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

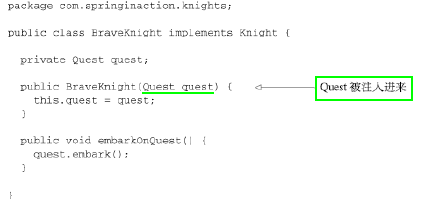
按传统做法，每个对象负责管理与自己相互协作的对象（依赖的对象）的引用，将会导致高度耦合和难以测试的代码。

* 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。

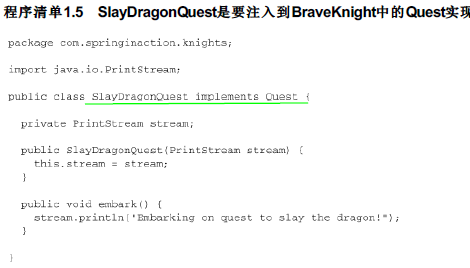
* 通过DI，对象的依赖关系将由系统中负责协调各对象的第三方组件在创建对象的时候进行设定。对象无需自行创建或管理它们的依赖关系，如图1.1所示，依赖关系将被自动注入到需要它们的对象当中去。



* 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。

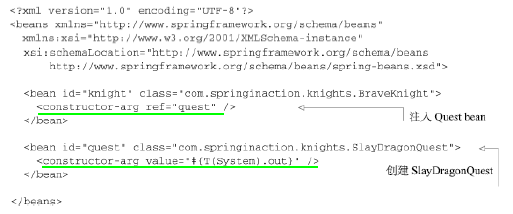
如一个对象只通过接口（而不是具体实现或初始化过程）来表明依赖关系，这种依赖就能够在对象本身毫不知情的情况下，用不同的具体实现进行替换。



* 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。

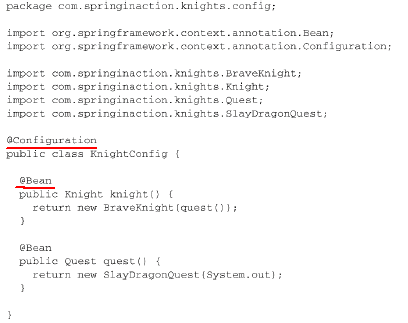
因knights.xml中的bean使用XML文件进行配置，所以选择ClassPathXmlApplicationContext[1]作为应用上下文比较合适。该类加载位于应用程序类路径下的一个或多个XML配置文件。



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。

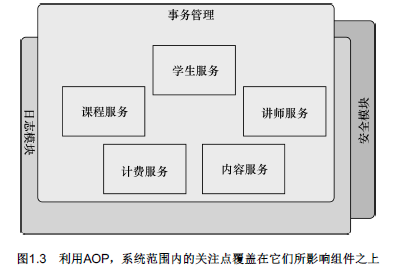
即使把这些关注点抽象为一个独立的模块，其他模块只是调用它的方法，但方法的调用还是会重复出现在各个模块中。

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。

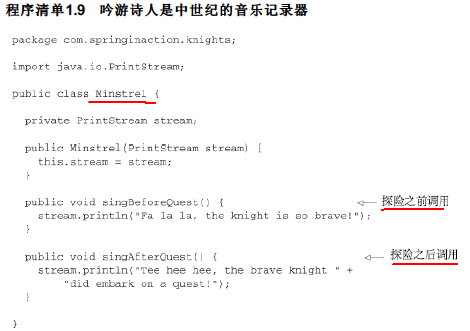
如图1.3所示，可把切面想象为覆盖在很多组件之上的一个外壳。应用由那些实现各自业务功能的模块组成。

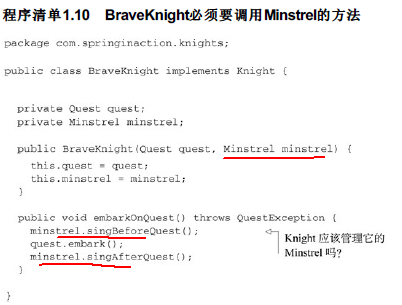
借助AOP，可使用各功能层去包裹核心业务层。这些层以声明方式灵活应用到系统中，核心应用甚至根本不知道它们的存在。可将安全、事务和日志关注点与核心业务逻辑相分离。



* 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。

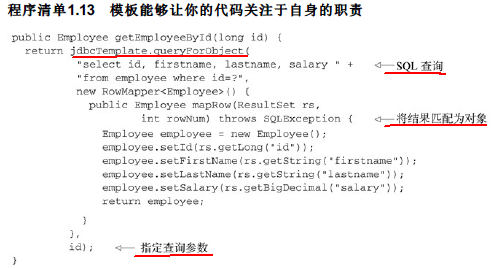


JMS、JNDI和使用REST服务通常也涉及大量的重复代码。

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：



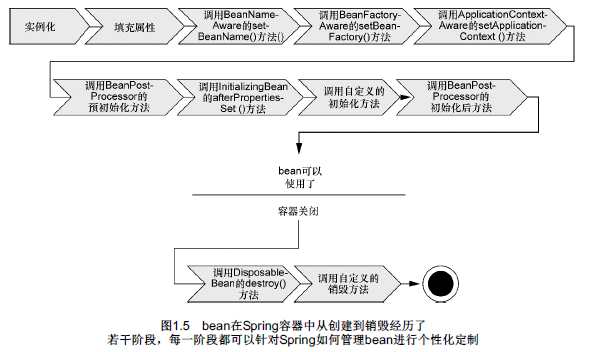
应用上下文准备就绪后，就可调用上下文的getBean()方法从Spring容器中获取bean。

**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。

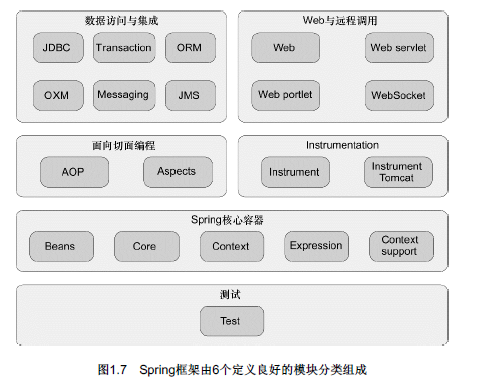
在Spring框架的范畴内，你会发现Spring简化Java开发的多种方式。但在Spring框架之外还存在一个构建在核心框架之上的庞大生态圈，它将Spring扩展到不同的领域，例如Web服务、REST、移动开发以及NoSQL。

**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。

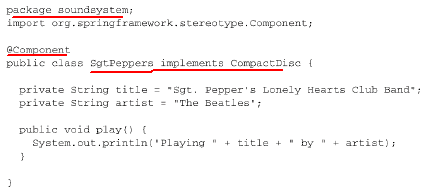
然后，会创建CDPlayer类，让Spring发现它，并将CompactDiscbean注入进来。

**2.2.1　创建可被发现的bean**

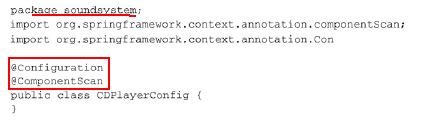
如不将CD插入（注入）到CD播放器中，CD播放器其实没有太大用处。所以， CD播放器依赖于CD才能完成它的使命。



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。

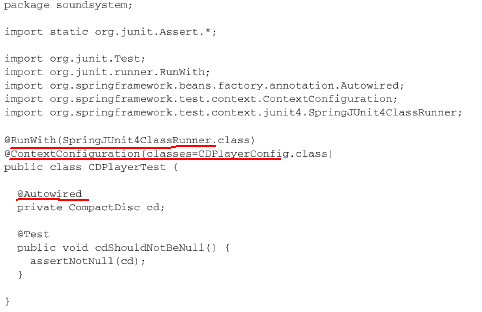
CDPlayerConfig类没有显式地声明任何bean，但使用了@ComponentScan注解，能在Spring中启用组件扫描。

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。

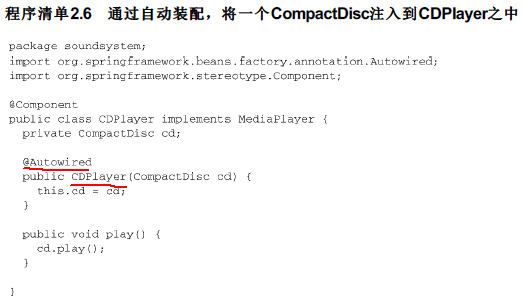
为basePackageClasses属性所设置的数组中包含了类。类所在的包将会作为组件扫描的基础包。

需能够将组件扫描得到的bean和它们的依赖装配在一起。

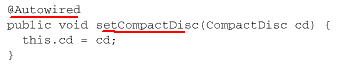
**2.2.4　通过为bean添加注解实现自动装配**

* 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

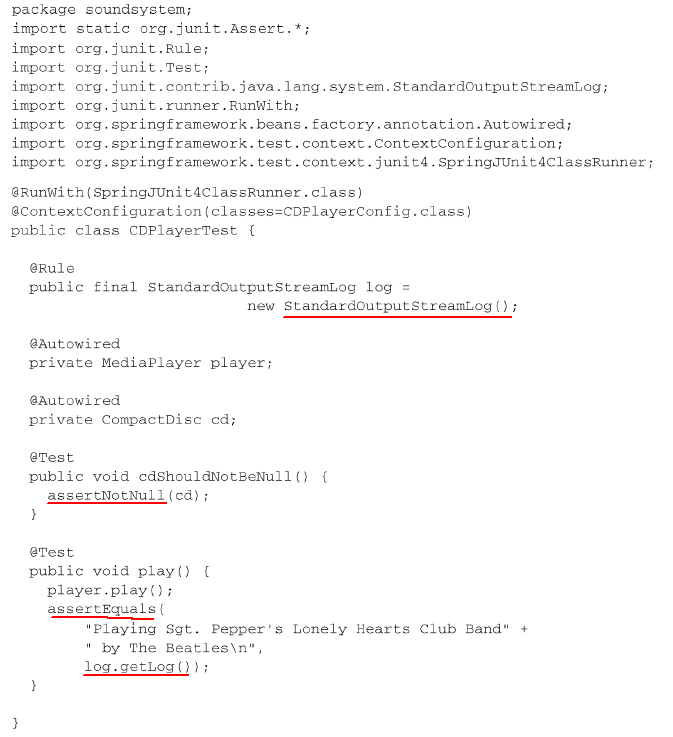


* @Autowired可用在类的任何方法上。

|  |
| --- |
| * 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。

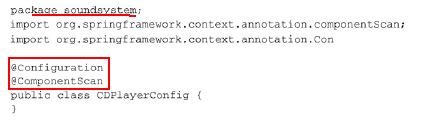
比如，想将第三方库中的组件装配到你的应用中，是没办法在其类上添加注解的，就不能用自动化装配方案。此时必须要采用显式装配的方式。（Java和XML）

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：

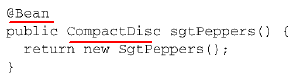


@Configuration表明该类是配置类，应包含在Spring应用上下文中如何创建bean的细节。

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

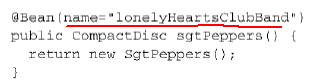
**2.3.2　声明简单的bean**

* 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

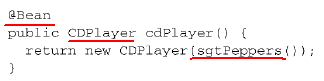


只要最终生成一个CompactDisc实例即可。

**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。

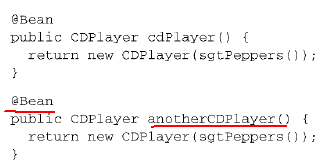
cdPlayer()的方法体与sgtPeppers()稍有区别。调用了需要传入CompactDisc对

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：

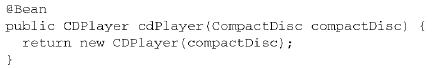


可将同一个SgtPeppers实例注入到任意数量的其他bean之中。

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)

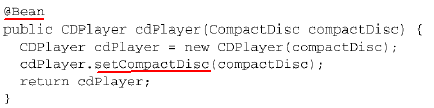


这里，cdPlayer()方法请求一个CompactDisc作为参数。

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**

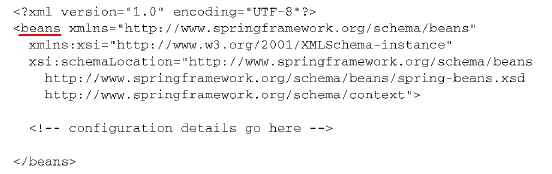
已看到如何让Spring自动发现和装配bean，如何进行手动干预，即通过JavaConfig显式地装配bean。

**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

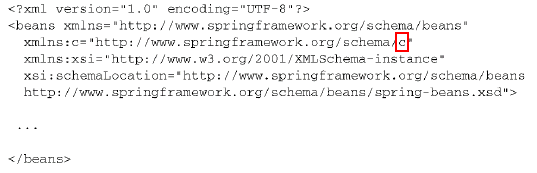
需在XML中声明CDPlayer并通过ID引用SgtPeppers：

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。

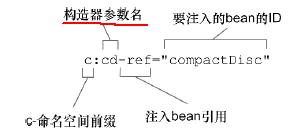
* c-命名空间：

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)

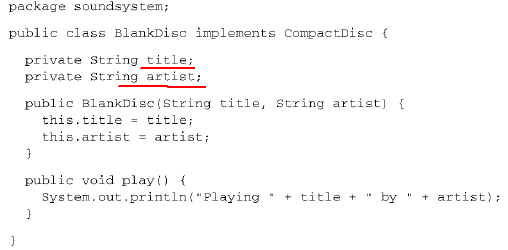


将参数名替换成了“0”，就是参数索引。因在XML中不允许数字作为属性的第一个字符，因此必须要添加一个下画线作为前缀。

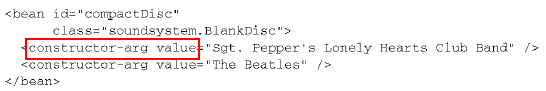
(2) 当只有一个属性需要注入的



* **将字面量注入到构造器中**

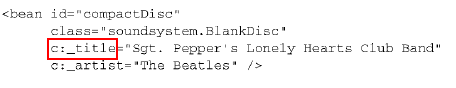
：

* <constructor-arg>

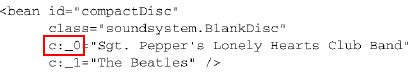


* c-命名空间的话

(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”

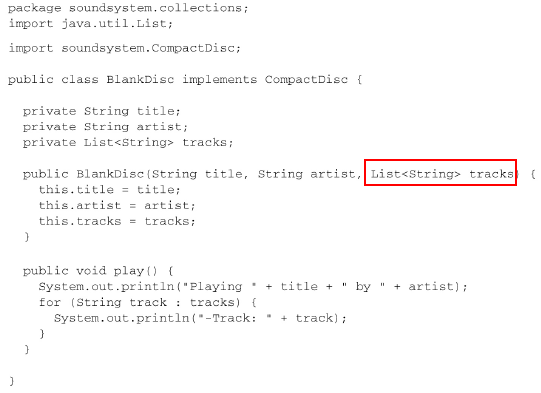


(2-1)如只有一个构造器参数的话。

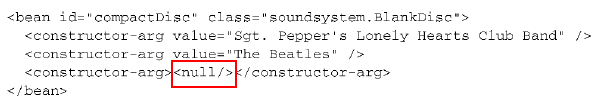


* 有一种情况<constructor-arg>能够实现，c-命名空间无法做到:装配集合

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



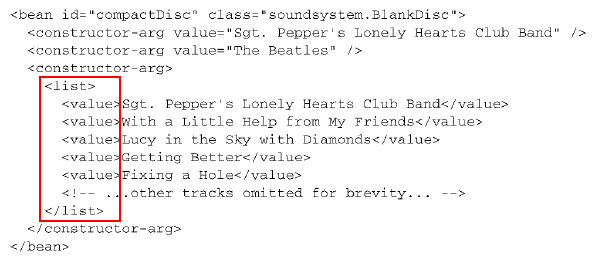
声明bean时，必须提供一个磁道列表。最简单的办法是将列表设置为null：



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。

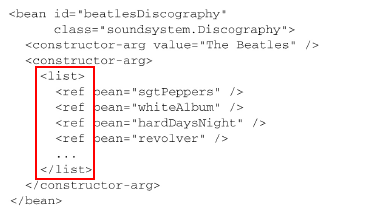
但当调用play()方法时，会遇到NullPointerException。

(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

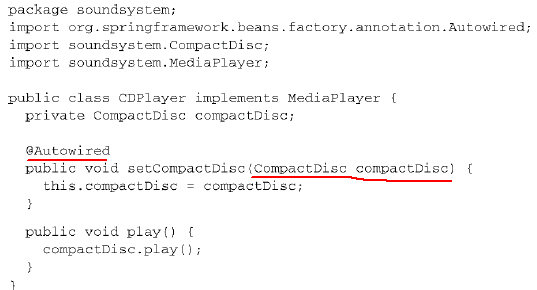
如是Set，重复值会被忽略掉，存放顺序也不会得以保证。(无序、不可重复)

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。

* 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。

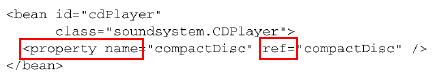
对BlankDisc，唱片名称、艺术家及磁道列表是强依赖，构造器注入

* 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）。

* p-命名空间，作为<property>元素的替代方案。

为启用p-命名空间，须在XML文件中与其他的命名空间一起对其进行声明：

可使用p-命名空间，按照以下的方式装配compactDisc属性：

p-命名空间中属性所遵循的命名约定与c-命名空间中的属性类似。图2.2阐述了p-命名空间属性是如何组成的。

图2.2　借助Spring的p-命名空间，将bean引用注入到属性中

首先，属性的名字使用了“p:”前缀，表明我们所设置的是一个属性。接下来就是要注入的属性名。最后，属性的名称以“-ref”结尾，这会提示

Spring要进行装配的是引用，而不是字面量。

将字面量注入到属性中

属性也可以注入字面量，这与构造器参数非常类似。作为示例，我们重新看一下BlankDisc bean。不过，BlankDisc这次完全通过属性注

入进行配置，而不是构造器注入。新的BlankDisc类如下所示：

现在，它不再强制要求我们装配任何的属性。你可以按照如下的方式创建一个BlankDiscbean，它的所有属性全都是空的：

当然，如果在装配bean的时候不设置这些属性，那么在运行期CD播放器将不能正常播放内容。play()方法可能会遇到的输出内容是“Playing

null by null”，随之会抛出NullPointerException异常，这是因为我们没有指定任何的磁道。所以，我们需要装配这些属性，可以借

助<property>元素的value属性实现该功能：

在这里，除了使用<property>元素的value属性来设置title和artist，我们还使用了内嵌的<list>元素来设置tracks属性，这与之

前通过<constructor-arg>装配tracks是完全一样的。

另外一种可选方案就是使用p-命名空间的属性来完成该功能：

与c-命名空间一样，装配bean引用与装配字面量的唯一区别在于是否带有“-ref”后缀。如果没有“-ref”后缀的话，所装配的就是字面量。

但需要注意的是，我们不能使用p-命名空间来装配集合，没有便利的方式使用p-命名空间来指定一个值（或bean引用）的列表。但是，我们可

以使用Spring util-命名空间中的一些功能来简化BlankDiscbean。

首先，需要在XML中声明util-命名空间及其模式：

util-命名空间所提供的功能之一就是<util:list>元素，它会创建一个列表的bean。借助<util:list>，我们可以将磁道列表转移

到BlankDisc bean之外，并将其声明到单独的bean之中，如下所示：

现在，我们能够像使用其他的bean那样，将磁道列表bean注入到BlankDisc bean的tracks属性中：

<util:list>只是util-命名空间中的多个元素之一。表2.1列出了util-命名空间提供的所有元素。

在需要的时候，你可能会用到util-命名空间中的部分成员。但现在，在结束本章前，我们看一下如何将自动化配置、JavaConfig以及XML配

置混合并匹配在一起。

**2.5　导入和混合配置**

在典型的Spring应用中，我们可能会同时使用自动化和显式配置。即便你更喜欢通过JavaConfig实现显式配置，但有的时候XML却是最佳的方

案。

幸好在Spring中，这些配置方案都不是互斥的。你尽可以将JavaConfig的组件扫描和自动装配和/或XML配置混合在一起。实际上，就像在2.2.1

小节中所看到的，我们至少需要有一点显式配置来启用组件扫描和自动装配。

关于混合配置，第一件需要了解的事情就是在自动装配时，它并不在意要装配的bean来自哪里。自动装配的时候会考虑到Spring容器中所有的

bean，不管它是在JavaConfig或XML中声明的还是通过组件扫描获取到的。

你可能会想在显式配置时，比如在XML配置和Java配置中该如何引用bean呢。让我们先看一下如何在JavaConfig中引用XML配置的bean。

**2.5.1　在JavaConfig中引用XML配置**

现在，我们临时假设CDPlayerConfig已经变得有些笨重，我们想要将其进行拆分。当然，它目前只定义了两个bean，远远称不上复杂的

Spring配置。不过，我们假设两个bean就已经太多了。

我们所能实现的一种方案就是将BlankDisc从CDPlayerConfig拆分出来，定义到它自己的CDConfig类中，如下所示：

compactDisc()方法已经从CDPlayerConfig中移除掉了，我们需要有一种方式将这两个类组合在一起。一种方法就是

在CDPlayerConfig中使用@Import注解导入CDConfig：

或者采用一个更好的办法，也就是不在CDPlayerConfig中使用@Import，而是创建一个更高级别的SoundSystemConfig，在这个类中

使用@Import将两个配置类组合在一起：

不管采用哪种方式，我们都将CDPlayer的配置与BlankDisc的配置分开了。现在，我们假设（基于某些原因）希望通过XML来配

置BlankDisc，如下所示：

现在BlankDisc配置在了XML之中，我们该如何让Spring同时加载它和其他基于Java的配置呢？

答案是@ImportResource注解，假设BlankDisc定义在名为cd-config.xml的文件中，该文件位于根类路径下，那么可以修

改SoundSystemConfig，让它使用@ImportResource注解，如下所示：

两个bean——配置在JavaConfig中的CDPlayer以及配置在XML中BlankDisc——都会被加载到Spring容器之中。因为CDPlayer中带

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

让我们继续这个练习，但是这一次，我们需要在XML中引用JavaConfig声明的bean。

**2.5.2　在XML配置中引用JavaConfig**

假设你正在使用Spring基于XML的配置并且你已经意识到XML逐渐变得无法控制。像前面一样，我们正在处理的是两个bean，但事情实际上会

变得更加糟糕。在被无数的尖括号淹没之前，我们决定将XML配置文件进行拆分。

在JavaConfig配置中，我们已经展现了如何使用@Import和@ImportResource来拆分JavaConfig类。在XML中，我们可以使用import元

素来拆分XML配置。

比如，假设希望将BlankDisc bean拆分到自己的配置文件中，该文件名为cd-config.xml，这与我们之前使用@ImportResource是一样的。

我们可以在XML配置文件中使用<import>元素来引用该文件：

现在，我们假设不再将BlankDisc配置在XML之中，而是将其配置在JavaConfig中，CDPlayer则继续配置在XML中。基于XML的配置该

如何引用一个JavaConfig类呢？

事实上，答案并不那么直观。<import>元素只能导入其他的XML配置文件，并没有XML元素能够导入JavaConfig类。

但是，有一个你已经熟知的元素能够用来将Java配置导入到XML配置中：<bean>元素。为了将JavaConfig类导入到XML配置中，我们可以这

样声明bean：

采用这样的方式，两种配置——其中一个使用XML描述，另一个使用Java描述——被组合在了一起。类似地，你可能还希望创建一个更高层次

的配置文件，这个文件不声明任何的bean，只是负责将两个或更多的配置组合起来。例如，你可以将CDConfig bean从之前的XML文件中移

除掉，而是使用第三个配置文件将这两个组合在一起：

不管使用JavaConfig还是使用XML进行装配，我通常都会创建一个根配置（root configuration），也就是这里展现的这样，这个配置会将两个

或更多的装配类和/或XML文件组合起来。我也会在根配置中启用组件扫描（通过<context:component-scan>或@ComponentScan）。

你会在本书的很多例子中看到这种技术。

**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，它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。

在本章中，我们看到了在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。不管你采用什么方

式，这些技术都描述了Spring应用中的组件以及这些组件之间的关系。

我同时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。但是，如果你确实需要显式配置Spring的话，应该优先选择基于Java

的配置，它比基于XML的配置更加强大、类型安全并且易于重构。在本书中的例子中，当决定如何装配组件时，我都会遵循这样的指导意见。

因为依赖注入是Spring中非常重要的组成部分，所以本章中介绍的技术在本书中所有的地方都会用到。基于这些基础知识，下一章将会介绍一

些更为高级的bean装配技术，这些技术能够让你更加充分地发挥Spring容器的威力。