ip_image044.jpeg

ip_image050.jpeg

<aop:aspectj-autoproxy />

**1.基本概念**

（1）IOC：控制反转。应用本身不负责依赖对象的创建和维护，由外部容器（如Spring）创建并维护依赖对象。此时控制权由应用转到了外部容器，控制权的转移就是控制反转。

（2）DI：依赖注入。在运行期，由外部容器动态的将依赖对象注入到组件中。Spring可以帮助进一步解耦，如PersonService bean中不用明确的为PersonDao成员变量new初值。而是通过Spring配置文件，在实例化这个bean的时候动态初始化PersonDao成员变量，根据需求可以初始化为StudentDaoImpl或TeacherDaoImpl。

**2.Spring配置文件**

Spring配置文件名称可以随意取，一般叫做ApplicationContext.xml,放在src下。和Struts中配置文件一样，Spring中配置文件也可以通过<import resource="...">来引入其他Spring配置文件。

启动Spring容器，加载Spring配置文件的方法是：

ApplicationContext ctx = new ClassPathXmlApplicationContext(String filePath);

之后就可以通过 ctx.getBean(配置文件中bean id)来获得Bean实例了，getBean默认返回Object类型，所以需要向下转型。

**3.实例化Bean的方式（三种）**

Spring实例化bean，创建bean对象的原理都是利用了Java反射机制

（1）通过构造器实例化bean

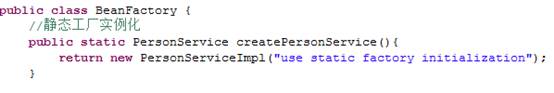
<bean id="PersonServiceBean" class="com.service.PersonServiceImpl"> 默认情况下在Spring加载时，会通过调用PersonServiceImpl中的构造器来实例化bean对象。

注意：使用这种方式来实例化bean时，bean必须包含默认的无参数构造器。

（2）使用静态工厂方法实例化bean

首先需要有一个工厂类PersonFactory，并且这个工厂类中有一个static的工厂方法用于返回bean对象。

如：



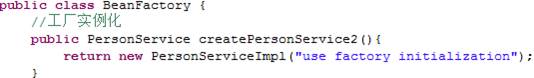
此时配置文件中配置：工厂类的class和工厂静态方法factory-method

ip_image004.jpeg

通过这个静态工厂的bean id就可以get到PersonServiceImpl对象

（3）使用实例工厂实例化bean

定义工厂类，并提供一个返回bean对象的工厂方法，不一定要static的，如



在配置文件中先定义工厂类的bean

ip_image008.jpeg

再定义PersonService 的bean 并指定 factory-bean 和 factory-method

ip_image010.jpeg

此处factory-bean就是上面为工厂类定义的bean的id。

总结：第一种默认实例化bean的方式是最为常用的，但使用工厂类实例化的好处在于可以自己编写对象创建过程的代码，而不仅仅局限于构造器中的内容。

**4.bean的实例化原理**

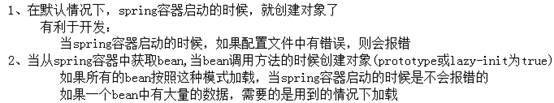
（1）默认情况下，Spring容器加载时配置在配置文件中的bean就会被实例化，且这些bean都是单例的。也就说不管调用多少次getBean()得到的都是同一个bean。

（2）为bean加上scope="prototype"的属性以后，bean不再在Spring加载时就初始化，而是在调用getBean方法时才会实例化，且是多例的。每次得到的getBean都是不同的对象。当bean被定义为多例时，无论lazy-init的值，都是调用getBean才会实例化。

 scope  的值：

|  |  |
| --- | --- |
| [singleton](http://www.redsaga.com/spring_ref/2.0/html/beans.html#beans-factory-scopes-singleton) | 在每个Spring IoC容器中一个bean定义对应一个对象实例。 |
| [prototype](http://www.redsaga.com/spring_ref/2.0/html/beans.html#beans-factory-scopes-prototype) | 一个bean定义对应多个对象实例。 |
| [request](http://www.redsaga.com/spring_ref/2.0/html/beans.html#beans-factory-scopes-request) | 在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例，它们依据某个bean定义创建而成。该作用域仅在基于web的SpringApplicationContext情形下有效。 |
| [session](http://www.redsaga.com/spring_ref/2.0/html/beans.html#beans-factory-scopes-global-session) | 在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的SpringApplicationContext情形下有效。 |
| [global session](http://www.redsaga.com/spring_ref/2.0/html/beans.html#beans-factory-scopes-global-session) | 在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |

（3）Spring什么时候创建对象？



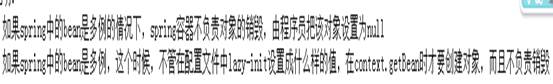
当将bean中属性lazy-init设置为true时，bean的初始化会被延时到getBean时候。

ip_image014.jpeg

lazy-init很少用。

（4）init-method , destory-method

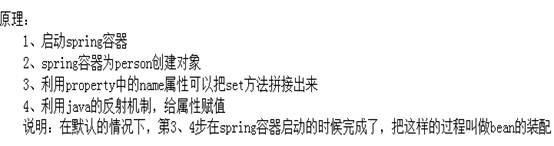




**5.依赖注入**

5.1 方式一：属性注入

步骤: 首先为javabean中的属性配置set，get方法，再在Spring配置文件中通过<bean>标签下的<property>标签来赋值。



（1）对于基本类型或者String类型的属性

ip_image022.jpeg

当注入基本类型的属性的值的时候，不论是int, float 还是 String类型的属性，value的值都是有引号的字符串。容器在注入时会自动完成类型转换。

（2）对引用类型属性的注入，使用ref

先定义引用类型的bean

ip_image024.jpeg

再通过这个bean的id，作为<property ref=""/>中的ref值注入到bean中。

ip_image026.jpeg

或者

ip_image028.jpeg

（3）对Set，List，Map类型属性的注入



（4）Properties类型



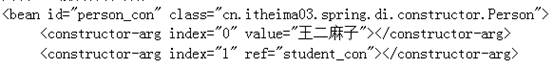
ip_image034.jpeg

5.2 方式二：构造器注入

<constructor-args>中有4个属性：



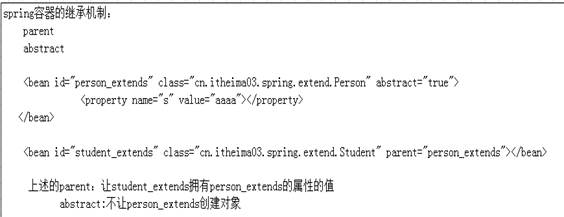
type一般可以省略，但当构造器存在重载的时候，type有时候需要写在构造器参数配置中。



ip_image040.jpeg

**6.Spring中继承抽象机制**

通过在bean中配置parent属性并指向另一个bean



**6.使用注解来实现依赖注入**

@Resource注解使用方法：

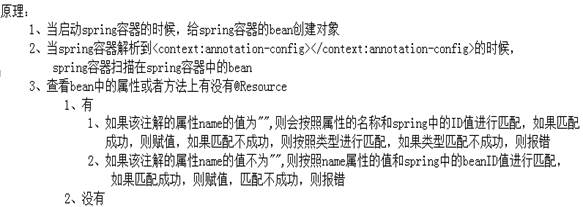
（1）配置文件中开启注解解析器：

ip_image044.jpeg

（2）此时bean中的属性不再需要配有get和set方法。这些属性不能是基本类型变量，必须是引用变量。

（3）但是bean中属性上需要使用@Resource注解,并可以选择性的配上name属性，如：

ip_image046.jpeg



@Autowired 是直接按属性的类型与bean类型进行匹配。

**7.类扫描机制**@Component

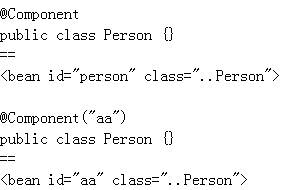
通过在类上使用@Component注解来省去在Spring配置文件中配置bean的过程。

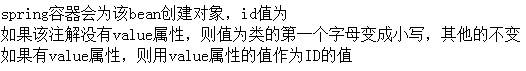
方法：

（1）在配置文件中加入解析器，此解析器不仅包含注解解析器还包含类解析器

ip_image050.jpeg

（2）在类上标注注解





一旦使用了@Component来自动为类创建bean对象，这个类中属性将不能通过<property>来注入值。对于引用类型可以使用@Resource(name="..")来注入。但是对于基本类型成员变量，则无法赋值。

更多典型化注解：

     @Controller:用于注释作为controller的bean

     @Service：用于注释作为service的bean

     @Repository：用于注释作为dao的bean

用法同@Component

注意：使用的注解越多，在xml文件中写的内容越少，但是这样反而效率会越低

**8.代理模式AOP**



代理对象中包含了对目标对象中方法的调用，只不过在此基础上，代理对象中又加入了一些额外的方法。使用代理对象目的是是目标方法在被调用前先调用一些额外的方法进行业务判断。

JDK中代理方法分为静态代理和动态代理，现在企业中多用动态代理。JDK中，Proxy可以帮助动态的创建代理对象。使用Proxy有个前提，就是目标对象必须实现了接口，只有目标对象是面向接口时才能使用Proxy。Proxy代理对象和目标对象实现了共同的接口。

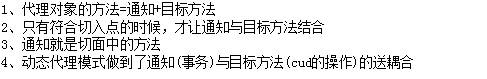
如果目标类没有实现接口的话，这个时候需要用到第三方的框架cglib来创建代理对象。此时，代理对象是目标对象的子类。

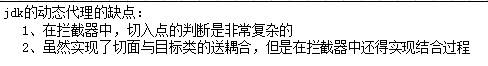
**Spring中的AOP**

切面：切面就相当于是一个代理对象

通知：通知就是切面中的各种方法

切入点：是对目标对象的判断，用来决定是否为该bean生成代理对象





Spring中使用AOP的方法

（1）现将目标类和切面类的bean配置在Spring 的配置文件中

（2）配置切面



<aop:config>中可以配置多个<aop:aspect>切面，且这些切面中的通知可以指向同一切入点。

切面中的通知类型：

（1）前置通知：<aop:before/>

ip_image064.jpeg

ip_image066.jpeg

（2）后置通知：<aop:after-returning/> 指定 returning属性

ip_image068.jpeg

ip_image070.jpeg

后置通知通过配置returning来定义目标方法返回值的变量名，并作为参数传给后置通知。这样在后置通知方法中就可以使用目标方法返回值了。

（3）最终通知: <aop:after/>

ip_image072.jpeg

ip_image074.jpeg

（4）异常通知：<aop:after-throwing/>, 配置 throwing属性

ip_image076.jpeg

ip_image078.jpeg

当目标方法执行并抛出异常时，异常通知会执行。在异常通知中配置throwing属性来指定目标方法抛出的异常的变量名，并将其作为参数方式异常通知方法中，这样在异常通知就可以查看目标方法中抛出的异常的信息了。

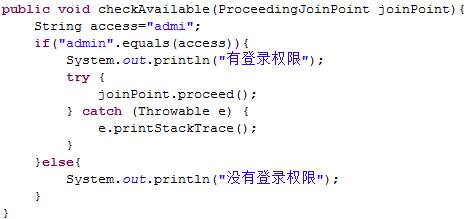
（5）环绕通知：<aop:around/>

ip_image080.jpeg

环绕通知中使用

ip_image082.jpeg

方法来调用目标方法，否则目标方法不会执行。



环绕通知需要接收 ProceedingJoinPoint作为参数。ProceedingJoinPoint: 连接点，客户端代理对象调用了哪个方法，哪个方法就是连接点。

环绕通知的意义：环绕通知可以控制目标方法的执行，前后置通知不可以。环绕通知中可以通过判断某个条件来控制目标方法的执行，例如权限判断。

切面中所有通知方法都可以有一个JoinPoint参数，可以通过调用API获取连接点的一些信息。ProceedingJoinPoint是JoinPoint的接口。

**Spring AOP工作流程**

1.根据xml配置文件，将所有的bean实例化

2.根据<aop:cut-point/>中expression的配置，对所有符合切入点表达式的bean创建代理对象。

3.再使用set方法，将各bean中的依赖对象注入。此时，如果注入的对象bean存在代理对象，则此时注入的就是代理对象。

例：

MVC架构中，对所有的service层对象匹配了切入点，则为他们生成了代理对象。此时当某个action中对某个service层对象进行调用时，实际上调用的代理对象，一系列配置的通知也会在目标方法调用时被执行。

使用SpringAOP统一处理异常方法，定义一个异常处理切面，配有异常通知。将service层对象作为切入点，则当service层方法抛出异常对象时，该异常处理切面会进行统一处理。

SpringAOP注解形式

