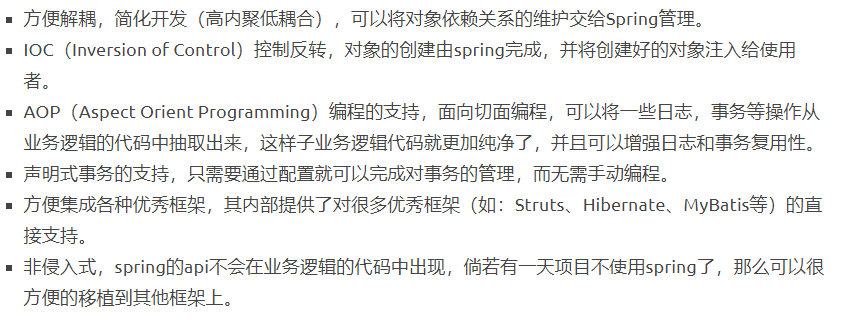
**Spring简介**

Spring 是个Java企业级应用的开源开发框架。Spring主要用来开发Java应用，但是有些扩展是针对构建J2EE平台的web应用。Spring框架目标是简化Java企业级应用开发，并通过POJO为基础的编程模型促进良好的编程习惯。

**Spring的优点**

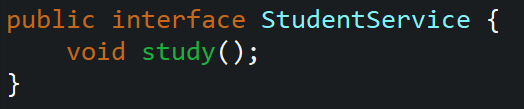


**编写Spring程序**

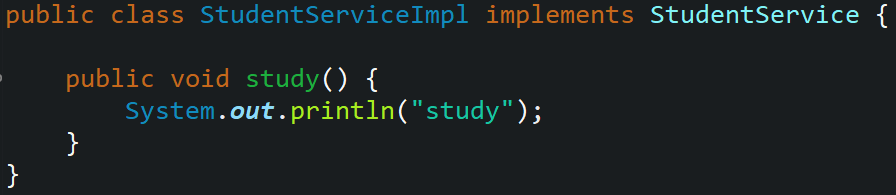
**1、导入jar包：spring-context**

**2、创建接口和实现类**

**接口**

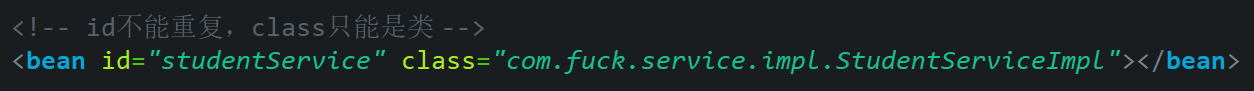


**实现类**



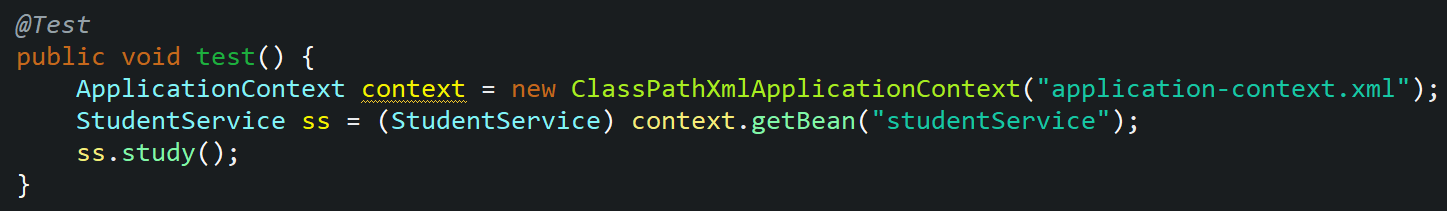
**3、添加spring的配置文件**

文件名可以随意命名。



**4、运行程序**

想要使用StudentService的话，需要开发者自己手动通过new关键字创建该接口实现类的对象。虽然使用了接口可以实现程序的解耦，但是实际上在代码中还是有new StudentServiceImpl的语句，这个地方还是存在一些耦合的。



**控制反转（IoC，Inversion of Control）**

**控制反转是一种思想。**指的是将创建对象的操作权交给容器，通过容器来装配和管理对象的创建，**控制反转其实就是对这些对象控制权的反转，控制权由程序本身反转给了外部容器。**

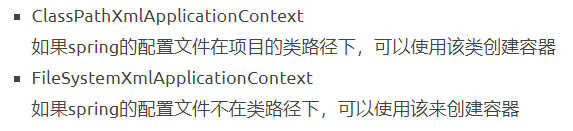
**依赖注入(DI，Dependency Injection)**

ioc是一种思想，有一些实现方式，其中较为常用的一种是依赖注入。**依赖注入是指程序运行过程中，若需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部容器，由外部容器创建后传递给程序。**依赖注入让Spring 的Bean之间以配置文件的方式组织在一起，而不是以硬编码的方式耦合在一起。在之前的第一个程序中实际上就是使用了依赖注入这种方式获取的StudentServiceImpl对象。

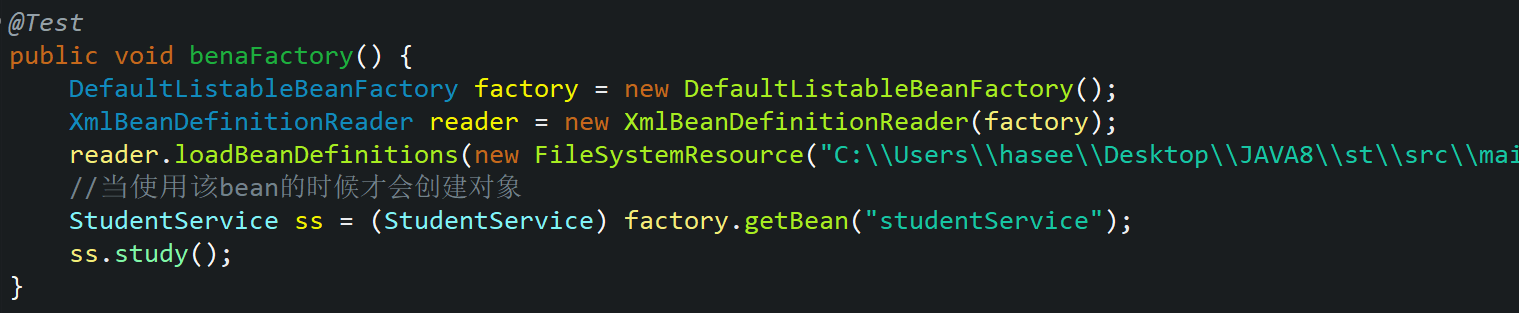
**IOC容器**

**ApplicationContext接口**

通常使用ApplicationContext充当spring IOC容器的角色，ApplicationContext使用了反射的方式创建bean对象，并且在读取配置文件之后将里面注册的bean全部创建对象。



**BeanFactory接口**



**用ApplicationContext还是BeanFactory**

ApplicationContext提供了比BeanFactory更多的功能，在实际工作中除非特殊情况，**例如需要在一些小内存或者资源受限的机器运行程序，这个时候可以选择使用BeanFactory，此时事务的管理、AOP功能将会失效**。因此，大部分情况下，建议使用ApplicationContext作为容器。

**bean的装配**

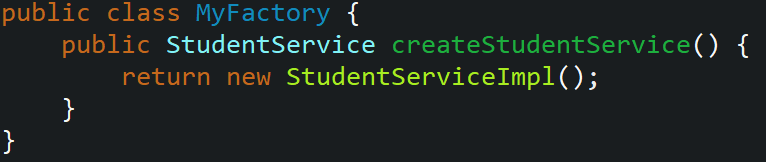
spring ioc容器会创建我们配置好的bean的对象，创建对象的方式有三种，分别是默认方式，实例工厂，静态工厂。

**1、默认方式（常用）**

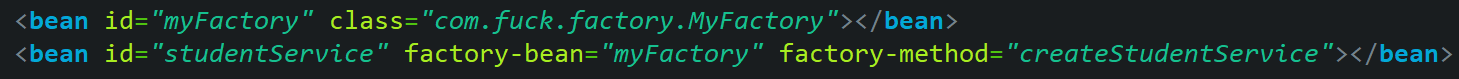
我们之前写的例子就是使用的默认方式，这种方式spring ioc容器会调用bean的无参构造方法来创建对象，所以此时**务必要保证这些bean有无参构造方法**。

**2、实例工厂**

**先定义一个工厂类，在该类中的方法都是非静态的**

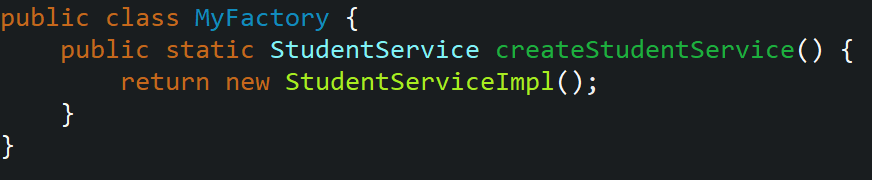


**在配置文件中告诉spring要使用的工厂类和方法。**



**3、静态工厂**

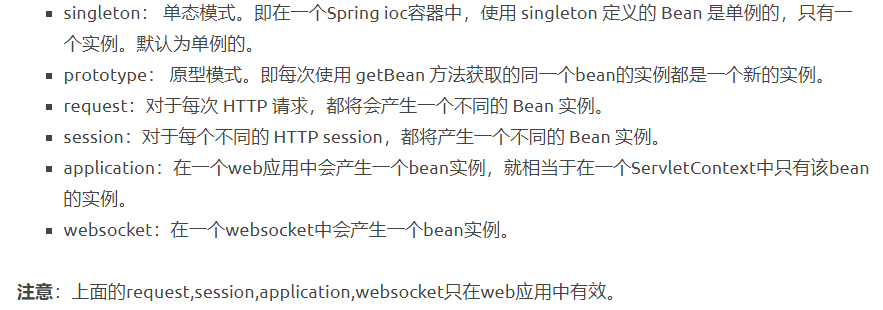
**先定义一个工厂类，里面的方法是static修饰的**



**在配置文件中告诉spring要使用的工厂类和方法。**



**bean的作用域**

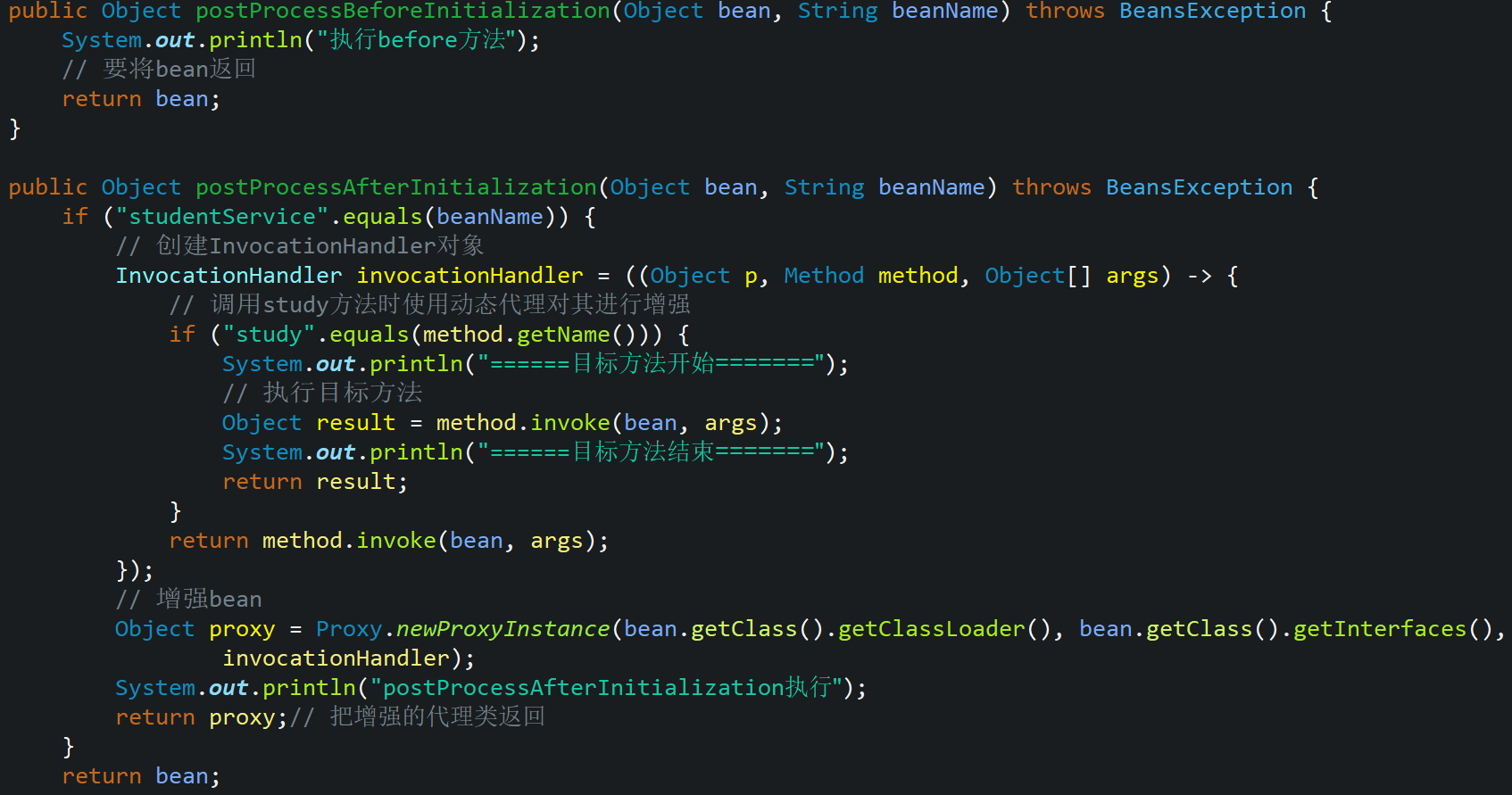




**BeanPostProcessor接口（不常用）**

实现了BeanPostProcessor接口的类被称为bean后处理器，在bean中方法的之前和之后会自动调用bean后处理器的两个方法，可以通过在这两个方法中编写代码来增强或扩展一些功能。

**创建一个类实现BeanPostProcessor接口，通过jdk自带的动态代理来实现对bean功能的增强**



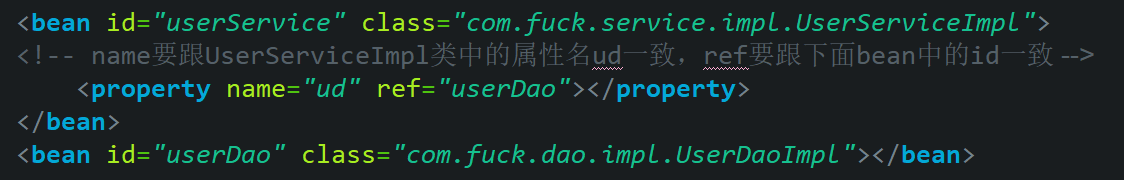
**依赖注入**

spring ioc容器初始化好bean的实例对象之后，会对该对象中的属性进行初始化，初始化的过程依然是由容器自动来完成，这个被称为是**依赖注入**。spring里面常用的注入方式有三种，**setter方法注入，构造方法注入，基于注解的注入**。还有一种方式是实现特定接口注入。由于这种方式采用侵入式编程，污染代码，所以已经不用了。

**setter方法注入**

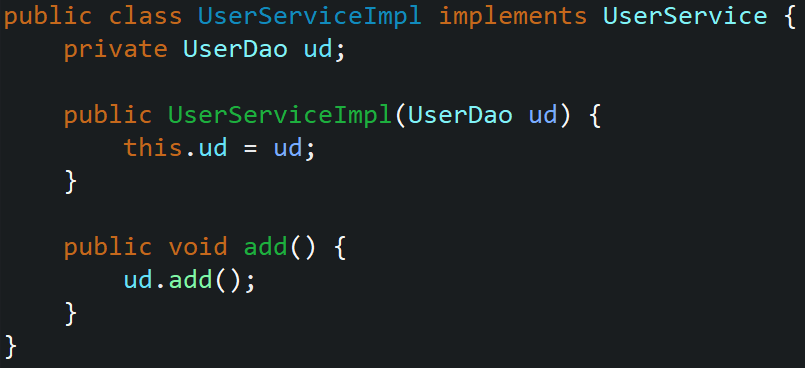
容器通过调用setter方法将对象注入，这种方式比较简单，所以使用的概率比较高。





**构造注入**

构造注入是容器通过构造方法将实例化的对象进行注入。

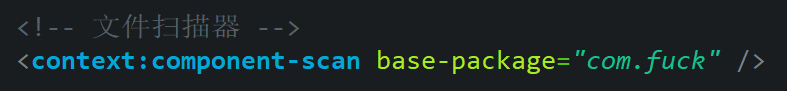




**基于注解的依赖注入（常用）**

**@Component、@Repository、@Service、@Controller**

随着bean的增多，spring的配置文件肯定会越来越臃肿，因此spring引入了注解。使用注解实现依赖注入的话，就不需要在applicationContext.xml中注册bean了，添加一个文件扫描器即可。



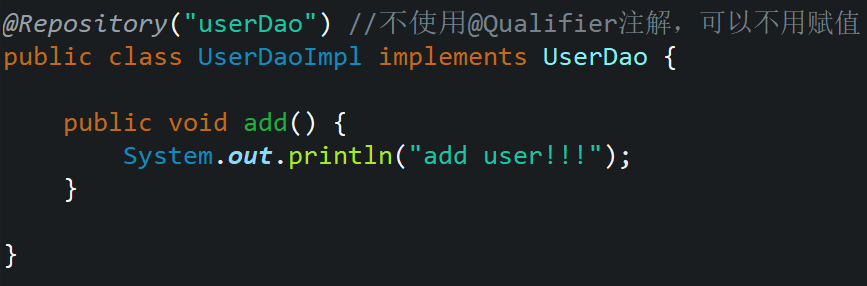
**spring中还提供了跟@Component等效的注解，通常情况下，我们会使用下面注解来代替@Component：**

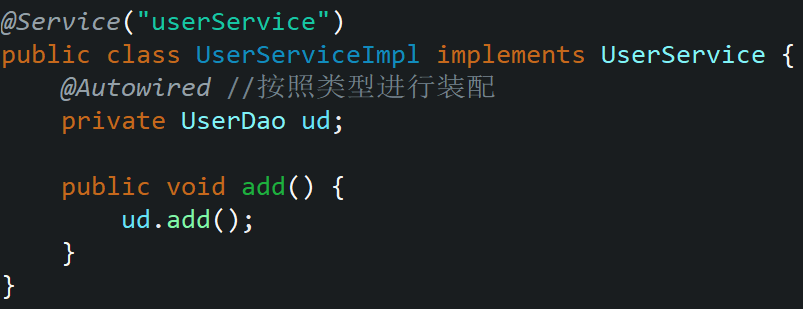
@Repository 用于对 DAO 实现类进行注解

@Service 用于对 Service 实现类进行注解

@Controller 用于对 Controller 实现类进行注解

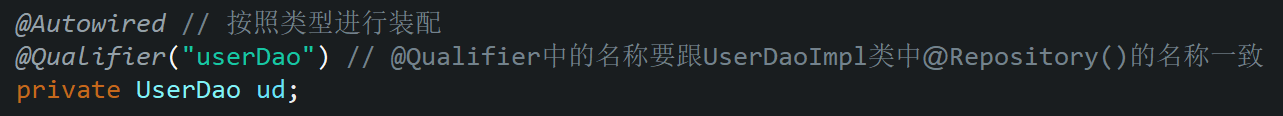
**使用上面这三个注解可以被spring更好地处理和与切面进行关联。**





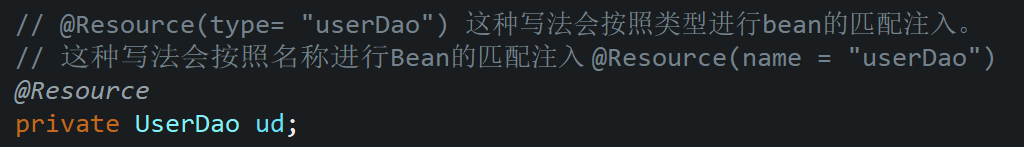
**@Autowired注解**

可以在UserDao的属性上面添加@Autowired注解，该注解默认使用按类型自动装配，即容器会查找UserDao类型的实例将其注入。还可以配合@Qualifier注解让@Autowired根据名称自动装配。



**@Resource注解**

@Resource是javax.annotation包下提供的，即该注解不是spring提供的，如果使用该注解的话需要保证你的jdk版本是6以上，也可以使用该注解替代@Autowired。



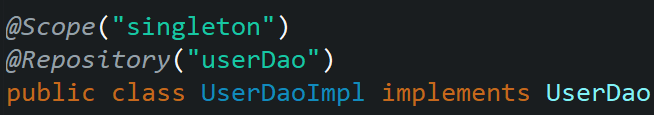
**@Autowired和@Resource的区别**

**@Autowired是spring提供的，@Resource是java提供的**，这两种注解在实际开发中二选一即可。

**@Autowired默认按类型进行自动装配，@Resource默认按对象的名称进行装配。**

**@Scope注解**

可以通过@Scope注解来指定bean的作用域，默认是singleton的。



**注解方式和xml方式**

注解的好处是，配置方便，直观。但其弊端也显而易见，以硬编码的方式写入到了Java代码中，其修改是需要重新编译代码的。

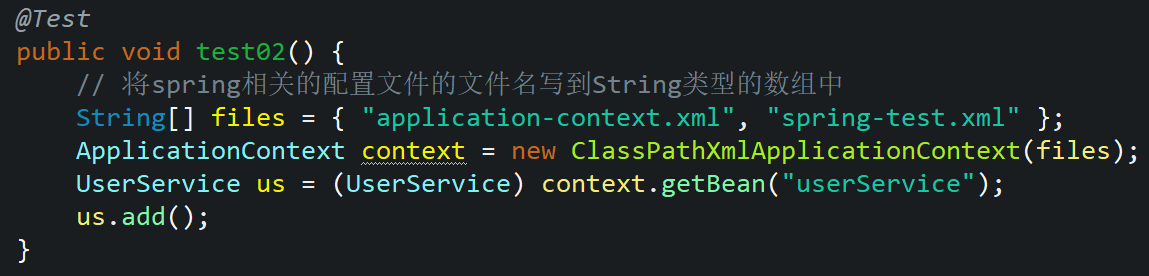
XML配置方式的最大好处是，对其所做修改，无需编译代码，只需重启服务器即可将新的配置加载。

**若注解与XML同用，XML的优先级要高于注解。**这样做的好处是，需要对某个Bean做修改，只需修改配置文件即可。当然，此时Bean类要有setter或构造器。

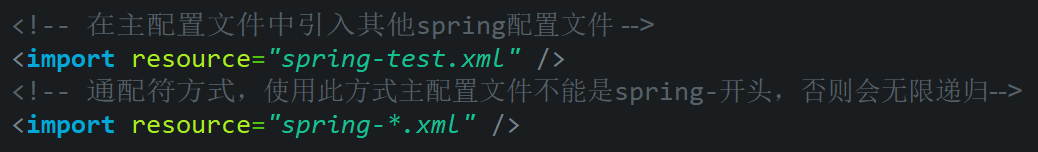
**多个 Spring 配置文件**

在实际应用里，随着应用规模的增加，系统中Bean数量也大量增加，导致配置文件变得非常庞大、臃肿。为了避免这种情况的产生，提高配置文件的可读性与可维护性，可以将Spring配置文件分解成多个配置文件。

**1、使用String类型数组**



**2、在配置文件中导入**



**AOP（Aspect Orient Programming）**

面向切面编程，是面向对象编程OOP的一种补充。**在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。**例如转账功能，在转账代码的前后需要一些非业务方面的处理，权限控制，记录日志，事务的开启与结束，这些代码就可以使用AOP将其切入到转账代码的前后，这样就可以很好地分离业务代码和非业务代码。

**AOP的优点就是降低代码之间的耦合，提高代码的复用性。**

**spring底层就是采用动态代理模式实现AOP的。**

**两种代理**

**JDK 的动态代理**

如果被代理了实现了接口，会默认使用jdk的动态代理。

**CGLIB的动态代理**

如果类没有实现接口，会使用CGLIB动态代理。

**两种方式的优缺点**

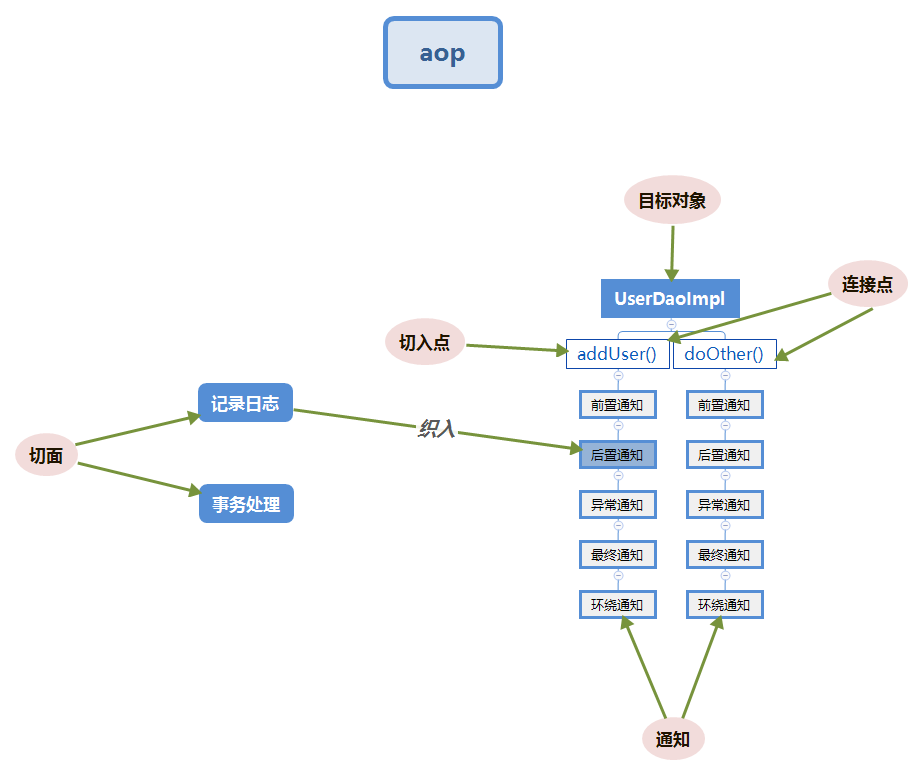
**从性能上讲**

使用字节码处理的CGLIB要比使用反射的JDK动态代理好。

**从耦合度上讲**

jdk要好于额外需要依赖字节码处理框架ASM的CGLIB。

**AOP的术语**

****

**目标对象（Target）**

目标对象指**将要被增强的对象**。即包含主业务逻辑的类的对象。上例中的UserDaoImpl 的对象若被增强，则该类称为目标类，该类对象称为目标对象。当然，不被增强，也就无所谓目标不目标了。

**切面（Aspect）**

切面**泛指非业务逻辑**。上例中的事务处理、日志处理就可以理解为切面。常用的切面有通知，实际就是对业务逻辑的一种增强。

**连接点（JoinPoint）**

连接点指**可以被切面织入的方法**。通常业务接口中的方法均为连接点。

**切入点（Pointcut）**

切入点指**切面具体织入的方法**。在UserDaoImpl类中，若addUser()被增强，而doOther()不被增强，则 addUser()为切入点，而doOther()仅为连接点。**被标记为final的方法是不能作为连接点与切入点**的，因为是不能被修改的，不能被增强的。

**通知（Advice）**

通知是切面的一种实现，可以完成简单织入功能（织入功能就是在这里完成的）。换个角度来说，通知定义了增强代码切入到目标代码的时间点，是目标方法执行之前执行，还是之后执行等。通知类型不同，切入时间不同。切入点定义切入的位置，通知定义切入的时间。

Advice有下面几种，这里使用常用的AspectJ方式：

**前置通知（Before advice）：在连接点之前执行，即目标方法执行之前执行。**

**后置通知（After returning advice）：在连接点正常结束之后执行，如果连接点抛出异常，则不执行。**

**异常通知（After throwing advice）：在连接点抛出异常后执行**

**最终通知（After (finally) advice）：在连接点结束之后执行，无论是否抛出异常，都会执行。**

**环绕通知（Around advice）：在连接点之前和之后均执行。**

**织入（Weaving）**

织入是指将切面代码插入到目标对象的过程。

**aop代理（AOP proxy）**

spring中的aop代理有两种：jdk自带的动态代理和CGLIB代理。

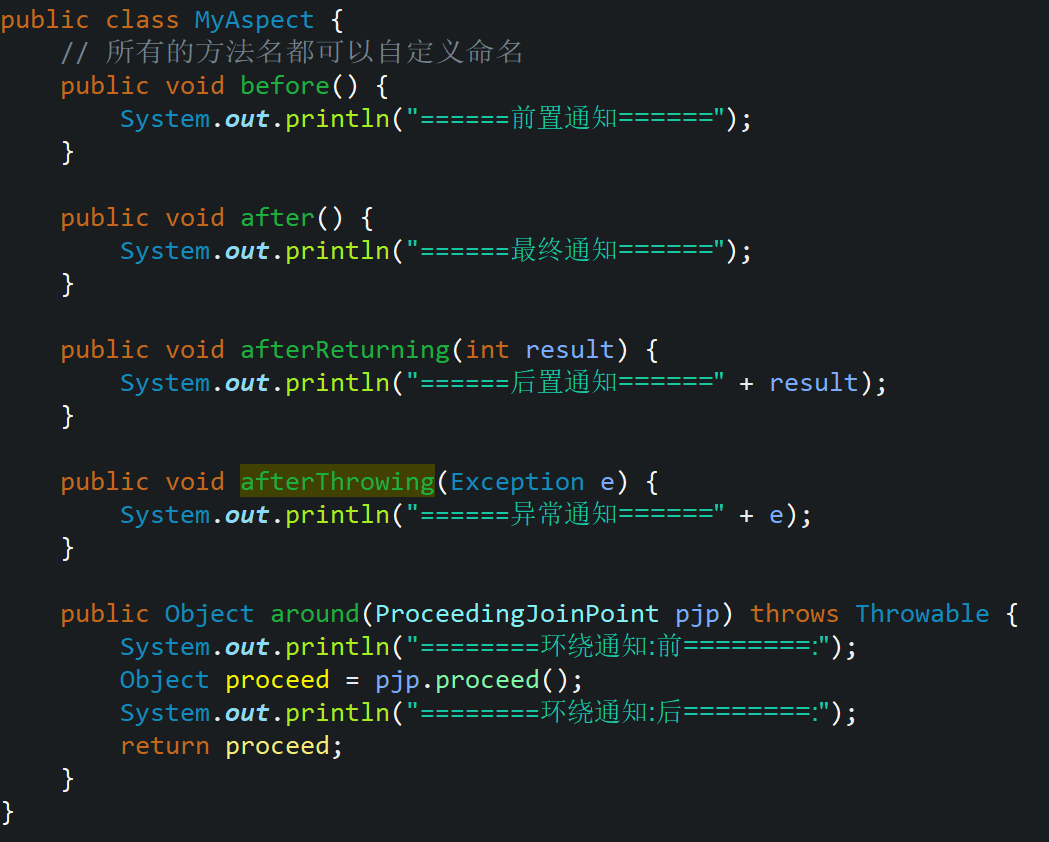
**AspectJ 对 AOP 的实现**

对于AOP这种编程思想，很多框架都进行了实现。Spring就是其中之一，可以完成面向切面编程。AspectJ这个框架也实现了AOP的功能，且其实现方式更为简捷，使用更为方便，而且还支持注解式开发。所以，Spring2.0版本中将AspectJ的对于AOP的实现也引入到了自己的框架中，不过织入仍然还是使用的spring的aop完成。**在Spring中使用AOP开发时，一般使用AspectJ的实现方式。**

**基于xml实现AOP的方式**

**1、导入jar包：aspectjweaver，spring-aspects**

**2、创建切面类**



**3、spring的配置文件**



配置文件中，除了要定义目标类与切面的Bean外，最主要的是在aop:config中进行aop的配置。而该标签的底层，会根据其子标签的配置，生成自动代理。

通过其子标签aop:pointcut定义切入点，该标签有两个属性，id与expression。分别用于指定该切入点的名称及切入点的值。expression的值为execution表达式。

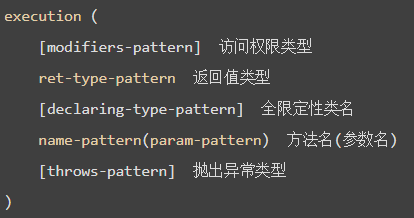
aop:aspect的ref属性用于指定使用哪个切面。

aop:aspect的子标签是各种不同的通知类型。不同的通知所包含的属性是不同的，但也有共同的属性。

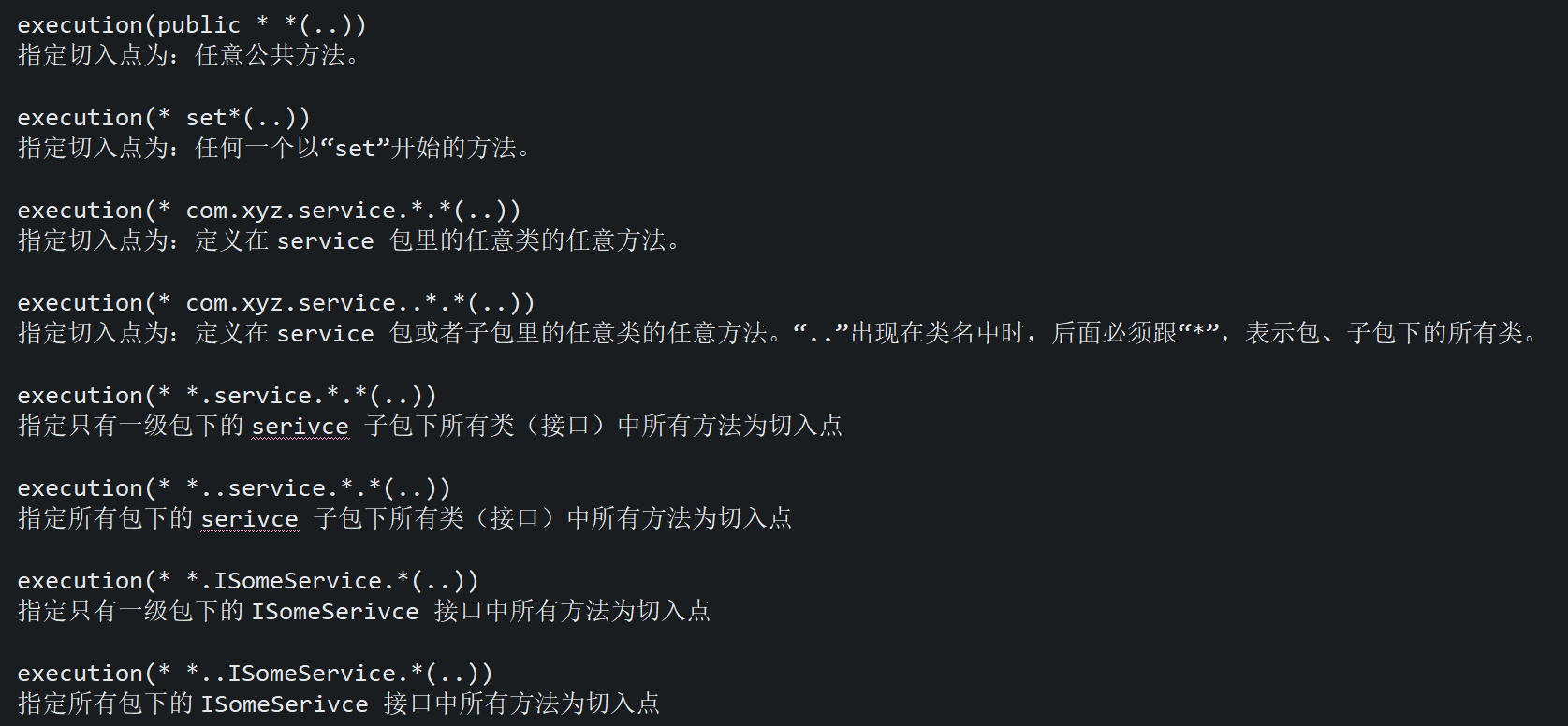
method：指定该通知使用的切面中的哪个方法。

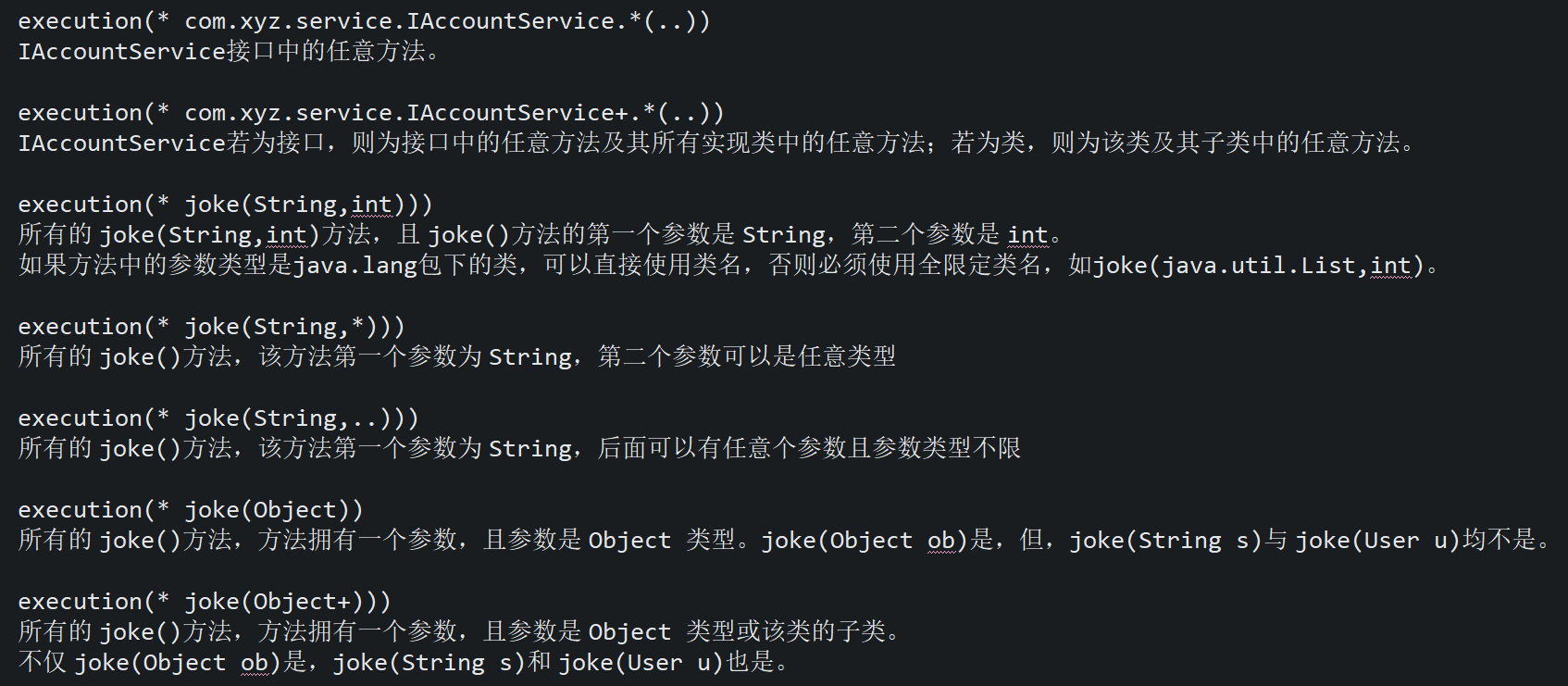
pointcut-ref：指定该通知要织入的切入点。

**AspectJ的切入点表达式**



切入点表达式要匹配的对象就是目标方法的方法名，顺序不能变。所以，execution 表达式中明显就是方法的签名。注意，表达式中加[ ]的部分表示可省略部分，各部分间用空格分开。





**基于注解的AOP的实现方式**

