## 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

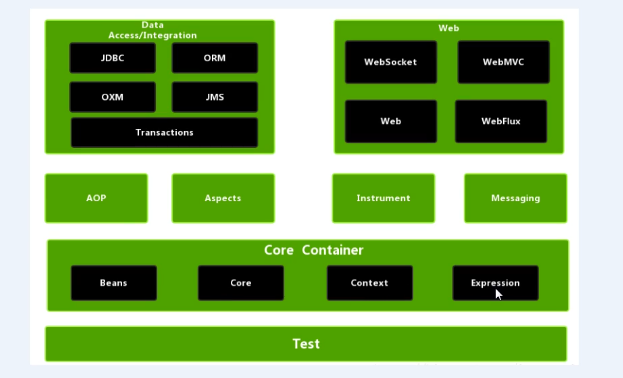
④：集成其他框架非常方便，很优秀

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

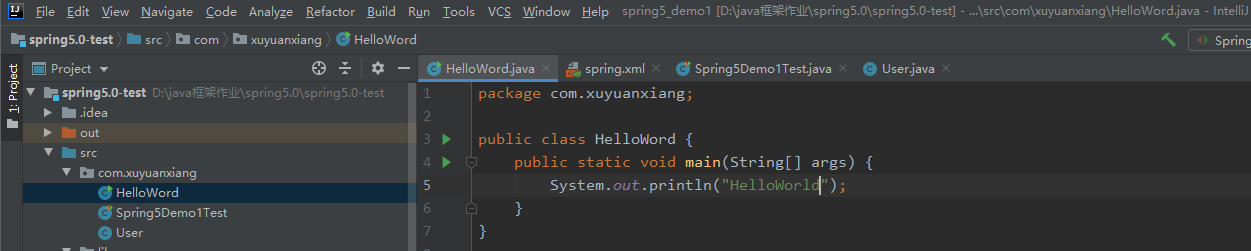
## 二 spring框架入门案例



上面便是spring5.0的所有内容

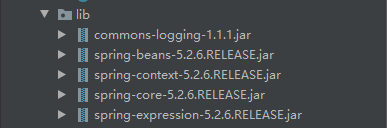
接下来演示spring的基本功能：

首先创建一个项目，暂时创建一个java基本项目就好了：

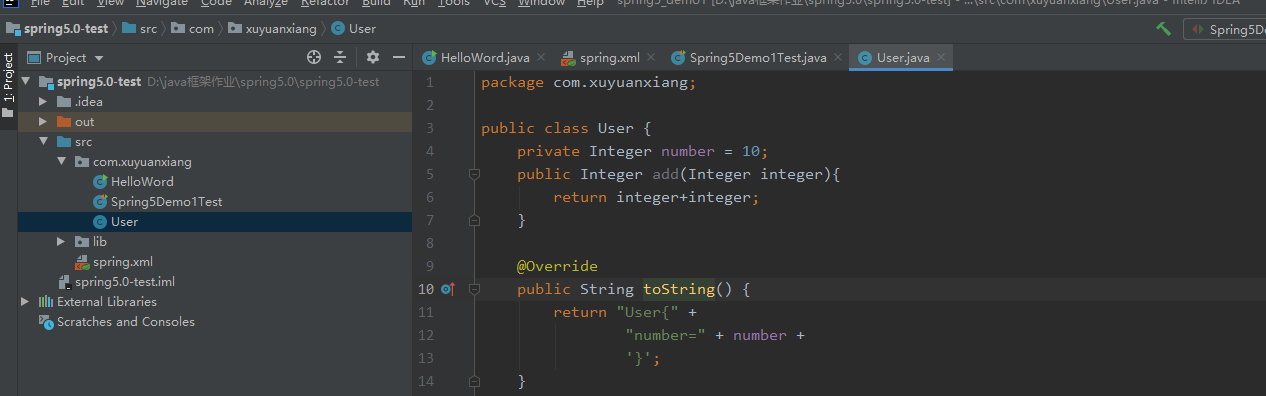


然后创建一个名为lib的包，把需要的jar包导入：

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！

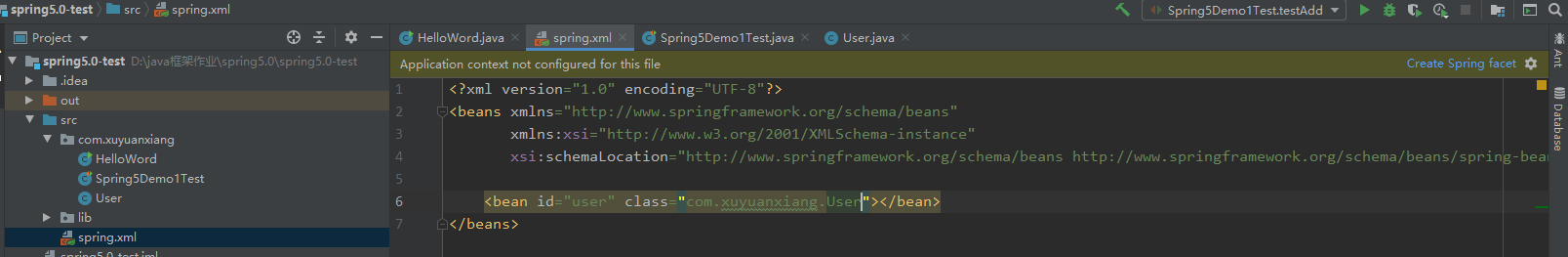


创建一个user类：

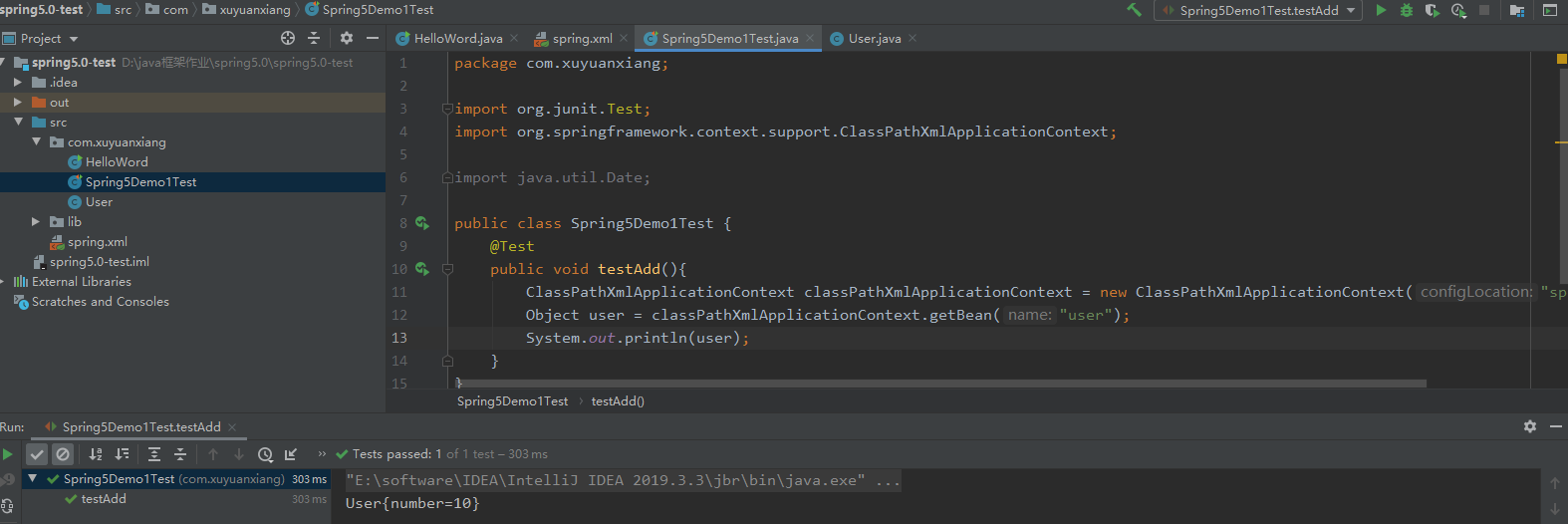


根路径（src）下创建一个spring.xml文件（名字无所谓，但是要一眼知道功能）

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）。



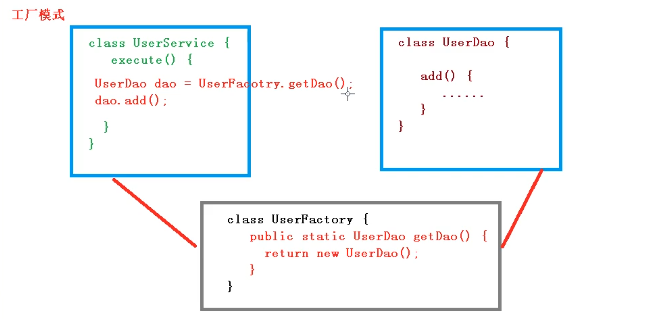
## 三 spring-ioc容器

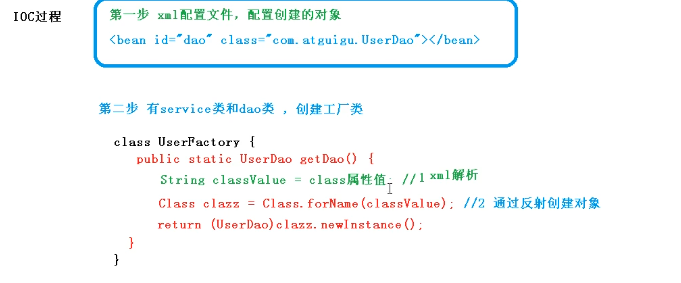
ioc控制反转实际上目的是解决程序之间的耦合度过高的问题：（耦合度低，我们修改代码的时候会非常容易）

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：





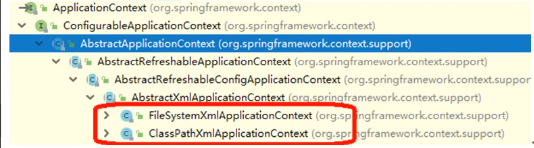
IOC的容器底层就是对象工厂

### 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

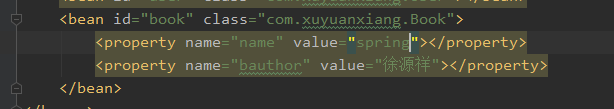
Name属性和id属性类似，但是name中可以写特殊字符，目前已经弃用！

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）

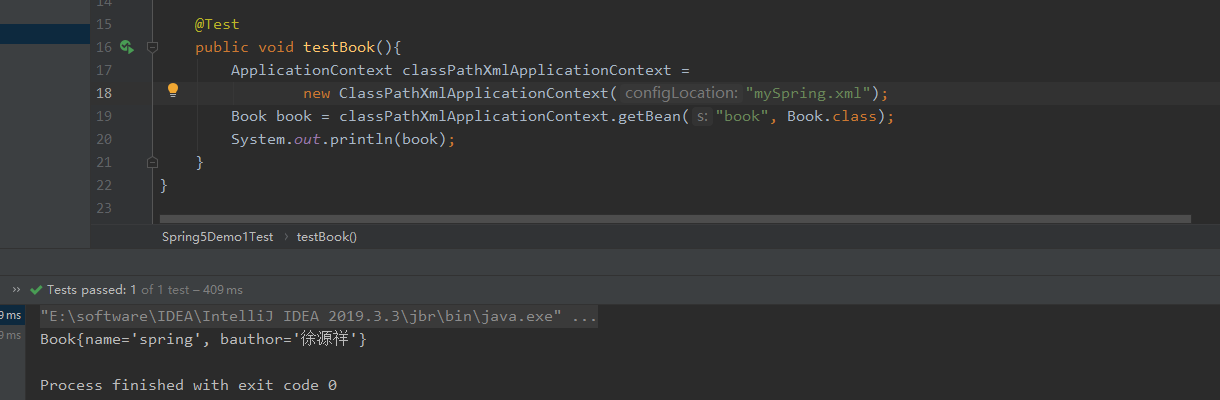
基于xml方式注入属性

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！）

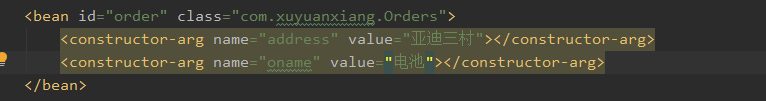


测试注入后的结果！

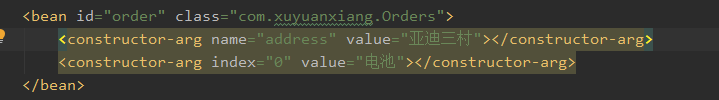


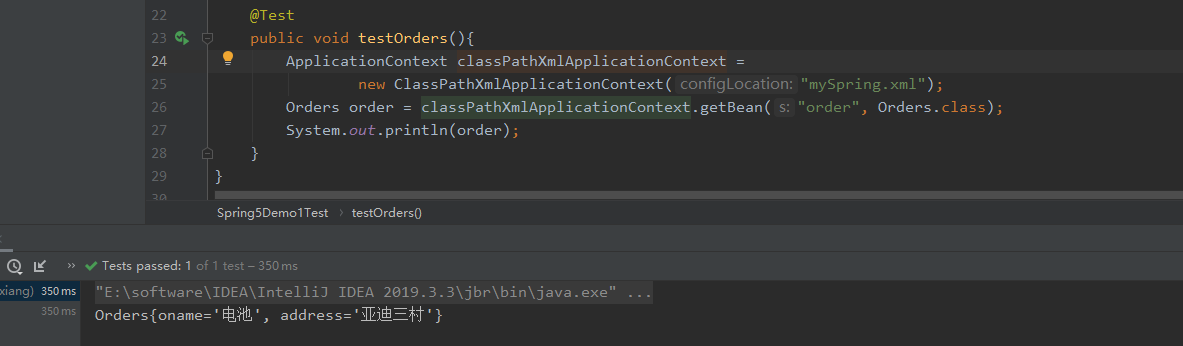
## 五 bean管理xml方式（有参构造方法注入属性）

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）



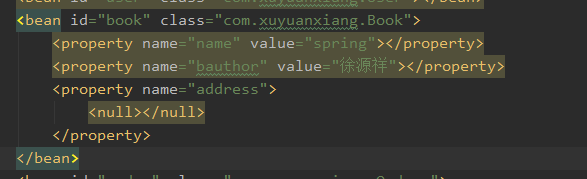
Index指定参数索引





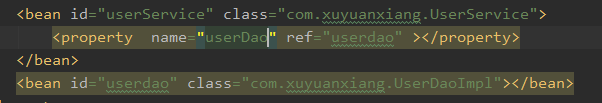
## 六 bean管理xml方式（注入空值和特殊符号）

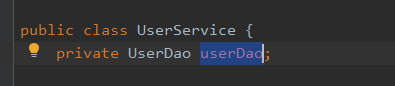
这样我们就可已将指定的参数注入为null，至于注入特殊符号就用到再差吧，太麻烦了



## 七 bean管理xml方式（注入外部bean）

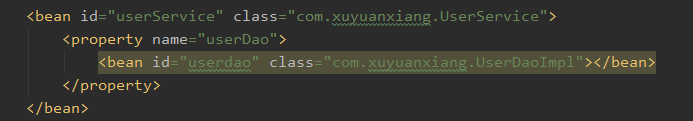
如此<bean>标签实现了外部bean的注入





## 八 bean管理xml方式（注入内部bean以及级联赋值）

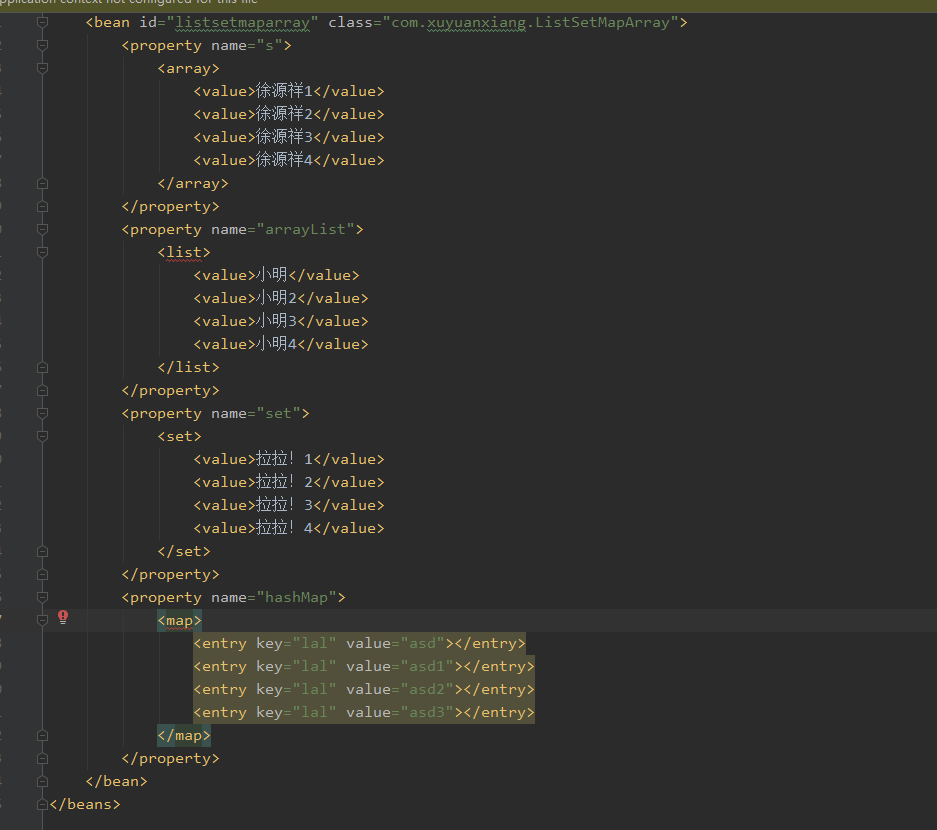
内部bean注入（个人觉的外部注入更清楚）



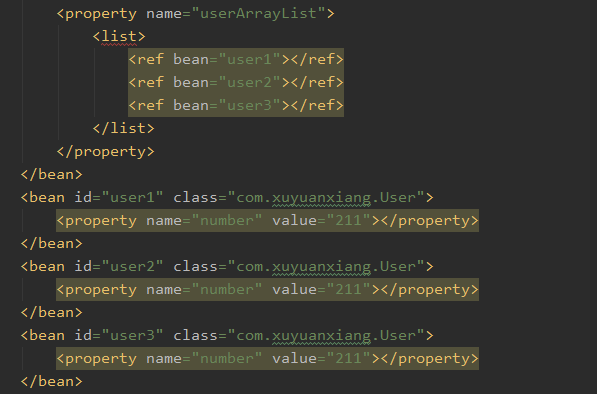
级联赋值：其实就是当我们的userDaoImpl中如果存在属性的话，依然可以继续进行<property>标签进行属性注入，这样就称作级联赋值！

## 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

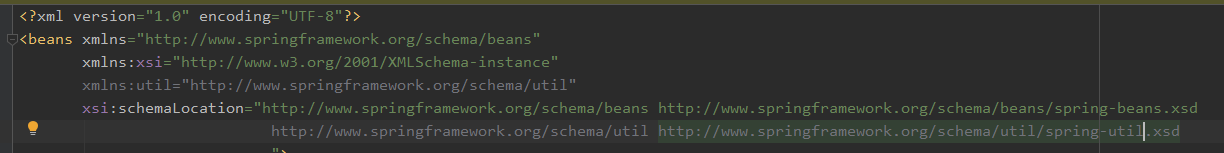


如果我们的集合中放置的不是普通类型而是我们的对象：可以采用如下方式，当然内置bean也可以

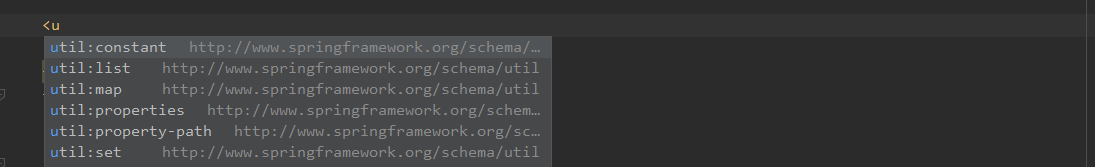


当我们有相同的配置的时候可以将公共的部分做成独立的部分，其他标签调用就好！

首先我们引入util的名称空间



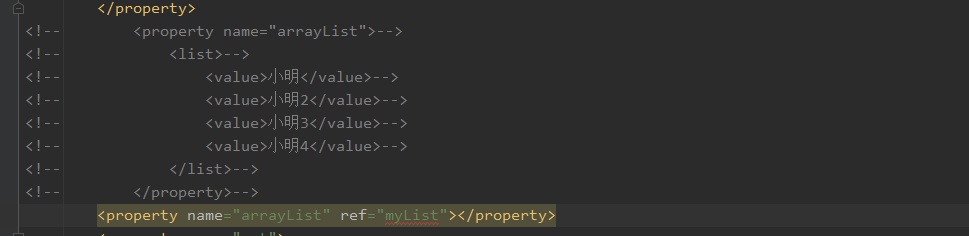
引入以后发现在输入标签的时候多了如下选择：



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】）

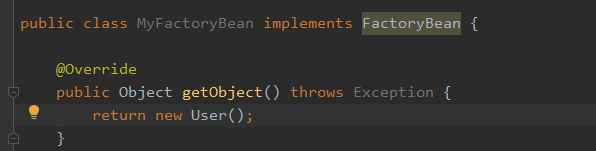
Spring有两种bean

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：



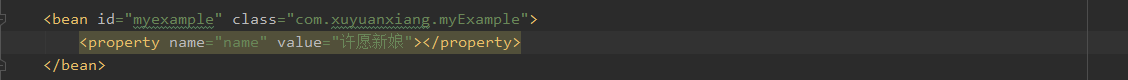
测试中发现原来的类转型已经无法成功了，我们只有转型成为user，并且输出也是user（这就实现了标签中的class属性和返回类型不统一的现象）

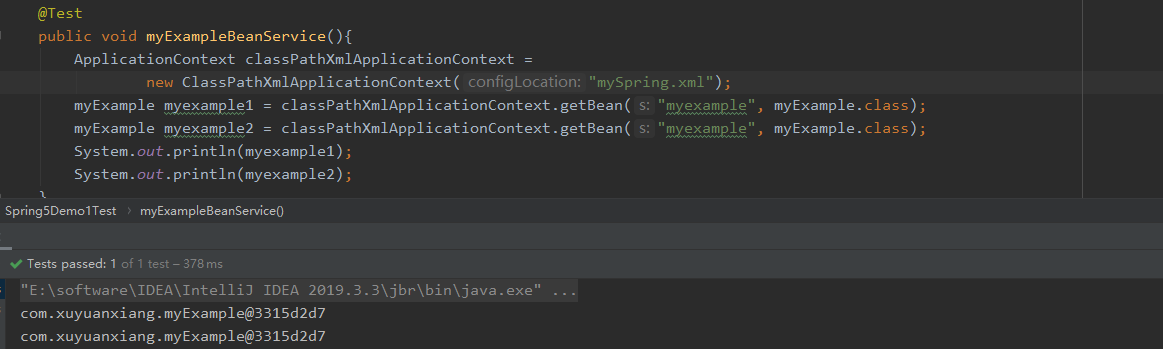


## 十一 bean管理xml方式（bean作用域）

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。

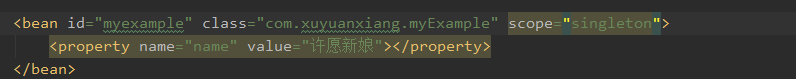
如下我们进行注入一个myexample的实例，结果发现在内存中的地址是相同的



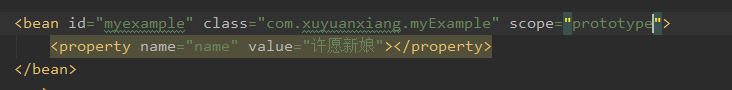


我们可以通过修改bean标签中的scope属性的值进行更改单例或者多例。

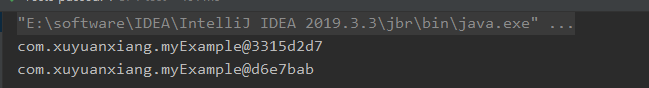
这是单例，不写也是默认这个值



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：

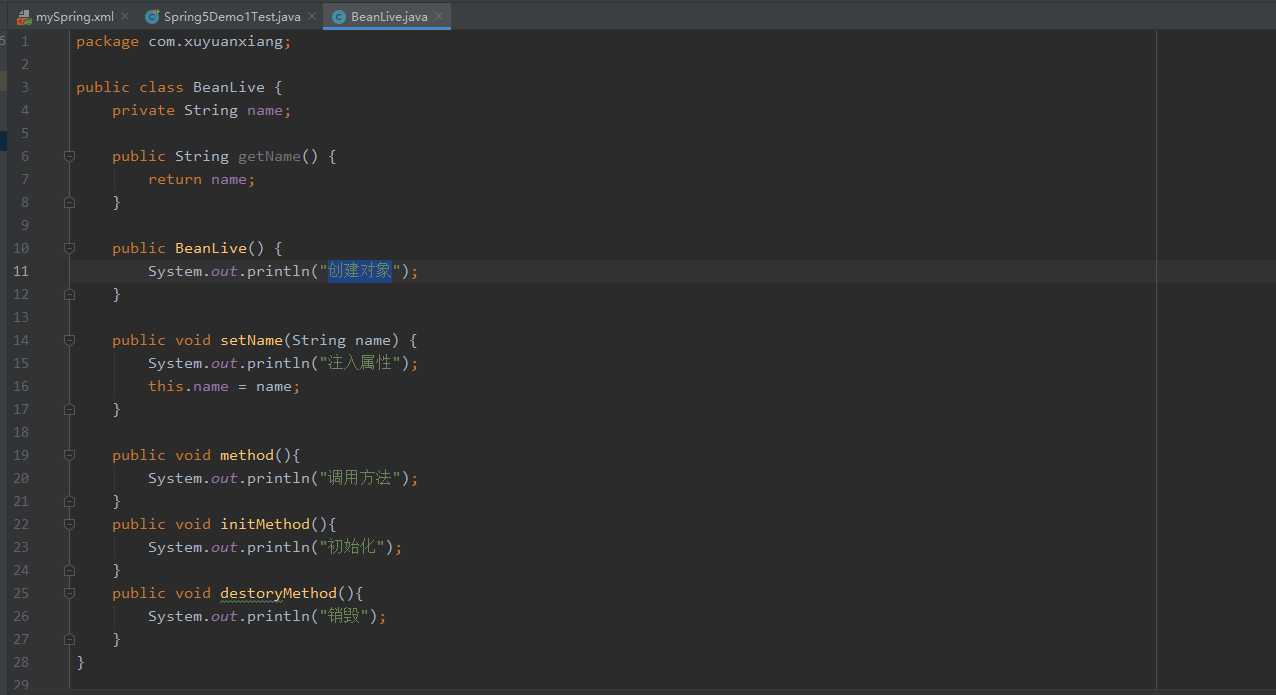
①：通过构造方法创建bean实例

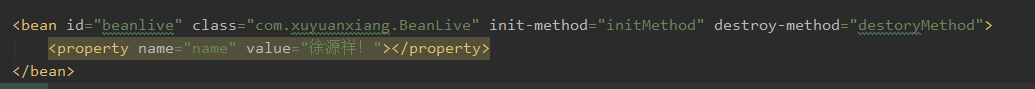
②：bean属性的注入（调用set）

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）

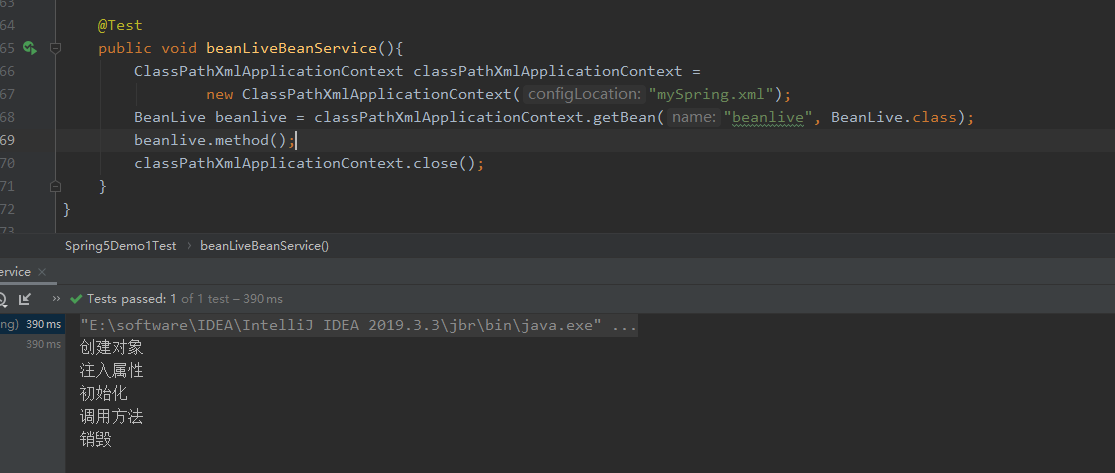
④：bean产生，可以调用

⑤：容器关闭的时候，调用bean的销毁方法（需要手动配置！）





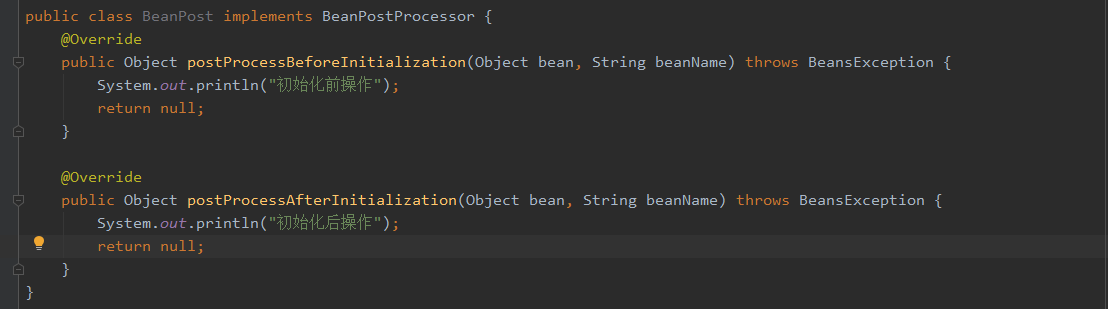
测试发现上面的运行顺序和我们预想的顺序一致！



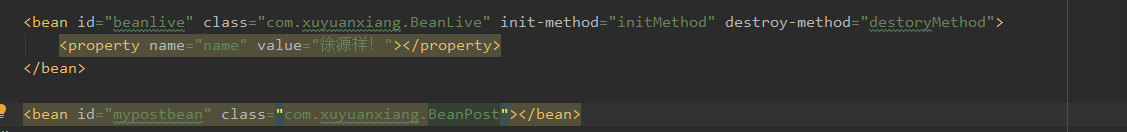
其实还有另外两步操作，统称为bean的后置操作

两步分别在bean初始化方法前后进行！

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！

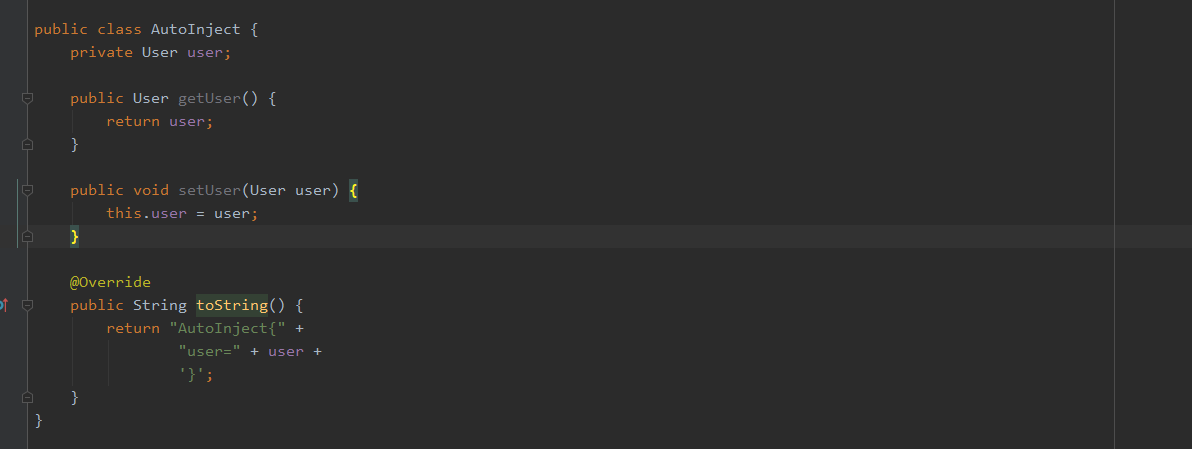
这里多出现的打印是因为我们一旦配置了beanpostprocessor，所有的bean注入都会生效！(即使有一些bean没有配置init方法也会进行初始化前操作初始化后操作！)



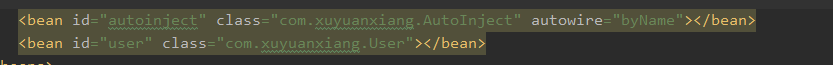
## 十三 bean管理xml方式（自动装配）

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！

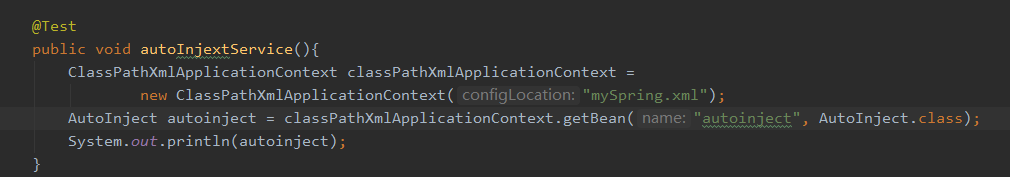
首先创建一个类，内部包含user的属性



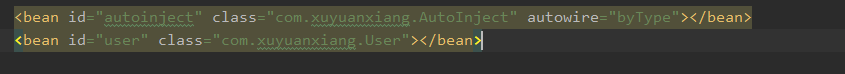
再配置文件中实现配置，这里采用自动配置的方式（byname）要求我们的id和实体类中user变量名相同！



测试封装成功



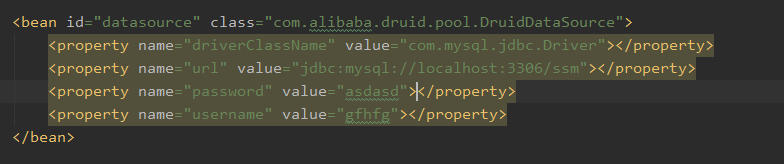
当然我们也可以byType，这样会自动识别类型进行注入，但是需要注意的是同类型只能有一个，否则无法识别！



## 十四 bean管理xml方式（外部属性文件）

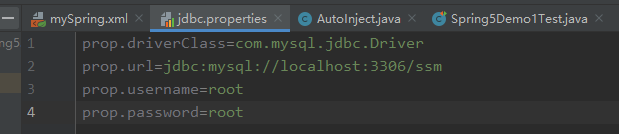
我们以前引入数据库连接池的时候采用的配置：

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！

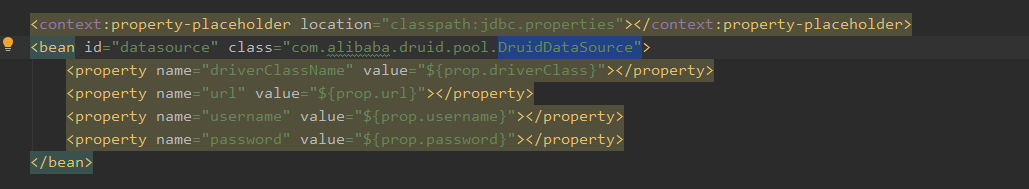
首先创建配置文件：



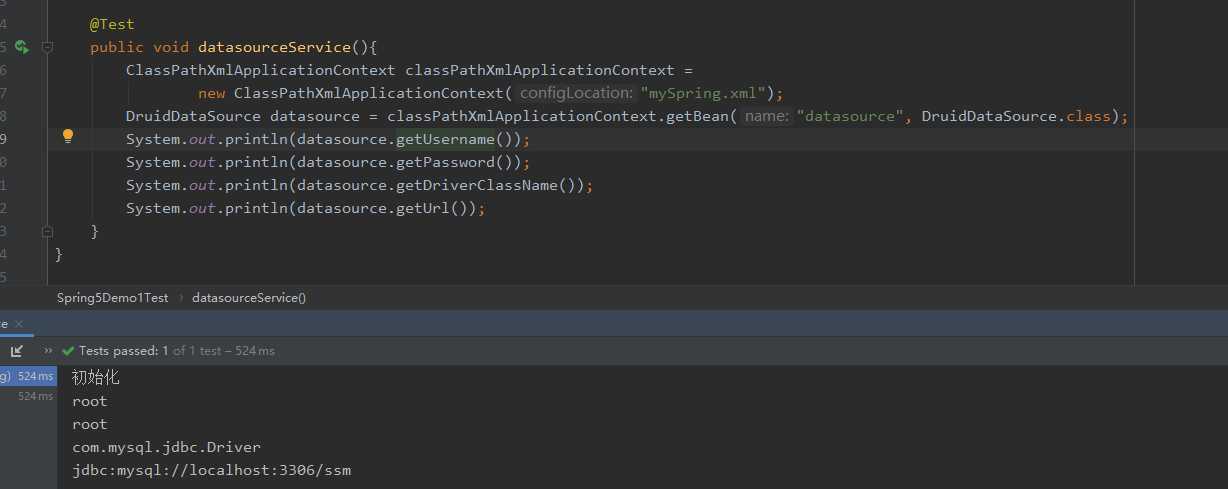
Spring配置文件中添加context的名称空间！



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

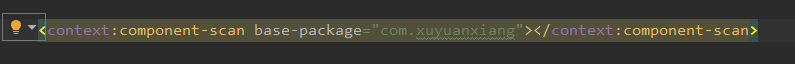
以上四个注解功能都一样，但是我们一般特定的服务使用特定的注解！当然混用也没有问题！

以下进行演示：

首先我们需要引入依赖：

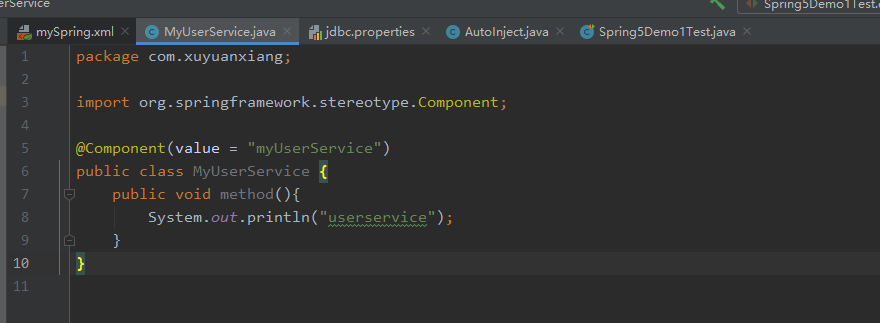


然后我们需要开启组件扫描：



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！）

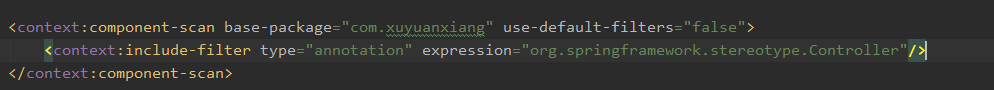
当然我们这里的@compont注解可以换成其他三个！



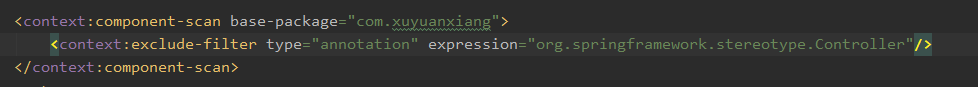
## 十六 bean管理注解方式（组件扫描配置）

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！



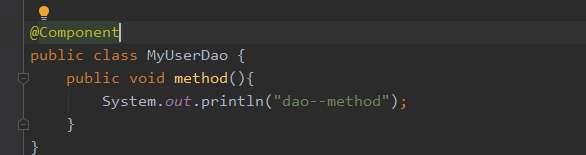
## 十七 bean管理注解方式（属性注入【autowired，qualifier，resource和value】）

@autowired：根据属性的类型进行注入

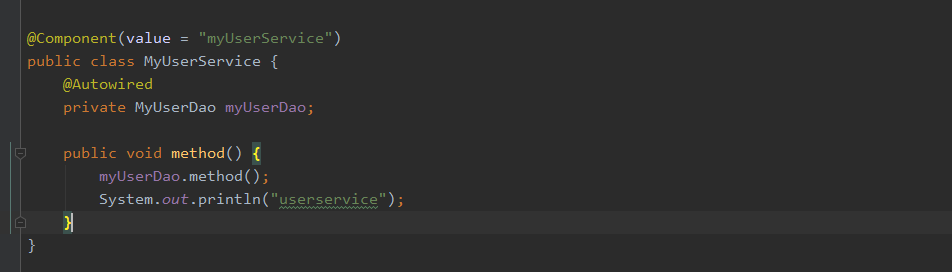
@qualifier：根据属性的名称进行注入

@resource：可以根据类型也可以根据名称

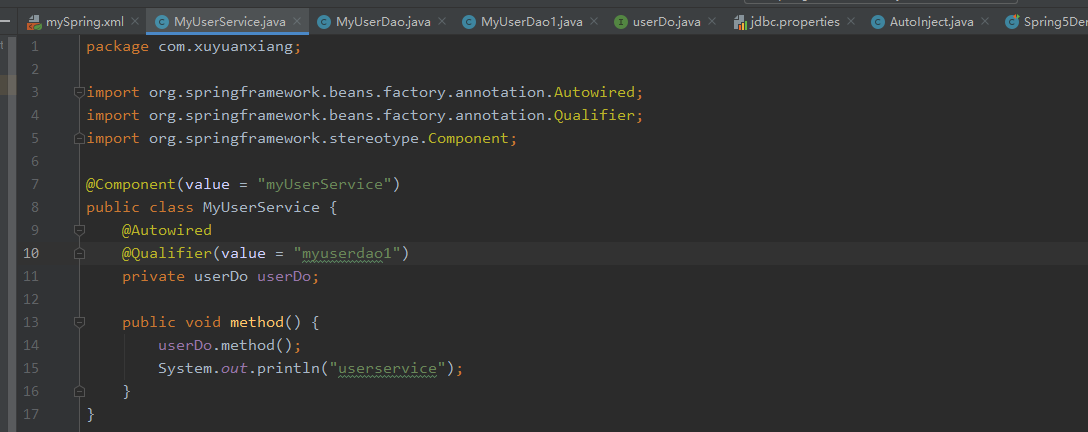
@value：针对普通类型进行注入



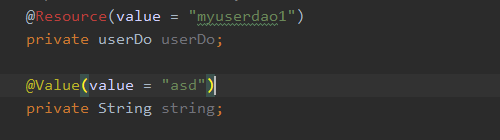
注入成功！



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！

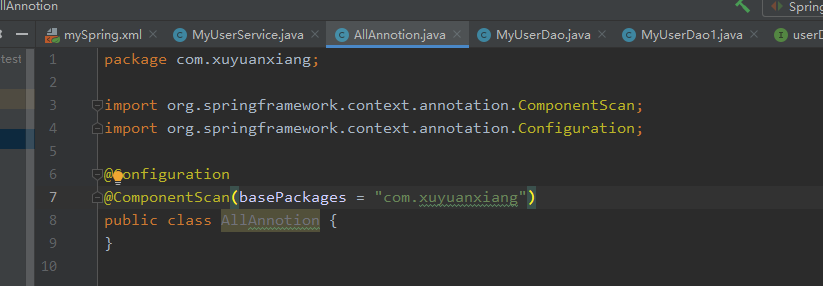


Resource和value使用如下！@resource没有属性的时候是类型注入，有value属性的时候是名称注入！

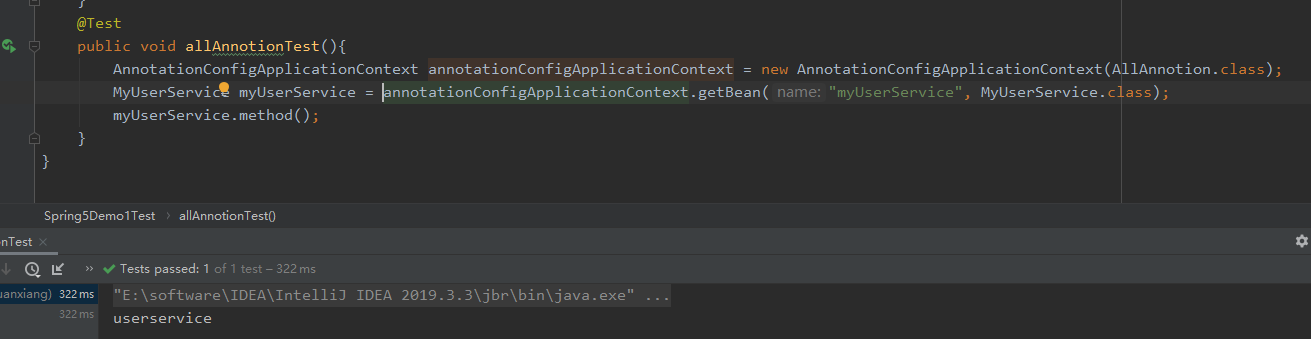


## 十八 bean管理注解方式（全注解开发）

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！）



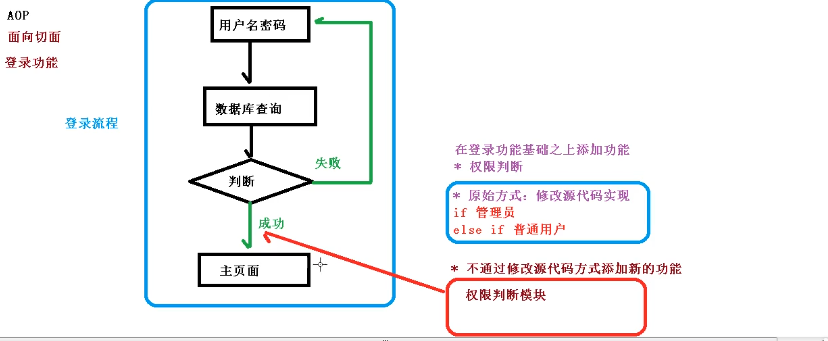
## 十九 aop的概念

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。

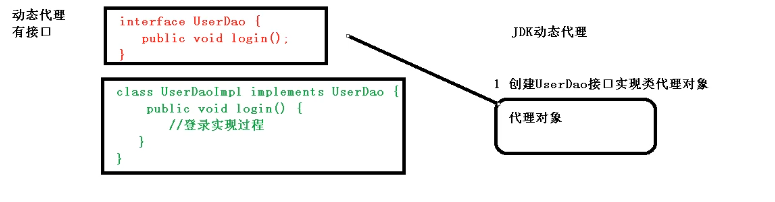
动态代理分为两种：

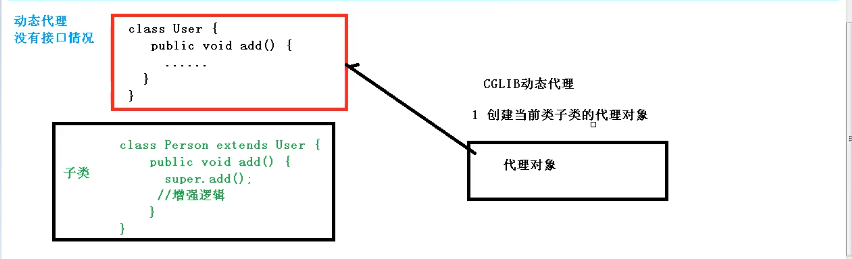
第一种：有接口的情况

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！）

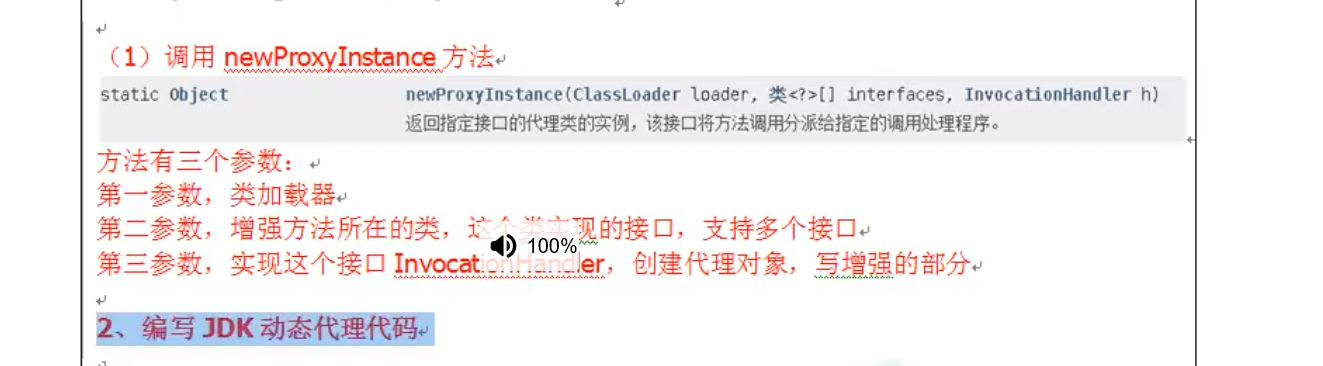
第二种：无接口的情况

使用的是cglib动态代理

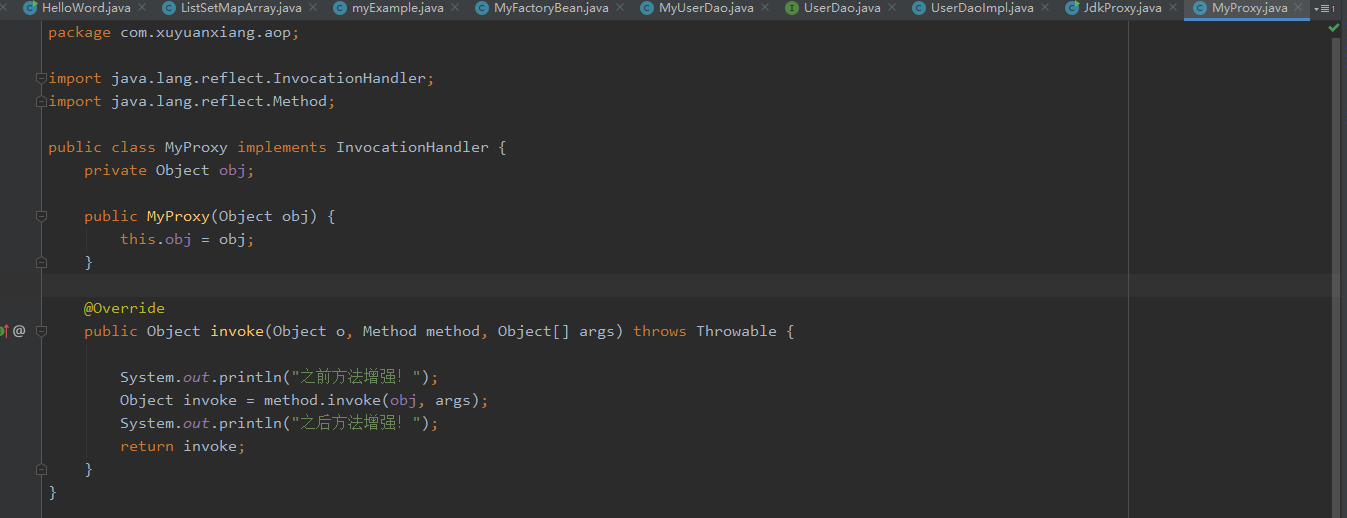




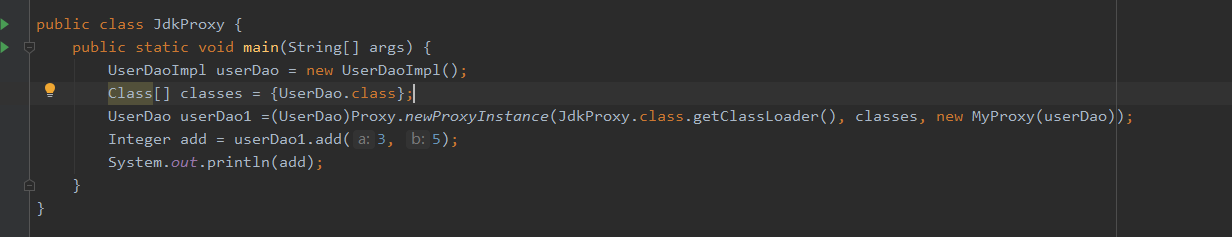
## 二十一 aop的底层原理（jdk动态代理）



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

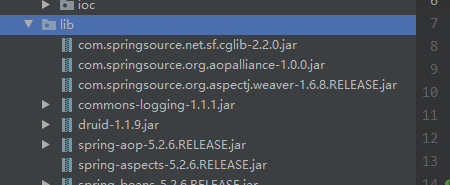
Aspectj本质上就是aop的框架，并不属于spring

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】）

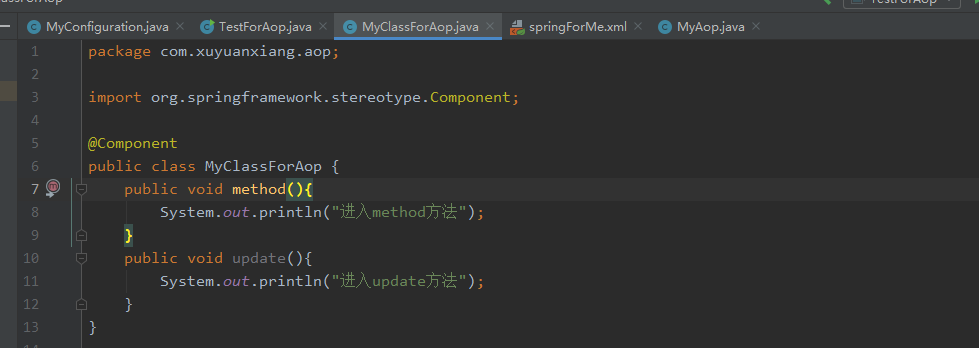
采用\*可以表示全部

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

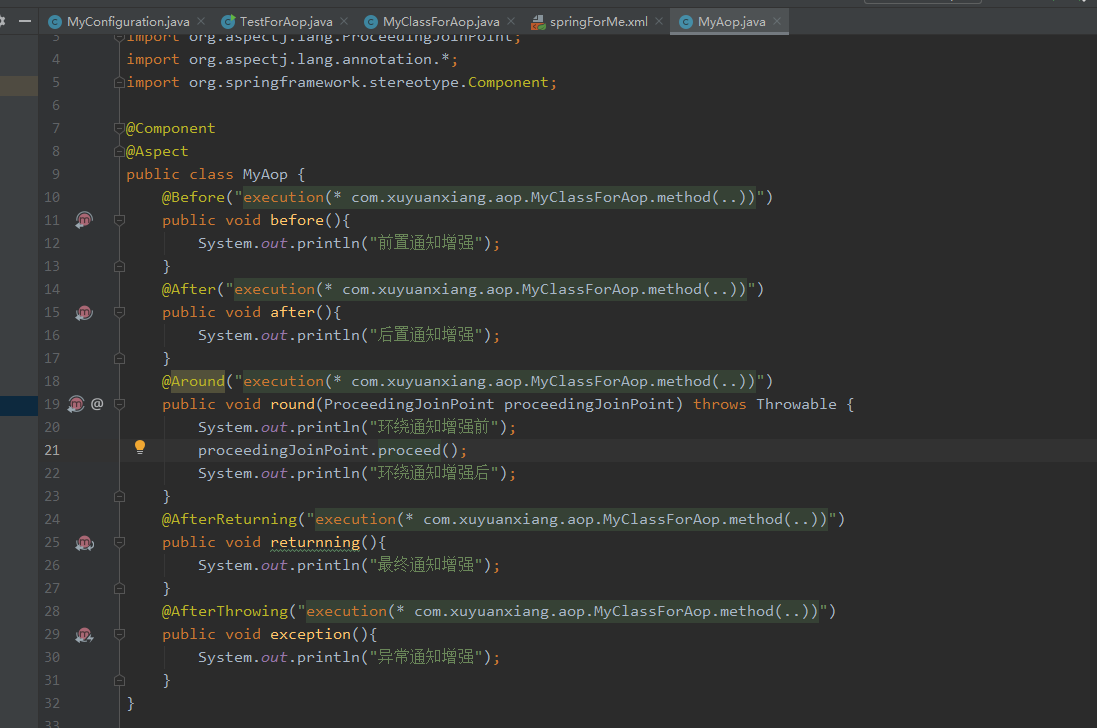
需要注意的是这里的一系列注解：

@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！



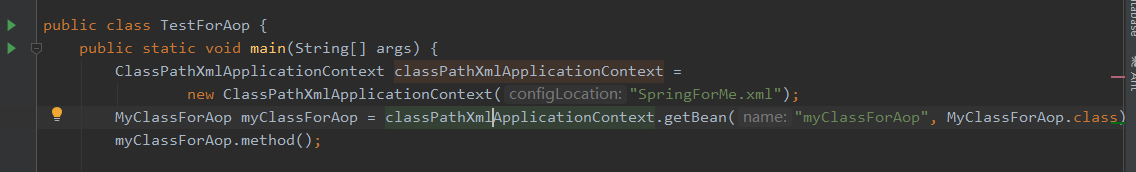
编写配置文件，首先引入命名空间

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

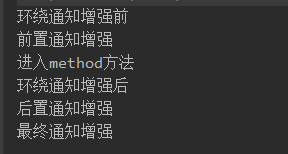
开启自动创建代理对象（主要针对@Aspect注解！）



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!）



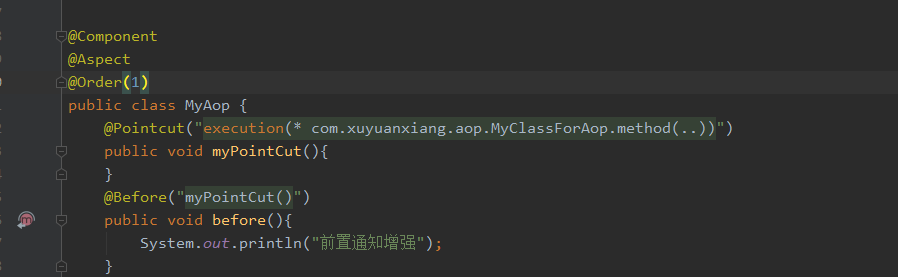
这里查看运行顺序！



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@pointcut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！

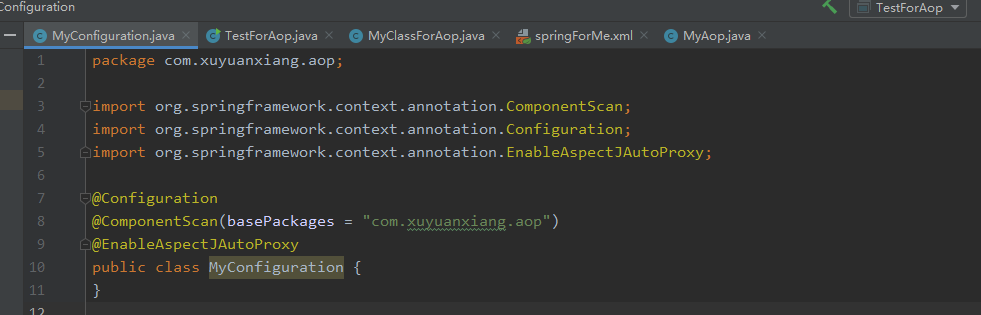
另外我们有多个aop切面的时候，可以采用@Order对aop的增强方法进行排序，序号（从零开始）小的先执行！



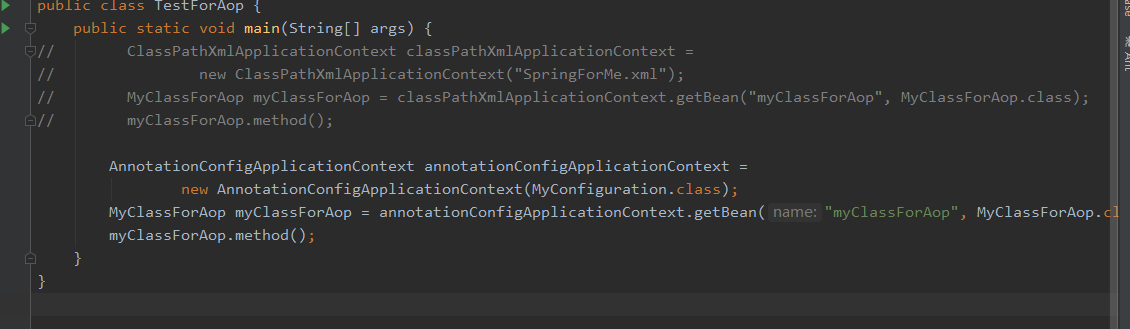
## 二十五 aop的 aspectJ全注解

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！

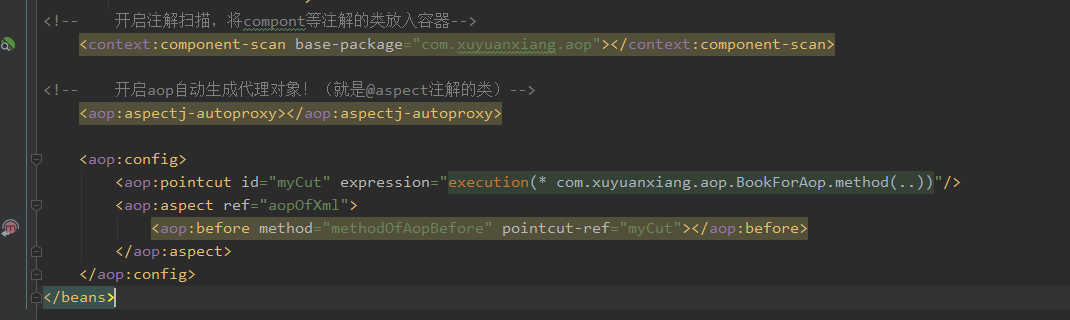


测试结果：正确



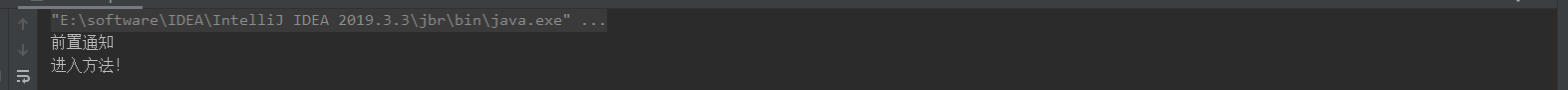
## 二十六 aop的 aspectJ配置文件

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！





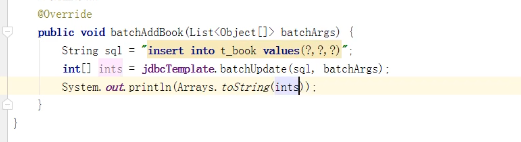
## 二十七 jdbcTemplate中的补充

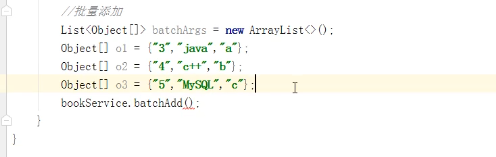
这里补充几个以前没有学过的方法！

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

## 二十八 事务的概念

事务是操作数据库的最小单元，指的是逻辑上的一组操作，要么大家一起成功，要么一起失败！

事务的四个特征：（acid）

①：原子性 最小单元

②：一致性 总量不变

③：隔离性 互不干扰

④：持久性 一旦生效，永久变化

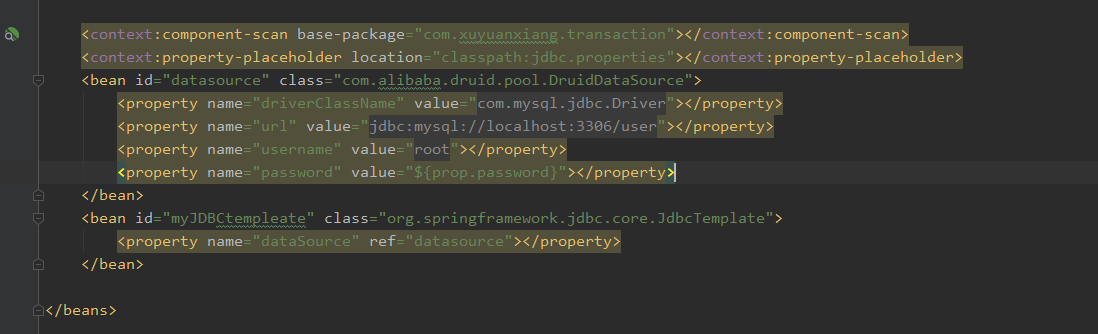
## 二十九 事务操作的环境搭建



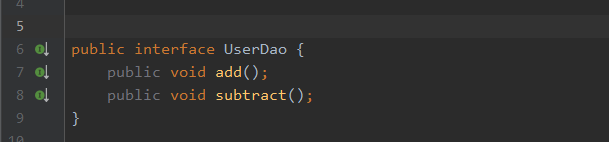


首先导入我们需要的jar包，这里需要采用jdbctemplate，所以导入了mysql以及spring-jdbc，又需要事务管理，导入spring-tx，而整合其他持久层框架需要spring-orm（这里不用 也可）

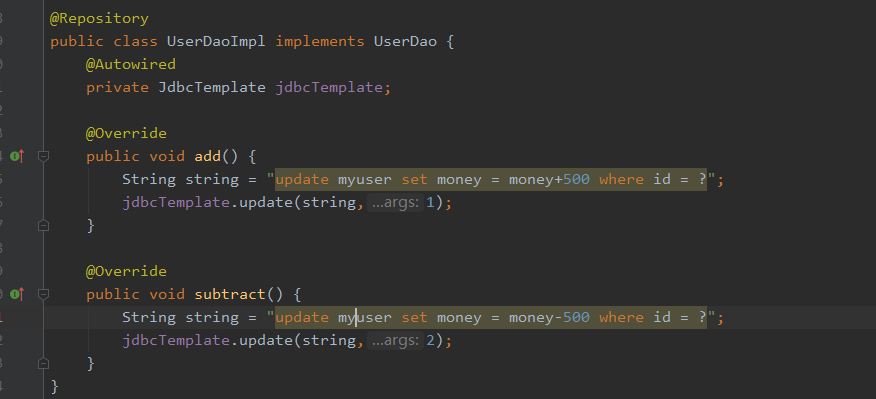
首先配置xml配置文件，做好jdbctemplate以及datasource的注入，开启componet那一堆的组件注入！



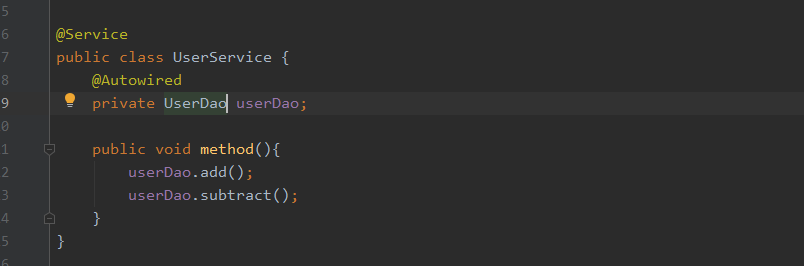
编写userdao接口



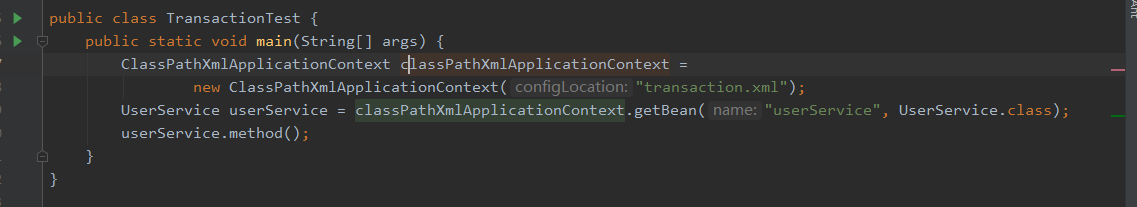
编写实现类



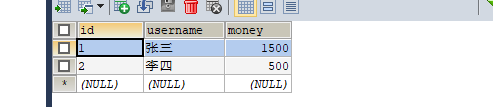
编写service接口同时实现账户的加减



再测试类中进行测试！



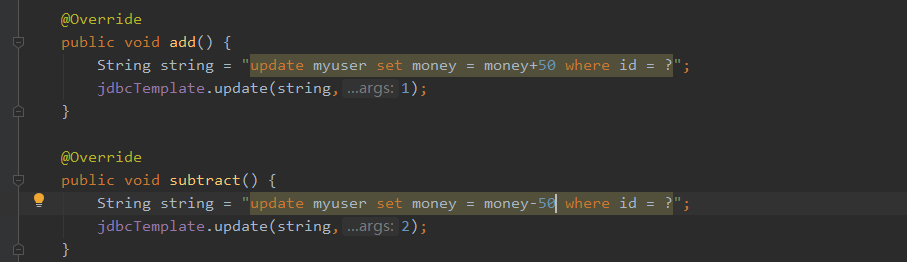
数据库中发现的确做出了交易



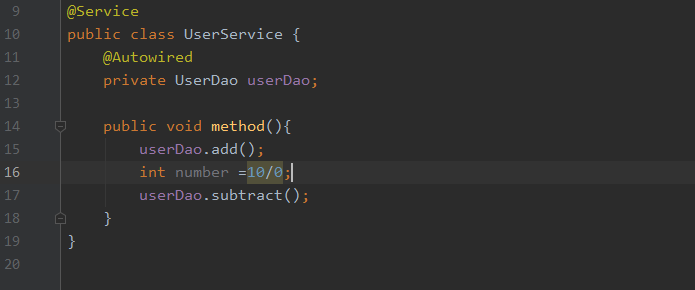
## 三十 事务操作的引入

上一节的代码正常运行的时候是没有问题的，可是当我们的方法运行到一半出现异常的时候，就会发生一些不应该出现的现象！

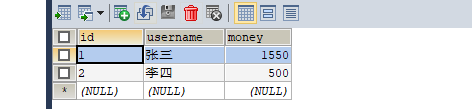
修改代码，减少交易额！



交易中间添加一个异常！



结果显示，一方加了钱，一方没有少钱！



以上问题就需要采用事务来解决！（我们事务一般添加在service层，他更符合原子性吧！）

我们有两种类型的事务可以解决如上问题：

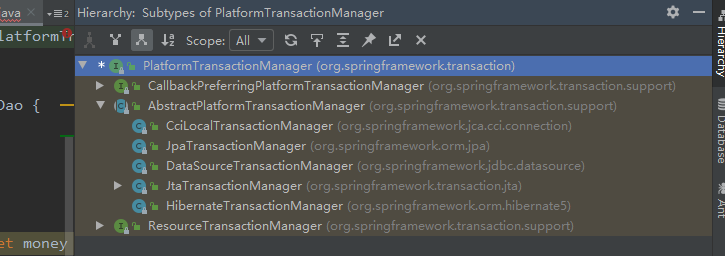
一种是编程式事务，一种是声明式事务。

编程式就是在我们的代码中添加try catch结构，如果发生异常，就进行回滚！但是这样很麻烦。

另一种是声明式事务：我们可以用xml进行配置也可以用注解方式进行配置！

Spring进行声明式事务管理的时候，底层使用的技术是aop！（底层是不是通过动态代理为service中的方法添加try catch！？）

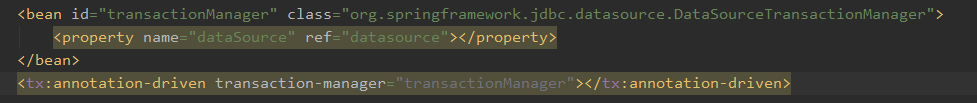
Spring中有一个事务管理器，不同的数据库操作方式会有不同的事务管理器。我们的mybatis以及jdbctemplate就是采用的DatasourceTransactionManager进行管理！



## 三十一 声明式事务管理的注解方式

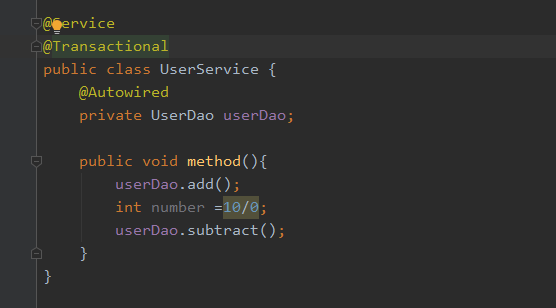
首先在配置文件中向ioc容器中注入事务管理器！

开启我们的事务注解支持！

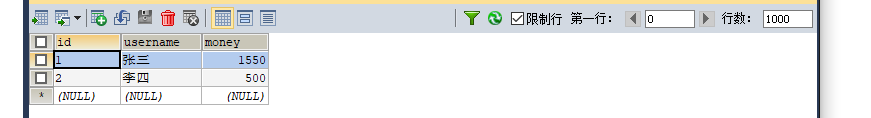


在我们的service类上标明@Transactionl注解

（@Transactionl注解也可以放在方法上）



此时再此运行，依然报错，但是数据库中数据没有发生变化！



## 三十二 声明式事务管理的注解方式（事务的传播行为）

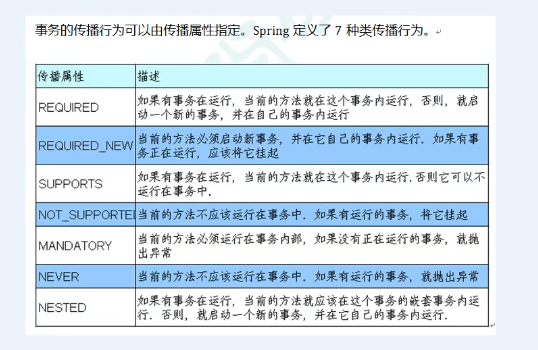
@Transactionl中可以配置很多的属性。



其中一个就是指定事务的传播行为：（propagation属性！）

当多事务进行互相调用的时候（例如：当一个被事务管理的方法调用没有被管理的方法的时候），事务是如何进行的？

如下就是propagation的所有选择，以及对应的效果，其中前两个最常用



## 三十三 声明式事务管理的注解方式（事务的隔离级别）

隔离级别是为了避免一系列问题：

脏读：一个未提交的事务读到了另一个未提交的事务（这是致命的问题！就例如张三转账给李四后又回滚。）

不可重复读：一个未提交的事务读取到了另一个已提交事务的修改数据（这个问题不致命，但是会影响员工办公，例如公司财务进行统计，具体金额时不时就会变动！）

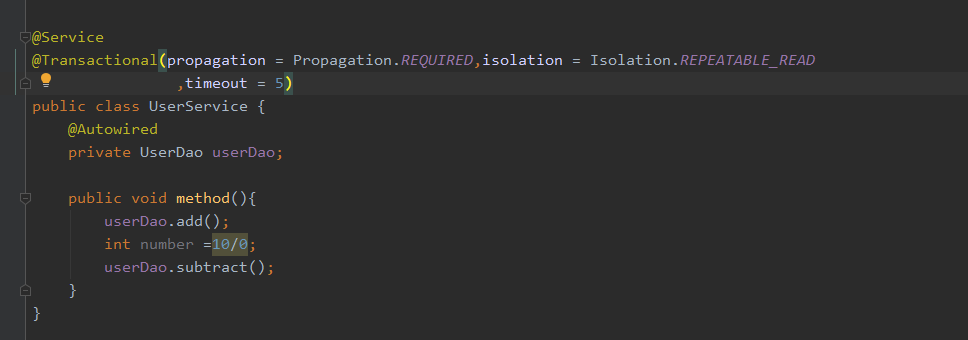
幻读：一个未提交事务读到另一个已添加的数据

我们不同的隔离级别会解决不同的问题，mysql默认是可重复读！



## 三十四 声明式事务管理的注解方式（其他参数）

其他参数有readonly（默认false），timeout（默认-1，无限等待），rollbackfor等等！

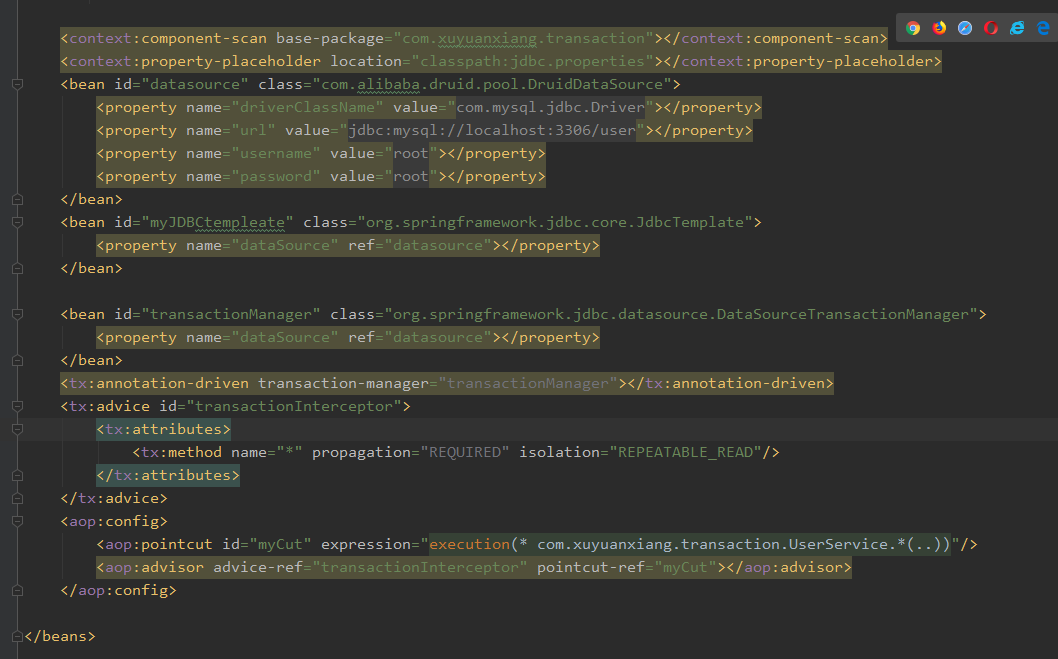


## 三十五 声明式事务管理的xml方式

这里进行全xml配置，其中的注解支持可以删除！

<tx:advice>内部配置事务属性规则

<aop:config>内部配置我们事务的切点



## 三十六 声明式事务管理（全注解方式）

其实就是写一个配置类，将我们xml文件中的各种功能实现一下（包括组件扫描，datasource注入，jdbctemplate注入，datasourcetranscationmanager注入，开启注解事务管理）

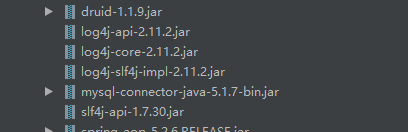


最后测试结果发现实现了相同的功能！

## 三十七 spring5整合日志

Spring5是基于java8以上的框架，运行时已经不支持log4j（移除了log4jconfiglistener）了，直接匹配log4j2。

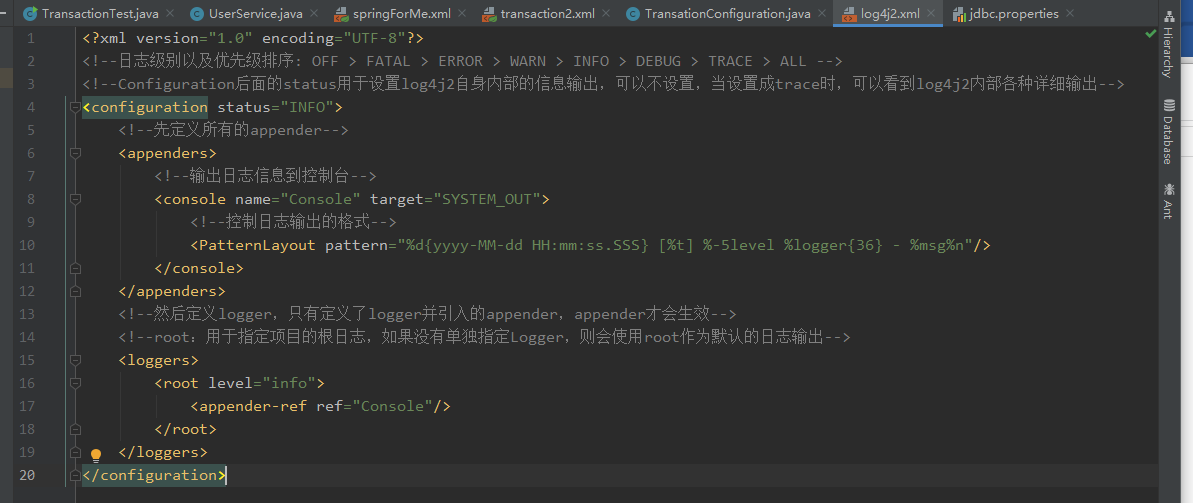
首先引入如下jar包:



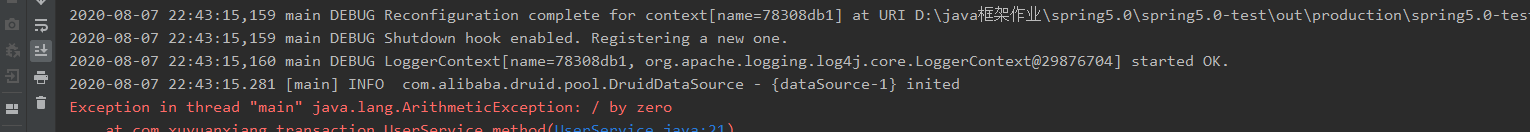
根目录下引入配置文件！



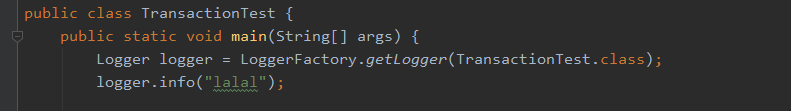
这里有配置文件的信息，我们一般直接搬用，不自己写！权限最大就是all！向下包涵



再次运行，发现多了好多信息!



当然我们可以在指定的类中获取日子类进行输出信息到控制台



## 三十八 spring5 @bullable注解

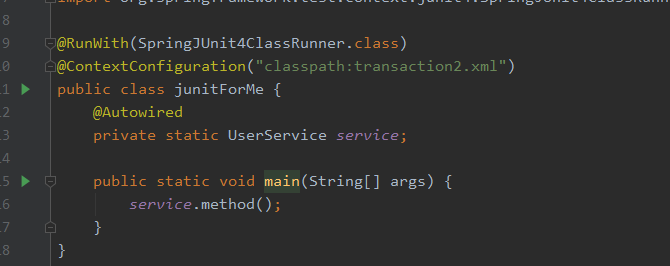
这个注解可以指定待方法，属性，参数上，分别表示方法返回值可以为空，属性可以为空，参数可以为空

## 三十九 spring5 junit5的整合

加入springtest jar包



以前我们可以如下进行test测试，比@Test方便一些



junit5就是把一些注解合并了，例如下面的注解就是上面注解的合体

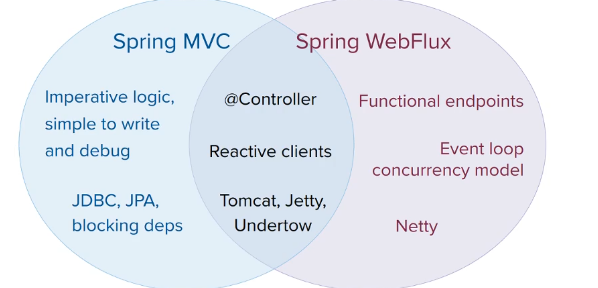


## 四十 spring5 新功能 webflux基本概念

### 一：简介

Springwebflux是spring5新添加的模块，功能类似springmvc，但是底层的原理有很大不同，它使用了当前比较流行的响应式编程。

### 二：mvc和webflux



两个框架口可以使用注解方式，都运行在tomcat等容器中。传统的web框架例如springmvc是基于servlet容器的，webflux是一种异步非阻塞的框架，这种框架在servlet3.1以后才被支持，核心是基于reactor（响应式编程）的相关api实现。

Webflux是响应式编程，mvc是命令时编程（代码一行一行执行！）

Webflux的优势：

异步非阻塞：在有限的资源下提高系统的吞吐量和伸缩性

函数时编程：spring5框架基于Java8，所以webflux中可以使用函数式编程的方式实现路由请求

Webflux和springmvc选择：

采用远程服务调度用的时候不妨采用webflux（例如微服务网关！spring gataway）

大部分项目采用springmvc就已经足够使用。

### 三：异步非阻塞

异步与同步：

发送一个请求之后，需要一直等待回应，之后才可以进行其他操作，这就是同步。

异步发送了请求之后，依然可以执行其他操作，不会一直等待！

阻塞和非阻塞

被调用者收到一个请求之后，不会立马给出回应，而是进行请求任务操作，完成后才会给出回应，这就是阻塞！

非阻塞就是收到请求后立马给出回应，之后再去执行请求

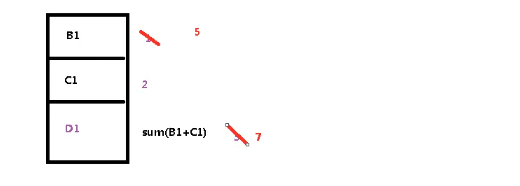
总结下来同步需要等待，异步无需等待，阻塞需要等待，非阻塞无需等待

## 四十一 spring5 新功能 webflux之响应式编程（java实现）

### 一：响应式编程：

就是面向数据流和变化传播的编程范式，可以在编程语言中很方便的表达静态或者动态的数据流，而相关计算机会自动的将变化的数据进行传播。

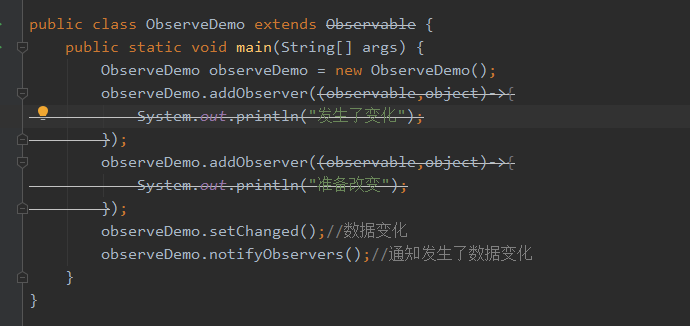
例如：我们在excel中，d1中指定一个方程，更改B1或者C1值后，结果随之变动！



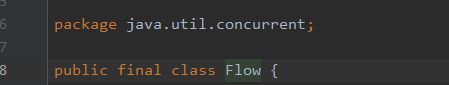
### 二：观察者模式设计思想：

Java8以及之前提供的

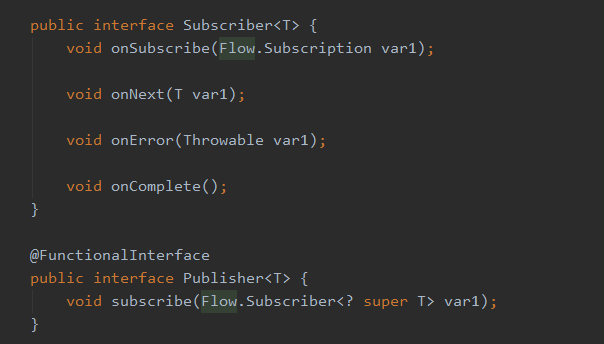
提供了观察者模式两个类：observer和observable（目前已经被废弃）



Java9以及之后提供的：这就是我们reactor底层使用的类



内有两个方法，一个通知发布，一个订阅



## 四十二 spring5 新功能 webflux之响应式编程（reactor）

### 一：Reactor介绍

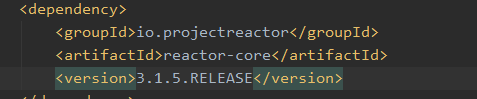
Reactor是满足reactice规范的框架

Reactor具有两个核心类，mono和flux，他们都实现了publisher接口，提供了丰富的操作符。Flux对象实现发布者，返回N个元素。

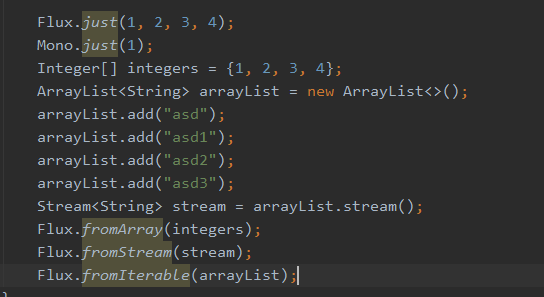
Mono对象实现发布者，返回0或者1个元素！

Flux和mono都是数据流的发布者，使用flux和mono都可以发出三种数据的信号：元素信号，错误信号，以及完成信号，后两者都代表着终止，会告诉数据流已经结束，错误信号还会返回一个错误信息给订阅者！

要想发布数据流，我们首先要导入对应的jar包：



使用方法Flux.just或者fromArray等进行发布数据流，当前没有订阅者，发布出去的数据流当然是不能够被采集到！



这里也发现mono的确不能返回多个元素的流

要想有输出，我们必须指定订阅。这里在订阅类中采用方法引用进行输出！



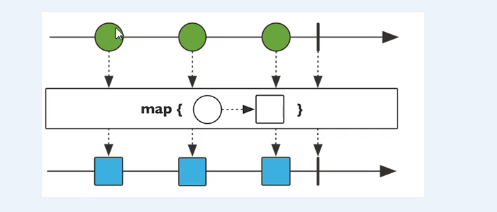
Flux和mono也可以直接发布错误信号或者完成信号。如果不发送完成信号，就代表这是一个无线流。

如果没有发送数据流就直接发送完成信号，表示这是空流！

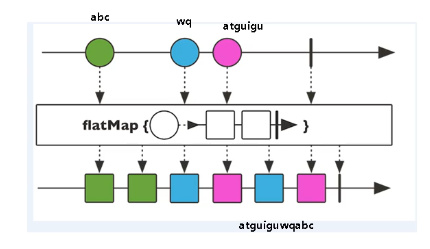
### 二：操作符：

对数据进行一次次操作，这些操作的方法就是操作符：（有非常多，这里介绍两个）

Map：将数据流中的元素映射为新的元素（就比如传进来一个1，我们对其进行+1，返回。这样元素变成了新的元素）



Flatmap：将元素映射为流（实际就是将我们流中的一个个元素都变成一个个流，然后合并成一个大的流（没有严格的顺序）返回（里面只有一个元素了？））

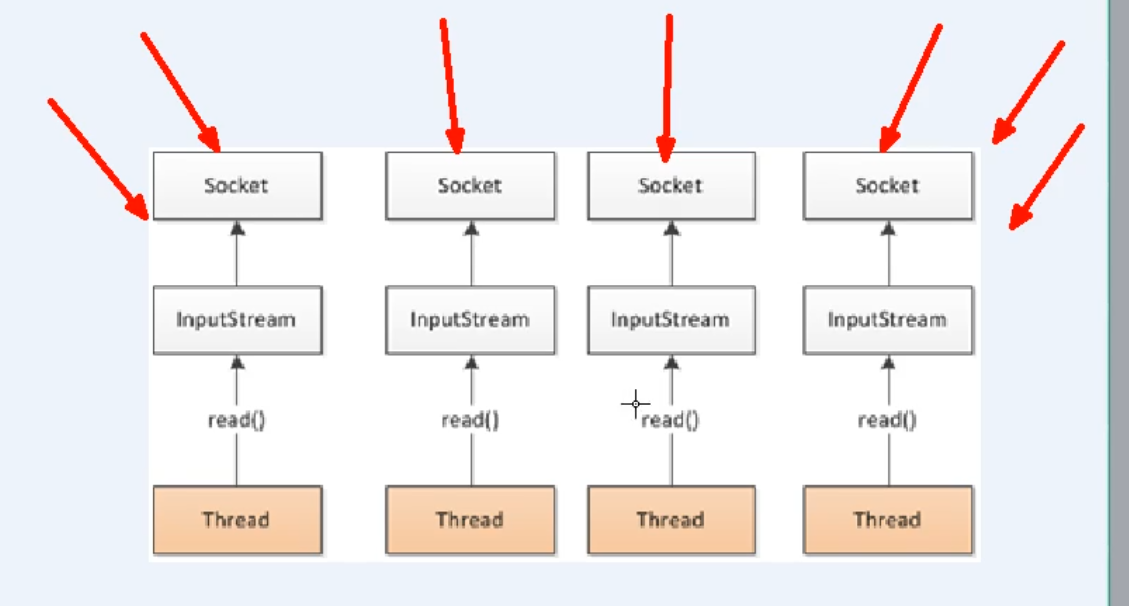


## 四十三 spring5 新功能 webflux执行流程以及核心api（重要！）

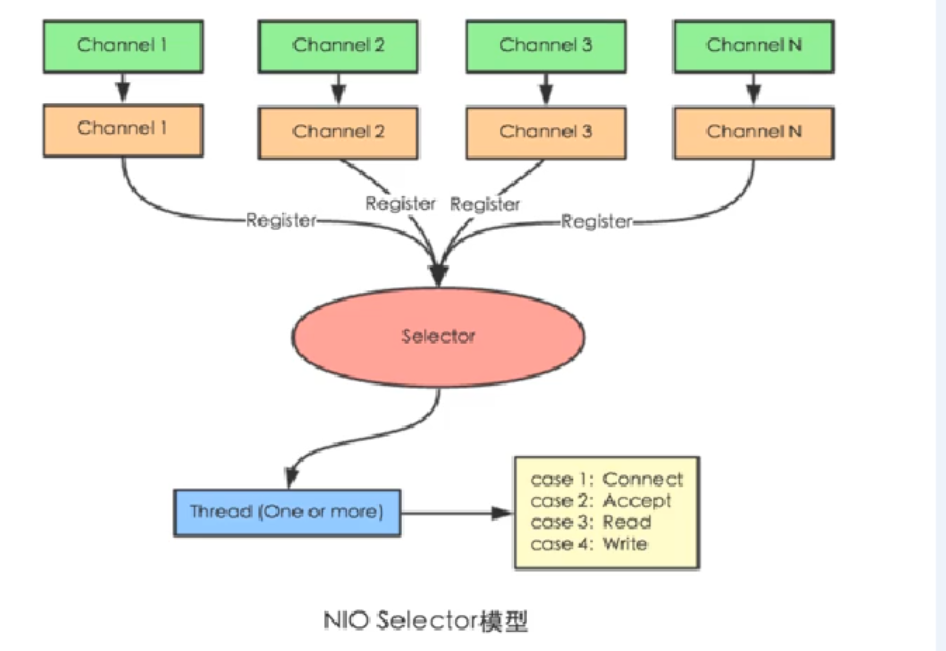
### 一：执行原理

Springwebflux基于reactor，默认使用的容器是Netty，netty是一种高性能的NIO框架，异步非阻塞框架！

BIO原理：每个流程占用一个线程！

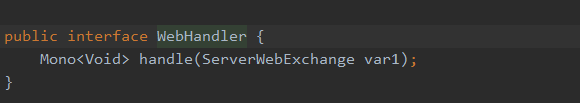


NIO原理：多个流程进入到select，被调配执行



Webflux执行原理：

有一个总接口webhander，里面有handle方法，返回一个mono



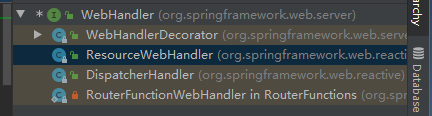
Webhandle具有如下实现类：

在dispatcherHandler中实现了wenhandler中的抽象方法：

里面的webhandlerDecorator主要负责修饰（使用了修饰者模式）

ResourceWebHandler主要负责静态资源的解析

RouterFunctionWebHandler主要负责路由！



实现类中的handler方法中的参数exchange表示就是http的那个请求

大概流程就是先判断handlermappings（这个会根据网页http请求寻找对应的方法）是不是空，是空返回错误，否则进行流处理，在流处理中又进行了两步操作符！



下面介绍几个主要的组件：

HandlerMaping :主要是将http请求匹配到指定的方法中去

HandlerAdapter :类似适配器，真正的去请求方法来处理业务

HandlerResultHandler :负责相应结果处理

### 二：reactor相关api

两个重要的接口：

RouterHandler：路由请求匹配（匹配http请求和对应的方法（即handler））

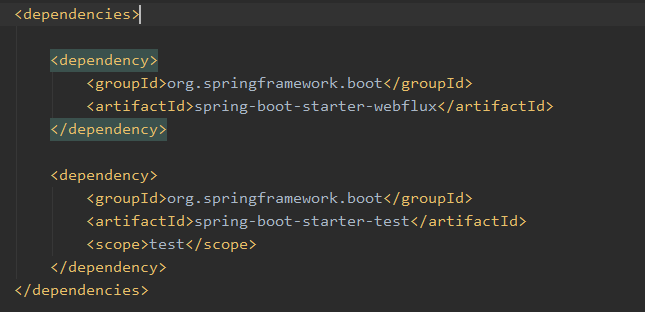
HandlerFunction：真正去执行方法（执行匹配的方法）

## 四十四 spring5 新功能 webflux注解式编程（重要！）

Springwebflux实现方式有两种，一种是注解式，另一种式函数式编程模式。

注解式与之前的mvc非常相似，只需要导入相关的jar包，我们的springboot就会自动配置相关的运行容器，默认的容器是netty

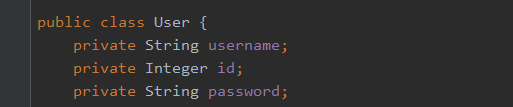
创建一个springboot项目，导入我们需要的jar包：



编写application.properties，修改端口号



创建user实体类，用来封装我们的数据！



创建service接口，注意方法的返回值和参数已经改变了



编写service实现类，这里引入map集合用作封装数据（忽略了持久层）！

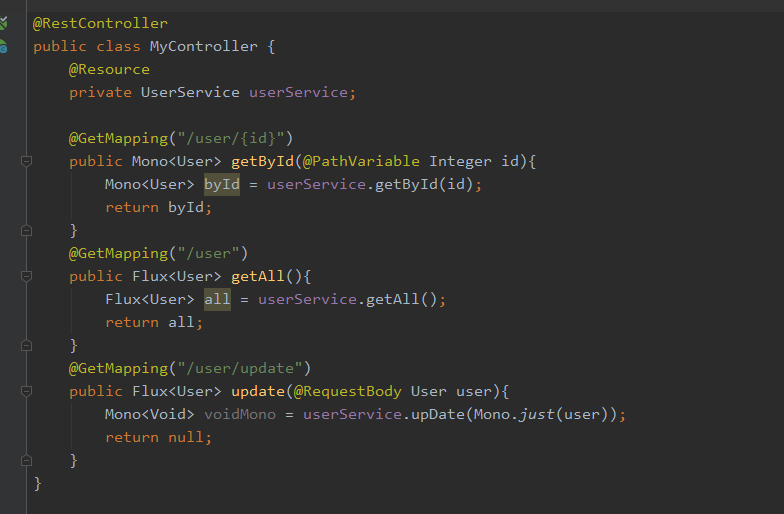
方法的参数改编成了Mono类以及flux类，用数据流来代替原来的实体类集合！

想要传入数据流就采用mono.just方法，然后将mono返回！

想要获取数据流的参数就采用Mono.doOnNext（）参数是一个consume函数式接口，这里写入了lambda表达式对流中的数据进行添加到map集合中。最后再调用thenEmpty将流终结，否则会变成无线流！

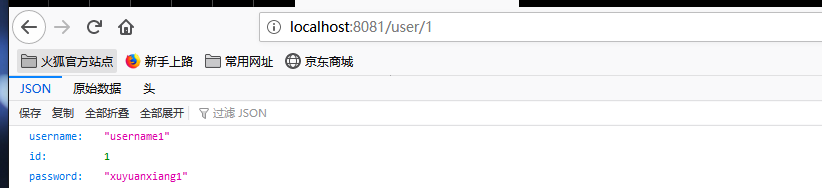


最后编写我们的controller层，与以前的mvc几乎一样，只是更改了方法的参数以及返回值类型



启动主启动类：项目启动的时候springboot会自动根据我们导入的jar包进行配置，默认使用netty为服务器！

项目测试成功



最后两种方式对比：

Mvc：同步阻塞 基于springmvc servlet Tomcat

Webflux：异步非阻塞 基于springwebflux reactor Netty

## 四十五 spring5 新功能 webflux函数式编程（重要！）

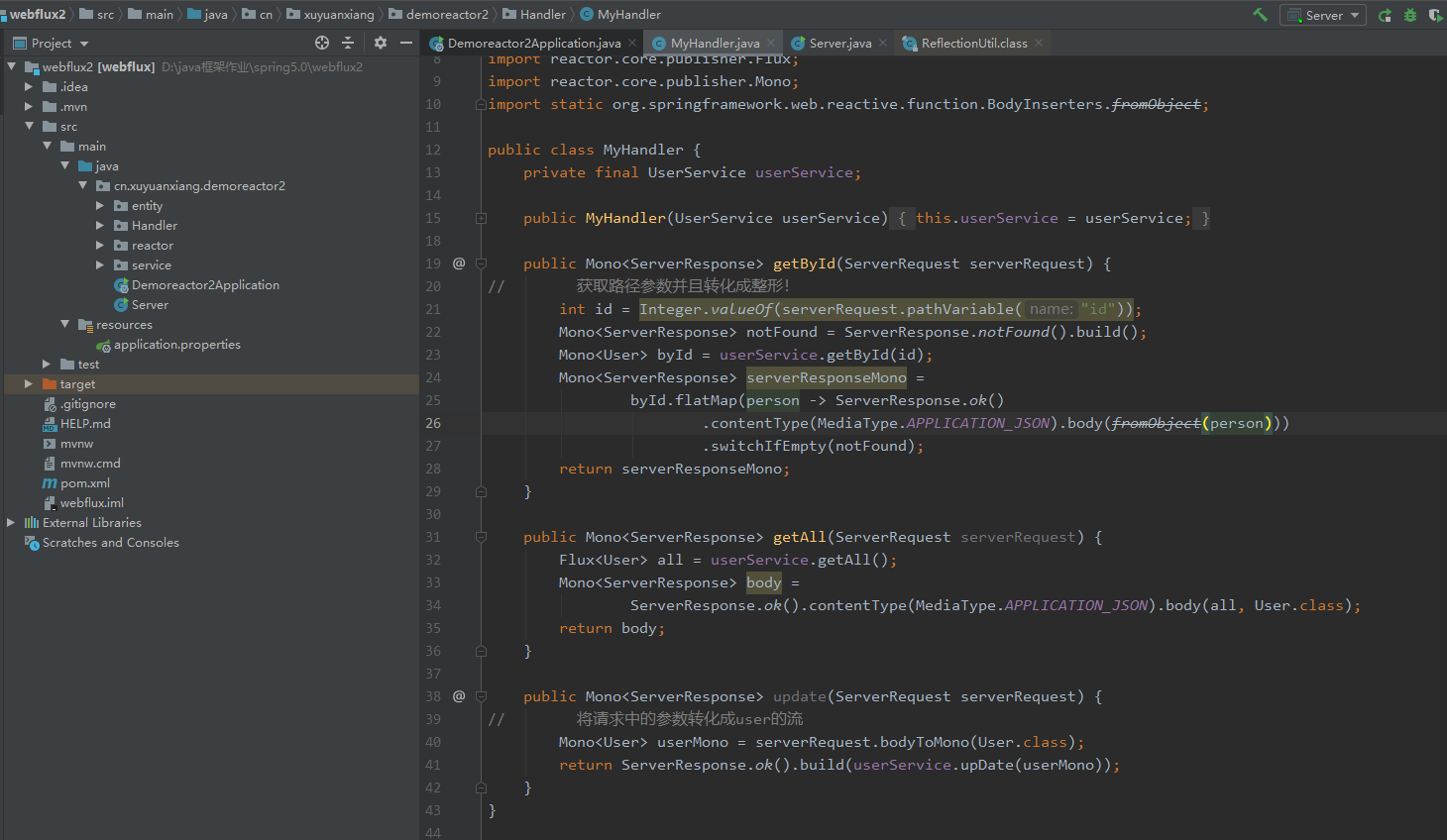
使用webflux函数式编程模型的时候，需要自己初始化服务器！

基于函数时编程模型的时候，需要使用两个核心接口：（routerFunction，handlerFunction）。

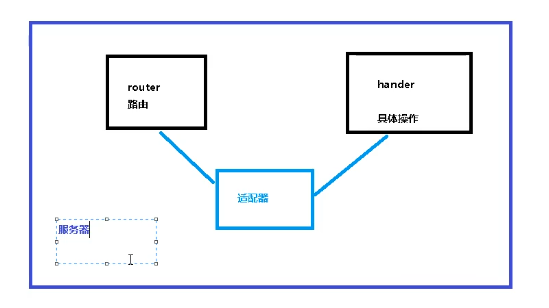
我们的任务就定义两个接口的实现，并启动需要的服务器！

Springwebflux的请求和响应不再是servletrequest和servletresponse，而是serverrequest和serverrespone。

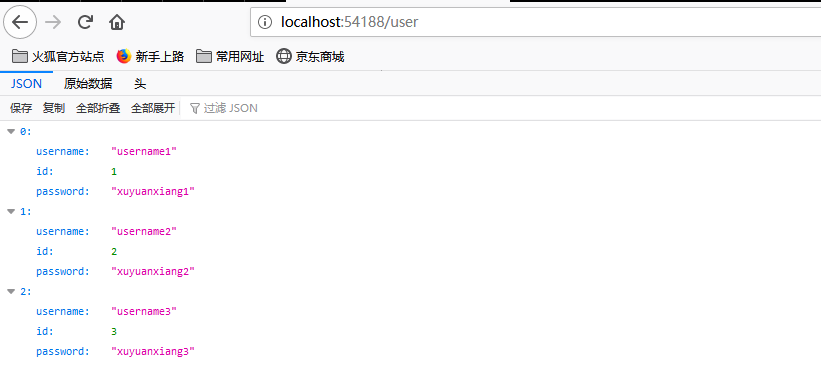
过程复杂：暂时不深究!











## 四十六 spring5 新功能 webflux函数式编程之webclient（重要！）

我们也可以通过java类去访问相应的url，例如下面通过创建webclient来进行网址的调用！

详细的方法用法以后再了解八！





