## 1 Spring 简介

Spring是一个开源框架，致力于简化企业级Java开发，促进代码的松散耦合，它成功的关键在于依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming，AOP）。

## 2 依赖注入

在Spring中，对象无需自己查找或创建与其所关联的其他对象。相反，容器负责把需要相互协作的对象引用赋予各个对象。创建应用对象之间协作关系的行为通常称为装配（wiring），这也是依赖注入（DI）的本质。

### 2.1 自动化装配Bean

Spring从两个角度来实现自动化装配：

（1）组件扫描（component scanning）：Spring会自动发现应用上下文中所创建的bean。

（2）自动装配（auto wiring）：Spring自动满足bean之间的依赖。

组件扫描和自动装配组合在一起就能发挥出强大的威力， 它们能够将显式配置降低到最少。

#### 2.1.1 @Component

在这个MP3和流式媒体音乐的时代，CD（compact disc）显得有点典雅甚至陈旧。尽管如此， CD为我们阐述DI如何运行提供了一个很好的样例。如果你不将CD插入（注入）到CD播放器中，那么CD播放器其实是没有太大用处的。所以，可以这样说，CD播放器依赖于CD才能完成它的使命。

为了在Spring中阐述这个例子，让我们首先在Java中建立CD的概念。程序清单2.1展现了CompactDisc，它是定义CD的一个接口：

CompactDisc接口在Java中定义了CD的概念。

|  |
| --- |
| **package** soundsystem;  **public** **interface** CompactDisc {  **void** play();  } |

带有@Component注解的CompactDisc实现类SgtPeppers

|  |
| --- |
| **package** soundsystem;  **import** org.springframework.stereotype.Component;  @Component  **public** **class** SgtPeppers **implements** CompactDisc {    **private** String title = "Sgt.Pepper's Lonely Hearts Club Band";  **private** String artist = "The Beatles";  @Override  **public** **void** play() {  System.***out***.println("Playing " + title + " by " + artist);  }  } |

@Component注解表明该类会作为组件类，并且告知Spring要为这个类创建Bean，而组件扫描默认是不启用的，还需要显式的配置一下Spring，从而命令它去寻找带有@Component注解的类，并为其创建Bean。

Spring应用上下文中所有的Bean都会给定一个ID，如果不指定，Spring会根据类名指定ID为sgtPeppers，也就是将类名的第一个字母变小写。我们也可以给Bean指定一个ID，如下所示：

|  |
| --- |
| @Component("sgtPeppers")  **public** **class** SgtPeppers **implements** CompactDisc {  …  } |

#### 2.1.2 @ComponentScan

@ComponentScan注解启用了组件扫描

|  |
| --- |
| **package** soundsystem;  **import** org.springframework.context.annotation.ComponentScan;  **import** org.springframework.context.annotation.Configuration;  @Configuration  @ComponentScan  **public** **class** CDPlayerConfig {  } |

CDPlayerConfig类并没有显式地声明任何bean，只不过它使用了@ComponentScan注解，这个注解能够在Spring中启用组件扫描。如果没有其它配置的话，@ComponentScan默认会扫描与配置类相同的包。 因为CDPlayerConfig类位于soundsystem包中，因此Spring将会扫描这个包以及这个包下的所有子包，查找带有@Component注解的类。这样的话，就能发现CompactDisc，并且会在Spring中自动为其创建一个bean。

如果没有为@ComponentScan设置任何属性，则会以该类所在包为基础包（basepackage）来扫描组件，我们也可以指定不同的基础包，如下所示：

|  |
| --- |
| @ComponentScan("soundsystem")  **public** **class** CDPlayerConfig {} |

如果想更加清楚的表明设置的是基础包，也可以通过basePackages属性进行配置：

|  |
| --- |
| @ComponentScan(basePackages="soundsystem")  **public** **class** CDPlayerConfig {} |

同样可以设置多个基础包：

|  |
| --- |
| @ComponentScan(basePackages={"soundsystem", "video"})  **public** **class** CDPlayerConfig {} |

除了将基础包设置String类型外，@ComponentScan还提供了另外一种方法， 那就是将其指定为包中所包含的类或接口：

|  |
| --- |
| @ComponentScan(basePackageClasses={CDPlayerConfig.**class**, SgtPeppers.**class**})  **public** **class** CDPlayerConfig {} |

#### 2.1.3 @Autowired

创建一个简单的JUnit测试，它会创建Spring上下文， 并判断CompactDisc是不是真的创建出来了。

|  |
| --- |
| **package** soundsystem;  **import** **static** org.junit.Assert.*assertNotNull*;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration(classes=CDPlayerConfig.**class**)  **public** **class** CDPlayerTest {    @Autowired  **private** CompactDisc cd;    @Test  **public** **void** cdShouldNotBeNull() {  *assertNotNull*(cd);  cd.play();  }  } |

CDPlayerTest使用了Spring的SpringJUnit4ClassRunner，以便在测试开始的时候自动创建Spring的应用上下文。注解@ContextConfiguration会告诉它需要在CDPlayerConfig中加载配置。因为CDPlayerConfig类中包含了@ComponentScan，因此最终的应用上下文中应该包含CompactDisc类型的Bean。

运行测试类，结果呈现绿色，并且会在控制台打印Playing Sgt.Pepper's Lonely Hearts Club Band by The Beatles。说明Spring能够发现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。

测试代码中有一个带有@Autowired注解的CompactDisc类型的cd属性。@Autowired注解表明当Spring创建CDPlayer类型的bean的时候，会在应用上下文中找到一个CompactDisc的bean，并将其赋值给cd。

@Autowired不仅能应用到属性上，还能应用到构造器上：

|  |
| --- |
| **public** **class** CDPlayerTest {    **private** CompactDisc cd;  @Autowired  **public** CDPlayerTest(CompactDisc cd) {  **this**.cd = cd;  }  …  } |

Spring创建CDPlayerbean的时候，会通过这个构造器来进行实例化并且会传入一个可设置给CompactDisc类型的bean。

@Autowired还能应用到setter方法上：

|  |
| --- |
| **public** **class** CDPlayerTest {    **private** CompactDisc cd;  @Autowired  **public** **void** setCd(CompactDisc cd) {  **this**.cd = cd;  }  …  } |

setter方法并不特别，@Autowired注解可以用在类的任何方法上：

|  |
| --- |
| **public** **class** CDPlayerTest {    **private** CompactDisc cd;  @Autowired  **public** **void** insertDisc(CompactDisc cd) {  **this**.cd = cd;  }  …  } |

不管是构造器、Setter方法还是其他的方法，Spring都会尝试满足方法参数上所声明的依赖。假如有且只有一个bean匹配依赖需求的话，那么这个bean将会被装配进来。

如果没有匹配的bean，那么在应用上下文创建的时候，Spring会抛出一个异常。为了避免异常的出现，可以将@Autowired的required属性设置为false：

|  |
| --- |
| @Autowired(required=**false**)  **public** **void** insertDisc(CompactDisc cd) {  **this**.cd = cd;  } |

将required属性设置为false时，Spring会尝试执行自动装配，但是如果没有匹配的bean的话，Spring将会让这个bean处于未装配的状态。但是，把required属性设置为false时，如果在代码中没有进行null检查，这个处于未装配状态的属性有可能会出现NullPointerException。

### 2.2 通过XML装配Bean

在使用XML为Spring装配bean之前，需要创建一个新的配置规范。这意味着要创建一个XML文件，并且要以<beans>元素为根，还需要在配置文件的顶部声明多个XML模式（XSD）文件，这些文件定义了配置Spring的XML元素。如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <!-- Configuration details go here -->  </beans> |

#### 2.2.1 声明一个简单的bean

|  |
| --- |
| <bean class=*"soundsystem.SgtPeppers"* /> |

创建这个bean的类通过class属性来指定的，并且要使用全限定的类名。如果不指定ID，Spring将会根据全限定类名来进行命名，本例中，bean的ID将会是“soundsystem.SgtPeppers#0”。其中“#0”是计数形式，用来区分其他相同类型的bean。如果声明另外一个SgtPeppers，并且没有明确进行标识，那么它自动得到的ID将会是“soundsystem.SgtPeppers#1”。为了方便我们引用，可以通过id属性为bean设置ID：

|  |
| --- |
| <bean id=*"sgtPeppers"* class=*"soundsystem.SgtPeppers"* /> |

#### 2.2.2 借助构造器注入初始化bean

在Spring XML配置中，只有一种声明bean的方式：使用<bean>元素并指定class属性。Spring会从这里获取必要的信息来创建bean。但是，在XML中声明DI时，会有多种可选的配置方案和风格。具体到构造器注入 有两种基本的配置方案可供选择：

（1）<contructor-arg>元素

（2）c-命名空间

两者的区别在很大程度就是是否冗长烦琐<constructor-arg>元素比使用c-命名空间会更加冗长，从而导致XML更加难以读懂。另外，有些事情<constructor-arg>可以做到，但是使用c-命名空间却无法实现。

使用<contructor-arg>元素注入：

|  |
| --- |
| <bean id=*"cdPlayer"* class=*"soundsystem.CDPlayer"*>  <constructor-arg ref=*"sgtPeppers"* />  </bean> |

当Spring遇到这个<bean>元素时，它会创建一个CDPlayer实例。 <constructor-arg>元素会告知Spring要通过ref属性将一个ID为sgtPeppers的bean引用传递到CDPlayer的构造器中。

使用c-命名空间注入：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:c=*"http://www.springframework.org/schema/c"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>    <bean id=*"sgtPeppers"* class=*"soundsystem.SgtPeppers"* />  <bean id=*"cdPlayer"* class=*"soundsystem.CDPlayer"* c:cd-ref=*"sgtPeppers"* />  </beans> |

c:cd-ref="sgtPeppers"，其中c:为c-命名空间前缀，cd为构造器参数名，-ref为注入bean的引用，"sgtPeppers"为要注入bean的ID。

同样可以将字面量注入bean，下面是CompactDisc的一个新实现：

|  |
| --- |
| **package** soundsystem;  **public** **class** BlankDisc **implements** CompactDisc {  **private** String title;  **private** String artist;    **public** BlankDisc(String title, String artist) {  **this**.title = title;  **this**.artist = artist;  }  @Override  **public** **void** play() {  System.***out***.println("Playing " + title + " by " + artist);  }  } |

使用<constructor-arg>元素注入字面量

|  |
| --- |
| <bean id=*"blankDisc"* class=*"soundsystem.BlankDisc"*>  <constructor-arg value=*"Sgt.Pepper's Lonely Hearts Club Band"* />  <constructor-arg value=*"The Beatles"* />  </bean> |

使用XML注入时，相较于注入bean的引用，注入字面量只需要使用后value属性去替换ref属性即可

使用c-命名空间注入字面量：

|  |
| --- |
| <bean id=*"blankDisc"* class=*"soundsystem.BlankDisc"*  c:title=*"Sgt.Pepper's Lonely Hearts Club Band"*  c:artist=*"The Beatles"* /> |

使用c-命名空间注入时，相较于注入bean的引用，注入字面量只需要去掉-ref。

XML注入相较于c-命名空间的优势在于XML可以注入集合，假设每张唱片有多个磁道，可以为唱片提供一个磁道列表：

|  |
| --- |
| **package** soundsystem;  **import** java.util.List;  **public** **class** BlankDisc **implements** CompactDisc {  **private** String title;  **private** String artist;  **private** List<String> tracks;    **public** BlankDisc(String title, String artist, List<String> tracks) {  **this**.title = title;  **this**.artist = artist;  **this**.tracks = tracks;  }  @Override  **public** **void** play() {  System.***out***.println("Playing " + title + " by " + artist);  **for** (String track : tracks) {  System.***out***.println("-Track: " + track);  }  }  } |

注入字面量列表如下：

|  |
| --- |
| <bean id=*"blankDisc"* class=*"soundsystem.BlankDisc"*>  <constructor-arg value=*"Sgt.Pepper's Lonely Hearts Club Band"* />  <constructor-arg value=*"The Beatles"* />  <constructor-arg>  <list>  <value>Getting Better</value>  <value>Fixing A Hole</value>  <value>Sgt.Pepper's Lonely Hearts Club Band</value>  </list>  </constructor-arg>  </bean> |

同样可以注入引用：

|  |
| --- |
| **private** List<CompactDisc> cds; |
| <bean id=*"blankDisc"* class=*"soundsystem.BlankDisc"*>  <constructor-arg value=*"The Beatles"* />  <constructor-arg>  <list>  <ref bean=*"sgtPeppers"*/>  <ref bean=*"blankDisc"*/>  </list>  </constructor-arg>  </bean> |

<set>和<list>元素的区别不大，其中最重要的不同在于当Spring创建要装配的集合时，所创建的是java.util.Set还是java.util.List。如果是Set的话，所有重复的值都会被忽略掉，存放顺序也不会得以保证。不过无论在哪种情况下，<set>或<list>都可以用来装配List、 Set甚至数组。

#### 2.2.3 属性注入

使用属性的setter方法，实现XML的属性注入：

|  |
| --- |
| **package** soundsystem;  **import** org.springframework.stereotype.Component;  @Component("sgtPeppers")  **public** **class** SgtPeppers **implements** CompactDisc {    **private** String title;  **private** String artist;  **public** **void** setTitle(String title) {  **this**.title = title;  }    **public** **void** setArtist(String artist) {  **this**.artist = artist;  }    @Override  **public** **void** play() {  System.***out***.println("Playing " + title + " by " + artist);  }  } |
| **package** soundsystem;  **public** **class** CDPlayer {    **private** CompactDisc cd;    **public** **void** setCd(CompactDisc cd) {  **this**.cd = cd;  }    **public** **void** init() {  cd.play();  }  } |

配置如下：

|  |
| --- |
| <bean id=*"sgtPeppers"* class=*"soundsystem.SgtPeppers"*>  <property name=*"title"* value=*"Sgt.Pepper's Lonely Hearts Club Band"* />  <property name=*"artist"* value=*"The Beatles"* />  </bean>  <bean id=*"cdPlayer"* class=*"soundsystem.CDPlayer"*>  <property name=*"cd"* ref=*"sgtPeppers"* />  </bean> |

　 将bean的属性提供一个setter方法，然后通过配置bean的<property>属性注入属性值或者属性对应类型的bean的引用。与构造器注入类似，我们可以使用p-命名空间注入：

|  |
| --- |
| <bean id=*"sgtPeppers"* class=*"soundsystem.SgtPeppers"*  p:title=*"Sgt.Pepper's Lonely Hearts Club Band"*  p:artist=*"The Beatles"*/>    <bean id=*"cdPlayer"* class=*"soundsystem.CDPlayer"*  p:cd-ref=*"sgtPeppers"*/> |

像构造器注入一样，<property>属性可以注入集合，而p-命名空间不可以：

|  |
| --- |
| **private** Set<String> keys;  **private** List<CompactDisc> cds;  **public** **void** setCds(List<CompactDisc> cds) {  **this**.cds = cds;  }  **public** **void** setKeys(Set<String> keys) {  **this**.keys = keys;  } |
| <bean id=*"cdPlayer"* class=*"soundsystem.CDPlayer"*>  <property name=*"keys"*>  <set>  <value>pause</value>  <value>play</value>  <value>stop</value>  </set>  </property>  <property name=*"cds"*>  <list>  <bean class=*"soundsystem.SgtPeppers"* />  <bean class=*"soundsystem.BlankDisc"* />  </list>  </property>  </bean> |

## 3 面向切面编程

### 3.1 什么是面向切面编程

在软件开发中，散布于应用中多处的功能被称为横切关注点（crosscutting concern）。横切关注点可以被描述为影响应用多处的功能。例如，安全就是一个横切关注点，应用中的许多方法都会涉及到安全规则。切面能帮助我们模块化横切关注点。

如果要重用通用功能的话，最常见的面向对象技术是继承（inheritance）或委托（delegation）。但是，如果在整个应用中都使用相同的基类，继承往往会导致一个脆弱的对象体系；而使用委托可能需要对委托对象进行复杂的调用。切面提供了取代继承和委托的另一种可选方案，而且在很多场景下更清晰简洁。在使用面向切面编程时，我们仍然在一个地方定义通用功能，但是可以通过声明的方式定义这个功能要以何种方式在何处应用，而无需修改受影响的类。横切关注点可以被模块化为特殊的类，这些类被称为切面（aspect）。这样做有两个好处：首先，现在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 因为它们只包含主要关注点（或核心功能）的代码，而次要关注点的代码被转移到切面中了。

### 3.2 AOP相关术语

#### 3.2.1 通知（Advice）

切面的工作被称为通知。通知定义了切面是什么以及何时使用。除了描述切面要完成的工作，通知还解决了何时执行这个工作的问题。它应该应用在某个方法被调用之前？之后？之前和之后都调用？还是只在方法抛出异常时调用？

Spring切面可以应用5种类型的通知：

* 前置通知（Before）：在目标方法被调用之前调用通知功能；
* 后置通知（After）：在目标方法完成之后调用通知， 此时不会关心方法的输出是什么；
* 返回通知（After-returning）：在目标方法成功执行之后调用通知；
* 异常通知（After-throwing）：在目标方法抛出异常后调用通知；
* 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。

#### 3.2.2 连接点（Join point）

我们的应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

#### 3.2.3 切入点（Pointcut）

如果说通知定义了切面的“什么”和“何时”的话，那么切入点就定义了“何处”。切入点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切入点。

#### 3.2.4 切面（Aspect）

切面是通知和切入点的结合。通知和切入点共同定义了切面的全部内容——它是什么,在何时和何处完成其功能。

### 3.3 切入点表达式

在Spring AOP中， 要使用AspectJ的切点表达式语言来定义切点。定义一个表演的接口Performance：

|  |
| --- |
| **package** concert;  **public** **interface** Performance {  **void** perform();  } |

Performance的perform()方法触发的通知。如下展现了一个切点表达式， 这个表达式能够设置当perform()方法执行时触发通知的调用。

|  |
| --- |
| *execution(\*\* concert.Performance.perform(..))* |

execution代表在方法执行时触发，第一个\*代表任意方法修饰符，第二个\*代表返回任意类型，concert.Performance代表方法所属的类，perform代表方法，（..）代表使用任意参数。

### 3.4 使用注解创建切面

我们已经定义了Performance接口，它是切面中切点的目标对象。现在，让我们使用AspecJ注解来定义切面。

如果一场演出没有观众的话，那不能称之为演出。从演出的角度来看， 观众是非常重要的，但是对演出本身的功能来讲，它并不是核心，这是一个单独的关注点。因此，将观众定义为一个切面，并将其应用到演出上就是较为明智的做法：

|  |
| --- |
| **package** concert;  **import** org.aspectj.lang.annotation.AfterReturning;  **import** org.aspectj.lang.annotation.AfterThrowing;  **import** org.aspectj.lang.annotation.Aspect;  **import** org.aspectj.lang.annotation.Before;  @Aspect  **public** **class** Audience {    @Before("execution(\*\* concert.Performce.perform(..))") // 表演之前，手机静音  **public** **void** silenceCell() {  System.***out***.println("Silencing cell phone");  }    @Before("execution(\*\* concert.Performce.perform(..))") // 表演之前，入座  **public** **void** takeSeats() {  System.***out***.println("Taking seats");  }    @AfterReturning("execution(\*\* concert.Performce.perform(..))") // 表演完成，鼓掌  **public** **void** applause() {  System.***out***.println("CLAP CLAP CLAP!");  }    @AfterThrowing("execution(\*\* concert.Performce.perform(..))") // 表演失败，退款  **public** **void** demanRefund() {  System.***out***.println("Demanding a refund");  }    } |

Audience类使用@AspectJ注解进行了标注。该注解表明Audience不仅仅是一个POJO，还是一个切面。Audience类中的方法都使用注解来定义切面的具体行为。

AspectJ提供了五个注解来定义通知，如下表：

|  |  |
| --- | --- |
| **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之前执行 |

这些注解都给定了一个切入点表达式作为它的值，同时，这四个方法的切点表达式都是相同的，这就造成了代码的冗余。我们也可以设置不同的切入点表达式。对于本例来说，可以使用@Pointcut注解来定义一个切入点表达式，然后在需要的时候去引用它：

|  |
| --- |
| @Pointcut("execution(\*\* concert.Performce.perform(..))")  **public** **void** performce() {}    @Before("performce()") // 表演之前，手机静音  **public** **void** silenceCell() {  System.***out***.println("Silencing cell phone");  } |

performance()方法的实际内容并不重要，在这里它实际上应该是空的。其实该方法本身只是一个标识，供@Pointcut注解依附。

至此，Audience只会是Spring容器中的一个bean。即便使用了AspectJ注解，但它并不会被视为切面，这些注解不会解析，也不会创建将其转换为切面的代理。我们在XML配置文件中通过Spring的aop命名空间启用AspectJ自动代理：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.3.xsd"*>    <aop:aspectj-autoproxy />  </beans> |

AspectJ自动代理会为使用@Aspect注解的bean创建一个代理，这个代理会围绕着所有该切面的切点所匹配的bean。在这种情况下，将会为Audience创建一个代理，Audience类中的通知方法将会在perform()调用前后执行。

创建环绕通知：

|  |
| --- |
| @Pointcut("execution(\*\* concert.Performce.perform(..))")  **public** **void** performce() {}    @Around("performce()")  **public** **void** watchPerformance(ProceedingJoinPoint jp) {  **try** {  System.***out***.println("Silencing cell phone");  System.***out***.println("Taking seats");  jp.proceed();  System.***out***.println("CLAP CLAP CLAP!");  } **catch** (Throwable e) {  System.***out***.println("Demanding a refund");  }  } |

@Around注解表明watchPerformance()方法会作为performance()切入点的环绕通知。这个方法接受ProceedingJoinPoint作为参数。这个对象是必须要有的，因为要在通知中通过它来调用被通知的方法。通知方法中可以做任何的事情，当要将控制权交给被通知的方法时，它需要调用ProceedingJoinPoint的proceed()方法。

### 3.5 在XML中创建切面

在Spring的aop命名空间中，提供了多个元素用来在XML中声明切面，如下表：

|  |  |
| --- | --- |
| **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> | 定义一个切入点 |

<aop:aspectj-autoproxy>元素能够自动代理AspectJ注解的通知类。aop命名空间的其他元素能够让我们直接在Spring配置中声明切面，而不需要使用注解。

例如，我们重新看一下Audience类，这一次我们将它所有的AspectJ注解全部移除掉：

|  |
| --- |
| **package** concert;  **import** org.aspectj.lang.ProceedingJoinPoint;  **public** **class** Audience {    **public** **void** silenceCell() {  System.***out***.println("Silencing cell phone");  }    **public** **void** takeSeats() {  System.***out***.println("Taking seats");  }    **public** **void** applause() {  System.***out***.println("CLAP CLAP CLAP!");  }    **public** **void** demanRefund() {  System.***out***.println("Demanding a refund");  }    **public** **void** watchPerformance(ProceedingJoinPoint jp) {  **try** {  System.***out***.println("Silencing cell phone");  System.***out***.println("Taking seats");  jp.proceed();  } **catch** (Throwable e) {  System.***out***.println("Demanding a refund");  }  }    } |

通过XML将无注解的Audience声明为切面：

|  |
| --- |
| <aop:config>  <!-- 引入audience Bean -->  <aop:aspect ref=*"audience"*>  <aop:pointcut expression=*"execute(\*\* concert.Performance.perform(..))"* id=*"performance"*/>  <!-- 前置通知 -->  <aop:before method=*"silenceCellPhone"* pointcut=*"performance"* />  <aop:before method=*"takeSeats"*  pointcut=*"execute(\*\* concert.Performance.perform(..))"* />  <!-- 后置通知 -->  <aop:after-returning method=*"applause"*  pointcut=*"execute(\*\* concert.Performance.perform(..))"* />  <!-- 异常通知 -->  <aop:after-throwing method=*"demandRefund"*  pointcut=*"execute(\*\* concert.Performance.perform(..))"* />  <!-- 环绕通知 -->  <aop:around method=*"watchPerformance"*  pointcut=*"execute(\*\* concert.Performance.perform(..))"* />  </aop:aspect>  </aop:config>  <bean id=*"audience"* class=*"concert.Audience"* /> |

大多数的AOP配置元素必须在<aop:config>元素的上下文内使用。使用<aop:aspect>元素声明了一个简单的切面。ref元素引用了一个POJO bean， 该bean实现了切面的功能——在这里就是audience。ref元素所引用的bean提供了在切面中通知所调用的方法。

当切入点重复时，我们可以使用<aop:pointcut>元素声明一个切入点：

|  |
| --- |
| <aop:pointcut expression=*"execute(\*\* concert.Performance.perform(..))"* id=*"performance"*/>  <aop:before method=*"silenceCellPhone"* pointcut=*"performance"* /> |

## 4 总结

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通过依赖注入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 的依赖注入的优点在于：

* 传统的代码，每个对象负责管理与自己需要依赖的对象，导致如果需要切换依赖对象的实现类时，需要修改多处地方。同时，过度耦合也使得对象难以进行单元测试。
* 依赖注入把对象的创造交给外部去管理,很好的解决了代码紧耦合（tight couple）的问题，是一种让代码实现松耦合（loose couple）的机制。
* 松耦合让代码更具灵活性，能更好地应对需求变动，以及方便单元测试。

由于采用反射的方式来实现依赖注入，在一定程度会影响性能。

AOP是面向对象编程的一个强大补充。通过AspectJ，可以把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。这有效减少了代码冗余，并让我们的类关注自身的主要功能，提高了开发效率。