

第 1 章

Spring 简介



当提到Java开发者社区时，我们总会想到19世纪40年代后期大批的淘金者在北美河流寻找黄金的情景。作为Java开发人员，我们的河流充满了开源的项目。但是像寻宝一样，找到一个有用的开源项目是一项艰巨且耗时的工作。尽管如此，仍有越来越多的开发者转向了开源工具和代码。开放源码在使用上带来了创新，同时其限制又少，让开发人员更专注于核心应用的构建。

许多Java开源项目的通病是仅为了实现最热门的技术或模式。另外一个问题是有些项目已经失去了发展的动力，比如某些项目的0.1版本看上去很有前景，但是却始终没有推出0.2版，更不用说1.0版了。话虽如此，但还是有很多高质量、对用户友好的项目，满足了实际应用的需要。本书为大家精心挑选了这些项目的一个子集，其一个特别棒的就是Spring框架。

自Rod Johnson的*Expert One-to-One J2EE Design and Development*一书（Wrox，2002年10月出版）中展示Spring代码的雏形开始，世界上很多知名的Java开发人员都为Spring贡献了代码，最新的版本是2.5版。

在本书中，大家会看到使用各种开源技术的应用，所有技术都统一在Spring框架上。利用Spring，应用开发人员可以使用各种各样的开源工具，而不需要写繁琐的基础代码，并且能大大降低应用与特定工具的耦合性。本章将对Spring框架进行简单介绍，若读者已经对Spring相当熟悉了，可以跳过本章直接阅读第2章，第2章介绍了安装和使用Spring完成经典的“Hello, World”应用。

我们的主要目的是尽力为读者提供Spring框架的全面参考，同时我们会提供大量实际的、针对应用的建议，而决不是克隆出来一本Spring框架文档。为了做到这一点，我们建立了一个使用Spring的完整应用示例，帮助读者更好地了解Spring框架的使用方法。

1.1 Spring 是什么

首先我们需要说明的是Spring的名称。我们将在书中大量使用Spring，但其代表的意思却不尽相同。它有时候代表Spring框架，有时候又代表Spring项目。其实二者的区别是很明显的，我们这样用应该不会令你感到困惑。

Spring框架的核心基于控制反转（inversion of control, IoC）原理。符合控制反转原理的应用使用配置文件来描述组件间的依赖，然后由控制反转框架来实现配置的依赖。“反转”意味着，应用不控制其结构，而是让控制反转框架来完成。例如，类Foo的一个实例依赖于类Bar的一个实例来进行某些操作。传统的方式是Foo使用new操作符或通过某种工厂类来创建一个Bar的实例。使用控制反转机制，Bar的实例（或者其子类的实例）是通过某些外部进程在运行时动态传递给Foo实例的。这种在运行时注入依赖的行为方式，使得控制反转后来被改称为另一个更形象的名称：依赖注入（dependency

injection, DI)。通过依赖注入来管理依赖的详细内容，我们将在第3章讨论。

Spring的依赖注入实现关注于松耦合：应用的组件应该尽可能少地了解其他组件。在Java中实现松耦合的最简单方式是面向接口编程。将你的应用代码想象成一个组件系统：在Web应用中，你需要用组件来处理HTTP请求，这时你需要使用包含应用业务逻辑的组件。业务逻辑组件再使用数据访问对象（DAO）把数据保存到数据库中。重要思想是每个组件并不了解使用了何种具体实现，它只看到了接口。因为每个应用组件只了解其他组件的接口，所以我们可以替换组件实现（或组件的整组/层），而不会影响被替换组件的使用者。Spring的依赖注入的核心是使用应用配置文件提供的信息来构建组件间依赖。让Spring设置依赖的最简便方式是在组件中遵循JavaBean命名规范，但这也不是强制要求（有关JavaBean的简单介绍，参见第3章）。

使用依赖注入时，允许依赖在代码之外被配置。JavaBean提供了一个标准机制来创建能使用标准方式配置的Java资源。在第3章，你将看到Spring如何使用JavaBean规范来构建依赖注入配置模型的核心。实际上，任何Spring管理的资源都作为bean被引用。如果你不熟悉JavaBean，请查看第3章前面的快速入门。

接口与依赖注入相得益彰。我们相信，每个阅读这本书的人都不会反面对面向接口的设计与编程增加了应用程序的灵活性，也更适合于单元测试。但要把采用接口设计的应用程序的组件依赖通过编码管理起来，其复杂性非常高，并且会给开发者带来额外的编码负担。通过采用依赖注入，为基于接口的设计而编写的额外代码将大大减少，近乎为零。同样，通过采用接口，我们也可以获得依赖注入的最大好处，因为bean可以用任何接口实现来满足依赖。

依赖注入的存在，使得Spring更像一个容器而非框架，它为应用程序类的实例提供其所需的所有依赖。但是它的侵入性比在EJB容器中创建持久化实体bean要小得多。最重要的是，Spring可以自动管理应用程序组件间的依赖关系。你只需创建一个配置文件用来描述依赖，Spring就能进行管理。使用Spring进行依赖注入，你需要做的仅是让你的类遵循JavaBean命名规范（在第4章你可以看到，使用Spring的方法注入支持还可以回避这一要求）——不需要从特定的类继承，或者遵循某种私有的命名规范。使用依赖注入，我们需要改动的代码仅是让JavaBean公开更多的属性，以便于在运行时注入更多的依赖关系。

说明容器是创建所有其他软件组件依赖的运行环境。Spring是一个容器，因为它创建了作为容器子类的应用组件。

框架是你可以使用它来创建应用程序的组件集合。Spring也是一个框架，因为它提供了建立应用的公共部分的组件，如数据访问支持、MVC支持等。

我们把对依赖注入的详细讨论放到了第3章，这里先看一下使用依赖注入而非传统方法的好处：

- ❑ 减少粘合代码：依赖注入带来的最大好处之一就是，它可以奇迹般地减少为粘合应用程序的各个组件而编写的大量代码。通常，这些代码很琐碎，因为创建依赖时需要构造一个新的类的对象。然而，当你需要从JNDI库中查询依赖，或者这些依赖不能直接使用时（比如访问远程资源），这些粘合代码可能会变得相当复杂。在这些情形下，依赖注入便可以简化粘合代码，因为它提供了自动的JNDI查询，以及对远程资源的自动代理。
- ❑ 依赖外置：你可以将依赖的配置外置，这样在重新配置时可以不再重新编译代码。这给你带来两个有趣的好处。首先，正如你会在第4章看到的，Spring的依赖注入是一种理想的配置方

4 第1章 Spring 简介

式，可以让你在外部自由的配置应用程序的所有选项。其次，依赖外置使得在不同的实现间切换变得非常容易。假使你有一个DAO组件，它使用PostgreSQL数据库进行数据操作，而你想升级到Oracle。使用依赖注入，你可以简单地重新配置业务对象的依赖关系，让它使用Oracle实现而非PostgreSQL。

- 统一管理依赖：采用传统的方式管理依赖时，你在依赖类内部任何需要的地方创建依赖的实例。更糟糕的是在一些大型的应用中，你经常使用工厂或定位器来查找依赖组件。这意味着，除实际依赖以外，你的代码同样也依赖于工厂或定位器。但在大部分简单应用中，你会让依赖关系在代码中散播，当试图改变它们时问题便会出现。但使用依赖注入，所有关于依赖的信息都通过一个简单的组件（Spring 控制反转容器）进行管理，这使得管理依赖变得既简单又不容易出错。
- 提高可测试性：在类设计时使用依赖注入，你可以轻松地替换依赖。这在测试应用时特别方便。假设一个业务对象在进行某些复杂的处理，其中使用一个DAO对象来访问关系数据库。你对测试DAO本身不感兴趣，而只是简单地希望采用不同的数据集来测试业务对象。在传统的方式中，通过业务对象来获得DAO实例，你的测试会变得很痛苦，因为你无法简单地把这个DAO实现替换为一个模拟实现，来返回测试数据。相反，你需要确认测试数据库中包含正确的数据，并使用完整的DAO实现来进行测试。使用依赖注入的话，你可以为DAO对象创建一个模拟实现，然后把它传递给业务对象进行测试。这种机制可以扩展到测试应用的任何层，特别是用于测试Web组件，你可以创建 `HttpServletRequest` 和 `HttpServletResponse` 的模拟实现。
- 提倡良好的程序设计：从整体上来说，依赖注入设计实际上就是接口设计。一个典型的基于注入的应用程序，其主要组件都定义为接口，然后这些接口的具体实现由依赖注入容器创建并装配到一起。在依赖注入与基于依赖注入的容器（例如Spring）出现之前，这种设计在Java中是可行的，但是通过使用Spring，你免费获得了完整的依赖注入容器，你可以集中精力建立应用逻辑，而不必去关注支持应用逻辑的框架。

从以上列举可以看到，依赖注入提供了很多好处，但它并不是没有缺点。特别是那些不熟悉代码的人，很难判断出某个特定的依赖到底是哪个实现被注入到哪个对象中。通常，这是对依赖注入不熟悉的人的唯一问题。但当他们有些经验的时候，便会发现Spring依赖注入提供的应用集中化的视野让他们能纵览全局。在多数情况下，依赖注入带来的巨大好处让这一小小的不足显得微不足道，但在规划你的应用时，你需要注意这一细节。

1.1.1 依赖注入之外的特性

除了先进的依赖注入功能之外，Spring内核本身也是杰出的工具。Spring具有很多精良的附加特性，而这些都是基于依赖注入的基本原理优雅地设计构建的。Spring提供构建应用的所有层的工具，从用于数据访问的API（应用编程接口）到先进的MVC（model view controller，模型-视图-控制器）功能。Spring具有的这些特性的重大好处在于，虽然Spring提供了它自己的方案，你仍然可以非常容易地与其他工具整合在一起，使这些工具平等地融入Spring家族。

1.1.2 使用 Spring 进行面向方面编程

AOP是当前编程领域中的一种技术。AOP提供了实现横切逻辑的功能——这一逻辑应用于你的应

用系统的很多地方，只需要编写一次，就可以将它自动在整个系统中实施。AOP吸引了众多开发人员的目光，然而它并没有做很多的宣传，完全是通过真正的实用性，在Java开发工具箱中占有一席之地。

目前有两种主流的AOP实现。静态AOP，比如AspectJ (www.apsectj.org)，提供了编译时解决方案来构建基于AOP的逻辑，并加入到应用程序中。动态AOP，比如Spring中的AOP，允许在运行时把横切逻辑应用到任意一段代码中。在Spring 2.5中，你还可以使用加载时动态织入，即当类加载器加载类时应用横切逻辑。两种不同的AOP方法都有其适用面，实际上，Spring提供了与AspectJ整合的功能。这将在第5章和第6章进行详细的讲解。

AOP有很多应用领域。很多传统的AOP示例都采用的典型例子是某种形式的日志，但是AOP有很多更有价值的用途。实际上，在Spring框架中，AOP就用于很多领域，特别是事务管理。第5~7章详细地讲解了Spring AOP的细节，我们展示了Spring框架内部和你的应用中使用AOP的典型示例。我们还会深入了解AOP的性能问题，以及一些使用传统技术比AOP更加合适的领域。

1.1.3 数据访问

Java世界中讨论得最多的话题可能就是数据访问与持久化。很多社区网站，比如 www.theserverside.com，包含大量关于最新、最强大的数据访问工具的文章和博客文章。

Spring提供了对一系列数据访问工具的良好集成。除此之外，Spring对JDBC标准API进行了简单封装，使得在很多项目中JDBC成为可行的方案。Spring从1.1版开始，支持JDBC、Hibernate、iBATIS和JDO（Java Data Objects）。

Spring对JDBC的支持使得编写基于JDBC的应用变得更加轻松，复杂的应用也是如此。对Hibernate、iBATIS和JDO的支持让本来就很简单的API变得更加简化，这减轻了开发者的开发负担。当使用Spring API通过工具来访问数据时，我们可以充分利用Spring对事务的良好支持。第15章会对这个问题进行详细讨论。

Spring最令人欣喜的特性之一，就是易于在同一应用中混合使用不同的数据访问技术。比如，对使用Oracle数据库的程序，可以通过Hibernate实现大部分的数据访问逻辑。而在希望使用某些Oracle的专有特性的，则可以通过Spring JDBC API来实现此部分的数据访问逻辑。

1.1.4 简化与整合 Java EE

最近关于Java EE API复杂性的讨论非常多，主要集中在EJB的复杂度方面。显然，从EJB 3.0的规范来看，专家小组已经采纳这些讨论，EJB 3.0也带来了一些简化和许多新特性。相比在EJB 2.0基础上的简化，使用Spring对Java EE技术的简化支持更加方便。例如，Spring提供了一些类来构造和访问EJB资源。这些类减轻了上述两项任务的繁琐工作，同时也为EJB提供了更加面向依赖注入的API。

对于通过JNDI来访问的资源，Spring抛弃那些复杂的查找代码，在运行时直接把JNDI管理的资源作为依赖注入到其他对象中。这带来的另一好处就是，我们的应用程序与JNDI解耦了，代码在将来可以更便于重用。

到版本1.0.2为止，Spring还没有支持JMS存取。但在1.1版本发布时，CVS代码库中已经包含很多相关代码。使用这些代码，可以简化对JMS目标的所有交互工作，可以节省因为在应用程序中使用JMS而不得不编写的重复代码数量。

第11~13章将会介绍Spring如何与重要的Java EE应用组件协作，第14章将介绍应用的集成问题，

6 第1章 Spring 简介

第20章将涉及管理Java EE应用的方案。

1.1.5 基于 Spring 的任务调度

很多高级应用都需要某种任务调度能力。不管是为了发送更新消息给用户还是执行一些整理任务，能够安排在某个预定义的时间点执行某项任务对开发者是非常有价值的。

Spring提供了两种计划调度机制：一种使用Timer类，它是从Java 1.3开始引入的；另一种使用Quartz任务调度引擎。基于Timer类的调度比较基础，它只能以毫秒为单位来进行固定间隔的调度。使用Quartz框架，我们可以建立复杂的任务调度安排，使用类似于Unix Cron格式来定义何时运行某个任务。

整个第11章将讲述基于Spring的任务调度。

1.1.6 Spring 对邮件的支持

在很多应用程序中，发送Email都是典型需求，因此在Spring框架中得到了优先考虑。Spring提供了发送Email的简化API，可以很好地和Spring依赖注入功能结合。Spring支持mail API的可插入实现，并且预先附带了两种实现：一种使用JavaMail框架，另一种使用Jason Hunter编写的MailMessage类，它可以到<http://servlets.com/cos>下载，是com.oreilly.servlet包的一部分。

Spring提供了在依赖注入容器中创建消息模版的能力，可以用它作为应用发送所有邮件的基础。这样，标题和发送者地址这样的邮件参数定制就变得简单了。但是现在还不支持在代码以外定制邮件内容。在第12章中，我们将详细地讲述邮件支持，并通过使用Velocity或FreeMarker这样的模板引擎来和Spring提供一个解决方案，本方案可以让我们在Java代码以外定制邮件内容。

除了简单的邮件发送代码，我们还将展示如何使用Spring事件实现完整的异步消息基础设施。届时，将利用Spring的JMX（Java管理扩展）特性来创建高效管理邮件队列的终端。

1.1.7 动态语言

Spring动态语言可以允许我们使用Java之外的语言来实现应用组件（Spring 2.5支持BeanShell, JRuby和Groovy）。这样你就可以将应用程序的部分代码外置，因此可以让管理员和高级用户轻松地在外部修改应用代码。不使用Spring我们也可以做到这一点，但将这个支持内置到Spring中意味着，应用程序的其他部分不会识别出该组件是用另一种语言实现的，而仅将其当成一个普通的Spring Bean。

许多大型的应用程序都会处理一些复杂的业务流程。使用Java进行处理并不困难，但是在很多情况下，随着时间的推移，用户希望能够自己进行维护。这时特定领域语言的实现就有了用武之地。

在第14章中，你将会看到如何在Spring 2.5中使用动态语言，以及如何来实现简单的特定领域语言。

1.1.8 远程访问

在Java中访问或发布远程组件从未如此简单过。采用Spring，我们可以通过其对远程访问的广泛支持，快速发布或访问远程服务。

Spring支持很多远程访问机制，包括Java RMI、JAX-RPC、Caucho Hessian和Caucho Burlap。除了这些远程协议之外，Spring 1.1还有它自己的基于HTTP的协议，采用标准的Java序列化。通过Spring的

动态代理功能，你可以创建一个远程资源的代理并作为依赖注入到组件中，这样便不再需要应用程序和特定的远程技术实现耦合，也减少了需要编写的程序代码。

除了更方便地访问远程组件，Spring还提供了把Spring管理的资源公开为一个远程服务的良好支持。可以把你的服务导出为上面提到的任何一种远程访问机制，而不需要在应用中编写任何特定的实现代码。

应用程序由不同的编程语言编写，并且可运行在不同的平台，这是使用远程服务最主要的理由。在第15章中，我们将展示如何利用运行在Unix系统上的Spring服务，让Java应用程序与C#富客户端桌面应用实现远程通信。

1.1.9 事务管理

Spring提供了很好的事务管理抽象层，可以进行编程式或者声明式事务控制。利用Spring的抽象层来管理事务，我们可以很轻松地改变底层的事务协议和资源管理方式。无需更改具体代码，就可以从一个简单的、本地的、特定资源的事务管理，迁移到全局的、多资源的事务管理上。第16章将详细讲述事务管理的细节。

1.1.10 Spring MVC 框架

Spring可用于几乎所有的地方，从只作为服务的应用到桌面程序和Web应用，它提供了很丰富的类库来支持基于Web的应用。当选择如何实现Web前端时，Spring使你拥有了最大的灵活性。

对于任意复杂度的Web应用而言，都有必要使用框架来分离处理逻辑和表现。要做到这一点，可以使用Spring的Struts支持或使用Spring自己的MVC框架。要实现复杂的页面流转，可以使用Spring Web Flow。还可以使用多种不同的视图技术，从JSP (JavaServer Pages) 和Apache的Jakarta Velocity到Apache POI (生成Microsoft Excel格式的文件) 和iText (生成Adobe PDF格式的文件)。Spring MVC框架实现很完善，可以满足你大部分的需求。对于那剩下的小部分，我们可以很容易地扩展此MVC框架，加入自定义功能。

Spring MVC支持的视图技术很广泛，而且还在不断扩展。Spring的标签库很好地增强了对JSP的标准支持，此外我们还可以充分利用对Jakarta Velocity、FreeMarker、Jakarta Tiles (从Struts中分离出来的) 和XSLT完全整合的支持。并且，我们还可以找到一些基础的视图组件，来给应用方便地增加Excel和PDF输出功能。

我们在第17章讲述Spring MVC实现。

1.1.11 Spring Web Flow

Web Flow代表一种开发Web应用的新方式，尤其是对那些依赖于相当复杂的页面间流转的应用。Web Flow大大简化了此类系统的实现。因为是一个Spring项目，它与Spring MVC框架紧密集成。就像Spring的MVC一样，Web Flow可以使用任何类型的视图技术。

我们将在第18章对Web Flow进行详细的讲解。

1.1.12 AJAX 技术

AJAX不仅是一个时髦的Web 2.0词汇，它同时也是创建富Web应用的一项重要技术。简而言之，它允许Web应用与服务器交互，并减少不必要的页面重载。你可能会认为，它与Spring框架没有多大

8 第1章 Spring 简介

的联系，但是如果需要建立一个高度互动的Web 2.0应用，你可能少不了它。我们将展示如何编写必要的代码，来添加AJAX功能到我们的Spring Web应用中。由于Spring不提供任何框架级别的基础设施用来处理AJAX Web应用，我们会为你展示可以运用到Web应用的一些重要设计和性能选择。

第18章将对AJAX应用程序进行更详细的介绍。

1.1.13 国际化

国际化在任何大型的应用中都是一个重要方面。Spring可以支持多语言的应用，我们将告诉你如何使用Spring处理这些复杂的任务。多语言应用的开发有两个方面的问题：第一，编写的代码必须没有任何硬编码的文字信息；第二，设计应用的方式必须保证以后很容易翻译。

对这些问题更深入的探讨，会引入对错误的处理：在大多数情况下，错误信息都使用一种语言——开发者的语言，我们将告诉你如何去克服这种限制。

第17章将讲解所有的国际化问题。

1.1.14 简化异常处理

在某个领域，Spring真的能帮助我们减少那些重复的代码，这个领域就是异常处理。Spring的核心理论之一就是：在Java中受检查异常（checked exception）被过度使用，框架不应该强制捕获任何无法修复的异常——这是我们都一致同意的观点。

实际上，很多框架减少了对受检查异常的处理。但是，其中很多采用的仍然是受检查异常，只是人为地减少了异常类的层次的粒度。使用Spring你会发现，它采用运行异常（unchecked exception）来给开发者提供便利，异常层次粒度更小。在本书中，我们可以看到Spring异常处理机制是如何减少代码，同时提高定位、分辨以及诊断错误代码的能力。

1.2 Spring 项目

Spring项目最引人注目地方一是其社区的活跃程度非常高，二是它与很多其他项目（比如CGLIB、Apache Geronimo和AspectJ）交叉影响、共同进步。开放源代码的一个经常被提及的好处就是假若明天项目停止开发，至少我们手头还有代码。但是让我们仔细面对该问题——对Spring这种一直在维护升级的大规模的代码库，你不会愿意自己被落下。因此，了解Spring社区的活跃和无数使用Spring框架作为核心的成功应用，能让你更放心。

1.2.1 Spring 的起源

如前所述，Spring的起源可追溯到Rod Johnson的*Expert One-to-One J2EE Design and Development*一书（Wrox，2002）。在这本书中Rod展示了他的Interface 21框架，他为自己的应用编写了本框架。该框架发布到开源世界后，便形成了现在我们所熟悉的Spring框架的基础。

Spring在早期的Beta版本和备选发布版本阶段发展得很迅速，第一个正式版本1.0在2004年3月24日发布。从那以后Spring发布了许多重要版本，到本书编写时为止，处于2.5版本^①。

1.2.2 Spring .NET

^① 2008年10月发布的Spring 2.5.6版是迄今为止最稳定的版本，已经在企业级Java系统开发中广泛使用。——译者注

Spring主框架是100%基于Java的。但由于Java版本的成功，.NET领域的开发者感觉落伍了；因此Mark Pollack和Rod Johnson启动了Spring .NET项目。两个项目有完全不同的开发团队，因此.NET项目对Java版的Spring框架的影响很小。实际上，这是非常重要的消息。与Java世界中普遍的看法相反，.NET并不是完全无用的产品，事实上，已经有好几个成功的.NET应用已经发布给了我们的客户。

该项目为Java与.NET互相取长补短开辟了新的发展道路（尤其是.NET在某些类似源代码级元数据之类的领域已经处于领先地位之后），并且也有助于在各自平台上创建更好的产品。另一个影响是，它让开发者在不同的平台间进行迁移更加容易，因为我们可以在两个平台都使用Spring。在其他项目比如Hibernate和iBATIS现在也有.NET的对应产品。你可以在www.springframework.net得到关于Spring .NET的更多信息。

1.2.3 Spring IDE

除了最简单的Spring应用之外，几乎所有应用中的配置文件都相当庞大而复杂，但是你却可以方便地使用某些IDE（集成开发环境）来帮助编写代码。

Spring IDE项目是Spring主项目的另一个分支，它是Eclipse平台的一个插件。使用Spring IDE，我们可以在编辑spring配置文件时获得完善的源代码高亮效果和代码提示功能。你可以减少配置文件出错，从而加快开发效率。除了Eclipse中使用Spring IDE之外，你还可以在Java和Spring开发中使用IntelliJ IDEA。Spring完美的支持IntelliJ IDEA 7.0。

1.2.4 Spring 安全系统（原 Acegi）

Spring安全模块是直接Acegi演变而来的。Acegi是一个基于Spring构建的安全系统，它提供了基于Spring应用程序所需的功能齐全的安全服务。包括多种认证后端、单点登录以及缓存。这本书中我们不会详细介绍Acegi，你可以在<http://acegisecurity.sourceforge.net/>查看更多的细节。Acegi项目的支持也可以通过Spring论坛<http://forum.springframework.org>获得。

1.2.5 Spring 的替代方案

回顾我们曾评论过的一些开源项目，我们会发现Spring并不是唯一提供依赖注入功能或从上至下建立应用的框架。事实上，如果仔细想想，有太多这样的框架。本着开放的精神，我们简略地介绍其中的几个，但是我们相信其中没有一个能提供Spring这样丰富的解决方案。

1. PicoContainer

PicoContainer (www.picocontainer.org) 是一个特别小（100 kB）的依赖注入容器，允许我们在除了PicoContainer本身不添加任何其他依赖的情况下使用依赖注入。因为PicoContainer就是一个单纯的依赖注入容器，所以我们会发现随着应用的不断扩展，不得不引入另外一个框架（比如Spring），那么如果一开始就用Spring岂不是更好。但是，假若需要的是一个微型的依赖注入容器，PicoContainer则是一个不错的选择。Spring的依赖注入容器包和框架中的其他部分是分离的，我们也可以很容易地只使用本部分，这样又能提供日后的扩展性。

2. NanoContainer

NanoContainer (www.nanocontainer.org) 是 PicoContainer的扩展，用于管理独立的PicoContainer容器。因为Spring也同样提供了所有标准依赖注入容器具有的功能，NanoContainer与Spring比并不占

10 第1章 Spring 简介

优势。NanoContainer最吸引人的的是其对脚本语言的支持，但是目前Spring也完全支持脚本语言。

3. Keel框架

Keel框架 (www.keelframework.org) 更像是一个元框架，因为它的大部分功能来自于其他框架，并通过它集成在一起。举例来说，它的依赖注入功能来自于Apache Avalon容器，而Web功能来自于Struts或类似的框架。Keel提供了很多相同组件的实现，并把它们集成到一个统一的结构中，这让你能够在对你的应用影响最小的情况下换出实现。尽管Keel的功能很广泛，但是它却并没有像Spring那样受欢迎。尽管我们研究Keel的时间还很短，但是我们认为，这是因为Keel的难度造成的。Spring对于各个层面的开发者来说都能立即使用，相对而言Keel就要复杂得多。话虽如此，Keel的特性还是相当不错的，它毫无疑问是Spring的直接竞争对手。

4. Google Guice

这个Guice（读“juice”）框架只关注于依赖注入。因此，它不是一个直接与Spring竞争的框架。事实上，我们可以在Guice上使用Spring管理bean。除了侧重点不同外，Spring和Guice的主要区别在于应用配置的差异。Guice使用自动装配或基于注解的配置，自动装配意味着框架已经检查了组件部分，并尝试猜测它们之间的依赖关系。这个猜测是基于依赖的类型和名称。因此，即使是Guice的创建者也承认（我们也完全同意）自动装配不适合大型的企业应用。对于复杂的应用来说，Guice的创建者建议使用基于注解的配置。Guice不像Spring，它不需要任何复杂的配置文件。

遗憾的是，使用Guice，我们只能添加Guice的注解。但即使是有这个不足，Guice也同样是一个很好的框架，它最大的优点是可与Spring一起使用。

1.3 示例代码

本书将使用两种主要方法来展示代码。在说明简单具体的问题时，我们创建目的单一的小应用程序。为了示范一些复杂的特性，我们将通过多个章节来创建一个大型的应用，可以在Apress网站 (<http://www.apress.com>) 下载这些示例源代码^①。

1.4 小结

本章从较高层次上展示了Spring框架的所有主要特性，也指出了详细讨论这些特性的章节。我们同样对一些将会贯穿于整本书的概念进行了阐述。在阅读本章之后，读者应该对“使用Spring可以做什么”有一个初步了解，接下来要探讨的就是Spring如何实现上述功能了。

在第2章中，我们将讲解运行一个最基本的Spring应用需要做的工作。我们将会告诉你如何获取Spring框架，并讨论包的选择、测试套件以及文档。另外，第2章还会介绍一些基础的Spring代码示例，其中包括经典的“Hello, World”程序，它们都是基于依赖注入的。准备好了吗？我们继续前进！

^① 读者也可登录图灵公司网站 (<http://www.turingbook.com>)，免费注册后下载本书示例的源代码。——编者注

第 6 章

AOP 进阶

6

本章将详细讨论Spring中的AOP特性。尤其是，将从更加现实的角度来看待这个问题：探讨Spring中的框架服务如何让我们能透明地使用AOP，在示例程序中讲述AOP的实际用法，讨论如何使用Spring和AspectJ的集成来克服Spring AOP的限制。

首先，我们将谈到@AspectJ。Spring 2.5给出了一种编写方面的新的方式。它能自动地将标有注解的类变成Spring AOP的方面。@AspectJ支持允许我们非常简洁地定义方面。因为@AspectJ方面就是Spring的bean，因此我们能使用所有Spring的依赖注入功能。

在第5章中我们简略地提到了引入，它允许我们使用熟悉的拦截器概念，动态地为任何一个对象加入任何一个接口的实现。

介绍完引入以后，我们将看看Spring AOP的核心类——ProxyFactoryBean是如何对我们的应用程序产生影响的。同时还将解释直接调用和代理调用之间的差异。

接下来，我们将讨论AspectJ的集成。AspectJ是一个功能完整、静态编译的AOP实现。AspectJ提供的功能要远远超过Spring AOP，不过它使用起来也比较复杂。上一章中我们讲到，如果发现Spring AOP缺少需要的功能时，使用AspectJ是一个很好的解决方案（尤其是涉及各种切入点类型时）。

最后，让我们看看如何在应用程序中使用面向方面编程。我们将忽略通常那些用例（比如日志和安全等）而给出一个更贴近实际的示例。

为了运行本章中的一些示例，我们需要取得AspectJ。它可以从<http://eclipse.org/aspectj>下载。本章中的示例就是使用的AspectJ 1.5.4版。

6.1 @AspectJ 注解

@AspectJ跟AspectJ没有关系。它是Spring用来解析连接点和通知的一组Java 5注解。这也说明@AspectJ的方面对AspectJ没有任何依赖，它们使用纯粹的Spring AOP。@AspectJ支持提供一个非常简便的方式创建方面。有一些IDE还利用@AspectJ支持来简化创建方面的过程。

让我们从编写臭名昭著的日志方面入手，见代码清单6-1。我们将为TestBean类上的所有方法调用编写一个包围通知。

代码清单6-1 简单的@AspectJ方面

```
@Aspect
public class LoggingAspect {

    @Around("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
```

```
public Object log(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("Before");
    Object ret = pjp.proceed();
    System.out.println("After");
    return ret;
}

}
```

第一个注解，@AspectJ简单地告诉Spring将这个bean视为一个方面，也就是说，从中找出切入点 and 通知。接着，让AspectJ切入点表达式标上@Around注解。这个通知非常简单：它记录下方法调用的开端、调用被通知的方法，最后再记录此方法调用的结束。接下来，让我们创建一个示例程序 and 这个TestBean类（见代码清单6-2）。

代码清单6-2 基于方面的TestBean和示例程序

```
// TestBean.java
public class TestBean {

    public void work() {
        System.out.println("work");
    }

    public void stop() {
        System.out.println("stop");
    }

}

// LoggingAspectDemo.java
public class LoggingAspectDemo {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "META-INF/spring/ataspectjdemo1-context.xml"
        );
        TestBean testBean = (TestBean) ac.getBean("test");
        testBean.work();
        testBean.stop();
    }

}
```

最后，给出剩下的最后一部分：代码清单6-3中的ataspectjdemo1-context.xml文件。

代码清单6-3 ApplicationContext配置文件

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

    <bean id="test" class="com.apress.prospring2.ch06.simple.TestBean"/>

</beans>
```

184 第6章 AOP 进阶

运行这个程序，它读取测试bean并调用其work()和stop()方法，但貌似方面并没有起作用。原因是我们忘记激活@AspectJ支持，而且没有在代码清单6-3中将LoggingAspect声明为一个Spring的bean。幸运的是，这两件事做起来都很容易。实际上，要做的只是更新一下ataspectjdemo1-context.xml文件（见代码清单6-4）。

代码清单6-4 修改后的ataspectjdemo1-context.xml

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

    <bean id="test" class="com.apress.prospring2.ch06.simple.TestBean"/>
    <bean class="com.apress.prospring2.ch06.simple.LoggingAspect"/>
    <aop:aspectj-autoproxy />

</beans>
```

加粗显示的那几行代码显示了配置的关键所在。首先，我们定义了aop命名空间。接着将LoggingAspect声明为一个普通的Spring bean。最后，我们使用<aop:aspectj-autoproxy />标签。<aop:aspectj-autoproxy />标签后面的代码是用来对所有至少拥有一个通知的bean执行回调处理的。Spring将为所有被通知的bean创建一个代理。现在来运行程序，会看到我们的日志通知得到了调用，程序的输出如下：

```
Before
work
After
Before
stop
After
```

注意到这个方面就是一个普通的Spring bean，这意味着可以使用Spring来设置它的依赖关系。举例来说，让我们改良一下LoggingAspect使其包含一个为前置和后置的日志输入自定义的消息（见代码清单6-5）。

代码清单6-5 ImprovedLoggingAspect

```
@Aspect
public class ImprovedLoggingAspect {
    private String beforeMessage;
    private String afterMessage;

    @Around("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println(String.format(this.beforeMessage,
            pjp.getSignature().getName(), Arrays.toString(pjp.getArgs())));
        Object ret = pjp.proceed();
        System.out.println(String.format(this.afterMessage,
            pjp.getSignature().getName(), Arrays.toString(pjp.getArgs())));
    }
}
```



```
        return ret;
    }

    @PostConstruct
    public void initialize() {
        Assert.notNull(this.beforeMessage,
            "The [beforeMessage] property of [" + getClass().getName() +
            "] must be set.");
        Assert.notNull(this.afterMessage,
            "The [afterMessage] property of [" + getClass().getName() +
            "] must be set.");
    }

    public void setBeforeMessage(String beforeMessage) {
        this.beforeMessage = beforeMessage;
    }

    public void setAfterMessage(String afterMessage) {
        this.afterMessage = afterMessage;
    }
}
```

在这里，可以看到我们将方面当作一个普通的Spring bean对待：定义了两个字段和一个标注了@PostConstruct的方法。代码清单6-6显示了使用ImprovedLoggingAspect对ataspectjdemo1-context.xml文件所作的修改。

代码清单6-6 修改后的ApplicationContext配置文件

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

    <bean id="test" class="com.apress.prospring2.ch06.simple.TestBean"/>
    <bean class="com.apress.prospring2.ch06.simple.ImprovedLoggingAspect">
        <property name="beforeMessage" value="Before %s %s"/>
        <property name="afterMessage" value="After %s %s"/>
    </bean>
    <aop:aspectj-autoproxy />

</beans>
```

运行这个程序，我们看到新的方面生效了。它的字段在配置文件中被赋值。程序打印出如下内容：

```
Before work []
work
After work []
Before stop []
stop
After stop []
```

可以看到创建方面并不比编写标准Java代码更难。大多数IDE也会提供对@AspectJ的支持（图6-1显示了IntelliJ IDEA 7中的@AspectJ支持）。

```
@Aspect
public class ImprovedLoggingAspect {
    private String beforeMessage;
    private String afterMessage;

    @Around("execution(* com.apress.prospring2.ch06.simple.*(..))")
    public Object log(ProceedingJoinPoint pjp) throws {
        System.out.println(String.format(this.beforeMessage,
            pjp.getSignature().getName(), Arrays.toString(pjp.getArgs())));
        Object ret = pjp.proceed();
        System.out.println(String.format(this.afterMessage,
            pjp.getSignature().getName(), Arrays.toString(pjp.getArgs())));
        return ret;
    }
}
```

图6-1 IntelliJ IDEA 7中的@AspectJ支持

在6.2节中，我们将更详细地介绍Spring中的@AspectJ支持。

6.2 @AspectJ 方面详解

现在我们已经编写出第一个@AspectJ方面，需要对它的特性进行更详细的讲解。看看如何创建切入点（包括最佳和推荐实践）和编写通知。让我们从切入点开始。我们将在任何引用标注了@Pointcut注解的方法时使用@pointcut。每当提到@Pointcut注解中用到的代码时，意思就是在说@pointcut表达式。为了说明这一点，请看下面这一小段代码：

```
@Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
private void testBeanExecution() { }
```

在这个片段中，testBeanExecution方法就是一个@pointcut，execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))就是一个切入点表达式。

6.2.1 切入点

在第一个方面中，我们在包围通知中使用了一个切入点。我们将这个切入点表达式指定成一个常量。如果想用同样的方面另外创建一个通知，就必须复制这个切入点表达式常量。为避免这种复制，我们可以使用@Pointcut注解来创建一个切入点。修改一下这个日志方面来使用@pointcut（见代码清单6-7）。

代码清单6-7 使用@Pointcut注解

```
@Aspect
public class LoggingAspectPC {
    private String beforeMessage;
    private String afterMessage;

    @Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
    private void testBeanExecution() { }

    @Around("testBeanExecution()")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println(String.format(this.beforeMessage,
            pjp.getSignature().getName(), Arrays.toString(pjp.getArgs())));
        Object ret = pjp.proceed();
        System.out.println(String.format(this.afterMessage,
            pjp.getSignature().getName(), Arrays.toString(pjp.getArgs())));
    }
}
```

```
        return ret;
    }

    @PostConstruct
    public void initialize() {
        Assert.notNull(this.beforeMessage,
            "The [beforeMessage] property of [" + getClass().getName() +
            "] must be set.");
        Assert.notNull(this.afterMessage,
            "The [afterMessage] property of [" + getClass().getName() +
            "] must be set.");
    }

    public void setBeforeMessage(String beforeMessage) {
        this.beforeMessage = beforeMessage;
    }

    public void setAfterMessage(String afterMessage) {
        this.afterMessage = afterMessage;
    }
}
```

加粗的代码显示我们已经用表达式`execution(* com.apress.prospring2.ch06.simple.TestBean. *(..))`创建了`testBeanExecution @pointcut`。日志通知使用了这个`@pointcut`，但现在我们可以使用相同的`@pointcut`添加另一个通知。代码清单6-8显示了使用了相同的`testBeanExecution @pointcut`的`LoggingAspectPC`的片段。

代码清单6-8 使用相同的切入点

```
@Aspect
public class LoggingAspectPC {
    @Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
    private void testBeanExecution() { }

    @Around("testBeanExecution()")
    public Object log(ProceedingJoinPoint jp) throws Throwable {
        ...
    }

    @After("testBeanExecution()")
    public void afterCall(JoinPoint jp) {
        System.out.println("After");
    }

    ....
}
```

加粗的代码显示出我们在两个通知中使用了相同的`@pointcut`。请注意`@pointcut`是私有的，这意味着只能在这个类中使用它。让我们进一步解释一下这个示例：我们将创建一组普通的`@pointcut`，然后将它们用在我们的方面上。因为一个`@pointcut`也就是一个带有`@Pointcut`注解的方法，我们可以在代码清单6-9中创建`SystemPointcuts`类。

代码清单6-9 `SystemPointcuts`类

```
public final class SystemPointcuts {
```

```
private SystemPointcuts() {  
    }  
  
    @Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean2.*(..))")  
    public void testBeanExecution() { }  
  
    @Pointcut("within(com.apress.prospring2.ch06.simple.TestBean2)")  
    public void fromTestBeanExecution() { }  
}
```

你可以注意到我们将类设为final并且创建了一个私有的构造器：因为只想让这个类作为@pointcut方法的容器，而不希望其他人创建该类的实例。接下来，代码清单6-10显示了现在我们能以任意数量的通知来使用SystemPointcuts了。

代码清单6-10 SystemPointcuts类中切入点的用法

```
@Aspect  
public class PointcutDemoAspect {  
  
    @Around("SystemPointcuts.testBeanExecution()")  
    public Object log(ProceedingJoinPoint pjp) throws Throwable {  
        System.out.println("Before");  
        Object ret = pjp.proceed();  
        System.out.println("After");  
        return ret;  
    }  
  
    @Around("SystemPointcuts.fromTestBeanExecution()")  
    public Object inTestBean(ProceedingJoinPoint pjp) throws Throwable {  
        System.out.println("In Test Bean");  
        Object ret = pjp.proceed();  
        System.out.println("<");  
        return ret;  
    }  
}
```

加粗的代码显示了我们使用在SystemPointcuts类中声明的@pointcut方法。我们将用TestBean2和SimpleBean类来完成这个示例，如图6-2所示。

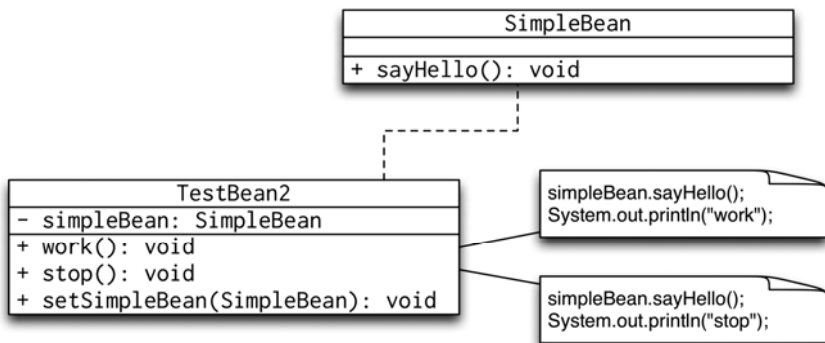


图6-2 TestBean2和SimpleBean的UML类图

我们创建了ApplicationContext配置文件来定义test和simple 这两个bean，并将simple bean注入test bean（见代码清单6-11）。

代码清单6-11 ApplicationContext配置文件

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

    <bean id="test" class="com.apress.prospring2.ch06.simple.TestBean2">
        <property name="simpleBean" ref="simple"/>
    </bean>
    <bean id="simple" class="com.apress.prospring2.ch06.simple.SimpleBean"/>
    <bean class="com.apress.prospring2.ch06.simple.PointcutDemoAspect"/>
    <aop:aspectj-autoproxy/>

</beans>
```

这个示例程序用代码清单6-11中的配置文件演示了Spring根据@pointcut方法使用上述通知的正确方法。为了演示within@pointcut，我们从TestBean2的外部来调用SimpleBean.sayHello()方法（见代码清单6-12）。

代码清单6-12 SystemPointcuts类的示例程序

```
public class PointcutDemo {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/META-INF/spring/ataspectjdemo2-context.xml"
        );
        TestBean2 testBean = (TestBean2) ac.getBean("test");
        SimpleBean simpleBean = (SimpleBean) ac.getBean("simple");
        testBean.work();
        testBean.stop();
        simpleBean.sayHello();
    }

}
```

运行示例程序，输出显示了Spring正确地创建了被通知bean。我们也使用了execution和within这两个@pointcut表达式。现在让我们通过了解可用的@AspectJ切入点表达式来完成对@pointcut表达式的讨论。

6.2.2 切入点表达式

尽管@AspectJ支持在切入点表达式中使用AspectJ语法，但Spring AOP却不能支持全系列的AspectJ切入点。表6-1总结了我们能在Spring AOP中使用的AspectJ切入点表达式。

表6-1 Spring AOP支持的@AspectJ切入点表达式

表 达 式	说 明
execution	匹配方法执行连接点。我们可以指定包、类或者方法名，以及方法的可见性、返回值和参数类型。这是应用的最为广泛的切入点表达式。例如， <code>execution(* com.apress..TestBean.*(..)</code> 表示执行 <code>com.apress</code> 包中 <code>TestBean</code> 中的所有方法，无论是何种参数或返回类型
within	匹配那些在已声明的类型中执行的连接点。例如， <code>within(com.apress..TestBean)</code> 匹配从 <code>TestBean</code> 的方法中产生的调用
this	通过用 <code>bean</code> 引用（即AOP代理）的类型跟指定的类型作比较来匹配连接点。例如， <code>this(SimpleBean)</code> 将只会匹配在 <code>SimpleBean</code> 类型的 <code>bean</code> 上的调用
target	通过用被调用的 <code>bean</code> 的类型和指定的类型作比较来匹配连接点。比如 <code>target(SimpleBean)</code> 将只会匹配在 <code>SimpleBean</code> 类型 <code>bean</code> 上方法的调用
args	通过比较方法的参数类型跟指定的参数类型来匹配连接点。例如， <code>args(String, String)</code> 将只会匹配那些有两个 <code>String</code> 类型参数的方法
@target	通过检查调用的目标对象是否具有特定注解来匹配连接点。例如， <code>@target(Magic)</code> 将会匹配带有 <code>@Magic</code> 注解的类中的方法调用
@args	跟 <code>args</code> 相似，不过 <code>@args</code> 检查的是方法参数的注解而不是它们的类型。例如， <code>@args(NotNull)</code> 会匹配所有那些包含一个被标注了 <code>@NotNull</code> 注解的参数的方法
@within	跟 <code>within</code> 相似，这个表达式匹配那些带有特定注解的类中执行的连接点。例如，表达式 <code>@within(Magic)</code> 将会匹配对带有 <code>@Magic</code> 注解的类型的 <code>bean</code> 上方法的调用
@annotation	通过检查将被调用的方法上的注解是否为指定的注解来匹配连接点。例如， <code>@annotation(Magic)</code> 将会匹配所有标有 <code>@Magic</code> 注解的方法调用
bean	通过比较 <code>bean</code> 的ID（或名称）来匹配连接点。我们也可以在 <code>bean</code> 名模式中使用通配符。例如， <code>bean("simple")</code> 将会匹配ID或名称为 <code>simple</code> 的 <code>bean</code> 中的连接点

我们可以使用`||`（或）和`&&`（与）运算符来组合多个切入点表达式，并使用`!`（非）运算符来对表达式的值取否。也可以在`@pointcut`方法或切入点表达式上使用这些运算符，或者直接编写切入点表达式（使用跟代码清单6-13中相似的代码）。

代码清单6-13 组合使用AspectJ切入点表达式

```
execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))
&& within(com.apress.prospring2..*)
```

另外，我们也可以把带有`@pointcut`注解的方法跟其他带有`@Pointcut`注解的方法或一个切入点表达式组合起来。代码清单6-14中的代码显示了3个公有的`@pointcut`方法：`same1`、`same2`和`same3`，它们的唯一区别是`@Pointcut`注解中的代码不同。

代码清单6-14 组合使用带有@Pointcut注解的方法

```
public final class SystemPointcuts {

    private SystemPointcuts() {

    }

    @Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
    private void testBeanExec() { }

    @Pointcut("within(com.apress.prospring2..*)")
```

```
private void withinProSpringPackage() { }

@Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean2.*(..) && " +
    "within(com.apress.prospring2..*)")
public void same1() { }

@Pointcut("execution(* com.apress.prospring2.ch06.simple.TestBean2.*(..) && " +
    "withinProSpringPackage())")
public void same2() { }

@Pointcut("testBeanExec() && withinProSpringPackage()")
public void same3() { }

}
```

在这里，可以看到有两个私有的@pointcuts，testBeanExec和withinProSpringPackage，并且在公共的@pointcuts same2和same3中使用私有的@pointcuts。在讨论如何在@AspectJ基础设施外使用@pointcuts之前，需要更详细地讨论一下切入点表达式。

6.2.3 探讨切入点表达式

共有十种类型的切入点表达式，每一种类型的语法都各不相同。即便是execution表达式，虽然表面看上去貌似简单明了，但是在完全利用其语法功能时也会变得非常复杂。

1. Execution表达式

Execution表达式语法在Spring中的用法跟AspectJ表达式的语法相同。代码清单6-15给出了正式的语法定义。

代码清单6-15 Execution切入点表达式的语法正式定义

```
execution(modifiers-pattern?
    ret-type-pattern
    declaring-type-pattern? name-pattern(param-pattern)
    throws-pattern?)
```

这其中的问号后缀(?)表示可选的表达式元素。换言之，我们大可不用理会它。让我们来分析我们用过的* com.apress.prospring2.ch06.simple.TestBean2.*(..)表达式：星号*表示任何返回类型（ret-type-pattern，返回类型模式），后面跟着一个全限定类名（declaring-type-pattern，声明类型模式）。我们在这个类名后又跟另一个星号*(..)，这表示一个任意名称、任意数量（包括零）和任意参数类型。因为我们没有指定修饰符模式（modifiers-pattern）或异常抛出模式（throws-pattern），Spring AOP将匹配任意修饰符和抛出任意异常的方法。

为了匹配com.apress.prospring2.ch06子包中任何类的任何方法（它可以包含任何参数、返回任何类型以及抛出任何异常），我们可能要将表达式写成* com.apress.prospring2.ch06..*.*(..)。对这个表达式进行分析，我们得知它可以匹配一个任意名字和任意参数（最后的*(..)）的方法，该方法在以com.apress.prospring2.ch06开头的包中的任意类（中间的*）中返回任意类型（开头的*），请注意包名与类名之间的..。而对那些更实际的例子，考虑一下代码清单6-16中的切入点表达式。

代码清单6-16 一个更实用的切入点表达式

```
public final class SystemPointcuts {  
  
    private SystemPointcuts() {  
  
    }  
  
    @Pointcut("execution(* com.apress.prospring2.ch06..*.*(..)) &&" +  
              "!execution(* com.apress.prospring2.ch06..*.set*(..)) &&" +  
              "!execution(* com.apress.prospring2.ch06..*.get*(..)")  
    public void serviceExecution() { }  
    ...  
}
```

你大约能猜到`serviceExecution @pointcut`用来干什么。我们可以将它用于`tx:advice`并让`com.apress.prospring2.ch06.services`包中的所有类的所有方法变得具有事务性，只要不是简单读取方法或设置方法就行。为简单起见，可以将读取方法看作是一个以`get`开头的无参方法，而设置方法就是一个任何简单的以`set`开头的方法。

提示我们更倾向于使用`@Transactional`注解来标识一个事务性方法。这甚至比使用`@pointcuts`更简单。

2. within表达式

`within`表达式的正式语法要比`execution`表达式的语法简单得多（见代码清单6-17）。

代码清单6-17 within切入点表达式的语法

```
within(declaring-type-pattern)
```

我们可以使用`..`和`*`这样普通的通配符。例如，要声明一个可以匹配在`com`包及其子包的任意类中对任意方法调用的执行过程的切入点，应该写成`within(com..*)`。

3. this表达式

`this`切入点表达式的语法跟`within`表达式的语法相似，唯一的区别是前者不能使用`..`或`*`这样的通配符。`this`切入点的语义会匹配某一对象上所有的方法执行过程，该对象的类型匹配指定的表达式，但我们怎样匹配`com`包及其子包下所有的类呢？因此，唯一允许的语法是`this(class-name)`。例如，`this(com.apress.prospring2.ch06.simple.TestBean2)`。

4. target表达式

`target`表达式的语法跟`this`表达式的完全一样。因为`target`表达式定义了一个可能匹配某一特定表达式的类的对象上的所有方法执行过程的切入点，所以它不能使用通配符。因此，唯一可用的语法就是`target(class-name)`，例如，`target(com.apress.prospring2.ch06.simple.SimpleBean)`。

5. args表达式

`args`表达式的语法是`args(type-pattern? (, type-pattern)*)`。换言之，我们可以指定零个、一个或多个`type-pattern`表达式。需要注意的是，当你在`execution`表达式中使用

argument-pattern时, 这个execution匹配会为形参的类型求值。args表达式匹配为传给方法的参数的实际类型求值。以代码清单6-18中的类举例来说明。

代码清单6-18 SimpleBean类

```
public class SimpleBean {  
  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
  
    public void x(CharSequence a, String b) {  
  
    }  
  
}
```

切入点execution(* SimpleBean.*(CharSequence, String))可以匹配方法x("A", "B")的调用, 因为这个方法名和形参类型都匹配。然而, 切入点execution(* SimpleBean.*(String, String))却不能匹配哪怕是使用两个String参数的方法调用 (一个StringBuilder和一个String也不行)。如果我们想创建一个切入点来匹配当且仅当实参为String,String时对x(CharSequence, String)方法的调用, 那么我们应该将表达式写成args(String, String)。

我们也可以在type-pattern中使用..通配符。如果要匹配某个方法的调用, 而该方法由一个Integer类型的参数开始, 以一个String类型的参数结尾, 中间包含任意类型的任意数量个参数, 那么我们就应该将表达式写成args(Integer, .., String)。

args表达式最常见的用法是在参数构建中, 我们将在6.2.6节介绍。

6. @target表达式

@target表达式是另一个需要一个完全类型名的简单表达式的例子。而且, 这个类型名应该表示为一个@interface。因此, 这种表达式的示例用法是@target(Magic)或@target(org.springframework.transaction.annotation.Transactional)。这里 Magic 和 Transactional 都是带有@Retention(RetentionPolicy.RUNTIME)注解的@interface, 而且它们的@Target注解包含ElementType.TYPE。这个切入点将匹配带有此注解的类型下所有方法的调用。如果要匹配带有特定注解的方法, 应该使用@annotation表达式。

7. @within表达式

@within表达式需要一个完全@interface类型名, 它将会匹配在具有特定注解的对象(或者方法)中对任意方法的调用。例如, 我们可以这样写, @within(StartsTransaction), 其中StartsTransaction是一个@interface。

8. @annotation表达式

跟@target表达式类似, @annotation表达式将匹配任何具有特定注解的方法的执行。@annotation(Transactional)便是一个很好的例子, 它会匹配具有Transactional注解的任意方法的执行。

9. @args表达式

@args表达式与args表达式相似, 它与后者唯一的是它比较参数的注解而不是参数的类型。我们可以

使用这个表达式来匹配所有对带有指定注解参数的方法的调用。我们也能使用跟在args表达式中相同的通配符。

10. bean表达式

这个表达式是真正Spring专有的。它将会匹配某一个bean中对所有方法的调用，而该bean的id或bean名能与特定名称相匹配。我们可以在bean名中使用星号通配符*。为了匹配simple bean中所有方法的调用，我们可以写成bean(simple)；为了匹配id或其中一个名称以Service结尾的bean中所有方法的调用，则可以写成bean(*Service)。

6.2.4 在XML中使用@Pointcuts

因为用@Pointcut注解定义的切入点是AOP专有的，我们甚至可以在XML配置中使用它们。如果你想让TestBean2中所有方法的执行都参与一个事务，我们可以在<aop:advisor . . ./>元素中使用SystemPointcuts.testBeanExecution()。代码清单6-19中的XML配置文件为使用此方法的一个示例。

代码清单6-19 在XML配置文件中使用时@Pointcuts

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

    <bean id="test" class="com.apress.prospring2.ch06.simple.TestBean2">
        <property name="simpleBean" ref="simple"/>
    </bean>
    <bean id="simple" class="com.apress.prospring2.ch06.simple.SimpleBean"/>
    <aop:config>
        <aop:advisor advice-ref="tx-advice"
                    pointcut="com.apress.prospring2.ch06.simple.➡
                        SystemPointcuts.testBeanExecution()"/>
    </aop:config>

    <bean id="transactionManager"
          class="com.apress.prospring2.ch06.simple.NoopTransactionManager"/>

    <tx:advice id="tx-advice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
</beans>
```

不用担心transactionManager bean和tx:advice。重要的代码已用粗体标识。我们使用<aop:advisor . . ./>元素定义了一个通知。这个通知引用了SystemPointcuts类中带有

@Pointcut注解的方法。通知的主体就是tx:advice。我们会在第16章探讨有关Spring中声明性事务支持的更多细节。现在，只要知道这个方法将会在一个事务中执行，如果在执行过程中没有抛出任何异常，则通知将提交此事务。如果目标方法抛出任何异常，则回滚此事务。

6.2.5 通知的类型

现在我们已经知道如何编写切入点表达式（不管我们是使用带@Pointcut注解的方法还是直接在XML中编写切入点表达式），还要看看能使用的通知的类型。之前我们已经使用过包围通知（在代码清单6-1中的@Aspect示例和代码清单6-19的XML配置文件中的tx:advice）了。我们将讨论这些基本的通知（前置通知、后置通知、抛出后通知和包围通知），然后来看看如何访问这些连接点中的参数。

1. 前置通知

让我们从前置通知开始。顾名思义，前置通知先于方法体之前执行，但是除非我们抛出一个异常，否则目标方法一定会执行。这使得前置通知适用于权限控制，我们可以检查调用者是否有权调用目标方法。如果没有，我们就抛出一个异常。看一下图6-3中的UML类图，它显示了我们将在前置通知讨论中用到的类。

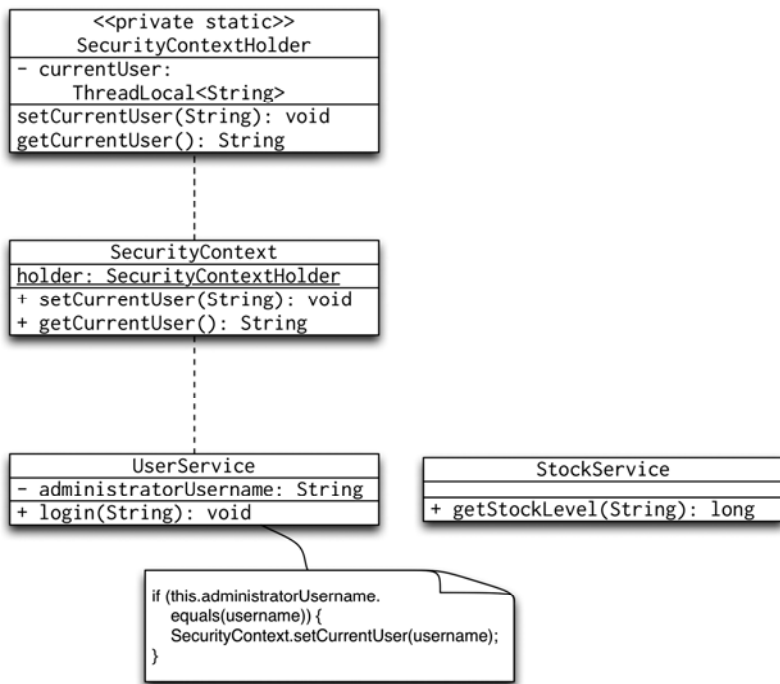


图6-3 前置通知类的UML类图

为确保StockService类的安全，现在只有当我们已经登录后才能调用它的方法。实际上，为了日后不用担心安全问题，我们要确保com.apress.prospring2.ch06.services包中的所有类的安

196 第6章 AOP 进阶

全。代码清单6-20中就是这个前置通知的简单实现。

代码清单6-20 前置通知的简单实现

```
@Aspect
public class BeforeAspect {

    @Before("execution(* com.apress.prospring2.ch06.services.*(..))")
    public void beforeLogin() throws Throwable {
        if (SecurityContext.getCurrentUser() == null)
            throw new RuntimeException("Must login to call this method.");
    }

}
```

乍看起来这肯定没问题。为了测试这个通知，我们将BeforeAspect类跟代码清单6-21中的XML配置结合起来。

代码清单6-21 前置通知的XML配置文件

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

    <bean id="userService" class="com.apress.prospring2.ch06.services.UserService"/>
    <bean id="stockService"
          class="com.apress.prospring2.ch06.services.StockService"/>
    <bean class="com.apress.prospring2.ch06.before.BeforeAspect"/>

    <aop:aspectj-autoproxy />

</beans>
```

我们将BeforeAspect作为一个Spring bean，并使用aspectj-autoproxy。这样我们就能使用代码清单6-22中的代码访问被设置了通知的服务。

代码清单6-22 前置通知示例程序

```
public class BeforeDemo {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/META-INF/spring/beforedemo1-context.xml"
        );
        StockService stockService = (StockService) ac.getBean("stockService");
        System.out.println(stockService.getStockLevel("ABC"));
    }

}
```

运行这个程序，程序失败并抛出一个消息为“必须登录以调用此方法”的RuntimeException异常。很好，程序生效了。我们可以在登录后调用这个服务方法来测试一下，于是我们增加了一个获取

UserService bean的调用并接着调用了它的login方法。如代码清单6-23中的代码片段所示。

代码清单6-23 获取UserService并登录的代码片段

```
UserService userService = (UserService) ac.getBean("userService");
userService.login("janm");
```

问题是login方法也被包含在services包的某一个类中。因此它也会接收到beforeLogin的通知。我们登录不了，是因为我们还没有登录。我们需要修改切入点表达式将login方法排除在外。可以直接写出切入点表达式，但要创建两个私有的@pointcut表达式并将它们用在前置通知的切入点表达式里。代码清单6-24显示了如何进行这个变更。

代码清单6-24 修改BeforeAspect

```
@Aspect
public class BeforeAspect {

    @Pointcut("execution(* com.apress.prospring2.ch06.services.*(..))")
    private void serviceExecution() { }

    @Pointcut(
        "execution(* com.apress.prospring2.ch06.services.UserService.login(..))")
    private void loginExecution() { }

    @Before("serviceExecution() && !loginExecution()")
    public void beforeLogin() throws Throwable {
        if (SecurityContext.getCurrentUser() == null)
            throw new RuntimeException("Must login to call this method.");
    }

}
```

现在当我们运行示例程序时，它成功了。我们被允许在还未登录时调用UserService.login方法，但我们必须登录以调用其他所有方法。

2. 后置通知

顾名思义，后置通知在目标方法正常完成后执行。所谓“正常完成”意思是该方法没有抛出任何异常。一般应用程序使用后置通知执行审计审核工作。让我们看一下代码清单6-25中后置通知的代码。

代码清单6-25 后置通知

```
@Aspect
public class AfterAspect {

    @AfterReturning("execution(* com.apress.prospring2.ch06.services.*(..))")
    public void auditCall() {
        System.out.println("After method call");
    }

}
```

这个方面为com.apress.prospring2.ch06.services包中所有类的所有方法的执行定义了一个后置通知。要实现这个示例程序，让我们从代码清单6-26的配置文件开始。

代码清单6-26 示例程序的XML配置文件

198 第6章 AOP 进阶

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

  <bean id="userService" class="com.apress.prospring2.ch06.services.UserService"/>
  <bean id="stockService"
    class="com.apress.prospring2.ch06.services.StockService"/>

  <bean class="com.apress.prospring2.ch06.afterreturning.AfterAspect"/>

  <aop:aspectj-autoproxy />

</beans>
```

在这里，我们定义了两个服务bean（userService和stockService）和AfterAspect bean。匿名的AfterAspect bean包含了我们的auditCall()后置通知。我们在示例程序中使用这个配置文件来简单地获取userService和stockService bean并使用它们的方法。示例程序的源代码如代码清单6-27所示。

代码清单6-27 后置通知的示例程序

```
public class AfterDemo {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/META-INF/spring/afterreturningdemo1-context.xml"
        );
        UserService userService = (UserService) ac.getBean("userService");
        userService.login("janm");

        StockService stockService = (StockService) ac.getBean("stockService");
        System.out.println(stockService.getStockLevel("ABC"));
    }

}
```

运行这个程序，其输出证实auditCall后置通知生效。程序打印出“After method call, After method call, 193734”。问题是这个信息不能反映任何有关调用的细节。正如我们之前写到过的，这个通知只知道有一个匹配切入点的方法正常完成了。但不知道究竟是哪一个方法，也不知道返回值是什么。我们可以使用JoinPoint参数或者@AfterReturning注解的返回字段来改进这个通知。代码清单6-28显示了第一种选择——为通知方法添加一个JoinPoint参数。

代码清单6-28 使用JoinPoint参数

```
@Aspect
public class AfterAspect {

    @AfterReturning("execution(* com.apress.prospring2.ch06.services.*(..))")
    public void auditCall(JoinPoint jp) {
        System.out.println("After method call of " + jp);
    }

}
```

我们可以使用JoinPoint参数接口来挖掘刚刚返回的那个方法的细节。也许能从JoinPoint中获取的最有趣的信息是被执行的方法以及它的参数。唯一的问题是我们不能使用JoinPoint来得到返回值。为了达到这个目的，必须停止使用JoinPoint参数并显式地为@AfterReturning注解设置returning()和argNames()属性。代码清单6-29显示了可以检查（并修改）返回值的通知。

代码清单6-29 使用returning()和argNames()属性

```
@Aspect
public class AfterAspect {

    @AfterReturning(
        pointcut = "execution(* com.apress.prospring2.ch06.services.*(..))",
        returning = "ret", argNames = "ret")
    public void auditCall(Object ret) {
        System.out.println("After method call of " + ret);
        if (ret instanceof User) {
            ((User)ret).setPassword("****");
        }
    }

}
```

注意我们定义了@AfterReturning注解的pointcut()、returning()和argNames()属性来创建更强大的后置通知。我们可以从方法调用中得到返回值。我们不能直接改变返回值，但如果返回类型允许，我们可以改变返回值的属性。这跟试图改变方法参数的值并将修改后的值传给调用代码相似。

3. 抛出后通知

接下来介绍的是抛出后通知，它在目标方法抛出一个异常时执行。利用抛出后通知，我们可以创建能访问已被抛出的异常的通知。而通知的参数中的异常类型能进一步对异常类型进行过滤。代码清单6-30显示了一个过滤所有从服务调用中抛出的IOException的方面。

代码清单6-30 IOException抛出后通知

```
@Aspect
public class AfterThrowingAspect {

    @AfterThrowing(
        pointcut = "execution(* com.apress.prospring2.ch06.services.*(..))",
        throwing = "ex", argNames = "ex")
    public void logException(IOException ex) {
        System.out.println("After method call of " + ex);
    }

}
```

200 第6章 AOP 进阶

加粗的代码显示了该通知最重要的部分。`@AfterThrowing`通知的`throwing()`属性指定了将会接收异常的参数，`argNames()`根据参数被声明的顺序指定了它们的名称。现在，运行代码清单6-31中的示例程序，它会失败，因为如果`sku`参数为`null`，`StockService.getStockLevel(String)`方法会抛出一个`NullPointerException`异常。

代码清单6-31 抛出后通知的示例程序

```
public class AfterThrowingDemo {  
  
    public static void main(String[] args) {  
        ApplicationContext ac = new ClassPathXmlApplicationContext(  
            "/META-INF/spring/afterthrowingdemo1-context.xml"  
        );  
        UserService userService = (UserService) ac.getBean("userService");  
        userService.login("janm");  
  
        StockService stockService = (StockService) ac.getBean("stockService");  
        System.out.println(stockService.getStockLevel(null));  
    }  
}
```

即使已经编写了抛出后通知（并且在`afterthrowingdemo1-context.xml`中正确配置了这些bean），这个通知也不会被执行，因为其参数中的异常类型是`IOException`，而不是`NullPointerException`。如果我们将通知的参数改为`Exception`，那它便能正常工作了，打印如下输出。

```
...  
Exception in thread "main" java.lang.NullPointerException  
After method call of java.lang.NullPointerException  
at com.apress.prospring2.ch06.services.StockService.➡  
getStockLevel(StockService.java:9)  
...
```

4. 后置通知

后置通知的最后一类是后置，或更正式来说是最终通知。无论目标方法正常完成还是抛出异常它都会执行。然而，我们得不到有关方法的返回值或任何已抛出异常的任何信息。大多数情况下，可以使用最终通知来释放资源，就像在Java中使用`try/catch/finally`一样。

5. 包围通知

包围通知是最强大且最复杂的通知，它围绕着方法调用而执行。因此，该通知需要至少一个参数，并且必须返回一个值。这个参数指定被调用的目标，而返回值指定目标的返回值，不用管返回值从何而来。你会发现包围通知一般用于事务的管理。通知会在进行目标调用之前开始一个事务，如果目标正常返回，则通知会提交该事务。如果碰上异常，则会进行回滚操作。包围通知的另外一个例子是缓存，代码清单6-32中的代码显示了一个简单的缓存包围通知。

代码清单6-32 缓存包围通知

```
@Aspect
public class CachingAspect {
    private Map<MethodAndArguments, Object> cache =
        Collections.synchronizedMap(
            new HashMap<MethodAndArguments, Object>());
    private Object nullValue = new Object();
    private static class MethodAndArguments {
        private Object target;
        private Object[] arguments;

        private MethodAndArguments(Object target, Object[] arguments) {
            this.target = target;
            this.arguments = arguments;
        }

        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            MethodAndArguments that = (MethodAndArguments) o;

            return Arrays.equals(arguments, that.arguments) &&
                target.equals(that.target);
        }

        public int hashCode() {
            int result;
            result = target.hashCode();
            result = 31 * result + Arrays.hashCode(arguments);
            return result;
        }
    }

    @Around("execution(* com.apress.prospring2.ch06.services.*(..))")
    public Object cacheCalls(ProceedingJoinPoint pjp) throws Throwable {
        Object cacheRet;
        final MethodAndArguments methodAndArguments =
            new MethodAndArguments(pjp.getTarget(), pjp.getArgs());
        cacheRet = this.cache.get(methodAndArguments);
        if (cacheRet == this.nullValue) return null;
        if (cacheRet == null) {
            Object ret = pjp.proceed();
            cacheRet = ret;
            if (cacheRet == null) cacheRet = this.nullValue;
            this.cache.put(methodAndArguments, cacheRet);
            return ret;
        }
        return cacheRet;
    }
}
```

这个方面中实现的缓存当然不是一个企业级缓存，但它能很好地展示包围通知。它采用 `ProceedingJoinPoint` 参数并返回一个 `Object`。该参数能提供我们想知道的有关目标方法执行的所有信息，并且调用代码会接收到通知返回的值。

你或许想知道是否必须要调用目标方法，或者是否可以多次对其进行调用。答案是肯定的。我们可以在 `advice` 里随心所欲：我们能让通知拥有处理重试的能力，它可以在放弃之前调用3次目标方法。

或者就像在代码清单6-32中看到的那样，我们可以完全跳过对目标方法的调用！

6.2.6 参数绑定

你可能已经注意到，有时我们需要知道从通知传入到目标方法的参数值。到目前为止，我们只看到显式地访问返回值。我们也可以使用`JoinPoint.getArgs()`来访问参数。即便这么做是可行的，但仍感觉工作量太大，因为这些参数是以一个`Object`数组传入的，我们不得不进行边界检查和自我转换。幸运的是，`@AspectJ`支持绑定，我们可以将一个目标的值绑定到通知的一个参数上。作为一个示例，来看一下代码清单6-33中的代码。

代码清单6-33 一个带有边界参数的Aspect

```
@Aspect
public class BindingAspect {

    @Around(value =
        "execution(* com.apress.prospring2.ch06.services.StockService.*(..)) " +
        "&& args(cutoffDate, minimumDiscount)",
        argNames = "pjp, cutoffDate, minimumDiscount")
    public Object discountEnforcement(ProceedingJoinPoint pjp, Date cutoffDate,
                                      BigDecimal minimumDiscount)
        throws Throwable {
        return pjp.proceed();
    }
}
```

在这里，可以看到`discountEnforcement`通知将会在`StockService`类中任意方法执行时执行（`execution`子句），只要它包含两个参数（`args`子句），`cutoffDate`参数的类型为`Date`并且`minimumDiscount`参数的类型为`BigDecimal`（根据通知方法的参数来判断）就行。除了从目标传入的两个参数外，通知方法还会接受`ProceedingJoinPoint`实例。另外还要注意`@Around`注解的`argNames()`属性，它明确指定通知方法的参数名为`pjp`、`cutoffDate`和`minimumDiscount`。`Spring AOP`基础设施将根据`argNames()`来决定哪个参数从切入点表达式中得到边界值。

提示 `argNames()`方法存在的原因是，一旦Java源代码被编译成字节码，就有可能判定出参数名。我们只能获取它们的索引和类型。不过我们是在切入点和绑定表达式中用到参数名。因此`argNames()`方法表示了一种能将参数名解析成它的索引的机制。

我们可以在`argNames()`属性中省略`JoinPoint`（及其子接口），如果它作为通知方法的第一个参数出现。尽管如此，我们不推荐这么做，因为我们很容易就会忘记`JoinPoint`应该是第一个参数。除此之外，一些IDE也会在`argNames`中的名称没有匹配通知方法的参数时报错，如图6-4所示。


```
@Around(value =
    "execution(* com.apress.prospring2.ch06.services.StockService.*(..)) &&" +
    "args(cutoffDate, minimumDiscount)",
    argNames = "cutoffDate, minimumDiscount")
    |
    | argNames should match formal method parameter names more...(%F1)
    | argNames should match formal method parameter names more...(%F1)
    |
    public Object discountEnforcement(ProceedingJoinPoint pjp, Date cutoffDate,
        BigDecimal minimumDiscount)
        throws Throwable {
        return pjp.proceed();
    }
```

图6-4 IntelliJ IDEA中的一个argNames()报错

如果你觉得argNames有一点笨拙，也可以不指定它，Spring AOP将使用类文件中的调试信息来判定参数名。通常情况下，这是默认不可用的，但我们可以使用-g:vars javac命令打开参数名调试信息。允许Spring AOP判定参数名称也会让编译后的*.class文件大小稍有增加，但这通常不是什么问题。

更重要的问题可能是通过-g:vars命令编译的*.class文件在反编译后将具有更好的可读性，并且编译器也不能执行优化来删除没用到的参数。如果我们正从事于开源应用程序，第一个问题不会有什么麻烦。如果仔细编写代码，避免留下多余的无用参数，第二个问题倒也没那么糟糕。如果我们的类文件没有必要的调试信息，Spring会尝试使用它们自己的类型来匹配参数。在代码清单6-33的示例中，Spring能轻易地进行这种匹配：ProceedingJoinPoint跟Date或BigDecimal并不继承自同一个类。同样地，Date和BigDecimal也是不同的类。如果Spring不能使用它们的类型安全地匹配这些参数（例如，被通知方法将ProceedingJoinPoint作为它的参数），它将会抛出一个AmbiguousBindingException异常。

6.2.7 引入

引入是Spring提供的AOP功能的重要组成部分。使用引入可以动态地在现有对象中加入新的功能。Spring中，可以在现有对象上引入任何接口的实现。你可能在想，这到底有什么用处，我们可以简单地在开发时添加一个功能，为什么还需要在运行时动态地引入？这个问题的答案很简单。当该功能是一个横切关注点而用传统面向对象方法难以实现的时候，我们就可以动态地添加这个新功能了。

Spring文档中列举了两个典型的引入用法：对象锁定和篡改检测。Spring文档中给出了对象锁定的实现。我们创建了一个Lockable接口，它定义了锁定和解锁对象所需的方法。这个接口用于让程序能够锁定一个对象，以免其内部状态被修改。现在，我们可以手动地为每一个支持锁定的对象实现该接口，不过，这样会造成类之间大量的重复代码。当然，我们可以把该实现重构成一个抽象类，不过这样就不得不使用实体继承，并用掉了唯一的一次继承名额，而且还得在每一个修改对象状态的方法中检查该对象是否已被锁定。显然这种解决办法并不理想，而且有可能会造成很多问题，维护工作也会如同梦魇。

使用引入可以解决所有这些问题。用引入可以把Lockable接口的所有实现集中到一个类中，然后，让所有需要锁定的对象采用这一个Lockable接口的实现。这些对象不仅采用了Lockable的实现，同时也成为Lockable的实例，从而可以通过对Lockable接口的instanceof检查，虽然它们的类并没有实现该接口。

显然,使用引入既解决了集中实现逻辑的问题,又不会影响类的实体继承结构。不过,要编写的检查对象锁定状态的代码那么多怎么办?好办。引入不过是方法拦截器的一个引申而已,因此,它能拦截其目标对象上的任何方法。使用这个功能,我们可以在对设置方法调用之前检查对象的锁定状态,如果已被锁定,就抛出异常。所有这些代码都被封装在一处,而需要锁定的对象可以对此一无所知。

引入是在程序中提供声明性服务的关键。例如,如果你构建的程序能够处理Lockable接口,那么你就能用引入声明性的定义,来标明哪些对象应该是Lockable类型的。

我们不在Lockable接口及其实现上浪费太多时间,因为Spring文档中详细讲解了这个案例。在这里让我们把注意力集中到另外一个文档中提到了但却没有实现的例子上:对象篡改检测。不过在开始之前,让我们先看看构建引入的基础知识。

1. 用引入进行调用跟踪

对象篡改检测是一个有用的技巧。一个典型的例子是在保存数据时用篡改检测来避免对数据库不必要的访问。如果一个对象被传给一个方法作修改,而后者并没有实际修改它,那么就完全没有必要向数据库发出一个update指令。这样,使用篡改检测可以大大提高程序吞吐量,尤其是数据库负荷已经很重或者数据库在远程网络上因而通讯成本高昂时。

可惜的是,这种功能很难手动实现,因为它要求在每个可能修改对象状态的方法中加入检查,以确认对象状态是否真的已被修改。如果我们考虑必须写的所有null检查和状态是否真的改变的检查,那么每个方法大约要加入8行代码。虽然可以把这些重构成一个方法,不过每次需要进行检查时我们还是要调用该方法。将其散布到这个包含大量不同的需要篡改检测的类的典型应用中,并维护这样一个程序将会更加困难。

在这里引入显然很有帮助。我们不想让每个要做篡改检测的类都继承一个基类,因为这样就浪费了唯一的继承机会,而我们也不想向每一个可能修改对象状态的方法中添加检查代码。通过使用引入,可以提供对象篡改检测问题的一个灵活解法,而无需编写大量重复易错的代码。

在这个例子中,我们将使用引入来构建一个统计信息收集框架。篡改检测的逻辑被封装在CallTracker接口中,它的一个实现连同自动篡改检测的拦截逻辑将被一起引入至合适的对象中。

2. CallTracker接口

这个统计方案的核心就是CallTracker接口,我们假想的程序就是用它来跟踪自身的健康状况的。我们不关心程序怎样使用CallTracker接口,而将注意力放在引入的实现上。代码清单6-34显示了CallTracker接口。

代码清单6-34 CallTracker接口

```
public interface CallTracker {  
  
    void markNormal();  
  
    void markFailing();  
  
    int countNormalCalls();  
  
    int countFailingCalls();  
  
    String describe();  
  
}
```

这没什么特别的，以mark开头的方法只用来增加正常调用或失败调用的计数，以count开头的方法返回对应的计数器值。

3. 创建一个混入体

下一步是编写实现CallTracker接口的代码，并将其引入到对象中去。这被称为混入体（mix in）。代码清单6-35显示了CallTracker接口的实现。同样，这个实现也非常简单。

代码清单6-35 DefaultCallTracker类

```
public class DefaultCallTracker implements CallTracker {  
    private int normalCalls;  
    private int failingCalls;  
  
    public void markNormal() {  
        this.normalCalls++;  
    }  
  
    public void markFailing() {  
        this.failingCalls++;  
    }  
  
    public int countNormalCalls() {  
        return this.normalCalls;  
    }  
  
    public int countFailingCalls() {  
        return this.failingCalls;  
    }  
  
    public String describe() {  
        return toString();  
    }  
  
    @Override  
    public String toString() {  
  
        final StringBuilder sb = new StringBuilder();  
        sb.append("DefaultCallTracker");  
        sb.append("{normalCalls=").append(normalCalls);  
        sb.append(", failingCalls=").append(failingCalls);  
        sb.append('}');  
        return sb.toString();  
    }  
}
```

206 第6章 AOP 进阶

要注意的第一件事就是CallTracker的实现，它包括私有的normalCalls和failingCalls字段。这个示例突出每个被通知对象都要有一个混入体的重要性：混入体不仅向对象中加入方法，还可以加入状态。如果多个不同的对象共享一个混入体实例，那么它们也共享这个状态。也就是说，一旦一个对象被修改，所有的对象都会显示成被修改的。

4. 创建混入体方面

下一步是创建一个包含后置和抛出后通知的方面以及混入体的声明。代码清单6-36显示了我们编写的代码。

代码清单6-36 为混入体创建通知

```
@Aspect
public class CallTrackerAspect {

    @Pointcut("execution(* com.apress.prospring2.ch06.services.*(..))")
    private void serviceCall() { }

    @DeclareParents(
        value = "com.apress.prospring2.ch06.services.*",
        defaultImpl = DefaultCallTracker.class)
    public static CallTracker mixin;

    @AfterReturning(
        value = "serviceCall() && this(tracker)",
        argNames = "tracker")
    public void normalCall(CallTracker tracker) {
        tracker.markNormal();
    }

    @AfterThrowing(
        value = "serviceCall() && this(tracker)",
        throwing = "t",
        argNames = "tracker, t")
    public void failingCall(CallTracker tracker, Throwable t) {
        tracker.markFailing();
    }
}
```

这个方面中的新增代码是mixin字段的@DeclareParents注解。它声明我们将使用DefaultCallTracker实现把CallTracker接口(字段的类型)引入到com.apress.prospring2.ch06.services包下所有的类中。

接着，我们定义了一个@pointcut——serviceCall()（因为我们将在两个独立的通知中使用它）。

然后我们使用切入点serviceCall() && this(tracker)创建一个后置通知。serviceCall()代表着@pointcut，而this(tracker)将匹配实现了CallTracker接口的对象上所有方法的执行（因为在通知方法参数里，tracker表达式的类型为CallTracker）。Spring AOP会将追踪器参数绑定到被通知对象上。因此，可以在通知体中使用tracker参数。

最后，我们使用相同的切入点表达式（serviceCall() && this(tracker)）编写抛出后通知。Spring AOP会将跟踪器参数绑定到failingCall通知方法的参数上。除了CallTracker参数外，我们在@AfterThrowing注解中定义我们想要接收抛出的异常作为failingCall方法的第二个参数。

Spring AOP会根据参数的类型（在此例中为Throwable）推断出异常的类型。

5. 把所学的放在一起

现在我们已经具备引入混入体和合适通知的方面了，可以编写一个使用已经实现的UserService和StockService bean的示例程序。代码清单6-37显示了此示例程序的源代码。

代码清单6-37 引入示例程序

```
public class IntroductionDemo {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/META-INF/spring/introductionsdemo1-context.xml"
        );
        UserService userService = (UserService) ac.getBean("userService");
        describeTracker(userService);
        userService.login("janm");
        userService.setAdministratorUsername("x");
        describeTracker(userService);

        StockService stockService = (StockService) ac.getBean("stockService");
        describeTracker(stockService);
        try {
            stockService.getStockLevel(null);
        } catch (Exception ignored) {

        }

        System.out.println(stockService.getStockLevel("ABC"));
        stockService.applyDiscounts(new Date(), new BigDecimal("10.0"));
        describeTracker(stockService);
    }

    private static void describeTracker(Object o) {
        CallTracker t = (CallTracker)o;
        System.out.println(t.describe());
    }

}
```

注意这里我们跟平常一样使用 userService 和 stockService bean：先调用 userService.login ("janm")，接着调用 userService.setAdministratorUsername ("x")。对 stockService bean 上的方法调用也大体相似。然而，现在可以将 userService 和 stockService 转换成 CallTracker，它们现在实现了新的引入接口。我们不需要编写除了代码清单6-38中的标准 XML 配置文件之外任何额外的配置，相信你不会对此太吃惊。

代码清单6-38 引入演示的XML配置文件

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



```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean id="userService"
    class="com.apress.prospring2.ch06.services.DefaultUserService"/>
<bean id="stockService"
    class="com.apress.prospring2.ch06.services.DefaultStockService"/>
<bean class="com.apress.prospring2.ch06.introductions.CallTrackerAspect"/>

<aop:aspectj-autoproxy />

</beans>
```

这实际上就是全部要做的。Spring AOP 会代理 `DefaultUserService` 和 `DefaultStockService`。让我们来更仔细地检查 `DefaultUserService`：它只实现了 `userService` 接口，但是使用 `CallTrackerAspect` 的 `userService` bean 类型是 `JdkDynamicAopProxy`，它不是一个 `DefaultUserService` 的实例。这个代理双双实现了 `UserService` 接口和 `CallTracker` 混入体接口。它拦截对所有方法的调用并将 `UserService` 接口上的调用委托给 `DefaultUserService`。这个代理也创建了一个 `DefaultCallTracker`（在 `@DeclareParents` 注解中被定义）的实例并将 `CallTracker` 接口上所有的调用委托给 `DefaultCallTracker` 实例。

运行此示例程序得到的输出理应在意料之中：

```
before userService.login("janm"):
    DefaultCallTracker{normalCalls=0, failingCalls=0}

after userService.setAdministratorUsername("x"):
    DefaultCallTracker{normalCalls=2, failingCalls=0}

before stockService.getStockLevel(null):
    DefaultCallTracker{normalCalls=0, failingCalls=0}
193734

after stockService.applyDiscounts(...):
    DefaultCallTracker{normalCalls=2, failingCalls=1}
```

此输出证实了 `CallTracker` 接口的确被引入到 `com.apress.prospring2.ch06.services` 包中的所有类中，也证实了所有的被通知类只对应一个 `DefaultCallTracker` 实例。

6. 引入小结

引入是 Spring AOP 最强大的功能之一。它们不仅允许我们扩展现有方法的功能，还允许动态地扩展对象实现的接口集。当程序和横切逻辑通过定义好的接口交流时，使用引入是实现该横切的绝好办法。总的来说，这就是那些我们希望声明性而非编程性控制的逻辑。

显然，引入是通过代理执行的，所以它会带来一些运行时的额外开销。另外，因为代理需要在运行时将每一个调用路由到合适的目标，所以代理上所有的方法都可以看作是被通知的。不过，相对于用引入所能实现的很多服务，这点儿性能开销是微不足道的。我们减少了实现服务需要的代码量，同时也从服务逻辑的完全集中化和添加到应用程序中的管理接口中获得较好的稳定性和可维护

性。

6.2.8 方面的生命周期

到目前为止，我们所写的方面都是单例的。更准确的说，Spring采用单例的@Aspect注解bean，并使用单例方面来通知目标。使用代码清单6-39中的方面将其并用于我们的示例程序。

代码清单6-39 简单的单例方面

```
@Aspect
public class CountingAspect {
    private int count;

    @Before("execution(* com.apress.prospring2.ch06.services.*(..))")
    public void count() {
        this.count++;
        System.out.println(this.count);
    }
}
```

让我们来完成示例，如代码清单6-40显示的演示程序所示。它获取我们熟悉的userService和stockService bean，并对userService.login()方法进行了两次调用，然后调用了一次stockService.getStockLevel()方法。

代码清单6-40 单例方面的示例程序

```
public class LifecycleDemo {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/META-INF/spring/lifecycledemo1-context.xml"
        );
        UserService userService = (UserService) ac.getBean("userService");
        StockService stockService = (StockService) ac.getBean("stockService");

        for (int i = 0; i < 2; i++) {
            userService.login("janm");
        }
        stockService.getStockLevel("A");
    }
}
```

最后是XML配置文件，它只是简单地声明了userService、stockService和CountingAspect bean。下面看一下代码清单6-41。

代码清单6-41 LifecycleDemo程序的XML配置

210 第6章 AOP 进阶

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

  <bean id="userService"

    class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
    class="com.apress.prospring2.ch06.services.DefaultStockService"/>
  <bean class="com.apress.prospring2.ch06.lifecycle.CountingAspect"/>

  <aop:aspectj-autoproxy />

</beans>
```

现在，当我们运行这个程序，会打印出“1, 2, 3”，这说明方面是以一个单例存在的。如果我们想要为不同目标保持状态，就需要将方面bean的scope改为prototype。如果将代码清单6-41的XML配置文件中的CountingAspect bean定义修改为如下形式。

```
<bean class="com.apress.prospring2.ch06.lifecycle.CountingAspect"
      scope="prototype"/>
```

再次运行示例程序，会打印出“1, 2, 1”。程序现在为所有目标只维护一个方面的实例。实际上，我们已经实现了perthis方面。可以显式地在@Aspect注解中使用一个perthis表达式：

```
@Aspect("perthis(execution(" +
    "* com.apress.prospring2.ch06.services.UserService.*(..))")")
public class PertargetCountingAspect {
    ...
}
```

修改后的结果是Spring AOP将为每一个执行被通知对象的独立对象创建方面的一个新实例。

Spring AOP支持的另一个生命周期策略是pertarget。它将为每一个在匹配连接点上的独立目标对象创建一个新的方面的实例，除此之外它跟perthis基本相似。

6.3 AOP 的框架服务

到目前为止，我们已经使用Spring中的@AspectJ支持来编写方面。我们依赖潜在的Spring AOP框架服务来处理带注解的bean并转化它们。如果不能使用注解（也许是因为运行的1.5之前版本的JDK，也许只是因为不喜欢注解），那么可以利用AOP命名空间来使用XML配置。在本节中，我们将讨论XML配置并揭示了在Spring AOP中的@AspectJ支持是如何将带注解的bean转化成Spring AOP支持的格式。

6.3.1 使用 AOP 命名空间创建第一个方面

下面，我们编写一个跟代码清单6-1中类似的方面，但这次只使用XML配置。我们从代码清单6-42开始，它显示了一个方面的XML配置。

代码清单6-42 XML配置文件

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

  <bean id="userService"
    class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
    class="com.apress.prospring2.ch06.services.DefaultStockService"/>

  <bean id="aspectBean" class="com.apress.prospring2.ch06.xml.AspectBean"/>

  <aop:config>
    <aop:pointcut id="serviceCall"
      expression="execution(* com.apress.prospring2.ch06.services.*(..))"/>

    <aop:aspect id="firstAspect" ref="aspectBean">
      <aop:before method="logCall" pointcut-ref="serviceCall"/>
    </aop:aspect>
  </aop:config>

</beans>
```

加粗的代码声明了一个切入点，它可以匹配`com.apress.prospring2.ch06.services`包下任何类中任何参数的任何方法的执行。我们编写的代码跟@AspectJ代码相似，只需展示代码清单6-43中AspectBean的代码。

代码清单6-43 AspectBean的代码

```
public class AspectBean {

    public void logCall(JoinPoint jp) {
        System.out.println(jp);
    }

}
```

以上代码跟@AspectJ代码相似，区别在于前者没有任何注解。代码清单6-44中的示例程序使用XML配置并调用业已熟知的userService和stockService bean上的方法。

代码清单6-44 XML AOP支持的示例程序

212 第6章 AOP 进阶

```
public class XmlDemo1 {  
  
    public static void main(String[] args) {  
        ApplicationContext ac = new ClassPathXmlApplicationContext(  
            "META-INF/spring/xmldemo1-context.xml"  
        );  
        UserService userService = (UserService) ac.getBean("userService");  
        StockService stockService = (StockService) ac.getBean("stockService");  
  
        userService.login("janm");  
        stockService.getStockLevel("A");  
    }  
}
```

示例程序中的代码跟@AspectJ方面的代码保持一致。换句话说，XML配置是完全透明的，而调用代码全然不知它正在调用被通知bean上的方法。运行示例程序得到如下输出：

```
> userService.login("janm"):  
    org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: ➡  
  
    execution(login)  
  
> stockService.getStockLevel("A"):  
    org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: ➡  
    execution(getStockLevel)
```

可以看到userService和stockService均被正确地通知到，并且AspectBean.logCall()方法获得了执行。

6.3.2 AOP 命名空间中的切入点

跟6.2节类似，我们将讨论切入点定义时用到的配置元素。要定义一个切入点，需要将<aop:pointcut . . ./>元素写作<aop:config>元素的一个子元素。让我们进一步看看代码清单6-43中代码的细节，代码清单6-45显示了XML配置文件中的一段。

代码清单6-45 XML配置文件的AOP部分

```
...  
    <aop:config>  
        <aop:pointcut id="serviceCall"  
            expression="execution(* com.apress.prospring2.ch06.services.*.*(..))"/>  
  
        <aop:aspect id="firstAspect" ref="aspectBean">  
            <aop:before method="logCall" pointcut-ref="serviceCall"/>  
        </aop:aspect>  
    </aop:config>  
...
```

加粗的代码表明有一个id为serviceCall的切入点，我们将其用于firstAspect的前置通知的定义中。跟@AspectJ支持类似，我们可以在切入点表达式中使用@pointcut。代码清单6-46显示了一个熟悉的@pointcut定义。

代码清单6-46 @pointcut的定义


```
public final class Pointcuts {  
    private Pointcuts() {  
  
    }  
  
    @Pointcut("execution(* com.apress.prospring2.ch06.services.*(..))")  
    public void serviceExecution() { }  
  
}
```

我们现在可以在 XML 配置中通过简单修改 `<aop:pointcut . . . />` 元素来使用 `Pointcuts.service- Execution()` 这个 `@pointcut` 了。

```
<aop:pointcut id="serviceCall"  
    expression="com.apress.prospring2.ch06.xml.Pointcuts.  
    serviceExecution()"/>
```

唯一的不足是又使用注解了。我们提倡在应用程序中只使用一种配置风格并坚持下去，混用 `@AspectJ` 和 XML 的配置会迅速导致难以管控的复杂。

回到纯粹的 XML 配置上来，你可以使用所有 Spring AOP 支持的切入点表达式，而且一些具备代码完成功能的 IDE 甚至支持基于 XML 的 Spring AOP（如图 6-5）。



图6-5 IntelliJ IDEA中XML AOP配置的代码完成功能

6.3.3 使用 AOP 命名空间创建通知

XML配置里的通知跟`@AspectJ`的通知相似。唯一的不同在于表示方面的bean就是简单的Spring bean，而且它的配置被分散在XML文件和Java代码中。aop命名空间跟`@AspectJ`支持一样强大。我们已经在代码清单6-42中见到过一个简单前置通知的例子了。让我们详细讨论XML的通知。跟`@AspectJ`的通知一样，XML的通知需要一个切入点。你可以引用一个已有的切入点（在通知的定义中使用`pointcut-ref`属性），或者直接指定一个切入点表达式（在通知的定义中使用`pointcut`属性）。代码清单6-47显示了这个前置通知的等价定义。

代码清单6-47 使用`pointcut-ref`和`pointcut`属性

214 第6章 AOP 进阶

```
...
<aop:aspect id="firstAspect" ref="aspectBean">
  <aop:before method="logCall" pointcut-ref="serviceCall"/>
</aop:aspect>

<aop:aspect id="firstAspect" ref="aspectBean">
  <aop:before method="logCall"
    pointcut="execution(* com.apress.prospring2.ch06.services.*(..))"/>
</aop:aspect>
...
```

差异很明显，在第一段中，我们使用<pointcut . . . />元素引用了一个已有的切入点。在第二段中，我们指定了一个切入点表达式。接下来，让我们来了解AOP命名空间中通知的更多细节。

1. 前置通知

要声明前置通知，需要创建<before>元素作为<aspect>元素的子元素。<before>元素的属性是method、pointcut（或pointcut-ref）和arg-names。你必须设置method和pointcut（或pointcut-ref）的属性值。method属性是一个你在<aspect>元素中引用的bean上的公共方法名，通过JoinPoint，其参数可以从没有参数到你在切入点表达式中绑定的所有参数。如果正在使用绑定，你应该考虑使用arg-names属性以帮助Spring AOP框架正确识别边界参数。

2. 后置通知

要创建后置通知，应创建一个<after-returning>元素作为<advice>元素的子元素。跟前置通知一样，应该为method和pointcut（或者pointcut-ref）设置属性值。多数情况下，你可能还想知道返回值。要达到这个目的，将returning属性设置成此方法的Object参数的名称。最后，也可以使用arg-names来指定method属性中的方法的参数名。让我们看看代码清单6-48中的代码，它显示了我们如何创建这样一个方面，它会统计对每一个方法的调用并对某些返回值进行审查。

代码清单6-48 使用后置通知

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

  <bean id="userService"
    class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
    class="com.apress.prospring2.ch06.services.DefaultStockService"/>

  <bean id="aspectBean" class="com.apress.prospring2.ch06.xml.AspectBean"/>
```

```
<aop:config>
  <aop:aspect id="firstAspect" ref="aspectBean">
    <aop:after-returning method="auditCall"
      pointcut="execution(* com.apress.prospring2.ch06.services.*(..)) && target(target)"
      arg-names="target, ret"
      returning="ret"/>
  </aop:aspect>
</aop:config>

</beans>
```

要注意，我们的切入点表达式指定了`services`包里任意类中任意方法的执行过程，并且我们将调用目标绑定到一个名为`target`的参数上。我们也会接收到目标从`AspectBean.auditCall()`方法上返回的返回值。注意`arg-names`属性，我们能清楚地看到`auditCall`方法需要两个参数，而且因为我们希望接收一个任意类型的目标并处理任意类型的返回值，因此参数的类型必须为`Object`。现在我们可以编写`AspectBean.auditCall()`方法的代码了（如代码清单6-49所示）。

代码清单6-49 AspectBean的auditCall方法

```
public class AspectBean {

    public void auditCall(Object target, Object ret) {
        System.out.println("After method call of " + ret);
        if (ret instanceof User) {
            ((User)ret).setPassword("****");
        }
    }

}
```

6

现在我们有了一个包含两个参数`target`和`ret`的`auditCall`方法。注意代码清单6-48中的XML配置以正确的顺序指定了参数名。这很容易弄错。幸好一些IDE会检测错误并给出一个正确建议。图6-6显示了当我们将参数的正确顺序弄颠倒时IntelliJ IDEA给出的错误纠正。

```
<aop:config>
  <aop:aspect id="firstAspect" ref="aspectBean">
    <aop:after-returning method="auditCall"
      pointcut="execution(* com.apress.prospring2.ch06.services.*(..)) && target(target)"
      arg-names="ret, target"
      returning="ret"/>
  </aop:aspect>
</aop:config>
```

arg-names should match formal method parameter names more... (XF1)

图6-6 IntelliJ IDEA中arg-names的错误检测

可以看出`<after-returning>`元素给出了与`@AspectJ`中的`@AfterReturning`注解相同的配置项。

3. 抛出后通知

我们已经知道，抛出后通知只有在匹配的方法抛出一个异常时才会执行。为了声明性地创建抛出后通知，我们在`<aspect>`元素中使用`%(after-throwing%)`。我们需要至少设置`method`、`pointcut`（或`pointcut-ref`）和`throwing`属性。大多数情况下，我们也要设置`arg-names`属性。事不宜迟，让我们看看代码清单6-50中的XML配置。

216 第6章 AOP 进阶

代码清单6-50 XML中的抛出后通知

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

  <bean id="userService"
    class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
    class="com.apress.prospring2.ch06.services.DefaultStockService"/>

  <bean id="aspectBean" class="com.apress.prospring2.ch06.xml.AspectBean"/>

  <aop:config>
    <aop:aspect id="afterThrowingAspect" ref="aspectBean">
      <aop:after-throwing method="healthMonitoring"
        pointcut="execution(* com.apress.prospring2.
          ch06.services.*(..) &*&*&*)
        target(target)"
        arg-names="target,ex"
        throwing="ex"/>
    </aop:aspect>
  </aop:config>
</beans>
```

这个抛出后通知将调用目标和抛出的异常绑定到`healthMonitoring`方法的参数，这意味着`healthMonitoring`方法需要包含两个参数。要捕获任何目标抛出的任何异常，参数的类型必须是`Object`和`Throwable`。不过为了展示基于参数类型的过滤，我们采用代码清单6-51中的代码。

代码清单6-51 抛出后方法和参数过滤

```
public class AspectBean {

    public void healthMonitoring(Object target, NullPointerException ex) {
        System.out.println("Target " + target + " has thrown " + ex);
    }

}
```

注意到第二个参数类型为`NullPointerException`，因此此通知只有在目标方法抛出`NullPointerException`时才会执行。为演示这个抛出后通知，我们将采用代码清单6-44中的代码，与其不同之处是我们调用如下方法：

```
stockService.getStockLevel(null);
```

4. 后置通知

后置通知（或者说最终通知）不管目标方法是正常结束还是抛出异常，都会在目标方法完成后执行。我们使用`<aspect>`元素中的`<after>`元素来声明后置通知。因为这是后置通知，所以只需设置`method`和`pointcut`（或`pointcut-ref`）属性。如果使用了参数绑定，那么我们可以使用`arg-names`

属性。鉴于这类通知的性质，我们不能使用`returning`和`throwing`属性。代码清单6-52展示了一个简单后置通知的示例。

代码清单6-52 配置后置通知

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

  <bean id="userService"
        class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
        class="com.apress.prospring2.ch06.services.DefaultStockService"/>

  <bean id="aspectBean" class="com.apress.prospring2.ch06.xml.AspectBean"/>

  <aop:config>
    <aop:aspect id="afterAspect" ref="aspectBean">
      <aop:after-throwing method="after"
        pointcut="execution(* com.apress.prospring2.ch06.services.*(..))
          & target(target)"
        arg-names="target"/>
    </aop:aspect>
  </aop:config>

</beans>
```

这个通知的方法(`AspectBean.after`)只需要一个参数，并且因为只想通知`UserService` bean，我们将其声明为`public void after(UserService target)`。

5. 包围通知

我们将这个最通用的通知留到最后介绍。顾名思义，包围通知包围在匹配方法执行的前后运行。因此我们编写代码时可以让其在匹配方法之前执行（我们可以操纵方法参数甚至完全跳过匹配方法的执行），也可以让其在匹配方法完成后执行。我们可以自由地使用标准的`try/catch`块来捕捉任何异常，也可以操纵返回值，只要匹配方法的返回值能赋给返回的类型（比如，如果目标方法返回`Number`类型，我们可以从通知中返回`Long`类型，但不能是`String`类型）。声明包围通知，可以使用`<aspect>`元素中的`<around>`元素。我们必须设置`method`和`pointcut`（或`pointcut-ref`）的属性值，还可能设置`arg-names`属性。代码清单6-53中的代码显示了我们的包围通知。

代码清单6-53 包围通知的XML配置

218 第6章 AOP 进阶

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

  <bean id="userService"
    class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
    class="com.apress.prospring2.ch06.services.DefaultStockService"/>

  <bean id="aspectBean" class="com.apress.prospring2.ch06.xml.AspectBean"/>

  <aop:config>
    <aop:aspect id="aroundAspect" ref="aspectBean">
      <aop:around method="censorStringArguments"
        pointcut="execution(* com.apress.prospring2.ch06.
          services.*(..)) and
          args(argument)"
        arg-names="pjp, argument"/>
    </aop:aspect>
  </aop:config>

</beans>
```

注意此处定义了两个参数，但我们在切入点表达式中压根没有使用pjp参数。这个包围通知的方法必须至少包含ProceedingJoinPoint参数（否则我们就调用不了匹配方法！）并且必须返回Object。除了ProceedingJoinPoint参数外，我们可以自由地按照意愿使用任意多的边界参数。在这个示例中，我们实现了一个审查通知方法（源代码如代码清单6-54所示）。

代码清单6-54 包围通知的方法

```
public class AspectBean {

    public Object censorStringArguments(ProceedingJoinPoint pjp, String argument)
        throws Throwable {

        Object[] arguments;
        if (argument != null) {
            System.out.println("censored " + argument + "!");
            arguments = new Object[] { "*****" };
        } else {
            arguments = new Object[] { null };
        }
        return pjp.proceed(arguments);
    }

    ...
}
```

在这里，我们可以看到方法的参数匹配arg-names的属性值并且第二个参数的类型是String，这意味着我们只能匹配具有一个String类型参数的方法。在我们这个小程序中，它们就是UserService.setAdministratorUsername、UserService.login和StockService.getStockLevel。

现在我们已经涵盖了使用XML配置的所有类型的通知，我们将介绍最后一部分——引入。

6.3.4 AOP 命名空间中的引入

这一节中最后要介绍的内容是引入。回想一下，引入通过代理被通知对象，向已有对象中添加额外的方法，并让代理实现那些接口，它们是在原有实现上声明的接口。要使用XML引入，我们要使用<aspect>元素下的<declare-parents>元素。要真正使用新声明的父类（比如说一个新实现的接口），需要创建至少一个使用被引入的接口的通知。我们将再次使用代码清单6-36中的示例，我们会把CallTracker接口引入到DefaultUserService和DefaultStockService中。代码清单6-55显示了引入CallTracker接口的XML配置。

代码清单6-55 在XML中声明引入

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

    <bean id="userService"
          class="com.apress.prospring2.ch06.services.DefaultUserService"/>
    <bean id="stockService"
          class="com.apress.prospring2.ch06.services.DefaultStockService"/>

    <bean id="aspectBean" class="com.apress.prospring2.ch06.xml.AspectBean"/>

    <aop:config>
        <aop:aspect id="aroundAspect" ref="aspectBean">
            <aop:declare-parents
                types-matching="com.apress.prospring2.
                    ch06.services.*"
                implement-interface="com.apress.prospring2.
                    ch06.introductions.CallTracker"
                default-impl="com.apress.prospring2.ch06.
                    introductions.DefaultCallTracker"/>

            <aop:after-returning method="normalCall"
                arg-names="tracker"
                pointcut="execution(* com.apress.prospring2.
                    ch06.services.*(..)) and this(tracker)"/>
            <aop:after-throwing method="failingCall"
                arg-names="tracker"
                pointcut="execution(* com.apress.prospring2.
                    ch06.services.*(..)) and this(tracker)"/>
        </aop:aspect>
    </aop:config>

</beans>
```

在 <declare-parents> 元素里，types-matching 属性指定了我们想要将在 implement-interface 定义的接口引入到哪些类。要完成 <declare-parents> 元素，我们需要指

220 第6章 AOP 进阶

定default-impl——它表示实现了implement-interface中的接口的类名。在此处，我们可以写成如下所示：

```
CallTracker usct = (CallTracker)ac.getBean("userService");
```

假设ac是代码清单6-55中XML配置文件中ApplicationContext的实例，这说明userService bean现在实现了CallTracker接口，尽管DefaultUserService（bean的真正类）只实现了UserService接口。唯一的问题是程序现在追踪不到调用了。DefaultCallTracker类中的normalCalls和failingCalls字段的值永远不会变。为了完成这个示例，我们需要创建两个通知：后置通知和抛出后通知。代码清单6-56显示了二者的配置。

代码清单6-56 后置通知和抛出后通知

```
...
<aop:config>
  <aop:aspect id="aroundAspect" ref="aspectBean">
    <aop:declare-parents ... />

    <aop:after-returning method="normalCall"
      arg-names="tracker"
      pointcut="execution(* com.apress.prospring2.
        ch06.services.*(..)) and this(tracker)"/>
    <aop:after-throwing method="failingCall"
      arg-names="tracker"
      pointcut="execution(* com.apress.prospring2.
        ch06.services.*(..)) and this(tracker)"/>
  </aop:aspect>
</aop:config>
...
```

要完成这两个通知，我们需要实现AspectBean中的normalCall和failingCall方法。两个后置通知都将arg-names属性值设置为tracker，所以这些方法需要一个名为tracker的参数，它的类型应该为CallTracker。我们会使用它的方法来对调用进行计数。代码清单6-57显示了这两个通知方法的实现。

代码清单6-57 实现后置通知的方法

```
public class AspectBean {
...
  public void normalCall(CallTracker tracker) {
    tracker.markNormal();
  }

  public void failingCall(CallTracker tracker) {
    tracker.markFailing();
  }
...
}
```

这两个方法非常简单，它们使用追踪器参数来增加对应的调用计数器的值。我们将在代码清单6-58的演示程序中完成它。这个程序在userService和stockService bean上调用方法，并使用引入的CallTracker接口将调用的统计结果显示出来。

代码清单6-58 引入的示例程序

```
public class XmlDemo6 {  
  
    public static void main(String[] args) {  
        ApplicationContext ac = new ClassPathXmlApplicationContext(  
            "/META-INF/spring/xmldemo6-context.xml"  
        );  
        UserService userService = (UserService) ac.getBean("userService");  
        StockService stockService = (StockService) ac.getBean("stockService");  
  
        userService.login("janm");  
        stockService.getStockLevel("A");  
        stockService.applyDiscounts(new Date(), BigDecimal.ONE);  
        describeTracker(userService);  
        describeTracker(stockService);  
    }  
  
    private static void describeTracker(Object o) {  
        CallTracker t = (CallTracker)o;  
        System.out.println(t.describe());  
    }  
}
```

程序的正常运行证明我们已成功将CallTracker接口引入到两个bean中。

6.4 风格选择

你也许在想究竟选择哪种Spring AOP风格。@AspectJ容易上手但要求使用JDK 1.5，基于XML的配置可以工作在JDK 1.5之前的版本上。我们建议尽量使用@AspectJ，而经验丰富的Spring开发者也许更熟悉基于XML的配置。但问题在于，使用基于XML的配置，开发者可能并不知道他们正在使用方面，而且XML配置将单一的功能单元分割到两个文件中。此外，我们不能像@AspectJ那样灵活地对切入点进行组合。回想一下，我们可以在XML配置中用一个切入点来表示切入点表达式，也可以使用pointcut-ref属性来表示一个已有切入点表达式的引用。但不能将一个已有切入点的引用和一个切入点表达式组合使用。例如，不能编写像代码清单6-59中那样的代码。

代码清单6-59 切入点和pointcut-ref的不合法组合

```
<aop:config>  
    <aop:pointcut id="x" expression="..." />  
    <aop:aspect ...>  
        <aop:before pointcut="x() and target(y)" />  
    </aop:aspect>  
</aop:config>
```

6.5 使用 Spring AOP 代理

Spring AOP支持使用代理。图6-7显示了一个代理模式的UML类图。

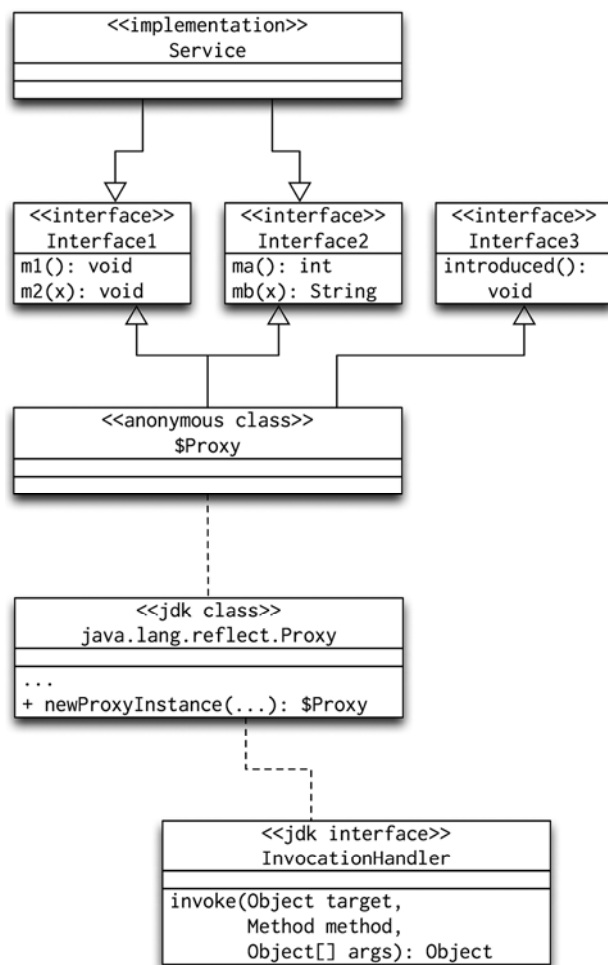


图6-7 一个代理模式的UML类图

上图显示的是一个JDK动态代理。该代理实现了`Interface1`、`Interface2`和`Interface3`。`InvocationHandler.invoke()`的实现处理对代理实现的接口上所有方法调用。在Spring AOP中，`InvocationHandler`持有一个被通知对象的引用。它处理所有对目标方法的前置、后置、抛出后以及包围通知。如果目标对象没有实现任何接口或者你不愿意使用JDK动态代理，那么可以使用CGLIB代理。CGLIB是一个字节码操作库，可以被用来代理一个没有实现任何接口的类。图6-8显示了一个CGLIB代理的类图。

6.5 使用 Spring AOP 代理 223

CGLIB代理实际上就是目标类的子类。因为我们不能重载final方法,在final方法上的通知将无法工作。同样地,目标(接受通知的类)的构造方法将会被调用两次:第一次是创建被通知类时,第二次是创建代理时。这两次构造方法的调用不会产生什么问题。如果我们只是在构造方法中检查参数和设置实例的字段值,那么我们随便调用多少次构造方法都没问题。不过,如果构造方法中包含某些业务逻辑,就可能会产生问题。我们可以通过在<aop:config.../>或<aop:aspectjautoproxy.../>元素中设置proxy-target-class="true"来控制Spring创建的代理类型。如果在配置文件中有不止一个<aop:config.../>或

<aop:aspectjautoproxy.../>元素并且至少其中之一指定了proxy-target-class="true",那么所有的代理都将使用CGLIB,就跟在每一个配置元素中都设定成proxy-target-class="true"一样。

代理的影响

现在让我们来看看代理可能对代码产生的影响。我们将采用StockService接口和它的实现类DefaultStockService作为示例。为了说明这一节中将要讨论的若干概念,我们将对DefaultStockService做一些细微改动。代码清单6-60显示了这些改动。

代码清单6-60 修改后的DefaultStockService

```
public class DefaultStockService implements StockService {  
  
    public long getStockLevel(String sku) {  
        try {  
            Thread.sleep(2000L);  
        } catch (InterruptedException ignored) {  
        }  
        return getPredictedStockLevel(sku) / 2L;  
    }  
  
    public long getPredictedStockLevel(String sku) {  
        return 6L * sku.hashCode();  
    }  
  
    public void applyDiscounts(Date cutoffDate, BigDecimal maximumDiscount) {  
        // do some work  
    }  
}
```

如果我们创建一个未被代理的DefaultStockService实例,然后调用getStockLevel("X")方法,则对getPredictedStockLevel的调用会作用在同一个实例上。代码清单6-61给出了更详尽的演

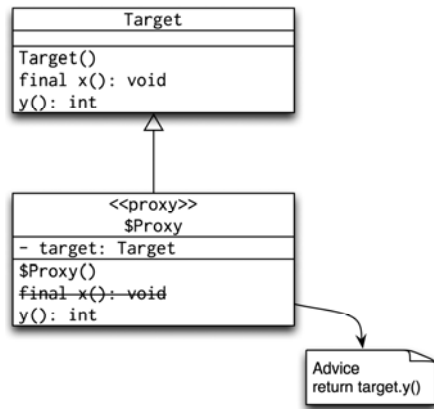


图6-8 CGLIB代理的类图

示。

代码清单6-61 调用未被代理的方法

```
public class ProxyDemo {  
  
    public static void main(String[] args) {  
        StockService dss = new DefaultStockService();  
        dss.getStockLevel("X");  
    }  
  
}
```

当调用`dss.getStockLevel("X")`时，`dss`直接引用了`DefaultStockService`对象，因此当`DefaultStockService.getStockLevel()`调用`DefaultStockService.getPredictedStockLevel()`方法时，效果同在`main()`方法中调用`((DefaultStockService)dss).getPredictedStockLevel()`一样。

接着，试想一个类似的情况，但这次`StockService`接口将作为`DefaultStockService`的一个代理。我们将从JDK动态代理开始。代码清单6-62显示了一个`StockService`的JDK动态代理。

代码清单6-62 StockService的JDK动态代理

```
public class ProxyDemo2 {  
  
    private static class DelegatingInvocationHandler  
        implements InvocationHandler {  
        private Object target;  
  
        private DelegatingInvocationHandler(Object target) {  
            this.target = target;  
        }  
  
        public Object invoke(Object target,  
                             Method method,  
                             Object[] args) throws Throwable {  
            return method.invoke(this.target, args);  
        }  
    }  
  
    public static void main(String[] args) {  
        DefaultStockService targetReference = new DefaultStockService();  
        StockService proxyReference =  
            (StockService) Proxy.newProxyInstance(  
                ProxyDemo2.class.getClassLoader(),  
                new Class<?>[] { StockService.class },  
                new DelegatingInvocationHandler(  
                    targetReference  
                ));  
        proxyReference.getStockLevel("X");  
    }  
  
}
```

当我们调用`proxyReference.getStockLevel("X")`时，实际上是在调用代理。代理的`InvocationHandler`使用`DefaultStockService`实例委托对方法的调用。因此，`proxyReference.getStockLevel()`方法是作用在与调用`DefaultStockService`上的

getPredictedStockLevel()方法不同的实例上的。

在我们讨论代理的实际影响之前，要先了解一下Spring是怎样创建代理的。正如我们在代码清单6-62中所见，即使是创建最简单的代理也有大量工作要做。Spring提供了ProxyFactory类来简化工作，它不仅能为目标对象创建代理，还能添加任何通知。代码清单6-63显示了ProxyFactory的简单用法。

代码清单6-63 使用ProxyFactory

```
public class ProxyDemo3 {  
  
    public static void main(String[] args) {  
        DefaultStockService target = new DefaultStockService();  
        ProxyFactory pf = new ProxyFactory(target);  
        pf.addInterface(StockService.class);  
  
        StockService stockService = (StockService) pf.getProxy();  
        stockService.getStockLevel("A");  
    }  
}
```

我们已经创建了目标对象DefaultStockService，然后使用ProxyFactory创建一个实现了StockService接口的代理。接着让我们看看怎样使用ProxyFactory的子类创建能接受通知的代理。要达到这个目的，可以看看代码清单6-64中简单的前置通知。

代码清单6-64 具有前置通知的简单方面

```
public class BeforeAspect {  
  
    @Before("execution(* com.apress.prospring2.ch06.services.*(..))")  
    public void simpleLog(JoinPoint jp) {  
        System.out.println("Before " + jp);  
    }  
}
```

这个方面没有什么特别之处。它在调用被通知方法之前简单地打印出一条消息。有了这个方面，现在我们将使用AspectJProxyFactory来创建一个接受通知的代理。代码清单6-65显示了创建被通知代理需要的代码。

代码清单6-65 使用AspectJProxyFactory

```
public class ProxyDemo4 {  
  
    public static void main(String[] args) {  
        DefaultStockService target = new DefaultStockService();  
        AspectJProxyFactory pf = new AspectJProxyFactory(target);  
        pf.addInterface(StockService.class);  
        pf.addAspect(BeforeAspect.class);  
  
        StockService stockService = (StockService) pf.getProxy();  
        stockService.getStockLevel("A");  
    }  
}
```

226 第6章 AOP 进阶

运行这个示例，它会显示与使用Spring中的

现在我们已经知道如何使用Spring创建一个被通知的代理了，还需要了解代理调用的意义，这非常重要。试想一个接收包围通知的对象，它为所有方法的调用创建和提交所有事务。如果我们取到被通知对象（也就是那个代理）并调用其方法，通知就会生效。不过当你在被通知对象内部调用被通知方法时，这个调用将不会被代理，并且通知也不会运行。因此，当我们在示例程序中使用StockService代理时，会看到只有对getStockLevel方法的调用被通知到了。而对getPredictedStockLevel方法的内部调用却没有。

避免这种情况的最好办法是不要在被通知类中锁住方法调用。某些情况下，链式调用会导致产生一些问题（我们将在第11章探讨一些你会碰到的问题）。不过，还有其他办法允许我们调用被通知对象并通过合适的代理管理它们。我们很不提倡这么做，但如果必须这样，可以使用AopContext.currentProxy()方法来获取代表this的代理。代码清单6-66显示了对DefaultStockService的修改并演示了如何使用此调用。

代码清单6-66 修改后的DefaultStockService

```
public class DefaultStockService implements StockService {

    public long getStockLevel(String sku) {
        try {
            Thread.sleep(2000L);
        } catch (InterruptedException ignored) {}
    }
    return ((StockService)AopContext.currentProxy()).
        getPredictedStockLevel(sku) / 2L;
}

public long getPredictedStockLevel(String sku) {
    return 6L * sku.hashCode();
}

public void applyDiscounts(Date cutoffDate, BigDecimal maximumDiscount) {
    // do some work
}

}
```

我们使用AopContext.currentProxy()代替this.getPredictedStockLevel(sku)来获取代理，并在代理上调用getPredictedStockLevel。如果现在我们试着运行一下示例程序，则会以失败告终。我们必须在ProxyFactory的配置中公开当前代理（如代码清单6-67所示）。

代码清单6-67 修改后的ProxyFactory配置

```
public class ProxyDemo4 {  
  
    public static void main(String[] args) {  
        DefaultStockService target = new DefaultStockService();  
        AspectJProxyFactory pf = new AspectJProxyFactory(target);  
        pf.addInterface(StockService.class);  
        pf.setExposeProxy(true);  
        pf.addAspect(BeforeAspect.class);  
  
        StockService stockService = (StockService) pf.getProxy();  
        stockService.getStockLevel("A");  
    }  
}
```

一切就绪。当我们运行这个程序，它显示出代码清单6-64中的前置通知在`getStockLevel`和`getPredictedStockLevel`上都运行了：

```
Before org.springframework.aop.aspectj.↗  
MethodInvocationProceedingJoinPoint: ↗  
execution(getStockLevel)  
Before org.springframework.aop.aspectj.↗  
MethodInvocationProceedingJoinPoint: ↗  
execution(getPredictedStockLevel)
```

虽然这是针对代理在链式调用被通知方法时所产生问题的解决方案，但我们强烈建议你对现有设计重新考量以避免这样的链式调用。除非已别无选择时再考虑使用`AopContext`并改变`ProxyFactory`的配置。这么做会将bean和Spring AOP紧紧耦合，让代码依赖于所使用的框架却正是Spring努力避免的啊！

6

6.6 AspectJ 集成

AOP为基于OOP的应用程序中的常见问题提供了强大的解决办法。使用Spring AOP时，我们可以利用精心选择的一部分AOP功能。一般情况下，这些功能足以让你解决程序中遇到的问题。不过，有时候你可能需要用到Spring AOP之外的AOP功能。这种情况下，你需要寻找提供更全面AOP支持的AOP实现，我们的首选是AspectJ。不仅如此，现在我们可以用Spring来配置AspectJ方面，因此AspectJ毫无疑问是Spring AOP的绝佳拍档。

AspectJ是一个功能全面的AOP实现。它用编译时的织入过程将方面引入到我们的代码中去。在AspectJ中，方面和切入点是类似Java的语法编写的，因此对于Java程序员来说上手比较容易。在这里我们不会花太多时间讲解AspectJ及其工作原理，因为这远远超出了本书的范围。只是简单讲讲AspectJ的几个例子，以及如何使用Spring配置它们。如果想更加深入的了解AspectJ，那么你一定要阅读Raminvas Ladded的著作*AspectJ in Action*（Manning，2003）一书。

6.6.1 创建第一个 AspectJ 方面

让我们从一个简单示例开始。首先创建一个方面并使用AspectJ编译器来将其织入。接下来，把这个方面配置成一个标准的Spring bean。我们能够这么做是因为每一个AspectJ方面都给出了一个方法——`aspectOf()`，可以用它来访问这个方面的实例。通过`aspectOf()`方法和Spring配置的一项特殊功能，我们可以让Spring为我们配置方面。这么做非常有好处。我们能在完全享用AspectJ的强大AOP功能集

的同时也不会丢掉Spring出色的依赖注入和配置能力。这也意味着应用程序不需要两套各自独立的配置方法。我们可以为所有Spring管理的bean和AspectJ的方面使用相同的Spring ApplicationContext方式。

在这个示例中，我们将使用AspectJ来通知com.apress.prospring2.ch06.services包中所有类的所有方法并分别在方法调用的前后打印一条消息。这些消息可以通过Spring变得可配置。代码清单6-68显示了这个StockServiceAspect方面（在目录com/apress/prospring2/ch06/aspectj下的名为StockServiceAspect.aj的文件中）。

代码清单6-68 StockServiceAspect方面

```
package com.apress.prospring2.ch06.aspectj;

public aspect StockServiceAspect {

    private String suffix;
    private String prefix;

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

    pointcut doServiceCall() :
        execution(* com.apress.prospring2.ch06.services.*(..));

    before() : doServiceCall() {
        System.out.println(this.prefix);
    }

    after() : doServiceCall() {
        System.out.println(this.suffix);
    }
}
```

大部分代码看上去很熟悉。我们实际上创建了一个名为StockServiceAspect的方面，就像一个普通Java类一样，我们给这个方面两个属性，suffix和prefix，当通知com.apress.prospring2.ch06.services包中所有类的所有方法时将用到它们。接着我们为一个单独的连接点，在这里就是服务方法的执行过程（AspectJ拥有大量的切入点，但它们超出了此示例的范围），定义了一个名为doServiceCall()的切入点。最后，我们定义了两个通知：一个在doServiceCall()切入点之前执行，另一个在其之后执行。前置通知打印一行包含前缀的文本，后置通知打印一行包含后缀的文本。代码清单6-69显示了这个方面在Spring中是怎样配置的。

代码清单6-69 配置一个AspectJ方面

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

  <bean id="userService"
        class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
        class="com.apress.prospring2.ch06.services.DefaultStockService"/>
  <bean class="com.apress.prospring2.ch06.aspectj.StockServiceAspect"
        factory-method="aspectOf">
    <property name="prefix" value="Before call"/>
    <property name="suffix" value="After call"/>
  </bean>

</beans>
```

可以看到, aspect bean的大部分配置跟标准bean的配置非常相似。唯一的区别是使用了<bean>标签的factory-method属性。factory-method属性允许那些采用传统工厂模式的类能无缝集成到Spring中去。例如, 如果有一个类Foo, 它有一个私有的构造方法和一个静态工厂方法——getInstance(), 使用factory-method可以让此类的bean成为Spring受控bean。每个单例AspectJ方面提供的aspectOf()方法允许我们访问这个方面的实例, 也因此允许Spring设置方面的属性值。不过, 请注意我们根本没有使用aop命名空间。由此来看, StockServiceAspect不是一个有效的Java源。为完成此示例, 代码清单6-70显示了使用stockService和userService bean的代码。

代码清单6-70 AspectJ的示例程序

```
public class AspectJDemo1 {
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "/META-INF/spring/aspectjdemo1-context.xml"
        );
        UserService userService = (UserService) ac.getBean("userService");
        userService.login("janm");

        StockService stockService = (StockService) ac.getBean("stockService");
        System.out.println(stockService.getStockLevel("ABC"));
    }
}
```

如果直接在IDE中运行程序, 它会失败。输出表明是因为缺少StockServiceAspect类。我们已预料到这个问题, 因为StockServiceAspect是一个方面, 而不是一个Java类。

```
...
Exception in thread "main" ➡
org.springframework.beans.factory.CannotLoadBeanClassException: ➡
Cannot find class [com.apress.prospring2.ch06.aspectj.StockServiceAspect] ➡
for bean with name 'com.apress.prospring2.ch06.aspectj.StockServiceAspect#0' ➡
defined in class path resource [META-INF/spring/aspectjdemo1-context.xml]; ➡
nested exception is java.lang.ClassNotFoundException: ➡
com.apress.prospring2.ch06.aspectj.StockServiceAspect
...
```

为了让示例程序正常运行, 我们必须使用AspectJ编译器。AspectJ编译器会将

StockServiceAspect 转化成一个Java类文件，并把方面的代码织入到被通知类中去。也就是说，AspectJ编译器将为DefaultStockService生成跟标准Java编译器不同的字节码，因为它会织入StockServiceAspect。除了编译时织入外，AspectJ编译器将根据StockServiceAspect.aj源文件生成可用的Java字节码文件。也许在编译后的StockServiceAspect类中最重要的方法是在Spring bean的定义中使用的public static Aspect aspectOf()方法。

6.6.2 编译示例程序

如果你熟悉AspectJ，你可以跳过这一节，否则请继续往下读，以了解如何在你的电脑上使用AspectJ编译器。第一步是从<http://www.eclipse.org/aspectj>获取AspectJ的发布包。下载安装包（.jar文件）并将AspectJ安装到\$ASPECTJ_HOME，一般是/usr/share/aspectj-1.5或C:\Program Files\aspectj1-5。当你安装好AspectJ编译器，在PATH环境变量上添加\$ASPECTJ_HOME/bin将会更加方便使用。现在在命令行中键入ajc -version，将会看到如下内容：

```
AspectJ Compiler 1.5.4 built on Thursday Dec 20, 2007 at 13:44:10 GMT
```

因为直接使用AspectJ编译器很复杂，我们使用Apache Ant来简化工作（更多有关Ant的信息请参见<http://ant.apache.org>）。AspectJ发布包包含了自定义的Ant任务。要安装这些自定义任务，将\$ASPECTJ_HOME/lib中的aspectjtools.jar复制到\$ANTHOME/lib下即可。

现在我们已经安装了AspectJ编译器并配置好Ant使其包含AspectJ自定义任务，让我们来看看这个构建文件（如代码清单6-71所示）。

代码清单6-71 Ant构建文件

```
<?xml version="1.0"?>
<project name="ch06" default="all" basedir="."
    xmlns:aspectj="antlib:org.aspectj">

    <property name="dir.src.main.java" value="./src/main/java"/>
    <property name="dir.src.main.resources" value="./src/main/resources"/>
    <property name="dir.module.main.build" value="./target/build-main"/>
    <property name="dir.lib" value="../../lib"/>
```



```
<property name="module.jar" value="ch06.jar"/>
<path id="module.classpath">
  <fileset dir="${dir.lib}" includes="**/*.jar"/>
  <fileset dir="${dir.lib}" includes="**/*.jar"/>
</path>

<target name="all">
  <aspectj:iajc>
    outjar="${module.jar}"
    sourceRootCopyFilter="**/*.java"
    source="1.5"
    target="1.5">
    <classpath refid="module.classpath"/>
    <sourceroots>
      <path location="${dir.src.main.java}">
        /com/apress/prospring2/ch06/services"/>
      <path location="${dir.src.main.java}">
        /com/apress/prospring2/ch06/aspectj"/>
      <path location="${dir.src.main.java}">
        /com/apress/prospring2/ch06/common"/>
      <path location="${dir.src.main.resources}">
        /com/apress/prospring2/ch06/common"/>
    </sourceroots>
    </aspectj:iajc>

    <java classname="com.apress.prospring2.ch06.aspectj.AspectJDemo1"
      fork="yes">
      <classpath>
        <path refid="module.classpath"/>
        <pathelement location="${module.jar}"/>
      </classpath>
    </java>

  </target>
</project>
```

当使用这个Ant脚本构建示例程序时，AspectJ编译器会织入方面并创建StockServiceAspect。使用ant运行代码清单6-70的示例程序，会打印出如下输出：

```
...
[java] DEBUG [main] CachedIntrospectionResults.<init>(265) | ➡
    Found bean property 'suffix' of type [java.lang.String]

[java] DEBUG [main] AbstractBeanFactory.getBean(197) | ➡
    Returning cached instance of singleton bean 'userService'

userService.login("janm")
[java] Before call
[java] After call
[java] DEBUG [main] AbstractBeanFactory.getBean(197) | ➡
    Returning cached instance of singleton bean 'stockService'

stockService.getStockLevel("ABC")
[java] Before call
stockService.getPredictedStockLevel("ABC")
[java] Before call
[java] After call
[java] After call
[java] 193734
```

以上输出清楚地表明这个方面起作用了！我们成功地通知了DefaultStockService和DefaultUserService方法并设置了方面的prefix和suffix属性的值。因为AspectJ编译器执行的是编译时织入。我们不用担心任何代理问题，通知DefaultStockService.getStockLevel中对getPredictedStockLevel的调用并未借助任何AopContext的魔力。

6.6.3 AspectJ 方面的作用域

默认情况下，AspectJ的方面都是单例的，一个类加载器对应一个单独的实例。如果我们需要根据切入点使用不同方面的实例，则编写和配置方面都是不同的。代码清单6-72显示了一个per this方面。“per this”的意思是AspectJ为每一个匹配的切入点创建一个新的方面的示例。

代码清单6-72 有状态的（per this）方面

```
package com.apress.prospring2.ch06.aspectj;

public aspect ThisCountingAspect perthis(doServiceCall()) {
    private int count;

    pointcut doServiceCall() :
        execution(* com.apress.prospring2.ch06.services.*(..));

    before() : doServiceCall() {
        this.count++;
        System.out.println("Before call");
    }

    after(Object target) : doServiceCall() && this(target) {
        System.out.println(target + " executed " + this.count + " times");
    }
}
```

唯一的问题是我们不能在Spring中配置方面。如果要在一般的原型代码中对其进行配置，则只能设置该方面的其中一个实例的属性，但这个实例并不一定就是连接点中用到的那个。如果我们需要一个区域性的AspectJ方面，就不能使用Spring来配置其属性。

6.7 加载时织入

加载时织入是当ClassLoader把被通知类加载到JVM时织入方面的过程。AspectJ通过一个Java代理提供加载时织入的支持。我们需要指定代理的JAR文件（细节请参看JVM的-javaagent命令行参数说明）。然而代理也有一些限制，显而易见的一个是代理只能工作在1.5或更高版本的JVM上，第二个限制是JVM在其整个范围内加载和使用代理。在较小的应用程序中这可能不是什么大问题，但当我们需要在一个虚拟机上部署多个应用时，我们可能需要更细粒度地对加载时织入过程进行控制。

Spring的加载时织入支持只做一件事：它允许我们为每一个ClassLoader控制加载时织入。这对在一个servlet容器或一个应用服务器上开发一个Web应用程序很有帮助。我们可以为每一个Web应用程序配置一个不同的加载时织入，并保持容器本身的类免受感染。另外，如果使用Spring加载时织入支持，我们可能也就不需要修改应用服务器或者servlet容器的配置了。

6.7.1 第一个加载时织入示例

我们将从一个简单的方面开始，它跟踪对所有服务方法的调用。这个方面很简单，其源代码如代码清单6-73所示。

代码清单6-73 一个加载时织入的方面

```
@Aspect
public class AuditAspect {

    @After(value =
        "execution(* com.apress.prospring2.ch06.services.*(..)) && " +
        "this(t)",
        argNames = "jp,t")
    public void audit(JoinPoint jp, Object t) {
        System.out.println("After call to " + t + " (" + jp + ")");
    }
}
```

现在有了这个方面，我们用它来通知 userService 和 stockService bean。使用 spring-agent.jar 作为 JVM 代理来替代 <aop:aspectj-autoproxy/>，并使用上下文命名空间来初始化加载时织入。除了 ApplicationContext 的 XML 配置文件之外，还需要创建 META-INF/aop.xml 文件。这个 aop.xml 文件是一个标准的 AspectJ 组件。它告诉 AspectJ 织入器在加载时织入哪些类。代码清单 6-74 中为 aop.xml 文件的内容。

代码清单6-74 META-INF/aop.xml文件

```
<!DOCTYPE aspectj PUBLIC
    "-//AspectJ//DTD//EN"
    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
    <weaver>
        <include within="com.apress.prospring2.ch06.services.*"/>
    </weaver>
    <aspects>
        <aspect name="com.apress.prospring2.ch06.ltw.AuditAspect"/>
    </aspects>
</aspectj>
```

加粗的代码告诉 AspectJ 织入器将 AuditAspect 织入到 com.apress.prospring2.ch06.services 包下的所有类中。为完成这个示例，代码清单 6-75 显示了带有 <context:load-time-weaver> 标签的 Application-Context 的 XML 配置文件。<context:load-time-weaver> 标签只有一个属性，aspectj-weaving。我们可以把它设成“on”来打开加载时织入，设成“off”来关闭加载时织入，设成“autodetect”在至少有一个 META-INF/aop.xml 文件时开启加载时织入。如果省略了 aspectj-weaving 属性，那么 Spring 就将其视为“autodetect”处理。

代码清单6-75 ApplicationContext的XML配置文件

234 第6章 AOP 进阶

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

  <bean id="userService"
        class="com.apress.prospring2.ch06.services.DefaultUserService"/>
  <bean id="stockService"
        class="com.apress.prospring2.ch06.services.DefaultStockService"/>

  <context:load-time-weaver />

</beans>
```

在运行这个程序看它能否进行加载时织入之前，我们需要添加`-javaagent`这个JVM参数。在这里，参数值为`-javaagent:../lib/org/springframework/spring/spring-agent.jar`。如果没有设置这个代理，程序会运行失败，并且抛出一个`IllegalStateException`异常。

```
...
Caused by: java.lang.IllegalStateException: ➤
ClassLoader [sun.misc.Launcher$AppClassLoader] ➤
does NOT provide an 'addTransformer(ClassFileTransformer)' method. ➤
Specify a custom LoadTimeWeaver or start your Java virtual machine with ➤
Spring's agent: -javaagent:spring-agent.jar
```

错误消息在我们预料之中，我们要将`spring-agent.jar`库指定为JVM代理。这么做以后，运行程序输出如下：

```
userService.login("janm")
After call to com.apress.prospring2.ch06.services.➤
  DefaultUserService@4cb44131 ➤
  (execution(void com.apress.prospring2.ch06.services.➤
    DefaultUserService.login(String)))

stockService.getStockLevel("ABC")
After call to com.apress.prospring2.ch06.services.➤
  DefaultStockService@197a64f2 ➤
  (execution(long com.apress.prospring2.ch06.services.➤
    DefaultStockService.getPredictedStockLevel(String)))
  DefaultStockService.getPredictedStockLevel("ABC")

After call to com.apress.prospring2.ch06.services.➤
  DefaultStockService@197a64f2 ➤
  (execution(long com.apress.prospring2.ch06.services.➤
    DefaultStockService.getStockLevel(String)))
```

193734

不仅程序以加载时织入的方式工作了，而且`DefaultStockService.getStockLevel`中对`DefaultStockService.getPredictedStockLevel`的调用也得到了通知，这表明代理是没有问题的，内存中`DefaultStockService`类的字节码跟它存储在硬盘上那一份并不一样，AspectJ织入器在程序的ClassLoader加载类之前就完成了工作。加载时织入不仅解决了代理问题，还去除了由代理导

致的性能下降。由于已经不存在代理，代码是跟被硬编码到每一个匹配的连接点上的通知的代码一起编译运行的。

6.7.2 LoadTimeWeaver的查找策略

Spring在JVM代理库spring-agent.jar中使用InstrumentationSavingAgent来保存由JVM提供的Instrumentation接口的当前实例。DefaultContextLoadTimeWeaver将会自动检测跟应用环境匹配得最好的LoadTimeWeaver实例。表6-2显示了不同环境下的LoadTimeWeaver实现。

表6-2 LoadTimeWeaver的实现

LoadTimeWeaver	环 境
InstrumentationLoadTimeWeaver	JVM以Spring的InstrumentationSavingAgent开始(使用-javaagent:\$LIB/spring-agent.jar)
WebLogicLoadTimeWeaver	BEA WebLogic 10或更高版本应用服务器上运行的LoadTimeWeaver实现类
GlassFishLoadTimeWeaver	工作在GlassFish V2应用服务器上
OC4JLoadTimeWeaver	Oracle 10.1.3.1或更高版本应用服务器上运行的LoadTimeWeaver实现类
ReflectiveLoadTimeWeaver	跟TomcatInstrumentableClassLoader一起使用，在Tomcat servlet容器和默认的LoadTimeWeaver实现中提供加载时织入
SimpleLoadTimeWeaver	只用于测试的LoadTimeWeaver实现(这里“只用于测试”的意思是它在新创建的ClassLoader上进行必要的织入转换)

无论使用哪种策略，意识到加载时织入使用的是AspectJ而不是@AspectJ是很重要的，你可能在看过代码清单6-74中代码后会这么想。这意味着我们不能使用bean()切入点的@AspectJ支持。

6.8 AOP 实践

当有人提到AOP时，我们第一个会想到日志，然后就是事务管理。不过，这些都只是AOP的特殊应用。以我们的开发经验，我们把AOP用于健康与性能监测、调用统计、缓存和错误回复。更高级的AOP用法还包括编译时结构标准检查。例如，我们可以编写一个方面来加强对只能调用某些类的某些方法的限制。然而更高级的案例都几乎与Spring AOP没有关系，因此我们还是来演示性能和健康监测。我们也可以引用第22章中Spring缓存模块的例子，它使用AOP实现声明性的缓存支持。

性能与健康监测

能够对部署在生产环境下的应用程序进行监控是很关键的。当我们使用日志时，在日志记录中寻找一个特定的问题通常很困难。如果能跟踪程序的性能并迅速找到任何明显的性能下降将会方便得多。同样地，记录程序运行时产生的异常也会很有用处。除了记录异常类型(和说明信息)之外，记录下所有可能产生异常的参数的值也很重要。为了帮助我们找到程序中大多数最令人头疼的那些问题，我们也会以异常类型和参数值将异常报告分组。

在实现这个方面之前，让我们来看看程序的其他组件。图6-9显示了程序组件的UML类图。

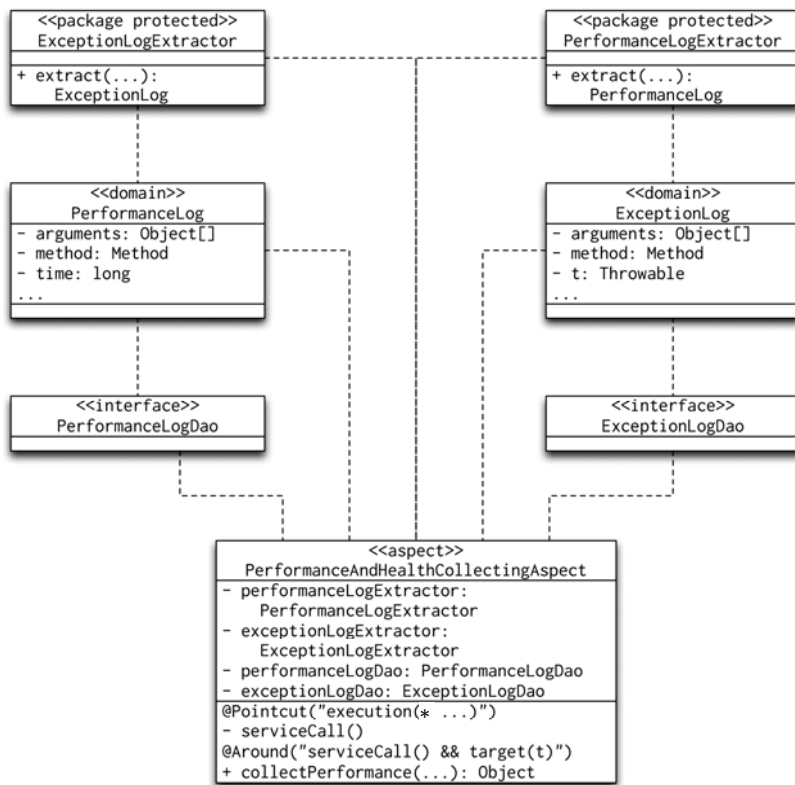


图6-9 主要组件的类图

现在，让我们看看PerformanceAndHealthCollectingAspect的实现，这个方面只有一个包围通知。这个通知收集有关方法执行时间和可能异常的统计数据。代码清单6-76显示了这个PerformanceAnd- HealthCollectingAspect。

代码清单6-76 PerformanceAndHealthCollectingAspect

```
@Aspect
public class PerformanceAndHealthCollectingAspect {

    private PerformanceLogDao performanceLogDao;
```



```
private ExceptionLogDao exceptionLogDao;
private PerformanceLogExtractor performanceLogExtractor =
    new PerformanceLogExtractor();
private ExceptionLogExtractor exceptionLogExtractor =
    new ExceptionLogExtractor();

@Pointcut("execution(* com.apress.prospring2.ch06.services.*.*(..))")
private void serviceCall() { }

@Around(value = "serviceCall() && target(target)",
    argNames = "pjp, target")
public Object collectPerformance(ProceedingJoinPoint pjp, Object target)
    throws Throwable {
    Throwable exception = null;
    Object ret = null;

    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start();
    try {
        ret = pjp.proceed();
    } catch (Throwable t) {
        exception = t;
    }
    stopWatch.stop();

    if (exception == null) {
        this.performanceLogDao.insert(
            this.performanceLogExtractor.extract(pjp, target),
            stopWatch.getLastTaskTimeMillis())
    } else {
        this.exceptionLogDao.insert(
            this.exceptionLogExtractor.extract(pjp, target),
            exception
        );
    }

    if (exception != null) throw exception;
    return ret;
}

public void setPerformanceLogDao(PerformanceLogDao performanceLogDao) {
    this.performanceLogDao = performanceLogDao;
}

public void setExceptionLogDao(ExceptionLogDao exceptionLogDao) {
    this.exceptionLogDao = exceptionLogDao;
}
}
```

可以看到我们记录了方法的性能和可能的异常。我们将使用这些 `PerformanceMonitoringAspect` 中的信息。根据它的名字知道，这个方面会监测程序的性能。它同时需要前置和后置通知。前置通知将记录方法调用的开始，后置通知将匹配并移除对应的开始记录。这么做的理由是我们要能在运行时找出一个比平时执行时间长得多的特定方法。如果我们使用包围通知实现过这个健康监测器，我们的代码则只能当其完成后才知道某方法执行较长时间。代码清单6-77给出了实现这个方面的雏形。

代码清单6-77 `PerformanceMonitoringAspect` 实现的雏形

```
@Aspect
public class PerformanceMonitoringAspect {

    @Pointcut("execution(* com.apress.prospring2.ch06.services.*(..))")
    private void serviceCall() { }

    @Before(value = "serviceCall() && target(target)",
            argNames = "jp, target")
    public void logStartCall(JoinPoint jp, Object target) {
        // log start call
    }

    @After(value = "serviceCall() && target(target)",
            argNames = "jp, target")
    public void logEndCall(JoinPoint jp, Object target) {
        // log end call
    }
}
```

日志构架必须为每个开始的调用插入一条记录，并在方法执行完毕后将记录移除。最后一个组件是 `PerformanceMonitor` 类，它周期性地扫描调用日志中所有的内容，检查 `PerformanceAndHealthMonitoringAspect` 中收集的统计数据。如果 `PerformanceMonitor` 找到一个比平时耗时长得多的调用，它就会向程序管理员发出警告。

6.9 小结

在本章中，我们结束了对 AOP 的讨论，探讨了切入点的高级选项，以及如何使用引入来扩展某个对象实现的接口集合。本章花大量篇幅讲解如何使用 Spring 框架服务来声明性地配置 AOP，这样就避免了将创建 AOP 的逻辑直接写进代码中。还花了些时间介绍 Spring 如何与 AspectJ 集成，以达到在不放弃 Spring 灵活性的同时，就能使用 AspectJ 的额外能力的目的。接着，我们讨论了如何使用带有加载时织入功能的 AspectJ 来最小化 Spring AOP 的代理带给应用程序的运行开销。最后，我们讨论了如何在实际程序中使用 AOP 解决具体程序的问题。

尽管我们已经讨论了怎样在 Spring 应用程序里使用 AOP，不过也只接触到 AspectJ 所有功能的皮毛。如果你对 AspectJ 的更多细节感兴趣，我们推荐两本绝佳读物。第一本是由 Adrian Colyer、Andy Clement、George Harley 和 Matthew Webster 合著的 *Eclipse AspectJ* (Addison-Wesley, 2005)，它为 AspectJ 语言提供了全面的介绍和参考资料；第二本是由 Ramnivas Laddad 所著的 *AspectJ in Action* (Manning, 2003)，它着重讲解 AspectJ 的语法并覆盖大量 AOP 的常规主题。

第 12 章

基于Spring的任务调度

12

大多数应用程序逻辑是用来反馈某种形式的用户行为的，例如点击一个按钮或提交一个表单。然而，在很多应用程序中存在无需与用户交互来调用的某种处理，通常是在固定时间间隔运行一次。例如，也许我们有个进程每小时清理一次临时文件，或者每天午夜从数据库导出数据并发送到一个外部系统。多数重要的应用都要求某种形式的调度支持，该调度若不是和应用的业务逻辑直接相关就是为系统做一些辅助工作。

如果你正在为应用程序建立调度任务，那么创建一个任务让它每小时或者一天运行一次是相当简单的。但是该任务需要在每周一、周三和周五的下午三点运行呢？编写代码可能就有些困难了，这也使得选择一个现有的任务调度解决方案比创建自己的调度框架更合理。

如果从编程的角度来讨论任务调度，我们倾向于讨论3种不同的概念。一个任务是一个需要被调度以指定时间间隔运行的工作单元。一个触发器是一个引发任务运行的条件，可能是一个固定的时间间隔或者是既定片段的数据。一个调度计划是一组触发器的集合，它管理任务的整个时限。一般通过实现某个接口或者扩展某个特定基类来封装一个任务。我们可以使用任务调度框架支持的任何方式定义触发器。一些框架可能只支持简单的基于时间间隔的触发器，但是其他一些，比如Quartz，提供了更灵活的触发器模式。通常情况下，在调度计划中一个任务只有一个触发器，因此术语“调度”和“触发器”经常是交换使用。

Spring对任务调度的支持有两种不同的形式：基于JDK Timer和基于Quartz。基于JDK Timer方式的任务调度为所有JVM 1.3及后续版本提供了任务调度能力，并且它没有Spring以外的依赖。基于Timer的任务调度比较简单，在定义任务调度时也只能提供有限的灵活性。然而，Timer支持建立在Java标准内并且不需要外部的依赖库，当你受限于程序大小或者企业策略时可以从其中受益。基于Quartz的任务调度具有更好的灵活性，允许我们定义更加接近现实世界的触发器，比如先前的每周一、周三和周五的下午三点运行任务的例子。

本章将讨论Spring包含的上述两种任务调度解决方案。本章专门讨论了3个核心主题：使用JDK Timer进行任务调度，基于Quartz的任务调度和任务调度时的需考虑因素。

我们将从关于Spring对基于JDK Timer的任务调度的支持的讨论开始。此小节介绍了基于Timer的任务调度可选的不同触发器类型，并说明了如何在不创建额外Java代码的情况下调度所有的复杂逻辑。

然后，我们将开始了解基于Quartz的复杂任务调度以及它与Spring集成的方式。我们专门测试了Quartz对于Cron表达式的支持是不是允许使用精确格式配置高度复杂的任务调度。在使用JDK Timer时，我们可以理解如何无需封装而调度所有的逻辑。

最后，我们将讨论对一个任务调度的实现以及为调度执行创建逻辑使用的模式进行选择时所需要考虑的各种因素。

12.1 使用JDK Timer调度任务

Spring支持的最基本的任务调度是基于JDK `java.util.Timer`类的。当使用`Timer`进行任务调度时，我们只能使用简单的基于时间间隔的触发器定义，这使得基于`Timer`的任务调度只适合那些需要在既定未来某个时间执行或者以固定周期运行的任务。

12.1.1 Timer触发器类型

基于`Timer`的任务调度给我们提供了以下3种触发器类型。

- ❑ 一次性（one-off）。当使用一个一次性触发器时，任务执行是在未来某个时间点调度，使用从某个时间开始的毫秒数来定义该时间点。任务执行以后，它就不会再次被调度使用。我们发现一次性触发器对于那些只需完成一次的任务是很适用的，因为你自己可能忘记去做这个任务。例如，如果一个Web应用程序被调度在下一周进行维护，那么我们可以调度一个任务，在维护开始时，切换到一个维护页面。
- ❑ 重复和固定延迟（fixed-delay）。当使用一个固定延迟触发器时，我们可以像使用一次性触发器那样调度此任务第一个执行，但是在一个给定时间间隔内它会被重新调度执行。当我们使用固定延迟时，时间间隔是相对于此任务上一次执行的时间。这意味着两次连续执行时间间隔几乎完全一样，即使执行可能“晚”于原先的调度。使用此种类型的触发器，你指定的时间间隔是连续两次执行的间隔。当希望尽可能让时间间隔是一个常数时，使用这种触发器。
- ❑ 重复和定时（fixed-rate）。定时触发器的功能和固定延迟触发器很相似，但是下一次执行时间总是基于最初被调度的执行时间。这意味着如果单个执行被延迟了，那后续执行也不会被延迟。使用此种类型的触发器，你指定的时间间隔并不是连续两次执行时间的实际时间间隔。在实际执行的时间点很重要而实际执行时间间隔不重要时，应该使用此种触发器。

我们可能发现，要形象化地显示固定延迟触发器和定时触发器之间的不同是很困难的。为了清晰地展示它们之间的区别，我们需要创建一个在执行中引起足够长延迟的示例，但是这相当困难。

考虑一个在13:00开始执行的任务，两次运行之间的指定间隔为30 min。任务一直运行良好，直到16:30，此时系统的负载过重，导致执行垃圾收集，而这导致实际运行完成时间迟于预计时间——任务运行时已经16:31了。现在如果使用固定延迟来调度任务，那么重要的是时间间隔，也就是说，我们希望两次实际执行的时间间隔是30 min，因此下次任务调度执行的时间是17:01而不是17:00。如果我们使用定时调度，时间间隔只定义了预期的调度，也就是说，我们期望以程序开始执行的时间作为基准点，每隔30 min运行一次，而不是基于上次执行的时间，所以任务调度在17:00执行。

两个触发器类型都有各自的用途。一般情况下，定时触发器用于下述情况：两次执行间的时间间隔规律的情况或者你想避免某次执行被延迟太长时间时，会导致两次执行发生的时间点太近的情况。使用定时触发器可以得到以上效果。一般来说，你在实时敏感系统中使用定时触发器，例如必须每小时整点执行的任务。

12.1.2 创建一个简单任务

402 第12章 基于Spring的任务调度

为了创建是一个使用Timer类的任务，你可以简单扩展TimerTask类并实现run()方法执行你的任务逻辑。代码清单12-1展示了一个简单TimerTask实现，它打印“Hello World!”到标准输出(stdout)。

代码清单12-1 创建基本的TimerTask类

```
package com.apress.prospring2.ch12.timer;

import java.util.TimerTask;

public class HelloWorldTask extends TimerTask {

    public void run() {
        System.out.println("Hello World!");
    }
}
```

这里你可以看到在run()方法中，我们简单地把“Hello World!”消息输出到标准输出。每次任务执行时，Timer就会调用TimerTask的run()方法。我们能够为此任务创建的最简单的触发器是在1s后运行此任务的一次性触发器。见代码清单12-2。

代码清单12-2 在HelloWorldTask类中使用一次性触发器

```
package com.apress.prospring2.ch12.timer;

import java.util.Timer;

public class OneOffScheduling {

    public static void main(String[] args) {
        Timer t = new Timer();
        t.schedule(new HelloWorldTask(), 1000);
    }
}
```

当使用JDK Timer类为一个指定触发器调度任务时，我们必须先创建一个Timer类的实例，然后使用schedule()或者scheduleAtFixedRate()方法。在代码清单12-2中，我们使用了schedule()方法来调度一个HelloWorldTask实例，使它在程序开始的1000ms延迟后运行。如果运行上述示例，1s延迟以后，我们会得到如下消息：

```
Hello World!
```

这种一次性的触发器一般没有什么用处，想想看你调度一个一次性任务会有多频繁，使它在程序启动后的任意时间段内运行。因此在创建一次性触发器时，你可以指定一个绝对时间。因此我们想创建一个任务，它将在一个重要生日前7天提示我们，我们可以用我们自己的调用来替换Timer.schedule()方法，如下所示：

```
Calendar cal = Calendar.getInstance();
cal.set(2008, Calendar.NOVEMBER, 30);
t.schedule(new HelloWorldTask(), cal.getTime());
```

在这个例子中，可以看到我们为日期2008年11月30日创建了一个Calendar实例，然后使用Calendar实例，我们调度HelloWorkdTask的运行时间。这显然比第一个例子更实用，因为无论程序什么时候运行，任务总是被调度在相同时间内运行。该方式的唯一缺陷是，我们将不会为2009年或

2010年的生日提示,除非我们显式地加入更多的触发器。不过我们使用一个可以重复执行的触发器避免此问题。

对于两种重复执行触发器,固定延迟和定时触发器,都使用同样的方式进行配置:你指定开始时间点,使用一个绝对日期或通过`schedule()`方法开始后的相对毫秒数进行定义,然后你以毫秒为单位指定一个时间间隔来控制后续执行的发生的时间。要记住“时间间隔”在你使用固定延迟和定时触发器时表示不同的含义。

我们可以调度`HelloWorldTask`在程序开始后延迟1 s运行,接着每3 s运行一次,代码参见代码清单12-3。

代码清单12-3 调度一个重复性任务

```
package com.apress.prospring2.ch12.timer;

import java.util.Timer;

public class FixedDelayScheduling {

    public static void main(String[] args) throws Exception{
        Timer t = new Timer();
        t.schedule(new HelloWorldTask(), 1000, 3000);
    }
}
```

如果我们运行程序,将看到第一条“Hello World!”消息在1 s后显示,而后每3 s显示一条“Hello World!”消息。要使用定时触发器来调度该任务,只需用`Timer.scheduleAtFixedRate()`方法替换掉`Timer.schedule()`方法即可,如代码清单12-4所示。

代码清单12-4 使用定时触发器来调度任务

```
package com.apress.prospring2.ch12.timer;

import java.util.Timer;

public class FixedRateScheduling {

    public static void main(String[] args) throws Exception {
        Timer t = new Timer();
        t.scheduleAtFixedRate(new HelloWorldTask(), 1000, 1000);
    }
}
```

和一次性触发器一样,可以让固定延迟和定时触发器使用一个固定时间启动。使用该方法,我们可以创建一个用于生日提示的触发器示例,它在一个指定的日期被触发然后每年都会运行一次。如代码清单12-5所示。

代码清单12-5 调度生日提示程序

```
package com.apress.prospring2.ch12.timer;

import java.util.Calendar;
```


404 第12章 基于Spring的任务调度

```
import java.util.Timer;

public class SimpleBirthdayReminderScheduling {

    private static final long MILLIS_IN_YEAR = 1000 * 60 * 60 * 24 * 365;

    public static void main(String[] args) {
        Timer t = new Timer();

        Calendar cal = Calendar.getInstance();
        cal.set(2008, Calendar.NOVEMBER, 30);
        t.schedule(new HelloWorldTask(), cal.getTime());

        t.scheduleAtFixedRate(new HelloWorldTask(), cal.getTime(),
                               MILLIS_IN_YEAR);
    }
}
```

在这个例子中，我们可以看到我们先计算了代表一年的毫秒数，然后创建一个Calendar实例并用它定义开始时间为11月30日而时间间隔为一年。现在，只要该程序一直运行，每年的11月30日“Hello World!”消息就会打印到标准输出。记住它并不是功能完善的示例，所以它没有真实的通知机制，并且每次我们希望增加一个新的生日提示时都需要修改代码。在下一小节，我们将使用Spring的JDK Timer支持类来创建一个更稳健的生日提示程序。

12.1.3 Spring对JDK Timer调度的支持

正如前一小节所看到的，使用JDK Timer和TimerTask类来创建和调度任务是很容易的。但是，我们在前一个例子中使用的方法有一些问题。首先，我们在程序中使用TimerTask实例而不是使用Spring。对于HelloWorldTask，这是可以接受的，因为HelloWorldTask无需配置该任务。但是，许多任务需要一些配置数据，因此我们应该使用Spring来管理它们，使程序易于配置。第二，触发器信息是硬编码到程序中的，这使得对任务被触发的时间上做任何修改都需要修改代码重新编译。最后，调度新任务或移除任务也需要修改程序代码，而在理想情况下我们应该可以在外部对它进行配置。使用Spring的Timer支持类，我们可以将所有的任务和触发器配置以及Timer创建的控制委托给Spring来处理，这样我们就可以在外部定义任务及其触发器。

Spring对Timer的支持的核心是由ScheduledTimerTask和TimerFactoryBean类组成的。ScheduledTimerTask类是对TimerTask的包装器实现，这样你就可以为这个任务定义触发器信息。使用TimerFactoryBean类，你可以让Spring使用配置创建触发器，并为一组指定的ScheduledTimerTask bean自动创建Timer实例。

1. 使用ScheduledTimerTask和TimerFactoryBean类

在我们深入讨论新的改善版生日提示程序之前，我们应该首先了解一下ScheduledTimerTask和TimerFactoryBean的工作基础。你需要为每一个待调度的任务配置任务类和一个包含触发器细节的ScheduledTimerTask实例。若你想为同一个任务创建多个触发器，你可以在多个ScheduledTimerTask实例间共享一个TimerTask实例。一旦你配置好了这些组件，只需简单配置一个TimerFactoryBean类并指定ScheduledTimerTask bean的列表。接着Spring创建一个Timer实例，并使用它来调度已被ScheduledTimerTask类定义的所有任务。

这听起来很复杂，但是实际上并不是这样。代码清单12-6展示了一个简单配置，它调度

HelloWorld- Task使之第一次运行前有1 s延迟而后每3 s运行一次。

代码清单12-6 使用TimerFactoryBean配置任务调度

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="job" class="com.apress.prospring2.ch12.timer.HelloWorldTask"/>

    <bean id="timerTask"
        class="org.springframework.scheduling.timer.ScheduledTimerTask">
        <property name="delay" value="1000" />
        <property name="period" value="3000" />
        <property name="timerTask" ref="job" />
    </bean>

    <bean id="timerFactory"
        class="org.springframework.scheduling.timer.TimerFactoryBean">
        <property name="scheduledTimerTasks">
            <list>
                <ref local="timerTask"/>
            </list>
        </property>
    </bean>
</beans>
```

这里你可以看到我们已经配置了一个bean，这是一个HelloWorldTask类型的任务，接着我们使用这个bean配置了一个ScheduledTimerTask类型的bean，设置启动延迟为1 000 ms，后续延迟为3 000 ms。配置的最后部分是timerFactory bean，它接受一个ScheduledTimerTask类型的bean列表。在此种情况下，我们只有一个任务要调度，由timerTask bean表示。在使用ScheduledTimerTask指定触发器信息时，你可以只提供毫秒数作为参数，这表示在程序启动以后有一段时间的延迟而不使用原有的启动时间。我们将在下一节创建生日提示程序时介绍这种方法。

当所有的调度计划和任务定义信息都包括在配置文件中，我们的示例程序几乎不需要做什么。实际上，我们所需要的就是加载ApplicationContext，Spring会自动创建Timer类并依据配置文件调度HelloWorldTask。代码如代码清单12-7所示。

代码清单12-7 TimerFactoryBeanExample类

```
package com.apress.prospring2.ch12.timer;

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

public class TimerFactoryBeanExample {
    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "ch12/src/conf/timer-context.xml");
        System.in.read();
    }
}
```

406 第12章 基于Spring的任务调度

如果运行此应用程序，你将看到消息“Hello World!”在程序开始的1 s延迟后，每3 s输出一次到标准输出。正如你从示例中看到的，在你的代码外配置任务调度是非常简单的。使用此方法，修改任务调度计划或者移除现存任务增加新的调度任务就简单多了。

2. 一个更实用的生日提示程序

在本小节中，我们使用Spring的Timer支持来创建一个更复杂的生日提醒程序。使用这个例子，我们希望能够调度多个提示任务，每个都有特定的配置来表明是为谁的生日进行提示。我们也希望能无需修改程序代码就可增加和移除提示。

开始之前，我们需要创建一个任务来执行实际的提示。因为我们将使用Spring创建这些任务，可以使用依赖注入的方式提供所有的配置数据。代码清单12-8展示了BirthdayReminderTask类。

代码清单12-8 BirthdayReminderTask类

```
package com.apress.prospring2.ch12.timer.bday;

import java.util.TimerTask;

public class BirthdayReminderTask extends TimerTask {

    private String who;

    public void setWho(String who) {
        this.who = who;
    }

    public void run() {
        System.out.println("Don't forget it is " + who
            + "'s birthday is 7 days");
    }
}
```

注意，我们在任务上定义了一个属性who，我们可以指定我们正在提示的是谁的生日。在真实的生日提示程序中，提示无疑应该使用e-mail或其他相似媒介通知。但是现在，则不得不将具有提示信息的内容发送到标准输出。

完成此任务后，我们就可以进入配置阶段了。但是，就像我们先前指出的那样，你使用ScheduledTimerTask时不可以通过日期来指定一个调度任务的起始时间。这对我们的应用程序是个问题，因为我们不可以把程序启动后的一个相对时间延迟作为触发器的起始时间。幸运的是，我们可以扩展ScheduledTimerTask类并重写getDelay()方法，TimerFactoryBean使用getDelay()方法判断应该分配多少延迟时间给触发器，这种做法很容易克服前面的问题。同时，我们也可以重写getPeriod()方法来返回代表一年时间的毫秒数，这样你就无需将该参数（毫秒数）增加到配置文件中。代码清单12-9展示了我们定义的ScheduledTimerTask和BirthdayScheduledTask类。

代码清单12-9 自定义BirthdayScheduledTask类

12.1 使用 JDK TTimerT 调度任务 407

12

```
package com.apress.prospring2.ch12.timer.bday;

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

import org.springframework.scheduling.timer.ScheduledTimerTask;

public class BirthdayScheduledTask extends ScheduledTimerTask {

    private static final long MILLIS_IN_YEAR = 1000 * 60 * 60 * 24 * 365;

    private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

    private Date startDate;

    public void setDate(String date) throws ParseException {
        startDate = dateFormat.parse(date);
    }

    public long getDelay() {
        Calendar now = Calendar.getInstance();
        Calendar then = Calendar.getInstance();
        then.setTime(startDate);

        return (then.getTimeInMillis() - now.getTimeInMillis());
    }

    public long getPeriod() {
        return MILLIS_IN_YEAR;
    }
}
```

在示例中，你可以看到我们为`BirthdayScheduledTask`定义了一个新属性`date`，它可以让我们指定任务的起始时间而不是一个延迟时间段。此属性是`String`类型，因为我们使用一个`SimpleDateFormat`的实例，根据格式`yyyy-MM-dd`来解析属性值为日期格式，如`2008-11-30`。你也看到我们重写了`getPeriod()`方法，`TimerFactoryBean`类使用它来配置触发器运行的时间间隔，此方法返回代表一年时间的毫秒数。也需注意我们对`getDelay()`方法的重写，使用`Calendar`类来计算当前时间和给定起始时间之间的毫秒数。接着该值作为程序开始时的延迟被返回。现在我们已经完成了示例程序的配置，如代码清单12-10所示。

代码清单12-10 生日提示程序的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

408 第12章 基于Spring的任务调度

```
<bean id="mum"
      class="com.apress.prospring2.ch12.timer.bday.BirthdayScheduledTask">
  <property name="date" value="2008-11-30" />
  <property name="fixedRate" value="true" />

  <property name="timerTask">
    <bean class="com.apress.prospring2.ch12.timer.bday.BirthdayReminderTask">
      <property name="who" value="Mum">
    </bean>
  </property>
</bean>

<bean id="timerFactory"
      class="org.springframework.scheduling.timer.TimerFactoryBean">
  <property name="scheduledTimerTasks">
    <list>
      <ref local="mum" />
    </list>
  </property>
</bean>
</beans>
```

上述代码对你来说应该相当熟悉。注意，我们使用我们自己的`BirthdayScheduledTask`类替换了`BirthdayTimerTask`类，只指定一个日期而没有指定延迟和时间间隔。我们用重写的`getDelay()`和`getPeriod()`方法来为`TimerFactoryBean`类提供延迟和时间间隔值。除此之外，注意我们设置`BirthdayScheduledTask` bean的`fixedRate`属性为`true`。该属性从`ScheduledTimerTask`继承，`TimerFactoryBean`使用它来判断是创建一个定时触发器还是创建固定延迟触发器。

3. 调度任意任务

当你进行任务调度时，常常需要调度现有逻辑的执行。在这种情况下，你可能不希望麻烦地创建一个仅用于包装逻辑的`TimerTask`类。幸运的是，你没有必要这样做。你能够使用`MethodInvokingTimerTaskFactoryBean`类调度任意给定的bean的任意方法或指定类的静态方法。如果你的逻辑需要，你甚至可以为方法提供参数。

代码清单12-11中的`FooBean`类是此种情况的示例。

代码清单12-11 FooBean类

```
package com.apress.prospring2.ch12.timer;

public class FooBean {

    public void someJob(String message) {
        System.out.println(message);
    }
}
```

如果我们希望调度`someJob()`方法，为它提供一个既定参数并在每3s运行一次，而不是创建一个`TimerTask`来完成此任务，我们可以只使用`MethodInvokingTimerTaskFactoryBean`创建一个`TimerTask`。它的配置文件见代码清单12-12。

代码清单12-12 使用MethodInvokingTimerTaskFactoryBean类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="target" class="com.apress.prospring2.ch12.timer.FooBean"/>

    <bean id="task" class="org.springframework.scheduling.timer.
        MethodInvokingTimerTaskFactoryBean">
        <property name="targetObject" ref="target" />
        <property name="targetMethod" value="someJob" />
        <property name="arguments" value="Hello World!" />
    </bean>

    <bean id="timerTask"
        class="org.springframework.scheduling.timer.ScheduledTimerTask">
        <property name="delay" value="1000" />
        <property name="period" value="3000" />
        <property name="timerTask" ref="task" />
    </bean>

    <bean id="timerFactory"
        class="org.springframework.scheduling.timer.TimerFactoryBean">
        <property name="scheduledTimerTasks">
            <list>
                <ref local="timerTask"/>
            </list>
        </property>
    </bean>

</beans>
```

我们使用一个 `MethodInvokingTimerTaskFactoryBean` 类的定义替换了我们自定义的 `TimerTask` bean 的定义。为了配置 `MethodInvokingTimerTaskFactoryBean`，我们设置需要调用的目标为对另外一个 bean 的引用，以及执行的方法和执行时使用的参数。`MethodInvokingTimerTaskFactoryBean` 类提供的 `TimerTask` 可以用普通的方式使用：包装在 `ScheduledTimerTask` 类中，并把它传递给 `TimerFactoryBean`。

代码清单12-13展示了一个简单的驱动程序来进行测试。

代码清单12-13 使用 `MethodInvokingTimerTaskFactoryBean` 类

```
package com.apress.prospring2.ch12.timer;

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

public class MethodInvokerScheduling {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "./ch12/src/conf/timerMethodInvoker.xml");
        System.in.read();
    }
}
```


运行这个示例将输出熟悉的结果,“Hello World!”消息即时出现在你的控制台。使用MethodInvokingTimerTaskFactoryBean消除了对创建自定义的TimerTask实现的需求,而后者只是包装了一个业务方法的执行。

基于JDK Timer的调度,使用一个简单易懂的框架提供了对一个程序基本调度需求的支持。尽管JDK Timer的触发器系统有点死板,但是它提供了基本的设计来让你完成简单的任务调度。使用Spring为Timer提供支持类,你可以在外部对任务进行配置,可以更容易地实现任务的添加和移除,而无需修改任何代码。你使用MethodInvokingTimerTaskFactoryBean可以避免创建除了调用一个业务方法以外什么都不做的TimerTask实现,这样也减少了你需要编写和维护的代码量。

当我们需要支持复杂的触发器时,例如每个周一、周三和周五下午3点执行一个任务,JDK Timer调度的主要缺陷就显露出来了。在12.2节,我们开始介绍Quartz引擎,它给任务调度提供了更全面的支持,并且和Timer任务调度一样与Spring完全集成。

12.2 使用 OpenSymphony Quartz 来调度任务

开源Quartz项目是一个专用的任务调度引擎,它可以在Java EE和Java SE中使用。Quartz提供了极其全面的特性,例如持久化任务、集群和分布式事务。在本书中不会涉及集群或分布式事务的特性——你可以在www.opensymphony.com/quartz发现更多的信息。Spring对Quartz的集成与Spring对Timer的集成在任务的声明式配置、触发器和调度上都非常相似。除此之外, Spring还提供额外的任务持久化特性,可以让Quartz的调度加入到Spring事务管理机制中。

12.2.1 Quartz 简介

Quartz是一个极其强大的任务调度引擎,我们不能期望在本章的剩余部分中覆盖到Quartz的所有方面。但是我们将讨论Quartz与Spring相关的主要方面,同时我们也将介绍如何在Spring应用程序中使用Quartz。和我们对Timer的介绍一样,我们先对Quartz有一个初步了解,然后再讨论Quartz与Spring的集成。

Quartz的核心由两个接口和两个类组成: Job和Scheduler接口, JobDetail和Trigger类。从它们的名字可以了解它们的主要用途。只有JobDetail类的角色不太清晰。不同于基于Timer的任务调度,任务并不是从一个实现Job接口的类的实例开始的。实际上, Quartz将在它需要时再创建Job类的实例。你可以使用JobDetail类封装任务状态,并传递信息给一个任务或在任务的连续执行中间保存信息。使用基于Timer的任务调度,并没有关于自己封装触发器逻辑的触发器概念。Quartz支持一种可插拔触发器的架构,这可以让你在合适的时机创建你自己的实现。但是,你很少会需要创建你自己的Trigger实现,因为Quartz已经提供了极其强大的CronTrigger类,可以让你使用Cron表达式来对任务执行进行细粒度的控制。

1. 简单的任务调度

用Quartz创建一个任务你只需简单创建一个实现Job接口的类。Job接口定义了一个execute()方法,你可以用它调用你的业务逻辑。Quartz传递JobExecutionContext实例给execute()方法,这可以让你访问当前执行任务的上下文数据。我们将在“使用JobDataMap类”小节详细介绍它。

代码清单12-14展示了一个简单的Job实现,它只向标准输出发送“Hello World”消息。

代码清单12-14 创建一个简单任务

```
package com.apress.prospring2.ch12.quartz;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class HelloWorldJob implements Job {

    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        System.out.println("Hello World!");
    }

}
```

为了调度此任务执行，我们需要先得到一个Scheduler实例，然后创建一个包含任务信息的JobDetail bean，最后创建一个Trigger管理任务的执行。代码如代码清单12-15所示。

代码清单12-15 在Quartz中调度任务

```
package com.apress.prospring2.ch12.quartz;

import java.util.Date;

import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SimpleTrigger;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;

public class HelloWorldScheduling {

    public static void main(String[] args) throws Exception {

        Scheduler scheduler = new StdSchedulerFactory().getScheduler();
        scheduler.start();

        JobDetail jobDetail = new JobDetail("helloWorldJob",
            Scheduler.DEFAULT_GROUP, HelloWorldJob.class);

        Trigger trigger = new SimpleTrigger("simpleTrigger",
            Scheduler.DEFAULT_GROUP, new Date(), null,
            SimpleTrigger.REPEAT_INDEFINITELY, 3000);

        scheduler.scheduleJob(jobDetail, trigger);
    }

}
```

代码的开始使用StdSchedulerFactory来得到一个Scheduler实例。我们不准深入讨论这个类的细节，但你可以从OpenSymphony Web网站上的Quartz文档中查阅更多信息。现在，我们知道StdSchedulerFactory.getScheduler()返回一个可运行的Scheduler实例。在Quartz中，一个Scheduler可以被启动（start）、中止（stop）和暂停（pause）。如果Scheduler没有启动或暂停，则没有触发器被激活，那么我们可以使用start()方法启动Scheduler。

412 第12章 基于Spring的任务调度

下一步,我们创建调度任务的JobDetail实例,并传递3个参数给构造方法。第一个参数是任务名,它用于引用此任务。当你使用一个Scheduler接口的管理方法时(如pauseJob()),任务将暂停。第二个参数是任务组名,这里使用默认名,任务组名用于引用集合起来的一组任务,如你可以使用Scheduler.pauseJobGroup()方法暂停一组任务。你应该注意每个组中的任务名是唯一的。第三个即最后一个参数,是实现了特定任务的类。

创建JobDetail实例后,我们继续创建一个Trigger。在本例中,我们使用SimpleTrigger类,它提供JDK Timer风格的触发器行为。传递给SimpleTrigger构造方法的第一个和第二个参数分别是触发器名和任务组名。这两个参数和JobDetail的参数作用相似。触发器名在它所在的任务组中必须是唯一的,否则会抛出一个异常。第三个和第四个参数都是Date类型,标识触发器的启动和结束时间。通过把结束时间设置为null,我们表明任务没有结束时间。这个为触发器指定结束时间的能力,在使用Timer时是不可用的。下一个参数是重复计数,这可以让你指定Trigger被激发的最大次数。我们可以使用REPEAT_INDEFINITELY来让触发器可以被激发无限次。最后一个参数是Trigger激发的时间间隔,是一个毫秒数。我们已经定义了一个3 s的时间间隔。

示例的最后步骤是调用Scheduler.schedule()来调度任务,使用JobDetail实例和Trigger实例作为参数。如果你运行这个应用程序,你会看到熟悉的“Hello World!”信息流不断地被发送到控制台。

2. 使用JobDataMap类

在先前的示例中,关于任务执行的所有信息都是从任务本身得到的。但你也可以通过JobDetail类或Trigger类将状态传递给任务。每个JobDetail实例都有一个相关的JobDataMap实例,它实现了Map接口并可以让你使用键值对将与任务相关的数据传递给任务。但使用这种方法时,有一些与任务相关的因素需要斟酌一下。我们将在后面名为“关于任务持久化”小节讨论。

当你使用多个Trigger实现调度同一个任务,或在每次独立触发上为任务提供不同的数据时,Trigger的存储数据是有用的。map的入口可以通过JobExecutionContext上的JobDataMap访问,而JobExecutionContext可以使用getMergedJobDataMap()方法获得。正如方法名所暗示的,JobExecutionContext上的JobDataMap是JobDetail和Trigger上的JobDataMap的合并,而若数据有冲突,存储在Trigger中的数据将覆盖存储在JobDetail中的数据。

在代码清单12-16中,你可以看到一个Job的示例,它使用合并后的JobDataMap数据执行任务处理。

代码清单12-16 使用JobDataMap类

```
package com.apress.prospring2.ch12.quartz;

import java.util.Map;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class MessageJob implements Job {

    public void execute(JobExecutionContext context) throws JobExecutionException {
```

12.2 使用 OpenSymphony Quartz 来调度任务 413

12

```
Map properties = context.getMergedJobDataMap();

System.out.println("Previous Fire Time: "
    + context.getPreviousFireTime());
System.out.println("Current Fire Time: " + context.getFireTime());
System.out.println("Next Fire Time: " + context.getNextFireTime());
System.out.println(properties.get("message"));
System.out.println(properties.get("jobDetailMessage"));
System.out.println(properties.get("triggerMessage"));
System.out.println("");
}
}
```

我们可以从合并后的JobDataMap中抽取对象，它使用message、jobDetailMessage和triggerMessage作为关键词并将其发送到标准输出。也要注意我们可以从JobExecutionContext得到该任务上一次、当前和下一次执行的相关信息。

在代码清单12-17中，你可以看到一个关于调度任务时如何用JobDetail的数据填充JobDataMap的示例。

代码清单12-17 向JobDetail的JobDataMap中添加数据

```
package com.apress.prospring2.ch12.quartz;

import org.quartz.Scheduler;
import org.quartz.SimpleTrigger;
import org.quartz.JobDetail;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;

import java.util.Map;
import java.util.Date;

public class JobDetailMessageScheduling {

    public static void main(String[] args) throws Exception {
        Scheduler scheduler = new StdSchedulerFactory().getScheduler();
        scheduler.start();

        JobDetail jobDetail = new JobDetail("messageJob",
            Scheduler.DEFAULT_GROUP, MessageJob.class);

        Map map = jobDetail.getJobDataMap();
        map.put("message", "This is a message from Quartz");
        map.put("jobDetailMessage", "A jobDetail message");

        Trigger trigger = new SimpleTrigger("simpleTrigger",
            Scheduler.DEFAULT_GROUP, new Date(), null,
            SimpleTrigger.REPEAT_INDEFINITELY, 3000);

        scheduler.scheduleJob(jobDetail, trigger);
    }
}
```

你可能发现这里的很多代码都和代码清单12-15中的一样。但需注意的是，一旦JobDetail实例被创建，我们会访问JobDataMap并向它增加两条消息，关键词分别为message和jobDetailMessage。若运行这个例子，并让它反复运行几次任务，你会得到和下面相似的输出：

414 第12章 基于Spring的任务调度

```
Previous Fire Time: null
Current Fire Time: Tue Oct 23 11:02:19 BST 2007
Next Fire Time: Tue Oct 23 11:02:22 BST 2007
This is a message from Quartz
A jobDetail message
null

Previous Fire Time: Tue Oct 23 11:02:19 BST 2007
Current Fire Time: Tue Oct 23 11:02:22 BST 2007
Next Fire Time: Tue Oct 23 11:02:25 BST 2007
This is a message from Quartz
A jobDetail message
null

Previous Fire Time: Tue Oct 23 11:02:22 BST 2007
Current Fire Time: Tue Oct 23 11:02:25 BST 2007
Next Fire Time: Tue Oct 23 11:02:28 BST 2007
This is a message from Quartz
A jobDetail message
null
```

你可以看到在前一次、当前和下一次执行时间的信息显示后，保存在JobDataMap中的两条消息被发送到标准输出。

代码清单12-18展示了一个示例，它也是在Trigger上提供数据和合并两个JobDataMap实例数据值。

代码清单12-18 在Trigger上使用JobDataMap

```
package com.apress.prospring2.ch12.quartz;

import org.quartz.JobDetail;

import org.quartz.Scheduler;
import org.quartz.SimpleTrigger;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;

import java.util.Date;

public class TriggerMessageScheduling {

    public static void main(String[] args) throws Exception {
        Scheduler scheduler = new StdSchedulerFactory().getScheduler();
        scheduler.start();

        JobDetail jobDetail = new JobDetail("triggerMessageJob",
            Scheduler.DEFAULT_GROUP, MessageJob.class);
        jobDetail.getJobDataMap().put("message", "This is a message from Quartz");
        jobDetail.getJobDataMap().put("jobDetailMessage", "My job details data.");

        Trigger trigger = new SimpleTrigger("simpleTrigger",
            Scheduler.DEFAULT_GROUP, new Date(), null,
            SimpleTrigger.REPEAT_INDEFINITELY, 3000);
        trigger.getJobDataMap().put("message", "Message from Trigger");
        trigger.getJobDataMap().put("triggerMessage", "Another trigger message.");
    }
}
```

```
        scheduler.scheduleJob(jobDetail, trigger);  
    }  
}
```

正如所见到的, JobDetail使用和先前相同的配置。我们只是向Trigger增加两条消息: 作为关键字的消息和触发器消息。运行这个实例产生与下面内容相似的输出:

```
Previous Fire Time: null  
Current Fire Time: Tue Oct 23 11:14:22 BST 2007  
Next Fire Time: Tue Oct 23 11:14:25 BST 2007  
Message from Trigger  
My job details data.  
Another trigger message.  
  
Previous Fire Time: Tue Oct 23 11:14:22 BST 2007  
Current Fire Time: Tue Oct 23 11:14:25 BST 2007  
Next Fire Time: Tue Oct 23 11:14:28 BST 2007  
Message from Trigger  
My job details data.  
Another trigger message.  
  
Previous Fire Time: Tue Oct 23 11:14:25 BST 2007  
Current Fire Time: Tue Oct 23 11:14:28 BST 2007  
Next Fire Time: Tue Oct 23 11:14:31 BST 2007  
Message from Trigger  
My job details data.  
Another trigger message.
```

注意关键词message的值是来自Trigger JobDataMap, 而不是我们在JobDetail中定义的。

可以看到, 使用Spring配置Quartz任务调度时, 可以在Spring配置文件中创建JobDataMap, 这可以让你将所有的任务配置在程序外部完成。

3. 使用CronTrigger类

在先前示例中, 我们使用SimpleTrigger类, 它提供的触发器功能和JDK Timer类提供的功能非常相似。然而, Quartz的优势是在于它使用CronTrigger提供对复杂触发器表达式的支持。CronTrigger是基于Unix Cron守护进程, 它是一个调度程序, 支持简单而强大的触发器语法。你可以使用CronTrigger快速并精确地定义SimpleTrigger类不可能或很难处理的触发器表达式。例如, 你可以定义一个触发器, 它“在14:00到17:00之间, 从每分钟的第3 s开始, 每隔5 s触发一次”或“在每个月的最后一个星期五触发”。

CronTrigger语法表达式, 也叫Cron表达式, 包括6个必需组件和一个可选组件。Cron表达式单独写一行, 并且每个组件间用空格隔开。只有最后或者说最右边的组件是可选的。表12-1详细说明了Cron的各个组件。

表12-1 Cron表达式的组件

位 置	含 义	允许的特殊字符
1	秒 (0-59)	, -、*和/
2	分 (0-59)	, -、*和/
3	小时 (0-23)	, -、*和/
4	日期 (1-31)	, -、*、/、?、L、W和C

416 第12章 基于Spring的任务调度

5	月（JAN-DEC或1-12）	, -、*和/
6	星期（SUN-SAT或1-7）	, -、*、/、?、L、C和#
7	年（可选，1970-2099），若为空，表示全部时间范围	, -、*和/

每一个组件接受一个指定范围的值，如秒和分接受0~59而日期则接受1~31。对于星期或月，你可以使用数字例如星期天数的1~7，或者文本SUN-SAT。

每一个部分也接受一组给定的特殊字符，例如当把“*”放在小时部分时表示每个小时，在星期部分使用一个表达式例如6L表示每个月最后一个星期五。表12-2说明了Cron的通配符和特殊字符。

表12-2 Cron表达式的通配符和特殊字符

特殊字符	说 明
*	任意值。这个特殊字符可以被使用在表达式的任何域，表示不应该检查该值。因此，我们的Cron表达式可以在1970~2099年的任意一天，任意月份和星期中的任意一天触发
?	无特定值。此特殊字符通常和其他指定的值一起使用，表示必须显示该值但不能检查
-	范围。例如小时部分10-12表示10:00、11:00和12:00
,	列分隔符。此特殊字符可以让你指定一系列的值，例如在星期域中指定MON、TUE和WED
/	增量。此特殊字符表示一个值的增量，例如0/1表示从0开始，每次增加1 min
L	L是英文单词Last的缩写。它在日期和星期域中表示有一点不同，挡在日期域中使用时，表示这个月的最后一天（3月31日、2月29日等）。当使用在星期域时，它永远是同一个值：7——星期六。当你希望使用星期中某一天时，L字符非常有用。例如，星期域中6L表示每个月的最后一个星期五
W	w只可以用在月域部分的天字段，指定临近一周（星期一到星期五）给该月的既定日期。设置值7w，触发器若第7此触发在一个星期六，那么它将在第六天触发。若第七天是星期日，触发器将在星期一触发，即第八天。注意触发器由于周六是第一天，实际触发发生在第三天
#	此特殊字符可使用在星期域中，表示该月的第几个星期。例如1#2表示每个月的第一个星期一
C	日历值。它可以使用在日期和星期域中。日期值是根据一个给定的日历计算出来的。在日期域中给定一个20c将在20日（日历包括20日）或20日后日历中包含的第一天（不包括20日）激活触发器。例如在一个星期域中使用6c表示日历中星期五（日历包括星期五）或者第一天（日历不包括星期五）

当写Cron表达式时，需要记住的最后一件事是夏时制的时间变化。由于夏时制的变化可能引起触发器在秋天时在Spring中被触发两次或从不触发。

实际上Cron表达式要比我们这里所讨论的有更多变化。你可以在CronTrigger类的Java文档中找到关于cron语法的更多介绍。

代码清单12-19展示了一个使用CronTrigger类的示例。

代码清单12-19 使用CronTrigger类

12.2 使用 OpenSymphony Quartz 来调度任务 417

```
package com.apress.prospring2.ch12.quartz;

import java.util.Map;

import org.quartz.CronTrigger;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;

public class CronTriggerExample {

    public static void main(String[] args) throws Exception {
        Scheduler scheduler = new StdSchedulerFactory().getScheduler();
        scheduler.start();

        JobDetail jobDetail = new JobDetail("messageJob",
            Scheduler.DEFAULT_GROUP, MessageJob.class);

        Map map = jobDetail.getJobDataMap();
        map.put("message", "This is a message from Quartz");

        String cronExpression = "3/5 * 14,15,16,17 * * ?";

        Trigger trigger = new CronTrigger("cronTrigger",
            Scheduler.DEFAULT_GROUP, cronExpression);

        scheduler.scheduleJob(jobDetail, trigger);
    }
}
```

对你来说上面的许多代码应该不陌生了。唯一的重要不同是我们使用Cron表达式。CronTrigger类的创建和SimpleTrigger的创建是非常相似的，你必须提供一个任务名和组名。为了帮助你理解示例中的Cron表达式，我们将它分解为几个部分。

第一个部分3/5，意思是每分钟的第3 s作为开始每5 s运行一次。第二个部分*代表每分钟。第三部分，14, 15, 16, 17，限制触发器只能够运行在14:00到17:59之间——也就是时间必须以14、15、16或17开头。接下来的部分都是通配符——?，表示触发器可以运行在一个星期的任意一天。这个表达式将使得触发器14:00到17:59之间每1 min的第3 s开始运行，并每5 s运行一次。

如果你运行该示例，根据你的运行时间，你要么看到一个空白的屏幕要么打印出一直增长的“Hello World!”消息。试着修改表达式的第一个部分来改变频率或每分钟触发器开始的时间。你也应该试着修改其他部分看看产生的效果。

CronTrigger类对于所有的触发器需求来说都是很好的。但是在你需要考虑一些例外情况时，表达式可能会变得过于复杂。例如，考虑每个周一、周三和周五的11:00和15:00都有个进程会检查一个用户的任务列表。现在考虑一下用户放假时，你想阻止触发器被触发会发生的事情。幸运的是，Quartz使用Calendar接口对这种需求提供了支持。你使用Calendar接口可以精确的从一个触发器的通常调度计划中显式地包含或者排除某一段时间。Quartz包含了6个Calendar接口的实现，其中一个HolidayCalendar类，它从一个触发器的调度计划中排除日期。代码清单12-20展示了对于先前示例的修改，它使用HolidayCalendar排除2007年12月25日。

代码清单12-20 使用HolidayCalendar显式排除日期

418 第12章 基于Spring的任务调度

```
package com.apress.prospring2.ch12.quartz;

import java.util.Calendar;
import java.util.Map;

import org.quartz.CronTrigger;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;
import org.quartz.impl.calendar.HolidayCalendar;

public class CronWithCalendarExample {

    public static void main(String[] args) throws Exception {
        Scheduler scheduler = new StdSchedulerFactory().getScheduler();
        scheduler.start();

        // create a calendar to exclude a particular date
        Calendar cal = Calendar.getInstance();
        cal.set(2007, Calendar.DECEMBER, 25);

        HolidayCalendar calendar = new HolidayCalendar();
        calendar.addExcludedDate(cal.getTime());

        // add to scheduler
        scheduler.addCalendar("xmasCalendar", calendar, true, false);

        JobDetail jobDetail = new JobDetail("messageJob",
            Scheduler.DEFAULT_GROUP, MessageJob.class);
        Map map = jobDetail.getJobDataMap();
        map.put("message", "This is a message from Quartz");

        String cronExpression = "3/5 * 14,15,16,17 * * ?";

        Trigger trigger = new CronTrigger("cronTrigger",
            Scheduler.DEFAULT_GROUP, cronExpression);

        trigger.setCalendarName("xmasCalendar");

        scheduler.scheduleJob(jobDetail, trigger);
    }
}
```

从上面代码中，你可以看到我们创建了一个 `HolidayCalendar` 的实例，然后使用 `addExcludedDate()` 方法排除了12月25日。我们使用创建好的 `Calendar` 实例的 `addCalendar()` 方法将 `Calendar` 加到 `Scheduler` 中，并给它一个名字：`xmasCalendar`。接着，在加入 `CronTrigger` 之前，我们将它和 `xmasCalendar` 关联起来。使用这个方法可以将你从创建复杂的 `Cron` 表达式中解放出来，而这些表达式只是排除了一些日期。

4. 关于任务持久化

`Quartz` 为任务持久化提供了支持，这可以让你在运行时增加任务或者对现存的任务进行修改，并为后续任务的执行持久化这些变更和增加的部分。中心概念就是 `JobStore` 接口，`Quartz` 执行持久化时将使用它的实现。默认情况下，`Quartz` 使用 `RAMJobStore` 实现，它简单的把任务放在内存中。其他可用实现是 `JobStoreCMT` 和 `JobStoreTX`。这两个类都使用一个配置好的数据源来持久化任务细

节，这就支持将任务的创建和修改作为事务的一部分。JobStoreCMT实现倾向于用在一个服务器环境里并参加容器管理事务。对于独立的程序，你应该使用JobStoreTX实现。Spring为JobStore提供了自己的LocalDataSourceJobStore实现，它可以参加Spring管理的事务。当我们讨论Spring对Quartz支持时我们会关注该实现。

从前面开始，你了解了如何修改JobDataMap的内容来在同一个任务的不同执行中传递消息。但是，如果你希望使用JobStore而不是RAMJobStore来运行该示例，你将发现它不能工作。原因是Quartz支持无状态和有状态的任务。在使用RAMJobStore并修改JobDataMap时，你实际上是直接修改保存的内容，任务的内容是不重要的，但是当你使用RAMJobStore以外的其他实现时就不同了。一个无状态的Job只能使用它被增加到Scheduler中时JobDataMap所保存的数据，而有状态的Job在每次执行之后都会持久化它的JobDataMap。为了使一个Job有状态，需要实现StatefulJob接口而不是Job接口。StatefulJob是Job的一个子接口，因此你无需额外实现Job接口。你也应该知道，由于Quartz把JobDataMap作为一个序列化的blob写到数据库中，所以在使用Job持久化时放在JobDataMap里的任何数据都应该是可序列化的。

12.2.2 Spring 对 Quartz 的支持

Spring对Quartz的集成与Spring对Timer的集成相似，它可以让你使用Spring配置文件对你的任务调度进行完全配置。除此之外，Spring还提供辅助类集成Quartz JobStore接口，可以在Spring配置文件中配置任务持久化，并将对任务的修改加入到Spring事务管理中。

1. 使用Spring调度任务

如你所料，很多需要用于调度Quartz任务的代码被放到Spring配置文件中。实际上，你仅仅需要在程序中加载ApplicationContext来让配置生效，Spring将自动启动调度器。

在代码清单12-21，你可以看到每3 s运行一次的MessageJob类的配置代码，而在先前代码清单12-16中需要配置。

代码清单12-21 声明式配置调度计划

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="job" class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass"
            value="com.apress.prospring2.ch12.quartz.MessageJob"/>
    </bean>
</beans>
```

```
<property name="jobDataAsMap">
  <map>
    <entry key="message"
      value="This is a message from the Spring config file!"/>
  </map>
</property>
</bean>

<bean id="trigger"
  class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail" ref="job"/>
  <property name="startDelay" value="1000"/>
  <property name="repeatInterval" value="3000"/>
  <property name="jobDataAsMap">
    <map>
      <entry key="triggerMessage"
        value="Trigger message from the Spring config file!"/>
    </map>
  </property>
</bean>

<bean id="schedulerFactory"
  class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref local="trigger"/>
    </list>
  </property>
</bean>
</beans>
```

在这里，我们使用了JobDetailBean类，它扩展了JobDetail类，并用一种可声明的方式配置任务数据。JobDetailBean提供了更多的Spring可访问的JavaBean风格的属性，它也设置了合理的默认值给那些不得不自己提供的属性。例如，我们通常并不想提供一个任务名或者组名。默认情况下，JobDetailBean使用<bean>标签的ID作为任务名，并使用Scheduler的默认组作为组名。注意我们可以通过jobDataAsMap的属性把数据加入到JobDataMap的属性中。属性名不仅仅是摆设——因为你不可以直接向JobDataMap的属性中添加数据。它是JobDataMap类型，Spring配置文件不支持该类型。

配置好JobDetailBean后，下一步是创建一个触发器。Spring提供两个类：SimpleTriggerBean和CronTriggerBean，它们包装了SimpleTrigger和CronTrigger类，可以让你声明式地配置它们并把它们和一个JobDetail bean关联——所有这些都是你的配置文件中完成的。注意在上面代码清单11-21的示例中，我们在程序开始时定义了1 s的延迟而后每3 s执行一次任务。默认情况下，SimpleTriggerBean把可重复执行的次数设置为无限。

需要配置的最后部分是关于SchedulerFactoryBean。默认情况下，SchedulerFactoryBean创建一个StdSchedulerFactory的实例，后者创建Scheduler的实现来替换的类应该实现SchedulerFactory接口。你需要配置调度计划的唯一属性是triggers属性，它接受一个TriggerBean的列表作为参数。

由于所有的任务调度配置都在配置文件中，因此你只需要很少的代码来实际启动Scheduler和执行Job实例。实际上，你需要做的所有工作就是创建ApplicationContext，如代码清单12-22所示。

代码清单12-22 声明式配置调度计划

```
package com.apress.prospring2.ch12.quartz.spring;

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

public class SimpleSpringQuartzIntegrationExample {

    public static void main(String[] args) {
        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "./ch12/src/conf/quartz-simple.xml");
    }
}
```

正如你所见到的，这个类除了使用代码清单12-21中的配置创建一个ApplicationContext类的实例之外不做任何工作。若运行这个程序并让调度的任务触发若干次，你最后得到下面的输出：

```
Previous Fire Time: null
Current Fire Time: Tue Oct 23 11:24:31 BST 2007
Next Fire Time: Tue Oct 23 11:24:34 BST 2007
This is a message from the Spring configuration file!
null
Trigger message from the Spring configuration file!

Previous Fire Time: Tue Oct 23 11:24:31 BST 2007
Current Fire Time: Tue Oct 23 11:24:34 BST 2007
Next Fire Time: Tue Oct 23 11:24:37 BST 2007
This is a message from the Spring configuration file!
null
Trigger message from the Spring configuration file!

Previous Fire Time: Tue Oct 23 11:24:34 BST 2007
Current Fire Time: Tue Oct 23 11:24:37 BST 2007
Next Fire Time: Tue Oct 23 11:24:40 BST 2007
This is a message from the Spring configuration file!
null
Trigger message from the Spring configuration file!
```

你会注意到它和前面一个MessageJob示例的运行结果非常相似，但是所显示的消息是在Spring配置文件中配置的。

2. 使用持久化任务

Quartz的一个重要特性是它能创建有状态、可持久化的任务。这让你可以使用一些在基于Timer的任务调度中所没有的强大功能。你可以使用持久化任务在运行时把任务添加到Quartz中，并且重启程序时任务仍然在你的程序中。你也可以使用修改两个Job执行期间传递的JobDataMap，而前面所做的修改在重启后仍然有效。

在本示例中，我们将调度两个任务，一个使用Spring配置机制而另一个将在运行时进行调度。我们将看到Quartz持久化机制如何为这些Job处理对JobDataMap的修改，以及在程序的后续执行中将发生什么情况。

在开始之前，你需要创建一个Quartz用于保存任务信息的数据库。在Quartz发布版中（使用1.6.0版），你将发现一个关于数据库选项的脚本，它可以用于选择不同的关系数据库。在示例中，我们使

422 第12章 基于Spring的任务调度

用Oracle, 但是你若使用其他的数据库, 并且Quartz为它提供一个数据库脚本时, 那就不应该有问题。在1.6.0版中, 使用Quartz发布版的docs/dbTables子目录。一旦你找到为你选择的数据库所使用的脚本, 可以对你的数据库执行它并检查数据库中是否创建了12个表, 每一个表的前缀都是quartz。

下一步, 创建测试Job。因为我們希望在Job执行中对JobDataMap进行修改, 我们需要提示Quartz让它把这个Job作为有状态的Job来处理。我们通过实现StatefulJob接口而不是Job接口来通知Quartz。可以参见代码清单12-23中的代码。

代码清单12-23 创建一个有状态的任务

```
package com.apress.prospring2.ch12.quartz.spring;

import java.util.Map;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.StatefulJob;

public class PersistentJob implements StatefulJob {

    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        Map map = context.getJobDetail().getJobDataMap();
        System.out.println "[" + context.getJobDetail().getName() + "]"
            + map.get("message");
        map.put("message", "Updated Message");
    }
}
```

StatefulJob接口没有为你的类增加需要实现的额外方法: 它仅是一个标识用来告诉Quartz应在每次执行之后持久化JobDetail。在这里, 可以看到我们显示保存在JobDataMap的消息和Job名。

下一步, 在Spring里配置Job并将一个DataSource配置给Scheduler, Scheduler将用此DataSource进行持久化, 如代码清单12-24所示。

代码清单12-24 在Spring中配置Quartz持久化

12.2 使用 OpenSymphony Quartz 来调度任务 423

12

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean id="job" class="org.springframework.scheduling.quartz.JobDetailBean">
    <property name="jobClass"
      value="com.apress.prospring2.ch12.quartz.spring.PersistentJob"/>
    <property name="jobDataAsMap">
      <map>
        <entry key="message" value="Original Message"/>
      </map>
    </property>
  </bean>

  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url"
      value="jdbc:oracle:thin:@oracle.devcake.co.uk:1521:INTL"/>
    <property name="username" value="PROSPRING"/>
    <property name="password" value="x*****6"/>
  </bean>

  <bean id="trigger"
    class="org.springframework.scheduling.quartz.SimpleTriggerBean">
    <property name="jobDetail" ref="job"/>
    <property name="startDelay" value="1000"/>
    <property name="repeatInterval" value="3000"/>
  </bean>

  <bean id="schedulerFactory"
    class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
      <list>
        <ref local="trigger"/>
      </list>
    </property>
    <property name="dataSource" ref="dataSource"/>
  </bean>
</beans>
```

你将发现这里许多的配置代码都是代码清单 12-21 中的配置代码，代码中最重要的部分是 `dataSource` bean。我们在代码中使用 `Spring` 的 `SingleConnectionDataSource` 类：这是对于测试环境时很有用的 `dataSource` 实现，但不能在产品运行环境中使用（具体原因请参考该类的 `Javadoc` 文档）。记住，你需要根据你的应用环境配置来修改连接细节。关于配置 `Spring` 中其他 `DataSource` 的更多细节，可以查看第 8 章。

我们使用配置好的 `dataSource` bean 来设置 `SchedulerFactoryBean` 的 `dataSource` 属性。通过这样做，我们让 `Spring` 创建一个 `Scheduler`，使用既定的 `DataSource` 配置持久化 `Job` 数据。实际上，这是由 `Spring` 自己的 `JobStore` 实现——`LocalDataSourceJobStore` 来完成的。

完成配置后，剩下的工作就只是将它加载到一个程序中，并在运行时加载另一个 `Job` 到 `Scheduler`。代码清单 12-25 展示了具体代码。

424 第12章 基于Spring的任务调度

```
package com.apress.prospring2.ch12.quartz.spring;

import java.util.Date;

import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SimpleTrigger;
import org.quartz.Trigger;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class SpringWithJobPersistence {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new FileSystemXmlApplicationContext(
            "./ch12/src/conf/quartzPersistent.xml");

        // get the scheduler
        Scheduler scheduler = (Scheduler) ctx.getBean("schedulerFactory");

        JobDetail job = scheduler.getJobDetail("otherJob",
            Scheduler.DEFAULT_GROUP);

        if (job == null) {
            // the job has not yet been created
            job = (JobDetail) ctx.getBean("job");
            job.setName("otherJob");
            job.getJobDataMap().put("message", "This is another message");

            Trigger trigger = new SimpleTrigger("simpleTrigger",
                Scheduler.DEFAULT_GROUP, new Date(), null,
                SimpleTrigger.REPEAT_INDEFINITELY, 3000);

            scheduler.scheduleJob(job, trigger);
        }
    }
}
```

上述代码无需解释。但是需要注意在我们调用第二个Job之前，我们使用Scheduler.getJobDetail()方法检查它是否已经在Scheduler中。这样我们就不会再这个程序的后续执行时覆盖此Job。

在你第一次运行示例时，你将得到类似下面的输出：

```
[otherJob]This is another message
[job]Original Message
[otherJob]Updated Message
[job]Updated Message
[otherJob]Updated Message
[job]Updated Message
```

如你所见，每个Job在第一次执行时，显示的消息是在Job调度时被配置在JobDataMap中的原始信息。在后续执行时，每一个Job显示了上一次执行时设置的已更新消息。若重启程序，你将会发现输出结果有些不同：

```
[otherJob]Updated Message  
[job]Updated Message  
[otherJob]Updated Message  
[job]Updated Message  
[otherJob]Updated Message  
[job]Updated Message
```

这次你可以看到,由于持久化了Job数据,你无需重新创建第二个Job并且JobDataMap仍使用程序上次执行所做的修改。

3. 使用Quartz调度任意任务

同基于Timer的调度类一样, Spring提供了使用Quartz调度执行任意方法的能力。我们不讨论细节,因为这和使用Timer的情况基本一样。使用MethodInvokingJobDetailFactoryBean代替MethodInvoking- TimerTaskFactoryBean,你可以自动创建JobDetail实现,而不是自动创建TimerTask实现。

12.3 任务调度时需考虑的因素

如果你打算在应用程序中增加任务调度能力,那在你选择使用何种调度器以及何种调度方式时,你应该考量一些因素。

12.3.1 选择一个调度器

你打算在应用程序中增加任务调度能力时,你所做的第一个决定是选择何种调度器。实际上这是一个很容易的选择。如果你只有简单地调度需求或存在不可以在程序中使用第三方库的限制,你应该使用基于Timer的任务调度。否则,使用Quartz。

即使你发现需求很简单,你还是使用Quartz,特别是你需要建立一个显式的Job实现时。使用此种方法,在你的需求变得更高级并且开始使用的是Quartz时,你可以很容易地增加持久性、事务性或更加复杂的触发器而无需修改Job相关的TimerTask。通常来说,我们发现使用Quartz将让我们对某种调度方式变得熟悉。既然有一种方法可以提供我们需要的所有功能,那我们开发人员就不用在这两种不同方式之间左右为难了。

12.3.2 剥离Job类中的任务逻辑

我们看到很多开发商将调度功能增加到一个应用程序时,所选择的常见做法就是把业务逻辑放在Job类或TimerTask类中。这通常是一个糟糕的做法。在大多数情况下,你需要依照需求来调度任务的执行,这就需要将任务逻辑从调度框架中剥离出来。

同时,你也没必要让你的业务逻辑和调度程序过度耦合。一个更好的方式是将业务逻辑放在一个独立的类中,然后为该类创建一个特定于你所选择的调度程序的简单包装器,也许更好的做法是使用合适的MethodInvoker*FactoryBean为你创建包装器。

12.3.3 任务执行和线程池

到目前为止,本章已经讨论了各种各样的任务调度方式,它们可以在指定时间点、指定时间间隔或指定时间点和时间间隔的结合来控制任务的执行。现在,我们将了解Spring中任务调度的另一种方

426 第12章 基于Spring的任务调度

式，它更少依赖时间点或时间间隔，而是即时或者事件触发任务执行。

例如，考虑一个Web服务器处理接踵而来的请求。建立这样服务器程序的简单方法是将每一个任务放在独立的线程中。这个可能工作绝对正常，但依赖于你正在构建的服务器及其环境。因为线程的创建需要时间和系统资源，所以你可能需要花费比任务执行更多的时间创建和销毁线程，甚至可能耗尽系统资源作为结束。为了稳定运行，服务器需要某些方式管理同时执行的任务。线程池和工作队列的概念正是用于处理这种情况。

1. java.util.concurrent包

Java 5 中一个令人欣喜的特性是增加了基于 Doug Lea 的 `util.concurrent` 包的 `java.util.concurrent` 包，它是提供高效并经良好测试的工具库，用来简化多线程应用程序的开发。该包提供 `Executor` 接口，它只定义一个 `execute(Runnable command)` 方法来执行 `Runnable` 任务。它从任务运行细节中抽象了任务提交。接口的实现提供了所有执行策略的排序方法：每任务一个线程、线程池或者同步执行命名的一些方法（你可以在 `Executor` 接口的 Javadoc 中发现上述方法的一些实现）。

为了给你一个小示例来演示如何使用 `Executor` 接口和子接口 `ExecutorService`，我们先创建一个执行任务，该任务使用先前代码清单 12-1 中 `HelloWorldTask` 的修补版本。我们使用 `HelloWorldTask` 的方式很简单，因为它扩展了实现 `Runnable` 接口的 `TimerTask` 类，但是我们不能看到使用不同 `Executor` 实现间的任务调度的不同之处。见代码清单 12-26。

代码清单 12-26 HelloWorldCountDownTask 类

```
package com.apress.prospring2.ch12;

public class HelloWorldCountDownTask implements Runnable {

    private String name;
    private int count = 4;

    public HelloWorldCountDownTask(String name) {
        this.name = name;
    }

    public void run() {
        while (count > 0) {
            count--;
            if (count == 0) {
                System.out.println(name + " says 'Hello World!'");
            } else {
                System.out.println(name + ": " + count);
                Thread.yield();
            }
        }
    }
}
```

任务做的事情是打印出从3开始的倒计时，接着调用 `Thread.yield()` 方法来使这个线程的执行暂停，这时其他线程可以被执行。作为一条语句，任务向世界说了一声“Hello World!”结束了执行。

接下来，正如代码清单 12-27 所展示的，我们将使用 `ExecutorService` 实现调度和执行这个任务。

代码清单12-27 使用ExecutorService调度执行任务

```
package com.apress.prospring2.ch12.executor;
import com.apress.prospring2.ch12.HelloWorldCountDownTask;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {

    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(2);

        service.execute(new HelloWorldCountDownTask("Anna"));
        service.execute(new HelloWorldCountDownTask("Beth"));
        service.execute(new HelloWorldCountDownTask("Charlie"));
        service.execute(new HelloWorldCountDownTask("Daniel"));

        service.shutdown();
    }
}
```

java.util.concurrent.Executors类为Executor和ExecutorService类提供了便利的工厂和工具方法。我们使用newFixedThreadPool()方法获得具有线程池的两个线程的ThreadPoolExecutor实例。然后我们提交了4个任务执行，并在所有任务执行结束后调用ExecutorService的shutdown()方法关闭ExecutorService。调用此方法以后，没有更多的任务增加到服务中。运行示例将打印输出如下信息：

```
Anna: 3
Anna: 2
Beth: 3

Anna: 1
Anna says 'Hello World!
Charlie: 3
Beth: 2
Charlie: 2
Beth: 1
Charlie: 1
Beth says 'Hello World!
Daniel: 3
Charlie says 'Hello World!
Daniel: 2
Daniel: 1
Daniel says 'Hello World!
```

注意这只同时执行两个任务，任务Charlie只在任务Anna执行完以后才被执行。你可以尝试线程池中不同数量的线程或不同Executor实现，会发现打印输出也会不同。

2. Spring的TaskExecutor接口

从2.0版开始，Spring就提供了先前讨论的Java 5 Executor框架的一个抽象接口。与java.util.concurrent.Executor一样，TaskExecutor接口只定义了一个execute(Runnable command)方法。它也可以在其他Spring组件内部使用，例如同步JMS和JCA环境支持，现在你无需使用Java 5就可以向自己的应用程序增加线程池特性。

428 第12章 基于Spring的任务调度

Spring提供了各种TaskExecutor的实现，见表12-3中的说明。

表12-3 Spring的TaskExecutor实现

实 现	说 明
SimpleAsyncTaskExecutor	此实现提供异步线程，每次调用都新建一个线程。它也可以设置并发总数限制以阻塞更多新的调用
SyncTaskExecutor	当我们选用该实现调度任务时，在调用线程中的执行会同步发生
ConcurrentTaskExecutor	该类实现了Spring的SchedulingTaskExecutor接口和Java5的java.util.concurrent.Executor接口，作为后者的封装类
SimpleThreadPoolTaskExecutor	此实现是Quartz的SimpleThreadPool类的子类，当线程池需要在Quartz和非Quartz组件间共享时非常有用
ThreadPoolTaskExecutor	此实现的使用方式和ConcurrentTaskExecutor实现相似，它开放的bean属性可以用来配置java.util.concurrent.ThreadPoolExecutor（需要Java 5支持）
TimerTaskExecutor	这个实现使用一个TimerTask作为后台的实现。在独立线程中调用，但在那个线程中是同步的
WorkManagerTaskExecutor	此实现使用CommonJ WorkManager作为底层实现，并实现了WorkManager接口

各种实现方式的不同之处在小示例中很容易看出来。代码清单12-28展示了Spring中3个TaskManager实现的配置：ThreadPoolTaskExecutor、SyncTaskExecutor和TimerTaskExecutor。

代码清单12-28 task-executor-context.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="threadPoolTaskExecutor"
          class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
        <property name="corePoolSize" value="5"/>
        <property name="maxPoolSize" value="10"/>
        <property name="queueCapacity" value="25"/>
    </bean>

    <bean id="synchTaskExecutor"
          class="org.springframework.core.task.SyncTaskExecutor"/>

    <bean id="timerTaskExecutor"
          class="org.springframework.scheduling.timer.TimerTaskExecutor">
        <property name="delay" value="3000"/>
        <property name="timer" ref="timer"/>
    </bean>

    <bean id="timer" class="java.util.Timer"/>
</beans>
```

然后我们可以使用已定义的bean从ApplicationContext中装载它们，并在一个TaskExecutor中使用，如代码清单12-29所示。

代码清单12-29 TaskExecutorExample类

```
package com.apress.prospring2.ch12.taskexecutor;

import org.springframework.core.task.TaskExecutor;
import com.apress.prospring2.ch12.HelloWorldCountDownTask;

public class TaskExecutorExample {

    private TaskExecutor taskExecutor;

    public TaskExecutorExample(TaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    public void executeTasks() {
        this.taskExecutor.execute(new HelloWorldCountDownTask("Anna"));
        this.taskExecutor.execute(new HelloWorldCountDownTask("Beth"));
        this.taskExecutor.execute(new HelloWorldCountDownTask("Charlie"));
        this.taskExecutor.execute(new HelloWorldCountDownTask("Daniel"));
    }
}
```

如你所见，我们使用和前面一样的4个HelloWorldCountDownTask实例。生成的输出强调了执行策略间的不同之处。如预期所料，代码清单12-30的SyncTaskExecutorExample类同步执行了任务。

代码清单12-30 SyncTaskExecutorExample类

```
package com.apress.prospring2.ch12.taskexecutor;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.task.TaskExecutor;

import java.io.IOException;

public class SynchTaskExecutorExample {

    public static void main(String[] args) throws IOException {
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "com/apress/prospring2/ch12/taskexecutor/task-executor-context.xml");

        TaskExecutor taskExecutor =
            (TaskExecutor)ctx.getBean("synchTaskExecutor", TaskExecutor.class);
        TaskExecutorExample example = new TaskExecutorExample(taskExecutor);
        example.executeTasks();
    }
}
```

运行代码，将产生和下面类似的输出：

430 第12章 基于Spring的任务调度

```
Anna: 3
Anna: 2
Anna: 1
Anna says 'Hello World!'
Beth: 3
Beth: 2
Beth: 1
Beth says 'Hello World!'
Charlie: 3
Charlie: 2
Charlie: 1
Charlie says 'Hello World!'
Daniel: 3
Daniel: 2
Daniel: 1
Daniel says 'Hello World!'
```

如果你的应用程序运行在Java 5或更高版本上,那么你可以配置此示例运行在任何Java 5规范的实现上。**Spring**的`ThreadPoolTaskExecutor`让你能够配置JDK 1.5的`ThreadPoolExecutor`的bean属性,作为一个**Spring TaskExecutor**来公开。另外,**Spring**为其他的Java 5 `Executor`实现提供了作为适配器类的`ConcurrentTaskExecutor`实现,它让从Java 1.4升级更容易。

适配器类实现了`TaskExecutor`和`Executor`两个接口。因为主要接口是`TaskExecutor`,异常处理遵循它的契约。例如,当任务执行不可以接受时,一个**Spring TaskRejectedException**异常将被抛出,而不会抛出`java.util.concurrent.RejectedExecutionException`异常。

`TaskExecutor`接口的一个更便利特性是它对运行时异常的包装。在一个任务失败抛出异常时,这种情况通常认为是致命的。若没有必要或没有可能修复,异常可能是非检查类型的,你的代码保持更轻量级,可以很容易地在各种`TaskExecutor`实现间切换。

12.4 小结

本章介绍了**Spring**中集成的各种任务调度机制。我们先讨论在使用JDK `Timer`时得到的基本任务调度支持和使用**Quartz**得到的更高级的支持。也学习了如何使用不同类型的触发器。另外,我们特别讨论了**Quartz**中的`CronTrigger`,它是一种创建符合现实生活的复杂调度计划的方法。

任务调度是企业级应用程序的一个重要组成部分,**Spring**提供了极好的支持来帮助你调度功能增加到应用中。在下一章中,我们将调查**Spring**对发送电子邮件的支持。