1. 核心特性：

依赖注入（dependency injection，DI）

面向切面编程（aspect-oriented programming，AOP）。

第**1**章　**Spring**之旅

**1.1　简化Java开发**

为了简化java开发，Spring采取了以下4种关键策略：

|  |
| --- |
| 基于POJO的轻量级和最小侵入性编程；  通过依赖注入和面向接口实现松耦合；  基于切面和惯例进行声明式编程；  通过切面和模板减少样板式代码。 |

**1.1.1　激发POJO的潜能**

很多框架通过强迫应用继承它们的类或实现它们的接口从而导致应用与框架绑死。

这种侵入式的编程方式在早期版本的Struts、WebWork、Tapestry以及无数其他的Java规范和框架中都能看到。

Spring不会强迫你实现Spring规范的接口或继承Spring规范的类，相反，在基于Spring构建的应用中，它的类通常没有任何痕迹表明你使用了Spring。最坏的场景是，一个类或许会使用Spring注解，但它依旧是POJO。

Spring通过DI来装配POJO。DI帮助应用对象彼此间保持松散耦合。

**1.1.2　依赖注入**

* DI如何实现

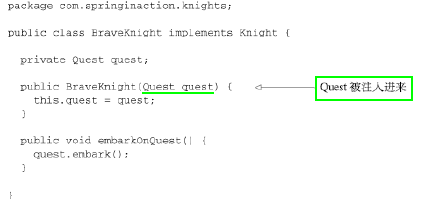
按传统做法，每个对象负责管理与自己相互协作的对象（依赖的对象）的引用，将会导致高度耦合和难以测试的代码。

* 耦合有两面性：

一方面，紧密耦合的代码难以测试、复用、理解，且典型地表现出“打地鼠”式的bug特性（修复一个bug，将会出现一个或者更多新bug）。

另一方面，一定程度的耦合又是必须的。为了完成有实际意义的功能，不同的类必须以适当的方式进行交互。总而言之，耦合是必须的，但应当被小心谨慎地管理。

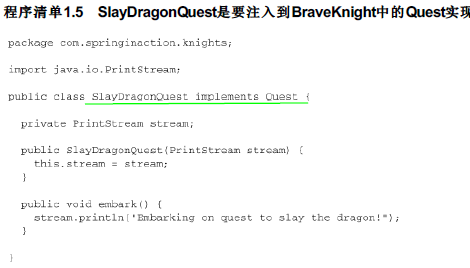
* 通过DI，对象的依赖关系将由系统中负责协调各对象的第三方组件在创建对象的时候进行设定。对象无需自行创建或管理它们的依赖关系，如图1.1所示，依赖关系将被自动注入到需要它们的对象当中去。



* BraveKnight没有自行创建探险任务，而是在构造时把探险任务作为构造器参数传入。是依赖注入的方式之一，即构造器注入。
* 传入的探险类型是Quest，是所有探险任务都必须实现的一个接口。所以，BraveKnight能够响应RescueDamselQuest、 SlayDragonQuest、 MakeRound TableRounderQuest等任意的Quest实现。

要点是BraveKnight没有与任何特定的Quest实现发生耦合。对它来说，被要求挑战的探险任务只要实现了Quest接口，那么具体是哪种类型的探险就无关紧要了。这就是DI所带来的最大收益——松耦合。

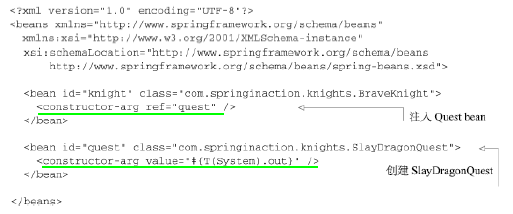
如一个对象只通过接口（而不是具体实现或初始化过程）来表明依赖关系，这种依赖就能够在对象本身毫不知情的情况下，用不同的具体实现进行替换。



* 装配：创建应用组件间协作的行为。

1. **XML方式(P20)**

knights.xml



现已声明BraveKnight和Quest关系，接下来只需装载XML配置文件，并把应用启动起来。

Spring通过应用上下文（Application Context）装载bean定义并把它们组装起来。

Spring应用上下文全权负责对象的创建和组装。

Spring自带了多种应用上下文实现，主要区别仅在于如何加载配置。

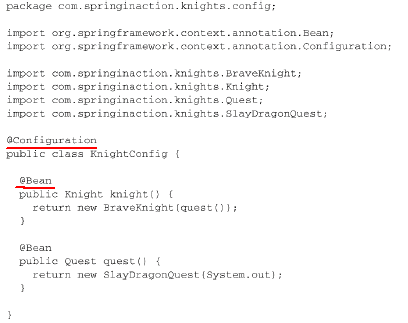
因knights.xml中的bean使用XML文件进行配置，所以选择ClassPathXmlApplicationContext[1]作为应用上下文比较合适。该类加载位于应用程序类路径下的一个或多个XML配置文件。



这里main()方法基于knights.xml创建了Spring应用上下文。随后它调用该应用上下文获取一个ID为knight的bean。得到Knight对象的引用后，只需简单调用embarkOnQuest()方法就可以执行所赋予的探险任务了。

注意这个类完全不知道英雄骑士接受哪种探险任务，且完全没有意识到这是由BraveKnight来执行。只有knights.xml文件知道哪个骑士执行哪种探险任务。

1. **基于java的配置**



尽管BraveKnight依赖于Quest，但它并不知道传递给它的是什么类型的Quest，也不知道这个Quest来自哪里。类似的，SlayDragonQuest依赖于PrintStream，但编码时它并不需要知道这个PrintStream是什么样子。

只有Spring通过它的配置，能了解这些组成部分如何装配起来。这样的话，就可在不改

变所依赖的类的情况下，修改依赖关系。

基于切面进行声明式编程。

**1.1.3　应用切面**

* DI能让相互协作的软件组件保持松散耦合

面向切面编程（aspect-oriented programming，AOP）允许把遍布应用各处的功能分离出来，

形成可重用组件。

往往被定义为促使软件系统实现关注点分离的一项技术。系统由许多不同组件组成，每一个组件各负责一块特定功能。除实现自身核心功能外，这些组件还经常承担着额外的职责。诸如日志、事务管理和安全这样的系统服务经常融入到自身具有核心业务逻辑的组件中去，这些系统服务通常被称为横切关注点，因它们会跨越系统的多个组件。

如将这些关注点分散到多个组件中去，代码将会带来双重的复杂性。

实现系统关注点功能的代码将会重复出现在多个组件中。意味着如你要改变这些关注点的逻辑，必须修改各个模块中的相关实现。

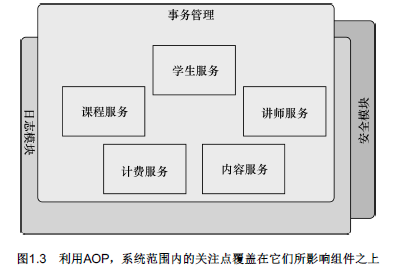
即使把这些关注点抽象为一个独立的模块，其他模块只是调用它的方法，但方法的调用还是会重复出现在各个模块中。

组件会因为那些与自身核心业务无关的代码而变得混乱。一个向地址簿增加地址条目的方法应该只关注如何添加地址，而不应该关注它是不是安全的或是否需要支持事务。

* AOP能使这些服务模块化，并以声明方式将它们应用到需影响的组件中。结果就是这些组件会具有更高的内聚性且会更加关注自身业务，完全不需了解涉及系统服务所带来复杂性。总之，AOP能够确保POJO的简单性。

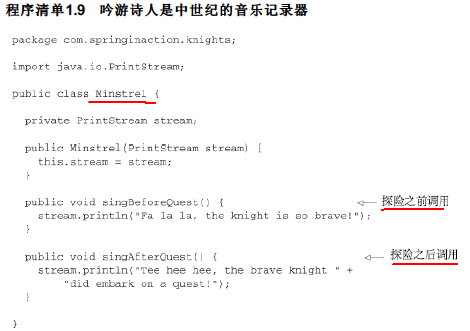
如图1.3所示，可把切面想象为覆盖在很多组件之上的一个外壳。应用由那些实现各自业务功能的模块组成。

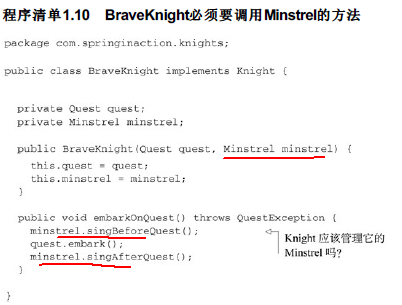
借助AOP，可使用各功能层去包裹核心业务层。这些层以声明方式灵活应用到系统中，核心应用甚至根本不知道它们的存在。可将安全、事务和日志关注点与核心业务逻辑相分离。



* AOP应用

每个人都熟知骑士所做的事情，因吟游诗人用诗歌记载了骑士的事迹并将其进行传唱。假设需要使用吟游诗人这个服务类来记载骑士的所有事迹。（未使用AOP的代码）





BraveKnight类变得复杂，如还需应对没有吟游诗人的场景，代码更复杂。

但利用AOP，可声明吟游诗人必须歌颂骑士的探险事迹，而骑士本身并不用直接访问Minstrel的方法。

要将Minstrel抽象为一个切面，需在Spring配置文件中声明它。

更新后的knights.xml文件，Minstrel被声明为一个切面。



用Spring的aop配置命名空间把Minstrel bean声明为一个切面。

首先，需把Minstrel声明为一个bean，然后在<aop:aspect>元素中引用该bean。

前置通知（before advice）。后置通知（after advice）。

pointcut-ref属性都引用了embank的切入点。该切入点是在前边的<pointcut>元素中定义的，并配置expression属性来选择所应用的通知。表达式语法采用AspectJ的切点表达式语言。

通过少量的XML配置，就可把Minstrel声明为一个Spring切面。

最重要的，Minstrel可被应用到BraveKnight中，而BraveKnight不需显式地调用它。实际上，BraveKnight完全不知道Minstrel的存在。

Spring简化Java开发的其他方式。

**1.1.4　使用模板消除样板式代码**

* 当编写的时候总会感觉以前曾经这么写过？是样板式的代码（boilerplate code）。

很多是因为使用Java API而导致的样板式代码。

常见范例是使用JDBC访问数据库查询数据。

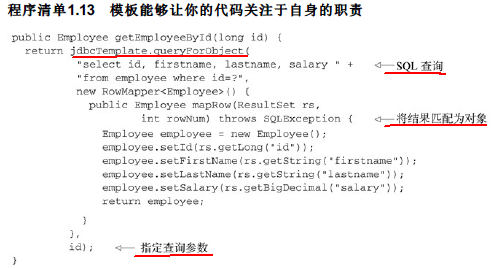


JMS、JNDI和使用REST服务通常也涉及大量的重复代码。

Spring旨在通过模板封装来消除样板式代码。

Spring的JdbcTemplate使得执行数据库操作时，避免传统的JDBC样板代码成为了可能。

例如，使用Spring的JdbcTemplate重写的getEmployeeById()方法仅关注于获取员工数据的核心逻辑，而不需迎合JDBC API的需求。



新版本的getEmployeeById()仅关注于从数据库中查询员工。

模板的queryForObject()方法需一个SQL查询语句，一个RowMapper对象（把数据映射为一个域对象），零或多个查询参数。

以前的JDBC样板式代码全部被封装到了模板中。

文件中如何配置bean和切面，但这些文件如何加载？被加载到哪里去了？

再了解下Spring容器，是应用中的所有bean驻留的地方。

**1.2　容纳Bean**

* 应用对象生存于Spring容器（container）中。

Spring容器负责创建，装配，配置并管理对象的整个生命周期(从生存到死亡:可能就是new到finalize()）。

* 容器是Spring框架的核心。Spring容器用DI管理构成应用的组件，会创建相互协作的组件间的关联。这些对象更简单干净，更易于理解，更易于重用且更易于进行单元测试。

Spring自带了多个容器实现，可归为两种不同类型:

* bean工厂（由org.springframework.beans.factory.beanFactory接口定义）是最简单的容器，提供基本的DI支持
* 应用上下文（由org.springframework.context.ApplicationContext接口定义）基于BeanFactory构建，并提供应用框架级别的服务，例如从属性文件解析文本信息及发布应用事件给感兴趣的事件监听者。

应用上下文要比bean工厂更受欢迎。

**1.2.1　使用应用上下文**

Spring自带多种应用上下文:

|  |
| --- |
| AnnotationConfigApplicationContext：从一或多个基于Java的配置类中加载Spring应用上下文。  AnnotationConfigWebApplicationContext：从一或多个基于Java的配置类中加载Spring Web应用上下文。  ClassPathXmlApplicationContext：从类路径下一或多个XML配置文件中加载上下文定义，把应用上下文的定义文件作为类资源。  FileSystemXmlapplicationcontext：从文件系统下的一个或多个XML配置文件中加载上下文定义。  XmlWebApplicationContext：从Web应用下的一个或多个XML配置文件中加载上下文定义。 |

现在先简单地用FileSystemXmlApplicationContext从文件系统中或使用ClassPathXml

ApplicationContext从类路径中加载应用上下文。将bean加载到bean工厂的过程相似。





如果想从Java配置中加载应用上下文：



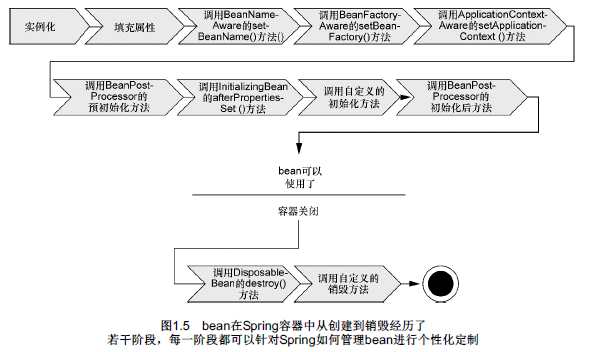
应用上下文准备就绪后，就可调用上下文的getBean()方法从Spring容器中获取bean。

**1.2.2　bean的生命周期**

传统Java应用中，用new进行bean实例化，bean就可使用了。

一旦该bean不再使用，由Java自动进行垃圾回收。

图1.5展示了bean装载到Spring应用上下文中的一个典型的生命周期过程。



|  |
| --- |
| 1．Spring对bean进行实例化；  2．Spring将值和bean的引用注入到bean对应的属性中；  3．如bean实现了BeanNameAware接口，Spring将bean的ID传递给setBean-Name()方法；  4．如bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()方法，将BeanFactory容器实例传入；  5．如果bean实现了ApplicationContextAware接口，Spring将调用setApplicationContext()方法，将bean所在的应用上下文的引用传入进来；  6．如bean实现了BeanPostProcessor接口，Spring将调用它们的post-ProcessBeforeInitialization()方法；  7．如果bean实现了InitializingBean接口，Spring将调用它们的after-PropertiesSet()方法。类似地，如果bean使用initmethod声明了初始化方法，该方法也会被调用；  8．如bean实现了BeanPostProcessor接口，Spring将调用post-ProcessAfterInitialization()．此时，bean已经准备就绪，可被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁；  10．如果bean实现了DisposableBean接口，Spring将调用它的destroy()接口方法。同样，如果bean使用destroy-method声明了销毁方法，该方法也会被调用。 |

为了从Spring的DI中受益，须将应用对象装配进Spring容器中。

现在首先浏览一下Spring的体系结构，了解一下Spring框架的基本组成部分和最新版本的Spring所发布的新特性。

**1.3　俯瞰Spring风景线**

Spring关注于通过DI、AOP和消除样板式代码来简化企业级Java开发。

即使这是Spring所能做的全部事情，那Spring也值得一用。

但是Spring实际上的功能超乎你的想象。

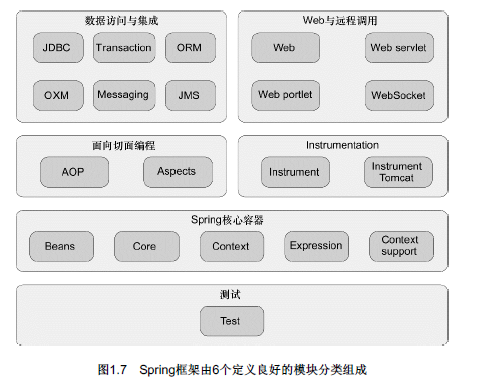
在Spring框架的范畴内，你会发现Spring简化Java开发的多种方式。但在Spring框架之外还存在一个构建在核心框架之上的庞大生态圈，它将Spring扩展到不同的领域，例如Web服务、REST、移动开发以及NoSQL。

**1.3.1　Spring模块**

20个不同的模块，每个模块有3个JAR文件（二进制类库、源码的JAR文件以及JavaDoc的JAR文件）。



模块依据其所属功能可以划分为6类不同的功能，如图1.7所示。



可自由地选择适合自身应用需求的Spring模块；

当Spring不能满足需求时，可考虑其他选择。

事实上，Spring甚至提供了与其他第三方框架和类库的集成点，这样

你就不需要自己编写这样的代码了。

* **Spring核心容器**

管理着Spring应用中bean的创建、配置和管理。

|  |
| --- |
| 该模块包括了Spring bean工厂，为Spring提供了DI功能。  还有多种Spring应用上下文的实现，每一种都提供了配置Spring的不同方式。  企业服务，例如E-mail、JNDI访问、EJB集成和调度。 |

所有Spring模块都构建于核心容器之上。

当配置应用时，其实隐式地使用了这些类。

* **Spring的AOP模块**

对面向切面编程提供了丰富支持。

开发切面的基础。

与DI一样，AOP可帮应用对象解耦。

借助于AOP，可将遍布系统的关注点（例如事务和安全）从它们所应用的对象中解耦出来。

* **数据访问与集成**
* 使用JDBC编写代码通常会导致大量的样板式代码，例如获得数据库连接、创建语句、处理结果集到最后关闭数据库连接。
* Spring的JDBC和DAO（Data Access Object）模块抽象了这些样板式代码，使数据库代码变得简单，还可避免因为关闭数据库资源失败而引发的问题。
* 该模块在多种数据库服务的错误信息之上构建了一个语义丰富的异常层，不需要解释那些隐晦专有的SQL错误信息！
* ORM模块.建立在对DAO的支持之上，并为多个ORM框架提供了一种构建DAO的简便方式。Spring没创建自己的ORM解决方案，而是对流行的ORM框架进行了集成，包括Hibernate、Java Persisternce API、Java Data Object和iBATIS SQL Maps。Spring的事务管理支持所有的ORM框架以及JDBC。
* 还包含了在JMS（Java Message Service）之上构建的Spring抽象层，会使用消息以异步的方式与其他应用集成。
* 从Spring 3.0开始，还包含对象到XML映射的特性，最初是Spring Web Service项目的一部分。
* 会使用Spring AOP模块为Spring应用中的对象提供事务管理服务。
* **Web与远程调用**

MVC（Model-View-Controller）模式:构建Web应用，将界面与应用逻辑分离。

Apache的Struts、JSF、WebWork和Tapestry都是MVC框架。

虽Spring能与多种流行MVC框架集成，但它的Web和远程调用模块自带了一个强大的MVC框架，有助于在Web层提升应用的松耦合水平。在第5章到第7章

除面向用户的Web应用，该模块还提供了多种构建与其他应用交互的远程调用方案。集成了RMI（Remote MethodInvocation）、Hessian、Burlap、JAX-WS，同时Spring还自带了一个远程调用框架：HTTP invoker。Spring还提供了暴露和使用REST API的良好支持。

第15章讨论Spring的远程调用功能。第16章学习如何创建和使用REST API。

* **Instrumentation**

为JVM添加代理（agent）。为Tomcat提供了一个织入代理，能为Tomcat传递类文件，就像这些文件是被类加载器加载的一样。Instrumentation使用场景非常有限，在本书中，不会介绍。

* **测试**

Spring为使用JNDI、Servlet和Portlet编写单元测试提供了一系列的mock对象实现。

对于集成测试，该模块为加载Spring应用上下文中的bean集合以及与Spring上下文中的bean进行交互提供了支持。

* + 1. **Spring Portfolio(用不到 没看)**

**1.4　Spring的新功能(只需了解 没看)**

第2章　装配Bean

* 本章内容：

|  |
| --- |
| 声明bean  构造器注入和Setter方法注入  装配bean  控制bean的创建和销毁 |

创建应用对象间关联关系的传统方法（通过构造器或查找）常会导致代码结构复杂，难被复用也很难进行单元测试。最坏的情况是，对象彼此间高度耦合，难以复用和测试。

Spring中，对象无需自己查找或创建与其所关联的其他对象。容器负责把需相互协作的对象引用赋予各个对象。

创建应用对象间协作关系的行为称装配（wiring），也是依赖注入（DI）的本质。

Spring中装配bean有多种方式。本章先介绍配置Spring容器最常见的三种方法。

**2.1　Spring配置的可选方案**

Spring容器负责创建应用程序中的bean并通过DI来协调对象间的关系。

开发人员需告诉Spring要创建哪些bean且如何将其装配在一起。

* Spring提供了三种主要的装配机制：

|  |
| --- |
| 在XML中进行显式配置。(3)  在Java中进行显式配置。(2)  隐式的bean发现机制和自动装配。(1) |

很多场景下，选择哪种方案很大程度上就是个人喜好问题

Spring的配置风格可互相搭配

建议尽可能使用自动配置机制。显式配置越少越好。

当必须显式配置bean时（比如，有些源码不是由你来维护的，而当你需要为这些代码配置bean的时候），推荐用类型安全且比XML更加强大的JavaConfig。

只有当想使用便利的XML命名空间，且在JavaConfig中没有同样的实现时，才应使用XML。

**2.2　自动化装配bean**

便利性方面，最强大的还是自动化配置。

Spring从两个角度实现自动化装配：

* 组件扫描（component scanning）：自动发现应用上下文中所创建的bean。
* 自动装配（autowiring）：自动满足bean之间的依赖。

首先，要创建CompactDisc类，Spring会发现它并将其创建为一个bean。

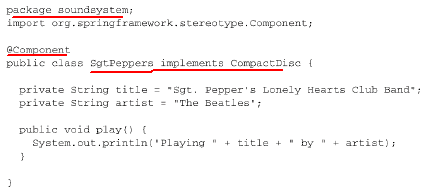
然后，会创建CDPlayer类，让Spring发现它，并将CompactDiscbean注入进来。

**2.2.1　创建可被发现的bean**

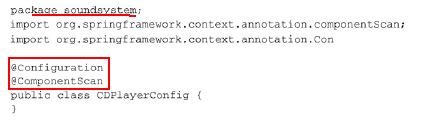
如不将CD插入（注入）到CD播放器中，CD播放器其实没有太大用处。所以， CD播放器依赖于CD才能完成它的使命。



作为接口，定义了CD播放器对一盘CD所能进行的操作。将CD播放器的任意实现与CD本身的耦合降低到最小程度。



|  |
| --- |
| * @Component注解表明该类会作为组件类，并告知Spring要为这个类创建bean。因此没有必要显式配置SgtPeppersbean. * 不过，组件扫描默认不启用。还需显式配置Spring，命令它去寻找带有@Component注解的类，并为其创建bean。 |



* 类CDPlayerConfig通过Java代码定义了Spring的装配规则。

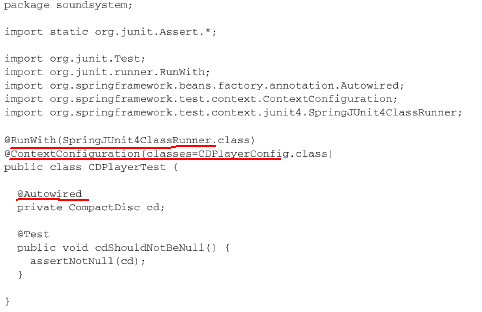
CDPlayerConfig类没有显式地声明任何bean，但使用了@ComponentScan注解，能在Spring中启用组件扫描。

如没有其他配置，@ComponentScan默认扫描与配置类相同的包。即Spring将会扫描soundsystem包及这个包下的所有子包，查找带有@Component注解的类。就能发现CompactDisc，且会在Spring中自动为其创建bean。

* XML配置：<context:component-scan> 通过XML启用组件扫描



为测试组件扫描功能，创建JUnit测试，会创建Spring上下文，并判断CompactDisc是不是真的创建出来了。



SpringJUnit4ClassRunner: 测试开始时自动创建Spring的应用上下文。

@ContextConfiguration: 告知需在CDPlayerConfig中加载配置。

因CDPlayerConfig类中包含@ComponentScan，应用上下文中应包含CompactDisc bean。

为证明，测试代码中有一个CompactDisc类型的属性，带有@Autowired注解，以便于将CompactDisc bean注入到测试代码中。

测试方法断言cd属性不为null。如它不为null，就意味着Spring能发现CompactDisc类，自动在Spring上下文中将其创建为bean并将其注入到测试代码之中。

在soundsystem包及其子包中，所有带有@Component注解的类都会创建为bean。

**2.2.2　为组件扫描的bean命名**

Spring应用上下文中所有bean都会给定一个ID。

没设置ID，Spring会根据类名为其指定一个ID。为sgtPeppers，是将类名的第一个字母变为小写。

如想为bean设置ID，需将期望ID作为值传递给@Component注解。

* 如想bean标识为lonelyHeartsClub：



* 另一种方式，使用@Named注解：

Spring支持将@Named作为@Component注解的替代方案。有细微差异，但大多数场景可互相替换。(不推荐)

**2.2.3　设置组件扫描的基础包**

* 为指定不同的基础包，需在@ComponentScan的value属性中指明包名：



* 如想更清晰地表明设置的是基础包，可通过basePackages属性进行配置：



基础包以String类型表示。但类型不安全（not type-safe）。

如重构代码，所指定的基础包可能就会出错。

* @ComponentScan还提供了另一种方法，将其指定为包中所包含的类或接口



basePackages被替换成basePackageClasses。

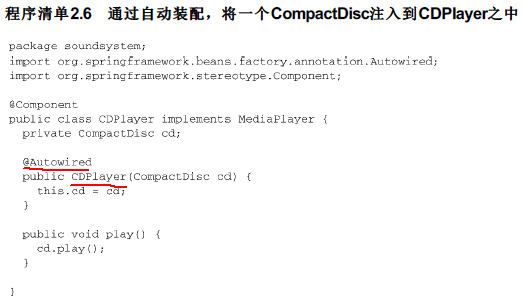
为basePackageClasses属性所设置的数组中包含了类。类所在的包将会作为组件扫描的基础包。

需能够将组件扫描得到的bean和它们的依赖装配在一起。

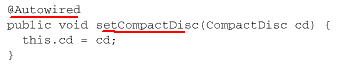
**2.2.4　通过为bean添加注解实现自动装配**

* 自动装配:让Spring自动满足bean依赖的方法

满足依赖过程中，会在Spring应用上下文中寻找匹配某个bean需求的其他bean。可借助Spring的@Autowired注解。

* 构造器上添加了@Autowired注解，表明当Spring创建CDPlayer bean的时候，会通过这个构造器来实例化且会传入一个可设置给CompactDisc类型的bean 
* @Autowired还能用在属性的Setter方法上。

比如说，如CDPlayer有setCompactDisc()方法

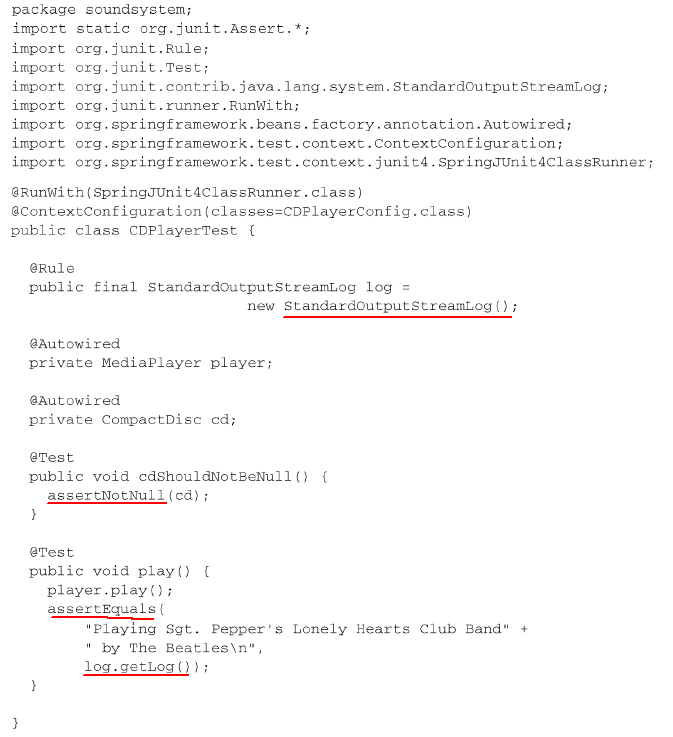


* @Autowired可用在类的任何方法上。

|  |
| --- |
| * 假如有且只有一个bean匹配依赖需求，该bean会被装配进来。 * 如没有匹配的bean，在应用上下文创建时，Spring会抛出一个异常。 * 为避免异常，可将@Autowired的required属性设置为false：   如没有匹配的bean的话，Spring将会让这个bean处于未装配状态。  但如代码中没进行null检查，可能会出现NullPointerException。   * 如有多个bean能满足依赖关系，Spring将会抛出异常，表明没明确指定要选择哪个bean进行自动装配。 * @Autowired是Spring特有的注解。可用@Inject替换，来源于Java依赖注入规范 |

**2.2.5　验证自动装配**

为验证这，修改一下CDPlayerTest，使其能够借助CDPlayer bean播放CD：



在play()测试方法中,可调用CDPlayer的play()方法，并断言它的行为与预期一致。

使用了StandardOutputStreamLog，该规则能基于控制台的输出编写断言。

在这里，我们断言SgtPeppers.play()方法的输出被发送到了控制台上。

但现在，先将组件扫描和自动装配放一边，看一下在Spring中如何显式地装配bean，首先从通过Java代码编写配置开始。

**2.3　通过Java代码装配bean(**显式**)**

* 有时自动化配置行不通，需明确配置Spring。

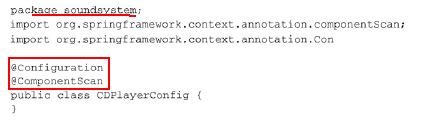
比如，想将第三方库中的组件装配到你的应用中，是没办法在其类上添加注解的，就不能用自动化装配方案。此时必须要采用显式装配的方式。（Java和XML）

显式配置时，JavaConfig更好，因它更强大、类型安全且对重构友好。因它就是Java代码，就像应用程序中的其他Java代码一样。

JavaConfig是配置代码。不应包含业务逻辑，也不应侵入到业务逻辑代码中。

**2.3.1　创建配置类**

重温样例中的CDPlayerConfig：

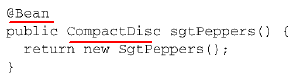


@Configuration表明该类是配置类，应包含在Spring应用上下文中如何创建bean的细节。

如移除了@ComponentScan注解，此时CDPlayerConfig类就没有任何作用。

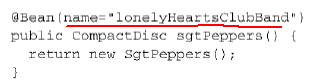
**2.3.2　声明简单的bean**

* 声明bean，需编写一个方法，会创建所需类型的实例，然后给这个方法添加@Bean注解。比方说，下面代码声明了CompactDisc bean：



@Bean会告诉Spring, 该方法将会返回一个对象，该对象要注册为Spring应用上下文中的bean。

* 默认bean的ID与带有@Bean注解的方法名一样。本例中，bean名将会是sgtPeppers。
* 如想为其设置成不同名字

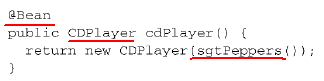


只要最终生成一个CompactDisc实例即可。

**2.3.3　借助JavaConfig实现注入**

前面声明的CompactDisc bean非常简单，自身没有其他依赖。

现在需声明CDPlayer bean，依赖于CompactDisc。

* 在JavaConfig中装配bean的最简单方式就是引用创建bean的方法。

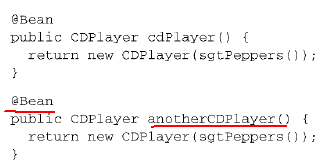
cdPlayer()的方法体与sgtPeppers()稍有区别。调用了需要传入CompactDisc对

象的构造器来创建CDPlayer实例。

看起来，CompactDisc通过调用sgtPeppers()得到，但情况并非完全如此。

因为sgtPeppers()方法上添加了@Bean注解，Spring将会拦截所有对它的调用，并确保直接返回该方法所创建的bean，而不是每次都对其进行实际的调用。

比如说，假设引入了一个其他的CDPlayer bean，和之前的那个bean完全一样：

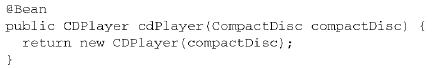


可将同一个SgtPeppers实例注入到任意数量的其他bean之中。

默认，Spring中的bean是单例，没必要为第二个CDPlayer bean创建完全相同的SgtPeppers实例。所以，Spring会拦截对sgtPeppers()的调用并确保返回的是Spring所创建的bean，也就是Spring本身在调用sgtPeppers()时所创建的CompactDiscbean。因此，两个CDPlayer bean会得到相同的SgtPeppers实例。

* 可看到，通过调用方法来引用bean的方式有点令人困惑。其实还有一种理解起来更为简单的方式：

方法1：(用CDPlayer的构造器实现了DI功能)

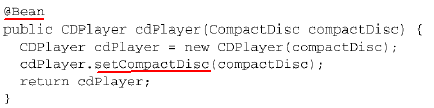


这里，cdPlayer()方法请求一个CompactDisc作为参数。

当Spring调用cdPlayer()创建CDPlayerbean时，会自动装配一个CompactDisc到配置方法之中。借助这种技术，cdPlayer()方法也能够将CompactDisc注入到CDPlayer的构造器中，而且不用明确引用CompactDisc的@Bean方法。

通过这种方式引用其他bean常是最佳选择，因它不会要求将CompactDisc声明到同一个配置类之中。可通过组件扫描功能自动发现或通过XML来进行配置。。不管CompactDisc用什么方式创建，Spring都会将其传入到配置方法中，并用来创建CDPlayer bean。

方法2: Setter方法注入CompactDisc：



**2.4　通过XML装配bean**

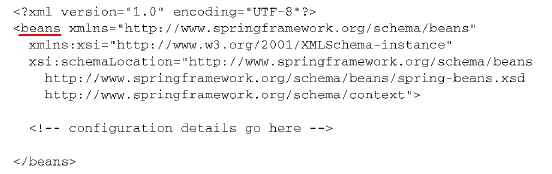
已看到如何让Spring自动发现和装配bean，如何进行手动干预，即通过JavaConfig显式地装配bean。

**2.4.1　创建XML配置规范**

使用JavaConfig时，要创建一个带有@Configuration注解的类

* XML配置中，要创建XML文件，以<beans>元素为根命名空间。

最为简单的Spring XML配置如下：



在配置文件顶部声明XML模式（XSD）文件，定义了配置Spring的XML元素。

**2.4.2　声明一个简单的<bean>**

<bean>类似于JavaConfig中的@Bean注解。

* 可按如下的方式声明CompactDisc bean：（唯一）



要使用全限定的类名。

本例中，bean的ID将根据全限定类名来命名，将会是“soundsystem.SgtPeppers#0”。“#0”是计数形式，用来区分相同类型的其他bean。

如声明了另一个SgtPeppers，且没明确标识，ID将会“soundsystem.SgtPeppers#1”。

为减少XML中繁琐配置，只对需按名字引用的bean（比如需将对它的引用注入到另一个bean中）进行明确命名。

当Spring发现<bean>元素，将会调用SgtPeppers的默认构造器来创建bean。

JavaConfig优于XML配置的原因：

|  |
| --- |
| * 在XML配置中，bean的创建显得更加被动，不过并没JavaConfig那样强大，在JavaConfig配置方式中，可通过任何可以想象到的方法来创建bean实例。 * Spring的XML配置并不能从编译期的类型检查中受益。 |

**2.4.3　借助构造器注入初始化bean**

在XML中声明DI时，有多种可选的配置方案和风格。

* 构造器注入，有两种：

|  |
| --- |
| <constructor-arg>元素  使用Spring 3.0所引入的c-命名空间 |

**区别：**

|  |
| --- |
| <constructor-arg>比c-命名空间更冗长，导致XML更难读懂。  另外，有些事情<constructor-arg>可做到，但c-命名空间却无法实现。 |

* **构造器注入bean引用**
* 现已声明SgtPeppers bean，且SgtPeppers类实现了CompactDisc接口

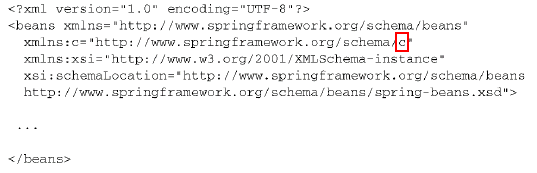
需在XML中声明CDPlayer并通过ID引用SgtPeppers：

当Spring遇到这个<bean>元素，会创建一个CDPlayer实例。

<constructor-arg>元素会告知Spring要将一个ID为compactDisc的bean引用传递到CDPlayer的构造器中。

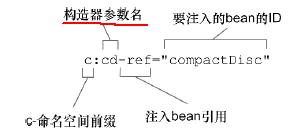
* c-命名空间：

1. 须在XML顶部声明其模式：



1. 之后就可用它来声明构造器参数了：





“-ref”，命名约定，会告诉Spring，正在装配的是一个bean的引用，这个bean的名字是compactDisc，而不是字面量“compactDisc”。

1. 替代方案：使用参数在整个参数列表中的位置信息：

(1)

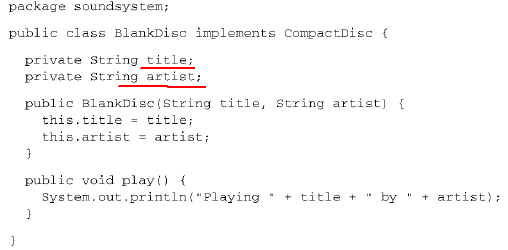


将参数名替换成了“0”，就是参数索引。因在XML中不允许数字作为属性的第一个字符，因此必须要添加一个下画线作为前缀。

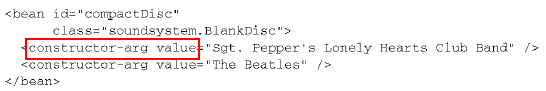
(2) 当只有一个属性需要注入的



* **将字面量注入到构造器中**

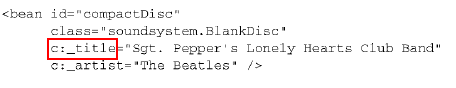
：

* <constructor-arg>

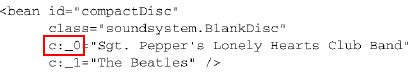


* c-命名空间的话

(1) 引用构造器参数的名字, 属性名中去掉了“-ref”后缀



(2)也可通过参数索引装配相同的字面量值：去掉了“-ref”

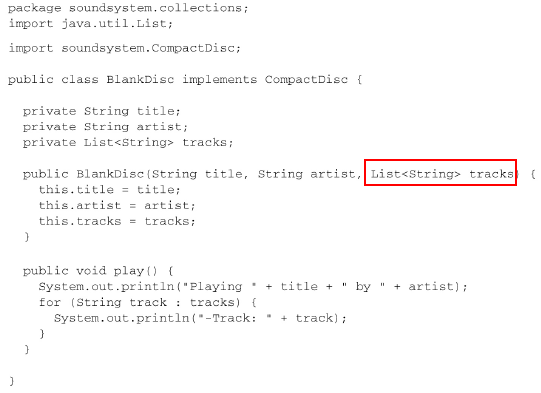


(2-1)如只有一个构造器参数的话。

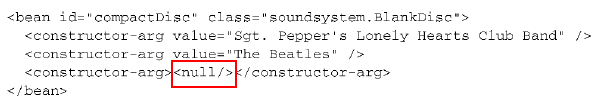


* 有一种情况<constructor-arg>能够实现，c-命名空间无法做到:装配集合

如使用CompactDisc为真正的CD建模，它应该有磁道列表概念。新的BlankDisc：



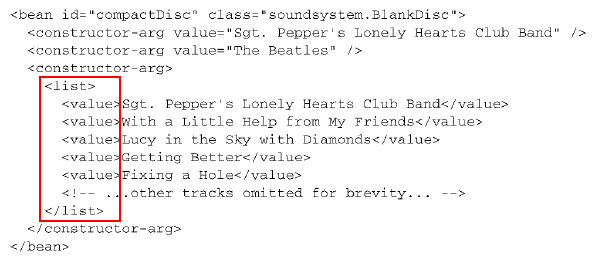
声明bean时，必须提供一个磁道列表。最简单的办法是将列表设置为null：



将null传递给构造器。并不是解决问题的好办法，但在注入期能正常执行。

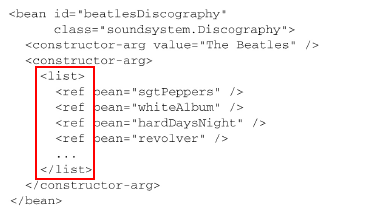
但当调用play()方法时，会遇到NullPointerException。

(1)方案1：可使用<list>元素将其声明为一个列表：



(2)也可使用<ref>元素替代<value>，实现bean引用列表的装配。





可按同样方式使用<set>元素

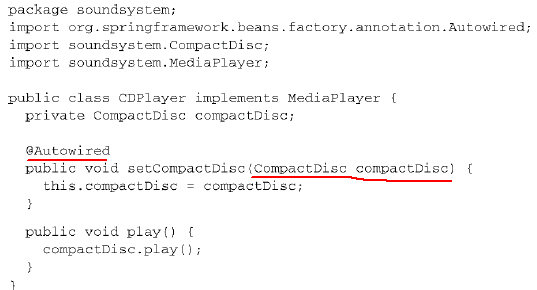
如是Set，重复值会被忽略掉，存放顺序也不会得以保证。(无序、不可重复)

目前，使用c-命名空间的属性无法实现装配集合的功能。

**2.4.4　设置属性（属性注入：setter）**

目前CDPlayer和BlankDisc类通过构造器注入，没用属性的Setter方法。

* + - 1. 如何用SpringXML实现属性注入。CDPlayer如下：



倾向于对强依赖用构造器注入，对可选性的依赖使用属性注入。

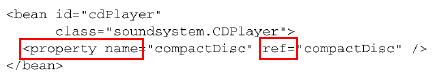
对BlankDisc，唱片名称、艺术家及磁道列表是强依赖，构造器注入

* CDPlayer没有构造器（除了默认构造器），也没强依赖。可采用：(错误)



Spring创建bean时不会有任何问题，但CDPlayerTest会因出现NullPointerException而导致测试失败，因并没注入CDPlayer的compactDisc属性：

正确：

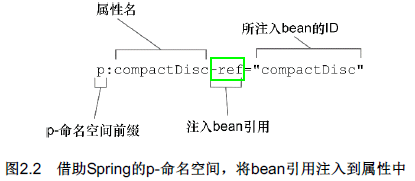


引用了ID为compactDisc的bean（通过ref属性），并将其注入到compactDisc属性中（通过setCompactDisc()方法）。

* + - 1. p-命名空间，<property>的替代方案。
         1. 装配引用

p-命名空间，按以下方式装配compactDisc属性：

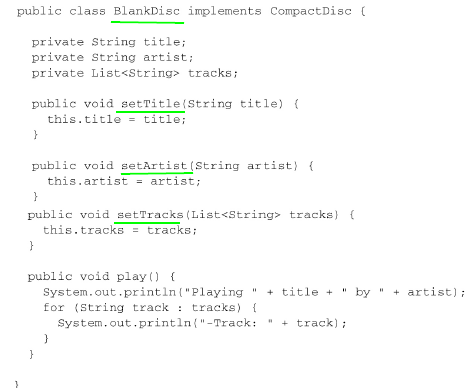




属性名以“-ref”结尾，会提示Spring要进行装配的是引用，而不是字面量。

* + - 1. 将**字面量**注入到**属性**中

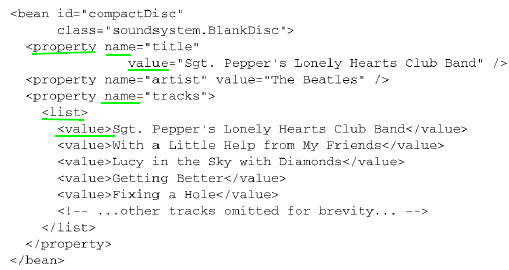
(1)属性也可注入字面量，与构造器参数类似。BlankDisc这次完全通过属性注入进行配置，而不是构造器注入：



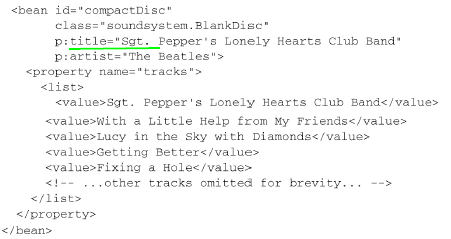
现在，不再强制要求装配任何属性。可按如下方式创建一个BlankDiscbean，所有属性都是空：



使用<property>元素的value属性来设置title和artist，还使用了内嵌的<list>元素来设置tracks属性:



(2)另一种可选方案是使用p-命名空间：

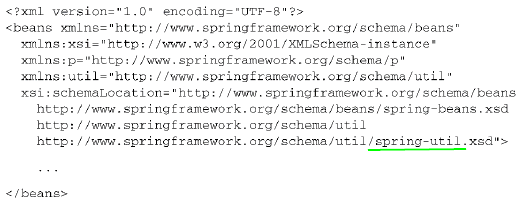


与c-命名空间一样，装配bean引用与字面量的唯一区别在于是否带有“-ref”后缀。如没有“-ref”后缀，装配的是字面量。

(3)但不能用p-命名空间来装配集合.

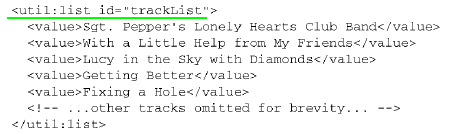
但可用Spring util-命名空间中的功能简化BlankDiscbean。

* 首先需在XML中声明util-命名空间及其模式：

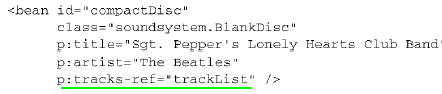


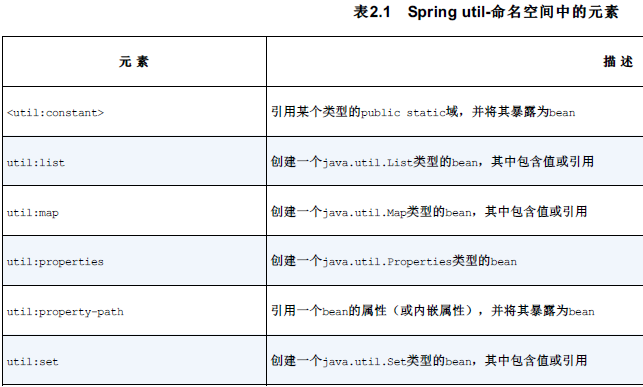
* <util:list>元素会创建一个列表的bean。

借助<util:list>，可将磁道列表转移到BlankDisc bean之外，并将其声明到单独的bean之中，如下所示：



现在就能像使用其他bean那样，将磁道列表bean注入到BlankDisc bean的tracks属性中：





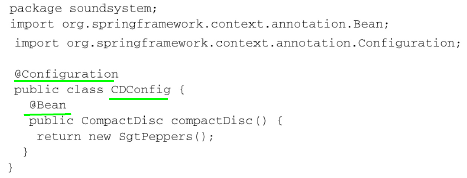
**2.5　导入和混合配置**

可能会同时用自动化和显式配置(JavaConfig的组件扫描)。但有时XML却是最佳方案。都不互斥。尽可以混合在一起。至少需要有一点显式配置来启用组件扫描和自动装配。

在自动装配时，并不在意要装配的bean来自哪里。会考虑Spring容器中所有bean，

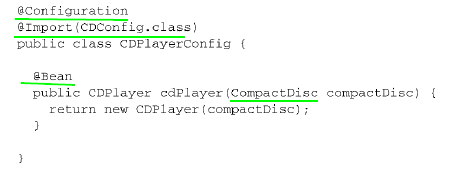
**2.5.1　在JavaConfig中引用XML配置**

现在临时假设CDPlayerConfig已经变得有些笨重，想要将其拆分。

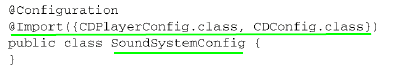
方案一:将BlankDisc从CDPlayerConfig拆分，定义到它自己的CDConfig类中:

compactDisc()已从CDPlayerConfig中移除，需有一种方式将这两个类组合在一起。

* 一种方法是在CDPlayerConfig中用@Import注解导入CDConfig：

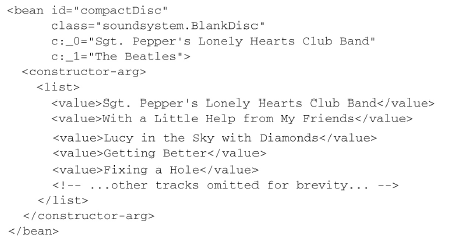


* 或者采用更好办法，不在CDPlayerConfig中用@Import，而是创建一个更高级别的SoundSystemConfig类,使用@Import将两配置类组合在一起：

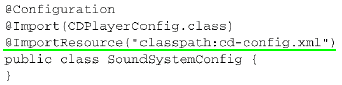


都将CDPlayer配置与BlankDisc配置分开。

现假设希望通过XML来配置BlankDisc：



该如何让Spring同时加载它和其他基于Java的配置呢？用@ImportResource注解。假设BlankDisc定义在名为cd-config.xml的文件中，位于根类路径下，那么可以修改SoundSystemConfig，让它使用@ImportResource注解，如下所示：



配置在JavaConfig中的CDPlayer及配置在XML中BlankDisc都会被加载到Spring容器中。

因CDPlayer中带有@Bean注解的方法接受一个CompactDisc作为参数，BlankDisc将会装配进来，此时与它是通过XML配置的没有任何关系。

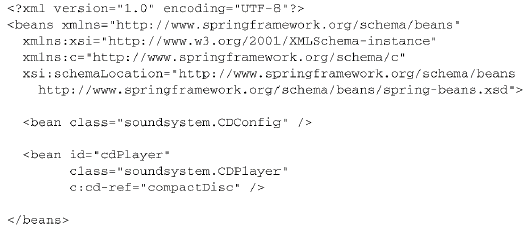
**2.5.2　在XML配置中引用JavaConfig**

假设正用Spring基于XML的配置且已意识到XML逐渐变得无法控制。需拆分。

XML中可用import元素来拆分XML配置。

假如希望将BlankDisc bean拆分到自己的配置文件cd-config.xml中，这与之前用@ImportResource一样。

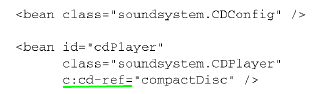
可在XML配置文件中使用<import>元素来引用该文件：



现在假设要把BlankDisc配置在JavaConfig中，CDPlayer则配置在XML中。

<import>只能导入其他XML配置文件，并没有XML元素能导入JavaConfig类。

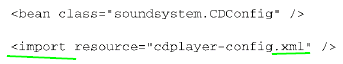
为将JavaConfig类导入到XML配置中，可这样声明bean：



两种配置: 一个用XML描述，另一个使用Java描述—被组合在一起。

类似地，可能还希望创建一个更高层次的配置文件，不声明任何bean，只负责将两个或更多的配置组合起来。

例如，可将CDConfig bean从之前XML文件中移除，而使用第三个配置文件将这两组合在一起：



不管用JavaConfig还是XML装配，通常都会创建一个根配置（root configuration），会将两个或更多的装配类和/或XML文件组合起来。

**2.6　小结**

Spring框架的核心是Spring容器。容器负责管理应用中组件的生命周期，会创建这些组件并保证它们的依赖能够得到满足，这样的话，组件才能完成预定的任务。

Spring中装配bean的三种主要方式：

自动化配置、基于Java的显式配置以及基于XML的显式配置。

建议尽可能使用自动化配置，以避免显式配置所带来的维护成本。

但如确实需要显式配置Spring，应优选基于Java的配置，比基于XML的配置更加强大、类型安全并且易于重构。

第3章【没看】高级装配

本章内容：

|  |
| --- |
| Spring profile  条件化的bean声明  自动装配与歧义性  bean的作用域  Spring表达式语言 |

在上一章中是最为核心的bean装配技术。在本章中，深入介绍高级技术。不会天天都用到

**3.1　环境与profile**

在开发软件的时候，有一个很大的挑战就是将应用程序从一个环境迁移到另外一个环境。开发阶段中，某些环境相关做法可能并不适合迁移到生产环境中，甚至即便迁移过去也无法正常工作。数据库配置、加密算法以及与外部系统的集成是跨环境部署时会发生变化的几个典型例子。

比如，考虑一下数据库配置。在开发环境中，我们可能会使用嵌入式数据库，并预先加载测试数据。例如，在Spring配置类中，我们可能会在一个带有@Bean注解的方法上使用EmbeddedDatabaseBuilder：这会创建一个类型为javax.sql.DataSource的bean，这个bean是如何创建出来的才是最有意思的。使

用EmbeddedDatabaseBuilder会搭建一个嵌入式的Hypersonic数据库，它的模式（schema）定义在schema.sql中，测试数据则是通过test-data.sql加载的。

当你在开发环境中运行集成测试或者启动应用进行手动测试的时候，这个DataSource是很有用的。每次启动它的时候，都能让数据库处于一个给定的状态。

尽管EmbeddedDatabaseBuilder创建的DataSource非常适于开发环境，但是对于生产环境来说，这会是一个糟糕的选择。在生产环境的配置中，你可能会希望使用JNDI从容器中获取一个DataSource。在这样场景中，如下的@Bean方法会更加合适：

通过JNDI获取DataSource能够让容器决定该如何创建这个DataSource，甚至包括切换为容器管理的连接池。即便如此，JNDI管理的DataSource更加适合于生产环境，对于简单的集成和开发测试环境来说，这会带来不必要的复杂性。

同时，在QA环境中，你可以选择完全不同的DataSource配置，可以配置为Commons DBCP连接池，如下所示：

显然，这里展现的三个版本的dataSource()方法互不相同。虽然它们都会生成一个类型为javax.sql.DataSource的bean，但它们的相似点也仅限于此了。每个方法都使用了完全不同的策略来生成DataSource bean。再次强调的是，这里的讨论并不是如何配置DataSource（我们将会在第10章更详细地讨论这个话题）。看起来很简单的DataSource实际上并不是那么简单。这是一个很好的例子，它表现了在不同的环境中某个bean会有所不同。我们必须要有一种方法来配置DataSource，使其在每种环境下都会选择最为合适的配置。

其中一种方式就是在单独的配置类（或XML文件）中配置每个bean，然后在构建阶段（可能会使用Maven的profiles）确定要将哪一个配置编译到可部署的应用中。这种方式的问题在于要为每种环境重新构建应用。当从开发阶段迁移到QA阶段时，重新构建也许算不上什么大问题。但是，从QA阶段迁移到生产阶段时，重新构建可能会引入bug并且会在QA团队的成员中带来不安的情绪。

值得庆幸的是，Spring所提供的解决方案并不需要重新构建。

**3.1.1　配置profile bean**

Spring为环境相关的bean所提供的解决方案其实与构建时的方案没有太大的差别。当然，在这个过程中需要根据环境决定该创建哪个bean和不创建哪个bean。不过Spring并不是在构建的时候做出这样的决策，而是等到运行时再来确定。这样的结果就是同一个部署单元（可能会是WAR文件）能够适用于所有的环境，没有必要进行重新构建。

在3.1版本中，Spring引入了bean profile的功能。要使用profile，你首先要将所有不同的bean定义整理到一个或多个profile之中，在将应用部署到每个环境时，要确保对应的profile处于激活（active）的状态。

在Java配置中，可以使用@Profile注解指定某个bean属于哪一个profile。例如，在配置类中，嵌入式数据库的DataSource可能会配置成如下所示：

我希望你能够注意的是@Profile注解应用在了类级别上。它会告诉Spring这个配置类中的bean只有在dev profile激活时才会创建。如果devprofile没有激活的话，那么带有@Bean注解的方法都会被忽略掉。

同时，你可能还需要有一个适用于生产环境的配置，如下所示：

在本例中，只有prod profile激活的时候，才会创建对应的bean。

在Spring 3.1中，只能在类级别上使用@Profile注解。不过，从Spring 3.2开始，你也可以在方法级别上使用@Profile注解，与@Bean注解一同使用。这样的话，就能将这两个bean的声明放到同一个配置类之中，如下所示：

程序清单3.1　@Profile注解基于激活的profile实现bean的装配

这里有个问题需要注意，尽管每个DataSource bean都被声明在一个profile中，并且只有当规定的profile激活时，相应的bean才会被创建，但是可能会有其他的bean并没有声明在一个给定的profile范围内。没有指定profile的bean始终都会被创建，与激活哪个profile没有关系。

在XML中配置profile

我们也可以通过<beans>元素的profile属性，在XML中配置profile bean。例如，为了在XML中定义适用于开发阶段的嵌入式数据库DataSourcebean，我们可以创建如下所示的XML文件：

与之类似，我们也可以将profile设置为prod，创建适用于生产环境的从JNDI获取的DataSource bean。同样，可以创建基于连接池定义的DataSource bean，将其放在另外一个XML文件中，并标注为qaprofile。所有的配置文件都会放到部署单元之中（如WAR文件），但是只

有profile属性与当前激活profile相匹配的配置文件才会被用到。

你还可以在根<beans>元素中嵌套定义<beans>元素，而不是为每个环境都创建一个profile XML文件。这能够将所有的profile bean定义放到

同一个XML文件中，如下所示：

程序清单3.2　重复使用元素来指定多个profile

除了所有的bean定义到了同一个XML文件之中，这种配置方式与定义在单独的XML文件中的实际效果是一样的。这里有三个bean，类型都

是javax.sql.DataSource，并且ID都是dataSource。但是在运行时，只会创建一个bean，这取决于处于激活状态的是哪个profile。

那么问题来了：我们该怎样激活某个profile呢？

**3.1.2　激活profile**

Spring在确定哪个profile处于激活状态时，需要依赖两个独立的属性：spring.profiles.active和spring.profiles.default。如

果设置了spring.profiles.active属性的话，那么它的值就会用来确定哪个profile是激活的。但如果没有设

置spring.profiles.active属性的话，那Spring将会查找spring.profiles.default的值。如

果spring.profiles.active和spring.profiles.default均没有设置的话，那就没有激活的profile，因此只会创建那些没有定义在profile中的bean。

有多种方式来设置这两个属性：

作为DispatcherServlet的初始化参数；

作为Web应用的上下文参数；

作为JNDI条目；

作为环境变量；

作为JVM的系统属性；

在集成测试类上，使用@ActiveProfiles注解设置。

你尽可以选择spring.profiles.active和spring.profiles.default的最佳组合方式以满足需求，我将这样的自主权留给读者。

我所喜欢的一种方式是使用DispatcherServlet的参数将spring.profiles.default设置为开发环境的profile，我会在Servlet上下文中进行设置（为了兼顾到ContextLoaderListener）。例如，在Web应用中，设置spring.profiles.default的web.xml文件会如下

所示：

程序清单3.3　在Web应用的web.xml文件中设置默认的profile

按照这种方式设置spring.profiles.default，所有的开发人员都能从版本控制软件中获得应用程序源码，并使用开发环境的设置（如嵌入式数据库）运行代码，而不需要任何额外的配置。

当应用程序部署到QA、生产或其他环境之中时，负责部署的人根据情况使用系统属性、环境变量或JNDI设置spring.profiles.active即可。当设置spring.profiles.active以后，至于spring.profiles.default置成什么值就已经无所谓了；系统会优先使用spring.profiles.active中所设置的profile。

你可能已经注意到了，在spring.profiles.active和spring.profiles.default中，profile使用的都是复数形式。这意味着你可以同时激活多个profile，这可以通过列出多个profile名称，并以逗号分隔来实现。当然，同时启用dev和prod profile可能也没有太大的意义，不过你可以同时设置多个彼此不相关的profile。

使用profile进行测试

当运行集成测试时，通常会希望采用与生产环境（或者是生产环境的部分子集）相同的配置进行测试。但是，如果配置中的bean定义在了profile中，那么在运行测试时，我们就需要有一种方式来启用合适的profile。

Spring提供了@ActiveProfiles注解，我们可以使用它来指定运行测试时要激活哪个profile。在集成测试时，通常想要激活的是开发环境的profile。例如，下面的测试类片段展现了使用@ActiveProfiles激活dev profile：

在条件化创建bean方面，Spring的profile机制是一种很棒的方法，这里的条件要基于哪个profile处于激活状态来判断。Spring 4.0中提供了一种更为通用的机制来实现条件化的bean定义，在这种机制之中，条件完全由你来确定。让我们看一下如何使用Spring 4和@Conditional注解

定义条件化的bean。

**3.2　条件化的bean**

假设你希望一个或多个bean只有在应用的类路径下包含特定的库时才创建。或者我们希望某个bean只有当另外某个特定的bean也声明了之后才会创建。我们还可能要求只有某个特定的环境变量设置之后，才会创建某个bean。

在Spring 4之前，很难实现这种级别的条件化配置，但是Spring 4引入了一个新的@Conditional注解，它可以用到带有@Bean注解的方法上。如果给定的条件计算结果为true，就会创建这个bean，否则的话，这个bean会被忽略。

例如，假设有一个名为MagicBean的类，我们希望只有设置了magic环境属性的时候，Spring才会实例化这个类。如果环境中没有这个属性，那么MagicBean将会被忽略。在程序清单3.4所展现的配置中，使用@Conditional注解条件化地配置了MagicBean。

程序清单3.4　条件化地配置bean可以看到，@Conditional中给定了一个Class，它指明了条件——在本例中，也就是MagicExistsCondition。@Conditional将会

通过Condition接口进行条件对比：

设置给@Conditional的类可以是任意实现了Condition接口的类型。可以看出来，这个接口实现起来很简单直接，只需提供matches()方法的实现即可。如果matches()方法返回true，那么就会创建带有@Conditional注解的bean。如果matches()方法返回false，将不会创建这些bean。

在本例中，我们需要创建Condition的实现并根据环境中是否存在magic属性来做出决策。程序清单3.5展现了MagicExistsCondition，这是完成该功能的Condition实现类：

程序清单3.5　在Condition中检查是否存在magic属性

在上面的程序清单中，matches()方法很简单但功能强大。它通过给定的ConditionContext对象进而得到Environment对象，并使用这个对象检查环境中是否存在名为magic的环境属性。在本例中，属性的值是什么无所谓，只要属性存在即可满足要求。如果满足这个条件的话，matches()方法就会返回true。所带来的结果就是条件能够得到满足，所有@Conditional注解上引用MagicExistsCondition的bean都会被创建。

话说回来，如果这个属性不存在的话，就无法满足条件，matches()方法会返回false，这些bean都不会被创建。

MagicExistsCondition中只是使用了ConditionContext得到的Environment，但Condition实现的考量因素可能会比这更多。matches()方法会得到ConditionContext和AnnotatedTypeMetadata对象用来做出决策。

ConditionContext是一个接口，大致如下所示：

通过ConditionContext，我们可以做到如下几点：

借助getRegistry()返回的BeanDefinitionRegistry检查bean定义；

借助getBeanFactory()返回的ConfigurableListableBeanFactory检查bean是否存在，甚至探查bean的属性；借助getEnvironment()返回的Environment检查环境变量是否存在以及它的值是什么；

读取并探查getResourceLoader()返回的ResourceLoader所加载的资源；

借助getClassLoader()返回的ClassLoader加载并检查类是否存在。

AnnotatedTypeMetadata则能够让我们检查带有@Bean注解的方法上还有什么其他的注解。像ConditionContext一样，AnnotatedTypeMetadata也是一个接口。它如下所示：

借助isAnnotated()方法，我们能够判断带有@Bean注解的方法是不是还有其他特定的注解。借助其他的那些方法，我们能够检查@Bean注解的方法上其他注解的属性。

非常有意思的是，从Spring 4开始，@Profile注解进行了重构，使其基于@Conditional和Condition实现。作为如何使用@Conditional和Condition的例子，我们来看一下在Spring 4中，@Profile是如何实现的。

@Profile注解如下所示：

注意： @Profile本身也使用了@Conditional注解，并且引用ProfileCondition作为Condition实现。如下所示，ProfileCondition实现了Condition接口，并且在做出决策的过程中，考虑到了ConditionContext和AnnotatedTypeMetadata中的多个因素。

程序清单3.6　ProfileCondition检查某个bean profile是否可用

我们可以看到，ProfileCondition通过AnnotatedTypeMetadata得到了用于@Profile注解的所有属性。借助该信息，它会明确地检查value属性，该属性包含了bean的profile名称。然后，它根据通过ConditionContext得到的Environment来检查［借助acceptsProfiles()方法］该profile是否处于激活状态。

**3.3　处理自动装配的歧义性**

在第2章中，我们已经看到如何使用自动装配让Spring完全负责将bean引用注入到构造参数和属性中。自动装配能够提供很大的帮助，因为它

会减少装配应用程序组件时所需要的显式配置的数量。

不过，仅有一个bean匹配所需的结果时，自动装配才是有效的。如果不仅有一个bean能够匹配结果的话，这种歧义性会阻碍Spring自动装配属性、构造器参数或方法参数。

为了阐述自动装配的歧义性，假设我们使用@Autowired注解标注了setDessert()方法：

在本例中，Dessert是一个接口，并且有三个类实现了这个接口，分别为Cake、Cookies和IceCream：

因为这三个实现均使用了@Component注解，在组件扫描的时候，能够发现它们并将其创建为Spring应用上下文里面的bean。然后，当Spring试图自动装配setDessert()中的Dessert参数时，它并没有唯一、无歧义的可选值。在从多种甜点中做出选择时，尽管大多数人并不会有什么困难，但是Spring却无法做出选择。Spring此时别无他法，只好宣告失败并抛出异常。更精确地讲，Spring会抛

出NoUniqueBeanDefinitionException：

当然，使用吃甜点的样例来阐述自动装配在遇到歧义性时所面临的问题多少有些牵强。在实际中，自动装配歧义性的问题其实比你想象中的更为罕见。就算这种歧义性确实是个问题，但更常见的情况是给定的类型只有一个实现类，因此自动装配能够很好地运行。

但是，当确实发生歧义性的时候，Spring提供了多种可选方案来解决这样的问题。你可以将可选bean中的某一个设为首选（primary）的bean，或者使用限定符（qualifier）来帮助Spring将可选的bean的范围缩小到只有一个bean。

**3.3.1　标示首选的bean**

如果你像我一样，喜欢所有类型的甜点，如蛋糕、饼干、冰激凌……它们都很美味。但如果只能在其中选择一种甜点的话，那你最喜欢的是哪一种呢？

在声明bean的时候，通过将其中一个可选的bean设置为首选（primary）bean能够避免自动装配时的歧义性。当遇到歧义性的时候，Spring将会使用首选的bean，而不是其他可选的bean。实际上，你所声明就是“最喜欢”的bean。

假设冰激凌就是你最喜欢的甜点。在Spring中，可以通过@Primary来表达最喜欢的方案。@Primary能够与@Component组合用在组件扫描的bean上，也可以与@Bean组合用在Java配置的bean声明中。比如，下面的代码展现了如何将@Component注解的IceCream bean声明为首选的bean：

或者，如果你通过Java配置显式地声明IceCream，那么@Bean方法应该如下所示：

如果你使用XML配置bean的话，同样可以实现这样的功能。<bean>元素有一个primary属性用来指定首选的bean：

不管你采用什么方式来标示首选bean，效果都是一样的，都是告诉Spring在遇到歧义性的时候要选择首选的bean。

但是，如果你标示了两个或更多的首选bean，那么它就无法正常工作了。比如，假设Cake类如下所示：

现在，有两个首选的Dessert bean：Cake和IceCream。这带来了新的歧义性问题。就像Spring无法从多个可选的bean中做出选择一样，它也无法从多个首选的bean中做出选择。显然，如果不止一个bean被设置成了首选bean，那实际上也就是没有首选bean了。就解决歧义性问题而言，限定符是一种更为强大的机制，下面就将对其进行介绍。

**3.3.2　限定自动装配的bean**

设置首选bean的局限性在于@Primary无法将可选方案的范围限定到唯一一个无歧义性的选项中。它只能标示一个优先的可选方案。当首选bean的数量超过一个时，我们并没有其他的方法进一步缩小可选范围。

与之相反，Spring的限定符能够在所有可选的bean上进行缩小范围的操作，最终能够达到只有一个bean满足所规定的限制条件。如果将所有的限定符都用上后依然存在歧义性，那么你可以继续使用更多的限定符来缩小选择范围。

@Qualifier注解是使用限定符的主要方式。它可以与@Autowired和@Inject协同使用，在注入的时候指定想要注入进去的是哪个bean。

例如，我们想要确保要将IceCream注入到setDessert()之中：

这是使用限定符的最简单的例子。为@Qualifier注解所设置的参数就是想要注入的bean的ID。所有使用@Component注解声明的类都会创

建为bean，并且bean的ID为首字母变为小写的类名。因此，@Qualifier("iceCream")指向的是组件扫描时所创建的bean，并且这个bean是IceCream类的实例。

实际上，还有一点需要补充一下。更准确地讲，@Qualifier("iceCream")所引用的bean要具有String类型的“iceCream”作为限定符。如果没有指定其他的限定符的话，所有的bean都会给定一个默认的限定符，这个限定符与bean的ID相同。因此，框架会将具有“iceCream”限定符的bean注入到setDessert()方法中。这恰巧就是ID为iceCream的bean，它是IceCream类在组件扫描的时候创建的。

基于默认的bean ID作为限定符是非常简单的，但这有可能会引入一些问题。如果你重构了IceCream类，将其重命名为Gelato的话，那此时会发生什么情况呢？如果这样的话，bean的ID和默认的限定符会变为gelato，这就无法匹配setDessert()方法中的限定符。自动装配会失败。

这里的问题在于setDessert()方法上所指定的限定符与要注入的bean的名称是紧耦合的。对类名称的任意改动都会导致限定符失效。创建自定义的限定符

我们可以为bean设置自己的限定符，而不是依赖于将bean ID作为限定符。在这里所需要做的就是在bean声明上添加@Qualifier注解。例如，它可以与@Component组合使用，如下所示：

在这种情况下，cold限定符分配给了IceCreambean。因为它没有耦合类名，因此你可以随意重构IceCream的类名，而不必担心会破坏自动

装配。在注入的地方，只要引用cold限定符就可以了：

值得一提的是，当通过Java配置显式定义bean的时候，@Qualifier也可以与@Bean注解一起使用：

当使用自定义的@Qualifier值时，最佳实践是为bean选择特征性或描述性的术语，而不是使用随意的名字。在本例中，我将IceCream

bean描述为“cold”bean。在注入的时候，可以将这个需求理解为“给我一个凉的甜点”，这其实就是描述的IceCream。类似地，我可以

将Cake描述为“soft”，将Cookie描述为“crispy”。

使用自定义的限定符注解

面向特性的限定符要比基于bean ID的限定符更好一些。但是，如果多个bean都具备相同特性的话，这种做法也会出现问题。例如，如果引入了这个新的Dessert bean，会发生什么情况呢：

不会吧？！现在我们有了两个带有“cold”限定符的甜点。在自动装配Dessert bean的时候，我们再次遇到了歧义性的问题，需要使用更多的限定符来将可选范围限定到只有一个bean。

可能想到的解决方案就是在注入点和bean定义的地方同时再添加另外一个@Qualifier注解。IceCream类大致就会如下所示：

Popsicle类同样也可能再添加另外一个@Qualifier注解：

在注入点中，我们可能会使用这样的方式来将范围缩小到IceCream：

这里只有一个小问题：Java不允许在同一个条目上重复出现相同类型的多个注解。[1]如果你试图这样做的话，编译器会提示错误。在这里，使

用@Qualifier注解并没有办法（至少没有直接的办法）将自动装配的可选bean缩小范围至仅有一个可选的bean。

但是，我们可以创建自定义的限定符注解，借助这样的注解来表达bean所希望限定的特性。这里所需要做的就是创建一个注解，它本身要使用@Qualifier注解来标注。这样我们将不再使用@Qualifier("cold")，而是使用自定义的@Cold注解，该注解的定义如下所示：

同样，你可以创建一个新的@Creamy注解来代替@Qualifier("creamy")：

当你不想用@Qualifier注解的时候，可以类似地创建@Soft、@Crispy和@Fruity。通过在定义时添加@Qualifier注解，它们就具有了@Qualifier注解的特性。它们本身实际上就成为了限定符注解。

现在，我们可以重新看一下IceCream，并为其添加@Cold和@Creamy注解，如下所示：

类似地，Popsicle类可以添加@Cold和@Fruity注解：

最终，在注入点，我们使用必要的限定符注解进行任意组合，从而将可选范围缩小到只有一个bean满足需求。为了得到IceCream

bean，setDessert()方法可以这样使用注解：

通过声明自定义的限定符注解，我们可以同时使用多个限定符，不会再有Java编译器的限制或错误。与此同时，相对于使用原始的@Qualifier并借助String类型来指定限定符，自定义的注解也更为类型安全。

让我们近距离观察一下setDessert()方法以及它的注解，这里并没有在任何地方明确指定要将IceCream自动装配到该方法中。相反，我们使用所需bean的特性来进行指定，即@Cold和@Creamy。因此，setDessert()方法依然能够与特定的Dessert实现保持解耦。任意满足这些特征的bean都是可以的。在当前选择Dessert实现时，恰好如此，IceCream是唯一能够与之匹配的bean。

在本节和前面的节中，我们讨论了几种通过自定义注解扩展Spring的方式。为了创建自定义的条件化注解，我们创建一个新的注解并在这个注解上添加了@Conditional。为了创建自定义的限定符注解，我们创建一个新的注解并在这个注解上添加了@Qualifier。这种技术可以用到很多的Spring注解中，从而能够将它们组合在一起形成特定目标的自定义注解。

现在我们来看一下如何在不同的作用域中声明bean。

**3.4　bean的作用域**

在默认情况下，Spring应用上下文中所有bean都是作为以单例（singleton）的形式创建的。也就是说，不管给定的一个bean被注入到其他bean多少次，每次所注入的都是同一个实例。

在大多数情况下，单例bean是很理想的方案。初始化和垃圾回收对象实例所带来的成本只留给一些小规模任务，在这些任务中，让对象保持无状态并且在应用中反复重用这些对象可能并不合理。

有时候，可能会发现，你所使用的类是易变的（mutable），它们会保持一些状态，因此重用是不安全的。在这种情况下，将class声明为单例 的bean就不是什么好主意了，因为对象会被污染，稍后重用的时候会出现意想不到的问题。

Spring定义了多种作用域，可以基于这些作用域创建bean，包括：

单例（Singleton）：在整个应用中，只创建bean的一个实例。

原型（Prototype）：每次注入或者通过Spring应用上下文获取的时候，都会创建一个新的bean实例。

会话（Session）：在Web应用中，为每个会话创建一个bean实例。

请求（Rquest）：在Web应用中，为每个请求创建一个bean实例。

单例是默认的作用域，但是正如之前所述，对于易变的类型，这并不合适。如果选择其他的作用域，要使用@Scope注解，它可以与@Component或@Bean一起使用。

例如，如果你使用组件扫描来发现和声明bean，那么你可以在bean的类上使用@Scope注解，将其声明为原型bean：

这里，使用ConfigurableBeanFactory类的SCOPE\_PROTOTYPE常量设置了原型作用域。你当然也可以使用@Scope("prototype")，但是使用SCOPE\_PROTOTYPE常量更加安全并且不易出错。

如果你想在Java配置中将Notepad声明为原型bean，那么可以组合使用@Scope和@Bean来指定所需的作用域：

同样，如果你使用XML来配置bean的话，可以使用<bean>元素的scope属性来设置作用域：

不管你使用哪种方式来声明原型作用域，每次注入或从Spring应用上下文中检索该bean的时候，都会创建新的实例。这样所导致的结果就是每次操作都能得到自己的Notepad实例。

**3.4.1　使用会话和请求作用域**

在Web应用中，如果能够实例化在会话和请求范围内共享的bean，那将是非常有价值的事情。例如，在典型的电子商务应用中，可能会有一个bean代表用户的购物车。如果购物车是单例的话，那么将会导致所有的用户都会向同一个购物车中添加商品。另一方面，如果购物车是原型作用域的，那么在应用中某一个地方往购物车中添加商品，在应用的另外一个地方可能就不可用了，因为在这里注入的是另外一个原型作用域的购物车。

就购物车bean来说，会话作用域是最为合适的，因为它与给定的用户关联性最大。要指定会话作用域，我们可以使用@Scope注解，它的使用方式与指定原型作用域是相同的：

这里，我们将value设置成了WebApplicationContext中的SCOPE\_SESSION常量（它的值是session）。这会告诉Spring为Web应用中的每个会话创建一个ShoppingCart。这会创建多个ShoppingCart bean的实例，但是对于给定的会话只会创建一个实例，在当前会话相关的操作中，这个bean实际上相当于单例的。

要注意的是，@Scope同时还有一个proxyMode属性，它被设置成了ScopedProxyMode.INTERFACES。这个属性解决了将会话或请求作用域的bean注入到单例bean中所遇到的问题。在描述proxyMode属性之前，我们先来看一下proxyMode所解决问题的场景。

假设我们要将ShoppingCart bean注入到单例StoreService bean的Setter方法中，如下所示：

因为StoreService是一个单例的bean，会在Spring应用上下文加载的时候创建。当它创建的时候，Spring会试图将ShoppingCart bean注入到setShoppingCart()方法中。但是ShoppingCart bean是会话作用域的，此时并不存在。直到某个用户进入系统，创建了会话之后，才会出现ShoppingCart实例。

另外，系统中将会有多个ShoppingCart实例：每个用户一个。我们并不想让Spring注入某个固定的ShoppingCart实例到StoreService中。我们希望的是当StoreService处理购物车功能时，它所使用的ShoppingCart实例恰好是当前会话所对应的那一个。

Spring并不会将实际的ShoppingCart bean注入到StoreService中，Spring会注入一个到ShoppingCart bean的代理，如图3.1所示。这个代理会暴露与ShoppingCart相同的方法，所以StoreService会认为它就是一个购物车。但是，当StoreService调用ShoppingCart的方法时，代理会对其进行懒解析并将调用委托给会话作用域内真正的ShoppingCart bean。现在，我们带着对这个作用域的理解，讨论一下proxyMode属性。如配置所示，proxyMode属性被设置成了ScopedProxyMode.INTERFACES，这表明这个代理要实现ShoppingCart接口，并将调用委托给实现bean。

如果ShoppingCart是接口而不是类的话，这是可以的（也是最为理想的代理模式）。但如果ShoppingCart是一个具体的类的话，Spring就没有办法创建基于接口的代理了。此时，它必须使用CGLib来生成基于类的代理。所以，如果bean类型是具体类的话，我们必须要将proxyMode属性设置为ScopedProxyMode.TARGET\_CLASS，以此来表明要以生成目标类扩展的方式创建代理。

尽管我主要关注了会话作用域，但是请求作用域的bean会面临相同的装配问题。因此，请求作用域的bean应该也以作用域代理的方式进行注入。

图3.1　作用域代理能够延迟注入请求和会话作用域的bean

**3.4.2　在XML中声明作用域代理**

如果你需要使用XML来声明会话或请求作用域的bean，那么就不能使用@Scope注解及其proxyMode属性了。<bean>元素的scope属性能够设置bean的作用域，但是该怎样指定代理模式呢？

要设置代理模式，我们需要使用Spring aop命名空间的一个新元素：

<aop:scoped-proxy>是与@Scope注解的proxyMode属性功能相同的Spring XML配置元素。它会告诉Spring为bean创建一个作用域代理。默认情况下，它会使用CGLib创建目标类的代理。但是我们也可以将proxy-target-class属性设置为false，进而要求它生成基于接口的代理：

为了使用<aop:scoped-proxy>元素，我们必须在XML配置中声明Spring的aop命名空间：

在第4章中，当我们使用Spring和面向切面编程的时候，会讨论Spring aop命名空间的更多知识。不过，在结束本章的内容之前，我们来看一下Spring高级配置的另外一个可选方案：Spring表达式语言（Spring Expression Language）。

**3.5　运行时值注入**

当讨论依赖注入的时候，我们通常所讨论的是将一个bean引用注入到另一个bean的属性或构造器参数中。它通常来讲指的是将一个对象与另一个对象进行关联。

但是bean装配的另外一个方面指的是将一个值注入到bean的属性或者构造器参数中。我们在第2章中已经进行了很多值装配，如将专辑的名字装配到BlankDisc bean的构造器或title属性中。例如，我们可能按照这样的方式来组装BlankDisc：

尽管这实现了你的需求，也就是为BlankDisc bean设置title和artist，但它在实现的时候是将值硬编码在配置类中的。与之类似，如果使用XML的话，那么值也会是硬编码的：

有时候硬编码是可以的，但有的时候，我们可能会希望避免硬编码值，而是想让这些值在运行时再确定。为了实现这些功能，Spring提供了两种在运行时求值的方式：

属性占位符（Property placeholder）。

Spring表达式语言（SpEL）。

很快你就会发现这两种技术的用法是类似的，不过它们的目的和行为是有所差别的。让我们先看一下属性占位符，在这两者中它较为简单，然后再看一下更为强大的SpEL。

**3.5.1　注入外部的值**

在Spring中，处理外部值的最简单方式就是声明属性源并通过Spring的Environment来检索属性。例如，程序清单3.7展现了一个基本的Spring配置类，它使用外部的属性来装配BlankDisc bean。

程序清单3.7　使用@PropertySource注解和Environment在本例中，@PropertySource引用了类路径中一个名为app.properties的文件。它大致会如下所示：

这个属性文件会加载到Spring的Environment中，稍后可以从这里检索属性。同时，在disc()方法中，会创建一个新的BlankDisc，它的构造器参数是从属性文件中获取的，而这是通过调用getProperty()实现的。

深入学习Spring的Environment

当我们去了解Environment的时候会发现，程序清单3.7所示的getProperty()方法并不是获取属性值的唯一方法，getProperty()方法有四个重载的变种形式：

前两种形式的getProperty()方法都会返回String类型的值。我们已经在程序清单3.7中看到了如何使用第一种getProperty()方法。但是，你可以稍微对@Bean方法进行一下修改，这样在指定属性不存在的时候，会使用一个默认值：

剩下的两种getProperty()方法与前面的两种非常类似，但是它们不会将所有的值都视为String类型。例如，假设你想要获取的值所代表的含义是连接池中所维持的连接数量。如果我们从属性文件中得到的是一个String类型的值，那么在使用之前还需要将其转换为Integer类型。但是，如果使用重载形式的getProperty()的话，就能非常便利地解决这个问题：

Environment还提供了几个与属性相关的方法，如果你在使用getProperty()方法的时候没有指定默认值，并且这个属性没有定义的话，获取到的值是null。如果你希望这个属性必须要定义，那么可以使用getRequiredProperty()方法，如下所示：

在这里，如果disc.title或disc.artist属性没有定义的话，将会抛出IllegalStateException异常。如果想检查一下某个属性是否存在的话，那么可以调用Environment的containsProperty()方法：

最后，如果想将属性解析为类的话，可以使用getPropertyAsClass()方法：除了属性相关的功能以外，Environment还提供了一些方法来检查哪些profile处于激活状态：

String[] getActiveProfiles()：返回激活profile名称的数组；

String[] getDefaultProfiles()：返回默认profile名称的数组；

boolean acceptsProfiles(String... profiles)：如果environment支持给定profile的话，就返回true。

在程序清单3.6中，我们已经看到了如何使用acceptsProfiles()。在那个例子中，Environment是从ConditionContext中获取到的，在bean创建之前，使用acceptsProfiles()方法来确保给定bean所需的profile处于激活状态。通常来讲，我们并不会频繁使用Environment相关的方法，但是知道有这些方法还是有好处的。直接从Environment中检索属性是非常方便的，尤其是在Java配置中装配bean的时候。但是，Spring也提供了通过占位符装配属性的方法，这些占位符的值会来源于一个属性源。

解析属性占位符

Spring一直支持将属性定义到外部的属性的文件中，并使用占位符值将其插入到Spring bean中。在Spring装配中，占位符的形式为使用“${... }”包装的属性名称。作为样例，我们可以在XML中按照如下的方式解析BlankDisc构造器参数：

可以看到，title构造器参数所给定的值是从一个属性中解析得到的，这个属性的名称为disc.title。artist参数装配的是名为disc.artist的属性值。按照这种方式，XML配置没有使用任何硬编码的值，它的值是从配置文件以外的一个源中解析得到的。（我们稍后会讨论这些属性是如何解析的。）

如果我们依赖于组件扫描和自动装配来创建和初始化应用组件的话，那么就没有指定占位符的配置文件或类了。在这种情况下，我们可以使用@Value注解，它的使用方式与@Autowired注解非常相似。比如，在BlankDisc类中，构造器可以改成如下所示：

为了使用占位符，我们必须要配置一个PropertyPlaceholderConfigurer bean或PropertySourcesPlaceholderConfigurer bean。从Spring 3.1开始，推荐使用PropertySourcesPlaceholderConfigurer，因为它能够基于Spring Environment及其属性源来

解析占位符。

如下的@Bean方法在Java中配置了PropertySourcesPlaceholderConfigurer：

如果你想使用XML配置的话，Spring context命名空间中的<context:propertyplaceholder>元素将会为你生成PropertySourcesPlaceholderConfigurer bean：

解析外部属性能够将值的处理推迟到运行时，但是它的关注点在于根据名称解析来自于Spring Environment和属性源的属性。而Spring表达式语言提供了一种更通用的方式在运行时计算所要注入的值。

**3.5.2　使用Spring表达式语言进行装配**

Spring 3引入了Spring表达式语言（Spring Expression Language，SpEL），它能够以一种强大和简洁的方式将值装配到bean属性和构造器参数中，在这个过程中所使用的表达式会在运行时计算得到值。使用SpEL，你可以实现超乎想象的装配效果，这是使用其他的装配技术难以做到的（甚至是不可能的）。

SpEL拥有很多特性，包括：

使用bean的ID来引用bean；

调用方法和访问对象的属性；

对值进行算术、关系和逻辑运算；

正则表达式匹配；

集合操作。

在本书后面的内容中你可以看到，SpEL能够用在依赖注入以外的其他地方。例如，Spring Security支持使用SpEL表达式定义安全限制规则。

另外，如果你在Spring MVC应用中使用Thymeleaf模板作为视图的话，那么这些模板可以使用SpEL表达式引用模型数据。作为起步，我们看几个SpEL表达式的样例，以及如何将其注入到bean中。然后我们会深入学习一些SpEL的基础表达式，它们能够组合起来形成更为强大的表达式。

SpEL样例

SpEL是一种非常灵活的表达式语言，所以在本书中不可能面面俱到地介绍它的各种用法。但是我们可以展示几个基本的例子，这些例子会激发你的灵感，有助于你编写自己的表达式。

需要了解的第一件事情就是SpEL表达式要放到“#{ ... }”之中，这与属性占位符有些类似，属性占位符需要放到“${ ... }”之中。下面所展现的可能是最简单的SpEL表达式了：

除去“#{ ... }”标记之后，剩下的就是SpEL表达式体了，也就是一个数字常量。这个表达式的计算结果就是数字1，这恐怕并不会让你感到丝毫惊讶。

当然，在实际的应用程序中，我们可能并不会使用这么简单的表达式。我们可能会使用更加有意思的表达式，如：它的最终结果是计算表达式的那一刻当前时间的毫秒数。T()表达式会将java.lang.System视为Java中对应的类型，因此可以调用其static修饰的currentTimeMillis()方法。

SpEL表达式也可以引用其他的bean或其他bean的属性。例如，如下的表达式会计算得到ID为sgtPeppers的bean的artist属性：

我们还可以通过systemProperties对象引用系统属性：

这只是SpEL的几个基础样例。在本章结束之前，你还会看到很多这样的表达式。但是，在此之前，让我们看一下在bean装配的时候如何使用这些表达式。

如果通过组件扫描创建bean的话，在注入属性和构造器参数时，我们可以使用@Value注解，这与之前看到的属性占位符非常类似。不过，在这里我们所使用的不是占位符表达式，而是SpEL表达式。例如，下面的样例展现了BlankDisc，它会从系统属性中获取专辑名称和艺术家的名字：

在XML配置中，你可以将SpEL表达式传入<property>或<constructor-arg>的value属性中，或者将其作为p-命名空间或c-命名空间条目的值。例如，在如下BlankDisc bean的XML声明中，构造器参数就是通过SpEL表达式设置的：

我们已经看过了几个简单的样例，也学习了如何将SpEL解析得到的值注入到bean中，那现在就来继续学习一下SpEL所支持的基础表达式吧。

表示字面值

我们在前面已经看到了一个使用SpEL来表示整数字面量的样例。它实际上还可以用来表示浮点数、String值以及Boolean值。

下面的SpEL表达式样例所表示的就是浮点值：

数值还可以使用科学记数法的方式进行表示。如下面的表达式计算得到的值就是98,700：

SpEL表达式也可以用来计算String类型的字面值，如：最后，字面值true和false的计算结果就是它们对应的Boolean类型的值。例如：在SpEL中使用字面值其实没有太大的意思，毕竟将整型属性设置为1，或者将Boolean属性设置为false时，我们并不需要使用SpEL。我承认在SpEL表达式中，只包含字面值情况并没有太大的用处。但需要记住的一点是，更有意思的SpEL表达式是由更简单的表达式组成的，因此了解在SpEL中如何使用字面量还是很有用处的。当组合更为复杂的表达式时，你迟早会用到它们。

引用bean、属性和方法

SpEL所能做的另外一件基础的事情就是通过ID引用其他的bean。例如，你可以使用SpEL将一个bean装配到另外一个bean的属性中，此时要使用bean ID作为SpEL表达式（在本例中，也就是sgtPeppers）：

现在，假设我们想在一个表达式中引用sgtPeppers的artist属性：

表达式主体的第一部分引用了一个ID为sgtPeppers的bean，分割符之后是对artist属性的引用。

除了引用bean的属性，我们还可以调用bean上的方法。例如，假设有另外一个bean，它的ID为artistSelector，我们可以在SpEL表达式

中按照如下的方式来调用bean的selectArtist()方法：

对于被调用方法的返回值来说，我们同样可以调用它的方法。例如，如果selectArtist()方法返回的是一个String，那么可以调用toUpperCase()将整个艺术家的名字改为大写字母形式：

如果selectArtist()的返回值不是null的话，这没有什么问题。为了避免出现NullPointerException，我们可以使用类型安全的运算符：

与之前只是使用点号（.）来访问toUpperCase()方法不同，现在我们使用了“?.”运算符。这个运算符能够在访问它右边的内容之前，确保它所对应的元素不是null。所以，如果selectArtist()的返回值是null的话，那么SpEL将不会调用toUpperCase()方法。表达式的返回值会是null。

在表达式中使用类型

如果要在SpEL中访问类作用域的方法和常量的话，要依赖T()这个关键的运算符。例如，为了在SpEL中表达Java的Math类，需要按照如下的方式使用T()运算符：

这里所示的T()运算符的结果会是一个Class对象，代表了java.lang.Math。如果需要的话，我们甚至可以将其装配到一个Class类型的bean属性中。但是T()运算符的真正价值在于它能够访问目标类型的静态方法和常量。

例如，假如你需要将PI值装配到bean属性中。如下的SpEL就能完成该任务：

与之类似，我们可以调用T()运算符所得到类型的静态方法。我们已经看到了通过T()调用System.currentTimeMillis()。如下的这个样例会计算得到一个0到1之间的随机数：

SpEL运算符

SpEL提供了多个运算符，这些运算符可以用在SpEL表达式的值上。表3.1概述了这些运算符。

表3.1　用来操作表达式值的SpEL运算符

运算符类型运　算　符

算术运算+、-、 \* 、/、%、^

比较运算< 、 > 、 == 、 <= 、 >= 、 lt 、 gt 、 eq 、 le 、 ge

逻辑运算and 、 or 、 not 、│

条件运算?: (ternary) 、 ?: (Elvis)

正则表达式matches

作为使用上述运算符的一个简单样例，我们看一下下面这个SpEL表达式：

这不仅是使用SpEL中乘法运算符（\*）的绝佳样例，它也为你展现了如何将简单的表达式组合为更为复杂的表达式。在这里PI的值乘以2，然后再乘以radius属性的值，这个属性来源于ID为circle的bean。实际上，它计算了circle bean中所定义圆的周长。

类似地，你还可以在表达式中使用乘方运算符（^）来计算圆的面积：

“^”是用于乘方计算的运算符。在本例中，我们使用它来计算圆半径的平方。

当使用String类型的值时，“+”运算符执行的是连接操作，与在Java中是一样的：

SpEL同时还提供了比较运算符，用来在表达式中对值进行对比。注意在表3.1中，比较运算符有两种形式：符号形式和文本形式。在大多数情况下，符号运算符与对应的文本运算符作用是相同的，使用哪一种形式均可以。

例如，要比较两个数字是不是相等，可以使用双等号运算符（==）：

或者，也可以使用文本型的eq运算符：

两种方式的结果都是一样的。表达式的计算结果是个Boolean值：如果counter.total等于100的话，为true，否则为false。

SpEL还提供了三元运算符（ternary），它与Java中的三元运算符非常类似。例如，如下的表达式会判断如果scoreboard.score>1000的话，计算结果为String类型的“Winner！”，否则的话，结果为Loser：

三元运算符的一个常见场景就是检查null值，并用一个默认值来替代null。例如，如下的表达式会判断disc.title的值是不是null，如果是null的话，那么表达式的计算结果就会是“Rattle and Hum”：

这种表达式通常称为Elvis运算符。这个奇怪名称的来历是，当使用符号来表示表情时，问号看起来很像是猫王（Elvis Presley）的头发。

计算正则表达式

当处理文本时，有时检查文本是否匹配某种模式是非常有用的。SpEL通过matches运算符支持表达式中的模式匹配。matches运算符对String类型的文本（作为左边参数）应用正则表达式（作为右边参数）。matches的运算结果会返回一个Boolean类型的值：如果与正则表达式相匹配，则返回true；否则返回false。

为了进一步解释matches运算符，假设我们想判断一个字符串是否包含有效的邮件地址。在这个场景下，我们可以使用matches运算符，如下所示：

探寻正则表达式语法的秘密超出了本书的范围，同时我们也应该意识到这里的正则表达式还不足够健壮来涵盖所有的场景。但对于演示matches运算符的用法，这已经足够了。

计算集合

SpEL中最令人惊奇的一些技巧是与集合和数组相关的。最简单的事情可能就是引用列表中的一个元素了：

这个表达式会计算songs集合中第五个（基于零开始）元素的title属性，这个集合来源于ID为jukebox bean。

为了让这个表达式更丰富一些，假设我们要从jukebox中随机选择一首歌：

“[]”运算符用来从集合或数组中按照索引获取元素，实际上，它还可以从String中获取一个字符。比如：

这个表达式引用了String中的第四个（基于零开始）字符，也就是“s”。[2]

SpEL还提供了查询运算符（.?[]），它会用来对集合进行过滤，得到集合的一个子集。作为阐述的样例，假设你希望得到jukebox中artist属性为Aerosmith的所有歌曲。如下的表达式就使用查询运算符得到了Aerosmith的所有歌曲：

可以看到，选择运算符在它的方括号中接受另一个表达式。当SpEL迭代歌曲列表的时候，会对歌曲集合中的每一个条目计算这个表达式。如果表达式的计算结果为true的话，那么条目会放到新的集合中。否则的话，它就不会放到新集合中。在本例中，内部的表达式会检查歌曲的artist属性是不是等于Aerosmith。

SpEL还提供了另外两个查询运算符：“.^[]”和“.$[]”，它们分别用来在集合中查询第一个匹配项和最后一个匹配项。例如，考虑下面的表达式，它会查找列表中第一个artist属性为Aerosmith的歌曲：

最后，SpEL还提供了投影运算符（.![]），它会从集合的每个成员中选择特定的属性放到另外一个集合中。作为样例，假设我们不想要歌曲对象的集合，而是所有歌曲名称的集合。如下的表达式会将title属性投影到一个新的String类型的集合中：

实际上，投影操作可以与其他任意的SpEL运算符一起使用。比如，我们可以使用如下的表达式获得Aerosmith所有歌曲的名称列表：

我们所介绍的只是SpEL功能的一个皮毛。在本书中还有更多的机会继续介绍SpEL，尤其是在定义安全规则的时候。

现在对SpEL的介绍要告一段落了，不过在此之前，我们有一个提示。在动态注入值到Spring bean时，SpEL是一种很便利和强大的方式。我们有时会忍不住编写很复杂的表达式。但需要注意的是，不要让你的表达式太智能。你的表达式越智能，对它的测试就越重要。SpEL毕竟只是String类型的值，可能测试起来很困难。鉴于这一点，我建议尽可能让表达式保持简洁，这样测试不会是什么大问题。

**3.6　小结**

我们在本章介绍了许多背景知识，在第2章所介绍的基本bean装配基础之上，又学习了一些强大的高级装配技巧。

首先，我们学习了Spring profile，它解决了Spring bean要跨各种部署环境的通用问题。在运行时，通过将环境相关的bean与当前激活的profile

进行匹配，Spring能够让相同的部署单元跨多种环境运行，而不需要进行重新构建。

Profile bean是在运行时条件化创建bean的一种方式，但是Spring 4提供了一种更为通用的方式，通过这种方式能够声明某些bean的创建与否要依赖于给定条件的输出结果。结合使用@Conditional注解和Spring Condition接口的实现，能够为开发人员提供一种强大和灵活的机制，

实现条件化地创建bean。

我们还看了两种解决自动装配歧义性的方法：首选bean以及限定符。尽管将某个bean设置为首选bean是很简单的，但这种方式也有其局限性，所以我们讨论了如何将一组可选的自动装配bean，借助限定符将其范围缩小到只有一个符合条件的bean。除此之外，我们还看到了如何创建自定义的限定符注解，这些限定符描述了bean的特性。

尽管大多数的Spring bean都是以单例的方式创建的，但有的时候其他的创建策略更为合适。Spring能够让bean以单例、原型、请求作用域或会话作用域的方式来创建。在声明请求作用域或会话作用域的bean的时候，我们还学习了如何创建作用域代理，它分为基于类的代理和基于接口的代理的两种方式。

最后，我们学习了Spring表达式语言，它能够在运行时计算要注入到bean属性中的值。

对于bean装配，我们已经掌握了扎实的基础知识，现在我们要将注意力转向面向切面编程（aspect-oriented programming ，AOP）了。依赖注入能够将组件及其协作的其他组件解耦，与之类似，AOP有助于将应用组件与跨多个组件的任务进行解耦。在下一章，我们将会深入学习在Spring中如何创建和使用切面。

[1]Java 8允许出现重复的注解，只要这个注解本身在定义的时候带有@Repeatable注解就可以。不过，Spring的@Qualifier注解并没有在定义时添加@Repeatable注解。

[2]不要责怪我，我不太认同这个名字。但是我必须承认，它看起来确实有点像猫王的头发。

第4章　面向切面的Spring

本章内容：

|  |
| --- |
| 面向切面编程的基本原理  通过POJO创建切面  使用@AspectJ注解  为AspectJ切面注入依赖 |

这些功能需多个地方用到，但又不想在每个点都明确调用它们。

日志、声明式事务、安全和缓存。

让应用对象只关注于自己业务领域问题，其他方面问题由其他应用对象来处理

横切关注点:散布于应用中多处的功能。

从概念上是与业务逻辑分离的（但往往会直接嵌入到业务逻辑之中）。

把这些横切关注点与业务逻辑相分离正是面向切面编程（AOP）要解决的问题。

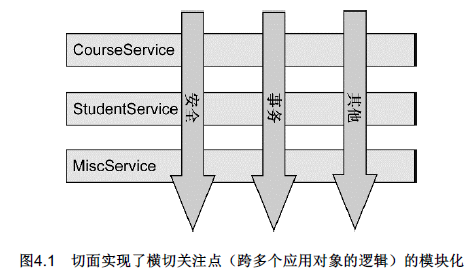
DI有助于应用对象间解耦，AOP可实现横切关注点与其所影响的对象间的解耦。

本章包括如何把普通类声明为一个切面和如何使用注解创建切面。

AspectJ——另一种流行的AOP实现——如何补充Spring AOP框架的功能。

**4.1　什么是面向切面编程**

横切关注点可被描述为影响应用多处的功能。如安全



模块核心功能都为特定业务领域服务，模块都需类似的辅助功能，例如安全和事务管理。

如要重用通用功能，最常见技术是继承或委托。

但如在整个应用中都使用相同基类，继承会导致一个脆弱的对象体系；

使用委托可能需要对委托对象进行复杂调用。

切面提供了取代继承和委托的另一种可选方案，且在很多场景下更清晰简洁。

仍在一个地方定义通用功能，可通过声明方式定义该功能要以何种方式在何处应用，而无需修改受影响的类。

横切关注点可被模块化为特殊的类，称为切面（aspect）。

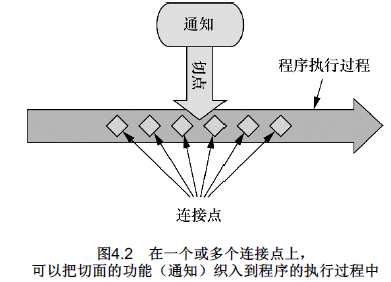
好处：

首先，每个关注点都集中于一个地方，而不是分散到多处代码中；

其次，服务模块更简洁，只包含主要关注点（或核心功能）代码，次要关注点代码被转移到切面中。

**4.1.1　定义AOP术语**

通知（advice）、切点（pointcut）和连接点（join point）。



* 通知（Advice）

通知:切面的工作(目标)。

通知定义了切面是什么(述切面要完成的工作)、何时用。

* 5种类型的通知：

|  |
| --- |
| * 前置通知（Before）：目标方法被调用前； * 后置通知（After）：目标方法完成后调用，不会关心方法的输出； * 返回通知（After-returning）：目标方法执行后调用； * 异常通知（After-throwing）：目标方法抛出异常后调用； * 环绕通知（Around）：在被通知方法调用前和调用后执行自定义的行为。 |

* 连接点（Join point）

应用可能有数以千计的时机应用通知。这些时机称为连接点。

连接点是在应用执行过程中能够插入切面的一个点。可是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可利用这些点插入到应用的正常流程之中，并添加新的行为。

* 切点（Poincut）

电力公司为每一个抄表员都分别指定某一块区域的住户。

一个切面并不需通知应用的所有连接点。

切点有助于缩小切面所通知的连接点的范围。

切点定义了“何处”。会匹配通知要织入的一或多个连接点。

通常用明确的类和方法名称，或正则表达式定义匹配的类和方法名称来指定切点。

* 切面（Aspect）：通知+切点

抄表员开始一天工作时，知道要做的事情（报告用电量）和从哪些房屋收集信息。

通知和切点共同定义了切面的全部内容—是什么，在何时和何处完成其功能。

* 引入（Introduction）

引入允许向现有类添加新方法或属性。

例如，我们可以创建一个Auditable通知类，该类记录了对象最后一次修改时的状态。setLastModified(Date)和一个实例变量来保存这个状态。然后，这个新方法和实例变量就可被引入到现有类中，从而可在无需修改这些现有的类的情况下，让其具有新的行为和状态。

* 织入（Weaving）

把切面应用到目标对象并创建新的代理对象的过程。

切面在指定的连接点被织入到目标对象中。

* 在目标对象的生命周期里的织入点：

|  |
| --- |
| * **编译期**   切面在目标类编译时被织入。需特殊的编译器。AspectJ的织入编译器。   * **类加载期**   目标类加载到JVM时。需特殊的类加载器（ClassLoader），可在目标类被引入应用前增强该目标类的字节码。  AspectJ 5的加载时织入（load-time weaving，LTW）。   * **运行期**   应用运行的某个时刻。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP用了。 |

通知包含了需要用于多个应用对象的横切行为；

连接点是程序执行过程中能够应用通知的所有点；

切点定义了通知被应用的具体位置（在哪些连接点）。

其中关键概念是切点定义了哪些连接点会得到通知。

**4.1.2　Spring对AOP的支持**

不是所有AOP框架都相同，在连接点模型上可能有强弱之分：字段修饰符级别、方法级别。

织入切面的方式和时机也有所不同。

创建切点来定义切面所织入的连接点是AOP框架的基本功能。

Spring和AspectJ间有大量协作，且Spring对AOP的支持在很多方面借鉴了AspectJ项目。

Spring提供了4种类型的AOP支持：

|  |
| --- |
| 基于代理的经典Spring AOP；（过于复杂，这里不介绍）  纯POJO切面；（声明式AOP）  @AspectJ注解驱动的切面；  注入式AspectJ切面（适用于Spring各版本）。 |

前三种都是Spring AOP实现的变体，Spring AOP构建在动态代理基础上，因此，Spring对AOP的支持局限于方法拦截。

现Spring引入了简单的声明式AOP和基于注解的AOP，Spring经典的AOP显得笨重和过于复杂，直接用 ProxyFactory Bean会让人厌烦。在本书中不再介绍经典的SpringAOP。

* 借助Spring的aop命名空间，可将纯POJO转换为切面。

实际上，这些POJO只是提供了满足切点条件时所要调用的方法。

需要XML配置，但这的确是声明式地将对象转换为切面的简便方式。

* Spring借鉴了AspectJ的切面，以提供注解驱动的AOP。本质上依然是Spring基于代理的AOP，但编程模型几乎与编写成熟的AspectJ注解切面完全一致。好处：能不使用XML来完成功能。
* 如AOP需求超过了简单的方法调用（如构造器或属性拦截），需考虑使用AspectJ来实现切面。

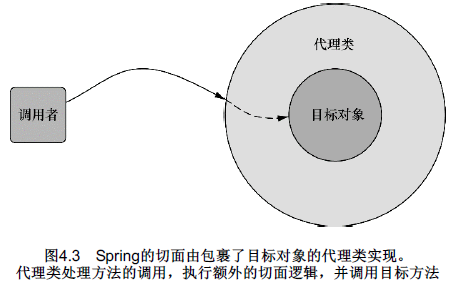
**Spring通知是Java编写的**

Spring所创建的通知是用标准的Java类编写的。

AspectJ与之相反。虽AspectJ现支持基于注解的切面，但AspectJ最初是以Java语言扩展的方式实现。通过特有AOP语言，可获得更强大和细粒度的控制，及更丰富的AOP工具集

**Spring在运行时通知对象**

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。



如图4.3所示，代理类封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。

直到应用需要被代理的bean时，Spring才创建代理对象。

在ApplicationContext从BeanFactory中加载所有bean时，Spring才会创建被代理的对象。

Spring运行时才创建代理对象，所以不需特殊编译器来织入Spring AOP的切面。

**Spring只支持方法级别的连接点**

通过用各种AOP方案可支持多种连接点模型。例如AspectJ和JBoss，除了方法切点，还提供了字段和构造器接入点。

因Spring基于动态代理，所以只支持方法连接点。

不支持字段连接点，无法创建细粒度通知，例如拦截对象字段的修改。

不支持构造器连接点，无法在bean创建时应用通知。

如需方法拦截之外的拦截功能，可利用Aspect来补充。

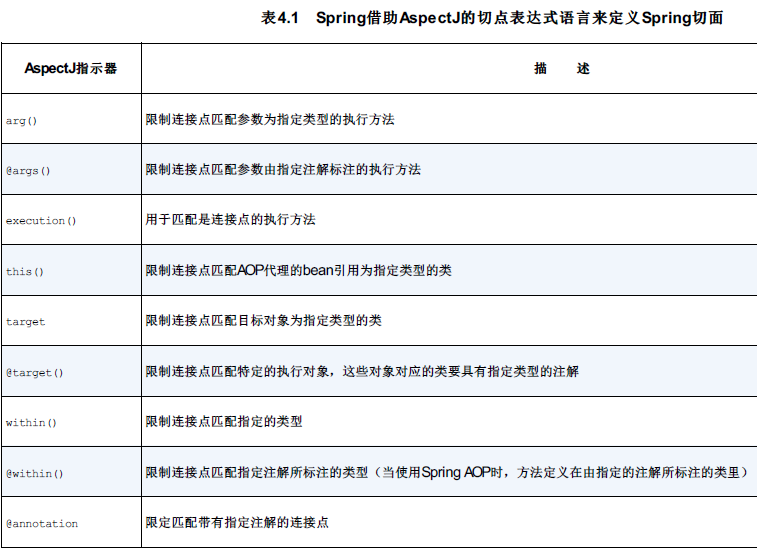
是时候学习如何在Spring中创建切面了，先从Spring的声明式AOP模型开始。

**4.2　通过切点来选择连接点**

切点用于准确定位应在什么地方应用切面(通知+切点)通知。

Spring AOP中，要使用AspectJ的切点表达式语言来定义切点。

Spring仅支持AspectJ切点指示器的一个子集。

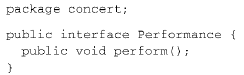


Spring中尝试用AspectJ其他指示器时，会抛出IllegalArgument-Exception异常

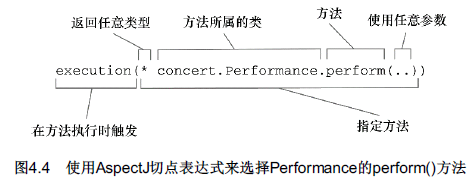
注意只有execution指示器是实际执行匹配的，是编写切点定义时最主要用的指示器,其他指示器都用来限制匹配

**4.2.1　编写切点**

为阐述Spring中切面，需有个主题来定义切面的切点。定义一Performance接口：

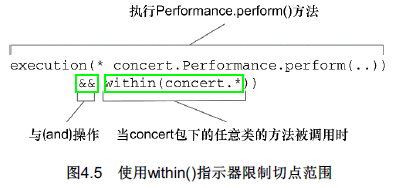


Performance可代表任何类型的现场表演，如舞台剧、电影或音乐会。假设我们想编写Performance的perform()方法触发的通知。



方法参数列表，（..）表明切点要选择任意的perform()方法，无论方法入参。

现假设需配置的切点仅匹配concert包。可用within()指示器来限制匹配



|  |
| --- |
| “&&”(and)  “||” (or)  “!” (not) |

**4.2.2　在切点中选择bean**

新的bean()指示器，允许在切点表达式中用bean的ID来标识bean。

bean()使用bean ID或bean名称作为参数来限制切点只匹配特定的bean。

例如，考虑如下的切点：



在这里，希望在执行Performance的perform()方法时应用通知，但限定bean的ID为woodstock。

**4.3　使用注解创建切面**

AspectJ 5引入的关键特性。通过少量注解把任意类转变为切面。

Performance接口，是切面中切点的目标对象。现在使用AspecJ注解来定义切面。

**4.3.1　定义切面**

* 程序清单4.1　Audience类：观看演出的切面



@AspectJ注解表明Audience不仅是个POJO[Plain Ordinary Java Object简单Java对象，实际就是普通JavaBean]，还是个切面。

Audience类中方法都使用注解来定义切面的具体行为。

Audience有四个方法，定义了观众在看演出时可能会做的事情。演出前，观众要就坐（takeSeats()）并将手机调至静音状态（silenceCellPhones()）。如演出很精彩的话，观众会鼓掌喝彩（applause()）。如演出没有达到观众预期，观众会要求退款（demandRefund()）。可看到，这些方法都使用了通知注解来表明它们应该在什么时候调用。

表4.2　Spring使用AspectJ注解来声明通知方法

|  |  |
| --- | --- |
| 注解 | 通知 |
| @After | 通知方法会在目标方法返回或抛出异常后调用 |
| @AfterReturning | 通知方法会在目标方法返回后调用 |
| @AfterThrowing | 通知方法会在目标方法抛出异常后调用 |
| @Around | 通知方法会将目标方法封装起来 |
| @Before | 通知方法会在目标方法调用前执行 |

takeSeats()和silence CellPhones()用到了@Before，应在演出开始前调用。

applause()用了@AfterReturning，会在演出成功返回后调用。

demandRefund()用了@AfterThrowing，会在抛出异常以后执行。

以上所有注解都用了同一个切点表达式作为值

* @Pointcut能在一个@AspectJ切面内定义可重用的切点。

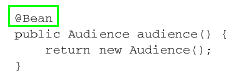
程序清单4.2　通过@Pointcut注解声明频繁使用的切点表达式



performance()的实际内容不重要，这里实际上应是空的。该方法本身只是个标识，供@Pointcut注解依附。

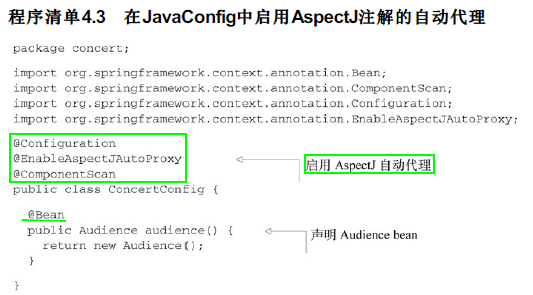
Audience类依然是一个POJO。能像使用其他的Java类那样调用它的方法，方法也能独立地进行单元测试。

* 像其他Java类一样，可装配为Spring中的bean：



如就此止步，Audience只会是Spring容器中的一个bean。不会被视为切面，注解不会解析，也不会创建将其转换为切面的代理。

1. 如使用JavaConfig，可在配置类的类级别上使用EnableAspectJ-AutoProxy注解启用自动代理功能



1. 如用XML装配bean，需使用Spring aop命名空间中的<aop:aspectj-autoproxy>



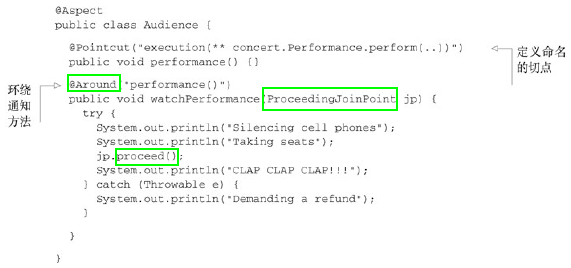
不管用JavaConfig还是XML，AspectJ自动代理都会为使用@Aspect注解的bean创建代理，会围绕着所有该切面的切点所匹配的bean。

Spring的AspectJ自动代理仅使用@AspectJ作为创建切面的指导，切面依然是基于代理的。本质上依然是Spring基于代理的切面。

**4.3.2　创建环绕通知**

最为强大。能让编写的逻辑将被通知的目标方法完全包装起来。实际上就像在一个通知方法中同时编写前置通知和后置通知。

程序清单4.5　使用环绕通知重新实现Audience切面



@Around表明watchPerformance()会作为performance()切点的环绕通知。

在这个通知中，观众在演出之前会将手机调至静音并就坐，演出结束后会鼓掌喝彩。像前面一样，如果演出失败的话，观众会要求退款。

效果与之前的前置和后置通知一样。但它们位于同一个方法中

接受ProceedingJoinPoint作为参数。通过它来调用被通知的方法。用proceed()

**4.3.3　处理通知中的参数**

重新看下2.4.4小节中的BlankDisc样例。play()方法会循环所有磁道并调用playTrack()方法。但也可通过playTrack()方法直接播放某一个磁道中的歌曲。

假设想记录每个磁道被播放次数。

一种方法是修改playTrack()方法，每次调用的时候记录数量。但看起来，这应该是切面要完成的任务。

* 为记录每个磁道播放次数，创建TrackCounter类，是通知playTrack()的一个切面。



不同点在于切点还声明了要提供给通知方法的参数。

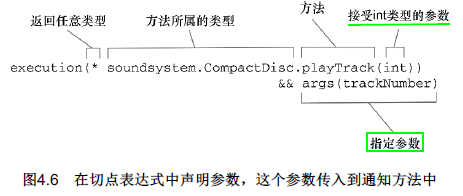


图4.6将切点表达式进行了分解，以展现参数是在什么地方指定。

args(trackNumber)：表明传递给playTrack()方法的int型参数也会传到通知中。

参数名称trackNumber也与切点方法签名中的参数相匹配。

通知方法通过@Before注解和命名切点trackPlayed(trackNumber)定义。

切点定义中的参数与切点方法中的参数名称一样，就完成了从命名切点到通知方法的参数转移。

* 现可在Spring配置中将BlankDisc和TrackCounter定义为bean，并启用AspectJ自动代理

程序清单4.7　配置TrackCount记录每个磁道播放的次数



* 测试。它会播放几个磁道并通过TrackCounter断言播放的数量。



目前为止，包装的都是被通知对象的已有方法。

让我们看一下如何通过编写切面，为被通知的对象引入全新的功能。

**4.3.4　通过注解引入新功能**

如Ruby和Groovy动态语言，可不用直接修改对象或类的定义，就能为对象或类增加新方法。

利用**引入**的AOP概念，切面可为Spring bean添加新方法。

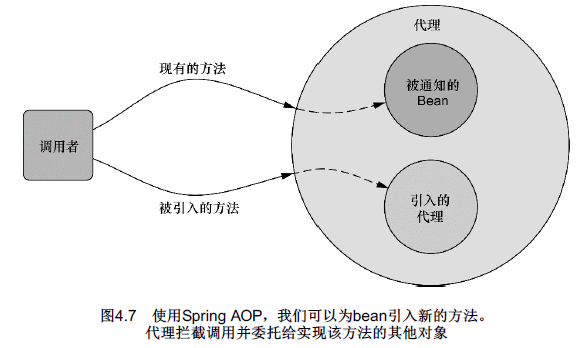
Spring中切面只是实现了所包装bean相同接口的代理。

如除了实现这些接口，代理也能暴露新接口的话，会怎样？

那样的话，切面所通知的bean看起来像是实现了新接口，即便底层实现类并没实现这些接口也无所谓。

图4.7　使用Spring AOP，可为bean引入新方法。

代理拦截调用并委托给实现该方法的其他对象



需注意的是，当引入接口的方法被调用，代理会把此调用委托给实现了新接口的其他对象。实际上，一个bean的实现被拆分到了多个类中。

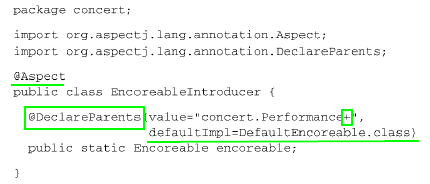
为了验证该主意，为示例中所有的Performance实现引入下面的Encoreable接口：



需有一种方式将这个接口应用到Performance实现中。

现假设能访问Performance的所有实现，并对其进行修改，让它们都实现Encoreable接口。但从设计的角度看，并不是最好的做法，不是所的Performance都是具有Encoreable特性的。另一方面，可能无法修改所有的Performance实现，当使用第三方实现并且没有源码的时候更是如此。【结论:该方法不好！】

借助于AOP的引入功能，可不必在设计上妥协或侵入性地改变现有实现。创建一个新的切面：



通过@DeclareParents注解，将Encoreable接口引入到Performance bean中。

|  |
| --- |
| @DeclareParents组成：   * value属性:哪种类型的bean要引入该接口。本例中是所有实现Performance的类型。（加号表示是Performance的所有子类型，而不是Performance本身。） * defaultImpl属性:指定了为引入功能提供实现的类。DefaultEncoreable * @DeclareParents注解:所标注的静态属性指明了要引入的接口。所引入的是Encoreable接口。 |

需要在Spring应用中将EncoreableIntroducer声明为一个bean：



Spring的自动代理机制会获取到它的声明，当Spring发现一个bean用了@Aspect注解，Spring就会创建一个代理，然后将调用委托给被代理的bean或被引入的实现，取决于调用的方法属于被代理的bean还是属于被引入的接口。

Spring中，注解和自动代理提供了一种很便利的方式来创建切面。非常简单，只涉及到最少的Spring配置。

但面向注解的切面声明有劣势：必须能够为通知类添加注解。必须要有源码。

如没有源码，或不想将AspectJ注解放到代码中，可在Spring XML配置文件中声明切面。

**4.4　在XML中声明切面**

本书前面内容中曾建立过一种原则:

基于注解的配置>基于Java的配置>基于XML的配置

但如果需声明切面，但又不能为通知类添加注解时，就必须转向XML配置。

表4.3　Spring的AOP配置元素能够以非侵入性的方式声明切面

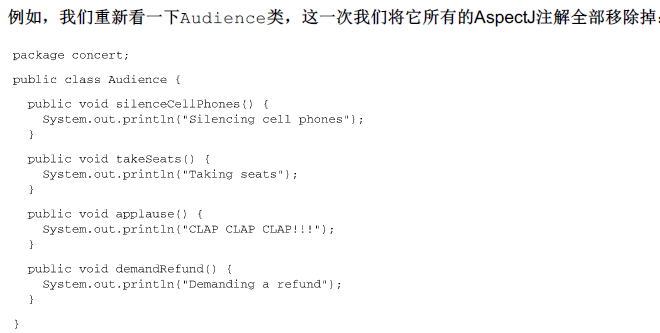
|  |  |
| --- | --- |
| AOP配置元素 | 用途 |
| <aop:advisor> | 定义AOP通知器 |
| <aop:after> | 定义AOP后置通知（不管被通知的方法是否执行成功） |
| <aop:after-returning> | 定义AOP返回通知 |
| <aop:after-throwing> | 定义AOP异常通知 |
| <aop:around> | 定义AOP环绕通知 |
| <aop:aspect> | 定义一个切面 |
| <aop:aspectj-autoproxy> | 启用@AspectJ注解驱动的切面 |
| <aop:before> | 定义一个AOP前置通知 |
| <aop:config> | 顶层的AOP配置元素。大多数的<aop:\*>元素必须包含在<aop:config>元素内 |
| <aop:declare-parents> | 以透明的方式为被通知的对象引入额外的接口 |
| <aop:pointcut> | 定义一个切点 |

<aop:aspectj-autoproxy>，能自动代理AspectJ注解的通知类。

aop命名空间的其他元素能让我们直接在Spring配置中声明切面，而不需用注解。

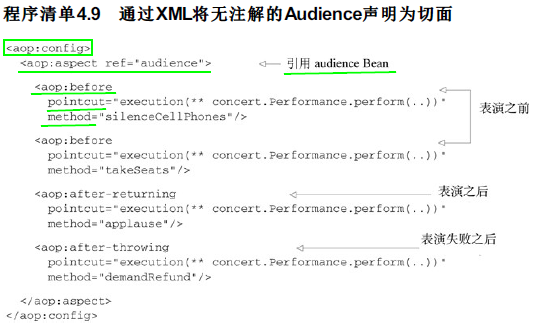
**4.4.1　声明前置和后置通知**

(1)



(2) 注册为Spring中的bean。

(3) 使用Spring aop命名空间中的元素，将没有注解的Audience类转换为切面。



把bean声明为切面时，总是从<aop:config>开始配置。

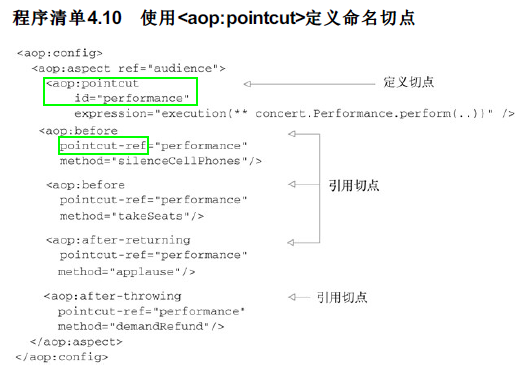
在<aop:config>内，可声明一个或多个通知器、切面或者切点。

用<aop:aspect>声明了一个简单切面。ref元素引用了一个POJO bean(audience)，该bean实现了切面功能，提供了在切面中通知所调用的方法。

例子:在执行perform()之前调用audience中的silenceCellPhone()方法。



* 将通用切点表达式抽取到切点声明中，该声明就能在所有通知元素中引用了。



<aop:pointcut>定义的切点可被同一个<aop:aspect>元素之内的所有通知元素引用。

如想让定义的切点能在多个切面用,可把<aop:pointcut>放在<aop:config>的范围内。

**4.4.2　声明环绕通知**

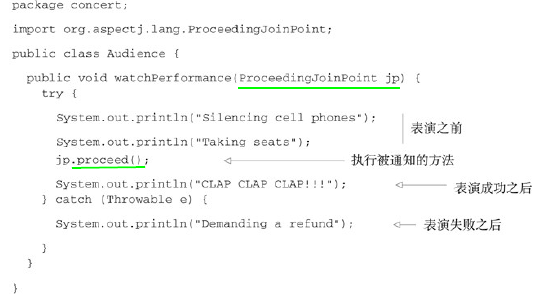
如不用成员变量存储信息，在前置通知和后置通知间共享信息非常麻烦。

例如，假设除了进场关闭手机和表演结束后鼓掌，还希望观众确保一直关注演出，并报告每个参赛者表演了多长时间。使用前置通知和后置通知实现该功能的唯一方式是在前置通知中记录开始时间并在某个后置通知中报告表演耗费的时间。这样必须在一个成员变量中保存开始时间。因Audience单例，如像这样保存状态，会存在线程安全问题。

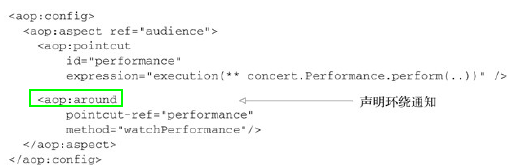
用环绕通知，可完成前置和后置通知的相同功能，且只需要在一个方法中实现。因整个通知逻辑在一个方法内实现，不需要使用成员变量保存状态。

例如，考虑程序清单4.11中新Audience类的watchPerformance()，没使用注解。

程序清单4.11　watchPerformance()方法提供了AOP环绕通知



程序清单4.12　在XML中使用<aop:around>元素声明环绕通知



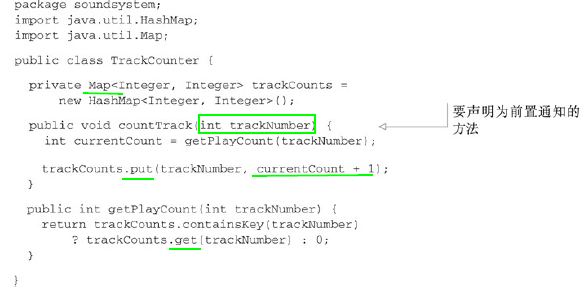
<aop:around>指定了切点和通知方法的名字。

**4.4.3　为通知传递参数**

4.3.3，用@AspectJ注解创建了切面，能记录CompactDisc上每个磁道播放次数。

(1)

程序清单4.13　无注解的TrackCounter



(2)Spring配置中声明了TrackCounter bean和BlankDisc bean，并将TrackCounter转化为切面。

程序清单4.14　在XML中将TrackCounter配置为参数化的切面



差别在于切点表达式中包含了一个参数，会传递到通知方法中。

唯一的差别在于这里使用and关键字而不是“&&”（因在XML中，“&”符号会被解析为实体的开始）。

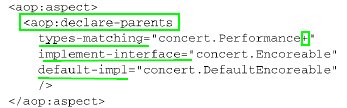
现在看一下如何使用aop命名空间声明引入切面。

**4.4.4　通过切面引入新功能**

4.3.4，借助AspectJ的@DeclareParents注解为被通知方法引入新方法。

但AOP引入并不是AspectJ特有的。<aop:declare-parents>，可实现相同功能。

* 如下XML代码片段与之前基于AspectJ的引入功能相同：



<aop:declare-parents>声明了此切面所通知的bean要在它的对象层次结构中拥有新的父类型。

本例中，类型匹配Performance接口的bean在父类结构中会增加Encoreable接口（由implement-interface指定）。

|  |
| --- |
| * default-impl用全限定类名来显式指定Encoreable的实现。(直接) * delegate-ref。引用了一个Spring bean作为引入的委托。(间接)需在Spring上下文中存在一个ID为encoreableDelegate的bean。 |

区别在于后者是Spring bean，本身可被注入、通知或使用其他的Spring配置。

**4.5　注入AspectJ切面**

AspectJ提供了SpringAOP(功能较弱)所不能支持的许多类型的切点。

1. 例如当需在创建对象时应用通知，构造器切点就非常方便。Spring AOP不行

2. AspectJ切面与Spring相互独立。应用AspectJ切面时几乎不会涉及到Spring。

3. 但是精心设计且有意义的切面很可能依赖其他类来完成它们的工作。如在执行通知时，切面依赖于一个或多个类，可在切面内部实例化这些协作对象。但更好的方式是，可借助Spring的依赖注入把bean装配进AspectJ切面中。

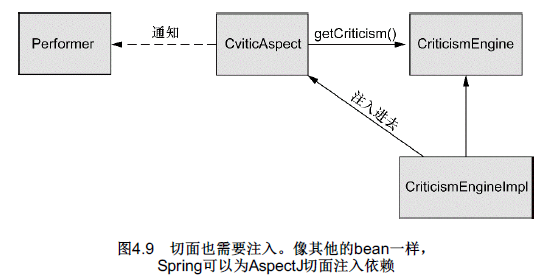
以切面方式创建评论员角色，会观看演出且会在演出之后提供一些批评意见。



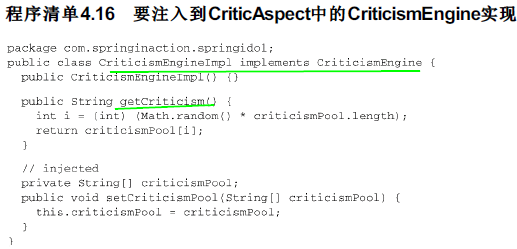
4.15中的performance()切点匹配perform()方法。当与afterReturning()配合使用时，可让该切面在表演结束时起作用。

4.15有趣的地方并不是在于评论员自己发表评论，实际上，CriticAspect与一个CriticismEngine对象相协作，在表演结束时，调用该对象的getCriticism()方法来发表一个苛刻的评论。

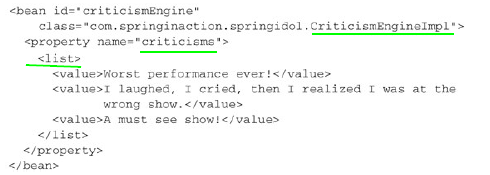
为避免CriticAspect和CriticismEngine间产生不必要的耦合，通过Setter依赖注入为CriticAspect设置CriticismEngine。

图4.9展示了此关系。  


像其他的bean一样，Spring可为AspectJ切面注入依赖

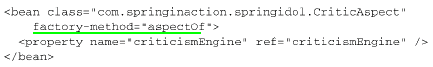


这个类可使用如下的XML声明为一个Spring bean。



现在有了一个要赋予CriticAspect的Criticism-Engine实现CriticismEngineImple。剩下的就是为CriticAspect装配CriticismEngineImple。

* AspectJ切面不需Spring就可织入到应用中。
* 如想使用Spring的依赖注入为AspectJ切面注入协作者，就需要在Spring配置中把切面声明为一个Spring配置中的<bean>。如下的<bean>声明会把criticismEnginebean注入到CriticAspect中：



最大的不同在于使用了factorymethod。

|  |
| --- |
| 通常Spring bean由Spring容器初始化，但AspectJ切面由AspectJ在运行期创建  等到Spring有机会为CriticAspect注入CriticismEngine时，CriticAspect已经被实例化了。  因为Spring不能负责创建CriticAspect，那就不能在 Spring中简单地把CriticAspect声明为一个bean。相反，需要一种方式为Spring获得已经由AspectJ创建的CriticAspect实例的句柄，从而可注入CriticismEngine。幸好，所有的AspectJ切面都提供了静态的aspectOf()方法，返回切面的单例。所以为了获得切面的实例，必须用factory-method来调用asepctOf()方法而不是调用CriticAspect的构造器方法。  总结: 即Spring不能像之前那样使用<bean>声明来创建一个CriticAspect实例—已在运行时由AspectJ创建完成了。  Spring需通过aspectOf()工厂方法获得切面的引用，然后像<bean>元素规定的那样在该对象上执行依赖注入 |

**4.6　小结**

AOP是面向对象编程的一个强大补充。

通过AspectJ，现在可把之前分散在应用各处的行为放入可重用的模块中。

显示地声明在何处如何应用该行为。

减少了代码冗余，并让我们的类关注自身的主要功能。

Spring提供了AOP框架，让我们把切面插入到方法执行的周围。

把通知织入前置、后置和环绕方法的调用中，为处理异常增加自定义的行为。

Spring中使用切面，可有多种选择。@AspectJ注解和简化的配置命名空间.

最后，当Spring AOP不能满足需求时，必须转向更为强大的AspectJ。

此时，已经覆盖了Spring框架的基础知识，了解到如何配置Spring容器及如何为Spring管理的对象应用切面。这些核心技术为创建松散耦合的应用奠定了坚实基础。

第2部分　Web中的Spring