定义出最适合自己的 IoC 容器的初始化过程。

Download at http://www.pin5i.com/

BeanDefinition 的资源定位由 ResourceLoader 通过统一的 Resource 接口来完成，这个 Resource 对各种形式的 BeanDefinition 的使用提供了统一接口。对于这些 BeanDefinition 的存在形式，相信大家都不会感到陌生。比如说，在文件系统中的 Bean 定义信息可以使用 FileSystemResource 来进行抽象；在类路径中可以使用前面提到的 ClassPathResource 来使用，等等。这个过程类似于容器寻找数据的过程，就像用水桶装水先要把水找到一样。

第二个关键的部分是 BeanDefinition 的载入，该载入过程把用户定义好的 Bean 表示成 IoC 容器内部的数据结构，而这个容器内部的数据结构就是 BeanDefinition，下面可以看到这个数据结构的详细定义。总地说来，这个 BeanDefinition 实际上就是 POJO 对象在 IoC 容器中的抽象，这个 BeanDefinition 定义了一系列的数据来使得 IoC 容器能够方便地对 POJO 对象也就是 Spring 的 Bean 进行管理。即 BeanDefinition 就是 Spring 的领域对象。下面我们会对这个载入的过程进行详细的分析，便于大家对整个过程有比较清楚的了解。

第三个过程是向 IoC 容器注册这些 BeanDefinition 的过程。这个过程是通过调用 BeanDefinitionRegistry 接口的实现来完成的，这个注册过程把载入过程中解析得到的 BeanDefinition 向 IoC 容器进行注册。可以看到，在 IoC 容器内部，是通过使用一个 HashMap 来持有这些 BeanDefinition 数据的。

值得注意的是，IoC 容器和上下文的初始化一般不包含 Bean 依赖注入的实现。一般而言，依赖注入发生在应用第一次向容器通过 getBean 索取 Bean 时。但有一个例外值得注意，在使用 IoC 容器时有一个预实例化的配置，这个预实例化是可以配置的，具体来说可以通过在 Bean 定义信息中的 lazyinit 属性来设定；有了这个预实例化的特性，用户可以对容器初始化过程作一个微小的控制；从而改变这个被设置了 lazyinit 属性的 Bean 的依赖注入的发生，使得这个 Bean 的依赖注入在 IoC 容器初始化时就预先完成了。有了以上的一个大概的轮廓，下面就详细地看一看在 IoC 容器的初始化过程中，BeanDefinition 的资源定位、载入和解析过程是怎么实现的。

2.3.1 BeanDefinition 的 Resource 定位

以编程的方式使用 DefaultListableBeanFactory 时，我们可以看到，首先定义一个 Resource 来定位容器使用的 BeanDefinition。这时使用的是 ClassPathResource，意味着 Spring 会在类路径中寻找以文件形式存在的 BeanDefinition 信息。

```
ClassPathResource res = new ClassPathResource("beans.xml");
```

这个定义的 Resource 并不能让 DefaultListableBeanFactory 直接使用，Spring 是通过 BeanDefinitionReader 来对这些信息进行处理的。在这里，我们也可以看到使用 ApplicationContext 相对于直接使用 DefaultListableBeanFactory 的好处。因为在 ApplicationContext 中，Spring 已经为我们提供了一系列加载不同 Resource 的读取器的实现，而 DefaultListableBeanFactory 只是一个纯粹的 IoC 容器，需要为它配置特定的读取器才能完成这些功能。当然，有利就有弊，使用 DefaultListableBeanFactory 这种更底层的容器，却能提高我们定制 IoC 容器的灵活性。

回到我们经常使用的 ApplicationContext 上来，例如 FileSystemXmlApplication-

Download at http://www.pin5i.com/

Context、ClassPathXmlApplicationContext 以及 XmlWebApplicationContext 等。简单地从这些类的名字上分析，可以清楚地看到它们可以提供哪些不同的 Resource 读入功能，比如 FileSystemXmlApplicationContext 可以从文件系统载入 Resource，ClassPathXmlApplicationContext 可以从 Class Path 载入 Resource，XmlWebApplicationContext 可以在 Web 容器中载入 Resource，等等。

下面以 FileSystemXmlApplicationContext 为例，通过分析这个 ApplicationContext 的实现来看看它是怎样完成这个 Resource 定位过程的。作为辅助，我们可以在图 2-4 中看到相应的 ApplicationContext 继承体系。

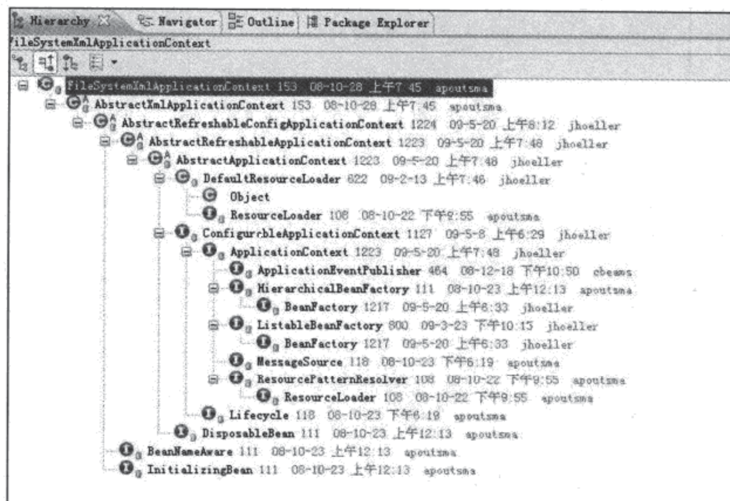


图 2-4 FileSystemXmlApplicationContext 的继承关系

从图 2-4 中可以看到，这个 FileSystemXmlApplicationContext 已经通过继承 AbstractApplicationContext 具备了 ResourceLoader 读入以 Resource 定义的 BeanDefinition 的能力，因为 AbstractApplicationContext 的基类是 DefaultResourceLoader。下面看看 FileSystemXmlApplicationContext 的具体实现，如代码清单 2-4 所示。

代码清单 2-4 FileSystemXmlApplicationContext 的实现

```
public class FileSystemXmlApplicationContext extends AbstractXmlApplicationContext {
 public FileSystemXmlApplicationContext() {
 }
 public FileSystemXmlApplicationContext(ApplicationContext parent) {
     super(parent);
 }
 //这个构造函数的 configLocation 包含的是 BeanDefinition 所在的文件路径。
 public FileSystemXmlApplicationContext(String configLocation) throws BeansException {
     this(new String[] {configLocation}, true, null);
 }
 //这个构造函数允许 configLocation 包含多个 BeanDefinition 的文件路径。
```

Download at http://www.pin5i.com/

```
    public FileSystemXmlApplicationContext(String[] configLocations) throws BeansException {
        this(configLocations, true, null);
    }
    /**
    *这个构造函数在允许 configLocation 包含多个 BeanDefinition 的文件路径的同时，
    *还允许指定自己的双亲 IoC 容器。
    */
    public FileSystemXmlApplicationContext(String[] configLocations, ApplicationContext
        parent) throws BeansException {
        this(configLocations, true, parent);
    }
    public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh)
        throws BeansException {
        this(configLocations, refresh, null);
    }
    /**
    *在对象的初始化过程中，调用 refresh 函数载入 BeanDefinition，这个 refresh
    *启动了 BeanDefinition 的载入过程，我们会在下面进行详细分析。
    */
    public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh,
        ApplicationContext parent)
            throws BeansException {
        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
    }
    /**
    *这是应用于文件系统中 Resource 的实现，通过构造一个 FileSystemResource 来
    *得到一个在文件系统中定位的 BeanDefinition。
    */
    /**
    *这个 getResourceByPath 是在 BeanDefinitionReader 的 loadBeanDefintion 中被调用的。
    *loadBeanDefintion 采用了模板模式，具体的定位实现实际上是由各个子类完成的。
    */
    protected Resource getResourceByPath(String path) {
        if (path != null && path.startsWith("/")) {
            path = path.substring(1);
        }
        return new FileSystemResource(path);
    }
}
```

在 FileSystemApplicationContext 中，我们可以看到实现了两个部分的功能，一部分是在构造函数中，对 configuration 进行处理，使得所有在配置在文件系统中的 XML 文件方式的 BeanDefnition 都能够得到有效的处理，比如实现了 getResourceByPath 方法，这个方法是一个模板方法，是为读取 Resource 服务的。对于 IoC 容器功能的实现，这里没有涉及，因为它继承了 AbstractXmlApplicationContext，关于 IoC 容器功能相关的实现，都是在 FileSystemXmlApplicationContext 中完成的，但是在构造函数中通过 refresh 来启动了 IoC 容器的初始化，这个 refresh 方法非常重要，也是我们以后分析容器初始化过程实现的一个重要入口。

Download at http://www.pin5i.com/

注意 FileSystemApplicationContext 是一个支持 XML 定义 BeanDefinition 的 ApplicationContext，并且可以指定以文件形式的 BeanDefinition 的读入，这些文件可以使用文件路径和 URL 定义来表示。在测试环境和独立应用环境中，这个 ApplicationContext 是非常的有用的。

根据图 2-5 的调用关系分析，我们可以清楚地看到整个 BeanDefinition 资源定位的过程。这个对 BeanDefinition 资源定位的过程，最初是由 refresh 来触发的，这个 refresh 的调用是在 FileSystemXmlBeanFactory 的构造函数中启动的。

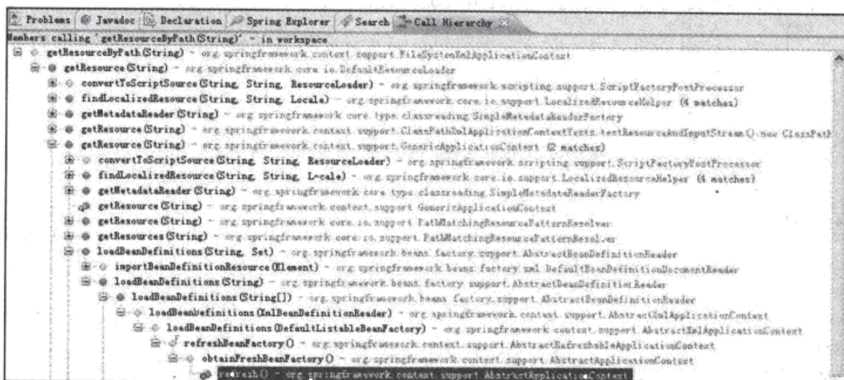


图 2-5 getResourceByPath 的调用关系

大家看了上面的调用过程可能会比较好奇，这个 FileSystemXmlApplicationContext 在什么地方定义了 BeanDefinition 的读入器 BeanDefinitionReader，从而完成 BeanDefinition 信息的读入呢？在前面分析过，在 IoC 容器的初始化过程中，BeanDefinition 资源的定位、读入和注册过程是分开进行的，这也是解耦的一个体现。关于这个读入器的配置，可以到 FileSystemXmlApplicationContext 的基类 AbstractRefreshableApplicationContext 中看看它是怎样实现的。

我们重点看看 AbstractRefreshableApplicationContext 的 refreshBeanFactory 方法的实现，这个 refreshBeanFactory 被 FileSystemXmlApplicationContext 构造函数中的 refresh 调用。在这个方法里，通过 createBeanFactroy 构建了一个 IoC 容器供 ApplicationContext 使用。这个 IoC 容器就是我们前面提到过的 DefaultListableBeanFactory，同时，它启动了 loadBeanDefinitions 来载入 BeanDefinition，这个过程和我们前面看到的编程式的使用 IoC 容器（XmlBeanFactory）的过程非常类似。

从代码清单 2-4 中可以看到，在初始化 FileSystmXmlApplicationContext 的过程中，通过 IoC 容器的初始化的 refresh 来启动整个调用，使用的 IoC 容器是 DefultListableBean Factory。具体的资源载入在 XmlBeanDefinitionReader 读入 BeanDefinition 时完成，在 XmlBeanDefinitionReader 的基类 AbstractBeanDefinitionReader 中可以看到这个

Download at http://www.pin5i.com/

载入过程的具体实现。对载入过程的启动，可以在 AbstractRefreshableApplicationContext 的 loadBeanDefinitions 方法中看到，如代码清单 2-5 所示。

代码清单 2-5 AbstractRefreshableApplicationContext 对容器的初始化

```
protected final void refreshBeanFactory() throws BeansException {
    //这里判断，如果已经建立了 BeanFactory，则销毁并关闭该 BeanFactory。
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    //这里是创建并设置持有的 DefaultListableBeanFactor 的地方。
    //同时调用 loadBeanDefinitions 再载入 BeanDefinitione 的信息。
    try {
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing XML document
    for " + getDisplayName(), ex);
    }
}
/**
*这就是在上下文中创建 DefaultListableBeanFactory 的地方，而 getInternalParentBeanFactory() 的
*具体实现可以参看 AbstractApplicationContext 中的实现，会根据容器已有的双亲 IoC 容器的信息来
*生成 DefaultListableBeanFactory 的双亲 IoC 容器。
*/
protected DefaultListableBeanFactory createBeanFactory() {
return new DefaultListableBeanFactory(getInternalParentBeanFactory());
}
/**
*这里是使用 BeanDefinitionReader 载入 Bean 定义的地方，因为允许有多种载入方式，
*虽然用得最多的是 XML 定义的形式，这里通过一个抽象函数把具体的实现委托给子类来完成。
*/
protected abstract void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
    throws IOException, BeansException;
public int loadBeanDefinitions(String location, Set actualResources) throws
    BeanDefinitionStoreException {
    //这里取得 ResourceLoader,使用的是 DefaultResourceLoader。
    ResourceLoader resourceLoader = getResourceLoader();
    if (resourceLoader == null) {
        throw new BeanDefinitionStoreException(
                "Cannot import bean definitions from location [" + location + "]:
    no ResourceLoader available");
    }
/**
*这里对 Resource 的路径模式进行解析，比如我们设定的各种 Ant 格式的路径定义，得到需要的
*Resource 集合，这些 Resource 集合指向我们已经定义好的 BeanDefinition 信息，可以是多个文件。
*/
    if (resourceLoader instanceof ResourcePatternResolver) {
        // Resource pattern matching available.
        try {
```

Download at http://www.pin5i.com/

```
// 调用 DefaultResourceLoader 的 getResource 完成具体的 Resource 定位。
                Resource[] resources = ((ResourcePatternResolver) resourceLoader).
        getResources(location);
                int loadCount = loadBeanDefinitions(resources);
                if (actualResources != null) {
                    for (int i = 0; i < resources.length; i++) {
                        actualResources.add(resources[i]);
                    }
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Loaded " + loadCount + " bean definitions from
        location pattern [" + location + "]");
                }
                return loadCount;
            }
            catch (IOException ex) {
                throw new BeanDefinitionStoreException(
                        "Could not resolve bean definition resource pattern [" +
        location + "]", ex);
            }
        }
        else {
            // Can only load single resources by absolute URL.
            // 调用 DefaultResourceLoader 的 getResource 完成具体的 Resource 定位。
            Resource resource = resourceLoader.getResource(location);
            int loadCount = loadBeanDefinitions(resource);
            if (actualResources != null) {
                actualResources.add(resource);
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Loaded " + loadCount + " bean definitions from
        location [" + location + "]");
            }
            return loadCount;
        }
    }
    //对于取得 Resource 的具体过程，我们可以看看 DefaultResourceLoader 是怎样完成的：
    public Resource getResource(String location) {
        Assert.notNull(location, "Location must not be null");
        //这里处理带有 classpath 标识的 Resource。
        if (location.startsWith(CLASSPATH_URL_PREFIX)) {
            return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.
        length()), getClassLoader());
        }
        else {
            try {
                // Try to parse the location as a URL...
                // 这里处理 URL 标识的 Resource 定位。
                URL url = new URL(location);
                return new UrlResource(url);
            }
            catch (MalformedURLException ex) {
                // No URL -> resolve as resource path.
                /**
                *如果既不是 classpath，也不是 URL 标识的 Resource 定位，则把 getResource 的重任
                *交给 getResourceByPath，这个方法是一个 protected 方法，默认的实现是得到一
                *个 ClassPathContextResource，这个方法常常会用子类来实现。
                */
```

Download at http://www.pin5i.com/

```
                return getResourceByPath(location);
            }
        }
    }
```

前面我们看到的 getResourceByPath 会被子类 FileSystemXmlApplicationContext 实现，这个方法返回的是一个 FileSystemResource 对象，通过这个对象 Spring 可以进行相关的 IO 操作，完成 BeanDefinition 的定位。分析到这里已经一目了然，它实现的就是对 path 进行解析，然后生成一个 FileSystemResource 对象并返回，如代码清单 2-6 所示。

代码清单 2-6　FileSystemXmlApplicationContext 生成 Resource 对象

```
protected Resource getResourceByPath(String path) {
    if (path != null && path.startsWith("/")) {
        path = path.substring(1);
    }
    return new FileSystemResource(path);
}
```

如果是其他的 ApplicationContext，那么对应地会生成其他种类的 Resource，比如 ClassPathResource、ServletContextResource 等。关于 Spring 中 Resource 的种类，可以在图 2-6 中的 Resource 类的继承关系中了解。作为接口的 Resource 定义了许多与 IO 相关的操作，这些操作也都可以从图 2-6 中 Resource 的接口定义中看到。这些接口对不同的 Resource 实现代表着不同的意义，是 Resource 的实现需要考虑的。

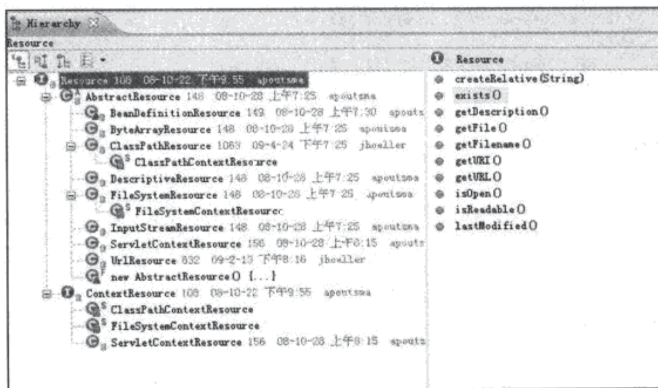


图 2-6　Resource 的定义和继承关系

从图 2-6 中我们可以看到 Resource 的定义和它的继承关系，通过对前面的实现原理的分析，我们以 FileSystemXmlApplicationContext 的实现原理为例子，了解了 Resource 定位问题的解决方案，即以 FileSystem 方式存在的 Resource 的定位实现。在 BeanDefinition 定位完成的基础上，就可以通过返回的 Resource 对象来进行 BeanDefinition 的载入了。在定位过程完成以后，为 BeanDefinition 的载入创造了 IO 操作的条件，但是具体的数据还没有开始读入。这些数据的读入将在下面看到的 BeanDefinition 的载入和解析中来完成。仍然以水桶为例子，这里就像如果要用水桶去打水，那么先要找到水源。这

Download at http://www.pin5i.com/

里完成对 Resource 的定位，就类似于水源已经找到了，下面就是打水的过程了，类似于把找到的水装到水桶里的过程。找水不简单，但与打水相比，我们发现打水更需要技巧。

2.3.2　BeanDefinition 的载入和解析

对 IoC 容器来说，BeanDefinition 的载入过程相当于把我们定义的 BeanDefinition 在 IoC 容器中转化成一个 Spring 内部表示的数据结构的过程。IoC 容器对 Bean 的管理和依赖注入功能的实现，是通过对其持有的 BeanDefinition 进行各种相关的操作来完成的。这些 BeanDefinition 数据在 IoC 容器里通过一个 HashMap 来保持和维护，当然这只是一种比较简单的维护方式，如果你觉得需要提高 IoC 容器的性能和容量，完全可以自己做一些扩展。我们从 DefaultListableBeanFactory 来入手看看 IoC 容器是怎样完成 BeanDefinition 载入的。这个 DefaultListableBeanFactory 已经是我们非常熟悉的基本 IoC 容器，在前面已经碰到过多次，相信大家对它一定不会感到陌生。为了了解这一点，我们先回到 IoC 容器的初始化入口，也就是到 refresh() 方法去看一看。这个方法的最初是在 FileSystemXmlApplicationContext 的构造函数中被调用的，它的调用意味着容器的初始化或数据更新，这些初始化和更新的数据当然就是 BeanDefinition，如代码清单 2-7 所示。

代码清单 2-7　启动 BeanDefinition 的载入

```
public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh,
    ApplicationContext parent)
        throws BeansException {
    super(parent);
    setConfigLocations(configLocations);
    //这里调用容器的 refresh，是载入 BeanDefinition 的入口。
    if (refresh) {
        refresh();
    }
}
```

对于容器的启动来说，refresh 是一个很重要的方法，我们看看它的实现。在 AbstractApplicationContext 类（它是 FileSystemXmlApplicationContext 的基类）中找到这个方法，它详细地描述了整个 ApplicationContext 的初始化过程，比如 BeanFactory 的更新，messagesource 和 postprocessor 的注册，等等。这里看起来更像是对 ApplicationContext 进行初始化的模板或执行提纲，这个执行过程为 IoC 容器 Bean 的生命周期管理提供了条件。这个 IoC 容器的 refresh 过程如代码清单 2-8 所示。

代码清单 2-8　对 IoC 容器的 refresh 的实现

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();
        // Tell the subclass to refresh the internal bean factory.
        // 这里是在子类中启动 refreshBeanFactory()的地方。
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
        // Prepare the Bean Factory for use in this context.
        prepareBeanFactory(beanFactory);
        try {
            // Allows post-processing of the bean factory in context subclasses.
```

Download at http://www.pin5i.com/

```
            postProcessBeanFactory(beanFactory);
            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);
            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);
            // Initialize message source for this context.
            initMessageSource();
            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();
            // Initialize other special beans in specific context subclasses.
            onRefresh();
            // Check for listener beans and register them.
            registerListeners();
            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);
            // Last step: publish corresponding event.
            finishRefresh();
        }
        catch (BeansException ex) {
            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();
            // Reset 'active' flag.
            cancelRefresh(ex);
            // Propagate exception to caller.
            throw ex;
        }
    }
}
```

我们进入到 AbstractRefreshableApplicationContext 的 refreshBeanFactory() 方法中，在这个方法里创建了 BeanFactory。在创建 IoC 容器前，如果已经有容器存在，那么需要把已有的容器销毁和关闭，保证在 refresh 以后使用的是新建立起来的 IoC 容器。这么看来，这个 refresh 非常像我们对容器的重启动，就像计算机的重启动那样。在建立好当前的 IoC 容器以后，开始了对容器的初始化过程，比如 BeanDefinition 的载入，具体的实现如代码清单 2-9 所示。

代码清单 2-9 AbstractRefreshableApplicationContext 的 refreshBeanFactory 方法

```
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
    //创建 IoC 容器，这里使用的是 DefaultListableBeanFactory。
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        //启动对 BeanDefintion 的载入。
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing XML document
```

Download at http://www.pin5i.com/

```
            for " + getDisplayName(), ex);
        }
    }
```

这里调用的 loadBeanDefinitions 实际上是一个抽象方法，那么实际的载入过程是在哪里发生的呢？我们看看前面提到的 loadBeanDefinitios 在 AbstractRefreshableApplicationContext 的子类 AbstractXmlApplicationContext 中的实现，在这个 loadBeanDefinitions 中，初始化了读取器 XmlBeanDefinitionReader，然后再把这个读取器在 IoC 容器中设置好（过程和编程式使用 XmlBeanFactory 是类似的），最后是启动读入器来完成 BeanDefinition 在 IoC 容器中的载入，如代码清单 2-10 所示。

代码清单 2-10 AbstractXmlApplicationContext 中的 loadBeanDefinitions

```
public abstract class AbstractXmlApplicationContext extends
    AbstractRefreshableConfigApplicationContext {
public AbstractXmlApplicationContext() {
}
public AbstractXmlApplicationContext(ApplicationContext parent) {
    super(parent);
}
//这里是实现 loadBeanDefinitions 的地方。
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
    throws IOException {
    // Create a new XmlBeanDefinitionReader for the given BeanFactory.
    /**
    *创建 XmlBeanDefinitionReader，并通过回调设置到 BeanFactory 中去，
    *创建 BeanFactory 的过程可以参考上文对编程式使用 IoC 容器的相关分析，这里和前面一样，
    *使用的也是 DefaultListableBeanFactory。
    */
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader
    (beanFactory);
    /**
    *Configure the bean definition reader with this context's
    *resource loading environment.
    */
    /**
    *这里设置 XmlBeanDefinitionReader，为 XmlBeanDefinitionReader
    *配置 ResourceLoader，因为 DefaultResourceLoader 是父类，所以 this 可以直接被使用。
    */
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
    /**
    *Allow a subclass to provide custom initialization of the reader,
    *then proceed with actually loading the bean definitions.
    */
// 这是启动 Bean 定义信息载入的过程。
    initBeanDefinitionReader(beanDefinitionReader);
    loadBeanDefinitions(beanDefinitionReader);
}
protected void initBeanDefinitionReader(XmlBeanDefinitionReader beanDefinitionReader) {
}
```

接着就是 loadBeanDefinitions 调用的地方，首先得到 BeanDefinition 信息的 Resource 定位，然后直接调用 XmlBeanDefinitionReader 读取，具体的载入过程是委托给 BeanDefinitionReader 完成的。因为这里的 BeanDefinition 是通过 XML 文件定义的，所以这

Download at http://www.pin5i.com/

里使用 XmlBeanDefinitionReader 来载入 BeanDefinition 到容器中，如代码清单 2-11 所示。

代码清单 2-11 XmlBeanDefinitionReader 载入 XmlBeanDefinitionReader

```
    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
        BeansException, IOException {
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
            reader.loadBeanDefinitions(configResources);
        }
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            reader.loadBeanDefinitions(configLocations);
        }
    }
    protected Resource[] getConfigResources() {
        return null;
    }
}
```

通过以上实现原理的分析，我们可以看到，在初始化 FileSystmXmlApplicationContext 的过程中，是通过调用 IoC 容器的 refresh 来启动整个 BeanDefinition 的载入过程的，这个初始化是通过定义的 XmlBeanDefinitionReader 来完成的。同时，我们也知道实际使用的 IoC 容器是 DefultListableBeanFactory，具体的 Resource 载入在 XmlBeanDefinitionReader 读入 BeanDefinition 时实现。因为 Spring 可以对应不同形式的 BeanDefinition。由于这里使用的是 XML 方式的定义，所以需要使用 XmlBeanDefinitionReader。如果使用了其他的 BeanDefinition 方式，就需要使用其他种类的 BeanDefinitionReader 来完成数据的载入工作。在 XmlBeanDefinitionReader 的实现中可以看到，是在 reader.loadBeanDefinitions 中开始进行 BeanDefinition 的载入的，而这时 XmlBeanDefinitionReader 的父类 AbstractBeanDefinitionReader 已经为 BeanDefinition 的载入做好了准备，如代码清单 2-12 所示。

代码清单 2-12 AbstractBeanDefinitionReader 载入 BeanDefinitions

```
public int loadBeanDefinitions(Resource[] resources) throws
        BeanDefinitionStoreException {
        //如果 Resource 为空，则停止 BeanDefinition 的载入。
        /**
        *然后启动载入 BeanDefinition 的过程，这个过程会遍历整个 Resource 集合所包含的
        *BeanDefinition 信息。
        */
        Assert.notNull(resources, "Resource array must not be null");
        int counter = 0;
        for (int i = 0; i < resources.length; i++) {
            counter += loadBeanDefinitions(resources[i]);
        }
        return counter;
}
```

这里调用的是 loadBeanDefinitions(Resource res) 方法，然而这个方法在 AbstractBeanDefinitionReader 类里是没有实现的，它是一个接口方法，具体的实现在 XmlBeanDefinitionReader 中。在读取器中，需要得到代表 XML 文件的 Resource，因为这个 Resource 对象封装了对 XML 文件的 IO 操作，所以读取器可以在打开 IO 流后得到 XML 的文

Download at http://www.pin5i.com/

件对象。有了这个 Document 对象以后，就可以按照 Spring 的 Bean 定义规则来对这个 XML 的文档树进行解析了，这个解析是交给 BeanDefinitionParserDelegate 来完成的，看起来实现脉络很清楚。具体可以参考代码实现，如代码清单 2-13 所示。

代码清单 2-13 对 BeanDefinition 的载入实现

```
//这里是调用的入口。
public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {
    return loadBeanDefinitions(new EncodedResource(resource));
}
//这里是载入 XML 形式的 BeanDefinition 的地方。
public int loadBeanDefinitions(EncodedResource encodedResource) throws
    BeanDefinitionStoreException {
    Assert.notNull(encodedResource, "EncodedResource must not be null");
    if (logger.isInfoEnabled()) {
        logger.info("Loading XML bean definitions from " + encodedResource.
    getResource());
    }
    Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.
    get();
    if (currentResources == null) {
        currentResources = new HashSet<EncodedResource>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
                "Detected recursive loading of " + encodedResource + " - check
    your import definitions!");
    }
    //这里得到 XML 文件，并得到 IO 的 InputSource 准备进行读取。
    try {
       InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
           return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
        }
        finally {
            inputStream.close();
        }
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
                "IOException parsing XML document from " + encodedResource.
    getResource(), ex);
    }
    finally {
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.set(null);
        }
    }
}
//具体的读取过程可以在 doLoadBeanDefinitions 方法中找到。
//这是从特定的 XML 文件中实际载入 BeanDefinition 的地方。
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
```

Download at http://www.pin5i.com/

```
        throws BeanDefinitionStoreException {
    try {
        int validationMode = getValidationModeForResource(resource);
        /**
        *这里取得XML文件的Document对象，这个解析过程是由 documentLoader 完成的，
        *这个documentLoader是DefaultDocumentLoader,在定义documentLoader的地方创建。
        */
        Document doc = this.documentLoader.loadDocument(
                inputSource, getEntityResolver(), this.errorHandler, validationMode,
    isNamespaceAware());
        /**
        *这里启动的是对BeanDefinition解析的详细过程，这个解析会使用到Spring的Bean
        *配置规则，是我们下面需要详细关注的地方。
        */
        return registerBeanDefinitions(doc, resource);
    }
    catch (BeanDefinitionStoreException ex) {
        throw ex;
    }
    catch (SAXParseException ex) {
        throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                "Line " + ex.getLineNumber() + " in XML document from " +
    resource + " is invalid", ex);
    }
    catch (SAXException ex) {
        throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                "XML document from " + resource + " is invalid", ex);
    }
    catch (ParserConfigurationException ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
                "Parser configuration exception parsing XML from " + resource,
    ex);
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
                "IOException parsing XML document from " + resource, ex);
    }
    catch (Throwable ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
                "Unexpected exception parsing XML document from " + resource,
    ex);
    }
}
```

感兴趣的读者，可以到 DefaultDocumentLoader 中去看看是怎样得到 Document 对象的，这里就不详细分析了。我们关心的是 Spring 的 BeanDefinion 是怎样按照 Spring 的 Bean 语义要求进行解析并转化为容器内部数据结构的，这个过程是在 registerBeanDefinitions (doc, resource) 中完成的。具体的过程是由 BeanDefinitionDocumentReader 来完成的，这个 registerBeanDefinition 还对载入的 Bean 的数量进行了统计。具体的过程如代码清单 2-14 所示。

代码清单 2-14　registerBeanDefinition 的代码实现

```
public int registerBeanDefinitions(Document doc, Resource resource) throws
    BeanDefinitionStoreException {
    // Read document based on new BeanDefinitionDocumentReader SPI.
    // 这里得到BeanDefinitionDocumentReader来对xml的BeanDefinition进行解析。
```

Download at http://www.pin5i.com/

```
        BeanDefinitionDocumentReader documentReader =
        createBeanDefinitionDocumentReader();
        int countBefore = getRegistry().getBeanDefinitionCount();
        // 具体的解析过程在这个 registerBeanDefinitions 中完成。
        documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
        return getRegistry().getBeanDefinitionCount() - countBefore;
    }
```

BeanDefinition 的载入包括两部分，首先是通过调用 XML 的解析器得到 document 对象，但这些 document 对象并没有按照 Spring 的 Bean 规则进行解析。在完成通用的 XML 解析以后，才是按照 Spring 的 Bean 规则进行解析的地方，按照 Spring 的 Bean 规则进行解析的过程是在 documentReader 中实现的。这里使用的 documentReader 是默认设置好的 DefaultBeanDefinitionDocumentReader。这个 DefaultBeanDefinitionDocumentReader 的创建是在以下的方法里完成的，然后再完成 BeanDefinition 的处理，处理的结果由 BeanDefinitionHolder 对象来持有。这个 BeanDefinitionHolder 除了持有 BeanDefinition 对象外，还持有了其他与 BeanDefinition 的使用相关的信息，比如 Bean 的名字、别名集合等。这个 BeanDefinitionHolder 的生成是通过对 Document 文档树的内容进行解析来完成的，可以看到这个解析过程是由 BeanDefinitionParserDelegate 来实现（具体在 processBeanDefinition 方法中实现）的，同时这个解析是与 Spring 对 BeanDefinition 的配置规则紧密相关的。具体的实现原理如代码清单 2-15 所示。

代码清单 2-15 创建 BeanDefinitionDocumentReader

```
protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader() {
    return BeanDefinitionDocumentReader.class.cast(BeanUtils.instantiateClass
    (this.documentReaderClass));
}
 //这样，得到了 documentReader 以后，为具体的 Spring Bean 的解析过程准备好了数据。
 /**
 *这里是处理 BeanDefinition 的地方，具体的处理是委托给 BeanDefinitionParserDelegate
 *来完成的，ele 对应于我们的 Spring BeanDefinition 中定义的 xml 元素。
 */
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate
    delegate) {
    /* BeanDefinitionHolder 是 BeanDefinition 对象的封装类，封装了 BeanDefinition，
    *Bean 的名字和别名。用它来完成向 IoC 容器注册。得到这个 BeanDefinitionHolder 实际上就意
    *味着获得了 BeanDefinition,是通过 BeanDefinitionParserDelegate 对 XML 元素的信息按照
    *Spring 的 Bean 规则进行解析得到的。
    */
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // 这里是向 IoC 容器注册解析得到的 BeanDefinition 的地方。
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
            getReaderContext().getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with
            name '" + bdHolder.getBeanName() + "'", ele, ex);
        }
        // 在 BeanDefinition 向 IoC 容器注册完以后，发送消息。
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition
```

Download at http://www.pin5i.com/

```
        (bdHolder));
    }
}
```

具体的 Spring BeanDefinition 的解析是在 BeanDefinitionParserDelegate 中完成的。这个类里包含了各种 Spring Bean 定义规则的处理，感兴趣的读者可以仔细研究。比如我们最熟悉的对 Bean 元素的处理是怎样完成的，也就是在 XML 定义文件中出现的<bean></bean>这个最常见的元素信息是怎样处理的。在这里，我们会看到那些熟悉的 BeanDefinition 定义的处理，比如 id、name、aliase 等属性元素。把这些元素的值从 XML 文件相应的元素的属性中读取出来以后，会被设置到生成的 BeanDefinitionHolder 中去。这些属性的解析还是比较简单的。对于其他元素配置的解析，比如各种 Bean 的属性配置，通过一个较为复杂的解析过程，这个过程是由 parseBeanDefinitionElement 来完成的。解析完成以后，会把解析结果放到 BeanDefinition 对象中并设置到 BeanDefinitionHolder 中去，如代码清单 2-16 所示。

代码清单 2-16 BeanDefinitionParserDelegate 对 bean 元素定义的处理

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition
    containingBean) {
    //这里取得在<bean>元素中定义的 id、name 和 aliase 属性的值。
    String id = ele.getAttribute(ID_ATTRIBUTE);
    String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
    List<String> aliases = new ArrayList<String>();
    if (StringUtils.hasLength(nameAttr)) {
        String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, BEAN_NAME_
    DELIMITERS);
        aliases.addAll(Arrays.asList(nameArr));
    }
    String beanName = id;
    if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
        beanName = aliases.remove(0);
        if (logger.isDebugEnabled()) {
            logger.debug("No XML 'id' specified - using '" + beanName +
                    "' as bean name and " + aliases + " as aliases");
        }
    }
    if (containingBean == null) {
        checkNameUniqueness(beanName, aliases, ele);
    }
    //这个方法会引发对 bean 元素的详细解析。
AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,
    beanName,containingBean);
    if (beanDefinition != null) {
        if (!StringUtils.hasText(beanName)) {
            try {
                if (containingBean != null) {
                    beanName = BeanDefinitionReaderUtils.generateBeanName(
                            beanDefinition, this.readerContext.getRegistry(),
                            true);
                }
                else {
                    beanName =
                    this.readerContext.generateBeanName(beanDefinition);
                    /**
                    *Register an alias for the plain bean class name, if still
                    *possible,if the generator returned the class name plus a
                    *suffix.This is expected for Spring 1.2/2.0 backwards
                    *compatibility.
```

Download at http://www.pin5i.com/

```
                         */
                        String beanClassName = beanDefinition.getBeanClassName();
                        if (beanClassName != null &&
                                beanName.startsWith(beanClassName) &&
                                beanName.length() > beanClassName.length() &&
!this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                            aliases.add(beanClassName);
                        }
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("Neither XML 'id' nor 'name' specified - " +
                                "using generated bean name [" + beanName + "]");
                    }
                }
                catch (Exception ex) {
                    error(ex.getMessage(), ele);
                    return null;
                }
            }
            String[] aliasesArray = StringUtils.toStringArray(aliases);
            return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
        }
        return null;
    }
```

我们看到了对 Bean 元素进行解析的过程，也就是 BeanDefinition 依据 XML 的<bean>定义被创建的过程。这个 BeanDefinition 可以看成是<bean>定义的抽象，如图 2-7 所示。这个数据对象里封装的数据大多都是与<bean>定义相关的，也有很多就是我们在定义 Bean 时看到的那些 Spring 标记，比如我们熟悉的 init-method、destroy-method、factory-method，等等，这个 BeanDefinition 数据类型是非常重要的，它封装了很多基本数据。有了这些基本数据，IoC 容器才能对 Bean 配置进行处理，才能实现相应的容器特性。

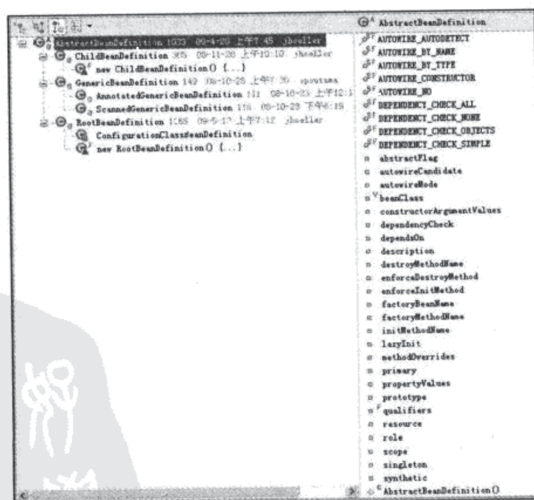


图 2-7　BeanDefinition 的数据定义

Download at http://www.pin5i.com/

看起来很熟悉吧，beanClass、description、lazyInit 这些属性都是在配置 Bean 时经常碰到的，原来都跑到这里来了。这个 BeanDefinition 是 IoC 容器体系中非常重要的核心数据结构。通过解析以后，这些数据已经做好在 IoC 容器里大显身手的准备了。对 BeanDefinition 的元素的处理如代码清单 2-17 所示，在这个过程中可以看到对 Bean 定义的相关处理，比如对元素 attribute 值的处理，对元素属性值的处理，对构造函数设置的处理，等等。

代码清单 2-17　对 BeanDefinition 定义元素的处理

```
public AbstractBeanDefinition parseBeanDefinitionElement(
        Element ele, String beanName, BeanDefinition containingBean) {
    this.parseState.push(new BeanEntry(beanName));
    /**
    *这里只读取定义的<bean>中设置的 class 名字，然后载入到 BeanDefinition 中去，
    *只是做个记录，并不涉及对象的实例化过程，对象的实例化实际上是在依赖注入时完成的。
    */
    String className = null;
    if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
        className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
    }
    try {
        String parent = null;
        if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
            parent = ele.getAttribute(PARENT_ATTRIBUTE);
        }
        //这里生成需要的 BeanDefinition 对象，为 Bean 定义信息的载入做准备。
        AbstractBeanDefinition bd = createBeanDefinition(className, parent);
        //这里对当前的 Bean 元素进行属性解析，并设置 description 的信息。
        parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
        bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
    DESCRIPTION_ELEMENT));
        //从名字可以清楚地看到，这里是对各种<bean>元素的信息进行解析的地方。
        parseMetaElements(ele, bd);
        parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
        parseReplacedMethodSubElements(ele, bd.getMethodOverrides());
        //解析<bean>的构造函数设置。
        parseConstructorArgElements(ele, bd);
        //解析<bean>的 property 设置。
        parsePropertyElements(ele, bd);
        parseQualifierElements(ele, bd);
        bd.setResource(this.readerContext.getResource());
        bd.setSource(extractSource(ele));
        return bd;
    }
/**
*下面这些异常是我们在配置 bean 出现问题时经常可以看到的，原来是在这里抛出的，这些检查是在
*createBeanDefinition 时进行的，会检查 bean 的 class 设置是否正确，比如这个类是不是能找到。
*/
    catch (ClassNotFoundException ex) {
        error("Bean class [" + className + "] not found", ele, ex);
    }
    catch (NoClassDefFoundError err) {
        error("Class that bean class [" + className + "] depends on not found",
        ele, err);
    }
    catch (Throwable ex) {
        error("Unexpected failure during bean definition parsing", ele, ex);
```

Download at http://www.pin5i.com/

```
        }
        finally {
            this.parseState.pop();
        }
        return null;
    }
```

上面是具体生成 BeanDefinition 的地方。在这里，我们举一个对 property 进行解析的例子来完成对整个 BeanDefinition 载入过程的分析，还是在类 BeanDefinitionParserDelegate 的代码中，它对 BeanDefinition 中的定义一层一层地进行解析，比如从属性元素集合到具体的每一个属性元素，然后才是对具体的属性值的处理。根据解析结果，对这些属性值的处理会封装成 PropertyValue 对象并设置到 BeanDefinition 对象中去，如代码清单 2-18 所示。

代码清单 2-18 对 BeanDefinition 中 Property 元素集合的处理

```
//这里对指定 bean 元素的 property 子元素集合进行解析。
public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
    //遍历所有 bean 元素下定义的 property 元素。
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (node instanceof Element && DomUtils.nodeNameEquals(node, PROPERTY_
    ELEMENT)) {
            //在判断是 property 元素后对该 property 元素进行解析的过程。
            parsePropertyElement((Element) node, bd);
        }
    }
}
public void parsePropertyElement(Element ele, BeanDefinition bd) {
    //这里取得 property 的名字。
    String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
    if (!StringUtils.hasLength(propertyName)) {
        error("Tag 'property' must have a 'name' attribute", ele);
        return;
    }
    this.parseState.push(new PropertyEntry(propertyName));
    try {
        /**
        *如果同一个 bean 中已经有同名 property 的存在，则不进行解析，直接返回。也就是说，
        *如果在同一个 bean 中有同名的 property 设置，那么起作用的只是第一个。
        */
        if (bd.getPropertyValues().contains(propertyName)) {
            error("Multiple 'property' definitions for property '" +
    propertyName + "'", ele);
            return;
        }
        /**
        *这里是解析 property 值的地方，返回的对象对应对 Bean 定义的 property 属性
        *设置的解析结果，这个解析结果会封装到 PropertyValue 对象中，然后设置到
        *BeanDefinitionHolder 中去。
        */
        Object val = parsePropertyValue(ele, bd, propertyName);
        PropertyValue pv = new PropertyValue(propertyName, val);
        parseMetaElements(ele, pv);
        pv.setSource(extractSource(ele));
        bd.getPropertyValues().addPropertyValue(pv);
    }
```

Download at http://www.pin5i.com/

```
        finally {
            this.parseState.pop();
        }
    }
    //这里取得property元素的值，也许是一个list或其他。
    public Object parsePropertyValue(Element ele, BeanDefinition bd, String propertyName) {
        String elementName = (propertyName != null) ?
                        "<property> element for property '" + propertyName + "'" :
                        "<constructor-arg> element";
        // Should only have one child element: ref, value, list, etc.
        NodeList nl = ele.getChildNodes();
        Element subElement = null;
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element && !DomUtils.nodeNameEquals(node, DESCRIPTION_
            ELEMENT) &&
                    !DomUtils.nodeNameEquals(node, META_ELEMENT)) {
                // Child element is what we're looking for.
                if (subElement != null) {
                    error(elementName + " must not contain more than one sub-element",
                    ele);
                }
                else {
                    subElement = (Element) node;
                }
            }
        }
        //这里判断property的属性，是ref还是value，不允许同时是ref和value。
        boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
        boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
        if ((hasRefAttribute && hasValueAttribute) ||
                ((hasRefAttribute || hasValueAttribute) && subElement != null)) {
                error(elementName +
                    " is only allowed to contain either 'ref' attribute OR 'value'
                    attribute OR sub-element", ele);
        }
        //如果是ref，创建一个ref的数据对象RuntimeBeanReference，这个对象封装了ref的信息。
        if (hasRefAttribute) {
            String refName = ele.getAttribute(REF_ATTRIBUTE);
            if (!StringUtils.hasText(refName)) {
                error(elementName + " contains empty 'ref' attribute", ele);
            }
            RuntimeBeanReference ref = new RuntimeBeanReference(refName);
            ref.setSource(extractSource(ele));
            return ref;
        } //如果是value，创建一个它的数据对象TypedStringValue，这个对象封装了value的信息。
        else if (hasValueAttribute) {
            TypedStringValue valueHolder = new TypedStringValue(ele.getAttribute
        (VALUE_ATTRIBUTE));
            valueHolder.setSource(extractSource(ele));
            return valueHolder;
        } //如果还有子元素，触发对子元素的解析。
        else if (subElement != null) {
            return parsePropertySubElement(subElement, bd);
        }
        else {
            // Neither child element nor "ref" or "value" attribute found.
            error(elementName + " must specify a ref or value", ele);
```

Download at http://www.pin5i.com/

```
            return null;
        }
    }
```

这里是对 property 子元素的解析过程，Array、List、Set、Map、Prop 等各种元素都会在这里进行解析，生成对应的数据对象，比如 ManagedList、ManagedArray、ManagedSet 等。这些 Managed 类是 Spring 对具体的 BeanDefinition 的数据封装。具体的解析过程读者可以去看看自己感兴趣的部分，比如 parseArrayElement、parseListElement、parseSetElement、parseMapElement、parsePropElement 对应着不同类型的数据解析，同时这些具体的解析方法在 BeanDefinitionParserDelegate 类中也都能够找到。因为方法命名很清晰，所以从方法名字上就能够很快地找到。下面，以对 Property 的元素进行解析的过程为例，通过它的实现来说明这个具体的解析过程是怎样完成的，如代码清单 2-19 所示。

代码清单 2-19　对属性元素进行解析

```
public Object parsePropertySubElement(Element ele, BeanDefinition bd, String
    defaultValueType) {
    if (!isDefaultNamespace(ele.getNamespaceURI())) {
        return parseNestedCustomElement(ele, bd);
    }
    else if (DomUtils.nodeNameEquals(ele, BEAN_ELEMENT)) {
        BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);
        if (nestedBd != null) {
            nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);
        }
        return nestedBd;
    }
    else if (DomUtils.nodeNameEquals(ele, REF_ELEMENT)) {
        // A generic reference to any name of any bean.
        String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
        boolean toParent = false;
        if (!StringUtils.hasLength(refName)) {
            // A reference to the id of another bean in the same XML file.
            refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
            if (!StringUtils.hasLength(refName)) {
                // A reference to the id of another bean in a parent context.
                refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
                toParent = true;
                if (!StringUtils.hasLength(refName)) {
                    error("'bean', 'local' or 'parent' is required for <ref>
                    element", ele);
                    return null;
                }
            }
        }
        if (!StringUtils.hasText(refName)) {
            error("<ref> element contains empty target attribute", ele);
            return null;
        }
        RuntimeBeanReference ref = new RuntimeBeanReference(refName, toParent);
        ref.setSource(extractSource(ele));
        return ref;
    }
    else if (DomUtils.nodeNameEquals(ele, IDREF_ELEMENT)) {
        return parseIdRefElement(ele);
    }
```

Download at http://www.pin5i.com/

```
        else if (DomUtils.nodeNameEquals(ele, VALUE_ELEMENT)) {
            return parseValueElement(ele, defaultValueType);
        }
        else if (DomUtils.nodeNameEquals(ele, NULL_ELEMENT)) {
            // It's a distinguished null value. Let's wrap it in a TypedStringValue
            // object in order to preserve the source location.
            TypedStringValue nullHolder = new TypedStringValue(null);
            nullHolder.setSource(extractSource(ele));
            return nullHolder;
        }
        else if (DomUtils.nodeNameEquals(ele, ARRAY_ELEMENT)) {
            return parseArrayElement(ele, bd);
        }
        else if (DomUtils.nodeNameEquals(ele, LIST_ELEMENT)) {
            return parseListElement(ele, bd);
        }
        else if (DomUtils.nodeNameEquals(ele, SET_ELEMENT)) {
            return parseSetElement(ele, bd);
        }
        else if (DomUtils.nodeNameEquals(ele, MAP_ELEMENT)) {
            return parseMapElement(ele, bd);
        }
        else if (DomUtils.nodeNameEquals(ele, PROPS_ELEMENT)) {
            return parsePropsElement(ele);
        }
        else {
            error("Unknown property sub-element: [" + ele.getNodeName() + "]", ele);
            return null;
        }
    }
```

我们看看类似 List 这样的属性配置是怎样被解析的，依然在 BeanDefinitionParser-Delegate 中，返回的是一个 List 对象，这个 List 是 Spring 定义的 ManagedList，作为封装 List 这类配置定义的数据封装，如代码清单 2-20 所示。

代码清单 2-20 解析 BeanDefinition 中的 List 元素

```
public List parseListElement(Element collectionEle, BeanDefinition bd) {
    String defaultElementType = collectionEle.getAttribute(VALUE_TYPE_ATTRIBUTE);
    NodeList nl = collectionEle.getChildNodes();
    ManagedList<Object> target = new ManagedList<Object>(nl.getLength());
    target.setSource(extractSource(collectionEle));
    target.setElementTypeName(defaultElementType);
    target.setMergeEnabled(parseMergeAttribute(collectionEle));
    //具体的List元素的解析过程。
    parseCollectionElements(nl, target, bd, defaultElementType);
    return target;
}
protected void parseCollectionElements(
        NodeList elementNodes, Collection<Object> target, BeanDefinition bd,
    String defaultElementType) {
    //遍历所有的元素节点，并判断其类型是否为Element。
    for (int i = 0; i < elementNodes.getLength(); i++) {
        Node node = elementNodes.item(i);
        if (node instanceof Element && !DomUtils.nodeNameEquals(node, DESCRIPTION_
    ELEMENT)) {
    /**
    *加入到target中，target是一个ManagedList，同时触发对下一层子元素的解析过程，
    *这是一个递归调用。
```

Download at http://www.pin5i.com/

```
                */
                target.add(parsePropertySubElement((Element) node, bd, defaultElement
                Type));
            }
        }
    }
```

经过这样逐层地解析，我们在 XML 文件中定义的 BeanDefinition 就被整个给载入到了 IoC 容器中，并在容器中建立了数据映射。在 IoC 容器中建立了对应的数据结构，或者说可以看成是 POJO 对象在 IoC 容器中的映像，这些数据结构可以以 AbstractBeanDefinition 为入口，让 IoC 容器执行索引、查询和操作。简单的 POJO 操作背后其实并不简单，经过以上的载入过程，IoC 容器大致完成了管理 Bean 对象的数据准备工作（或者说是初始化过程）。但是，重要的依赖注入实际上在这个时候还没有发生，现在，在 IoC 容器 BeanDefinition 中存在的还只是一些静态的配置信息。严格地来说，这时候的容器还没有完全起作用，要完全发挥容器的作用，还需完成数据向容器的注册。

2.3.3 BeanDefinition 在 IoC 容器中的注册

我们已经分析过 BeanDefinition 在 IoC 容器中载入和解析的过程。在这些动作完成以后，用户定义的 BeanDefinition 信息已经在 IoC 容器内建立起了自己的数据结构以及相应的数据表示，但此时这些数据还不能让 IoC 容器直接使用，需要在 IoC 容器中对这些 BeanDefinition 数据进行注册。这个注册为 IoC 容器了提供更友好的使用方式，在 DefaultListableBeanFactory 中，是通过一个 HashMap 来持有载入的 BeanDefinition 的，这个 HashMap 的定义在 DefaultListableBeanFactory 可以看到，如下所示。

```
/** Map of bean definition objects, keyed by bean name */
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap
    <String, BeanDefinition>();
```

将解析得到的 BeanDefinition 向 IoC 容器中的 beanDefinitionMap 注册的过程是在载入 BeanDefinition 完成后进行的，注册的调用过程如图 2-8 所示。

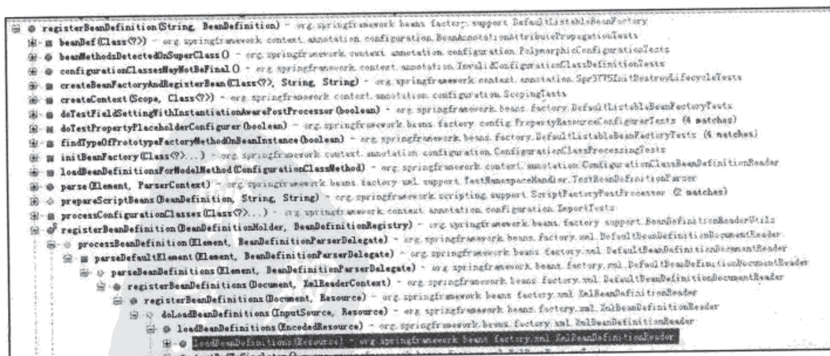


图 2-8 registerBeanDefinition 的调用过程

Download at http://www.pin5i.com/

我们跟踪以上的代码调用去看一下具体的注册实现，在 DefaultListableBeanFactory 中实现了 BeanDefinitionRegistry 的接口，这个接口的实现完成 BeanDefinition 向容器的注册。这个注册过程不复杂，就是把解析得到的 BeanDefinition 设置到 hashMap 中去。需要注意的是，如果遇到同名的 BeanDefinition 的情况，进行处理的时候需要依据 allowBeanDefinitionOverriding 的配置来完成。具体的是实现如代码清单 2-21 所示。

代码清单 2-21 BeanDefinitio 注册的实现

```
//---------------------------------------------------------------------
// Implementation of BeanDefinitionRegistry interface.
//---------------------------------------------------------------------

public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
        throws BeanDefinitionStoreException {
    Assert.hasText(beanName, "'beanName' must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");
    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(
            beanDefinition.getResourceDescription(), beanName,
                    "Validation of bean definition failed", ex);
        }
    }
    //注册的过程需要 synchronized，保证数据的一致性。
    synchronized (this.beanDefinitionMap) {
    /**
    *这里检查是不是有相同名字的 BeanDefinition 已经在 IoC 容器中注册了，如果有
    *相同名字的 BeanDefinition，但又不允许覆盖，那么抛出异常。
    */
        Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
        if (oldBeanDefinition != null) {
            if (!this.allowBeanDefinitionOverriding) {
                throw new BeanDefinitionStoreException(
                beanDefinition.getResource Description(), beanName,
                        "Cannot register bean definition [" + beanDefinition + "]
                         for bean '" + beanName +
                        "': There is already [" + oldBeanDefinition + "] bound.");
            }
            else {
                if (this.logger.isInfoEnabled()) {
                    this.logger.info("Overriding bean definition for bean '" +
                    beanName +"': replacing [" + oldBeanDefinition + "] with [" +
                    beanDefinition + "]");
                }
            }
        }
/**
*这是正常注册 BeanDefinition 的过程，把 Bean 的名字存入到 beanDefinitionNames 的同时，
*把 beanName 作为 Map 的 key，把 beanDefinition 作为 value 存入到 IoC 容器持有的
*beanDefinitionMap 中去。
*/
        else {
```

Download at http://www.pin5i.com/

```
                this.beanDefinitionNames.add(beanName);
                this.frozenBeanDefinitionNames = null;
            }
            this.beanDefinitionMap.put(beanName, beanDefinition);
            resetBeanDefinition(beanName);
        }
    }
```

完成了 BeanDefinition 的注册，就完成了 IoC 容器的初始化过程。此时，在我们使用的 IoC 容器 DefaultListableBeanFactory 中已经建立了整个 Bean 的配置信息，而且这些 BeanDefinition 已经可以被容器使用了，它们都可在 beanDefinitionMap 里检索和使用。容器的作用就是对这些信息进行处理和维护。这些信息是容器建立依赖反转的基础，有了这些基础数据，下面我们接着看看在 IoC 容器中，依赖注入是怎样完成的。

2.4 IoC 容器的依赖注入

假设当前 IoC 容器已经载入了用户定义的 Bean 信息，并开始分析依赖注入的原理。首先，注意到依赖注入的过程是用户第一次向 IoC 容器索要 Bean 时触发的，当然也有例外，也就是我们可以在 BeanDefinition 信息中通过控制 lazy-init 属性来让容器完成对 Bean 的预实例化。这个预实例化实际上也是一个完成依赖注入的过程，但它在初始化的过程中完成，稍后我们会详细分析这个预实例化的处理。当用户向 IoC 容器索要 Bean 时，如果读者还有印象，那么一定还记得在基本的 IoC 容器接口 BeanFactory 中，有一个 getBean 的接口定义，这个接口的实现就是触发依赖注入发生的地方。为了进一步了解这个依赖注入过程的实现，我们从 DefaultListableBeanFactory 的基类 AbstractBeanFactory 入手去看看 getBean 的实现，如代码清单 2-22 所示。

代码清单 2-22 getBean 触发的依赖注入

```
//---------------------------------------------------------------------
// Implementation of BeanFactory interface.
// 这里是对 BeanFactory 接口的实现，比如 getBean 接口方法。
// 这些 getBean 接口方法最终是通过调用 doGetBean 来实现的。
//---------------------------------------------------------------------
public Object getBean(String name) throws BeansException {
    return doGetBean(name, null, null, false);
}
public <T> T getBean(String name, Class<T> requiredType) throws BeansException {
    return doGetBean(name, requiredType, null, false);
}
public Object getBean(String name, Object... args) throws BeansException {
    return doGetBean(name, null, args, false);
}
public <T> T getBean(String name, Class<T> requiredType, Object[] args) throws
    BeansException {
    return doGetBean(name, requiredType, args, false);
}
//这里是实际去取 bean 的地方，也是触发依赖注入发生的地方。
protected <T> T doGetBean(
        final String name, final Class<T> requiredType, final Object[] args,
    boolean typeCheckOnly)
```

Download at http://www.pin5i.com/

```
        throws BeansException {
    final String beanName = transformedBeanName(name);
    Object bean;
    // Eagerly check singleton cache for manually registered singletons.
    //先从缓存中去取,处理已经被创建过的单件模式的bean，对这种bean的请求不需要重复地创建。
    Object sharedInstance = getSingleton(beanName);
    if (sharedInstance != null && args == null) {
        if (logger.isDebugEnabled()) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                logger.debug("Returning eagerly cached instance of singleton
                bean '" + beanName +
                "' that is not fully initialized yet - a consequence of a
                circular reference");
            }
            else {
                logger.debug("Returning cached instance of singleton bean '" +
                beanName + "'");
            }
        }
    /**
    *这里的getObjectForBeanInstance完成的是FactoryBean的相关处理,
    *以取得FactoryBean的生产结果，我们在前面介绍过BeanFactory和FactoryBean的区别,
    *这个过程我们在下面还会详细地分析。
    */
bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
    }
    else {
        // Fail if we're already creating this bean instance:
        // We're assumably within a circular reference.
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }
    // Check if bean definition exists in this factory.
    /**
    *这里对IoC容器里的BeanDefintion是否存在进行检查，检查是否能在当前的BeanFactory中
    *取到我们需要的bean。如果在当前的工厂中取不到，则到双亲BeanFactory中去取；如果当前的
    *双亲工厂取不到，那就顺着双亲BeanFactory链一直向上查找。
    */
        BeanFactory parentBeanFactory = getParentBeanFactory();
        if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
            // Not found -> check parent.
            String nameToLookup = originalBeanName(name);
            if (args != null) {
                // Delegation to parent with explicit args.
                return (T) parentBeanFactory.getBean(nameToLookup, args);
            }
            else {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
        }
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }
        //这里根据bean的名字取得BeanDefinition.
        final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
```

Download at http://www.pin5i.com/

```
checkMergedBeanDefinition(mbd, beanName, args);
// Guarantee initialization of beans that the current bean depends on.
/**
*取当前bean的所有依赖bean，这样会触发getBean的递归调用，直至取到一个
*没有任何依赖的bean为止。
*/
String[] dependsOn = mbd.getDependsOn();
if (dependsOn != null) {
    for (String dependsOnBean : dependsOn) {
        getBean(dependsOnBean);
        registerDependentBean(dependsOnBean, beanName);
    }
}
// Create bean instance.
/**
*这里创建Singleton bean的实例，通过调用createBean方法，这里有一个
*回调函数getObject，会在getSingleton中去调用ObjectFactory的createBean
*/
// 下面会进入到createBean中去进行详细分析。
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, new ObjectFactory() {
        public Object getObject() throws BeansException {
            try {
                return createBean(beanName, mbd, args);
            }
            catch (BeansException ex) {
                /**
                *Explicitly remove instance from singleton cache: It
                *might have been put there eagerly by the creation
                *process, to allow for circular reference resolution.
                *Also remove any beans that received a temporary
                *reference to the bean.
                /*
                destroySingleton(beanName);
                throw ex;
            }
        }
    });
   bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
}
//这里是创建prototype bean的地方。
else if (mbd.isPrototype()) {
    // It's a prototype -> create a new instance.
    Object prototypeInstance = null;
    try {
        beforePrototypeCreation(beanName);
        prototypeInstance = createBean(beanName, mbd, args);
    }
    finally {
        afterPrototypeCreation(beanName);
    }
    bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
}
else {
    String scopeName = mbd.getScope();
    final Scope scope = this.scopes.get(scopeName);
```

Download at http://www.pin5i.com/

```
                if (scope == null) {
                    throw new IllegalStateException("No Scope registered for scope
                    '" + scopeName + "'");
                }
                try {
                    Object scopedInstance = scope.get(beanName, new ObjectFactory() {
                        public Object getObject() throws BeansException {
                            beforePrototypeCreation(beanName);
                            try {
                                return createBean(beanName, mbd, args);
                            }
                            finally {
                                afterPrototypeCreation(beanName);
                            }
                        }
                    });
                    bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
                }
                catch (IllegalStateException ex) {
                    throw new BeanCreationException(beanName,
                            "Scope '" + scopeName + "' is not active for the
                            current thread; " +"consider defining a scoped Proxy
                            for this bean if you intend to refer to it from a
                            singleton",ex);
                }
            }
        }
        // Check if required type matches the type of the actual bean instance.
        /**
        *这里对创建出来的bean进行类型检查，如果没有问题，就返回这个新创建出来的bean，
        *这个bean已经是包含了依赖关系的bean。
        */
        if (requiredType != null && bean != null && !requiredType.isAssignableFrom
        (bean.getClass())) {
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean. getClass());
        }
        return (T) bean;
    }
```

这个就是依赖注入的入口，在这里触发了依赖注入，而依赖注入的发生是在容器中的 BeanDefinition 数据已经建立好的前提下进行的。“程序=数据+算法”，很经典的一句话，前面的 BeanDefinition 就是数据，下面看看这些数据是怎样为依赖注入服务的。虽然依赖注入的过程不涉及复杂的算法问题，但这个过程也不简单，因为我们都知道，对于 IoC 容器的使用，Spring 提供了许多参数的配置，每一个参数配置实际上代表了一个 IoC 容器的实现特性，这些特性的实现很多都需要在依赖注入的过程中或者对 Bean 进行生命周期管理的过程中来完成。尽管我们可以用最简单的方式来描述 IoC 容器，说它就是一个 hashMap 而已。是的，我们可以这样说，但只能说这个 hashMap 是容器的最基本的数据结构，而不是 IoC 容器的全部。Spring IoC 容器的价值体现在一系列相关的产品特性上，这些产品特性以依赖反转模式的实现为核心，为用户更好地使用依赖反转提供便利，从而实现了一个完整的 IoC 容器产品。这些产品特性的实现并不是一个简单的过程，它提供了一个成熟的 IoC 容器产品来供用户使用。所以，尽管它没有什么独特的算法，但却可以看成是一个成功的软件工程产品，有许多值得我们学习的地方。

Download at http://www.pin5i.com/

关于这个依赖注入的详细过程我们会在下面进行分析。接着看 createBean 中的实现，在这个过程中，Bean 对象会依据 BeanDefinition 定义的要求生成出来。在 AbstractAutowireCapableBeanFactory 中实现了这个 createBean，createBean 不但生成了需要的 Bean，还对 Bean 初始化进行了处理，比如实现了在 BeanDefinition 中的 init-method 属性定义，Bean 后置处理器的实现，等等。具体的过程如代码清单 2-23 所示。

代码清单 2-23 AbstractAutowireCapableBeanFactory 中的 createBean

```
protected Object createBean(final String beanName, final RootBeanDefinition
mbd, final Object[] args)
        throws BeanCreationException {
    AccessControlContext acc = AccessController.getContext();
    return AccessController.doPrivileged(new PrivilegedAction<Object>() {
        public Object run() {
            if (logger.isDebugEnabled()) {
                logger.debug("Creating instance of bean '" + beanName + "'");
            }
            // Make sure bean class is actually resolved at this point.
            // 这里判断需要创建的bean 的是否可以实例化，这个类是否可以通过类装载器来载入。
            resolveBeanClass(mbd, beanName);

            // Prepare method overrides.
            try {
                mbd.prepareMethodOverrides();
            }
            catch (BeanDefinitionValidationException ex) {
                throw new BeanDefinitionStoreException(mbd.getResourceDescription(),
                        beanName, "Validation of method overrides failed", ex);
            }

            try {
                /**
                * Give BeanPostProcessors a chance to return a proxy
                * instead of thetarget bean instance.
                */
                // 如果bean 配置了 PostProcessor，那么这里返回的是一个 proxy。
                Object bean = resolveBeforeInstantiation(beanName, mbd);
                if (bean != null) {
                    return bean;
                }
            }
            catch (Throwable ex) {
                throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "BeanPostProcessor before instantiation of bean failed", ex);
            }
            //这里是创建bean 的调用。
            Object beanInstance = doCreateBean(beanName, mbd, args);
            if (logger.isDebugEnabled()) {
                logger.debug("Finished creating instance of bean '" + beanName + "'");
            }
            return beanInstance;
        }
    }, acc);
}
//我们接着到 doCreateBean 中去看看 bean 是怎样生成的:
protected Object doCreateBean(final String beanName, final RootBean Definition
```

Download at http://www.pin5i.com/

```
mbd, final Object[] args) {
// Instantiate the bean.
// 这个 BeanWrapper 是用来持有创建出来的 bean 对象的。
BeanWrapper instanceWrapper = null;
// 如果是 singleton，先把缓存中的同名 bean 清除。
if (mbd.isSingleton()) {
    instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
}
//这里是创建 bean 的地方，由 createBeanInstance 来完成。
if (instanceWrapper == null) {
    instanceWrapper = createBeanInstance(beanName, mbd, args);
}
final Object bean = (instanceWrapper != null ? instanceWrapper.get Wrapped
Instance() : null);
Class beanType = (instanceWrapper != null ? instanceWrapper.getWrapped Class() :
null);

// Allow post-processors to modify the merged bean definition.
synchronized (mbd.postProcessingLock) {
    if (!mbd.postProcessed) {
        applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        mbd.postProcessed = true;
    }
}
/**
* Eagerly cache singletons to be able to resolve circular references
* even when triggered by lifecycle interfaces like BeanFactoryAware.
*/
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircular
References && isSingletonCurrentlyInCreation(beanName));
if (earlySingletonExposure) {
    if (logger.isDebugEnabled()) {
        logger.debug("Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
    }
    addSingletonFactory(beanName, new ObjectFactory() {
        public Object getObject() throws BeansException {
            return getEarlyBeanReference(beanName, mbd, bean);
        }
    });
}

// Initialize the bean instance.
/**
* 这里是对 bean 的初始化，依赖注入往往在这里发生，这个 exposedObject 在初始化处理完以后会
* 返回作为依赖注入完成后的 bean。
*/
Object exposedObject = bean;
try {
    populateBean(beanName, mbd, instanceWrapper);
    exposedObject = initializeBean(beanName, exposedObject, mbd);
}
catch (Throwable ex) {
    if (ex instanceof BeanCreationException && beanName.equals(((BeanCreation
       Exception) ex).getBeanName())) {
        throw (BeanCreationException) ex;
    }
    else {
```

Download at http://www.pin5i.com/

```
                throw new BeanCreationException(mbd.getResourceDescription(),beanName,
                "Initialization of bean failed", ex);
            }
        }
        if (earlySingletonExposure) {
            Object earlySingletonReference = getSingleton(beanName, false);
            if (earlySingletonReference != null) {
                if (exposedObject == bean) {
                    exposedObject = earlySingletonReference;
                }
                else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean
        (beanName)) {
                    String[] dependentBeans = getDependentBeans(beanName);
                    Set<String> actualDependentBeans = new LinkedHashSet<String>
                    (dependentBeans.length);
                    for (String dependentBean : dependentBeans) {
                        if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                            actualDependentBeans.add(dependentBean);
                        }
                    }
                    if (!actualDependentBeans.isEmpty()) {
                        throw new BeanCurrentlyInCreationException(beanName,
                                "Bean with name '" + beanName + "' has been injected
                            into other beans [" +
                                StringUtils.collectionToCommaDelimitedString
                             (actualDependentBeans) + "] in its raw version as part
                                of a circular reference, but has eventually been " +
                            "wrapped. This means that said other beans do not
                                use the final version of the " +
                            "bean. This is often the result of over-eager type
                                matching - consider using " +
                            "'getBeanNamesOfType' with the 'allowEagerInit'
                                flag turned off, for example.");
                    }
                }
            }
        }
        // Register bean as disposable.
        try {
            registerDisposableBeanIfNecessary(beanName, bean, mbd);
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Invalid destruction signature", ex);
        }
        return exposedObject;
    }
```

这里我们看到与依赖注入关系特别密切的方法有 createBeanInstance 和 populateBean，下面分别到这两个方法里看看发生了什么。在 createBeanInstance 中生成了 Bean 所包含的 Java 对象，这个对象的生成有很多种不同的方式，可以通过工厂方法生成，也可以通过容器的 autowire 特性生成，这些生成方式都是由相关的 BeanDefinition 来指定的。如代码清单 2-24 所示，可以看到不同生成方式对应的实现。

代码清单 2-24 Bean 包含的 Java 对象的生成

```
    protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition
    mbd, Object[] args) {
```

Download at http://www.pin5i.com/

```
        // Make sure bean class is actually resolved at this point.
        // 确认需要创建的bean实例的类可以实例化。
        Class beanClass = resolveBeanClass(mbd, beanName);
        //这里使用工厂方法对bean进行实例化。
        if (mbd.getFactoryMethodName() != null)  {
            return instantiateUsingFactoryMethod(beanName, mbd, args);
        }
        // Shortcut when re-creating the same bean...
        if (mbd.resolvedConstructorOrFactoryMethod != null) {
            if (mbd.constructorArgumentsResolved) {
                return autowireConstructor(beanName, mbd, null, args);
            }
            else {
                return instantiateBean(beanName, mbd);
            }
        }
        // Need to determine the constructor...
        // 使用构造函数进行实例化。
        Constructor[] ctors = determineConstructorsFromBeanPostProcessors(beanClass,
        beanName);
        if (ctors != null ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_
                CONSTRUCTOR ||
                mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
            return autowireConstructor(beanName, mbd, ctors, args);
        }
        // No special handling: simply use no-arg constructor.
        // 使用默认的构造函数对bean进行实例化。
        return instantiateBean(beanName, mbd);
    }
    //我们看看最常见的实例化过程instantiateBean:
    protected BeanWrapper instantiateBean(String beanName, RootBeanDefinition mbd) {
    /**
    *使用默认的实例化的策略对bean进行实例化，默认的实例化策略是CglibSubclassingInstantiation
        *Strategy，也就是用cglib来对bean进行实例化。
        */
        //我们接着回去看看CglibSubclassingInstantiationStrategy的实现。
        try {
            Object beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, this);
            BeanWrapper bw = new BeanWrapperImpl(beanInstance);
            initBeanWrapper(bw);
            return bw;
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Instantiation of bean failed", ex);
        }
    }
```

在这里用到了cglib对Bean进行实例化，cglib是一个常用的字节码生成器的类库，它提供了一系列的API来提供Java的字节码生成和转换的功能。在Spring AOP中也是使用了cglib来对Java的字节码进行了增强。我们看看在IoC容器里是怎样使用cglib来生成Bean对象的，到类SimpleInstantiationStrategy中去看一下，这个Strategy是Spring用来生成Bean对象的默认类。它提供了两种实例化Java对象的方法，一种是通过BeanUtils，它使用了JDK的反射功能，一种是通过cglib来生成的，如代码清单2-25所示。

Download at http://www.pin5i.com/

代码清单 2-25　使用 SimpleInstantiationStrategy 生成 Java 对象

```
public class SimpleInstantiationStrategy implements InstantiationStrategy {
public Object instantiate(
        RootBeanDefinition beanDefinition, String beanName, BeanFactory owner) {
    // Don't override the class with CGLIB if no overrides.
    if (beanDefinition.getMethodOverrides().isEmpty()) {
    //这里取得指定的构造器或者生成对象的工厂方法来对 bean 进行实例化。
        Constructor constructorToUse = (Constructor)beanDefinition. resolvedConstructor
        OrFactoryMethod;
        if (constructorToUse == null) {
            Class clazz = beanDefinition.getBeanClass();
            if (clazz.isInterface()) {
                throw new BeanInstantiationException(clazz, "Specified class is
                an interface");
            }
            try {
               constructorToUse = clazz.getDeclaredConstructor((Class[]) null);
                beanDefinition.resolvedConstructorOrFactoryMethod =constructorToUse;
            }
            catch (Exception ex) {
                throw new BeanInstantiationException(clazz, "No default
                constructor found", ex);
            }
        }
    /**
    *通过 BeanUtils 进行实例化，这个 BeanUtils 的实例化通过 Constructor 来实例化 bean,
    *在 BeanUtils 中可以看到具体的调用 ctor.newInstance(args)。
    */
        return BeanUtils.instantiateClass(constructorToUse, null);
    }
    else {
        // Must generate CGLIB subclass.
        //使用 CGLIB 来实例化对象。
        return instantiateWithMethodInjection(beanDefinition, beanName, owner);
    }
}
```

在 cglibSubclassingInstantiationStrategy 中我们可以看到具体的实例化过程和 cglib 的使用方法，这里就不对 cglib 的使用进行过多的阐述了。如果读者有兴趣，可以去阅读 cglib 的使用文档，不过这里的 Spring 代码可以为使用 cglib 提供很好的参考。这里的 Enhancer 类，已经是 cglib 的类了，通过这个 Enhancer 来完成 Java 对象的生成，使用的是 Enhancer 的 create 方法。如代码清单 2-26 所示。

代码清单 2-26　使用 cglib 的 Enhancer 生成 Java 对象

```
public Object instantiate(Constructor ctor, Object[] args) {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(this.beanDefinition.getBeanClass());
    enhancer.setCallbackFilter(new CallbackFilterImpl());
    enhancer.setCallbacks(new Callback[] {
            NoOp.INSTANCE,
            new LookupOverrideMethodInterceptor(),
            new ReplaceOverrideMethodInterceptor()
    });
    //使用 cglib 的 create 生成实例化的 bean 对象。
    return (ctor == null) ?
            enhancer.create() :
```

Download at http://www.pin5i.com/

```
            enhancer.create(ctor.getParameterTypes(), args);
    }
```

在实例化 Bean 对象生成的基础上，我们看看 Spring 是怎样对这些对象进行处理的，也就是 Bean 对象生成以后，怎样把这些 Bean 对象的依赖关系设置好，完成整个依赖注入过程。这里涉及对各种 Bean 对象的属性的处理过程（即依赖关系处理的过程），这些依赖关系处理的依据就是已经解析得到的 BeanDefinition。详细地了解这个过程，需要回到前面的 populateBean 方法，这个方法在 AbstractAutowireCapableBeanFactory 中的实现如代码清单 2-27 所示。

代码清单 2-27　populateBean 的实现

```
protected void populateBean(String beanName, AbstractBeanDefinition mbd,
    BeanWrapper bw) {
    /**
    *这里取得在 BeanDefinition 中设置的 property 值，这些 property 来自对
    *BeanDefinition 的解析。
    */
    //具体的解析过程可以参看载入和解析 BeanDefinition 的分析。
    PropertyValues pvs = mbd.getPropertyValues();
    if (bw == null) {
        if (!pvs.isEmpty()) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Cannot apply property
                    values to null instance");
        }
        else {
            // Skip property population phase for null instance.
            return;
        }
    }
    /**
    * Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
    * state of the bean before properties are set. This can be used, for example,
    * to support styles of field injection.
    */
    boolean continueWithPropertyPopulation = true;
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp = (InstantiationAware
                BeanPostProcessor) bp;
                if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
    beanName)) {
                    continueWithPropertyPopulation = false;
                    break;
                }
            }
        }
    }
    if (!continueWithPropertyPopulation) {
        return;
    }
    //开始进行依赖注入过程，先处理 autowire 的注入。
    if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
            mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
        MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
        // Add property values based on autowire by name if applicable.
```

Download at http://www.pin5i.com/

```
            // 这里是对 autowire 注入的处理，根据 bean 的名字或者。type 进行 autowire 的过程。
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
                autowireByName(beanName, mbd, bw, newPvs);
            }
            // Add property values based on autowire by type if applicable.
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
                autowireByType(beanName, mbd, bw, newPvs);
            }
            pvs = newPvs;
        }
        boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
        boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.
        DEPENDENCY_CHECK_NONE);
        if (hasInstAwareBpps || needsDepCheck) {
            PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependency
            Check(bw);
            if (hasInstAwareBpps) {
                for (BeanPostProcessor bp : getBeanPostProcessors()) {
                    if (bp instanceof InstantiationAwareBeanPostProcessor) {
                        InstantiationAwareBeanPostProcessor ibp = (InstantiationAware
                        BeanPostProcessor) bp;
                        pvs = ibp.postProcessPropertyValues(pvs, filteredPds,
                        bw.getWrappedInstance(), beanName);
                        if (pvs == null) {
                            return;
                        }
                    }
                }
            }
            if (needsDepCheck) {
                checkDependencies(beanName, mbd, filteredPds, pvs);
            }
        }
        //对属性进行注入。
        applyPropertyValues(beanName, mbd, bw, pvs);
    }
    //我们到 applyPropertyValues 中去看看具体的对属性进行解析然后注入的过程：
    protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper
        bw, PropertyValues pvs) {
        if (pvs == null || pvs.isEmpty()) {
            return;
        }
        MutablePropertyValues mpvs = null;
        List<PropertyValue> original;
        if (pvs instanceof MutablePropertyValues) {
            mpvs = (MutablePropertyValues) pvs;
            if (mpvs.isConverted()) {
                // Shortcut: use the pre-converted values as-is.
                try {
                    bw.setPropertyValues(mpvs);
                    return;
                }
                catch (BeansException ex) {
                    throw new BeanCreationException(
                            mbd.getResourceDescription(), beanName, "Error setting
                            property values", ex);
                }
            }
            original = mpvs.getPropertyValueList();
```

Download at http://www.pin5i.com/

```
    }
    else {
        original = Arrays.asList(pvs.getPropertyValues());
    }
    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }
    /**
    *注意这个 BeanDefinitionValueResolver 对 BeanDefinition 的解析
    *是在这个 valueResolver 中完成的。
    */
    BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver
    (this, beanName, mbd, converter);
    // Create a deep copy, resolving any references for values.
    // 这里为解析值创建一个拷贝，拷贝的数据将会被注入到 bean 中。
   List<PropertyValue> deepCopy = new ArrayList<PropertyValue>(original.size());
    boolean resolveNecessary = false;
    for (PropertyValue pv : original) {
        if (pv.isConverted()) {
            deepCopy.add(pv);
        }
        else {
            String propertyName = pv.getName();
            Object originalValue = pv.getValue();
            Object resolvedValue = valueResolver.resolveValueIfNecessary(pv,
            originalValue);
            Object convertedValue = resolvedValue;
            boolean convertible = bw.isWritableProperty(propertyName) &&
            !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
            if (convertible) {
               convertedValue = convertForProperty(resolvedValue, propertyName,
               bw, converter);
            }
            /**
            * Possibly store converted value in merged bean definition,
            * in order to avoid re-conversion for every created bean instance.
            */
            if (resolvedValue == originalValue) {
                if (convertible) {
                    pv.setConvertedValue(convertedValue);
                }
                deepCopy.add(pv);
            }
            else if (originalValue instanceof TypedStringValue && convertible &&
                    !(convertedValue instanceof Collection ||
                    ObjectUtils.isArray(convertedValue))) {
                    pv.setConvertedValue(convertedValue);
                    deepCopy.add(pv);
            }
            else {
                resolveNecessary = true;
                deepCopy.add(new PropertyValue(pv, convertedValue));
            }
        }
    }
    if (mpvs != null && !resolveNecessary) {
        mpvs.setConverted();
    }
```

Download at http://www.pin5i.com/

```
        // Set our (possibly massaged) deep copy.
        // 这里是依赖注入发生的地方，会在 BeanWrapperImpl 中完成。
        try {
            bw.setPropertyValues(new MutablePropertyValues(deepCopy));
        }
        catch (BeansException ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Error setting property
                    values", ex);
        }
    }
```

这里通过使用 BeanDefinitionResolver 来对 BeanDefinition 进行解析，然后注入到 property 中。下面到 BeanDefinitionValueResolver 中去看一下解析过程的实现，我们举对 Bean reference 进行 Resolve 的例子，如图 2-9 所示，可以看到整个 Resolve 的过程，具体地对 Bean reference 进行 Resolve 的过程如代码清单 2-28 所示。

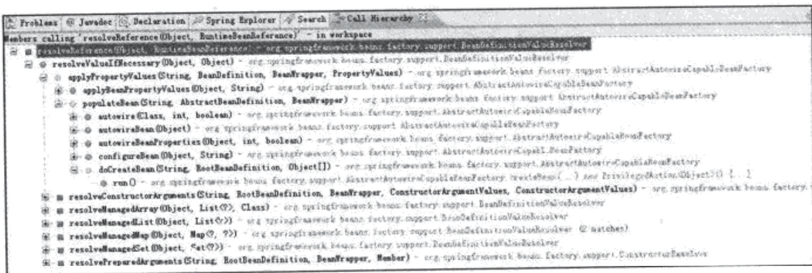


图 2-9　Resolve 的调用过程

代码清单 2-28　对 Bean Reference 的解析

```
private Object resolveReference(Object argName, RuntimeBeanReference ref) {
    try {
        /**
        *从 RuntimeBeanReference 取得 reference 的名字，这个 RuntimeBeanReference 是在载入
        *BeanDefinition 时根据配置生成的。
        */
            String refName = ref.getBeanName();
            refName = String.valueOf(evaluate(refName));
            //如果 ref 是在双亲 IoC 容器中，那就到双亲 IoC 容器中去取。
            if (ref.isToParent()) {
                if (this.beanFactory.getParentBeanFactory() == null) {
                    throw new BeanCreationException(
                            this.beanDefinition.getResourceDescription(), this.beanName,
                            "Can't resolve reference to bean '" + refName +
                            "' in parent factory: no parent factory available");
                }
                return this.beanFactory.getParentBeanFactory().getBean(refName);
            }
    /**
    *在当前 IoC 容器中去取 bean，这里会触发一个 getBean 的过程，如果依赖注入没有发生，
    *这里会触发相应的依赖注入的发生。
    */
            else {
```

Download at http://www.pin5i.com/

```
                Object bean = this.beanFactory.getBean(refName);
                this.beanFactory.registerDependentBean(refName, this.beanName);
                return bean;
            }
        }
        catch (BeansException ex) {
            throw new BeanCreationException(
                    this.beanDefinition.getResourceDescription(), this.beanName,
                    "Cannot resolve reference to bean '" + ref.getBeanName() + "'
                    while setting " + argName, ex);
        }
    }
//我们看看对其他类型的属性进行注入的例子，比如array和list等:
private Object resolveManagedArray(Object argName, List<?> ml, Class elementType) {
        Object resolved = Array.newInstance(elementType, ml.size());
        for (int i = 0; i < ml.size(); i++) {
            Array.set(resolved, i,
                resolveValueIfNecessary(
                        argName + " with key " + BeanWrapper.PROPERTY_KEY_
                        PREFIX + i + BeanWrapper.PROPERTY_KEY_SUFFIX,
                        ml.get(i)));
        }
        return resolved;
}
 // For each element in the managed list, resolve reference if necessary.
private List resolveManagedList(Object argName, List<?> ml) {
        List<Object> resolved = new ArrayList<Object>(ml.size());
        for (int i = 0; i < ml.size(); i++) {
            resolved.add(
                resolveValueIfNecessary(
                        argName + " with key " + BeanWrapper.PROPERTY_KEY_
                        PREFIX + i + BeanWrapper.PROPERTY_KEY_SUFFIX,
                        ml.get(i)));
        }
        return resolved;
}
```

这两种属性的注入都调用了 resolveValueIfNecessary，这个方法包含了所有对注入类型的处理，这个 resolveValueIfNecessary 的实现，如代码清单 2-29 所示。

代码清单 2-29 resolveValueIfNecessary 的实现

```
public Object resolveValueIfNecessary(Object argName, Object value) {
        /**
        * We must check each value to see whether it requires a runtime reference
        * to another bean to be resolved.
        */
        /**
        * 这里对 RuntimeBeanReference 进行解析，RuntimeBeanReference
        *是在对 BeanDefinition 进行解析是生成的数据对象。
        */
        if (value instanceof RuntimeBeanReference) {
            RuntimeBeanReference ref = (RuntimeBeanReference) value;
            return resolveReference(argName, ref);
        }
        else if (value instanceof RuntimeBeanNameReference) {
            String refName = ((RuntimeBeanNameReference) value).getBeanName();
            refName = String.valueOf(evaluate(refName));
            if (!this.beanFactory.containsBean(refName)) {
```

Download at http://www.pin5i.com/

```
                throw new BeanDefinitionStoreException(
                        "Invalid bean name '" + refName + "' in bean reference for " +
                        argName);
            }
            return refName;
        }
        else if (value instanceof BeanDefinitionHolder) {
            /**
            * Resolve BeanDefinitionHolder: contains BeanDefinition with name
            *and aliases.
            */
            BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
            return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.get
        BeanDefinition());
        }
        else if (value instanceof BeanDefinition) {
            // Resolve plain BeanDefinition, without contained name: use dummy name.
            BeanDefinition bd = (BeanDefinition) value;
            return resolveInnerBean(argName, "(inner bean)", bd);
        }
    //这里对ManageArray进行解析。
        else if (value instanceof ManagedArray) {
            // May need to resolve contained runtime references.
            ManagedArray array = (ManagedArray) value;
            Class elementType = array.resolvedElementType;
            if (elementType == null) {
                String elementTypeName = array.getElementTypeName();
                if (StringUtils.hasText(elementTypeName)) {
                    try {
                        elementType = ClassUtils.forName(elementTypeName, this.beanFactory.
                        getBeanClassLoader());
                         array.resolvedElementType = elementType;
                    }
                    catch (Throwable ex) {
                        // Improve the message by showing the context.
                        throw new BeanCreationException(
                                this.beanDefinition.getResourceDescription(),
                                        this.beanName,
                                "Error resolving array type for " + argName, ex);
                    }
                }
                else {
                    elementType = Object.class;
                }
            }
            return resolveManagedArray(argName, (List<?>) value, elementType);
        }
    //这里对ManageList进行解析。
        else if (value instanceof ManagedList) {
            // May need to resolve contained runtime references.
            return resolveManagedList(argName, (List<?>) value);
        }
    //这里对ManageSet进行解析。
        else if (value instanceof ManagedSet) {
            // May need to resolve contained runtime references.
            return resolveManagedSet(argName, (Set<?>) value);
        }
    //这里对ManageMap进行解析。
```

Download at http://www.pin5i.com/

```
        else if (value instanceof ManagedMap) {
            // May need to resolve contained runtime references.
            return resolveManagedMap(argName, (Map<?, ?>) value);
        }
    //这里对 ManageProperties 进行解析。
        else if (value instanceof ManagedProperties) {
            Properties original = (Properties) value;
            Properties copy = new Properties();
            for (Map.Entry propEntry : original.entrySet()) {
                Object propKey = propEntry.getKey();
                Object propValue = propEntry.getValue();
                if (propKey instanceof TypedStringValue) {
                    propKey = ((TypedStringValue) propKey).getValue();
                }
                if (propValue instanceof TypedStringValue) {
                    propValue = ((TypedStringValue) propValue).getValue();
                }
                copy.put(propKey, propValue);
            }
            return copy;
        }
    //这里对 TypedStringValue 进行解析。
        else if (value instanceof TypedStringValue) {
            // Convert value to target type here.
            TypedStringValue typedStringValue = (TypedStringValue) value;
            Object valueObject = evaluate(typedStringValue.getValue());
            try {
                Class resolvedTargetType = resolveTargetType(typedStringValue);
                if (resolvedTargetType != null) {
                    return this.typeConverter.convertIfNecessary(valueObject,
                    resolvedTargetType);
                }
                else {
                    return valueObject;
                }
            }
            catch (Throwable ex) {
                // Improve the message by showing the context.
                throw new BeanCreationException(
                        this.beanDefinition.getResourceDescription(),
                                this.beanName,
                        "Error converting typed String value for " + argName, ex);
            }
        }
        else {
            return evaluate(value);
        }
    }
    //对 runtimeBeanReference 类型的注入在 resolveReference 中:
    private Object resolveReference(Object argName, RuntimeBeanReference ref) {
        try {
    /**
    *从 RuntimeBeanReference 取得 reference 的名字，这个 RuntimeBeanReference
    *是在载入 BeanDefinition 时根据配置生成的。
    */
            String refName = ref.getBeanName();
            refName = String.valueOf(evaluate(refName));
    //如果 ref 是在双亲 IoC 容器中，那就到双亲 IoC 容器中去取。
```

Download at http://www.pin5i.com/

```
        if (ref.isToParent()) {
            if (this.beanFactory.getParentBeanFactory() == null) {
                throw new BeanCreationException(
                                this.beanDefinition.getResourceDescription(),
                                this.beanName,
                        "Can't resolve reference to bean '" + refName +
                        "' in parent factory: no parent factory available");
            }
            return this.beanFactory.getParentBeanFactory().getBean(refName);
        }
/**
*在当前 IoC 容器中去取 bean，这里会触发一个 getBean 的过程，如果依赖注入没有发生，
*这里会触发相应的依赖注入的发生。
*/
        else {
            Object bean = this.beanFactory.getBean(refName);
            this.beanFactory.registerDependentBean(refName, this.beanName);
            return bean;
        }
    }
    catch (BeansException ex) {
        throw new BeanCreationException(
                this.beanDefinition.getResourceDescription(), this.beanName,
                "Cannot resolve reference to bean '" + ref.getBeanName() + "'
                while setting " + argName, ex);
    }
}
//对 manageList 的处理过程在 resolveManagedList 中:
private List resolveManagedList(Object argName, List<?> ml) {
    List<Object> resolved = new ArrayList<Object>(ml.size());
    for (int i = 0; i < ml.size(); i++) {
        //递归的对 List 的元素进行 resolve。
        resolved.add(
           resolveValueIfNecessary(
                        argName + " with key " + BeanWrapper.PROPERTY_KEY_
                        PREFIX + i + BeanWrapper.PROPERTY_KEY_SUFFIX,
                        ml.get(i)));
    }
    return resolved;
}
```

在完成这个 resolve 过程后，就已经为依赖注入准备好了条件，这是真正把 Bean 对象设置到它所依赖的另一个 Bean 的属性中去的地方，其中处理的属性是各种各样的。依赖注入是在 BeanWrapper 的 setPropertyValues 中实现的，而具体的完成却是在 BeanWrapper 的子类 BeanWrapperImpl 中，如代码清单 2-30 所示。

代码清单 2-30 BeanWraper 完成 Bean 的属性值注入

```
private void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv) throws
    BeansException {
    String propertyName = tokens.canonicalName;
    String actualName = tokens.actualName;
    if (tokens.keys != null) {
       // Apply indexes and map keys: fetch value for all keys but the last one.
        PropertyTokenHolder getterTokens = new PropertyTokenHolder();
        getterTokens.canonicalName = tokens.canonicalName;
        getterTokens.actualName = tokens.actualName;
        getterTokens.keys = new String[tokens.keys.length - 1];
        System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.
```

Download at http://www.pin5i.com/

```
length - 1);
    Object propValue;
    //getPropertyValue 取得 bean 中注入对象的引用，比如 Array、List、Map、Set 等。
    try {
        propValue = getPropertyValue(getterTokens);
    }
    catch (NotReadablePropertyException ex) {
        throw new NotWritablePropertyException(getRootClass(), this.nested
                Path + propertyName,
                "Cannot access indexed value in property referenced " +
                "in indexed property path '" + propertyName + "'", ex);
    }
    // Set value for last key.
    String key = tokens.keys[tokens.keys.length - 1];
    if (propValue == null) {
        throw new NullValueInNestedPathException(getRootClass(), this.nested
                Path + propertyName,
                "Cannot access indexed value in property referenced " +
                "in indexed property path '" + propertyName + "': returned null");
    } //这里对 Array 进行注入。
    else if (propValue.getClass().isArray()) {
        Class requiredType = propValue.getClass().getComponentType();
        int arrayIndex = Integer.parseInt(key);
        Object oldValue = null;
        try {
            if (isExtractOldValueForEditor()) {
                oldValue = Array.get(propValue, arrayIndex);
            }
            Object convertedValue =
        this.typeConverterDelegate. convertIfNecessary(
                    propertyName, oldValue, pv.getValue(), requiredType);
            Array.set(propValue, Integer.parseInt(key), convertedValue);
        }
        catch (IllegalArgumentException ex) {
            PropertyChangeEvent pce =
                    new PropertyChangeEvent(this.rootObject,this.nestedPath
                    + propertyName, oldValue, pv.getValue());
            throw new TypeMismatchException(pce, requiredType, ex);
        }
        catch (IllegalStateException ex) {
            PropertyChangeEvent pce =
                    new PropertyChangeEvent(this.rootObject,this.nestedPath
                    + propertyName, oldValue, pv.getValue());
            throw new ConversionNotSupportedException(pce, requiredType, ex);
        }
        catch (IndexOutOfBoundsException ex) {
            throw new InvalidPropertyException(getRootClass(),this.nestedPath
+ propertyName,"Invalid array index in property path '" + propertyName
+ "'", ex);
        }
    } //这里对 List 进行注入。
    else if (propValue instanceof List) {
        PropertyDescriptor pd = getCachedIntrospectionResults().getPropertyDescriptor
        (actualName);
        Class requiredType =
        GenericCollectionTypeResolver. getCollectionReturnType(
                pd.getReadMethod(), tokens.keys.length);
        List list = (List) propValue;
        int index = Integer.parseInt(key);
```

Download at http://www.pin5i.com/

```
            Object oldValue = null;
            if (isExtractOldValueForEditor() && index < list.size()) {
                oldValue = list.get(index);
            }
            try {
                Object convertedValue = this.typeConverterDelegate. convertIf
Necessary(
                        propertyName, oldValue, pv.getValue(), requiredType);
                if (index < list.size()) {
                    list.set(index, convertedValue);
                }
                else if (index >= list.size()) {
                    for (int i = list.size(); i < index; i++) {
                        try {
                            list.add(null);
                        }
                        catch (NullPointerException ex) {
                            throw new InvalidPropertyException(getRootClass(),
                            this.nestedPath + propertyName,
                                    "Cannot set element with index " + index +
                            " in List of size " +
                                    list.size() + ", accessed using property
                            path '" + propertyName +
                                    "': List does not support filling up gaps
                            with null elements");
                        }
                    }
                    list.add(convertedValue);
                }
            }
            catch (IllegalArgumentException ex) {
                PropertyChangeEvent pce =
                        new PropertyChangeEvent(this.rootObject,this.nestedPath
                        + propertyName, oldValue, pv.getValue());
                throw new TypeMismatchException(pce, requiredType, ex);
            }
        } //里对 Map 进行注入。
        else if (propValue instanceof Map) {
            PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
            Class mapKeyType = GenericCollectionTypeResolver.getMapKeyReturnType(
                    pd.getReadMethod(), tokens.keys.length);
            Class mapValueType = GenericCollectionTypeResolver.getMapValueReturnType(
                    pd.getReadMethod(), tokens.keys.length);
            Map map = (Map) propValue;
            Object convertedMapKey;
            Object convertedMapValue;
            try {
                /**
                * IMPORTANT: Do not pass full property name in here - property editors
                * must not kick in for map keys but rather only for map values.
                */
                convertedMapKey = this.typeConverterDelegate.convertIfNecessary
                (key, mapKeyType);
            }
            catch (IllegalArgumentException ex) {
                PropertyChangeEvent pce =
                        new PropertyChangeEvent(this.rootObject,
this.nestedPath + propertyName, null, pv.getValue());
```

Download at http://www.pin5i.com/

```
                throw new TypeMismatchException(pce, mapKeyType, ex);
            }
            Object oldValue = null;
            if (isExtractOldValueForEditor()) {
                oldValue = map.get(convertedMapKey);
            }
            try {
                /**
                * Pass full property name and old value in here, since we want full
                * conversion ability for map values.
                */
                convertedMapValue = this.typeConverterDelegate.convertIfNecessary(
                        propertyName, oldValue, pv.getValue(), mapValueType, null,
                        new MethodParameter(pd.getReadMethod(), -1,
                        tokens.keys.length + 1));
            }
            catch (IllegalArgumentException ex) {
                PropertyChangeEvent pce =
                    new PropertyChangeEvent(this.rootObject, this.nested Path
+ propertyName, oldValue, pv.getValue());
                throw new TypeMismatchException(pce, mapValueType, ex);
            }
            map.put(convertedMapKey, convertedMapValue);
        }
        else {
            throw new InvalidPropertyException(getRootClass(), this.nestedPath +
            propertyName,
                    "Property referenced in indexed property path '" +propertyName +
                    "' is neither an array nor a List nor a Map; returned value
was [" + pv.getValue() + "]");
        }
}//这里对非集合类的域进行注入。
else {
    PropertyDescriptor pd = pv.resolvedDescriptor;
    if (pd == null|| !pd.getWriteMethod().getDeclaringClass().isInstance
(this.object)) {
        pd = getCachedIntrospectionResults().getPropertyDescriptor(actualName);
        if (pd == null || pd.getWriteMethod() == null) {
            PropertyMatches matches =PropertyMatches.forProperty(propertyName,
            getRootClass());
            throw new NotWritablePropertyException(
                    getRootClass(), this.nestedPath + propertyName,
                    matches.buildErrorMessage(), matches.getPossibleMatches());
        }
        pv.getOriginalPropertyValue().resolvedDescriptor = pd;
    }
    Object oldValue = null;
    try {
        Object originalValue = pv.getValue();
        Object valueToApply = originalValue;
        if (!Boolean.FALSE.equals(pv.conversionNecessary)) {
            if (pv.isConverted()) {
                valueToApply = pv.getConvertedValue();
            }
            else {
                if (isExtractOldValueForEditor() && pd.getReadMethod() !=null) {
                    Method readMethod = pd.getReadMethod();
                    if (!Modifier.isPublic(readMethod.getDeclaringClass().
getModifiers())) {
```

Download at http://www.pin5i.com/

```
                        readMethod.setAccessible(true);
                    }
                    try {
                        oldValue = readMethod.invoke(this.object);
                    }
                    catch (Exception ex) {
                        if (logger.isDebugEnabled()) {
                            logger.debug("Could not read previous value of
                            property '" +
                                    this.nestedPath + propertyName + "'", ex);
                        }
                    }
                }
                valueToApply = this.typeConverterDelegate.convertIfNecessary
                (oldValue,originalValue, pd);
            }
            pv.getOriginalPropertyValue().conversionNecessary =
            (valueToApply ! = originalValue);
        }
        //这里取得注入属性的set方法，通过反射机制，把对象注入进去。
        Method writeMethod = pd.getWriteMethod();
        if (!Modifier.isPublic(writeMethod.getDeclaringClass().getModifiers())) {
            writeMethod.setAccessible(true);
        }
        writeMethod.invoke(this.object, valueToApply);
    }
    catch (InvocationTargetException ex) {
        PropertyChangeEvent propertyChangeEvent =
                new PropertyChangeEvent(this.rootObject, this.nestedPath +
                propertyName, oldValue, pv.getValue());
        if (ex.getTargetException() instanceof ClassCastException) {
            throw new TypeMismatchException(propertyChangeEvent,
            pd.getPropertyType(),ex.getTargetException());
        }
        else {
            throw new MethodInvocationException(propertyChangeEvent,
ex.getTargetException());
        }
    }
    catch (IllegalArgumentException ex) {
        PropertyChangeEvent pce =
                new PropertyChangeEvent(this.rootObject, this.nestedPath +
                propertyName, oldValue, pv.getValue());
        throw new TypeMismatchException(pce, pd.getPropertyType(), ex);
    }
    catch (IllegalStateException ex) {
        PropertyChangeEvent pce =
                new PropertyChangeEvent(this.rootObject, this.nestedPath +
                propertyName, oldValue, pv.getValue());
        throw new ConversionNotSupportedException(pce, pd.getPropertyType(), ex);
    }
    catch (IllegalAccessException ex) {
        PropertyChangeEvent pce =
                new PropertyChangeEvent(this.rootObject, this.nestedPath +
                propertyName, oldValue, pv.getValue());
        throw new MethodInvocationException(pce, ex);
    }
  }
}
```

Download at http://www.pin5i.com/

这样就完成了对各种 bean 属性的依赖注入过程。笔者以前探寻过 Spring 2.0 的源代码，在这里发现 Spring 3.0 的源代码已经有了很大的改进，整个过程更为清晰了，特别是关于依赖注入的这一部分。如果读者有兴趣，可以分别研究并比较一下 Spring 2.0 和 Spring 3.0 对这部分的实现，这样可以更清晰地看到 Spring 源代码的演进过程，也可以看到 Spring 团队对代码进行重构的思路。

在 Bean 的创建和对象依赖注入的过程中，需要依据 BeanDefinition 中的信息来递归地完成依赖注入。从上面可以看到几个递归过程，这些递归都是以 getBean 为入口的。一个是在上下文体系中查找需要的 Bean 和创建 Bean 的递归调用；另一个递归在依赖注入时，通过递归调用容器的 getBean 方法，得到当前 Bean 的依赖 Bean，同时也触发对依赖 Bean 的创建和注入。在对 Bean 的属性进行依赖注入时，解析的过程也是一个递归的过程。这样，根据依赖关系，一层一层地完成 Bean 的创建和注入，直到最后完成当前 Bean 的创建，有了这个顶层 Bean 的创建和对它的属性依赖注入的完成，也意味着和当前 Bean 相关的整个依赖链的注入完成。

在 Bean 建立和依赖注入完成以后，在 IoC 容器中建立起一系列靠依赖关系联系起来的 Bean，这个 Bean 已经不是简单的 Java 对象了。这个 Bean 系列建立完成以后，通过 IoC 容器的相关接口方法，就可以非常方便地让上层应用使用了。回到前面的关于水桶的例子，到这里，我们不但找到了水源，而且成功地把水装到了水桶中，同时对水桶里的水完成了一系列的处理，比如消毒、煮沸……尽管还是水，但经过一系列的处理以后，这些水已经是开水了，已经可以直接饮用了！

2.5 容器其他相关特性的实现

在前面的 IoC 原理分析中，我们对 IoC 容器的主要功能进行了分析，比如 BeanDefinition 的载入和解析，依赖注入的实现，等等。为了更全面地理解 IoC 容器的特性，下面对容器的一些其他相关特性的实现原理也进行简要的分析，这些特性都是我们在使用容器时会经常遇到的。这些特性其实很多，这里只选择了几个例子供读者参考。在了解了 IoC 容器的整体运行原理以后，对这些特性的分析不再是一件困难的事情。如果读者对其他 IoC 容器的功能特性感兴趣，也可以按照相同的思路进行分析。

2.5.1 lazy-init 属性和预实例化

在 IoC 容器的初始化过程中，主要的工作是对 BeanDefinition 的定位、载入、解析和注册。此时依赖注入并没有发生，依赖注入发生在应用第一次向容器索要 Bean 时。向容器索要 Bean 是通过 getBean 的调用来完成的，这个 getBean 是容器提供 Bean 服务的最基本的接口。对于容器的初始化也有一种例外情况，就是用户可以通过设置 Bean 的 lazy-init 属性来控制预实例化的过程。这个预实例化在初始化容器时完成 Bean 的依赖注入，毫无疑问，这种容器的使用方式会对容器初始化的性能有一些影响，但却能够提高应用第一次取得 Bean 的性能。因为应用在第一次取得 Bean 时，依赖注入已经结束了，应用可以取到现成的 Bean。

Download at http://www.pin5i.com/

我们回头看看在上下文的初始化过程中，也就是 refresh 中的代码实现，可以看到预实例化是整个 refresh 初始化 IoC 容器的一个步骤。在 AbstractApplicationContext 中看看这个 refresh 方法的实现，这个初始化的过程在前面分析 IoC 容器初始化时已经分析过，只不过是从载入和注册 BeanDefinition 的角度进行分析的。

下面，我们将从 lazy-init 属性配置实现的角度进行分析。对这个属性的处理也是容器 refresh 的一部分，在 finishBeanFactoryInitialization 的方法中，封装了对 lazy-init 属性的处理，实际的处理是在 DefaultListableBeanFactory 这个基本容器的 preInstantiateSingletons 方法完成的。这个方法对单件 Bean 完成预实例化，这个预实例化的完成巧妙地委托给容器来实现。如果需要预实例化，那么就直接在这里采用 getBean 去触发依赖注入，与正常依赖注入的触发相比，只有触发的时间和场合不同。在这里，依赖注入发生在容器 refresh 的过程中，也就是发生在 IoC 容器初始化的过程中，而不像一般的依赖注入是发生在 IoC 容器初始化完成以后，第一次向容器 getBean 时。具体的实现脉络清晰而简洁，如代码清单 2-31 所示。

代码清单 2-31 refresh 中的预实例化

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();
        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);
        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);
            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);
            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);
            // Initialize message source for this context.
            initMessageSource();
            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();
            // Initialize other special beans in specific context subclasses.
            onRefresh();
            // Check for listener beans and register them.
            registerListeners();
            // Instantiate all remaining (non-lazy-init) singletons.
            // 这里是对lazy-init属性进行处理的地方。
            finishBeanFactoryInitialization(beanFactory);
            // Last step: publish corresponding event.
            finishRefresh();
        }
        catch (BeansException ex) {
            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();
            // Reset 'active' flag.
            cancelRefresh(ex);
            // Propagate exception to caller.
            throw ex;
        }
```

Download at http://www.pin5i.com/

```
        }
    }
    //我们到 finishBeanFactoryInitialization 中去看一下具体的处理过程:
    protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory
        beanFactory) {
        // Stop using the temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(null);
        // Allow for caching all bean definition metadata, not expecting further changes.
        beanFactory.freezeConfiguration();

        // Instantiate all remaining (non-lazy-init) singletons.
        /**
        * 这里调用的是 BeanFactory 的 preInstantiateSingletons，这个方法
        *是由 DefaultListableBeanFactory 实现的。
        */
        beanFactory.preInstantiateSingletons();
    }
    //在 DefaultListableBeanFactory 中的 preInstantiateSingletons 是这样的:
    public void preInstantiateSingletons() throws BeansException {
        if (this.logger.isInfoEnabled()) {
            this.logger.info("Pre-instantiating singletons in " + this);
        }
        //在这里就开始去 getBean 了，也就是去触发 bean 的依赖注入。
        /**
        *这个 getBean 和在上面分析的触发依赖注入的过程是一样的，只是发生的地方不同。
        *如果不设置 lazy-init 属性，那么这个依赖注入是发生在容器初始化结束以后。第一次
        *向容器 getBean 时，如果设置了 lazy-init 属性，那么依赖注入发生在容器初始化的过程中，
        *会对 beanDefinitionMap 中所有的 bean 进行依赖注入，这样在初始化过程结束以后，
        *向容器 getBean 得到的就是已经准备好的 bean，不需要进行依赖注入。
        */
        synchronized (this.beanDefinitionMap) {
            for (String beanName : this.beanDefinitionNames) {
                RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
                if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                    if (isFactoryBean(beanName)) {
                        FactoryBean factory = (FactoryBean)getBean(FACTORY_BEAN_
                        PREFIX + beanName);
                        if (factory instanceof SmartFactoryBean && ((SmartFactoryBean)
                        factory).isEagerInit()) {
                            getBean(beanName);
                        }
                    }
                    else {
                        getBean(beanName);
                    }
                }
            }
        }
    }
```

根据上面的分析得知，我们可以通过 lazy-init 属性来对整个 IoC 容器的初始化和依赖注入过程做一些简单的控制。这些控制是可以由容器的使用者来决定的，具体来说，可以通过在 BeanDefinition 中设置 lazy-init 属性来进行控制。这为我们使用容器提供了一定的灵活性，如果了解了这些控制原理，可以帮助我们更好地利用这些特性，希望这些小技巧能够对读者更好地使用容器提供帮助。

Download at http://www.pin5i.com/

2.5.2　FactoryBean 的实现

下面看看常见的工厂 Bean 是怎样实现的，这些 FactoryBean 为应用生成需要的对象，这些对象往往是经过特殊处理的，比如像 ProxyFactoryBean 这样的特殊 Bean。FactoryBean 的生产特性是在 getBean 中起作用的，我们看到下面的调用：

```
bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
```

我们看看这个 getObjectForBeanInstance 做了哪些处理，如图 2-10 所示，描述的是整个调用过程中涉及的方法。对 getObjectForBeanInstance 的实现，在这个方法里可以看到和我们在 FactoryBean 中常见的 getObject 方法的接口，详细的实现过程如代码清单 2-32 所示。

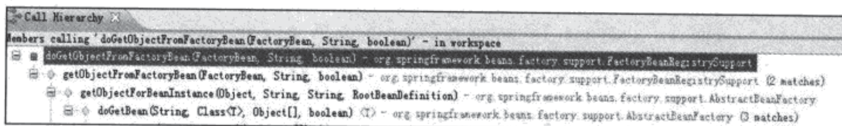


图 2-10　FactoryBean 生产方法的调用

代码清单 2-32　FactoryBean 特性的实现

```
protected Object getObjectForBeanInstance(
        Object beanInstance, String name, String beanName, RootBeanDefinition mbd) {
    /**
    * Don't let calling code try to dereference the factory if
    * the bean isn't a factory.
    */
    // 如果这里不是对 FactoryBean 的调用,那么结束处理。
    if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof
    FactoryBean)) {
        throw new BeanIsNotAFactoryException(transformedBeanName(name),beanInstance.
    getClass());
    }
    /**
    * Now we have the bean instance, which may be a normal bean or a FactoryBean.
    * If it's a FactoryBean, we use it to create a bean instance, unless the
    * caller actually wants a reference to the factory.
    */
    if (!(beanInstance instanceof FactoryBean) ||BeanFactoryUtils.isFactory
    Dereference(name)) {
        return beanInstance;
    }
    Object object = null;
    if (mbd == null) {
        object = getCachedObjectForFactoryBean(beanName);
    }
    if (object == null) {
        // Return bean instance from factory.
        FactoryBean factory = (FactoryBean) beanInstance;
        // Caches object obtained from FactoryBean if it is a singleton.
        if (mbd == null && containsBeanDefinition(beanName)) {
            mbd = getMergedLocalBeanDefinition(beanName);
```

Download at http://www.pin5i.com/

```
        }
        boolean synthetic = (mbd != null && mbd.isSynthetic());
        //这里从 FactoryBean 中得到 bean。
        object = getObjectFromFactoryBean(factory, beanName, !synthetic);
    }
    return object;
}
protected Object getObjectFromFactoryBean(FactoryBean factory, String beanName,
    boolean shouldPostProcess) {
    if (factory.isSingleton() && containsSingleton(beanName)) {
        synchronized (getSingletonMutex()) {
            Object object = this.factoryBeanObjectCache.get(beanName);
            if (object == null) {
                object = doGetObjectFromFactoryBean(factory, beanName,
    shouldPostProcess);
                this.factoryBeanObjectCache.put(beanName, (object != null ?
    object : NULL_OBJECT));
            }
            return (object != NULL_OBJECT ? object : null);
        }
    }
    else {
        return doGetObjectFromFactoryBean(factory, beanName, shouldPostProcess);
    }
}
private Object doGetObjectFromFactoryBean(
        final FactoryBean factory, final String beanName, final Boolean
        shouldPostProcess)
        throws BeanCreationException {
    AccessControlContext acc = AccessController.getContext();
    return AccessController.doPrivileged(new PrivilegedAction<Object>() {
        public Object run() {
            Object object;
            //这里调用 factory 的 getObject 方法来从 FactoryBean 中得到 bean。
            try {
                object = factory.getObject();
            }
            catch (FactoryBeanNotInitializedException ex) {
                throw new BeanCurrentlyInCreationException(beanName, ex.toString());
            }
            catch (Throwable ex) {
                throw new BeanCreationException(beanName, "FactoryBean threw
                exception on object creation", ex);
            }

            /**
            * Do not accept a null value for a FactoryBean that's not fully
            * initialized yet: Many FactoryBeans just return null then.
            */
            if (object == null && isSingletonCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(
                        beanName, "FactoryBean which is currently in creation
                        returned null from getObject");
            }
            if (object != null && shouldPostProcess) {
                try {
                    object = postProcessObjectFromFactoryBean(object, beanName);
                }
                catch (Throwable ex) {
```

Download at http://www.pin5i.com/

```
                    throw new BeanCreationException(beanName, "Post-processing
    of the FactoryBean's object failed", ex);
                }
            }
            return object;
        }
    }, acc);
}
```

这里返回的已经是作为工厂的 FactoryBean 生产的产品，并不是 FactoryBean 本身。这种 FactoryBean 的机制可以为我们提供一个很好的封装机制，比如封装 Proxy、RMI、JNDI 等。经过对 FactoryBean 实现过程的原理分析，相信读者会对 getObject 方法有很深刻的印象。这个方法就是主要的 FactoryBean 的接口，需要实现特定的工厂的生产过程，至于这个生产过程是怎样和 IoC 容器整合的，就是我们在上面分析的内容。

2.5.3 BeanPostProcessor 的实现

BeanPostProcessor 是使用 IoC 容器时经常会遇到的一个特性，这个 Bean 的后置处理器是一个监听器，它可以监听容器触发的事件。把它向 IoC 容器注册以后，使得容器中管理的 Bean 具备接收 IoC 容器事件回调的能力。BeanPostProcessor 的使用非常简单，只需要设计一个具体的后置处理器来实现。同时，这个具体的后置处理器需要实现接口类 BeanPostProcessor，然后设置到 XML 的 Bean 配置文件中。这个 BeanPostProcessor 是一个接口类，它有两个接口方法，一个是 postProcessBeforeInitialization，为在 Bean 的初始化前提供回调入口；一个是 postProcessAfterInitialization，为在 Bean 的初始化以后提供回调入口，这两个回调的触发都是和容器管理 Bean 的生命周期相关的。这两个回调方法的参数都是一样的，分别是 Bean 的实例化对象和 Bean 的名字，为具体的处理提供基本的回调输入，如代码清单 2-33 所示。

代码清单 2-33　BeanPostProcessor 接口定义

```
public interface BeanPostProcessor {
/**
 * Apply this BeanPostProcessor to the given new bean instance <i>before</i> any
 * bean initialization callbacks (like InitializingBean's
 * <code>afterPropertiesSet</code> or a custom init-method).The bean will already be
 * populated with property values. The returned bean instance may be a wrapper
 * around the original.
 * /
Object postProcessBeforeInitialization(Object bean, String beanName) throws
    BeansException;
/**
 * Apply this BeanPostProcessor to the given new bean instance <i>after</i> any bean
 * initialization callbacks (like InitializingBean's <code>
 * afterPropertiesSet</code> or a custom init-method). The bean will already be
 * populated with property values. The returned bean instance may be a wrapper
 * around the original.
 * <p>In case of a FactoryBean, this callback will be invoked for both the
   FactoryBean
 * instance and the objects created by the FactoryBean (as of Spring 2.0). The
 * post-processor can decide whether to apply to either the FactoryBean or
 * created objects or both through corresponding <code>bean instanceof FactoryBean
 * </code> checks.<p>This callback will also be invoked after a short-circuiting triggered by a
```

Download at http://www.pin5i.com/

```
     * {@link InstantiationAwareBeanPostProcessor#postProcessBeforeInstantiation} method,
     * in contrast to all other BeanPostProcessor callbacks.
     */
    Object postProcessAfterInitialization(Object bean, String beanName) throws
        BeansException;
    }
```

对于这些接口是在什么地方与 IoC 结合在一起的，可以看一下这个方法的调用关系，如图 2-11 所示。

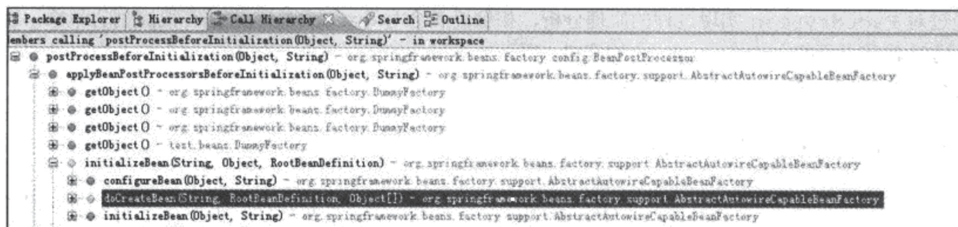


图 2-11　IoC 容器触发对 postProcessBeforeInitialization 接口的调用

postProcessBeforeInitialization 是在 populateBean 完成之后被调用的。我们从 BeanPostProcessor 中的一个回调接口入手，对另一个回调接口 postProcessAfter-Initialization 方法的调用，实际上也是在同一个地方封装完成的，这个地方就是 populateBean 方法中的 initializeBean 调用。关于这一点，读者会在接下来的分析中看得很清楚。在前面对 IoC 的依赖注入进行分析时，对这个 populateBean 有过分析，这个方法实际上完成了 Bean 的依赖注入，在容器中建立 Bean 的依赖关系，是容器功能实现的一个很重要的部分。节选 doCreateBean 中的代码就可以看到 postProcessBeforeInitialization 调用和 populateBean 调用的关系，如下所示。

```
    Object exposedObject = bean;
    try {
        populateBean(beanName, mbd, instanceWrapper);
        /**
        *在对 Bean 的生成和依赖注入完成以后，开始对 Bean 进行初始化，这个初始化过程
        *包含了对后置处理器的 postProcessBeforeInitialization 回调。
        */
        exposedObject = initializeBean(beanName, exposedObject, mbd);
    }
```

具体的初始化过程也是 IoC 容器完成依赖注入的一个重要部分。在 initializeBean 方法里，需要使用 Bean 的名字，完成依赖注入以后的 Bean 对象，以及这个 Bean 对应的 BeanDefinition。在这些输入的帮助下，完成 Bean 的初始化工作，这些工作包括为类型是 BeanNameAware 的 Bean 设置 Bean 的名字，为类型是 BeanClassLoaderAware 的 Bean 设置类装载器，为类型是 BeanFactoryAware 的 Bean 设置其自身所在的 IoC 容器以供回调使用。当然，还有对 postProcessBeforeInitialization/postProcessAfterInitialization 的回调和初始化属性 init-method 的处理等。经过这一系列的初始化处理之后，得到的结果就是我们

Download at http://www.pin5i.com/

可以正常使用的由 IoC 容器托管的 Bean 了。具体的实现过程如代码清单 2-34 所示。

代码清单 2-34 IoC 容器对 Bean 的初始化

```
/**
 * Initialize the given bean instance, applying factory callbacks
 * as well as init methods and bean post processors.
 * <p>Called from {@link #createBean} for traditionally defined beans,
 * and from {@link #initializeBean} for existing bean instances.
 */
protected Object initializeBean(String beanName, Object bean, RootBeanDefinition
    mbd) {
    if (bean instanceof BeanNameAware) {
        ((BeanNameAware) bean).setBeanName(beanName);
    }
    if (bean instanceof BeanClassLoaderAware) {
        ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());
    }
    if (bean instanceof BeanFactoryAware) {
        ((BeanFactoryAware) bean).setBeanFactory(this);
    }
    /**
    *这里是对后置处理器 BeanPostProcessors 的 postProcessBeforeInitialization
    *的回调方法调用。
    */
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
    beanName);
    }
    /**
    *调用 Bean 的初始化方法，这个初始化方法是在 BeanDefinition 中通过定义 init-method 属性指定的。
    *同时，如果 Bean 实现了 InitializingBean 接口，那么 Bean 的 afterPropertiesSet 实现也会被调用。
    */
    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
                (mbd != null ? mbd.getResourceDescription() : null),
                beanName, "Invocation of init method failed", ex);
    }
    /**
    *这里是对后置处理器 BeanPostProcessors 的 postProcessAfterInitialization
    *的回调方法调用。
    */
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }
    return wrappedBean;
}
/**
*对设置好的 BeanPostProcessors 的 postProcessBeforeInitialization 回调进行依次调用。
*/
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean,
    String beanName)
        throws BeansException {
    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
```

Download at http://www.pin5i.com/

```
            result = beanProcessor.postProcessBeforeInitialization(result, beanName);
        }
        return result;
    }
    /**
    *对设置好的BeanPostProcessors的postProcessAfterInitialization回调进行依次调用。
    */

    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean,
        String beanName)
            throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
            result = beanProcessor.postProcessAfterInitialization(result, beanName);
        }
        return result;
    }
```

从以上的代码实现中可以看到，这两个 Bean 后置处理器定义的接口方法，围绕着 Bean 定义的 init-method 方法调用，与 IoC 容器对 Bean 的管理有机地结合起来了。对于这个特性的理解，离不开对 IoC 容器基本实现原理的理解。了解了 Bean 后置处理器的实现原理后，就能更灵活地使用它。这些 IoC 容器的附加特性还有很多，它们代表了容器的一些特色和高级的使用技巧，掌握这些特性对应用开发有很大的帮助。下面就分析一下容器的 autowiring 特性是怎样实现的。

2.5.4 autowiring 的实现原理

在前面对 IoC 容器实现原理的分析中，一直是通过 BeanDefinition 的属性值和构造函数以显式的方式实现 Bean 的依赖关系管理的。在 Spring 中，相对这种显式的依赖管理方式，IoC 容器还提供了自动依赖装配的方式，为应用使用容器提供更大的方便。在自动装配中，不需要对 Bean 属性做显式的依赖关系声明，只需要配置好 autowire（自动依赖装配）属性，IoC 容器会根据这个属性的配置，使用反射自动地查找属性的类型或名字，然后基于属性的类型或名字来自动匹配 IoC 容器中的 Bean，从而自动地完成依赖注入。

这是一个很有诱惑力的功能特性，使用它可以完成依赖关系管理的自动化，但是使用时一定要注意，计算机只是在自动地执行，它是不会思考的。使用这个特性的优点是能够减少用户配置 Bean 的工作量，但如果使用不当，也会为应用带来不可预见的后果，就像我们使用其他方式的自动化一样。所以，使用起来需要多一些小心和谨慎。

从 autowiring 使用上可以知道，这个 autowiring 属性是在对 Bean 属性进行依赖注入时起作用的。对 Bean 属性依赖注入的实现原理，我们在前面已经做过分析。回顾那部分的内容，我们不难发现，对 autowirng 属性的处理，从而完成对 Bean 的属性的自动依赖装配。节选 AbstractAutowireCapableBeanFactory 的 populateBean 方法中与 autowiring 实现相关的部分，我们可以清楚地看到这个特性在容器中实现的入口。也就是说，对属性 autowire 的处理是 populateBean 处理过程的一个部分。在 populateBean 的实现中，在处理一般的 Bean 之前，先对 autowiring 属性进行处理。如果当前的 Bean 配置了 autowire_by_name 和 autowire_by_type 属性，那么调用相应的 autowireByName 方法和 autowireByType 方

Download at http://www.pin5i.com/

法。这两个方法很巧妙地应用了 IoC 容器的特性。例如，对于 autowire_by_name，它首先通过反射机制从当前 Bean 中得到需要注入的属性名，然后使用这个属性名向容器去申请与之同名的 Bean，这样实际又触发了另一个 Bean 的生成和依赖注入的过程。实现过程如代码清单 2-35 所示。

代码清单 2-35 populateBean 对 autowire 的处理

```
//开始进行依赖注入过程，先处理 autowire 的注入。
if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
        mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
    MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
    // Add property values based on autowire by name if applicable.
    // 这里是对 autowire 注入的处理，根据 bean 的名字或者 type 进行 autowire 的过程。
    if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
        autowireByName(beanName, mbd, bw, newPvs);
    }
    // Add property values based on autowire by type if applicable.
    if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
        autowireByType(beanName, mbd, bw, newPvs);
    }
    pvs = newPvs;
}
```

在做了一些简单的对 autowiring 类型的逻辑判断以后，通过调用 autowireByName 和 autowireByType 来完成自动依赖装配。以 autowireByName 为例来看看容器的自动依赖装配功能是怎样实现的；对 autowireByName 来说，它首先需要得到当前 Bean 的属性名，这些属性名已经在 BeanWrapper 和 BeanDefinition 中封装好了，然后就是对着一系列属性名进行匹配的过程。在这个匹配的过程中，因为已经有了属性的名字，那就可以直接使用这个属性名字作为 Bean 名字向容器 getBean，这个 getBean 会触发当前 Bean 的依赖 Bean 的依赖注入的过程；从而得到属性对应的依赖 Bean。在这个 getBean 完成后，把这个依赖 Bean 注入到当前 Bean 的属性中去；这样就完成了这个依赖属性名自动完成依赖注入的过程。对 autowireByType 的实现，和 autowireByName 的实现过程是非常类似的，感兴趣的读者可以自己进行分析。这些 autowiring 的实现如代码清单 2-36 所示。

代码清单 2-36 autowire_by_name 的实现

```
protected void autowireByName(
    String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutableProperty
    Values pvs) {
    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
    for (String propertyName : propertyNames) {
        if (containsBean(propertyName)) {
            /**
            *使用取得的当前 Bean 的属性名作为 Bean 的名字，向 IoC 容器 getBean，
            *然后把从容器得到的 Bean 设置到当前 Bean 的属性中去。
            */
            Object bean = getBean(propertyName);
            pvs.addPropertyValue(propertyName, bean);
            registerDependentBean(propertyName, beanName);
            if (logger.isDebugEnabled()) {
                logger.debug(
                    "Added autowiring by name from bean name '" + beanName
                    + "' via property '" + propertyName +
                    "' to bean named '" + propertyName + "'");
            }
```

Download at http://www.pin5i.com/

```
        }
        else {
            if (logger.isTraceEnabled()) {
                logger.trace("Not autowiring property '" + propertyName + "' of
                bean '" + beanName + "' by name: no matching bean found");
            }
        }
    }
}
```

2.6 小结

在本章中，我们紧密地结合 Spring 的源代码，对容器的实现原理进行了详细的分析，旨在为读者整理出一条清晰的线索。其中包括 IoC 容器和上下文的基本工作原理、容器的初始化过程、依赖注入的实现，等等。总的来说，关于容器的基本工作原理，我们可以大致地整理出以下几个方面：

- BeanDefinition 的定位。对 IoC 容器来说，它为我们管理 POJO 之间的依赖关系提供了帮助，但客户也需要依据 Spring 的定义规则提供 Bean 定义信息，我们可以使用各种形式的 Bean 定义信息，其中常用的是使用 XML 的文件格式。在 Bean 定义方面，Spring 为用户提供了很大的灵活性。在初始化 IoC 容器的过程中，首先需要定位到这些有效的 Bean 定义信息，这里 Spring 使用 Resource 这个接口来统一这些 Bean 定义信息，而这个定位由 ResourceLoader 来完成。如果使用上下文，ApplicationContext 本身就为客户提供了定位的功能。因为上下文本身就是 DefaultResourceLoader 的子类。如果使用基本的 BeanFactory 作为 IoC 容器，客户需要做的额外工作就是为 BeanFactory 指定相应的 Resource 来完成 Bean 信息的定位。
- 容器的初始化。在使用上下文时，需要一个对它进行初始化的过程，完成初始化以后，这个 IoC 容器才是可用的。这个过程的入口是在 refresh 中实现的，这个 refresh 相当于容器的初始化函数。在初始化过程中，比较重要的部分是对 BeanDefinition 信息的载入和注册工作。相当于在 IoC 容器中需要建立一个 BeanDefinition 定义的数据映像，Spring 为了达到载入的灵活性，把载入的功能从 IoC 容器中分离出来，由 BeanDefinitionReader 来完成 Bean 定义信息的读取、解析和 IoC 容器内部 BeanDefinition 的建立。在 DefaultListableBeanFactory 中，这些 BeanDefinition 被维护在一个 HashMap 中，以后的 IoC 容器对 Bean 的管理和操作就是通过这些建立起来的 BeanDefinition 来完成的。

在容器初始化完成以后，IoC 容器的使用就准备好了，这时只是在 IoC 容器内部建立了 BeanDefinition，具体的依赖关系还没有注入。在客户第一次向 IoC 容器请求 Bean 时，IoC 容器对相关的 Bean 依赖关系进行注入。如果需要提前注入，客户通过 lazy-init 属性可以进行预实例化，这个预实例化是上下文初始化的一部分，起到提前完成依赖注入的控制作用。在依赖注入完成以后，IoC 容器就会保持这些具备依赖关系的 Bean 让客户来直接使用。这时可以通过 getBean 来取得 Bean，这些 Bean 不是简单的 Java 对象，而是已经包含了对象之间依赖关系的 Bean，尽管这些依赖注入的过程对用户来说是不可见的。

Download at http://www.pin5i.com/

在对 IoC 容器的分析中，我们重点讲解了 BeanFactory 和 ApplicationContext 体系、ResourceLoader、refresh 初始化、容器的 loadBeanDefinition 和注册、容器的依赖注入、预实例化和 FactoryBean 的工作原理等。通过对这些实现过程的深入分析，我们可以初步了解 IoC 容器的基本工作原理和它的基本特性的实现思路。了解了 IoC 容器的基本实现原理后，我们对容器的其他特性的实现原理也进行了分析。这些特性包括 init-lazy 预实例化、BeanFactory、Bean 后置处理器以及 autowiring 特性的实现。这些特性对我们更灵活和更有技巧地使用 IoC 容器有很大的帮助。但是，由于 Spring IoC 容器的内涵特性非常丰富，这里并没有对其工作原理进行面面俱到的分析。如果读者感兴趣，可以参考本章的分析方法和思路，对自己感兴趣的内容继续进行分析。

Download at http://www.pin5i.com/

第3章

Spring AOP的实现

好雨知时节，当春乃发生。
随风潜入夜，润物细无声。
野径云俱黑，江船火独明。
晓看红湿处，花重锦官城。
——【唐】杜甫《春夜喜雨》

3.1 Spring AOP概述

3.1.1 AOP概念回顾

AOP是Aspect-Oriented Programming（面向方面编程）的简称，维基百科对它的解释如下所示。

> **维基百科对“AOP”相关概念的叙述**
>
> Aspect是一种新的模块化机制，用来描述分散在对象、类或函数中的横切关注点（crosscutting concern）。从关注点中分离出横切关注点是面向侧面的程序设计的核心概念。分离关注点使得解决特定领域问题的代码从业务逻辑中独立出来，业务逻辑的代码中不再含有针对特定领域问题代码的调用，业务逻辑同特定领域问题的关系通过侧面来封装、维护，这样原本分散在整个应用程序中的变动就可以很好地管理起来。

这里提到的概念是从模块化出发的，面向对象设计其实也是一种模块化的方法，它把相关的数据及其处理方法放在了一起，与单纯的使用子函数进行封装相比，面向对象的模块化特性更完备，它体现了计算的一个基本原则——让计算尽可能地靠近数据。这样一来，代码组织起来就更加整齐和清晰，一个类就是一个基本的模块。很多程序的功能还可以通过设计类的继承关系而得到重用，进一步提高了开发效率。后来又出现了各种各样的设计模式供我们使用，让我们在设计程序功能时更加得心应手。

虽然我们利用面向对象的方法可以很好地组织代码，也可以通过继承关系实现代码重用，但是程序中总是会出现一些重复的代码，而且不太方便使用继承的方法把它们重用和管理起来。它们功能重复并且需要作用在不同的地方，虽然可以对这些代码做一些简单的封装，使之成为一些公共函数，但是在这种显式的调用中，使用它们并不是很方便。例如，这个公共函数在什么情况下可以使用，能不能更灵活地使用，等等。

在使用这些公共函数时，往往也需要进行一些逻辑设计。也就是需要代码实现来支持，而这些逻辑代码也是需要维护的。这时就是AOP大显身手的时候了，使用AOP后，不仅可以将这些重复

Download at http://www.pin5i.com/

的代码抽取出来单独维护，在需要使用时统一调用这些公共代码，还可以为如何使用这些公共代码提供丰富灵活的手段。这虽然与设计公共子模块有几分相似，但在传统的公共子模块调用中，除了直接硬调用之外并没有其他的手段，而 AOP 为处理这一类问题提供了一套完整的理论和灵活多样的实现方法。也就是说，通过 AOP 提出横切的概念以后，在把模块功能正交化的同时，也在此基础上提供了一系列横切的灵活实现。比如通过使用 Proxy 代理对象、拦截器、字节码翻译技术等，通过这一系列已有的 AOP 或者 AOP 实现技术，来实现切面应用的各种编织实现和环绕增强；为了更好地应用 AOP 技术，技术专家们还成立了 AOP 联盟来探讨 AOP 的标准化，有了这些支持，AOP 的发展就更快了。关于 AOP 技术，可以到 AOP 联盟的文档里找到一些相关的介绍，从而提高我们对 AOP 的理解。比如在 AOP 联盟的网站上提到的以下 AOP 技术：

- AspectJ：源代码和字节码级别的编织器，用户需要使用不同于 Java 的新语言。
- AspectWerkz：AOP 框架，使用字节码动态编织器和 XML 配置。
- JBoss-AOP：基于拦截器和元数据的 AOP 框架，运行在 JBoss 应用服务器上。

以及在 AOP 中使用到的一些相关的技术实现：

- BCEL（Byte-Code Engineering Library）：Java 字节码操作类库，具体的信息可以参见项目网站：http://jakarta.apache.org/bcel/。
- Javassist：Java 字节码操作类库，JBoss 的一个子项目，项目信息可以参见项目网站：http://jboss.org/javassist/。

对应于现有的 AOP 实现方案，AOP 联盟对它们进行了一定程度的抽象，从而定义出 AOP 体系结构，结合这个体系结构去了解 AOP 技术，对我们理解 AOP 的概念是非常有帮助的，这个 AOP 体系结构如图 3-1 所示。

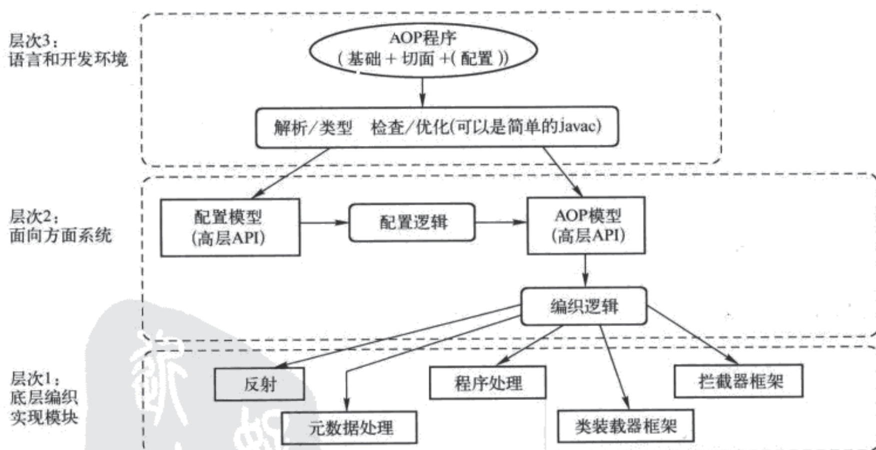


图 3-1 AOP 联盟定义的 AOP 体系结构

AOP 联盟定义的 AOP 体系结构把与 AOP 相关的概念大致分为了由高到低、从使用到实现的三个层次。从上往下，最高层是语言和开发环境，在这个环境中可以看到几个重要的概念：基础可

Download at http://www.pin5i.com/

以视为待增强对象或者说目标对象；切面通常包含对于基础的增强应用；配置可以看成是一种编织或者说配置，通过在 AOP 体系中提供这个配置环境，可以把基础和切面结合起来，从而完成切面对目标对象的编织实现。

在 Spring AOP 实现中，是使用 Java 语言来实现增强对象与切面增强应用的，并为这两者的结合提供了配置环境。对于编织配置，可以使用 IoC 容器来完成；对于 POJO 对象的配置，本来就是 Spring 的核心 IoC 容器的强项。因此，对于使用 Spring 的 AOP 开发而言，使用 POJO 就能完成 AOP 任务。但是，对于其他的 AOP 实现方案，可能需要使用特定的实现语言、配置环境甚至是特定的编译环境。例如在 AspectJ 中，尽管切面增强的对象是 Java 对象，但却需要使用特定的 Aspect 语言和 AspectJ 编译器。AOP 体系结构的第二个层次是为语言和开发环境提支持的，在这个层次中可以看到 AOP 框架的高层实现，主要包括配置和编织实现两部分内容。例如配置逻辑和编织逻辑实现本身，以及对这些实现进行抽象的一些高层 API 封装。这些实现和 API 封装，为前面提到的语言和开发环境的实现提供了有力的支持。

最底层是编织的具体实现模块，图 3-1 中看到的各种技术，都可以作为编织逻辑的具体实现方法，比如反射、程序处理、拦截器框架、类装载框架、元数据处理等。阅读完本章对 Spring AOP 实现原理的分析，我们会了解到，在 Spring AOP 中，使用的是 Java 本身的语言特性，比如 Java Proxy 代理类、拦截器这些技术，来完成 AOP 编织的实现。

对 Spring 平台或者说生态系统来说，AOP 是 Spring 框架的核心功能模块之一。AOP 与 IoC 容器的结合使用，为应用开发或者 Spring 自身功能的扩展都提供了许多便利。Spring AOP 的实现和其他特性的实现一样，除了可以使用 Spring 本身提供的 AOP 实现之外，还封装了业界优秀的 AOP 解决方案 AspectJ 来让应用使用。在本章中，我们主要对 Spring 自身的 AOP 实现原理进行分析；在这个 AOP 实现中，Spring 充分利用了 IoC 容器 Proxy 代理对象以及 AOP 拦截器的功能特性，通过这些对 AOP 基本功能的封装机制，为用户提供了 AOP 的实现框架。所以，要了解这些 AOP 的基本实现，需要我们对 Java 的 Proxy 机制有一些基本了解。在 Spring 中，有一些相关的概念与 AOP 设计相对应；在本章的内容中，按照笔者个人的粗浅理解，我们会结合 Spring 的 AOP 实现，温故而知新，在以下的小节中对一些相关的 AOP 概念，先简单地回顾一下；然后再逐步展开对 AOP 实现原理的分析，包在这些实现原理的分析中，包括了代理对象的生成，AOP 拦截器的实现，等等。这里以 ProxyFactoryBean 和 ProxyFactory 为例子来进行说明。通过分析它们背后隐藏的实现原理来了解 Spring 的 AOP 模块。和前面一样，我们需要在 Eclipse 环境中导入对应的 AOP 代码模块来辅助这些原理实现的分析，具体需要导入的包是：org.springframework.aop，这个代码包里包含了 Spring AOP 的实现代码。

3.1.2 Advice 通知

Advice（通知）：定义在连接点做什么，为切面增强提供织入接口。在 Spring AOP 中，它主要描述 Spring AOP 围绕方法调用而注入的切面行为。Advice 是 AOP 联盟定义的一个接口，具体的接口定义在 org.aopalliance.aop.Advice 中。在 Spring AOP 的实现中，使用了这个统一接口，并通过这个接口为 AOP 切面增强的织入功能做了更多的细化和扩展，比如提供了更具体的通知类型，像 BeforeAdvice、AfterAdvice、ThrowAdvice 等。作为 Spring AOP 定义的接口类，具体的切面增强可

Download at http://www.pin5i.com/

以通过这些接口集成到 AOP 框架中去发挥作用。对于这些接口类，我们下面会进行逐一详细地讨论，我们从接口 BeforeAdvice 开始，首先了解它的类层次关系，如图 3-2 所示。

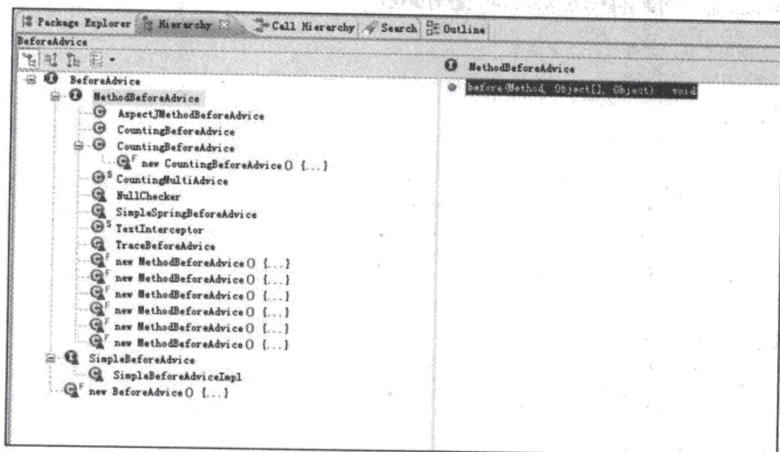


图 3-2 BeforeAdvice 的类层次关系

在 BeforeAdvice 的继承关系中，定义了为待增强的目标方法设置的前置增强接口 MethodBeforeAdvice，使用这个前置接口需要实现一个回调函数：

```
void before(Method method, Object[] args, Object target) throws Throwable;
```

作为回调函数，before 方法的实现在 advice 中被配置到目标方法后，会在调用目标方法时被回调。具体的调用参数有：Method 对象，这个参数是目标方法的反射对象；object[]对象数组，这个对象数组中包含的是目标方法的输入参数。以 CountingBeforeAdvice 为例，用它来说明 BeforeAdvice 的具体使用，CountingBeforeAdvice 是接口 MethodBeforeAdvice 的具体实现，如代码清单 3-1 所示，它的实现比较简单，完成的工作是统计被调用的方法次数。作为切面增强实现，它会根据调用方法的方法名进行统计，把统计结果根据方法名和调用次数作为键值对放入一个 map 中。

代码清单 3-1 CountingBeforeAdvice 的实现

```
public class CountingBeforeAdvice extends MethodCounter implements MethodBeforeAdvice {
//实现 before 回调接口，这是接口 MethodBeforeAdvice 的要求。
public void before(Method m, Object[] args, Object target) throws Throwable {
    count(m);
}
}
```

这里调用了 count 方法，使用了目标方法的反射对象作为参数，完成对调用方法名的统计工作。count 方法在 CountingBeforeAdvice 的基类 MethodCounter 中实现，如代码清单 3-2 所示。这个切面增强完成的统计实现并不复杂，它在对象中维护一个哈希表，用来存储统计数据。在统计过程中，首先通过目标方法的反射对象得到方法名，然后进行累加，把统计结果放到维护的哈希表中。如果需要统计数据，就到这个哈希表中根据 key 来获取。

Download at http://www.pin5i.com/

代码清单 3-2 MethodCounter 实现统计目标方法调用次数

```
public class MethodCounter implements Serializable {
// 这个 Hashmap 用来存储方法名和调用次数的键值对。
private HashMap<String, Integer> map = new HashMap<String, Integer>();
//所有的调用次数，不管是什么方法名。
private int allCount;
//CountingBeforeAdvice 的调用入口。
protected void count(Method m) {
    count(m.getName());
}
//根据目标方法的方法名统计调用次数。
protected void count(String methodName) {
    Integer i = map.get(methodName);
    i = (i != null) ? new Integer(i.intValue() + 1) : new Integer(1);
    map.put(methodName, i);
    ++allCount;
}
//根据方法名取得调用的次数。
public int getCalls(String methodName) {
    Integer i = map.get(methodName);
    return (i != null ? i.intValue() : 0);
}
//取得所有的方法调用次数。
public int getCalls() {
    return allCount;
}
public boolean equals(Object other) {
    return (other != null && other.getClass() == this.getClass());
}

public int hashCode() {
    return getClass().hashCode();
}
}
```

在 Advice 的实现体系中，Spring 还提供了 AfterAdvice 的这种通知类型，它的类接口关系如图 3-3 所示。

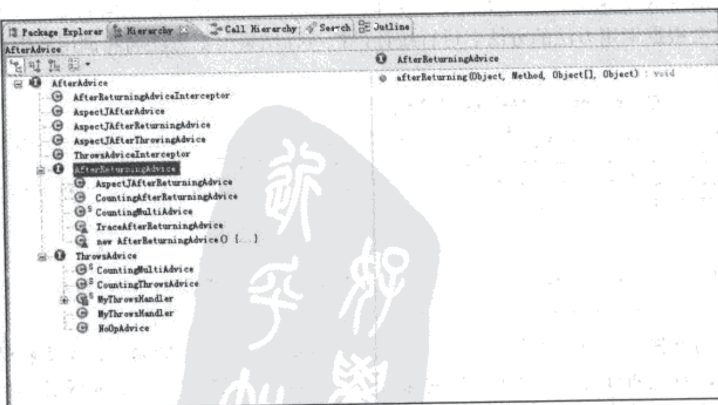


图 3-3 AfterAdvice 的接口关系

Download at http://www.pin5i.com/

在如图 3-3 所示的 AfterAdvice 类接口关系中，有一系列对 AfterAdvice 的实现和接口扩展，比如 AfterReturnningAdvice。我们以 AfterReturningAdvice 通知的实现为例，分析一下 AfterAdvice 通知类型的实现原理。在 AfterReturningAdvice 接口中定义了接口方法，如下所示。

```
void afterReturning(Object returnValue, Method method, Object[] args, Object
    target) throws Throwable;
```

AfterReturing 方法也是一个回调函数，AOP 应用需要在这个接口实现中，提供切面增强的具体设计，在这个 Advice 通知被正确配置以后，在目标方法调用结束并成功返回的时候，接口实现会被 Spring AOP 回调。对于回调参数，有目标方法的返回结果、反射对象以及调用参数（AOP 把这些参数都封装在一个对象数组中传递进来）等。与前面分析 BeforeAdvice 一样，在 Spring AOP 的包中，同样可以看到一个 CountingAfterReturningAdvice，作为熟悉 AfterReturningAdvice 使用的例子；它的实现基本上与 CountingBeforeAdvice 是一样的，如代码清单 3-3 所示。

代码清单 3-3 CountingAfterReturningAdvice 的实现

```
public class CountingAfterReturningAdvice extends MethodCounter implements
    AfterReturningAdvice {
public void afterReturning(Object o, Method m, Object[] args, Object target)
    throws Throwable {
    count(m);
}
}
```

在实现 AfterReturningAdvice 的接口方法 AfterReturning 中，可以调用 MethodCounter 的 count 方法，从而完成根据方法名来对目标方法调用的次数进行统计。count 方法调用的实现与 CountingBeforeAdvice 基本上是一样的，所不同的是调用发生的时间。尽管增强逻辑相同，但是，如果它实现不同的 AOP 通知接口，就会被 AOP 编织到不同的调用场合中。尽管它们完成的增强行为是一样的，都是根据目标方法名对调用次数进行统计，但是它们的最终实现却有很大不同，一个是在目标方法调用前实现切面增强，一个是在目标方法成功调用返回结果后实现切面增强。由此可见，AOP 技术给应用带来的灵活性，使得相同的代码完全可以根据应用的需要灵活地出现在不同的应用场合。

了解了 BeforeAdvice 和 AfterAdvice，在 Spring AOP 中还可以看到另外一种 Advice 通知类型——ThrowsAdvice，它的类层次关系如图 3-4 所示。

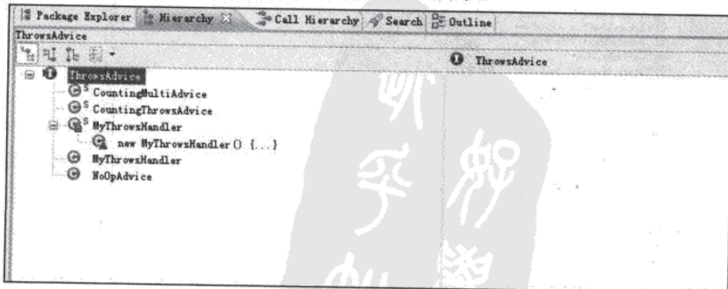


图 3-4 ThrowsAdvice 的类层次关系

Download at http://www.pin5i.com/

ThrowsAdvice 并没有指定需要实现的接口方法，它在抛出异常时被回调，这个回调是 AOP 使用反射机制来完成的。我们可以通过 CountingThrowsAdvice 了解 ThrowsAdvice 的使用方法，如代码清单 3-4 所示。

代码清单 3-4 CountingThrowsAdvice 的实现

```
public static class CountingThrowsAdvice extends MethodCounter implements ThrowsAdvice {
    public void afterThrowing(IOException ex) throws Throwable {
        count(IOException.class.getName());
    }

    public void afterThrowing(UncheckedException ex) throws Throwable {
        count(UncheckedException.class.getName());
    }
}
```

在 AfterThrowing 方法中，从输入的异常对象中得到异常的名字并进行统计。count 方法同样是在 MethodCounter 中实现，与前面的两个 Advice 的实现相同，只是 CountingBeforeAdvice 和 CountingAfterReturningAdvice 统计的是目标方法的调用次数。在这里，count 方法完成的是根据异常名称统计抛出异常的次数。

3.1.3 Pointcut 切点

Pointcut（切点）：决定 Advice 通知应该作用于哪个连接点，也就是说通过 Pointcut 切点来定义需要增强的方法的集合，这些集合的选取可以按照一定的规则来完成。在这种情况下，Pointcut 通常意味着标识方法，例如，这些需要增强的地方可以是被某个正则表达式进行标识，或根据某个方法名进行匹配等。

为了方便用户使用，Spring AOP 提供了具体的切点供用户使用，关于切点，可以看看 Spring AOP 中的类继承体系，如图 3-5 所示。

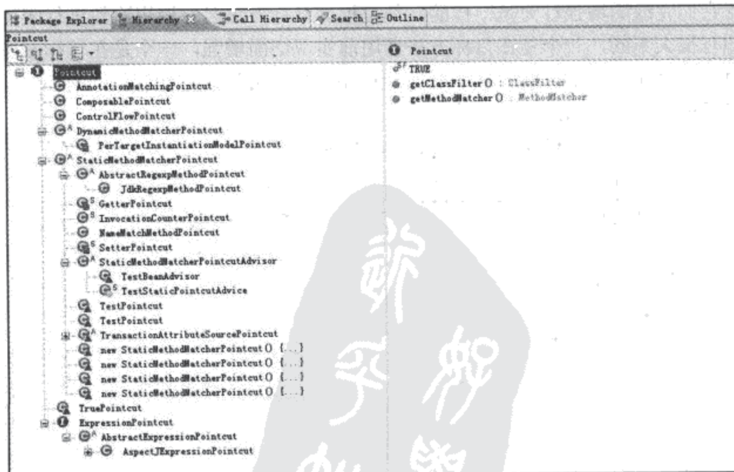


图 3-5 Spring AOP 的 Pointcut 类继承关系

Download at http://www.pin5i.com/

在 Pointcut 的基本接口定义中可以看到，需要返回一个 MethodMatcher；对于 Point 的匹配判断功能，具体是由这个返回的 MethodMatcher 来完成的，也就是说，由这个 MethodMatcher 来判断是否需要对当前方法调用进行增强，或者说对当前调用方法是否需要应用配置好的 advice 通知。在 Pointcut 的类继承关系中，我们以正则表达式切点 JdkRegexpMethodPointcut 的实现原理为例，来具体了解切点 Pointcut 的工作原理。JdkRegexpMethodPointcut 类完成通过正则表达式对方法名进行匹配的功能。在 JdkRegexpMethodPointcut 的基类 StaticMethodMatcherPointcut 的实现中可以看到，设置的 MethodMatcher 为 StaticMethodMatcher，同时 JdkRegexpMethodPointcut 也是这个 MethodMatcher 的子类，它的类层次关系如图 3-6 所示。

图 3-6　StaticMetnodMatcher 的类层次关系

如图 3-6 所示，在 Pointcut 中，通过这样的类继承关系，MethodMatcher 对象实际上是可以被配置成 JdkRegexpMethodPointcut 来完成方法的匹配判断的。在 JdkRegexpMethod-Pointcut 中，可以看到 matches 方法，这个方法是 MethodMatcher 定义的接口方法；在 JdkRegexpMethodPointcut 的实现中，这个 matches 方法就是使用正则表达式来对方法名进行匹配的地方。关于在 AOP 框架中对 matches 方法的调用，我们会在下面具体分析 Spring AOP 实现中看到。要了解 matches 在 AOP 框架中的调用位置，比较简单的方法就是从 matches 方法作为起始点，去对它的方法调用关系进行追溯，可以看到对 matches 方法的调用关系如图 3-7 所示。

图 3-7　对 JdkRegexpMethodPointcut 的 matches 方法的调用关系

Download at http://www.pin5i.com/

在对 matches 方法的调用关系中可以看到，是 JdkDynamicAopProxy 的 invoke 方法中触发了对 matches 方法的调用，熟悉 Proxy 使用的读者，一定会想到，这个 invoke 方法应该就是 Proxy 对象进行代理回调的入口方法，这个 invoke 回调的实现是使用 JDK 动态代理完成 AOP 功能的一部分，关于这部分实现原理，在下面 AOP 的实现分析中有详细的阐述。我们这里的重点是了解 Pointcut 的实现原理，比如去具体看看 matches 本身的实现，对于 JdkRegexpMethod-Pointcut 的 matches 方法，它的是实现如代码清单 3-5 所示。

代码清单 3-5 JdkRegexpMethodPointcut 使用 matches 完成匹配

```
protected boolean matches(String pattern, int patternIndex) {
    Matcher matcher = this.compiledPatterns[patternIndex].matcher(pattern);
    return matcher.matches();
}
```

在 JDKRegexpMethodPointcut 中，通过 JDK 来实现正则表达式的匹配，在代码清单 3-5 中可以看得很清楚；如果要详细了解使用 JDK 的正则表达式匹配功能，可以参考 JDK 的 API 文档。我们接着看看其他的 Pointcut。

在 Spring AOP 中，还提供了其他的 MethodPointcut，比如通过方法名匹配进行 Advice 匹配的 NameMatchMethodPointcut。它的 matches 方法实现很简单，匹配的条件是方法名相同或者方法名相匹配，如代码清单 3-6 所示。

代码清单 3-6 NameMatchMethodPointcut 的 matches

```
public boolean matches(Method method, Class targetClass) {
    for (String mappedName : this.mappedNames) {
        if (mappedName.equals(method.getName()) || isMatch(method.getName(),
        mappedName)) {
            return true;
        }
    }
    return false;
}
protected boolean isMatch(String methodName, String mappedName) {
    return PatternMatchUtils.simpleMatch(mappedName, methodName);
}
```

3.1.4 Advisor 通知器

Advisor（通知器）：当我们完成对目标方法的切面增强设计（advice）和关注点的设计（pointcut）以后，需要一个对象把它们结合起来，完成这个作用的就是 Advisor。通过 Advisor，可以定义应该使用哪个 Advice 并在哪个 Pointcut 使用它，也就是说通过 Advisor 把 Advice 和 Pointcut 结合起来了，这个结合为应用使用 IoC 容器配置 AOP 应用，或者说即开即用地使用 AOP 基础设施。在 Spring AOP 中，我们举一个 Advisor 的实现（DefaultPointcutAdvisor）作为例子，去了解 Advisor 的工作原理。在 DefaultPointcutAdvisor 中有两个属性，分别是 Advice 和 Pointcut，通过这两个属性，可以分别配置 Advice 和 Pointcut，DefaultPointcutAdvisor 的实现如代码清单 3-7 所示。

代码清单 3-7 DefaultPointcutAdvisor 的实现

```
public class DefaultPointcutAdvisor extends AbstractGenericPointcutAdvisor
    implements Serializable {
```

Download at http://www.pin5i.com/

```
    private Pointcut pointcut = Pointcut.TRUE;
    public DefaultPointcutAdvisor() {
    }
    public DefaultPointcutAdvisor(Advice advice) {
        this(Pointcut.TRUE, advice);
    }
    public DefaultPointcutAdvisor(Pointcut pointcut, Advice advice) {
        this.pointcut = pointcut;
        setAdvice(advice);
    }
    public void setPointcut(Pointcut pointcut) {
        this.pointcut = (pointcut != null ? pointcut : Pointcut.TRUE);
    }
    public Pointcut getPointcut() {
        return this.pointcut;
    }
    public String toString() {
        return getClass().getName() + ": pointcut [" + getPointcut() + "]; advice ["
        + getAdvice() +"]";
    }
}
```

在 DefaultPointcutAdvisor 中默认的 Pointcut 被设置为 Pointcut.True，这个 Pointcut.True 是在 Pointcut 接口中被定为的：

```
    Pointcut TRUE = TruePointcut.INSTANCE;
```

TruePointcut 的 INSTANCE 是一个单件。在它的实现中，可以看到单件模式的具体应用和它的典型使用方法，比如使用 static 类变量来持有单件实例，使用 private 私有构造函数来确保除了在当前单件实现中，单件不会再次被创建和实例化，从而保证它的"单件"特性。在 TruePointcut 的 methodMatcher 实现中，使用的是 TrueMethodMatcher 作为方法匹配器。在这个方法匹配器中，它对任何的方法匹配要求都返回 true 的结果，也就是说对任何方法名的匹配要求，它都会返回匹配成功的结果，和 TruePointcut 一样，这个 TrueMethodMatcher 也是一个单件实现。TruePointcut 和 TrueMethodMatcher 的实现如代码清单 3-8 和代码清单 3-9 所示。

代码清单 3-8　TruePointcut 的实现

```
class TruePointcut implements Pointcut, Serializable {
public static final TruePointcut INSTANCE = new TruePointcut();
// Enforce Singleton pattern.
private TruePointcut() {
}
public ClassFilter getClassFilter() {
    return ClassFilter.TRUE;
}
public MethodMatcher getMethodMatcher() {
    return MethodMatcher.TRUE;
}
```

代码清单 3-9　TrueMethodMatcher 的实现

```
class TrueMethodMatcher implements MethodMatcher, Serializable {
public static final TrueMethodMatcher INSTANCE = new TrueMethodMatcher();
// Enforce Singleton pattern.
private TrueMethodMatcher() {
}
```

Download at http://www.pin5i.com/

```
    public boolean isRuntime() {
        return false;
    }
    public boolean matches(Method method, Class targetClass) {
        return true;
    }
    public boolean matches(Method method, Class targetClass, Object[] args) {
        // Should never be invoked as isRuntime returns false.
        throw new UnsupportedOperationException();
    }
    private Object readResolve() {
        return INSTANCE;
    }

    public String toString() {
        return "MethodMatcher.TRUE";
    }
    }
```

3.2　建立 AopProxy 代理对象

3.2.1　配置 ProxyFactoryBean

从这部分开始，我们进入到 Spring AOP 的实现部分，在分析 Spring AOP 的实现原理中，我们主要以 ProxyFactoryBean 的实现作为例子和实现的基本线索进行分析。很大一个原因是因为 ProxyFactoryBean 是在 Spring IoC 环境中，创建 AOP 应用的最底层方法，也是最灵活的方法，Spring 通过它完成了对 AOP 使用的封装，以它的实现为入口逐层深入，是很好的一条帮助我们理解 Spring AOP 实现的学习路径。

在了解 ProxyFactoryBean 的实现之前，我们先简要地了解一下 ProxyFactoryBean 的配置和使用，在基于 XML 配置 Spring 的 Bean 的时候，往往需要一系列的配置步骤来使用 ProxyFactoryBean 和 AOP，比如以下这些步骤：

1）定义使用的通知器 Advisor，这个通知器应该作为一个 Bean 来定义。很重要的一点，这个通知器的实现定义了需要对目标对象进行增强的切面行为，也就是 Advice 通知。

2）定义 proxyFactoryBean，把它作为另一个 Bean 来定义，它是封装 AOP 功能的主要类。在配置 ProxyFactoryBean 时，需要设定与 AOP 实现相关的重要属性，比如 ProxyInterface、interceptorNames 和 target 等。从属性名称可以看出，interceptorNames 属性的值往往设置为需要定义的通知器，因为这些通知器在 ProxyFactoryBean 的 AOP 配置下，是通过使用代理对象的拦截器机制起作用的。所以，这里依然沿用了拦截器这个名字，也算是旧瓶装新酒吧。

3）定义 target 属性，作为 target 属性注入的 Bean，是需要用 AOP 通知器中的切面应用来增强的对象，也就是前面我们提到的 base 对象。

有了这些配置，就可以使用 ProxyFactoryBean 完成 AOP 的基本功能了。关于配置的例子，如代码清单 3-10 所示。与前面提到的配置步骤相对应，除了定义了 ProxyFactoryBean 的 AOP 封装，还定义了一个 Advisor，取名为 testAdvisor。作为 ProxyFactory 配置的一部

Download at http://www.pin5i.com/

分，还需要配置拦截的方法调用接口和目标对象。这些基本的配置，是使用 ProxyFactoryBean 实现 AOP 功能的重要组成，对于这些配置的实现和作用机制，也是我们后面重点分析的内容。

代码清单 3-10　配置 ProxyFactoryBean

```
<bean id="testAdvisor" class="com.abc.TestAdvisor"/>
<bean id="testAOP" class="org.springframework.aop.ProxyFactoryBean>
 <property name="proxyInterfaces"><value>com.test.AbcInterface"</value></property>
 <property name="target">
    <bean class="com.abc.TestTarget"/>
  </property>
  <property name="interceptorNames">
    <list><value> testAdvisor</value></list>
  </property>
</bean>
```

掌握这些配置后，就可以具体看一看这些 AOP 是如何实现的。也就是说，切面应用是怎样通过 ProxyFactoryBean 对 target 对象起作用的。对于这个 ProxyFactoryBean，我们先了解一下 ProxyFactoryBean 的类层次关系，如图 3-8 所示。在这个类层次关系中，可以看到用来完成 AOP 应用的类，比如 AspectJProxyFactory、ProxyFactory 和 ProxyFactoryBean，它们都在同一个类的继承体系下，都是 ProxyConfig、AdvisedSupport 和 ProxyCreatorSupport 的子类。作为大家的共同基类，ProxyConfig 可以看成是一个数据类，这个数据基类为像 ProxyFactoryBean 这样的子类提供了配置属性。在另一个基类 AdvisedSupport 的实现中，它封装了 AOP 中对通知和通知器的相关操作，这些操作对于不同的 AOP 的代理对象的生成都是一样的，但对于具体的 AOP 代理对象的创建，AdvisedSupport 把它交给它的子类们去完成。ProxyCreatorSupport 看成是其子类创建 AOP 代理对象的一个辅助类，通过继承以上提到的基类的功能实现，具体的 AOP 代理对象的生成，根据不同的需要分别由 ProxyFactoryBean、AspectJProxyFactory 和 ProxyFactory 来完成。对于需要使用 AspectJ 的 AOP 应用，AspectJProxyFactory 起到集成 Spring 和 AspectJ 的作用；对于使用 Spring AOP 的应用，ProxyFactoryBean 和 ProxyFactoy 都提供了 AOP 功能的封装，只是如果使用 ProxyFactoryBean，可以在 IoC 容器中完成声明式配置，而使用 ProxyFactory，则需要编程式的使用 Spring AOP 的功能；对于它们是如何封装实现 AOP 功能的，在本章后面的小节中给出详细的分析，在这里，通过这些类层次关系的了解，希望能先给读者留下一个大致的印象。

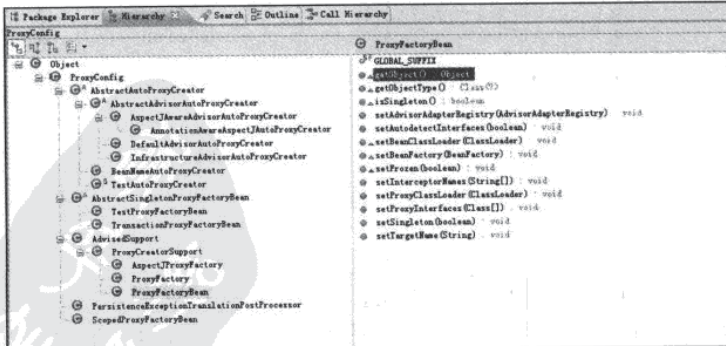


图 3-8　ProxyFactoryBean 相关的类层次关系

Download at http://www.pin5i.com/

3.2.2 ProxyFactoryBean 生成 AopProxy

在 Spring AOP 的使用中，我们已经了解到，可以通过 ProxyFactoryBean 来配置目标对象和方面行为。这个 ProxyFactoryBean 是一个 FactoryBean，对 FactoryBean 这种 Spring 应用中经常出现的 Bean 的工作形式，读者一定不会感到陌生，对于 FactoryBean 的工作原理，我们在结合 IoC 容器的实现原理分析中已经做过阐述。在 ProxyFactoryBean 中，通过 interceptorNames 属性来配置已经定义好的通知器 Advisor。虽然这里的名字叫 interceptNames，但值得注意的是，实际上却是供 AOP 应用配置通知器的地方。在 ProxyFactoryBean 中，需要为 target 目标对象生成 Proxy 代理对象，从而为 AOP 横切面的编织做好准备工作。这些具体的代理对象生成工作，在以后的实现原理分析中，我们可以看到是通过 JDK 的 Proxy 或 CGLIB 来完成的。

在 ProxyFactoryBean 中，它的 AOP 实现需要依赖 JDK 或者 CGLIB 提供的 Proxy 特性。从 FactoryBean 中获取对象，是用 getObject() 方法作为入口完成的；让我们进入 ProxyFactoryBean 的实现中去，这个我们一点也不陌生的 getObject 方法，是 FactoryBean 需要实现的接口。对 ProxyFactoryBean 来说，把需要对 target 目标对象增加的增强处理，都通过 getObject 方法进行封装了，这些增强处理是为 AOP 功能的实现提供服务的。对于 getObject 的实现如代码清单 3-11 所示。getObject 方法首先对通知器链进行初始化，通知器链封装了一系列的拦截器，这些拦截器都要从配置中读取，然后为代理对象的生成做好准备。在生成代理对象的时候，因为 Spring 中有 sigleton 类型和 prototype 类型这两种不同的 Bean，所以这里对代理对象的生成需要做一个区分。

代码清单 3-11 ProxyFactoryBean 的 getObject

```
public Object getObject() throws BeansException {
    //这里初始化通知器链
    initializeAdvisorChain();
    //这里对 singleton 和 protype 的类型进行区分,生成对应的 proxy.
    if (isSingleton()) {
        return getSingletonInstance();
    }
    else {
        if (this.targetName == null) {
            logger.warn("Using non-singleton proxies with singleton targets is 
    often undesirable. " +
                    "Enable prototype proxies by setting the 'targetName' property.");
        }
        return newPrototypeInstance();
    }
}
```

为 Proxy 代理对象配置 Advisor 链是在 initializeAdvisorChain 方法中完成的，如代码清单 3-12 所示。这个初始化过程有一个标志位 advisorChainInitialized，这个标志用来表示通知器链是否已经初始化。如果已经初始化，那么这里就不会再初始化，而是直接返回。也就是说，这个初始化的工作，发生在应用第一次通过 ProxyFactoryBean 去获取代理对象的时候。在完成这个初始化之后，接着会读取配置中出现的所有通知器，这个取得通知器的过程也比较简单，把通知器的名字交给容器的 getBean 方法就可以了，这是通过对 IoC 容器实现的一个回调来

Download at http://www.pin5i.com/

完成的。然后把从 IoC 容器中取得的通知器加入到拦截器链中，这个动作是由 addAdvisorOnChainCreation 方法来实现的。

代码清单 3-12 对 Advisor 配置链的初始化

```
private synchronized void initializeAdvisorChain() throws AopConfigException,
    BeansException {
    if (this.advisorChainInitialized) {
        return;
    }
    if (!ObjectUtils.isEmpty(this.interceptorNames)) {
        if (this.beanFactory == null) {
            throw new IllegalStateException("No BeanFactory available anymore
            (probably due to serialization) " +
            "- cannot resolve interceptor names " + Arrays.asList(this.
            interceptorNames));
        }
        /**
        * Globals can't be last unless we specified a targetSource
        * using the property...
        */
        if (this.interceptorNames[this.interceptorNames.length - 1].endsWith
        (GLOBAL_SUFFIX) && this.targetName == null && this.targetSource ==
        EMPTY_TARGET_
    SOURCE) {
            throw new AopConfigException("Target required after globals");
        }

        // Materialize interceptor chain from bean names.
        // 这里是添加advisor链的调用，是通过interceptorNames属性来进行配置的。
        for (String name : this.interceptorNames) {
            if (logger.isTraceEnabled()) {
                logger.trace("Configuring advisor or advice '" + name + "'");
            }

            if (name.endsWith(GLOBAL_SUFFIX)) {
                if (!(this.beanFactory instanceof ListableBeanFactory)) {
                    throw new AopConfigException(
                            "Can only use global advisors or interceptors with
                            a ListableBeanFactory");
                }
                addGlobalAdvisor((ListableBeanFactory) this.beanFactory,
                        name.substring(0, name.length() - GLOBAL_SUFFIX.length()));
            }

            else {
                // If we get here, we need to add a named interceptor.
                // We must check if it's a singleton or prototype.
                Object advice;
                if (this.singleton || this.beanFactory.isSingleton(name)) {
                    // Add the real Advisor/Advice to the chain.
                    advice = this.beanFactory.getBean(name);
                }
                else {
                   // It's a prototype Advice or Advisor: replace with a prototype.
                    /**
                    *Avoid unnecessary creation of prototype bean just for
                    *advisor chain initialization.
                    */
```

Download at http://www.pin5i.com/

```
                        advice = new PrototypePlaceholderAdvisor(name);
                    }
                    addAdvisorOnChainCreation(advice, name);
                }
            }
        }
        this.advisorChainInitialized = true;
    }
```

生成 singleton 的代理对象在 getSingletonInstance() 的代码中完成，这个方法是 ProxyFactoryBean 生成 AopProxy 代理对象的调用入口。在代理对象中会封装对 target 目标对象的调用，也就是说针对 target 对象的方法调用行为，会被这里生成的代理对象所拦截。具体的生成过程是，首先需要读取 ProxyFactoryBean 中的配置，为生成代理对象做好必要的准备，比如设置代理的方法调用接口等。对于在 getSingletonInstance() 方法中代理对象的生成过程，如代码清单 3-13 所示。

代码清单 3-13 生成单件代理对象

```
private synchronized Object getSingletonInstance() {
    if (this.singletonInstance == null) {
        this.targetSource = freshTargetSource();
        if (this.autodetectInterfaces && getProxiedInterfaces().length == 0
        && !isProxyTargetClass()) {
            // Rely on AOP infrastructure to tell us what interfaces to proxy.
            Class targetClass = getTargetClass();
            if (targetClass == null) {
                throw new FactoryBeanNotInitializedException("Cannot determine
                target class for proxy");
            }
// 这里设置代理对象的接口。
setInterfaces(ClassUtils.getAllInterfacesForClass(targetClass, this.proxyClassLoader));
        }
        // Initialize the shared singleton instance.
        super.setFrozen(this.freezeProxy);
        // 注意这里的方法会使用 ProxyFactory 来生成我们需要的 Proxy.
        this.singletonInstance = getProxy(createAopProxy());
    }
    return this.singletonInstance;
}
//使用 createAopProxy 返回的 AopProxy 来得到代理对象。
protected Object getProxy(AopProxy aopProxy) {
    return aopProxy.getProxy(this.proxyClassLoader);
}
```

这里出现了 AopProxy 类型的对象，Spring 使用这个 AopProxy 接口类把 AOP 代理对象的实现与框架的其他部分有效地分离开来。AopProxy 是一个接口，它有两个子类实现，一个是 Cglib2AopProxy，另一个是 JdkDynamicProxy。顾名思义，对这两个 AopProxy 接口的子类实现，Spring 分别使用 CGLIB 和 JDK 来生成需要的 Proxy 代理对象。

具体的代理对象的生成，是在 ProxyFactoryBean 的基类 AdvisedSupport 的实现中通过 AopProxyFactory 完成的，这个代理对象要么从 JDK 中生成，要么借助 CGLIB 获得。因为 ProxyFactoryBean 本身就是 AdvisedSupport 的子类，在 ProxyFactoryBean 中获得 AopProxy 是很方便的，具体的 AopProxy 生成过程，我们可以在 ProxyCreatorSupport 中看到。至于需要生成什么样的代理对象，信息都封装在 AdvisedSupport 里，这个对象也是生成

Download at http://www.pin5i.com/

AopProxy 的方法的输入参数，这里设置为 this 本身，因为 ProxyCreatorSupport 本身就是 AdvisedSupport 的子类。在 ProxyCreatorSupport 中生成代理对象的入口实现，如代码清单 3-14 所示。

代码清单 3-14 ProxyCreatorSupport 生成 AopProxy 对象

```
protected final synchronized AopProxy createAopProxy() {
    if (!this.active) {
        activate();
    }
    /**
    *通过 AopProxyFactory 取得 AopProxy，这个 AopProxyFactory 是在
    *初始化函数中定义的，使用的是 DefaultAopProxyFactory。
    */
    return getAopProxyFactory().createAopProxy(this);
}
```

这里使用了 AopProxyFactory 来创建 AopProxy，AopProxyFactory 使用的是 DefaultAopProxyFactory。这个被使用的 AopProxyFactory，作为 AopProxy 的创建工厂对象，是在 ProxyFactoryBean 的基类 ProxyCreatorSupport 中被创建的。在创建 AopProxyFactory 的时候，它被设置为 DefaultAopProxyFactory。很显然，Spring 给出了这个默认的 AopProxyFactory 工厂的实现。有了这个 AopProxyFactory 对象以后，问题就转换为：在 DefaultAopProxyFactory 中 AopProxy 是怎样生成的问题了。

关于 AopProxy 代理对象的生成，需要考虑使用哪种生成方式，如果目标对象是接口类，那么适合使用 JDK 来生成代理对象，否则 Spring 会使用 CGLIB 来生成目标对象的代理对象。为了满足不同的代理对象生成的要求，DefaultAopProxyFactory 作为 AopProxy 对象的生产工厂，可以根据不同的需要生成这两种 AopProxy 对象。对于 AopProxy 对象的生产过程，在 DefaultAopProxyFactory 中创建 AopProxy 的代码中可以清楚地看到，但这是一个比较高层次的 AopProxy 代理对象的生成过程，如代码清单 3-15 所示。所谓高层次，是指在 DefaultAopProxyFactory 创建 AopProxy 的过程中，对不同的 AopProxy 代理对象的生成，所涉及的生成策略和场景做了相应的设计，但是对于具体的 AopProxy 代理对象的生成，最终并没有由这个 DefaultAopProxyFactory 来完成，比如对 JDK 和 CGLIB 这些具体的技术的使用，对具体的实现层次的代理对象的生成，是由 Spring 封装的 JdkDynamicAopProxy 和 CglibProxyFactory 类来完成的。

在 AopProxy 代理对象的生成过程中，首先要从 AdvisedSupport 对象中取得配置的目标对象，这个目标对象是实现 AOP 功能所必需的，道理很简单，AOP 完成的是切面应用对目标对象的增强，皮之不存，毛将焉附，这个目标对象可以看做是“皮”，而 AOP 切面增强就是依附于这块皮上的“毛发”。如果这里发现没有配置目标对象的话，会直接抛出异常，提醒 AOP 应用，需要提供正确的目标对象的配置。在对目标对象配置的检查完成以后，需要根据配置的情况来决定使用什么方式来创建 AopProxy 代理对象。一般而言，默认的方式是使用 JDK 来产生 AopProxy 代理对象，但是如果遇到配置的目标对象不是接口类的实现的时候，会使用 CGLIB 来产生 AopProxy 代理对象；在使用 CGLIB 来产生 AopProxy 代理对象的时候，因为 CGLIB 是一个第三方的类库，本身不在 JDK 的基本类库中，所以需要在 CLASSPATH 路径中正确地配置，以便能够加载和使用。在 Spring 中，使用 JDK 和 CGLIB 来生成 AopProxy 代理对象的工作，是由

Download at http://www.pin5i.com/

JdkDynamicAopProxy 和 CglibProxyFactory 来完成的。对于详细的代理对象的生成过程，在下面的小节中，将逐个进行详细的分析。

代码清单 3-15　在 DefaultAopProxyFactory 中创建 AopProxy

```
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
    if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSupplied
    ProxyInterfaces(config)) {
        Class targetClass = config.getTargetClass();
        if (targetClass == null) {
            throw new AopConfigException("TargetSource cannot determine target
            class: " +
            "Either an interface or a target is required for proxy creation.");
        }//如果 targetClass 是接口类，使用 JDK 来生成 Proxy。
        if (targetClass.isInterface()) {
            return new JdkDynamicAopProxy(config);
        }
        if (!cglibAvailable) {
            throw new AopConfigException(
                    "Cannot proxy target class because CGLIB2 is not available. " +
                    "Add CGLIB to the class path or specify proxy interfaces.");
        } //如果不是接口类要生成 proxy，那么使用 cglib 来生成。
        return CglibProxyFactory.createCglibProxy(config);
    }
    else {
        return new JdkDynamicAopProxy(config);
    }
}
```

3.2.3　JDK 生成 AopProxy 代理对象

我们都知道对于 AopProxy 代理对象，可以由 JDK 或 CGLIB 来生成，而 JdkDynamicAopProxy 和 Cglib2AopProxy 实现的都是 AopProxy 接口，它们的层次关系如图 3-9 所示。

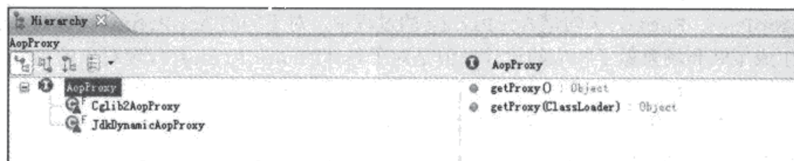


图 3-9　AopProxy 接口

我们先看看在 AopProxy 接口实现中，JdkDynamicAopProxy 是怎样完成 AopProxy 代理对象生成工作的。这个代理对象的生成过程如代码清单 3-16 所示。在 JdkDynamicAopProxy 中，使用了 JDK 的 Proxy 类来生成代理对象，在生成 Proxy 对象之前，首先需要从 advised 对象中取得代理对象的代理接口配置，然后调用 Proxy 的 newProxyInstance 方法，最终得到对应的 Proxy 代理对象；在生成代理对象的时候，需要指明三个参数，分别是类装载器、代理接口和 Proxy 回调方法所在的对象，这个对象需要实现 InvocationHandler 接口；InvocationHandler 接口定义了 invoke 方法，提供代理对象的回调入口；对于 JdkDynamicAopProxy，它本身实现了 InvocationHandler 接口和 invoke 方法，这个 invoke 方法是 Proxy 代理对象的回调方法，所以可以使用 this 来把 JdkDynamicAopProxy

Download at http://www.pin5i.com/

指派给 Proxy 对象；也就是说 JdkDynamicAopProxy 对象本身，在 Proxy 代理的接口方法被调用时，会促发 invoke 方法的回调。在这个回调方法里，完成了 AOP 编织实现的封装。在这里，我们先重点关注 AopProxy 代理对象的生成，Proxy 代理对象的 invoke 实现，将是我们后面详细分析 AOP 实现原理的重要部分。

代码清单 3-16 JdkDynamicAopProxy 生成 Proxy 代理对象

```
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.
        getTargetSource());
    }
    Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
    findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
    //这里调用 JDK 生成 Proxy 的地方。
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}
```

3.2.4 CGLIB 生成 AopProxy 代理对象

在 AopProxy 接口实现中，可以看到使用 CGLIB 来生成 Proxy 代理对象，这个 Proxy 代理对象的生成，可以在 Cglib2AopProxy 的代码实现中看到，同样是在 AopProxy 的接口方法 getProxy 的实现完成的，如代码清单 3-17 所示。可以看到具体对 CGLIB 的使用，比如对 Enhancer 对象的配置，以及通过 Enhancer 对象生成代理对象的过程。在这个生成代理对象的过程中，需要注意的是对 Enhancer 对象 callback 回调的设置，正是通过这些 callback 回调，封装了 Spring AOP 的实现，就像我们在前面看到的 JDK 的 Proxy 对象的 invoke 回调方法一样；在 Enhancer 的 callback 回调设置中，实际上是通过设置 DynamicAdvisedInterceptor 拦截器来完成 AOP 功能的，如果读者感兴趣的话，可以在 getCallbacks 方法实现中看到回调 DynamicAdvisedInterceptor 的设置：

```
Callback aopInterceptor = new DynamicAdvisedInterceptor(this.advised);
```

对于这个 DynamicAdvisedInterceptor 中的回调实现，我们将在后面详细地进行分析。

代码清单 3-17 Cglib2AopProxy 生成 AopFroxy 代理对象

```
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating CGLIB2 proxy: target source is " + this.advised.
        getTargetSource());
    }
    //从 advised 中取得在 IoC 容器中配置的 target 对象。
    try {
        Class rootClass = this.advised.getTargetClass();
        Assert.state(rootClass != null, "Target class must be available for
        creating a CGLIB proxy");
        Class proxySuperClass = rootClass;
        if (AopUtils.isCglibProxyClass(rootClass)) {
            proxySuperClass = rootClass.getSuperclass();
            Class[] additionalInterfaces = rootClass.getInterfaces();
            for (Class additionalInterface : additionalInterfaces) {
                this.advised.addInterface(additionalInterface);
            }
        }
```

Download at http://www.pin5i.com/

```
        // Validate the class, writing log messages as necessary.
        validateClassIfNecessary(proxySuperClass);
        // Configure CGLIB Enhancer...
        // 创建并配置 CGLIB 的 Enhancer，这个 Enhancer 对象是 CGLIB 的主要操作类。
        Enhancer enhancer = createEnhancer();
        if (classLoader != null) {
            enhancer.setClassLoader(classLoader);
            if (classLoader instanceof SmartClassLoader &&
                    ((SmartClassLoader) classLoader).isClassReloadable
                    (proxySuperClass)) {
                enhancer.setUseCache(false);
            }
        }
        /**
        *设置 Enhancer 对象，包括设置代理接口，回调方法，回调方法来自
        *advised 的 IoC 配置，比如使用 AOP 的 DynamicAdvisedInterceptor 拦截器。
        */
        enhancer.setSuperclass(proxySuperClass);
        enhancer.setStrategy(new UndeclaredThrowableStrategy(UndeclaredThrowable
        Exception.class));
        enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(this.advised));
        enhancer.setInterceptDuringConstruction(false);
        Callback[] callbacks = getCallbacks(rootClass);
        enhancer.setCallbacks(callbacks);
        enhancer.setCallbackFilter(new ProxyCallbackFilter(
                this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap,
                this.fixedInterceptorOffset));
        Class[] types = new Class[callbacks.length];
        for (int x = 0; x < types.length; x++) {
            types[x] = callbacks[x].getClass();
        }
        enhancer.setCallbackTypes(types);
        // 通过 Enhancer 生成代理对象。
        Object proxy;
        if (this.constructorArgs != null) {
            Proxy = enhancer.create(this.constructorArgTypes, this.constructorArgs);
        }
        else {
            proxy = enhancer.create();
        }
        return proxy;
    }
    catch (CodeGenerationException ex) {
        throw new AopConfigException("Could not generate CGLIB subclass of class [" +
                this.advised.getTargetClass() + "]: " +
                "Common causes of this problem include using a final class or a
                non-visible class",ex);
    }
    catch (IllegalArgumentException ex) {
        throw new AopConfigException("Could not generate CGLIB subclass of class [" +
                this.advised.getTargetClass() + "]: " +
                "Common causes of this problem include using a final class or a
                non-visible class",ex);
    }
    catch (Exception ex) {
        // TargetSource.getTarget() failed.
        throw new AopConfigException("Unexpected AOP exception", ex);
    }
}
```

Download at http://www.pin5i.com/

这样，通过使用 AopProxy 对象封装 target 目标对象之后，通过 ProxyFactoryBean 的 getObject 方法得到的对象就不是一个普通的 Java 对象了，而是一个 AopProxy 代理对象。在 ProxyFactoryBean 中配置的 target 目标对象，这个时候已经不会直接暴露给应用，而是作为 AOP 实现的一部分。对 target 目标对象的方法调用会首先被 AopProxy 代理对象拦截，对于不同的 AopProxy 代理对象生成方式，会使用不同的拦截回调入口。例如，对于 JDK 的 AopProxy 代理对象，使用的是 InvocationHandler 的 invoke 回调入口；而对于 CGLIB 的 AopProxy 代理对象，使用的是设置好的 callback 回调，这是由对 CGLIB 的使用来决定的；在这些 callback 回调中，对于 AOP 实现，是通过 DynamicAdvisedInterceptor 来完成的，而 DynamicAdvisedInterceptor 的回调入口是 intercept 方法。通过这一系列的准备，已经为实现 AOP 的横切机制奠定了基础，在这个基础上，AOP 的 Advisor 已经可以通过 AopProxy 代理对象的拦截机制，对需要它进行增强的 target 目标对象发挥切面的强大威力了。

可以把 AOP 的实现部分看成由基础设施准备和 AOP 运行辅助这两个部分组成。这里的 AopProxy 代理对象的生成，可以看成是一个静态的 AOP 基础设施的建立过程。通过这个准备过程把代理对象、拦截器这些待调用的部分都准备好，等待着 AOP 运行过程中对这些基础设施的使用。对于应用触发的 AOP 应用，会涉及 AOP 框架的运行和对 AOP 基础设施的使用。这些动态的运行部分，是从我们前面提到的拦截器回调入口开始的，这些拦截器调用的实现原理和 AopProxy 代理对象生成一样，也是 AOP 实现的重要组成部分，同时也是我们下面要重点分析的内容，让我们继续深入到 AopProxy 代理对象的回调实现中去看一看，慢慢地揭开 Spring AOP 实现的另一层神秘的面纱。

3.3 Spring AOP 拦截器调用的实现

3.3.1 JdkDynamicAopProxy 的 invoke 拦截

上面我们看到了在 Spring 中通过 ProxyFactoryBean 实现 AOP 功能的第一步，得到 AopProxy 代理对象的基本过程，以及通过使用 JDK 和 CGLIB 最终产生 AopProxy 代理对象的实现原理。下面我们看看 AopProxy 代理对象的拦截机制是怎样发挥作用和实现 AOP 功能的。在 JdkDynamicAopProxy 中生成 Proxy 对象时，我们回顾一下它的 AopProxy 代理对象的生成调用，如下所示。

```
Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
```

这里的 this 参数对应的是 InvocationHandler 对象，InvocationHandler 是 JDK 定义的反射类的一个接口，这个接口定义了 invoke 方法，而这个 invoke 方法是作为 JDKProxy 代理对象进行拦截的回调入口出现的。我们看到，在 JdkDynamicAopProxy 实现了 InvocationHandler 接口，也就是说当 Proxy 对象的代理方法被调用时，JdkDynamicAopProxy 的 invoke 方法作为 Proxy 对象的回调函数而被触发，从而通过 invoke 的具体实现，来完成对目标对象方法调用的拦截或者说功能增强的工作。下面我们看看 JdkDynamicAopProxy 的 invoke 方法实现，如代码清单 3-18 所示。可以看到，对 Proxy 对象完成的代理设置是在 invoke 方法中完成

Download at http://www.pin5i.com/

的，这些设置包括获取目标对象、拦截器链，同时把这些对象作为输入，创建了 ReflectiveMethodInvocation 对象，通过这个 ReflectiveMethodInvocation 对象来完成对 AOP 功能实现的封装。在这个 invoke 方法中，包含了一个完整的拦截器链对目标对象的拦截过程，比如获得拦截器链并对其中的拦截器进行配置，逐个运行拦截器链里的拦截增强，直到最后对目标对象方法的运行，等等。

代码清单 3-18 AopProxy 代理对象的回调

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    MethodInvocation invocation = null;
    Object oldProxy = null;
    boolean setProxyContext = false;
    TargetSource targetSource = this.advised.targetSource;
    Class targetClass = null;
    Object target = null;
    try {
        if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
            // The target does not implement the equals(Object) method itself.
            return equals(args[0]);
        }
        if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
            // The target does not implement the hashCode() method itself.
            return hashCode();
        }
        if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&
                method.getDeclaringClass().isAssignableFrom(Advised.class)) {
            // Service invocations on ProxyConfig with the proxy config...
            return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
        }
        Object retVal = null;
        if (this.advised.exposeProxy) {
            // Make invocation available if necessary.
            oldProxy = AopContext.setCurrentProxy(proxy);
            setProxyContext = true;
        }
        /**
        * May be null. Get as late as possible to minimize the time we "own" the
        * target, in case it comes from a pool.
        */
        //得到目标对象的地方。
        target = targetSource.getTarget();
        if (target != null) {
            targetClass = target.getClass();
        }
        // Get the interception chain for this method.
        // 这里获得定义好的拦截器链。
        List<Object> chain = this.advised.getInterceptorsAndDynamicInterception
    Advice(method, targetClass);
        /**
        * Check whether we have any advice. If we don't, we can fallback on
        * direct reflective invocation of the target, and avoid creating a MethodInvocation.
        */

        // 如果没有设定拦截器，那么我们就直接调用 target 的对应方法。
        if (chain.isEmpty()) {
```

Download at http://www.pin5i.com/

```
            /**
             * We can skip creating a MethodInvocation: just invoke the target directly
             * Note that the final invoker must be an InvokerInterceptor so we
             * know it does nothing but a reflective operation on the target, and no hot
             * swapping or fancy proxying.
             */
            retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
        }
        else {
            // We need to create a method invocation...
            /**
             * 如果有拦截器的设定，那么需要调用拦截器之后才调用目标对象的相应方法，
             * 通过构造一个 ReflectiveMethodInvocation 来实现，下面我们会看
             * 这个 ReflectiveMethodInvocation 类的具体实现。
             */
            invocation = new ReflectiveMethodInvocation(proxy, target, method,
            args, targetClass, chain);
            // Proceed to the joinpoint through the interceptor chain.
            //沿着拦截器链继续前进。
            retVal = invocation.proceed();
        }

        // Massage return value if necessary.
        if (retVal != null && retVal == target && method.getReturnType().
        isInstance(Proxy) && !RawTargetAccess.class.isAssignableFrom
        (method.getDeclaringClass())) {
            /**
             * Special case: it returned "this" and the return type of the method
             * is type-compatible. Note that we can't help if the target sets
             * a reference to itself in another returned object.
             */
            retVal = Proxy;
        }
        return retVal;
    }
    finally {
        if (target != null && !targetSource.isStatic()) {
            // Must have come from TargetSource.
            targetSource.releaseTarget(target);
        }
        if (setProxyContext) {
            // Restore old proxy.
            AopContext.setCurrentProxy(oldProxy);
        }
    }
}
```

3.3.2 Cglib2AopProxy 的 intercept 拦截

在分析 Cglib2AopProxy 的 AopProxy 代理对象生成的时候，我们知道对于 AOP 的拦截调用，其回调是在 DynamicAdvisedInterceptor 对象中实现的，这个回调的实现在 intercept 方法中，如代码清单 3-19 所示。Cglib2AopProxy 的 intercept 回调方法的实现和 JdkDynamicAopProxy 的回调实现是非常类似的，只是在 Cglib2AopProxy 中构造的是 CglibMethodInvocation 对象来完成拦截器链的调用，而在 JdkDynamicAopProxy 中是通过

Download at http://www.pin5i.com/

构造 ReflectiveMethodInvocation 对象来完成这个功能的。

代码清单 3-19 DynamicAdvisedInterceptor 的 intercept

```
public Object intercept(Object proxy, Method method, Object[] args, MethodProxy
    methodProxy) throws Throwable {
    Object oldProxy = null;
    boolean setProxyContext = false;
    Class targetClass = null;
    Object target = null;
    try {
        if (this.advised.exposeProxy) {
            // Make invocation available if necessary.
            oldProxy = AopContext.setCurrentProxy(proxy);
            setProxyContext = true;
        }
        /**
        * May be <code>null</code>. Get as late as possible to minimize the time we
        * "own" the target, in case it comes from a pool.
        */
        target = getTarget();
        if (target != null) {
            targetClass = target.getClass();
        }
        //从advised中取得配置好的AOP通知。
        List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice
        (method, targetClass);
        Object retVal = null;
        /**
        * Check whether we only have one InvokerInterceptor: that is,
        * no real advice, but just reflective invocation of the target.
        */
        // 如果没有AOP通知配置，那么直接调用target对象的调用方法。
        if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {
            /**
            * We can skip creating a MethodInvocation: just invoke the target directly.
            * Note that the final invoker must be an InvokerInterceptor, so we know
            * it does nothing but a reflective operation on the target, and no hot
            * swapping or fancy proxying.
            */
            retVal = methodProxy.invoke(target, args);
        }
        else {
            //通过CglibMethodInvocation来启动advice通知。
            retVal = new CglibMethodInvocation(proxy, target, method, args,
            targetClass, chain, methodProxy).proceed();
        }
        retVal = massageReturnTypeIfNecessary(proxy, target, method, retVal);
        return retVal;
    }
    finally {
        if (target != null) {
            releaseTarget(target);
        }
        if (setProxyContext) {
            // Restore old proxy.
            AopContext.setCurrentProxy(oldProxy);
        }
    }
}
```

Download at http://www.pin5i.com/

3.3.3 目标对象方法的调用

如果没有设置拦截器，那么会对目标对象的方法直接进行调用，对于 JdkDynamicAopProxy 代理对象，这个对目标对象的方法调用是通过 AopUtils 使用反射机制完成的，在 AopUtils.invokeJoinpointUsingReflection 的方法中实现，如代码清单 3-20 所示。在这个调用中，首先得到调用方法的反射对象，然后使用 invoke 启动对方法反射对象的调用。

代码清单 3-20 使用反射完成目标对象的方法调用

```
public static Object invokeJoinpointUsingReflection(Object target, Method method,
    Object[] args)
    throws Throwable {
    // Use reflection to invoke the method.
    // 这里是使用反射调用 target 对象方法的地方。
    try {
        ReflectionUtils.makeAccessible(method);
        return method.invoke(target, args);
    }
    catch (InvocationTargetException ex) {
        // Invoked method threw a checked exception.
        // We must rethrow it. The client won't see the interceptor.
        throw ex.getTargetException();
    }
    catch (IllegalArgumentException ex) {
        throw new AopInvocationException("AOP configuration seems to be invalid:
        tried calling method [" + method + "] on target [" + target + "]", ex);
    }
    catch (IllegalAccessException ex) {
        throw new AopInvocationException("Could not access method [" + method +
        "]", ex);
    }
}
```

对于使用 Cglib2AopProxy 的代理对象，它对目标对象的调用是通过 CGLIB 的 MethodProxy 对象来直接完成的，这个对象的使用是由 CGLIB 的设计来决定的；对于具体的调用在 DynamicAdvisedInterceptor 的 intercept 方法中可以看到，使用的是 CGLIB 封装好的功能，相对 JdkDynamicAopProxy 的实现来说，形式上看起来较为简单，但它们的功能却都是一样的，都是完成对目标对象方法的调用，具体的代码实现如下。

```
retVal = methodProxy.invoke(target, args);
```

3.3.4 AOP 拦截器链的调用

在了解了对目标对象的直接调用以后，我们开始进入 AOP 实现的核心部分了，对于 AOP 是怎样完成对目标对象的增强的，这些实现是封装在 AOP 拦截器链中，由一个个具体的拦截器来完成的。

尽管我们在上面看到，使用 JDK 和 CGLIB 会生成不同的 AopProxy 代理对象，从而构造了不同的回调方法来启动对拦截器链的调用，比如在 JdkDynamicAopProxy 中的 invoke 方法，以及 Cglib2AopProxy 中使用 DynamicAdvisedInterceptor 的 intercept 方法。它们都使用了不同的 AopProxy 代理对象，但最终对 AOP 拦截的处理可谓殊途同归：它们对拦截器链的

Download at http://www.pin5i.com/

调用都是在 ReflectiveMethodInvocation 中通过 proceed 方法实现的。在这个 proceed 方法里，会逐个运行拦截器的拦截方法。在运行拦截器的拦截方法之前，需要对代理方法完成一个匹配判断，通过这个匹配判断来决定拦截器是否满足切面增强的要求。大家一定还记得，我们前面提到的，在 Pointcut 切点中需要进行 matches 的匹配过程，就是这个 matches 调用对方法进行匹配判断，来决定是否需要实行通知增强；以下看到的调用就是进行 matches 的地方，具体的处理过程在 ReflectiveMethodInvocation 的 proceed 方法中，如代码清单 3-21 所示。在 proceed 方法中先进行判断，如果现在已经运行到拦截器链的末尾，那么就会直接调用目标对象的实现方法；否则，沿着拦截器链继续进行，得到下一个拦截器，通过这个拦截器进行 matches 判断，判断是否适用于横切增强的场合，如果是，从拦截器中得到通知器，并启动通知器的 invoke 方法进行切面增强。在这个过程结束以后，会迭代调用 proceed 方法，直到拦截器链中的拦截器都完成以上的拦截过程为止。

代码清单 3-21 拦截器的运行

```
public Object proceed() throws Throwable {
    //  We start with an index of -1 and increment early.
    /**
    *如果拦截器链中的拦截器迭代调用完毕，这里开始调用 target 的函数，
    *这个函数是通过反射机制完成的，具体实现在：AopUtils.invokeJoinpointUsingReflection 方法里面。
    */
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethod
    Matchers.size() - 1) {
        return invokeJoinpoint();
    }
    //这里沿着定义好的 interceptorOrInterceptionAdvice 链进行处理。
    Object interceptorOrInterceptionAdvice =
    this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptor
    Index);
    if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamic
    MethodMatcher) {
        /**
        * Evaluate dynamic method matcher here: static part will already have
        * been evaluated and found to match.
        */
        /**
        *这里对拦截器进行动态匹配的判断，还记得我们前面分析的 pointcut 吗？
        *这里是触发进行匹配的地方，如果和定义的 pointcut 匹配，那么这个 advice 将会得到执行。
        */
        InterceptorAndDynamicMethodMatcher dm =
          (InterceptorAndDynamicMethodMatcher)  interceptorOrInterceptionAdvice;
        if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
            return dm.interceptor.invoke(this);
        }
        else {
            // Dynamic matching failed.
            // Skip this interceptor and invoke the next in the chain.
            // 如果不匹配，那么 proceed 会被递归调用，直到所有的拦截器都被运行过为止。
            return proceed();
        }
    }
    else {
        /**
        * It's an interceptor, so we just invoke it: The pointcut will have
```

Download at http://www.pin5i.com/

```
         * been evaluated statically before this object was constructed.
         */
        //如果是 interceptor，直接调用这个 interceptor 对应的方法。
        return((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
    }
}
```

以上就是整个拦截器及 target 目标对象方法被调用的过程。小荷才露尖尖角，我们已经在这里看到对 advice 通知的调用入口了，虽然这个大名鼎鼎的 advice 到现在还没有完全现身，但我们已经看到了它的运行轨迹；我们先提出一个疑问来提提大家的兴趣：这些 advisor 是怎样从配置文件中获得并配置到 proxy 的拦截器链中去的？我们平常使用的 advice 通知是怎样起作用的？这些都是了解 AOP 实现原理的重要问题，下面我们就这些问题已经展示的线索继续展开分析，去寻求这些问题的答案吧。

3.3.5 配置通知器

在整个 AopProxy 代理对象的拦截回调过程中，让我们先回到 ReflectiveMethodInvocation 类的 proceed 方法，在这个方法里，可以看到得到了配置的 interceptorOrInterceptionAdvice，如下所示。

```
Object interceptorOrInterceptionAdvice =
    this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
```

这个 interceptorOrInterceptionAdvice 是获得的拦截器，它通过拦截器机制对目标对象的行为增强起作用。这个拦截器来自于 interceptorsAndDynamicMethodMatchers，具体来说，它是 interceptorsAndDynamicMethodMatchers 持有的 List 中的一个元素。关于如何配置拦截器的问题，就被转化为这个 List 中的拦截器元素是从哪里来、在哪里配置的问题。我们接着对 invoke 调用进行回放，回到 JdkDynamicAopProxy 中的 invoke 方法中，可以看到这个 List 中的 interceptors 是在哪个调用中获取的。对于 Cglib2AopProxy，也有类似的过程，只不过这个过程是在 DynamicAdvisedInterceptor 的 intercept 回调中实现的，如下所示。

```
List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
```

在上面的代码中可以看到，获取 interceptors 的操作是由 advised 对象完成的，这个 advised 是一个 AdvisedSupport 对象，从类的继承关系上看，这个 AdvisedSupport 类同时也是 ProxyFactoryBean 的基类。从 AdvisedSupport 的代码里可以看到 getInterceptorsAndDynamicInterceptionAdvice 的实现，如代码清单 3-22 所示。在这个方法里取得了拦截器链，在取得拦截器链的时候，为提高取得拦截器链的效率，还为这个拦截器链设置了缓存。

代码清单 3-22 AdvisedSupport 取得拦截器

```
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method,
    Class targetClass) {
    /**
    *这里使用了 cache，会从 cache 去取已有的 inteceptor 链，但是第一次
    *还是需要自己动手生成的。这个 inteceptor 链的生成是由 advisorChainFactory 完成的，
    *在这里使用的是 DefaultAdvisorChainFactory。
    */
```

Download at http://www.pin5i.com/

```
        MethodCacheKey cacheKey = new MethodCacheKey(method);
        List<Object> cached = this.methodCache.get(cacheKey);
        if (cached == null) {
            cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
                    this, method, targetClass);
            this.methodCache.put(cacheKey, cached);
        }
        return cached;
    }
```

取得拦截器链的工作是由配置好的 advisorChainFactory 来完成的，从名字上可以猜到，它是一个生成通知器链的工厂；在这里，advisorChainFactory 被配置成一个 DefaultAdvisorChainFactory 对象，在 DefaultAdvisorChainFactory 中实现了 interceptor 链的获取过程，如代码清单 3-23 所示。在这个获取过程中，首先设置了一个 List，其长度是由配置的通知器的个数来决定的，这个配置就是我们在 XML 中对 ProxyFactoryBean 做的 interceptNames 属性的配置；然后，DefaultAdvisorChainFactory 会通过一个 AdvisorAdapterRegistry 来实现拦截器的注册，在后面我们会看到，这个 AdvisorAdapterRegistry 对 advice 通知的织入功能起了很大的作用，关于这个 AdvisorAdapterRegistry 对象的实现原理，我们会在下面分析通知是如何实现增强的部分进行详细阐述。有了这个 AdvisorAdapterRegistry 注册器，由它来对从 ProxyFactoryBean 配置中得到的通知进行适配，从而获得相应的拦截器，再把它加入到前面设置好的 List 中去，完成这个所谓的拦截器注册过程。在这些拦截器适配和注册过程完成以后，这个 List 中的拦截器会被 JDK 生成的 AopProxy 代理对象的 invoke 方法，或者 CGLIB 代理对象的 intercept 拦截方法取得，并启动拦截器的 invoke 调用，最终触发通知的切面增强。

代码清单 3-23 DefaultAdvisorChainFactory 生成拦截器链

```
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
        Advised config, Method method, Class targetClass) {

    /**
     * This is somewhat tricky... we have to process introductions first,
     * but we need to preserve order in the ultimate list.
     */
    //advisor 链已经在 config 中持有了，这里我们可以直接使用。
    List<Object> interceptorList = new ArrayList<Object>
    (config.getAdvisors().length);
    boolean hasIntroductions = hasMatchingIntroductions(config, targetClass);
   AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
    for (Advisor advisor : config.getAdvisors()) {
            if (advisor instanceof PointcutAdvisor) {
            // Add it conditionally.
            PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
            if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClass
    Filter().matches(targetClass)) {
/**
*拦截器链是通过 AdvisorAdapterRegistry 来加入的，这个 AdvisorAdapterRegistry
*对 advice 织入起了很大的作用，在后面的分析中会看到。
*/
MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
    MethodMatcher mm = pointcutAdvisor.getPointcut().gEtMethodMatcher();
```

Download at http://www.pin5i.com/

```
//使用MethodMatchers的matches方法进行匹配判断。
                    if (MethodMatchers.matches(mm, method, targetClass, hasIntroductions)) {
                        if (mm.isRuntime()) {
                        /**
                        * Creating a new object instance in the getInterceptors()method
                        * isn't a problem as we normally cache created chains.
                        */
                            for (MethodInterceptor interceptor : interceptors) {
                                interceptorList.add(new InterceptorAndDynamicMethod
                                Matcher(interceptor, mm));
                            }
                        }
                        else {
                            interceptorList.addAll(Arrays.asList(interceptors));
                        }
                    }
                }
            }
            else if (advisor instanceof IntroductionAdvisor) {
                IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
                if (config.isPreFiltered() || ia.getClassFilter().matches(targetClass)) {
                    Interceptor[] interceptors = registry.getInterceptors(advisor);
                    interceptorList.addAll(Arrays.asList(interceptors));
                }
            }
            else {
                Interceptor[] interceptors = registry.getInterceptors(advisor);
                interceptorList.addAll(Arrays.asList(interceptors));
            }
        }
        return interceptorList;
    }
    // Determine whether the Advisors contain matching introductions.
    private static boolean hasMatchingIntroductions(Advised config, ClasstargetClass) {
        for (int i = 0; i < config.getAdvisors().length; i++) {
            Advisor advisor = config.getAdvisors()[i];
            if (advisor instanceof IntroductionAdvisor) {
                IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
                if (ia.getClassFilter().matches(targetClass)) {
                    return true;
                }
            }
        }
        return false;
    }
```

事实上，这里的advisor通知器是从AdvisorSupport中取得的，从对它调用过程上看会非常地清楚，如图3-10所示。

在ProxyFactoryBean的getObject方法中对advisor进行初始化的时候，从XML配置中获取了advisor通知器。在ProxyFactoryBean中，我们看看对advisor进行初始化的代码实现，如代码清单3-24所示。在这个初始化的advisor的取得中，可以看到对IoC容器的一个getBean回调，由这个对IoC容器的getBean调用来得到配置好的advisor通知器。

Download at http://www.pin5i.com/

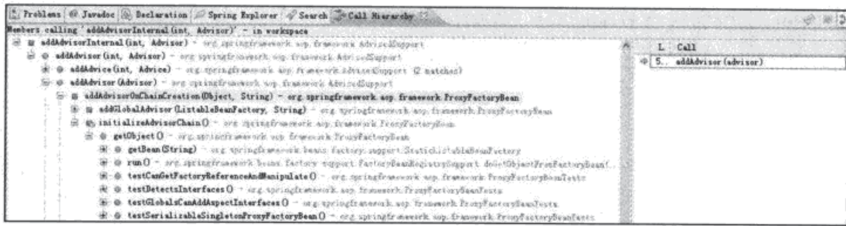


图 3-10 Advisor 的调用过程

代码清单 3-24 在拦截器链的初始化中获取 advisor 通知器

```
private synchronized void initializeAdvisorChain() throws AopConfigException,
    BeansException {
    if (this.advisorChainInitialized) {
        return;
    }
    if (!ObjectUtils.isEmpty(this.interceptorNames)) {
        if (this.beanFactory == null) {
            throw new IllegalStateException("No BeanFactory available anymore
            (probably due to serialization) " +
            "- cannot resolve interceptor names " + Arrays.asList(this.
            interceptorNames));
        }
        // Globals can't be last unless we specified a targetSource using the
    property...
        if (this.interceptorNames[this.interceptorNames.length - 1].endsWith
    (GLOBAL_SUFFIX) &&
                this.targetName == null && this.targetSource == EMPTY_TARGET_
    SOURCE) {
            throw new AopConfigException("Target required after globals");
        }
        // Materialize interceptor chain from bean names.
        for (String name : this.interceptorNames) {
            if (logger.isTraceEnabled()) {
                logger.trace("Configuring advisor or advice '" + name + "'");
            }
            if (name.endsWith(GLOBAL_SUFFIX)) {
                if (!(this.beanFactory instanceof ListableBeanFactory)) {
                    throw new AopConfigException(
                            "Can only use global advisors or interceptors with
                            a ListableBeanFactory");
                }
                addGlobalAdvisor((ListableBeanFactory) this.beanFactory,
                        name.substring(0, name.length() - GLOBAL_SUFFIX.length()));
            }
            else {
                // If we get here, we need to add a named interceptor.
                // We must check if it's a singleton or prototype.
                Object advice;
                if (this.singleton || this.beanFactory.isSingleton(name)) {
                // Add the real Advisor/Advice to the chain.
                /**
                * 这里是取得advisor的地方，是通过beanFactory取得的，
                *把interceptorNames这个List中的interceptor名字交给
                *beanFactory就可以了，然后通过调用BeanFactory的getBean去获取。
                */
```

Download at http://www.pin5i.com/

```
                advice = this.beanFactory.getBean(name);
            }
            else {
                // It's a prototype Advice or Advisor: replace with a prototype.
                /**
                 * Avoid unnecessary creation of prototype bean just for
                 *advisor chain initialization.
                 */
                advice = new PrototypePlaceholderAdvisor(name);
            }
            addAdvisorOnChainCreation(advice, name);
        }
    }
  }
  this.advisorChainInitialized = true;
}
```

advisor 通知器的取得是由 IoC 容器完成的，但是在 ProxyFactoryBean 中是如何获得 IoC 容器，然后通过回调 IoC 容器的 getBean 方法来得到需要的通知器 advisor 呢？这涉及 IoC 容器的实现原理，在我们使用 DefaultListableBeanFactory 作为 IoC 容器使用的时候，由于它的基类是 AbstractAutowireCapableBeanFactory，在这个 AbstractAutowireCapable-BeanFactory 中，可以看到一个对 Bean 进行初始化的 initializeBean 方法；在这个 Bean 的初始化过程中，对 IoC 容器在 Bean 中的回调进行了设置，首先，判断这个 Bean 的类型是不是实现了 BeanFactoryAware 接口，如果它实现了 BeanFactoryAware 接口，那么它就一定实现了 BeanFactoryAware 定义的接口方法，通过这个接口方法，可以把 IoC 容器设置到 Bean 自身定义的一个属性中去。这样，在这个 Bean 的自身实现中，就能够得到它所在的 IoC 容器，从而调用 IoC 容器的 getBean 方法，完成对 IoC 容器的回调，就像一个有特异功能的 Bean 一样，除了使用为自己设计的功能之外，还可以去调用它所在的容器的功能，如下所示。

```
if (bean instanceof BeanFactoryAware) {
        ((BeanFactoryAware) bean).setBeanFactory(this);
}
```

对 IoC 容器的使用，如果需要回调容器，前提是当前的 Bean 需要实现 BeanFactoryAware 接口，这个接口只需要实现一个接口方法 setBeanFactory，同时设置一个属性来持有 BeanFactory 的 IoC 容器，就可以在 Bean 中取得 IoC 容器进行回调了。在 IoC 容器对 Bean 进行初始化的时候，会对 Bean 的类型进行判断，如果这是一个 BeanFactoryAware 的 Bean 类型，那么 IoC 容器会调用这个 Bean 的 setBeanFactory 方法，完成对这个 BeanFactory 在 Bean 中的设置。具体来说，对于 ProxyFactoryBean，它实现了这个接口，所以在它初始化完成以后，可以在 Bean 中使用容器进行回调。这里设置进去的 this 对象就是 Bean 所在的 IoC 容器，一般而言是 DefaultListableBeanFactory 对象。通过这个设置，在得到这个设置好的 BeanFactory 以后，ProxyFactoryBean 就可以通过回调容器的 getBean 去获取配置在 Bean 定义文件中的通知器了，获取通知器就是向 IoC 容器 getBean 的过程。了解 IoC 容器的实现原理的读者都知道，这个 getBean 是 IoC 容器一个非常基本的方法。在调用时，ProxyFactoryBean 需要给出通知器的名字，而这些名字都是在 interceptorNames 的 List 中已经配置好的，在 IoC 对 FactoryBean 进行依赖注入时，会直接注入到 FactoryBean 的 interceptorNames 属性中。完成这个过程以后，ProxyFactoryBean 就获得了配置的通知器，为切面增强做好准备。

Download at http://www.pin5i.com/

3.3.6 Advice 通知的实现

经过前面的分析，我们看到在 AopProxy 代理对象的生成的时候，AopProxy 代理对象的拦截器也同样建立起来了，对于拦截器的拦截调用和最终目标对象的方法调用，也都看到了它们相应的实现原理。但是，对于 AOP 实现的重要部分，Spring AOP 定义的通知是怎样实现对目标对象的增强的呢？本节将探讨这个问题。在为 AopProxy 代理对象配置拦截器的实现中，有一个取得拦截器的配置过程，这个过程是由 DefaultAdvisorChainFactory 实现的，而在这个工厂类中，它负责生成拦截器链，在它的 getInterceptorsAndDynamicInterceptionAdvice 方法中，有一个适配和注册过程，就是在这个适配和注册过程中，通过配置 Spring 预先设计好的拦截器，Spring 加入了它对 AOP 实现的处理。为了解这个详细的过程，我们先从 DefaultAdvisorChainFactory 的实现开始，如代码清单 3-25 所示。在 DefaultAdvisorChainFactory 的实现中，首先构造了一个 GlobalAdvisorAdapterRegistry 单件，然后，对配置的 Advisor 通知器逐个进行遍历，这些通知器链都是配置在 interceptorNames 中的，这点我们已经不陌生；从 getInterceptorsAndDynamicInterceptionAdvice 传递进来的 advised 参数对象中可以方便地取到配置的通知器，有了这些通知器，接着就是一个由 GlobalAdvisorAdapterRegistry 来完成的拦截器的适配和注册过程。

代码清单 3-25 DefaultAdvisorChainFactory 使用 GlobalAdvisorAdapterRegistry 得到 AOP 拦截器

```
//得到注册器 GlobalAdvisorAdapterRegistry，这是一个单件模式的实现。
AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
for (Advisor advisor : config.getAdvisors()) {
    if (advisor instanceof PointcutAdvisor) {
        // Add it conditionally.
        PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
        if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClass
        Filter().matches(targetClass)) {
            //从 GlobalAdvisorAdapterRegistry 中取得 MethodInterceptor 的实现。
           MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
            MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();
            if (MethodMatchers.matches(mm, method, targetClass, hasIntroductions)) {
                if (mm.isRuntime()) {
                    /**
                    * Creating a new object instance in the getInterceptors() method
                    * isn't a problem as we normally cache created chains.
                    */
                    for (MethodInterceptor interceptor : interceptors) {
                        interceptorList.add(new InterceptorAndDynamicMethod
                        Matcher(interceptor, mm));


                    }
                }
                else {
interceptorList.addAll(Arrays.asList(interceptors));
                    }
                }
            }
        }
```

Download at http://www.pin5i.com/

仔细揣摩了以上代码的读者一定会注意到，在这个 GlobalAdvisorAdapterRegistry 中隐藏着不少 AOP 实现的重要细节，它的 getInterceptors 方法为 AOP 实现作出了很大的贡献，就是这个方法封装着 advice 织入实现的入口，我们先从 GlobalAdvisorAdapterRegistry 的实现入手，如代码清单 3-26 所示。从代码上看，GlobalAdvisorAdapterRegistry 的实现很简洁，起到的基本上是一个适配器的作用，但同时也是一个单件模式的应用，它为 Spring AOP 模块提供了一个 DefaultAdvisorAdapterRegistry 单件，这个 DefaultAdvisorAdapterRegistry 是我们后面要分析的重点，像它的名字一样，由它来完成各种通知的适配和注册工作。

代码清单 3-26 GlobalAdvisorAdapterRegistry 的实现

```
public abstract class GlobalAdvisorAdapterRegistry {
    /**
    * Keep track of a single instance so we can return it to classes that request it.
    */
    private static final AdvisorAdapterRegistry instance = new DefaultAdvisor
    AdapterRegistry();

    // Return the singleton DefaultAdvisorAdapterRegistry instance.
    public static AdvisorAdapterRegistry getInstance() {
        return instance;
    }
}
```

到这里，神秘的面纱慢慢地开始被揭开了，在 DefaultAdvisorAdapterRegistry 中设置了一系列的 adapter 适配器，正是这些 adapter 适配器的实现，为 Spring AOP 的 advice 提供编织能力，让我们先到 DefaultAdvisorAdapterRegistry 中去看一看究竟在这里发生了什么，如代码清单 3-27 所示。首先，我们看到了一系列在我们的 AOP 应用中，与使用到的 Spring AOP 的 advice 通知相对应的 adapter 适配实现，并看到了对这些 adapter 的具体使用。具体地说，对它们的使用主要体现在两个方面，一是调用 adapter 的 support 方法，通过这个方法来判断取得的 advice 属于什么类型的 advice 通知，从而根据不同的 advice 类型来注册不同的 AdviceInterceopter，也就是我们前面看到的那些拦截器；而另一方面，对于这些 AdviceInterceopter，不需要我们操心，都是 Spring AOP 框架设计好了的，它们是为实现不同的 advice 功能提供服务的。有了这些 AdviceInterceopter，我们可以方便地使用由 Spring 提供的各种不同的 advice 来设计 AOP 应用。也就是说，正是这些 AdviceInterceptor 最终实现了 advice 通知在 AopProxy 代理对象中的织入功能。

代码清单 3-27 DefaultAdvisorAdapterRegistry 的实现

```
public class DefaultAdvisorAdapterRegistry implements AdvisorAdapterRegistry,
    Serializable {
    //持有 AdvisorAdaptor 的 List，它的 Adapter 是与实现 Spring AOP 的 advice 增强功能相对应的。
    private final List<AdvisorAdapter> adapters = new ArrayList<AdvisorAdapter>(3);
    // Create a new DefaultAdvisorAdapterRegistry, registering well-known adapters.
    /**
    *这里把已有的 advice 实现的 Adapter 加入进来，有我们非常熟悉的
    *MethodBeforeAdvice、AfterReturningAdvice、ThrowsAdvice 这些 AOP 的 advice 封装实现。
    */
    public DefaultAdvisorAdapterRegistry() {
        registerAdvisorAdapter(new MethodBeforeAdviceAdapter());
        registerAdvisorAdapter(new AfterReturningAdviceAdapter());
        registerAdvisorAdapter(new ThrowsAdviceAdapter());
```

Download at http://www.pin5i.com/

```
    }
    public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {
        if (adviceObject instanceof Advisor) {
            return (Advisor) adviceObject;
        }
        if (!(adviceObject instanceof Advice)) {
            throw new UnknownAdviceTypeException(adviceObject);
        }
        Advice advice = (Advice) adviceObject;
        if (advice instanceof MethodInterceptor) {
            // So well-known it doesn't even need an adapter.
            return new DefaultPointcutAdvisor(advice);
        }
        for (AdvisorAdapter adapter : this.adapters) {
            // Check that it is supported.
            if (adapter.supportsAdvice(advice)) {
                return new DefaultPointcutAdvisor(advice);
            }
        }
        throw new UnknownAdviceTypeException(advice);
    }
    //这里是在DefaultAdvisorChainFactory中启动的getInteceptors方法。
    public MethodInterceptor[] getInterceptors(Advisor advisor) throwsUnknownAdvice
    TypeException {
        List<MethodInterceptor> interceptors = new ArrayList<MethodInterceptor>(3);
        //从Advisor通知器配置中取得advice通知。
        Advice advice = advisor.getAdvice();
        //如果通知是MethodInterceptor类型的通知，直接加入interceptors的List中，不需要适配。
        if (advice instanceof MethodInterceptor) {
            interceptors.add((MethodInterceptor) advice);
        }
        /**
        *对通知进行适配，使用已经配置好的Adapter: MethodBeforeAdviceAdapter,
        *AfterReturningAdviceAdapter以及ThrowsAdviceAdapter。
        *然后从对应的adapter中取出封装好AOP编织功能的拦截器。
        */
        for (AdvisorAdapter adapter : this.adapters) {
            if (adapter.supportsAdvice(advice)) {
                interceptors.add(adapter.getInterceptor(advisor));
            }
        }
        if (interceptors.isEmpty()) {
            throw new UnknownAdviceTypeException(advisor.getAdvice());
        }
        return interceptors.toArray(new MethodInterceptor[interceptors.size()]);
    }
    public void registerAdvisorAdapter(AdvisorAdapter adapter) {
        this.adapters.add(adapter);
    }
}
```

在 DefaultAdvisorAdapterRegistry 的 getInterceptors 调用中，我们可以看到 MethodBeforeAdviceAdapter、AfterReturningAdviceAdapter 以及 ThrowsAdviceAdapter 这几个通知适配器，从名字上可以看到，它们完全是和 advice 一一对应的，在这里，它们作为适配器被加入到 adaper 的 List 中来了。换一个角度，从这几个类的设计层次和关系上

Download at http://www.pin5i.com/

看，它们都是实现 AdvisorAdapter 接口的同一层次的类，只是各自承担着不同的适配任务，一对一地服务于不同的 advice 实现。它们的类层次关系如图 3-11 所示。

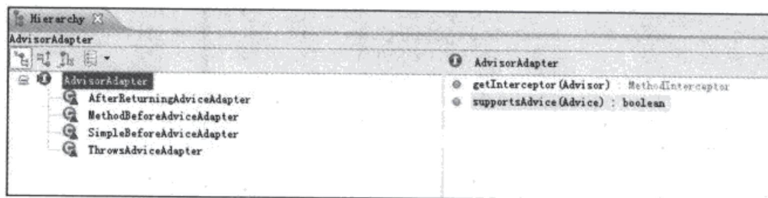


图 3-11　AdvisorAdapter 接口及其实现

我们举 MethodBeforeAdviceAdapter 为例，看看它的具体实现，如代码清单 3-28 所示。这个 MethodBeforeAdviceAdapter 的实现并不复杂，它实现了 AdvisorAdapter 的两个接口方法，一个是 supportsAdvice，它对 advice 的类型进行判断，如果 advice 是 MethodBeforeAdvice 的实例，那么返回值为 true；另一个是对 getInterceptor 接口方法的实现，它把 advice 通知从通知器中取出，然后创建一个 MethodBeforeAdviceInterceptor 对象，通过这个 MethodBeforeAdviceInterceptor 对象把取得的 advice 通知包装起来，然后返回。

代码清单 3-28　MethodBeforeAdviceAdapter 的实现

```
class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {
    public boolean supportsAdvice(Advice advice) {
        return (advice instanceof MethodBeforeAdvice);
    }
    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
        return new MethodBeforeAdviceInterceptor(advice);
    }
}
```

到这里就非常清楚了，Spring AOP 为了实现 advice 的织入，设计了特定的拦截器对这些功能进行了封装，虽然应用不会直接用到这些拦截器，但却是 advice 发挥作用的必不可少的基础设施。接着这条线索，我们还是使用 MethodBeforeAdviceInterceptor 作为例子，看看它是怎样完成对 advice 的封装的，如代码清单 3-29 所示。这个 MethodBeforeAdviceInterceptor 完成的是对 MethodBeforeAdvice 通知的封装，可以在 MethodBeforeAdviceInterceptor 设计的 invoke 回调方法中看到首先触发了 advice 的 before 回调，然后才是 MethodInvocation 的 proceed 方法调用。看到这里，就已经和前面我们在 ReflectiveMethodInvocation 中的 proceed 分析中联系起来了。在 AopProxy 代理对象触发的 ReflectiveMethodInvocation 的 proceed 方法中，在取得拦截器以后，启动了对拦截器 invoke 方法的调用。按照 AOP 的配置规则，ReflectiveMethodInvocation 触发的拦截器 invoke 方法，最终会根据不同的 advice 类型，触发 Spring 对不同的 advice 的拦截器封装，比如对 MethodBeforeAdvice，就会最终触发 MethodBeforeAdviceInterceptor 的 invoke 方法。在这个 MethodBeforeAdviceInterceptor 方法中，先调用 advice 的 before 方法，即 MethodBeforeAdvice 所需要的对目标对象的增强效果：在方法调用之前完成通知增强。

Download at http://www.pin5i.com/

代码清单 3-29 MethodBeforeAdviceInterceptor 的实现

```
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, Serializable {
    private MethodBeforeAdvice advice;
    /**
     * Create a new MethodBeforeAdviceInterceptor for the given advice.
     * @param advice the MethodBeforeAdvice to wrap
     */
    public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
        Assert.notNull(advice, "Advice must not be null");
        this.advice = advice;
    }
    //这个 invoke 方法是拦截器的回调方法，会在代理对象的方法被调用的时候触发回调。
    public Object invoke(MethodInvocation mi) throws Throwable {
        this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() );
        return mi.proceed();
    }
}
```

了解了 MethodBeforeAdviceInterceptor 的实现原理，对于其他的 advice 的实现也是可以举一反三的，比如对于 AfterReturningAdviceInterceptor 的实现，它和 MethodBeforeAdviceInterceptor 实现不同的地方，就是在 AfterReturningAdviceInterceptor 的 invoke 方法中，先完成了 MethodInvocation 的 proceed 调用，也就是目标对象的方法调用，然后再启动 advice 的 afterReturning 回调，这些实现原理在代码中可以很清楚地看到，如代码清单 3-30 所示。

代码清单 3-30 AfterReturningAdviceInterceptor 的实现

```
public class AfterReturningAdviceInterceptor implements MethodInterceptor,
    AfterAdvice,Serializable {
    private final AfterReturningAdvice advice;
    /**
     * Create a new AfterPeturningAdviceInterceptor for the given advice.
     * @param advice the AfterReturningAdvice to wrap
     */
    public AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {
        Assert.notNull(advice, "Advice must not be null");
        this.advice = advice;
    }
    public Object invoke(MethodInvocation mi) throws Throwable {
        Object retVal = mi.proceed();
        this.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(),
        mi.getThis());
        return retVal;
    }
}
```

对于 ThrowAdvice 的实现原理，和上面两种情况是非常类似的，也是封装在对应的 AdviceInterceptor 中实现的，如代码清单 3-31 所示。只是相对于 MethodBeforeAdvice 和 AfterReturningAdvice 的回调方法调用，ThrowAdvice 的回调方法调用比前两者的实现要复杂一些：它维护了 exceptionHandlerMap 来对应不同的方法调用场景，这个 exceptionHandlerMap 中 handler 的取得，是与触发 ThrowAdvise 增强的异常相关的。

代码清单 3-31 ThrowsAdviceInterceptor 的实现

```
public class ThrowsAdviceInterceptor implements MethodInterceptor, AfterAdvice {
    private static final String AFTER_THROWING = "afterThrowing";
```

Download at http://www.pin5i.com/

```
    private static final Log logger = LogFactory.getLog(ThrowsAdviceInterceptor.
    class);
    private final Object throwsAdvice;
    // Methods on throws advice, keyed by exception class.
    private final Map<Class, Method> exceptionHandlerMap = new HashMap<Class,
    Method>();
    public ThrowsAdviceInterceptor(Object throwsAdvice) {
        Assert.notNull(throwsAdvice, "Advice must not be null");
        this.throwsAdvice = throwsAdvice;
        //配置ThrowsAdvice的回调方法。
        Method[] methods = throwsAdvice.getClass().getMethods();
        for (Method method : methods) {
            if (method.getName().equals(AFTER_THROWING) &&
                (method.getParameterTypes().length == 1 || method.getParameter
                Types().length == 4) &&
    Throwable.class.isAssignableFrom(method.getParameterTypes()[method.getParame
    terTypes().length - 1])
                ) {
                // Have an exception handler.
    this.exceptionHandlerMap.put(method.getParameterTypes()
    [method.getParameterTypes().length - 1], method);
                if (logger.isDebugEnabled()) {
                    logger.debug("Found exception handler method: " + method);
                }
            }
        }
        if (this.exceptionHandlerMap.isEmpty()) {
            throw new IllegalArgumentException(
                    "At least one handler method must be found in class [" +
                    throwsAdvice.getClass() + "]");
        }
    }
    public int getHandlerMethodCount() {
        return this.exceptionHandlerMap.size();
    }
    private Method getExceptionHandler(Throwable exception) {
        Class exceptionClass = exception.getClass();
        if (logger.isTraceEnabled()) {
            logger.trace("Trying to find handler for exception of type [" +
            exceptionClass.getName() + "]");
        }
        Method handler = this.exceptionHandlerMap.get(exceptionClass);
        while (handler == null && !exceptionClass.equals(Throwable.class)) {
            exceptionClass = exceptionClass.getSuperclass();
            handler = this.exceptionHandlerMap.get(exceptionClass);
        }
        if (handler != null && logger.isDebugEnabled()) {
            logger.debug("Found handler for exception of type [" +
            exceptionClass.getName() + "]: " + handler);
        }
        return handler;
    }
    public Object invoke(MethodInvocation mi) throws Throwable {
        /**
        *把对目标对象的方法调用放入try/catch中，并在catch中触发ThrowAdvice的回调，
        *然后把异常接着向外抛出，不做过多的处理。
        */
        try {
            return mi.proceed();
```

Download at http://www.pin5i.com/

```
        }
        catch (Throwable ex) {
            Method handlerMethod = getExceptionHandler(ex);
            if (handlerMethod != null) {
                invokeHandlerMethod(mi, ex, handlerMethod);
            }
            throw ex;
        }
    }
    //通过反射启动对 ThrowAdvice 回调方法的调用。
    private void invokeHandlerMethod(MethodInvocation mi, Throwable ex, Method
    method) throws Throwable {
        Object[] handlerArgs;
        if (method.getParameterTypes().length == 1) {
            handlerArgs = new Object[] { ex };
        }
        else {
            handlerArgs = new Object[] {mi.getMethod(), mi.getArguments(), mi.
            getThis(), ex};
        }
        try {
            method.invoke(this.throwsAdvice, handlerArgs);
        }
        catch (InvocationTargetException targetEx) {
            throw targetEx.getTargetException();
        }
    }
}
```

3.3.7 ProxyFactory 实现 AOP

我们回到 3.2.1 节中提到的 Spring AOP 的类层次关系，从中可以看到除了使用 ProxyFactoryBean 实现 AOP 应用之外，还可以使用 ProxyFactory 来实现 Spring AOP 的功能，只是在使用 ProxyFactory 的时候，需要编程式的完成 AOP 应用的设置。我们举一个使用 ProxyFactory 的例子，如代码清单 3-32 所示。

代码清单 3-32 ProxyFactory 的使用

```
TargetImpl target = new TargetImpl();
ProxyFactory aopFactory = new ProxyFactory(target);
aopFactory.addAdvisor(yourAdvisor);
aopFactory.addAdvice(yourAdvice);
TargetImpl targetProxy = (TargetImpl)aopFactory.getProxy();
```

对于使用 ProxyFactory 实现 AOP 功能，它的实现原理与 ProxyFactoryBean 的实现原理是一样的，只是在最外层的表现形式上有所不同：ProxyFactory 没有使用 FactoryBean 的 IoC 封装，而是通过直接继承 ProxyCreatorSupport 的功能来完成 AOP 的属性配置。至于其他 ProxyCreatorSupport 的子类，ProxyFactory 取得 AopProxy 代理对象其实是和 ProxyFactoryBean 是一样的。一般来说，也是通过 getProxy 为入口，由 DefaultAopProxy-Factory 来完成的，关于取得 AopProxy 的详细分析和以后对拦截器调用的实现原理，前面我们都分析过了，这里就不再重复了。对 ProxyFactory 实现感兴趣的读者，可以从以下的源代码中看到它与 ProxyFactoryBean 实现上不同的地方，ProxyFactory 的实现代码如代码清单 3-33 所示。从代码清单上可以看到在 ProxyFactory 的 getProxy 方法，由这个方法去取得

Download at http://www.pin5i.com/

AopProxy 代理对象，这个 getProxy 方法的实现使用了 ProxyFactory 的基类 ProxyCreatorSupport 的 createProxy 方法来生成 AopProxy 代理对象，而这个 AopProxy 代理对象的生成是由 AopProxyFactory 来完成的，它会生成 JDK 或者 CGLIB 的代理对象。从这里的 getProxy 的实现开始，ProxyFactory 和 ProxyFactoryBean 在 AOP 的功能实现上基本上都是一样的，包括以后拦截器的调用等。

代码清单 3-33 ProxyFactory 的实现

```
public class ProxyFactory extends ProxyCreatorSupport {
    public ProxyFactory() {
    }
    public ProxyFactory(Object target) {
        Assert.notNull(target, "Target object must not be null");
        setInterfaces(ClassUtils.getAllInterfaces(target));
        setTarget(target);
    }
    public ProxyFactory(Class[] proxyInterfaces) {
        setInterfaces(proxyInterfaces);
    }
public ProxyFactory(Class proxyInterface, Interceptor interceptor) {
        addInterface(proxyInterface);
        addAdvice(interceptor);
    }
public ProxyFactory(Class proxyInterface, TargetSource targetSource) {
        addInterface(proxyInterface);
        setTargetSource(targetSource);
    }
    public <T> T getProxy() {
        return (T) createAopProxy().getProxy();
    }
    public <T> T  getProxy(ClassLoader classLoader) {
        return (T) createAopProxy().getProxy(classLoader);
    }
    public static Object getProxy(Class proxyInterface, Interceptor interceptor) {
        return new ProxyFactory(proxyInterface, interceptor).getProxy();
    }
    public static Object getProxy(Class proxyInterface, TargetSource targetSource) {
        return new ProxyFactory(proxyInterface, targetSource).getProxy();
    }
public static Object getProxy(TargetSource targetSource) {
        if (targetSource.getTargetClass() == null) {
            throw new IllegalArgumentException("Cannot create class proxy for
    TargetSource with null target class");
        }
        ProxyFactory proxyFactory = new ProxyFactory();
        proxyFactory.setTargetSource(targetSource);
        proxyFactory.setProxyTargetClass(true);
        return proxyFactory.getProxy();
    }
}
```

Download at http://www.pin5i.com/

3.4 Spring AOP 的高级特性

了解了 Spring AOP 的基本实现，下面我们来看一个关于使用 Spring AOP 高级特性的例子，来了解它的实现原理。在使用 Spring AOP 时，对目标对象的增强是通过拦截器来完成的。对于一些应用场合，需要对目标对象本身进行一些处理，比如，如何从一个对象池或对象工厂中获得目标对象等。这样，我们需要使用 Spring 的 TargetSource 接口特性，在这里，我们把这类 AOP 特性当作高级特性的一种，从这些 AOP 特性的实现原理的了解上，可以看到对 AOP 基本特性的灵活运用。

在 Spring 中，提供了许多现成的 TargetSource 实现，比如下面的 HotSwappableTarget-Source，这个 HotSwappableTargetSource 使得用户可以以线程安全的方式切换目标对象，提供所谓的热交换功能。这个特性是很有用的，尽管它的开启需要 AOP 应用进行显式的配置。但这个配置并不复杂，在使用时只需要把这个 HotSwappableTargetSource 配置到 ProxyFactoryBean 的 target 属性就可以了，在需要更换真正的目标对象时，调用 HotSwappableTargetSource 的 swap 方法就可以完成。由此可见，对 HotSwappableTarget Source 的热交换功能的使用，是需要触发 swap 方法调用的。这个 swap 方法的实现很简单，它完成 target 对象的替换，也就是说，它使用新的 target 对象来替换原有的 target 对象。为了保证线程安全，需要把这个替换方法设为 synchronized 方法，如代码清单 3-34 所示。

代码清单 3-34 HotSwappableTargetSource 的 swap 方法

```
public synchronized Object swap(Object newTarget) throws IllegalArgumentException {
    Assert.notNull(newTarget, "Target object must not be null");
    Object old = this.target;
    this.target = newTarget;
    return old;
}
public synchronized Object getTarget() {
    return this.target;
}
```

这个 target 是怎样在 AOP 中起作用的呢？了解一下对 getTarget 的调用就很清楚了，这个 HotSwappableTargetSource 只是对真正的 target 做了一个简单的封装，以提供热交换的能力，并没有其他特别之处。对 getTarget 的方法调用关系，如图 3-12 所示。

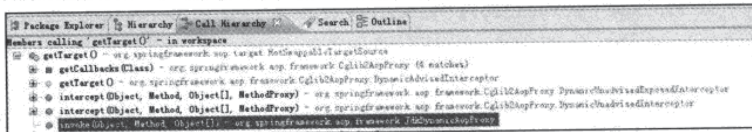


图 3-12 AOP 对 getTarget 的调用

我们以 JdkDynamicAopProxy 的实现为例子，可以看到在 AOP 对 Proxy 代理对象进行 invoke 方法调用的时候，会使用这个 getTarget 调用取得真正的目标对象，如果已经调用过 swap 方法完成目标对象的热交换，那么交给 AOP 的已经是交换后的目标对象了，如代码清单 3-35 所示。具体来说，在 invoke 方法中，我们看到代理对象的取得，是在 AopProxy 代理对象的拦截器起作用之前，通过 targetSource.getTarget() 的调用来取得的，而这个代理对象是否被

Download at http://www.pin5i.com/

更换过，是由对 swap 方法的调用来负责的。因而，在 invoke 方法中，可以看到对于使用了什么样的代理对象，都不会对拦截器的行为做任何的改变。

代码清单 3-35 invoke 获取目标对象

```
target = targetSource.getTarget();
if (target != null) {
    targetClass = target.getClass();
}
// Get the interception chain for this method.
// 这里获得定义好的拦截器链。
List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice
    (method, targetClass);
/**
* Check whether we have any advice. If we don't, we can fallback on direct
* reflective invocation of the target, and avoid creating a MethodInvocation.
*/
// 如果没有设定拦截器，那么就直接调用 target 的对应方法。
if (chain.isEmpty()) {
    /**
    * We can skip creating a MethodInvocation: just invoke the target directly
    * Note that the final invoker must be an InvokerInterceptor so we know it does
    * nothing but a reflective operation on the target, and no hot swapping or
    * fancy proxying.
    */
    retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
}
else {
    // We need to create a method invocation...
    // 如果有拦截器的设定，那么需要调用拦截器之后才调用目标对象的相应方法。
    //通过构造一个 ReflectiveMethodInvocation 来实现。
invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass,
    chain);
    // Proceed to the joinpoint through the interceptor chain.
    retVal = invocation.proceed();
}
// Massage return value if necessary.
if (retVal != null && retVal == target && method.getReturnType().isInstance(proxy) &&
    !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
    /**
    * Special case: it returned "this" and the return type of the method
    * is type-compatible. Note that we can't help if the target sets
    * a reference to itself in another returned object.
    */
    retVal = proxy;
}
return retVal;
```

通过这个 getTarget 方法，完成了 HotSwappableTargetSource 与 AOP 的集成。这个热交换功能为 AOP 的使用提供了更多便利，对构建应用的基础服务是非常有帮助的，比如可以在运行时支持经常改变的对象的重新配置。对于其他 AOP 的高级特性，有兴趣的读者可以结合自己的需要进行分析。

3.5 小结

在本章中，我们对最基本 Spring AOP 的实现方式进行了解析。具体地说，通过使用

Download at http://www.pin5i.com/

ProxyFactoryBean/ProxyFactory 的实现原理作为例子，对 Spring AOP 的基本实现和工作原理进行了一些梳理和分析。ProxyFactoryBean 是在 IoC 环境中创建代理的一个很灵活的方法，与其他方法相比，虽然有些繁琐，但却并不妨碍我们从 ProxyFactoryBean 入手，去了解 AOP 在 Spring 中的基本实现。

在 Spring AOP 的基本实现中，可以了解 Spring 是如何得到 AopProxy 代理对象的，以及它是如何利用 AopProxy 代理对象来对拦截器进行处理的。Proxy 代理对象的使用，在 Spring AOP 的实现过程中是非常重要的一个部分，Spring AOP 充分利用了像 Java 的 Proxy、反射技术以及第三方字节码技术实现 CGLIB 这些技术方案，通过这些技术完成了 AOP 需要的 AopProxy 代理对象的生成。回顾整个通过 ProxyFactoryBean 实现 AOP 的过程我们可以看到，在它的实现中，首先需要对目标对象进行正确的配置，以及对拦截器的正确配置，以便 AopProxy 代理对象得以顺利产生；这些配置既可以通过配置 ProxyFactoryBean 的属性来完成，也可以通过编程式的使用 ProxyFactory 来实现。这两种 AOP 的使用方式，只是在表面配置的方式上不同而已，对于内在的 AOP 实现原理它们是一样的。在生成 AopProxy 代理对象的时候，Spring AOP 设计了专门的 AopProxyFactory 作为 AopProxy 代理对象的生产工厂，由它来负责产生相应的 AopProxy 代理对象，在使用 ProxyFactoryBean 来得到 AopProxy 代理对象的时候，它默认地使用的 AopProxy 代理对象的生产工厂是 DefaultAopProxyFactory 对象。这个对象是一个在 AopProxy 生产过程中比较重要的类，它定义了 AopProxy 代理对象的生成策略，从而决定使用哪一种 AopProxy 代理对象的生成技术，是使用 JDK 的 Proxy 类还是使用 CGLIB 来完成生产任务。而对于最终的 AopProxy 代理对象的产生，则是交给 JdkDynamicAopProxy 和 Cglib2AopProxy 这两个具体的工厂来完成 AopProxy 代理对象的生产的，它们使用了不同的生产技术，一种使用的是 JDK 的 Proxy 技术，它使用 InvocationHandler 对象的 invoke 完成回调，而另一种则是使用 CGLIB 的技术来生成 AopProxy 代理对象。

在得到 AopProxy 代理对象后，在代理的接口方法被调用执行的时候，也就是当 AopProxy 暴露代理的方法被调用的时候，前面定义的 Proxy 机制就起作用了。当 Proxy 对象暴露的方法被调用时，并不是直接地运行目标对象的调用方法，而是会根据 Proxy 的定义，改变了原有的目标对象方法调用的运行轨迹。这种改变体现在，首先会触发对这些方法调用进行拦截，这些拦截为对目标调用的功能增强提供了工作空间；拦截过程在 JDK 的 Proxy 代理对象中，是通过 invoke 方法来完成的，这个 invoke 方法是虚拟机触发的一个回调。而在 CGLIB 的 Proxy 代理对象中，拦截是由设置好的回调 callback 方法来完成的。有了这些拦截器的拦截作用，才会有 AOP 切面增强大显身手的舞台。

在 ProxyFactoryBean 的回调中，首先会根据配置来对拦截器是否与当前的调用方法相匹配来进行判断。如果当前的调用方法与配置的拦截器相匹配，那么相应的拦截器就会开始发挥作用，这个过程是一个遍历的过程，它会遍历在 Proxy 代理对象中设置的拦截器链中的所有拦截器。经过这个过程后，在代理对象中定义好的拦截器链里的拦截器会被逐一调用，直到整个拦截器的调用完成为止。在对拦截器的调用完成以后，才是我们最后看到的对目标对象（target）的方法调用。这样，一个普通的 Java 对象的功能就得到了增强，这种增强和现有的目标对象的设计是正交解耦的，这也是 AOP 需要达到的一个目标。

在拦截器的调用过程中，实际上已经封装了 Spring 对 AOP 的实现，比如对各种通知器的增强

Download at http://www.pin5i.com/

织入功能。尽管我们在使用 Spring AOP 的时候，看到的是一些 advice 的使用，但实际上这些 AOP 应用中接触到的这些 advice 通知是不能直接对目标对象完成增强的，为了完成 AOP 应用需要的对目标对象的增强，Spring AOP 做了许多工作。这些工作包括，对应于每种 advice 通知，Spring 设计了对应的 AdviceAdapter 通知适配器，正是这些 AdviceAdapter 通知适配器，实现了 advice 通知对目标对象的不同的增强方式。对于这些 AdviceAdapter 通知适配器，在 AopProxy 代理对象的回调方法中，需要有一个注册机制，它们才能发挥作用。完成这个注册过程之后，实际上在拦截器链中运行的拦截器，已经是经过这些 AdviceAdapter 适配过的拦截器了。有了这些拦截器，再去结合 AopProxy 代理对象的拦截回调机制，才能够让 advice 通知对目标对象的增强作用实实在在地发生。谁知盘中餐，粒粒皆辛苦，在软件开发的世界里，真是没有什么免费午餐。看起来简洁易用的 AOP，和 IoC 容器的实现一样，背后同样蕴含着许多艰苦的努力。

熟悉 AOP 使用的读者还知道，除了提供 AOP 的一些基本功能之外，Spring 还提供了许多其他高级特性让用户更加方便地使用 AOP。对于这些高级特性，在本章中，我们选取了 HotSwappableTargetSource 来对它的实现原理进行分析，一叶知秋，一管窥豹，希望能够在这里为那些对 AOP 其他特性的实现感兴趣的读者打开一扇窗。

在本章中，我们提到了 Proxy、反射等 Java 虚拟机特性的使用，CGLIB 的使用以及在它们建立的 Proxy 对象的基础上，对拦截器特性的灵活运用，这些特性都是我们掌握本章内容的背景知识和重要基础。同时，不妨可以反过来看，通过了解本章中 AOP 的实现原理，也为我们使用这些 Java 虚拟机的特性以及 CGLIB 的技术，提供了生动而精彩的应用案例。在 AOP 的实现中，还有一个值得注意的地方就是，在 ProxyFactoryBean 得到 advisor 配置的实现过程中，是通过回调 IoC 容器的 getBean 方法来完成的，这个处理既简洁又巧妙，是灵活使用 IoC 容器功能的一个非常好的实例。以上这些，都是在本章中，除了 Spring AOP 实现原理本身之外，非常值得我们学习和研究的地方。

Download at http://www.pin5i.com/

Download at http://www.pin5i.com/

第二部分

Spring组件实现篇

Download at http://www.pin5i.com/

Spring 组件涵盖的范围很广，比如 Web 应用环境与 Spring MVC、JDBC 应用、O/R 映射、事务处理、远端调用等。但是，实际上 Spring 的组件并非只是局限于这几个模块，这里涉及的只是整个 Spring 组件体系中的很小一部分。

大家对 Spring 的组件系统或多或少都有些直接的体会，因为这些组件毕竟是整个 Spring 系统中最为活跃和非常引人入胜的部分。Spring 的目标是为 Java EE 应用开发人员提供便利，这些便利往往体现在对这些 Spring 组件的使用上。有了这些组件，有了 Spring 支持的 POJO 开发，在把开发人员从传统 Java EE 开发方式中解放出来的同时，也为 Java EE 应用开发提供了犀利的武器。随着技术和市场需求的发展，这些纳入到 Spring 体系的组件实现也在不断地发展和丰富，给人日新月异之感。一方面，组件的种类在不断增加，在同一种类的组件里面，往往集成了若干个优秀的具体产品实现来满足用户不同的技术选择；另一方面，随着组件功能的丰富和产品升级，Spring 组件的相应实现部分也会随之不断地更新。

乱花渐欲迷人眼，浅草才能没马蹄。选择多了，对应用开发人员对组件的评估和选择能力也提出了更高的要求。为了更好地使用 Spring 平台，了解这些企业应用组件在 Spring 中的实现原理，的确是一件能够提高我们的技术水平和知识修养的事情，并且对我们使用 Spring 进行应用开发，会有直接的帮助。

本书只选取了在组件系统中，应用最普遍的一些模块进行初步的探讨和分析，一方面是因为大家对这些组件的使用都比较熟悉，另一方面是因为作为 Java EE 应用平台的 Spring，尽管集成了许多优秀的组件实现为应用开发服务，但是只要能透彻分析其中具有代表性的组件，对其他组件进行分析时应该也能够举一反三。

注意 笔者深知这部分内容博大精深，尽管已经尽了自己最大的努力，却难免在分析中有所疏忽和遗漏，敬请读者批评指正。通过这些粗浅的分析，希望能提高读者自己通过分析源代码来了解平台设计原理的能力和兴趣。在开发中应用 Spring 组件时，如果遇到迷惑不解之处，不妨从这些组件的源代码中探个究竟。

从技术的依赖层次上看，这些组件的实现都是建立在 Spring 的核心（前面已经讲过的 IoC/AOP 模块）的基础上的。虽然说涉及企业应用的方方面面，同时每一个组件的实现都自成体系，但了解其底层的技术实现却是提高我们理解这些企业应用组件在 Spring 中实现的有效手段，也对提高我们使用这些组件的能力有事半功倍的效果。我们已经深入 Spring 丛林的腹地，入宝山岂能空手归，让我们继续努力前进，去探寻我们的宝藏吧！

Download at http://www.pin5i.com/

第 4 章

Spring MVC 与 Web 环境

松下问童子，言师采药去。
只在此山中，云深不知处。
——【唐】贾岛《寻隐者不遇》

4.1 概述

使用 Spring 的开发人员，对很多 Web 开发技术都比较熟悉，比如 SSH 技术架构，也就是我们熟悉的 Structs+Spring+Hibernate 的技术组合，它们是 Web 应用开发中最常用的技术架构之一。众所周知，这个技术架构以 Structs 作为 Web 框架来帮助应用构建 UI，Spring 作为应用平台，Hibernate 作为 O/R 映射的数据持久化层实现。这个技术组合全部由开源软件组成，其中 Struts 是 Apache 旗下的一个项目，Hibernate 已经成为 JBoss/RedHat 产品组合中的一员，而 Spring 作为一个开源应用平台，与其他成熟的商用应用服务器一样，也是早就深入人心。看来这个组合的流行，也并非没有它的道理。

在这个技术组合中，Hibernate 是一个独立的 ORM 数据持久化实现产品，对于 ORM 数据持久化实现，Spring 本身并不提供自己独立的解决方案。在 Spring 中，对于数据持久化的功能实现，Spring 提供了 JDBC 的封装，虽然在 Spring JDBC 中，可以实现一些简单的数据记录到 Java 对象的数据转换，但和 Hibernate 相比，毕竟显得有些单薄，还不能算是一个在 ORM 领域独当一面的独立产品。Spring 在这持久化层支持的处理上，与在 Web UI 层的情况有些不同。在 Web 层，Spring 提供了 MVC 解决方案，也就是本章中，我们将对它的实现原理进行详细的分析。

为什么会演进成这样一种情况，是由 Spring 自身的发展策略来决定的。我们现在无从知晓，也许在这里面蕴含着不少有趣的小故事，也许都在 Rod Johnson 的记忆中吧，都云作者痴，谁解其中味，让我们这些 Spring 的粉丝，一起期待他的回忆录吧，看看会不会有机会了解这些平台发展背后的有趣往事。作为一个开放的应用平台，Spring 没有理由不为 Hibernate 的使用提供平台级别的支持，因为 Hibernate 是一个足够流行的持久化框架，失去了对 Hibernate 的支持，也许会失去许多对 Hibernate 情有独钟的应用开发者在选择应用平台时的青睐。从另一角度上看，Spring 的 MVC 框架和 Struts 相比，Spring JDBC 封装和 Hibernate 相比，也许是因为竞争力有些欠缺的原因，而没有能够成为普遍应用技术组合中的组成部分而被广泛认可。毕竟术业有专攻，闻道有先后，Spring 在技术组合中作为应用平台的枢纽地位是不容置疑的。从这点上，可以从另一个方面看到 Spring 江湖地位的确立方式，以及由此看到 Spring 所具备的技术优势及其产品定位。

Download at http://www.pin5i.com/

总的来说，在这个技术组合中 Spring 起到的是一个应用平台的作用，有点像企业应用的"操作系统"，从而为企业应用资源的使用提供一个一致地环境。具体地说，Spring 提供的平台特性有 IoC 容器、AOP、事务处理、持久化驱动等。这里提到的平台有点像是基础设施，以日常生活中常见的基础设施的使用为例来对比说明，通过这种方式，也许能帮助读者建立更形象的理解。

在软件产品开发中，如果某些特性的使用比较普遍，那么这些特性往往可以考虑作为平台特性来实现，然后通过对平台特性进行有效的封装，把它开放给应用使用，从而有效地提高应用开发效率。就像在现代社会中，有了电力、网络、铁路、航空这些基础设施后，整个社会的运行效率和机器工业时代以前相比，简直不可同日而语。从另一个角度上看，这些所谓的平台特性也是相对的，它们往往在一开始也是作为应用特性来实现的，就像电力的使用一开始也是贵族才能享有的便利一样。旧时王谢堂前燕，飞入寻常百姓家，随着社会和经济的发展，电慢慢开始进入普通老百姓的日常生活，成为人们日常生活中不可缺少的一部分。这种进入是以一种基础设施和成熟产品的方式完成的，这样才能具备规模效应，去分摊基础设施建设的前期建设成本。

基于这种对比，我们可以惊奇地发现，对于基础软件的开发和应用，如果采用开源软件的开发方式来完成，可以看到开源的开发方式发挥着非常奇妙的作用。一方面，通过开源它可以广泛地收集基础需求；另一方面，通过开源又有效地分摊了前期的开发、测试以及一些应用培育的成本。Linux 的成功就是一个例子。同样地，Spring 的蓬勃发展也有类似的因素在发挥作用。

一个优秀的平台对提高应用的开发效率是大有帮助的，它能让应用开发者站在巨人的肩膀上，这一点毋庸置疑。打个比方来说，由于现在有铁路、飞机和轮船，我们出门旅行时再也不需要完全依赖双腿去完成长途跋涉。在软件产开发中也一样，因为平台已经提供了许多现成的特性实现，不需要应用对这些基础特性从头开始构思、设计和实现，而重点应该在于关注应用的特性需求本身。

在 Structs+Spring+Hibernate 的技术组合中。虽然 Web 层的应用框架是由 Structs 来完成的，但 Spring 自己也带有 MVC 框架供用户开发 Web 应用提供支持，对于 Web UI 的开发来说，通常也是不错的选择。在本章中，我们会对 Spring 与 Web 应用相关的实现进行分析，旨在让读者了解 Spring 作为应用平台是怎样在 Web 应用中起作用的，这个分析实际上从实现原理上可以大致分为两个部分，一部分着重于 Spring 的 IoC 容器是怎样在 Web 应用环境中发挥作用的，对于使用其他的 Web 框架的应用，比如 SSH 中使用 Structs 作为 Web 框架，一定要考虑它在 Web 环境中如何完成与 IoC 容器的集成，这个部分的分析是非常具有参考价值的；另一部分的内容着重于分析 Spring 自己的 Web MVC 框架的实现原理，通过分析 Spring MVC 的实现，可以帮助读者举一反三，深入地了解 MVC 框架的工作原理。

与前面分析源代码的部分一样，我们在这里需要将 `org.springframework.web` 以及 `org.springframework.web.servlet` 这两个包所在的工程导入到 Eclipse 中。

4.2 Web 环境中的 Spring MVC

第 2 章分析了 IoC 容器的基本实现，下面我们来看看在典型的 Web 环境中，Spring IoC 容器是

Download at http://www.pin5i.com/

如何在 Web 环境中被载入并起作用的。Spring 并不是天生就能在 Web 容器中起作用的，同样也需要一个启动过程，把自己的核心 IoC 容器导入，并在 Web 容器中建立起来。具体说来，这个启动过程就是在 Web 容器的启动过程中载入 IoC 容器并将其初始化，然后为我们提供 Bean 的管理服务的过程。在建立起 MVC 框架以后，可以对 Web 请求执行相应的处理。这里以 Tomcat 作为使用的 Web 容器为例子来进行分析，它使用 web.xml 作为应用的部署描述符。我们在 web.xml 中常常看到这样的与 Spring 相关的部署描述，如代码清单 4-1 所示。

代码清单 4-1　Tomcat 的 web.xml 对 Spring MVC 的部署描述

```
<servlet>
    <servlet-name>sample</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>sample</servelet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
<listener>
<listener-class>
  org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

我们在这里看到的部署描述是 Spring MVC 与 Tomcat 的接口部分，在这个部署描述文件里，首先定义了一个 Servlet 对象，它是 Spring MVC 的 DispatcherServlet。该 Servlet 是 MVC 中很重要的一个类，起着分发请求的作用，它的实现原理是我们下面要详细分析的内容，这里暂且略过。

同时，在部署描述中，为这个 DispatcherServlet 定义了对应的 URL 映射，它为这个 Servlet 指定了需要处理的 HTTP 请求。我们还看到对 context-param 参数的配置，这个 context-param 参数配置，用来指定 Spring IoC 容器读取 Bean 定义的 XML 文件的路径。在这里，这个配置文件被定义为/WEB-INF/applicationContext.xml，在这个文件中，可以看到 Spring 应用的 Bean 配置。最后，作为 Spring MVC 的启动类，ContextLoaderListener 被定义为一个监听器，这个监听器是和 Web 服务器的生命周期以及与之关联的事件处理联系在一起的，由 ContextLoaderListener 监听器负责完成 IoC 容器在 Web 环境中的启动工作，这个启动过程也是我们下面要详细分析的内容。

我们看到的 DispatchServlet 和 ContextLoaderListener 提供了在 Web 容器中对 Spring 的入口，也就是说 ServletContext 为 Spring 的 IoC 容器提供了一个宿主环境，在这个宿主环境中建立起一个 IoC 容器的体系。这个 IoC 容器体系是通过 ContextLoaderListener 的初始化来建立的。然后，把 DsipatchServlet 这个 Servlet 作为 Spring MVC 处理 Web 请求的转发器建立起来，有了这些基本配置，建立在 IoC 容器基础上的 Spring MVC 就可以正常地发挥它的作用了。在了解 Spring MVC 在 Web 容器中的配置以后，我们先来看看 IoC 容器在 Spring MVC 中的启动过程是怎样实现的。

Download at http://www.pin5i.com/

4.3 IoC 容器在 Spring MVC 中的启动

4.3.1 Web 容器中的上下文

IoC 容器的启动过程就是建立上下文的过程，该上下文是与 ServletContext 相伴而生的，同时也是 IoC 容器在 Web 应用环境中的具体表现之一。我们称由 ContextLoaderListener 启动的上下文为根上下文，在这个根上下文的基础上，还有一个与 Web MVC 相关的上下文用来保存控制器（DispatcherServlet）需要的 MVC 对象，作为根上下文的子上下文，构成了一个层次化的上下文体系。

在 Web 容器中启动 Spring 应用程序时，首先是建立根上下文，然后是建立这个上下文体系的过程。我们先从 Web 容器中的上下文入手，看看在 Web 环境中的上下文设置有哪些特别之处，然后再到 ContextLoaderListener 中去了解整个容器启动的过程。为了方便在 Web 环境中使用 IoC 容器，Spring 为 Web 应用提供了上下文的扩展接口 WebApplicationContext 来满足启动过程的需要，这个 WebApplicationContext 接口的类层次关系，如图 4-1 所示。

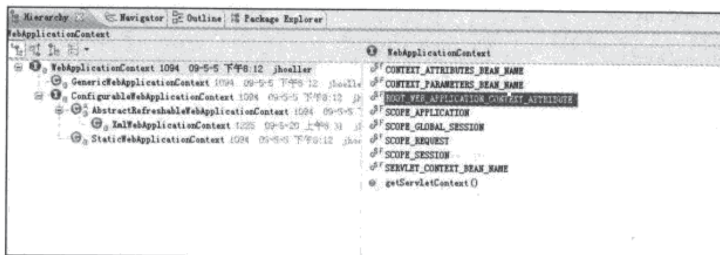


图 4-1 Spring MVC 的 WebApplicationContext 接口

这个接口类定义的接口方法比较简单，如代码清单 4-2 所示。在这个接口里，定义了一个 getServletContext 方法，通过这个方法可以得到当前 Web 容器的 Servlet 上下文环境，通过这个方法，相当于提供了一个 Web 容器级别的全局环境。

代码清单 4-2 WebApplicationContext 接口

```
public interface WebApplicationContext extends ApplicationContext {
    //这里定义的常量用于在 ServletContext 中存取根上下文。
    String ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE = WebApplicationContext.class.
    getName() + ".ROOT";
    /**
    *对 WebApplicationContext 来说，需要得到 Web 容器的 ServletContext，通过这个方法可以
    *取得 web 容器的 ServletContext。
    */
    ServletContext getServletContext();
}
```

在启动过程，Spring 会使用一个默认的 WebApplicationContext 实现作为 IoC 容器。这个默认使用的 IoC 容器就是 XmlWebApplicationContext，我们在图 4-2 中可以看到这个类的继承关系。

Download at http://www.pin5i.com/

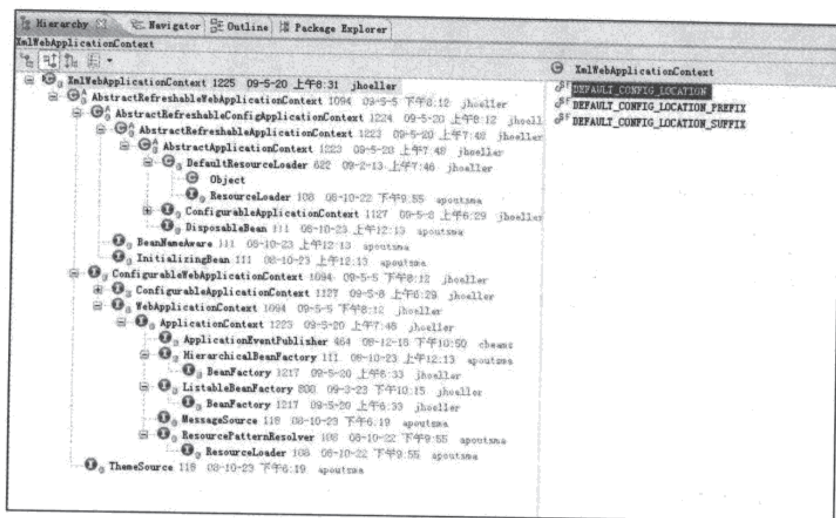


图 4-2 XmlWebApplicationContext 的继承关系

在这个继承关系中，XmlWebApplicationContext 是从 ApplicationContext 继承下来的，在基本的 ApplicationContext 的功能基础上，增加了对 Web 环境和 XML 配置定义的处理。在 XmlWebApplicationContext 的初始化过程中，Web 容器中的 IoC 容器被建立起来，从而在 Web 容器中建立起整个 Spring 应用。我们到 XmlWebApplicationContext 中去看看 IoC 容器的具体启动过程，如代码清单 4-3 所示。与对 IoC 容器的初始化的分析一样，我们同样看到了 loadBeanDefinition 对 BeanDefinition 的载入。在 Web 环境中，对定位 BeanDefinition 的 Resource 有特别的要求，对这个要求的处理体现在 getDefaultConfigLocations 方法的处理中。可以看到在这里使用了默认的 BeanDefinition 的配置路径，这个路径在 XmlWebApplicationContext 中已经作为一个常量定义好了，就是/WEB-INF/applicationContext.xml。

代码清单 4-3 XmlWebApplicationContext 的实现

```
public class XmlWebApplicationContext extends
AbstractRefreshableWebApplicationContext {
// Default config location for the root context.
/**
*这里是设置默认 BeanDefinition 的地方，在/WEB-INF/applicationContext.xml 文件里，如果不特
*殊指定其他文件，IoC 容器会从这里读取 BeanDefinition 来初始化 IoC 容器。
*/
public static final String DEFAULT_CONFIG_LOCATION = "/WEB-INF/
    applicationContext.xml";
// Default prefix for building a config location for a namespace.
public static final String DEFAULT_CONFIG_LOCATION_PREFIX = "/WEB-INF/";
// Default suffix for building a config location for a namespace .
public static final String DEFAULT_CONFIG_LOCATION_SUFFIX = ".xml";
// loadBeanDefinition 就像前面对 IoC 容器的分析一样，这个加载过程在容器 refresh()时启动。
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws
```

Download at http://www.pin5i.com/

```
        IOException {
        // Create a new XmlBeanDefinitionReader for the given BeanFactory.
        /**
        * 对于 XmlWebApplicationContext，当然是使用 XmlBeanDefinitionReader 来对 BeanDefi
        *nition 信息进行解析。
        */
        XmlBeanDefinitionReader beanDefinitionReader =
        new XmlBeanDefinitionReader (beanFactory);
        // Configure the bean definition reader with this context's resource loading
        environment.
        /**
        *这里设置 ResourceLoader，因为 XmlWebApplicationContext 是 DefaultResource 的子类，
        *所以这里同样会使用 DefaultResourceLoader 来定位 BeanDefinition。
        */
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
        /**
        * Allow a subclass to provide custom initialization of the reader,
        * then proceed with actually loading the bean definitions.
        */
        initBeanDefinitionReader(beanDefinitionReader);
        //这里使用定义好的 XmlBeanDefinitionReader 来载入 BeanDefinition。
        loadBeanDefinitions(beanDefinitionReader);
    }
    protected void initBeanDefinitionReader
        (XmlBeanDefinitionReader beanDefinition Reader) {
    }
    /**
    *如果有多个 BeanDefinition 的文件定义，需要逐个载入，都是通过 reader 来完成的，这个初始化过程
    *是由 refreshBeanFactory 方法来完成的，这里只是负责载入 BeanDefinition。
    */
    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
        BeansException, IOException {
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            for (String configLocation : configLocations) {
                reader.loadBeanDefinitions(configLocation);
            }
        }
    }
    //这里是取得 Resource 位置的地方，使用默认的配置位/WEB-INF/applicationContext.xml.
    protected String[] getDefaultConfigLocations() {
        if (getNamespace() != null) {
            return new String[] {DEFAULT_CONFIG_LOCATION_PREFIX + getNamespace() +
        DEFAULT_CONFIG_LOCATION_SUFFIX};
        }
        else {
            return new String[] {DEFAULT_CONFIG_LOCATION};
        }
     }
}
```

4.3.2 ContextLoader 建立 Web 环境的根上下文

对于 Spring 承载的 Web 应用而言，可以指定在 Web 应用程序启动时载入 IoC 容器（或者称为 WebApplicationContext）。这个功能是由诸如 ContextLoaderListener 这样的类来完成

Download at http://www.pin5i.com/

的，它是在 Web 容器中配置的监听器。这个 ContextLoaderListener 通过使用 ContextLoader 来完成实际的 WebApplicationContext，也就是 IoC 容器的初始化工作。ContextLoder 就像是 Spring 应用程序在 Web 容器中的启动器。这个启动过程是在 Web 容器中发生的，所以我们需要根据 Web 容器部署的要求来定义 ContextLoader，这些配置我们在概述中已经介绍了，这里就不重复了。

为了解 IoC 容器在 Web 容器中的启动原理，这里我们对启动器 ContextLoaderListener 的实现进行分析。这个监听器是启动根 IoC 容器并把它载入到 Web 容器的主要功能模块，也是整个 Spring Web 应用加载 IoC 的第一个地方。从加载过程我们可以看到，首先从 Servlet 事件中得到 ServletContext，然后可以读取配置在 web.xml 中的各个相关的属性值，接着 ContextLoder 会实例化 WebApplicationContext，并完成其载入和初始化过程。这个被初始化的第一个上下文作为根上下文而存在，当这个根上下文被载入后，它被绑定到 Web 应用程序的 ServletContex 上。任何需要访问根上下文的应用程序代码都可以从 WebApplicationContextUtils 类的静态方法中得到，具体取得根上下文的方法如下所示：

```
WebApplicationContext getWebApplicationContext(ServletContext sc)
```

下面我们来分析具体的根上下文的载入过程，在 ContextLoaderListener 中，实现的是 ServletContextListener 接口，这个接口里的函数会结合 Web 容器的生命周期而被调用。因为 ServletContextListener 是 ServletContext 的监听者，如果 ServletContext 发生变化，会触发出相应的事件，而监听器一直在对这些事件进行监听，如果接收到了监听的事件，就会作出预先设计好的响应动作。对于 ServletContext 的变化而触发的监听器的响应，举例来说，具体包括在服务器启动时，ServletContext 被创建的时候；服务器关闭时，ServletContext 将被销毁的时候等。对应这些事件及 Web 容器状态的变化，在监听器中定义了对应的事件响应的回调方法。比如在服务器启动时，ServletContextListener 的 contextInitialized()方法被调用，服务器将要关闭时，ServletContextListener 的 contextDestroyed()方法被调用。了解了 Web 容器中监听器的工作原理，下面我们看看在服务器启动时，ContextLoaderListener 的调用完成了什么，如代码清单 4-4 所示。在这个初始化回调中 ContextLoader 被创建，同时会使用创建出来的 ContextLoader 来完成 IoC 容器的初始化。

代码清单 4-4 ContextLoaderListener 的 context 初始化

```
public void contextInitialized(ServletContextEvent event) {
    //因为本身就是 ContextLoader 的子类，这里可以直接使用 ContextLoader 来初始化 IoC 容器。
    this.contextLoader = createContextLoader();
    if (this.contextLoader == null) {
        this.contextLoader = this;
    }
this.contextLoader.initWebApplicationContext(event.getServletContext());
}
```

具体的初始化工作交给 ContextLoader 来完成，我们到 ContextLoader 中去看看这个过程是怎样实现的，如代码清单 4-5 所示。在这个初始化过程中，完成根上下文在 Web 容器中的创建，这个根上下文是作为在 Web 容器中的唯一的实例而存在的，如果在这个初始化过程中，发现已经有根上下文被创建了，这里会抛出异常告知创建失败。在根上下文创建成功以后，会被存到 Web 容器的 ServletContext 中去，供需要时使用。存取这个根上下文的路径是由 Spring 预先设

Download at http://www.pin5i.com/

置好的，在 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE 的属性中定义了这个路径，这个路径默认地设置为：

```
ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE = WebApplicationContext.class.getName() +
    ".ROOT".
```

代码清单 4-5 ContextLoader 对 IoC 容器的初始化

```
//这里开始对 WebApplicationContext 进行初始化。
public WebApplicationContext initWebApplicationContext(ServletContext servletContext)
        throws IllegalStateException, BeansException {
    //判断在 ServletContext 中是否已经有根上下文存在。
    if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_
    CONTEXT_ATTRIBUTE) != null) {
        throw new IllegalStateException(
                "Cannot initialize context because there is already a root app-
                lication context present - " +
                "check whether you have multiple ContextLoader* definitions in
                your web.xml!");
    }
    servletContext.log("Initializing Spring root WebApplicationContext");
    if (logger.isInfoEnabled()) {
        logger.info("Root WebApplicationContext: initialization started");
    }
    long startTime = System.currentTimeMillis();
    try {
        // Determine parent for root web application context, if any.
        // 这里载入根上下文的双亲上下文。
        ApplicationContext parent = loadParentContext(servletContext);
        /**
        * Store context in local instance variable, to guarantee that
        * it is available on ServletContext shutdown.
        */
        /**
        * 这里创建在 ServletContext 中存储的根上下文 ROOT_WEB_APPLICATION_CONTEXT，同时
        * 把它存到 ServletContext 中，注意这里使用的 ServletContext 的属性值是 ROOT_WEB_
        * APPLICATION_CONTEXT_ATTRIBUTE，以后的应用都是根据这个属性值取得根上下文的。
        */
        this.context = createWebApplicationContext(servletContext, parent);
AservletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_
    TTRIBUTE, this.context);
currentContextPerThread.put(Thread.currentThread().getContextClassLoader(),this.
    context);
        if (logger.isDebugEnabled()) {
            logger.debug("Published root WebApplicationContext as ServletContext
    attribute with name [" +
WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE + "]");
        }
        if (logger.isInfoEnabled()) {
            long elapsedTime = System.currentTimeMillis() - startTime;
            logger.info("Root WebApplicationContext: initialization completed in
            " + elapsedTime + " ms");
        }
        return this.context;
    }
    catch (RuntimeException ex) {
        logger.error("Context initialization failed", ex);
servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_
    ATTRIBUTE, ex);
```

Download at http://www.pin5i.com/

```
            throw ex;
        }
        catch (Error err) {
            logger.error("Context initialization failed", err);
    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_
        ATTRIBUTE, err);
            throw err;
        }
    }
```

对于具体的根上下文的创建，可以参考代码清单 4-6 所示。在代码清单中，可以看到对根上下文的参数设置，比如为其设置了双亲上下文，对 ServletContext 的引用，等等。在这些基本的设置完成以后，通过 refresh 方法的调用，从而重启整个 IoC 容器，就像我们看到的对一般的 IoC 容器的初始化过程一样。关于这个 refresh 方法的调用，它的功能实现我们已经很熟悉了，这里就不再赘述了。

代码清单 4-6　创建根上下文

```
protected WebApplicationContext createWebApplicationContext(
        ServletContext servletContext, ApplicationContext parent) throws
    Beans Exception {
    //这里判断使用什么样的类在 Web 容器中作为 IoC 容器。
    Class contextClass = determineContextClass(servletContext);
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
        throw new ApplicationContextException("Custom context class
    [" + conte xtClass.getName() + "] is not of type [" +
                ConfigurableWebApplicationContext.class. getName() + "]");
    }
    /**
    *直接实例化需要产生的 IoC 容器，并设置 IoC 容器的各个参数，然后通过 refresh 启动容器的初始化。
    */
    ConfigurableWebApplicationContext wac =
            (ConfigurableWebApplicationContext)
    BeanUtils.instantiateClass(contextClass);
    wac.setId(servletContext.getServletContextName());
    //设置双亲上下文。
    wac.setParent(parent);
    //设置 ServletContext 以及配置文件的位置参数。
    wac.setServletContext(servletContext);
wac.setConfigLocation(servletContext.getInitParameter(CONFIG_LOCATION_PARAM));
    customizeContext(servletContext, wac);
    //启动容器的初始化，我们已经很熟悉的 refresh 调用。
    wac.refresh();
    return wac;
}
```

使用什么样的类作为上下文是在 determineContextClass 方法中确定的，如代码清单 4-7 所示。在确定使用何种 IoC 容器的过程中，可以看到应用可以通过在部署描述符中指定使用什么样的 IoC 容器，这个指定操作是通过 CONTEXT_CLASS_PARAM 参数的设置完成的。如果没有指定特定的 IoC 容器，将使用默认的 IoC 容器，也就是 XmlWebApplicationContext 对象来作为在 Web 环境中被使用的 IoC 容器。

代码清单 4-7　确定使用的 IoC 容器

```
protected Class determineContextClass(ServletContext servletContext) throws
    App licationContextException {
```

Download at http://www.pin5i.com/

```
        //这里读取在 ServletContext 中对 CONTEXT_CLASS_PARAM 参数的配置。
        String contextClassName = servletContext.getInitParameter(CONTEXT_CLASS_PARAM);
        /**
        *如果在 ServletContext 中配置了需要使用的 CONTEXT_CLASS，那就使用这个 class，
        *当然前提是这个 class 是可用的。
        */
        if (contextClassName != null) {
            try {
                return ClassUtils.forName(contextClassName,
        ClassUtils.getDefaultCla ssLoader());
            }
            catch (ClassNotFoundException ex) {
                throw new ApplicationContextException(
                        "Failed to load custom context class [" + contextClassName
                        + "]", ex);
            }
        }
        else {//如果没有额外的配置，那么使用默认的 ContextClass。
            contextClassName = defaultStrategies.getProperty(WebApplicationContext.
        class.getName());
            try {
                return ClassUtils.forName(contextClassName,
                ContextLoader.class.get ClassLoader());
            }
            catch (ClassNotFoundException ex) {
                throw new ApplicationContextException(
                        "Failed to load default context class [" + contextClassName
                        + "]", ex);
            }
        }
    }
```

这就是 IoC 容器在 Web 容器中的启动过程，与我们在应用中启动 IoC 容器的方式相类似，所不同的是这里需要考虑 Web 容器的环境特点。比如各种参数的设置，以及 IoC 容器与 Web 容器 ServletContext 的结合，等等。在初始化这个上下文以后，该上下文会被存储到 SevletContext 中，这样就建立了一个全局的关于整个应用的上下文。同时，在启动 Spring MVC 时，我们还会看到这个上下文会被以后的 DispatcherServlet 在进行自己持有的上下文的初始化时，设置为 DispatcherServlet 自带的上下文的双亲上下文。这个过程我们将在分析 Spring MVC 实现时清楚地看到。

4.4 Spring Web MVC 的启动

4.4.1 DispacherServlet 概述

在完成对 ContextLoaderListener 的初始化以后，Web 容器就开始初始化 DispatcherServlet，这个初始化的启动与在 web.xml 中对载入次序进行的定义有关。DispatcherServlet 会建立自己的上下文来持有 Spring MVC 的 Bean 对象，在建立这个自己持有的 IoC 容器的时候，会从 ServletContext 中得到根上下文作为 DispatcherServlet 持有上下文的双亲上下文。有了这个根上下文，然后再对自己持有的上下文进行初始化，最后把这个上下文保存到

Download at http://www.pin5i.com/

ServletContext 中，供以后检索和使用。

为了解这个过程，可以从 DispatcherServlet 的父类 FrameworkServlet 的代码入手，去探寻 DispatcherServlet 的启动过程，同时也是 Spring MVC 的启动过程。ApplicationContext 的创建过程和 ContextLoder 创建根上下文的过程有许多类似的地方，关于根上下文的创建，已经在上面的章节中分析过了。要查看 DispatcherServlet 的源代码，我们需要在 Eclipse 中导入 org.springframework.web.servlet 这个工程包。下面来看一下这个 DispatcherServlet 的类的继承关系，以便于对这个 DispatcherServlet 在 Spring MVC 框架中的地位有大致的了解，如图 4-3 所示。

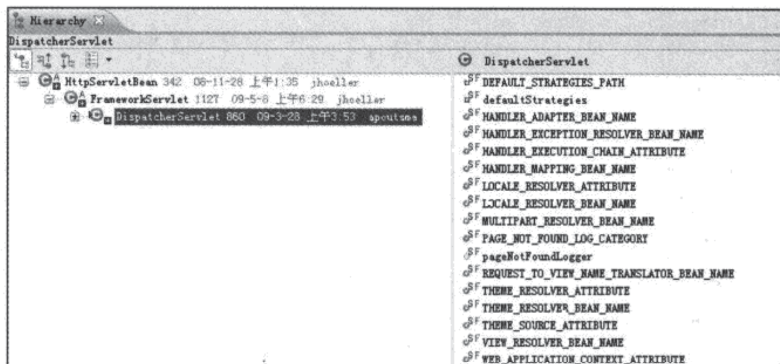


图 4-3 DispatcherSerlvet 的继承体系

4.4.2 DispacherServlet 的启动和初始化

DispatcherServlet 的启动与 Servlet 的启动过程是相联系的。在 Servlet 的初始化过程中，Servlet 的 init 方法会被调用，以进行初始化。我们在 DispatcherServlet 的基类 HttpServletBean 中可以看到这个初始化的过程，如代码清单 4-8 所示。在初始化开始时，需要读取配置在 ServletContext 中的 Bean 属性参数，这些属性参数被设置在 web.xml 的 Web 容器初始化参数中。使用编程式的方式来设置这些 Bean 属性，在这里可以看到对 PropertyValues、BeanWrapper 的使用。对于这些和依赖注入相关的类的使用，我们在分析 IoC 容器的初始化部分，尤其是在依赖注入实现分析的部分，有过亲密接触。只是这里的依赖注入是与 Web 容器初始化相关的，初始化过程由 HttpServletBean 来完成。

接着会执行 DispatcherServlet 持有的 IoC 容器的初始化过程，在这个初始化过程中，一个新的上下文被建立起来，这个 DispatcherServlet 持有的上下文被设置为根上下文的子上下文。可以大致认为，根上下文是和 Web 应用相对应的一个上下文，而 DispatcherServlet 持有的上下文是和 Servlet 对应的一个上下文，在一个 Web 应用中，往往可以容纳多个 Servlet 存在。与此相对应，对于应用在 Web 容器中的上下体系，也是很类似的，一个根上下文可以作为许多 Servlet 上下文的双亲上下文。了解了这一点，对在 Web 环境中 IoC 容器中的 Bean 设置和检

Download at http://www.pin5i.com/

索会有更多的了解，因为在向 IoC 容器 getBean 的时候，IoC 容器会首先向其双亲上下文去取 getBean，也就是说，对于根上下文中定义的 Bean，它们是可以被各个 Servlet 持有的上下文得到和共享的。在 DispatcherServlet 持有的上下文被建立起来以后，它也需要和其他 IoC 容器一样完成初始化，这个初始化也是通过 refresh 方法来完成的。最后，DispatcherServlet 给这个自己持有的上下文起一个名称，并把它设置到 Web 容器的上下文中，起的名字和在 web.xml 中设置的 DispatcherServlet 的 Servlet 名称有关，从而保证了这个上下文在 Web 环境上下文体系中的唯一性。

代码清单 4-8 DispatcherServlet 的初始化

```
public final void init() throws ServletException {
    if (logger.isDebugEnabled()) {
        logger.debug("Initializing servlet '" + getServletName() + "'");
    }
    // Set bean properties from init parameters.
    try {
        PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(),
    this.requiredProperties);
        BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
        ResourceLoader resourceLoader =
    new ServletContextResourceLoader(getServletContext());
        bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader));
        initBeanWrapper(bw);
        bw.setPropertyValues(pvs, true);
    }
    catch (BeansException ex) {
        logger.error("Failed to set bean properties on servlet '" +
    getServlet Name() + "'", ex);
        throw ex;
    }
    // Let subclasses do whatever initialization they like.
    // 调用子类的initServletBean进行具体的初始化。
    initServletBean();
    if (logger.isDebugEnabled()) {
        logger.debug("Servlet '" + getServletName() + "' configured successfully");
    }
}
// initServletBean的初始化过程在FrameworkServlet中完成。
protected final void initServletBean() throws ServletException, BeansException {
    getServletContext().log("Initializing Spring FrameworkServlet '" +
    getServletName() + "'");
    if (this.logger.isInfoEnabled()) {
        this.logger.info("FrameworkServlet '" + getServletName() +
    "': initialization started");
    }
    long startTime = System.currentTimeMillis();
    //这里初始化上下文。
    try {
        this.webApplicationContext = initWebApplicationContext();
        initFrameworkServlet();
    }
    catch (ServletException ex) {
        this.logger.error("Context initialization failed", ex);
        throw ex;
    }
    catch (BeansException ex) {
```

Download at http://www.pin5i.com/

```
            this.logger.error("Context initialization failed", ex);
            throw ex;
        }
        if (this.logger.isInfoEnabled()) {
            long elapsedTime = System.currentTimeMillis() - startTime;
            this.logger.info("FrameworkServlet '" + getServletName() +
        "': initialization completed in " + elapsedTime + " ms");
        }
    }

    protected WebApplicationContext initWebApplicationContext() throws BeansException {
        WebApplicationContext wac = findWebApplicationContext();
        if (wac == null) {
            // No fixed context defined for this servlet - create a local one.
            /**
            * 这里调用 WebApplicationContextUtils 静态类来得到根上下文，这个根上下文是保存在
            * ServletContext 中的，把它作为当前 MVC 上下文的双亲上下文。
            */
            WebApplicationContext parent =
    WebApplicationContextUtils.getWebApplicationContext(getServletContext());
            wac = createWebApplicationContext(parent);
        }
        if (!this.refreshEventReceived) {
            /**
            *Apparently not a ConfigurableApplicationContext with refresh support:
            *triggering initial onRefresh manually here.
            */
            onRefresh(wac);
        }
        //把当前建立的上下文存到 ServletContext 中，注意使用的属性名是和当前 Servlet 名相关的。
        if (this.publishContext) {
            // Publish the context as a servlet context attribute.
            String attrName = getServletContextAttributeName();
            getServletContext().setAttribute(attrName, wac);
            if (this.logger.isDebugEnabled()) {
                this.logger.debug("Published WebApplicationContext of servlet '" +
                getServletName() +
                "' as ServletContext attribute with name [" + attrName + "]");
            }
        }
        return wac;
    }
```

在这里，这个 MVC 的上下文就被建立起来了，具体取得根上下文的过程是在 WebApplicationContextUtils 中实现的，如代码清单 4-9 所示。这个根上下文是在 ContextLoader 设置到 ServletContext 中去的，使用的属性是 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，同时对于这个 IoC 容器的 Bean 配置文件，ContextLoade 也进行了设置，默认的位置是在/WEB-INF/applicationContext.xml 文件中。由于这个根上下文是 DispatcherServlet 建立的上下文的双亲上下文，所以根上下文中管理的 Bean 也是可以被 DispatcherServlet 的上下文使用的。如果我们还有印象，会想起通过 getBean 向 IoC 容器获取 Bean 时，容器会先到它的双亲 IoC 容器中获取 getBean，这些我们在分析 IoC 容器的实现原理时重点讲解过。

Download at http://www.pin5i.com/

代码清单 4-9　取得根上下文

```
public static WebApplicationContext getWebApplicationContext(ServletContext sc) {
    /**
    * 使用了 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,这个属性代表的根上下文在
    * ContextLoaderListener 初始化的过程中被建立，并被设置到 ServletContext 中。
    */
    return getWebApplicationContext(sc, WebApplicationContext.ROOT_WEB_APPLICATION_
    CONTEXT_ATTRIBUTE);
}
```

DispatcherServlet 的上下文的建立过程与前面看到的建立根上下文的过程非常类似，如代码清单 4-10 所示。建立 DispatcherServlet 的上下文，需要把根上下文作为参数传递给它。然后使用反射技术来实例化上下文对象，并为它设置参数，根据默认的配置，这个上下文对象也是 XmlWebApplicationContext 对象，这个类型是在 DEFAULT_CONTEXT_CLASS 参数里设置好并让 BeanUtilis 使用的。在实例化结束以后，需要为这个上下文对象设置好一些基本的配置，这些配置包括它的双亲上下文、Bean 定义配置的文件位置等，在这些配置完成以后，最后通过调用 IoC 容器的 refresh 方法来完成 IoC 容器的最终初始化，这和在我们前面对 IoC 容器实现原理的分析中，看到的对 IoC 容器初始化的过程是一致的。

代码清单 4-10　FrameworkServlet 建立 WebApplicationContext

```
protected WebApplicationContext createWebApplicationContext(WebApplicationContext
    parent)
        throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Servlet with name '" + getServletName() +
                "' will try to create custom WebApplicationContext context of
                class '" +
                getContextClass().getName() + "'" + ", using parent context ["
                + parent + "]");
    }
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom
    (getContextClass())) {
        throw new ApplicationContextException(
                "Fatal initialization error in servlet with name '" +
                getServlet Name() +
                "': custom WebApplicationContext class [" +
                getContextClass().getName() +
                "] is not of type ConfigurableWebApplicationContext");
    }
    //实例化需要的具体上下文对象，并为这个上下文对象设置属性。
    /**
    *这里使用的是 DEFAULT_CONTEXT_CLASS, 这个 DEFAULT_CONTEXT_CLASS 被设置为 XmlWebApp-
    *licationContext.class, 所以在 DispatcherServlet 中使用的 IoC 容器是 XmlWebApplication-
    *Context。
    */
    ConfigurableWebApplicationContext wac =
            (ConfigurableWebApplicationContext) BeanUtils.instantiateClass
    (getContextClass());
    wac.setId(getServletContext().getServletContextName()+"."+getServletName());
    //这里配置双亲上下文，就是在 ContextLoader 中建立的根上下文。
    wac.setParent(parent);
    //设置 ServletContext 的引用和其他相关的配置信息。
wac.setServletContext(getServletContext());
```

Download at http://www.pin5i.com/

```
        wac.setServletConfig(getServletConfig());
        wac.setNamespace(getNamespace());
        wac.setConfigLocation(getContextConfigLocation());
        wac.addApplicationListener(new SourceFilteringListener(wac, this));
        postProcessWebApplicationContext(wac);
        //这里同样是通过 refresh 来调用容器的初始化过程。
        wac.refresh();
        return wac;
    }
```

这时候，我们看到 DispatcherServlet 中的 IoC 容器已经被建立起来了，这个 IoC 容器是根上下文的子容器。这样的设置，使得对具体的一个 Bean 定义查找过程来说，如果要查找一个由 DispatcherServlet 所在 IoC 容器来管理的 Bean，系统会先到根上下文中去查找。通过这一系列在 Web 容器中执行的动作，在这个上下文体系建立和初始化完毕的基础上，Spring MVC 就可以发挥其作用了，下面我们就来分析一下 Spring MVC 的具体实现。

4.5 Spring MVC 的实现

4.5.1 DispatcherSevlet 的 MVC 初始化

在前面分析 DispatchServlet 的初始化过程中可以看到，DispatchServlet 持有一个以自己的 Servlet 名称命名的 IoC 容器，这个 IoC 容器是一个 WebApplicationContext 对象，在这个 IoC 容器建立起来以后，意味着 DispatcherServlet 拥有自己的 Bean 定义空间，这为我们使用各个独立的 xml 文件来配置 MVC 中各个 Bean 创造了条件。由于在初始化结束以后，与 Web 容器相关的加载过程实际上已经完成了，Spring MVC 的具体实现和普通的 Spring 应用程序的实现并没有太大的差别，下面我们从 Spring MVC 的初始化过程入手。在 Spring MVC DispatcherServer 的初始化过程中，以对 HandlerMapping 的初始化调用作为触发点，去了解 Spring MVC 模块初始化的方法调用关系，如图 4-4 所示。可以看到，这个调用关系最初是由 HttpServletBean 的 init 方法触发，这个 HttpServletBean 是 HttpServlet 的子类，接着会在 HttpServletBean 的子类 FrameworkServlet 中对 IoC 容器完成初始化。在这个初始化方法中，会调用 DispatcherServlet 的 initStrategies 方法，在这个 initStrategies 方法中，启动整个 Spring MVC 框架的初始化。

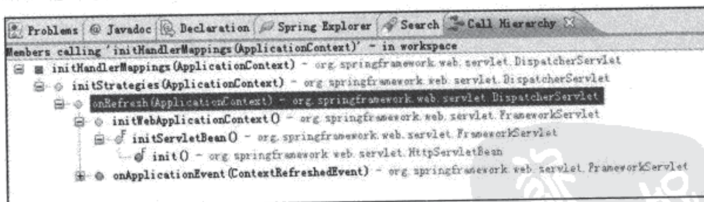


图 4-4 DispatcherSevlet 对 MVC 的初始化

在这个调用关系中我们可以看到对 MVC 的初始化是在 DispatcherServlet 的 initStrategies 完成的，它们包括对各种 MVC 框架里的实现元素，比如支持国际化的

Download at http://www.pin5i.com/

LocalResolver，支持 request 映射的 HandlerMappings，以及视图生成的 ViewResolver 的初始化，等等，如代码清单 4-11 所示。

代码清单 4-11 对 MVC 框架的初始化

```
protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
    initRequestToViewNameTranslator(context);
    initViewResolvers(context);
}
```

对于具体的初始化过程，根据上面的方法名称可以很容易理解。我们先看看 initHandlerMappings ()，这里的 mapping 关系起到的作用是，为 HTTP 请求找到相应的 Controller（控制器），从而使用这些 Controller 去完成设计好的数据处理工作。HandlerMappings 完成对 MVC 中 Controller 的定义和配置，只不过在 Web 这个特定的应用环境中，这些控制器是与具体的 HTTP 请求相对应的。我们到 DispatcherServlet 中去看看这个初始化过程的具体实现，如代码清单 4-12 所示。在对 HandlerMapping 初始化的过程中，完成的功能是把在 Bean 配置文件中配置好的 handlerMapping 从 IoC 容器中取得。

代码清单 4-12 对 HandlerMapping 的初始化

```
private void initHandlerMappings(ApplicationContext context) {
    this.handlerMappings = null;
    /**
    *这里导入所有的 HandlerMapping Bean，这些 Bean 可以在当前的 DispatcherServlet
    *的 IoC 容器中，也可能在其双亲上下文中。
    */
    //这个 detectAllHandlerMappings 默认值设为 true，即默认地从所有的 IoC 容器中取。
    if (this.detectAllHandlerMappings) {
        // Find all HandlerMappings in the ApplicationContext, including ancestor contexts.
        Map<String, HandlerMapping> matchingBeans =
                BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
                Handler Mapping.class, true, false);
        if (!matchingBeans.isEmpty()) {
            this.handlerMappings = new ArrayList<HandlerMapping>(matchingBeans.
            values());
            // We keep HandlerMappings in sorted order.
            OrderComparator.sort(this.handlerMappings);
        }
    }
    else { //可以根据名称从当前的 IoC 容器中通过 getBean 获取 handlerMapping。
        try {
            HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME,
            Handler Mapping.class);
            this.handlerMappings = Collections.singletonList(hm);
        }
        catch (NoSuchBeanDefinitionException ex) {
            // Ignore, we'll add a default HandlerMapping later.
        }
    }
```

Download at http://www.pin5i.com/

```
        /**
        * Ensure we have at least one HandlerMapping, by registering
        * a default HandlerMapping if no other mappings are found.
        */
        /**
        * 如果没有找到handerMappings，那么需要为Servlet设定默认的handlerMappings，
        * 这些默认的值可以设置在DispatcherServlet.properties中。
        */
        if (this.handlerMappings == null) {
           this.handlerMappings = getDefaultStrategies(context, HandlerMapping. class);
            if (logger.isDebugEnabled()) {
                 logger.debug("No HandlerMappings found in servlet '" +
        getServletName() + "': using default");
            }
        }
    }
```

经过以上的读取过程，handlerMappings 变量就已经获取在 BeanDefinition 中配置好的映射关系了。其他的初始化过程和 handlerMappings 比较类似，都是直接从 IoC 容器中读入配置，所以在这里的 MVC 的初始化过程是建立在 IoC 容器已经初始化完成的基础上的。至于这些上下文是如何获得的，可以参见前面对 IoC 容器在 Web 环境中加载的实现原理的分析。

4.5.2 HandlerMapping 的配置

上面分析了 DispatcherServlet 对 Spring MVC 框架的初始化过程，在此基础上，我们再进一步分析 HandlerMapping 的实现原理，看看这个 MVC 框架中比较关键的控制部分是如何实现的。

我们看到，在初始化完成的时候，在上下文环境中已定义的所有 HandlerMapping 都已经被加载了，这些加载的 HandlerMapping 被放在一个 List 中并被排序，存储着 HTTP 请求对应的映射数据，对于这个 List 中的每一个元素，它都对应着一个具体 HandlerMapping 的配置，而一般每一个 HandlerMapping 可以持有一系列从 URL 请求到 Controller 的映射，比如，我们举 SimpleUrlHandlerMaaping 这个 HandlerMapping 为例，在这个 SimpleUrlHandler-Mapping 中，就定义了一个 map 来持有这一系列的映射关系。通过这些在 HandlerMapping 中定义的映射关系，也就是说这些 URL 请求和控制器的对应关系，使得 Spring MVC 应用可以根据 HTTP 请求来确定一个对应的 Controller。具体来说，这些映射关系是通过接口类 HanderMapping 来封装的，在 HandlerMapping 接口中定义了一个 getHandler 方法，通过这个方法，可以获得与 HTTP 请求对应的 HandlerExecutionChain，其中封装了具体的 Controller 对象，如代码清单 4-13 所示。

代码清单 4-13 HandlerMapping 接口

```
public interface HandlerMapping {
  String PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE = HandlerMapping.class.getName()
     + ".pathWithinHandlerMapping";
  String URI_TEMPLATE_VARIABLES_ATTRIBUTE = HandlerMapping.class.getName() + ".
     uriTemplateVariables";
  /**
  *通过调用getHandler实际上返回的是一个HandlerExecutionChain，这是典型的Command模式的使
```

Download at http://www.pin5i.com/

```
 *用，这个 HandlerExecutionChain 不但持有 handler 本身，还包括了处理这个 HTTP 请求相关的拦截器。
 */
 HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
}
```

这个 HandlerExecutionChain 的实现看起来比较简洁，它持有一个 Interceptor 链和一个 handler 的对象，这个 handler 对象实际上就是 HTTP 请求对应的 Controller，在持有这个 handler 对象的同时，在 HandlerExecutionChain 中还设置了一个拦截器链，通过这个拦截器链里的拦截器，可以为 handler 对象提供功能的增强。而要完成这些工作，需要对拦截器链和 handler 都进行配置，这些配置都是在 HandlerExecutionChain 的初始化函数中完成的。为了维护这个拦截器链和 handler，它还提供了一系列与拦截器链维护相关的一些操作，比如可以为拦截器链增加拦截器的 addInterceptor 方法等。HandlerExecutionChain 的实现如代码清单 4-14 所示。

代码清单 4-14 HandlerExecutionChain 的实现

```
public class HandlerExecutionChain {
  private final Object handler;
  private HandlerInterceptor[] interceptors;
  private List<HandlerInterceptor> interceptorList;
  public HandlerExecutionChain(Object handler) {
     this(handler, null);
  }
  public HandlerExecutionChain(Object handler, HandlerInterceptor[] interceptors) {
     if (handler instanceof HandlerExecutionChain) {
          HandlerExecutionChain originalChain = (HandlerExecutionChain) handler;
          this.handler = originalChain.getHandler();
          this.interceptorList = new ArrayList<HandlerInterceptor>();
  CollectionUtils.mergeArrayIntoCollection(originalChain.getInterceptors(),
     this.interceptorList);
          CollectionUtils.mergeArrayIntoCollection(interceptors,
     this.interceptor List);
     }
     else {
          this.handler = handler;
          this.interceptors = interceptors;
     }
  }
  public Object getHandler() {
     return this.handler;
  }
  public void addInterceptor(HandlerInterceptor interceptor) {
     initInterceptorList();
     this.interceptorList.add(interceptor);
  }
  public void addInterceptors(HandlerInterceptor[] interceptors) {
     if (interceptors != null) {
          initInterceptorList();
          this.interceptorList.addAll(Arrays.asList(interceptors));
     }
  }
  private void initInterceptorList() {
     if (this.interceptorList == null) {
          this.interceptorList = new ArrayList<HandlerInterceptor>();
     }
     if (this.interceptors != null) {
          this.interceptorList.addAll(Arrays.asList(this.interceptors));
```

Download at http://www.pin5i.com/

```
            this.interceptors = null;
        }
    }
    public HandlerInterceptor[] getInterceptors() {
        if (this.interceptors == null && this.interceptorList != null) {
            this.interceptors = this.interceptorList.toArray(new HandlerInterceptor
        [this.interceptorList.size()]);
        }
        return this.interceptors;
    }
    public String toString() {
        return String.valueOf(this.handler);
    }
}
```

HandlerExecutionChain 中定义的 Handler 和 Interceptor 需要在定义 HandlerMapping 时配置好，例如对具体的 SimpleURLHandlerMapping，要做的就是根据 URL 映射的方式，注册 Handler 和 Interceptor，从而维护一个反映这种映射关系的 handlerMap。当需要匹配 HTTP 请求时，需要查询这个 handlerMap 里的信息来得到对应的 HandlerExecutionChain。但是，这些信息是什么时候配置好的呢？这里有一个注册过程，这个注册过程在容器对 Bean 进行依赖注入时发生，实际上是通过一个 bean 的 postProcessor 来完成的。如果想要了解这个调用过程，可以参考图 4-5。以 SimpleHanderMapping 为例，需要注意的是，这里用到了对容器的回调，只有 SimpleHanderMapping 是 Application-ContextAware 的子类才能启动这个注册过程。这个注册过程完成的是反映 URL 和 Controller 之间映射关系的 handlerMap 的建立，对于这个注册过程，具体的分析我们将在后面进行。

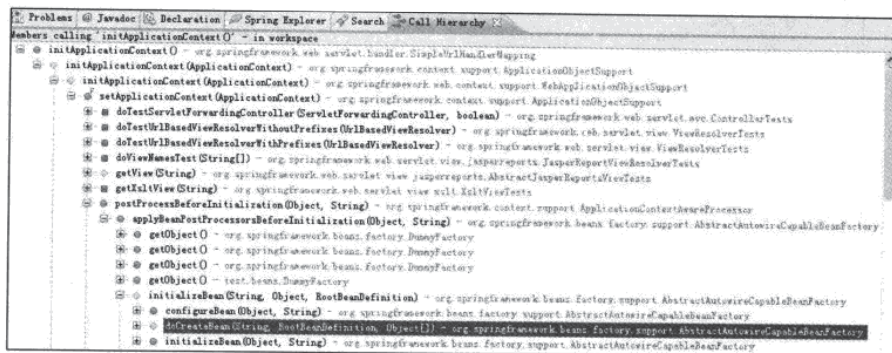


图 4-5　启动 HandlerMapping 的注册

我们可以在图 4-6 中看到 SimpleHanderMapping 的继承关系。

了解了这些调用关系的发生，进一步分析 SimpleUrlHandlerMapping 的注册过程是如何完成的，如代码清单 4-15 所示。

代码清单 4-15　SimpleUrlHandlerMapping 注册 Handler

```
public void initApplicationContext() throws BeansException {
    super.initApplicationContext();
    registerHandlers(this.urlMap);
```

Download at http://www.pin5i.com/

```
    }
    protected void registerHandlers(Map<String, Object> urlMap) throws BeansException
        {
        if (urlMap.isEmpty()) {
            logger.warn("Neither 'urlMap' nor 'mappings' set on
        SimpleUrlHandlerMap ping");
        }
        else {
            //这里对 Bean 的配置进行解析，然后调用基类的 registerHandler 完成注册。
            for (Map.Entry<String, Object> entry : urlMap.entrySet()) {
                String url = entry.getKey();
                Object handler = entry.getValue();
                // Prepend with slash if not already present.
                if (!url.startsWith("/")) {
                    url = "/" + url;
                }
                // Remove whitespace from handler bean name.
                if (handler instanceof String) {
                    handler = ((String) handler).trim();
                }
                registerHandler(url, handler);
            }
        }
    }
```

图 4-6 SimpleUrlMapping 的继承关系

这个 SimpleUrlHandlerMapping 的注册过程的完成，很大一部分需要它的基类来配合，这个基类就是 AbstractUrlHandlerMapping，我们看看在 AbstractUrlHandlerMapping 中的处理是怎样的，如代码清单 4-16 所示。在这个处理过程中，如果使用 Bean 的名称作为映射，那么直接从容器中获取这个 HTTP 映射对应的 Bean，然后还需要对不同的 URL 配置进行解析处理，比如在 HTTP 请求中把 URL 配置成“/”和通配符“/*”的情况，以及正常 URL 请求的处理，完成这个解析处理过程以后，会把 URL 和 handle 作为键值对放到一个 handlerMap 中去。

Download at http://www.pin5i.com/

代码清单 4-16 AbstractUrlHandlerMapping 对 handler 的注册

```
protected void registerHandler(String urlPath, Object handler) throws BeansException,
    IllegalStateException {
    Assert.notNull(urlPath, "url path must not be null");
    Assert.notNull(handler, "Handler object must not be null");
    Object resolvedHandler = handler;
    // Eagerly resolve handler if referencing singleton via name.
    //如果直接用 bean 名称进行映射，那就直接从容器中获取 handler。
    if (!this.lazyInitHandlers && handler instanceof String) {
        String handlerName = (String) handler;
        if (getApplicationContext().isSingleton(handlerName)) {
            resolvedHandler = getApplicationContext().getBean(handlerName);
        }
    }
    Object mappedHandler = this.handlerMap.get(urlPath);
    if (mappedHandler != null) {
        if (mappedHandler != resolvedHandler) {
            throw new IllegalStateException(
                    "Cannot map handler [" + handler + "] to url path ["+urlPath +
                    "]: There is already handler ["+resolvedHandler+"] mapped.");
        }
    }
    else { //处理 URL 是"/"的映射，把这个"/"映射的 controller 设置到 rootHandler 中。
        if (urlPath.equals("/")) {
            if (logger.isInfoEnabled()) {
                logger.info("Root mapping to handler [" + resolvedHandler + "]");
            }
            setRootHandler(resolvedHandler);
        }
        //处理 URL 是"/*"的映射，把这个"/"映射的 controller 设置到 defaultHandler 中。
        else if (urlPath.equals("/*")) {
            if (logger.isInfoEnabled()) {
               logger.info("Default mapping to handler[" + resolvedHandler + "]");
            }
            setDefaultHandler(resolvedHandler);
        }
        //处理正常的 URL 映射，设置 handlerMap 的 key 和 value，分别对应于 URL 和映射的 controller。
        else {
            this.handlerMap.put(urlPath, resolvedHandler);
            if (logger.isInfoEnabled()) {
                logger.info("Mapped url path [" + urlPath + "] onto handler ["
    + resolvedHandler + "]");
            }
        }
    }
}
```

我们这里看到的 handlerMap 是一个 HashMap，其中保存了 URL 请求和 Controller 的映射关系，这个 handlerMap 是在 AbstractUrlHandlerMapping 中定义的，如下所示。

```
private final Map<String, Object> handlerMap =
    new LinkedHashMap<String, Object>();
```

有了这个配置好的 URL 请求和 handler 映射数据的 handlerMap，就已经为 Spring MVC 响应 HTTP 请求准备好了基本的映射数据，根据这个 handlerMap 以及设置于其中的映射数据，可以方便地由 URL 请求得到它所对应的 handler，有了这些准备工作，Spring MVC 就开始敞开怀抱，静静地等待 HTTP 请求的到来了。

Download at http://www.pin5i.com/

4.5.3 使用 HandlerMapping 完成请求的映射处理

我们继续通过 SimpleUrlHandlerMapping 的实现来分析 HandlerMapping 的接口方法 getHandler，该方法会根据在初始化时得到的映射关系来生成 DispatcherServlet 需要的 HandlerExecutionChain，也就是在这个 getHandler 方法中，是实际使用 HandlerMapping 完成请求的映射处理的地方。回到前面的 HandlerExecutionChain 的执行过程，首先在 AbstractHandlerMapping 中看到启动 getHandler 的调用的实现，如代码清单 4-17 所示。

代码清单 4-17 AbstractHandlerMapping 的 getHandler 调用

```
public final HandlerExecutionChain getHandler(HttpServletRequest request) throws
    Exception {
    Object handler = getHandlerInternal(request);
    //使用默认的Handler，也就是"/"对用的handler。
    if (handler == null) {
        handler = getDefaultHandler();
    }
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?
    // 这里通过名称取出对应的Handler Bean。
    if (handler instanceof String) {
        String handlerName = (String) handler;
        handler = getApplicationContext().getBean(handlerName);
    }
    //这里把Handler封装到HandlerExecutionChain中去并加上拦截器。
    return getHandlerExecutionChain(handler, request);
}
protected HandlerExecutionChain getHandlerExecutionChain(Object handler,
    HttpServletRequest request) {
    if (handler instanceof HandlerExecutionChain) {
        HandlerExecutionChain chain = (HandlerExecutionChain) handler;
        chain.addInterceptors(getAdaptedInterceptors());
        return chain;
    }
    else {
        return new HandlerExecutionChain(handler, getAdaptedInterceptors());
    }
}
```

取得 handler 的具体过程在 getHandlerInternal 中实现，这个方法接受 HTTP 请求作为参数，它的实现在 AbstractHandlerMapping 的子类 AbstractUrlHandlerMapping 中，这个实现过程，包括从 HTTP 请求中得到 URL，并根据 URL 到 urlMapping 中去得到 handler，这个实现过程如代码清单 4-18 所示。

代码清单 4-18 AbstractUrlHandlerMapping 的 getHandlerInternal

```
protected Object getHandlerInternal(HttpServletRequest request) throws Exception {
    //这里从request中得到请求的URL路径。
    String lookupPath = this.urlPathHelper.getLookupPathForRequest(request);
    /**
    *这里使用得到的URL路径对Handler进行匹配，得到对应的Handler，如果没有对应的Hanlder，
```

Download at http://www.pin5i.com/

```
    *返回null，这样默认的Handler会被使用。
    */
    Object handler = lookupHandler(lookupPath, request);
    if (handler == null) {
        // We need to care for the default handler directly, since we need to
        // expose the PATH WITHIN HANDLER MAPPING ATTRIBUTE for it as well.
        Object rawHandler = null;
        if ("/".equals(lookupPath)) {
            rawHandler = getRootHandler();
        }
        if (rawHandler == null) {
            rawHandler = getDefaultHandler();
        }
        if (rawHandler != null) {
            validateHandler(rawHandler, request);
            handler = buildPathExposingHandler(rawHandler, lookupPath, null);
        }
    }
    if (handler != null && logger.isDebugEnabled()) {
        logger.debug("Mapping [" + lookupPath + "] to handler '" + handler + "'");
    }
    else if (handler == null && logger.isTraceEnabled()) {
        logger.trace("No handler mapping found for [" + lookupPath + "]");
    }
    return handler;
}
    /**
    *lookupHandler是根据URL路径，启动在handlerMap中对handler的检索，并最终返回
    *handler对象。
    */
protected Object lookupHandler(String urlPath, HttpServletRequest request) throws
    Exception {
    // Direct match?
    Object handler = this.handlerMap.get(urlPath);
    if (handler != null) {
        validateHandler(handler, request);
        return buildPathExposingHandler(handler, urlPath, null);
    }
    // Pattern match?
    String bestPathMatch = null;
    for (String registeredPath : this.handlerMap.keySet()) {
        if (getPathMatcher().match(registeredPath, urlPath) &&
                (bestPathMatch == null || bestPathMatch.length() <
    registered Path.length())) {
            bestPathMatch = registeredPath;
        }
    }
    if (bestPathMatch != null) {
        handler = this.handlerMap.get(bestPathMatch);
        validateHandler(handler, request);
        String pathWithinMapping =
    getPathMatcher().extractPathWithinPattern(best PathMatch, urlPath);
        Map<String, String> uriTemplateVariables =
                getPathMatcher().extractUriTemplateVariables(bestPathMatch, urlPath);
        return buildPathExposingHandler(handler, pathWithinMapping,
    uriTemplate Variables);
    }
    // No handler found...
    return null;
}
```

Download at http://www.pin5i.com/

经过这一系列对 HTTP 请求进行解析和匹配 handler 的过程，得到了与请求对应的 handler 处理器，在返回的 handler 中，已经完成了在 HandlerExecutionChain 的封装工作，为 handler 对 HTTP 请求的响应做好了准备。然而，在 MVC 中还有一个重要的问题是，请求是怎样实现分发，从而到达对应的 handler 那里去的呢？有了前面的分析，如果再对前端 HTTP 请求分发的过程有所了解，那么就基本上可以看到 HTTP 请求在 MVC 框架中处理过程的全貌了。

4.5.4 Spring MVC 对 HTTP 请求的分发处理

让我们重新回到 DispatcherServlet，它是 Spring MVC 框架中非常重要的一个类，它不但建立了自己持有的 IoC 容器，同时还肩负着请求分发处理的重任。在 MVC 框架初始化完成以后，对 HTTP 请求的处理是在 doService() 方法中完成的。DispatcherServlet 是 HttpServlet 的子类，与其他 HttpServlet 一样，可以通过 doService() 来响应 HTTP 的请求。然而，依照 Spring MVC 的使用，我们知道业务逻辑的调用入口是在 handler 的 handle 函数中实现的，这里就是连接这两个部分的地方。我们去看看 DispatcherServlet 的 doService 的实现，如代码清单 4-19 所示。

代码清单 4-19 DispatcherServlet 的 doService

```
protected void doService(HttpServletRequest request, HttpServletResponse response)
     throws Exception {
     if (logger.isDebugEnabled()) {
          String requestUri = new UrlPathHelper().getRequestUri(request);
          logger.debug("DispatcherServlet with name '" + getServletName() + "'
          processing " + request.getMethod() +
          " request for [" + requestUri + "]");
     }
     /**
     * Keep a snapshot of the request attributes in case of an include,
     * to be able to restore the original attributes after the include.
     */
     Map<String, Object> attributesSnapshot = null;
     if (WebUtils.isIncludeRequest(request)) {
          logger.debug("Taking snapshot of request attributes before include");
          attributesSnapshot = new HashMap<String, Object>();
          Enumeration attrNames = request.getAttributeNames();
          while (attrNames.hasMoreElements()) {
               String attrName = (String) attrNames.nextElement();
               if (this.cleanupAfterInclude || attrName.startsWith
     ("org.springframework.web.servlet")) {
                    attributesSnapshot.put(attrName, request.getAttribute(attrName));
               }
          }
     }
     // Make framework objects available to handlers and view objects.
     request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
     getWebApplicationContext());
     request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
     request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
     request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());
     try {
```

Download at http://www.pin5i.com/

```
            //这个 doDispatch 是分发请求的入口。
            doDispatch(request, response);
        }
        finally {
            // Restore the original attribute snapshot, in case of an include.
            if (attributesSnapshot != null) {
                restoreAttributesAfterInclude(request, attributesSnapshot);
            }
        }
    }
```

我们看到，对请求的处理实际上是由 doDispatch()来完成的，这个方法很长，但是过程简单明了，如代码清单 4-20 所示。这个 doDispatcher 方法是 DispatcherServlet 完成 Dispatcher 的主要方法，包括准备 ModelAndView，调用 getHandler 来响应 HTTP 请求，然后通过执行 Handler 的处理来得到返回的 ModelAndView 结果，最后把这个 ModelAndView 对象交给相应的视图对象去呈现。在这里，可以看到 MVC 模式核心的实现，同时，也是在这里，完成了模型、视图和控制器的紧密结合。

代码清单 4-20 DispatcherServlet 的 doDispatch

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response)
        throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    int interceptorIndex = -1;
    //为视图准备好一个 ModelAndView，这个 ModelAndView 持有 handler 处理请求的结果。
    try {
        ModelAndView mv = null;
        boolean errorView = false;
        try {
            processedRequest = checkMultipart(request);
            // Determine handler for the current request.
            // 根据请求得到对应的 handler,hander 的注册以及 getHandler 的实现在前面已经分析过。
            mappedHandler = getHandler(processedRequest, false);
            if (mappedHandler == null || mappedHandler.getHandler() == null) {
                noHandlerFound(processedRequest, response);
                return;
            }
            // Apply preHandle methods of registered interceptors.
            // 调用 hander 的拦截器，从 HandlerExecutionChain 中取出 Interceptor 进行前处理。
            HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();
            if (interceptors != null) {
                for (int i = 0; i < interceptors.length; i++) {
                    HandlerInterceptor interceptor = interceptors[i];
                    if (!interceptor.preHandle(processedRequest, response,
                    mappedHandler.getHandler())) {
                        triggerAfterCompletion(mappedHandler, interceptorIndex,
                    processedRequest, response, null);
                        return;
                    }
                    interceptorIndex = i;
                }
            }
            // Actually invoke the handler.
            /**
            * 这里是实际调用 handler 的地方，在执行 handler 之前，用 HandlerAdapter
            * 先检查一下 handler 的合法性是不是按 Spring 的要求编写的 handler。
```

Download at http://www.pin5i.com/

```
             * handler 处理的结果封装到ModelAndView对象，为视图提供展现数据。
             */
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
            /**
             * 这里通过调用HandleAdapter的handle方法，实际上触发对Controller的
             * handleRequest方法的调用。
             */
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
            // Do we need view name translation?
            if (mv != null && !mv.hasView()) {
                mv.setViewName(getDefaultViewName(request));
            }
            // Apply postHandle methods of registered interceptors.
            if (interceptors != null) {
                for (int i = interceptors.length - 1; i >= 0; i--) {
                    HandlerInterceptor interceptor = interceptors[i];
                    interceptor.postHandle(processedRequest, response,
                    mappedHandler.getHandler(), mv);
                }
            }
        }
        catch (ModelAndViewDefiningException ex) {
            logger.debug("ModelAndViewDefiningException encountered", ex);
            mv = ex.getModelAndView();
        }
        catch (Exception ex) {
            Object handler = (mappedHandler !=
                    null ? mappedHandler.getHandler():null);
            mv = processHandlerException(processedRequest, response, handler, ex);
            errorView = (mv != null);
        }
        // Did the handler return a view to render?
        // 这里使用视图对ModelAndView数据的展现。
        if (mv != null && !mv.wasCleared()) {
            render(mv, processedRequest, response);
            if (errorView) {
                WebUtils.clearErrorRequestAttributes(request);
            }
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("Null ModelAndView returned to DispatcherServlet
                with name '" + getServletName() +
                "': assuming HandlerAdapter completed request handling");
            }
        }
        // Trigger after-completion for successful outcome.
       triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
   response, null);
   }
   catch (Exception ex) {
        // Trigger after-completion for thrown exception.
       triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
   response, ex);
        throw ex;
   }
   catch (Error err) {
        ServletException ex =
```

Download at http://www.pin5i.com/

```
        new NestedServletException("Handler processing failed",err);
            // Trigger after-completion for thrown exception.
            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
            response, ex);
            throw ex;
        }
        finally {
            // Clean up any resources used by a multipart request.
            if (processedRequest != request) {
                cleanupMultipart(processedRequest);
            }
        }
    }
```

我们很清楚地看到和 MVC 框架紧密相关的代码，比如如何得到和 HTTP 请求相对应的 HandlerExecutionChain，执行 handler 和把模型数据展现到视图中去的过程。这个 handler 的请求处理过程是一个比较典型的 Command 模式的应用，我们下面看看 Handler 在 DispatcherServlet 中是如何取得的，这样就和前面对 handlerMapping 的分析接续起来了，下面是 getHandler 的代码，这个 getHandler 在 DispatcherServlet 中，如代码清单 4-21 所示。

代码清单 4-21　DispatcherServlet 取得 handler

```
protected HandlerExecutionChain getHandler(HttpServletRequest request, boolean
    cache) throws Exception {
    HandlerExecutionChain handler = (HandlerExecutionChain) request.getAttribute
    (HANDLER_EXECUTION_CHAIN_ATTRIBUTE);
    if (handler != null) {
        if (!cache) {
            request.removeAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE);
        }
        return handler;
    }
    //从 HandlerMapping 中取 handler 的调用，与前面对 handlerMapping 的分析在这里衔接上了。
    for (HandlerMapping hm : this.handlerMappings) {
        if (logger.isTraceEnabled()) {
            logger.trace(
                    "Testing handler map [" + hm + "] in DispatcherServlet with
                    name '" + getServletName() + "'");
        }
        handler = hm.getHandler(request);
        if (handler != null) {
            if (cache) {
                request.setAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE, handler);
            }
            return handler;
        }
    }
    return null;
}
```

在以上的代码实现中，可以看到在 DispatcherServlet 取 handler 的时候，首先会在 HttpRequest 中取 handler，相当于取一个缓存中的 handler，这个 handler 对应于 HTTP 的 HANDLER_EXECUTION_CHAIN_ATTRIBUTE 属性位置，这个属性位置被定义为 DispatcherServlet.class.getName() + ".HANDLER"；如果通过这样的方式得不到 handler，那么会通过在 DispatcherServlet 中持有的 HandlerMapping 来生成一个，我们

Download at http://www.pin5i.com/

看到在这个由 HandlerMapping 得到 Handler 的过程中，会遍历当前持有的所有 HandlerMapping，因为在 DispatcherServlet 中可能定义了不止一个 HandlerMapping，在这些一系列的 HandleMapping 中，只要找到了一个需要的 Handler 就会停止查找，而返回当前已经得到的 handler。在找到 handle 以后，通过 handler 返回的是一个 HandlerExecutionChain 对象，里面包含了最终的 Controller 和定义的一个拦截器链。对于这个过程，我们在前面对 SimpleUrlHandlerMapping 的实现中已经分析过了，在那里可以了解，getHandler 是怎样得到一个 HandlerExecutionChain 的。得到 HandlerExecutionChain 以后，DispatcherServlet 通过 HandlerAdapter 对这个 Handler 的合法性进行判断，然后返回适配结果，这个处理过程如代码清单 4-22 所示。

代码清单 4-22 DispatcherServlet 的 getHandlerAdapter

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    //对持有的所有 adapter 进行匹配。
    for (HandlerAdapter ha : this.handlerAdapters) {
        if (logger.isTraceEnabled()) {
            logger.trace("Testing handler adapter [" + ha + "]");
        }
        if (ha.supports(handler)) {
            return ha;
        }
    }
    throw new ServletException("No adapter for handler [" + handler +
    "]: Does your handler implement a supported interface like Controller?");
}
```

我们通过判断知道这个 handler 是不是 Controller 接口的实现，比如我们可以看看对于具体 HandlerAdapter 的实现来了解这个适配过程，以 SimpleControllerHandlerAdapter 的实现作为例子，看看这个判断是怎样起作用的，如代码清单 4-23 所示。这个判断通过 support 方法来实现，很简单，判断当前的 handler 是不是 Controller 对象，如果是 Controller 对象，那么返回 true，如果不是 Controller 对象，那么返回 false。

代码清单 4-23 SimpleControllerHandlerAdapter 的实现

```
public class SimpleControllerHandlerAdapter implements HandlerAdapter {
  //判断将要调用的 handler 是不是 Controller。
  public boolean supports(Object handler) {
    return (handler instanceof Controller);
  }
  public ModelAndView handle(HttpServletRequest request,
    HttpServletResponse response,Object handler)
        throws Exception {
    return ((Controller) handler).handleRequest(request, response);
  }
  public long getLastModified(HttpServletRequest request, Object handler) {
    if (handler instanceof LastModified) {
        return ((LastModified) handler).getLastModified(request);
    }
    return -1L;
  }
}
```

经过前面一系列的处理，得到了 handler 对象，接着就可以开始调用 handler 对象中的 HTTP 响应动作了。在 handler 里封装了应用业务逻辑，由这些逻辑对 HTTP 请求进行相应的处

Download at http://www.pin5i.com/

理，生成各种需要的数据，并把这些数据封装到 ModelAndView 对象中去，这个 ModelAndView 的数据封装是 Spring MVC 框架的要求。对 handler 来说，这些都是通过调用 handler 的 handleRequest 方法来触发完成的。在得到 ModelAndView 对象以后，这个 ModelAndView 对象会被交给 MVC 模式中的视图类，由视图类对 ModelAndVie 对象中的数据进行呈现。对于视图呈现的调用入口，我们在 DispatcherServlet 的 doDispatch 方法实现，它的调用入口是 render 方法，关于这个方法的具体实现，下面我们会对它进行详细的分析。

4.6　Spring MVC 视图的呈现

4.6.1　DispatcherServlet 视图呈现概述

在前面，我们分析了 Spring MVC 中的 M（Model）和 C（Controller）相关的实现。其中的 M 可以大致地对应成 ModelAndView 的生成，而 C 大致可以对应到 DispatcherServlet 和与用户业务逻辑有关的 handler 实现。在 Spring MVC 框架中，DipatcherServlet 起到了非常核心的作用，是整个 MVC 框架的调度枢纽。对于我们下面关心的视图呈现功能，它的调用入口同样在 DispatcherServlet 中的 doDispatch 方法中实现。具体来说，在 DispatcherServlet 中，对视图呈现的处理是在 render 方法调用中完成的，它的实现如代码清单 4-24 所示。为了完成视图的呈现工作，需要从 ModelAndView 对象中取得视图对象，然后调用视图对象的 render 方法，由这个视图对象来完成特定的视图呈现工作。同时，由于我们现在是在 Web 的环境中，所以视图对象的呈现往往需要与 HTTP 请求和响应相关的处理，对于这些对象，它们会作为参数传到视图对象的 render 方法中，供 render 方法使用。

代码清单 4-24　DispatcherServlet 的 render

```
protected void render(ModelAndView mv, HttpServletRequest request,
    HttpServlet Response response) throws Exception {
    // Determine locale for request and apply it to the response.
    // 从 request 中读取 locale 信息，并设置 response 的 locale 值。
    Locale locale = this.localeResolver.resolveLocale(request);
    response.setLocale(locale);
    View view = null;
     // 对根据 ModleAndView 中设置的视图名称进行解析，得到对应的视图对象。
    if (mv.isReference()) {
        // We need to resolve the view name.
        view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale,
    request);
        if (view == null) {
            throw new ServletException(
                    "Could not resolve view with name '" + mv.getViewName() +
                    "' in servlet with name '" + getServletName() + "'");
        }
    }// 有可能在 ModelAndView 中已经直接包含了 View 对象，那就直接使用。
    else {
        // No need to lookup: the ModelAndView object contains the actual View object.
        view = mv.getView();
        if (view == null) {
            throw new ServletException("ModelAndView [" + mv + "] neither contains a
```

Download at http://www.pin5i.com/

```
                view name nor a " +
                "View object in servlet with name '" + getServletName() + "'");
            }
        }
        // Delegate to the View object for rendering.
        if (logger.isDebugEnabled()) {
            logger.debug("Rendering view [" + view + "] in DispatcherServlet with
            name '" + getServletName() + "'");
        }
        //调用 view 实现对数据进行呈现，并通过 HttpResponse 把视图呈现给 http 客户端。
        view.render(mv.getModelInternal(), request, response);
    }
```

从上面可以看到，该视图的呈现过程是这样的，它需要在 ModelAndView 中寻找视图对象的逻辑名，如果已经在 ModelAndView 中设置了视图对象的名称，那就对这个名称进行解析，从而得到实际需要使用的视图对象。还有一种可能就是在 ModelAndView 中已经持有了最终完成视图呈现的视图对象，如果是这种情况，那么这个视图对象是可以直接使用的。不管如何，得到这个视图对象以后，都是通过调用这个视图对象的 render 方法，从而完成数据的显示过程，对应于不同的视图类型，往往对应着不同视图对象的实现。在了解这些特定的视图对象实现之前，我们看看 DispatcherServlet 是如何通过解析视图的逻辑名得到视图对象的，这个解析视图的过程如代码清单 4-25 所示。

代码清单 4-25　DispatcherServlet 解析视图

```
protected View resolveViewName(String viewName,
        Map<String, Object> model,
        Locale locale,
        HttpServletRequest request) throws Exception {
    //调用 ViewResolver 来进行解析.
    for (ViewResolver viewResolver : this.viewResolvers) {
        View view = viewResolver.resolveViewName(viewName, locale);
        if (view != null) {
            return view;
        }
    }
    return null;
}
```

ViewResolver 的解析过程可以参考常见的 BeanNameViewResolver 的 resolveViewName 实现。实现方法很简单，直接到上下文中通过名称的对应关系把作为 View 对象的 bean 取过来，如代码清单 4-26 所示。首先取得当前的 IoC 容器，然后判断在 IoC 容器中是否含有指定名称的视图 Bean，如果有，则通过 getBean 去取。

代码清单 4-26　从上下文中解析视图

```
public View resolveViewName(String viewName, Locale locale) throws BeansExcept-
    ion {
    ApplicationContext context = getApplicationContext();
    if (!context.containsBean(viewName)) {
        // Allow for ViewResolver chaining.
        return null;
    }
    return (View) context.getBean(viewName, View.class);
}
```

这样就得到了 View 对象，下面就看看 View 对象的拿手好戏。在分析 Viwe 的实现之前，先

Download at http://www.pin5i.com/

来看看 Spring MVC 中的 View 的继承体系。从这个继承体系上可以看到 Spring MVC 常用的视图都提供了支持，比如 JPS/JSTL 视图、FreeMaker 视图、Velocity 视图、Excel 和 PDF 视图等。对于这些丰富的视图支持，Spring MVC 通过与第三方框架的集成来实现，为开发 Web 应用的 UI 部分提供了许多便利。具体的继承关系如图 4-7 所示，在对这些 View 的相关类有了一个大致的了解后，后面将分析几种常见 View 的实现。

图 4-7　Spring MVC 提供的视图支持

4.6.2　JSP 视图的实现

使用 JSP 的页面作为 Web UI，是使用 Java 设计 Web 应用比较常见的选择之一，如果在 JSP 中使用 Jstl（JSP Standard Tag Library）来丰富 JSP 的功能，在 Spring MVC 中就需要使用 JstlView 来作为 View 对象，从而对数据进行视图呈现。JstlView 没有实现 render 方法，使用的 render 方法是在它的基类 AbstractView 中实现的，对于 AbstractView，它的 render 方法实现如代码清单 4-27 所示。

代码清单 4-27　AbstractView 的 render

```
public void render(Map<String, ?> model, HttpServletRequest request, HttpServlet-
    Response response) throws Exception {
    if (logger.isTraceEnabled()) {
        logger.trace("Rendering view with name '" + this.beanName + "'with model"
        + model + " and static attributes " + this.staticAttributes);
    }
    // 这里把所有的相关信息都收集到一个 Map 里。
    // Consolidate static and dynamic model attributes.
    Map<String, Object> mergedModel =
            new HashMap<String, Object>(this.staticAttributes.size() + (model !=
            null ? model.size() : 0));
    mergedModel.putAll(this.staticAttributes);
    if (model != null) {
        mergedModel.putAll(model);
```

Download at http://www.pin5i.com/

```
        }
        // Expose RequestContext?
        if (this.requestContextAttribute != null) {
            mergedModel.put(this.requestContextAttribute, createRequestContext(request,
            response, mergedModel));
        }
        prepareResponse(request, response);
        //展现模型数据到视图的调用方法。
        renderMergedOutputModel(mergedModel, request, response);
    }
```

这个基类的 render 方法实现并不复杂，它主要完成一些数据的准备工作，比如把所有的数据模型进行整合，放到一个 mergedModel 对象里面，mergeModel 是一个 HashMap，然后调用 renderMergedOutputModel()方法。renderMergedOutputModel 是一个模板方法，它的实现在 InternalResourceView 中完成。InternalResourceView 也是 JstlView 的基类，它实现 renderMergedOutputModel 方法，具体的实现过程如代码清单 4-28 所示。

代码清单 4-28 InternalResourceView 的 renderMergedOutputModel

```
    protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse
        response) throws Exception {
        // Determine which request handle to expose to the RequestDispatcher.
        HttpServletRequest requestToExpose = getRequestToExpose(request);
        // Expose the model object as request attributes.
        // 对数据进行处理，把模型对象存放到 ServletContext 中。
        exposeModelAsRequestAttributes(model, requestToExpose);
        // Expose helpers as request attributes, if any.
        exposeHelpers(requestToExpose);
        // Determine the path for the request dispatcher.
        // 获取 InternalResource 定义的内部资源路径。
        String dispatcherPath = prepareForRendering(requestToExpose, response);
        // Obtain a RequestDispatcher for the target resource (typically a JSP).
        //把请求转发到前面获取的内部资源路径中去。
        RequestDispatcher rd = requestToExpose.getRequestDispatcher(dispatcherPath);
        if (rd == null) {
            throw new ServletException(
                    "Could not get RequestDispatcher for [" + getUrl() + "]: check that
        this file exists within your WAR");
        }
        // If already included or response already committed, perform include, else
        forward.
        if (useInclude(requestToExpose, response)) {
            response.setContentType(getContentType());
            if (logger.isDebugEnabled()) {
                logger.debug("Including resource [" + getUrl() + "]
        in InternalResourceView '" + getBeanName() + "'");
            }
            rd.include(requestToExpose, response);
        }
        else {
          // Note: The forwarded resource is supposed to determine the content typeitself.
            /**
            *转发请求到内部定义好的资源上，比如 JSP 页面，JSP 页面的展现由 Web 容器负责，在这种情况下，
            *View 只是起到转发请求的作用。
            */
            exposeForwardRequestAttributes(requestToExpose);
```

Download at http://www.pin5i.com/

```
        if (logger.isDebugEnabled()) {
            logger.debug("Forwarding to resource [" + getUrl() + "]
    in InternalResourceView '" + getBeanName() + "'");
        }
        rd.forward(requestToExpose, response);
    }
}
```

在上面的代码中，可以看到对模型数据进行处理，这个处理是在 exposeModelAsRequestAttributes 方法实现的，这是一个设计在 AbstractView 中的方法，这个 exposeModelAsRequestAttributes 把 ModelAndView 中的模型数据和其他请求数据都放到 HttpServletRequest 的属性中去，这样一来，这些数据就可以通过 HttpServletRequest 的属性被得到和使用了，关于 exposeModelAsRequestAttributes 在 AbstractView 中的实现，如代码清单 4-29 所示。

代码清单 4-29 AbstractView 的 exposeModelAsRequestAttributes

```
protected void exposeModelAsRequestAttributes(Map<String, Object> model,
    HttpServletRequest request) throws Exception {
    for (Map.Entry<String, Object> entry : model.entrySet()) {
        String modelName = entry.getKey();
        Object modelValue = entry.getValue();
        if (modelValue != null) {
            request.setAttribute(modelName, modelValue);
            if (logger.isDebugEnabled()) {
                logger.debug("Added model object '" + modelName + "' of type ["
                + modelValue.getClass().getName() +
                "] to request in view with name '" + getBeanName() + "'");
            }
        }
        else {
            request.removeAttribute(modelName);
            if (logger.isDebugEnabled()) {
                logger.debug("Removed model object '" + modelName +
                        "' from request in view with name '" + getBeanName() + "'");
            }
        }
    }
}
```

回到数据处理部分的 exposeHelper();，这是一个模板方法，在 JstlView 中的实现如代码清单 4-30 所示，包含了对 Jstl 的相关处理。

代码清单 4-30 JstlView 的 exposeHelper

```
protected void exposeHelpers(HttpServletRequest request) throws Exception {
    if (this.messageSource != null) {
        JstlUtils.exposeLocalizationContext(request, this.messageSource);
    }
    else {
        JstlUtils.exposeLocalizationContext(new RequestContext(request,
    getServletContext()));
    }
}
```

在这些处理的基础上，实际的数据到页面的输出是由 InternalResourceView 来完成的，render 过程完成资源的重定向处理。需要做的是，在得到实际视图的 InternalResource 路径以后，把请求转发到资源中去。如何得到资源的路径呢？在 InternalResourceView 中可以看

Download at http://www.pin5i.com/

到调用，如代码清单 4-31 所示。在这里可以看到，从 request 中取得 URL 的路径，并对取得的路径进行了相应的处理。

代码清单 4-31　InternalResourceView 的 prepareForRendering

```
protected String prepareForRendering(HttpServletRequest request,
    HttpServletResponse response)
        throws Exception {
    //从 request 中去取 URL 的路径。
    String path = getUrl();
    if (this.preventDispatchLoop) {
        String uri = request.getRequestURI();
        if (path.startsWith("/") ? uri.equals(path) : uri.equals(StringUtils.
    applyRelativePath(uri, path))) {
            throw new ServletException("Circular view path [" + path + "]: would
                dispatch back " +
                    "to the current handler URL [" + uri + "] again. Check your
                        ViewResolver setup! " +
                    "(Hint: This may be the result of an unspecified view, due
                        to default view name generation.)");
        }
    }
    return path;
}
```

在得到 URL 路径之后，使用 RequestDispatcher 把请求转发到这个资源上，于是就完成了带 JSTL 的 JSP 页面的展现。

4.6.3　ExcelView 的实现

除了能够使用 JSP 这种常用的页面呈现外，Spring MVC 还整合了其他常用数据格式的页面展现，比如 Excel 数据。在呈现 Excel 视图时，Spring 并没有开发自己的 Excel 实现方案，而是使用已有的 Java Excel 解决方案来生成 Excel 文件，然后通过与 MVC 框架的整合，把生成的 Excel 文件输出到 HTTP 的 Response 中，在 HTTP 的客户端展现出来。在 Spring 3.0 版本中，分别提供了 POI 和 JExcelAPI 两个方案在 MVC 框架中的整合，它们的使用分别对应于两个 View 类：AbstractExcelView 和 AbstractJExcelView。在这里，我们选取 POI 的实现例子，对在 Spring MVC 中展示 Excel 视图的实现原理做一个简要的分析。

提示　POI 是 Apache 开源软件项目。项目的目的是通过使用 POI 的 Java API 直接操作各种文档的数据，文档包括 Microsoft 的 OLE 2 Compound 格式的文档和 Office OpenXML 格式的文档。通过 POI 的纯 Java 实现，可以对 Excel、Word、Powerpoint 的文档数据进行读写。感兴趣的读者可以去详细了解 POI 项目的具体情况，POI 的官方网站地址：http://poi.apache.org/index.html。

在 AbstractExcelView 中，Excel 视图的呈现是通过 POI 来完成的，如代码清单 4-32 所示。可以看到对 POI 的对象 HSSFWorkbook 的使用，它用来在 POI 中抽象 Excel 文件的对象。这个工作簿可以从模板 Excel 文件里取得，模板 Excel 文件可以通过 URL 来指定，也可以通过 HSSFWorkbook 对象来生成一个新的 Excel 文件。在得到代表 Excel 文件的 HSSFWorkbook 对象以后，就是通过这个对象对 Excel 文件中的数据进行处理的过程。这些文件的数据处理在 AbstractExcelView 中没有实现，是交给应用去完成的，这里为该实现定

Download at http://www.pin5i.com/

义了一个抽象方法 buildExcelDocument，应用需要实现该抽象方法，以完成自己的数据操作。

完成 Excel 的数据操作后，Excel 文件就已经准备好了，下面就是把它输出到 HTTP 客户端的过程。首先需要设置 HTTP 响应的输出类型，以便客户端进行识别。完成设置后，把 HSSFWorkbook 对象代表的数据输出到 HTTP 响应中，这样就完成了在服务器端的 Excel 视图呈现过程。

代码清单 4-32 AbstractExcelView 的 renderMergedOutputModel

```
protected final void renderMergedOutputModel(
        Map<String, Object> model, HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    //HSSFWorkBook 是 POI 的对象，用来抽象 Excel 的 WorkBook。
    HSSFWorkbook workbook;
    //从 URL 链接创建 POI 的 Excel 文档对象作为模板。
    if (this.url != null) {
        workbook = getTemplateSource(this.url, request);
    } //如果没有模板，直接创建新的 POI 的 Excel 对象。
    else {
        workbook = new HSSFWorkbook();
        logger.debug("Created Excel Workbook from scratch");
    }
    /**
    *这个方法是一个抽象方法，由子类来实现，在子类实现中，由应用来设计数据写入 Excel 文档的具体
    *操作，比如哪个 cell 写入哪些数据等。
    */
    buildExcelDocument(model, workbook, request, response);

    // Set the content type.
    // 设置 Response 的文档类型，使用的文档类型是："application/vnd.ms-excel"。
    response.setContentType(getContentType());
    // Should we set the content length here?
    // response.setContentLength(workbook.getBytes().length);
    // Flush byte array to servlet output stream.
    // 把 Excel 数据写入到 Servlet 的 Response 中，呈现到 HTTP 客户端。
    ServletOutputStream out = response.getOutputStream();
    workbook.write(out);
    out.flush();
}
//Creates the workbook from an existing XLS document.
//从已有的 xls 文件创建 POI 的 workbook 对象。
protected HSSFWorkbook getTemplateSource(String url, HttpServletRequest request)
    throws Exception {
    LocalizedResourceHelper helper =
    new LocalizedResourceHelper(getApplicationContext());
    Locale userLocale = RequestContextUtils.getLocale(request);
    Resource inputFile = helper.findLocalizedResource(url, EXTENSION, userLocale);

    // Create the Excel document from the source.
    if (logger.isDebugEnabled()) {
        logger.debug("Loading Excel workbook from " + inputFile);
    }
    POIFSFileSystem fs = new POIFSFileSystem(inputFile.getInputStream());
    return new HSSFWorkbook(fs);
}
/**
```

Download at http://www.pin5i.com/

```
     * Subclasses must implement this method to create an Excel HSSFWorkbook
     * document,given the model.
     */
    protected abstract void buildExcelDocument(
            Map<String, Object> model, HSSFWorkbook workbook, HttpServletRequest request,
        HttpServletResponse response)
            throws Exception;
    /**
     * Convenient method to obtain the cell in the given sheet, row and column.
     * <p>Creates the row and the cell if they still doesn't already exist.
     * Thus, the column can be passed as an int, the method making the needed downcasts.
     */
    //该方法可以帮助在Excel文档中定位具体的cell单元。
    protected HSSFCell getCell(HSSFSheet sheet, int row, int col) {
        HSSFRow sheetRow = sheet.getRow(row);
        if (sheetRow == null) {
            sheetRow = sheet.createRow(row);
        }
        HSSFCell cell = sheetRow.getCell((short) col);
        if (cell == null) {
            cell = sheetRow.createCell((short) col);
        }
        return cell;
    }
    //Convenient method to set a String as text content in a cell.
    //该方法可以帮助在Excel的cell单元中写入具体的值.
    protected void setText(HSSFCell cell, String text) {
        cell.setCellType(HSSFCell.CELL_TYPE_STRING);
        cell.setCellValue(new HSSFRichTextString(text));
    }
```

对需要输出 Excel 的视图文档的应用来说，只需要继承 AbstractExcelView，然后需要实现 buildExcelDocument 方法的具体操作，在实现 buildExcelDocument 方法中，可以使用在 Controller 中已经产生的模型数据，也可以从其他数据源读取数据，从而完成 Excel 文件的生成。通过 AbstractExcelView，用户的 Excel 数据可以很容易地与 MVC 框架结合起来，为 Web 用户提供了 Excel 文档的视图支持。如果需要增加对其他文档类型的支持，可以参考 Excel 视图的实现，比如可以完成对 Word 和 PPT 文档的视图实现等。

在 Spring 的源代码中，在提供了 Spring MVC 框架代码的同时，还提供了一些测试用例来测试 AbstractView 的功能实现，在这些测试用例中，就有关于如何生成 Excel 文档的应用实例，为我们创建自己的 Excel 文档视图提供了很好的参考。关于这些测试代码，我们可以到 ExcelViewTests 中去看看，以下的这个测试用例为用户使用 AbstractExcelView 提供了很好的参考，如代码清单 4-33 所示。这个测试用例展示了如何使用 POI 的 API 来生成 Excel 中的 Sheet，以及如何在相应的 cell 位置生成数据。作为测试用例，还可以看到对 Excel 写入数据的验证实现。在 Spring 源代码中，还有许多这样的测试用例实现，它们对我们学习 Spring API 的使用和测试用例设计来说，都是很好的参考资料。

代码清单 4-33 测试用例 testJExcel

```
    public void testJExcel() throws Exception {
        AbstractJExcelView excelView = new AbstractJExcelView() {
            /**
            *buildExcelDocument生成具体的Excel文档，比如在哪个sheet的哪个cell
```

Download at http://www.pin5i.com/

```
            *单元写入什么样的数据。
            */
            protected void buildExcelDocument(Map model,
                    WritableWorkbook wb,
                    HttpServletRequest request,
                    HttpServletResponse response)
                    throws Exception {
                WritableSheet sheet = wb.createSheet("Test Sheet", 0);

                // test all possible permutation of row or column not existing
                sheet.addCell(new Label(2, 4, "Test Value"));
                sheet.addCell(new Label(2, 3, "Test Value"));
                sheet.addCell(new Label(3, 4, "Test Value"));
                sheet.addCell(new Label(2, 4, "Test Value"));
            }
        };
        //因为没有整合到DispatcherServlet和IoC容器中，所以需要手动启动render方法触发视图的呈现。
        excelView.render(new HashMap(), request, response);
        //这里是测试验证部分，从Response中读入数据，从而验证前面数据的写入是否正确。
        Workbook wb = Workbook.getWorkbook(new ByteArrayInputStream
        (response.getContentAsByteArray()));
        assertEquals("Test Sheet", wb.getSheet(0).getName());
        Sheet sheet = wb.getSheet("Test Sheet");
        Cell cell = sheet.getCell(2, 4);
        assertEquals("Test Value", cell.getContents());
    }
```

4.6.4 PDF 视图的实现

MVC 作为一个框架，为整合各种视图提供了非常大的便利。前面已经讲过 Excel 视图的产生，接下来讲一讲 PDF 视图的实现。同样地，PDF 视图的实现在 AbstractPdfView 中完成，Spring 使用的也是第三方开源的 PDF 解决方案 iText。

提示　iText 是免费的基于 Java 语言的 PDF 文件生成类库，可以方便 Java 运行环境（比如 Servlet）集成，为使用者提供对 PDF 文件的内容操作功能。项目的官方网站 http://www.lowagie.com/iText/。

---

iText 与 Spring MVC 的集成部分在 AbstractPdfView 的代码中，与前面实现 Excel 视图呈现的实现相类似，如代码清单 4-34 所示。在这里可以看到使用 iText 创建 PDF 文件并输出到 HTTP 响应的过程。在 iText 中，使用 Document 对象来抽象对 PDF 文件的使用。

代码清单 4-34　AbstractPdfView

```
//在构造函数中为HTTP的Response设置文档类型: "application/pdf"。
public AbstractPdfView() {
    setContentType("application/pdf");
}
protected final void renderMergedOutputModel(
        Map<String, Object> model, HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    // IE workaround: write into byte array first.
    ByteArrayOutputStream baos = createTemporaryOutputStream();
    // Apply preferences and build metadata.
    // 创建iText的与PDF文件操作相关的对象。
```

Download at http://www.pin5i.com/

```
        Document document = newDocument();
        PdfWriter writer = newWriter(document, baos);
        prepareWriter(model, writer, request);
        buildPdfMetadata(model, document, request);
        // Build PDF document.
        // 创建 PDF 文件的内容，具体的创建过程交给子类的 buildPdfDocument 方法去完成。
        document.open();
        buildPdfDocument(model, document, writer, request, response);
        document.close();
        // Flush to HTTP response.
        // 输出到 HTTP Response，将 PDF 视图呈现到客户端。
        writeToResponse(response, baos);
    }
```

如何使用好 AbstractPdfView？同样可以到 AbstractPdfView 的测试用例中看个明白，这些测试用例在 PdfViewTests 中，我们可以参考这个对 PDF 的测试代码来看看具体的 AbstractPdfView 是如何使用的，如代码清单 4-35 所示。在这个测试用例里，通过 AbstractPdfView 的实现来为 PDF 文件添加一段文字："this should be in the PDF"，输出到 HTTP 响应中以后，再从 HTTP 响应中读入以前写入的数据，完成一个环回的数据验证。

代码清单 4-35 测试用例 PdfViewTests

```
    public class PdfViewTests extends TestCase {
    public void testPdf() throws Exception {
        final String text = "this should be in the PDF";
        MockHttpServletRequest request = new MockHttpServletRequest();
        MockHttpServletResponse response = new MockHttpServletResponse();
        //使用 AbstractPdfView 为 PDF 添加字符串。
        AbstractPdfView pdfView = new AbstractPdfView() {
            protected void buildPdfDocument(Map model, Document document, PdfWriter
        writer,HttpServletRequest request, HttpServletResponse response) throws
        Exception {
                document.add(new Paragraph(text));
            }
        };
        //把 PDF 视图通过 HTTP Response 呈现到客户端，并从 HTTP 的 Response 中回读数据。
        pdfView.render(new HashMap(), request, response);
        byte[] pdfContent = response.getContentAsByteArray();
        assertEquals("correct response content type", "application/pdf", response.
        getContentType());
        assertEquals("correct response content length", pdfContent.length,
        response.getContentLength());
        // Rebuild iText document for comparison.
        // 生成一个 PDF 文件，内容与使用 PDF 视图写入到 HTTP 的 Response 中的一样。
        Document document = new Document(PageSize.A4);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PdfWriter writer = PdfWriter.getInstance(document, baos);
        writer.setViewerPreferences(PdfWriter.AllowPrinting | PdfWriter.PageLayout
        SinglePage);
        document.open();
        document.add(new Paragraph(text));
        document.close();
        byte[] baosContent = baos.toByteArray();
        assertEquals("correct size", pdfContent.length, baosContent.length);
        //对回读数据和新生成的 PDF 数据进行字节比较。
        int diffCount = 0;
        for (int i = 0; i < pdfContent.length; i++) {
            if (pdfContent[i] != baosContent[i]) {
                diffCount++;
```

Download at http://www.pin5i.com/

```
            }
        }
        assertTrue("difference only in encryption", diffCount < 70);
    }
```

AbstractPdfView 为服务器向客户端呈现 PDF 文件视图，提供了很大的帮助。有了这个 AbstractPdfView 的帮助，应用只需要使用 iText 的相关类库的 PDF 文档操作功能，专注于用户数据在 PDF 中的生成，也就是在 buildPdfDocument 实现 PDF 数据的生成，就可以方便地完成 PDF 视图在 HTTP 环境中地呈现。关于 PDF 文件视图的配置和它在 HTTP 中的呈现实现，以及它与 MVC 环境的集成，都不需要花费太多的精力，这些工作都由 Spring MVC 完成了。

4.7 小结

本章对整个 Spring MVC 框架的运行过程和实现进行了简要的分析，从在 Web 环境中建立 Spring IoC 容器的实现原理入手，先分析了 Spring IoC 容器在 Web 容器中的配置和初始化完成过程。从整个体系上看，这些 Web 应用可以看成是一个 Spring 应用，与一般的 Spring 应用并无太大的差别，都需要配置 IoC 容器和各种 Bean 定义等。在理解了 Spring 的 IoC 容器实现原理的基础上，这些内容并不难理解。只是在这里，因为 Web 容器存在一定的特殊性，所以在配置上，需要使用 Spring 作为平台的 Web 应用有一些与 Web 环境相对应的特殊处理，比如对 Servlet 和 ServletContext 的使用等。

对 Spring 作为应用平台的 Web 应用开发而言，Spring 为它们提供了 Spring MVC 框架，作为一个像 Structs 这样的 Web 框架的替代。当然，作为应用平台，Spring 并不会强制应用对 Web 框架的选择，但对 Web 应用开发而言，选择直接使用 Spring MVC 可以给应用开发带来许多便利。因为 Spring MVC，很好地提供了与 Web 环境中的 IoC 容器的集成，同时，和其他 Web 应用一样，使用 Spring MVC 应用只需要专注于处理逻辑和视图呈现的开发（当然这些开发需要符合 Spring MVC 的开发习惯）。在视图呈现部分，Spring MVC 同时也集成了许多现有的 Web UI 实现，比如像 Excel、PDF 这些文档视图的生成，因为集成其他方案实在可以说是 Spring 的拿手好戏，从这种一致性的开发模式上看，它在很大程度上降低了 Web 应用开发的门槛。

在理解了整个应用背景之后，Spring MVC 的整体实现就比较好理解了。通过逐步分析 Spring MVC 的实现原理，我们对 MVC 模式的具体实现有了深刻的认识。具体看来，整个 Spring MVC 的运作是以 DispatcherServlet 为中心来进行控制的。总的来说，Spring MVC 的实现大致由以下几个步骤完成：

1）需要建立 Controller 控制器和 HTTP 请求之间的映射关系，也就是说在 Spring MVC 实现中，是如何根据请求得到对应的 Controller 的？通过分析可以看到，在 Spring MVC 中，这个工作是由在 HandlerMapping 中封装的 HandlerExecutionChain 对象来完成的，而对于 Controller 控制器和 HTTP 请求的映射关系的配置，是在 Bean 定义中描述并在 IoC 容器初始化时，通过初始化 HandlerMapping 来完成的，这些定义的映射关系会被载入到一个 handlerMap 中被使用。

2）在初始化过程中，Controller 对象和 HTTP 请求之间的映射关系建立好以后，为 Spring MVC 接收 HTTP 请求并完成响应处理做好了准备。在 MVC 框架接收到 HTTP 请求的时候，

Download at http://www.pin5i.com/

DispatcherServlet 会根据具体的 URL 请求信息，在 HandlerMapping 中进行查询，从而得到对应的 HandlerExecutionChain，在这个 HandlerExecutionChain 中封装了配置的 Controller，有了请求对应的 Controller，它会完成请求的响应动作，生成需要的 ModelAndView 对象，在这个对象中，就像它的名字所表示的一样，可以从这个对象得到 Model 模型数据和视图对象。

3）得到了 ModelAndView 以后，DispatcherServlet 把获得的模型数据交给特定的视图对象，从而完成这些数据的视图呈现工作，这个视图呈现由视图对象的 render 方法来完成。对应于不同的视图对象，render 方法会完成不同的视图呈现处理，为用户提供丰富的 Web UI 表现。

Spring MVC 在视图呈现部分的实现充分发挥了 Spring 一贯以来的兼容并蓄风格，为 Web 应用对各种视图的实现提供了丰富选择。本章选取了 JSP 视图、Excel 视图和 PDF 视图，以它们做为例子，对这些视图实现的基本原理进行了简要的分析。在对 Excel 视图和 PDF 视图呈现的实现原理的分析中，Spring MVC 使用了许多第三方的解决方案，比如像 POI、iText 这些类库。这些第三方解决方案，很巧妙地与 Spring MVC 框架整合在一起，为应用提供视图生成的帮助，在减轻了用户生成视图的负担的同时，也非常清晰地体现了 Spring 作为应用平台和服务集成环境带给应用开发的价值。

Download at http://www.pin5i.com/

第 5 章

数据库操作组件的实现

胜日寻芳泗水滨，无边光景一时新。
等闲识得东风面，万紫千红总是春。
——【宋】朱熹《春日》

5.1 Spring JDBC 和 Spring ORM 概述

在 Java 开发环境中，通过 JDBC 技术，Java 语言的客户端可以访问数据库的数据，比如 CURD（创建、更新、查询、删除）等对数据库数据的基本操作。尽管在实际的应用中，对应于不同的数据库产品，还需要有相对应的数据库驱动作为支持，但由于有了 JDBC 和 SQL，对数据库应用而言，其程序的可移植性是大大增加了。

JDBC 已经能够满足大部分用户操作数据库数据的需求，作为应用开发平台的 Spring ，也对这些数据库操作需求提供了很好的支持。在 Spring JDBC 模块中提供了许多使用 JDBC 的模板和驱动模块，为 Spring 应用操作关系数据库提供很了大的便利。本章会对 Spring JDBC 的实现进行分析，分析过程需要 Spring JDBC 的源代码的支持，我们需要将 `org.springframework.jdbc` 包导入到本地的 Eclipse 环境中。

作为一种面向对象语言，Java 为面向对象原则（封装、继承、多态）的实现提供了语言及运行环境支持。然而，由于这些面向对象的原则是从软件工程的基础上发展而来的，与从数学理论中发展起来的关系数据库技术在基础上就存在着很大的不同。这样，在利用 Java 语言进行开发时，在与关系数据库打交道的过程中，就出现了一些不匹配的地方，为了解决这些不匹配，出现了 ORM 技术。随着技术的发展，现在已经有不少成熟的 Java ORM 产品供开发者选择。

同样地，Spring 在它的 ORM 包里提供了对许多 ORM 产品的支持。对于开源软件来说，Hibernate 和 iBatis 是应用较为广泛的两个 ORM 产品，所以本书选择了这两个产品作为例子对 Spring ORM 的实现进行分析。在对 Hibernate 和 iBastis 的驱动支持的分析过程中，读者可以体会到 Spring 为简化用户使用 ORM 产品所做的一些努力。在分析过程中，同样需要源代码作为支持，读者需要在 Eclipse 环境中导入 `org.springframework.orm` 包。

5.2 Spring JDBC 模板类的实现

5.2.1 JdbcTeamplate 的基本使用

如果大家使用过 Spring JDBC，那肯定不会对 `JdbcTemplate` 感到陌生。在 Spring JDBC 中，

Download at http://www.pin5i.com/

JdbcTemplate 是操作数据库的类，它提供了许多便利的数据库操作方法，比如查询、更新等。而且，在 Spring 中，有许多类似 JdbcTemplate 的模板类，使用方法也都非常类似，比如我们在 ORM 包里还会看到 HibernateTemplate 等。

简单看来，这些在 Spring 中设计和实现好的模板类都是通过回调函数的使用来完成其功能的，对满足应用的数据库操作需求而言，应用程序只需要在回调接口中实现自己需要的定制行为，比如使用客户设计好的 SQL 语句等，就能够完成对数据库的数据操作。不过，Spring JDBC 在这层回调函数的基础上进行了再次封装，为用户提供了许多数据库操作的现成方法，在一定程度上更便于用户使用。JdbcTemplate 是一个很重要的类，它的类继承关系和主要的实现方法如图 5-1 所示，我们可以看到一系列的 execute 方法实现，除了这些方法实现，在 JdbcTemplate 中还提供了 query、update 等的实现，感兴趣的读者可以直接到 JdbcTemplate 的源代码实现中去了解。

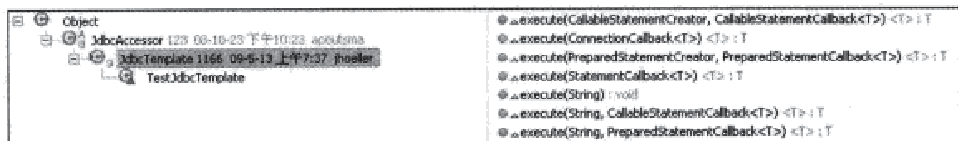


图 5-1　JdbcTemplate 的类继承关系和主要实现方法

在使用 JdbcTemplate 时，一些有特点的回调函数是我们比较熟悉的，比如代码清单 5-1，它大致展示了 JdbcTemplate 的最基本使用方式。

代码清单 5-1　使用 JdbcTemplate

```
JdbcTemplate temp = new JdbcTemplate(datasource);
class ExecuteStatementCallback implements StatementCallback<Object>, SqlProvider {
        public Object doInStatement(Statement stmt) throws SQLException
    {
            //用户定义的数据库操作代码或 Spring 为我们封装的数据库操作实现。
            stmt.execute(sql);
            return null;
        }
        public String getSql() {
            return sql;
        }
}
temp.sexecute(new ExecuteStatementCallback());
```

在模板的回调方法 doInStatement 中嵌入的是用户对数据库进行操作的代码，可以由 Spring 来完成（前面提到过，Spring JDBC 在这个最基本的模板上还提供了进一步的封装），或者由客户应用直接完成，然后通过 JdbcTemplate 的 execute 方法就可以完成相应的数据库操作。

5.2.2　JdbcTemplate 的 execute 实现

下面来进一步分析 JdbcTemplate 中的代码是如何完成使命的，以 JdbcTemplate.execute() 为例。这个方法是在 JdbcTemplate 中被其他方法调用的基本方法之一，应用程序往往使用这个方法来执行基本的 SQL 语句，如代码清单 5-2 所示。在 execute 的实现中看到了对数据库

Download at http://www.pin5i.com/

进行操作的基本过程，比如需要取得数据库 Connection，根据应用对数据库操作的需要创建数据库的 Statement，对数据库操作进行回调，处理数据库异常，最后把数据库 Connection 关闭，等等。这里展示了使用 JDBC 完成数据库操作的完整过程，只是在 Spring 中，对这些较为通用的 JDBC 使用，通过 JdbcTemplate 进行了一个封装而已。

代码清单 5-2 JdbcTemplate 的 execute 方法

```
//execute 方法执行的是输入的 SQL 语句。
public void execute(final String sql) throws DataAccessException {
    if (logger.isDebugEnabled()) {
        logger.debug("Executing SQL statement [" + sql + "]");
    }
    class ExecuteStatementCallback implements StatementCallback<Object>,SqlProvider {
        public Object doInStatement(Statement stmt) throws SQLException {
            stmt.execute(sql);
            return null;
        }
        public String getSql() {
            return sql;
        }
    }
    execute(new ExecuteStatementCallback());
}
//这是使用 java.sql.Statement 处理静态 SQL 语句的方法。
public <T> T execute(StatementCallback<T> action) throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");
    //这里取得数据库的 Connection，它已经在 Spring 的事务管理之下。
    Connection con = DataSourceUtils.getConnection(getDataSource());
    Statement stmt = null;
    try {
        Connection conToUse = con;
        if (this.nativeJdbcExtractor != null &&
this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {
            conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
        }
        //创建 Statement。
        stmt = conToUse.createStatement();
        applyStatementSettings(stmt);
        Statement stmtToUse = stmt;
        if (this.nativeJdbcExtractor != null) {
            stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
        }
        //这里调用回调函数。
        T result = action.doInStatement(stmtToUse);
        handleWarnings(stmt);
        return result;
    }
    catch (SQLException ex) {
        /**
        * Release Connection early, to avoid potential connection pool deadlock
        * in the case when the exception translator hasn't been initialized yet.
        */
        /**
        *如果捕捉到数据库异常，把数据库 Connection 释放，同时抛出一个经过 Spring 转换过的
        *Spring 数据库异常。
        */
        //Spring 做了一项有意义的工作，就是把这些数据库异常统一到自己的异常体系里了。
```

Download at http://www.pin5i.com/

```
            JdbcUtils.closeStatement(stmt);
            stmt = null;
            DataSourceUtils.releaseConnection(con, getDataSource());
            con = null;
            throw getExceptionTranslator().translate("StatementCallback",
        getSql(action), ex);
        }
        finally {
            JdbcUtils.closeStatement(stmt);
            //释放数据库 connection。
            DataSourceUtils.releaseConnection(con, getDataSource());
        }
    }
```

5.2.3 JdbcTemplate 的 query 实现

对于 JdbcTemplate 中给出的 query、update 等常用方法的实现，很多都是依赖于前面提到的 execute 方法。下面将详细分析 JdbcTemplate 中 query 方法的实现原理，如代码清单 5-3 所示。在 query 中，是通过使用 PreparedStatementCallback 的回调方法 doInPreparedStatement 来实现的，在回调函数中，可以看到 PreparedStatement 的执行以及查询结果的返回等处理。

代码清单 5-3 JdbcTemplate 的 query 方法

```
    public <T> T query(
            PreparedStatementCreator psc, final PreparedStatementSetter pss, final
        ResultSetExtractor<T> rse)
            throws DataAccessException {
        Assert.notNull(rse, "ResultSetExtractor must not be null");
        logger.debug("Executing prepared SQL query");
        //调用 excute 并设置 excute 的回调函数。
        return execute(psc, new PreparedStatementCallback<T>() {
            public T doInPreparedStatement(PreparedStatement ps) throws SQLException {
                //准备查询结果集。
                ResultSet rs = null;
                try {
                        if (pss != null) {
                        pss.setValues(ps);
                    }
                    //这里执行的 SQL 查询。
                    rs = ps.executeQuery();
                    ResultSet rsToUse = rs;
                    if (nativeJdbcExtractor != null) {
                        rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
                    }//返回需要的记录集合。
                    return rse.extractData(rsToUse);
                }
                finally {
                    //最后关闭查询的记录集，数据库连接在 execute() 中释放。
                    JdbcUtils.closeResultSet(rs);
                    if (pss instanceof ParameterDisposer) {
                        ((ParameterDisposer) pss).cleanupParameters();
                    }
                }
            }
```

Download at http://www.pin5i.com/

```
        });
    }
```

5.2.4 使用数据库 Connection

在以上的这些对数据库的操作中，使用了辅助类 DataSourceUtils，Spring 通过这个辅助类来对数据的 Connection 进行管理。比如通过它来完成打开和关闭 Connection 等操作。DataSourceUtils 对这些数据库 Connection 管理的实现，如代码清单 5-4 所示。在数据库应用中，数据库 Connection 的使用往往与事务管理有很紧密的联系，这里也可以看到与事务处理相关的操作，比如 Connection 和当前线程的绑定等。

代码清单 5-4 DataSourceUtils 对数据库连接的管理

```
//这是取得连接的调用，实现是通过调用 doGetConnection 完成的，执行了异常的转换操作。
public static Connection getConnection(DataSource dataSource) throws CannotGet
    JdbcConnectionException {
    try {
        return doGetConnection(dataSource);
    }
    catch (SQLException ex) {
        throw new CannotGetJdbcConnectionException
    ("Could not get JDBC Connection",ex);
    }
}
public static Connection doGetConnection(DataSource dataSource) throws SQLExcep-
    tion {
    Assert.notNull(dataSource, "No DataSource specified");
    /**
    *把对数据库的 Connection 放到事务管理中进行管理，这里使用 TransactionSynchronizationManager
    *中定义的 ThreadLocal 变量来和线程绑定数据库连接。如果在 TransactionSynchronizationManager
    *中已经有与当前线程绑定数据库连接，那就直接取出来使用。
    */
    ConnectionHolder conHolder = (ConnectionHolder) TransactionSynchronization
    Manager.getResource(dataSource);
    if (conHolder != null && (conHolder.hasConnection() || conHolder.isSynchron-
    izedWithTransaction())) {
        conHolder.requested();
        if (!conHolder.hasConnection()) {
            logger.debug("Fetching resumed JDBC Connection from DataSource");
            conHolder.setConnection(dataSource.getConnection());
        }
        return conHolder.getConnection();
    }
    // Else we either got no holder or an empty thread-bound holder here.
    /**
    * 这里得到需要的数据库 Connection，在 Bean 配置文件中定义好的，同时最后把新打开的数据库
    * Connection 通过 TransactionSynchronizationManager 和当前线程绑定起来。
    */
    logger.debug("Fetching JDBC Connection from DataSource");
    Connection con = dataSource.getConnection();
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
```

Download at http://www.pin5i.com/

```
            logger.debug("Registering transaction synchronization for JDBC Connection");
            // Use same Connection for further JDBC actions within the transaction.
            // Thread-bound object will get removed by synchronization at transaction completion.
            ConnectionHolder holderToUse = conHolder;
            if (holderToUse == null) {
                holderToUse = new ConnectionHolder(con);
            }
            else {
                holderToUse.setConnection(con);
            }
            holderToUse.requested();
            TransactionSynchronizationManager.registerSynchronization(
                    new ConnectionSynchronization(holderToUse, dataSource));
            holderToUse.setSynchronizedWithTransaction(true);
            if (holderToUse != conHolder) {
                TransactionSynchronizationManager.bindResource(dataSource, holderToUse);
            }
        }
        return con;
    }
```

实际的 DataSource 对象是如何得到的呢？很显然，需要在上下文中进行配置，这个 DataSource 对象作为 JdbcTemplate 基类 JdbcAccessor 的属性，可以通过 IoC 容器的依赖注入设置到 JdbcTemplate 中，或者在使用 JdbcTemplate 的时候，应用可以直接在初始化时提供 DataSource 对象注入给 JdbcTemplate。

对于 DataSource 的缓冲池实现， 用户可以通过定义 Apache Jakarta Commons DBCP 或 C3P0 提供的 DataSource 来完成，然后只要在 IoC 容器中配置好给 JdbcTemplate 就可以使用了。同时，有了 JdbcTemplate 封装的方法，对一些简单的 JDBC 操作的使用是非常方便的，JdbcTemplate 里面提供了许多现成的查询方法，比如 queryForInt、queryForObject 等，已经能够很好地满足一些简单的 JDBC 查询处理了。如果需要使用数据的更新功能，和查询一样，JdbcTemplate 也提供了许多不同参数类型的 update 方法供用户使用，这些 update 方法的实现原理与 query 的实现原理基本相同，感兴趣的读者可以参考 query 的实现进一步进行分析，这里就不重复了。

5.3 Spring JDBC 中 RDBMS 操作对象的实现

从上面我们看到，JdbcTemplate 提供了许多简单查询和更新的功能。但是，如果需要更高层次的抽象，以及更面向对象的方法来访问数据库，Spring 为我们提供了 org.springframework.jdbc.object 包，里面包含了 SqlQuery、SqlMappingQuery、SqlUpdate 和 StoredProcedure 等类，这些类都是 Spring JDBC 应用程序可以使用的。但要注意，在使用这些类时需要为它们配置好 JdbcTemplate 作为其基本的操作实现，因为在它们的功能实现中，对数据库操作的那部分实现基本上还是依赖于 JdbcTemplate 完成的。

下面将对其中一些 RDBMS 操作对象的实现进行探讨。在进入具体的讨论之前，先来看看

Download at http://www.pin5i.com/

这些 RDBMS 对象的基本继承关系，如图 5-2 所示。在这个继承关系图中，我们可以看到 RdbmsOperation 的一系列子类，如 StoreProcedure、SqlQuery、MappingSql Query、SqlUpdate 等。这些子类构成了 RDBMS 体系，为通过 JDBC 使用来完成数据库操作提供了更强大的功能。

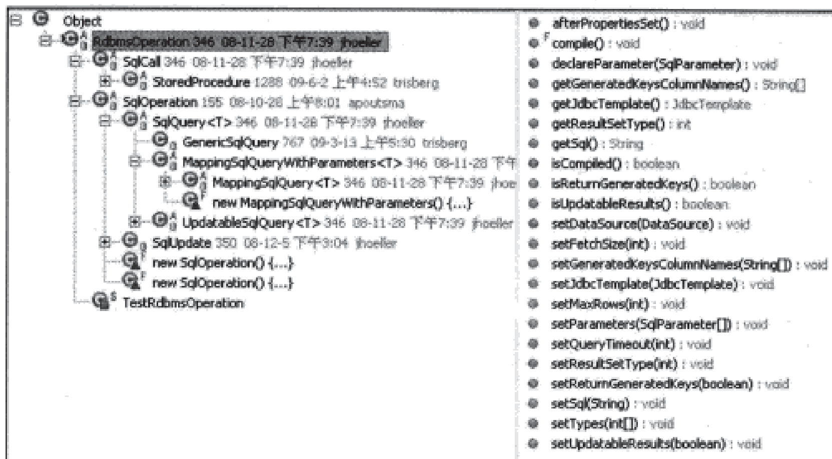


图 5-2 RDBMS 的继承关系

5.3.1 SqlQuery 的实现

Spring 除了提供对 JDBC 的基本操作的支持之外，还为应用在更高层面上使用关系数据库提供了许多支持，这些支持都建立在 Spring JDBC 实现的基础上。这样，用户可省去许多重复的手工代码，充分地发挥了框架的作用。比如说，我们可以使用 MappingSqlQuery 来将数据库表的数据记录直接映射到一个对象集合，这是一个很有用的特性，类似一个简单的 O/R 映射实现。

在了解这个特性的实现之前，可以先回顾一下它的基本用法。看看代码清单 5-5 中的示例就很容易理解了，它演示了 MappingSqlQuery 的基本使用。在使用 MappingSqlQuery 完成这个数据转换功能的时候，需要用户扩展一个 MappingSqlQuery 实现，并在用户扩展类的初始化函数中，对 SQL 查询语句和查询参数进行设置，然后调用 compile 最终完成这些设置。

此外，需要应用设置具体的数据转换代码的实现，比如需要把数据库记录转换成什么样的 Java 数据对象，如何设置数据对象的数据域和数据库记录数据域的对应关系，在这里很像一个简单的 ORM 实现，只是对于一般的 ORM 实现，往往把这个映射关系设置到一个配置文件中去。而在 Spring JDBC 中，是通过使用 Java 代码的编码方式来完成的。这部分数据转换代码，会在对数据库的查询结束后执行，从而完成数据查询记录到 Java 数据对象的转换。

Download at http://www.pin5i.com/

代码清单 5-5 使用 MappingSqlQuery

```
private class CustomerMappingQuery extends MappingSqlQuery {
    public CustomerMappingQuery(DataSource ds) {
        //把 DataSource 和查询的 SQL 语句设置到 MappingSqlQuery 的属性中。
          super(ds, "SELECT id, name FROM customer WHERE id = ?");
        //declareParameter 和 compile 实现了什么，我们将在下面分析。
          super.declareParameter(new SqlParameter("id", Types.INTEGER));
          compile();
    }
    //把查询得到的记录集合中的记录转换为对象的具体方法。
    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Customer cust = new Customer();
        cust.setId((Integer) rs.getObject("id"));
        cust.setName(rs.getString("name"));
        return cust;
    }
}
```

有了以上的数据转换操作设置以后，下面看看这个功能在应用中是如何被使用的，如代码清单 5-6 所示。它的使用很简单，需要创建一个扩展了 MappingSqlQuery 的子类的 CustomerMappingQuery 对象，在 CustomerMappingQuery 的设计中，参数声明和执行的 SQL 语句是已经设计好的。接着，只要根据参数声明的设计，为参数指定配置，然后执行 execute 方法，就可以得到一系列的 Java 数据对象，这些 Java 数据对象都是根据数据库查询结果和数据转换来生成的，有了它们数据就可以被 Java 应用直接使用了。

代码清单 5-6 在应用中使用 MappingSqlQuery

```
public Customer getCustomer(Integer id) {
    CustomerMappingQuery custQry = new CustomerMappingQuery(dataSource);
    Object[] parms = new Object[1];
    parms[0] = id;
    List customers = custQry.execute(parms);
    if (customers.size() > 0) {
        return (Customer) customers.get(0);
    }
    else {
        return null;
    }
}
```

可以看到，整个对 MappingSqlQuery 使用的过程是非常简洁的。在设计好数据的映射代码之后，查询得到的记录已经按照前面的设计转换为对象 List 了，一条查询记录对应于一个数据对象，可以把数据库的数据记录直接映射成 Java 对象在程序中使用，同时又可避免使用第三方 ORM 工具的配置，对于简单的数据映射场合是非常方便的。在 mapRow 方法的实现中提供的数据转换规则，和我们使用 Hibernate 时的 hbm 文件起到的作用是非常类似的。

下面从 declareParameter 入手来分析一下这个功能是如何实现的，其具体的实现在 MappingSqlQuery 的基类 RdbmsOperation 的源代码中可以看到，如代码清单 5-7 所示。RdbmsOperation 实现的参数声明和完成的工作并不复杂，只是把需要声明的参数加入到一个声明参数列表中。为了防止参数的重复加入，需要对 isCompile 变量进行判断，这个 isCompile 变量会在 compile 调用时被设置，标识 compile 过程已经执行过了。

Download at http://www.pin5i.com/

代码清单 5-7 RdbmsOperation 的 declareParameter

```
public void declareParameter(SqlParameter param) throws InvalidDataAccess
    ApiUsageException {
    //声明参数只能在 compile 之前，否则声明是无效的，并会抛出异常。
    if (isCompiled()) {
        throw new InvalidDataAccessApiUsageException("Cannot add parameters once
    the query is compiled");
    }
    /**
    *声明就是把参数加入到 declaredParameters 中去，这个 declaredParameters
    *是定义好的一个 LinkedList<SqlParameter>()属性，供 compile 使用。
    */
    this.declaredParameters.add(param);
}
```

Compile 的过程同样也在 RdbmsOperation 里完成，如代码清单 5-8 所示。这个 compile 完成的工作不多，首先是通过对 isCompile 变量的判断，然后调用 compileInternal 来完成 compile 的具体操作，接着把 isCompile 设置为 true。

代码清单 5-8 RdbmsOperation 的 compile

```
public final void compile() throws InvalidDataAccessApiUsageException {
    /**
    *如果没有做过 compile，执行以下的代码，否则什么都不做，如果已经做过 compile，
    *在 declareParameters 时就会抛出异常，所以这里没有异常抛出。
    */
    if (!isCompiled()) {
        if (getSql() == null) {
            throw new InvalidDataAccessApiUsageException("Property 'sql' is required");
        }
        try {
            this.jdbcTemplate.afterPropertiesSet();
        }
        catch (IllegalArgumentException ex) {
            throw new InvalidDataAccessApiUsageException(ex.getMessage());
        }
        //调用 compileInternal 完成具体的 compile 过程，并设置 compiled 标志位。
        compileInternal();
        this.compiled = true;
        if (logger.isDebugEnabled()) {
            logger.debug("RdbmsOperation with SQL [" + getSql() + "] compiled");
        }
    }
}
```

compileInternal 方法在 SqlOperation 里完成，如代码清单 5-9 所示。在 compileInternal 中，生成了一个 PrepareStatementCreatorFactory 作为 Statement 的工厂，这个工厂负责生成声明参数的 Statement，对于它的使用我们会在查询执行时看到。

代码清单 5-9 CompileInternal 的实现

```
protected final void compileInternal() {
    /**
    *这里是对参数的 compile 过程，所有的参数都在 getDeclaredParameters 里面，
    *生成了一个 PreparedStatementCreatorFactory.
    */
   this.preparedStatementFactory = new PreparedStatementCreatorFactory(getSql(),
```

Download at http://www.pin5i.com/

```
        getDeclaredParameters());
        this.preparedStatementFactory.setResultSetType(getResultSetType());
    this.preparedStatementFactory.setUpdatableResults(isUpdatableResults());
    this.preparedStatementFactory.setReturnGeneratedKeys(isReturnGeneratedKeys());
        if (getGeneratedKeysColumnNames() != null) {
    this.preparedStatementFactory.setGeneratedKeysColumnNames(getGeneratedKeysColumnNames());
        }
    this.preparedStatementFactory.setNativeJdbcExtractor(getJdbcTemplate().getNative
        JdbcExtractor());
        onCompileInternal();
    }
```

在完成了 compile 之后，对 MappingSqlQuery 的准备工作就基本完成了。在执行查询时，实际上执行的是 SqlQuery 的 executeByNamedParam 方法，这个方法需要完成的工作包括配置 SQL 语句，配置数据记录到数据对象的转换的 RowMapper，然后使用 JdbcTemplate 来完成数据的查询，并启动数据记录到 Java 数据对象的转换，如代码清单 5-10 所示。

代码清单 5-10 SqlQuery 的 executeByNamedParam

```
    public List<T> executeByNamedParam(Map<String, ?> paramMap, Map context) throws
        DataAccessException {
        validateNamedParameters(paramMap);
        //得到需要执行的 SQL 语句。
        ParsedSql parsedSql = getParsedSql();
        MapSqlParameterSource paramSource = new MapSqlParameterSource(paramMap);
        String sqlToUse = NamedParameterUtils.substituteNamedParameters(parsedSql,
        paramSource);
        //配置好 SQL 语句需要的 Parameters 及 rowMapper，这个 rowMapper 完成数据记录到对象的转换。
       Object[] params = NamedParameterUtils.buildValueArray(parsedSql, paramSource,
        getDeclaredParameters());
        RowMapper<T> rowMapper = newRowMapper(params, context);
        /**
        *我们又看到了 JdbcTemplate,这里使用 JdbcTemplate 来完成对数据库的查询操作,
        *所以我们说 JdbcTemplate 是非常基本的操作类。
        */
            return getJdbcTemplate().query(newPreparedStatementCreator(sqlToUse, params),
         rowMapper);
    }
```

在这里，通过这些封装和 Spring 提供的模板代码，只需要定义 SQL 语句和 SqlParameter 就能够满足我们的要求了。同时，通过这些特性的使用，也大大增强了应用代码的模块化和可维护性。

通过分析，我们发现最后的数据库操作是由 JdbcTemplate 来完成的。在这里可以看到对 JdbcTemplate 的灵活使用。从另一个方面看，这些已有的 Spring JDBC 实现中，对 JdbcTemplate 的扩展，对于对直接使用 JdbcTemplate 完成复杂数据库操作的应用而言，也是很好的参考。通过使用 Spring JDBC 中提供的 SqlQuery 基本特性，应用免去了手工处理 ResultSet 数据，并将其中每一条数据记录进行逐个迭代，手工转化为 Java 数据对象的繁琐过程，有数据库开发经验的读者，一定对这个过程有比较深刻的印象。尽管这个过程对大家来说可能并不复杂，但是通过使用 Spring JDBC 来完成这个基本特性，至少有一个好处就是为这种转换的实现提供了一个统一的模式。

从软件工程角度来说，这种统一模式的使用很好地体现了平台带给应用开发的一个好处，因为应用平台已经为应用开发人员之间的沟通创建了一个极为有效的交流媒介，就像面向对象语言的那些设计模式一样。这也对提高软件开发的效率有着极大的帮助，就像在《人月神话》中提到的那

Download at http://www.pin5i.com/

样，在软件产品开发中，其中一个重要的问题就是如何维护概念的完整性。这个“概念完整性”包括了软件需求本身的完整性，以及通过产品实现得到的对软件需求完整性的反馈及验证，从而体现出来的一种需求到实现的完整性，甚至还包括在产品实现过程中的完整性维护等。这一系列完整性的需求体现了软件工程自身内在的特点，从而也带来了各种不同层次的沟通挑战，这些沟通挑战具体体现在：如何从应用领域范围内得到有效的软件产品需求的沟通挑战；在完成从软件产品需求到产品实现之间有效设计范围内的沟通挑战；还有在产品实现过程中的沟通挑战等。在这里，采用应用平台至少能够得到的一个好处，就像我们在前面看到的，在功能需求实现中，在对一个具体特性的使用过程中体现出来的那种一致性。这种一致性，至少能够在实现层面上，对沟通挑战的应对起到一个正面的作用。这种沟通平台的建立和对沟通挑战的成功应对，对于一个软件产品最终能够走向成功的贡献是巨大的，这种正面的模式建立得越多，效果发挥得越充分，那么，软件产品成功的可能性也就越大。

5.3.2 SqlUpdate 的实现

与前面分析 SqlQuery 类似，SqlUpdate 也是一个常用的 RDBMS 类，但它主要提供对数据的 update 功能，它的使用和实现原理与前面提到的 SqlQuery 非常类似。同样地，我们先看看对这个类的基本使用，如代码清单 5-11 所示。首先也是需要对数据源、SQL 查询语句和参数声明进行设置，然后调用 compile 来完成设置过程。使用起来也是非常地简洁，对应用来说，只需要提供具体的参数对象的值，并调用 update 方法就可以完成整个数据的更新过程，至于数据库 Connection 的管理，事务处理场景的处理这些在数据库操作中都会涉及的基本过程，都由作为应用平台的 Spring 来帮我们打点了。

代码清单 5-11 对 SqlUpdate 的使用

```
public class UpdateCreditRating extends SqlUpdate {
    //注入数据源，提供SQL语句以及声明参数，然后完成Compile.
    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }
    //提供具体的参数对象，并调用update完成数据库数据的更新。
    public int run(int id, int rating) {
        Object[] params =
            new Object[] {
                new Integer(rating),
                new Integer(id)};
        return update(params);
    }
}
```

具体的 update 过程是由 SqlUpdate 里的 updateByNamedParam 方法完成的，它的具体实现与 SqlQuery 的实现一样，也是使用 JdbcTemplate 来完成 update 工作的，如代码清单 5-12 所示。

代码清单 5-12 SqlUpdate 的 updateByNamedParam

```
public int updateByNamedParam(Map<String, ?> paramMap) throws DataAccessException {
    validateNamedParameters(paramMap);
```

Download at http://www.pin5i.com/

```
        // 设置 SQL 和配置 SQL 的参数。
        ParsedSql parsedSql = getParsedSql();
        MapSqlParameterSource paramSource = new MapSqlParameterSource(paramMap);
        String sqlToUse = NamedParameterUtils.substituteNamedParameters(parsedSql,
        paramSource);
        Object[] params = NamedParameterUtils.buildValueArray(parsedSql,paramSource,
        getDeclaredParameters());
        //调用 JdbcTemplate 进行 update。
        int rowsAffected = getJdbcTemplate().update(newPreparedStatementCreator(sqlToUse,
        params));
        checkRowsAffected(rowsAffected);
        return rowsAffected;
    }
```

5.3.3 SqlFunction

在了解了 SqlQuery 和 SqlUpdate 的使用和实现原理之后，在 Spring JDBC 的 RDBMS 体系中，有一个 SqlFunction 类也是非常值得注意的，它是 MappingSqlQuery 的子类。参考 Spring 使用手册的说明，我们可以大致了解 SqlFunction 的基本功能：在这个 SqlFunction RDBMS 操作类中，它封装了一个 SQL“函数”包装器（wrapper），使用这个包装器可以查询并返回一个单行结果集，对于这个结果集，SqlFunction 默认地返回一个 int 型的对象，如果需要返回其他的对象类型，可以仿照类似 JdbcTemplate 中的 queryForXxx 的实现，由应用进行对它扩展以实现返回对象的调整。使用 SqlFunction 的优势在于，用户不必手动的创建 JdbcTemplate，对于这些设置，SqlFunction 已经替用户完成了。在使用 SqlFunction 的时候，我们看到由于 SqlFunction 是一个具体类，通常在使用时并不需要设计它的子类。它的使用方法很简单，基本过程是这样的，首先需要创建该类的实例，然后声明 SQL 语句以及参数就可以调用相关的 run 方法完成 SQL 语句的执行，这种执行可以重复进行。下面可以看到的是一个使用 SqlFunction 从而返回指定数据库中特定的数据表的记录行数的简单例子，如代码清单 5-13 所示。在代码中可以看到，为了使用 SqlFunction，首先需要创建一个 SqlFunction 对象，创建时需要为它指定数据源和执行的 SQL 语句。这里指定的 SQL 语句很简单，只需要查询表的记录数目。创建完成以后执行 compile，然后就可以调用 SqlFunction 的 run 方法完成指定的 SQL 语句的执行，得到查询数据记录行数的返回结果。

代码清单 5-13 SqlFunction 使用实例

```
public int countRows() {
    SqlFunction sf = new SqlFunction(dataSource, "select count(*) from mytable");
    sf.compile();
    return sf.run();
}
```

了解 SqlFunction 的基本使用以后，接下来我们到 SqlFunction 的 run 方法中去看看其具体的实现过程，如代码清单 5-14 所示，这是 SqlFunction 执行 SQL 语句的实现部分。

代码清单 5-14 SqlFunction 的 run

```
public int run(Object... parameters) {
     Object obj = super.findObject(parameters);
     if (!(obj instanceof Number)) {
          throw new TypeMismatchDataAccessException("Couldn't convert result object ["
     + obj + "] to int");
     }
```

Download at http://www.pin5i.com/

```
        return ((Number) obj).intValue();
    }
```

在 run 方法中，调用的是 super.findOjbect 方法，这个方法在 SQLFunction 的基类 SqlQuery 中实现，如代码清单 5-15 所示。对 SqlFunction 而言，它定义了 SingleColumnRowMapper 来完成数据记录到数据对象的转换，而不需要应用去实现这个转换的具体设计，这个数据转换器 SingleColumnRowMapper 是 Spring 定义好的，起到的作用与我们在使用 MappingSqlQuery 对象时的 mapRow 方法一样，完成数据库数据查询记录到 Java 数据对象的转换实现。

代码清单 5-15 SqlQuery 的 findObject

```
public T findObject(Object[] params, Map context) throws DataAccessException {
    //调用 SqlQuery 的 execute 方法来执行数据库操作。
    List<T> results = execute(params, context);
    return DataAccessUtils.singleResult(results);
}
public T findObject(Object... params) throws DataAccessException {
    return findObject(params, null);
}
public List<T> execute(Object[] params, Map context) throws DataAccessException {
    validateParameters(params);
    //取得 rowMapper，它是在 SqlFunction 里定义的 SingleColumnRowMapper<T>。
    RowMapper<T> rowMapper = newRowMapper(params, context);
    //调用 JdbcTemplate 的 query 来完成，并使用 rowMapper 来完成数据到对象的转换。
    return getJdbcTemplate().query(newPreparedStatementCreator(params), rowMapper);
}
```

在这里，最后仍然是通过 JdbcTemplate 来完成数据库操作的。如果读者对这些具体的 JdbcTemplate 的数据库操作实现感兴趣，可以参考前面对 JdbcTemplate 的分析。

5.4 Spring 驱动 Hibernate 的实现

ORM 产品出现之后，简化了许多复杂的面向对象数据到数据库持久化的工作。相对于对于 ORM 产品的直接使用，对于 Spring 应用开发而言，可以通过 Spring 平台提供的 ORM 产品方案，更方便地使用各种持久化工具，比如 Hibernate 和 iBatis。作为一个成熟和知名的开源软件项目，Hibernate 在很大程度上已经成为 Java ORM 产品的一个杰出代表。本节的内容对于具备单独使用 Hibernate 基础的读者，是很好理解的，因为在 Spring 中，它不提供具体的 ORM 实现，而只是为应用提供对 ORM 产品的集成环境和使用平台。为了让平台用户更好地使用 Hibernate 这个 ORM 产品，Spring 为 Hibernate 用户提供了更为便利的 API 使用封装，这些使用封装通过 HibernateTemplate 这样的类来完成。有了这些封装，Spring 为应用更好地使用像 Hibernate 这样的第三方产品提供了便利。这种使用关系与操作系统中驱动程序的作用很类似，所以这里就借用了这个概念。在本书中，我们把这种平台对第三方产品的支持称为对组件的驱动支持。对于 Hibernate 这样的第三方产品，我们把它类比为可以在操作系统中看到的设备组件，希望这样能为读者提供一个形象的理解，从而对 Spring 的架构有更好的认识。

Download at http://www.pin5i.com/

5.4.1 配置 Hibernate 的 SessionFactory

下面对 Spring 封装 Hibernate 的实现进行分析，我们还是从对 Hibernate 的配置实现入手。从配置中可以看到，在 Spring 中使用 Hibernate 一般会使用 LocalSessionFactoryBean 作为一个基本的配置 Bean，它是一个 FactoryBean。对于 FactoryBean 的实现原理，在第 2 章中分析过，顾名思义，这个 LocalSessionFactoryBean 是用来对 Hibernate 的 Session 进行管理的。我们可以在 LocalSessionFactoryBean 的基类 AbstractSessionFactoryBean 中看到有关的实现，如代码清单 5-16 所示。对于这样一个 FactoryBean，我们会直接到它的 getObject 方法中去了解它是如何产生具体对象的。在 getObject 方法中，可以看到，没有太多对目标对象的生成和修饰，它完成一件非常单纯的任务，就是把已经在 IoC 容器中配置好的 SessionFactory 返回。

代码清单 5-16 AbstractSessionFactoryBean

```
// Return the singleton SessionFactory.
//getObject ( ) 实际上是 FactoryBean 的生产方法.
public SessionFactory getObject() {
    return this.sessionFactory;
}
```

在 Hibernate 中用到的这个 SessionFactory 是一个单件，作为一个工厂对象，使用它来生成 Session。对于 Session 的使用，使用过 Hibernate 的读者都应该有印象，这个 Session 是作为 Hibernate 完成对象持久化的上下文出现的，同时它也是 Hibernate 封装对象持久化操作的一个主要类。在这里，我们看到了怎样得到 Session，回过头来看看，对于一开始我们提到的生成 Session 的单件 SessionFactory，它又是在如何产生的呢？对于它的创建，我们看到是在 AbstractSessionFactoryBean 中完成的，如代码清单 5-17 所示。可以看到，SessionFactory 的创建是在容器依赖注入完成以后，由 IoC 容器的回调方法 afterPropertiesSet 来完成的。对于这个 IoC 容器回调方法的启动，我们可以看到 LocalSessionFactoryBean 实现了 InitializingBean 接口，而这个 InitializingBean 接口的 afterPropertiesSet 方法会被 IoC 容器回调，是 IoC 容器对 Bean 进行生命周期管理的一部分。

代码清单 5-17 AbstractSessionFactoryBean 创建 SessionFactoryBean

```
public void afterPropertiesSet() throws Exception {
    //buildSessionFactory 是通过配置信息得到 SessionFactory 的地方。
    SessionFactory rawSf = buildSessionFactory();
    this.sessionFactory = wrapSessionFactoryIfNecessary(rawSf);
    afterSessionFactoryCreation();
}
```

我们先看看 SessionFactory 是怎样创建的，在方法 buildSessionFactory 中，这个方法很长，但实现却不难理解。代码里包含了创建 Hibernate 的 SessionFactory 以及配置 Hibernate 的详尽过程，其中有一个很长的对 SessionFactory 进行配置的清单，以及 SessionFactory 的各个属性的配置。具体的代码在 LocalSessionFactoryBean 中，如代码清单 5-18 所示。

代码清单 5-18 LocalSessionFactoryBean 的 buildSessionFactory

```
protected SessionFactory buildSessionFactory() throws Exception {
    // Create Configuration instance.
```

Download at http://www.pin5i.com/

```
        Configuration config = newConfiguration();
        /**
        * 配置数据源、事务管理器和LobHandler到Holder中，这些Holder是
        * ThreadLocal变量，这样这些资源就和线程绑定了。
        */
        DataSource dataSource = getDataSource();
        if (dataSource != null) {
            // Make given DataSource available for SessionFactory configuration.
            configTimeDataSourceHolder.set(dataSource);
        }
        if (this.jtaTransactionManager != null) {
            // Make Spring-provided JTA TransactionManager available.
    configTimeTransactionManagerHolder.set(this.jtaTransactionManager);
        }
        if (this.cacheProvider != null) {
            // Make Spring-provided Hibernate CacheProvider available.
            configTimeCacheProviderHolder.set(this.cacheProvider);
        }
        if (this.lobHandler != null) {
            // Make given LobHandler available for SessionFactory configuration.
           // Do early because because mapping resource might refer to custom types.
            configTimeLobHandlerHolder.set(this.lobHandler);
        }
        /**
        * Analogous to Hibernate EntityManager's Ejb3Configuration:
        * Hibernate doesn't allow setting the bean ClassLoader explicitly,
        * so we need to expose it as thread context ClassLoader accordingly.
        */
        Thread currentThread = Thread.currentThread();
       ClassLoader threadContextClassLoader = currentThread.getContextClassLoader();
        boolean overrideClassLoader =
                (this.beanClassLoader != null && !this.beanClassLoader.equals(thread
     ContextClassLoader));
        if (overrideClassLoader) {
            currentThread.setContextClassLoader(this.beanClassLoader);
        }
        /**
        * 很长的配置清单，对Hibernate的各个属性进行配置，这里通过
        * Hibernate的Configuration来实现配置。
        */
        try {
            if (isExposeTransactionAwareSessionFactory()) {
                /**
                * Set Hibernate 3.1+ CurrentSessionContext implementation,
                * providing the Spring-managed Session as current Session.
                * Can be overridden by a custom value for the corresponding Hibernate property.
                */
                config.setProperty(
                        Environment.CURRENT_SESSION_CONTEXT_CLASS, SpringSession
        Context.class.getName());
            }
            if (this.jtaTransactionManager != null) {
                // Set Spring-provided JTA TransactionManager as Hibernate property.
                config.setProperty(
                        Environment.TRANSACTION_STRATEGY, JTATransactionFactory.
        class.getName());
                config.setProperty(
                        Environment.TRANSACTION_MANAGER_STRATEGY, LocalTransaction
```

Download at http://www.pin5i.com/

```
ManagerLookup.class.getName());
        }
        else {
            /**
            * Makes the Hibernate Session aware of the presence of a
            * Spring-managed transaction.
            */
            // Also sets connection release mode to ON_CLOSE by default.
            config.setProperty(
                    Environment.TRANSACTION_STRATEGY, SpringTransactionFactory.
                    class.getName());
        }
        if (this.entityInterceptor != null) {
            // Set given entity interceptor at SessionFactory level.
            config.setInterceptor(this.entityInterceptor);
        }
        if (this.namingStrategy != null) {
            // Pass given naming strategy to Hibernate Configuration.
            config.setNamingStrategy(this.namingStrategy);
        }
        if (this.typeDefinitions != null) {
            // Register specified Hibernate type definitions.
            Mappings mappings = config.createMappings();
            for (TypeDefinitionBean typeDef : this.typeDefinitions) {
                mappings.addTypeDef(typeDef.getTypeName(), typeDef.getTypeClass(),
                typeDef.getParameters());
            }
        }
        if (this.filterDefinitions != null) {
            // Register specified Hibernate FilterDefinitions.
            for (FilterDefinition filterDef : this.filterDefinitions) {
                config.addFilterDefinition(filterDef);
            }
        }
        if (this.configLocations != null) {
            for (Resource resource : this.configLocations) {
                // Load Hibernate configuration from given location.
                config.configure(resource.getURL());
            }
        }
        if (this.hibernateProperties != null) {
            // Add given Hibernate properties to Configuration.
            config.addProperties(this.hibernateProperties);
        }
        if (dataSource != null) {
            Class providerClass = LocalDataSourceConnectionProvider.class;
            if(isUseTransactionAwareDataSource() || dataSource instanceof Transaction
            AwareDataSourceProxy) {
                providerClass = TransactionAwareDataSourceConnectionProvider.class;
            }
            else if (config.getProperty(Environment.TRANSACTION_MANAGER_STRATEGY)!=null){
                providerClass = LocalJtaDataSourceConnectionProvider.class;
            }
            // Set Spring-provided DataSource as Hibernate ConnectionProvider.
            config.setProperty(Environment.CONNECTION_PROVIDER, providerClass.
            getName());
        }
        if (this.cacheProvider != null) {
            // Expose Spring-provided Hibernate CacheProvider.
```

Download at http://www.pin5i.com/

```
        config.setProperty(Environment.CACHE_PROVIDER, LocalCacheProvider
        Proxy.class.getName());
    }
    if (this.mappingResources != null) {
        // Register given Hibernate mapping definitions, contained in resource files.
        for (String mapping : this.mappingResources) {
            Resource resource = new ClassPathResource(mapping.trim(), this.bean
            ClassLoader);
            config.addInputStream(resource.getInputStream());
        }
    }
    if (this.mappingLocations != null) {
        // Register given Hibernate mapping definitions, contained in resource files.
        for (Resource resource : this.mappingLocations) {
            config.addInputStream(resource.getInputStream());
        }
    }
    if (this.cacheableMappingLocations != null) {
        /**
        * Register given cacheable Hibernate mapping definitions,
        * read from the file system.
        */
        for (Resource resource : this.cacheableMappingLocations) {
            config.addCacheableFile(resource.getFile());
        }
    }
    if (this.mappingJarLocations != null) {
        // Register given Hibernate mapping definitions, contained in jar files.
        for (Resource resource : this.mappingJarLocations) {
            config.addJar(resource.getFile());
        }
    }
    if (this.mappingDirectoryLocations != null) {
        // Register all Hibernate mapping definitions in the given directories.
        for (Resource resource : this.mappingDirectoryLocations) {
            File file = resource.getFile();
            if (!file.isDirectory()) {
                throw new IllegalArgumentException(
                        "Mapping directory location [" + resource + "] does
                        not denote a directory");
            }
            config.addDirectory(file);
        }
    }
    /**
    * Tell Hibernate to eagerly compile the mappings that we registered,
    * for availability of the mapping information in further processing.
    */
    // 编译Hibernate需要的mapping信息.
    postProcessMappings(config);
    config.buildMappings();
    if (this.entityCacheStrategies != null) {
        // Register cache strategies for mapped entities.
        for (Enumeration classNames = this.entityCacheStrategies.property
            Names(); classNames.hasMoreElements();) {
            String className = (String) classNames.nextElement();
            String[] strategyAndRegion =
            StringUtils.commaDelimitedListToStringArray
            (this.entityCacheStrategies.getProper ty(className));
```

Download at http://www.pin5i.com/

```
                if (strategyAndRegion.length > 1) {
                    config.setCacheConcurrencyStrategy(className, strategyAndRegion[0],
                    strategyAndRegion[1]);
                }
                else if (strategyAndRegion.length > 0) {
                   config.setCacheConcurrencyStrategy(className, strategyAndRegion[0]);
                }
            }
        }
        if (this.collectionCacheStrategies != null) {
            // Register cache strategies for mapped collections.
            for (Enumeration collRoles = this.collectionCacheStrategies.
                propertyNames(); collRoles.hasMoreElements();) {
                String collRole = (String) collRoles.nextElement();
                String[] strategyAndRegion =
                StringUtils.commaDelimitedListToStringArray
                (this.collectionCacheStrategies.getPr operty(collRole));
                if (strategyAndRegion.length > 1) {
                    config.setCollectionCacheConcurrencyStrategy(collRole, strategyAnd
                    Region[0], strategyAndRegion[1]);
                }
                else if (strategyAndRegion.length > 0) {
                    config.setCollectionCacheConcurrencyStrategy(collRole,
                   strategyAnd Region[0]);
                }
            }
        }
        if (this.eventListeners != null) {
            // Register specified Hibernate event listeners.
            for (Map.Entry<String, Object> entry : this.eventListeners.entrySet()) {
                String listenerType = entry.getKey();
                Object listenerObject = entry.getValue();
                if (listenerObject instanceof Collection) {
                   Collection<Object> listeners = (Collection<Object>) listenerObject;
                    EventListeners listenerRegistry = config.getEventListeners();
                    Object[] listenerArray =
                    (Object[]) Array.newInstance(listenerRegistry.
                    getListenerClassFor(listenerType), listeners.size());
                    listenerArray = listeners.toArray(listenerArray);
                    config.setListeners(listenerType, listenerArray);
                }
                else {
                    config.setListener(listenerType, listenerObject);
                }
            }
        }
        // Perform custom post-processing in subclasses.
        postProcessConfiguration(config);
        // Build SessionFactory instance.
        // 这里是根据Configuration配置创建SessionFactory的地方。
        logger.info("Building new Hibernate SessionFactory");
        this.configuration = config;
        return newSessionFactory(config);
    }
    //最后把和线程绑定的资源清空。
    finally {
        if (dataSource != null) {
            // Reset DataSource holder.
            configTimeDataSourceHolder.set(null);
```

Download at http://www.pin5i.com/

```
            }
            if (this.jtaTransactionManager != null) {
                // Reset TransactionManager holder.
                configTimeTransactionManagerHolder.set(null);
            }
            if (this.cacheProvider != null) {
                // Reset CacheProvider holder.
                configTimeCacheProviderHolder.set(null);
            }
            if (this.lobHandler != null) {
                // Reset LobHandler holder.
                configTimeLobHandlerHolder.set(null);
            }
            if (overrideClassLoader) {
                // Reset original thread context ClassLoader.
    currentThread.setContextClassLoader(threadContextClassLoader);
            }
        }
    }
    //调用 Configuration 生成 SessionFactory.
    protected SessionFactory newSessionFactory(Configuration config) throws
        HibernateException {
        return config.buildSessionFactory();
    }
```

在上面的代码中可以看到，LocalSessionFactory 的作用包括：首先读取 Hibernate 的配置，然后生成 SessionFactory。SessionFactory 生成之后，我们看看在 Spring 中是怎样使用 Hibernate 的。

5.4.2 HibernateTemplate 的实现

与 JDBC 的使用类似，在使用 Hibernate 完成 O/R 映射工作时，Spring 为用户提供了 HibernateTemplate。在使用 HibernateTemplate 时，和 JdbcTemplate 一样，Spring 仍然使用相同的模式，即通过 execute 回调来完成。为了解 HibernateTemplate 的实现，我们可以分析它的源代码，如代码清单 5-19 所示。

代码清单 5-19 HibernateTemplate 的 execute

```
    public <T> T execute(HibernateCallback<T> action) throws DataAccessException {
        return doExecute(action, false, false);
    }
    protected <T> T doExecute(HibernateCallback<T> action,
        boolean enforceNewSession,boolean enforceNativeSession)
            throws DataAccessException {
        Assert.notNull(action, "Callback object must not be null");
        //这里取得 Hibernate 的 Session，取得的过程会在下面进行分析.
        //是否强制需要新的 Session?如果需要，那么直接通过 SessionFactory 打开一个新的 Session。
        //否则需要结合配置和当前 Transaction 的情况来使用 Session。
        Session session = (enforceNewSession ?
                SessionFactoryUtils.getNewSession(getSessionFactory(), getEntity
               Interceptor()):getSession());
        //判断是否存在 Transaction，如果有，那么使用的就是当前 Transaction 的 Session。
        boolean existingTransaction = (!enforceNewSession &&
                (!isAllowCreate() || SessionFactoryUtils.isSessionTransactional
                (session, getSessionFactory())));
```

Download at http://www.pin5i.com/

```
        if (existingTransaction) {
            logger.debug("Found thread-bound Session for HibernateTemplate");
        }
        FlushMode previousFlushMode = null;
        try {
            previousFlushMode = applyFlushMode(session, existingTransaction);
            enableFilters(session);
            Session sessionToExpose =
                    (enforceNativeSession||isExposeNativeSession()?session: create
                    SessionProxy(session));
            //对 HibernateCallBack 中回调函数的调用，Session 作为参数可以让回调函数使用。
            T result = action.doInHibernate(sessionToExpose);
            flushIfNecessary(session, existingTransaction);
            return result;
        }
        catch (HibernateException ex) {
            throw convertHibernateAccessException(ex);
        }
        catch (SQLException ex) {
            throw convertJdbcAccessException(ex);
        }
        catch (RuntimeException ex) {
            // Callback code threw application exception...
            throw ex;
        } //如果存在 Transaction,那么在当前的回调使用完 Session 后，不关闭这个 Session。
        finally {
            if (existingTransaction) {
                logger.debug("Not closing pre-bound Hibernate Session after
                HibernateTemplate");
                disableFilters(session);
                if (previousFlushMode != null) {
                    session.setFlushMode(previousFlushMode);
                }
            } //如果不存在 Transaction,那么关闭当前的 Session。
            else {
                // Never use deferred close for an explicitly new Session.
                if (isAlwaysUseNewSession()) {
                    SessionFactoryUtils.closeSession(session);
                }
                else {
                    SessionFactoryUtils.closeSessionOrRegisterDeferredClose
                    (session, getSession Factory());
                }
            }
        }
    }
```

从代码中可以看到，execute 方法对 Hibernate 的处理与单独使用 Hibernate 完成 O/R 映射的过程非常类似。在使用 Hibernate 时，往往需要考虑事务管理，而在 execute 方法中，也需要对事务管理的使用做一些处理，这些都和 Session 的获得有关。

如果需要创建 Session，可由已经准备好的 SessionFactory 来完成。在完成了对 Hibernate 的 FlushMode 的设置以后，通过调用定义好的回调实现（比如在 execute 方法参数中的 HibernateCallback 对象的 doInHibernate 方法），从而通过 Hibernate 来完成的数据持久化操作。可以看到，在这个回调方法中封装了对 Hibernate 的具体使用。在回调完成以后，最后完成对

Download at http://www.pin5i.com/

当前 Session 的处理。对于是否关闭当前 Session，是需要根据事务管理的情况来决定的。

对于我们常用的 HibernateTemplate 的 API，在通过它们在 Spring 应用中使用 Hibernate 的时候，大多数都是通过调用前面分析过的 HibernateTemplate 的 execute 方法完成的，这个实现一般而言通过为 excute 方法提供 HibernateCallback 的回调来完成。以 HibernateTemplate 中实现的 find 方法为例来说明这个实现原理，对于这个 find 方法，它负责根据给出的 HQL 来完成 Hibernate 查询。在这个例子中，我们可以看到 Spring 使用 Hibernate 的实现思路，我们从它在 HibernateTemplate 中的具体实现入手，如代码清单 5-20 所示。在这个 find 方法里，我们看到了定义 HibernateCallback 回调的地方，这个回调方法通过 session 来创建一个 query，在为这个 query 对象配置好参数以后，启动查询得到查询结果，查询结果是一个持有对象的 List。这里对 Hibernate 的使用，与我们单独使用 Hibernate 的 query 和 HQL 语句来完成对象查询的实现是一样的。

代码清单 5-20 HibernateTemplate 的 find 方法的实现

```
public List find(final String queryString, final Object... values) throws
    DataAccessException {
    return executeWithNativeSession(new HibernateCallback<List>() {
        //这里是提供回调函数的地方。
        public List doInHibernate(Session session) throws HibernateException {
            //使用 Session 创建 Hibernate 的 query.
            Query queryObject = session.createQuery(queryString);
            prepareQuery(queryObject);
            //这里为 query 配置参数，并返回 query 的结果.
            if (values != null) {
                for (int i = 0; i < values.length; i++) {
                    queryObject.setParameter(i, values[i]);
                }
            }
            return queryObject.list();
        }
    });
}
```

除了上面看到的例子，HibernateTemplate 中还有许多其他的方法，通过名字就可以大致了解到它们所代表的对 Hibernate 的相关操作。对于这些操作，因为只要看看源代码，就会发现在 HibernateTemplate 的回调函数中使用的就是 Hibernate 的 API。在 HibernateTemplate 中实现的是对 Hibernate API 的封装，通过这一层 template 的封装来简化 Hibernate 的使用。

5.4.3 Session 的管理

在这些对 Hibernate API 的使用中离不开 Session ，它是 Hibernate 处理持久化对象的上下文，也是使用 Hibernate 完成持久化工作的一个核心类。我们在前面已经看到，Spring 对 Hibernate Session 的管理功能是由 SessionFactoryUtils 提供的，如代码清单 5-21 所示。在得到 Session 的过程中，考虑到事务处理的情况，如果当前线程已经绑定事务，而这个事务是通过 Hibernate 的 Session 来实现的，那么 Session 用的 Connection 就应该是与当前线程绑定的那个。对于新建 Session 的情况，和单独使用 SessionFactory 打开新 Session 一样，使用 SessionFactory 的 openSession 来创建新的 Session。这些与 Session 相关的动作执行都和当

Download at http://www.pin5i.com/

前的事务管理状态有很大的关系，关于 Spring 的事务管理，我们会在第 6 章进行详细的分析。

代码清单 5-21 SessionFactoryUtils 创建 Session

```
//这个方法使用 SessionFactory 创建新的 Session。
public static Session getNewSession(SessionFactory sessionFactory, Interceptor
    entityInterceptor) {
    Assert.notNull(sessionFactory, "No SessionFactory specified");
    try {
        SessionHolder sessionHolder = (SessionHolder)
    TransactionSynchronization Manager.getResource(sessionFactory);
        if (sessionHolder != null && !sessionHolder.isEmpty()) {
            if (entityInterceptor != null) {
                return  sessionFactory.openSession(sessionHolder.getAnySession().
               connection(), entityInterceptor);
            }
            else {
                return sessionFactory.openSession
               (sessionHolder.getAnySession(). connection());
            }
        }
        else {
            if (entityInterceptor != null) {
                return sessionFactory.openSession(entityInterceptor);
            }
            else {
                return sessionFactory.openSession();
            }
        }
    }
    catch (HibernateException ex) {
        throw new DataAccessResourceFailureException
      ("Could not open Hibernate Session", ex);
    }
}
```

对于不需要强制新建 Session 的情况，Session 的取得就相对比较复杂，在 SessionFactoryUtils 中的 getSession 的实现如代码清单 5-22 所示。它包括了与事务处理相关的部分和 Session 与线程绑定的处理，根据事务处理的情况来得到需要的 Session，这个 Session 可以是以前与线程绑定的那个 Session，也可以是新创建的。

代码清单 5-22 SessionFactoryUtils 的 getSession

```
public static Session getSession(
        SessionFactory sessionFactory, Interceptor entityInterceptor,
        SQLExceptionTranslator jdbcExceptionTranslator) throws DataAccessResource
   FailureException {
    try {
        return doGetSession(sessionFactory,entityInterceptor,jdbcExceptionTranslator,true);
    }
    catch (HibernateException ex) {
        throw new DataAccessResourceFailureException("Could not open Hibernate
       Session", ex);
    }
}
//doGetSession 是实际取得 Session 的地方。
private static Session doGetSession(
        SessionFactory sessionFactory, Interceptor entityInterceptor,
        SQLExceptionTranslator jdbcExceptionTranslator, boolean allowCreate)
```

Download at http://www.pin5i.com/

```
        throws HibernateException, IllegalStateException {
    Assert.notNull(sessionFactory, "No SessionFactory specified");
    //取得和当前线程绑定的SessionHolder。
    SessionHolder sessionHolder = (SessionHolder) TransactionSynchronization
    Manager.getResource(sessionFactory);
    if (sessionHolder != null && !sessionHolder.isEmpty()) {
        // pre-bound Hibernate Session.
        // 当前线程中有SessionHolder，说明在前面线程的执行中已经创建过Session了。
        Session session = null;
        if (TransactionSynchronizationManager.isSynchronizationActive() &&
                sessionHolder.doesNotHoldNonDefaultSession()) {
            /**
            * Spring transaction management is active ->
            * register pre-bound Session with it for transactional flushing.
            */
            // 在Spring的Transaction管理中，使用的是与当前事务绑定的Session。
            session = sessionHolder.getValidatedSession();
            if (session != null && !sessionHolder.isSynchronizedWithTransaction()) {
                logger.debug("Registering Spring transaction synchronization
               for existing Hibernate Session");
        TransactionSynchronizationManager.registerSynchronization(
        new SpringSessionSynchronization(sessionHolder, sessionFactory,
        jdbcExceptionTranslator, false));
                sessionHolder.setSynchronizedWithTransaction(true);
                // Switch to FlushMode.AUTO, as we have to assume a thread-bound Session.
                // with FlushMode.MANUAL, which needs to allow flushing within the transaction.
                // 设置FlushMode的状态。
                FlushMode flushMode = session.getFlushMode();
                if (flushMode.lessThan(FlushMode.COMMIT) &&
!TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {
                    session.setFlushMode(FlushMode.AUTO);
                    sessionHolder.setPreviousFlushMode(flushMode);
                }
            }
        }
        else {
            // No Spring transaction management active -> try JTA transaction synchronization.
            session = getJtaSynchronizedSession(sessionHolder, sessionFactory,
    jdbcExceptionTranslator);
        } //返回已有的Session，这种情况下，不需要创建新的Session。
        if (session != null) {
            return session;
        }
    }
    //创建新的Session，因为当前线程中还没有建立过Session。
    logger.debug("Opening Hibernate Session");
    Session session = (entityInterceptor != null ?
            sessionFactory.openSession(entityInterceptor) : sessionFactory.openSession());
    // Use same Session for further Hibernate actions within the transaction.
    // Thread object will get removed by synchronization at transaction completion.
    // 这里把新创建的Session与线程绑定，并同时根据事务管理器的设置进行设置。
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
        /**
        * We're within a Spring-managed transaction, possibly
        * from JtaTransactionManager.
        */
        logger.debug("Registering Spring transaction synchronization for new
    Hibernate Session");
```

Download at http://www.pin5i.com/

```
            SessionHolder holderToUse = sessionHolder;
            if (holderToUse == null) {
                holderToUse = new SessionHolder(session);
            }
            else {
                holderToUse.addSession(session);
            }
            if (TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {
                session.setFlushMode(FlushMode.MANUAL);
            }
            TransactionSynchronizationManager.registerSynchronization(
                    new SpringSessionSynchronization(holderToUse, sessionFactory, jdbc
                    ExceptionTranslator, true));
            holderToUse.setSynchronizedWithTransaction(true);
            if (holderToUse != sessionHolder) {
                TransactionSynchronizationManager.bindResource(sessionFactory, holderToUse);
            }
        }
        else {
            // No Spring transaction management active -> try JTA transaction synchronization.
            registerJtaSynchronization(session, sessionFactory, jdbcExceptionTranslator,
            sessionHolder);
        }
        // Check whether we are allowed to return the Session.
        if (!allowCreate && !isSessionTransactional(session, sessionFactory)) {
            closeSession(session);
            throw new IllegalStateException("No Hibernate Session bound to thread, " +
                "and configuration does not allow creation of non-transactional one
               here");
        }
        return session;
    }
```

在得到需要的 Session 以后，用户就可以像其他的使用在 Spring 中出现的 Template 那样，来使用 Hibernate 的基本功能了。我们可以在 HibernateTemplate 的使用中看到，Spring 已经为 Session 的获取和关闭以及与事务处理相关的工作，比如和线程的绑定操作等都做好了封装，这些都是在 Hibernate 应用中需要处理的基本过程。从以上的实现中可以看到，这些重复的处理过程都在 HibernateTemplate 里完成了。

5.5 Spring 驱动 iBatis 的实现

5.5.1 创建 SqlMapClient

iBatis 也是一个优秀的 ORM 开源产品，它的实现有其独到之处。相比于 Hibernate 动态生成 SQL 的实现方式，iBatis 采用 XML 描述的 SQL 语句来控制读取数据库的操作，这让用户使用起来觉得非常方便。与其他的 ORM 产品相比，iBatis 的特点就是它使用起来也非常简单。iBatis 是 Apache 基金会的一个子项目，如果想了解 iBatis 项目的详细情况，可以到网站 http://ibatis.apache.org 看一看。

与前面分析 Hibernate 的实现一样，这里也从 Spring 读取 iBatis 的配置入手，了解 Spring 驱动 iBatis 的实现。在 Spring 的 IoC 容器中，iBatis 实例通常通过 SqlMapClientFactoryBean 来设置，这个 FactoryBean 的具体实现如代码清单 5-23 所示。在 SqlMapClientFactoryBean 中

Download at http://www.pin5i.com/

完成 SqlMapClient 的创建，SqlMapClient 是用户使用 iBatis 操作数据库的主要类。这个创建过程包括了一些对 SqlMapClient 的配置过程（比如对数据源 DataSource 的配置）以及配置参数等。这个创建过程是在 afterPropertiesSet 中完成的，它在依赖注入完成以后被 IoC 容器回调。对于这个 InitializingBean 接口的 IoC 回调方法，我们都已经很熟悉了。

代码清单 5-23 SqlMapClientFactoryBean 的 getObject

```
//返回 SqlMapClient。
public SqlMapClient getObject() {
    return this.sqlMapClient;
}
public void afterPropertiesSet() throws Exception {
    if (this.lobHandler != null) {
        // Make given LobHandler available for SqlMapClient configuration.
        // Do early because because mapping resource might refer to custom types.
        configTimeLobHandlerHolder.set(this.lobHandler);
    }
    //这里是创建 SqlMapClient 的入口。
    try {
        this.sqlMapClient = buildSqlMapClient(this.configLocations, this.mappingLocations,
        this.sqlMapClientProperties);
        // Tell the SqlMapClient to use the given DataSource, if any.
        if (this.dataSource != null) {
            TransactionConfig transactionConfig = (TransactionConfig) this.transaction
            ConfigClass.newInstance();
            DataSource dataSourceToUse = this.dataSource;
            if (this.useTransactionAwareDataSource && !(this.dataSource instanceof
               TransactionAwareDataSourceProxy)) {
                dataSourceToUse = new TransactionAwareDataSourceProxy(this.dataSource);
            }
            transactionConfig.setDataSource(dataSourceToUse);
transactionConfig.initialize(this.transactionConfigProperties);
            applyTransactionConfig(this.sqlMapClient, transactionConfig);
        }
    }
    finally {
        if (this.lobHandler != null) {
            // Reset LobHandler holder.
            configTimeLobHandlerHolder.set(null);
        }
    }
}
//具体的 SqlMapClient 的创建过程。
protected SqlMapClient buildSqlMapClient(
        Resource[] configLocations, Resource[] mappingLocations, Properties properties)
        throws IOException {
    if (ObjectUtils.isEmpty(configLocations)) {
        throw new IllegalArgumentException("At least 1 'configLocation' entry is
    required");
    }
    SqlMapClient client = null;
    SqlMapConfigParser configParser = new SqlMapConfigParser();
    for (Resource configLocation : configLocations) {
        InputStream is = configLocation.getInputStream();
        try {
            client = configParser.parse(is, properties);
        }
```

Download at http://www.pin5i.com/

```
        catch (RuntimeException ex) {
            throw new NestedIOException("Failed to parse config resource: " +
    configLocation, ex.getCause());
        }
    }
    if (mappingLocations != null) {
        SqlMapParser mapParser = SqlMapParserFactory.createSqlMapParser(configParser);
        for (Resource mappingLocation : mappingLocations) {
            try {
                mapParser.parse(mappingLocation.getInputStream());
            }
            catch (NodeletException ex) {
                throw new NestedIOException("Failed to parse mapping resource:
    " + mappingLocation, ex);
            }
        }
    }
    return client;
}
```

5.5.2 SqlMapClientTemplate 的实现

在 SqlMapClient 创建完以后，Spring 还是使用它一贯的办法，封装一个 Template 类，在对 iBatis 的 SqlMapClient 的使用中，用 SqlMapClientTemplate 来封装对 iBatis 的操作。具体实现在 SqlMapClientTemplate 的源代码中，如代码清单 5-24 所示。

代码清单 5-24 SqlMapClientTemplate 的 execute

```
//使用 SqlMapExecutor 来完成数据的操作.
public <T> T execute(SqlMapClientCallback<T> action) throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");
    Assert.notNull(this.sqlMapClient, "No SqlMapClient specified");
    /**
    * We always needs to use a SqlMapSession, as we need to pass a Spring-managed
    * Connection (potentially transactional) in. This shouldn't be necessary if
    * we run against a TransactionAwareDataSourceProxy underneath, but unfortunately
    * we still need it to make iBATIS batch execution work properly: If iBATIS
    * doesn't recognize an existing transaction, it automatically executes the
    * batch for every single statement...
    */
    SqlMapSession session = this.sqlMapClient.openSession();
    if (logger.isDebugEnabled()) {
        logger.debug("Opened SqlMapSession [" + session + "] for iBATIS operation");
    }
    Connection ibatisCon = null;
    //这里获取 DataSource 数据源.
    try {
        Connection springCon = null;
        DataSource dataSource = getDataSource();
        boolean transactionAware = (dataSource instanceof TransactionAwareDataSourceProxy);
        // Obtain JDBC Connection to operate on...
        /**
        *这里获取 Connection，如果已经在 Spring 的事务管理之下，数据源可以直接使用，
        *否则，使用 DataSourceUtils 来产生需要的 Connection，并将得到的 Connection
        *置于 Spring 的事务管理之中
        */
        try {
```

Download at http://www.pin5i.com/

```
            ibatisCon = session.getCurrentConnection();
            if (ibatisCon == null) {
                springCon = (transactionAware ?
                dataSource.getConnection() : DataSourceUtils.doGet
                Connection(dataSource));
                session.setUserConnection(springCon);
                if (logger.isDebugEnabled()) {
                    logger.debug("Obtained JDBC Connection [" + springCon + "]
                    for iBATIS operation");
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Reusing JDBC Connection [" + ibatisCon + "]
                    for iBATIS operation");
                }
            }
        }
        catch (SQLException ex) {
            throw new CannotGetJdbcConnectionException("Could not get JDBC
            Connection", ex);
        }
        // Execute given callback...
        // 这里执行 SqlMapClientCallback 的回调。
        try {
            return action.doInSqlMapClient(session);
        }
        catch (SQLException ex) {
            throw getExceptionTranslator().translate("SqlMapClient operation",
            null, ex);
        } //释放 DataSource。
        finally {
            try {
                if (springCon != null) {
                    if (transactionAware) {
                        springCon.close();
                    }
                    else {
                        DataSourceUtils.doReleaseConnection(springCon, dataSource);
                    }
                }
            }
            catch (Throwable ex) {
                logger.debug("Could not close JDBC Connection", ex);
            }
        }
        // Processing finished - potentially session still to be closed.
    }
    finally {
        /**
        * Only close SqlMapSession if we know we've actually opened it
        * at the present level.
        */
        if (ibatisCon == null) {
            session.close();
        }
    }
}
```

Download at http://www.pin5i.com/

对于 SqlMapClientTemplate 的其他方法，与 Spring 封装的 HibernateTemplate 实现一样，也是通过调用这个 execute 方法并提供回调函数来实现的，以我们常用的 queryForObject 为例，去了解这个 queryForObject 的实现过程，这个实现过程如代码清单 5-25 所示。

代码清单 5-25 SqlMapClientTemplate 的 queryForObject

```
public Object queryForObject(final String statementName, final Object parameterObject)
        throws DataAccessException {
    //这里调用了 execute，同时提供了使用 SqlMapExecutor 的回调函数。
    return execute(new SqlMapClientCallback<Object>() {
        public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
            return executor.queryForObject(statementName, parameterObject);
        }
    });
}
```

在得到需要的 SqlMapClient 以后，应用就可以像使用其他 Template 那样使用 iBatis 的基本功能了。在 SqlMapClientTemplate 中，Spring 已经为我们对 DataSource 的获取和关闭、事务处理的绑定做好了封装，同时在这些基础上为用户提供了许多便利的 iBatis 的操作实现。与前面的 JdbcTemplate、HibernateTemplate 一样，实现的方法也非常类似。感兴趣的读者不妨自己比较一下这三种 Template 实现的异同，相信能加深对这些 Spring 数据库操作组件的理解。

5.6 小结

本章详细讲解了 Spring 与数据库操作相关的部分。在 Spring 中，数据库的操作组件大致可以分为两个大类：一类是 Spring 通过 JDBC 的封装，从而为用户直接提供对数据库进行操作的组件；另一类是 Spring 通过集成现有的 ORM 工具，在使用 ORM 工具的基础上，提供对数据库进行操作的组件。对于已有很多 ORM 领域的优秀产品实现，都成为 Spring ORM 部分需要集成的目标，其中包括我们熟悉的 Hibernate、iBatis、JPA、JDO 等。在本章中，我们选取了 Hibernate 和 iBatis 作为例子来分析 Spring 对 ORM 产品驱动的实现原理。

首先涉及的内容是对 Spring JDBC 实现原理的分析，通过了解 Spring JDBC 的实现原理我们发现，使用 Spring JDBC 可以帮助我们完成许多基本的数据库操作，这些基本操作在 Spring 中是通过 JdbcTeamplate 来提供的。在 JdbcTemplate 的基础上，Spring 还提供了许多 RDBMS 对象，通过对这些对象的操作，可以帮助我们更便利地进行数据库应用的开发。在这里，并没有使用像 Hibernate 这样的 ORM 方案，但通过灵活运用 RDBMS 的功能，也能完成一些简单的数据和 Java 对象的映射工作，这在很大程度上方便了使用 JDBC 进行数据库操作的应用开发人员。例如，可以使用 MappingSqlQuery 来将表数据直接映射到一个对象集合。这有点像简单的 ORM 工具完成的功能，但是因为映射比较简单，使用基本的 JDBC 就可以了，还没有复杂到需要引入第三方 ORM 工具的地步。从这个角度来看，Spring 的确为用户提供了许多灵活的选择来为应用代码的实现提供支持。至于选择什么样的技术方案来支撑应用的开发，需要 Spring 应用开发者根据自己的项目特点来确定。但 Spring 已经为用户准备了非常丰富的备选方案，在这点上也充分体现了 Spring 的应用平台价值。

Download at http://www.pin5i.com/

与通过 JdbcTemplate 来提供对 JDBC 操作的支持一样，Spring 为使用 Hiberante 实现 O/R 映射的应用提供了 HibernateTemplate。对于 HibernateTemplate，它的使用和实现的方法都与 JdbcTemplate 非常类似，同时 HibernateTemplate 还封装了对 Hibernate Session 的管理，免去了应用自己去实现与线程绑定，以及与事务管理相结合的重复工作。通过这一层简单的封装，让用户在 Spring 的环境中使用 Hibernate 更加简单高效。

iBatis 作为使用上比较简洁的另一个知名 ORM 产品，Spring 没有理由不对它提供支持。在这里，与为 Hibernate 提供支持一样，Spring 依然通过 FactoryBean 来完成对 iBatis 的配置，使用 Template 来封装相应的操作。尽管 iBatis 的使用已经很简单了，但经过 Spring 内部的封装后，iBatis 的使用变得更加简洁而有力了。

万紫千红总是春，对于企业应用中很普遍的数据库操作，Spring 提供了这么多的技术选择，就像在春天到来的时候，在广阔的原野上盛开着的朵朵鲜花，任君采撷；同时，通过 Spring 的有效封装，在这些选择的实现中，又体现出一种内在的和谐与统一。理解了其中的机理，足以让我们举一反三，而自由地去选择适合自己的数据库操作方案，从而更游刃有余地去满足应用的需求。

Download at http://www.pin5i.com/

第 6 章

Spring 事务处理的实现

子曰："参乎！吾道一以贯之。"
——【春秋】孔子《论语》里仁篇

6.1　Spring 与事务处理

Java EE 应用中的事务处理是一个重要并且涉及范围很广的领域。事务管理的实现往往涉及并发和数据一致性方面的问题，如果具有这方面的知识背景，可以增强对事务处理的理解。作为应用平台的 Spring，提供了在多种环境中配置和使用事务处理的能力，也就是说通过使用 Spring 的事务处理，可以把事务处理的工作统一起来，并为事务处理提供通用的支持。

由于这方面的内容比较多且比较复杂，本章只阐述一些在事务处理中最为基本的使用场景，即 Spring 是怎样实现对单个数据库局部事务的处理的。在涉及单个数据库局部事务的事务处理中，事务的最终实现和数据库的支持是紧密相关的。对局部数据库事务来说，一个事务处理的操作单元往往对应着一系列的数据库操作。数据库产品对这些数据库的 SQL 操作已经提供了原子性的支持，对 SQL 操作而言，它的操作结果有两种：一种是提交成功，数据库操作成功；另一种是回滚，数据库操作不成功，恢复到操作以前的状态。

在事务处理中，事务处理单元的设计与相应的业务逻辑设计有很紧密的联系。在很多情况下，都不会只有一个单独的数据库操作，而是一组数据库操作。在事务处理过程中，首先涉及的是事务处理单元划分的问题。在 Spring 中，借助 IoC 容器的强大配置能力，为应用提供了声明式的事务划分方式，这种声明式的事务处理为 Spring 应用使用事务管理提供了统一的方式。有了 Spring 事务管理的支持，只需要通过一些简单的配置，应用就能完成复杂的事务处理工作，从而为用户使用事务处理提供了很大的方便。

关于声明式事务管理的实现，我们将会在本章中进行详细的分析。如果我们了解 Spring IoC 和 AOP 的实现原理，会对深刻理解声明式事务处理的实现有很大的帮助。对 IoC 容器和 AOP 的实现原理感兴趣的读者，可以参考本书的第 2 章和第 3 章的内容，从而为了解事务处理的实现做好准备工作。同时，本章涉及的具体事务管理实现原理，需要参考 Spring 的一些源代码，需要在 Eclipse 中导入相应的项目包 `org.springframework.transaction`。

6.2　声明式事务处理的基本过程

在使用 Spring 声明式事务处理的时候，一种常用的方法是，通过结合 IoC 容器和 Spring 已有的 `TransactionProxyFactoryBean` 来对事务管理进行配置，比如，可以在这个

Download at http://www.pin5i.com/

TransactionProxyFactoryBean 中为事务方法配置传播行为、并发事务隔离级别这些事务处理的属性，从而来对声明式事务的处理提供指导。具体来说，在以下的内容中，在对声明式事务处理的原理分析中，对于声明式事务处理的实现，大致可以分为以下几个部分：

- 读取和处理在 IoC 容器中配置的事务处理属性，并转化为 Spring 事务处理需要的内部数据结构。具体来说，这里涉及的类是 TransactionAttributeSourceAdvisor，从名字中可以知道，它是一个 AOP 通知器，Spring 使用这个通知器来完成对事务处理属性值的处理。处理的结果是，在 IoC 容器中配置的事务处理属性信息，会被读入并转化成 TransactionAttribute 表示的数据对象，这个数据对象是 Spring 对事物处理属性值的数据抽象，对这些属性的处理是和 TransactionProxyFactoryBean 拦截下来的事务方法的处理结合起来的。
- Spring 事务处理模块实现的统一的事务处理过程。这个通用的事务处理过程，包含了处理事务配置属性，以及与线程绑定完成事务处理的过程，Spring 通过 TransactionInfo 和 TransactionStatus 两个数据对象，在事务处理过程中记录和传递相关执行场景。
- 底层的事务处理实现。对于底层的事务操作，Spring 委托给具体的事务处理器来完成，这些具体的事务处理器就是我们在 IoC 容器中，配置声明式事务处理的时候配置的 PlatformTransactionManager 的具体实现，比如像 DataSourceTransactionManager 和 HibernateTransactionManager 等。对于这两个具体的事务处理器的实现原理，也是我们本章分析的内容。

6.2.1 事务处理拦截器的配置

和以前的思路一样，我们从声明式事务处理的基本使用入手，来了解它的基本实现原理。大家都已经很熟悉了，在使用声明式事务处理的时候，需要在 IoC 容器中配置 TransactionProxyFactoryBean，这是一个 FactoryBean，对于 FactoryBean 这个在 Spring 中使用得很多的工厂 Bean，大家一定不会陌生，因为看到 FactoryBean 会让我们立刻想起它的 getObject 方法。我们看看在 TransactionProxyFactoryBean 中，是如何通过 AOP 功能来完成事务管理配置的，如代码清单 6-1 所示。从代码清单中，可以看到 Spring 为声明式事务处理的实现做的一些准备工作：这些准备工作，包括为 AOP 配置基础设施，这些基础设施包括设置拦截器 TransactionInterceptor、通知器 DefaultPointcutAdvisor 或 TransactionAttributeSourceAdvisor。同时，在 TransactionProxyFactoryBean 的实现中，还可以看到注入进来的 PlatformTransactionManager 和事务处理属性 TransactionAttribute 等。

代码清单 6-1 TransactionProxyFactoryBean

```
public class TransactionProxyFactoryBean extends AbstractSingletonProxyFactoryBean
        implements BeanFactoryAware {
    /**
    * 这个拦截器TransactionInterceptor通过AOP发挥作用，通过这个拦截器实现，Spring
    * 封装了事务处理实现，关于它的具体实现，我们下面会详细的分析。
    */
    private final TransactionInterceptor transactionInterceptor = new Transaction
    Interceptor();
```

Download at http://www.pin5i.com/

```
    private Pointcut pointcut;
    //通过依赖注入的 PlatformTransactionManager。
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionInterceptor.setTransactionManager(transactionManager);
    }
    //通过依赖注入的事务属性，以 Properties 的形式出现。
    //把在 BeanDefinition 中读到的事务管理的属性信息，注入到 TransactionInterceptor 中去。
    public void setTransactionAttributes(Properties transactionAttributes) {
    this.transactionInterceptor.setTransactionAttributes(transactionAttributes);
    }
    public void setTransactionAttributeSource(TransactionAttributeSource
    transaction AttributeSource) {
    this.transactionInterceptor.setTransactionAttributeSource
    (transactionAttribu teSource);
    }
    public void setPointcut(Pointcut pointcut) {
        this.pointcut = pointcut;
    }
    public void setBeanFactory(BeanFactory beanFactory) {
        this.transactionInterceptor.setBeanFactory(beanFactory);
    }
    //这里创建 Spring AOP 对事务处理的 Advisor。
    protected Object createMainInterceptor() {
        this.transactionInterceptor.afterPropertiesSet();
        if (this.pointcut != null) {
            //这里使用默认的通知器 DefaultPointcutAdvisor，并为通知器配置事务处理拦截器。
            return new DefaultPointcutAdvisor(this.pointcut, this.transactionInterceptor);
        }
        else {
            /**
            * 如果没有配置 pointcut 的话，使用 TransactionAttributeSourceAdvisor 作为通知器，
            * 并为通知器设置 TransactionInterceptor 作为拦截器。
            */
            return new TransactionAttributeSourceAdvisor(this.transactionInterceptor);
        }
    }
}
```

在以上代码实现中，可以看到 AOP 配置的完成，一个值得注意的问题是，Spring 的 TransactionInterceptor 配置是在什么时候启动的，从而成为 Advisor 通知器中的一部分呢？要了解这一点，从对 createMainInterceptor 方法的调用分析中可以看到，这个 createMainInterceptor 方法在 IoC 容器完成 Bean 的依赖注入时，通过 initializeBean 方法被调用，关于具体的调用过程，如图 6-1 所示。

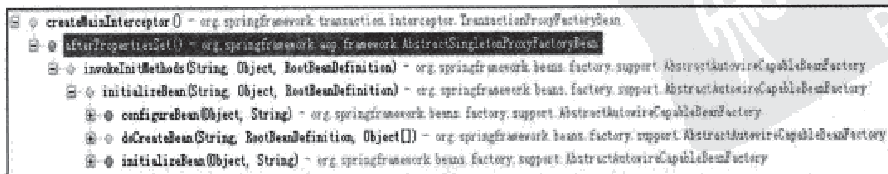


图 6-1 调用 createMainInterceptor 的过程

我们在 TransactionProxyFactoryBean 中看到的 afterPropertiesSet 方法，是

Download at http://www.pin5i.com/

Spring 事务处理完成 AOP 配置的地方，这个 afterPropertiesSet 方法的功能实现如代码清单 6-2 所示。在代码清单中可以看到，在建立 TransactionProxyFactoryBean 的事务处理拦截器的时候，首先需要对 ProxyFactoryBean 的目标 Bean 设置进行检查，如果这个目标 Bean 的配置是正确的，就会通过创建一个 ProxyFactory 对象，从而实现 AOP 的使用，在 afterPropertiesSet 的方法实现中，可以看到为 ProxyFactory 生成代理对象、配置通知器、设置代理接口方法等设置。

代码清单 6-2 TransactionProxyFactoryBean 的 afterPropertiesSet

```
public void afterPropertiesSet() {
    //必须配置 target 的属性，同时需要是一个 bean reference。
    if (this.target == null) {
        throw new IllegalArgumentException("Property 'target' is required");
    }
    if (this.target instanceof String) {
        throw new IllegalArgumentException("'target' needs to be a bean reference,
        not a bean name as value");
    }
    if (this.proxyClassLoader == null) {
        this.proxyClassLoader = ClassUtils.getDefaultClassLoader();
    }
    //TransactionProxyFactoryBean 使用 ProxyFactory 完成 AOP 的基本功能。
    //ProxyFactory 提供 Proxy 对象，并将 TransactionInterceptor 设置为 target 方法调用的拦截器。
    ProxyFactory proxyFactory = new ProxyFactory();
    if (this.preInterceptors != null) {
        for (Object interceptor : this.preInterceptors) {
proxyFactory.addAdvisor(this.advisorAdapterRegistry.wrap(interceptor));
        }
    }
    //这里是 Spring 加入通知器的地方。
    //可以加入 DefaultPointcutAdvisor 和 TransactionAttributeSourceAdvisor。
    //调用 TransactionProxyFactoryBean 的 createMainInterceptor 方法来生成 Advisors。
    /**
    * 在 ProxyFactory 的基类 AdvisedSupport 中，维护了一个用来持有 advice 的 LinkedList，
    * 通过对这个 LinkedList 的元素执行添加、修改、删除的操作，用来管理配置给 ProxyFactory 的通知器。
    */
proxyFactory.addAdvisor(this.advisorAdapterRegistry.wrap(createMainInterceptor()));
    if (this.postInterceptors != null) {
        for (Object interceptor : this.postInterceptors) {
proxyFactory.addAdvisor(this.advisorAdapterRegistry.wrap(interceptor));
        }
    }
    proxyFactory.copyFrom(this);
    //这里创建 AOP 的目标源，与在其他地方使用 ProxyFactory 的地方没有什么差别。
    TargetSource targetSource = createTargetSource(this.target);
    proxyFactory.setTargetSource(targetSource);
    if (this.proxyInterfaces != null) {
        proxyFactory.setInterfaces(this.proxyInterfaces);
    }
    else if (!isProxyTargetClass()) {
        // Rely on AOP infrastructure to tell us what interfaces to proxy.
        proxyFactory.setInterfaces(
ClassUtils.getAllInterfacesForClass(targetSource.getTargetClass(), this.proxy
    ClassLoader));
    }
```

Download at http://www.pin5i.com/

```
        //这里设置代理对象。
        this.proxy = proxyFactory.getProxy(this.proxyClassLoader);
    }
    /**
    * ProxyFactory是如何生成Proxy对象的，可以到ProxyFactory的实现中去了解一下，
    * 在AOP实现原理的分析中已经分析过了。
    */
    public <T> T  getProxy(ClassLoader classLoader) {
        return (T) createAopProxy().getProxy(classLoader);
    }
    /**
    * 调用createAopProxy()方法来生成Proxy对象，这个方法在ProxyFactory的基类
    * ProxyCreatorSupport中实现:
    */
    protected final synchronized AopProxy createAopProxy() {
        if (!this.active) {
            activate();
        }//这里使用DefaultAopProxyFactory来创建AopProxy。
        /**
         * 因为这个ProxyFactory类本身就是ProxyConfig的子类，所以这里创建AOP Proxy的
         * 过程和一般AOP Proxy的创建过程是一样的。
         */
        return getAopProxyFactory().createAopProxy(this);
    }
```

DefaultAopProxyFactory 创建 AOPProxy 的过程，在第 3 章 AOP 的实现原理的部分已经分析过了，在这里就不再重复了。可以看到，通过以上的这一系列步骤，作为 Spring 为实现事务处理而设计的拦截器 TransctionInterceptor，已经设置到 ProxyFactory 生成的 AOP 代理对象中去了，这里的 TransctionInterceptor 是作为 AOP Advice 的来实现它的功能的。在 IoC 容器中，配置的其他与事务处理有关的属性，比如我们熟悉的 transactionManager 和事务处理的属性，也同样都会被设置到已经定义好的 TransactionInterceptor 中去。对于这些属性配置，在 TransactionInterceptor 对事务方法进行拦截的时候会起作用。在 AOP 配置完成以后，我们看到在 Spring 声明式事务处理实现中的一些重要的类已经悄然登场，比如 TransactionAttributeSourceAdvisor 和 TransactionInterceptor。正是这些类通过 AOP 封装了 Spring 对事务处理的基本实现，下面让我们到这些类的实现中去看看它们完成的工作。

6.2.2 事务处理配置的读入

在 AOP 配置完成的基础上，下面让我们回到 TransactionProxyFactoryBean 中去，以 TransactionAttributeSourceAdvisor 的实现为入口，了解具体的事务属性配置是如何被读入的，如代码清单 6-3 所示。

代码清单 6-3 TransactionInterceptor 的实现

```
//与其他Advisor一样，同样需要定义AOP中用到的Interceptor和Pointcut。
//Interceptor使用的是我们已经见过面的拦截器：TransactionInterceptor。
private TransactionInterceptor transactionInterceptor;

//对于pointcut，定义一个内部类TransactionAttributeSourcePointcut private final.
```

Download at http://www.pin5i.com/

```
    TransactionAttributeSourcePointcut pointcut = new TransactionAttributeSourcePointcut() {
        @Override
        /**
        * 这里调用 transactionInterceptor 来得到事务的配置属性，在对 Proxy 的方法
        * 进行匹配调用时，会使用到这些配置属性。
        */
        protected TransactionAttributeSource getTransactionAttributeSource() {
            return (transactionInterceptor !=
    null ? transactionInterceptor.getTransaction AttributeSource() : null);
        }
    }
```

在声明式事务处理中，是通过对目标对象的方法调用进行拦截实现的，这个拦截通过 AOP 发挥作用。对于拦截的启动，首先需要对方法调用是否需要拦截进行判断，而判断的依据是那些在 TransactionProxyFactoryBean 中为目标对象设置的事务属性。也就是说，需要判断当前的目标方法调用是不是一个配置好的，需要进行事务处理的方法调用。具体来说，这个匹配判断在 TransactionAttributeSourcePointcut 中完成，它的实现如代码清单 6-4 所示。在代码清单中，我们可以看到一个在 AOP 的 Pointcut 类中，我们已经很熟悉的 matches 方法，这个方法的实现原理在第 3 章 AOP 实现中已经做过分析，这里就不重复了。在这个为事务处理服务的 TransactionAttributeSourcePointcut 的 matches 方法实现中，它首先把事务方法的属性配置读取到 TransactionAttributeSource 对象中，有了这些事务处理的配置以后，会根据当前方法调用的 Method 对象和目标对象，对是否需要启动事务处理拦截器进行一个判断。

代码清单 6-4 TransactionAttributeSourcePointcut 的 matches

```
public boolean matches(Method method, Class targetClass) {
    TransactionAttributeSource tas = getTransactionAttributeSource();
    return (tas == null || tas.getTransactionAttribute(method, targetClass) != null);
}
```

在 Pointcut 的 matches 判断过程中，会用到 transactionAttributeSource 对象，这个对象，是在对 TransactionInterceptor 进行依赖注入时就配置好了的。它的设置是在 TransactionInterceptor 的基类 TransactionAspectSupport 中完成的，配置的是一个 NameMatchTransactionAttributeSource 对象，这个配置过程如代码清单 6-5 所示。

代码清单 6-5 配置 transactionAttributeSource

```
//配置 transactionAttributeSource。
/**
* 是一个 NameMatchTransactionAttributeSource 对象，并把在 IoC 容器中设置的事务
* 处理属性配置到这个 transactionAttributeSource 中。
*/
public void setTransactionAttributes(Properties transactionAttributes) {
    NameMatchTransactionAttributeSource tas = new NameMatchTransactionAttribute
  Source();
    tas.setProperties(transactionAttributes);
    this. transactionAttributeSource = tas;
}
```

在以上的代码实现中，NameMatchTransactionAttributeSource 作为 Transaction AttributeSource 的具体实现，是实际完成事务处理属性读入和匹配的地方。对于 NameMatchTransactionAttributeSource 是怎样实现事务处理属性的读入和匹配的，如代

Download at http://www.pin5i.com/

码清单 6-6 所示。我们看到，在对事务属性 TransactionAttributes 的设置中，会从事务处理属性配置中读取事务方法名和配置属性，在得到配置的事务方法名和属性以后，会把它们作为键值对加入到一个 nameMap 中。

在应用调用目标方法的时候，因为这个目标方法已经被 TransactionProxyFactoryBean 代理，所以 TransactionProxyFactoryBean 需要判断这个调用方法是否是事务方法。这个判断的实现，是通过在 NameMatchTransactionAttributeSource 中，能否为这个调用方法返回事务属性来完成的。具体的实现过程是这样的：首先，使用调用方法名作为索引在 nameMap 中去查找相应的事务处理属性值，如果能够找到，那么就说明该调用方法和事务方法是直接对应的；如果找不到，那么就会遍历整个 nameMap，对保存在其中的每一个方法名，使用 PatternMatchUtils 进行命名模式上的匹配。这里使用 PatternMatchUtils 进行匹配，是因为在设置事务方法的时候，就像我们常见的那样，可以不需要为事务方法设置一个完整的方法名，而是可以通过设置方法名的命名模式来完成，比如可以通过像对通配符“*”的使用等，所以，如果直接通过方法名没能够匹配上，而通过方法名的命名模式能够匹配上，这个方法也是需要进行事务处理的方法。相对应的，它所配置的事务处理属性也会从 nameMap 中取出来，从而触发事务处理拦截器的拦截。

代码清单 6-6 NameMatchTransactionAttributeSource 的实现

```
//设置配置的事务方法。
public void setProperties(Properties transactionAttributes) {
    TransactionAttributeEditor tae = new TransactionAttributeEditor();
    Enumeration propNames = transactionAttributes.propertyNames();
    while (propNames.hasMoreElements()) {
        String methodName = (String) propNames.nextElement();
        String value = transactionAttributes.getProperty(methodName);
        tae.setAsText(value);
        TransactionAttribute attr = (TransactionAttribute) tae.getValue();
        addTransactionalMethod(methodName, attr);
    }
}

public void addTransactionalMethod(String methodName, TransactionAttribute attr) {
    if (logger.isDebugEnabled()) {
        logger.debug("Adding transactional method[" + methodName + "]
    with attribute [" + attr + "]");
    }
    this.nameMap.put(methodName, attr);
}
//判断对调用的方法是不是事务方法，如果是事务方法，那么取出相应的事务配置属性。
public TransactionAttribute getTransactionAttribute(Method method, Class
targetClass) {
    // look for direct name match.
    //判断当前目标调用的方法与配置的事务方法，是否直接匹配。
    String methodName = method.getName();
    TransactionAttribute attr = this.nameMap.get(methodName);
    //如果不能直接匹配，就通过调用 PatternMatchUtils 来进行匹配判断。
    if (attr == null) {
        // Look for most specific name match.
        String bestNameMatch = null;
        for (String mappedName : this.nameMap.keySet()) {
            if (isMatch(methodName, mappedName) &&
```

Download at http://www.pin5i.com/

```
                    (bestNameMatch= =null||bestNameMatch.length()<=mappedName.length())) {
                    attr = this.nameMap.get(mappedName);
                    bestNameMatch = mappedName;
                }
            }
        }
        return attr;
    }
    //事务方法的匹配判断，详细的匹配过程在 PatternMatchUtils 中实现。
    protected boolean isMatch(String methodName, String mappedName) {
        return PatternMatchUtils.simpleMatch(mappedName, methodName);
    }
```

对 getTransactionAttribute 的典型调用过程如图 6-2 所示。

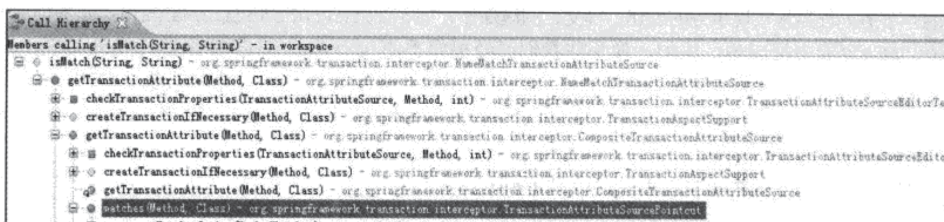


图 6-2 getTransactionAttribute 的典型调用过程

通过以上过程，可以得到与目标对象调用方法相关的 TransactionAttribute 对象，在这个对象中，封装了事务处理的配置。具体来说，在前面的匹配过程中，如果匹配返回的结果是 null 对象，那么说明当前的调用方法不是一个事务方法，不需要纳入 Spring 统一的事务管理中，因为它并没有配置在 TransactionProxyFactoryBean 的事务处理设置中。如果返回的 TransactionAttribute 对象不是 null，那么这个返回的 TransactionAttribute 对象就已经包含了对事务方法的配置信息，对应于这个事务方法的具体事务配置也已经读入到 TransactionAttribute 对象中来了，为 TransactionInterceptor 做好了对调用的目标方法添加事务处理的准备。

6.3 事务处理拦截器的实现

在完成以上的准备工作以后，经过 TransactionProxyFactoryBean 的 AOP 包装，此时如果对目标对象进行方法调用，其调用起作用的对象实际上是一个 Proxy 代理对象，对目标对象方法的调用，不会直接作用在 TransactionProxyFactoryBean 设置的目标对象上，而会被设置的事务处理拦截器拦截。而对于在 TransactionProxyFactoryBean 的 AOP 实现中，获取 Proxy 对象的过程，这个过程并不复杂，TransactionProxyFactoryBean 作为一个 FactoryBean，对这个 Bean 的对象的引用，是通过调用 TransactionProxyFactoryBean 的 getObject 方法来得到的。我们对这个方法已经很熟悉了，如代码清单 6-7 所示。

代码清单 6-7 TransactionProxyFactoryBean 的 getObject

```
//返回的是 Proxy，它是 ProxyFactory 生成的 AOP 代理，已经封装了对事务处理的拦截器配置。
```

Download at http://www.pin5i.com/

```
    public Object getObject() {
        if (this.proxy == null) {
            throw new FactoryBeanNotInitializedException();
        }
        return this.proxy;
    }
```

关于如何对 AOP 代理起作用，在第 3 章中已经分析过，回顾一下，大家会注意到一个重要的 invoke 方法，这个 invoke 方法是 Proxy 代理对象的回调方法，在 Proxy 对象的代理方法被调用时触发这个回调。在事务处理拦截器 TransactionInterceptor 中，invoke 方法的实现如代码清单 6-8 所示。可以看到，首先获得调用方法的事务处理配置，这个取得事务处理配置的过程已经在前面分析过了。在得到事务处理配置以后，会取得配置的 PlatformTransactionManager，由这个事务处理器来实现事务的创建、提交、回滚操作。PlatformTransactionManager 事务处理器是在 IoC 容器中配置的，比如，像那些我们已经很熟悉的 DataSourceTransactionManager 和 HibernateTransactionManager。有了这一系列的具体事务处理器的配置，在 Spring 事务处理模块的统一管理下，由这些具体的事务处理器来完成事务的创建、提交、回滚等底层的事务操作。

代码清单 6-8　TransactionInterceptor 的 invoke 回调

```
    public Object invoke(final MethodInvocation invocation) throws Throwable {
        // Work out the target class: may be <code>null</code>.
        /**
        * The TransactionAttributeSource should be passed the target class
        * as well as the method, which may be from an interface.
        */
        Class targetClass = (invocation.getThis()!=
        null ? invocation.getThis().getClass():null);
        // If the transaction attribute is null, the method is non-transactional.
        // 读取事务的属性配置，通过TransactionAttributeSource对象取得。
        final TransactionAttribute txAttr =
    getTransactionAttributeSource().getTransactionAttribute(invocation.getMethod(),
        targetClass);
        //根据TransactionProxyFactoryBean的配置信息，取得具体的事务处理器。
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        final String joinpointIdentification = methodIdentification(invocation.getMethod());
        /**
        * 区分不同类型的PlatformTransactionManager，因为它们的调用方式不同。
        * 对CallbackPreferringPlatformTransactionManager来说，需要回调函数来实现事务的创建和
        * 提交。对非CallbackPreferringPlatformTransactionManager来说，不需要通过回调函数来
        * 实现
        * 事务的创建和提交。像DataSourceTransactionManager就不是CallbackPreferring
        * PlatformTransactionManager，不需要通过回调的方式来使用。
        */
        if (txAttr==null||!(tm instanceof CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with getTransaction and commit/rollback
        calls.
            /**
            * 创建事务，同时把创建事务过程中得到的信息放到TransactionInfo中去，
            * TransactionInfo是保存当前事务状态的对象。
            */
            TransactionInfo txInfo=createTransactionIfNecessary(tm,txAttr, joinpoint
        Identification);
            Object retVal = null;
```

Download at http://www.pin5i.com/

```
        try {
            // This is an around advice: Invoke the next interceptor in the chain.
            // This will normally result in a target object being invoked.
            // 这里的调用使得处理沿着拦截器链进行，使得最后目标对象的方法得到调用。
            retVal = invocation.proceed();
        }
        catch (Throwable ex) {
            // target invocation exception
            // 如果在事务处理方法调用中出现了异常，事务处理如何进行，要根据具体的情况考虑回滚或提交。
            completeTransactionAfterThrowing(txInfo, ex);
            throw ex;
        }
        finally {//把与线程绑定的TransactionInfo设置为oldTransationInfo。
            cleanupTransactionInfo(txInfo);
        }
        //通过事务处理器来对事务进行提交。
        commitTransactionAfterReturning(txInfo);
        return retVal;
    }
    else {
        /**
        * It's a CallbackPreferringPlatformTransactionManager: pass a
        * TransactionCallback in.
        */
        // 采用回调的方法来使用事务处理器。
        try {
            Object result=((CallbackPreferringPlatformTransactionManager) tm).execute
    (txAttr, new TransactionCallback<Object>() {
            public Object doInTransaction(TransactionStatus status) {
        TransactionInfo txInfo = prepareTransactionInfo(tm, txAttr, joinpoint
        Identification, status);
                            try {
                            return invocation.proceed();
                            }
                            catch (Throwable ex) {
                                if (txAttr.rollbackOn(ex)) {
                        // A RuntimeException: will lead to a rollback.
                                    if (ex instanceof RuntimeException) {
                                        throw (RuntimeException) ex;
                                    }
                                    else {
                    throw new ThrowableHolderException(ex);
                                    }
                                }
                                else {
                            // A normal return value: will lead to a commit.
                                    return new ThrowableHolder(ex);
                                }
                            }
                            finally {
                                cleanupTransactionInfo(txInfo);
                            }
                        }
                    });
            // Check result: It might indicate a Throwable to rethrow.
            if (result instanceof ThrowableHolder) {

                throw ((ThrowableHolder) result).getThrowable();
```

Download at http://www.pin5i.com/

```
                    }
                    else {
                        return result;
                    }
                }
                catch (ThrowableHolderException ex) {
                    throw ex.getCause();
                }
            }
        }
```

在这个 invoke 方法的实现中，可以看到整个事务处理在 AOP 拦截器中实现的全过程。同时，它也是 Spring 采用 AOP 封装事务处理和实现声明式事务处理的核心部分。这部分实现是一个桥梁，它胶合了具体的事务处理和 Spring AOP 框架，可以看成是一个 Spring AOP 应用，在这个桥梁搭建完成以后，Spring 事务处理的实现就开始慢慢地浮出水面了。

6.4 事务处理的实现

6.4.1 事务处理的编程式使用

声明式事务处理的即开即用特性为用户提供了很大的方便。与对 IoC 容器的使用相类似，Spring 的事务处理也可以通过编程的方式进行使用，了解这种使用方式，对我们理解 Spring 事务处理的实现是有很大帮助的，对于 Spring 事务处理的编程式使用，如代码清单 6-9 所示。

代码清单 6-9 事务处理的编程式使用

```
    TransactionDefinition td = new DefaultTransactionDefinition();
    TransactionStatus status = transactionManager.getTransaction(td);
try{
    //这里是需要进行事务处理的方法调用。
    }catch (ApplicationException e) {
        transactionManager.rollback(status);
        throw e
    }
    transactionManager.commit(status);
```

在编程式使用事务处理中，使用 DefaultTransactionDefinition 对象来持有事务处理属性。同时，在创建事务的过程中得到一个 TransactionStatus 对象，然后通过直接调用 transactionManager 的 commit 和 rollback 方法来完成事务处理。在这个编程式使用事务管理的过程中，没有看到框架特性的使用，非常地简单和直接，很好地说明了事务管理的基本实现过程，以及在 Spring 事务处理实现中，涉及如 TransationStatus、TransactionManager 等主要的类，对这些类的使用与声明式事务处理的最终实现是一样的。

与编程式使用事务管理不同，在使用声明式事务处理的时候，因为涉及 Spring 框架对事务处理的统一管理，以及对并发事务和事务属性的处理，看到的是一个比较复杂的处理过程，但这个过程对使用声明式事务处理的应用来说基本上是不可见的，而是由 Spring 框架替我们完成的。有了这些背景铺垫和前面对 AOP 封装事务处理的了解，下面我们就来看看 Spring 是如何为提供声明式事务处理的，Spring 在这个相对较为复杂的过程中封装了什么。这层封装包括了在事务处理中，事务的创建、提交和回滚这些比较核心的操作。下面将对相关内容进行详细分析。

Download at http://www.pin5i.com/

6.4.2 事务的创建

作为声明式事务处理实现的起点，我们需要注意 TransactionInterceptor 拦截器的 invoke 回调中使用的 createTransactionIfNecessary 方法。这个方法是在 TransactionInterceptor 的基类 TransactionAspectSupport 中实现的。为了解这个方法的实现，我们先到 TransactionInterceptor 的基类实现 TransactionAspectSupport 中去看看，并以这个方法的实现为入口，了解 Spring 是如何根据当前的事务状态和事务属性配置，来完成事务创建的。这个 TransactionAspectSupport 的 createTransactionIfNecessary 方法作为事务创建的入口，如代码清单 6-10 所示。在 createTransactionIfNeccessary 方法调用中，可以看到两个重要的数据对象 TransactionStatus 和 TransactionInfo 的创建，这两个对象持有的数据是事务处理器对事务进行处理的主要依据，对它们的使用贯穿着整个事务处理的全过程。

代码清单 6-10 TransactionAspectSupport 的 createTransactionIfNecessary

```
protected TransactionInfo createTransactionIfNecessary(Method method, Class
    targetClass) {
    // If the transaction attribute is null, the method is non-transactional.
    // 首先读取事务方法调用的事务配置属性。
    TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute
    (method, targetClass);
    // 确定使用的 PlatformTransactionManager。
    PlatformTransactionManager tm = determineTransactionManager(txAttr);
    return createTransactionIfNecessary(tm, txAttr, methodIdentification(method));
}
protected TransactionInfo createTransactionIfNecessary(
        PlatformTransactionManager tm, TransactionAttribute txAttr, final String
    joinpointIdentification) {
    // If no name specified, apply method identification as transaction name.
    if (txAttr != null && txAttr.getName() == null) {
        txAttr = new DelegatingTransactionAttribute(txAttr) {
            @Override
            public String getName() {
                return joinpointIdentification;
            }
        };
    }
    //这个 TransactionStatus 封装了事务执行的状态信息。
    TransactionStatus status = null;
    if (txAttr != null) {
        if (tm != null) {
            //这里使用了我们定义好的事务方法的配置信息。
            /**
            * 事务创建由事务处理器来完成，同时返回 TransactionStatus 来记录
            * 当前的事务状态，包括已经创建的事务。
            */
            status = tm.getTransaction(txAttr);
        }
        else {

            if (logger.isDebugEnabled()) {
                logger.debug("Skipping transactional joinpoint [" + joinpoint
                Identification +
```

Download at http://www.pin5i.com/

```
                "] because no transaction manager has been configured");
            }
        }
    }
    //准备 TransactionInfo，它封装了事务处理的配置信息以及 TransactionStatus。
    return prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);
}
protected TransactionInfo prepareTransactionInfo(PlatformTransactionManager tm,
        TransactionAttribute txAttr, String joinpointIdentification, TransactionStatus
    status) {

    TransactionInfo txInfo = new TransactionInfo(tm, txAttr, joinpointIdentification);
    if (txAttr != null) {
        // We need a transaction for this method.
        if (logger.isTraceEnabled()) {
            logger.trace("Getting transaction for["+txInfo.getJoinpointIdentification()+"]");
        }
        // The transaction manager will flag an error if an incompatible tx already exists.
        /**
        * 这里为 TransactionInfo 设置 TransactionStatus，这个 TransactionStatus
        * 很重要，它持有管理事务处理需要的数据，比如，transaction 对象就是由
        * TransactionStatus 来持有的。
        */
        txInfo.newTransactionStatus(status);
    }
    else {
        /**
        * The TransactionInfo.hasTransaction() method will return
        * false. We created it only to preserve the integrity of
        * the ThreadLocal stack maintained in this class.
        */
        if (logger.isTraceEnabled())
            logger.trace("Don't need to create transaction for [" +
                    joinpointIdentification+"]: This method isn't transactional.");
    }
    /**
    * We always bind the TransactionInfo to the thread, even if we didn't create
    * a new transaction here. This guarantees that the TransactionInfo stack
    * will be managed correctly even if no transaction was created by this aspect.
    */
    /**
    * 这里把当前的 TransactionInfo 与线程绑定，同时在 TransactionInfo 中有一个变量
    * 来保存以前的 TransactionInfo，这样就持有了一连串与事务处理相关的 TransactionInfo。
    */
    // 虽然不一定需要创建新的事务，但是总会在请求事务时创建 TransactionInfo。
    txInfo.bindToThread();
    return txInfo;
}
```

在以上的处理过程完成之后，可以看到具体的事务创建就可以交给事务处理器来完成了。在事务的创建过程中已经为事务的管理做好了准备，包括记录事务处理状态以及绑定事务信息和线程等。下面我们到事务处理器中去了解一下更底层的事务创建过程，这个方法被代码清单 6-10 中的 `tm.getTransaction(txAttr)` 调用触发，生成一个 `TransactionStatus` 对象，封装了底层事务对象的创建。可以看到，在 `AbstractPlatformTransactionManager` 中，提供了创建事

Download at http://www.pin5i.com/

务的模板，这个模板会被具体的事务处理器所使用，如代码清单 6-11 所示。可以看到，AbstractPlatformTransactionManager 会根据事务属性配置和当前进程绑定的事务信息，对事务是否需要创建、怎样创建进行一些通用的处理，然后把事务创建的底层工作交给具体的事务处理器完成。尽管对于具体的事务处理器而言，它们完成事务创建的过程各不相同，但是对于不同的事务处理器来说，它们对事务属性和当前进程事务信息的处理都是相同的，这些相同的处理部分，就是在代码清单 6-11 中看到的，在 AbstractPlatformTransactionManager 的实现中完成的，这个实现过程是 Spring 提供统一事务处理的一个重要部分。

代码清单 6-11　AbstractPlatformTransactionManager 的 getTransaction

```
public final TransactionStatus getTransaction(TransactionDefinition definition)
     throws TransactionException {
     /**
     * 这个 doGetTransaction 是抽象函数，Transaction 对象的取得由具体的事务处理器实现，
     * 比如 DataSourceTransactionManager.
     */
     Object transaction = doGetTransaction();
     // Cache debug flag to avoid repeated checks.
     boolean debugEnabled = logger.isDebugEnabled();
     //如果没有设置事务属性，那么使用默认的事务属性 DefaultTransactionDefinition.
     /**
     * 关于这个 DefaultTransactionDefinition，我们在前面编程式使用事务处理的时候遇
     * 到过。这个 DefaultTransactionDefinition 的默认事务处理属性是：propagationBehavior =
     * PROPAGATION_REQUIRED;isolationLevel=ISOLATION_DEFAULT;timeout=
     * TIMEOUT_DEFAULT;readOnly = false;
     */
     if (definition == null) {
          // Use defaults if no transaction definition given.
          definition = new DefaultTransactionDefinition();
     }
     //检查当前线程是否存在事务，如果存在，需要根据在事务属性中定义的事务传播属性配置来处理事务的产生。
     if (isExistingTransaction(transaction)) {
     // Existing transaction found -> check propagation behavior to find out how
     to behave.
          //这里对当前线程中已经有事务存在的情况进行处理，结果封装在 TransactionStatus 中。
          return handleExistingTransaction(definition, transaction, debugEnabled);
     }
     // Check definition settings for new transaction.
     // 检查事务属性中 timeout 的设置是否合理。
     if (definition.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
          throw new InvalidTimeoutException("Invalid transaction timeout", definition.
     getTimeout());
     }
     // No existing transaction found -> check propagation behavior to find out
     how to proceed.
     //当前没有事务存在，这时需要根据事务属性设置来创建事务。
     //这里会看到对事务传播属性设置的处理，比如mandatory、required、required_new、nested等。
     //这里的处理对我们理解这些属性的使用是非常有帮助的。
     if (definition.getPropagationBehavior()== TransactionDefinition.PROPAGATION_MANDATORY) {
          throw new IllegalTransactionStateException(
                    "No existing transaction found for transaction marked with propagation
     'mandatory'");
     }
     else if (definition.getPropagationBehavior() == TransactionDefinition.
```

Download at http://www.pin5i.com/

```
        PROPAGATION_REQUIRED ||
                definition.getPropagationBehavior() == TransactionDefinition.
        PROPAGATION_REQUIRES_NEW ||
            definition.getPropagationBehavior() == TransactionDefinition.
        PROPAGATION_NESTED) {
            SuspendedResourcesHolder suspendedResources = suspend(null);
            if (debugEnabled) {
                logger.debug("Creating new transaction with name [" + definition.
        getName() + "]: " + definition);
            }
            try {
            /**
            * 这里是创建事务的调用，由具体的事务处理器完成，比如
            * HibernateTransactionManager 和 DataSourceTransactionManager 等。
            */
                doBegin(transaction, definition);
            }
            catch (RuntimeException ex) {
                resume(null, suspendedResources);
                throw ex;
            }
            catch (Error err) {
                resume(null, suspendedResources);
                throw err;
            }
            boolean newSynchronization=(getTransactionSynchronization()!=SYNCHRONIZATION_NEVER);
            /**
            * 返回 TransactionStatus 封装事务执行情况，默认情况下为
            * getTransactionSynchronization=SYNCHRONIZATION_ALWAYS,
            * 所以在这种情况下，newSynchronization 为 true。
            */
            return newTransactionStatus(
                    definition, transaction, true, newSynchronization, debugEnabled,
        suspendedResources);
        }
        else {
            // Create "empty" transaction: no actual transaction, but potentially
        synchronization.
            /**
            * TransationStatus 没有 transaction 对象，因为在 newTransactionStatus
            * 中对应于 transaction 的参数是 null。
            */
            boolean newSynchronization = (getTransactionSynchronization() ==
        SYNCHRONIZATION_ALWAYS);
            return newTransactionStatus(definition, null, true, newSynchronization,
        debugEnabled, null);
        }
    }
```

基于代码清单 6-11，可以看到 AbstractTransactionManager 提供的创建事务的实现模板，在这个模板的基础上，具体的事务处理器需要定义自己的实现完成底层的事务创建工作，比如，需要实现 isExistingTransaction 和 doBegin 方法。关于这些由具体事务处理器实现的方法，我们会在下面结合具体的事务处理器实现（比如 DataSourceTransactionManager 和 HibernateTransactionManager）进行分析。

Download at http://www.pin5i.com/

对事务创建的结果而言，会生成一个 TransactionStatus 对象，通过这个对象来保存事务处理需要的基本信息，这个对象与我们前面提到过的 TransactionInfo 对象联系在一起，TransactionStatus 是 TransactionInfo 的一个属性。然后会把 TransactionInfo 保存在 ThreadLocal 对象里，这样使得当前线程可以通过 ThreadLocal 对象取得 TransactionInfo，以及与这个事务对应的 TransactionStatus 对象，从而把事务的处理信息与调用事务方法的当前线程绑定起来。在 AbstractPlatformTransactionManager 创建事务的过程中可以看到 TransactionStatus 的创建，如代码清单 6-12 所示。

代码清单 6-12 AbstractPlatformTransactionManager 的 newTransactionStatus

```
protected DefaultTransactionStatus newTransactionStatus(
        TransactionDefinition definition, Object transaction,
    boolean new Transaction,
        boolean newSynchronization, boolean debug, Object suspendedResources) {
    //这里判断是不是新事务，如果是新事务，那么需要把事务属性存放到当前线程中。
    /**
    * TransactionSynchronizationManager 维护了一系列的 ThreadLocal 变量来保持事务属性
    * 比如并发事务隔离级别，是否有活跃的事务等。
    */
    boolean actualNewSynchronization = newSynchronization &&
            !TransactionSynchronizationManager.isSynchronizationActive();
    if (actualNewSynchronization) {
TransactionSynchronizationManager.setActualTransactionActive(transaction!=null);
    TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(
        (definition.getIsolationLevel()!=TransactionDefinition.ISOLATION_DEFAULT)?
            definition.getIsolationLevel() : null);
TransactionSynchronizationManager.setCurrentTransactionReadOnly(definition.isRea
    dOnly());
TransactionSynchronizationManager.setCurrentTransactionName(definition.getName());
        TransactionSynchronizationManager.initSynchronization();
    }//这里把结果记录在 DefaultTransactionStatus 中返回。
    return new DefaultTransactionStatus(
            transaction, newTransaction, actualNewSynchronization,
            definition.isReadOnly(), debug, suspendedResources);
}
```

新事务的创建是比较好理解的，这里需要根据事务属性配置进行创建。所谓创建，首先是把创建工作交给具体的事务处理器来完成，比如 DataSourceTransactionManager，把创建的事务对象在 TransactionStatus 中保存下来，然后把其他的事务属性和线程 ThreadLocal 变量进行绑定。

相对于创建全新事务的另一种情况是，在创建当前事务时，线程中已经有事务存在。这种情况同样需要处理，在声明式事务处理中，在当前线程调用事务方法的时候，就会考虑事务的创建处理，这个处理在方法 handleExistingTransaction 中完成，如代码清单 6-13 所示。这里对现有事务的处理，会涉及事务传播属性的具体处理，比如 PROPAGATION_NOT_SUPPORTED、PROPAGATION_REQUIRES_NEW 等。

代码清单 6-13 handleExistingTransaction 的实现

```
private TransactionStatus handleExistingTransaction(
        TransactionDefinition definition, Object transaction, boolean debugEnabled)
        throws TransactionException {
```

Download at http://www.pin5i.com/

```
    /**
    *如果当前线程已有事务存在，而当前事务的配置的传播属性设置是 never，那么抛出异常，
    *说明这种情况是有问题的，Spring 无法处理当前的事务创建。
    */
    if (definition.getPropagationBehavior()==TransactionDefinition.PROPAGATION_NEVER){
        throw new IllegalTransactionStateException(
                "Existing transaction found for transaction marked with propagation 'never'");
    }
    /**
    *如果当前事务的配置属性是 PROPAGATION_NOT_SUPPORTED，同时当前线程已经存在事务了，
    *那么将事务挂起。
    */
    if(definition.getPropagationBehavior()==TransactionDefinition.PROPAGATION_NOT_
    SUPPORTED) {
        if (debugEnabled) {
            logger.debug("Suspending current transaction");
        }
        Object suspendedResources = suspend(transaction);
        boolean newSynchronization=(getTransactionSynchronization()==
    SYNCHRONIZATION_ ALWAYS);
        /**
        *注意这里的参数，transaction 为 null，newTransaction 为 false，意味着事务方法非事
        *务地执行。同时挂起事务的信息记录也保存在 TransactionStatus 中，
        *这里包括了进程 ThreadLocal 对事务信息的记录。
        */
        return newTransactionStatus(
              definition,null,false,newSynchronization,debugEnabled,suspendedResources);
    }
    /**
    *如果当前事务的配置属性是 PROPAGATION_REQUIRES_NEW，创建新事务，同时把当前线程中存在的
    *事务挂起。与创建全新事务的过程相类似，区别在于，在创建全新事务时不用考虑已有事务的挂起，但
    *在这里，需要考虑已有事务的挂起处理。
    */
    if(definition.getPropagationBehavior()==TransactionDefinition. PROPAGATION_
    REQUIRES_NEW) {
        if (debugEnabled) {
            logger.debug("Suspending current transaction, creating new transaction
    with name [" + definition.getName() + "]");
        }
        SuspendedResourcesHolder suspendedResources = suspend(transaction);
        try {
            doBegin(transaction, definition);
        }
        catch (RuntimeException beginEx) {
            resumeAfterBeginException(transaction, suspendedResources, beginEx);
            throw beginEx;
        }
        catch (Error beginErr) {
            resumeAfterBeginException(transaction,suspendedResources,beginErr);
            throw beginErr;
        }
        boolean newSynchronization=(getTransactionSynchronization()!=SYNCHRONIZATION_NEVER);
//挂起事务的信息记录保存在 TransactionStatus 中，这里包括进程 ThreadLocal 对事务信息的记录。
        return newTransactionStatus(
                definition, transaction, true, newSynchronization, debugEnabled,
```

Download at http://www.pin5i.com/

```
suspendedResources);
}
//嵌套事务的创建。
if(definition.getPropagationBehavior()==TransactionDefinition.PROPAGATION_NESTED){
    if (!isNestedTransactionAllowed()) {
        throw new NestedTransactionNotSupportedException(
                "Transaction manager does not allow nested transactions by
                default - " +
                "specify'nestedTransactionAllowed'property with value'true'");
    }
    if (debugEnabled) {
        logger.debug("Creating nested transaction with name["+definition.
        getName()+"]");
    }
    if (useSavepointForNestedTransaction()) {
        /**
        *Create savepoint within existing Spring-managed transaction,
        *through the SavepointManager API implemented by TransactionStatus.
        *Usually uses JDBC 3.0 savepoints.Never activates Spring synchronization.
        */
        DefaultTransactionStatus status =
                newTransactionStatus(definition, transaction, false, false,
                debugEnabled, null);
        status.createAndHoldSavepoint();
        return status;
    }
    else {
        /**
        *Nested transaction through nested begin and commit/rollback calls.
        *Usually only for JTA: Spring synchronization might get activated here
        *in case of a pre-existing JTA transaction.
        */
        doBegin(transaction, definition);
        boolean newSynchronization = (getTransactionSynchronization() !=
        SYNCHRONIZATION_NEVER);
        return newTransactionStatus(definition,transaction,true,newSynchronization,
        debugEnabled,null);
    }
}

// Assumably PROPAGATION_SUPPORTS or PROPAGATION_REQUIRED.
if (debugEnabled) {
    logger.debug("Participating in existing transaction");
}/**
*这里判断在当前事务方法中的属性配置与已有事务的属性配置是否一致，如果不一致，那么不执行事
*务方法并抛出异常。
*/
if (isValidateExistingTransaction()) {
    if(definition.getIsolationLevel()!=TransactionDefinition.ISOLATION_DEFAULT){
        Integer currentIsolationLevel = TransactionSynchronizationManager.
getCurrentTransactionIsolationLevel();
        if (currentIsolationLevel==null||currentIsolationLevel!=
            definition. getIsolationLevel()) {
            Constants isoConstants=DefaultTransactionDefinition.constants;
            throw new IllegalTransactionStateException("Participating transaction
            with definition [" +
            definition + "] specifies isolation level which is
            incompatible with existing transaction: " +
            (currentIsolationLevel != null ?
```

Download at http://www.pin5i.com/

```
                            isoConstants.toCode(currentIsolationLevel,
DefaultTransactionDefinition.PREFIX_ISOLATION) :
                            "(unknown)"));
            }
        }
        if (!definition.isReadOnly()) {
            if(TransactionSynchronizationManager.isCurrentTransactionReadOnly()){
                throw new IllegalTransactionStateException("Participating transaction
                with definition [" +
                definition + "] is not marked as read-only but existing
                transaction is");
            }
        }
    }//返回 TransactionStatus，注意第三个参数 false 代表当前事务方法没有使用新的事务。
    boolean newSynchronization=
     (getTransactionSynchronization()!=SYNCHRONIZATION_NEVER);
    return newTransactionStatus(definition,transaction,false,
    newSynchronization, debugEnabled, null);
}
```

6.4.3　事务的挂起

事务的挂起涉及线程与事务处理信息的保存，可以看一下事务挂起的实现，如代码清单 6-14 所示。

代码清单 6-14　事务的挂起

```
//返回的 SuspendedResourcesHolder 会作为参数传给 TransactionStatus。
protected final SuspendedResourcesHolder suspend(Object transaction) throws
    TransactionException {
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
        List<TransactionSynchronization>suspendedSynchronizations=
        doSuspendSynchronization();
         try {
            Object suspendedResources = null;
            /**
            *把挂起事务的处理交给具体事务处理器去完成，如果具体的事务处理器不支持事务挂起，
            *那么默认是抛出异常 TransactionSuspensionNotSupportedException。
            */
            if (transaction != null) {
                suspendedResources = doSuspend(transaction);
            }
    //这里在线程中保存与事务处理有关的信息，并将线程里相关的 ThreadLocal 变量重置。
         String name=
         TransactionSynchronizationManager.getCurrentTransactionName();
         TransactionSynchronizationManager.setCurrentTransactionName(null);
         boolean readOnly =
         TransactionSynchronizationManager. isCurrent TransactionReadOnly();
         TransactionSynchronizationManager.setCurrentTransactionReadOnly(false);
         Integer isolationLevel=
         TransactionSynchronizationManager. getCurrent TransactionIsolationLevel();
         TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(null);
         boolean wasActive =
         TransactionSynchronizationManager. isActual TransactionActive();
         TransactionSynchronizationManager.setActualTransactionActive(false);
         return new SuspendedResourcesHolder(
                    suspendedResources,suspendedSynchronizations,name,readOnly,
         isolationLevel, wasActive);
```

Download at http://www.pin5i.com/

```
        }
        catch (RuntimeException ex) {
            // doSuspend failed - original transaction is still active...
            doResumeSynchronization(suspendedSynchronizations);
            throw ex;
        }
        catch (Error err) {
            // doSuspend failed - original transaction is still active...
            doResumeSynchronization(suspendedSynchronizations);
            throw err;
        }
    }
    else if (transaction != null) {
        // Transaction active but no synchronization active.
        Object suspendedResources = doSuspend(transaction);
        return new SuspendedResourcesHolder(suspendedResources);
    }
    else {
        // Neither transaction nor synchronization active.
        return null;
    }
}
```

有了这些基本的考虑，就可以完成声明式事务处理的创建了。对声明式事务处理而言，它能使事务处理应用的开发变得简单，但是简单的背后却蕴含着平台付出许多的努力，看到这里相信大家都已经有一个深刻的体会了。

6.4.4　事务的提交

下面我们来看看事务提交是如何实现的。有了前面的对事务创建的分析，下面来分析一下 Spring 中声明式事务处理的事务提交是如何完成的。事务提交的入口调用在 TransactionInteceptor 的 invoke 方法中实现，如以下的代码所示。

```
commitTransactionAfterReturning(txInfo);
```

在这个调用中，我们看到的 txInfo 是 TransactionInfo 对象，这个对象是在创建事务时生成的，同时，生成的 TransactionStatus 对象就包含在这个对象中。这个 commitTransactionAfterReturning 方法在 TransactionInteceptor 的实现部分是比较简单的，它通过直接调用事务处理器来完成事务提交，如代码清单 6-15 所示。

代码清单 6-15　TransactionInteceptor 的事务提交调用入口

```
protected void commitTransactionAfterReturning(TransactionInfo txInfo) {
    if (txInfo != null && txInfo.hasTransaction()) {
        if (logger.isTraceEnabled()) {
            logger.trace("Completing transaction for [" +
    txInfo. getJoinpointIdentification() + "]");
        }
        txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
    }
}
```

与前面分析事务的创建过程一样，我们这里需要到事务处理器中去看看事务是如何提交的。同样地，在 AbstractPlatformTransactionManager 中也有一个模板方法来支持具体的事务处理器对事务提交的实现。这个模板方法的实现与前面我们看到的 getTransaction 是很类似的，如代码清单 6-16 所示。

Download at http://www.pin5i.com/

代码清单 6-16 AbstractPlatformTransactionManager 的 commit

```
public final void commit(TransactionStatus status)throws TransactionException {
    //TransactionStatus 中标识事务已经结束。
    if (status.isCompleted()) {
        throw new IllegalTransactionStateException(
        "Transaction is already completed-do not call commit or rollback
        more than once per transaction");
    }
    //如果事务处理过程中发生了异常，调用回滚。
    DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
    if (defStatus.isLocalRollbackOnly()) {
        if (defStatus.isDebug()) {
            logger.debug("Transactional code has requested rollback");
        }//这里处理回滚。
        processRollback(defStatus);
        return;
    }
    if (!shouldCommitOnGlobalRollbackOnly()&&defStatus.isGlobalRollbackOnly()){
        if (defStatus.isDebug()) {
            logger.debug("Global transaction is marked as rollback-only but
            transactional code requested commit");
        }
        processRollback(defStatus);
        //Throw UnexpectedPollbackException only at outermost transaction boundary
        // or if explicitly asked to.
        if (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {
            throw new UnexpectedRollbackException(
            "Transaction rolled back because it has been marked as
            rollback-only");
        }
        return;
    }
    //处理提交的入口。
    processCommit(defStatus);
}

private void processCommit(DefaultTransactionStatus status) throws TransactionException {
    try {
        boolean beforeCompletionInvoked = false;
        try {
            //事务提交的准备工作由具体的事务处理器来完成。
            prepareForCommit(status);
            triggerBeforeCommit(status);
            triggerBeforeCompletion(status);
            beforeCompletionInvoked = true;
            boolean globalRollbackOnly = false;
            if (status.isNewTransaction()||isFailEarlyOnGlobalRollbackOnly()) {
                globalRollbackOnly = status.isGlobalRollbackOnly();
            }//这里是嵌套事务的处理。
            if (status.hasSavepoint()) {
                if (status.isDebug()) {
                    logger.debug("Releasing transaction savepoint");
                }
                status.releaseHeldSavepoint();
            }
/**
*下面对根据当前线程中保存的事务状态进行处理，如果当前的事务是一个新事务，调用具体事务处理器的完
```

Download at http://www.pin5i.com/

```
*成提交，如果当前所持有的事务不是一个新事务，则不提交，提交由已经存在的事务来完成。
*/
            else if (status.isNewTransaction()) {
                if (status.isDebug()) {
                    logger.debug("Initiating transaction commit");
                }//具体的事务提交由具体的事务处理器来完成。
                doCommit(status);
            }
            /**
            *Throw UnexpectedRollbackException if we have a global rollback-only
            *marker but still didn't get a corresponding exception from commit.
            */
            if (globalRollbackOnly) {
                throw new UnexpectedRollbackException(
                "Transaction silently rolled back because it has been marked
                as rollback-only");
            }
        }
        catch (UnexpectedRollbackException ex) {
            // Can only be caused by doCommit.
            triggerAfterCompletion(status, TransactionSynchronization.STATUS_
            ROLLED_BACK);
            throw ex;
        }
        catch (TransactionException ex) {
            // Can only be caused by doCommit.
            if (isRollbackOnCommitFailure()) {
                doRollbackOnCommitException(status, ex);
            }
            else {
                triggerAfterCompletion(status,TransactionSynchronization. STATUS_
                UNKNOWN);
            }
            throw ex;
        }
        catch (RuntimeException ex) {
            if (!beforeCompletionInvoked) {
                triggerBeforeCompletion(status);
            }
            doRollbackOnCommitException(status, ex);
            throw ex;
        }
        catch (Error err) {
            if (!beforeCompletionInvoked) {
                triggerBeforeCompletion(status);
            }
            doRollbackOnCommitException(status, err);
            throw err;
        }

        /**
        *Trigger afterCommit callbacks, with an exception thrown there
        *propagated to callers but the transaction still considered as committed.
        */
        try {
            triggerAfterCommit(status);
        }
        finally {
            triggerAfterCompletion(status, TransactionSynchronization.STATUS_
```

Download at http://www.pin5i.com/

```
COMMITTED);
            }

        }
        finally {
            cleanupAfterCompletion(status);
        }
    }
```

可以看到，事务提交的准备工作和提交的完成，都是由具体的事务处理器来实现的。当然，对这些事务提交的处理，需要通过对 TransactionStatus 保存的事务处理的相关状态进行判断。提交过程涉及 AbstractPlatformTransactionManager 中的 doCommit 和 prepareForCommit 方法，它们都是抽象方法，实现都在具体的事务处理器中完成，下面我们会在对具体事务处理器的实现原理的分析中，看到对这些实现方法进行的具体分析。

6.4.5 事务的回滚

除了事务的创建、挂起和提交，我们来看一看事务的回滚是如何完成的。回滚处理和处理事务提交非常相似，如代码清单 6-17 所示。

代码清单 6-17 AbstractPlatformTransactionManager 的 processRollback

```
private void processRollback(DefaultTransactionStatus status) {
    try {
        try {
            triggerBeforeCompletion(status);
            //嵌套事务的回滚处理。
            if (status.hasSavepoint()) {
                if (status.isDebug()) {
                    logger.debug("Rolling back transaction to savepoint");
                }
                status.rollbackToHeldSavepoint();
            }//当前事务调用方法中新建事务的回滚处理。
            else if (status.isNewTransaction()) {
                if (status.isDebug()) {
                    logger.debug("Initiating transaction rollback");
                }//这个 doRollback 处理是由具体的事务处理器来完成的。
                doRollback(status);
            }//如果在当前事务调用方法中没有新建事务的回滚处理。
            else if (status.hasTransaction()) {
                if(status.isLocalRollbackOnly()||isGlobalRollbackOnParticipation-
    Failure()){
                    if (status.isDebug()) {
                        logger.debug(
                        "Participating transaction failed - marking existing
                        transaction as rollback-only");
                    }
                    doSetRollbackOnly(status);
                }//由线程中的前一个事务来处理回滚，这里不执行任何操作。
                else {
                    if (status.isDebug()) {
                        logger.debug(
                        "Participating transaction failed - letting transaction
                        originator decide on rollback");
                    }
```

Download at http://www.pin5i.com/

```
                }
            }
            else {
                logger.debug("Should roll back transaction but cannot - no
                transaction available");
            }
        }
        catch (RuntimeException ex) {
            triggerAfterCompletion(status,TransactionSynchronization.STATUS_UNKNOWN);
            throw ex;
        }
        catch (Error err) {
            triggerAfterCompletion(status,TransactionSynchronization.STATUS_UNKNOWN);
            throw err;
        }
        triggerAfterCompletion(status,TransactionSynchronization.STATUS_ROLLED_BACK);
    }
    finally {
        cleanupAfterCompletion(status);
    }
}
```

以上对事务的创建、挂起、提交、回滚的实现原理进行了分析，希望能够为读者理清一条基本的线索。这些过程的实现都比较复杂，一方面体现在这些处理中，会涉及很多事务属性的处理；另一方面会涉及事务处理过程中状态的设置。同时在事务处理的过程中，有许多处理也需要根据相应的状态来完成。这样看来，在事务处理实现的基本过程中，就会产生许多事务处理的操作分支。由于篇幅原因，不能在这里为大家一一详细阐述，敬请谅解，有兴趣的读者可以在此基础上，根据自己的需要进一步分析和研究。但总的来说，在事务执行的实现过程中，作为执行控制的 TransactionInfo 和 TransactionStatus 对象是特别值得我们注意的，比如它们如何与线程进行绑定，如何记录事务的执行情况等。同时，如果大家在配置事务属性时有什么疑惑，不妨直接看看这些事务属性的处理过程，通过对这些实现原理的了解，可以极大地提高对这些事务处理属性使用的理解程度。

6.5 具体事务处理器的实现

下面，我们以 DataSourceTransactionManager 和 HibernateTransactionManager 两个常用的事务处理器为例，探讨一下在具体的事务处理器中，是如何实现事务创建、提交和回滚这些底层的事务处理操作的。

6.5.1 DataSourceTransactionManager 的实现

我们先看看 DataSourceTransactionManager，在这个事务处理器中它的实现直接与事务处理的底层实现相关，如代码清单 6-18 所示。可以看到，它是 AbstractPlatformTransactionManager 的子类，在 AbstractPlatformTransactionManager 中已经为事务实现设计好了一系列的模板方法，比如事务提交、回滚的处理等。在 DataSourceTransactionManager 中，可以看到对模板方法中的一些抽象方法的具体实现。例如，在 DataSourceTransactionManager 的

Download at http://www.pin5i.com/

doBegin 方法实现中，由它来负责事务的创建工作。具体来说，如果使用 DataSource 创建事务，最终是通过设置 Connection 的 AutoCommit 属性来为事务处理进行配置。在实现过程中，需要把数据库的 Connection 和当前的线程进行绑定。对于事务的提交和回滚，都是直接调用 Connection 的提交和回滚方法来完成的，在这个实现过程中，如何取得事务处理场景中的 Connection 对象，也是一个值得注意的地方。

代码清单 6-18 DataSourceTransactionManager

```
public class DataSourceTransactionManager extends AbstractPlatformTransactionManager
    implements ResourceTransactionManager, InitializingBean {
//这是注入的 DataSource。
private DataSource dataSource;
//这里是产生 Transaction 的地方，为 Transaction 的创建提供服务。
/**
*对数据库而言，是由 Connection 来完成事务工作的。这里把数据库的 Connection 对象放到一个
*ConnectionHolder 中，然后封装到一个 DataSourceTransactionObject 对象中，在这个封装过程中
*增加了许多为事务处理服务的控制数据。
*/
protected Object doGetTransaction() {
    DataSourceTransactionObject txObject = new DataSourceTransactionObject();
    txObject.setSavepointAllowed(isNestedTransactionAllowed());
    //获取与当前线程绑定的数据库 Connection，它是在第一个事务开始的地方与线程绑定的。
    ConnectionHolder conHolder =
       (ConnectionHolder)TransactionSynchronizationManager.getResource(this.dataSource);
    txObject.setConnectionHolder(conHolder, false);
    return txObject;
}

//判断是否已经存在事务，由 ConnectionHolder 的 isTransactionActive 属性来控制。
protected boolean isExistingTransaction(Object transaction) {
    DataSourceTransactionObject txObject=(DataSourceTransactionObject) transaction;
    return(txObject.getConnectionHolder()!=null && txObject.getConnectionHolder().
    isTransactionActive());
}

//这里是处理事务开始的地方。
protected void doBegin(Object transaction, TransactionDefinition definition) {
    DataSourceTransactionObject txObject=(DataSourceTransactionObject) transaction;
    Connection con = null;

    try {
        if (txObject.getConnectionHolder() == null ||
                txObject.getConnectionHolder().isSynchronizedWithTransaction())
{
            Connection newCon = this.dataSource.getConnection();
            if (logger.isDebugEnabled()) {
               logger.debug("Acquired Connection["+newCon+"]for JDBC transaction");
            }
            txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
        }

        txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
        con = txObject.getConnectionHolder().getConnection();

        Integer previousIsolationLevel=DataSourceUtils.prepareConnectionForTransaction
    (con, definition);
```

Download at http://www.pin5i.com/

```
            txObject.setPreviousIsolationLevel(previousIsolationLevel);

            /**
            *Switch to manual commit if necessary.This is very expensive in some JDBC drivers,
            *so we don't want to do it unnecessarily (for example if we've explicitly
            *configured the connection pool to set it already).
            */
            // 这里是数据库Connnection完成事务处理的重要配置，需要把autoCommit属性关掉。
            if (con.getAutoCommit()) {
                txObject.setMustRestoreAutoCommit(true);
                if (logger.isDebugEnabled()) {
                    logger.debug("Switching JDBC Connection["+con+"]to manual commit");
                }
                con.setAutoCommit(false);
            }
            txObject.getConnectionHolder().setTransactionActive(true);

            int timeout = determineTimeout(definition);
            if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
                txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
            }

            // Bind the session holder to the thread.
            // 把当前的数据库Connection和线程绑定。
            if (txObject.isNewConnectionHolder()) {
                TransactionSynchronizationManager.bindResource (getDataSource(),
        txObject.getConnectionHolder());
            }
        }

        catch (SQLException ex) {
            DataSourceUtils.releaseConnection(con, this.dataSource);
            throw new CannotCreateTransactionException("Could not open JDBC Connection
        for transaction", ex);
        }
    }
    //事务的提交过程。
    protected void doCommit(DefaultTransactionStatus status) {
        //取得Connection以后，通过Connection进行提交。
        DataSourceTransactionObject txObject =
        (DataSourceTransactionObject) status.getTransaction();
        Connection con = txObject.getConnectionHolder().getConnection();
        if (status.isDebug()) {
            logger.debug("Committing JDBC transaction on Connection [" + con + "]");
        }
        try {
            con.commit();
        }
        catch (SQLException ex) {
           throw new TransactionSystemException("Could not commit JDBC transaction",ex);
        }
    }

    //事务的回滚过程，使用Connection的rollback方法。
    protected void doRollback(DefaultTransactionStatus status) {
        DataSourceTransactionObject txObject =
        (DataSourceTransactionObject) status.getTransaction();
        Connection con = txObject.getConnectionHolder().getConnection();
```

Download at http://www.pin5i.com/

```
        if (status.isDebug()) {
            logger.debug("Rolling back JDBC transaction on Connection["+con + "]");
        }
        try {
            con.rollback();
        }
        catch (SQLException ex) {
            throw new TransactionSystemException("Could not roll back JDBC transaction",ex);
        }
    }
```

上面我们看到了使用 DataSourceTransactionManager 实现事务创建、提交和回滚的过程，基本上与我们单独使用 Connection 实现事务处理是一样的，也是通过设置 autoCommit 属性，调用 Connection 的 commit 和 rollback 方法来完成的。看到这里，大家一定会感到非常熟悉了。而我们在声明式事务处理中，看到的那些事务处理属性，并不在 DataSourceTransactionManager 里完成，这和我们在前面看到的分析是一致的。

6.5.2 HibernateTransactionManager 的实现

了解 DataSourceTransactionManager 实现事务处理的方法以后，如果我们熟悉 Hibernate 事务的使用，HibernateTransactionManager 事务处理的实现也不难理解，其过程如代码清单 6-19 所示。和我们平时单独使用 Hibernate 一样，是通过对 Session 的管理来完成事务处理实现的，在代码清单中，可以看到通过获得 Hibernate 的 Session、Session 属性的配置以及通过 Session 得到 Hibernate 的 Transaction 对象来完成事务创建、提交和回滚的过程。

代码清单 6-19 HibernateTransactionManager 的实现

```
//这里创建 HibernateTransactionObject，它是设置 Session 以及 DataSource 对象的地方。
protected Object doGetTransaction() {
    HibernateTransactionObject txObject = new HibernateTransactionObject();
    //是否允许嵌套事务，在这里进行设置。
    txObject.setSavepointAllowed(isNestedTransactionAllowed());
    //从线程中取得 SessionHolder，它是在事务开始时与线程绑定的。
    //把取得的 SessionHolder 设置到 TransactionObject 里面去。
    SessionHolder sessionHolder =
            (SessionHolder) TransactionSynchronizationManager.getResource
    (getSessionFactory());
    if (sessionHolder != null) {
        if (logger.isDebugEnabled()) {
            logger.debug("Found thread-bound Session [" +
                    SessionFactoryUtils.toString(sessionHolder.getSession())+"]
                    for Hibernate transaction");
        }
        txObject.setSessionHolder(sessionHolder);
    }
    else if (this.hibernateManagedSession) {
        try {
            Session session = getSessionFactory().getCurrentSession();
            if (logger.isDebugEnabled()) {
                logger.debug("Found Hibernate-managed Session [" +
                SessionFactoryUtils.toString(session) +
                "] for Spring managed transaction");
            }
            txObject.setExistingSession(session);
```

Download at http://www.pin5i.com/

```
        }
        catch (HibernateException ex) {
            throw new DataAccessResourceFailureException(
                    "Could not obtain Hibernate-managed Session for Spring-
    managed transaction", ex);
        }
    }
    //在 TransactionOjbect 中设置 DataSource，它也是与线程绑定的。
    if (getDataSource() != null) {
        ConnectionHolder conHolder = (ConnectionHolder)
                TransactionSynchronizationManager.getResource(getDataSource());
        txObject.setConnectionHolder(conHolder);
    }

    return txObject;
}
//Hibernate 事务开始的实现。
protected void doBegin(Object transaction, TransactionDefinition definition) {
    HibernateTransactionObject txObject = (HibernateTransactionObject) transaction;

    if (txObject.hasConnectionHolder() && !txObject.getConnectionHolder().
    isSynchronizedWithTransaction()) {
        throw new IllegalTransactionStateException(
                "Pre-bound JDBC Connection found! HibernateTransactionManager
                    does not support " +
                "running within DataSourceTransactionManager if told to manage
                    the DataSource itself. " +
                "It is recommended to use a single HibernateTransactionManager
                    for all transactions " +
               "on a single DataSource,no matter whether Hibernate or JDBC access.");
    }

    Session session = null;
    /**
    *如果 SessionHolder 没有被创建，那么这里创建 Hibernate 的 Session，并把创建的 Session
    *放到 SessionHolder 中去。
    */
    try {
        if (txObject.getSessionHolder() == null || txObject.getSessionHolder().
    isSynchronizedWithTransaction()) {
            Interceptor entityInterceptor = getEntityInterceptor();
            Session newSession = (entityInterceptor != null ?
                    getSessionFactory().openSession(entityInterceptor) :
                                                getSessionFactory().openSession());
            if (logger.isDebugEnabled()) {
                logger.debug("Opened new Session["+SessionFactoryUtils.toString
                (newSession) + "] for Hibernate transaction");
            }
            txObject.setSession(newSession);
        }
        //这里从 SessionHolder 中取得 session，为创建 HibernateTransaction 做准备。
        session = txObject.getSessionHolder().getSession();

        if(this.prepareConnection && isSameConnectionForEntireSession(session))
        {
           //We're allowed to change the transaction settings of the JDBC Connection.
            if (logger.isDebugEnabled()) {
                logger.debug(
```

Download at http://www.pin5i.com/

```
                "Preparing JDBC Connection of Hibernate Session [" + Session
                FactoryUtils.toString(session) + "]");
            }
            Connection con = session.connection();
            Integer previousIsolationLevel=DataSourceUtils.prepareConnectionForTransaction
    (con, definition);
            txObject.setPreviousIsolationLevel(previousIsolationLevel);
        }
        else {
            //Not allowed to change the transaction settings of the JDBC Connection.
            if(definition.getIsolationLevel()!=TransactionDefinition.ISOLATION_DEFAULT){
                //We should set a specific isolation level but are not allowed to...
                throw new InvalidIsolationLevelException(
                        "HibernateTransactionManager is not allowed to support
                            custom isolation levels: " +
                        "make sure that its 'prepareConnection' flag is on (the
                            default) and that the " +
                        "Hibernate connection release mode is set to 'on_close'
                            (SpringTransactionFactory's default). " +
                        "Make sure that your LocalSessionFactoryBean actually
                            uses SpringTransactionFactory: Your " +
                       "Hibernate properties should *not* include a 'hibernate.
                            transaction.factory_class' property!");
            }
            if (logger.isDebugEnabled()) {
                logger.debug(
                        "Not preparing JDBC Connection of Hibernate Session[" +
                                SessionFactoryUtils.toString(session) + "]");
            }
        }

        if (definition.isReadOnly() && txObject.isNewSession()) {
            // Just set to NEVER in case of a new Session for this transaction.
            // 当事务方法被配置为ReadOnly时，设置session的FlushMode.
            session.setFlushMode(FlushMode.MANUAL);
        }
        //对非ReadOnly事务配置session的FlushMode。
        if (!definition.isReadOnly() && !txObject.isNewSession()) {
            // We need AUTO or COMMIT for a non-read-only transaction.
            FlushMode flushMode = session.getFlushMode();
            if (flushMode.lessThan(FlushMode.COMMIT)) {
                session.setFlushMode(FlushMode.AUTO);
                txObject.getSessionHolder().setPreviousFlushMode(flushMode);
            }
        }
        //这个Transaction是我们在使用Hibernate时常用的Transaction。
        Transaction hibTx = null;

        // Register transaction timeout.
        // 为Hibernate的Transaction设置timeout，并开启事务。
        int timeout = determineTimeout(definition);
        if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
            /**
            *Use Hibernate's own transaction timeout mechanism on Hibernate 3.1+
            *Applies to all statements, also to inserts, updates and deletes!
            */
            hibTx = session.getTransaction();
            hibTx.setTimeout(timeout);
```

Download at http://www.pin5i.com/

```
            hibTx.begin();
        }
        else {
            // Open a plain Hibernate transaction without specified timeout.
            // 创建并开始事务，在不需要设置 timeout 属性的场合。
            hibTx = session.beginTransaction();
        }

        // Add the Hibernate transaction to the session holder.
        /**
        *把 Hibernate 的 Transaction 设置到 TransactionObject 的 SessionHolder 里面，这
        *个 SessionHolder 会和线程绑定。
        */
        txObject.getSessionHolder().setTransaction(hibTx);

        //Register the Hibernate Session's JDBC Connection for the DataSource, if set.
        if (getDataSource() != null) {
            Connection con = session.connection();
            ConnectionHolder conHolder = new ConnectionHolder(con);
            if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
                conHolder.setTimeoutInSeconds(timeout);
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Exposing Hibernate transaction as JDBC transaction["+con+"]");
            }
           TransactionSynchronizationManager.bindResource(getDataSource(), conHolder);
            txObject.setConnectionHolder(conHolder);
        }

        // Bind the session holder to the thread.
        // 如果是新的 SessionHolder，把它和当前线程绑定。
        if (txObject.isNewSessionHolder()) {
            TransactionSynchronizationManager.bindResource(getSessionFactory(),
                                                 txObject.getSessionHolder());
        }
        //在 SessionHolder 中进行状态标志，标识事务已经开始。
        txObject.getSessionHolder().setSynchronizedWithTransaction(true);
    }

    catch (Exception ex) {
        if (txObject.isNewSession()) {
            try {
                if (session.getTransaction().isActive()) {
                    session.getTransaction().rollback();
                }
            }
            catch (Throwable ex2) {
                logger.debug("Could not rollback Session after failed transaction
                begin", ex);
            }
            finally {
                SessionFactoryUtils.closeSession(session);
            }
        }
        throw new CannotCreateTransactionException("Could not open Hibernate
                                              Session for transaction", ex);
    }
}
```

Download at http://www.pin5i.com/

```
//事务挂起的处理。
protected Object doSuspend(Object transaction) {
    HibernateTransactionObject txObject = (HibernateTransactionObject) transaction;
    //把当前的SessionHolder从线程中和TransactionObject释放。
    txObject.setSessionHolder(null);
    SessionHolder sessionHolder =
            (SessionHolder) TransactionSynchronizationManager. UnbindResource
    (getSessionFactory());
    //把当前的ConnectionHolder从线程中和TransactionObject释放。
    txObject.setConnectionHolder(null);
    ConnectionHolder connectionHolder = null;
    if (getDataSource() != null) {
        connectionHolder = (ConnectionHolder) TransactionSynchronizationManager.
    unbindResource(getDataSource());
    }
    return new SuspendedResourcesHolder(sessionHolder, connectionHolder);
}
/**
*为事务提交做的准备，如果配置成 FlushBeforeCommit 并且是新事务，flush Session 中的数据，
*然后把FlushMode设置为MANUAL。
*/
protected void prepareForCommit(DefaultTransactionStatus status) {
    if (this.earlyFlushBeforeCommit && status.isNewTransaction()) {
        HibernateTransactionObject txObject=(HibernateTransactionObject) status.
    getTransaction();
        Session session = txObject.getSessionHolder().getSession();
        if (!session.getFlushMode().lessThan(FlushMode.COMMIT)) {
            logger.debug("Performing an early flush for Hibernate transaction");
            try {
                session.flush();
            }
            catch (HibernateException ex) {
                throw convertHibernateAccessException(ex);
            }
            finally {
                session.setFlushMode(FlushMode.MANUAL);
            }
        }
    }
}
//事务提交的完成。
protected void doCommit(DefaultTransactionStatus status) {
    //取得当前的Hibernate Transaction.
    HibernateTransactionObject txObject = (HibernateTransactionObject) status.
    getTransaction();
    if (status.isDebug()) {
        logger.debug("Committing Hibernate transaction on Session [" +
SessionFactoryUtils.toString(txObject.getSessionHolder().getSession())+"]");
    }
    try {//通过Hibernate的Transaction完成提交。
        txObject.getSessionHolder().getTransaction().commit();
    }
    catch (org.hibernate.TransactionException ex) {
        // Assumably from commit call to the underlying JDBC connection.
        throw new TransactionSystemException ("Could not commit Hibernate
    transaction", ex);
    }
    catch (HibernateException ex) {
```

Download at http://www.pin5i.com/

```
            // Assumably failed to flush changes to database.
            throw convertHibernateAccessException(ex);
        }
    }
    //事务回滚的处理。
    protected void doRollback(DefaultTransactionStatus status) {
        //取得 Hibernate 的 Transaction。
        HibernateTransactionObject txObject = (HibernateTransactionObject) status.
        getTransaction();
        if (status.isDebug()) {
            logger.debug("Rolling back Hibernate transaction on Session [" +
    SessionFactoryUtils.toString(txObject.getSessionHolder().getSession()) + "]");
        }
        try {//通过 Hibernate 的 Transaction 完成回滚。
            txObject.getSessionHolder().getTransaction().rollback();
        }
        catch (org.hibernate.TransactionException ex) {
            throw new TransactionSystemException("Could not roll back Hibernate
            transaction", ex);
        }
        catch (HibernateException ex) {
            // Shouldn't really happen, as a rollback doesn't cause a flush.
            throw convertHibernateAccessException(ex);
        }
        finally {
            if (!txObject.isNewSession() && !this.hibernateManagedSession) {
                // Clear all pending inserts/updates/deletes in the Session.
                // Necessary for pre-bound Sessions, to avoid inconsistent state.
                txObject.getSessionHolder().getSession().clear();
            }
        }
    }
```

在这里我们看到了 HibernateTransactionManager 对事务处理的实现，这些最终的事务处理是通过 Hibernate 的 Transaction 的 commit、rollback 方法调用来完成的，与单独使用 Hibernate 的事务处理没有太多区别。需要注意的是，对使用的 Session 和 Transaction 的取得，因为涉及并发事务处理，所以这些对象往往都是与线程绑定的，Spring 通过一个 SessionHolder 来完成这些事务对象的管理，并通过 ThreadLocal 对象来实现和线程的绑定。

6.6 小结

总体来说，从声明式事务的整个实现中我们看到，声明式事务处理完全可以看成是一个具体的 Spring AOP 应用。从这个角度上看，Spring 事务处理的实现本身，就为应用开发者提供了一个非常优秀的 AOP 应用参考实例。在 Spring 的声明式事务处理中，采用了 IoC 容器的 Bean 配置，来为事务方法调用提供事务属性设置，从而为应用对事务处理的使用提供方便。有了声明式的使用方式，可以把对事务处理的实现与应用代码分离出来。从 Spring 实现的角度来看，声明式事务处理的大致实现过程是这样的：在为事务处理配置好 AOP 的基础设施（比如，对应的 Proxy 代理对象和事务处理 Interceptor 拦截器对象）之后，首先需要完成的是对这些事

Download at http://www.pin5i.com/

务属性配置的读取，这些属性的读取处理是在 TransactionInterceptor 中实现的；在完成这些事务处理属性的读取之后，Spring 为事务处理的具体实现做好了准备。可以看到，Spring 声明式事务处理的过程，同时也是一个整合事务处理实现到 Spring AOP 和 IoC 容器中去的过程。我们在整个过程中可以看到下面的一些要点，在这些要点中，体现了对 Spring 框架的基本特性的灵活使用。

- 如何封装各种不同事务处理环境下的事务处理，比如，对于不同的具体事务处理器的实现，这些事务处理实现有 DataSource 的 Connection、Hibernate 的 Transaction 等，怎样通过一个统一的方式来实现通用的事务处理过程。
- 如何读取事务处理属性值，在事务处理属性正确读取的基础上结合事务处理代码，从而完成在既定的事务处理配置下事务处理方法的实现。
- 如何灵活地使用 Spring AOP 框架，对事务处理进行封装，提供给应用即开即用的声明式事务处理功能。

在这个过程中，有几个 Spring 事务处理的核心类是我们需要关注的。其中包括 TransactionInterceptor，它是使用 AOP 实现声明式事务处理的拦截器，封装了 Spring 对声明式事务处理实现的基本过程；还有就是 TransactionAttributeSource 和 TransactionAttribute 这两个类，它们封装了对声明式事务处理属性的识别，以及信息的读入和配置。我们看到的 TransactionAttribute 对象，可以视为对事务处理属性的数据抽象，如果在使用声明式事务处理的时候，应用没有配置这些属性，Spring 将为用户提供 DefaultTransactionAttribute 对象，在这个 DefaultTransactionAttribute 对象中，提供了默认的事务处理属性设置。

在事务处理过程中，可以看到 TransactionInfo 和 TransactionStatus 这两个对象，它们是存放事务处理信息的主要数据对象，它们通过与线程的绑定来实现事务的隔离性。具体来说，TransactinInfo 对象本身就像是一个栈，对应着每一次事务方法的调用，它会保存每一次事务方法调用的事务处理信息。值得注意的是，在 TransactionInfo 对象中，它持有 TransactionStatus 对象，这个 TransactionStatus 对象是非常重要的，它掌管着事务执行的详细信息，包括具体的事务对象、事务执行状态、事务设置状态等。在事务的创建、启动、提交和回滚的过程中，都需要与这个 TransactionStatus 对象中的数据打交道。在这些与事务管理有关的数据准备完成之后，具体的事务处理是由事务处理器 TransactionManager 来完成的。在事务处理器完成事务处理的过程中，与具体事务处理器无关的操作都被封装到 AbstractPlatformTransactionManager 里面实现了，通过这个抽象的事务处理器，为不同的具体事务处理器提供了通用的事务处理模板，它封装了在事务处理过程中，与具体事务处理器无关的公共的事务处理部分。对于具体的事务处理器的实现，我们可以在具体的事务处理器（比如 DataSourceTransactionManager 和 HibernateTransactionManager）的实现中看到最为底层的事务创建、挂起、提交、回滚操作。

在 Spring 中，也可以通过编程式的方法来使用事务处理器，以帮助我们处理事务。在编程式的事务处理使用中，TransactionDefinition 是定义事务处理属性的类，对于事务处理属性，Spring 还提供了一个默认的事务属性 DefaultTransactionDefinition 来供用户使

Download at http://www.pin5i.com/

用，这种事务处理方式在实现上看起来比声明式事务处理要简单，但编程式实现事务处理却会造成事务处理与业务代码的紧密耦合，因而不经常被使用。尽管如此，我们在这里举编程式使用事务处理的例子，是因为通过了解编程式事务处理的使用，可以清楚地了解 Spring 统一实现事务处理的大致过程。有了这个背景知识，结合对声明式事务处理实现原理的详细分析，比如在声明式事务处理中使用 AOP 对事务处理进行封装，对事务属性配置进行的处理，与线程绑定从而处理事务并发并结合事务处理器的使用等，能够在很大程度上提高我们对整个 Spring 事务处理实现的理解。

Download at http://www.pin5i.com/

第 7 章

Spring 远端调用的实现

子曰："有朋自远方来，不亦乐乎"。
——【春秋】孔子《论语》学而篇

7.1 Spring 远端调用概述

在企业应用开发中，为了达到提高应用可靠性或者平衡资源使用的目的，常常需要考虑分布式计算的解决方案。在分布式计算中，常常涉及服务器系统中各种不同进程之间的通信与计算交互，远端调用 (RMI) 是实现这种计算场景的一种有效方式。此外，还存在着另一种情况，在这种应用场景中，与那些典型的基于 HTML 的 B/S 应用不同，客户端程序需要完成对服务器端应用的直接调用，这也是需要远端调用大显身手的场合。

Spring 中提供了轻量级的远端调用模块，从而为我们在上面提到的应用场景开发，提供平台支持。根据 Spring 的既定策略，它依然只是起到一个集成平台的作用，而并不期望在实现方案上，与已有的远端调用方案形成竞争。也就是说，在 Spring 远端调用架构中，具体的通信协议设计、通信实现，以及在服务器和客户端对远端调用的处理封装，Spring 没有将其作为实现重点，在这个技术点上，并不需要重新发明轮子。对 Spring 来说，它所要完成的工作，是在已有远端调用技术实现的基础上，通过 IoC 与 AOP 的封装，让应用更方便地使用这些远端调用服务，并能够更方便灵活地与现有应用系统实现集成。通过 Spring 封装以后，应用使用远端过程调用非常方便，既不需要改变原来系统的相关实现接口，也不需要为远端调用功能增加新的封装负担。因此，这种使用方式，在某种程度上，可以称为轻量级的远端调用方案。

了解这些背景之后，本章会具体阐述 Spring 是如何完成对已有的远端调用方案的封装的。如果要查看 Spring 远端调用的源代码实现，需要把包 `org.springframework.context` 导入到 Eclipse 的工程环境中来。在这个工程中，包含了远端调用的实现部分，从名字上就可以看到，这个包里面包含的 `remoting` 部分的实现，就是 Spring 远端调用的实现代码。

在实现远端调用的过程中，往往需要涉及客户端和服务器端的相关设置，这些设置通过 Spring 的 IoC 容器就可以很好地完成。同时，Spring 为远端调用的实现提供了许多不同的方案。如 RMI、HTTP 调用器、第三方远端调用库 Hessian/Burlap、基于 Java RMI 的解决方案等。Spring 来对这些方案的具体支持，如果从 Spring 实现架构的角度来看，可以体现在如图 7-1 所示的一系列相关类的设计中。在这个类的层次关系中，首先可以看到的是，Spring 为 `RemotingSupport` 设计的一系列子类，这些子类是 Spring 用来封装远端服务客户端所使用地。相应地，在具体实现上，这些类有一系列的拦截器实现，在这些拦截器中完成了对客户端远端调用的主要封装，和 Spring 提供的

Download at http://www.pin5i.com/

与具体远端调用实现对应的 FactoryBean 一起，构成了远端调用客户端的基础设施。其次，在这个类的层次关系中还可以看到，在 RemoteService 下有一系列子类，这些子类是为远端调用的服务器端的实现提供导出服务的，通过这个服务导出的设计支持，客户端可以完成对这些导出的远端服务的调用。

下面，我们将对 HTTP 调用器、Hessian/Burlap 的远端调用、RMI 远端调用的实现原理进行分析。在这些实现原理的分析中，大致包含了基本配置、客户端实现和服务器端实现这几个基本的部分。在本章的分析中，读者会看到很多对图 7-1 中出现的类所作的实现原理的分析，比如封装 HTTP 调用器客户端的 HttpInvokerProxyBean 和 HttpInvokerClientInterceptor，为 Hessian/Burlap 提供客户端封装服务的 Hessian 的 HessianPrxoyFactoryBean/BurlapPrxoyFactoryBean 和 HessianClientInterceptor/BurlapClientInterceptor，为封装 RMI 提供客户端服务的 RmiProxyFactoryBean 和 RmiClientInterceptor，等等。在开始分析之前，希望这个类层次图能够让读者对 Spring 远端调用模块有一个初步的印象和了解。Spring 对不同的远端调用的实现封装，基本上都采用了类似的模式来完成，比如，在客户端都是通过相关的 ProxyFactoryBean 和 ClientInterceptor 来完成的，在服务器端是通过 ServiceExporter 来导出远端的服务对象的。有了这些统一的命名规则，应用配置和使用远端调用会非常方便，同时，通过对这些 Spring 远端调用基础设施实现原理的分析，还可以看到一些常用处理方法的技术实现，比如对代理对象的使用、拦截器的使用、通过 afterPropertiesSet 来启动远端调用基础设施的建立，等等，这些都是在 Spring 中常用的技术。

图 7-1 远端服务的相关实现类及其关系

Download at http://www.pin5i.com/

7.2　Spring HTTP 调用器的实现原理

7.2.1　配置 HTTP 调用器客户端

顾名思义，HTTP 调用器是基于 HTTP 协议提供的一种远端调用方案。使用 HTTP 调用器和使用 Java RMI 一样，需要使用 Java 的序列化机制来完成客户端和服务器端的通信。在 Spring 中，我们从客户端的基础配置模块 HttpInvokerProxyFactoryBean 入手，对 HTTP 调用器的实现原理进行分析。这个 HttpInvokerProxyFactoryBean 是一个 FactoryBean，这个工厂 Bean 的作用是为客户端 HTTP 调用器提供服务配置，使用我们熟悉的 FactoryBean 的设计方式，来封装客户端需要的远端代理对象，这是 Spring 远端调用模块处理客户端封装的一个模式。下面简单回顾一下 HttpInvoker 的使用，在使用 HttpInvoker 时，首先需要配置客户端的 HttpInvokerProxyFactoryBean，然后需要设置应用 Bean 对 ProxyFactoryBean 的配置，具体的客户端配置如代码清单 7-1 所示。在代码清单中可以看到对 HttpInvokerProxyFactoryBean 的使用配置，比如需要配置客户端访问的远端服务的 URL 地址，设置远端调用服务的服务接口，然后把这个 ProxyFactory 设置到客户端应用 Bean 的 remoteService 属性中去。有了这个设置，客户端应用就已经准备就绪了，它就可以像调用本地调用一样享用远端的服务了。

代码清单 7-1　配置 HTTP 调用器的客户端访问设置

```
<bean id="proxy"class="org.springframework.remoting.httpinvoker.
    HttpInvoker ProxyFactoryBean">
<property name="serviceUrl">
<value>http://yourhost:8080/yourURL</>
</property>
<property name="serviceInterface">
<value>yourInterface<value/>
</property>
</bean>
<bean id="yourBean" class="yourClass">
<property name="remoteService">
    <ref bean="proxy"/>
</property>
```

在这些对 IoC 容器的配置中，在 HttpInvokerProxyFactoryBean 中封装了对应的远端服务的信息，比如域名、端口号和服务所在的 URL，这些都是访问远端服务调用所需要的信息。同时，由于使用的是 HttpInvoker，所以在 URL 中指定的协议是 HTTP 协议。对访问远端服务调用的客户端而言，它只要持有 HttpInvokerProxyFactoryBean 提供的代理对象，就可以方便地使用远端调用了，使用起来很简单。对客户端来说就像本地调用一样，在远端调用过程中，发生的数据通信以及与远端服务的交互，都被 Spring 使用 Proxy 代理类进行了封装，对客户端是透明的。具体地说，这些封装实现在 HttpInvokerProxyFactoryBean 这个 FactoryBean 生产出来的代理对象中完成，下面看看这个 HttpInvokerProxyFactoryBean 是怎样工作的。

7.2.2　HTTP 调用器客户端的实现

了解 HTTP 调用器的基本设置后，下面看看在 HttpInvokerProxyFactoryBean 中，

Download at http://www.pin5i.com/

是如何完成对远端服务客户端的封装的，这些封装实现如代码清单 7-2 所示。在 HtttpInvokerProxyFactory 中，设置了 serviceProxy 对象作为远端服务的本地代理对象。同时，在依赖注入完成以后，通过 afterPropertiesSet 来对远端调用完成设置。这个 afterPropertiesSet 完成的设置包括：使用 ProxyFactory 生成代理对象，为代理对象设置代理接口方法并把 ProxyFactory 生成的代理对象设置给 serviceProxy。可以看到，在我们熟悉的 FactoryBean 的接口方法 getObject 的实现中，把这个 serviceProxy 对象，也就是把生成的代理对象，提供给了访问远端调用的客户端应用。

代码清单 7-2 HttpInvokerProxyFactoryBean 的实现

```
public class HttpInvokerProxyFactoryBean extends HttpInvokerClientInterceptor
        implements FactoryBean<Object> {
    //这是远端对象的代理。
    private Object serviceProxy;
    @Override
    //在注入完成之后，设置远端对象代理。
    public void afterPropertiesSet() {
        super.afterPropertiesSet();
        //需要配置远端调用的接口。
        if (getServiceInterface() == null) {
            throw new IllegalArgumentException("Property 'serviceInterface' is
            required");
        }/**
         *这里使用 ProxyFactory 来生成远端代理对象，注意这个 this，因为 HttpInvokerProxy
         *FactoryBean 的基类是 HttpInvokerClientInterceptor，
         *所以代理类的拦截器被设置为 HttpInvokerClientInterceptor。
         */
        this.serviceProxy=new ProxyFactory(getServiceInterface(), this).getProxy
    (getBeanClassLoader());
    }
    //FactoryBean 生产对象的入口。返回的是 serviceProxy 对象，这是一个代理对象。
    public Object getObject() {
        return this.serviceProxy;
    }
    public Class<?> getObjectType() {
        return getServiceInterface();
    }
    public boolean isSingleton() {
        return true;
    }
```

通过 FactoryBean 的封装，getObject 取得的实际上是一个代理对象。在代码实现中，可以看到为这个代理对象配置了一个拦截器 HttpInvokerClientInterceptor，在这个拦截器中，拦截了对代理对象的方法调用。我们到拦截器的实现中去看看它具体做了什么，如代码清单 7-3 所示。对于这个拦截器，触发的拦截行为在 invoke 回调方法中实现，在这个回调方法中，所做的处理是通过 HTTP 请求触发远端服务，在触发远端服务的时候，通过生成一个 MethodInvocation 对象，从而封装了当前代理方法调用的具体调用场景。然后，代理对象会把这个 MethodInvocation 对象作为参数，通过 HTTP 的 Java 对象序列化机制传输到服务器端，从而交给远端服务去执行。在远端服务执行完成后，服务器端的服务对象会把执行结果返回，这个执行结果会封装在 RemoteInvocationResult 对象中，同样也是通过 HTTP 的 Java 对象序列化机制，回送到客户端，从而交由客户端应用来使用。

Download at http://www.pin5i.com/

代码清单 7-3　HttpInvokerClientInterceptor 的 invoke 回调

```
//对代理对象的方法调用入口。
public Object invoke(MethodInvocation methodInvocation) throws Throwable {
    if (AopUtils.isToStringMethod(methodInvocation.getMethod())) {
        return "HTTP invoker proxy for service URL [" + getServiceUrl() + "]";
    }
    //创建 RemoteInvocation 对象，它封装了对远端的调用，这些远端调用通过序列化的机制完成。
    RemoteInvocation invocation = createRemoteInvocation(methodInvocation);
    RemoteInvocationResult result = null;
    try {
        //这里是对远端调用的入口。
        result = executeRequest(invocation, methodInvocation);
    }
    catch (Throwable ex) {
        throw convertHttpInvokerAccessException(ex);
    }
    try {//返回远端调用的结果。
        return recreateRemoteInvocationResult(result);
    }
    catch (Throwable ex) {
        if (result.hasInvocationTargetException()) {
            throw ex;
        }
        else {
            throw new RemoteInvocationFailureException("Invocation of method[" +
                methodInvocation.getMethod() +
                 "]failed in HTTP invoker remote service at["+getServiceUrl()+"]",ex);
        }
    }
}
```

我们可以看到，RemoteInvocation 是由 DefaultRemoteInvocationFactory 来创建的，它创建出来的 RemoteInvocation 实际上是一个数据对象，在这个数据对象中，封装了调用的具体信息，比如调用方法名、参数、参数类型等。这些封装都可以在 RemoteInvocation 的实现中看到。远端调用的具体实现过程，是由 executeRequest 来完成的，如代码清单 7-4 所示。在代码清单中可以看到，是通过使用 HttpInvokerRequestExecutor 来触发远端调用的。

代码清单 7-4　HttpInvokerRequestExecutor 的 executeRequest

```
//通过 HttpInvokerRequestExecutor 的 executeRequest 来完成调用。
protected RemoteInvocationResult executeRequest (RemoteInvocation invocation)
    throws Exception {
    return getHttpInvokerRequestExecutor().executeRequest(this, invocation);
}
```

可以看到，HttpInvokerRequestExecutor 是一个 SimpleHttpInvokerRequestExecutor。也就是说，我们可以到 SimpleHttpInvokerRequestExecutor 中去看看 executeRequest 方法调用的实现，如代码清单 7-5 所示。在 SimpleHttpInvokerRequestExecutor 的实现中，封装了整个 HTTP 调用器客户端实现的基本过程：首先，它会打开一个 HTTP 链接，接着通过 HTTP 的对象序列化，把封装好的调用场景，也就是在前面生成的 RemoteInvocation 传送到服务器端，请求服务响应；其次，在服务器端完成服务以后，会把执行结果，以对象序列化的方式回送给 HTTP 响应（HttpResponse）；最后，客户端应用，也就是在这个 executeRequest 方法

Download at http://www.pin5i.com/

中，会从 HTTP 响应中读出远端服务的执行结果。在这里，使用了 HTTP 的请求/响应机制，来实现对远端方法的访问与执行结果返回，这个过程，与我们熟悉的 Servlet 实现机制是大致一样的。对比其他的 Spring 远端调用方案的实现，HttpInvoker 的实现特点在于，它使用的是 Java 虚拟机提供的基本特性，比如 HTTP 的传输机制、对象的序列化和反序列化。相对于在 Spring 中提供的，像 Hessian/Burlap 这些方案而言，使用 HTTP 调用器的完成远端调用的应用不需要引用第三方类库，因而使用起来是非常方便的。

代码清单 7-5 SimpleHttpInvokerRequestExecutor 的 doExecuteRequest

```
/**
*这是 HTTP 调用器实现的基本过程，通过 HTTP 的 request 和 reponse 来完成通信，
*在通信的过程中传输的数据是序列化的对象。
*/
protected RemoteInvocationResult doExecuteRequest(
        HttpInvokerClientConfiguration config, ByteArrayOutputStream baos)
        throws IOException, ClassNotFoundException {
    //打开一个标准 J2SE HttpURLConnection。
    HttpURLConnection con = openConnection(config);
    prepareConnection(con, baos.size());
    //远端调用封装成 RemoteInvocation 对象，它通过序列化被写到对应的 HttpURLConnection 中去。
    writeRequestBody(config, con, baos);
    //这里取得远端服务返回的结果，然后把结果转换成 RemoteInvocationResult 返回。
    validateResponse(config, con);
    InputStream responseBody = readResponseBody(config, con);
    return readRemoteInvocationResult(responseBody, config.getCodebaseUrl());
}
//把序列化对象输出到 HttpURLConnection 去。
protected void writeRequestBody(
        HttpInvokerClientConfiguration config, HttpURLConnection con, ByteArray
    OutputStream baos)
        throws IOException {
    baos.writeTo(con.getOutputStream());
}
/**
*为使用 HttpURLConnection 完成对象序列化，需要进行一系列的配置。
*比如配置请求方式为 post，配制请求属性等。
*/
protected void prepareConnection(HttpURLConnection con,int contentLength)throws
    IOException {
    con.setDoOutput(true);
    con.setRequestMethod(HTTP_METHOD_POST);
    con.setRequestProperty(HTTP_HEADER_CONTENT_TYPE, getContentType());
    con.setRequestProperty(HTTP_HEADER_CONTENT_LENGTH,
    Integer.toString (contentLength));
    LocaleContext locale = LocaleContextHolder.getLocaleContext();
    if (locale != null) {
        con.setRequestProperty(HTTP_HEADER_ACCEPT_LANGUAGE,
    StringUtils.toLanguageTag(locale.getLocale()));
    }
    if (isAcceptGzipEncoding()) {
        con.setRequestProperty(HTTP_HEADER_ACCEPT_ENCODING, ENCODING_GZIP);
    }
}
//获得 HTTP 响应的 IO 流。
protected InputStream readResponseBody(HttpInvokerClientConfiguration config,
```

Download at http://www.pin5i.com/

```
        HttpURLConnection con)
            throws IOException {
        //如果是通过gzip压缩，那么需要先解压。
        if (isGzipResponse(con)) {
            // GZIP response found - need to unzip.
            return new GZIPInputStream(con.getInputStream());
        }
        else {
            // Plain response found.
            // 正常的HTTP响应输出。
            return con.getInputStream();
        }
    }
```

可以看到对远端服务执行结果的返回对象的处理是在 AbstractHttpInvokerRequestExecutor 中实现的，在通过 HTTP 把对象反序列化之后，会把远端的服务结果封装成 RemoteInvocationResult 对象，这部分实现如代码清单 7-6 所示。这个客户端对远端服务调用结果的处理过程并不复杂，它是一个通过 Java 对象的反序列化，从 HTTP 响应中得到服务执行结果的过程。

代码清单 7-6 AbstractHttpInvokerRequestExecutor 的 readRemoteInvocationResult

```
//把返回的对象封装到RemoteInvocationResult中去。
protected RemoteInvocationResult readRemoteInvocationResult(InputStream is, String
    codebaseUrl)
        throws IOException, ClassNotFoundException {

    ObjectInputStream ois=createObjectInputStream(decorateInputStream(is),codebaseUrl);
    try {
        return doReadRemoteInvocationResult(ois);
    }
    finally {
        ois.close();
    }
}
protected RemoteInvocationResult doReadRemoteInvocationResult(ObjectInputStream ois)
        throws IOException, ClassNotFoundException {
    Object obj = ois.readObject();
    if (!(obj instanceof RemoteInvocationResult)) {
        throw new RemoteException("Deserialized object needs to be assignable to type["+
                RemoteInvocationResult.class.getName() + "]: " + obj);
    }
    return (RemoteInvocationResult) obj;
}
```

看到这里，我们大致了解了在 HTTP 调用器客户端中完成远端调用的基本过程。简单来说，这个过程是这样的：首先由客户端应用调用代理方法，在调用发生以后，代理类会先运行拦截器，对代理的方法调用进行拦截。在拦截器的拦截行为中，首先，会把本地发生的方法调用进行封装，具体来说，就是封装成 MethodInvocation 对象；其次，把这个 MethodInvocation 对象，通过序列化和 HTTP 请求发送到服务器端，在服务器端的处理完成以后，会通过 HTTP 响应返回处理结果，这个处理结果，被封装在 RemoteInvocationResult 对象中。整个过程是一个典型的 HTTP 客户机-服务器实现的基本过程，只是在这里传输的数据是序列化的 Java 对象，而不是那些常见的经过 URL 请求，得到的 HTML 页面响应而已。

Download at http://www.pin5i.com/

7.2.3 配置 HTTP 调用器远端服务器端

在了解了客户端的实现原理以后，我们来看看在服务器端是怎样实现对远端服务请求的服务响应的。同样地，与客户端需要使用 IoC 容器进行配置，才能使用远端服务一样，在服务器端同样也是需要做一些简单的配置。通过这些配置，可以把远端调用服务在服务器端导出，暴露给客户端使用。与客户端的远端调用的设置很类似，在服务器端的配置也很简单，我们在这里简单地回顾一下，如代码清单 7-7 所示。在配置中，可以看到需要设置远端服务对应的 URL，还需要设置提供服务的 Bean，这个 Bean 是由应用来完成的服务实现。可以看到，在这个设置中，还有一个叫 servicebean 的 Bean，这个 Bean 的服务方法会被调用，从而完成服务的最终执行。同时，需要在 serviceInterface 中，设置提供的服务接口方法，设置 Proxy 的代理方法调用。

代码清单 7-7 HTTP 调用器服务器端服务的导出设置

```
<bean name="/remoteServiceURL"class="org.springframework.remoting.httpinvoker.
    HttpInvokerServiceExporter">
<property name="service">
<ref bean="serviebean"/>
</property>
<property name="serviceInterface">
<value>yourInterface</value>
</property>
</bean>
```

通过这些简单的配置，就可以在服务器端导出远端服务。具体的远端服务是通过 service 属性中配置的 Bean 来提供的，这个服务 Bean 封装在 HttpInvokerServiceExporter 中，这个 HttpInvokerServiceExporter 封装了对 HTTP 协议的处理以及 Java 对象的序列化功能，然后通过 Proxy 代理类进行封装，从而成为 HTTP 调用器服务器端的基础设施。

7.2.4 HTTP 调用器服务器端的实现

在服务器端使用 Spring HTTP 远端调用，需要配置 HttpInvokerServiceExporter，作为远端服务的服务导出器。在这个服务导出器中，需要配置对应的 URL 请求，以及需要导出的供客户端使用的服务对象、服务接口等信息的配置。HttpInvokerServiceExporter 的使用是与 Spring MVC 结合在一起的，实际上，是一个我们熟悉的 Spring MVC 框架中的 Controller。在本书第 4 章 Spring MVC 实现原理的分析中，我们知道，在 Spring MVC 中，具体的映射和转发是由 DispatcherServlet 来完成的。然后，通过这个 Spring MVC 框架，由配置好的 Controller 来完成对应的 URL 的数据处理。

下面，我们看看在这个服务导出器中，是如何对客户端发起的远端调用的服务请求进行响应的，过程如代码清单 7-8 所示。我们可以看到，一个和客户端访问远端服务的过程相逆的处理过程。具体来说，这个过程是这样的，在执行需要的服务之前，首先会从 HTTP 请求中读取 RemoteInvocation 对象，读者在前面对客户端的实现原理分析中，已经见到过这个 RemoteInvocation 对象了，它封装了访问远端服务的调用场景，比如具体的远端方法名、调用参数等。

在服务器端，会从 HTTP 请求中反序列化得到 RemoteInvocation 对象，有了这个对象以

Download at http://www.pin5i.com/

后，会调用配置好的的服务方法，来执行请求的远端服务。在服务执行完成以后，通过 HTTP 响应把执行结果通过对象的序列化输出到客户端，从而完成整个服务器端的服务过程，从代码清单 7-8 中可以看到，这个过程很清晰，包括了服务请求接受、服务执行以及最后返回服务结果的完整过程。

代码清单 7-8 HttpInvokerServiceExporter 的 handleRequest 实现

```
//Controller 的执行入口，对相应的 HttpRequest 进行响应。
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    try {
        //从 HttpRequest 中得到序列化的 RemoteInvocation 对象。
        RemoteInvocation invocation = readRemoteInvocation(request);
        //这是对服务对象的调用，依据 RemoteInvocation 对象封装的调用要求来完成。
        RemoteInvocationResult result=invokeAndCreateResult(invocation,getProxy());
        //把服务对象的结果，通过 HttpResponse 返回。
        writeRemoteInvocationResult(request, response, result);
    }
    catch (ClassNotFoundException ex) {
       throw new NestedServletException("Class not found during deserialization", ex);
    }
}
```

可以看到，响应 HTTP 请求的过程是非常简单明了的。下面我们来看看，是怎样从请求中反序列化得到 RemoteInvocation 对象的。这个对象封装了对服务调用的基本信息，比如调用的方法名、参数、参数类型等。了解 HTTP 调用器远端调用客户端实现的读者，一定不会对这个数据对象感到陌生。在服务器端，得到 RemoteInvocation 对象的过程，如代码清单 7-9 所示。从代码清单中可以看到，这个得到 RemoteInvocation 数据对象的过程实际上是一个对象反序列化的实现过程。在实现中，服务器端首先会从 HttpRequest 中得到一个对象的输入流，然后从这个输入流中读取对象，最后，把这个对象转型成 RemoteInvocation 类型的对象返回。如果得到的对象不是 RemoteInvocation 对象，还会抛出异常，表示服务器只兼容由 HTTP 调用器远端调用客户端发起的服务请求。

代码清单 7-9 HttpInvokerServiceExporter 的 readRemoteInvocation

```
protected RemoteInvocation readRemoteInvocation(HttpServletRequest request,
    InputStream is)
        throws IOException, ClassNotFoundException {
//从 ObjectInputStream 中反序列化对象，同时转化成 RemoteInvocation.
    ObjectInputStream ois=createObjectInputStream(decorateInputStream(request,is));
    try {
        return doReadRemoteInvocation(ois);
    }
    finally {
        ois.close();
    }
}
//对 ObjectInputStream 的读取，然后作为 RemoteInvocation 返回。
protected RemoteInvocation doReadRemoteInvocation(ObjectInputStream ois)
        throws IOException, ClassNotFoundException {
    Object obj = ois.readObject();
    if (!(obj instanceof RemoteInvocation)) {
        throw new RemoteException("Deserialized object needs to be assignable to
                  type["+RemoteInvocation.class.getName() + "]: " + obj);
    }
```

Download at http://www.pin5i.com/

```
        return (RemoteInvocation) obj;
    }
```

在通过 HTTP 请求得到客户端传过来的 RemoteInvocation 对象以后，就可以进行服务方法的调用了。服务调用需要的基本信息，都封装在 RemoteInvocation 对象中。这个服务调用过程，是由 invokeAndCreateResult 方法来实现的，具体的实现过程如代码清单 7-10 所示。这个方法完成的任务就像它的名字一样，由它来启动服务并创建服务执行结果。对于服务执行结果，是通过生成一个 RemoteInvocationResult 对象来封装的。

代码清单 7-10　服务器端的服务执行

```
protected RemoteInvocationResult invokeAndCreateResult(RemoteInvocation invocation,
        Object targetObject) {
        try {
            Object value = invoke(invocation, targetObject);
            return new RemoteInvocationResult(value);
        }
        catch (Throwable ex) {
            return new RemoteInvocationResult(ex);
        }
    }
```

在代码清单 7-10 中看到的 invoke 方法封装了服务器端调用的主体，这个 invoke 方法在 HttpInvokerServiceExporter 的基类 RemoteInvocationSerializingExporter 中实现，如代码清单 7-11 所示。具体的服务执行是由 DefaultRemoteInvocationExecutor 来完成的，这个执行器执行服务需要的参数有两个：一个是客户端访问场景的数据封装，也就是 MethodInvocation 对象；另一个是提供服务的具体服务对象，这个服务对象在 IoC 容器中配置，并被通过依赖注入设置进来。有了 MethodInvocation 对象封装的调用场景，服务对象就可以根据需要去完成服务了。

代码清单 7-11　RemoteInvocationSerializingExporter 的 invoke

```
protected Object invoke(RemoteInvocation invocation, Object targetObject)
            throws NoSuchMethodException,IllegalAccessException,InvocationTargetException{
        if (logger.isTraceEnabled()) {
            logger.trace("Executing " + invocation);
        }
        try{//调用 RemoteInvocationExecutor，这个执行器是 DefaultRemoteInvocationExecutor.
            return getRemoteInvocationExecutor().invoke(invocation, targetObject);
        }
        catch (NoSuchMethodException ex) {
            if (logger.isDebugEnabled()) {
                logger.warn("Could not find target method for " + invocation, ex);
            }
            throw ex;
        }
        catch (IllegalAccessException ex) {
            if (logger.isDebugEnabled()) {
                logger.warn("Could not access target method for " + invocation, ex);
            }
            throw ex;
        }
        catch (InvocationTargetException ex) {
            if (logger.isDebugEnabled()) {
               logger.debug("Target method failed for"+invocation,ex.getTargetException());
            }
```

Download at http://www.pin5i.com/

```
            throw ex;
        }
    }
```

具体的服务执行，是由执行器 DefaultRemoteInvocationExecutor 来完成的，这个执行器的 invoke 方法的实现很简单也很巧妙，在代码实现中可以看到，它把调用交给了 RemoteInvocation 对象，如代码清单 7-12 所示。

代码清单 7-12 DefaultRemoteInvocationExecutor 的 invoke

```
public Object invoke(RemoteInvocation invocation, Object targetObject)
        throws NoSuchMethodException,IllegalAccessException,InvocationTargetException{
    Assert.notNull(invocation, "RemoteInvocation must not be null");
    Assert.notNull(targetObject, "Target object must not be null");
    return invocation.invoke(targetObject);
}
```

绕了一圈又回到 MethodInvocation 的设计中来了，但这里看到的 MethodInvocation 对象，实际上已经是在服务器端的对象了，由于是服务器端的对象，因而它可以很方便地获取服务器端的服务资源。Spring 实现服务器对象的调用也是很简单明了的，如代码清单 7-13 所示。在代码清单中可以看到，它使用了反射技术来完成方法调用得到 Method 对象，然后，调用 Method 对象的 invoke 方法，并在 invoke 方法中设置了具体执行服务的目标对象，以及执行方法的输入参数，从而完成远端调用服务的执行。

代码清单 7-13 MethodInvocation 的 invoke

```
public Object invoke(Object targetObject)
        throws NoSuchMethodException,IllegalAccessException,InvocationTargetException {
    //取得服务对象的调用方法，通过反射完成调用，并得到调用结果返回。
    //调用方法名、参数类型以及调用参数都是在客户端封装好，并通过HTTP的Java序列化传递到服务器端。
    Method method=targetObject.getClass().getMethod(this.methodName,this.parameterTypes);
    return method.invoke(targetObject, this.arguments);
}
```

服务对象的方法调用完成之后，会把调用结果通过 HTTP 响应和对象序列化传给 HTTP 调用器客户端，从而完成整个 HTTP 调用器的远端调用过程，如代码清单 7-14 所示。在这里，使用 HTTP 响应，传回服务执行结果的过程与处理正常的 HTTP 响应类似。有一点需要注意，在使用 HttpResponse 的输出流之前，需要为输出流设置 ContentType 属性，这个属性会被设置为 application/x-java-serialized-object 的值，表示此时在流里传输的是 Java 的序列化对象，在这个传输过程中，用 HTTP 作为数据的传输通道。

代码清单 7-14 writeRemoteInvocationResult 返回调用结果

```
protected void writeRemoteInvocationResult(
      HttpServletRequest request,HttpServletResponse response,RemoteInvocationResult result)
      throws IOException {
//设置Response的ContentType属性，设置为application/x-java-serialized-object.
    response.setContentType(getContentType());
    writeRemoteInvocationResult(request,response,result,response.getOutputStream());
}
//输出到HTTP的Response，然后把Response关闭。
protected void writeRemoteInvocationResult(
        HttpServletRequest request,HttpServletResponse response, RemoteInvocationResult
    result, OutputStream os)
        throws IOException {
    ObjectOutputStream oos=createObjectOutputStream(decorateOutputStream (request,
```

Download at http://www.pin5i.com/

```
            response, os));
        try {
            doWriteRemoteInvocationResult(result, oos);
            oos.flush();
        }
        finally {
            oos.close();
        }
    }
```

这样，经过这一系列的处理过程，服务执行结果对象又回到了 HTTP 的远端调用客户端。在客户端从 HTTP 响应读取对象之后，它把这个看起来像是在本地实现，其实是由远端服务对象完成的调用结果，交给发起远端调用的客户端调用方法，从而最终完成整个远端调用的过程。这个过程很有特点，它使用了 HTTP 的请求和响应作为通信通道。在这个通信通道里面，并没有再做进一步的附加的通信协议的封装，而且，在这个处理过程中，使用的都是 Java 和 Spring 框架已有的特性，比如，通过 IoC 的配置以及代理对象拦截器的封装处理，再加 Java 的序列化和反序列化，以及在服务器端的 Spring MVC 框架的使用。通过这些已有的技术实现，让使用者感觉它的实现风格非常的简洁轻快，整个代码实现，阅读起来也让人感到非常赏心悦目。

7.3 Spring Hession/Burlap 的实现原理

前面我们看到了，使用 HTTP 调用器完成远端调用的实现原理，在 HTTP 调用器的实现中，使用的都是 Java 和 Spring 框架的已有特性。作为应用平台的 Spring，还为用户使用远端调用提供了其他选择，在这些选择中，包括集成的开源轻量级的第三方远程调用协议实现，比如 Caocho 公司发布的一个轻便的二进制协议 Hessian，以及另一个基于 XML 的协议实现 Burlap。尽管二者一个使用二进制协议，一个使用 XML 协议，但都是建立在使用 HTTP 协议的基础上，把 HTTP 作为其传输数据的基本协议。关于 Hessian 和 Burlap 的使用和相关信息，有兴趣的读者，可以到它的官方网站（http://hessian.caucho.com/）去获取相应的信息。

7.3.1 Hessian/Burlap 客户端的配置

在使用 HessianProxyFactoryBean 实现远端调用时，首先，同样需要在客户端配置 HessianProxyFactoryBean，看到这个 ProxyBean，一定是 Spring 又发挥了 AOP 的强大功能。我们使用一个简单的例子，回顾一下这个配置，如代码清单 7-15 所示。

代码清单 7-15 Hessian 客户端设置

```
<bean id="hessianProxy"class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
<property name="serviceUrl">
<value>http://yourhost:8080/serviceURL</value>
</property>
<property name="serviceInterface">
<value>yourInterface</value>
</property>
</bean>
```

可以看到，在代码清单 7-15 中的配置，需要设置远端调用的服务地址，这个时候，因为 Hessian 是基于 HTTP 来完成传输的，所以在这个设置中，需要给出 HTTP 协议的域名/IP 地址、

Download at http://www.pin5i.com/

端口号和服务所在的 URL 地址。而这些服务 URL 地址，需要和服务器端的服务约定好，同时需要在服务器端导出服务，才能让远端调用服务器顺利地响应客户端的服务请求。在服务器端，对服务对象导出的 IoC 配置，如代码清单 7-16 所示。同样可以看到，对服务 URL 地址、服务对象 service 以及服务接口 serviceInterface 等属性的配置，这些配置都是和客户端配置相一致的。

代码清单 7-16 Hessian 服务器端设置

```
<bean name="/serviceURL"class="org.springframework.remoting.caucho.HessianServiceExporter">
<property name="service">
<ref bean="seviceBean">
</property>
<property name="serviceInterface">
<value>yourInterface</value>
</property>
</bean>
```

7.3.2 Hessian 客户端的实现

和前面一样，我们从客户端的实现原理入手，去了解 Spring 是怎样通过封装 Hessian 来提供远端调用服务的。让我们到 HessianProxyFactoryBean 的代码中，去看一下 HessianProxyFactory 对 Hessian 客户端封装的实现，如代码清单 7-17 所示。

代码清单 7-17 HessianProxyFactoryBean 的实现

```
public class HessianProxyFactoryBean extends HessianClientInterceptor implements
    FactoryBean<Object> {
//这个对象是 Proxy 代理对象。
private Object serviceProxy;

@Override
//依赖注入完成以后，设置 Proxy 代理对象。
public void afterPropertiesSet() {
    super.afterPropertiesSet();
    /**
    *通过 ProxyFactory 生成代理对象，拦截器使用 HessianClientInterceptor，因为
    *HessianProxyFactoryBean 本身是 HessianClientInterceptor 的子类,所以这里使用 this
    *为代理对象设置拦截器，getServiceInterface 取得在 BeanDefinition 中定义的接口，关于
    *ProxyFactory 是怎样使用设置的接口定义和拦截器
    */
    //生成代理对象的实现原理，可以参考本书对 Spring AOP 实现的原理分析。
    this.serviceProxy = new ProxyFactory(getServiceInterface(), this). getProxy
    (getBeanClassLoader());
}

/**
*这是 FactoryBean 生产对象的方法，调用容器的 getBean 实际上取得的是 serviceProxy,
*也就是 Proxy 代理对象。
*/
public Object getObject() {
    return this.serviceProxy;
}
public Class<?> getObjectType() {
    return getServiceInterface();
}
public boolean isSingleton() {
```

Download at http://www.pin5i.com/

```
        return true;
    }
    }
```

可以在代码清单 7-17 中看到，FactoryBean 的主要功能是完成代理 Proxy 对象的生成和拦截器的设置。而在通过这个 FactoryBean 来实现 Hessian 远端调用的设置中，驱动 Hessian 的过程是被封装在设置的拦截器 HessianClientInterceptor 中来完成的。可以看到，这些设置都是在 afterPropertiesSet 这个 IoC 容器的回调方法中实现的。

在生成了代理对象和为代理对象设置好拦截器之后，我们来看看，在拦截器中是怎样完成对远端调用的具体封装和实现的。它们在 HessianClientInterceptor 中实现，如代码清单 7-18 所示。在这里使用了 HessianProxyFactory 创建的 Proxy 代理对象来完成具体的远端调用，这个 Hessian 的 Proxy 对象的使用，是通过 Method 的反射调用来完成的。值得注意的是，这个 Proxy 对象是 Hessian 的一个实现类，而不是我们通常看到的 Java 的 Proxy 代理对象。对 HessianProxyFactory 的 HessianClientInterceptor 设置，是由 IoC 容器完成注入的，然后在 prepare 方法中，也就是在 afterPropertiesSet 的方法调用中，实现了 Hessian 的 Proxy 对象的创建。

代码清单 7-18　HessianClientInterceptor 的 invoke

```
public Object invoke(MethodInvocation invocation) throws Throwable {
    if (this.hessianProxy == null) {
        throw new IllegalStateException("HessianClientInterceptor is not properly
    initialized - " +
                "invoke 'prepare' before attempting any operations");
    }
    ClassLoader originalClassLoader = overrideThreadContextClassLoader();
        try/**
        {*这里是驱动 hessian 完成远端调用的入口，这个 hessianProxy 是由
        *HessianProxyFactory 生成的代理对象，这个 HessianProxyFactory
        *是 hessian 的类，已经不是 Spring 框架的内容了。
        */
        return invocation.getMethod().invoke(this.hessianProxy,invocation.getArguments());
    }
    catch (InvocationTargetException ex) {
        if (ex.getTargetException() instanceof HessianRuntimeException) {
            HessianRuntimeException hre=
            (HessianRuntimeException)ex.getTargetException();
            Throwable rootCause=(hre.getRootCause()!=
            null ? hre.getRootCause() : hre);
            throw convertHessianAccessException(rootCause);
        }
        else if(ex.getTargetException()instanceof UndeclaredThrowableException)
        {
            UndeclaredThrowableException utex = (UndeclaredThrowableException)
            ex.getTargetException();
            throw convertHessianAccessException(utex.getUndeclaredThrowable());
        }
        throw ex.getTargetException();
    }
    catch (Throwable ex) {
        throw new RemoteProxyFailureException(
        "Failed to invoke Hessian proxy for remote service["+getServiceUrl()+"]",ex);
    }
    finally {
```

Download at http://www.pin5i.com/

```
            resetThreadContextClassLoader(originalClassLoader);
        }
    }
    //依赖注入完成之后，使用 Hessian 的准备工作。
    public void afterPropertiesSet() {
        super.afterPropertiesSet();
        prepare();
    }
    //调用 createHessianProxy，这里的 proxyFactory 是 Hessian 的类 HessianProxyFactory。
    public void prepare() throws RemoteLookupFailureException {
        try {
            this.hessianProxy = createHessianProxy(this.proxyFactory);
        }
        catch (MalformedURLException ex) {
            throw new RemoteLookupFailureException("Service URL [" + getServiceUrl()
        + "] is invalid", ex);
        }
    }

    /**
    *这是调用 HessianProxyFactory 来生成客户端 stub 的地方，通过调用 create 方法，和我们
    *独立使用 HessianProxyFactory 是一样的。
    */
    //关于 HessianProxyFactory 的具体使用，可以参考 Hessian 的使用文档。
    protected Object createHessianProxy(HessianProxyFactory proxyFactory) throws
        MalformedURLException {
        Assert.notNull(getServiceInterface(), "'serviceInterface' is required");
        return proxyFactory.create(getServiceInterface(), getServiceUrl());
    }
```

这里使用的 HessianProxyFactory，是由 Hessian 提供的类，由它来具体完成通过 Hessian 的远端调用。在完成调用之前，已经通过 HessianProxyFactory 的 create 方法，实现了调用前的准备工作。这个准备工作，为远端对象创建了客户端的 Hession 的 Proxy。有了这个 Proxy，就可以像调用本地对象方法一样，调用这个 Proxy 的方法，中间的通信和交互过程都由 Hessian 封装好了。

7.3.3 Burlap 客户端的实现

Burlap 客户端的实现原理和 Hessian 客户端的实现原理是非常类似的。与 Hessian 一样，Spring 为 Burlap 的使用设计了 BurlapProxyFactoryBean，这个 BurlapProxyFactoryBean 的实现，如代码清单 7-19 所示。

代码清单 7-19 BurlapProxyFactoryBean 的实现

```
public class BurlapProxyFactoryBean extends BurlapClientInterceptor implements
    FactoryBean<Object> {
private Object serviceProxy;
@Override
public void afterPropertiesSet() {
    super.afterPropertiesSet();
    this.serviceProxy = new ProxyFactory(getServiceInterface(), this).getProxy
    (getBeanClassLoader());
}

public Object getObject() {
```

Download at http://www.pin5i.com/

```
        return this.serviceProxy;
    }
        Public class<?>getobjectType(){
        return getServiceInterface();
    }
    public boolean isSingleton() {
        return true;
    }
    }
```

在 BurlapProxyFactoryBean 中，除了使用 BurlapClientInterceptor 作为代理对象的拦截器之外，其他的实现和 HessianProxyFactoryBean 的实现比较基本上是完全一样的。

在代码实现中，BurlapClientInterceptor 的实现中对 Burlap 使用的封装是非常漂亮的，这个过程和使用 Hessian 非常类似，如代码清单 7-20 所示。

代码清单 7-20　BurlapInterceptor 的实现

```
public class BurlapClientInterceptor extends UrlBasedRemoteAccessor implements
    MethodInterceptor {
//这里创建proxyFactory，它是BurlapProxyFactory对象。
private BurlapProxyFactory proxyFactory = new BurlapProxyFactory();
private Object burlapProxy;
public void setProxyFactory(BurlapProxyFactory proxyFactory) {
    this.proxyFactory=(proxyFactory!=
    null?proxyFactory:new BurlapProxyFactory());
}
/**
*可以为BurlapProxyFactory设置属性，比如username和passwd，
*对于这些属性的具体含义，可以参考Burlap的使用文档。
*/
public void setUsername(String username) {
    this.proxyFactory.setUser(username);
}
public void setPassword(String password) {
    this.proxyFactory.setPassword(password);
}
public void setOverloadEnabled(boolean overloadEnabled) {
    this.proxyFactory.setOverloadEnabled(overloadEnabled);
}

//这里为使用BurlapProxyFactory做准备。
public void afterPropertiesSet() {
    super.afterPropertiesSet();
    prepare();
}
public void prepare() throws RemoteLookupFailureException {
    try {
        this.burlapProxy = createBurlapProxy(this.proxyFactory);
    }
    catch (MalformedURLException ex) {
        throw new RemoteLookupFailureException("Service URL [" + getServiceUrl()
        + "] is invalid", ex);
    }
}

/**
*通过BurlapProxyFactory的create为客户端创建stub对象，这个stub封装了对远端
```

Download at http://www.pin5i.com/

```
    *对象的调用，是一个非常标准的 Proxy 模式的使用例子。
    */
    protected Object createBurlapProxy(BurlapProxyFactory proxyFactory) throws
        MalformedURLException {
        Assert.notNull(getServiceInterface(),"Property'serviceInterface'is required");
        return proxyFactory.create(getServiceInterface(), getServiceUrl());
    }

    //Proxy 代理类对方法调用的入口，封装了对 Burlap 远端对象的调用。
    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (this.burlapProxy == null) {
            throw new IllegalStateException("BurlapClientInterceptor is not properly
            initialized - " +
            "invoke 'prepare' before attempting any operations");
        }

        ClassLoader originalClassLoader = overrideThreadContextClassLoader();

        try {//这里调用 burlapProxy 的方法，它是经过 burlap 封装的远端对象的 stub 方法。
            return invocation.getMethod().invoke(this.burlapProxy,invocation.getArguments());
        }
        catch (InvocationTargetException ex) {
            if (ex.getTargetException() instanceof BurlapRuntimeException) {
                BurlapRuntimeException bre = (BurlapRuntimeException) ex.getTargetException();
                Throwable rootCause = (bre.getRootCause() != null ? bre.getRootCause() : bre);
                 throw convertBurlapAccessException(rootCause);
            }
            else if (ex.getTargetException() instanceof UndeclaredThrowableException) {
                 UndeclaredThrowableException utex = (UndeclaredThrowableException)
                 ex.getTargetException();
                 throw convertBurlapAccessException(utex.getUndeclaredThrowable());
            }
            throw ex.getTargetException();
        }
        catch (Throwable ex) {
            throw new RemoteProxyFailureException(
           "Failed to invoke Burlap proxy for remote service["+getServiceUrl()+"]",ex);
        }
        finally {
            resetThreadContextClassLoader(originalClassLoader);
        }
    }

    protected RemoteAccessException convertBurlapAccessException(Throwable ex) {
        if (ex instanceof ConnectException) {
            return new RemoteConnectFailureException(
                    "Cannot connect to Burlap remote service at["+getServiceUrl()+"]", ex);
        }
        else {
            return new RemoteAccessException(
                "Cannot access Burlap remote service at["+getServiceUrl() + "]", ex);
        }
    }

    }
```

上面分析 Spring 驱动 Hessian/Burlap 在客户端实现远端调用的基本原理，作为远端调用实现的一部分，Spring 充分利用了 Hessian/Burlap 的已有特性，灵活地使用 Proxy 代理对象和

Download at http://www.pin5i.com/

拦截器，为远端对象创建了本地的 Proxy，有了这些客户端 Proxy，就可以通过 Hessian/Burlap 透明地完成整个远端调用，应用不需要关心具体的通信和交互过程，也能借助 Spring 提供的远端调用基础设施完成远端调用过程。在这个过程中，值得注意的是，在客户端使用远端调用的时候，需要和定义好的服务器端导出服务相配合。这就涉及 Spring 对 Hessian/Burlap 服务器端服务的导出，以及具体的服务实现，在了解了客户端的基本实现以后，下面我们就去看看 Spring Hessian/Burlap 在服务器端的实现原理。

7.3.4 Hessian/Burlap 服务器端的配置

和前面一样，我们从了解 Spring Hessian/Burlap 远端调用的服务器端配置入手，在应用使用 Hessian/Burlap 远端调用时，通常需要在 IoC 容器中，对服务的导出进行配置，如代码清单 7-21 所示。可以看到，这些配置与 HTTP 调用器的服务器端配置非常类似。在配置中，同样需要为相应的 ServiceExporter 配置服务 URL 地址、远端服务实现以及服务接口定义等基本的属性信息。

代码清单 7-21 Hessian/Burlap 的服务器端设置

```
<bean name="/serviceURL"class="org.springframework.remoting.caucho.HessianServiceExporter">
<property name="service">
    <ref bean="sevice"/>
</property>
<property name="serviceInterface">
    <value>yourInterface</value>
</property>
</bean>
<bean name="/serviceURL"class="org.springframework.remoting.caucho.BurlapServiceExporter">
<property name="service">
    <ref bean="sevice"/>
</property>
<property name="serviceInterface">
    <value>yourInterface</value>
</property>
</bean>
```

从配置中可以看到，服务的导出是通过对应的 ServiceExporter 完成的，在这些 ServiceExporter 中，完成了对 Hessina/Burlap 服务器端的封装处理。

7.3.5 Hessian 服务器端的实现

我们到 HessianServiceExporter 中，去看看它们是怎样通过 HessianServiceExporter 完成服务器端的服务导出的，如代码清单 7-22 所示。在这个类的实现中，可以看到它完成的基本上是桥梁工作，通过这个桥梁，HessianServiceExporter 把远端服务整合到 Spring MVC 框架中去，将提供的服务封装到 HttpRequestHandler 的 handleRequest 方法中去完成， 从而借助 Spring MVC 的实现完成了服务在服务器端的导出。

代码清单 7-22 HessianServiceExporter 的实现

```
public class HessianServiceExporter extends HessianExporter implements HttpRequestHandler{

//Processes the incoming Hessian request and creates a Hessian response.
```

Download at http://www.pin5i.com/

```
/**
* HessianServiceExporter 实际上是一个 Spring MVC 框架里的 Controller，通过 handleRequest
* 完成对 Request 的响应，将得到的服务执行结果输出到 response 中去。这里涉及 Spring MVC
* 的实现，关于 MVC 的具体实现原理，可以参考本书的第 4 章。
*/
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    if (!"POST".equals(request.getMethod())) {
        throw new HttpRequestMethodNotSupportedException(request.getMethod(),
                new String[]{"POST"},"HessianServiceExporter only supports POST requests");
    }
    //设置 response 的输出类型。
    response.setContentType(CONTENT_TYPE_HESSIAN);
    try {//这是对服务器端的远端对象方法的调用。
      invoke(request.getInputStream(), response.getOutputStream());
    }
    catch (Throwable ex) {
     throw new NestedServletException("Hessian skeleton invocation failed", ex);
    }
}
}
```

从代码清单 7-22 中可以看到，在准备好了 HTTP 的 Response 之后，服务器端对服务的调用和执行结果的返回都是由 invoke 方法来完成的，invoke 方法是在 HessianServiceExporter 的基类 HessianExporter 中实现的。在得到了从 HTTP 请求中获取的输入流，以及从 HTTP 响应中获取的输出流对象之后，通过调用 doinvoke 方法来完成服务的执行。在这个 doinvoke 方法的实现中，我们可以看到，最终是通过对 Hessian 的使用来完成服务的封装和执行的。在这个实现过程中，由于 Hessian 有不同的实现版本，在使用的时候为了保证与客户端使用的协议的兼容性，需要在 Hessian 的服务器端对客户端使用的 Hessian 版本进行判断，然后根据不同的版本使用情况，进行相应的选择和处理。以上的一系列实现过程，如代码清单 7-23 所示。

代码清单 7-23 HessianExporter 的 invoke

```
//这里创建 HessianSkeleton，它是 Hessian 完成服务器端服务的 Proxy 类。
public void prepare() {
    checkService();
    checkServiceInterface();
    /**
    *getProxyForService 读取 BeanDefinition 的设置，生成 Proxy 对象，这个 Proxy 对象的
    *target 设置成 service 属性定义的 bean，这个 bean 是具体完成远端服务的对象。最终，这个
    *Proxy 的生成过程在 RemoteExpoerter 类中完成。
    */
    this.skeleton=new HessianSkeleton(getProxyForService(),getServiceInterface());
}

public void invoke(InputStream inputStream,OutputStream outputStream)throws Throwable{
    Assert.notNull(this.skeleton, "Hessian exporter has not been initialized");
    ClassLoader originalClassLoader = overrideThreadContextClassLoader();
    try {
        doInvoke(inputStream, outputStream);
    }
    finally {
        resetThreadContextClassLoader(originalClassLoader);
    }
}
```

Download at http://www.pin5i.com/

```
public void doInvoke(final InputStream inputStream,final OutputStream outputStream)
     throws Throwable {
     InputStream isToUse = inputStream;
     OutputStream osToUse = outputStream;
     if (this.debugLogger != null && this.debugLogger.isDebugEnabled()) {
         PrintWriter debugWriter=
         new PrintWriter(new CommonsLogWriter(this.debugLogger));
         isToUse = new HessianDebugInputStream(inputStream, debugWriter);
         osToUse = new HessianDebugOutputStream(outputStream, debugWriter);
     }
     int code = isToUse.read();
     int major;
     int minor;
     AbstractHessianInput in;
     AbstractHessianOutput out;
     //判断客户端 Hessian 的版本，使用不同的 AbstractHessianInput 和 AbstractHessianOutput.
     if (code == 'H') {
         major = isToUse.read();
         minor = isToUse.read();
         if (major != 0x02) {
             throw new IOException("Version"+major+"."+minor+"is not understood");
         }
         in = new Hessian2Input(isToUse);
         out = new Hessian2Output(osToUse);
         in.readCall();
     }
     else if (code == 'c') {
         major = isToUse.read();
         minor = isToUse.read();
         in = new HessianInput(isToUse);
         if (major >= 2) {
             out = new Hessian2Output(osToUse);
         }
         else {
             out = new HessianOutput(osToUse);
         }
     }
     else {
         throw new IOException("Expected 'H' (Hessian 2.0) or 'c' (Hessian 1.0)
     in hessian input at " + code);
     }

     if (this.serializerFactory != null) {
         in.setSerializerFactory(this.serializerFactory);
         out.setSerializerFactory(this.serializerFactory);
     }
     //调用 HessianSkeleton 完成服务，关于 HessianSkeleton 的使用，可以参考 Hessian 的使用说明。
     try {
         this.skeleton.invoke(in, out);
     }
     finally {
         try {
             in.close();
             isToUse.close();
         }
         catch (IOException ex) {
             // ignore
         }
         try {
```

Download at http://www.pin5i.com/

```
                out.close();
                osToUse.close();
            }
            catch (IOException ex) {
                // ignore
            }
        }
    }
```

我们可以看到，这里使用了 Hessian 提供的 HessianSkeleton 来完成服务的提供。为了使用 HessianSkeleton，需要做一些准备工作，Spring 已经为用户封装好了这些配置，有了这一层的封装，比用户直接使用 Hessian 来完成远端调用要方便许多。

7.3.6 Burlap 服务器端的实现

Burlap 服务器端的封装和使用的实现方式也与 Hessian 非常相似，它也是通过 BurlapServiceExporter 来完成远端服务的导出的，如代码清单 7-24 所示。BurlapServiceExporter 是 Spring MVC 的一个 Controller，负责接受 HTTP 请求，并在 HttpRequestHandler 的 handleRequest 方法中启动对服务方法的调用。

代码清单 7-24 BurlapServiceExporter 的实现

```
public class BurlapServiceExporter extends BurlapExporter implements
    HttpRequestHandler {

//Processes the incoming Burlap request and creates a Burlap response.
/**
*作为一个 Spring MVC 的 Controller，在 handleRequest 方法中实现对服务请求的响应。在实现中，
*调用 invoke 方法调用来完成对 serviceURL 的服务请求的处理。
*/
public void handleRequest(HttpServletRequest request,HttpServletResponse response)
        throws ServletException, IOException {
//Burlap 只接受 POST 的 HTTP 请求。
    if (!"POST".equals(request.getMethod())) {
        throw new HttpRequestMethodNotSupportedException(request.getMethod(),
        new String[]{"POST"},"BurlapServiceExporter only supports POST requests");
    }
    try {
      invoke(request.getInputStream(), response.getOutputStream());
    }
    catch (Throwable ex) {
      throw new NestedServletException("Burlap skeleton invocation failed", ex);
    }
}
}
```

Exporter 是 MVC 的一个 Controller，从 MVC 框架中可以取得 HTTP 请求和响应，然后把这些对象作为参数传递给 Burlap。这个过程是通过调用 invoke 方法来完成的。这个 invoke 方法是在 BurlapExporter 中实现的，如代码清单 7-25 所示。在 BurlapExporter 对象中，定义了 Burlap 封装服务器端服务的 BurlapSkeleton 对象，并在 afterPropertiesSet 中，同样也为服务的导出做好了准备。这些准备包括对服务导出的配置进行检查以及 BurlapSkeleton 对象的创建等。准备工作完成以后，Spring 的 Burlap 服务器端就开始等待服务请求的到来。客户端发出的服务请求，在由 BurlapServiceExporter 完成接收以后，服务的执行转而由 BurlapExporter 的 invoke 方法来完成。这些在 invoke 方法的实现中都可以看

Download at http://www.pin5i.com/

到，接着，它会启动 Burlap 的 Skeleton 对象来完成服务执行以及执行结果返回。最后，在完成服务响应以后，关闭 HTTP 响应的 IO 流。

代码清单 7-25 BurlapExporter 的实现

```
public class BurlapExporter extends RemoteExporter implements InitializingBean {
//定义的 BurlapSkeleton，实现 Burlap 服务的 Proxy 对象。
private BurlapSkeleton skeleton;
//在依赖注入完成后初始化 Burlap 的服务器端配置。
public void afterPropertiesSet() {
    prepare();
}
/**
*创建 BurlapSkeleton 对象，该对象封装了提供远端调用的服务对象，这个服务对象作为 target 封装在
*一个 Proxy 对象中，这个 Proxy 对象由 getProxyForService 方法取得，这个 getProxyForService
*方法具体实现在 RemoteExporter 中，负责读取 BeanDefinition 中对 BurlapExporter 的配置。
*/
public void prepare() {
    checkService();
    checkServiceInterface();
    this.skeleton = new BurlapSkeleton(getProxyForService(),getServiceInterface());
}
//直接调用 BurlapSkeleton 来提供远端服务，具体的通信协议处理和服务调用过程已经被 Burlap 封装了。
public void invoke(InputStream inputStream,OutputStream outputStream) throws Throwable{
    Assert.notNull(this.skeleton, "Burlap exporter has not been initialized");
    ClassLoader originalClassLoader = overrideThreadContextClassLoader();
    try {
        this.skeleton.invoke(new BurlapInput(inputStream),new BurlapOutput(outputStream));
    }
    finally {
        try {
            inputStream.close();
        }
        catch (IOException ex) {
            // ignore
        }
        try {
            outputStream.close();
        }
        catch (IOException ex) {
            // ignore
        }
        resetThreadContextClassLoader(originalClassLoader);
    }
}
}
```

Burlap 服务器端的服务过程与使用 Hessian 的服务器端的服务过程非常类似，它们都是通过配置 ServiceExporter 来完成的，让我们大致回顾一下它们的工作原理。在 ServiceExporter 中，首先，通过 RemoteExporter 的 getProxyForService 取得封装了服务对象的 Proxy 对象。这个过程在 Hessian 和 Burlap 中都是一样的，都是在 IoC 容器对 Bean 对象的依赖注入完成以后，通过实现 InitializingBean 的 afterPropertiesSet 方法来完成的，这个后置处理主要完成的是对 Skeleton 对象的配置工作。这个 Skeleton 对象是 Hessian/Burlap 实现远端服务服务器端的基础设施，用来为客户端提供服务请求的响应。在 Skeleton 准备好以后，就可以直接使用 Hessian 和 Burlap 了，这与我们脱离 Spring 应用环

Download at http://www.pin5i.com/

境，独立使用 Hessian/Burlap 是一样的。不过，通过 Spring 的封装，让习惯了 IoC 容器配置的用户，使用这两个第三方类库更方便了。在使用中，用户只需要对远端服务进行配置，就可以实现远端服务的即开即用，方便程度类似于在 Spring 中事务处理的声明式使用，有了 Spring 的远端调用模块，为应用开发打点好了许多远端调用的实现细节。

7.4 Spring RMI 的实现

与我们前面看到的其他远端调用方案实现一样，Spring 通过对 IoC 容器和 AOP 的使用，为用户提供了基于 RMI 机制的远端调用服务。在使用 RMI 实现远端调用服务的时候，它的网络通信实现是基于 TCP/IP 协议完成的，而不是通过前面我们看到的 HTTP 协议，从而完成数据的通信传输。在 Spring 的 RMI 实现中，集成了标准的 RMI-JRMP 解决方案，这个方案是 Java 虚拟机实现的一部分，它使用 Java 序列化来完成对象的传输，是一个 Java 到 Java 环境的分布式处理技术，因而不涉及异构平台的处理。Spring 在封装 RMI 远端调用服务的时候，支持传统的 RMI 实现方式，但在这个传统实现方式的基础上，还提供了 RMI 调用器的实现方案作为一个简化方案。这个 RMI 调用器的实现方式，就像我们前面看到的 HTTP 调用器的实现那样，通过使用普通的 Java 业务接口，就能够提供远端服务，并不需要实现传统 RMI 需要的 Remote 接口，使用起来也非常方便。

7.4.1 Spring RMI 客户端的配置

在使用 Spring RMI 的时候，毫不例外，仍然需要对客户端和服务器端进行相应的配置，我们先从客户端的配置入手，如代码清单 7-26 所示。然后和以前的分析一样，从客户端的实现开始，再接着分析服务器端的实现。在 RMI 客户端的配置中，沿袭了 Spring 远端调用方案的一致风格，比如，可以再一次看到对 ProxyFactoryBean 的使用，只是在 RMI 客户端的配置中，配置的 ProxyFactoryBean 被换成了 RMIProxyFactoryBean 而已，它的配置也与其他 Spring 远端调用方案非常类似，只是在配置 serviceUrl 时，需要使用 RMI 作为协议，以及与此相对应的的通信端口，在这里使用 1099 端口，从而通过 TCP/IP 完成数据的通信和交互，这些不同是很好理解的，因为在底层的通信支持上，RMI 采用了不同的协议和实现方式。

代码清单 7-26 Spring RMI 的客户端配置

```
<bean id="rmiProxy"class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
<property name="serviceUrl">
    <value>rmi://YourHostName:1099/YourService</value>
</property>
<property name="serviceInterface">
    <value>YourServiceInterface</value>
</property>
</bean>
<bean id ="rmiClient" class="yourClass">
<property name="YourServiceInterface">
    <ref bean="rmiProxy">
</property>
</bean>
```

Download at http://www.pin5i.com/

7.4.2 Spring RMI 客户端的实现

在 Spring RMI 客户端中，使用的仍然是我们已经非常熟悉的 ProxyFactoryBean，即 RmiProxyFactoryBean。它完成的功能是对 RMI 客户端的封装，比如，生成代理对象、查询得到 RMI 的 stub 对象，并通过这个 stub 对象发起相应的 RMI 远端服务请求，如代码清单 7-27 所示。

代码清单 7-27　RmiProxyFactoryBean 的实现

```
public class RmiProxyFactoryBean extends RmiClientInterceptor implements
    FactoryBean<Object>, BeanClassLoaderAware {
//通过 ProxyFactory 生成的代理对象，代理对象的代理方法和拦截器都会在其生成时被设置好。
private Object serviceProxy;
@Override
/**
* 在依赖注入完成以后，容器回调 afterPropertiesSet，通过 ProxyFactory 生成代理对象，
* 这个代理对象的拦截器是 RmiClientInterceptor。
*/
public void afterPropertiesSet() {
    super.afterPropertiesSet();
    if (getServiceInterface() == null) {
        throw new IllegalArgumentException("Property 'serviceInterface' is required");
    }
    this.serviceProxy = new ProxyFactory(getServiceInterface(), this).getProxy
    (getBeanClassLoader());
}
//FactoryBean 的接口方法，返回生成的代理对象 serviceProxy。
public Object getObject() {
    return this.serviceProxy;
}
public Class<?> getObjectType() {
    return getServiceInterface();
}
public boolean isSingleton() {
    return true;
}
}
```

RMI 客户端基础设施的封装，是由拦截器 RmiClientInterceptor 来完成的，这个拦截器的设置，是在 RmiProxyFactoryBean 生成的代理对象中完成的。在拦截器中，我们首先看到的是对 stub 对象的获取，作为实现 RMI 的基本准备，这个获取 stub 的实现是在拦截器的 afterPropertiesSet 方法中完成的。在实现中，我们看到 Spring 还为这个 stub 对象提供了缓存，从而提高对它的性能。从 Spring 的代码实现中，可以看到拦截器获取 stub 的实现如代码清单 7-28 所示。

代码清单 7-28　拦截器 RmiClientInterceptor 获取 stub 的实现

```
/**
* 建立 RMI 基础设施的调用，仍然是在 afterPropertiesSet 方法中实现。这个 RmiClientInterceptor
* 实现了 InitializingBean 接口，所以会被 IoC 容器回调。
*/
public void afterPropertiesSet() {
    super.afterPropertiesSet();
    prepare();
}

//Fetches RMI stub on startup, if necessary.
```

Download at http://www.pin5i.com/

```
/**
* 这里为RMI客户端准备stub，这个stub通过lookupStub方法获得，并且会在第一次生成之后，放到缓存中去。
*/
public void prepare() throws RemoteLookupFailureException {
    // Cache RMI stub on initialization?
    if (this.lookupStubOnStartup) {
        Remote remoteObj = lookupStub();
        if (logger.isDebugEnabled()) {
            if (remoteObj instanceof RmiInvocationHandler) {
                logger.debug("RMI stub [" + getServiceUrl() + "] is an RMI invoker");
            }
            else if (getServiceInterface() != null) {
                boolean isImpl = getServiceInterface().isInstance(remoteObj);
                logger.debug("Using service interface [" + getServiceInterface().getName() +
                    "] for RMI stub [" + getServiceUrl() + "] - " +
                    (!isImpl ? "not " : "") + "directly implemented");
            }
        }
        if (this.cacheStub) {
            this.cachedStub = remoteObj;
        }
    }
}
/**
 * Create the RMI stub, typically by looking it up.
 * <p>Called on interceptor initialization if "cacheStub" is "true";
 * else called for each invocation by {@link #getStub()}.
 * <p>The default implementation looks up the service URL via
 * <code>java.rmi.Naming</code>. This can be overridden in subclasses.
 */
//获得RMI stub对象的地方。
protected Remote lookupStub() throws RemoteLookupFailureException {
    try {
        Remote stub = null;
        if (this.registryClientSocketFactory != null) {
            /**
            *RMIClientSocketFactory specified for registry access.
            *Unfortunately, due to RMI API limitations, this means
            *that we need to parse the RMI URL ourselves and perform
            *straight LocateRegistry.getRegistry/Registry.lookup calls.
            */
            URL rul = new URL(null, getServiceUrl(), new DummyURLStreamHandler());
            String protocol = rul.getProtocol();
            if (protocol != null && !"rmi".equals(protocol)) {
                throw new MalformedURLException("Invalid URL scheme'"+protocol
                +"'");
            }
            String host = url.getHost();
            int port = url.getPort();
            String name = url.getPath();
            if (name != null && name.startsWith("/")) {
                name = name.substring(1);
            }
            Registry registry = LocateRegistry.getRegistry(host, port,
            this.registryClientSocketFactory);
            stub = registry.lookup(name);
        }
        else {
            // Can proceed with standard RMI lookup API...
```

Download at http://www.pin5i.com/

```
            stub = Naming.lookup(getServiceUrl());
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Located RMI stub with URL [" + getServiceUrl() + "]");
        }
        return stub;
    }
    catch (MalformedURLException ex) {
        throw new RemoteLookupFailureException("Service URL [" + getServiceUrl()
        "] is invalid", ex);
    }
    catch (NotBoundException ex) {
        throw new RemoteLookupFailureException(
                "Could not find RMI service [" + getServiceUrl() + "] in RMI
                registry", ex);
    }
    catch (RemoteException ex) {
        throw new RemoteLookupFailureException("Lookup of RMI stub failed", ex);
    }
}
```

获取了 stub 之后，当 RMI 客户端的代理方法被调用时，会触发拦截器 RmiClientInterceptor 的 invoke 回调方法。invoke 回调方法的实现，如代码清单 7-29 所示。在 invoke 回调中，可以看到 RMI 远端调用的发生，其中 Spring 采用了两种方式，一种是使用 RMI 调用器的方式，这种方式和 HTTP 调用器的使用非常类似，是通过一个自定义的 RemoteInvocation 来封装调用的场景的，有了这层封装，相当于为 RMI 的远程调用设计了一个 RMI 调用器，来简化通过 RMI 的远端调用实现；另一种是使用传统的 RMI 远端调用方式的实现，这是使用 RMI 的读者非常熟悉的。

代码清单 7-29 RmiClientInterceptor 的 invoke

```
// 拦截器对代理对象方法调用的回调，在实现中取得 RMI 的 stub 对象，然后调用 doInvoke 完成 RMI 调用。
public Object invoke(MethodInvocation invocation) throws Throwable {
    Remote stub = getStub();
    try {
        return doInvoke(invocation, stub);
    }
    catch (RemoteConnectFailureException ex) {
        return handleRemoteConnectFailure(invocation, ex);
    }
    catch (RemoteException ex) {
        if (isConnectFailure(ex)) {
            return handleRemoteConnectFailure(invocation, ex);
        }
        else {
            throw ex;
        }
    }
}
/**
* 具体的 RMI 调用发生的地方，如果 stub 是 RmiInvocationHandler 实例，那么使用 RMI 调用器
* 来完成这次远端调用，否则，使用传统的 RMI 调用方式。
*/
protected Object doInvoke(MethodInvocation invocation, Remote stub) throws Throwable {
    if (stub instanceof RmiInvocationHandler) {
        // RMI invoker
        try {
            return doInvoke(invocation, (RmiInvocationHandler) stub);
```

Download at http://www.pin5i.com/

```
            }
            catch (RemoteException ex) {
                throw RmiClientInterceptorUtils.convertRmiAccessException(
                    invocation.getMethod(), ex, isConnectFailure(ex), getServiceUrl());
            }
            catch (InvocationTargetException ex) {
                Throwable exToThrow = ex.getTargetException();
                RemoteInvocationUtils.fillInClientStackTraceIfPossible(exToThrow);
                throw exToThrow;
            }
            catch (Throwable ex) {
                throw new RemoteInvocationFailureException("Invocation of method ["
                + invocation.getMethod() +
                "] failed in RMI service [" + getServiceUrl() + "]", ex);
            }
        }
        else {
            // traditional RMI stub
            try {
                return RmiClientInterceptorUtils.invokeRemoteMethod(invocation, stub);
            }
            catch (InvocationTargetException ex) {
                Throwable targetEx = ex.getTargetException();
                if (targetEx instanceof RemoteException) {
                    RemoteException rex = (RemoteException) targetEx;
                    throw RmiClientInterceptorUtils.convertRmiAccessException(
                    invocation.getMethod(), rex, isConnectFailure(rex), get
                    ServiceUrl());
                }
                else {
                    throw targetEx;
                }
            }
        }
    }
```

这里可以看到整个 RMI 客户端的实现原理，以及 AOP 的灵活运用。同时，Spring 的 RMI 远端调用解决方案也有它自身的特点。在它的实现中，可以看到它不但兼容了传统的 RMI 远端调用方式，而且在 Java RMI 实现基础上，还做了一层 RMI 调用器的封装，这种 RMI 调用器的实现，使得用户在通过 RMI 实现远端调用的时候，使用方式上更加灵活了。

7.4.3 Spring RMI 服务器端的配置

了解了 Spring RMI 客户端的配置和实现原理，下面我们从了解 Spring RMI 的服务器端的设置开始去了解它的实现。Spring RMI 服务器端的设置的一个简要实例，如代码清单 7-30 所示。在 RMI 中，基于 TCP/IP 协议，而不是 HTTP 来实现基本的网络通信，RMI 的网络通信是 Java RMI 实现的一部分，所以这里不需要使用 Spring MVC 的 `DispatcherServlet` 来配置远端服务的服务 URL 以及控制服务请求的转发。在服务器端的配置中，除了需要指定提供服务的 Bean，以及代理的服务接口之外，还需要通过 `servcieName` 属性来配置服务的导出位置，同时使用 `registryPort` 来指定 RMI 监听的 TCP/IP 端口。

Download at http://www.pin5i.com/

代码清单 7-30 Spring RMI 服务器端设置

```
<bean id="rmiService" class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service">
        <ref bean="yourServiceBean">
    </property>
    <property name="serviceInterface">
        <value>YourServiceInterface</value>
    </property>
    <property name="serviceName">
        <value>yourService</value>
    </property>
    <property name="registryPort">
        <value>1099</value>
    </property>
</bean>
```

有了这个配置，可以看到 Spring RMI 服务器端的基础设施是由 RmiServiceExporter 来实现的，这个核心的实现类，是我们下面分析 Spring RMI 服务器端实现原理的重点。

7.4.4 Spring RMI 服务器端的实现

在 Spring RMI 服务器端，通过 RmiServiceExporter 来导出 RMI 服务，我们看看这个类的具体实现，如代码清单 7-31 所示。在 RMI 的导出器中，建立 RMI 服务器端的实现，主要集中在 prepare 方法中，这个方法在 afterPropertiesSet 被调用，对于 afterPropertiesSet 的使用，我们已经很熟悉了。在建立 RMI 服务器端导出服务基础设施的过程中，首先需要对一些基本的参数设置进行检查，比如是否设置了服务提供 Bean，是否设置了服务位置，等等。接着，在正确取得服务对象的基础上，导出器会把这个服务对象通过 RMI 机制导出，然后在完成 RMI 注册以后，客户端就可以开始查询和使用 RMI 远端服务了。

代码清单 7-31 RmiServiceExporter 的实现

```
//在 afterPropertiesSet 方法中，开始建立 RMI 服务器端的基础设施，在 prepare 方法中实现。
public void afterPropertiesSet() throws RemoteException {
    prepare();
}
/**
 * Initialize this service exporter, registering the service as RMI object.
 * <p>Creates an RMI registry on the specified port if none exists.
 * @throws RemoteException if service registration failed.
 */
public void prepare() throws RemoteException {
     // 检查提供服务的 Bean 和 serviceName 属性是否设置正确，如果没有设置，抛出异常。
     checkService();
     if (this.serviceName == null) {
        throw new IllegalArgumentException("Property 'serviceName' is required");
     }
     // Check socket factories for exported object.
     if (this.clientSocketFactory instanceof RMIServerSocketFactory) {
         this.serverSocketFactory = (RMIServerSocketFactory) this.clientSocketFactory;
     }
     if ((this.clientSocketFactory != null && this.serverSocketFactory == null) ||
             (this.clientSocketFactory == null && this.serverSocketFactory != null)) {
         throw new IllegalArgumentException(
         "Both RMIClientSocketFactory and RMIServerSocketFactory or none
         required");
```

Download at http://www.pin5i.com/

```
        }

        // Check socket factories for RMI registry.
        if (this.registryClientSocketFactory instanceof RMIServerSocketFactory) {
            this.registryServerSocketFactory =
        (RMIServerSocketFactory) this.registry ClientSocketFactory;
        }
        if (this.registryClientSocketFactory == null &&
        this.registryServerSocket Factory != null) {
            throw new IllegalArgumentException(
                    "RMIServerSocketFactory without RMIClientSocketFactory for registry
                    not supported");
        }
        // Determine RMI registry to use.
        if (this.registry == null) {
            this.registry = getRegistry(this.registryHost, this.registryPort,
                this.registryClientSocketFactory, this.registryServerSocketFactory);
        }
        // Initialize and cache exported object.
        /**
        * 这里是取得服务对象的地方，需要根据服务的设置来完成服务对象的获取，如果服务对象实现了 Java
        * 的 Remote 接口，那么取得的是标准的 RMI 服务，否则，使用 RMI 调用器。
        */
        this.exportedObject = getObjectToExport();
        if (logger.isInfoEnabled()) {
            logger.info("Binding service '" + this.serviceName + "' to RMI registry:
            " + this.registry);
        }
        // Export RMI object.
        if (this.clientSocketFactory != null) {
            UnicastRemoteObject.exportObject(
                this.exportedObject, this.servicePort, this.clientSocketFactory,
                this.serverSocketFactory);
        }
        else {
            UnicastRemoteObject.exportObject(this.exportedObject, this.servicePort);
        }
        // Bind RMI object to registry.
        // 把 RMI 服务对象和注册器绑定，供客户端查询。
        try {
            if (this.replaceExistingBinding) {
                this.registry.rebind(this.serviceName, this.exportedObject);
            }
            else {
                this.registry.bind(this.serviceName, this.exportedObject);
            }
        }
        catch (AlreadyBoundException ex) {
            // Already an RMI object bound for the specified service name...
            unexportObjectSilently();
            throw new IllegalStateException(
                    "Already an RMI object bound for name '" + this.serviceName + "':
                    " + ex.toString());
        }
        catch (RemoteException ex) {
            // Registry binding failed: let's unexport the RMI object as well.
            unexportObjectSilently();
            throw ex;
        }
    }
```

Download at http://www.pin5i.com/

7.5 小结

本章对 Spring 远端调用模块的实现原理进行了详细的分析，选取了 Spring 已有的远端调用实现作为例子，分析了 HTTP 调用器、第三方远端协议 Hessian/Burlap 以及 RMI 远端调用的实现原理。前两种解决方案，采用的都是 HTTP 作为数据传输的协议，而 RMI 远端调用的实现是基于 TCP/IP 来完成的。我们看到，在前两者的实现中，使用了 HTTP 作为基本的通信协议是它们实现中相同的点，它们实现的不同点在于如何完成远端调用的通信封装上。在 HTTP 调用器的实现中，是通过 Spring 来进行封装的，它利用了现有 Java 虚拟机的特性，而不依赖于第三方的解决方案，使用 Java 对象的序列化和反序列化，从而完成远端调用过程的通信以及调用交互。对于 Spring 远端调用 Hessian/Burlap 的解决方案而言，通过 Hessian/Burlap 这两个第三方解决方案，封装了具体的通信过程，了解它们使用的读者知道，这两个协议处理方案，一个使用的是二进制协议（Hessian），另一个使用 XML 来进行通信的实现（Burlap）。在 Spring 通过它们实现远端调用时，是将这些通信过程交由 Hessian/Burlap 来处理的，而 Spring 只需要按照这两个第三方组件的要求，准备好相应的客户端 `stub` 和服务器端的 `skeleton` 就可以了。Spring 远端调用模块中，使用 RMI 为远端调用提供封装支持，其实现过程与其他远端调用的实现，有很多类似的地方，只不过在了解 RMI 的封装实现中，除了可以使用传统的 RMI 调用方式来完成调用以外，Spring 还给应用提供了一个 RMI 调用器的封装。

在具体的实现中，通过 IoC 容器用户可以声明式地定义远端调用，这在很大程度上便利了应用开发。Spring 通过使用 `Proxy` 代理对象和注入相应的拦截器，为用户实现使用远端调用模块的封装，用户需要使用远端调用功能的时候，只需要进行简单的配置就可以完成远端调用。这些配置由两部分组成：一部分在服务器端，主要是对服务导出进行配置，在这个服务导出中，需要对 `ServiceExporter` 类进行配置，比如定义提供服务的远端服务对象、提供服务的 URL 地址，等等；另一部分实现在客户端，需要对 `ProxyFactoryBean` 进行配置。在 Spring 远端调用模块中，由 `ServiceExporter` 和 `ProxyFactoryBean` 来封装远端调用客户端和服务器端的处理，比如对 HTTP 调用器、Hessian/Burlap 以及 RMI 远端调用这些远端调用方案，都可以看到 `ServiceExpoerter` 和 `ProxyFactory` 的活跃身影。在 Spring 中，可以看到一系列 `ServiceExporter` 和 `ProxyFactory` 的设计实现，它们与 HTTP 调用器、Hessian/Burlap、RMI 远端调用实现是一一对应的。从这点上看，在整个 Spring 远端调用模块中，其设计是非常整齐和规范的。

在本章中，希望通过对以上这些远端调用实现原理的分析，能够为读者更好地使用 Spring 远端调用方案、在选取和评估不同的远端调用方案的时候，提供有价值的参考，从而更好地满足应用的需求。

Download at http://www.pin5i.com/

第 8 章

安全框架 ACEGI 的实现

人间四月芳菲尽，山寺桃花始盛开。
长恨春归无觅处，不知转入此中来。
——【唐】白居易 《大林寺桃花》

8.1 Spring ACEGI 安全框架概述

8.1.1 概述

作为 Spring 丰富生态系统中的一个非常典型的应用，安全框架 Spring ACEGI 的使用是非常普遍的。尽管它不属于 Spring 平台的范围，但由于它建立在 Spring 的基础上，因此可以方便地与 Spring 应用集成，从而方便地为基于 Spring 的应用提供安全服务。

作为一个完整的 Java EE 安全应用解决方案，ACEGI 能够为基于 Spring 构建的应用项目提供全面的安全服务，它可以处理应用需要的各种典型的安全需求。例如，用户的身份验证、用户授权等。ACEGI 因为其优秀的实现，而被 Spring 开发团队推荐作为 Spring 应用的通用安全框架，随着 Spring 的广泛传播而被广泛应用。在各种有关 Spring 的书籍、文档和应用项目中，都可以看到它活跃的身影。

关于 ACEGI 项目的具体情况，可以到它的官方网站[⊖]去了解，在那里可以获得许多 ACEGI 的帮助文档和应用实例，因为它本身也是一个开源项目。对 ACEGI 的实现原理感兴趣的读者，也可以很方便的获得它的源代码，作为学习、使用和扩展 ACEGI 的一个基础。

通过前面几章的内容，我们已经大致掌握了 Spring 的基本实现原理，这些 Spring 的基本实现原理的掌握，已经为我们使用 ACEGI 框架打下了很好的基础。反过来说，由于 ACEGI 本身也是一个非常典型的 Spring 应用，了解它的实现原理，对我们开发其他 Spring 应用也具有很高的参考价值，对扩展我们使用 Spring 的视野也是非常有帮助的。

在深入了解 ACEGI 的实现原理之前，和前面我们学习 Spring 的实现一样，同样需要做一些必要的准备工作。首先，我们需要下载 ACEGI 的源代码，根据 ACEGI 网站的提示，其源代码的 SVN 路径是 http://ACEGIsecurity.svn.sourceforge.net/svnroot/ACEGIsecurity/spring-security。在本章中，我们检出的 ACEGI 的源代码的基线为 1.0.2，以这个版本的源代码作为研究对象来分析 ACEGI 的实现原理。使用 SVN 下载源代码的过程，和下载 Spring 源代码的过程是一样的，在本书的第 1 章中，已经详细介绍过获取源代码的过程，这里就不再赘述了。

⊖ http://www.acegisecurity.org/

Download at http://www.pin5i.com/

8.1.2　使用 Spring IDE

将源代码检出到 Eclipse 本地环境以后，熟悉 Spring 应用的读者一定会注意到，ACEGI 通过 Spring 的 IoC 容器，管理了许多自己在框架中实现的、为安全需求提供服务的 Bean 的配置，以及这些 Bean 之间的依赖关系。根据我们之前的分析经验，如果能够以一种直观的方式，去了解这些 Bean 的设置以及它们相互之间的依赖关系，对我们理解 ACEGI 的实现原理是非常有帮助的。在这方面，Spring 也是考虑得非常周到，它为 Spring 为开发人员提供了一个 Spring IDE 工具。通过这个工具的帮助，可以方便地管理 Spring 应用中所用到的 Bean 的配置，这是 Spring 开发者的得力助手，熟悉它的使用，无疑会让 Spring 的开发如虎添翼。下面，我们就简要地介绍一下这个 IDE 工具的使用，为读者提供一些参考。

具体来说，作为 Spring 的子项目，Spring IDE 是以 Eclipse 插件的形式与我们的应用开发环境集成在一起的，从而为 Spring 的应用开发提供有力的工具支持。关于 Spring IDE 的详细介绍，感兴趣的读者可以到它的官方网站[1]上去了解。如果需要在 Eclipse 里安装 Spring IDE，那么需要得到 Spring IDE 的更新网址，根据网站上的信息，当前的更新地址为 http://dist.springframework.org/release/IDE。具体的安装过程可以参考本书第 1 章，关于 Eclipse 插件安装的例子，在那个例子中，我们说明了 SVN 的 Eclipse 插件安装，详细介绍了 Eclipse 插件的安装过程。读者可以参考第 1 章的内容，来完成 Spring IDE 的安装，同时也可以把它当作一个安装 Eclipse 插件的小练习，在这里，我们就不详细描述这个插件的安装过程了，而把重点放在介绍这个插件的使用上。

Eclipse 的 Spring IDE 插件安装完成以后，可以在 Eclipse 中打开 Spring IDE 的相关视图，如图 8-1 所示：

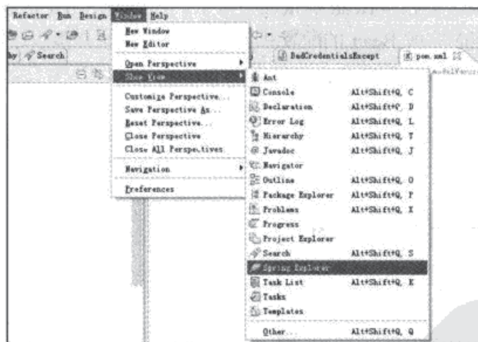


图 8-1　在 Eclipse 中打开 Spring IDE 视图

打开视图以后，可以看到 Spring Explorer 的视图。在使用 Spring IDE 之前，首先，需要把应用项目加入到 Spring IDE 中，从而把 Spring 项目加入到 Eclipse IDE 中进行管理，这个操作如图 8-2 所示。

[1] http://springide.org

Download at http://www.pin5i.com/

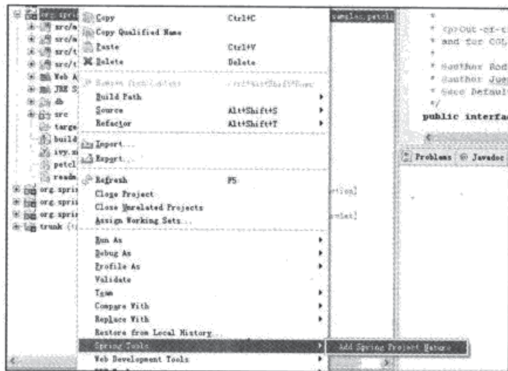


图 8-2 把项目加入到 Spring IDE 中

把项目添加到 Spring IDE 中以后，细心的读者一定会注意到，在 Eclipse 的 Explorer 视图的项目列表显示中，我们刚刚加入 Spring IDE 进行管理的项目的标示上，会有一个小小的“s”字母的标识，这个标识意味着我们已经成功地把项目加入到 Spring IDE 中进行管理了，这时，这个项目已经可以在 Spring IDE 的 Explorer 中进行 Bean 的管理了。做完这项基本的准备工作之后，要使用 IDE 工具完成 Spring Bean 的管理，我们还需要完成另外一项准备工作，即需要把项目中的 XML 配置文件加入到 Spring IDE 中去。这个操作完成的步骤如下：首先，打开 Spring Explorer，点击已经加入的 Spring 项目，右键打开 properties 的标签页；然后打开 Spring 下的 bean support，在 Config Files 中通过 Add 按钮加入 Bean 配置文件，如图 8-3 所示。在这两个准备工作完成以后，就可以使用 Spring IDE 来管理这些 XML 文件和 Bean 的依赖关系了。通过使用 IDE，我们可以用图形的方式来浏览 Bean 及其依赖关系，同时完成 Bean 的配置和 Java 源代码的相互索引。有了 Spring IDE 工具的支持，对我们管理 Spring 的 Bean 依赖关系是非常方便的。

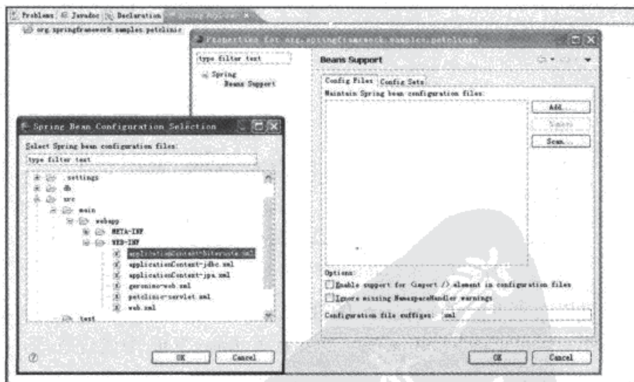


图 8-3 添加 Bean 配置文件

Download at http://www.pin5i.com/

在 Spring Explorer 中添加完 XML 配置文件后，可以在 Spring Explorer 里看到详细的 Bean 信息，而且 Explorer 还提供了方便的图形方式来展示 Bean 依赖关系，如图 8-4 所示。

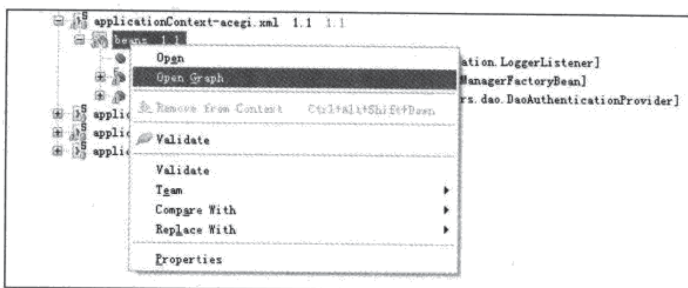


图 8-4 打开 Bean 的依赖图示

8.1.3 ACEGI 的 Bean 配置

有了工具的支持，下面我们来看看，使用 Spring ACEGI 的应用是如何对 ACEGI 进行配置的。使用图示的方式，可以给 Spring 应用开发带来很大的便利，使得对 Bean 的配置和管理更清晰明了，在 Spring IDE 中，我们可以看到 ACEGI 中实现的 Bean 和它们之间相互依赖关系的形象表现，如图 8-5 所示。

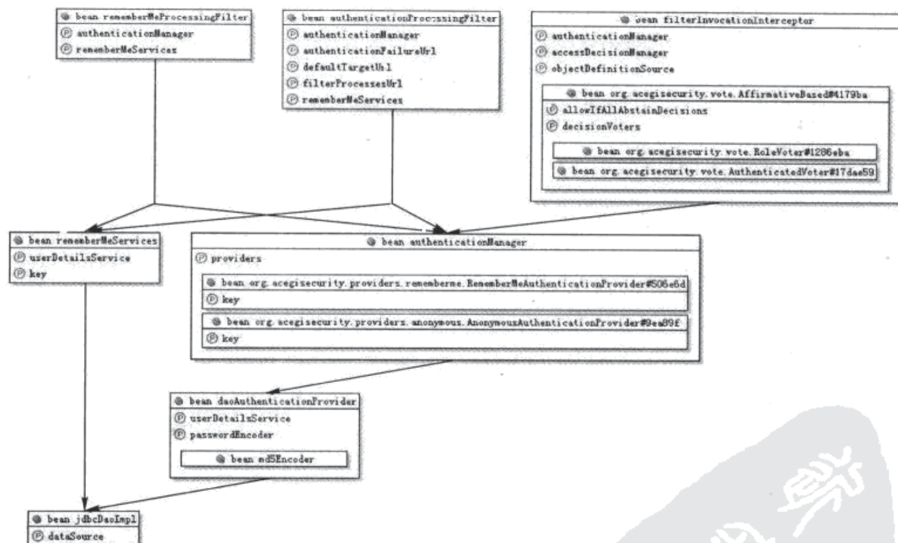


图 8-5 ACEGI 的 Bean 配置

Download at http://www.pin5i.com/

通过这张 Bean 配置图，我们可以看到安全框架实现的几种基本模块类型，ACEGI 框架通过这些基本模块的协作，为应用提供安全服务。在这些基本模块中，首先，我们看到的是各种 Filter（过滤器），ACEGI 通过这些过滤器与 ACEGI 的应用环境进行集成，并通过这些过滤器来为应用提供安全方面的功能增强；其次，是在验证过程中，起到核心作用的 AuthenticationManager 以及 AuthenticationProvider 的配置，它们是完成验证工作的主要实现，同时也是为验证提供用户数据的模块。作为 Spring 应用 Bean 的配置，它们都是通过使用 IoC 容器，从而完成自身提供的服务与平台的集成，在这方面它们与其他 Spring 应用在 IoC 容器中的配置相比，并没有太大的差别。

8.2 配置 Spring ACEGI

在说明 ACEGI 的实现原理的时候，我们从配置 Spring ACEGI 入手。举一个例子来说明 ACEGI 的实现，这个例子实现的是一个以 Web 表单登录为入口，从而为用户实现应用安全的工作，它的工作包括用户登录、授权等一些实现。在 Web 的应用开发中，这也是一个非常典型的安全应用。

和我们前面的分析习惯一样，首先，我们需要了解的是基于 Web 页面登录的过滤器配置，这个配置如代码清单 8-1 所示。通过配置使得这个 Web 页面过滤器会对登录请求进行拦截，并由配置来决定拦截的实现要求。可以在配置中看到，它指定了过滤器的拦截行为，以及实现表单登录的具体细节。这些具体的拦截器起作用的实现细节包括：使用什么样的验证器、验证失败后的页面跳转到的 URL 以及默认的目标 URL 等。

代码清单 8-1 Web 页面的登录过滤器

```
<bean id="authenticationProcessingFilter"
    class="org.ACEGIsecurity.ui.webapp AuthenticationProcessingFilter">
    <property name="authenticationManager" ref="authenticationManager"/>
    <property name="authenticationFailureUrl" value="/ACEGIlogin.jsp?login_error=1"/>
    <property name="defaultTargetUrl" value="/secure/index.jsp"/>
    <property name="filterProcessesUrl" value="/j_ACEGI_security_check"/>
    <property name="rememberMeServices" ref="rememberMeServices"/>
</bean>
```

接下来，我们看到的配置是对 URL 资源请求实现的安全需求配置，如代码清单 8-2 所示。这个 URL 安全配置，是通过 FilterSecurityInterceptor 拦截器的配置来实现的。在这个拦截器的配置中，需要设置 authenticationManager 属性来设置验证器，同时需要设置 accessDecisionManager 属性来设置授权器，并在 objectDefinitionSource 属性中，配置对各个 URL 请求的安全需求和用户接入权限。例如，对于不同的 URL 请求，对应着什么样的接入角色等，在如下所示的配置实例中，可以看到像/manage.manage/**这样的请求，配置成只允许具有 ROLE_SUPERVISOR 角色的用户才有获取权限等。

代码清单 8-2 资源请求的安全性配置

```
<bean id="filterInvocationInterceptor"
    class="org.ACEGIsecurity.intercept.web.FilterSecurityInterceptor">
    <property name="authenticationManager" ref="authenticationManager"/>
    <property name="accessDecisionManager">
        <bean class="org.ACEGIsecurity.vote.AffirmativeBased">
```

Download at http://www.pin5i.com/

```
            <property name="allowIfAllAbstainDecisions" value="false"/>
            <property name="decisionVoters">
                <list>
                    <bean class="org.ACEGIsecurity.vote.RoleVoter"/>
                    <bean class="org.ACEGIsecurity.vote.AuthenticatedVoter"/>
                </list>
            </property>
        </bean>
    </property>
    <property name="objectDefinitionSource">
        <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /manage.manage/**=ROLE_SUPERVISOR
            /secure/**=IS_AUTHENTICATED_REMEMBERED
            /home.home/**=IS_AUTHENTICATED_REMEMBERED
            /**=IS_AUTHENTICATED_ANONYMOUSLY
        </value>
    </property>
</bean>
```

在 HTTP 请求过滤器的配置中，是由配置好的 authenticationManager 来完成身份验证，并通过 FilterSecurityInterceptor 的 authenticationManager 属性由 IoC 容器注入进来的。下面，我们了解一下验证器的具体配置实例，如代码清单 8-3 所示。在这个 authenticationManager 的配置中，为了完成验证的需要，首先，需要为这个验证器配置一个身份数据提供器，比如在代码示例中，这个配置的身份数据提供器是通过为 providers 属性实现注入来完成的。我们可以看到，在配置中提供了一个数据库验证提供器 daoAuthenticationProvider 来为验证器提供身份数据。在对身份数据提供器的配置中，细心的读者一定会注意到，这个 daoAuthenticationProvider 是作为一个 List 属性元素而被注入的，也就是说，在 authenticationManager 的配置中，我们可以为它配置一系列的身份数据提供器，同时完成多种身份数据源的提供。在这个例子中，只是使用了一个身份数据提供器的配置，也就是我们看到的 daoAuthenticationProvider，它的功能是为验证器提供以关系数据库作为存储方式的用户验证数据。

代码清单 8-3 authenticationManager 的配置

```
<bean id="authenticationManager"
    class="org.ACEGIsecurity.providers.ProviderManager"><property name="providers">
        <list>
            <ref local="daoAuthenticationProvider"/>
            <bean class=
    "org.ACEGIsecurity.providers.anonymous.Anonymous AuthenticationProvider">
                <property name="key" value="changeThis"/>
            </bean>
            <bean class=
    "org.ACEGIsecurity.providers.rememberme.RememberMe AuthenticationProvider">
                <property name="key" value="changeThis"/>
            </bean>
        </list>
    </property>
</bean>
```

沿着验证器的配置，我们接下来看到的是 authenticationProvider 的配置，这就是已经在上面出现过的、称为验证数据提供器的模块。这个验证数据提供器起到的作用，我们在前面已经

Download at http://www.pin5i.com/

简单提到过了。通过它，可以为用户身份的验证实现提供服务器端的用户数据，它的具体配置如代码清单 8-4 所示。在代码清单中，使用的是基于数据库的验证数据提供器，也就是说，我们通过使用数据库来存储用户信息。当然，使用数据库存储用户数据只是用户数据存储的一种方式，另外比较常见的还有 LDAP 目录服务器，也是一种常用的存储用户信息的方式。甚至更简单的数据文件也都可以作为用户数据的存储方式。使用什么样的用户数据存储方式，完全取决于应用的需要。但前面介绍的 IoC 配置，为我们配置这些各种不同的验证数据提供器提供了一个统一的使用方式。

在具体的 authenticationProvider 的 Bean 配置中，还需要定义一个 userDetailService 属性，该属性是一个 DAO 对象，用它来完成从数据库中读入用户信息的工作。正是这些读取到的用户数据，为用户身份的验证实现提供了数据基础，有了这些服务器端的用户数据，把它与用户的身份数据输入进行比较，就可以完成用户的验证过程。如果我们使用其他的数据验证提供器，这个数据的验证过程也是大致相同的。只是在服务器端，应用可以使用不同的身份数据的存储方式，因而会导致不同数据读取方式的使用，从而可以从不同的数据源得到用户数据。这些都与应用的设计方案有关，比如，在应用中如果采用了 LDAP 目录服务器，用它来完成用户数据的存储和验证，就需要在实现用户身份验证的时候，在 userDetailService 的配置和实现中，提供一个 LDAP 客户端实现，通过它来完成基于 LDAP 协议的用户数据的读取，从而为用户验证的有效实现提供数据。

代码清单 8-4 authenticationProvider 的配置

```
<bean id="daoAuthenticationProvider"
    class="org.ACEGIsecurity.providers.dao.Dao AuthenticationProvider">
    <property name="userDetailsService" ref="jdbcDaoImpl"/>
    <property name="passwordEncoder">
        <bean id="md5Encoder"
    class="org.ACEGIsecurity.providers.encoding.Md5PasswordEncoder"/>
    </property>
</bean>
<!-- UserDetailsService is the most commonly frequently ACEGI Security interface
    implemented by end users -->
<bean id="jdbcDaoImpl" class="org.ACEGIsecurity.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource"><ref bean ="dataSource"/></property>
</bean>
```

既然我们使用了数据库来存储用户信息，和使用 Spring 实现数据库应用一样，在配置中，我们自然可以看到对数据库数据源的各种配置，这些配置如代码清单 8-5 所示。其中，可以看到我们熟悉的 HSQLDB 作为数据库实现，并为数据源设置了数据库驱动、用户名、密码、数据库 URL 等基本的配置信息，熟悉数据库使用的读者可以看到，这里的配置与其他在 Spring 应用中使用数据库数据源的方式是一样的。

关于 HSQLDB

HSQLDB 是一个开源的关系数据库引擎，由于是纯 Java 的实现，在 Java 应用开发中使用起来非常方便，是作为 Java 数据库应用开发、测试的非常好的选择。关于 HSQLDB 这个开源数据库产品，有兴趣的读者可以到它的网站 http://hsqldb.org/去了解它的详细情况和使用方法。

代码清单 8-5 配置数据库数据源

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManager
    DataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
```

Download at http://www.pin5i.com/

```
        <property name="url" value="jdbc:hsqldb:hsql://localhost/xdb"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>
```

以上我们看到的这些基本配置，从顶层到底层，基本上涵盖了在使用 ACEGI 的过程中，框架涉及的基本模块。了解了以上的配置，可以带给我们对 ACEGI 的一个初步印象。下面我们就去看看，这些在设置中出现的各个模块是怎样起作用的。关于这一点，我们需要深入到 ACEGI 的框架实现中去看看，它们是怎样分工合作，从而出色完成 ACEGI 所肩负的安全工作的。下面将对安全框架实现中涉及的一些关键过程（比如过滤器的实现、验证以及授权的过程等）进行分析，这也是我们接下来要分析的重点。

8.3 ACEGI 的 Web 过滤器实现

前面，我们举的是一个使用 ACEGI 框架为 Web 应用提供安全服务配置的例子。在这个例子中，安全服务以 Web 登录页面为入口，在 ACEGI 框架中支持这个 Web 登录页面功能实现的是一个 Servlet 过滤器，这个过滤器就是我们前面看到的 authenticationProcessingFilter，它对特定的 Web 请求进行拦截，并在拦截过程中完成对使用 Web 页面进行登录所需要的身份验证以及授权的处理。

一般而言，在一个 Web 应用程序中，可以注册多个过滤器，每个过滤器可以对一个或一组 Servlet 程序进行拦截。如果对这些不同的过滤器，都设置了同一个 Servlet 作为拦截的目标对象，那么 Web 容器会把这些过滤器组合起来，形成一个包含了一连串过滤器的过滤器链。这个过滤器链对目标 Servlet 的拦截顺序，与它们在 web.xml 中的配置顺序是一致的。有了这些配置，对于具体的调用过程，如果我们从 Servlet 的过滤器实现原理上来看，它是以过滤器接口的 doFilter 方法来做为拦截行为入口的。在这个 doFilter 方法中，会通过调用 FilterChain.doFilte 方法，来激活下一个过滤器的 doFilter 调用。这样，就会出现一个沿着过滤器链逐个进行拦截的过程，直到对在 Filter 链中出现的，最后一个 Filter 的 doFilter 方法的调用。而在最后一个 Filter 的 doFilter 方法中，调用 FilterChain.doFilter 方法时，将会激活目标 Servlet 的 service 方法，从而结束过滤器链对 Servlet 请求的拦截过程。

在 ACEGI 配置中，是通过 AuthenticationProcessingFilter 的过滤功能来启动 Web 页面的用户验证实现的。AuthenticationProcessingFilter 过滤器的基类是 AbstractProcessingFilter，在这个 AbstractProcessingFilter 的实现中，可以看到验证过程的实现模板，在这个实现模板中，可以看到它定义了实现验证的基本过程，如代码清单 8-6 所示。在代码清单 8-6 中，可以看到我们熟悉的 doFilter 方法，其中，在这个 doFilter 方法中实现了对 HTTP 请求的拦截。在把对 Servlet 的请求拦截下来以后，它会调用验证子类去完成验证工作，验证子类使用的是 AuthenticationProcessingFilter 过滤器。在验证工作完成以后，与 Servlet 过滤器的执行一样，框架会顺着过滤器链，继续执行下一个过滤器的拦截动作，直到最终拦截目标 Servlet 的 service 方法执行结束。最后，会根据验证结果，决定页面的跳转并启动页面的跳转动作。这个页面跳转，是在拦截器拦截结束以及 Servlet 的 service 方法调用完成以后发生的。

Download at http://www.pin5i.com/

代码清单 8-6 AbstractProcessingFilter 的 doFilter

```
    public void doFilter(ServletRequest request, ServletResponse response,
     FilterChain chain)
        throws IOException, ServletException {
     // 检验是不是符合 ServletRequest/SevletResponse 的要求。
        if (!(request instanceof HttpServletRequest)) {
            throw new ServletException("Can only process HttpPServletRequest");
        }
        if (!(response instanceof HTTPServletResponse)) {
            throw new ServletException("Can only process HttpServletResponse");
        }
        HttpServletRequest HttpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        if (requiresAuthentication(httpRequest, httpResponse)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Request is to process authentication");
            }
// 定义 ACEGI 中的 Authentication 对象，从而通过这个对象，来持有用户验证信息。
            Authentication authResult;

            try {
                onPreAuthentication(httpRequest, httpResponse);
/**
* 具体验证过程委托给子类完成，比如通过 AuthenticationProcessingFilter 来完成基于
* Web 页面的用户验证。
*/
                authResult = attemptAuthentication(httpRequest);
            } catch (AuthenticationException failed) {
                // Authentication failed.
                unsuccessfulAuthentication(httpRequest, httpResponse, failed);
                return;
            }
            // Authentication success.
            if (continueChainBeforeSuccessfulAuthentication) {
                chain.doFilter(request, response);
            }
// 验证工作完成后的后续工作，跳转到相应的页面，跳转的页面路径已经做好了配置。
            successfulAuthentication(httpRequest, httpResponse, authResult);
            return;
        }
        chain.doFilter(request, response);
    }
```

在以上的代码清单中，可以看到具体的验证工作是交由子类 AuthenticationProcessingFilter 来完成的。在 AuthenticationProcessingFilter 中，其验证过程是在它的 attemptAuthentication 方法中实现的，如代码清单 8-7 所示。在 attemptAuthentication 方法的实现中，首先，会从 HTTP 请求中取得用户名和密码，在得到 HTTP 请求中的用户验证信息以后，会生成一个 Token 对象来封装这些信息，最后把这个 Token 对象交给配置的 authenticationManager 验证器来完成具体验证工作，从而完成 Web 应用需要的用户验证。

从通过 HTTP 输入得到验证信息开始，到开始启动 authenticationManager 验证器，直到用户验证的最终完成，这时开始进入到与 Web 页面处理无关的工作，这些工作是由验证器来完成的，对于验证器实现原理的分析，是我们在下一部分的重点分析内容。

Download at http://www.pin5i.com/

代码清单 8-7　AuthenticationProcessingFilter 的 attemptAuthentication

```
public Authentication attemptAuthentication(httpServletRequest request)
    throws AuthenticationException {
//从 HTTP 请求中获取登录的用户名和密码。
    String username = obtainUsername(request);
    String password = obtainPassword(request);
    if (username == null) {
        username = "";
    }
    if (password == null) {
        password = "";
    }
// 使用得到的用户名和密码, 创建 Token 对象。
    UsernamePasswordAuthenticationToken authRequest =
new UsernamePassword AuthenticationToken (username, password);

    // Place the last username attempted into HttpSession for views.
    request.getSession().setAttribute(ACEGI_SECURITY_LAST_USERNAME_KEY, username);

    // Allow subclasses to set the "details" property.
    setDetails(request, authRequest);
// 这里启动 AuthenticationManager 完成验证。
    return this.getAuthenticationManager().authenticate(authRequest);
}
```

8.4　ACEGI 验证器的实现

8.4.1　AuthenticationManager 的 authenticate

在 ACEGI 框架中，完成验证工作的主要类是 AuthenticationManager，我们称这个模块为验证器。在这个验证器的实现中，我们可以看到完成验证的调用入口是 authenticate 方法，对于这个 authenticate 方法，我们一定不会感到陌生，因为在 AuthenticationProcessingFilter 的 attemptAuthentication 方法中我们已经看到了对它的调用。具体而言，我们可以从 AbstractAuthenticationManager 的 authenticate 方法中，看到一个实现用户验证的最基本过程，这个基本过程如代码清单 8-8 所示。

代码清单 8-8　AbstractAuthenticationManager 的 authenticate

```
public final Authentication authenticate(Authentication authRequest)
    throws AuthenticationException {
    try {
        /**
        * doAuthentication 是一个抽象方法, 由具体的 AuthenticationManager 实现, 从而
        * 完成验证工作。传入的参数是一个 Authentication 对象, 在这个对象中已经封装了从
        * HttpServletRequest 中得到的用户名和密码, 这些信息都是在页面登录时用户输入的。
        */
        Authentication authResult = doAuthentication(authRequest);
        copyDetails(authRequest, authResult);
        return authResult;
    } catch (AuthenticationException e) {
        e.setAuthentication(authRequest);
```

Download at http://www.pin5i.com/

```
            throw e;
        }
    }

    /**
     * Copies the authentication details from a source Authentication object to a
     * destination one, provided the latter does not already have one set.
     */
    private void copyDetails(Authentication source, Authentication dest) {
        if ((dest instanceof AbstractAuthenticationToken) && (dest.getDetails()
     == null)) {
            AbstractAuthenticationToken token = (AbstractAuthenticationToken) dest;
            token.setDetails(source.getDetails());
        }
    }
    protected abstract Authentication doAuthentication(Authentication authentication)
        throws AuthenticationException;
```

顺着前面的分析，我们继续往下看，在 ProviderManager 中，对用户的验证工作是在 doAuthentication 方法中完成的，如代码清单 8-9 所示。在实际应用中，在进行验证的时候，考虑到可能会有多个 Provider 来提供存储在服务器端的用户数据（关于这一点，我们一定还记得，在 IoC 容器配置中，这个 Provider 被配置成 List 中的一个元素。）因而，在这里的验证实现中，验证器会逐一遍历配置好的这个 List，也就是会遍历这个 List 中的多个数据提供器，来完成用户验证，然后返回验证结果。最后，退出对数据提供器链的逐一遍历工作。

代码清单 8-9 ProviderManager 的 doAuthentication

```
    public Authentication doAuthentication(Authentication authentication)
        throws AuthenticationException {
/**
* 这里取得配置好的provider链的迭代器，在配置时可以配置多个provider，这里我们
* 配置的是DaoAuthenticationProvide，它使用数据库来保存用户的用户名和密码信息。
*/
        Iterator iter = providers.iterator();
        Class toTest = authentication.getClass();
        AuthenticationException lastException = null;
        while (iter.hasNext()) {
            AuthenticationProvider provider = (AuthenticationProvider) iter.next();
            if (provider.supports(toTest)) {
                logger.debug("Authentication attempt using " + provider.getClass().
      getName());
// 这个result对象用来包含验证得到的结果信息。
                Authentication result = null;
                try {//使用Provider来完成用户的验证。
                    result = provider.authenticate(authentication);
                    sessionController.checkAuthenticationAllowed(result);
                } catch (AuthenticationException ae) {
                    lastException = ae;
                    result = null;
                }
                if (result != null) {
                    sessionController.registerSuccessfulAuthentication(result);
                    publishEvent(new AuthenticationSuccessEvent(result));
                    return result;
                }
            }
        }
```

Download at http://www.pin5i.com/

```
        if (lastException == null) {
            lastException = new ProviderNotFoundException(messages.getMessage
    ("ProviderManager.providerNotFound",
     new Object[] {toTest.getName()}, "No AuthenticationProvider
    found for {0}"));
        }
        // Publish the event.
        String className = exceptionMappings.getProperty(lastException.getClass().
    getName());
        AbstractAuthenticationEvent event = null;
        if (className != null) {
            try {
                Class clazz = getClass().getClassLoader().loadClass(className);
                Constructor constructor = clazz.getConstructor(new Class[] {
                            Authentication.class, AuthenticationException.class
                        });
                Object obj = constructor.newInstance(new Object[] {authentication,
                            lastException});
                Assert.isInstanceOf(AbstractAuthenticationEvent.class, obj, "Must
                be an AbstractAuthenticationEvent");
                event = (AbstractAuthenticationEvent) obj;
            } catch (ClassNotFoundException ignored) {}
            catch (NoSuchMethodException ignored) {}
            catch (IllegalAccessException ignored) {}
            catch (InstantiationException ignored) {}
            catch (InvocationTargetException ignored) {}
        }
        if (event != null) {
            publishEvent(event);
        } else {
            if (logger.isDebugEnabled()) {
                logger.debug("No event was found for the exception " + lastException.
                getClass().getName());
            }
        }
        // Throw the exception.
        throw lastException;
    }
```

8.4.2 DaoAuthenticationProvider 的实现

前面，我们看到了对验证数据提供器的逐个遍历，从而得到验证结果的过程。下面，我们看看对于这个遍历过程的一个具体实现。即在一个具体的验证数据提供器中，是怎样从数据库中取出用户信息，从而完成用户验证的。为了说明这个工作过程，我们举一个大家已经很熟悉的 DaoAuthenticationProvider 作为例子。在这个 DaoAuthenticationProvider 的基类 AbstractUserDetailsAuthenticationProvider 实现中，定义了验证的处理模板，如代码清单 8-10 所示。它的基本处理过程包括：对用户信息的缓存处理、判断用户状态、从数据库中读取用户信息、构造 UserDetails 对象、启动用户输入信息和服务器用户信息的对比，从而最终完成整个验证过程，并返回验证结果等这些基本的实现步骤。

代码清单 8-10　AbstractUserDetailsAuthenticationProvider 的 authenticate

```
    public Authentication authenticate(Authentication authentication)
        throws AuthenticationException {
```

Download at http://www.pin5i.com/

```
        Assert.isInstanceOf(UsernamePasswordAuthenticationToken.class, authentication,
  messages.getMessage("AbstractUserDetailsAuthenticationProvider.onlySupports",
              "Only UsernamePasswordAuthenticationToken is supported"));
        // 这里取得用户输入的用户名。
        String username = (authentication.getPrincipal() == null) ? "NONE_PROVIDED":
      authentication.getName();
  /**
  * 如果配置了缓存，从缓存中去取以前存入的用户验证信息，也就是 UserDetail 对象，这是为用户验证
  * 提供的用户信息缓存，有了这个缓存，在再次需要验证时，就不用每次都到数据库中去取用户数据了。
  */
        boolean cacheWasUsed = true;
        UserDetails user = this.userCache.getUserFromCache(username);
  /**
  * 在缓存中没有取到用户数据，设置标志位，根据这个标志位，会把这次取到的服务器端用户信息存入缓存中去。
  */
        if (user == null) {
            cacheWasUsed = false;
            try {//这里是调用 UserDetailService 去用户数据库，取得用户信息的地方。
                user = retrieveUser(username, (UsernamePasswordAuthenticationToken)
                        authentication);
            } catch (UsernameNotFoundException notFound) {
                if (hideUserNotFoundExceptions) {
                    throw new BadCredentialsException(messages.getMessage(
  "AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad credentials"));
                } else {
                    throw notFound;
                }
            }
            Assert.notNull(user, "retrieveUser returned null - a violation of the
     interface contract");
        }
  // 得到了用户信息以后，判断当前用户的状态，比如账户是否被锁定、用户是否有效、账户是否过期等。
        if (!user.isAccountNonLocked()) {
            throw new LockedException(messages.getMessage
            ("AbstractUserDetails AuthenticationProvider.locked",
            "User account is locked"));
        }
        if (!user.isEnabled()) {
            throw new DisabledException(messages.getMessage
            ("AbstractUserDetails AuthenticationProvider.disabled",
            "User is disabled"));
        }

        if (!user.isAccountNonExpired()) {
            throw new AccountExpiredException(messages.getMessage
            ("AbstractUser DetailsAuthenticationProvider.expired",
            "User account has expired"));
        }
        /**
        * This check must come here, as we don't want to tell users
        * about account status unless they presented the correct credentials.
        */
  /**
  * 这里是验证过程，在 retrieveUser 中保存着从数据库中得到用户的信息，在 additionalAuthentication
  * Checks 方法实现中，完成用户输入信息和服务器端的用户信息的对比工作。如果验证通过，那么构造一个
  * Authentication 对象，这个对象会被以后的授权器使用，如果验证不通过，直接抛出异常，结束验证。
  */
        try {
```

Download at http://www.pin5i.com/

```
            additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken)
            authentication);
        } catch (AuthenticationException exception) {
          if(cacheWasUsed) {
               /**
               * There was a problem, so try again after checking
               * we're using latest data (ie not from the cache).
               */
               cacheWasUsed = false;
               user = retrieveUser(username, (UsernamePasswordAuthenticationToken)
                                            authentication);
               additionalAuthenticationChecks(user,
               (UsernamePasswordAuthenticationToken)authentication);
          } else {
              throw exception;
          }
        }
        if (!user.isCredentialsNonExpired()) {
            throw new CredentialsExpiredException(messages.getMessage(
                 "AbstractUserDetailsAuthenticationProvider.credentialsExpired",
                        "User credentials have expired"));
        }
// 根据前面设定的缓存标志，决定是不是要把当前的用户信息存入缓存，以供下次验证使用。
        if (!cacheWasUsed) {
            this.userCache.putUserInCache(user);
        }
        Object principalToReturn = user;
        if (forcePrincipalAsString) {
            principalToReturn = user.getUsername();
        }
// 最后返回 Authentication 对象，这个对象记录了验证结果。
        return createSuccessAuthentication(principalToReturn, authentication, user);
    }
```

8.4.3 读取数据库用户信息

在 ACEGI 实现中，我们知道，获取服务器端用户信息的工作是由具体的 Provider 来完成的。比如，在配置文件中看到的 DaoAuthenticationProvider 就是一个典型的 Provider 实现。我们接着前面的分析，看看在 DaoAuthenticationProvider 中，怎样使用 JdbcDaoImpl 这个服务对象，从数据库中读取用户信息，如代码清单 8-11 所示。这个使用 JdbcDaoImpl 实现数据加载的过程，是通过 Spring JDBC 来完成的，在代码清单 8-11 中可以看到，使用 SQL 语句查询出数据记录，然后把这些数据记录转换成 Java 数据对象这两个基本过程的实现。

代码清单 8-11 DaoAuthenticationProvider 读取用户信息

```
    protected final UserDetails retrieveUser(String username,
    UsernamePassword AuthenticationToken authentication)
       throws AuthenticationException {
       UserDetails loadedUser;
       try {
          loadedUser = this.getUserDetailsService().loadUserByUsername(username);
       } catch (DataAccessException repositoryProblem) {
          throw new AuthenticationServiceException(repositoryProblem.getMessage(),
```

Download at http://www.pin5i.com/

```
        repositoryProblem);
        }
        if (loadedUser == null) {
            throw new AuthenticationServiceException(
                "UserDetailsService returned null, which is an interface contract
                violation");
        }
        return loadedUser;
    }
```

在 JdbcDaoImpl 中，可以看到对数据库中用户信息的读取实现，如代码清单 8-12 所示。在代码实现中，有两条定义好的 SQL 语句：一条完成的功能是从 users 表中根据用户名取得用户信息，这些用户信息包括用户名、密码、用户有效标志；另一条 SQL 语句完成的功能是从 authorities 表中，根据用户名取得用户的权限信息。有了这两条 SQL 语句，JdbcDaoImpl 使用 Spring JDBC 的 SqlQuery 来完成具体的查询工作，得到 User 对象返回。在这个 User 对象中，封装了查询得到的用户名、密码、权限等基本信息。

我们看到，由于这两条 SQL 语句的设计是确定的，因而在使用 JdbcDaoImpl 的时候，对数据库的查询也是固定的，所以对用户数据库表的设计也有一些特定的要求，只有用户数据表的设计满足了这些要求，ACEGI 才能起作用。比如，这些用户信息的查询，从 SQL 语句的设计中，可以看到查询的表是 users 表，需要查询的数据域是 username、password、enabled 域等。这些表和数据域的设计，都是使用 ACEGI 以及数据库存储用户数据，完成用户验证的默认设置。也就是说，在这种应用场景下，需要匹配好 SQL 查询语句和数据库中存储用户信息的数据表设计。如果这两者不匹配，验证会无法正常运行，对于这种情况，应用开发者要么调整用户信息数据库的表结构设计，要么重新定义这里的 SQL 查询，才能使得 ACEGI 能够顺利地使用数据库来完成用户的验证工作。

代码清单 8-12 JdbcDaoImpl 的实现

```
public static final String DEF_USERS_BY_USERNAME_QUERY = "SELECT username,
 password,enabled FROM users WHERE username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY = "SELECT username,
 authority FROM authorities WHERE username = ?";
public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException, DataAccessException {
      // 使用 Spring JDBC SqlMappingQuery 来完成用户信息的查询。
    List users = usersByUsernameMapping.execute(username);
      // 根据输入的用户名，没有查询到相应的用户信息。
    if (users.size() == 0) {
        throw new UsernameNotFoundException("User not found");
    }
      // 如果查询到一个用户列表，使用列表中的第一个作为查询得到的用户。
    UserDetails user = (UserDetails) users.get(0);
      // 使用 Spring JDBC SqlMappingQuery 来完成用户权限信息的查询。
    List dbAuths = authoritiesByUsernameMapping.execute(user.getUsername());
    addCustomAuthorities(user.getUsername(), dbAuths);
    if (dbAuths.size() == 0) {
        throw new UsernameNotFoundException("User has no GrantedAuthority");
    }
    GrantedAuthority[] arrayAuths = (GrantedAuthority[]) dbAuths.toArray(new
 GrantedAuthority[dbAuths.size()]);
    String returnUsername = user.getUsername();
    if (!usernameBasedPrimaryKey) {
```

Download at http://www.pin5i.com/

```
            returnUsername = username;
        }
        // 根据查询的用户信息和权限信息，构造User对象返回。
        return new User(returnUsername, user.getPassword(), user.isEnabled(),
                true, true, true, arrayAuths);
    }
```

对于数据库查询的具体实现，是由 ACEGI 对 usersByUsernameMapping 以及 authoritiesByUsernameMapping 这些类的设计来决定的。在这些类的设计实现中，灵活地使用了 Spring JDBC 的 SqlQuery 特性。对于它们的实现原理，我们下面做一个简要分析，对于 usersByUsernameMapping 的实现，如代码清单 8-13 所示。可以看到，在 SqlQuery 实现的初始化函数中，对 SQL 语句的配置，这个 SQL 语句就是在前面看到的"SELECT username, password,enabled FROM users WHERE username = ?"，在 SQL 语句中，设置的配置参数 username 的具体值会从 HTTP 请求中取得，然后传递进来，从而完成对 users 表的用户信息查询。在 mapRow 方法中，对查询得到的数据记录进行数据转换，每一条记录会转换为一个 Java 类型的 UserDetails 数据对象，在这个数据对象中，分别对应着用户名、密码和用户是否有效的用户信息。

代码清单 8-13 UsersByUsernameMapping 的实现

```
protected class UsersByUsernameMapping extends MappingSqlQuery {
    protected UsersByUsernameMapping(DataSource ds) {
        //设置查询用户信息的SQL语句。
        super(ds, usersByUsernameQuery);
        //配置SQL语句的参数。
        declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
    //查询记录结果转换成数据对象。
    protected Object mapRow(ResultSet rs, int rownum)
        throws SQLException {
        String username = rs.getString(1);
        String password = rs.getString(2);
        boolean enabled = rs.getBoolean(3);
        UserDetails user = new User(username, password, enabled, true, true, true,
                new GrantedAuthority[] {new GrantedAuthorityImpl("HOLDER")});
        return user;
    }
}
```

在用户数据的数据库查询中，另一个使用的查询类 AuthoritiesByUsernameMapping 的实现，如代码清单 8-14 所示，它的实现与 UsersByUsernameMapping 的实现是非常相似的。对于它们的相似之处我们这里不多说了。对于它们的区别，最明显地体现在它们使用了不同的 SQL 查询语句，并根据这些不同的 SQL 查询语句的查询结果，生成对应于不同数据查询结果的 Java 数据对象。具体来说，在 UsersByUsernameMapping 中，会根据查询结果生成 UserDetails 对象，在这个对象中存储了用户的基本信息，比如用户名和密码等。而对于 AuthoritiesByUsernameMapping 的查询，会根据查询结果生成 GrantedAuthorityImpl 对象，在 GrantedAuthorityImpl 对象中存储的是用户的权限信息。

Download at http://www.pin5i.com/

代码清单 8-14　AuthoritiesByUsernameMapping 的实现

```
protected class AuthoritiesByUsernameMapping extends MappingSqlQuery {
   protected AuthoritiesByUsernameMapping(DataSource ds) {
      super(ds, authoritiesByUsernameQuery);
      declareParameter(new SqlParameter(Types.VARCHAR));
      compile();
   }
   protected Object mapRow(ResultSet rs, int rownum)
      throws SQLException {
      String roleName = rolePrefix + rs.getString(2);
      GrantedAuthorityImpl authority = new GrantedAuthorityImpl(roleName);
      return authority;
   }
}
```

8.4.4　完成用户信息的对比验证

前面我们看到从数据库中得到用户信息的基本过程，接下来验证器要做的工作是，通过对比用户的输入信息和服务器端的用户信息，来确定用户的身份和权限，完成验证工作。这个数据的对比工作是在 DaoAuthenticationProvider 中完成的，它的具体实现过程由 additional-AuthenticationChecks 方法完成，如代码清单 8-15 所示。在这个对比过程中，存储在数据库中的用户的密码，由于是经过 MD5 加密的，因此在实现密码比对之前，需要使用配置的 passwordEncoder 对象，来对用户输入的密码进行 MD5 加密，在得到加密后的输入密码以后，才能将这个加密密码与在服务器数据库中设置的密码进行比较。根据比较结果，如果两者相同，那么通过验证；否则，将会抛出异常，表明验证失败。

代码清单 8-15　DaoAuthenticationProvider 的 additionalAuthenticationChecks

```
protected void additionalAuthenticationChecks(UserDetails userDetails,
   UsernamePasswordAuthenticationToken authentication)
   throws AuthenticationException {
   Object salt = null;
   if (this.saltSource != null) {
      salt = this.saltSource.getSalt(userDetails);
   }
   if (!passwordEncoder.isPasswordValid(userDetails.getPassword(), authentication.
 getCredentials().toString(), salt)) {
      throw new BadCredentialsException(messages.getMessage(
            "AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad
 credentials"), userDetails);
   }
}
```

通过以上的描述，我们对 ACEGI 进行验证的整个过程进行了分析。在这里，回顾一下这个过程的主要实现，我们看到，这个过程从 AuthenticationProcessingFilter 拦截 HTTP 请求开始，接着，会从 HTTP 请求中得到用户输入的用户名和密码，并将这些输入的用户信息放到 Authentication 对象中。然后，将这个 Authentication 对象传递给 AuthenticatioManager 使用，让验证器完成用户验证功能的具体实现。在验证器的实现中，验证器会通过持有的 Authentication 对象，把它和在服务器端取得的用户信息进行对比，从而最终完成用户验证。在验证完成以后，ACEGI 会把通过验证的、有效的用户信息封装在一个 Authentication 对象中，

Download at http://www.pin5i.com/

供以后的授权器使用。在这个验证的实现过程中，需要根据 ACEGI 要求的，配置好各种 Provider、UserDetailService 以及密码 Encoder 对象，通过这些对象的协作和功能实现，完成服务器用户数据的获取，以及与用户输入信息的比对和最终的用户验证工作。

8.5 ACEGI 授权器的实现

前面我们对 ACEGI 拦截 Web 请求并进行用户验证的实现原理进行了分析。对于安全框架而言，用户验证完成以后，ACEGI 已经为用户授权的实现做好了准备。关于用户验证和授权之间的关系，我们举一个日常生活中的例子来说明，ACEGI 就像一位称职的，负责安全保卫工作的警卫，在它的工作中，不但要对来访人员的身份进行检查（通过口令识别身份），还可以根据识别出来的身份，赋予其不同权限的钥匙，从而可以打开不同的门禁，得到不同级别的服务。从这点上看，与在这个场景中的“警卫”人员承担的角色一样，ACEGI 在 Spring 应用系统中，起到的也是类似的保卫系统安全的作用，而验证和授权分别对应于警卫识别来访者身份和为其赋予权限的过程。

8.5.1 与 Web 环境的接口 FilterSecurityInterceptor

在本节中，我们会对 ACEGI 授权的实现原理进行分析。我们以 Web 应用环境中使用 ACEGI 来实现授权的实现作为例子，在这个例子中，首先看到的是 ACEGI 授权实现与 Web 环境的接口实现，这个接口实现是由在 IoC 容器中配置的 FilterSecurityInterceptor 拦截器来完成的。这个 FilterSecurityInterceptor 拦截器的实现，如代码清单 8-16 所示。在 FilterSecurityInterceptor 拦截器中，与前面看到的其他 Servlet 拦截器一样，也是通过 doFilter 方法来对 HTTP 请求进行拦截和处理的。对于 FilterSecurityInterceptor 的拦截，它实现的处理主要体现在：首先，需要对 HTTP 请求进行检查，判断在当前的调用场合是否有安全检查的需要。因为，拦截器有可能已经对 HTTP 请求完成过安全检查了，如果是这样，这里就不需要再重复这个过程了；其次，拦截器会根据判断结果，调用基类 AbstractSecurityInterceptor 的 beforeInvocation 方法去完成具体的检查工作。

代码清单 8-16 FilterSecurityInterceptor 的实现

```
    // 这里是拦截器拦截 HTTP 请求的入口
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
         FilterInvocation fi = new FilterInvocation(request, response, chain);
        invoke(fi);
}
// 这是具体的拦截调用实现的地方。
    public void invoke(FilterInvocation fi) throws IOException, ServletException {
        if ((fi.getRequest() != null) && (fi.getRequest().getAttribute(FILTER_
     APPLIED) != null)
            && observeOncePerRequest) {
            // Filter already applied to this request and user wants us to observce
            // once-per-request handling, so don't re-do security checking.
            fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
```

Download at http://www.pin5i.com/

```
        } else {
            // First time this request being called, so perform security checking.
/**
 * 第一次收到相应的请求，需要做安全检测，同时把标志 FILTER_APPLIED 设置为 true，下次
 * 如果再有请求就不会做相同的安全检查了。
 */
            if (fi.getRequest() != null) {
                fi.getRequest().setAttribute(FILTER_APPLIED, Boolean.TRUE);
            }
// 这里是做安全检查的地方。
            InterceptorStatusToken token = super.beforeInvocation(fi);
// 顺着拦截器链继续处理。
            try {
                fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
            } finally {
                super.afterInvocation(token, null);
            }
        }
    }
```

我们来看看在 FilterSecurityInterceptor 的基类 AbstractSecurityInterceptor 中是如何实现对 HTTP 请求进行安全检查和资源使用授权的。这个实现过程在 beforeInvocation 中完成，如代码清单 8-17 所示。在 beforeInvocation 的实现中，首先，需要读取 IoC 容器中 Bean 的配置，在这些属性配置中配置了对 HTTP 请求资源的安全需求，比如，对于哪个角色的用户可以接入哪些 URL 请求资源，等等。然后会到 SecurityContextHolder 对象中取得 Authentication 对象，这个 SecurityContextHolder 相当于一个全局的缓存，在从 SecurityContextHolder 中取得 Authentication 对象之后，我们看到在这个 Authentication 对象中，封装了用户名、密码、角色等用户信息。而这些用户信息是在用户验证成功后生成的，在用户数据生成之后，通过 Authentication 对象来完成封装，从而设置到 SecurityContextHolder 中去。在这里，我们看到这个 Authentication 对象起到一个用户数据对象的作用。在进行具体授权之前，授权器会对一些例外的应用场景进行处理，比如，此时应用如果还没有完成用户验证，那么，在这种场景下授权器会首先通过 AuthenticationManager 验证器先进行这个验证过程。

做好以上的准备工作以后，授权器就可以开始进行授权工作了。这个授权工作是由 AccessDecisionManager 授权器来完成的，此时，完成授权工作需要的数据已经准备好了，比如 Authentication 对象、配置属性等，这些数据都已经从 SecurityContextHolder 和 IoC 容器配置中读取出来了。关于具体的授权实现，我们可以在对 AccessDecisionManager 的实现原理的分析中清楚地看到。

代码清单 8-17 AbstractSecurityInterceptor 的 beforeInvocation

```
    protected InterceptorStatusToken beforeInvocation(Object object) {
        Assert.notNull(object, "Object was null");
        if (!getSecureObjectClass().isAssignableFrom(object.getClass())) {
            throw new IllegalArgumentException("Security invocation attempted for object"
                + object.getClass().getName()
                + " but AbstractSecurityInterceptor only configured to support secure
                              objects of type: "
                + getSecureObjectClass());
        }
/**
 * 这里读取 FilterSecurityInterceptor 配置的 ObjectDefinitionSource 属性，这些
```

Download at http://www.pin5i.com/

```
* 属性配置了资源的安全设置。
*/
        ConfigAttributeDefinition attr = this.obtainObjectDefinitionSource().
    getAttributes(object);
        if ((attr == null) && rejectPublicInvocations) {
            throw new IllegalArgumentException(
                "No public invocations are allowed via this AbstractSecurityInterceptor.
    This indicates a configuration error because the AbstractSecurityInterceptor.
    rejectPublicInvocations property is set to 'true'");
        }
        if (attr != null) {
            if (logger.isDebugEnabled()) {
                logger.debug("Secure object: " + object.toString() + "; ConfigAttributes:"
                + attr.toString());
            }
            /**
            * We check for just the property we're interested in (we do
            * not call Context.validate() like the ContextInterceptor).
            */
/**
* 这里从 SecurityContextHolder 中去取 Authentication 对象，一般在用户验证成功以后，
* 会生成 Authentication 对象存放到 SecurityContextHolder 中去。
*/
            if (SecurityContextHolder.getContext().getAuthentication() == null) {
                credentialsNotFound(messages.getMessage
                ("AbstractSecurityInterceptor.authenticat ionNotFound",
                "An Authentication object was not found in the SecurityContext"),
                                    object, attr);
            }
            /**
            * Attempt authentication if not already authenticated, or user
            * always wants reauthentication.
            */
// 如果前面没有处理验证，这里要先进行用户验证。
            Authentication authenticated;
            if (!SecurityContextHolder.getContext().getAuthentication().isAuthenticated() ||
    alwaysReauthenticate) {
                try {
//调用配置好的 AuthenticationManager 处理用户验证，如果用户验证不成功，抛出异常结束处理。
                    authenticated = this.authenticationManager.authenticate(Security
    ContextHolder.getContext().getAuthentication());
                } catch (AuthenticationException authenticationException) {
                    throw authenticationException;
                }
                /**
                * We don't authenticated.setAuthentication(true), because
                * each provider should do that.
                */
                if (logger.isDebugEnabled()) {
                    logger.debug("Successfully Authenticated: " + authenticated.toString());
                }
// 把用户验证成功后得到的 Authentication 保存到 SecurityContextHolder 中，供下次使用。
SecurityContextHolder.getContext().setAuthentication(authenticated);
            } else {
// 这里处理已经通过用户验证的请求，先从 SecurityContextHolder 中取得 Authentication 对象。
                authenticated = SecurityContextHolder.getContext().getAuthentication();

                if (logger.isDebugEnabled()) {
```

Download at http://www.pin5i.com/

```
                logger.debug("Previously Authenticated: " + authenticated.toString());
            }
        }

        // Attempt authorization.
// 这是开始处理授权过程。
        try {
//调用配置好的 AccessDecisionManager 来完成授权工作，注意这个 decide 方法，它为用户设置权限。
            this.accessDecisionManager.decide(authenticated, object, attr);
        } catch (AccessDeniedException accessDeniedException) {
            AuthorizationFailureEvent event = new AuthorizationFailureEvent
            (object, attr, authenticated, accessDeniedException);
            publishEvent(event);
            throw accessDeniedException;
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Authorization successful");
        }
        AuthorizedEvent event = new AuthorizedEvent(object, attr, authenticated);
        publishEvent(event);
        // Attempt to run as a different user.
        Authentication runAs = this.runAsManager.buildRunAs(authenticated, object, attr);
        if (runAs == null) {
            if (logger.isDebugEnabled()) {
                logger.debug("RunAsManager did not change Authentication object");
            }
            return new InterceptorStatusToken(authenticated, false, attr, object);
    // no further work post-invocation
        } else {
            if (logger.isDebugEnabled()) {
                logger.debug("Switching to RunAs Authentication: " + runAs.toString());
            }
            SecurityContextHolder.getContext().setAuthentication(runAs);
            return new InterceptorStatusToken(authenticated, true, attr, object);
    // revert to token.Authenticated post-invocation
        }
    } else {
        if (logger.isDebugEnabled()) {
            logger.debug("Public object - authentication not attempted");
        }
        publishEvent(new PublicInvocationEvent(object));
        return null; // no further work post-invocation
    }
}
```

8.5.2 授权器的实现

为用户授权是由 AccessDecisionManager 授权器来完成的，授权的过程在授权器的 decide 方法中实现，这个方法的名字非常地生动形象。因为我们知道，授权的过程基本上可以看成是一个决策的过程。在这个决策的实现中，调用了 decide 方法，这个 decide 方法是 AccessDecisionManger 定义的一个接口方法，通过这个接口方法，可以对应好几个具体的授权器实现，关于这些不同的授权器之间的关系，如图 8-6 所示。我们可以看到，这些不同的授权器实现，就像它们的名字所隐含的寓意一样，分别对应着不同的授权规则，通过这些不同的规则来完成

Download at http://www.pin5i.com/

授权工作。比如，在 AffirmativeBased 授权器中，定义的授权规则是：只要有一个配置好的投票器表示同意，授权就会生效；如果没有同意票，但是有反对票，那么授权被否决。

我们举另一个授权器的例子来进行说明，在 ConsensusBased 授权器中实现的授权机制，是一个类似于少数服从多数的决策过程。在这个规则中，票数多的就获得决定权，也就是说，如果同意的投票器数目大于否决的投票器数目，那么授权通过；否则，授权被否决。而对于 UnanimousBased 授权器，它的授权规则基本上反映了一票否决的决策过程，在这个决策中，如果遇到一个投票器反对，那么就会拒绝授权。从这些授权器的源代码实现中，可以看到这几个授权器具体授权规则的实现，以及它们清晰的授权判断逻辑。感兴趣的读者不妨打开源代码研究一下。对于那些需要对授权规则进行配置的应用，弄清楚这些规则无疑是非常重要的。因为，尽管是相同的投票，不同的决策规则将会导致不同的授权结果的产生，从而，会对应用的安全配置有着完全不同的影响。

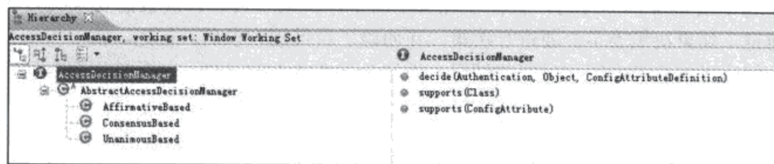


图 8-6 授权器的实现

对于授权器完成决策的规则实现，在这里，我们以 AffirmativeBased 授权器为例，看看在 AffirmativeBased 授权器中，实现的一票决定授权规则是怎样完成的，这个实现过程，如代码清单 8-18 所示。

代码清单 8-18 AffirmativeBased 的 decide

```
public void decide(Authentication authentication, Object object, ConfigAttribute
 Definition config)
    throws AccessDeniedException {
    //取得配置投票器的迭代器，可以用来遍历所有的投票器。
    Iterator iter = this.getDecisionVoters().iterator();
    int deny = 0;
    while (iter.hasNext()) {
        //取得当前投票器的投票结果。
        AccessDecisionVoter voter = (AccessDecisionVoter) iter.next();
        int result = voter.vote(authentication, object, config);
        /**
        * 对投票结果进行处理，如果是遇到 ACCESS_GRANT 的结果，授权直接通过。
        * 否则，累计 ACCESS_DENIED 的投票票数。
        */
        switch (result) {
        case AccessDecisionVoter.ACCESS_GRANTED:
            return;
        case AccessDecisionVoter.ACCESS_DENIED:
            deny++;
            break;
        default:
            break;
        }
```

Download at http://www.pin5i.com/

```
            }
    //如果有反对票，那么拒绝授权。
            if (deny > 0) {
                throw new AccessDeniedException(messages.getMessage("AbstractAccessDecision
                Manager.accessDenied","Access is denied"));
            }
    // 这里对弃权票进行处理，对全是弃权票的情况进行处理,默认是不通过，这种处理情况，
    是由allowIfAllAbstainDecisions 变量来控制的。
            // To get this far, every AccessDecisionVoter abstained
              checkAllowIfAllAbstainDecisions();
        }
```

8.5.3 投票器的实现

有了授权器，相当于建立了一个决策机制或决策委员会。具体的决策议题以及具体的决策结果，取决于每个决策个体的行为，就像在现实生活中，决策委员会的决策需要采纳委员的意见一样。具体地说，如果决策委员会要进行决策，在明确了决策规则以后，首先，需要得到各个委员的决策意见，然后再综合考虑得到最终的结果，而这个综合考虑的过程，就是应用授权器应用授权规则的过程。我们知道，在计算机的世界里不是 0 就是 1，不是赞成就是反对，没有什么含糊的地方。投票器就像是决策委员会的委员们，它与授权器的关系就像是委员和委员会的关系，根据对自己的判断投出自己神圣的一票。而授权器的授权规则，也是明确而确定的，这样，就可以得到一个确定无疑的授权结果，有了这个授权结果，应用就可以对资源进行安全方面的全面而有效的控制了。

为了理解这个具体的实现过程，我们以投票器 RoleVoter 的 vote 方法的实现原理作为例子进行简要的说明，这个 RoleVoter 的 vote 实现，如代码清单 8-19 所示。在代码清单中，可以看到，在投票前，首先设置了一个变量来记录投票结果，这个投票结果的初始值被设置为 ACCESS_ABSTAIN；然后投票器会读入对 URL 资源配置的安全需求，根据这些配置安全需求逐条进行判断。如果资源配置的授权角色与当前请求用户的角色配置相匹配，安全需求得到满足，那么将会投票表示同意；反之，投票表示反对。在这个判断完成以后，最后会返回投票结果，完成投票器的判断工作。

代码清单 8-19 RoleVoter 的 vote 实现

```
        public int vote(Authentication authentication, Object object, ConfigAttribute
        Definition config) {
            int result = ACCESS_ABSTAIN;
              //这里取得资源的安全配置。
            Iterator iter = config.getConfigAttributes();
            while (iter.hasNext()) {
                ConfigAttribute attribute = (ConfigAttribute) iter.next();
                //这个 support 判断安全配置属性是否存在，并且是否已 ROLE 为前缀进行角色配置。
                if (this.supports(attribute)) {
                    result = ACCESS_DENIED;
    // 这里对资源配置的安全授权级别进行判断，也就是匹配 ROLE 为前缀的角色配置。
    // 遍历每个配置属性，如果其中一个匹配该主体持有的 GrantedAuthority，则访问被允许。
                    // Attempt to find a matching granted authority.
                    for (int i = 0; i < authentication.getAuthorities().length; i++) {
                        if (attribute.getAttribute().equals(authentication.getAuthorities()
        [i].getAuthority())) {
```

Download at http://www.pin5i.com/

```
                    return ACCESS_GRANTED;
                }
            }
        }
    }
    return result;
}
```

结合授权器和投票器的实现，可以看到整个授权的实现过程。在 Web 应用中，使用 ACEGI 为应用提供安全性服务，会从 FilterSecurityInterceptor 拦截 HTTP 请求入手，接着，在读取在 IoC 容器中配置的对 URL 资源的安全需求以后，会把这些配置信息交由 AccessDecisionManager 授权器进行授权决策。在 ACEGI 中，框架已经为应用提供了若干不同种类的授权器来使用，这些不同的授权器，分别对应着不同的授权规则。在配置好授权器以后，为了发扬民主决策精神，ACEGI 应用还需要配置一些投票器来完成投票工作，支持授权的决策过程，而这些投票器所起的作用，就是具体对资源的安全请求进行一个判断，同意或者不同意，投票器会提交自己的判断结果；然后，授权器会根据投票器提交的投票结果，用授权规则进行汇总判断，得到最后的决策结果，从而完成最终的授权工作。最终决定该用户通过 HTTP 请求所要求的 URL 资源，根据应用的安全配置，是否有权限可以获取。

8.6 小结

本章对 Spring ACEGI 的实现进行了简要的分析。综合来说，在 ACEGI 的框架实现中，应用的安全需求管理主要是由过滤器、验证器、用户数据提供器、授权器、投票器，这几个基本模块的协作一起完成的。从架构的关系上来看，ACEGI 安全框架可以看成是一个基于 Spring 的应用。既然这样，前面我们提到的，这些实现安全需求管理的 ACEGI 的基本模块，都需要在 IoC 容器中得以正确的配置，从而才能有效地发挥其作用。因而，在本章的开头，我们就是从对这些 ACEGI 框架基本模块在 Spring 的 IoC 容器中的配置入手，开始对这些基本模块的实现原理进行分析的，我们希望通过这样一个配置场景，能够为读者提供一个对 ACEGI 的概览，以及一个更为完整而全面的理解。

对于在本章提到的那些 ACEGI 的基本模块的实现原理，我们的分析是从整个 ACEGI 应用，从接入安全请求入手到最终授权实现，这个基本线索来进行描述的。根据这个描述逻辑，首先，我们看到的是，在 Web 应用环境中，ACEGI 设计了各种 Servlet 过滤器，通过这一系列的过滤器，从而实现对请求的拦截和处理，这些过滤器包括 FilterSecurityInterceptor、AuthenticationProcessingFilter 等，这些过滤器和拦截器是在 ACEGI 中实现在 Web 环境中安全拦截功能的一些典型代表。有了这些拦截器，可以为在应用中切入 ACEGI 安全机制提供各种各样的入口。这些入口根据不同的拦截器配置，可以是在 Web 页面请求中，基本安全机制的实现，也可以是按照 RFC1945 的要求，对基本的身份验证请求的处理实现，等等。

在本章中，由于 Web 应用的普遍性，为了分析 ACEGI 的工作原理，我们以在 Web 页面请求中，应用安全机制的拦截器实现为入口，详细地说明了 ACEGI 的实现原理。在整个 ACEGI 实现中，我们可以看到，在安全配置的实现上，具体说来，除了前面我们提到的各种拦截器的设计之外，大概还可以分为，用户验证和授权这两大部分。

Download at http://www.pin5i.com/

对于用户验证模块，首先，我们也是从它的配置入手。对于用户验证，是通过配置 ACEGI 设计好的 AuthenticationManager 验证器来完成的。同时，在 AuthenticationManager 的配置基础上，需要配置各种为 ProvideManager 服务的用户数据提供器，通过这些用户数据提供器，来提供验证所需要的、存储在服务器端的用户信息，从而为用户验证的完成提供有效的用户数据支持。在验证器完成用户验证工作以后，它会把验证结果封装在 Authentication 对象中，接着把这个 Authentication 对象，通过存储在 SecurityContextHolder 中，作为应用的全局数据供授权和应用使用。

在了解用户验证实现以后，我们接着对授权模块进行了分析，在这个授权模块中，应用需要配置 AccessDecisionManager 决策器，这些决策器有许多不同的种类，通过这些决策器的选取和配置，来设置用户授权的决策规则。具体来说，像 AffirmativeBased、ConsensusBased、UnanimousBased 等，都对应着不同的授权决策规则。这些授权器的实现，分别代表着不同的决策规则，比如一票决定、一票否决、少数服从多数等。在基于这些授权器决策规则实现的基础上，ACEGI 框架还为用户配置了一系列的 Voter 投票器，这些投票器，由用户配置好以后，根据其设置的投票规则对请求进行投票，对安全需求表示同意或者反对，从而得到投票结果供授权器汇总。最后由授权器根据这些投票结果，应用授权规则得到最终的授权结果。

在本章中，我们举了一个比较常用的 RoleVoter 投票器作为例子，来分析投票器的实现原理，大致来说，在实现中，投票器会遍历每个 URL 配置的安全需求，在完成逐个的安全检查和判断之后，把是否给予授权的结果，返回给授权器使用。在手上有了投票结果以后，根据授权规则，授权器就可以根据权限，来对资源的获取实现安全控制了。从而，最终完成用户授权的实现，为 ACEGI 所承担的安全服务职责提供有力的支持。

Download at http://www.pin5i.com/

第三部分

Spring 应用篇

Download at http://www.pin5i.com/

从核心到外围，再到 Spring 的生态系统和典型应用，这是本书的写作思路，也是笔者学习计算机系统的习惯方法。很希望能够通过这种方式，让读者对 Spring 有一个系统性的了解。在前面的两个部分中，我们探讨了 Spring 框架的核心和基本驱动组件的实现原理，下面我们将对 Spring 典型应用的实现原理进行一些分析，这些分析包括像 Spring 安全框架 ACEGI 以及 Spring 源代码包中自带的 Petclinic 实例的实现原理。

严格来说，尽管这些应用实例，并不是 Spring 框架中的平台实现部分，但我们可以将它们看作是 Spring 生态系统中的重要组成部分。正是这些以 Spring 应用平台为核心的、丰富的生态系统的组成，为使用 Spring 平台开发应用提供了非常大的帮助，围绕 Spring 核心繁衍出来的开放的生态系统，也为满足应用开发的特定需求提供了舞台，从这点上看，它体现了基于开源开发而形成的软件生态系统的独特魅力，是大家的添砖加瓦，让它果实累累。在本章中，希望通过对这两个应用实现原理的分析，一方面可以让我们了解 Spring 框架的使用技巧；另一方面，对于 ACEGI（尽管严格来说，它并不属于 Spring 框架的基本内容，但在安全领域却也是独当一面并且被普遍使用的。）的简单介绍，希望能够为使用 ACEGI 的开发人员提供一个更为深入的视角。

作为应用开发人员，我们都知道，对于每一个具体的应用，如若深入到细节，每一个独立的领域应用，它的配置和使用其实都并不简单，对于这些配置和使用 Spring 的技巧，本身也是非常丰富并值得深入探讨的。为了使应用开发者对 Spring 更快地上手，在对使用 Spring 进行一些基本而完整演示的同时，又能相对独立于应用领域的内容为用户提供应用实例，是推广 Spring 产品的一个挑战。而在本书中提到的 PetClinic 实例，就是 Spring 团队应对这个挑战的一个解决方案，通过这个 PetClinic 实例，为 Spring 使用者提供了一个活生生的应用实例，深入、完整却又不失通用性。在这个例子中，我们可以看到，使用 Spring 完成数据库应用开发的完整配置和基本的实现过程。在某种程度上，这个完整实例可以看成是我们使用 Spring 时，由 Spring 团队为我们提供的 HelloWorld 应用。毫无疑问，这个应用是我们学习开发 Spring 应用的最佳参考资料之一。或者，对于应用开发而言，它还有一个重要的作用，就是还可以把它作为建立 Spring 应用开发项目的模板和构建起点，就像在建造房屋的时候，需要搭建的脚手架那样来进行使用。

好了，不多说了，让我们直奔主题吧！

Download at http://www.pin5i.com/

第 9 章

Spring petclinic 应用实例

红豆生南国，春来发几枝。
愿君多采撷，此物最相思。
——【唐】王维 《相思》

9.1 petclinic 概述

通过前面的内容，我们已经对 Spring 框架的实现原理有了清晰的了解。Spring 的源代码发布包中还提供了一个应用实例——petclinic，这个应用实例是为用户提供的一个 Spring 应用的参考设计。在 petclinic 应用实例中，展现了 Spring 构架数据库应用的能力。麻雀虽小，五脏俱全，作为一个完整的应用实例，我们可以在 petclinic 中，看到很多 Spring 特性的使用，比如，使用 IoC 容器完成 Bean 的配置，使用 Spring MVC 构建 Web 表现层，以及通过一些常用的数据库操作组件，比如 JDBC、Hibernate 和 JPA 来实现数据库数据操作的具体使用实例。在 petclinic 应用部署成功以后，通过一些简单的界面浏览，我们可以了解到，这个实例实现背后的一些用户故事。通过了解这些实现的用户故事，可以让我们先从用户的角度对 petclinic 有一个大致的了解。

首先，通过浏览器打开它的主界面，如图 9-1 所示，petclinic 应用的主界面是非常简单的，在这个主界面中，给出了与用户故事有关的几个链接，比如 find owner、Display all veterinarians 等。

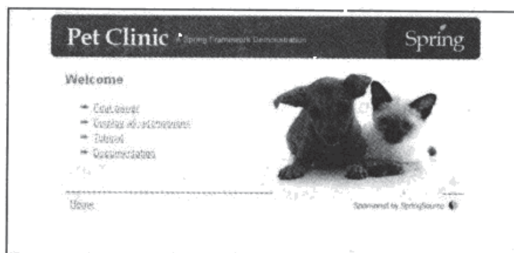


图 9-1 petclinic 的主页面

petclinic 提供了对 owner 数据进行管理的功能，可以通过点击 Find owner 链接了解该功能，如图 9-2 所示。在 owner 管理界面里，可以完成对 owner 的查询和添加操作。不难理解，这些涉及 owner 数据的相关工作，需要数据持久化操作的支持。关于 Find Owner 的实现，可以点击 Find owner

Download at http://www.pin5i.com/

链接了解，这时，在页面上，就会出现查询 owner 列表的界面，如图 9-2 和 9-3 所示。

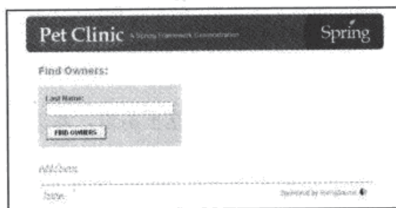


图 9-2 petclinic 的 owner 管理功能

图 9-3 查询得到 owner 列表

在得到 owner 列表以后，可以点击 owner 链接，进入到 owner 信息管理页面，如图 9-4 所示。在这个管理页面中，可以对 owner 的信息以及它拥有的 Pet 信息进行管理。

在图 9-2 所示的 owner 管理界面，点击 add owner 链接，可以增加 owner，如图 9-5 所示。

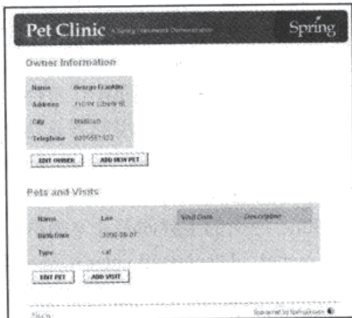


图 9-4 管理 owner 信息

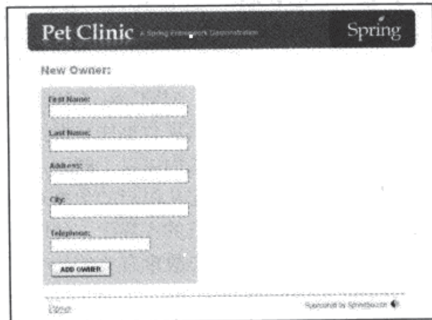


图 9-5 增加 owner

在 petclinic 的主页面中，点击 Display all veterinarians，可以得到 veterinarians 的列表显示，如图 9-6 所示。

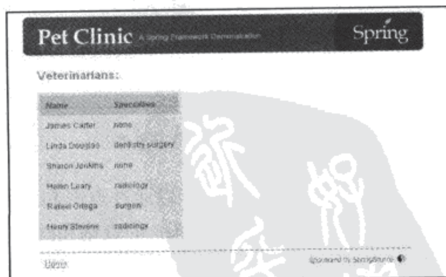


图 9-6 查询 veterinarians 列表

Download at http://www.pin5i.com/

了解了上面这些用户故事后，相信大家已经对 petclinic 实例的功能实现有一个基本的认识了。作为基于 Spring 和 Web 界面的数据库应用，在详细研究 petclinic 实现的时候，需要读者对面向对象设计、Java 语言、Servlet 的基本原理、JSP 页面设计、数据库操作以及 Web 容器的使用，有一些基本的了解。有了这些背景知识，下面让我们到 petclinic 的实现中，去看看这些基本功能是如何实现的。

9.2 部署环境及数据库

在部署 petclinic 的时候，我们使用 Tomcat 作为部署环境，这里就以 Tomcat 作为例子来进行说明。在 Tomcat 中，使用 web.xml 文件作为部署描述文件，这个文件在源代码包中的位置是 org.springframework.samples.petclinic/src/main/webapp/WEB-INF/web.xml。打开这个文件，可以看到一些 Spring 应用与 Web 环境相关的配置，如代码清单 9-1 所示。在代码清单中，可以看到对 ContextLoaderListener 的配置，了解 Spring MVC 工作原理的读者，对这个 ContextLoaderListener 一定不会觉得陌生。作为部署在 Servlet Web 服务器中的一个监听器，它的作用是在 Web 环境中载入 Spring 的 IoC 容器。在 web.xml 中，还可以看到另外一个重要设置，就是 DispatcherServlet 的配置，DispatcherServlet 是 Spring MVC 的核心类，在 Spring MVC 中，它起到一个请求分发并连接请求处理、应用数据和视图展现的作用，这是 MVC 框架实现中的一个核心功能。关于 ContextLoaderListener 和 DispatcherServlet 的实现，在第 4 章中已经详细地阐述过。至于在使用上，这两个类的使用我们都是很熟悉的，它们的使用也是很简单的，但它们背后隐藏的实现，相信读过本书第 4 章的读者，一定能够体会到这两个类中蕴含的努力，在这里我们点到为止，就不再细说了。

代码清单 9-1 Tomcat 部署描述文件 web.xml

```
<!--
    -配置 IoC 的 Web 容器载入器 ContextLoaderListener，在关于 Spring MVC 一章中有详细
    -描述，这个 ContextLoaderListener 负责在 Web 环境中建立 ioc 容器体系。对于在 web 容器
    -中建立起来的根上下文，使用默认的 Bean 配置文件是/WEB-INF/application Context.xml.
-->
<listener>
<listener-class>org.springframework.web.context.ContextLoader
    Listener</listener-class>
</listener>
<!--
    -设置 Spring MVC 的 DisptacherServlet，作为请求 MVC 框架中的分发器，负责分发
    -请求给注册的控制器 Controller 完成执行，DispatcherServlet 会建立自己的上下文，
    -这个上下文的双亲上下文，是由 ContextLoader Listener 建立起来的根上下文。
    -DispatcherServlet 的上下文使用的默认 Bean 配置文件是，{servlet-name}-servlet.xml，
    -对于 petclinic 应用来说，就是 petclinic-servlet.xml
-->
<servlet>
    <servlet-name>petclinic</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>

<!--
```

Download at http://www.pin5i.com/

```
    -把".do"类型的url请求，分发到DispatcherServlet来处理。
-->
<servlet-mapping>
    <servlet-name>petclinic</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

以上我们看到的，就是在 Web 环境中与 petclinic 应用密切相关的配置。在 petclinic 应用中，可以在/org.springframework.samples.petclinic/db 目录下，看到相关的数据库创建脚本，比如我们想知道数据库表结构是怎样建立的，可以在文件/org.springframework.samples.petclinic/db/mysql/initDB 中看到建立数据库表的 SQL 脚本，这些脚本操作如代码清单 9-2 所示。在创建脚本中，可以看到在 petclinic 应用中，建立的数据库表有 vets、specialties、vet_specialties、types、owners、pets、visits，从而为应用的基本数据提供存储的空间，从 SQL 语句中也可以清楚地看到每个表的表结构。我们以建立 vets 表的脚本为例，进行简单的说明，它定义了以下字段：id、first、name、last_name，这些字段的类型分别是 INT、VARCHAR 和 VARCHAR 型。

代码清单 9-2 建立 petclinic 数据库表

```
USE petclinic;
CREATE TABLE vets (
  id INT(4) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(30),
  last_name VARCHAR(30),
  INDEX(last_name)
) engine=InnoDB;
```

9.3 petclinic 的 Bean 配置

建立了 Web 容器和数据库的基本配置以后，接下来，我们开始讨论 petclinic 应用的 Spring Bean 配置。可以看到，Bean 配置在 petclinic-servlet.xml 文件中完成，如代码清单 9-3 所示。在代码清单 9-3 中，包括了对 Spring MVC 控制器、视图、异常页面以及国际化资源文件的设置。

代码清单 9-3 petclinic 的 Bean 配置

```
<!--
    -配置Spring MVC的Controller，这些Controllor使用@Controller标识，IoC容器会自动识别
    -识别的对象包含在org.springframework.smaples.petclinic.web包中标识还包括
    -@RequestMapping，用来标识Controller对应的URL请求。
-->
<context:component-scan base-package="org.springframework.samples.petclinic.web"/>
<!--
    -这个bean标识handler的方法。
-->
<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethod
Handler Adapter">
    <property name="webBindingInitializer">
        <bean class="org.springframework.samples.petclinic.web. ClinicBinding
Initializer"/>
    </property>
</bean>
```

Download at http://www.pin5i.com/

```
<!--
    -这个 Bean 对应用发生的异常进行转换，由 Spring 应用处理，而不是把这些异常交给 Web 容器处理。
-->
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="org.springframework.web.servlet.PageNotFound">
pageNot Found</prop>
            <prop key="org.springframework.dao.DataAccessException">
dataAccess Failure</prop>
            <prop key="org.springframework.transaction.TransactionException">
dataAccessFailure</prop>
        </props>
    </property>
</bean>

<!--
    -视图的配置，把视图转交给 InternalResourceViewResolver 和 BeanNameViewResolver 来呈现。
-->
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="xml" value="#{vets.contentType}"/>
            <entry key="atom" value="#{visits.contentType}"/>
        </map>
    </property>
    <property name="order" value="0"/>
</bean>

<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"
 p:order="1"/>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
p:prefix="/WEB-INF/jsp/"
        p:suffix=".jsp" p:order="2"/>

<!--
    -国际化配置，资源定义文件的名字为 messages_xx。
-->
<bean id="messageSource" class="org.springframework.context.support.
ResourceBundleMessageSource"
        p:basename="messages"/>
```

9.4 petclinic 的 Web 页面实现

在 IoC 容器的 Bean 配置中，完成以上的基础配置以后，接下来让我们看看 petclinic 应用的实现部分。petclinic 是一个 Web 应用，在表现层上，它使用的是 JSP 技术以及 Spring MVC 框架作为整个 Web 表现层的技术实现。我们从应用的 Web UI 入手，先看看 petclinic 的 JSP 页面的实现。我们先从 JSP 页面入手，这些 JSP 设计文件，保存在/org.springframework.samples.petclinic/src/main/webapp/WEB-INF/jsp 目录下。我们以 petclinic 的主界面/vets 界面为例，进行一下说明，可以看到这个/vets 页面的请求分发的实现，是由 ClinicController 控制器来完成的，该控制器的实现，如代码清单 9-4 所示。ClinicController 对这个 URL 请求的处理是比较简单的，它会从数据库中

Download at http://www.pin5i.com/

查询得到 vets 列表，并把列表数据交给视图进行呈现。

代码清单 9-4 ClinicController 处理 URL 请求

```
// 处理 URL 请求"/",到"welcome.jsp"完成视图呈现。
@RequestMapping("/")
public String welcomeHandler() {
    return "welcome";
}

/**
 * 处理 URL 请求"/vets",通过 clinic 对象查询得到数据库中的 Vets 列表，并交给 vets.jsp 进行视图呈现。
 */
@RequestMapping("/vets")
public ModelMap vetsHandler() {
    Vets vets = new Vets();
    vets.getVetList().addAll(this.clinic.getVets());
    return new ModelMap(vets);
}
```

在对应的 JSP 中，可以看到对视图的呈现处理，比如在 welcome.jsp 文件中，看到对应于 "/"URL 请求的响应页面，作为响应页面的主页面设计，在这个主页面中，我们可以看到设计的 /vets 链接，点击这个链接，可以在页面上看到 veterinarians 列表，如代码清单 9-5 所示。

代码清单 9-5 welcome.jsp 中/vets 的链接

```
<li><a href="<spring:url value="/vets" htmlEscape="true" />">Display all veterinarians
    </a></li>
```

对应于这个 /vets 的请求，会被 ClinicController 的 vetsHandler 处理，在 vetsHandler 的处理中，会从数据库中查询得到 Vets 的列表，然后把这个列表交给 vets.jsp 去完成展示，对于 vets.jsp 的实现，如代码清单 9-6 所示。在 vets.jsp 设计的 JSP 页面中，构造了一个 HTML 的 table 元素，然后使用 thead 构造 table 的题头。接着使用 JSTL 标签 c:forEach 从 ModelAndView 对象中，得到 vets 的数据列表，然后通过对列表数据进行遍历，把 Vet 对象的相应数据在表格列表中显示出来，具体的显示数据包括了 vet 对象的 firstName 和 lastName 数据等。

代码清单 9-6 vets.jsp

```
<table>
  <thead>
    <th>Name</th>
    <th>Specialties</th>
  </thead>
  <c:forEach var="vet" items="${vets.vetList}">
    <tr>
      <td>${vet.firstName} ${vet.lastName}</td>
      <td>
    <c:forEach var="specialty" items="${vet.specialties}">
          ${specialty.name}
        </c:forEach>
        <c:if test="${vet.nrOfSpecialties == 0}">none</c:if>
      </td>
    </tr>
  </c:forEach>
</table>
```

Download at http://www.pin5i.com/

9.5 petclinic 的领域对象实现

petclinic 应用实例的业务逻辑相对简单，因为作为一个数据库应用参考实例，设计 petclinic 实例的目的是希望通过一些常用的数据库数据操作的实现，比如对数据的增、删、改操作，以及通过与 Spring Web 层集成来为使用 Spring 的应用开发提供参考。基于这个出发点，在 petclinic 的实现中，并没有设计复杂的业务逻辑。我们从 petclinic 的领域对象设计入手，到 org.springframework.samples.petclinic 的设计中，可以了解一下 petclinic 业务逻辑层的数据对象设计。这部分的设计，以前面看到的 vets 为例，如代码清单所示 9-7 所示。可以看到，这个 Vets 类很简单，它持有一个 List 对象，在 List 对象中，持有的基本元素是 Vet 对象。

代码清单 9-7 数据对象 Vets

```
public class Vets {
private List<Vet> vets;
@XmlElement
public List<Vet> getVetList() {
    if (vets == null) {
        vets = new ArrayList<Vet>();
    }
    return vets;
}
}
```

在 Vets 中包含了一个 List，这个 List 的元素是 Vet 对象，作为一个基本的数据对象，Vet 的实现如代码清单 9-8 所示。在 Vets 对象的实现中，一方面，在它的设计中继承了 Person 类的基本数据，另一方面，它还包含了一个 Specialty 的 Set。对于 Vet，Specialty 数据对象在它们的持久化设计中，都有对应的数据库表来对应。比如，在数据库中我们看到的建立好的 vets 表、vet_specialties 表等。

代码清单 9-8 数据对象 Vet

```
public class Vet extends Person {
private Set<Specialty> specialties;
protected void setSpecialtiesInternal(Set<Specialty> specialties) {
    this.specialties = specialties;
}
protected Set<Specialty> getSpecialtiesInternal() {
    if (this.specialties == null) {
        this.specialties = new HashSet<Specialty>();
    }
    return this.specialties;
}
@XmlElement
public List<Specialty> getSpecialties() {
    List<Specialty> sortedSpecs = new ArrayList<Specialty>(getSpecialties Internal());
    PropertyComparator.sort(sortedSpecs, new MutableSortDefinition("name", true,true));
    return Collections.unmodifiableList(sortedSpecs);
}
public int getNrOfSpecialties() {
    return getSpecialtiesInternal().size();
}
public void addSpecialty(Specialty specialty) {
```

Download at http://www.pin5i.com/

```
        getSpecialtiesInternal().add(specialty);
    }
    }
```

9.6 petclinic 数据库操作的实现

在建立了领域对象的基础上，petclinic 展示了如何使用 Spring 通过各种不同的方式来完成各种持久化的工作，比如 JDBC、Hibernate、JPA 等解决方案的使用。下面，我们就分别对在 petclinic 中是如何使用这些数据库持久化方案的进行一些阐述。

9.6.1 使用 JDBC 的数据库操作

我们从最基本的 JDBC 的使用，来开始我们的 petclinic 的持久化实现之旅。使用 JDBC 进行数据库操作的 Spring Bean 的配置，如代码清单 9-9 所示。在代码清单 9-9 中，可以看到对数据库的基本设置、数据库连接池以及事务处理都进行了配置。具体地说，在配置 JDBC 数据源的时候，使用了 jdbc.properties 配置文件里的配置信息，这些配置信息包括 jdbc.driverClassName 标识的数据库驱动、jdbc.URL 标识的数据库位置、jdbc.username 标识的操作数据库的用户名、jdbc.password 标识的操作数据库配置的密码等等。有了以上这些基本的配置，就为应用配置好了数据库的数据源（datasource），在数据源的配置完成之后，就可以使用数据库了，但在应用中往往需要把数据库的操作置于事务处理的环境之中。具体来说，这就需要把刚刚配置好的数据源设置到 DataSourceTransactionManager 中去，从而通过这个 DataSourceTransactionManager，启动 Spring 为事务处理准备的统一处理机制，满足应用对数据库操作事务处理的需求。关于 Spring 事务管理的具体实现，感兴趣的读者可以在第 6 章中看到详细的分析，在这里我们就不重复了。

代码清单 9-9 JDBC 数据库操作的 Bean 配置

```
<!--
-数据库 JDBC 设置在 jdbc.properties 文件中，在这个文件里可以看到对各中数据库使用的配置，
-比如数据库驱动、URL、用户名、密码，等等。
-->
<context:property-placeholder location="classpath:jdbc.properties"/>

<!--配置数据源，这里使用了 DBCP 的链接池-->
<bean id="dataSource"
    class="org.springframework.samples.petclinic.config.Dbcp DataSourceFactory"
        p:driverClassName="${jdbc.driverClassName}" p:url="${jdbc.url}"
        p:username="${jdbc.username}" p:password="${jdbc.password}" p:populate=
    "${jdbc.populate}"
        p:schemaLocation="${jdbc.schemaLocation}" p:dataLocation="${jdbc.dataLocation}"/>

<!-- 为 JDBC 数据库操作配置事务处理 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.Data SourceTransactionManager"
        p:dataSource-ref="dataSource"/>
```

对数据库的具体操作是由 SimpleJdbcClinic 来完成的。我们以对 Vets 的查询实现为例，来了解这个类的具体实现，如代码清单 9-10 所示。可以看到，它使用了 JdbcTemplate.query 来完成

Download at http://www.pin5i.com/

具体的数据查询，这种数据查询是使用 JdbcTemplate 的时比较常用的一种方式。具体地说，在使用 JdbcTemplate.query 的 query 方法的时候，需要为这个 query 配置 SQL 语句，为查询提供最基本的数据库操作配置，其他的工作就由 Spring 来替应用完成了，这样就可以完成数据的查询。可以看到，这种使用的方式是很简洁和方便的。在代码清单 9-10 中，还使用了带参数的 JdbcTemplate.query 方法，这个方法的使用，在对 specialties 进行查询的时候可以看到，它是通过为 JdbcTemplate.query 使用的 SQL 语句设置查询参数来完成的。这个查询参数是一个 ParameterizedRowMapper 对象，完成了这些设置，就可以实现对 specialties 的数据查询了。

代码清单 9-10　SimpleJdbcClinic 查询 Vets

```
@Transactional(readOnly = true)
public Collection<Vet> getVets() throws DataAccessException {
    synchronized (this.vets) {
        if (this.vets.isEmpty()) {
            refreshVetsCache();
        }
        return this.vets;
    }
}
@ManagedOperation
@Transactional(readOnly = true)
public void refreshVetsCache() throws DataAccessException {
    synchronized (this.vets) {
        this.logger.info("Refreshing vets cache");
        // Retrieve the list of all vets.
        this.vets.clear();
        this.vets.addAll(this.simpleJdbcTemplate.query(
                "SELECT id, first_name, last_name FROM vets ORDER BY last_name,
    first_name",ParameterizedBeanPropertyRowMapper.newInstance(Vet.class)));
        // Retrieve the list of all possible specialties.
        final List<Specialty> specialties = this.simpleJdbcTemplate.query(
        "SELECT id, name FROM specialties",
        ParameterizedBeanPropertyRowMapper.newInstance(Specialty.class));
        // Build each vet's list of specialties.
        for (Vet vet : this.vets) {
            final List<Integer> vetSpecialtiesIds = this.simpleJdbcTemplate.query(
                    "SELECT specialty_id FROM vet_specialties WHERE vet_id=?",
                    new ParameterizedRowMapper<Integer>() {
                        public Integer mapRow(ResultSet rs,int row) throws SQL
    Exception {
                            return Integer.valueOf(rs.getInt(1));
                        }},
                    vet.getId().intValue());
            for (int specialtyId : vetSpecialtiesIds) {
                Specialty specialty = EntityUtils.getById(specialties, Specialty.
                class, specialtyId);
                vet.addSpecialty(specialty);
            }
        }
    }
}
```

9.6.2　使用 Hibernate 的数据库操作

在 petclinic 的实现中，petclinic 应用实例还提供了使用 Hibernate 来完成数据库操作的参考，使

Download at http://www.pin5i.com/

用 Hibernate 对数据库进行操作的 Bean 的配置，如代码清单 9-11 所示。在这些 Bean 的配置中，与前面我们看到的使用 JDBC 完成数据持久化操作相同的是，Hibernate 与 JDBC 实现数据库操作，首先都需要设置数据库数据源；而不同的是，在 Hibernate 的实现中，需要在数据源设置的基础上，配置 Hibernate 的 SessionFactory。这时使用 Hibernate 的要求，在对 SessionFactory 的配置中，我们看到，在 petclinic 中，使用的是 Spring 为便利 Hibernate 的使用，而提供的 LocalSessionFactoryBean。通过对 LocalSessionFactoryBean 的配置，来简化对 SessionFactory 的配置工作。实际上，在这里的配置中的大部分，就是对 Hibernate 的 SessionFactory 的配置，而这些配置是需要根据对 Hibernate 的使用要求来完成的。在这些配置中，包含了一系列应用使用 Hibernate 的属性配置，这些属性配置，包括像对数据的映射关系 hbm 文件的设置、对 hibernate.dialect 的设置，等等。在配置好 SessionFactory 之后，为了使用 Spring 提供的统一的事务处理环境，与前面我们看到的，通过 JDBC 使用 DataSource 一样，需要完成事务管理器的配置。只不过，在这里使用的是 HibernateTransactionManager 这个事务管理器，来支持应用在对 Hibernate 的使用中，完成事务处理的实现。

代码清单 9-11 Hibernate 的 Bean 配置

```
<!-数据库配置在 jdbc.properties 中-->
<context:property-placeholder location="classpath:jdbc.properties"/>
<!--使用 Apache Commons DBCP 作为数据库连接池，配置 DataSource. -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
    method="close"
        p:driverClassName="${jdbc.driverClassName}" p:url="${jdbc.url}" p:username
    ="${jdbc.username}"
        p:password="${jdbc.password}"/>
<!-- 配置 Hibernate SessionFactory，hbm 映射表在 petclinic.hbm.xml 中 -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSession
    FactoryBean"
        p:dataSource-ref="dataSource" p:mappingResources="petclinic.hbm.xml">
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
            <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
            <prop key="hibernate.generate_statistics">${hibernate.generate_statistics}
    </prop>
        </props>
    </property>
    <property name="eventListeners">
        <map>
            <entry key="merge">
                <bean class="org.springframework.orm.hibernate3.support.
    IdTransferringMergeEventListener"/>
            </entry>
        </map>
    </property>
</bean>
<!-- 配置 HibernateTransactionManager 作为事务管理器-->
<bean id="transactionManager" class="org.springframework.orm.hibernate3.Hibernate
    TransactionManager"
        p:sessionFactory-ref="sessionFactory"/>
```

我们都知道，在 Hibernate 实现 O/R 映射的时候，需要使用 hbm 文件来得到定义好的数据映射关系。关于具体数据映射关系的实现，我们可以到 petclinic.hbm.xml 中，详细了解这些数据

Download at http://www.pin5i.com/

映射关系的设计。关于 hbm 文件的详细设计的语法，感兴趣的读者可以到 Hibernate 的使用手册或参考文件中去了解。在这里，我们以 Vet 数据对象的 O/R 映射设计为例，只是简单地看看这些数据映射是如何进行配置的，这些配置如代码清单 9-12 所示。在这个映射配置中，可以看到：首先是 id 生成策略的设计；其次是 Vet 对象的 Java 数据属性与数据库表的数据域的对应关系的设计；最后，通过一个多对多的对应关系的映射设计，为 specialtiesInternal 集合完成映射设计。通过这个集合的映射设计，使用 Hibernate 的特性可以方便地在 Vet 对象中得到的一个包含 Specialty 对象的集合。

代码清单 9-12 Vet 的 hbm 映射

```
<class name="org.springframework.samples.petclinic.Vet" table="vets">
    <id name="id" column="id">
        <generator class="identity"/>
    </id>
    <property name="firstName" column="first_name"/>
    <property name="lastName" column="last_name"/>
    <set name="specialtiesInternal" table="vet_specialties">
        <key column="vet_id"/>
        <many-to-many column="specialty_id" class="org.springframework.samples.
    petclinic.Specialty"/>
    </set>
</class>
```

具体来说，对数据库的具体操作，是由 HibernateClinic 来完成的，关于这个 HibernateClinic 的设计，我们以对 Vets 的查询为例去了解其具体的实现，如代码清单 9-13 所示。在代码清单中，我们看到，在 HibernateClinic 的设计中，并没有使用我们熟悉的 HibernateTemplate 来完成数据操作，而是直接使用由 LocalSessionFactory 得到的 Session 来完成数据的查询，在这个数据的查询中，可以看到对 HQL 查询语句的使用。这个 HQL 查询语句"from Vet vet order by vet.lastName, vet.firstName"，表明需要从 Vet 表中，根据 Vet 的 lastName 和 firstName 来进行排序，得到所有 Vet 对象数据的集合。

代码清单 9-13 HibernateClinic 的 getVets

```
public Collection<Vet> getVets() {
    return sessionFactory.getCurrentSession().createQuery("from Vet vet order by
    vet.lastName, vet.firstName").list();
}
```

9.6.3 使用 JPA 的数据库操作

在 Petclinic 中，同样还提供了使用 JPA 的持久化设计的参考。对 JPA 数据库操作的 Bean 的配置，如代码清单 9-14 所示。在使用 JPA 中，它的使用从配置上看与使用 Hibernate 的配置非常类似。不同的是，它是通过配置 JPA EntityManagerFactory，来实现对 JPA 的使用的。在对 JPA 的使用中，Spring 通过 JpaTransactionManager 事务管理器为应用提供事务处理服务。在代码清单 9-14 所示的配置中，对 EntityManagerFactory 和 JpaTransactionManager 的具体配置，我们都可以清楚地看到。

代码清单 9-14 JPA 数据库操作的 Bean 配置

```
<!-数据库配置在 jdbc.properties 中
<context:property-placeholder location="classpath:jdbc.properties"/>
```

Download at http://www.pin5i.com/

```
<!--使用 Apache Commons DBCP 作为数据库连接池，配置 DataSource. -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
    method="close"
        p:driverClassName="${jdbc.driverClassName}" p:url="${jdbc.url}" p:username=
    "${jdbc.username}"
        p:password="${jdbc.password}"/>

<!-- 配置 JPA EntityManagerFactory -->
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainer
    EntityManagerFactoryBean"
        p:dataSource-ref="dataSource">
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter"
                p:databasePlatform="${jpa.databasePlatform}" p:showSql="${jpa.showSql}"/>
        <!--
        <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter"
                p:database="${jpa.database}" p:showSql="${jpa.showSql}"/>
        -->
        <!--
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
                p:database="${jpa.database}" p:showSql="${jpa.showSql}"/>
        -->
    </property>
</bean>

<!-- 配置 JpaTransactionManager 作为事务管理器-->
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"
            p:entityManagerFactory-ref="entityManagerFactory"/>
```

具体地说，对数据库的具体操作是由 EntityManagerClinic 来完成的，关于这个 EntityManagerClinic 的实现，与前面一样，我们同样还是以对 Vets 的查询为例，来看看这个查询的具体实现，如代码清单 9-15 所示，可以看到，在 EntityManagerClinic 中，使用 JPA 与使用 Hibernate 的查询实现一样，也是非常简单明了的。

代码清单 9-15 EntityManagerClinic 的 getVets

```
public Collection<Vet> getVets() {
    return this.em.createQuery("SELECT vet FROM Vet vet ORDER BY vet.lastName,
    vet.firstName").getResultList();
}
```

9.7 小结

在本章中，我们分析了 Spring 源代码中自带 petclinic 实例的实现，相比本书的其他章节，本章也许是技术性最为基本、最接近用户应用，内容也最为直接明了的一个部分，因为，作为 Spring 自带的应用实例，就像是 Spring 的 Web 和数据库应用的 HelloWorld 程序，它的工作是为用户提供了一个使用 Spring 的起点。

在本章中，对 petclinic 的分析，有点像在生物学实验中，我们使用的那些观察切片一样，在 petclinic 这个丰富的应用样本中，我们从上到下截取了一个剖面进行分析。在这个剖面的切片中，首先，我们看到的是在 Spring 应用中，在 Web 环境中的配置，以及在 IoC 容器中对

Download at http://www.pin5i.com/

Spring Bean 的配置，这些都是我们使用 Spring 开发 Web 应用必须掌握的基本内容。如果大家仔细阅读过本书对 IoC 容器、Web MVC 的实现进行分析的相关内容，在这里，一定会有一种尽管似曾相识的感觉，却又看山不是山，看水不是水的别样感受，这其实同样也是我自己的一个切身体验。在不了解平台实现的时候，在使用 Spring 完成应用设计的时候，一开始并没有感到太多的特别，使用就使用了，按照说明文档配置就可以了，根据一个好的应用实例配置就可以了，应用也能顺利地运转了。但是，如果我们深入思考，在我们完成这些配置的时候，在这一系列以 Spring 为中心的精妙机制背后，到底发生了什么？正是因为有太多类似于这样的思考，才促使自己有了写作本书的一点勇气。静静的聆听自己内心的声音，仔细审视自己内心的冲动，去努力探寻框架特别是开源框架背后的实现，一定会有意想不到的收获和别样的体验！

本章的内容尽管浅显直白，却希望构造一个 Spring 的应用场景，为我们再次深入了解做一个准备。因为，在前面的章节中，我们一直是在 Spring 平台中，深入地挖掘和探寻，就像在茂密的热带雨林中，不顾一切地前进和探险；而在这里，我们可以通过这个简单的原汁原味的 petclinic 实例，能够为大家了解 Spring 的实现，提供一个不同的角度，为这片雨林打开一片蔚蓝的天空，可以让我们有一个从应用往平台内部窥视的角度和契机。在这个应用的每一个点上，都有其仔细探究的广阔空间，都蕴含着丰富的宝藏。例如，看到 Web 环境的配置的时候，我们一定会想到，这些配置在 Spring 平台内部是如何实现的。比如，在对 Spring Web MVC 的实现分析中，分析过的 ContextLoader 的实现、DispatcherServlet 的实现等。

在 petclinic 应用的设计中，我们还可以看到一个层次分明的软件设计结构，在这个软件设计结构中，涉及了从应用 UI 到数据层的各个方面，具体包括了 Web UI 的 JSP 页面设计，Spring Web MVC 框架的使用，业务逻辑中领域对象的设计，直至对数据库数据的各种不同的操作方案，以及通过对 Spring 事务管理器的使用来完成的事务处理。而这些设计实例，基本上涵盖了 Spring 应用的许多基本方面和 Spring 平台特性的主体部分。当然，因为作为应用实例，在 petclinic 中我们并没有看到复杂业务逻辑的实现，没有看到使用远端调用实现分布式处理，没有使用 ACEGI 安全框架来满足资源的安全需求，等等，这是这个实例有所不足的地方。尽管，有这些不足之处，但是学习 petclinic 的实现并不会妨碍我们通过这个实例，去了解 Spring 平台的整个视野。同样地，有了这个实例，可以让应用开发人员有了一个现成的，随手可用的参考实现，同时提供了一个从应用开发人员熟悉的角度，让大家系统观察 Spring 的机会。这些，都要感谢 Spring 开发团队细致而扎实的工作，以及他们为 Spring 的广泛推广而付出的良苦用心。

与所有学习其他复杂知识的过程一样，深入学习一个优秀的软件框架或者平台的过程，绝对不可能一蹴而就的。因为，在框架的设计和实现中，不仅蕴含着对软件产品需求的深刻认识、软件开发经验的深厚积累和结晶，而且还包含不少对软件设计和架构的深刻体会，特别是像 Spring 这样一个如此基础性的应用平台，其对 Java EE 普遍需求的深刻洞察和敏锐直觉，同样是值得我们慢慢去体会的。通过前面的章节，我们已经完成了一次充满乐趣的 Spring 旅程，在这次旅程中，我们选择了一条最为基本的路径，穿越了 Spring 的茂密丛林。也许这次走马观花似的旅程，对我们来说只是一次粗浅的，只见树木不见森林的探险，因而，不免让我们心有所憾；但现在，经过艰苦跋涉，我们又来到了丛林历险的另一个起点，这个时候，已经对 Spring 丛林内部地貌有相当了解的读者，一定已经又跃跃欲试，想要再次开始自己的 Spring 丛林之旅了。对于这次即将开始的 Spring 之旅，

Download at http://www.pin5i.com/

这一次，我们的建议是，不妨直接从飞机上跳伞，借助 Eclipse IDE 环境以及 Spring 源代码的帮助，带着自己在应用 Spring 平台的时候，所遇到的各种看起来千奇百怪的具体问题，直接进入到自己感兴趣的源代码实现部分去一探究竟。因为它们其实并不神秘，而这个过程也一定充满着乐趣。另外，我们也相信，通过本书读者一定已经掌握了在 Spring 丛林中探险的基本能力，也一定找到了自己那把开启宝藏大门的钥匙。

Download at http://www.pin5i.com/