## 一 spring框架概念

### **一** 框架概述：

①：Spring是一个轻量级的开源javaee应用程序框架，他可以独立进行使用，也可以和其他框架进行联合使用。

②：目的在于解决企业开发的复杂性！

③：Spring有（不局限于）两大核心的部分：ioc和aop

1. ioc：控制反转，我们把创建对象的权力交给spring而不是我们自己把握
2. aop：面向切面，在不修改源代码的情况下进行功能增强（代理模式）

④：Spring框架主要有其部分：spring core，spring aop，spring orm，spring dao，spring context，spring web，spring web mvc。

### **二 框架特点**

①：方便解耦，简化开发（ioc降低耦合性）

②：aop编程的支持

③：整合了junit，方便测试

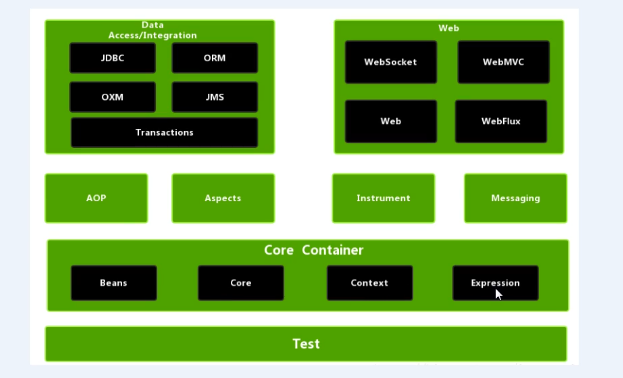
④：集成其他框架非常方便，很优秀

⑤：降低java api等的操作，例如对jdbc进行了封装

⑥：提供了事务管理

⑦：设计思路非常优秀，学习模板

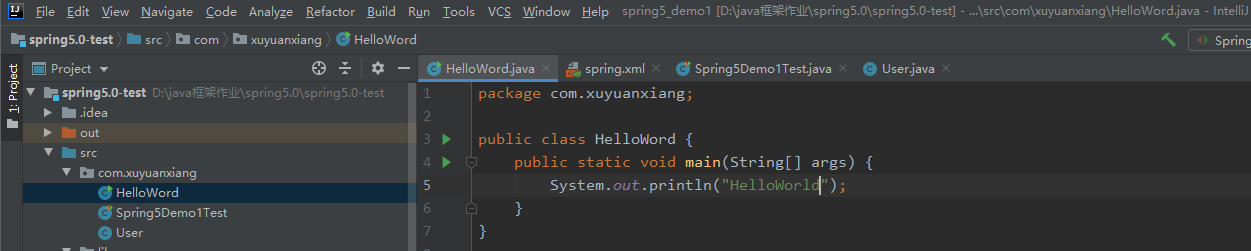
## 二 spring框架入门案例



上面便是spring5.0的所有内容

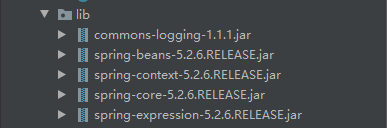
接下来演示spring的基本功能：

首先创建一个项目，暂时创建一个java基本项目就好了：

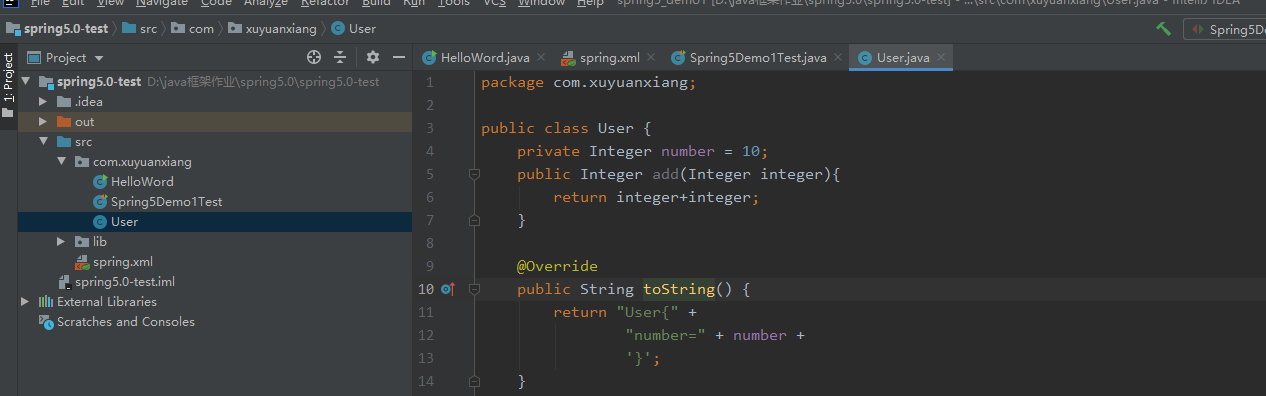


然后创建一个名为lib的包，把需要的jar包导入：

这里的四个包对应我们前面图片的四个core，另外一个是log包，也是必须的！

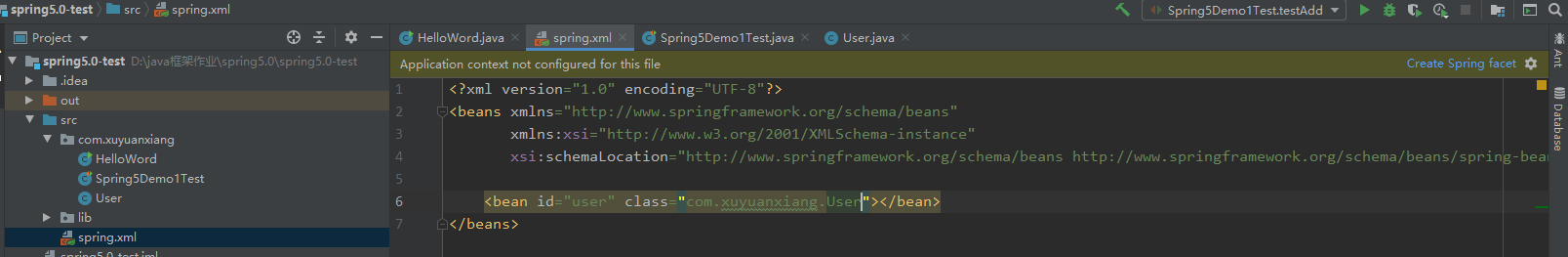


创建一个user类：

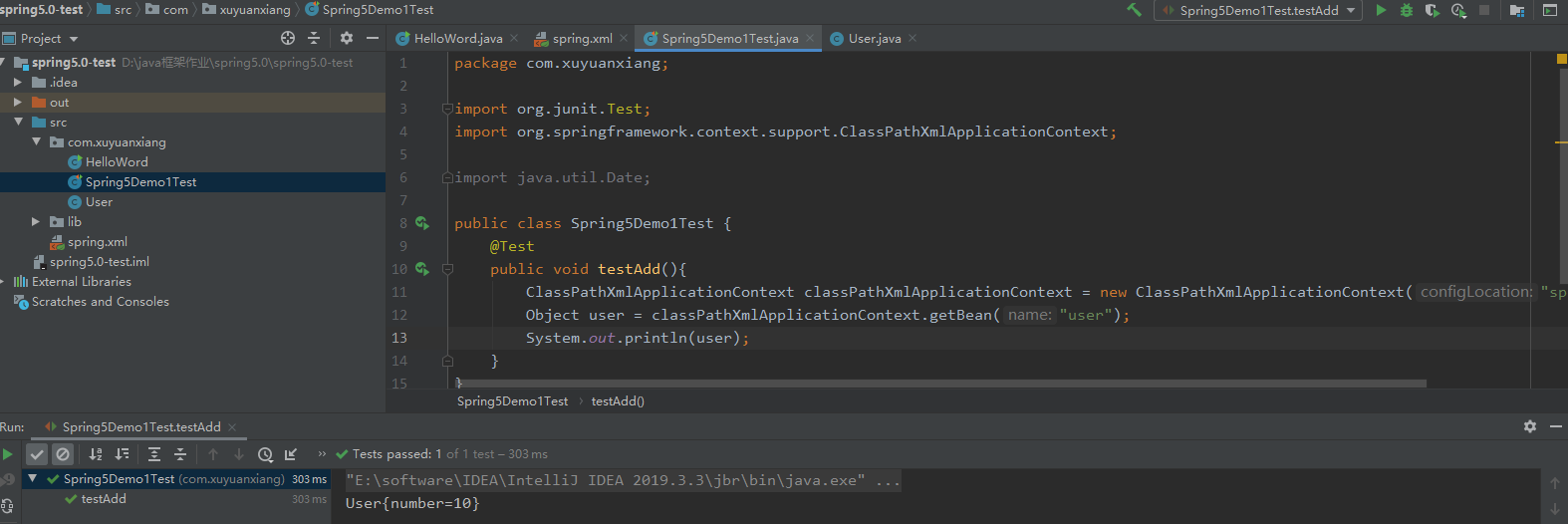


根路径（src）下创建一个spring.xml文件（名字无所谓，但是要一眼知道功能）

里面写入一个<bean>标签，创建一个user类交给spring容器



最后编写一个测试类进行测试，我们在这里没有采用new的方式却获得了user的实体类，利用的就是spring的ioc控制反转技术！当然这里的读取spring配置文件的方法有两个：一个是我们这个直接读取类路径（src）的，另外一个是读取指定路径的（FileSysterm\*\*\*\*\*Context）。



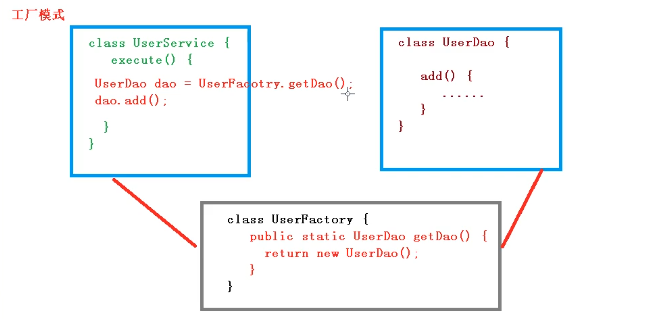
## 三 spring-ioc容器

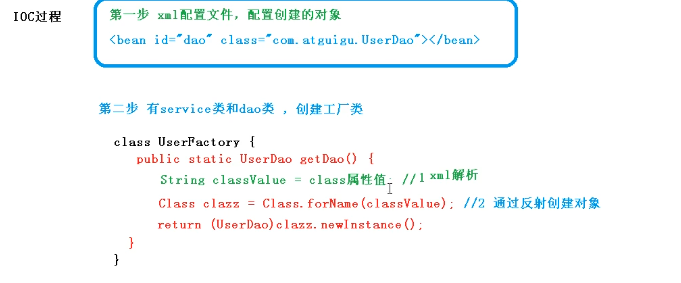
ioc控制反转实际上目的是解决程序之间的耦合度过高的问题：（耦合度低，我们修改代码的时候会非常容易）

ioc的底层原理分别是：xml解析，反射，工厂模式！

类似的解决方法有工厂模式：（但是工厂模式引入了新的耦合~！）

### 一：解耦的模式：





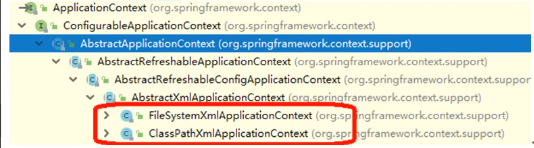
IOC的容器底层就是对象工厂

### 二：Spring提供的ioc容器实现的方式有两种：

①：BeanFactory：ioc容器的基本能实现，是spring内部使用的接口，一般不提供给开发人员！（特点是加载文件的时候是不会创建对象的，只有在使用的时候才会创建对象！（懒加载））

②：ApplicationContext：是BeanFactory的子接口，具有更加强大的功能！提供给开发人员使用！（特点是加载配置文件的时候之间创建对象！）这种操作我们建议在启动的时候就完成了！节约项目运行时间

### 三：ApplicationContext接口的实现类



此界面由control+h打开

第一个是类路径（就是src 下的路径）

第二个是系统盘符路径（在c或者d盘中的具体路径）

## 四 bean管理xml方式（创建对象和set注入属性）

Bean管理就是指

Spring对象创建

基于xml方式创建对象



在spring配置文件中指定<bean>标签，标签中添加属性

这里的id属性是此对象的唯一标识用于从容器中获取！

Class属性表示类的全路径

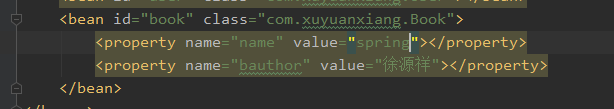
Name属性和id属性类似，但是name中可以写特殊字符，目前已经弃用！

在创建对象的时候会自动执行无参的构造方法！（必须要有无参构造方法，否则报错。当然我们也可以在xml配置文件中配置构造方法）

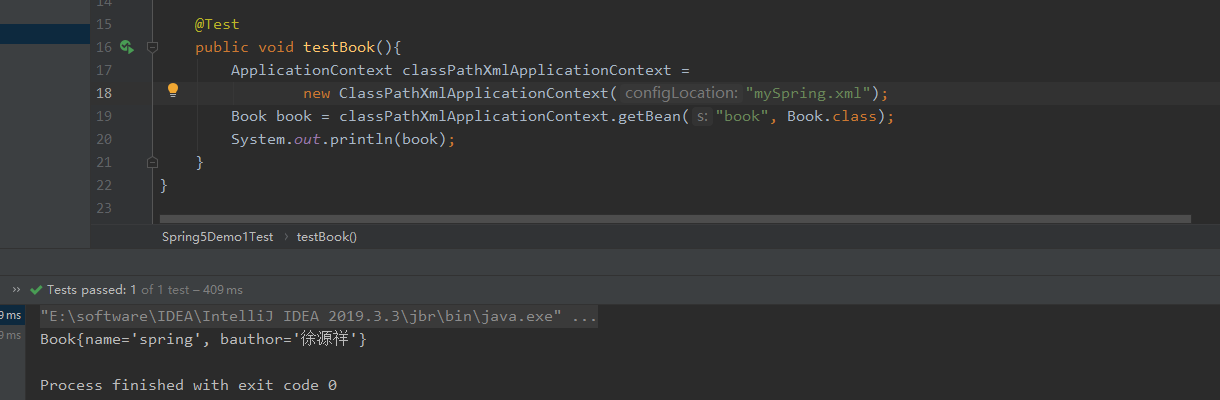
基于xml方式注入属性

Di：依赖注入，就是注入属性。（需要实体类中有set方法）

第一种注入方式（通过无参构造器先创建出对象，在调用set方法！）

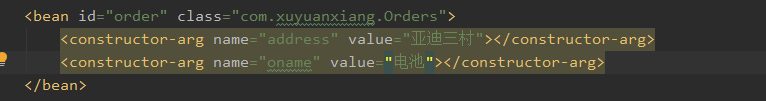


测试注入后的结果！

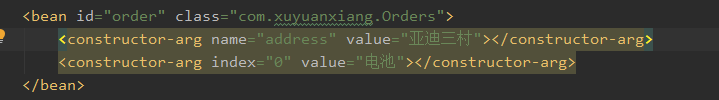


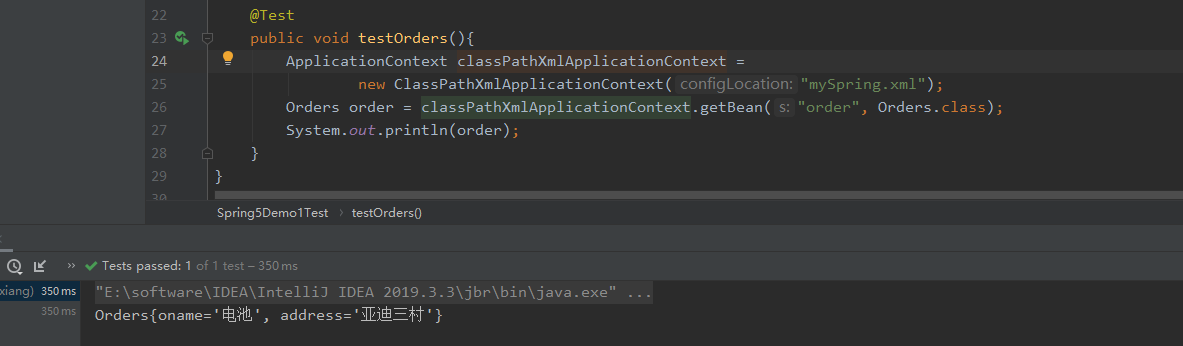
## 五 bean管理xml方式（有参构造方法注入属性）

第二种注入方式：（这种方式直接通过有参构造方法创建对象）



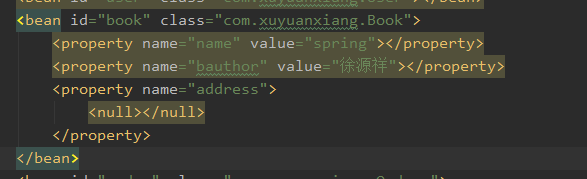
Index指定参数索引





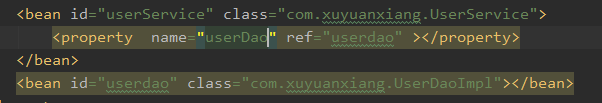
## 六 bean管理xml方式（注入空值和特殊符号）

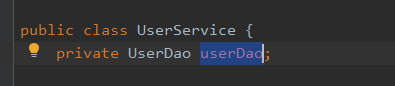
这样我们就可已将指定的参数注入为null，至于注入特殊符号就用到再差吧，太麻烦了



## 七 bean管理xml方式（注入外部bean）

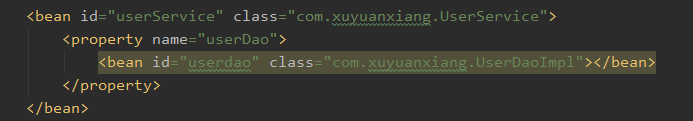
如此<bean>标签实现了外部bean的注入





## 八 bean管理xml方式（注入内部bean以及级联赋值）

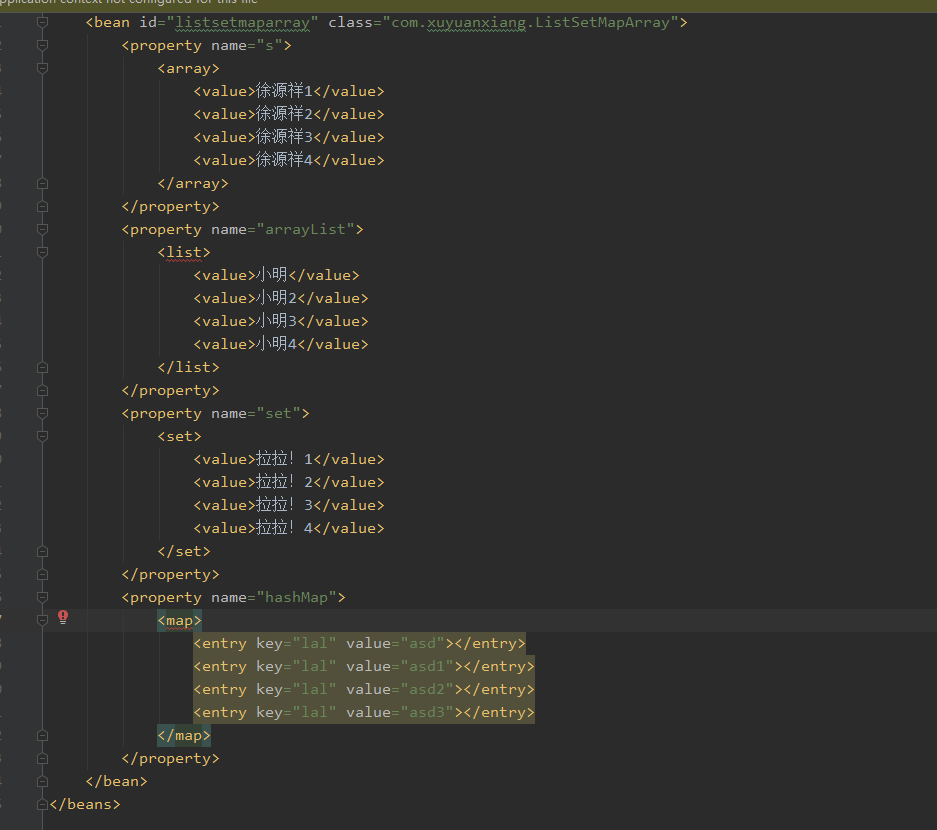
内部bean注入（个人觉的外部注入更清楚）



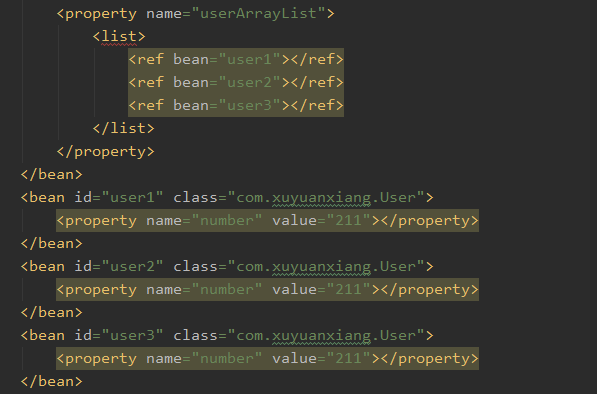
级联赋值：其实就是当我们的userDaoImpl中如果存在属性的话，依然可以继续进行<property>标签进行属性注入，这样就称作级联赋值！

## 九 bean管理xml方式（注入集合类型属性）

如下实现集合类型的注入

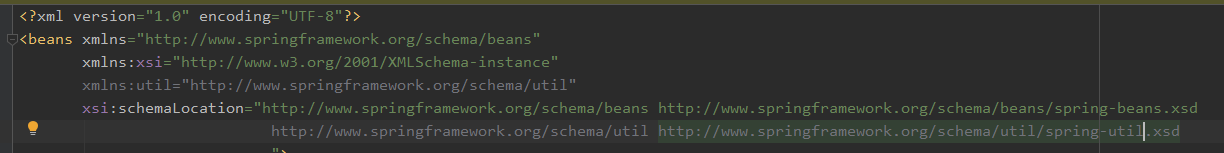


如果我们的集合中放置的不是普通类型而是我们的对象：可以采用如下方式，当然内置bean也可以

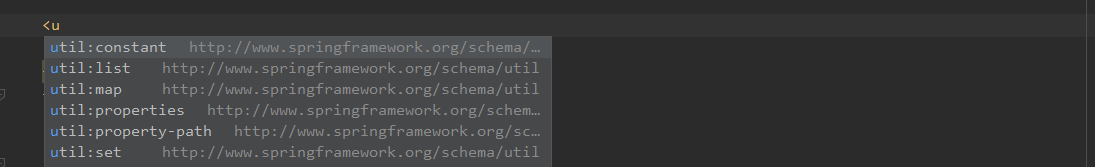


当我们有相同的配置的时候可以将公共的部分做成独立的部分，其他标签调用就好！

首先我们引入util的名称空间



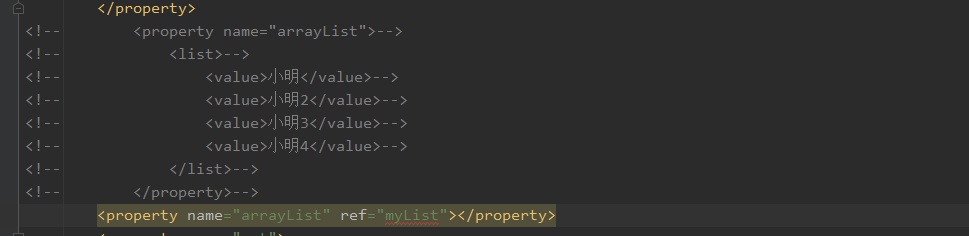
引入以后发现在输入标签的时候多了如下选择：



我们自定义一个list的标签



将原来的注入方式注释掉，采用了新的注入方式（直接用ref属性引导myList就好！）



## 十 bean管理xml方式（工厂bean【FactoryBean】）

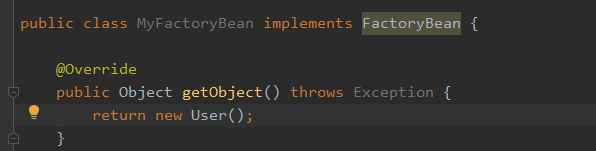
Spring有两种bean

一种是普通的bean（我们标签中写的class属性是什么类型就返回什么类型，以前的模式都是这样的）

一种是工厂bean（标签中class属性的类型可能和返回的类型不同）

这里我们演示第二种：

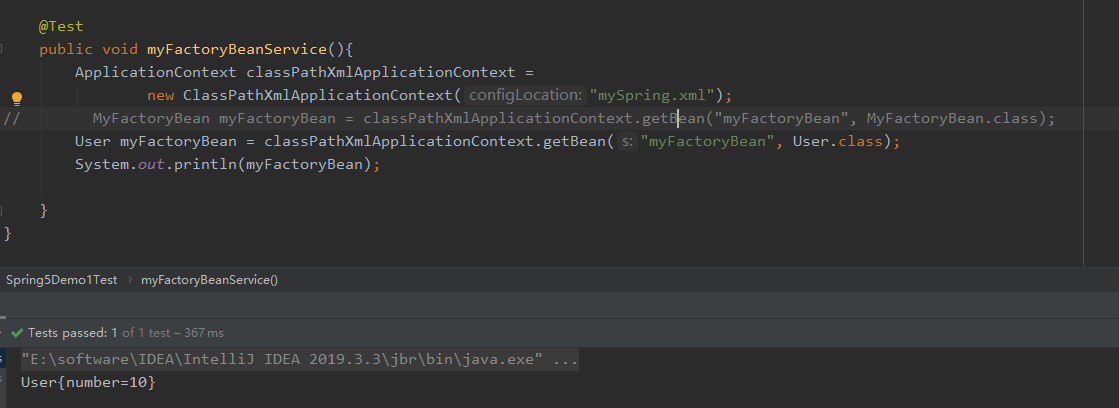
首先创建一个类继承FanctoryBean接口，实现其中的方法：



进行标签配置：



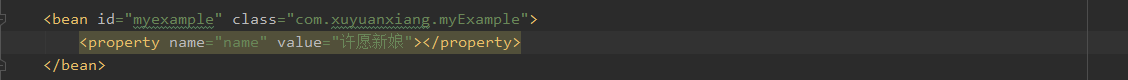
测试中发现原来的类转型已经无法成功了，我们只有转型成为user，并且输出也是user（这就实现了标签中的class属性和返回类型不统一的现象）

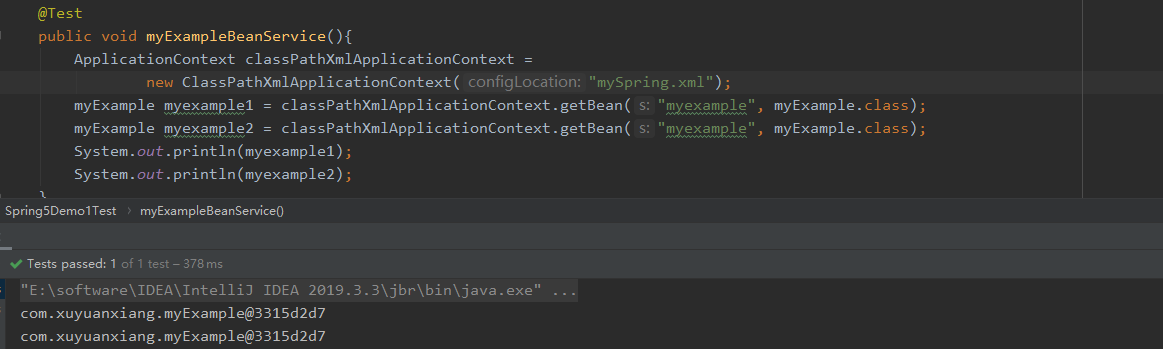


## 十一 bean管理xml方式（bean作用域）

我们通过bean标签创建出的实例，分为单例以及多例两种，我们默认产生的是单例的。

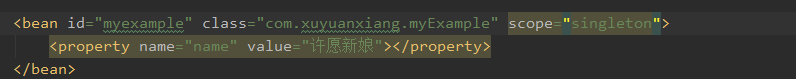
如下我们进行注入一个myexample的实例，结果发现在内存中的地址是相同的



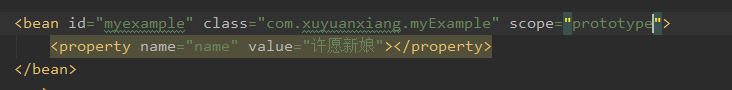


我们可以通过修改bean标签中的scope属性的值进行更改单例或者多例。

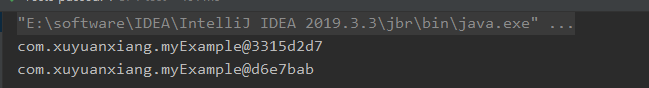
这是单例，不写也是默认这个值



这就是多例



此时输出的地址值已经不同了（而且是无线多例的，永不重复）



单例和多例非常不同的另一个方面在于：

单例是在加载配置文件的时候就产生的。

多例是在要使用对象的时候才会产生。

Scope有四个值，另外两个不常用（request，session）

Request就是在一次请求的作用域，session就是在一次会话的作用域！

以上就是所有的作用域！

## 十二 bean管理xml方式（bean的生命周期）

生命周期就是一个对象从产生到销毁的时间

Bean的生命周期：

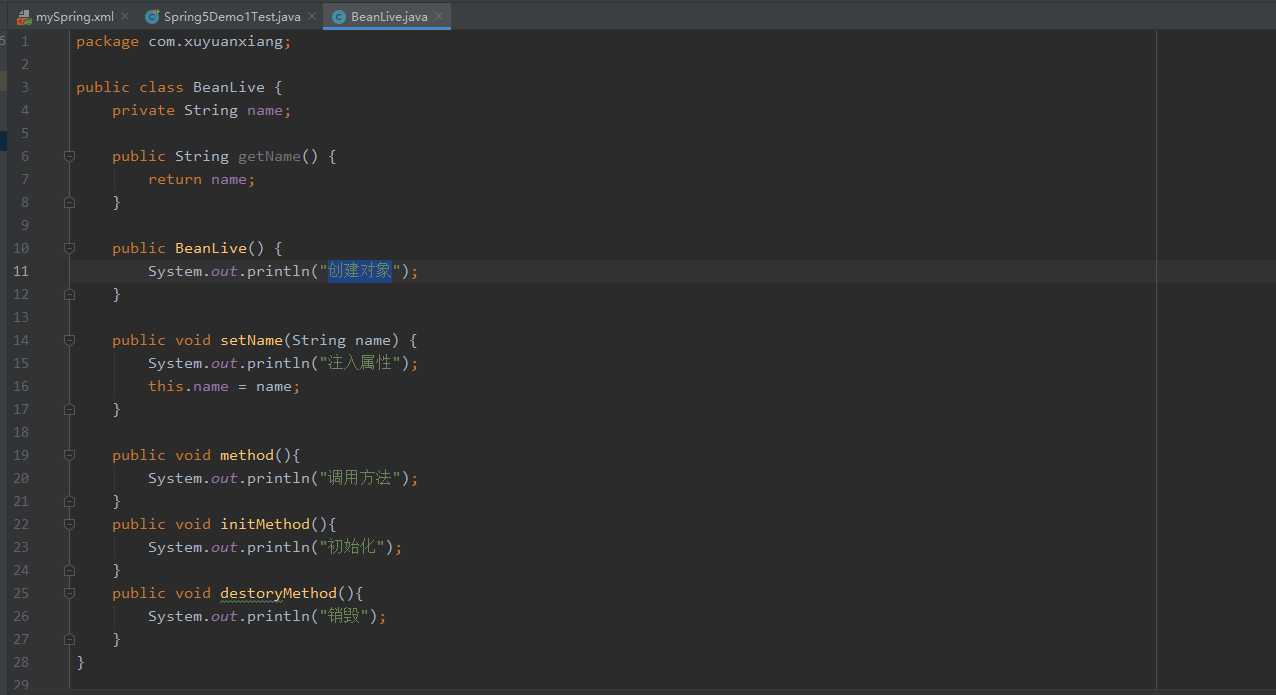
①：通过构造方法创建bean实例

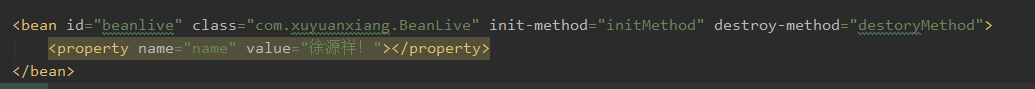
②：bean属性的注入（调用set）

③：调用bean的初始化方法（需要手动配置一个类的初始化方法）

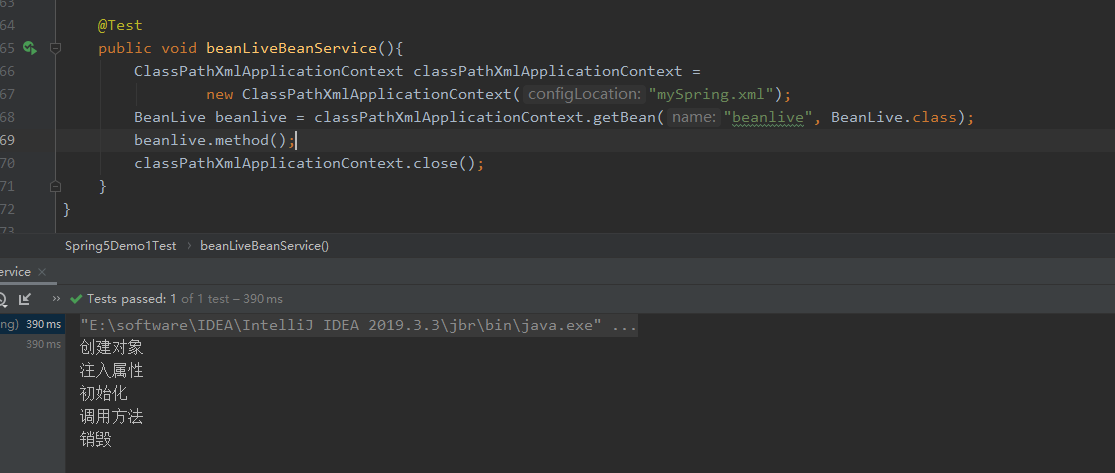
④：bean产生，可以调用

⑤：容器关闭的时候，调用bean的销毁方法（需要手动配置！）





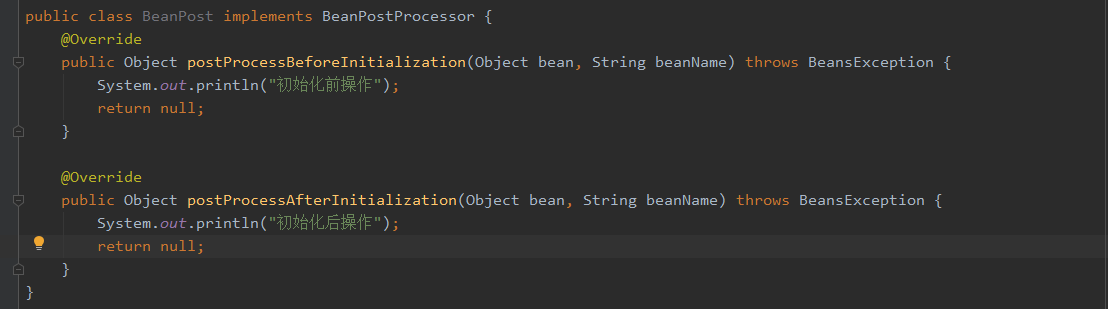
测试发现上面的运行顺序和我们预想的顺序一致！



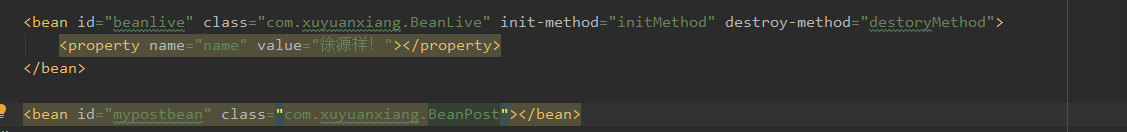
其实还有另外两部操作，统称为bean的后置操作

两步分别在bean初始化前后进行！

创建一个类继承beanpostprocessor



将我们的类进行实例化



此时再运行我们的测试类发现：

在我们的init方法前后都加入了我们的方法！

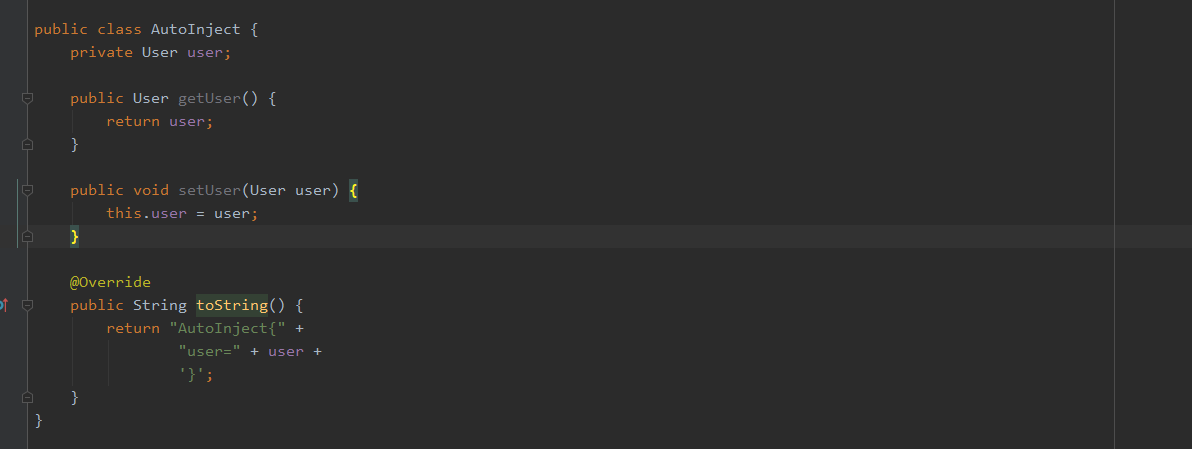
这里多出现的打印是因为我们一旦配置了beanpostprocessor，所有的bean注入都会生效！(即使有一些bean没有配置init方法也会进行初始化前操作初始化后操作！)



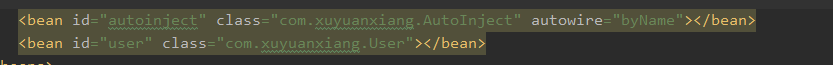
## 十三 bean管理xml方式（自动装配）

我们前面对属性进行赋值的时候都是通过配置标签进行手动注入的！

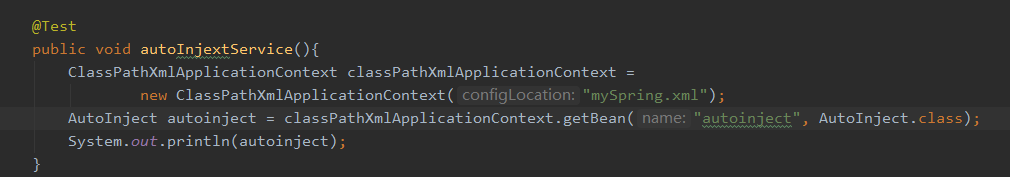
首先创建一个类，内部包含user的属性



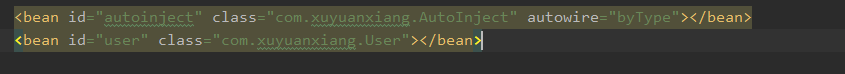
再配置文件中实现配置，这里采用自动配置的方式（byname）要求我们的id和实体类中user变量名相同！



测试封装成功



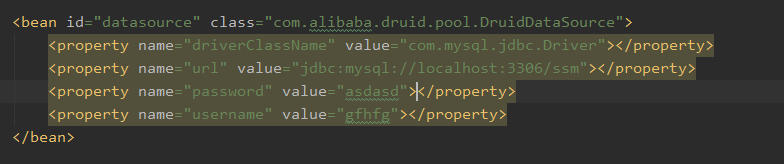
当然我们也可以byType，这样会自动识别类型进行注入，但是需要注意的是同类型只能有一个，否则无法识别！



## 十四 bean管理xml方式（外部属性文件）

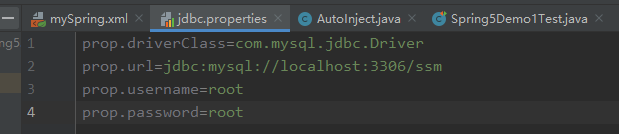
我们以前引入数据库连接池的时候采用的配置：

这里我们的value都写死了，非常不利于修改！

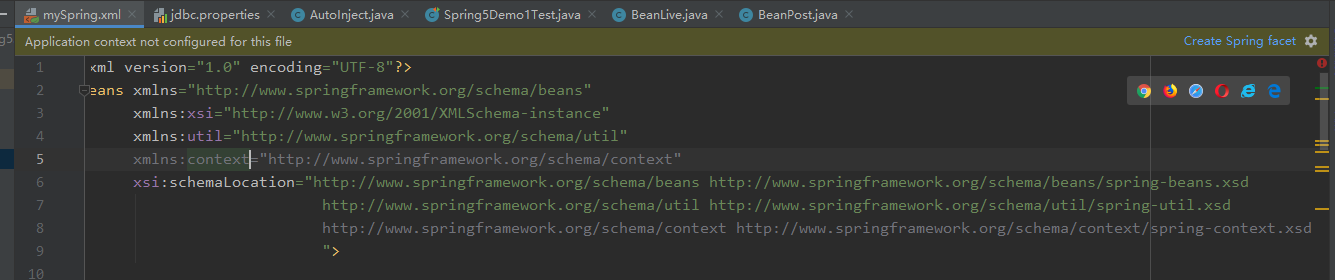


现在我们要将value分离出来！

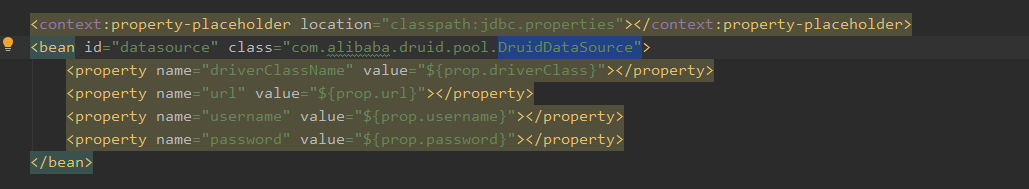
首先创建配置文件：



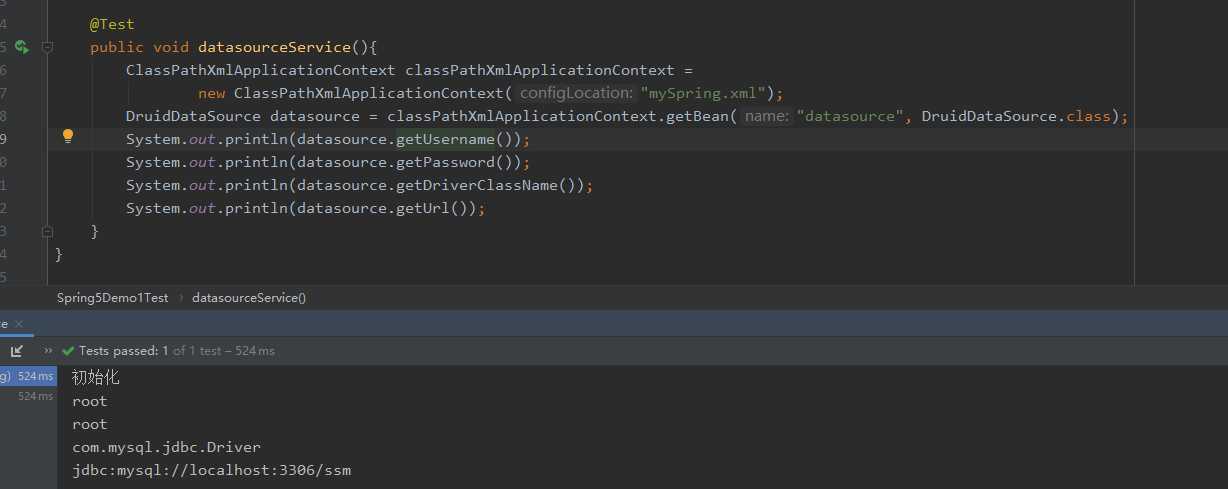
Spring配置文件中添加context的名称空间！



在datasource中修改配置！引入配置文件



测试发现我们的druidDataSource已经注入成功！



## 十五 bean管理注解方式（创建对象）

Spring针对bean管理提供了以下注解：

①：@component

②：@service

③：@controller

④：@repository

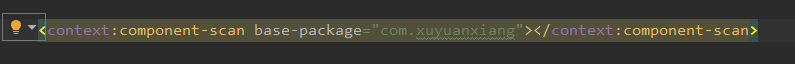
以上四个注解功能都一样，但是我们一般特定的服务使用特定的注解！当然混用也没有问题！

以下进行演示：

首先我们需要引入依赖：

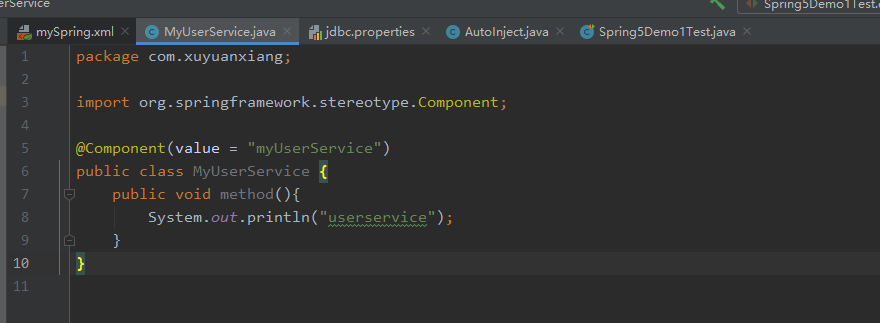


然后我们需要开启组件扫描：



编写我们的类，头顶@compont注解，里面的value属性命名。这里表示就是将我们的类注入到ioc容器中，且id就是value中的值。（如果不写value，默认就是类名首字母小写！）

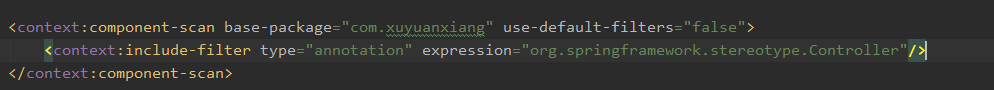
当然我们这里的@compont注解可以换成其他三个！



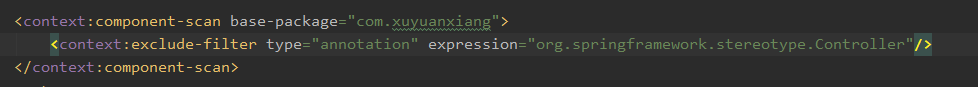
## 十六 bean管理注解方式（组件扫描配置）

这里属性值为false表示不使用默认的扫描方式！（默认扫描方式是上面四种注解都扫描）

然后我们再里面再次配置自己的扫描规则（这里只扫描controller注解，其他注解失效！）



下面这种扫描方式则是使用默认的扫描规则但是排除controller注解！



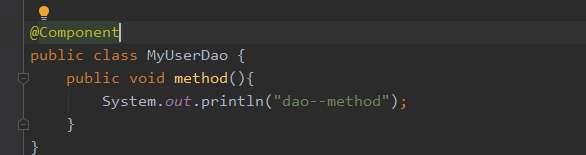
## 十七 bean管理注解方式（属性注入【autowired，qualifier，resource和value】）

@autowired：根据属性的类型进行注入

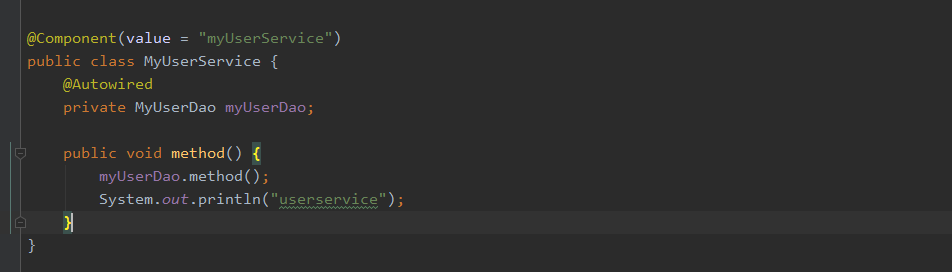
@qualifier：根据属性的名称进行注入

@resource：可以根据类型也可以根据名称

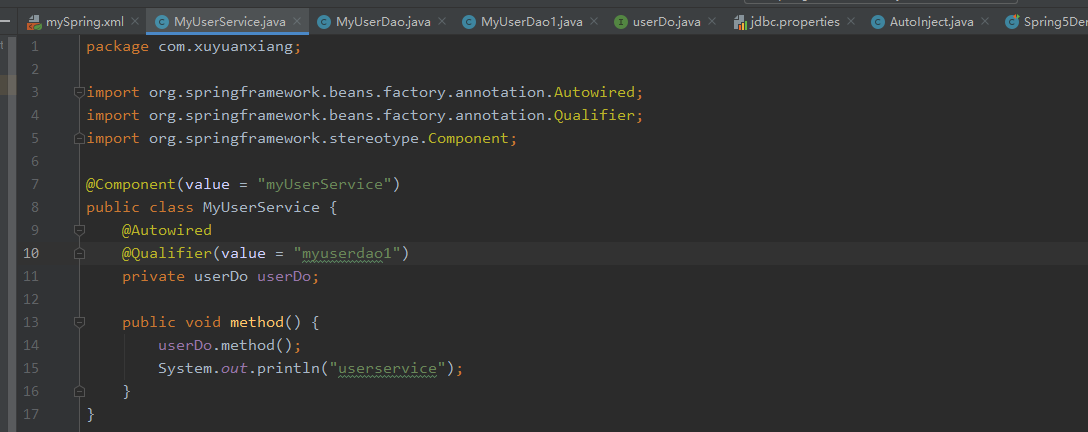
@value：针对普通类型进行注入



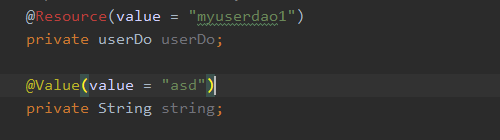
注入成功！



这里传入的如果是接口，就会默认寻找他的子类进行注入！如果有多个子类，就需要加上我们的qualifier注解指定名称了！

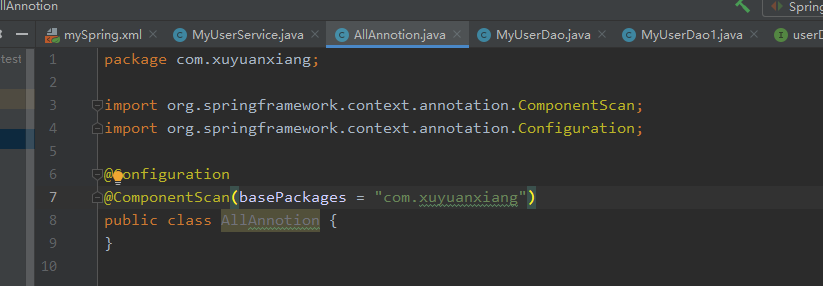


Resource和value使用如下！@resource没有属性的时候是类型注入，有value属性的时候是名称注入！

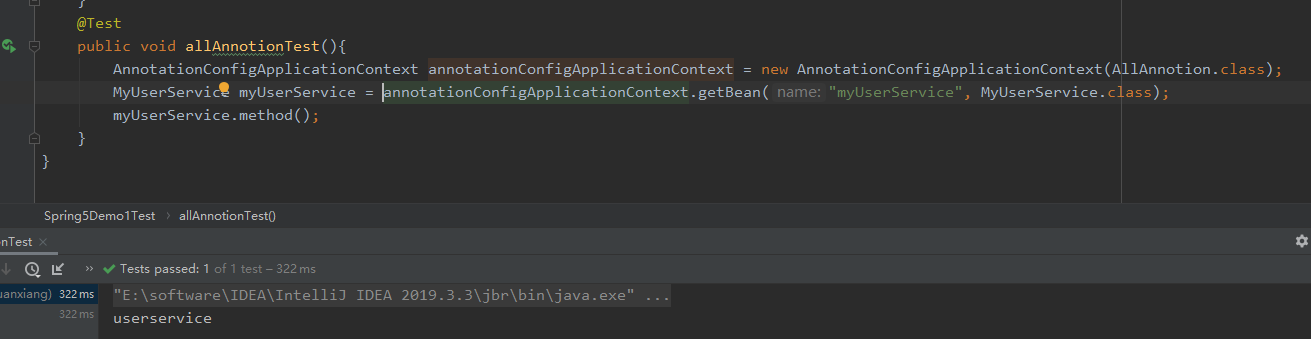


## 十八 bean管理注解方式（全注解开发）

首先创建一个配置类，头顶@Configuration注解表示这是一个配置类！头顶@ComponentScan表示需要扫描的组件包！



测试中需要修改一下解析方法（从xml换成了annotation！）



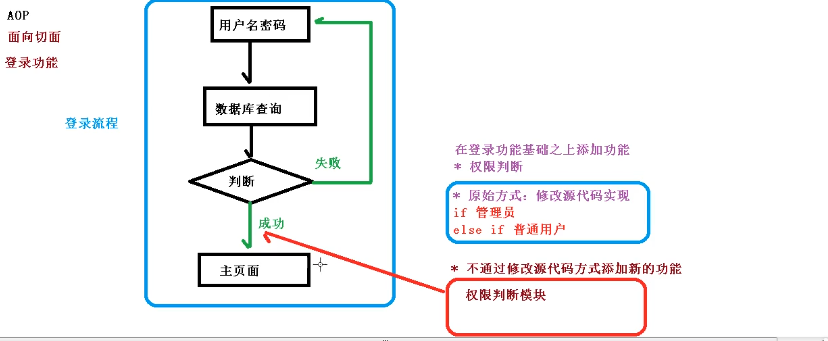
## 十九 aop的概念

Aop也即是面向切面编程

他可以对业务逻辑的各部分进行隔离，降低业务的耦合度，提高程序的可重用性！

通过aop我们可以在不修改源代码的前提下对源代码的功能进行增强！

例如下面的情况！



## 二十 aop的底层原理

Aop底层使用的是动态代理的方式。

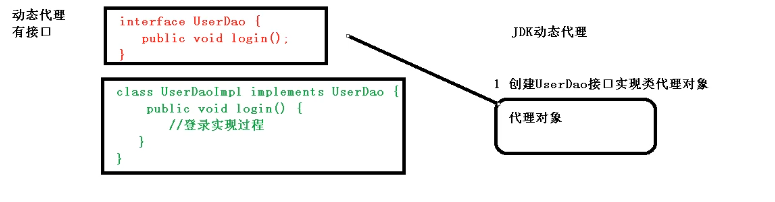
动态代理分为两种：

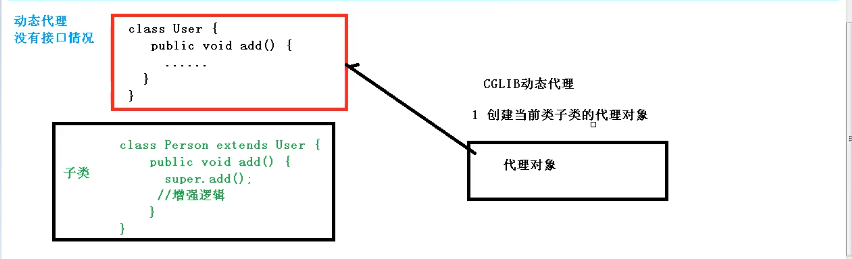
第一种：有接口的情况

使用的是jdk的动态代理（创建接口实现类的代理对象进行增强！）

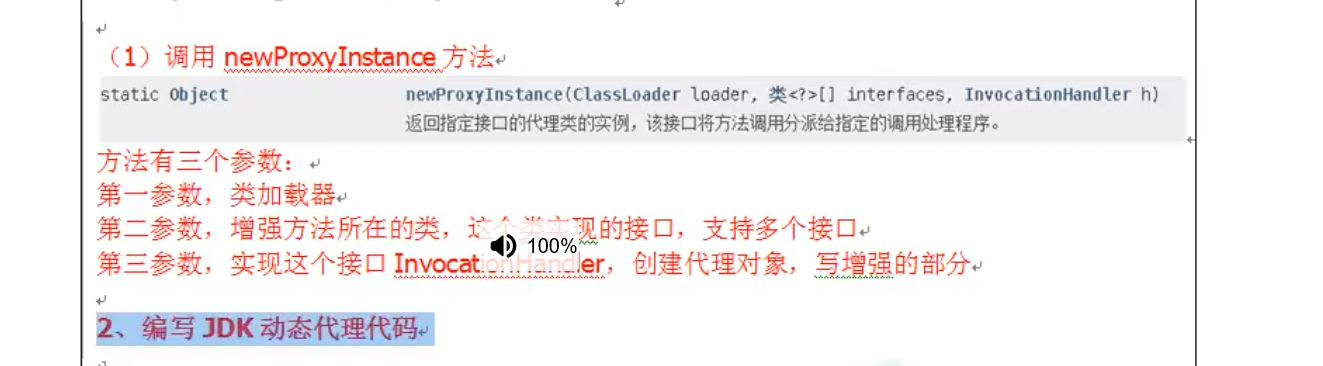
第二种：无接口的情况

使用的是cglib动态代理

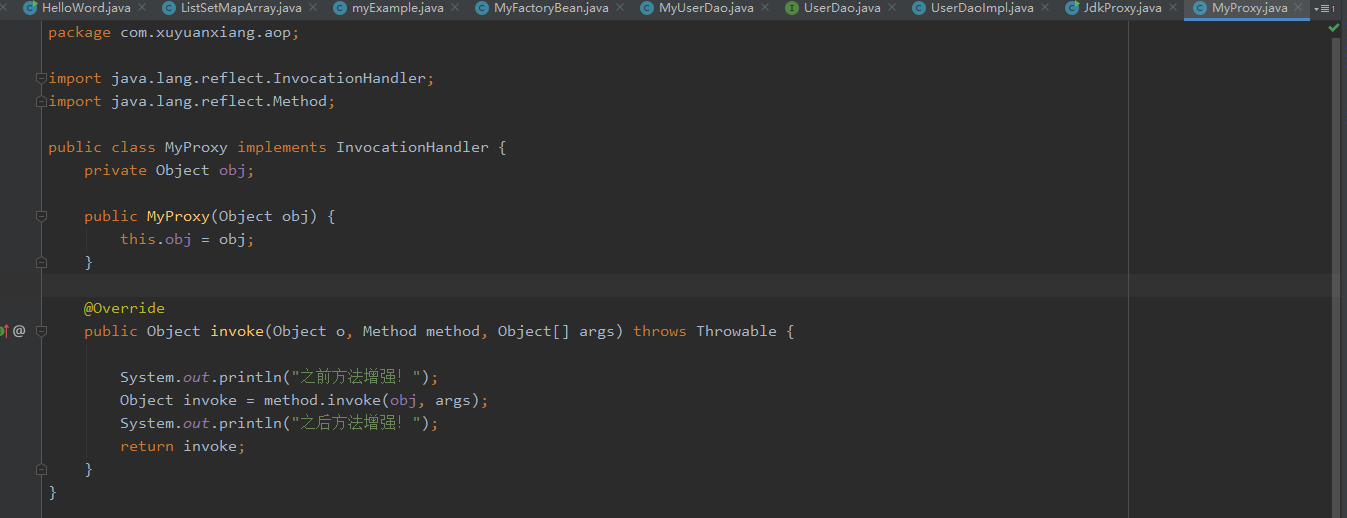




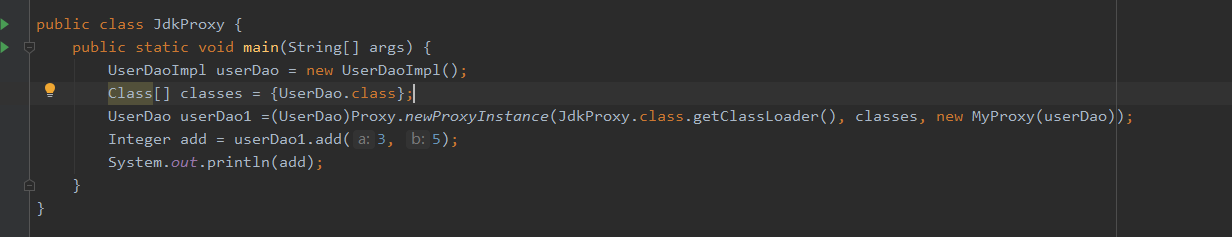
## 二十一 aop的底层原理（jdk动态代理）

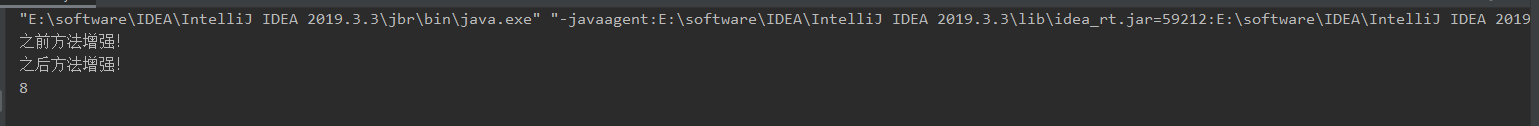


首先创建一个类实现invocationHandler接口，实现其中的invoke方法，同时写一个构造函数用来传入待增强的类！



使用Proxy中的方法创建我们的代理对象，此时采用代理对象进行方法调用，就会直接进入invoke方法中，实现了增强！





## 二十二 aop的专业术语

①：连接点：

类中所有可以增强的方法！

②：切入点：

实现了增强的方法！

③：通知：

实际增强的逻辑部分

前置通知 在我们实际方法前进行执行！

后置通知 在我们实际方法后进行执行

环绕通知 实际方法前后都执行

异常通知 产生异常的时候执行

最终通知 无论如何都会执行

④：切面：

是一个动作，指我们将增强的方法插入切点的过程！

## 二十三 aop操作准备：

Spring框架一般基于aspectj实现aop操作

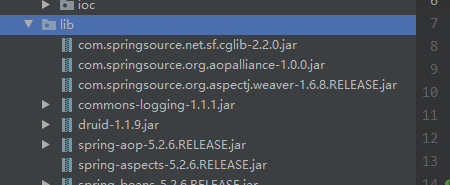
Aspectj本质上就是aop的框架，并不属于spring

我们实现aop有两种方式：

配置文件

注解（这种更方便简洁！）

首先引入依赖：新引入四个



切点表达式：

Execution（【权限修饰符】【返回类型】【类全路径】【方法名称】【参数列表】）

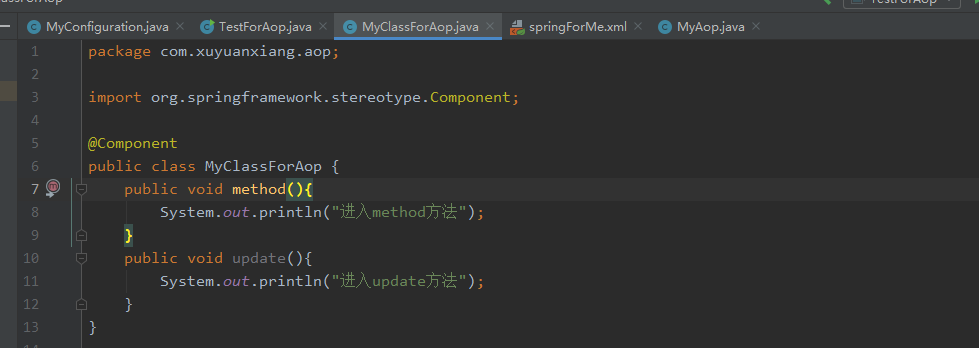
采用\*可以表示全部

例如：表示user类下的所有方法

Execution（\* com.xuyuanxiang.ico.User.\*(..)）

## 二十四 aop的 aspectJ注解

首先创建一个类用来被aop增强！这里支持component注解，放到容器中！



再创建一个aop的类用作增强的逻辑

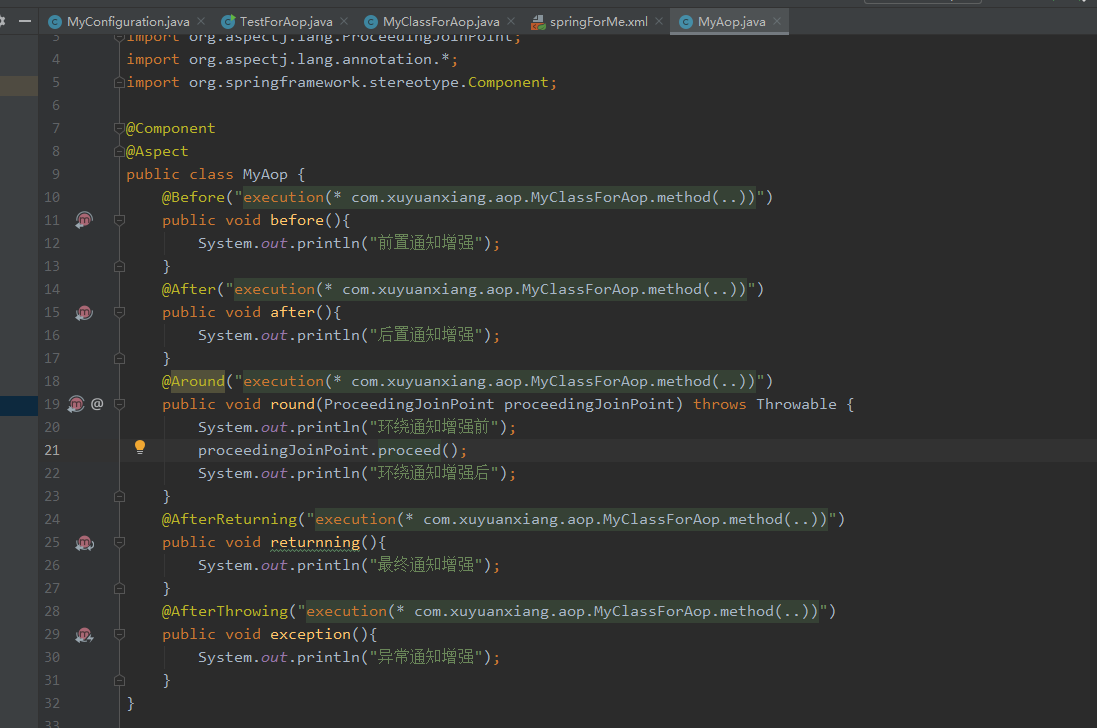
需要注意的是这里的一系列注解：

@After无论如何都会执行，即使异常

@Around环绕通知需要放入一个参数，方法中执行proceed方法代表执行了切点方法！

@Afterreturning后置通知代表方法执行完返回值之后执行。

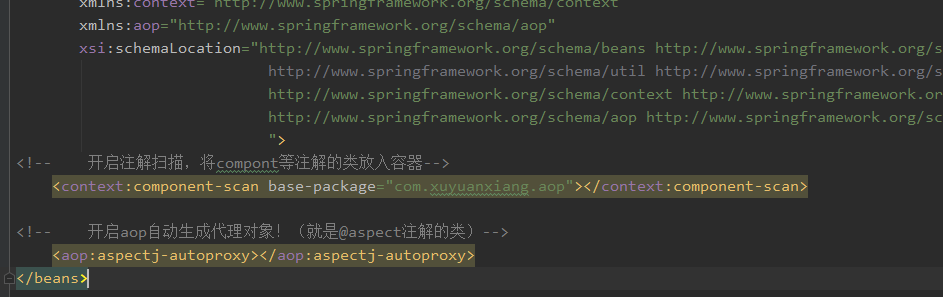
头顶的@Aspect注解表示这是一个增强aop类！



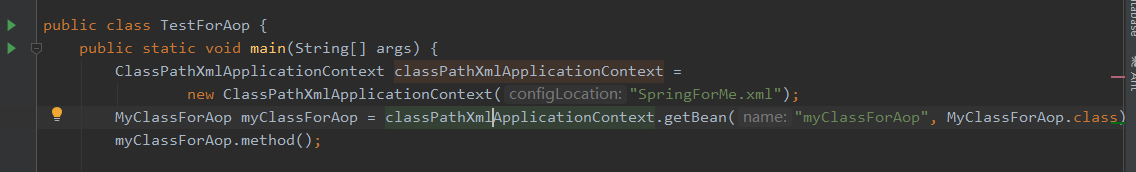
编写配置文件，首先引入命名空间

开启容器组件扫描（主要针对cmoponent，controller，service，reopsity注解！）

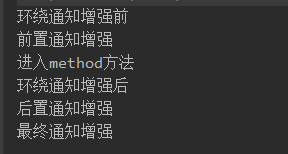
开启自动创建代理对象（主要针对@Aspect注解！）



测试类中进行解析xml文件，调用方法后发现我们的方法已经被增强了！（怀疑将底层的ioc容器中的类替换掉了!）



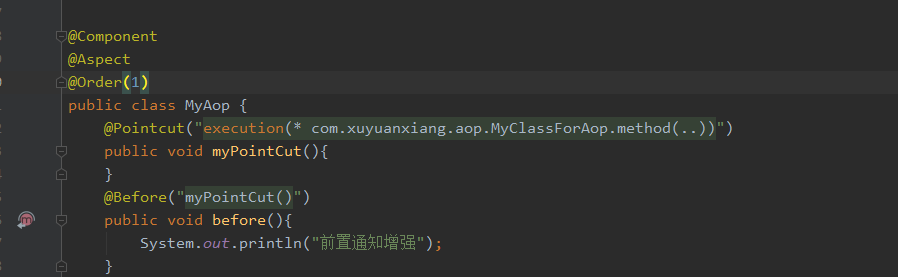
这里查看运行顺序！



当然我们可以对多次出现的切入点进行抽取！

如下，我们自己再写一个方法，头顶@poincut注解，此后其他方法就可以直接调用这个方法（注意要加括号！）获得切点！

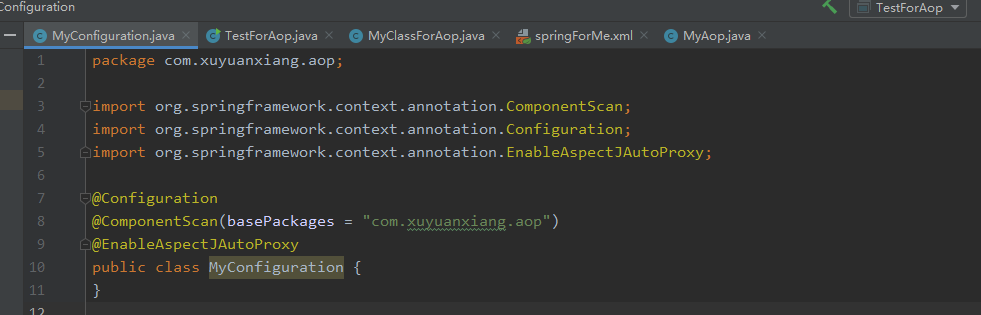
另外我们有多个aop切面的时候，可以采用@Order对aop的增强方法进行排序，序号（从零开始）小的先执行！



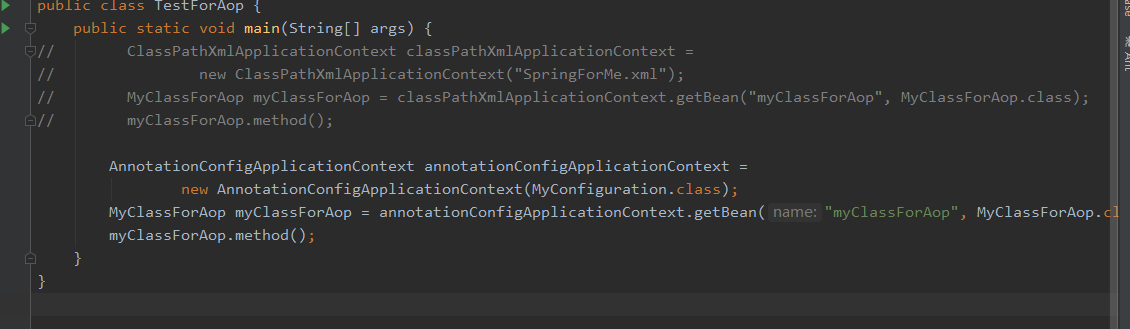
## 二十五 aop的 aspectJ全注解

首先编写一个配置类，指定@Configuration注解表明这是配置类！

@EnableAspectjAutoProxy注解表示支持自动生成代理对象！

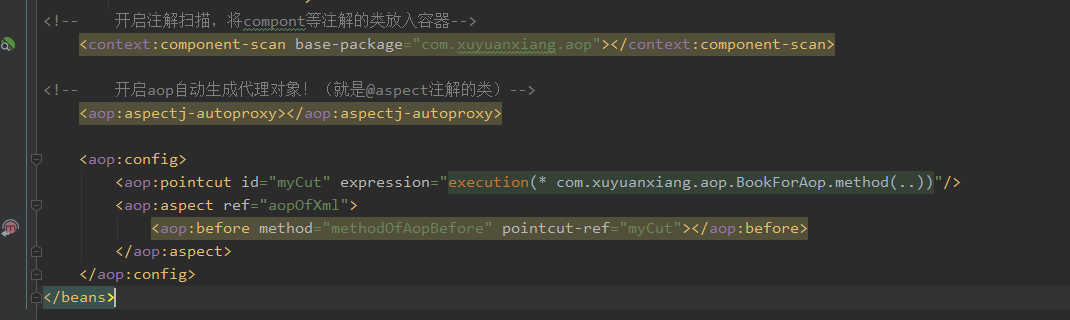


测试结果：正确

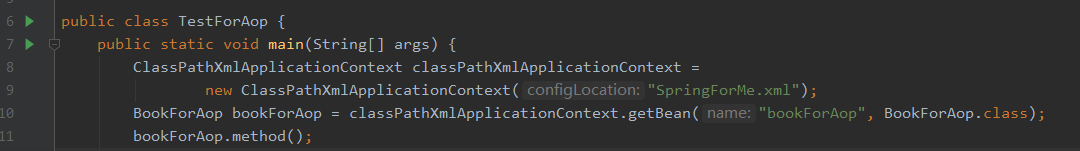


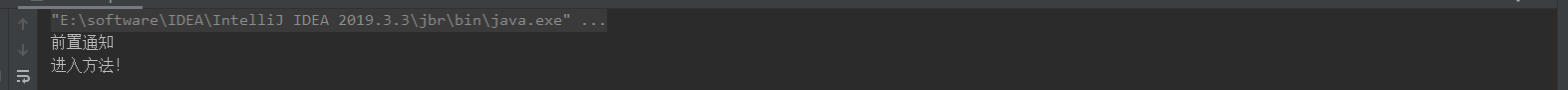
## 二十六 aop的 aspectJ配置文件

Xml文件中做出如下配置，其中aop：aspect标签中的ref属性指定了我们的切面！其中获取的类来自我们的自动扫描的容器组件！



测试发现结果正确！





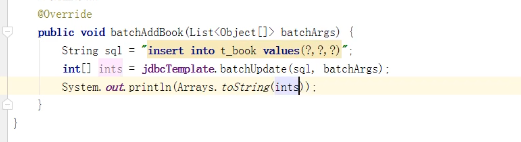
## 二十七 jdbcTemplate中的补充

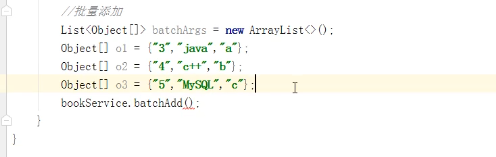
这里补充几个以前没有学过的方法！

批量增加：

利用batchupdate方法，参数传入sql语句和一个list集合（里面防止参数的数组）

方法运行的时候会遍历集合取出每一个数组当作参数放到我们的sql语句中！





批量修改：

原理同上，只不过sql需要更改！连方法名称都是我们的batchupdate

批量删除：

原理同上，只不过sql需要更改！连方法名称都是我们的batchupdate